SISTEMAS ELECTRÓNICOS

PROYECTO FPGA SOKOBAN

ISAAC RODRÍGUEZ GARCÍA

SERGIO LEÓN DONCEL

ÁLVARO GARCÍA LORA

3ºGIERM

ÍNDICE

1. INTRODUCCIÓN

1.1. Dinámica de juego (básico)

Sokoban es un juego clásico de lógica rompecabezas inventado en Japón en el año 1984. Su título significa "encargado de almacén", por lo que, como se intuye, el objetivo del juego es moverse en un espacio cerrado y empujar cajas hasta las casillas designadas para ello.

Las reglas son sencillas:

- Sólo se permite empujar la caja (no permitiéndose arrastrarlas).

- Una caja sólo puede desplazarse si tiene un hueco libre detrás suya, sin permitirse empujar más de una caja a la vez.

A pesar de la simplicidad de estas pautas, se trata de un juego que puede llegar a tener niveles extremadamente complejos. Por esta razón, Sokoban se ha convertido en uno de los juegos de ingenio más populares y atractivos de nuestros tiempos.

Para el movimiento del jugador, se utilizarán los botones de la placa Basys3 T18, U17, W19, T17, que serán respectivamente los controles Up, Down, Left y Right.

1.2. Mejoras introducidas

1.2.1. Huecos

En esta versión de juego, hemos decidido implementar además huecos, los cuales también siguen un criterio fácil:

- El jugador no puede pasar por encima de ellos ni saltarlos.

- Se permite arrojar una caja sobre ellos de forma que tras ello el jugador sí pueda pasar por encima, o que la caja que intente empujar pase por esta casilla.

Su implementación, como se desarrollará más adelante en este documento, se basa en añadir codificaciones asociadas a las distintas casuísticas que pueden encontrarse (hueco vacío, con caja dentro de él, con caja y jugador encima, con caja sobre el hueco con caja).

1.2.2. Teletransporte

Se trata de un nuevo objeto, que no forma parte del juego original, pero agrega mecánicas muy interesantes para diseñar niveles. Sigue las siguientes pautas:

- No se permite teletransportar cajas (de intentarse colocar la caja encima no se permitirá el movimiento), su uso queda restringido al jugador.

- El teletransporte es inmediato, en el momento que el jugador se coloque encima de la casilla de teletransporte, su posición se actualizará a la casilla del otro teletransporte que exista en el nivel (no se contempla que haya más de 2 teletransportes en un mismo nivel).

De nuevo, esta mejora se lleva a cabo agregando codificaciones que contemplen los casos posibles (casilla de teletransporte sola, con jugador encima). Más

adelante se explicará con detalle cómo en el gestor de juego se recorre la memoria de tablero en busca del teletransporte a donde se moverá el jugador.

1.2.3. Temporización

Se ha decidido informar al jugador del tiempo que tarda en completar el juego, mostrando por pantalla un temporizador (para no utilizar recursos innecesarios, y conociendo que el juego constará de 4 niveles, se considera que el tiempo máximo empleado será de 1h).

Este temporizador tendrá 2 modos de funcionamiento según la dificultad escogida: incremental o decremental cada segundo. Los detalles se comentarán más adelante.

1.2.4. Contador de movimientos

Se mostrará en pantalla el número de pasos que el jugador utiliza para pasar de ronda, reseteándose este contador cuando se avance de nivel (se contempla un máximo de 100 movimientos).

1.2.5. Modos de dificultad

En el inicio del juego, existe una pantalla inicial que servirá para la elección del modo de juego:

- *Modo 0*: Dificultad normal. Pulsar el botón de reset reinicia el nivel que se está jugando. No existe límite de tiempo para terminar la ronda, por lo que el temporizador funcionará en modo incremental.

- *Modo 1*: Dificultad difícil. Pulsar el botón de reset reinicia el juego, por lo que se debe comenzar desde la fase inicial. Existe un límite de tiempo de 1:30 minutos, por lo que el temporizador funcionará en modo decremental. Cuando se agote el tiempo, se reiniciará el juego comenzando por la primera fase.

Para elegir el modo, se utilizará el switch V17. La posición inferior se corresponde al modo 0 y la superar al 1.

* No se permite cambiar la dificultad a mitad de juego.

1.2.6. Ayuda (solución automática)

Lo primero de todo, es tener en cuenta que su uso sólo estará permitido cuando se juegue en dificultad normal. Para activarlo, se deberá colocar en la posición superior al switch V16 y posteriormente pulsar reset.

Esta ayuda consiste en el movimiento automático del jugador, de forma que se muestre el camino más corto y sencillo a recorrer para resolver la fase por la que se encuentre el usuario.

Tener en cuenta que sólo sirve como ayuda, ya que, tras terminar de mostrarse la solución, el usuario deberá completar por sus propios medios la fase.

1.2.7. Mensajes por pantalla

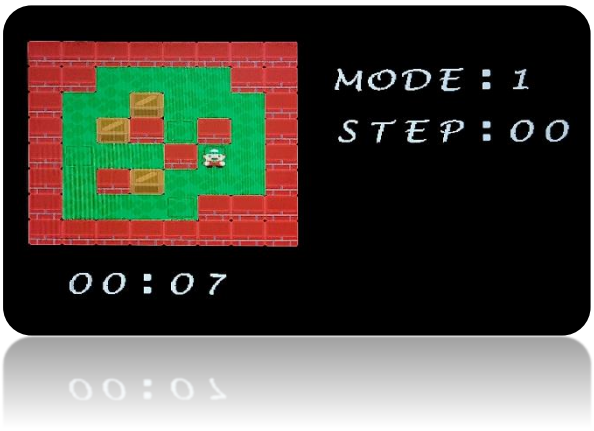
Además de dibujar el tablero de juego en la esquina superior izquierda, también contemplamos la posibilidad de mostrar mensajes en este mismo espacio. Para ello, creamos una nueva memoria de *mensajes* que contiene tres codificaciones correspondientes a estos mensajes que vamos a enseñar durante el juego:

- *FASE COMPLETA*: este mensaje se mostrará al terminar una fase, y permanecerá en pantalla durante 2.68 segundos.
- *JUEGO COMPLETO*: el mensaje será enviado al terminar la última fase, en nuestro caso, la cuarta, y por tanto, haber terminado el juego. También se verá durante 2.68 segundos.
- *SELECT MODE*: al comenzar el juego nos aparecerá este mensaje a modo de pantalla de start, indicándonos que debemos elegir el modo de juego. Se mostrará en pantalla hasta que pulsemos un botón para comenzar el juego.

1.3. Diseño de niveles

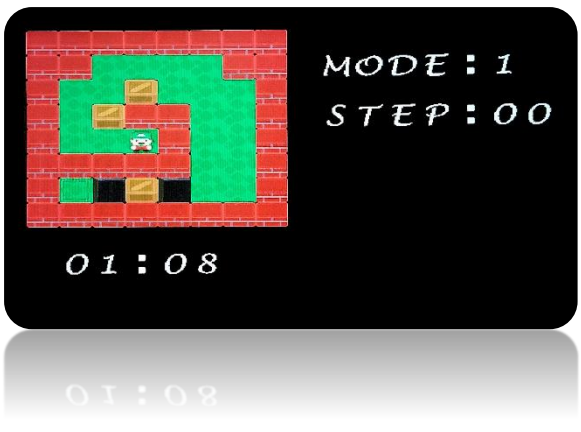
El juego ha sido diseñado en 4 niveles, de forma escalonada, donde los primeros son más fáciles y luego se van introduciendo nuevos objetos y complejidades.

