

## Proyecto Optativo 1: Fecha de entrega 20 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. Todo esto se hará con el procesador MIPS de 32 bits estudiado en clase.

Partiremos de un MIPS segmentado como el visto en clase **sin anticipación de operandos**, que **resuelve los saltos en la etapa ID**, y **escribe en el banco de registros en los flancos de bajada**. Además este procesador incluye una unidad de suma en FP en la etapa EX. Esta unidad tarda un número de ciclos variable.

Para este procesador nos piden que eliminemos los riesgos estructurales que genera la nueva instrucción, así como que minimicemos y gestionemos los riesgos de datos y de control.

### Paso 1: Trabajo inicial

Descargar de Moodle el código VHDL del procesador y los ejemplos de prueba (se explican allí). Estudiar el código del procesador y comprender cómo funciona. Identificar los componentes del código con los componentes del procesador en las transparencias de clase (es buena idea hacerse una copia ampliada del MIPS y copiar los nombres de las señales usadas en VHDL para conectar y nombrar los componentes principales). Ejecutar el código de ejemplo y ver cómo se ejecuta ciclo a ciclo. Entender cómo las señales van pasando de una etapa a otra atravesando los bancos intermedios. Comprobar que los registros y la memoria tienen el valor deseado. Eliminar los nops y ver cómo las dependencias se convierten en riesgos y el resultado no es el correcto. Ejecutar la prueba con FP y comprobar que el resultado no es el correcto, ya que no existe aún unidad de detección ni se ha modificado la unidad de control para que la FP funcione bien.

### Paso 2: Solucionar el riesgo estructural

Incluir una unidad de suma en FP nos permite incluir una nueva instrucción en el repertorio: ADDFP:

ADDFP rd,rs,rt:

rd  $\leftarrow$  rs ADDFP rt;

PC  $\leftarrow$  PC + 4;

Esta instrucción utiliza los mismos operandos que una add normal pero activa la utilización de la unidad de FP. Para ello se añade una nueva señal de control **FP\_add**. **FP\_add** se debe activar cuando la operación que hay en la etapa EX es una suma en FP. Cuando **FP\_add** esté activo la unidad de suma FP comienza a trabajar. Necesita varios ciclos, cuando termina avisa activando la señal *done*. Cuando esto ocurra el resultado deberá guardarse en ALUout para ello habrá que dar el valor correcto al nuevo mux que hemos incluido para elegir entre la salida de la ALU de enteros y la de la unidad de suma FP:

Como ADDFP ocupa la etapa EX varios ciclos, las instrucciones que hay detrás deberán parar hasta que ADDFP termine, dado que no pueden avanzar porque la etapa EX está ocupada. Debéis incluir el hardware necesario para detectar cuándo hay que parar y modificar el procesador para que detenga las etapas necesarias. También hay que pensar qué se envía a la etapa MEM cuando la etapa EX se detiene. Debéis realizar los cambios necesarios para garantizar que lo que se envía no pueda alterar el estado del sistema.

### Paso 3: Gestión de riesgos de datos

En primer lugar debéis implementar un módulo de anticipación de operandos para reducir los riesgos de datos en la medida de lo posible. Seguiremos una estrategia similar a la vista en clase colocando dos multiplexores en la etapa EX junto a la unidad de anticipación que los controla.

Al igual que en la versión vista en clase todavía existen riesgos de datos. Debéis diseñar un módulo que los detecte y si una instrucción no puede conseguir los operandos que necesita debéis parar tanto ID como IF hasta que el riesgo desaparezca. Cuando se para la etapa ID, como en el caso anterior hay que inyectar a la etapa de ejecución una instrucción modificada de tal forma que no pueda alterar el estado del sistema.

**Importante:** No se deben de realizar paradas innecesarias. Por ejemplo, una NOP no tiene que parar para esperar a sus operandos, o un SW no debe provocar una parada por una dependencia como productor sobre registro porque el SW no produce datos. Para saber si una instrucción escribe o no en registro su registro destino, o si lee un dato de memoria, o si es un salto, lo más fácil es mirar sus señales de control.

