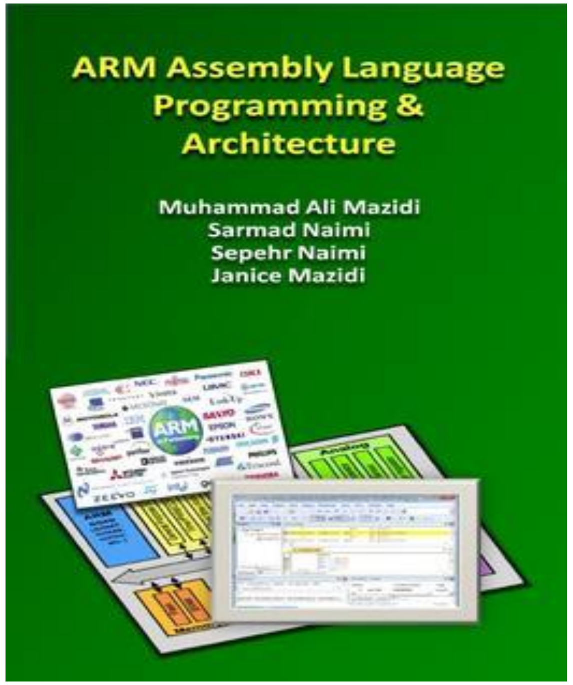
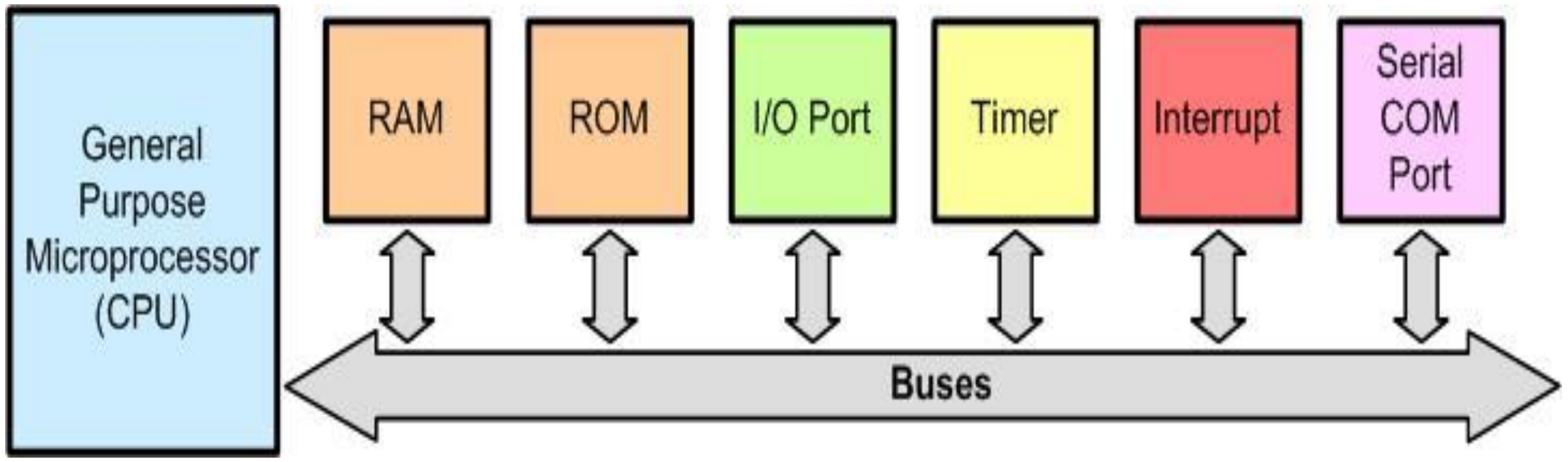
The History of ARM and Microcontrollers
Chapter 1

ARM Assembly Language Programming & Architecture

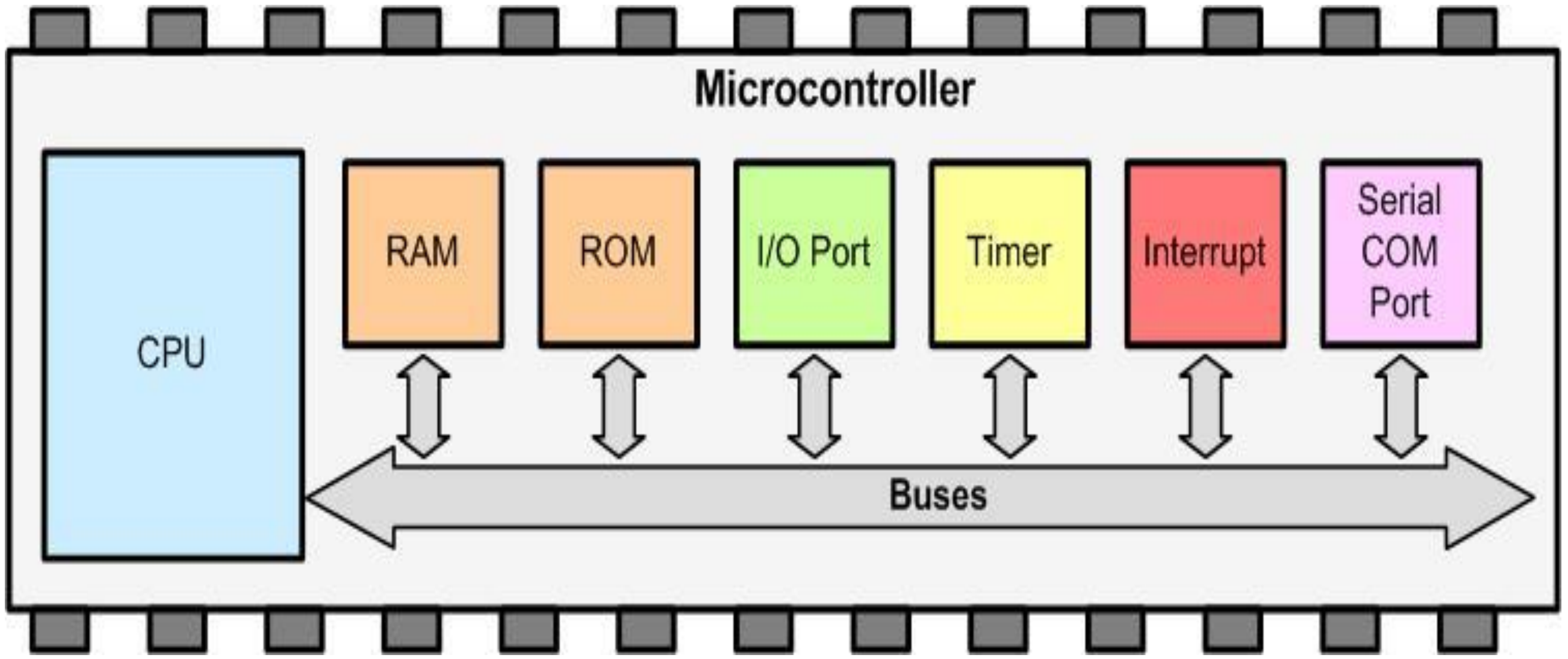


Microprocessors and Microcontrollers



A Computer Made by General Purpose Microprocessor

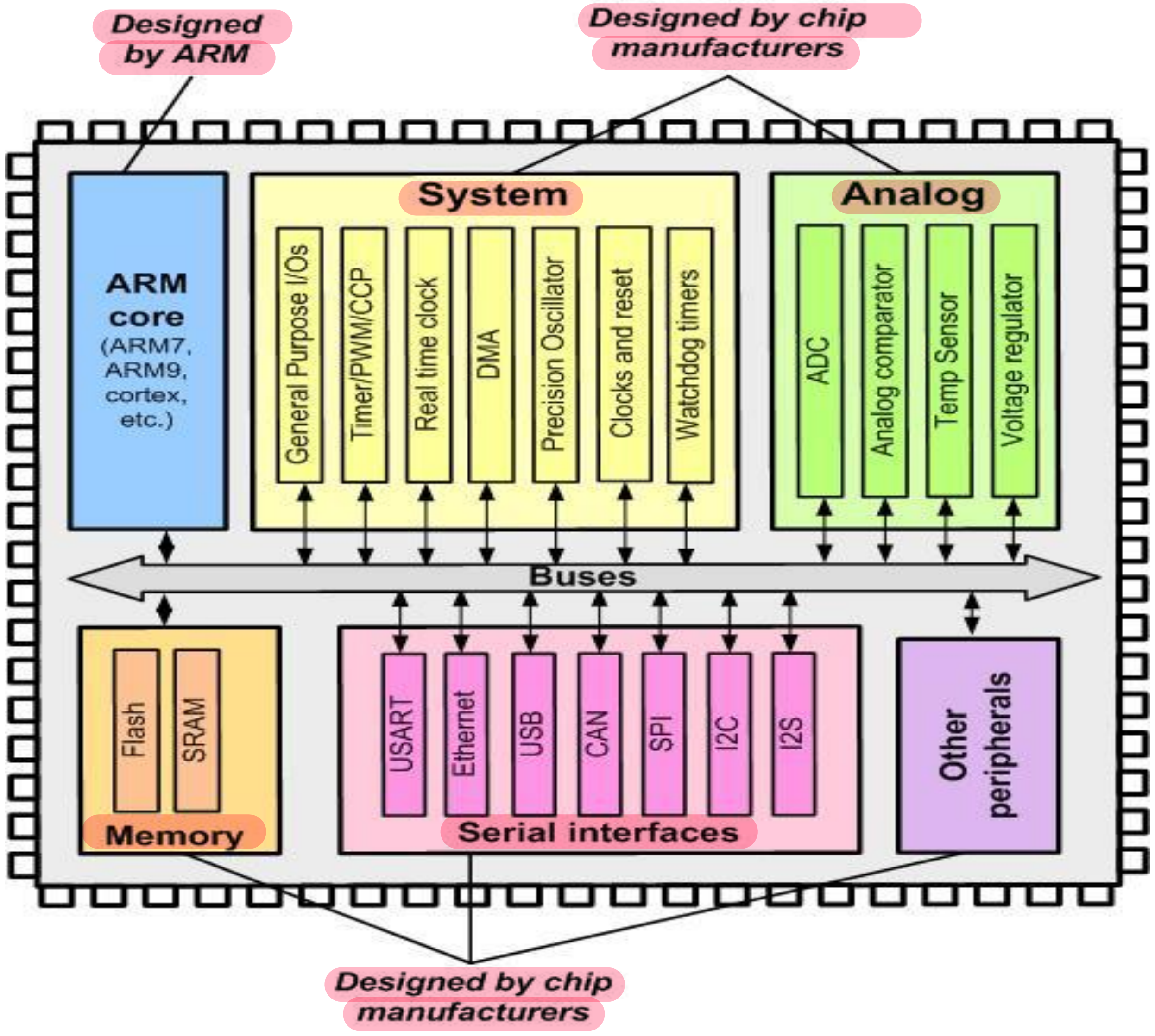
The microprocessors do not contain RAM, ROM, or I/O peripherals. As a result, they must be connected externally to RAM, ROM and I/O through buses



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.

