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.

Características del procesador a implementar

El procesador que implementaremos estará basado en la arquitectura SISA (*Simple Instruction Set Architecture*). Se trata de un procesador RISC de 16 bits con 8 registros de propósito general y 8 específicos para coma flotante.

En este documento no incluiremos los detalles relativos al funcionamiento completo de esta arquitectura ni al funcionamiento de cada una de las instrucciones del repertorio de instrucciones. Toda esta información está en unos documentos adjuntos (SISA-I, SISA-F y SISA-S). Sí que introduciremos una visión general de la arquitectura y una tabla resumen con el juego de instrucciones.

Visión general de la arquitectura

En primer lugar veamos una breve visión general de la arquitectura, indicando los aspectos relevantes desde el punto de vista del lenguaje máquina (estado del computador, tipos de datos, modos de direccionamiento, formato de las instrucciones, etc.).

SISA es una arquitectura RISC de 16 bits, que puede operar con números naturales, enteros y en coma flotante pequeños (small-floats) de 16 bits. Destacamos los siguientes aspectos:

- El conjunto de instrucciones es reducido. Solamente hay 55 instrucciones diferentes.
- Todas las instrucciones de lenguaje máquina son de un tamaño fijo de 16 bits (2 bytes, 1 palabra).
- En el procesador hay 8 registros generales de 16 bits (R0, R1,..., R7), 8 registros (F0,…, F7) de coma flotante de 16 bits (small-floats) y 8 registros con funcionalidades especiales (S0,…, S7)
- Todos los operandos fuente y destino de las instrucciones que efectúan cálculos aritméticos, lógicos, comparaciones, etc. tienen los operandos en registros del procesador y dejan el resultado en uno de esos registros. Estas instrucciones tienen dos registros fuente y uno destino que pueden ser distintos. Para mover datos entre memoria y registros existen instrucciones de tipo load (LDB, LD y LDF) y store (STB, ST, STF).

- La memoria almacena el programa en lenguaje máquina que se está ejecutando y los datos del programa. La memoria se direcciona a nivel de byte (de 8 bits). Las direcciones de memoria son de 16 bits. La capacidad de la memoria es de 2^16 palabras de 8 bits cada una.
- El procesador puede generar diversas excepciones: Instrucción ilegal, dirección de memoria mal alineada, *overflow* en operación de coma flotante, división por cero en operación de coma flotante, división por cero en operación de enteros, con o sin signo, etc.
- Hay instrucciones para la gestión de entrada/salida y para la gestión de interrupciones.

Memoria

Esta arquitectura tiene un único espacio de direccionamiento de memoria, que se usa tanto para código como para datos. Una dirección de memoria consta de 16 bits y la unidad de direccionamiento es el byte: se pueden direccionar hasta 2^16 bytes de memoria.

El acceso a datos se efectúa mediante las instrucciones tipo load (lectura) y store (escritura), en sus variantes de byte (8 bits: LDB, STB) y word o float (16 bits: LD, ST y LDF, STF). La instrucción LDB lee un byte de memoria y lo escribe en un registro general R, extendiendo el bit de signo hasta completar los 16 bits del registro R. La instrucción STB escribe en un byte de memoria los 8 bits de menor peso de un registro general R. Las instrucciones LD y ST transfieren words (16 bits, 2 bytes) entre los registros generales R y memoria mientras que las instrucciones LDF y STF, que también transfieren 16 bits, 2 bytes, lo hacen entre los registros de coma flotante F y memoria. La dirección lógica de memoria a la que se accede con estas instrucciones se obtiene según el modo de direccionamiento registro base más desplazamiento.

Los accesos a datos deben estar alineados a su tamaño natural:

- Los accesos a palabras, con las instrucciones LD y ST, deben estar alineados a direcciones múltiplo de 2 (con el bit de menor peso de la dirección con valor 0).
- Los accesos a floats, con las instrucciones LDF y STF, deben estar alineados a direcciones múltiplo de 2 (con el bit de menor peso de la dirección con valor 0).
- Los accesos a bytes, con las instrucciones LDB y STB, siempre están alineadas por definición, puesto que el byte es la unidad de direccionamiento.

Los bytes consecutivos de memoria que forman un dato de tamaño word (16 bits, 2 bytes) o float (16 bits, 2 bytes) o una instrucción (16 bits, 2 bytes) se numeran, o direccionan, siguiendo el convenio little-endian.

- Como el byte es la unidad de direccionamiento (la mínima cantidad de memoria que se puede direccionar) no tiene sentido hablar de sistema de direcciones little-endian o big-endian cuando se trata de acceder a un dato de tamaño byte. Las únicas instrucciones que soportan datos de tamaño byte son la LDB, que carga un byte de memoria en un registro general, extendiendo el bit se signo para ocupar los 16 bits del registro y la instrucción STB que almacena en memoria los 8 bits de menor peso del registro general fuente.
- Un word en memoria lo forman 2 bytes contiguos con direcciones @ y @+1, estando la dirección @ alineada a word, esto es, con el bit de menor peso a cero, @<0> = 0. El word se direcciona con la dirección @ (la instrucción LD o ST calcula la dirección efectiva @. El byte contenido en la dirección @ es el byte de menor peso de la palabra (el que tiene los bits 7:0) y el de dirección @+1 el de mayor peso (bits 15:8).
- Un float en memoria lo forman 2 bytes consecutivos siguiendo el mismo convenio que para el Word

Cualquier instrucción a ejecutar debe estar en memoria alineada a direcciones múltiplo de 2 (con el bit de menor peso de la dirección con valor 0, como un dato de tamaño word, o como un float).

Cualquier acceso a memoria mal alineado, tanto para instrucciones como para datos, genera una excepción del tipo "Acceso a memoria mal alineado".

Secuenciamiento Implícito. Registro PC

SISA es una arquitectura con secuenciamiento implícito. Tiene un registro especial denominado PC (*Program Counter*) que direcciona la memoria para efectuar la búsqueda de las instrucciones. Todas las instrucciones, después de su ejecución, dejan el valor del PC modificado. Después de ir a buscar a memoria la instrucción que indica el PC, y antes de decodificarla, se incrementa el PC en 2 unidades, ya que 2 es el tamaño en bytes de una instrucción, (PC←PC + 2). Esta actualización del PC se hace sea cual sea la instrucción a ejecutar, ya que se efectúa antes de su decodificación. El valor del PC durante la ejecución, propiamente dicha, de la instrucción, indica cuál es la dirección de memoria donde se encuentra la siguiente instrucción a ejecutar. A este valor del PC, durante la ejecución de una instrucción, lo denominamos PC actualizado, PCup (*updated*).

Las instrucciones de ruptura de secuencia (saltos) son las únicas instrucciones que durante su ejecución pueden volver a modificar el valor del PC. Las instrucciones de salto que calculan la dirección destino del salto (dirección de la siguiente instrucción a ejecutar) de modo relativo al PC lo hacen sumando un desplazamiento al PC actualizado, PCup. El desplazamiento, que se encuentra en la instrucción codificado con 8 bits, se multiplica por 2, antes de ser sumado al PCup. Se multiplica por 2 para poder acceder a instrucciones más alejadas de la actual que si se sumara el desplazamiento tal cual, y porque las instrucciones siempre se encuentran alineadas en direcciones pares de memoria. Se puede acceder a instrucciones que están desde -127 a 128 instrucciones de la instrucción de salto ((PC+2-128*2)/2,...,PC+2+127*2)/2).

En el documento adjunto, para no resultar repetitivo, al explicar que hace cada instrucción al ejecutarse, no se especifica explícitamente como se incrementa el PC antes de la decodificación de la instrucción. Esto es, no se incluye la acción: PC ← PC + 2.

Bancos de registros

En uno de los documentos adjuntos puede verse las características de los bancos de registros: generales (R), de coma flotante (F) y especiales (S) donde además se incluye la funcionalidad de cada uno de estos registros especiales.

Entrada/salida, tipos de datos, modos de direccionamiento, excepciones/interrupciones

En uno de los documentos adjuntos puede verse toda la información detallada de estos temas. No los resumimos aquí ya que para esta primera implementación de procesador no serán necesarios.

Repertorio de instrucciones

En uno de los documentos adjuntos se detalla el conjunto de instrucciones de lenguaje máquina, agrupadas por familias, donde los miembros de cada familia comparten algún tipo de función. Para cada familia se detallan las instrucciones que la forman, los mnemotécnicos usados en el lenguaje ensamblador SISA, el código binario y la función que realiza cada instrucción. Por último, en el anexo de ese documento se muestra una ficha para cada instrucción en la que se indica el formato en código binario, la semántica de la instrucción, la sintaxis en ensamblador, una descripción textual de su funcionamiento y algún ejemplo de cómo la instrucción modifica el estado del computador.

El juego de instrucciones del SISA está formado por 6 grandes bloques de instrucciones: las instrucciones de operación, las de comparación, las de acceso a memoria, las de salto, las instrucciones de movimiento de inmediato a registro y las instrucciones de sistema.

A continuación describiremos brevemente estos conjuntos de instrucciones.

Instrucciones de operación

Las instrucciones aritméticas ADD y SUB manipulan datos enteros de 16 bits en complemento a 2 como naturales de 16 bits en binario, ya que en ningún caso se detecta si el resultado se representable o no en 16 bits según la codificación utilizada.