- Nivel 1:

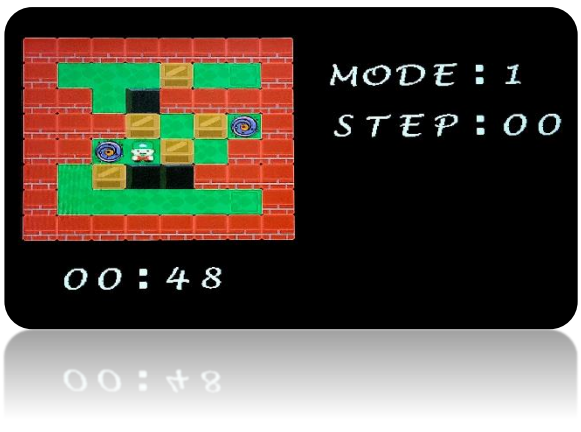


- Nivel 2:

- Nivel 3:



- Nivel 4:

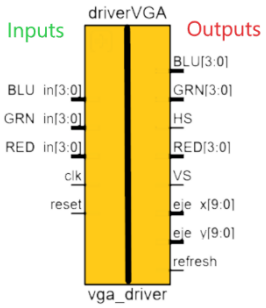


2. COMPONENTES

2.1. Driver VGA

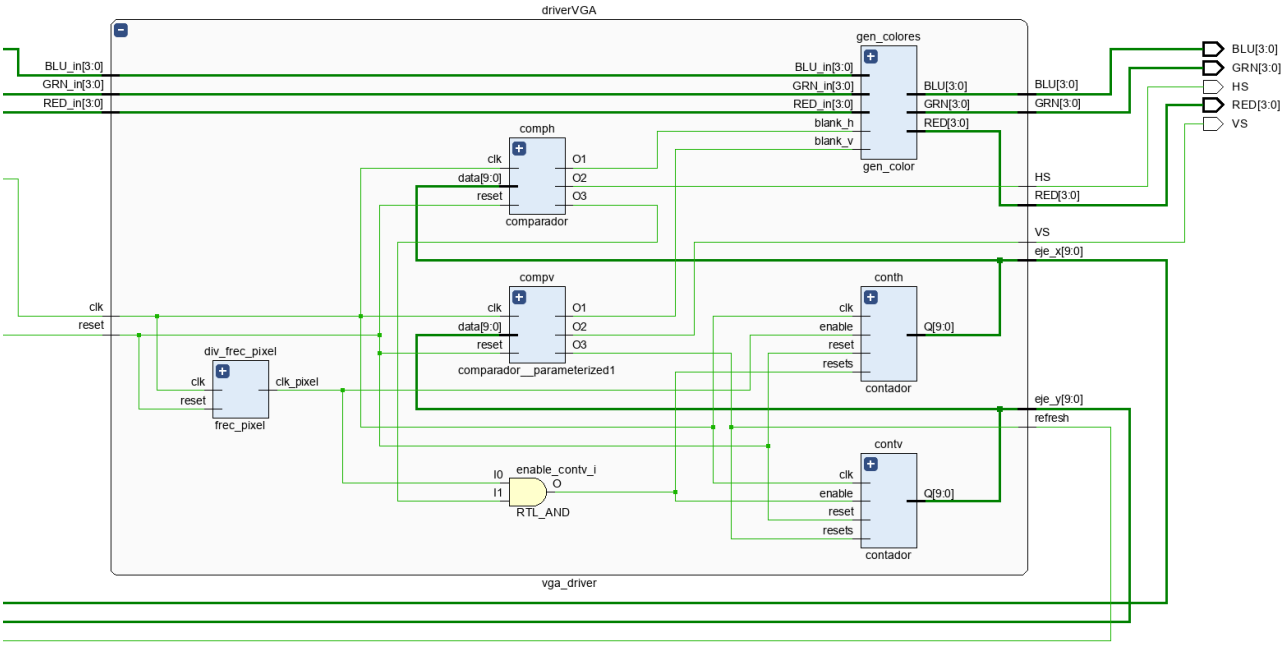
- *Descripción*: El objetivo de este bloque es el de controlar la pantalla de un monitor en protocolo VGA a 60 Hz, con resolución 640x480 píxeles y con una gama de 4096 colores.
- *Entidad*:

```
entity vga_driver is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           RED_in : in STD_LOGIC_VECTOR (3 downto 0);
           GRN_in : in STD_LOGIC_VECTOR (3 downto 0);
           BLU_in : in STD_LOGIC_VECTOR (3 downto 0);
           eje_x: out STD_LOGIC_VECTOR (9 downto 0);
           eje_y:out STD_LOGIC_VECTOR (9 downto 0);
           refresh : out STD_LOGIC;
           HS : out STD_LOGIC;
           VS : out STD_LOGIC;
           RED : out STD_LOGIC_VECTOR (3 downto 0);
           GRN : out STD_LOGIC_VECTOR (3 downto 0);
           BLU : out STD_LOGIC_VECTOR (3 downto 0));
end vga_driver;
```



- *Descripción de puertos:*
  - *RED_in, GRN_in, BLU_in*: colores a representar en el píxel seleccionado de la pantalla según el bloque *Dibuja*.
  - *eje_x, eje_y*: indican el píxel que se está dibujando en pantalla.
  - *refresh*: indica cuándo la pantalla ha sido actualizada por completo, es decir, cuando se llega a la última fila y columna de la matriz de píxeles de la pantalla.
  - *HS, VS:* pulsos de sincronismos horizontal y vertical.
  - *RED, GRN, BLU*: señales analógicas que regularán el color del píxel a modificar.
- *Descripción de arquitectura:*
  Se explicará brevemente ya que es fundamentalmente idéntico a la práctica previamente realizada.
  Este bloque funcionará como una especie de top que instancie distintos componentes para obtener el resultado deseado:
  - *Divisor frecuencia*: para obtener una frecuencia de píxel de 25 MHz, debemos dividir la frecuencia de reloj disponible (100 MHz) entre cuatro, utilizando un contador.
  - *Contador*: miden los ejes horizontal y vertical.
  - *Comparador*: en función de los contadores, indica si se está fuera de pantalla, cuando se activa las señales de sincronismo horizontal y vertical, cuando se salta de línea o termina la pantalla.
  - *Gen_colores*: saca como salida los colores de entrada recibidos desde *Dibuja*, excepto cuando se está fuera de pantalla, donde envía todo 0.

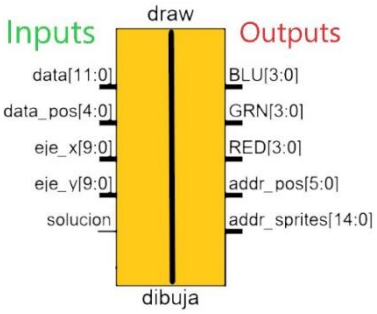
* Se ha decidido sacar como salida *O3*, con el nombre de *refresh*, para indicar el fin de pantalla y enviarlo posteriormente a *gestor_juego*.



