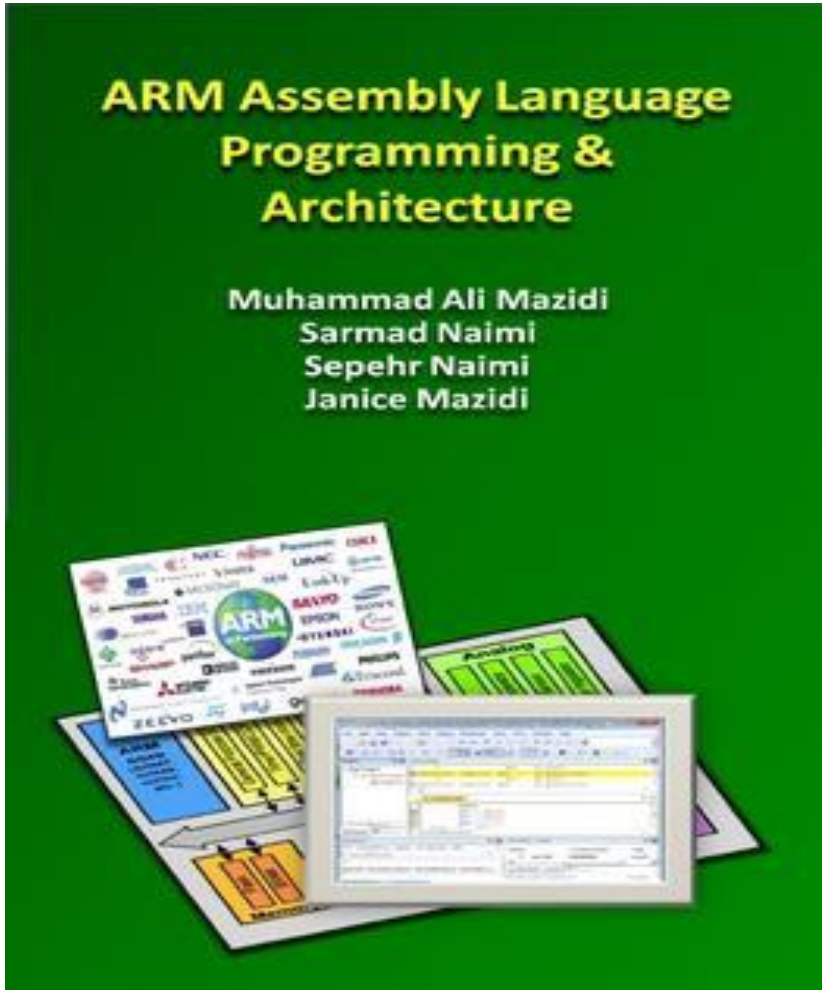
MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

ARM Assembly Language Programming & Architecture



ARM Assembly Language Programming & Architecture by Mazidi, et al.

Embedded Systems (CSE_2253)

Text Books:

1. Muhammad Ali Mazidi, Sarmad Naimi, Sepehr Naimi, Shujen Chen, *ARM Assembly Language Programming & Architecture* (2e), MicroDigitalEd, 2016
2. Jonathan W. Valvano., *Embedded systems: Introduction to Arm(r) Cortex-M Microcontrollers (5e),* Createspace Independent publishing platform, June 2014.
3. Jonathan W. Valvano., *Embedded systems: real-time interfacing to ARM Cortex-M microcontrollers (4e),* Createspace Independent Publishing Platform, 2017.

Reference Books:

1. UM10360, LPC 176x/5x User Manual, NXP Semiconductors, Rev. 4.1, 2016.
2. Toulson and Tim Wilmshurst., *Fast and Effective Embedded System Design applying the ARM mbed*, Elsevier, 2017.
3. Joseph V., *A definitive Guide to ARM Cortex-M3 and Cortex-M4 processors (3e),* Elsevier, 2014.

Embedded Systems
(CSE_2253)

Module -1

Faculty:
Dr. Shilpa Ankalaki
Assistant Professor
Department of CSE
Manipal Institute of Technology Bengaluru

Module 1

Module -1	**Teaching Hours**
INTRODUCTION TO EMBEDDED SYSTEMS AND ARM CORTEX-M MICROCONTROLLER: Introduction to Embedded Systems, Microprocessors and Microcontrollers, An overview of ARM-Cortex- M Architecture: General purpose registers, ARM memory map, Load store instructions in ARM, ARM CPSR, ARM Data format, Pseudo instructions and Directives, Introduction to ARM Assembly Programming, Some ARM Addressing modes, Advanced Indexed Addressing Mode, ADR, LDR and PC Relative addressing, RISC Architecture in ARM **Text 1: Ch 1: 1.1, Ch 2: 2.1-2.10, Ch 6:6.1, 6.2, 6.5** **Text 2: Ch 2: 2.1**	**07 Hours**

Embedded Systems

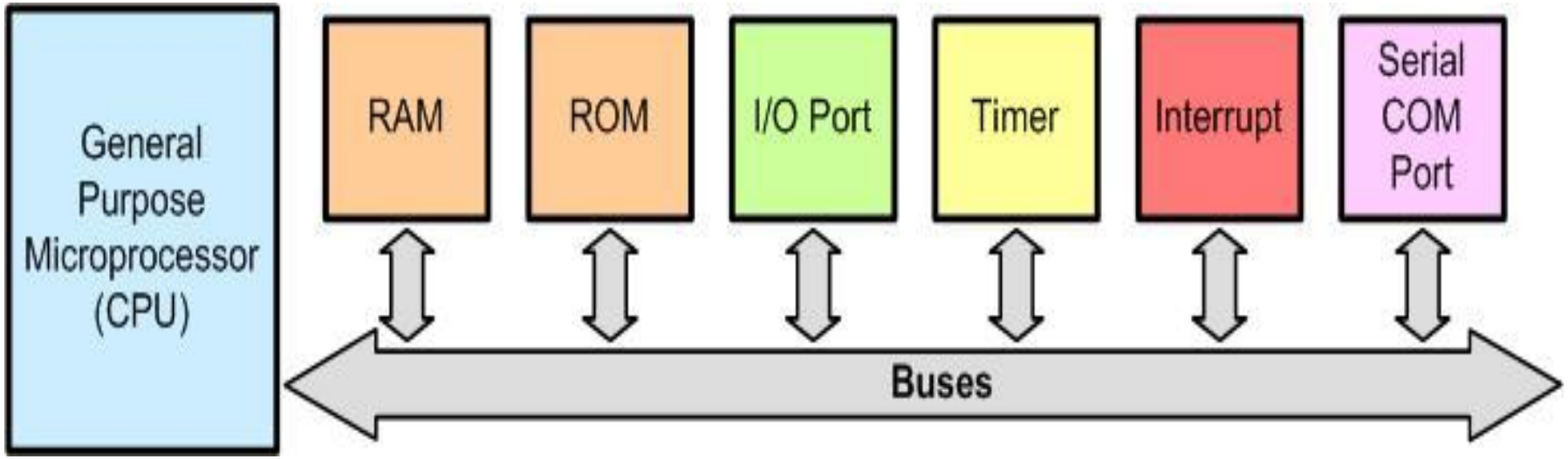
- Embedded microcomputer system:
- **"Embedded" ->** hidden inside so one can't see it.
- **"micro"** -> small
- **"computer"** -> contains a processor, memory, and a means to exchange data with the external world.
- **"system"** -> means multiple components interfaced together for a common purpose.

Another name for embedded systems is **Cyber-Physical Systems**, introduced in 2006 by Helen Gill of the National Science Foundation, because these systems combine the intelligence of a computer with the physical objects of our world.

In an embedded system, we use **ROM for storing the software and fixed constant data and RAM for storing temporary information**. Many microcomputers employed in embedded systems use Flash EEPROM, which is an electrically erasable programmable ROM, because the information can easily be erased and reprogrammed.

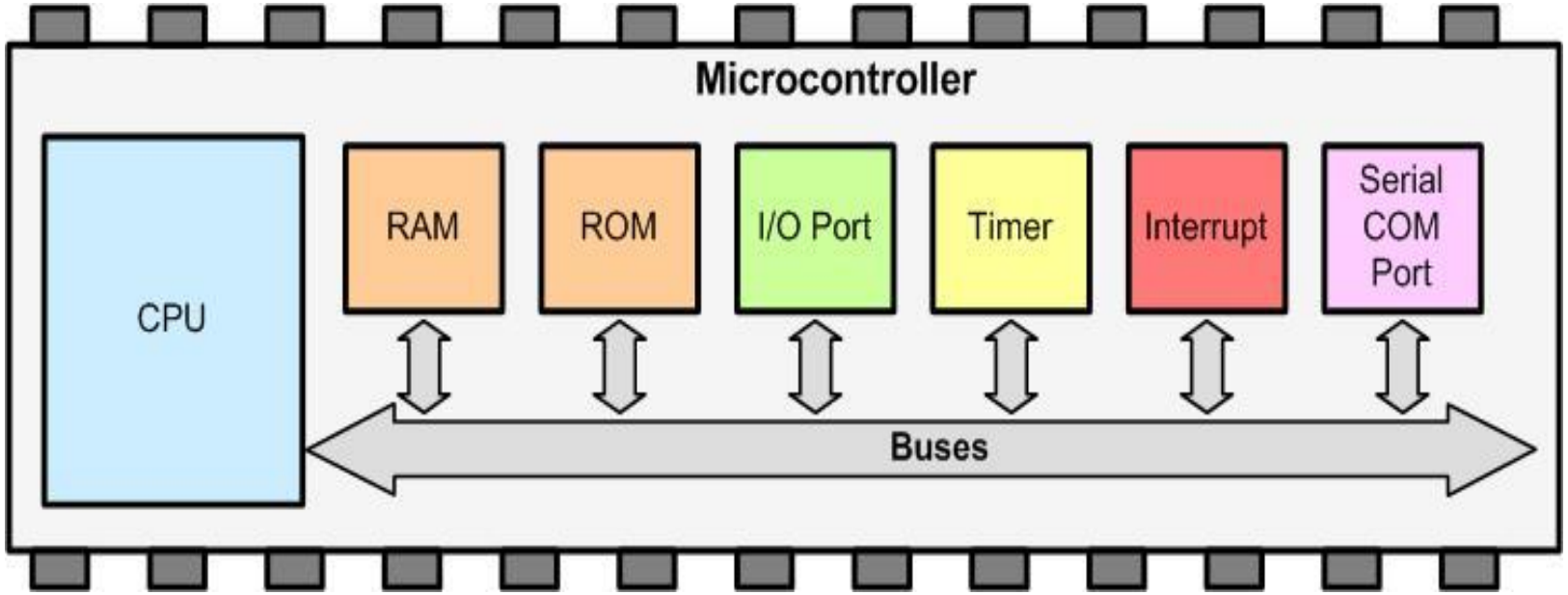
Microprocessors and Microcontrollers

MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)



A Computer Made by General Purpose Microprocessor

The microprocessors do not contain RAM, ROM, or I/O peripherals. As a result, they must be connected externally to RAM, ROM and I/O through buses



In microcontroller, CPU, RAM, ROM, and I/Os, are put together on a single IC chip and it is called *microcontroller*. SOC (System on Chip) and MCU (Micro Controller Unit) are other names used to refer to microcontrollers.

- Embedded to refer to systems that do not look and behave like a typical computer. Most embedded systems do not have a keyboard, a graphics display, or secondary storage (disk).
- There are two ways to develop embedded systems. The first technique uses a microcontroller, like the ARM Cortex M series.
- In general, there is no operating system, so the entire software system is developed. These devices are suitable for low-cost, low-performance systems.
- On the other hand, one can develop a high-performance embedded system around a more powerful microcontroller such as the ARM Cortex A-series. These systems typically employ an operating system and are first designed on a development platform, and then the software and hardware are migrated to a stand-alone embedded platform.

A Brief History of the Microcontrollers

- **1980s and 1990s**, Intel and Motorola dominated the field of microprocessors and microcontrollers.
- **In the late 1990s, the ARM microcontroller started to challenge the dominance of Intel and Motorola in the 32-bit market.**
- **32-bit microcontrollers**
  - **x86**
  - **PIC32**
  - **ColdFire**
  - **PowerPC**

Two factors can be important in choosing a microcontroller:

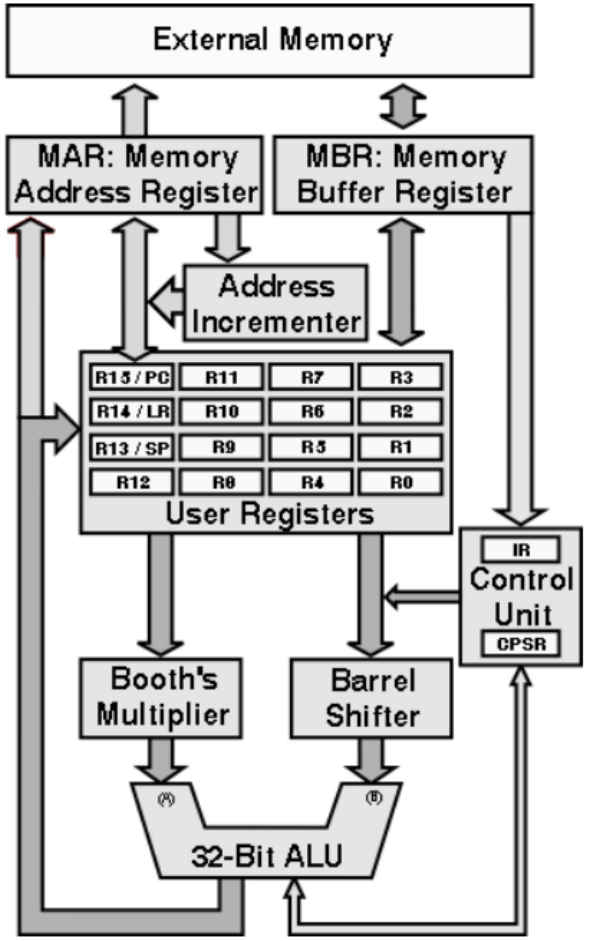
- **Chip characteristics:** Some of the factors in choosing a microcontroller chip are clock speed, power consumption, price, and on-chip memories and peripherals.
- **Available resources:** Other factors in choosing a microcontroller include the IDE compiler, legacy software, and multiple sources of production.

A brief history of the ARM

- The ARM came out of a company called **Acorn Computers** in United Kingdom in the 1980s.
- Professor **Steve Furber of Manchester University worked with Sophie Wilson** to define the ARM architecture and instructions.
- The VLSI Technology Corp. produced the first ARM chip in 1985 for Acorn Computers and was designated as Acorn RISC Machine (ARM).
- Apple Corp. got interested in using the ARM chip for the PDA (personal digital assistants) products.
- This renewed interest in the chip led to the creation of a new company called ARM (Advanced RISC Machine).
- This new company bet its entire fortune on selling the rights to this new CPU to other silicon manufacturers and design houses. Since the early 1990s, an ever increasing number of companies have licensed the right to make the ARM chip.

ARM - One CPU, many peripherals

- ARM has defined the details of architecture, registers, instruction set, memory map, and timing of the ARM CPU and holds the copyright to it.
- The various design houses and semiconductor manufacturers license the IP (intellectual property) for the CPU and can add their own peripherals as they please.
- It is up to the licensee (design houses and semiconductor manufactures) to define the details of peripherals such as I/O ports, serial port UART, timer, ADC, SPI, DAC, I2C, and so on.
- As a result while the CPU instructions and architecture are same across all the ARM chips made by different vendors, their peripherals are not compatible. This is the only drawback of ARM
- The good news is the IDE (integrated development environment) such as Keil (see www.keil.com) or IAR (see www.IAR.com) do provide peripheral libraries for chips from various vendors and make the job of programming the peripherals much easier.
- In recent years ARM provides the IP for some peripherals such as UART and SPI, but unlike the CPU architecture, its adoption is not mandatory and it is up to the chip manufacturer whether to adopt it or not.


External Memory
MAR: Memory Address Register
MBR: Memory Buffer Register
Address Incrementer
R15 / PC
R11
R7
R3
R14 / LR
R10
R6
R2
R13 / SP
R9
R5
R1
R12
R8
R4
R0
User Registers
IR
Control Unit
CPSR
Booth's Multiplier
Barrel Shifter
(A)
(B)
32-Bit ALU

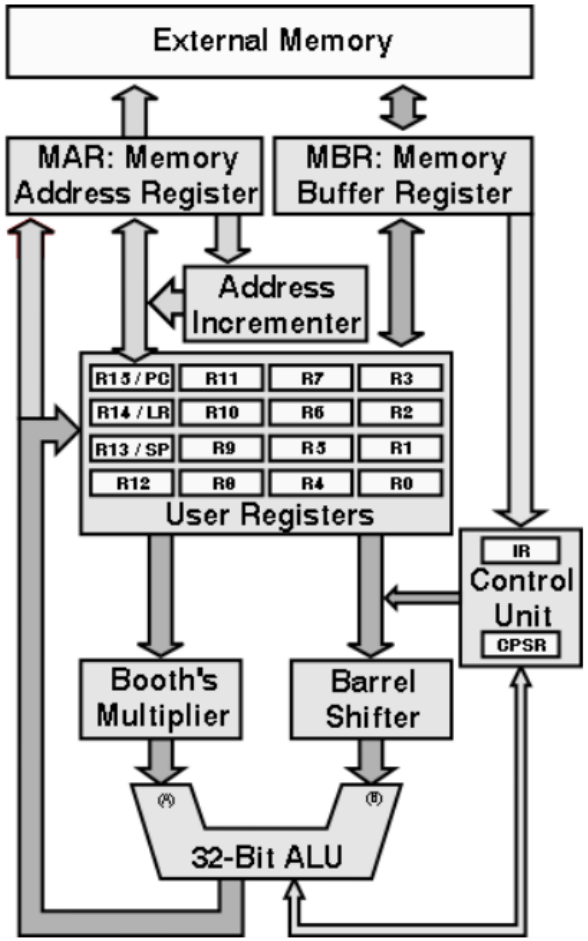
Arm Architecture

The ARM is a Reduced Instruction Set Computer (RISC) system and includes the attributes typical to that type of system:

- A large array of uniform registers.
- A load/store model of data-processing where operations can only operate on registers and not directly on memory. This requires that all data be loaded into registers before an operation can be preformed, the result can then be used for further processing or stored back into memory.
- A small number of addressing modes with all load/store addresses begin determined from registers and instruction fields only.
- A uniform fixed length instruction (32-bit).

In addition to these traditional features of a RISC system the ARM provides a number of additional features:

- Separate Arithmetic Logic Unit (ALU) and shifter giving additional control over data processing to maximize execution speed.
- Auto-increment and Auto-decrement addressing modes to improve the operation of program loops.
- Conditional execution of instructions to reduce pipeline flushing and thus increase execution speed.



ARM CORE DATA FLOW MODEL

MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

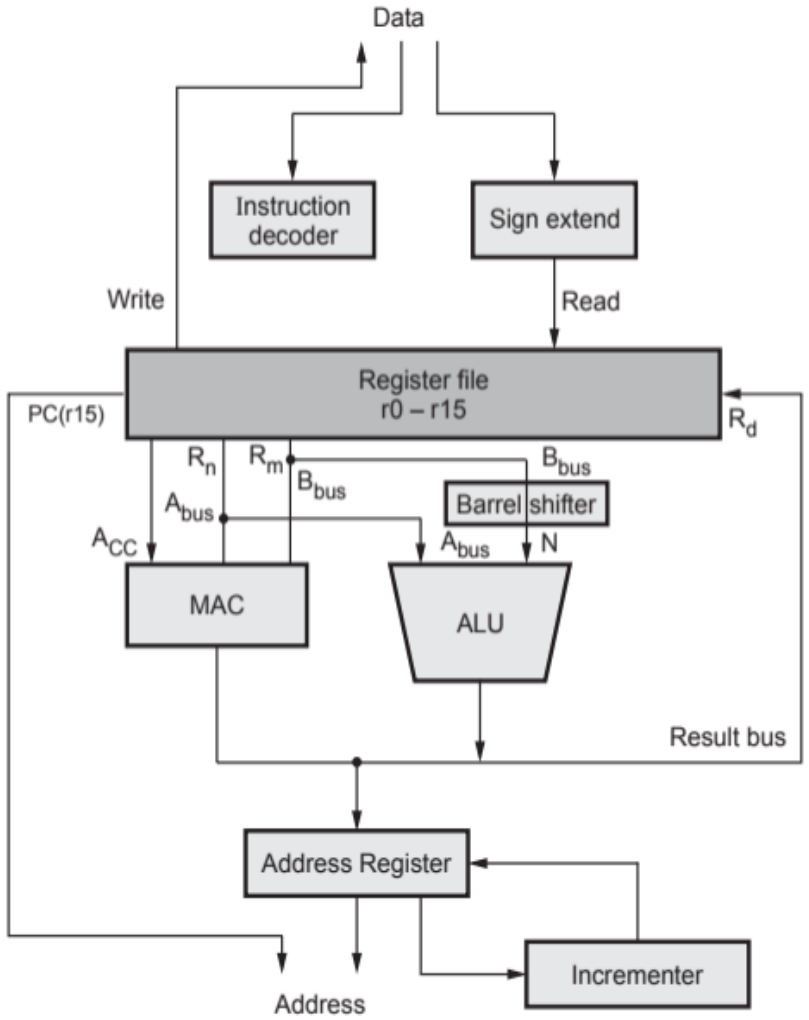


Fig. 2.1.1 ARM core dataflow model

Fig. shows the basic structure of ARM core and how data moves between its different parts.

An ARM core dataflow model may be thought of as functional units connected by data buses, with the arrows representing data flow, the lines representing buses, and the boxes representing either an operating unit or a storage region. The diagram depicts both the data flow and the abstract components that make up an ARM core. The Data bus is where data enters the CPU core. The data might be an executable command or a data object.

The load-store architecture is used by the ARM processor, as it is by all RISC processors. This implies there are two sorts of instructions for moving data in and out of the processor: load instructions copy data from memory to core registers, while store instructions copy data from registers to memory. Data processing instructions that directly manipulate data in memory are not available. As a result, data processing is limited to registers. The register file—a storage bank made up of 32-bit registers—is where data objects are stored. Most instructions consider the registers as having signed or unsigned 32-bit values since the ARM core is a 32-bit processor.