Para desplazar bits, tenemos las instrucciones SHA y SHL, que escriben en el registro destino el resultado de desplazar el operando A a la derecha o a la izquierda el número de bits que indica el operando B, interpretado como un número en complemento a 2. Valores positivos indican desplazamiento a la izquierda y valores negativos desplazamiento a la derecha. La instrucción SHA realiza desplazamientos aritméticos y la instrucción SHL, desplazamientos lógicos.

También tenemos las instrucciones lógicas AND, OR, XOR y NOT que hacen respectivamente las operaciones bit a bit and, or, xor y not.

Para multiplicar, tenemos las instrucciones MUL, MULH y MULHU. La instrucción MUL escribe en el registro destino los 16 bits de menor peso de la multiplicación del operando A por B. El resultado es correcto tanto para número enteros como para naturales. No se detecta overflow ni para enteros ni para naturales.

La instrucción MULH escribe en el registro destino los 16 bits de mayor peso de la multiplicación del operando A por B, interpretando los operandos y el resultado como números enteros codificados en complemento a dos.

La instrucción MULHU escribe en el operando destino los 16 bits de mayor peso de la multiplicación del registro origen A por B, considerando números naturales codificados en binario.

Del mismo modo, tenemos las instrucciones DIV y DIVU que hacen la división entera en el primer caso (los operandos los trata como enteros) y la división natural (los operandos los trata como naturales).

Finalmente, en este apartado también tenemos una versión de la suma, pero con inmediato: ADDI, que escribe en el registro destino los 16 bits de menor peso de la suma del contenido del operando A con la extensión de signo del inmediato de 6 bits. En este caso, el número entero debe ser representable en complemento a dos en 6 bits, por lo tanto su rango de valores debe estar entre: $-32 \leq N \leq 31$.

Instrucciones de comparación

Para comparar, tenemos las instrucciones CMPLT, CMPLE, CMPEQ, CMPLTU y CMPLEU, que respectivamente hacen la comparación '$<$' de enteros, '$\leq$' de enteros, '==', '$<$' de naturales y '$\leq$ ' de naturales.

Instrucciones de acceso a memoria

Para acceder a la memoria, el SISA lo puede hacer de dos formas: en modo word de 16 bits o en modo byte de 8 bits. En el primer caso, las instrucciones de load y store no llevan ningún sufijo, pero en el caso de las operaciones de byte llevan el sufijo B.

Así, las instrucciones LDB y STB llevan un inmediato de 6 bits que se extiende para calcular la dirección efectiva, formada por la suma del valor inmediato con el registro indicado en el operando. El resultado de la operación con la instrucción LDB es leer 8 bits de memoria y colocarlos en los 8 bits de menor peso del registro destino, extendiendo el signo a los 8 bits de más peso. La instrucción STB escribe un byte en memoria en la dirección especificada por el registro y el inmediato

En el caso de las instrucciones LD y ST, el inmediato se multiplicará por dos ya que no se admiten direcciones efectivas impares en este modo.

Instrucciones de salto

Hay dos tipos de instrucciones de salto en el computador: las relativas al PC y las absolutas. Las primeras son BZ y BNZ. Estas llevan un inmediato de 8 bits codificado en la instrucción, que se multiplicará por dos y se sumará al PC para encontrar la dirección efectiva del salto. La primera salta en el caso de que el registro valga cero y la segunda lo hace cuando el valor sea distinto de cero.

Las instrucciones de salto que trabajan con direcciones absolutas son JZ, JNZ, JMP y JAL. Las dos primeras toman el registro del operando A y lo copian en el PC en el caso de que el operando B sea cero o sea diferente de cero respectivamente. La instrucción JMP salta siempre y simplemente toma el registro y lo copia en el PC. Finalmente, la instrucción JAL implementa el *jump and link*, funciona de manera similar al JMP pero la siguiente dirección en la secuencia la almacena en el registro destino.

Movimiento de inmediato a registro

Las instrucciones que nos permiten copiar inmediatos a registros son MOVI y MOVHI. La primera toma el inmediato codificado en 8 bits y lo escribe en el registro destino de 16 bits, extendiendo el signo.

En el caso de la instrucción MOVHI, los 8 bits del inmediato se copian en la parte alta del registro (8 bits de más peso). De esta forma se pueden copiar valores de 16 bits completos con estas dos instrucciones: primero hay que ejecutar el MOVI para escribir la parte baja y después MOVHI para sobrescribir la parte de la extensión del signo y tener el valor completo de 16 bits en el registro.

Instrucciones de sistema

Estas instrucciones sólo se pueden ejecutar en el modo privilegiado del procesador. La ejecución de estas instrucciones en el modo usuario provoca que se lance la excepción de protección y sea capturada por el sistema operativo.

Las instrucciones EI y DI habilitan o inhiben las interrupciones respectivamente. Dentro del gestor de interrupciones, la instrucción RETI vuelve al modo usuario restaurando el registro S1 en el PC y los *flags* del registro S0 al S7.

La instrucción GETIID obtiene el identificador de interrupción cuando estamos en la RSI (rutina de servicio de interrupciones). En el momento de ejecutarla, copiamos el identificador en el registro destino de la instrucción.

También tenemos las instrucciones RDS y WRS para leer y escribir respectivamente el banco de registros de sistema, ya que tenemos un conjunto de registros especiales que sólo se pueden leer y escribir en el modo sistema.

Finalmente, tenemos el conjunto de instrucciones que operan con el TLB (*Translation Lookaside Buffer*) de datos y de instrucciones: WRPI, WRPD, WRVI y WRVD. Las instrucciones que terminan con una 'I' escriben en el TLB de instrucciones, mientras que las instrucciones acabadas con una 'D" lo hacen en el TLB de datos. Las que tienen una 'P' escriben el *tag* físico, en la posición indicada por el registro del operando A y las que tienen una 'V', escriben el *tag* virtual.

En la siguiente tabla tenéis un resumen del formato y codificación de las 55 instrucciones SISA

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0		
0 0 0 0	Rd	Ra	f f f	Rb	Op. Lógicas y Aritméticas	AND, OR, XOR, NOT ADD, SUB, SHA, SHL
0 0 0 1	Rd	Ra	f f f	Rb	Comparación con y sin signo	CMPLT, CMPLE, -, CMPEQ CMPLTU, CMPLEU, -, -
0 0 1 0	Rd	Ra	n n n n n n		Add inmediato	ADDI
0 0 1 1	Rd	Ra	n n n n n n		Load	LD
0 1 0 0	Rb	Ra	n n n n n n		Store	ST
0 1 0 1	Rd	0	n n n n n n n n		Mover Inmediato	MOVI
	Ra Rd	1				MOVHI
0 1 1 0	Rb	0	n n n n n n n n		Salto condicional modo relativo al PC	BZ
		1				BNZ
0 1 1 1	Rd	0	n n n n n n n n		Input	IN
	Rb	1			Output	OUT
1 0 0 0	Rd	Ra	f f f	Rb	Extensión aritmética	MUL, MULH, MULHU, - DIV, DIVU, -, -
1 0 0 1	Fd	Fa	f f f	Fb	Op/Cmp Float	ADDF, SUBF, MULF, DIVF CMPLT, CMPLEF, -, CMPEQF
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, -, -
	0 0 0	Ra				CALLS
			f f f f f f		Reservadas Fut. Ampl.	42 códigos
1 0 1 1	Fd	Ra	n n n n n n		Load Float	LDF
1 1 0 0	Fb	Ra	n n n n n n		Store Float	STF
1 1 0 1	Rd	Ra	n n n n n n		Load Byte (8 bits)	LDB
1 1 1 0	Rb	Ra	n n n n n n		Store Byte (8 bits)	STB
1 1 1 1			0		Reservadas Fut. Ampl.	32 códigos
	0 0 0	0 0 0	1	f f f f f	Instrucciones especiales	EI, DI, -, -
	0 0 0	0 0 0				RETI, -, -, -
	Rd	0 0 0				GETIID, -, -, -
	Rd	Sa				RDS, -, -, -
	Sd	Ra				WRS, -, -, -
	Rb	Ra				WRPI, WRVI, WRPD, WRVD
	0 0 0	Ra				FLUSH, -, -, -
	1 1 1	1 1 1				-, -, -, HALT

Estrategia de diseño

Para describir el procesador, seguiremos una estrategia cíclica de: diseño, implementación y prueba, para los diferentes tipos de instrucciones del lenguaje máquina. Para hacer esto, primero partiremos de un procesador base que sólo será capaz de mover inmediatos a registro. Es decir, sólo implementará las instrucciones MOVI, MOVHI y HALT. Una vez verificado este diseño, iremos implementando las diferentes instrucciones por tipo según este orden:

1. Procesador base. Instrucciones MOVI, MOVHI y HALT.
2. Acceso a memoria. Instrucciones: LD, ST, LDB y STB. A partir de esta etapa, ya se podrá probar en una implementación física y simularlo adecuadamente.
3. Instrucciones aritmético-lógicas, comparaciones, sumas con inmediato, multiplicaciones y divisiones de enteros.
4. Saltos relativos y absolutos, con o sin condición. Instrucciones: JZ, JNZ, JMP, JAL, BZ y BNZ.

