

EMBEDDED SYSTEMS

- Course Code : 18EC62
- CIE Marks :40
- Lecture Hours/Week : 03 + 2 (Tutorial)
- SEE marks :60
- Total Number of Lecture Hours : 50 (10 Hrs / Module) E
- Exam Hours : 03
- CREDITS : 04

MODULE -2

ARM Cortex M3 Instruction Sets and Programming: Assembly basics, Instruction list and description, Thumb and ARM instructions, Special instructions, Useful instructions, CMSIS, Assembly and C language Programming (Text 1: Ch-4, Ch—10.1 to 10.6)

ARM Cortex M3 Instruction Sets and Programming

Contents: Assembly basics, Instruction list and description, Useful instructions, Memory mapping, Bit-band operations and CMSIS, Assembly and C Language Programming.

Assembly Basics

This chapter provides some insight into the instruction set in the Cortex™-M3 and examples for a number of instructions.

Assembler Language: Basic Syntax

In assembler code, the following instruction formatting is commonly used:

Label	opcode	operand1,	operand2,,	; Comments
-------	--------	-----------	-----------	--------	------------

- The **label** is optional. Some of the instructions might have a label in front of them so that the address of the instructions can be determined using the label.
- Then, the **opcode** (the instruction) followed by a number of **operands**. Normally, the first operand is the destination of the operation. The number of operands in an instruction depends on the type of instruction, and the syntax format of the operand can also be different.
- The text after each semicolon (;) is a **comment**. These comments do not affect the program operation, but they can make programs easier to understand.

- For example, immediate data are usually in the form **#number**, as shown here:

MOV R0, #0x12; *Set R0 = 0x12 (hexadecimal)*

MOV R1, #'A'; *Set R1 = ASCII character A*

- **Assembler Language-Use of Suffixes:** in Cortex-M3, the conditional execution suffixes are usually used for branch instructions. However, other instructions can also be used with the conditional execution suffixes if they are inside an IF-THEN instruction block.
- Examples of Suffixes in Instructions

S Update Application Program Status register (APSR) (flags);

for example: **ADDS** R0, R1 ; *this will update APSR*

EQ, NE, LT, GT, and Conditional execution; EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater so on Than, and so forth.

For example: **BEQ**<Label> ; Branch if equal

The difference between thumb instruction format and Unified Assembler Language is as follows:

Table 2.1: Difference between Thumb and UAL

Sl No	Thumb instruction format	Unified Assembler Language (UAL)
1	ADD R0, R1 (2 operands are specified in an instruction). The operation is $R0 = R0 + R1$.	ADD R0, R0, R1 (3 operands are specified in an instruction). The operation is $R0 = R0 + R1$.
2	By default, it updates the flag bits.	Flag bits are updated only by specifying <code>_S</code> in the mnemonic. Ex: ADDS .

- Unified Assembler Language (UAL) was developed to allow selection of 16-bit and 32-bit instructions and to make it easier to port applications between ARM code and Thumb code by using the same syntax for both
- By default, the instruction is narrow (16 bits).
Ex: **MOV** R1, #18 and **MOV.N** R1, #18 are same. (.N means narrow).
- For 32 bit instructions, **MOV.W** R1, #18 is used. (.W means wide).

Assembler Directives

Assembler directives are instructions that direct the assembler to do something. Example:

EQU(Equate) - Constants can be defined using EQU and then it can be used in program code

Ex: `NVIC_IRQ_SET EQU 0xE000E100`
`NVIC_IRQ_ENABLE EQU 0x1`

A number of data definition directives are available for instruction of constants inside assembly code.

Example:

DCI (Define Constant Instruction) - is used to code an instruction if the assembler cannot generate the exact instruction that the user wants and if the user know the binary code for instruction.

Ex: **DCI** 0xBE00 ; Breakpoint (BKPT 0) – 16 bit instruction

DCB (Define Constant Byte) – Byte size constant values such as characters can be used to define using DCB.

Ex: `LDR R0, =hellotxt`
`BL printtext`
`Hellotxt DCB -hello\nll, 0 ; null terminated string`

DCD (Define Constant Data) – Word size constant values to define binary data in assembler code is done through DCD.

Ex: `LDR R3, =my_num`
`LDR R4, [R3];`
`my_num DCD 0x12345678`

Instruction set classification

The instruction set used in ARM Cortex M3 are classified as follows:

1. Moving data within the processor
2. Memory access instructions
3. Arithmetic operations
4. Logic Operations
5. Shift and rotate instructions
6. Sign extend instructions
7. Data reverse instructions
8. Bit field processing instructions
9. Program flow control instructions
10. Memory Barrier instructions
11. IT instruction block
12. Saturation/Conversion instructions
13. Table branch byte and Table branch half word instructions
14. Miscellaneous instructions

Moving data within processor

One of the most basic functions in a processor is transfer of data. In the Cortex-M3, data transfers can be of one of the following types:

1. Moving data between register and register.
2. Moving data between memory and register.
3. Moving data between special register and register.
4. Moving an immediate data value into a register.

Moving data between register and register

- The command to move data between registers is **MOV** (move).

Example

MOV R8, R3; moving data from register R3 to register R8.

- Another instruction can generate the negative value of the original data is **MVN** (move negative).

Example

MVN R8, R3; The MVN instruction takes the value of R3, performs a bitwise logical NOT operation on the value, and places the result into R8.

Table 2.2: Example of MOV and MVN instruction

Example 1	Example 2
Before	Before
R1= 0x00000000	R1 = 0x00000000
R0 = 0x00000004	R0 = 0x00000004
MOV R1, R0	MVN R1, R0
After	After
R1= 0x00000004	R1 = 0xFFFFFFF3
R0 = 0x00000004	R0 = 0x00000004

Moving data between memory and register

- i. **LDR and STR, LDM and STM**
- ii. **PUSH and POP**

LDR and STR

The basic instructions for accessing memory are Load and Store.

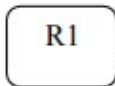
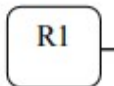

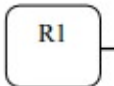
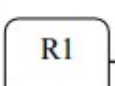
- Load (LDR) transfers data from memory to registers,
- Store (STR) transfers data from registers to memory.
- The transfers can be in different data sizes (byte, half word, word, and double word).

Table 2.2: Commonly Used Memory Access Instructions

Example	Description
LDRB Rd, [Rn, #offset]	Read byte from memory location Rn+offset
LDRH Rd, [Rn, #offset]	Read half word from memory location Rn+offset
LDR Rd, [Rn, #offset]	Read word from memory location Rn+offset
LDRD Rd1,Rd2, [Rn, #offset]	Read double word from memory location Rn+offset
STRB Rd, [Rn, #offset]	Store byte to memory location Rn+offset
STRH Rd, [Rn, #offset]	Store half word to memory location Rn+offset
STR Rd, [Rn, #offset]	Store word to memory location Rn+offset
STRD Rd1,Rd2, [Rn, #offset]	Store double word to memory location Rn+offset

Table 2.3: Examples of Commonly Used Memory Access Instructions

Examples on LDR instruction													
Before Execution													
<div><div>R1</div><div><table><tr><th>Address</th><th>Data</th></tr><tr><td>1000002A</td><td>0xABCDEF54H</td></tr><tr><td>1000002E</td><td>0x12345678H</td></tr><tr><td>10000032</td><td>0x24567893H</td></tr><tr><td>10000036</td><td>0x88564478H</td></tr></table></div></div>				Address	Data	1000002A	0xABCDEF54H	1000002E	0x12345678H	10000032	0x24567893H	10000036	0x88564478H
Address	Data												
1000002A	0xABCDEF54H												
1000002E	0x12345678H												
10000032	0x24567893H												
10000036	0x88564478H												
Example 1	Example 2	Example 3	Example 4										
LDRB R2, [R1,#4]	LDRH R2, [R1, #4]	LDR R2, [R1, #4]	LDRD R2, R3, [R1, #4]										
Operation: R2=mem8[R1+4]	Operation: R2= mem16[R1+4]	Operation: R2= mem32 [R1+4]	Operation: R2, R3= Word from memloc[R1+4], [R1+8] respectively										
After Execution: R2=93H	After Execution: R2=7893H	After Execution: R2=0x24567893H	After Execution: R2=0x24567893H, R3=0x88564478H										

Examples on STR instruction																					
Before Execution																					
 <table border="1" data-bbox="617 220 1006 399"> <thead> <tr> <th>Address</th><th>Data</th></tr> </thead> <tbody> <tr> <td>1000002A</td><td>0xABCDEF54</td></tr> <tr> <td>1000002E</td><td>0x12345678</td></tr> <tr> <td>10000032</td><td>0x24567893</td></tr> <tr> <td>10000036</td><td>0x88564478</td></tr> </tbody> </table>	Address	Data	1000002A	0xABCDEF54	1000002E	0x12345678	10000032	0x24567893	10000036	0x88564478											
Address	Data																				
1000002A	0xABCDEF54																				
1000002E	0x12345678																				
10000032	0x24567893																				
10000036	0x88564478																				
Example 1	Example 2																				
STRB R2, [R1, #4]	STRH R2, [R1, #4]																				
Before Execution: R2=0x324565CDH	Before Execution: R2=0x324565CDH																				
Operation: mem8[R1+4]=R2	Operation: Mem16[R1+4]= R2																				
After Execution:	After Execution:																				
 <table border="1" data-bbox="397 808 787 987"> <thead> <tr> <th>Address</th><th>Data</th></tr> </thead> <tbody> <tr> <td>1000002A</td><td>0xABCDEF54H</td></tr> <tr> <td>1000002E</td><td>0x12345678H</td></tr> <tr> <td>10000032</td><td>0x245678CDH</td></tr> <tr> <td>10000036</td><td>0x88564478H</td></tr> </tbody> </table>	Address	Data	1000002A	0xABCDEF54H	1000002E	0x12345678H	10000032	0x245678CDH	10000036	0x88564478H	 <table border="1" data-bbox="1023 808 1412 987"> <thead> <tr> <th>Address</th><th>Data</th></tr> </thead> <tbody> <tr> <td>1000002A</td><td>0xABCDEF54H</td></tr> <tr> <td>1000002E</td><td>0x12345678H</td></tr> <tr> <td>10000032</td><td>0x245665CDH</td></tr> <tr> <td>10000036</td><td>0x88564478H</td></tr> </tbody> </table>	Address	Data	1000002A	0xABCDEF54H	1000002E	0x12345678H	10000032	0x245665CDH	10000036	0x88564478H
Address	Data																				
1000002A	0xABCDEF54H																				
1000002E	0x12345678H																				
10000032	0x245678CDH																				
10000036	0x88564478H																				
Address	Data																				
1000002A	0xABCDEF54H																				
1000002E	0x12345678H																				
10000032	0x245665CDH																				
10000036	0x88564478H																				
Example 3	Example 4																				
STR R2, [R1, #4]	STRD R2, R3, [R1, #4]																				
Before Execution: R2=0x324565CDH	Before Execution: R2=0x324565CDH, R3=0x88564478H																				
Operation: Mem32[R1+4]= R2	Operation: [R1+4] = R2, [R1+8] = R3																				
After Execution:	After Execution																				
 <table border="1" data-bbox="381 1438 787 1617"> <thead> <tr> <th>Address</th><th>Data</th></tr> </thead> <tbody> <tr> <td>1000002A</td><td>0xABCDEF54H</td></tr> <tr> <td>1000002E</td><td>0x12345678H</td></tr> <tr> <td>10000032</td><td>0x324565CDH</td></tr> <tr> <td>10000036</td><td>0x88564478H</td></tr> </tbody> </table>	Address	Data	1000002A	0xABCDEF54H	1000002E	0x12345678H	10000032	0x324565CDH	10000036	0x88564478H	 <table border="1" data-bbox="1023 1438 1412 1617"> <thead> <tr> <th>Address</th><th>Data</th></tr> </thead> <tbody> <tr> <td>1000002A</td><td>0xABCDEF54H</td></tr> <tr> <td>1000002E</td><td>0xDE285349H</td></tr> <tr> <td>10000032</td><td>0x324565CDH</td></tr> <tr> <td>10000036</td><td>0x88564478H</td></tr> </tbody> </table>	Address	Data	1000002A	0xABCDEF54H	1000002E	0xDE285349H	10000032	0x324565CDH	10000036	0x88564478H
Address	Data																				
1000002A	0xABCDEF54H																				
1000002E	0x12345678H																				
10000032	0x324565CDH																				
10000036	0x88564478H																				
Address	Data																				
1000002A	0xABCDEF54H																				
1000002E	0xDE285349H																				
10000032	0x324565CDH																				
10000036	0x88564478H																				