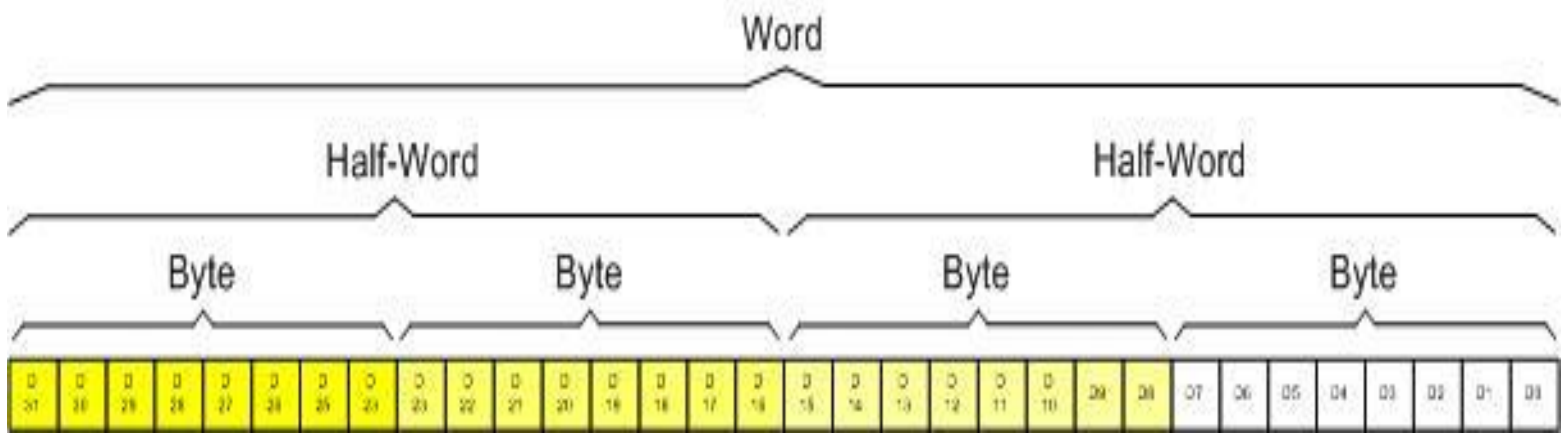
ARM has defined the details of architecture, registers, instruction set, memory map, and timing of the ARM CPU and holds the copyright to it. The various design houses and semiconductor manufacturers license the IP (intellectual property) for the CPU and can add their own peripherals as they please. 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 Assembly Language Programming & Architecture by Mazidi, et al.

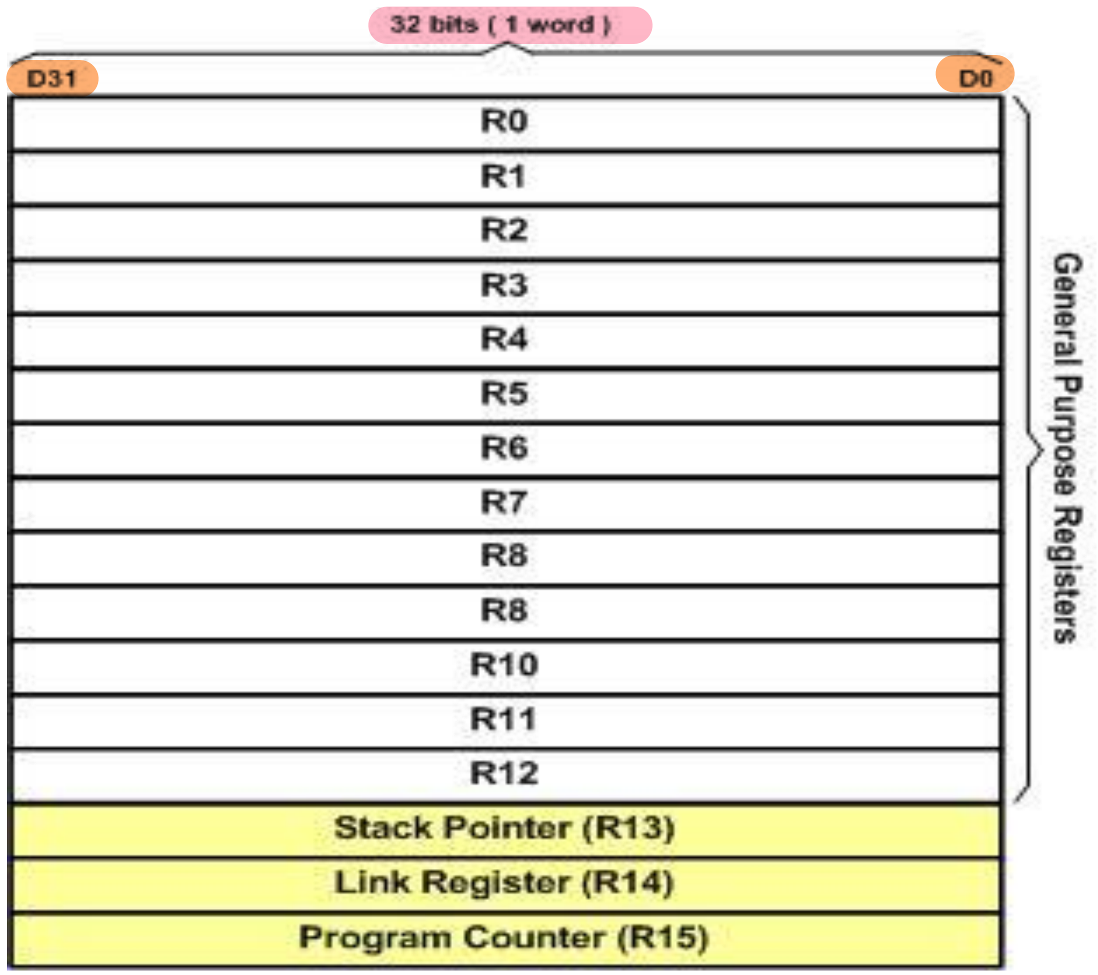
ARM Architecture



ARM Registers Data Size

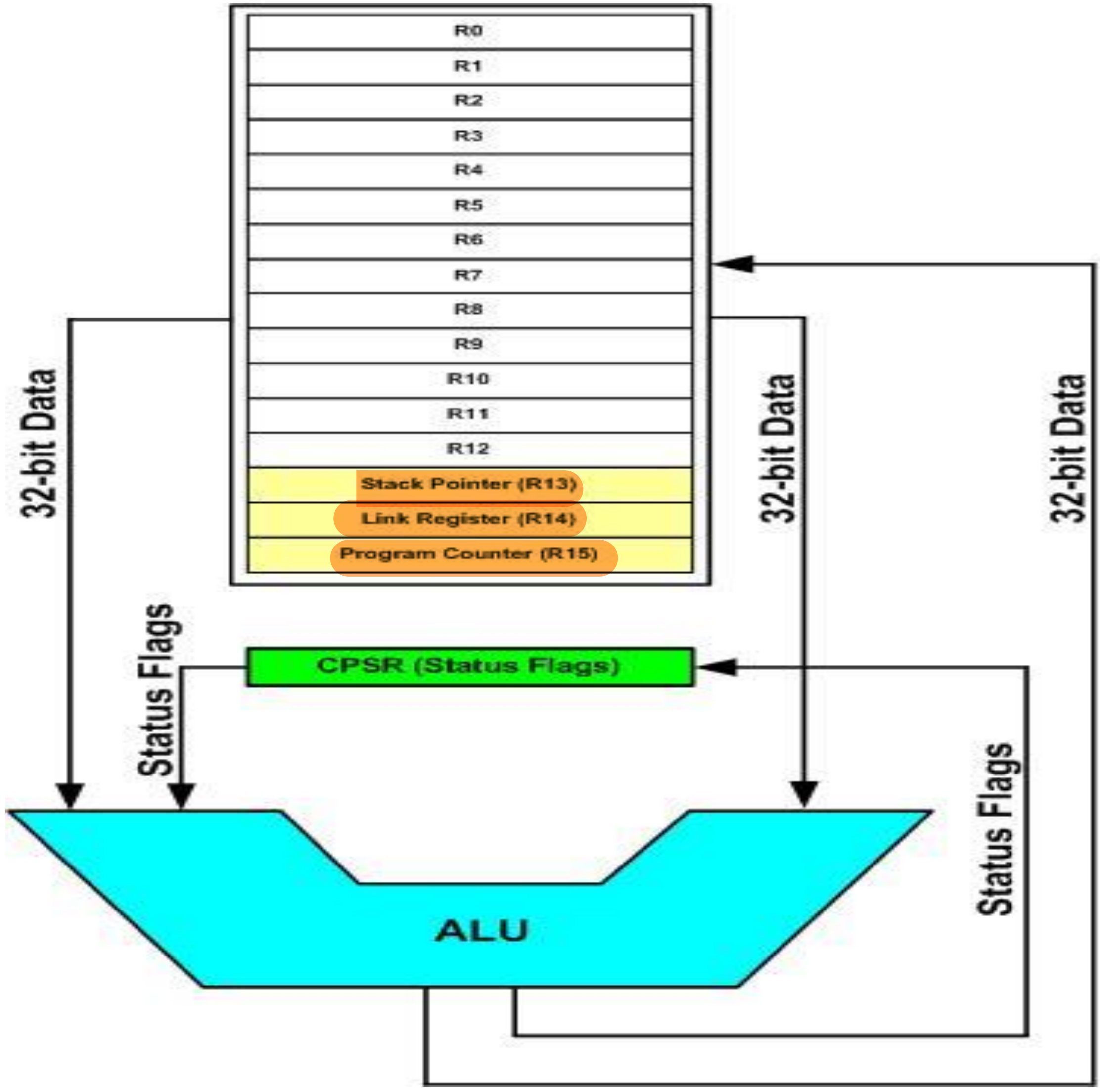
ARM supports byte, half word (16 bit) and word (32 bit) data types.

