Chapter 1
68000 Assembly Language

Latest update: 07/09/2022

Table of Contents

I. Introduction

1. Machine Language

The machine language (or machine code) is the native language of a microprocessor. It is made up of successive binary words called ‘operation code’ or ‘opcode’. Each opcode represents an operation that can be executed by the microprocessor.

The machine language is the sole language that can be executed by a microprocessor and each microprocessor has its own machine language. Nevertheless, some microprocessors (particularly those belonging to a same family) can have compatible or similar languages.

In this lesson, we will study the **Motorola 68000 microprocessor**.

2. Assembler

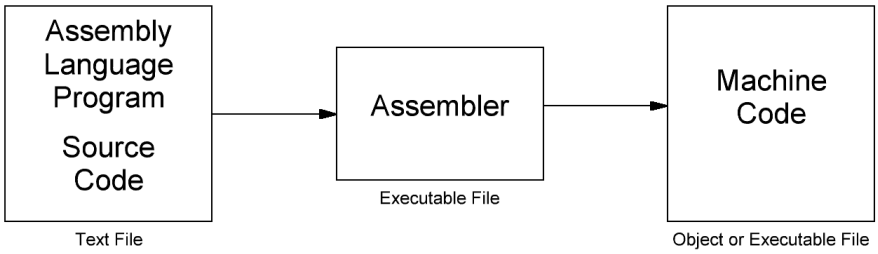
The term ‘assembler’ has two different meanings:
- It can refer to a programming language: the assembly language.
- It can refer to a computer program: the assembly program.

2.1. Assembly Language

The assembly language (or assembler) gives names to machine code instructions. These names are called ‘mnemonics’. Therefore, the assembly language is very close to the machine language, but much more readable for human beings.

2.2. Assembly Program

The assembly program (or assembler) is the computer program that converts an assembly language program into machine code. This conversion process is referred to as ‘assembly’. Several assembly programs can be found for the same assembly language (as several C compilers can be found for the C language).



Once the source code has been converted, the machine code can be loaded into memory and executed by the microprocessor.

3. Assembly Language Program Format

Here is an example of an assembly language program:

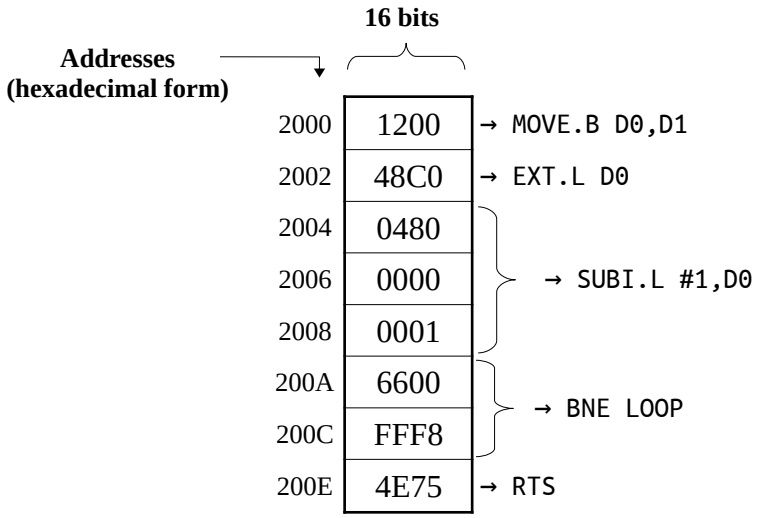
```
        ORG     $2000   ; Set the origin of the program.
START   MOVE.B  D0,D1   ; D0 → D1.
        EXT.L   D0      ; Sign extension.
LOOP    SUBI.L  #1,D0   ; D0 - 1 → D0.
        BNE     LOOP    ; Go to LOOP if D0 is not equal to zero.
        RTS             ; Return from subroutine.
```

This source code contains some new instructions for you. Do not try to understand them for the time being. In itself, this program does not do anything interesting. It is just an example to identify the different parts of a source code.

A source code contains one instruction per line and an instruction is divided into four fields.

Label	Mnemonic	Operand	Comment
	ORG	$2000	; Set the origin of the program.
START	MOVE.B	D0,D1	; D0 → D1.
	EXT.L	D0	; Sign extension.
LOOP	SUBI.L	#1,D0	; D0 - 1 → D0.
	BNE	LOOP	; Go to LOOP if D0 is not equal to zero.
	RTS		; Return from subroutine.

Once the program has been assembled and loaded into memory, here is what the machine code looks like:



Opcodes are stored in the memory as 16-bit words. It is noteworthy that some instructions require more memory space than others.

About the 68000,

- An instruction is made up of at least one 16-bit word (2 bytes).
- An instruction can be as long as five 16-bit words (10 bytes).

For a reason discussed later, the first line (`ORG $2000`) has not been converted into machine code.

Representation of the assembly language program and its associated machine code:

```
                          ORG     $2000  ; Set the origin of the program.
002000 1200          START  MOVE.B D0,D1  ; D0 → D1.
002002 48C0                 EXT.L  D0     ; Sign extension.
002004 0480 00000001 LOOP   SUBI.L #1,D0  ; D0 - 1 → D0.
00200A 6600 FFF8            BNE    LOOP   ; Go to LOOP if D0 is not equal to zero.
00200E 4E75                 RTS           ; Return from subroutine.
```

3.1. Label Field

The label field is the first field of an instruction.

A label is optional. If present, it comes at the beginning of a line. Otherwise, the field must contain at least one white-space character (blank or tab).

The assembler gives the label the address where the instruction following the label will be loaded into memory. For instance, in our case:

- The label `START` will be assigned the value 2000_{16}.
- The label `LOOP` will be assigned the value 2004_{16}.

These labels can then be used as operands in the source code. The assembler will replace them by their address values. Therefore, the programmer does not have to care about address locations.

Labels are commonly used in branch instructions.

3.2. Mnemonic Field

The mnemonic field contains the name of an **instruction** or the name of an **assembler directive**.

An instruction can be converted into machine code. It belongs to the instruction set of a microprocessor.

An assembler directive (also called ‘pseudo-operation’ or ‘pseudo-op’) cannot be converted into machine code. It does not belong to the instruction set of a microprocessor.

An assembler directive belongs to the assembly program, not to the assembly language. It allows the programmer to give directives to the assembler.

That is the reason why the first line of our program was not converted into machine code. The `ORG` mnemonic does not represent an instruction of the assembly language but an instruction of the assembly program. In other words, it is an assembler directive. This pseudo-op tells the assembler to set the origin of the program at the address specified in the operand field (2000_{16} in our example).

A size attribute can be added to the mnemonic in order to specify the size of the operands. The three main size attributes are:

- **`.B`** – Byte operands (8 bits)
- **`.W`** – Word operands (16 bits)
- **`.L`** – Long-word operands (32 bits)

There is also the attribute **`.S`** (Short) that can be used by some branch instructions. It is then similar to the attribute **`.B`** (Byte).

3.3. Operand Field

Some instructions or assembler directives require data. Operands are used to specify the location of this data.

Concerning the 68000, some instructions do not require any operands, some require one operand and some require two. If there are two operands, the left one is referred to as the ‘source operand’ and the right one is referred to as the ‘destination operand’.

