

## Proyecto Optativo 1: Fecha de entrega 26 de Abril

# MIPS segmentado y gestión de riesgos

### Resumen

El objetivo de este proyecto es aprender a trabajar con un lenguaje de descripción de hardware y un entorno de simulación basado en bancos de prueba y cronogramas, tomando como caso de estudio el mismo procesador MIPS de 32 bits utilizado en clase.

Como punto de partida, se facilitan los fuentes en VHDL correspondientes a un MIPS 32 **con anticipación de operandos**, que **resuelve los saltos en la etapa ID**, y **escribe en el banco de registros en los flancos de bajada**. Además **incluye un predictor de saltos** muy sencillo. Algunos de los módulos están vacíos, y el proyecto consiste en completarlos para que todo funcione correctamente,

En la primera parte del proyecto modificaremos este MIPS para que soporte dos nuevas instrucciones. Para ello debéis modificar tanto la Unidad de Control como la ruta de datos.

A continuación solucionaremos los riesgos de datos. En este caso hay que completar la lógica que se encarga de la anticipación de operandos y de detectar los riesgos que quedan y detener la ejecución de las etapas IF e ID hasta que se solucionen.

Finalmente hay que resolver los riesgos de control. Para hacerlo de forma más eficiente utilizamos un sencillo predictor de salto que almacena la información relativa al último salto realizado (dirección de salto, dirección de la instrucción y decisión tomada). Cuando se realice una predicción habrá que comprobar si se acertado o no, y en caso de fallo rectificar la ejecución. Es una simplificación de un tipo de predictor muy sencillo, que se conoce como BTB (Branch Target Buffer), y que normalmente en procesadores comerciales de propósito general utilizan como etapa previa a predictores más complejos y precisos.

### Paso 1: Trabajo inicial

Estudiar el código del procesador y comprender cómo funciona. Identificar los componentes del código con los componentes del procesador en las transparencias de clase. Ejecutar el código de ejemplo y ver cómo se ejecuta ciclo a ciclo. Comprobar que los registros y la memoria tienen el valor deseado. Identificar los errores que se producen debido a la funcionalidad sin completar (indicada en el propio código).

### Paso 2: Añadir nuevas instrucciones

Nos piden que añadamos dos sencillas instrucciones:

**Load address** (la): carga una dirección en un registro. Se parece mucho a un lw, pero no accede a memoria. Sólo calcula la dirección y la guarda en un registro. Normalmente es una *pseudoinstrucción*, típica en MIPS, SPARC, HLAASM... que se implementa con un load y un or. En nuestro caso la implementaremos como instrucción real, sujeta a esta descripción:

la rt, inmed(rs):

rt ← rs + SignExt( inmed ), PC ← PC + 4

**BNE**: salta si los registros no son iguales. Similar al BEQ, pero con la condición contraria.

### Paso 3: Gestión de riesgos de datos

En primer lugar debéis diseñar el controlador de la red de anticipación que gestiona los dos muxes que encontraréis ya colocados a la entrada de la ALU. Después debéis diseñar la lógica que detecte si la instrucción que está en la etapa ID tiene sus operandos disponibles en el banco de registros. En caso contrario debéis parar tanto ID como IF hasta que el riesgo desaparezca, y enviar a la etapa de ejecución las señales que correspondan a una instrucción con el código de operación de una NOP. Se deben detectar los riesgos tanto en las instrucciones originales como de las dos que hemos añadido (**la, bne**).

Hay que tener en cuenta que hay instrucciones que no usan la red de anticipación porque realizan todas sus operaciones en ID.

**Importante:** No se deben de realizar paradas innecesarias. Por ejemplo, una NOP no tiene que parar para esperar a sus operandos por una falsa dependencia con otra instrucción que escriba en r0, y un SW no debe provocar una parada por una dependencia load-uso que no existe.

### Paso 4: Gestión de riesgos de control

Debéis diseñar la gestión del módulo de predicción dinámica de saltos. El módulo es básicamente un registro que almacenará la información del último salto realizado, y un comparador que nos avisa si la instrucción que se va a ejecutar tiene la misma etiqueta que la instrucción almacenada en el predictor. El registro almacena la siguiente información:

- **Etiqueta (8 bits):** Nos permite identificar la instrucción de salto almacenada. Para estar seguros de que es la misma deberíamos almacenar su dirección completa, es decir el valor del PC, pero para ahorrar espacio sólo se usan los 8 bits menos significativos (del 9 al 2, dado que los dos últimos son siempre "00"). Esto puede generar falsos positivos que se gestionarán como cualquier otro fallo del predictor. Además, en lugar de usar PC como etiqueta, usaremos PC+4. La razón es que PC+4 ya se almacena en el MIPS base en la etapa de ID (en la señal PC4\_ID) y podemos usar ese mismo valor para actualizar la etiqueta cuando corresponda.
- **Dirección de salto (32 bits):** Almacenamos la dirección de salto completa.
- **Predicción (1 bit):** Indica qué ocurrió la última vez que se ejecutó la instrucción de salto ('0' no se saltó, '1' sí se saltó).
- **Validez (1 bit):** Indica si el registro contiene un dato válido. Al resetear el procesador válido se pone a '0'. Cuando se escriba la información de un salto se pondrá a 1 y se mantendrá así hasta el próximo reset.

