
PRÁCTICA 2b:

“UNIDAD DE INSTRUCCIÓN SEGMENTADA (II)”

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

Objetivos:

- Implementar la lógica para resolver los riesgos de datos y de control en un procesador segmentado.

Desarrollo:

Para el desarrollo de la práctica se partirá de un simulador del MIPS con instrucciones enteras. El simulador permite interpretar directamente código ensamblador del MIPS y simula una unidad de instrucción segmentada en 5 etapas (IF, ID, EX, MEM y WB). El código que resuelve los riesgos de datos y de control no está incluido.

El presente boletín se organiza como sigue: descripción de los ficheros que componen el simulador, las instrucciones implementadas, estructuras de datos utilizadas, estructura del simulador del MIPS segmentado y, finalmente, los ejercicios a realizar.

Estructura del simulador

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

main.c Programa principal del simulador. Encargado de leer el ensamblador y ejecutar las distintas fases de la unidad de instrucción segmentada.

main.h Contiene todas las variables compartidas del simulador: memoria de instrucciones y datos, registros de uso general, registros inter-etapa, señales de control, etc.

tipos.h Contiene las definiciones de todas las estructuras de datos utilizadas en el simulador: memoria de instrucciones y datos, banco de registros, registros inter-etapa, formatos de instrucción, 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-html.c, presentacion-txt.c Contiene las funciones para la presentación de los resultados.

mips.c Contiene la implementación de las fases IF, ID, EX, MEM y WB de la unidad de instrucción. *Este es el fichero que modificaremos en esta práctica.*

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

Instrucciones implementadas

El simulador ejecuta todas las instrucciones enteras del MIPS, tanto en versiones registro–registro como registro–inmediato, con la excepción de la instrucción de salto incondicional. En particular, es capaz de ejecutar el siguiente repertorio de instrucciones:

- Carga/almacenamiento

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

- Aritméticas, lógicas y de desplazamiento

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

- Comparación:

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

- Salto condicional

bnez Ry, desp	bc1t desp
beqz Ry, desp	bc1f desp

- Otras

nop
trap

Estructuras de datos

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

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 doble palabra: 64 bits */

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

Formatos de instrucción

Los formatos de instrucción se representan mediante un tipo enumerado:

```
/* Formatos de instrucción */
typedef enum {FormatoR, FormatoI, FormatoJ} formato_t;
```

Las instrucciones se representan mediante la siguiente estructura de datos:

```
typedef struct {
    codop_t      codop;          /* Código de operación */
    formato_t    tipo;           /* Formato */

    byte         Rfuente1,       /* Registro fuente 1 */
              Rfuente2;         /* Registro fuente 2 */
    byte         Rdestino;       /* Registro destino */
    half         inmediato;      /* Valor Inmediato */
} instruccion_t;
```

Banco de registros

El banco de registros es un vector compuesto por elementos del tipo `reg_int_t`, con un único campo, valor.

```
typedef struct {
    dword        valor;          /* Valor del registro */
} reg_int_t;
```

Registros inter-etapa

Los registros inter-etapa se representan mediante una estructura que contiene cada uno de los campos necesarios:

- Registro IF/ID:

```
typedef struct {
    instruccion_t IR;           /* IR */
    dword         NPC;          /* PC+4 */
} IF_ID_t;
```

- Registro ID/EX:

```
typedef struct {
    instruccion_t IR;           /* IR */
    dword NPC;                 /* PC+4 */
    dword Ra,                  /* Valores de los registros*/
    Rb;
    dword Imm;                 /* Inmediato con signo extendido */
} ID_EX_t;
```

- Registro EX/MEM:

```
typedef struct {
    instruccion_t IR;           /* IR */
    dword ALUout;               /* Resultado ALU */
    dword data;                 /* Dato a escribir */
    boolean cond;               /* Resultado condicion de salto */
} EX_MEM_t;
```

- Registro MEM/WB:

```
typedef struct {
    instruccion_t IR;           /* IR */
    dword ALUout;               /* Resultado ALU */
    dword MEMout;               /* Resultado Memoria */
} MEM_WB_t;
```