2.2. Dibuja

- *Descripción*: La finalidad del bloque es la de sacar como salida los colores del píxel indicado. Para ello, leerá datos de las memorias, en función de lo que decida *gestor_sprites*, y dependiendo de la posición de pantalla a dibujar decidirá el valor de sus salidas.
- *Entidad*:

```
entity dibuja is
    Port ( eje_x : in STD_LOGIC_VECTOR (9 downto 0);
           eje_y : in STD_LOGIC_VECTOR (9 downto 0);
           solucion: in STD_LOGIC;
           data: in STD_LOGIC_VECTOR (11 downto 0);
           data_pos: in STD_LOGIC_VECTOR (4 downto 0);
           RED : out STD_LOGIC_VECTOR (3 downto 0);
           GRN : out STD_LOGIC_VECTOR (3 downto 0);
           BLU : out STD_LOGIC_VECTOR (3 downto 0);
           addr_sprites: out STD_LOGIC_VECTOR (14 downto 0);
           addr_pos: out STD_LOGIC_VECTOR (5 downto 0)
           );
end dibuja;
```

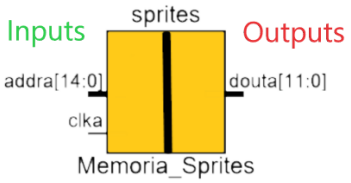


- *Descripción de puertos:*
  - *solucion*: indica si se ha activado la ayuda de resolución automática.
  - *data*: se corresponde con el color del píxel de la imagen.
  - *data_pos*: indica la posición de la memoria que deseamos leer.
  - *addr_sprites*: indica la dirección de memoria a leer.

- *Descripción de arquitectura:* Por defecto, las salidas *RED, GRN y BLU* estarán a 0. En función de las coordenadas x e y de la pantalla que reciba, decide si mostrar los colores de *data* (para tablero, contador de tiempo, modo de dificultad y contador de movimientos), sólo sus 8 bits más significativos (para pintar de amarillo el indicador de solución) o todo en negro (a 0).

2.3. Memorias

2.3.1. Sprites



- *Descripción*: Almacena las imágenes asociadas a los distintos objetos usados en el juego. Cada imagen ocupa un total de 1024 posiciones de memoria (32 x 32 pixeles).

- *Características:*

Tipo de memoria	*Profundidad de memoria*	*Anchura de memoria*	*Longitud de Direcciones*	*Fichero. coe asociado*
Single Port ROM	22528 (1024 x 22)	12 bits	15 bits	memoria_sprites.coe

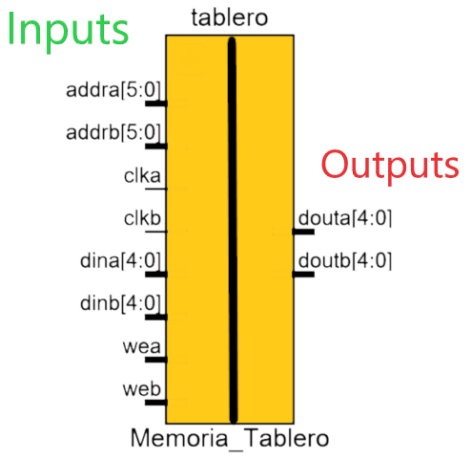
- *Codificación empleada:*

Código de palabra	RRRRGGGGBBBB

- *Direcciones:*

addr (14 downto 10)	Código del objeto
addr (9 downto 5)	eje_y (4 downto 0) *(Fila dentro de la casilla)*
addr (4 downto 0)	eje_x (4 downto 0) *(Columna dentro de la casilla)*

2.3.2. Tablero



- *Descripción:* Almacena los distintos objetos en su correcta disposición actual en el tablero de juego. Se trata de una memoria dinámica que va cambiando su contenido durante el desarrollo de las distintas fases del juego. Cuando se produce un cambio de fase su contenido cambia a la correspondiente gracias a la copia de los mapas (en su estado inicial) que se encuentran en la memoria de *Fases*.

- *Características:*

Tipo de memoria	*Profundidad de memoria*	*Anchura de memoria*	*Longitud de Direcciones*	*Fichero. coe asociado*
True Dual Port RAM	64 (8 x 8 casillas)	5 bits	6 bits	No necesita fichero de inicialización

- *Puertos:*
  - *Puerto A:* Lectura y escritura. Interacción con el bloque de Gestor de Juego. Mediante este puerto la memoria cambia de contenido para guardar el tablero correcto durante el desarrollo del juego. El encargado de hacerlo es la máquina de estados.
  - *Puerto B:* Sólo lectura. Interacción con el bloque *Dibuja*. Se lee la información contenida en la memoria para procesarla y representar las imágenes asociadas a cada objeto.

- *Codificación empleada:*

  Al igual que la memoria de Fases la codificación de las palabras de memoria se ha realizado en decimal y es la siguiente:

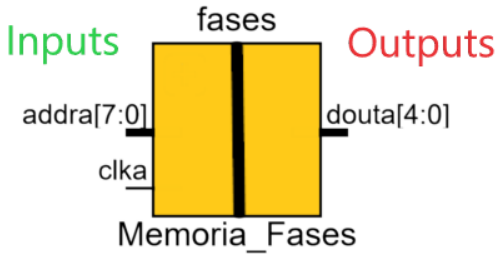
0	Jugador vista frontal
1	Jugador vista trasera
2	Jugador vista lateral derecha
3	Jugador vista lateral izquierda
4	Fondo
5	Casilla objetivo
6	Caja
7	Muro
8	Caja con objetivo
9	Hueco
10	Hueco con caja
11	Hueco con caja+ Caja encima
12	Hueco con caja+ Player frontal
13	Hueco con caja+ Player trasera
14	Hueco con caja+ Player derecha
15	Hueco con caja+ Player izquierda
16	Teletransporte
17	Jugador vista frontal + Objetivo
18	Jugador vista trasera + Objetivo
19	Jugador vista lateral derecha + Objetivo
20	Jugador vista lateral izquierda + Objetivo
21	Teletransporte + Player

- *Direcciones:*

addrb (5 downto 3)	eje_y (7 downto 5) *(Fila del tablero)*
addrb (2 downto 0)	eje_x (7 downto 5) *(Columna del tablero)*
addra	Asociado a la posición y movimientos del jugador por el tablero

2.3.3. Fases

- *Descripción:* Contiene los distintos mapas asociados a las diferentes fases del juego. Cada mapa es cargado en la memoria *Tablero* cuando se da una transición entre fases en el juego.



