CHAPTER 5

Load and Store

A load instruction sets a register to a particular value. The value might be a constant directly specified in the program or a value that is stored in the memory. A store instruction saves the value held in a register to the memory.

5.1 Load Constant into Registers

Many assembly instructions use constant numbers, often called *immediate numbers*. One command usage is to set a register to a specific constant value.

`MOV Rd, #<immed_8>`	Move 8-bit immediate value (0-255) to the register
`MVN Rd, #<immed_8>`	Move the bitwise inverse of 8-bit immediate value (0-255) to the register
`MOVT Rd, #<immed_16>`	Move 16-bit immediate value to top halfword [31:16] of the register. Bottom halfword unaltered.
`MOVW Rd, #<immed_16>`	Move 16-bit immediate value to bottom halfword [15:0] of the register and clear top halfword [31:16]
`LDR Rt, =#<immed_8>`	Equivalent to `MOV`
`LDR Rt, =#<immed_32>`	A pseudo instruction

Table 5-1. Instructions for loading constants into a register.

5.1.1 Data Movement Instruction MOV and MVN

All immediate numbers start with a "#" sign. If the immediate number is less than 8 bits, we can use `MOV` to set the register value.

```
MOV r0, #0xFF          ; Set r0 to the hexadecimal value 0xFF
MOV r0, #0b10011100    ; Set r0 to the binary value 10011100
MOV r0, #54            ; Set r0 to the decimal value 54
MOV r0, #0d54          ; Set r0 to the decimal value 54
```

If the immediate number has 32 bits, we can use MOV to set the register value if the immediate number can be obtained by using the following format:

```
#immed_32 = #immed_8 ROR (2 × #immed_4)
```

where ROR is the circular right rotate. For example, right rotating 0xAF by 24 bits can get 0x0000AF00.

```
0x0000AF00 = 0xAF ROR (2×12)
```

Besides, these instructions can also use a few 32-bit values with some regular patterns, such as 0xABABABAB, 0x00AB00AB, and 0xAB00AB00.

5.1.2 Pseudo Instruction LDR and ADR

A *pseudo instruction* is an instruction that is available to use in an assembly program, but not directly supported by the microprocessor. Compilers translate it to one or multiple actual machine instructions when the assembler builds the program into an executable. Pseudo instructions are provided for the convenience of programmers.

As introduced later in this chapter, LDR loads data from the memory to a register. However, LDR can be a pseudo instruction that loads an immediate number into a register. A pseudo instruction is not a real machine instruction, but it provides convenience for programmers and improves the readability of programs. The assembler translates a pseudo instruction into one or multiple actual machine instructions.

```
        LDR  r0, =array       ; Pseudo instruction
        LDR  r1, [r0]         ; Not a pseudo instruction

        LDR  r2, =0x12345678  ; Pseudo instruction
        ADD  r1, r1, r2       ; r1 = r1 + r2
        STR  r1, [r0]         ; Save r1 to memory

        AREA myData, DATA     ; Directive: declare a data area
        ALIGN                 ; Directive: align on a word boundary
                              ; Allocate padding bytes if necessary to make the
                              ; array to align properly
array DCW  1, 2, 3, 4, 5
```

Example 5-1. Using LDR pseudo instruction to load a memory address or an immediate number into a register.

Another widely used pseudo instruction in ARM assembly language is ADR (stands for address), which sets a register to a memory address within a certain range, as shown in Example 5-2. The syntax difference between LDR and ADR is that LDR needs an equal sign ("=") but ADR does not. The assembler translates an ADR instruction into an ADD or SUB instruction with one source operand as PC.

Also, the pseudo instruction LDR is different from the LDR instruction for accessing memory. For example, "LDR r1, =0x12345678" is a pseudo instruction, and "LDR r1, [r0]" is a real machine instruction that loads a word from the memory. The assembler can distinguish them by checking the format of the operands specified in LDR.

```
loop ADD r1, r2, r3
     ADR r4, loop    ; A pseudo instruction, translated to "SUB r4,pc,#12"
```

Example 5-2. Using ADR pseudo instruction to load a memory address into a register

5.1.3 Comparison of LDR, ADR, and MOV

While ADR can only load a memory address label into a register, LDR is more versatile and can load an immediate number up to 32 bits. The real instructions translated from the LDR pseudo instruction depend on the immediate number.

- If the constant number can fit into the 12-bit immediate number format used by a MOV or MVN instruction, compilers translate the LDR pseudo instruction to MOV or MVN.
- Otherwise, compilers translate it into a regular LDR instruction that uses PC-relative memory address.

LDR can load a 32-bit constant to a register.

MOV can only load a 12-bit constant into a register.

ADR can load a memory address.

In the latter case, the immediate numbers are directly stored together with the instruction code in the machine executable. As introduced in Chapter 1.2, the executable is stored in the instruction memory.

