# Emulation of the MIPS MSA ISA using Docker and QEMU MIPS64

In this assignment, we are going to emulate the operation of the multimedia extension of MIPS64 (called MIPS MSA) using the QEMU MIPS64 emulator. QEMU is a popular virtual machine emulator that uses cross-compilation. In this case, we will use it to emulate the execution of programs on an architecture different from our native machine: MIPS64.

## 1. Installation of the tools

Follow the instructions below to install the MIPS64 cross-compiler and the QEMU MIPS64 emulator. This step-by-step process is verified to work on a Windows 11 machine.

1. Install Docker Desktop and launch the daemon.
2. Copy-paste the provided files `ex1.c` and `ex2.c` onto a directory WITHOUT SPACES, run the Windows command line (`cmd`), go to that directory and run an Ubuntu 20.04 LTS (focal) container like this:

```
docker run -it -v .:/home -w /home ubuntu:focal /bin/bash
```

3. Install the necessary packets:

```
root@74e63dd00e90:/# apt update
root@74e63dd00e90:/# apt -y upgrade
root@74e63dd00e90:/# apt -y install gcc-10-mipsisa64r6-linux-gnuabi64
root@74e63dd00e90:/# apt -y install qemu-user
```

4. If everything worked, you will see that your working Linux directory will be bound to the local Windows directory where the command in Step 2 was launched. Any file or directory modified/edited/deleted from Windows will be visible in the Linux machine, and vice-versa.
5. From now on, you can use your preferred text editor to edit `ex1.c` and/or `ex2.c` from Windows and the changes made will be visible in the Linux machine. Additionally, you will be able to compile and execute the code from the Linux machine.

## 2. Example: Solution of exercise 1 (Appendix)

The file `ex1.c` (described in the Appendix of this document) is provided as an example to perform the compilation process in 3 versions of this exercise, using the C `#DEFINE` directive and the `ADDVI_VERSION` and `MSA_VERSION` macros.

a. A scalar version (the code is compiled with no C macro defined).

b. A MIPS MSA version using the `addvi.b`[1] instruction (the code is compiled if the `ADDVI_VERSION` macro is defined).
c. Another MIPS MSA version, this time using the `adds_u.b`[2] instruction (the code is compiled if the `ADDS_VERSION` macro is defined).

### a. Scalar version

Compile the first version of the program (scalar, without macros) using the `mipsisa64r6-linux-gnuabi64-gcc-10` compiler. An executable file, `ex1-scalar.elf`, will be generated.

```
root@74e63dd00e90:/# mipsisa64r6-linux-gnuabi64-gcc-10 -mmsa ex1.c
-o ex1-scalar.elf -static
```

Run it with the `qemu-mips64` emulator and save the reference output (file `ex1-scalar.out`). This output will be used to verify the correct result of the other versions b and c.

```
root@74e63dd00e90:/# qemu-mips64 -cpu I6400 ex1-scalar.elf > ex1-
scalar.out
```

The code that is compiled in this case is the following:

```
for (i=0;i<16384;i++) {
   if (red[i]< 250)
      red[i] += 5;
   else
      red[i] = 255;
}
```

### b. Version with MIPS MSA (using addvi.b)

Compile the second version of the program (i.e., the one that uses MIPS MSA and the `addvi.b` instruction) using *inline assembly* (link1, link2 GNU GCC Online Docs)

```
root@74e63dd00e90:/# mipsisa64r6-linux-gnuabi64-gcc-10 -mmsa ex1.c
-o ex1-addvi.elf -static \
-DADDVI_VERSION
```

The code that is compiled in this case is the following:

```
__asm volatile(
"                       li          $4,0x4000   \n"
```

---

[1] This instruction performs a normal addition of two elements using the MIPS MSA multimedia extension, so it is necessary to use the `bsel` instruction afterward to achieve saturated addition in the 'if-then-else' statement (original code).

[2] This instruction performs a 'saturated add', having the same effect as the 'if-then-else' statement in the original code in a single instruction. Therefore, there is no need for a separate `bsel` here. You can refer to the attached document 'MIPS 64 Manual.pdf' for more information.

```
"                       move        $5,%[Red]   \n"
"                       li          $6,0x10     \n"
"                       ldi.b       $w1,0xfa    \n"
"                       ldi.b       $w2,0xff    \n"
"loop:                  ld.b        $w3,0($5)   \n"
"                       clt_u.b     $w4,$w3,$w1 \n"
"                       addvi.b     $w5,$w3,0x5 \n"
"                       bsel.v      $w4,$w2,$w5 \n"
"                       st.b        $w4,0($5)   \n"
"                       sub         $4,$4,$6    \n"
"                       dadd        $5,$5,$6    \n"
"                       bgtz        $4,loop     \n"
"                       nop                     \n"
:
: [Red] "r" (red)
: "memory","$4","$5","$6"
);
```

You will notice that there are some instructions with a syntax different from what was seen in class, but their functionality is similar.

