## Contents

# Chapter – 7 ARM and THUMB Instruction sets

**Learning objectives:**

**After reading this chapter, reader can understand the following:**

- Clear understanding on instruction set supported by ARM
    1. **Data processing instructions.**
    2. **Conditional instructions**
    3. **Load and store instructions**
    4. **Multiply instructions**
    5. **Software interrupt instructions**
    6. **Branch instructions**
    7. **Barrel shifting operation**
- Clarity on THUMB instruction set.
- Assembly level programming for ARM Core.

## 7.1 Introduction

As all other microcontrollers, ARM has got set of instructions for ARM state and THUMB state. Reader will be taken through the instructions carefully one after another with a clear description. There are many instructions available in ARM state and the attention will be paid to THUMB state instructions after dealing with ARM instructions. First the summary of all available instructions is presented and then each instruction is explained.

Instruction Set is broadly divided into the following areas:

1. Data processing instructions.
2. Conditional instructions
3. Load and store instructions
4. Multiply instructions

5. Software interrupt instructions
6. Branch instructions
7. Barrel shifting operation

Reader will be taken through all of these one by one. Few points to be noted before jumping deeper into the instruction sets are:

a. All ARM state instructions are 32 bits instructions.
b. All instructions will need 3 operands.
c. Most of the instructions could be executed in one cycle.

## 7.2 Data processing instructions

The data processing instructions are comprised of Arithmetic instructions, logical instructions, data movement instructions and comparison instructions as well. The summary of all the available data processing instructions are presented in the following table 7.1. From that table all the instructions will be dealt in a brief.

Table 7.1 Data processing instructions

| OPCODE | Mnemonic | Operation Performed | Brief Interpretation |
|---|---|---|---|
| 0000 | AND | Rd=RN AND OP2 | Logical Bitwise AND operation |
| 0001 | EOR | Rd=RN XOR OP2 | Logical Bitwise EXOR operation. |
| 0010 | SUB | Rd=RN-OP2 | Subtract |
| 0011 | RSB | Rd=OP2-OP1 | Reverse Subtraction |
| 0100 | ADD | Rd=RN+OP2 | Addition |
| 0101 | ADC | Rd=RN+OP2+C | Addition with carry. |
| 0110 | SBC | Rd=RN+ OP2+C-1 | Subtract with carry(borrow). |
| 0111 | RSC | Rd=OP2- RN+C-1 | Reverse subtraction with carry. |
| 1000 | TST | SCC RN AND OP2 | Test operation |
| 1001 | TEQ | SEC RN EXOR OP2 | Test for equivalence |
| 1010 | CMP | SEC RN—OP2 | Compare the two operands. |
| 1011 | CMN | SEC RN+OP2 | Negate Compare of the operands. |
| 1100 | ORR | Rd= RN OR OP2 | Logical bitwise OR |

|  |  |  |  |
|---|---|---|---|
| **1101** | **MOV** | **Rd =OP2** | **MOVE operation** |
| **1110** | **BIC** | **Rd = RN NAND OP2** | **Clears the bit in perspective** |
| **1111** | **MVN** | **Rd=NOT OP2** | **Negate the Operand in perspective.** |

Taking the easier and most frequently used Arithmetic instructions first, the following table 7.2 is framed. Rn mentioned in the table is nothing but one of the registers that reader has come across in the figure 6.2.

## *7.2.1 Arithmetic operations*

Following instructions fall in the category of arithmetic instructions and they are the back bone of all the arithmetic operations that are carried out.

Table 7.2 Arithmetic Instructions

| Instruction | Description with syntax |
|---|---|
| **ADD** | **S**elf explanatory instruction. It is meant for adding two operands and result will be sent to the third operand specified. It is referred to be as Rd. Instruction format is **ADD operand1 + operand2.** This is a 32 bit addition operation instruction.<br><br>**ADD r0, r1, r2;** // instruction for adding two 32 bit numbers.<br><br>Where r1 and r2 will have the operands to be added and result will be moved to the destination register r0. r0 is seen as the Rd here.<br><br>**ADD R0,R0,#1**<br><br>Increment R0 by 1, where # represents the immediate addressing mode of operation. |
| **ADC** | Add with carry operation can be performed with ADC. Instruction format is similar to that of ADD. It will have an added responsibility of adding with carry. That's it.<br><br>**ADC operand1 + operand2 + carry;** // instruction for adding |

| | |
|---|---|
| | two operands with the carry. |
| **SUB** | As simple as ADD instruction. To subtract two operands one can use this instruction.<br><br>**sub r0, r0, r1 ; //subtract r1 from r0**<br><br>This instruction subtracts the **<r1>** operand from the **<r0>** operand, storing the result in **<r0>**. The subtraction is a thirty-two bit signed operation.<br><br>*Example:*<br><br>**SUB R0, R0, #1 ;** Decrement R0, this is an immediate mode of addressing with having immediate data in place. This instruction will decrement R0 by 1. |
| **SBC** | Subtract with Borrow (i.e. similar to add with carry).<br><br>SBC operand1 - operand2 + carry – 1; // Instruction used to subtract with borrow. |
| **RSB** | This instruction performs subtraction without carry, but reverses the order in which its operands are subtracted. The instruction:<br><br>**RSB <dest>,<lhs>,<rhs>**<br><br>Will do the operation,<br><br>**<dest> = <rhs> - <lhs>**<br><br>**Example:**<br><br>**RSB R0, R1, #2; R0** = 2 - R0 will be performed. |
| **RSC** | With carry version of RSB instruction. |

### *7.2.2 Logical operations*

The next step for the reader to look into is the logical instructions. Table 7.3 presents the list of available logical instructions which comes under the roof of data processing instructions.

Table 7.3 Logical Instructions

| Instruction | Description |
|---|---|
| **AND** | All of the readers must be aware of the logical ANDing. A truth table is presented below with two operands and the results. d below: |

| R0 | R1 | R0 AND R1 |
|---|---|---|
| **0** | 0 | 0 |
| **0** | 1 | 0 |
| **1** | 0 | 0 |
| **1** | 1 | 1 |

In the ARM AND instruction, this operation is applied to all 32 bits in the operands. That is, bit 0 of the operand1 is ANDed with bit 0 of the operand2 and stored in bit 0 of the destination register mentioned in the instruction. It will go on like this till the 32nd bit.

**Example:**

AND R0, R0, R5

| Instruction | Description |
|---|---|
| **EOR (Exclusive OR)** | Similarly XOR operation would be performed bit wise here with this instruction. A truth table is presented below for the same: |

| R0 | R1 | R0 EOR R1 |
|---|---|---|
| **0** | 0 | 0 |
| **0** | 1 | 1 |
| **1** | 0 | 1 |
| **1** | 1 | 0 |

In the ARM AND instruction, this operation is applied to all 32 bits in the operands. That is, bit 0 of the operand1 is XORed with bit 0 of the operand2 and stored in bit 0 of the destination register mentioned in the instruction. It will go on like this till the 32nd bit.

EOR  R0,R0,#1 ;'Toggle' state of bit 0

**ORR (logical OR)**

This will perform logical OR operations bit wise. As ever, a truth table is presented for the reader to understand the story better.

| R0 | R1 | R0 OR R1 |
|---|---|---|
| **0** | 0 | 0 |
| **0** | 1 | 1 |
| **1** | 0 | 1 |
| **1** | 1 | 1 |

In the ARM OR instruction, this operation is applied to all 32 bits in the operands. That is, bit 0 of the operand1 is ORed with bit 0 of the operand2 and stored in bit 0 of the destination register mentioned in the instruction. It will go on like this till the 32nd bit.

