Fundamentals of Embedded and Real Time Systems

MODULE 06

TAMER AWAD

Review Module 05

Module 06

Assignment Reviews
- Assignment 02
- Assignment 03

Embedded Software Architecture
- Round-Robin
- Round-Robin with Interrupt
- Function-queue-scheduling architecture
- Real-Time Operating System architecture

C Programming – Functions (continued)
- Modules
- Recursive Calls
- AAPCS – ARM Architecture Procedure Call Standard

Pitfalls in Embedded
- Stack Overflow
- Array index out of bound
- Stack corruption

Arguments & Pointers
- Passing arguments in C
- Passing pointer arguments (the Swap example)
- Returning pointer values

Assembly Programming
- Use of Assembly code
- Cortex-M4 instruction set overview
- Assembly function general structure
- Demo: Create Assembly Code

Assignment
- Assignment 05

Assignment reviews

Embedded Software Architecture

- Round-Robin
- Round-Robin with interrupt
- Function-Queue-Scheduling Architecture
- Real-Time Operating System Architecture

Embedded Software Architecture

- Embedded software architecture is the basic structure that can be used to put the system together.
- Most important factor is how much control you need to have over **system response**.
- If response-time requirements are loose, then system can be written with a simpler architecture.
- A system with stringent response-time requirements, different deadlines and different priorities will require a more complex architecture.

“Round-Robin”

What’s special about this architecture? In what situation this method would not work?

- **Priority** – none, everything runs in sequence.
- **Response time** – the sum of all tasks.
- **Impact of changes** – significant: changing the execution time of tasks or adding tasks will impact all other tasks.
- **Simplicity**, no shared data problems.

```
void main(void) {
  while (TRUE) {
    if (IO_DEVICE_A_NEED_SERVICE) {
      // take care of device A
    }

    if (IO_DEVICE_B_NEED_SERVICE) {
      // take care of device B
    }

    if (IO_DEVICE_C_NEED_SERVICE) {
      // take care of device C
    }
  }
}
```

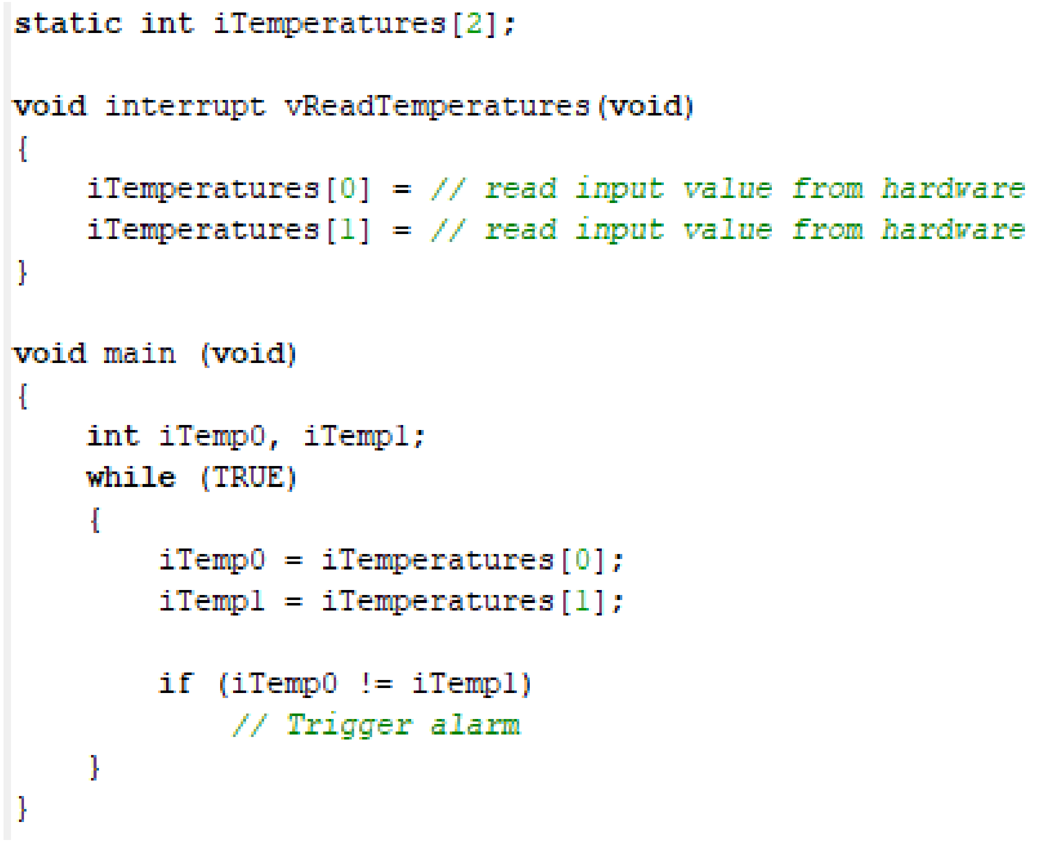
“Round-Robin with Interrupts”

- Interrupts deal with the urgent needs of the hardware and then set flags.
- The main loop polls the flags and does any follow-up processing required by interrupt.
- **Priority** – Interrupts get priority over main loop
- **Response time**:
    - The sum of all tasks + the execution times of any interrupt routines that happen to occur.
- **Impact of changes** – Less significant for interrupt service routines. Same as Round Robin for main loop.
- **Shared data** – must deal with data shared with interrupt service routines

```
SET_VECTOR(P3AD,button_isr);
SET_VECTOR(TIMER1, display_isr);

while(1) {
    if (FLAG_A is set)
        read_temp();
}
```

Shared data problem - Example

- Interrupts introduced shared data problem:
  - They are difficult to find
  - They do not happen every time the code runs
- <u>Example:</u>
  - System monitors two temperatures
  - Expecting both temperatures to be always equal
  - If not equal, trigger alarm

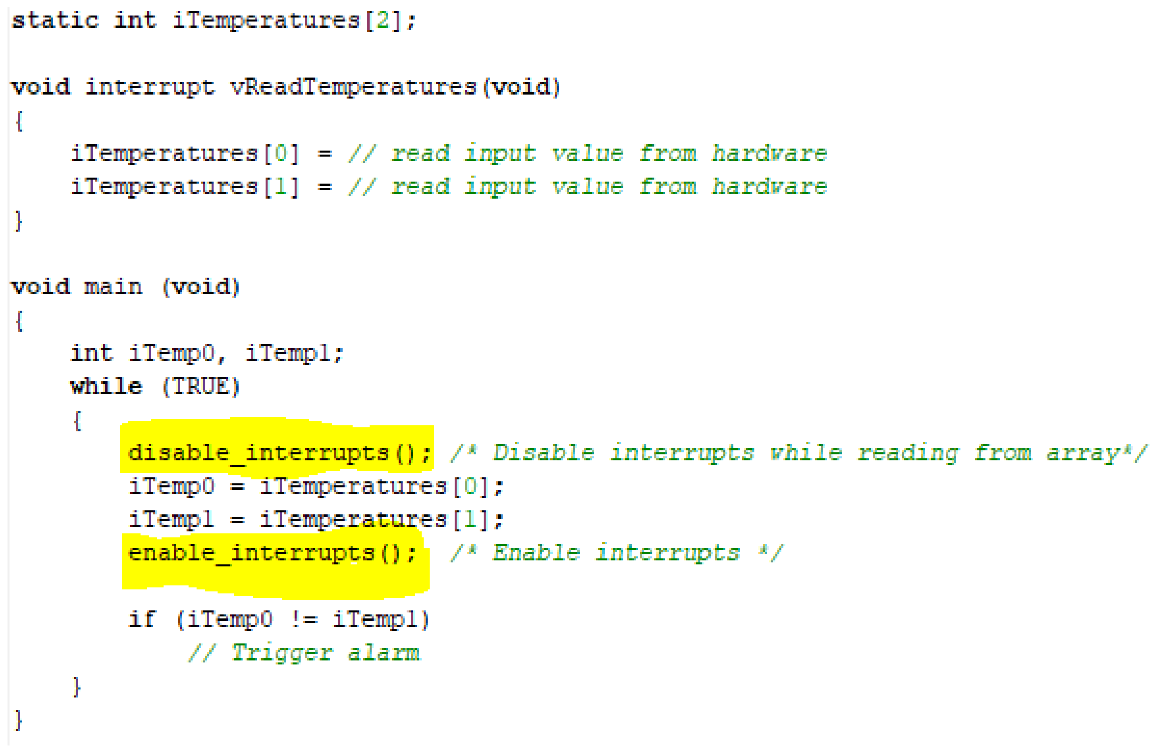


```
static int iTemperatures[2];

void interrupt vReadTemperatures(void)
{
    iTemperatures[0] = // read input value from hardware
    iTemperatures[1] = // read input value from hardware
}

void main (void)
{
    int iTemp0, iTemp1;
    while (TRUE)
    {
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];

        if (iTemp0 != iTemp1)
            // Trigger alarm
    }
}
```

Shared data problem

Solution:

- Disable interrupts whenever the task code uses the shared data
- A lib or RTOS will provide a function that can disable and enable interrupts.
- In assembly language, you can invoke the processor’s instructions that enable and disable interrupts.
  - *CPSID & CPSIE*
  - *(Cortex-M4 Devices Generic User Guide , 3-2)*



```
static int iTemperatures[2];

void interrupt vReadTemperatures(void)
{
    iTemperatures[0] = // read input value from hardware
    iTemperatures[1] = // read input value from hardware
}

void main (void)
{
    int iTemp0, iTemp1;
    while (TRUE)
    {
        disable_interrupts(); /* Disable interrupts while reading from array*/
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];
        enable_interrupts();  /* Enable interrupts */

        if (iTemp0 != iTemp1)
            // Trigger alarm
    }
}
```

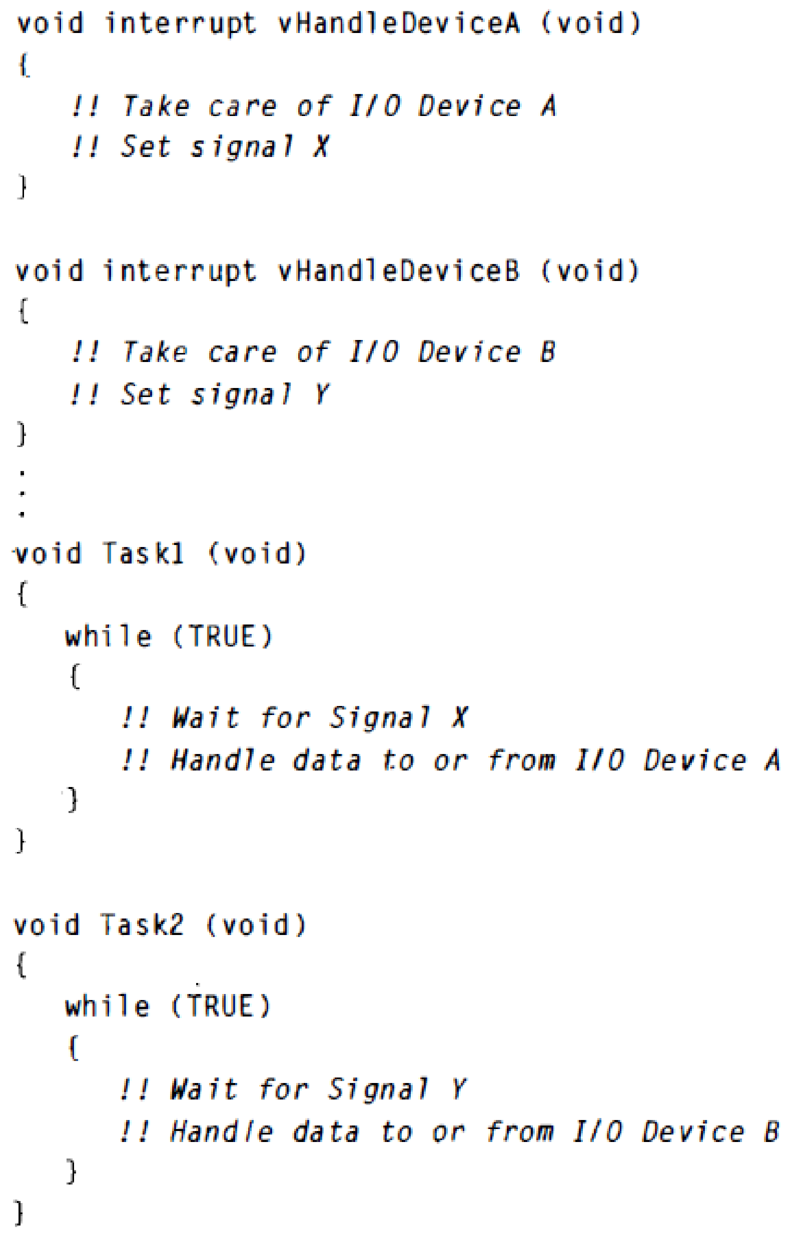
“Function-Queue-Scheduling Architecture”

- How does it work:
  - Interrupt routines (in priority order) add function pointers to a queue of function pointers for the main function to call.
  - The main routine just reads pointers form the queue and calls the functions.
- Response Time:
  - Worst wait for the highest priority task is the length of the longest task function + the execution times of any interrupt routines that happen to occur.
- Impact of Changes:
  - The response for the lower priority task may get worse.
- Simplicity:
  - Must deal with shared data and must write function queue code.

```
void interrupt vHandleDeviceA (void)
{
    !! Take care of I/O Device A
    !! Put function_A on queue of function pointers
}

void interrupt vHandleDeviceB (void)
{
    !! Take care of I/O Device B
    !! Put function_B on queue of function pointers
}

void main (void)
{
    while (TRUE)
    {
        while (!!Queue of function pointers is empty)
            ;

        !! Call first function on queue
    }
}

void function_A (void)
{
    !! Handle actions required by device A
}

void function_B (void)
{
    !! Handle actions required by device B
}
```

“RTOS”

- Interrupt routines take care of the most urgent operations (same as with the other architectures)
- They “signal” work for the code function (Task) to do.
- The “signaling” is handled by the RTOS (no need to use shared variables)
- No loop in our code decides what needs to be done next.
- Code in the RTOS decides which of Tasks should run.
- The RTOS knows about all the tasks and will run the most “urgent” first (i.e. higher priority).



```
void interrupt vHandleDeviceA (void)
{
    !! Take care of I/O Device A
    !! Set signal X
}

void interrupt vHandleDeviceB (void)
{
    !! Take care of I/O Device B
    !! Set signal Y
}
.
.
.
void Task1 (void)
{
    while (TRUE)
    {
        !! Wait for Signal X
        !! Handle data to or from I/O Device A
    }
}

void Task2 (void)
{
    while (TRUE)
    {
        !! Wait for Signal Y
        !! Handle data to or from I/O Device B
    }
}
```

More on Functions

- Modules
- Recursion
- AAPCS

Modules - Modular Programming in C

- Having the program in one file is impractical and inconvenient.
- How do you organize medium-sized or larger C programs?
- Modular programming is one way of managing the complexity
- Groups related sets of functions together into a **module**
- The **module** is divided into an interface (.h) and an implementation (.c)
- Clients modules import the interface so that they can access the functions in the module.
- Modules provide abstraction, and information-hiding, making the large-scale structure of a program easier to understand.
- Speeds up compilation. Only the changed files need to be recompiled.
- *Source:* https://www.embedded.com/modular-programming-in-c/
- **Demo**: Delay Function as a Module

Demo: Modules + Recursion + AAPCS

- Modules (Separate delay into .c & .h)
- #ifndef
- Separate compilation (more efficient)
- Linker Errors
- Demo recursive function
  - Example: Factorial function.
    - 0! = 1
    - 3! = 3 * 2 * 1
    - n! = n * (n-1) * … * 1
- Function usage examples:
  - Capturing return value
  - Using function inside expressions
  - Ignoring function return value

Questions

What is a recursive function?

What is the down-side of recursive calls?

What other alternatives could be used?