Exercise: Run it and compare its output with the previous section.

### c. Version with MIPS MSA (using adds_u.b)

Compile the third version of the program (i.e., the one that uses MIPS MSA and the `adds_u.b` instruction), using *inline assembly* en GCC:

```
root@74e63dd00e90:/# mipsisa64r6-linux-gnuabi64-gcc-10 -mmsa ex1.c
-o ex1-adds.elf -static \
-DADDS_VERSION
```

The code that is compiled in this case is the following:

```
__asm volatile(
"                       li          $4,0x4000         \n"
"                       li          $5,0x10           \n"
"                       move        $6,%[Red]         \n"
"                       ldi.b       $w0,0x5           \n"
"loop:                  ld.b        $w1,0($6)         \n"
"                       adds_u.b    $w1,$w1,$w0       \n"
"                       st.b        $w1,0($6)         \n"
"                       sub         $4,$4,$5          \n"
"                       dadd        $6,$6,$5          \n"
"                       bgtz        $4,loop           \n"
"                       nop                           \n"
:
: [Red] "r" (red)
: "memory","$4","$5","$6"
);
```

Exercise: Run it and compare its output with the two previous sections. The outputs should be identical.

## 3. Example and assignment: Exercise 2 (Appendix)

In this case, the file `ex2.c` is provided as an example to perform the compilation process in 2 versions of this exercise, using the C `#DEFINE` directive and the `MSA_VERSION` macro.

### a. Scalar version

Compile the first version of the program (scalar, without macros), run it, and save the reference output. This output will be used to verify the correct result of version b.

```
root@74e63dd00e90:/# mipsisa64r6-linux-gnuabi64-gcc-10 -mmsa ex2.c
-o ex2-scalar.elf -static
root@74e63dd00e90:/# qemu-mips64 -cpu I6400 ex2-scalar.elf > ex2-
scalar.out
```

The code that is compiled in this case is the following:

```
for (i=0; i<256; i++) {
    a[i]=b[i]+c[i];
    if (a[i]==b[i])
       d[i]=a[i]*3;
    b[i]=a[i]-5;
}
```

### b. [EXERCISE] Version with MIPS MSA

Fill in the space between the lines `#ifdef MSA_VERSION` and `#else` in the file `ex2.c` to write a version of the previous code that implements the same algorithm but using MIPS MSA. To do this, as in the previous example, you should compile it using *inline assembly* en GCC:

```
root@74e63dd00e90:/# mipsisa64r6-linux-gnuabi64-gcc-10 -mmsa ex2.c
-o ex2-msa.elf -static \
-DMSA_VERSION
```

The initial code provided to you, which will be compiled, and which you need to complete, is as follows:

```
__asm volatile(
"                       ld.d        $w7,0(%[Rthree])  \n"
"                       ld.d        $w8,0(%[Rfive])   \n"
// YOUR CODE
:
: [Ra] "r" (a),
  [Rb] "r" (b),
  [Rc] "r" (c),
  [Rd] "r" (d),
```

```
  [Rthree] "r" (three),
  [Rfive] "r" (five)
: "memory", "$4", "$5", "$6"
);
```

In this exercise, you should use the floating-point instructions FADD.D, FCEQ.D, FMUL.D, and FSUB.D from MIPS MSA (you can see their operation in the attached document 'MIPS 64 Manual.pdf').

Finally, the line `"memory", "$4", "$5", "$6"` indicates that this function modifies the `memory`, as well as registers `$4`, `$5`, and `$6`. These are known as the 'side effects' of executing this code. If your code modifies additional registers, you need to add them to this list.

After completing the exercise and verifying its correct output, write a brief report (.pdf file, approximately 1 page) summarizing the activities performed and include screenshots of your code executions. **Assignments without this document will not be evaluated.**

## Appendix

**Exercise 1**. Let us assume an image with 128x128 pixels coded in RGB. An image processing algorithm is executed on it, in such a way that the R component of each pixel is incremented by 5, saturating its value to 255 if needed (the G and B components remain the same as before). Write a piece of assembly code, using MIPS64 SIMD instructions, functionally equivalent to the following pseudo-code:

```
for (i=0;i<16384;i++) {
   if (red[i]< 250)
      red[i] += 5;
   else
      red[i] = 255;
}
```

NOTE: For simplicity, it is assumed that the R components of all pixels of this image are stored in a consecutive way from the memory address pointed by vector C.

**Exercise 2**. Let us assume 4 vectors (A, B, C and D) with floating-point 256 components (i.e., they have 64 bits each). Write a piece of assembly code, using MIPS64 SIMD instructions, functionally equivalent to the following pseudo-code:

```
for (i=0; i<256; i++) {
    a[i]=b[i]+c[i];
    if (a[i]==b[i])
       d[i]=a[i]*3;
    b[i]=a[i]-5;
}
```