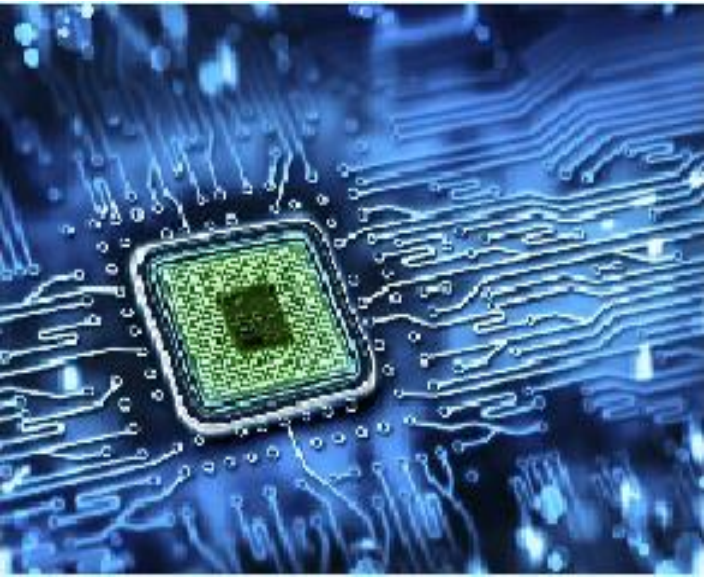
# 16CS402 - Embedded Systems

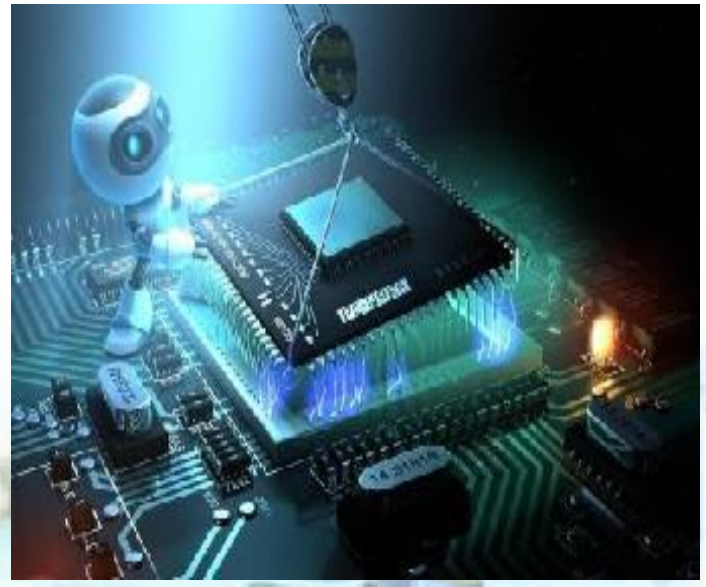
VIGNAN'S
Foundation for Science, Technology & Research
(Deemed to be UNIVERSITY)
-Estd. u/s 3 of UGC Act 1956

## 4th Year 1st Semester



## CSE

## UNIT - IV




**Dr. V. VIJAYARAGHAVAN, M.E., Ph.D.,**

Assistant Professor - ECE, VFSTR, Guntur, AP.

# ARM Programming

- Assembly Programming, General Structure of Assembly Language
- Writing Programs
- Branch, Load and Store Instructions
- Read-Only and Read/Write Memory
- Multiple Register Load and Store

# Assembly Programming

**There are two items used in Assembly language Programming**

1. **Instructions**
2. **Directives (Assembler directives or Pseudo Instructions)**

**Instruction:**
**In ARM Assembly language line will contains an instruction, directive or pseudo instruction.**

## 10.13 | Starting Assembly Language Programming

If you have any previous experience of assembly language programming, you will kn
that there are two items used therein—instructions and directives—the former are ex
cutable statements which are 'executed' by the processor. The latter, that is, directives
non-executable statements relating to the assembler. They are used to give the assemb
necessary information to perform the assembly process smoothly. For some processo
directives are also called pseudo instructions. For ARM, pseudo instructions are spec
directives issued to the processor which causes certain instructions to be executed. Th
they are also executable statements. **Thus for ARM, an assembly language line will co
tain an instruction, directive or pseudo instruction.**

Writing and testing a program for ARM is done in a computer, usually a PC, wh
is called the host computer. The host computer should have the program developm
tools for ARM. Since the program written in ARM assembly language is assembl

# Assembly Programming

## 10.13 | Starting Assembly Language Programming

If you have any previous experience of assembly language programming, you will know that there are two items used therein—instructions and directives—the former are executable statements which are 'executed' by the processor. The latter, that is, directives are non-executable statements relating to the assembler. They are used to give the assembler necessary information to perform the assembly process smoothly. For some processors, directives are also called pseudo instructions. For ARM, pseudo instructions are special directives issued to the processor which causes certain instructions to be executed. Thus, they are also executable statements. **Thus for ARM, an assembly language line will contain an instruction, directive or pseudo instruction.**

Writing and testing a program for ARM is done in a computer, usually a PC, which is called the host computer. The host computer should have the program development tools for ARM. Since the program written in ARM assembly language is assembled in a PC which has a different processor (usually some version of Pentium), the process is called 'cross assembly'. After the program in tested, it is converted to a hex file and burned into our processor (i.e. ARM).

The output of an assembly or compilation process has atleast two areas.

i) A code area. This is usually a read-only area.
ii) A data area. This is usually a read-write area.

The default area for code is Read-Only and for data it is Read-Write.

Let's understand some fundamental directives first.

# Assembly Programming…….

### 10.13.1 | The AREA Directive

The first thing we do when we start assembly language programming is to define an area. There is a directive named 'AREA' for this. This directive names the area and sets its attributes. The attributes are placed after the name, separated by commas.

***Example***

```
AREA        SORT, CODE, READ ONLY
AREA        TABLE, DATA
```

The first area defined above is given the name SORT; it contains a code, and is read only. The word 'read only' is optional. The second AREA directive has the name TABLE and it contains data and though not mentioned will correspond to the Read Write area (as it is a data area).

### 10.13.2 | The ENTRY Directive

The ENTRY directive marks the first instruction to be executed within an application. Because an application cannot have more than one entry point, the ENTRY directive can appear in only one of the source modules.

### 10.13.3 | The END Directive

This directive tells the assembler to stop reading. Anything written after the END directive will be ignored by the assembler. So every assembly language source module must finish with an END directive, on a line by itself.

# General Structure of Assembly Language

## 10.14 | General Structure of an Assembly Language Line

The general form of source lines in assembly language is:

```
{label} {instruction|directive|pseudo-instruction} {;comment}
```

Some points to keep in mind are listed as follows:

i) Instructions, pseudo-instructions and directives must be preceded by a white space, such as a space or a tab, even if there is no label. This means that they should not be written in the label space (extreme left of the line).
ii) Instruction mnemonics and register names can be written in uppercase or lowercase, but not mixed.
iii) Labels are symbols that represent addresses. The address given by a label is calculated during assembly. The assembler calculates the address of a label relative to the origin of the area where the label is defined. Assigning labels eases the programmer's burden as he does not have to concern himself with numerical values. The location counter in the assembler keeps on incrementing as labels are encountered.

Typical assembly language lines are

```
NOO     MOV R1, R2, LSL #2      ;copy the content of R2 left shifted, to R1
NUMS    DCW 2354, 5678          ;define two data half words
```

In the above two lines, NOO and NUMS are the labels, MOV (with operands) is an instruction and DCW is a directive, which is explained in the following section.

# General Structure of Assembly Language…….

## 10.14.1 | Directives for Defining Data

Before we go deep into programming, we need to understand a few directives of the assembler which define and describe different kinds of data. Data which is used in a program can be bytes or words or half words. We define data and assign labels to their corresponding addresses. Defining data implies allocating space for data. The space we allocate corresponds to memory addresses, which are identified by labels. Data, when stored in memory is defined accordingly, using directives.

DCB defines data byte, DCW defines 16 bits or a half word and DCD defines a word (32 bits).

***Examples***

```
NUMS    DCB    9, 82, 71
NUMB    DCW    0x6787, 0x4564
NUMBR   DCD    0x00000123, 0x67890900
```

In the above, the first line shows data which are bytes. The first byte 9, has the address NUMS, 82 has the address NUMS + 1, and 71 has the address NUMS + 1.

The second line has address NUMB for the first half word, and NUMB + 2 for the second half word. In the last case, the addresses of the words are NUMBR and NUMBR + 4.

Keep in mind that a byte needs only one address, half word needs two, and a word requires four memory addresses.

# General Structure of Assembly Language…….

## 10.14.2 | The EQU Directive

This is a frequently used directive, and is used to equate a numeric constant to a label. The constant may be data or address. Examples are as follows:

```
FACTR       EQU   35
BASE_ADDR   EQU   0x40000000
```

## 10.14.3 | Constants Allowed

The constants that can be used are numbers (decimals, hex or having any other base), characters, strings and Boolean

- Decimal., say, 346, 6748, etc.
- Hexadecimal. For example, 0x12345678, 0xFCE45, etc.
- *n_xxx* where: *n* is a base between 2 and 9 and *xxx* is a number in that base
- Characters: They are to be enclosed within single quotes, 'e', 'R', etc.
- Strings: They are characters enclosed within double quotes "mine", "non", etc.
- Boolean: TRUE or FALSE

# General Structure of Assembly Language…….

## 10.14.4 | The RN Directive

The names of the general purpose registers have been introduced as R0, R1, R2, etc.

When we use them for loading operands, it is possible that we have a confusion as to which data has been loaded into which register. To ease out this problem, there is a way for giving variable names to registers. Suppose we need to use R0 for loading the value of X, and R1 for loading Y, we use the directive RN as follows:

```
X RN 0
Y RN 1
```

This method can be used for any of the registers.

```
DAT1    RN      8
DET     RN      10
```

# Writing Assembly Language Programs

## 10.15 | Writing Assembly Programs

Now, that we have got used to writing some instructions, let's get down to writing a complete program. This will let us get a feel of the programming process, after which we can learn more important instructions and write bigger and better programs.

**Example 10.6**

Write a program to find the sum of 3X + 4Y + 9Z, where X = 2, Y = 3 and Z = 4.

***Solution***

```
            AREA SUMM, CODE, READONLY
X RN 1                              ;register R1 is named X
Y RN 2                              ;register R2 is named Y
Z RN 3                              ;register R3 is named Z
            ENTRY
            MOV X,#2                ;load X = 2 into register R1
            MOV Y,#3                ;load Y = 3 into register R2
            MOV Z,#4                ;load Z = 4 into register R3
            ADD R1,R1,R1,LSL#1      ;R1 = 3X
            MOV R2,R2,LSL#2         ;R2 = 4Y
            ADD R3,R3,R3,LSL#3      ;R3 = 9Z
            ADD R1,R1,R2            ;R1 = R1+R2 i.e. 3X+4Y
            ADD R1,R1,R3            ;R1 = R1+R3 i.e. 3X+4Y+9Z
STOP        B   STOP                ;continue branching at STOP
            END                     ;end of the assembly file
```

# Writing Assembly Language Programs…….

Since this is the first complete program we are writing, it is important to make some observations regarding it.

i) SUMM is the name of the code AREA defined. The term 'Read only' is optional, as by default, a code area corresponds to the Read only memory only.
ii) As assembly language line has the label field at the left, and the opcode field to its right. For the Keil assembler, you may find that writing the word 'AREA' in the label field will generate an error message. But the directive RN is to be positioned in the label field itself. No instructions should be in the label field.
iii) The ENTRY directive should be followed by an instruction or pseudo instruction.
iv) The program involves multiplication and addition. Since the multiply instruction is a 'complex' one involving the use of special hardware (more power dissipation and more clock cycles), it is not used. Instead multiplication is achieved by the use of shift and add instructions.
v) The last instruction is an unconditional branch instruction (mnemonic 'B') and it continually branches to the same label STOP. This is done so that control does not go any instruction beyond this location. Any code has to finally be burned into ROM. Many embedded programs have their last line as this kind of self branching, since we don't want the next memory locations in code memory to be accessed.

# Branch Instructions

## 10.16 | Branch Instructions

For any processor, branching is a very important operation. The power to change the sequence of execution is obtained by branching, which may be conditional or unconditional. Most processors have 'jump' and 'call' instructions for changing the sequence of execution. ARM does all this by different forms of a 'branch' instruction. It has the mnemonic 'B' for branch. The four different forms of branch instruction are given in Table 10.13

Let's see the usage of each of them.

Branching implies transferring control to a new memory location which is expressed as a 'label'. Hence the format of any branch instruction is B label. Branching is made conditional by appending the mnemonic B with the necessary condition.