AAPCS

- AAPCS = Arm Architecture Procedure Call Standard
- Defines the convention between calling functions and subroutines, among other things.
  - Parameters and return result passing - For simple cases, input parameters can be passed to a function using R0 (first parameter), R1 (second parameter), R2 (third parameter), and R3 (fourth parameter).
  - Usually the return value of a function is stored in R0.
  - If more than four parameters need to be passed to a function, the stack will be used.
- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042f/index.html

Register Usages & Requirements in Function calls – According to **AAPCS**

Register	**Function Call Behavior**
R0-R3, R12	Caller Saved Registers: - Contents in these registers can be changed by a function. - Assembly code calling a function might need to save the values in these registers if they are required for operations in later stages
R4-R11	Callee Saved Registers: - Contents in these registers must be retained by a function. - If a function needs to use these registers for processing, they need to be saved to the stack and restored before function return.
R14 (LR)	The value in the Ling Register needs to be saved to the stack if the function is not a “leaf” function, i.e. contains a “BL” or “BLX” instruction (calling another function) because the value in LR will be overwritten when “BL” or “BLX” is executed.
R13 (SP), R15 (PC)	Should not be used for normal processing

Pitfalls in Embedded

- Stack
- Stack Overflow
- Array index out of bound
- Stack corruption

Stack memory

- Stack is a kind of memory usage mechanism that allows a portion of memory to be used as Last-In-First-Out data storage buffer.
- ARM processors use the main system memory for stack memory operations, and have the PUSH instruction to store data in stack and the POP instruction to retrieve data from stack.
- Stack can be used for:
  - Temporary storage of original data when a function being executed needs to use registers (in the register bank) for data processing. The values can be restored at the end of the function so the program that called the function will not lose its data.
  - Passing of information to functions or subroutines.
  - For storing local variables.
  - To hold processor status and register values in the case of exceptions such as an interrupt.

Source: *The Definitive Guide to ARM Cortex M3 & M4*

Stack memory

- The Cortex-M processors use a stack memory model called “full-descending stack.”
  - When the processor is started, the SP is set to the end of the memory space reserved for stack memory.
- For each PUSH operation, the processor first decrements the SP, then stores the value in the memory location pointed by SP.
- During operations, the SP points to the memory location where the last data was pushed to the stack.
- In a POP operation, the value of the memory location pointed by SP is read, and then the value of SP is incremented automatically.

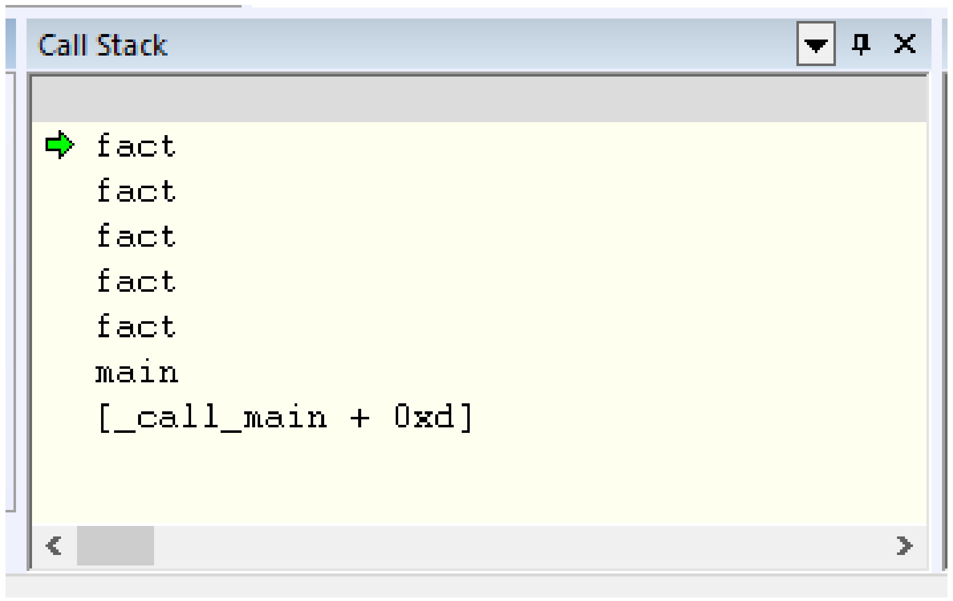
Source: *The Definitive Guide to ARM Cortex M3 & M4*

Demo: Stackoverflow & Array OOB

- Take factorial to the breaking point
  - Add local array inside of factorial
- The Callstack view
- Stack-overflow
- Array index out of bound and stack corruption

Callstack view

Shows all the functions currently nested on the stack



```
fact
fact
fact
fact
fact
main
[_call_main + 0xd]
```

Stackoveflow and BusFault

Occurs if the call stack pointer exceeds the stack bound.

The program hits the BusFault_Handler

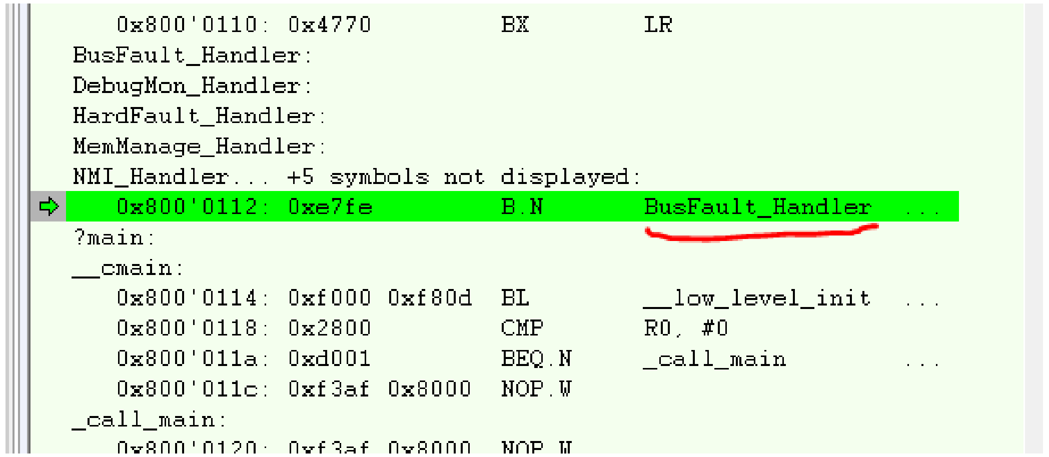
The Exception handler provided in the standard IAR startup code.

This is linked with the main program by the linker.

The BusFault exception is a hardware mechanism implemented in the CPU to handle the situation when the CPU is forced to access nonexistent memory.

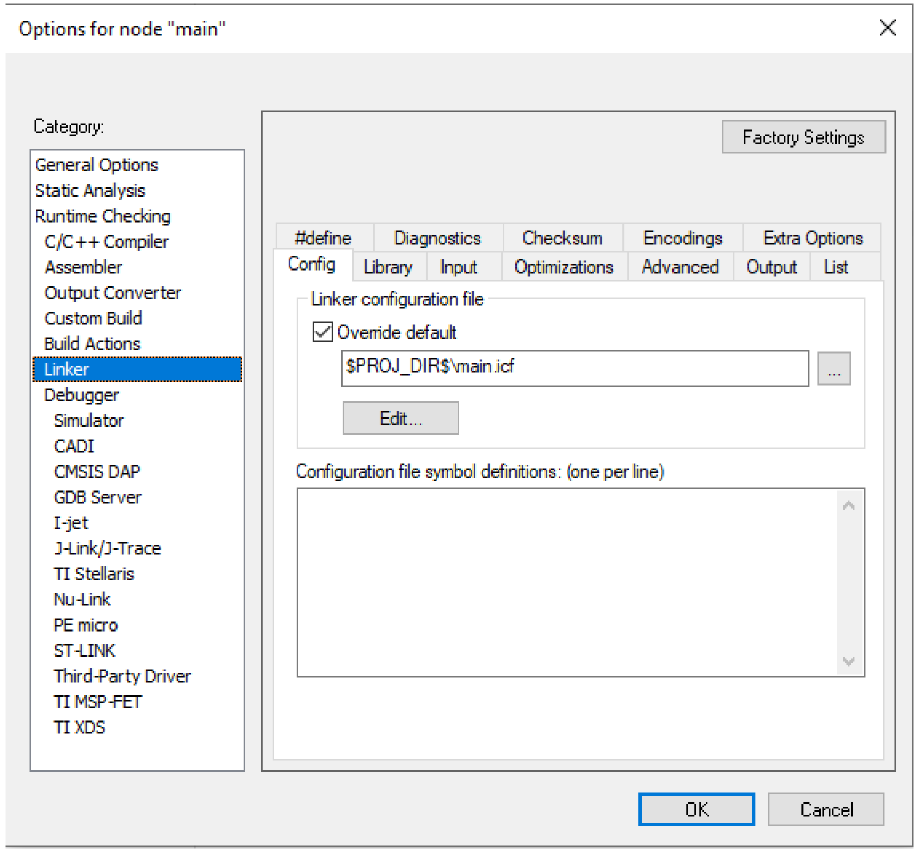
The IAR startup code implements the BusFault exception as well as all other exceptions as an endless loop.

You can provide your own code that could do something else; for example reset the CPU (will see that in the future)



```
   0x800'0110: 0x4770         BX         LR
BusFault_Handler:
DebugMon_Handler:
HardFault_Handler:
MemManage_Handler:
NMI_Handler... +5 symbols not displayed:
   0x800'0112: 0xe7fe         B.N        BusFault_Handler   ...
?main:
__cmain:
   0x800'0114: 0xf000 0xf80d  BL         __low_level_init   ...
   0x800'0118: 0x2800         CMP        R0, #0
   0x800'011a: 0xd001         BEQ.N      _call_main         ...
   0x800'011c: 0xf3af 0x8000  NOP.W
_call_main:
   0x800'0120: 0xf3af 0x8000  NOP.W
```

How to change the stack size?


Options for node "main"

Category:

General Options
Static Analysis
Runtime Checking
C/C++ Compiler
Assembler
Output Converter
Custom Build
Build Actions
Linker
Debugger
Simulator
CADI
CMSIS DAP
GDB Server
I-jet
J-Link/J-Trace
TI Stellaris
Nu-Link
PE micro
ST-LINK
Third-Party Driver
TI MSP-FET
TI XDS

Factory Settings

#define | Diagnostics | Checksum | Encodings | Extra Options
Config | Library | Input | Optimizations | Advanced | Output | List

Linker configuration file

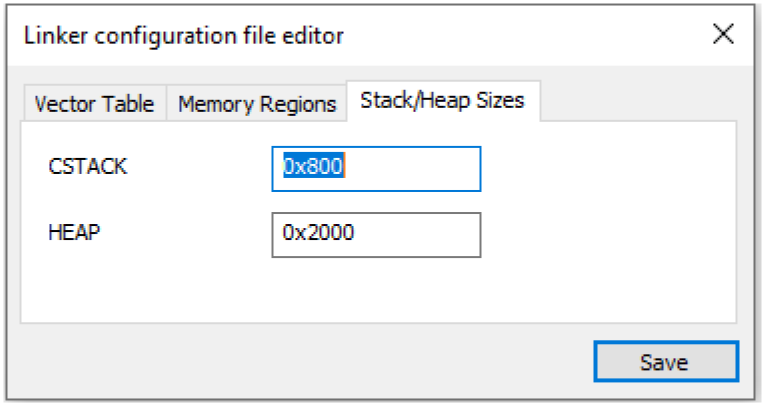
[✓] Override default

$PROJ_DIR$\main.icf

Edit...

Configuration file symbol definitions: (one per line)

OK | Cancel


Linker configuration file editor

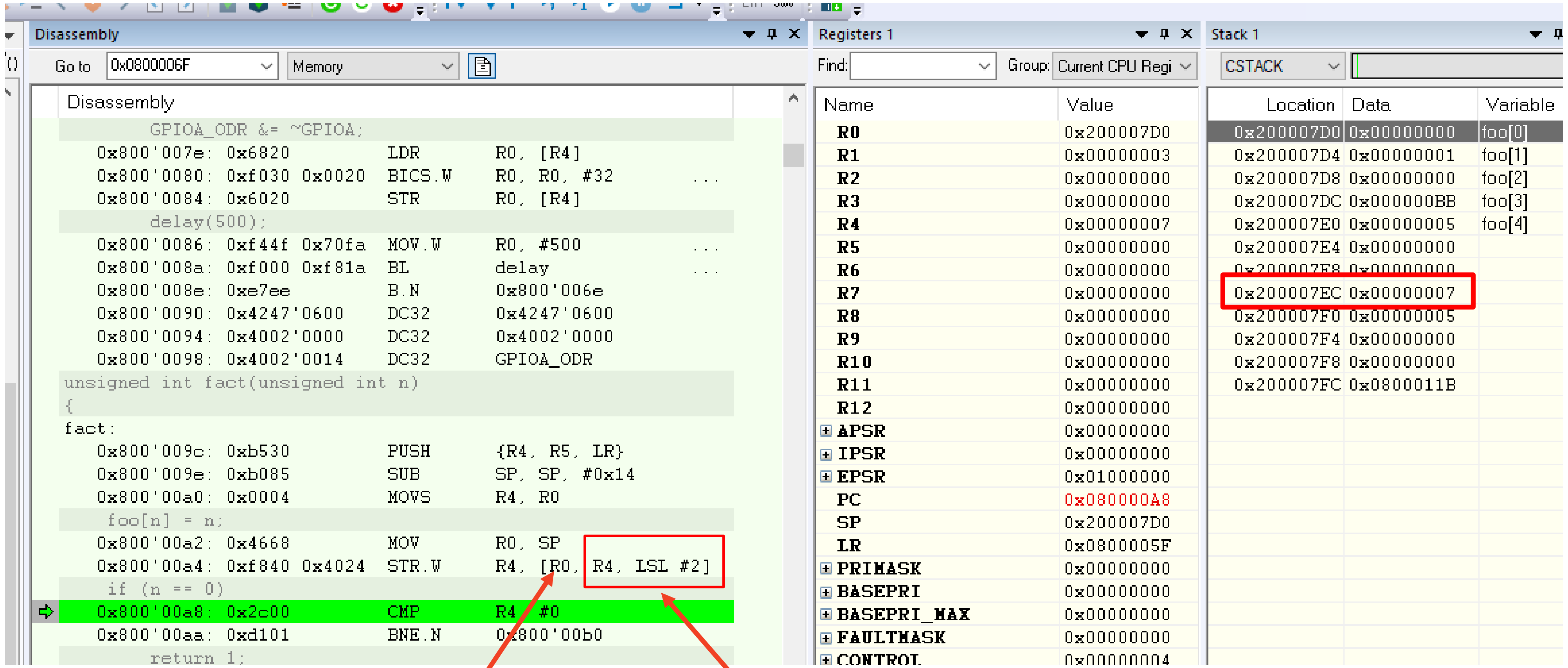
Vector Table | Memory Regions | Stack/Heap Sizes

CSTACK 0x800

HEAP 0x2000

Save

Corrupting the stack – Array index out of bound



```
Disassembly
        GPIOA_ODR &= ~GPIOA;
    0x800'007e: 0x6820          LDR      R0, [R4]
    0x800'0080: 0xf030 0x0020   BICS.W   R0, R0, #32          ...
    0x800'0084: 0x6020          STR      R0, [R4]
        delay(500);
    0x800'0086: 0xf44f 0x70fa   MOV.W    R0, #500             ...
    0x800'008a: 0xf000 0xf81a   BL       delay                ...
    0x800'008e: 0xe7ee          B.N      0x800'006e
    0x800'0090: 0x4247'0600     DC32     0x4247'0600
    0x800'0094: 0x4002'0000     DC32     0x4002'0000
    0x800'0098: 0x4002'0014     DC32     GPIOA_ODR
unsigned int fact(unsigned int n)
{
fact:
    0x800'009c: 0xb530          PUSH     {R4, R5, LR}
    0x800'009e: 0xb085          SUB      SP, SP, #0x14
    0x800'00a0: 0x0004          MOVS     R4, R0
    foo[n] = n;
    0x800'00a2: 0x4668          MOV      R0, SP
    0x800'00a4: 0xf840 0x4024   STR.W    R4, [R0, R4, LSL #2]
    if (n == 0)
⇨   0x800'00a8: 0x2c00          CMP      R4, #0
    0x800'00aa: 0xd101          BNE.N    0x800'00b0
        return 1;
```

Name	Value
R0	0x200007D0
R1	0x00000003
R2	0x00000000
R3	0x00000000
R4	0x00000007
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
⊞ APSR	0x00000000
⊞ IPSR	0x00000000
⊞ EPSR	0x01000000
PC	0x080000A8
SP	0x200007D0
LR	0x0800005F
⊞ PRIMASK	0x00000000
⊞ BASEPRI	0x00000000
⊞ BASEPRI_MAX	0x00000000
⊞ FAULTMASK	0x00000000
⊞ CONTROL	0x00000004

Location	Data	Variable
0x200007D0	0x00000000	foo[0]
0x200007D4	0x00000001	foo[1]
0x200007D8	0x00000000	foo[2]
0x200007DC	0x000000BB	foo[3]
0x200007E0	0x00000005	foo[4]
0x200007E4	0x00000000	
0x200007E8	0x00000000	
0x200007EC	0x00000007	
0x200007F0	0x00000005	
0x200007F4	0x00000000	
0x200007F8	0x00000000	
0x200007FC	0x0800011B	

Array base address

Offset = Array index (R4) multiplied by 4 bytes (Left-shift by 2)

Arguments & Pointers

- Passing arguments in C
- Passing pointer arguments (the Swap example)
- Returning pointer values

Passing arguments in C

- C passes function arguments by value
- Only the argument’s value is copied to the internal variable to initialize it.
- Internally the function uses this copy rather than the original argument
- This means, that a function will never change the original argument.
- But what if you want to change the actual arguments?
  - Classic example: Swap function

Demo

- -Arguments passing in C
- -Pointer arguments via swap function example
- -Returning pointers in C

The swap function

```
void swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
void main(void)
{
    int x = 1;
    int y = 2;
    swap(x, y);
}
```

```
void swap(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
void main(void)
{
    int x = 1;
    int y = 2;
    swap(&x, &y);
}
```

Extending on the Swap function

How about swapping the pointers?