- When signed 8-bit and 16-bit integers are read from memory and stored in a register, the sign extend hardware transforms them to 32-bit values. Rn and Rm are the two source registers of ARM instructions, while Rd is the result or destination register. The internal buses A and B are used to read source operands from the register file.
- The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) computes a result using the Rn and Rm register values from the A and B buses. The outcome of data processing instructions is written directly to the register file in Rd. The ALU is used in load and store instructions to create an address that is stored in the address register and broadcast over the Address bus.
- Because the ARM7 architecture is based on the von Neuman architecture, the same bus is used to load both instructions and data.

- One important feature of the ARM is that register Rm alternatively can be preprocessed in the barrel shifter before it enters the ALU.
- Together the barrel shifter and ALU can calculate a wide range of expressions and addresses. After passing through the functional units, the result in Rd is written back to the register file using the Result bus.
- For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location.
- The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

Barrel shifter

A barrel shifter is a digital circuit capable of shifting a data word by a specific number of bits in a single clock cycle. This hardware unit can execute data shifting operations on the data in the register bank.

MAC (Multiply and accumulate unit)

Basic summation operations on data in registers are supported.

Any operation's result can be written back to the register bank, or if the instruction needs memory access, the result is sent to the address register.

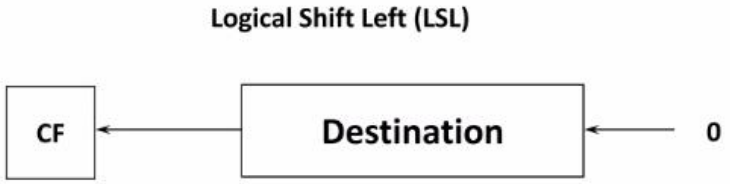
Address register:

This contains the address from which data or instruction needs to be fetched.

This register is connected to an incrementing unit.

- The ARM doesn't have actual shift instructions.
- Instead it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.
- Shifts left by the specified amount (multiplies by powers of two) e.g.

  LSL #5 => multiply by 32

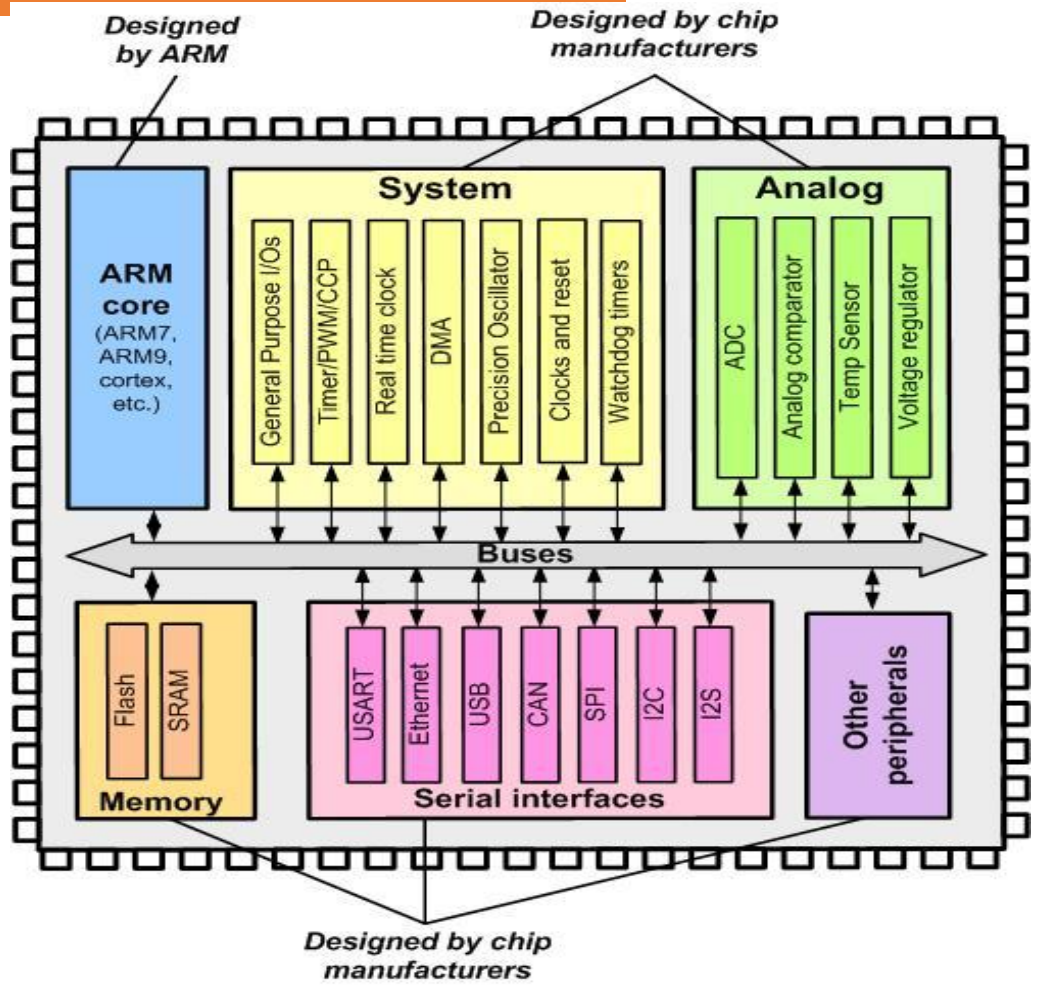



```
MOV R0, #3
MOV R1, #7
ADD R2, R0, R1, LSL #3
```

R2= R0+R1*8

ARM Vendors

Actel	Analog Devices	Atmel
Broadcom	Cypress	Ember
Dust Networks	Energy	Freescale
Fujitso	Nuvoton	NXP
Renesas	Samsung	ST
Toshiba	Texas Instruments	Triad Semiconductor


Designed by ARM
Designed by chip manufacturers
ARM core (ARM7, ARM9, cortex, etc.)
System
General Purpose I/Os
Timer/PWM/CCP
Real time clock
DMA
Precision Oscillator
Clocks and reset
Watchdog timers
Analog
ADC
Analog comparator
Temp Sensor
Voltage regulator
Buses
Flash
SRAM
Memory
USART
Ethernet
USB
CAN
SPI
I2C
I2S
Serial interfaces
Other peripherals
Designed by chip manufacturers

Processor Modes

User Mode: The least privileged mode for user applications.

- Restricted access to certain instructions and resources.
- User applications typically run in this mode.

System Mode: Privileged mode using the same registers as user mode. Privileged Modes provide elevated privileges for system-level operations

Supervisor Mode (SVC): A privileged mode entered when handling system calls or exceptions.

- Higher privilege than User Mode.
- Access to a broader range of instructions and resources.
- Handling system calls and managing OS-related operations.

Abort Mode (ABT): Entered when a data or instruction prefetch abort occurs.

- Provides the ability to handle memory access faults.
- Handling memory access faults, such as data aborts or instruction prefetch aborts.

Undefined Mode (UND): Entered when an undefined instruction is encountered. Allows handling of undefined instruction exceptions

IRQ Mode and FIQ Mode: Older modes for handling interrupt requests (IRQ) and fast interrupt requests (FIQ). Handling general-purpose interrupts (IRQ) or high-priority interrupts (FIQ).

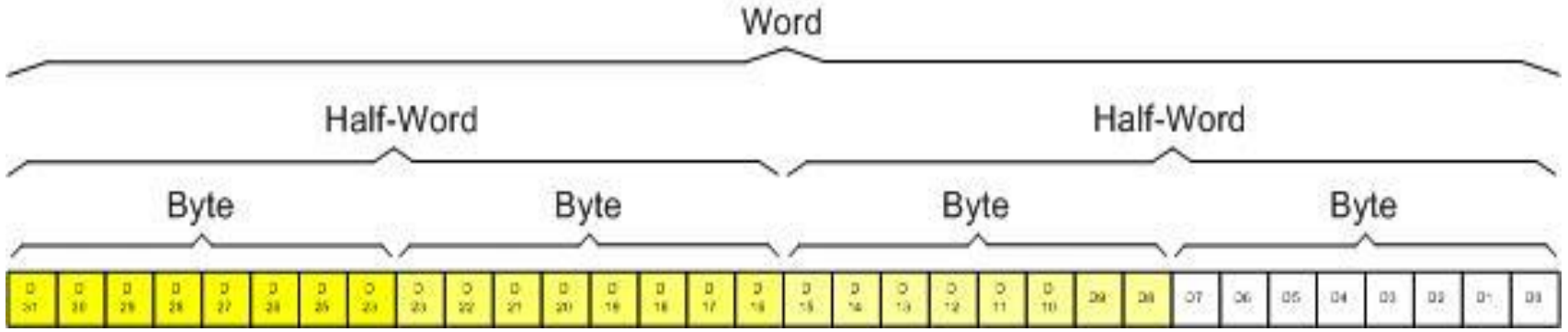
Processor mode		Description
User	usr	Normal program execution mode
FIQ	fiq	Fast Interrupt for high-speed data transfer
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks

Table 3.1: ARM processor modes

ARM Architecture- General Purpose Registers (GPR) in ARM

MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

- ARM microcontrollers have many registers for arithmetic and logic operations. In the CPU, registers are used to store information temporarily. That information could be a byte of data to be processed, or an address pointing to the data to be fetched.
- All of ARM registers are 32-bit wide. The 32 bits of a register are shown in Figure 2-1.
- With a 32- bit data type, any data larger than 32 bits must be broken into 32-bit chunks before it is processed.
- Although the ARM default data size is 32-bit many assemblers also support the single bit, 8-bit, and 16-bit data types
- In ARM the 16-bit data is referred to as **half-word.** Therefore ARM supports **byte, half-word (two byte), and word (four bytes) data types.**



General Purpose Registers

- In ARM there are 13 general purpose registers. They are R0–R12. All of these registers are 32 bits wide.
- The general purpose registers in ARM are the same as the accumulator in other microprocessors. They can be used by all arithmetic and logic instructions.
- The ARM core has three special function registers of R13, R14, and R15

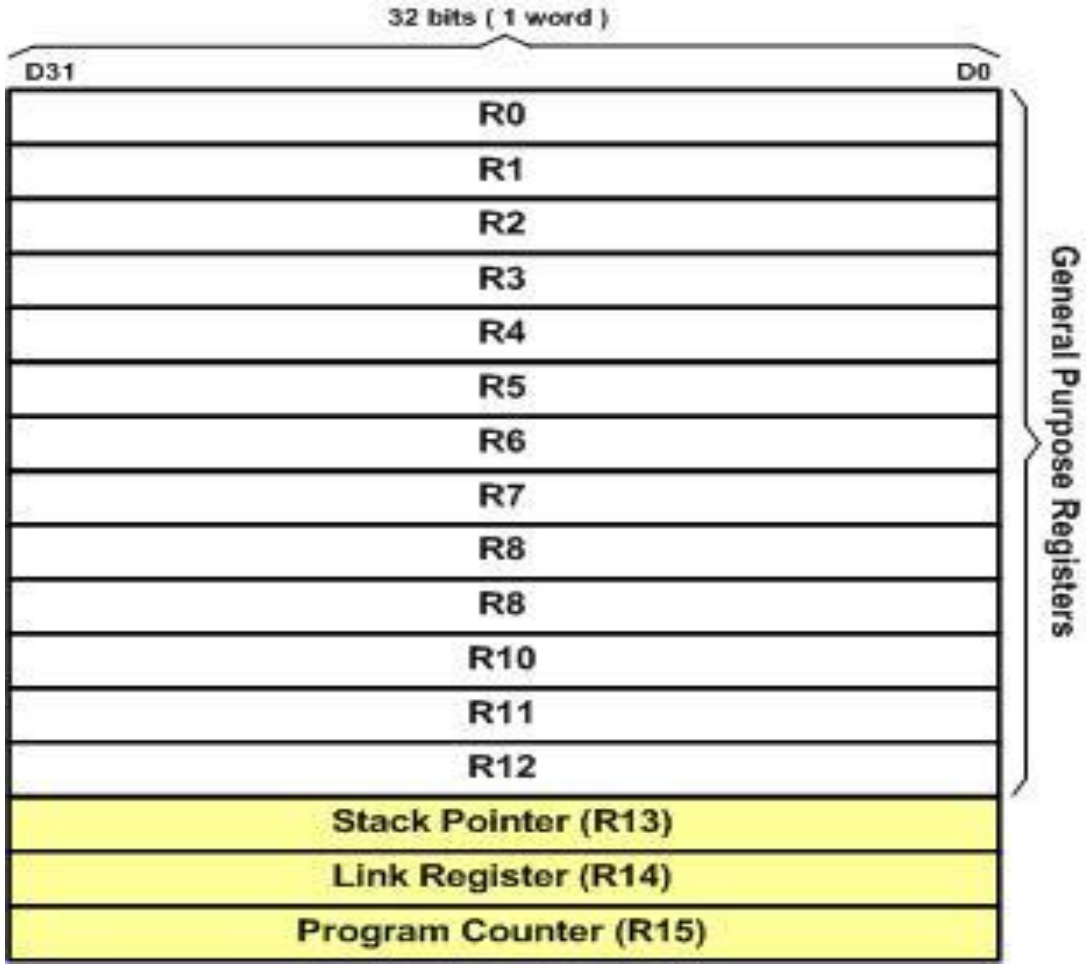


Figure 2- 2: ARM Registers

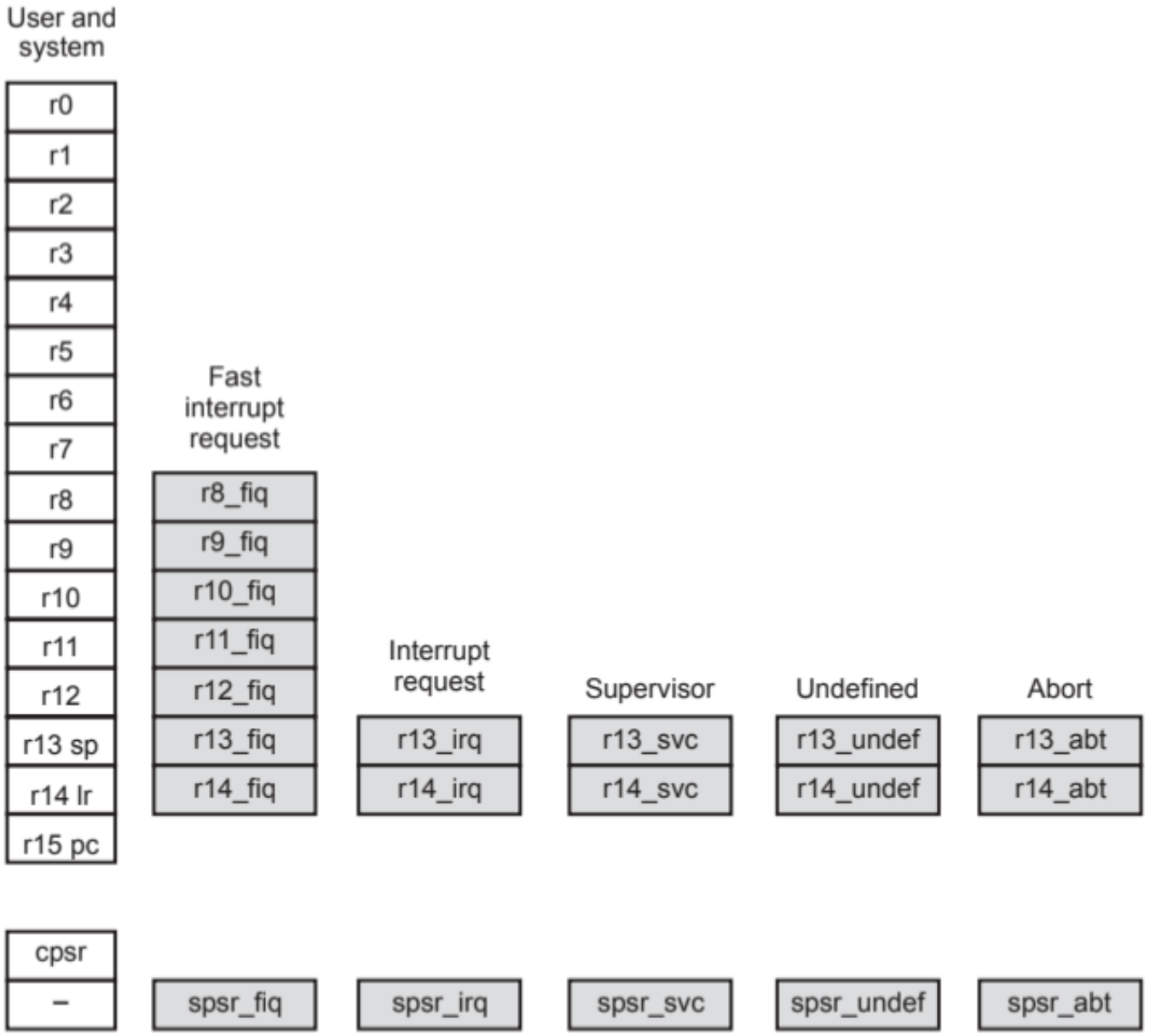


Fig. 2.2.1 Programming model of ARM processor

ARM Registers

- All ARM instructions are four bytes long (one 32-bit word) and are always aligned on a word boundary. This means that the bottom two bits of the PC are always zero, and therefore the PC contains only 30 non-constant bits.

The Unbanked Registers r0-r7

- Registers r0 to r7 are unbanked registers. This means that each of them refers to the same 32-bit physical register in all processor modes.
- They are completely general-purpose registers, with no special uses implied by the architecture, and can be used wherever an instruction allows a general-purpose register to be specified.

ARM Registers

Banked Registers, r8 - r14

- Registers r8 to r14 are banked registers.
- The physical register referred to by each of them depends on the current processor mode. Where a particular physical register is intended, without depending on the current processor mode, a more specific name is used. Almost all instructions allow the banked registers to be used wherever a general-purpose register is allowed.
- Out of 37 registers, 20 registers which are shown shaded in Fig. 2.2.1 are the banked registers. Fig. 2.2.1 also shows which banked registers are used in which mode. Banked registers of a particular mode are denoted by, r number_mode.
- For example, **supervisor, mode has banked registers r13_svc, r14_svc and spsr_svc**.
- On the other hand, abort mode has banked registers r13-abt, r14-abt and spsr-abt.
- Registers r8 to r12 have two banked physical registers each. The first group of physical registers are referred to as r8_usr to r12_usr and the second group as r8_fiq to r12_fiq. The r8_usr to r12_usr group is used in all processor modes other e than FIQ mode, and the other is used in FIQ mode.
- Registers r13 and r14 have six banked physical registers each. One is used in User and System modes, while each of the remaining five is used in one of the five exception modes.

ARM Instruction Format

- The ARM CPU uses the tri-part instruction format for most instructions. One of the most common format is

instruction destination,source1,source2

Depending on the instruction the source2 can be a register, immediate (constant) value, or memory. The destination is often a register or read/write memory.

MOV instruction

Simply stated, the MOV instruction copies data into register or from register to register. It has the following formats:

MOV Rn,Op2 ; load Rn register with Op2 (Operand2). ;Op2 can be immediate

MOV R2,#0x25 ;load R2 with 0x25 (R2 = 0x25)

MOV R5,R7 ;copy contents of R7 into R5 (R5 = R7)

To write a comment in Assembly language we use ';'.

