CHAPTER 5

Load and Store

A load instruction sets a register to some value. The value might be a constant directly specified in the program or a value that is stored in 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 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 an 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 pseudo 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 detail.

```
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 and Little Endian

Cortex-M processors support both big and little endian. The endian specifies the byte order if a data element has multiple bytes, as shown in Figure 5-1. We can use REV, which reserves the byte order, to convert the endian (See Chapter 4.7).

- 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 at the lowest address, and the low-order byte at the highest address. (The big end comes first.)

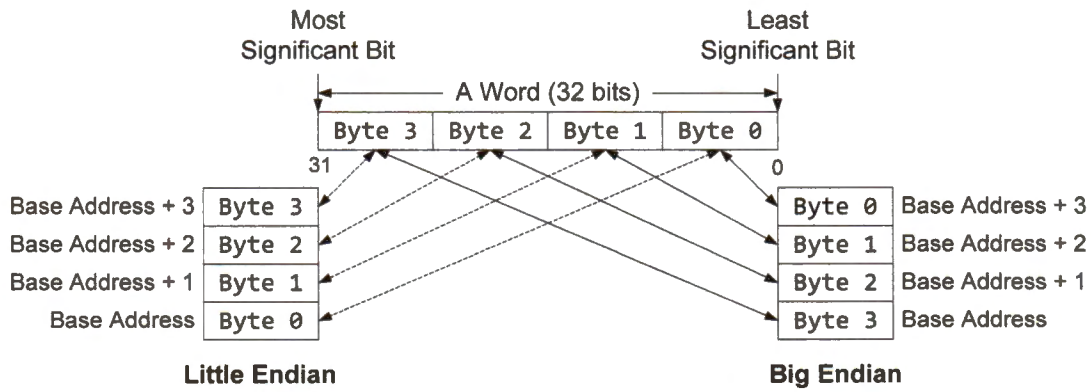


Figure 5-1. Comparison of little endian and big endian

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.

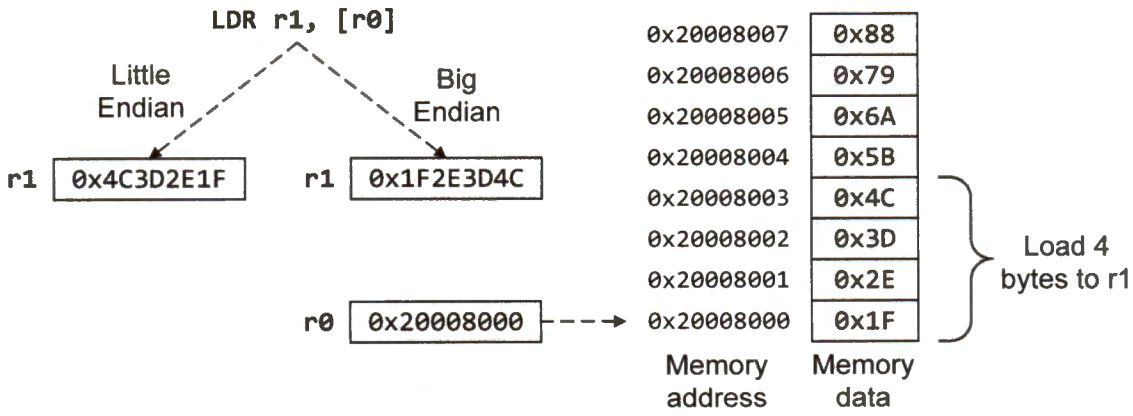


Figure 5-2. Register `r1` is `0x4C3D2E1F` under little endian, and `0x1F2E3D4C` under big endian.

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 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 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 addressing 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.

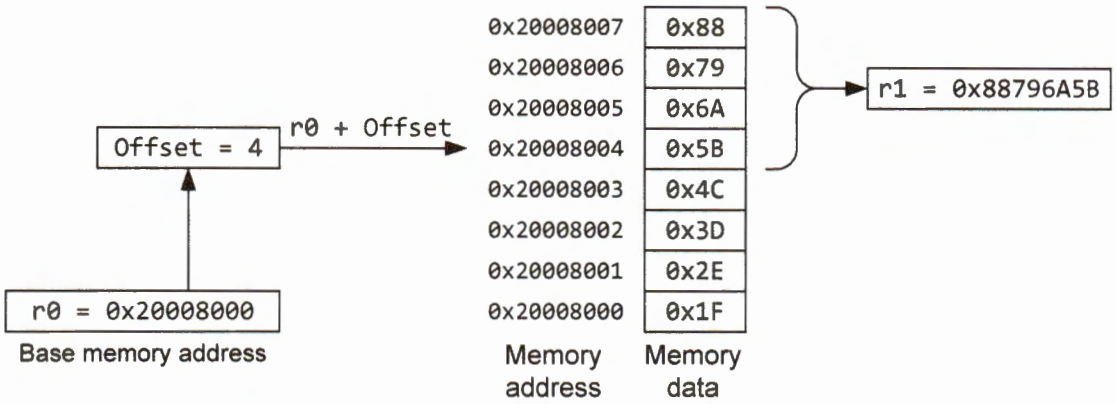


Figure 5-3. Pre-index (r1 ← memory[r0 + 4], r0 remains unchanged)

```
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.

