EMBEDDED SYSTEMS (18EC62)

MODULE – 2

ARM Cortex-M3 Instruction Set and Programming

Assembly Basics

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, you will find the opcode (the instruction) followed by a number of operands.

Assembler Language: Basic Syntax (continued)

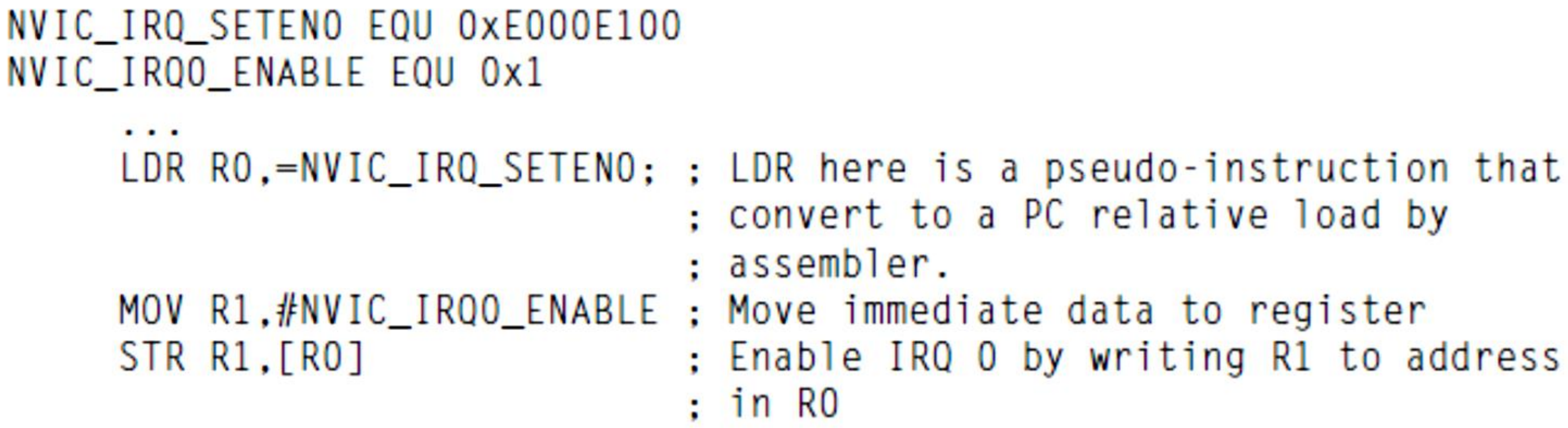
- Normally, the first operand is the destination of the operation.
- The number of operands in an instruction depends on the type of instruction, and the syntax format of the operand can also be different.
    - For example, immediate data are usually in the form *#number*, as shown here:

```
MOV R0, #0x12 ; Set R0 = 0x12 (hexadecimal)
MOV R1, #'A'  ; Set R1 = ASCII character A
```

- The text after each semicolon (;) is a comment.
    - These comments do not affect the program operation, but they can make programs easier for humans to understand.

Assembler Language: Basic Syntax (continued)

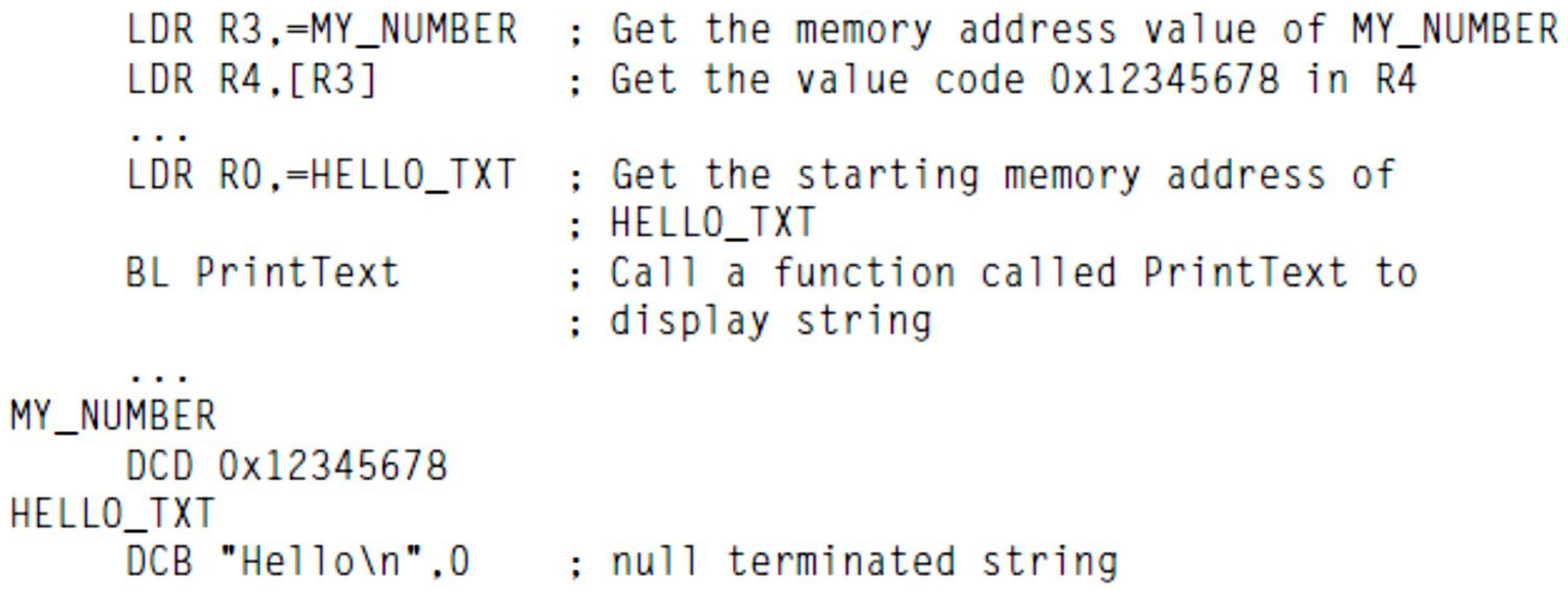
- Constants can be defined using EQU directive, and then they can be used in the program.



```
NVIC_IRQ_SETEN0 EQU 0xE000E100
NVIC_IRQ0_ENABLE EQU 0x1
     ...
     LDR R0,=NVIC_IRQ_SETEN0;  ; LDR here is a pseudo-instruction that
                               ; convert to a PC relative load by
                               ; assembler.
     MOV R1,#NVIC_IRQ0_ENABLE  ; Move immediate data to register
     STR R1,[R0]               ; Enable IRQ 0 by writing R1 to address
                               ; in R0
```

Assembler Language: Basic Syntax (continued)

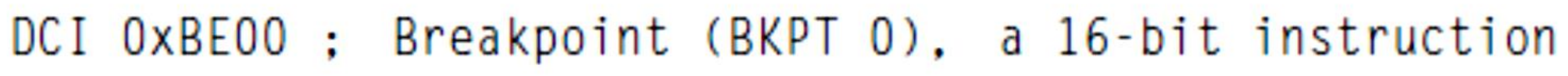
- DCB (Define Constant Byte) can be used for byte size constant values, such as characters, and Define Constant Data (DCD) for word size constant values to define binary data.



```
        LDR R3,=MY_NUMBER   ; Get the memory address value of MY_NUMBER
        LDR R4,[R3]         ; Get the value code 0x12345678 in R4
        ...
        LDR R0,=HELLO_TXT   ; Get the starting memory address of
                            ; HELLO_TXT
        BL PrintText        ; Call a function called PrintText to
                            ; display string
        ...
MY_NUMBER
        DCD 0x12345678
HELLO_TXT
        DCB "Hello\n",0     ; null terminated string
```

Assembler Language: Basic Syntax (continued)

- A number of data definition directives are available for insertion of constants inside assembly code.
  - For example, DCI (Define Constant Instruction) can be used to code an instruction if the assembler cannot generate the exact instruction that you want and if you know the binary code for the instruction.



```
DCI 0xBE00 ;  Breakpoint (BKPT 0), a 16-bit instruction
```

Assembler Language: Use of Suffixes

- In assembler for ARM processors, instructions can be followed by suffixes, as shown in Table 4.1.

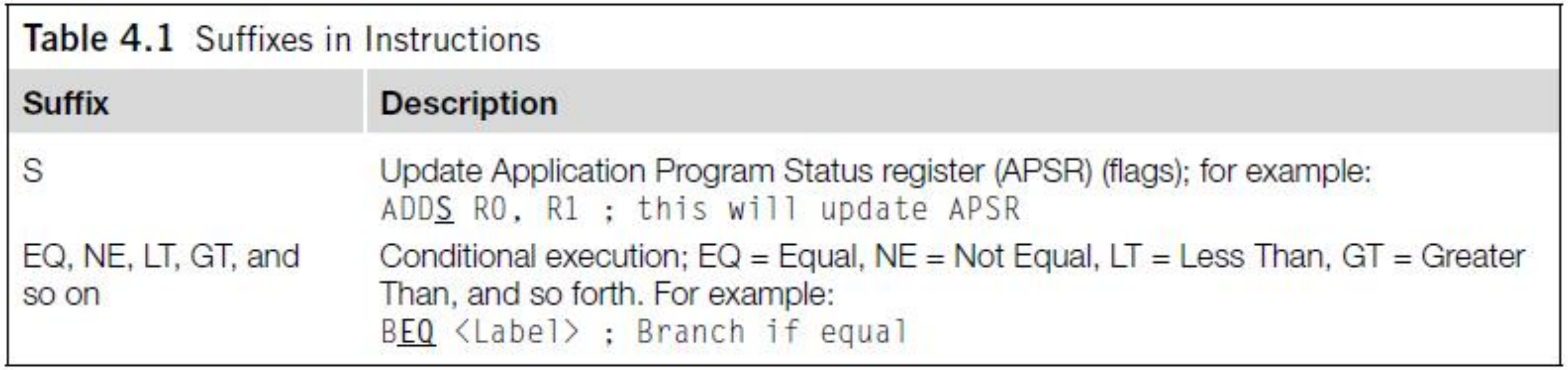


Table 4.1 Suffixes in Instructions

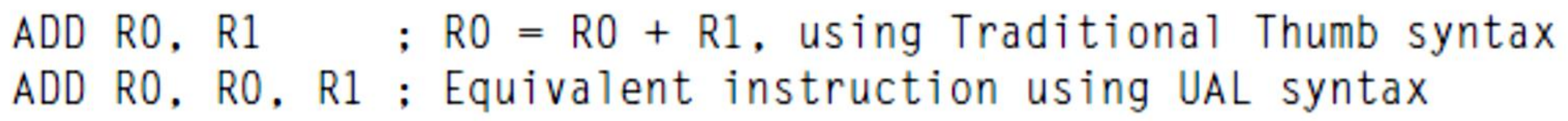
Suffix	**Description**
S	Update Application Program Status register (APSR) (flags); for example: `ADDS R0, R1 ; this will update APSR`
EQ, NE, LT, GT, and so on	Conditional execution; EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, and so forth. For example: `BEQ <Label> ; Branch if equal`

Assembler Language: Use of Suffixes (continued)

- For the 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.
  - In those cases, the *S* suffix and the conditional execution suffixes can be used at the same time.

Assembler Language: Unified Assembler Language

- The Unified Assembler Language (UAL) was developed to allow selection of 16-bit and 32-bit instructions.
  - Supports and gets the best out of the Thumb-2 instruction set.
- With UAL, the syntax of Thumb instructions is now the same as for ARM instructions.
  - Makes it easier to port applications between ARM code and Thumb code by using the same syntax for both.



```
ADD R0, R1      ; R0 = R0 + R1, using Traditional Thumb syntax
ADD R0, R0, R1 ; Equivalent instruction using UAL syntax
```

Assembler Language: Unified Assembler Language (continued)

- The traditional Thumb syntax can still be used.
- The choice between whether the instructions are interpreted as traditional Thumb code or the new UAL syntax is normally defined by the directive in the assembly file.
  - For example, with ARM assembler tool, a program code header with “CODE16” directive implies the code is in the traditional Thumb syntax, and “THUMB” directive implies the code is in the new UAL syntax.

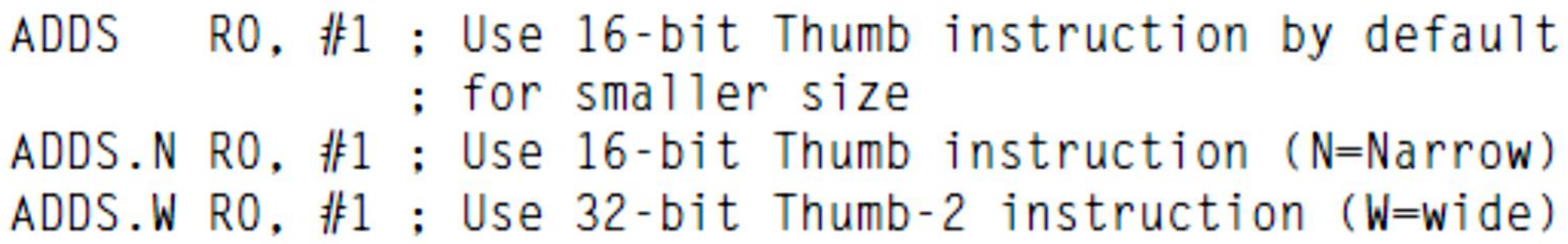
Assembler Language: Unified Assembler Language (continued)

- One thing we need to be careful with reusing traditional Thumb is that some instructions change the flags in APSR, even if the *S* suffix is not used.
- However, when the UAL syntax is used, whether the instruction changes the flag depends on the *S* suffix.
  - For example,

```
AND  R0, R1     ; Traditional Thumb syntax
ANDS R0, R0, R1 ; Equivalent UAL syntax (S suffix is added)
```

Assembler Language: Unified Assembler Language (continued)

- With the new instructions in Thumb-2 technology, some of the operations can be handled by either a Thumb instruction or a Thumb-2 instruction.
  - For example, R0 = R0 + 1 can be implemented as a 16-bit Thumb instruction or a 32-bit Thumb-2 instruction.
- With UAL, you can specify which instruction you want by adding suffixes:



```
ADDS   R0, #1 ; Use 16-bit Thumb instruction by default
              ; for smaller size
ADDS.N R0, #1 ; Use 16-bit Thumb instruction (N=Narrow)
ADDS.W R0, #1 ; Use 32-bit Thumb-2 instruction (W=wide)
```

Assembler Language: Unified Assembler Language (continued)

- The .W (wide) suffix specifies a 32-bit instruction.
  - If no suffix is given, the assembler tool can choose either instruction but usually defaults to 16-bit Thumb code to get a smaller size.
- Depending on tool support, you may also use the .N (narrow) suffix to specify a 16-bit Thumb instruction.
- In most cases, applications will be coded in C, and the C compilers will use 16-bit instructions if possible due to smaller code size.
  - However, when the immediate data exceed a certain range or when the operation can be better handled with a 32-bit Thumb-2 instruction, the 32-bit instruction will be used.

Assembler Language: Unified Assembler Language (continued)

- The 32-bit Thumb-2 instructions can be half word aligned.
  - For example, you can have a 32-bit instruction located in a half word location.

```
0x1000 : LDR r0,[r1] ;a 16-bit instructions (occupy 0x1000-0x1001)
0x1002 : RBIT.W r0   ;a 32-bit Thumb-2 instruction (occupy
                     ;  0x1002-0x1005)
```

- Most of the 16-bit instructions can only access registers R0–R7.
  - 32-bit Thumb-2 instructions do not have this limitation.
  - However, use of PC (R15) might not be allowed in some of the instructions.

Instruction List

16-Bit Data Processing Instructions

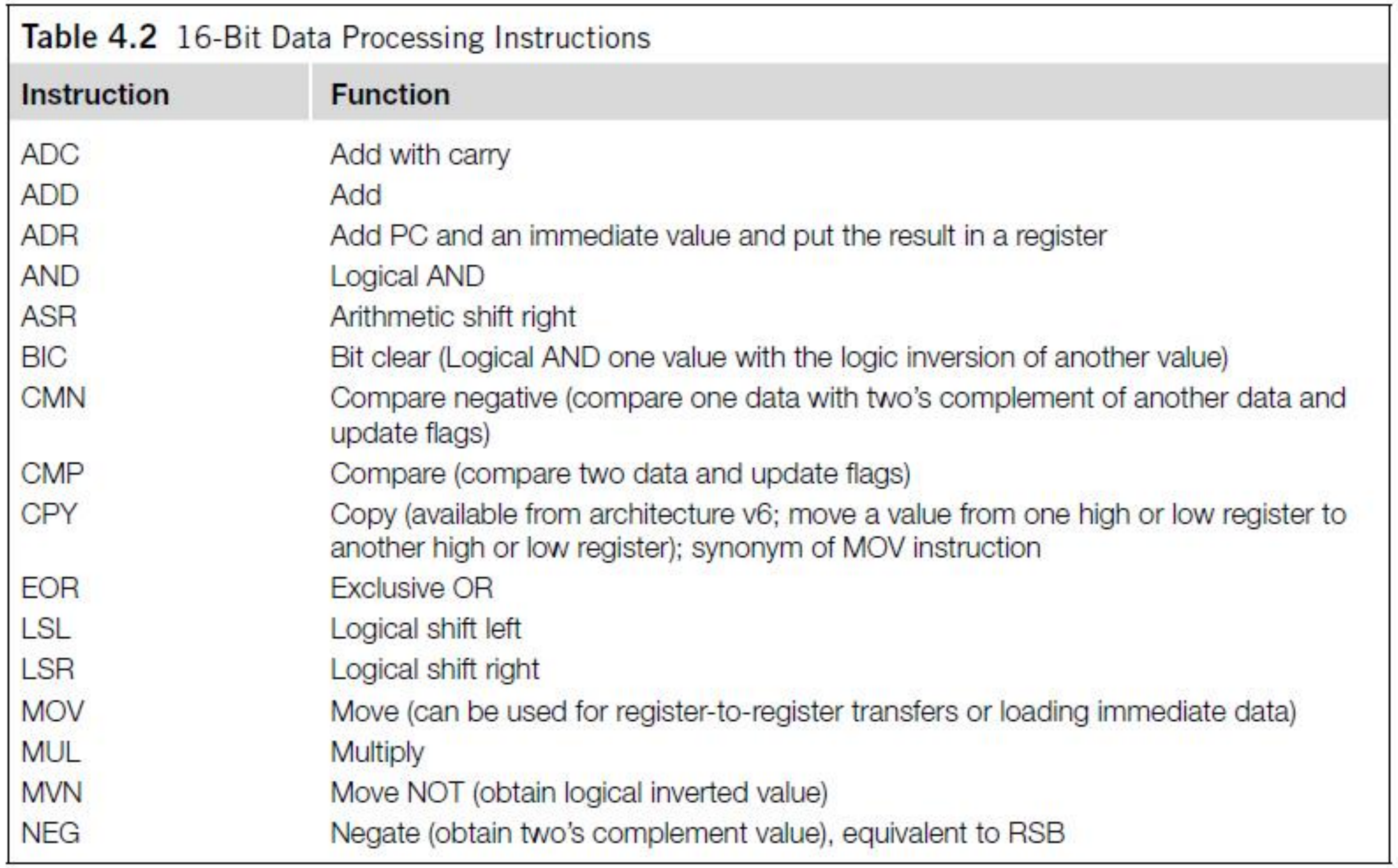


Table 4.2 16-Bit Data Processing Instructions

Instruction	Function
ADC	Add with carry
ADD	Add
ADR	Add PC and an immediate value and put the result in a register
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (Logical AND one value with the logic inversion of another value)
CMN	Compare negative (compare one data with two's complement of another data and update flags)
CMP	Compare (compare two data and update flags)
CPY	Copy (available from architecture v6; move a value from one high or low register to another high or low register); synonym of MOV instruction
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MOV	Move (can be used for register-to-register transfers or loading immediate data)
MUL	Multiply
MVN	Move NOT (obtain logical inverted value)
NEG	Negate (obtain two's complement value), equivalent to RSB

16-Bit Data Processing Instructions (continued)

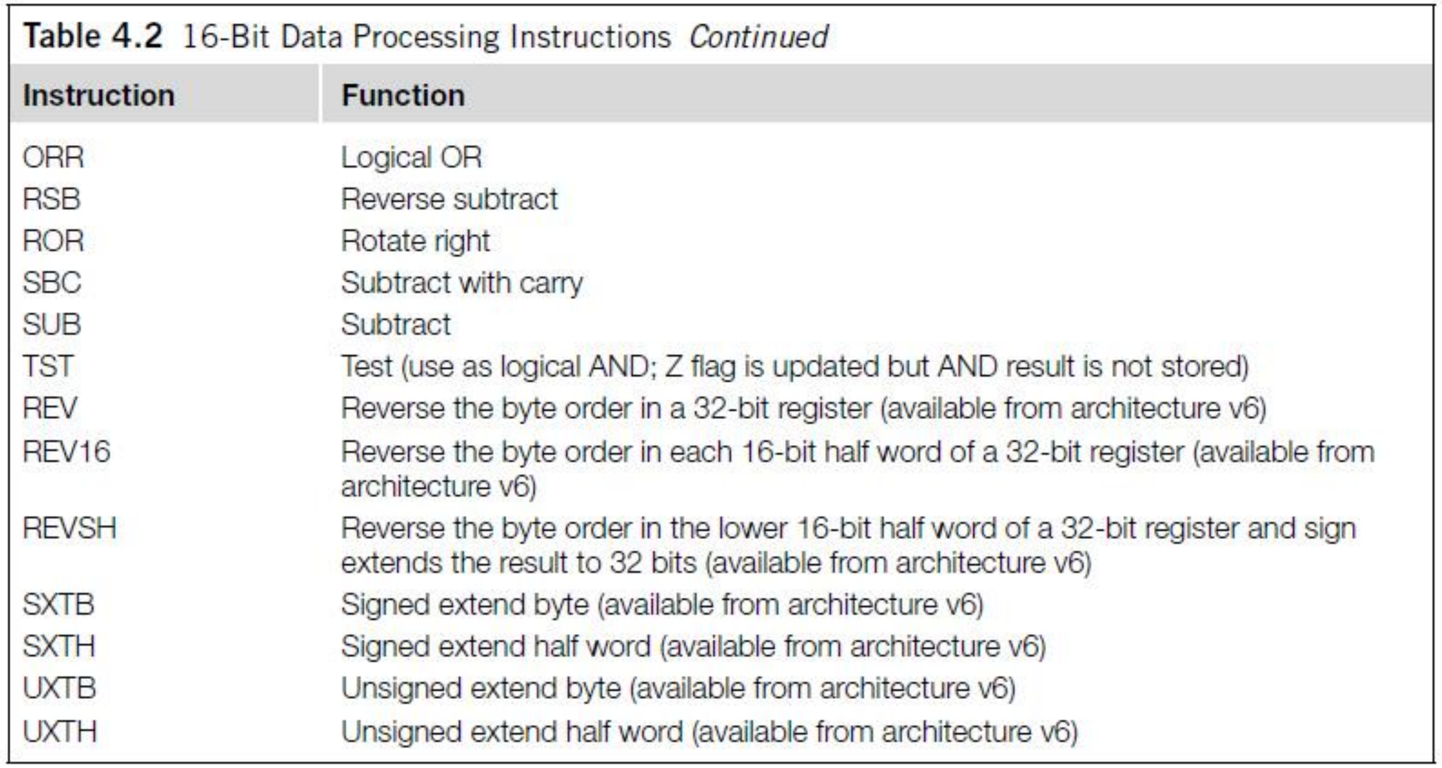


Table 4.2 16-Bit Data Processing Instructions *Continued*

Instruction	Function
ORR	Logical OR
RSB	Reverse subtract
ROR	Rotate right
SBC	Subtract with carry
SUB	Subtract
TST	Test (use as logical AND; Z flag is updated but AND result is not stored)
REV	Reverse the byte order in a 32-bit register (available from architecture v6)
REV16	Reverse the byte order in each 16-bit half word of a 32-bit register (available from architecture v6)
REVSH	Reverse the byte order in the lower 16-bit half word of a 32-bit register and sign extends the result to 32 bits (available from architecture v6)
SXTB	Signed extend byte (available from architecture v6)
SXTH	Signed extend half word (available from architecture v6)
UXTB	Unsigned extend byte (available from architecture v6)
UXTH	Unsigned extend half word (available from architecture v6)

16-Bit Branch Instructions

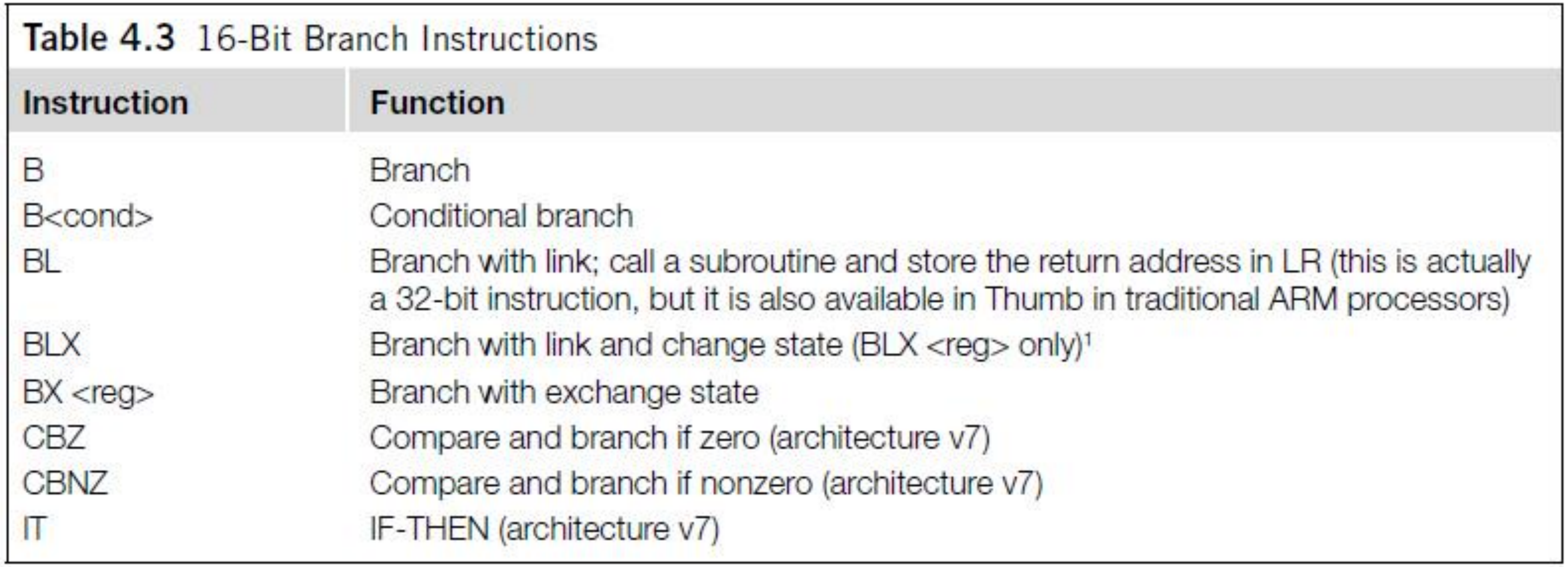


Table 4.3 16-Bit Branch Instructions

Instruction	Function
B	Branch
B<cond>	Conditional branch
BL	Branch with link; call a subroutine and store the return address in LR (this is actually a 32-bit instruction, but it is also available in Thumb in traditional ARM processors)
BLX	Branch with link and change state (BLX <reg> only)[1]
BX <reg>	Branch with exchange state
CBZ	Compare and branch if zero (architecture v7)
CBNZ	Compare and branch if nonzero (architecture v7)
IT	IF-THEN (architecture v7)

16-Bit Load and Store Instructions

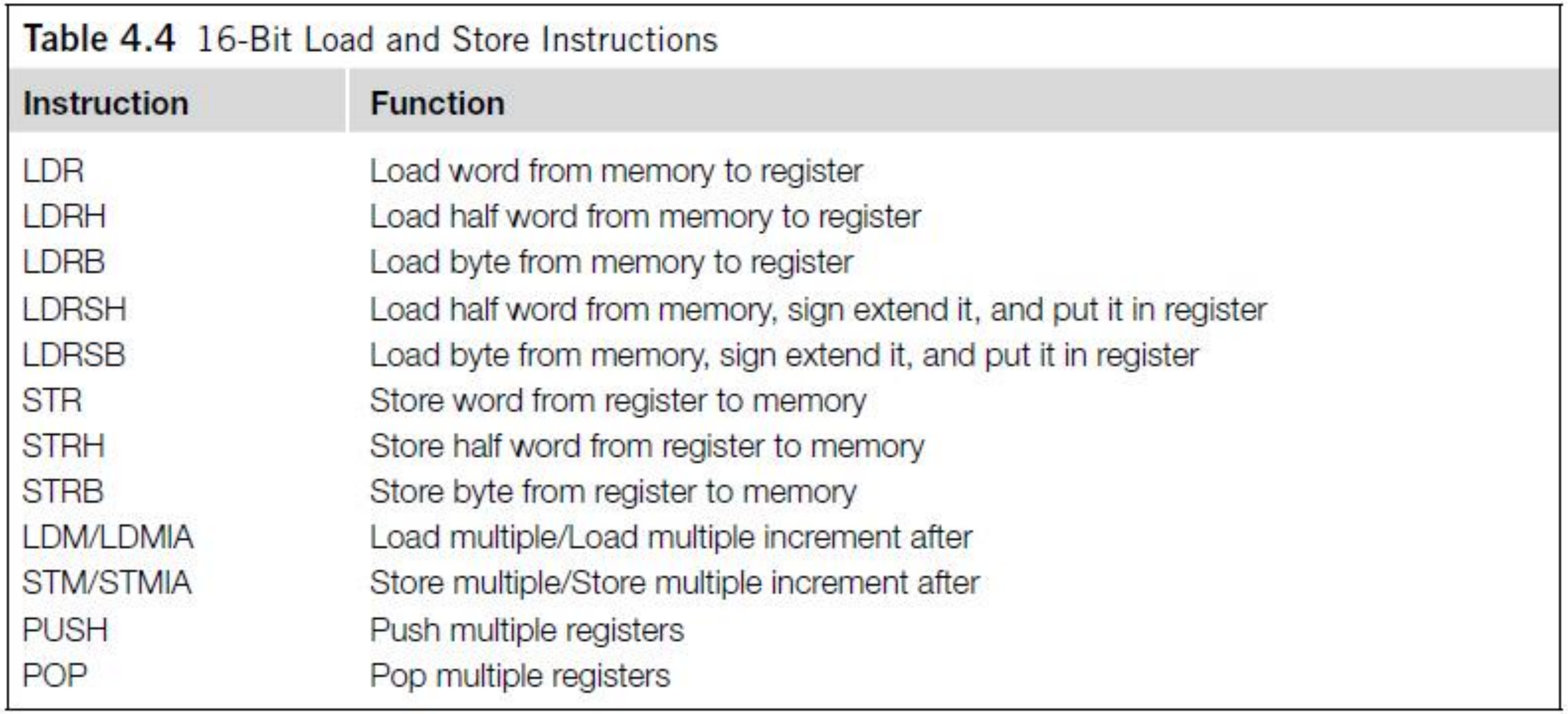


Table 4.4 16-Bit Load and Store Instructions

Instruction	Function
LDR	Load word from memory to register
LDRH	Load half word from memory to register
LDRB	Load byte from memory to register
LDRSH	Load half word from memory, sign extend it, and put it in register
LDRSB	Load byte from memory, sign extend it, and put it in register
STR	Store word from register to memory
STRH	Store half word from register to memory
STRB	Store byte from register to memory
LDM/LDMIA	Load multiple/Load multiple increment after
STM/STMIA	Store multiple/Store multiple increment after
PUSH	Push multiple registers
POP	Pop multiple registers