Instruction	Description
ADD Rd, Rn,Op2*	ADD Rn to Op2 and place the result in Rd
ADC Rd, Rn,Op2	ADD Rn to Op2 with Carry and place the result in Rd
AND Rd, Rn,Op2	AND Rn with Op2 and place the result in Rd
BIC Rd, Rn,Op2	AND Rn with NOT of Op2 and place the result in Rd
CMP Rn,Op2	Compare Rn with Op2 and set the status bits of CPSR**
CMN Rn,Op2	Compare Rn with negative of Op2 and set the status bits
EOR Rd, Rn,Op2	Exclusive OR Rn with Op2 and place the result in Rd
MVN Rd,Op2	Place NOT of Op2 in Rd
MOV Rd,Op2	MOVE (Copy) Op2 to Rd
ORR Rd, Rn,Op2	OR Rn with Op2 and place the result in Rd
RSB Rd, Rn,Op2	Subtract Rn from Op2 and place the result in Rd
RSC Rd, Rn,Op2	Subtract Rn from Op2 with carry and place the result in Rd
SBC Rd, Rn,Op2	Subtract Op2 from Rn with carry and place the result in Rd
SUB Rd, Rn,Op2	Subtract Op2 from Rn and place the result in Rd
TEQ Rn,Op2	Exclusive-OR Rn with Op2 and set the status bits of CPSR
TST Rn,Op2	AND Rn with Op2 and set the status bits of CPSR

* ***Op2 can be an immediate 8-bit value #K which can be 0–255 in decimal, (00–FF in hex). Op2 can also be a register Rm. Rd, Rn and Rm are any of the general purpose registers***

** ***CPSR is discussed later in this chapter***

*** ***The instructions are discussed in detail in the next chapters***

Addressing Modes

Addressing without Offset

```
LDR r1, [r0]          ; r0 holds the memory address
```

OLOGY

Name	Alternative Name	ARM Examples
Register to register	Register direct	MOV R0, R1
Absolute	Direct	LDR R0, MEM
Literal	Immediate	MOV R0, #15 ADD R1, R2, #12
Indexed, base	Register indirect	LDR R0, [R1]
Pre-indexed, base with displacement	Register indirect with offset	LDR R0, [R1, #4]
Pre-indexed, autoindexing	Register indirect pre-incrementing	LDR R0, [R1, #4]!
Post-indexing, autoindexed	Register indirect post-increment	LDR R0, [R1], #4
Double Reg indirect	Register indirect Register indexed	LDR R0, [R1, R2]
Double Reg indirect with scaling	Register indirect indexed with scaling	LDR R0, [R1, R2, LSL #2]
Program counter relative		LDR R0, [PC, #offset]

State the contents of R2, R1, and memory location 0x20 after the following program:

```
MOV    R2,#0x5
MOV    R1,#0x2
ADD     R2, R1,R2
ADD     R2,R1,R2
MOV    R5,#0x20
STRB    R2,[R5]
```

Location	Data
R2	5
R1	
0x20	

Location	Data
R2	5
R1	2
0x20	

Location	Data
R2	7
R1	2
0x20	

Location	Data
R2	9
R1	2
0x20	

Location	Data
R2	9
R1	2
0x20	9

```
MOV     R2,#0x5          ;load R2 with 5  (R2 = 0x05)
MOV     R1,#0x2          ;load R1 with 2 (R1 = 0x02)
ADD      R2, R1,R2        ;R2 = R1 + R2
ADD      R2,R1,R2         ;R2 = R1 + R2
MOV     R5,#0x20         ;R5 = 0x20
STRB     R2,[R5]          ;store R2 into location pointed to by R5
```

State the contents of RAM locations 0x92 to 0x96 after the following program is executed:


```
MOV     R1,#0x99
MOV     R6,#0x92
STRB     R1,[R6]
;(location 0x92)
ADD      R6,R6,#1
MOV     R1,#0x85
STRB     R1,[R6]
;(location 0x93)
ADD      R6,R6,#1
MOV     R1,#0x3F
STRB     R1,[R6]
ADD      R6,R6,#1
MOV     R1,#0x63
STRB     R1,[R6]
ADD      R6,R6,#1
MOV     R1,#0x12
STRB     R1,[R6]
```

State the contents of RAM locations 0x92 to 0x96 after the following program is executed:

```
MOV     R1,#0x99          ;R1 = 0x99
MOV     R6,#0x92          ;R6 = 0x92
STRB     R1,[R6]           ;store R1 into location pointed to by R6
;(location 0x92)
ADD      R6,R6,#1          ;R6 = R6 + 1
MOV     R1,#0x85          ;R1 = 0x85
STRB     R1,[R6]           ;store R1 into location pointed to by R6
;(location 0x93)
ADD      R6,R6,#1          ;R6 = R6 + 1
MOV     R1,#0x3F          ;R1 = 0x3F
STRB     R1,[R6]           ;store R1 into location pointed to by R6
ADD      R6,R6,#1          ;R6 = R6 + 1
MOV     R1,#0x63          ;R1 = 0x63
STRB     R1,[R6]           ;store R1 into location pointed to by R6
ADD      R6,R6,#1          ;R6 = R6 + 1
MOV     R1,#0x12          ;R1 = 0x12
STRB     R1,[R6]
```

TUTE OF TECHNOLOGY

)al)

Address	Data
0x92	0x99
0x93	0x85
0x94	0x3F
0x95	0x63
0x96	0x12

ARM CPSR

- The flag register in the ARM is called as **current program status register** (CPSR). It is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information.
- Each exception mode also has a **saved program status register** (spsr), that is used to preserve the value of the cpsr when the associated exception occurs.
- **Note User mode and System mode do not have an SPSR, because they are not exception modes. All instructions which read or write the SPSR UNPREDICTABLE when executed in User mode or System mode.**
-

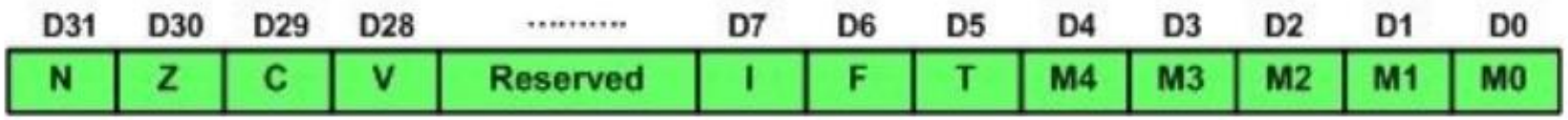


Figure 2- 11: CPSR (Current Program Status Register)

ARM CPSR

- **Control Flags (0-7):** The control bit change when exception arises and can be altered by software only when the processor in in privileged mode.
- **Bits 0-4 (Mode Select Bits): Processor modes**
- These bits determine the processor mode as shown in Table

Processor mode	Mode Select Bits [4 : 0]
Abort	1 0 1 1 1
Fast interrupt request	1 0 0 0 1
Interrupt request	1 0 0 1 0
Supervisor	1 0 0 1 1
System	1 1 1 1 1
Undefined	1 1 0 1 1
User	1 0 0 0 0

Table 2.3.1 Processor mode

Bit 5 (Thumb State Bit) :

- This bit gives the state of the core. The state of the core determines which instruction set is being executed.
- The T flag bit is used to indicate the ARM is in Thumb state. Thumb state is a mode of operation for ARM Processor that allows them to execute 16-bit instructions.

Bits 6 and 7 (Interrupt Masks) :

There are two interrupts available on the ARM processor core:

- Interrupt Request (IRQ) and
- Fast Interrupt Request (FIQ).

- These are maskable interrupts, and their masking is controlled by bits 6 and 7 of CPSR. Bit 6(F) controls FIQ, and bit 7(1) controls IRQ.
- When a bit is set to binary 1, the corresponding interrupt request is masked, and when a bit is 0, the interrupt is available.

Condition code flags (Bit 28 - 31):

- These flags in the cpsr can be tested by most instructions to determine whether the instruction is to be executed.

Bit 28 (Overflow flag, V) : It is set in one of two ways:

- For an addition or subtraction, V is set to 1 if signed overflow occurred, regarding the operands and result as two's complement signed integers.
- "For non addition/subtractions V is normally left unchanged.

Bit 29 (Carry flag, C): It is set in one of four ways:

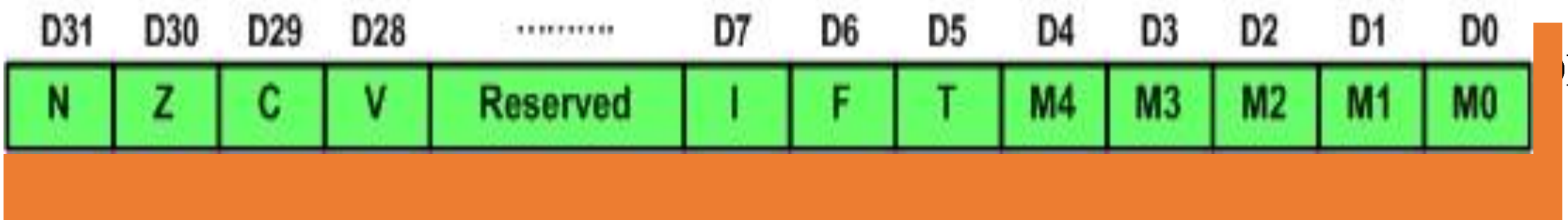
- For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-addition/subtractions, C is normally left unchanged.

Bit 30 (Zero flag, Z)

• It is set to 1 if the result of the instruction is zero (which often indicates an equal result from a comparison), and to 0 otherwise.

Bit 31 (Negative flag, N)

• It is set to bit 31 of the result of the instruction. If this result is regarded as a two's complement signed integer, then N = 1 if the result is negative and N = 0 if it is positive or zero.



Show the status of the C and Z flags after the addition of

- 0x0000009C and 0xFFFFFF64
- 0x0000009C and 0xFFFFFF69

;assume
R1 = 0x0000009C and
R2 = 0xFFFFFF64
ADDS R2,R1,R2
;add R1 to R2 and place the result in R2

C = 1 because there is a carry beyond the D31 bit.
Z = 1 because the R2 (the result) has value 0 in it after the addition.

0x0000009C	0000 0000 0000 0000 0000 0000 1001 1100
0xFFFFFF64 +	1111 1111 1111 1111 1111 1111 0110 0100
0x100000000 1	0000 0000 0000 0000 0000 0000 0000 0000

Show the status of the C and Z flags after the addition of

- 0x0000009C and 0xFFFFFF64
- 0x0000009C and 0xFFFFFF69

```
;assume R1 = 0x0000009C and R2 = 0xFFFFFF69
ADDS    R2,R1,R2          ;add R1 to R2 and place the result in R2
```

C = 1 because there is a carry beyond the D31 bit.
Z = 0 because the R2 (the result) does not have value 0 in it after the addition. (R2=0x00000005)

0x0000009C	0000 0000 0000 0000 0000 0000 1001 1100
0xFFFFFF69 +	1111 1111 1111 1111 1111 1111 0110 1001
0x100000005 1	0000 0000 0000 0000 0000 0000 0000 0101

Solved Example

Show the status of the Z flag during the execution of the following program:

```
MOV    R2,#4          ;R2 = 4
MOV    R3,#2          ;R3 = 2
MOV    R4,#4          ;R4 = 4
SUBS   R5,R2,R3       ;R5 = R2 - R3 (R5 = 4 - 2 = 2)
SUBS   R5,R2,R4       ;R5 = R2 - R4 (R5 = 4 - 4 = 0)
```

Solution:

The Z flag is raised when the result is zero. Otherwise, it is cleared (zero). Thus:

After	Value of R5	Z flag
SUBS R5,R2,R3	2	0
SUBS R5,R2,R4	0	1

3. Find the C and Z flag bits for the following code:

```
;assume R2 = 0xFFFFFF9F
;assume R1 = 0x00000061
ADDS    R2, R1, R2
```

4. Find the Z flag bit for the following code:

```
;assume R7 = 0x22
;assume R3 = 0x22
ADDS    R7, R3, R7
```

5. Find the C and Z flag bits for the following code:

```
;assume R2 = 0x67
;assume R1 = 0x99
ADDS    R2, R1, R2
```

Flag Bits Affected by Different Instructions

- Some instructions affect all the four flag bits C, Z, V, and N (e.g., ADDS). But some instructions affect no flag bits at all. The branch instructions are in this category. Some instructions affect only some of the flag bits. The logic instructions (e.g., ANDS) are in this category

Instruction	Flags Affected
ANDS	C, Z, N
ORRS	C, Z, N
MOVS	C, Z, N
ADDS	C, Z, N, V
SUBS	C, Z, N, V
B	No flags
Note that we cannot put S after B instruction.	

Table 2- 5: Flag Bits Affected by Different Instructions

- Find the status of Z flag and C Flag

```
MOV R0, #15
MOV R1,#10
SUBS R2, R0,R1
```

- Find the status of C, Z, N Flag

```
LDR R1,=0x0F000006
MOVS R2,R1,LSL #8
```

Flag bits and decision making

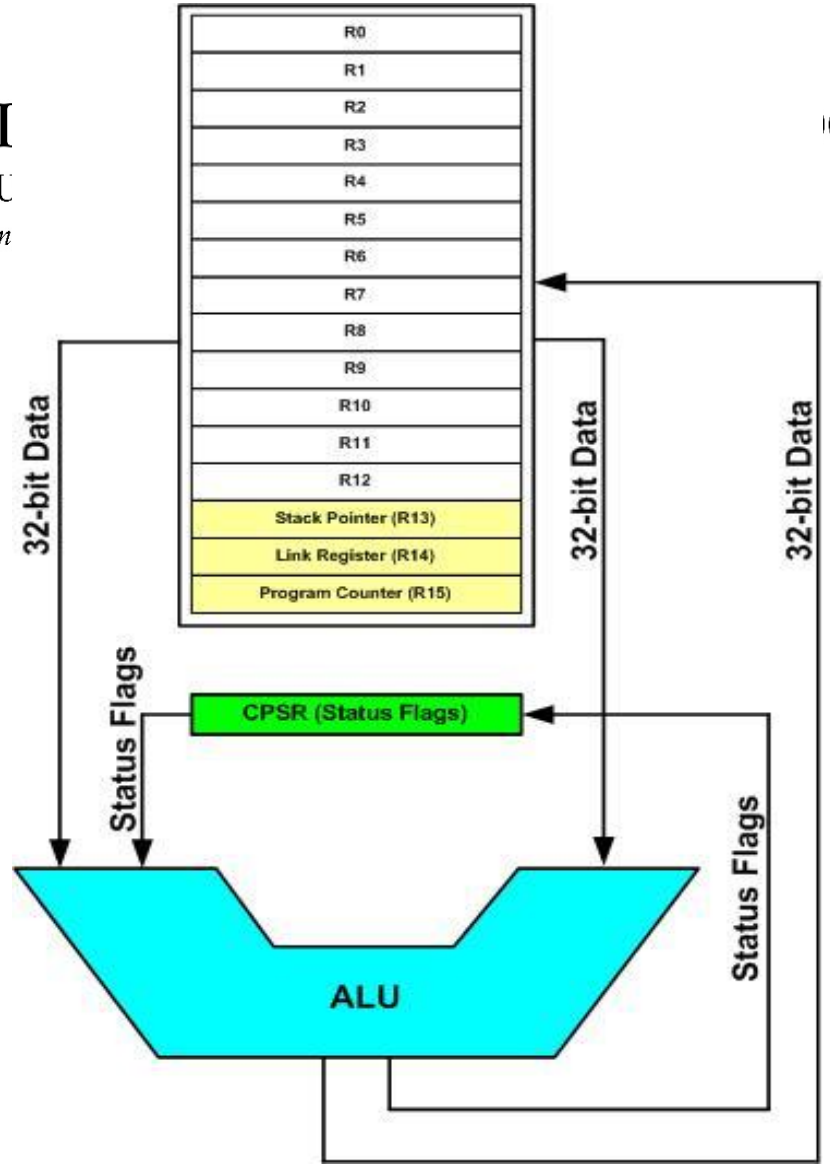
- There are instructions that will make a conditional jump (branch) based on the status of the flag bits.

Instruction	Flags Affected
BCS	Branch if C = 1
BCC	Branch if C = 0
BEQ	Branch if Z = 1
BNE	Branch if Z = 0
BMI	Branch if N = 1
BPL	Branch if N = 0
BVS	Branch if V = 1
BVC	Branch if V = 0

Table 2- 6: ARM Branch (Jump) Instructions Using Flag Bits

The Special Function Registers in ARM

- In ARM the R13, R14, R15, and CPSR (current program status register) registers are called *SFRs (special function registers)* since each one is dedicated to a specific function.
- A given special function register is dedicated to specific function such as status register, program counter, stack pointer, and so on. The function of each SFR is fixed by the CPU designer at the time of design because it is used for control of the microcontroller or keeping track of specific CPU status.
- The R13 is set aside for stack pointer. The R14 is designated as link register which holds the return address when the CPU calls a subroutine and the R15 is the program counter (PC).
- The PC (program counter) points to the address of the next instruction to be executed.
- The CPSR (current program status register) is used for keeping condition flags among other things. In contrast to SFRs, the GPRs (R0-R12) do not have any specific function and are used for storing general data.



ARM Registers and ALU

ARM Data Format

ARM data type

ARM has four data types. They are bit, byte (8-bit), half-word (16-bit) and word (32 bit).

Data format representation

There are several ways to represent a byte of data in the ARM assembler. The numbers can be in hex, binary, decimal, or ASCII formats.

Hex numbers

To represent Hex numbers in an ARM assembler we put 0x (or 0X) in front of the number

MOV R1,#0x99

Decimal numbers

To indicate decimal numbers in some ARM assemblers such as Keil we simply use the decimal (e.g., 12) and nothing before or after it.

MOV R7,#12

Binary numbers

To represent binary numbers in an ARM assembler we put 2_ in front of the number.

MOV R6,#2_10011001

Numbers in any base between 2 and 9

To indicate a number in any base n between 2 and 9 in an ARM assembler we simply use the n_ in front of it.

ASCII characters

To represent ASCII data in an ARM assembler we use single quotes as follows:

LDR R3,#‘2’ ;R3 = 00110010 or 32 in hex. ASCII of 2.

To represent a string, double quotes are used; and for defining ASCII strings (more than one character), we use the DCB directive.

ARM assembly language module

An ARM assembly language module has several constituent parts. These are:

- Extensible Linking Format (ELF) sections (defined by the AREA directive).
- Application entry (defined by the ENTRY directive).
- Program end (defined by the END directive).

Assembler Directives

- Assembler directives are the commands to the assembler that direct the assembly process.
- They do not generate any machine code i.e. they do not contribute to the final size of machine code and they are assembler specific

AREA:

The AREA directive tells the assembler to define a new section of memory. The memory can be code (instructions) or data and can have attributes such as READONLY, READWRITE and so on. This is used to define one or more blocks of indivisible memory for code or data to be used by the linker. The following is the format:

AREA sectionname attribute, attribute, …

The following line defines a new area named mycode which has CODE and REASDONLY attributes:

AREA mycode, CODE, READONLY

Commonly used attributes are CODE, DATA, READONLY, READWRITE

READONLY:

It is an attribute given to an area of memory which can only be read from. It is by **default for CODE.** This area is used to write the instructions.

READWRITE:

It is attribute given to an area of memory which can be read from and written to. **It is by default for DATA.**

CODE:

It is an attribute given to an area of memory used for executable machine instructions. **It is by default READONLY memory.**

DATA:

It is an attribute given to an area of memory used for data and no instructions can be placed in this area. **It is by default READWRITE memory.**

ALIGN:

It is used to indicate how memory should be allocated according to the addresses. When the ALIGN is used for CODE and READONLY, it is aligned in **4-bytes address boundary by default since the ARM instructions are 32 bit word.** If it is written as ALIGN = 3, it indicates that the information should be placed in memory with addresses of 2^3, that is for example **0x50000, 0x50008, 0x50010, 0x50018**

and so on.

EXPORT:

The EXPORT directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files.

DCD (Define constant word):

Allocates a word size memory and initializes the values. Allocates one or more words of memory, aligned on 4-byte boundaries and defines initial run time contents of the memory.

ENTRY:

The ENTRY directive declares an entry point to the program. It marks the first instruction to be executed. In applications using the C library, an entry point is also contained within the C library initialization code. Initialization code and exception handlers also contain entry points

END:

It indicates to the assembler the end of the source code. The END directive is the last line of the ARM assembly program and anything after the END directive in the source file is ignored by the assembler.

EQU:

Associate a symbolic name to a numeric constant.

RN (equate)

This is used to define a name for a register. The RN directive does not set aside a seperate storage for the name, but associates a register with that name. It improves the clarity.

```
VAL1 RN R1     ;define VAL1 as a name for R1
```

EQU (equate)

This is used to define a constant value or a fixed address. The EQU directive does not set aside storage for a data item, but associates a constant number with a data or an address label so that when the label appears in the program, its constant will be substituted for the label. The following uses EQU for the counter constant, and then the constant is used to load the R2 register:

```
COUNT EQU 0x25

MOV R2, #COUNT ;R2 = 0x25
```

Advantages of using EQU:

Assume that a constant (a fixed value) is used throughout the program, and the programmer wants to change its value everywhere. By the use of EQU, the programmer can change it once and the assembler will change all of its occurrences throughout the program. This allows the programmer to avoid searching the entire program trying to find every occurrence.

Using EQU for fixed data assignment

To get more practice using EQU to assign fixed data, examine the following:

```
DATA1  EQU    0x39              ;the way to define hex value
DATA2  EQU    2_00110101        ;the way to define binary value (35 in hex)
DATA3  EQU    39                ;decimal numbers (27 in hex)
DATA4  EQU    '2'               ;ASCII characters
```

RN (equate)

“RN” is used to give a CPU register a name. The RN directive allows the programmer to associate a register with a name. It improves the readability of the code.

Program 2-2: An ARM Assembly Language Program Using RN Directive

```
;ARM Assembly Language Program To Add Some Data
;and store the SUM in R3.

VAL1    RN      R1      ;define VAL1 as a name for R1
VAL2    RN      R2      ;define VAL2 as a name for R2
SUM     RN      R3      ;define SUM as a name for R3

        AREA    PROG_2_2, CODE, READONLY
ENTRY
MOV     VAL1, #0x25             ;R1 = 0x25
MOV     VAL2, #0x34             ;R2 = 0x34
ADD     SUM, VAL1,VAL2          ;R3 = R2 + R1
HERE    B       HERE
        END
```

INCLUDE directive : The include directive tells the ARM assembler to add the contents of a file to our program (like the #include directive in C language).

Assembler data allocation directives:

In most Assembly languages there are some directives to allocate memory and initialize its value. In ARM Assembly language DCB, DCD, and DCW allocate memory and initialize them. The SPACE directive allocates memory without initializing it.

Directive	Description
DCB	Allocates one or more bytes of memory, and defines the initial runtime contents of the memory
DCW	Allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory.
DCWU	Allocates one or more halfwords of memory, and defines the initial runtime contents of the memory. The data is not aligned.
DCD	Allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.
DCDU	Allocates one or more words of memory and defines the initial runtime contents of the memory. The data is not aligned.