**Table 10.13** | List of Branch Instructions

| Mnemonic | Instruction |
|---|---|
| B | Branch |
| BL | Branch and link |
| BX | Branch and Exchange |
| BLX | Branch Exchange with link |


**Figure 10.16 Format of a branch instruction**

# Conditional Execution:

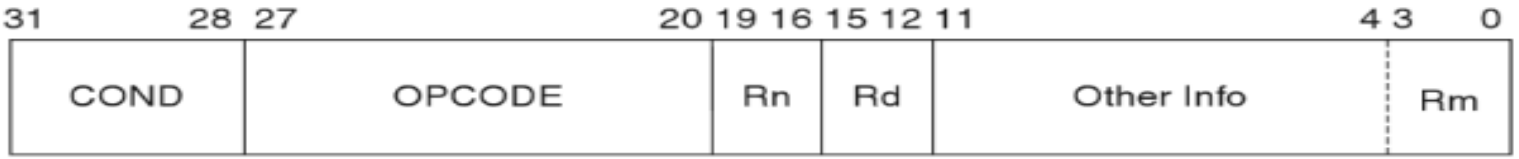



**Figure 10.15** | Format of a typical instruction

**Table 10.7** | List of Conditions, Codes and Corresponding Flag Status

| Cond | Mnemonic | Meaning | Condition Flag State |
|---|---|---|---|
| 0000 | EQ | Equal | Z = 1 |
| 0001 | NE | Not Equal | Z = 0 |
| 0010 | CS/HS | Carry set/unsigned > = | C = 1 |
| 0011 | CC/LO | Carry clear/unsigned < | C = 0 |
| 0100 | MI | Minus/Negative | N = 1 |
| 0101 | PL | Plus/Positive or Zero | N = 0 |
| 0110 | VS | Overflow | O = 1 |
| 0111 | VC | No overflow | O = 0 |
| 1000 | HI | Unsigned higher | C = 1 & Z = 0 |
| 1001 | LS | Unsigned lower or same | C = 0 \| Z = 1 |
| 1010 | GE | Signed > = | N = = V |
| 1011 | LT | Signed < | N! = V |
| 1100 | GT | Signed > | Z = = 0, N = = V |
| 1101 | LE | Signed < = | Z = = 1 or N! = V |
| 1110 | AL | Always | |
| 1111 | (NV) | Unpredictable | |

Explanation with conditional execution is needed if asks Branch in 10marks question

# Branch Instructions…….

***Examples***

```
        B       NEW     ;transfers control unconditionally to location NEW
STOP    B       STOP    ;continually branches to its own label STOP
        BNE     NOO     ;branch to NOO if Z flag is not set
        BHI     LUX     ;branch if high, i.e., if C = 1
```

The format of a branch instruction is as shown in Figure 10.16. Target addresses are 'relative'. What this means is that when a branch instruction is taken up, the PC (program counter) value and the value specified by the instruction are algebraically added.

The target is specified as a 24-bit signed number; this number is shifted left (logically) twice (so that the two LSB bits are zero. This makes all target address to be 'aligned' (Ref Secion 10.5.2). The left shifting also multiplies the number by 4. This makes the target to have 26 bits, that is, the maximum range is between +/– 225 (one bit is for sign, remember). This number is added to the PC value. In short, what is done with the 24-bit immediate number, by the instruction is that *it shifts it left by two bits, sign extends it to 32 bits, and adds it to PC.* Thus, the maximum range for branching is only +/- 32 MB. ($2^{25} = 2^{20} \times 2^5$; $2^{20} = 1$ MB, $2^5 = 32$). For branch addresses beyond this range, the PC can be directly loaded with the target address. Now see this simple program which calculates the factorial of 10.

# Branch Instructions…….

**Example 10.7** Now see this simple program which calculates the factorial of 10.

```
        AREA FACTO, CODE        ;define the code area
        ENTRY                   ;entry point
        MOV R1, #10             ;R1 = 10
        MOV R2, #1              ;R2 = 1
REPT    MUL R2, R1, R2          ;R2 = R2 xR2
        SUBS R1, R1, #1         ;R1 = R1- 1
        BNE   REPT              ;branch to REPT if Z! = 0
STOP    B   STOP                ;last line
        END
```

This is a very simple program which finds the factorial of 10. It can be used to find the factorial of any other number (except 0), provided the factorial does not exceed 32 bits in size. The technique is to multiply the number with the 'number-1' recursively. Meanwhile, a counter also decrements by 1(which is done by subtraction), and when the counter is 0, the Z flag is set. The multiplication is then stopped. The factorial is available in the register R2. The branch instruction used is a conditional one, that is, BNE which tests the Zero flag. The instruction before it, that is, SUB has been appended with the 'S' suffix to ensure the setting of flags.

Now let's see another example which uses conditional branching. This program performs division by repeated subtraction.

# Branch Instructions…….

**Example 10.8**

Now let's see another example which uses conditional branching. This program performs division by repeated subtraction.

```
        AREA DIV, CODE
        ENTRY
        MOV R1, #500          ;Move the dividend to R1
        MOV R2, #16           ;Move the divisor to R2
        MOV R3, #0            ;R3 = 0
        MOV R4, R1            ;copy the dividend to R4
REPT    SUBS R4, R4, R2       ;subtract and set flags
        ADDPL R3, R3, #1      ;add if N = 1 i.e. MSB of R$ is +ve
        BPL REPT              ;repeat the loop if the MSB is +ve
        ADDMI R4, R4, R2      ;if MSB of R4 is –ve, add R2 to R4
STOP    B    STOP
        END
```

This program performs division by repeated subtraction. Here 500 is to be divided by 16. The method is to subtract 16 from 500 repeatedly until the result becomes negative. The branch instruction BPL REPT means Branch to label REPT if plus (PL), i.e., if N = 0.

Besides conditional branching, there are the ADD and SUB instructions also, which are conditional—the condition used is the status of the sign flag N.

The steps of the program are as follows:

i) Subtract 16 from 500, and check if the result is +ve or –ve. This can be verified by checking the N flag which corresponds to the MSB of the resultant number. The condition flags are updated by the subtraction operation (using the suffix S).

ii) If the number (in R4) is +ve, it means that subtraction can be repeated unhindered. Each time this is verified, the quotient register (R3) is incremented by 1.

iii) When the result of subtraction becomes – ve, (the condition 'MI' for minus), add the divisor to this negative number (in R3).

iv) In this problem, when 16 is subtracted 31 (0x1F) times from 500, the value in R4 is +ve. One more subtraction makes the N flag to be set, and the number R4 to be negative.

v) To this –ve number add the divisor. This makes it equal to the remainder which is 4, in this case.

vi) Thus, we get 31 (0x1F) as the quotient (in R3) and 4 as the remainder (in R4)

# Branch Instructions…….

## 10.16.1 | Subroutines/Procedures

In Table 10.13, there is another form of the branch instruction which is BL standing for 'Branch and Link'. Recollect that a procedure (also called subroutines, functions, etc.) means that a new program sequence is taken up, but control returns to the original point after that. Most processors (including ARM) use stacks to store the return addresses and return instructions to handle procedure calls. ARM has an additional feature to handle procedures in a simpler manner. Recollect a register named the 'Link Register'. When a BL instruction is encountered, the PC value is changed to that of the target, but the old PC value is copied to the LR register. At the end of the procedure, the LR value can be copied back to the PC.

Now let's write a program which calls a procedure.

## Example10.9

Write a program to calculate $3x^2 + 5Y^2$, where X = 8 and Y = 5

***Solution***

```
            AREA PROCED,CODE
            ENTRY

            MOV R2,#8                ;to calculate 3X² +5Y²
            BL SQUARE                ;call the SQUARE procedure
            ADD R1,R3,R3,LSL #1      ;3X²
            MOV R2,#5                ;R2 = 5
            BL SQUARE                ;call the SQUARE procedure
            ADD R0,R3,R3,LSL #2      ;5Y²
            ADD R4,R1,R0             ;R4 = R1+R0 i.e 3X² +5Y²
STOP        B STOP                   ;last line in the execution

SQUARE      MUL R3,R2,R2             ;the SQUARE procedure
            MOV PC,LR                ;return LR back to PC
            END
```

# Branch Instructions…….

The salient points of this program are as follows:

i) A procedure named SQUARE has been used. This procedure uses the multiply instruction to find the square of any number. The number to be squared is passed to the procedure using the register R2. The square of the number is returned to the main program in R3.

ii) There are two numbers, X and Y, whose squares are to be found. Calling the procedure amounts to just writing the instruction BL SQUARE. This instruction will cause a branching to the procedure named SQUARE. It also copies the current PC value to the link register (LR).

iii) The procedure has only two instructions: one to perform squaring, and the other to copy the LR content back to PC. The second instruction causes a return to the main program.

iv) We need two multiplications, in addition to the squaring operation. These two, that is, $3X^2$ and $5Y^2$ are achieved by shifting and adding. The MUL instruction is used as little as possible because it takes more time, and causes higher power dissipation.

v) The last step is adding $3X^2$ and $5Y^2$ which are now in R1 and R0. The sum is available in R4.

vi) Note that the last program line to be executed is STOP B STOP, even though it is not the last line in the assembly file.

In Table 10.13 there are two more forms for the branch instruction. BX stands for Branch and Exchange. BLX is for Branch Link and Exchange. The Exchange feature is applicable when ARM and THUMB instructions are being used, and it is needed to switch from one set to another.

# Branch and Branch with Link and Exchange:

## Branch and Branch with Link (B, BL)

Syntax: (or) Assembler Format: **B {L} {cond} <target address>**

- Branch (B) instruction is similar to Jump instruction (no need to return)
- Branch with Link (BL) instruction is similar to CALL instruction (need to return)

Binary encoding:


Ex: BNE loop1

- The range of the branch instruction is +/- 32 Mbytes.

# Branch and Branch with Link and Exchange…

An unconditional jump:

```
          B       LABEL   ; unconditional jump ..
          ..
LABEL     ..              ; .. to here
```

To execute a loop ten times:

```
          MOV     r0, #10 ; initialize loop counter
LOOP      ..
          SUBS    r0, #1  ; decrement counter setting CCs
          BNE     LOOP    ; if counter <> 0 repeat loop..
          ..              ; .. else drop through
```

To call a subroutine:

```
          ..
          BL      SUB     ; branch and link to subroutine SUB
          ..              ; return to here
          ..
SUB       ..              ; subroutine entry point
          MOV     PC, r14 ; return

          B       LABEL
```

Conditional subroutine call:

```
          ..
          CMP     r0, #5  ; if r0 < 5
          BLLT    SUB1    ; then call SUB1
          BLGE    SUB2    ; else call SUB2
          ..
```

# Branch and Branch with Link and Exchange…

## Branch, Branch with Link and exchange (BX, BLX)

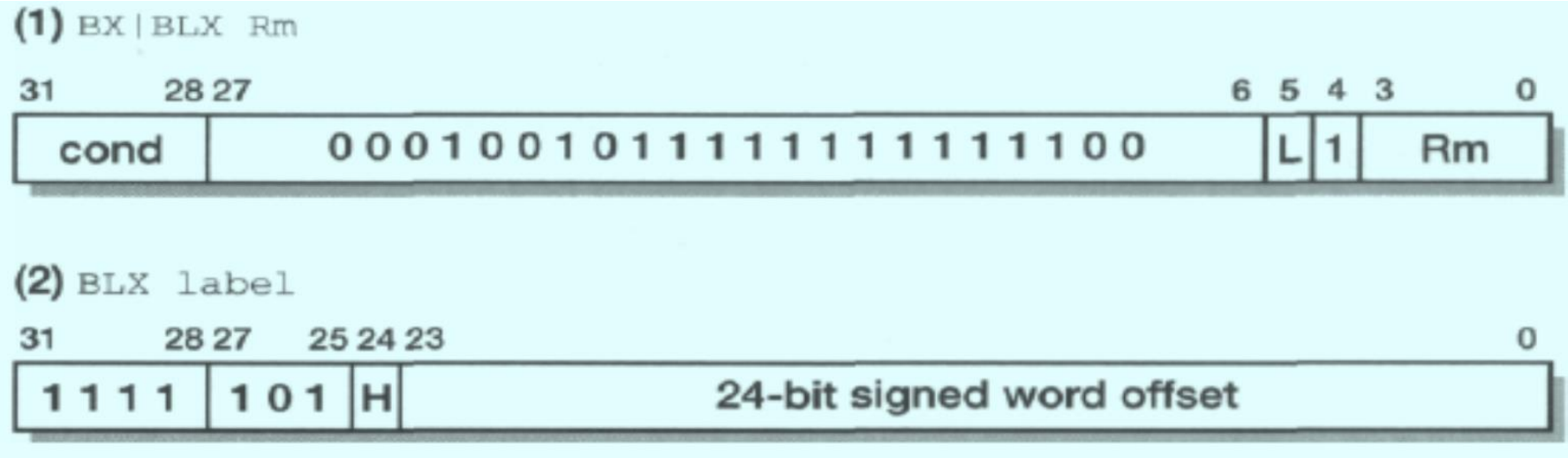
Syntax: (or) Assembler Format:

1. **B {L} X {cond} Rm**
2. **BLX <target address>**

These instructions are available on ARM chips which support the Thumb (16-bit) instruction set, and are a mechanism for switching the processor to execute Thumb instructions or for returning symmetrically to ARM and Thumb calling routines. A similar Thumb instruction causes the processor to switch back to 32-bit ARM instructions. BLX is available only on ARM processors that support architecture v5T.