Other 16-Bit Instructions

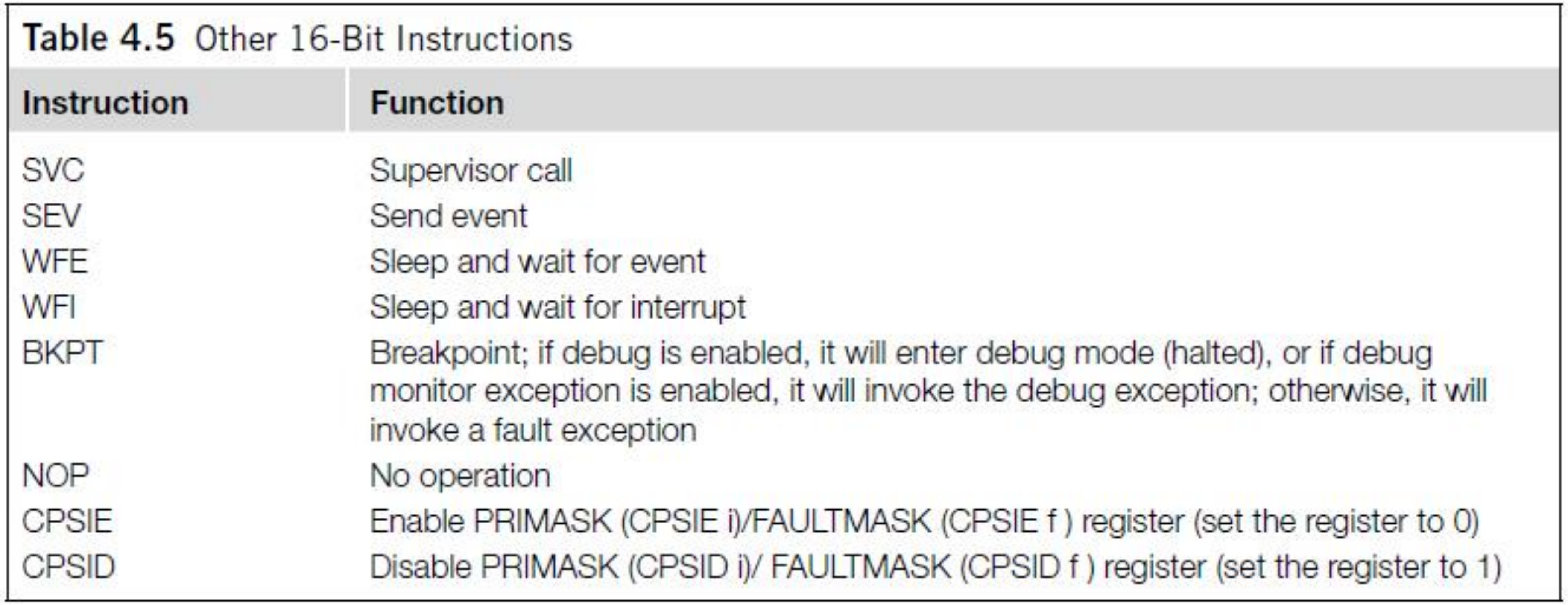


Table 4.5 Other 16-Bit Instructions

Instruction	Function
SVC	Supervisor call
SEV	Send event
WFE	Sleep and wait for event
WFI	Sleep and wait for interrupt
BKPT	Breakpoint; if debug is enabled, it will enter debug mode (halted), or if debug monitor exception is enabled, it will invoke the debug exception; otherwise, it will invoke a fault exception
NOP	No operation
CPSIE	Enable PRIMASK (CPSIE i)/FAULTMASK (CPSIE f) register (set the register to 0)
CPSID	Disable PRIMASK (CPSID i)/ FAULTMASK (CPSID f) register (set the register to 1)

32-Bit Data Processing Instructions

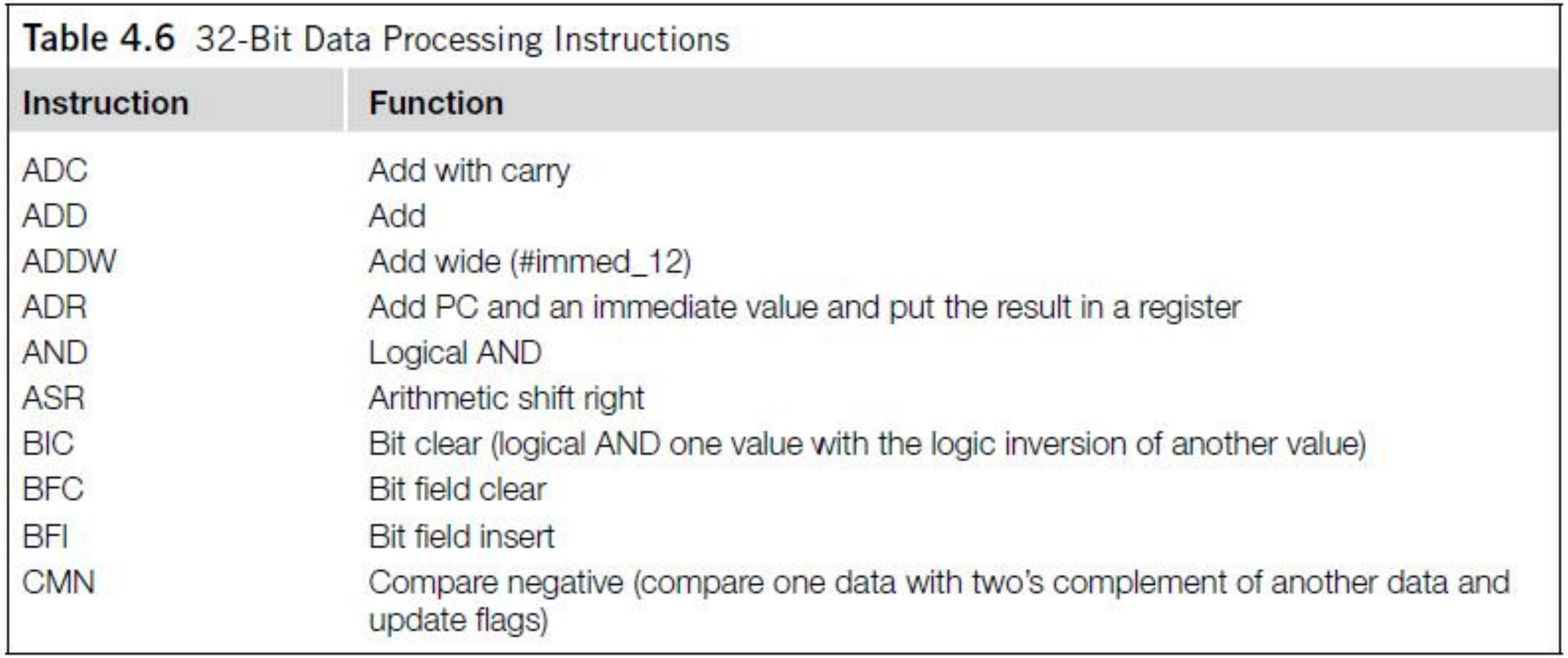


Table 4.6 32-Bit Data Processing Instructions

Instruction	Function
ADC	Add with carry
ADD	Add
ADDW	Add wide (#immed_12)
ADR	Add PC and an immediate value and put the result in a register
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (logical AND one value with the logic inversion of another value)
BFC	Bit field clear
BFI	Bit field insert
CMN	Compare negative (compare one data with two's complement of another data and update flags)

32-Bit Data Processing Instructions (continued)

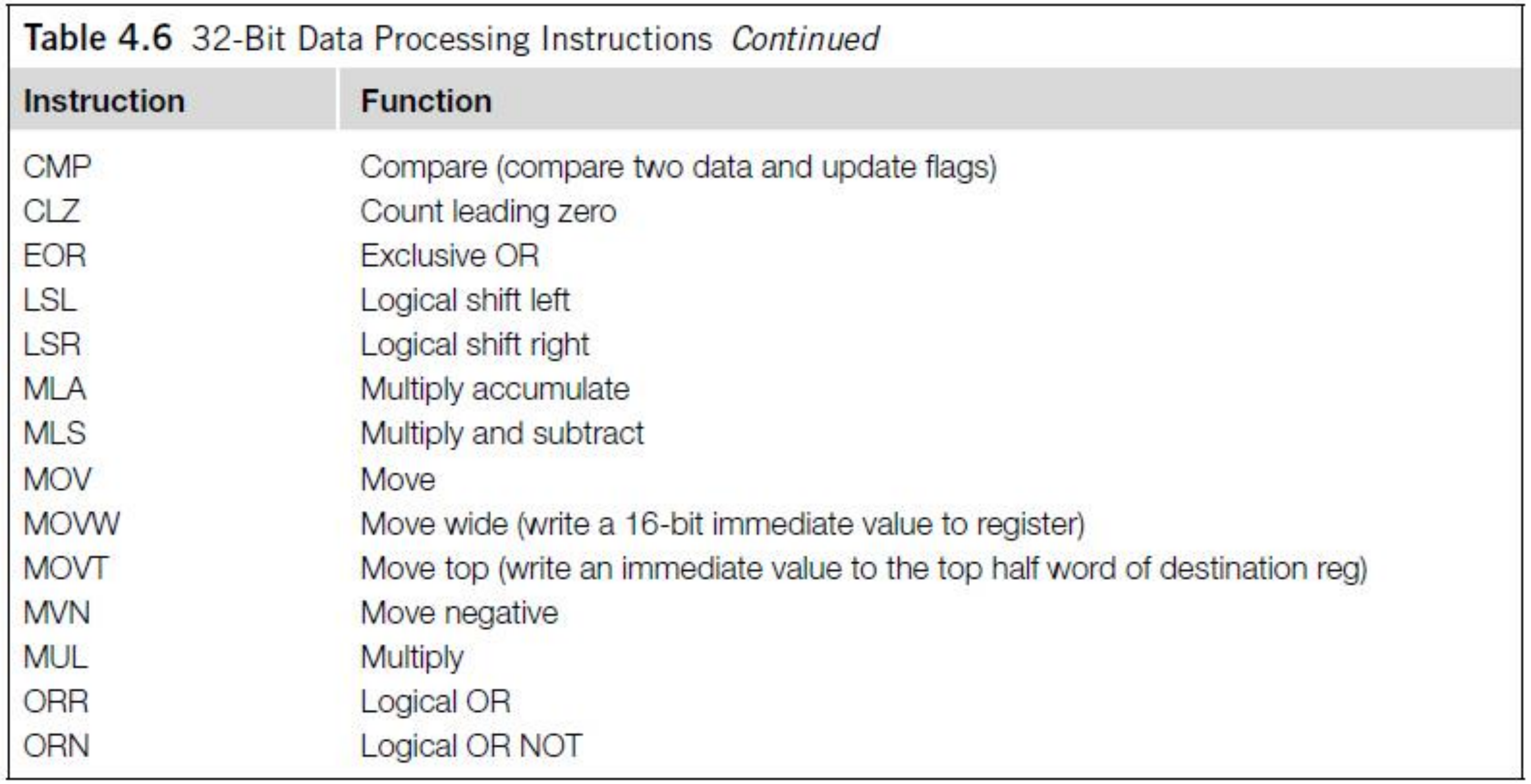


Table 4.6 32-Bit Data Processing Instructions *Continued*

Instruction	Function
CMP	Compare (compare two data and update flags)
CLZ	Count leading zero
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MLA	Multiply accumulate
MLS	Multiply and subtract
MOV	Move
MOVW	Move wide (write a 16-bit immediate value to register)
MOVT	Move top (write an immediate value to the top half word of destination reg)
MVN	Move negative
MUL	Multiply
ORR	Logical OR
ORN	Logical OR NOT

32-Bit Data Processing Instructions (continued)

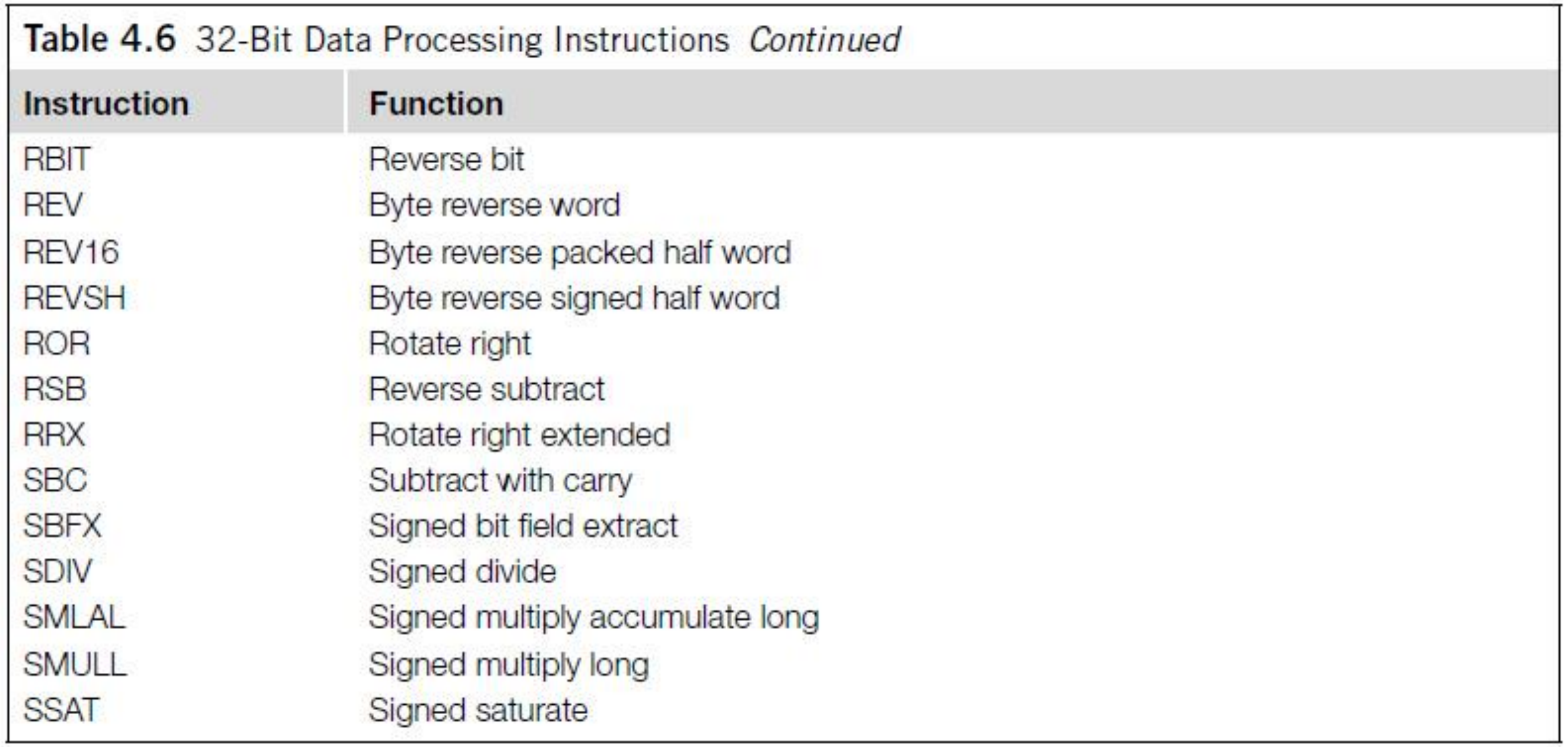


Table 4.6 32-Bit Data Processing Instructions *Continued*

Instruction	Function
RBIT	Reverse bit
REV	Byte reverse word
REV16	Byte reverse packed half word
REVSH	Byte reverse signed half word
ROR	Rotate right
RSB	Reverse subtract
RRX	Rotate right extended
SBC	Subtract with carry
SBFX	Signed bit field extract
SDIV	Signed divide
SMLAL	Signed multiply accumulate long
SMULL	Signed multiply long
SSAT	Signed saturate

32-Bit Data Processing Instructions (continued)

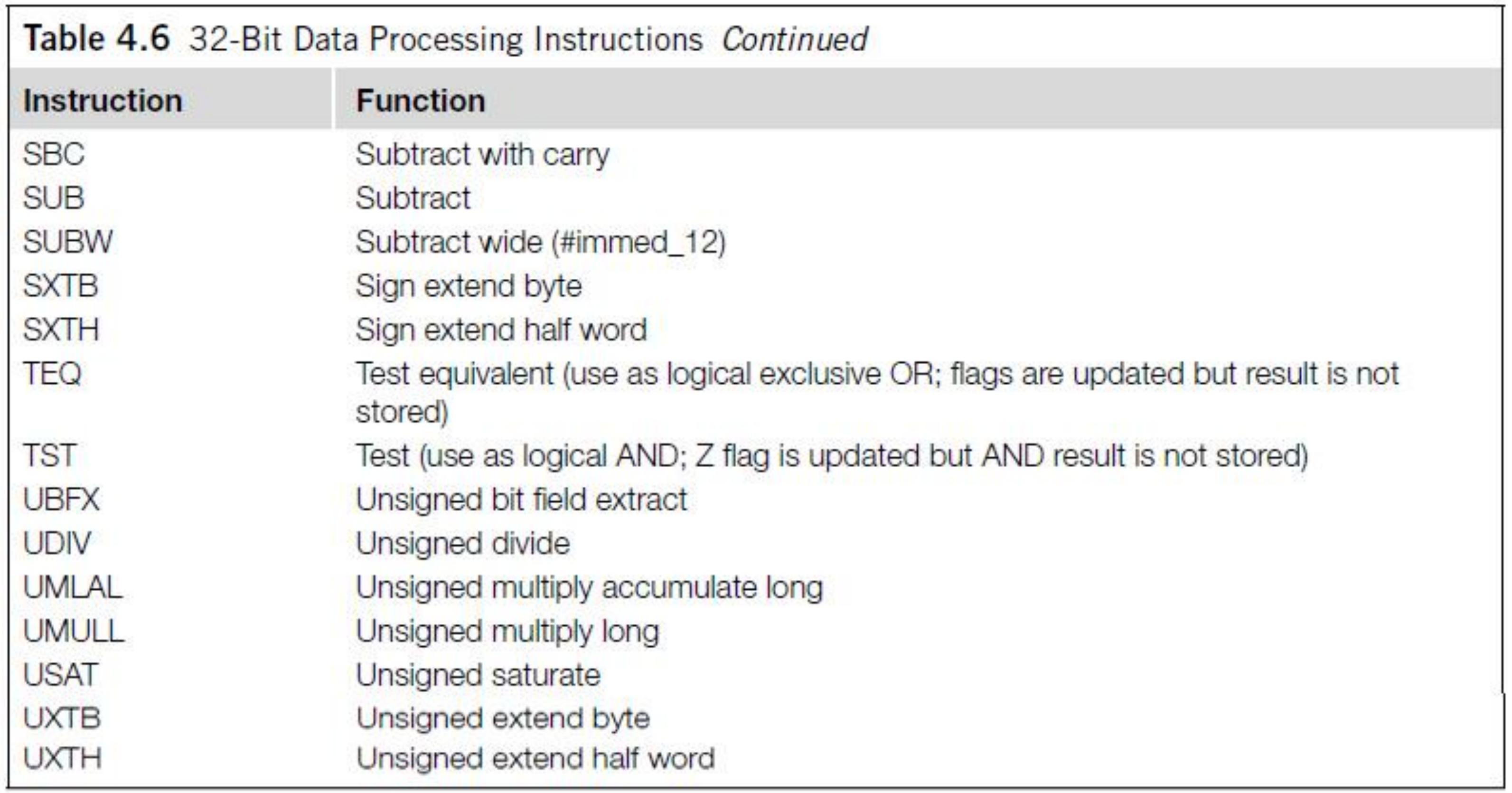


Table 4.6 32-Bit Data Processing Instructions *Continued*

Instruction	Function
SBC	Subtract with carry
SUB	Subtract
SUBW	Subtract wide (#immed_12)
SXTB	Sign extend byte
SXTH	Sign extend half word
TEQ	Test equivalent (use as logical exclusive OR; flags are updated but result is not stored)
TST	Test (use as logical AND; Z flag is updated but AND result is not stored)
UBFX	Unsigned bit field extract
UDIV	Unsigned divide
UMLAL	Unsigned multiply accumulate long
UMULL	Unsigned multiply long
USAT	Unsigned saturate
UXTB	Unsigned extend byte
UXTH	Unsigned extend half word

32-Bit Load and Store Instructions

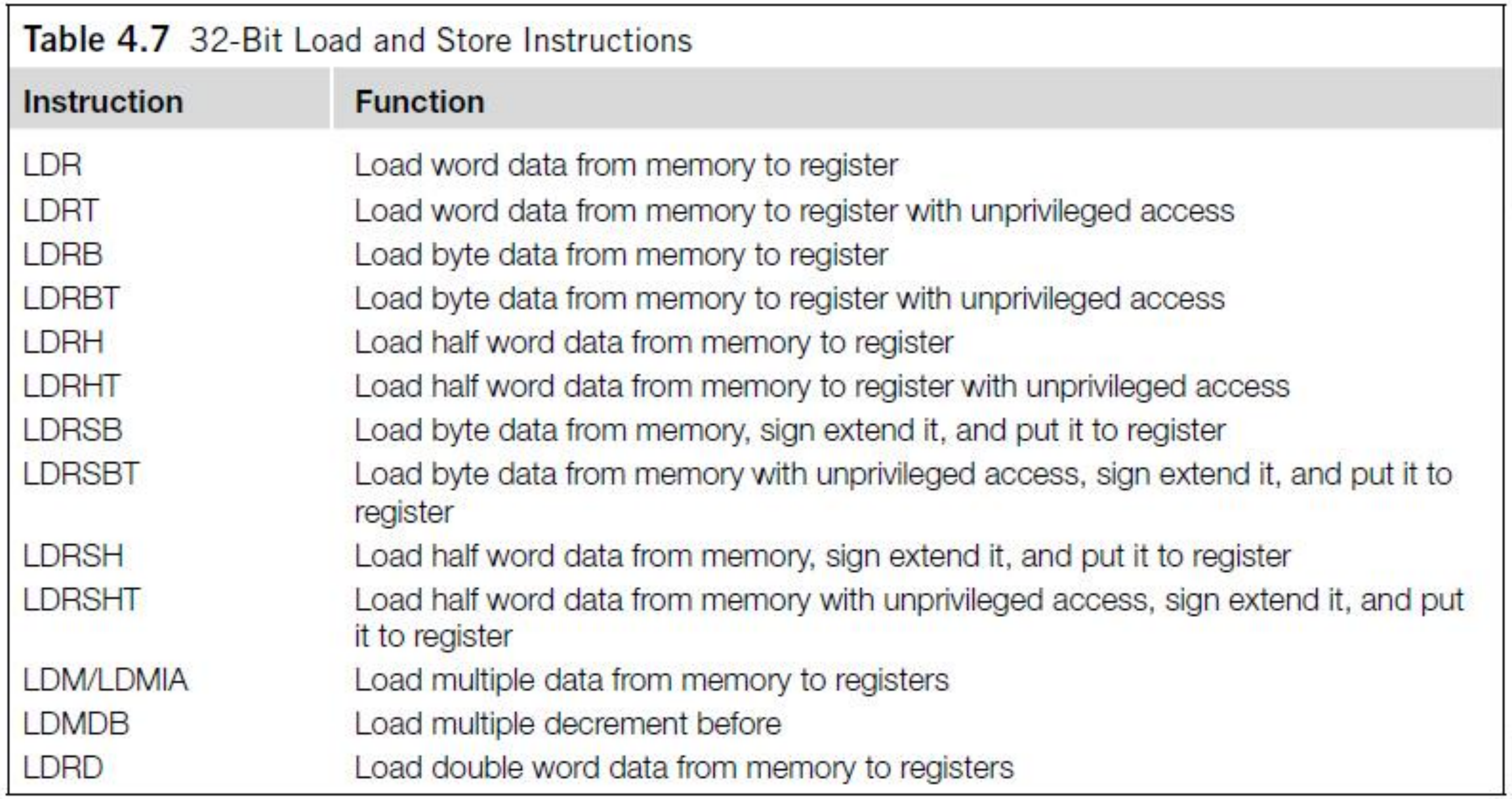


Table 4.7 32-Bit Load and Store Instructions

Instruction	Function
LDR	Load word data from memory to register
LDRT	Load word data from memory to register with unprivileged access
LDRB	Load byte data from memory to register
LDRBT	Load byte data from memory to register with unprivileged access
LDRH	Load half word data from memory to register
LDRHT	Load half word data from memory to register with unprivileged access
LDRSB	Load byte data from memory, sign extend it, and put it to register
LDRSBT	Load byte data from memory with unprivileged access, sign extend it, and put it to register
LDRSH	Load half word data from memory, sign extend it, and put it to register
LDRSHT	Load half word data from memory with unprivileged access, sign extend it, and put it to register
LDM/LDMIA	Load multiple data from memory to registers
LDMDB	Load multiple decrement before
LDRD	Load double word data from memory to registers

32-Bit Load and Store Instructions (continued)

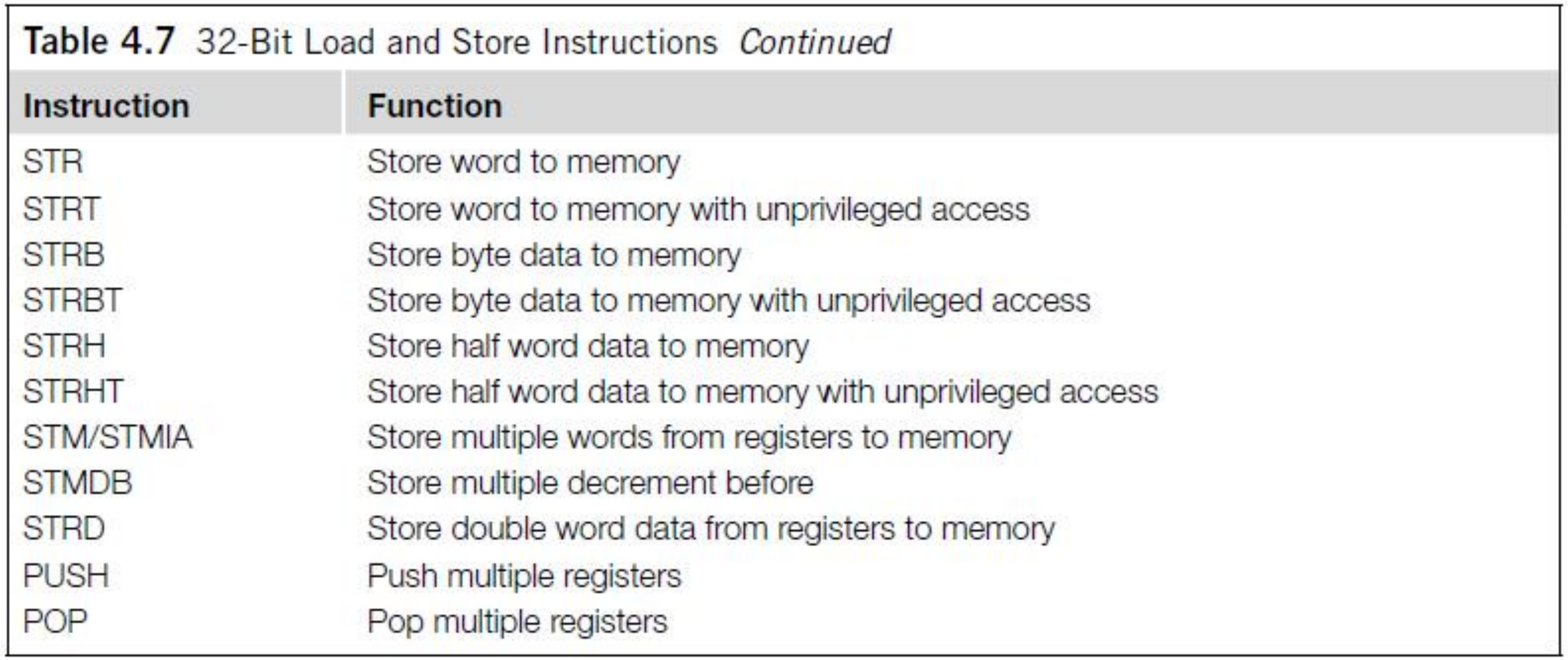


Table 4.7 32-Bit Load and Store Instructions *Continued*

Instruction	Function
STR	Store word to memory
STRT	Store word to memory with unprivileged access
STRB	Store byte data to memory
STRBT	Store byte data to memory with unprivileged access
STRH	Store half word data to memory
STRHT	Store half word data to memory with unprivileged access
STM/STMIA	Store multiple words from registers to memory
STMDB	Store multiple decrement before
STRD	Store double word data from registers to memory
PUSH	Push multiple registers
POP	Pop multiple registers

32-Bit Branch Instructions

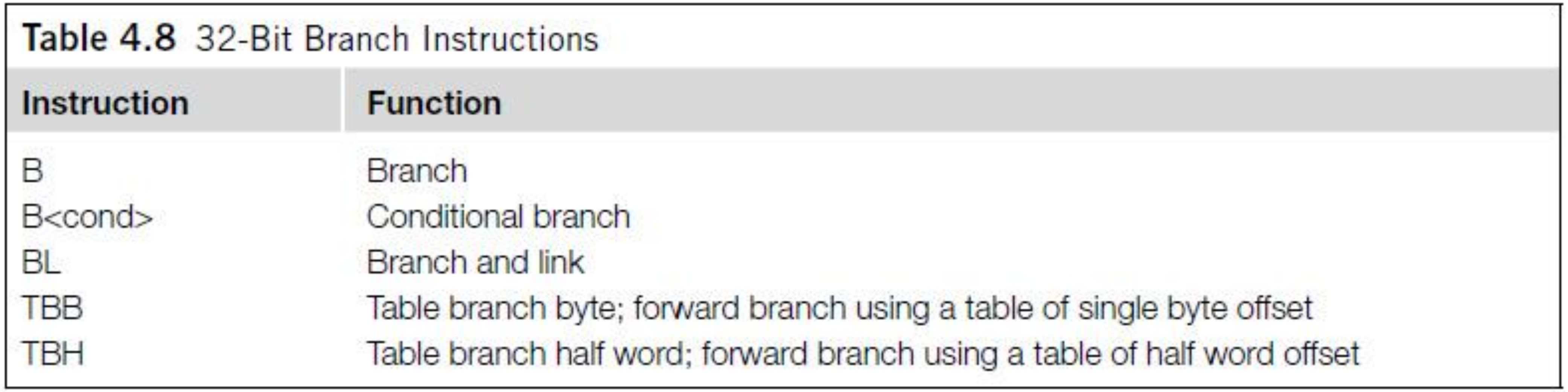


Table 4.8 32-Bit Branch Instructions

Instruction	Function
B	Branch
B<cond>	Conditional branch
BL	Branch and link
TBB	Table branch byte; forward branch using a table of single byte offset
TBH	Table branch half word; forward branch using a table of half word offset

Other 32-Bit Instructions

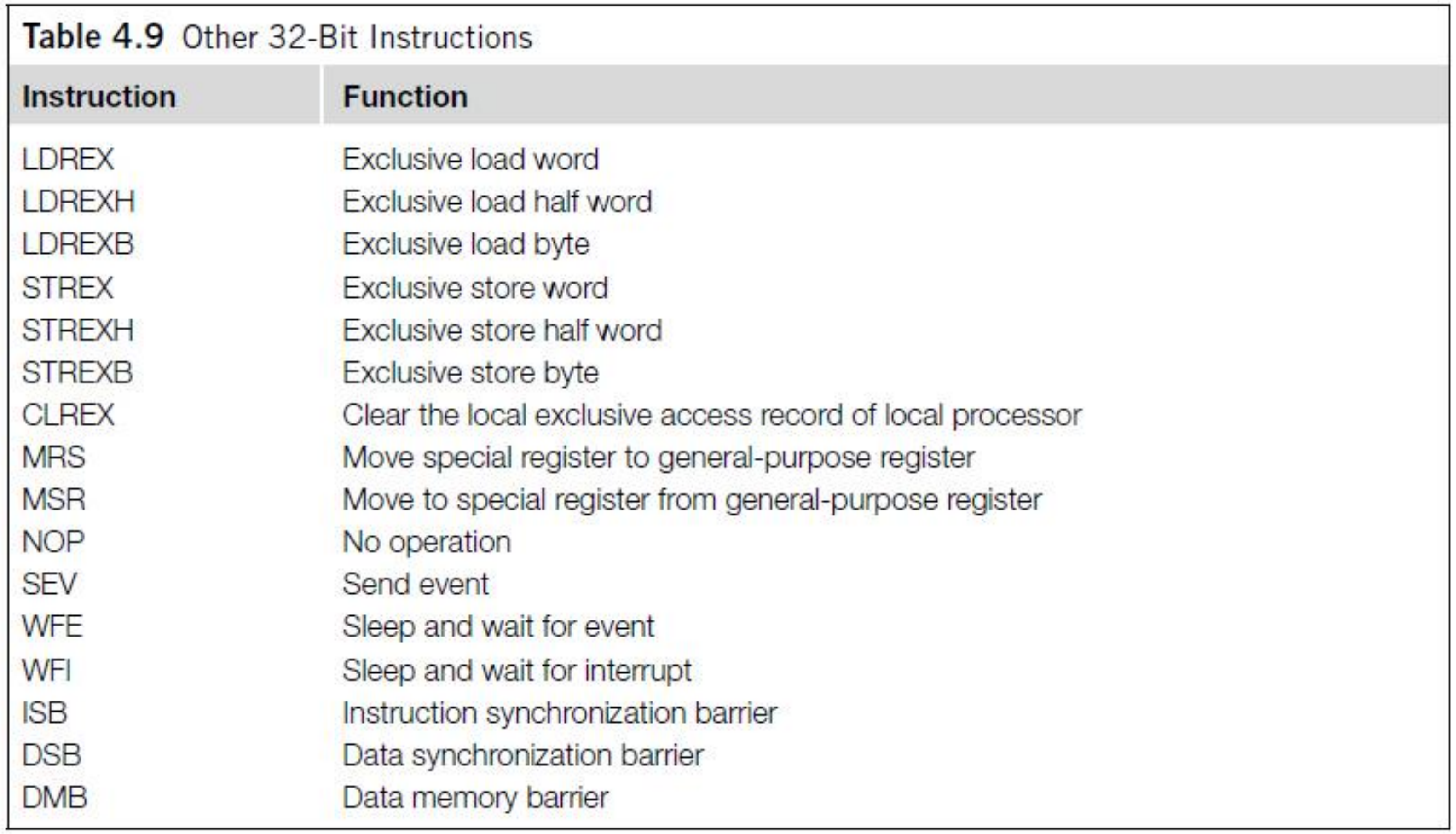


Table 4.9 Other 32-Bit Instructions

Instruction	Function
LDREX	Exclusive load word
LDREXH	Exclusive load half word
LDREXB	Exclusive load byte
STREX	Exclusive store word
STREXH	Exclusive store half word
STREXB	Exclusive store byte
CLREX	Clear the local exclusive access record of local processor
MRS	Move special register to general-purpose register
MSR	Move to special register from general-purpose register
NOP	No operation
SEV	Send event
WFE	Sleep and wait for event
WFI	Sleep and wait for interrupt
ISB	Instruction synchronization barrier
DSB	Data synchronization barrier
DMB	Data memory barrier

Instruction Descriptions

Assembler Language: Moving Data

- 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:
  - Moving data between register and register
  - Moving data between memory and register
  - Moving data between special register and register
  - Moving an immediate data value into a register

Assembler Language: Moving Data (continued)

- The command to move data between registers is MOV (move).
  - For example, the instruction

```
MOV R8, R3
```

  moves data from register R3 to register R8.
- Another instruction can generate the negative value of the original data; it is called MVN (move NOT).

```
MVN R8, R3
```

performs a bitwise logical NOT operation on data from register R3 and moves it to register R8.