Compilers replace the LDR pseduo instruction with an actual load instruction with a PC-relative memory address to load the immediate number from the instruction memory. Chapter 5.4.3 introduces PC-relative addressing in details.

```
LDR r1, =2              ; Translated to:  MOV r1, #2
LDR r2, =-2             ; Translated to:  MVN r0, #1
LDR r3, =0x12345678     ; Translated to:  LDR r2, [pc, #offset1]
LDR r4, =myAddress      ; Translated to:  LDR r2, [pc, #offset2]
                        ; LDR with a PC-relative address
```

Example 5-3. Pseudo instruction LDR.

Note the syntax for specifying the constant number in MOV and LDR are different.

```
LDR r0, =0xFF           ; '=' before the constant
MOV r0, #0xFF           ; '#' before the constant
```

5.2 Big Endian and Little Endian

Cortex-M processors support both big endian and little endian. The endian specifies the byte order if a data element has multiple bytes, as shown in Figure 5-1.

- Little endian means the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. (The little end comes first.)
- Big endian means the high-order byte of the number is stored in memory at the lowest address, and the low-order byte at the highest address. (The big end comes first.)

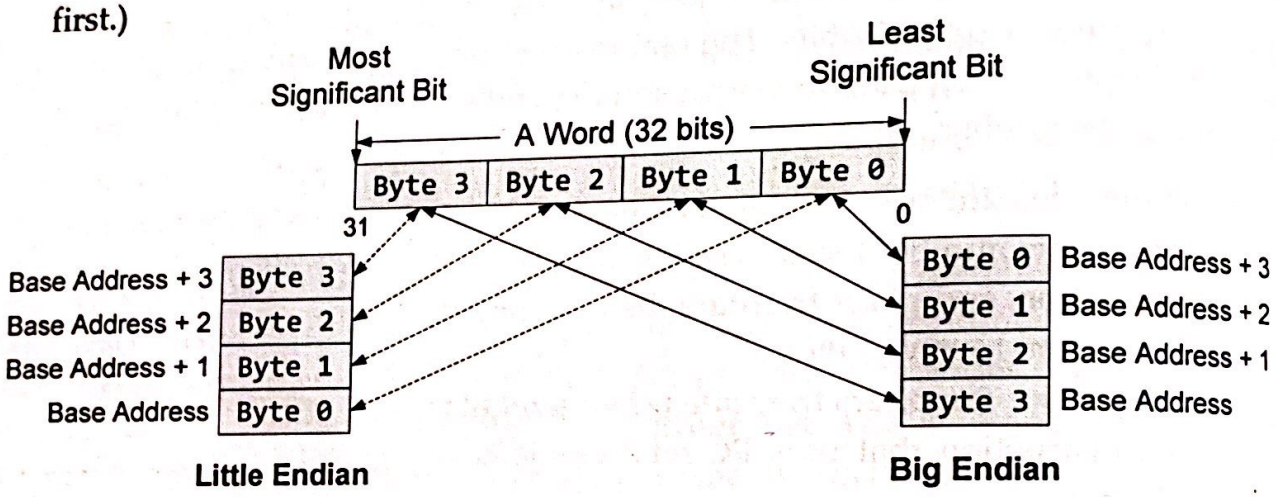


Figure 5-1. Comparison of little endian and big endian

In the example given in Figure 5-2, the assembly instruction "`LDR r1, [r0]`" loads a 32-bit value from the memory address `0x20008000` to register `r1`. Register `r1` has different results, depending on whether the big or little endian is used.

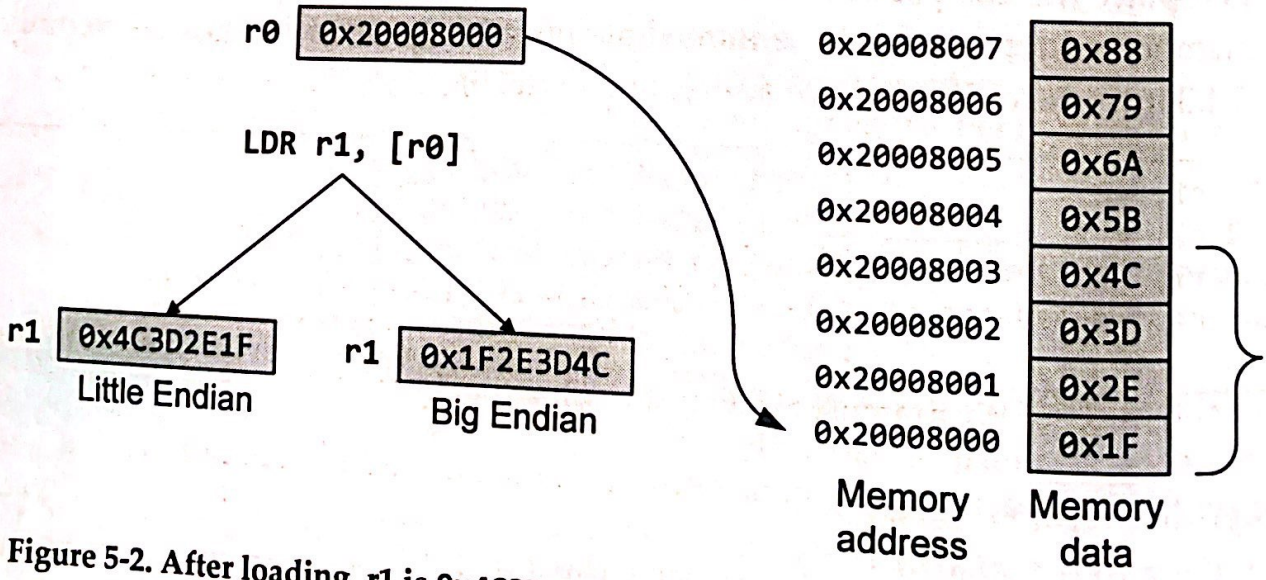


Figure 5-2. After loading, r1 is `0x4C3D2E1F` if little endian, and `0x1F2E3D4C` if big endian.