LDM and STM

- Multiple Load and Store operations can be combined into single instructions called: LDM (Load Multiple) and STM (Store Multiple).
- The exclamation mark (!) in the instruction specifies whether the register Rd should be updated after the instruction is completed

Example **LDMIA** R0!, {R2 - R5}
STMIA R1!, {R2 - R5}

Table 2.4: Examples of Commonly Used Memory Access Instructions

Example	Description
LDMIA Rd!, <Reg list>	Read multiple words from memory location specified by Rd. Address increments(IA) after each transfer(16 bit Thumb instruction)
STMIA Rd!,<Reg list>	Store multiple words to memory location specified by Rd. Address increments(IA) after each transfer(16 bit Thumb instruction)
LDMIA.W Rd(!), <Reg list>	Read multiple words from memory location specified by Rd. Address increments(IA) after each read(.W specifies that it is a 32 bit Thumb2 instruction)
LDMDB.W Rd(!), <Reg list>	Read multiple words from memory location specified by Rd. Address decrements before(DB) each read(.W specifies that it is a 32 bit Thumb2 instruction)
STMIA.W Rd(!), <Reg list>	Write multiple words to memory location specified by Rd. Address increment after each read (.W specifies that it is a 32 bit Thumb2 instruction)
STMDB.W Rd(!), <Reg list>	Write multiple words to memory location specified by Rd. Address decrement before each read (.W specifies that it is a 32 bit Thumb2 instruction)

Preindexing and Postindexing

- ARM processors also support memory accesses with preindexing and postindexing.
- For preindexing, the register holding the memory address is adjusted. The memory transfer then takes place with the updated address.
- Example: LDR.W R0,[R1, #offset]! ; Read memory [R1+offset], with R1 update to R1+offset
; The use of the -! indicates the update of base register R1
- Postindexing memory access instructions carry out the memory transfer using the base address specified by the register and then update the address register afterward.
- The preindexing and postindexing memory access instructions include load and store instructions of various transfer sizes.

Table 2.5: Examples of Preindexed Memory Access Instructions

LDR.W Rd, [Rn, #offset]!	Preindexing load instructions for various sizes (word, byte, half word and double word)
LDRB.W Rd, [Rn, #offset]!	
LDRH.W Rd, [Rn, #offset]!	
LDRD.W Rd1,Rd2, [Rn, #offset]!	
LDRSB.W Rd, [Rn, #offset]!	Preindexing load instructions for various sizes with sign extend (byte, half word)
LDRSH.W Rd, [Rn, #offset]!	
STR.W Rd, [Rn, #offset]!	Preindexing store instructions for various sizes (word, byte, half word and double word)
STRB.W Rd, [Rn, #offset]!	
STRH.W Rd, [Rn, #offset]!	
STRD.W Rd1,Rd2, [Rn, #offset]!	

Examples on LDR instruction – Pre indexed memory access**Before Execution**

Address	Data
10000032	0xABCDEF54H
1000002E	0x12345678H
1000002A	0x24567893H
10000026	0x88564478H

R1 →

Sl No	Instruction	Before Execution	Operation	After Execution:
Ex 1	LDR.W R2, [R1, #8]	R2=0x324565CDH	R2= Mem32[R1+8]	R2 = 0x12345678H
Ex 2	LDRB.W R3, [R1,#4]	R3=0x324565CDH	R3 = Mem8[R1+4]	R3 = 0x00000093H
Ex 3	LDRH.W R4, [R1, #4]	R4 = 0x83502167H	R4 = Mem16[R1+4]	R4 = 0x00007893H
Ex 4	LDRD.W R2, R3, [R1,#8]	R2=0x324565CDH, R3 = 0x83502167H	R2 =Mem32[R1+8], R3 = Mem32[R1+C]	R2 = 0x12345678H, R3 = 0xABCDEF54H
Ex 5	LDRSB.W R2, [R1,#4]	R2=0x324565CDH	R2=signext{Mem8 [R1+4]}	R2 = 0xFFFFF93H
Ex 6	LDRSH.W R2, [R1,#4]	R2=0x324565CDH	R2=signext{Mem16 [R1+4]}	R2 = 0x00007893H

Examples on STR instruction – Pre indexed memory access

Ex 7	STR.W R2, [R1,#8]	R2=0x324565CDH	[R1+8] = [1000002E] = 0x324565CDH
Ex 8	STRB.W R3, [R1,#4]	R3 = 0x83502167H	[R1+4] = [1000002A] = 0x 24567867H
Ex 9	STRH.W R3, [R1,#4]	R3 = 0x83502167H	[R1+4] = [1000002A] = 0x 24562167H
Ex 10	STRD.W R3, R4, [R1,#8]	R3 =0x83502167H, R2=0x324565CDH	[R1+8] = [1000002E] = 0x83502167H, [R1+C] = [10000032] =0x324565CD H

Table 2.6: Examples of Postindexed Memory Access Instructions

LDR.W Rd, [Rn], #offset	Post indexing load instructions for various sizes (word, byte, half word and double word)
LDRB.W Rd, [Rn], #offset	
LDRH.W Rd, [Rn], #offset	
LDRD.W Rd1,Rd2, [Rn], #offset	
LDRSB.W Rd, [Rn], #offset	Post indexing load instructions for various sizes with sign extend (byte, half word)
LDRSH.W Rd, [Rn], #offset	
STR.W Rd, [Rn], #offset	Post indexing store instructions for various sizes (word, byte, half word and double word)
STRB.W Rd, [Rn], #offset	
STRH.W Rd, [Rn], #offset	
STRD.W Rd1,Rd2, [Rn], #offset	

Examples on LDR and STR instruction – Post indexed memory access**Before Execution**

Address	Data
10000098	0xABCDEF54H
10000094	0x12345678H
10000090	0x24567893H
1000008C	0x88564478H

R1 →

Sl No	Instruction	Before Execution	Operation	After Execution:
Ex 1	LDR.W R2, [R1], #8	R2=0x324565CDH	R2= Mem32[R1]	R2 = 0x88564478H R1 points to 0x10000094H
Ex 2	LDRD.W R4,R3, [R1],#4	R4 =0x22334455H, R3 =0x46789120H	R4=Mem32[R1], R3 = Mem32[R1+4]	R4 = 0x88564478H R3 = 0x24567893H R1 points to 0x10000090H
Ex 3	STRH.W R4, [R1], #4	R4 = 0x83502167H	Mem16[R1]=R4	[1000008C] =0x88562167H R1 points to 0x10000090H
Ex 4	STRD.W R3,R2, [R1], #8	R3=0x324565CDH, R2 = 0x83502167H	R3 = Mem32[R1], R2 = Mem32[R1+8]	[1000008C] = 0x324565CDH, [10000094] = =0x83502167H R1 points to 0x10000094H

PUSH and POP Instructions**Syntax**

PUSH<reg> ; Decrement and store

POP <reg> ; Get and increment

Multiple register PUSH and POP operation

PUSH {R0, R4-R7, R9} ;Push R0, R4, R5, R6, R7, R9 into stack memory

POP {R2,R3} ;Pop R2 and R3 from stack Usually a PUSH instruction will have a corresponding POP with the same register list, but this is not always necessary.

For example, a common exception is when POP is used as a function return:

PUSH {R0-R3, LR} ; Save register contents at beginning of subroutine processing.

POP {R0-R3, PC} ; restore registers and return in this case, instead of popping the LR register back and then branching to the address in LR, we POP the address value directly in the program counter.

Moving data between special register and register

MSR and MRS Instructions

MSR and MRS instructions are used to access the special registers in Cortex M3. The special registers are listed as shown in Table 2.7

Table 2.7: Special Register names for MRS and MSR Instructions

Symbol	Description
IPSR	Interrupt status register
EPSR	Execution status register (read as zero)
APSR	Flags from previous operation
IEPSR	A composite of IPSR and EPSR
IAPSR	A composite of IPSR and APSR
EAPSR	A composite of EPSR and APSR
PSR	A composite of APSR, EPSR and IPSR
MSP	Main Stack Pointer
PSP	Process Stack Pointer
PRIMASK	Normal exception mask register
BASEPRI	Normal exception priority mask register
BASEPRI_MAX	Same as normal exception priority mask register, with conditional write (new priority level must be higher than the old level).
FAULTMASK	Fault exception mask register (also disables normal interrupt)
CONTROL	Control register

Syntax: MRS <general purpose reg>,<special register>

MSR < special register >,< general purpose reg >

Example:

MRS R0, PSR ; Read Processor status word into R0.

MSR CONTROL, R1 ; Write value of R1 into control register

Note: Except accessing the APSR, the use MSR or MRS to access other special registers only in privileged mode

Moving an immediate data value into a register

- Moving immediate data into a register is a common thing to do.
- For example, to access a peripheral register, we need to put the address value into a register beforehand.
- For small values (8 bits or less), MOVs (move with status update) can be used.
 - Ex: MOV R0, #0xFFH ; Set R0 = 0xFF (hexadecimal)
 - MOV R1, #'S' ; Set R1 = ASCII character S
 - Ex: MOVS R0, # 0x12H ; Set R0 to 0x12.
- For a larger value (over 8 bits), we need to use a Thumb-2 move instruction.
 - Ex: MOVW.W R0, #0x789AH ; Set R0 lower half to 0x789AH

MOVT (Move Top)

- **Syntax:** MOVT {cond} Rd, #imm16
- Where cond is an optional condition code. Rd is the destination register. imm16 is a 16- bit immediate constant.
- **Operation:** MOVT writes a 16-bit immediate value imm16 to the top halfword, Rd[31:16] of its destination register. The write does not affect lower half word Rd[15:0].
- The MOVW, MOVT instruction pair enables the user to generate any 32-bit constant.
- **Restrictions:** Rd must not be SP and must not be PC.
- **Condition Flags:** This instruction does not change the flags.