DCB directive (define constant byte)

The DCB directive allocates a byte size memory and initializes the values.

```
MYVALUE         DCB       5             ;MYVALUE = 5
MYMSAGE         DCB       “HELLO WORLD”          ;string
```

DCW directive (define constant half-word)

The DCW directive allocates a half-word size memory and initializes the values.

```
MYDATA          DCW      0x20, 0xF230, 5000, 0x9CD7
```

DCD directive (define constant word)

The DCD directive allocates a word size memory and initializes the values.

```
MYDATA          DCD      0x200000, 0xF30F5, 5000000, 0xFFFF9CD7
```

ADR directive

To load registers with the addresses of memory locations we can also use the ADR pseudo-instruction which has a better performance. ADR has the following syntax:

ADR Rn, label

Ex:

ADR R2, OUR_FIXED_DATA ; point to OUR_FIXED_DATA

ALIGN

This is used to make sure data is aligned in 32-bit word or 16-bit half word memory address. The following uses ALIGN to make the data 32-bit word aligned:

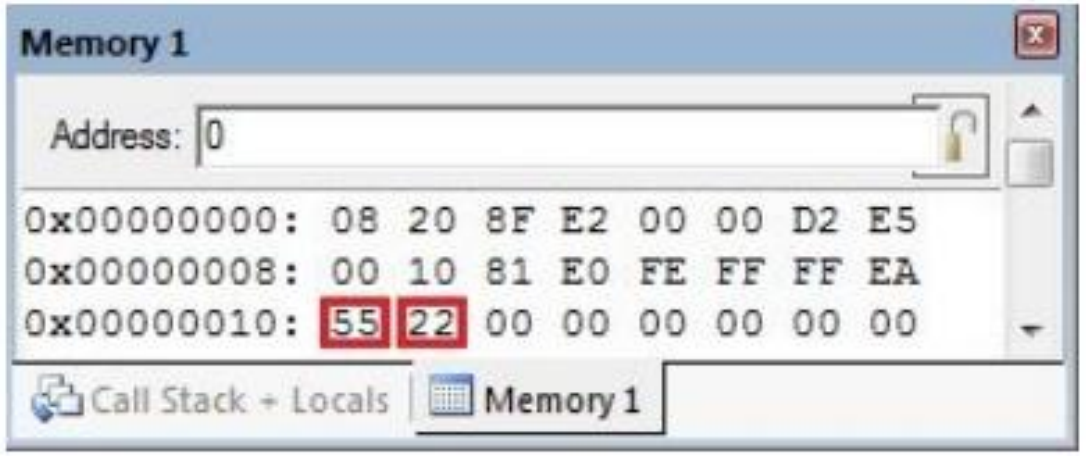
```
ALIGN 4 ;the next instruction is word (4 bytes) aligned …
ALIGN 2 ;the next instruction is half-word (2 bytes) aligned
```

Solved Examples:

Compare the result of using ALIGN in the following programs

```
AREA E2_7A, READONLY, CODE
ENTRY
ADR R2,DTA
LDRB R0,[R2]
ADD R1,R1,R0
H1 B H1
DTA DCB 0x55 DCB 0x22
END
```

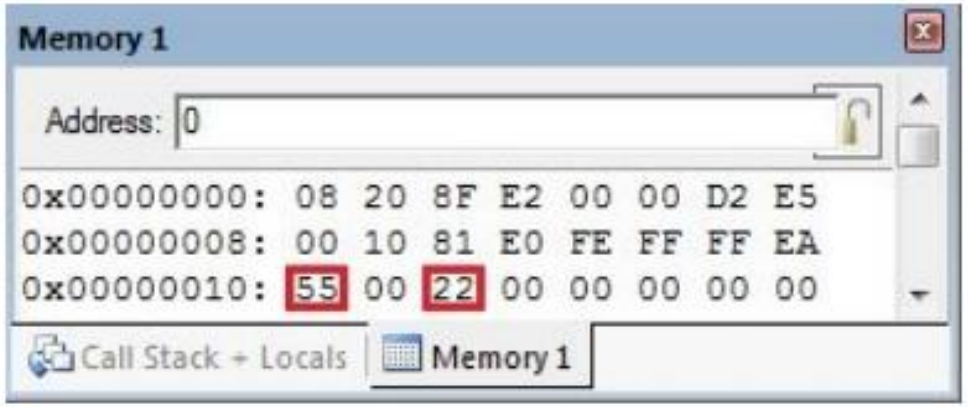
When there is no ALIGN directive the DCB directive allocates the first empty location for its data. In this example, address 0x10 is allocated for 0x55. So 0x22 goes to address 0x11.



```
    AREA    E2_7B,READONLY,CODE
    ENTRY
    ADR     R2,DTA
    LDRB    R0,[R2]
    ADD     R1,R1,R0
H1      B       H1

DTA     DCB     0x55
    ALIGN   2
    DCB     0x22
    END
```

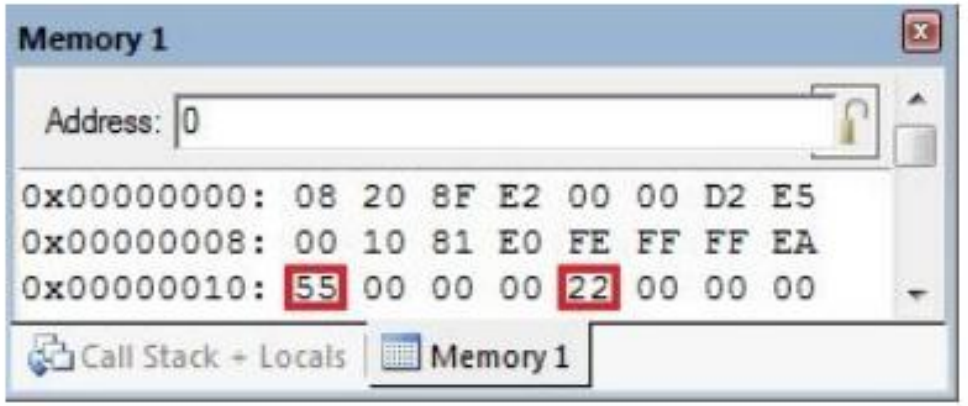
In the example the ALIGN is set to 2 which means the data should be put in a location with even address. The 0x55 goes to the first empty location which is 0x10. The next empty location is 0x11 which is not a multiple of 2. So, it is filled with 0 and the next data goes to location 0x12.



```
    AREA    E2_7C,READONLY,CODE
    ENTRY
    ADR     R2,DTA
    LDRB    R0,[R2]
    ADD     R1,R1,R0
H1      B       H1

DTA     DCB     0x55
    ALIGN   4
    DCB     0x22
    END
```

In the example the ALIGN is set to 4 which means the data should go to locations whose address is multiple of 4. The 0x55 goes to the first empty location which is 0x10. The next empty locations are 0x11, 0x12, and 0x13 which are not a multiple of 4. So, they are filled with 0s and the next data goes to location 0x14.



SPACE directive

Using the SPACE directive we can allocate memory for variables.

```
LONG_VAR SPACE 4      ;Allocate 4 bytes
OUR_ALFA SPACE 2      ;Allocate 2 bytes
```

Pseudo Instructions:

LDR Pseudo Instruction

LDR Rd,=32-bit_immidiate_value

Notice the = sign used in the syntax. The following pseudo-instruction loads R7 with 0x112233.

```
LDR R7, =0x112233
```

To load values less than 0xFF, "MOV Rd, #8-bit_immidiate_value" instruction is used since it is a real instruction of ARM, therefore more efficient in code size.

ADR Pseudo Instruction

To load registers with the addresses of memory locations we can also use the ADR pseudo-instruction which has a better performance. ADR has the following syntax:

```
ADR Rn, label
ADR R2, OUR_FIXED_DATA      ;point to OUR_FIXED_DATA
```

An Assembly language instruction consists of four fields:

[label] mnemonic [operands] [;comment]

The first column of each line is always considered as label. Thus, be careful to press a Tab at the beginning of each line that does not have label; otherwise, your instruction is considered as a label and an error message will appear when compiling.

Rules for labels in Assembly language

By choosing label names that are meaningful, a programmer can make a program much easier to read and maintain. There are several rules that names must follow.

- First, each label name must be unique.
- The names used for labels in Assembly language programming consist of alphabetic letters in both uppercase and lowercase, the digits 0 through 9, and the special characters question mark (?), period (.), at (@), underline (_), and dollar sign ($).
- The first character of the label must be an alphabetic character. In other words, it cannot be a number.
- Every assembler has some reserved words that must not be used as labels in the program. Foremost among the reserved words are the mnemonics for the instructions. For example, “MOV” and “ADD” are reserved because they are instruction mnemonics.

ARM Memory Map and Memory Access

The ARM CPU uses 32-bit addresses which gives us a maximum of 04GB (gigabytes) of memory space.

This 4GB of memory space has addresses, 0x00000000 to 0xFFFFFFFF

meaning each byte is assigned a unique address (ARM is a byte-addressable CPU).

ARM Memory Map and Memory Access

D31 D24	D23 D16	D15 D8	D7 D0	
0x00000003	0x00000002	0x00000001	0x00000000	0x00000000
0x00000007	0x00000006	0x00000005	0x00000004	0x00000004
0x0000000B	0x0000000A	0x00000009	0x00000008	0x00000008
0x0000000F	0x0000000E	0x0000000D	0x0000000C	0x0000000C
⋮	⋮	⋮	⋮	
0xFFFFFFF3	0xFFFFFFF2	0xFFFFFFF1	0xFFFFFFF0	0xFFFFFFF0
0xFFFFFFF7	0xFFFFFFF6	0xFFFFFFF5	0xFFFFFFF4	0xFFFFFFF4
0xFFFFFFFB	0xFFFFFFFA	0xFFFFFFF9	0xFFFFFFF8	0xFFFFFFF8
0xFFFFFFFF	0xFFFFFFFE	0xFFFFFFFD	0xFFFFFFFC	0xFFFFFFFC

ARM Assembly Language Programming & Architecture by Mazidi, et al.

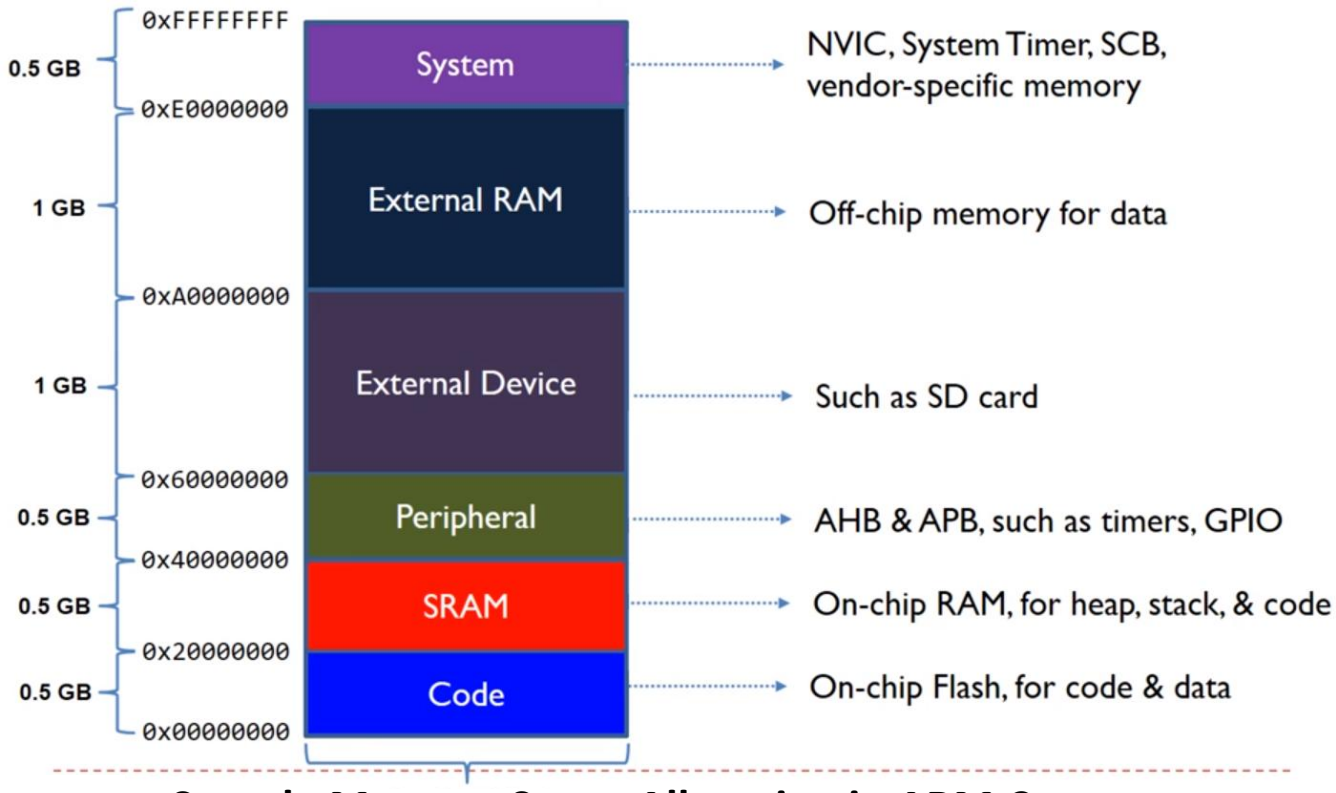
The 4GB of memory space is divided into three regions:

code, data, and peripheral devices.

PPB – Private Peripheral Bus

No ARM manufacturer has populated the entire 4 GB of memory space with on-chip ROM, RAM, and I/O peripherals.

Address range	Name	Description
0x00000000-0x1FFFFFFF	Code	ROM or Flash memory
0x20000000-0x3FFFFFFF	SRAM	SRAM region used for on-chip RAM
0x40000000-0x5FFFFFFF	Peripheral	On-chip peripheral address space
0x60000000-0x9FFFFFFF	RAM	Memory, cache support
0xA0000000-0xDFFFFFFF	Device	Shared and non-shared device space
0xE0000000-0xFFFFFFFF	System	PPB and vendor system peripherals



Sample Memory Space Allocation in ARM Cortex

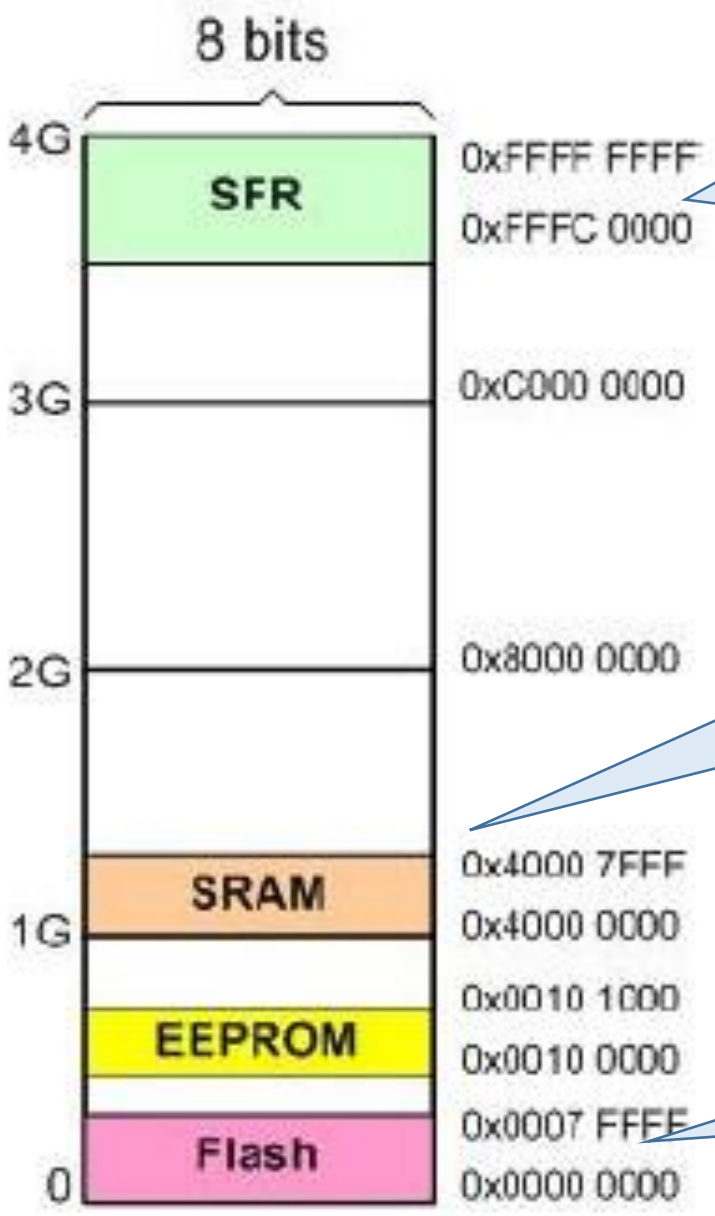
The address locations of memory used by

varies among the family members and **chip manufacturers**

Flash ROM is used for program code

SRAM for scratch pad data

Memory-mapped I/O ports for peripherals



This area is dedicated to general purpose I/O (GPIO) and special function registers (SFRs) of peripherals such as timers, serial communication, ADC, and so on.

A RAM space ranging from a few kilobytes to several hundred kilobytes is set aside mainly for data storage. The data RAM space is used for data variables and stack and is accessed by the microcontroller instructions.

The program space is used for the program code. In ARM microcontroller chips, the code ROM space is of Flash type memory.

Memory space allocation in the ARM

The ARM has 4G bytes of directly accessible memory space. This memory space has addresses 0 to 0xFFFFFFFF. The 4G bytes of memory space can be divided into five sections. They are as follows:

1. **On-chip peripheral and I/O registers:** This area is dedicated to **general purpose I/O (GPIO) and special function registers (SFRs)** of peripherals such as timers, serial communication, ADC, and so on. In other words, ARM uses memory-mapped I/O.
2. **On-chip data SRAM:** A RAM space ranging from a few kilobytes to several hundred kilobytes is set aside mainly for data storage. **The data RAM space is used for data variables and stack and is accessed by the microcontroller instructions. It is Volatile memory.**
3. **On-chip EEPROM:** A block of memory from 1K bytes to several thousand bytes is set aside for EEPROM memory. In some applications the EEPROM is used for program code storage, it is used for **saving critical data**. Not all ARM chips have on-chip EEPROM.

4. On-chip Flash ROM: A block of memory from a few kilobytes to several hundred kilobytes is set aside for program space. The program space is used for the program code.

5. Off-chip DRAM space: A DRAM memory ranging from few megabytes to several hundred mega bytes can be implemented for external memory connection.

Differences among the on-chip Flash ROM, data SRAM, and EEPROM memories in ARM microcontrollers used for embedded products

- The data SRAM is used by the **CPU for data variables and stack**, whereas the EEPROMs are considered to be memory that one can also add **externally to the chip**. In other words, while many ARM microcontroller chips have no EEPROM memory, it is very unlikely for an ARM microcontroller to have no on-chip data SRAM.

- The **on-chip Flash ROM is used for program code**, while **the EEPROM is used most often for critical system data that must not be lost if power is cut off**.

- The data SRAM is volatile memory and its contents are lost if the power to the chip is cut off. Since volatile data SRAM is used for dynamic variables (constantly changing data) and stack. **We need EEPROM memory to secure critical system data that does not change very often and will not be lost in the event of power failure**.

- The on-chip Flash ROM is **programmed and erased in block size**. **The block size is 8, 16, 32, or 64 bytes or more depending on the chip technology**. That is not the case with EEPROM, since the **EEPROM is byte programmable and erasable**. Both the EEPROM and Flash memories have limited number of erase/write cycles, which can be 100,000 or more.

- While there is an absolute standard for the ARM instructions that all licensees of ARM must follow, there is no standard for exact locations and types of memory and peripherals.
- Therefore the licensees can implement the memory and peripherals as they choose. For this reason the amount and the address locations of memory used by Flash ROM, SRAM, and I/O peripherals varies among the family members and chip manufacturers.
- The ARM manufacturer datasheet should give you the details of the memory map for both on-chip and off-chip memory and peripherals

How to calculate size of memory by given a range of address?

Address Allocations in the memory will start from '0'

range = last address - first address + 1

To convert from bytes to megabytes use the relationship

```
1 MB = 1 Megabyte = 1024 * 1 KB = 1,048,576 bytes.
```

```
First @      Last @
```

range1: FD00 0000 to FDFF FFFF

range2 : D000 0000 to DFFF FFFF

range3 : FA00 0000 to FBFF FFFF

A given ARM chip has the following address assignments. Calculate the space and the amount of memory given to each section.
(a) Address range of 0x00100000 – 0x00100FFF for EEPROM
(b) Address range of 0x40000000 – 0x40007FFF for SRAM
(c) Address range of 0x00000000 – 0x0007FFFF for Flash
(d) Address range of 0xFFFC0000 – 0xFFFFFFFF for peripherals

A given ARM chip has the following address assignments. Calculate the space and the amount of memory given to each section.
(a) Address range of 0x00100000 – 0x00100FFF for EEPROM
(b) Address range of 0x40000000 – 0x40007FFF for SRAM
(c) Address range of 0x00000000 – 0x0007FFFF for Flash
(d) Address range of 0xFFFC0000 – 0xFFFFFFFF for peripherals

Solution:

(a) With address space of 0x00100000 to 00100FFF, we have 00100FFF – 00100000 = 0FFF bytes. Converting 0FFF to decimal, we get 4,095 + 1, which is equal to 4K bytes.

(b) With address space of 0x40000000 to 0x40007FFF, we have 40007FFF – 40000000 = 7FFF bytes. Converting 7FFF to decimal, we get 32,767 + 1, which is equal to 32K bytes.

(c) With address space of 0000 to 7FFFF, we have 7FFFF – 0 = 7FFFF bytes. Converting 7FFFF to decimal, we get 524,287 + 1, which is equal to 512K bytes.

(d) With address space of FFFC0000 to FFFFFFFF, we have FFFFFFFF–FFFC0000 = 3FFFF bytes. Converting 3FFFF to decimal, we get 262,143 + 1, which is equal to 256K bytes.

Find the address space range of each of the following memory of an ARM chip:

(a) 2 KB of EEPROM starting at address 0x80000000

(b) 16 KB of SRAM starting at address 0x90000000

(c) 64 KB of Flash ROM starting at address 0xF0000000

Find the address space range of each of the following memory of an ARM chip:

(a) 2 KB of EEPROM starting at address 0x80000000

(b) 16 KB of SRAM starting at address 0x90000000

(c) 64 KB of Flash ROM starting at address 0xF0000000

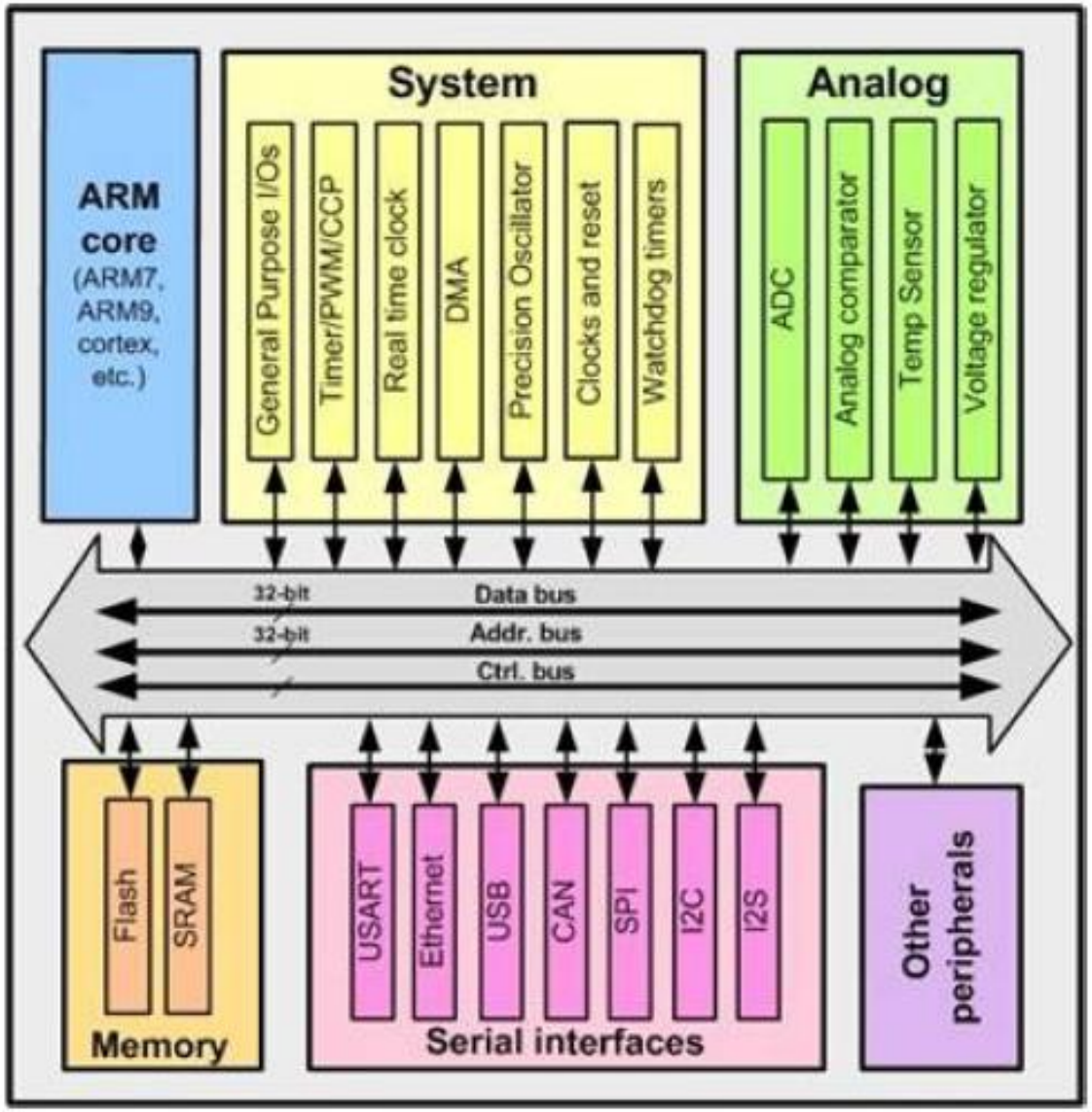
Solution:

(a) With 2K bytes of on-chip EEPROM memory, we have 2048 bytes (2 × 1024 = 2048). This maps to address locations of 0x80000000 to 0x800007FF. Notice that 0 is always the first location.

