Example 1:

To set the fourth bit (bit[3]) of the 32-bit data at 0x20000000 address to '1', we will write '1' to address 0x2200000C, which is the alias address of the fourth bit of the 32-bit data at 0x20000000.

```
;Bit-band Operation

LDR R1, =0x2200000C ;Setup address
MOV R0, #1              ;Load data
STR R0, [R1]            ;Write
```

Example 2:

In this example, we demonstrate read and write operations to/from the bit-band alias region:

1. Store the value 0x3355AACC at memory address 0x20000000.
2. Read from memory address 0x22000008. This read access is remapped to access 0x20000000. The returned value is 1 (bit[2], the third bit, of 0x3355AACC), it is like reading of bit[2] from memory address 0x20000000.
3. Write 0x0 to to memory address 0x22000008. This write access is remapped into a READ-MODIFY-WRITE to 0x20000000. The value 0x3355AACC is read from memory, bit 2 is cleared, and resulting in 0x3355AAC8, which is then written back to memory address 0x20000000.
4. Now read 0x20000000. That gives you a return value of 0x3355AAC8 (bit[2] cleared).

1.8 Exceptions and interrupts

1.8.1 What are exceptions?

Exceptions are events that cause changes to program flow. In the ARM architecture, interrupts are one type of exception. Interrupts are usually generated from peripheral or external inputs, and in some cases they can be triggered by software. The exception handlers for interrupts are also referred to as Interrupt Service Routines (ISR). When an exception occurs, the processor will perform the following tasks (See Figure 1.24):

1. Suspend the current executing task in the main code;
2. Perform some hardwired routines, such as saving registers;
3. Execute a part of the program called the exception handler or interrupt service routine (ISR), including a return-from-interrupt instruction at the end;
4. Resume running the main code.

Entering exception handler: Hardwired CPU response activities

After the exception is triggered, the different activities that should be taken to handle this exception are:

- Finish the current instruction, except for lengthy instructions, will be abandoned and restarted after the ISR execution;

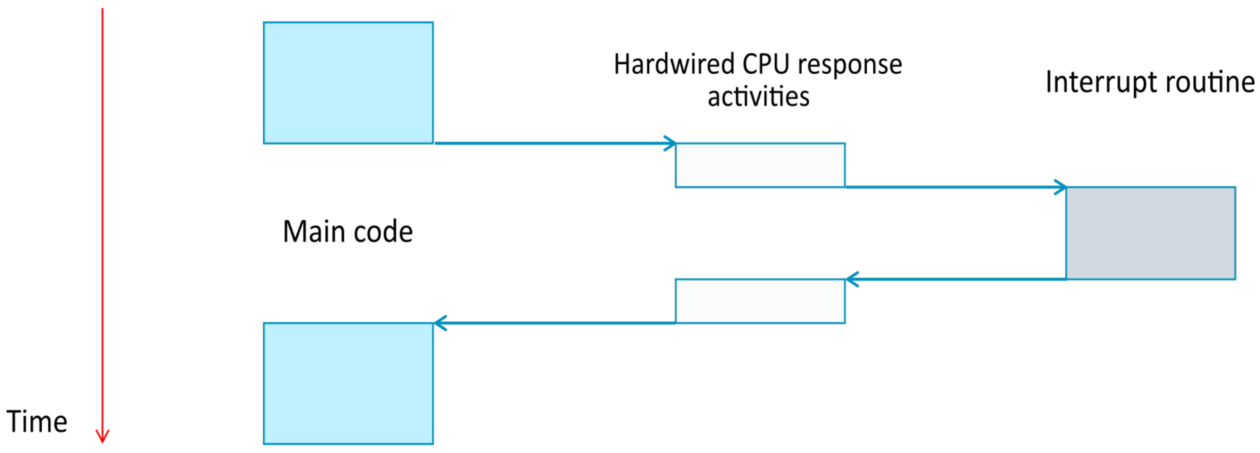


Figure 1.24: Interrupt or Exception Processing Sequence

- Save the current architectural state (pushing eight registers) onto the stack xPSR, PC(R15), LR(R14), R12, R3, R2, R1, R0. This operation is called “Stacking” (See Figure 1.25) ;
- Switch code execution into handler/privileged mode and the processor must be using MSP;
- Load IPSR (Interrupt Program Status Register) with the correct exception number to be executed (handler type) which is in [8:0] bit position (≥ 1 and ≤ 255) on the xPSR register;
- Load PC with the address of the exception handler which will be found on the vector table;
- Load the EXC_RETURN value into the LR, which indicates from which stack pointer to restore registers: MSP (0) or PSP (1), and which mode to return to: Handler (0) or Thread (1) (the EXC_RETURN value = 0xFFFF_FFF9 indicates return to thread with MSP) ;
- Start executing the code of the exception handler, which can only be stopped by another higher priority interrupt, and registers values will be pushed to the stack.

Usually, 16 clock cycles are needed from the exception request to the execution of the first instruction in the handler.

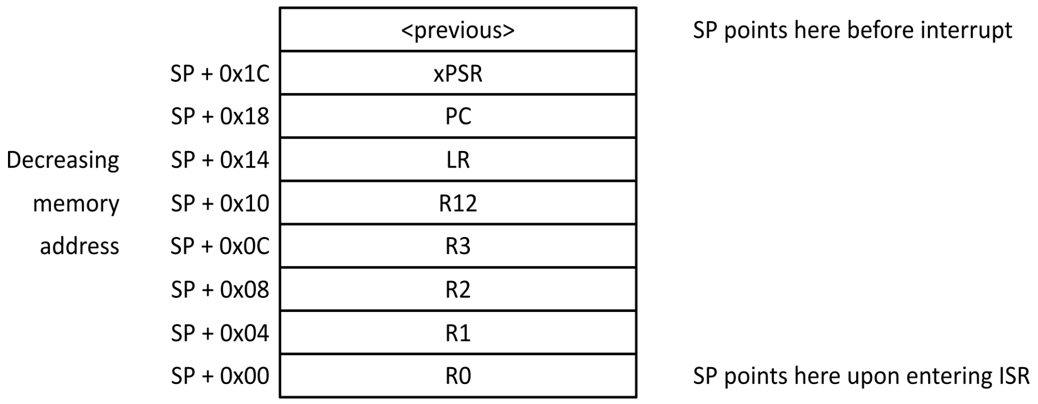


Figure 1.25: Pushing current architecture context onto current stack.

Note: *The used SP (main (MSP) or process (PSP)) depends on the operating mode, as determined by second bit (bit[1]) of the CONTROL register, while the Handler mode always uses MSP.*

Exiting an exception handler

Exiting the exception handler involves three steps:

1. Execute the instruction triggering the exception return processing; two methods are possible:
    - As there is no special instruction for returning from an exception or interrupt, it will be depends on where the return address has been stored:
        - If the return address is still in the LR, use the regular instruction **BX LR**, which branches to the address stored in the Link Register (LR) by loading the PC with the contents of LR.
        - If the return address has been stored on the stack, use **POP ..., PC** to pop the address from the stack into the PC.
    - Using the special value **EXC_RETURN**, which will trigger the exception return mechanism when it is loaded into the Program Counter (PC):
        - Use **BX LR** if the EXC_RETURN value is still in the LR.
        - Use **POP** if EXC_RETURN has been saved on the stack.
          Note: *The EXC_RETURN value lies in the memory range 0xF0000000 to 0xFFFFFFFF. This range is in the system region defined as non-executable in the architecture, thus preventing accidental execution of non-instruction data.*
2. Check EXC_RETURN (bit 2) to determine from which stack (MSP or PSP) the context should be restored (e.g., PC=0xFFFF_FFF9 indicates a return to thread mode using the main stack pointer (MSP)).
3. Perform the "unstacking" operation, where R0, R1, R2, R3, R12, LR, PC, and xPSR are restored, setting the SP back to its previous value, and reactivating thread mode. The system then resumes the same execution state as before the interrupt.

