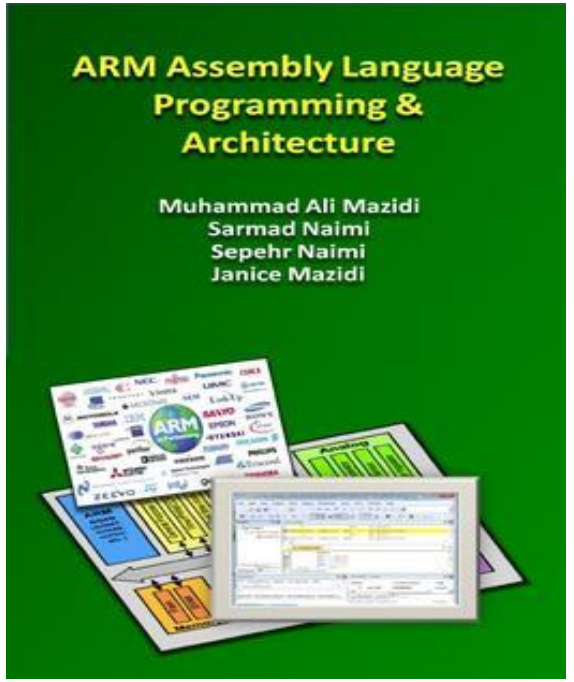
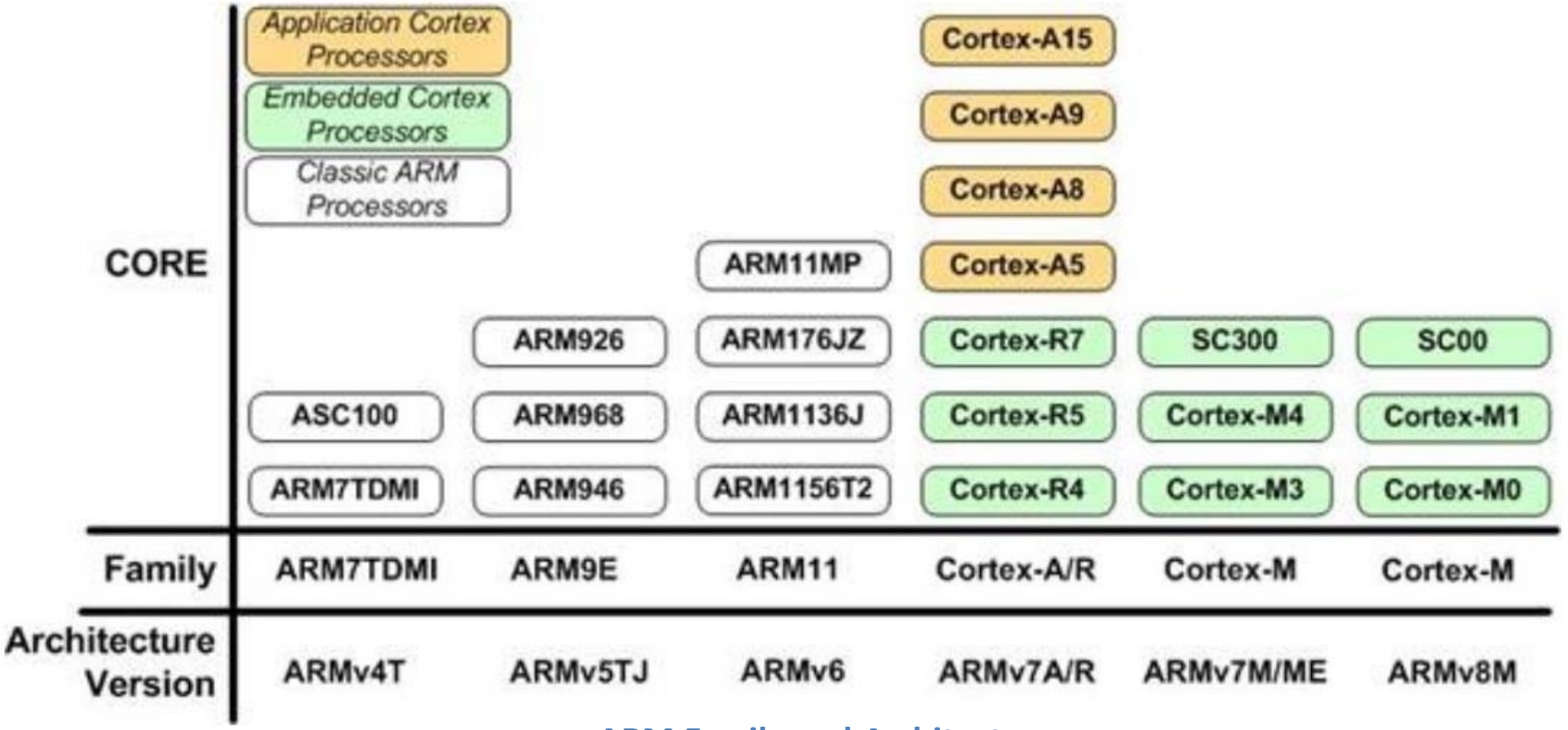
The History of ARM and Microcontrollers
Chapter 1

ARM Assembly Language Programming & Architecture



- 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)
- Unable to compete with x86
- That is when Apple Corp. got interested in using the ARM chip
- So, in 1990 as ARM (Advanced RISC Machines)Ltd was formed by the joint venture between Acorn Computers, Apple Computer (now Apple Inc.) and VLSI Technology.
- Apple invested the cash, VLSI Technology provided the tools, and Acorn provided the 12 engineers and with that Arm was born
- ARM does not manufactures any chip but sells the IP rights other silicon manufacturers and design houses.

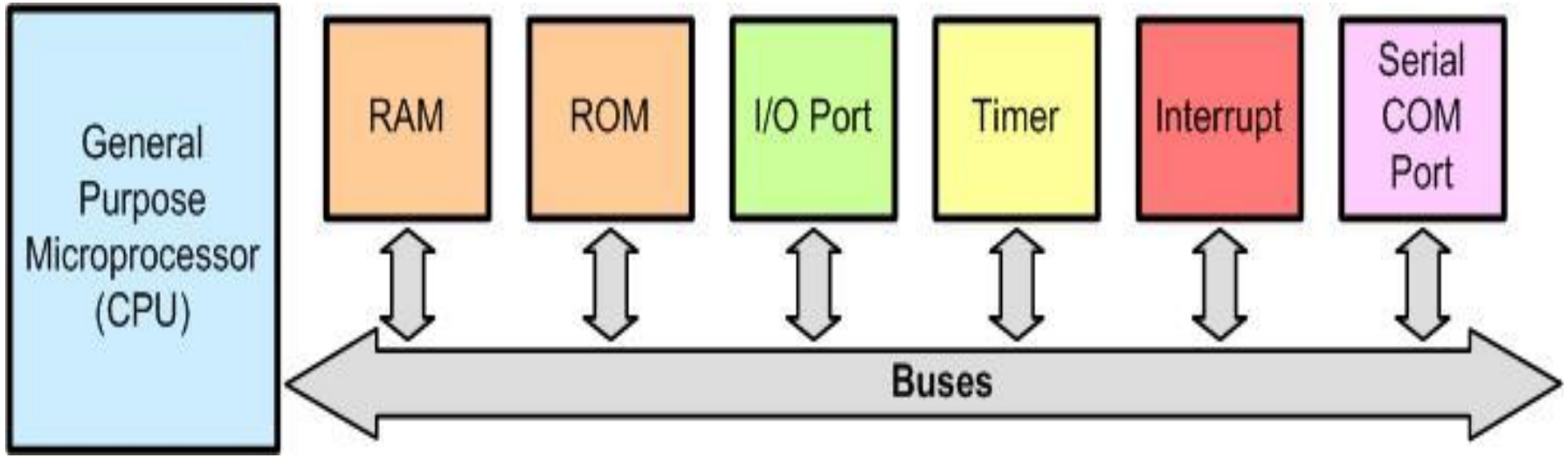
ARM family variations



ARM Family and Architecture

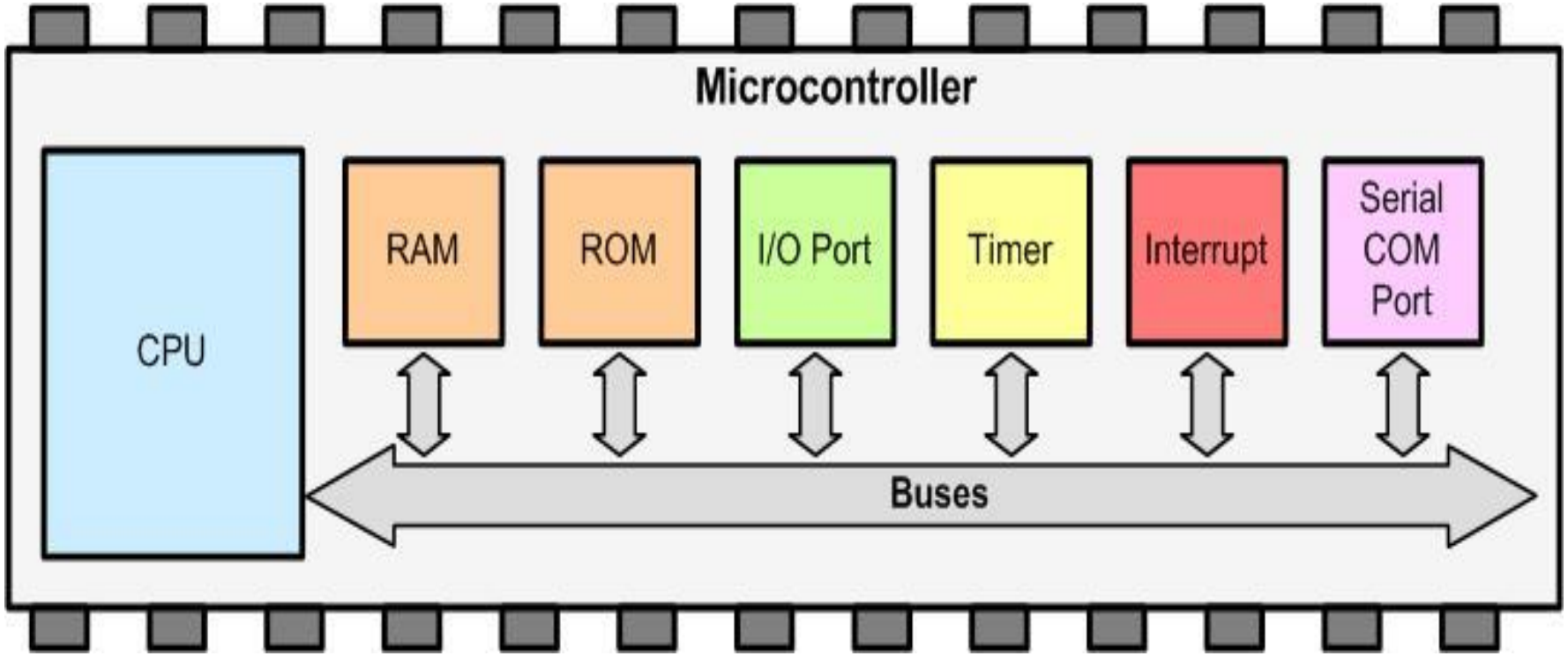
Note: we cannot use the terms ARM family and ARM architecture interchangeably