Example		
Before Execution	Instruction	After Execution
R3=0x12345678H	MOVT R3, #0xF8CDH	R3= 0x F8CD 5678H

Arithmetic instructions**Table 2.8:** Examples of Arithmetic Instructions

Instruction	Operation	Description
ADD Rd, Rn, Rm	$Rd = Rn + Rm$	ADD operation
ADD Rd, Rd, Rm	$Rd = Rd + Rm$	
ADD Rd, #immd	$Rd = Rd + \#immd$	
ADD Rd, Rn, #immd	$Rd = Rn + \#immd$	
ADC Rd, Rn, Rm	$Rd = Rn + Rm + \text{carry}$	ADD with carry
ADC Rd, Rd, Rm	$Rd = Rd + Rm + \text{carry}$	
ADC Rd, #immd	$Rd = Rd + \#immd + \text{carry}$	
ADDW Rd, Rn, #immd	$Rd = Rn + \#immd$	
SUB Rd, Rn, Rm	$Rd = Rn - Rm$	SUBTRACT
SUB Rd, #immd	$Rd = Rd - \#immd$	
SUB Rd, Rn, #immd	$Rd = Rn - \#immd$	
SBC Rd, Rm	$Rd = Rd - Rm - \text{borrow}$	
SBC.W Rd, Rm, #immd	$Rd = Rm - \#immd - \text{borrow}$	SUBTRACT with borrow (not carry)
SBC.W Rd, Rn, Rm	$Rd = Rn - Rm - \text{borrow}$	
RSB.W Rd, Rn, #immd	$Rd = \#immd - Rn$	
RSB.W Rd, Rn, Rm	$Rd = Rm - Rn$	
MUL Rd, Rm	$Rd = Rd * Rm$	Multiply
MUL.W Rd, Rn, Rm	$Rd = Rn * Rm$	
UDIV Rd, Rn, Rm	$Rd = Rn / Rm$	Unsigned and Signed divide
SDIV Rd, Rn, Rm	$Rd = Rn / Rm$	

Addition Instructions

Ex: 1. ADD R3, R1, R0

Before execution: R1=0x12345678H R0=0xABCDEF45H, R3=0x23657894H

Operation: R3=R1+R0

After execution: R3=0xBE0245BDH

Ex:2. ADD R3,R3,R1

Before execution: R1=0x12345678H, R3=0x23657894H

Operation: R3=R3+R1

After execution: R3=0x3599CF0CH

Ex: 3. ADD R3, #0x34624856H

Before execution: R3=0x12345678H

Operation: R3=R3+0x34624856H

After execution: R3= 0x46969ECEH

Ex:4. ADD R3, R1, # 0xABCDEF45 H

Before execution: R1=0x12345678H,

Operation: R3=R1+0xABCDEF45H

After execution: R3=0xBE0245BDH

Ex5. ADC R3, R1, R0

Before execution: R1=0x12345678H, R0=0xABCDEF45H, R3=0x23657894H , carry=1

Operation: R3=R1+R0+carry

After execution: R3=0xBE0245BEH

Ex6. ADC R3, R3, R0

Before execution: R0=0xABCDEF45H, R3=0x23657894H , carry=1

Operation: R3=R3+R0+carry

After execution: R3=0xCF3367DAH

Ex7. ADC R3, #0xABCDEF45

Before execution: R3=0x23657894 H, carry=1

Operation: R3=R3+#0xABCDEF45H+carry

After execution: R3=0xCF3367DAH

Subtraction Instructions

Table 2.8: Examples of Subtraction Instructions

Example 1	Example 2	Example 3
SUB R2, R1, R0	SUB R2, #0x00000003	SUB R2, R1, #0x00000003
<i>Before Execution</i> R2=0x00000000H R1=0x00000004H R0=0x00000003H	<i>Before Execution</i> R2 = 0x00000007H	<i>Before Execution</i> R1 = 0x0000000AH
<i>Operation:</i> R2 = R1 – R0	<i>Operation:</i> R2 = R2 – 0x00000003H	<i>Operation:</i> R2 = R1 – 0x00000003H
<i>After Execution</i> R2 = 0x00000001H R1 = 0x00000004H R0 = 0x00000003H	<i>After Execution</i> R2 = 0x00000004H	<i>After Execution</i> R2 = 0x00000007H
Example 4	Example 5	Example 6
SBC R1, R0	SBC.W R2, R1, R0	RSB.W R2, R1, #45H
<i>Before Execution</i> B=1, C=0, R0=9AH, R1=0x0000002BH	<i>Before Execution</i> B=1, C=0, R0=9AH, R1=0x0000002BH	<i>Before Execution</i> R1=0x00000002H
<i>Operation:</i> R1= R1 – R0 - borrow	<i>Operation:</i> R2 = R1 – R0 - borrow	<i>Operation:</i> R2= #45H– R1
<i>After Execution</i> R2=0xFFFFFFFF90H	<i>After Execution</i> R2=0xFFFFFFFF90H	<i>After Execution</i> R2=0x00000043H
Example 7		
RSB.W R2, R1, R0		
<i>Before Execution</i> R1=0x00000022H, R0=0x00000049H		
<i>Operation:</i> R2 = R0 – R1		
<i>After Execution</i> R2=0x00000027H		

Multiplication Instructions

Table 2.9: 32 bit multiply instructions

Instruction	Operation
SMULL RdLo, RdHi, Rn, Rm Operation: {RdHi, RdLo} = $R_n * R_m$	32 bit multiply instructions for signed values
SMLAL RdLo, RdHi, Rn, Rm Operation: {RdHi, RdLo} += $R_n * R_m$	
UMULL RdLo, RdHi, Rn, Rm Operation: {RdHi, RdLo} = $R_n * R_m$	32 bit multiply instructions for unsigned values
UMLAL RdLo, RdHi, Rn, Rm Operation: {RdHi, RdLo} += $R_n * R_m$	
MLA R10, R1, R2, R5 Operation: $R_{10} = (R_1 * R_2) + R_5$	Multiply with Accumulator
MLS R4, R5, R6, R7 Operation: $R_4 = R_7 - (R_5 * R_6)$	Multiply with Subtract

Table 2.10: Example to illustrate the difference between SMULL and UMULL instructions

Ex: -3 x 2 (smull)	Ex: 3 x 2 (umull)
MOV R2, # - 0x0003	MOV R2, # 0x0003
MOV R3, # 0x0002	MOV R3, # 0x0002
Before Execution:	Before Execution:
R2 = 0xFFFFFFFDDH (-3), R3 = 0x00000002H (2)	R2 = 0x00000003H (3), R3 = 0x00000002H (2)
After Execution: R4 = 0xFFFFFFFFA (-6), R5 = 0xFFFFFFF (F)	After Execution: R4 = 0x00000006H (6), R5 = 0x00000000H (0)

Division Instructions

The mnemonic for signed and unsigned divide instructions is as follows: SDIV.W, UDIV.W.

The result is $Rd = Rn/Rm$

Example 1: Program to divide two 32-bit data AREA
MULTIPLICATION, CODE, READONLY EXPORT

main

NUM1 DCD &BBBBBBBB

NUM2 DCD &22222222

_main

LDR R0, NUM1

; Read the first data

LDR R1, NUM2

; Read the second data

UDIV /SDIV R3, R0, R1

; Divide R0 by R1, R3=5 (quotient)

END

Logical instructions

Table 2.11: Logic Operation Instructions

Instruction	Operation	Description
AND Rd, Rn	$Rd = Rd \& Rn$	Bitwise AND
AND.W Rd, Rn, #immd	$Rd = Rn \& \#immd$	
AND.W Rd, Rn, Rm	$Rd = Rn \& Rm$	
ORR Rd, Rn	$Rd = Rd Rn$	Bitwise OR
ORR.W Rd, Rn, #immd	$Rd = Rn \#immd$	
ORR.W Rd, Rn, Rm	$Rd = Rn Rm$	
BIC Rd, Rn	$Rd = Rd \& (\sim Rn)$	Bit clear
BIC.W Rd, Rn, #immd	$Rd = Rn \& (\sim \#immd)$	
BIC.W Rd, Rn, Rm	$Rd = Rn \& (\sim Rm)$	
ORN.W Rd, Rn, #immd	$Rd = Rn (\sim \#immd)$	Bitwise OR NOT
ORN.W Rd, Rn, Rm	$Rd = Rn (\sim Rm)$	
EOR Rd, Rn	$Rd = Rd \wedge Rn$	Bitwise XOR
EOR.W Rd, Rn, #immd	$Rd = Rn \wedge \#immd$	
EOR.W Rd, Rn, Rm	$Rd = Rn \wedge Rm$	

Table 2.12: Examples on Logic Operation Instructions

SI No.	Examples		
	Instruction	Before Execution	After Execution
1.	AND R2, R3 <i>Operation:</i> $R2 = R2 \& R3$	$R2 = 0xE2352356H$ $R3 = 0x79254361H$	$R2 = 0x60250340H$
2.	ORR R1, R4, R3 <i>Operation:</i> $R1 = R4 R3$	$R4 = 0xE2352356H$ $R3 = 0x79254361H$	$R1 = 0xFB356377H$
3.	BIC.W R6, R0, #2 <i>Operation:</i> $R6 = R0 \& (\sim\#2)$	$R0 = 0x79254361H$	$R6 = 0x79254361H$
4.	ORN.W R4, R1, R0 <i>Operation:</i> $R4 = R1 \sim R0$	$R1 = 0xE2352356H$ $R0 = 0x79254361H$	$R4 = 0xE6FFBFDEH$
5.	EOR.W R2, R3, R4 <i>Operation:</i> $R2 = R3 \wedge R4$	$R3 = 0xE2352356H$ $R4 = 0x79254361H$	$R2 = 0x9B106037H$

Shift and Rotate Instructions

Table 2.12: Shift and Rotate Instructions

Instruction	Operation	Description
ASR Rd, Rn, #immd	$Rd = Rn \gg \text{immd}$	Arithmetic shift right
ASR Rd, Rn	$Rd = Rd \gg Rn$	
ASR.W Rd, Rn, Rm	$Rd = Rn \gg Rm$	
LSL Rd, Rn, #immd	$Rd = Rn \ll \text{immd}$	Logical shift left
LSL Rd, Rn	$Rd = Rd \ll Rn$	
LSL.W Rd, Rn, Rm	$Rd = Rn \ll Rm$	
LSR Rd, Rn, #immd	$Rd = Rn \gg \text{immd}$	Logical shift right
LSR Rd, Rn	$Rd = Rd \gg Rn$	
LSR.W Rd, Rn, Rm	$Rd = Rn \gg Rm$	
ROR Rd, Rn	$Rd \text{ rot by } Rn$	Rotate right
ROR.W Rd, Rn, #immd	$Rd = Rn \text{ rot by immd}$	
ROR.W Rd, Rn, Rm	$Rd = Rn \text{ rot by } Rm$	
RRX, W Rd, Rn	$\{C, Rd\} = \{Rn, C\}$	Rotate right extended

Logical Shift Left (LSL)



Logical Shift Right (LSR)



Rotate Right (ROR)



Arithmetic Shift Right (ASR)



Rotate Right eXtended (RRX)



Table 2.13: Examples on Shift and Rotate Instructions

Sl No.	Examples		
	Instruction	Before Execution	After Execution
1.	ASR R2, R3, #2 Operation: R2 = R3 >> 2	1. R3=0xE2352356H 2. R3 = 0x79254361H	R2 = 0xF88d48D5H R2 = 0x1E4950D8H
2.	LSL R1,#3 Operation: R1 = R1 <<3	R1 = 0xABCD1234H	R1 = 0x5E6891A0H
3.	LSR R6, #1 Operation: R6 = R6 >>1	R6 = 0xA2BC3567H	R6 = 0x515E1AB3H
4.	ROR R4, #5 Operation: R4 = R4 rotate right by 5	R4 = 0x689243ACH	R4 = 0x6344921DH
5.	RRX R4, #3 Operation: R4 = R4 rotate with carry by 3	R4 = 0x689243ACH Carry = 0	R4 = 0x0D124875H Carry = 1

Note: Why Is There Rotate Right But No Rotate Left?