For instance, in the following instruction: `MOVE.B D0,D1`

- **D0** is the source operand (it will not be modified by the instruction).
- **D1** is the destination operand (it will be modified by the instruction).

We will see later on what **D0** and **D1** are about.

Some operands can be numbers. The 68000 assembly language allows the base-2, base-16 and base-10 representations:

- A binary number (base 2) is prefixed with the character **‘%’** (e.g. `%10001001`).
- A hexadecimal number (base 16) is prefixed with the character **‘$’** (e.g. `$2EF8`).
- A decimal number (base 10) is not prefixed.

3.4. Comment Field

The purpose of this field is to make comments on the program.

Comments must be preceded by a semicolon; the assembler ignores any characters following a semicolon.

Comments have no effect on the generated machine code, but they are fundamental in making a source code easier to understand. Good comments are an essential part of good coding practice.

Note:
A comment may appear anywhere on a line, even at the beginning.

II. 68000 Architecture

1. Address and Data Buses

The 68000 has a 23-bit address bus and a 16-bit data bus, which allow it to access 8 Mi words of 16 bits.

The 68000 has also some control signals on its control bus (and particularly **UDS** and **LDS**) that allow it to access the memory byte per byte and perform as if its address bus had 24 lines.

Therefore, in this chapter, we will consider that the 68000 has a **24-bit address bus** and is able to access 16 Mi words of 8 bits, that is to say **16 MiB**.

Even if the 68000 is able to access the memory byte per byte, its memory space is physically made up of 16-bit words, which implies, among other things, the following characteristics:

- An instruction is made up of at least one 16-bit word.
- An instruction is always located at an even address.
- A byte can be accessed from either an even or an odd address.
- A 16-bit word can only be accessed from an even address.
- A 32-bit word can only be accessed from an even address.

2. Privilege Modes

The 68000 operates in one of two levels of privilege: the supervisor mode or the user mode.

2.1. Supervisor Mode

The supervisor mode has the higher level of privilege. In this mode, all the instructions of the microprocessor can be executed.

The supervisor mode is mostly used by operating systems, which require an absolute control of the computer.

2.2. User Mode

The user mode has the lower level of privilege. In this mode, some instructions of the microprocessor cannot be executed. These instructions are referred to as ‘privileged instructions’ and can only be executed in the supervisor mode.

The user mode is mostly used by applications running on an operating system. Limited privileges are required in order to protect the system against application crashes.

3. Registers

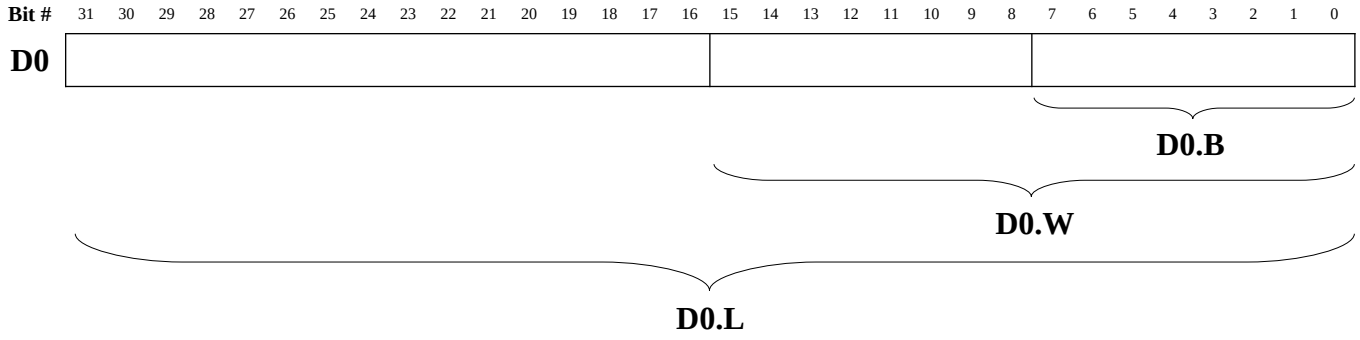
A register is a temporary storage unit within a microprocessor.

3.1. Data Registers

The 68000 has eight 32-bit data registers: **D0**, **D1**, **D2**, **D3**, **D4**, **D5**, **D6** and **D7**.

They are general-purpose registers and can hold any type of data. They can be accessed by 8-bit, 16-bit or 32-bit operations.

Example of the register **D0**:

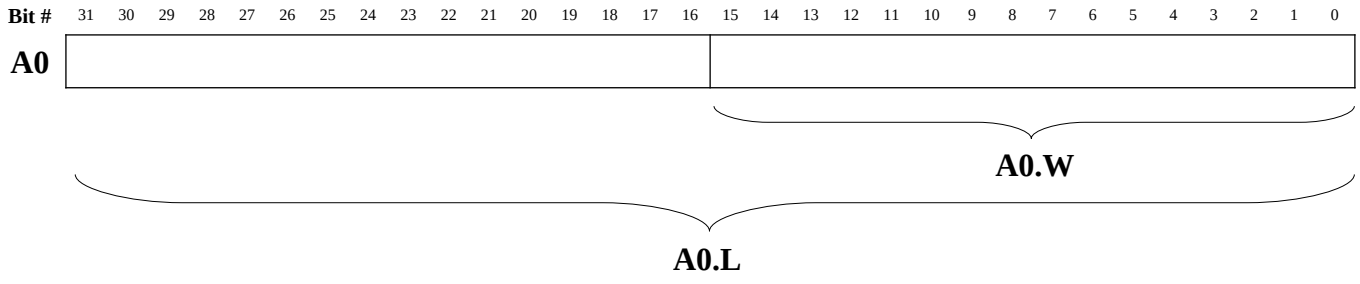


3.2. Address Registers and Stack Pointers

The 68000 has eight 32-bit address registers: **A0**, **A1**, **A2**, **A3**, **A4**, **A5**, **A6** and **A7**.

These registers hold addresses. They can be accessed by 16-bit and 32-bit operations.

Example of the register **A0**:



Since the 68000 has 24 address lines, the eight most significant bits of the address registers will be ignored. These bits will never be sent to the address bus. For instance, if **A0** = $67**9809AC** or if **A0** = $F5**9809AC** or if **A0** = $00**9809AC**, the memory cell that can be accessed from this register is located at the address **$9809AC**. In this chapter, the eight most significant bits of the address registers will always be set to zero.

The 68000 has two 32-bit stack pointers.

- **SSP** – Supervisor Stack Pointer
- **USP** – User Stack Pointer

These two stack pointers can only be accessed through the address register **A7**:

- When the supervisor mode is active, the address register **A7** is actually the register **SSP**.
- When the user mode is active, the address register **A7** is actually the register **USP**.

For instance, when the register **A7** is written to in the supervisor mode, then read from in the user mode, the read value will be different from the written value. This duplication of the register **A7** is useful for handling one stack per mode.

We will explain how a stack works later on.

