Computer Architectures **02LSEOV 02LSEOQ [MA-ZZ]**	Delivery date: Thursday 1/11
Laboratory **3**	Expected delivery of lab_03.zip must include: - `program_2_a.s`, `program_2_b.s` and `program_2_c.s` - this file compiled and if possible in pdf format.

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

- Code address bus: 12
- Data address bus: 12
- Pipelined FP arithmetic unit (latency): 4 stages
- Pipelined multiplier unit (latency): 8 stages
- divider unit (latency): not pipelined unit, 12 clock cycles
- Forwarding is enabled
- Branch prediction is disabled
- Branch delay slot is disabled
- *Integer ALU: 1 clock cycle*
- *Data memory: 1 clock cycle*
- *Branch delay slot: 1 clock cycle*.

1) Starting from the assembly program you created in the previous lab called **`program_2.s`**, modify it in the following:

```
for (i = 1; i <= 30; i++){
        v3[i] = v1[i]*v2[i];
        v4[i] = v3[i]/v2[i];
        v5[i] = v4[i]+v2[i];
}
```

   a. Detect manually the different data, structural and control hazards that provoke a pipeline stall.

      - One RAW data hazard is produced by the mul.d operation (operand in f2 is read before it is fetched from the memory via the previous l.d);
      - One RAW data hazard is produced also during the ID phase of the div.d operation because of the stall of the mul.d;
      - Seven RAW data hazards are produced by the div.d operation (waiting for the mul.d to complete);
      - Seven RAW data hazards are produced also during the ID phase of the add.d operation because of the stall of the div.d;
      - Eleven RAW data hazards are produced by the add.d operation (waiting for the div.d to complete);
      - Eight RAW data hazards and one structural hazard are produced during the second s.d instruction (waiting for the div.d to complete and then blocked by the MEM phase of that instruction);

- Two RAW data hazards and one structural hazard are produced during the third s.d instruction (waiting for the add.d to complete and then blocked by the MEM phase of that instruction);
- One RAW data hazard is produced during the ID phase of the branch operation (branches read operands in ID, and for this reason it is waiting for the update on r2 to be computed).

The total number of stalls is therefore:

$$RAW = 30\ loops * (1 + 1 + 7 + 7 + 11 + 8 + 2 + 1)\ \frac{stalls}{loop}$$
$$= 1140\ stalls$$
$$STR = 30\ loops * (1 + 1)\ \frac{stalls}{loop} = 60\ stalls$$

As reported by the simulator.

For clock cycles counting, we have:

- Outside the loop:
  - Five clock cycles for r1 initialization;
  - One clock cycle for r2 initialization.
- Inside the loop:
  - One clock cycle for each of the load instructions;
  - Nine clock cycles for the mul.d operation;
  - Twelve clock cycles for the div.d operation;
  - Four clock cycles for the add.d operation;
  - One clock cycle for the store instruction of f5;
  - One clock cycle for the r2 update operation;
  - Two clock cycles for the branch instruction;
  - One clock cycle for the (only fetched for the first 29 iterations) halt instruction.

In total, we have:

$$6\ cycles + 30\ loops * (1 + 1 + 9 + 12 + 4 + 1 + 1 + 2 + 1)\frac{cycles}{loop}$$
$$= 966\ cycles$$

As reported by the simulator.

b. Optimize the program by re-scheduling the program instructions in order to eliminate the most hazards provoking stalls. Compute manually the number of clock cycles the new program (**`program_2_a.s`**) requires to execute, and compare the obtained results with the ones obtained by the simulator.

Given the data dependencies among the three operations, there is not much to do in terms of simple rescheduling (changing the order of the operations would change their final results). At the same time, moving the load and store operations has no effect on the clock cycles count and it is therefore useless.
An optimization that may be performed is to loop from the end of the vector: this has the effect of removing the RAW stall in the branch (r1 update is moved as the first instruction of the loop).

In this case, the clock cycles count decreases by one outside the loop (no need for r2 setup) and by one inside the loop (one less stall), resulting in:

$$5\ cycles + 30\ loops * (1 + 1 + 9 + 12 + 4 + 1 + 1 + 1 + 1)\frac{cycles}{loop}$$
$$= 935\ cycles$$

As reported by the simulator.

If we consider not only the possibility of reordering the operations but also their logical meaning, we can figure out that:

$$v3[i] = v1[i] * v2[i]$$

$$v4[i] = \frac{v3[i]}{v2[i]} = \frac{v1[i] * v2[i]}{v2[i]} = v1[i]$$

$$v5[i] = v4[i] + v2[i] = v1[i] + v2[i]$$

It is therefore completely useless to perform the division operation, and at the same time possible to run in parallel the multiplication and addition operations. This is exploited in a little more optimized version of the program (**`program_2_a_opt.s`**), which is also provided.
With this optimization, which is not strictly speaking a simple reordering of operations, the program generates only 90 RAW stalls:

- Two RAW stalls (and a STR stall) are produced for any loop during the store of the result of the addition;
- A RAW stall (and a STR stall) is produced for any loop during the store of the result of the multiplication.