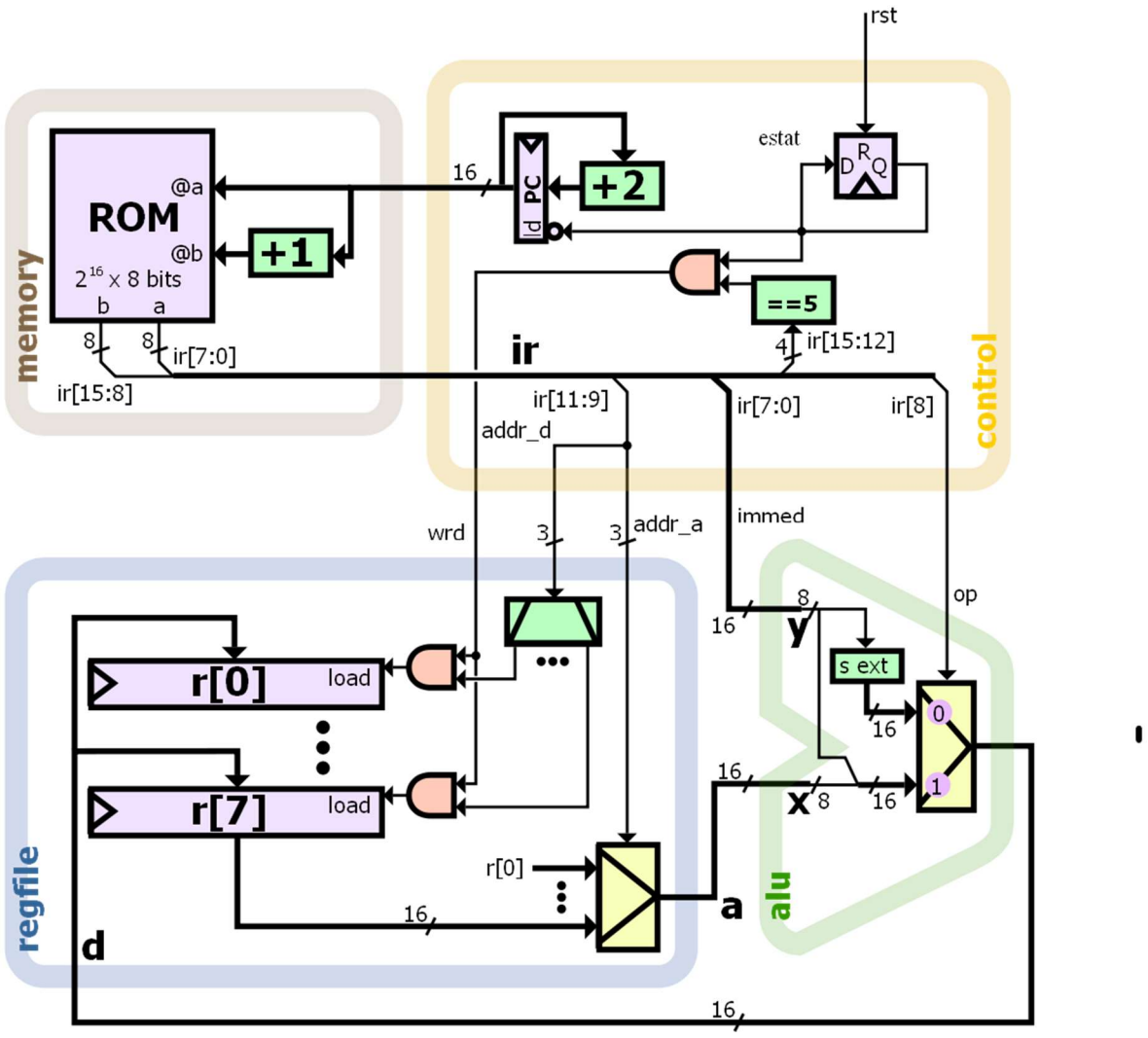
5. Entrada / Salida: se implementarán las instrucciones IN y OUT sobre dos dispositivos sencillos. A continuación se añadirá soporte para el procesador pueda recibir interrupciones y finalmente se implementarán todos los dispositivos junto con las instrucciones EI, DI y GETIID.
6. Instrucciones de sistema: RDS, WRS, CALLS, RETI, WRPI, WRPD, WRVI y WRVD.
7. Excepciones.

Etapa 1: El procesador base

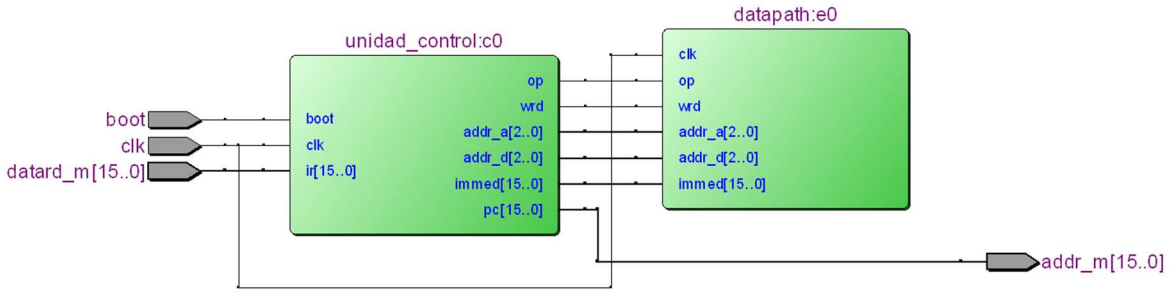
Lo primero que vamos a realizar es un procesador que sólo sea capaz de ejecutar las instrucciones MOVI, MOVHI y HALT. Este procesador tiene que tener una unidad de control que haga el *fetch* de memoria y decodifique la instrucción, así como una unidad de ejecución que haga pasar el inmediato por la ALU y lo almacene en un registro del Banco de registros.

En esta etapa, el sistema no dispone de memoria de datos, sólo de instrucciones. Por tanto, el sistema se simplifica considerablemente. El sistema completo constará de una memoria de instrucciones y del procesador base. Esta memoria contendrá instrucciones MOVI / MOVHI / HALT en el lenguaje máquina del SISA.

En la siguiente figura podéis ver un esquema (de los muchos posibles) de todo el sistema a nivel de bloques. El bloque de la memoria, la unidad de control, el banco de registros y la ALU.



En la siguiente figura podemos ver un esquema de las conexiones de los dos grandes bloques del procesador: La unidad de control y el datapath (banco de registros + ALU).



Memoria

Como las instrucciones MOVI y MOVHI no hacen accesos a memoria, sólo utilizaremos la memoria para leer las instrucciones a ejecutar. Es decir, por el momento será una memoria ROM (de sólo lectura). Esta memoria debe poder leer palabras de 16 bits, pero las direcciones serán de acceso a byte. Por lo tanto, para acceder a la primera palabra, la dirección será la 0, mientras que para acceder a la siguiente palabra, la dirección será la 2.

Para ello, utilizamos una memoria de 65.536 posiciones (2^{16}), ya que 16 son los bits que tiene la dirección. Cada posición tendrá un byte. Para acceder a una palabra, haremos dos accesos de tipo byte: uno para obtener el byte de menor peso y el otro para acceder al de mayor peso, en este caso sólo tendremos que sumar 1 a la dirección y obtener el byte correspondiente. El dato completo, en este caso una instrucción, estará formado por la salida del primer byte en los bits del 7 al 0 y el segundo byte serán los bits del 15 al 8.

Por la salida *rd_data* sale el contenido de la memoria con el signo extendido si se trata de un byte o bien sale una palabra formada por el contenido de la dirección + 1 en el byte de mayor peso, y el contenido de la dirección indicada en el byte de menor peso.

En el siguiente esquema podéis ver las entradas y salidas de la memoria. Este bloque ya os lo daremos implementado en VHDL.

Banco de registros

El banco de registros tiene ocho registros de propósito general. De momento y sólo necesitamos un puerto de lectura y uno de escritura.

Unidad Aritmético Lógica (ALU)

La ALU debe poder ejecutar las dos instrucciones MOVI y MOVHI, para ello el bloque tendrá dos entradas de 16 bits (los operandos *x* e *y*) y una salida también de 16 bits (operando *w*). Lo único que tiene que hacer es copiar uno de los operandos de la entrada *y* en la salida (para la operación MOVI) y combinar los 8 bits de menor peso de los operandos *x* e *y* para formar un número de 16 bits (para la instrucción MOVHI).

Unidad de Control

El módulo de control es el más complejo. Debe gobernar las señales de todo el computador. En esta etapa, el procesador ejecuta una instrucción por ciclo, ya que no tenemos ningún riesgo estructural. En este bloque, el PC se carga en el siguiente ciclo con el valor PC +2, excepto cuando nos encontramos con la instrucción HALT. En este caso, el procesador ya no carga el PC con ningún otro valor.

Este bloque tiene como entrada la instrucción en código máquina del SISA (señal *ir*) y como salidas la dirección del registro de lectura (*addr_a*), la de escritura (*addr_d*), la señal de permiso de escritura del registro (*wrd*), la operación a realizar en la ALU (op) y el inmediato (*immed*).

Empecemos con la implementación

Primera versión del Banco de registros.

