PRÁCTICA 5ª:
"ALGORITMO DE TOMASULO: *Issue* Y *Writeback*"

Arquitectura e Ingeniería de Computadores (3º curso)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Objetivos:

- Implementar y evaluar las fases *Issue* y *Writeback* del algoritmo de gestión dinámica de instrucciones conocido como Algoritmo de Tomasulo.

Desarrollo:

Para el desarrollo de la práctica se partirá de un simulador del MIPS (MIPS/OOO), el cual es capaz de aplicar planificación dinámica de instrucciones aplicando el algoritmo de Tomasulo. El simulador acepta como entrada un archivo en lenguaje ensamblador y le falta por implementar parte de las etapas ISSUE y WB del algoritmo de Tomasulo. El simulador posee un conjunto de instrucciones enteras reducido, e instrucciones de coma flotante aritméticas y de carga/almacenamiento de doble precisión.

El presente boletín se organiza como sigue: pseudo-código del algoritmo de Tomasulo, ejercicios a realizar, y un apendice con la explicación de la estructura del simulador, estructuras de datos utilizadas, estructura de la unidad de gestión dinámica de instrucciones, y el cuerpo del código que se debe modificar.

Simulador del Algoritmo de Tomasulo

El apéndice A nos muestra información sobre el simulador del MIPS con ejecución fuera de orden que se va a utilizar a lo largo de esta práctica.

En esta sesión sólo se van a modificar los fichero en C cuyo nombre sigue la estructura `FICHERO_alum.c` y sólo en aquellas secciones donde aparezca el comentario:

```
/* INSERTAR CÓDIGO */
```

A continuación se muestran las instrucciones que tiene implementadas el simulador y el pseudo-código de alguna de las fases del algoritmo de Tomasulo que se deberán implementar/completar por parte del alumno.

Instrucciones implementadas

Las instrucciones implementadas en el simulador se muestran la siguiente tabla:

Enteras	Coma flotante
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	

Pseudo-código del algoritmo de Tomasulo

A continuación se muestra el pseudo-código del algoritmo de Tomasulo, para las fases de *Issue*, *Execution* y *Writeback*.

- *Issue*

```
// Datos provenientes de la decodificación:

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

Si {s:estación de reserva o buffer} libre y
   {b:entrada en el Reorder Buffer} libre

  // Estación de reserva o buffer

  RS[s].ocupado ó LB[s].ocupado ó SB[s].ocupado := SI
  RS[s].OP ó LB[s].OP ó SB[s].OP := I_OP
  RS[s].rob ó LB[s].rob := b // Enlaza con entrada del RB

  // Operandos
  // NOTA: Regs hace referencia a Rfp ó Rint
  //  dependiendo de la instrucción
```

```
// Operando 1

Si {I_OP es ALU, LOAD, STORE o BRANCH}
  Si (Regs[I_S1].rob == MARCA_NULA) // Lee valor
    RS[s].V1 := Regs[I_S1].valor
    RS[s].Q1 := MARCA_NULA
  Sino
    Si RB[Regs[I_S1].rob].completado // Lee RB
      RS[s].V1 := RB[Regs[I_S1].rob].valor
      RS[s].Q1 := MARCA_NULA
    Sino // Anota entrada del RB
      RS[s].Q1 := Regs[I_S1].rob

// Operando 2

Si {I_OP es ALU o STORE}
  Si (Regs[I_S2].rob == MARCA_NULA) // Lee valor
    RS[s].V2 ó SB[s].V2 := Regs[I_S2].valor
    RS[s].Q2 ó SB[s].Q2 := MARCA_NULA
  Sino
    Si RB[Regs[I_S2].rob].completado // Lee RB
      RS[s].V2 ó SB[s].V2 := RB[Regs[I_S2].rob].valor
      RS[s].Q2 ó SB[s].Q2 := MARCA_NULA
    Sino // Anota entrada del RB
      RS[s].Q2 ó SB[s].Q2 := Regs[I_S2].rob

// Desplazamiento, en su caso: LOAD y STORE
Si instr es LOAD o STORE
  LB[s].desplaz ó SB[s].desplaz := I_INM

// Reorder Buffer

RB[b].ocupado := SI
RB[b].op := I_OP
RB[b].completado := NO
Si {I_OP es ALU ó LOAD}
  RB[b].dest := I_D
Si {I_OP es STORE}
  RB[b].dest := s
Si {I_OP es BRANCH}
  RB[b].dest := dir  // La calcula Issue
  RB[b].pred := pred // Lo que indique el predictor

// Reserva del registro destino
Si {I_OP es ALU o LOAD}
  Regs[I_D].rob := b // Enlaza con entrada del RB
```

