Lab. 1: “PERFORMANCE ANALYSIS”

Computer Architecture and Engineering (3^{rd} course)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Objectives:

- Using of the Amdahl's law.
- Evaluating and comparing the performance of different architectures.

Development:

Amdahl's law.

Amdahl's law quantifies the maximum expected overall system speed-up (S') when a part of that system, used during a fraction (F) of its execution time, is enhanced with a (local) speed-up of S. This is the expression relating S', F and S :

$$S' = \frac{1}{1 - F + \frac{F}{S}}$$

However, it is sometimes hard to compute F, either because applications' source code is not available or because the use of the considered component is distributed all along the execution time of applications. In such cases, F can be obtained using the Amdahl's law in a ″reverse way″ .

Doing this encompasses with the execution of the application under study on two variants of the considered component, whose speed-up (S) must be known. The quotient of both execution times is the global speed-up S'. Knowing S and S', it is only a matter of finding the value of F.

Let us introduce an example to illustrate these ideas. Assume that the goal is to obtain the fraction of time F employed by some application. The application makes computations with matrices (`matrix`), in order to compute scalar products. The scalar product of two vector $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_n\}$ of size n can be defined as,

$$A.B = \{a_1, a_2, ..., a_n\}.\{b_1, b_2, ..., b_n\} = a_1b_1 + a_2b_2 + ... + a_nb_n$$

this is a basic operation in matrix multiplication. The idea here is to test two implementations of this operation, one relying on scalar instructions and another using SSE (*Streaming SIMD Extensions*) instructions.

```
float Scalar(float *s1,
             float *s2,
             int size)
{

    int i;
    float prod = 0.0;

    for(i=0; i<size; i++) {
      prod += s1[i] * s2[i];
    }

    return prod;

} // end Scalar()
```

Figure 1: Scalar product implemented with standard instructions.

```
float ScalarSSE(float *m1,
                float *m2,
                int size)
{
    float prod = 0.0;
    int i;
    __m128 X, Y, Z;

    Z = _mm_setzero_ps(); /* all to 0.0 */
    for(i=0; i<size; i+=4) {
      X = _mm_load_ps(&m1[i]);
      Y = _mm_load_ps(&m2[i]);
      X = _mm_mul_ps(X, Y);
      Z = _mm_add_ps(X, Z);
    }

    for(i=0; i<4; i++) {
      prod += Z[i];
    }

    return prod;

} // end ScalarSSE()
```

Figure 2: Scalar product implemented with SSE instructions.

Computing the local speed-up *S*

The functions ***Scalar()*** and ***ScalarSSE()*** implement the two versions of the scalar product. The implementation with SSE instructions should be faster because these instructions can operate simultaneously with different data. In our case, it handles 4 floating point numbers at the same time. Both scalar- and SSE-based implementations can be checked in Figure 1 and 2 respectively.

The quotient between the execution time of these functions is the S appearing in the Amdahl's law formula. In order to exercise the code of considered functions, a simple program in C is used. Figure 3 shows the code of this program, also available in file `scalar.c`.

Looking at the code we can see that two lines have the following comment at the end,

```
\\ MODIFY
```

we will work with these lines in order to obtain de execution times. We need to obtain the execution time with both implementations (t_{std} for the implementation with standard instructions, t_{sse} for the implementation with SSE instructions), and the overload due to the loop and initializations (t_{load}). Then, the execution time of each implementation of the scalar product can be easily deduced by computing $t_{std} - t_{load}$ on one hand, and to $t_{sse} - t_{load}$ on the other hand.

```
int main(int argc, char * argv[]) {

    int     i;
    time_t  start, stop;
    double  avg_time;
    double  cur_time;
    int     rep=10;
    int     msize=MSIZE;
    float   fvalue;
    char   *cmd;

    if (argc == 2) {
      rep = atoi(argv[1]);
    } else if (argc == 3) {
      rep = atoi(argv[1]);
      msize = atoi(argv[2]);
    } // end if/else
    fprintf(stderr, "Rep = %d / size = %d\n", rep, msize);

    for(i=0; i<rep; i++) {
      init_vector(vector_in, msize);
      init_vector(vector_in2, msize);
      fvalue = Scalar(vector_in2, vector_in, msize);       // MODIFY
      //fvalue = ScalarSSE(vector_in2, vector_in, msize);  // MODIFY
    } // end for

    exit(0);

} // end main()
```

Figure 3: Program `scalar.c` employed to obtain the execution time of both implementations.

To obtain t_{std} we have to check that the file `scalar.c` have the line that call the function *textbfScalar()* uncommented and the line invoking the fuction *textbfScalarSSE()* commented.

```
  ...

 fvalue = Scalar(vector_in2, vector_in, msize);       // MODIFY
 //fvalue = ScalarSSE(vector_in2, vector_in, msize);  // MODIFY
  ...
```

Then, the program can be compiled using the following command:

```
gcc -O0 -msse -o scalar-std scalar.c
```

and then executing it and reading the time given by the `time` command. Nevertheless we have to set a fixed processor speed before executing it in order to obtain meaningful measures. By default processor cores speed is configured as in mode *ondemand*. Then the cores speed will vary depending on the processor load. To set a fixed speed we employ the `cpufreq-set` command, and we can inspect the current mode by mean of the `cpufreq-info` command. We will set the processor speed to 3.3Ghz by means of. The parameter `-c #` allows selecting the core we will fix,

```
cpufreq-set -c 0 -f 3.3GHz
cpufreq-set -c 1 -f 3.3GHz
cpufreq-set -c 2 -f 3.3GHz
cpufreq-set -c 3 -f 3.3GHz
```

and we will verify with,

```
cpufreq-info
```

or,

```
cat /proc/cpuinfo
```

now we can run the program.

NOTE: in linux the current directory is NOT in the PATH by default, then we have to add ./ previous to every local program in order to execute it. To avoid this we can run or add to the $HOME/.basrc file the following line:

```
export PATH=./:$PATH
```

Now we can run the program `scalar-std`,

```
time scalar-std 100000 1024
```

The parameters in the command simply mean that we want the loop to iterate 100000 times working on vectors of 1024 components. Write down the time employed by the program (*user+system*) as t_{std}.

Now we edit again the file `scalar.c` and we comment the line that call the function ***Scalar()*** and we uncomment the line containing the function ***ScalarSSE()***,

```
  ...

 //fvalue = Scalar(vector_in2, vector_in, msize);   // MODIFY
 fvalue = ScalarSSE(vector_in2, vector_in, msize);  // MODIFY
  ...
```

we compile and run the program again to obtain t_{sse},

```
gcc -O0 -msse -o scalar-sse scalar.c
time scalar-sse 100000 1024
```

write down the new time. Compiler flag ***-msse*** is required to allow the use of SSE instructions.

Finally we edit again the file `scalar.c` and we comment both lines containing scalar product calls. This way we can estimate the overload due to the loop and initializations.

```
    ...
 //fvalue = Scalar(vector_in2, vector_in, msize);      // MODIFY
 //fvalue = ScalarSSE(vector_in2, vector_in, msize);   // MODIFY
    ...
```

we compile and run the new program,

```
gcc -O0 -msse -o scalar-load scalar.c
time scalar-load 100000 1024
```

writing down the new time as t_{load}. With the obtained data we can calculate S as,

$$S = \frac{t_{std} - t_{load}}{t_{sse} - t_{load}}$$

Once obtained S, the local speed-up, we will calculate the execution time of the application `matrix` with both implementations. The relation between resulting execution times will give us the global speed-up (S'). With S and S' we can use the Amdahl law to find the local fraction of time (F) that the scalar product is used originally in the application `matrix`.

Computation of the local fraction of time *F*

We will apply the previous described process to obtain the fraction of time that the scalar product is executed in application `matrix`.

We need the source code of the application to obtain the execution time with both implementations and to obtain S',

$$S' = \frac{t_{mat-std}}{t_{mat-sse}}$$

To get $t_{mat-std}$ we have to edit the file `matrix.c` and lool for the macro **__SCALAR_PROD()** at the very beginning of the file.

```
    ...
 #define __SCALAR_PROD(v1, v2, s)  Scalar(v1, v2, s);
 //#define __SCALAR_PROD(v1, v2, s)  ScalarSSE(v1, v2, s);
 //#define __SCALAR_PROD(v1, v2, s)
    ...
```

in this experiment only the standard version have to be uncomment as seen in the previous code. This way the application `matrix` will use this implementation of the scalar product throughout the code.

compile and execute,

```
gcc -O0 -msse -o matrix-std matrix.c -lm
time matrix-std 1 1024
```

only one iteration is performed and matrices are of size of $1024x1024$ in order the size of the vectors to be the same as in the previous experiments. Write down the time as $t_{mat-std}$.

Next edit again the file `matrix.c` and let only uncommented the line that defines the macro that use the scalar product implementation using SSE as seen in the following code,

```
   ...
//#define __SCALAR_PROD(v1, v2, s)  Scalar(v1, v2, s);
#define __SCALAR_PROD(v1, v2, s)  ScalarSSE(v1, v2, s);
//#define __SCALAR_PROD(v1, v2, s)
   ...
```

Now, compile and execute,

```
gcc -O0 -msse -o matrix-sse matrix.c -lm
time matrix-sse 1 1024
```

write down the time as $t_{mat-sse}$.

⇒ With the data obtained, compute the local fraction of time that the application `matrix` use the scalar product.

Experimental local fraction of time computation (F_{exp})

Although this is not always possible, due to the kind of code of the application `matrix` and the component under study, we can compute experimentally the fraction of time. To do that we have only to define the macro that define the scalar product as empty.

```
   ...
//#define __SCALAR_PROD(v1, v2, s)  Scalar(v1, v2, s);
//#define __SCALAR_PROD(v1, v2, s)  ScalarSSE(v1, v2, s);
#define __SCALAR_PROD(v1, v2, s)
   ...
```

this way we can obtain the time that the application employs in the other tasks. We will refer to this time as $t_{mat-res}$ and will help us to compute experimentally the fraction of time (F_{exp}) by means of the formula,

$$F_{exp} = \frac{t_{mat-std} - t_{mat-res}}{t_{mat-std}}$$

compile and execute this new version,

```
gcc -O0 -msse -o matrix-res matrix.c -lm
time matrix-res 1 1024
```

and write down the value $t_{mat-res}$.

⇒ Compute F_{exp}. Compare the experimental value and the value obtained by means of the Amdahl law.

Performance analysis of architectures.

This section focuses on the measure of the performance of lab computers using the following programs:

- two synthetic *benchmarks* (**dhrystone**, for integer arithmetic and **whetstone**, for floating point arithmetic)
- two real applications: the C language compiler **gcc**, that only uses integer arithmetic and the **xv** applications, that processes images.

To cope with the aforementioned goals, programs will be executed and their execution times will be measured using the `time` order:

- **dhrystone** (10.000.000 iterations):

  ```
  time dhrystone
  ```

  Indicate the 10.000.000 iterations that must be performed.

  Note the execution time of the program $T_{dhrystone}$.

- **whetstone** (10.000 iterations):

  Now we type:

  ```
  time whet-h 10000
  ```

  Note the execution time T_{whet-h} and the other data provided by the program.

- C language compiler, compiling the *xv* program

  Let's compile the application xv. Assume source code is located in our directory at folder `xv-310a/` . Type the following orders:

  ```
  cd xv-310a
  make clean
  time make
  ```

  Note the execution time T_{gcc}.

- *xv* application

  Now we execute the **xv** application that have just compile:

  ```
  time xv-310a/xv -wait 5 mundo.jpg
  ```

  Note the execution time T_{xv}.

The following table shows the data (execution times in seconds) obtained from the execution of the application in three different machines. Machine C is the one where we are working on in the lab:

Program/Machine	A	B	C
dhrystone	5	18	
whetstone	2.5	10	
gcc	40	130	
xv	4.5	15	

$\Rightarrow$ Compare the performance of the three computers attending to three different measures. The first one will consider each program in an isolated way. The second one will use the arithmetic mean of executions times, and the third one will apply the geometric mean of execution times normalized to machine B.

2a Laboratory: "PIPELINED INSTRUCTION UNIT (I)"

Computer architecture and engineering ($3th$ course)
E.T.S. de Ingeniería Informática (ETSINF)
Dept. of Computer Engineering (DISCA)

Goals:

- Knowing the use of a pipelined processor simulator.
- Analyzing the impact of control and data hazards in the performance of a pipelined instruction unit.
- Performing DLX assembler programs

Content:

The "DLX" (*DeLuXe*) was introduced as an example computer in the first edition (1990 year) of the famous *Computer Architecture: A Quantitative Approach* book written by J. Hennessy and D. Patterson. It is a 32-bits computer implementing a MIPS-like instruction set. At the end of this assignment a brief description of this instruction set is included.