El interfaz del predictor incluye:

- Un puerto de lectura que devuelve la dirección de salto y la predicción correspondiente a la etiqueta de la instrucción actual (PC4 (9 downto 2) )
- Un puerto de escritura que permite actualizar la información almacenada. Para ello se asigna un '1' a la señal update y se envía la nueva etiqueta, la dirección de salto y la predicción.

```

entity branch_predictor is Port (
    clk : in  STD_LOGIC;
    reset : in  STD_LOGIC;
-- Puerto de lectura
    PC4 : in  STD_LOGIC_VECTOR (7 downto 0);
    branch_address_out : out  STD_LOGIC_VECTOR (31 downto 0); --
    prediction_out : out  STD_LOGIC; -- indica si hay que saltar
-- Puerto de escritura
    PC4_ID: in  STD_LOGIC_VECTOR (7 downto 0);
    branch_address_in : in  STD_LOGIC_VECTOR (31 downto 0);
    prediction_in : in  STD_LOGIC;
    update: in  STD_LOGIC);
end branch_predictor;

```

Con este módulo el procesador gestionará los saltos de la siguiente forma:

- En IF el predictor recibe la etiqueta de la instrucción actual y la compara con la etiqueta que almacena internamente. En caso de que sean iguales, que el predictor tenga un dato válido, y que la predicción sea '1' se dará la orden de realizar el salto. En cualquier otro caso no se salta.
- La información relativa a la predicción se enviará a la etapa ID para poder comprobar si la predicción fue correcta o no.
- En ID se comprueba si se ha tomado la decisión correcta. En caso de acierto no hay que hacer nada. Pero si se detecta un fallo de predicción, hay que solucionarlo:

Fallo de predicción de sentido		Fallo de predicción de dirección
Predicción: T (Tomado = salta)	Evaluación: NT (no había que saltar)	La dirección predicha en IF no coincide con la calculada en ID

- Puede haber fallo de predicción por ejemplo si dos instrucciones de salto distintas tienen la misma etiqueta y usamos la dirección de la primera al ejecutar la segunda.
- En caso de detectar un fallo se realizarán tres acciones:
  - **Desactivar** la instrucción incorrecta. Para ello basta con substituir el código de operación de la instrucción que se acaba de leer de memoria por el código de una NOP.
  - **Saltar** a la dirección correcta (que dependiendo de si se salta o no, será PC4\_ID o DirSalto\_ID)
  - **Actualizar** el predictor. Para ello activamos su señal de update y le enviamos los datos correctos (etiqueta, dirección de salto y predicción)

## Resultados

Debéis comprobar que el diseño funciona correctamente. Os damos un ejemplo de cómo hacer un banco de pruebas que ejecuta el programa que está cargado en memoria. Para cargar un nuevo programa hay que escribir las instrucciones en hexadecimal en la memoria de instrucciones. Si queremos que trabaje con unos datos determinados podemos editar el contenido de la memoria de datos. El ejemplo de Moodle no es un banco de pruebas exhaustivo. **Diseñad uno o varios programas de prueba en los que se compruebe que el procesador funciona correctamente para todos los casos representativos.** Cada anticipación, cada parada, o cada caso de predicción debe probarse al menos una vez. La calidad de vuestras pruebas es un elemento clave en la evaluación de este proyecto.

Es obligatorio presentar una memoria con las siguientes partes: a) Explicación breve de vuestro diseño; b) Hardware añadido y qué función lógica realiza; c) Impacto en rendimiento de la gestión de los riesgos que habéis diseñado; d) Pruebas realizadas para verificar que funciona correctamente, incluyendo los fuentes de las mismas; y e) : resumen de vuestras aportaciones, y cuantificación de horas dedicadas por cada miembro del grupo a cada apartado del proyecto.

**IMPORTANTE:** La verificación de que vuestro diseño funciona es una parte fundamental de la práctica. No entreguéis nada sin estar seguros de que funciona. ***Si entregáis un diseño sin daros cuenta de que funciona mal la nota de la práctica será un 0.***

## Anexo 1: Estimación del tiempo que hay que dedicar

Esta es nuestra estimación:

- Paso 1 y toma de contacto con el simulador: 4 horas
- Diseño del paso 2: 1 hora
- Diseño del paso 3: 1 hora
- Diseño del paso 4: 1 hora
- Depuración y ajustes: 10 horas
- Memoria: 3 horas

Estos números asumen que lleváis la asignatura al día y comprendéis lo que hemos hecho en teoría, prácticas y problemas. De no ser así os puede costar 2x, 3x, ∞. No tiene sentido empezar a hacer este proyecto sin entender primero muy bien cómo funciona el MIPS que hemos trabajado en clase.