- *Execution*

```
Para {cada operador} hacer
  Si {hay estaciones o buffers con operandos listos}
    Selecciona_una()
    Operación():
       -ALU: Operación en la UAL
       -BRANCH: Cálculo de la condición de salto
       -LOAD/STORE: Cálculo de dirección
       -LOAD/STORE: Acceso a memoria
    Liberar_Operador()
```

- *WriteBack*

```
Si {hay una estación o buffer s con resultados disponibles}

  // Volcado de resultados al bus

  BUS.valor := RS[s].resultado
  BUS.codigo := RS[s].rob
  BUS.condicion := RS[s].condicion
  // Libera la estación de reserva

  RS[s].ocupado := NO

  // Reorder Buffer

  RB[BUS.codigo].valor := BUS.valor // Copia al RB
  RB[BUS.codigo].completado := SÍ // lista para Commit
  RB[BUS.codigo].condicion := BUS.condicion
  // Lectura de resultados

  Para {s: estación de reserva} hacer
    // Operando 1
    Si RS[s].Q1 = BUS.codigo
      RS[s].V1 := BUS.valor // lee dato del bus
      RS[s].Q1 := MARCA_NULA // borra marca

    // Operando 2
    Si RS[s].Q2 = BUS.codigo
      RS[s].V2 := BUS.valor // lee dato del bus
      RS[s].Q2 := MARCA_NULA // borra marca
```

```
Para {s: buffer de carga} hacer
  // Operando 1
  Si LB[s].Q1 = BUS.codigo
    LB[s].V1 := BUS.valor // lee dato del bus
    LB[s].Q1 := MARCA_NULA // borra marca

Para {s: buffer de almacenamiento} hacer
  // Operando 1
  Si SB[s].Q1 = BUS.codigo
    SB[s].V1 := BUS.valor // lee dato del bus
    SB[s].Q1 := MARCA_NULA // borra marca

  // Operando 2
  Si SB[s].Q2 = BUS.codigo
    SB[s].V2 := BUS.valor // lee dato del bus
    SB[s].Q2 := MARCA_NULA // borra marca
```

Ejercicios a realizar

1. Implementación del algoritmo de Tomasulo.

   Tras familiarizarse con las estructuras de datos y la estructura del simulador, implementar las fases *Issue*[1] y *Writeback* del algoritmo de Tomasulo.

   Dichas fases se implementarán dentro de las funciones `fase_ISS` (ver el fichero `f_lanzamiento_alum.c`) y `fase_WB` (ver el fichero `f_transferencia_alum.c`), respectivamente. Existe una estructura previa en dichas funciones que se muestra en el apendice A.

   Para la edición de los ficheros se puede utilizar cualquiera de los editores disponibles: `vi`, `[x]emacs` o `kate`.

   Para la compilación del simulador `mips-ooo` se debe ejecutar la orden `make` en el directorio donde se encuentran los fuentes y el fichero `Makefile`.

2. Comprobar el funcionamiento del algoritmo de Tomasulo.

   Una vez implementado y compilado el algoritmo de Tomasulo, se comprobará su funcionamiento mediante los siguientes ejemplos:

   a) Ejemplo que contiene el fichero `ejemplo.s`.

```
.data                ; Comienzo de los datos de memoria
a: .double 10.5
b: .double 2
c: .double 20

s1: .space 8
s2: .space 8

.text                ; Comienzo del fragmento de código

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

trap 0 ; Final del programa
```

   Para invocar la ejecución del simulador se utilizará la sintaxis del siguiente ejemplo:

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

[1] Para simplificar la implementación de *Issue*, en esta fase sólo se requiere el código correspondiente a la instrucciones de coma flotante (carga/almacenamiento y aritméticas).

Esta orden generará un fichero en formato **html** por cada ciclo con la información sobre el estado de la máquina, que se puede visualizar mediante un navegador.