Example: **`ORRS R0,R0,R1 ;`**

**BIC (BIT Clear)**

This instruction is meant for getting the NOT of RHS (second operand) and with that result AND with first operand will be done. An example will be handy here in the form of truth table.

This will clear the <lhs> bit if the <rhs> bit is set, and leaving the <lhs> bit unaltered otherwise.

| <lhs> Operand 1 | <rhs> Operand 2 | NOT of <rhs> Operand 2 | <lhs> AND NOT <rhs> |
|---|---|---|---|
| **0** | 0 | 1 | 0 |
| **0** | 1 | 0 | 0 |
| **1** | 0 | 1 | 1 |
| **1** | 1 | 0 | 0 |

Example: **`BIC R0,R0,R1 ;`**

### *7.2.3 Comparison operations*

Well, the reader can next move to the Comparison operations supported by ARM core. Table 7.4 summarizes the set of comparison instructions.

Table 7.4 Comparison Instructions

| Instruction | Description |
|---|---|
| **CMP** | Compare instruction and the operation of the comparison is carried out by subtracting the two operands that are being subtracted. It does not store the result anywhere. Instead status bits will be set accordingly based on the result of the operation.<br><br>Example:<br><br>CMP  R0,R1; Get greater of R1, R0 in R0 |
| **CMN** | Same is the purpose as the previous one, but there is a small change here. There will be a negation operation of the second operand (rhs) before the comparison is being performed. One simple example would do the task here.<br><br>Example:<br><br>CMN  R1,#1  ;Compare R1 with -1 |
| **TST** | It is used to perform operand1 AND operand2. Similar to Logical ANDing. |
| **TEQ** | It is used to perform the XOR operations on the two operands passed as arguments. The flags will be set based on the results. And the syntax is very simple and easier to remember:<br><br>TEQ  <lhs>,<rhs><br><br>One important use of TEQ is to test if two operands are equal, that too without affecting the state of the C flag as CMP could do. After such an operation, Z=1 if the operands are equal, or 0 if not.<br><br>Example:<br><br>TEQ R0, #0; See if R0 = 0. |

## *7.2.4 Data movement operations*

This is the last set of instructions which fall under Data processing instructions. The name itself clearly suggests that these set of instructions help in moving instructions from one register to another. The table 7.5 summarizes the list of data movement instructions supported by ARM state of operation with the ARM core.

Table 7.5 Data movement Instructions

| Instruction | Description |
|---|---|
| **MOV and MOVS** | MOV instruction moves data from the source to destination. The syntax of the instruction is similar to that of the 8051 microcontroller.<br><br>MOV &lt;dest&gt;,&lt;Source&gt;<br><br>Example:<br><br>MOV r0, r1; put R1 into R0.<br><br>One more addition in the learning process is usage of S with the instructions.<br><br>Example with flags:<br><br>MOVS r0, r1; put R1 into R0, setting the flags.<br><br>MOV r0, #01; Immediate addressing mode |
| **MVN** | MVN instruction moves the logically NOTed value of its operand &lt;RHS&gt; to the register specified by &lt;dest&gt;.<br><br>Syntax:<br><br>MVN &lt;dest&gt;,&lt;Operand&gt;<br><br>Example:<br><br>MVNS R0, R0 ;Invert all bits of R0, setting flags |

## 7.3Conditional Execution

Almost all ARM instructions have the ability of conditional execution. The conditions that are available for operation are listed in the table 7.6 below.

Table 7.6 conditional codes

| OPCODE | Mnemonic | Flag Status | Brief Interpretation |
|---|---|---|---|
| **0000** | EQ | Z is set | Equal to condition is checked. |
| **0001** | NE | Z is cleared | Not Equal condition is checked. |
| **0010** | CS/HS | C is Set | Carry bit is Set/ Unsigned Greater than Equal to condition is checked. |
| **0011** | CC/LO | C is Cleared | Carry bit is cleared/ Lesser than condition is tested. |
| **0100** | MI | N is Set | Signed Negative condition is evaluated |
| **0101** | PL | N is Cleared | Signed Positive or Zero condition is evaluated. |
| **0110** | VS | V is Set | Occurrence of Overflow is checked. |
| **0111** | VC | V is Cleared | Non-Occurrence of Overflow is checked. |
| **1000** | HI | C is Set & Z is Cleared | Unsigned Greater than Condition |
| **1001** | LS | C is Cleared or Z is Set | Unsigned Less Than or Equal Condition. |
| **1010** | GE | N equals V | Signed Greater than or Equal  Condition |
| **1011** | LT | N is not V | Signed Less Than Condition. |
| **1100** | GT | Z is Cleared & N equals V | Signed Greater than. |
| **1101** | LE | Z is Set & N not V | Signed Less Than. |
| **1110** | AL | - | Unconditional Checking (Occurs Always). |

|  |  |  |  |
|---|---|---|---|
| **1111** | NV | NO change | Never Occurs. |

Reader might have got a query by now on how these can be used? It is not that tough. Few examples would be helpful for the reader.

When someone wishes to execute an instruction conditionally, post fixing the instruction with appropriate condition code would do.

For an example a simple add instruction would be like

**ADD r0, r1, r2;** r0 = r1 + r2 will be performed and result will be stored in r0.

To include a simple condition as the instruction should execute if at all zero flag is set, then the instruction should be altered like,

**ADDEQ r0, r1, r2;** if zero flag set then add.

By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect).To get the conditional flags hit and affected, the S bit of the instruction needs to be set by post fixing the instruction (and any condition code) with an "S". •ADDS r0, r1, r2; would be the best example. Reader has been introduced with this some time back in MOV and MOVS instructions clearly. Here it could serve as a recap.

## 7.4 Load and Store Instructions

Since we are already aware of a fact that ARM uses a LOAD and STORE architecture, there cannot be an operation of data movement without using the registers. So the registers are mandatory for accomplishing any sort of data transfer operation. Load and Store instructions are meant for the data transfer from memory to registers and vice versa. The following table 7.7 summarizes the available instructions.

There are three major classes of load and store instructions:

- Single register data transfer and for this purpose ARM provides LDR / STR.
- Block data transfer and LDM/STM are used for accomplishing block data transfer.
- Single Data Swap can be performed by SWP instruction.

Table 7.7 Load and Store instructions

| Instruction | Description |
|---|---|
| **STR and STRB** | One should first decide on what is the operation to be done. Whether to read the data from the memory or storing the data into the memory. STR is meant for storing the data into the memory. When someone wishes to store the data onto the memory, the location where the data has to be stored needs to be specified. So mentioning the address becomes inevitable. One can quote the address in one of the two ways. They are<br><br>• Pre-indexed addressing<br>• Post-indexed addressing.<br><br>One simple example would do the stuff. First let the reader be introduced with pre-indexed addressing.<br><br>**<u>Pre-indexed addressing:</u>**<br><br>Syntax for the pre-indexed addressing is presented below:<br><br>**STR{cond} <source>,[<base>{,<offset_value>}]** |

|  |  |
|---|---|
|  | So what is what in the syntax should be known now:<br><br>&lt;source&gt; is the source which holds the data to be transferred.<br><br>&lt;base&gt; is the register which contains the base address of the memory location required for storage..<br><br>&lt;offset&gt; is a field which is not mandatory actually. It may be used and in case of it being used, then the address to be calculated can be found out with the following equation.<br><br>Address where data has to be put = &lt;base&gt;+&lt;offset&gt;<br><br>Example usage:<br><br>**STR R0,[R1,#25]**<br><br>Above operation will store the data available at R0 to the address obtained by adding the base address specified by R1 + 25. Offset need not be just positive always. It can be negative as well. Following simple example will reveal how it can be.<br><br>**STR R0,[R1,#-25]**<br><br>Which stores R0 at R1-25.<br><br>STRB can be used for performing operations based on bytes. STR alone will help in getting the storage operations for words.<br><br>**<u>Post-indexed addressing:</u>**<br><br>This is the second available addressing mode. POST, means after the completion. It is simple; offset would not be added to the base address until the instruction has been executed.<br><br>**STR{cond} &lt;srce&gt;,[&lt;base&gt;],&lt;offset&gt;** |
| LDR | This is yet another instruction that comes under load and store category. This instruction is helpful in loading.<br><br>The syntax is simple and the following it can be represented.<br><br>**LDR &lt;dest&gt;,&lt;expression&gt;** |

| | |
|---|---|
| | Where **`<expression>`** yields an address.<br><br>Here for this case the PC will be used as the base register. In the case of being address not falling in right range there wills an error message for sure. -4095 to +4095 is the range. |