What are exception sources ?

Exceptions are processed by the Nested Vectored Interrupt Controller (NVIC). These exceptions can be (See Figure 1.26):

1. **Interrupt Requests (IRQs)** generated by on-chip peripherals or from external interrupt inputs through I/O ports.
2. **Non-Maskable Interrupt (NMI) request** that could be used by a watchdog timer or brownout detector (a voltage supply monitoring level unit).
3. **Timer called SysTick** that can generate a periodic timer interrupt request, which can be used for simple timing control.
4. **Processor exception events** - These could be fault events indicating system error conditions or exceptions generated by software to support embedded OS operations.

1.8.2 Microcontroller interrupts

As mentioned in the previous section, the interrupts can be generated by on-chip peripherals, processor exception events or from external through I/O port, so the different interrupts can be classified to:

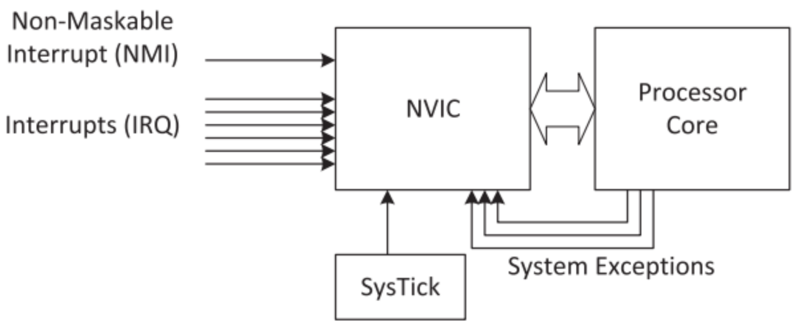


Figure 1.26: Various exception sources.

1. **Hardware interrupts:** they are based on asynchronous external events; Examples : interrupt is asserted, character is received on serial port, or ADC converter finishes conversion for example;
2. **Exceptions, faults, software interrupts:** They usually occur synchronously which means they are the result of undesirable behavior of executed instructions; Examples : undefined instructions, overflow occurs for a given instruction

Each exception source has an exception number that can be found in various registers, including the IPSR. It is used to determine the exception vector addresses stored in a vector table. Exception numbers 1 to 15 are classified as system exceptions, and exceptions 16 and above are for interrupts. Cortex-M4 processors can support up to 240 interrupt inputs; however, the typical range is from 16 to 100.

1.8.3 Nested vectored interrupt controller (NVIC)

The NVIC is a part of the Cortex-M4. It is programmable and its registers are located in the System Control Space (SCS) of the memory map (from 0xE000E000 to 0xE000EFFF). The NVIC handles the exceptions and interrupt configurations, prioritization, and interrupt masking. The NVIC has the following features:

Flexible exception and interrupt management

Each interrupt (apart from the NMI) can be enabled or disabled and can have its pending status set or cleared by software. The NVIC can handle various types of interrupt sources:

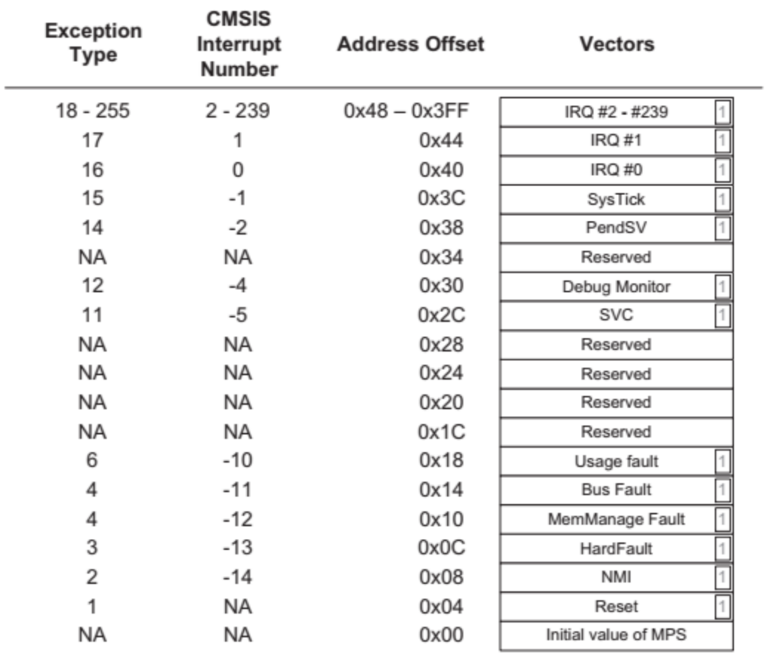
- **Pulsed interrupt request** - when the NVIC receives a pulse at its interrupt input, the pending status is set and held until the interrupt gets serviced.
- **Level triggered interrupt request** - the interrupt source holds the request high until the interrupt is serviced.

Nested exception/interrupt support

When an exception occurs, the NVIC will compare the priority level of this exception to the current level. If the new exception has a higher priority, the current running task will be suspended. This process is called “preemption.” When the higher priority exception handler is complete, it resumes the exception that was running previously.

Vectored exception/interrupt entry

The Cortex-M processors automatically locate the starting point of the exception handler from a vector table in the memory (See Figure 1.27).



Exception Type	CMSIS Interrupt Number	Address Offset	Vectors	
18 - 255	2 - 239	0x48 – 0x3FF	IRQ #2 - #239	1
17	1	0x44	IRQ #1	1
16	0	0x40	IRQ #0	1
15	-1	0x3C	SysTick	1
14	-2	0x38	PendSV	1
NA	NA	0x34	Reserved	
12	-4	0x30	Debug Monitor	1
11	-5	0x2C	SVC	1
NA	NA	0x28	Reserved	
NA	NA	0x24	Reserved	
NA	NA	0x20	Reserved	
NA	NA	0x1C	Reserved	
6	-10	0x18	Usage fault	1
4	-11	0x14	Bus Fault	1
4	-12	0x10	MemManage Fault	1
3	-13	0x0C	HardFault	1
2	-14	0x08	NMI	1
1	NA	0x04	Reset	1
NA	NA	0x00	Initial value of MPS	

Figure 1.27: Exception types (LSB of exception vectors should be set to 1 to indicate Thumb state)

Interrupt masking

The Cortex-M4 provides several interrupt masking registers, such as the PRIMASK special register, which allows disabling all exceptions except HardFault and NMI. Alternatively, the BASEPRI register can be used to selectively mask exceptions or interrupts below a certain priority level.

In addition, the NVIC saves, and automatically restores, a set of the CPU registers (R0-R3, R12, PC, PSR, and LR), does a quick entry to the next pending interrupt without a complete pop/push sequence.

1.8.4 Vector table

The vector table is an array of word data located in the system memory, with each entry representing the starting address of one exception type (Figure 1.27). The vector table is relocatable, and its relocation is controlled by the Vector Table Offset Register (VTOR) in the NVIC. After a reset, the VTOR is set to 0 by default; hence, the vector table is initially located at address 0x0. Since the Cortex-M processors can support only Thumb instructions, the LSB of all the exception vectors should be set to 1.

1.9 Instruction Set

The Instruction Set Architecture (ISA) of ARM has evolved from ARM instructions to Thumb-2, as illustrated in Figure 1.27. ARM Cortex-M processors are exclusively based on Thumb-2 technology, which is mix of Arm and Thumb-1 Instruction sets, Benefiting from both 32-bit Arm (high performance) and 16-bit Thumb-1 (high code density). Compared to 32-bit Arm instructions set, Thumb-2 code size is reduced by 26%, with similar performance.