(b) With 16K bytes of on-chip SRAM memory, we have 16,384 bytes (16 × 1024 = 16,384), and 16,384 locations gives 0x90000000–0x90003FFF.

(c) With 64K we have 65,536 bytes (64 × 1024 = 65,536), therefore, the memory space is 0xF0000000 to 0xF000FFFF.

ARM buses and memory access



Memory Connection Block Diagram in ARM

ARM systems use DRAM for the RAM memory, just like the x86 and Pentium PCs.

DRAM as primary memory to store both the operating systems and the applications.

DRAM as primary memory to store both the operating systems and the applications.

In such systems, the Flash memory will be holding the POST (power on self test), BIOS (basic Input/output systems) and boot programs.

Same as x86 system, such systems have both on-chip and off-chip high speed SRAM for cache.

ARM chips on the market with some on-chip Flash ROM, SRAM, and memory decoding circuitry for connection to external (off-chip) memory.

Memory Connection Block Diagram in ARM

D31–D0 Data bus: The 32-bit data bus of the ARM provides the 32-bit data path to the on-chip and offchip memory and peripherals. They are grouped into 8-bit data chunks, D0–D7, D8–D15, D16–D23, and D24–D31.

A31–A0: These signals provide the 32-bit address path to the on-chip and off-chip memory and peripherals. Since the ARM supports data access of byte (8 bits), half word (16 bits), and word (32 bits), the buses must be able to access any of the 4 banks of memory connected to the 32-bit data bus.

The A0 and A1 are used to select one of the 4 bytes of the D31-D0 data bus.

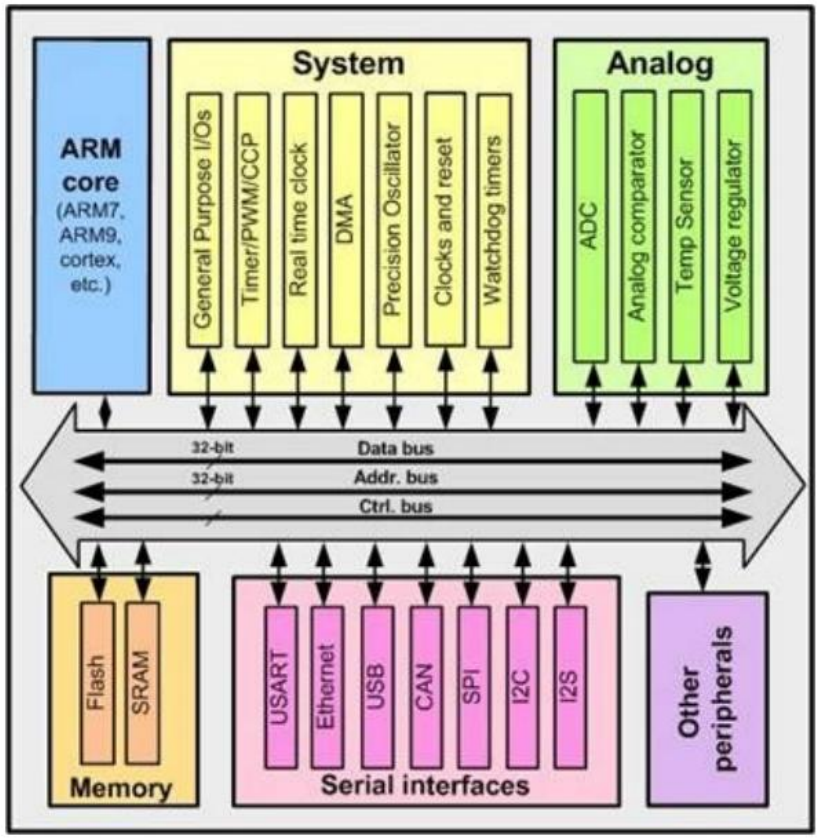


Figure 6-2: Memory Connection Block Diagram in ARM

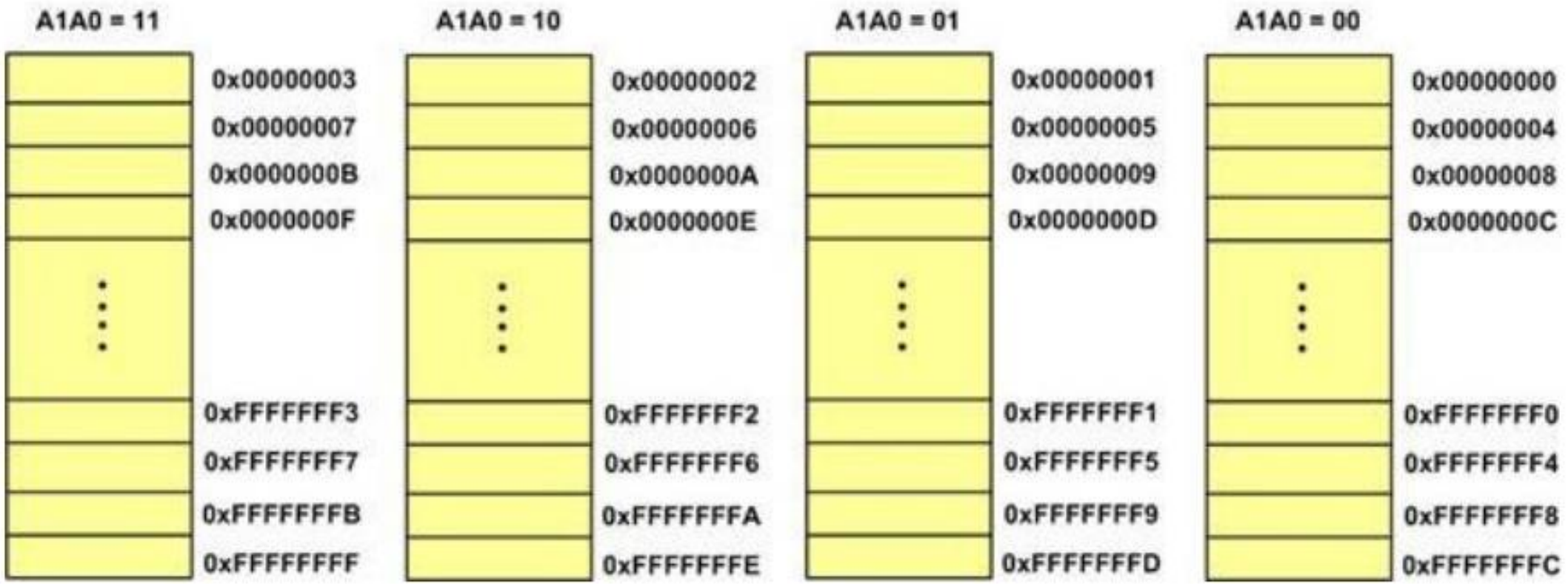


Figure 6-3: Memory Block Diagram in ARM

- **AHB and APB buses :** The ARM CPU is connected to the on-chip memory via an AHB (advanced high-performance bus). The AHB is used not only for connection to on-chip ROM and RAM, it is also used for connection to some of the high speed I/Os (input/output) such as GPIO (general purpose I/O).
- ARM chip also has the APB (advanced peripherals bus) bus dedicated for communication with the on-chip peripherals such as timers, ADC, serial COM, SPI, I2C, and other peripheral ports.
- While we need the 32-bit data bus between CPU and the memory (RAM and ROM), many slower peripherals are 8 or 16 bits and there is no need for entire fast 32-bit data bus pathway. For this reason, ARM uses the AHB-to-APB bridge to access the slower on-chip devices such as peripherals.
- Also since peripherals do not need a high speed bus, a bridge between AHB and APB allows going from the higher speed bus of AHB to lower speed bus of peripherals. The AHB bus allows a single-cycle access. See Figure 6-4 for AHB-to-APB bridge

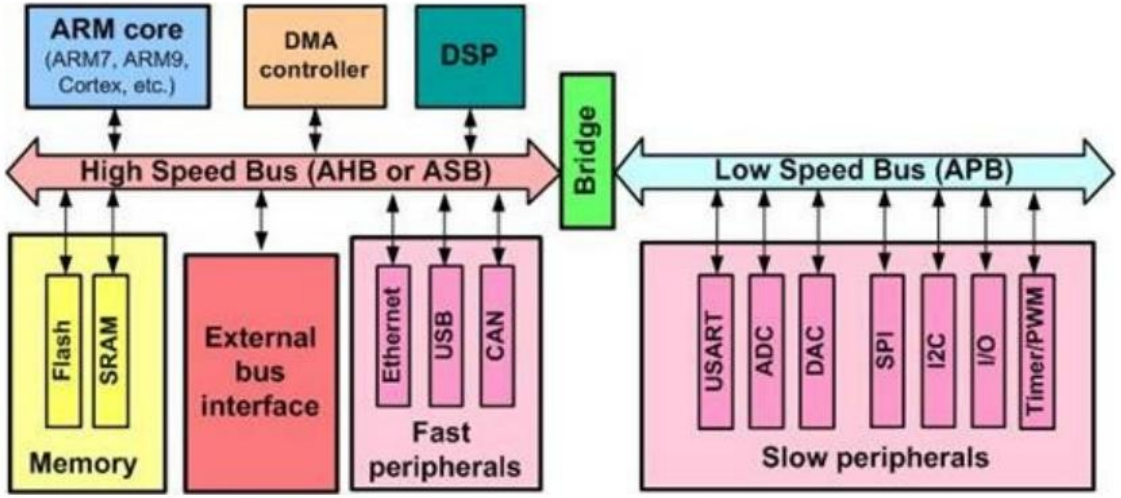


Figure 6-4: AHB and APB in ARM

Bus cycle time

- **Bus cycle time:** To access a device such as memory or I/O, the CPU provides a fixed amount of time called a bus cycle time. During this bus cycle time, the read or write operation of memory or I/O must be completed.
- Memory cycle time (MC) : The bus cycle time used for accessing memory is often referred to as **MC (memory cycle) time.**
- **Memory Read cycle time:** The time from when the CPU provides the addresses at its address pins to when the data is expected at its data pins is called **memory read cycle time.**
- While for **on-chip memory the cycle time can be 1 clock**, in the **off-chip memory the cycle time is often 2 clocks.** If memory is slow and its access time does not match the MC time of the CPU, extra time can be requested from the CPU to extend the read cycle time**. This extra time is called a wait state (WS).**

- Memory Cycle Time=Bus Cycle Time× (Clocks +Number of WS)

- Bus Cycle Time=1/ Bus Speed

- Calculate the memory cycle time of a 50-MHz bus system with (a) 0 WS, (b) 1 WS, and (c) 2 WS. Assume that the bus cycle time for off-chip memory access is 2 clocks.

- Calculate the memory cycle time of a 50-MHz bus system with (a) 0 WS, (b) 1 WS, and (c) 2 WS. Assume that the bus cycle time for off-chip memory access is 2 clocks.

Solution:

1/50 MHz = 20 ns is the bus clock period. Since the bus cycle time of zero wait states is 2 clocks, we have:

Memory cycle time with 0 WS $2 \times 20 = 40$ ns

Memory cycle time with 1 WS $40 + 20 = 60$ ns

Memory cycle time with 2 WS $40 + 2 \times 20 = 80$ ns

Bus bandwidth

- The rate of data transfer is generally called bus bandwidth. In other words, bus bandwidth is a measure of how fast buses transfer information between the CPU and memory or peripherals. The wider the data bus, the higher the bus bandwidth.
- The advantage of the wider external data bus comes at the cost of increasing the die size for system on-chip (SOC) or the printed circuit board size for off-chip memory
- The speed of the CPU must be matched with the higher bus bandwidth; otherwise, there is no use for a fast CPU.
- Bus bandwidth is measured in MB (megabytes) per second and is calculated as follows:

$$\text{bus bandwidth} = (1/\text{bus cycle time}) \times \text{bus width in byte}$$

Calculate memory bus bandwidth for the following CPU if the bus speed is 100 MHz.

- **ARM Thumb with 0 WS and 1 WS**
- **ARM with 0 WS and 1 WS**

Assume that the bus cycle time for off-chip memory access is 2 clocks.

Calculate memory bus bandwidth for the following CPU if the bus speed is 100 MHz.

- **ARM Thumb with 0 WS and 1 WS (16-bit data bus)**
- **ARM with 0 WS and 1 WS (32-bit data bus)**

Assume that the bus cycle time for off-chip memory access is 2 clocks.

Solution: The memory cycle time for both is 2 clocks, with zero wait states. With the 100 MHz bus speed we have a bus clock of 1/100 MHz = 10 ns.

(a) Bus bandwidth = (1/(2 × 10 ns)) × 2 bytes = 100M bytes/second (MB/s)

(b) With 1 wait state, the memory cycle becomes 3 clock cycles 3 × 10 = 30 ns and the memory bus bandwidth is = (1/30 ns) × 2 bytes = 66.6 MB/s

(c) (b) Bus bandwidth = (1/(2 × 10 ns)) × 4 bytes = 200 MB/s With 1 wait state, the memory cycle becomes 3 clock cycles 3 × 10 = 30 ns and the memory bus bandwidth is = (1/30 ns) × 4 bytes = MB/s

Exercise

Calculate the memory bus bandwidth for the following systems.

- a. ARM thumb of 200 MHz bus speed and 0 WS
- b. ARM of 80 MHz bus speed and 1 WS

Code memory region

- The 4 GB of ARM memory space is organized as 1G × 32 bits since the ARM instructions are 32-bit.
- The internal data bus of the ARM is 32-bit, allowing the transfer of one instruction into the CPU every clock cycle.
- This is one of the benefits of the RISC fixed instruction size. The fetching of an instruction in every clock cycle can work only if the code is word aligned, meaning each instruction is placed at an address location ending with 0, 4, 8, or C.

Code memory region

Example:

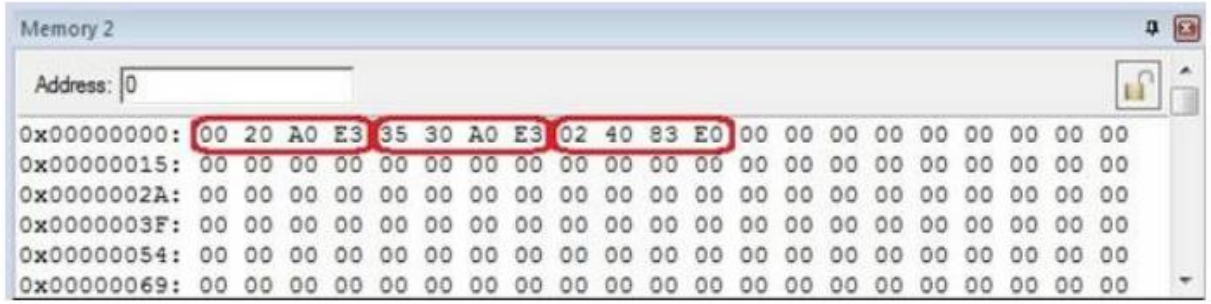
- Compile and debug the following code in Keil and see the placement of instructions in memory locations.

```
AREA ARMex, CODE, READONLY
ENTRY
MOV R2,#0x00 ;R2=0x00
MOV R3,#0x35 ;R3=0x35
ADD R4,R3,R2
END ;Mark end of file
```

Solution :

As you can see in the figure, the first MOV instruction starts from location 0x00000000, the second MOV instruction starts from location 0x00000004 and the ADD instruction starts from location 0x00000008.

The following image displays the first locations of memory. The code of the first MOV instruction is located in the first word (four bytes) of memory which is word aligned. The same rule applies for the other instructions. Note that the code of MOV R2,0 is E3 A0 20 00 but 00 20 A0 E3 is stored in the memory. W



SRAM memory region

- A section of the memory space is used by SRAM. The SRAM can be on-chip or offchip (external). The same way that every ARM chip has some on-chip Flash ROM for code, a portion of the memory region is used by the on-chip SRAM. This on-chip SRAM is used by the CPU for scratch pad to store parameters. It is also used by the CPU for the purpose of the stack.

Data misalignment in SRAM

The case of misaligned data has a major effect on the ARM bus performance. If the data is aligned, for every memory read cycle, the ARM brings in 4 bytes of information (data or code) using the D31–D0 data bus. Such data alignment is referred to as word alignment. To make data word aligned, the least significant digits of the hex addresses must be 0, 4, 8, or C (in hex).

- **The placement of data in SRAM by the programmer that can be nonaligned and therefore subject to memory access penalty.**
- The single cycle access of memory is also used by ARM to bring into registers 4 bytes of data every clock cycle assuming that the data is aligned.

- The use of align directive for RAM data makes sure that each word is located at an address location ending with address of 0, 4, 8, or C.
- If our data is word size (using DCDU directive) then the use of align directive at the start of the data section guaranties all the data placements will be word aligned. When a word size data is defined using the DCD directive, the assembler aligns it to be word aligned

Accessing non-aligned data

- In the 8-bit system, accessing a word (4 bytes) is treated like accessing four consecutive bytes regardless of the address location. Since accessing a byte takes one memory cycle, accessing 4 bytes will take 4 memory cycles. In the 32-bit system, accessing a word with an aligned address takes one memory cycle. That is because each byte is carried on its own data path of D0–D7, D8–D15, D16–D23, and D24–D31 in the same memory cycle

LDR R1,[R0]

As a case of aligned data, assume that R0 = 0x80000000. In this instruction, 4 bytes contents of memory locations 0x80000000 through 0x80000003 are being fetched in one cycle. In only one cycle, the ARM CPU accesses locations 0x80000000 through 0x80000003 and puts it in R1

Now assuming that R0 = 0x80000001 in this instruction, 8 bytes contents of memory locations 0x80000000 through 0x80000007 are being fetched in two consecutive cycles but only 4 bytes of it are used.
In the first cycle, the ARM CPU accesses locations 0x80000000 through 0x80000003 and puts them in R1 only the desired three bytes of locations 0x800000001 through 0x80000003. In the second cycle, the contents of memory locations 0x8000004 through 0x80000007 are accessed and only the desired byte of 0x80000004 is put into R1

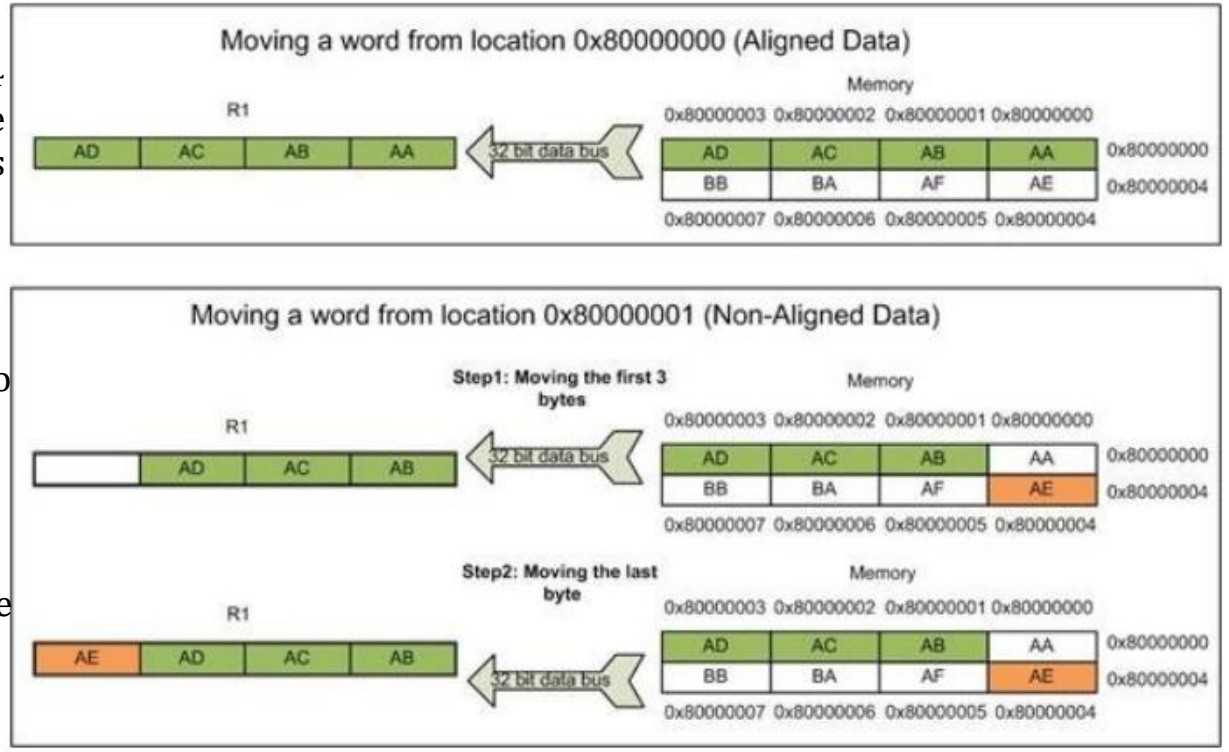


Figure 6-5: Memory Access for Aligned and Non-aligned Data

Example 6-4

Show the data transfer of the following cases and indicate the number of memory cycle times it takes for data transfer. Assume that R2 = 0x4598F31E.

```
LDR     R1,=0x40000000          ;R1=0x40000000
LDR     R2,=0x4598F31E          ;R2=0x4598F31E
STR     R2,[R1]          ;Store R2 to location 0x40000000
ADD     R1,R1,#1         ;R1 = R1 + 1 = 0x40000001
STR     R2,[R1]          ;Store R2 to location 0x40000001
ADD     R1,R1,#1         ;R1 = R1 + 1 = 0x40000002
STR     R2,[R1]          ;Store R2 to location 0x40000002
ADD     R1,R1,#1         ;R1 = R1 + 1 = 0x40000003
STR     R2,[R1]          ;Store R2 to location 0x40000003
```

Exercise

- Show the data transfer of the following LDRB instructions with appropriate figures and indicate the number of memory cycle times it takes for data transfer.