## 7.5 Multiplication Instructions

There are a set of instructions available and set beside for getting the multiplication operations done. They are all listed and explained in detail as follows in table 7.8. There are two basic instructions supported in ARM for multiplication. They are MUL and MLA. The syntaxes and other related information is summarized in the following table 7.8

Table 7.8 Multiplication instructions

| Instruction | Description |
|---|---|
| **MUL** | Syntax can first be presented here and then, the explanation could be given.<br><br>Syntax:<br><br>**MUL <dest,<lhs>,<rhs> ;** which can be framed as **MUL R0, R1, R2;** where R0 is the destination register where the result of the operation would be stored and R1 and R2 are the operands to be multiplied.<br><br>One thing to note here is, no immediate addressing mode is supported in ARM Multiplication instruction MUL. |
| **MLA** | It is expanded as multiply accumulate. Syntax is needed here as well for someone to understand the instruction with ease. It is closely similar to MUL instruction.<br><br>Syntax:<br><br>**MLA <dest>,<lhs>,<rhs>,<add>;** which can be framed as MLA R0, R1,R2, R10; where R0 will be have the end result. R1 and R2 are the operands to be multiplied and before storing the results, the ADD register's value will be added with the product obtained and then it will be store in the destination register. I.e. R0 in this case.<br><br>In simple words, MLA is again the similar multiplication, but adds register <add> before storing the result. |

## 7.6 Software interrupt instructions

It is a simple and most important instruction that one could refer to immediately. It is just one instruction SWI, whose assembler format will not have any variants or options.

When SWI is run, the CPU will immediately enter the supervisor mode and will save the return address in R14_SVC (reader may recollect the way the registers are framed and modes of operation). From the time the supervisor mode is entered, OS will be taking over and set of operations requested by the USER mode will be performed. . SWI has to be framed carefully as way in which SWI is run is solely dependent on the expression part of the instruction.

## 7.7 Branching instructions

There are two kinds of branching instruction supported in ARM core. They are normal branching and branching with link. Reader will be introduced one after another in a descriptive manner. The following table 7.9 has the branching instructions summarized.

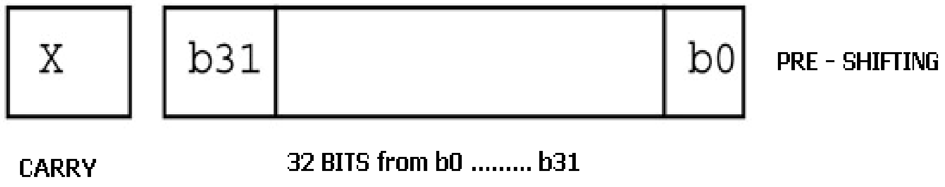
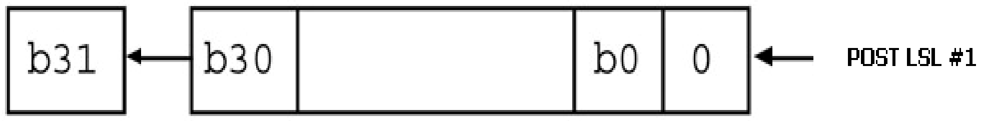
Table 7.9 Branching instructions

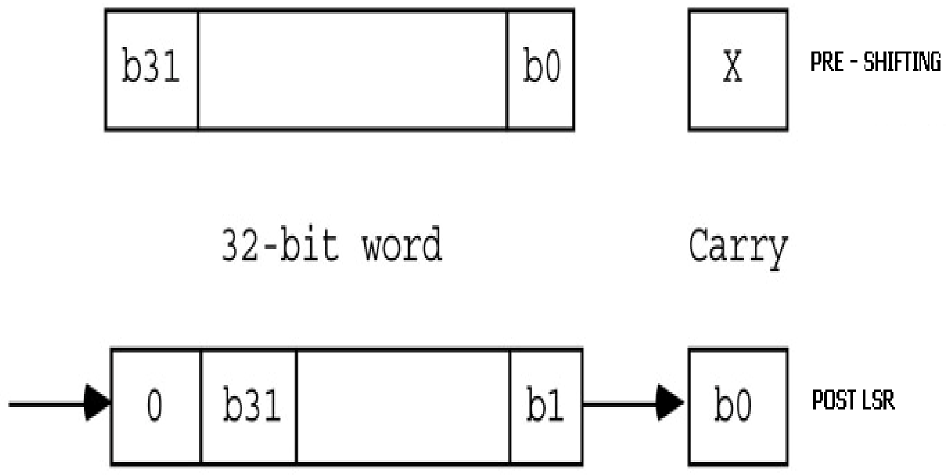
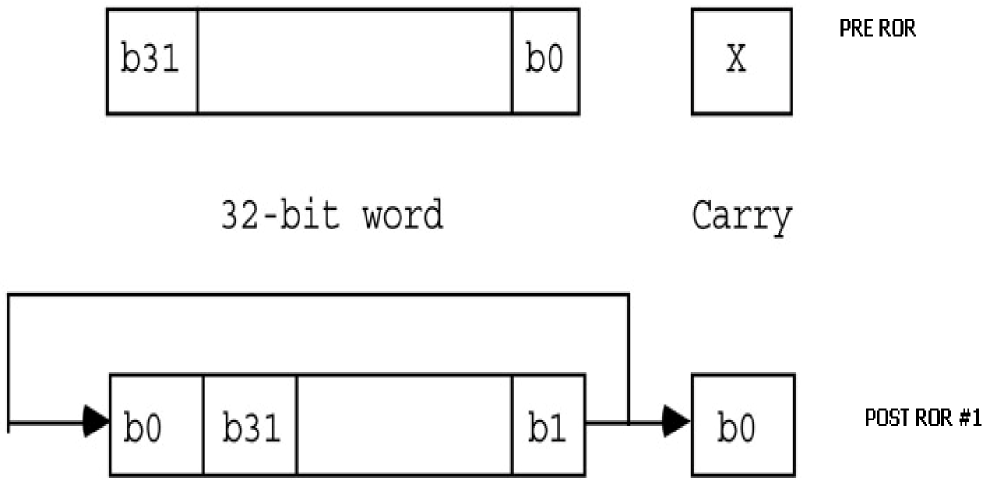
| Instruction | Description |
|---|---|
| **B** | It is a simple branching operation where the instruction just a single letter B.<br><br>Syntax of the instructions is presented as follows: B <expression><br><br><expression> specified the address within the program to which the control has to be transferred. Normally labels will be used for this purpose and the same would be defined somewhere else in the code.<br><br>***B loop;*** Branching to loop and the same can be defined somewhere in the code.<br><br>***.Loop ADD R1, R2, R3;*** The branch can come here and can work on the task designated. |
| **BL** | B is having one more variant of its kind. It is nothing but BL which is expanded as Branch with Link. This instruction has to perform a link operation before the branching has to be performed. It is kind of keeping a paper mark. Current value of the R15 register should be stored in R14 and the branching action can then be taken.<br><br>The address that ARM CORE saves in R14 is the address of the instruction that immediately follows the **`BL`**. So task becomes easier here. Returning from subroutine is simple. An action of just moving the content of R14 back to R15 would do the task.<br><br>**`MOVS R15,R14`**<br><br>Since S is also included in the instruction shown above, the status of flags will also be restored automatically.<br><br>**`BL`**'s syntax:<br><br>**`BL <expression>`** |

