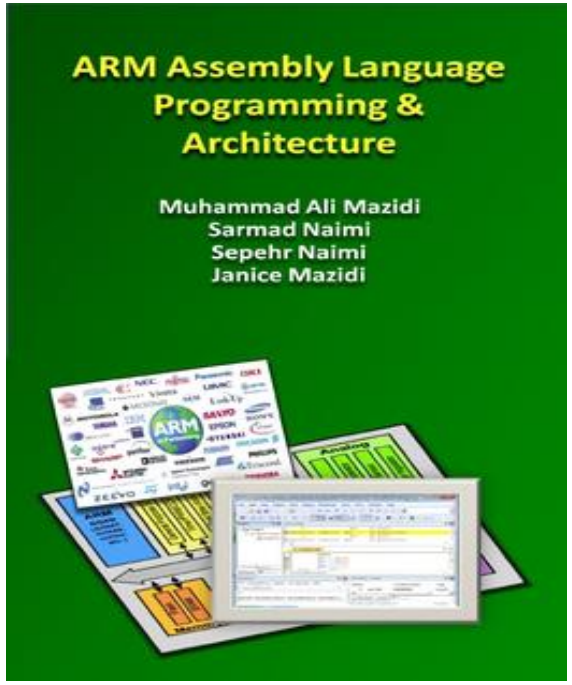
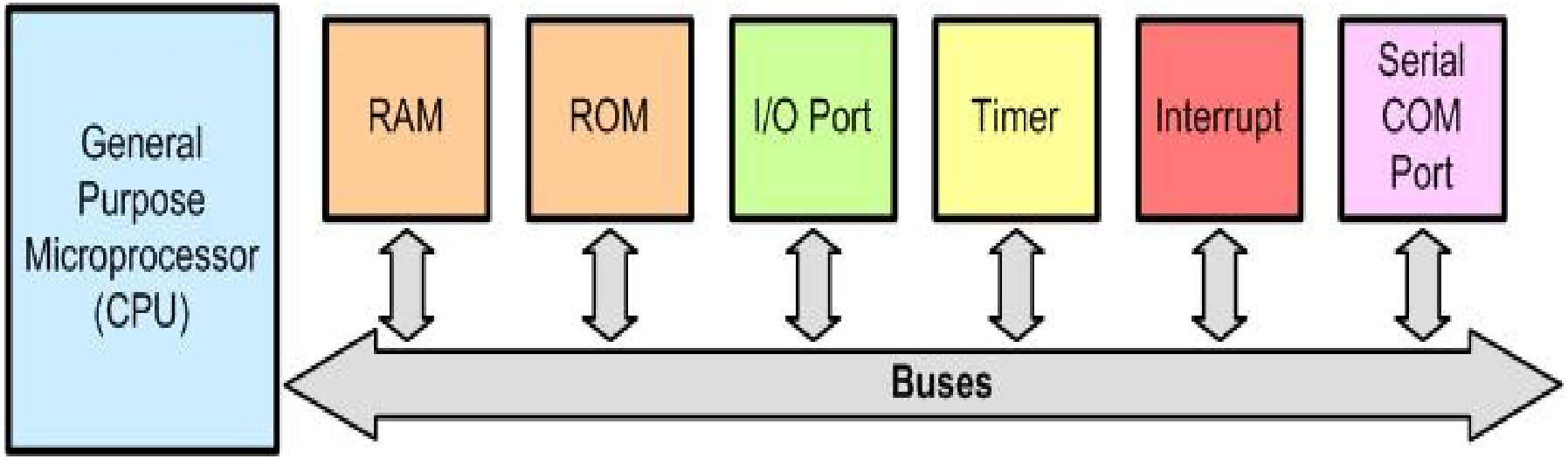
The History of ARM and Microcontrollers
Chapter 1