- *Características:*

Tipo de memoria	*Profundidad de memoria*	*Anchura de memoria*	*Longitud de Direcciones*	*Fichero. coe asociado*
Single Port ROM	256 (64 x 4 fases)	5 bits	8 bits	memoria_fases.coe

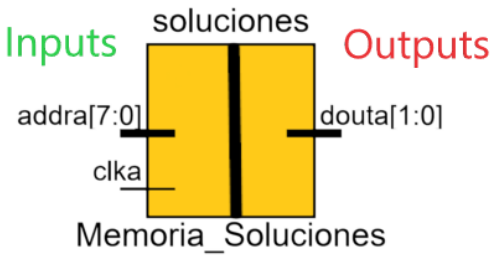
- *Codificación empleada:*

0	Jugador vista frontal
1	Jugador vista trasera
2	Jugador vista lateral derecha
3	Jugador vista lateral izquierda
4	Fondo
5	Casilla objetivo
6	Caja
7	Muro
8	Caja con objetivo
9	Hueco
10	Hueco con caja
11	Hueco con caja+ Caja encima
12	Hueco con caja+ Player frontal
13	Hueco con caja+ Player trasera
14	Hueco con caja+ Player derecha
15	Hueco con caja+ Player izquierda
16	Teletransporte
17	Jugador vista frontal + Objetivo
18	Jugador vista trasera + Objetivo
19	Jugador vista lateral derecha + Objetivo
20	Jugador vista lateral izquierda + Objetivo
21	Teletransporte + Player

- *Direcciones:*

addra (7 downto 6)	Fase (0,1,2,3) (2 bits = codificación de 4 fases)
addra (5 downto 3)	eje_y (7 downto 5) *(Fila del tablero)*
addra (2 downto 0)	eje_x (7 downto 5) *(Columna del tablero)*

2.3.4. Soluciones



- *Descripción:* Almacena las secuencias de movimientos necesarias para resolver automáticamente cada una de las fases.

- *Características:*

Tipo de memoria	*Profundidad de memoria*	*Anchura de memoria*	*Longitud de Direcciones*	*Fichero. coe asociado*
Single Port ROM	256	2 bits	8 bits	memoria_solu.coe

- *Codificación empleada:*

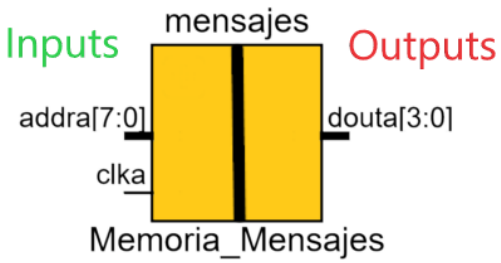
0	Arriba
1	Derecha
2	Abajo
3	Izquierda

- *Direcciones:*

addra (7 downto 6)	Fase (0,1,2,3) (2 bits = codificación de 4 fases)
addra (5 downto 3)	eje_y (7 downto 5) *(Fila del tablero)*
addra (2 downto 0)	eje_x (7 downto 5) *(Columna del tablero)*

2.3.5. Mensajes

- *Descripción:* Almacena los mensajes (conjuntos de secuencias de letras codificadas en decimal) que se mostrarán en el tablero de juego cuando se supere cada una de las fases ("FASE COMPLETA"), el juego ("JUEGO COMPLETO") y pantalla inicial de selección de modo ("SELECT MODE").



- *Características:*

Tipo de memoria	*Profundidad de memoria*	*Anchura de memoria*	*Longitud de Direcciones*	*Fichero. coe asociado*
Single Port ROM	192 (3 mensajes x8x8)	4 bits	8 bits	memoria_mensajes.coe

- *Codificación empleada:*

0	CASILLA NEGRA
1	A
2	C
3	D
4	E
5	F
6	G
7	J
8	L
9	M
10	O
11	P
12	S
13	T
14	U
15	CASILLA BLANCA

Mensajes:

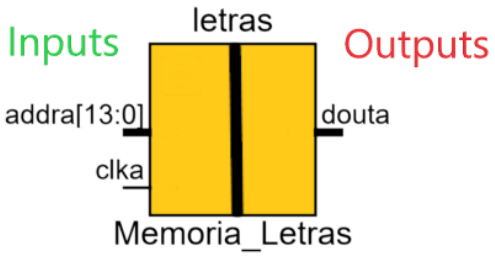
1er mensaje → FASE COMPLETA
2º mensaje → JUEGO COMPLETO
3er mensaje → SELECT MODE

- *Direcciones:*

addra (7)	Indicador de modo selección (select_mode) 0 → Desarrollo juego 1 → Pantalla de start
addra (6)	Indicador de fin de juego *(fin_juego)* 0 → Desarrollo juego 1 → Juego completado (finalizada última de las fases)
addra (5 downto 3)	eje_y (7 downto 5) *(Fila del tablero)*
addra (2 downto 0)	eje_x (7 downto 5) *(Columna del tablero)*

2.3.6. Letras

- *Descripción:* Almacena las imágenes asociadas a las distintas letras usadas en los mensajes. Cada imagen ocupa un total de 1024 posiciones de memoria (32 x 32 pixeles).



- *Características:*

Tipo de memoria	*Profundidad de memoria*	*Anchura de memoria*	*Longitud de Direcciones*	*Fichero. coe asociado*
Single Port ROM	16384 (16 x 1024)	1 bit	14 bits	memoria_letras.coe

- *Codificación empleada:*

Código de palabra	0 → NEGRO // 1 → BLANCO

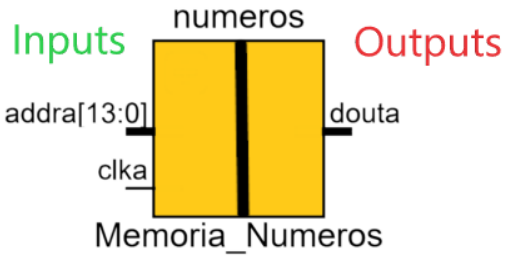
(Debido a que todos los mensajes son en blanco y negro, las imágenes de las letras pueden codificarse en la memoria con 0 y 1 ahorrando de esta forma almacenamiento. Posteriormente una vez leído el dato de la memoria se asocia el 0 con el color negro y 1 con el blanco)

- *Direcciones:*

addr (14 downto 10)	Código de la letra
addr (9 downto 5)	eje_y (4 downto 0) *(Fila dentro de la casilla)*
addr (4 downto 0)	eje_x (4 downto 0) *(Columna dentro de la casilla)*

2.3.7. Números

- *Descripción:* Contiene las imágenes asociadas a los números usados en los contadores, tanto de temporización como de movimientos.



- *Características:*

Tipo de memoria	*Profundidad de memoria*	*Anchura de memoria*	*Longitud de Direcciones*	*Fichero. coe asociado*
Single Port ROM	11264 (1024 x 11)	1 bit	14 bits	memoria_numeros.coe

- *Codificación empleada:*

Código de palabra	0 → NEGRO // 1 → BLANCO

(Debido a que todos los números son en blanco y negro, sus imágenes pueden codificarse en la memoria con 0 y 1, al igual que ocurre con las letras)

- *Direcciones:*

addr (14 downto 10)	Número en binario
addr (9 downto 5)	eje_y (4 downto 0) *(Fila dentro de la casilla)*
addr (4 downto 0)	eje_x (4 downto 0) *(Columna dentro de la casilla)*

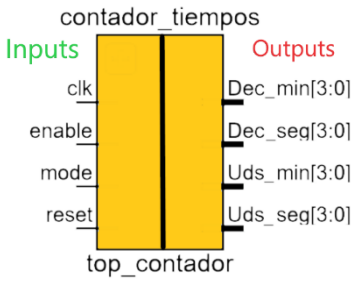
NOTA: Los números se encuentran ordenados en la memoria (por tanto en el fichero de inicialización .coe) empezando por el 0 hasta el 9 y por último se encuentra el carácter ':' usado también para los contadores)

2.4. Contador de tiempos

- *Descripción*: Como su propio nombre indica, se trata de un contador de decenas y unidades que cambia su valor cada segundo. En función del modo, servirá para contar el tiempo de juego o calcular el tiempo restante que tiene el jugador para completar el juego en difícil, tal y como se comentó en puntos anteriores.

