Practical 1
First Steps in 68000 Assembly Language

Approximate duration: 2 hrs.

Prerequisite: students are required to have read the lecture notes from page 1 to page 13.

The aim of this practical is to familiarize yourself with the different steps in assembling 68000 programs. First, you will study simple programs; then you will be able to design, assemble and debug your own.

Step 1

Let us start with some very simple instructions in order to get the basics.

- **Open a new file** in the text editor.
- **Save it** under the name "`Step_01.asm`".
- **Type the text below:**

```
            org     $4

Vector_001  dc.l    Main

            org     $500

Main        move.b  #$41,d0
            move.w  #65535,d1
            move.l  #-5,d2
```

- Let us begin by understanding each of these lines.

  The `ORG` assembler directive can be found on the first line. This directive specifies the address where the next lines will be assembled (address `$4` in our case).

  The 68000 uses the first 1,024 instructions to assign exception vectors. There are 256 vectors and all of them are 32 bits wide. They are stored in the memory from the address `$0` to the address `$3FF`. Exception handling will not be touched on here. Therefore, this part of memory should not be accessed.

  However, there is at least one vector we will need to use: the **vector 1**. It is located at the address `$4` and holds the entry point of the program, which will be loaded into the **PC** register after the 68000 has been reset. In other words, the 68000 initializes the **PC** register with the contents of the address $4 and starts to execute the program pointed at by **PC**.

The first thing to do in a source code is to specify the entry point of a program. That is the reason why our program starts with the `ORG $4` directive followed by the `DC.L Main` directive. The latter loads the `Main` address, which is the entry point of our program, into the memory location `$4`.

These first two lines have to be present in all of your source codes in order to initialize properly the 68000.

It is noteworthy that the `Vector_001` label is optional. It is specified for information only.

Let us go on with the main program. The `ORG $500` directive tells us that it will be located at the address `$500`.

The main program is made up of several `MOVE` instructions, which are used to initialize some data registers with immediate data. Let us point out that the `Main` label is placed just before the first instruction to execute.

- You can now assemble the program. **Press `[F9]` under Windows or `[F8]` under Linux.**

  The assembly result is logged to the window at the bottom of the screen. If an error occurs, the line numbers will be shown here. You will then be able to correct your source code.

  If no errors are found, the following line should be displayed:
  `End of assembly - no errors were found.`

- You can now run the program. **Press `[F10]` under Windows or `[F5]` under Linux.**

  A new window appears in the middle of the screen: it is the **68000 debugger (d68k)**. It allows you to debug all your programs down to the smallest detail.

  Its main window shows your assembly program in a slightly different way: assembler directives and labels have disappeared; they have been replaced by addresses and machine code.

  The `ORI` instructions that can be seen do not belong to your program. Actually, the contents of memory are reset to zero and the `ORI.B #0,D0` instruction matches the machine code `0000 0000`.

The lowest part of the window displays all the 68000 registers and a small part of memory.

The highest part of the window, which is a gray zone, says that no instructions have been executed yet. As a first step, we will ignore this zone.

- You can now execute your first instruction. **Press `[F11]`.**

  The **D0** register has just been modified: it is displayed in red. Each register is displayed in its 32-bit hexadecimal form. So, the displayed value is `$00000041`.

  The ASCII form of **D0** is displayed to the right of its hexadecimal form. Note that the value `$41` is the ASCII code of the `'A'` character.

- **Press `[F11]`** again in order to execute the next instruction.

  This time, the register in red is **D1**. The loaded value is 65,535 and its 32-bit hexadecimal representation is `$0000FFFF`.

  Any register can be displayed in the most commonly used representations (decimal, hexadecimal, binary, signed, unsigned, etc.).

- **For instance, click on the `[D1]` button.**

  A new window appears and **D1** is converted into each of the most commonly used forms.

  This window can also be used as a conversion calculator. For example, type the value `-1` then click on the button **`[=]`**. Try the following expression as well: `$40+D0` and `(5+%101)*4`

- The value of the register can be modified. For instance, **type the value 255 and click on the `[OK]` button.**

  D1 has been modified and its new value is `$000000FF`.

- **Press `[F11]`** again in order to execute the next instruction.

  This time, **D2** is modified. Be careful, the value `-5` is displayed in its 32-bit hexadecimal form. To see its decimal representation, just click on the **`[D2]`** button.

- **Close the debugger.**

Step 2

Let us go on with a program that adds the contents of two 8-bit registers.