The General Purpose Registers



ARM Registers

The Special Function Registers in ARM



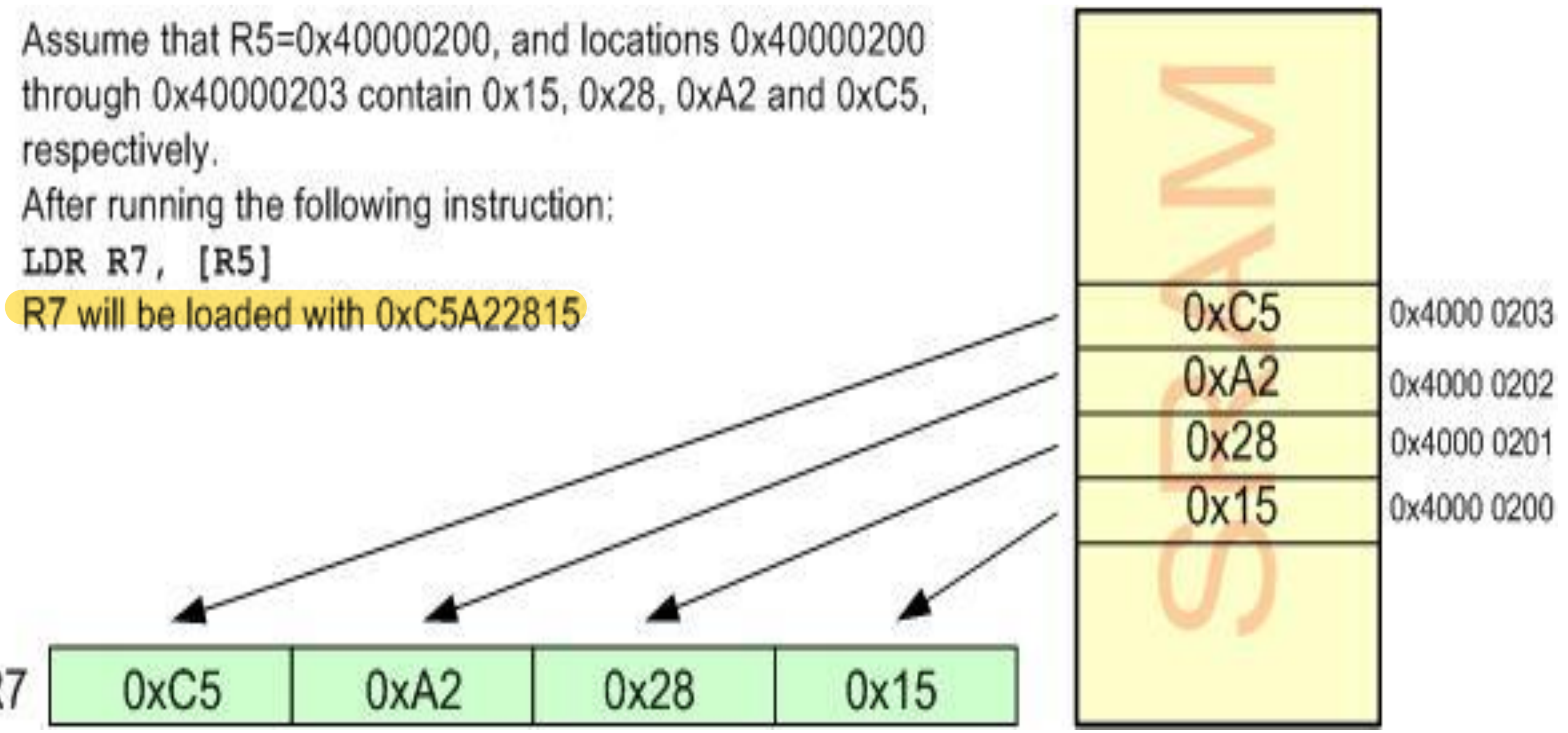
ARM Registers and ALU

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 to SFRs, the GPRs (R0-R12) do not have any specific function and are used for storing general data.

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.



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

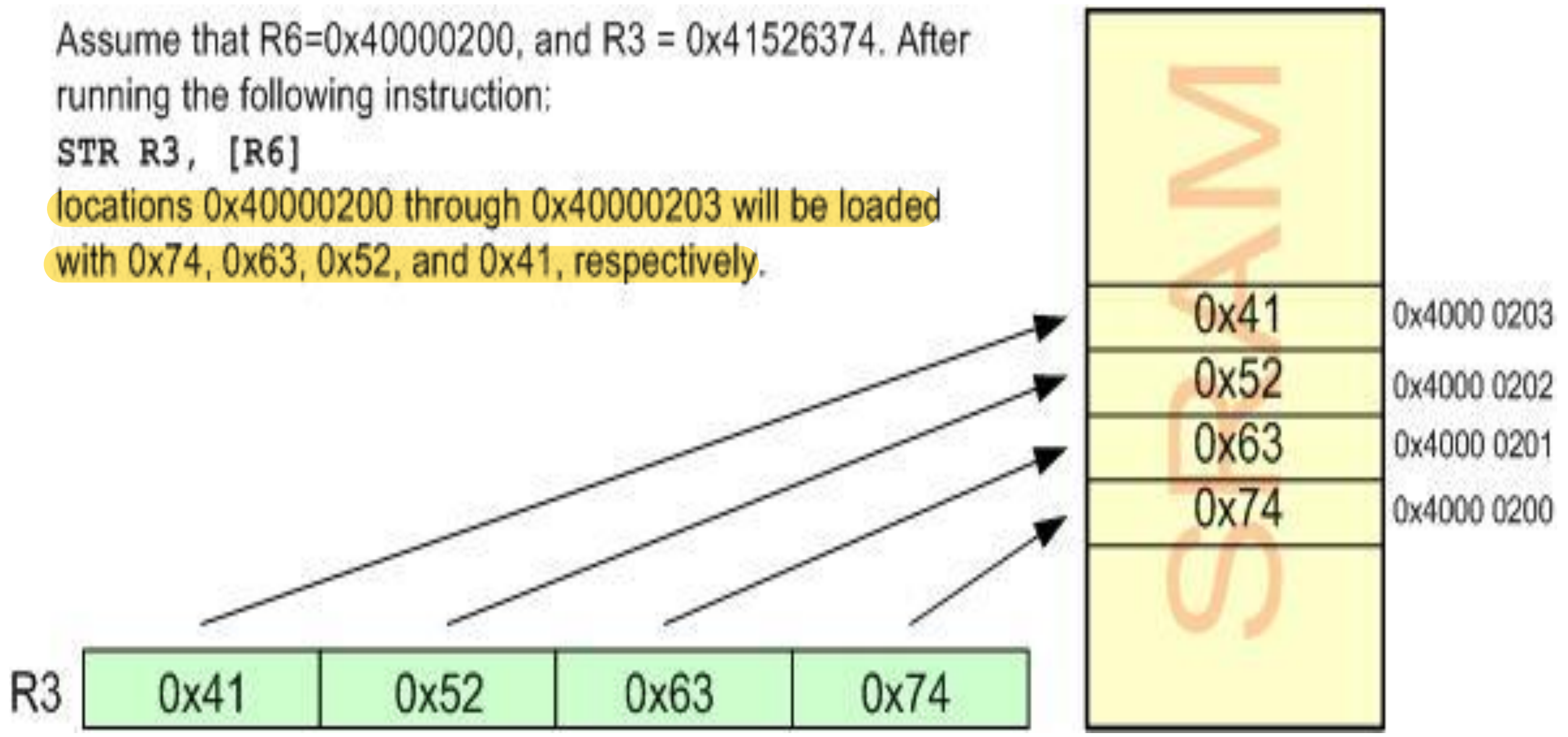
STR Rx,[Rd] instruction

;store register Rx into locations pointed to by Rd

Assume that R6=0x40000200, and R3 = 0x41526374. After running the following instruction:

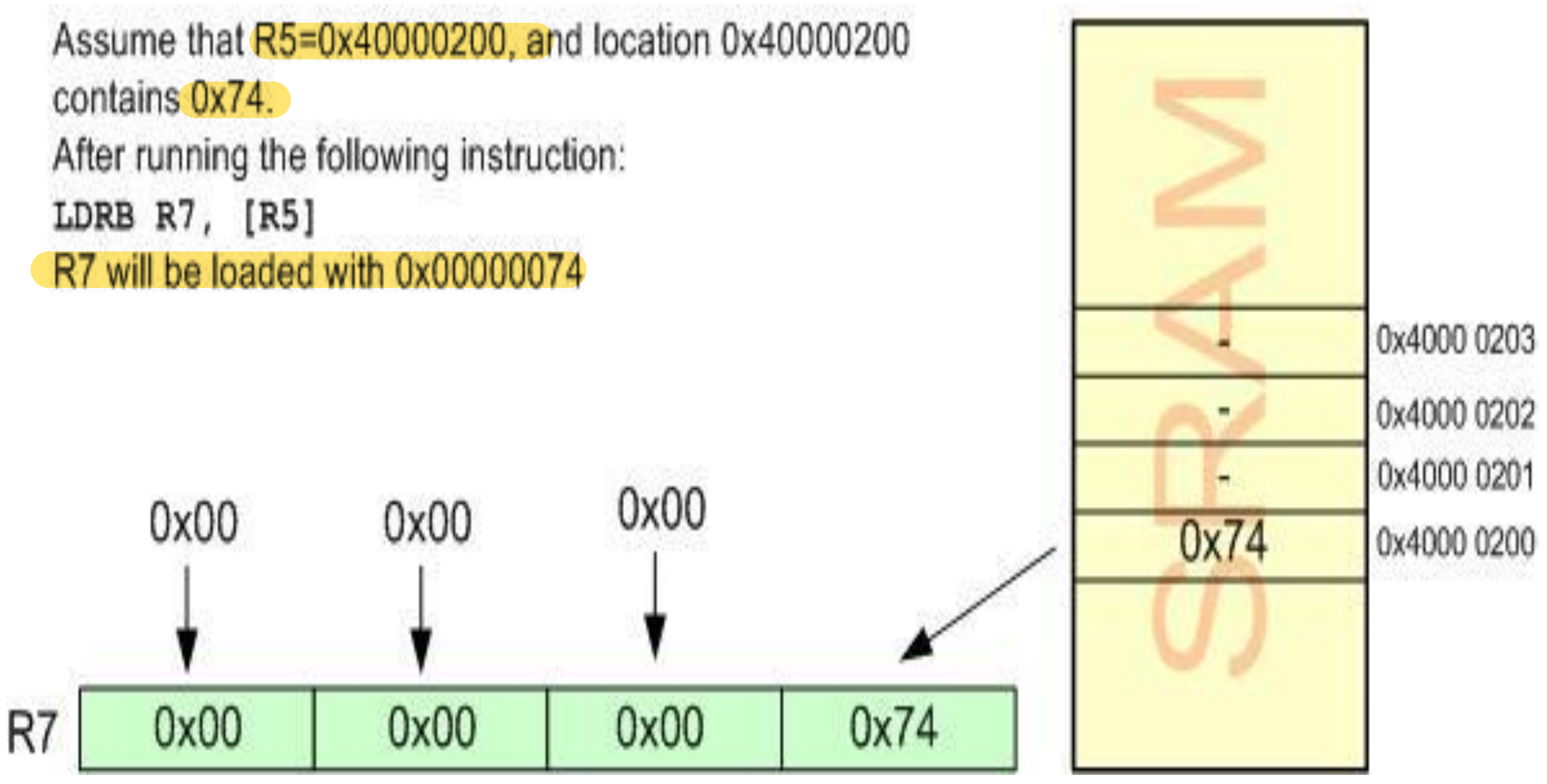
```
STR R3, [R6]
```

locations 0x40000200 through 0x40000203 will be loaded with 0x74, 0x63, 0x52, and 0x41, respectively.