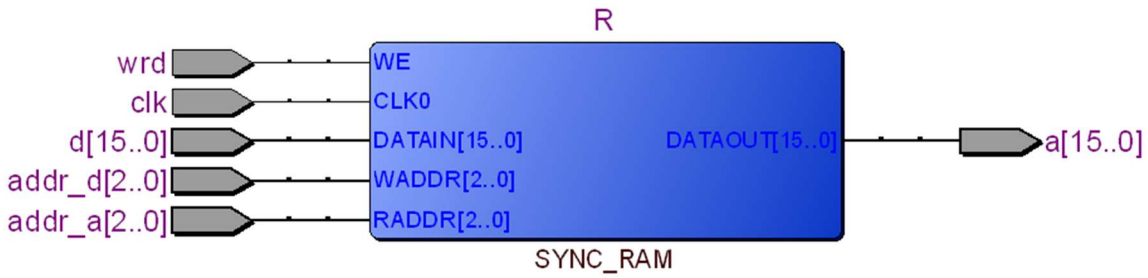
En esta sección vamos a realizar la primera versión del banco de registros del procesador. Esta primera versión estará simplificada con respecto a la final, de manera que sólo dispondremos de un puerto de escritura y otro de lectura.

El banco de registros está compuesto por 8 registros de 16 bits cada uno. Estos registros están direccionados mediante señales de 3 bits, *addr_a* para seleccionar el registro fuente y *addr_d* para seleccionar el registro destino.

La señal de control *wrd* indica cuando se permite la escritura en el registro seleccionado por *addr_d*. El valor debe escribirse en el registro en el instante del flanco ascendente de la señal de reloj *clk*.

El puerto de escritura en el banco de registros se llama *d* y el puerto de lectura *a*, ambos de 16 bits.

En la siguiente figura podemos ver las entradas y salidas del Banco de registros



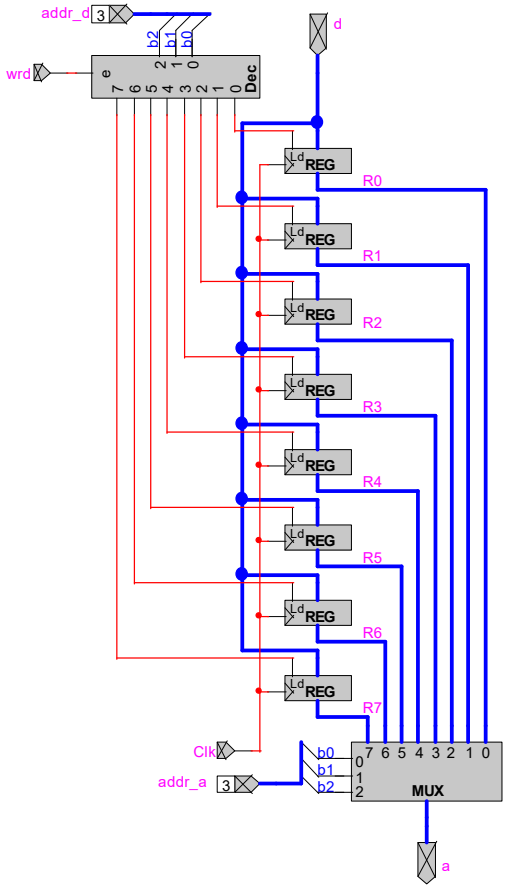
Así pues, la cabecera del diseño en vhld es como sigue:

```
ENTITY regfile IS
  PORT (clk         : IN  STD_LOGIC;
        wrd         : IN  STD_LOGIC;
        d           : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        addr_a      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        addr_d      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        a           : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END regfile;

ARCHITECTURE Structure OF regfile IS
BEGIN
        .
        .
        .
END Structure;
```

Internamente, el banco de registros está compuesto por 8 registros de 16 bits cada uno conectados al puerto de escritura *d*. Para facilitar su comprensión veamos un posible esquema de bloques. En ese esquema se podría utilizar un decodificador

de entrada *addr_d* para seleccionar el registro a escribir, siempre que la señal de permiso de escritura *wrd* esté activa. Así mismo, un multiplexor cuya selección es la señal *addr_a* elige que salida de registro alimentará al puerto de salida *a*. Un esquema de esta descripción del comportamiento se muestra en la siguiente figura.



La implementación en VHDL de un banco de registros no tiene porque parecerse en nada al esquema de bloques presentado. Es más, se puede hacer con muy pocas líneas y sin definir ni multiplexores, ni decodificadores, ni biestables, …

Implementad un diseño en vhdl para este Banco de registros. Completad el contenido del fichero **regfile.vhd**, que ya contiene la cabecera, con vuestra implementación.

Primera versión de la ALU básica

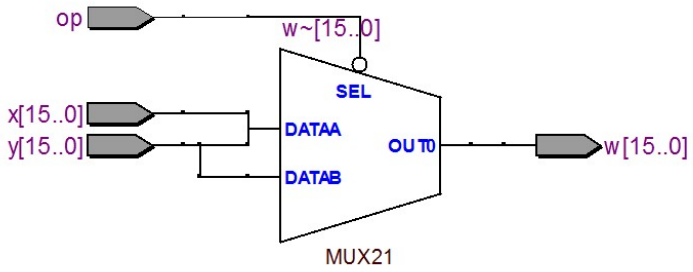
La primera versión de la ALU del procesador será muy simple, únicamente constará de dos operaciones:

- MOVI: Esta instrucción guarda los 8 bits de menor peso de la codificación de la instrucción con extensión de signo en el registro destino. Será la unidad de control la que se encargará de la extensión de signo, por lo que la ALU sólo tiene que dar como salida de esta operación el valor inmediato recibido por la señal *y*. Esta instrucción es ejecutada por la ALU siempre que el código de operación *op* sea 0.

- MOVHI: Esta instrucción escribe en el banco de registros los 16 bits formados por los 8 bits de menor peso de la codificación de la instrucción a los que se les concatenan por la derecha los 8 bits de menor peso del registro destino. Así pues, cuando el código de operación *op* de la ALU sea 1, esta debe asignar a la salida la palabra de 16 bits formada por la concatenación de los 8 bits de menor peso de la señal *y* y los 8 bits de menor peso de la señal *x*.

Las señales de entrada de la ALU están formadas por la señal *op* de 1 bit, que selecciona entre la operación MOVI (*op*=0) y la operación MOVHI (*op*=1). Las entradas *x* e *y* de 16 bits son los dos operandos sobre los que se aplican las operaciones. Por último, la señal *w* de 16 bits es el resultado de la operación que realiza la ALU.

Un esquema de bloques de la ALU se muestra a continuación:



La cabecera del diseño de la ALU en VHL sería la siguiente:

```
ENTITY alu IS
  PORT (x     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        y     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        op    : IN  STD_LOGIC;
        w     : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END alu;

ARCHITECTURE Structure OF alu IS
BEGIN
        .
        .
        .
END Structure;
```

Implementad un diseño en vhdl para esta ALU. Completad el contenido del fichero **alu.vhd**, que ya contiene la cabecera, con vuestra implementación.

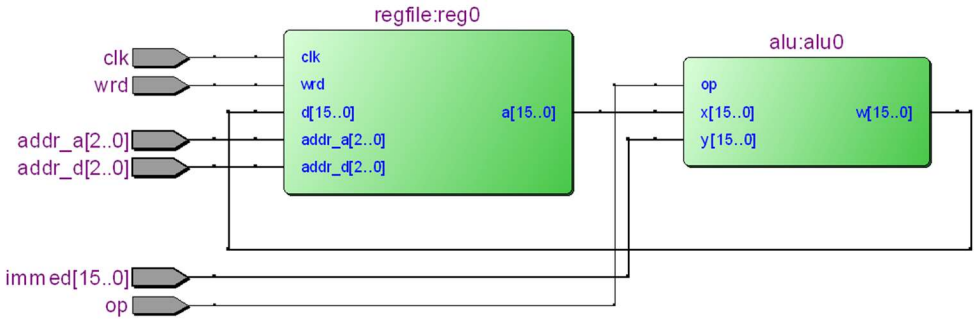
Primera versión del Datapath

El bloque **datapath** es el que hace el trabajo en el procesador. Por ahora, sólo escribe en el Banco de registros y mueve los bits correspondientes (dependiendo de si se trata del MOVI o MOVHI) en la ALU. El funcionamiento es bastante elemental, ya que en esta etapa del procesador sólo tiene que almacenar datos en registros.

De momento el datapath de nuestro procesador es muy simple. Únicamente contiene los dos bloques que hemos creado anteriormente; la ALU y el Banco de registros. Este bloque no realiza ninguna función en si mismo. Simplemente instancia y une los dos bloques creados mediante cables. Las entradas al datapath son:

- *op*: El código de operación de la ALU.
- *immed*: El valor inmediato con extensión de signo proveniente de la unidad de control.
- *clk*: La señal de reloj.
- *addr_a*: La dirección del registro de lectura del banco de registros.
- *addr_d*: La dirección del registro de escritura del banco de registros.
- *wrd*: El permiso de escritura en el banco de registros.

Estas entradas alimentan a los bloques que hemos creado anteriormente según se muestra en el siguiente esquema de bloques. A la instancia del Banco de registros la hemos llamado reg0 y a la de la ALU la hemos llamado alu0.



