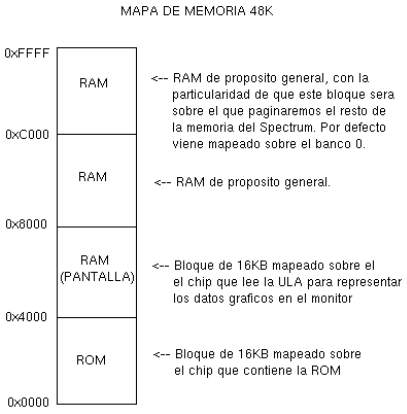
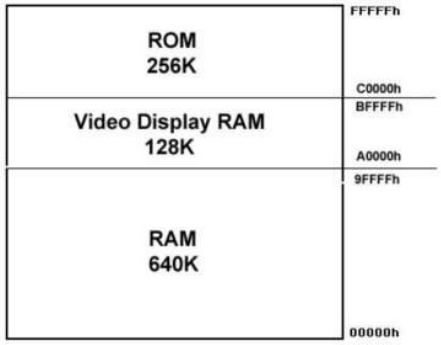
Subetapa 2.2: Controlador de memoria

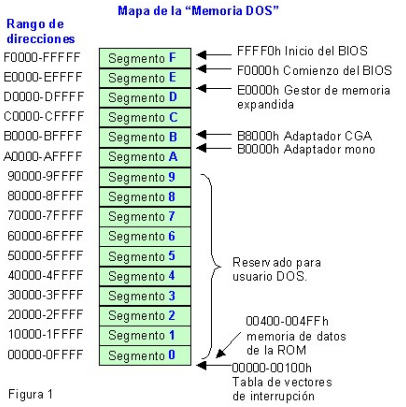
En esta etapa se sustituirá el *stub* de memoria usado para simular y verificar el comportamiento de las instrucciones de acceso a memoria de la subetapa 2.1 por un controlador de memoria. Este controlador usará los chips de memoria de la placa de desarrollo. Una vez adaptado a los componentes de la placa de desarrollo de la FPGA se probará el diseño físicamente.

El modelo de memoria que vamos a implementar en el procesador que estamos diseñando será parecido al que tiene las consolas y teléfonos móviles actuales. Realmente se va a parecer más al que tenían los ordenadores personales de 8 bits de la década de los 80 (para los nostálgicos como el profesor: Commodore 64, Amstrad CPC, ZX Spectrum, Nintendo Gameboy, …). Tenemos un espacio de direccionamiento de 64 KBytes, 2^{16} posiciones de memoria (desde la 0x0000 hasta la 0xFFFF) de un *byte*. Una parte de las posiciones de memoria corresponden a la memoria RAM para ejecutar aplicaciones y otra parte para la memoria ROM (o flash) donde se ubica el firmware/sistema operativo/bios del procesador.

En las siguientes figuras podéis ver como ejemplo el mapa de memoria de un ZX Spectrum en el que el bus de direcciones era de 16 bits (64 KB) y el de un PC (8086, 8088, 80286) donde el bus de direcciones era de 20 bits (1 MB).







Mapa de memoria del Zx Spectrum *Mapa de memoria del PC*

El modelo de memoria que implementaremos en nuestro procesador tendrá los primeros 48 KBytes de memoria RAM (de 0x0000 a 0xBFFF) para las aplicaciones y los últimos 16 KBytes (de 0xC000 a 0xFFFF) para la ROM, que en nuestro caso será donde se ubicará el firmware del sistema operativo que ejecutaremos (cuando lo desarrollemos).

Modelo de simulación realista

El modelo de simulación que hemos realizado cumple con lo que tiene que hacer: leer y escribir a memoria. Pero si nuestra intención es utilizar el procesador en una placa comercial de desarrollo, entonces el modelo de memoria que estamos utilizando es poco realista. En primer lugar, estamos usando una memoria RAM para simular la ROM donde estaría almacenado el programa que se ejecuta. Después, no sabemos exactamente cómo serán los chips de memoria física que tendrá la placa de desarrollo. Para hacer una simulación más realista, primero tenemos que conocer dónde haremos la implementación.

Las placas de desarrollo de que disponemos (DE1 y DE2-115) tienen chips de memoria SDRAM, SRAM, Flash y EPROM a nuestra disposición. Nosotros usaremos una parte de los chips SRAM (48KB), por simplicidad, para implementar la memoria RAM y una parte de los chips de memoria Flash (16KB) para implementar la parte correspondiente a la memoria ROM.

Ambas placas tienen un oscilador a 50Mhz que lo usaremos como reloj base. La placa DE2-115 dispone de 2 MB SRAM y 8 MB Flash. La placa DE1 dispone de 512 KB SRAM y 4 MB Flash. Ambas placas tienen suficiente capacidad para implementar nuestras necesidades de memoria.