ARM Assembly
Language Programming
& Architecture

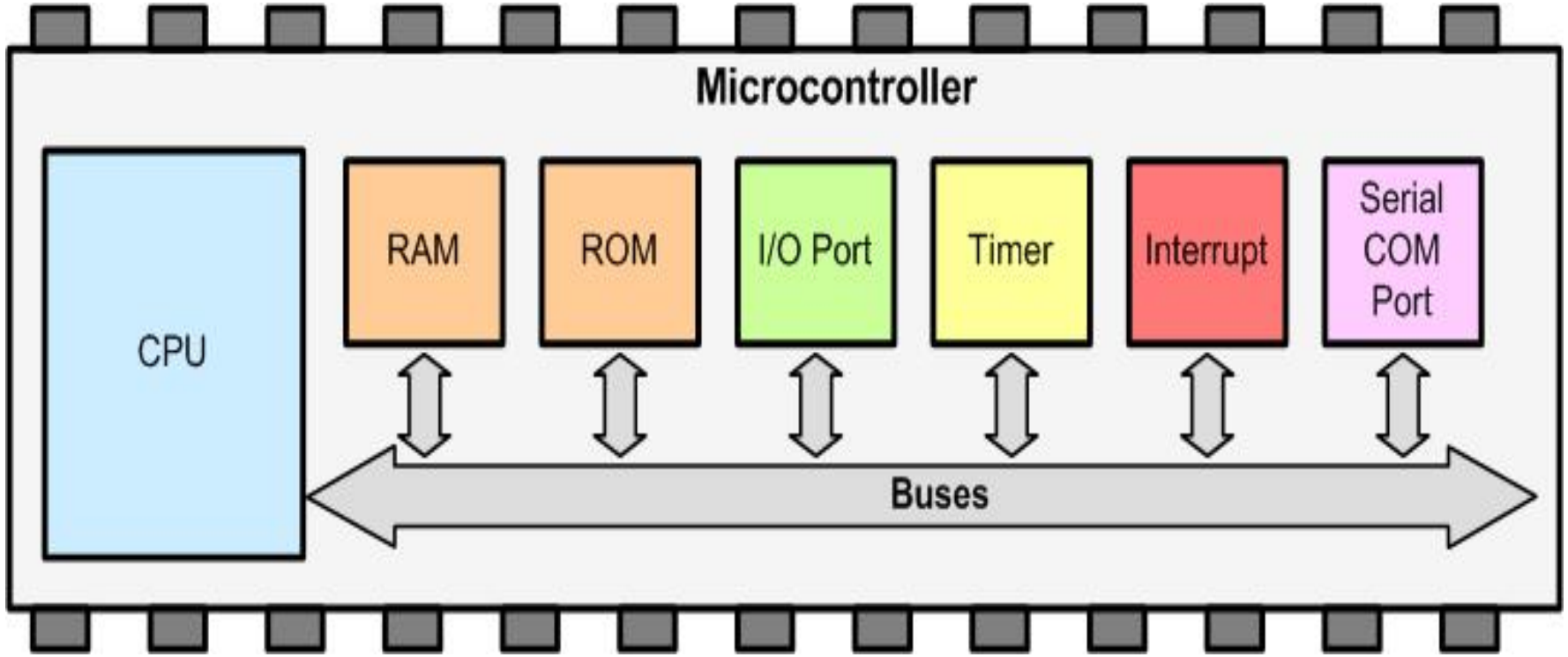


Microprocessors and Microcontrollers



A Computer Made by General Purpose Microprocessor

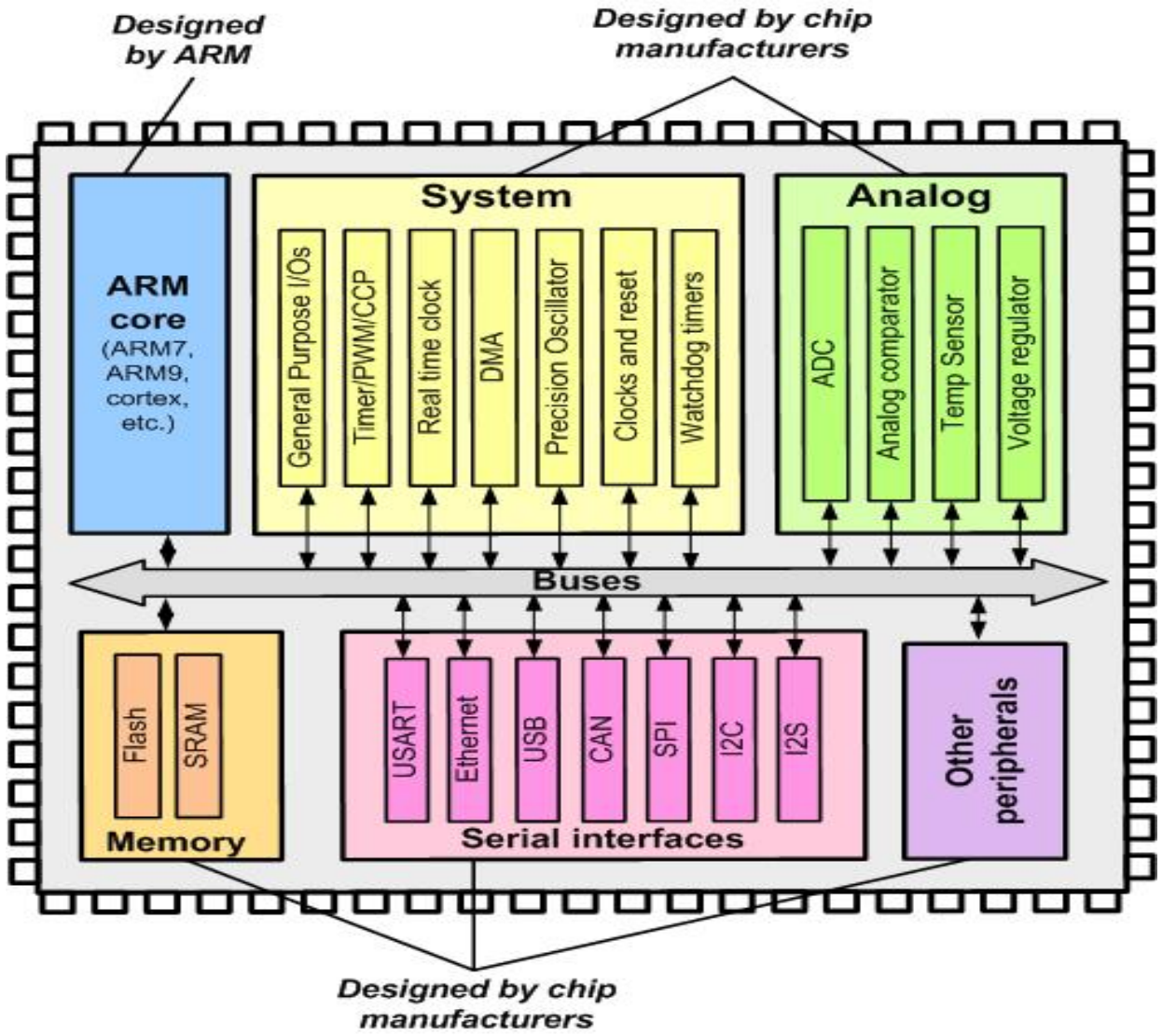
The microprocessors do not contain RAM, ROM, or I/O peripherals. As a result, they must be connected externally to RAM, ROM and I/O through buses



Simplified View of the Internal Parts of Microcontrollers (SOC)

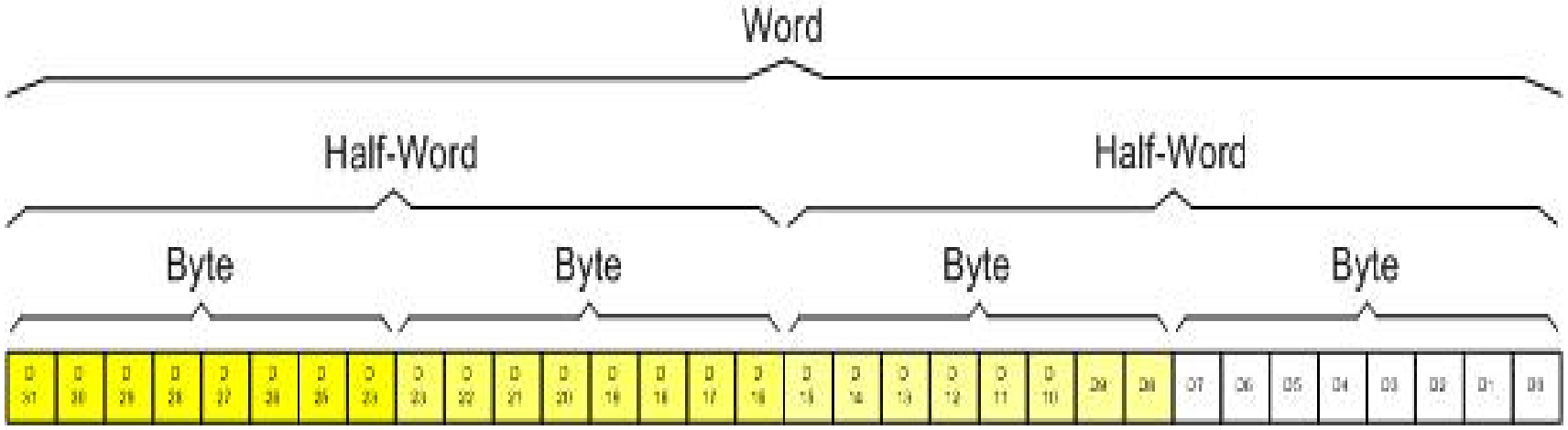
In microcontroller, CPU, RAM, ROM, and I/Os, are put together on a single IC chip and it is called *microcontroller*. SOC (System on Chip) and MCU (Micro Controller Unit) are other names used to refer to microcontrollers.

