# PRÁCTICA 5ª: ALGORITMO DE TOMASULO:*Commit*

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 la fase *Commit* del algoritmo de gestión dinámica de instrucciones conocido como Algoritmo de Tomasulo.
- Implementar y evaluar mecanismos de predicción dinámica de saltos.

## 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 la etapa COMMIT del algoritmo de Tomasulo con especulación. El simulador posee un conjunto de instrucciones enteras reducido, e instrucciones de coma flotante aritméticas y de carga/almacenamiento de doble precisión.

## 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, tampones de lectura y escritura, *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, tampones de lectura y escritura, 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_alum.c** Contiene las funciones para la predicción de saltos. *Éste fichero se deberá modificar.*

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

**f_lanzamiento.c** Contiene la implementación de la fase de lanzamiento de instrucciones multiciclo (Issue) del algoritmo de Tomasulo con especulación.

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

**f_transferencia.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.

**f_confirmacion_alum.c** Contiene la implementación de la fase de confirmación (Commit) del algoritmo de Tomasulo con especulación. *Este fichero se deberá modificar.*

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

# Instrucciones implementadas

| 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 con especulación

A continuación se muestra el pseudo-código de la etapa COMMIT del algoritmo de Tomasulo con especulación:

- *Commit*.

```
Si {instrucción en la cabeza del ROB, ha terminado}
     RB[RB_inicio].instr

  Si (RB[RB_inicio].instr=Salto) y {Predicción incorrecta}
     Liberar registros:
```

```
        Rfp[i].rob := MARCA_NULA, Rint[i].rob := MARCA_NULA.
    Borrar estaciones de reserva, menos los TE confirmados:
        RS[i].ocupado := NO.
    Liberar los operadores (excepto el de memoria si está escribiendo)
    Escribir en NPC (Control_1.NPC) el valor correcto.
    Borrar reorder buffer:
        RB[i].ocupado := NO.
Sino Si (RB[RB_inicio].instr=Almacenamiento)
    Confirmar escritura:
        TE[RB[RB_inicio].dest].confirm := SI;
Sino
    Actualizar registros:
        Regs[RB[RB_inicio].dest].valor := RB[RB_inicio].valor;
    ¿Ninguna otra instrucción posterior escribe sobre este registro?
        Si (Regs[RB[RB_inicio].dest].rob = RB_inicio)
          Liberar registro:
              Regs[RB[RB_inicio].dest].rob := MARCA_NULA;
 Liberar entrada en reorder buffer:
 RB[RB_inicio].ocupado := NO;
```

# Implementación de la condición de salto

El simulador lleva incorporado un predictor del tipo *Branch Target Buffer* de 1 bit que ofrece la predicción y la dirección de destino al final de la etapa de búsqueda de instrucciones (IF). La predicción se almacena en el campo `prediccion` de la entrada correspondiente del *reorder buffer*, para la posterior comprobación en la etapa COMMIT.

En todas las instrucciones de salto condicional (`BEQZ`, `BNEZ`, `BC1T` y `BC1F`), la condición se almacena directamente en el campo `valor` del reorder buffer, como un entero (`valor.i`), como resultado de la ejecución de la propia instrucción de salto.

Un valor igual a `1` en el campo `valor` indica que se debe saltar, y un valor igual a `0` que el salto no es efectivo. La dirección de destino del salto se encuentra en el campo `dest` del reorder buffer.

# Ejercicios a realizar

1. Implementación del algoritmo de Tomasulo con especulación.

   Tras familiarizarse con las estructuras de datos y la estructura del simulador, implementar la etapa COMMIT del algoritmo de Tomasulo con especulación. Dicha etapa se implementará dentro de la función `fase_COM_alum()` (fichero `f_confirmacion_alum.c`). Existe una estructura previa en dicha función que se muestra en el apéndice C.

   Para la edición de los ficheros se puede utilizar cualquiera de los editores disponibles: `vi`, `[x]emacs` o `nedit` (editor al estilo WordPad).

   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`. No olvidar ejecutar "`export PATH=$PATH:.`" para que se busque la orden en el directorio actual.

2. Comprobar y evaluar el funcionamiento del algoritmo de Tomasulo con especulación.

   Una vez implementado y compilado el algoritmo de Tomasulo con especulación, se comprobará su funcionamiento utilizando para ello los siguientes ejemplos.

   *a*) Comprobar el funcionamiento del programa que se encuentra en el fichero `ejemplo.s`.