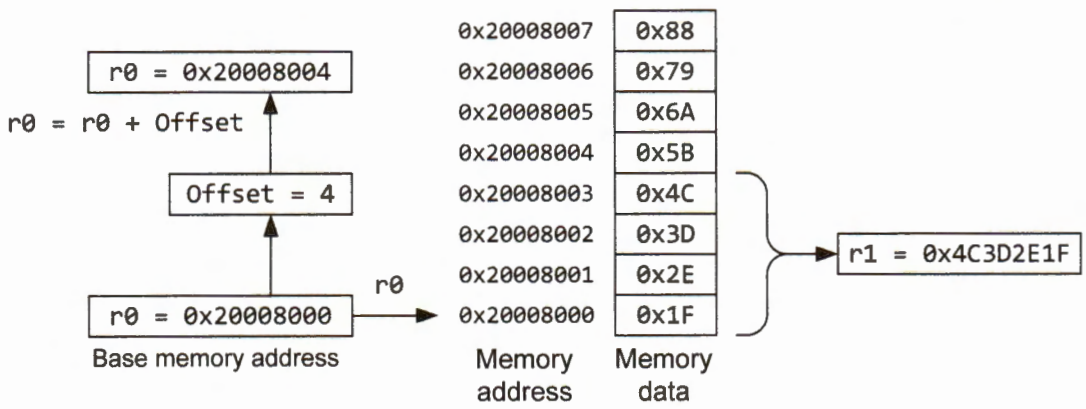


Figure 5-4. Post-index (r1 ← memory[r0], r0 ← r0 + 4)

```
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.

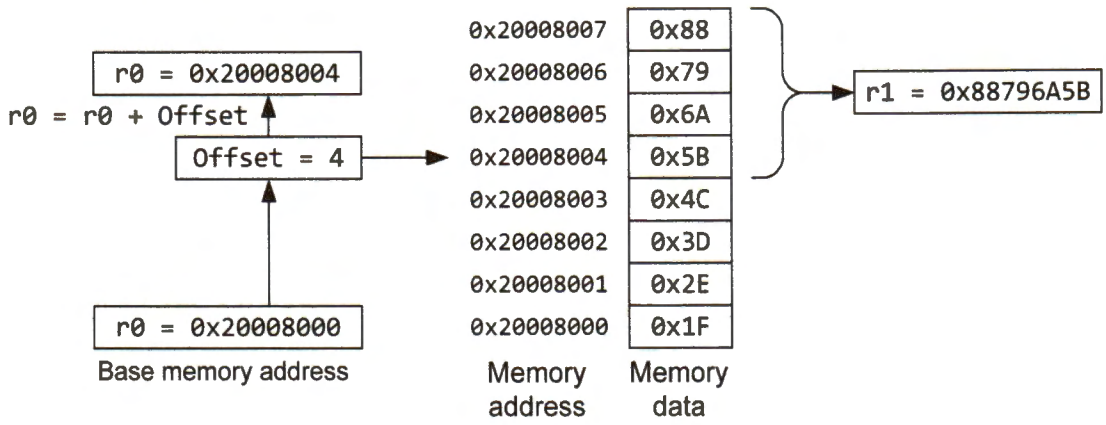


Figure 5-5. Pre-index with update (r1 ← memory[r0 + 4], r0 ← r0 + 4)

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 program does not use it, the compiler may translate a memory index by using `PC`-relative addressing 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, pointing 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.
- Thus, 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 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 Mode	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 register 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 saved to or read 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.

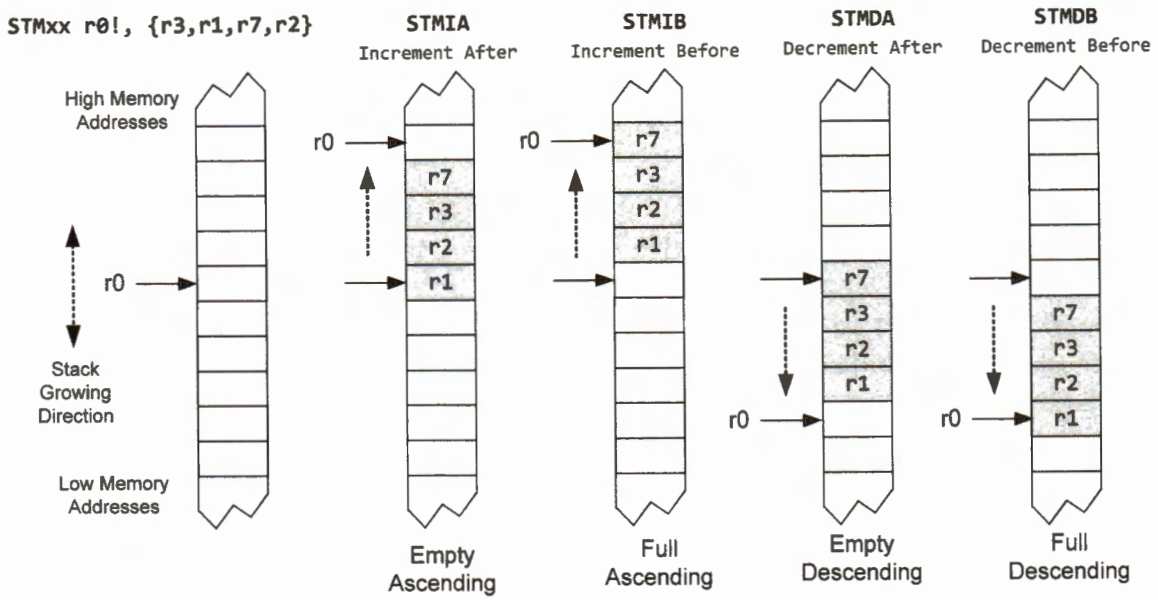


Figure 5-6. Example of four different memory addressing mode for STM.

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`). Like `STM`, the order in which registers are listed does not matter in an `LDM` instruction. The lowest-numbered register is loaded from the lowest memory address.

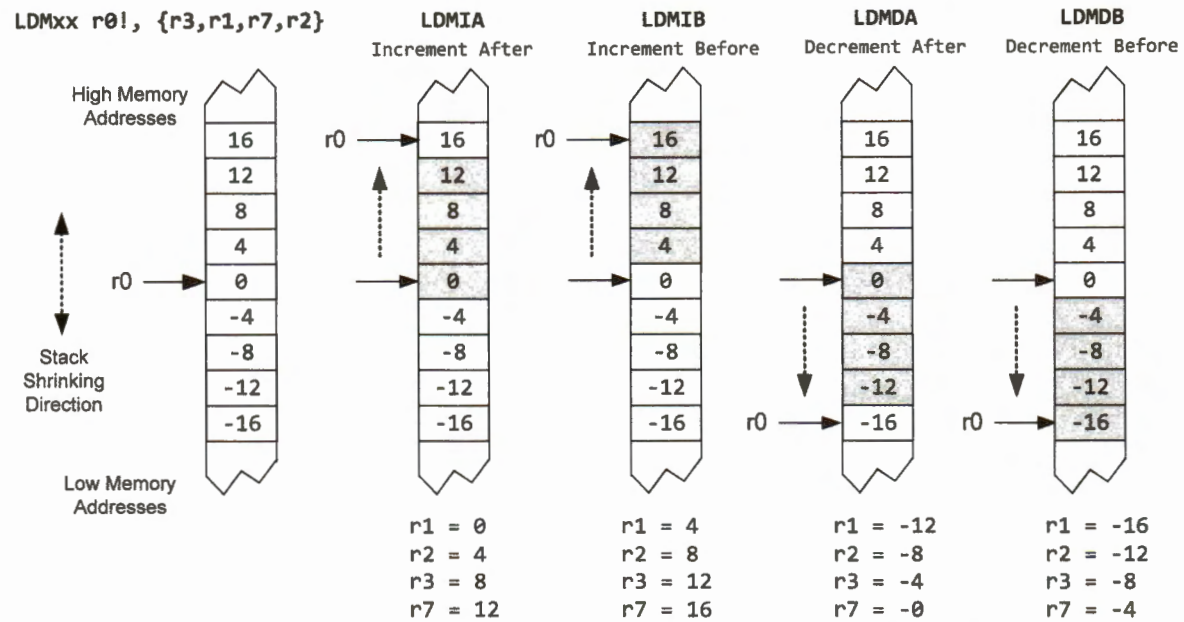


Figure 5-7. Example of four different memory address modes for LDM.

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 memory from little endian to big endian, without using the REV instruction. Make sure that 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) read load a 16-bit signed and unsigned number from memory 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.

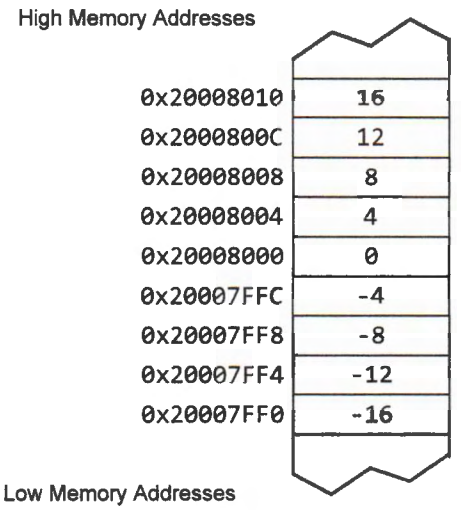
```
(1) LDMIA r0, {r1, r3, r7, r6, r2}
(2) LDMIB r0, {r1, r3, r7, r6, r2}
(3) LDMDA r0, {r1, r3, r7, r6, r2}
(4) LDMDB r0, {r1, r3, r7, r6, r2}
```

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
```

