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 2: El subsistema de memoria

Una vez que ya disponemos de una implementación del procesador base que es capaz de ejecutar las instrucciones MOVI, MOVHI y HALT, vamos a añadirle las instrucciones de acceso a memoria. Para ello, usaremos las memorias externas a la FPGA que dispone la placa de desarrollo. En esta etapa el procesador deberá funcionar en una implementación física de FPGA. Esta fase la dividiremos en dos subetapas:

1. Se añadirá toda la lógica de lectura y escritura de memoria en el procesador base, pero aún se continuará con el *stub* de memoria de la fase anterior para poder hacer una simulación y verificar correctamente el diseño. Este diseño se simulará con el *Modelsim* de la misma manera que en el capítulo anterior.
2. Finalmente, con el procesador diseñado y probado, se sustituirá el *stub* de memoria usado para simular el comportamiento por un controlador de memoria que use los chips de memoria de la placa de desarrollo. Una vez adaptado a los componentes de la placa de desarrollo de la FPGA se probará el diseño físicamente.

Subetapa 2.1: Lógica de memoria

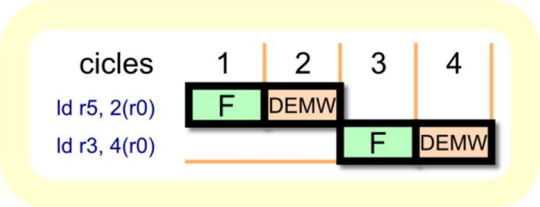
Antes teníamos el módulo de ejecución (el *datapath*) sin lógica. Ahora al añadir hardware en el procesador para hacer lecturas y escrituras a memoria, el *datapath* deberá realizar algunas acciones.

Las instrucciones de acceso a memoria que implementaremos serán LD, ST, LDB y STB. Aunque el procesador SISA dispone de instrucciones de coma flotante, no las implementaremos. El formato, comportamiento y codificación exacta de estas instrucciones podéis encontrarlo en el documento adjunto SISA-F.

Veamos cuales serían las principales modificaciones que hay que hacer al procesador para soportar estas nuevas instrucciones.

Procesador multiciclo

En primer lugar, como la memoria de instrucciones y de datos es la misma, tenemos un riesgo estructural en las instrucciones tipo *load* y *store*. Lo resolveremos de la manera más sencilla: haremos el procesador multiciclo. En el primer ciclo, haremos el *fetch* (F), mientras que en el segundo ciclo hacemos el *decode*, el acceso a memoria y la ejecución de la instrucción (DEMW). En el siguiente ciclo se escribe en el registro mientras se hace el *fetch* de la siguiente instrucción. Por lo tanto, tenemos un valor fijo de 2 ciclos por instrucción, como podemos ver en la siguiente figura.



El bus de direcciones de memoria, pues, se utilizará en el primer ciclo por el PC, mientras que en el segundo ciclo lo será para la dirección del acceso a memoria (en el caso de que sea un *load* o un *store*).

Además, ahora tendremos que almacenar la instrucción que viene de memoria en un registro nuevo (**IR**) para que las señales de control puedan ser válidas también en el ciclo de ejecución.

Cálculo del inmediato de la ALU

Por otra parte, según la implementación de las instrucciones de acceso a memoria, los inmediatos codificados en las instrucciones LD y ST deben multiplicarse por 2 en el caso de los accesos a nivel de *word* antes de que la ALU pueda sumar la dirección base del registro más el inmediato.

Registro destino

También debemos contar que en el caso del *load*, lo que se debe escribir en el registro destino no es el resultado de la ALU sino lo que viene de memoria.

En este caso, el módulo *datapath* debe encargarse de seleccionar el valor que enviaremos al puerto de escritura del Banco de registros. Debe escoger entre el valor procedente de la ALU o el procedente de la memoria.

ALU

También deberá seleccionar el origen del valor que pondrá en el bus de direcciones de la memoria. El cálculo de la dirección efectiva de las instrucciones de acceso a memoria se hace en la ALU. Si está en el ciclo de *fetch* deberá escoger el valor del PC y si no el valor procedente de la ALU. Por ello, necesitará una señal que le indique si estamos en el ciclo F (0) o en el ciclo DEMW (1).

La ALU que habíamos descrito en la etapa anterior del procesador base no era capaz ni siquiera de sumar (ya que tampoco lo necesitaba). Ahora, las instrucciones de memoria deben calcular la dirección efectiva, así que tenemos que añadir al menos un sumador.

Banco de registros

Necesitamos en el Banco de registros otro puerto de lectura para las instrucciones de tipo *store*, ya que han de leer simultáneamente la dirección base y también el contenido del registro que se escribirá en memoria. Por ello, añadiremos un puerto lectura que denominaremos *B*. En este nuevo Banco de registros tendremos una entrada de selección de registros que llamaremos *addr_b*. Este es el registro que se leerá y el bus *B* será el bus de salida con el contenido del registro.

Módulo de control

Ya tenemos descritas todas las modificaciones necesarias de la parte de ejecución para que las instrucciones de acceso a memoria funcionen correctamente. Ahora necesitamos modificar la lógica del módulo de control para que las señales tengan algún valor. Ahora ya no cargaremos el PC siempre sino sólo en el ciclo F. Además, algunas de las señales de

control sólo deberán activarse en uno de los dos ciclos. Por tanto necesitaremos saber en todo momento en que ciclo estamos.

Las modificaciones de la implementación de los módulos

Modificaciones en el Banco de registros.

La única modificación que hay que hacer en el Banco de registros es añadir un puerto de lectura adicional. Los 8 registros también estarán direccionados mediante la señal *addr_b*, de 3 bits, para seleccionar el registro fuente que debe salir por el puerto *B*. Ahora el Banco de registros del procesador ya estará totalmente completo para esta etapa.

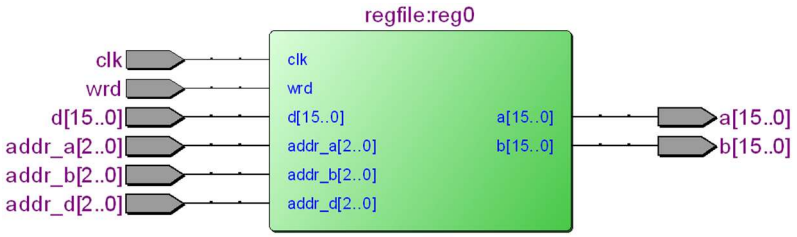
Así pues, las entradas al Banco de registros son:

- *clk*: La señal de reloj.
- *wrd*: El permiso de escritura en el Banco de registros.
- *d*: El bus de datos con el valor que vamos a almacenar en el Banco de registros.
- *addr_d*: La dirección del registro de escritura del Banco de registros.
- *addr_a*: La dirección del registro de lectura del Banco de registros por el puerto *A*.
- *addr_b*: La dirección del registro de lectura del Banco de registros por el puerto *B*.