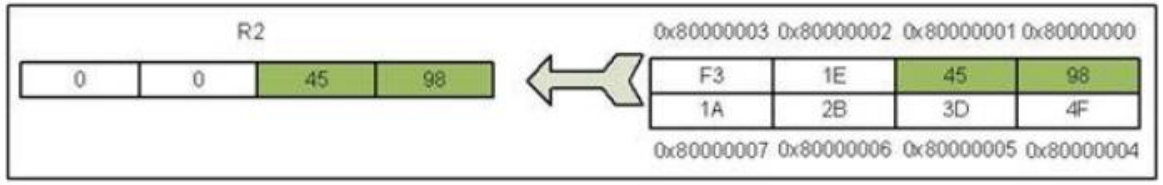
```
LDR R1,=0x80000000
LDR R3,=0xF31E4598
LDR R4,=0x1A2B3D4F
STR R3,[R1]
STR R4,[R1,#4]
LDRB R2,[R1]
LDRB R2,[R1,#1]
LDRB R2,[R1,#2].
```

Example

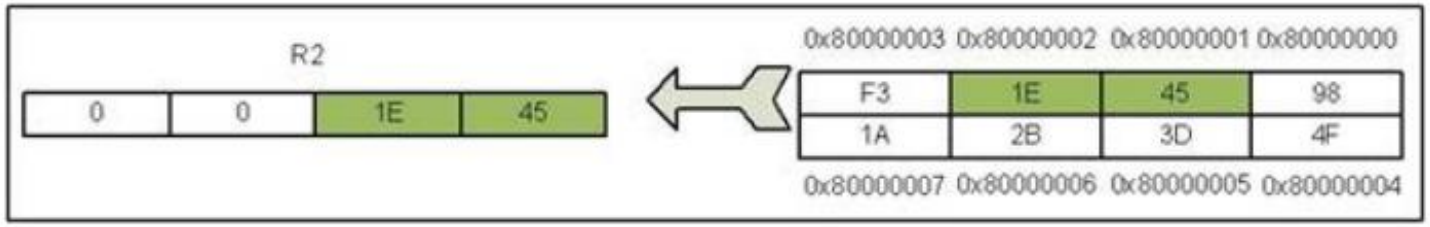
- Show the data transfer of the following LDRH instructions and indicate the number of memory cycle times it takes for data transfer.

```
LDR R1,=0x80000000 ;R1=0x80000000
LDR R3,=0xF31E4598 ;R3=0xF31E4598
LDR R4,=0x1A2B3D4F ;R4=0x1A2B3D4F
STR R3,[R1] ;(STR R3,[R1])stores R3 to location 0x80000000
STR R4,[R1,#4] ;(STR R4,[R1+4]) stores R4 to location 0x80000004
LDRH R2, [R1] ;loads two bytes from location 0x80000000 to R2
LDRH R2, [R1,#1] ;loads two bytes from location 0x80000001 to R2
LDRH R2, [R1,#2] ;loads two bytes from location 0x80000002 to R2
LDRH R2, [R1,#3] ;loads two bytes from location 0x80000003 to R
```

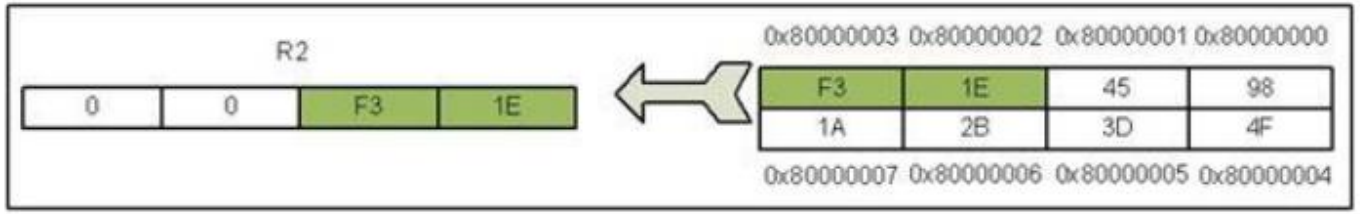
- In the LDRH R2,[R1] instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed but only 0x80000000 and 0x80000001 are used to get the 16 bits to R2. This address is halfword aligned since the least significant digit is 0. Therefore, it takes only one memory cycle to transfer the data. Now, R2=0x00004598



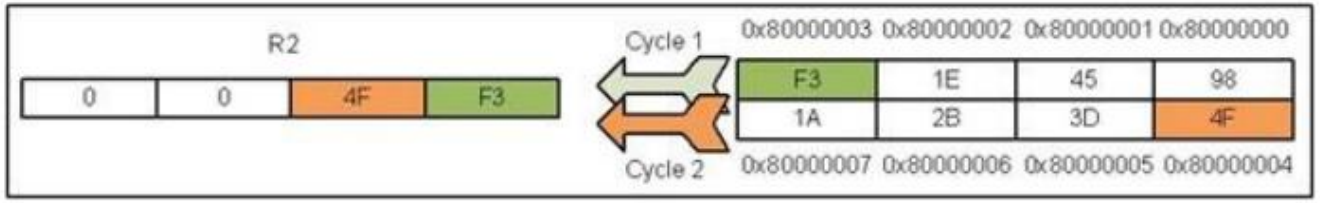
For the LDRH R2,[R1,#1], instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed, but only 0x80000001 and 0x80000002 are used to get the 16 bits to R2. Therefore, it takes only one memory cycle to transfer the data. Now, R2=0x00001E45.



- For the LDRH R2,[R1,#2], instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed, but only 0x80000002 and 0x80000003 are used to get the 16 bits to R2. Therefore, it takes only one memory cycle to transfer the data. Now, R2=0x0000F31E.



For the LDRH R2,[R1,#3] instruction, in the first memory cycle, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed, but only 0x80000003 is used to get the lower 8 bits to R2. In the second memory cycle, the address locations 0x80000004, 0x80000005, 0x80000006, and 0x80000007 are accessed where only the 0x80000004 location is used to get the upper 8 bits to R2. Now, R2=0x00004FF3



Exercise

Show the data transfer of the following LDRB instructions and indicate the number of memory cycle times it takes for data transfer.

```
LDR     R1,=0x80000000          ;R1=0x80000000
LDR     R3,=0xF31E4598          ;R3=0xF31E4598
LDR     R4,=0x1A2B3D4F          ;R4=0x1A2B3D4F
STR     R3,[R1]
;Store R3 to location 0x80000000
STR     R4,[R1,#4]
;(STR R4,[R1+4]) Store R4 to location 0x80000004
        LDRB    R2,[R1]
;load one byte from location 0x80000000 to R2
        LDRB    R2,[R1,#1]
;(LDRB R2,[R1+1]) load one byte from location 0x80000001
        LDRB    R2,[R1,#2]
;(LDRB R2,[R1+2]) load one byte from location 0x80000002
        LDRB    R2,[R1,#3]
;(LDRB R2,[R1+3]) load one byte from location 0x80000003
```

Little Endian vs. Big Endian

- In storing data, the ARM follows the little endian convention.
- The little endian places the least significant byte (little end of the data) in the low address and the big endian is the opposite.
- The origin of the terms big endian and little endian is from a Gulliver's Travels story about how an egg should be opened: from the big end or the little end. ARM supports both little and big endian.
- In ARM little endian is the default. Some ARM chip manufacturers provide an option for changing it to big endian.

- Show how data is placed after execution of the following code using

a) little endian and

b) big endian.

```
LDR R2,=0x7698E39F ;R2=0x7698E39F
LDR R1,=0x80000000
STR R2,[R1]
```

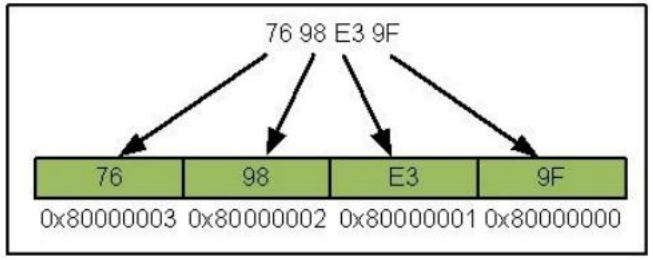
a) For little endian we have:

Location 80000000 = (9F)

Location 80000001 = (E3)

Location 80000002 = (98)

Location 80000003 = (76)



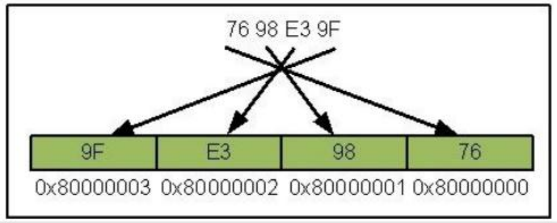
b) For big endian we have:

Location 80000000 = (76)

Location 80000001 = (98)

Location 80000002 = (E3)

Location 80000003 = (9F)



Load and Store Instructions in ARM

MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

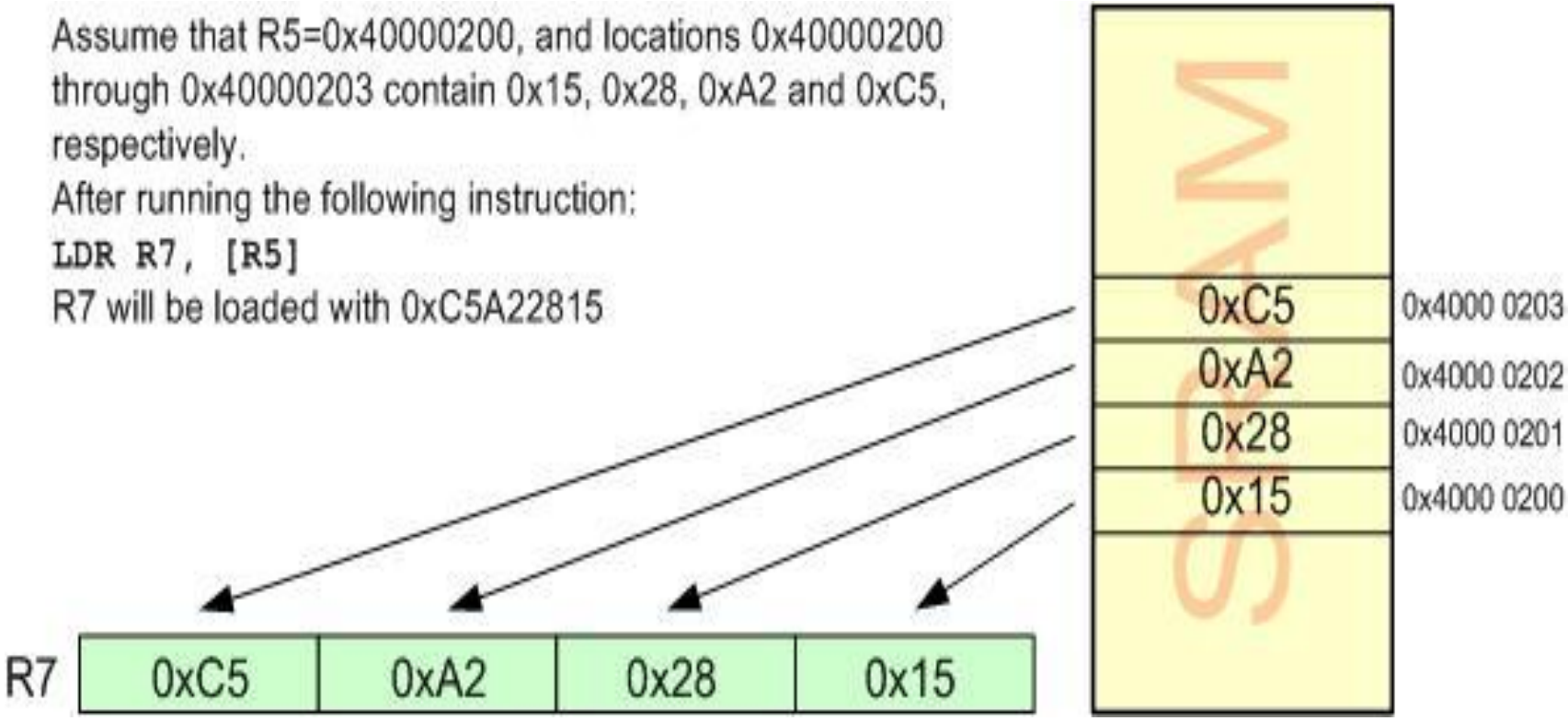
- Every instruction of ARM is fixed at 32-bit. The fixed size instruction is one of the most important characteristics of RISC architecture.

LDR Rd, [Rx] instruction
;load Rd with the contents of location pointed to by Rx register.

Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x15, 0x28, 0xA2 and 0xC5, respectively.
After running the following instruction:

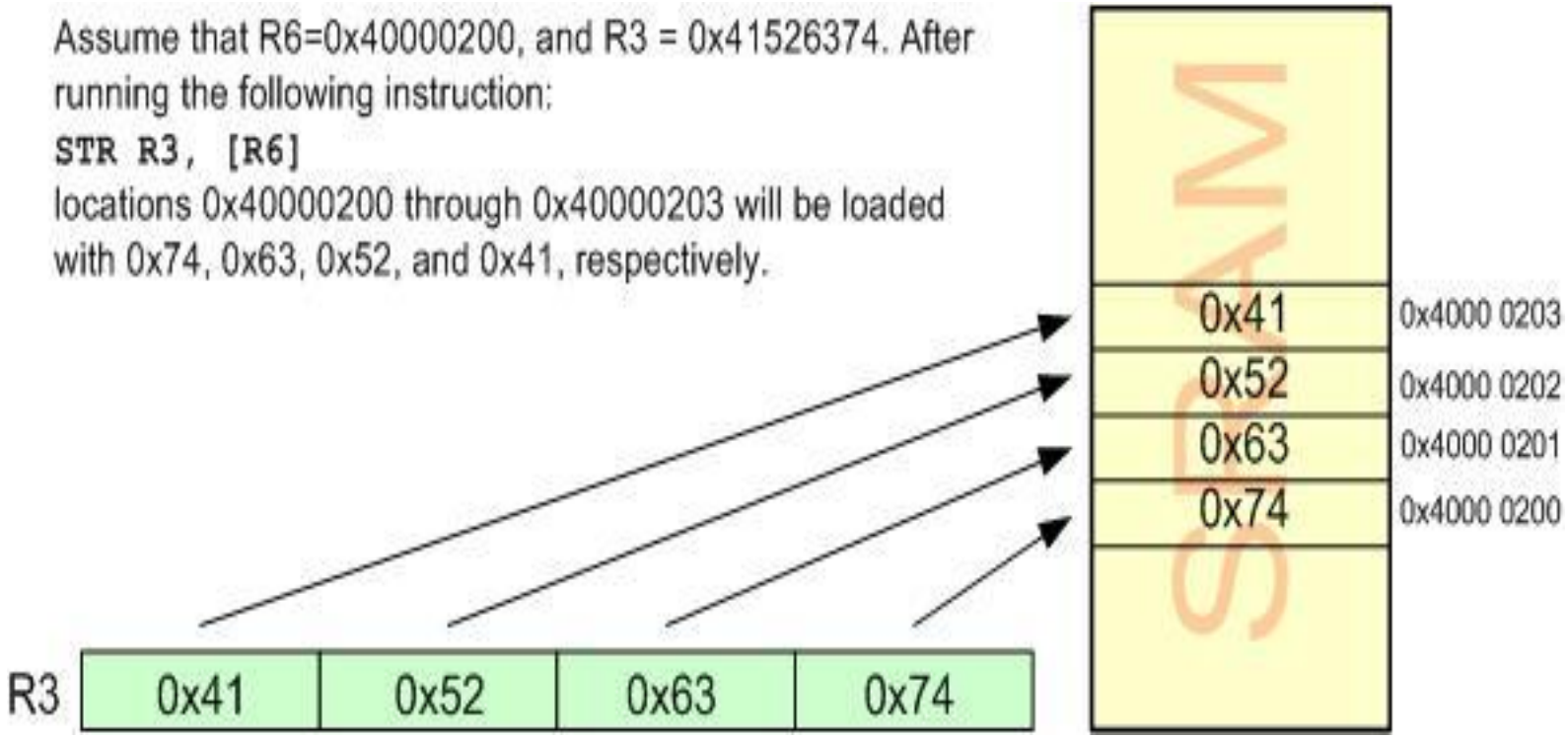
```
LDR R7, [R5]
```

R7 will be loaded with 0xC5A22815



Executing the LDR Instruction

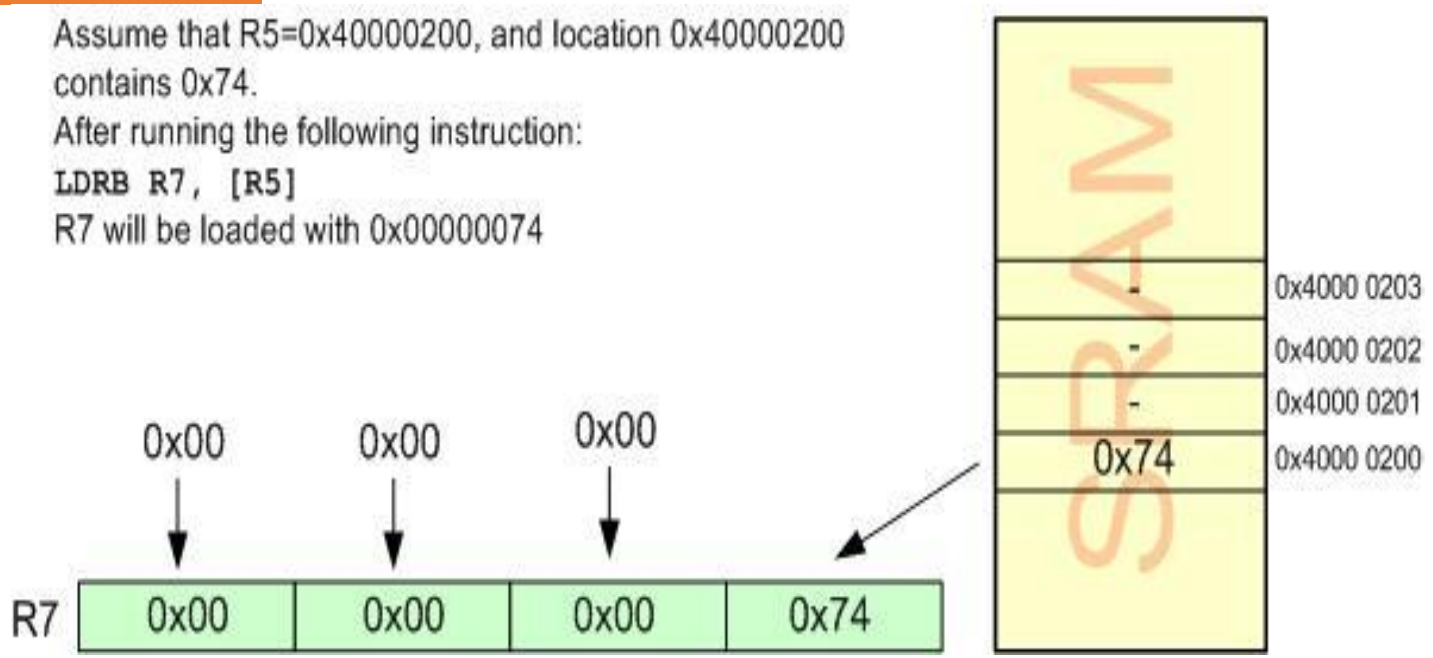
STR Rx,[Rd] instruction
;store register Rx into locations pointed to by Rd



Executing the STR Instruction

LDRB Rd, [Rx] instruction

- The LDRB instruction tells the CPU to load (copy) one byte from an address pointed to by Rx into the lower byte of Rd. After this instruction is executed, the lower byte of Rd will have the same value as memory location pointed to by Rx. It must be noted that the unused portion (the upper 24 bits) of the Rd register will be all zeros, as shown in Figure 2-7.



executing the LDRB Instruction

```
LDRB    Rd, [Rx]        ;load Rd with the contents of the location
                        ; pointed to by Rx register.
```