El fichero `ejemplo.sign` contiene el resumen de los estados del procesor correspondientes a la ejecución correcta del fichero `ejemplo.s`. En caso de existir alguna diferencia con dicho fichero, el simulador informaría del ciclo en el que se ha producido el error. Si accedemos al estado de la ruta de datos correspondiente a dicho ciclo, podemos observar (en rojo y cursiva) qué campos son incorrectos. En caso de que falte alguna marca, se muestra el signo "??".

Aun así, se debe comprobar su correcto funcionamiento, tanto lógico como temporal. Para ello, se deberán tener en cuenta las latencias de cada uno de los operadores (por defecto 3 ciclos para la carga/almacenamiento, 4 ciclos para la suma/resta y 7 ciclos para la multiplicación/división).

Indica cuál es el tiempo de ejecución total (en ciclos) del programa.

b) Comprobar el funcionamiento del bucle DAXPY ($a\vec{x} + \vec{y}$). El fichero `daxpy.s` contiene el código en ensamblador.

Se deberá comprobar su correcto funcionamiento, con la configuración inicial de los operadores. El fichero resumen utilizado en este caso será `daxpy1.sign`:

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

Indica cuál es el tiempo de ejecución total (en ciclos) del programa.

Seguidamente, aumentar el tamaño de los vectores del programa `daxpy.s` a 64 elementos y obtener el tiempo de ejecución en ciclos, el CPI y el número de operaciones en coma flotante por ciclo. Dado que el simulador ya debe funcionar correctamente, no generaremos ficheros HTML, invocando el simulador con la opción '`-s`':

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

c) Comprobar el funcionamiento de `daxpy.s` con la siguiente modificación:

```
./mips-ooo -t daxpy2.sign -f daxpy.s -l 1:2:c:1:1
```

El parámetro "-l" indica 1 unidad de acceso a memoria, de 2 ciclos de latencia, convencional, 1 buffer de carga y 1 buffer de almacenamiento.

Observa la inserción de ciclos de parada debido a la falta de espacio en las estaciones de reserva.

Indica cuál es el tiempo de ejecución total (en ciclos) del programa.

Seguidamente, aumentar el tamaño de los vectores del bucle DAXPY a 64 elementos (fichero `daxpy64.s` y obtener el tiempo de ejecución en ciclos, el CPI y el número de operaciones en coma flotante por ciclo:

```
./mips-ooo  -f daxpy64.s -l 1:2:c:1:1
```

A. Estructura del simulador

El simulador MIPS/OOO está compuesto de los siguientes ficheros en lenguaje C:

main.c Programa principal del simulador. Encargado de leer el ensamblador, ejecutar las distintas fases del algoritmo e imprimir los resultados.

main.h Contiene todas las variables compartidas del simulador: operadores, estaciones de reserva, buffers de carga y almacenamiento, *Reorder Buffer (ROB)*, memoria de datos, etc.

tipos.h Contiene las definiciones de todas las estructuras de datos utilizadas en el simulador: operadores, estaciones de reserva, buffers de carga y almacenamiento, ROB, memoria de datos, etc.

input.lex.l Contiene la descripción léxica del lenguaje ensamblador utilizado.

input.yacc.y Contiene las reglas gramaticales para el análisis sintáctico del lenguaje ensamblador.

etiquetas.c, etiquetas.h Contiene el manejo de etiquetas del ensamblador.

presentacion.c, presentacion.h Contiene las funciones para la impresión de los resultados.

prediccion.c Contiene las funciones para la predicción de saltos.

f_busqueda.c Contiene la implementación de la fase de búsqueda de instrucciones (IF).

f_lanzamiento_alum.c Contiene la implementación de la fase de lanzamiento de instrucciones multiciclo (Issue) del algoritmo de Tomasulo con especulación. *Este fichero se deberá modificar.*

f_ejecucion.c Contiene la implementación de la fase de ejecución de las instrucciones.

f_transferencia_alum.c Contiene la implementación de la fase de transferencia por el bus común de datos y escritura en el ROB (WB) del algoritmo de Tomasulo con especulación. *Este fichero se deberá modificar.*

f_confirmacion.c Contiene la implementación de la fase de confirmación (Commit) del algoritmo de Tomasulo con especulación.

instrucciones.h Contiene los códigos de operación de las instrucciones implementadas y algunas macros de utilidad.