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

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

   Esta orden generará un fichero en formato **html** por cada ciclo con la información sobre el estado de la máquina. Para visualizarlo se puede utilizar un navegador.

   El fichero `ejemplo.sign` contiene el resumen de los estados del procesador correspondientes a la ejecución correcta del fichero `ejemplo.s`. En caso de existir alguna diferencia con dicho fichero, el simulador informará 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 "??".

   Comprobar su correcto funcionamiento, tanto lógico como temporal. Para ello, se deberán tener en cuenta los tiempos de evaluación 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).

   *b*) Comprobar el funcionamiento de un bucle de DAXPY (`daxpy.s`).

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

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

3. Analizar el tiempo de ejecución para diversas opciones de configuración.

   Utilizando la versión del bucle DAXPY que opera con vectores de 64 elementos (fichero `daxpy64.s`), obtener el tiempo de ejecución en ciclos, el CPI y el número de operaciones en coma flotante por ciclo para diversas opciones de configuración del procesador:

   - Parámetros por defecto:

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

   - Parámetros por defecto, sin predictor de saltos:

     ```
     mips-ooo -f daxpy64.s -s -b 0
     ```

   - Procesador superescalar de 2 vías:

     ```
     mips-ooo -f daxpy64.s -s -v 2:2:2
     ```

- Procesador superescalar de 2 vías, ROB=32 entradas:

```
mips-ooo -f daxpy64.s -s -v 2:2:2 -r 32
```

- Procesador superescalar de 4 vías, ROB=32 entradas:

```
mips-ooo -f daxpy64.s -s -v 4:4:4 -r 32
```

- Procesador superescalar de 4 vías, ROB=32 entradas, multiplicador segmentado:

```
mips-ooo -f daxpy64.s -s -v 4:4:4 -r 32 -m 1:7:s:2
```

- Procesador superescalar de 4 vías, ROB=32 entradas, multiplicador segmentado, dos operadores de carga/almacenamiento:

```
mips-ooo -f daxpy64.s -s -v 4:4:4 -r 32 -m 1:7:s:2 -l 2:2:c:3:3
```

- Procesador superescalar de 4 vías, ROB=64 entradas, multiplicador segmentado (4 ER), dos operadores de carga/almacenamiento (4 buffers):

```
mips-ooo -f daxpy64.s -s -v 4:4:4 -r 64 -m 1:7:s:4 -l 2:2:c:4:4
```

4. Comprobar y evaluar el funcionamiento del predictor *Branch Target Buffer*.

   Para las pruebas se utilizará la configuración original del simulador con respecto a la gestión dinámica de instrucciones: tamaño del *reorder buffer* (20 entradas), número de estaciones de reserva (5 de enteros, 3 de suma/resta flotante, 2 de mult/div flotante, 3 tampones de lectura y escritura), etc.

   *a*) Comprobar el funcionamiento del predictor BTB utilizando el algoritmo de ordenación que se encuentra en el fichero `ordena.s`. Dicho programa realiza la ordenación de un vector mediante el método de la burbuja.

   Para comprobar el correcto funcionamiento del predictor, ejecutar el simulador con el fichero de firmas `ordena1.sign`. Observar los instantes en el que cada instrucción de salto obtiene y actualiza la predicción.

```
mips-ooo -t ordena1.sign -f ordena.s
```

   Seguidamente, se evaluará el comportamiento del predictor BTB utilizando un tamaño de vector mayor. Para ello, ejecutaremos el código suministrado en el fichero `ordena_largo.s`. No se utilizarán ficheros de firmas. Se deberán anotar las estadísticas obtenidas para su posterior comparación.

```
mips-ooo -f ordena_largo.s
```

   *b*) Aumentar a 4 entradas el tamaño del *buffer* en el predictor BTB (fichero `main.h`) para evitar los problemas de reemplazamiento. Comprueba su correcto funcionamiento con el programa `ordena.s` y el fichero de firmas `ordena2.sign`.

```
mips-ooo -t ordena2.sign -f ordena.s -b 4
```

Evaluar el comportamiento del predictor BTB utilizando el fichero `ordena_largo.s`, sin utilizar ficheros de firmas. Se deberán anotar las estadísticas obtenidas y compararlas con las anteriormente obtenidas.

```
mips-ooo -f ordena_largo.s -b 4
```

*c*) Manteniendo el mismo tamaño de la tabla de BTB, modificar el predictor BTB para que utilice un estado para la predicción de 2 bits. Para ello se deberán modificar las funciones de consulta y actualización de la predicción que se encuentran en el fichero `prediccion_alum.c`.