The following piece of DLX code executes the following vector operation: $\vec{Z} = a + \vec{X} + \vec{Y}$:

```
; z = a + x + y
; Size of vectors: 16 words
; Vector x
        .data
x:      .word 0,1,2,3,4,5,6,7,8,9
        .word 10,11,12,13,14,15
; Vector y
y:      .word 100,100,100,100,100,100,100,100,100,100
        .word 100,100,100,100,100,100

; Vector z
; 16 elements are 64 bytes
z:      .space 64

; Scalar a
a:      .word -10

; The code
        .text

start:
        add r1,r0,x
        add r4,r1,#64 ; 16*4
        add r2,r0,y
        add r3,r0,z
        lw r10,a(r0)

loop:
        lw r12,0(r1)
        add r12,r10,r12
        lw r14,0(r2)
        add r14,r12,r14
        sw 0(r3),r14
        add r1,r1,#4
        add r2,r2,#4
        add r3,r3,#4
        seq r5,r4,r1
        beqz r5,loop

        trap #0         ; Program end
```

This program is saved in file `APXPY.S`.

Use of the DLX computer simulator

1. Execute the simulator of the DLX pipelined instruction unit (**DLXide**). It is able to simulate both the execution cycle of DLX instructions and the flow of such instructions through the computer datapath. All DLX instructions operating on the integer register file are supported. Instructions and data cache are separated (*Harvard* architecture). Registers are written and read in the first and second half of the clock cycle, respectively. Control hazards can be solved by using various strategies: using *stalls*, *predict-not-taken* y delayed branch, with two alternative *delay-slot*s of one or three instructions respectively. Data hazards can be solved by inserting stall or applying the *forwarding* technique. It must be noted that the simulated datapath changes according to the strategy employed to solve the hazards.

   In order to execute the simulator go to the directory created when sources (downloaded from PoliformaT) are decompressed, and executed the program **dlxide** from directory **DLXide**. In order to configure the simulated, go to menu **Simulador**, option *Configuración DLX*. The program will open a dialog windows showing the various available strategies that can be used to solve data and control hazards. By default, stalls are inserted in both cases.

2. Load the program `APXPY.S`. In order to do so, go to menu **Archivos**, option *Abrir*, and select the corresponding file. Once the program loaded, it must be assembled (use menu **Simulador**, option *Ensamblar*). If the program contains errors, they will be shown in the low part of the program window. After error correction, the program must be re-assemble. In case of containing no error, the program is stored in the memory of the simulated computer, and the action is reported to the user.

   Check that the loaded code corresponds to the one showed before.

3. Create the simulation window to executed the program (**Simulador** menu, *Ejecutar* option). A new window will be opened. The instructions–time diagram will be shown in such window. A window with multiple tabs enabling the inspection of the computer state will be also provided.

   The *Configuración* tab enables remembering which are the selected strategies for solving hazards. The *Registros* tab let us visualizing general use registers' content. By double clicking the "value"field you can alter its value. By clicking the left button the base used for coding numbers can be changed. PC Indexado and *Memoria* tabs enable visualizing the content of instruction and data cache, respectively. Finally, the *Estadísticas* tab reports the number of taken cycles, the set of executed instructions, the inserted stalls and the applied short-circuits.

   The simulator enables the execution of programs step (cycle) by step (cycle), forward various cycles (*Ejecutar Ciclos* button) or complete its execution until the end (*Ejecutar* button). After each clock cycles, the instructions–time diagram is updated. When an stall is inserted, stages keeping the same instructions are shown in italic. The computer data path is also updated. Each new fetched instruction has a new assigned color, which is used for signaling the path followed by the instruction within the instruction unit. When in one stage there is no instruction, it is shown the text `-nop-`[1]. In the low part of the data path window it also appears some control signals that are activated to insert stalls, abort instructions under execution or apply short-circuits:

   - *IF.stall*: Keeps the instruction in the IF stage in the same stage during the following clock cycle. It also sends to the ID stage the equivalent to a `nop` instruction.
   - *ID.stall*: Keeps the instruction in the ID stage in the same stage during the following clock cycle. It also sends to the EX stage the equivalent to a `nop` instruction.
   - *ID.nop, EX.nop, MEM.nop*: Converts the instruction in the corresponding stage to a `nop` instruction.
   - *WBtoMEM,WBtoEX,WBtoID,MEMtoEX,MEMtoID*: Applies a short-circuit between involved stages, by providing the adequate control signals to multiplexors.

   Using the default configuration (control hazards: *stalls* and data hazards: *stalls*), execute the program step by step for the first loop iteration. Observe how instructions evolve inside the pipelined instruction unit and the insertion of stalls when hazards are detected. Execute the whole program and check the results stored in memory at the address labelled as *z*, where the result vector is allocated. Note the number of executed instruction, the execution time and the number of inserted stalls. Compute the resulting CPI.

4. Modify the configuration of the simulator to solve control hazards using the *predict-not-taken* technique. Execute the program step by step for the first loop iteration.

[1]Avoid any confusion with the real `nop` operation

Observe how instructions evolve inside the pipelined instruction unit and the insertion of stalls when hazards are detected. Execute the whole program and check the results stored in memory at the address labelled as *z*, where the result vector is allocated. Note the number of executed instruction, the execution time and the number of inserted stalls. Compute the resulting CPI.

5. Keep the strategy for solving control hazards to *predict-not-taken* and modify the configuration of the simulator for solving data hazards relying on the *forwarding* technique. Execute the program step by step for the first loop iteration. Observe how instructions evolve inside the pipelined instruction unit and the insertion of stalls when hazards are detected. Execute the whole program and check the results stored in memory at the address labelled as *z*, where the result vector is allocated. Note the number of executed instruction, the execution time and the number of inserted stalls. Compute the resulting CPI.

Modification of the provided code.

The goal of this part of the lab is to carry out some code modification in order to reduce as much as possible the number of stalls.

1. Choose the *predict-not-taken* and *forwarding* techniques and modify the code to reduce the penalty relating to data hazards. Take into account that the only instructions that can introduce stall to solve data hazards are load instructions. Execute the program until the end and note the number of executed instructions, the execution time and the number of stalls. Compute the resulting CPI.

2. Keeping the *forwarding* technique to solve data hazards, select the *delay-slot 3* as a strategy to solve control hazards. Using the code produced for the previous section, modify such code for its execution trying to fill the existing *delay-slot* with useful instructions. Execute the program until the end and note the number of executed instructions, the execution time and the number of stalls. Compute the resulting CPI.

3. Keeping the *forwarding* technique to solve data hazards, select the *delay-slot 1* as a strategy to solve control hazards. Using the code produced for section 1, modify such code for its execution trying to fill the existing *delay-slot* with useful instructions. Execute the program until the end and note the number of executed instructions, the execution time and the number of stalls. Compute the resulting CPI.

Optimisation of a program.

The high level code shows how to sort a vector using the selection sort method:

```
...
FOR i := 0 TO n-2 DO
  FOR j := i+1 TO n-1 DO
    IF a(i)>a(j) THEN
       temp := a(i);
```

```
        a(i) := a(j);
        a(j) := temp;
     END;
   END;
END;
...
```

The equivalent DLX assembler program can be found in file `ORDENA.S`.

1. Take a little time to study and to understand the assembler version. Then configure the simulator in order to use the *forwarding* and the *predict-not-taken* techniques. Execute the program using the simulator and check its correct behavior. Analyze its execution time and its CPI.

2. Modify the produced program in order to use the delayed brach technique with a *delay-slot* set to 1. Configure the simulator and execute the program checking its correct behavior. Optimize the program in order to reduce the number of inserted stalls and filling the *delay-slot* with useful instructions. Analyze its execution time and its CPI.

El juego de instrucciones del DLX.

Instruction type/opcode	Instruction meaning
Data transfers	**Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR**
`LB,LBU,SB`	Load byte, load byte unsigned, store byte
`LH,LHU,SH`	Load half word, load half word unsigned, store half word
`LW,SW`	Load word, store word (to/from integer registers)
`LF,LD,SF,SD`	Load SP float, load DP float, store SP float, store DP float
`MOVI2S, MOVS2I`	Move from/to GPR to/from a special register
`MOVF, MOVD`	Copy one FP register or a DP pair to another register or pair
`MOVFP2I,MOVI2FP`	Move 32 bits from/to FP registers to/from integer registers
Arithmetic/logical	**Operations on integer or logical data in GPRs; signed arithmetic trap on overflow**
`ADD,ADDI,ADDU, ADDUI`	Add, add immediate (all immediates are 16 bits); signed and unsigned
`SUB,SUBI,SUBU, SUBUI`	Subtract, subtract immediate; signed and unsigned
`MULT,MULTU,DIV,DIVU`	Multiply and divide, signed and unsigned; operands must be FP registers; all operations take and yield 32-bit values
`AND,ANDI`	And, and immediate
`OR,ORI,XOR,XORI`	Or, or immediate, exclusive or, exclusive or immediate
`LHI`	Load high immediate—loads upper half of register with immediate
`SLL, SRL, SRA, SLLI, SRLI, SRAI`	Shifts: both immediate (`S__I`) and variable form (`S__`); shifts are shift left logical, right logical, right arithmetic
`S__,S__I`	Set conditional: "__" may be `LT,GT,LE,GE,EQ,NE`
Control	**Conditional branches and jumps; PC-relative or through register**
`BEQZ,BNEZ`	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
`BFPT,BFPF`	Test comparison bit in the FP status register and branch; 16-bit offset from PC+4
`J, JR`	Jumps: 26-bit offset from PC+4 (`J`) or target in register (`JR`)
`JAL, JALR`	Jump and link: save PC+4 in R31, target is PC-relative (`JAL`) or a register (`JALR`)
`TRAP`	Transfer to operating system at a vectored address
`RFE`	Return to user code from an exception; restore user mode
Floating point	**FP operations on DP and SP formats**
`ADDD,ADDF`	Add DP, SP numbers
`SUBD,SUBF`	Subtract DP, SP numbers
`MULTD,MULTF`	Multiply DP, SP floating point
`DIVD,DIVF`	Divide DP, SP floating point
`CVTF2D, CVTF2I, CVTD2F, CTD2I, CVTI2F, CVTI2D`	Convert instructions: `CVTx2y` converts from type `x` to type `y`, where `x` and `y` are `I` (integer), `D` (double precision), or `F` (single precision). Both operands are FPRs.
`__D,__F`	DP and SP compares: "__" = `LT,GT,LE,GE,EQ,NE`; sets bit in FP status register

2nd Lab: "PIPELINED INTRUCTION UNIT"

Computer Architecture and Engineering (3rd year)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Goals:

- Complete logic required to solve data and control hazards in a pipelined processor.

Development:

The development of this lab is based on the use of a MIPS simulator working with integer instructions. The simulator allows direct interpretation of MIPS assembler code. It simulates a 5-stage pipelined (IF, ID, EX, MEM y WB). The code required to solve data and control hazards is not included in the simulator.

This document structures as follows. It first describes which are the files included in the simulator, then the implemented instructions, data structures under use, the internal structure of the pipelined MIPS simulator and, finally, the exercises to be carried out.

Structure of the simulator

The MIPS simulator includes the following file written in C:

main.c Main simulator program. It is in charge of reading the assembler and executing the different stages of the pipelined instruction unit.

main.h It contains all simulator shared variables: instruction and data cache, general purpose registers, inter-stage registers, control signals, etc.

tipos.h It contains definitions of all data instructions used in the simulator: instruction and data cache, register files, inter-stage registers, instruction formats, etc.

input.lex.l It contains the lexical description of the supported assembler language.

input.yacc.y It contains the grammatical rules for the synthactic analysis of the considered assembler code.

etiquetas.c, etiquetas.h They contain code related to the management of assembler labels.

presentacion-html.c,presentacion-txt.c They contain all functions required for presenting results.

mips.c It contains the implementation of all the stages of the instruction unit: IF, ID, EX, MEM and WB. *This is the file to be modified along this lab.*

Implemented instructions

The simulator executes all MIPS integer instructions, including versions register–register and register–immediate. Unconditional branch instructions are not supported. More precisely, it supports the following instruction set:

```
  ADD Rx, Ry, Rz    ADDI Rx, Ry, #Inm    LW Rx, desp(Ry)
  SUB Rx, Ry, Rz    SUBI Rx, Ry, #Inm    SW Rx, desp(Ry)
  AND Rx, Ry, Rz    ANDI Rx, Ry, #Inm    BEQZ Rx, desp
  OR Rx, Ry, Rz     ORI Rx, Ry, #Inm     BNEZ Rx, desp
  XOR Rx, Ry, Rz    XORI Rx, Ry, #Inm    NOP
  SRA Rx, Ry, Rz    SRAI Rx, Ry, #Inm    TRAP
  SRL Rx, Ry, Rz    SRLI Rx, Ry, #Inm
  SLL Rx, Ry, Rz    SLLI Rx, Ry, #Inm
  SEQ Rx, Ry, Rz    SEQI Rx, Ry, #Inm
  SNE Rx, Ry, Rz    SNEI Rx, Ry, #Inm
  SGT Rx, Ry, Rz    SGTI Rx, Ry, #Inm
  SGE Rx, Ry, Rz    SGEI Rx, Ry, #Inm
  SLT Rx, Ry, Rz    SLTI Rx, Ry, #Inm
  SLE Rx, Ry, Rz    SLEI Rx, Ry, #Inm
```

Data structures

Hereafter paragraphs describe existing data structures (defined at file `tipos.h`) their use.

