Proyecto de Ingeniería de Computadores (PEC)

Objetivo del curso

El objetivo de esta asignatura es que el alumno aprenda a desarrollar un prototipo de un computador o un SoC (System on Chip) en un chip programable sobre una placa base para crear un mini-ordenador. Se pondrán en práctica algunos de los conocimientos adquiridos en asignaturas anteriores sobre el diseño de la microarquitectura de un procesador, sobre el diseño e implementación de software de sistema, y sobre el diseño de sistemas digitales.

Fases principales del proyecto.

1) Aprendizaje de las herramientas de desarrollo para los chips programables (FPGA) y práctica del lenguaje de descripción del hardware VHDL.
2) Implementar pequeños componentes o dispositivos en el chip programable de la placa base.
3) Implementar una primera versión simplificada del procesador en una FPGA (sin memoria externa, ni soporte para el sistema operativo o dispositivos externos)
4) Implementar una versión completa del procesador.
5) Programar un sistema de arranque (BIOS) para el Sistema Operativo en el procesador.
6) Evaluar el rendimiento de varias aplicaciones sobre la plataforma que se ha diseñado.

Objetivo de las sesiones

En estas sesiones vamos a implementar físicamente un procesador sencillo en la FPGA. Lo haremos por etapas. En cada etapa cogeremos el trabajo realizado en la anterior y le añadiremos o modificaremos algún componente.

Etapa 4: Los Saltos.

En esta etapa daremos soporte para las instrucciones de ruptura del secuenciamiento implícito (instrucciones de salto). Se explicará cómo funcionan los saltos en el SISA. Ahora no incluiremos la instrucción de llamada a sistema CALLS, ya que necesitamos tener implementado el modo sistema en el procesador y eso vendrá más adelante.

Las instrucciones de salto que tiene el SISA son: BZ, BNZ, JMP, JZ, JNZ y JAL. Las dos primeras: BZ y BNZ tienen parte de la dirección destino codificada en la propia instrucción. En realidad es el desplazamiento relativo al contador de programa (PC), por tanto, para codificar estas instrucciones sólo se necesita un registro (que será el que se utilice para calcular la condición del salto) y una etiqueta que el ensamblador traducirá en un desplazamiento. Este desplazamiento está dividido por dos ya que todas las instrucciones ocupan dos bytes y el inmediato ocupa los ocho bits de menor peso, por lo tanto no podemos hacer saltos más allá de +255 y -256 bytes desde la posición del PC actual.

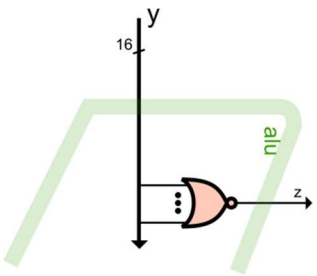
Por otra parte, las instrucciones JMP, JZ, JNZ y JAL tienen un registro que indica la dirección final del salto. Como los registros son de 16 bits, entonces se puede saltar a cualquier dirección direccionable por SISA. Las instrucciones JZ y JNZ, aparte del registro destino del salto, tienen otro registro de control para saber si es cero o no y así poder saltar: JZ salta si este registro tiene un valor igual a cero y JNZ salta si este registro tiene un valor distinto de cero. La instrucción JMP hace un salto incondicional, por lo tanto sólo tiene un registro que es la dirección del salto y JAL salta a la dirección del salto pero salvando en otro registro el PC actualizado (*PCup*).