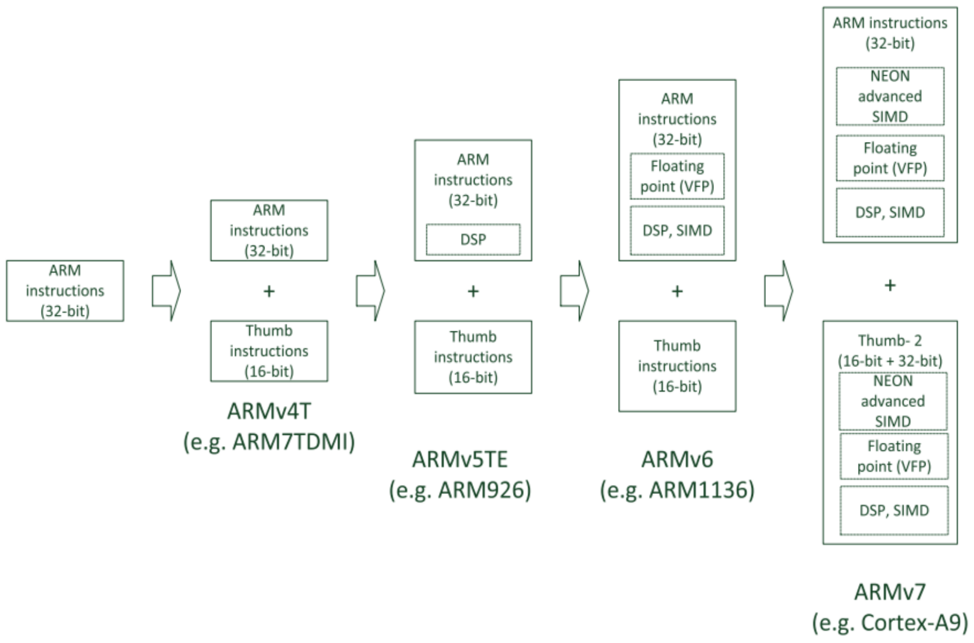

ARM instructions (32-bit)
ARM instructions (32-bit)
+
Thumb instructions (16-bit)
ARMv4T
(e.g. ARM7TDMI)
ARM instructions (32-bit)
DSP
+
Thumb instructions (16-bit)
ARMv5TE
(e.g. ARM926)
ARM instructions (32-bit)
Floating point (VFP)
DSP, SIMD
+
Thumb instructions (16-bit)
ARMv6
(e.g. ARM1136)
ARM instructions (32-bit)
NEON advanced SIMD
Floating point (VFP)
DSP, SIMD
+
Thumb- 2 (16-bit + 32-bit)
NEON advanced SIMD
Floating point (VFP)
DSP, SIMD
ARMv7
(e.g. Cortex-A9)

Figure 1.28: Evolution of the ARM Instruction Set Architecture

The instruction set support in the current Cortex-M processors is illustrated in Figure 1.29, where the 16 instructions are for to general data processing and I/O control tasks. while advanced data processing, hardware division, bit field manipulation, MAC (Multiply Accumulate) are 32-bit instructions and supported by both the Cortex-M3 and M4. Additionally, there are other instructions specifically designed for DSP applications or floating-point operations, and these are supported by the Cortex-M4 only.

Assembly language syntax

The general format for an assembly instruction for ARM Keil compilers is as follows:

```
label
      mnemonic   operand1,   operand2,   operand3,        ; Comments
```

Where:

- **Label** is used as a reference to an address location.
- **Mnemonic** is a symbolic abbreviation that replaces the instruction opcode.
- **Operand1** can be:
  - The destination of the operation for data processing instructions.
  - The register into which data is loaded for a memory read instruction (except multiple load instructions).
  - The register that holds the data to be written to memory for a memory write instruction (except multiple store instructions).
- **Operand2** is normally the source of the operation and is usually a register.
- **Operand2** is the source operand and may be a register, an immediate number, a register shifted by a constant amount of bits (using the Barrel shifter), or a register plus an offset.

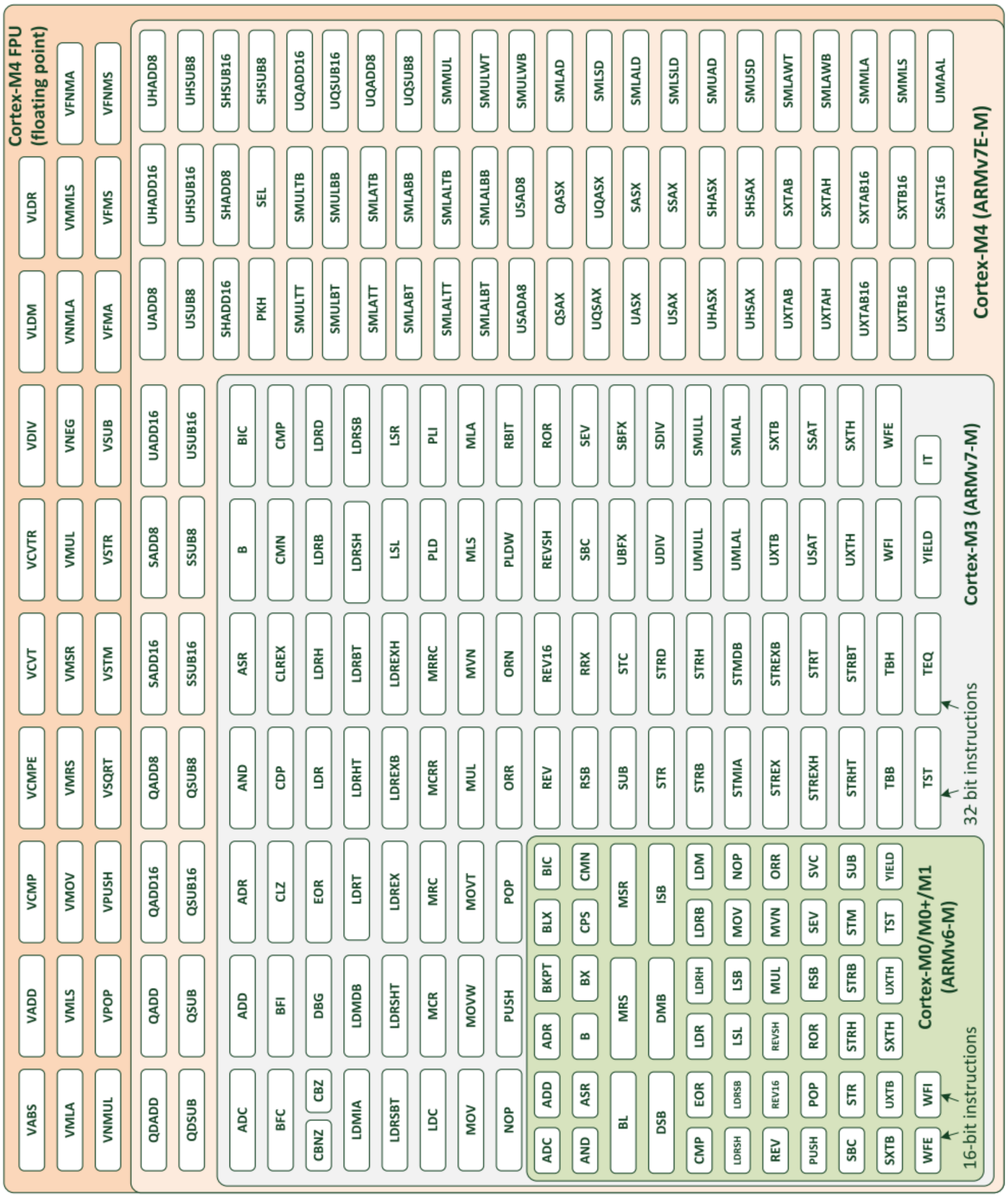


Figure 1.29: Instruction set of the Cortex-M processors

- **Comments** are written after a semicolon (";"), which does not affect the program.