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



ARM Simplified Block Diagram

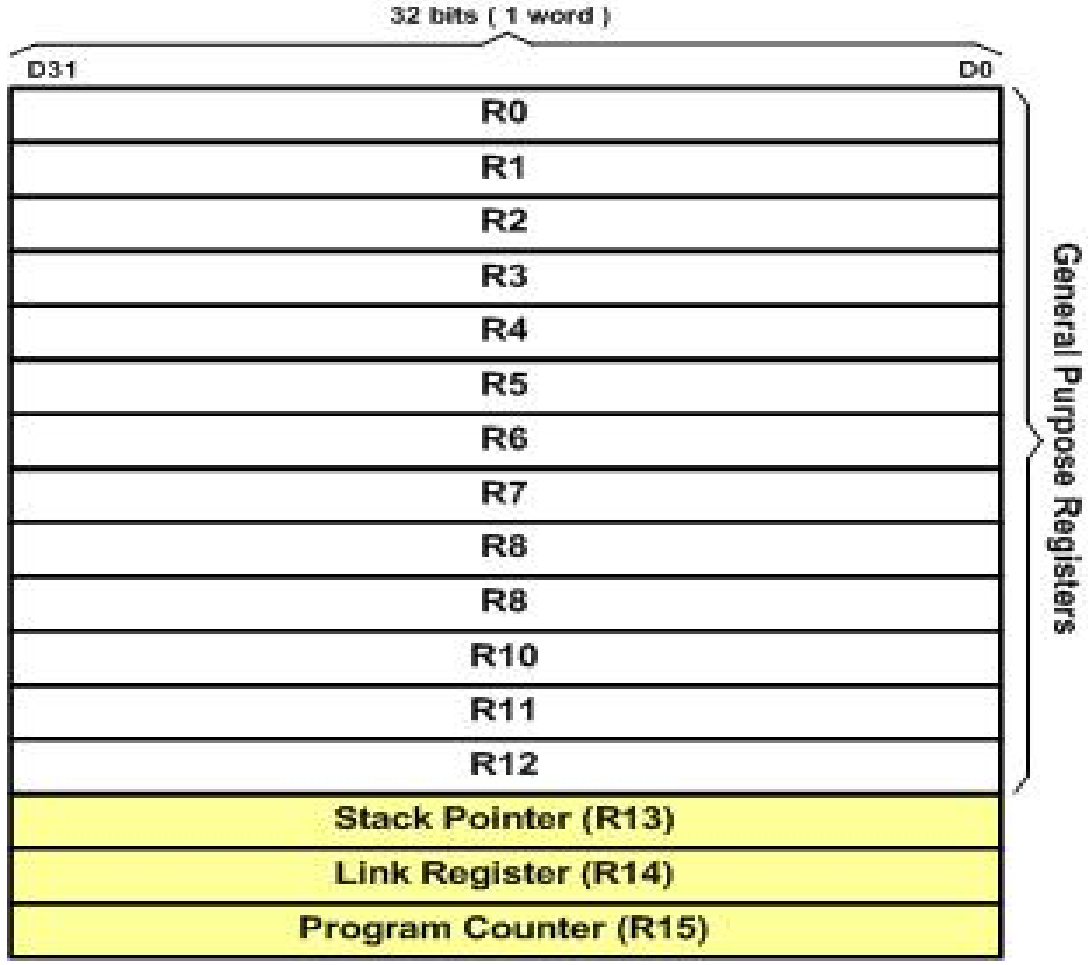
ARM Architecture



ARM Registers Data Size

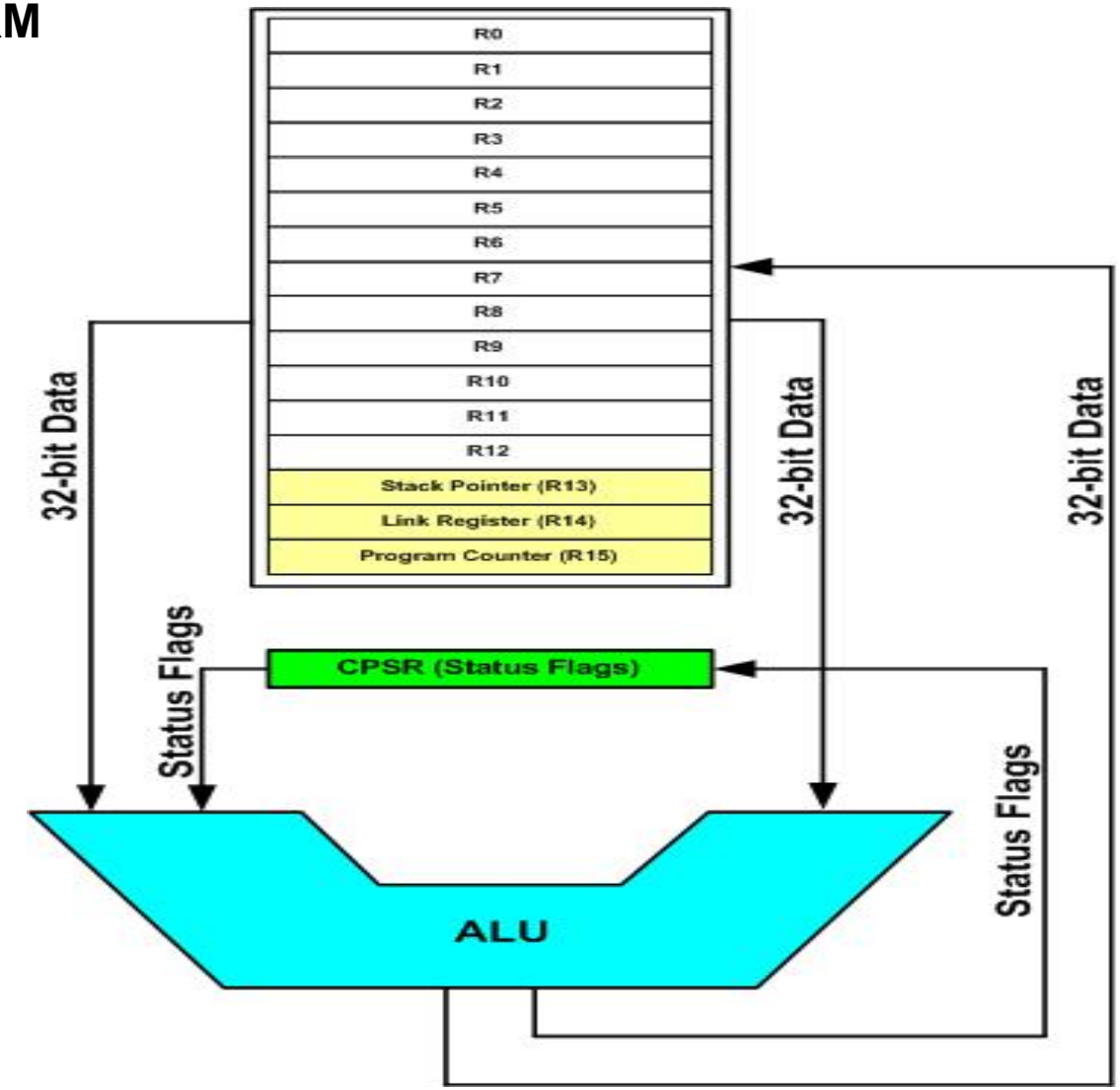
ARM supports byte, half word (16 bit) and word (32 bit) data types.

