

## Laboratory 1

Expected delivery of lab\_01.zip including:

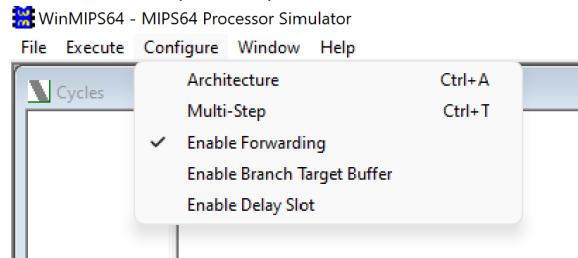
- program\_0.s
- lab\_01.pdf (fill and export this file to pdf)

Please, configure the winMIPS64 processor architecture with the *Base Configuration* provided in the following:

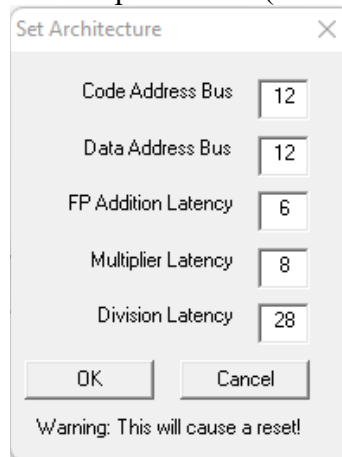
- Integer ALU: 1 clock cycle
- Data memory: 1 clock cycle
- Branch delay slot: 1 clock cycle
- Code address bus: 12
- Data address bus: 12
- Pipelined FP arithmetic unit (latency): 6 stages
- Pipelined FP multiplier unit (latency): 8 stages
- FP divider unit (latency): not pipelined unit, 28 clock cycles
- Forwarding optimization is disabled
- Branch prediction is disabled
- Branch delay slot optimization is disabled.

Use the Configure menu:

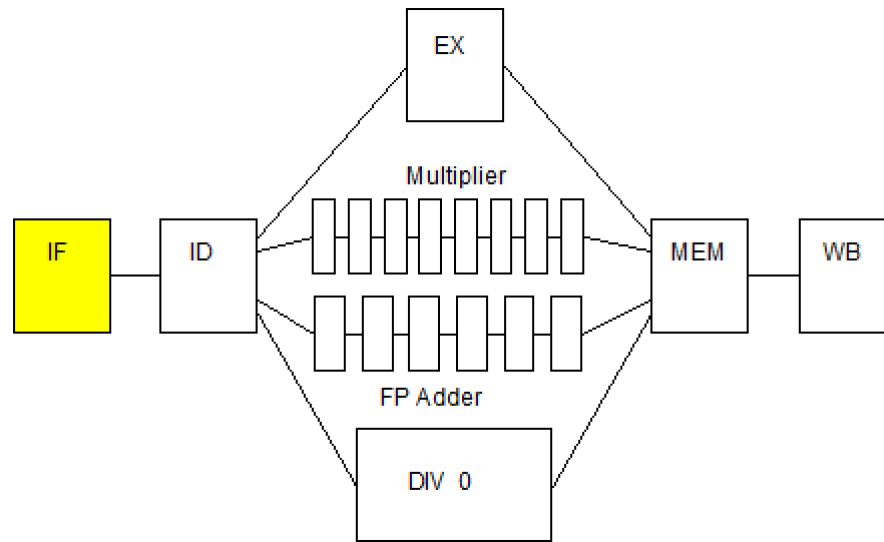
- Running the *WinMIPS* simulator, launching the graphical user interface (*folder\_to\_simulator*)...\winMIPS64\winmips64.exe
- Disable ALL the optimization (a mark appears when they are enabled)
- Browse the Architecture menu (Ctrl-A)



- Modify the defaults Architectural parameters (where needed)



- Verify in the Pipeline window that the configuration is effective (usually in the left bottom window)



1) Exercise your assembly skills.

Write and run an assembly program called **program\_0.s (to be delivered)** for the *MIPS64* architecture.

The program must:

1. Given three arrays of 9 8-bit integer numbers ( $v1, v2, v3$ ), check **for each one of them** if the content corresponds to a **palindrome** sequence of numbers. If yes, use three 8-bit unsigned variables (*flags*) to store the results. The variables will be equal to 1 if the sequence is palindrome, 0 otherwise.
2. **Only for the palindrome arrays**, compute the sum element by element and place the result in another array  $v4$  (i.e.  $v4[i] = v2[i] + v3[i]$  supposing that only  **$v2$  and  $v3$  are palindromes**)

Example of a vectors sequence containing only 9 numbers:

```

v1:      .byte      2, 6, -3, 11, 9, 11, -3, 6, 2
v2:      .byte      4, 7, -10, 3, 11, 9, 7, 6, 4, 7
v3:      .byte      9, 22, 5, -1, 9, -1, 5, 22, 9
flag1:   .space 1
flag2:   .space 1
flag3:   .space 1
v4:      .space 9

```

2) Use the *WinMIPS* simulator.

Identify and use the main components of the simulator:

- a. Running the *WinMIPS* simulator
  - Launch the graphic interface  
...\\winMIPS64\\winmips64.exe
- b. Load your program in the simulator:

- Load the program from the **File→Open** menu (*CTRL-O*). In the case the of errors, you may use the following command in the command line to compile the program and check the errors:  
`...\winMIPS64\asm program_0.s`

- Run your program step by step (*F7*), identifying the whole processor behavior in the six simulator windows:

**Pipeline, Code, Data, Register, Cycles and Statistics**

- Collect the clock cycles to fill the following table **(fill all required data in the table before exporting this file to pdf format to be delivered)**.