# Branch and Branch with Link and Exchange…

Binary encoding:




Ex: BX R0

The branch target is specified in a register, Rm. Bit[0] of Rm is copied into the T bit in the CPSR and bits[31:1] are moved into the PC:

- If Rm[0] is 1, the processor switches to execute Thumb instructions and begins executing at the address in Rm aligned to a half-word boundary by clearing the bottom bit.
- If Rm[0] is 0, the processor continues executing ARM instructions and begins executing at the address in Rm aligned to a word boundary by clearing Rm[l:0].

# Load/Store Instructions

## 10.18 | Load and Store Instructions

ARM is a RISC architecture, and one of the features of RISC is that of being a 'load store' architecture. Loading is the process of getting data from memory into a register, and storing is just the reverse process. In ARM, data is brought into registers using a load instruction, and only then can it be used for data processing. After computation, the result can be 'stored' in memory. The memory in question is 'RAM' which is the read/write memory. RAM is volatile and is used for temporary storage of data in the course of computations. The only instructions which access RAM are 'load' and 'store'. All registers can be accessed using these instructions, but programmers are advised to exercise caution when accessing critical registers like the PC, SP, etc.

The syntax for load or store is

LDR/STR {<cond>}<Rd>, <addressing mode>

Rd is the source register for store and destination register for load.

The addressing mode gives us the necessary information to get the 'effective address', which is the actual memory address to be accessed. The addressing mode is indirect because the memory address is not to be specified directly in the instruction, rather a base register is mandatorily used. For the simplest case, an example of LOAD and STORE instructions are as follows:

```
LDR     R1, [R2]     ;copy into R1 the content of memory specified in R2
STR     R1, [R2]     ;store the content of R1 into the memory address specified in R2
```

This implies that the load/store instruction must be preceded by an instruction which copies the address into R2. We will soon get to know how this is done. There are various ways of specifying the effective address. The barrel shifter can be part of the address specifying mechanism.

# Load/Store Instructions…….

## Example 10.11

How is the effective memory address calculated in the following load and store instructions?

i) LDR R3, [R2, LSL #2]
ii) STR R9, [R1, R2, ROR #2]
iii) LDR R4, [R3, R2]
iv) STR R5, [R4, R3, ASL #4]

### Solution

i) LDR R3, [R2, LSL #2]
In this the effective address is the content of R2 left shifted by 2, i.e. multiplied by 4
ii) STR R9, [R1, R2, ROR #2]
Here, the effective address is specified by R1, R2 and a right rotation. To calculate it, the content of R2 is rotated twice by 2, and then added to the content of R1.
vi) LDR R4, [R3, R2]
The effective address here is the sum of R3 and R2.
vii) STR R5, [R4, R3, ASL #4]
The effective address is the sum of the content of R4 and the arithmetically left shifted (by 4) content of R3.

# Load/Store Instructions…….

## 10.18.1 | Bytes, Half Words and Words

Now, let's see another aspect of load and store instructions. ARM has instructions to transfer specifically a word (32 bits), half word (16 bits) or a byte (8 bits) between

**Table 10.16** | List of Load and Store Instructions

| | | | |
|---|---|---|---|
| LDR | Load Word | STR | Store Word |
| LDRH | Load Half Word | STRH | Store Half Word |
| LDRSH | Load Signed Half Word | | |
| LDRB | Load Byte | STRB | Store Byte |
| LDRSB | Load Signed Byte | | |

memory and registers. There are also instructions which differentiate between signed and unsigned data.

There are instructions which clearly indicate the kind of data to be moved. See Table 10.16. From the table, we understand that we can load and store parts of a 32-bit word by using B for byte and H for half word, along with the load and store instructions.

If a memory location contains a 32-bit word, we can move the LSB (assuming little endian format) into a register by using LDRB, or the lower half of the word by using LDRH. Let's clarify this by an example.

# Load/Store Instructions…….

## Example 10.12

Two memory areas are being referenced and two registers are used as pointers:

R1 = 0x00000100

R2 = 0x40001200

Figures 10.18a and b show the data addresses and corresponding data.

Show the content of memory, after the execution of the following instructions:

| Address | Byte Stored |
|---|---|
| 0 x00000100 | 56 |
| 0 x00000101 | 23 |
| 0 x00000102 | 0D |
| 0 x00000103 | AE |

**Figure 10.18a** | Address and data

| Address | Byte Stored |
|---|---|
| 0 x40001200 | 00 |
| 0 x40001201 | 00 |
| 0 x40001202 | 00 |
| 0 x40001203 | 00 |

**Figure 10.18b** | Address and data

i) LDR R3, [R1]

ii) LDRB R3, [R1]

iii) LDRH R3, [R1]

iv) STRB R3, [R2] given that R3 = 0xAE0D2356

For this case, show half word and word storage as well.

# Load/Store Instructions…….

```
LDR     R1, [R7]     ;R1 = 0xCDEF8204…….Case 1
LDRSH   R2, R7]      ;R2 = 0xFFFF8204……...Case 2
LDRSB   R3, [R7]     ;R3 = 0x00000004……....Case 3
```

For case 1, the 32 bits are copied to R1. For case 2, only the lower 16 bits are to be copied. The MSB of the 16-bit half word is '1', and this is extended to 32 bits while copying to R2. That's how the upper 16 bits of R2 become FFFF.

For case 3, the lowest byte alone is copied. Its MSB is 0. As such, the rest of R3 is filled with zeros, i.e., the sign bit '0' is extended to fill the upper 24 bits. You can also observe in Table 10.16 that there are no store instructions for **signed bytes or signed half words.** This is because storing simply means placing numbers in memory. These numbers may be signed, unsigned data or code—it is only when the user brings it to a register, is the processing on that number done. Only then it is necessary for that number to be interpreted as signed or unsigned.

### 10.18.3 | Indexed Addressing Modes

In this mode, the effective address calculation can be done before a load/store is executed or afterwards. Let's see what it is all about.

#### 10.18.3.1 | *Pre-indexed Addressing Mode*

Observe the instruction LDR R0, [R7, #4]. Here R7 is the base register and the effective address is R7 + 4. The data at this effective address is copied to R7.

Next, see the instruction STR R1, [R5, R6, LSL #2]. The effective address = R5 + R6 left shifted twice.

In the above two instructions, there is a notable feature, however. After the load/store is done, the base address content remains unchanged, that is, the effective address is not copied to the base register. But if we want the base address to contain the effective address, just suffix the instruction by the character '!' and then 'write back' occurs.

Consider the instruction LDR R2, [R6, #-8] !. In this, after the loading operation is done, R6 has the effective address written back into it.

# Load/Store Instructions…….

## Example 10.13

Calculate the effective addresses and explain what each instruction does.

i) STRB R2, [R6, R7, #0x24]!
ii) LDRSH R4, [R10, R11, ASR #4]

***Solution***

i) STRB R2, [R6, R7, #0x24]!
The effective address is the sum of the contents of R6, R7 and the number 0x24. The content of R2 is stored in the effective address. After that, the effective address is copied to R6.

ii) LDRSH R4, [R10, R11, ASR #4]
Here, the effective address is the sum of the contents of R10 and R11 after arithmetically shifting it right by 4 positions. The half word in this address is loaded to R4. The contents of the base register remains unchanged.

# Load/Store Instructions…….

### 10.18.3.2 | *Post-indexed Addressing Mode*

In this mode, the effective address calculation is done after the execution of the specific instruction has been done.

Take the case of the instruction LDR R0, [R4], #4

Here the data pointed by the content of R4 is first copied to R0. After that, the content of R4 is changed to R4 + 4. There is no need of the '!' operation because that is exactly what post-indexing does.

## Example 10.14

Let's add 10 numbers which are in memory. The numbers are 16-bit long, that is, half words, and use two byte spaces. The pre-indexed mode of addressing with write back is used to index the half words which have addresses with a spacing of 2 between them. The instruction LDRH R2, [R7, #2]! does the indexing of the 16-bit numbers.

***Solution***

```
        AREA DADD, CODE, READONLY
        ENTRY

        LDR R7, = TABLE     ;copy the address of Table to R7
STRT    MOV R0,#9           ;R0 = 9
        LDRH R1,[R7]        ;load 1st number from memory to R1
REPT    LDRH R2, [R7,#2]!   ;pre-indexed with writeback
        ADD R1,R1,R2        ;R1 = R1+R2
        SUBS R0,R0,#1       ;R0 = R0-1
        BNE REPT            ;repeat the addition until R0 = 0
STOP    B STOP              ;last line
TABLE   DCW 3456,7859,1234,9876,3452,3214,7864,0987,2032
        END
```

# Load/Store Instructions…….

Let's examine the salient features of this program.

i) The numbers to be added are stored in code memory, just after the last line of the program.
ii) There are 10 numbers to be added. The first number is loaded into register R1 and the rest are loaded one by one into R2.
iii) R0 is used as a counter to the numbers. In a general case, if there are N numbers to be added, R0 = N-1. Here N = 10.
iv) The loading of the numbers to R2 is done using a loop. Since the numbers are half words, their addresses are to be incremented by2. This is done very efficiently by the pre-indexed addressing with write back scheme. After one half word is accessed, the effective address is written back to the base register R7 in readiness for accessing the next half word.
v) The address corresponding to TABLE is a 32-bit constant. It is calculated by the assembler, and loaded into R7 using the techniques mentioned in Section 10.17.

## Single Word & Unsigned Byte Data Transfer Instructions (LDR, STR)

These instructions are the most flexible way to transfer single bytes or words of data between ARM's registers and memory. Transferring large blocks of data is usually better done using the multiple register transfer instructions, and recent ARM processors also support instructions for transferring half-words and signed bytes.

**The pre-indexed form of the instruction:**

LDR | STR {Cond} {B} Rd, [Rn, <offset>] {!}

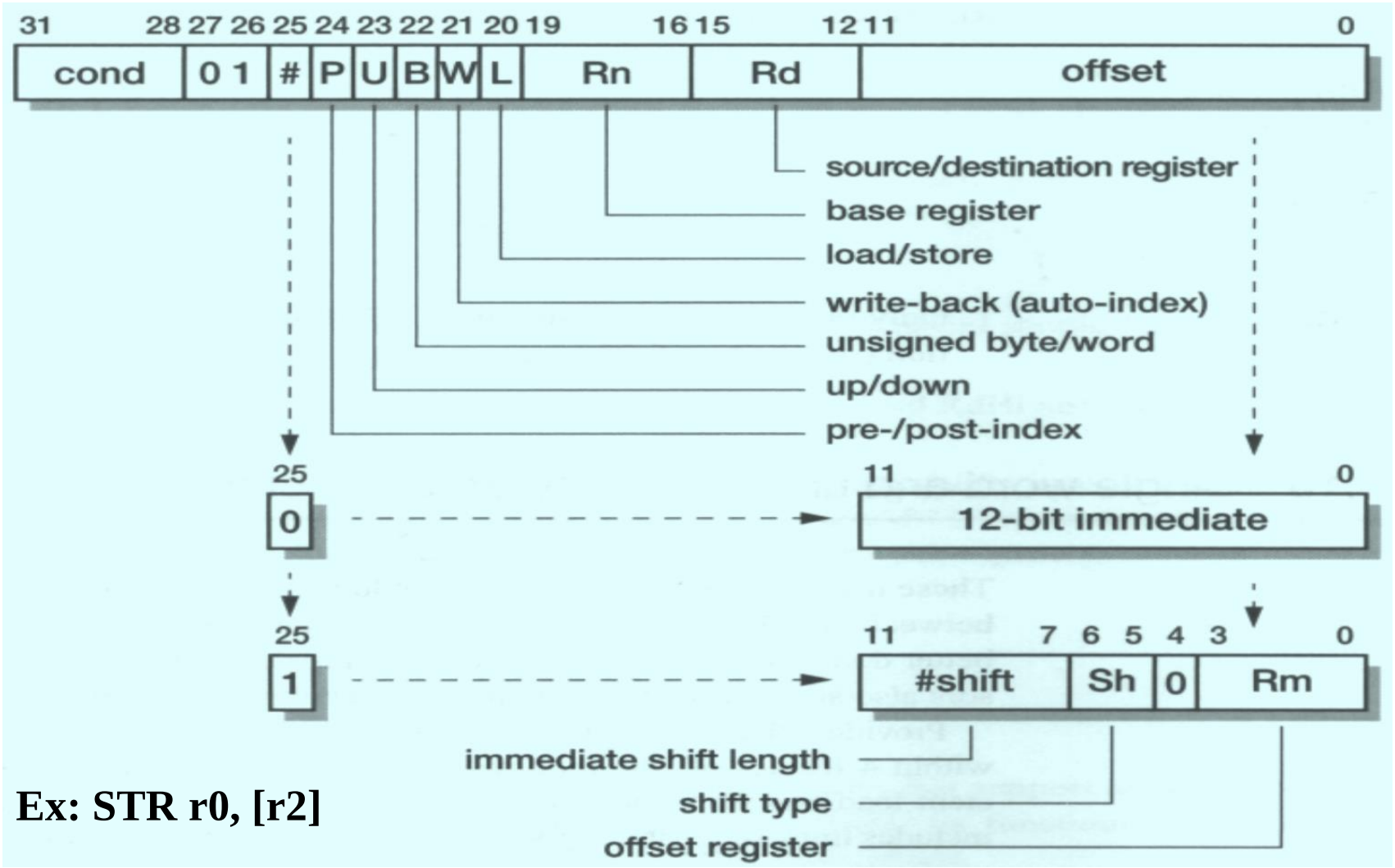
**The post-indexed form of the instruction:**

LDR | STR {Cond} {B} {T} Rd, [Rn], <offset>

LDR is 'load register', STR is 'store register';
the optional 'B' selects an unsigned byte transfer, the default is word; <offset> may be +/- <l2-bit immediate> or +/- Rm and ! selects write-back (auto-indexing) in the pre-indexed form.