The General Purpose Registers



ARM Registers

The Special Function Registers in ARM



ARM Registers and ALU

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

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.

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.

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

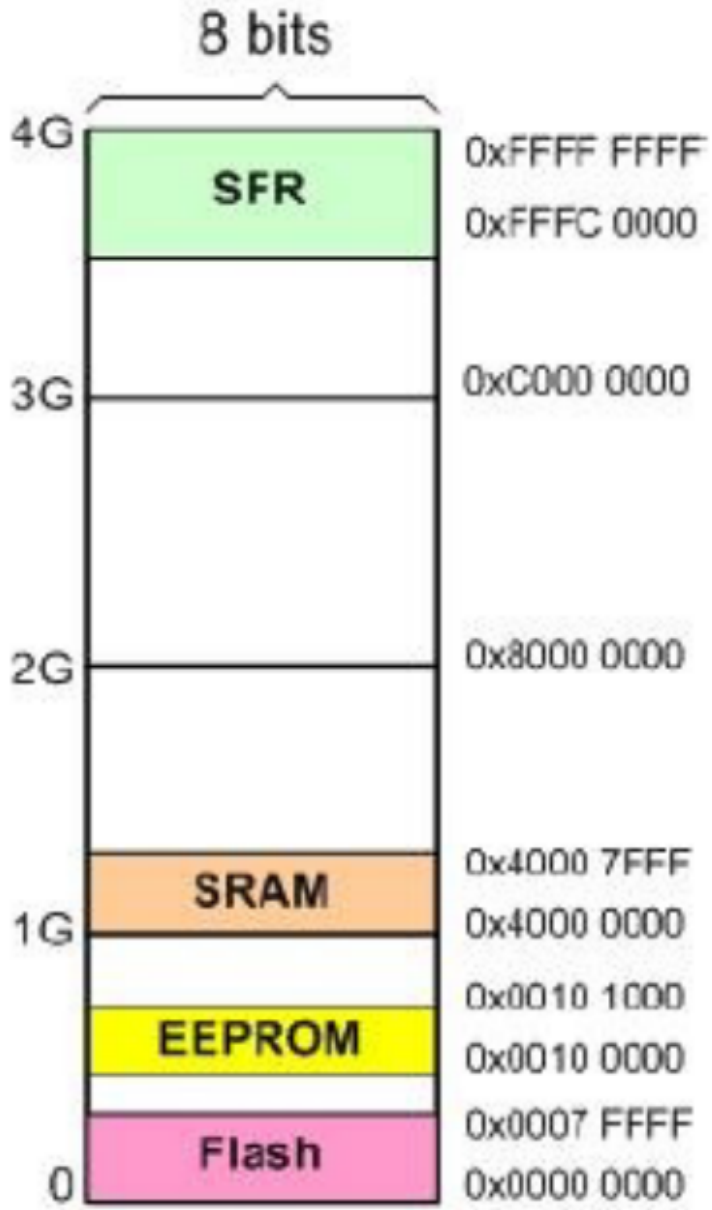
Solution:

(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.