- *Entidad*:

```
entity top_contador is
    Port ( clk : in STD_LOGIC;
           enable: in STD_LOGIC;
           reset : in STD_LOGIC;
           mode: in STD_LOGIC;
           Dec_min : out STD_LOGIC_VECTOR (3 downto 0);
           Uds_min : out STD_LOGIC_VECTOR (3 downto 0);
           Dec_seg : out STD_LOGIC_VECTOR (3 downto 0);
           Uds_seg : out STD_LOGIC_VECTOR (3 downto 0));
end top_contador;
```



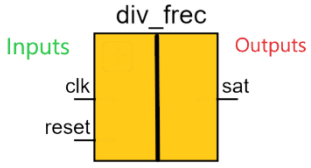
- *Descripción de puertos:*
  - *mode*: nos informa de la dificultad en la que se está jugando.
  - *Dec_min, uds_min, dec_seg, uds_seg*: los valores asociados a los contadores de decenas y unidades de minutos y segundos respectivamente.

- *Descripción de arquitectura:* Instanciamos los 4 contadores necesarios, interconectando la saturación de uno con el *enable* del siguiente.

* Observar que el reset de *div_frec* es el *enable* de entrada, ya que como se verá en el top del proyecto, éste se activa al finalizar una fase o estar en la pantalla de selección de modo, por lo que en ese momento no se debe contar. Es decir, es un enable de lógica invertida.

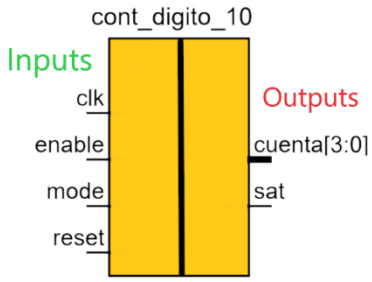
2.4.1. Divisor de frecuencia

Ya que la frecuencia de reloj de la que se dispone es de 10 MHz, eso significa que en un segundo es capaz de contar 10 millones. Por tanto, realizando un contador hasta esa cantidad, podemos generar un pulso cada segundo gracias a la señal de saturación.



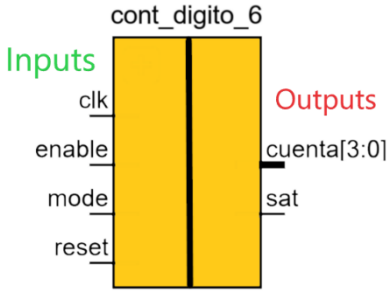
2.4.2. Contador módulo 10

Nos servirá para la segunda cifra de minutos y segundos del temporizador. Con la señal *mode* seleccionamos si queremos una cuenta ascendente o descendente.



2.4.3. Contador módulo 6

Se utilizará para la primera cifra de minutos y segundos del temporizador. Con la señal *mode* seleccionamos si queremos una cuenta ascendente o descendente. Además, gracias a la parametrización N, en modo descendente la cuenta se iniciará a 1 para minutos y 3 para segundos.



2.5. Contador de movimientos

Se implementará con 2 instancias de contador de módulo 10, de forma que se pueda contar hasta 99 movimientos.

```
cont_mov_uds: contador_movimientos
port map(
        clk => clk,
        enable => cuenta_mov_s,
        sat => sat_uds_s,
        reset => reset_mov,
        cuenta => uds_mov_s
);

cont_mov_dec: contador_movimientos
port map(
        clk => clk,
        enable => sat_uds_s,
        sat => OPEN,
        reset => reset_mov,
        cuenta => dec_mov_s
);
```

Su funcionamiento es sencillo. Cada ciclo de reloj en el que el *enable* esté a '1', se incrementará en uno la cuenta.

Como se puede ver, el *enable* se rige según la señal *cuenta_mov_s*, por lo que a cada movimiento del jugador se incrementa. Como la señal sólo durará un ciclo de reloj en alto (pulso) no se produce problema incrementando de más.

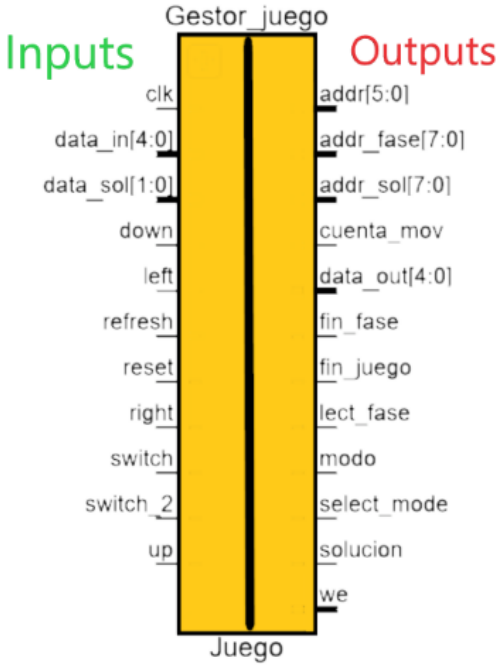
2.6. Gestor de Juego (Máq. estados)

- *Descripción:*

Se trata del bloque principal de nuestro proyecto, y el que se encarga de hacer funcionar todo el juego. Se estructura principalmente como una máquina de estados con un *reset* síncrono.

- *Entidad:*

```
entity Juego is
    Port ( up : in STD_LOGIC;
           down : in STD_LOGIC;
           right : in STD_LOGIC;
           left : in STD_LOGIC;
           refresh: in STD_LOGIC;
           clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           switch: in STD_LOGIC;
           switch_2: in STD_LOGIC;
           data_in : in STD_LOGIC_VECTOR (4 downto 0);
           data_sol : in STD_LOGIC_VECTOR (1 downto 0);
           modo: out STD_LOGIC;
           solucion: out STD_LOGIC;
           we : out STD_LOGIC_VECTOR (0 downto 0);
           addr_fase : out STD_LOGIC_VECTOR (7 downto 0);
           addr_sol : out STD_LOGIC_VECTOR (7 downto 0);
           addr : out STD_LOGIC_VECTOR (5 downto 0);
           lect_fase: out STD_LOGIC;
           fin_fase : out STD_LOGIC;
           fin_juego : out STD_LOGIC;
           cuenta_mov: out STD_LOGIC;
           select_mode: out STD_LOGIC;
           data_out : out STD_LOGIC_VECTOR (4 downto 0));
end Juego;
```



- *Descripción de los puertos:*

up, down, right, left	Botones de la FPGA que son usados para mover el jugador
refresh	Señal de entrada que emite un pulso cada vez que se termina de pintar la pantalla
clk	Señal de reloj de la placa
reset	Botón central de la FPGA que se usará como *reset*
switch	Primer interruptor de la FPGA, indicará el modo de juego
switch_2	Segundo interruptor de la FPGA, activa el modo solución
data_in	Entrada que nos proporciona los datos de la memoria de mensajes o del tablero según corresponda
data_sol	Entrada de los datos de la memoria de soluciones
modo	Salida que indica el modo de juego
solución	Señal que se toma el valor '1' cuando está activado el modo de solución
we	Señal de *write enable*, indica si estamos leyendo ('0') o escribiendo ('1') sobre la memoria del tablero
addr_fase	Señal que indica la dirección a leer de la memoria de fases

addr_sol	Señal que indica la dirección a leer de la memoria de soluciones
addr	Señal que indica la dirección a leer/escribir de la memoria del tablero
lect_fase	Salida que indica si se está leyendo de la memoria de fases
fin_fase	Señal que se activa a '1' cuando estamos en la pantalla de fin de fase
fin_juego	Señal que se activa a '1' cuando estamos en la pantalla de fin de juego
cuenta_mov	Señal que emite un pulso cada vez que se realiza un nuevo movimiento del jugador
select_mode	Señal que se activa a '1' cuando estamos en la pantalla de selección de modo
data_out	Valor de escritura en la memoria del tablero

- *Descripción de la arquitectura:*

Como ya se comentó previamente, el gestor de juego funciona principalmente como una máquina de estados. Para ello, se crea un tipo de variable *tipo_estado* que puede tomar el valor de todos los estados de esta, además de crear una variable estado que indica el estado en el que nos encontramos.