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


Assume that R5=0x40000200, and location 0x40000200 contains 0x74.
After running the following instruction:

```
LDRB R7, [R5]
```

R7 will be loaded with 0x00000074

xecuting the LDRB Instruction

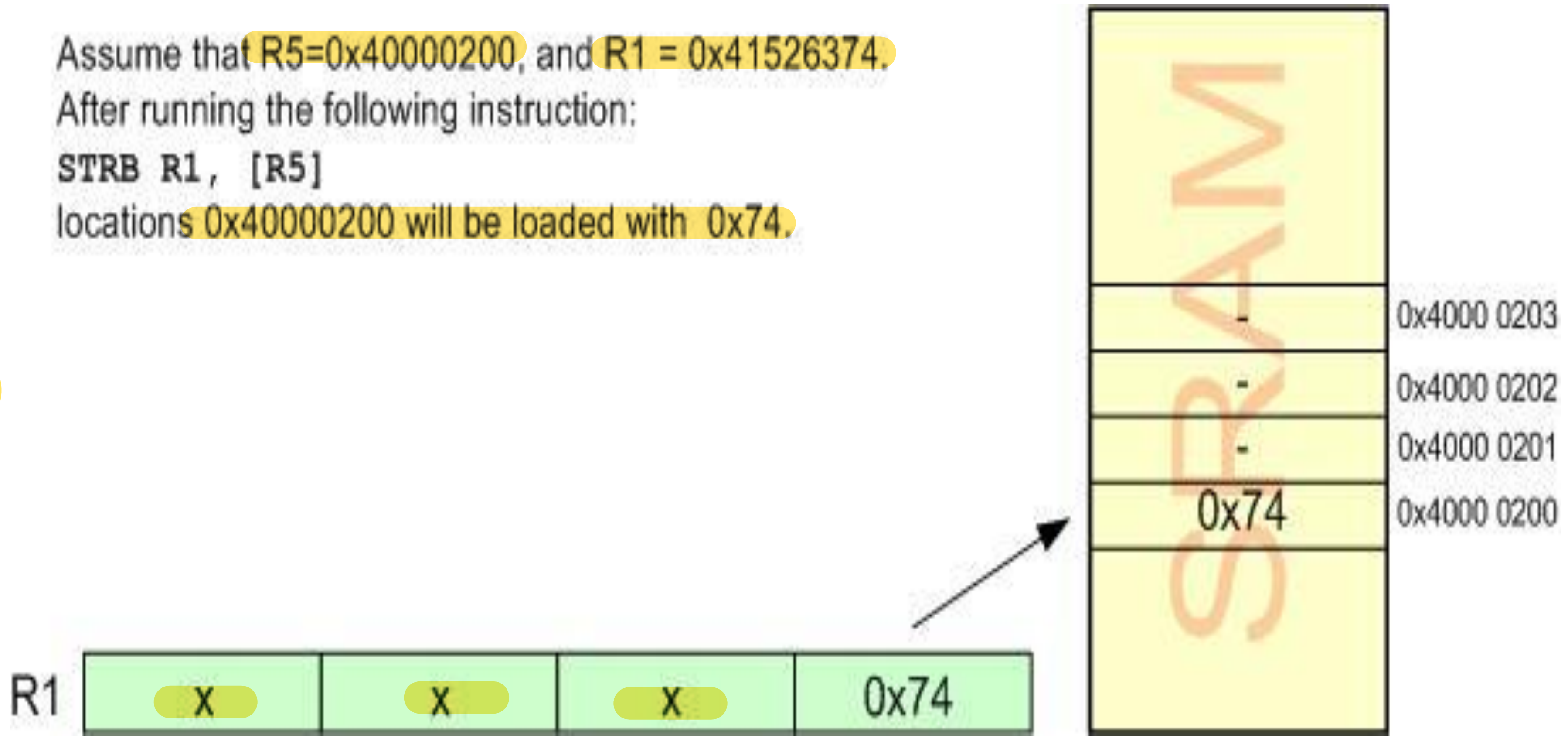
STRB Rx,[Rd] instruction

;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]
```

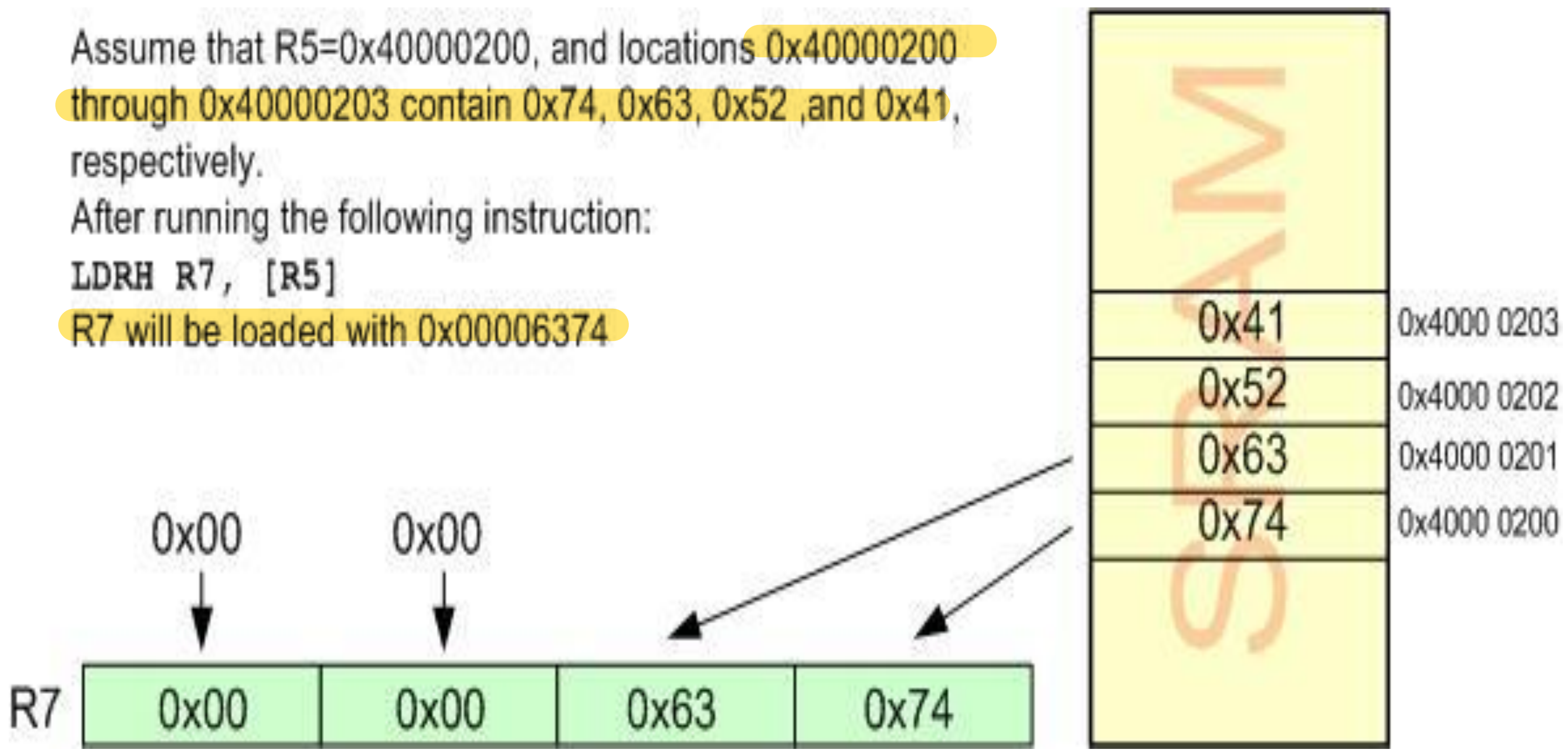
locations 0x40000200 will be loaded with 0x74.



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:

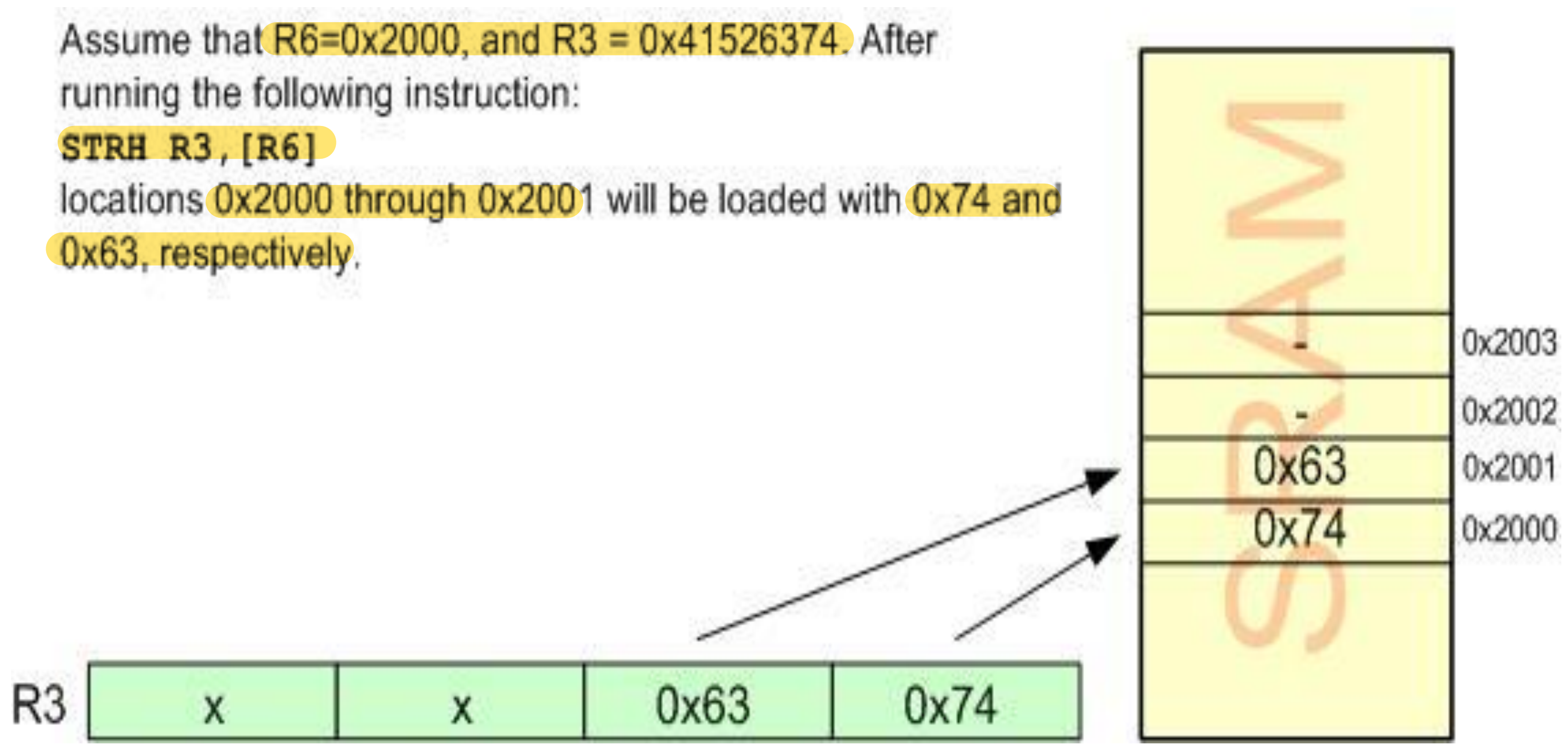
```
LDRH R7, [R5]
```

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

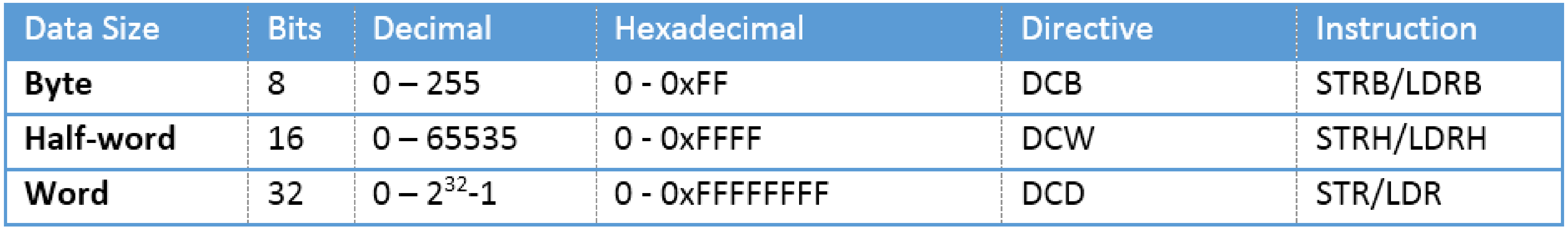


Assume that R6=0x2000, and R3 = 0x41526374. After running the following instruction:

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

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

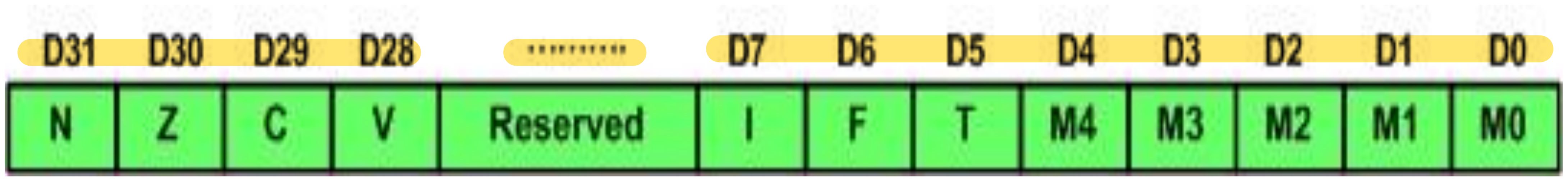
Executing the STRH Instruction



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. See the ARM manual.

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
```