CHAPTER 6

Branch and Conditional Execution

Normally instructions of an assembly program run in the same sequential order as they are listed in the program. When one instruction completes, the program counter is incremented by the control unit within the processor and ordinarily points to the next instruction. However, modifying the program counter at runtime can dynamically change the execution order. We call it changing the flow of control. There are four major approaches to alter the flow of control:

1. branch instructions,
2. conditional execution,
3. calling a subroutine, and
4. interrupts.

This chapter focuses on the first two approaches. Chapter 8 discusses subroutines, and Chapter 11 presents interrupt.

6.1 Condition Testing

Most assembly instructions can be selectively executed based on the N, Z, C, and V flags of the application program status register (APSR). Table 6-1 lists the condition flags for comparing signed and unsigned numbers.

Compare	Signed	Unsigned	Relationship Tested
==	EQ	EQ	Equal to
!=	NE	NE	Not equal to
>	GT	HI	Greater than
≥	GE	HS	Greater than or equal to
<	LT	LO	Less than
≤	LE	LS	Less than or equal to

Table 6-1. Summary of the comparison suffix for signed and unsigned numbers

These flags provide convenience for programmers and improve the code readability.

For example, the following two assembly instructions calculate the absolute value of a signed integer stored in register r1. The second instruction RSB is executed if r1 is less than 0. The condition flag "LT" tests the negative flag, and the processor ignores the RSB instruction if the negative flag is 0.

```
CMP   r1, #0        ; CMP updates N, Z, C, and V flags
RSBLT r1, r1, #0    ; Run r1 = 0 - r1 if r1 < 0. LT = signed Less Than.
```

Cortex-M processors have 15 condition flags, as summarized in Table 6-2. These condition flags check whether N, Z, C, and V meet specific requirements. When an instruction has no conditional flag, it defaults to "AL" and is always executed.

Suffix	Description	Flags tested	Logic Implementation
EQ	EQual	Z = 1	Z
NE	Not Equal	Z = 0	$\bar{Z}$
CS/HS	unsigned Higher or Same	C = 1	C
CC/LO	unsigned LOwer	C = 0	$\bar{C}$
MI	MInus (negative)	N = 1	N
PL	PLus (positive or zero)	N = 0	$\bar{N}$
VS	oVerflow Set	V = 1	V
VC	oVerflow Clear	V = 0	$\bar{V}$
HI	unsigned HIgher	C = 1 & Z = 0	$C\bar{Z}$
LS	unsigned Lower or Same	C = 0 or Z = 1	$\bar{C} + Z$
GE	signed Greater or Equal	N = V	$NV + \bar{N}\bar{V}$
LT	signed Less Than	N != V	$N\bar{V} + \bar{N}V$
GT	signed Greater Than	Z = 0 & N = V	$\bar{Z}(NV + \bar{N}\bar{V})$
LE	signed Less than or Equal	Z = 1 or N ! = V	$Z + N\bar{V} + \bar{N}V$
AL	ALways		

Table 6-2. Summary of flag testing for various signed and unsigned comparisons

The CMP instruction "CMP r0, r1" is equivalent to the subtraction operation r0 – r1, except the result is discarded.

When two registers in the instruction "CMP r0, r1" represent unsigned integers,

- the carry flag is set if no borrow occurs during the subtraction (*i.e.*, r0 ≥ r1), and
- the carry flag is cleared if borrow does occur during the subtraction (*i.e.*, r0 < r1).

Therefore, the HS, LO, HI and LS suffix check the zero flag (if necessary) and the carry flag.

When two registers in the instruction "`CMP r0, r1`" represent signed numbers, Table 6-3 summarizes the meaning of all four possible combinations of the negative flag (`N`) and the overflow flag (`V`).

	N = 0	N = 1
V = 0	r0 ≥ r1	r0 < r1
V = 1	r0 < r1	r0 ≥ r1

Table 6-3. The meaning of the overflow and negative flags of "`CMP r0, r1`" if register `r0` and `r1` hold signed numbers.

Table 6-4 gives the detailed explanation of how to get the conclusions listed in Table 6-3.

- When two signed numbers are subtracted, there are two possible scenarios, in which overflow occurs: (1) the result of subtracting a positive number from a negative number is positive, or (2) the result of subtracting a negative number from a positive number is negative.
- When subtracting two numbers with the same sign, no overflow would occur.

In sum, if overflow occurs, the result is incorrect, and its sign indicated by the `N` flag is opposite to the sign of the actual result.

	N = 0	N = 1
V = 0	No overflow has occurred, implying the result is correct. The result is non-negative. Thus, r0 – r1 ≥ 0, *i.e.*, r0 ≥ r1.	No overflow has occurred, implying the result is correct. The result is negative. Thus, r0 – r1 < 0, *i.e.*, r0 < r1.
V = 1	Overflow has occurred, implying the result is incorrect. The result is mistakenly reported as non-negative, but it should be negative. Thus, r0 – r1 < 0 in reality, *i.e.*, r0 < r1.	Overflow has occurred, implying the result is incorrect. The result is mistakenly reported as negative, but it should be non-negative. Thus, r0 – r1 ≥ 0 in reality, *i.e.*, r0 ≥ r1.

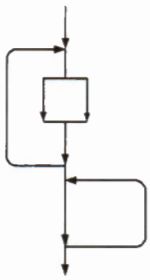
Table 6-4. The signed greater or equal (GE) checks whether `V` is equal to `N`.

Table 6-4 leads to the following conclusions:

- If `N` = `V`, then `r0` is signed greater than or equal to `r1`.
- If `N` ≠ `V`, then `r0` is signed less than `r1`.

Therefore, the signed greater or equal suffix (`GE`) and the signed greater than suffix (`GT`) check whether the `N` flag is the same as the `V` flag.

6.2 Branch Instructions



A branch instruction is used to change the flow of program execution from a normal sequential order. It allows the microprocessor to begin execution a different set of instructions. There are two types of branch instructions: unconditional and conditional.

- An ***unconditional*** branch instruction always loads the memory address of the designated instruction into the program counter and starts to execute the new program flow. Assembly programs use a label to denote the designated instruction.
- A ***conditional*** branch instruction first checks whether a specific condition is satisfied or not. If the condition is satisfied, the processor then starts to execute the designated instruction, instead of the next sequential instruction. A conditional branch instruction is equivalent to "if the *condition* is true, then go to the *label*." When the program jumps away, we say the branch is taken. Otherwise, the branch is not taken.