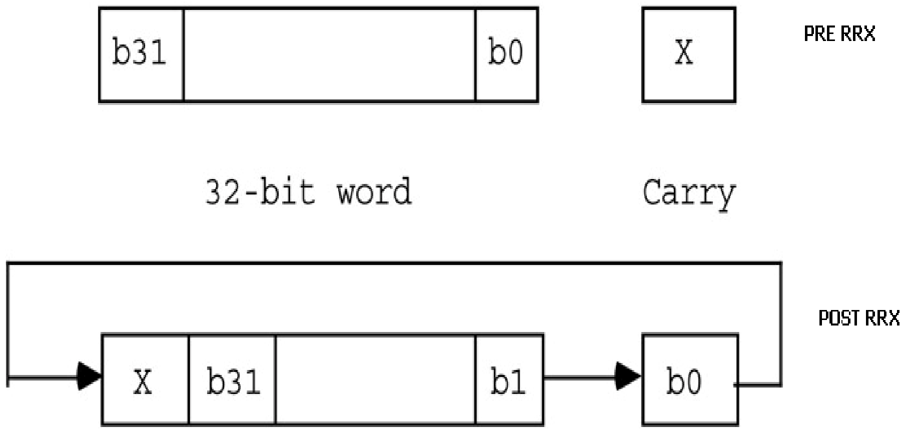
## 7.8 Barrel shifting operation

ARM has no support for the shift instructions. Barrel shifter is used supporting the shifting operations. Reader will be taken through the barrel shifting operations shortly.

Table 7.10 Shifting instructions

| Instruction | Description |
|---|---|
| **LSL** | LSL is expanded as Logical Shift Immediate. It takes the number of bits to be shifted as the argument. Syntax is very simple and is presented as follows:<br><br>LSL #n where, n is the number of the bits by which the value is shifted. One simple example would depict the operation clearly. Reader is already aware that 32 bit operations are supported in ARM state.<br><br> X<br>b31<br>b0<br>PRE - SHIFTING<br>CARRY<br>32 BITS from b0 ......... b31<br><br>After n shifts (i.e. number specified in the syntax), n zero bits have been shifted in on the right side as shown below and the carry bit is set to bit 32-n of the original word.  One<br><br> <br><br>One question may get raised now. What will be case if the programmer specifies LSL #0? Nothing is the effect. LSL can be used in sync with instructions. One such example is presented below with **MOV r3, r4, LSL #2**<br><br>**Pre Execution:**<br><br>Assuming r3=5 and r4 = 8.<br><br>**Post Execution:**<br><br>The example shown above will multiply r4 by 4 and then will move the result to r3.<br><br>When the instruction is executed: r3 = r4 * 2 ^2. Since the value 2 is specified in the LSL it has been taken as 2 ^2. Similarly if it is LSL #n it will be taken as 2 ^ |

| | |
|---|---|
| | n. ASL is similar to that of LSL. |
| **LSR** | It is logical shift right. A diagrammatic explanation would get a better understanding.<br><br>An **LSR** by one bit is shown below in the following figures:<br><br> b31<br>b0<br>X<br>PRE - SHIFTING<br>32-bit word<br>Carry<br>0<br>b31<br>b1<br>b0<br>POST LSR<br><br>The diagrams are self explanatory and easy for someone to understand it. |
| **ROR** | Rotate Right is the expansion of ROR. An example as ever will help the reader in understanding the instruction better.<br><br> b31<br>b0<br>X<br>PRE ROR<br>32-bit word<br>Carry<br>b0<br>b31<br>b1<br>b0<br>POST ROR #1<br><br>Here the instruction ROR #1 is taken for explanation and the picture is self explanatory. One can see from the second picture that the bit-0 is rotated as shown in the figure. There is no instruction called ROL. |

| | |
|---|---|
| **RRX** | Rotate Right Extended is abbreviated as RRX. This is a unique instruction and there is no need for specifying the count. It will anyways rotate by one bit always. The diagrammatic approach will help the reader in getting through this instruction.<br> b31<br>b0<br>X<br>PRE RRX<br>32-bit word<br>Carry<br>POST RRX<br>X<br>b31<br>b1<br>b0<br>From the figure shown above one can easily understand that the bit 0 is shifted into the carry slot and old content of the carry is shifted into bit 31. |

## 7.9 Stack in ARM

Stack, should be a familiar term for the reader by now. It has been completely discussed with respect to 8051 and the same is going to be handled for the ARM controller as well. There are two instructions that come into play when someone talks on STACK in concurrence with ARM.

They are LDM and STM. They can be related to PUSH and POP of 8051. LDM and STM are used for the purpose of storing and retrieving actions. As reader should be aware of, stack is a memory area which can grow on addition of data to it and will obviously shrink when data is popped out of it. POPPING will happen first for the lastly added data; simply saying it is nothing other than well known term called as LIFO, Last in First Out. A simple example for showing up the LIFO's working is presented below in figure 7.1

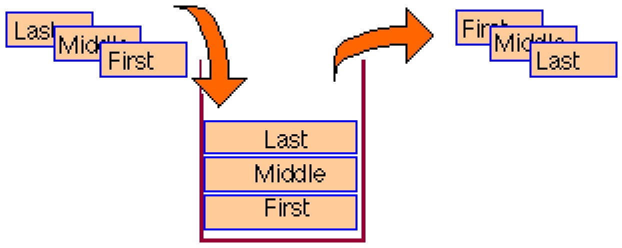


Figure 7.1 LIFO

Well, the next terminology to know is the stack pointer. It is used to point the stack. In ARM there is no special register called as a stack pointer. But instead it will be stored in any one of the available general purpose registers.

A diagrammatic representation would be useful for understanding the SP's functionality. Figure 7.2 can be referred for the purpose.