3.3. Program Counter

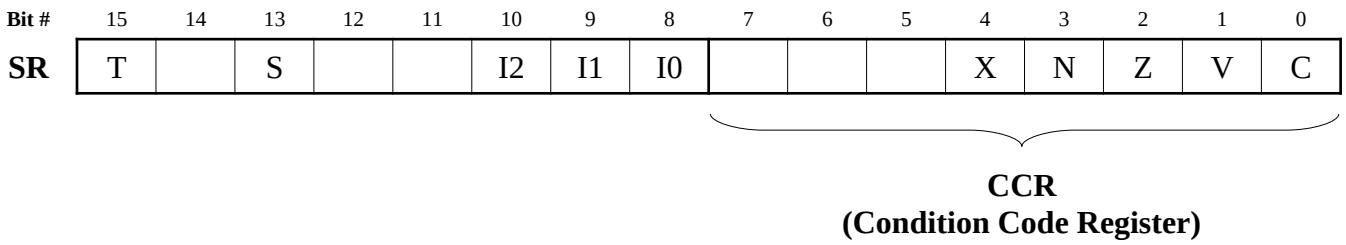
The program counter **PC** is a 32-bit register that holds the address of the next instruction to be executed. The contents of the **PC** are automatically modified by the microprocessor.

As for the address registers, the eight most significant bits of the **PC** are ignored and only its 24 least significant bits are sent to the address bus.

3.4. Status Register

The status register **SR** is a 16-bit register that holds some information about the state of the 68000.

The 16-bit register **SR** can only be accessed in the supervisor mode. However, the 8-bit register **CCR** (Condition Code Register), which can be accessed in the user mode, is made up of the eight least significant bits of the **SR**. Therefore, the eight least significant bits of the **SR** can be accessed from the **CCR**.



The different bits of the **SR** are referred to as ‘flags’. These flags contain additional information about the result of an operation (bits from 0 to 7, also called ‘condition codes’) or about the state of the microprocessor (bits from 8 to 15).

The general functions of the flags are as follows:

- **C** – Carry – Set if a carry or a borrow occurs (unsigned overflow); otherwise clear.
- **V** – Overflow – Set if a signed overflow occurs; otherwise clear.
- **Z** – Zero – Set if the result equals zero; otherwise clear.
- **N** – Negative – Set if the most significant bit of the result is set; otherwise clear.
- **X** – Extend – Set to the same value of **C** for most of the instructions.
- **I0**, **I1**, **I2** – Interrupt Priority Mask. We will never use interrupts in this chapter.
- **S** – Supervisor State – Set if the supervisor state is active; otherwise clear.
- **T** –Trace Mode – Set if the trace mode is active. We will never use the trace mode in this chapter.

The behaviour of some condition codes can slightly differ between instructions. Therefore, the programmer has to refer to the datasheet of the 68000 in order to know exactly how an instruction affects the condition codes.

Here are some examples of how the condition codes **C**, **V**, **Z** and **N** are affected by addition instructions:

- **Addition of 8-bit numbers (.B)**
  $7A + $86 = $100 (the 8-bit result is $00)
  C = 1, **V** = 0, **Z** = 1, **N** = 0

- **Addition of 16-bit numbers (.W)**
  $9F00 + $8E00 = $12D00 (the 16-bit result is $2D00)
  C = 1, **V** = 1, **Z** = 0, **N** = 0

- **Addition of 32-bit numbers (.L)**
  $70000000 + $10000000 = $80000000
  C = 0, **V** = 1, **Z** = 0, **N** = 1

- **N** = 1, if the most significant bit of the result is one.
- **Z** = 1, if the result equals zero.
- **C** = 1, if a carry occurs (assuming that the numbers are unsigned).
- **V** = 1, if an overflow occurs (assuming that the numbers are signed).
  To determine the value of **V** for an addition, perform the addition assuming that the numbers and the result are signed. Then **V** = 1, if one of the two conditions below is met:
  - The sum of two positive numbers is negative.
  - The sum of two negative numbers is positive.

III. The Main Assembler Directives

1. The Assembler Directive ORG

The assembler directive `ORG` means ‘origin’ and tells the assembler where instructions or data must be loaded into memory.

Several `ORG` directives may appear on a source code. (Be careful not to overwrite instructions or data.)

Examples :

```
        ORG     $1000

        ; These instructions are to be loaded
        ; into memory starting at address $1000.
        LEA     $2000,A0
        CLR.B   D1
        JSR     $5000

        ORG     $5000

        ; These instructions are to be loaded
        ; into memory starting at address $5000.
        MOVE.B  (A0)+,D0
        ADDQ.B  #1,D1
        RTS
```

2. The Assembler Directive EQU

The assembler directive `EQU` means ‘equate’ and tells the assembler to equate a label to a numerical constant. The assembler will then replace each occurrence of the label by its numerical constant.

Examples :

```
START       EQU     $1000
PRINT       EQU     $5000
COUNT1      EQU     200
COUNT2      EQU     850

            ORG     START           ; ORG     $1000

            MOVE.L  #COUNT1,D0      ; MOVE.L  #200,D0
            JSR     PRINT           ; JSR     $5000

            MOVE.L  #COUNT2,D0      ; MOVE.L  #850,D0
            JSR     PRINT           ; JSR     $5000
```

3. The Assembler Directive DC

The assembler directive `DC` means ‘define constant’ and tells the assembler to load data into memory. These data can be decimal, hexadecimal or binary numbers but also strings of characters (characters are stored as ASCII codes).

Examples:

```
        ORG     $1000

        DC.B    10,5,7,$7a,255,%11111001
        DC.B    "Hello World",13,10,0
        DC.W    5,6
        DC.L    5,6
```

Contents of the memory from the address $1000:

```
1000 0A 05 07 7A FF F9                           DC.B    10,5,7,$7a,255,%11111001
1006 48 65 6C 6C 6F 20 57 6F 72 6C 64 0D 0A 00   DC.B    "Hello World",13,10,0
1014 00 05 00 06                                 DC.W    5,6
1018 00 00 00 05 00 00 00 06                     DC.L    5,6
```

4. The Assembler Directive DS

The assembler directive `DS` means ‘define storage’ and tells the assembler to reserve memory space which can then be used by the program to store data dynamically without overwriting any other data or instructions.

Examples:

```
        ORG     $5000

TAB1    DS.B    4  ; Reserve 4 bytes of storage     (4 x  8 bits)  ; TAB1 = $5000
TAB2    DS.W    3  ; Reserve 3 words of storage     (3 x 16 bits)  ; TAB2 = $5004
TAB3    DS.L    1  ; Reserve 1 long word of storage (1 x 32 bits)  ; TAB3 = $500A
NEXT                                                               ; NEXT = $500E
```

IV. Addressing Modes

An addressing mode is the way in which the location of data required by an instruction is specified.

In order to illustrate the different addressing modes, we will go through practical examples using the `MOVE` instruction. This instruction copies the contents of a source operand into a destination operand; the size of the operation can be 8, 16 or 32 bits.

```
MOVE.B  source,destination  ; source → destination (8-bit operation)
MOVE.W  source,destination  ; source → destination (16-bit operation)
MOVE.L  source,destination  ; source → destination (32-bit operation)
```

1. Effective Address

The effective address can be seen as the location of an operand. **Caution! An effective address can be different from a memory address.** An operand can be located in a register, in the memory or in the opcodes of an instruction.