En el caso de la placa DE1 tiene un único chip de memoria SRAM de 512KB de la marca ISSI. Es el modelo IS61LV25616 o el IS61WV25616EDBLL y funciona a 10ns. Este chip de memoria está organizado en palabras de 16 bits y su velocidad máxima de funcionamiento es de 100 Mhz. El chip de memoria Flash es de la marca Spansion, tiene de 4 MBytes y funciona a 70ns. Es el modelo S29AL032D y está organizado en *bytes*.

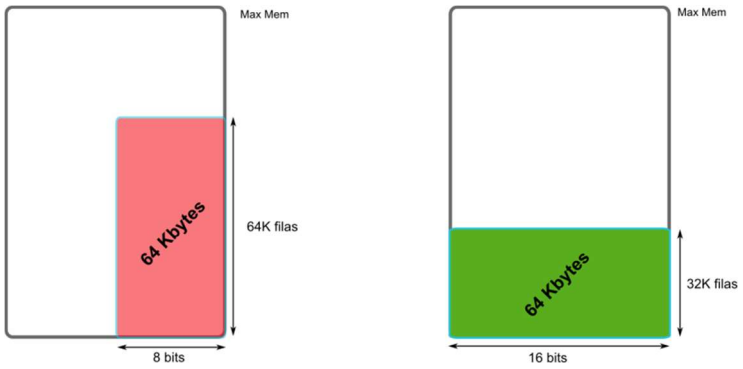
En el caso de la placa DE2-115 tiene un único chip de memoria SRAM de 2MB de la marca ISSI. Es el modelo IS61WV102416BLL y funciona a 8ns. Este chip de memoria está organizado en palabras de 16 bits y su velocidad máxima de funcionamiento es de 125 Mhz. El chip de memoria Flash es de la marca Spansion, tiene 8 MBytes y funciona a 90ns. Es el modelo S29GL032N y está organizado en *bytes*.

En los *datasheets* correspondientes podéis encontrar toda la información necesaria: diagramas de tiempo de las señales, los distintos métodos de escritura, …

Además, estas memorias son muy rápidas. Las memorias SRAM de la placa pueden funcionar a una velocidad superior a la máxima de la FPGA. Esto nos garantiza que podremos hacer lecturas y escrituras en un único ciclo. Esta característica no es habitual en los procesadores actuales.

Si queremos adaptar nuestro diseño para que funcione físicamente, debemos hacer un controlador de memoria para los correspondientes chips de memoria de la placa. Para saber cómo están conectados estos chips a la FPGA, disponemos del manual del fabricante de la placa donde se describen las conexiones.

Como el tamaño físico de las memorias de las placas de desarrollo es mucho más grande que el requerido en el diseño del procesador, deberemos decidir qué parte de esas memorias utilizamos. Las dos alternativas principales son usar el mismo número de filas y ancho que el espacio de direccionamiento del procesador (opción roja), o usar todo el ancho de la palabra de la memoria física reduciendo el número de filas que usaremos (opción verde).



La primera opción tiene la ventaja que podemos hacer coincidir las direcciones de memoria que genera el procesador con las de la memoria física. Entonces el controlador de memoria sólo debe añadir los bits que faltan a la dirección física. Por el contrario, como necesitamos hacer lecturas de *words* en memoria (por ejemplo: la lectura de la instrucción a ejecutar) necesitaremos hacer dos accesos consecutivos a posiciones distintas de memoria y luego concatenar los bytes. Estos dos accesos a memoria deberán hacerse en un único ciclo de reloj del procesador. Esto complica el controlador de memoria y lo hace bastante más ineficiente.

La segunda opción es más simple. Aunque las direcciones de memoria generadas por el procesador no coinciden con las que recibirá la memoria, la transformación que tiene que hacer el controlador de memoria es simplemente un desplazamiento de bits. En el caso que se desee leer únicamente un *byte*, el controlador de memoria puede usar el bit 0 de la dirección de acceso a memoria que genera el procesador para seleccionar el *byte* adecuado. En el procesador que estamos diseñando usaremos esta segunda opción.

El controlador de memoria habrá de tener presente algunos detalles que lo hagan independiente del procesador, y así no tener que modificarlo. El sistema SISA tiene un bus de direcciones de 16 bits, con un espacio de direccionamiento de 2^{16} = 65536 *bytes* y por tanto, sólo una parte del chip de SRAM es necesaria para implementar la memoria. Utilizaremos las primeras 48KB de memoria de la SRAM direccionándolas a nivel de *word*. O sea, de las direcciones 0x00000 a la

0x05FFF de la SRAM. Las direcciones que recibe el chip de memoria no coinciden con las que genera el procesador SISA debido a que el nivel de direccionamiento es distinto. Tampoco coinciden en la anchura del bus de direcciones. El controlador de memoria deberá encargarse de hacer la transformación de las direcciones de la forma adecuada.