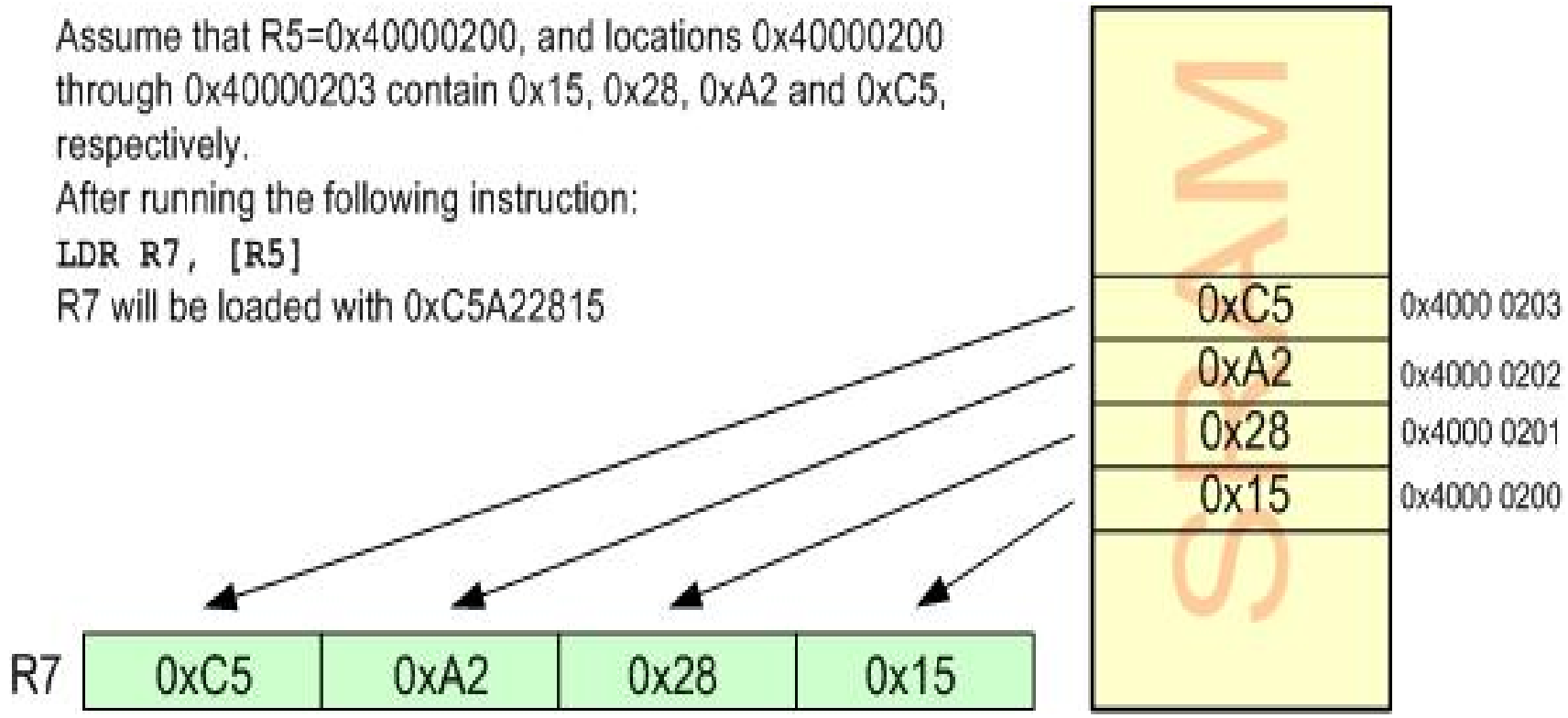


ARM Assembly Language Programming & Architecture by Mazidi, et al.

Load and Store Instructions in ARM

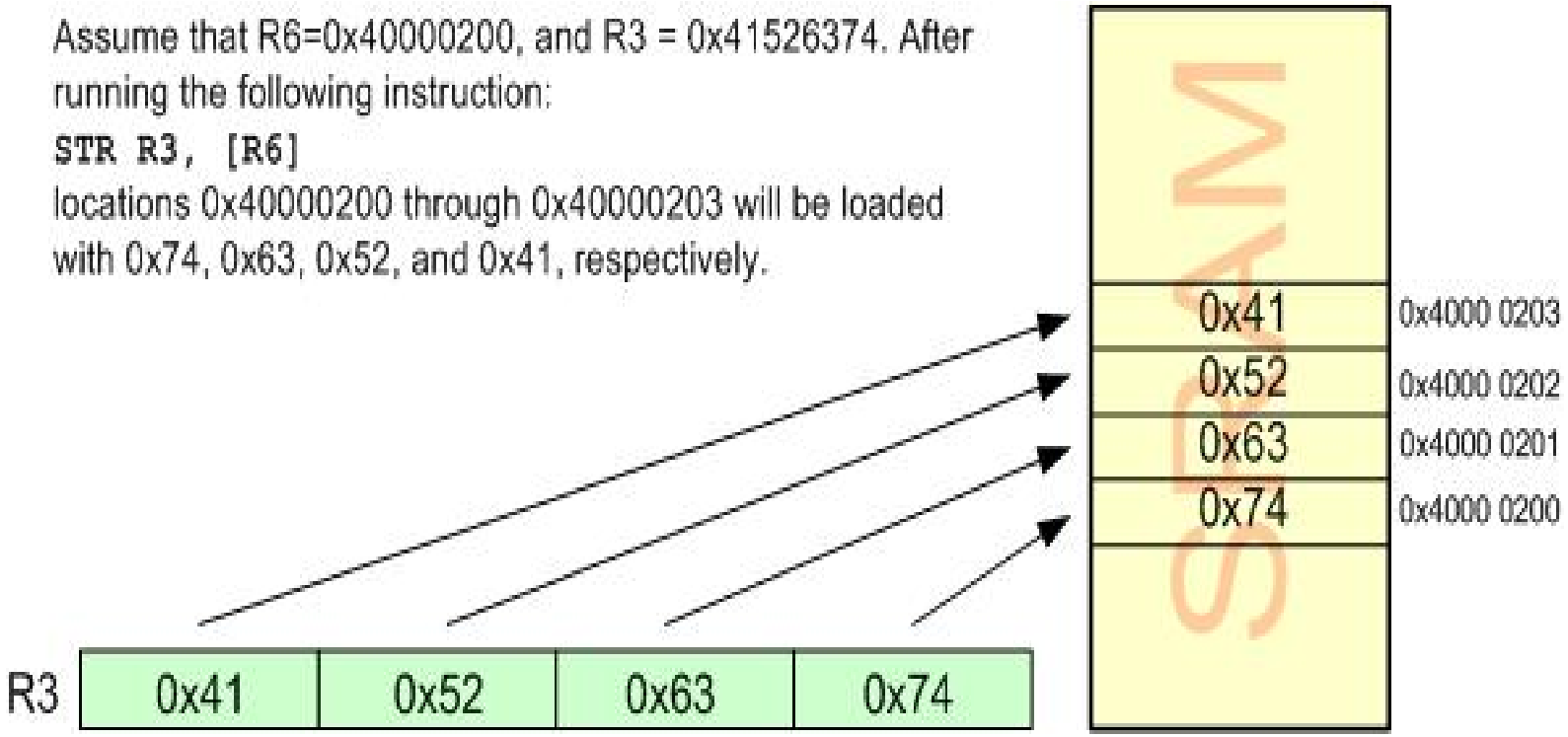
Every instruction of ARM is fixed at 32-bit. The fixed size instruction is one of the most important characteristics of RISC architecture.