# Single Word and Unsigned Byte Data Transfer and Half Word and Signed Byte Data Transfer Instructions…

**Binary Encoding:**




**Ex: STR r0, [r2]**

# Single Word and Unsigned Byte Data Transfer and Half Word and Signed Byte Data Transfer Instructions…

## Half-Word and Signed Byte Data Transfer Instructions

These instructions are used to transfer signed bytes or half words of data between ARM's registers and memory. The addressing modes available with these instructions are a subset of those available with the unsigned byte and word forms.

**The pre-indexed form of the instruction:**

LDR | STR {Cond} H | SH | SB Rd, [Rn, <offset>] { ! }

**The post-indexed form of the instruction:**

LDR | STR {Cond} H | SH | SB Rd, [Rn], <offset>

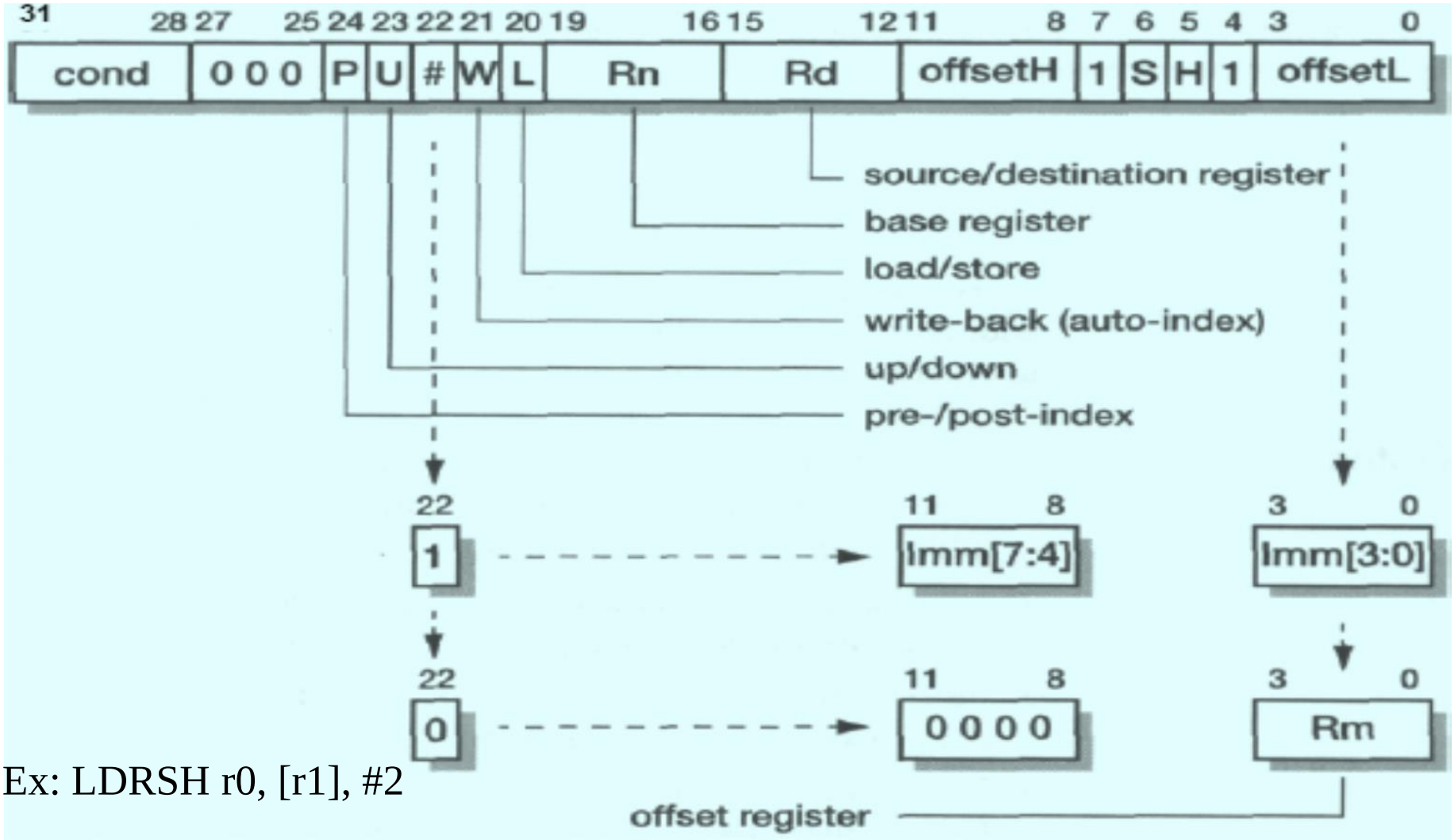
| S | H | Data type |
|---|---|---|
| 1 | 0 | Signed byte |
| 0 | 1 | Unsigned half-word |
| 1 | 1 | Signed half-word |

These instructions are very similar to the word and unsigned byte transfer, but here the immediate offset is limited to eight bits and The S and H bits define the type of the operand to be transferred. Since there is no difference between storing signed and unsigned data, the only relevant forms of this instruction format are:

- Load signed byte, signed half-word or unsigned half-word.
- Store half-word.

# Single Word and Unsigned Byte Data Transfer and Half Word and Signed Byte Data Transfer Instructions…

**Binary Encoding:**




Ex: LDRSH r0, [r1], #2

# Read only and Read/Write Memory

The two memory areas defined by the compiler are

- '**Readonly**' for code, and '**Read/write**' for data.

- Usually this corresponds to ROM and RAM in a physical system.
- RAM is used for intermediate results, for temporary storage, etc., as this is volatile memory.
- We can store data permanently in the readonly memory, process it and copy it in RAM.
- In the readonly memory, data is written using directives like DCD, DCW, etc. From there, it is copied to readwrite memory using load and store instructions

# Read only and Read/Write Memory…….

In Example 10.15, three memory areas have been defined: one in readonly, and two in readwrite memory. What is accomplished is just the transfer of a word from readonly memory to readwrite memory. In readwrite memory, one part is a space of 60 bytes. The next is a word space which is initialized to 0. After the execution of the program, the number **653451134** is copied to both these spaces.

**Example 10.15**

```
        AREA FIRST, CODE, READONLY
        ENTRY
        LDR R7, = NUMS     ;load the address of NUMS in R7
        LDR R8, = NUMS1    ;load the address of NUMS1 in R8
        LDR R9, = NUMS2    ;load the address of NUMS2 in R9
        LDR R1,[R7]        ;load the word to R1
        STR R1,[R9]        ;store the word in R1 in NUMS2
        STR R1,[R8]        ;store the word in R1 in NUMS1
STOP    B STOP
NUMS    DCD 653451134
        AREA SECOND, DATA, READWRITE
NUMS2   SPACE 60
NUMS1   DCD 0
        END
```

The number is specified in decimal **653451134**
Its equal Hexa value is **26F2DF7E**

# Read only and Read/Write Memory…….

## Example 10.16

```
        AREA STRIN1, CODE, READONLY
        ENTRY

STRT    LDR R1, = SOURCE   ;pointer to source string
        LDR R0, = DESTIN   ;pointer to destination string
        BL COPY            ;call procedure for copying
STOP    B STOP             ;last line of execution

COPY    LDRB R2, [R1],#1   ;Load byte and update address.
        STRB R2, [R0],#1   ;Store byte and update address.
        CMP R2, #0         ;Check for 0
        BNE COPY           ;repeat until the string is over
        MOV PC,LR          ;return to calling program

SOURCE DCB "I am sam",0

        AREA STRIN2,DATA,READWRITE
DESTIN  DCB 0
        END
```

# Read only and Read/Write Memory…….

Example 10.16 uses many of the programming aspects that we have been discussing so far. Let's have a look at the important features of this program.

- There is an ASCII string written in readonly memory using the DCB directive. Such a string is enclosed in double quotes and each character is a byte.
- One readonly and one read/write memory areas have been defined.
- After the ASCII string, a 0 is used as a terminating character. The arrival of this 0 in R2 is used to check whether the required transfer of the string is done.
- The instructions for loading and storing are suffixed by ‘B’ which indicates that only a byte is to be transferred.
- Post-indexed mode of addressing is used for load and store. The addresses need to be incremented only by 1, as only a byte is transferred.
- The instructions for loading and storing are in a procedure named COPY. The procedure is called by the BL instruction which does branching and also copies the current PC to the link register. The last line of the procedure is copying the LR back to PC. This constitutes the ‘return’ to the main program.

# Multiple Register Load and Store

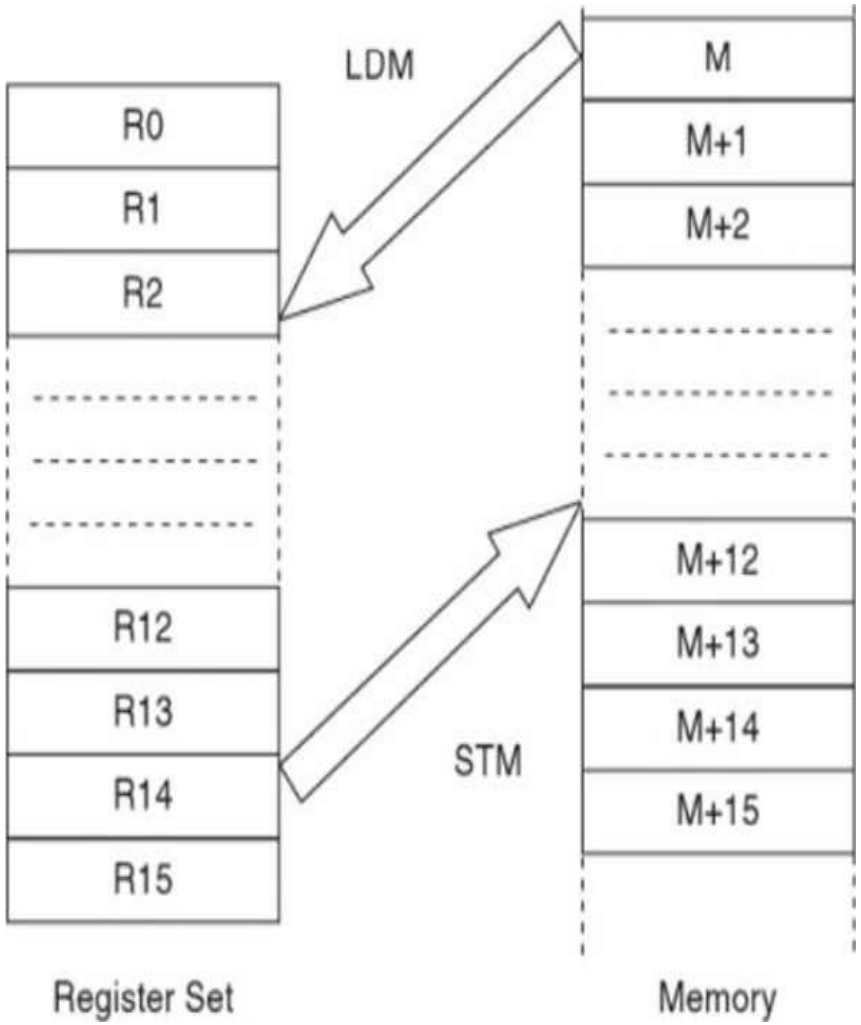
An advanced (or let's say an extended) form of loading and storing, wherein multiple registers are involved. But only data in the form of words (32 bits) can be handled by these instruction. The mnemonic of multiple load and store is LDM/STM.