En el caso de la memoria flash es un poco más complicado. Usaremos las primeras 16KB de la Flash, de la dirección 0x000000 a la 0x003FFF. Pero esta zona de memoria corresponderá con la zona que empieza en la dirección lógica 0xC000 del SISA. Será el controlador de memoria quien deberá realizar la transformación de espacios de direcciones. Además esta zona de memoria es de sólo lectura, por tanto el controlador de memoria también se encargará de impedir las escrituras.

A parte, tenemos un problema en la lectura de palabras de 16 bits de la memoria Flash, ya que el direccionamiento que utiliza es a nivel de *byte* y en un mismo ciclo no se pueden leer dos *bytes* a la vez. Por lo tanto, utilizaremos la segunda mitad del ciclo para leer el segundo *byte* en el caso de que tengamos que hacer lecturas en la ROM a nivel de *word*. Al necesitar hacer dos lecturas en un único ciclo necesitaremos una señal de reloj adicional.

Controlador de memoria

El controlador que diseñaremos debe acceder correctamente a los chips. Para ello deberemos consultar los *datasheets* del fabricante de la memoria para ver los diagramas temporales de las señales para hacer las lecturas y escrituras. Hay muchas formas de implementar un controlador de memoria, pero la más sencilla (y probablemente la más ineficiente) consiste en generar los valores de las señales en el orden correcto mediante un grafo de estados. Puesto que hay que hacer una lectura o escritura en un único ciclo del procesador, el grafo de estados deberá transicionar varios estados en dicho tiempo, por lo que el controlador deberá tener un reloj más rápido que el del procesador.

Para generar este reloj y ajustarnos a la frecuencia adecuada podríamos usar un PLL (*Phase-locked loop*) de los que dispone la FPGA. Como aún no hemos calculado el tiempo de ciclo de procesador (ya que está a medio implementar), de momento nos simplificaremos la faena. Al controlador de memoria le enviaremos el reloj a 50Mhz y al procesador le enviaremos una señal de reloj 8 veces más lenta (podemos usar un contador de para hacer un divisor de frecuencia). O sea a 6,25Mhz. Esto nos va a permitir crear un grafo de estados de control en el controlador de memoria que en su secuencia de ejecución más larga tenga hasta 8 ciclos, aunque dependiendo del diseño que hagamos, probablemente no necesitemos más de 3 o 4 ciclos. Si luego vemos que no necesitamos tantos ciclos podemos adaptar este divisor de frecuencia al valor más adecuado.

A la hora de implementar el controlador y dependiendo del modo de escritura que estemos utilizando, también debemos tener en cuenta que la memoria SRAM es muy rápida (10ns) y es asíncrona, lo que quiere decir que si bajamos la señal del *write enable* mientras se está calculando la dirección efectiva es posible que se escriban a otras direcciones no deseadas en memoria. Estos problemas pueden existir debido a la diferencia de velocidades entre el FPGA y la SRAM. Este problema es bastante atípico, porque normalmente es el procesador quien es más rápido que la memoria, pero en este caso ocurre al revés.

En caso de que tengamos problemas, podemos usar el analizador lógico *SignalTap II* para comprobar que no se estén produciendo este tipo de situación y poder ajustar mejor las señales de reloj. ;-)

Otra solución para implementar el controlador de memoria y no hacer un grafo de estados, es tener más de una señal de reloj de entrada a distintas frecuencias (y sincronizadas) y usar esas señales para enmascarar a cada momento las señales de control de la memoria para que tengan su valor correcto sólo en el momento adecuado.

