|  |  |
| --- | --- |
| **Architetture dei Sistemi di Elaborazione** | Delivery date:  Friday 15 October 2021 |
| **Laboratory**  **1** | Expected delivery of lab\_01.zip including:   * Program\_0.s * Program\_1.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:   * remove the flags (where activating Enable options) * Browse the Architecture menu 🡪 | Modify the defaults Architectural parameters (where needed)    🡨 Verify in the Pipeline window that the configuration is effective |
|  |

1. Exercise your assembly skills.

To write an assembly program called **program\_0.s (to be delivered**) for the *MIPS64* architecture and to execute it.

The program must:

1. Given one array (a), find how many times the 8-bit unsigned value val=0x2f is included in it. The array contains 100 unsigned 16-bit integer numbers. Looks for the occurrence in both the most significant **byte** and the least significant **byte** of the 16-bit integer numbers. Store the result in a 8-bit variable (res).

Example (with only 10 elements):

a = 5, 0x01a1,0x2f, 17, 0x2fe0, 1, 0x96a4, 0x2f, 3, 0x2f

res = 4

1. Exercise your assembly skills and use the main components of the simulator:

To write an assembly program called **program\_1.s (to be delivered**) for the *MIPS64* architecture and to execute it.

The program must:

1. Given 2 arrays (a and b), compute their signed sum and store each result in a third array (i.e., c[i] = a[i] + b[i]). Each array contains 30 8-bit integer numbers.
2. Create two variables allocated in memory: threshold\_high and threshold\_low. For each element c[i], check whether it is greater or lesser than a fixed threshold: if it is greater, increase threshold\_high, else increase threshold\_low. Assume the threshold is equal to 0x0. Note that 0x00 belongs to threshold\_high.
3. Search for **both** the maximum and minimum in the array c. The program saves the obtained value in two variables allocated in memory, called max and min respectively.

Identify and use the main components of the simulator:

* 1. Running the *WinMIPS* simulator
* Launch the graphic interface

...\winMIPS64\winmips64.exe

* 1. Assembly and check your program:
     + 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\_1.s

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

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

* 1. Repeat the process (a-b-c) for program\_0.S

Table 1: **Programs performance for the processor’s base configurations**

|  |  |
| --- | --- |
|  | Number of clock cycles |
| program\_0.S | 1496 |
| program\_1.S | 765 |

1. Perform execution time measurements.

Search in the winMIPS64 folder the following benchmark programs:

* 1. isort.s
  2. mult.s
  3. program\_0.s (your program)
  4. program\_1.s (your program)

Starting from the basic configuration with no optimizations, compute by simulation the number of cycles required to execute these programs (and then the weighted arithmetic mean). Assume a processor frequency of 5MHz. Then, vary the program weights as specified by the following Configurations. Compute the weighted arithmetic mean 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

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

1. Configuration 2

Assume that the weight of the program program\_0.s is 55%.

1. Configuration 3

Assume that the weight of the program isort.s is 40%.

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

|  |  |  |  |
| --- | --- | --- | --- |
| Program | Conf. 1 | Conf. 2 | Conf. 3 |
| isort.s |  |  |  |
| mult.s |  |  |  |
| program\_0.s |  |  |  |
| program\_1.s |  |  |  |
| TOTAL TIME (us) |  |  |  |

For time computations, use a clock frequency of 5MHz.

**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

.asciiz <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 seperated 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

l.d - load 64-bit floating-point

s.d - 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