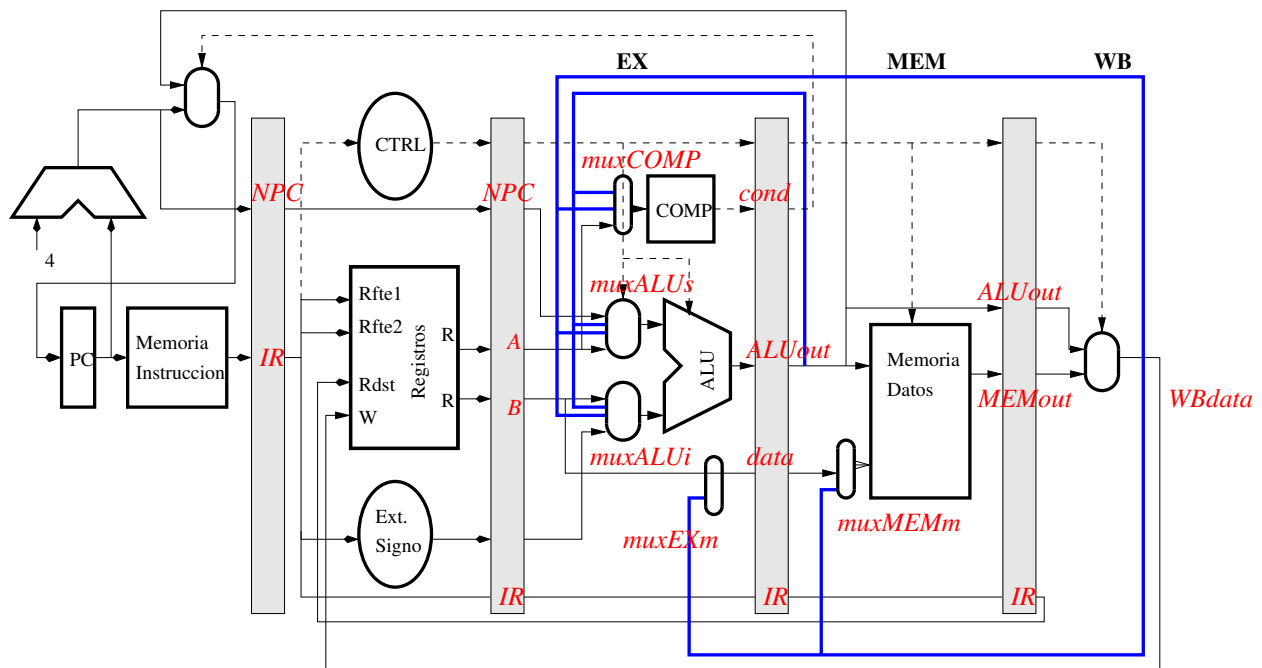
Otras estructuras de datos

- Define cómo se resuelven los riesgos de datos:

```
typedef enum {
    parada,                    /* Inserta ciclos de parada */
    cortocircuito,              /* Aplica cortocircuito+ciclos de parada */
    ninguno
} riesgos_d_t;
```

- Define cómo se resuelven los riesgos de control:

```
typedef enum {
    stall,                      /* Inserta 3 ciclos de parada */
    pnt,                        /* Predict-not-taken, 3 ciclos */
    ds3,                        /* Salto retardado, con ds=3 */
    ds2,                        /* Salto retardado, con ds=2 */
    ds1,                        /* Salto retardado, con ds=1 */
} riesgos_c_t;
```



Estructura de unidad de instrucción segmentada

La unidad de instrucción segmentada está compuesta por los siguientes elementos (ver figura):

Memoria de instrucciones. Almacena el programa a ejecutar. Es direccionable a la palabra¹. Está representada por la variable `MI (main.h)`, del tipo `memoria_instruccion_t (tipos.h)`. El tamaño de la memoria viene indicado por la constante `TAM_MEMO_INSTRUC (tipos.h)`.

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

Operador aritmético. Realiza las operaciones aritméticas correspondientes a la fase EX. Está representada por la función `operacion_ALU (codop, in1, in2) (mips.c)`, donde `codop`, `in1` e `in2` representan la función a la realizar y los dos datos a operar, respectivamente. La función devuelve el resultado de la operación correspondiente.

Evaluación de la condición de salto. Calcula la condición de salto. Está representada por la función `operacion_COMP (codop, in1) (mips.c)`, donde `codop` e `in1` representan la función a la realizar y el dato a evaluar, respectivamente. La función devuelve si se debe o no saltar.