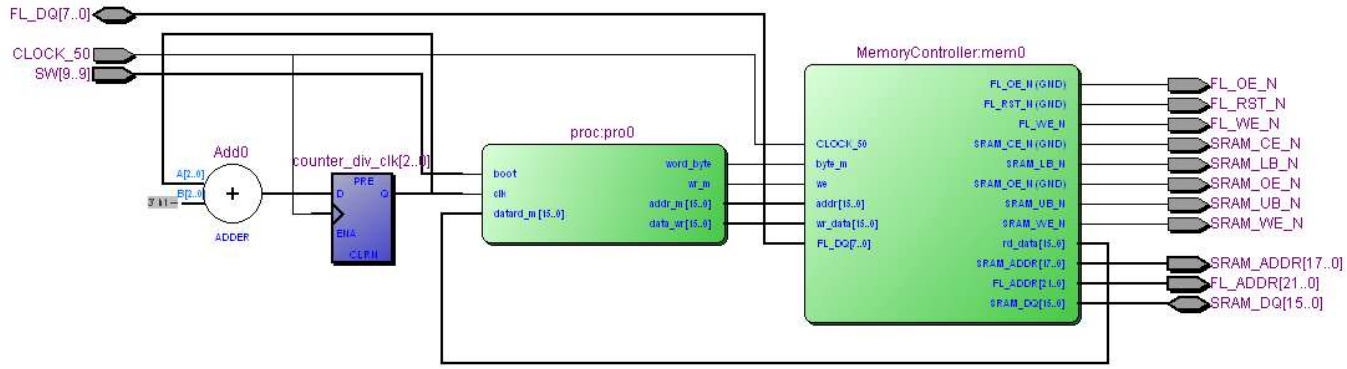
Módulo SISA SoC

SoC significa *System on Chip* y representa todo un sistema (controladores, relojes, dispositivos, procesadores, DMAs, ...) que conforma un computador entero pero diseñado para que quepa todo en un chip. A menudo, cuando se trabaja con dispositivos programables (como las FPGA), estos no implementan sólo un procesador sino un sistema completo. Este es

nuestro caso, en esta parte aún no tenemos dispositivos de entrada y salida, pero ya hemos creado un nuevo controlador: el de memoria. Por lo tanto, formaremos un módulo superior (el sistema) que incluya el procesador SISA (la entidad **proc**) y el controlador de memoria, y que los conecte adecuadamente. A parte, aprovecharemos este módulo para generar las señales de reloj con las frecuencias adecuadas para cada uno de los componentes. Más adelante, este módulo de sistema **SoC** contendrá los controladores de los dispositivos de entrada y salida y toda la lógica para formar el computador entero.

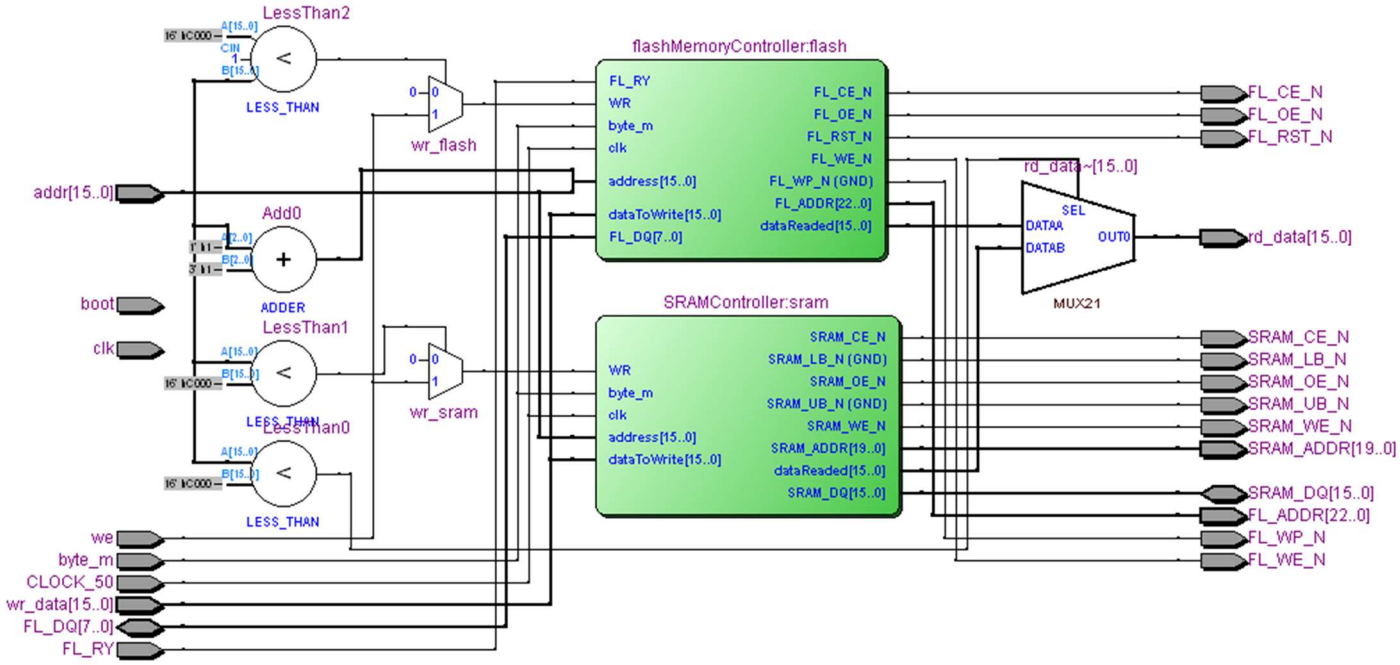
Este es el módulo de nivel superior de la FPGA y es el que se conecta con los dispositivos externos como pueden ser los chips de memoria. Ahora deberíamos crear una entidad llamada **sisa** que será el módulo de nivel superior. Esta entidad será parecida al ***test_sisa*** de la etapa anterior pero modificada. En ella eliminaremos el *stub* de memoria que creamos al procesador base y lo sustituiremos por las conexiones reales a los chips.