Microprocessors and Microcontrollers



A Computer Made by General Purpose Microprocessor

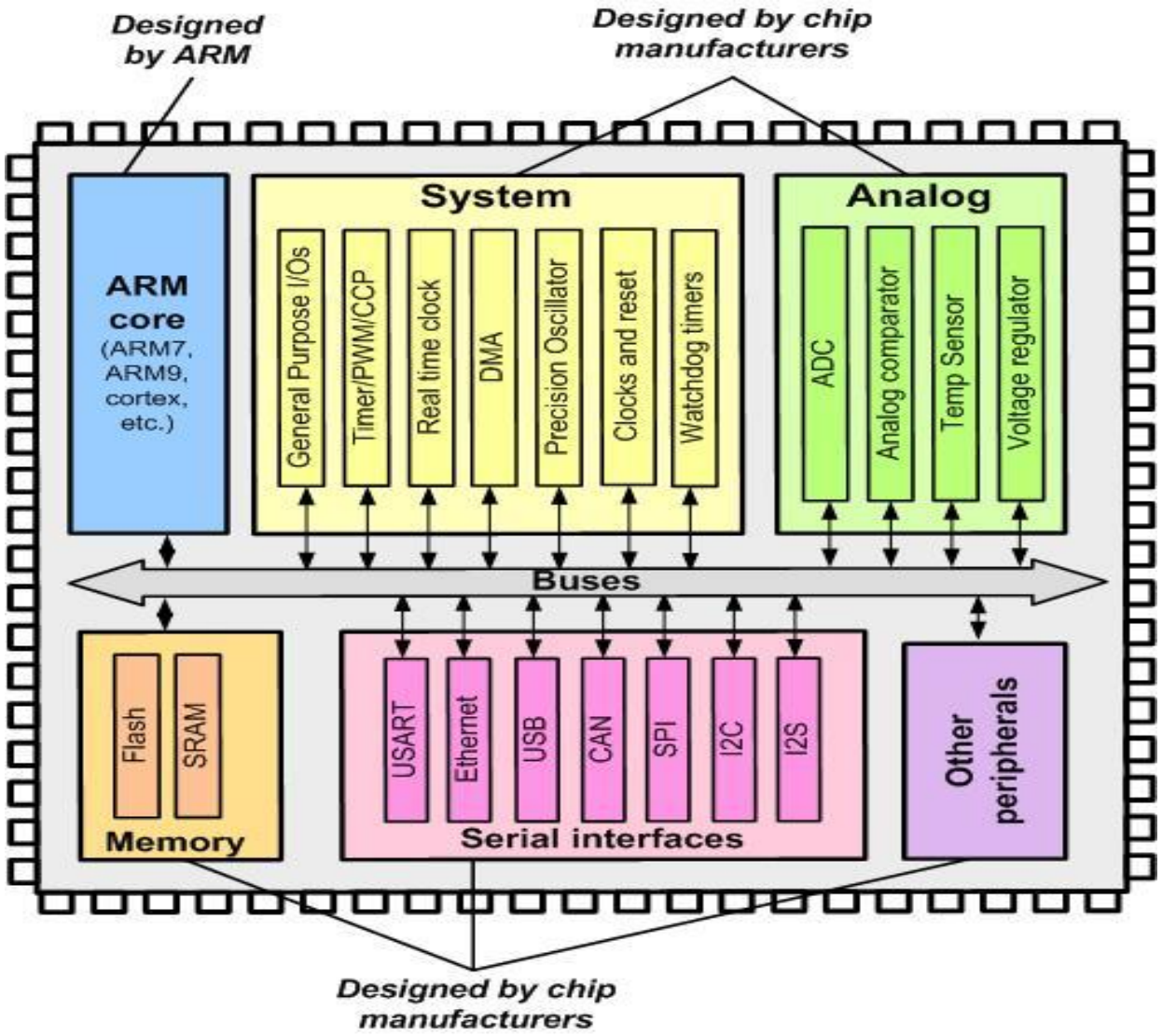
The microprocessors do not contain RAM, ROM, or I/O peripherals. As a result, they must be connected externally to RAM, ROM and I/O through buses



Simplified View of the Internal Parts of Microcontrollers (SOC)

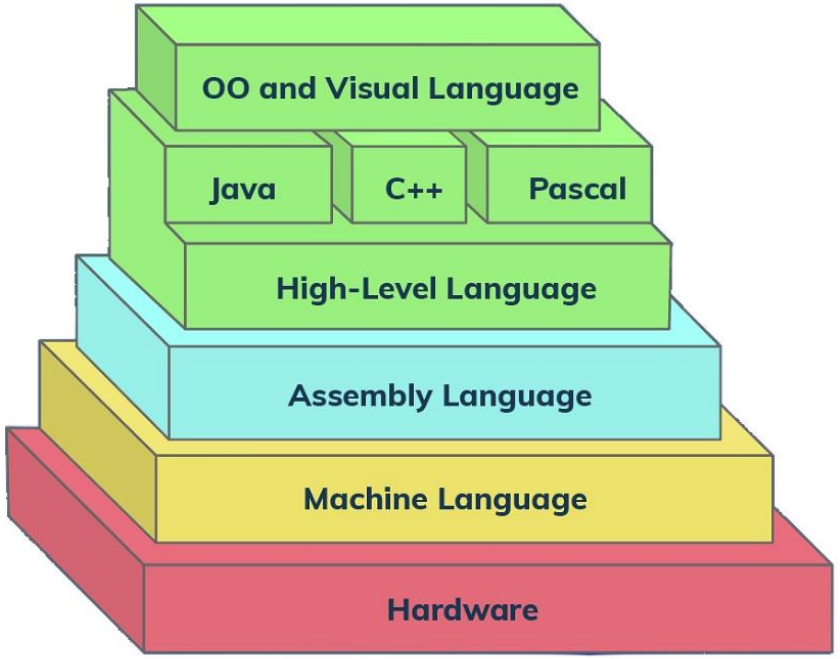
In microcontroller, CPU, RAM, ROM, and I/Os, are put together on a single IC chip and it is called *microcontroller*. SOC (System on Chip) and MCU (Micro Controller Unit) are other names used to refer to microcontrollers.

- ARM has defined the details of architecture, registers, instruction set, memory map, and timing of the ARM CPU. It 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 want.
- 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.



ARM Simplified Block Diagram

- A machine language contains sequences of 0's and 1's that a hardware can execute.
- Writing and understanding machine language is humanly impossible.
- When programmer want to dictate an instruction that a computer should perform, they use Assembly language.
- Assembly language is written in textual form.
- An assembler translates a file containing assembly language code into the corresponding machine language.



Hierarchy of Programming Languages

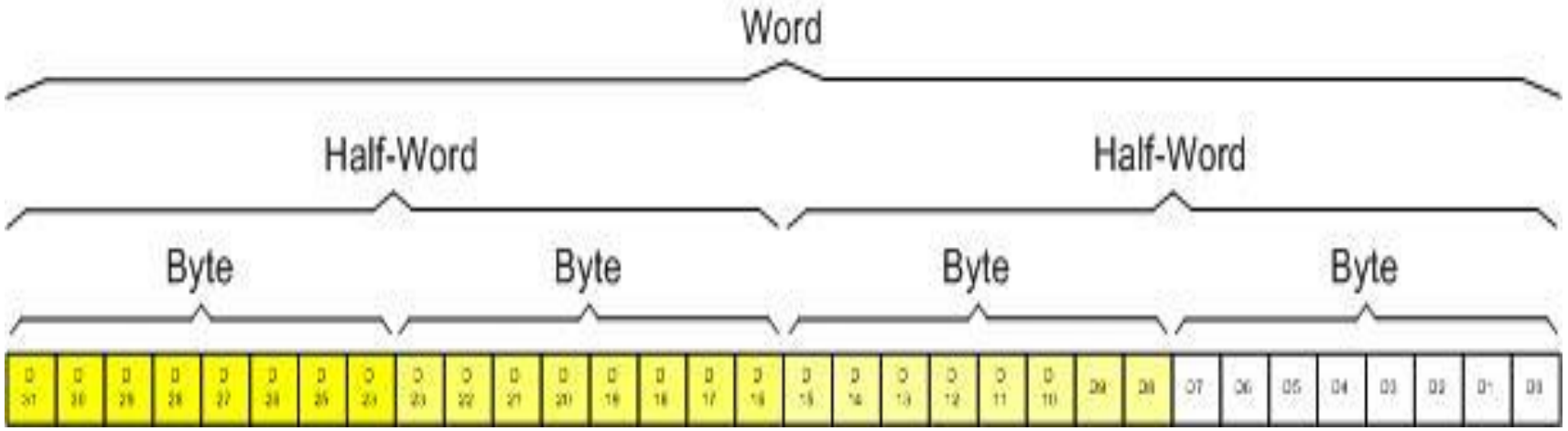
Assembly Language

- Assembly language is a low-level programming language for a programmable device.
- It is closest to the Machine language
- It is often specific to a particular computer architecture➔ There are multiples of Assembly languages

- Programmers who write codes at Higher Level may not be aware of the hardware beneath
- The knowledge of assembly code can be useful
  - For writing highly optimized code
  - While writing device driver programs
  - While writing portions of boot programs
  - OS development

ARM Architecture

- CPUs use registers to store data temporarily.