B. Estructuras de datos

A continuación se describirán las estructuras de datos utilizadas (que se encuentran en el fichero `tipos.h`) y su utilización.

B.1. Tipos básicos

Los tipos básicos utilizados son:

```
typedef int8_t byte; /* Un byte: 8 bits */
typedef int16_t half; /* Media palabra: 16 bits */
typedef int32_t word; /* Una palabra: 32 bits */
typedef int64_t dword; /* Una doble palabra: 64 bits */

typedef uint32_t ciclo_t;

typedef enum {NO=0, SI=1} boolean; /* Valor lógico */

typedef byte    codop_t;        /* Código de operación */

typedef byte    marca_t;        /* Tipo marca/código */
```

NOTA: La constante `MARCA_NULA`, definida en el fichero `main.h`, se utiliza como marca nula para los campos de marca de las estaciones de reserva.

```
typedef union
{
  dword         int_d;    /* Datos enteros */
  double        fp_d;     /* Datos coma flotante */
} valor_t;                /* Dato utilizado */
```

NOTA: Al manejarse dos tipos de datos (enteros y coma flotante de doble precisión, ambos de 64 bits) y al existir algunos campos de ciertas estructuras que permiten ambos tipos, habrá que diferenciar en cada caso que tipo de datos se está utilizando. Así pues, para realizar esta diferenciación en aquellos casos que corresponda (tipo `valor_t`), se utilizarán las extensiones `.int_d` para enteros y `.fp_d` para datos en coma flotante respectivamente. Por ejemplo:

```
valor_t val;

val.int_d= 45;
...
val.fp_d= 57.2;


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

B.2. Bancos de registros

Los bancos de registros son vectores compuestos por elementos del tipo `valor_t`. Los campos que tiene cada registro son: `valor` y `rob`.

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

typedef struct {
  valor_t       valor;          /* Valor del registro */
  marca_t       rob;            /* Marca del registro */
} reg_t;
```

B.3. Estaciones de reserva

Una estación de reserva está compuesta por elementos del tipo `estacion_t`. Los campos que tiene cada entrada son: bit de ocupado, operación a realizar, marca del primer operando, valor del primer operando, marca del segundo operando y valor del segundo operando, dirección memoria, bit de confirmación de escritura y entrada en el *reorder buffer* de la instrucción destinataria.

Adicionalmente, la estación de reserva tiene un campo `resultado` que contiene el valor del resultado obtenido tras realizar la operación. La existencia de este campo permite liberar el operador justo al acabar la operación, y no al final de la fase de transferencia del algoritmo de Tomasulo con especulación.

Finalmente, se añade un campo `orden`, que permite averiguar la antigüedad de la instrucción que lanzo dicha operación, y un campo `PC`, para uso exclusivo de las funciones de visualización.

```
typedef struct {
  boolean       ocupado;        /* Bit de ocupado */
  codop_t       OP;             /* Código de operación a realizar */

  marca_t       Q1;             /* Marca del primer operando. ALU */
  valor_t       V1;             /* Valor del primer operando. ALU */

  marca_t       Q2;             /* Marca del segundo operando. ALU y SB */
  valor_t       V2;             /* Valor del segundo operando. ALU y SB */

  dword         direccion;      /* Dirección de acceso a memoria. LB y SB */
  dword         desplazamiento; /* Desplazamiento en el acceso a memoria. LB y SB
  boolean       confirm;        /* Indica si la operación de almacenamiento
                                   ha sido confirmada (commit). SB */

  marca_t       rob;            /* Indica para quien es la operación.  */

  valor_t       resultado;      /* Resultado de la operación */
  boolean       condicion;      / *  Condicion de salto. RS enteras  */
  dword          PC;             /* Posición de memoria de la instrucción */
  ciclo_t        orden;          /* Orden de la instrucción */
  ...
} estacion_t;
```

Las estaciones de reserva de enteros, del sumador/restador y del multiplicador/divisor, y los buffers de carga y de almacenamiento, usarán el mismo tipo de estación de reserva (`estacion_t`), para facilitar la programación del simulador.

B.4. Reorder buffer

El *reorder buffer* es un vector compuesto por elementos del tipo `reorder_t`. Los campos que tiene cada entrada son: bit de ocupado, operación a realizar, estado de la operación, destino de la operación, resultado de la operación y excepciones producidas por la instrucción.