Nevertheless, instructions that handle multiple loads and stores have a different syntax and the number of operands depends on the instruction type, some instructions do not require any operand, while others may need just one.

The following example demonstrates how to add two registers together in ARM assembler.

```
ADD   R0 ,   R2 ,   R3        ; R0 = R2 + R3
```

"ADD" is a mnemonic for arithmetic addition, register R0 is the destination operand, and registers R2 and R3 are two source operands.

Note: Assembly code can be assembled by either Arm assembler (armasm) or assembly tools from a variety of vendors (e.g. GNU tool chain). When using the GNU tool chain, the syntax for labels and comments is slightly different.

The instructions in the Cortex-M3 and Cortex-M4 processors can be divided into various groups based on functionality:

- Moving data within the processor
- Memory accesses
- Arithmetic operations
- Logic operations
- Shift and Rotate operations
- Conversion (extend and reverse ordering) operations
- Bit field processing instructions
- Program flow control (branch, conditional branch, conditional execution, and function calls)
- Multiply accumulate (MAC) instructions
- Divide instructions
- Memory barrier instructions
- Exception-related instructions
- Sleep mode-related instructions
- Other functions

In addition, the Cortex-M4 processor supports the Enhanced DSP instructions:

- SIMD operations and packing instructions
- Adding fast multiply and MAC instructions
- Saturation algorithms
- Floating point instructions (if the floating point unit is present)

1.9.1 Moving data within the processor

Moving data within the processor might be:

→ Move data from one register to another
→ Move data between a register and a special register (e.g., CONTROL, PRIMASK, FAULTMASK, BASEPRI)
→ Move an immediate constant into a register

Table 1.2 shows some examples of these operations:

Table 1.2: Instructions for Transferring Data within the Processor

Instruction	Dest	Source	Operations
MOV	R4,	R0	; Copy value from R0 to R4
MOVS	R4,	R0	; Copy value from R0 to R4 with APSR (flags) update
MRS	R7,	PRIMASK	; Copy value of PRIMASK (special register) to R7
MSR	CONTROL,	R2	; Copy value of R2 into CONTROL (special register)
MOV	R3,	#0x34	; Set R3 value to 0x34 (immediate value is 8-bit)
MOVS	R3,	#0x34	; Set R3 value to 0x34 with APSR update
MOVW	R6,	#0x1234	; Set R6 to a 16-bit constant 0x1234
MOVT	R6,	#0x8765	; Set the upper 16-bit of R6 to 0x8765
MVN	R3,	R7	; Move negative value of R7 into R3

The following conclusions can be drawn from Table 1.2:

- The suffix "S" is appended to the MOV instruction to update the APSR flags based on the operation's result.
- The MOVW instruction is used to set a register to a larger immediate value (between 9-bit to 16-bit).
- To set a register to a 32-bit immediate data value, one can utilize a pseudo instruction called "LDR," or alternatively, employ a combination of MOVW and MOVT instructions as explained in the example below.

```
LDR    R0 ,   = 0x12345678      ; Set R0 to 0x12345678
```

```
MOVW    R0 ,   #0x789A          ; Set R0 to 0x0000789A
MOVT    R0 ,   #0x3456          ; Set upper 16-bit of R0 to 0x3456,
                                ; now R0 = 0x3456789A
```

1.9.2 Memory access instructions

There are numerous memory access instructions available in the Cortex-M3 and Cortex-M4 processors. However, for normal data transfers, the instructions available are given in Table 1.3.

Immediate offset (pre-index)

For "pre-index" addressing, the base register is combined with an immediate constant value (offset) to calculate the memory address before accessing memory, as shown in the example below:

```
LDRB    R0 ,   [R1 ,  #0x3]     ; Read a byte value from address R1+ 0x3,
                                ; and store the read data in R0.
```

Table 1.3: Memory Access Instructions for Various Data Sizes

Data Type	**Load (Read from Memory)**	**Store (Write to Memory)**
8-bit unsigned	LDRB	STRB
8-bit signed	LDRSB	STRB
16-bit unsigned	LDRH	STRH
16-bit signed	LDRSH	STRH
32-bit	LDR	STR
Multiple 32-bit	LDM	STM
Double-word (64-bit)	LDRD	STRD
Stack operations (32-bit)	POP	PUSH

Table 1.4 shows a list of commonly used load and store instructions.

Table 1.4: Memory Access Instructions with Immediate Offset

Instruction (**#offset field is optional**)	**Description**
LDRB Rd, [Rn, #offset]	Read byte from memory location (mer. loc.) Rn + offset (8-bit)
LDRSB Rd, [Rn, #offset]	Read and signed extend byte from memory location Rn + offset
LDRH Rd, [Rn, #offset]	Read half-word from memory location Rn + offset
LDRSH Rd, [Rn, #offset]	Read and signed extended half-word from mer. loc. 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

R Adding the exclamation mark (!) to the end of all instructions in Table 1.4 will enable the destination register to hold back the address when the instruction is completed (the 16-bit versions of these instructions only support low registers (R0-R7) and do not provide write back.). For example:

```
LDR   R0 ,   [R1 ,  #0x8]!      ; After the access to memory[R1+0x8]
                               ; R1 is updated to R1+0x8
```

PC-related addressing (Literal)

The load instructions in Table 1.4 can generate the address value from the current PC value and an offset value by replacing the Rn register with PC, as shown in the example below.

```
LDRB    Rd ,    [PC,  #offset]     ; Load unsigned byte into Rd using PC offset
```

Register offset (pre-index)

Another useful address mode is the register offset. This mode is used in the processing of data arrays where the address is a combination of a base address and an offset calculated from an index value. The index value can also be shifted by a distance of 0 to 3 bits before being added to the base register. For example:

```
LDR   R3 ,   [R0,  R2,  LSL  #2]     ; Read memory[R0+(R2 ≪ 2)] into R3
```

The shift operation is optional. You can have a simple operation like

```
STR   R5 ,   [R0,   R7]       ; Write R5 into memory[R0+R7]
```

Post-index

Memory access instructions with post-index addressing mode also have an immediate offset value. However, the offset is not used during the memory access, but is used to update the address register after the data transfer is completed. This mode is very useful for processing data in an array. An instruction using this mode is illustrated in the following example::

```
LDR  R0 ,  [R1] ,  #offset     ; Read memory[R1], then R1 updated to R1+offset
```

This addressing mode can be utilized with all the instructions listed in Table 1.4.

R When the post-index memory addressing mode is used, there is no need to use the exclamation mark (!) sign because the base address register is always updated if the data transfer is completed successfully.

Multiple load and multiple store

The ARM architecture allows you to read or write multiple data that are contiguous in memory. The LDM (Load Multiple registers) and STM (Store Multiple registers) instructions only support 32-bit data. They support two types of pre-indexing:

- **IA**: **I**ncrement address **A**fter each read/write
- **DB**: **D**ecrement address **B**efore each read/write

The LDM and STM instructions are demonstrated in Table 1.5.

The < reg list > in Table 1.5 is the register list. It contains at least one register, and:

- Start with "{" and end with "}"
- Use "-" (hypen) to indicate range. For example, R0-R4 means R0, R1, R2, R3 and R4.
- Use "," (comma) to separate each register

Table 1.5: Multiple Load/Store Memory Access Instructions

Instruction	Description
LDMIA Rn, < reg list >	Read multiple words from memory location specified by Rn. Address Increment After (IA) each read.
LDMDB Rn, < reg list >	Read multiple words from memory location specified by Rn. Address Decrement Before (DB) each read.
STMIA Rn, < reg list >	Write multiple words to memory location specified by Rn. Address increment after each write.
STMDB Rn, < reg list >	Write multiple words to memory location specified by Rn. Address Decrement Before each write.