## The LDM Instruction

**LDM*{cond}* *address-mode* Rn{!},reg-list {^}**

- Rn is the base register for the load operation. The address stored in this register is the starting address for the load operation.
- There can be a number of modes for specifying the address,
- ***register-list*** is a comma-delimited list of symbolic register names and register ranges enclosed in braces. There must be at least one register in the list. Register ranges are specified with a dash. For example, {R0-R5, R9} is a list.
- The ! option is for 'write back' and the ^ option is relevant for interrupts. Write back is not to be specified if the base register Rw is in ***register-list.***

# Multiple Register Load and Store…….




Multiple register load means that multiple memory locations are to be accessed, and loaded into multiple registers. There is a 'base register' acting as a pointer for the first memory location to be accessed. This register is then incremented or decremented to point to the next memory addresses.

# Multiple Register Load and Store…….

There are four options for handling this. The base register can be incremented or decremented by 4 (one word needs four addresses) for each register in the operation, and the increment or decrement can occur before or after the operation. The suffixes for these options are as follows:

**IA - increment after**
**IB - increment before**
**DA - decrement after**
**DB - decrement before**

Consider the instruction LDM DA R0, {R4-R9}

The base register here is R0. Let us assume it holds the number 0x45000000. The operation of this instruction is that the 32-bit word at that address is pointed by R0, and that word copied to R4. Then the address is decremented to point to the next word. So the new address is [R0-4], and this word is copied to R5. The sequence of decrementing the address and loading data from memory is done for the registers R4, R5, R6, R7, R8 and R9.

This single instruction replaces six LDR instructions. Is there any advantage in this? As far as execution is concerned, it is 'No'. All the six load operations have to be done. But note that only one 'instruction fetch*cycle is needed for the six load operations together. So there is definitely some savings in terms of time.

# Multiple Register Load and Store…….

There are four options for handling this. The base register can be incremented or decremented by 4 (one word needs four addresses) for each register in the operation, and the increment or decrement can occur before or after the operation. The suffixes for these options are as follows:

**IA - increment after ,** **IB - increment before ,**
**DA - decrement after ,** **DB - decrement before**

**Consider the instruction LDMDA R0, {R4-R9}**

The base register here is R0. Let us assume it holds the number 0x45000000. The operation of this instruction is that the 32-bit word at that address is pointed by R0, and that word copied to R4. Then the address is decremented to point to the next word. So the new address is [R0-4], and this word is copied to R5. The sequence of decrementing the address and loading data from memory is done for the registers R4, R5, R6, R7, R8 and R9.

This single instruction replaces six LDR instructions. Is there any advantage in this? As far as execution is concerned, it is 'No'. All the six load operations have to be done. But note that only one 'instruction fetch*cycle is needed for the six load operations together. So there is definitely some savings in terms of time.

**What Is the difference in operation of the following instruction?**
**LDMIA R10,{ R9, R1 - R5}**

Here the base address is in R10, and after each data transfer, it is incremented by 4. In the destination register list, R9 is specified first, but the processor has a particular way of handling the list. The lowest register will always be loaded from the lowest address in memory, and the highest register from the highest address. Here R1 gets the data in the address pointed by R10.

# Load/Store Instructions…….

## 10.20.2 | The STM instruction

This has the same format as the LDM instruction. Consider the instruction

```
STMIA R1, {R2-R4}
```

This will be equivalent to the instructions

```
STR R2, [R1]
STR R3, [R1. #4]
STR R4, [R1. #8]]
```

After the sequences of four stores are over, the base content does not vary, however. If you need it to be changed to that of the final address, the writeback operator '!' is to be used. So write the instruction as STMIA R1!,{R2-R4}

Now let's use the LDM and STM instructions to simply Example 10.16 which transfers bytes from one portion of memory (Readonly) to another portion (Read/write). But the multiple load/store instructions can be used only for words (32 bits). So Example 10.17 has been modified and used to move 6 words.

# Load/Store Instructions…….

**Example 10.17**

```
        AREA STRIN1, CODE, READONLY
        ENTRY

        LDR R1, = SOURCE        ;pointer to source
        LDR R0, = DESTIN        ;pointer to destination
        LDMIA R1,{R2-R8}        ;Load six words to R2-R8
        STMIA R0,{R2-R8}        ;Store six words in destination
STOP    B STOP

SOURCE DCD 0x675889,0x1234568,0x9876543,0x2345678,0x8907653

        AREA STRIN2,DATA,READWRITE  ;define the R/W memory area
DESTIN DCD 0                        ;
        END
```

Here 6 words from the source memory have been copied to six registers using just one instruction. In the next instruction, these six words are stored in the destination memory

Note how simple, the program is.

Example 10.17 illustrates the idea that block data transfers can be simplified using the multiple register instructions. But their real importance is for stack implementation. Stacks are a necessity for any processor; stacks are needed for storing data temporarily and also for storing return addresses and register values during procedure calls. We will see this now. For those who are not very familiar with the concept of stacks, here is a brief review.

# Load/Store Instructions…….

## 10.20.3 | Stack

A stack is an area in memory, the accessing of which is done in a special way. Most stacks are Last-In First-Out (LIFO) type stacks. This means that the last data that was stored is the first one that can be taken out. It is sequential access that is done, and not random access. Two operations are defined for a stack, that is, the PUSH, in which data is written into the stack, and POP in which data is read out and loaded into registers. The stack has a pointer to its top which is called the Stack pointer (SP). For ARM, this is register R13. This means that the address of the top of the stack is to be available in SP.

### 10.20.3.1 | *Types of Stacks*

#### *Ascending/Descending and Empty/Full*

An ascending stack grows upwards. It starts from a low memory address and, as items are pushed onto it, progresses to higher memory addresses. A descending stack grows downwards. It starts from a high memory address, and as items are pushed onto it, it progresses to lower memory addresses.

In an **empty stack**, the stack pointer points to the next free (empty) location on the stack, i.e., to the place where the next item to be pushed, will be stored. In a **full stack**, the stack pointer points to the topmost item in the stack, that is the location of the last item pushed onto the stack. In practice, stacks are almost always **full and descending**. Most stacks are 'Full descending' types.

Let's consider a descending stack in which SP is first decremented and then data is pushed in. The reverse occurs for the POP operation. Stacks allow data to be pushed or popped only as words (32 bits for ARM). Consider that SP = 0x50002000, and the contents of R1 and R2 are pushed in At the end of the operation we find that SP = SP-8 = 0x50001FF8. ARM does not have a mnemonic for PUSH, instead it uses the STM instruction. To simplify the use of the STM/LDM instructions corresponding to PUSH and POP for different types of stacks, Table 10.18 can be referred to.

# Load/Store Instructions…….

***For the kind of stack that we are talking about now, what is the instruction we can use for pushing the contents of registers R1 to R3?***

The answer is STMDB SP! {R1-R3}. We need SP to be used as the base register. For pushing in, SP is first decremented, and then storing is done. So we use the suffix 'DB' along with SP. The operator '!' is used such the decremented value is available in SP.

See this simple program (Example 10.18) in which SP is initialized to 0x40000200. Some values are loaded into registers R1 to R3. Using the STMDB instruction, the content of the three registers, that is, 3 words are pushed to the stack, and will be available in memory. At the end of the program, SP will be found to have the value of 0x400001F4.

**Table 10.18** | Types of Stacks and Corresponding Instructions to be Used

| Stack Type | Push | Pop |
|---|---|---|
| Full Descending | STMFD (DB) | LDMFD (IA) |
| Full Ascending | STMFA (IB) | LDMFA (DA) |
| Empty Descending | STMED (DA) | LDMED (IB) |
| Empty Ascending | STMEA (IA) | LDMEA (DB) |

# Load/Store Instructions…….

## Example 10.18

```
        AREA STCK, CODE, READONLY
        ENTRY
        LDR SP, = 0x40000200
        MOV R1,#1
        MOV R2,#2
        MOV R3,#3
        STMDB SP!,{R1-R3}
STOP    B STOP
        END
```

Now, in this program, if the STM instruction is changed to STMIA, the stack becomes an ascending stack, and the value of SP will be 0x4000200C, after program execution. Thus, it is obvious that a stack is a data structure which can be defined by software.

# Load/Store Instructions…….

## 10.20.4 | Stacks and Subroutines/Procedures

For most processors, procedures use a stack to store the return address. A procedure is taken up by a 'CALL' instruction. This causes the action of pushing the current value of PC onto the stack. The procedure ends with a 'RETURN' instruction. This causes the PC value to be popped back.

For ARM, so far (Section 10.16.1) we have used procedures without the necessity of a stack. That is because the Link Register (LR) keeps the return address, when a procedure is called. But think of the case of nested procedures. There is only one link register for a mode, and a new procedure will overwrite the existing link register which stores the details of the previous procedure, and very soon things may go out of hand.

In such cases, a stack is a necessity. Each time a procedure is called, the PC value is saved in the LR, as is the usual case. When a nested procedure comes in, the content of the link register is pushed on to the stack, and popped out from the stack when exiting the procedure. Figure 10.20 shows the sequence of actions needed to take care of a nested procedure.

Now, let's try to understand the sequence of actions indicated by Figure 10.20.

In the main program, we define a stack by giving a value to the stack pointer (SP).

# Load/Store Instructions…….

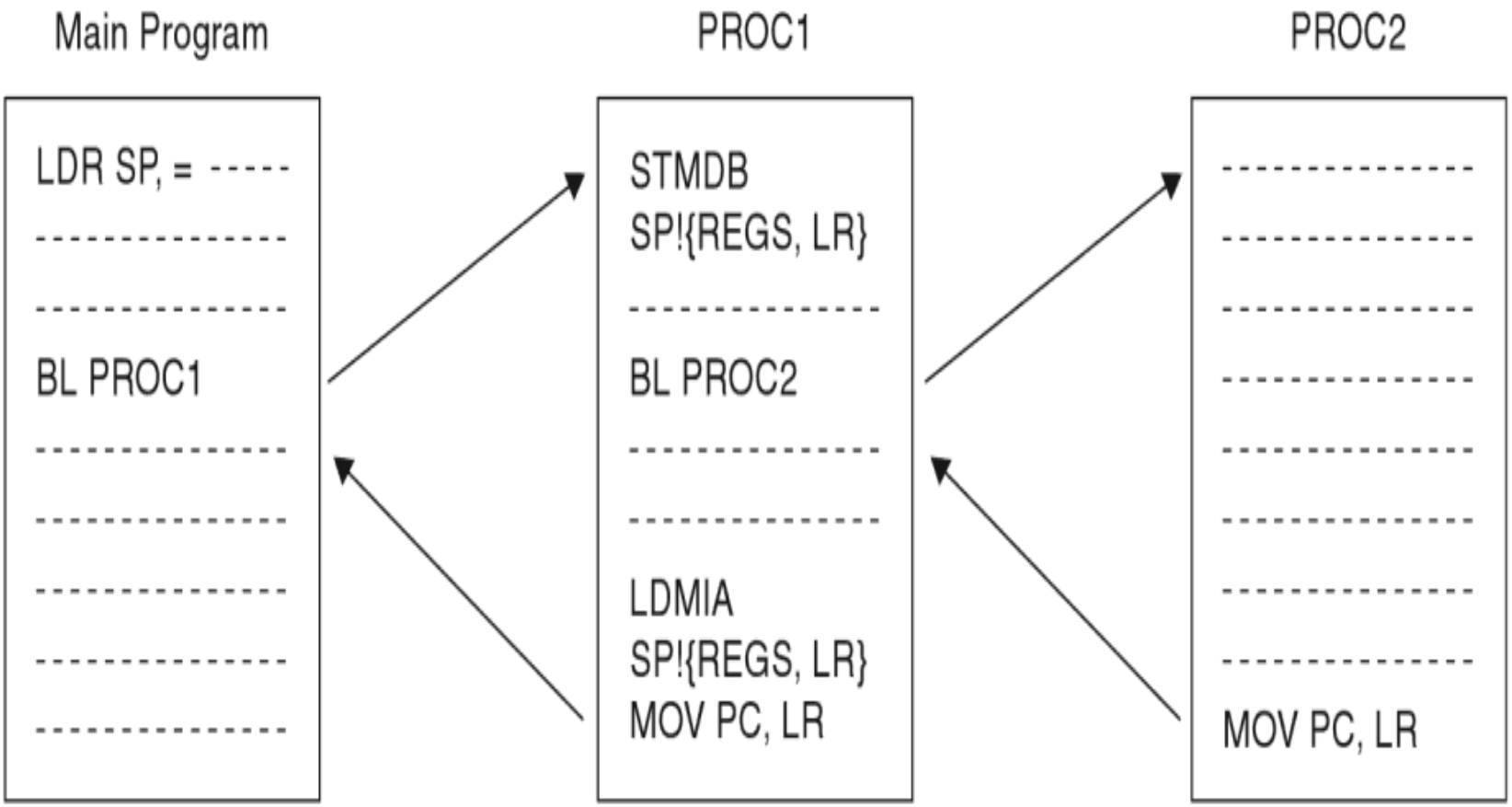



**Figure 10.20** | Sequence of actions needed for a nested procedure

# Load/Store Instructions…….

The main program has a procedure named PROC1 which is called by the instruction BL PROC1. This instruction causes the current PC value to be copied to LR. In PROC1, since we anticipate a nested procedure, we push LR and the working registers to stack using the instruction STMDB SP!,{REGS,LR}. Thus the content of LR is safely stored in the stack.