Se deberá analizar la corrección de las modificaciones utilizando el programa `ordena.s` y el fichero de firmas `ordena3.sign`.

```
mips-ooo -t ordena3.sign -f ordena.s -b 4
```

Evaluar el comportamiento del predictor BTB utilizando el fichero `ordena_largo.s` y sin utilizar ficheros de firmas. Se deberán anotar las estadísticas obtenidas y compararlas con las anteriormente obtenidas.

```
mips-ooo -f ordena_largo.s -b 4
```

# A. Estructuras de datos

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

## A.1. Tipos básicos

Los tipos básicos utilizados son:

```
typedef unsigned char   byte;   /* Un byte: 8 bits */
typedef short           half;   /* Media palabra: 16 bits */
typedef int32_t         word;   /* Una palabra: 32 bits */
typedef int64_t         dword;  /* Una palabra: 64 bits */

typedef unsigned long   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
{
  NONE,
  EX,
  WB
```

```
} estado_t;                     /* Estado de una operación */

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

## A.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`. El campo `ocupado` (*bit de bloqueo*) se ha eliminado, por corresponder de forma biunívoca con la condición `rob != MARCA_NULA`.

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

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

## A.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       Qj;             /* Marca del primer operando. ALU */
  valor_t       Vj;             /* Valor del primer operando. ALU */

  marca_t       Qk;             /* Marca del segundo operando. ALU y TE */
  valor_t       Vk;             /* Valor del segundo operando. ALU y TE */
```

```
  word          direccion;      /* Dirección de acceso a memoria. TL y TE */
  boolean       confirm;        /* Indica si la operación de escritura
                                   ha sido confirmada (commit). TE */

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

  valor_t       resultado;      /* Resultado de la operación */

  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 tampones de lectura y de escritura, usarán el mismo tipo de estación de reserva (`estacion_t`), para facilitar la programación del simulador.

## A.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 */

  estado_t      estado;         /* Estado de la oepración */

  dword         dest;           /* Registro destino, TE 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;
```

## A.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;
```

## A.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;
```

# B. Estructura de la unidad de gestión dinámica de instrucciones

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

**Banco de Registros de Coma Flotante** Contiene los registros de coma flotante. Está representada por la variable `Rfp` (`main.h`), del tipo `reg_t []` (`tipos.h`). El número de registros viene indicado por la constante `TAM_REGISTROS` (`main.h`).

**Banco de Registros Enteros** Contiene los registros enteros. Está representada por la variable `Rint`, del tipo `reg_t []`. El número de registros viene indicado por la constante `TAM_REGISTROS` (`main.h`).

**Reorder Buffer** Almacena las instrucciones lanzadas hasta que llegan a la fase de confirmación. Esta representado por la variable `RB` (`main.h`), del tipo `reorder_t []` (`tipos.h`). El número de entradas viene indicado por la constante `TAM_REORDER` (`main.h`).

**Estaciones de Reserva del Sumador/Restador** Contiene las estaciones de reserva del operador de suma/resta. Está representada por la variable `RS` (`main.h`), del tipo `estacion_t []` (`tipos.h`), en el rango [`INICIO_RS_SUMA_RESTA`, `FIN_RS_SUMA_RESTA`]. El número de estaciones de reserva viene indicado por la constante `TAM_RS_SUMA_RESTA` (`main.h`).

**Operador Sumador/Restador** Se encarga de realizar las operaciones de suma y resta de números de coma flotante. Está representado por la variable `Op` y la entrada `OPER_SUMREST` (`Op[OPER_SUMREST]`) (`main.h`), del tipo `operador_t` (`tipos.h`).

El operador **no** está segmentado. El tiempo de evaluación viene indicado por la constante `TEVAL_SUMREST` (`main.h`).

**Estación de Reserva del Multiplicador/Divisor** Contiene las estaciones de reserva del operador de multiplicación/división. Está representada por la variable `RS`, en el rango [`INICIO_RS_MULT_DIV`, `FIN_RS_MULT_DIV`]. El número de estaciones de reserva viene indicado por la constante `TAM_RS_MULT_DIV` (`main.h`).

**Operador Multiplicador/Divisor** Se encarga de realizar las operaciones de multiplicación y división de números de coma flotante. Está representado por la variable `Op` y la entrada `OPER_MULTDIV` (`Op[OPER_MULTDIV]`).

El operador **no** está segmentado. El tiempo de evaluación viene indicado por la constante `TEVAL_MULTDIV`.