5.3 Accessing Data in Memory

A load instruction retrieves data stored at some memory address and saves the data in a specific register. A store instruction does the opposite. It saves the content of a register to the memory at a given memory address.

When an assembly program accesses data in the memory, the memory address must be in a register. Example 5-4 assumes the memory address is in register `r0`. The program loads a 32-bit integer to register `r1`, increases it by 4, and saves the result in the memory.

```
; Suppose r0 = 0x82000004
LDR r1, [r0]    ; r1 = a word (4 bytes) in memory starting at 0x82000004
ADD r1, r1, #4  ; r1 = r1 + 4
STR r1, [r0]    ; Save 4 bytes into memory starting at 0x82000004
```

Example 5-4. Loading a word from the memory

5.4 Memory Addressing

5.4.1 Pre-index, Post-index, and Pre-index with Update

Cortex-M processors support flexible memory addressing. They provide three memory address modes: pre-index, post-index, and pre-index with update, as shown in Table 5-2. Each mode has a base memory address (saved in a register) and a byte offset.

1. In the ***pre-index*** format, the target memory address is the base memory address plus the offset. The base memory address remains unchanged.
2. In the ***pre-index with update*** format, three steps are involved. First, it calculates the target memory address as the base plus the offset. Then, it accesses the data at the destination memory address. Finally, it updates the base memory.
3. In the ***post-index*** format, two steps are involved. First, it updates the base memory address as the sum of the base memory address and offset. Then it accesses the data by using the updated base memory address.

Memory Address Mode	Example	Equivalent
Pre-index	`LDR r1, [r0, #4]`	`r1 ← memory[r0 + 4],` `r0 remains unchanged.`
Pre-index with update	`LDR r1, [r0, #4]!`	`r1 ← memory[r0 + 4]` `r0 ← r0 + 4`
Post-index	`LDR r1, [r0], #4`	`r1 ← memory[r0]` `r0 ← r0 + 4`

Table 5-2. Three memory addressing formats

The following gives three examples to compare these three addressing modes. Suppose register r0 has an initial value of 0x20008000, and the processor uses little-endian.

```
LDR r1, [r0, #4]  ; Pre-index
```

As shown in Figure 5-3, register r0 remains unchanged. After loading the word stored at memory address 0x20008004, the data in register r1 is 0x88796A5B.

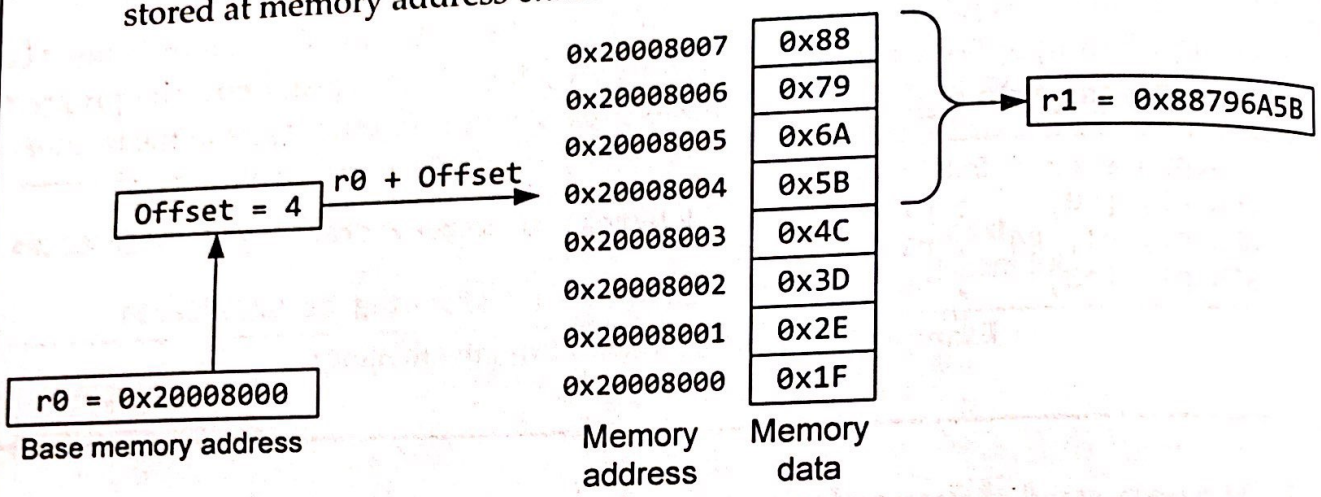


Figure 5-3. Pre-index (r1 ← memory[r0 + 4], r0 is unchanged)

```
LDR r1, [r0], #4  ; Post-index
```

As shown in Figure 5-4, the value in register r0 is incremented by the offset after loading. Register r1 is fetched from the memory address 0x20008000.