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



Executing the LDR Instruction

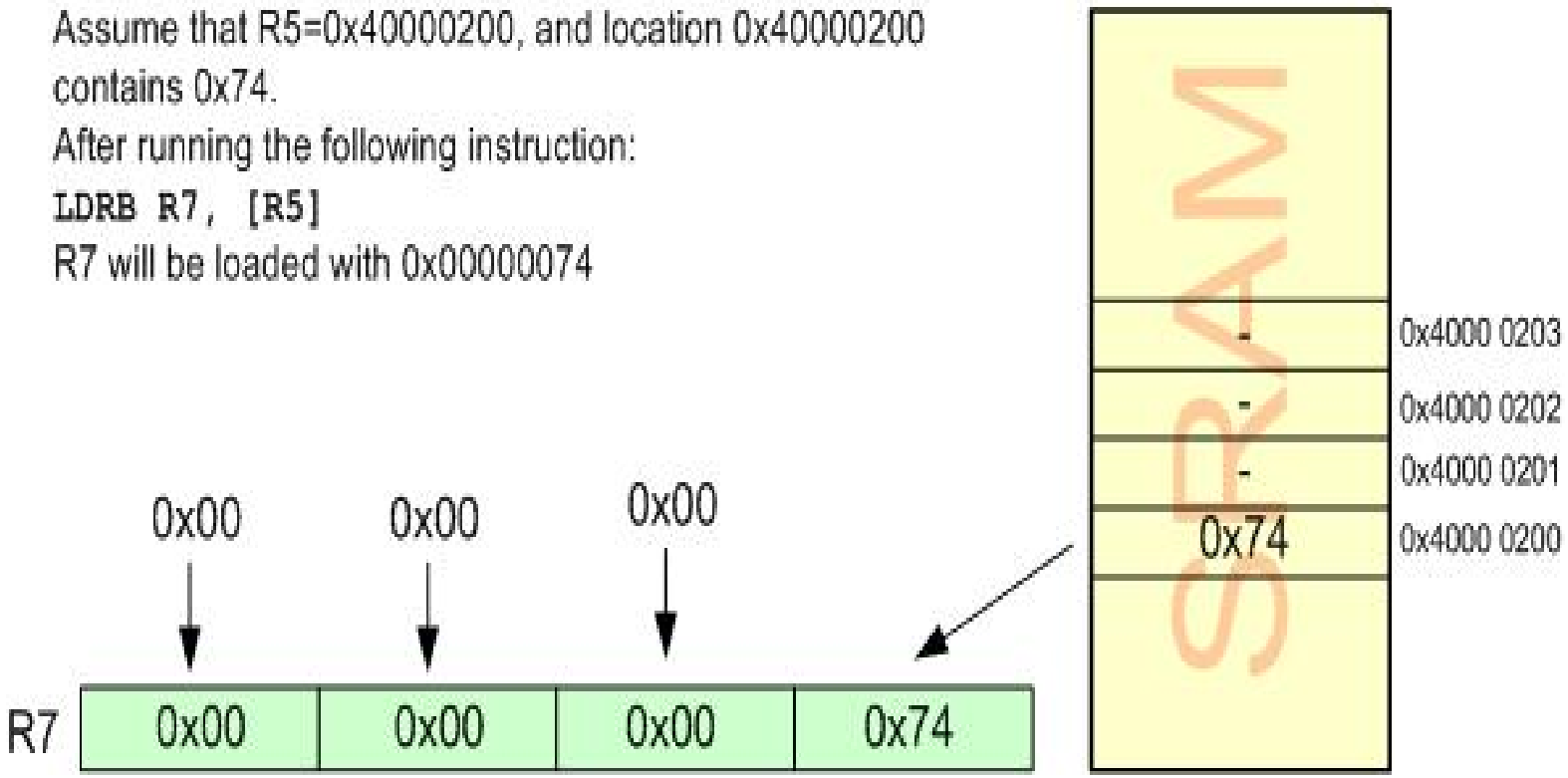
STR Rx,[Rd] instruction
;store register Rx into locations pointed to by Rd