Consultad la documentación adjunta SISA-F para ver el funcionamiento más detallado de las instrucciones de salto. Antes de implementarlas en el procesador tened claro cómo se codifican estas instrucciones. Recordad que el inmediato se divide por dos al generar el lenguaje máquina, luego el procesador ya se encargará de volver a multiplicarlo. Además, en los saltos condicionales BZ y BNZ los desplazamientos son relativos al PC actualizado (*PCup*) y no al PC actual. Así, si queremos codificar la instrucción `BZ R0,0` (si R0 es igual a cero se queda en la misma instrucción), el lenguaje máquina que le correspondería sería `0x60FF` (al *PCup* hay que sumarle -1*2). Si queremos hacer una instrucción similar pero que avance cuatro instrucciones en el programa, deberíamos escribir la siguiente instrucción `BZ R0,6` en lenguaje

ensamblador, que se codificaría en lenguaje máquina como `0x6003` (al *PCup* hay que sumarle 3*2). Por lo tanto, los inmediatos en las instrucciones de salto en el lenguaje ensamblador SISA no pueden ser valores impares.

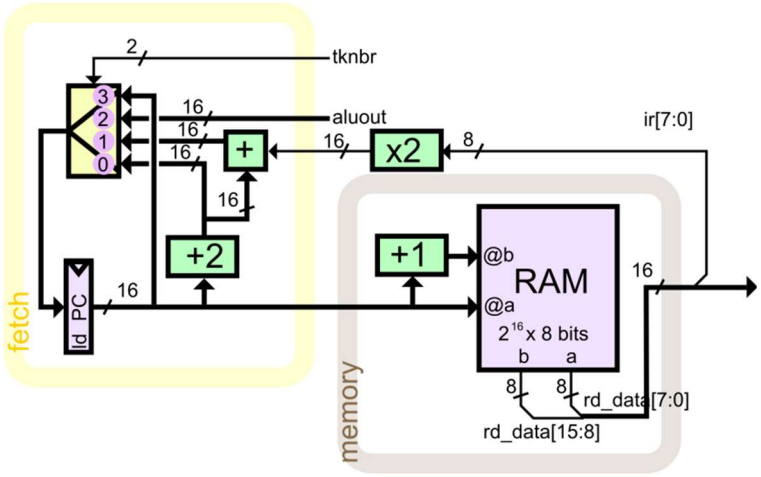
Implementación

Para implementar las instrucciones de salto condicional: BZ, BNZ, JZ y JNZ haremos atravesar el contenido del registro de control por la ALU. En la implementación anterior de la ALU añadimos un bit ***z*** de salida que indica si el valor de la salida ***w*** de la ALU es igual o distinto del valor 0. Este bit será el que utilizaremos para averiguar si el contenido del registro de control es igual o distinto de cero. Así es como se os mostró en la implementación SISP-I-1 que estudiasteis en IC. Pero el lenguaje máquina SISA no específica nada sobre ello, por tanto se podría implementar de muchísimas formas. Por ejemplo, también se podría haber colocado el circuito que calcula la salida ***z*** conectado a la entrada ***y*** de la ALU como podemos ver en la siguiente figura y ahorrarnos el proceso anterior. Esto es porque no necesitamos hacer ninguna operación con el contenido de este registro de control y además podemos usar la ALU para realizar cualquier otra operación. La elección es vuestra.



Fetch

Los cambios que debemos hacer en la parte de control es donde tenemos el PC, habrá que añadir un multiplexor y pasar dos nuevas entradas: la nueva dirección relativa al PC en el caso de las instrucciones BZ y BNZ y la segunda entrada añadida, la dirección absoluta para las instrucciones de tipo Jxx. Un posible esquema simplificado de esta parte del *fetch* queda representado en la siguiente figura (la última entrada del multiplexor, que contiene el mismo PC, no es necesario implementarla ahora, ya que sólo se usa en caso un de fallo del TLB que aún no hemos implementado).



Módulo del datapath

También debemos hacer una pequeña modificación en el módulo de ejecución para el caso de la instrucción JAL. El problema es que no tenemos preparado el procesador para poder almacenar el contenido del PC en un registro general para poder hacer el *Jump and Link* que esta instrucción realiza.

En este caso, podemos hacer pasar el PC actualizado al módulo de ejecución (*datapath*) y se lo añadiremos de entrada al multiplexor que selecciona la entrada al registro de escritura ***d***. Para ello deberemos actualizar las correspondientes señales de control del multiplexor.

