PRÁCTICA 3:
"UNIDAD DE INSTRUCCIÓN SEGMENTADA (II)"

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

Objetivos:

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

Desarrollo:

Para el desarrollo de la práctica se partirá de un simulador del RISCV. El simulador interpreta código ensamblador del RISC V, incluyendo tanto instrucciones enteras como de coma flotante. En esta práctica, trabajaremos con programas que utilizan únicamente instrucciones enteras. Este tipo de instrucciones se simulan en una unidad de instrucción segmentada en 5 etapas (IF, ID, EX, MEM y WB). Sin embargo, el código que resuelve los riesgos de datos y de control no está incluido en la versión proporcionada del simulador, y su realización constituye el objetivo de la práctica.

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 RISC V segmentado y, finalmente, los ejercicios a realizar.

Estructura del simulador

El simulador del RISC V está escrito en el lenguaje de programación C. Se compone de diversos módulos, entre los que resaltaremos los siguientes:

main.c Programa principal del simulador. Encargado de leer el ensamblador y ejecutar las distintas etapas 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.

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

riscv.c Contiene la implementación de las etapas de la unidad de instrucción. *Modificaremos este fichero en la práctica.*

riscv_int.c Contiene la implementación de la unidad aritmética de enteros, la lógica de detección de riesgos de datos y la lógica para aplicar cortocircuitos entre instrucciones enteras. *Modificaremos este fichero en la práctica.*

Instrucciones implementadas

El simulador soporta el conjunto rv64i del RISC V.

Estructuras de datos

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

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

Formatos de instrucción

Los formatos de instrucción se representan mediante un tipo enumerado:

```
/* Formatos de instruccion */
typedef enum {FormatoR, FormatoI, FormatoS,
              FormatoB, FormatoU, 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          rs1,    /* Registro fuente 1 */
                rs2,    /* Registro fuente 2 */
                rs3;    /* Registro fuente 3 */
  byte          rd;     /* Registro destino */
  half          imm;    /* 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 {
  valor_t    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         PC;        /* PC */
} IF_ID_t;
```