In the procedure PROC1, another procedure is called by the instruction BL PROC2. This instruction causes the copying of the present PC to LR. In PROC2, there is the instruction MOV PC, LR at the end. This will get the PC value back from LR, and thus execution goes back to PROC1.

At the end of PROC1, there is the stack based instruction LDMIA SP!, {REGS, LR}. This retrieves the contents of LR. This is given back to PC by the instruction MOV PC, LR. Thus, execution goes back to the main program.

Any part of memory can be defined as a stack, by simply defining the content of the stack pointer register. Let's write a procedure using a stack.

# Load/Store Instructions…….

```
        AREA NESTED, CODE, READONLY
        ENTRY
        LDR R7, = 0X40000000
        LDR SP, = 0x40000210    ;define SP
        MOV R1,#1
        MOV R2,#2
        MOV R3,#3
        BL PROC1                ;Call PROC1
        LDR R6,[R7]             ;load R6
STOP    B STOP

PROC1   STMDB SP!,{LR,R1-R3}    ;save registers and LR on stack
        MOV R1,#0x34
        MOV R2,#0x45
        MOV R3,#0xDC
        BL PROC2                ;call PROC2
        STR R5,[R7]             ;store R5
        LDMIA SP!,{R1-R3,LR}    ;retrieve registers from stack
        MOV PC,LR               ;copy LR to PC

PROC2   ADD R4,R2,R1            ;the nested procedure PROC2
        ADD R5,R4,R3
        MOV PC,LR               ;go back to PROC1
        END
```

# Load/Store Instructions…….

Example 10.19 shows the instance of a nested procedure and the use of the stack. The example is in tune with the sequence outlined by Figure 10.20. Nothing very important is achieved by the program, But it shows how any nested procedure can be written. PROC1 changes the contents of registers R1, R2 and R3, but since they have already been saved on the stack by the STMDB instruction, their contents can be retrieved while returning to the main program.

PROC2 adds the new contents of R1, R2 and R3, and returns. In PROC1, the sum in R5 is stored in the memory location pointed by R7. Later, in the main program, this content is loaded to R6.

# Load/Store Instructions…….

```
        AREA NUM,DATA,READONLY
        ARRAY DCD 2,7,4,5,11,17,3,15,8,6,9,19,10,23,20

        AREA COD,CODE
ENTRY
        LDR R0, = ARRAY     ;load ARRAY to R0
        LDMIA R0,{R1-R10}   ;load 10 numbers to R1 to R10
        MOV SP,#0x40000000  ;location of R/W memory
        STMIA SP,{R1-R10}   ;store the 10 numbers
        ADD SP,#40
        ADD R0,#40
        LDMIA R0,{R1-R5}    ;load next set of 5 numbers
        STMIA SP,{R1-R5}    ;store them
        MOV SP,#0x40000000  ;address of Read-write memory
        MOV R1,#0           ;Initialize counter R1 to zero
        MOV R3,SP
        LOOP1 MOV R2,#0     ;outer loop, counter from one end
        MOV R4,SP
        LOOP2 CMP R2,#14
        BEQ OUTER           ;branch to OUTER
        ADD R2,#1           ;increment the counter
        LDR R0,[R4]         ;stored as 4 bytes
                            ;hence a jump of 4
        LDR R5,[R4,#4]
        ADD R4,#4
        CMP R0,R5           ;comparing nearby values
        BLT LOOP2
        MOV R6,R0
        MOV R0,R5           ;swapping and storing them
        MOV R5,R6
        STR R5,[R4]
        SUB R4,#4
        STR R0,[R4]
        ADD R4,#4
        B LOOP2
  OUTER
        ADD R1,#1
        CMP R1,#15
        BNE LOOP1
```

```
STOP    B STOP
        END
```

Dr. V.Vijayaraghavan,
Assistant Professor-ECE,
VFSTR, Guntur

# Multiple Register Transfer Instructions:

The ARM multiple register transfer instructions allow any subset (or all) of the 16 registers visible in the current operating mode to be loaded from or stored to memory.

**The normal form of the instruction is:**

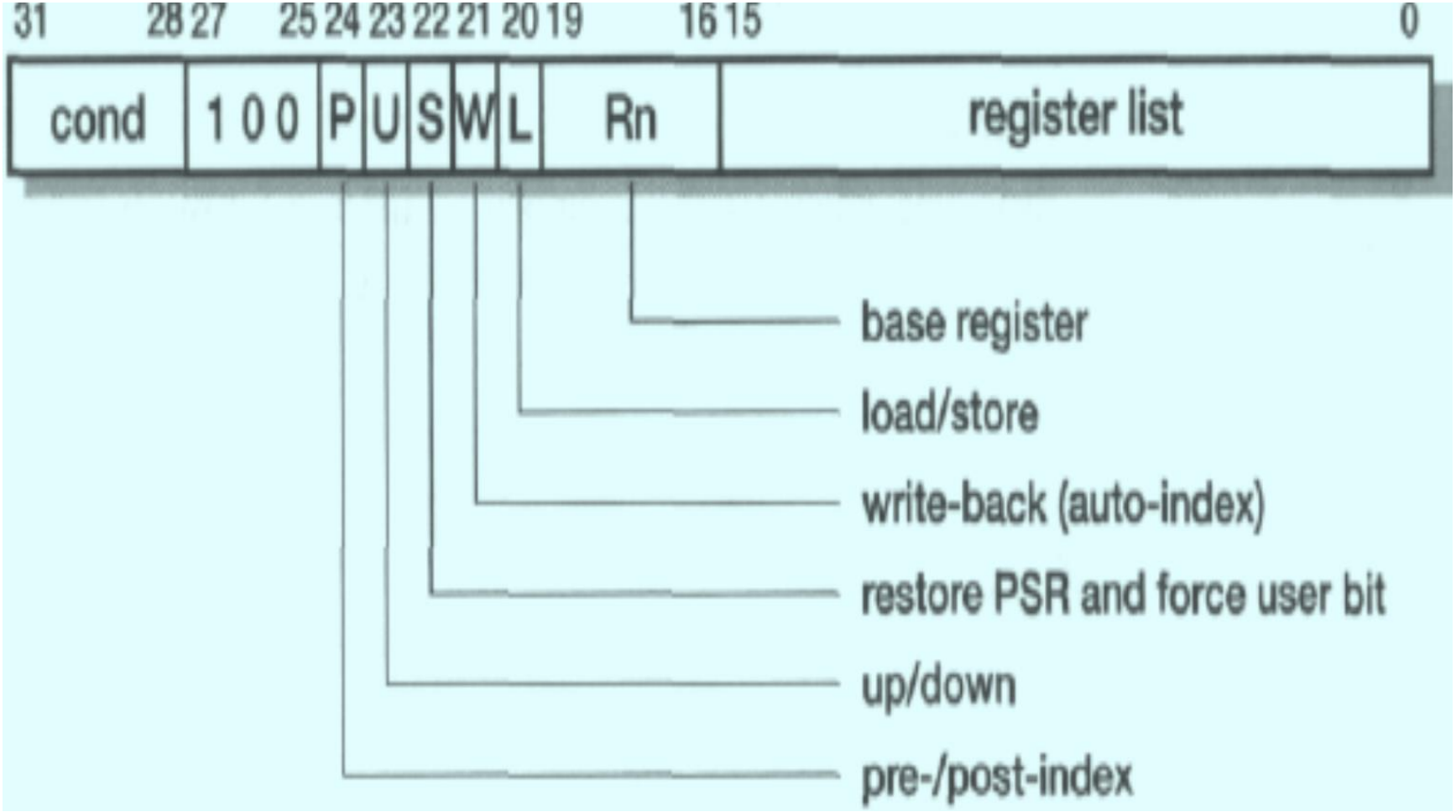
LDM | STM {Cond} <add mode> Rn {!}, <registers list>

The register list in the bottom 16 bits of the instruction includes a bit for each visible register, with bit 0 controls r0, bit 1 controls r1, and so on up to bit 15 which controls the transfer of the PC.

The base address will be incremented (U = 1) or decremented (U = 0) before (P = 1) or after (P = 0) each word transfer.

# Multiple Register Transfer Instructions…

**Binary Encoding:**




Ex: STMFD r13 !, {r0-r2, r14}

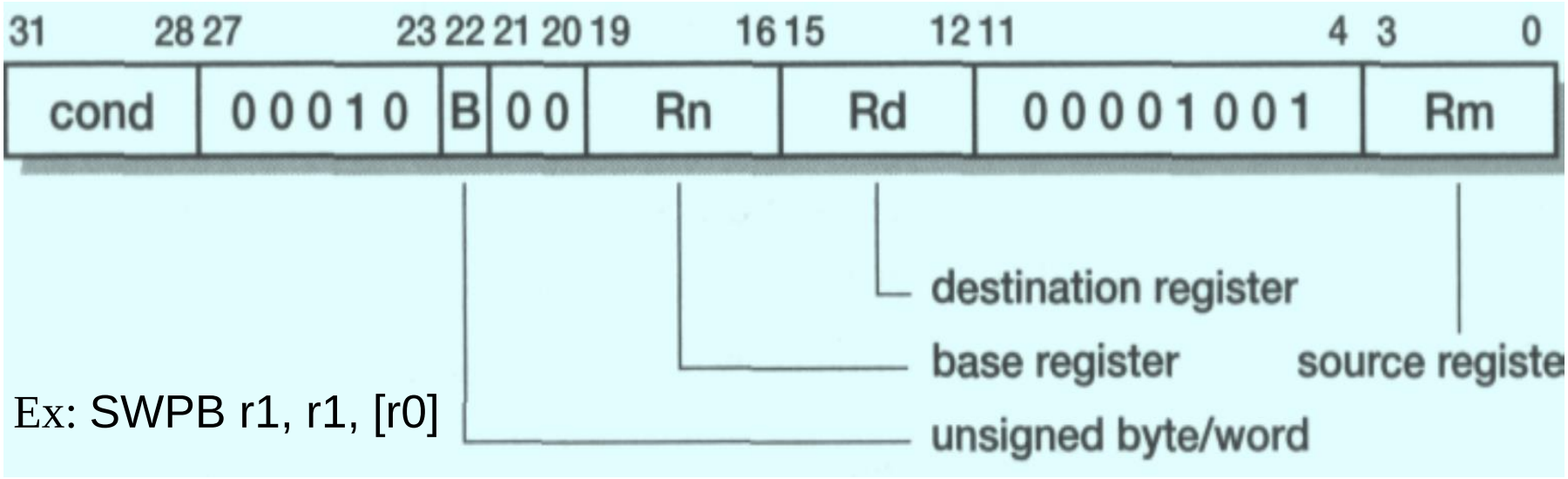
# Swap Instructions:

## Swap Memory and Register Instructions (SWP)

Swap instructions combine a load and a store of a word or an unsigned byte in a single instruction. These instructions are little used outside their role in the construction of semaphores.

**Syntax:** **SWP {Cond} {B} Rd, Rm, [Rn]**

**Binary Encoding:**




Ex: SWPB r1, r1, [r0]

# Software Interrupt:

## Software Interrupt (SWI)

The software interrupt instruction is used for calls to the operating system and is often called a 'supervisor call'. It puts the processor into supervisor mode and begins executing instructions from address 0x08.

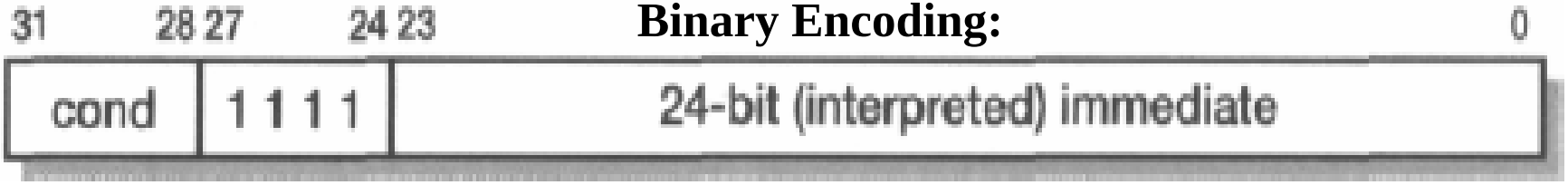
**Syntax: SWI {cond} <24 bit immediate>**

The 24-bit immediate field does not influence the operation of the instruction but may be interpreted by the system code.

1. Save the address of the instruction after the SWI (PC) into r14_svc.
2. Save the CPSR into SPSR_svc.
3. Enter supervisor mode & disable IRQ by setting CPSR[4:0] to 100112 and CPSR[7] to l.
4. Set the PC to $08_{16}$ and begin executing the instructions there.