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
```

An Assembly language instruction consists of four fields:

[label] mnemonic [operands] [;comment]

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

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:

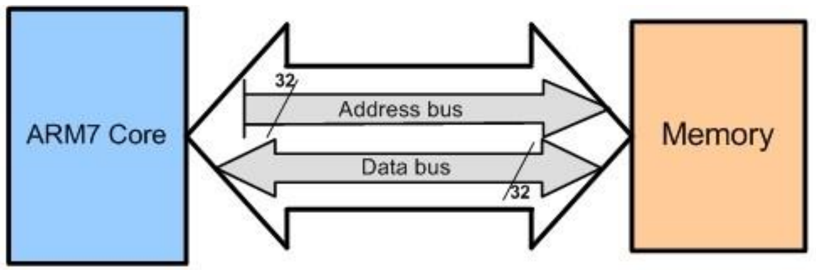
ADR Rn, label

ADR R2, OUR_FIXED_DATA ;point to OUR_FIXED_DATA

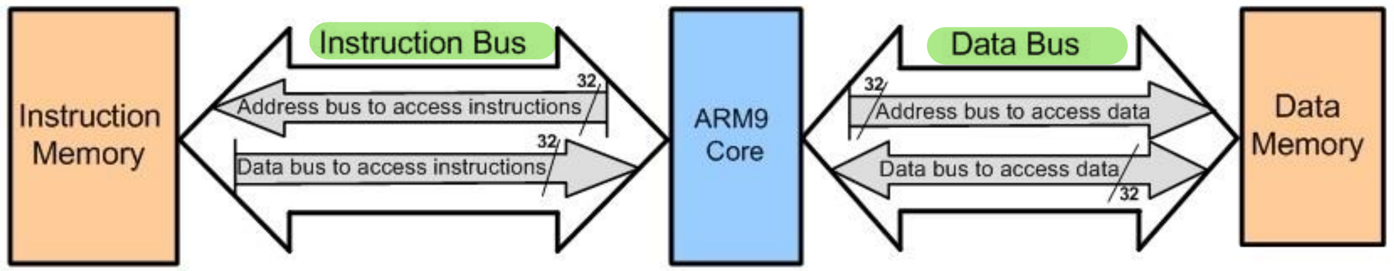
Little endian vs. big endian

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.

Harvard and von Neumann architectures in the ARM



(a) Von Neumann



(b) Harvard

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 data bus, 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]`

MOV Instruction

MOV RD, op2 ; op2 can be a register or 8 bit constant

- MOVT instruction that can load any value in the range 0x0000 to 0xFFFF into the most significant half of a register, without altering the contents of the least significant half.
- The LDR Rd,=const pseudo-instruction generates the most efficient single instruction to load any 32-bit number.
- MOV32 instruction can also be used to load any 32-bit number.

```
MOV32 R2, #5
```

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

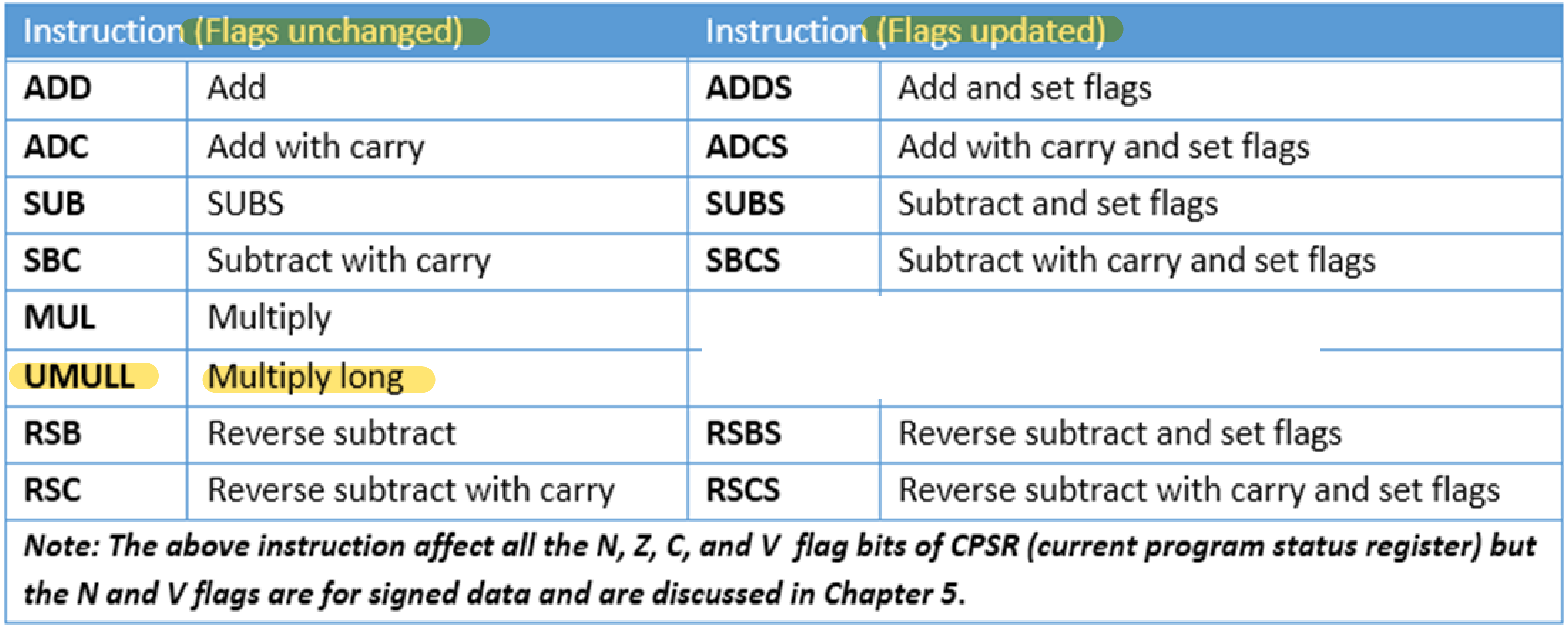
To represent ASCII data in an ARM assembler we use single quotes as follows:

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.

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.

Arithmetic and Logic Instructions



Instruction (Flags unchanged)		Instruction (Flags updated)	
ADD	Add	**ADDS**	Add and set flags
ADC	Add with carry	**ADCS**	Add with carry and set flags
SUB	SUBS	**SUBS**	Subtract and set flags
SBC	Subtract with carry	**SBCS**	Subtract with carry and set flags
MUL	Multiply		
UMULL	Multiply long		
RSB	Reverse subtract	**RSBS**	Reverse subtract and set flags
RSC	Reverse subtract with carry	**RSCS**	Reverse subtract with carry and set flags

Note: The above instruction affect all the N, Z, C, and V flag bits of CPSR (current program status register) but the N and V flags are for signed data and are discussed in Chapter 5.

Arithmetic Instructions and Flag Bits for Unsigned Data

If suffix S is used after the opcode, CPSR register will be effected by the result. Instructions without S executes without having any effect on the flags