Basic types

Existing basic types are:

```
typedef unsigned char   byte;   /* One byte: 8 bits */
typedef short           half;   /* Half-word : 16 bits */
typedef int             word;   /* One word: 32 bits */

typedef enum {NO=0, SI=1} boolean; /* Logic value */
```

Instruction format

Instruction formats are represented through an enumerated type:

```
/* Instruction formats */
typedef enum {FormatoR, FormatoI, FormatoJ} formato_t;
```

Instructions are represented through the following data structure:

```
typedef struct {
  codop_t       codop;          /* Operation code */
  formato_t     tipo;           /* Format */

  byte          Rfuente1,       /* Source register 1 */
```

```
                 Rfuente2;       /* Source register 2 */
  byte           Rdestino;       /* Destination register */
  half           inmediato;      /* Immediate Value */
} instruccion_t;
```

Register file

The register file is a vector including elements of type `reg_int_t`, with a single field, named *valor*.

```
typedef struct {
  word           valor;              /* Register value */
} reg_int_t;
```

Inter-stage registers

Inter-stage registers are represented through an struct containing all the following fields:

- IF/ID register:

```
  typedef struct {
    instruccion_t IR;               /* IR */
    word          NPC;              /* PC+4 */

    word          iPC;
    long          orden;            /* Display information */
  } IF_ID_t;
```

- ID/EX Register:

```
  typedef struct {
    instruccion_t IR;               /* IR */
    word          NPC;              /* PC+4 */
    word          Ra,               /* Registers' value*/
                  Rb;
    word          Imm;              /* Immediate value with extended sign */

    word          iPC;
    long          orden;            /* Display information */
  } ID_EX_t;
```

- EX/MEM Register:

```
  typedef struct {
    instruccion_t IR;               /* IR */
    word          ALUout;           /* Result */
    word          data;             /* Data to be written */
    boolean       cond;             /* Result defining a branch condition */
    word          iPC;
    long          orden;          /* Display information */
  } EX_MEM_t;
```

- MEM/WB Register:

```
  typedef struct {
    instruccion_t IR;             /* IR */
    word          ALUout;         /* Result */
    word          MEMout;         /* Result */

    word          iPC;
    long          orden;          /* Display information */
  } MEM_WB_t;
```

Other data structures

- It defines how are data hazards solved:

```
  typedef enum {
    parada,               /* by stall insertion */
    cortocircuito,        /* by forwarding + stalls */
    ninguno               /* nothing */
  } riesgos_d_t;
```

- It defines how are control hazards solved:

```
  typedef enum {
    stall,                /* By stall insertion*/
    pnt,                  /* Predict-not-taken */
    ds3,                  /* Delayed branch, with delay slot(ds) =3 */
    ds2,                  /* Delayed branch, with ds=2 */
    ds1,                  /* Delayed branch, with ds=1 */
  } riesgos_c_t;
```

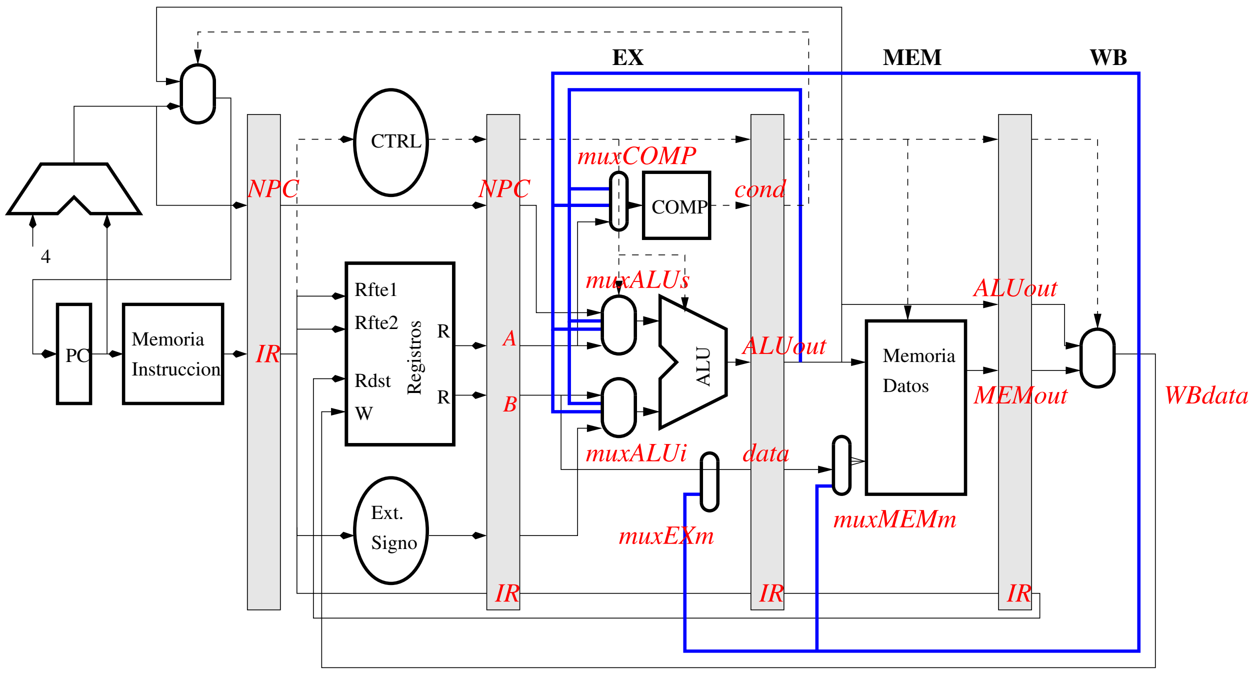
Structure of a pipelined instuction unit

The pipelined instruction unit is defined using the following elements (see figure):

Instructions memory. Stores the program to be executed. It can be addressed by word [1]. `MI` variable (of type `memoria_instruc_t`) represents such memory. Memory size is provided by constant `TAM_MEMO_INSTRUC` (see `tipos.h` file).

Integer register file. It contains integer registers. It is represented by variable `Rint`, of type `reg_int_t []`. The number of register is defined by constant `TAM_REGISTROS` (see `main.h` file).

[1] successive words are assigned to consecutive memory addresses



Arithmetic operator. It performs the arithmetic operations in stage EX. It is represented by function `operacion_ALU (codop,in1,in2)` (see `mips.c` file), where `codop`, `in1` and `in2` respectively denote the operation to be carried out and and the operands to use.

Evaluation of branch conditions. Compute the condition of a branch. It is represented by the textttoperacion_COMP (codop,in1) function (see `mips.c` file), where `codop` and `in1` respectively represents the operation to be carried out and the data to be evaluated. This function returns whether the branch must be taken or not.

Multiplexor of the upper input of the arithmetic operator. It is represented by function `mux_ALUsup (npc,ra,mem,wb)` (see`mips.c` file), where `npc,ra,mem` and `wb` represent the inputs to the multiplexor. This function returns the selected input.

Multiplexor of the lower input of the arithmetic operator. It is represented by function `mux_ALUinf (rb,imm,mem,wb)` (see`mips.c` file), where `rb,imm,mem` y `wb`represent the inputs to the multiplexor. This function returns the selected input.

Multiplexor of the comparator input (branches). It is represented by function `mux_COMP (ra,mem,wb)` (see `mips.c` file), where `ra,mem` and `wb` represent the input comparators. The function returns the selected input.

Data multiplexor to be written in memory (EX stage). It is represented by function `mux_EXmem (rb,wb)` (see `mips.c` file), where `rb` and `wb` represent the multiplexor inputs. The function returns the selected input.

Data memory. It stores the data to be used by the program. It is addressed on a byte basis [2]. It is represented by variable texttMD, of type `memoria_datos_t`. Memory size is provided by constant `TAM_MEMO_DATOS` (see `tipos.h` file).

[2]successive words are located at addresses differing in 4 bytes

Multiplexor of data to be written in memory (MEM stage). It is represented by function `mux_MEMmem (rb,wb)` (see file`mips.c`), where `rb` and `wb` represent the multiplexor inputs. The function returns the selected input.

Output of the multiplexor of WB stage. It represents the data to be written in the register file during WB. It is represent by variable `WBdata` (see `main.h` file), of type `word`.

Inter–stage registers. Their names come from the stages they interconnect. They are the following ones:

- `IF_ID`, of type `IF_ID_t`.
- `ID_EX`, of type `ID_EX_t`.
- `EX_MEM`, of type `EX_MEM_t`.
- `MEM_WB`, of type `MEM_WB_t`.

Current and new PC value They are representes by `PC` and `PCn` variables, of type `word`. The following instruction will be fetched from the location in `PCn`.

Signals generated according to the type of instruction They are related to each stage. They detect certain features of the instruction located at each corresponding stage. Signals are represented by the following functions:

- `usaIR_fuente1_ID()`. The instruction in stage ID uses source register fuente1.
- `usaIR_fuente2_ID()`. The instruction in stage ID uses source register fuente2.
- `usaIR_fuente1_EX()`. The instruction in stage EX uses source register fuente1.
- `usaIR_fuente2_EX()`. The instruction in stage EX uses source register fuente2.
- `usaIR_destino_EX()`. The instruction in stage EX uses destination register.
- `usaIR_destino_MEM()`. The instruction in stage MEM uses destination register.
- `usaIR_destino_WB()`. The instruction in stage WB uses destination register.

Control signals The simulador has the following control signals, all of them of type `boolean`:

- `IFstall`: When it becomes active (`IFstall=SI`), it stalls the instruction in the IF stage during the following clock cycle.
- `IDstall`: When it becomes active instruction in ID stage is stalled during the following clock cycle.
- `IFnop`: When it becomes active , it transforms the instruction in the IF stage into a `nop` instruction, which will be sent to ID during the following clock cycle.
- `IDnop`: When it becomes active , it transforms the instruction in the ID stage into a `nop` instruction, which will be sent to EX during the following clock cycle.

- `EXnop`: When it becomes active , it transforms the instruction in the EX stage into a `nop` instruction, which will be sent to MEM during the following clock cycle.

MIPS simulator pseudo-code

After initializing data structures, the simulator main program loads the file containing the program to execute. Then it assembles and executes the main loop in the simulator, whose pseudo-code looks as follows:

```
/* Main loop of the MIPS simulator */

  /*** Fase: WB **************/
  writing_stage(): [fase_escritura ()]
    -write into the register

  /*** Fase: MEM **************/
  memory_stage(): [ fase_memoria ()]
    -control hazard detection %detectar riesgos de control
    -use of shortcirtuits %aplicar cortocircuitos
    -memory access, if necessary %acceso a memoria, en su caso

  /*** Fase: EX *************/
  execution_stage(): [fase_ejecucion ()]
    -control hazard detection %detectar riesgos de control
    -use of shortcirtuits %aplicar cortociruitos
    -operation in ALU/COMP

  /*** Fase: ID **************/
  decoding_stage(): [fase_decodificacion()]
    -data hazard detection %detectar riesgos de datos
    -control hazard detection %detectar riesgos de control
    -register reading %leer registros

  /*** Fase: IF ***********/
  fetching_stage(): [fase_busqueda()]
    -instruction fetch %buscar instrucción
    -PC updating %actualizar PC

  Cycle++; %Ciclo++;
  print_state; %imprimir_estado;
  next_clock_cycle(); [impulso_reloj()]
```

Exercicies

First of all, check the simulator by using a simple fragment of code without data dependencies. This codes is in the file `ejemplo.s`

```
    .ireg 15,60 ; r1=15, r2=60
```

```
        .data
a:      .word  100
b:      .word  0
        .text
start: bnez r5,end
        add r3,r1,r2
        sub r4,r1,r2
        and r5,r1,r2
        or r6,r1,r2
        xor r7,r1,r2
        lw r1,a(r0)
        sw b(r0),r3
        sgt r2,r3,r4
        beqz r0,inicio ; unconditional branch
end:
```

In order to execute the simulator use the command `mips`. Simulator accepts several parameters:

```
mips -s salida -d riesgos_datos -c riesgos_control -f fichero.s
```

Parameter `-s salida` signals how we want the results. Options are `ciclo, final, html`, which respectively correspond to the execution step by step, the final results and the generation of results in *html*.

Parameter `-d riesgos_datos` defines the strategy that is used to solve data hazards. `n,p,c` are options corresponding to, respectively, no-strategy, stall insertion and forwarding.