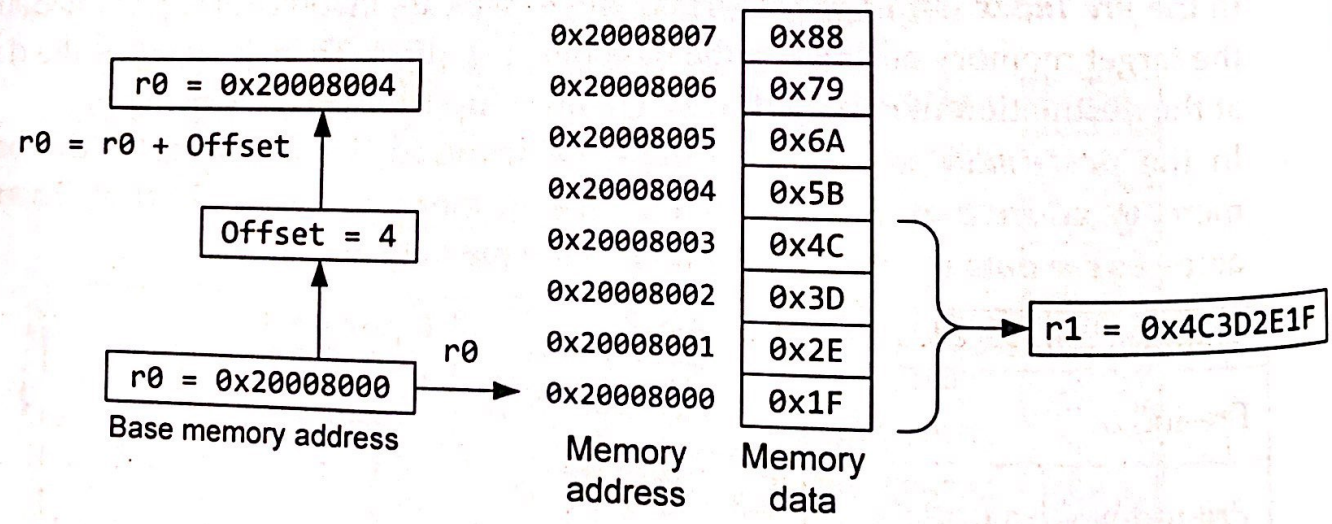


Figure 5-4. Post-index (r1 ← memory[r0], r0 ← r0 + 4)

```
LDR r1, [r0, #4]!   ; Pre-index with update
```

As shown in Figure 5-5, the value in register r0 is incremented by the offset after loading. Both the post-index and the pre-index with update change the base memory address. However, different with the post-index, the pre-index with update retrieves the word from the memory address 0x20008004, instead of 0x20008000.

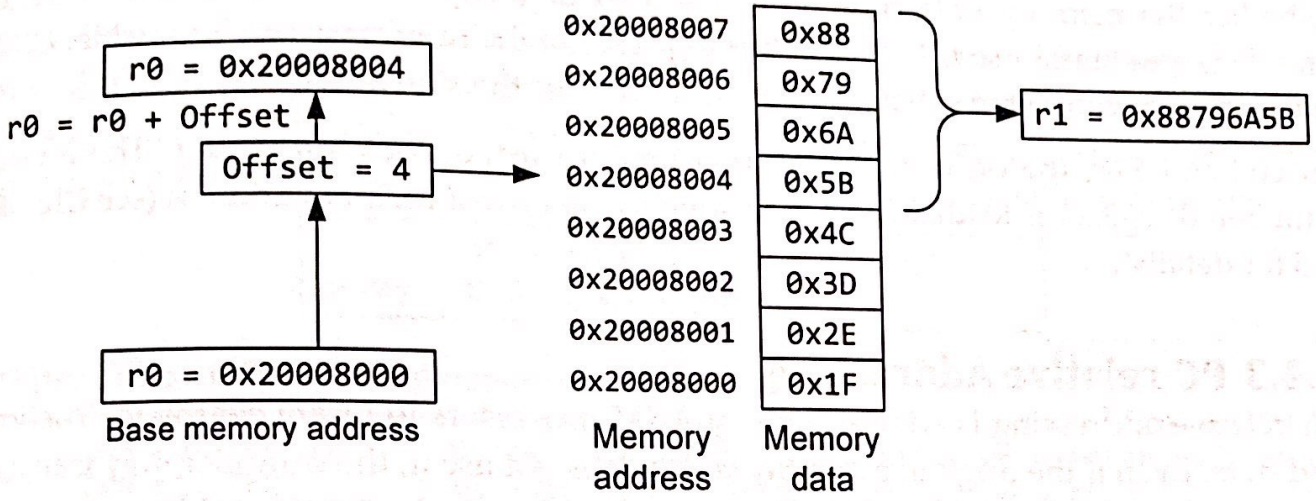


Figure 5-5. Pre-index with update (r1 ← memory[r0 + 4], r0 ← r0 + 4)

Table 5-3 summarizes the results of three addressing modes described above.