Two types of addressing modes can be distinguished:

- **The addressing modes not specifying a memory location** (the effective address is not a memory address). These are the ‘direct modes’ and the ‘immediate data’.
- **The addressing modes specifying a memory location** (the effective address is a memory address). These are the ‘indirect modes’ and the ‘absolute modes’.

In the datasheet of the 68000, the effective address is often abbreviated to **<ea>**. Therefore, we will use the same abbreviation.

2. Addressing Modes Not Specifying a Memory Location

The operand is located either in a register or in the opcode of an instruction. Determining a specific memory address is irrelevant.

2.1. Data Register Direct: Dn

The effective address is a data register. The operand can be accessed directly from a data register. The size of the operand is relative to the size of the operation (8, 16, or 32 bits).

Examples:
Initial values: `D0 = $11223344` and `D1 = $AABBCCDD`

```
MOVE.B  D0,D1  ; D0.B → D1.B ; D1 = $AABBCC44 (8-bit operation)
MOVE.W  D0,D1  ; D0.W → D1.W ; D1 = $AABB3344 (16-bit operation)
MOVE.L  D0,D1  ; D0.L → D1.L ; D1 = $11223344 (32-bit operation)
```

2.2. Address Register Direct: An

The effective address is an address register. The operand can be accessed directly from an address register. The size of the operand is relative to the size of the operation (16 or 32 bits).

Examples:
Initial values: `D0 = $11223344` and `A0 = $005030B0`

```
MOVE.W  A0,D0  ; A0.W → D0.W ; D0 = $1122 30B0
MOVE.L  A0,D0  ; A0.L → D0.L ; D0 = $0050 30B0
```

Note:
The `MOVE` instruction cannot use any address register as a destination operand. Instead, the `MOVEA` or `LEA` instructions can be used to load a value in the address registers.

2.3. Immediate Data: #<data>

The operand is located in the opcode of an instruction. The data is specified in the source and preceded by the character '**#**'.

In this mode, the effective address is irrelevant because the data is immediately available from the instruction.

Examples:
Initial value: `D0 = $11223344`

```
MOVE.B  #$FF,D0   ; D0 = $112233FF (The 8-bit value is loaded into D0.B)
MOVE.W  #$7A8,D0  ; D0 = $11227A8 (The 16-bit value is loaded into D0.W)
MOVE.L  #$7A8,D0  ; D0 = $000007A8 (The 32-bit value is loaded into D0.L)
```

Notes:

- The character '**$**' is used to represent numbers in hexadecimal representations. Therefore, the three instructions below are equivalent to the three instructions above:

```
MOVE.B  #255,D0   ; D0 = $112233FF (The 8-bit value is loaded into D0.B)
MOVE.W  #1960,D0  ; D0 = $11227A8 (The 16-bit value is loaded into D0.W)
MOVE.L  #1960,D0  ; D0 = $000007A8 (The 32-bit value is loaded into D0.L)
```

  With $255_{10} = FF_{16}$ and $1960_{10} = 7A8_{16}$

- An immediate value can never be used as a destination operand. Obviously, it is not possible to load any data into an immediate value.

3. Addressing Modes Specifying a Memory Location

The operand is located in memory. Therefore the effective address is the memory address where the operand is located.

We will use the initial values below for the different examples of the addressing modes.
Memory and registers will be reinitialized for each addressing mode.

Initial values:

```
D0 = $11223344  A0 = $00001000  PC = $00002000
D1 = $AABBCCDD  A1 = $00001008
D2 = $0000FFFF  A2 = $00001010
D3 = $00000003  A3 = $00000006

$001000 21 45 87 AF B5 F3 3C 32
$001008 AD 45 39 98 9A 9B 9C 9D
$001010 03 69 01 00 12 0A 0D C9
```

3.1. Address Register Indirect: (An)

The effective address is the value of the address register: **<ea> = An**

Examples:

```
MOVE.B  (A0),D0   ; ($1000) → D0.B ; D0 = $11223321
MOVE.W  (A0),D0   ; ($1000) → D0.W ; D0 = $11222145
MOVE.L  (A0),D0   ; ($1000) → D0.L ; D0 = $214587AF

MOVE.B  D1,(A0)   ; D1.B → ($1000) ; $1000 DD 45 87 AF B5 F3 3C 32
MOVE.W  D1,(A0)   ; D1.W → ($1000) ; $1000 CC DD 87 AF B5 F3 3C 32
MOVE.L  D1,(A0)   ; D1.L → ($1000) ; $1000 AA BB CC DD B5 F3 3C 32

MOVE.B  (A1),(A2) ; ($1008) → ($1010) ; $1010 AD 69 01 00 12 0A 0D C9
MOVE.W  (A1),(A2) ; ($1008) → ($1010) ; $1010 AD 45 01 00 12 0A 0D C9
MOVE.L  (A1),(A2) ; ($1008) → ($1010) ; $1010 AD 45 39 98 12 0A 0D C9
```

3.2. Address Register Indirect with Postincrement: (An)+

The effective address is the value of the address register: **<ea> = An**

Once the 68000 has accessed the operand, the address register is incremented. The increment is relative to the size of the operation:

- **An = An + 1** for an 8-bit operation (**.B**)
- **An = An + 2** for a 16-bit operation (**.W**)
- **An = An + 4** for a 32-bit operation (**.L**)

Examples:

```
MOVE.W  (A0)+,D0 ; ($1000) → D0.W ; D0 = $11222145 ; A0 = $1002
MOVE.L  (A0)+,D0 ; ($1002) → D0.L ; D0 = $87AFB5F3 ; A0 = $1006

MOVE.B  D1,(A1)+ ; D1.B → ($1008) ; $1008 DD 45 39 98 9A 9B 9C 9D ; A1 = $1009
MOVE.B  D1,(A1)+ ; D1.B → ($1009) ; $1008 DD DD 39 98 9A 9B 9C 9D ; A1 = $100A
MOVE.W  D1,(A1)+ ; D1.W → ($100A) ; $1008 DD DD CC DD 9A 9B 9C 9D ; A1 = $100C
MOVE.L  D1,(A1)+ ; D1.L → ($100C) ; $1008 DD DD CC DD AA BB CC DD ; A1 = $1010
```

3.3. Address Register Indirect with Predecrement: –(An)

The address register is decremented before the 68000 accesses the operand. The decrement is relative to the size of the operation:

- **An = An – 1** for an 8-bit operation (**.B**)
- **An = An – 2** for a 16-bit operation (**.W**)
- **An = An – 4** for a 32-bit operation (**.L**)

The effective address is the value of the address register. **Caution! This value is the new value of the address register (the decremented value): <ea> = An**

Examples:

```
MOVE.B  -(A1),D0 ; A1 = $1007 ; ($1007) → D0.B ; D0 = $11223332
MOVE.B  -(A1),D0 ; A1 = $1006 ; ($1006) → D0.B ; D0 = $1122333C
MOVE.W  -(A1),D0 ; A1 = $1004 ; ($1004) → D0.W ; D0 = $1122B5F3
MOVE.L  -(A1),D0 ; A1 = $1000 ; ($1000) → D0.L ; D0 = $214587AF

MOVE.B  D1,-(A2) ; A2 = $100F ; D1.B → ($100F) ; $1008 AD 45 39 98 9A 9B 9C DD
MOVE.B  D1,-(A2) ; A2 = $100E ; D1.B → ($100E) ; $1008 AD 45 39 98 9A 9B DD DD
MOVE.W  D1,-(A2) ; A2 = $100C ; D1.W → ($100C) ; $1008 AD 45 39 98 CC DD DD DD
MOVE.L  D1,-(A2) ; A2 = $1008 ; D1.L → ($1008) ; $1008 AA BB CC DD CC DD DD DD
```