Executing the STR Instruction

LDRB Rd, [Rx] instruction

;load 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

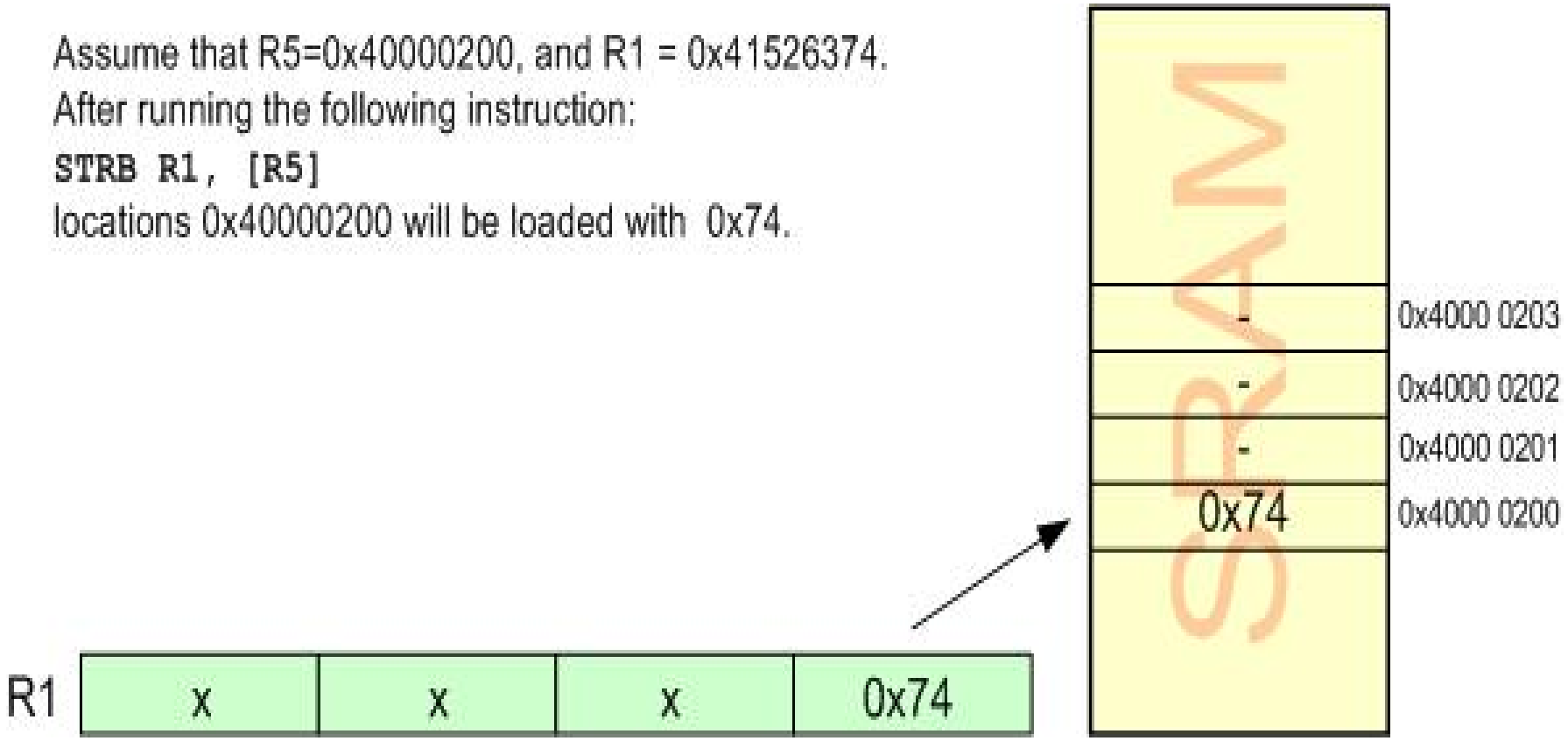
STRB Rx,[Rd] instruction

;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

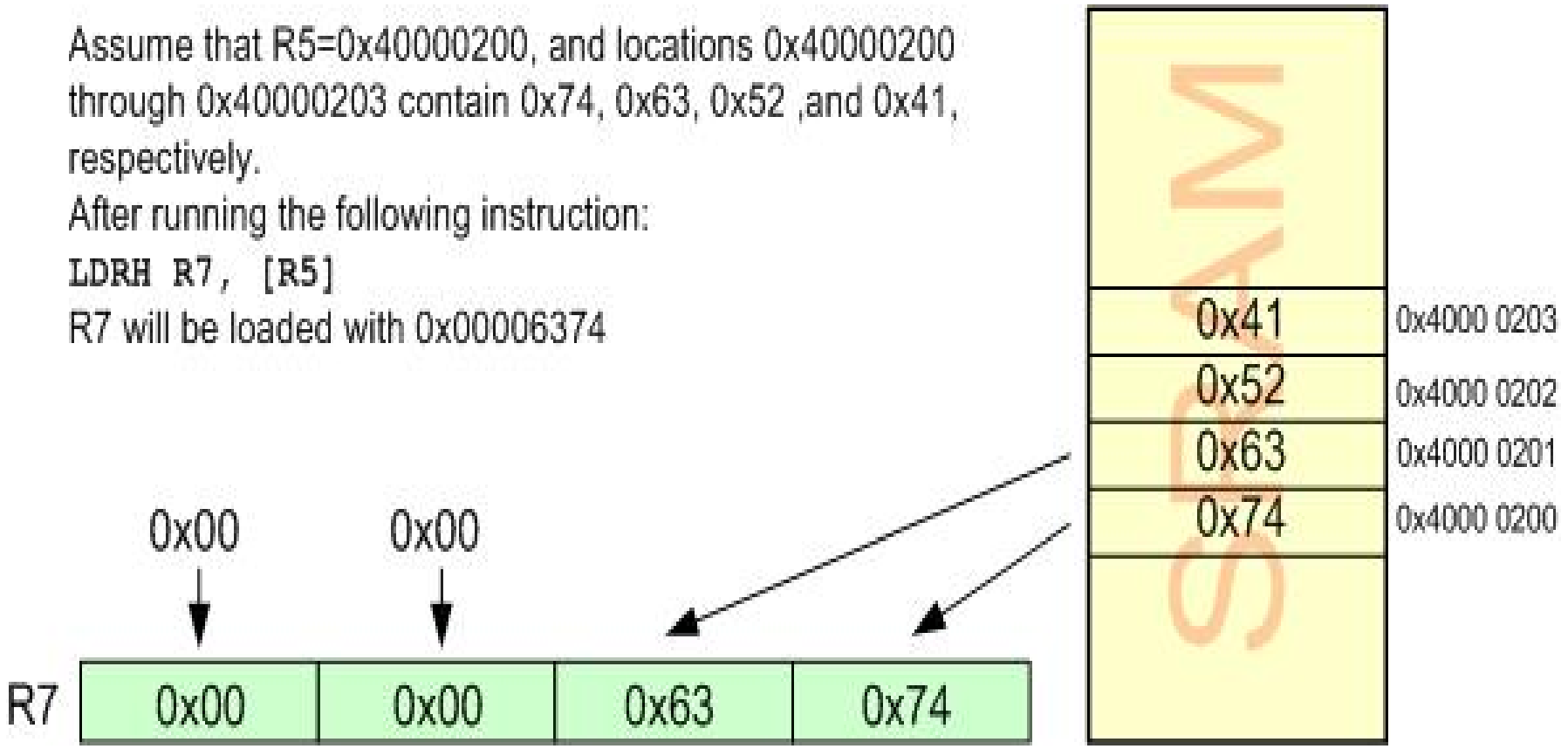
;load Rd with the half-word (16-bit or 2 bytes) pointed to by Rx register

Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x74, 0x63, 0x52 ,and 0x41, respectively.