Instruction	Result of r0	Result of r1	Comment
LDR r1, [r0, #4]	0x20008000	0x88796A5B	Pre-index
LDR r1, [r0], #4	0x20008004	0x4C3D2E1F	Post-index
LDR r1, [r0, #4]!	0x20008004	0x88796A5B	Pre-index with update

Table 5-3. Example of three addressing modes

5.4.2 Load and Store Instructions

Table 5-4 and Table 5-5 list load and store instructions. Only the pre-index format is presented. However, load and store instructions with the other two formats are similar.

LDR Rt, [Rn, #offset]	Load word, *Rt ← mem[Rn + offset]*
LDRB Rt, [Rn, #offset]	Load byte, *Rt ← mem[Rn + offset]*
LDRH Rt, [Rn, #offset]	Load halfword, *Rt ← mem[Rn + offset]*
LDRSB Rt, [Rn, #offset]	Load signed byte, *Rt ← Sign Extend (mem[Rn + offset])*
LDRSH Rt, [Rn, #offset]	Load signed halfword, *Rt ← Sign Extend (mem[Rn + offset])*
LDM Rn, register_list	Load multiple words

Table 5-4. Load data of different sizes from memory to a register

`STR Rt, [Rn, #offset]`	Store word, *mem[Rn + offset] ← Rt*
`STRB Rt, [Rn, #offset]`	Store lower byte, *mem[Rn + offset] ← Rt*
`STRH Rt, [Rn, #offset]`	Store lower halfword, *mem[Rn + offset] ← Rt*
`STM Rn, register_list`	Store multiple words

Table 5-5. Store value of a register in memory

When a byte or halfword is loaded into a 32-bit register, we should draw attention to whether the memory data represents a signed or unsigned number. If it is a signed number, we should use `LDRSB` or `LDRSH` to preserve the number's sign and value. `LDRSB` and `LDRSH` perform sign extension, which duplicates the sign bit.

In `LDM` and `STM`, the order in which registers are listed does not matter. The lowest-numbered register is loaded from or written to the lowest memory address (see Chapter 5.5 for details).

5.4.3 PC-relative Addressing

PC-relative addressing is widely used by ARM processors to locate nearby instructions and data. Even if the original assembly codes does not use it, the compiler may translate a memory index by using PC-relative addressing in order to achieve position-independent addressing.

The target memory address is as follows:

$$Target\ Memory\ Address = PC + 4 + Offset$$

The program counter (`PC`) always incremented by 4, making it point to the next 32-bit instruction or the next two 16-bit instructions. Even if a 16-bit Thumb assembly instruction reads `PC`, the value returned is the address of this instruction plus 4 bytes.

PC-relative addressing is often used to set a register to a complicated value. For example, the program needs to set register `r1` to `0xF1234567`. We cannot use the instruction "`MOV r1, #0xF1234567`" because the constant number is too large. As an alternative, we use the pseudo instruction "`LDR r1, =0xF1234567`".

The compiler translates the above `LDR` pseudo instruction into a PC-relative `LDR` instruction.

- Suppose the constant `0xF1234567` is stored at the memory location `0x08000144`.
- The compiler uses the PC-relative addressing for the load word instruction. If the memory address of the load word (`LDR`) instruction is `0x0800012C`, the difference between `0x08000144` and `0x0800012C` is 24 in decimal.
- As a result, the memory address is `[pc, #20]`. The target address is `pc` + 4 + 20.

The following shows the translated PC-relative load instruction.

```
0x0800012C  LDR  r1, [pc, #20]  ; @0x08000144
...         ...
0x08000144  DCW  0x4567   ; lower halfword
0x08000146  DCW  0xF123   ; upper halfword
```

Example 5-5. Using PC-relative addressing to load a large constant number into a register

5.4.4 Example of Accessing an Array

The following three examples iterates through an array of five 32-bit integers by using three different addressing modes. Suppose we want to load an array of five integers into registers r1, r2, r3, r4, and r5. The following uses three different address modes to access the array and calculate the sum of the array. Assume the array is defined as follows.

```
        AREA myData, DATA, READWRITE
array   DCD 1, 2, 3, 4, 5
```

(1) Iterate an array by using pre-index

```
LDR r0, =array    ; Using LDR pseudo instruction, r0 = array address
LDR r1, [r0]      ; r1 = array[0]. After loading, r0 = array
LDR r2, [r0, #4]  ; r2 = array[1]. After loading, r0 = array + 4
LDR r3, [r0, #8]  ; r3 = array[2]. After loading, r0 = array + 8
LDR r4, [r0, #12] ; r4 = array[3]. After loading, r0 = array + 12
LDR r5, [r0, #16] ; r5 = array[4]. After loading, r0 = array + 16
```