Un posible esquema de bloques de esta entidad se muestra a continuación:



En él puede verse como las entradas son el reloj, la señal de *boot* y los datos procedentes de las memorias (en buses bidireccionales) y como salen todas las señales para controlar estas memorias.

Un posible esquema para el controlador de memoria seria el siguiente:



En él puede verse que a su vez está formado por dos controladores de memoria distintos para cada uno de los dos tipos de memoria (la SRAM y la Flash). Este módulo se encarga de gestionar correctamente los permisos de escritura de ambos y de seleccionar la fuente del dato que pondrá en el bus de lectura del procesador.

Finalmente tendríamos un controlador para cada una de los tipos de memoria que se encargará de gestionar sus señales de control generando sus valores correctos en los momentos adecuados.

La implementación de los módulos

Vamos a implementar los módulos necesarios para la memoria del SISA en los chips de la placa de desarrollo. En primer lugar prescindiremos de la memoria flash por motivos de tiempo de desarrollo, simplicidad y seguridad. Así que usaremos únicamente la SRAM para implementar los 64 KBytes direccionables, pero aun así, el controlador de memoria no deberá permitir escrituras en los 16 KBytes más altos ya que son los que técnicamente corresponderían a la zona de la ROM

Las señales de control y las restricciones temporales para controlar la memoria flash son bastante más complejas que las de la SRAM. Además, la memoria Flash de la placa es el único componente que podemos dejar inutilizable por culpa de un error en el diseño. Las memorias Flash tienen un tiempo de vida limitado y depende del número de escrituras que hagamos en ellas. Su tiempo de vida es de aproximadamente unas 10.000 escrituras. Así que un error en el controlador de memoria podría dejar inservible la memoria flash en menos de un segundo. Por este motivo sólo usaremos la memoria SRAM pero trataremos la parte superior del espacio direccionable como si fuese una ROM (sin poder hacer escrituras).

Implementación del módulo del controlador de memoria SRAM.

Este módulo se encarga de controlar el chip de memoria SRAM de la placa de desarrollo. En el *datasheet* que nos suministra el fabricante podemos ver los modos y los cronogramas de tiempo de las señales que nos permiten hacer lecturas y escrituras. A este módulo le llamaremos **SRAMController.vhd**.

El número de entradas depende ya un poco de la opción escogida para implementar el controlador. Pero básicamente son las siguientes. Ahora tendremos un bus bidireccional que corresponde al bus de datos de la memoria.

En una implementación posible para este módulo estas serían las señales de entrada y salida.

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

- *clk*: La señal de reloj.
- *address*: Es la dirección de memoria a la que el procesador SISA desea acceder (de 16 bits).
- *dataToWrite*: Es el valor procedente del procesador que se escribirá en la memoria en el caso de hacer un *store*.
- *WR*: Señal que le indica a la memoria si debe leer o escribir un valor.
- *byte_m*: Señal que le indica a la memoria si ha de realizar un acceso a un *word* o debe realizar un acceso a un *byte* y extenderlo de signo.

Las salidas de la **SRAMController** son:

- *SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N* y *SRAM_WE_N*: Son las señales de control de la memoria descritas en el datasheet del fabricante.
- *SRAM_ADDR*: Es el bus de direcciones correspondiente al chip de memoria. Su valor depende de donde hayamos mapeado los 64KB de memoria del procesador SISA. Su ancho depende del modelo de placa de desarrollo que estemos usando.
- *dataReaded*: Es el valor que se ha leído de la memoria. En caso de que se hubiese leído un *byte* se habría extendido el signo.

El bus *SRAM_DQ* es un bus bidireccional (de entrada y salida) de la **SRAMController** por el que van los datos hacia y desde el chip de memoria. Al ser un bus bidireccional implica que diversos componentes pondrán datos en él en distintos

momentos. Hemos de garantizar en el diseño que dos componentes nunca pongan un valor en el bus de forma simultánea. Por ello, cuando un componente no desee escribir en el bus deberá poner sus salidas en alta impedancia (High-Z).