We can append the condition suffix to the branch instruction "`B`" to form different conditional branch instructions, as summarized in Table 6-5. For example, "`BEQ`" compares two register values, and the branch is taken if the source operands are equal.

	Instruction	Description	Flags tested
Unconditional Branch	`B label`	Branch to label	none
Conditional Branch	`BEQ label`	Branch if **EQ**ual	`Z = 1`
	`BNE label`	Branch if **N**ot **E**qual	`Z = 0`
	`BCS/BHS label`	Branch if unsigned **H**igher or **S**ame	`C = 1`
	`BCC/BLO label`	Branch if unsigned **LO**wer	`C = 0`
	`BMI label`	Branch if **MI**nus (Negative)	`N = 1`
	`BPL label`	Branch if **PL**us (Positive or Zero)	`N = 0`
	`BVS label`	Branch if o**V**erflow **S**et	`V = 1`
	`BVC label`	Branch if o**V**erflow **C**lear	`V = 0`
	`BHI label`	Branch if unsigned **HI**gher	`C = 1 & Z = 0`
	`BLS label`	Branch if unsigned **L**ower or **S**ame	`C = 0 or Z = 1`
	`BGE label`	Branch if signed **G**reater or **E**qual	`N = V`
	`BLT label`	Branch if signed **L**ess **T**han	`N != V`
	`BGT label`	Branch if signed **G**reater **T**han	`Z = 0 & N = V`
	`BLE label`	Branch if signed **L**ess than or **E**qual	`Z = 1 or N = !V`

Table 6-5. List of unconditional and conditional branch instructions

Note some ARM processors can directly support all branch instructions listed above. However, Cortex-M processors do not directly support these conditional branch instructions. Instead, compilers translate conditional branch instructions to `if-then-else` (`IT`) instructions. `IT` performs the same flag testing as presented in Table 6-5.

Program flow control structures such as *if-then*, *if-then-else*, *for* loop, and *while* loop use (1) `CMP` instructions followed by a branch instruction, (2) conditionally executed instructions (see Chapter 6.3), or (3) a combination of both. Table 6-6 summarizes conditional branch instructions for the comparison of signed and unsigned numbers.

Comparison	Signed	Unsigned
==	BEQ	BEQ
!=	BNE	BNE
>	BGT	BHI
≥	BGE	BHS
<	BLT	BLO
≤	BLE	BLS

Table 6-6. Comparison of branch instructions used for signed and unsigned comparison

Example: Go to the labeled instruction if two numbers are equal.

```
CMP r1, r2
BEQ label
```

When comparing `0xFFFFFFFF` or `0x00000001`, which is greater? When they are unsigned integers, the first number is larger. However, if they are signed numbers, the second one is larger. When the program is written in assembly, it is the programmer's responsibility to tell the processor how to interpret data. If written in C, their corresponding variables are declared explicitly by programmers as signed or unsigned numbers.

When two numbers are unsigned integers, branch instructions should use an unsigned condition suffix.

C Program	Assembly Program

`uint32_t x, y, z;`
`x = 0x00000001;`
`y = 0xFFFFFFFF;`
 	
`if (x > y)`
`  z = 1;`
`else`
`  z = 0;`
	`  MOV r5, #0x00000001 ; r5 = x`
	`  MOV r6, #0xFFFFFFFF ; r6 = y`
	`  CMP r5, r6`
	`  BLS else  ; branch if ≤`
	`then  MOV r7, #1  ; z = 1`
	`  B  endif  ; skip next instruction`
	`else  MOV r7, #0  ; z = 0`
	`endif`

Example 6-1. Implementation of *if*-statement that compares two unsigned integers

When these two numbers are signed integers, branch instructions should use a signed condition suffix.

C Program	Assembly Program

`int32_t x, y, z;`
 	
`x = 1; // 0x00000001`
`y = -1; // 0xFFFFFFFF`
 	
`if (x > y)`
`  z = 1;`
`else`
`  z = 0;`
	`MOVS r5, #0x00000001 ; r5 = x`
	`MOVS r6, #0xFFFFFFFF ; r6 = y`
	
	`CMP r5, r6`
	`BLE then ; branch if signed ≤`
	`MOVS r7, #1 ; z = 1`
	`B endif ; skip next instruction`
	`then MOVS r7, #0 ; z = 0`
	`endif`

Example 6-2. Implementation of *if*-statement that compares two signed integers

It is often that an assembly program compares against zero and checks whether the branch should be taken or not. Instructions `CBZ` (compare and branch on zero) and `CBNZ` (compare and branch on non-zero) are available to improve the performance of this common case by reducing one instruction.

One limitation is that `CBZ` and `CBNZ` can only branch forward, and the branch destination must be within 4 to 130 bytes after the instruction. The following shows example usages and their equivalent implement.

```
CBZ  r1, label      ⇒      CMP  r1, #0
                           BEQ  label  ; branch if equal
```

```
CBNZ r1, label      ⇒      CMP  r1, #0
                           BNE  label  ; branch if not equal
```

In addition to these standard branch instructions, the following are special branch instructions that call a subroutine. Chapter 8.1 gives detailed descriptions and examples.

- "`BL label`" instruction copies the memory address of the instruction immediately after the `BL` instruction into the link register (`r14`), and then branches to the instruction addressed by the label.
- "`BX Rm`" is like "`BL label`" except that the target instruction address is saved in register `Rm`.
- "`BLX Rm`" first places the address of the next instruction after the `BLX` instruction into the link register and then branches to the address held in `Rm`.

BL label	Branch with link. *LR = PC + 4; PC = label*
BLX Rm	Branch with link and exchange. *LR = PC + 4; PC = Rm*
BX Rm	Branch and exchange. *PC = Rm*

Table 6-7. Branch instructions that call a subroutine.

6.3 Conditional Execution

Besides four data comparison instructions (CMP, CMN, TEQ, and TST), most instructions can update the program status flags (N, Z, C, and V) if the suffix S is added. One of the salient features of ARM assembly language is that an instruction can be executed optionally based on the program status flags. This feature is often not available in other assembly languages.

The condition flags introduced in Chapter 6.1 can be a suffix of almost all instructions to implement conditional execution. The conditional branch instructions presented in the previous section are a special case of conditional execution.

We take the add instruction as an example to illustrate conditional execution. By default, the instruction "ADD r3, r2, r1" is always executed no matter what value the program status flags are. The conditional flag, such as EQ, can be appended to ADD to form a conditionally executed instruction ADDEQ, as shown in Table 6-8.