(2) Iterate an array by using post-index

```
LDR r0, =array    ; Using LDR pseudo instruction, r0 = array address
LDR r1, [r0], #4  ; r1 = array[0]. After loading, r0 = array + 4
LDR r2, [r0], #4  ; r2 = array[1]. After loading, r0 = array + 8
LDR r3, [r0], #4  ; r3 = array[2]. After loading, r0 = array + 12
LDR r4, [r0], #4  ; r4 = array[3]. After loading, r0 = array + 16
LDR r5, [r0], #4  ; r5 = array[4]. After loading, r0 = array + 20
```

(3) Iterate an array by using pre-index with update

```
LDR r0, =array    ; Using LDR pseudo instruction, r0 = array address
LDR r1, [r0]      ; r1 = array[0]. After loading, r0 = array
LDR r2, [r0, #4]! ; r2 = array[1]. After loading, r0 = array + 4
LDR r3, [r0, #4]! ; r3 = array[2]. After loading, r0 = array + 8
LDR r4, [r0, #4]! ; r4 = array[3]. After loading, r0 = array + 12
LDR r5, [r0, #4]! ; r5 = array[4]. After loading, r0 = array + 16
```

The above example codes only work well for a short array. If the length of the array is long, then the assembly program needs to use conditional branch instructions (see Chapter 6) to implement a loop in order to iterate the array.

5.5 Loading and Storing Multiple Registers

A sequence of registers can be stored in consecutive memory locations in one assembly instruction. Similarly, multiple words can be loaded from sequential memory locations to registers in one instruction too.

There are four different addressing modes for loading and storing multiple registers, as shown in Table 5-6.

Addressing Modes	Description	Instructions
IA	Increment After	STMIA, LDMIA
IB	Increment Before	STMIB, LDMIB
DA	Decrement After	STMDA, LDMDA
DB	Decrement Before	STMDB, LDMDB

Table 5-6. Four different addressing modes for STM and LDM

- IA: The memory address is incremented by 4 after a word is loaded or stored.
- IB: The memory address is incremented by 4 before a word is loaded or stored.
- DA: The memory address is decremented by 4 after a word is loaded or stored.
- DB: The memory address is decremented by 4 before a word is loaded or stored.

The assembly instruction format is as follows:

```
STMxx rn{!}, {register_list}
LDMxx rn{!}, {register_list}
```

where the base register rn holds the starting memory location and xx is one of the addressing modes (IA, IB, DA, or DB).

- The exclamation mark ! is optional. If it is specified, the instruction writes a modified value back to rn. If it is omitted, register rn is not updated.
- The order in which registers are listed in the register list does not matter at all. When multiple registers are stored or loaded, they are sorted by name, and the lowest-numbered register is stored to or loaded from the lowest memory address.

Figure 5-6 shows the result of "STMxx r0!, {r3,r1,r7,r2}" under four different address modes (where xx = IA, IB, DA, or DB). Note the order in which the four registers are listed does not matter. Register r1, the lowest numbered register, is always stored at the lowest memory address in all memory address modes. For STMIA and STMDA, the base register r0 points to an empty memory location at the end, while it points to a valid data item for STMIB and STMDB.