Las salidas del Banco de registro son:

- *a*: El bus de datos con el valor del registro seleccionado por la señal *addr_a*.
- b: El bus de datos con el valor del registro seleccionado por la señal *addr_b*.

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_b  : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        addr_d  : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        a       : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        b       : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END regfile;

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

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.

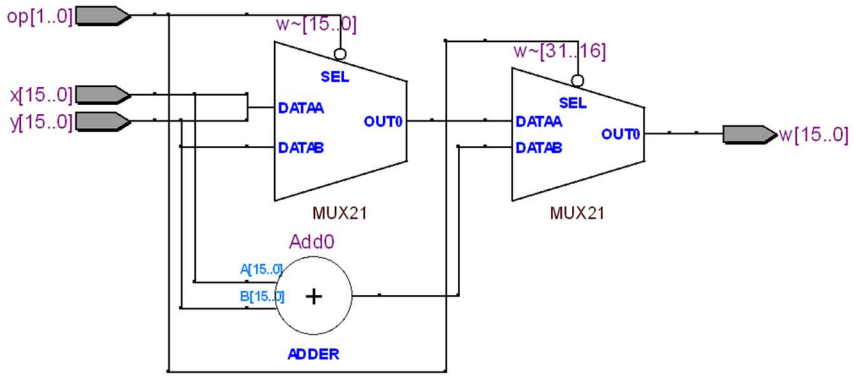
Modificaciones en la ALU.

Como el cálculo de la dirección efectiva para acceder a la memoria mediante las instrucciones de acceso a memoria se hace mediante una suma, la ALU debe ser capar de hacer sumas entre sus dos entradas.

Las entradas de la ALU siguen siendo las señales *x* e *y*, y la salida sigue siendo la señal *w*. Los dos operandos sobre los que se aplican las operaciones y el resultado son de 16 bits.

Como la ALU debe realizar tres operaciones distintas, ahora la señal de control *op* estará formada por dos bits. Así que cuando la señal *op* valga 00 se efectuará la operación MOVI, cuando *op* valga 01 se efectuará la operación MOVHI y finalmente cuando *op* valga 10 hará la suma entre las dos entradas. La suma no tendrá en cuenta ningún *overflow*.

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



La cabecera del diseño de la ALU en VHDL es como sigue:

```
ENTITY alu IS
  PORT (x  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        y  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        op : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
        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.

Modificaciones en el datapath

El bloque **datapath** es el que hace el trabajo en el procesador. Antes esté bloque no realizaba ninguna función en si mismo. Simplemente instanciaba y unía el Banco de registros y la ALU mediante cables. Ahora debe realizar algunas tareas.

Una de las tareas es seleccionar la procedencia del dato que vamos a escribir en el Banco de registros. Ahora el dato puede proceder de la ALU (en caso de las instrucciones de movimiento) o de la memoria (en el caso de los *loads*). Para esta selección usaremos una señal que hemos llamado *in_d*.

Otra tarea que tiene que realizar es determinar como será el inmediato que entrará por la entrada *y* de la ALU. Cuando se trate de una instrucción de movimiento el inmediato será el mismo que recibiremos de la unidad de control, pero no

siempre cuando se trate de una instrucción de acceso a memoria. Según la implementación de las instrucciones de acceso a memoria, los inmediatos codificados en las instrucciones LD y ST deben multiplicarse previamente por 2 en el caso de los accesos a nivel de *word*. Para determinar si hay que multiplicar por 2 o no el inmediato usaremos una señal que la hemos llamado *immed_x2* y que la generará la unidad de control.

Por último, debe decidir qué valor pone en el bus de direcciones de la memoria, en función del ciclo este bus deberá contener el PC o la dirección efectiva de las instrucciones de acceso a memoria. Para ello usaremos una señal que hemos llamado *ins_dad*.

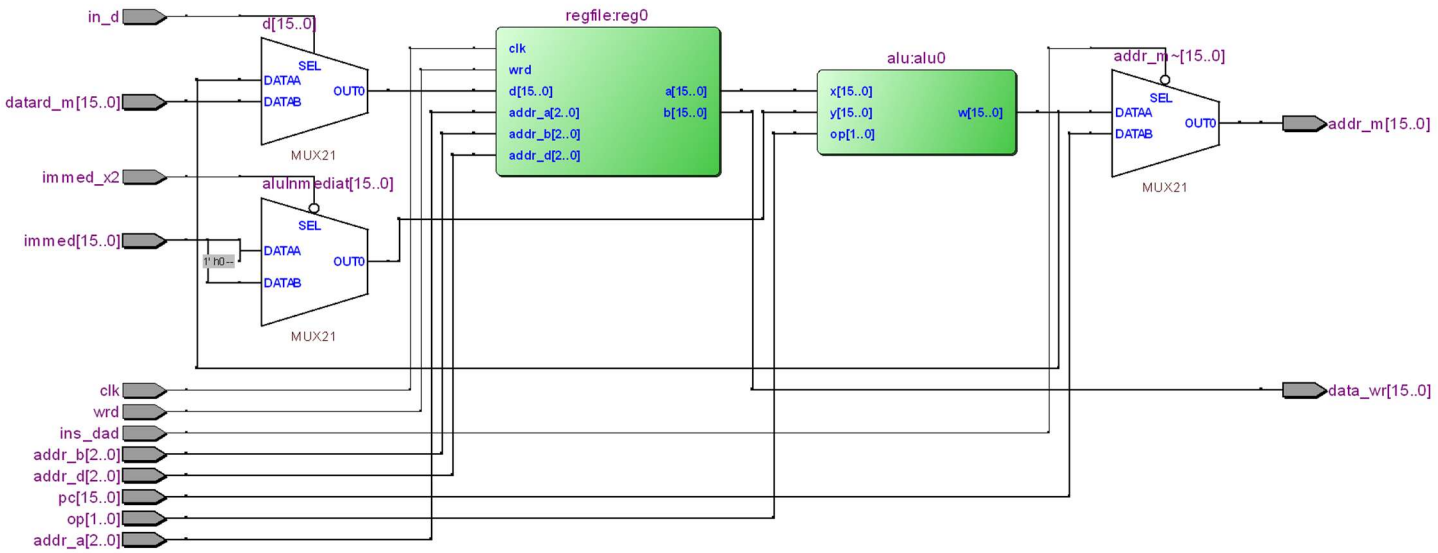
Las entradas al **datapath** son:

- *clk*: La señal de reloj.
- *op*: El código de operación de la ALU.
- *wrd*: El permiso de escritura en el Banco de registros.
- *addr_a*: La dirección del registro de lectura del Banco de registros por el puerto *A*.
- *addr_b*: La dirección del registro de lectura del Banco de registros por el puerto *B*.
- *addr_d*: La dirección del registro de escritura del Banco de registros.
- *immed*: El valor inmediato con extensión de signo proveniente de la unidad de control.
- *immed_x2*: la señal que determina si hay que desplazar el inmediato o no.
- *datard_m*: Los datos provenientes de la memoria en caso de un *load*.
- *ins_dad*: Esta señal a 1 indica que en el bus de direcciones de la memoria pondremos la salida de la ALU y si vale 0 pondremos el PC.
- *pc*: El Program Counter procedente de la unidad de control para poder enviarlo al bus de direcciones de la memoria.
- *in_d*: Esta señal a 1 indica que en el Banco de registros almacenaremos el valor procedente de la memoria y si vale 0 almacenaremos el valor procedente de la ALU.

Ahora el **datapath** tiene salidas. Las salidas son:

- *addr_m*: Es la dirección de memoria a la que se accederá en cada momento.
- *data_wr*: Es el valor que se envía a la memoria en el caso de hacer un *store*. Que en este caso es el valor que leemos por el puerto *B* del Banco de registros.

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 como sigue:

```
ENTITY datapath IS
  PORT (clk         : IN  STD_LOGIC;
        op          : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
        wrd         : IN  STD_LOGIC;
        addr_a      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        addr_b      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        addr_d      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        immed       : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        immed_x2    : IN  STD_LOGIC;
        datard_m    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        ins_dad     : IN  STD_LOGIC;
        pc          : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        in_d        : IN  STD_LOGIC;
        addr_m      : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        data_wr     : OUT 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.

Nueva Unidad de Control

Ahora la unidad de control debe encargarse de varias tareas. Debe gestionar correctamente el registro PC, el registro IR y la lógica del multiciclo.

El funcionamiento del PC es como en la versión anterior, con la salvedad que ahora sólo cargaremos el PC cuando empieza el ciclo F dado que el procesador actual es multiciclo. Modificaremos la señal *ldpc* que genera el módulo *control_l* para que nos indique cuando debemos cargar el nuevo PC.

También deberemos tener un registro IR que guarde el valor procedente de la memoria con la instrucción. Es necesario guardarlo porque algunos de sus valores serán necesarios en ambos ciclos y puede que la instrucción ya no esté en el bus de datos de la memoria cuando la necesitemos.

El funcionamiento del IR es muy parecido al del PC. 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 IR. Este valor será una instrucción equivalente a una instrucción NOP (*No OPeration*), de la cual no dispone el repertorio de instrucciones SISA. Nosotros lo inicializaremos con el valor 0x0000 que corresponde la instrucción AND R0, R0, R0 que de momento no está implementada. Esta instrucción, una vez ejecutada, no modifica el estado del computador. Cuando el procesador esté en su funcionamiento normal (*boot*=0), al final del ciclo de *fetch* el IR debe almacenar el valor que viene de la memoria. Para determinar si hay que memorizar un valor o no en el IR vamos a usar una señal llamada *ldir* que la generará la lógica encargada del multiciclo.

Antes de ver como esta implementado la unidad de control vamos a ver la lógica encargada del multiciclo y los cambios en la lógica de control.

Lógica del multiciclo

Vamos a crear una nueva entidad que se encargue de la lógica relacionada con el multiciclo. Básicamente será un grafo de estados con dos estados: uno para el estado de *fetch* (F) y otro para el resto de la ejecución (DEMW). En cada estado generará los valores adecuados para cada señal que dependa del ciclo de ejecución.

La señales que dependen del ciclo en que se ejecutan son los permisos de escritura para el Banco de registros y la memoria. Por ejemplo, el permiso de escritura del Banco de registros sólo se activará en el ciclo DEMW para que cuando transicione al ciclo F de la siguiente instrucción, se almacene el valor en el registro (si se estaba ejecutando una instrucción que escribe en el Banco de registros). No podemos activar esta señal en el ciclo F porque si no, escribiríamos valores incorrectos en los registros. Recordad que los datos se escriben en el Banco de registros al final del ciclo de ejecución, en el flanco ascendente de reloj.

Por tanto, como utilizaremos en adelante señales que dependen del ciclo donde estamos, vamos a crear un módulo nuevo que llamaremos **multi.vhd**, que implementará esta lógica.

Como el valor de las señales las genera la lógica de control (**control_l**) utilizaremos esté grafo de estados para filtrar aquellas que sólo deban tomar su valor en un ciclo concreto.

Veamos primero las modificaciones a la lógica de control.

Modificaciones de la Lógica de Control

Veamos que modificaciones hay que hacer en la entidad llamada **control_l**. Esta entidad sigue teniendo como única entrada la instrucción proveniente de la memoria de instrucciones (*ir*). 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 del puerto *A*.
- *addr_b*: Dirección del registro fuente del puerto *B*.
- *addr_d*: Dirección del registro destino.
- *immed*: Valor inmediato con extensión de signo extraído de la instrucción.
- *wr_m*: Permiso de escritura en la memoria si es una instrucción ST o STB.
- *in_d*: Esta señal a 1 indica que en el Banco de registros almacenaremos el valor procedente de la memoria y si vale 0 almacenaremos el valor procedente de la ALU.
- *immed_x2*: La señal que determina si hay que desplazar el inmediato o no. Debe valer 1 si se ejecuta una instrucción de acceso a *word* como son las instrucciones LD y ST.
- *word_byte*: La señal indica si el acceso a memoria es a nivel de *byte* o *word*. Sólo debe valer 1 cuando se esta ejecutando una instrucción LDB o STB.