Assembler Language: Moving Data (continued)

- The basic instructions for accessing memory are Load and Store.
- Load (LDR) transfers data from memory to registers, and Store (STR) transfers data from registers to memory.
- The transfers can be in different data sizes (byte, half word, word, and double word), as outlined in Table 4.14.

Assembler Language: Moving Data (continued)

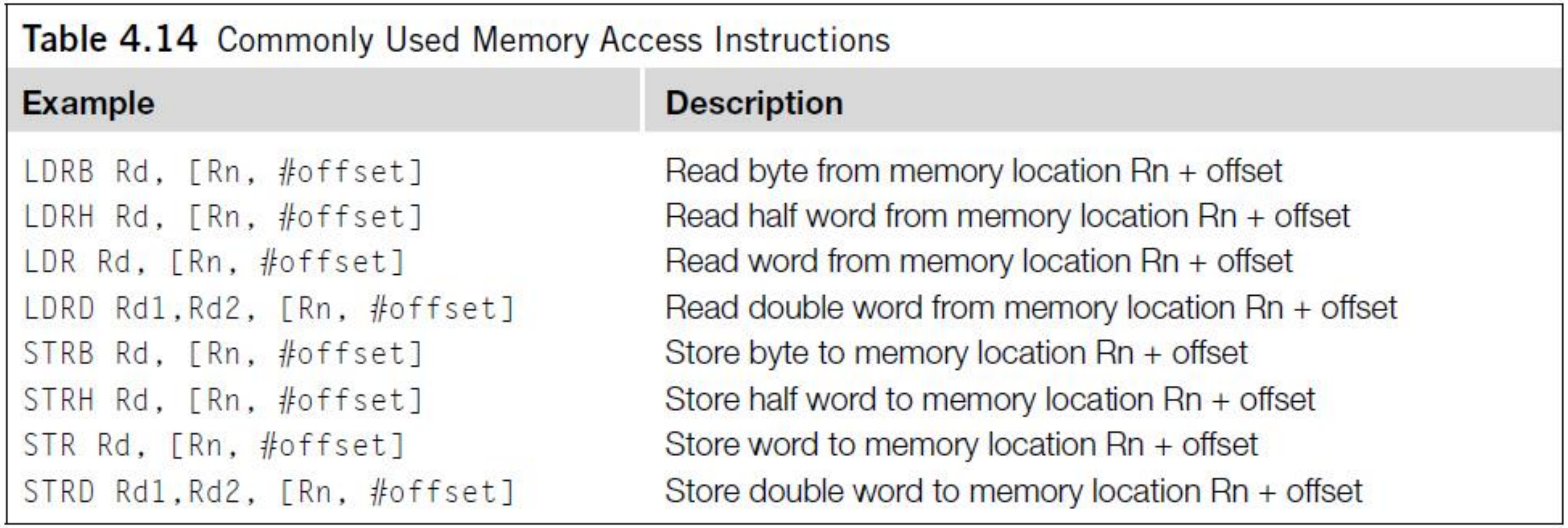


Table 4.14 Commonly Used Memory Access Instructions

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

Assembler Language: Moving Data (continued)

- Multiple Load and Store operations can be combined into single instructions called LDM (Load Multiple) and STM (Store Multiple), as outlined in Table 4.15.

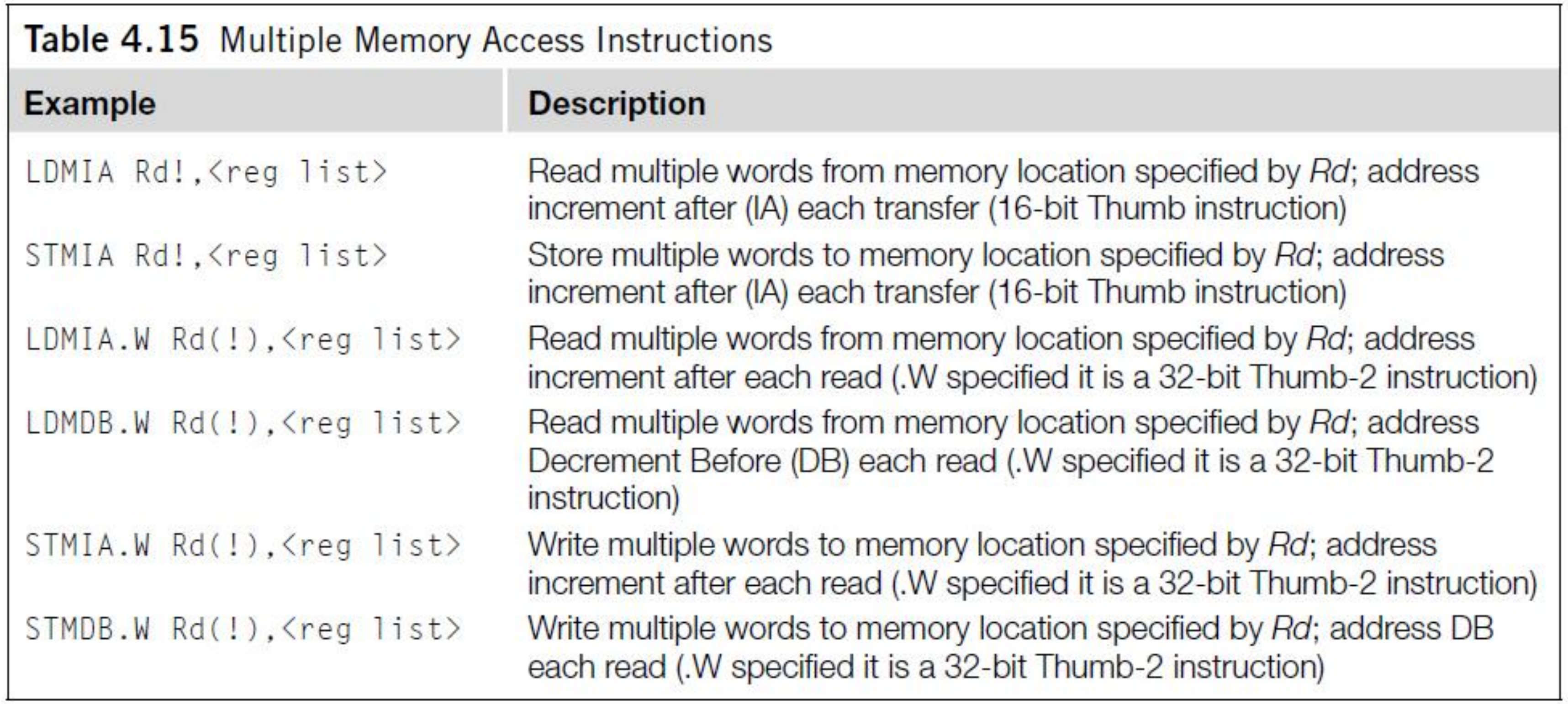


Table 4.15 Multiple Memory Access Instructions

Example	Description
`LDMIA Rd!,<reg list>`	Read multiple words from memory location specified by *Rd*; address increment after (IA) each transfer (16-bit Thumb instruction)
`STMIA Rd!,<reg list>`	Store multiple words to memory location specified by *Rd*; address increment after (IA) each transfer (16-bit Thumb instruction)
`LDMIA.W Rd(!),<reg list>`	Read multiple words from memory location specified by *Rd*; address increment after each read (.W specified it is a 32-bit Thumb-2 instruction)
`LDMDB.W Rd(!),<reg list>`	Read multiple words from memory location specified by *Rd*; address Decrement Before (DB) each read (.W specified it is a 32-bit Thumb-2 instruction)
`STMIA.W Rd(!),<reg list>`	Write multiple words to memory location specified by *Rd*; address increment after each read (.W specified it is a 32-bit Thumb-2 instruction)
`STMDB.W Rd(!),<reg list>`	Write multiple words to memory location specified by *Rd*; address DB each read (.W specified it is a 32-bit Thumb-2 instruction)

Assembler Language: Moving Data (continued)

- The exclamation mark (!) in the instruction specifies whether the register *Rd* should be updated after the instruction is completed.
  - For example, if R8 equals 0x8000:

```
STMIA.W R8!, {R0-R3} ; R8 changed to 0x8010 after store
                     ; (increment by 4 words)
STMIA.W R8 , {R0-R3} ; R8 unchanged after store
```

Assembler Language: Moving Data (continued)

- 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.
    - For 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.
    - The “!” is optional; without it, the instruction would be just a normal memory transfer with offset from a base address.
- The preindexing memory access instructions include load and store instructions of various transfer sizes (see Table 4.16).

Assembler Language: Moving Data (continued)

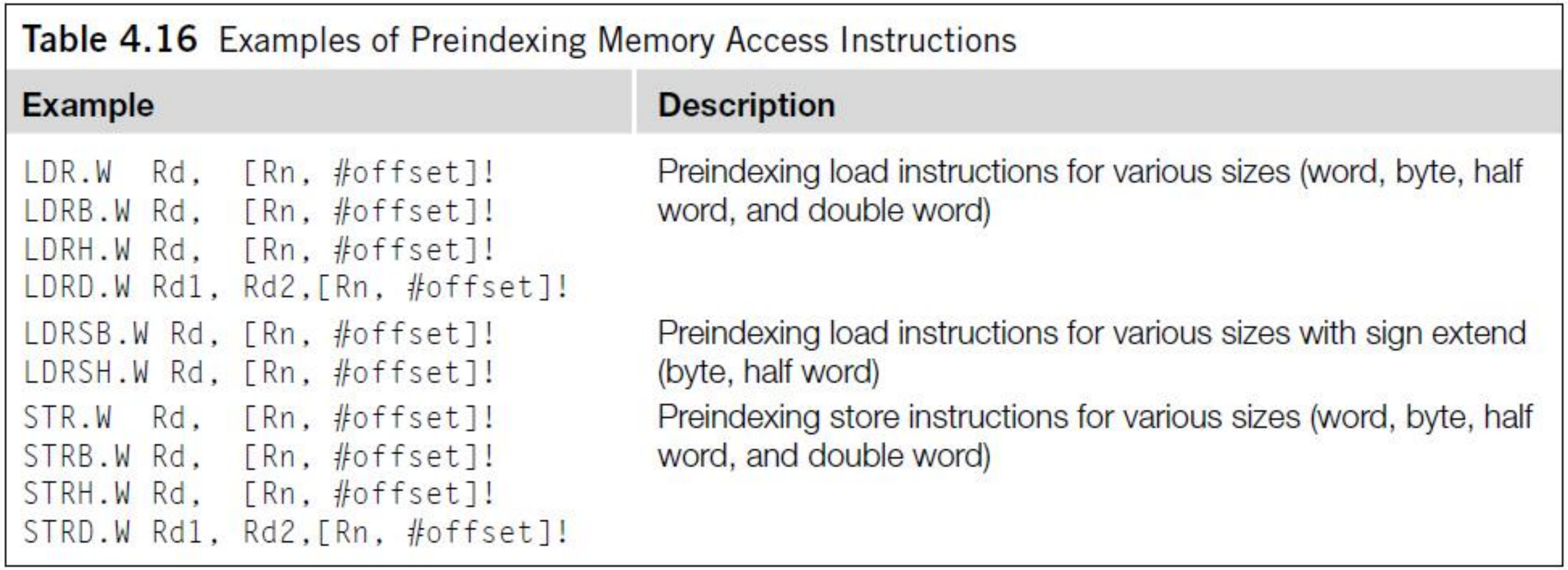


Table 4.16 Examples of Preindexing Memory Access Instructions

Example	Description

`LDR.W   Rd,   [Rn, #offset]!`
`LDRB.W  Rd,   [Rn, #offset]!`
`LDRH.W  Rd,   [Rn, #offset]!`
`LDRD.W  Rd1, Rd2,[Rn, #offset]!`
	Preindexing load instructions for various sizes (word, byte, half word, and double word)

`LDRSB.W Rd, [Rn, #offset]!`
`LDRSH.W Rd, [Rn, #offset]!`
	Preindexing load instructions for various sizes with sign extend (byte, half word)

`STR.W   Rd,   [Rn, #offset]!`
`STRB.W  Rd,   [Rn, #offset]!`
`STRH.W  Rd,   [Rn, #offset]!`
`STRD.W  Rd1, Rd2,[Rn, #offset]!`
	Preindexing store instructions for various sizes (word, byte, half word, and double word)

Assembler Language: Moving Data (continued)

- Postindexing memory access instructions carry out the memory transfer using the base address specified by the register and then update the address register afterward.
    - For example,

```
LDR.W R0,[R1], #offset ; Read memory[R1], with R1
                       ; updated to R1+offset
```

- When a postindexing instruction is used, there is no need to use the "!" sign, because all postindexing instructions update the base address register.
- Similarly to preindexing, postindexing memory access instructions are available for different transfer sizes (see Table 4.17).

Assembler Language: Moving Data (continued)

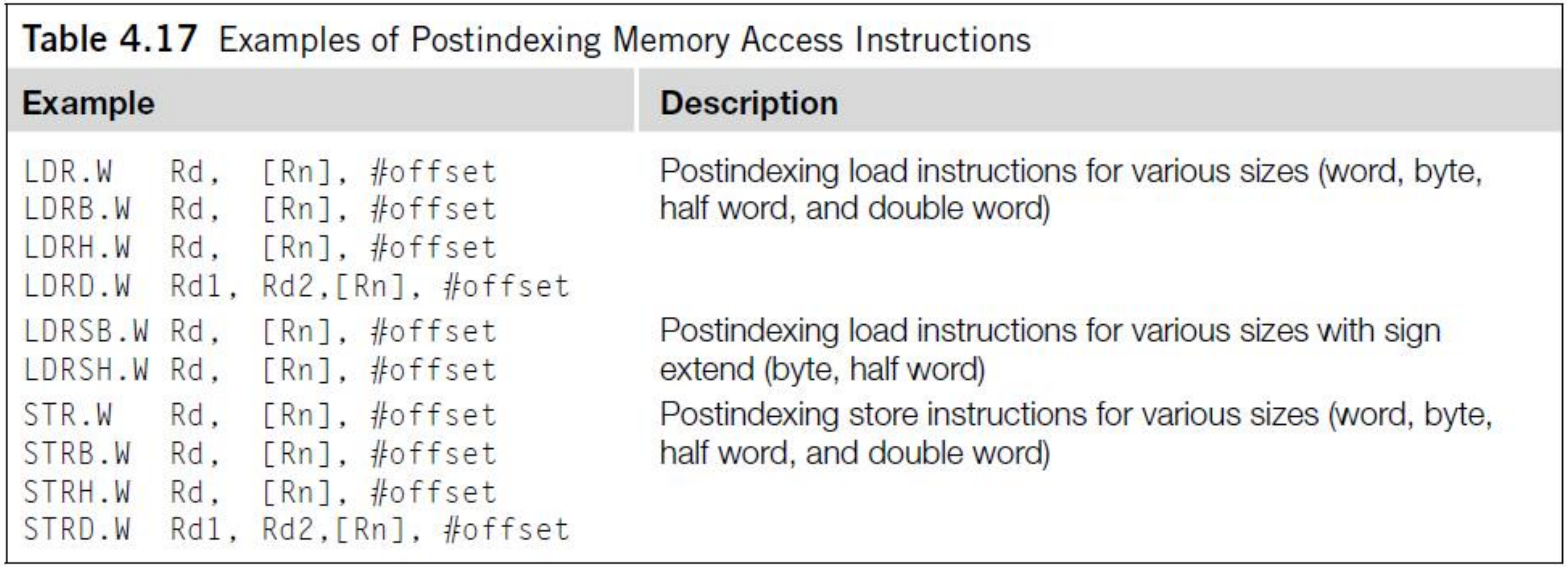


Table 4.17 Examples of Postindexing Memory Access Instructions

Example	Description

`LDR.W    Rd,   [Rn], #offset`
`LDRB.W   Rd,   [Rn], #offset`
`LDRH.W   Rd,   [Rn], #offset`
`LDRD.W   Rd1, Rd2,[Rn], #offset`
	Postindexing load instructions for various sizes (word, byte, half word, and double word)

`LDRSB.W Rd,   [Rn], #offset`
`LDRSH.W Rd,   [Rn], #offset`
	Postindexing load instructions for various sizes with sign extend (byte, half word)

`STR.W    Rd,   [Rn], #offset`
`STRB.W   Rd,   [Rn], #offset`
`STRH.W   Rd,   [Rn], #offset`
`STRD.W   Rd1, Rd2,[Rn], #offset`
	Postindexing store instructions for various sizes (word, byte, half word, and double word)

Assembler Language: Moving Data (continued)

- Two other types of memory operation are stack PUSH and stack POP.
  - For example,

```
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.

Assembler Language: Moving Data (continued)

- To access special registers, we use the instructions MRS and MSR.
  - For example,

```
MRS R0, PSR      ; Read Processor status word into R0
MSR CONTROL, R1  ; Write value of R1 into control register
```

- Unless you're accessing the APSR, you can use MSR or MRS to access other special registers only in privileged mode.

Assembler Language: Moving Data (continued)

- Moving immediate data into a register is a common thing to do.
- For example, you might want to access a peripheral register, so you need to put the address value into a register beforehand.
- For small values (8 bits or less), you can use MOVS (move).
  - For example,

```
MOVS R0, #0x12 ; Set R0 to 0x12
```

- For a larger value (over 8 bits), you might need to use a Thumb-2 move instruction.
  - For example,

```
MOVW.W R0, #0x789A ; Set R0 to 0x789A
```

Assembler Language: Moving Data (continued)

- Or if the value is 32-bit, you can use two instructions to set the upper and lower halves:

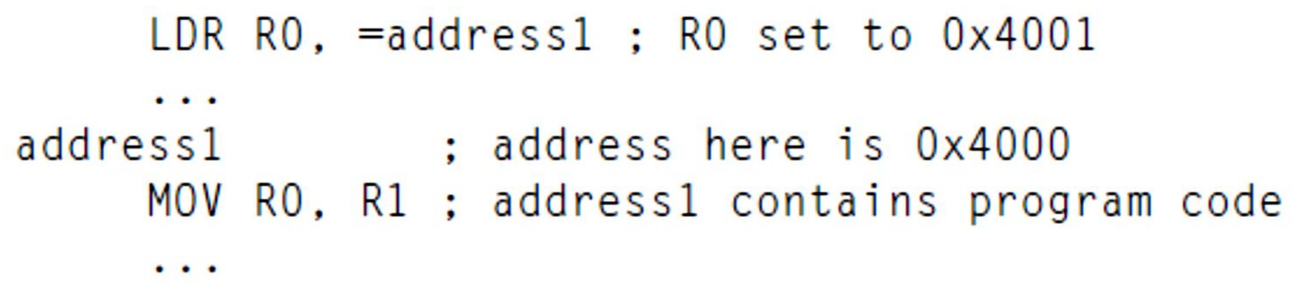
```
MOVW.W R0,#0x789A ; Set R0 lower half to 0x789A
MOVT.W R0,#0x3456 ; Set R0 upper half to 0x3456. Now
                  ; R0=0x3456789A
```

LDR and ADR Pseudo-Instructions

- LDR and ADR pseudo-instructions can be used to set registers to a program address value.
- This is not a real assembler command, but the ARM assembler will convert it into a PC relative load instruction to produce the required data.
- To generate 32-bit immediate data, using LDR is recommended rather than the MOVW.W and MOVT.W combination because it gives better readability and the assembler might be able to reduce the memory being used if the same immediate data are reused in several places of the same program.

LDR and ADR Pseudo-Instructions (continued)

- For LDR, if the address is a program address value, the assembler will automatically set the LSB to 1.
    - For example,



```
          LDR R0, =address1 ; R0 set to 0x4001
          ...
     address1        ; address here is 0x4000
          MOV R0, R1 ; address1 contains program code
          ...
```

- You will find that the LDR instruction will put 0x4001 into R1; the LSB is set to 1 to indicate that it is Thumb code.

LDR and ADR Pseudo-Instructions (continued)

- If *address1* is a data address, LSB will not be changed.
  - For example,

```
        LDR R0, =address1 ; R0 set to 0x4000
        ...
address1        ; address here is 0x4000
        DCD 0x0 ; address1 contains data
        ...
```

LDR and ADR Pseudo-Instructions (continued)

- For ADR, you can load the address value of a program code into a register without setting the LSB automatically.
  - For example,

```
          ADR R0, address1
          ...
      address1        ; (address here is 0x4000)
          MOV R0, R1 ; address1 contains program code
          ...
```

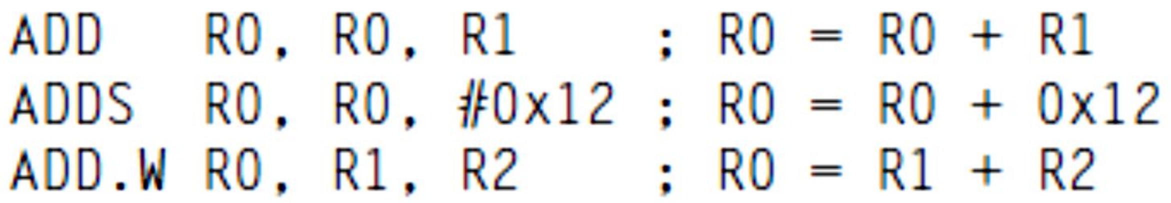
- You will get 0x4000 in the ADR instruction.
- Note that there is no equal sign (=) in the ADR statement.

LDR and ADR Pseudo-Instructions (continued)

- LDR obtains the immediate data by putting the data in the program code and uses a PC relative load to get the data into the register.
- ADR tries to generate the immediate value by adding or subtracting instructions (for example, based on the current PC value).
- As a result, it is not possible to create all immediate values using ADR, and the target address label must be in a close range.
- However, using ADR can generate smaller code sizes compared with LDR.

Assembler Language: Processing Data

- The Cortex-M3 provides many different instructions for data processing.
- Many data operation instructions can have multiple instruction formats.
  - For example, an ADD instruction can operate between two registers or between one register and an immediate data value:



```
ADD    R0, R0, R1    ; R0 = R0 + R1
ADDS   R0, R0, #0x12 ; R0 = R0 + 0x12
ADD.W  R0, R1, R2    ; R0 = R1 + R2
```

- These are all ADD instructions, but they have different syntaxes and binary coding.

Assembler Language: Processing Data (continued)

- With the traditional Thumb instruction syntax, when 16-bit Thumb code is used, an ADD instruction can change the flags in the PSR.
- However, 32-bit Thumb-2 code can either change a flag or keep it unchanged.
  - To separate the two different operations, the S suffix should be used if the following operation depends on the flags:

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

Assembler Language: Processing Data (continued)

- Aside from ADD instructions, the arithmetic functions that the Cortex-M3 supports include subtract (SUB), multiply (MUL), and unsigned and signed divide (UDIV/SDIV).
- Table 4.18 shows some of the most commonly used arithmetic instructions.
- These instructions can be used with or without the “S” suffix to determine if the APSR should be updated.
  - In most cases, if UAL syntax is selected and if “S” suffix is not used, the 32-bit version of the instructions would be selected as most of the 16-bit Thumb instructions update APSR.

Assembler Language: Processing Data (continued)

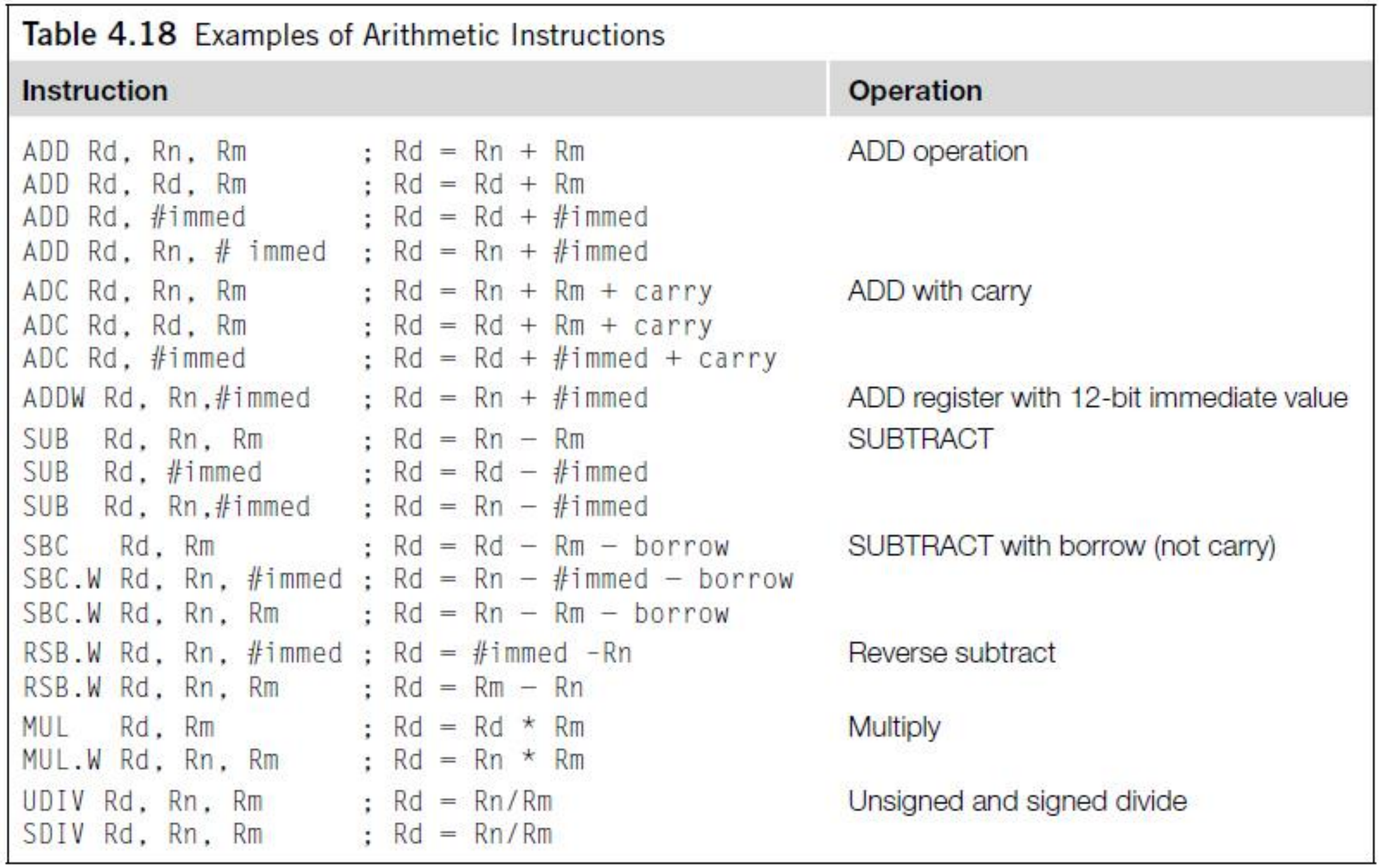


Table 4.18 Examples of Arithmetic Instructions

Instruction	Operation
`ADD Rd, Rn, Rm ; Rd = Rn + Rm`	ADD operation
`ADD Rd, Rd, Rm ; Rd = Rd + Rm`	
`ADD Rd, #immed ; Rd = Rd + #immed`	
`ADD Rd, Rn, # immed ; Rd = Rn + #immed`	
`ADC Rd, Rn, Rm ; Rd = Rn + Rm + carry`	ADD with carry
`ADC Rd, Rd, Rm ; Rd = Rd + Rm + carry`	
`ADC Rd, #immed ; 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 ; Rd = Rd − #immed`	
`SUB Rd, Rn,#immed ; Rd = Rn − #immed`	
`SBC Rd, Rm ; Rd = Rd − Rm − borrow`	SUBTRACT with borrow (not carry)
`SBC.W Rd, Rn, #immed ; Rd = Rn − #immed − borrow`	
`SBC.W Rd, Rn, Rm ; Rd = Rn − Rm − borrow`	
`RSB.W Rd, Rn, #immed ; Rd = #immed -Rn`	Reverse subtract
`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`	

Assembler Language: Processing Data (continued)

- The Cortex-M3 also supports 32-bit multiply instructions and multiply accumulate instructions that give 64-bit results.
- These instructions support signed or unsigned values (see Table 4.19).

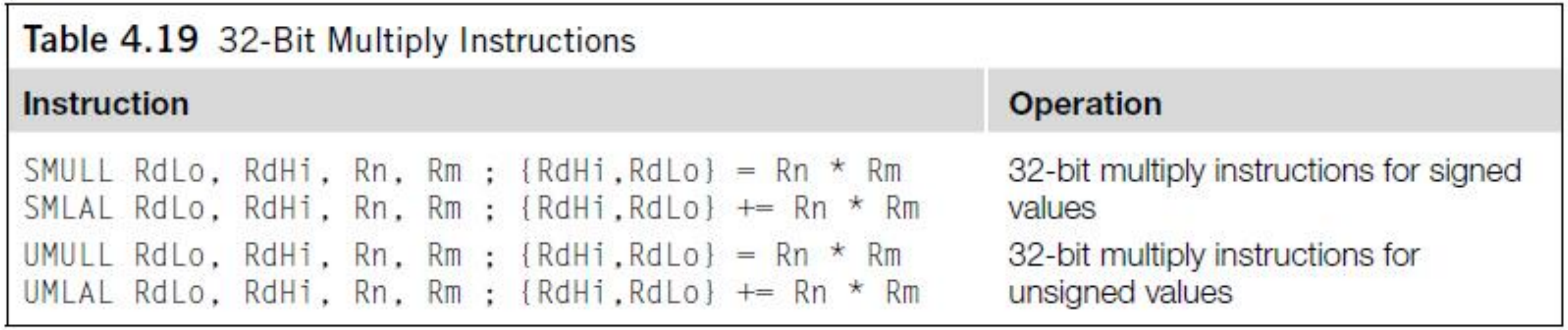


Table 4.19 32-Bit Multiply Instructions

Instruction	Operation

`SMULL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} = Rn * Rm`
`SMLAL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} += Rn * Rm`
	32-bit multiply instructions for signed values

`UMULL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} = Rn * Rm`
`UMLAL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} += Rn * Rm`
	32-bit multiply instructions for unsigned values

Assembler Language: Processing Data (continued)

- Another group of data processing instructions are the logical operations instructions and logical operations such as AND, ORR (or), and shift and rotate functions.
- Table 4.20 shows some of the most commonly used logical instructions.
- These instructions can be used with or without the "S" suffix to determine if the APSR should be updated.
  - If UAL syntax is used and if "S" suffix is not used, the 32-bit version of the instructions would be selected as all of the 16-bit logic operation instructions update APSR.

Assembler Language: Processing Data (continued)

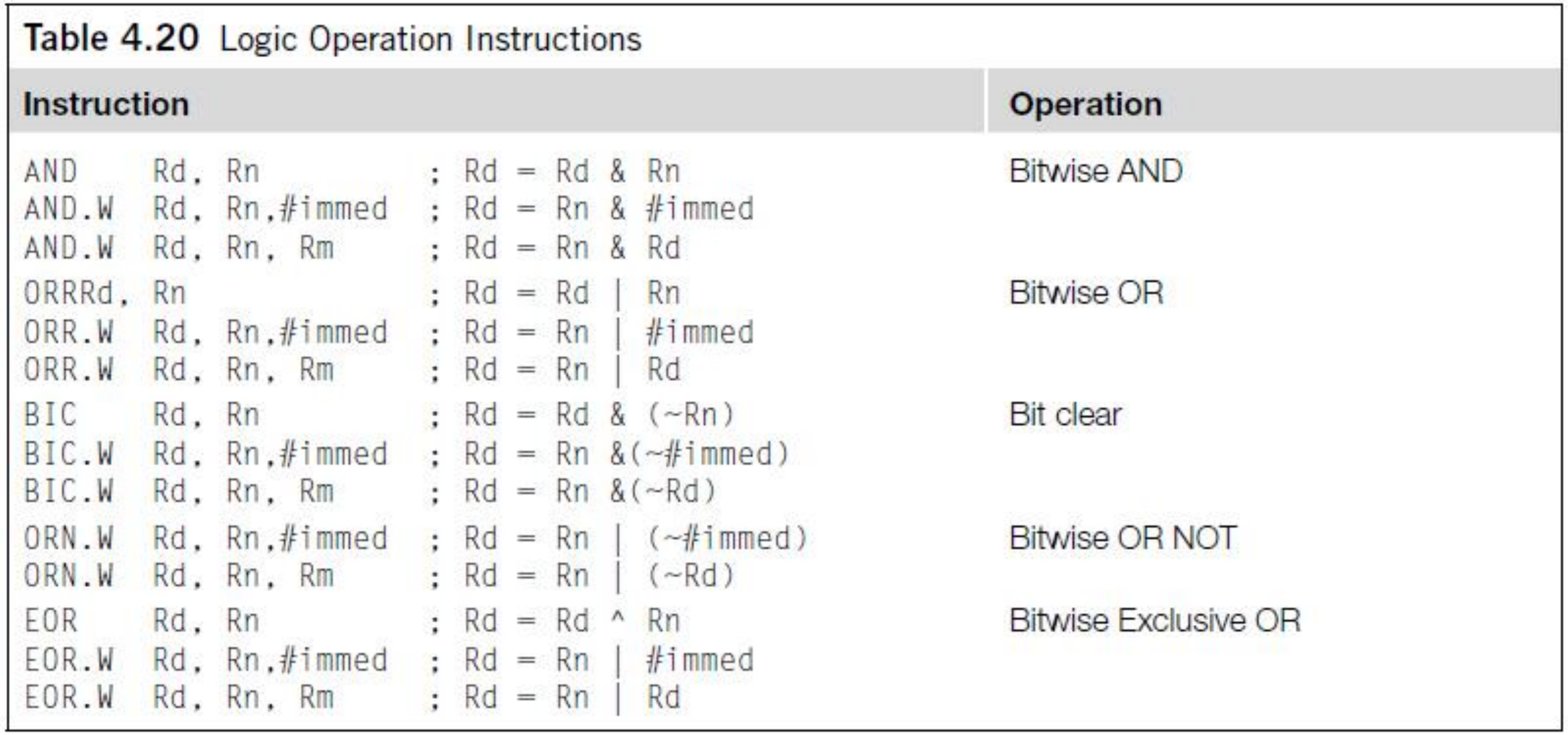


Table 4.20 Logic Operation Instructions

Instruction	Operation
`AND Rd, Rn ; Rd = Rd & Rn`	Bitwise AND
`AND.W Rd, Rn,#immed ; Rd = Rn & #immed`	
`AND.W Rd, Rn, Rm ; Rd = Rn & Rd`	
`ORRRd, Rn ; Rd = Rd | Rn`	Bitwise OR
`ORR.W Rd, Rn,#immed ; Rd = Rn | #immed`	
`ORR.W Rd, Rn, Rm ; Rd = Rn | Rd`	
`BIC Rd, Rn ; Rd = Rd & (~Rn)`	Bit clear
`BIC.W Rd, Rn,#immed ; Rd = Rn &(~#immed)`	
`BIC.W Rd, Rn, Rm ; Rd = Rn &(~Rd)`	
`ORN.W Rd, Rn,#immed ; Rd = Rn | (~#immed)`	Bitwise OR NOT
`ORN.W Rd, Rn, Rm ; Rd = Rn | (~Rd)`	
`EOR Rd, Rn ; Rd = Rd ^ Rn`	Bitwise Exclusive OR
`EOR.W Rd, Rn,#immed ; Rd = Rn | #immed`	
`EOR.W Rd, Rn, Rm ; Rd = Rn | Rd`	

Assembler Language: Processing Data (continued)

- The Cortex-M3 provides rotate and shift instructions.
- In some cases, the rotate operation can be combined with other operations (for example, in memory address offset calculation for load/store instructions).
- For standalone rotate/shift operations, the instructions shown in Table 4.21 are provided.
- Again, a 32-bit version of the instruction is used if “S” suffix is not used and if UAL syntax is used.

Assembler Language: Processing Data (continued)

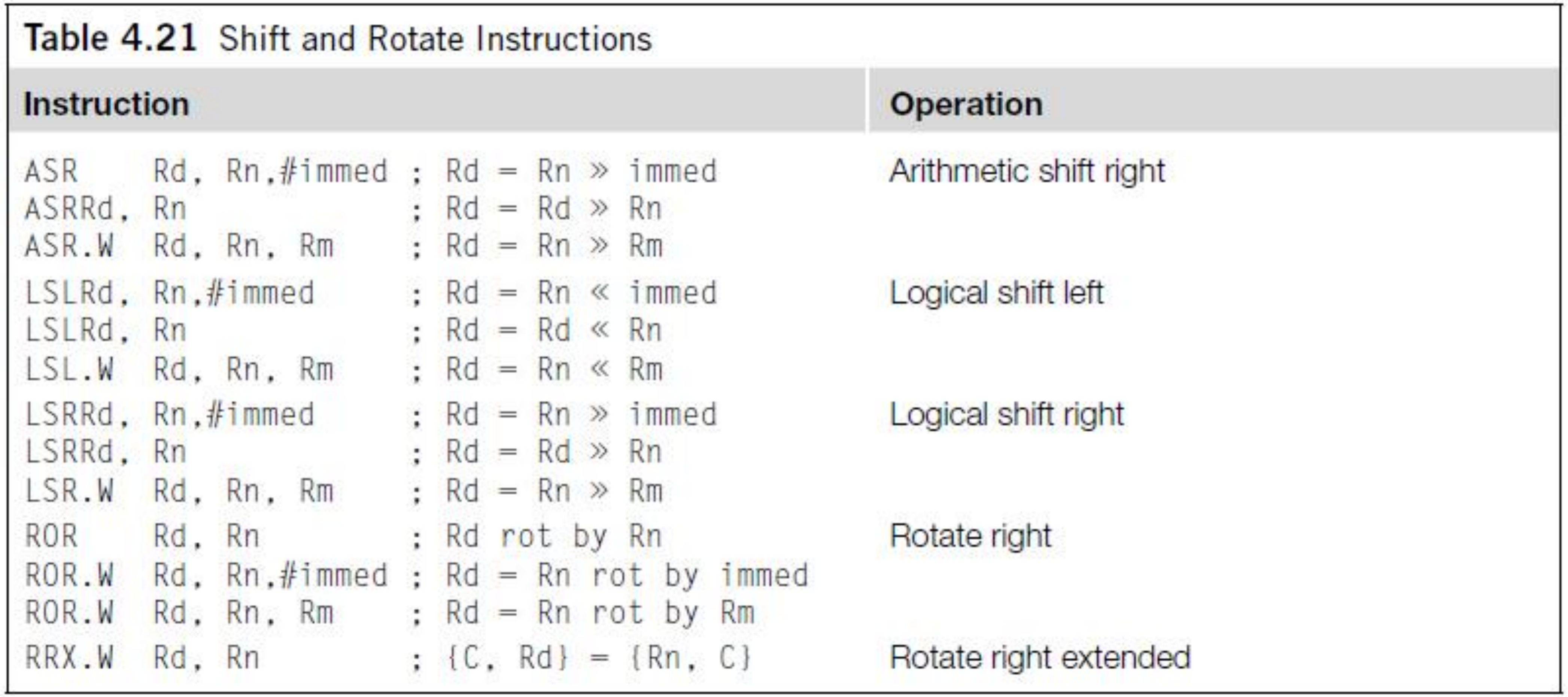


Table 4.21 Shift and Rotate Instructions

Instruction	Operation
`ASR    Rd, Rn,#immed ; Rd = Rn » immed`	Arithmetic shift right
`ASRRd, Rn            ; Rd = Rd » Rn`	
`ASR.W  Rd, Rn, Rm    ; Rd = Rn » Rm`	
`LSLRd, Rn,#immed     ; Rd = Rn « immed`	Logical shift left
`LSLRd, Rn            ; Rd = Rd « Rn`	
`LSL.W  Rd, Rn, Rm    ; Rd = Rn « Rm`	
`LSRRd, Rn,#immed     ; Rd = Rn » immed`	Logical shift right
`LSRRd, Rn            ; Rd = Rd » Rn`	
`LSR.W  Rd, Rn, Rm    ; Rd = Rn » Rm`	
`ROR    Rd, Rn        ; Rd rot by Rn`	Rotate right
`ROR.W  Rd, Rn,#immed ; Rd = Rn rot by immed`	
`ROR.W  Rd, Rn, Rm    ; Rd = Rn rot by Rm`	
`RRX.W  Rd, Rn        ; {C, Rd} = {Rn, C}`	Rotate right extended

Assembler Language: Processing Data (continued)

- In UAL syntax, the rotate and shift operations can also update the carry flag if the S suffix is used (and always update the carry flag if the 16-bit Thumb code is used).
  - See Figure 4.1.
- If the shift or rotate operation shifts the register position by multiple bits, the value of the carry flag *C* will be the last bit that shifts out of the register.

Assembler Language: Processing Data (continued)

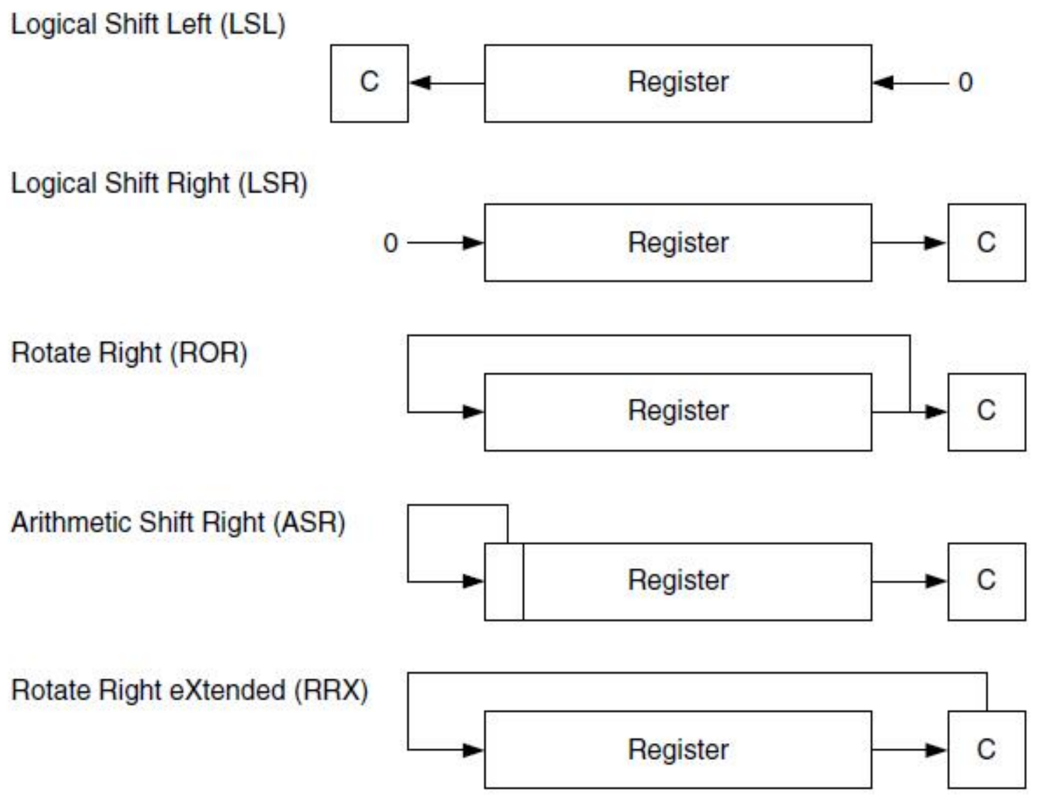


FIGURE 4.1

Shift and Rotate Instructions.

Assembler Language: Processing Data (continued)

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.

Assembler Language: Processing Data (continued)

- For conversion of signed data from byte or half word to word, the Cortex-M3 provides the two instructions shown in Table 4.22.
- Both 16-bit and 32-bit versions are available.
  - The 16-bit version can only access low registers.

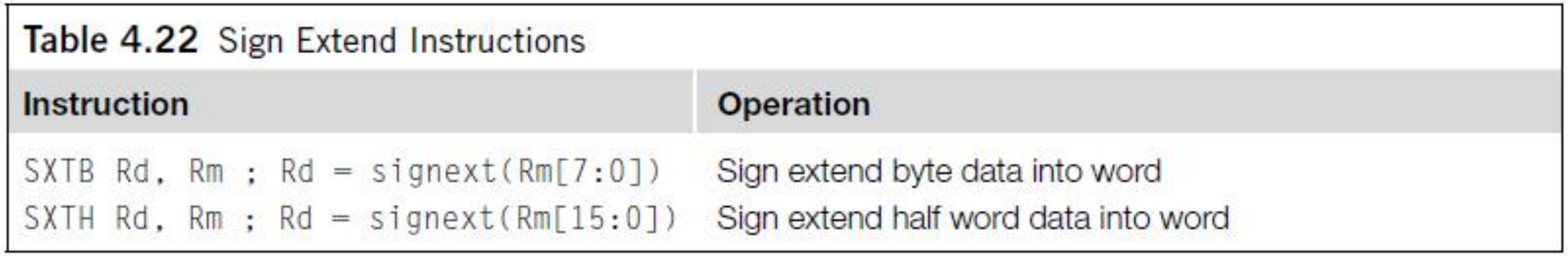


Table 4.22 Sign Extend Instructions

Instruction	Operation
`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

Assembler Language: Processing Data (continued)

- Another group of data processing instructions is used for reversing data bytes in a register (see Table 4.23).
- These instructions are usually used for conversion between little endian and big endian data.
  - See Figure 4.2.
- Both 16-bit and 32-bit versions are available.
  - The 16-bit version can only access low registers.

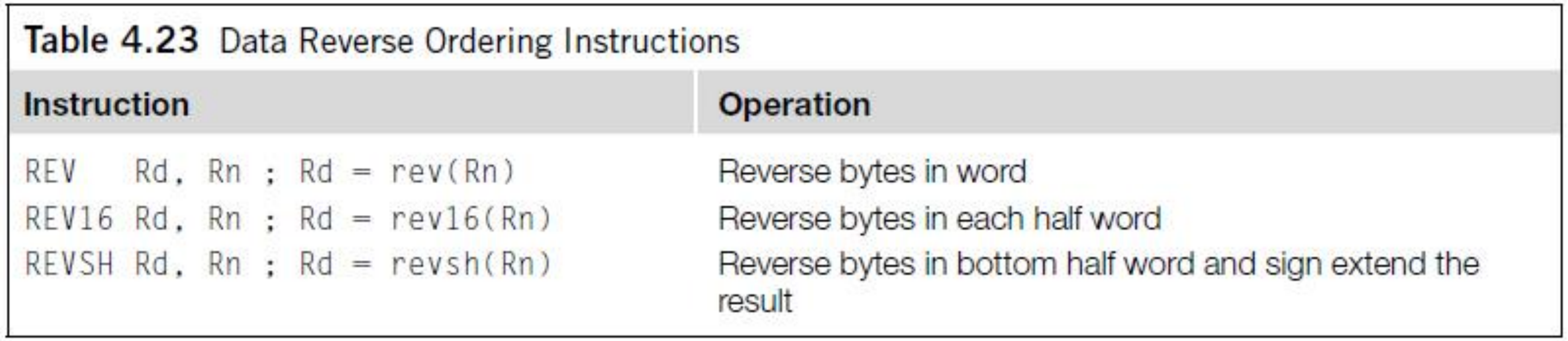


Table 4.23 Data Reverse Ordering Instructions

Instruction	Operation
`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

Assembler Language: Processing Data (continued)

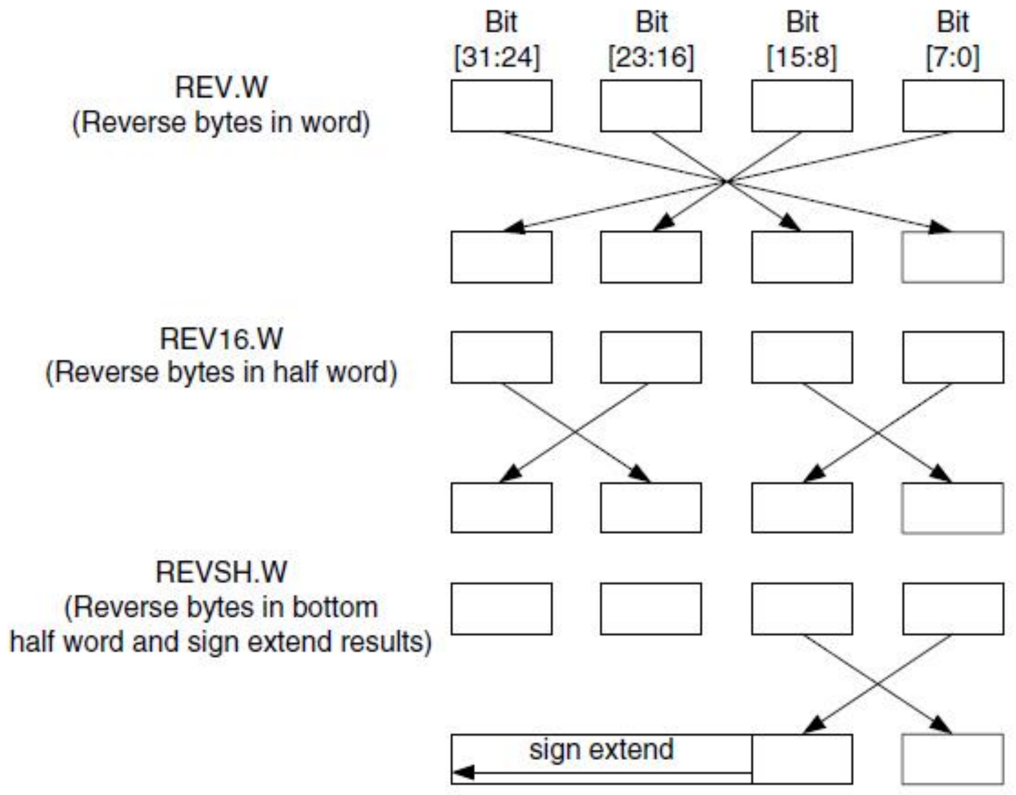


FIGURE 4.2

Operation of Reverse instructions.

Assembler Language: Processing Data (continued)

- The last group of data processing instructions is for bit field processing.
  - They include the instructions shown in Table 4.24.

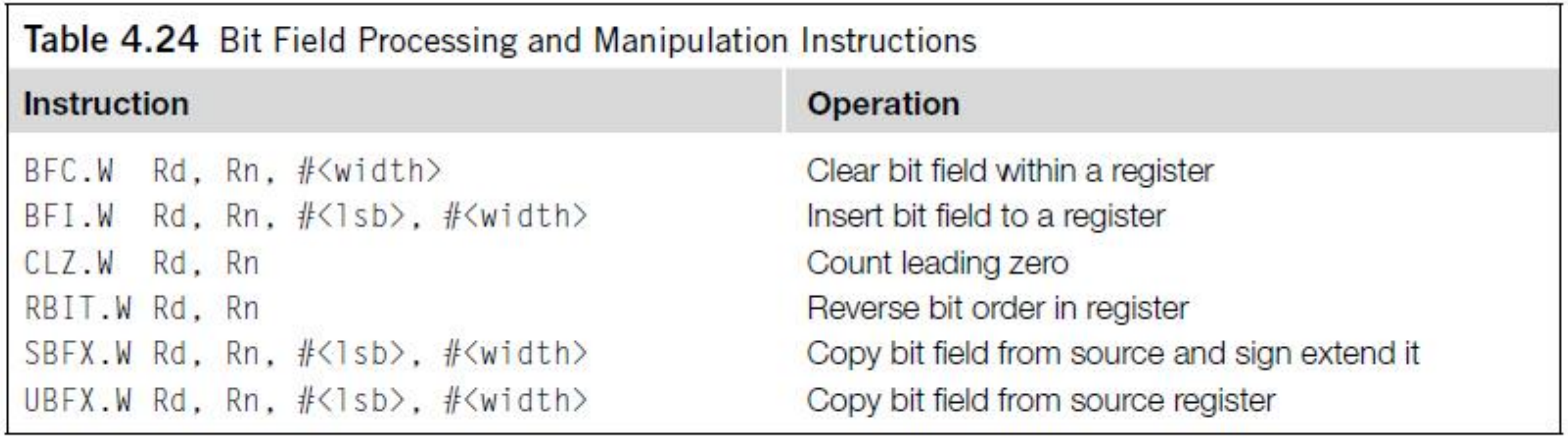
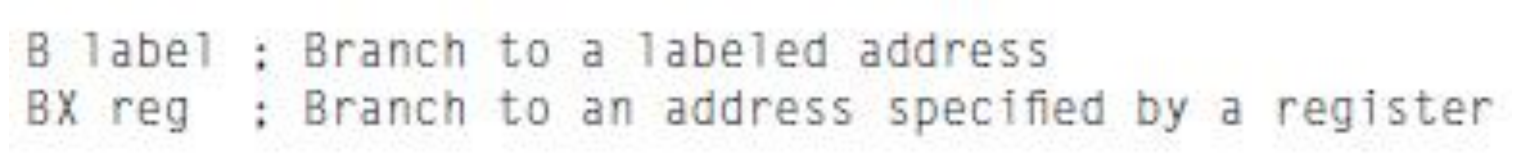


Table 4.24 Bit Field Processing and Manipulation Instructions

Instruction	Operation
`BFC.W  Rd, Rn, #<width>`	Clear bit field within a register
`BFI.W  Rd, Rn, #<lsb>, #<width>`	Insert bit field to a register
`CLZ.W  Rd, Rn`	Count leading zero
`RBIT.W Rd, Rn`	Reverse bit order in register
`SBFX.W Rd, Rn, #<lsb>, #<width>`	Copy bit field from source and sign extend it
`UBFX.W Rd, Rn, #<lsb>, #<width>`	Copy bit field from source register

Assembler Language: Call and Unconditional Branch

- The most basic branch instructions are as follows:

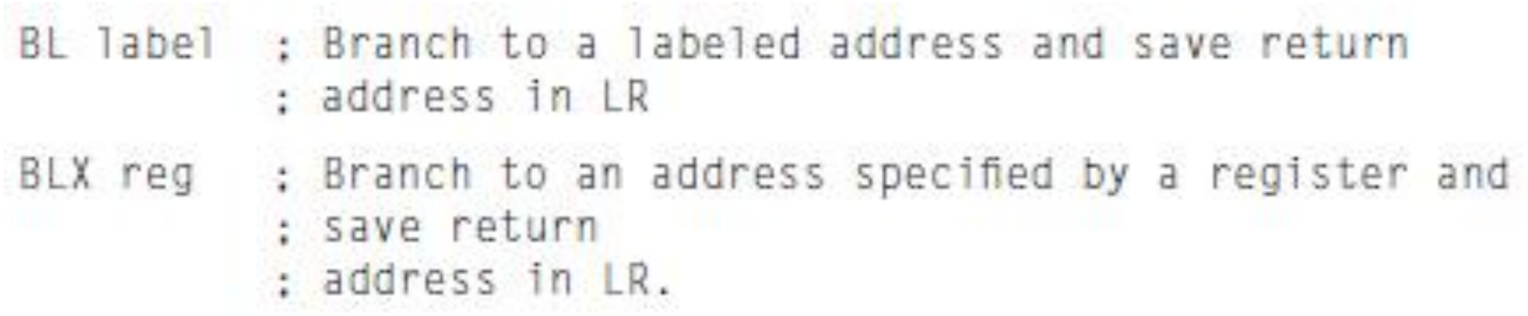


```
B label  ; Branch to a labeled address
BX reg   ; Branch to an address specified by a register
```

- 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.

Assembler Language: Call and Unconditional Branch (continued)

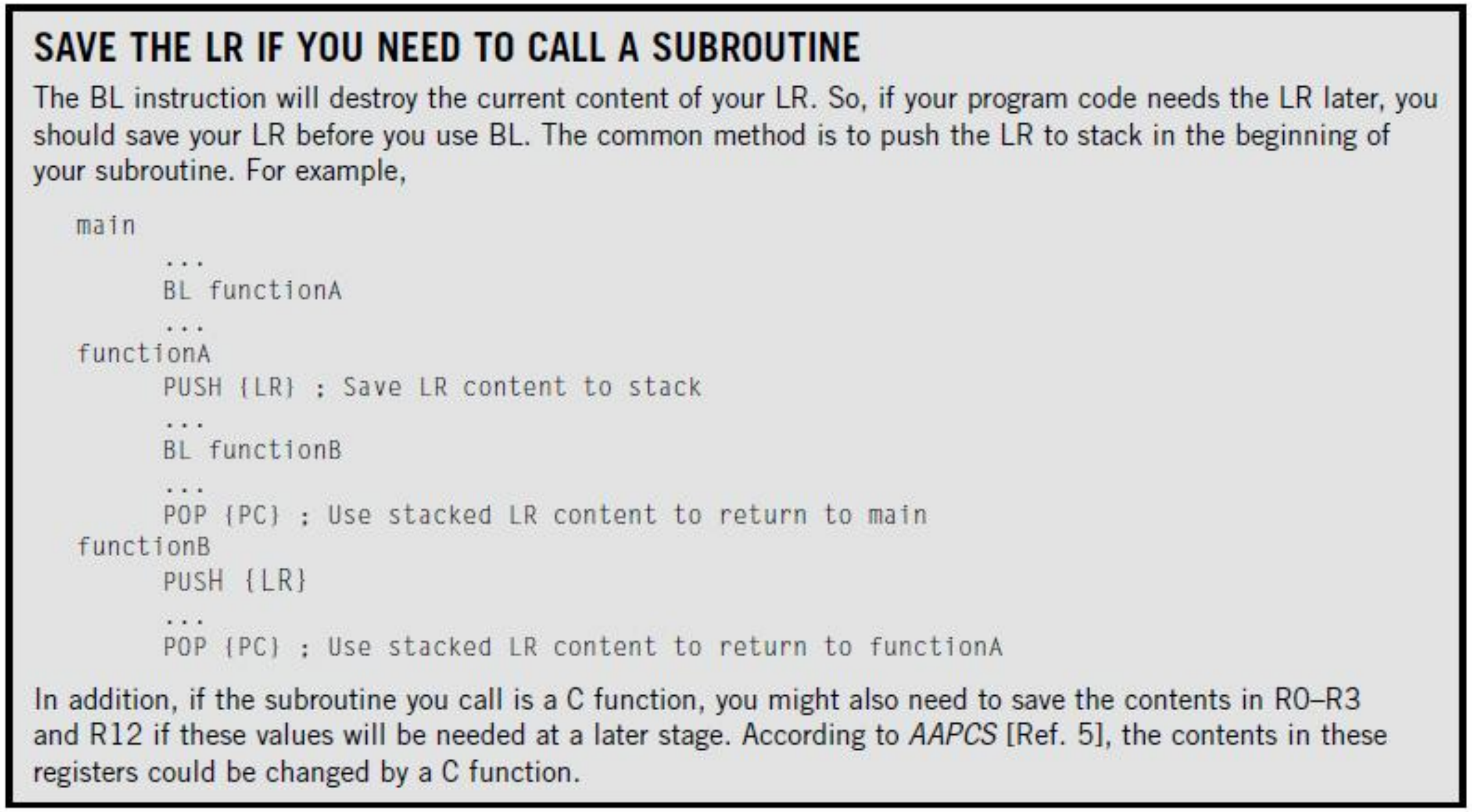
- To call a function, the branch and link instructions should be used.



```
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.

Assembler Language: Call and Unconditional Branch (continued)



SAVE THE LR IF YOU NEED TO CALL A SUBROUTINE

The BL instruction will destroy the current content of your LR. So, if your program code needs the LR later, you should save your LR before you use BL. The common method is to push the LR to stack in the beginning of your 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. According to *AAPCS* [Ref. 5], the contents in these registers could be changed by a C function.

Assembler Language: 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, as shown in Table 4.25.

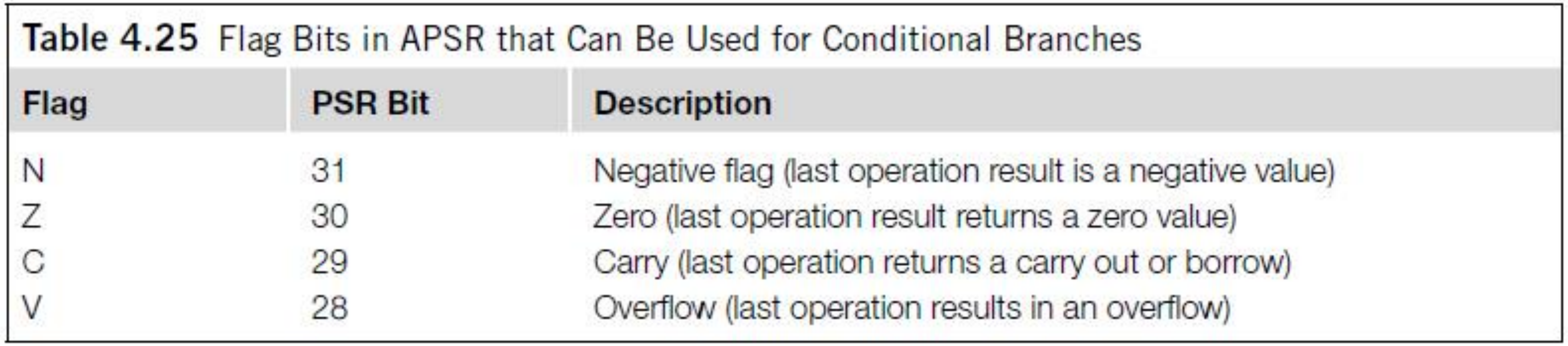


Table 4.25 Flag Bits in APSR that Can Be Used for Conditional Branches

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

Assembler Language: Decisions and Conditional Branches (continued)

- With combinations of the four flags (*N, Z, C,* and *V*), 15 branch conditions are defined, as shown in Table 4.26.
- Using these conditions, branch instructions can be written as, 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
```

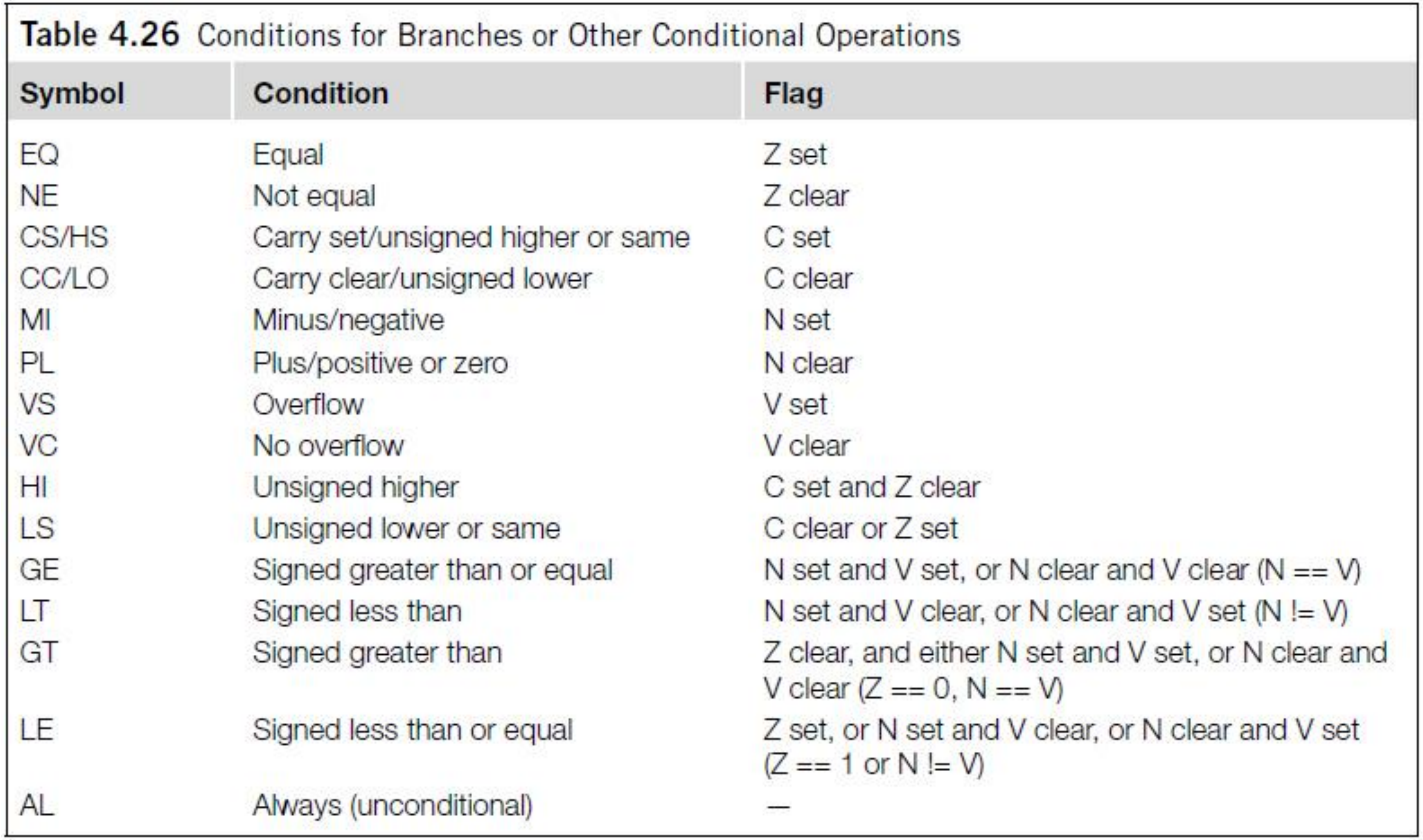
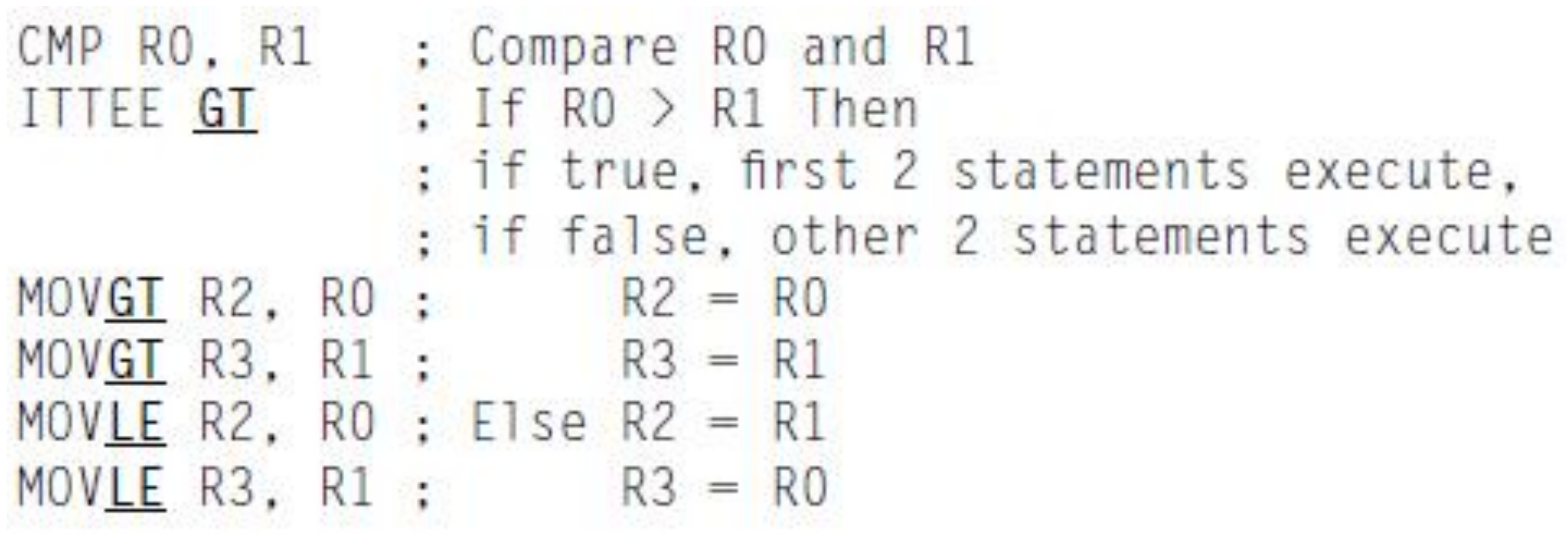


Table 4.26 Conditions for Branches or Other Conditional Operations

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)	—

Assembler Language: Decisions and Conditional Branches (continued)

- 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
```

Assembler Language: Decisions and Conditional Branches (continued)

- 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

Assembler Language: 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.
  - CBZ (compare and branch if zero)
  - CBNZ (compare and branch if nonzero)
- The APSR value is not affected by the CBZ and CBNZ instructions.