La cabecera de la entidad en VHDL es como sigue:

```
ENTITY datapath IS
  PORT (clk         : IN  STD_LOGIC;
        op          : IN  STD_LOGIC;
        wrd         : IN  STD_LOGIC;
        addr_a      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        addr_d      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        immed       : IN  STD_LOGIC_VECTOR(15 DOWNTO 0));
END datapath;

ARCHITECTURE Structure OF datapath IS
BEGIN
        .
        .
        .
END Structure;
```

Implementad un diseño en vhdl para este datapath. Completad el contenido del fichero **datapath.vhd**, que ya contiene la cabecera, con vuestra implementación.

Primera versión de la Unidad de Control

La unidad de control, a pesar de ser muy simple, es la parte más compleja de esta primera versión del procesador. Dentro de la unidad de control se regula el contador de programa (PC) y se decodifican las instrucciones que llegan de la memoria para generar las señales de control del datapath. Para mantener un diseño modular y ordenado, separaremos la parte de la unidad de control encargada del contador de programa de la encargada de la decodificación de instrucciones (lógica de control).

Así pues, en primer lugar nos ocuparemos de la lógica de control y para ello crearemos una entidad llamada **control_l**. Esta entidad tiene como única entrada la instrucción proveniente de la memoria de instrucciones. Como salidas genera las señales de control del datapath:

- *op*: Indica la operación que va a realizar la ALU.
- *ldpc*: Esta señal a 1 indica que se puede incrementar el program counter (PC). Se mantendrá a 1 hasta que se ejecute una instrucción del tipo HALT, momento en el que permanecerá a 0 para detener el procesador.
- *wrd*: Permiso de escritura en el banco de registros.
- *addr_a*: Dirección del registro fuente.
- *addr_d*: dirección del registro destino.
- *immed*: Valor inmediato con extensión de signo extraído de la instrucción.

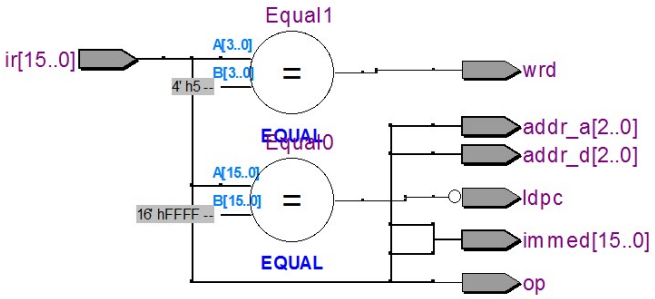
Para poder decodificar correctamente las instrucciones es necesario saber cómo se codifican. La codificación de todas las instrucciones del procesador la podemos encontrar en una tabla resumen al principio de este documento. La siguiente

tabla muestra la codificación de las instrucciones MOVI y MOVHI, la instrucción HALT se codifica con todos los bits del formato de instrucción a 1.

MOVI Rd, 0xN8	0	1	0	1	d	d	d	0	n	n	n	n	n	n	n	n
MOVHI Rd, 0xN8	0	1	0	1	d a	d a	d a	1	n	n	n	n	n	n	n	n

Según la implementación del ensamblador, de dirección del registro de escritura se encuentra en los bits 11 a 9 de la instrucción. La dirección del registro de lectura sólo la necesitaremos en el caso de la instrucción MOVHI, ya que debe leer la parte baja del registro de escritura y mantener su contenido (sólo cambian los bits más altos), por lo tanto la dirección del registro de lectura es la misma que el de escritura. La función *op* de la ALU puede ser directamente el bit 8 de la instrucción, según la codificación del ensamblador ya que es el bit que distingue entre las instrucciones MOVI y MOVHI. El campo inmediato son los 8 bits de menor peso de la instrucción, extendiendo el signo a 16 bits. Finalmente, la señal HALT se activa cuando la instrucción son sólo unos (0xFFFF).

Un posible esquema del bloque **control_l** se muestra a continuación:



La cabecera de la entidad en VHDL es como sigue:

```
ENTITY control_l IS
  PORT (ir          : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        op          : OUT STD_LOGIC;
        ldpc        : OUT STD_LOGIC;
        wrd         : OUT STD_LOGIC;
        addr_a      : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        addr_d      : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        immed       : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END control_l;

ARCHITECTURE Structure OF control_l IS
BEGIN
        .
        .
        .
END Structure;
```

Implementad un diseño en vhdl para esta entidad. Completad el contenido del fichero **control_l.vhd**, que ya contiene la cabecera, con vuestra implementación.

Una vez diseñada la lógica de control ya podemos ocuparnos del *program counter*. Nuestro procesador ejecutará una instrucción por ciclo. Puesto que las instrucciones son de 2 bytes de longitud y el acceso a memoria es a nivel de byte, el program counter debe incrementarse en 2 unidades a cada ciclo de reloj a partir de que la señal *boot* valga 0 después de haber valido 1 y siempre que no se ejecute la instrucción HALT.

Cuando la señal de *boot* valga 1 es como si se estuviese haciendo un reset al procesador, por tanto debe cargarse un valor por defecto en el PC. Este valor será la posición de memoria donde se encuentra la primera instrucción que se ejecutará cuando ya no se esté reseteando. El valor inicial del PC será el 0xC000 (más adelante ya se verá el motivo de este valor y no otro).

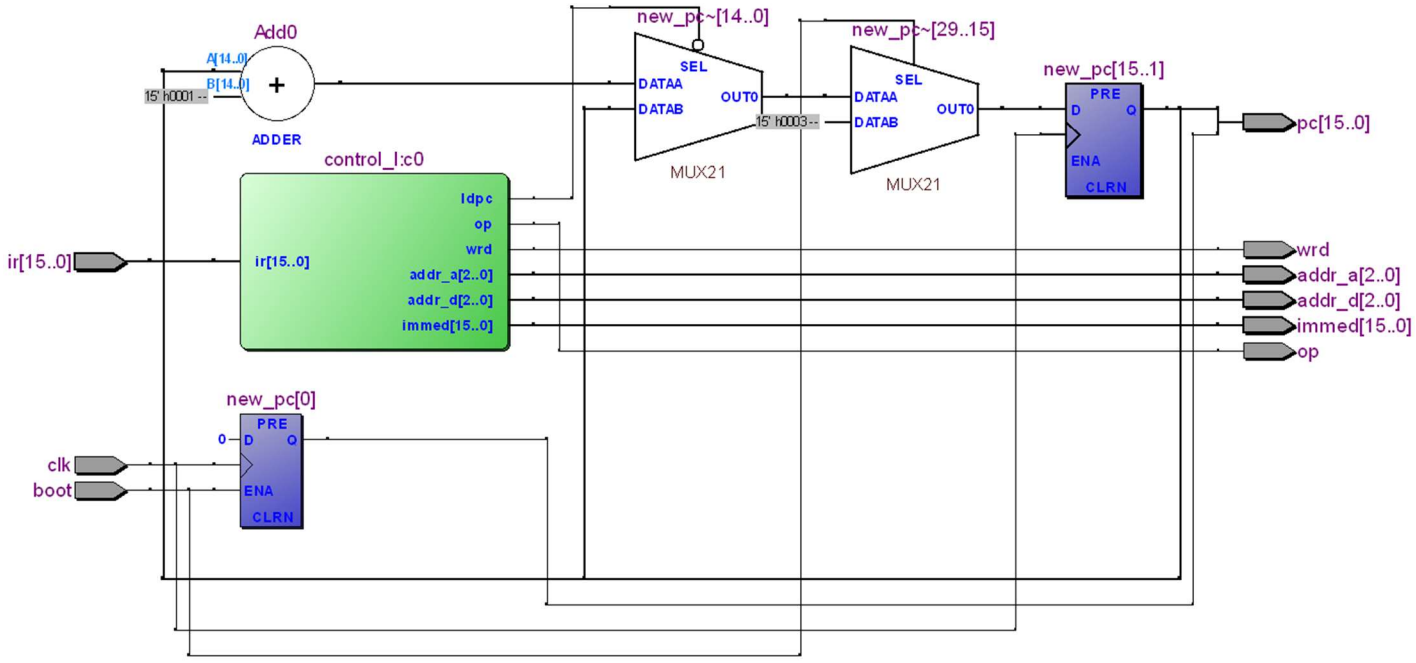
Así pues, las entradas de la **unidad de control** son:

- *ir*: La instrucción obtenida de la memoria de instrucciones.
- *clk*: La señal de reloj.
- *boot*: Señal de arranque del procesador. El program counter valdrá 0xC000 y no se incrementará mientras esta señal valga 1.

Las salidas de la **unidad de control** son:

- *pc*: El valor del contador de programa del procesador.
- *wrd*: El permiso de escritura en el banco de registros.
- *addr_a*: La dirección del registro fuente.
- *addr_d*: La dirección del registro destino.
- *immed*: el valor inmediato con extensión de signo extraído de la instrucción.
- *op*: Indica la operación que va a realizar la ALU.