Para poder decodificar correctamente las instrucciones es necesario saber cómo se codifican. La siguiente tabla muestra la codificación de las instrucciones LD, ST, LDB y STB.

15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0		
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
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

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_VECTOR(1 DOWNTO 0);
        ldpc        : OUT STD_LOGIC;
        wrd         : OUT STD_LOGIC;
        addr_a      : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        addr_b      : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        addr_d      : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        immed       : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        wr_m        : OUT STD_LOGIC;
        in_d        : OUT STD_LOGIC;
        immed_x2    : OUT STD_LOGIC;
        word_byte   : OUT STD_LOGIC);
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.

Módulo multiciclo

Veamos como se comporta el módulo **multi.vhd**. Este módulo recibe las señales procedentes de la lógica de control (control_l) y filtra aquellas que son dependientes del ciclo de ejecución utilizando un grafo de estados (*ver anexo*) de dos ciclos: F (*fetch*) y DEMW (resto ejecución).

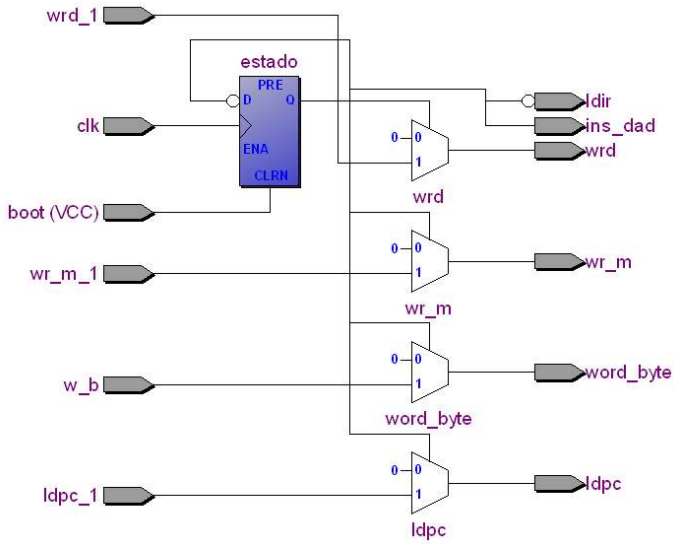
Así pues, las entradas de la **multi** son:

- *clk*: La señal de reloj.
- *boot*: Señal de arranque del procesador. En este caso hace que se ponga la máquina de estados en el estado inicial, el de *fetch*.
- *ldpc_l*: Señal *ldpc* generada por la lógica de control.
- *wrd_l*: Señal *wrd* generada por la lógica de control.
- *wr_m_l*: Señal *wr_m* generada por la lógica de control.
- *w_b*: Señal *word_byte* generada por la lógica de control.

Las salidas de la **multi** son:

- *ldpc*: Señal que, o bien vale el valor de *ldpc* generado por la lógica de control cuando se está en el ciclo de DEMW o 0 en otro caso (hasta que implementemos las instrucciones de salto).
- *wrd*: Señal que, o bien vale el valor de *wrd* generado por la lógica de control cuando se está en el ciclo de DEMW o 0 en otro caso.
- *wr_m*: Señal que, o bien vale el valor de *wr_m* generado por la lógica de control cuando se está en el ciclo de DEMW o 0 en otro caso.
- *word_byte*: Señal *word_byte* generada por la lógica de control y que sólo debe dejarse pasar en el ciclo de DEMW. En el ciclo F debe valer 0 ya que el acceso a la memoria, para traerse una instrucción, es a nivel de *word*.
- *ins_dad*: Esta señal a 1 le indicará al *datapath* que en el bus de direcciones de la memoria deberá poner la salida de la ALU y si vale 0 deberá poner el PC. Básicamente nos dice si estamos al ciclo F o DEMW
- *ldir*: Es la señal que indica que cargaremos un nuevo valor en el IR, sólo se activa en el ciclo F

El esquema de bloques de una posible implementación de la entidad **multi** se muestra en la siguiente figura:



La cabecera de la entidad en VHDL es la siguiente:

```
ENTITY multi IS
  PORT(
        clk          : IN  STD_LOGIC;
        boot         : IN  STD_LOGIC;
        ldpc_l       : IN  STD_LOGIC;
        wrd_l        : IN  STD_LOGIC;
        wr_m_l       : IN  STD_LOGIC;
        w_b          : IN  STD_LOGIC;
        ldpc         : OUT STD_LOGIC;
        wrd          : OUT STD_LOGIC;
        wr_m         : OUT STD_LOGIC;
        ldir         : OUT STD_LOGIC;
        ins_dad      : OUT STD_LOGIC;
        word_byte    : OUT STD_LOGIC);
END multi;

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

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

Modificaciones a la Unidad de Control

Ahora que ya tenemos implementada la lógica de control y la lógica del multiciclo podemos finalizar la unidad de control para que una las entidades y gestione correctamente el registro PC y el registro IR como hemos descrito anteriormente.

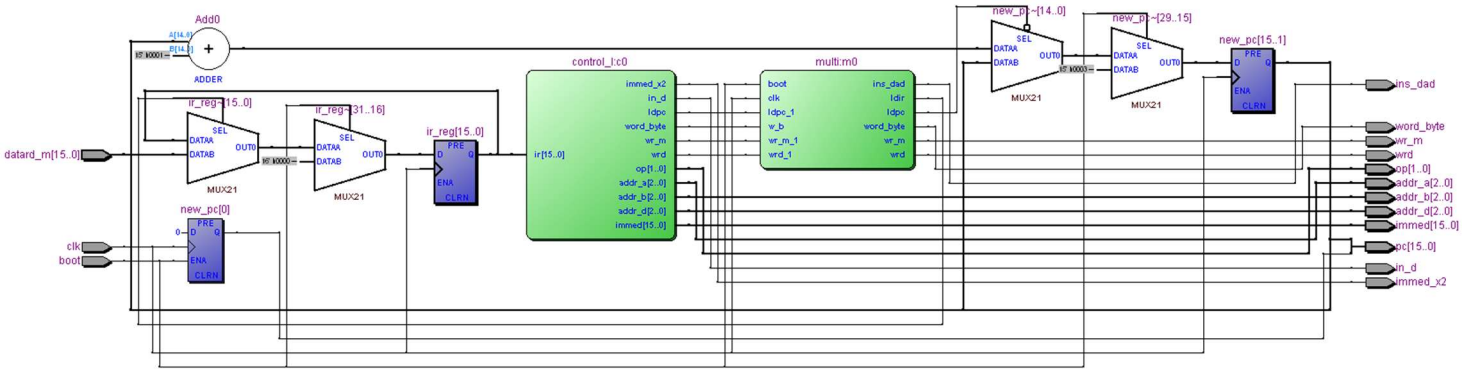
Así pues, las entradas de la **unidad de control** son:

- *clk*: La señal de reloj.
- *boot*: La señal de arranque del procesador. El Program Counter valdrá 0xC000 y no se incrementará mientras esta señal valga 1.
- *datard_m*: El valor leído de la memoria, ya sea una instrucción o un dato.

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

- *wrd*: El permiso de escritura en el Banco de registros.
- *addr_a*: La dirección del registro fuente del puerto *A* del Banco de registros.
- *addr_b*: La dirección del registro fuente del puerto *B* del Banco de registros.
- *addr_d*: La dirección del registro destino del Banco de registros.
- *immed*: El valor inmediato con extensión de signo extraído de la instrucción.
- *op*: Indica la operación que va a realizar la ALU.
- *pc*: Es el valor del Program Counter para que ahora el *datapath* pueda enviarlo a la memoria.
- *ins_dad*: Señal que le indica al *datapath* si en el bus de direcciones de la memoria debe poner la salida de la ALU o el PC.
- *in_d*: Señal que le indica al *datapath* si el valor que se almacenará en el Banco de registros procede de la ALU o de la memoria.
- *immed_x2*: Señal que le indica al *datapath* si hay que desplazar el inmediato o no.
- *wr_m*: Señal que le indica a la memoria si debe leer o escribir un valor.
- *word_byte*: Señal que le indica a la memoria si leer un *word* o debe leer un *byte* y extenderlo de signo.

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;
        datard_m  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        op        : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        wrd       : OUT STD_LOGIC;
        addr_a    : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        addr_b    : 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);
        ins_dad   : OUT STD_LOGIC;
        in_d      : OUT STD_LOGIC;
        immed_x2  : OUT STD_LOGIC;
        wr_m      : OUT STD_LOGIC;
        word_byte : OUT STD_LOGIC);
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.

Nueva versión del procesador

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

El módulo **proc** es el bloque de más alto nivel e instancia a todos los demás (unidad de control y *datapath*). Sigue teniendo sólo 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, pero ahora tiene más salidas: el bus de direcciones de la memoria, el bus de datos de la memoria, y las señales de control de esta memoria.

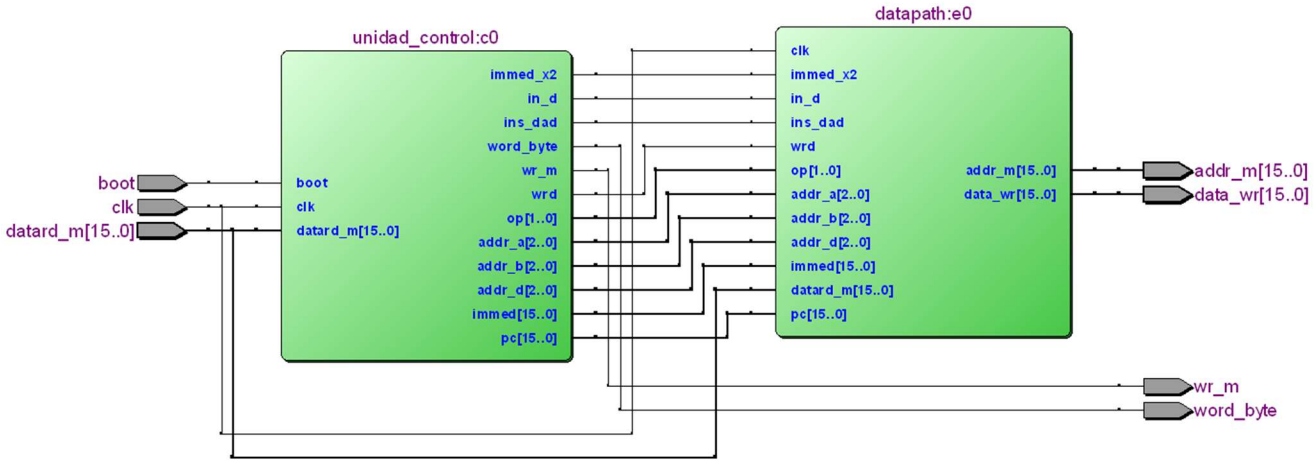
Las entradas al procesador son:

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

Las salidas del procesador son:

- *addr_m*: Es la dirección de memoria a la que se accederá en cada momento.
- *data_wr*: Es el valor que se envía a la memoria en el caso de hacer un *store*.
- *wr_m*: Es el permiso de escritura en la memoria. Esta es la señal *wr_m* generada por la lógica de control.
- *word_byte*: Señal *word_byte* generada por la lógica de control.

La estructura de bloques del procesador es la siguiente:



La cabecera de la entidad en VHDL es la siguiente:

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

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

Implementad un diseño en vhdl para esta entidad. Completad el contenido del fichero **proc.vhd**, que ya contiene la cabecera, con vuestra implementación. Con esto ya tenemos el procesador listo.

Simulación y prueba del procesador

Una vez terminado el diseño del procesador vamos a volver probar el diseño usando el programa de simulación *Modelsim*.

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" 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.

Ahora sólo tenemos que probarlo. En el módulo de prueba tenemos 3 nuevas señales que salen del procesador: *datawr*, *wr_m* y *word_byte*, que llevan respectivamente el dato a escribir, si escribimos o no y si la lectura o escritura es a nivel de *word* o *byte*. Éstas las hemos conectado a las entradas del módulo de prueba de memoria: *we* (write enable) y *byte_m* (lectura o escritura a nivel de *byte*).

Tenemos que hacer un programa que pruebe las instrucciones nuevas: LD, ST, LDB y STB. Recordad que la ALU multiplica por 2 los inmediatos codificados en las instrucciones LD y ST (accesos a nivel de *word*). En el lenguaje SISA, el inmediato de las instrucciones de acceso a memoria representa el desplazamiento en *bytes* que hay que sumar al registro base para obtener la dirección efectiva de acceso a la memoria. Así, si queremos acceder a los primeros tres elementos de un vector de *bytes* que estén almacenados consecutivamente a partir de la dirección que contiene el registro *Ra*, las instrucciones que pondríamos serían las siguientes: `LDB Rd,0(Ra)` ; `LDB Rd,1(Ra)` y `LDB Rd,2(Ra)`. En cambio, si el vector fuese de *words*, para acceder a los primeros tres elementos deberíamos poner las siguientes tres instrucciones: `LD Rd,0(Ra)` ; `LD Rd,2(Ra)` y `LD Rd,4(Ra)`. Es el compilador el que al detectar que la instrucción de acceso a memoria es a nivel de *words*, divide por dos el inmediato antes de codificarlo al lenguaje máquina. De modo que valores impares para el inmediato de las instrucciones LD y ST no están permitidos ya que no son codificables en el lenguaje máquina.

En el fichero de test os hemos dejado ya ensamblado el siguiente programa escrito el lenguaje ensamblador SISA.