Parameter `-c riesgos_control` denotes the control strategy to use. `p,t,3,2,1` are options corresponding to, respectively, stall insertion, *predict-not-taken* and delayed branch with *branch delay slot=3,2,1*.

By default, output is *html*, there is no data hazard detection and stalls are inserted in order to solve control hazards.

More precisely, we write:

```
mips -f ejemplo.s
```

This command will generate an **html** file for each cycle with all the information related to the state of the machine. Such files can be visualized using any web client (such as `firefox or konqueror`). **You are advised to delete these files before each new simulation:**

```
rm *.html
mips -f ejemplo.s
```

Check the correct behavior of the simulator, and the expected result of the execution. Obtained results must conform the following ones:

r0	r1	r2	r3	r4	r5	r6	r7	a	b
0	100	1	75	-45	12	63	51	100	75

However, the provided simulator only includes the necessary code to solve control hazards by inserting stalls or using the delayed branch technique with a *delay slot=3*. However, it is no able to detect or solve data hazards.

In this lab, the simulator must be enriched with new strategies for hazard detection and solving. In all cases, file `mips.c` much be modified by adding the necessary code in order perform the required actions.

In order to edit the files you can use any one of the available editors, such as `vi`, `emacs`, `[gk]edit` or `kate`.

The compilation of the `mips` simulator must be performed by executing the command `make` in the directory where the sources and the `Makefile` file are located.

In order to check the correct behavior of applied modifications, use the proof files provided.

1. Modify the MIPS simulator in order to detect and solve data hazards by inserting stalls. More precisely, data hazards must be solved for the following sequence of instructions (saved in the `datos1.s` file):

   ```
   .ireg 15,60,0,65 ; r1=15, r2=60, r3=0, r4=65
   .text
   add r3,r1,r2
   sub r3,r3,r4
   ; r3 value must be 10
   ```

   To do so, the function decoding instructions (function `fase_decodificacion`) must be modified by writing the code activating necessary control signals.

   Check modifications by executing:

   ```
   mips -d p -f datos1.s
   ```

2. Modify the MIPS simulator in order to detect and solve data hazards by applying short-circuits.

   - First, data hazards provoked by the aforementioned sequence of code (file `datos1.s`) must be solved. Such sequence of instructions do not require the insertion of stalls. To do so, modify the function implementing the multiplexors located at the entry of the arithmetic logic operator (functions `mux_ALUsup` and `mux_ALUinf`).

     Check the modification by executing:

     ```
     mips -d c -f datos1.s
     ```

   - In this exercise the data hazard provoked by a load instruction followed by an arithmetic instruction (saved in file `datos2.s`):

     ```
             .ireg 0,0,0,65 ; r1=0, r2=0, r3=0, r4=65
             .data
     a:      .word 100
             .text
             lw r3,a(r0)
             sub r3,r3,r4
             ; Result must be r3=35
     ```

In this case, in addition to the activation of the corresponding short-circuit, the insertion of a stall is necessary in the decoding stage. Consequently, it must be modified both the function implementing the multiplexors located at the input of the arithmetic-logic operator (functions `mux_ALUsup` and `mux_ALUinf`), and the function (`fase_decodificacion`) implementing the decoding of instructions.

Check your code using the following command:

```
mips -d c -f datos2.s
```

3. Modify the MIPS simulator in order to solve control hazards using a *predict-not-taken* strategy.

   In order to do so, the function `fase_búsqueda` (the one carrying out with instruction fetching) must be modified.

   In order to check this modification, the following code (saved in `suma.s`) can be used.

```
        ; add vector componente until it is found
        ; a component equal to 0
        ; the result is stored in a
        .data
a:      .word  0
y:      .word  1,2,3,0,4,5,6,7,8
        .text
        add r2,r0,y; r2 traverse y
        add r1,r0,r0; r1=0
        nop ; avoid unforeseen data hazards
bucle:  lw r3,0(r2); r3 es y[i]
        add r1,r3,r1; this hazard is already solved
        add r2,r2,#4
        bnez r3,bucle
        sw a(r0),r1
final:
```

   Check your code using the following command:

```
mips -d c -c t -f suma.s
```

3^{rd} Laboratory: "STATIC INSTRUCTIONS SCHEDULING"

Computer Architecture and Engineering (3^{rd} year)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Goal:

- Know, understand and apply some static instructions scheduling techniques.

Assignment:

The simulator of the MIPS computer with multicycle instructions

The simulator **mips-m** enables the execution of programs written using the MIPS assembler. It support a subset of the MIPS instruction set, including integer and floating-point instructions. The set of supported instructions is detailed in the annex of this lab.

The simulated processor does not integrate dynamic instruction scheduling. Solving data hazards is performed by inserting stalls or applying the forwarding technique, with stall insertion whenever necessary. Control hazards can be solved inserting stall, with *predict-not-taken* or using the delayed branch technique, with a *delay-slot* of one, two or three instructions. For the execution of multicycle instructions it is available a load/store unit, a multiplication operator, an adder and a comparison operator. All of them are pipelined and their latency can be configured.

The simulator executes using the following command line:

```
mips-m -s <results> -d <data-hazards> -c <control-hazards>
       -l <lat> -a <lat> -m <lat> -k <lat> -f <file.s>
```

where:

- <results> conditions how the result of the simulation will be shown. There are two options:
  - **time**: Shows in the terminal the execution. time.
  - **final**: Shows in the terminal the execution time, registers and memory content after the execution of the program.
  - **html**(*): Generates several html files with the state of the execution step by step and the final results. These results can be seen by opening the file **index.html** in a web browser. This is the default option.
- <data-hazards> Signals how data hazards are solved. There exist three options:
  - **n**: There is no logic for solving data hazards.

- **p**: Data hazards are solved by inserting stalls.
- **c**(*): Data hazards are solved using shorcircuits, and inserting also the necessary stalls. This is the default option.

- <control hazards> Signal how control hazards are solved. There exist three options: cómo se resuelven los riesgos de control. Hay tres opciones:
  - **p**: They are solved by inserting stalls.
  - **t**(*): They are solved using the *predict-not-taken* technique. This is the default option.
  - **3**: They are solved using the delayed branch technique, with a *delay-slot*=3.
  - **2**: They are solved using the delayed branch technique, with a *delay-slot*=2.
  - **1**: They are solved using the delayed branch technique, with a *delay-slot*=1.
- <lat> indicates the latency of the multicycle floating-point pipelined operator. It must be of at least 2 cycles:
  - *l*: Load/store operator. By default, 2 cycles.
  - *a*: Add/sub operator. By default, 4 cycles.
  - *m*: Mult/div operator. By default, 7 cycles.
  - *k*: Comparison operator. By default, 4 cycles.
- <file.s> is the name of the file containing the assembler code to execute.

Example of a MIPS program

Find following the assembler code of a loop adding an escalar value to an array stored in memory ($\vec{Z} = a + \vec{Y}$, bucle DAPY).

```
start:
        dadd r1,r0,y     ; r1 contains address y
        dadd r2,r0,z     ; r2 contains address z
        l.d f0,a(r0)     ; f0 contains a
        dadd r3,r1,#512  ; 64 elements are 512 bytes
loop:
        l.d f2,0(r1)
        add.d f4,f0,f2
        s.d f4, 0(r2)
        dadd r1,r1,#8
        dadd r2,r2,#8
        dsub r4,r3,r1
        bnez r4,loop
        nop             ; delay slot, if necessary

        trap #0         ; Program end
```

This program is stored in file `dapy.s`. It can be executed, showing its execution results in html files and solving data and control hazards through short-circuits and *predict-not-taken* respectively, using the following command:

```
mips-m -s html -d c -c t -f dapy.s
```

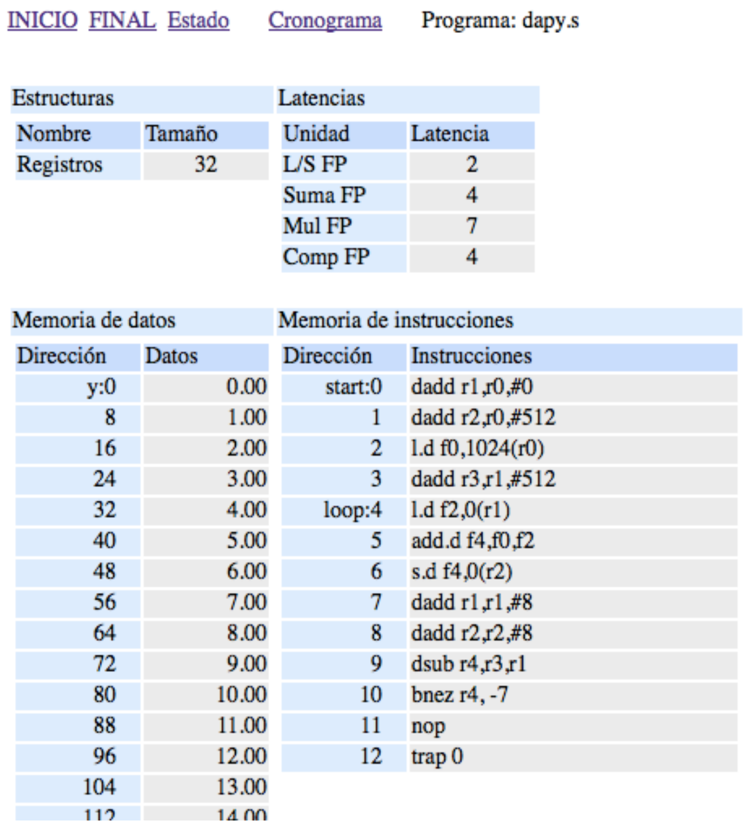
Then, opening file index.html using a web browser will show the configuration of the processor, the initial memory content and also several links enabling a navigation through results:

- INICIO. Shows the processor configuration and the initial memory content.
- FINAL. Shows performance results after executing programs, the processor configuration and the final content of memory. Checking the final memory content enables verifying the proper execution of the program.
- Estado. Shows the state of the execution unit in a given cycle, indicating which instruction is hold by each processor stage. Events and control signal that are activated in the processor (hazards, short-circuits) are also shown. Finally, it is also provided the content of registers and memory at the end of the analyzed cycle. In this page one can find links to the state pages corresponding to 1 or 5 cycles before or after the current one. The page also shows the instructions–time diagram.
- Cronograma. Shows the instructions–time diagram of the program execution. The last cycle shown corresponds to the current cycle. In this page one can find links corresponding to diagrams located at 1 or 5 cycles before or after the current one. The page also shows the state of the processor.

Figure 1 shows the content of the generated *index.html* file. It shows the size of the register file and the latencies related to the multicycle units. First, the size of the register file and the latencies of the multicycle units are shown. It is also reported the initial content of the data and instructions memory zone.

Following the link FINAL will open file *final.html*. Figure 2 shows its content in our example. First, the execution performance results are reported: execution time, executed instructions, CPI, floating point operations and floating point operations per cycle. Then, the configuration of the processor is shown and it is reported the final content of the data and instructions memory. In the right side of the figure one can see that, the resulting array starts at memory location z, thus enabling the verification of the program correctness.

If the link Estado is followed, file *estadoXXX.html* will be opened, where XXX represents the execution cycle, starting at "001". Figure 3 shows its content for cycle 23 of our example. First, they are shown the links to pages *index.html* and *final.html*. Links to files representing the state of the computer 5 cycles before ([-5]), the cycle before ([-1]), the following cycle ([+1]) and 5 cycles after ([+5]) are also available. It is also shown a link to the instructions–time diagram until the current cycle (Crono). Then, execution unit stages are shown, indicating which instruction is in each stage. Empty stages held the equivalent to a hop instruction (-nop-). Since integer and floating point registers files are separated, it is possible to have until one integer and one floating point instruction in the WB stage. Control signals activated when a hazard is detected are also shown. The short-circuits



INICIO FINAL Estado Cronograma Programa: dapy.s

Estructuras		Latencias	
Nombre	Tamaño	Unidad	Latencia
Registros	32	L/S FP	2
		Suma FP	4
		Mul FP	7
		Comp FP	4

Memoria de datos		Memoria de instrucciones	
Dirección	Datos	Dirección	Instrucciones
y:0	0.00	start:0	dadd r1,r0,#0
8	1.00	1	dadd r2,r0,#512
16	2.00	2	l.d f0,1024(r0)
24	3.00	3	dadd r3,r1,#512
32	4.00	loop:4	l.d f2,0(r1)
40	5.00	5	add.d f4,f0,f2
48	6.00	6	s.d f4,0(r2)
56	7.00	7	dadd r1,r1,#8
64	8.00	8	dadd r2,r2,#8
72	9.00	9	dsub r4,r3,r1
80	10.00	10	bnez r4, -7
88	11.00	11	nop
96	12.00	12	trap 0
104	13.00		
112	14.00		

Figura 1: Content of file *index.html*

applied also does. Then, one can see the content of integer registers (R0 to R31), floating point registers (F0 to F31) and the floating point state register (FPSR). Finally, the content of the data memory is shown. Files containing the state of the processor enable a step by step execution of the program. This is how the assembler code that written by anyone can be debugged.