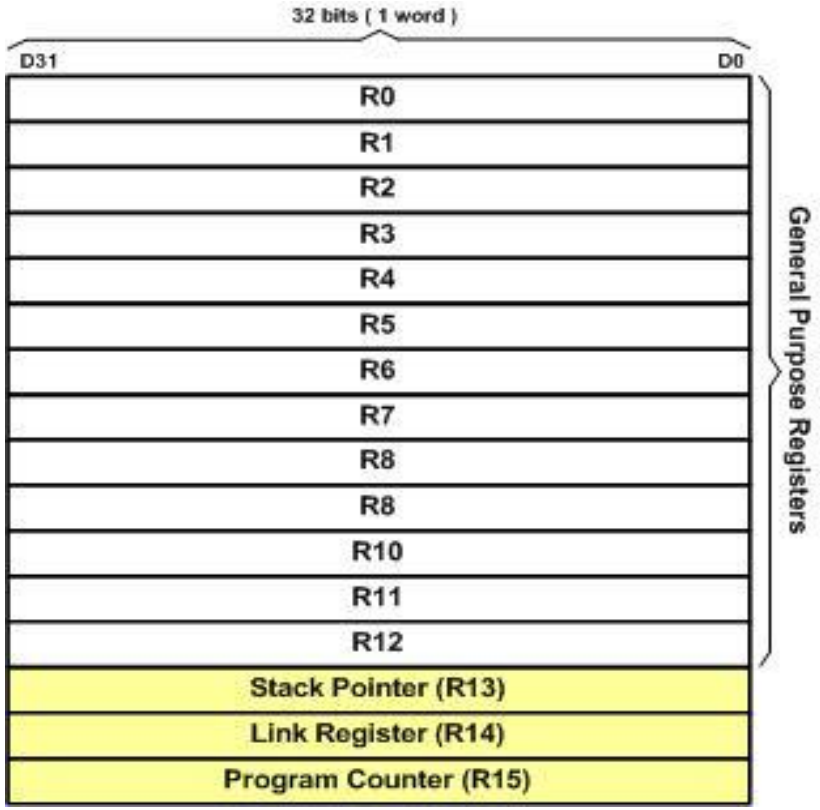


ARM Registers Data Size

ARM supports byte, half word (16 bit) and word (32 bit) data types.

The General Purpose Registers

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



ARM Registers

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

ARM Instruction Format

instruction destination,source1,source2

MOV instruction

```
MOV Rn,Op2        ;load Rn register with Op2 (Operand2)
Examples
MOV R2,#0x25      ;load R2 with 0x25 (R2 = 0x25)
MOV R5,R7         ;copy contents of R7 into R5 (R5 = R7)
```

Following points should be noted for immediate value

- We put # in front of every immediate value.
- If we want to present a number in hex, we put a 0x in front of it. If we put nothing in front of a number, it is in decimal. For example, in "MOV R1,#50", R1 is loaded with 50 in decimal, whereas in "MOV R1,#0x50", R1 is loaded with 50 in hex (80 in decimal).
- If values 0 to FF are moved into a 32-bit register, the rest of the bits are assumed to be all zeros. For example, in "MOV R1,#0x5" the result R1=00000000000000000000000000000101 in binary.
- Moving an immediate value larger than 255 (FF in hex) into the register will cause an error.

ADD instruction

```
ADD Rd,Rn,Op2      ;ADD Rn to Op2 and store the result in Rd
                   ;Op2 can be Immediate value #K (K is between 0 and 255)
                   ;or Register Rm
```

Example

```
MOV R1,#0x25       ;copy 0x25 into R1 (R1 = 0x25)
MOV R7,#0x34       ;copy 0x34 into R1 (R7 = 0x34)
ADD R5,R1,R7       ;add value R7 to R1 and put it in R5
                   ;(R5 = R1 + R7)
```

OR

```
MOV R1,#0x25       ;load (copy) 0x25 into R1 (R1 = 0x25)
ADD R5,R1,#0x34    ;add 0x34 to R1 and put it in R5
                   ;(R5 = R1 + 0x34)
```

SUB instruction

```
SUB Rd,Rn,Op2    ;Rd=Rn – Op2
```

Example

```
MOV R1,#0x34     ;load (copy) 0x34 into R1 (R1=0x34)
SUB R5,R1,#0x25  ;R5 = R1 – 0x25 (R1 = 0x34 – 0x25)
```

In ADD and SUB, Rn can be omitted if Rd and Rn are the same.

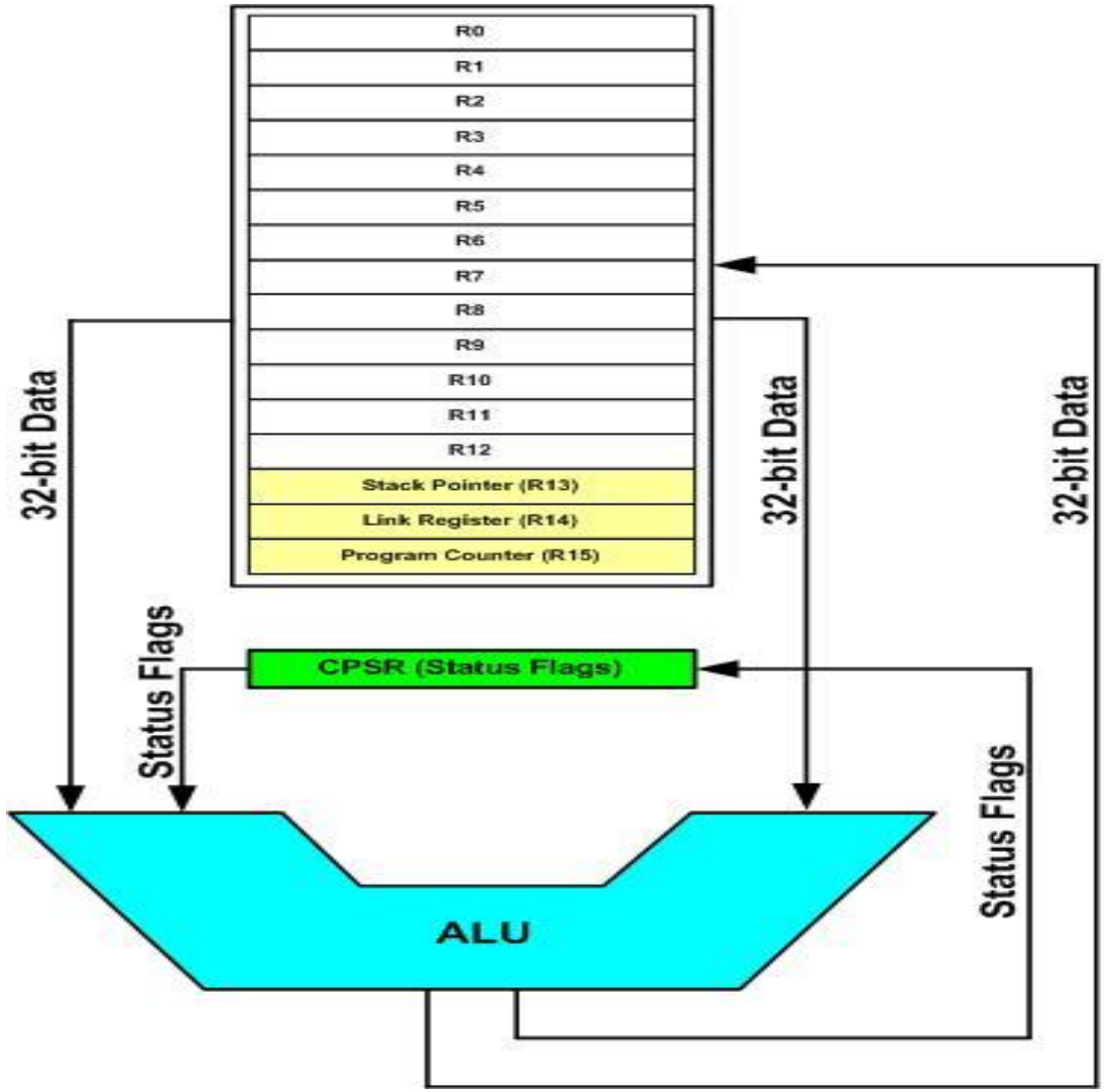
```
SUB R1,R1,#0x25 ;R1=R1-0x25
SUB R1,#0x25     ;R1=R1-0x25

SUB R1,R1,R2     ;R1=R1-R2
SUB R1,R2        R1=R1-R2

ADD R1,R1,#0x25          ;R1=R1+0x25
ADD R1,#0x25     ;R1=R1+0x25

ADD R1,R1,R2     ;R1=R1+R2
ADD R1,R2        ;R1=R1+R2
```

The Special Function Registers in ARM



ARM Registers and ALU

- R13, R14, R15, and CPSR (current program status register) ➔ *SFRs (special function registers)* and each one is dedicated to a specific function.
- R13 is stack pointer.
- R14 is designated as link register which holds the return address of the subroutine.
  - A subroutine is a block of code that performs a specific task based on some arguments and optionally returns a result.
- R15 is the program counter (PC). The PC holds 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.