Además de *tipo_estado*, también hemos creado el tipo de variable *tipo_mov*, que puede ser arriba, abajo, izquierda o derecha. La señal *mov* será declarada de este tipo, e indicará el próximo movimiento que pretende realizar el jugador al pulsar cualquier botón.

Por otra parte, el diseño del bloque se realizará en dos procesos, por lo que a cada señal le corresponde otra que denominaremos añadiéndole delante 'p_', de forma que en la máquina de estados le daremos valores a estas últimas, y cuando se dé un flanco de subida del reloj cada señal tomará el valor de su análoga '*p_señal*', es decir, se actualizarán de forma síncrona.

Del mismo modo se usan el resto de señales creadas:

pos_x, pos_y	Almacenan la posición actual del jugador en el tablero, para así no tener que recorrer la memoria buscándolo cada vez que se realice un movimiento
addr_s	Se usa para cumplir la función de la entrada *addr*, pero al ser de tipo *unsigned* permite incrementarla
addr_copia	También es una dirección, que en este caso es usada para leer de la memoria de fases y copiar en el tablero al inicio de estas
fase	Indica en la fase que nos encontramos
cont	Se usa para determinar un tiempo de espera en ciertos estados
cont_mov_sol	Almacena el número de movimiento que hay que leer de cada fase cuando estamos en el modo solución
mode, solution_mode	De tipo booleanas, son las análogas a las salidas modo y solución de este bloque

Finalmente, tenemos 9 'flags', declaradas también como booleanas, que se activarán en función de que se produzcan determinados eventos, y servirán para decidir a qué estado avanzar:

flag_0	El jugador está sobre la casilla objetivo
flag_1	Se ha validado el movimiento de la caja, por lo que habrá que pasar por el estado de *escribir_caja*
flag_2	Indica que la caja se va a colocar sobre la casilla objetivo
flag_3	Indica que se va a mover una caja que está sobre una casilla objetivo
flag_4	Indica que el jugador se va a colocar sobre un hueco con caja
flag_5	Se va a colocar una caja sobre un hueco vacío
flag_6	Se va a colocar una caja sobre un hueco con caja
flag_7	Indica que la caja que se quiere mover está sobre un hueco con caja
flag_8	El jugador se dirige hacia un teletransporte

En cuanto a la máquina de estados, lo primero que tenemos en cuenta es el *reset*, que se ejecuta de forma síncrona. Cuando se pulsa este botón, se inicializan a cero todas las señales de dirección, posición, contadores, etc. Para determinar el próximo estado al que iremos, tendremos en cuenta que si estamos en la pantalla de selección de modo debemos quedarnos en ese estado, mientras que en caso contrario accederemos a *leer_fase*. Por otro lado, si estamos en el modo difícil comenzaremos de nuevo por la primera fase, mientras que si estamos en modo fácil se resetea la fase, pero permanecemos en esta misma, y si está activo el segundo interruptor, se activa el modo solución.

En caso de no detectarse el *reset*, pasamos a la máquina de estados. Lo primero que haremos es darle valores por defecto a todas las señales y salidas, para así evitar que se generen 'latches' sin tener que darle un valor a cada una en cada estado. Posteriormente, con una sentencia del tipo case, definimos todos los estados:

- *start*: en la pantalla de start o selección de modo, leemos el primer interruptor para determinar el modo de juego, y esperaremos a que se pulse cualquiera de los cuatro botones de dirección y se dé un *refresh* para pasar al siguiente estado.
- *espera_boton_start*: en este estado esperamos a que se suelte el botón pulsado, esperando además otro *refresh* para evitar rebotes de este. Además, seguimos leyendo el valor del primer switch por si cambiamos este mientras que mantenemos pulsado algún botón.
- *lee_fase*: leemos la dirección adecuada de la memoria de fases, siendo sus dos primeros bits correspondientes a la fase y los otros seis a la posición del tablero que queremos leer.
- *espera_copia*: espera de un ciclo de reloj para tener disponible el valor leído anteriormente.
- *copia_fase*: copiamos el valor leído en la dirección correspondiente al tablero, y aumentamos *addr_copia* para recorrer el tablero entero. Además, si el dato leído fuera el jugador, guardamos esa dirección del

tablero como posición del jugador en las señales *pos_x*, *pos_y*. Cuando se termina de recorrer el tablero, pasamos a reposo.

- *reposo*: si no estamos en modo solución, esperaremos a que se pulse algún botón y se dé un pulso de *refresh*; si estamos en este modo, esperaremos medio segundo mediante un contador (ya que si no se movería demasiado rápido) y pasaremos a leer el movimiento de la memoria de soluciones.
- *espera_boton*: al haber pulsado un botón, esperamos a que este se suelte, actualizando el movimiento deseado según el botón pulsado. Además, lo hacemos esperando de nuevo un pulso de *refresh*, ya que en algunos casos puede que los rebotes duren más del tiempo de pintado de la pantalla, por lo que podrían darse movimientos no deseados.
- *lee_mov_sol*: leemos y actualizamos el movimiento de la memoria de soluciones, incrementando el contador de movimientos de esta.
- *lee_destino*: leemos lo que hay en la dirección de destino del jugador según el movimiento deseado.
- *espera_lectura_sig*: espera de un ciclo de reloj para tener disponible el valor leído de la posición de destino.
- *validar_mov*: conociendo lo que hay en la casilla de destino, determinamos si se puede ejecutar o no el movimiento:
    - Muro o hueco vacío: no podemos movernos, vuelta a reposo.
    - Fondo: pasamos a escribir sobre la posición actual.
    - Casilla objetivo: activamos la *flag_0* y pasamos a escribir sobre la posición actual.
    - Hueco con caja: activamos la *flag_4* y pasamos a escribir sobre la posición actual.
    - Caja sobre hueco con caja: activamos la *flag_7* y pasamos a leer la posición de destino de la caja.
    - Caja: pasamos a leer la posición de destino de la caja.
    - Caja sobre objetivo: activamos la *flag_3* y pasamos a leer la posición de destino de la caja.
    - Teleport: activamos la *flag_8* y pasamos a escribir sobre la posición actual.
- *escribir_actual*: escribir sobre la posición actual del jugador lo que corresponda, según si el jugador estaba sobre la casilla objetivo, sobre hueco con caja o sobre un teleport. También actualizamos la posición del jugador según el movimiento para pintar posteriormente este en su nueva posición. Si el jugador se dirige hacia un teleport (*flag_8* activada) habrá que recorrer la memoria para buscar el otro teleport; si no, pasamos directamente a escribir el jugador.
- *inicializar_addr_teleport*: inicialización de la dirección para recorrer la memoria.
- *busca_teleport*: leemos lo que hay en el tablero, y si es un teleport que no está en la posición actual del jugador, guardamos esa dirección en la posición y pasamos a escribir el jugador. Si no, seguimos incrementando la dirección de lectura.