Multiplexor de la entrada superior del operador aritmético. Está representado por la función `mux_ALUsup (npc, ra, mem, wb) (mips.c)`, donde `npc`, `ra`, `mem` y `wb` representan las entradas al mismo. La función devuelve como resultado uno de los parámetros de entrada, según corresponda.

¹palabras sucesivas tienen asignadas direcciones consecutivas

Multiplexor de la entrada inferior del operador aritmético. Está representado por la función `mux_ALUinf (rb, imm, mem, wb) (mips.c)`, donde `rb`, `imm`, `mem` y `wb` representan las entradas al mismo. La función devuelve como resultado uno de los parámetros de entrada, según corresponda.

Multiplexor de la entrada del comparador (saltos). Está representado por la función `mux_COMP (ra, mem, wb) (mips.c)`, donde `ra`, `mem` y `wb` representan las entradas al mismo. La función devuelve como resultado uno de los parámetros de entrada, según corresponda.

Multiplexor de los datos a escribir en memoria (etapa EX). Está representado por la función `mux_EXmem (rb, wb) (mips.c)`, donde `rb` y `wb` representan las entradas al mismo. La función devuelve como resultado uno de los parámetros de entrada, según corresponda.

Memoria de datos. Almacena los datos del programa a ejecutar. Es direccionable al byte². Está representada por la variable `MD (main.h)`, del tipo `memoria_datos_t (tipos.h)`. El tamaño de la memoria viene indicado por la constante `TAM_MEMO_DATOS (tipos.h)`.

Multiplexor de los datos a escribir en memoria (etapa MEM). Está representado por la función `mux_MEMmem (rb, wb) (mips.c)`, donde `rb` y `wb` representan las entradas al mismo. La función devuelve como resultado uno de los parámetros de entrada, según corresponda.

Salida del multiplexor de la fase WB. Representa el dato que se va a escribir en el banco de registros en la fase WB. Está representado por la variable `WBdata (main.h)`, de tipo `dword`.

Registros inter-etapa. Su nombre viene de las etapas que interconectan. Son los siguientes:

- `IF_ID`, del tipo `IF_ID_t (tipos.h)`.
- `ID_EX`, del tipo `ID_EX_t (tipos.h)`.
- `EX_MEM`, del tipo `EX_MEM_t (tipos.h)`.
- `MEM_WB`, del tipo `MEM_WB_t (tipos.h)`.

Valor actual y nuevo valor del PC. Lo representa las variables `PC` y `PCn`, del tipo `dword`. La siguiente instrucción se buscará en la dirección indicada en `PCn`.

Señales generadas a partir del tipo de instrucción Estan asociadas a cada una de las etapas, y detectan si se dan ciertas condiciones de la instrucción presente en la etapa correspondiente. Las señales están representadas por las siguientes funciones:

- `hay_fuente1_ID()`. Indica que la instrucción que está en la etapa ID hace uso del registro `fuente1`, por lo que el campo "Rfuente1" del registro de instrucción contiene un valor válido.

²palabras sucesivas tienen asignadas direcciones que difieren en 4

- `hay_fuente2_ID()`. Indica que la instrucción que está en la etapa ID hace uso del registro `fuentes2`.
- `hay_fuente1_EX()`. Indica que la instrucción que está en la etapa EX hace uso del registro `fuentes1`.
- `hay_fuente2_EX()`. Indica que la instrucción que está en la etapa EX hace uso del registro `fuentes2`.
- `hay_destino_EX()`. Indica que la instrucción que está en la etapa EX hace uso del registro destino.
- `hay_destino_MEM()`. Indica que la instrucción que está en la etapa MEM hace uso del registro destino.
- `hay_destino_WB()`. Indica que la instrucción que está en la etapa WB hace uso del registro destino.

Señales de control El simulador dispone de las siguientes señales de control, todas ellas del tipo `boolean`:

- `IFstall`: Al activarla (`IFstall=SI`) mantiene la instrucción en la fase IF al siguiente ciclo de reloj, entregando además una instrucción `nop` a la etapa ID.
- `IDstall`: Al activarla mantiene la instrucción en la fase ID al siguiente ciclo de reloj, entregando además una instrucción `nop` a la etapa EX.
- `IFnop`: Al activarla pasará una instrucción `nop` a la fase ID al siguiente ciclo de reloj.
- `IDnop`: Al activarla pasará una instrucción `nop` a la fase EX al siguiente ciclo de reloj.
- `EXnop`: Al activarla pasará una instrucción `nop` a la fase MEM al siguiente ciclo de reloj.