Program Counter in the ARM

- The PC (program counter) points to the address of the next instruction to be executed.
- PC is of 32 bits ➔ 2^{32} = 4G byte (4GB) memory can be accessed by PC, i.e., 0x00000000 to 0xFFFFFFFF

Memory space allocation in the ARM

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.
3. **On-chip EEPROM:** A block of memory from 1K bytes to several thousand bytes is set aside for EEPROM memory.

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.

On-chip Memory Size for some ARM Chips

Company	Device	Flash (K Bytes)	RAM (K Bytes)	I/O Pins
Atmel	AT91SAM7X512	512	128	62
NXP	LPC2367	512	58	70
ST	STR750FV2	256	16	72
TI	TMS470R1A256	256	12	49
Freescale	MK10DX256VML7	256	64	74

Memory mapped I/O in the ARM

- In x86 had two distinct spaces: the I/O space and memory space. Here I/O ports are accessed using IN and OUT instructions, whereas the memory address space is accessed using the MOV instruction.
- In the ARM CPU we have only one space and it is memory space. And it accessed by both IO and memory. This mapping of the I/O ports to memory space is called memory mapped I/O.

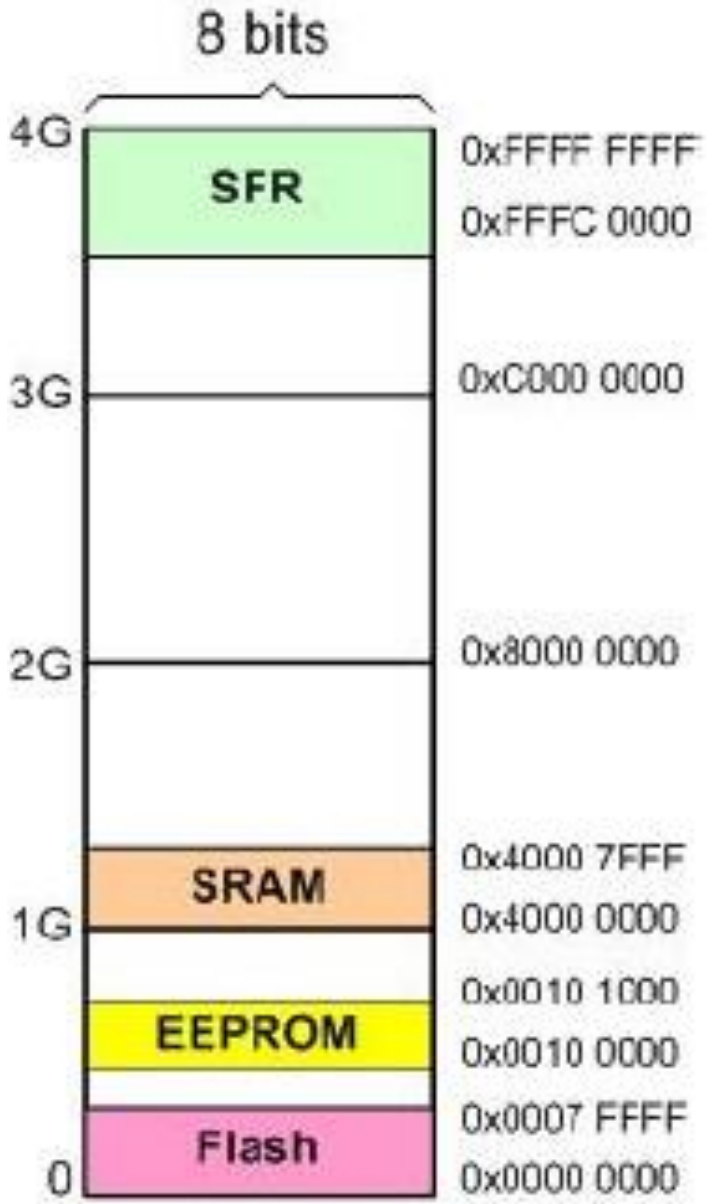
A given ARM chip has the following address assignments. Calculate the space and the amount of memory given to each section.
(a) Address range of 0x00100000 – 0x00100FFF for EEPROM
(b) Address range of 0x40000000 – 0x40007FFF for SRAM
(c) Address range of 0x00000000 – 0x0007FFFF for Flash
(d) Address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

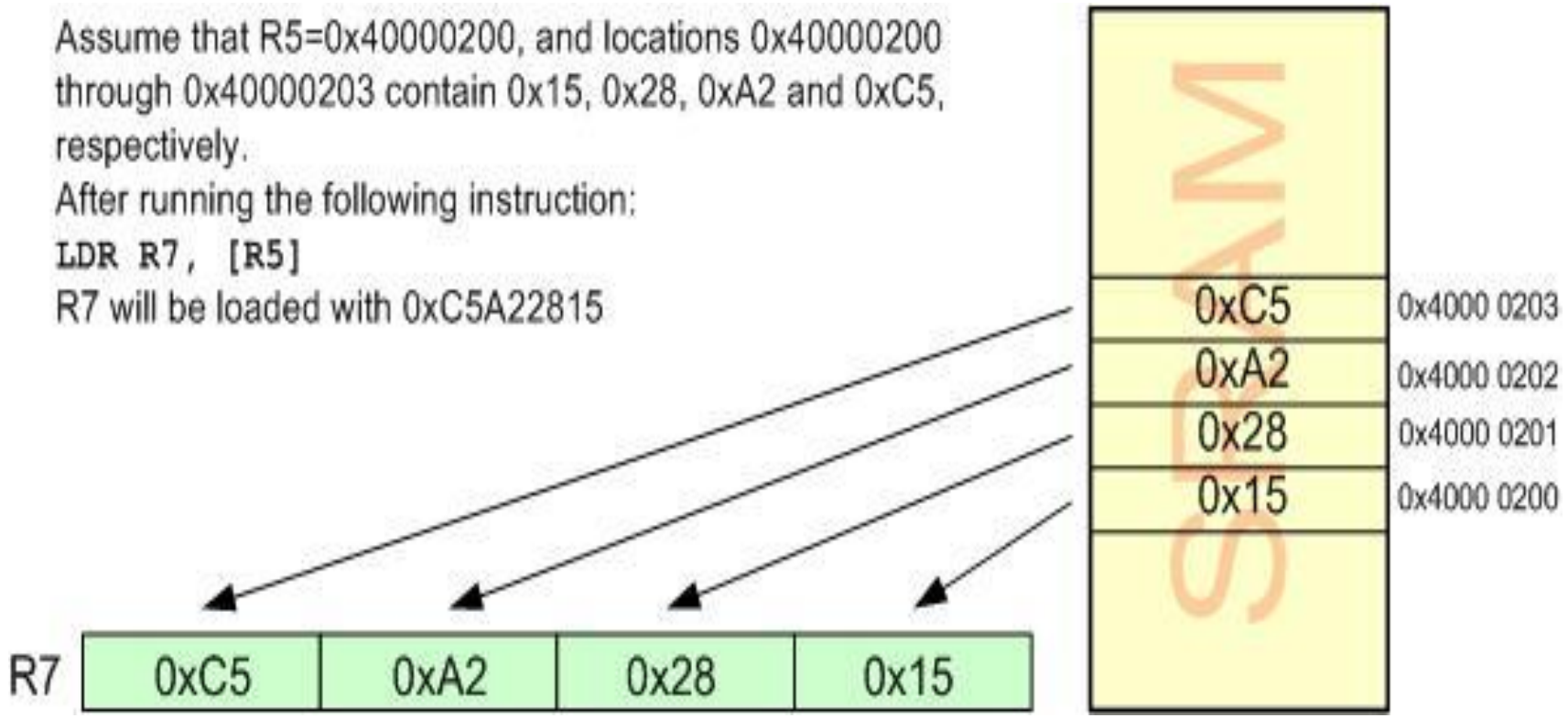


ARM Assembly Language Programming & Architecture by Mazidi, et al.