El esquema de bloques de una posible implementación de la unidad de control se muestra en la siguiente figura:



La cabecera de la entidad en VHDL es la siguiente:

```
ENTITY unidad_control IS
  PORT (boot        : IN  STD_LOGIC;
        clk         : IN  STD_LOGIC;
        ir          : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        op          : OUT STD_LOGIC;
        wrd         : OUT STD_LOGIC;
        addr_a      : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        addr_d      : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        immed       : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        pc          : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END unidad_control;
```

```
ARCHITECTURE Structure OF unidad_control IS
BEGIN
        .
        .
        .
END Structure;
```

Implementad un diseño en vhdl para esta entidad. Completad el contenido del fichero **unidad_control.vhd**, que ya contiene la cabecera, con vuestra implementación.

Primera versión del procesador

Ahora ya hemos hecho todos los componentes del procesador base. Sólo tenemos que unir los módulos y conectarlo todo para formar el procesador. Este paso es muy sencillo y consiste simplemente en interconectar adecuadamente la unidad de control con el datapath y con las entradas y salidas del procesador.

Primero, el módulo de ejecución (el datapath) se conecta cable a cable con el módulo de control. El módulo de control, sin embargo, necesita de dos buses adicionales: el de direcciones para enviar el PC a memoria y el de datos para recibir la palabra hacia el IR. Estos dos buses los sacaremos del procesador hacia fuera.

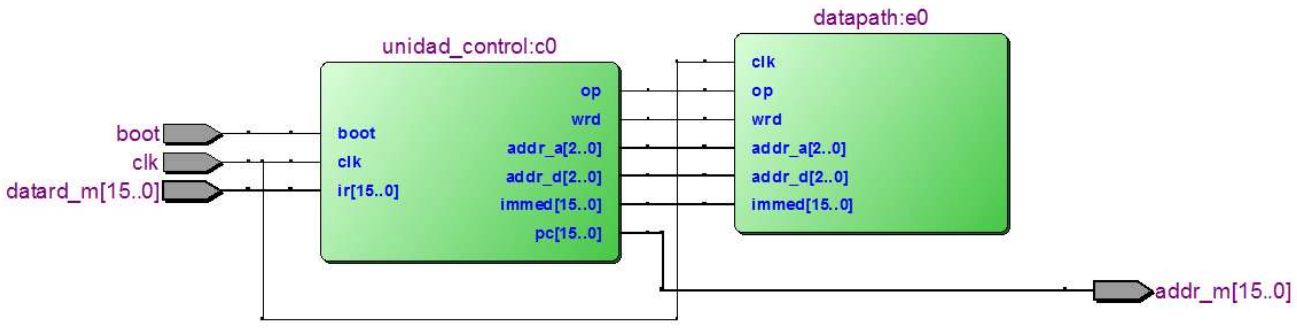
A este bloque le llamaremos **proc**. Este es el bloque más importante que es el de más alto nivel e instancia a todos los demás. Sólo tiene tres entradas: la entrada del reloj, la señal de *boot* (el reset) para cuando el procesador se encienda y la entrada de datos de memoria, y sólo una salida: el bus de direcciones.

Las entradas al procesador son:

- *boot*: La señal que indica el arranque del procesador.
- *clk*: La señal de reloj.
- *datard_m*: Los datos provenientes de la memoria.

El procesador cuenta solo con una salida, *addr_m*, que es la dirección a la memoria para cargar datos (de momento solo instrucciones).

La estructura de bloques del procesador es la siguiente:



La cabecera de la entidad en VHDL es la siguiente:

```
ENTITY proc IS
  PORT (boot        : IN  STD_LOGIC;
        clk         : IN  STD_LOGIC;
        datard_m    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        addr_m      : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END proc;
```

```
ARCHITECTURE Structure OF proc IS
BEGIN
        .
        .
        .
END Structure;
```

Y con esto y un bizcocho hasta mañana a las ocho. Ya hemos creado nuestro primer procesador.

Simulación y prueba del procesador

Una vez terminado el diseño del procesador es conveniente realizar una simulación para comprobar que funciona adecuadamente. Para probar este diseño, utilizaremos el programa de simulación *Modelsim* de la empresa *Mentor Graphics* que es uno de los más utilizados en la industria de diseño de circuitos digitales. Este programa permite hacer simulaciones complejas de sistemas descritos tanto en Verilog como en VHDL.

Para hacer la simulación, primero debemos desarrollar un módulo de test que pruebe el circuito de más alto nivel, en este caso **proc**. Este módulo debe crear la señal de reloj y establecer la señal de reset correctamente para que el procesador empiece a funcionar. Para crear la señal de reloj, como no sabemos aún la tecnología que usaremos ni el tiempo de ciclo del circuito final, nos da igual utilizar una frecuencia u otra. Por escoger un valor, trabajaremos con 50MHz (tiempo de semi-ciclo 10ns).

A este módulo de test lo hemos llamado **test_sisa** y lo encontraréis en el directorio "Test-Memoria-hexadecimal" junto al fichero **memory.vhd** que contiene la implementación de la memoria y una función para inicializarla con un fichero. Esta función carga el contenido de un fichero (*contingut.memoria.hexa.rom*) a partir de la dirección 0xC000, que es donde se supone que comienza el área de sólo lectura del sistema. El contador de programa (PC) comienza a ejecutar código en esa dirección. Más adelante veremos el formato de este fichero y cómo funciona la simulación.

Así que si deseamos ejecutar un programa en nuestro procesador, lo primero que deberemos es traducir las instrucciones en lenguaje ensamblador al formato binario y luego pasarlas a hexadecimal. Supongamos que deseamos ejecutar el programa formado por las siguientes instrucciones en lenguaje SISA.

```
Ensamblador         Binario                 Hexadecimal
movi  r5, 53        0101 1010 0011 0101     5a35
movhi r5, 68        0101 1011 0100 0100     5b44
movi  r0, -1        0101 0000 1111 1111     50ff
movhi r0, 16        0101 0001 0001 0000     5110
halt                1111 1111 1111 1111     ffff
movi  r1, 35        0101 0010 0010 0011     5223
```

Una vez tengamos las instrucciones traducidas al hexadecimal, podemos crear el fichero que será cargado en la memoria del procesador. El fichero *contingut.memoria.hexa.rom* es un fichero de texto que contiene los valores hexadecimales que forman el programa a ejecutar. Cada línea del fichero corresponde a una posición de memoria (a partir de la 0xC000). Por tanto contendrá dos caracteres hexadecimales (8 bits) que forman la instrucción en formato little-endian. En este caso de ejemplo el fichero contendría lo siguiente:

```
35
5A
44
5B
FF
50
10
51
FF
```

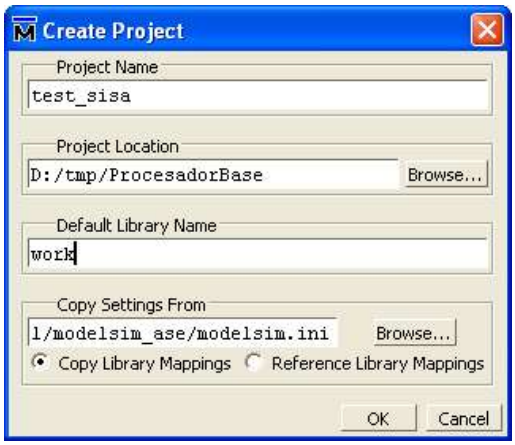
```
FF
23
52
```

Podéis editar este archivo para realizar vuestras propias pruebas.

Si no deseáis traducir a mano cada una de las instrucciones, en el Anexo A podéis ver como usar una herramienta de compilación automática disponible únicamente para Linux.

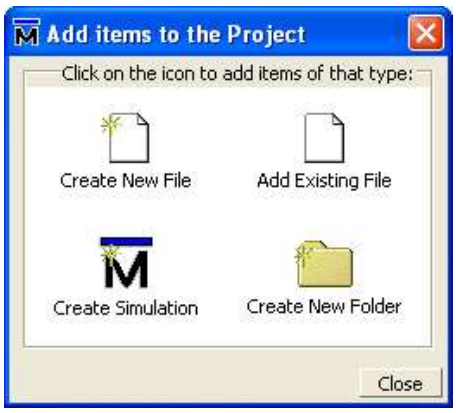
Antes de empezar a simular nuestro procesador deberemos copiar los ficheros **test_sisa.vhd**, **memory.vhd** y **contingut.memoria.hexa.rom** del directorio "Test-Memoria-hexadecimal" al directorio principal de nuestro proyecto. Si no lo hacemos tendremos un error de ejecución más adelante al iniciar la simulación.

Ejecutaremos el *ModelSim* y cerraremos la ventana de bienvenida si aparece. En el menú superior, pulsaremos "**File→New→Project…"**. En la ventana que se abre, escribiremos como nombre de proyecto "**test_sisa**" y en "*Project Location*" seleccionaremos la ruta donde está nuestro proyecto, y luego pulsaremos "**OK**".