After running the following instruction:

```
LDRH R7, [R5]
```

R7 will be loaded with 0x00006374



Executing the LDRH Instruction

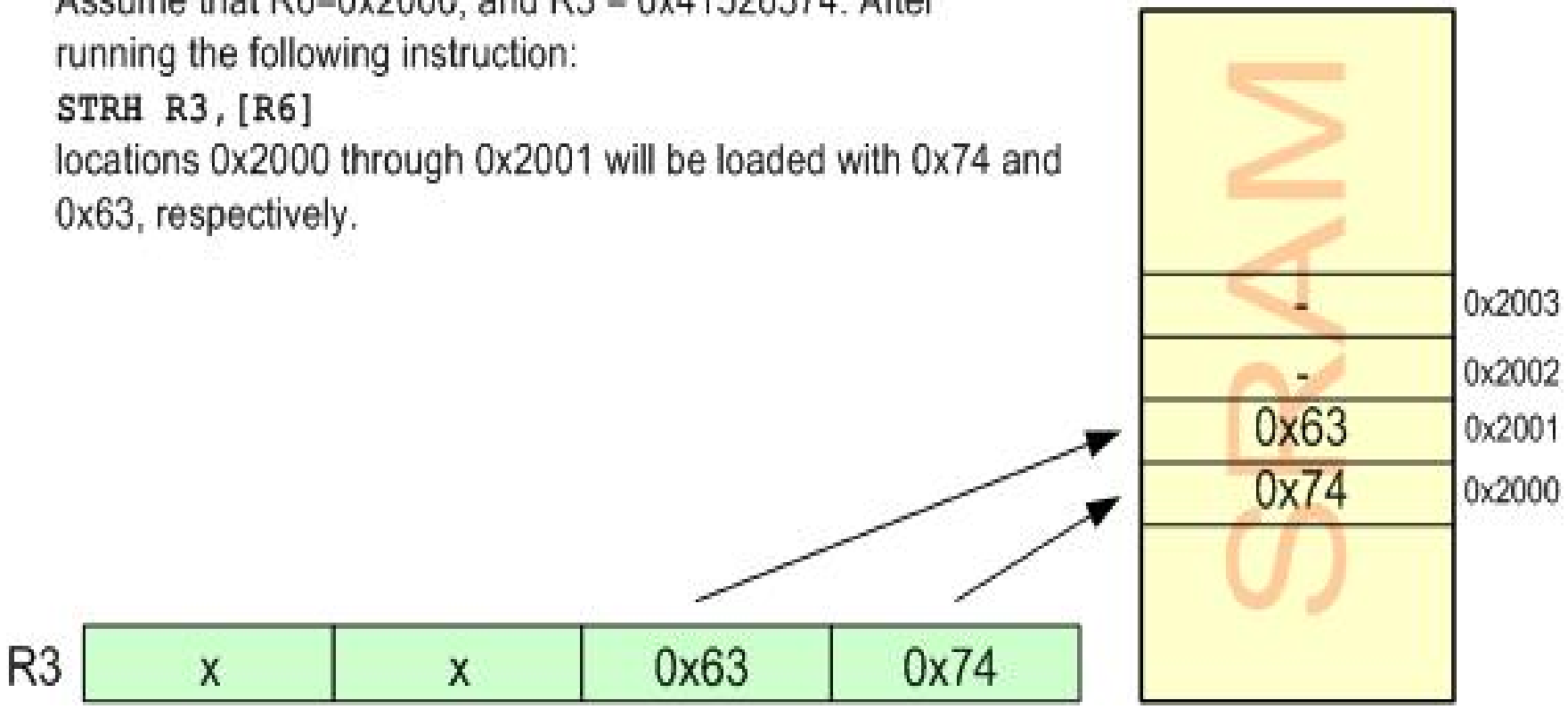
STRH Rx,[Rd] instruction

;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

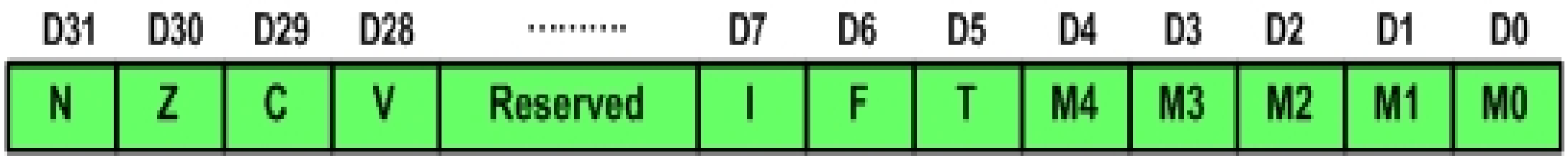
Little endian vs. big endian

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.

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

ARM CPSR (Current Program Status Register)



CPSR (Current Program Status Register)

S suffix and the status register

Most of ARM instructions can affect the status bits of CPSR according to the result. 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. That means, for example, ADDS instead of ADD is used.

C, the carry flag

This flag is set whenever 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

The zero flag reflects the result of an arithmetic or logic operation. If the result is zero, then Z = 1. Therefore, Z = 0 if the result is not zero.

N, the negative flag

The negative flag reflects the result of an arithmetic operation. 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

This flag is set 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.

The T flag bit is used to indicate the ARM is in Thumb state. The I and F flags are used to enable or disable the interrupt.

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:

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 separate storage for the name, but associates a register with that name. It improves the

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
```

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.

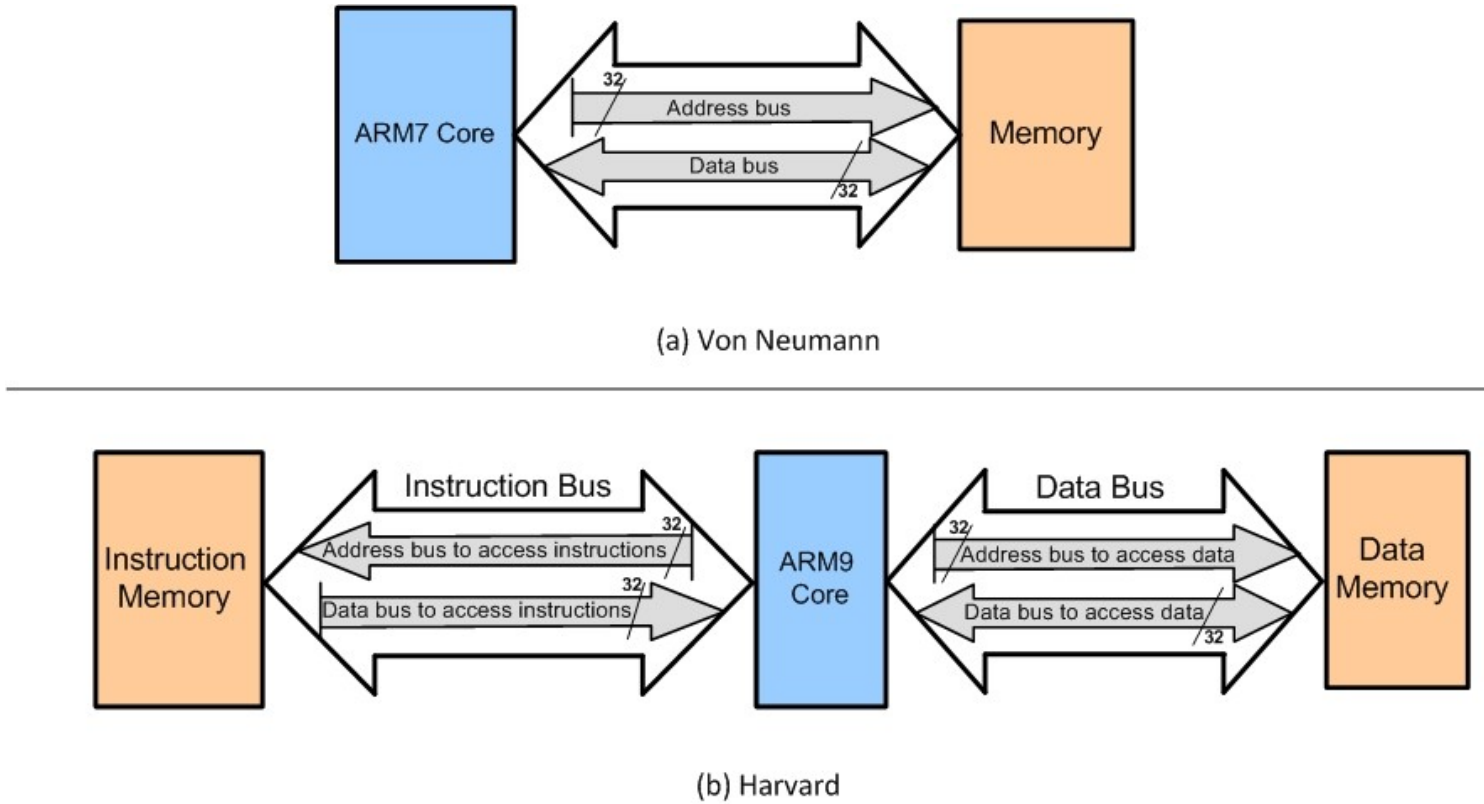
Some Widely Used ARM Memory Allocation Directives

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.

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

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 addressing mode

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, 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
```

Table 6-2 summarizes these addressing modes. Each of these indexed addressing mode can be used with offset of fixed value or offset of a shifted register.

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

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]

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