Load and Store Instructions in ARM

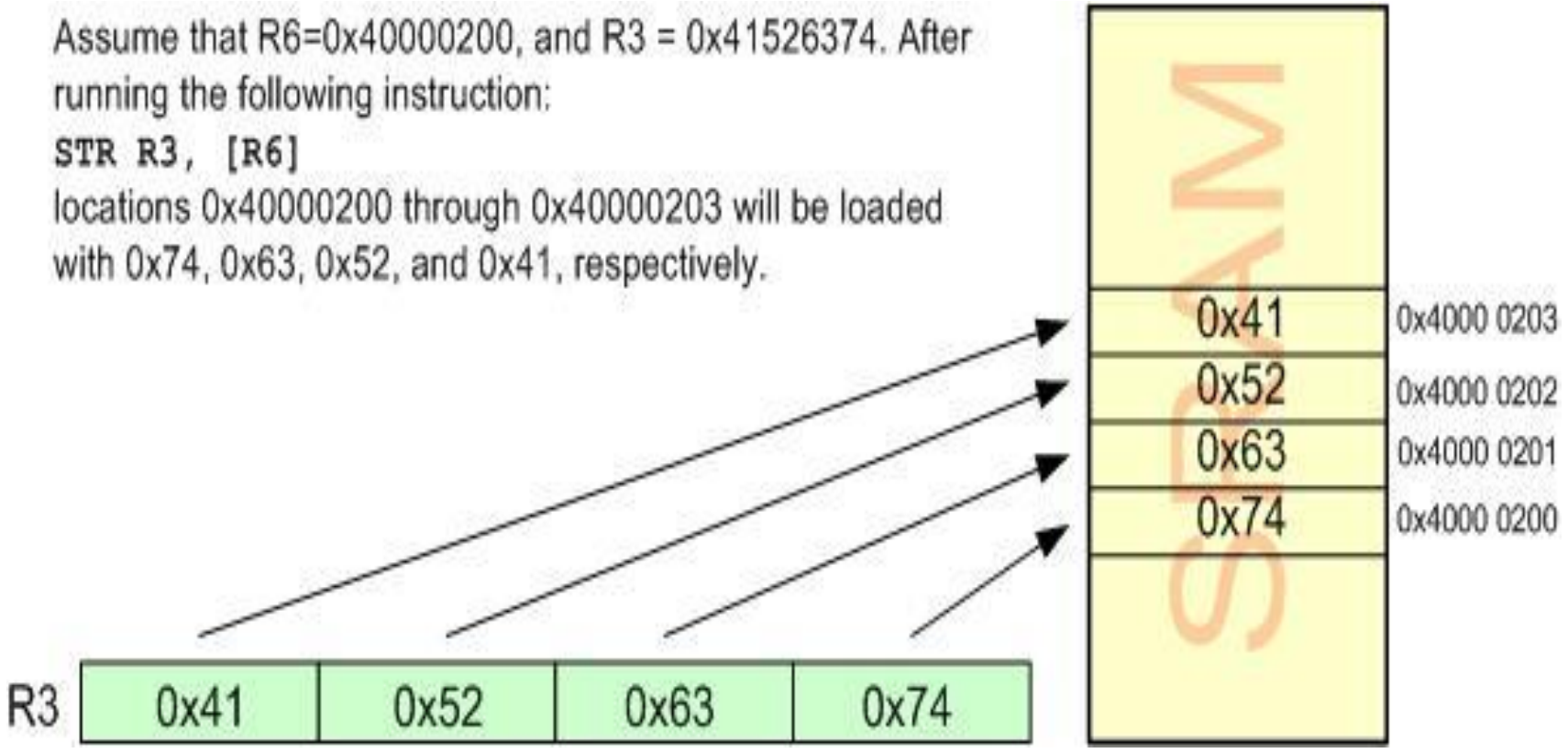
Every instruction of ARM is fixed at 32-bit. The fixed size instruction is one of the most important characteristics of RISC architecture.

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



Executing the LDR Instruction

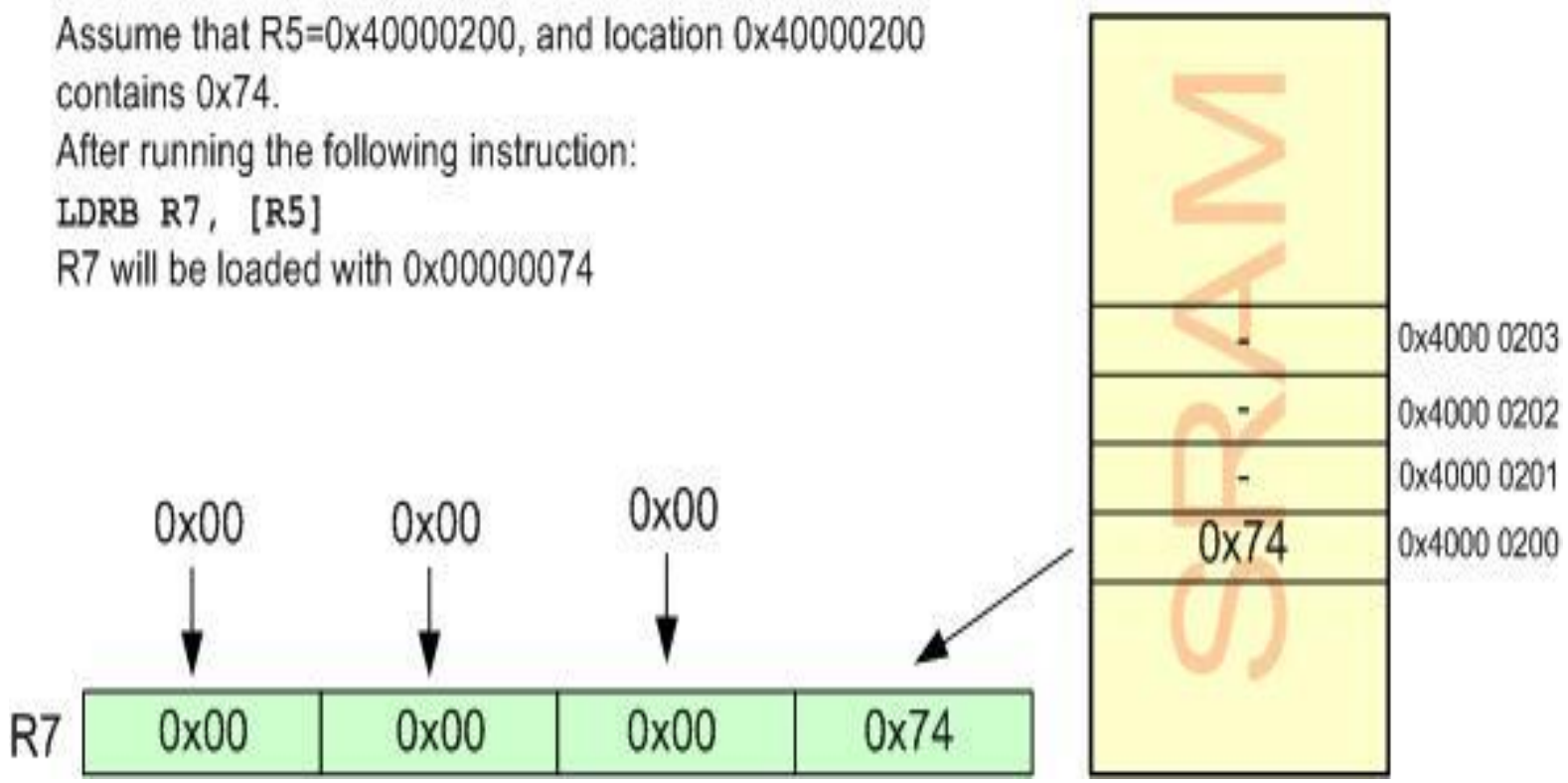
STR Rx,[Rd] instruction ;store register Rx into locations pointed to by Rd



Executing the STR Instruction

LDRB Rd, [Rx] instruction (LoaD Register Byte)

;load Rd with the contents of the location pointed to by Rx register. After this instruction is executed, the lower byte of Rd will have the same value as memory location pointed to by Rx. The upper 24 bits of the Rd register will be all zeros



executing the LDRB Instruction

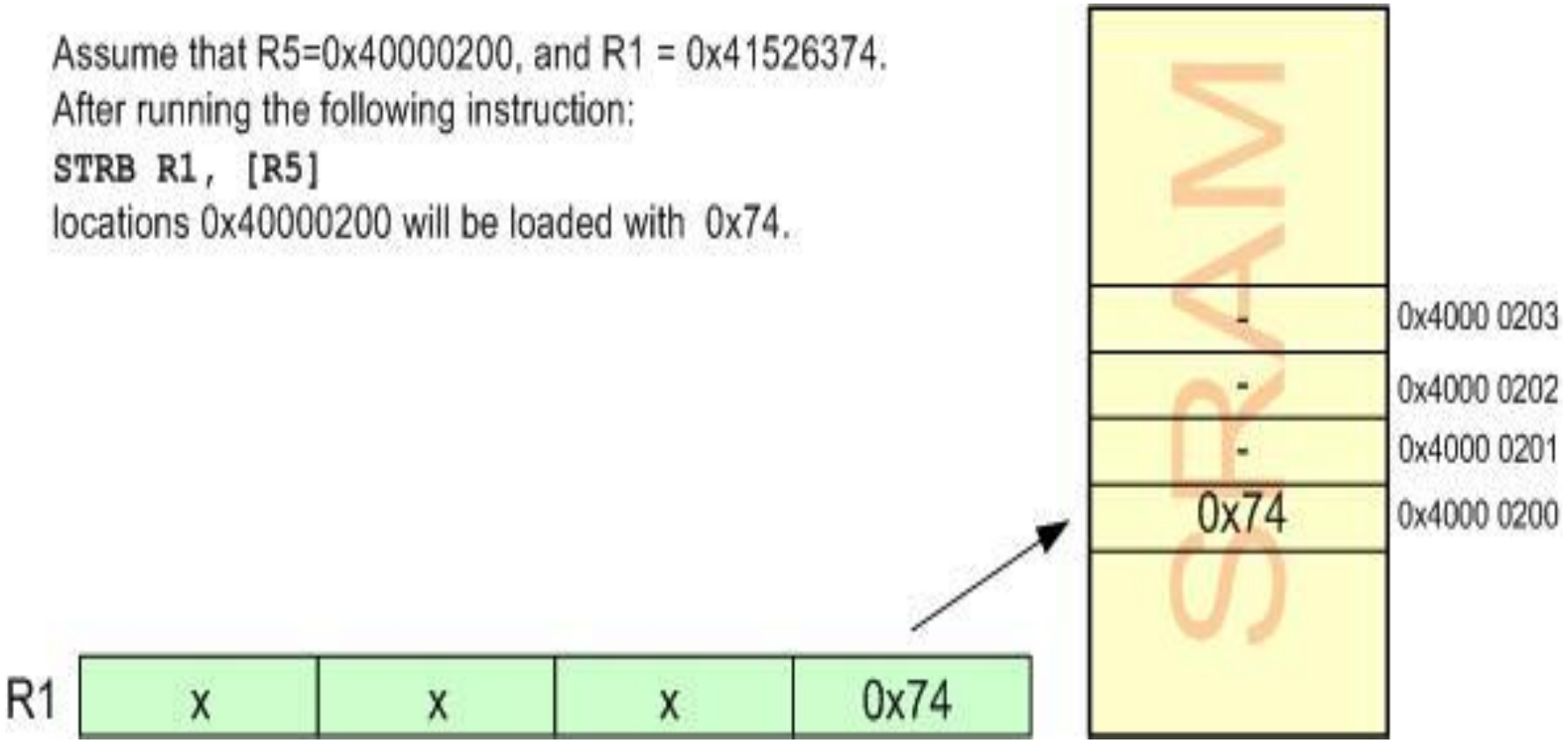
STRB Rx,[Rd] instruction

;store the byte in register Rx into location pointed to by Rd

Assume that R5=0x40000200, and R1 = 0x41526374.
After running the following instruction:

```
STRB R1, [R5]
```