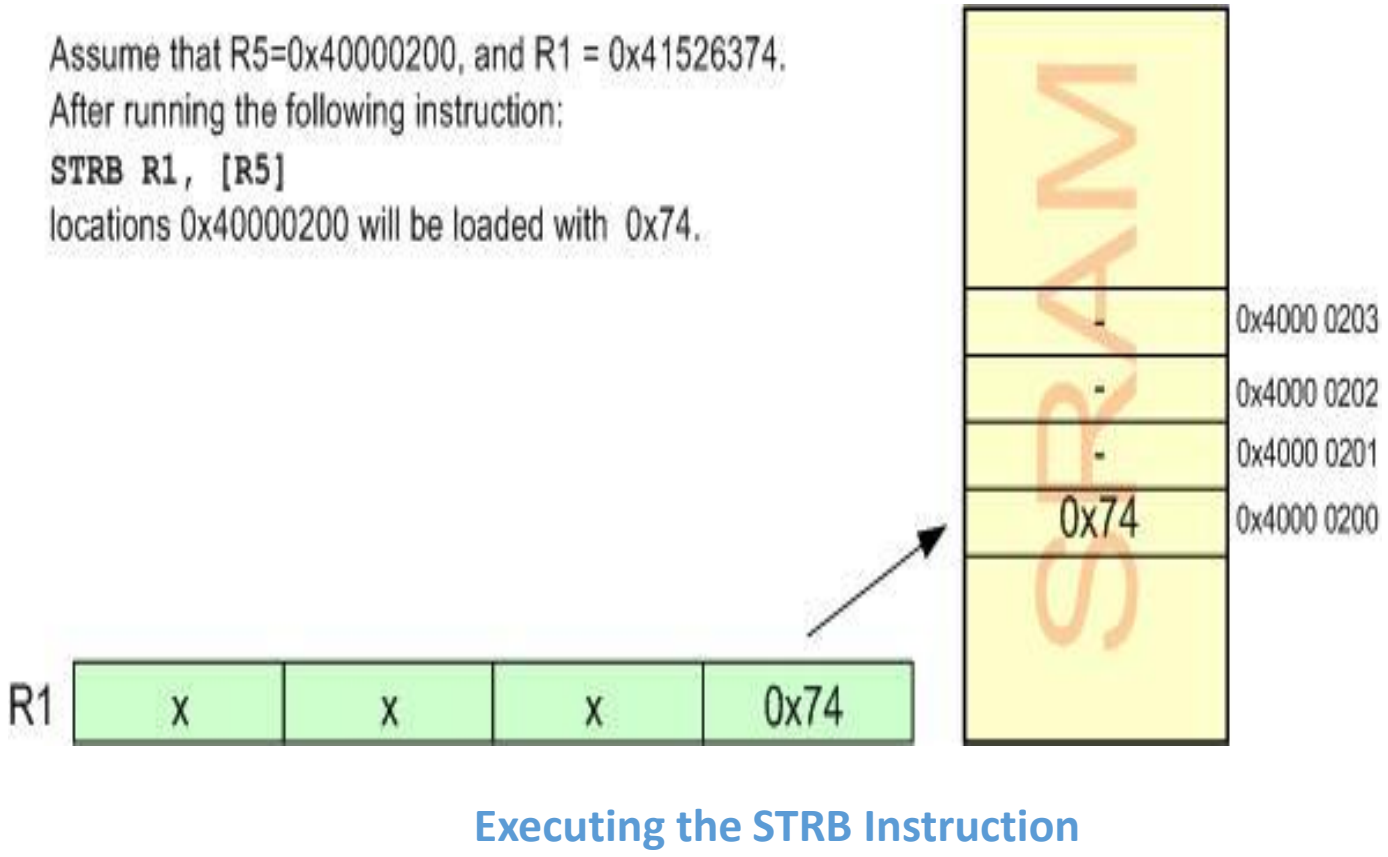
Data Size	Bits	Decimal	Hexadecimal	Load instruction used
Byte	8	0 – 255	0 - 0xFF	LDRB
Half-word	16	0 – 65535	0 - 0xFFFF	LDRH
Word	32	$0 – 2^{32}-1$	0 - 0xFFFFFFFF	LDR

STRB Rx,[Rd] instruction

- The STRB instruction tells the CPU to store (copy) the byte value in Rx to an address location pointed to by the Rd register. After this instruction is executed, the memory locations pointed to by the Rd will have the same byte as the lower byte of the Rx, as shown in Figure 2-8.

```
STRB    Rx, [Rd]        ;store the byte in register Rx into
                        ;location pointed to by Rd
```

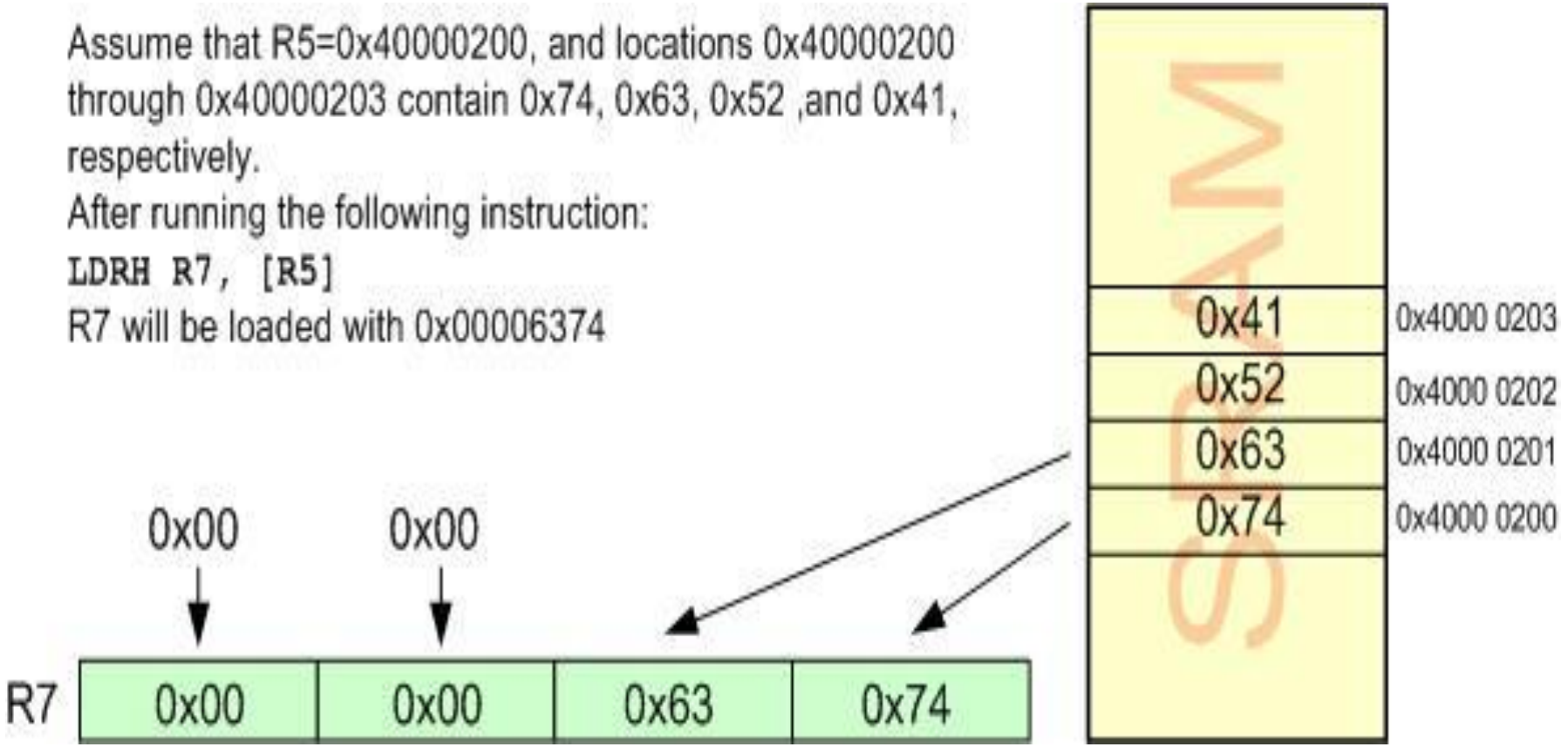
```
;assume R5 = 0x40000100
MOV R1,#0x55 ;R1 = 55 (in hex)
STRB R1,[R5] ;copy R1 location pointed to by R5
```



Executing the STRB Instruction

LDRH Rd, [Rx] instruction

;load Rd with the half-word (16-bit or 2 bytes) pointed to by Rx register



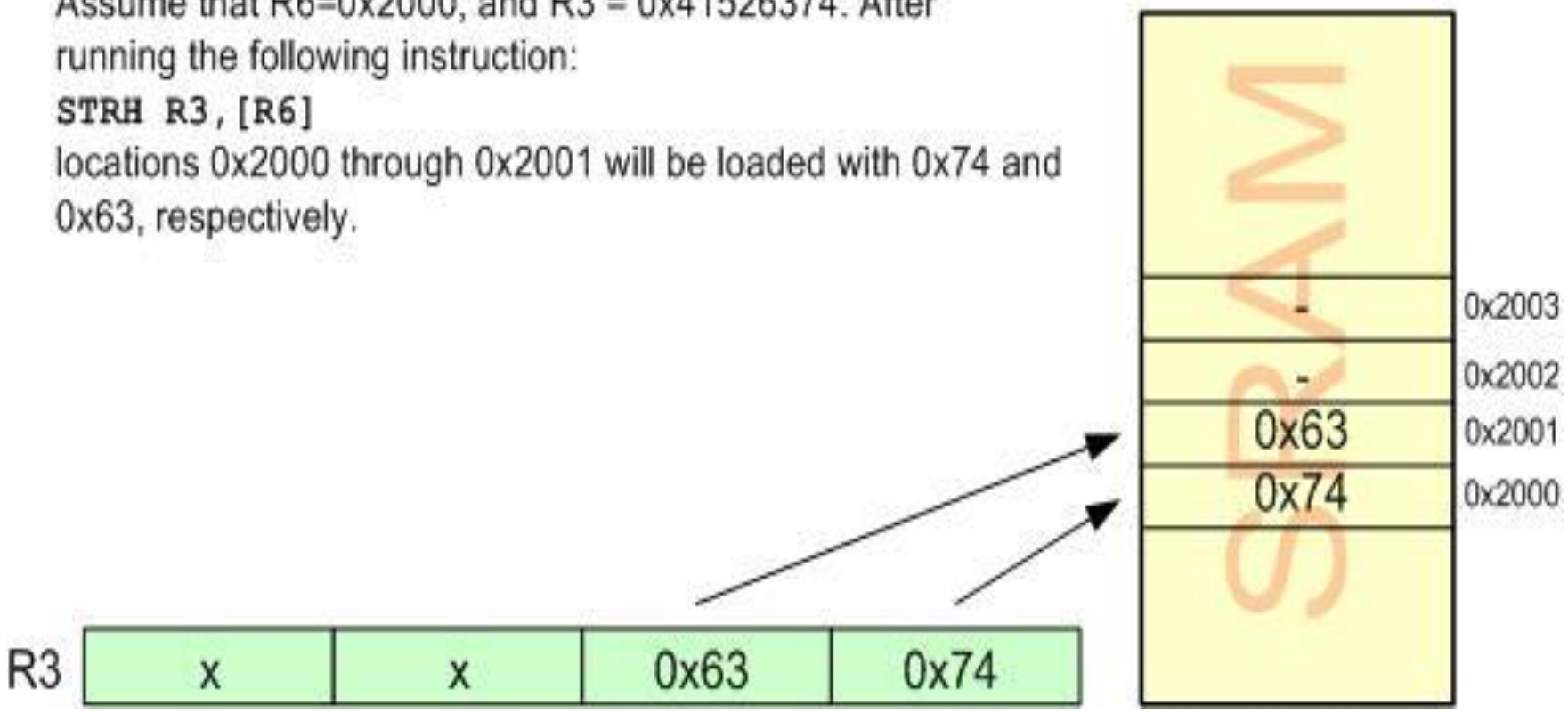
Executing the LDRH Instruction

STRH Rx,[Rd] instruction
;store half-word (2-byte) in register Rx into locations pointed to by Rd

Assume that R6=0x2000, and R3 = 0x41526374. After running the following instruction:

```
STRH R3,[R6]
```

locations 0x2000 through 0x2001 will be loaded with 0x74 and 0x63, respectively.



Executing the STRH Instruction

ARM Addressing Modes

- The CPU can access operands (data) in various ways, called addressing modes. The number of addressing modes is determined when the microprocessor is designed and cannot be changed. Using advanced addressing modes the accessing of different data types and data structures (e.g. arrays, pointers, classes) .
- Some of the simple ARM addressing modes are:
    1. register
    2. immediate
    3. register indirect (indexed addressing mode)
    4. PC relative Addressing mode

Register addressing mode

- The register addressing mode involves the use of registers to hold the data to be manipulated. Memory is not accessed when this addressing mode is executed; therefore, it is relatively fast.

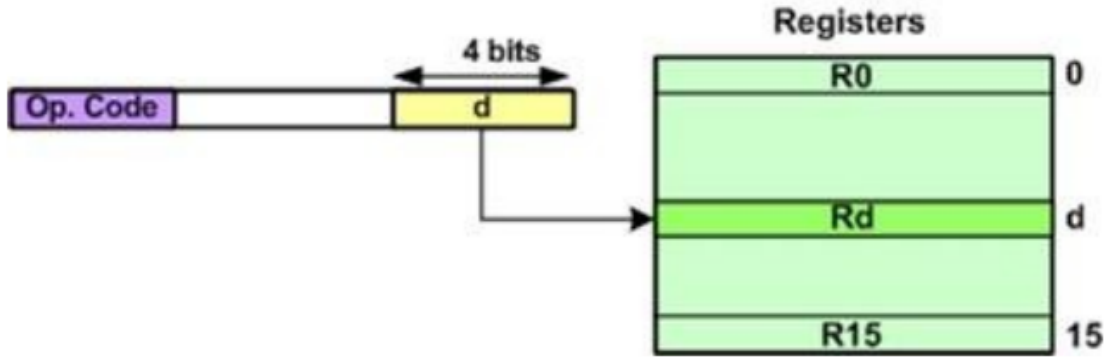


Figure 2- 24: Register Addressing Mode

Examples of register addressing mode are as follow:

```
MOV    R6,R2          ;copy the contents of R2 into R6
ADD    R1,R1,R3       ;add the contents of R3 to contents of R1
SUB    R7,R7,R2       ;subtract R2 from R7
```

Immediate addressing mode

- In the immediate addressing mode, the source operand is a constant. In immediate addressing mode, as the name implies, when the instruction is assembled, the operand comes immediately after the opcode. For this reason, this addressing mode executes quickly.

```
MOV    R9,#0x25          ;move 0x25 into R9
MOV    R3,#62            ;load the decimal value 62 into R3
ADD    R6,R6,#0x40       ;add 0x40 to R6
```

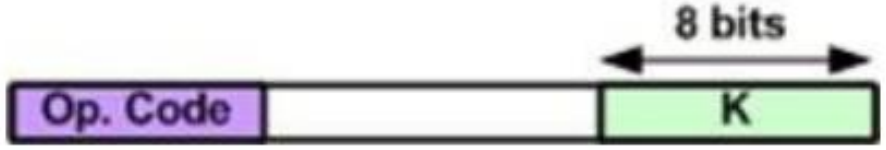


Figure 2- 25: Immediate Addressing Mode

Register Indirect Addressing Mode (Indexed addressing mode)

MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

- In the register indirect addressing mode, the address of the memory location where the operand resides is held by a register

```
STR      R5,[R6]          ;move R5 into the memory location
;pointed to by R6
LDR      R10,[R3]         ;move into R10 the contents of the
                           ;memory location pointed to by R3.
```

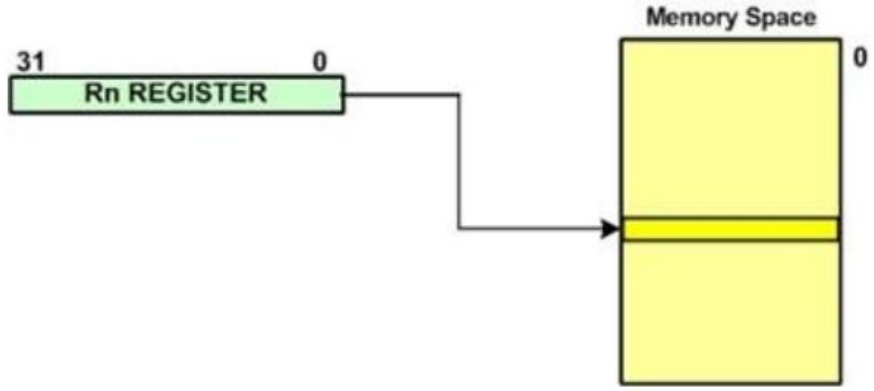


Figure 2- 26: Register Indirect Addressing Mode

ARM- Advanced Indexed addressing mode

MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

- In the indexed addressing mode, a register is used as a pointer to the data location. The ARM provides three indexed addressing modes. These modes are: preindex, preindex with write back, and post index.

Indexed Addressing Mode	Syntax	Pointing Location in Memory	Rm Value After Execution
Preindex	LDR Rd,[Rm,#k]	Rm+#k	Rm
Preindex with WB*	LDR Rd,[Rm,#k]!	Rm+#k	Rm + #k
Postindex	LDR Rd,[Rm],#k	Rm	Rm + #k

**WB means Writeback*

*** Rd and Rm are any of registers and #k is a signed 12-bit immediate value between -4095 and +4095*

Table 6-2: Indexed Addressing in ARM

Offset	Syntax	Pointing Location
Fixed value	LDR Rd,[Rm,#k]	Rm+#k
Shifted register	LDR Rd,[Rm,Rn, <shift>]	Rm+(Rn shifted <shift>)

** Rn and Rm are any of registers and #k is a signed 12-bit immediate value between -4095 and +4095*

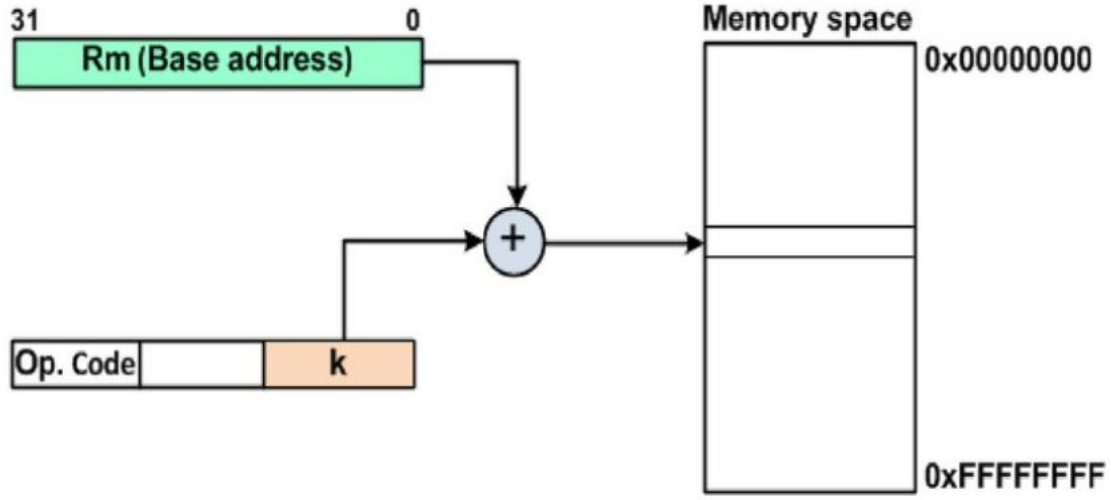
*** <shift> is any of shifts studied in Chapter3 like LSL#2*

Table 6-3: Offset of Fixed Value vs. Offset of Shifted Register

Preindexed addressing mode with fixed offset

- In this addressing mode, a register and a positive or negative immediate value are used as a pointer to the data location. The value of register does not change after instruction is executed. This addressing mode can be used with STR, STRB, STRH, LDR, LDRB, and LDRH

- STR Rd, [Rm, #k]
- LDR Rd, [Rm, #k]

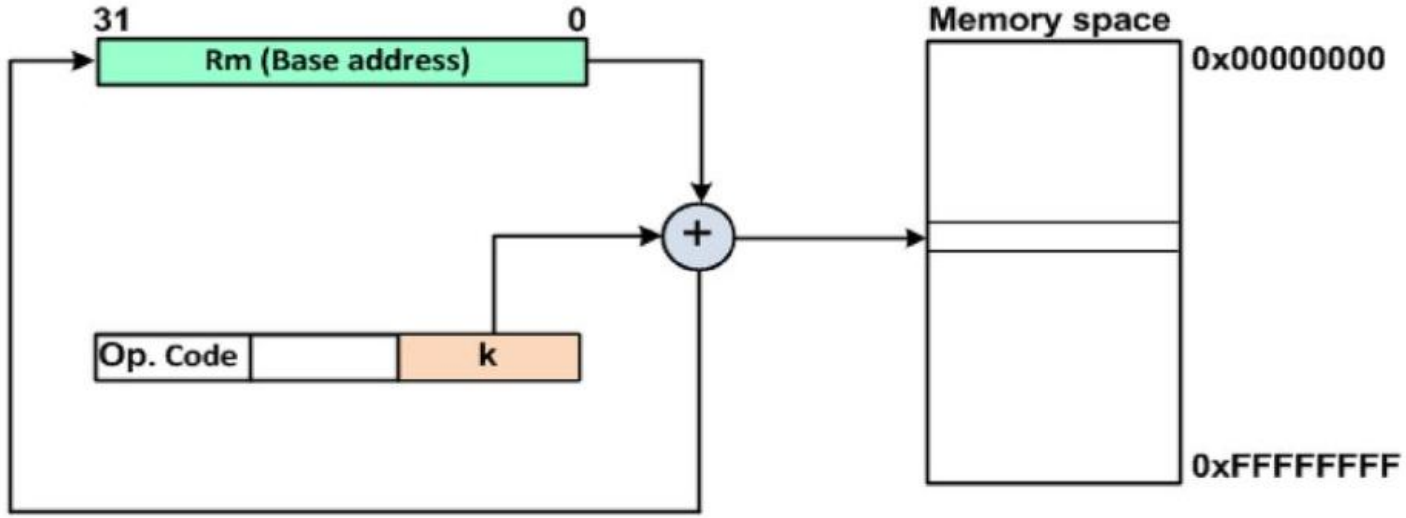


LDR R5, [R2, #4] @ R5 = contents of location R2+4

Preindexed addressing mode with write-back and fixed offset

This addressing mode is like preindexed addressing mode with fixed offset except that the calculated pointer is written back to the pointing register. We put ! after the instruction to tell the assembler to enable writeback in the instruction.