La cabecera en VHDL de esta posible implementación de la entidad es la siguiente:

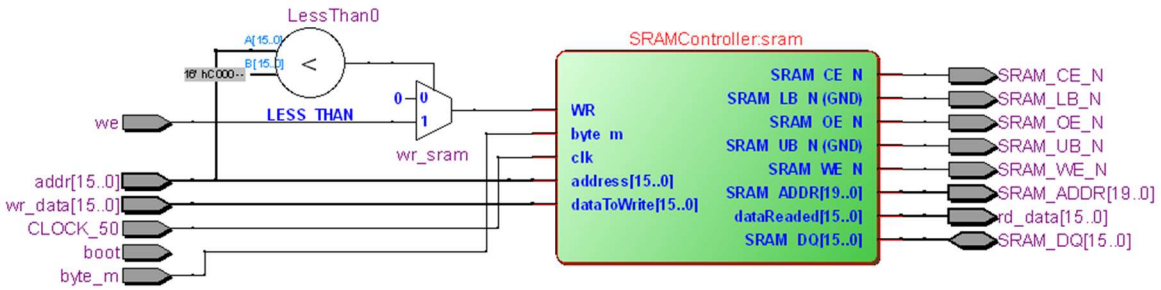
```
ENTITY SRAMController IS
  PORT (clk         : in  std_logic;
        SRAM_ADDR   : out std_logic_vector(1? downto 0); -- 17 para DE1 y 19 para DE2
        SRAM_DQ     : inout std_logic_vector(15 downto 0);
        SRAM_UB_N   : out std_logic;
        SRAM_LB_N   : out std_logic;
        SRAM_CE_N   : out std_logic := '1';
        SRAM_OE_N   : out std_logic := '1';
        SRAM_WE_N   : out std_logic := '1';
        address     : in std_logic_vector(15 downto 0) := "0000000000000000";
        dataReaded  : out std_logic_vector(15 downto 0);
        dataToWrite : in std_logic_vector(15 downto 0);
        WR          : in std_logic;
        byte_m      : in std_logic := '0');
END SRAMController;

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

Implementad un diseño en vhdl para esta entidad. Completad el contenido del fichero **SRAMController.vhd**, que ya contiene la cabecera, con nuestra implementación. Podéis adaptar y cambiar las señales a vuestra implementación.

Implementación del controlador de memoria.

Este módulo debería controlar los dos tipos de memoria, adecuar los mapeos de memoria, etc…, tal y como se habría descrito al principio de esta etapa de desarrollo del procesador. Pero como ahora sólo tenemos un tipo su única función es la de inhibir las escrituras en la zona alta de la memoria (de 0xC000 a 0xFFFF) que es la que correspondería a la ROM. A este módulo le llamaremos **MemoryController.vhd**.

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



La cabecera en VHDL de la entidad, correspondiente al esquema de bloques de la figura anterior, es la siguiente:

```
ENTITY MemoryController IS
  PORT (addr        : in  std_logic_vector(15 downto 0);
        wr_data     : in  std_logic_vector(15 downto 0);
        rd_data     : out std_logic_vector(15 downto 0);
        we          : in  std_logic;
        byte_m      : in  std_logic;
        CLOCK_50    : in  std_logic;
        SRAM_ADDR   : out std_logic_vector(1? downto 0); -- 17 para DE1 y 19 para DE2
        SRAM_DQ     : inout std_logic_vector(15 downto 0);
        SRAM_UB_N   : out std_logic;
```

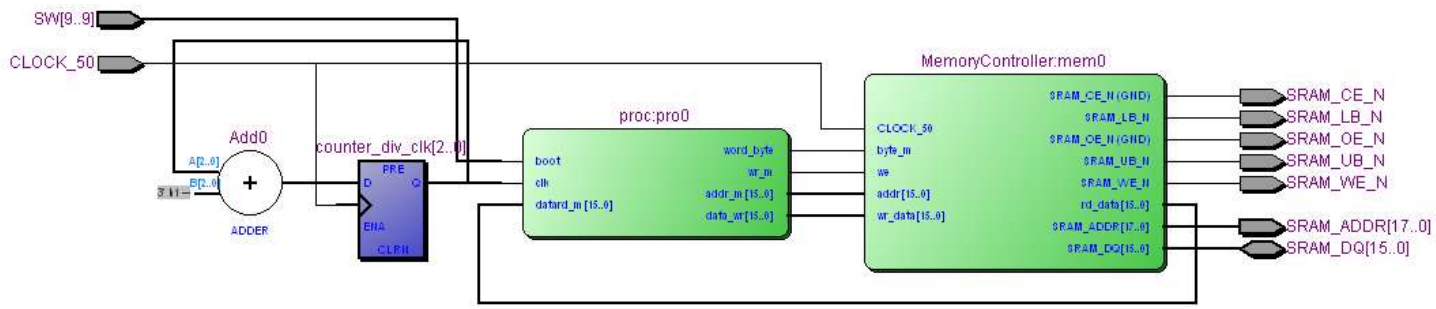
```
        SRAM_LB_N   : out std_logic;
        SRAM_CE_N   : out std_logic := '1';
        SRAM_OE_N   : out std_logic := '1';
        SRAM_WE_N   : out std_logic := '1');
END MemoryController;

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

Implementad un diseño en vhdl para esta entidad. Completad el contenido del fichero **MemoryController.vhd**, que ya contiene la cabecera, con nuestra implementación. Podéis adaptar y cambiar las señales a vuestra implementación.