### Paso 4: Gestión de riesgos de control

Incorporar a la unidad de detección de riesgos la detección del riesgo de control para que se produzcan las detenciones necesarias, en lugar de tener que recurrir a la solución retardada. Para mejorar el rendimiento se trabajará prediciendo salto no tomado. Es decir cuando tengamos una instrucción de salto en ID, si no se debe saltar no haremos nada, y si se debe saltar deberemos anular la instrucción que acabamos de leer de memoria en la etapa fetch, para ello sustituiremos su código de operación por el de una NOP.

**¡Cuidado!** Nuestros saltos comparan las salidas del banco de registros en la etapa ID y por tanto no se pueden beneficiar de la red de anticipación. La unidad de detención debe tenerlo en cuenta.

**Importante:** hay que tener en cuenta que en un mismo ciclo se pueden producir varias condiciones de parada, por ejemplo un riesgo estructural por un ADDFP y un riesgo de datos o control en la instrucción que lo sigue. El sistema debe funcionar bien en esos casos.

## Resultados

Debéis comprobar que el diseño funciona correctamente. Os damos un par de ejemplos 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. **Los ejemplos de Moodle no son 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. Todos los cortos y paradas posibles deberán probarse al menos una vez. **La exhaustividad y adecuación de vuestros bancos de pruebas es un elemento importante que tenemos en cuenta en la evaluación del proyecto.**

**Es obligatorio presentar una memoria con todas las siguientes partes:** a) Explicación breve de vuestro diseño; b) Hardware añadido y qué función lógica realiza; c) Impacto en rendimiento de cada una de las modificaciones con respecto al procesador inicial; d) Pruebas realizadas para verificar que funciona correctamente, incluyendo los fuentes de las mismas

en la entrega; y e) Conclusiones: 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**

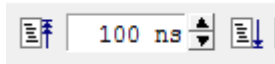
Después añadir los fuentes que os damos. Al empezar aparece una opción para hacerlo. Además pulsando el botón derecho en la ventana del proyecto podéis elegir **Add to Project -> existing file**.

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

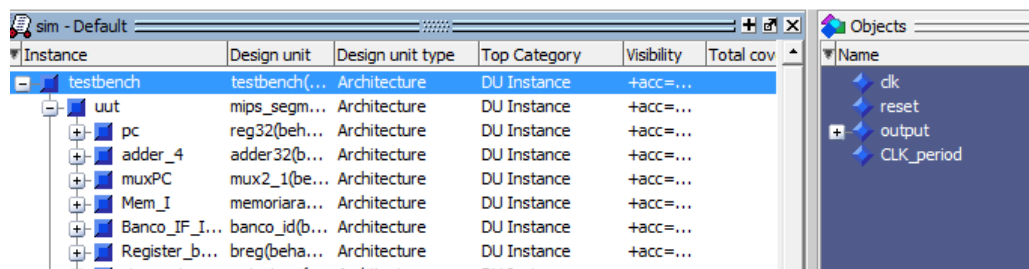
habéis compilado están por defecto en la librería `work`. Buscad la que queráis simular y seleccionarla. En este simulador podemos:

- 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`.

### 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  
ADDFP: 100000

#### 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

Las instrucciones `lw` y `sw` no realizan ninguna conversión al cargar / almacenar un valor de / en memoria. Por tanto, hay codificar a mano los valores en coma flotante que queramos manejar en las pruebas. Muchas calculadoras incorporan la conversión a /de IEEE754. también existen muchas en Internet; os dejamos dos de ellas:

- [http://www.binaryconvert.com/convert\\_float.html](http://www.binaryconvert.com/convert_float.html)
- <https://www.h-schmidt.net/FloatConverter/IEEE754.html>.