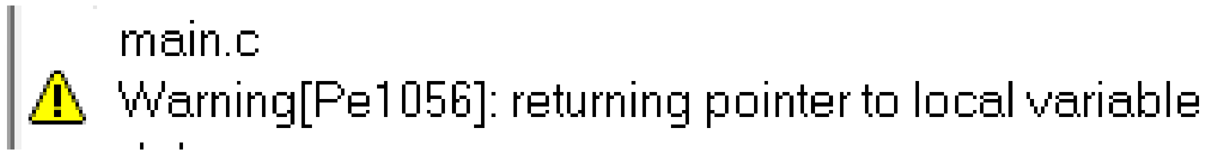
What will happen?

How do you fix it?

→ **Assignment 05**

```
void swap_effective(int* x, int* y)
{
   // swapping the values
   int temp = *x;
   *x = *y;
   *y = temp;

   // swapping the pointers??
   int* tempPtr = x;
   x = y;
   y = tempPtr;
}
```

Another pitfall - Returning pointers in C



```
main.c
Warning[Pe1056]: returning pointer to local variable
```

Never ignore the warning above.

Returning a pointer to a “local” variable is always a bad idea!!

Remember: All local variables go “out-of-scope” once the function returns.

Question: How can we fix it?

Assembly Programming

- Use of Assembly code
- Ways to add Assembly code
- Interaction between C & Assembly
- Cortex-M4 instruction set overview
- Assembly function general structure
- Demo: Create Assembly Code

Use of assembly code in projects

For small projects, it is possible to develop the whole application in assembly language.

However, this is rare in application development because:

It is much harder as you need to have a good understanding of the instructions.

It can be very complicated when the application requires complex data processing.

Device drivers (e.g., header files) from microcontroller vendors are in C. If you

need to access the peripherals in assembly, you need to create your own header files and driver libraries.

Mistakes are not easy to spot and debug.

Assembly program code is not portable.

Use of assembly code in projects

However, there are some situations where we may need to use assembly language for some parts of the project:

- To allow direct manipulation of stack memory (e.g., context-switching code in embedded OSes)
- To optimize for maximum speed/performance or minimum program size for a specific task
- To reuse assembly code from old projects
- To learn about processor architecture

Ways to add assembly code to C project

- Assembly file:
  - The assembly code could be functions implemented in an assembly file.
    - These functions can then be called from C code.
    - Can also call C code from the assembly function.
- Embedded assembler:
  - The assembly code could be functions implemented within a C file using compiler-specific features.
- Inline assembler:
  - The assembly code can be instruction sequences inserted inside C code using inline assembler.

Interaction between C & assembly

- In order to allow assembly code and C code to work together, must adhere to the AAPCS standard.
- A function or a subroutine should retain the values in R4-R11, R13, R14.
  - If these registers will be changed during the function execution, then their values must be saved to the stack and restored before the return to the calling code
- Input parameters can be passed to a function using R0 (first parameter), R1 (second parameter), R2 (third parameter), and R3 (fourth parameter).
- If more than four parameters need to be passed to a function, the stack will be used.
- The return value of a function is stored in R0.
- If an assembly function needs to call a C function, it should ensure that the currently selected stack pointer points to a doubleword aligned address location (e.g., 0x20002000, 0x20002008, 0x20002010, etc.).
- Following the requirements outlined in AAPCS enables better software reusability and avoids potential problems when integrating your assembly code with program code or middleware from third parties.

Calling a C function from assembly

```
int my_add_c(int x1, int x2, int x3, int x4)
{
    return (x1 + x2 + x3 + x4);
}
```

```
MOVS R0, #0x1 ; First parameter (x1)
MOVS R1, #0x2 ; Second parameter (x2)
MOVS R2, #0x3 ; Third parameter (x3)
MOVS R3, #0x4 ; Fourth parameter (x4)
IMPORT my_add_c
BL my_add_c ; Call “my_add_c” function.
            ; Result stored in R0
```

The Caller Saved Registers (R0 to R3, and R12) can be changed by a C function. So if there is any data in these registers that are needed later, we need to save them first.

Calling an assembly function from C

```
EXPORT My_Add

My_Add FUNCTION

ADDS R0, R0, R1

ADDS R0, R0, R2

ADDS R0, R0, R3

BX LR ; Return result in R0

ENDFUNC
```

If we need to modify callee saved registers (R4 to R11), we need to push these registers on to the stack first and then restore them before exiting the function.

We also need to save LR if the function is going to call another function (i.e. not a leaf function) because the value of LR will be changed when executing BL or BLX.

```
extern int My_Add(int x1, int x2, int x3, int x4);

...

int y;

...

y = My_Add(1, 2, 3, 4); // call the My_Add function
```

Embedded assembler

In ARM toolchains, a feature called embedded assembler allows you to implement assembly functions/subroutines inside a C file.

Need to add the __asm keyword in front of the function declaration.

```
__asm int My_Add(int x1, int x2, int x3, int x4)
{
ADDS R0, R0, R1
ADDS R0, R0, R2
ADDS R0, R0, R3
BX LR ; Return result in R0
}
```

Inline assembler

The inline assembler feature is also available in ARM C compilers. For example:

```
int qadd8(int i, int j)
{
  int res;
  __asm
  {
    QADD8 res, i, j
  }
  return res;
}
```

Assembly: Typical Instruction Format

```
Opcode DestReg, Operand2

Opcode DestReg, SrcReg, Operand2
```

- Instructions may have two or three operands
- First operand is (almost) always a destination register
- Operand2
  - Specify another register
  - A small fixed constant
- Instructions are encoded as 16-bit or 32-bit values
  - Generally prefer 16 bit encodings, because uses less program spaces

Cortex-M4 Instruction Set

The instruction set is divided into various groups:

- Moving data within the processor
- Memory accesses
- Arithmetic operations
- Logic operations
- Program flow control
- Other functions…

Moving data within the processor

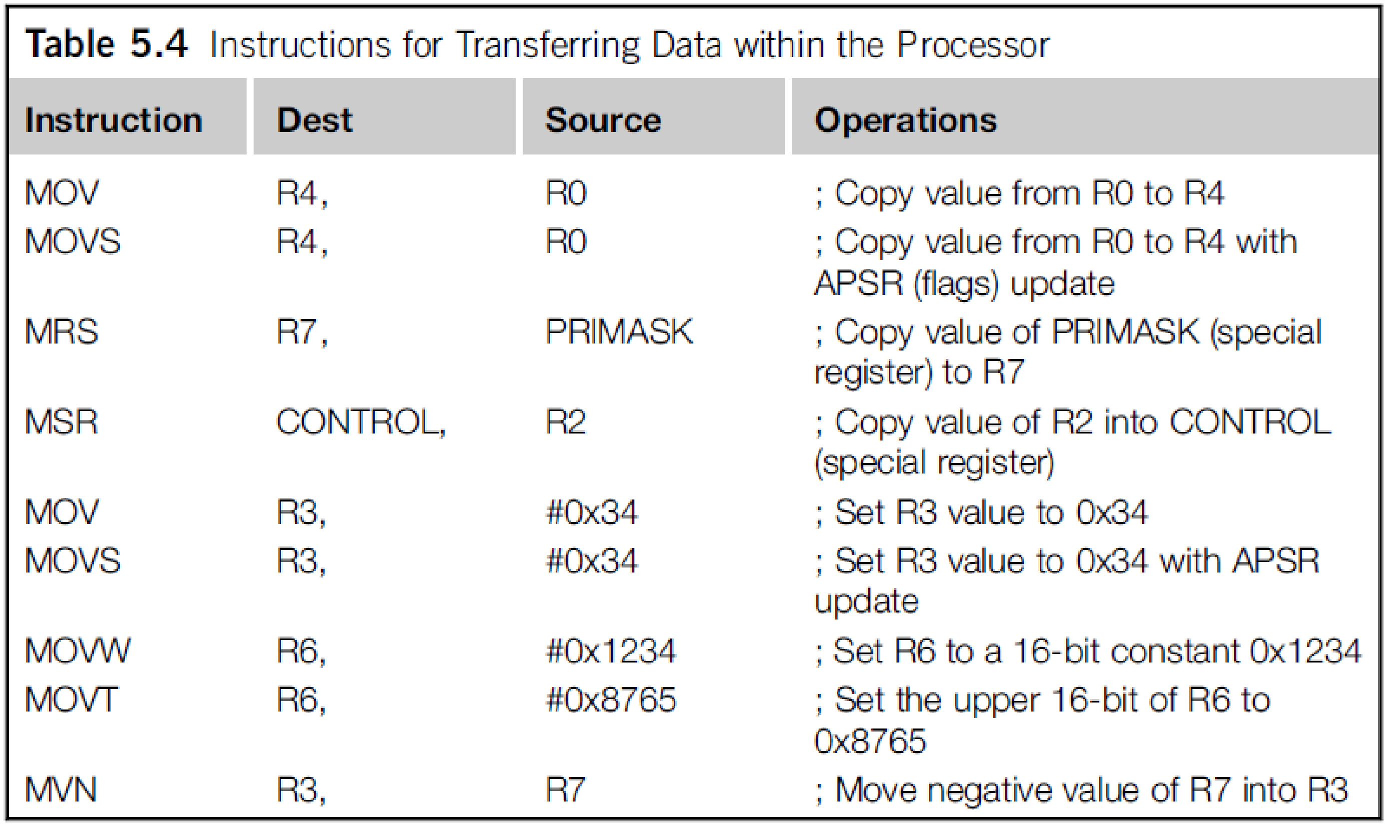


Table 5.4 Instructions for Transferring Data within the Processor

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

Memory access instructions

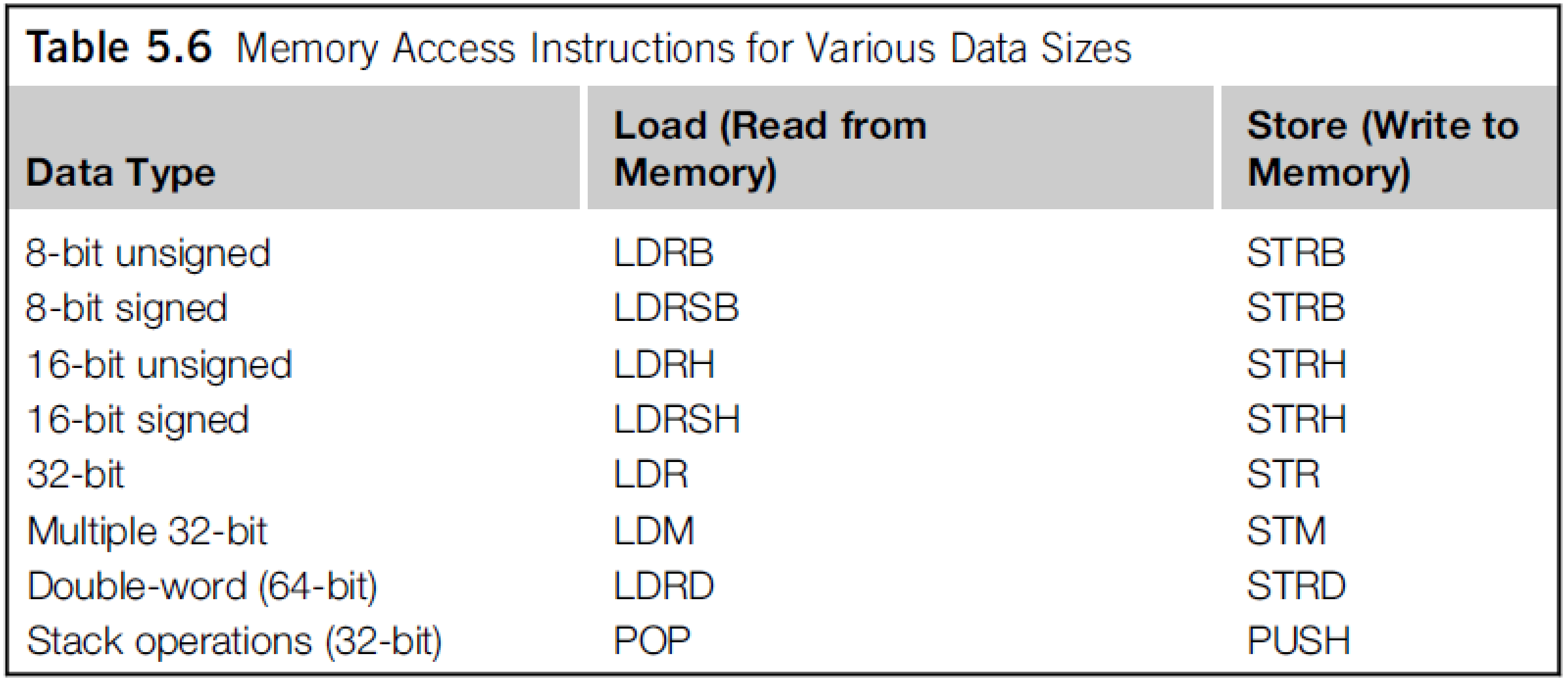


Table 5.6 Memory Access Instructions for Various Data Sizes

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

Immediate Offset (pre-index) and Write Back

- The memory address of the data transfer is the sum of a register value and an immediate constant value (offset), also referred to as “pre-index” addressing. For example:

```
LDRB R0, [R1, #0x3] ; Read a byte value from address R1+0x3, and store the read data in R0.
                    ; The offset value can be positive or negative.
```

- This addressing mode supports write back of the register holding the address. For example:

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

Memory Access instructions with Immediate Offset

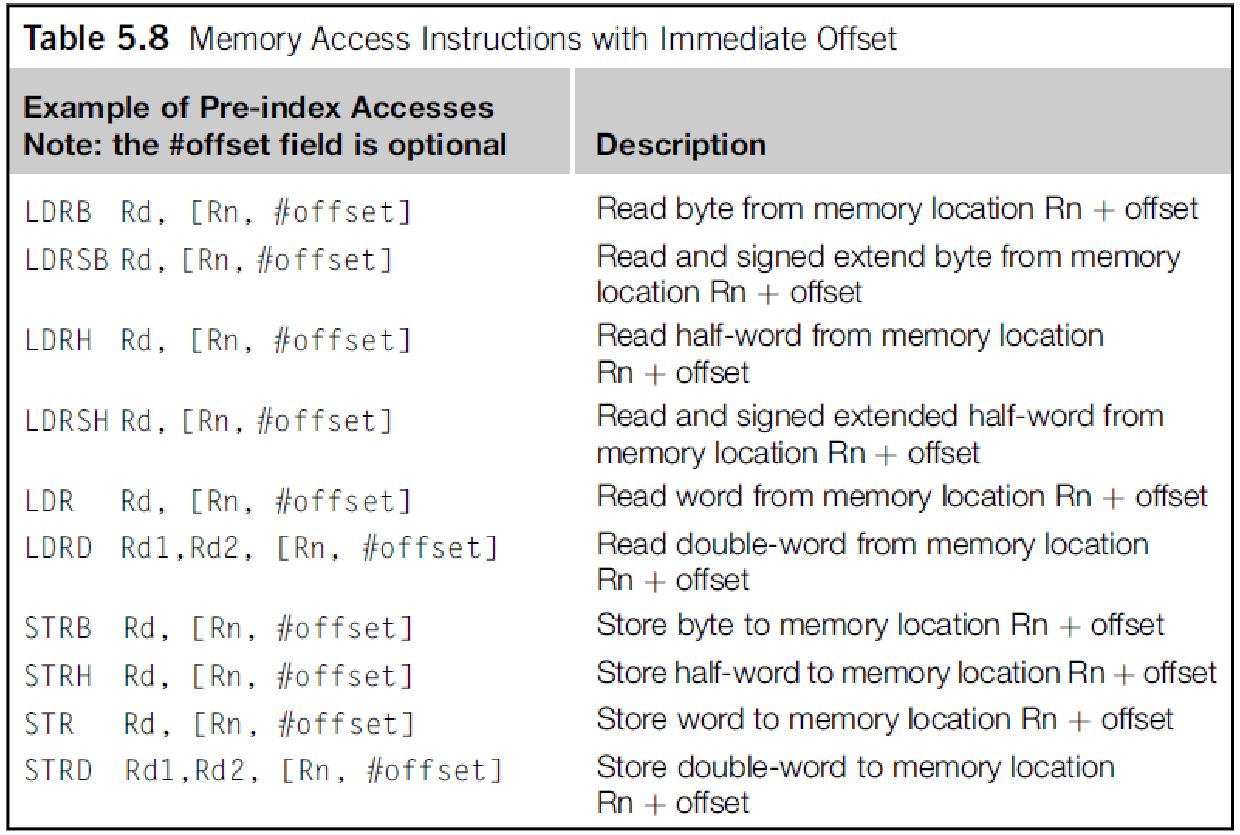


Table 5.8 Memory Access Instructions with Immediate Offset

Example of Pre-index Accesses Note: the #offset field is optional	**Description**
`LDRB Rd, [Rn, #offset]`	Read byte from memory location Rn + offset
`LDRSB Rd, [Rn, #offset]`	Read and signed extend byte from memory location Rn + offset
`LDRH Rd, [Rn, #offset]`	Read half-word from memory location Rn + offset
`LDRSH Rd, [Rn, #offset]`	Read and signed extended half-word from 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

Memory Access instructions with Immediate Offset and Write Back

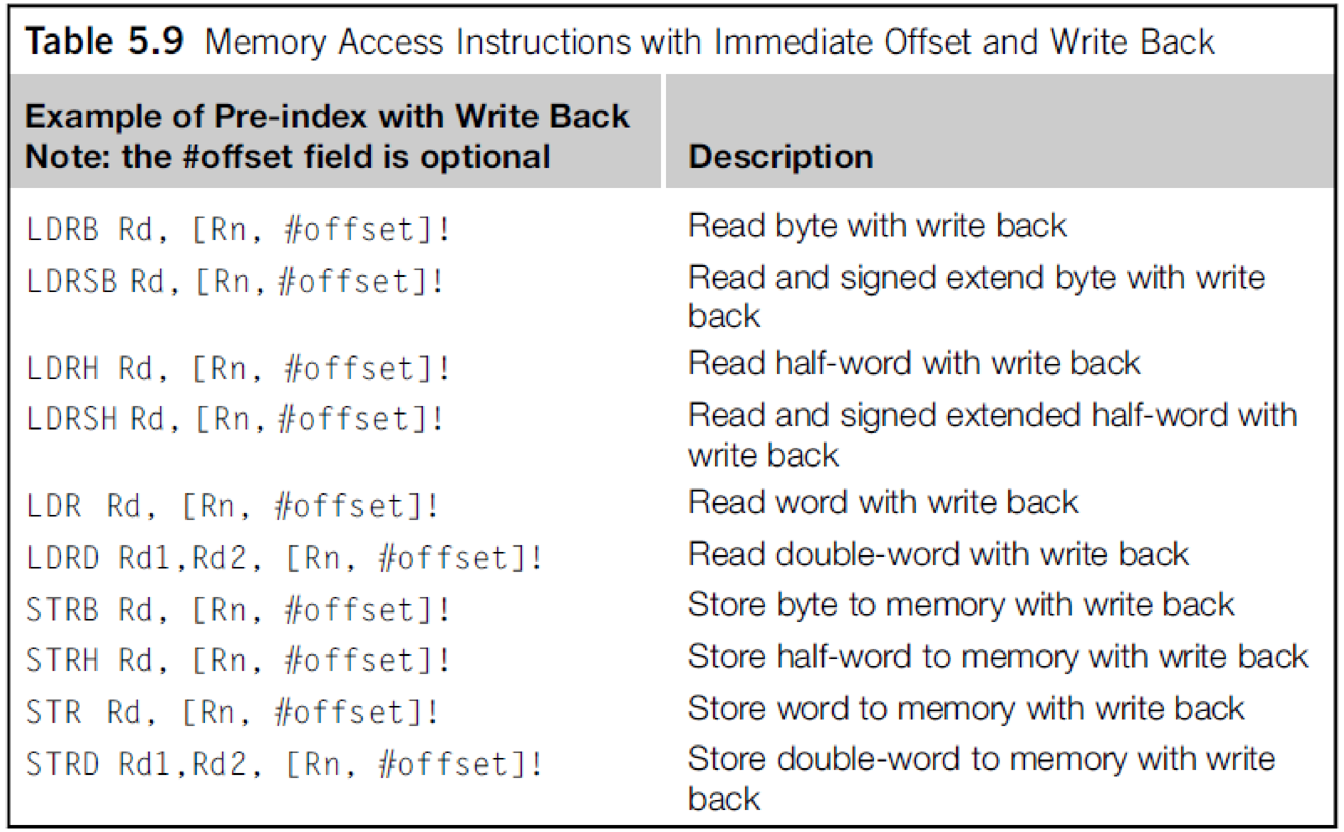


Table 5.9 Memory Access Instructions with Immediate Offset and Write Back

Example of Pre-index with Write Back Note: the #offset field is optional	Description
`LDRB Rd, [Rn, #offset]!`	Read byte with write back
`LDRSB Rd, [Rn, #offset]!`	Read and signed extend byte with write back
`LDRH Rd, [Rn, #offset]!`	Read half-word with write back
`LDRSH Rd, [Rn, #offset]!`	Read and signed extended half-word with write back
`LDR Rd, [Rn, #offset]!`	Read word with write back
`LDRD Rd1,Rd2, [Rn, #offset]!`	Read double-word with write back
`STRB Rd, [Rn, #offset]!`	Store byte to memory with write back
`STRH Rd, [Rn, #offset]!`	Store half-word to memory with write back
`STR Rd, [Rn, #offset]!`	Store word to memory with write back
`STRD Rd1,Rd2, [Rn, #offset]!`	Store double-word to memory with write back

Memory access instructions with PC related addressing

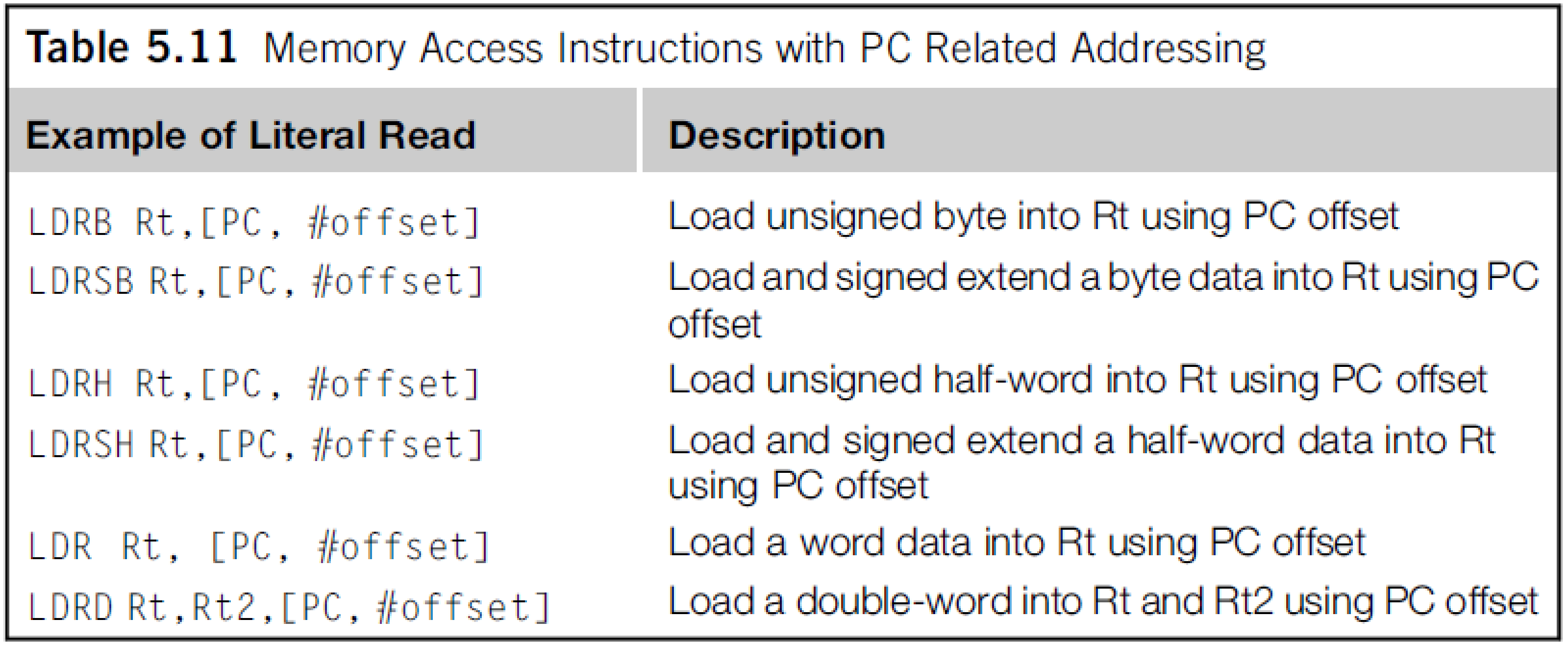


Table 5.11 Memory Access Instructions with PC Related Addressing

Example of Literal Read	Description
`LDRB Rt,[PC, #offset]`	Load unsigned byte into Rt using PC offset
`LDRSB Rt,[PC, #offset]`	Load and signed extend a byte data into Rt using PC offset
`LDRH Rt,[PC, #offset]`	Load unsigned half-word into Rt using PC offset
`LDRSH Rt,[PC, #offset]`	Load and signed extend a half-word data into Rt using PC offset
`LDR Rt, [PC, #offset]`	Load a word data into Rt using PC offset
`LDRD Rt,Rt2,[PC, #offset]`	Load a double-word into Rt and Rt2 using PC offset

Register offset (pre-index)

- This is often used in the processing of data arrays where the address is a combination of a base address and offset calculated form an index value. To make this address calculation even more efficient, the index value can be shifted by a distance of 0 to 3 bits. For example:

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

- Refer to earlier demo: *Corrupting the stack – Array index out of bounds*

Memory access instructions with Register Offset

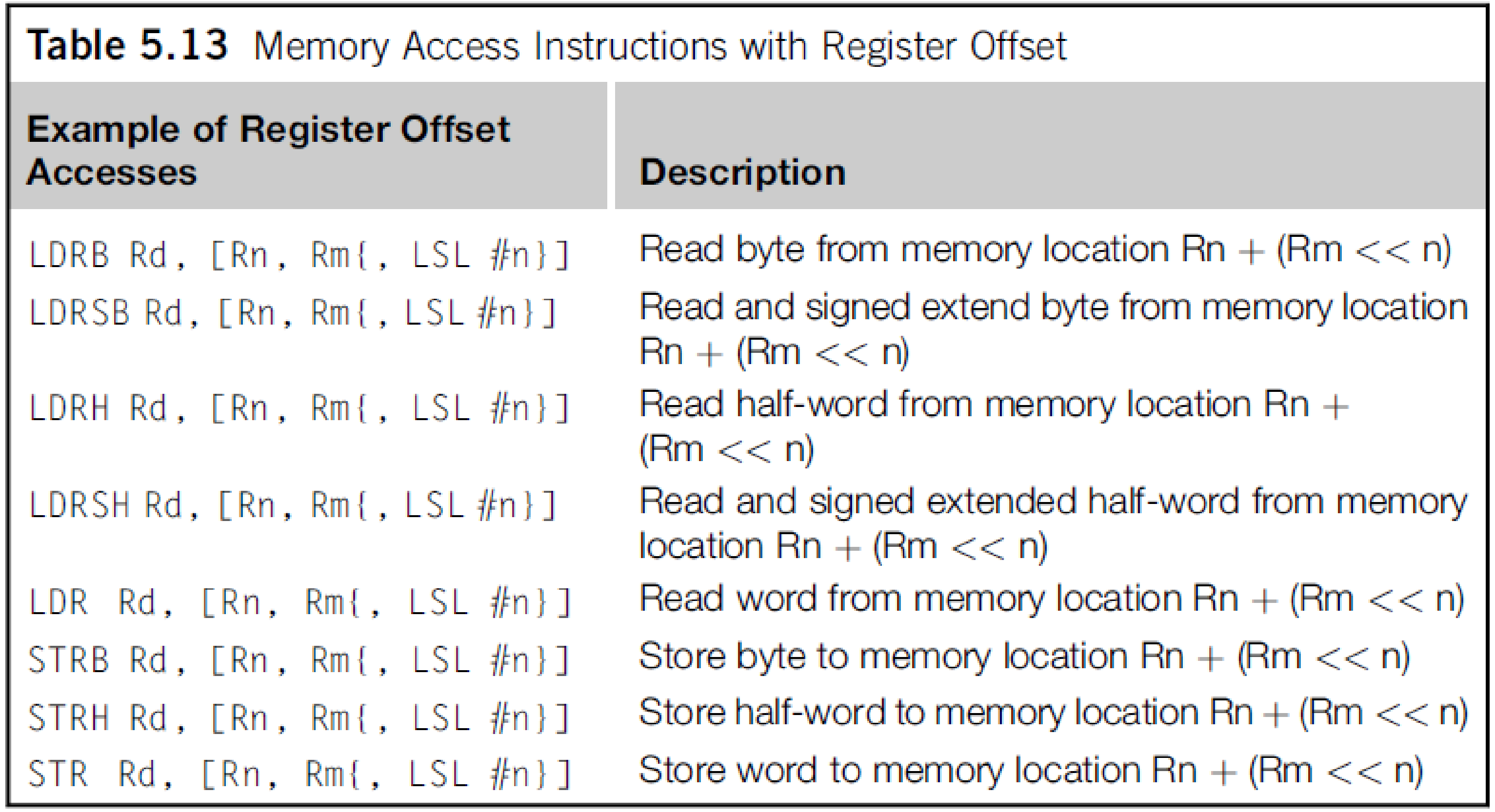


Table 5.13 Memory Access Instructions with Register Offset

Example of Register Offset Accesses	Description
LDRB Rd, [Rn, Rm{, LSL #n}]	Read byte from memory location Rn + (Rm << n)
LDRSB Rd, [Rn, Rm{, LSL #n}]	Read and signed extend byte from memory location Rn + (Rm << n)
LDRH Rd, [Rn, Rm{, LSL #n}]	Read half-word from memory location Rn + (Rm << n)
LDRSH Rd, [Rn, Rm{, LSL #n}]	Read and signed extended half-word from memory location Rn + (Rm << n)
LDR Rd, [Rn, Rm{, LSL #n}]	Read word from memory location Rn + (Rm << n)
STRB Rd, [Rn, Rm{, LSL #n}]	Store byte to memory location Rn + (Rm << n)
STRH Rd, [Rn, Rm{, LSL #n}]	Store half-word to memory location Rn + (Rm << n)
STR Rd, [Rn, Rm{, LSL #n}]	Store word to memory location Rn + (Rm << n)

Register offset (post-index)

- Memory access instructions with post-index addressing mode also have an immediate offset value. However, the offset is not used during the memory access, but is used to update the address register after the data transfer is completed. For example:

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

- When the post-index memory addressing mode is used, there is no need to use the exclamation mark (!) sign.
- cannot be used with R15(PC) o R14(SP).
- The offset value can be positive or negative.