```
ADD Rd,Rn,Op2          ;Rd = Rn + Op2
ADC Rd,Rn,Op2          ;Rd = Rn + Op2 + C
```

The instructions ADD and ADC are used to add two operands. The destination operand must be a register. The Op2 operand can be a register or immediate. Remember that memory-to-register or memory-to-memory arithmetic and logic operations are never allowed in ARM Assembly language since it is a RISC processor. The instruction could change any of the Z, C, N, or V bits of the status flag register, as long as we use the ADDS or ADCS instead of ADD or ADC. ADC is used in the addition of multiword data.

SUB Rd,Rn,Op2 ;Rd = Rn - Op2

In ARM SUB instruction is executed as follows:

1. Take the 2’s complement of the subtrahend (Op2 operand).
2. Add it to the minuend (Rn operand).
3. Place the result in destination Rd.
4. Set the carry flag if there is a carry.

These four steps are performed for every SUBS instruction by the internal hardware of the ARM CPU. It is after these four steps that the result is obtained and the flags are set. We must look at the carry flag (not the sign flag) to determine if the result is positive or negative. After the execution of SUBS, if C=1, the result is positive; if C = 0, the result is negative and the destination has the 2’s complement of the result.

Analyze the following instructions:

```
MOV R1,#0x4C ;R1 = 0x4C
MOV R2,#0x6E ;R2 = 0x6E
SUBS R0,R1,R2 ;R0 = R1 – R2
```

Solution:

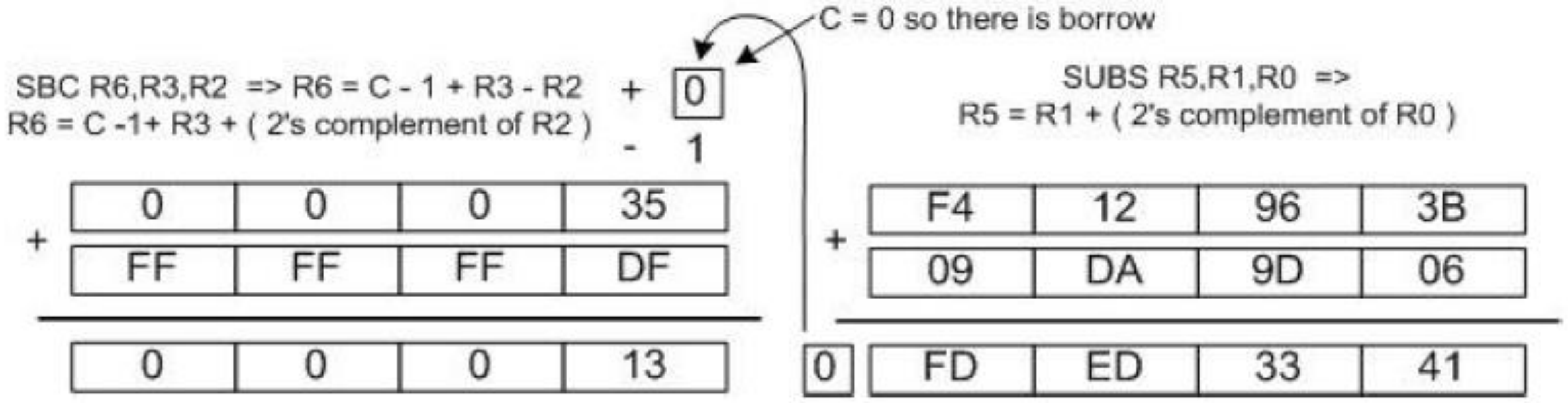
Following are the steps for “SUB R0,R1,R2”:

```
  4C            0000004C
–6E           + FFFFFF92 (2’s complement of 0x6E)
– 22          0 FFFFFFDE (C = 0 step 4) result is negative
```

SBC Rd,Rn,Op2 ;Rd = Rn – Op2 – 1 + C

This instruction is used for subtraction of multiword (data larger than 32-bit) numbers. In ARM the carry flag is not inverted after subtraction and carry flag is invert of borrow. To invert the carry flag while running the subtract with borrow instruction it is implemented
as "Rd = Rn – Op2 – 1 + C"

```
LDR R0,=0xF62562FA              ;R0 = 0xF62562FA,
LDR R1,=0xF412963B              ;R1 = 0xF412963B
MOV R2,#0x21                    ;R2 = 0x21
MOV R3,#0x35                    ;R3 = 0x35
SUBS R5,R1,R0           ;R5 = R1 – R0
            ; =0xF412963B – 0xF62562FA, and C = 0
SBC R6,R3,R2                    ;R6 = R3 – R2 – 1 + C
            ; = 0x35 – 0x21 – 1 + 0 = 0x13
```



RSB Rd,Rn,Op2 ;Rd = Op2 – Rn

This instruction can be used to get 2’s complement of a 32-bit operand

```
MOV R1,#0x1  ;R1=1
RSB R0,R1,#0  ;R0= 0 – R1 = 0 – 1
```

This is one way to get a fixed value of 0xFFFFFFFF in a register. R0=0xFFFFFFFF.

RSC Rd,Rn,Op2 ;Rd = Op2 – Rn – 1 + C

This instruction can be used to get the 2's complement of the 64-bit operand.

Show how to create 2's complement of a 64-bit data in R0 and R1 register. The R0 hold the lower 32-bit.

Solution:

```
LDR R0,=0xF62562FA          ;R0 = 0xF62562FA
LDR R1,=0xF812963B          ;R1 = 0xF812963B
RSB R5,R0,#0                ;R5 = 0 – R0
                      ; = 0 – 0xF62562FA = 9DA9D06 and C = 0
RSC R6,R1,#0                ;R6 = 0 – R1 – 1 + C
                      ; = 0 – 0xF812963B – 1 + 0 = 7ED69C4
```

Multiplication of unsigned numbers in ARM

Instruction	Source 1	Source 2	Destination	Result
MUL	Rn	Op2	Rd (32 bits)	Rd=Rn×Op2
UMULL	Rn	Op2	RdLo, RdHi (64 bits)	RdLo:RdHi=Rn×Op2

Note 1: *Using MUL for word × word multiplication preserves only the lower 32 bit result in Rd and the rest are dropped. If the result is greater than 0xFFFFFFFF, then we must use UMULL (unsigned Multiply Long) instruction.*

Note 2: *In some CPUs the C flag is used to indicate the result is greater than 32-bit but this is not the case with ARM MUL instruction.*

```
MUL Rd,Rn,Op2                ;Rd = Rn × Op2
```

All the operands must be in registers. Immediate value is not allowed as an operand.

UMULL (unsigned multiply long)

```
UMULL RdLo,RdHi,Rn,Op2          ;RdHi:RdLo = Rn × Op2
```

In unsigned long multiplication, the operands must be in registers. After the multiplication, the destination registers will contain the result. Notice that the left most register, RdLo, will hold the lower word and the higher portion beyond 32-bit is saved in the second register, RdHi.

Multiply and Accumulate Instruction in ARM

```
MLA Rd,Rm,Rs,Rn          ;Rd = Rm × Rs + Rn
```

In multiplication and add, the operands must be in registers. After the multiplication and add, the destination register will contain the result.

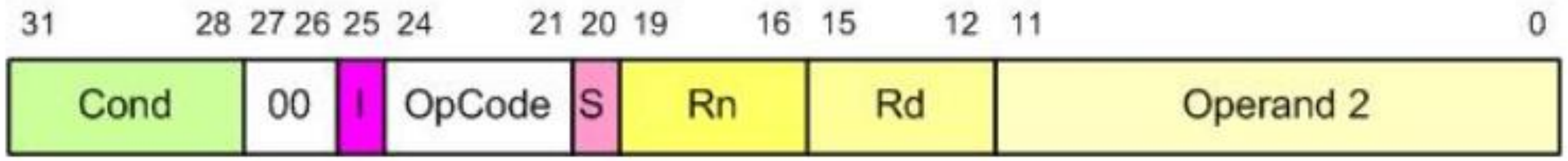
Notice that multiply and add can produce a result greater than 32-bit, if the MLA instruction is used, the destination register will hold the lower word (32-bit) and the portion beyond 32-bit is dropped.

UMLAL RdLo,RdHi,Rn,Op2   ;RdHi:RdLo = Rn × Op2 + RdHi:RdLo

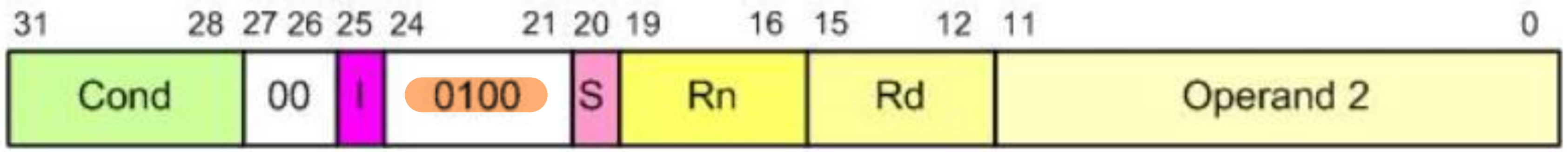
In multiplication and add, the operands must be in registers. Notice that the addend and the high word of the destination use the same registers, the two left most registers in the instruction. It means that the contents of the registers which have the addend will be changed after execution of UMLAL instruction.

Division of unsigned numbers in ARM

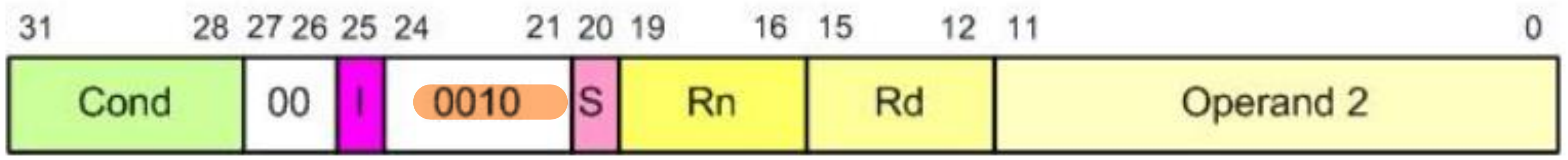
It is done using repeated subtraction



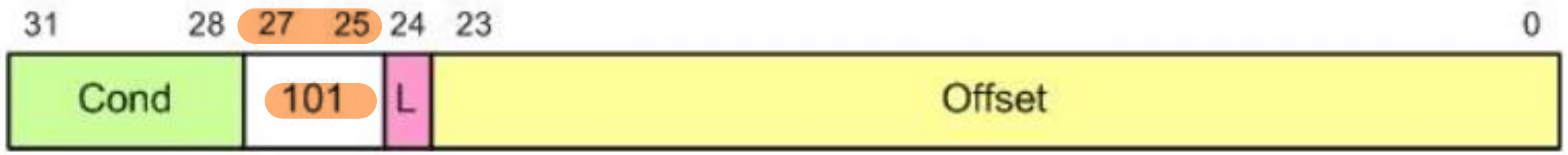
General Formation of Data Processing Instructions



ADD Instruction Formation



SUB Instruction Formation



Branch Instruction Formation

Logic Instructions

Instruction (Flags Unchanged)	Action	Instruction (Flags Changed)	Hexadecimal
AND	ANDing	**ANDS**	Anding and set flags
ORR	ORRing	**ORS**	Oring and set flags
EOR	Exclusive-ORing	**EORS**	Exclusive Oring and set flags
BIC	Bit Clearing	**BICS**	Bit clearing and set flags

