Computer Architectures
2nd part labs – lab 1
WinMIPS64 introduction

Considering the MIPS64 architecture presented in the following:

- Integer ALU: 1 clock cycle
- Data memory: 1 clock cycle
- FP multiplier unit: pipelined 8 stages
- FP arithmetic unit: pipelined 4 stages
- FP divider unit: not pipelined unit, 12 clock cycles
- branch delay slot: 1 clock cycle
- forwarding is enabled.

1) Write an assembly program (**program_1.s**), using a text editor, for the *MIPS64* architecture able to find the maximum among 10 64-bit integer values saved in memory. The obtained value must be saved in memory using a variable called *result*.

a) Use a loop-based program for searching the maximum value.

2) Identifying main components of the simulator:
   a. Assembly and debug your program:
      - Using the command line:
        ...\winMIPS64\asm program.s
   b. Launch the WinMIPS simulator
      - Launch the graphic interface
        ...\winMIPS64\winmips64.exe
   c. Load your program
      CTRL-O (*File Open*)
   d. Disable all features present in the *Configure* menu
      1. Disable Forwarding
      2. Disable branch target buffer (*winmips64 v1.5*)
      3. Disable Delay Slot
   e. Run your program step by step (*F7*), identifying the whole processor behavior in the six simulator windows:
      Pipeline, **Code**, **Data**, **Register**, **Cycles** and **Statistics**
   f. Enable one by one the features included in the *Configuration* (see *2.d*) menu analyzing the processor behavior, and highlighting differences in the final statistics.

3) Write an assembly program (**program_2.s**) for the *winMIPS64* architecture described before able to implement the following piece of code described at high-level:

```
for (i = 1; i <= 100; i++){
        v5[i] = v1[i]*v2[i];
        v6[i] = v2[i]/v3[i];
        v7[i] = v1[i]+v4[i];
}
```

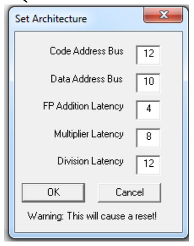
   a. Assume that the vectors v1[], v2[], v3[], and v4[] are allocated previously in memory and contains 100 double precision floating point values; assume also that v3[] does not contain 0 values. Additionally, the vectors v5[], v6[], and v7[] are free vectors also allocated in memory.

4) Calculate by hand, how many clock cycles take the program to execute?
5) Compute the same calculation using the *winMIPS64* simulator.
6) Compare the results obtained in the points 4 and 5, and provide some explanation if the results are different.
7) Considering the *branch delay slot* enabled, use the static *scheduling* technique to re-schedule the code developed in point 1 (**program_2.s**) in order to eliminate the most data hazards. Using the new program **program_3.s,** show the timing results and compare them with the previous results.
8) Starting from the previous program (**program_3.s**) and exploiting the optimization techniques called *loop-unrolling,* and if necessary *register renaming*, rewrite once again the initial program. Show the timing results of the new program (**program_4.s**) and compare the statistics values provided by the simulator for the developed programs.

9) Using the WinMIPS64 simulator, validate experimentally the Amdahl's law, defined as follows:

$$\text{speedup}_{\text{overall}} = \frac{\text{execution time}_{\text{old}}}{\text{execution time}_{\text{new}}} = \frac{1}{(1-\text{fraction}_{\text{enhanced}}) + \dfrac{\text{fraction}_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}}}$$

   i) Write an assembly program
   ii) Select one of the processor architectural parameters related with multicycle instructions (`Menu`→`Configure`→`Architecture`)


Set Architecture

Code Address Bus	12
Data Address Bus	10
FP Addition Latency	4
Multiplier Latency	8
Division Latency	12

Warning: This will cause a reset!

   iii) Modify the parameter in the available range and compute the speedup on every case, then compare the obtained results against the ones calculated using the Amdahl's in every case.

Appendix: *winMIPS64 Instruction Set*

WinMIPS64

<u>The following assembler directives are supported</u>

.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.

<u>The following instructions are supported</u>

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