Memory access instructions with Post-Indexing

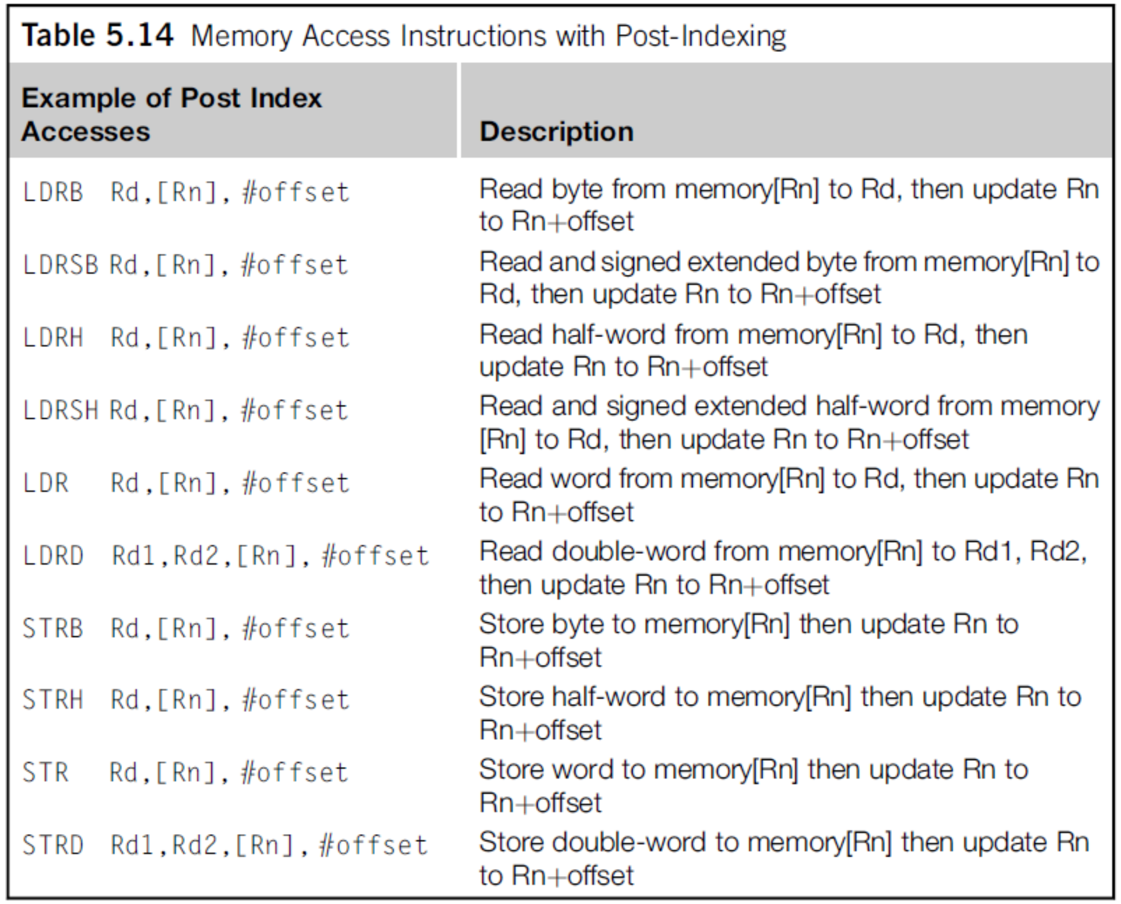


Table 5.14 Memory Access Instructions with Post-Indexing

Example of Post Index Accesses	Description
`LDRB  Rd,[Rn], #offset`	Read byte from memory[Rn] to Rd, then update Rn to Rn+offset
`LDRSB Rd,[Rn], #offset`	Read and signed extended byte from memory[Rn] to Rd, then update Rn to Rn+offset
`LDRH  Rd,[Rn], #offset`	Read half-word from memory[Rn] to Rd, then update Rn to Rn+offset
`LDRSH Rd,[Rn], #offset`	Read and signed extended half-word from memory [Rn] to Rd, then update Rn to Rn+offset
`LDR   Rd,[Rn], #offset`	Read word from memory[Rn] to Rd, then update Rn to Rn+offset
`LDRD  Rd1,Rd2,[Rn], #offset`	Read double-word from memory[Rn] to Rd1, Rd2, then update Rn to Rn+offset
`STRB  Rd,[Rn], #offset`	Store byte to memory[Rn] then update Rn to Rn+offset
`STRH  Rd,[Rn], #offset`	Store half-word to memory[Rn] then update Rn to Rn+offset
`STR   Rd,[Rn], #offset`	Store word to memory[Rn] then update Rn to Rn+offset
`STRD  Rd1,Rd2,[Rn], #offset`	Store double-word to memory[Rn] then update Rn to Rn+offset

Multiple load and multiple store

- The ARM architecture allows reading or writing multiple data that are contiguous in memory.
  - The LDM (Load Multiple registers) & STM (Store Multiple registers) instructions.
  - Only support 32-bit data
- Two type of pre-indexing:
  - IA: **I**ncrement address **A**fter each read/write
  - DB: **D**ecrement address **B**efore each read/write
- For example, the following instructions read address 0x20000000 to 0x2000000F (four words) into R0 to R3:

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

Multiple Load/Store Memory Access Instructions

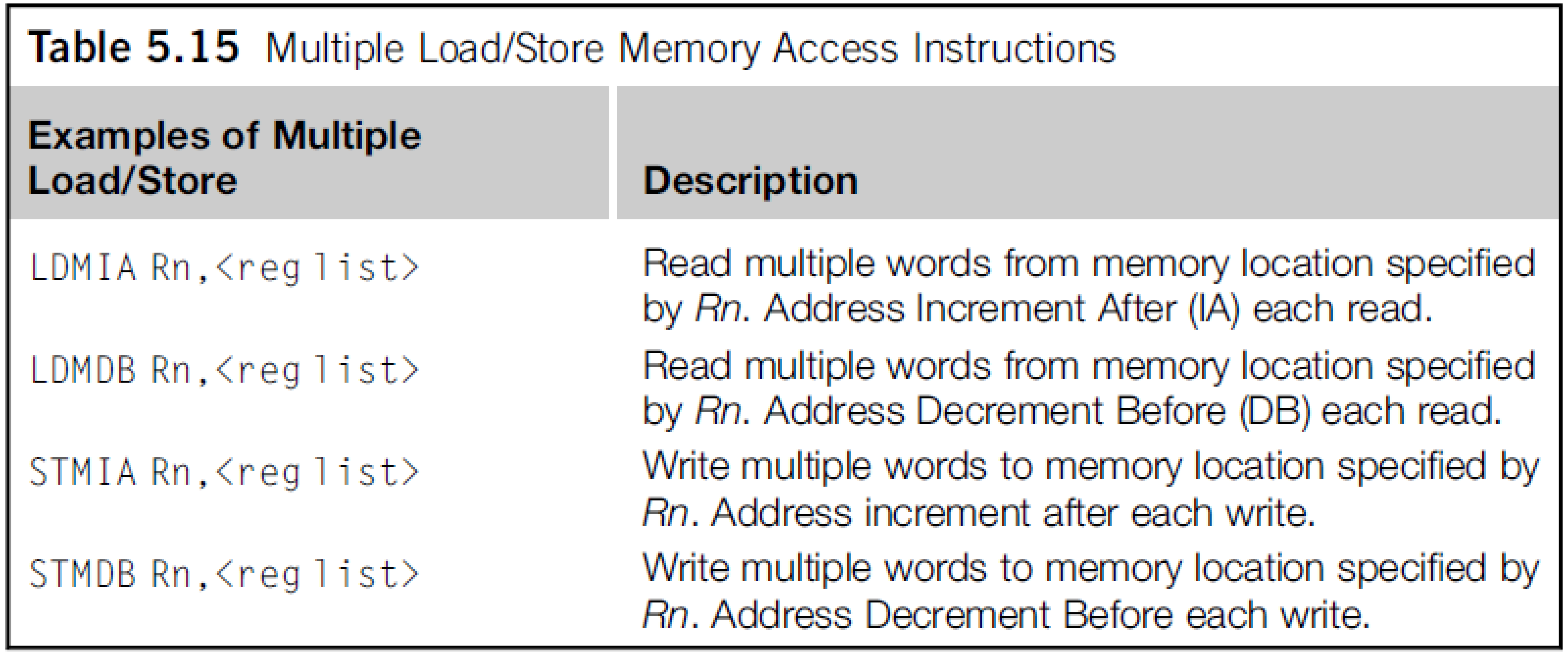


Table 5.15 Multiple Load/Store Memory Access Instructions

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

Multiple Load/Store Memory Access Instructions with Write Back

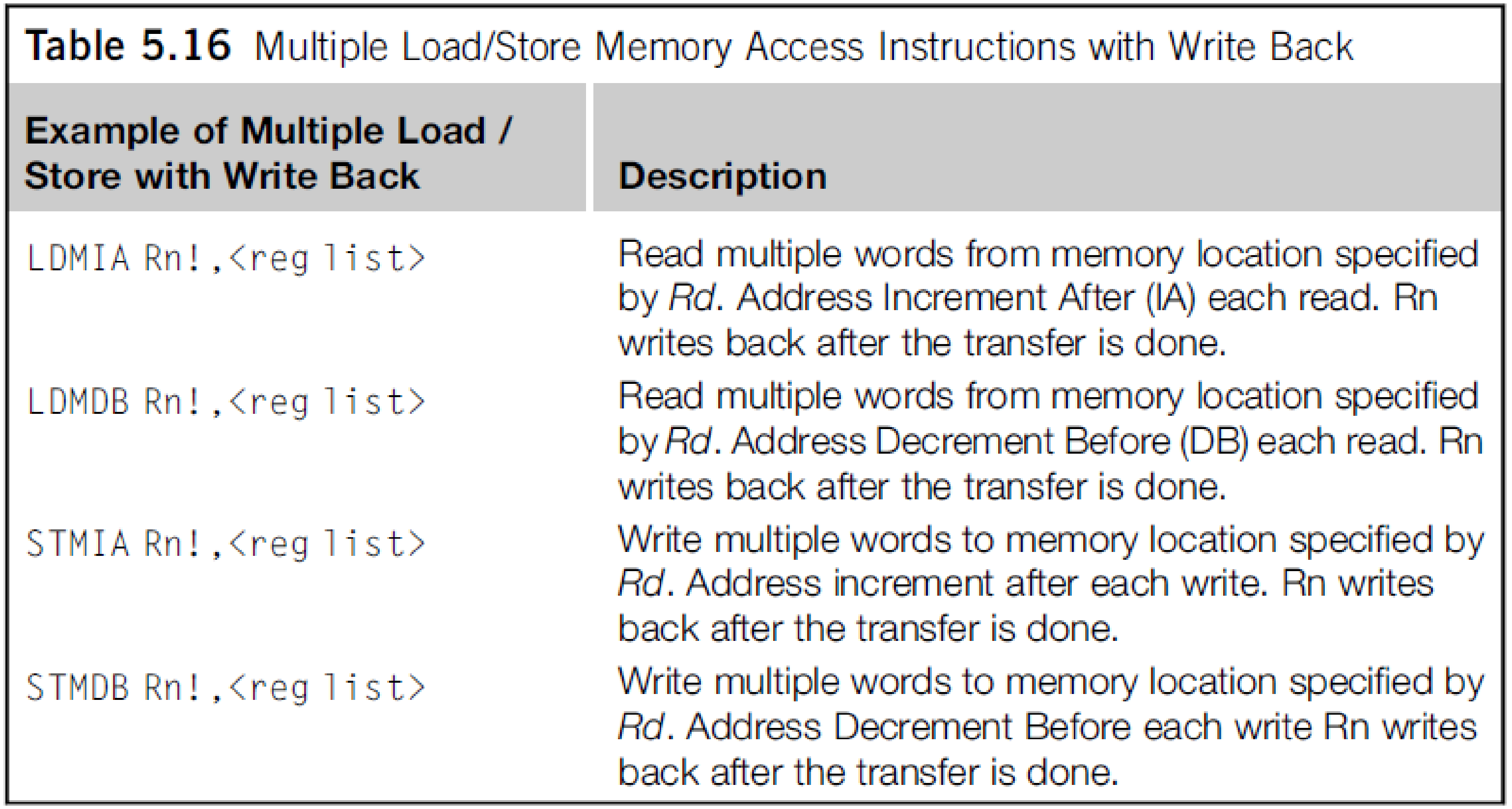


Table 5.16 Multiple Load/Store Memory Access Instructions with Write Back

Example of Multiple Load / Store with Write Back	Description
`LDMIA Rn!,<reg list>`	Read multiple words from memory location specified by *Rd*. Address Increment After (IA) each read. Rn writes back after the transfer is done.
`LDMDB Rn!,<reg list>`	Read multiple words from memory location specified by *Rd*. Address Decrement Before (DB) each read. Rn writes back after the transfer is done.
`STMIA Rn!,<reg list>`	Write multiple words to memory location specified by *Rd*. Address increment after each write. Rn writes back after the transfer is done.
`STMDB Rn!,<reg list>`	Write multiple words to memory location specified by *Rd*. Address Decrement Before each write Rn writes back after the transfer is done.

Stack Push & Stack Pop instructions

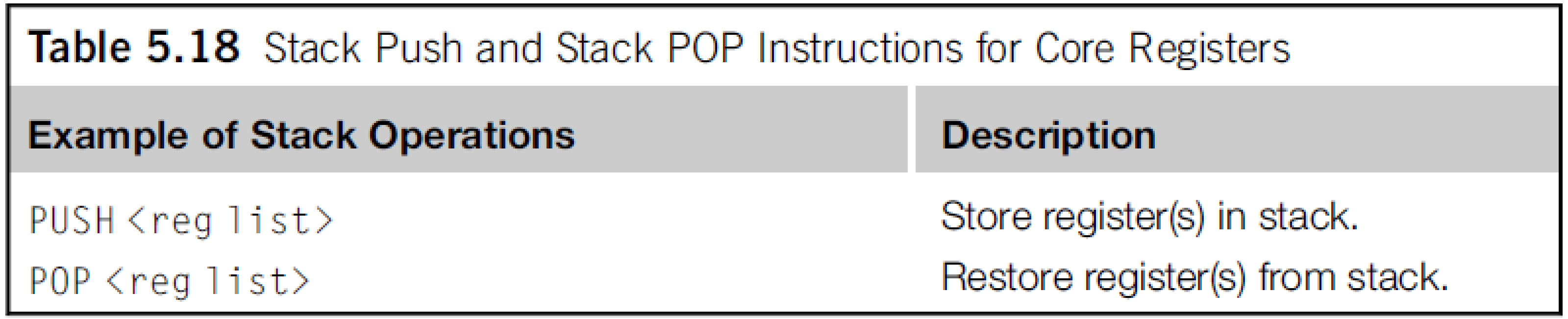


Table 5.18 Stack Push and Stack POP Instructions for Core Registers

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

Example:

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

SP – relative addressing

- As the stack memory is very often used for local variables, accessing these variables requires SP-relative addressing.
- At the beginning of a function the SP value can be decremented to reserve space for local variables
- The local variables can then be accessed using SP-related addressing.
- At the end of the function, the SP is incremented to return to the original value, which frees the allocated stack space before returning to the calling code.