Implementación del módulo SISA.

El módulo **sisa** es el bloque de más alto nivel e instancia a todos los demás. De momento al procesador y al controlador de memoria. Crearemos la entidad llamada sisa. Esta entidad será parecida al test_sisa de la etapa anterior pero modificada. En ella eliminaremos el *stub* de memoria que creamos al procesador base y lo sustituiremos por las conexiones reales a los chips. Este módulo generará las señales de reloj con las frecuencias adecuadas para el procesador (6,25MHz) y para la memoria (50MHz)

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



En el puede verse como las entradas son el reloj, la señal de *boot* (procedente de un botón) y los datos procedentes de la memoria (en buses bidireccionales) y como salen todas las señales para controlar estas memorias.

Utilizad un *switch*, el SW[9] concretamente, para inicializar el procesador.

La cabecera en VHDL de la entidad, correspondiente al esquema de bloques de la figura anterior, es la siguiente:

```
ENTITY sisa IS
  PORT (CLOCK_50    : IN  STD_LOGIC;
        SRAM_ADDR   : out std_logic_vector(1? downto 0); -- 17 para DE1 y 19 para DE2
        SRAM_DQ     : inout std_logic_vector(15 downto 0);
        SRAM_UB_N   : out std_logic;
        SRAM_LB_N   : out std_logic;
        SRAM_CE_N   : out std_logic := '1';
        SRAM_OE_N   : out std_logic := '1';
        SRAM_WE_N   : out std_logic := '1';
        SW          : in  std_logic_vector(9 downto 9)); --para el reset
END sisa;

ARCHITECTURE Structure OF sisa IS
BEGIN
        .
        .
```

```
.
END Structure;
```

Implementad un diseño en vhdl para esta entidad. Completad el contenido del fichero **sisa.vhd**, que ya contiene la cabecera, con nuestra implementación. Podéis adaptar y cambiar las señales a vuestra implementación. Ahora ya tenemos el módulo completo de todo el sistema (*System on Chip*) y ya podemos probarlo en la placa de desarrollo.

Simulación de la memoria.

Como etapa intermedia, primero se sustituirá el *stub* de memoria de la etapa anterior por un modelo de simulación comercial exacto similar al chip de memoria que se utilizará finalmente cuando el diseño funcione en la placa de desarrollo. De este modo, el diseño se podrá simular con el *Modelsim* de la misma manera que en la etapa anterior.

Al igual que en la etapa anterior, tenéis disponible un módulo de test llamado **test_sisa** y lo encontraréis en el directorio "Test". Además, también encontrareis los ficheros que contienen la implementación de la memoria y una función para inicializarla con un fichero. Esta función carga el contenido de un fichero (*contingut.memoria.hexa16.rom*) a partir de la dirección 0x06000, que es donde se supone que comienza el área de sólo lectura del sistema y donde debe encontrarse la primera instrucción que se ejecutará. El formato de este fichero es similar al de la etapa anterior pero ahora en cada fila hay valores agrupados en 16 bits en vez de 8 bits. Antes de usar el módulo **test_sisa** debéis revisar como se ha hecho el *boot* del procesador para que sea todo consistente con vuestro diseño.

La memoria comercial emulada es la Cypress CY7C1021BN cuyo comportamiento es igual a las que hay en la placa de desarrollo. La memoria es de 1-Mbit (64K x 16) SRAM. Eso significa que tiene 64k filas de 16 bits cada una, o sea 128KB. Como el SISA sólo gestiona 64KB y el nivel de direccionamiento es a nivel de byte, sólo se usarán las 64KB primeras y la dirección de *boot* es la 0x06000. La especificación en vhdl de la memoria simulada está formada por tres ficheros. El fichero "**async_64Kx16.vhd**" contiene el comportamiento de la memoria y los ficheros "**package_utility.vhd**" y "**package_timing.vhd**" contienen las definiciones de los retardos de las señales y algunas utilidades.

Prueba física en la placa de desarrollo.

Una vez estemos seguros de que nuestro controlador de memoria funciona correctamente con el simulador, podemos pasar a probarlo en la placa de desarrollo. Para probar nuestro procesador debemos realizar dos tareas: grabar el procesador en la FPGA, poner el programa que deseemos ejecutar en la memoria SRAM y *bootar* el procesador.

Los pasos para hacer esta tarea son los siguientes.