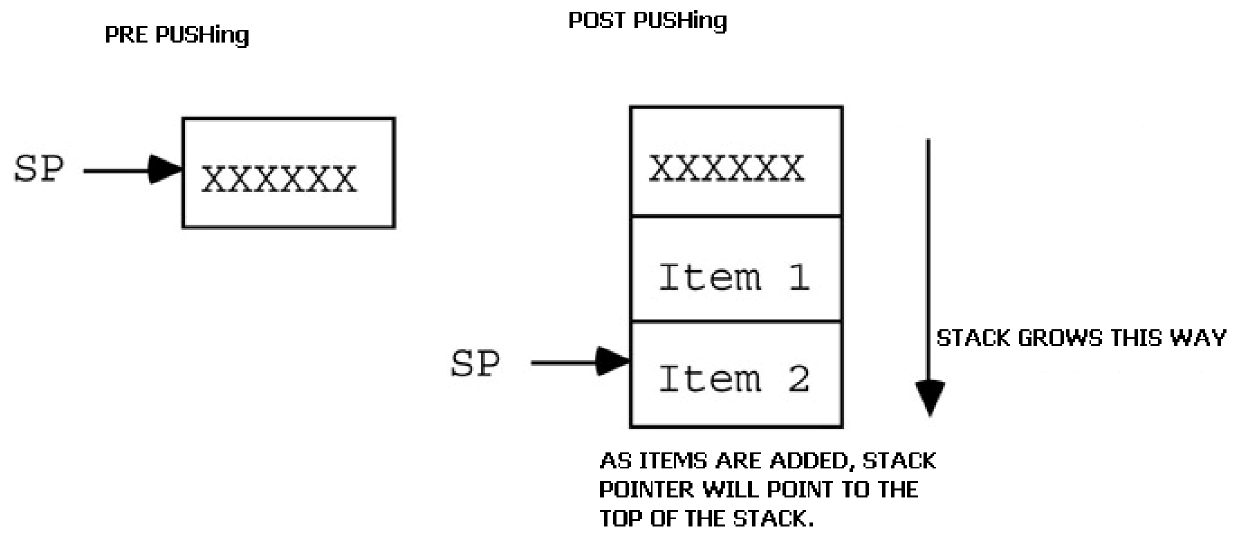


Figure 7.2 Stack growth

Pre PUSHing an item SP will be pointing to the previously added item of the stack. When an item is getting added to the stack, SP will point to the new item that that has been added. Once one item is moved out of stack, SP will point to the previous item itself. Before PUSHing or POPing two things should be remembered. When someone tries to add an item to the stack, it should not be full. There should be enough space for holding the new entrant. When POPing, there should be enough data to be popped out. I.e. one cannot POP out of empty stack.

**STM:**

STM is the instructions which help in PUSHing items onto the stack. The syntax of the instruction

```
STM<type> <base>{!}, <registers>
```

<Type> can be one of the four letters which serves as a mark of stack being full or empty with E or F. Similarly ascending or descending mode of the stack is specified by A or D.

Next thing in the syntax is <base>.Base register is nothing but the stack pointer. ! Option is also present in the syntax and it will cause the write back. Next thing is the <registers> and it will specify the registers which have to be pushed onto the stack.

Our first example of an STM instruction is:

STMED R13! {R1, R2, R5}

R1, R2 and R5 will be saved and R13 will serve as the stack pointer. E is the option specified and stack is Empty.

## <u>One more format of STM</u>

Following is the simple instruction STMFD R13, {R1-R15}, where with this information STM will store all the registers from R1 to R15, without affecting the register R13.

## 7.10 Some sample programs with ARM Core

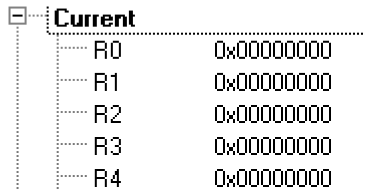
Here are the sample programs with the ARM core have been presented. Reader might not just stop here. More can be worked out in the same area. Reader can take this as a good start and can work out more complex problems.  Keil for ARM has to be use for the compilation of the following codes. Once installed the same procedure as discussed in the execution of 8051 has to be followed. The only difference is in selecting the controller model and manufacturer name.

**Example: 1**

```
AREA  RESET, CODE, READONLY
; AREA is a directive which helps in specifying the region where the code
; has to be stored. Here it is RESET. There are other options as well.
ENTRY
   MOV R0, #0x1
              ;  Loading Register R0 with 0x1;
   MOV R1, #0xA
              ;  Loading Register R1 with 0xA;
   ADD R3, R0, R1
              ; Adding them together to get the result stored in R3.
   B      .
              ; looping. BL can also be used.
END
```

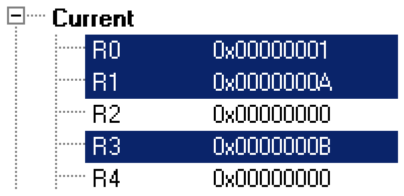
Execution Result:

**Before Execution**



```
Current
   R0     0x00000000
   R1     0x00000000
   R2     0x00000000
   R3     0x00000000
   R4     0x00000000
```

**After Execution**



```
Current
   R0     0x00000001
   R1     0x0000000A
   R2     0x00000000
   R3     0x0000000B
   R4     0x00000000
```

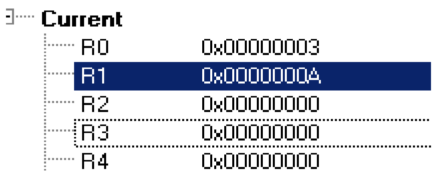
**Example: 2**

Write a program to SWAP two numbers with ARM state instruction set.

```
AREA  RESET, CODE, READONLY
; AREA is a directive which helps in specifying the region where the code
; has to be stored. Here it is RESET. There are other options as well.
ENTRY
   MOV R0, #0x3
   ; First value is moved to R0
   MOV R1, #0x0A
   ; Second value is moved to R1
   MOV R2, #0
   ; Keeping the third register, swapping is initiated.
   MOV R2, R0
   ; Copying the content of R0 to third register R2.
   MOV R0, R1
   ; So now, R1 can be moved to R0.
   MOV R1, R2
   ; Restore R0's default value in R1. Swapping is done.
   B      .
   ; looping. BL can also be used.
END
```

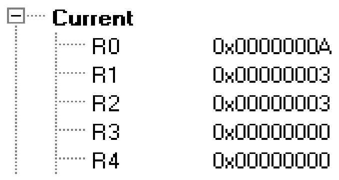
**Execution Result**

Before Execution



```
Current
   R0      0x00000003
   R1      0x0000000A
   R2      0x00000000
   R3      0x00000000
   R4      0x00000000
```

After Execution



```
Current
   R0      0x0000000A
   R1      0x00000003
   R2      0x00000003
   R3      0x00000000
   R4      0x00000000
```

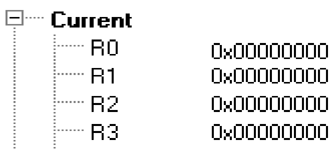
**Example: 3**

Write a program to get one’s complement of a value.

```
AREA  RESET, CODE, READONLY
; AREA is a directive which helps in specifying the region where the code
; has to be stored. Here it is RESET. There are other options as well.
ENTRY
  MOV R0, #0x0
  ; Value to be complemented.
  MVN R0, R0
  ; MVN will NOT (Invert the value and will store it in R0)
  B     .
; looping. BL can also be used.
END
```

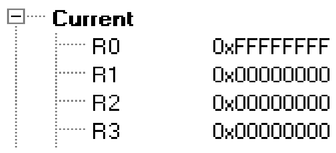
**Execution Result**

Before Execution



```
Current
  R0          0x00000000
  R1          0x00000000
  R2          0x00000000
  R3          0x00000000
```

After Execution



```
Current
  R0          0xFFFFFFFF
  R1          0x00000000
  R2          0x00000000
  R3          0x00000000
```

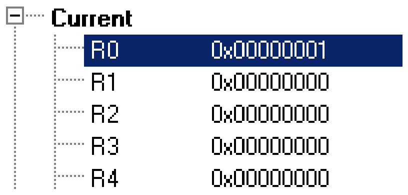
**Example: 4**

Write a program to get two’s complement of a value.

```
AREA  RESET, CODE, READONLY
; AREA is a directive which helps in specifying the region where the code
; has to be stored. Here it is RESET. There are other options as well.
ENTRY
  MOV R0, #0x1
  ; Value to be complemented.
  MVN R0, R0
  ; MVN will NOT (Invert the value and will store it in R0)
  ADD R1, R0, #1
  ; Adding 1 to 1's complemented result will fetch the two's complemented result.
  B     .
 ; looping. BL can also be used.
END
```