SP – relative addressing

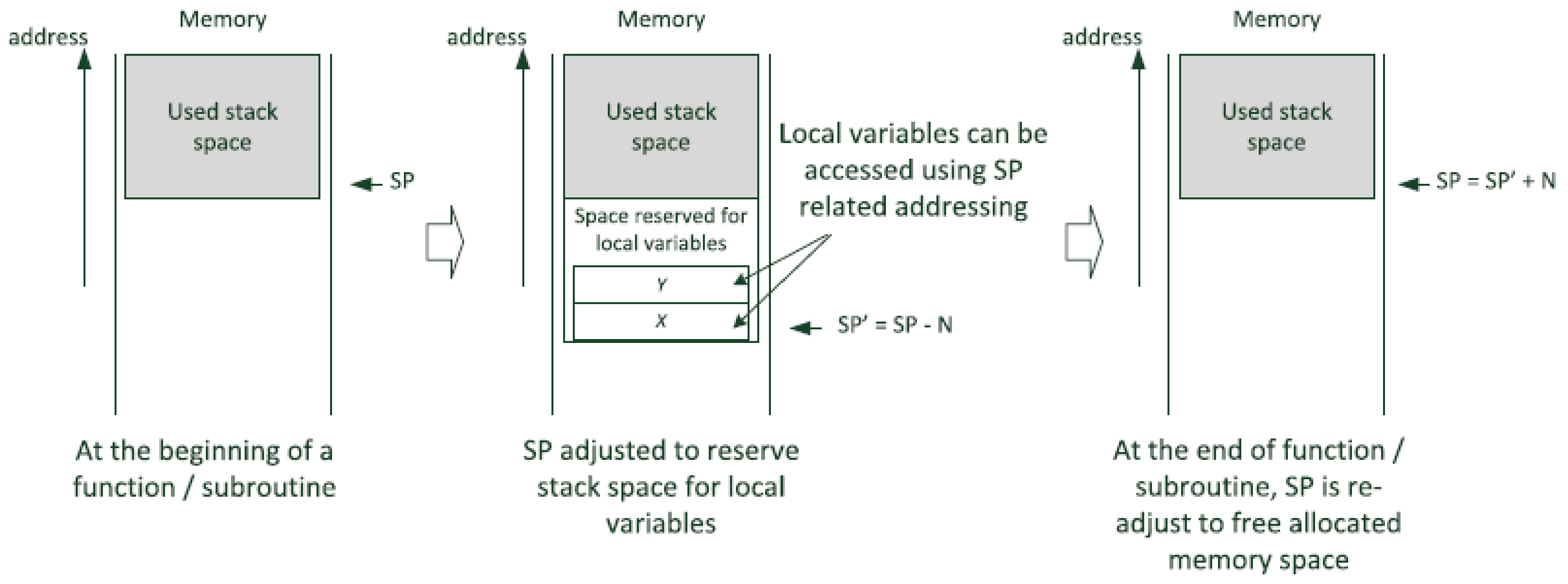

address
Memory
Used stack space
SP
At the beginning of a function / subroutine
address
Memory
Used stack space
Space reserved for local variables
Y
X
Local variables can be accessed using SP related addressing
SP' = SP - N
SP adjusted to reserve stack space for local variables
address
Memory
Used stack space
SP = SP' + N
At the end of function / subroutine, SP is re-adjust to free allocated memory space

FIGURE 5.5

Local variable space allocation and accesses in stack

Arithmetic Data Operations

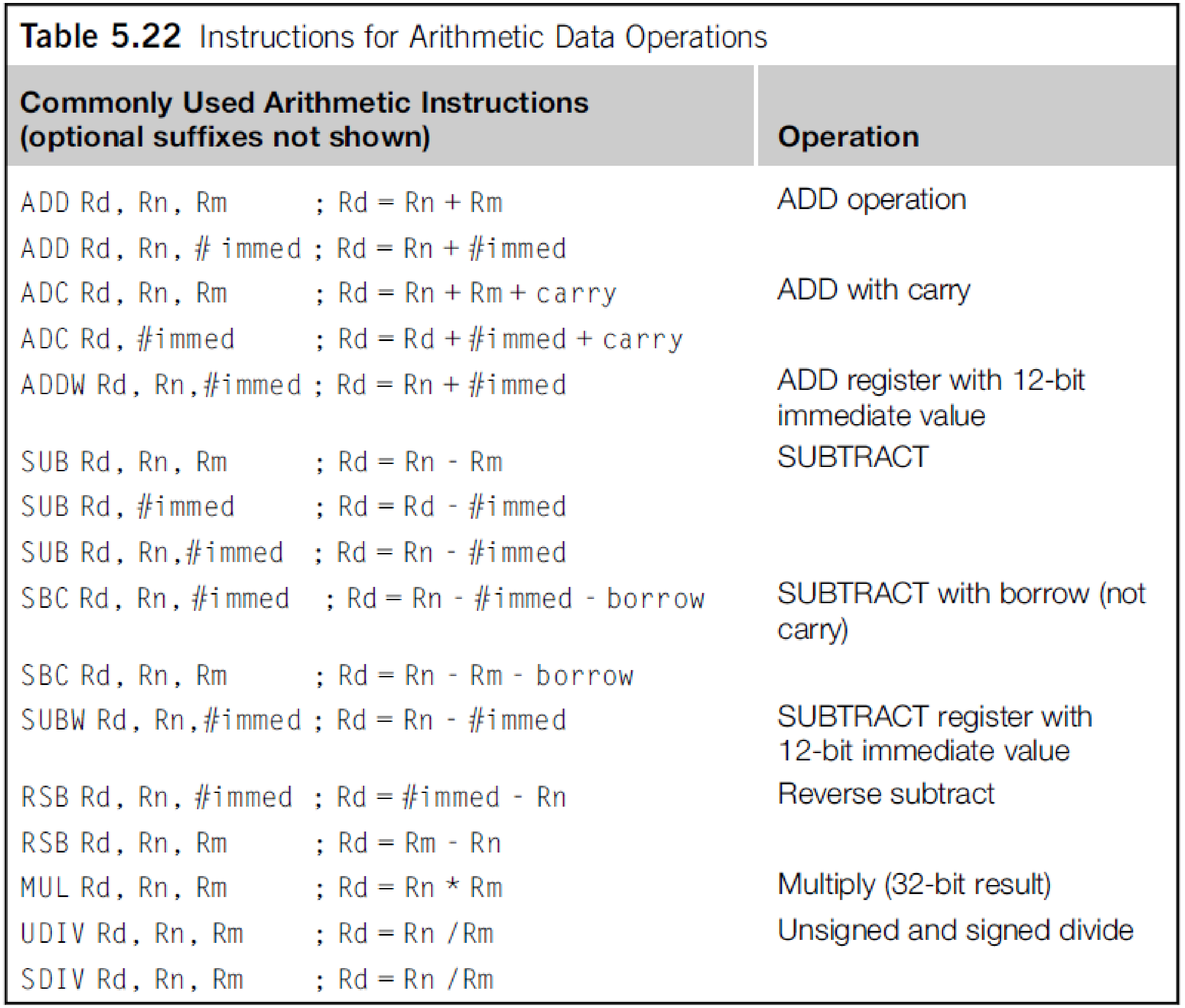


Table 5.22 Instructions for Arithmetic Data Operations

Commonly Used Arithmetic Instructions (optional suffixes not shown)	**Operation**
ADD Rd, Rn, Rm ; Rd = Rn + Rm	ADD operation
ADD Rd, Rn, # immed ; Rd = Rn + #immed	
ADC Rd, Rn, Rm ; Rd = Rn + Rm + carry	ADD with 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, Rn, #immed ; Rd = Rn - #immed - borrow	SUBTRACT with borrow (not carry)
SBC Rd, Rn, Rm ; Rd = Rn - Rm - borrow	
SUBW Rd, Rn,#immed ; Rd = Rn - #immed	SUBTRACT register with 12-bit immediate value
RSB Rd, Rn, #immed ; Rd = #immed - Rn	Reverse subtract
RSB Rd, Rn, Rm ; Rd = Rm - Rn	
MUL Rd, Rn, Rm ; Rd = Rn * Rm	Multiply (32-bit result)
UDIV Rd, Rn, Rm ; Rd = Rn /Rm	Unsigned and signed divide
SDIV Rd, Rn, Rm ; Rd = Rn /Rm	

Logical Operations

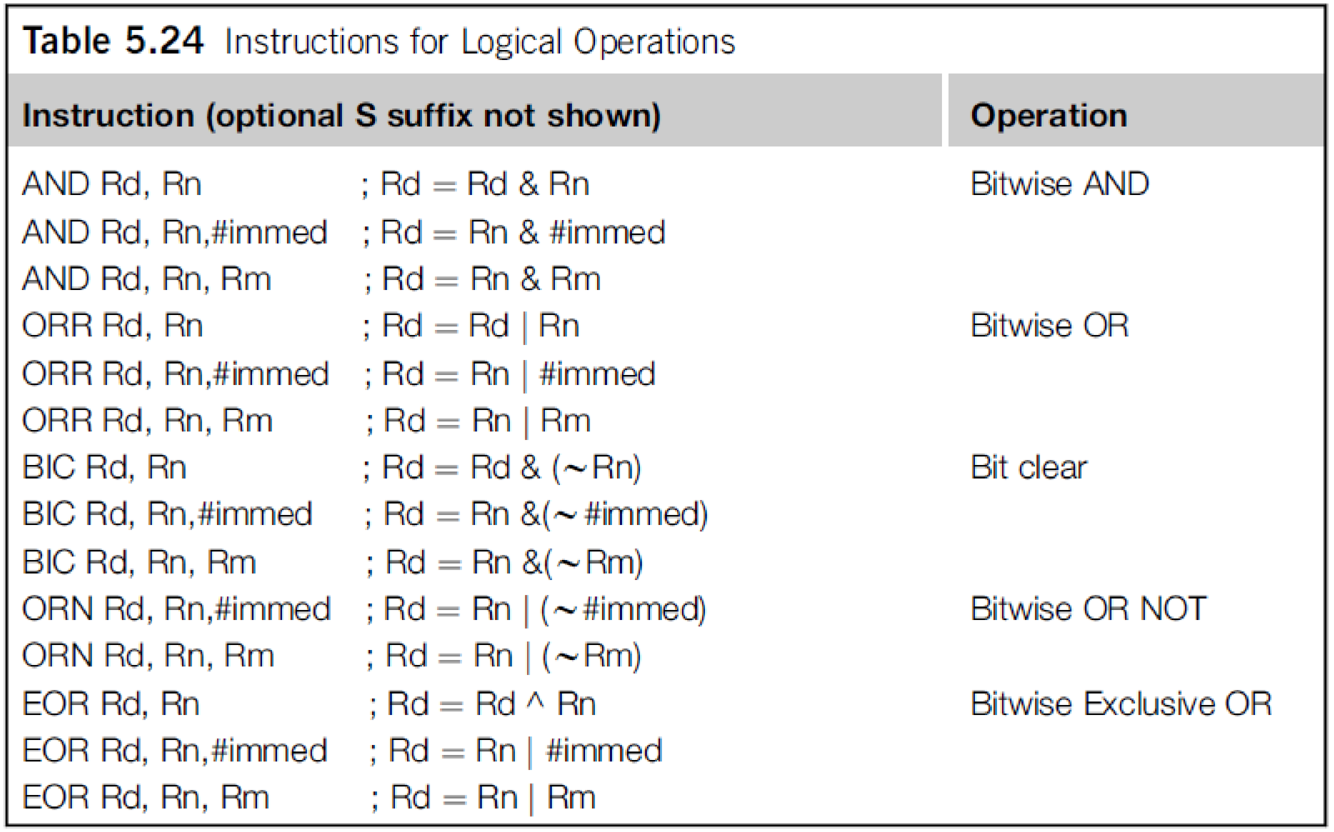


Table 5.24 Instructions for Logical Operations

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

Shift and Rotate Operations

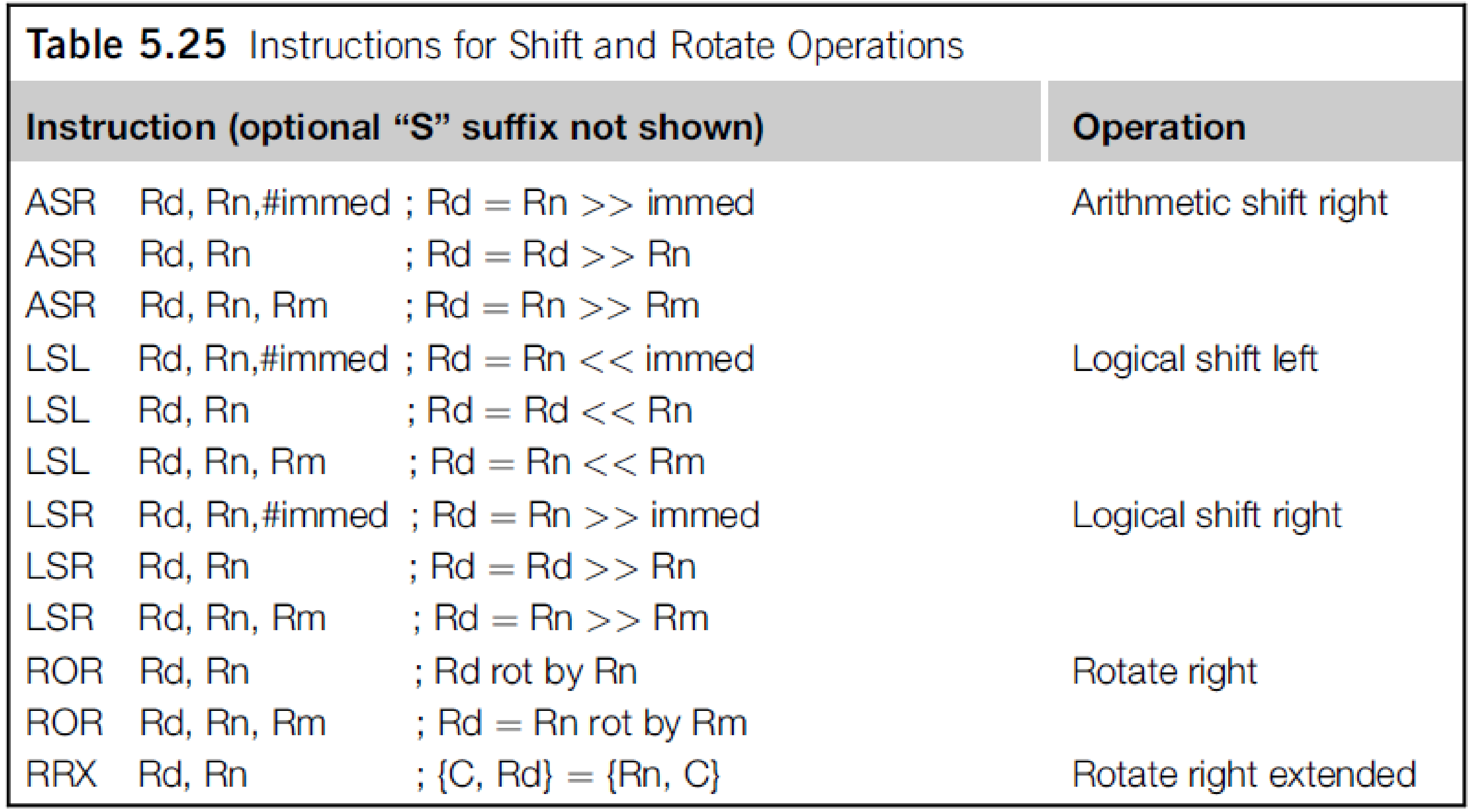
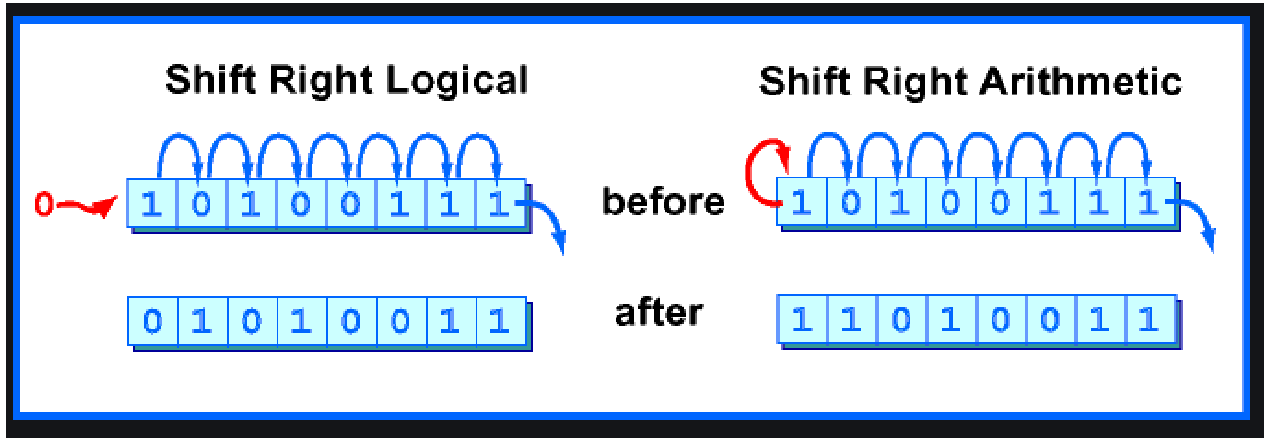


Table 5.25 Instructions for Shift and Rotate Operations

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

Notes - Shift and Rotate Operations

- Left rotation of n = 11100101 by 3 makes n = 00101111 (Left shifted by 3 and first 3 bits are put back in last)
- Right rotation of n = 11100101 by 3 makes n = 10111100 (Right shifted by 3 and last 3 bits are put back in first)
- Arithmetic shift preserve sign bit, whereas Logical shift can not preserve sign bit.