Following the link Cronograma will open the file *cronoXXX.html*, where XXX represent the execution cycle, starting at "001". Figure 4 shows its content for cycle 23 in our example. As can be seen, it is shown the instructions–time diagram corresponding to the execution of the program until the considered cycle. One can also move to one or five cycles before and after the current one. In addition, the state of the current cycle (link Estado) can be analyzed or one can directly go to the state corresponding to one of the cycles (using the links in the upper part of the instructions–time diagram).

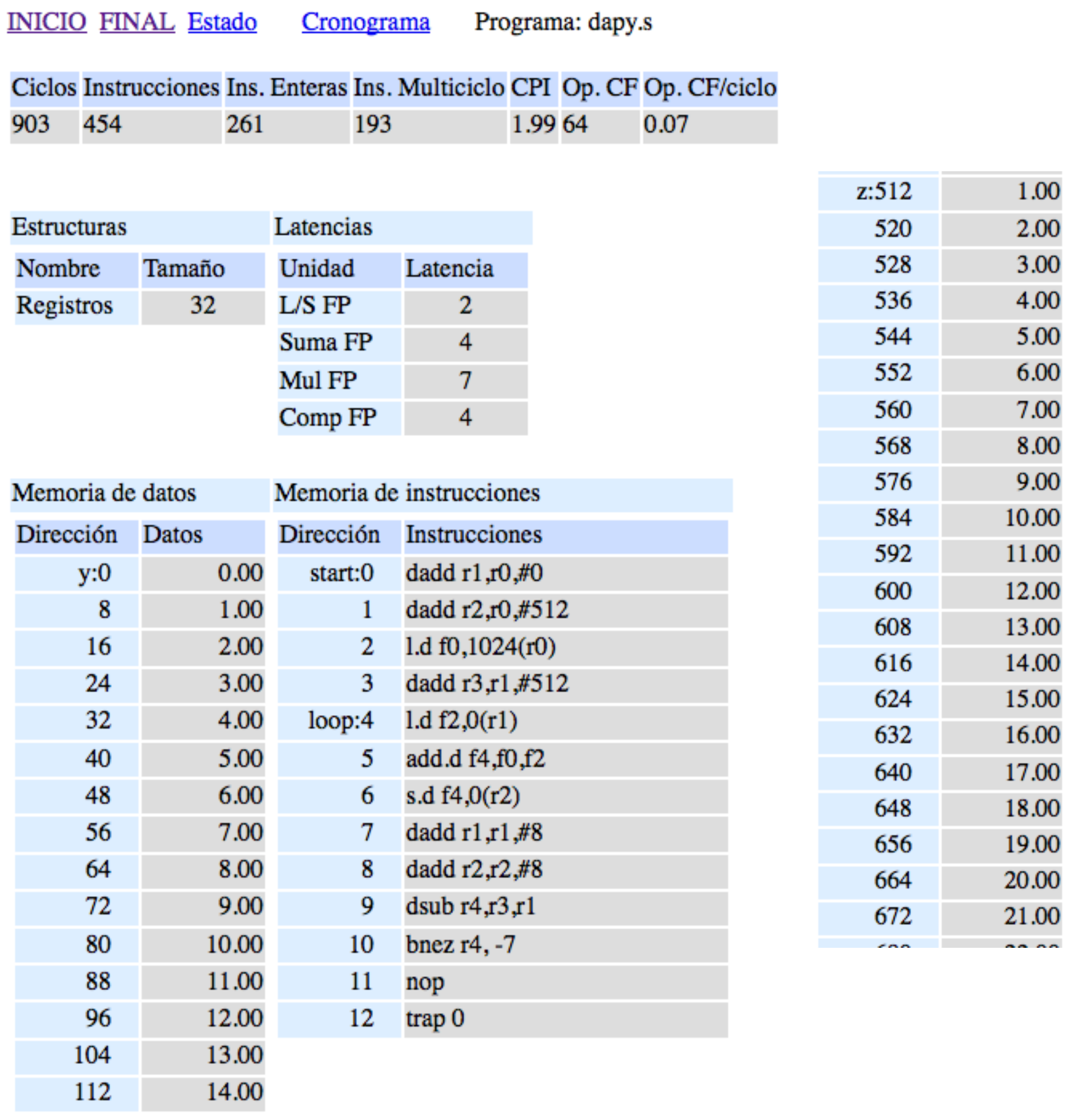
After executing the program, check that it has been stored in the address labelled as `z` a 64 data array with the expected content. Note the execution time of the program and the resulting CPI.

Program modification using static instructions scheduling

1. *Loop unrolling*

   Basically, the *loop unrolling* technique replicates the base code of a loop several times, decreasing the number of resulting iterations.

   In our example, since the maximum number of stalls required to solve the RAW



INICIO FINAL Estado Cronograma Programa: dapy.s

Ciclos	Instrucciones	Ins. Enteras	Ins. Multiciclo	CPI	Op. CF	Op. CF/ciclo
903	454	261	193	1.99	64	0.07

Estructuras	
Nombre	Tamaño
Registros	32

Latencias	
Unidad	Latencia
L/S FP	2
Suma FP	4
Mul FP	7
Comp FP	4

Memoria de datos	
Dirección	Datos
y:0	0.00
8	1.00
16	2.00
24	3.00
32	4.00
40	5.00
48	6.00
56	7.00
64	8.00
72	9.00
80	10.00
88	11.00
96	12.00
104	13.00
112	14.00

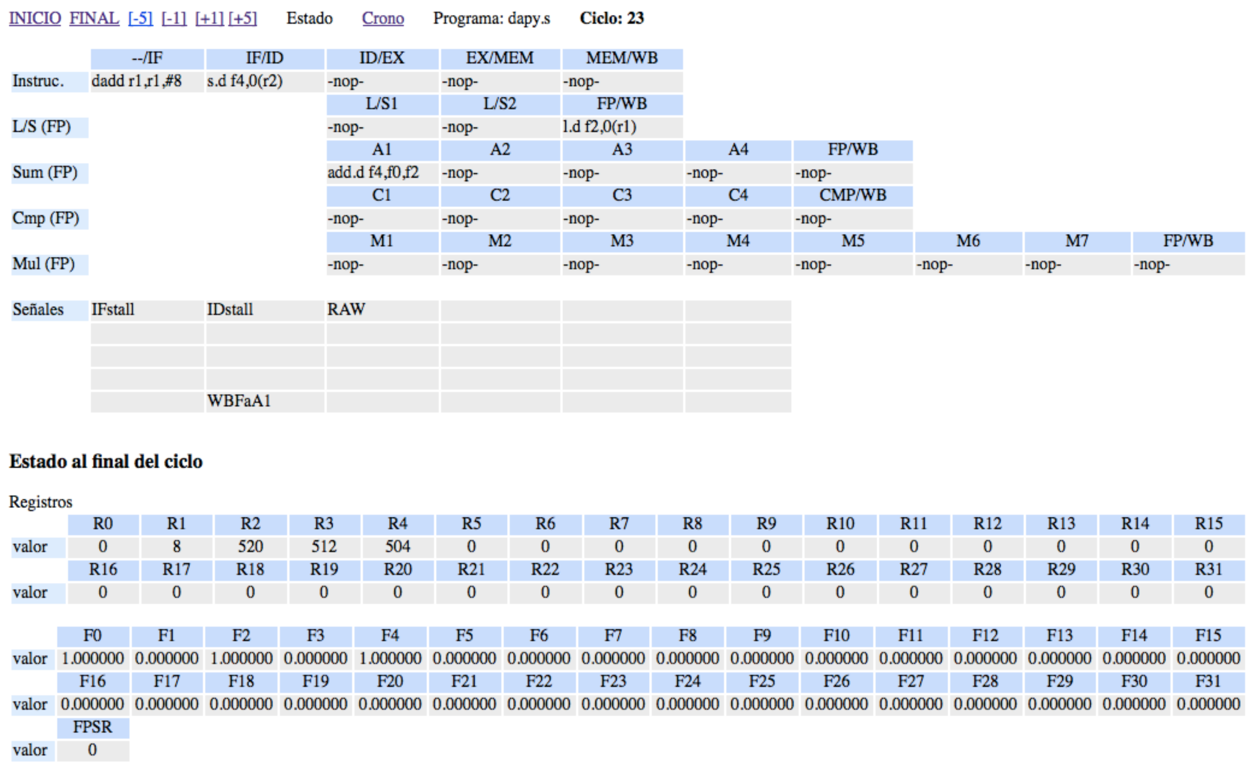
Memoria de instrucciones	
Dirección	Instrucciones
start:0	dadd r1,r0,#0
1	dadd r2,r0,#512
2	l.d f0,1024(r0)
3	dadd r3,r1,#512
loop:4	l.d f2,0(r1)
5	add.d f4,f0,f2
6	s.d f4,0(r2)
7	dadd r1,r1,#8
8	dadd r2,r2,#8
9	dsub r4,r3,r1
10	bnez r4, -7
11	nop
12	trap 0

z:512	1.00
520	2.00
528	3.00
536	4.00
544	5.00
552	6.00
560	7.00
568	8.00
576	9.00
584	10.00
592	11.00
600	12.00
608	13.00
616	14.00
624	15.00
632	16.00
640	17.00
648	18.00
656	19.00
664	20.00
672	21.00

Figura 2: Content of file *final.html*

hazard is 3 cycles, the code of the loop $\vec{Z} = a + \vec{Y}$ must be replicated 4 times, as shown hereafter. It must be noted that some registers have been renamed to delete name dependences:

```
start:
        dadd r1,r0,y      ; r1 contains the address of y
        dadd r2,r0,z      ; r2 contains the address of z
        l.d f0,a(r0)      ; f0 contains a
        dadd r3,r1,#512   ; 64 elements are 512 bytes
loop:
        l.d f2,0(r1)
        add.d f4,f0,f2
        s.d f4,0(r2)
        l.d f6,8(r1)
        add.d f8,f0,f6
```



INICIO FINAL [-5] [-1] [+1] [+5] Estado Crono Programa: dapy.s **Ciclo: 23**

	--/IF	IF/ID	ID/EX	EX/MEM	MEM/WB			
Instruc.	dadd r1,r1,#8	s.d f4,0(r2)	-nop-	-nop-	-nop-			
			L/S1	L/S2	FP/WB			
L/S (FP)			-nop-	-nop-	l.d f2,0(r1)			
			A1	A2	A3	A4	FP/WB	
Sum (FP)			add.d f4,f0,f2	-nop-	-nop-	-nop-	-nop-	
			C1	C2	C3	C4	CMP/WB	
Cmp (FP)			-nop-	-nop-	-nop-	-nop-	-nop-	
			M1	M2	M3	M4	M5	M6	M7	FP/WB
Mul (FP)			-nop-	-nop-	-nop-	-nop-	-nop-	-nop-	-nop-	-nop-

Señales	IFstall	IDstall	RAW			
		WBFaA1				

Estado al final del ciclo

Registros

	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
valor	0	8	520	512	504	0	0	0	0	0	0	0	0	0	0	0
	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	R31
valor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
valor	1.000000	0.000000	1.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31
valor	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
	FPSR															
valor	0															

Figura 3: Content of file *estado023.html*

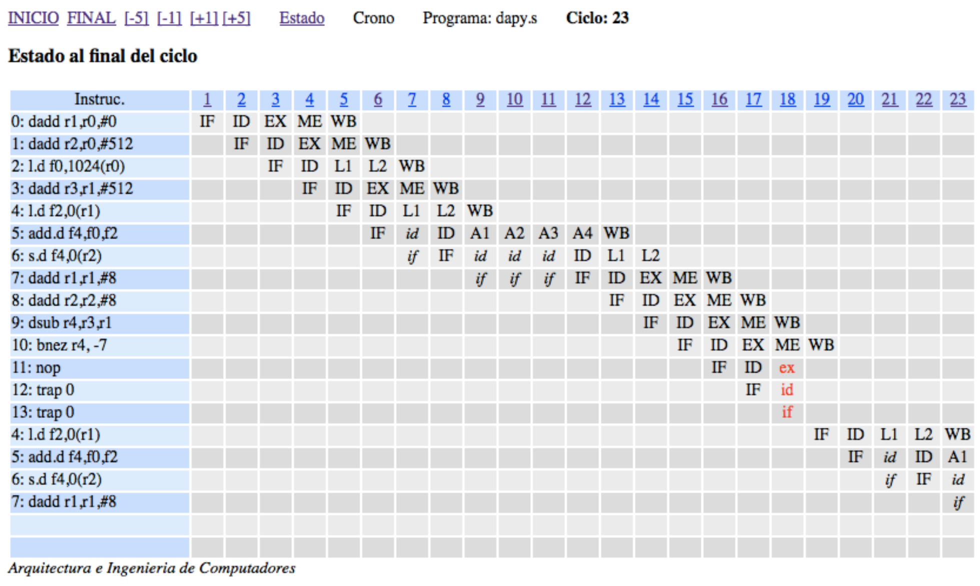
```
s.d f8,8(r2)
l.d f10,16(r1)
add.d f12,f0,f10
s.d f12,16(r2)
l.d f14,24(r1)
add.d f16,f0,f14
s.d f16,24(r2)
dadd r1,r1,#32
dadd r2,r2,#32
dsub r4,r3,r1
bnez r4,loop
nop                ; delay slot, if necessary

trap #0            ; Program end
```

This program is stored in file `dapyu1.s`. Execute this new program:

```
mips-m -s html -d c -c t -f dapyu1.s
```

Check the correctness of the result and note the execution time. Calculate the resulting CPI. Quantify the speedup with respect to the original program.