- *incr_addr_teleport*: incrementamos la dirección de búsqueda de teleport.
- *espera_incr_teleport_1, espera_teleport_2*: esperamos dos ciclos de reloj, uno para que se actualice la dirección y otro para tener disponible el valor de lectura.
- *escribir_player*: representamos el jugador en su nueva posición:
    - Si están activas *flag_4* o *flag_7*, pintamos el jugador sobre hueco con caja, mirando hacia la dirección adecuada según el movimiento.
    - Si están activas *flag_0* o *flag_3*, pintamos el jugador sobre el objetivo, mirando hacia la dirección adecuada según el movimiento.
    - Si está activa la *flag_8*, pintamos el jugador sobre el teleport, mirando de frente.
    - Si no hay ninguna de las 'flags' anteriores, se pinta el jugador sobre el fondo, mirando a la dirección adecuada según el movimiento.

    Además, si está activa la *flag_1* habrá que pasar a escribir la caja, y si no, volvemos a reposo.

- *lee_destino_caja*: leemos la posición de destino de la caja, es decir, dos posiciones más allá del jugador.
- *espera_lectura_sig_caja*: espera de un ciclo de reloj para tener disponible el valor leído del tablero en la posición de destino de la caja.
- *validar_mov_caja*: comprobamos si se puede realizar el movimiento de la caja, según el valor leído:
    - Muro o teleport: desactivamos la *flag_3* si estaba activa y volvemos a reposo.
    - Fondo: se activa la *flag_1* y pasamos a *escribir_hueco_caja* si estaba activa la *flag_4*, y a *escribir_actual* en caso contrario.
    - Casilla objetivo: lo mismo que para el fondo pero activando también la *flag_2*.
    - Hueco vacío: lo mismo que para el fondo pero activando también la *flag_5*.
    - Hueco con caja: lo mismo que para el fondo pero activando también la *flag_6*.
    - Caja, caja sobre hueco con caja o caja sobre objetivo: desactivamos la *flag_3* y *flag_4* si estaban activas y volvemos a reposo.
- *escribir_caja*: escribimos en la casilla correspondiente la caja que se ha movido:
    - Si está activa la *flag_2*, escribimos una caja sobre objetivo y desactivamos esta 'flag'.
    - Si está activa la *flag_5*, escribimos una caja sobre hueco vacío y desactivamos esta 'flag'.
    - Si está activa la *flag_6*, escribimos una caja sobre hueco con caja y desactivamos esta 'flag'.
    - En caso contrario, escribimos la caja sobre el fondo.

Además, en todos los casos desactivamos la *flag_1* y como hemos movido una caja, tenemos que inicializar la dirección para recorrer la memoria y determinar si hemos terminado la fase.

- *escribir hueco caja*: dibuja un hueco con caja donde antes se encontraba el jugador, desactivando la *flag_4* y actualizando la posición del jugador según el movimiento.
- *iniciallizar_addr*: inicializamos la dirección para recorrer el tablero después de mover una caja.
- *comprobar_fin*: primero miramos si la *flag_0* está activa, ya que en ese caso el jugador estará sobre la casilla objetivo y en ningún caso se habrá terminado la fase, por lo que volveremos a reposo. Si no lo está, leeremos el valor del tablero, y si este es una casilla objetivo tampoco habremos terminado la fase, volviendo a reposo. Recorreremos la memoria entera, para lo cual pasaremos por el estado de *incr_addr*, y si ha llegado al final sin encontrar una casilla objetivo, habremos terminado la fase.
- *incr_addr*: incrementa la dirección de lectura del tablero para recorrer este entero, volviendo a *comprobar_fin*.
- *fin*: si se ha terminado la fase, llegamos a este estado, en el que se activa la señal *fin_fase* y se pone a cero el contador de movimientos de solución por si hemos llegado aquí en ese modo. Si estamos en la fase final, activaremos también *fin_juego*. Por otra parte, esperaremos 2.68 segundos con un contador para mostrar el mensaje correspondiente por pantalla, y si no estábamos en el modo solución, aumentamos en uno la fase; si estábamos en la última fase, pasamos a la pantalla de start o selección de modo, y si no, volvemos a *lee_fase*. En caso de haber hecho la fase en el modo solución, volvemos a leer y copiar la misma fase y desactivamos el modo de solución. Por último, inicializamos a cero la dirección para la lectura de fase en el estado *lee_fase*.

Terminada la máquina de estados, solo nos faltaría un proceso aparte que usaremos para convertir las señales *mode* y *solution_mode*, que eran booleanas por su mayor simplicidad de casos, a las salidas modo y solución, que son de la forma STD_LOGIC.

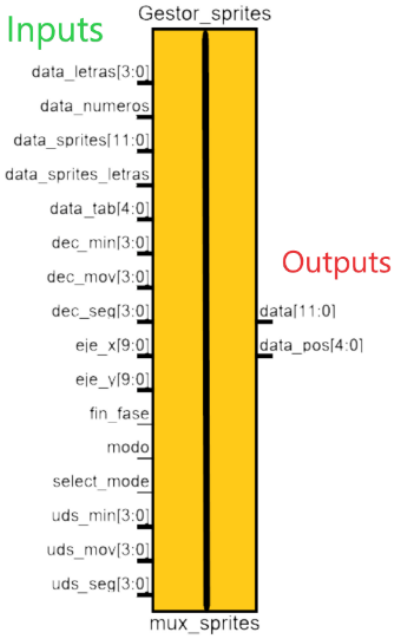
2.7. Gestor de Sprites

- *Descripción:*

Este bloque es el encargado de decidir, según el estado del juego y el lugar de la pantalla que se esté pintando, de qué memoria se deben obtener los *Sprites* que serán enviados al bloque *Dibuja*.

- *Entidad:*

```
entity mux_sprites is
    Port ( data_sprites : in STD_LOGIC_VECTOR (11 downto 0);
           data_sprites_letras : in STD_LOGIC_VECTOR (0 downto 0);
           fin_fase : in STD_LOGIC;
           data_tab : in STD_LOGIC_VECTOR (4 downto 0);
           data_letras : in STD_LOGIC_VECTOR (3 downto 0);
           eje_x : in STD_LOGIC_VECTOR (9 downto 0);
           eje_y : in STD_LOGIC_VECTOR (9 downto 0);
           data_numeros : in STD_LOGIC_VECTOR (0 downto 0);
           uds_seg : in STD_LOGIC_VECTOR (3 downto 0);
           dec_seg : in STD_LOGIC_VECTOR (3 downto 0);
           uds_min : in STD_LOGIC_VECTOR (3 downto 0);
           dec_min : in STD_LOGIC_VECTOR (3 downto 0);
           modo : in STD_LOGIC;
           select_mode : in STD_LOGIC;
           uds_mov : in STD_LOGIC_VECTOR (3 downto 0);
           dec_mov : in STD_LOGIC_VECTOR (3 downto 0);
           data : out STD_LOGIC_VECTOR (11 downto 0);
           data_pos : out STD_LOGIC_VECTOR (4 downto 0));
end mux_sprites;
```