Table 1: Program performance for the specific processor configurations

Program	Clock cycles	Number of Instructions	Clocks per instruction CPI)	Instructions per Clock (IPC)
program_0	242	145	1669	$5.99 \cdot 10^{-4}$

### 3) Perform execution and time measurements.

Measure the processor performance by running a benchmark of programs. Change the weights of the programs as indicated in the following to evaluate how these variations may produce different performance results.

Search in the winMIPS64 folder the following benchmark programs:

- testio.s
- mult.s
- series.s
- program\_0.s (your program)

Starting from the basic configuration with no optimizations, compute by simulation the number of cycles required to execute these programs; in this initial scenario, it is assumed that the weight of the programs is the same (25%) for everyone. Assume a processor frequency of 1.75 kHz (*a very old technology node*).

Then, change processor configuration and vary the programs' weights as follows. Compute again the performance for every case and fill the table below **(fill all required data in the table before exporting this file to pdf format to be delivered)**..:

#### 1) Configuration 1

- Enable Forwarding
- Disable branch target buffer
- Disable Delay Slot

Assume that the weight of all programs is the same (25%).

#### 2) Configuration 2

- Enable Forwarding
- Enable branch target buffer
- Disable Delay Slot

Assume that the weight of all programs is the same (25%).

3) Configuration 3

Configuration 1, but assume that the weight of the program *your program* is 43.33%.

4) Configuration 4

Configuration 1, but assume that the weight of the program `series.s` is 60%.

Table 2: **Processor performance for different weighted programs**

Program	No opt	Conf. 1	Conf. 2	Conf. 3	Conf. 4
testio.s	0.412	0.271	0.246	0.271	0.271
mult.s	1.072	0.559	0.526	0.559	0.559
series.s	0.314	0.133	0.134	0.133	0.133
program 0.s	0.138	0.117	0.115	0.117	0.117
TOTAL Time (@ 1.75kHz)	0.484s	0.27s	0.255s	0.255s	0.206s

## **Appendix: winMIPS64 Instruction Set**

---

### ***WinMIPS64***

#### **The following assembler directives are supported**

.data - start of data segment  
.text - start of code segment  
.code - start of code segment (same as .text)  
.org <n> - start address  
.space <n> - leave n empty bytes  
.ascii <s> - enters zero terminated ascii string  
.ascii <s> - enter ascii string  
.align <n> - align to n-byte boundary  
.word <n1>,<n2>.. - enters word(s) of data (64-bits)  
.byte <n1>,<n2>.. - enter bytes  
.word32 <n1>,<n2>.. - enters 32 bit number(s)  
.word16 <n1>,<n2>.. - enters 16 bit number(s)  
.double <n1>,<n2>.. - enters floating-point number(s)

where <n> denotes a number like 24, <s> denotes a string like "fred", and  
<n1>,<n2>.. denotes numbers separated by commas.

#### **The following instructions are supported**

lb - load byte  
lbu - load byte unsigned  
sb - store byte  
lh - load 16-bit half-word  
lhu - load 16-bit half word unsigned  
sh - store 16-bit half-word  
lw - load 32-bit word  
lwu - load 32-bit word unsigned  
sw - store 32-bit word  
ld - load 64-bit double-word  
sd - store 64-bit double-word  
ld - load 64-bit floating-point  
sd - store 64-bit floating-point  
halt - stops the program  
  
daddi - add immediate  
daddui - add immediate unsigned  
andi - logical and immediate  
ori - logical or immediate  
xori - exclusive or immediate  
lui - load upper half of register immediate  
slti - set if less than or equal immediate  
sltiu - set if less than or equal immediate unsigned

beq - branch if pair of registers are equal  
bne - branch if pair of registers are not equal  
beqz - branch if register is equal to zero  
bnez - branch if register is not equal to zero  
  
j - jump to address  
jr - jump to address in register  
jal - jump and link to address (call subroutine)  
jalr - jump and link to address in register (call subroutine)

dsll - shift left logical  
dsrl - shift right logical  
dsra - shift right arithmetic  
dsllv - shift left logical by variable amount  
dsrlv - shift right logical by variable amount  
dsrav - shift right arithmetic by variable amount  
movz - move if register equals zero  
movn - move if register not equal to zero  
nop - no operation  
and - logical and  
or - logical or  
xor - logical xor

slt - set if less than  
sltu - set if less than unsigned  
dadd - add integers  
daddu - add integers unsigned  
dsub - subtract integers  
dsubu - subtract integers unsigned

add.d - add floating-point  
sub.d - subtract floating-point  
mul.d - multiply floating-point  
div.d - divide floating-point  
mov.d - move floating-point  
cvt.d.l - convert 64-bit integer to a double FP format  
cvt.l.d - convert double FP to a 64-bit integer format  
c.lt.d - set FP flag if less than  
c.le.d - set FP flag if less than or equal to  
c.eq.d - set FP flag if equal to  
bc1f - branch to address if FP flag is FALSE  
bc1t - branch to address if FP flag is TRUE  
mtc1 - move data from integer register to FP register  
mfc1 - move data from FP register to integer register