Add instruction	Condition	Flags tested
ADDEQ r3, r2, r1	Add if EQual	Add if Z = 1
ADDNE r3, r2, r1	Add if Not Equal	Add if Z = 0
ADDHS r3, r2, r1	Add if Unsigned Higher or Same	Add if C = 1
ADDLO r3, r2, r1	Add if Unsigned LOwer	Add if C = 0
ADDMI r3, r2, r1	Add if Minus (Negative)	Add if N = 1
ADDPL r3, r2, r1	Add if PLus (Positive or Zero)	Add if N = 0
ADDVS r3, r2, r1	Add if oVerflow Set	Add if V = 1
ADDVC r3, r2, r1	Add if oVerflow Clear	Add if V = 0
ADDHI r3, r2, r1	Add if Unsigned HIgher	Add if C = 1 & Z = 0
ADDLS r3, r2, r1	Add if Unsigned Lower or Same	Add if C = 0 or Z = 1
ADDGE r3, r2, r1	Add if Signed Greater or Equal	Add if N = V
ADDLT r3, r2, r1	Add if Signed Less Than	Add if N != V
ADDGT r3, r2, r1	Add if Signed Greater Than	Add if Z = 0 & N = V
ADDLE r3, r2, r1	Add if Signed Less than or Equal	Add if Z = 1 or N = !V

Table 6-8. Conditionally executed ADD instruction

Conditionally executed instructions can help facilitate the implementation of the selection and loop control structures. The following gives an example.

```
CMP    r1, r0        ; Perform r1 - r0 but discard the subtraction result
ADDSPL r3, r3, #1    ; Increment r3 by 1 and update flags if r1 ≥ r0
```

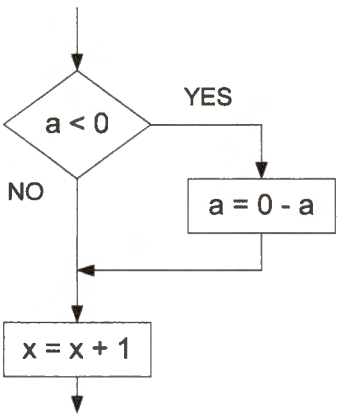
6.4 If-then Statement

An *if-then* statement in C selectively executes a block of code based on whether a given Boolean condition is true or false.

- If the condition is true or non-zero, the block is executed.
- Otherwise, the block is skipped, and the control returns to the first statement after the if-then statement.

The following example calculates the absolute value of a signed integer *a* and increases the variable *x* by 1.

```
C Program
if (a < 0 ) {
  a = 0 - a;
}
x = x + 1;
```



Assuming variable *a* and *x* are stored in register r1 and r2, respectively, the following gives two assembly implementations equivalent to the above C code.

An *if-then* statement can be implemented by using a conditional branch instruction (Example 6-3) or a conditionally executed instruction (Example 6-4).

```
        ; r1 = a, r2 = x
        CMP r1, #0          ; Compare a with 0
        BGE endif           ; Go to endif if a ≥ 0
then    RSB r1, r1, #0      ; a = - a
endif   ADD r2, r2, #1      ; x = x + 1
```

Example 6-3. An *if-then* statement can be implemented by using a conditional branch.

```
        ; r1 = a, r2 = x
        CMP   r1, #0        ; Compare a with 0
        RSBLT r1, r1, #0    ; a = 0 - a if a < 0
        ADD   r2, r2, #1    ; x = x + 1
```

Example 6-4. An *if-then* statement can be carried out by using conditional execution.

Compared with conditional branch instructions, conditionally executed instructions are concise and provide convenience for programmers. However, we should use conditionally executed instructions only when the if-statement body is short. What's more, in a nested-*if* statement, conditional branches are often preferred.

Compound Boolean expression

In mathematics, a Boolean (or logical) condition can be a compound expression combined with logical operators AND, OR, and NOT.

C language uses three special symbols as logical operators: && for Boolean AND, || for Boolean OR, and ! for Boolean NOT. In C, we write a Boolean condition like this:

```
x > 20 && x < 25
x == 20 || x == 25
!(x == 20 || x == 25)
```

The NOT operator (!) has higher precedence than the AND operator (&&), which has higher precedence than the OR operator (||).

If-then statement with a compound logical OR expression

A compound logical expression combined by logical OR can be implemented by multiple comparison instructions that test each simple logical expression. Example 6-5 shows how to implement an *if*-statement with a compound logic OR expression.

C Program	Assembly Program

`// x is a signed integer`
`if(x <= 20 || x >= 25){`
`  a = 1;`
`}`
	`; r0 = x, r1 = a`
	`CMP r0, #20 ; compare x and 20`
	`BLE then ; go to then if x <= 20`
	`CMP r0, #25 ; compare x and 25`
	`BLT endif ; go to endif if x < 25`
	`then MOV r1, #1 ; a = 1`
	`endif`

Example 6-5. A generic approach to implementing *if-then* with a compound logical OR

Example 6-6 gives a simplified assembly implementation that uses conditionally executed instructions.

C Program	Assembly Program

`// x is a signed integer`
`if(x <= 20 || x >= 25){`
`  a = 1;`
`}`
	`; r0 = x, r1 = a`
	`CMP r0, #20 ; compare x and 20`
	`MOVLE r1, #1 ; a = 1 if x <= 20`
	`CMP r0, #25 ; compare x and 25`
	`MOVGE r1, #1 ; a = 1 if x >= 25`

Example 6-6. Using Conditional execution to implement a compound logical OR.

Sometimes conditional comparison (such as CMPNE) and conditional execution can simplify the program, as shown below.

C Program	Assembly Program

`if(x == 20 || x == 25){`
`  a = 1;`
`}`
	`; r0 = x, r1 = a`
	`CMP  r0, #20  ; compare x and 20`
	`CMPNE r0, #25  ; CMP if r0 ≠ 25`
	`MOVEQ r1, #1  ; r1 = 1 if Z = 1`

Example 6-7. Conditional comparison (such as CMPNE) tests a compound expression

However, using conditional branch and execution can only implement an if-then statement in which the actions performed are simple. A generic approach to implementing an *if-then* structure with a compound logic OR expression is to use conditional branch instructions.

If-then statement with a compound logical AND expression

It is harder to test in assembly a compound logical expression combined by AND. De Morgan's laws are often used to break a logical AND compound expression into a logical OR expression.

$$\overline{A \text{ and } B} = \overline{A} \text{ or } \overline{B}$$

For example:

$$\begin{aligned}\overline{x > 20 \text{ and } x < 25} &= \overline{x > 20} \text{ or } \overline{x < 25} \\ &= x \leq 20 \text{ or } x \geq 25\end{aligned}$$

Therefore, when the condition of the if-statement is x > 20 && x < 25, in the assembly implementation given in Example 6-8, we test whether x ≤ 20 or x ≥ 25.