3.4. Address Register Indirect with Displacement: d16(An)

The effective address is the sum of an address register and a displacement: **<ea> = An + d16**

d16 is a 16-bit signed displacement: **–32768 ≤ d16 ≤ +32767**

Note:
There are two equivalent syntaxes for this addressing mode: **d16(An)** and **(d16,An)**
We will use the first one: **d16(An)**

Examples:

```
MOVE.B  -5(A1),D0 ; ($1003) → D0.B ; D0 = $112233AF
MOVE.W  4(A1),D0  ; ($100C) → D0.W ; D0 = $11229A9B

MOVE.B  D1,-1(A1) ; D1.B → ($1007) ; $1000 21 45 87 AF B5 F3 3C DD
MOVE.L  D1,-4(A1) ; D1.L → ($1004) ; $1000 21 45 87 AF AA BB CC DD
```

3.5. Address Register Indirect with Displacement and Index: d8(An,Xn)

The effective address is the sum of an address register, a displacement and an index:
<ea> = An + d8 + Xn

- **d8** is an 8-bit signed displacement: **–128 ≤ d8 ≤ +127**
- **Xn** is a 16-bit or 32-bit signed index. This index can be either a data register or an address register: **Dn.W, Dn.L, An.W or An.L**.

Note:
There are two equivalent syntaxes for this addressing mode: **d8(An,Xn)** and **(d8,An,Xn)**
We will use the first one: **d8(An,Xn)**

Examples:

```
MOVE.B  2(A1,A3.W),D0  ; ($1010) → D0.B ; D0 = $11223303
MOVE.W  1(A1,D3.L),D0  ; ($100C) → D0.W ; D0 = $11229A9B

MOVE.B  D1,0(A1,D2.W)  ; D1.B → ($1007) ; $1000 21 45 87 AF B5 F3 3C DD
MOVE.L  D1,-3(A1,D2.W) ; D1.L → ($1004) ; $1000 21 45 87 AF AA BB CC DD
```

3.6. Program Counter Indirect with Displacement: d16(PC)

The effective address is the sum of the program counter and a displacement: **<ea> = PC + d16**

d16 is a 16-bit signed displacement: **–32768 ≤ d16 ≤ +32767**

Notes:
There are two equivalent syntaxes for this addressing mode: **d16(PC)** and **(d16,PC)**
We will use the first one: **d16(PC)**

During the programming process, the programmer does not know the value of the program counter (**PC**) for each instruction. But if this value is unknown, the value of the effective address cannot be known either. Actually, this addressing mode is unusable if we use the syntax **d16(PC)** as it is. That is the reason why, in a source code, the programmer specifies the effective address directly without looking at either the value of the program counter or the value of the displacement.

Therefore, the syntax used in practice is: <ea>(PC)

The assembler will calculate the displacement according to the effective address and the program counter. The programming process is then simplified because the effective address appears clearly in the source code. As a result, the programmer does not have to perform the addition himself (**PC + d16**).

Examples:

```
MOVE.B  $1010(PC),D0  ; ($1010) → D0.B ; D0 = $11223303
MOVE.W  $100C(PC),D0  ; ($100C) → D0.W ; D0 = $11229A9B
MOVE.L  $1002(PC),D0  ; ($1002) → D0.L ; D0 = $87AFB5F3

; The MOVE instruction cannot use this address mode as a destination operand.
```

3.7. Program Counter Indirect with Displacement and Index: d8(PC,Xn)

The effective address is the sum of the program counter, a displacement and an index:
<ea> = PC + d8 + Xn

- **d8** is an 8-bit signed displacement: **–128 ≤ d8 ≤ +127**
- **Xn** is a 16-bit or 32-bit signed index. This index can be either a data register or an address register: **Dn.W, Dn.L, An.W or An.L**.

We will never use this addressing mode in this chapter.

3.8. Absolute Long: (xxx).L

The effective address is directly specified in the source code. The address is 32 bits wide.

Examples:

```
MOVE.B  $1010,D0  ; ($1010) → D0.B ; D0 = $11223303
MOVE.W  $100C,D0  ; ($100C) → D0.W ; D0 = $11229A9B

MOVE.B  D1,$1007  ; D1.B → ($1007) ; $1000 21 45 87 AF B5 F3 3C DD
MOVE.L  D1,$1004  ; D1.L → ($1004) ; $1000 21 45 87 AF AA BB CC DD
```

3.9. Absolute Short: (xxx).W

This addressing mode is similar to the ‘absolute long mode’, but deals with 16-bit addresses only. So the execution of an instruction is faster.

We will never use this addressing mode in this chapter.

4. Examples

In order to illustrate in more detail how addressing modes work, here are some instructions where all effective-address calculations are shown. The contents of registers (except the **PC**) and memory that have just been modified will also be specified.

For each instruction, memory and registers will be reset to the values below:

Initial values:
```
D0 = $0000FFFF  A0 = $00001000  PC = $00002000
D1 = $00000004  A1 = $00001008
D2 = $FFFFF000  A2 = $00001010

$001000  21 45 87 AF B5 F3 3C 32
$001008  AD 45 39 98 9A 9B 9C 9D
$001010  03 69 01 00 12 0A 0D C9
```

4.1. `MOVE.W A1,D2`

Source	Destination
A1.W **#$1008**	**D2.W**

D2 = $FFFF**1008**

4.2. `MOVE.W (A1),D2`

Source	Destination
(A1) **#$AD45**	**D2.W**

D2 = $FFFF**AD45**

4.3. `MOVE.L #$100A,D2`

Source	Destination
#$100A **#$0000100A**	**D2.L**

D2 = $0000100A

4.4. `MOVE.L $100A,D2`

Source	Destination
($100A) **#$39989A9B**	**D2.L**

D2 = $39989A9B

4.5. `MOVE.W #36,(A0)`

Source	Destination
#36 **#$0024**	(A0) **($1000)**

```
$001000  00 24 87 AF B5 F3 3C 32
```

4.6. `MOVE.B D1,(A1)+`

Source	Destination
D1.B **#$04**	(A1) **($1008)**

```
$001008  04 45 39 98 9A 9B 9C 9D      A1 = $00001009
```

4.7. `MOVE.L $1004,-(A2)`

Source	Destination
($1004) **#$B5F33C32**	(A2) ($1010 - 4) **($100C)**

```
$001008  AD 45 39 98 B5 F3 3C 32      A2 = $0000100C
```

4.8. `MOVE.L -(A2),-(A2)`

Source	Destination
(A2) ($1010 - 4) ($100C) **#$9A9B9C9D**	(A2) ($100C - 4) **($1008)**

```
$001008  9A 9B 9C 9D 9A 9B 9C 9D      A2 = $00001008
```