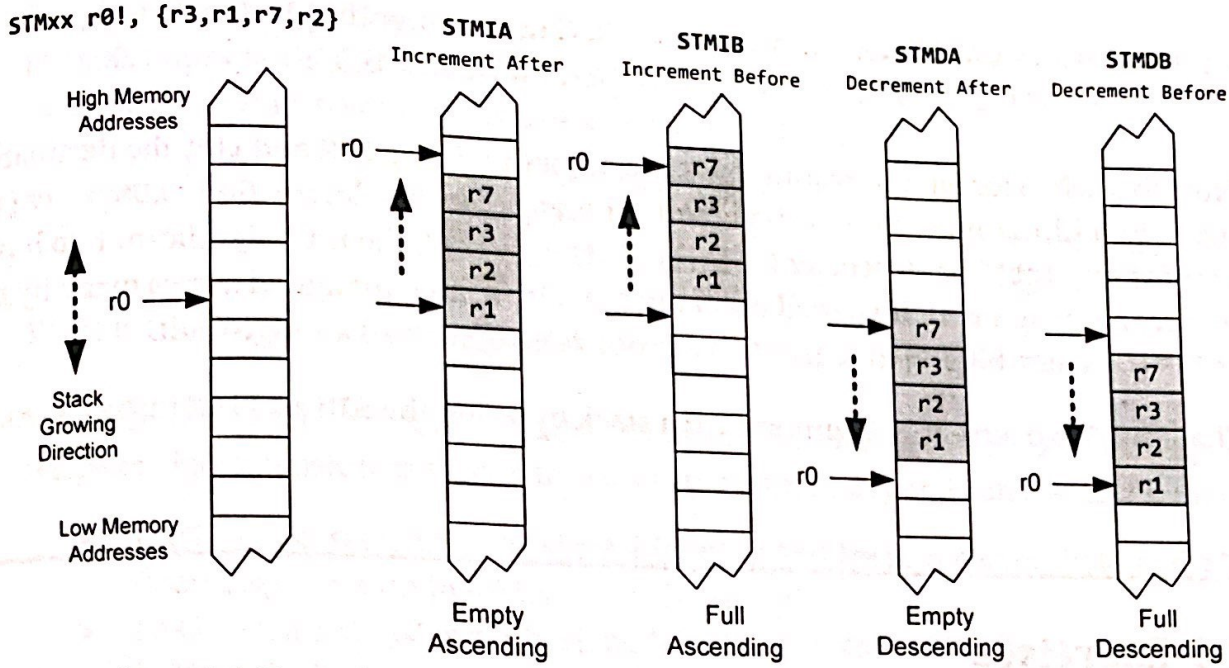


Figure 5-6. Example of four different memory addressing mode for STM.

Figure 5-7 shows the result of "`LDMxx r0!, {r3,r1,r7,r2}`" under four different address modes (where `xx` = `IA`, `IB`, `DA`, or `DB`). Similar to `STM`, the register order listed in an `LDM` instruction does not matter. The lowest-numbered register is loaded from the lowest memory address.

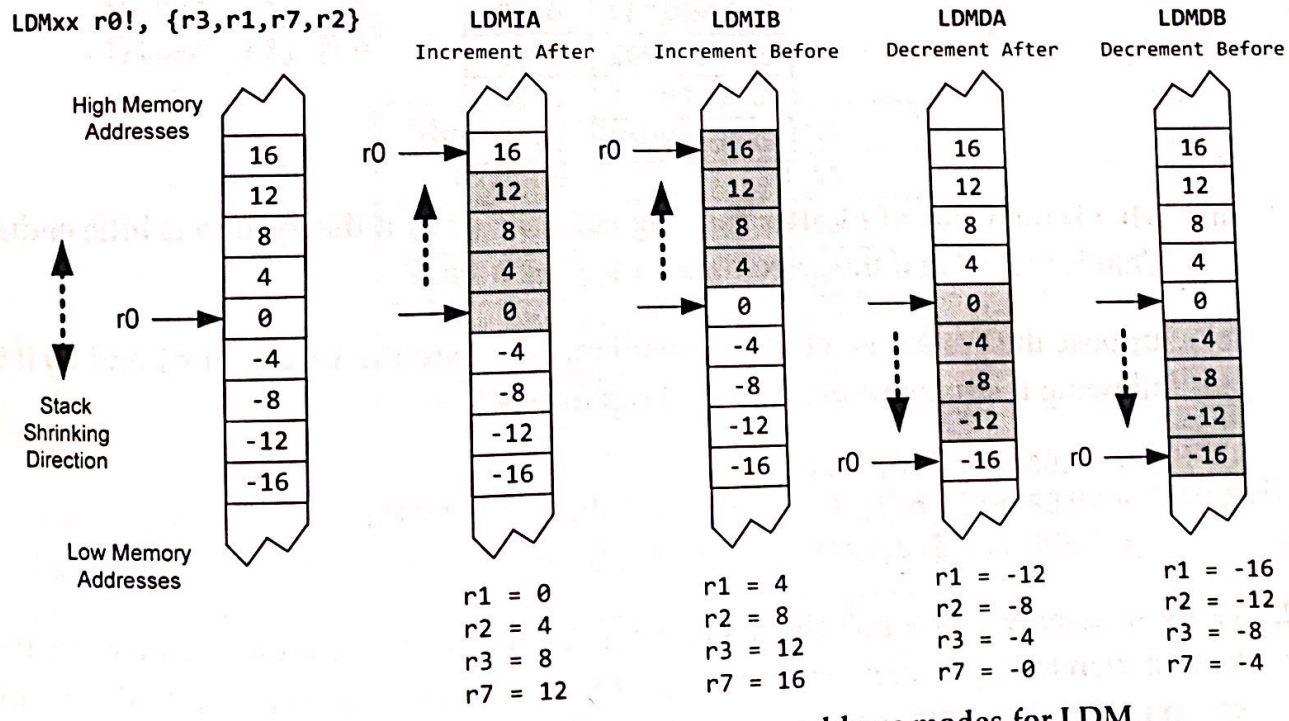


Figure 5-7. Example of four different memory address modes for LDM.

For Cortex-M processors, `STM` is `STMIA`, and `LDM` is `LDMIA`. The following are synonyms.

- STM = STMIA (Increment After) = STMEA (Empty Ascending)
- LDM = LDMIA (Increment After) = LDMFD (Full Descending)

Note that when loading or storing multiple values by using STM and LDM, the destination memory address must be word-aligned. Therefore, in the instruction "LDMxx rn{!}, {register_list}" or "STMxx rn{!}, {register_list}", the least significant two bits of register rn are ignored. If aligned checking is enabled, any unaligned access made by LDM or STM (*i.e.* when bit[1:0] in register rn are not zero) generates a usage fault.

Chapter 8.3 explains how to implement a stack by using the STM and LDM instructions.