Adicionalmente, se añade un campo `orden`, que permite averiguar la antigüedad de la instrucción que lanzo dicha operación, para uso exclusivo de las funciones de visualización, un campo `PC`, que contiene la dirección de la instrucción.

```
typedef struct {
  boolean       ocupado;        /* Bit de ocupado */
  codop_t       OP;             /* Código de operación a realizar */

  boolean       completado;     /* Estado de la operación */

  dword         dest;           /* Registro destino, SB o dirección dest. */
  valor_t       valor;          /* Resultado de la operación */

  int           prediccion;     /* Indica si se ha predicho que se saltaba o no *

  int           excepcion;      /* Indica si se ha producido alguna
                                   excepción al ejecutar está
                                   instrucción */
  dword         PC;             /* Posición de memoria de la instrucción */
  ciclo_t       orden;          /* Orden de la instrucción */
} reorder_t;
```

B.5. El predictor *Branch Target Buffer*

El *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;     /* Dirección de la instrucción de salto */
    estado_predic_t     estado; /* Estado del predictor */
    dword               destino;/* Dirección de destino */

    ciclo_t             orden;  /* Antiguedad de la última consulta */
} entrada_btb_t;
```

B.6. Estructuras adicionales

Se detalla a continuación las estructuras utilizadas para la implementación de los operadores aritméticos y de carga/almacenamiento, y el bus común de transferencia.

El bus de datos se compone de una estructura del tipo `bus_comun_t`, que se compone de dos campos: lineas para la transferencia de los códigos/marcas, y lineas para la transferencia de los datos.

```
typedef struct {
  marca_t       codigo;         /* Lineas para los códigos */
  valor_t       valor;          /* Lineas de datos */
} bus_comun_t;
```

Cada uno de los operadores se compone de una estructura del tipo `operador_t`, cuyos campos son: bit de ocupado, código de la estación activa, entrada del *reorder buffer*, número de ciclos ejecutados de la operación activa y tiempo de evaluación del operador.

```
typedef struct {
  boolean       ocupado;        /* Bit de ocupado */
  int           estacion;       /* Estación de reserva en uso */
  marca_t       codigo;         /* Código del reorder buffer */
  int           ciclo;          /* Ciclo actual de la operación */
  int           Teval;          /* Tiempo de evaluación */

  ciclo_t       orden;          /* Orden de la instrucción */
} operador_t;
```

C. Estructura de la unidad de gestión dinámica de instrucciones

La unidad de gestión dinámica está compuesta por los siguientes elementos:

Estructura	Variable	Tipo	Tamaño
Regs. FP	`Rfp`	`reg_t []`	`TAM_REGISTROS`
Regs. Enteros	`Rint`	`reg_t []`	`TAM_REGISTROS`
Reorder Buffer	`RB`	`reorder_t []`	`TAM_REORDER`
Est. Reserva S/R	`RS`	`estacion_t []`	`TAM_RS_SUMREST`: `INICIO_RS_SUMREST` .. `FIN_RS_SUMREST`
Est. Reserva M/D	`RS`	`estacion_t []`	`TAM_RS_MULTDIV`: `INICIO_RS_MULTDIV` .. `FIN_RS_MULTDIV`
Est. Reserva Enteros	`RS`	`estacion_t []`	`TAM_RS_ENTEROS`: `INICIO_RS_ENTEROS` .. `FIN_RS_ENTEROS`
Buffer Carga	`LB`	`estacion_t []`	`TAM_BUFFER_CARGA`: `INICIO_BUFFER_CARGA` .. `FIN_BUFFER_CARGA`
Buffer Almacenamiento	`SB`	`estacion_t []`	`TAM_BUFFER_ALMACEN`: `INICIO_BUFFER_ALMACEN` .. `FIN_BUFFER_ALMACEN`
Operadores	`Op`	`operador_t []`	`INICIO_OP_ENTEROS` .. `FIN_OP_ENTEROS` `INICIO_OP_SUMREST` .. `FIN_OP_SUMREST` `INICIO_OP_MULTDIV` .. `FIN_OP_MULTDIV` `INICIO_OP_DIRECCIONES` .. `FIN_OP_DIRECCIONES` `INICIO_OP_MEMDATOS` .. `FIN_OP_MEMDATOS`
Branch Target Buffer	`BTB`	`entrada_btb_t []`	`TAM_BUFFER_PREDIC`
Bus común	`BUS`	`bus_comun_t`	

D. Fuentes