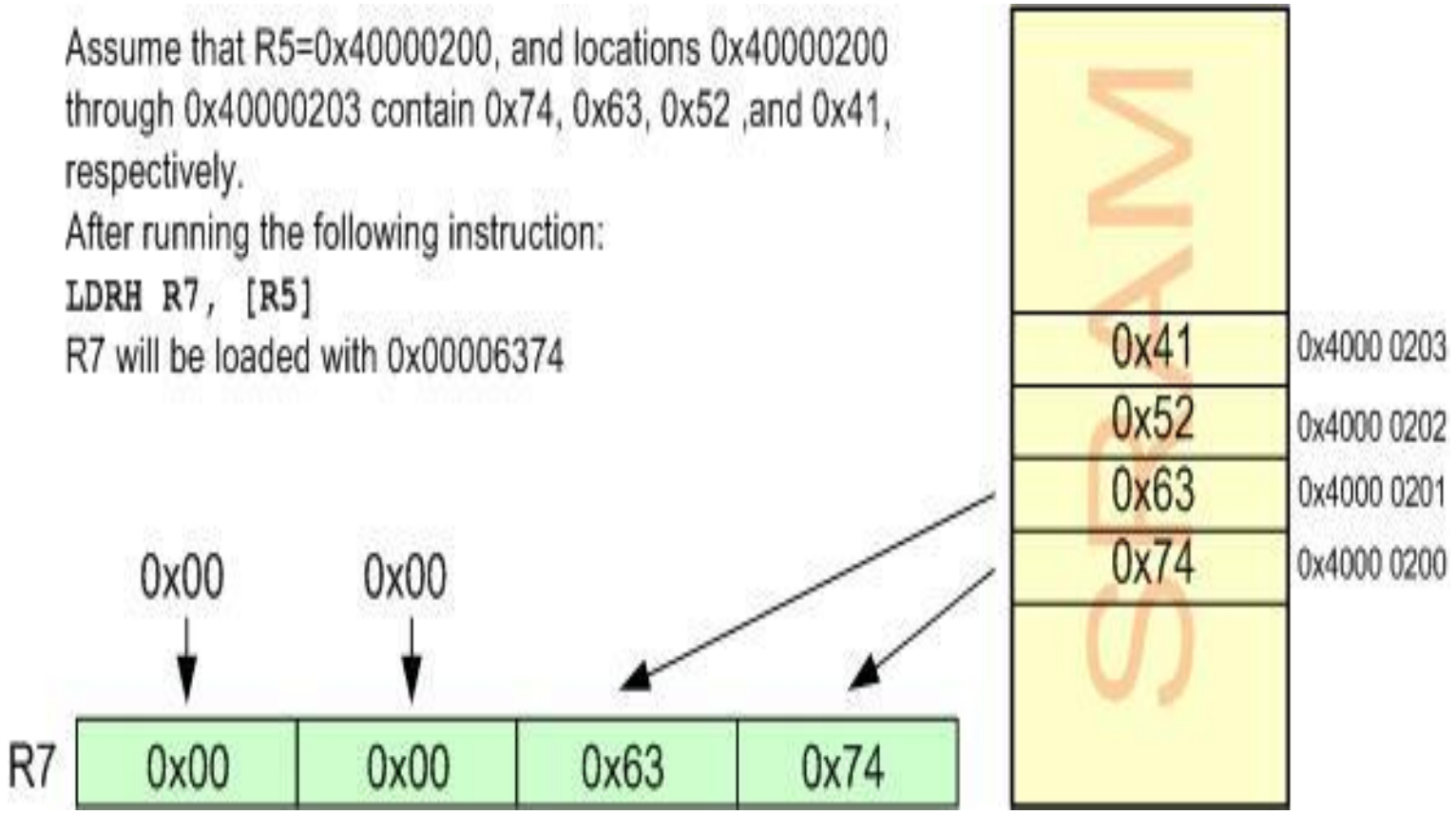
locations 0x40000200 will be loaded with 0x74.



Executing the STRB Instruction

LDRH Rd, [Rx] instruction (LoaD Register Halfword)

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

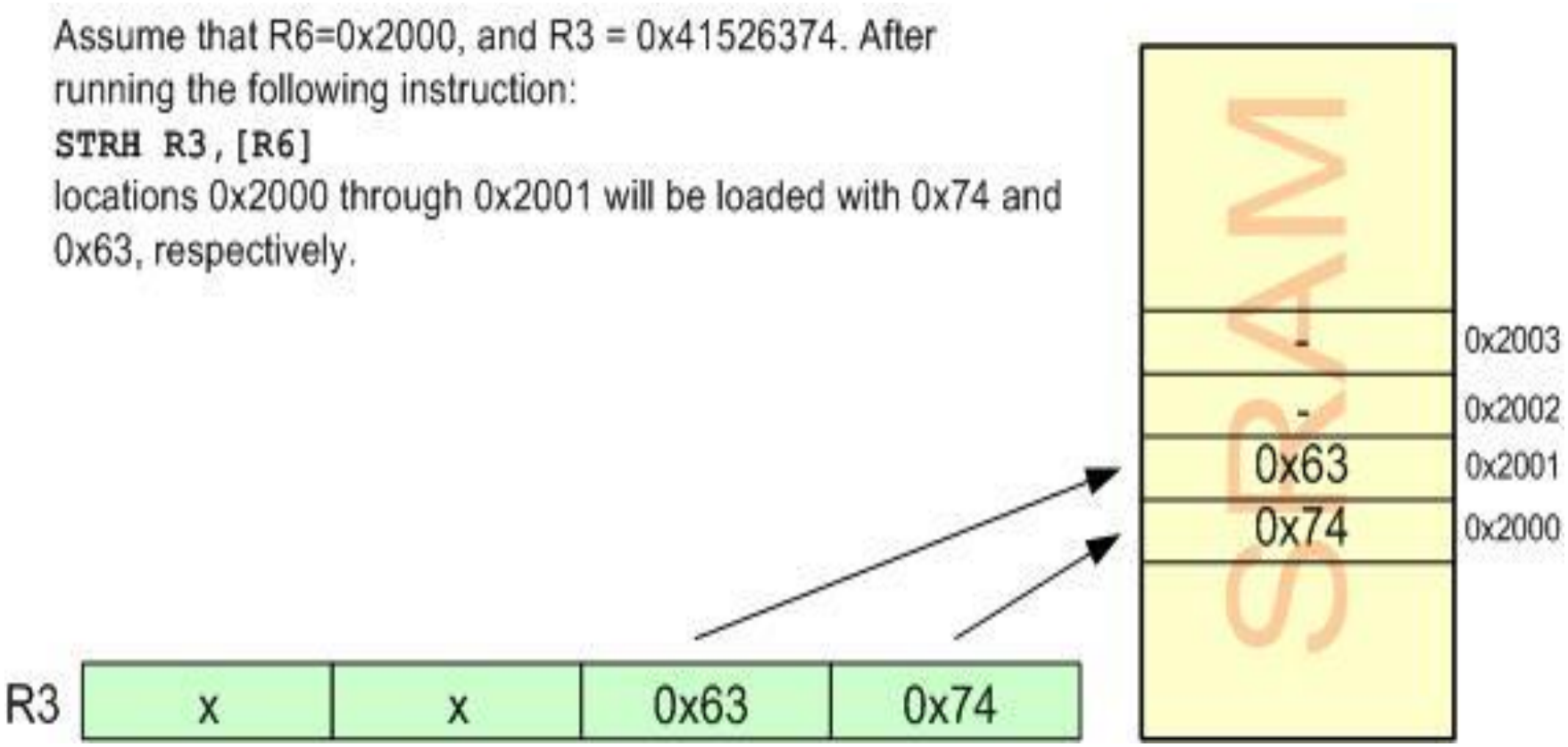


ARM Assembly Language Programming & Architecture by Mazidi, et al.

Executing the LDRH Instruction

STRH Rx,[Rd] instruction

;store half-word (2-byte) in register Rx into locations pointed to by Rd



Executing the STRH Instruction

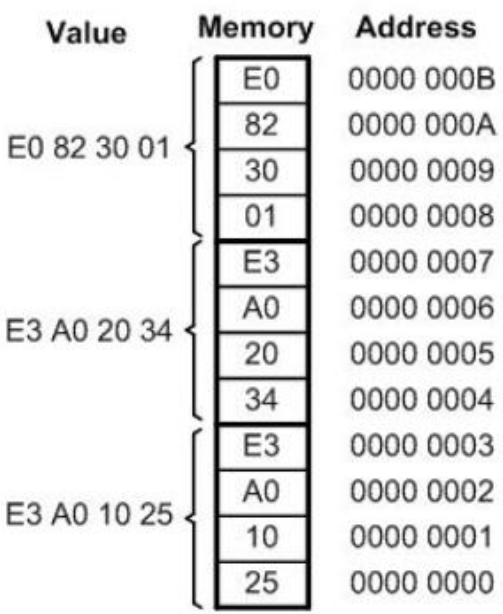
ARM Assembly Language Programming & Architecture by Mazidi, et al.