To return to the instruction after the SWI the system routine must not only copy r14_svc back into the PC, but it must also restore the CPSR from SPSR_svc.

**Binary Encoding:**



# The Thumb Instruction Set:

The Thumb instruction set addresses the issue of code density. It may be viewed as a compressed form of a subset of the ARM instruction set. Thumb instructions map onto ARM instructions, and the Thumb programmer's model maps onto the ARM programmer's model. Implementations of Thumb use dynamic decompression in an ARM instruction pipeline and then instructions execute as standard ARM instructions within the processor.

Thumb is not a complete architecture. Thumb is fully supported by ARM development tools, and an application can mix ARM and Thumb subroutines flexibly to optimize performance or code density on a routine-by-routine basis.

## The Thumb bit in the CPSR

ARM processors which support the Thumb instruction set can also execute the standard 32-bit ARM instruction set, and the interpretation of the instruction stream at any particular time is determined by bit 5 of the CPSR, the T bit. If T is set the processor interprets the instruction stream as 16-bit Thumb instructions, otherwise it interprets it as standard ARM instructions.

# The Thumb Instruction Set:

All Thumb instructions are 16-bits long
–ARM instructions are 32-bits long

Most Thumb instructions are executed unconditionally
–All ARM instructions are executed conditionally

Many Thumb data processing instructions use a **2-address** format (the destination register is the same as one of the source registers)
–ARM use 3-address format

Thumb instruction are less regular than ARM instruction formats, as a result of the dense encoding

**Thumb properties**
–Thumb requires **70%** space of the ARM code
–Thumb uses **40%** more instructions than the ARM code
–With 32-bit memory, the ARM code is **40%** faster than the Thumb code
–With 16-bit memory, the Thumb code is **45%** faster than the ARM code
–Thumb uses **30%** less external memory power than ARM code

# The Thumb Instruction Set…

## Thumb entry

ARM cores start up, after reset, executing ARM instructions. The normal way they switch to execute Thumb instructions is by executing a Branch and Exchange instruction (BX).

## Thumb exit

An explicit switch back to an ARM instruction stream can be caused by executing a Thumb BX instruction. ARM instruction stream takes place whenever an exception is taken.

## The Thumb programmer's model

The instruction set gives full access to the eight 'Low' general purpose registers r0 to r7, and makes extensive use of r13 to r15 for special purposes:

- r13 is used as a stack pointer (SP).
- r14 is used as the link register (LR).
- r15 is the program counter (PC).

# The Thumb Instruction Set…

## Thumb-ARM similarities

All Thumb instructions are 16 bits long.

- The load-store architecture with data processing, data transfer and control flow inst.
- Support for 8-bit byte, 16-bit half-word and 32-bit word data types.
- A 32-bit unsegmented memory.

## Thumb-ARM Differences

- Most Thumb instructions are executed unconditionally.
- Many Thumb data processing instructions use a 2-address format.
- Thumb instruction formats are less regular than ARM instruction formats.

## Thumb branch instructions

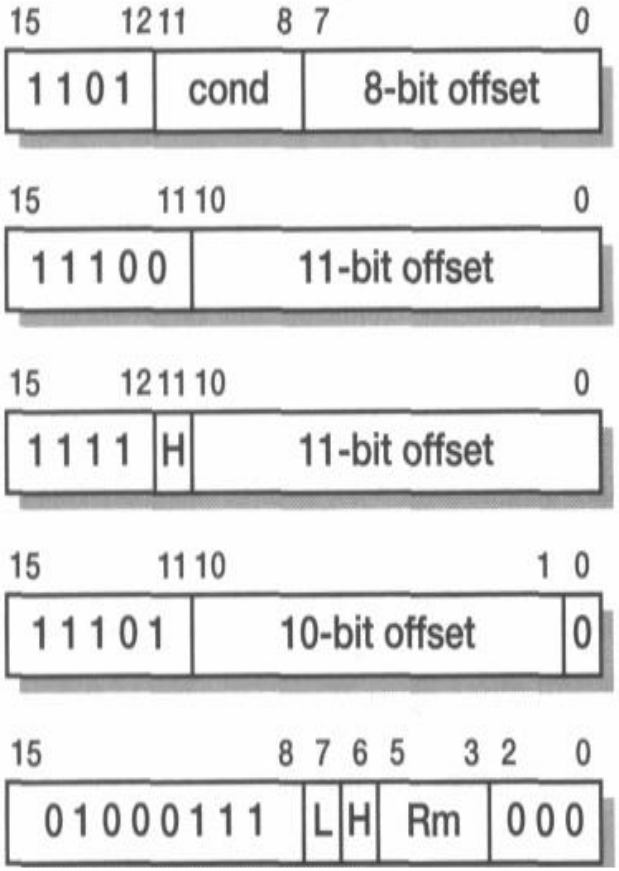
The ARM instructions have a large (24-bit) offset field which clearly will not fit in a 16-bit instruction format. Therefore the Thumb instruction set includes various ways of subsetting the functionality.

# The Thumb Instruction Set…

**Thumb branch instruction binary encodings:**

Typical uses of branch instructions include:

1. short conditional branches to control (for example) loop exit;
2. medium-range unconditional branches to 'goto' sections of code;
3. long-range subroutine calls.




B<cond> <label>; format1 - Thumb target

B <label> ; format2 - Thumb target

BL <label> ; format3 - Thumb target

BLX <label> ; format3a - ARM target

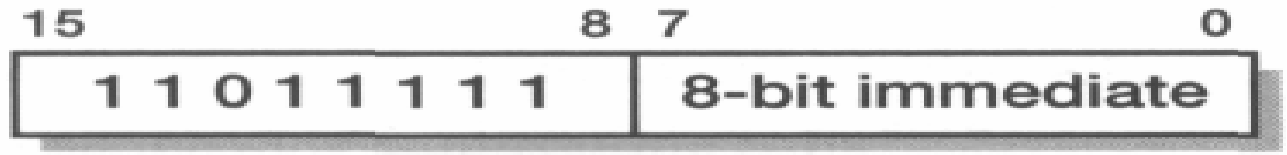
B{L}X Rm ; format4 - ARM or Thumb target

# The Thumb Instruction Set…

## Thumb Software Interrupt Instruction

The Thumb software interrupt instruction behaves exactly like the ARM equivalent and the exception entry sequence causes the processor to switch to ARM execution.

**Thumb software interrupt binary encoding:**




This instruction causes the following actions:

- The address of the next Thumb instruction is saved in r14_svc.
- The CPSR is saved in SPSR_svc.
- The processor disables IRQ, clears the Thumb bit and enters supervisor mode by modifying the relevant bits in the CPSR.
- The PC is forced to address 0x08.

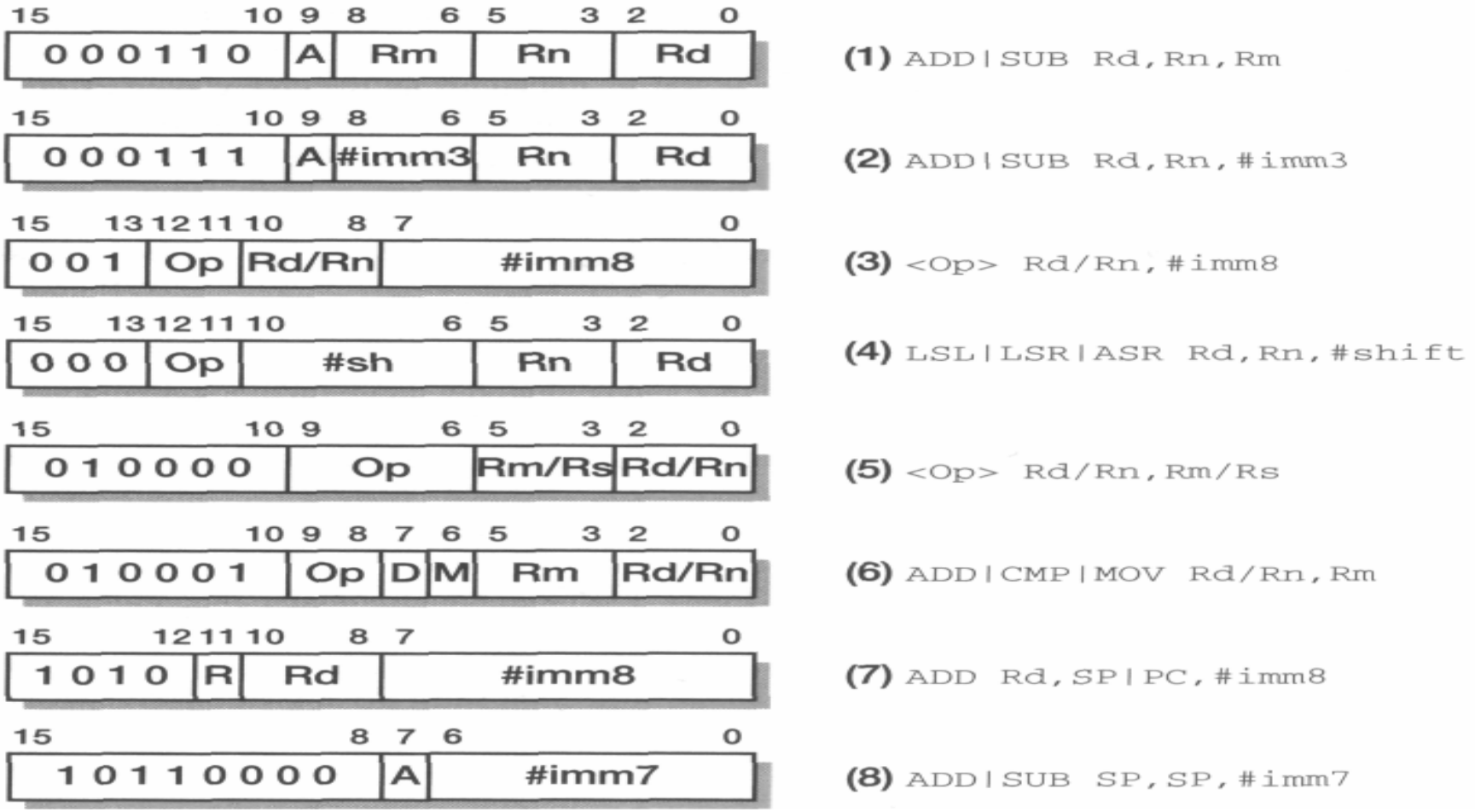
The ARM instruction SWI handler is then entered. The normal return instruction restores the Thumb execution state.

**Syntax:** **SWI <8-bit immediate>**

# The Thumb Instruction Set…

## Thumb Data Processing Instructions:

Thumb data processing instructions comprise a highly optimized set of fairly complex formats covering the operations most commonly required by a compiler.



# The Thumb Instruction Set…

| ARM instruction | | | | Thumb instruction | | | |
|---|---|---|---|---|---|---|---|
| MOVS | Rd, | #<#imm8> | | ;MOV | Rd, | #<#imm8> | |
| MVNS | Rd, | Rm | | ;MVN | Rd, | Rm | |
| CMP | Rn, | #<#imm8> | | ;CMP | Rn, | #<#imm8> | |
| CMP | Rn, | Rm | | ;CMP | Rn, | Rm | |
| CMN | Rn, | Rm | | ;CMN | Rn, | Rm | |
| TST | Rn, | Rm | | ;TST | Rn, | Rm | |
| ADDS | Rd, | Rn, | #<#imm3> | ;ADD | Rd, | Rn, | #<#imm3> |
| ADDS | Rd, | Rd, | #<#imm8> | ;ADD | Rd, | #<#imm8> | |
| ADDS | Rd, | Rn, | Rm | ;ADD | Rd, | Rn, | Rm |
| ADCS | Rd, | Rd, | Rm | ;ADC | Rd, | Rm | |
| SUBS | Rd, | Rn, | #<#imm3> | ;SUB | Rd, | Rn, | #<#imm3> |
| SUBS | Rd, | Rd, | #<#imm8> | ;SUB | Rd, | #<#imm8> | |
| SUBS | Rd, | Rn, | Rm | ;SUB | Rd, | Rn, | Rm |
| SBCS | Rd, | Rd, | Rm | ;SBC | Rd, | Rm | |
| RSBS | Rd, | Rn, | #0 | ;NEC | Rd, | Rn | |
| MOVS | Rd, | Rm, | LSL #<#sh> | ;LSL | Rd, | Rm, | #<#sh> |

These instructions all map onto ARM data processing (including multiply) instructions. Although ARM supports a generalized shift on one operand together with an ALU operation in a single instruction, the Thumb instruction set separates shift and ALU operations into separate instructions