- **Open a new file** in the text editor.
- **Save it** under the name "`Step_02.asm`".
- **Type the text below:**

```
            org     $4

Vector_001  dc.l    Main

            org     $500

Main        move.b  #18,d0  ; 18 -> D0.B
            move.b  #12,d1  ; 12 -> D1.B
            add.b   d0,d1   ; D0.B + D1.B -> D1.B (12 + 18 = 30).
```

- **Assemble, correct your mistakes if any, then launch the debugger.**

- **Run your program step by step by pressing `[F11]`.**

- Once the last instruction has been executed, make sure that the value of **D1** is 30_{10}. To do so, you can click on the **`[D1]`** button.

- Now, let us consider the gray zone at the top of the window, which displays the last instructions that have been executed by the 68000.

  The debugger allows you to undo these instructions.

- **Press `[Back Space]`.**

  The 68000 and the memory go back to the state they were before executing the `ADD` instruction. **D1** goes back to 12.

- **Press `[Back Space]` twice.**

  We have gone back to the beginning.

- So far, we have always executed the instructions step by step, which is the “step into” mode.

  However, the emulator can execute instructions successively. Breakpoints are then required.

- You can place an **address breakpoint**.

  To do so, **click on the border to the left of the instruction** (located at the address `$00050A`) **that follows your program.** A red background denotes the address breakpoint.

  You can now run your program up to this breakpoint.

- **Press `[F9]`.**

  Be careful! If you do not place the breakpoint, you will have to press **`[Escape]`** to stop the emulator.

  As you can see, the three instructions have been executed up to the breakpoint.

- The 68000 can be reset by pressing **`[Ctrl+R]`**.

  It is noteworthy that the registers have not been reset to zero. The 68000 never resets its data and address registers when it is reset. Therefore, you should never assume that a register is set to zero when the microprocessor is switched on. It is the emulator, not the 68000, that sets the registers to zero when the debugger is launched.

- **Close the debugger.**

  You can also place an **instruction breakpoint**.

- **Add the `ILLEGAL` instruction after the `ADD` instruction.**

```
            org     $4

Vector_001  dc.l    Main

            org     $500

Main        move.b  #18,d0  ; 18 -> D0.B
            move.b  #12,d1  ; 12 -> D1.B
            add.b   d0,d1   ; D0.B + D1.B -> D1.B (12 + 18 = 30).

            illegal         ; Instruction breakpoint.
```

- **Assemble and launch the debugger.**

  You can now run your program up to this breakpoint.

- **Press `[F9]`.**

The emulator stops at the `ILLEGAL` instruction. Feel free to use this instruction in order to stop a program. Note that a blue background denotes an instruction breakpoint.

> **Be careful!** You have to be aware that for the 68000 the **`ILLEGAL instruction is not a breakpoint`**. This instruction is merely redirected by the debugger so that you can use it as a breakpoint.

- **Close the debugger.**

Step 3

In this part, we are going to add the contents of two 16-bit memory cells.

- **Open a new file** in the text editor.
- **Save it** under the name "`Step_03.asm`".
- **Type the text below:**

```
            org     $4

Vector_001  dc.l    Main

            org     $500

Main        move.w  NUMBER1,d0  ; (NUMBER1) -> D0.W
            add.w   NUMBER2,d0  ; (NUMBER2) + D0.W -> D0.W
            move.w  d0,SUM      ; D0.W -> (SUM)

            illegal

            org     $550

NUMBER1     dc.w    $2222       ; The number $2222 is in the address NUMBER1.
NUMBER2     dc.w    $5555       ; The number $5555 is in the address NUMBER2.
SUM         ds.w    1           ; Reserve 16 bits to store the sum.
```

This time, the numbers to add are no longer in registers but in memory. First of all, let us determine where these numbers are located.

Thanks to the `ORG $550` directive, we can easily deduce that the first number is located at the address `$550`. Since the size of the `DC` assembler directive is 16 bits, the address of the second number is `$552`. The sum will then be stored in the address `$554`. In other words, the labels will be assigned the following values:

- ➔ `NUMBER1 = $550`
- ➔ `NUMBER2 = $552`
- ➔ `SUM     = $554`

- **Assemble and launch the debugger.**

- You can display the contents of memory in the hexadecimal and ASCII forms. **Click on the tab labeled `[Mémoire]`.**

  The contents of memory are displayed from the address `$500`. The first bytes are the machine code of your program.

  Identify where the address `$550` is located and check that the two numbers `$2222` and `$5555` are in the right place. Identify the memory location where the sum will be stored once the program has been executed.

  It is noteworthy that a tooltip is displayed when the mouse pointer is moving over a byte. This tooltip specifies the address where the byte is located as well as its hexadecimal, decimal and ASCII forms.

- **Click on the tab labeled `[Désa]`** in order to go back to the disassembly view.