Data Size	Bits	Decimal	Hexadecimal	Directive	Instruction
Byte	8	0 – 255	0 - 0xFF	DCB	STRB/LDRB
Half-word	16	0 – 65535	0 - 0xFFFF	DCW	STRH/LDRH
Word	32	$0 – 2^{32}-1$	0 - 0xFFFFFFFF	DCD	STR/LDR

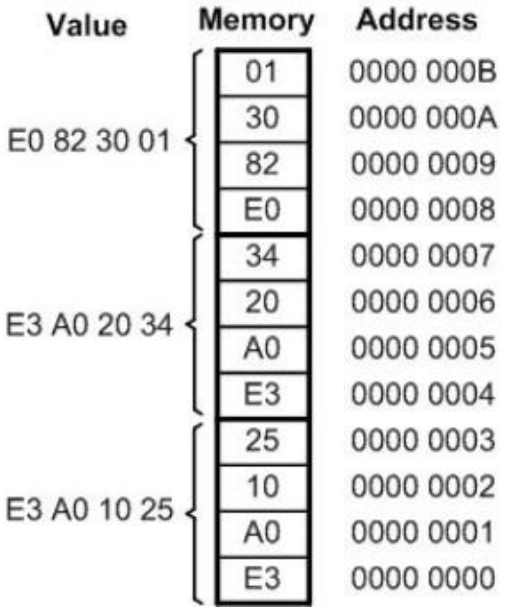
Unsigned Data Range in ARM and associated Instructions

Little endian vs. big endian

In the big endian method, the high byte goes to the low address, whereas in the little endian method, the high byte goes to the high address and the low byte to the low address.

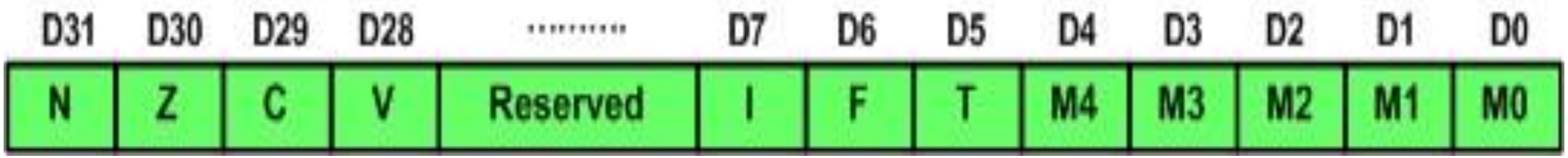


Little Endian Convention



Big Endian Convention

ARM CPSR (Current Program Status Register)

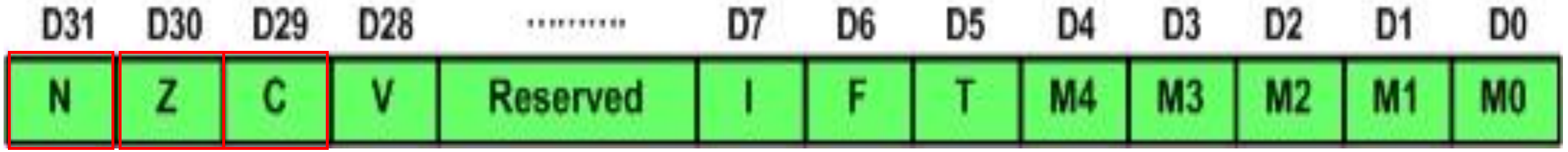


CPSR (Current Program Status Register)

The bits N, Z, C, and V are called conditional flags, meaning that they indicate some conditions that result after an instruction is executed.

S suffix and the status register

If we need an instruction to update the value of status bits in CPSR, we have to put S suffix at the end of instructions. For example, ADDS instead of ADD is used.



C, the carry flag

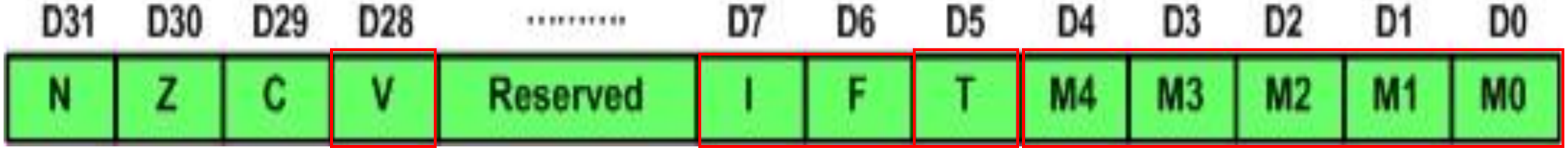
C=1; if there is a carry out from the D31 bit. This flag bit is affected after a 32-bit addition or subtraction.

Z, the zero flag

Z = 1; When result of ALU operation is zero. Whereas when Z = 0, the result is not zero.

N, the negative flag

It reflects the result of an arithmetic operation for signed numbers. If the D31 bit of the result is zero, then N = 0 and the result is positive. If the D31 bit is one, then N = 1 and the result is negative. The negative and V flag bits are used for the signed number arithmetic operations



V, the overflow flag

V=1; whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit.

- In general, the carry flag is used to detect errors in unsigned arithmetic operations while the overflow flag is used to detect errors in signed arithmetic operations.

T, Thumb state

T=1 Thumb state➔ ARM instructions are of 16 bit. It is used to increase the code density at the cost of performance

- The I and F flags are used to enable or disable the interrupt.

- M4 M3 M2 M1 M0 ➔ Modes of ARM

Mode Bits					Mode
1	0	0	0	0	User
1	0	0	0	1	FIQ
1	0	0	1	0	IRQ
1	0	0	1	1	Supervisor
1	0	1	1	1	Abort
1	1	0	1	1	Undefined
1	1	1	1	1	System

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

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

- While instructions tell the CPU what to do, directives (also called pseudo instructions) give directions to the assembler.

For example, the MOV and ADD instructions are commands to the CPU,
But EQU, END, and ENTRY are directives to the assembler.

- The directives help us develop our program easier and make our program legible (more readable).
- 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.

The following is the format:
AREA sectionname attribute, attribute, …
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:

- The ALIGN directive aligns the data element or instructions on an address that is a multiple of its parameter.
- When the ALIGN is used for CODE and READONLY, it is aligned in 4-bytes address boundary by default. This is because, the ARM instructions are 32 bit word.

EXPORT and IMPORT:

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

A project may contain multiple source files. You may need to use a symbol in a source file that is defined in another source file. In order for a symbol to be found by a different program file, we need to declare that symbol name as a global variable. The EXPORT directive declares a symbol that can be used in different program files. GLOBAL is a synonym for EXPORT. The IMPORT directive provides the assembler with a name that is not defined in the current assembly.

```
AREA    Example , CODE, READONLY
        IMPORT  User_Code       ; Import the function name from
                                ; other source file.
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external
                                ; modules.
DoAdd   ADD     R0,R0,R1
```

DCD (Define constant word):
Allocates a word size memory and initializes the values.

ENTRY:
The ENTRY directive declares an entry point to the program. It marks the first instruction to be executed.

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.

```
COUNT           EQU     0x25
…       …       ….
   MOV   R2, #COUNT     ;R2 = 0x25
```

RN (equate)
This is used to define a name for a register.
VAL1 RN R1 ;define VAL1 as a name for R1