- Registro ID/EX:

```
typedef struct {
  instruccion_t IR;         /* IR */
  dword         PC;         /* PC */
  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:

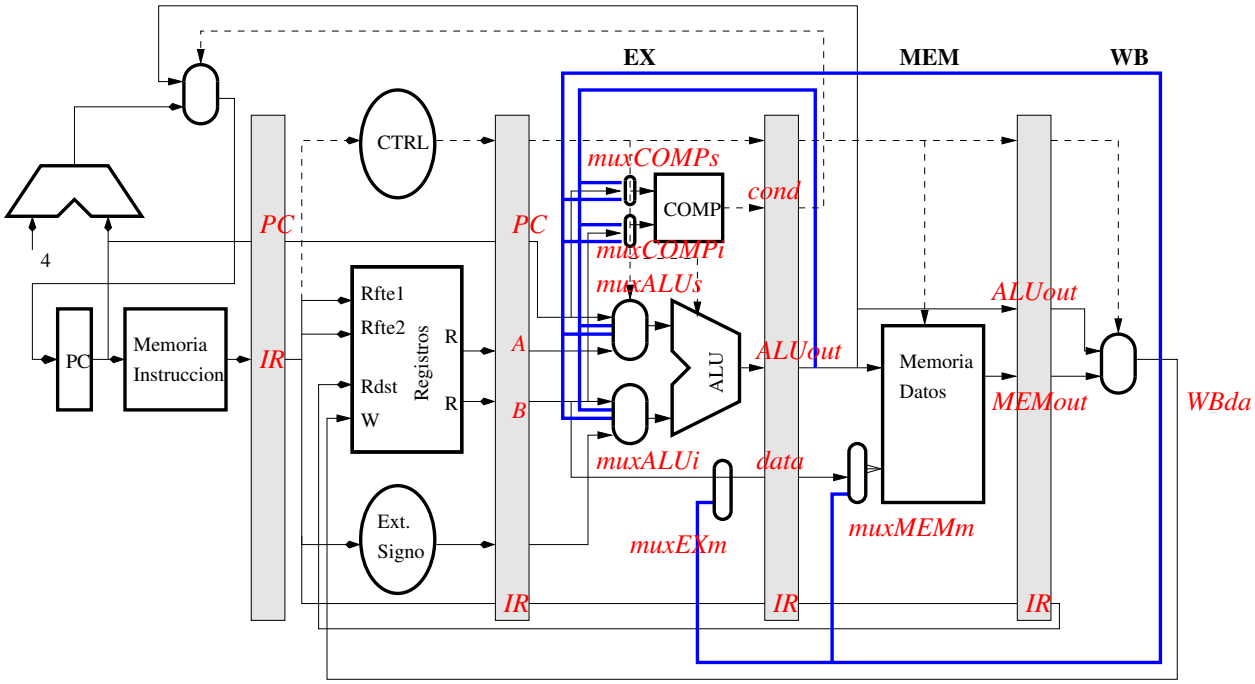
```
typedef struct {
  instruccion_t IR;          /* IR */
  dword         ALUout;      /* Resultado ALU */
  dword         MEMout;      /* Resultado Memoria */
} MEM_WB_t;
```

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 {
  stall3,    /* Inserta 3 ciclos de parada */
  stall2,    /* Inserta 3 ciclos de parada */
  stall1,    /* Inserta 3 ciclos de parada */
  pnt3,      /* Predict-not-taken, 3 ciclos */
  pnt2,      /* Predict-not-taken, 2 ciclos */
  pnt1       /* Predict-not-taken, 1 ciclos */
} 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 al byte[1]. Está representada por la variable `MI` (`main.h`), del tipo `memoria_instruciones_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 etapa EX. Está representada por la función `operacion_ALU (codop,in1,in2)` (`riscv_int.c`), donde `codop`, `in1` e `in2` representan la instrucción aritmética a ejecutar y los dos

[1]Bytes sucesivos tienen asignadas direcciones consecutivas y palabras sucesivas tienen asignadas direcciones que difieren en 4.

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,in2)` (`riscv_int.c`), donde `codop`, `in1` e `in2` representan la instrucción de salto a ejecutar y los operandos 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 (pc,ra,mem,wb)` (`riscv_int.c`), donde `pc`, `ra`, `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 inferior del operador aritmético. Está representado por la función `mux_ALUinf (rb,imm,mem,wb)` (`riscv_int.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 superior del comparador (saltos). Está representado por la función `mux_COMPsup (ra,mem,wb)` (`riscv_int.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 la entrada inferior del comparador (saltos). Está representado por la función `mux_COMPinf (rb,mem,wb)` (`riscv_int.c`), donde `rb`, `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)` (`riscv_int.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 byte. 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)` (`riscv_int.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 etapa WB. Representa el dato que se va a escribir en el banco de registros en la etapa 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`).

Por ejemplo, si queremos conocer el identificador del registro fuente1 de la instrucción que está en la etapa ID, utilizaremos `IF_ID.IR.rs1`.

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

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

Comprobación del tipo de instrucción. Las siguientes macros o funciones permiten analizar determinados campos de una instrucción:

- `es_load(inst)`. Indica que la instrucción `inst` es de carga.
- `rs1_valido(inst)`. Indica que la instrucción `inst` utiliza un registro fuente 1 válido.
- `rs2_valido(inst)`. Indica que la instrucción `inst` utiliza un registro fuente 2 válido.
- `rd_valido(inst)`. Indica que la instrucción `inst` utiliza un registro destino válido.

A modo de ejemplo, se muestra el contenido de una de ellas:

```
boolean rs1_valido(instruccion_t inst) {
    // Versión simplificada.
    // Para instr. aritméticas, lógicas, desp., comparación, saltos
    // Hay Rfte1 si no es x0
    return (
            (inst.rs1 != 0)
           );
```

Por ejemplo:

- si queremos saber si la instrucción que está en la etapa ID utiliza (va a leer) el campo registro fuente1, utilizaremos la función `rs1_valido(IF_ID.IR)`.
- si queremos saber si la instrucción que está en la etapa EX utiliza (va a escribir) el campo registro destino, utilizaremos la macro `rd_valido(ID_EX.IR)`.
- si queremos saber si la instrucción que está en la etapa ID es de carga, utilizaremos la llamada `es_load(IF_ID.IR)`.

Ejemplo de lógica de control. En la siguiente secuencia de instrucciones, hay que introducir dos ciclos de parada para resolver el riesgo de datos generado:

```
add x1,x2,x3  IF ID EX ME WB
sub x4,x1,x5     IF id id ID EX ME WB
```

Para el primer ciclo de parada, la lógica de control debe comprobar:

- Si el registro destino de la instrucción que está en la etapa EX (registro de segmentación ID_EX) coincide con el registro fuente1 de la instrucción que está en la etapa ID (registro de segmentación IF_ID).
- Que la instrucción que está en la etapa EX produce un resultado (en su registro destino).
- Que la instrucción que está en la etapa ID consume un resultado (en su registro fuente1).

y formalmente, utilizando las estructuras del procesador, macros y señales de control:

```
if ((ID_EX.IR.rd == IF_ID.IR.rs1) &&
     rd_valido(ID_EX.IR) && rs1_valido(IF_ID.IR)))
{
    IDstall = SI;
    IFstall = SI;
}
```

Pseudo-código del simulador del RISC V

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 RISC V */

  /*** Etapa: WB **************/
  fase_escritura():
    -escribir registro
```

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

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

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

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

      Ciclo++;
      imprimir_estado;
      impulso_reloj();
```

Ejercicios a realizar

En primer lugar, probaremos el simulador con el siguiente fragmento sencillo de código, que no tiene riesgos de datos, y que está almacenado en el fichero `ejemplo.s`:

```
        # adds the components of vector y until it finds
        # a component equal to 0
        # stores the result in a

        # the result must be a=6

        .data
a:      .dword  0
y:      .dword  1,2,3,0,4,5,6,7,8
        .text
        add t1,x0,x0  # t1=0
        add t3,x0,x0  # t3=0
        addi t2,gp,y  # t2 traverses y
        nop
loop:   add t1,t3,t1
        ld t3,0(t2)   # t3 is y[i]
        addi t2,t2,8
```

```
        nop
        bnez t3,loop  # if t3<>0
        sd t1,a(gp)
end:    ori a7,x0,10
        ecall
```

Para invocar la ejecución del simulador se utilizará la orden `riscv-m`. El simulador acepta varios parámetros. Puede obtenerse el detalle de todos los parámetros aceptados consultando el boletín de la Práctica 2 de la asignatura o también ejecutando la orden `riscv-m -?`.

En este caso, lo ejecutaremos sin lógica de detección de riesgos de datos y resolviendo los riesgos de control insertando (3) ciclos de parada:

```
riscv-m -d n -c s3 -f ejemplo.s
```

Esta orden generará un fichero en formato **html** por cada ciclo con la información sobre el estado de la máquina, más los ficheros inicial `index.html` y de resultados `final.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".

⇒ Comprueba el correcto funcionamiento del simulador, y que el resultado de la ejecución es el esperado.

El simulador suministrado solo incluye el código necesario para resolver los **riesgos de control insertando 3 ciclos de parada**. 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. Para ello, deberán modificarse los ficheros `riscv.c` y `riscv_int.c`, añadiendo el 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`.

Tras cada modificación de los fuentes, hay que compilar el simulador `riscv-m` utilizando la orden `make` en el directorio donde se encuentren los fuentes del simulador:

```
make
```

Para comprobar el correcto funcionamiento de las modificaciones realizadas, se utilizaran los ficheros de prueba suministrados, tal y como se explica a continuación.

1. Modifica el simulador del RISC V 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 (almacenada en el fichero `datos1.s`):

   ```
   # the result must be t3=30, t4=25 y t5=35
   .ireg 0,0,0,0,0,10,20 # t1=10, t2=20
   .text
   add t3,t1,t2
   ```

   hecho ✓