Source: [*https://chortle.ccsu.edu/assemblytutorial/Chapter-14/ass14_13.html*](https://chortle.ccsu.edu/assemblytutorial/Chapter-14/ass14_13.html)

Compare and Test Instructions

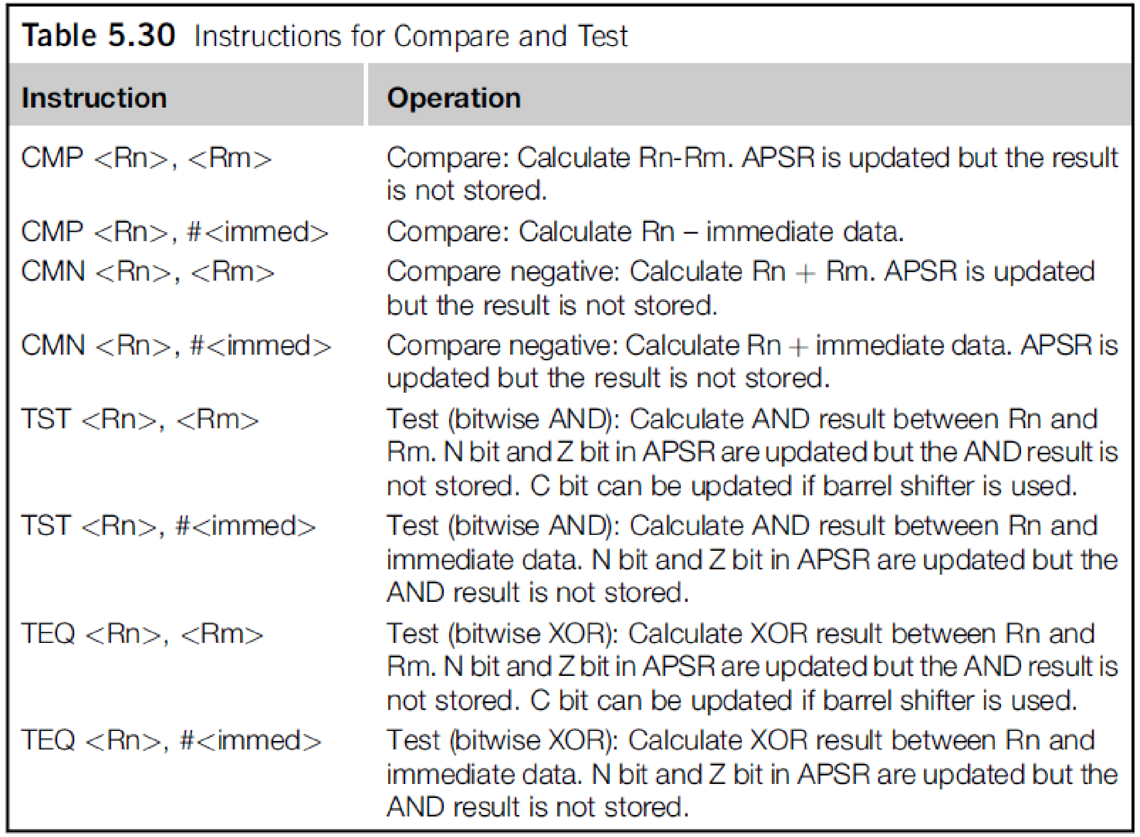


Table 5.30 Instructions for Compare and Test

Instruction	Operation
CMP <Rn>, <Rm>	Compare: Calculate Rn-Rm. APSR is updated but the result is not stored.
CMP <Rn>, #<immed>	Compare: Calculate Rn – immediate data.
CMN <Rn>, <Rm>	Compare negative: Calculate Rn + Rm. APSR is updated but the result is not stored.
CMN <Rn>, #<immed>	Compare negative: Calculate Rn + immediate data. APSR is updated but the result is not stored.
TST <Rn>, <Rm>	Test (bitwise AND): Calculate AND result between Rn and Rm. N bit and Z bit in APSR are updated but the AND result is not stored. C bit can be updated if barrel shifter is used.
TST <Rn>, #<immed>	Test (bitwise AND): Calculate AND result between Rn and immediate data. N bit and Z bit in APSR are updated but the AND result is not stored.
TEQ <Rn>, <Rm>	Test (bitwise XOR): Calculate XOR result between Rn and Rm. N bit and Z bit in APSR are updated but the AND result is not stored. C bit can be updated if barrel shifter is used.
TEQ <Rn>, #<immed>	Test (bitwise XOR): Calculate XOR result between Rn and immediate data. N bit and Z bit in APSR are updated but the AND result is not stored.

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

- These instructions do not have the “S” suffix because the APSR is always updated.

Branch Instructions

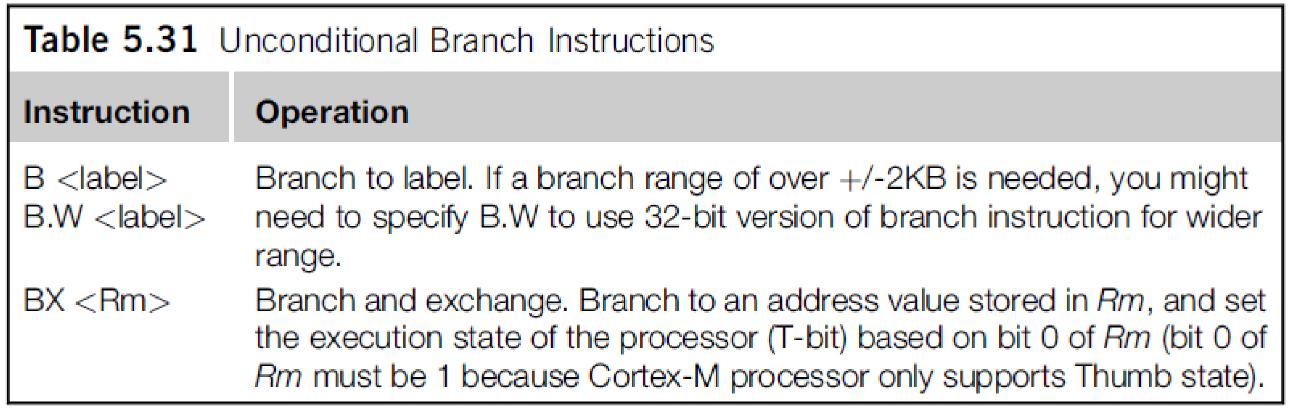


Table 5.31 Unconditional Branch Instructions

Instruction	Operation
B <label> B.W <label>	Branch to label. If a branch range of over +/-2KB is needed, you might need to specify B.W to use 32-bit version of branch instruction for wider range.
BX <Rm>	Branch and exchange. Branch to an address value stored in *Rm*, and set the execution state of the processor (T-bit) based on bit 0 of *Rm* (bit 0 of *Rm* must be 1 because Cortex-M processor only supports Thumb state).

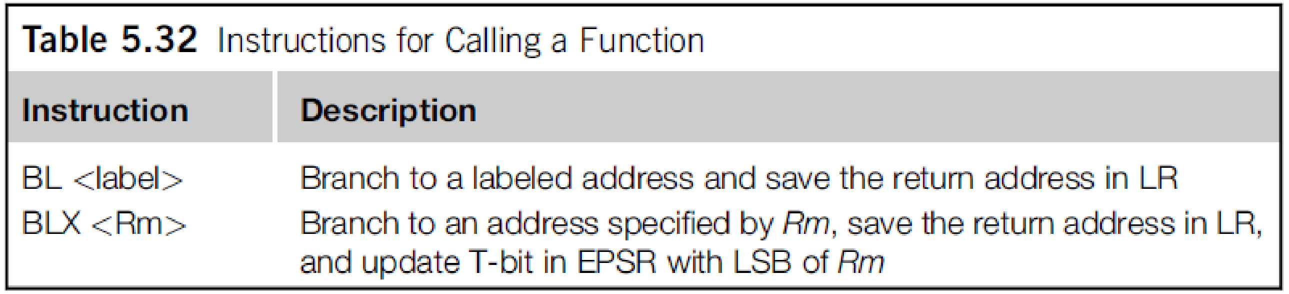
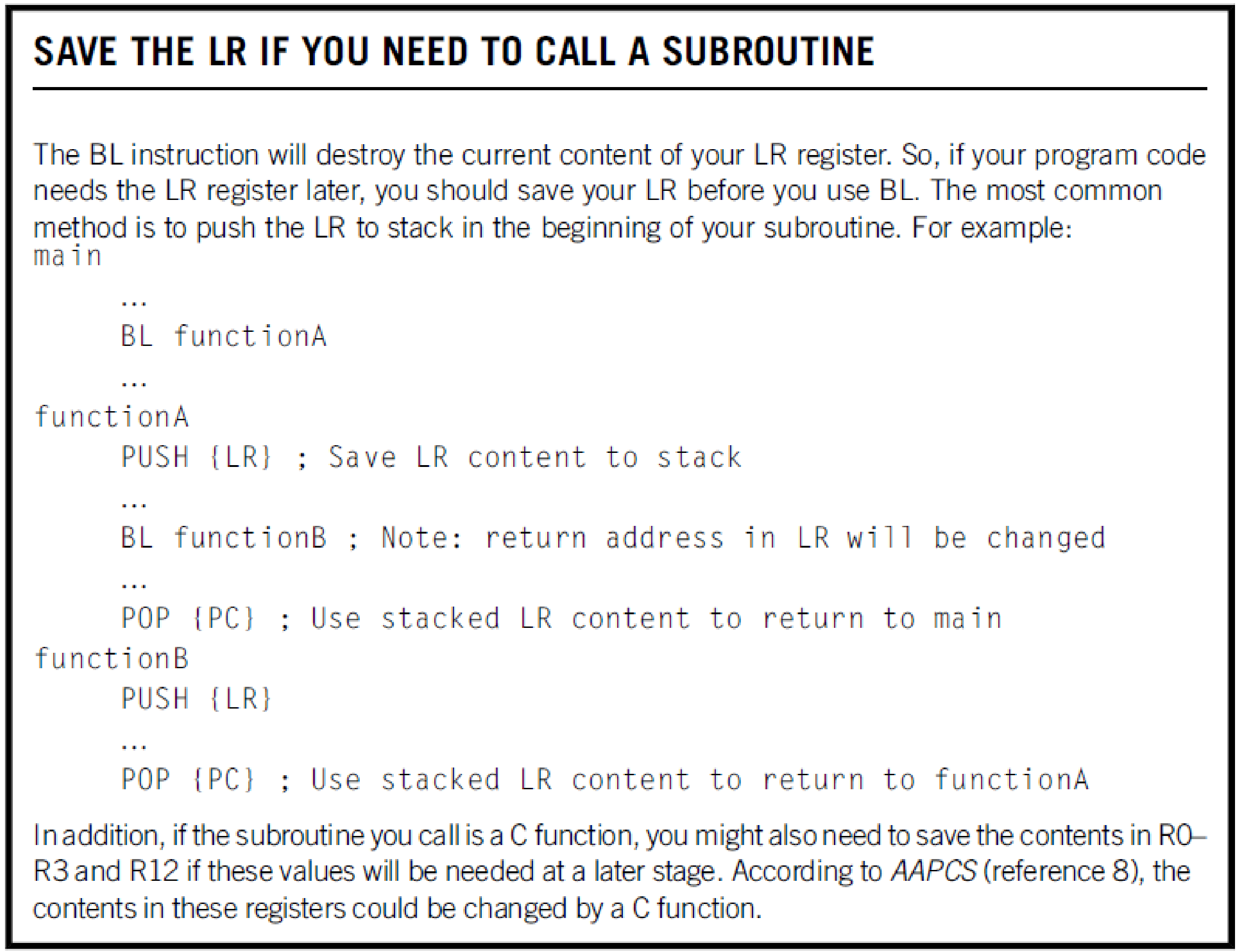


Table 5.32 Instructions for Calling a Function

Instruction	Description
BL <label>	Branch to a labeled address and save the return address in LR
BLX <Rm>	Branch to an address specified by *Rm*, save the return address in LR, and update T-bit in EPSR with LSB of *Rm*

Notes:

- The Program Counter (PC) is set to the branch target address
- The Link Register (LR) is updated to hold the return address if BL is used
- Since the Cortex-M3 and M4 processors only support the Thumb state, the LSb of the register used in a BX or BLX operation must be set to 1. Otherwise, it indicates an attempt to switch to the ARM state and will result in a fault exception.



SAVE THE LR IF YOU NEED TO CALL A SUBROUTINE

The BL instruction will destroy the current content of your LR register. So, if your program code needs the LR register later, you should save your LR before you use BL. The most 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 ; Note: return address in LR will be changed
     ...
     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* (reference 8), the contents in these registers could be changed by a C function.

When calling a subroutine

Conditional branches

Conditional branches are executed conditionally based on the current value in APSR

APSR flags are affected by:

- Instructions with S suffix. EX: ADDS
- Compare (CMP) and Test (TST)
- Writing to APSR directly

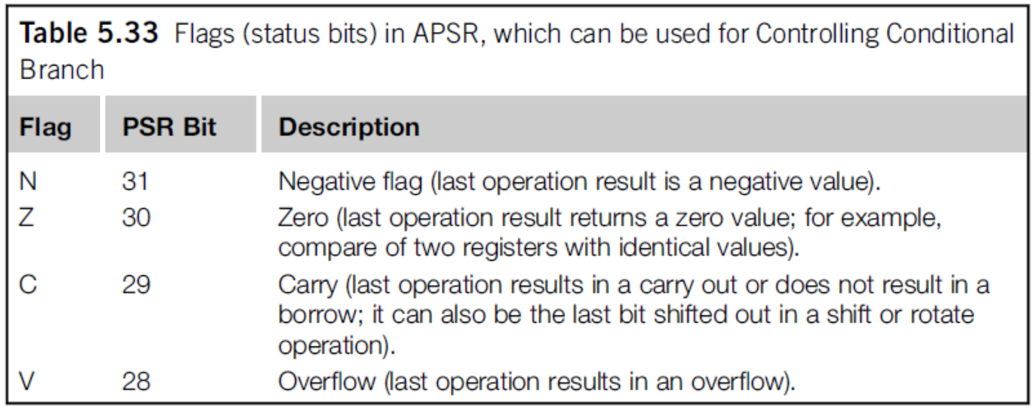


Table 5.33 Flags (status bits) in APSR, which can be used for Controlling Conditional Branch

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; for example, compare of two registers with identical values).
C	29	Carry (last operation results in a carry out or does not result in a borrow; it can also be the last bit shifted out in a shift or rotate operation).
V	28	Overflow (last operation results in an overflow).