The total clock cycles count is therefore as low as 455 cycles:

- Outside the loop:
    - Five clock cycles for r1 initialization.
- Inside the loop:
    - One clock cycle for the r1 update operation;
    - One clock cycle for each of the load instructions;
    - One clock cycle for the store operation of the "result of the division" (store f1 in v4);
    - Eight clock cycles for the mul.d operation;
    - One clock cycle for the store operation of the result of the multiplication (store f3 in v3);
    - One clock cycle for the branch instruction;
    - One clock cycle for the (only fetched for the first 29 iterations) halt instruction.

c. Starting from **`program_2_a.s`**, enable the *branch delay slot* and re-schedule some instructions in order to improve the previous program execution time. Compute manually the number of clock cycles the new program (**`program_2_b.s`**) requires to execute, and compare the obtained results with the ones obtained by the simulator.

The branch delay slot technique can be used to move a store operation after the branch: thanks to the branch delay slot, in fact, this will be in any case executed. In this particular case, we experience one less RAW stall in the store operation for f5 (it has been delayed by one clock cycle).
The resulting clock cycles count is modified as follow:

- Two less clock cycles are used, inside the loop, since the branch can terminate its execution before the add.d and the store operation of f5 produces one less stall;
- One more clock cycle is used, outside the loop, to perform the halt.

This results in:

$$5\,cycles + 30\,loops * (1+1+9+12+4+1+1)\frac{cycles}{loop} + 1\,cycle$$
$$= 876\,cycles$$

As reported by the simulator.

By using the same store and branch exchange on the logically optimized version of the program (**`program_2_b_opt.s`**), we get similar results (396 cycles, with 60 RAW stalls).

d. Unroll 3 times the program (**`program_2_b.s`**), if necessary re-schedule some instructions and renaming the used registers. Compute manually the number of clock cycles the new program (**`program_2_c.s`**) requires to execute, and compare the obtained results with the ones obtained by the simulator.

To perform the unrolling of the loop, different register renaming has been adopted; in particular, registers r2 and r3 have been used to store the other two indexes for accessing v1 and v2 (differing always by 8 from each other) and registers f6 to f15 to store the other temporary results. The instructions order which proved to be the one with less stalls is the following:

- At the beginning, loads and indices updates are interleaved with the multiplication operations (just one RAW is produced during the last mul.d, caused by the not yet completed load in f12);
- In the middle, divisions, additions and stores are interleaved in order to reduce the number of RAW and STR stalls (caused exclusively by the long division operation);
- At the end, the last stores are executed, leading to a bunch of RAW and STR stalls caused by the need to wait for completion of the last division and addition.

The stalls and clock cycles counts are reported in the attached pdf file (557 cycles, 140 RAW stalls, 200 STR stalls).

By applying the same principles on the more optimized version of the program (**`program_2_c_opt.s`**) and reorganizing a little more the operations, we can reach as low as 297 cycles, with no RAW stall and 40 STR stalls. The exact computation is reported in the attached pdf file as well.

Complete the following table with the obtained results:

Program / Clock cycle computation	`program_2.s`	`program_2_a.s`	`program_2_b.s`	`program_2_c.s`
`By hand`	966 cycles 1140 RAW 60 STR	935 cycles 1110 RAW 60 STR ——— 455 cycles 90 RAW 60 STR	876 cycles 1080 RAW 60 STR ——— 396 cycles 60 RAW 60 STR	557 cycles 140 RAW 200 STR ——— 297 cycles 0 RAW 40 STR
`By simulation`	966 cycles 1140 RAW 60 STR	935 cycles 1110 RAW 60 STR ——— 455 cycles 90 RAW 60 STR	876 cycles 1080 RAW 60 STR ——— 396 cycles 60 RAW 60 STR	557 cycles 140 RAW 20 STR ——— 297 cycles 0 RAW 40 STR

Compare the results obtained in the point 1, and provide some explanation in the case the results are different.

Eventual explanation:

The WinMIPS64 simulator does not count well the structural stalls caused by the unpipelined division module: even if they are correctly identified and reported in the pipeline evolution window, they are not considered for the overall count of structural stalls. Thus, the value reported in the simulator is much lower than the one computed by hand, since the value computed by the emulator only takes into account the structural stalls caused by the store instructions.