Assembler Language: Combined Compare and Conditional Branch (continued)

- 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
```

Assembler Language: Combined Compare and Conditional Branch (continued)

- The usage of CBNZ is similar to CBZ, apart from the fact that the branch is taken if the Z flag is not set (result is not zero). For example,

```
status = strchr(email_address, '@');
if (status == 0){//status is 0 if @ is not in email_address
      show_error_message();
      exit(1);
      }
```

- This can be compiled into the following:

```
    ...
    BL   strchr
    CBNZ R0, email_looks_okay ; Branch if result is not zero
    BL   show_error_message
    BL   exit
email_looks_okay
    ...
```

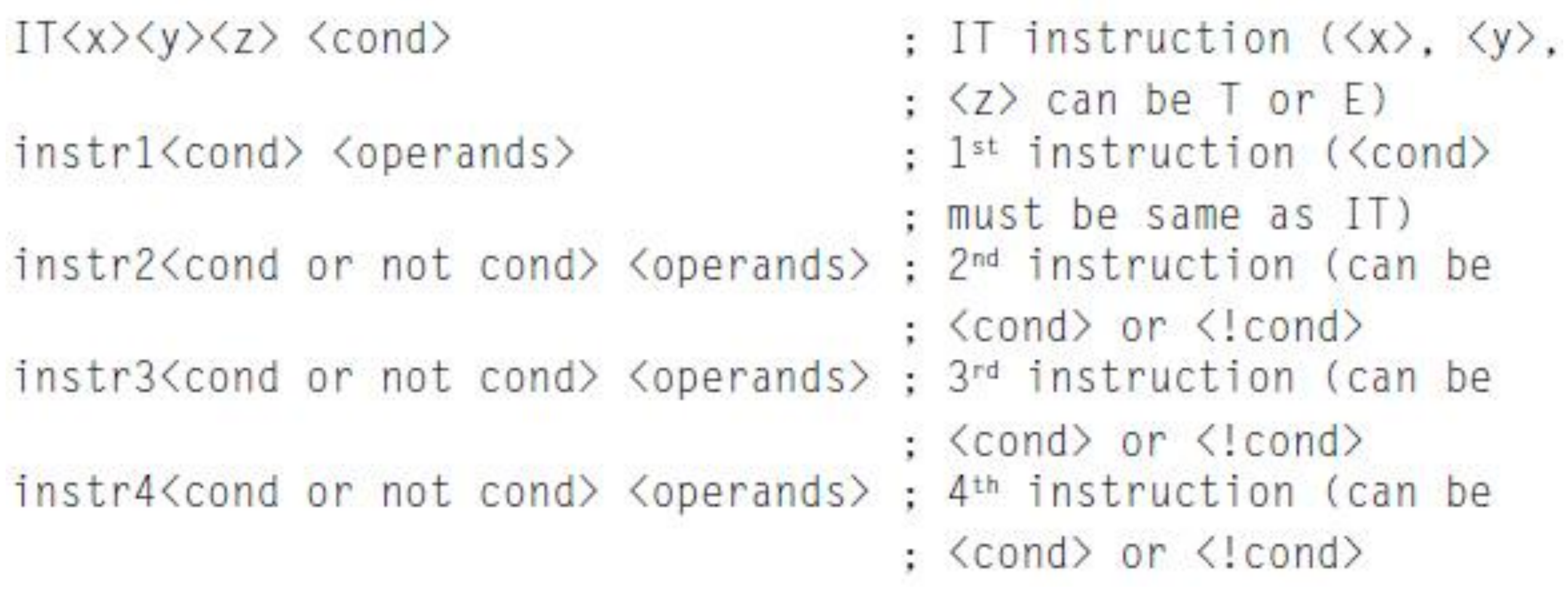
Assembler Language: Conditional Execution Using IT Instructions

- 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.

Assembler Language: Conditional Execution Using IT Instructions (continued)

- 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).

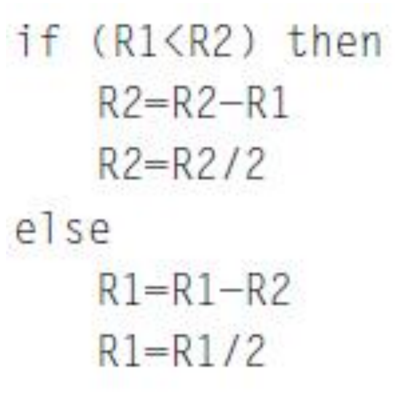
Assembler Language: Conditional Execution Using IT Instructions (continued)



```
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>
```

Assembler Language: Conditional Execution Using IT Instructions (continued)

- 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
```

Assembler Language: Conditional Execution Using IT Instructions (continued)

- 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
```

Assembler Language: Conditional Execution Using IT Instructions (continued)

- 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.

Assembler Language: Instruction Barrier and Memory Barrier Instructions

- Barrier 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, DSB instruction should be used.
  - 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.

Assembler Language: Instruction Barrier and Memory Barrier Instructions (continued)

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

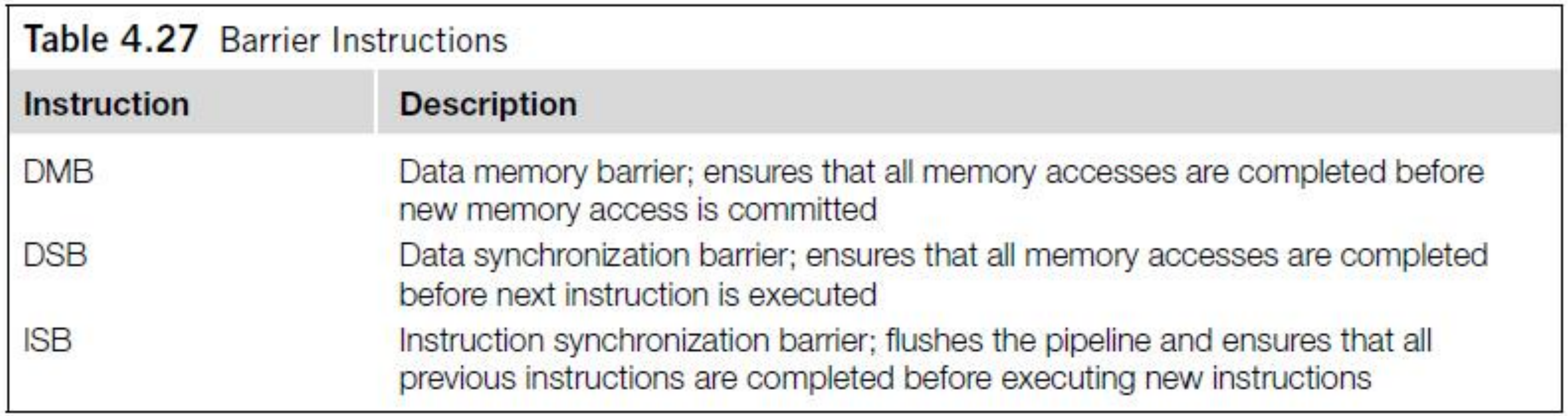


Table 4.27 Barrier Instructions

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

Assembler Language: Instruction Barrier and Memory Barrier Instructions (continued)

- The memory barrier instructions can be accessed in C using Cortex Microcontroller Software Interface Standard (CMSIS) compliant device driver library as follows:

```
void __DMB(void); // Data Memory Barrier
void __DSB(void); // Data Synchronization Barrier
void __ISB(void); // Instruction Synchronization Barrier
```

Assembler Language: Instruction Barrier and Memory Barrier Instructions (continued)

- 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 we want to make sure our application is portable, we should ensure an ISB instruction is used after updating to CONTROL register.

Assembler Language: Instruction Barrier and Memory Barrier Instructions (continued)

- 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.

Assembler Language: Saturation Operations

- The Cortex-M3 supports two instructions that provide signed and unsigned saturation operations:
  - SSAT (for signed data type)
  - USAT (for unsigned data type)
- 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, as shown in Figure 4.3.

Assembler Language: Saturation Operations (continued)

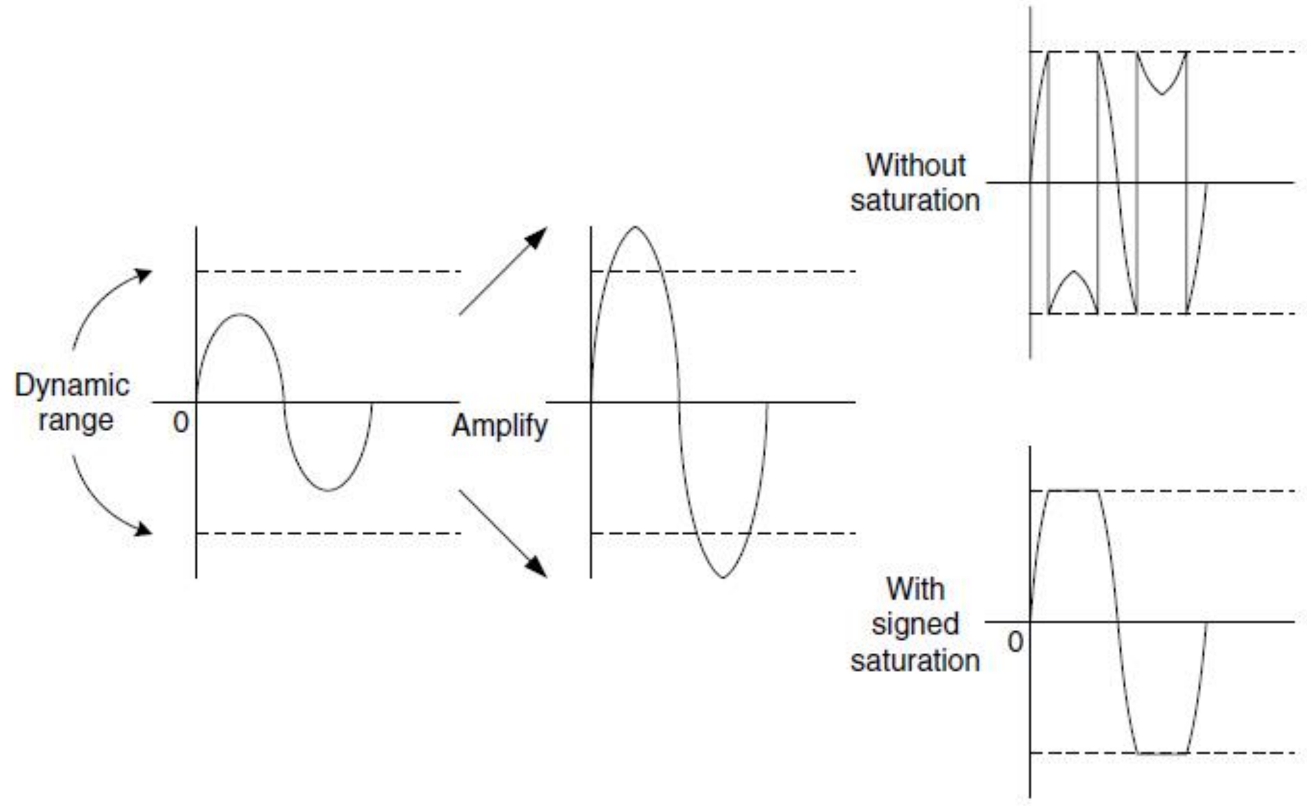


FIGURE 4.3

Signed Saturation Operation.

Assembler Language: Saturation Operations (continued)

- 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.
- The instruction syntax of the SSAT and USAT instructions is as shown in Table 4.28.

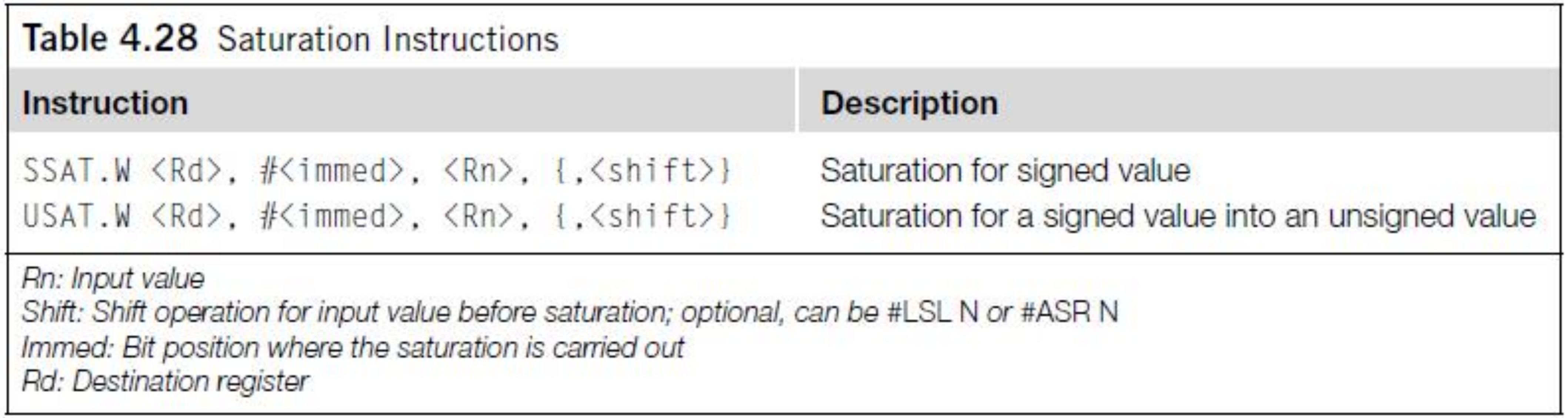


Table 4.28 Saturation Instructions

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

Rn: Input value
Shift: Shift operation for input value before saturation; optional, can be #LSL N or #ASR N
Immed: Bit position where the saturation is carried out
Rd: Destination register

Assembler Language: Saturation Operations (continued)

- 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 (see Table 4.29).

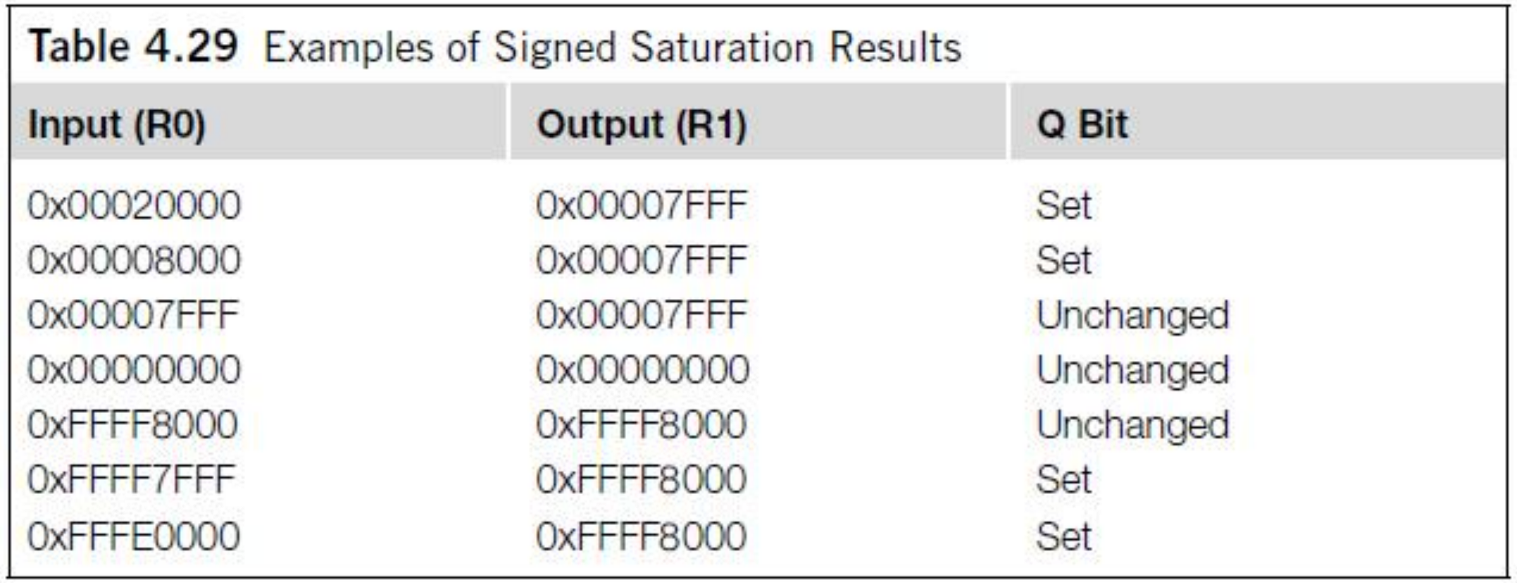


Table 4.29 Examples of Signed Saturation Results

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

Assembler Language: Saturation Operations (continued)

- 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
```

- This will provide a saturation feature that has the properties shown in Figure 4.4.

Assembler Language: Saturation Operations (continued)

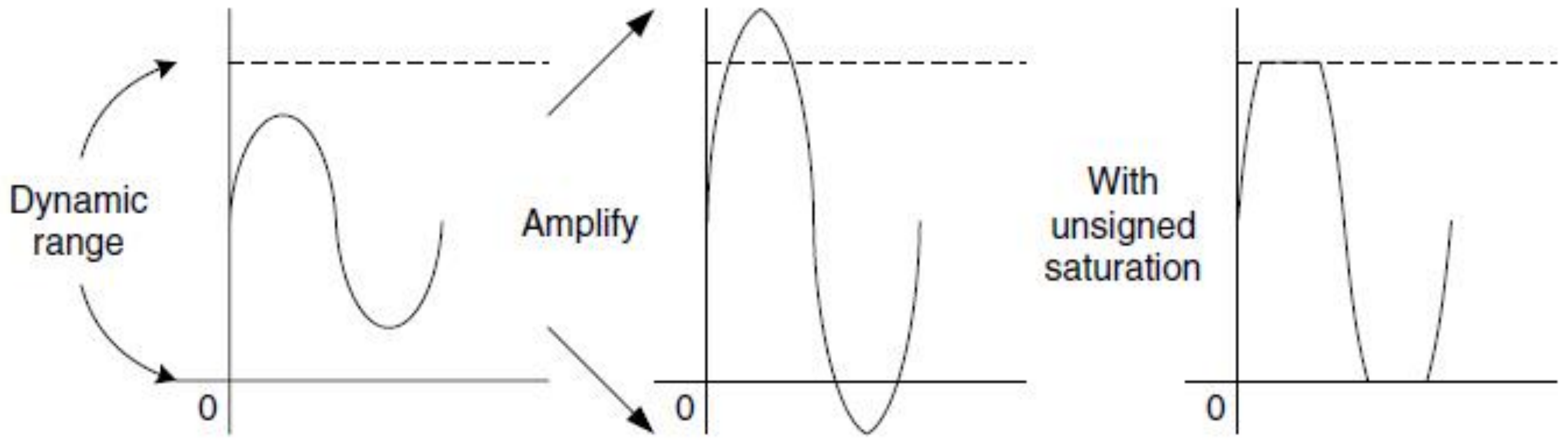


FIGURE 4.4

Unsigned Saturation Operation.

Assembler Language: Saturation Operations (continued)

- For the preceding 16-bit saturation example instruction, the output values shown in Table 4.30 can be observed.

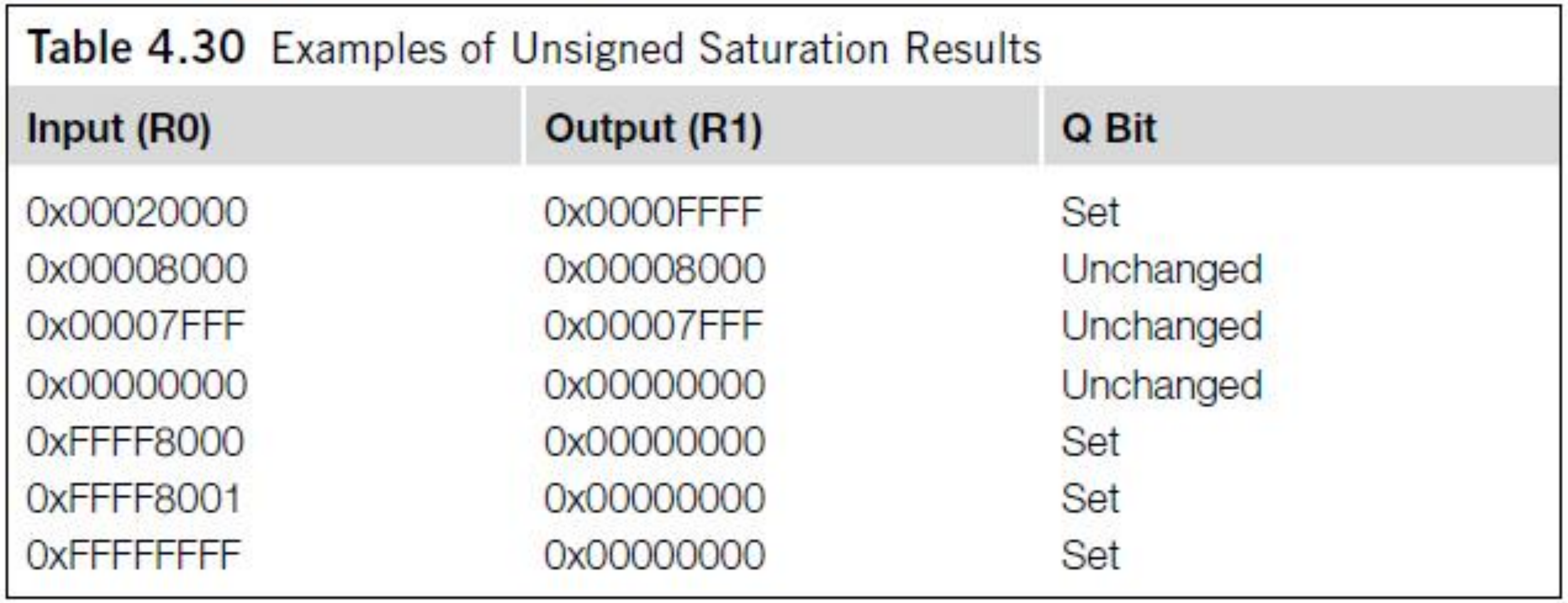


Table 4.30 Examples of Unsigned Saturation Results

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

Several Useful Instructions in the Cortex-M3

MSR and MRS

- MSR and MRS instructions provide access to the special registers in the Cortex-M3.
- Syntax:

```
MRS <Rn>, <SReg> ; Move from Special Register
MSR <SReg>, <Rn> ; Write to Special Register
```

where <*SReg*> could be one of the options shown in Table 4.31.

- For example, the following code can be used to set up the process stack pointer:

```
LDR R0,=0x20008000 ; new value for Process Stack Pointer (PSP)
MSR PSP, R0
```

MSR and MRS (continued)

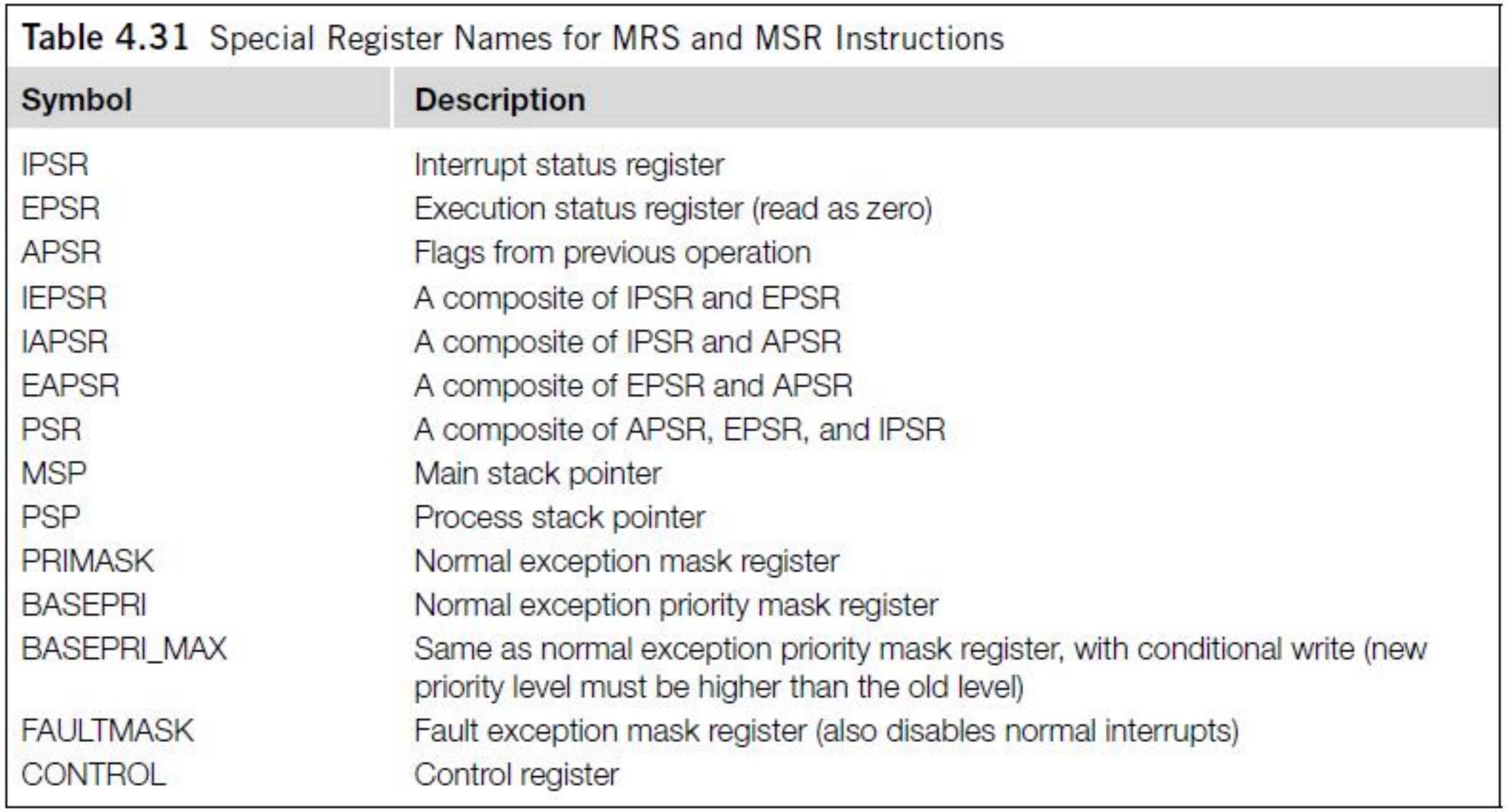


Table 4.31 Special Register Names for MRS and MSR Instructions

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 interrupts)
CONTROL	Control register

More on the IF-THEN Instruction Block

- The IF-THEN (IT) instructions allow up to four succeeding instructions (called an *IT block*) to be conditionally executed.
- They are in the following formats as shown in Table 4.32, where,
  - <*x*> specifies the execution condition for the second instruction
  - <*y*> specifies the execution condition for the third instruction
  - <*z*> specifies the execution condition for the fourth instruction
  - <*cond*> specifies the base condition of the instruction block; the first instruction following IT executes if <*cond*> is true

More on the IF-THEN Instruction Block (continued)

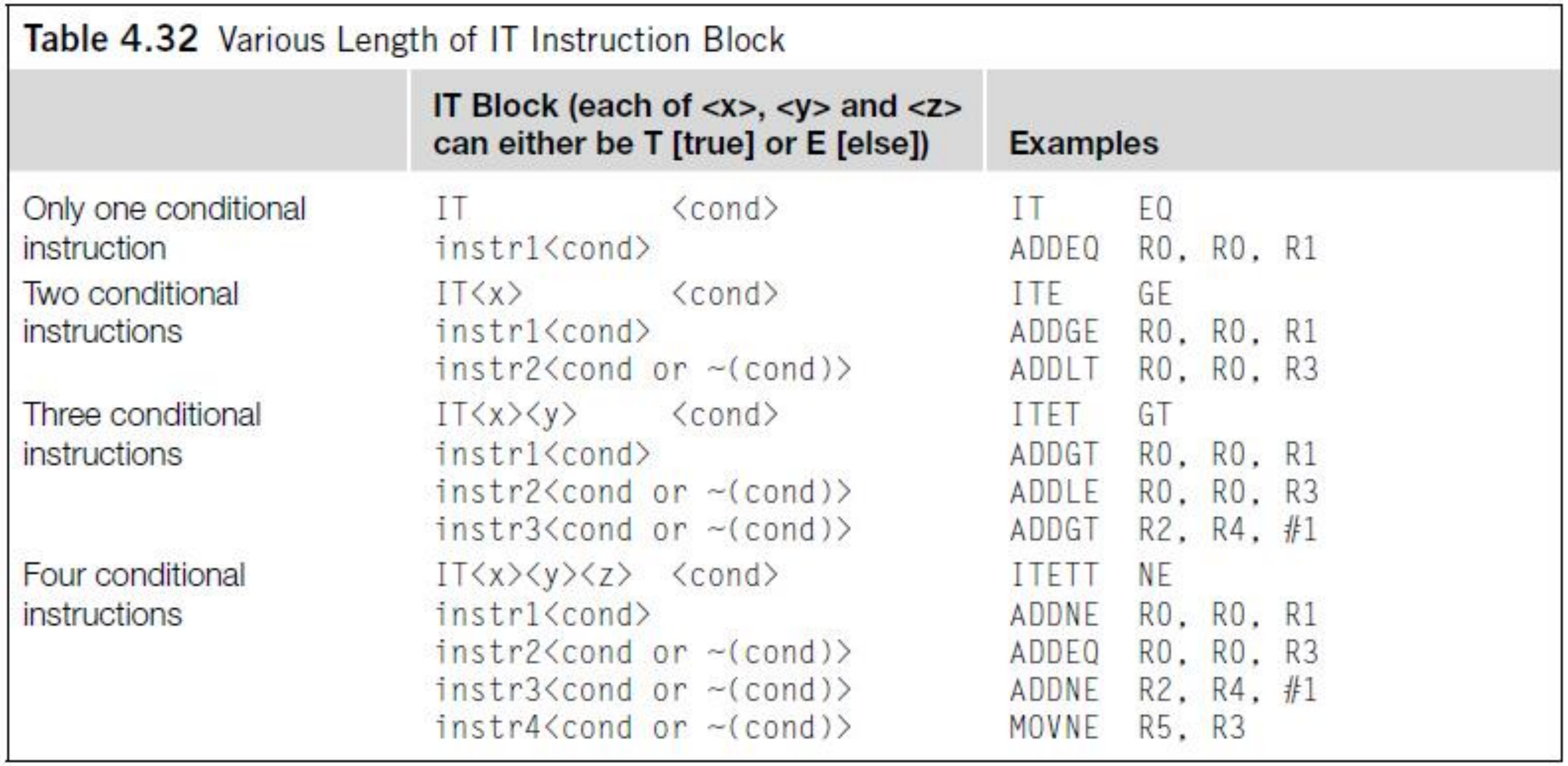


Table 4.32 Various Length of IT Instruction Block

	IT Block (each of <x>, <y> and <z> can either be T [true] or E [else])	**Examples**

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 ~(cond)>`
		`ITE GE`
		`ADDGE R0, R0, R1`
		`ADDLT R0, R0, R3`

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

More on the IF-THEN Instruction Block (continued)

- The *<cond>* part uses the same condition symbols as conditional branch.
  - If “AL” is used as *<cond>*, then you cannot use “E” in the condition control as it implies the instruction should never get executed.
- Each of *<x>*, *<y>*, and *<z>* can be either *T* (THEN) or *E* (ELSE), which refers to the base condition *<cond>*, whereas *<cond>* uses traditional syntax such as EQ, NE, GT, or the like.