4.9. `MOVE.B 5(A1),-1(A1,D0.W)`

Source	Destination
5(A1) (A1 + 5) ($1008 + 5) ($100D) **#$9B**	-1(A1,D0.W) (A1 + D0.W - 1) ($1008 – 1 - 1) **($1006)**

```
$001000  21 45 87 AF B5 F3 9B 32
```

4.10. `MOVE.W 2(A1,D1.L),-6(A2)`

Source	Destination
2(A1,D1.L) (A1 + D1 + 2) ($1008 + 4 + 2) ($100E) **#$9C9D**	-6(A2) (A2 - 6) ($1010 - 6) **($100A)**

```
$001008  AD 45 9C 9D 9A 9B 9C 9D
```

4.11. `MOVE.W $1000(PC),$100A`

Source	Destination
($1000) **#$2145**	**($100A)**

```
$001008  AD 45 21 45 9A 9B 9C 9D
```

V. Branch Instructions

1. Unconditional Branch Instructions

The 68000 has two unconditional branch instructions:

BRA	Unconditional branch
JMP	Unconditional jump

These two instructions are similar. Nonetheless, they have slight differences in terms of addressing modes and machine code, but this point will not be touched on here. So we will assume that when the operand is either an address or a label, these two instructions are equivalent and interchangeable.

Example:

```
        ORG     $1000

        BRA     NEXT    ; Unconditional branch to NEXT.
        CLR.L   D1      ; This instruction is not executed.

NEXT    MOVE.L  #5,D0
        RTS
```

The `BRA` instruction can be replaced by the `JMP` instruction:

```
        ORG     $1000

        JMP     NEXT    ; Unconditional branch to NEXT.
        CLR.L   D1      ; This instruction is not executed.

NEXT    MOVE.L  #5,D0
        RTS
```

2. Conditional Branch Instructions

The way in which a conditional branch instruction is executed depends on one condition. These are the two alternatives:

- If the condition is satisfied, the branch is taken.
- If the condition is not satisfied, the branch is not taken.

Whether or not the condition is satisfied depends on the value of one or several condition codes.

These instructions can be found in the manual. Refer to the `Bcc` instruction description (Branch condition code). The two characters ‘cc’ must be replaced by the required condition (e.g. `BNE`, `BEQ`, `BGE`, etc.).

2.1. One-Flag Comparison Branch Instructions

Mnemonic	Condition	Branch if
BPL	Plus	N = 0
BMI	Minus	N = 1
BNE	Not Equal	Z = 0
BEQ	Equal	Z = 1
BVC	Overflow Clear	V = 0
BVS	Overflow Set	V = 1
BCC	Carry Clear	C = 0
BCS	Carry Set	C = 1

The `TST` instruction, which modifies the **N** and **Z** flags, is commonly used with the `BPL`, `BMI`, `BNE` and `BEQ` instructions.

Examples:

```
        TST.L   D1
        BEQ     NEXT1   ; Branch if D1.L = 0 (if Z = 1)

        TST.W   D2
        BNE     NEXT2   ; Branch if D2.W ≠ 0 (if Z = 0)

        TST.B   D3
        BMI     NEXT3   ; Branch if D3.B < 0 (if N = 1)

        TST.L   D4
        BPL     NEXT4   ; Branch if D4.L ≥ 0 (if N = 0)

        TST.B   D5
        BMI     NEXT5   ; Branch if D5.B < 0 (if N = 1)
        BEQ     NEXT6   ; Branch if D5.B = 0 (if Z = 1)
```

2.2. Unsigned and Signed Comparison Branch Instructions

Unsigned Comparison		Signed Comparison		Branch if the result of an operation is
Mnemonic	Condition	Mnemonic	Condition	
BHI	Higher	BGT	Greater Than	greater than 0
BHS	Higher or Same	BGE	Greater or Equal	greater than or equal to 0
BLO	Lower	BLT	Less Than	less than 0
BLS	Lower or Same	BLE	Less or Equal	less than or equal to 0

Notes:

- `BHS` is an alias for the `BCC` instruction (these two instructions are the same).
- `BLO` is an alias for the `BCS` instruction (these two instructions are the same).

These conditional branch instructions are commonly used with the `CMP` instruction in the following way:

```
        CMP    source,destination
        Bcc    <label>              ; Branch if destination <condition> source
```

Examples:

```
        CMP.L  D0,D1
        BHI    NEXT      ; Branch if D1.L > D0.L (unsigned comparison)
```

```
        CMP.W  D0,D1
        BGT    NEXT      ; Branch if D1.W > D0.W (signed comparison)
```

```
        CMP.B  D0,D1
        BLS    NEXT      ; Branch if D1.B ≤ D0.B (unsigned comparison)
```

```
        CMP.L  D0,D1
        BLE    NEXT      ; Branch if D1.L ≤ D0.L (signed comparison)
```

3. Loop Examples

The following loop structure is commonly used in the 68000 assembly language:

```
        MOVE.W  #n,D7        ; n → D7
                             ; (D7.W = loop counter)

LOOP    ; ...                ; Body of loop
        ; ...                ; repeated n times

        SUBQ.W  #1,D7        ; D7.W - 1 → D7.W (if D7.W = 0, Z is set to 1)
        BNE     LOOP         ; Branch if D7.W ≠ 0 (if Z = 0)
```

Example of a typical ‘for loop’ in C and assembly languages:

C language:

```c
#define MAX 17

int i;

for (i = 0; i < MAX; i++)
{
    /* Body of loop */
}
```

Assembly language:

```
MAX     EQU     17          ; Set the value of the label MAX.

        CLR.L   D0          ; 0 → D0
        MOVE.L  #MAX,D7     ; MAX → D7

LOOP    CMP.L   D7,D0
        BGE     DONE        ; Branch to DONE if D0 ≥ D7

        ; ...               ; Body of loop
        ; ...               ; repeated as long as D0 < D7

        ADDQ.L  #1,D0       ; D0 + 1 → D0
        BRA     LOOP        ; Branch to LOOP
DONE
```

4. The DBRA Instruction

The DBRA instruction is commonly used in loop structures. It decrements the low-order **16 bits** of a data register by one, and branches if this value is different from –1.

Example:

```
        MOVE.W  #n,D7       ; n → D7
                            ; (D7.W = loop counter)

LOOP    ; ...               ; Body of loop
        ; ...               ; repeated n + 1 times

        DBRA    D7,LOOP     ; D7.W - 1 → D7.W ; Branch to LOOP if D7.W ≠ -1
```

Note:
The DBRA mnemonic is in fact an alias for the DBF instruction. Refer to the DBcc instruction description.