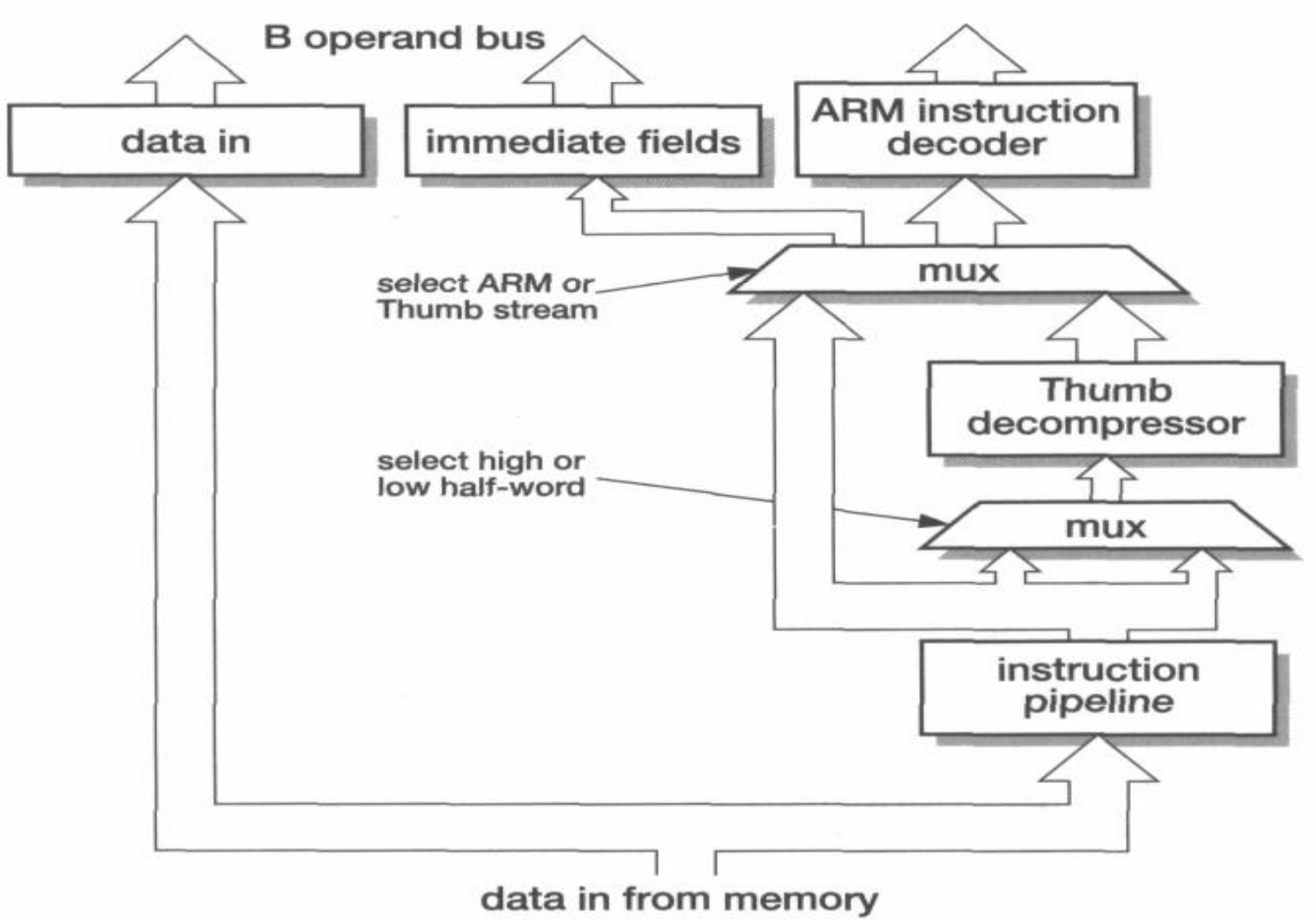
# Thumb Implementation:

The Thumb instruction set can be incorporated into a 3-stage pipeline ARM processor macrocell with relatively minor changes to most of the processor logic. The biggest addition is the Thumb instruction decompressor in the instruction pipeline; this logic translates a Thumb instruction into its equivalent ARM instruction.

The Thumb decompressor performs a static translation from the 16-bit Thumb instruction into the equivalent 32-bit ARM instruction. This involves performing to translate the major and minor opcodes, zero-extending the 3-bit register to give 4-bit specifiers and mapping other fields across as required.

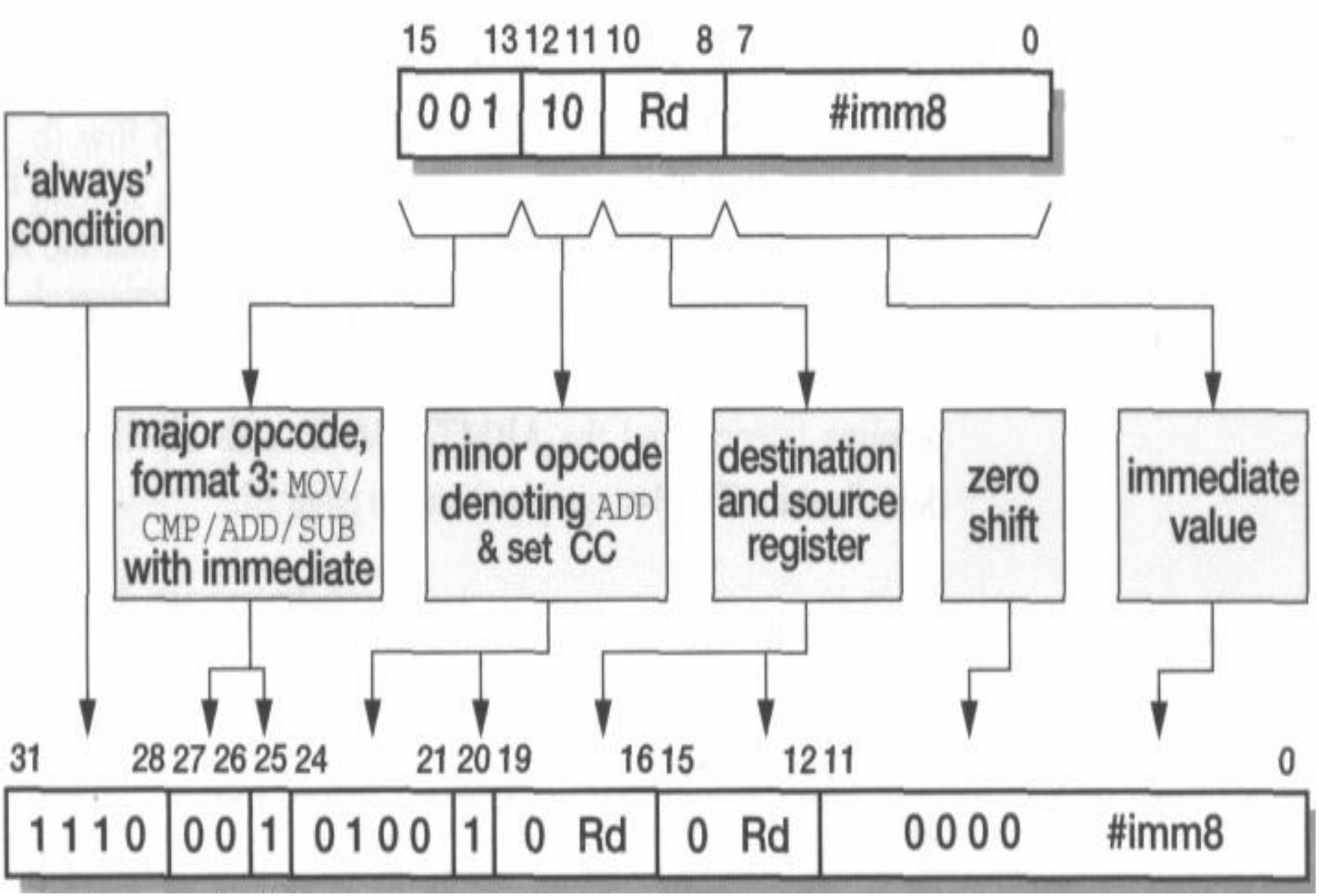
- Since the only conditional Thumb instructions are branches, the condition 'always' is used in translating all other Thumb instructions.
- Thumb data processing instruction should modify the condition codes in the CPSR is implicit in the Thumb opcode; this must be made explicit in the ARM instruction.
- The Thumb 2-address format can always be mapped into the ARM 3-address format by replicating a register specifier. The simplicity of the decompression logic is crucial to the efficiency of the Thumb instruction set.

# Thumb Implementation…




*The Thumb instruction decompressor organization*

# Thumb Implementation…




*Thumb to ARM instruction mapping*

# Thumb Applications:

Thumb instructions are 16 bits long and encode the functionality of an ARM instruction in half the number of bits, but since a Thumb instruction typically has less semantic content than an ARM instruction, a particular program will require more Thumb instructions than it would have needed ARM instructions. The ratio will vary from program to program, but in a typical example Thumb code may require 70% of the space of ARM code.

## Thumb properties

- The Thumb code requires 70% of the space of the ARM code.
- The Thumb code uses 40% more instructions than the ARM code.
- With 32-bit memory, the ARM code is 40% faster than the Thumb code.
- With 16-bit memory, the Thumb code is 45% faster than the ARM code.
- Thumb code uses 30% less external memory power than ARM code.

# Thumb Applications…

**So where performance is all-important, a system should use 32-bit memory and run ARM code. Where cost and power consumption are more important, a 16-bit memory system and Thumb code may be a better choice.**

## Thumb Systems

• A high-end 32-bit ARM system may use Thumb code for certain non-critical routines to save power or memory requirements.
• A low-end 16-bit system may have a small amount of on-chip 32-bit RAM for critical routines running ARM code, but use off-chip Thumb code for all non-critical routines.

- Thumb code will enable off-chip ROMs to give good performance on an 8- or 16-bit bus, saving cost and improving battery life.
- Mobile telephone and pager applications incorporate real-time digital signal processing (DSP) functions that may require the full power of the ARM.

# Writing Simple Assembly Language Programs:

**'Hello World' program**

ARM Program:

```
        AREA    HelloW,CODE,READONLY
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
        ENTRY                       ; code entry point
START   ADR     r1, TEXT            ; r1 -> "Hello World"
LOOP    LDRB    r0, [r1], #1        ; get the next byte
        CMP     r0, #0              ; check for text end
        SWINE   SWI_WriteC          ; if not end print ..
        BNE     LOOP                ;  .. and loop back
        SWI     SWI_Exit            ; end of execution
TEXT    =       "Hello World",&0a,&0d,0
        END                         ; end of program source
```

# Writing Simple Assembly Language Programs…

**'Hello World' program**

Thumb Program:

```
        AREA    HelloW Thumb,CODE,READONLY
SWI WriteC      EQU    &0        ; output character in r0
SWI_Exit        EQU    &11        ; finish program
        ENTRY                      ; code entry point ;
        CODE32                     enter in ARM state

        ADR     r0, START+1        ;get Thumb entry address
        BX      r0                 ;enter Thumb area
        CODE16                     ;Thumb code follows..
START   ADR     r1, TEXT           ;r1 -> "Hello World"
LOOP    LDRB    r0, [r1]           ;get the next byte
        ADD     r1, r1, #1         ;increment pointer  **T
        CMP     r0, #0             ;check for text end
        BEQ     DONE               ;finished?            **T
        SWI     SWI WriteC         ;if not end print . .
        B       LOOP                 ;.. and loop back
DONE    SWI     SWI Exit           ;end of execution
        ALIGN                      ;to ensure ADR works
TEXT    DATA
                "Hello World",&0a,&0d,&00
        END
```

# REFERENCE BOOKS




- Lyla B. Das, "Embedded Systems: An Integrated Approach", 1st Edition, Pearson, 2012.
- Raj Kamal, "Embedded Systems Architecture, Programming and Design", 2nd Edition, McGraw Hill, 2009.
- Kenneth J. Ayala, "The 8051 microcontroller : Architecture, Programming", 3rd Edition, Thomson Learning, 2007.
- David E. Simon, "An Embedded Software Primer", 1st Edition, Pearson, 2008.
- Marilyn Wolf, "Computers as Components - Principles of Embedded Computing System Design", 3rd Edition, Morgan Kaufmann Publisher (Elsevier), 2012.
- Jean J. Labrosse, "Embedded System Building Blocks: Complete and Ready-to-Use Modules in C", 2nd Edition, CRC Press, 1999.
- Frank Vahid and Tony Givargis, "Embedded System Design: A Unified Hardware/Software Introduction", 3rd Edition, John Wiley & Sons, 2006.
- K.V.K.K. Prasad, "Embedded Real-Time Systems: Concepts, Design & Programming", Dream Tech Press, 2005.
- Steve Furber, "ARM System on Chip Architecture", Addison Wesley, 2nd Ed., 2000.
- Andrew N Sloss, Dominic Symes and Chris Wright, "ARM system developers guide", Morgan Kaufmann Publisher-Elsevier, 2008.

# THANK YOU...