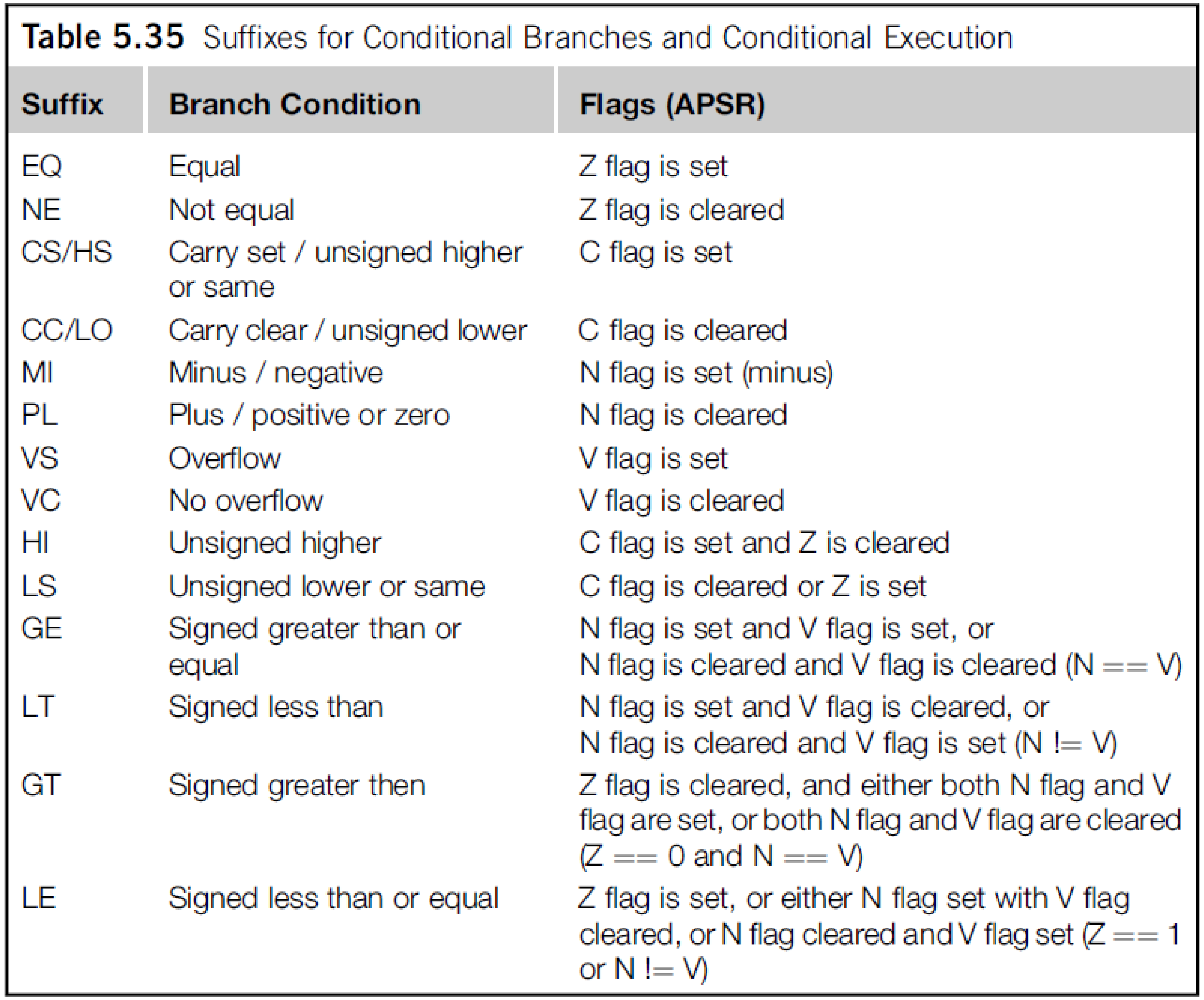


Table 5.35 Suffixes for Conditional Branches and Conditional Execution

Suffix	Branch Condition	Flags (APSR)
EQ	Equal	Z flag is set
NE	Not equal	Z flag is cleared
CS/HS	Carry set / unsigned higher or same	C flag is set
CC/LO	Carry clear / unsigned lower	C flag is cleared
MI	Minus / negative	N flag is set (minus)
PL	Plus / positive or zero	N flag is cleared
VS	Overflow	V flag is set
VC	No overflow	V flag is cleared
HI	Unsigned higher	C flag is set and Z is cleared
LS	Unsigned lower or same	C flag is cleared or Z is set
GE	Signed greater than or equal	N flag is set and V flag is set, or N flag is cleared and V flag is cleared (N == V)
LT	Signed less than	N flag is set and V flag is cleared, or N flag is cleared and V flag is set (N != V)
GT	Signed greater then	Z flag is cleared, and either both N flag and V flag are set, or both N flag and V flag are cleared (Z == 0 and N == V)
LE	Signed less than or equal	Z flag is set, or either N flag set with V flag cleared, or N flag cleared and V flag set (Z == 1 or N != V)

Suffixes for Conditional Branches & Conditional Execution

Instructions for Conditional Branches

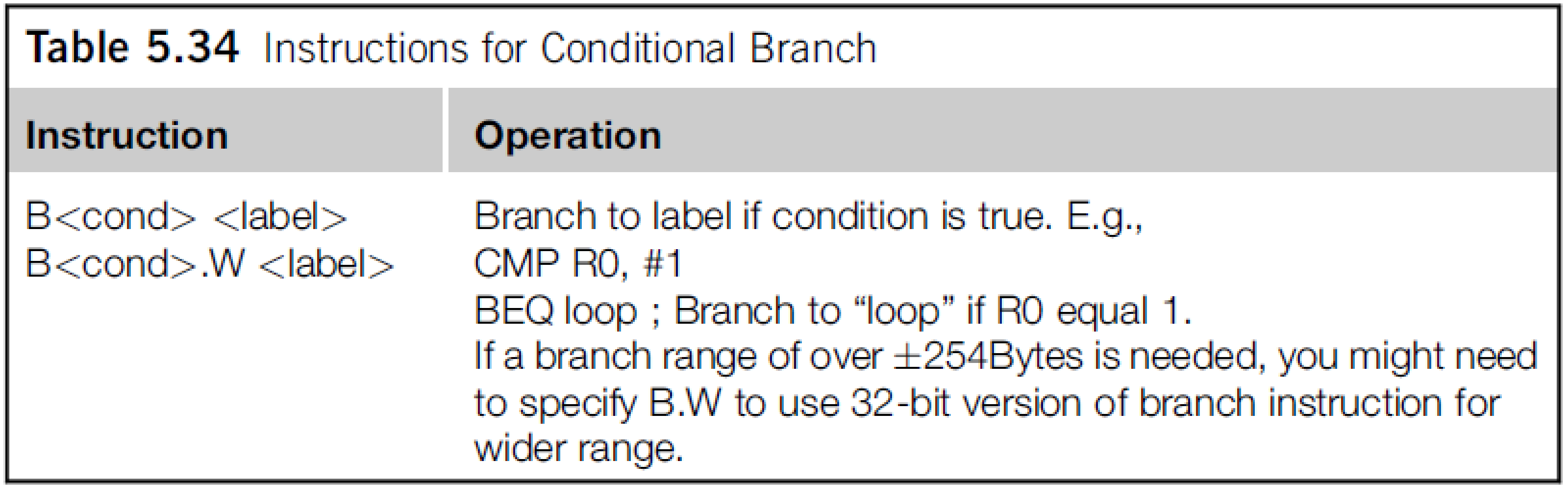


Table 5.34 Instructions for Conditional Branch

Instruction	**Operation**
B<cond> <label> B<cond>.W <label>	Branch to label if condition is true. E.g., CMP R0, #1 BEQ loop ; Branch to "loop" if R0 equal 1. If a branch range of over ±254Bytes is needed, you might need to specify B.W to use 32-bit version of branch instruction for wider range.

Conditional Branche Example

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

p2                  ; label p2
  MOVS R3, #2

p3                  ; label p3
  …                 ; other subsequence operations
```

Instructions for setting and clearing PRIMASK and FAULTMASk

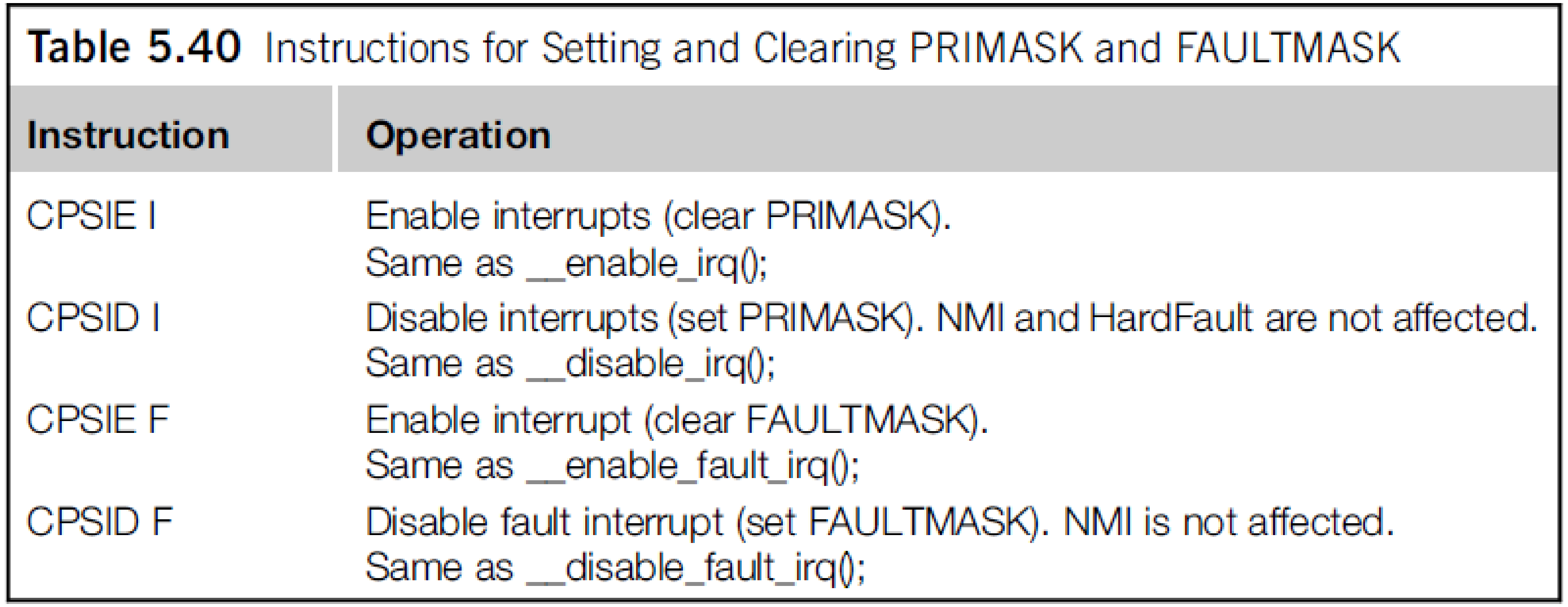
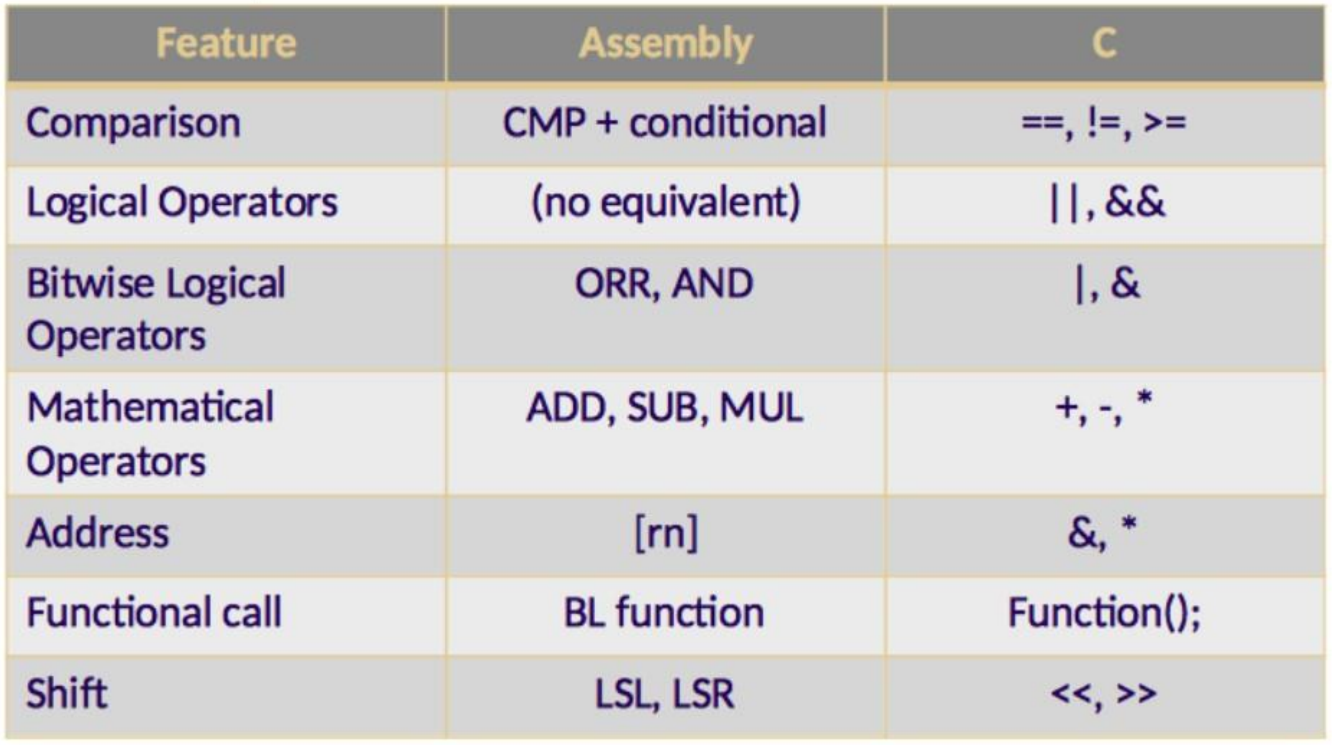


Table 5.40 Instructions for Setting and Clearing PRIMASK and FAULTMASK

Instruction	Operation
CPSIE I	Enable interrupts (clear PRIMASK). Same as __enable_irq();
CPSID I	Disable interrupts (set PRIMASK). NMI and HardFault are not affected. Same as __disable_irq();
CPSIE F	Enable interrupt (clear FAULTMASK). Same as __enable_fault_irq();
CPSID F	Disable fault interrupt (set FAULTMASK). NMI is not affected. Same as __disable_fault_irq();

C and Assembly

Most C compilers generate assembly code from C, then invoke an assembler Many C operations map very closely to assembly language instructions. Some examples



Feature	Assembly	C
Comparison	CMP + conditional	==, !=, >=
Logical Operators	(no equivalent)	||, &&
Bitwise Logical Operators	ORR, AND	|, &
Mathematical Operators	ADD, SUB, MUL	+, -, *
Address	[rn]	&, *
Functional call	BL function	Function();
Shift	LSL, LSR	<<, >>

Assembly function general structure

1. Prolog (saving register contents to the stack memory if necessary)
2. Allocate stack space memory for local variables (decrement SP)
3. Copy some of R0 to R3 (input parameters) to high registers (R8 to R12) for later use (optional)
4. Carry out processing/calculation
5. Store result in R0 if a result is to be returned
6. Stack adjustment to free space for local variables (increment SP)
7. Epilog (restore register values from stack)
8. Return

Demo

Create Assembly Code

(c)copyright 2019 Tamer Awad - EMBSYS100 AU19

Assignment 05

Suggested Reading

- ***"An Embedded Software Primer" by David E. Simon***
  - Chapter 4.3: The Shared-Data Problem
  - Chapter 5: Survey of Software Architecture
- ***"The Definitive Guide to ARM Cortex M3 & M4" by Joseph Yiu (Third Edition)***
  - Chapter 5.6: Instruction set
  - Chapter 20.1, 20.2, 20.3, 20.5
- ***"The Cortex-M4 Device Generic User Guide"***
  - *Chapter 3.1*