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 */
```