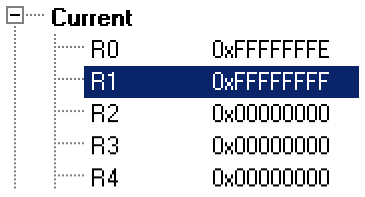
**Execution Result**

Before Execution



```
Current
  R0          0x00000001
  R1          0x00000000
  R2          0x00000000
  R3          0x00000000
  R4          0x00000000
```

After Execution



```
Current
  R0          0xFFFFFFFE
  R1          0xFFFFFFFF
  R2          0x00000000
  R3          0x00000000
  R4          0x00000000
```

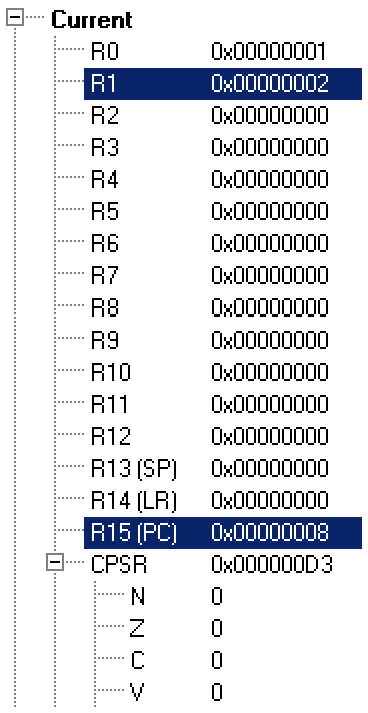
**Example: 5**

Write a program to find greatest of two numbers.

```
AREA  RESET, CODE, READONLY
; AREA is a directive which helps in specifying the region where the code
; has to be stored. Here it is RESET. There are other options as well.
ENTRY
  MOV R0, #0x1
  ; Moving the first value.
  MOV R1, #0x2
  ; Moving the second value to be compared with the first one.
 CMP R0, R1
 ; Comparing, result would not be stored somewhere. Instead CPSR
            ; can be analysed for the flag status. Here N flag will be set as
            ; R0 is having value smaller that R1. N is negative flag.
            ; Incase R0 has greater value that R1, then N flag won't be set
            ; Incase they are equal, Zero flag will be set.
            B      .
              ; looping. BL can also be used.
END
```

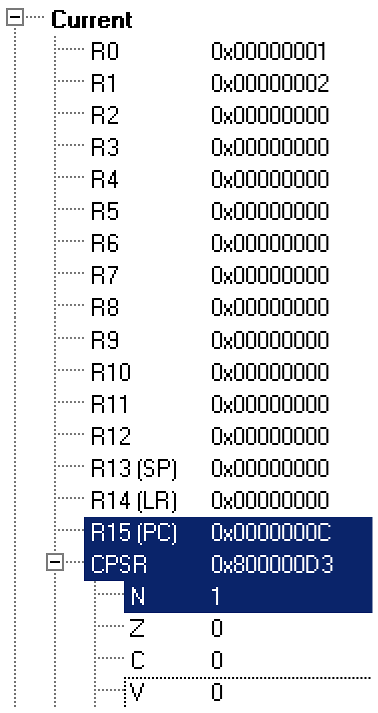
**Execution Result**

Before Execution



```
Current
  R0          0x00000001
  R1          0x00000002
  R2          0x00000000
  R3          0x00000000
  R4          0x00000000
  R5          0x00000000
  R6          0x00000000
  R7          0x00000000
  R8          0x00000000
  R9          0x00000000
  R10         0x00000000
  R11         0x00000000
  R12         0x00000000
  R13 (SP)    0x00000000
  R14 (LR)    0x00000000
  R15 (PC)    0x00000008
  CPSR        0x000000D3
    N         0
    Z         0
    C         0
    V         0
```

After Execution



```
Current
  R0          0x00000001
  R1          0x00000002
  R2          0x00000000
  R3          0x00000000
  R4          0x00000000
  R5          0x00000000
  R6          0x00000000
  R7          0x00000000
  R8          0x00000000
  R9          0x00000000
  R10         0x00000000
  R11         0x00000000
  R12         0x00000000
  R13 (SP)    0x00000000
  R14 (LR)    0x00000000
  R15 (PC)    0x0000000C
  CPSR        0x800000D3
    N         1
    Z         0
    C         0
    V         0
```

## 7.11 THUMB State in ARM Core

Core has two execution states ARM and Thumb; it switches between states using BX Instruction. Advanced Risc Machine (ARM) coined the thumb set for the purpose of RISC processor cores. Thumb is a compressed and 16 bit representation of a subset of the ARM instruction set. Many complex functions require multiple instruction in RISC, in this scenario thumb is used to reduce the memory cost of extra instructions. It also increases the performance from narrow memory and optimizes code density. Like ARM, Thumb also uses load store architecture for data processing, data transfer and control flow instructions. The standard chip that includes the Thumb instruction set is the ARM7TDMI where "T" specifies Thumb.

Differences between ARM and Thumb are presented in the following table 7.11

Table 7.11 THUMB Versus ARM

| Thumb | ARM |
|---|---|
| **Most Thumb instructions are unconditionally executed** | All ARM instructions are conditionally executed |
| **Employs 2 address format** | Employs 3 addresses format |
| **Has explicit shift opcodes** | Implements shifts as operand modifiers |
| **Thumb instruction formats are less regular due to dense coding** | Regular |

### 7.11.1 16 BIT – THUMB INSTRUCTION SET

These instructions operate on the restricted view of the ARM registers. Here the destination register is same as one of the source registers.

**THUMB BIT OR PROGRAM STATUS REGISTERS**

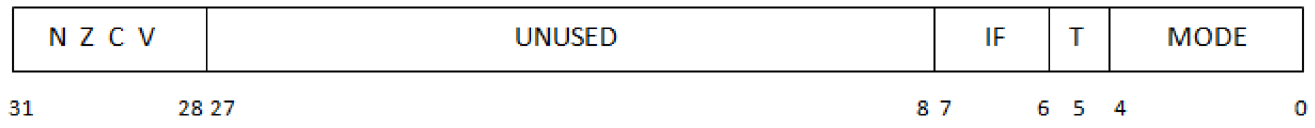


Figure 7.3 CPSR format.

From the above figure 7.3, the ‘T’ bit in the CPSR controls the interpretation of the instruction stream

1. N – Negative or less than
2. Z – Zero
3. C – Carry or Borrow or Extend
4. V – Overflow
5. UNUSED – Reserved
6. MODE – M1, M2, M3, M4 are the Mode bits
7. T – State bit (1: Thumb, 0: ARM)
8. F – FIQ disable
9. I – IRQ disable

### *7.11.2 THUMB ACCESSIBLE REGISTERS*

- All the registers in thumb are not directly accessible. Figure 7.4 can be referred for the better understanding.
- Low register r0 – r7 : Completely accessible
- High register r8 – r12 : accessible only with MOV, ADD, CMP; only CMP sets the condition code flags
- SP (Stack Pointer), LR (Link Register) and PC (Program Counter) : Limitedly accessible
- CPSR: directly accessible
- SPSR: not accessible