For example, the following instructions read address 0x20000000 to 0x2000000F (four words) into R0 to R3:

```
LDR    R4 ,   =0x20000000       ; Set R4 to 0x20000000 (address)
LDMIA    R4 ,   { R0-R3 }       ; Read 4 words and store them to R0 - R3
```

The register list can be non-contiguous such as { R1, R3, R5-R7, R9, R11-R12 },which contains R1, R3, R5, R6, R7, R8, R11, R12.

Similar to other load/store instructions, you can use write back with STM and LDM. For example:

```
LDR    R8,   =0x8000           ; Set R8 to 0x8000 (address)
STMIA    R8! ,   { R0-R3 }      ; R8 change to 0x8010 after the store
```

Stack push and pop

Stack push and pop are another form of the store multiple and load multiple. They use the currently selected stack pointer for address generation, which can either be the Main Stack Pointer (MSP), or the Process Stack Pointer (PSP). Table 1.6 shows the syntax of these Instructions.

Table 1.6: Stack Push and Stack POP Instructions for Core Registers

Example of Stack Operations	Description
PUSH < reg list >	Store register(s) in stack.
POP < reg list >	Restore register(s) from stack.

The register list syntax is the same as LDM and STM. For example:

```
PUSH    {R0 , R4-R7 , R9}        ; PUSH R0, R4, R5, R6, R7, R9 into stack
POP    {R2 , R3}                 ; POP R2 and R3 from stack
```

1.9.3 Arithmetic operations

The most commonly used arithmetic instructions for Cortex-M3 and Cortex-M4, including ADD (addition), SUB (subtraction), MUL (multiplication), and UDIV/SDIV (unsigned and signed division), are shown in Table 1.7.

Table 1.7: Instructions for Arithmetic Data Operations

Commonly Used Arithmetic Instructions		Operation
ADD Rd , Rn , Rm	; Rd = Rn + Rm	ADD operation
ADD Rd , Rn , #immed_8	; Rd = Rn + #immed	
ADC Rd , Rn , Rm	; Rd = Rn + Rm + carry	ADD with carry
ADC Rd , #immed_8	; Rd = Rd + #immed + carry	
ADDW Rd , Rn , #immed	; Rd = Rn + #immed	ADD register with 12-bit immediate value
SUB Rd , Rn , Rm	; Rd = Rn - Rm	SUBTRACT
SUB Rd , #immed_8	; Rd = Rd - #immed	
SUB Rd , Rn , #immed_8	; Rd = Rn - #immed	
SBC Rd , Rn , #immed_8	; Rd = Rn - #immed - borrow	SUBTRACT with borrow
SBC Rd , Rn , Rm	; Rd = Rn - Rm - borrow	
SUBW Rd , Rn , #immed	; Rd = Rn - #immed	SUBTRACT register with 12-bit immediate value
RSB Rd , Rn , #immed_8	; Rd = #immed - Rn	Reverse subtract
RSB Rd , Rn , Rm	; Rd = Rm - Rn	
MUL Rd , Rn , Rm	; Rd = Rn * Rm	Multiply (32-bit result)
UDIV Rd , Rn , Rm	; Rd = Rn / Rm	Unsigned and signed divide
SDIV Rd , Rn , Rm	; Rd = Rn / Rm	

The instructions in Table 1.7 can be used with or without the "S" suffix to specify whether the APSR should be updated or not, as shown in the following example.

```
ADD    R0 , R1 , R2                   ; Flag unchanged
ADDS    R0 , R1 , R2                  ; Flag change
```

Both the Cortex-M3 and Cortex-M4 processors support 32-bit multiply instructions and multiply accumulate (MAC) instructions that give 32-bit and 64-bit results as shown in Table 1.8.

Table 1.8: Instructions for Multiply and MAC (Multiply Accumulate)

Instructions (no "S" suffix, APSR is not updated) and Description
MLA Rd, Rn , Rm , Ra ; Rd = Ra + Rn * Rm 32-bit MAC (Multiply Accumulate) instruction, 32-bit result
MLS Rd , Rn , Rm , Ra ; Rd = Ra - Rn * Rm 32-bit MLS (Multiply and Subtract) instruction, 32-bit result
SMULL RdLo, RdHi, Rn, Rm ; { RdHi , RdLo } = Rn * Rm 32-bit SMULL (Signed Multiply Long) instruction for signed values, 64-bit result
SMLAL RdLo, RdHi, Rn, Rm ; { RdHi , RdLo } += Rn * Rm 32-bit SMLAL (Signed Multiply Accumulate Long) instruction, it adds the 64-bit product to the value stored in registers RdLo and RdHi.
UMULL RdLo, RdHi, Rn, Rm ; { RdHi,RdLo } = Rn * Rm 32-bit (Unsigned Multiply Long) instruction for unsigned values, 64-bit result
UMLAL RdLo, RdHi, Rn, Rm ; { RdHi , RdLo } += Rn * Rm 32-bit UMLAL (Unsigned Multiply Accumulate Long) instruction, 64-bit result.

1.9.4 Logic operations

The commonly used logic operation instructions, such as AND, OR, exclusive OR and so on, are given in Table 1.9. The 16-bit versions of these instructions update the flags in APSR. If the "S" suffix is not specified, the assembler will convert them into 32-bit instructions.

R To use the 16-bit versions of Table 1.9 instructions, the operation must be between two registers with the destination being one of the source registers. Also, the registers used must be low registers (R0-R7), and the "S" suffix should be used (APSR update). The ORN instruction is not available in 16-bit form.

1.9.5 Shift and rotate instructions

As shown in Figure 1.30, the second ALU operand is equipped with Barrel shifter, which is special digital circuit quick shift rotation. The various shift and rotate instructions are shown in Table 1.10 and illustrated in Figure 1.31.

1.9.6 Data conversion operations (extend and reverse ordering)

When a signed integer is converted to another signed integer with more bits, the sign bit (i.e., the most significant bit or the leftmost bit) should be duplicated to maintain the integer's sign. Duplicating the sign bit is called sign extension. When an unsigned integer is converted to another unsigned integer with more bits, zero extension is deployed to

Table 1.9: Instructions for Logical Operations

Instruction (optional S suffix not shown)		Description
AND Rd, Rn	; Rd = Rd & Rn	Bitwise AND
AND Rd , Rn , #immed	; Rd = Rn & #immed	
AND Rd , Rn , Rm	; Rd = Rn & Rm	
ORR Rd , Rn	; Rd = Rd | Rn	Bitwise OR
ORR Rd , Rn , #immed	; Rd = Rn | #immed	
ORR Rd , Rn , Rm	; Rd = Rn | Rm	
BIC Rd , Rn	; Rd = Rd & (~Rn)	Bit clear
BIC Rd , Rn , #immed	; Rd = Rn & (~#immed)	
BIC Rd , Rn , Rm	; Rd = Rn & (~ Rm)	
ORN Rd , Rn , #immed	; Rd = Rn | (~#immed)	Bitwise OR NOT
ORN Rd , Rn , Rm	; Rd = Rn | (~Rm)	
EOR Rd , Rn	; Rd = Rd ^ Rn	Bitwise Exclusive OR
EOR Rd , Rn , #immed	; Rd = Rn ^ #immed	
EOR Rd , Rn , Rm	; Rd = Rn ^ Rm	