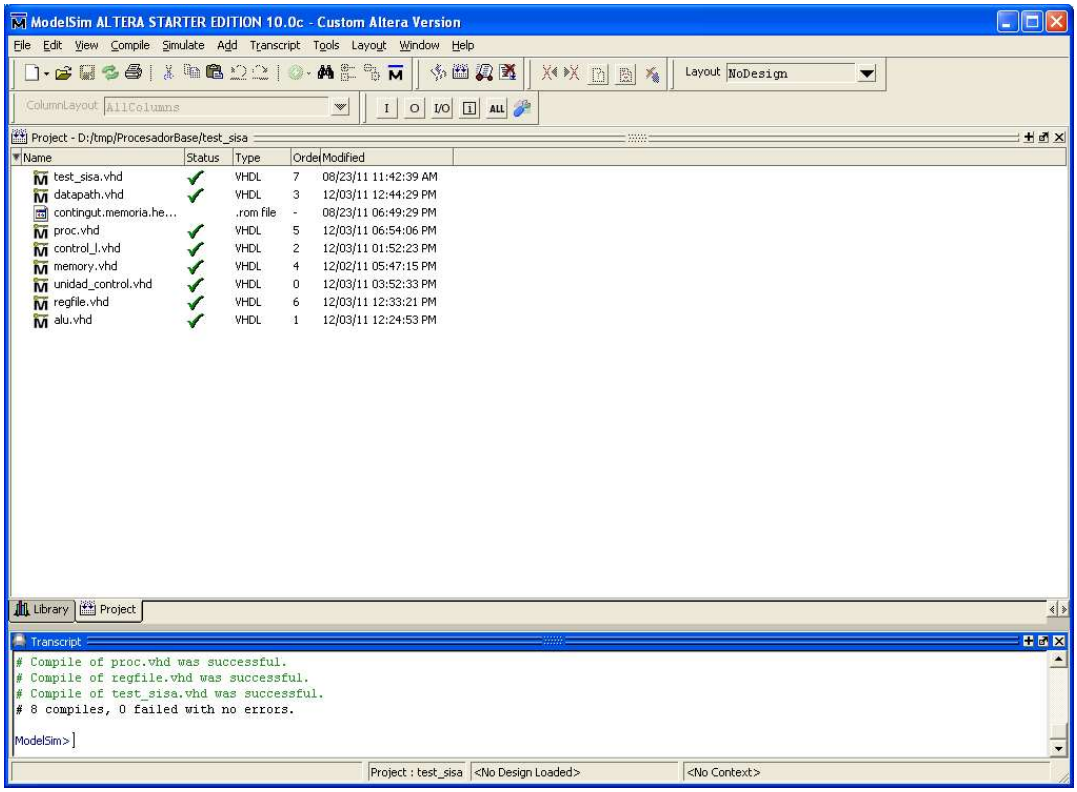
En el *ModelSim*, se abrirá una pequeña ventana de título "*Add items to the Project*" y en ella seleccionaremos "*Add Existing File*". A continuación pulsaremos "**Browse…**" y buscaremos los siguientes ficheros (podemos seleccionar más de un fichero a la vez pero si no se encuentran todos en la misma ruta podemos añadir sólo algunos y luego volver a pulsar "*Add Existing File*" para añadir los demás):

- *test_sisa.vhd*
- *memory.vhd*
- Todos los ficheros VHDL que creamos en el diseño del procesador (*alu.vhd, regfile.vhd, datapath.vhd, control_l.vhd, unidad_control.vhd* y *proc.vhd*)

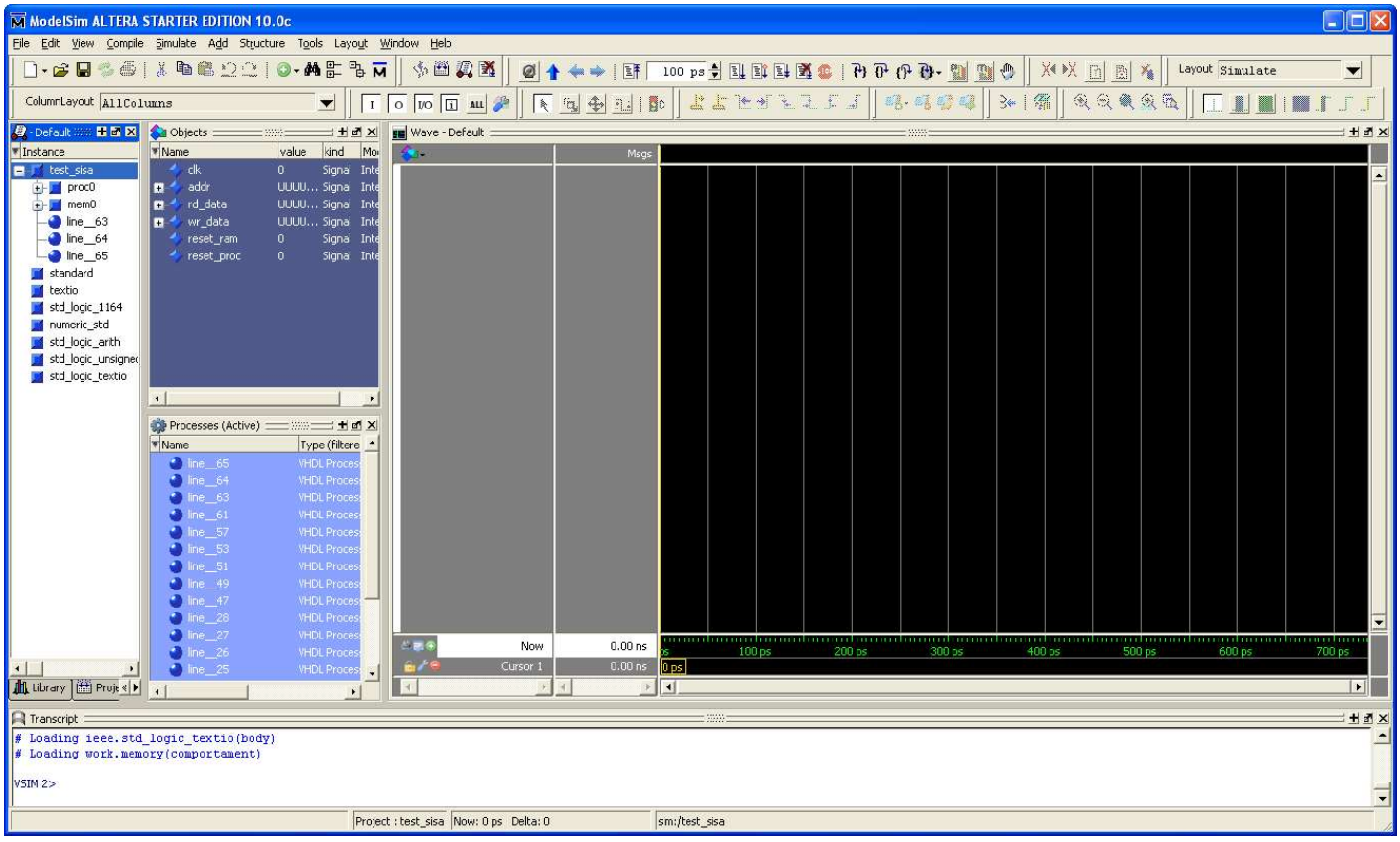


Una vez añadidos todos pulsa el botón "**Close**" para salir de la ventana. Ahora podemos ver como se han añadido todos los ficheros proyecto de simulación.

En el menú principal pulsaremos **Compile→Compile All**. Si todo va bien veremos como se compilan los 8 ficheros sin ningún error.



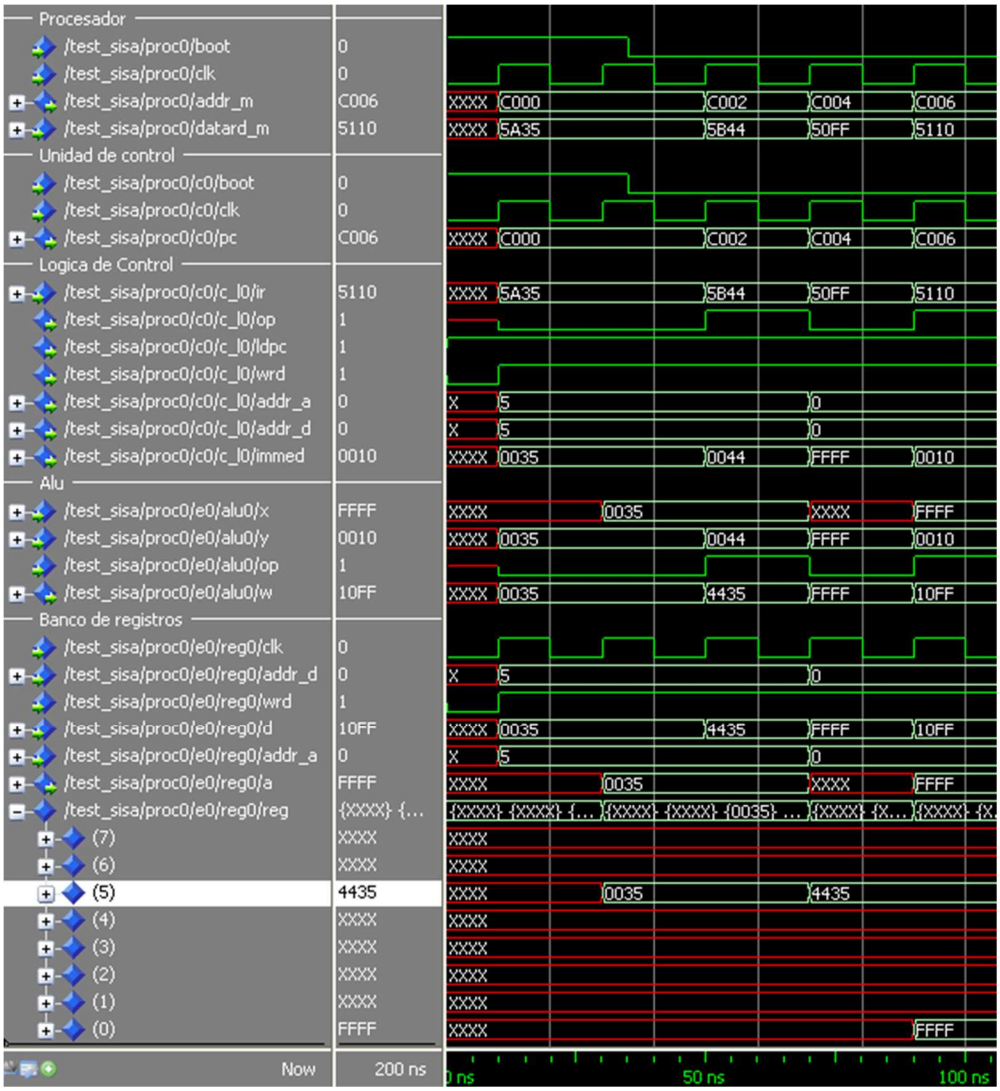
A continuación pulsaremos "**Simulate->Start Simulation…",** En la ventana que se abre buscaremos la librería de nombre "**work**" (seguramente la primera en la lista) y clicaremos sobre el símbolo "+" para desplegarla. Seleccionaremos **test_sisa** y pulsaremos "**Ok**". Ahora se nos abrirá una ventana similar a la siguiente:



Observad como en el recuadro de la izquierda aparece la jerarquía del diseño con **test_sisa** a la cabeza. Podemos desplegar los módulos del procesador pulsando sobre los símbolos "+". De aquí elegiremos los módulos cuyas señales queremos comprobar pulsando botón derecho sobre el módulo y seleccionando en el menú emergente la opción "**Add→To Wave→All items in region**". Repetiremos el proceso para los módulos **proc**, **alu** y **reg** (notad que los nombres que aquí aparecen no son los de las entidades, sino los de las instancias que hayáis elegido vosotros). Una vez colocadas las señales que nos interesan, escribiremos "**run 100ns**" en la consola para indicar que queremos simular 100 nano segundos. Como la señal de reloj que hemos escogido es de 50 MHz (el tiempo de ciclo es de 20ns) nos dará para aproximadamente ver la inicialización del procesador y la ejecución de las 3 primeras instrucciones.

Una vez hecho esto, habrán aparecido varias señales en la pantalla con valores en binario. Para poder verlas con más comodidad, vamos a seleccionar representación en hexadecimal de todos los valores. Seleccionaremos todas las señales que se han añadido (clicaremos en la primera y manteniendo la tecla *shift* pulsada clicaremos en la última) y pulsaremos el botón derecho del ratón, en el menú emergente seleccionaremos la opción "**Radix→Hexadecimal**". Ahora vamos a alejar la vista, ya que el zoom por defecto es demasiado grande (está en una escala de ps). Pulsaremos el icono de lupa con un "-" en el interior que se encuentra en la parte superior de la ventana hasta que toda la simulación sea visible.

Una vez hecho esto ya podemos comprobar fácilmente si se encuentran los valores esperados en cada componente. Éstos dependerán de las instrucciones codificadas en el fichero *contingut.memoria.hexa.rom*. Para las instrucciones que hemos usado de ejemplo los valores deberían ser:



Si nuestro diseño funciona correctamente, podremos observar que si *boot* vale 1, cuando llega el primer flanco ascendente de reloj se inicializa el procesador. Mostrando por el bus *addr_m*, que indica la dirección de memoria a acceder, el valor 0xC000 que es la dirección de memoria donde se encuentra la primera instrucción a ejecutar. Por el bus de memoria

datard_m se empieza a recibir la instrucción codificada que hay en esa posición de memoria. Justo en ese momento comienza la ejecución de la instrucción MOVI R5, 53. Mientras *boot* continúe valiendo 1 se reejecutará esta instrucción indefinidamente. Esto garantiza que el tiempo mínimo que se ejecutará esta instrucción es de un ciclo completo. Cuando la señal de *boot* deja de valer 1, el valor que sale por el bus *addr_m* se incrementa en 2 unidades a cada ciclo.

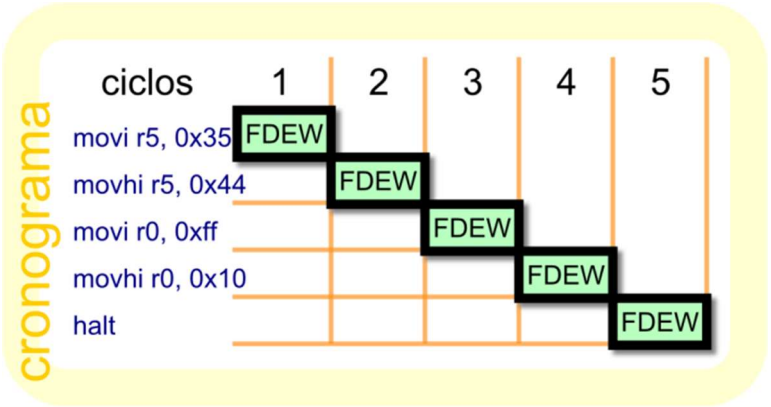
Cuando llega el primer flanco ascendente de reloj con la señal de *boot* a 0 empieza a ejecutarse la instrucción siguiente (en la dirección 0xC002), el valor 0x35 se escribe en el registro R5 como se visualiza en la parte inferior de la imagen. Los valores de las señales antes de la inicialización del procesador dependen de como se haya hecho la inicialización y el sistema de *boot* y pueden no parecerse a los que aparecen en la figura.

Si continuamos la ejecución hasta el final, podemos ver como se escribe el 0x44 en la parte alta del registro R5, después se extiende el signo de 0xFF y se guarda en el registro R0 y finalmente se modifica la parte alta del registro R0 para escribir el valor 0x10. El registro R5 queda con el número 0x4435 y el R0 con el valor 0x10FF como debe ser.

Si deseamos continuar con la simulación podemos escribir en la consola el comando "**run 60ns**" (tres ciclos más) y deberíamos ver que después de ejecutar la instrucción HALT, el procesador ya no incrementa el PC y no ejecuta la instrucción que hay después del HALT en el fichero de prueba. Por tanto, el diseño pasa correctamente el juego de pruebas.

En la siguiente figura podemos ver el cronograma de simulación simplificado donde se observa que las instrucciones se ejecutan en 1 ciclo por instrucción. El ciclo FDEW corresponde a las etapas de *Fetch*, *Decode*, *Execute* y *Writeback*.

Al final del ciclo, se escribe el resultado de la ejecución en el registro destino, justo cuando llega el flanco ascendente. Por lo tanto, tenemos un procesador de un solo ciclo no segmentado.



Modelsim

El programa de simulación *Modelsim* dispone de una consola en la que se pueden introducir comandos a través del teclado para realizar acciones. Es más, todas las opciones a través de los menús no son más que una interfaz gráfica que finalmente genera un comando en la consola. Desde la consola se pueden realizar fácilmente acciones útiles que desde la interfaz gráfica son costosas o imposibles. Por ejemplo:

- El comando "*radix*" especifica la base por defecto que será usada en todas las representaciones de los valores. Así que si escribimos el comando "*radix -hexadecimal*" veremos el valor de todas las señales de los cronogramas en hexadecimal.
- El comando "*restart -f*" vuelve a cargar los elementos de diseño y restablece el tiempo de simulación a cero. Actuando como el reinicio de una simulación.

Así que se recomienda echar una ojeada al listado de comandos del manual de referencia de esta herramienta (*ModelSim® Reference Manual*).

Anexo A: Herramienta ensamblador

Disponemos de un conjunto de herramientas para compilar, debuggar y emular programas escritos para el lenguaje SISA. Estas herramientas están desarrolladas para Linux (para distribuciones de 32 y 64 bits). Las herramientas y su documentación os las podéis bajar de la página web de la asignatura.

Veamos como sería el fichero en ensamblador SISA correspondiente al ejemplo propuesto en la documentación. Editamos un fichero de texto, al que llamaremos "movi.s" e introducimos lo siguiente:

```
.text
  movi r5, 53
  movhi r5, 68
  movi r0, -1
  movhi r0, 16
  halt
  movi r1, 35
```

Para compilar este fichero debemos ejecutar el siguiente comando.

```
$ sisa-as movi.s -o movi.o
```

Ahora si desensamblamos el fichero resultante obtendremos la codificación de las instrucciones tal y como las tenemos que poner en la memoria. Para desensamblar el código ejecutamos el siguiente comando:

```
$ sisa-objdump -d movi.o
```

El resultado que obtendremos será parecido al siguiente:

```
movi.o: file format elf32-sisa

Disassembly of section .text

00000000 <.text>:
      0: 5a35 movi r5, 53
      2: 5b44 movhi r5, 68
      4: 50ff movi r0, -1
      6: 5110 movhi r0, 16
      8: ffff halt
      a: 5223 movi r1, 35
```

En la columna que hay antes del lenguaje ensamblador aparecen las instrucciones ensambladas y codificadas en hexadecimal.