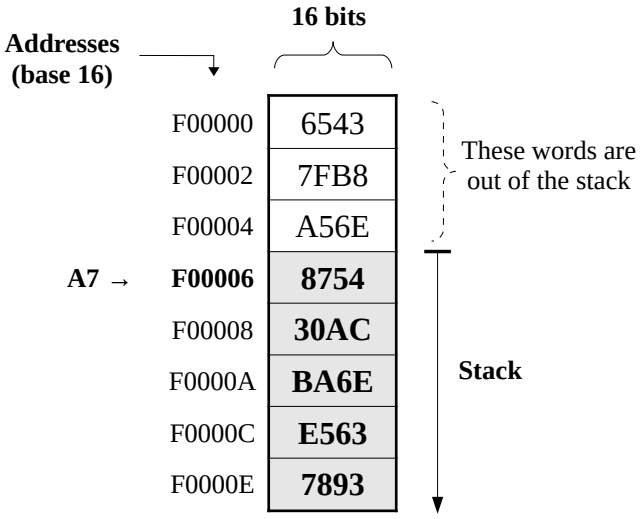
VI. The Stack

1. Definitions and Principle

The stack is a particular memory space where temporary data can be stored. It is a last-in-first-out queue (LIFO); in other words, new items are pushed onto the top of the stack, and an item is popped off the top of the stack to be retrieved (e.g. as in a stack of plates).

The register **A7** is the stack pointer: it points to **the top** of the stack.

Example for **A7** = $F00006:



Only 16-bit and 32-bit words can be pushed onto or popped off the stack (no bytes).

The memory space reserved for the stack must be large enough to satisfy the needs of the running program. Therefore, the programmer or the operating system should initialize the stack pointer accordingly.

In this chapter, we will ignore the exact value of the stack pointer. In itself, this value is irrelevant and does not contribute to the understanding of stack mechanisms. Consequently, we will only specify the variations of the stack pointer (increments and decrements) assuming that it points to a large enough memory space.

Two steps are usually needed to push an item onto the stack or to pop it off:

Step	Push Operation	Pop Operation
1	Decrement **A7**	Read the item from (**A7**)
2	Write the item to (**A7**)	Increment **A7**

Let us consider the program below; the instructions modifying the stack are numbered from 1 to 8:

```
     MOVE.L  #$11112222,D0  ; #$11112222 → D0.L
     MOVE.L  #$AAAABBBB,D1  ; #$AAAABBBB → D1.L

(1)  MOVE.W  #$42,-(A7)     ; Push the data #$0042
(2)  MOVE.L  #$53,-(A7)     ; Push the data #$00000053
(3)  MOVE.W  D0,-(A7)       ; Push the data #$2222 (the value of D0.W)
(4)  MOVE.L  D1,-(A7)       ; Push the data #$AAAABBBB (the value of D1.L)

     CLR.W   D0             ; 0 → D0.W ; D0.W is modified.
     CLR.L   D1             ; 0 → D1.L ; D1.L is modified.

(5)  MOVE.L  (A7)+,D1       ; Pop the data #$AAAABBBB and store it in D1.L
(6)  MOVE.W  (A7)+,D0       ; Pop the data #$2222 and store it in D0.W
(7)  MOVE.L  (A7)+,D2       ; Pop the data #$00000053 and store it in D2.L
(8)  MOVE.W  (A7)+,D3       ; Pop the data #$0042 and store it in D3.W
```

The different states of the stack are as follows:

Contents of the stack before instruction 1

	2FA8
	3642
	EB89
	87AF
	2118
	11BC
	C948
	72C8
A7 →	**AF3D**
	CF4C
	7893

Contents of the stack after instruction 1

	2FA8
	3642
	EB89
	87AF
	2118
	11BC
	C948
A7 →	**0042**
	AF3D
	CF4C
	7893

Contents of the stack after instruction 2

	2FA8
	3642
	EB89
	87AF
	2118
A7 →	**0000**
	0053
	0042
	AF3D
	CF4C
	7893

Contents of the stack after instruction 3

	2FA8
	3642
	EB89
	87AF
A7 →	**2222**
	0000
	0053
	0042
	AF3D
	CF4C
	7893

Contents of the stack after instruction 4

	2FA8
	3642
A7 →	**AAAA**
	BBBB
	2222
	0000
	0053
	0042
	AF3D
	CF4C
	7893

Contents of the stack after instruction 5

	2FA8
	3642
	AAAA
	BBBB
A7 →	**2222**
	0000
	0053
	0042
	AF3D
	CF4C
	7893

Contents of the stack after instruction 6

	2FA8
	3642
	AAAA
	BBBB
	2222
A7 →	**0000**
	0053
	0042
	AF3D
	CF4C
	7893

Contents of the stack after instruction 7

	2FA8
	3642
	AAAA
	BBBB
	2222
	0000
	0053
A7 →	**0042**
	AF3D
	CF4C
	7893

Contents of the stack after instruction 8

	2FA8
	3642
	AAAA
	BBBB
	2222
	0000
	0053
	0042
A7 →	**AF3D**
	CF4C
	7893

2. The MOVEM Instruction

The `MOVEM` instruction (Move Multiple Registers) can access several registers at the same time. It is commonly used to push several registers onto the stack or to pop them off with one instruction only.

Its syntax is as follows:
- `MOVEM <list>,<ea>`
- `MOVEM <ea>,<list>`

The `<list>` operand is a register list. **Only the `MOVEM` instruction can deal with a register list**. The `<ea>` operand (the effective address) must specify a memory location (‘indirect’ or ‘absolute’ addressing modes).

A register list can hold any data or address registers. The registers must be separated by the character ‘/’ (e.g. `D0/D1/A6/A4/D5`).

When the data or address registers follow one another, they can be separated by the character ‘`-`’ in order to specify a group of registers. For example, the two following lists are equivalent:
- `D0/D1/D2/D3/A2/A3/A4/A5/A6`
- `D0-D3/A2-A6`

The order of the registers in a list is irrelevant. For example, the three following lists are equivalent:
- `D6/A4/A2/A5/D1/A6/D2/D3`
- `D1/D2/D3/D6/A2/A4/A5/A6`
- `D1-D3/D6/A2/A4-A6`

The `MOVEM` instruction accesses the registers by using two predefined orders:
- The standard order: **D0**, **D1**, **D2**, **D3**, **D4**, **D5**, **D6**, **D7**, **A0**, **A1**, **A2**, **A3**, **A4**, **A5**, **A6**, **A7**
- The reverse order: **A7**, **A6**, **A5**, **A4**, **A3**, **A2**, **A1**, **A0**, **D7**, **D6**, **D5**, **D4**, **D3**, **D2**, **D1**, **D0**

The 68000 uses the reverse order only when the addressing mode of the destination operand is the ‘address register indirect with predecrement mode’: –(An). Otherwise, the standard order is used.

For instance:
- The instruction `MOVEM.L D0/A4,(A0)` accesses the register **D0**, then the register **A4**.
- The instruction `MOVEM.L D0/A4,-(A0)` accesses the register **A4**, then the register **D0**.

The `MOVEM` instruction is commonly used to push several registers onto the stack or to pop them off with one instruction only.

For instance, the following instruction:

```
MOVEM.L D1-D3/A4/A5,-(A7)  ; Push A5, then A4, then D3, then D2, then D1.
```

is equivalent to the five instructions below:

```
MOVE.L  A5,-(A7)           ; Push A5
MOVE.L  A4,-(A7)           ; Push A4
MOVE.L  D3,-(A7)           ; Push D3
MOVE.L  D2,-(A7)           ; Push D2
MOVE.L  D1,-(A7)           ; Push D1
```

And the following instruction:

```
MOVEM.L (A7)+,D1-D3/A4/A5  ; Pop D1, then D2, then D3, then A4, then A5.
```

is equivalent to the five instructions below:

```
MOVE.L  (A7)+,D1           ; Pop D1
MOVE.L  (A7)+,D2           ; Pop D2
MOVE.L  (A7)+,D3           ; Pop D3
MOVE.L  (A7)+,A4           ; Pop A4
MOVE.L  (A7)+,A5           ; Pop A5
```

VII. Subroutines

A subroutine is a fairly independent piece of code that performs a specific task. It is similar to a function in high-level languages (e.g. the C language). A subroutine can be called from the main part of a program, from another subroutine, or from itself. In the latter case, the subroutine is said to be recursive.