More on the IF-THEN Instruction Block (continued)

- Here is an example of IT use:

```
if (R0 equal R1) then {
  R3 = R4 + R5
  R3 = R3/2
  } else {
  R3 = R6 + R7
  R3 = R3/2
  }
```

- This can be written as follows:

```
CMP R0, R1         ; Compare R0 and R1
ITTEE EQ           ; If R0 equal R1, Then-Then-Else-Else
ADDEQ R3, R4, R5   ; Add if equal
ASREQ R3, R3, #1   ; Arithmetic shift right if equal
ADDNE R3, R6, R7   ; Add if not equal
ASRNE R3, R3, #1   ; Arithmetic shift right if not equal
```

SDIV and UDIV

- The syntax for signed and unsigned divide instructions is as follows:

```
SDIV.W <Rd>, <Rn>, <Rm>
UDIV.W <Rd>, <Rn>, <Rm>
```

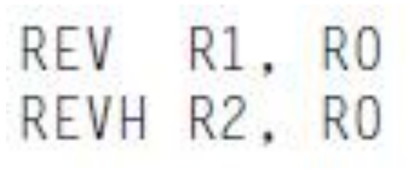
- The result is Rd = Rn/Rm. For example,

```
LDR     R0,=300 ; Decimal 300
MOV     R1,#5
UDIV.W R2, R0, R1
```

- This will give you an R2 result of 60 (0x3C).

REV, REVH, and REVSH

- REV reverses the byte order in a data word, and REVH reverses the byte order inside a half word.
- For example, if R0 is 0x12345678,



```
REV  R1, R0
REVH R2, R0
```

- After executing the above instructions, R1 will become 0x78563412, and R2 will be 0x34127856.
- REV and REVH are particularly useful for converting data between big endian and little endian.

REV, REVH, and REVSH (continued)

- REVSH is similar to REVH except that it only processes the lower half word, and then it sign extends the result.
- For example, if R0 is 0x33448899,

```
REVSH R1, R0
```

- After executing the above instruction, R1 will become 0xFFFF9988.

Reverse Bit

- The RBIT instruction reverses the bit order in a data word. The syntax is as follows:

```
RBIT.W <Rd>, <Rn>
```

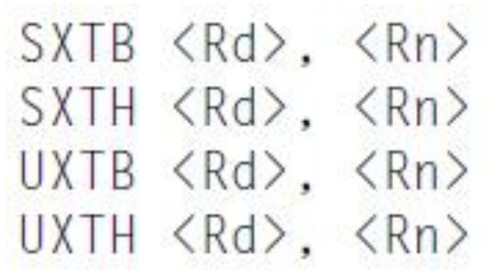
- This instruction is very useful for processing serial bit streams in data communications. For example, if R0 is 0xB4E10C23 (binary value 1011_0100_1110_0001_0000_1100_0010_0011), then,

```
RBIT.W R0, R1
```

- After executing above instruction, R0 will become 0xC430872D (binary value 1100_0100_0011_0000_1000_0111_0010_1101).

SXTB, SXTH, UXTB, and UXTH

- The four instructions SXTB, SXTH, UXTB, and UXTH are used to extend a byte or half word data into a word.
- The syntax of the instructions is as follows:

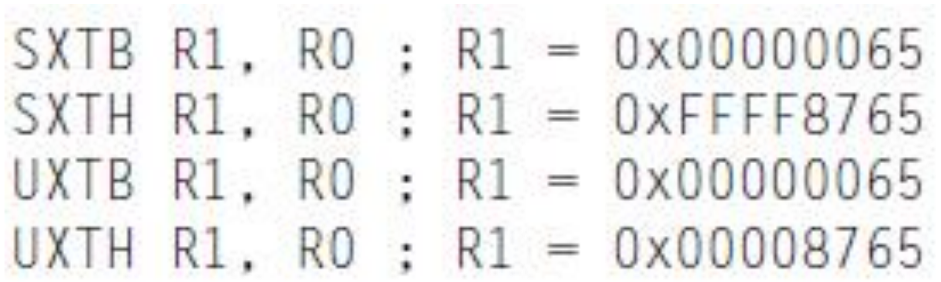


```
SXTB <Rd>, <Rn>
SXTH <Rd>, <Rn>
UXTB <Rd>, <Rn>
UXTH <Rd>, <Rn>
```

- For SXTB/SXTH, the data are sign extended using bit[7]/bit[15] of Rn.
- With UXTB and UXTH, the value is zero extended to 32-bit.

SXTB, SXTH, UXTB, and UXTH (continued)

- For example, if R0 is 0x55AA8765:



```
SXTB R1, R0 ; R1 = 0x00000065
SXTH R1, R0 ; R1 = 0xFFFF8765
UXTB R1, R0 ; R1 = 0x00000065
UXTH R1, R0 ; R1 = 0x00008765
```

Bit Field Clear and Bit Field Insert

- Bit Field Clear (BFC) clears 1–31 adjacent bits in any position of a register.
- The syntax of the instruction is as follows:

```
BFC.W <Rd>, <#lsb>, <#width>
```

- For example,

```
LDR   R0,=0x1234FFFF
BFC.W R0, #4, #8
```

- This will give R0 = 0x1234F00F.

Bit Field Clear and Bit Field Insert (continued)

- Bit Field Insert (BFI) copies 1–31 bits (#width) from one register to any location (#lsb) in another register.
- The syntax is as follows:

```
BFI.W <Rd>, <Rn>, <#lsb>, <#width>
```

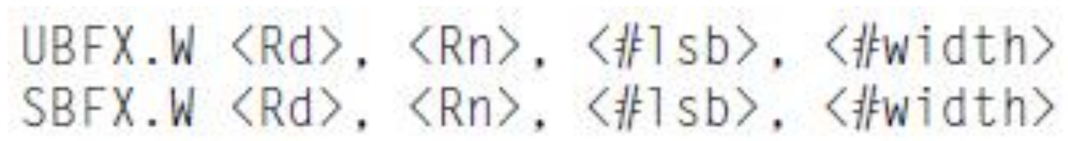
- For example,

```
LDR   R0,=0x12345678
LDR   R1,=0x3355AACC
BFI.W R1, R0, #8, #16 ; Insert R0[15:0] to R1[23:8]
```

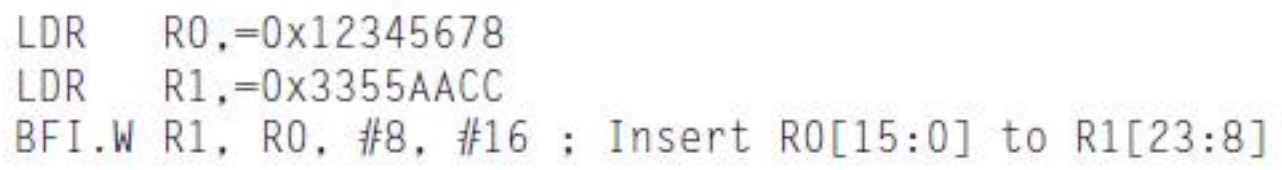
- This will give R1 = 0x335678CC.

UBFX and SBFX

- UBFX and SBFX are the unsigned and signed bit field extract instructions.
- The syntax of the instructions is as follows:



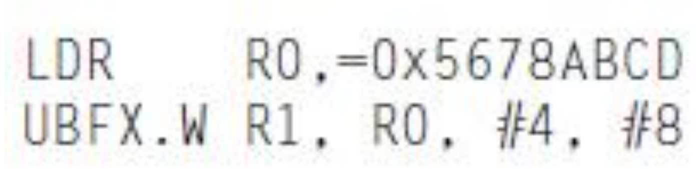
```
UBFX.W <Rd>, <Rn>, <#lsb>, <#width>
SBFX.W <Rd>, <Rn>, <#lsb>, <#width>
```



```
LDR   R0,=0x12345678
LDR   R1,=0x3355AACC
BFI.W R1, R0, #8, #16 ; Insert R0[15:0] to R1[23:8]
```

UBFX and SBFX (continued)

- UBFX extracts a bit field from a register starting from any location (specified by #lsb) with any width (specified by #width), zero extends it, and puts it in the destination register.
- For example,

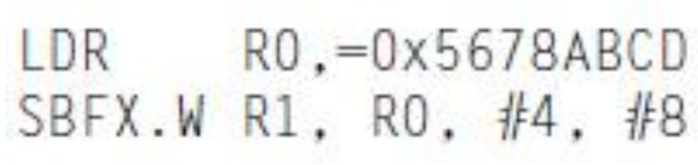


```
LDR     R0,=0x5678ABCD
UBFX.W R1, R0, #4, #8
```

- This will give R1 = 0x000000BC.

UBFX and SBFX (continued)

- Similarly, SBFX extracts a bit field, but its sign extends it before putting it in a destination register.
- For example,

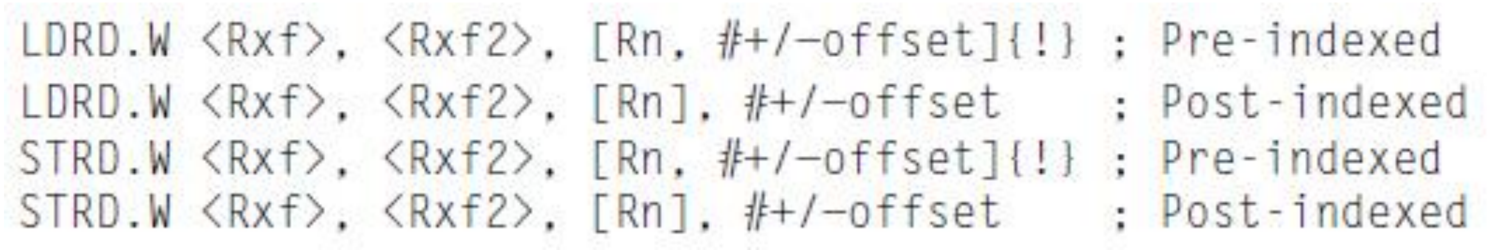


```
LDR     R0,=0x5678ABCD
SBFX.W  R1, R0, #4, #8
```

- This will give R1 = 0xFFFFFFBC.

LDRD and STRD

- The two instructions LDRD and STRD transfer two words of data from or into two registers.
- The syntax of the instructions is as follows:



```
LDRD.W <Rxf>, <Rxf2>, [Rn, #+/-offset]{!} ; Pre-indexed
LDRD.W <Rxf>, <Rxf2>, [Rn], #+/-offset    ; Post-indexed
STRD.W <Rxf>, <Rxf2>, [Rn, #+/-offset]{!} ; Pre-indexed
STRD.W <Rxf>, <Rxf2>, [Rn], #+/-offset    ; Post-indexed
```

where <*Rxf*> is the first destination/source register and <*Rxf2*> is the second destination/source register.

LDRD and STRD (continued)

- For example, the following code reads a 64-bit value located in memory address 0x1000 into R0 and R1:

```
LDR    R2,=0x1000
LDRD.W R0, R1, [R2] ; This will gives R0 = memory[0x1000],
                    ; R1 = memory[0x1004]
```

- Similarly, we can use STRD to store a 64-bit value in memory.
- In the following example, preindexed addressing mode is used:

```
LDR    R2,=0x1000          ; Base address
STRD.W R0, R1, [R2, #0x20] ; This will gives memory[0x1020] = R0,
                           ; memory[0x1024] = R1
```

Table Branch Byte and Table Branch Halfword

- Table Branch Byte (TBB) and Table Branch Halfword (TBH) are for implementing branch tables.
- The TBB instruction uses a branch table of byte size offset, and TBH uses a branch table of half word offset.
- Since the bit 0 of a program counter is always zero, the value in the branch table is multiplied by two before it’s added to PC.
- Furthermore, because the PC value is the current instruction address plus four, the branch range for TBB is $(2 \times 255) + 4 = 514$, and the branch range for TBH is $(2 \times 65535) + 4 = 131074$.
- Both TBB and TBH support forward branch only.

Table Branch Byte and Table Branch Halfword (continued)

- TBB has this general syntax:

```
TBB.W [Rn, Rm]
```

where Rn is the base memory offset and Rm is the branch table index.

- The branch table item for TBB is located at Rn + Rm.
- Assuming we used PC for Rn, we can see the operation as shown in Figure 4.5.

Table Branch Byte and Table Branch Halfword (continued)

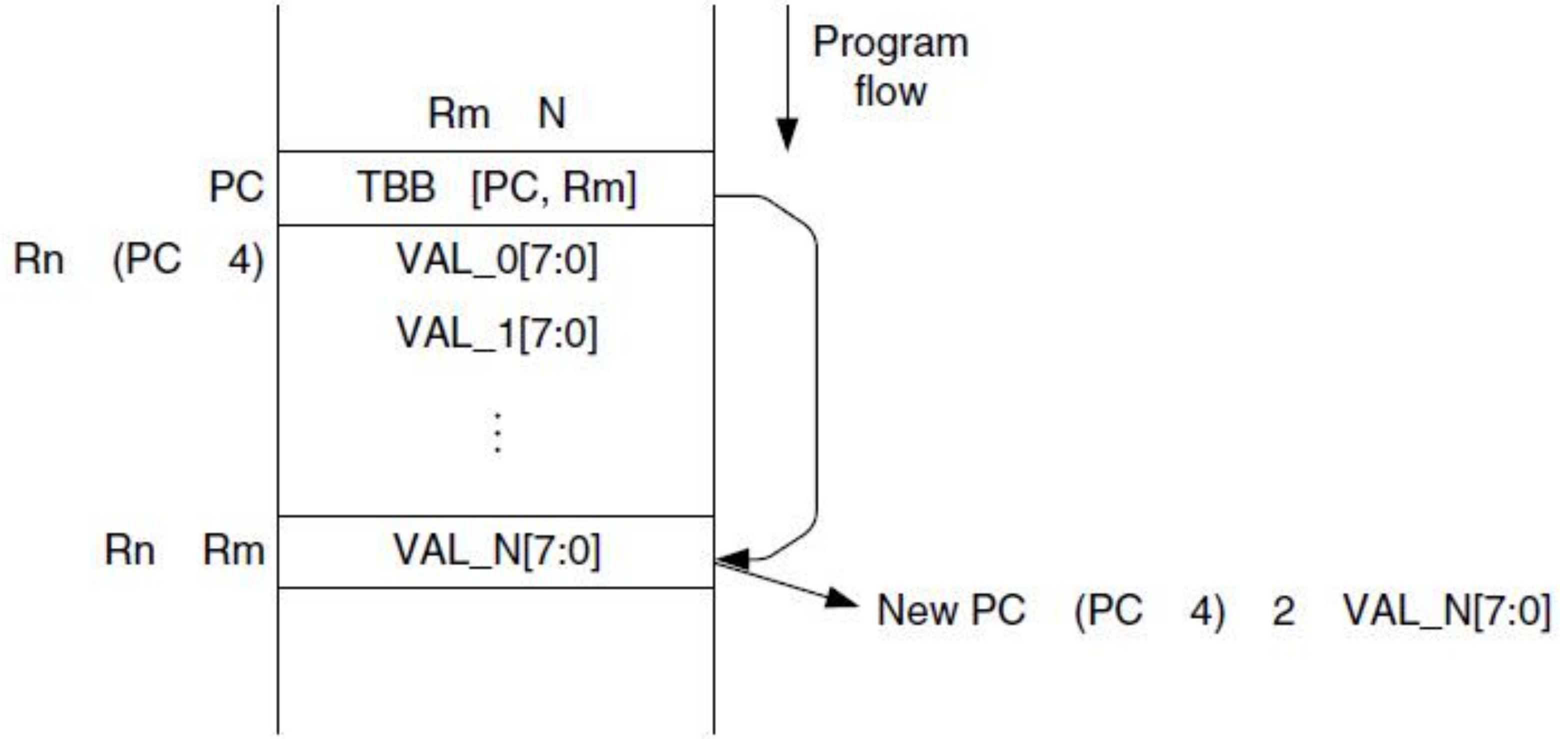


FIGURE 4.5

TBB Operation.

Table Branch Byte and Table Branch Halfword (continued)

- For TBH instruction, the process is similar except the memory location of the branch table item is located at Rn + 2 x Rm and the maximum branch offset is higher.
- Again, we assume that Rn is set to PC, as shown in Figure 4.6.

Table Branch Byte and Table Branch Halfword (continued)

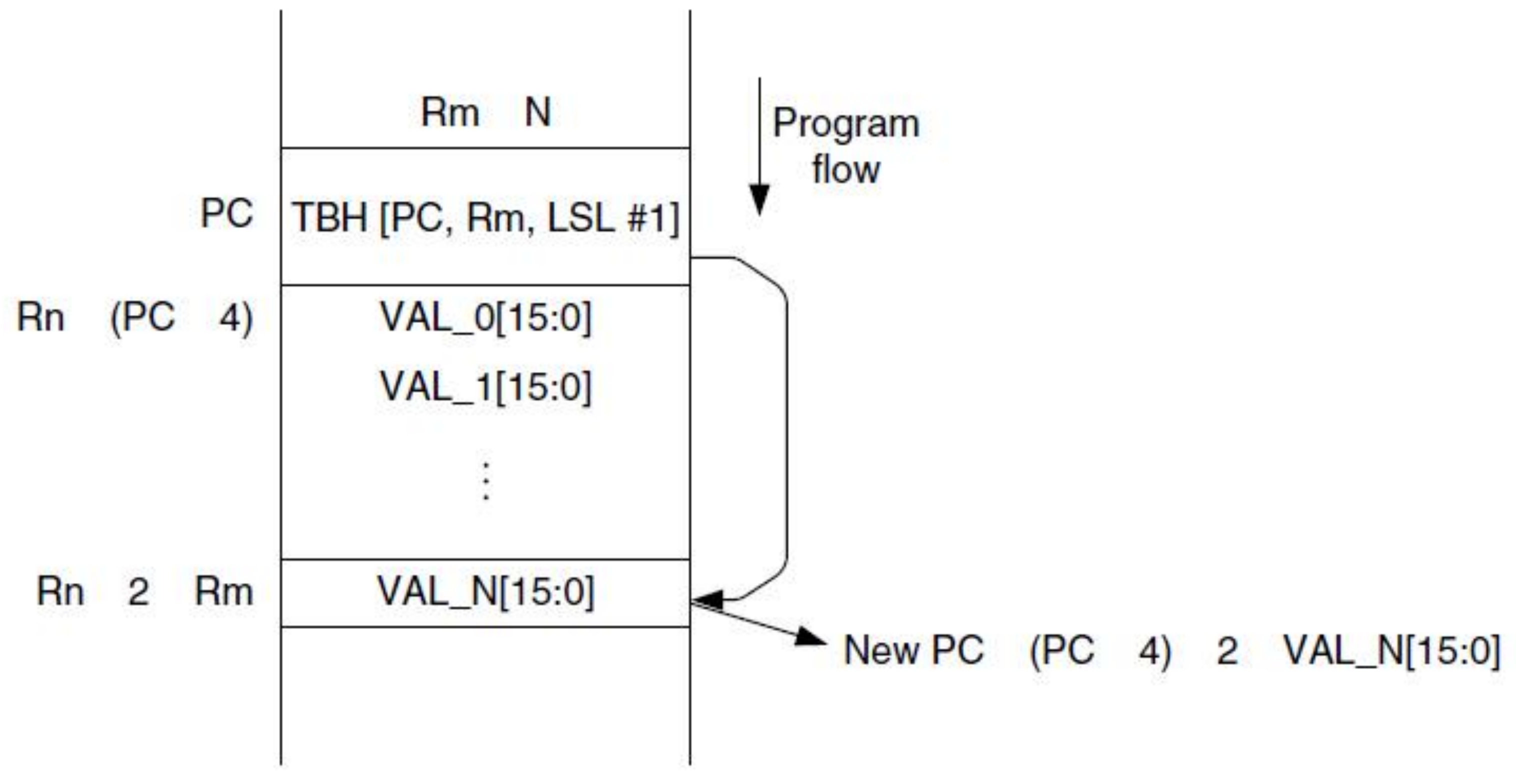


FIGURE 4.6

TBH Operation.

Table Branch Byte and Table Branch Halfword (continued)

- If *Rn* in the table branch instruction is set to R15, the value used for *Rn* will be PC + 4 because of the pipeline in the processor.
- The coding syntax for calculating TBB/TBH branch table content could be dependent on the development tool.
- When the TBB instruction is executed, the current PC value is at the address labeled as *branchtable* (because of the pipeline in the processor).

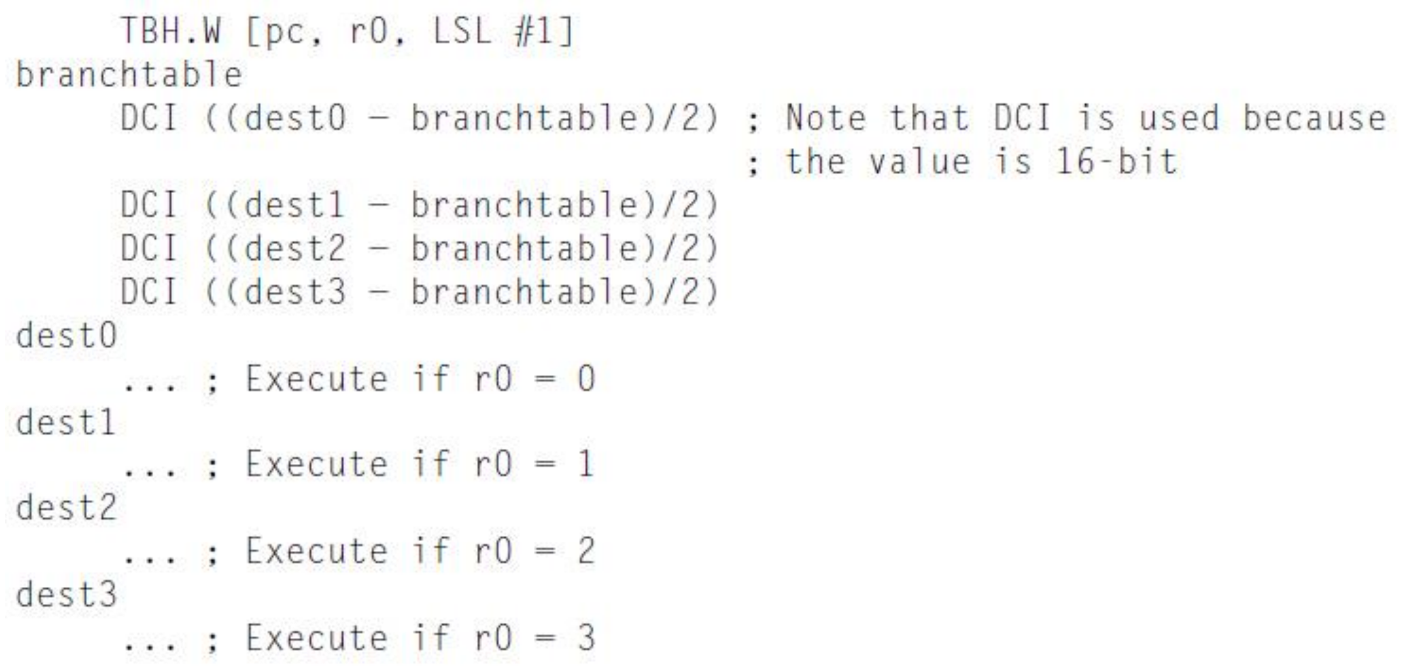
Table Branch Byte and Table Branch Halfword (continued)

- In ARM assembler (*armasm*), the TBB branch table can be created in the following way:

```
     TBB.W [pc, r0] ; when executing this instruction, PC equal
                    ; branchtable
branchtable
     DCB ((dest0 - branchtable)/2) ; Note that DCB is used because
                                   ; the value is 8-bit
     DCB ((dest1 - branchtable)/2)
     DCB ((dest2 - branchtable)/2)
     DCB ((dest3 - branchtable)/2)
dest0
     ... ; Execute if r0 = 0
dest1
     ... ; Execute if r0 = 1
dest2
     ... ; Execute if r0 = 2
dest3
     ... ; Execute if r0 = 3
```

Table Branch Byte and Table Branch Halfword (continued)

- Similarly, for TBH instructions, it can be used as follows:



```
        TBH.W [pc, r0, LSL #1]
branchtable
        DCI ((dest0 - branchtable)/2) ; Note that DCI is used because
                                      ; the value is 16-bit
        DCI ((dest1 - branchtable)/2)
        DCI ((dest2 - branchtable)/2)
        DCI ((dest3 - branchtable)/2)
dest0
        ... ; Execute if r0 = 0
dest1
        ... ; Execute if r0 = 1
dest2
        ... ; Execute if r0 = 2
dest3
        ... ; Execute if r0 = 3
```

Cortex-M3 Programming

Overview

- 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.
  - However, some people prefer to use assembly language or a combination of C and assembly language in their projects.
- The procedure of building and downloading the resultant image files to the target device is largely dependent on the tool chain used.

A Typical Development Flow

- Various software programs are available for developing Cortex-M3 applications.
- The concepts of code generation flow in terms of these tools are similar.
- For the most basic uses, we will need assembler, a C compiler, a linker, and binary file generation utilities.
- For ARM solutions, the RealView Development Suite (RVDS) or RealView Compiler Tools (RVCT) provide a file generation flow, as shown in Figure 10.1.

A Typical Development Flow (continued)

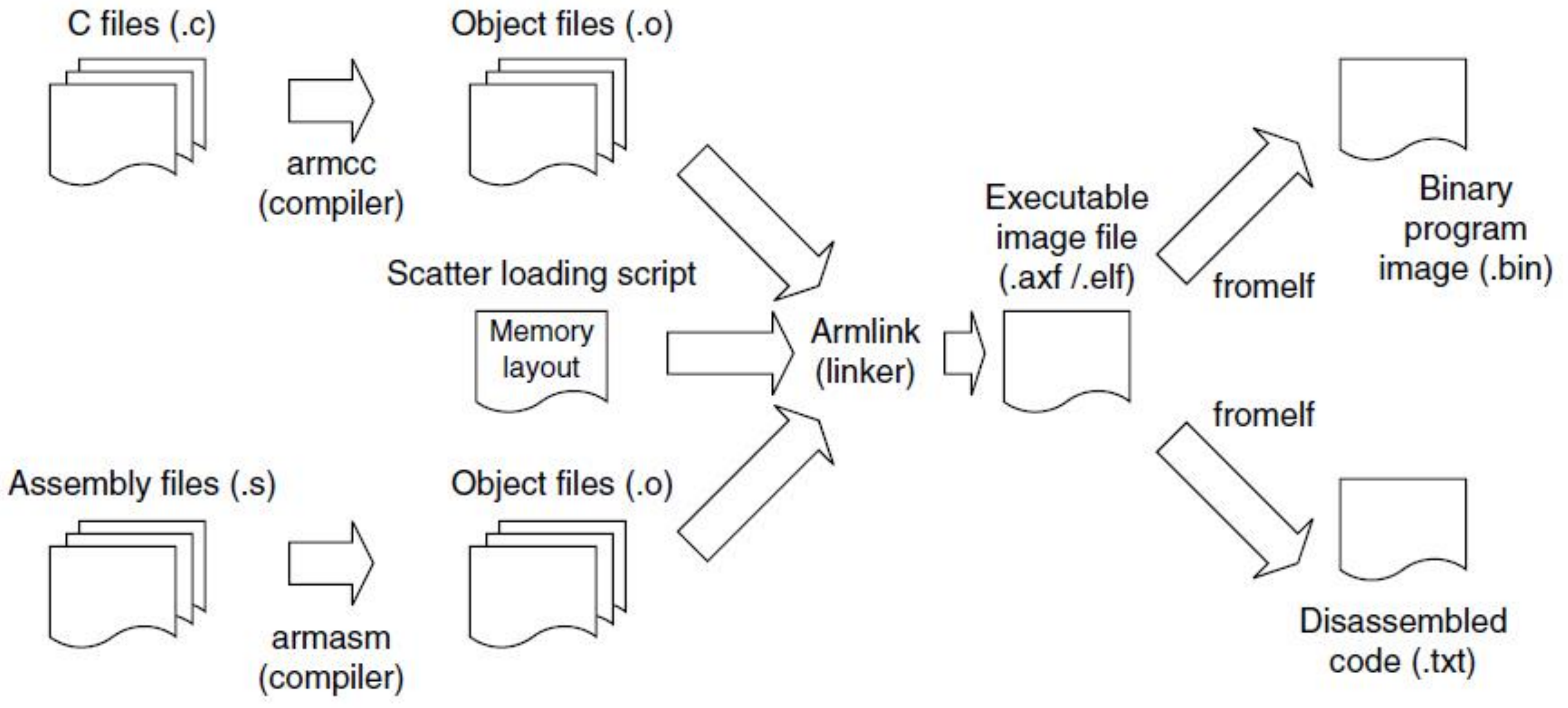


FIGURE 10.1

Example Flow Using ARM Development Tools.

A Typical Development Flow (continued)

- The scatter-loading script is optional but often required when the memory map becomes more complex.
- Besides these basic tools, RVDS also contains a large number of utilities, including an Integrated Development Environment (IDE) and debuggers.

Using C

Using C

- For beginners in embedded programming, using C language for software development on the Cortex-M3 processor is the best choice.
- Programming in C with the Cortex-M3 processor is made even easier as most microcontroller vendors provide device driver libraries written in C to control peripherals.
  - These can then be included into the project.
- Since modern C compilers can generate very efficient code, it is better to program in C than spending a lot of time to try to develop complex routines in assembly language, which is error prone and less portable.

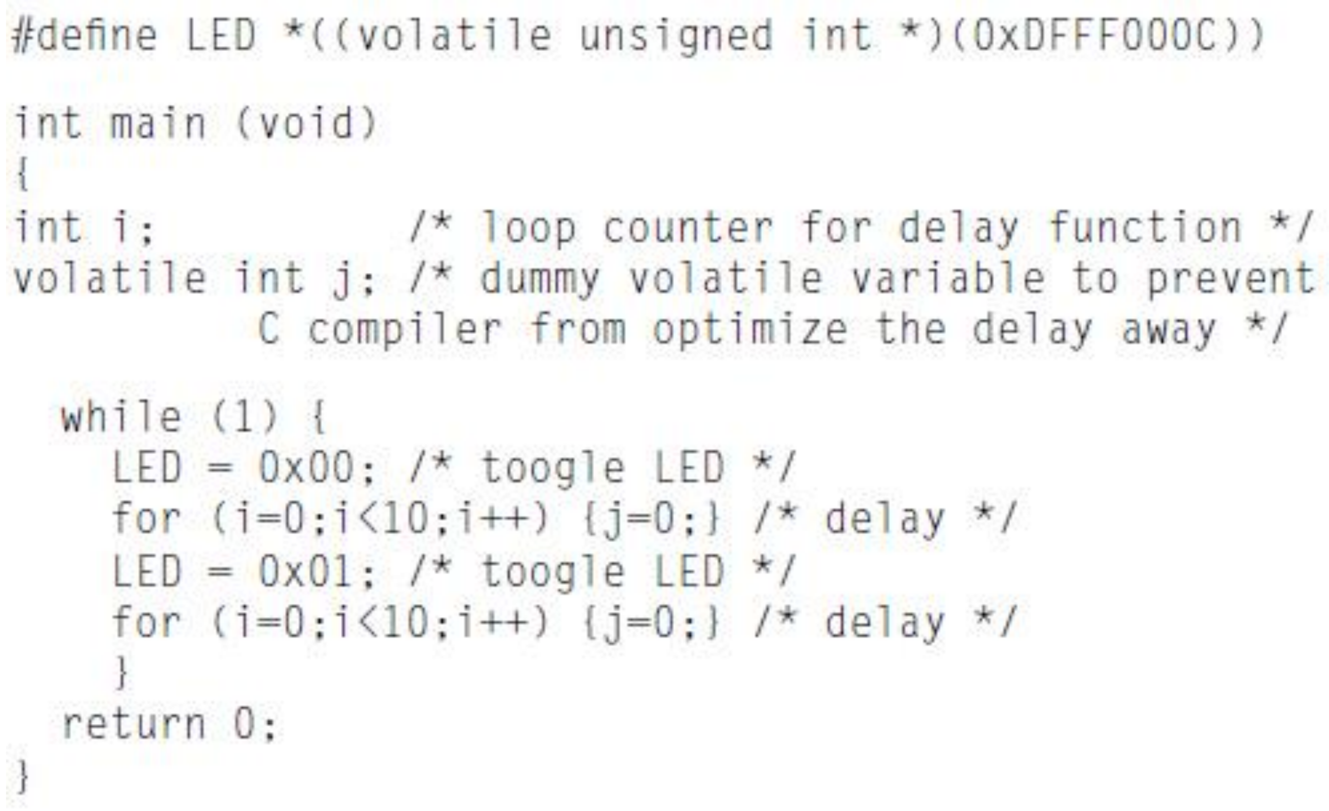
Using C (continued)

- C has the advantage of being portable and easier for implementing complex operations, compared with assembly language.
- Since it’s a generic computer language, C does not specify how the processor is initialized.
  - For these areas, tool chains can have different approaches.
  - The best way to get started is to look at example codes.
  - For users of ARM C compiler products, such as RVDS or Keil RealView Microcontroller Development Kit (MDK-ARM), a number of Cortex-M3 program examples are already included in the installation.