The rotate left operation can be replaced by a rotate right operation with a different rotate offset.

For example, a rotate left by 4-bit operation can be written as a rotate right by 28-bit instruction, which gives the same result and takes the same amount of time to execute.

Sign Extension instructions

Table 2.14: Sign Extension Instructions with example

Instruction	Operation	Description
SXTB <Rd>, <Rm>	Rd = signext (Rm[7:0])	Sign extend byte data into word
SXTH <Rd>, <Rm>	Rd = signext (Rm[15:0])	Sign extend half word data into word
UXTB <Rd>, <Rn>	Rd = unsignext(Rn[7:0])	Unsign extend byte data into word
UXTH <Rd>, <Rn>	Rd = unsignext(Rn[15:0])	Unsign extend half word data into word
Example: Before execution: R0 = 0x55AA8765H, R1 = 0x9A2378ABH		
1. Instruction: SXTB R1(d), R0(s)	2. Instruction: SXTH R0(d), R1(s)	
After execution: R0 = 0x55AA8765H, R1 = 0x00000065H	After execution: R0 = 0x55AA8765H, R1 = 0xFFFF8765H	
3. Instruction: UXTB R1(d), R0(s)	4. Instruction: UXTH R0(d), R1(s)	
After execution: R0 = 0x55AA8765H, R1 = 0x00000065H	After execution: R0 = 0x55AA8765H, R1 = 0x00008765H	

Data Reverse order Instructions

Table 2.14: Data Reverse order Instructions with example

Instructions	Operation	Description
REV Rd, Rn	Rd = rev(Rn)	Reverse bytes in word
REV16 Rd, Rn	Rd = rev16(Rn)	Reverse bytes in each half word
REVSH Rd, Rn	Rd = revsh(Rn)	Reverse bytes in bottom half word and sign extend the result
Example: Before Execution: R0 = 0x1234789AH, R1 = 0x9A23C8ABH		
Instruction: REV R0, R1	Instruction: REV16 R0, R1	Instruction: REVSH R0, R1
After Execution: R1 = 0x9A23C8ABH R0 = 0xABC8239AH,	After Execution: R1 = 0x9A23C8ABH R0 = 0x 239AABC8H,	After Execution: R1 = 0x9A23C8ABH R0 = 0x FFFFABC8H,

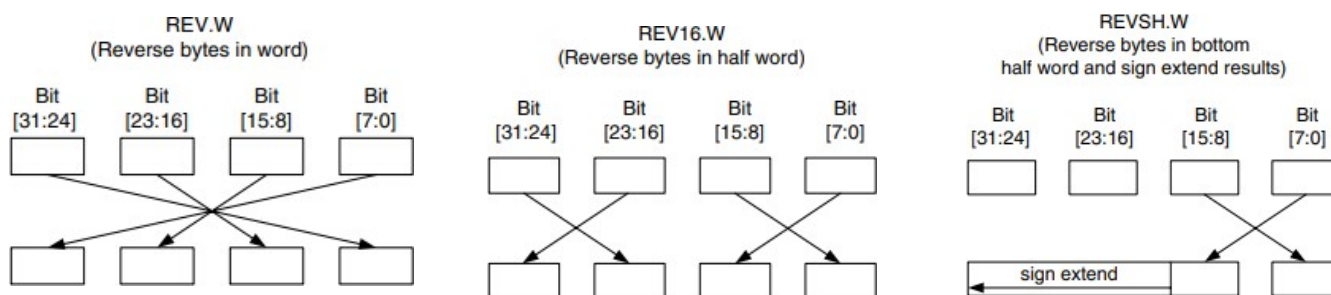



Figure: 1.2: Operation of Reverse instructions.

Bit field manipulation and extraction instructions

Table 2.15: Bit field manipulation and extraction instructions with an example

Sl No.	Instruction	Operation	Description
1	BFC Rd, #<LSB>, #width	Clear the bits starting from LSB bit till the width towards MSB, rest of the bits are unaffected.	Bit Field Clear
<p>Example: BFC R0, #4, #10 Before Execution: R0 = 0x12345678H R0 = 0001 0010 0011 0100 0101 0110 0111 1000 (clear bit 4 to bit 13) After Execution: R0 = 0x12344008H R0 = 0001 0010 0011 0100 0100 0000 0000 1000</p>			
2	BFI Rd, Rn, #<LSB>, #width	Insert the source register bits from 0 th bit till the width towards MSB into the destination register bits from LSB till width towards MSB. Rest of the bits are unaffected.	Bit Field Insert
<p>Example: BFI R1, R2, #8, #12 Before Execution: R1 = 0x34221DBDH, R2 = 0xDCBC4533H R2 = 1101 1100 1011 1100 0100 0101 0011 0011 R1 = 0011 0100 0010 0010 0001 1101 1011 1101 (Replace) After Execution: R1 = 0011 0100 0010 0101 0011 0011 1011 1101 R1 = 0x342533BDH, R2 = 0xDCBC4533H</p>			
3	RBIT Rd, Rn	Reverse every bit of source register and store it in destination register	Reverse Bit
<p>Example: RBIT R1, R2 Before Execution: R1 = 0x12345678H, R2 = 0x10018008H</p>  <p>After Execution: R1 = 0x10018008H, R2 = 0x10018008H</p>			

4	CLZ.W Rd, Rn	Counts the leading zeroes in source register and place the count value in destination register	Count leading zeros
Example: CLZ.W R3, R4 <i>Before Execution:</i> R3 = 0x12345678H, R4 = 0x 22556688H <i>After execution:</i> R3 = 0x00000002H			
5	UBFX.W <Rd>, <Rn>, <#LSB>. <#width>	Extracts a bit field from a register starting from any location (specified by LSB) with any width (specified by #width), zero extends it, puts in destination register	Unsigned Bit Field Extraction
Example: LDR R0, =0x5678ABCDH UBFX.W R1, R0, #4, #8 R1 = 0x000000BCH			
6	SBFX.W <Rd>, <Rn>, <#LSB>. <#width>	SBFX extracts a bit field, but its sign extends it before putting it in a destination register.	Signed Bit Field Extraction
Example: LDR R0, =0x5678ABCDH SBFX.W R1, R0, #4, #8 R1 = 0xFFFFFBCBs			

Call and Unconditional Branch

The most basic branch instructions are as follows:

- B label ; Branch to a labeled address
- BX reg ; Branch to an address specified by a register

In BX instructions, the LSB of the value contained in the register determines the next state (Thumb/ ARM) of the processor. In the Cortex-M3, because it is always in Thumb state, this bit should be set to 1. If it is zero, the program will cause a usage fault exception because it is trying to switch the processor into ARM state.

To call a function, the branch and link instructions should be used.

BL label ; Branch to a labeled address and save return address in LR

BLX reg ; Branch to an address specified by a register and save return address in LR.

With these instructions, the return address will be stored in the link register (LR) and the function can be terminated using BX LR, which causes program control to return to the calling process. However, when using BLX, make sure that the LSB of the register is 1. Otherwise the processor will produce a fault exception because it is an attempt to switch to the ARM state. You can also carry out a branch operation using MOV instructions and LDR instructions. For example,

- MOV R15, R0 ; Branch to an address inside R0
- LDR R15, [R0] ; Branch to an address in memory location specified by R0
- POP {R15} ; Do a stack pop operation, and change the program counter value to the result value.

When using these methods to carry out branches, you also need to make sure that the LSB of the new program counter value is 0x1. Otherwise, a usage fault exception will be generated because it will try to switch the processor to ARM mode, which is not allowed in the Cortex-M3 redundancy.

The BL instruction will destroy the current content of LR. So, the program code needs the LR later, we should save LR before use BL. The common method is to push the LR to stack in the beginning of subroutine. For example,

```
main
...
    BL functionA
...functionA
    PUSH {LR} ; Save LR content to stack ...
    BL functionB ...
    POP {PC} ; Use stacked LR content to return to main
functionB PUSH {LR}
...
    POP {PC} ; Use stacked LR content to return to functionA
```

In addition, if the subroutine you call is a C function, you might also need to save the contents in R0–R3 and R12 if these values will be needed at a later stage.

: Decisions and Conditional Branches

Most conditional branches in ARM processors use flags in the APSR to determine whether a branch should be carried out. In the APSR, there are five flag bits; four of them are used for branch decisions (see Table 4.25). There is another flag bit at bit[27], called the *Q flag*. It is for saturation math operations and is not used for conditional branches.