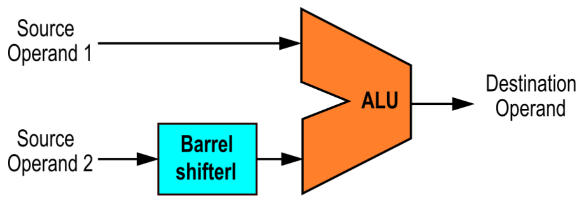


Figure 1.30: Barrel shifter for quick shift rotation

Table 1.10: Instructions for Shift and Rotate Operations

Instruction (optional "S" suffix not shown)		**Description**
LSL Rd, Rn,#immed	; Rd = Rn $<<$ immed	Logical shift left
LSL Rd, Rn	; Rd = Rd $<<$ Rn	
LSL Rd, Rn, Rm	; Rd = Rn $<<$ Rm	
LSR Rd, Rn,#immed	; Rd = Rn $>>$ immed	Logical shift right
LSR Rd, Rn	; Rd = Rd $>>$ Rn	
LSR Rd, Rn, Rm	; Rd = Rn $>>$ Rm	
ASR Rd, Rn,#immed	Rd = Rn $>>$ immed	Arithmetic shift right
ASR Rd, Rn	; Rd = Rd $>>$ Rn	
ASR Rd, Rn, Rm	; Rd = Rn $>>$ Rm	
ROR Rd, Rn	; Rd rot by Rn	Rotate right
ROR Rd, Rn, Rm	; Rd = Rn rot by Rm	
RRX Rd, Rn	; {C, Rd} = {Rn, C}	Rotate right extended

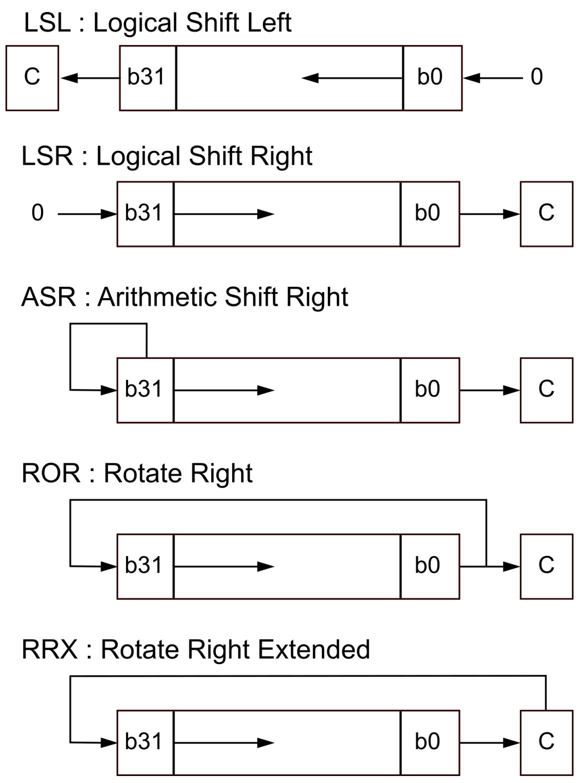

LSL : Logical Shift Left
C
b31
b0
0
LSR : Logical Shift Right
0
b31
b0
C
ASR : Arithmetic Shift Right
b31
b0
C
ROR : Rotate Right
b31
b0
C
RRX : Rotate Right Extended
b31
b0
C

Figure 1.31: Shift and rotate operations.

place zeros in the upper bits of the output. The example blow clarify the principal of this conversion.

```
int_8    a = -1;          // a signed 8-bit integer, a = 0xFF
int_16   b = -2;          // a signed 16-bit integer, b = 0xFFFE
int_32   c;               // a signed 32-bit integer
c = a;                    // sign extension, c = 0xFFFFFFFF
c = b;                    // sign extension, c = 0xFFFFFFFE

uint_8   d = 1;           // an unsigned 8-bit integer, d = 0x01
uint_32  e;               // an unsigned 32-bit integer
e = d;                    // zero extension, e = 0x00000001
```

The following program shows how to use instructions of sign and zero extension. Assume the value of register r0 is 0x11228091.

```
SXTB  r1,  r0     ; r1 = 0xFFFFFF91, sign extend a byte
SXTH  r1,  r0     ; r1 = 0xFFFF8091, sign extend a halfword
UXTB  r1,  r0     ; r1 = 0x00000091, zero extend a byte
UXTH  r1,  r0     ; r1 = 0x00008091, zero extend a halfword
```

Instructions for reversing bits or bytes are often employed to convert data between little-endian and big-endian formats. These instructions are listed in Table 1.11, and the process of reversal is illustrated in Figure 1.32.

Table 1.11: Instructions for Bit and Byte Order Reversal

Instruction	Description
RBIT Rd, Rn	Reverse bit order in a word. *for (i = 0; i < 32; i++) Rd[i] ←Rn[31- i]*
REV Rd, Rn	Reverse byte order in a word. *Rd[31:24]← Rn[7:0], Rd[23:16]←Rn[15:8],* *Rd[15:8] ← Rn[23:16],Rd[7:0] ← Rn[31:24]*
REV16 Rd, Rn	Reverse byte order in each halfword. *Rd[15:8]← Rn[7:0], Rd[7:0]←Rn[15:8],* *Rd[31:24] ← Rn[23:16],Rd[23:16] ← Rn[31:24]*
REVSH Rd, Rn	Reverse byte order in bottom halfword and sign extend. *Rd[15:8]← Rn[7:0], Rd[7:0]←Rn[15:8],* *Rd[31:16] ← Rn[7] & 0xFFFF*

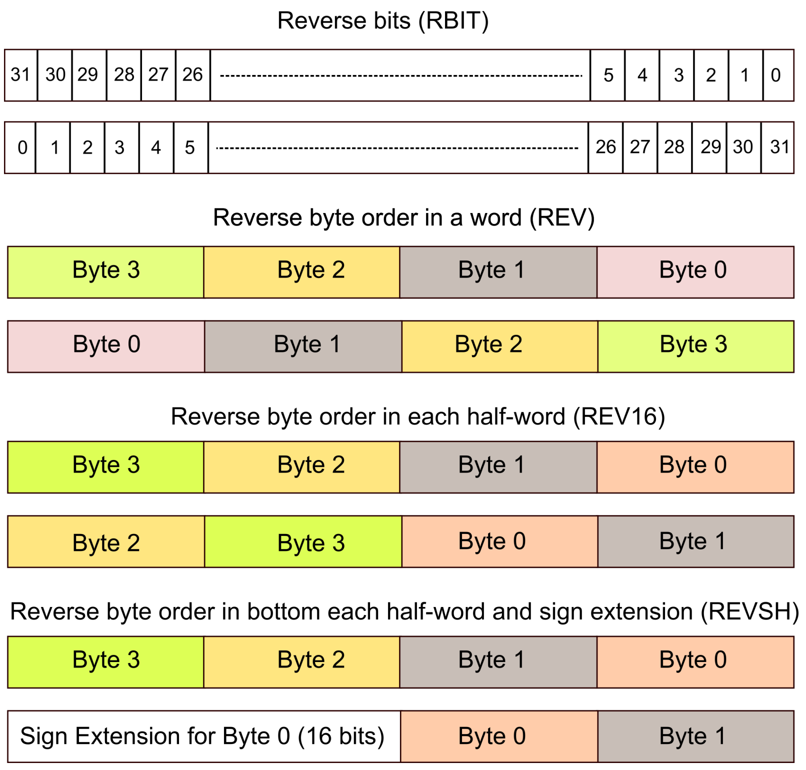

Reverse bits (RBIT)
31 30 29 28 27 26 5 4 3 2 1 0
0 1 2 3 4 5 26 27 28 29 30 31
Reverse byte order in a word (REV)
Byte 3 Byte 2 Byte 1 Byte 0
Byte 0 Byte 1 Byte 2 Byte 3
Reverse byte order in each half-word (REV16)
Byte 3 Byte 2 Byte 1 Byte 0
Byte 2 Byte 3 Byte 0 Byte 1
Reverse byte order in bottom each half-word and sign extension (REVSH)
Byte 3 Byte 2 Byte 1 Byte 0
Sign Extension for Byte 0 (16 bits) Byte 0 Byte 1

Figure 1.32: Reverse bit or byte order

1.9.7 Bit-field processing instructions

To make the Cortex-M3 and Cortex-M4 processor an excellent architecture for control applications, these processors support a number of bit-field processing operations,as listed in Table 1.12.