```
        addi t4,t3,-5
        addi t5,t3,5
end:    ori a7,x0,10
        ecall
```

⇒ Dibuja el diagrama instrucciones–tiempo correspondiente a la ejecución de la secuencia de código, insertando los ciclos de parada que sean necesarios.

⇒ Tomando como ejemplo la lógica de control mostrada en la página 7, se debe modificar la función que realiza la detección de riesgos de datos en la etapa de decodificación de las instrucciones (función `detectar_riesgos_datos` en el archivo `riscv_int.c`), escribiendo el código que activa las señales de control necesarias (`IFstall`, `IDstall`).

⇒ Tras compilar con éxito el simulador, comprueba su correcto funcionamiento ejecutando:

```
riscv-m -d p -c s3 -f datos1.s
```

y verifica que los registros involucrados tienen el valor correcto.

2. Modifica el simulador RISC V para detectar y resolver los riesgos de datos **aplicando cortocircuitos**.

   a) 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 si se emplean cortocircuitos.

   ⇒ Dibuja el diagrama instrucciones–tiempo correspondiente a la ejecución de la secuencia de código, aplicando los cortocircuitos que sean necesarios.

   ⇒ Ahora modifica las funciones que implementan los multiplexores superior (operando fuente1) e inferior (operando fuente2) ubicados a la entrada del operador aritmético-lógico. Estas funciones son `mux_ALUsup` y `mux_ALUinf`, respectivamente, y están ubicadas en el fichero `riscv_int.c`.

   ⇒ Tras compilar con éxito el simulador, comprueba el correcto funcionamiento del simulador modificado ejecutando:

   ```
   riscv-m -d c -c s3 -f datos1.s
   ```

   y verifica que los registros involucrados tienen el valor correcto.

   b) 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`:

   ```
            # the result must be t3=10, t4=5 y t5=15

            .data
   a:       .dword 10
            .text
            ld t3,a(gp)
   ```

```
        addi t4,t3,-5
        addi t5,t3,5
end:    ori a7,x0,10
        ecall
```

En este caso, además de activar el cortocircuito correspondiente, se debe insertar un ciclo de parada en la etapa de decodificación.

⇒ Dibuja el diagrama instrucciones–tiempo correspondiente a la ejecución de la secuencia de código, insertando los ciclos de parada y aplicando los cortocircuitos que sean necesarios.

⇒ Como se observa, además de la modificación realizada en la función que implementa el multiplexor superior asociado a la entrada del operador aritmético-lógico (`mux_ALUsup` en `riscv_int.c`) (ya realizado en el apartado 2*b*), debe modificarse la función que realiza la detección de riesgos en la etapa de decodificación de las instrucciones (función `detectar_riesgos_datos` en el fichero `riscv_int.c`).

⇒ Tras compilar con éxito el simulador, comprueba el correcto funcionamiento del simulador modificado ejecutando:

```
riscv-m -d c -c s3 -f datos2.s
```

y verifica que los registros involucrados tienen el valor correcto.

3. Modifica el simulador RISC V para resolver los riesgos de control mediante la estrategia *predict-not-taken*.

   ⇒ Para ello, se debe modificar la función que realiza la etapa de búsqueda de las instrucciones (función `fase_busqueda` en el fichero `riscv.c`). Puedes inspirarte en el código que implementa la estrategia de inserción de ciclos de parada (stall3).

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

```
        # adds the components of vector y until it finds
        # a component equal to 0
        # stores the result in a

        # the result must be a=6

        .data
a:      .dword  0
y:      .dword  1,2,3,0,4,5,6,7,8
        .text
        add t1,x0,x0  # t1=0
        add t3,x0,x0  # t3=0
        addi t2,gp,y  # t2 traverses y
loop:   add t1,t3,t1
        ld t3,0(t2)   # t3 is y[i]
        addi t2,t2,8
        bnez t3,loop  # if t3<>0
```

```
        sd t1,a(gp)
end:    ori a7,x0,10
        ecall
```

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

```
riscv-m -d p -c pnt3 -f suma.s
```

Observa que ahora se cancelan las instrucciones solo cuando el salto es efectivo. Verifica que la posición de memoria `a` tiene almacenado el valor correcto.