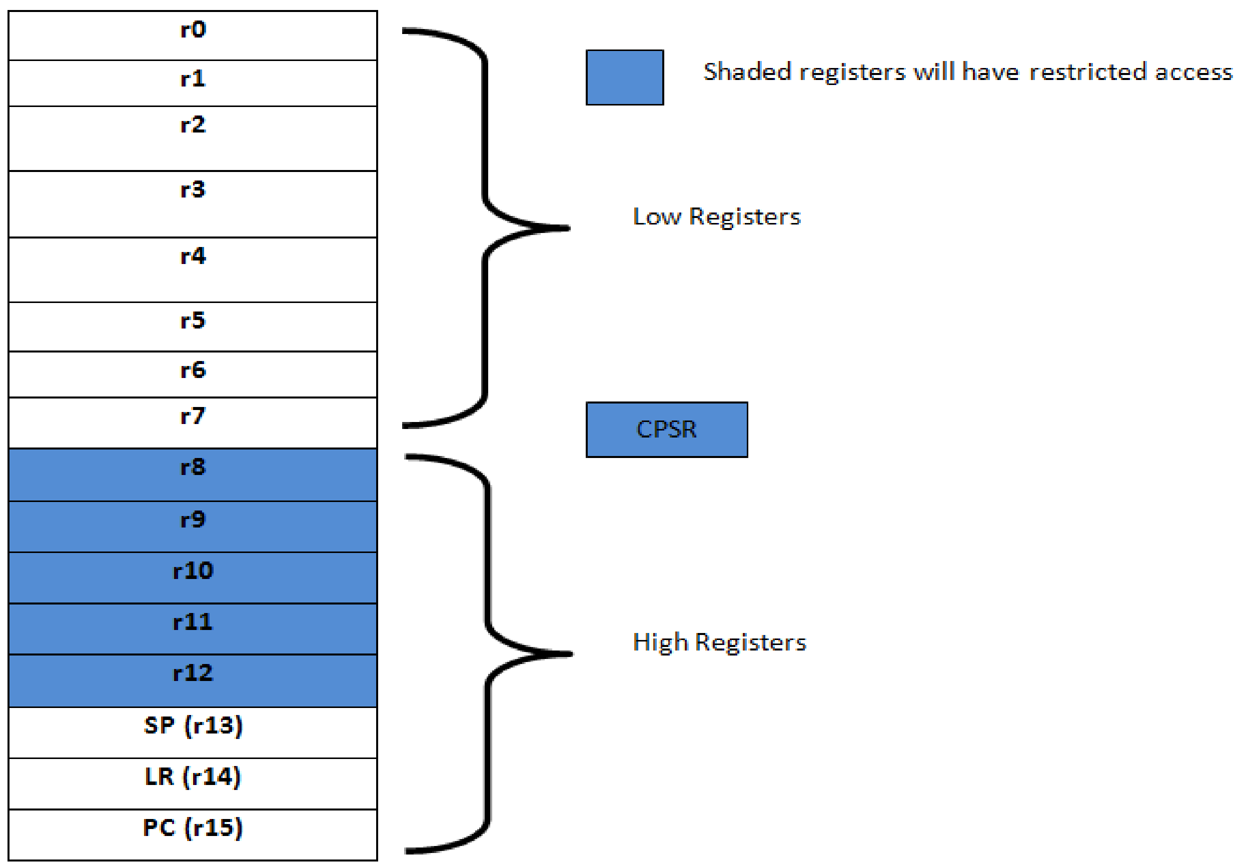


Figure 7.4 Register structure for THUMB

## *7.11.3 THUMB INSTRUCTIONS*

The following table 7.12 summarizes the instruction set followed by THUMB state and comparison with ARM is also presented for easier understanding.

Table 7.12 THUMB Versus ARM instruction set

| MNEMONIC | INSTRUCTION/DESCRIPTION | SYNTAX | ARM-CODE EQUIVALENT |
|---|---|---|---|
| **ADC** | It is used to add the numbers with a carry. | ADC Rd, Rs | ADCS Rd, Rd, Rs |
| **ADD** | It is used to add two numbers without carry | ADD Rd, Rs, Rn | ADDS Rd, Rs, Rn |
| **AND** | It is a logical instruction which is used to compare two numbers | AND Rd, RS | ANDS Rd, Rd, Rs |
| **ASR** | Arithmetic Shift Right | ASR Rd, Rs | MOVS Rd, Rd, ASR Rs |
| **B** | Unconditional Branch | B label | B label |
| **BCC** | Conditional Branch | BCC label | BCC label |
| **BIC** | Bit Clear | BIC Rd, Rs | BICS Rd, Rd, Rs |
| **BL** | Branch and Link | BL Label | BL Label |
| **BX** | Branch and Exchange | BX Hs | BX Hs |
| **CMN** | Compare Negative | CMN Rd, Rs | CMN Rd, Rs |
| **CMP** | Compare | CMP Rd, #offset 8 | CMP Rd, #offset 8 |
| **EOR** | EOR | EOR Rd, Rs | EORS Rd, Rd, Rs |
| **LDMIA** | Load Multiple | LDMIA Rb! , {RList} | LDMIA Rb! , {RList} |
| **LDR** | Load Word | LDR Rd, [PC, #Imm] | LDR Rd, [PC, #Imm] |

| | | | |
|---|---|---|---|
| **LDRB** | Load Byte | LDRB Rd, [Rb, Ro] | LDRB Rd, [Rb, Ro] |
| **LDRH** | Load half word | LDRH Rd, [Rb, #Im] | LDRH Rd, [Rb, #Im] |
| **LSL** | Logical Shift Left | LSL Rd, Rs, #offset 5 | MOVS Rd, Rs, LSL#offset 5 |
| **LDSRB** | Load sign-extended byte | LDRSB Rd, [Rb, Ro] | LDSRB Rd, [Rb, Ro] |
| **LDSRH** | Loan sign-extended half word | LDRSH Rd, [Rb, Ro] | LDRSH Rd, [Rb, Ro] |
| **LSR** | Logical Shift Right | LSR Rd, Rs | MOVS Rd, Rd, LSR Rs |
| **MOV** | Move Register | MOV Rd, #offfset 8 | MOV Rd, #offfset 8 |
| **MUL** | Multiply | MUL Rd, Rs | MULS Rd, Rs, Rd |
| **MVN** | Move NOT Register | MVN Rd, Rs | MVNS Rd, Rs |
| **NEG** | Negate | NEG Rd, Rs | RSBS Rd, Rs, #0 |
| **ORR** | Logical OR | ORR Rd, Rs | ORRS Rd, Rd, Rs |
| **POP** | It is used to POP registers | POP {Rlist] | LDMIA R13!, {Rlist} |
| **PUSH** | It is used to PUSH registers | PUSH {Rlist} | STMDB R13!, {Rlist} |
| **ROR** | Rotate Right | ROR Rd, Rs | MOVS Rd, Rd, ROR Rs |
| **SBC** | It used to subtract two numbers with carry | SBC Rd, Rs | SBCS Rd, Rd, Rs |
| **STMIA** | Store Multiple | STMIA Rb!, {Rlist} | STMIA Rb!, {Rlist} |
| **STR** | Store word | STR Rd, [Rb, Ro] | STR Rd, [Rb, Ro] |
| **STRB** | Store byte | STRB Rd, [Rb, Ro] | STRB Rd, [Rb, Ro] |

| | | | |
|---|---|---|---|
| **STRH** | Store half word | STRH Rd, [Rb, Ro] | STRH Rd, [Rb, Ro] |
| **SWI** | Software Interrupt | SWI Values | SWI Values |
| **SUB** | It is used to subtract two numbers without carry | SUB Rd, Rs, Rn | SUB Rd, Rs, Rn |
| **TST** | Test bits | TST Rd, Rs | TST Rd, Rs |