That code can be easily modified to eliminate all data hazards, and fill the *delay slot* with a useful instruction:



INICIO FINAL [-5] [-1] [+1] [+5] Estado Crono Programa: dapy.s **Ciclo: 23**

Estado al final del ciclo

Instruc.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0: dadd r1,r0,#0	IF	ID	EX	ME	WB																		
1: dadd r2,r0,#512		IF	ID	EX	ME	WB																	
2: l.d f0,1024(r0)			IF	ID	L1	L2	WB																
3: dadd r3,r1,#512				IF	ID	EX	ME	WB															
4: l.d f2,0(r1)					IF	ID	L1	L2	WB														
5: add.d f4,f0,f2						IF	*id*	ID	A1	A2	A3	A4	WB										
6: s.d f4,0(r2)							*if*	IF	*id*	*id*	*id*	ID	L1	L2									
7: dadd r1,r1,#8									*if*	*if*	*if*	IF	ID	EX	ME	WB							
8: dadd r2,r2,#8													IF	ID	EX	ME	WB						
9: dsub r4,r3,r1														IF	ID	EX	ME	WB					
10: bnez r4, -7															IF	ID	EX	ME	WB				
11: nop																IF	ID	ex					
12: trap 0																	IF	id					
13: trap 0																		if					
4: l.d f2,0(r1)																			IF	ID	L1	L2	WB
5: add.d f4,f0,f2																				IF	*id*	ID	A1
6: s.d f4,0(r2)																					*if*	IF	*id*
7: dadd r1,r1,#8																							*if*

Arquitectura e Ingenieria de Computadores

Figura 4: Content of file *crono023.html*

```
start:
        dadd r1,r0,y      ; r1 contains the address of y
        dadd r2,r0,z      ; r2  contains the address of z
        l.d f0,a(r0)      ; f0 contains a
        dadd r3,r1,#512   ; 64 elements are 512 bytes
loop:
        l.d f2,0(r1)
        l.d f6,8(r1)
        l.d f10,16(r1)
        l.d f14,24(r1)
        add.d f4,f0,f2
        add.d f8,f0,f6
        add.d f12,f0,f10
        add.d f16,f0,f14
        s.d f4,0(r2)
        s.d f8,8(r2)
        s.d f12,16(r2)
        s.d f16,24(r2)
        dadd r1,r1,#32
        dsub r4,r3,r1
        bnez r4,loop
        dadd r2,r2,#32

        trap #0          ; Program end
```

This program is stored in file `dapyu.s`. Execute the program considering that control hazards are managed using the delayed branch technique with a *branch delay slot* of 1 instruction:

```
mips-m -s html -d c -c 1 -f dapyu.s
```

Check the correctness of the result and note the execution time. Calculate the resulting CPI. Quantify the speedup with respect to the original program.

2. *Software pipelining.*

   Basically, *software pipelining* replaces the original loop by another one where executed instructions belong to different iterations of the original loop, which eliminates hazards.

   The modified version of the loop $\vec{Z} = a + \vec{Y}$ is the following one:

```
start:
        dadd r1,r0,y      ; r1 contains the address of y
        dadd r2,r0,z      ; r2 contains the address of z
        l.d f0,a(r0)      ; f0 contains a
        dadd r3,r1,#512   ; 64 elements are 512 bytes
prepara:
        l.d f2,0(r1)
        add.d f4,f0,f2
        l.d f2,8(r1)
        dadd r1,r1,#16

loop:
        s.d f4, 0(r2)
        add.d f4,f0,f2
        l.d f2,0(r1)
        dadd r1,r1,#8
        dsub r4,r3,r1
        bnez r4,loop
        dadd r2,r2,#8

resto:
        s.d f4, 0(r2)
        add.d f4,f0,f2
        s.d f4, 8(r2)

        trap #0          ; Program end
```

   This program is stored in file `dapysp.s`. Execute the following command:

```
mips-m -s html -d c -c 1 -f dapysp.s
```

   Check the correctness of the result and note the execution time. Calculate the resulting CPI. Quantify the speedup with respect to the original program.

Development of a new program

In this section of the lab we will assume that latencies for the adder and the multiplier are of 2 and 4 cycles respectively (options `-a 2 -m 4` for running the simulator).

1. Write the MIPS code for the execution of the operation $\vec{Z} = a * \vec{X} + \vec{Y}$ (DAXPY loop). Consider that each array to process contains 64 floating point numbers.

   Take as reference the program stored in file `daxpy.s`.

   IMPORTANT: In case of having an error of type "undefined label (etiqueta indefinida) o syntax error", please check whether the error is one of those listed in the Annex A of this lab.

   Execute the program in the simulator. Do not forget to signal the simulator the method to solve hazards (options `-d c -c 1`) and the operator latencies. Evaluate the resulting performance.

2. Apply the *loop unrolling* technique to the developed code, reorganizing the code, when necessary in order to reduce the inserted stalls.

   Use as reference the program developed in section a), by copying it previously to another file (such as `daxpyu.s`). Write the new code and execute it. Do not forget to signal the simulator the method to solve hazards and the operator latencies. Evaluate the resulting performance and compare it with the base version.

   Have stalls been eliminated? If this is not the case, explain the causes of this situation.

Subset of MIPS instructions supported by the simulator

- Load/store

ld Rx, desp(Ry)
sd Rz, desp(Ry)

- Arithmetic, logic and shift

dadd Rx, Ry, Rz	daddi Rx, Ry, Imm
dsub Rx, Ry, Rz	dsubi Rx, Ry, Imm
and Rx, Ry, Rz	andi Rx, Ry, Imm
or Rx, Ry, Rz	ori Rx, Ry, Imm
xor Rx, Ry, Rz	xori Rx, Ry, Imm
dsra Rx, Ry, Rz	dsra Rx, Ry, Imm
dsll Rx, Ry, Rz	dsll Rx, Ry, Imm
dsrl Rx, Ry, Rz	dsrl Rx, Ry, Imm

- Comparison:

seq Rx, Ry, Rz	seq Rx, Ry, Imm
sne Rx, Ry, Rz	sne Rx, Ry, Imm
sgt Rx, Ry, Rz	sgt Rx, Ry, Imm
slt Rx, Ry, Rz	slt Rx, Ry, Imm
sge Rx, Ry, Rz	sge Rx, Ry, Imm
sle Rx, Ry, Rz	sle Rx, Ry, Imm

- Conditional branch

bnez Ry, Desp	bc1t Desp
beqz Ry, Desp	bc1f Desp

- Floating point load/store

l.d Fx, desp(Ry)
s.d Fz, desp(Ry)

- Floating point arithmetic

add.d Fx, Fy, Fz
sub.d Fx, Fy, Fz
mul.d Fx, Fy, Fz
div.d Fx, Fy, Fz

- Floating point comparison

c.eq.d Fy, Fz
c.ne.d Fy, Fz
c.lt.d Fy, Fz
c.le.d Fy, Fz
c.gt.d Fy, Fz
c.ge.d Fy, Fz

- Others

nop
trap

Annex A

Most common errors are:

- The file has been edited in Windows and it includes carriage returns '\r' that can be eliminated using the command: tr -d "\r"$< fichero\_original > fichero\_sin\_r$
- .data is missing in the code.
- One of the included labels has been duplicated.
- The adequate instruction to load double precision floating point data is *l.d*. The use of *ld* is not correct in this case.
- The adequate instruction to add *dword* integers is *dadd*. The use of *add* is not correct in this case.
- The adequate instruction to add double precision floating point data is *dadd.d*. The use of *add* is not correct in this case.
- The adequate instruction to substract *dword* integers is *dsub*. The use of *sub* is not correct in this case.
- The adequate instruction to multiply double precision floating point data is *mul.d*. The use of *mult* is not correct in this case.
- The adequate instruction to store double precision floating point data is *s.d*. The use of *sd* is not correct in this case.

PRÁCTICA 4ª: "TOMASULO ALGORITHM: *Issue* AND *Writeback*"

Computer Architecture and Engeneering (3^{rd} year)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Goals:

- Implementing and evaluating the Tomasulo algorithm, an algorithm for dynamic instruction scheduling.

Desarrollo:

The starting point of this lab is the MIPS/OOO simulator with support to floating point instructions and dynamic instruction scheduling with speculation using the Tomasulo algorithm. The simulator accepts input files written in assembler, but it lacks of implementation in the ISSUE and WB stages of the Tomasulo algorithm. It also supports a reduced set of integer instructions and double precision floating point arithmetic and load/store instructions.

This assignment organizes as follows. First, it explains the simulator structure and its related data structures. Then, it details the structure of the dynamic instruction scheduling unit and the pseudo-code of the Tomasulo algorithm. Finally, it proposes a set of exercises.

Structure of the simulator

The MIPS/OOO simulator estructures in the following C-language files:

main.c Main simulator program. It is responsible for reading assembler files and execute the various phases of the algorithm.

main.h It contains all simulator shared variables: operators, reservation stations, read and write buffers, instructions queue, data memory, etc.

tipos.h It contains the definitions of all data structures used in the simulator: operators, reservation stations, read and write buffers, *Reorder Buffer (ROB)*, data memory, etc.

input.lex.l It contains the lexical description of the supported assembler language.

input.yacc.y It contains the grammatical rules for the syntactic analysis of the assembler language.

etiquetas.c, etiquetas.h It contains the code to manage assembler labels.

presentacion.c, presentacion.h It contains functions for provision of results.

prediccion.c It contains functions for branch prediction.

f_busqueda.c It contains the implementation of the IF stage.

f_lanzamiento.c It contains the implementation of the Issue stage of multicycle instructions of the Tomasulo algorithm with speculation. *This file will be modified in this lab.*

f_ejecucion.c It contains the implementation of the execution stage of instructions.

f_transferencia.c It contains the implementation of the transfer of data through the common bus and the writing of information in the ROB (WB) that is carried out by the Tomasulo algorithm with speculation. *This is one of the files to be modified in this lab.*

f_confirmacion.c It contains the implementation of the Commit stage of the Tomasulo algorithm with speculation.

instrucciones.h It contains operation codee of implementes instructions and some utility macros.

Supported instructions

Integer	Floating Point
LD Rx, desp(Ry)	L.D Fx, desp(Ry)
SD Ry, desp(Rx)	S.D Fy, desp(Rx)
DADD Rx, Ry, Rz	ADD.D Fx, Fy, Fz
DSUB Rx, Ry, Rz	SUB.D Fx, Fy, Fz
DADDI Rx, Ry, valor	
DSUBI Rx, Ry, valor	
	MUL.D Fx, Fy, Fz
	DIV.D Fx, Fy, Fz
	C.GT.D Fx, Fy
	C.LT.D Fx, Fy
BEQZ Rx, desp	BC1F desp
BNEZ Rx, desp	BC1T desp
TRAP #N	

Data structures

A description of the data structures used by the simulator (defined in the `tipos.h` file) and their use can be found hereafter.