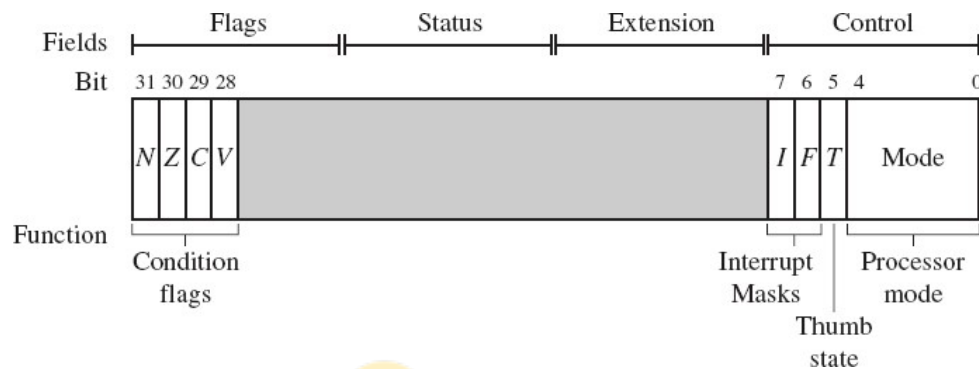


Table 2.16: Flag Bits in APSR that Can Be Used for Conditional Branches

Flag	PSR Bit	Description
N	31	Negative flag(last operation result is a negative value)
Z	30	Zero (last operation result is a zero value)
C	29	Carry flag (last operation returns a carry out or borrow)
V	28	Overflow (last operation results in an overflow)

- Z (Zero) flag: This flag is set when the result of an instruction has a zero value or when a comparison of two data returns an equal result.
- N (Negative) flag: This flag is set when the result of an instruction has a negative value (bit 31 is 1).
- C (Carry) flag: This flag is for unsigned data processing—for example, in add (ADD) it is set when an overflow occurs; in subtract (SUB) it is set when a borrow did not occur (borrow is the invert of carry).
- V (Overflow) flag: This flag is for signed data processing; for example, in an add (ADD), when two positive values added together produce a negative value, or when two negative values added together produce a positive value.
- With combinations of the four flags (N, Z, C, and V), 15 branch conditions are defined (see Table 2.17).
- Using these conditions, branch instructions can be written,
- For example, BEQ label ; Branch to address 'label' if Z flag is set.
- We can also use the Thumb-2 version if your branch target is further away.
- For example, BEQ.W label; Branch to address 'label' if Z flag is set.

Table 2.17: Conditions for Branches or Other Conditional Operations

Symbol	Condition	Flag
EQ	Equal	Z set
NE	Not equal	Z clear
CS/HS	Carry set/unsigned higher or same	C set
CC/LO	Carry clear/unsigned lower	C clear
MI	Minus/negative	N set
PL	Plus/positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N set and V set, or N clear and V clear ($N == V$)
LT	Signed less than	N set and V clear, or N clear and V set ($N != V$)
GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear ($Z == 0, N == V$)
LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set ($Z == 1$ or $N != V$)
AL	Always (unconditional)	—

The defined branch conditions can also be used in IF-THEN-ELSE structures. For example,

```

CMP R0, R1 ; Compare R0 and R1
ITTEE GT ; If R0 > R1 Then if true, first 2 statements execute, if false, other 2 statements execute
MOVGT R2, R0 ; R2 = R0
MOVGT R3, R1 ; R3 = R1
MOVLE R2, R0 ; Else R2 = R1
MOVLE R3, R1 ; R3 = R0

```

APSR flags can be affected by the following:

- Most of the 16-bit ALU instructions
- 32-bit (Thumb-2) ALU instructions with the *S* suffix; for example, ADDS.W
- Compare (e.g., CMP) and Test (e.g., TST, TEQ)
- Write to APSR/xPSR directly

Most of the 16-bit Thumb arithmetic instructions affect the *N*, *Z*, *C*, and *V* flags. With 32-bit Thumb-2 instructions, the ALU operation can either change flags or not change flags.

For example ADDS.W R0, R1, R2 ; This 32-bit Thumb instruction updates flag

ADD.W R0, R1, R2 ; This 32-bit Thumb instruction does not update flag

The compare (CMP) instruction subtracts two values and updates the flags (just like SUBS), but the result is not stored in any registers. CMP can have the following formats:

CMP R0, R1 ; Calculate $R0 - R1$ and update flag

CMP R0, #0x12 ; Calculate $R0 - 0x12$ and update flag

A similar instruction is the CMN (compare negative). It compares one value to the negative (two's

complement) of a second value; the flags are updated, but the result is not stored in any registers:

```
CMN R0, R1 ; Calculate R0 – (-R1) and update flag
CMN R0, #0x12 ; Calculate R0 – (-0x12) and update flag
```

The TST (test) instruction is more like the AND instruction. It ANDs two values and updates the flags. However, the result is not stored in any register. Similarly to CMP, it has two input formats:

```
TST R0, R1 ; Calculate R0 AND R1 and update flag
TST R0, #0x12 ; Calculate R0 AND 0x12 and update flag
```

Combined Compare and Conditional Branch

With ARM architecture v7-M, two new instructions are provided on the Cortex-M3 to supply a simple compare with zero and conditional branch operations. These are CBZ (compare and branch if zero) and CBNZ (compare and branch if nonzero).

The compare and branch instructions only support forward branches.

```
For example, i = 5; while (i != 0) { func1(); ; call a function
i--;
}
```

This can be compiled into the following:

```
...
MOV R0, #5 ; Set loop counter
loop1 CBZ R0, loop1exit ; if loop counter = 0 then exit the loop
BL func1 ; call a function
SUB R0, #1 ; loop counter decrement
B loop1 ; next loop
loop1exit...
```

The APSR value is not affected by the CBZ and CBNZ instructions.

Conditional Execution Using IT Instructions

The IT (IF-THEN) block is very useful for handling small conditional code.

It avoids branch penalties because there is no change to program flow.

It can provide a maximum of four conditionally executed instructions.

In IT instruction blocks, the first line must be the IT instruction, detailing the choice of execution, followed by the condition it checks.

The first statement after the IT command must be TRUE-THEN- EXECUTE, which is always written as *ITxyz*, where *T* means THEN and *E* means ELSE. The second through fourth statements can be either THEN (true) or ELSE (false):

```
IT<x><y><z><cond> ; IT instruction (<x>, <y>, <z> can be T or E)
instr1<cond><operands> ; 1st instruction (<cond> must be same as IT)
instr2<cond or not cond><operands> ; 2nd instruction (can be <cond> or <!cond>)
instr3<cond or not cond><operands> ; 3rd instruction (can be <cond> or <!cond>)
instr4<cond or not cond><operands> ; 4th instruction (can be <cond> or <!cond>)
```

If a statement is to be executed when *<cond>* is false, the suffix for the instruction must be the opposite of the condition. For example, the opposite of EQ is NE, the opposite of GT is LE, and so on. The following code shows an example of a simple conditional execution:

```
if (R1<R2) then
    R2=R2-R1
    R2=R2/2
else
    R1=R1-R2
    R1=R1/2
```

In assembly,

```
CMP    R1, R2        ; If R1 < R2 (less then)
ITTEE  LT            ; then execute instruction 1 and 2 (indicated by T)
                        ; else execute instruction 3 and 4 (indicated by E)
SUBLT.W R2,R1        ; 1st instruction LSRLT.W R2,#1 ; 2nd instruction
SUBGE.W R1,R2        ; 3rd instruction (notice the GE is opposite of LT)
LSRGE.W R1,#1        ; 4th instruction
```

We can have fewer than four conditionally executed instructions. The minimum is 1. we need to make sure the number of *T* and *E* occurrences in the IT instruction matches the number of conditionally executed instructions after the IT.

If an exception occurs during the IT instruction block, the execution status of the block will be stored in the stacked PSR (in the IT/Interrupt-Continuable Instruction [ICI] bit field). So, when the exception handler completes and the IT block resumes, the rest of the instructions in the block can continue the execution correctly. In the case of using multicycle instructions (for example, multiple load and store) inside an IT block, if an exception takes place during the execution, the whole instruction is abandoned and restarted after the interrupt process is completed.

Instruction Barrier and Memory Barrier Instructions

The Cortex-M3 supports a number of barrier instructions. These instructions are needed as memory systems get more and more complex. In some cases, if memory barrier instructions are not used, race conditions could occur.

For example, if the memory map can be switched by a hardware register, after writing to the memory switching register you should use the DSB instruction. Otherwise, if the write to the memory switching register is buffered and takes a few cycles to complete, and the next instruction accesses the switched memory region immediately, the access could be using the old memory map. In some cases, this might result in an invalid access if the memory switching and memory access happen at the same time. Using DSB in this case will make sure that the write to the memory map switching register is completed before a new instruction is executed.

The following are the three barrier instructions in the Cortex-M3:

- DMB
- DSB
- ISB

The DSB and ISB instructions can be important for self-modifying code. For example, if a program changes its own program code, the next executed instruction should be based on the updated program. However, since the processor is pipelined, the modified instruction location might have already been fetched. Using DSB and then ISB can ensure that the modified program code is fetched again.

Architecturally, the ISB instruction should be used after updating the value of the CONTROL register. In the Cortex-M3 processor, this is not strictly required. But if you want to make sure your application is portable, you should ensure an ISB instruction is used after updating to CONTROL register.

DMB is very useful for multi-processor systems. For example, tasks running on separate processors might use shared memory to communicate with each other. In these environments, the order of memory accesses to the shared memory can be very important. DMB instructions can be inserted between accesses to the shared memory to ensure that the memory access sequence is exactly the same as expected.

Table 2.18: Barrier Instructions

Instruction	Description
DMB	Data memory barrier; ensures that all memory accesses are completed before new memory access is committed
DSB	Data synchronization barrier; ensures that all memory accesses are completed before next instruction is executed
ISB	Instruction synchronization barrier; flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

Saturation Operations

The Cortex-M3 supports two instructions that provide signed and unsigned saturation operations: SSAT and USAT (for signed data type and unsigned data type, respectively). Saturation is commonly used in signal processing—for example, in signal amplification. When an input signal is amplified, there is a chance that the output will be larger than the allowed output range. If the value is adjusted simply by removing the unused MSB, an overflowed result will cause the signal waveform to be completely deformed.

The saturation operation does not prevent the distortion of the signal, but at least the amount of distortion is greatly reduced in the signal waveform.

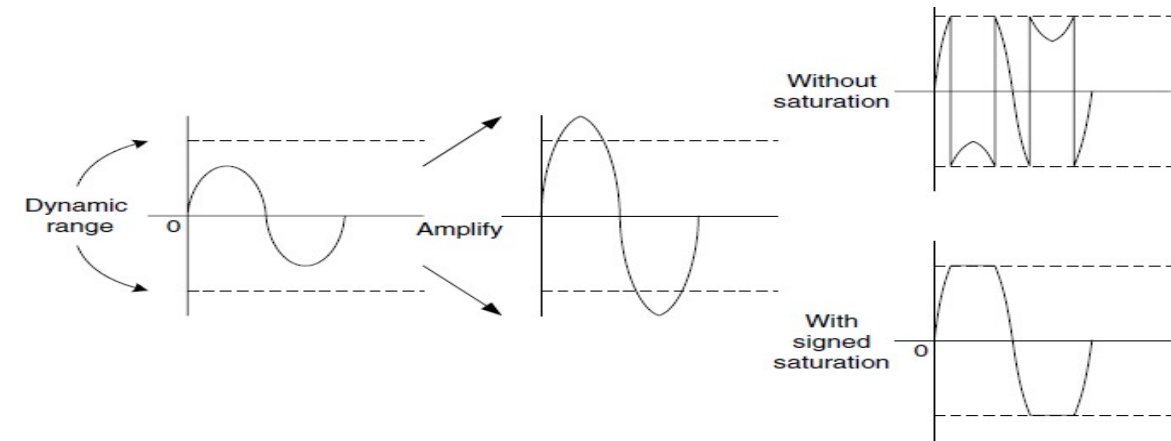


Figure 2.4 Signed Saturation Operation

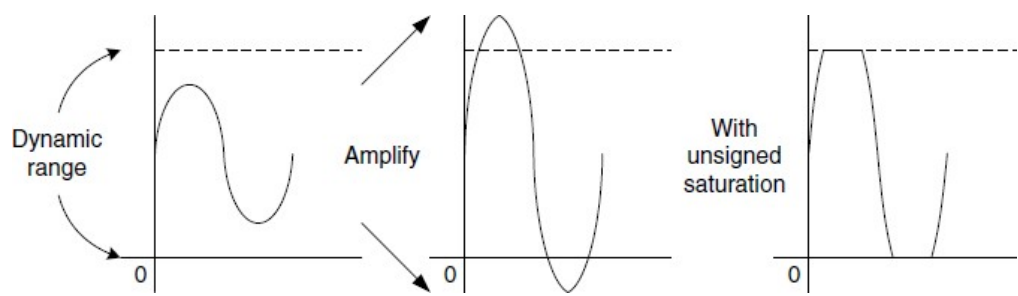


Figure 2.5 Unsigned Saturation Operation

Table 2.19: Saturation Instructions

Instruction	Description
SSAT.W <Rd>, #<immed>, <Rn>, {,<shift>}	Saturation for signed value
USAT.W <Rd>, #<immed>, <Rn>, {,<shift>}	Saturation for a signed value into an unsigned value
<i>Rn: Input value</i> <i>Shift: Shift operation for input value before saturation; optional, can be #LSL N or #ASR N</i> <i>Immed: Bit position where the saturation is carried out</i> <i>Rd: Destination register</i>	