The following example shows the use of BFC and BFI instructions:

```
LDR   R0, =0x1234FFFF
BFC   R0, #4, #8      ; after execution, R0 will be 0x1234F00F
```

```
LDR   R0, =0x12345678
LDR   R1, =0x1133AACC
BFC   R0, R1, #8, #16     ; after execution, R0 will be 0x115678CC
```

The use of instructions UBFX and SBFX, which extract adjacent bits from one register, are demonstrated in the following example:

```
LDR   R0, =0x7934CADF
UBFX   R1, R0, #4, #8     ; after execution, R1 will be 0x000000AD
```

```
LDR   R0, =0x1278C0AE
SBFX   R1, R0, #4, #8     ; after execution, R1 will be 0xFFFFFF0A
```

Table 1.12: Instructions for Bit-Field Processing

Instruction	Operation
BFC Rd, #<lsb>, #<width>	**B**it **F**ield **C**lear. *Rd[(width+lsb-1):lsb]* $\leftarrow$ *0*
BFI Rd, Rn, #<lsb>, #<width>	**B**it **F**ield **I**nsert. *Rd[(width+lsb-1):lsb]* $\leftarrow$ *Rn[(width-1):0]*
RBIT Rd, Rn	**R**everse **BIT** order in register
UBFX Rd, Rn, #<lsb>, #<width>	**U**nsigned **B**it **F**ield **E**xtract *Rd[(width-1):0]* $\leftarrow$ *Rn[(width+lsb-1):lsb]* *Rd[31:width]*$\leftarrow$ *Replicate(0)*
SBFX Rd, Rn, #<lsb>, #<width>	**S**igned **B**it **F**ield **E**xtract *Rd[(width-1):0]* $\leftarrow$ *Rn[(width+lsb-1):lsb]* *Rd[31:width]*$\leftarrow$ *Replicate(Rn[width+lsb-1])*

1.9.8 Compare and test

The compare and test instructions are used to update the flags in the APSR, which may then be used by a conditional branch or conditional execution. Table 1.13 listed these instructions.

Table 1.13: Instructions for Compare and Test

Instruction	Mnemonic	Operation Explanation
CMP <Rn>, <Rm>;	Compare	Set NZCV flags on *Rn* - *Rm* (result is not stored).
CMP <Rn>, #<immed>	Compare	Set NZCV flags on *Rn* - *immediate data.*
CMN <Rn>, <Rm>;	Compare negative	Set NZCV flags on *Rn* + *Rm.*
CMN <Rn>, #<immed>	Compare negative	Set NZCV flags on *Rn* + *immediate data.*
TST <Rn>, <Rm>;	Test (bitwise AND)	Set NZ/C flags on *Rn* AND *Rm* .
TST <Rn>, #<immed>	Test (bitwise AND)	Set NZ/C flags on *Rn* AND *immediate data* .
TEQ <Rn>, <Rm>;	Test equivalence	Set NZ/C flags on *Rn* EOR *Rm* .
TEQ <Rn>, #<immed>	Test equivalence	Set NZ/C flags on *Rn* EOR *immediate data* .

R

1. The results for the compare and test listed instructions are not stored.
2. For TST and TEQ, they are bitwise instructions and update the N and Z flags only, and the C flag if a barrel shifter is used.

1.9.9 Program flow control

There are several types of instructions for program flow control:

- Branch
- Function call
- Conditional branch
- Combined compare and conditional branch
- Conditional execution (IF-THEN instruction)
- Table branch

Branch

A number of instructions can cause branch operations:

- Branch instructions (e.g., B, BX)
- A data processing instruction that updates R15 (PC) (e.g., MOV, ADD)
- A memory read instruction that writes to PC (e.g., LDR, LDM, POP)

The most basic branch instructions are given in Table 1.14.

Table 1.14: Unconditional Branch Instructions

Instruction	**Operation**
B <label>	Branch to label. If a branch range of over $+/-2$KB is needed,
B.W <label>	you might need to specify B.W to use 32-bit version of branch instruction for wider range.
BX <Rm>	Branch and eXchange. *PC = Rm*

Function calls

To call a function, the Branch and Link (BL) instruction or Branch and Link with eXchange (BLX) instructions can be used (Table 5.32). They execute the branch and at the same time save the return address to the Link Register (LR).

Table 1.15: Instructions for Calling a Function

Instruction	**Description**
BL <label>	Branch with Link. *LR = PC + 4; PC = label*
BLX <Rm>	Branch with Link and eXchange. *LR = PC + 4; PC = Rm*

Notes:

- Since the Cortex-M3 and M4 processors only support the Thumb state, the LSB of the register used in a BLX operation must be set to 1. Otherwise, it indicates an attempt to switch to the ARM state and will result in a fault exception.
- Before using the BL instruction, you should save your LR beforehand if you need its actual value by pushing it to the stack.

Conditional branches

Conditional branches are executed conditionally based on the NZCV flags of the APSR register, which can be affected by the following:

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

The branch condition is indicated by a suffix attached to the B instruction. If the branch range is greater than ±250 bytes, the 32-bit version of the branch through B.W instruction is used, as shown in Table 1.16.

Table 1.16: Instructions for Conditional Branch

Instruction	**Operation**

`B<cond> <label>`
`B<cond>.W <label>`
	Branch to label if condition is true. E.g.,
	CMP R0, #1 ; compare the content of R0 with 1
	BEQ loop ;suffix "EQ" is for equal

By appending the 14th conditional suffixes to the branch instruction "B", Table 1.17 summarizes the resulting conditional branch instructions.

A simple conditional branch example can be illustrated through the program flow depicted in Figure 1.33, which can be implemented using conditional branch and simple branch instructions:

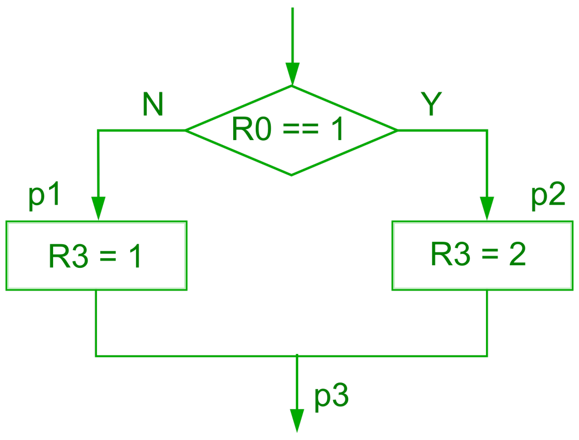


Figure 1.33: Simple condition branch

```
        CMP R0, #1          ; compare R0 to 1
        BEQ p2              ; if Equal, then go to p2
        MOVS R3, #1         ; R3 = 1
        B p3                ; go to p3
        p2                  ; label p2
        MOVS R3, #2         ; R3 = 2
        p3                  ; label p3
```

Table 1.17: Instructions Description and Flags Tested

Instruction	**Description**	**Flags Tested**
`BEQ <label>`	; Branch if **EQ**ual	Z = 1
`BNE <label>`	; Branch if **N**ot **E**qual	Z = 0
`BCS/BHS <label>`	; Branch if Unsigned **H**igher or **S**ame	C = 1
`BCC/BLO <label>`	; Branch if Unsigned **LO**wer	C = 0
`BMI <label>`	; Branch if **MI**nus (Negative)	N = 1
`BPL <label>`	; Branch if **PL**us (Positive or Zero)	N = 0
`BVS <label>`	; Branch if o**V**erflow **S**et	V = 1
`BVC <label>`	; Branch if o**V**erflow **C**lear	V = 0
`BHI <label>`	; Branch if Unsigned **HI**gher	C = 1 & Z = 0
`BLS <label>`	; Branch if Unsigned **L**ower or **S**ame	C = 0 or Z = 1
`BGE <label>`	; Branch if Signed **G**reater or **E**qual	N = V
`BLT <label>`	; Branch if Signed **L**ess **T**han	N != V
`BGT <label>`	; Branch if Signed Greater Than	Z = 0 & N = V
`BLE <label>`	; Branch if Signed **L**ess than or **E**qual	Z = 1 or N != V

```
        .               ; other subsequent operations
```

Another example showing the comparison of two signed integers 0xFFFFFFFF and 0x00000001 through the implementation of an *if*-statement in C and assembly language is shown in Table 1.18. .