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

LDR R7, =0x112233➔ Loads R7 with 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
```

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.

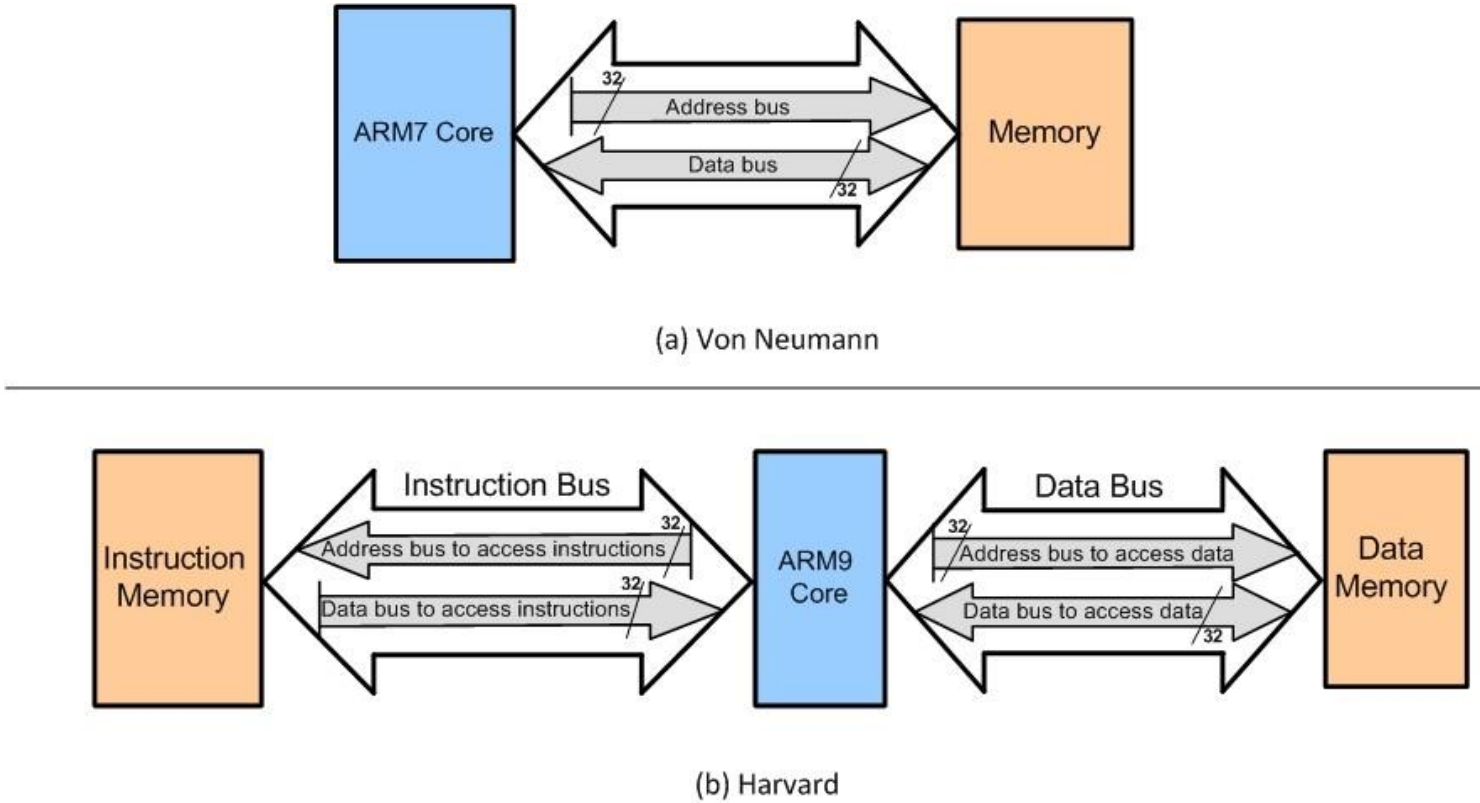
Some Widely Used ARM Memory Allocation Directives

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.

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,

- Here the CPU puts the value of R5 (0x40000200) 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

Literal (Immediate) MOV R0, #15

ADD R1, R2, #12

Indexed addressing mode

In the indexed addressing mode, a register is used as a pointer to the data location.

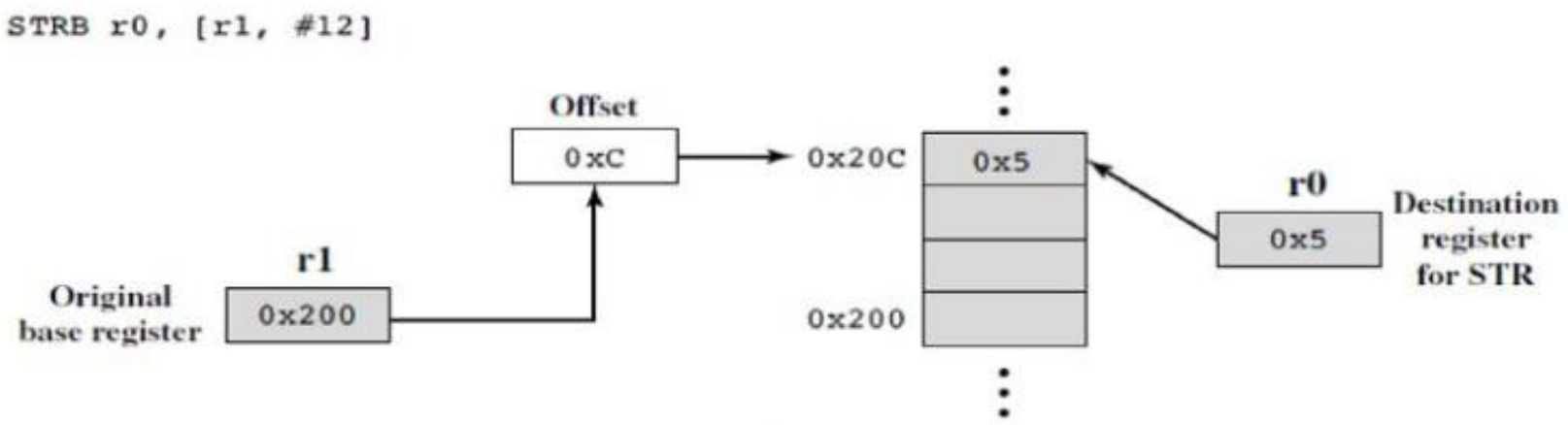
Register indirect: LDR R0, [R1]

The ARM provides three indexed addressing modes.

- preindex
- preindex with write back
- post index.

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

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



```
STR r1, [r0, #4]
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

0x20008000

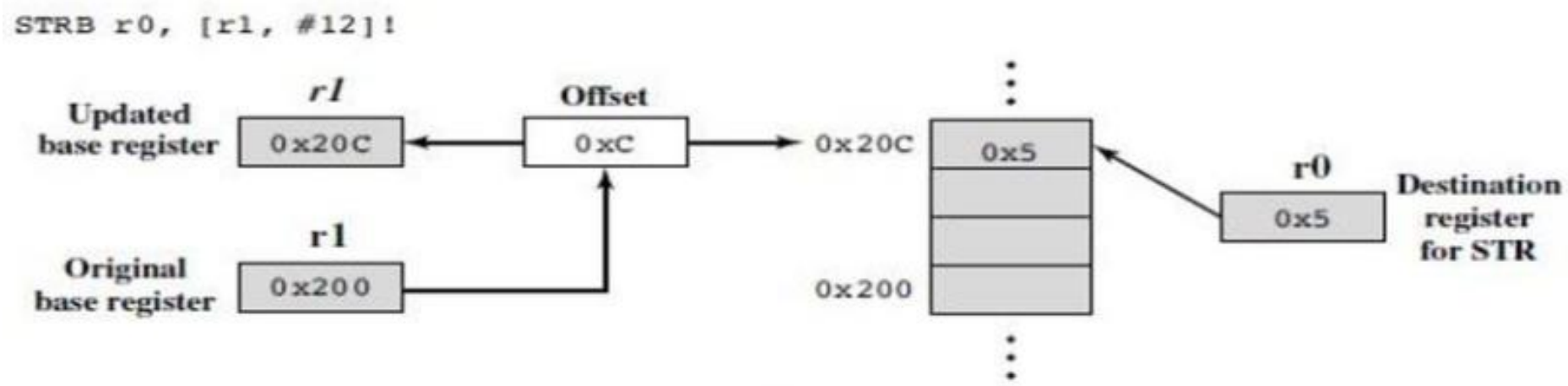
r0 after store

0x20008000

Memory Address	Memory Data
0x20008007	0x76
0x20008006	0x54
0x20008005	0x32
0x20008004	0x10
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

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

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



```
STR r1, [r0, #4]!      ;pre-increment
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

0x20008000

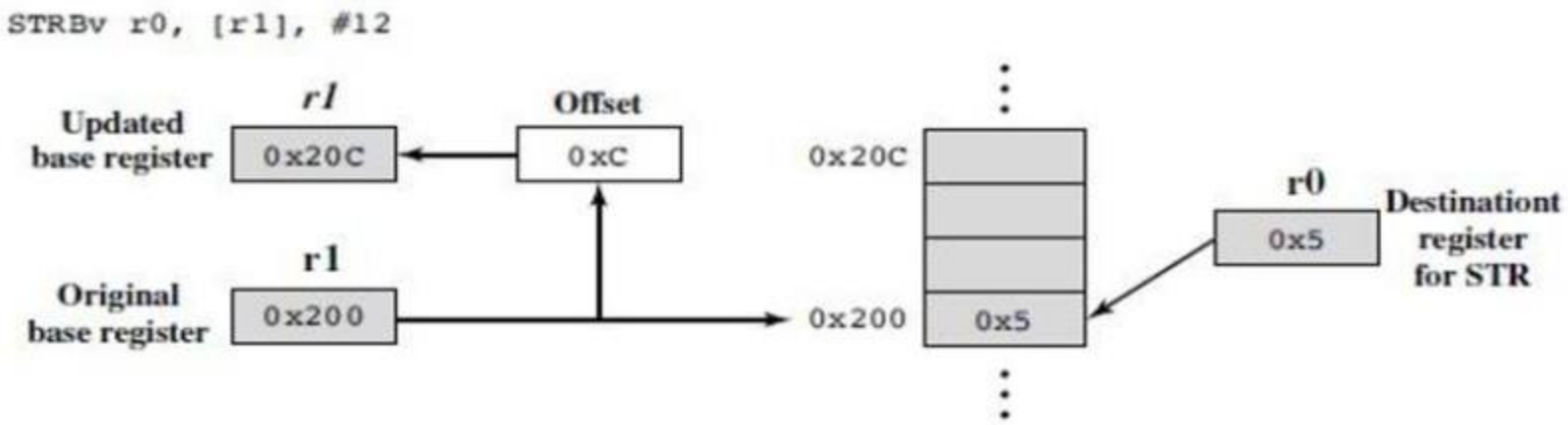
r0 after store

0x20008004

Memory Address	Memory Data
0x20008007	0x76
0x20008006	0x54
0x20008005	0x32
0x20008004	0x10
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

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

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



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

```
STR r1, [r0], #4   ;post-increment
; r0 = 0x20008000,  r1=0x76543210
```

r0 before store

0x20008000

r0 after store

0x20008004

Memory Address	Memory Data
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x76
0x20008002	0x54
0x20008001	0x32
0x20008000	0x10

Double Reg indirect (Register indirect indexed)

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

```
LDR r2, [r0, r1]
; r0 = 0x20008000,  r1=0x00000004,
  r2=0x00000000
```

r2 before load

0x00000000

r2 after load

0x76543210

Memory Address	Memory Data
0x20008007	0x76
0x20008006	0x54
0x20008005	0x32
0x20008004	0x10
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

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

LDR R0, [R1, r2, LSL #2] ; R0= R1+[4*r2]

```
LDR r0, [r1, r2, lsl #2]
; r1 = 0x20008000,  r2=0x00000001
```

```
Offset = 0x00000001 x 4
       = 0x00000004

Address = 0x20008000
        +0x00000004
        = 0x20008004

r0 after load
     0x76543210
```

Memory Address	Memory Data
0x20008007	0x76
0x20008006	0x54
0x20008005	0x32
0x20008004	0x10
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Program counter relative

LDR R0, [PC, #offset]

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

RISC Features

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