5.6 Exercises

1. Suppose r0 = 0x20008000, and the memory layout is as follows:

Address	Data
0x20008007	0x79
0x20008006	0xCD
0x20008005	0xA3
0x20008004	0xFD
0x20008003	0x0D
0x20008002	0xEB
0x20008001	0x2C
0x20008000	0x1A

   a) What is the value of r1 after running LDR r1, [r0] if the system is little endian? What is the value if the system uses the big-endian?

   b) Suppose the system is set as little endian. What are the values of r1 and r0 if the following instructions are executed separately?

   - LDR r1, [r0, #4]
   - LDR r1, [r0], #4
   - LDR r1, [r0, #4]!

2. Write an assembly program that converts a 32-bit integer stored in the memory from little endian to big endian, without using the REV instruction. Make sure the result is saved back to the memory.

3. Suppose `r0 = 0x20000000` and `r1 = 0x12345678`. All bytes in memory are initialized to `0x00`. Suppose the following assembly program has run successfully. Draw a table to show the memory value if the processor uses little endian.

```
STR r1, [r0], #4
STR r1, [r0, #4]!
STR r1, [r0, #4]
```

4. What is the memory value of Question 3 if the processor uses big endian?

5. When an 8-bit or 16-bit data is loaded from the data memory into a 32-bit register, whether sign extension or zero extension is performed depends on the data's sign.

   - `LDRSB` (load register with signed byte) `LDRSH` loads a signed byte and `LDRB` (load register with byte) for an unsigned byte.
   - `LDRSH` (load register with signed halfword) and `LDRH` (load register with halfword) load a 16-bit signed and unsigned number into a register, respectively.

   What is the value in register `r1` in the following instructions if `r0 = 0x20008000`? Assume the system is little endian.

```
(1) LDRSB r1, [r0]
(2) LDRSH r1, [r0]
(3) LDRB  r1, [r0]
(4) LDRH  r1, [r0]
```

Memory address	Data
0x20008002	0xA1
0x20008001	0xB2
0x20008000	0xC3
0x20007FFF	0xD4
0x20007FFE	0xE5

6. Suppose `r0 = 0x20008000`. What address is register `r7` loaded from in the following instructions? What is the value of `r0` after executing each instruction? Assume each instruction runs separately, *i.e.* they are not part of a program.

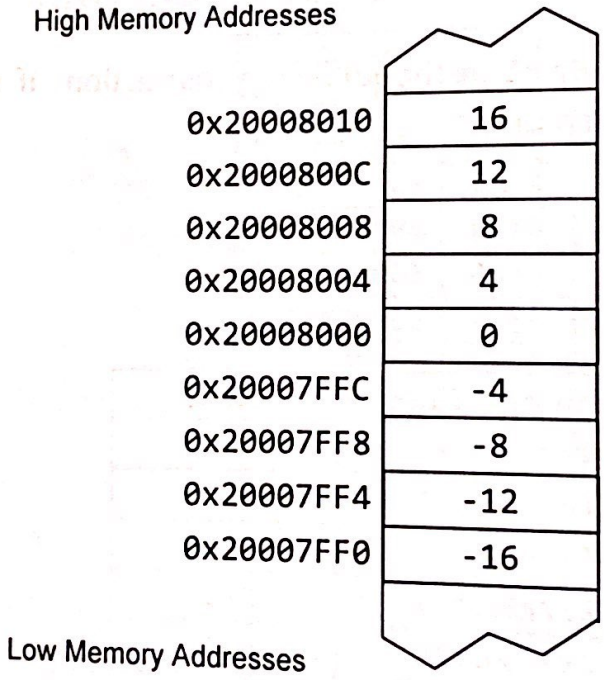
```
(1) LDMIA r0, {r1, r3, r7, r6, r2}
(2) LDMIB r0, {r1, r3, r7, r6, r2}
(3) LDMDA r0, {r1, r3, r7, r6, r2}
(4) LDMDB r0, {r1, r3, r7, r6, r2}
```

7. Suppose `r0 = 0x20008000`. What address is register `r7` stored at in the following instructions? What is the value of `r0` after executing each instruction? Assume each instruction runs separately, *i.e.* they are not part of a program.

```
(1) STMIA r0!, {r3, r9, r7, r1, r2}
(2) STMIB r0!, {r3, r9, r7, r1, r2}
(3) STMDA r0!, {r3, r9, r7, r1, r2}
(4) STMDB r0!, {r3, r9, r7, r1, r2}
```

8. Suppose `r0 = 0x20008000`. What is the value in register `r0`, `r3`, `r5`, `r7`, and `r9` after running the following instructions? Assume each instruction runs separately, *i.e.* they are not part of a program.

```
(1) LDMDB r0, {r3, r7, r9, r5}
(2) LDMIA r0, {r7, r3, r9, r5}
(3) LDRIB r0, {r3, r9, r5, r7}
(4) LDRDA r0, {r9, r5, r7, r3}
```



9. In the following load instruction based on PC-relative addressing, what is the address range in which the target data can be located? Assume the memory address of this instruction is `0x10004000`. The PC-relative offset is a 12-bit signed integer.

```
LDR r1, =label
```