Besides the destination register, the Q-bit in the APSR can also be affected by the result. The Q flag is set if saturation takes place in the operation, and it can be cleared by writing to the APSR.

For example, if a 32-bit signed value is to be saturated into a 16-bit signed value, the following instruction can be used:

```
SSAT.W R1, #16, R0
```

Similarly, if a 32-bit unsigned value is to saturate into a 16-bit unsigned value, the following instruction can be used: USAT.W R1, #16, R0

Table 2.20: Examples of Signed Saturation Results Table 2.21: Examples of Unsigned Saturation Results

Input (R0)	Output (R1)	Q Bit
0x00020000	0x00007FFF	Set
0x00008000	0x00007FFF	Set
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0xFFFF8000	Unchanged
0xFFFF7FFF	0xFFFF8000	Set
0xFFFE0000	0xFFFF8000	Set

Input (R0)	Output (R1)	Q Bit
0x00020000	0x0000FFFF	Set
0x00008000	0x00008000	Unchanged
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0x00000000	Set
0xFFFF8001	0x00000000	Set
0xFFFFFFFF	0x00000000	Set

Memory Maps

The Cortex-M3 processor has a fixed memory map. This makes it easier to port software from one Cortex- M3 product to another. For example, components described in previous sections, such as Nested Vectored Interrupt Controller (NVIC) and Memory Protection Unit (MPU), have the same memory locations in all Cortex-M3 products. Nevertheless, the memory map definition allows great flexibility so that manufacturers can differentiate their Cortex-M3-based product from others.

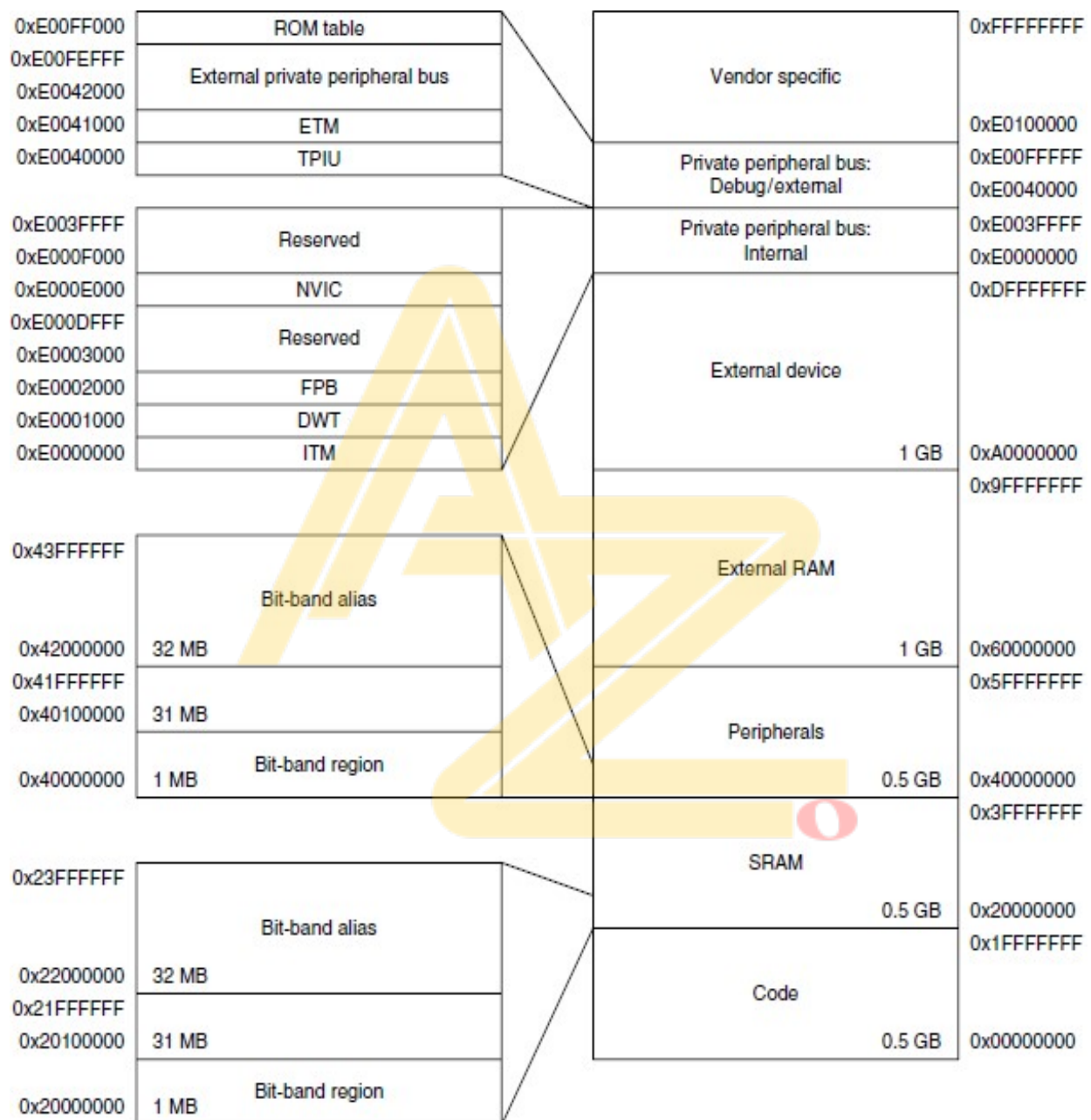


Figure 1.5 Cortex-M3 Predefined Memory Map.

Memory Access Attributes

Bufferable: Write to memory can be carried out by a write buffer while the processor continues on next instruction execution.

Cacheable: Data obtained from memory read can be copied to a memory cache so that next time it is accessed the value can be obtained from the cache to speed up the program execution.

Executable: The processor can fetch and execute program code from this memory region. **Sharable:** Data in this memory region could be shared by multiple bus masters. Memory system needs to ensure coherency of data between different bus masters in shareable memory region

Code - This executable region is for program code. Data can also be stored here. This region is Executable, Cacheable, Bufferable

SRAM - This executable region is for data storage. Code can also be stored here. This region includes bit band and bit band alias areas. This region is Executable, Cacheable, Bufferable

SRAM Bit-band alias - Direct accesses to this memory range behave as SRAM memory accesses, but this region is also bit addressable through bit-band alias.

SRAM bit-band region - Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write.

PERIPHERAL - This region includes bit band and bit band alias areas. Instruction Execution not allowed. This region is Non-executable, Non-Cacheable, Bufferable.

Peripheral Bit-band alias - Direct accesses to this memory range behave as peripheral memory accesses, but this region is also bit addressable through bit-band alias.

Peripheral bit-band region - Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write.

EXTERNAL DEVICE - This region is used for external device and Shared memory. Non-executable, Non-Bufferable

EXTERNAL RAM - This region is intended for either on-chip or off-chip memory and this region is used for data. Executable, Cacheable.

SYSTEM LEVEL This region is for private peripherals and vendor-specific devices. It is nonexecutable.

PPB memory range-noncacheable, nonbufferable

Vendor-specific memory region- bufferable and noncacheable.

PRIVATE PERIPHERAL BUS - INTERNAL - Provides access to the:

Instrumentation Trace Macrocell (ITM)

Data Watchpoint and Trace (DWT)

Flashpatch and Breakpoint (FPB)

System Control Space (SCS)

MPU and Nested Vectored Interrupt Controller (NVIC)

PRIVATE PERIPHERAL BUS - EXTERNAL - Provides access to the

Trace Port Interface Unit (TPIU)

Embedded Trace Macrocell (ETM)

ROM table

Implementation-specific areas of the PPB memory map.

2.13.1 Bit-Band Operations

Bit-band operation support allows a single load/store operation to access (read/write) to a single data bit. In the Cortex-M3, this is supported in two predefined memory regions called bit-band regions. One of them is located in the first 1 MB of the SRAM region, and the other is located in the first 1 MB of the peripheral region. These two memory regions can be accessed like normal memory, but they can also be accessed via a separate memory region called the bit-band alias. When the bit-band alias address is used, each individual bit can be accessed separately in the least significant bit (LSB) of each word-aligned address.

For example, to set bit 2 in word data in address 0x20000000, instead of using three instructions to read the data, set the bit, and then write back the result, this task can be carried out by a single instruction. The assembler sequence for these two cases could be like the one shown in Figure 1.6.