Pseudo-código del simulador del MIPS

El programa principal del simulador, tras inicializar las estructuras de datos, carga el fichero que contiene el programa a ejecutar, lo ensambla y ejecuta el bucle principal del simulador, cuyo pseudo-código se muestra seguidamente:

```
/* Bucle principal del simulador MIPS */

/** Fase: WB *****/
fase_escritura():
    -escribir registro

/** Fase: MEM *****/
fase_memoria():
    -detectar riesgos de control
    -aplicar cortocircuitos
    -acceso a memoria, en su caso
```

```

/** Fase: EX *****/
fase_ejecucion():
    -detectar riesgos de control
    -aplicar cortocircuitos
    -operacion en la ALU/COMP

/** Fase: ID *****/
fase_decodificacion():
    -detectar riesgos de datos
    -detectar riesgos de control
    -leer registros

/** Fase: IF *****/
fase_busqueda();
    -buscar instrucción
    -actualizar PC

Ciclo++;
imprimir_estado;
impulso_reloj();

```

Ejercicios a realizar

En primer lugar, probaremos el simulador con un sencillo fragmento de código que no tenga dependencias de datos, el cual está almacenado en el fichero `ejemplo.s`

```

; Inicializar valores en registros
.ireg 15,60 ; r1=15, r2=60

.data
a:    .dword 100
b:    .dword 0

.text
inicio: bnez r5,final
        dadd r3,r1,r2
        dsub r4,r1,r2
        and r5,r1,r2
        or r6,r1,r2
        xor r7,r1,r2
        ld r1,a(r0)
        sd r3,b(r0)
        sgt r2,r3,r4
        beqz r0,inicio ; salto incondicional
final:

```

Para invocar la ejecución del simulador se utilizará la orden `mips`. El simulador acepta varios parámetros:

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


donde:

- <resultados> indica cómo se ofrecerá el resultado de la simulación. Hay varias opciones:
 - **tiempo**: Muestra el tiempo de ejecución en el terminal.
 - **final**: Muestra el tiempo de ejecución, los registros y el contenido de la memoria tras la ejecución en el terminal.
 - **ciclo**: Muestra la ejecución ciclo a ciclo en el terminal.
 - **html(*)**: Genera varios archivos html con el estado de la ejecución ciclo a ciclo así como los resultados finales. Los resultados se visualizan abriendo en un navegador el archivo **index.html**. Esta es la opción por defecto.
 - **html-final**: Genera un archivo html **final.html** con el resultado final de la ejecución.
- <riesgos de datos> indica cómo se resuelven los riesgos de datos. Hay tres opciones:
 - **n**: No hay lógica para resolver los riesgos de datos.
 - **p(*)**: Se resuelven los riesgos de datos insertando ciclos de parada. Esta es la opción por defecto.
 - **c**: Se resuelven los riesgos de datos mediante la técnica de la anticipación o cortocircuito, insertando asimismo los ciclos de parada necesarios.
- <riesgos de control> indica cómo se resuelven los riesgos de control. Hay cinco opciones:
 - **p(*)**: Se resuelven los riesgos de control insertando ciclos de parada. Esta es la opción por defecto.
 - **t**: Se resuelven los riesgos de control mediante *predict-not-taken*.
 - **3**: Se resuelven los riesgos de control mediante la técnica del salto retardado, con valor del *delay-slot*=3.
 - **2**: Se resuelven los riesgos de control mediante la técnica del salto retardado, con valor del *delay-slot*=2.
 - **1**: Se resuelven los riesgos de control mediante la técnica del salto retardado, con valor del *delay-slot*=1.

- <archivo.s> es el nombre del archivo que contiene el código en ensamblador.

En particular, escribiremos:

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

Esta orden generará un fichero en formato **html** por cada ciclo con la información sobre el estado de la máquina, más un fichero inicial `index.html`. Estos ficheros pueden visualizarse mediante un navegador. Por defecto, el simulador borra los archivos `html` al iniciar una nueva simulación, salvo que se le pase el parámetro “-n”.