```
/*************************************************************
 *
 * Func: fase_FP_ISS
 *
 * Desc: Implementa la fase 'issue' del algoritmo de Tomasulo
 *
 *************************************************************/

void fase_ISS_alum() {
    /*************************************/
    /*  Variables locales                */
    /*************************************/

    int s;
    marca_t b;

    /*************************************/
    /*  Cuerpo función                   */
    /*************************************/

    /* Decodificación */

#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

    /*** VISUALIZACIÓN ****/
    PC_ISS = I_PC;
    /**********************/

    /*** Si no sale correctamente hay que parar */

    if (Control_2.Cancelar || IF_ISS_2.ignorar) {
        Control_1.Parar = NO;
        return;
    } else if (Control_1.Cancelar || IF_ISS_2.cancelar) { /* Este ciclo está cance
        /*** VISUALIZACIÓN ****/
        muestra_fase("X", I_ORDEN);
        /**********************/
        return;
    } else if (Control_1.Parar) {
        /*** VISUALIZACIÓN ****/
```

```
        muestra_fase("i", I_ORDEN);
        /**********************/

        /* Si la instrucción anterior del mismo grupo se ha parado,
         * entonces esta instrucción ni siquiera se intenta */

        return;
    } else {
        /*** VISUALIZACIÓN ****/
        muestra_fase("I", I_ORDEN);
        /**********************/

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

    /*** Busca un hueco en la cola */

    if (RB_long < TAM_REORDER) {
        b = RB_fin;
    } else {
        return; /* No hay huecos en el ROB */
    }

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

    /*** Lanza la instruccion */

    switch (I_OP) {
        case OP_L_D:
            /*** Busca un hueco en el \emph{buffer} de carga */
            for (s = INICIO_BUFFER_CARGA; s <= FIN_BUFFER_CARGA; s++)
                if (!LB[s].ocupado) break;

            if (s > FIN_BUFFER_CARGA) return;
            /* No hay sitio en la estación de reserva */

            /*** Reserva el \emph{buffer} de carga */
            LB[s].ocupado = SI;
            LB[s].OP = I_OP;
            LB[s].rob = b;

            /*** Operando 1 (en Rint) ***/
            if (Rint[I_S1].rob == MARCA_NULA) {
                LB[s].V1 = Rint[I_S1].valor;
                LB[s].Q1 = MARCA_NULA;
            } else if (RB[Rint[I_S1].rob].completado) {
                LB[s].V1 = RB[Rint[I_S1].rob].valor;
                LB[s].Q1 = MARCA_NULA;
            } else {
```

```
                LB[s].Q1 = Rint[I_S1].rob;
            } /* endif */

            /*** Operando 2 ***/
            LB[s].Q2 = MARCA_NULA;

            /*** Desplazamiento */
            LB[s].desplazamiento = I_INM;

    /*** Reserva la entrada del ROB */
            RB[b].ocupado = SI;
            RB[b].OP = I_OP;
            RB[b].dest = I_D;
            RB[b].completado = NO;

            /*** Reserva del registro destino */
            Rfp[I_D].rob = b;

    /*** VISUALIZACION ***/
            LB[s].estado = PENDIENTE;
            LB[s].orden = I_ORDEN;
            LB[s].PC = I_PC;
            RB[b].orden = I_ORDEN;
            RB[b].PC = I_PC;

            break;
        case OP_S_D:
            /*** Busca un hueco en el buffer de almacenamiento */

/* INSERTAR CÓDIGO */

            /*** Reserva el buffer de almacenamiento */

        /* INSERTAR CÓDIGO */

            /*** La instrucción de almacenamiento se debe confirmar */
            SB[s].confirm = NO;

            /*** Operando 1 (en Rint) ***/

/* INSERTAR CÓDIGO */

            /*** Operando 2 (en Rfp) ***/

/* INSERTAR CÓDIGO */

            /*** Desplazamiento */

/* INSERTAR CÓDIGO */
```

```
    /*** Reserva la entrada del ROB */

/* INSERTAR CÓDIGO */

    /*** VISUALIZACION ***/
            SB[s].estado = PENDIENTE;
            SB[s].orden = I_ORDEN;
            SB[s].PC = I_PC;
            RB[b].orden = I_ORDEN;
            RB[b].PC = I_PC;


            break;
        case OP_ADD_D:
        case OP_SUB_D:
            /*** Busca un hueco en la estación de reserva */