```
AND Rd, Rn, Op2          ;Rd = Rn ANDed Op2
ORR Rd, Rn, Op2          ;Rd = Rn ORed Op2
EOR Rd,Rn,Op2             ;Rd = Rn Ex-ORed with Op2
BIC Rd,Rn,Op2            ;clear certain bits of Rn specified by the Op2 and place the result in Rd
```

In all these instructions Op2 can be register or immediate value

The BIC (bit clear) instruction is used to clear the selected bits of the Rn register. The selected bits are held by Op2. The bits that are HIGH in Op2 will be cleared and bits with LOW will be left unchanged. In reality, the BIC instruction performs AND operation on Rn register with the complement of Op2 and places the result in destination register

```
MOV R1,#0x0F
MOV R2,#0xAA
BIC R3,R2,R1            ;now R3 = 0xAA ANDed with 0xF0 = 0xA0
```

```
MVN Rd, Rn              ;move negative of Rn to Rd
```

The MVN (move negative) instruction is used to generate one's complement of an operand. For example, the instruction "MVN R2,#0" will make R2=0xFFFFFFFF. Look at the following example:

```
LDR R2,=0xAAAAAAAA              ;R2 = 0xAAAAAAAA
MVN R2,R2                       ;R2 = 0x55555555
```

It must be noted that the instruction "MVN Rd,#0" is widely used to load the fixed value of 0xFFFFFFFF into destination register.

Comparison of unsigned numbers

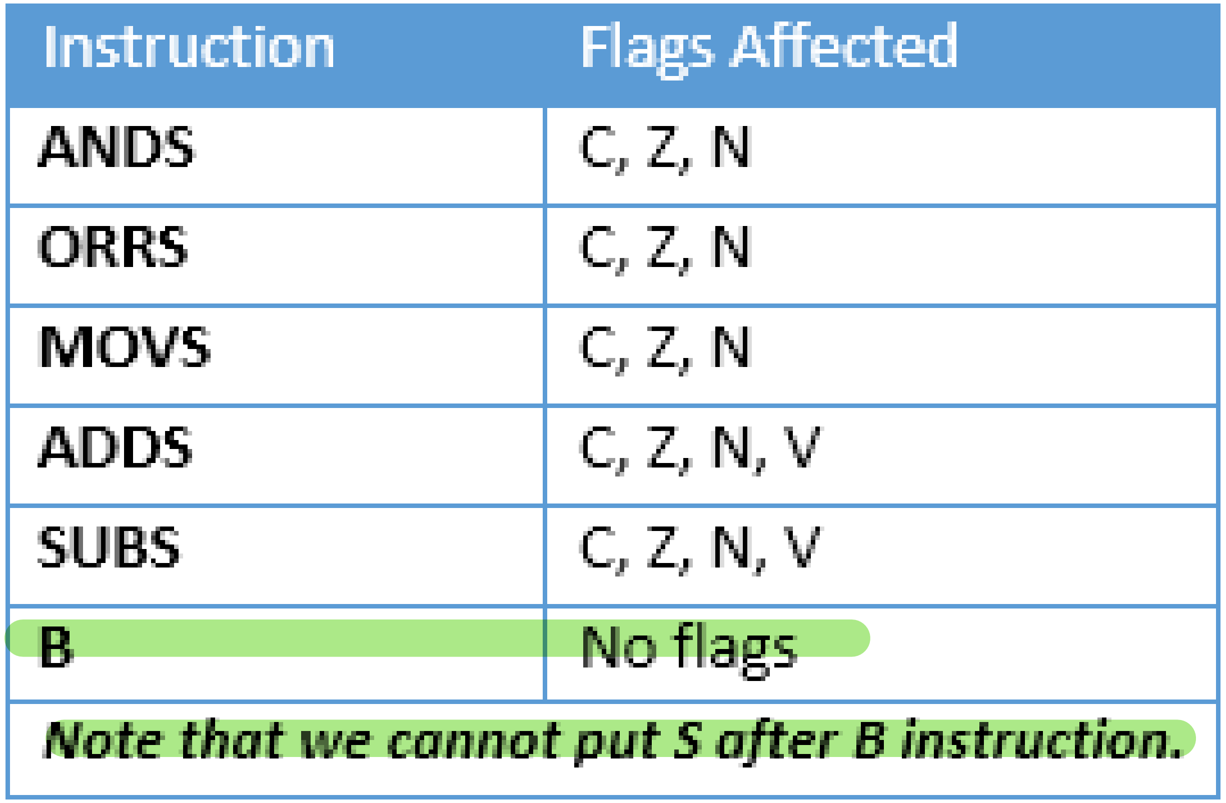
CMP Rn,Op2 ;compare Rn with Op2 and set the flags

- The CMP instruction compares two operands and changes the flags according to the result of the comparison. The operands themselves remain unchanged. The second source operands can be a register or an immediate value not larger than 0xFF.
- It must be emphasized that “CMP Rn,Op2” instruction is really a subtract operation. Op2 is subtracted from Rn (Rn – Op2) and the result is discarded and flags are set accordingly.
- Although all the C, S, Z, and V flags reflect the result of the comparison, only C and Z are used for unsigned numbers, as shown below:

	C	Z
Rn > Op2	1	0
Rn = Op2	1	1
Rn < Op2	0	0

```
        LDR R1,=0x35F          ;R1 = 0x35F

        LDR R2,=0xCCC          ;R2 = 0xCCC
        CMP R1,R2              ;compare 0x35F with 0xCCC
        BCC OVER               ;branch if C = 0
        MOV R1,#0              ;if C = 1, then clear R1
OVER    ADD R2,R2,#1           ;R2 = R2 + 1 = 0xCCC + 1 = 0xCCD
```



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.	

Flag Bits Affected by Different Instructions

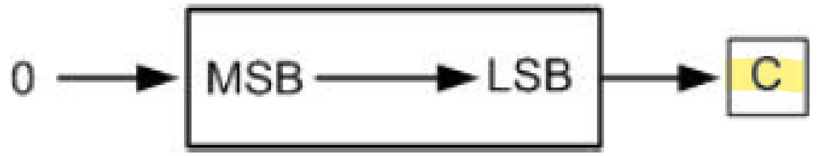
Rotate and Barrel Shifter

we can perform the shift and rotate operations as part of other instructions such as MOV

The process instructions can be used in one of the following forms:
1. opcode Rd, Rn, Rs (e.g. ADD R1,R2,R3)
2. opcode Rd, Rn, immediateValue (e.g. ADD R2,R3,#5)

ARM is able to shift or rotate the second argument before using it as the argument.

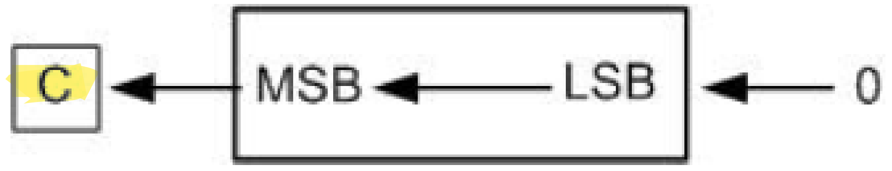
LSR   Logical Shift Right



```
MOV R0,#0x9A                ;R0 = 0x9A
MOVS R1,R0,LSR #3      ;shift R0 to right 3 times
                            ;and then move (copy) the result to R1
```

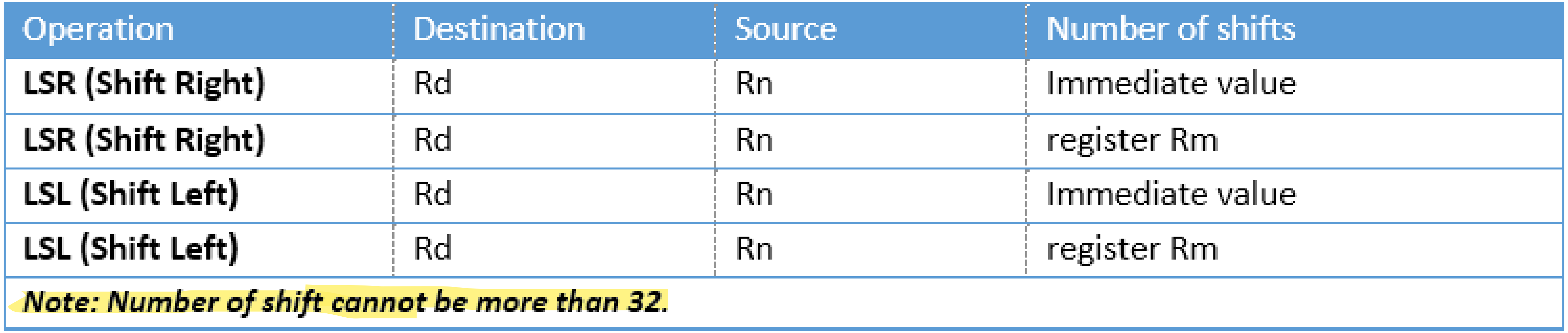
```
MOV R0,#0x9A
MOV R2,#0x03
MOV R1,R0,LSR R2      ;shift R0 to right R2 times and move the result to R1
```

LSL Logical Shift Left



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

LDR R1,=0x0F000006
MOV R0,#0x08
MOV R2,R1,LSL R0
```

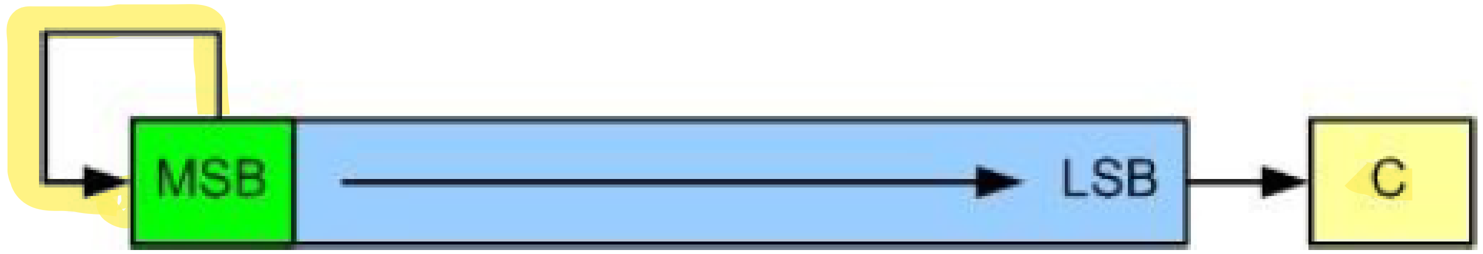


Operation	Destination	Source	Number of shifts
LSR (Shift Right)	Rd	Rn	Immediate value
LSR (Shift Right)	Rd	Rn	register Rm
LSL (Shift Left)	Rd	Rn	Immediate value
LSL (Shift Left)	Rd	Rn	register Rm
Note: Number of shift cannot be more than 32.			