  Before executing the program, note the small yellow arrow on the left (only on Linux version). It points to the next instruction to execute, which can also be seen to the right of the **`[PC]`** button.

- **Press `[F11]`**.

  The number `$2222` is loaded into **D0**.

- **Press `[F11]`**.

  The number `$5555` is added to **D0**. Then, the new value of **D0** is:
  `$2222` + `$5555` = `$7777`

- **Press `[F11]`**.

  The sum is loaded into memory at the address `$554`.

- **Click on the tab labeled `[Mémoire]`** and check that the sum is in the right address.

- **Close the debugger.**

Step 4

In this step, we will work out the sum of five numbers contained in a tab. Each number is an 8-bit integer. We assume that the sum will never be greater than 255.

- **Open a new file** in the text editor.
- **Save it** under the name `"Step_04.asm"`.
- **Type the text below:**

```
            org     $4

Vector_001  dc.l    Main

            org     $500

Main        movea.l #TAB,a0     ; Initialize A0 with the address of the tab.

            move.b  (a0)+,d0    ; Number 1        -> D0.B ; A0 + 1 -> A0
            add.b   (a0)+,d0    ; Number 2 + D0.B -> D0.B ; A0 + 1 -> A0
            add.b   (a0)+,d0    ; Number 3 + D0.B -> D0.B ; A0 + 1 -> A0
            add.b   (a0)+,d0    ; Number 4 + D0.B -> D0.B ; A0 + 1 -> A0
            add.b   (a0),d0     ; Number 5 + D0.B -> D0.B

            move.b  d0,SUM      ; D0.B -> (SUM)

            illegal

            org     $550

TAB         dc.b    18,3,5,9,14 ; Tab containing the 5 numbers.
SUM         ds.b    1           ; Reserve 8 bits to store the sum.
```

- **Assemble and launch the debugger.**

- **Press `[F11]`.**

  The **A0** register is initialized with the start address of the tab. Its new address, which is displayed in red, is `$550`.

  The debugger displays the contents of memory from the address of the register (in hexadecimal and ASCII forms). Therefore, the five numbers to add can be seen to the right of the hexadecimal value of the register.

- **Press `[F11]`.**

  The first number to add is loaded into **D0**.

  The **A0** register has been incremented. The contents of memory – to the right of the hexadecimal value of the register – are displayed from the address `$551`. In other words, the first displayed number is the second number of the tab.

- Finish executing the program (press **[F11]** several times or **[F9]** only once).

- Check that the sum stored in the address `SUM` (`$555`) is right (use the **[Mémoire]** tab).

- **Close the debugger.**

- Data in tabs can also be accessed by using the address register indirect with displacement mode: d16(An).

```
            org     $4

Vector_001  dc.l    Main

            org     $500

Main        movea.l #TAB,a0     ; Initialize A0 with the address of the tab.

            move.b  (a0),d0     ; Number 1         -> D0.B
            add.b   1(a0),d0    ; Number 2 + D0.B -> D0.B
            add.b   2(a0),d0    ; Number 3 + D0.B -> D0.B
            add.b   3(a0),d0    ; Number 4 + D0.B -> D0.B
            add.b   4(a0),d0    ; Number 5 + D0.B -> D0.B

            move.b  d0,SUM      ; D0.B -> (SUM)

            illegal

            org     $550

TAB         dc.b    18,3,5,9,14 ; Tab containing the 5 numbers.
SUM         ds.b    1           ; Reserve 8 bits to store the sum.
```

Step 5

1. Without using the assembler and the debugger, determine the result of the following additions as well as the values of the **N**, **Z**, **V** and **C** flags.
   - 8-bit addition: `$B4 + $4C`
   - 16-bit addition: `$B4 + $4C`
   - 16-bit addition: `$4AC9 + $D841`
   - 32-bit addition: `$FFFFFFFF + $00000015`

2. Use the debugger to check your answers. To do so, write a program that performs the four additions above. Assemble it, run it, check the results and the values of the flag.

Step 6

Write a program that performs a 128-bit addition.

Inputs : **D3:D2:D1:D0** = 128-bit integer (**D0** contains the 32 least significant bits).
D7:D6:D5:D4 = 128-bit integer (**D4** contains the 32 least significant bits).

Output : **D3:D2:D1:D0** = **D3:D2:D1:D0** + **D7:D6:D5:D4**

Use the `ADD` and `ADDX` instructions only.

Step 7

Write a few rotate instructions that modify **D1** so that it takes the values below. For each case, the initial value of **D1** is `$76543210`.

- **D1** = `$76543120`
- **D1** = `$75640213`
- **D1** = `$54231067`
- **D1** = `$05634127`

Use the `ROL`, `ROR` and `SWAP` instructions only.