Compare and branches

It is common for an assembly program to compare against zero before branching. So, the ARMv7-M architecture provides two compare and branch instructions, **CBZ** (**C**ompare and **B**ranch if **Z**ero) and **CBNZ** (**C**ompare and **B**ranch if **N**on **Z**ero). CBZ and CBNZ are very useful in loop structures such as while loops. Table 1.19 shows how to use CBZ to implement *while*-statement.

Conditional execution (IF-THEN instruction)

Besides conditional branches, Cortex-M3 and Cortex-M4 processors also support conditional execution using the IT instruction, which forms the IT block. The syntax of this instruction comes with three optional suffixes of "T" (then) and "E" (else), noted by (x, y ,z) in the instruction syntax:

```
        IT{x{y{z}}} cond
```

Where:

Table 1.18: Implementation of *if*-statement that compares two signed integers

C Program	Assembly Program

`signed int x, y, z;`
`x = 1; // example`
`y = -1; // 0xFFFFFFFF`
`if (x > y)`
`z = 1;`
`else`
`z = 0;`
	`MOVS R5, #0x00000001 ; R5 - x`
	`MOVS R6, #0xFFFFFFFF ; R6 - y`
	`CMP R5, R6`
	`BLE then ; branch if signed`
	`MOVS R7, #1 ; z = 1`
	`B endif ; skip next instruction`
	`then MOVS R7, #0 ; z = 0`
	`endif`

Table 1.19: Implementing *while*-statement using CBZ instruction

C Program	Assembly Program

`i = 5;`
`while (i != 0) {`
`func1(); //call func1`
`i--;`
`}`
	`      MOV R0, #5     ; Set loop counter`
	`loop1 CBZ R0, loop1exit   ; exit when counter = 0`
	`      BL func1       ; call a function`
	`      SUBS R0, #1    ; loop counter decrement`
	`      B loop1        ; next loop`
	`loope1exit`

x, y, z: specify the condition switches for the second, third, and fourth instructions in the IT block, respectively. They can be either "T" (then) suffix that applies the condition "cond" to the instruction or "E" (else) suffix that applies the inverse condition of "cond" to the instruction.

Different combinations of "T" and "E" sequence are possible:

- Just one conditional execution instruction: IT
- Two conditional execution instructions: ITT, ITE
- Three conditional execution instructions: ITTT, ITTE, ITET, ITEE
- Four conditional execution instructions: ITTTT, ITTTE, ITTET, ITTEE, ITETT, ITETE, ITEET, ITEEE

Table 1.20 listed various forms of IT instruction block sequence and examples:

R In some assembler tools (e.g. Keil MDK-ARM), the assembler can automatically insert the required IT instruction, as shown in Table 1.21.

Table branches

The Cortex-M3 and Cortex-M4 support two table branch instructions: TBB (Table Branch Byte) and TBH (Table Branch Half-word), which are often used to implement switch

Table 1.20: IT Instruction Block of Various Sizes

IT block	**IT block syntax**	**Example**

Only one conditional instruction	`IT <cond>`
	`instr1<cond>`
		`IT EQ`
		`ADDEQ R0, R0, R1`

Two conditional instructions	`IT<x> <cond>`
	`instr1<cond>`
	`instr2<cond or w(cond)>`
		`ITE GE`
		`ADDGE R0, R0, R1`
		`ADDLT R0, R0, R3`

Three conditional instructions	`IT<x><y> <cond>`
	`instr1<cond>`
	`instr2<cond or w(cond)>`
	`instr3<cond or w(cond)>`
		`ITET GT`
		`ADDGT R0, R0, R1`
		`ADDLE R0, R0, R3`
		`ADDGT R2, R4, #1`

Four conditional instructions	`IT<x><y><z> <cond>`
	`instr1<cond>`
	`instr2<cond or w(cond)>`
	`instr3<cond or w(cond)>`
	`instr4<cond or w(cond)>`
		`ITETT NE`
		`ADDNE R0, R0, R1`
		`ADDEQ R0, R0, R3`
		`ADDNE R2, R4, #1`
		`MOVNE R5, R3`

statements in C code. TBB causes single-byte offsets, while TBH causes half-word offsets. Their syntax is as follows:

```
TBB [Rn, Rm]
TBH [Rn, Rm, LSL #1]
```

Where:

`Rn`: stores the base address of the branch table. If `Rn` is the Program Counter (PC) register, then the address of the table is the address of the byte immediately following the `TBB` or `TBH` instruction.

`Rm`: Is the branch table index. For halfword tables, `LSL #1` doubles the value in `Rm` to form the right offset into the table.

Exmaple 01

```
ADR.W R0, BranchTable_Byte
TBB [R0, R1] ; R1 is the index, R0 is the base address of the branch table.

Case1
; an instruction sequence follows
Case2
; an instruction sequence follows
Case3
; an instruction sequence follows
```

Table 1.21: Automatic Insertion of IT Instruction in ARM Assembler

Original Assembler Code	**Disassembled Assembly Code from Generated Object File**

`...`
`CMP R1, #2`
`ADDEQ R0, R1, #1`
`...`
	`...`
	`CMP R1, #2`
	`IT EQ`
	`ADDEQ R0, R1, #1`
	`...`

```
BranchTable_Byte
DCB 0 ; Case1 offset calculation.
DCB ((Case2-Case1)/2) ; Case2 offset calculation.
DCB ((Case3-Case1)/2) ; Case3 offset calculation.
```

Exmaple 02

The following assembly code uses TBB to implement a switch statement, which converts a numeric score to its corresponding letter grade.

Table 1.22: Converting Score to letter grade

C Program

```c
unsigned int score;
char grade;

switch(score) {
case 10:
grade = ’A’;
break;
case 9:
grade = ’B’;
break;
case 8:
grade = ’C’;
break;
case 7:
grade = ’D’;
break;
case 6:
grade = ’E’;
break;
default:
grade = ’F’;
}
```

Assembly Program

```
; R0 = numeric score ( 0 <= r0 <= 10 )
; R1 = Letter grade
SUBS R2, R0, #6   ; R2 is branch index
CMP  R2, #5
BHS  default  ; branch if unsigned R2 >= 5
; R2 is the index;
; PC = PC + 4 + 2 x BranchTable[R2]
TBB [PC, R2]   ; Table Branch Byte

BranchTable

DCB (case_6 - BranchTable)/2  ; index = 0
DCB (case_7 - BranchTable)/2  ; index = 1
DCB (case_8 - BranchTable)/2  ; index = 2
DCB (case_9 - BranchTable)/2 ; index = 3
DCB (case_10 - BranchTable)/2 ; index = 4

ALIGN

case_10
MOV R1, #0x41   ; ASCII ’A’ = 0x41
B exit

case_9:
MOV R1, #0x42   ; ASCII ’B’ = 0x42
B exit

case_8:
MOV R1, #0x43   ; ASCII ’C’ = 0x43
B exit

case_7:
MOV R1, #0x43   ; ASCII ’D’ = 0x44
B exit

case_6:
MOV R1, #0x43   ; ASCII ’E’ = 0x45
B exit

default:
MOV R1, #0x43   ; ASCII ’F’ = 0x46
B exit

exit
```