Módulo de control

Ahora tenemos que añadir la lógica de control de estas nuevas señales que tenemos y modificar ligeramente la de otras que ya usábamos. Ahora, como novedad, las señales *tknbr* (que es nueva) y *in_d* son de dos bits cada una.

Para poder decodificar e implementar correctamente las instrucciones es necesario saber cómo se codifican. La siguiente tabla muestra un resumen la codificación de las nuevas instrucciones.

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0		
0 1 1 0	Rb	0	n n n n n n n n		Salto condicional modo relativo al PC	BZ
		1				BNZ
1 0 1 0	Rb	Ra	0 0 0 f f f		Ruptura de secuencia modo registro	JZ, JNZ
	0 0 0	Ra				-, JMP
	Rd	Ra				JAL, -, -, CALLS
			f f f f f f		Reservadas Fut. Ampl.	42 códigos

El código de operación de las instrucciones BZ y BNZ es el 6, mientras que el de las instrucciones Jxx es el 10. Para diferenciar las instrucciones Jxx se utilizan los bits 2:0 tal como se indica en el manual del SISA.

Podemos utilizar la ALU si hemos puesto el circuito que calcula la salida ***z*** conectado a la entrada ***y*** de la ALU, para que en las instrucciones Jxx deje pasar la dirección absoluta del salto y se pueda enviar a la unidad de control. No debe calcular nada, simplemente copiar en la salida ***w*** el valor que le llega por la entrada ***x***. Otra alternativa hubiese sido coger directamente el valor que sale por el puerto ***a*** de Banco de registros y enviarlo a la unidad de control que se encarga del *fetch*.

También debemos notar que el registro de control está codificado en los bits 11:9 para las instrucciones de salto condicionales (BZ, BNZ, JZ y JNZ), por tanto, tenemos que modificar la lógica de la señal *Rb_N* para hacer pasar el registro de control por bus ***y*** de la ALU cuando sea conveniente.

También debemos tener en cuenta que escribiremos en el banco de registros en el caso de la instrucción JAL, por lo tanto tenemos que añadir la lógica necesaria para la señal del permiso de escritura del Banco de registros.

La lógica del salto es la más complicada (señal *tknbr*) porque debemos tener en cuenta el bit ***z*** de la ALU y si se trata de las instrucciones Jxx, Bxx o un HALT. También debemos comprobar cómo queda la lógica de la señal de control *ldpc*.

Con esto ya tenemos toda la lógica necesaria para que los saltos puedan funcionar.

Implementad en vhdl un nuevo diseño con estas modificaciones.

Simulación y ejecución de programas

El procesador ya está listo para probar las nuevas instrucciones. Podemos probar el funcionamiento de estas nuevas instrucciones de varias formas

a) Simulando nada más la entidad **proc** con el *stub* de memoria de la etapa 2.1
b) Simulando la entidad **sisa** con el *stub* de memoria de la etapa 2.2 que emulaba a una SRAM comercial.
c) Programando la FPGA con nuestro diseño. Previamente habremos cargado el programa que deseemos ejecutar en la memoria SRAM mediante la aplicación del panel de control de las placas de desarrollo.

Juegos de prueba

Con estas modificaciones realizadas, probaremos los saltos con dos juegos de prueba. Uno probará las instrucciones BZ y BNZ y el otro las instrucciones JMP, JZ, JNZ y JAL. Obviamente no se pueden probar todos los casos, pero sí los más significativos.