```
movi r0, 64
movi r1, 65
movi r2, 66
movi r3, 67
movi r4, 68
movi r5, 80
; probamos los stores de byte
stb  2(r0), r1
stb  4(r3), r2
```

```
stb  0(r5), r3
stb  -12(r5), r4 ; off. negativo
stb  -2(r0), r5
; probamos los loads de byte
movi r7, 74
ldb  r4, -8(r7)
ldb  r3, 4(r3)
ldb  r2, 14(r2)
ldb  r1, 4(r0)
movi r7, 14
movhi r7, 0xc0 ; %r7 = 0xc00e
ldb  r6, -1(r7) ; lectura byte de ROM
stb  0(r0), r1
stb  1(r0), r6
stb  2(r0), r4
stb  3(r0), r2
stb  4(r0), r3
; loads de word
ld   r1, -2(r3)
ld   r2, 0(r3)
ld   r3, 2(r3)
ld   r7, -14(r7) ; lectura word de ROM
; volcado de los reg. - stores de word
movi r0, 64
st   32(r0), r1
st   34(r0), r2
st   36(r0), r3
st   38(r0), r4
st   40(r0), r6
st   42(r0), r7
halt
```

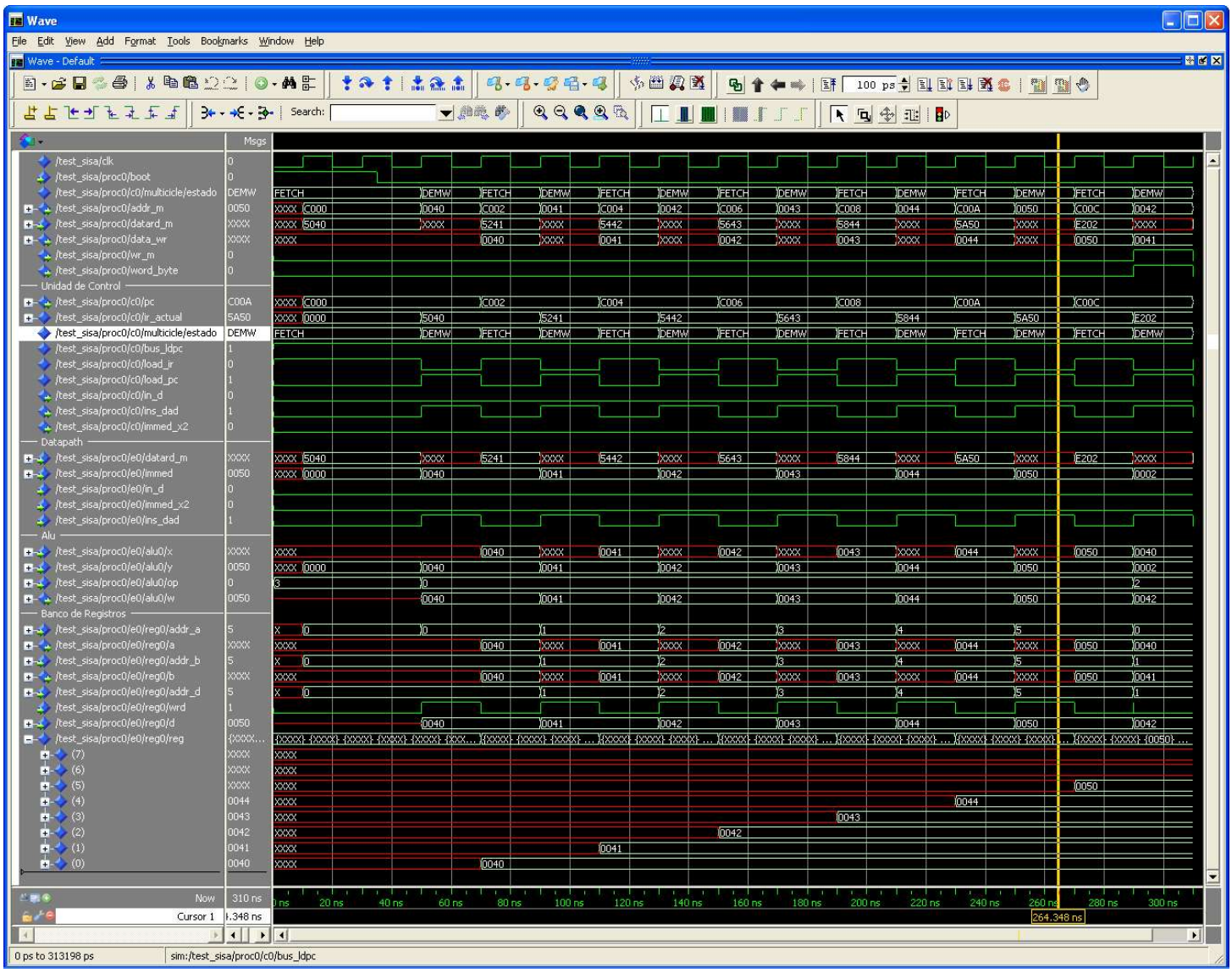
Este ejemplo hace lecturas y escrituras en tamaño de *word* y *byte*. Contiene tanto *offsets* positivos como negativos y lecturas y escrituras en el área de memoria que correspondería a la RAM y lecturas en área de memoria que correspondería a la ROM. Aunque de momento sólo se utiliza una RAM como ROM, la ROM estará en las direcciones de la 0xC000 a la 0xFFFF.

El resultado de la ejecución de este código deberían ser las siguientes modificaciones en memoria (los números están todos escritos en hexadecimal):