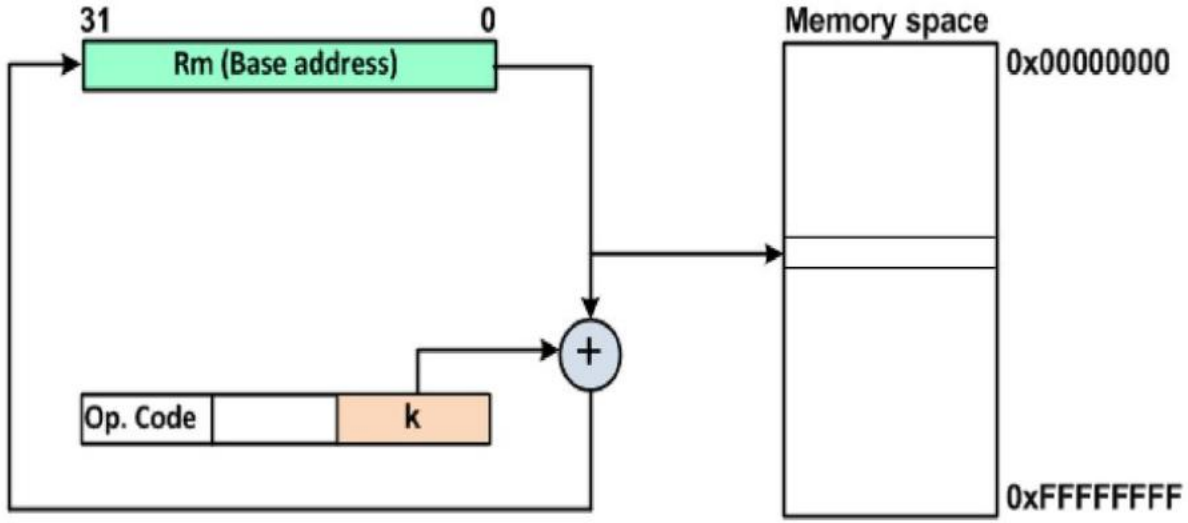
- LDR Rd, [Rm, k]!
- STR Rd, [Rm, k]!



Postindexed addressing mode with fixed offset

- This addressing mode is like preindexed addressing mode with writeback and fixed offset except that the instruction is executed on the location that Rn is pointing to regardless of offset value. After running the instruction, the new value of the pointer is calculated and written back to the pointing register

- LDR Rd, [Rm], #k
- STR Rd, [Rm], #k

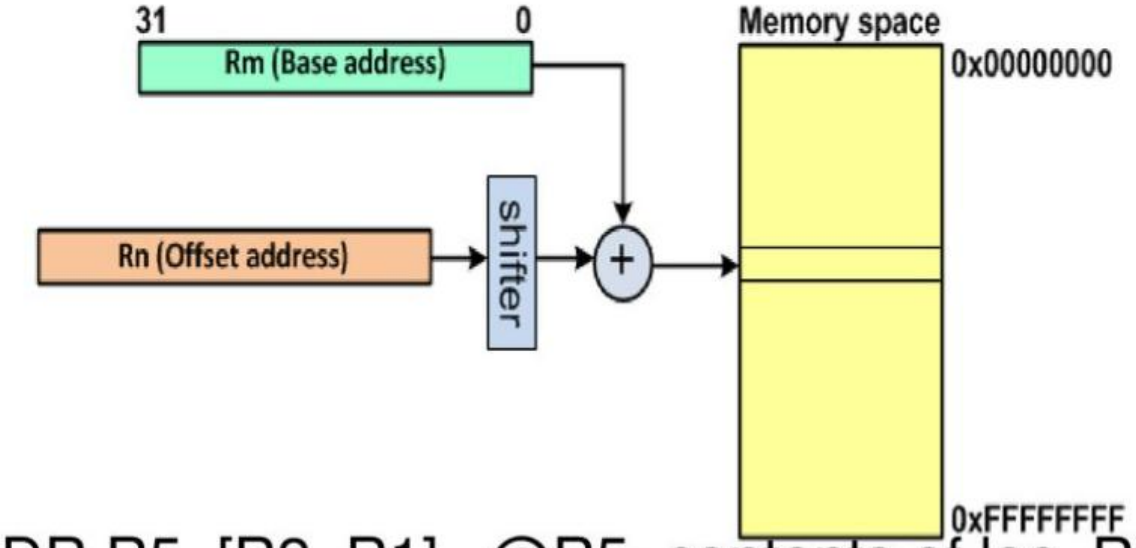


Refer Examples from 6.8 to 6.13 from textbook 1

Preindexed address mode with offset of a shifted register

- LDR Rd, [Rm, Rn]
- STR Rs, [Rm, Rn]

- LDR Rd, [Rm, Rn,shift]
- STR Rs, [Rm, Rn,shift]



- LDR R5, [R2, R1] @R5=contents of loc. R2+R1
- LDR R6, [R2, R1, LSL #2] @ R6= [R2+(R1<<2)]

Scaled register postindex

The following instructions are some examples of scaled register postindex in load and store instructions:

```
STR       R1,[R2],R3,LSL #2
;store R1 at location R2 of memory and write back
                    ;R2 + (R3 × 4) to R2.
LDR       R1,[R2],R3,LSL #2
;load location R2 of memory to R1 and write back
                    ;R2 + (R3 × 4) to R2.
```

PC Relative Addressing mode

- In indexed addressing modes, any registers including the PC (R15) register can be used as the pointer register.
  - For example, the following instruction reads the contents of memory location PC+4: LDR
    - LDR R0,[PC,offset] EA= Address of current instruction + 8 + offset

      LDR R0,[PC,#4]
- In this way, the data which has a known distance from the current executing line can be accessed.
- The PC register points 8 bytes (2 instructions) ahead of executing instruction.
- As a result, "LDR R0,[PC,#4]" accesses a memory location whose address is 4+8 bytes ahead of the current instruction.
- For instance, **if "LDR R0,[PC,#4]" is located in address 0x10 the effective address is: 0x10 + 8 + 4 = 0x1C.**

Implementing the ADR Directive (LDR Rn,=value)

- The ADR directive uses the PC relative addressing mode to load registers. For example see the following program:

```
    AREA    LOOKUP_EXAMPLE,READONLY,CODE
    ENTRY
    ADR     R2,OUR_FIXED_DATA   ;point to OUR_FIXED_DATA
    LDRB    R0,[R2]             ;load R0 with the contents
    ;of memory pointed to by R2
    ADD     R1,R1,R0            ;add R0 to R1
HERE    B       HERE            ;stay here forever
OUR_FIXED_DATA
    DCB     0x55,0x33,1,2,3,4,5,6
    DCD     0x23222120,0x30
    DCW     0x4540,0x50
    END
```

At compile time, the ADR is replaced with "ADD R2,PC,#0x08". Since the instruction is in address 0x00, the instruction accesses location 0 + 8 + 0x08 = 0x10. As shown in the Figure, 0x10 is the address of OUR_FIXED_DATA

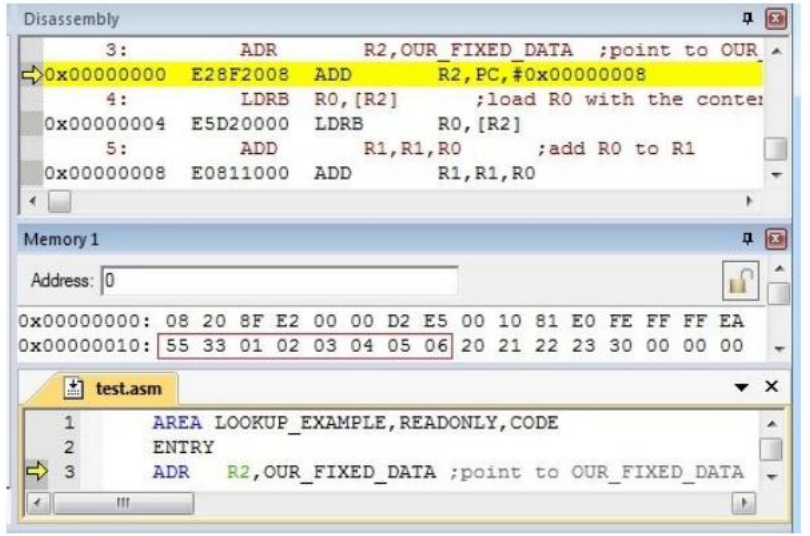


Figure 6- 11: Memory Dump for ADR Instruction

Implementing the LDR Directive

MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

- To implement the LDR directive, assembler stores the value as a fixed data in program memory and then accesses it using the LDR instruction and the PC relative addressing mode.

Program 6-3: LDR Directive

```
AREA    EXAMPLE,READONLY,CODE
ENTRY
LDR     R0,=0x12345678
LDR     R1,=0x86427531
ADD     R2,R0,R1
H1      B       H1
END
```

For example, 0x12345678 is stored in memory locations 0x10–0x13, and the LDR directive is replaced with LDR R0,[PC,#0x08]. Since PC=0, the LDR R0, =0x1234567 is located at address 0x000000. Now we have 0+8+8=16=0x10.

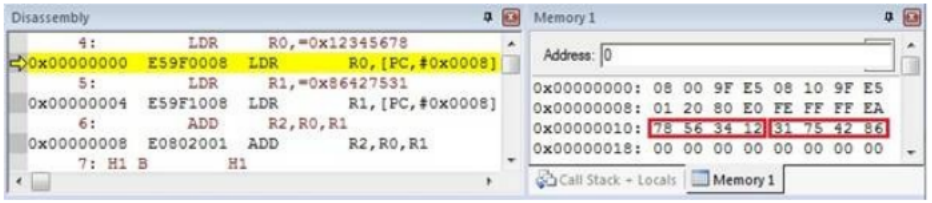


Figure 6- 12: Memory Dump for LDR Instruction

ADR Directive is more optimized compared to LDR

2.9.1 Von-Neumann Architecture and Harvard Architecture

- The Von Neumann Architecture is named after the mathematician and early computer scientist John Von Neumann. Von Neumann machines have shared signals and memory for code and data. Thus, the program can be easily modified by itself since it is stored in read-write memory.
- Fig. 2.9.1 shows Von Neumann Architecture.

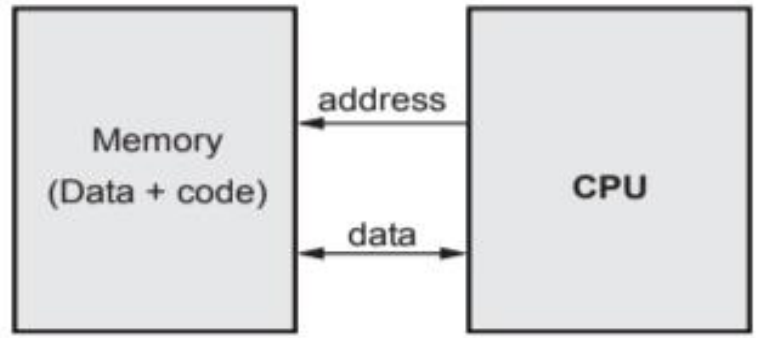


Fig. 2.9.1 Von Neumann architecture

- The CPU contains many internal registers that store values used internally. The Program Counter (PC) one of the internal register which holds the address in memory of an instruction.
- The CPU fetches the instruction from memory, decodes the instruction and executes it.
- PC does not directly determine what the machine does next, but only indirectly by pointing to an instruction in memory.
- The name **Harvard Architecture** comes from the Harvard Mark I relay-based computer. The most obvious characteristic of the Harvard Architecture is that it has physically separate signals and storage for code and data memory.

- It is possible to access program memory and data memory simultaneously. Typically, code (or program) memory is read-only and data memory is read-write. Therefore, it is impossible for program contents to be modified by the program itself.
- Fig. 2.9.2 shows **Harvard Architecture**.
- Most DSPs use Harvard architecture for streaming data.

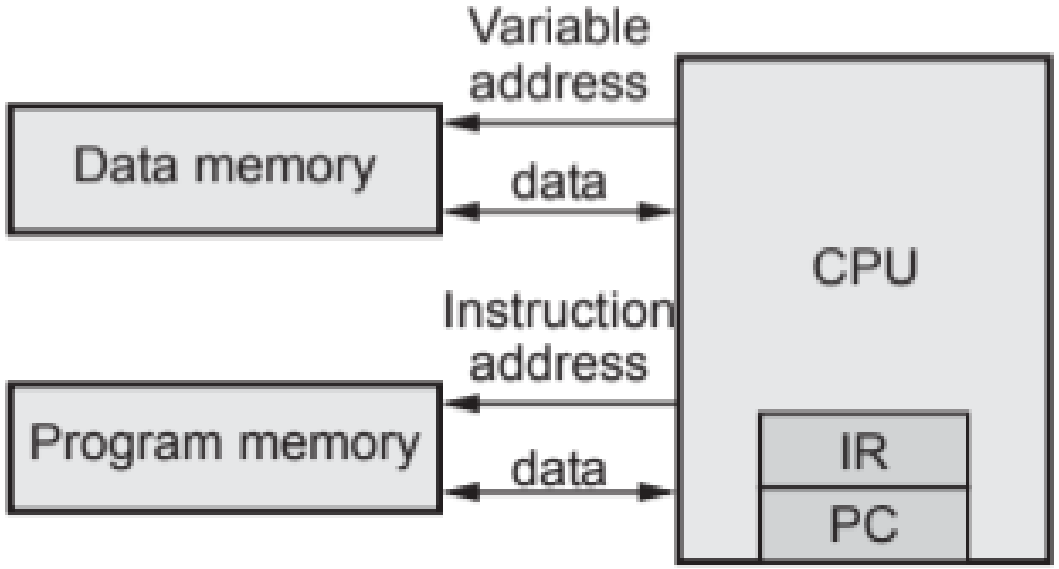
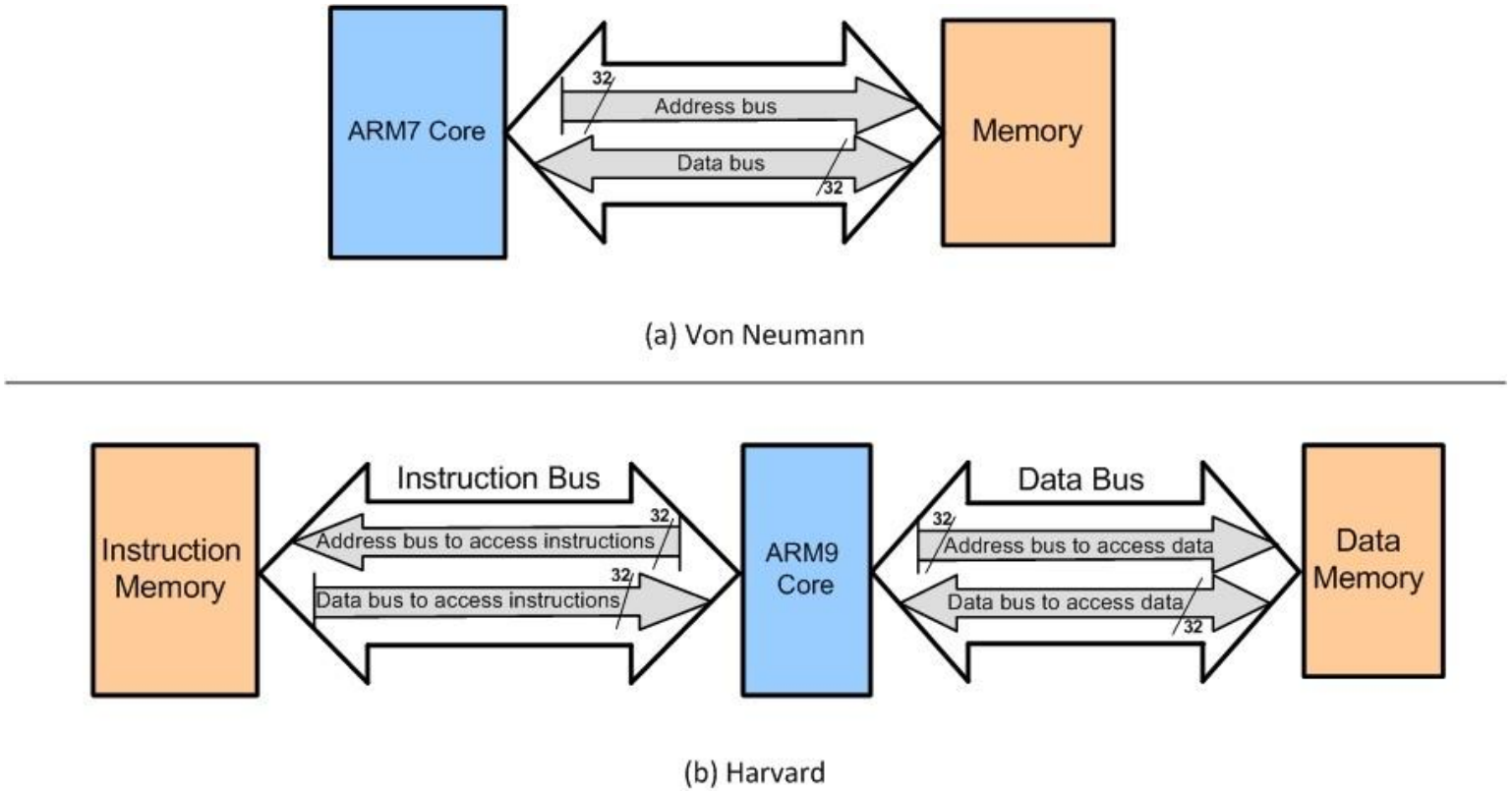


Fig. 2.9.2 Harvard architecture

Sr. No.	Von-Neumann architecture	Harvard architecture
1.	In Von-Neumann structure you can explore program memory and make any operation on data memory by the mean of CPU.	In Harvard the memory is split in two parts and the CPU can't explore or make operations on such parts.
2.	Von Neumann machines have shared signals and memory for code and data.	It is possible to access program memory and data memory simultaneously.
3.	Single shared bus for instruction and data fetching.	Separate buses for instruction and data fetching.
4.	Program can be easily modified by itself since it is stored in read-write memory.	Program can not be modified easily.

Harvard and von Neumann architectures in the ARM



In von Neumann, there are no separate buses for code and data memory. In Harvard, there are separate buses for code and data memory.

When the CPU wants to execute the “LDR Rd,[Rx]” instruction, it puts Rx on the address bus of the system, and receives data through the
data bus. For example, to execute “LDR R2,[R5]”, assuming that R5 = 0x40000200, the CPU puts the value of R5 on the address bus. The Memory puts the contents of location 0x40000200 on the data
bus. The CPU gets the contents of location 0x40000200 through the data bus and brings into CPU and puts it in R2.
The “STR Rx,[Rd]” instruction is executed similarly. The CPU puts Rd on the address bus and the contents of Rx on the data bus. The memory location whose address is on the address bus receives the contents of data bus.

Addressing modes

Register to register (Register direct) MOV R0, R1

Absolute (Direct) LDR R0, MEM

Literal (Immediate) MOV R0, #15

ADD R1, R2, #12

Indexed, base (Register indirect) LDR R0, [R1]

Pre-indexed, base with displacement (Register indirect with offset)

```
 LDR R0, [R1, #4]            ;Load R0 from [R1+4]
 LDR R0, [R1, #-4]
```

Pre-indexed with autoindexing (Register indirect with pre-incrementing)

```
 LDR R0, [R1, #4]!          ; Load R0 from [R1+4], R1 = R1 +4
```

Post-indexing with autoindexed (Register indirect with post-increment)

```
 LDR R0, [R1], #4                ; Load R0 from [R1], R1 = R1 +4
```

Double Reg indirect (Register indirect indexed)

```
 LDR R0, [R1, R2]         ; Load R0 from [R1 + R2]
```

Double Reg indirect with scaling (Register indirect indexed with scaling)

```
 LDR R0, [R1, R2, LSL #2]
```

Program counter relative LDR R0, [PC, #offset]

RISC Features

- Fixed Instruction Size
- Large no of registers(less memory operations)
- Smaller inst. Set
- Single clock cycle inst. Execution
- Hardwiring
- Load/Store Architecture

Review Question

1. *Explain some basic addressing modes of ARM.*

2.11 RISC Architecture in ARM

- ARM designers have used three ways to increase the processing power of the CPU :

1. Increase in the clock frequency of the chip.
2. Use of Harvard architecture, which means using separate buses for the code and the data memory.

3. Change in the internal architecture of the CPU and use what is called RISC architecture.

2.11.1 Features of RISC Implemented by ARM

- **Fixed instruction size :** It helps the CPU to decode instruction quickly.
- **Large number of CPU registers :** It avoids the need for large amounts of interactions with memory. So we can use data items stored in registers multiple times without multiple memory accesses. This is advantageous since memory accesses are costly.
- **Small instruction set :** The limited number of instructions leads to two disadvantages.
  - It makes the job of assembly language programmers much more tedious and difficult. For this reason, RISC is used more commonly in high-level language environments such as the C programming language rather than assembly language environments.
  - The limited number of instructions in RISC leads to a large program size and hence program use of more memory.
- **Single-cycle execution of most of the instructions :** It executes more than 99 % of instructions in a single cycle.

- **Separate buses for data and code :** This means that it uses Harvard architecture. Thus, it is possible to access program memory and data memory simultaneously.
- **Limited number of addressing modes and microcodes :** It requires fewer transistors and hence is implemented using hardwire method instead of microprogramming.
- **Load/store architecture :** Load/store architecture prevents RISC microprocessors from manipulating data present in memory. There is no direct way of doing arithmetic and logic operations between a register and the contents of external memory locations. All these instructions must be performed by first bringing both operands into the registers inside the CPU, then performing the arithmetic or logic operation, and then sending the result back to memory. In CISC processors, data present in memory can be manipulated. However, there might be a delay in accessing the data from external memory. Then the whole process would be stalled, preventing other instructions from proceeding in the pipeline.

2.11.2 Comparison between RISC and CISC

Sr. No.	Characteristics	RISC	CISC
1.	Instruction size	Fixed	Varies
2.	Instruction length	4 bytes	1, 2, 3 or 4 bytes
3.	Number of Instruction	Less	More
4.	Instruction decoding	Easy (quick) to decode	Serial (slow) to decode
5.	Instruction sementics	Almost always one simple operation	Varies from simple to complex ; possibly many dependent operations per instruction
6.	Addressing Modes	Complex addressing modes are synthesized in software.	Supports complex addressing modes.
7.	Instruction execution speed	Medium	Slow (depend on complexity of instruction)
8.	Instruction execution	By hardware. Simple instructions taking one cycle.	By microprogram. Complex instructions taking multiple cycles.
9.	Registers	Many, general purpose	Few, may be special purpose
10.	Memory references	Not combined with operations, i.e., load/store architecture	Combined with operations in many different types of instructions
11.	Hardware	Simple	Complicated
12.	Hardware design focus	Take the advantage of implementations with one pipeline and no microcode	Take the advantage of microcoded implementations
13.	Memory access	Rarely	Frequently
14.	Instruction format	Regular, consistent placement of fields	Field placement varies
15.	Pipelined	Highly pipelined.	Not pipelined or less pipelined.
16.	Conditional jump	Can be based on a bit anywhere in memory.	Conditional jump is usually based on status register bit.
17.	Compiler	Complicated	Simple
18.	Examples	ARM, 8051, ATMEL, AVR, etc.	Intel X86, Motorola 68000 series.