C Program	Assembly Program

`if(x > 20 && x < 25){`
`  a = 1;`
`}`
	`; Assume r0 = x, r1 = a`
	`CMP  r0, #20  ; compare x with 20`
	`BLE  endif  ; go to endif if x <= 20`
	`CMP  r0, #25  ; compare x with 25`
	`BGE  endif  ; go to endif if x >= 25`
	`MOVS r1, #1  ; a = 1`
	`endif`

Example 6-8. Using De Morgan's laws to convert a logical AND to a logical OR

If-then statement with a compound logical AND and OR expression

When a compound logical expression includes both AND and OR operators, the techniques introduced previously must be combined.

```
if ( x == 5 || (x > 20 && x < 25) )
    a = 1;
```

The following gives an example implementation.

```
        ; Assume r0 = x, r1 = a
        CMP   r0, #5      ; compare x with 5
        BEQ   then        ; if x == 5, go to then

        CMP   r0, #20     ; compare x with 20
        BLE   endif       ; go to endif if x ≤ 20

        CMP   r0, #25     ; compare x with 25
        BGE   endif       ; go to endif if x ≥ 25

then    MOVS  r1, #1      ; a = 1
endif
```

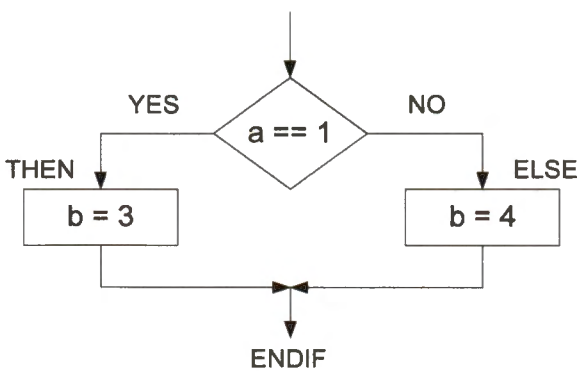
Example 6-9. Assembly implementation of a logic expression with both AND and OR

6.5 If-then-else Statement

The ***if-then-else*** statement selects one of two alternative sets of statements to execute. It first evaluates the given Boolean condition.

- If the condition is true, the statements following the *if* statement are executed.
- Otherwise, the statements following the *else* statement are executed.

```
C Program
if (a == 1)
   b = 3;
else
   b = 4;
```



In the C program shown above, variable *b* is set to 3 if *a* is 1; otherwise, *b* is set to 4.

Assume the content of variable *a* is stored in register r1, and *b* in register r2, Example 6-10 gives two equivalent implementations of the above *if-else* C program. One implementation is based on branch instructions, and the other uses conditionally executed instructions.

To make assembly code easy to understand, we should give each label a meaningful name, such as "then", "else", and "endif".

Assembly Program 1	Assembly Program 2

; r1 = a, r2 = b
CMP r1, #1 ; compare a and 1
BNE else ; go to else if a ≠ 1
then MOV r2, #3 ; b = 3
B endif ; go to endif
else MOV r2, #4 ; b = 4
endif
	; r1 = a, r2 = b
	CMP r1, #1 ; compare a and 1
	MOVEQ r2, #3 ; b = 3 if a = 1
	MOVNE r2, #4 ; b = 4 if a ≠ 1

Example 6-10. Assembly implementation of *if-then-else* based on conditional branch and conditional execution

6.6 For Loop

The *for* loop repeatedly executes a block of codes if the specified condition is satisfied. A for loop contains three expressions, as shown below.

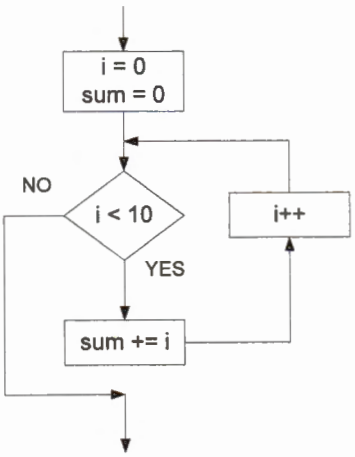
- The ***initial expression*** is executed only once often to initialize loop indices.
- The ***condition expression*** is tested before each iteration is executed. The loop body is executed if the condition expression is true. Note the loop body is skipped if the condition expression is false at the very first time it is evaluated.
- The ***loop expression*** is typically used to increment or decrement loop indices after each loop.

```
for (initial_expression; condition_expression; loop_experssion) {
  // loop body
  ...
}
```

The following C program calculates the sum of the first 10 non-negative integers.

```
C Program
int i;
int sum = 0;

for(i = 0; i < 10; i++){
  sum += i;
}
```



Assume r0 = *i* and r2 = *sum*. Example 6-11 gives three different approaches to translating the above *for* loop into the assembly.

Assembly Program 1

```
      MOV r0, #0  ; i
      MOV r1, #0  ; sum

      B   check
loop  ADD r1, r1, r0
      ADD r0, r0, #1
check CMP r0, #10
      BLT loop
endloop
```

Assembly Program 2

```
      MOV r0, #0  ; i
      MOV r1, #0  ; sum


loop  CMP r0, #10
      BGE endloop
      ADD r1, r1, r0
      ADD r0, r0, #1
      B   loop
endloop
```

Assembly Program 3

```
      MOV r0, #0  ; i
      MOV r1, #0  ; sum


loop  CMP   r0, #10
      ADDLT r1, r1, r0
      ADDLT r0, r0, #1
      BLT   loop
endloop
```

Example 6-11. Assembly implementation of *for* loop based on conditional branch and conditional execution

6.7 While Loop

A *while* loop tests the condition expression before executing the loop body. If the condition expression is true, the loop body is then executed. Otherwise, the loop is terminated. Thus, the loop body may not be performed.

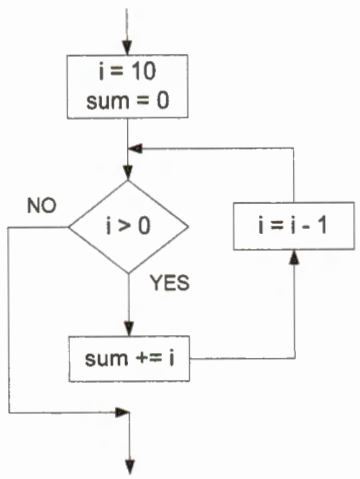
```
while (condition_expression) {
  // Loop body
  ...
}
```

The following C program uses a *while* loop to calculate the sum of the first 10 integers, starting with 0.

C Program

```
int i = 10;
int sum = 0;

while( i > 0 ){
  sum += i;
  i--;
}
```



Assume the variable *i* is saved in register r0 and the variable *sum* is stored in r1, Example 6-12 gives three different assembly implementations of the above while loop.

Assembly Program 1

```
      MOV  r0, #10 ; i
      MOV  r1, #0  ; sum

      B    check
loop  ADD  r1, r1, r0
      SUB  r0, r0, #1
check CMP  r0, #0
      BGT  loop
endloop
```

Assembly Program 2

```
      MOV  r0, #10 ; i
      MOV  r1, #0  ; sum


loop  CMP  r0, #0
      BLE  endloop
      ADD  r1, r1, r0
      SUB  r0, r0, #1
      B    loop
endloop
```

Assembly Program 3

```
      MOV  r0, #10 ; i
      MOV  r1, #0  ; sum


loop  CMP    r0, #0
      ADDGT  r1, r1, r0
      SUBGT  r0, r0, #1
      BGT    loop
endloop
```

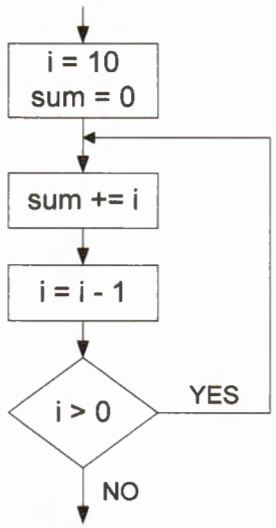
Example 6-12. Assembly implementation of *while* loop

The first implementation checks the condition expression at the end of the loop. The second and third implementations check the conditional expression at the beginning of the loop. Since the loop body is not large, the last implementation uses conditionally executed instructions.