**Estación de Reserva de Operaciones Enteras** Contiene las estaciones de reserva del operador de enteros. Está representada por la variable `RS`, en el rango [`INICIO_RS_ENTEROS`, `FIN_RS_ENTEROS`]. El número de entradas disponibles viene indicado por la constante `TAM_RS_ENTEROS` (`main.h`).

**Operador de Enteros** Se encarga de realizar las operaciones enteras. Está representado por la variable `Op` y la entrada `OPER_ENTEROS` (`Op[OPER_ENTEROS]`).

El operador **no** está segmentado. El tiempo de evaluación viene indicado por la constante `TEVAL_ENTEROS`.

**Tampón de lectura** Contiene las estaciones de reserva del operador de carga/almacenamiento para la operación de lectura. Está representada por la variable `TL` (alias de `RS`), del tipo `estacion_t []`, en el rango [`INICIO_TAMPON_LECT`, `FIN_TAMPON_LECT`]. El número de tampones viene indicado por la constante `TAM_TAMPON_LECT` (`main.h`).

**Tampón de escritura** Contiene las estaciones de reserva del operador de carga/almacenamiento para la operación de escritura. Está representada por la variable `TE` (alias de `RS`), en el rango [`INICIO_TAMPON_ESCR`, `FIN_TAMPON_ESCR`]. El número de tampones viene indicado por la constante `TAM_TAMPON_ESCR` (`main.h`).

**Operador de Carga/Almacenamiento** Se encarga de realizar las operaciones de lectura y escritura de la memoria de datos. Está representado por la variable `Op` y la entrada `OPER_MEMDATOS` (`Op[OPER_MEMDATOS]`).

El operador **no** está segmentado. El tiempo de evaluación viene indicado por la constante `TEVAL_MEMORIA`.

Para saber qué operación está realizando basta con analizar el valor almacenado en el campo `estacion`. Si dicho valor está comprendido dentro del rango correspondiente a los tampones de escritura ([`INICIO_TAMPON_ESCR` .. `FIN_TAMPON_ESCR`] ), se trata de un almacenamiento. En caso contrario, se trata de una carga.

**Branch Target Buffer** Almacena la predicción de los saltos ejecutados. Esta representado por la variable `BTB` (`main.h`), del tipo `entrada_btb_t []` (`tipos.h`). El número de entradas viene indicado por la constante `TAM_BUFFER_PREDIC` (`main.h`).

**Bus común** Se encarga de las trasferencias entre los diversos componentes del sistema. Está representado por la variable `BUS` (`main.h`), del tipo `bus_comun_t` (`tipos.h`).

# C. Fuentes

```
/*************************************************************
 *
 * Func: fase_COM_alum
 * Desc: Implementa la fase 'COMMIT' del algoritmo de
 *       Tomasulo con especulación
 *
 *************************************************************/
void fase_COM_alum ( )
{
  /***********************************/
  /*  Variables locales              */
  /***********************************/
  int           i;
  /***********************************/
  /*  Cuerpo función                 */
  /***********************************/
  ..................................

  if (!terminando &&
      RB[RB_inicio].ocupado &&
      RB[RB_inicio].estado == WB)
    {
      /*** Atención de las excepciones */
      ..................................

      /*** Confirmación de la intrucción */

      switch (RB[RB_inicio].OP)
        {
        case OP_NOP:
        case OP_TRAP:
          /*** No hace nada ***/
          break;
          /*** SALTOS ***/
        case OP_BC1T:
        case OP_BC1F:
        case OP_BNEZ:
        case OP_BEQZ:
        case OP_BNE:
        case OP_BEQ:
            /* Actualizar la predicción (haya o no haya habido fallo) */
            actualizar_prediccion(RB[RB_inicio].PC, RB[RB_inicio].orden,
                                  RB[RB_inicio].valor.int_d, RB[RB_inicio].dest);

            if (RB[RB_inicio].valor.int_d == RB[RB_inicio].prediccion) {
                /*** Estadisticas ***/
                estat.saltos_acertados++;
            } else {
                /*** Predicción incorrecta ***/

                /* Liberar los registros */

                /*** INSERTAR CÓDIGO ***/
```

```
            /* Liberar las estaciones de reserva */

            /*** INSERTAR CÓDIGO ***/

            /* Liberar los operadores */

            /*** INSERTAR CÓDIGO ***/

            /* Preparar la búsqueda de la dirección correcta */

            /* Control_1.NPC= ??? */ /*** INSERTAR CÓDIGO ***/

            Control_1.Cancelar= SI;

            /* Liberar el reorder buffer e inicializarlo */

            /*** INSERTAR CÓDIGO ***/

            /* Inicializar la cola circular del reorder buffer */

            RB_long= 0;
            RB_inicio= 0;
            RB_fin= 0;

            return ;

        } /* endif */
        break;
        /*** ALMACENAMIENTOS EN MEMORIA ***/
    case OP_SD:
    case OP_FP_S_D:
        /*** Confirma la intrucción de escritura */

        /*** INSERTAR CÓDIGO ***/
        break;
        /*** OPERACIONES ENTERAS CON ESCRITURA EN REGISTROS */
    case OP_LD:
    case OP_DADD:
    case OP_DSUB:
    case OP_DADDI:
    case OP_DSUBI:
    case OP_FP_LT_D: /* Escriben en un registro entero que usamos como */
    case OP_FP_GT_D: /* registro de estado del coprocesador de C.F. */
        if (RB[RB_inicio].dest != 0) /* El registro R0 no se modifica */
            /*** INSERTAR CÓDIGO ***/

        /*** INSERTAR CÓDIGO ***/

        break;
        /*** OPERACIONES DE COMA FLOTANTE CON ESCRITURA EN REGISTROS */
    case OP_FP_L_D:
    case OP_FP_ADD_D:
    case OP_FP_SUB_D:
    case OP_FP_MUL_D:
    case OP_FP_DIV_D:
        /*** INSERTAR CÓDIGO ***/
```