/* INSERTAR CÓDIGO */

            /*** Reserva el operador virtual */

/* INSERTAR CÓDIGO */

            /*** Operando 1 (en Rfp) ***/

/* INSERTAR CÓDIGO */

            /*** Operando 2 (en Rfp) ***/

/* INSERTAR CÓDIGO */

            /*** Reserva la entrada del ROB */

/* INSERTAR CÓDIGO */

            /*** Reserva del registro destino */

/* INSERTAR CÓDIGO */

    /*** VISUALIZACION ***/
            RS[s].estado = PENDIENTE;
            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:
```

```
            /*** Busca un hueco en la estación de reserva */

/* INSERTAR CÓDIGO */

            /*** Reserva el operador virtual */

/* INSERTAR CÓDIGO */

            /*** Operando 1 ***/

/* INSERTAR CÓDIGO */

            /*** Operando 2 ***/

/* INSERTAR CÓDIGO */

            /*** Reserva la entrada del ROB */

/* INSERTAR CÓDIGO */

    /*** Reserva del registro destino */

/* INSERTAR CÓDIGO */

    /*** VISUALIZACION ***/
            RS[s].estado = PENDIENTE;
            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 */

    /*** La instrucción se ha lanzado correctamente */

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

    return;

} /* end fase_ISS */


...
```

```
/*************************************************************
 *
 * Func: fase_FP_WB
 *
 * Desc: Implementa la fase 'WB' del algoritmo de Tomasulo
 *
 *************************************************************/

void fase_WB_alum() {
    /*************************************/
    /*  Variables locales                */
    /*************************************/

    marca_t i, s;

    ciclo_t orden;

    /*************************************/
    /*  Cuerpo función                   */
    /*************************************/

    /*** VISUALIZACIÓN ****/
    for (i = 0; i < TAM_ESTACIONES; i++) {
        if (RS[i].ocupado && RS[i].estado == FINALIZADA) {
            muestra_fase("-", RS[i].orden);
        } /* endif */
    } /* endif */
    /**********************/

    /*** Busca RS con resultados disponibles */

    orden = MAX_ORDEN;
    s = 0;

    for (i = 0; i < TAM_ESTACIONES; i++) {
        if (RS[i].ocupado && RS[i].estado == FINALIZADA && RS[i].orden < orden) {
            s = i;
            orden = RS[i].orden;
        } /* endif */
    } /* endif */

    if (orden >= MAX_ORDEN) return; /* No hay ninguna RS con resultados disponible

    /*** Volcado de resultados */

/* INSERTAR CÓDIGO */


    /*** Libera la RS */
```

```
/* INSERTAR CÓDIGO */


    /*** VISUALIZACIÓN ****/
    RS[s].estado = PENDIENTE;
    BUS.excepcion = RS[s].excepcion;
    if (BUS.excepcion == EXC_NONE) {
      muestra_fase("WB", RS[s].orden);
    } else {
      muestra_fase("<font color=\"red\">WB</font>", RS[s].orden);
    }
    /**********************/

    /*** Lectura de resultados */

    /* Reorder buffer */

/* INSERTAR CÓDIGO */


    if (BUS.excepcion != EXC_NONE) return; /* Si hay una excepción nadie utiliza e

    /* Estaciones de reserva */

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

/* INSERTAR CÓDIGO */

    } /* endfor */

    for (s = INICIO_RS_SUMREST;
            s <= FIN_RS_SUMREST; s++) {

/* INSERTAR CÓDIGO */

    } /* endfor */

    for (s = INICIO_RS_MULTDIV;
            s <= FIN_RS_MULTDIV; s++) {

/* INSERTAR CÓDIGO */

    } /* endfor */

    for (s = INICIO_BUFFER_CARGA;
            s <= FIN_BUFFER_CARGA; s++) {

/* INSERTAR CÓDIGO */
```

```
    } /* endfor */

    for (s = INICIO_BUFFER_ALMACEN;
            s <= FIN_BUFFER_ALMACEN; s++) {

/* INSERTAR CÓDIGO */

    } /* endfor */

} /* end fase_WB */
```