6.8 Do While Loop

C Program

```
int sum = 0;
int i = 10;
do{
   sum += i;
   i--;
} while( i > 0 );
```



The *do-while* loop is like the *while* loop. The key difference is that the condition expression is evaluated before executing the loop body in the while loop, whereas the condition expression is assessed at the end of each iteration in the do-while loop.

Therefore, the *while* loop executes the loop body zero or more times sequentially, whereas the do-while loop executes the loop body at least once.

```
do {
  // loop body
    ...
} while (condition_expression)
```

Again, we use the example of calculating the sum of the first 10 integers, starting with 0. Example 6-12 gives two equivalent assembly implementations (r0 = i, r1 = sum).

Assembly Program 1			Assembly Program 2		
	MOV r0, #10	; i = 10		MOV r0, #10	; i = 0
	MOV r1, #0	; sum = 0		MOV r1, #0	; sum = 0
loop	ADD r1, r1, r0	; sum += i	loop	ADD r1, r1, r0	; sum += i
	SUB r0, r0, #1	; i--		SUBS r0, r0, #1	; i--
	CMP r0, #0			BGT loop	
	BGT loop		endloop		
endloop					

Example 6-13. Assembly implementation of *do-while* loop

The second implementation uses SUBS that performs subtraction and updates the N, Z, C, and V flags. Therefore, there is no need to use CMP before the BGT instruction.

6.9 Continue Statement

A *continue* statement in a loop is to skip the remaining statements in the current iteration and transfer the control to the next iteration of the loop. Example 6-14 calculates the sum of all integers between 0 and 9, excluding 5. The assembly program uses the condition code "NE" to skip the add instruction if r0 equals 5.

C Program	Assembly Program		

int i;
int sum = 0;
 	
for(i = 0; i < 10; i++) {
  if (i == 5) // skip 5
  continue;
  sum += i;
}
		MOVS r0, #0
		MOVS r1, #0
		
		CMP r0, #10
		BGE endloop
		CMP r0, #5
		ADDNE r1, r1, r0
		ADD r0, r0, #1
		B loop
			; i = 0
			; sum = 0
			
			; sum += i
			; i++

	endloop		

Example 6-14. Assembly implementation of *continue* in a loop

6.10 Break Statement

A ***break*** statement is to exit the current loop, including for, while, and do-while. It is useful when the number of iterations in a loop cannot be predetermined. When there are nested loops, the break statement terminates the nearest enclosing loop. It is easy to confuse the break and continue statements.

The following two C programs illustrate the difference between break and continue.

Example code for break	Example code for continue

`for(int i = 0; i < 5; i++){`
`  if (i == 2) break;`
`  printf("%d, ", i)`
`}`
	`for(int i = 0; i < 5; i++){`
	`  if (i == 2) continue;`
	`  printf("%d, ", i)`
	`}`

`Output: 0, 1,`	`Output: 0, 1, 3, 4,`

Example 6-15. Comparing *break* and *continue*

The break statement is translated to a conditional or an unconditional branch statement in assembly. The following shows how the break statement is implemented by a combination of `CBNZ` and `B` instructions.

C Program	Assembly Program

`// Find string length`
`char str[] = "hello";`
`int len = 0;`
`char *p = str;`
 	
`for( ; ; ) {`
 	
`  if (*p == '\0')`
`  break;`
 	
`  p++;`
`  len++;`
`}`
	`; r0 = string memory address`
	`; r1 = string length`
	`MOV r1, #0 ; len = 0`
	
	`loop LDRB r2, [r0]`
	
	`CBNZ r2, notZero`
	`B endloop`
	
	`notZero ADD r0, r0, #1 ; p++`
	`ADD r1, r1, #1 ; len++`
	
	`B loop`
	`endloop`

Example 6-16. Implementation of *break* in assembly

6.11 Switch Statement

A ***switch*** statement in C allows the program to make multiple choices based on a switch expression. If the value of the expression matches with one of the predetermined set of

integer values defined in the program, the program branches accordingly. When there are many choices, the switch statement makes the program more structured and easier to read than a combination of *if-then* or *if-then-else* statements.

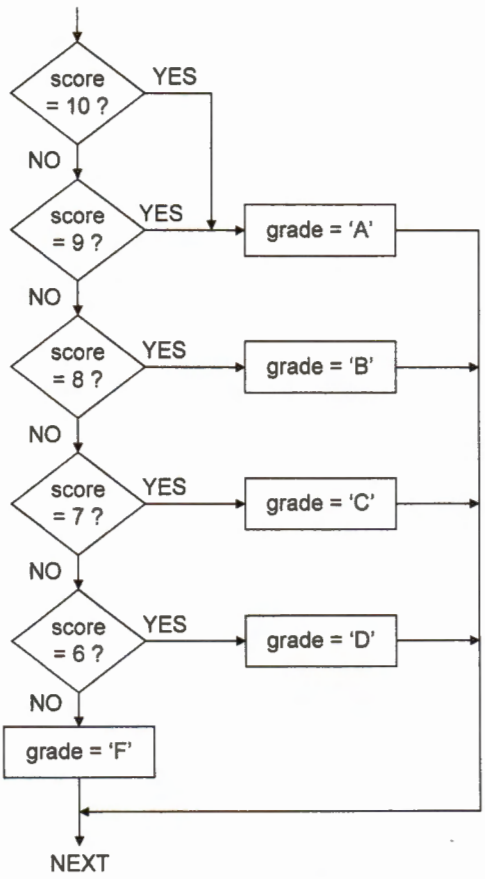


Figure 6-1. Flowcharts of an example *switch* program

The body of a switch structure consists of an optional default label, a series of case labels and case expressions, and lists of statements for each instance.

- The switch expression must be evaluated to an integer or a character.
- If the switch expression does not match any of the case constants, the default label is then selected.
- A break statement in the switch structure is used to exit the switch.
- After exiting, the processor then executes the instruction immediately after the switch structure.

Example 6-17 gives the implementation in C to convert a numeric grade to its corresponding letter grade. Figure 6-1 gives the flowchart.

The assembly code can use the table branch byte (`TBB`) instruction to implement the switch statement.

```
TBB [PC, r0]    ; PC = PC + 4 + 2 × BranchTable[r0]
```

`TBB` relies on the branch table defined immediately after the `TBB` instruction. In the branch table, each table item takes one byte, and it represents the offset in halfwords between the current `PC` and the memory address of the target instruction. The memory address of the instruction to which the program should branch is calculated as follows:

```
target = PC + 4 + 2 × BranchTable[r0]
```

Thus, the program counter (PC) is

$$PC = PC + 4 + 2 \times BranchTable[r0]$$

TBB [Rn, Rm]	Table branch byte
TBH [Rn, Rm, LSL #1]	Table branch halfword

Table 6-9. Table branch instructions

The table branch halfword (TBH) instruction is like TBB. However, each item in the branch table takes halfwords.

```
TBH [PC, r0]    ; PC = PC + 4 + 2 × BranchTable[r0]
```

We use a simple example to illustrate how to use TBB or TBH instruction to implement a switch statement in assembly.