Asumido lo anterior, diseñar la solución es lo que menos cuesta. Lo que realmente lleva trabajo es comprobar que todo está bien y, si no lo está, solucionarlo. Si lo intentáis hacer a última hora no lo haréis bien y probablemente suspenderéis el proyecto.

Cuando nos deis los datos de tiempo dedicado por favor comparadlo con nuestra estimación y analizar las divergencias. La utilidad de vuestro análisis dependerá de que registréis bien los datos. En otras palabras tratad de ser profesionales y registrar el número de horas que dedicáis cada día en lugar de dar un número a ojo al acabar.

## Anexo 2: Entorno de trabajo


La práctica se puede realizar con **cualquier simulador de VHDL** (por ejemplo para Linux o Mac se puede usar GHDL). Uno de los más utilizados es ModelSim que cuenta con una versión gratuita para estudiantes. Se puede descargar en:

[http://www.mentor.com/company/higher\\_ed/modelsim-student-edition](http://www.mentor.com/company/higher_ed/modelsim-student-edition)

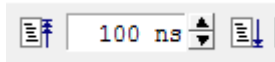
Para usar ModelSim debéis crear un proyecto: **File-> New-> Project**

Después añadir los fuentes que os damos. Al empezar aparece una opción para hacerlo. Además pulsando el botón derecho en la ventana del proyecto podéis elegir **Add to Project -> existing file**.

Desde el proyecto podéis compilar vuestro código y os indicará los errores de sintaxis. Para solucionarlos deberéis usar un editor de texto (no está incluido en ModelSim).

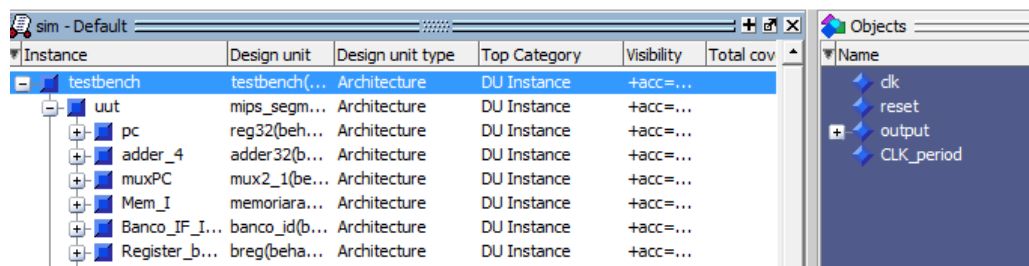
Cuando penséis que vuestro diseño es correcto y lo hayáis compilado hay que simularlo. Para ello debéis definir un banco de pruebas y generar un cronograma. Podéis trabajar a partir de los que os damos (por ejemplo el Mips\_test). **Estos ejemplos son sólo el punto de partida, deberéis modificarlos para probar que todo funciona bien.** Para simular hay que pulsar el botón simular  y después buscar el banco de pruebas. Todas las entidades de vhdl que habéis compilado están por defecto en la librería `work`. Buscad la que queráis simular y seleccionarla. En este simulador podemos:

- Elegir el tiempo que queremos simular: Se escribe en la pestaña



y se presiona el botón de la derecha. Pulsando el de la izquierda reiniciamos la simulación.

- Elegir las señales que queremos ver: para ello en el menú de la ventana "sim" podemos desplegar todos los componentes de nuestro diseño. Al seleccionar un componente aparecen sus señales y las podemos arrastrar a la simulación. Además con el botón derecho y eligiendo la opción de "properties" podemos cambiar el color y el formato (binario, hexadecimal, decimal) de las señales para que sea más fácil seguir la simulación. También es muy útil agruparlas. Si no aparece el cronograma hay que activarlo eligiendo la opción "wave" en el menú "View".



Importante: dar color a algunas señales u ordenarlas para seguirlas mejor es muy útil. Después de hacerlo hay que darle a "save format". Así en la siguiente simulación podéis recuperar el formato usando "load". Se guarda con la extensión `.do`.

En Moodle hay algunos videos que os pueden ayudar a empezar a trabajar con ModelSim

### **ANEXO 3: formato de las instrucciones y de datos fp**

El formato es el que hemos estudiado en clase (tema 2). Algunas matizaciones:

#### **Codificación de los códigos de operación (campo op):**

NOP: 000000      Arit: 000001      LW: 000010      SW: 000011      BEQ:000100  
LA: 001000      BNE: 000101

#### **Codificación de las operaciones de la ALU (campo funct y señal ALU\_ctrl):**

+: 00000      -: 00001      AND:00010      OR: 00011

#### **Tamaño de las memorias de datos e instrucciones:**

Son memorias de 128 palabras de 32 bits. Las direcciones son de 32 bits, pero sólo se tienen en cuenta los 7 correspondientes a las 128 palabras.