1. Programar la FPGA con la configuración para el Panel de control de la placa de desarrollo.
2. Con la aplicación de Panel de Control del fabricante de la placa de desarrollo, cargar el programa que deseemos ejecutar en la dirección 0x6000 de la SRAM. El contenido de esta memoria no se perderá mientras tenga alimentación eléctrica, aunque se reprograme la FPGA. El formato del fichero que contiene el programa a ejecutar es distinto del que usábamos en el *stub* de memoria y difiere un poco si usamos la placa DE1 o la DE2-115. Además la extensión del fichero debe ser "*.hex*". Consultad el formato concreto en el manual del fabricante.
3. Programar la FPGA con el procesador que hemos diseñado.
4. Ejecutar nuestro programa (desactivando la señal de *boot*).

Si el resultado de nuestro programa modifica la memoria SRAM y queremos consultar su contenido para ver que la simulación ha sido correcta haremos los siguientes pasos:

5. Volver a programar la FPGA con la configuración para el Panel de control de la placa de desarrollo
6. Con el Panel de control volcaremos el contenido de la SRAM a un fichero especificando la dirección desde donde queremos empezar el volcado y su tamaño.

7. Analizar el contenido del fichero para ver si todo ha funcionado como deseábamos.

Una vez se enciende la placa base el contenido de la memoria SRAM es completamente aleatorio. Como la mayoría de juegos de prueba de esta etapa son para comprobar si se escribe bien en memoria, para facilitar la rápida localización de los resultados en el fichero de volcado es recomendable previamente haber inicializado la zona de la memoria donde vamos ha hacer las escrituras con un valor conocido. Por ejemplo, podríamos tener un fichero cuyo contenido sean todo ceros para usarlo con el Panel de control para borrar las primeras posiciones de la memoria SRAM antes de programar nuestro procesador en la FPGA.

Para la simulación en la placa de desarrollo, es extremadamente recomendable que modifiquéis el módulo **sisa** para que incluya el driver para los 4 visores 7 segmentos que desarrollasteis en la tarea 7 de la primera sesión y mostréis constantemente en él, el valor de PC.

Anexo A: Errores típicos

Uno de los errores más frecuentes que aparecen al implementar el controlador de memoria es, no haber tenido en cuenta que en los buses bidireccionales (INOUT) pueda haber dos dispositivos intentando escribir un valor simultáneamente. Esto puede pasar simplemente porque hayamos omitido en el vhdl que debe hacer con el bus un módulo cuando no está haciendo ni lecturas ni escrituras. Hay que asegurarse que los módulos que acceden a un bus bidireccional, cuando no lo usen, pongan las señales a alta impedancia (High-Z).

Otro error frecuente es crear un grafo de estados y en cada estado asignar valores directamente a las señales de salida de la entidad. Las salidas de las entidades son por defecto *unbuffered*, aunque se puede cambiar en el *Quartus II*. Así que si en un estado no asignamos un valor a una señal no podemos asegurar que valor tendrá mientras estemos en ese estado (no se mantiene el último valor asignado). Hemos de asegurarnos que en cada estado asignemos valores a todas las señales que puedan influir en el comportamiento del componente.

Una solución a esto es declarar una señal local al módulo para ir actualizando su valor en cada estado que sea necesario y asignar directamente el valor de esa señal a la salida.

Fragmento de ejemplo de código potencialmente peligroso:

```
ARCHITECTURE rtl OF maquina_estados IS
  TYPE   tipo_estado IS (s0, s1, s2, s3);
  SIGNAL estado : tipo_estado;
BEGIN
  . . .
  -- La salida solo depende del estado actual
  PROCESS (estado)
  BEGIN
        CASE estado IS
              WHEN s0 => SRAM_WE_N <= '0';
              WHEN s1 => SRAM_WE_N <= '1';
              WHEN s2 => SRAM_WE_N <= '0';
              WHEN s3 => SRAM_WE_N <= '0';
        END CASE;
  END PROCESS;
END rtl;
```

Fragmento de código donde el valor de la señal se mantiene si en un estado no le asignamos un nuevo valor:

```
ARCHITECTURE rtl OF maquina_estados IS
  TYPE   tipo_estado IS (s0, s1, s2, s3);
  SIGNAL estado : tipo_estado;
  SIGNAL permiso_escritura : std_logic := '0';
BEGIN
  . . .
  SRAM_WE_N <= permiso_escritura;
  -- La salida solo depende del estado actual
  PROCESS (estado)
  BEGIN
        CASE estado IS
              WHEN s0 => permiso_escritura <= '0';
              WHEN s1 => permiso_escritura <= '1';
              WHEN s2 => permiso_escritura <= '0';
              WHEN s3 => permiso_escritura <= '0';
        END CASE;
  END PROCESS;
END rtl;
```