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

      /* Actualizar el reorder buffer */

      RB[RB_inicio].ocupado= NO;
      RB_inicio= (RB_inicio + 1) % TAM_REORDER;
      RB_long--;
    } /* endif */
  return ;
} /* end fase_COM_alum */
```

```
/*************************************************************
 *
 * Func: obtener_prediccion
 *
 * Desc: Obtiene la predicción para una instrucción dada. Devuelve cierto si la
 * instruccion era un salto, modificando el valor del parometro 'prediccion'.  Si la
 * prediccion es 'salto tomado' (prediccion == SI), se modifica el valor del parometro
 * 'destino' para indicar la direccion de destino del salto.
 *
 *************************************************************/

boolean obtener_prediccion_alum
(
    word        PC,
    ciclo_t       orden,
    boolean *   prediccion,
    dword *      destino
 )
{
    /*************************************/
    /*  Variables locales                */
    /*************************************/

    int i;
    boolean encontrado;

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

    encontrado= NO;
    for (i=0; i<TAM_BUFFER_PREDIC; i++) {
        if (BTB[i].PC == PC) {
            encontrado= SI;
            estat.saltos_encontrados++;
            BTB[i].orden= orden;

            if (BTB[i].estado == SALTA) {
                *prediccion= SI;
                *destino= BTB[i].destino;
            }
            else {
                *prediccion= NO;
            } /* endif */

            break;
        } /* endif */
    } /* endfor */

    return (encontrado);

} /* end obtener_prediccion */

/*************************************************************
 *
 * Func: actualizar_prediccion
```

```
 *
 * Desc: Actualiza la predicción para una instrucción dada.
 *
 ************************************************************/

void actualizar_prediccion_alum
(
    word        PC,
    ciclo_t       orden,
    boolean     condicion,
    dword        destino
 )
{
    /************************************/
    /*  Variables locales               */
    /************************************/

    int i;
    boolean     encontrado;

    ciclo_t       antiguedad;
    int         mas_antigua;

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

    antiguedad= LONG_MAX;
    mas_antigua= -1;

    estat.saltos_ejecutados++;
    encontrado= NO;
    for (i=0; i<TAM_BUFFER_PREDIC; i++) {
        if (BTB[i].PC == PC) {
            encontrado= SI;
            BTB[i].destino= destino;

            /* Actualiza el estado */
            if (condicion) {
                BTB[i].estado= SALTA;
            }
            else {
                BTB[i].estado= NO_SALTA;
            } /* endif */

            break;
        } /* endif */

        /* Algoritmo de remplazamiento LRU */
        ................................
    } /* endfor */

    if (!encontrado) {
        ................................

        /* Actualiza el estado. Inicialmente se supone en NO_SALTA */
```

```
        if (condicion) {
            BTB[mas_antigua].estado= SALTA;
        }
        else {
            BTB[mas_antigua].estado= NO_SALTA;
        } /* endif */
    } /* endif */

} /* end actualizar_prediccion */
```