Example of a Simple C Program Using RealView Development Site

- A normal program for the Cortex-M3 contains at least the “main” program and a vector table.
- Let’s start with the most basic main program that toggles an Light Emitting Diode (LED):

Example of a Simple C Program Using RealView Development Site (continued)



```
#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; /* toogle LED */
    for (i=0;i<10;i++) {j=0;} /* delay */
    LED = 0x01; /* toogle LED */
    for (i=0;i<10;i++) {j=0;} /* delay */
    }
  return 0;
}
```

Example of a Simple C Program Using RealView Development Site (continued)

- This file is named “blinky.c.”
- For the vector table, we create a separate C program called “vectors.c.”
- The file “vectors.c” contains the vector table, as well as a number of dummy exception handlers (these can be customized for target application later on):

```
typedef void(* const ExecFuncPtr)(void) __irq;
extern int __main(void);

/*
 * Dummy handlers Exception Handlers
 */
__irq void NMI_Handler(void)
{   while(1); }
__irq void HardFault_Handler(void)
{   while(1); }
__irq void SVC_Handler(void)
{   while(1); }
__irq void DebugMon_Handler(void)
{   while(1); }
__irq void PendSV_Handler(void)
{   while(1); }
__irq void SysTick_Handler(void)
{   while(1); }
__irq void ExtInt0_IRQHandler(void)
{   while(1); }
__irq void ExtInt1_IRQHandler(void)
{   while(1); }
__irq void ExtInt2_IRQHandler(void)
{   while(1); }
__irq void ExtInt3_IRQHandler(void)
{   while(1); }
```

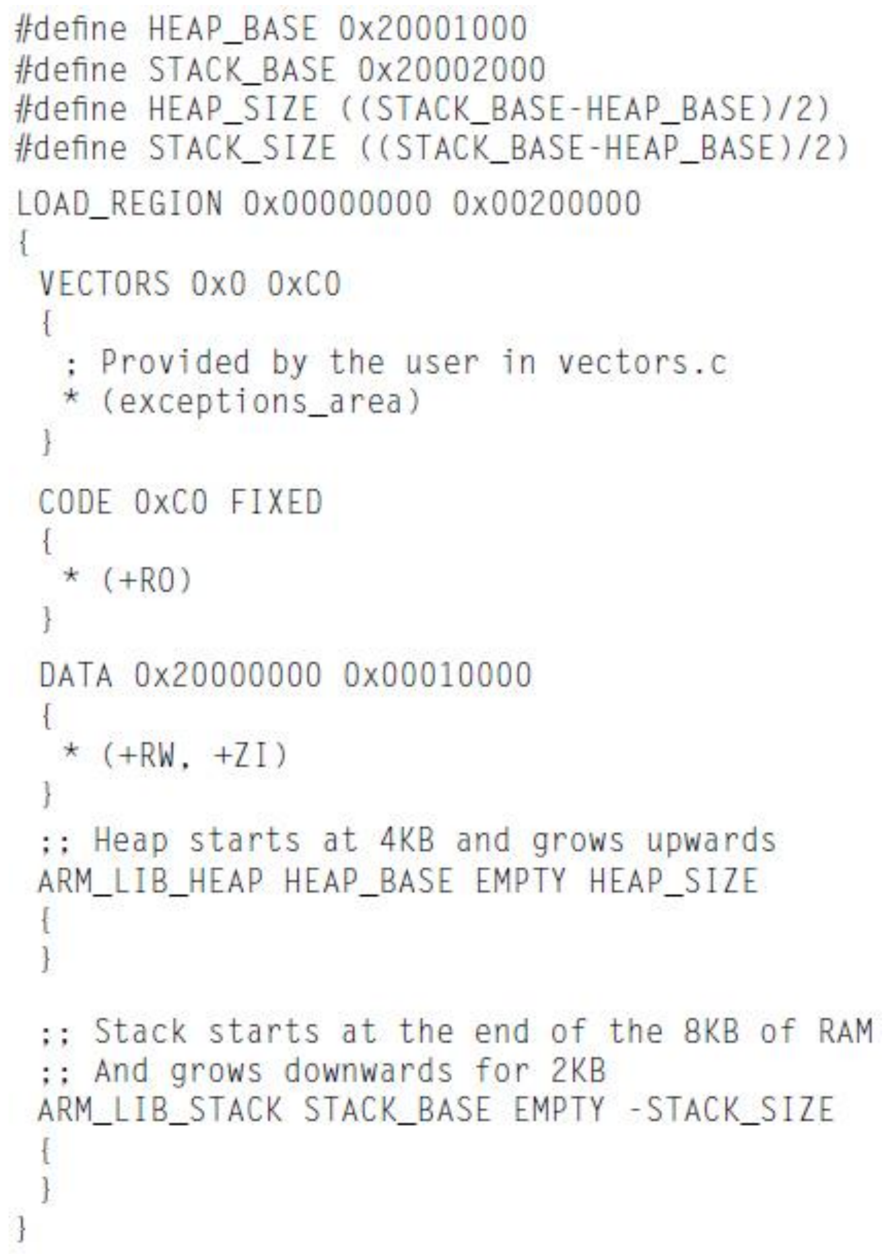
```
#pragma arm section rodata="exceptions_area"
ExecFuncPtr exception_table[] = { /* vector table */
    (ExecFuncPtr)0x20002000,
    (ExecFuncPtr)__main,
    NMI_Handler, /* NMI */
    HardFault_Handler,
    0, /* MemManage_Handler in Cortex-M3 */
    0, /* BusFault_Handler in Cortex-M3 */
    0, /* UsageFault_Handler in Cortex-M3 */

    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    0, /* Reserved */
    SVC_Handler,
    0, /* DebugMon_Handler in Cortex-M3 */
    0, /* Reserved */
    PendSV_Handler,
    SysTick_Handler,

    /* External Interrupts*/
    ExtInt0_IRQHandler,
    ExtInt1_IRQHandler,
    ExtInt2_IRQHandler,
    ExtInt3_IRQHandler
};
#pragma arm section
```

Example of a Simple C Program Using RealView Development Site (continued)

- Assuming you are using RVDS, you can compile the program using the following command line:

```
$> armcc -c -g -W blinky.c -o blinky.o
$> armcc -c -g -W vectors.c -o vectors.o
```

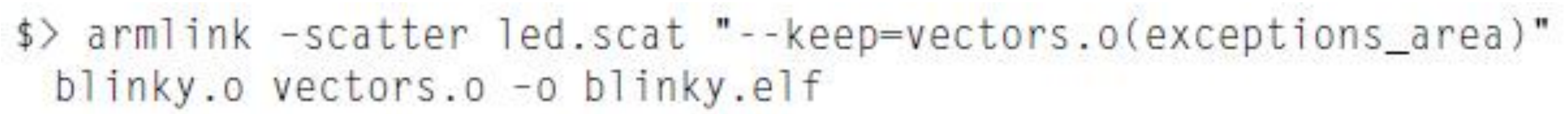
- Then the linker can be used to generate the program image.
- A scatter loading file “led.scat” is used to tell the linker the memory layout and to put the vector table in the starting of the program image.
- The “led.scat” is:



```
#define HEAP_BASE 0x20001000
#define STACK_BASE 0x20002000
#define HEAP_SIZE ((STACK_BASE-HEAP_BASE)/2)
#define STACK_SIZE ((STACK_BASE-HEAP_BASE)/2)
LOAD_REGION 0x00000000 0x00200000
{
 VECTORS 0x0 0xC0
 {
  ; Provided by the user in vectors.c
  * (exceptions_area)
 }
 CODE 0xC0 FIXED
 {
  * (+RO)
 }
 DATA 0x20000000 0x00010000
 {
  * (+RW, +ZI)
 }
 ;; Heap starts at 4KB and grows upwards
 ARM_LIB_HEAP HEAP_BASE EMPTY HEAP_SIZE
 {
 }

 ;; Stack starts at the end of the 8KB of RAM
 ;; And grows downwards for 2KB
 ARM_LIB_STACK STACK_BASE EMPTY -STACK_SIZE
 {
 }
}
```

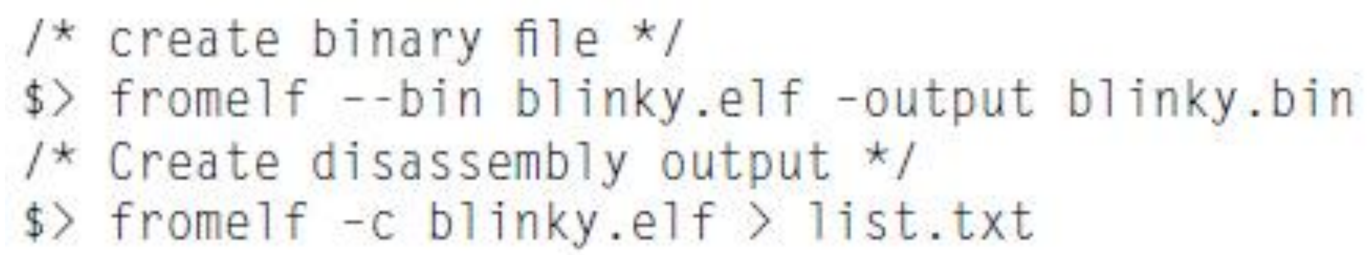
Example of a Simple C Program Using RealView Development Site (continued)

- And the command line for the linker is



```
$> armlink -scatter led.scat "--keep=vectors.o(exceptions_area)"
  blinky.o vectors.o -o blinky.elf
```

- The executable image “blinky.elf” is now generated.
- We can convert it to binary file and disassembly file using *fromelf*.



```
/* create binary file */
$> fromelf --bin blinky.elf -output blinky.bin
/* Create disassembly output */
$> fromelf -c blinky.elf > list.txt
```

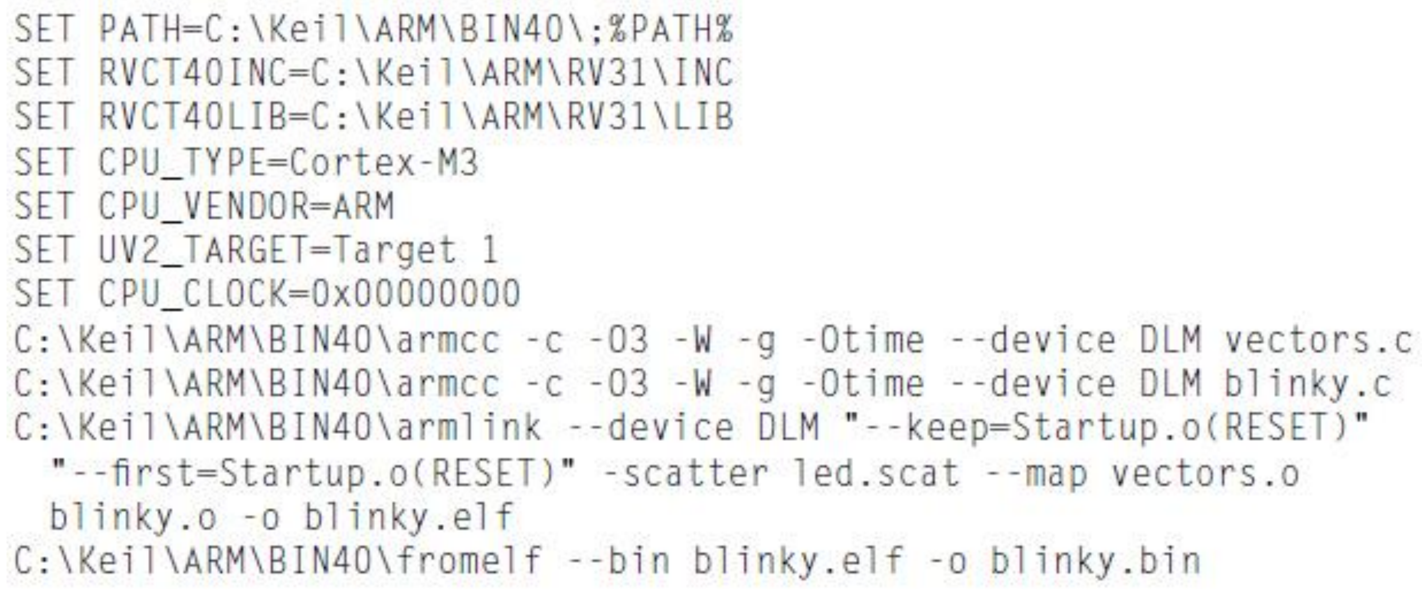
Compile the Same Example Using Keil MDK-ARM

- For users of Keil MDK-ARM, it is possible to compile the same program as in RVDS.
- However, the command line options and a few symbols in the linker script (scatter loading file) have to be modified.
- Based on the example in the previous section, scatter loading file “led.scat” is needed to be modified to

```
#define HEAP_BASE 0x20001000
#define STACK_BASE 0x20002000
#define HEAP_SIZE ((STACK_BASE-HEAP_BASE)/2)
#define STACK_SIZE ((STACK_BASE-HEAP_BASE)/2)

LOAD_REGION 0x00000000 0x00200000
{
 VECTORS 0x0 0xC0
 {
   ; Provided by the user in vectors.c
   * (exceptions_area)
 }

 CODE 0xC0 FIXED
 {
   * (+RO)
 }

 DATA 0x20000000 0x00010000
 {
   * (+RW, +ZI)
 }

 ;; Heap starts at 4KB and grows upwards
 Heap_Mem HEAP_BASE EMPTY HEAP_SIZE
 {
 }

 ;; Stack starts at the end of the 8KB of RAM
 ;; And grows downwards for 2KB
 Stack_Mem STACK_BASE EMPTY -STACK_SIZE
 {
 }
}
```

Compile the Same Example Using Keil MDK-ARM (continued)

- And the compile sequence can be created in a DOS batch file



```
SET PATH=C:\Keil\ARM\BIN40\;%PATH%
SET RVCT40INC=C:\Keil\ARM\RV31\INC
SET RVCT40LIB=C:\Keil\ARM\RV31\LIB
SET CPU_TYPE=Cortex-M3
SET CPU_VENDOR=ARM
SET UV2_TARGET=Target 1
SET CPU_CLOCK=0x00000000
C:\Keil\ARM\BIN40\armcc -c -O3 -W -g -Otime --device DLM vectors.c
C:\Keil\ARM\BIN40\armcc -c -O3 -W -g -Otime --device DLM blinky.c
C:\Keil\ARM\BIN40\armlink --device DLM "--keep=Startup.o(RESET)"
  "--first=Startup.o(RESET)" -scatter led.scat --map vectors.o
  blinky.o -o blinky.elf
C:\Keil\ARM\BIN40\fromelf --bin blinky.elf -o blinky.bin
```

- In general, it is much easier to use the µVision IDE to create and compile projects rather than using command lines.

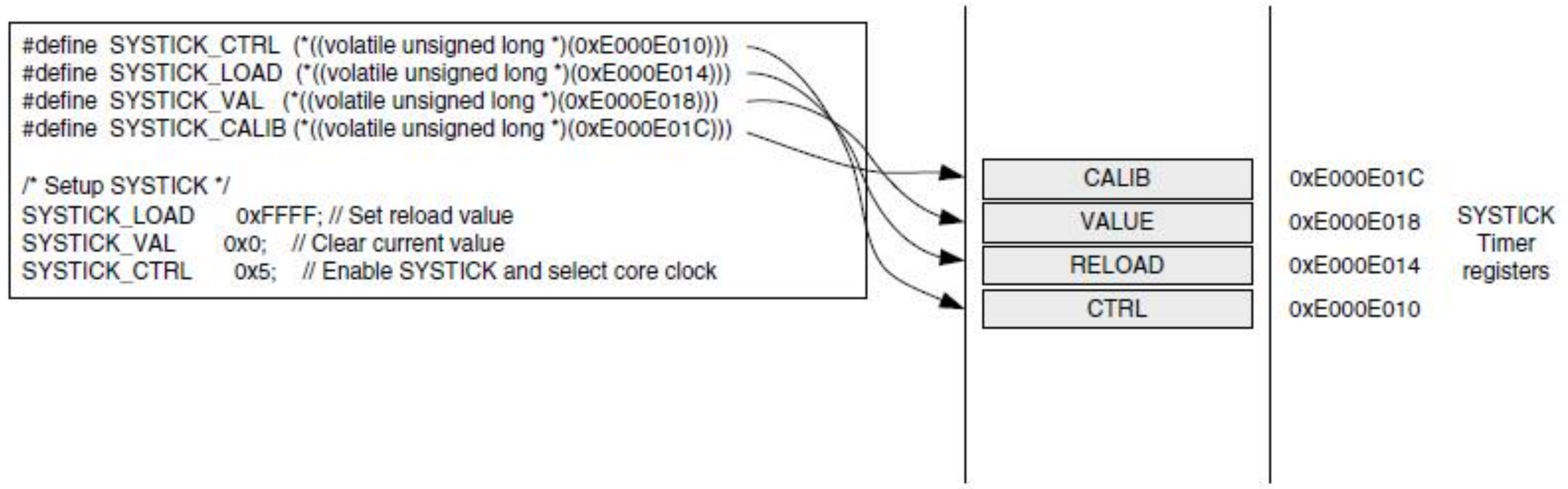
Accessing Memory-Mapped Registers in C

- There are various methods to access memory-mapped peripheral registers in C language.
  - Method 1: Accessing Peripheral Registers as Pointers.
  - Method 2: Accessing Peripheral Registers as Pointers to Elements in a Data Structure.
  - Method 3: Defining Peripheral-Based Address Using Scatter Loading File.
- For illustration, we will use the System Tick (SYSTICK) Timer in the Cortex-M3 as an example peripheral to demonstrate different access methods in C language.
  - The SYSTICK is a 24-bit timer which contains only four registers.

Accessing Memory-Mapped Registers in C (continued)

Method 1:

- Each register is defined as pointer separately.
  - This is illustrated in Figure 10.2.



```
#define  SYSTICK_CTRL  (*((volatile unsigned long *)(0xE000E010)))
#define  SYSTICK_LOAD  (*((volatile unsigned long *)(0xE000E014)))
#define  SYSTICK_VAL   (*((volatile unsigned long *)(0xE000E018)))
#define  SYSTICK_CALIB (*((volatile unsigned long *)(0xE000E01C)))

/* Setup SYSTICK */
SYSTICK_LOAD     0xFFFF; // Set reload value
SYSTICK_VAL      0x0;    // Clear current value
SYSTICK_CTRL     0x5;    // Enable SYSTICK and select core clock
```

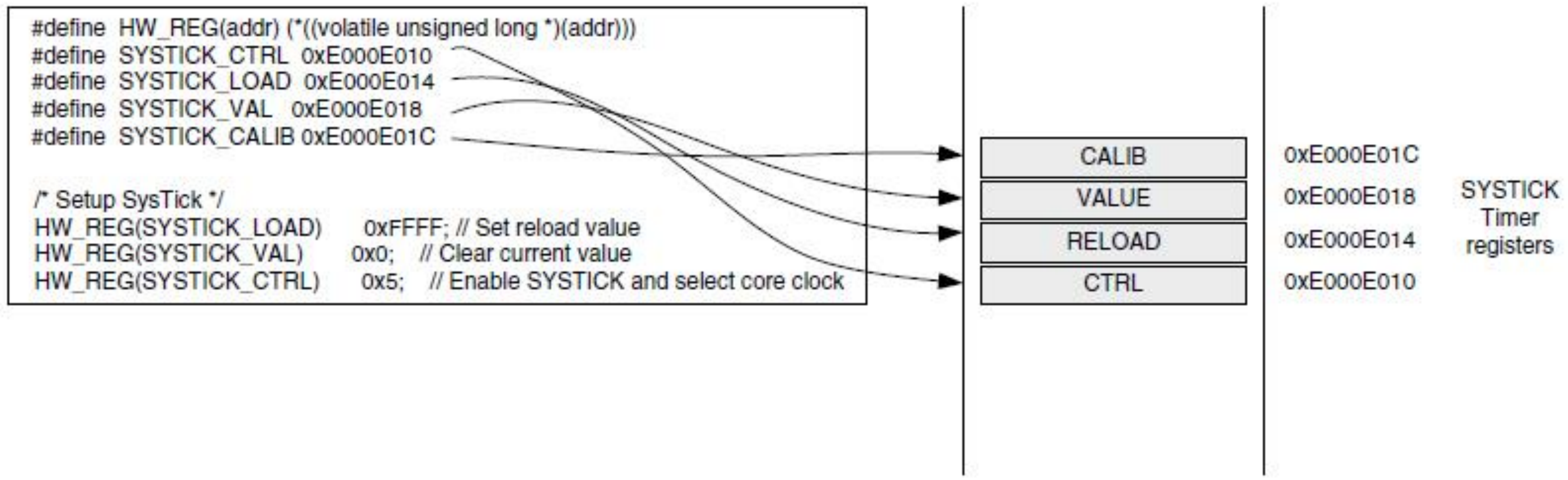
FIGURE 10.2

Accessing Peripheral Registers as Pointers.

Accessing Memory-Mapped Registers in C (continued)

- Based on the same method, we can define a macro to convert address values to C pointer.
- The C-code looks a bit different, but the generated code is the same as previous implementation.
  - This is illustrated in Figure 10.3.

Accessing Memory-Mapped Registers in C (continued)



```
#define  HW_REG(addr) (*((volatile unsigned long *)(addr)))
#define  SYSTICK_CTRL  0xE000E010
#define  SYSTICK_LOAD  0xE000E014
#define  SYSTICK_VAL   0xE000E018
#define  SYSTICK_CALIB 0xE000E01C

/* Setup SysTick */
HW_REG(SYSTICK_LOAD)     0xFFFF; // Set reload value
HW_REG(SYSTICK_VAL)     0x0;   // Clear current value
HW_REG(SYSTICK_CTRL)     0x5;   // Enable SYSTICK and select core clock
```

FIGURE 10.3

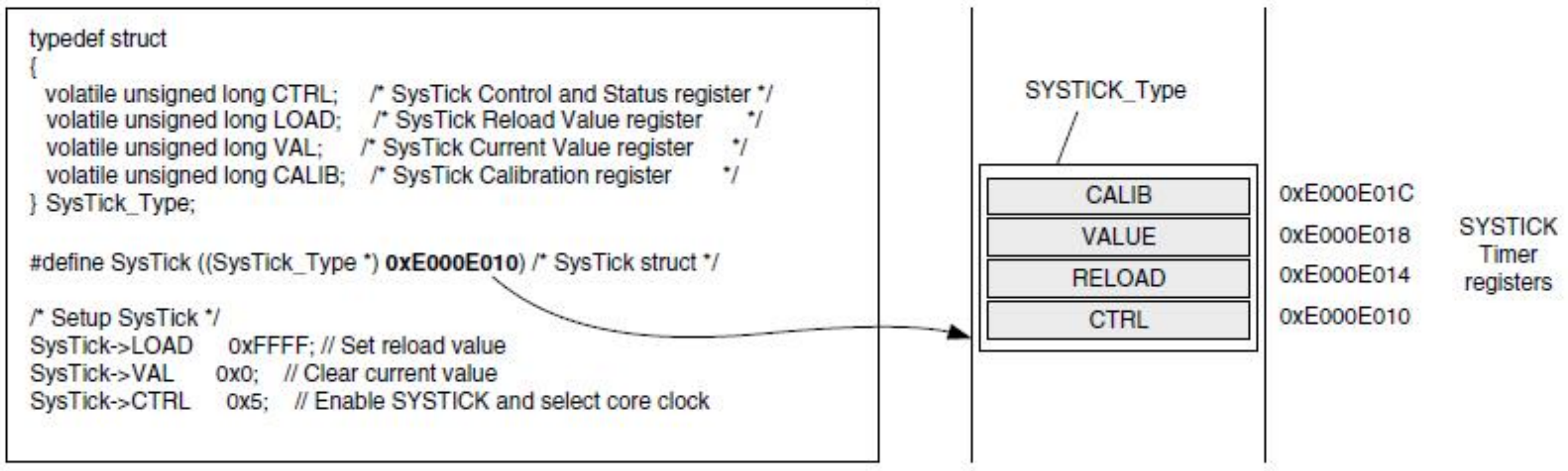
Alternative Way of Accessing Peripheral Registers as Pointers.

Accessing Memory-Mapped Registers in C (continued)

Method 2:

- The registers can be defined as a data structure, and then define a pointer of the defined structure.
- This is the method used in CMSIS compliant device driver libraries.
- This is illustrated in Figure 10.4.

Accessing Memory-Mapped Registers in C (continued)



```
typedef struct
{
  volatile unsigned long CTRL;    /* SysTick Control and Status register */
  volatile unsigned long LOAD;    /* SysTick Reload Value register       */
  volatile unsigned long VAL;     /* SysTick Current Value register      */
  volatile unsigned long CALIB;   /* SysTick Calibration register        */
} SysTick_Type;

#define SysTick ((SysTick_Type *) 0xE000E010) /* SysTick struct */

/* Setup SysTick */
SysTick->LOAD    0xFFFF; // Set reload value
SysTick->VAL     0x0;    // Clear current value
SysTick->CTRL    0x5;    // Enable SYSTICK and select core clock
```

FIGURE 10.4

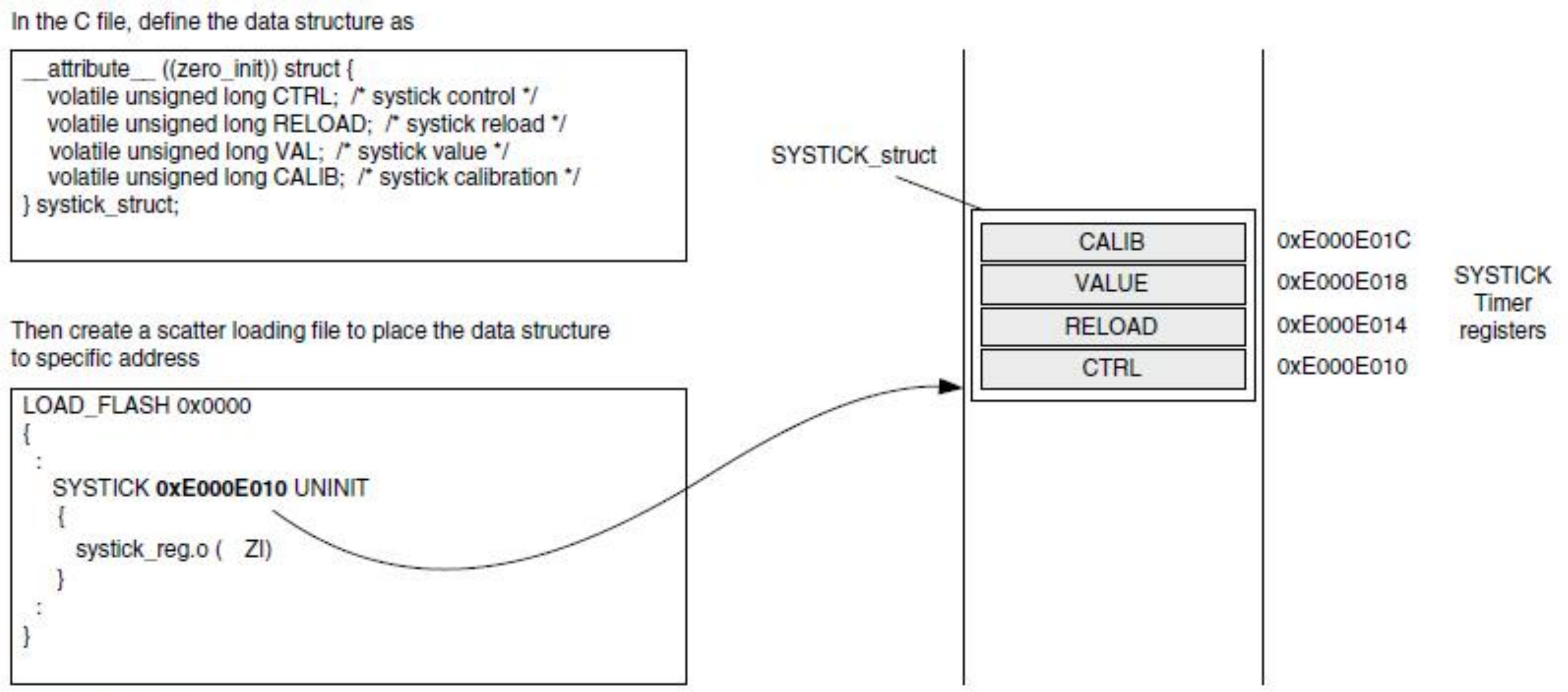
Accessing Peripheral Registers as Pointers to Elements in a Data Structure.

Accessing Memory-Mapped Registers in C (continued)

Method 3:

- This method also uses data structure, but the base address of the peripheral is defined using a scatter loading file (or linker script) during linking stage.
  - This is illustrated in Figure 10.5.

Accessing Memory-Mapped Registers in C (continued)


In the C file, define the data structure as

```
__attribute__ ((zero_init)) struct {
   volatile unsigned long CTRL;  /* systick control */
   volatile unsigned long RELOAD;  /* systick reload */
   volatile unsigned long VAL;  /* systick value */
   volatile unsigned long CALIB;  /* systick calibration */
} systick_struct;
```

Then create a scatter loading file to place the data structure to specific address

```
LOAD_FLASH 0x0000
{
  :
    SYSTICK 0xE000E010 UNINIT
    {
      systick_reg.o (  ZI)
    }
  :
}
```

FIGURE 10.5

Defining Peripheral-Based Address Using Scatter Loading File.

Accessing Memory-Mapped Registers in C (continued)

- Method 1 is the simplest, however, it can result in less efficient code compared with the others as the address value for the registers are stored separately as constant.
- As a result, the code size can be larger and might be slower as it requires more accesses to the program memory to set up the address values.
- However, for peripheral control code that only access to one register, the efficiency of method 1 is identical to others.

Accessing Memory-Mapped Registers in C (continued)

- Method 2 is possibly the most commonly used.
- It allows the registers in a peripheral to share just one constant for base address value.
- The immediate offset address mode can be used for access of each register.
- This is the method used in CMSIS.

Accessing Memory-Mapped Registers in C (continued)

- Method 3 has the same efficiency as method 2, but it is less portable due to the use of a scatter loading file (scatter loading file syntax is tool chain specific).
- Method 3 is required when you are developing a device driver library for a peripheral that is used in multiple devices, and the base address of the peripheral is not known until in the linking stage.

Intrinsic Functions

- Use of the C language can often speed up application development, but in some cases, we need to use some instructions that cannot be generated using normal C-code.
- Some C compilers provide intrinsic functions for accessing these special instructions.
- Intrinsic functions are used just like normal C functions.
- For example, ARM compilers (including RealView C Compilers and Keil MDK-ARM) provide the intrinsic functions listed in Table 10.1 for commonly used instructions.