```
Mem[3f:3e] = xx50
Mem[41:40] = e244
Mem[43:42] = 4341
Mem[45:44] = xx42
Mem[47:46] = 42xx
Mem[51:50] = xx43
Mem[61:60] = e244
Mem[63:62] = 4341
Mem[65:64] = xx42
Mem[67:66] = 0041
Mem[69:68] = ffe2
Mem[6b:6a] = 5040
```

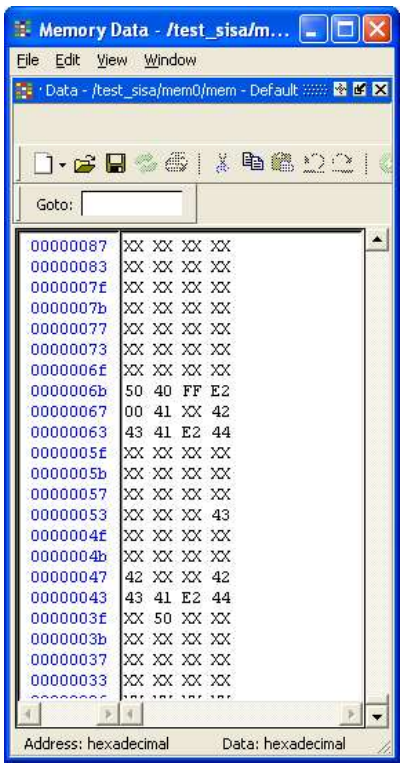
Vamos ahora a simular el programa con el *Modelsim*. Antes de empezar a simular nuestro procesador deberemos copiar los ficheros **test_sisa.vhd**, **memory.vhd** y **contingut.memoria.hexa.rom** del directorio Test al directorio principal de nuestro proyecto. Si no lo hacemos tendremos un error de ejecución más adelante al iniciar la simulación. Para simular el programa seguiremos los pasos descritos en la etapa anterior: Crearemos un nuevo proyecto en el *Modelsim*, importaremos todos los ficheros, los compilaremos e iniciaremos la simulación del módulo **test_sisa**.

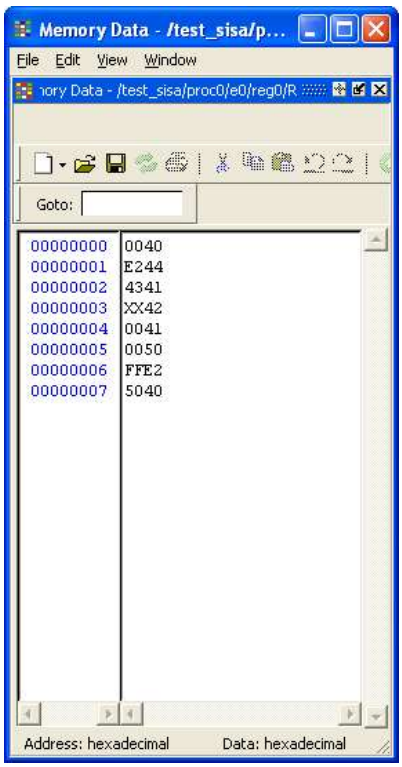
Añadiremos las señales que consideremos interesantes y ejecutaremos unos 310ns (run 310ns). Veremos en el cronograma, similar al de la siguiente figura, como se han ejecutado las 7 primeras instrucciones del programa de ejemplo, las de movimiento que inicializan los registros y la primera de acceso a memoria. Recordad que el tiempo de ciclo es de 20ns y que cada instrucción tarda 2 ciclos.



Si simulamos hasta el final del programa (29 instrucciones más * 2 ciclos/instrucción * 20ns/ciclo ➔ run 1160ns), podemos inspeccionar el contenido de la memoria para ver si el juego de pruebas funciona correctamente. El programa *Modelsim* dispone de una vista para ver el contenido de todas las memorias de nuestro diseño. Para ello seleccionamos la opción "View ➔ Memory List(w)" del menú principal del *Modelsim*. Se nos abrirá una ventana con el listado de todas las memorias definidas en nuestro diseño: el Banco de registros y la memoria de test. Seleccionamos la memoria que deseamos ver y se nos abrirá una ventana llamada "*Memory Data*" con su contenido. En esta ventana, con el botón derecho del ratón podemos seleccionar la opción "*Properties…*" donde podremos escoger el formato en que deseamos ver el contenido (binario, hexadecimal, …) y el número de valores por cada línea.

Si la simulación ha sido correcta, el contenido final de la memoria y del banco de registros deberían ser como en las siguientes figuras.



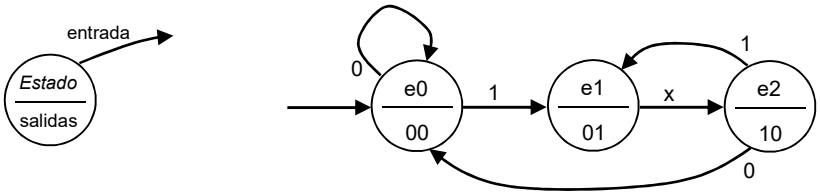


Por tanto, como podemos observar el contenido final de la memoria es exactamente lo que debería ser y por lo tanto el circuito pasa el juego de pruebas.

Cread un nuevo juego de pruebas propio (distinto al del ejemplo) que pruebe el procesador. Escribid el programa en ensamblador SISA, ensambladlo y cread el nuevo fichero de memoria para el test.

Anexo A: Implementación de grafos de estados

Implementar grafos de estados es VHDL es relativamente sencillo. Suponed que queremos implementar el siguiente grafo de estados.



Hay varias maneras de hacerlo. La primera manera es creando un código VHDL directamente en el editor. Hay muchos estilos distintos para crear máquinas de estado del modelo de Moore y cada programador puede crear el suyo. El estilo que genera un código más leíble es el que separa las acciones en dos *process* distintos: uno para calcular el estado siguiente y otro para calcular las salidas. Este estilo es el que nos propone el *quartus II* si usamos sus plantillas.

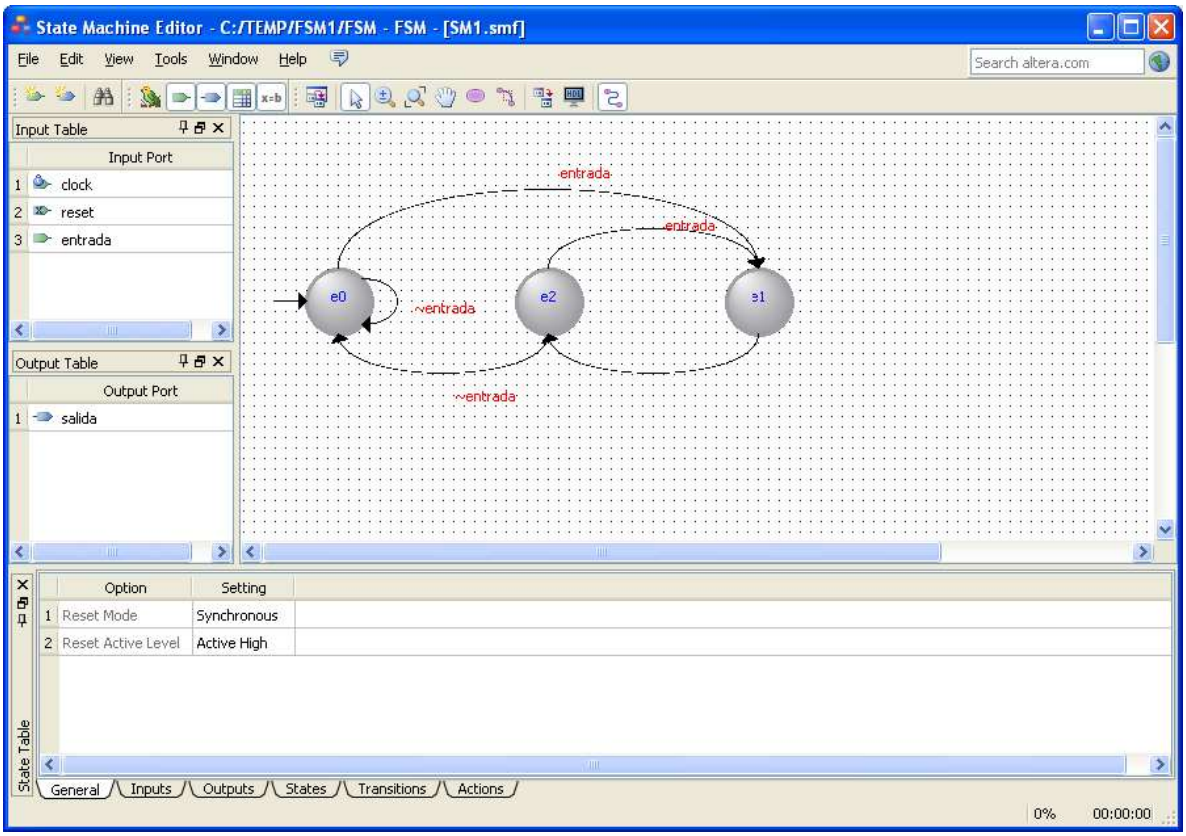
Para crear un grafo de estados directamente en VHDL (con el proyecto ya creado) seleccionamos **File→New** en el menú de pestañas superior. En la ventana emergente seleccionamos **Design Files→VHDL file**. En el fichero que se abre crearemos el diseño de nuestro grafo de estados del modelo de Moore utilizando el lenguaje de definición de hardware VHDL. Para ello utilizaremos la plantilla que *quartus* nos proporciona. Para usar la plantilla, simplemente buscamos en la parte superior de la ventana de código que acabamos de crear el icono correspondiente a la opción "*Insert Template*". Se abrirá una ventana emergente con un listado desplegable de tipo árbol a la izquierda de la ventana. Seleccionamos ***VHDL → Full Designs → State Machines → Four-State Moore State Machine***, seleccionamos la opción **insert** y luego **close**, y nos aparecerá el código en el editor. Una vez adecuado a nuestras necesidades nos debería quedar un código similar al siguiente:

```
library ieee;
use ieee.std_logic_1164.all;

entity FSM is
   port( clk     : in  std_logic;
         reset   : in  std_logic;
         entrada : in  std_logic;
         salidas : out std_logic_vector(1 downto 0)
   );
end entity;

architecture rtl of FSM is
   -- Construye un tipo enumerado para una máquina de tres estados (más fácil de leer el código)
   type state_type is (e0, e1, e2);

   -- Registro para mantener el estado actual
   signal estado_actual : state_type;

begin
   -- Lógica de transición entre estados.
   -- Calcula el estado siguiente a partir del actual y las entradas.
   -- Las transiciones entre estados son síncronas.
   process (clk, reset)
   begin
          if reset = '1' then
                 estado_actual <= e0;
          elsif (rising_edge(clk)) then
                 case estado_actual is
                        when e0=>
                               if entrada = '1' then
                                      estado_actual <= e1;
                               else
                                      estado_actual <= e0;
                               end if;
                        when e1=>
                               estado_actual <= e2;
                        when e2 =>
                               if entrada = '1' then
                                      estado_actual <= e1;
                               else
```

```
                                        estado_actual <= e0;
                                end if;
                        end case;
                end if;
        end process;

        -- Las salidas solo dependen del estado actual.
        -- Las salidas solo se actualizan cuando el estado cambia.
        process (estado_actual)
        begin
                case estado_actual is
                        when e0 =>
                                salidas <= "00";
                        when e1 =>
                                salidas <= "01";
                        when e2 =>
                                salidas <= "10";
                end case;
        end process;
end rtl;
```

Cuando hay muchas señales y/o muchos estados, al hacer cambios en el grafo de estados es fácil olvidarse alguna cosa si tecleamos directamente el código VHDL. *Quartus* dispone de un editor de grafos de estados que evitará posiblemente que cometamos algunos descuidos habituales. Con esta herramienta podremos crear gráficamente un grafo de estados y determinar las transiciones y los valores de las salidas para cada estado.

Para crear un grafo de estados con este editor seleccionamos **File→New** en el menú de pestañas superior. En la ventana emergente seleccionamos **Design Files→ State Machine File**. Se abrirá un editor gráfico y en el podremos empezar a dibujar el grafo de estados. Este editor creará un fichero con la extensión *".smf"*. En la parte superior de la ventana de la herramienta hay un icono llamado "*State Tool*" que permite crear los estados que queramos y el icono "*Transition Tool*" que permite crear las transiciones. También podemos utilizar el icono "*State Machine Wizard*" que nos permite crear (y modificar) un grafo de estados automáticamente indicando sus principales características.



Con las pestañas de la parte inferior de la ventana de la herramienta podremos modificar fácilmente todos los valores del grafo de estados en cualquier momento. Finalmente podremos generar el código VHDL para el proyecto con el icono "*Generate HDL File*".