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

C Program

```
uint32_t score;
char grade;

switch (score){
  case 10:

  case 9:
    grade = 'A';
    break;

  case 8:
    grade = 'B';
    break;

  case 7:
    grade = 'C';
    break;

  case 6:
    grade = 'D';
    break;

  default:
    grade = 'F';
    break;
}
```

Assembly Program

```
    ; r0 = numeric score ( 0 ≤ r0 ≤ 10 )
    ; r1 = letter grade

    SUBS r2, r0, #6    ; r2 is branch index
    CMP  r2, #5
    BHS  default       ; branch if unsigned r2 ≥ 5

    ; r2 is the index;
    ; pc = pc + 4 + 2 × BranchTable[r2]
    TBB  [pc, r2]      ; Table Branch Byte

BranchTable
    DCB  (case_6 - BranchTable)/2      ; index = 0
    DCB  (case_7 - BranchTable)/2      ; index = 1
    DCB  (case_8 - BranchTable)/2      ; index = 2
    DCB  (case_10_9 - BranchTable)/2   ; index = 3
    DCB  (case_10_9 - BranchTable)/2   ; index = 4

    ALIGN

case_10_9
    MOV  r1, #0x41      ; ASCII 'A' = 0x41
    B    exit

case_8
    MOV  r1, #0x42      ; ASCII 'B' = 0x42
    B    exit
```

```
case_7
    MOV  r1, #0x43      ; ASCII 'C' = 0x43
    B    exit

case_6
    MOV  r1, #0x44      ; ASCII 'D' = 0x44
    B    exit

default
    MOV  r1, #0x46      ; ASCII 'F' = 0x46

exit
```

Example 6-17. Converting score to letter grade

6.12 Exercises

1. Translate the following code into a C program and explain what it does.

```
        MOV r2, #1
        MOV r1, #1

loop    CMP r1, r0
        BGT done
        MUL r2, r1, r2
        ADD r1, r1, #1
        B   loop

done    MOV r0, r2
```

2. We can use the REV instruction to perform conversion between 32-bit little-endian and big-endian numbers. Write an assembly program that uses bitwise operators, such as &, |, ^, <<, and >>, to implement the endian conversion. You cannot use REV in your program.

3. Define an array with 10 unsigned integers in assembly code, and write an assembly program that calculates the mean of these 10 integers (truncating the result to an integer).

4. Define an array with 10 unsigned integers a_i ($0 \leq i \leq 9$) in assembly code, and write an assembly program that calculates the sum of the cube of these 10 unsigned integers.

$$sum = \sum_{i=0}^{9} a_i^3$$

The following defines the array and its size in the data memory.

```
        AREA myData, DATA
array   DCD  2, 4, 7, 3, 1, 2, 10, 11, 5, 13
size    DCD  10
```

5. Write an assembly program that converts all lowercase letters to their corresponding upper cases.

6. Write an assembly program that calculates the kinetic energy (E), $E = MC^2$, where the mass (M) is 15 kg and is stored in `r0`. C is the speed of light (299,792,458 m/s) and is stored in `r1`. The result E has 32-bit and is stored in register `r2`.

7. Write an assembly program that calculates the value of the following integer expression:

$$7x^2 + 9xy + \frac{3x}{y} + 11x + 13y + 5$$

where unsigned integers $x = 4$ and $y = 2$.

8. Test for complex roots in solution to the following quadratic equation:

$$ax^2 + bx + c = 0$$

The solution has complex roots if $b^2 - 4ac$ is smaller than `0` and real roots otherwise. Suppose a, b, and c are signed integers and they are stored in register `r0`, `r1`, and `r2`. Write an assembly program that set register `r3` to `1` if the solution has complex roots and `0` otherwise.

9. Write an assembly program that calculates the following function. Assume register `r0` holds the signed integer x, and register `r1` saves the result.

$$f(x) = \begin{cases} -1 & if\ x < 0 \\ 0 & if\ x = 0 \\ 1 & if\ x > 0 \end{cases}$$

10. Write an assembly program that calculates the following cost function. Assume the unsigned integer input x is stored in register `r0` and the cost is in register `r1`.

$$cost(x) = \begin{cases} 9x & if\ x \leq 10 \\ 8x & if\ x > 10\ and\ x \leq 100 \\ 7x & if\ x > 100\ and\ x \leq 1000 \\ 6x & if\ x > 1000 \end{cases}$$

11. Translate the following C program into an assembly program. The C program finds the minimal value of three signed integers. Assume *a*, *b*, and *c* is stored in register `r0`, `r1`, and `r3`, respectively. The result *min* is saved in register `r4`.

```
if (a ≤ b && a < c) {
   min = a;
} else if (b < a && b < c) {
   min = b;
} else {
   min = c;
}
```

12. Assume two dates are stored in memory as follows. Write an assembly program to compare these two dates. If date1 comes before date2, set register `r0` to `1`; otherwise, set `r0` to `-1`.

```
        AREA myData, DATA
date1   DCD 12, 31, 2014    ; month, day, year
date2   DCD 01, 20, 2013    ; month, day, year
```

13. Write an assembly program that calculates the sum as given below. Variable n is saved in register `r0`, and the *sum* is stored in register `r1`.

$$sum = \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2$$

14. Write an assembly program that calculates the factorial of a non-negative integer n. Assume n is given in register `r0`, and the result is saved in register `r1`.

$$f(n) = \prod_{i=1}^{n} i = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

15. When a two-dimensional (2D) matrix is declared in a C program, the matrix, in fact, is stored as a one-dimensional array in memory. C program uses a row-major approach to convert a 2D matrix into a 1D array. The following gives an example of storing a 3-by-3 matrix in memory.

Index	(0, 0)	(0, 1)	(0, 2)	(1, 0)	(1, 1)	(1, 2)	(2, 0)	(2, 1)	(2, 2)
Content	1	2	3	4	5	6	7	8	9
Memory offset in bytes	0	4	8	12	16	20	24	28	32

1st Row 2nd Row 3rd Row

Translate the following C program to an assembly program. Your assembly program must consist of two nested loops.

```
int a[4][3] = {
    {11, 12, 13}, // first row
    {21, 22, 23}, // second row
    {31, 32, 33}, // third row
    {41, 42, 43}  // fourth row
};

void main(void) {
   int i, j;
   for(i = 0; i < 4; i++)
      for(j = 0; j < 3; j++)
          a[i][j] = 2*a[i][j];
   return;
}
```

16. Write an assembly program that transposes the matrix defined in the previous question. Your assembly program should have two nested loops. In linear algebra, the transpose of a matrix $[a_{ij}]_{m\times n}$ is $[a_{ji}]_{n\times m}$.

$$[a_{ij}]_{m\times n}^{T} = [a_{ji}]_{n\times m}$$

For example:

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \\ 41 & 42 & 43 \end{bmatrix}^{T} = \begin{bmatrix} 11 & 21 & 31 & 41 \\ 12 & 22 & 32 & 42 \\ 13 & 23 & 33 & 43 \end{bmatrix}$$