The 68000 has three instructions relative to subroutines:

BSR	Branch to subroutine
JSR	Jump to subroutine
RTS	Return from subroutine

The differences between the `BSR` and `JSR` instructions are the same as those between the `BRA` and `JMP` instructions. These two instructions are similar. Nonetheless, they have slight differences in terms of addressing modes and machine code, but this point will not be touched on here. So we will assume that when the operand is either an address or a label, these two instructions are equivalent and interchangeable.

Let us take the following example: a subroutine **`GetMin`** returns the smallest signed value between the registers **D1** and **D2**. The returned value is placed in the register **D0**. The subroutine can be called as many times as needed as shown below:

```
Main    …
        …               ; Any instructions
        …
        moveq.l #15,d1  ; 15 → D1
        moveq.l #25,d2  ; 25 → D2
        jsr     GetMin  ; Call GetMin (15 → D0)
Next1   …
        …               ; Any instructions
        …
        moveq.l #30,d1  ; 30 → D1
        moveq.l #16,d2  ; 16 → D2
        jsr     GetMin  ; Call GetMin (16 → D0)
Next2   …
        …               ; Any instructions
        …
GetMin  cmp.l   d1,d2   ; Compare D1 and D2.
        ble     d2min   ; Branch to d2min if D2 ≤ D1 (signed comparison).

d1min   move.l  d1,d0   ; D1 < D2, therefore D1 → D0.
        rts             ; Return from subroutine.

d2min   move.l  d2,d0   ; D2 ≤ D1, therefore D2 → D0.
        rts             ; Return from subroutine.
```

This program loads values into the registers **D1** and **D2** before calling the subroutine **`GetMin`**. The subroutine call is performed by the instruction `JSR GetMin` (`BSR GetMin` would be the same).

The instructions of the subroutine are executed until an RTS instruction is encountered. Then, the processor returns to the calling program (i.e. just after the calling point):

- In the first case, the `RTS` instruction returns to `Next1` (`Next1` is the return address).
- In the second case, the `RTS` instruction returns to `Next2` (`Next2` is the return address).

Therefore, the question which must be considered is:
How does the `RTS` instruction know the return address?

The answer is simple:
The return address is pushed onto the stack before the processor jumps to the subroutine.

Both `JSR` and `BSR` instructions perform the two following operations:

- They push the return address onto the top of the stack. The return address is the address of the instruction following either the `JSR` or the `BSR` instruction. This address is always 32 bits wide.
- They jump to the subroutine.

In order to return from a subroutine, an `RTS` instruction pops the return address off the top of the stack and loads it into the program counter (**PC**). In other words, the `RTS` instruction is a branch instruction whose destination address is at the top of the stack.

VIII. The Main Instructions of the 68000

This part gives an overview of the main instructions of the 68000. The instructions, accompanied by a brief description, are classified according to function. A complete description of each instruction can be found in the 'M68000 Family Programmer's Reference Manual'.

Most instructions have several declensions. For instance, four suffixes are commonly used: **A** (Address), **I** (Immediate), **Q** (Quick) and **X** (Extended).

- Instructions suffixed with **A** (Address) are designed to handle address registers.
- Instructions suffixed with **I** (Immediate) use immediate data as a source operand.
- Instructions suffixed with **Q** (Quick) use limited immediate data as a source operand. This limitation allows those instructions to be executed more quickly.
- Instructions suffixed with **X** (Extended) use the **X** flag as an additional source operand.

1. Data Movement Instructions

These instructions copy the contents of a source operand into a destination operand:
source → destination

MOVE	Move data from source to destination
MOVEA	Move address
MOVEQ	Move quick
MOVEM	Move multiple registers
LEA	Load effective address

2. Integer Arithmetic Instructions

2.1. Addition

These instructions add a source operand to a destination operand:
source + destination → destination

ADD	Add binary
ADDA	Add address
ADDI	Add immediate
ADDQ	Add quick
ADDX	Add with extend (source + destination + **X** → destination)

2.2. Subtraction

These instructions subtract a source operand from a destination operand:
destination – source → destination

SUB	Subtract binary
SUBA	Subtract address
SUBI	Subtract immediate
SUBQ	Subtract quick
SUBX	Subtract with extend (destination – source – **X** → destination)

2.3. Multiplication

These instructions multiply a source operand by a destination operand:
source × destination → destination

MULS	Signed multiply
MULU	Unsigned multiply

2.4. Division

These instructions divide a destination operand by a source operand:
destination ÷ source → destination

DIVS	Signed divide
DIVU	Unsigned divide

Notes:
- The quotient is loaded into the lower word of the destination operand (the 16 least significant bits).
- The remainder is loaded into the upper word of the destination operand (the 16 most significant bits).

2.5. Other

CLR	Clear an operand (0 → destination)
EXT	Sign extend (sign-extended destination → destination)
NEG	Negate (0 – destination → destination)
NEGX	Negate with extend (0 – destination – **X** → destination)

3. Boolean Instructions

AND	Logical AND
ANDI	Logical AND immediate
OR	Logical OR
ORI	Logical OR immediate
EOR	Logical EXCLUSIVE OR
EORI	Logical EXCLUSIVE OR immediate
NOT	Logical complement

4. Shift and Rotate Instructions

ASL	Arithmetic shift left
ASR	Arithmetic shift right
LSL	Logical shift left
LSR	Logical shift right
ROL	Rotate left
ROR	Rotate right
ROXL	Rotate with extend left
ROXR	Rotate with extend right
SWAP	Swap register halves

5. Bit Manipulation Instructions

BCLR	Test bit and clear
BSET	Test bit and set
BCHG	Test bit and change
BTST	Test bit

6. Test and Comparison Instructions

TST	Test operand
CMP	Compare
CMPA	Compare address
CMPI	Compare immediate
CMPM	Compare memory to memory

The test and comparison instructions do not modify the destination operand; only the condition codes (flags of the **CCR**) are affected.

The `TST` instruction updates the **N** and **Z** flags according to the value of its operand.

For instance, the instruction `TST.L D0` will set:

- **Z** to 0 if **D0** ≠ 0
- **Z** to 1 if **D0** = 0
- **N** to 0 if **D0** ≥ 0
- **N** to 1 if **D0** < 0

A comparison instruction subtracts a source operand from a destination operand (Dst - Src). The **N**, **Z**, **V** and **C** flags are updated according to the result. However, unlike a subtraction instruction, **the result is not stored in the destination operand** (the result is lost).

For instance, the instruction `CMP.L D0,D1` performs the subtraction **D1 – D0** and updates the condition codes according to the result. The register **D1** is not affected.