Saltos relativos

Para probar las instrucciones de salto relativo, para cada una escogeremos una situación en la que salta y otra situación en que no salta. Por lo tanto, probaremos 4 casos. El siguiente código corresponde a este juego de pruebas.

```
      movi r0, 0
      movi r1, 1
      movi r2, 2
      bz   r1, a
      st   0(r0), r1
   a: st   2(r0), r2
      bz   r0, b
      st   4(r0), r1
   b: bnz  r1, c
      st   6(r0), r2
   c: bnz  r0, d
      st   8(r0), r1
   d: halt
```

Como podemos ver, el programa hace una primera instrucción BZ en la línea 4 y no debería saltar porque el registro de control es el *r1* y este contiene un 1, por lo tanto el programa tendría que escribir en la posición de memoria 0 y 2 que son los dos primeros *stores* que hay en el código. Después, en la línea 7, nos encontramos con otra instrucción BZ pero con el registro *r0*, que contiene un cero y en este caso debería saltar y no realizarse el tercer *store*.

Posteriormente tenemos dos instrucciones BNZ, la primera salta y la segunda no. Por lo tanto, debería producirse el último *store* en la dirección 8. El resultado de la memoria debería ser el siguiente:

```
Mem[01:00] = 0001
Mem[03:02] = 0002
Mem[09:08] = 0001
```

Escribimos en la memoria el programa y después de ejecutar el procesador con los cambios, verificamos que efectivamente el contenido de la memoria es lo que debería ser.

Fijaos que este código de ejemplo no realiza ningún salto a posiciones anteriores a la actual. Cread un código que compruebe el funcionamiento de los saltos hacia atrás.

Saltos absolutos

La prueba con las instrucciones de salto absoluto es un poco más compleja, ya que una vez ensamblado el programa, debemos desensamblarlo para saber cuáles son las direcciones finales de los saltos, ya que no podemos usar etiquetas (al menos sin el enlazador o *linker*). Como el programa se ejecutará a partir de la dirección `0xC000`, debemos tener en cuenta los desplazamientos y la cantidad de instrucciones que ponemos (ya que cada instrucción ocupa 2 bytes).

En este caso, preparamos 4 porciones de código de cuatro instrucciones cada una, separadas entre sí y terminadas en la instrucción HALT. La penúltima instrucción de cada porción de código será un salto. Si no salta, simplemente el programa se detiene (debido al HALT y no hace lo que debería hacer). El código de esta prueba es el siguiente:

```
movi r0, 0
movi r1, 1
movi r2, 14
movi r3, 3
shl  r3, r3, r2
movi r5, 32 ; @ s2
movi r4, 40 ; @ s1
movi r7, 48 ; @ s4
movi r6, 56 ; @ s3
or   r4, r4, r3
or   r5, r5, r3
or   r6, r6, r3
or   r7, r7, r3
movi r2, 64
jz   r0, r4 ; Salta a s1
```

```
    halt
.org 32
s2:
    st   2(r2), r1
    addi r1, r1, 1
    jmp  r6 ; Salta a s3
    halt
s1:
    st   0(r2), r1
    addi r1, r1, 1
    jnz  r1, r5 ; Salta a s2
    halt
s4:
    st   6(r2), r1
    st   8(r2), r3
    st   0(r0), r1
    halt ; Fin del programa
s3:
    st   4(r2), r1
    addi r1, r1, 1
    jal  r3, r7 ; Salta a s4
    halt
```

Si nos fijamos en el programa, las primeras 15 instrucciones son para construir las direcciones de las etiquetas de *s1*, *s2*, *s3* y *s4*. Una vez se han acabado de inicializar los registros, simplemente empezamos los saltos con la instrucción JZ que debería saltar, si no el programa se cuelga en el HALT posterior.

El programa salta de s1 a s2, de s2 a s3 y de s3 a s4. El último salto, es con la instrucción JAL que debería almacenar la dirección actual en *r3* y luego en *s4* la almacena en la memoria. El contenido final de la RAM, debería ser el siguiente:

```
Mem[01:00] = 0004
Mem[41:40] = 0001
Mem[43:42] = 0002
Mem[45:44] = 0003
Mem[47:46] = 0004
Mem[48:48] = c03e (@ jal +2)
```

Después de ejecutar estos juegos de pruebas, también deberíamos pasar los juegos de prueba hechos en las etapas anteriores para garantizar que no hemos añadido ninguna incompatibilidad con los diseños anteriores.