Similarly, bit-band support can simplify application code if we need to read a bit in a memory location. For example, if we need to determine bit 2 of address 0x20000000, we use the steps outlined in Figure The assembler sequence for these two cases could be like the one shown in Figure. 17

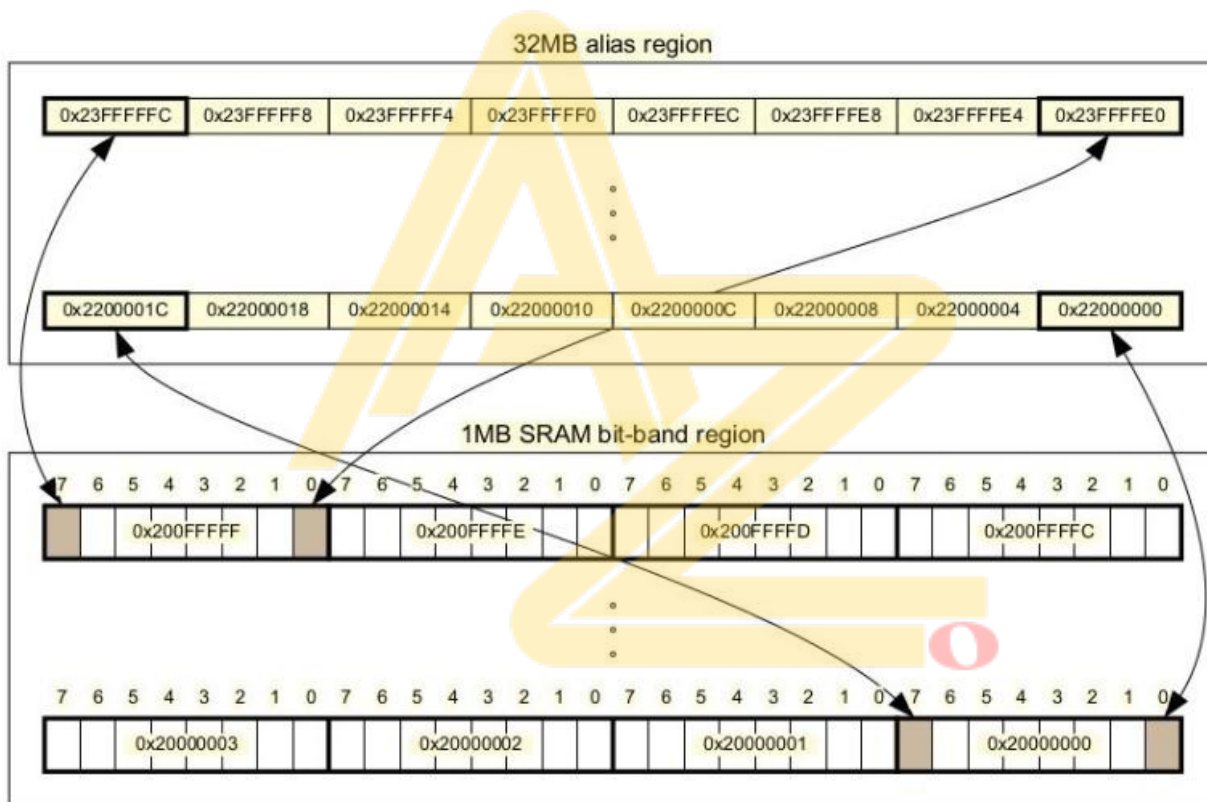
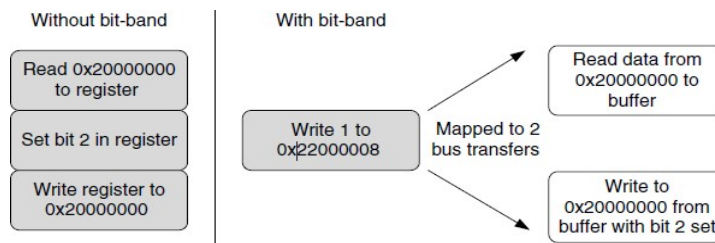


Figure 1.6 Bit Accesses to Bit-Band Region via the Bit-Band Alias



Without bit-band	With bit-band
<pre>LDR R0,=0x20000000 ; Setup address LDR R1, [R0] ; Read ORR.W R1, #0x4 ; Modify bit STR R1, [R0] ; Write back result</pre>	<pre>LDR R0,=0x22000008 ; Setup add MOV R1, #1 ; Setup dat STR R1, [R0] ; Write</pre>

Figure 1.8 Example Assembler Sequence to Write a Bit with and without Bit-Band

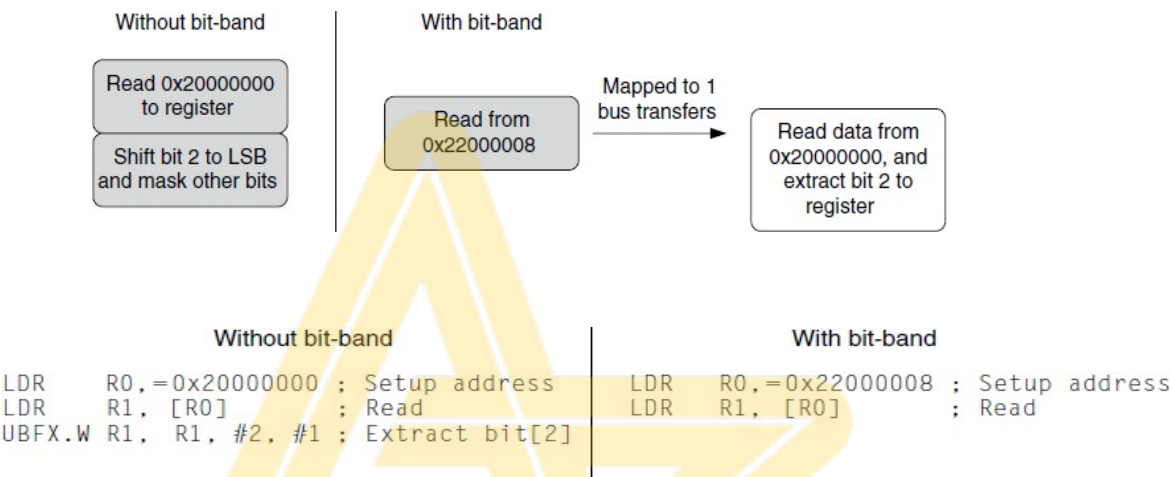


Figure 1.9: Read from the Bit-Band Alias.

Table 2.21: Remapping of Bit-Band Addresses SRAM Region

Bit-Band Region	Aliased Equivalent
0x20000000 bit[0]	0x22000000 bit[0]
0x20000000 bit[1]	0x22000004 bit[0]
0x20000000 bit[2]	0x22000008 bit[0]
...	...
0x20000000 bit[31]	0x2200007C bit[0]
0x20000004 bit[0]	0x22000080 bit[0]
...	...
0x20000004 bit[31]	0x220000FC bit[0]
...	...
0x200FFFFC bit[31]	0x23FFFFFC bit[0]

Table 2.22: Remapping of Bit-Band Addresses in Peripheral Memory Region

Bit-Band Region	Aliased Equivalent
0x40000000 bit[0]	0x42000000 bit[0]
0x40000000 bit[1]	0x42000004 bit[0]
0x40000000 bit[2]	0x42000008 bit[0]
...	...
0x40000000 bit[31]	0x4200007C bit[0]
0x40000004 bit[0]	0x42000080 bit[0]
...	...
0x40000004 bit[31]	0x420000FC bit[0]
...	...
0x400FFFFC bit[31]	0x43FFFFFC bit[0]

Bit-band region: This is a memory address region that supports bit-band operation.

Bit-band alias: Access to the bit-band alias will cause an access (a bit-band operation) to the bit-band region. A memory remapping is performed

Advantages of Bit-band operation

- Reading the whole register
- Masking the unwanted bits
- Comparing and branching
- Reading the status bit via the bit-band alias (get 0 or 1)
- Comparing and branching
- READ-MODIFY-WRITE sequence cannot be interrupted by other bus activities

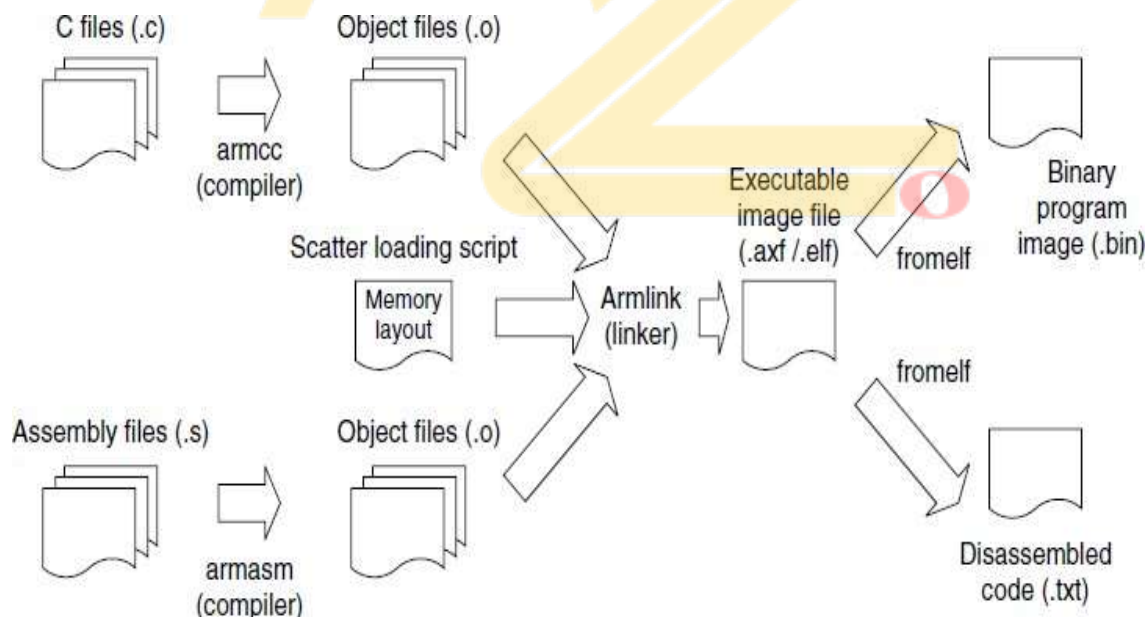
Cortex-M3 Programming

The Cortex™-M3 can be programmed using either assembly language, C language, or other high-level languages like National Instruments LabVIEW.

- For most embedded applications using the Cortex-M3 processor, the software can be written entirely in C language.
- There are of course some people who prefer to use assembly language or a combination of C and assembly language in their projects.

A Typical Development Flow

- The concepts of code generation flow in terms of these tools are similar.
- For the most basic uses, you will need assembler, a C compiler, a linker, and binary file generation utilities.
- For ARM solutions, the Real View Development Suite (RVDS) or Real View Compiler Tools (RVCT) provide a file generation flow



Using Assembly

- ✓ For small projects, it is possible to develop the whole application in assembly language.
- ✓ Using assembler, best optimization is possible, but it increases the development time.
- ✓ Handling complex data structures or function library management can be extremely difficult.

Even when the C language is used in a project, in some situations part of the program is implemented in assembly language as follows:

- Functions that cannot be implemented in C, such as direct manipulation of stack data or special instructions that cannot be generated by the C compiler in normal C-code.
- Timing-critical routines.
- Tight memory requirements, causing part of the program to be written in assembly to get the smallest memory size.