Basic types

Basic data types are:

```
typedef unsigned char   byte;   /* One byte:: 8 bits */
typedef short           half;   /* Half word: 16 bits */
typedef int32_t         word;   /* One word: 32 bits */
typedef int64_t         dword;  /* One dword: 64 bits */

typedef unsigned long   ciclo_t;

typedef enum {NO=0, SI=1} boolean; /* Logic values */

typedef byte    codop_t;        /* Operation code type */

typedef byte    marca_t;        /* Mark/code type */
```

NOTE: Constant `MARCA_NULA` is defined in file `main.h`. It is used as a null mark for the mark fields of reservation stations.

```
typedef union
{
  dword         i;    /* Integer data */
  double        f;    /* Floating point data */
} valor _t;                       /* Data in use */
```

NOTE: Both integer and double precision floating point data under consideration are 64 bits datatypes. Since certain fields in certain data structures of the simulator enable the use of both datatypes, you must differentiate in each case which type of data are you referring to. In order to do this, use whenever working `valor_t` variables the extension `.i` to indicate that the value in the variable is of integer type and the extension `.f` to indicate that the value is a floating point value. Consider the following example:

```
valor_t val;

val.i= 45;
...
val.f= 57.2;


typedef enum
{
  NONE,
  EX,
  WB
} estado_t;                    /* Reservation station */

typedef enum
{
  NO_SALTA,
  NO_SALTA_UN_FALLO,
  SALTA_UN_FALLO,
  SALTA
} estado_predic_t;             /* 2 bits predictor state */
```

Registers files

The floating point registers file is a vector with elements of type `reg_fp_t`. Fields for each register are: value and mark. Field *bit de bloqueo* has been deleted, since it corresponds to condition `rob != MARCA_NULA`.

```
/*** Banco de registros ********/

typedef struct {
  double        valor;         /* Register value */
  marca_t       rob;           /* Register mark */
} reg_fp_t;
```

The integer registers file is a vector with elements of type `reg_int_t`. Fields for each register are: value and mark. Field *bit de bloqueo* has been deleted, since it corresponds to condition `rob != MARCA_NULA`.

```
typedef struct {
  dword        valor;          /* Register value */
  marca_t       rob;           /* Register mark */
} reg_int_t;
```

Reservation stations

Una estación de reserva está compuesta por elementos del tipo `estacion_t`. Fields of each entry are: busy bit, operation to be carried out, mark and value of the first operand, mark and value of the second operand, memory address, write confirmation bit and the entry of the *reorder buffer* of the destination instruction.

n addition, a field `orden` is added, in order to enable the knowledge of the age of instructions issuing the operations, and a field `PC` that is exclusively used by visualization purposes.

```
typedef struct {
  boolean       ocupado;          /* Busy bit */
  codop_t       OP;               /* Code of the operation to be carried out */

  marca_t       Qj;               /* Mark of first operand. ALU */
  double        Vj;               /* Mark of second operand. ALU */

  marca_t       Qk;               /* Mark of second operand. ALU and write buffer *
  double        Vk;               /* Value of second operand. ALU and read buffer *

  dword         direccion;        /* Memory address. read and write buffer */
  boolean       confirm;          /* Signals whether the write operation has been c

  marca_t       rob;              /* Indicates which is the destination of the oper

  dword         PC;               /*  Memory position of the instruction */
  ulong         orden;            /* Instruction order */

} estacion_t;
```

Reservation stations of the Adder/Substracter and the Multiplier/Divider, and the read and write buffers, will use all the same type of reservation stations of type `estacion_t`. This will easy the simulator programming.

Reorder buffer

The *reorder buffer* is a vector populated with elements of type `reorder_t`. The fields of each entry are: busy bit, operation state, destination of the operation and the exceptions triggered by the operation.

In addition, a field `orden` is used in order to discover the age of the instruction triggering the operation. For the purpose of visualization, a field `PC` containing the address of the instruction is also included.

```
typedef struct {
  boolean       ocupado;          /* Busy bit */
  codop_t       OP;               /* Code of the operation co carry out */

  estado_t      estado;           /* Operation state */

  dword         dest;             /* Destination register, write buffer or destinat
  valor_t       valor;          /* Result of the operation */

  int           prediccion;     /* Signals whether prediction is taken or not  */

  int           excepcion;      /* Signals the existence of any exception during
  dword         PC;             /* Instruction memory position */
  ciclo_t       orden;          /* Instruction order */
} reorder_t;
```

Branch Target Buffer Predictor

The *Branch Target Buffer* es un vector compuesto por elementos del tipo `entrada_btb_t`. Los campos que tiene cada entrada son: dirección de la instrucción de salto almacenada, estado de la predicción, dirección de destino y antigüedad de la última consulta.

```
typedef struct {
    dword               PC;     /* Address of the branch instruction */
    estado_predic_t     estado; /* Predictor state */
    dword               destino; /* Destination address */

    ciclo_t             orden;  /* Age of the last check */
} entrada_btb_t;
```

Additional structures

This section details the structures used for the implementation of the arithmetic and load/store operators, and the common data bus.

The data bus defines in terms of an structure of type `bus_comun_t`. This estructure has three fields: busy line, lines for the transfer of codes and marks, and lines for the transfer of data.

```
typedef struct {
  boolean       ocupado;        /* Busy line */
  marca_t       codigo;         /* Code line */
  double        valor;          /* Data lines */
} bus_comun_t;
```

Each operator defines in terms of a structure of type `operador_t`, which provides the following fields: busy bit, code of the active reservation station, entry of the *reorder buffer*, number of cycles executed by the active operation and the evaluation time of the operator.

```
typedef struct {
  boolean       ocupado;       /* Busy bit */
  int           estacion;       /* Reservation station under use */
  marca_t       codigo;         /* Reorder buffer code */
  int           ciclo;         /* Current cycle of the operation */
  int           Teval;          /* Evaluation time */

  ciclo_t       orden;          /* Instruction order */
} operador_t;
```

Once the operation has been finished, the result is stored in an output register, which will be in charge of feeding the common bus. The output register is of type `reg_operador_t`, and it includes the following fields: busy bit, entry of the *reorder buffer* to which the result belongs and the resulting value finally obtained.

```
typedef struct {
  boolean       ocupado;         /* Busy bit */
  int           estacion;        /* Reservation station that must be free */
  marca_t       codigo;          /* Reorder buffer code */
  valor_t       valor;           /* Results of the operation */

  ciclo_t       orden;           /* Order of the instruction */
} reg_operador_t;
```

The existence of an output register enables the liberation of the operator just after finishing the operation, in stead of doing it at the end of the WB stage of the Tomasulo algorithm with speculation. Such output register must become **free** once the result is written in the data bus.

Structure of the dynamic instruction scheduling unit

The dynamic instruction scheduling unit is defined in terms of:

Floating point register file It contains the floating point registers. It is represented by variable `Rfp` (`main.h`), of type `reg_fp_t []` (`tipos.h`). The number of registers is defined by constant `TAM_REGISTROS` (`main.h`).

Integer register file It contains the integer registers. It is represented by variable `Rint`, of type `reg_int_t []`. The number of registers is defined by constant `TAM_REGISTROS` (`main.h`).

Reorder Buffer It stores issued instruction until they are committed. It is represented by the variable `RB` (`main.h`), of type `reorder_t []` (`tipos.h`). The number of entries is defined by the constant `TAM_REORDER` (`main.h`).

Adder/substractor reservation stations It contains the reservation stations of the add/sub operator. It is represented by variable `RS` (`main.h`), of type `estacion_t []` (`tipos.h`), in the interval [`INICIO_RS_SUMA_RESTA`, `FIN_RS_SUMA_RESTA`]. The number of reservation stations is defined by constant `TAM_RS_SUMA_RESTA` (`main.h`).

Adder/substractor operator It is in charge of performing floating point additions and subtractions . It is represented by variable `Op` (`main.h`) and the entry `OPER_SUMREST` (`Op[OPER_SUMREST]`) (`main.h`), of type `operador_t` (`tipos.h`).

The operation it is **not** pipelined. Its evaluation time is defined by constant `TEVAL_SUMREST` (`main.h`).

Output register of the adder/substractor operator It temporarily stores results of the adder/substractor operator, until its transfer through the bus. It is represented by variable `RegOp` and the entry `OPER_SUMREST` (`RegOp[OPER_SUMREST]`) (`main.h`), of type `reg_operador_t` (`tipos.h`).

Availability of data is signaled by the `ocupado` field of the register. It is **free** once its content is read (WB stage).

Reservation station of the Multiplier/Divider It contains the reservation stations of the multiplier/divider operator. It is represented by variable `RS`, in the rank [`INICIO_RS_MULT_DIV`, `FIN_RS_MULT_DIV`]. The number of reservation stations is provided by the constant `TAM_RS_MULT_DIV` (`main.h`).

Multiplier/Divider operator It is in charge of floating point multiplications and divisions. It is represented by variable
texttttOp and entry `OPER_MULTDIV` (`Op[OPER_MULTDIV]`).

The operador is **not** pipelined. Evaluation time is defined by constant `TEVAL_MULTDIV` (`main.h`).

Output register of the Multiplier/Divider It temporally stores multiplication/division results, until their transfer through the bus. It is represented by variable `RegOp` and the entry `OPER_MULTDIV`
(`RegOp[OPER_MULTDIV]`).

Presence of available data is signaled through the `ocupado` register field. It is **free** once its content is read (WB stage).

Reservation stations for integer operations It contains reservation stations for the integer operator. It is represented by the variable `RS`, in the rank [`INICIO_RS_ENTEROS`, `FIN_RS_ENTEROS`]. The number of available entries is defined by the constant `TAM_RS_ENTEROS` (`main.h`).

Integer Operator It carries out integer operations. It is represented by variable `Op` and the entry `OPER_ENTEROS` (`Op[OPER_ENTEROS]`).

The operator is **not** pipelined. The evaluation time is defined by the constant `TEVAL_ENTEROS`.

Output register of the Integer Operator It temporally saves results produced by the integer operator, until their transfer through the bus. It is represented by the variable `RegOp` and the entry`OPER_ENTEROS` (`RegOp[OPER_ENTEROS]`).

The presence of available data is signaled through the `ocupado` field of the register. You must **free** the register once it is read (WB stage).

Read buffer It contains the reservation stations of the load/store operator for load operations. It is represented by variable `TL` (alias of `RS`), of type `estacion_t []`, in the rank [`INICIO_TAMPON_LECT`, `FIN_TAMPON_LECT`]. The number of buffers is provided by the constant `TAM_TAMPON_LECT` (`main.h`).

Write buffer It contains the reservation stations of the load/store operator for store operations. It is represented by variable `TE` (alias of `RS`), in the rank [`INICIO_TAMPON_ESCR`, `FIN_TAMPON_ESCR`]. The number of buffers is provided by the constant `TAM_TAMPON_ESCR` (`main.h`).

Load/store operator It carries out the data memory read and write operations. It is represented by variable `Op` and the entry `OPER_MEMDATOS` (`Op[OPER_MEMDATOS]`).

The operador is **not** pipelined. Evaluation time is defined by constant `TEVAL_MEMORIA` (`main.h`).

In order to determine which operation is under execution, you must check the value saved in field `estacion`. If such value is within the interval ([`INICIO_TAMPON_ESCR` .. `FIN_TAMPON_ESCR`]), the operation is an store operation. Otherwise, it is a load operation.

Output register of the load/store operator It temporally saves results of the load/store operator until its transfer through the bus. It is represented by variable `RegOp` and the entry `OPER_MEMDATOS` (`RegOp[OPER_MEMDATOS]`).

Presence of available data is signaled through the `ocupado` register field. It is **free** once its content is read (WB stage).

Branch Target Buffer It stores the prediction of executed branches. It is represented by the variable `BTB` (`main.h`), of type `entrada_btb_t []` (`tipos.h`). The number of entries is provided by the constant `TAM_BUFFER_PREDIC` (`main.h`).

Data common bus It is in charge of data transfers among system components. It is represented by variable `BUS` (`main.h`), of type `bus_comun_t` (`tipos.h`).

Pseudo-code for the Tomasulo algorithm

This section introduces the pseudo-code of the *Issue*, *Execution* and *Writeback* stages of the Tomasulo algorithm[1].