### *7.11.4 ARM VS. THUMB CODES (A Comparison)*

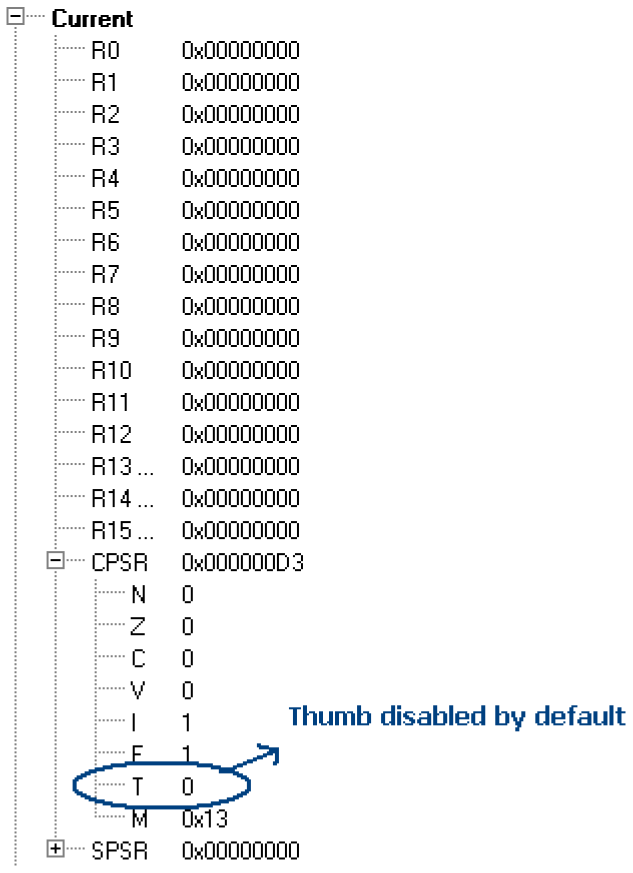
Table 7.13 is presented with examples from both the ARM and THUMB codes and number of bytes occupied by both the states.

Table 7.13 ARM Versus THUMB

| ARM code | Thumb code |
|---|---|
| **ARM Divide** | Thumb Divide |
| **; IN: r0(value), r1 (divisor)** | ; IN: r0(value), r1(divisor) |
| **; OUT: r2(MODulus), r3(DIVide)** | ; OUT: r2(MODulus), r3(DIVide) |
| **MOV r3, #0** | MOV r3, #0 |
| **loop** | loop |
| **SUBS r0, r0, r1** | ADD r3, #1 |
| **ADDGE r3, r3, #1** | SUB r0,r1 |
| **BGE loop** | BGE loop |
| **ADD r2, r0, r1** | SUB r3, #1 |
| | ADD r2, r0, r1 |
| **Total no of bytes = 20 bytes** | Total no of bytes = 12 bytes |

### *7.11.5 How to set THUMB state*

D3 is the default value for the CPSR as shown in the following figure 7.5. So by default one can observe that the Thumb state is disabled. To get it enabled, as already discussed CPSR should be accessed and T bit should be set.



```
Current
  R0       0x00000000
  R1       0x00000000
  R2       0x00000000
  R3       0x00000000
  R4       0x00000000
  R5       0x00000000
  R6       0x00000000
  R7       0x00000000
  R8       0x00000000
  R9       0x00000000
  R10      0x00000000
  R11      0x00000000
  R12      0x00000000
  R13 ...  0x00000000
  R14 ...  0x00000000
  R15 ...  0x00000000
  CPSR     0x000000D3
    N      0
    Z      0
    C      0
    V      0
    I      1
    F      1
    T      0
    M      0x13
  SPSR     0x00000000
```

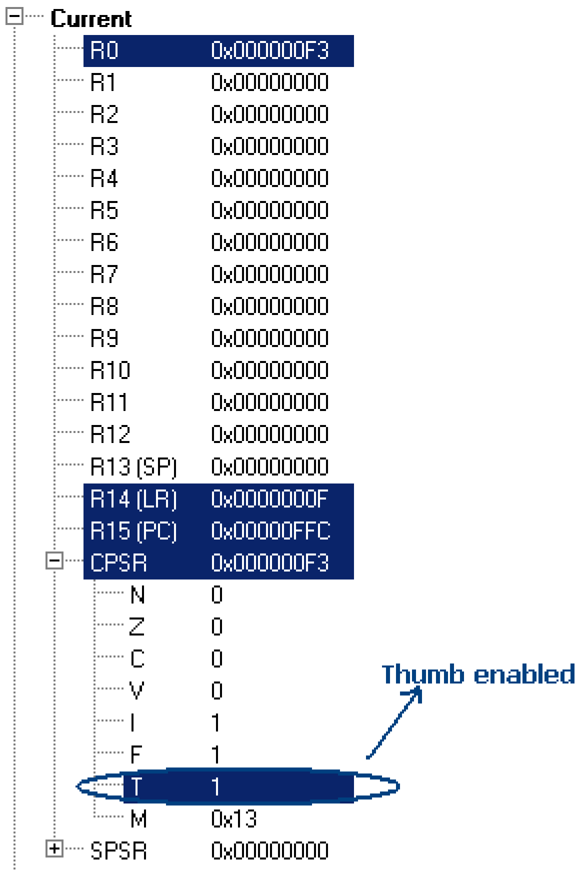
Figure 7.5 Default value for CPSR

Setting the T bit can be done by adding 0x20 to the D3. It will then set the T bit and eventually the THUMB mode will be set. The following piece of code when run can enable the THUMB mode.

## Program to set the THUMB state:

```
 AREA  RESET, CODE, READONLY
 ENTRY
     MRS R0, CPSR
    ; Copying the content of current CPSR to R0
    ADD R0, R0, #0x20
    ; Adding 0x20 in such a way that T bit will be set.
    MSR cpsr_c, r0
    ; Added result should be moved to CPSR with MSR instruction.
    B     .
 END
```

After executing the above written program, one can see the CPSR for the status. It will clearly have T bit set as shown in figure 7.7.



```
Current
  R0          0x000000F3
  R1          0x00000000
  R2          0x00000000
  R3          0x00000000
  R4          0x00000000
  R5          0x00000000
  R6          0x00000000
  R7          0x00000000
  R8          0x00000000
  R9          0x00000000
  R10         0x00000000
  R11         0x00000000
  R12         0x00000000
  R13 (SP)    0x00000000
  R14 (LR)    0x0000000F
  R15 (PC)    0x00000FFC
  CPSR        0x000000F3
    N         0
    Z         0
    C         0
    V         0
    I         1
    F         1
    T         1
    M         0x13
  SPSR        0x00000000
```

Figure 7.6 Setting the THUMB

Programmer can select the state of operation based on the requirement with these set of instructions.

## *Points to remember*

- ARM supports THUMB state and ARM State of instructions.
- ARM has instruction set for
    - Data processing instructions.
    - Conditional instructions
    - Load and store instructions
    - Multiply instructions
    - Software interrupt instructions
    - Branch instructions
    - Barrel shifting operation
- There is no exclusive shift instructions supported in ARM; instead one should use the barrel shifter for the purpose.
- ARM state supports 32 bits instruction where Thumb will support 16 bits instruction.
- ARM has nothing dedicated as stack as 8051 is made with. Instead programmer should allocate it through the instructions.

## *Exercise Programs*

1. Write a program to find greatest of three numbers.

2. Write a program to SWAP two values without using the third variable.

3. Write a simple code to find out if a number is prime or not.

4. Write a program to SWAP upper and lower nibbles of a number.

5. Write a program to disassemble a value. E.g. 54H to 05H and 04H.

6. Write a program to convert ASCII value to BINARY.

7. Write a program to convert HEXADECIMAL to BINARY.