The Interface between Assembly and C

In various situations, assembly code and the C program interact. For example:

- ✓ When embedded assembly is used in C program code.
- ✓ When C program code calls a function or subroutine implemented in assembler in a separate file. When an assembly program calls a C function or subroutine

Examples: ASSEMBLY CODE TO ADD TEN NUMBERS

STACK_TOP EQU 0x20002000; constant for SP starting value

AREA |Header Code |, CODE

DCD STACK_TOP ; Stack top

DCD Start ; Reset vector

ENTRY ; Indicate program ;execution start here

Start ; Start of main program

; initialize registers

MOV r0, #10 ; Starting loop counter value

MOV r1, #0 ; starting result

MOV r0, #10 ; Starting loop counter value

MOV r1, #0 ; starting result

; Calculated 10+9+8+...+1

loop

ADD r1, r0 ; R1 = R1 + R0

SUBS r0, #1 ; Decrement R0, update flag ("S" suffix)

BNE loop ; If result not zero jump to loop Result is now in R1

deadloop


B deadloop ; Infinite loop

END ; End of file

Examples: Simple C Program to blink LED

```
#define LED *((volatile unsigned int *) (0xDFFF000C))

int main (void)
{
    int i; /* loop counter for delay function */
    volatile int j; /* dummy volatile variable to prevent
C compiler from optimize the delay away */
    while (1) {
        LED = 0x00; /* toggle LED */
        for (i=0;i<10;i++) {j=0;} /* delay */
        LED = 0x01; /* toggle LED */
        for (i=0;i<10;i++) {j=0;} /* delay */
    }
    return 0;
}
```



CMSIS: Cortex Microcontroller Software Interface Standard

The aims of CMSIS are to:

1. Improve software portability and reusability
2. Enable software solution suppliers to develop products that can work seamlessly with device
3. Libraries from various silicon vendors allow embedded developers to develop software quicker with an easy-to-use and standardized software interface
4. Allow embedded software to be used on multiple compiler products
5. Avoid device driver compatibility issues when using software solutions from multiple sources

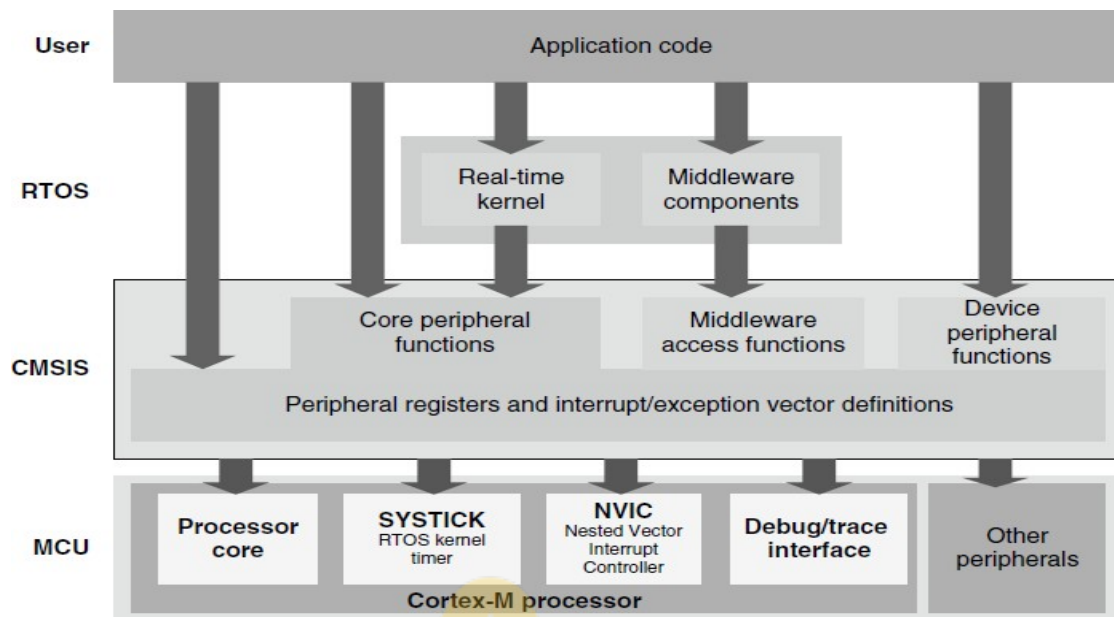
Areas of standardization

- **Hardware Abstraction Layer (HAL) for Cortex-M processor registers:** This includes standardized register definitions for NVIC, System Control Block registers, SYSTICK register, MPU registers, and a number of NVIC and core feature access functions.
- **Standardized system exception names:** This allows OS and middleware to use system exceptions easily without compatibility issues.
- **Standardized method of header file organization:** This makes it easier for users to learn new cortex microcontroller products and improve software portability.
- **Common method for system initialization:** Each Microcontroller Unit (MCU) vendor provides a SystemInit() function in their device driver library for essential setup and configuration, such as initialization of clocks.
- **Standardized intrinsic functions:** By having standardized intrinsic functions, software reusability and portability are considerably improved.
- **Common access functions for communication:** This provides a set of software interface functions for common communication interfaces including universal asynchronous receiver/transmitter (UART), Ethernet, and Serial Peripheral Interface (SPI).
- **Standardized way for embedded software to determine system clock frequency:** A software variable called System Frequency is defined in device driver code. This allows embedded OS to set up the SYSTICK unit based on the system clock frequency

Organization of CMSIS

The CMSIS is divided into multiple layers as follows:

1. Core Peripheral Access Layer :Name definitions, address definitions, and helper functions to access core registers and core peripherals.
2. Middleware Access Layer: Common method to access peripherals for the software industry. Targeted communication interfaces include Ethernet, UART and SPI. Allows portable software to perform communication tasks on any Cortex microcontrollers that support the required communication interface.
3. Device Peripheral Access Layer (MCU specific): Name definitions, address definitions, and driver code to access peripherals
4. Access Functions for Peripherals (MCU specific): Optional additional helper functions for peripherals



Using CMSIS

Since the CMSIS is incorporated inside the device driver library, there is no special setup requirement for using CMSIS in projects.

For each MCU device, the MCU vendor provides a header file, which pulls in additional header files required by the device driver library, including the Core Peripheral Access Layer defined by ARM.

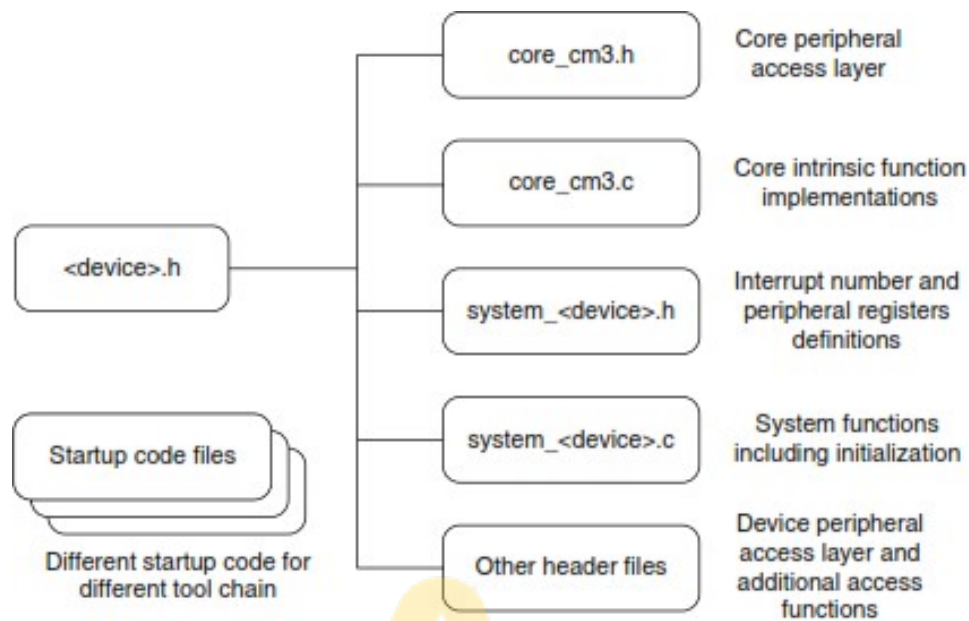
The file **core_cm3.h** contains the peripheral register definitions and access functions for the Cortex-M3 processor peripherals like NVIC, System Control Block registers, and SYSTICK registers.

The **core_cm3.h** file also contains declaration of CMSIS intrinsic functions to allow C applications to access instructions that cannot be generated using IEC/ISO C language. In addition, this file also contains a function for outputting a debug message via the Instrumentation Trace Module (ITM).

The **system_<device>.h** file contains microcontroller specific interrupt number definitions, and peripheral register definitions.

The **system_<device>.c** file contains a microcontroller specific function called **SystemInit** for system initialization.

In addition, CMSIS compliant device drivers also contain start-up code (which contains the vector table



CMSIS Files

Benefits of CMSIS

- Better software portability and reusability.
- Allows software to be quickly ported between Cortex-M3 and other Cortex-M processors, reducing time to market. For embedded OS vendors and middleware providers, the advantages of the CMSIS are significant.
- By using the CMSIS, their software products can become compatible with device drivers from multiple microcontroller vendors.

Without the CMSIS, the software vendors either have to include a small library for Cortex-M3 core functions or develop multiple configurations of their product so that it can work with device libraries from different microcontroller vendors.

Recommended Questions

1. What do you mean by UAL? Give example instructions indicating its relevance
2. How the data definition directives are used in ARM Cortex-M3? Provide example instructions.
3. How the suffixes are used in assembler language? Provide example instructions to use suffixes.
4. Explain the data movement instructions in ARM cortex M3.
5. Write a note on LDR and ADR Pseudo-Instructions.
6. Explain the arithmetic instruction set in ARM cortex M3 with an example
7. Explain the logical instruction set in ARM cortex M3 with an example
8. Explain the shift and rotate instructions available in ARM cortex M3 with an example
9. Analyze the following instructions and write the contents of the registers after the execution of each instruction

Assume: **R0=0x00000088**, **R1=0x00001111**, **R2=0x000000A**

ADD R3,R0,R1

BIC R4,R1,#0X04

ORR R4, R2

REV R5,R4

10. Explain the following instructions with suitable example:

Arithmetic Instructions	Logic Operation Instructions	Shift and Rotate Instructions	Sign Extend Instructions	Data Reverse Ordering Instructions	Bit field processing Instructions
ADD	AND	ASR	SXTB	REV	BFC
ADC	ORR	LSL	SXTH	REV16	BFI
SUB	BIC	LSR		REVSH	CLZ
SBC	ORN	ROR			RBIT
RSB	EOR	RRX			SBFX
MUL					UBFX
SMULL					
SMLAL					
UMULL					
UMLAL					
UDIV					
SDIV					

11. Briefly describe the branch (*unconditional and conditional branch*) and call instructions in ARM cortex M3.

12. Explain the following instructions with an example in cortex M3

Branch and Call	B	BX	BL		
Compare Instructions	CMP	CMN	TST	CBZ	CBNZ
Saturation instructions	SSAT	USAT			
Memory Barrier Instructions	DMB	DSB	ISB		

13. Explain the IF THEN (IT) instruction in ARM Cortex M3 with an example.

14. Describe the instruction barrier and memory barrier instructions in arm cortex m3.

15. Write the memory map and explain memory access attributes in Cortex M3.

16. Describe the Bit-band operations in ARM cortex M3 and mention its advantages

17. With a neat diagram explain the organization of CMSIS

18. Explain the typical program development flow in Arm cortex M3 and describe how to interface assembly and C language.

19. Write an ALP to find the sum of first 10 integers

20. Write a C program to toggle an LED with a small delay in Cortex M3