Intrinsic Functions (continued)

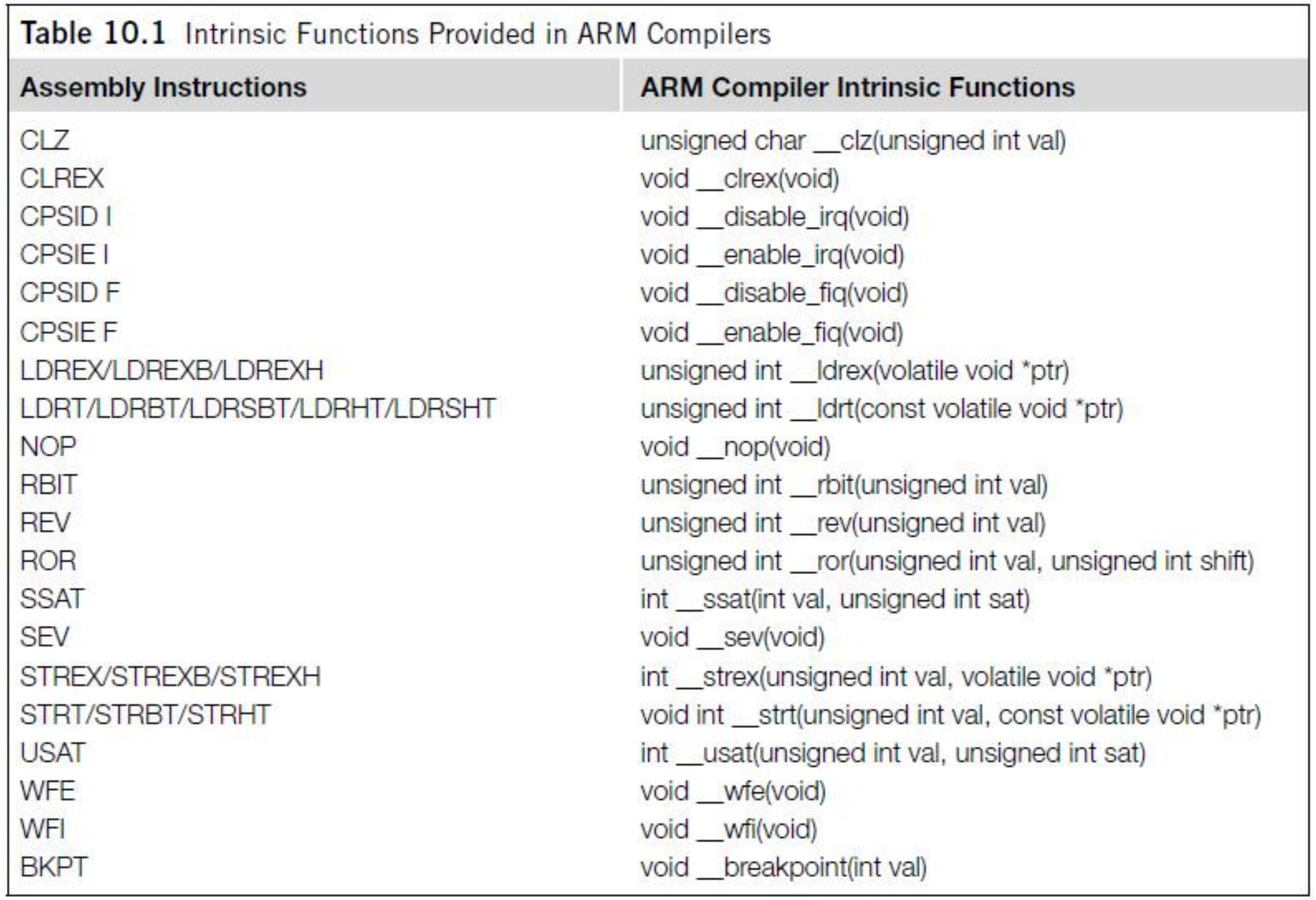


Table 10.1 Intrinsic Functions Provided in ARM Compilers

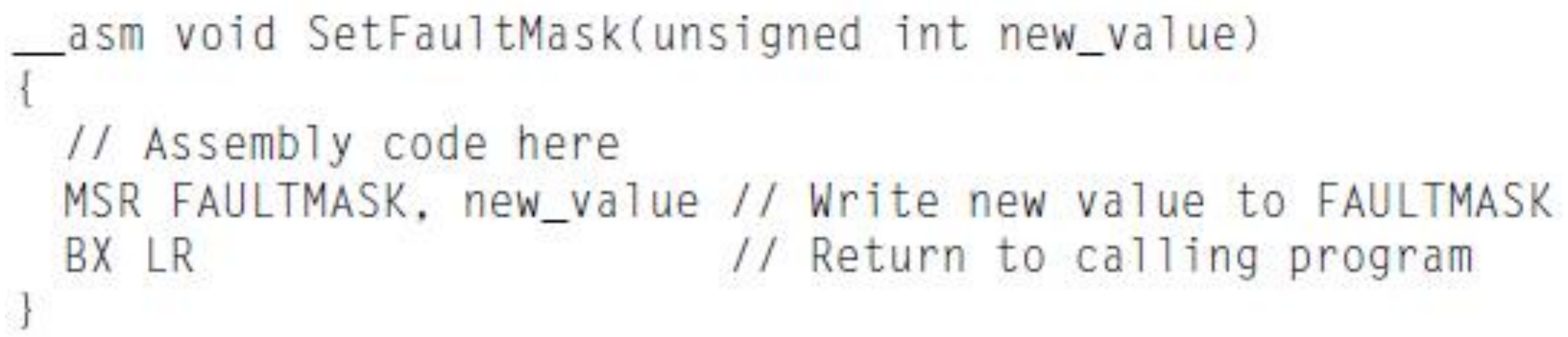
Assembly Instructions	ARM Compiler Intrinsic Functions
CLZ	unsigned char __clz(unsigned int val)
CLREX	void __clrex(void)
CPSID I	void __disable_irq(void)
CPSIE I	void __enable_irq(void)
CPSID F	void __disable_fiq(void)
CPSIE F	void __enable_fiq(void)
LDREX/LDREXB/LDREXH	unsigned int __ldrex(volatile void *ptr)
LDRT/LDRBT/LDRSBT/LDRHT/LDRSHT	unsigned int __ldrt(const volatile void *ptr)
NOP	void __nop(void)
RBIT	unsigned int __rbit(unsigned int val)
REV	unsigned int __rev(unsigned int val)
ROR	unsigned int __ror(unsigned int val, unsigned int shift)
SSAT	int __ssat(int val, unsigned int sat)
SEV	void __sev(void)
STREX/STREXB/STREXH	int __strex(unsigned int val, volatile void *ptr)
STRT/STRBT/STRHT	void int __strt(unsigned int val, const volatile void *ptr)
USAT	int __usat(unsigned int val, unsigned int sat)
WFE	void __wfe(void)
WFI	void __wfi(void)
BKPT	void __breakpoint(int val)

Embedded Assembler and Inline Assembler

- As an alternative to using intrinsic functions, we can also directly access assembly instructions in C-code.
- This is often necessary in low-level system control or when we need to implement a timing critical routine and decide to implement it in assembly for the best performance.
- Most ARM C compilers allow to include assembly code in form of *inline assembler*.

Embedded Assembler and Inline Assembler (continued)

- In the ARM compiler, assembly code can be added inside the C program.
- For example, assembly functions can be inserted in C programs this way:



```
__asm void SetFaultMask(unsigned int new_value)
{
  // Assembly code here
  MSR FAULTMASK, new_value // Write new value to FAULTMASK
  BX LR                    // Return to calling program
}
```

Cortex Microcontroller Software Interface Standard (CMSIS)

CMSIS

- The Cortex-M3 microcontrollers are gaining momentum in the embedded application market, as more and more products based on the Cortex-M3 processor and software that support the Cortex-M3 processor are emerging.
- There are also a number of companies providing embedded software solutions, including codecs, data processing libraries, and various software and debug solutions.
- The CMSIS was developed by ARM to allow users of the Cortex-M3 microcontrollers to gain the most benefit from all these software solutions and to allow them to develop their embedded application quickly and reliably.

CMSIS (continued)

- The Cortex Microcontroller Software Interface Standard (CMSIS) was started in 2008 to improve software usability and inter-operability of ARM microcontroller software.
- It is integrated into the driver libraries provided by silicon vendors, providing a standardized software interface for the Cortex-M3 processor features, as well as a number of common system and I/O functions.
- The library is also supported by software companies including embedded OS vendors and compiler vendors.

CMSIS (continued)

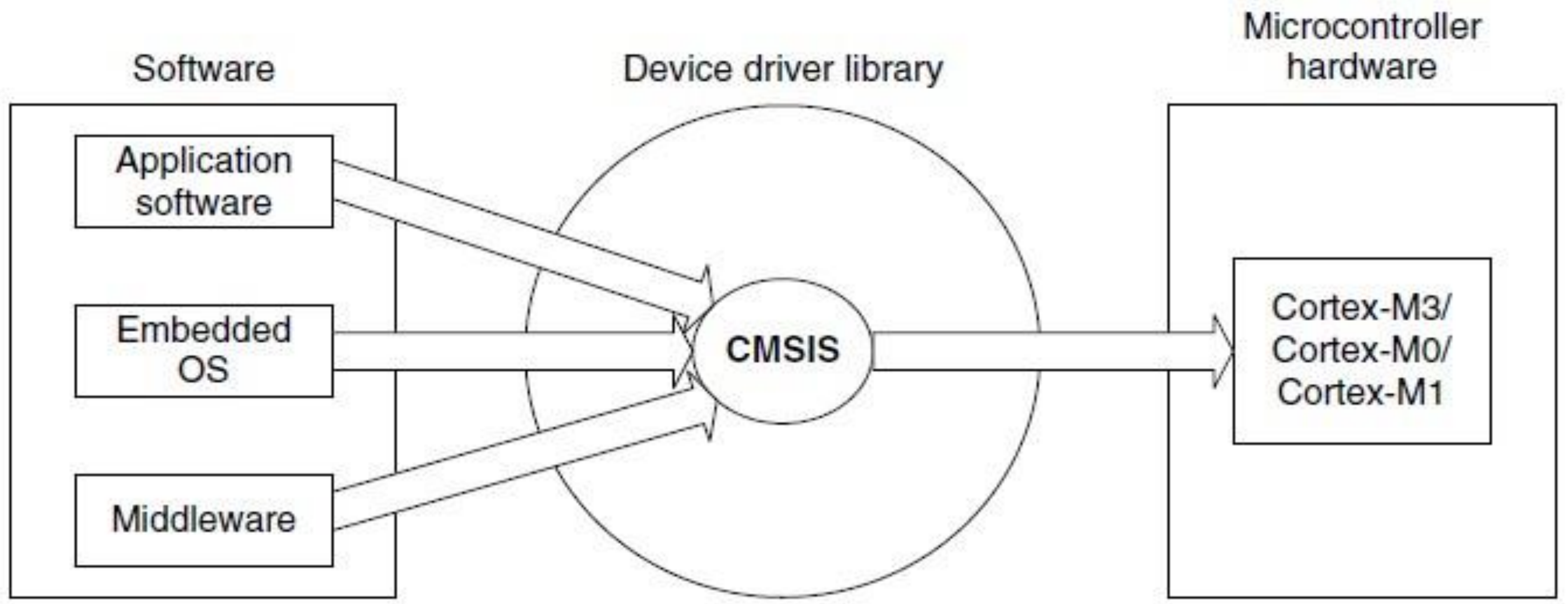


FIGURE 10.6

CMSIS Provides a Standardized Access Interface for Embedded Software Products.

CMSIS (continued)

- The aims of CMSIS are to:
  - improve software portability and reusability
  - enable software solution suppliers to develop products that can work seamlessly with device libraries from various silicon vendors
  - allow embedded developers to develop software quicker with an easy-to-use and standardized software interface
  - allow embedded software to be used on multiple compiler products
  - avoid device driver compatibility issues when using software solutions from multiple sources

CMSIS – Areas of Standardization

- The scope of CMSIS involves standardization in the following areas:
  - *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.
    - Again, this helps new users to start to use Cortex-M microcontrollers and aids software portability.

CMSIS – Areas of Standardization (continued)

- *Standardized intrinsic functions:* Intrinsic functions are normally used to produce instructions that cannot be generated by IEC/ISO C.
  - 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).
  - By having these common access functions in the device driver library, reusability and portability of embedded software are improved.
- *Standardized way for embedded software to determine system clock frequency:* A software variable called *SystemFrequency* 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

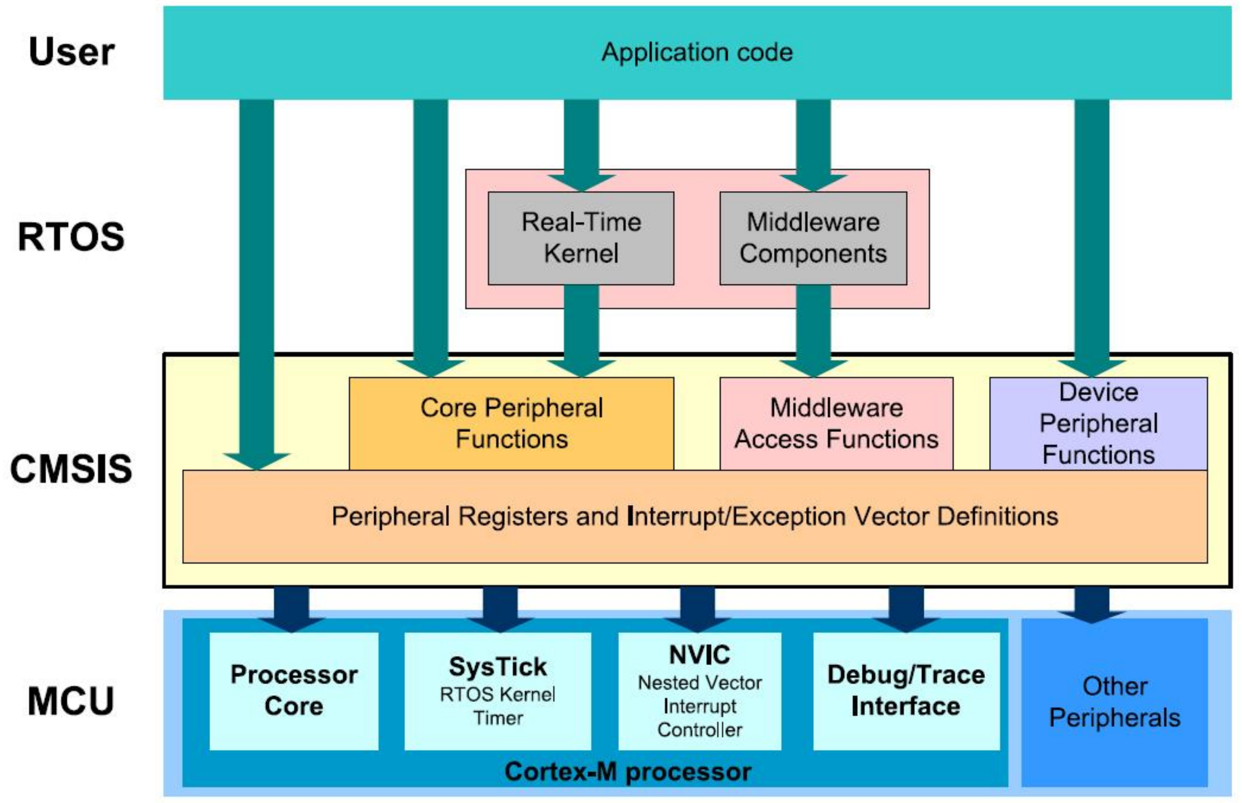

User
Application code
RTOS
Real-Time Kernel
Middleware Components
CMSIS
Core Peripheral Functions
Middleware Access Functions
Device Peripheral Functions
Peripheral Registers and Interrupt/Exception Vector Definitions
MCU
Processor Core
SysTick RTOS Kernel Timer
NVIC Nested Vector Interrupt Controller
Debug/Trace Interface
Other Peripherals
Cortex-M processor

FIGURE 10.7

CMSIS Structure.

Organization of CMSIS (continued)

- The CMSIS is divided into multiple layers as follows:
  - Core Peripheral Access Layer
    - Name definitions, address definitions, and helper functions to access core registers and core peripherals
  - 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

Organization of CMSIS (continued)

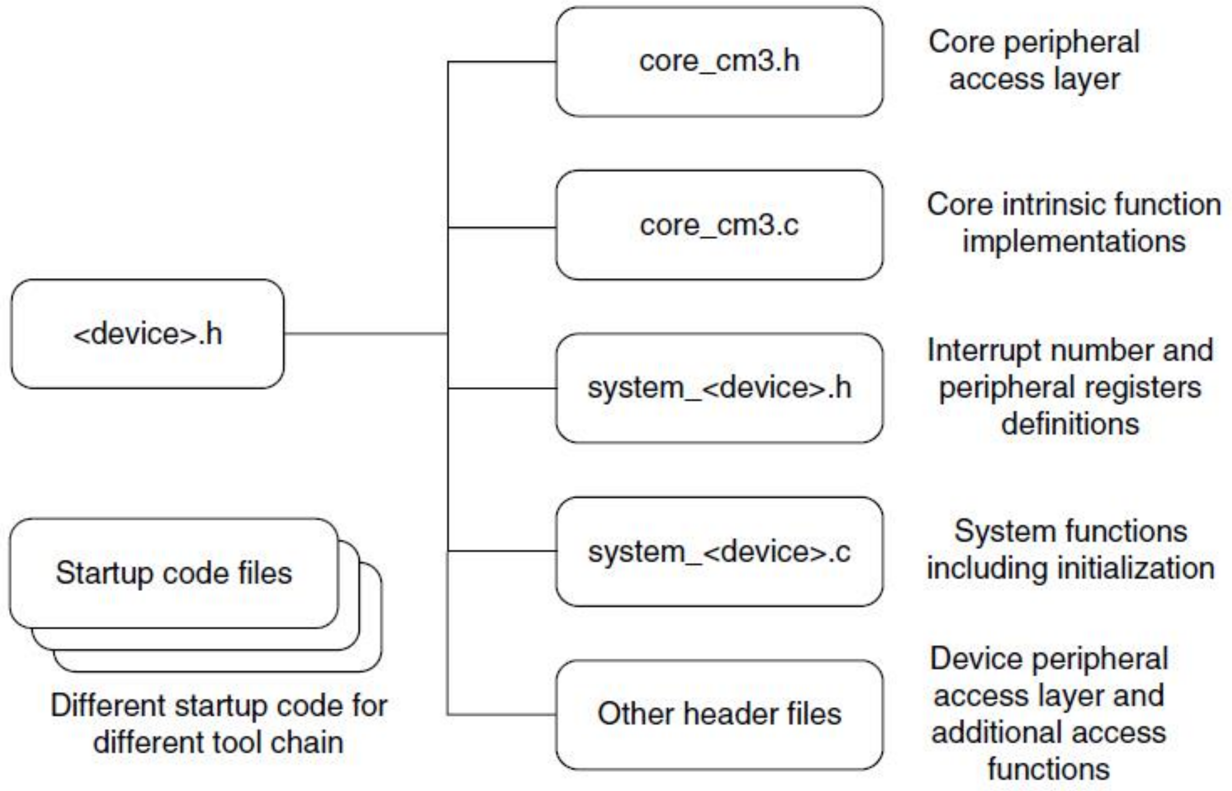
- Device Peripheral Access Layer (MCU specific)
  - Name definitions, address definitions, and driver code to access peripherals
- Access Functions for Peripherals (MCU specific)
  - Optional additional helper functions for peripherals

- The role of these layers is summarized in Figure 10.7.

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 (as shown in Figure 10.8).

Using CMSIS (continued)



core_cm3.h
Core peripheral access layer
core_cm3.c
Core intrinsic function implementations
<device>.h
system_<device>.h
Interrupt number and peripheral registers definitions
Startup code files
system_<device>.c
System functions including initialization
Different startup code for different tool chain
Other header files
Device peripheral access layer and additional access functions

FIGURE 10.8

CMSIS Files.

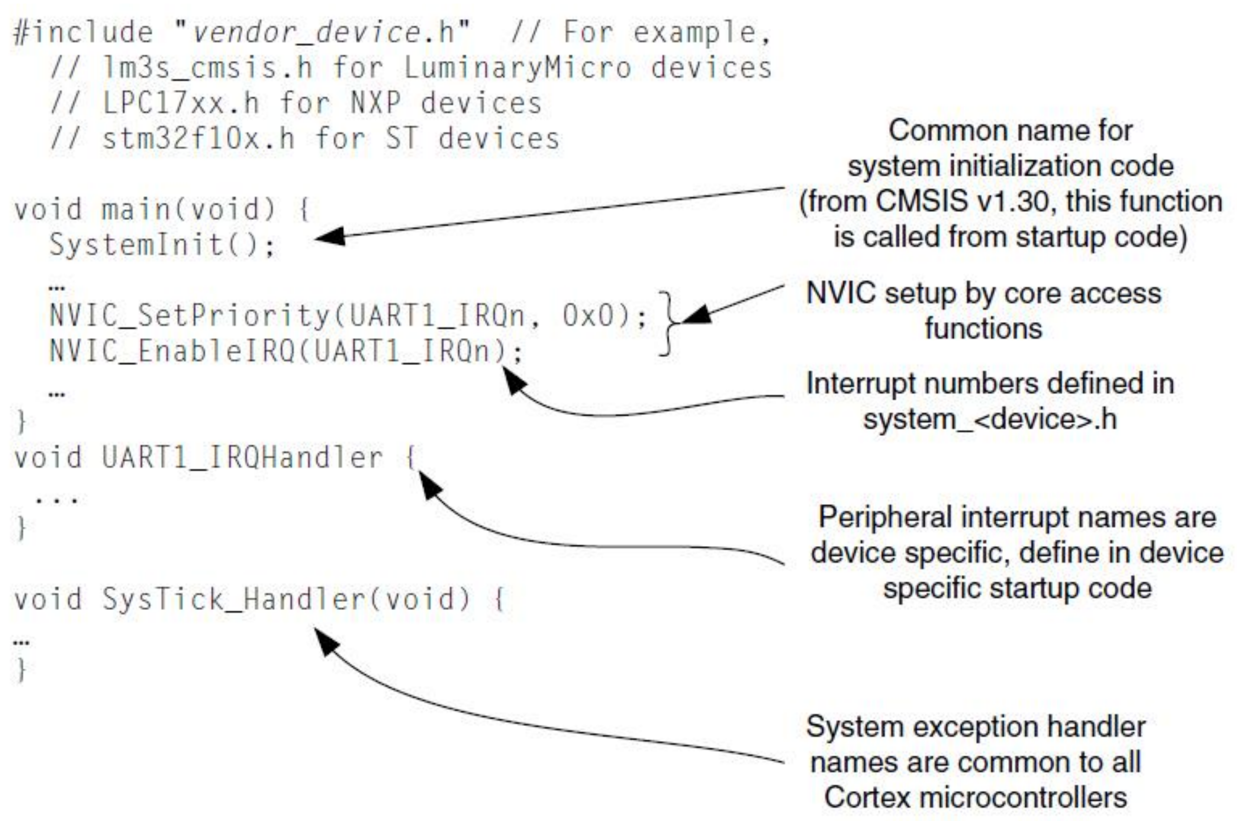
Using CMSIS (continued)

- 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.
  - declaration of CMSIS intrinsic functions to allow C applications to access instructions that cannot be generated using IEC/ISO C language.
  - a function for outputting a debug message via the Instrumentation Trace Module (ITM).
- The file *core_cm3.c* contains implementation of CMSIS intrinsic functions that cannot be implemented in *core_cm3.h* using simple definitions.

Using CMSIS (continued)

- 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) for various supported compilers, and CMSIS version of intrinsic functions to allow embedded software access to all processor core features on different C compiler products.

Using CMSIS (continued)



```
#include "vendor_device.h"  // For example,
  // lm3s_cmsis.h for LuminaryMicro devices
  // LPC17xx.h for NXP devices
  // stm32f10x.h for ST devices

void main(void) {
  SystemInit();
  ...
  NVIC_SetPriority(UART1_IRQn, 0x0);
  NVIC_EnableIRQ(UART1_IRQn);
  ...
}
void UART1_IRQHandler {
 ...
}

void SysTick_Handler(void) {
...
}
```

Common name for system initialization code (from CMSIS v1.30, this function is called from startup code)
NVIC setup by core access functions
Interrupt numbers defined in system_<device>.h
Peripheral interrupt names are device specific, define in device specific startup code
System exception handler names are common to all Cortex microcontrollers

FIGURE 10.9

CMSIS Example.

Using CMSIS (continued)

- A simple example of using CMSIS in your application development is shown in Figure 10.9.
- To use the CMSIS to set up interrupts and exceptions, we need to use the exception/interrupt constants defined in the *system_<device>.h*.
  - These exception and interrupt constants are different from the exception number used in the core internal registers (e.g., Interrupt Program Status Register [IPSR]).
  - For CMSIS, negative numbers are for system exceptions and positive numbers are for peripheral interrupts.

Using CMSIS (continued)

- For development of portable code, you should use the core access functions to access core functionalities and middleware access functions to access peripheral.
  - This allows the porting of software to be minimized between different Cortex microcontrollers.

Benefits of CMSIS

- The main advantage is much better software portability and reusability.
  - Besides easy migration between different Cortex-M3 microcontrollers, it also 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, by using the CMSIS, their software products can become compatible with device drivers from multiple microcontroller vendors, including future microcontroller products that are yet to be released (see Figure 10.10).
  - 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.

Benefits of CMSIS (continued)

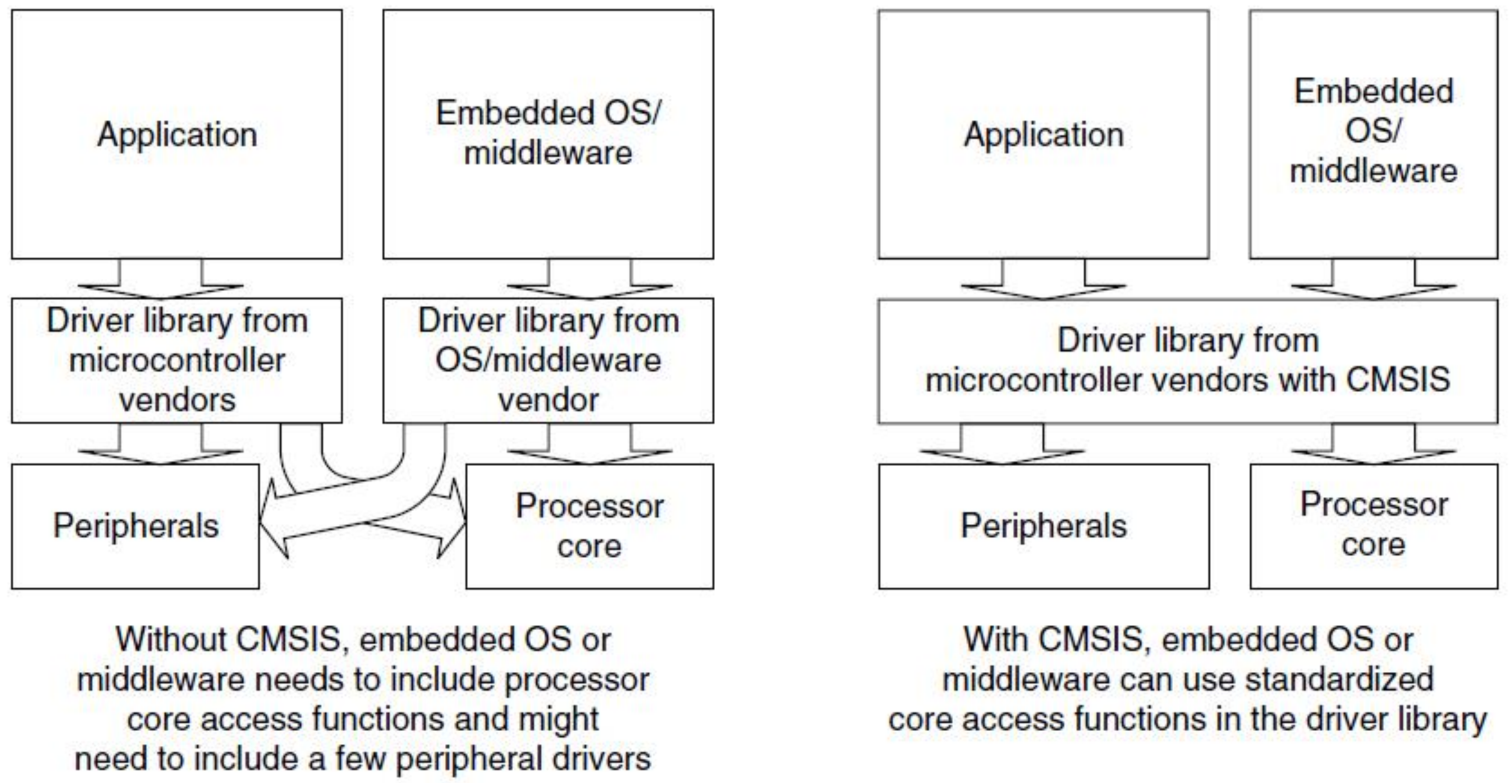

Application
Embedded OS/ middleware
Driver library from microcontroller vendors
Driver library from OS/middleware vendor
Peripherals
Processor core
Without CMSIS, embedded OS or middleware needs to include processor core access functions and might need to include a few peripheral drivers
Application
Embedded OS/ middleware
Driver library from microcontroller vendors with CMSIS
Peripherals
Processor core
With CMSIS, embedded OS or middleware can use standardized core access functions in the driver library

FIGURE 10.10

CMSIS Avoids Overlapping Driver Code.

Benefits of CMSIS (continued)

- The CMSIS has a small memory footprint (less than 1 KB for all core access functions and a few bytes of RAM).
  - It also avoids overlapping of core peripheral driver code when reusing software code from other projects.
- Since CMSIS is supported by multiple compiler vendors, embedded software can compile and run with different compilers.
  - As a result, embedded OS and middleware can be MCU vendor independent and compiler tool vendor independent.
  - Before availability of CMSIS, intrinsic functions were generally compiler specific and could cause problems in retargetting the software in a different compiler.

Benefits of CMSIS (continued)

- Since all CMSIS compliant device driver libraries have a similar structure, learning to use different Cortex-M3 microcontrollers is even easier as the software interface has similar look and feel.
  - No need to relearn a new application programming interface.
- CMSIS is tested by multiple parties and is Motor Industry Software Reliability Association (MISRA) compliant, thus reducing the validation effort required for developing your own NVIC or core feature access functions.

Using Assembly

Using Assembly

- For small projects, it is possible to develop the whole application in assembly language.
  - However, this is often much harder for beginners.
- Using assembler, one might be able to get the best optimization, though it might increase the development time, and it could be easy to make mistakes.
- In addition, handling complex data structures or function library management can be extremely difficult in assembler.

Using Assembly (continued)

- Yet 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

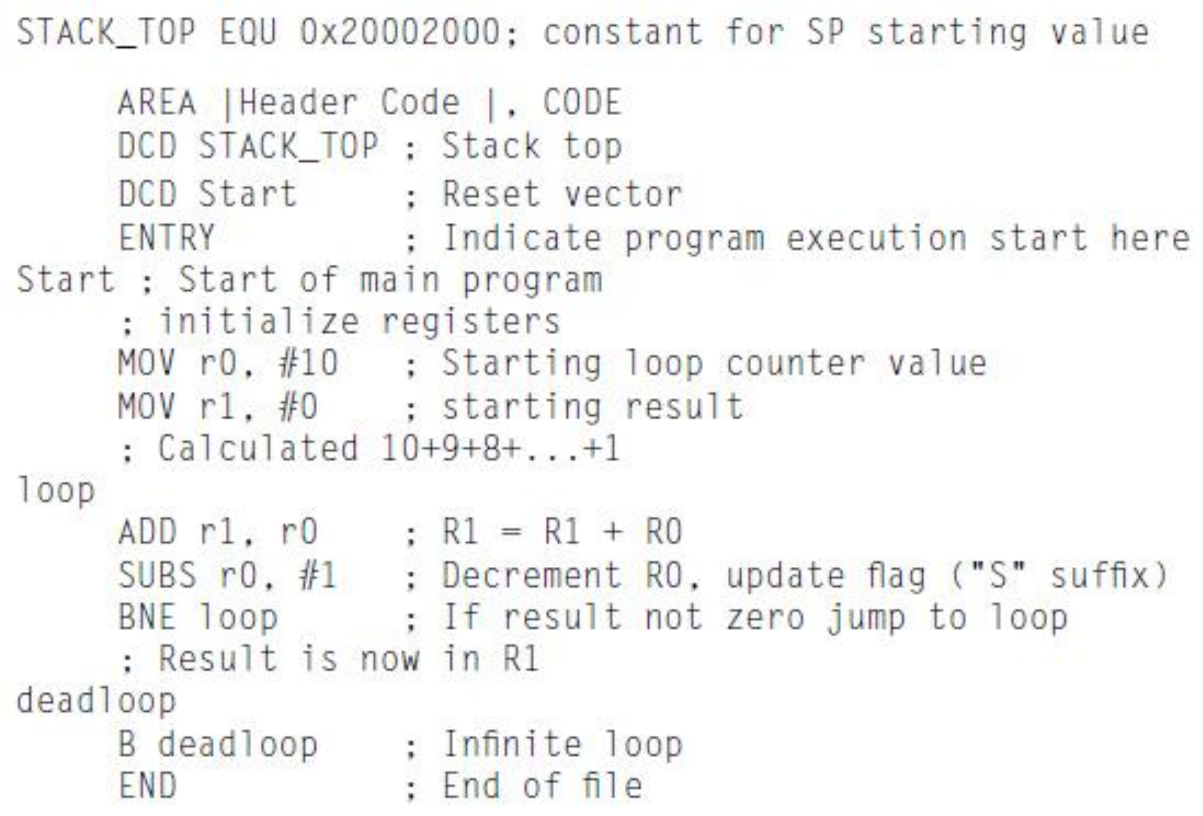
- In various situations, assembly code and the C program interact.
- For example,
  - When embedded assembly (or inline assembler, in the case of the GNU tool chain) is used in C program code
  - When C program code calls a function or subroutine implemented in assembler in a separate file
  - When an assembly program calls a C function or subroutine
- In these cases, it is important to understand how parameters and return results are passed between the calling program and the function being called.
  - The mechanisms of these interactions are specified in the *ARM Architecture Procedure Call Standard [AAPCS]*.

The Interface between Assembly and C (continued)

- For simple cases, when a calling program needs to pass parameters to a subroutine or function, it will use registers R0–R3, where R0 is the first parameter, R1 is the second, and so on.
- Similarly, R0 is used for returning a value at the end of a function.
- R0–R3 and R12 can be changed by a function or subroutine whereas the contents of R4–R11 should be restored to the previous state before entering the function, usually handled by stack PUSH and stack POP.
- If a C function is called by an assembly code, the effect of a possible register change to R0–R3 and R12 will need to be taken into account.
- If the contents of these registers are needed at a later stage, these registers might need to be saved on the stack and restored after the C function completes.

Example for Assembly Programming

- Consider a simple program to add first ten integers.



```
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
     ; 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
```

Example for Assembly Programming (continued)

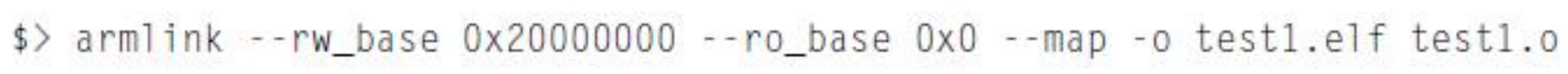
- This simple program contains the initial stack pointer (SP) value, the initial program counter (PC) value, and setup registers and then does the required calculation in a loop.
- Assuming ARM RealView compilation tools are used, this program can be assembled using

```
$> armasm --cpu cortex-m3 -o test1.o test1.s
```

- The *-o* option specifies the output file name.
- The test1.o is an object file.

Example for Assembly Programming (continued)

- We then need to use a linker to create an executable image (ELF).
- This can be done by



```
$> armlink --rw_base 0x20000000 --ro_base 0x0 --map -o test1.elf test1.o
```

- Here, *--ro-base 0x0* specifies that the read-only region (program ROM) starts at address 0x0; *--rwbase* specifies that the read/write region (data memory) starts at address 0x20000000.
- The *--map* option creates an image map, which is useful for understanding the memory layout of the compiled image.

Example for Assembly Programming (continued)

- Finally, we need to create the binary image

```
$> fromelf --bin --output test1.bin test1.elf
```

- For checking that the image looks like what we wanted, we can also generate a disassembled code list file by

```
$> fromelf -c --output test1.list test1.elf
```

- If everything works fine, ELF image or binary image can be loaded into the hardware or instruction set simulator for testing.

References

1. Joseph Yiu, ***"The Definitive Guide to the ARM Cortex-M3"***, 2nd Edition, Newnes (Elsevier), 2010.
2. https://www.arm.com