Comprobar el correcto funcionamiento del simulador, y que el resultado de la ejecución es el esperado. Los resultados obtenidos deben ser los indicados seguidamente:

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

Sin embargo, el simulador suministrado sólo incluye el código necesario para resolver los riesgos de control insertando ciclos de parada o mediante la técnica del salto retardado con `delay slot=3`. Sin embargo, **no es capaz de detectar ni de resolver los riesgos de datos**.

La labor a realizar en esta práctica consiste en añadir al simulador nuevas estrategias para detectar y resolver los riesgos. En todos los casos, debe modificarse el fichero `mips.c`, añadiendo del código necesario para realizar las acciones indicadas.

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

Para la compilación del simulador `mips` se debe ejecutar el comando `make` en el directorio donde se encuentren los fuentes del simulador y el fichero `Makefile`.

Para comprobar el correcto funcionamiento de las modificaciones realizadas, se utilizarán los ficheros de prueba suministrados.

1. Modificar el simulador del MIPS para detectar y resolver los riesgos de datos **insertando ciclos de parada**. En particular, pretendemos resolver el riesgo de datos provocado por la siguiente secuencia de instrucciones (almacenado en el fichero `datos1.s`):

```
.ireg 10,20,0,0,0 ; r1=10, r2=20
.text
dadd r3,r1,r2
dsub r4,r3,#5
dadd r5,r3,#5
; el resultado debe ser r3=30, r4=25 y r5=35
```

Para ello, se debe modificar la función que realiza la fase de decodificación de las instrucciones (función `fase_decodificacion`) (`mips.c`), escribiendo el código que activa las señales de control necesarias (`IFstall`, `IDstall`).

Comprobar el correcto funcionamiento del simulador modificado ejecutando:

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

2. Modificar el simulador MIPS para detectar y resolver los riesgos de datos **aplicando cortocircuitos**.
 - En primer lugar, pretendemos resolver el riesgo de datos provocado por la misma secuencia de código del apartado anterior (fichero `datos1.s`), la cual no requiere insertar ciclos de parada.

Para ello, se debe modificar las funciones que implementan los multiplexores superior (operando fuente1) e inferior (operando fuente2) ubicados a la entrada del operador aritmético lógico. Como en el código de prueba el riesgo de datos únicamente afecta únicamente al operando fuente1 de las instrucciones, solo es necesario modificar la función `mux_ALUsup (mips.c)`.

Comprobar el correcto funcionamiento del simulador modificado ejecutando:

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

- Ahora pretendemos resolver el riesgo de datos provocado por una **instrucción de carga seguida por una instrucción aritmética**. El código de prueba está almacenado en el fichero `datos2.s`:

```
.ireg 10,20,0,0,0
.data
a:    .dword 10
      .text
      ld r3,a(r0)
      dsub r4,r3,#5
      dadd r5,r3,#5
      ; el resultado debe ser r3=10, r4=5 y r5=15
```

En este caso, además de activar el cortocircuito correspondiente, se debe insertar un ciclo de parada en la fase de decodificación. Por lo tanto, se debe modificar tanto la función que implementa el multiplexores superior asociado a la entrada del operador aritmético lógico (`mux_ALUsup, mips.c`), como la función que realiza la fase de decodificación de las instrucciones (función `fase_decodificacion, mips.c`).

Comprobar el correcto funcionamiento del simulador modificado ejecutando:

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

3. Modificar el simulador MIPS para resolver los riesgos de control mediante la estrategia *predict-not-taken*.

Para ello, se debe modificar la función que realiza la fase de búsqueda de las instrucciones (función `fase_búsqueda, mips.c`).

Para probar esta modificación, puede emplearse el código siguiente, almacenado en el fichero `suma.s`:

```
; suma las componentes del vector hasta que encuentra
; una componente a 0
; almacena el resultado en a
.data
a:    .dword 0
y:    .dword 1,2,3,0,4,5,6,7,8
      .text
      dadd r1,r0,r0 ; r1=0
      dadd r3,r3,r0 ; r3=0
      dadd r2,r0,y   ; r2 recorre y
```

```
bucle:  dadd r1,r3,r1
        ld r3,0(r2)    ; r3 es y[i]
        dadd r2,r2,#8
        bnez r3,bucle ; si r3<>0
        sd r1,a(r0)
final:
        ; el resultado debe a=6
```

Comprobar el correcto funcionamiento del simulador modificado, resolviendo los riesgos de datos con ciclos de parada:

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

Observa que ahora se cancelan las instrucciones sólo cuando el salto es efectivo.