- *Issue*

```
// Instruction decoding information:

-ALU: I_OP, I_D, I_S1, I_S2
-LOAD: I_OP, I_D
-STORE: I_OP, I_S2
-BRANCH: I_OP, I_S1, dir, pred

If {s:reservation station or buffer} free and
   {b:entry in the Reorder Buffer} free, then

  // Reorder Buffer

  RB[b].ocupado := YES
  RB[b].op := I_OP
  If {I_OP is ALU ó LOAD}
    RB[b].dest := I_D
  If {I_OP is STORE}
    RB[b].dest := s
  If {I_OP is BRANCH}
    RB[b].dest := dir // Computed by Issue
    RB[b].pred := pred // What the predictor says
  If {I_OP is STORE}
    RB[b].estado := WB // Stores only waits for commitment
  Else
    RB[b].estado := EX

  // Reservation stations or buffers

  RS[s].rob or TL[s].rob := b // Links with the RB entry
  RS[s].ocupado or TL[s].ocupado or TE[s].ocupado := YES
  RS[s].OP or TL[s].OP or TE[s].OP := I_OP
  If {I_OP is STORE}
    TE[s].confirm := NO // Stores must be confirmed

  // Operand 1
  // NOTE: Regs refers to Rfp or Rint
  // depending on the instructions
```

[1]In order to simplify the implementation of the algorithm in the simulator, it is not included the code for the management of the fields related to the memory address of loads and stores

```
    If {I_OP is ALU or BRANCH}
      If NO(Regs[S1].ocupado) then // Read value
        RS[s].Vj := Regs[I_S1].valor
        RS[s].Qj := MARCA_NULA
      Else
        If RB[Regs[S1].rob].estado=WB then // Read RB
          RS[s].Vj := RB[Regs[I_S1].rob].valor
          RS[s].Qj := MARCA_NULA
        Else // Take note of the RB entry
          RS[s].Qj := Regs[I_S1].rob

    // Operand 2

    If {I_OP is ALU or STORE}
      If NO(Regs[S2].ocupado) then // Read value
        RS[s].Vk or TE[s].Vk := Regs[I_S2].valor
        RS[s].Qk or TE[s].Qk := MARCA_NULA
      Else
        If RB[Regs[S2].rob].estado=WB then // Read RB
          RS[s].Vk or TE[s].Vk := RB[Regs[I_S2].rob].valor
          RS[s].Qk or TE[s].Qk := MARCA_NULA
        Else // Take note of the RB entry
          RS[s].Qk or TE[s].Qk := Regs[I_S2].rob

    // Reservation of the Destination Register

    If {I_OP is ALU or LOAD}
      Regs[I_D].rob := b // Link with the entry of the RB
```

- *Execution*

```
  For {each operator} do
    If {reservation stations with all operands available} then
      Select_one()
      Operation():
         -ALU: Operation in the ALU
         -LOAD/STORE: Memory access
         -SALTO: Computation of Branch condition
      Free_Operator()
```

- *Writeback*

```
  If {there is an operator with results available} then

    // Write the results
```

```
Place data into the common bus
Place code into the common bus

// Lectura de resultados

For {s: reservation station} do
  // Operand 1
  If RS[s].Qj=código then
    RS[s].Vj := dato // read data from the bus
    RS[s].Qj := MARCA_NULA // delete the mark

  // Operand 2
  If RS[s].Qk=código then
    RS[s].Vk := dato // read data from the bus
    RS[s].Qk := MARCA_NULA // delete the mark

For {s: write buffer} do
  // Operando 1
  If TE[s].Qj=código then
    TE[s].Vj := dato // read data from the bus
    TE[s].Qj := MARCA_NULA // delete the mark

  // Operando 2
  If TE[s].Qk=código then
    TE[s].Vk := dato // read data from the bus
    TE[s].Qk := MARCA_NULA // delete the mark

// Reorder Buffer

RB[código].valor := dato // Copy to the RB
RB[código].estado := WB // Ready for Commit

// Free the reservation station
RS[RegOp[operador].estacion].ocupado= NO;
```

Exercices

1. Implementation of the Tomasulo algorithm.

   Once you are familiar with data structures and the structure of the simulator, implement *Issue* and *WB* stages of the Tomasulo algorithm for arithmetic operations.

   These stages will be implemented into functions `fase_FP_ISS` (see file`f_lanzamiento.c`) and `fase_FP_WB` (see file `f_transferencia.c`), respectively. There exist a previous structure in such functions that is reported in appendix A.

   You can edit files using any of the available editors: `vi`, `[x]emacs` o `nedit` (WordPad-like editor).

The simulador `mips-ooo` can be compiled using the command `make` in the directory where sources and the `Makefile` file are available.

2. Check the behavior of the Tomasulo algorithm.

   Once implemented and compiled the Tomasulo algorithm, you will check its behavior using the following examples:

   a) Example containing file `ejemplo.s`.

   ```
   .data                ; Data memory starts here
   a: .double 10.5
   b: .double 2
   c: .double 20

   s1: .space 8
   s2: .space 8

   .text                ; Code starts here

   l.d f0, a(r0) ; Load a
   l.d f1, b(r0) ; Load b
   l.d f2, c(r0) ; Load c
   add.d f4, f0, f1     ; t1= a + b
   mul.d f5, f2, f4     ; t2= c * t1
   s.d f4, s1(r0) ; Store t1
   s.d f5, s2(r0) ; Store t2

   trap 0 ; End of the program
   ```

   Execute the simulator using the command:

   ```
   mips-ooo -t ejemplo.sign -f ejemplo.s
   ```

   The command will generare a file in **html** format for each cycle containing the information of the state of the machine that can be visualized using any web client. File `ejemplo.sign` contains a summary of the states of the processor corresponding to the correct execution of file `ejemplo.s`. In case of a difference with such file, the simulator will indicate in which cycle is located the existing error. If the state of the data path at this cycle is accessed, it is possible to observe (in red and italic) which fields are incorrect. In case a mark is missing, the sign "??" will be shown.

   The correct behavior of the provided implementation must be check at both temporal and logical levels. To do so, operator latencies must be taken into account (by default 3 cycles for load/store, 4 cycles for add/sub and 7 cycles for mult/div).

   Provide the total execution time (in cycles) of the program.

   b) Check the behavior of the DAXPY ($a\vec{x} + \vec{y}$) loop. File `daxpy.s` contains the assembler code.

The correct behavior of the implementation must be tested with the initial configuration of operators. The summary file that must be used in this case is `daxpy1.sign`:

```
mips-ooo -t daxpy1.sign -f daxpy.s
```

Provide the execution time (in cycles) of the program.

Then, increase the size of the vectors used in the program `daxpy.s` to 64 elements and obtain the execution time in cycles, the CPI and the number of floating point operations per cycle. Since the simulator works correctly, do not generate HTML files, so include the option '`-s`' in the command as follows:

```
mips-ooo -s -f daxpy64.s
```

c) Check the behavior of the `daxpy.s` program using the following configuration:

- 2 read buffers, 2 write buffers, 3 add/sub reservation stations and 3 mult/div reservation stations.

Edit file `main.h` and recopile the simulator. Use a new vector with 8 elements in the daxpy loop (file `daxpy.s`).

Check the effect of reducing the number of available read and write buffers. The file summary that must be used in this case is `daxpy2.sign`:

```
mips-ooo -t daxpy2.sign -f daxpy.s
```

Provide the execution time (in cycle) of the program.

Now, increase the size of the vectors used in the DAXPY loop to 64 elements (file `daxpy64.s`) and obtain the execution time in cycles, the CPI and the number of floating point operations per cycle:

```
mips-ooo -s -f daxpy64.s
```

1. Apendice A

```
/****************************************************************
 *
 * Func: fase_FP_ISS
 *
 * Desc: Implements the ’issue’ stage of the Tomasulo algorithm
 *
 ****************************************************************/

void fase_ISS ()
{
    /*************************************/
    /*  Local variables                */
    /*************************************/

    int  s;
    marca_t b;

    /*************************************/
    /*  Function body                   */
    /*************************************/

    /* Decoding */

#define I_OP IF_ISS_2.IR.codop
#define I_S1 IF_ISS_2.IR.Rfuente1
#define I_S2 IF_ISS_2.IR.Rfuente2
#define I_D IF_ISS_2.IR.Rdestino
#define I_INM IF_ISS_2.IR.inmediato
#define I_PC IF_ISS_2.PC
#define I_ORDEN IF_ISS_2.orden
#define I_EXC IF_ISS_2.excepcion
#define I_PRED IF_ISS_2.prediccion

    /*** Visualization ****/
    PC_ISS= I_PC;
    /*********************/

    /*** If it does not finish correctly, then stop execution */

    if (Control_1.Cancelar) { /* This cycle is cancelled */
        /*** Visualization ****/
        muestra_fase("X", I_ORDEN);
        /*********************/
        return;
    }
    else if (Control_2.Cancelar) {
        return;
    }
    else {
        /*** Visualization ****/
        muestra_fase("I", I_ORDEN);
        /**********************/

        Control_1.Parar= SI;
    } /* endif */

    /*** Look for an empty position in the ROB */

    if (RB_long < TAM_REORDER)
        b= RB_fin;
    else
        return; /* No empty positions in the ROB */

    RB[b].excepcion= EXC_NONE;
    RB[b].prediccion= I_PRED;

    /*** Instruction issue */

    switch (I_OP) {
    case OP_L_D:
        /*** Look for an empty position in the read buffer */
        for (s= INICIO_TAMPON_LECT; s<= FIN_TAMPON_LECT; s++)
            if (!TL[s].ocupado) break;

        if (s > FIN_TAMPON_LECT) return ;
        /* No empty positions in the read buffer */

        /*** Reserve the entry of the ROB */
        RB[b].ocupado= SI;
        RB[b].OP= I_OP;
        RB[b].dest= I_D;
        RB[b].estado= EX;

        /*** Reserve the read buffer */
        TL[s].rob= b;
        TL[s].ocupado= SI;
        TL[s].OP= I_OP;

        /*** Address computation */
        if (Rint[I_S1].rob == MARCA_NULA)
            TL[s].direccion= I_INM + Rint[I_S1].valor;
        else if (RB[Rint[I_S1].rob].estado == WB)
            TL[s].direccion= I_INM + RB[Rint[I_S1].rob].valor.i;

        /*** Reserve the destination register */
        Rfp[I_D].rob= b;

/*** Visualization ***/
```

```
        TL[s].orden= I_ORDEN;
        TL[s].PC= I_PC;
        RB[b].orden= I_ORDEN;
        RB[b].PC= I_PC;

        break;

    case OP_S_D:
        /*** Look for an empty position in the write buffer */

/* INSERT CODE */

        /*** Reserve the entry of the ROB */

/* INSERT CODE */

        /*** Reserve the write buffer */

/* INSERT CODE */

        /*** Compute the address */

/* INSERT CODE */

        /*** Operand 2 ***/

/* INSERT CODE */

/*** Visualization ***/
        TE[s].orden= I_ORDEN;
        TE[s].PC= I_PC;
        RB[b].orden= I_ORDEN;
        RB[b].PC= I_PC;

        break;

    case OP_ADD_D:
    case OP_SUB_D:
        /*** Look for an empty position in the write buffer */

/* INSERT CODE */

        /*** Reserve the entry of the ROB */

/* INSERT CODE */

        /*** Reserve the virtual operator */

/* INSERT CODE */
```

```
        /*** Operand 1 ***/

/* INSERT CODE */

        /*** Operand 2 ***/

/* INSERT CODE */

        /*** Reserve the destination register */

/* INSERT CODE */

/*** Visualization ***/
        RS[s].orden= I_ORDEN;
        RS[s].PC= I_PC;
        RB[b].orden= I_ORDEN;
        RB[b].PC= I_PC;

        break;

    case OP_MUL_D:
    case OP_DIV_D:
        /*** Look for an empty position in the reservation station */

/* INSERT CODE */

        /*** Look for an empty position in the ROB */

/* INSERT CODE */

        /*** Reserve the virtual operator */

/* INSERT CODE */

        /*** Operand 1 ***/

/* INSERT CODE */

        /*** Operand 2 ***/

/* INSERT CODE */

        /*** Reserve the destination register */

/* INSERT CODE */

/*** Visualization ***/
        RS[s].orden= I_ORDEN;
        RS[s].PC= I_PC;
        RB[b].orden= I_ORDEN;
        RB[b].PC= I_PC;

        break;

    default:
        fprintf(stderr, "ERROR (%s:%d): Operacion no implementada\n", __FILE__, _
        exit(1);
        break;
    } /* endswitch */

    /*** The instruction has been correctly issued */

    Control_1.Parar= NO;
    RB_fin= (RB_fin + 1) % TAM_REORDER;
    RB_long++ ;

    return ;

} /* end fase_ISS */


...

/***************************************************************
 *
 * Func: fase_FP_WB
 *
 * Desc: Implements the stage 'WB' of the Tomasulo algorithm
 *
 **************************************************************/

void fase_WB ()
{
    /*************************************/
    /*  Local variables                 */
    /*************************************/

    short i;
    marca_t s;

    ciclo_t orden;
    short operador;

    /*************************************/
    /*  Function body                    */
    /*************************************/

    /*** Look for an operator with available results */

    orden= Cycle;
    operador= 0;

    for (i= 0; i < OPERATORS; i++) {
        if (RegOp[i].ocupado && RegOp[i].orden < orden) {
            operador= i;
            orden= RegOp[i].orden;
} /* endif */
    } /* endif */

    if (orden >= Cycle) return ;  /* No operator available with results ready to u

    /*** Free the output register of the operator */
    RegOp[operador].ocupado= NO;

    /*** Write the results in the Common Data bus */

/* INSERT CODE */

    /*** Read of results */

    /* Reservation stations */

    for (s= INICIO_RS_ENTEROS;
         s<= FIN_RS_ENTEROS; s++) {

/* INSERT CODE */

    } /* endfor */

    for (s= INICIO_RS_SUMA_RESTA;
         s<= FIN_RS_SUMA_RESTA; s++) {

/* INSERT CODE */

    } /* endfor */

    for (s= INICIO_RS_MULT_DIV;
         s<= FIN_RS_MULT_DIV; s++) {

/* INSERT CODE */

    } /* endfor */

    /* Write buffer */

    for (s= INICIO_TAMPON_ESCR;
         s<= FIN_TAMPON_ESCR; s++) {

/* INSERT CODE */
```

```
    } /* endfor */

    /* Reorder buffer */

/* INSERT CODE */

    /*** Free the reservation station */

/* INSERT CODE */

    /*** VISUALIZATION ****/
    muestra_fase("WB", RB[BUS.codigo].orden);
    /**********************/

} /* end fase_WB */
```