Logic Shift operations for unsigned numbers in ARM

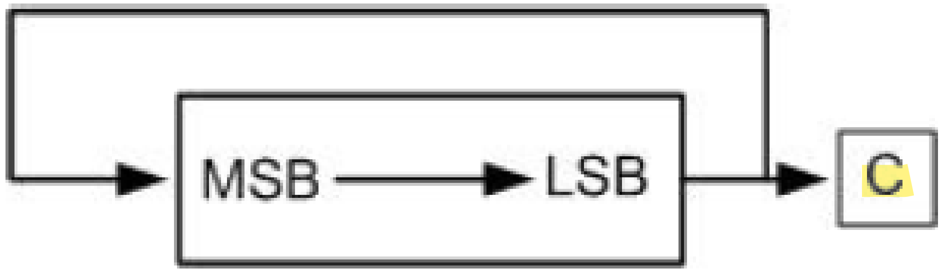
ASR (arithmetic shift right)

MOV Rn,Op2, ASR count



```
MOV R0,#-10          ;R0 = -10 = 0xFFFFFFF6
MOV R3,R0,ASR #1     ;R0 is arithmetic shifted right once
                          ;R3 = 0xFFFFFFFB = -5
```

ROR (rotate right)



```
MOV R1,#0x36
MOVS R1,R1,ROR #1
```

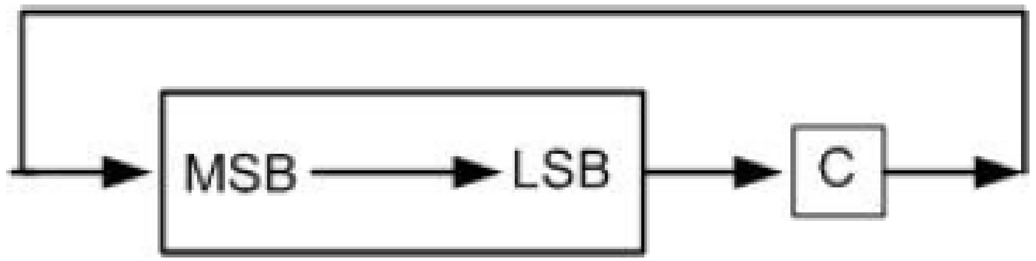
```
MOV R1,#0x36
MOV R0,#3
MOVS R1,R1,ROR R0
```

Rotate left

There is no rotate left option in ARM since one can use the rotate right (ROR) to do the job. That means instead of rotating left n bits we can use rotate right 32–n bits to do the job of rotate left. Using this method does not give us the proper carry if actual instruction of ROL was available

```
LDR R0,=0x00000072
;R0 = 0000 0000 0000 0000 0000 0000 0111 0010
MOVS R0,R0,ROR #31
;R0 = 0000 0000 0000 0000 0000 0000 1110 0100 C=0
```

RRX rotate right through carry



In RRX the LSB is moved to C and C is moved to the MSB. In reality, C flag acts as if it is part of the operand. That means the RRX is like 33-bit register since the C flag is 33rd bit. The RRX takes no arguments and the number of times an operand to be rotated is fixed at one.

```
;assume C=0
MOV R2,#0x26
;R2 = 0000 0000 0000 0000 0000 0000 0010 0110
MOVS R2,R2,RRX
;R2 = 0000 0000 0000 0000 0000 0000 0001 0011 C=0
```

Operation	Destination	Source	Number of Rotates
ROR (Rotate Right)	Rd	Rn	Immediate value
ROR (Rotate Right)	Rd	Rn	register Rm
RRX (Rotate Right Through Carry)	Rd	Rn	1 bit

Rotate operations for unsigned numbers in ARM

Shift and Rotate Instructions

In ARM Cortex M all the above shift and rotate instructions can be used as independent instructions

```
LSL Rd, Rm, Rn
LSL R0,R2,#8
LSL R2,R0,R1

LSLS Rd, Rm, Rn
LSLS R0,R2,#8
LSLS R2,R0,R1
```

Similarly, LSR and ROR

```
LSR Rd, Rm, Rn
LSRS Rd, Rm, Rn
ROR Rd, Rm, Rn
RORS Rd, Rm, Rn
```

```
RRX Rd,Rm                    ;Rd=rotate Rm right 1 bit position
```

Function: Each bit of Rm register is shifted from left to right one bit. The RRX does not update the flags.

```
LDR R2,=0x00000002
RRX R0,R2                              ;R0=R2 is shifted right one bit
                                       ;now, R0=0x00000001
```

RRXS Rotate Right with extend (update the flags)

```
RRXS Rd,Rm                   ;Rd=rotate Rm right 1 bit position
```

Function: Each bit of Rm register is shifted from left to right one bit. The RRXS updates the flags.

```
LDR R2,=0x00000002
RRXS R0,R2                   ;R0=R2 is shifted right one bit
                                       ;now, R0=0x00000001
```

EQ Z set equal
NE Z clear not equal
CS/HS C set unsigned higher or same
CC/LO C clear unsigned lower
MI N set negative
PL N clear positive or zero
VS V set overflow
VC V clear no overflow
HI C set and Z clear unsigned higher
LS C clear or Z set unsigned lower or same
GE N equals V signed greater or equal
LT N not equal to V signed less than
GT Z clear AND (N equals V) signed greater than
LE Z set OR (N not equal to V) signed less than or equal
AL (ignored) always (usually omitted)

Instruction	Flags Affecting the branch
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

ARM Branch (Jump) Instructions Using Flag Bits

Conditional Execution

- If we do not add a condition after an instruction, it will be executed unconditionally because the default is not to check the flags and execute unconditionally.
- If we want an instruction to be executed only when a condition is met, we put the condition syntax right after the instruction.
- This feature of ARM allows the execution of an instruction conditionally based on the status of Z, C, V and N flags. To do that, the ARM instructions have set aside the most significant 4 bits of the instruction field for the conditions. The 4 bits gives us 16 possible conditions.

To make an instruction conditional, simply we put the condition syntax in front of it.

```
MOV R1,#10    ;R1 = 10
MOV R2,#12    ;R2 = 12
CMP R2,R1     ;compare 12 with 10, Z = 0 because they are not equal
MOVEQ R4,#20 ;this line is not executed because the condition EQ is not met

CMP R1,#0     ;compare R1 with 0
ADDNE R1,R1,#10               ;this line is executed if Z = 0 (if in the last CMP
                           ;operands were not equal)
```

Note that we can add both S and condition to syntax of an instruction. It is common to put S after the condition. See the following examples:

```
ADDNES R1,R1,#10
;this line is executed and set the flags if Z = 0
```

BCD and ASCII Conversion

(1) unpacked BCD

(2) packed BCD.

(3) ASCII numbers

BL (branch and link)

- Subroutines are often used to perform tasks that need to be performed frequently.
- In the ARM there is only one instruction for call and that is BL (branch and link).
- To use BL instruction for call, we must leave the R14 register unused since that is where the ARM CPU stores the address of the next instruction where it returns to resume executing the program.
- ARM automatically saves in the link register (LR), the R14, the address of the instruction immediately below the BL.

- When a subroutine is called by the BL instruction, control is transferred to that subroutine, and the processor saves the PC (program counter) in the R14 register and begins to fetch instructions from the new location.

- After finishing execution of the subroutine, we must use “BX LR“ instruction to transfer control back to the caller. Every subroutine needs “BX LR” as the last instruction for return address.

Branching beyond 32M byte limit

- To branch beyond the address space of 32M bytes, we use BX (branch and exchange) instruction.
- The "BX Rn" instruction uses register Rn to hold target address. Since Rn can be any of the R0–R14 registers and they are 32-bit registers, the "BX Rn" instruction can land anywhere in the 4G bytes address space of the ARM.
- In the instruction "BX R2" the R2 is loaded into the program counter (R15) and CPU starts to fetch instructions from the target address pointed to by R15, the program counter.

TST (Test)

```
TST Rn,Op2              ;Rn AND with Op2 and flag bits are updated
```

The TST instruction is used to test the contents of register to see if any bit is set to HIGH.
After the operands are ANDed together the flags are updated.
After the TST instruction if result is zero, then Z flag is raised and one can use BEQ (branch equal) to make decision.
example:

```
          MOV R0,#0x04        ;R0=00000100 in binary
          LDR R1,=myport      ;port address
OVER  LDRB R2,[R1]   ;load R2 from myport
          TST R2,R0           ;is bit 2 HIGH?
          BEQ OVER            ;keep checking
```

In TST, the Op2 can be an immediate value of less than 0xFF.

example:

```
            LDR R1,=myport      ;port address
OVER  LDRB R2,[R1]              ;load R2 from myport
            TST R2,#0x04              ;is bit 2 HIGH?
            BEQ OVER            ;keep checking
```

TEQ (test equal)

- TEQ Rn,Op2 ;Rn EX-ORed with Op2 and flag bits are set

- The TEQ instruction is used to test to see if the contents of two registers are equal.

- After the source operands are Ex-ORed together the flag bits are set according to the result.

- After the TEQ instruction if result is 0, then Z flag is raised and one can use BEQ (branch zero) to make decision.

```
        TEMP EQU 100
        MOV R0,#TEMP            ;R0 = Temp
        LDR R1,=myport          ;port address
OVER    LDRB R2,[R1]            ;load R2 from myport
        TEQ R2,R0               ;is it 100?
        BNE OVER                ;keep checking
```