- *Descripción de los puertos:*

data_sprites	Entrada de los datos RGB de la memoria de los sprites de las imágenes del juego
data_sprites_letras	Entrada de los datos de la memoria de las letras en formato de un bit (blanco y negro)
fin_fase	Entrada que indica que se acaba de terminar una fase
data_tab	Dato leído de la memoria del tablero
data_letras	Dato leído de la memoria de mensajes
eje_x eje_y	Entradas que indican el número de píxel de la pantalla que se está dibujando en el *driver_VGA*
data_numeros	Entrada de los datos leídos de la memoria de números en formato de un bit (blanco y negro)
uds_seg, dec_seg, uds_min, dec_min	Datos del valor de los contadores de tiempo (unidades de segundos, decenas de segundos, unidades de minutos y decenas de minutos respectivamente)
modo	Entrada que indica el modo de juego (normal o difícil)
select_mode	Entrada que indica que nos encontramos en la pantalla de start o selección de modo de juego
uds_mov dec_mov	Datos del valor de los contadores de movimientos (unidades y decenas, respectivamente)
data	Salida que irá conectada al bloque *Dibuja*, indicándole el código RGB que debe dibujar en la pantalla en cada caso
data_pos	Salida que indica la dirección de lectura de las memorias que contienen sprites, letras o números

- *Descripción de la arquitectura:*

La estructura principal de este bloque es un proceso con varias sentencias *if/elsif* que dependen de los valores de las entradas *eje_x* y *eje_y*. De este modo, le pasaremos al bloque *Dibuja* los datos RGB adecuados para pintar la pantalla. Además, para facilitar esto, a cada letra o número le corresponderá una casilla de 32x32 píxeles al igual que en el tablero.

En primer lugar, si nos encontramos en la zona de debajo del tablero, donde vamos a representar el contador de tiempo, leeremos los datos de la memoria de números, concatenándolos doce veces (ya que el dato leído será de 1 bit, y la codificación RGB es de 12), y la dirección de lectura será, respectivamente, la de las decenas de minutos, unidades de minutos, los dos puntos, decenas de segundos y unidades de segundos.

Por otro lado, si estamos en la zona derecha del tablero, distinguimos tres líneas, en las que leeremos de la memoria de letras cuando haya que escribir letras y de la memoria de números para representar estos últimos (los datos de la memoria de letras se concatenan doce veces por la misma razón que los de los números). En la cuarta fila, escribiremos **MODE:** y a continuación un **0** si estamos en el modo normal o un **1** si estamos en modo difícil. En la cuarta fila pondrá **STEP:** y un número de dos dígitos que representa el número de pasos o movimientos que se han realizado en la fase actual. En la sexta fila, finalmente, escribiremos **SOL** si nos encontramos en el modo de solución, y no se escribirá nada en caso contrario. Además, se dibujará en amarillo, para lo cual simplemente le pasamos los datos del color rojo y verde y dejaremos a 0 los datos del color azul.

Por último, para el resto de la pantalla distinguiremos dos casos, aunque realmente solo usaremos esos datos en las 64 casillas que ocupa el tablero. Si están activas las entradas de fin de fase o selección de modo, tomaremos los datos de la memoria de letras, usando para ello la memoria de mensajes, y mostraremos uno de los 3 mensajes según corresponda a fin de fase, fin de juego o pantalla de selección de modo. En caso contrario, simplemente leeremos del tablero la codificación a leer y tomaremos los datos de la memoria de *Sprites*.

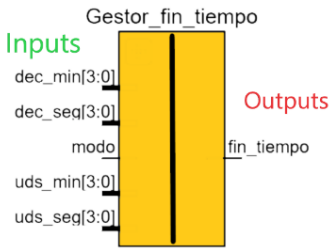
2.8. Gestor fin de tiempo

- *Descripción:*

Este bloque se encarga de activar la señal de fin de tiempo cuando se acaba el tiempo en modo difícil.

- *Entidad:*

```
entity logica_fin_tiempo is
    Port ( uds_seg : in STD_LOGIC_VECTOR (3 downto 0);
           dec_seg : in STD_LOGIC_VECTOR (3 downto 0);
           uds_min : in STD_LOGIC_VECTOR (3 downto 0);
           dec_min : in STD_LOGIC_VECTOR (3 downto 0);
           modo : in STD_LOGIC;
           fin_tiempo : out STD_LOGIC);
end logica_fin_tiempo;
```



- *Descripción de los puertos:*

uds_seg dec_seg uds_min dec_min	Entradas que determinan el valor de los contadores de tiempo
modo	Indica el modo de juego
fin_tiempo	Salida que se activa si el temporizador ha llegado a cero en modo difícil

- *Descripción de la arquitectura:*

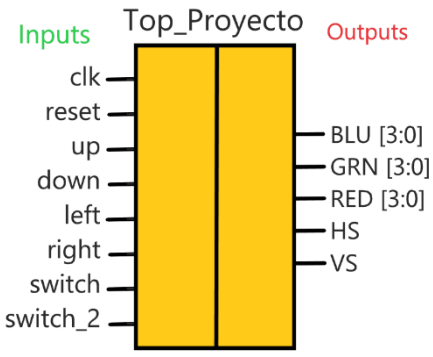
Simplemente se trata de un proceso con una sentencia *if* que pone a '1' la señal *fin_tiempo* si el modo difícil está activo y el temporizador llega a 59:59. Lo activamos con este valor y no con 00:00 ya que, en caso contrario, la señal se activaría si estuviéramos en la pantalla de selección de modo y activamos el modo difícil, haciendo que se vaya al estado de leer fase. Con esto se evita ese error, ya que así el contador llegará primero a cero, y al desbordar se activa *fin_tiempo*.

3. BLOQUE DE JERARQUÍA SUPERIOR (TOP_PROYECTO)

En este apartado se detalla la interconexión de las instancias de los distintos componentes anteriormente explicados, así como el funcionamiento global en conjunto de todos los bloques, englobándolos dentro de un bloque de jerarquía superior (*Top_proyecto*).

- *Entidad:*

```
entity Top_proyecto is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           up: in STD_LOGIC;
           down: in STD_LOGIC;
           left: in STD_LOGIC;
           right: in STD_LOGIC;
           switch: in STD_LOGIC;
           switch_2: in STD_LOGIC;
           HS : out STD_LOGIC;
           VS : out STD_LOGIC;
           RED : out STD_LOGIC_VECTOR (3 downto 0);
           GRN : out STD_LOGIC_VECTOR (3 downto 0);
           BLU : out STD_LOGIC_VECTOR (3 downto 0));
end Top_proyecto;
```



- *Descripción de los puertos:*

clk	Señal de reloj de la placa
reset	Botón central de la FPGA que se usará como *reset*

down, up, left, right	Botones de la FPGA que son usados para mover el jugador
switch	Primer interruptor de la FPGA, indicará el modo de juego
switch_2	Segundo interruptor de la FPGA, activa el modo solución
HS	Pulso de sincronismo horizontal
VS	Pulso de sincronismo vertical
RED	Salida que indica con 4 bits la cantidad de color rojo que contendrá el píxel a representar por pantalla
GRN	Salida que indica con 4 bits la cantidad de color verde que contendrá el píxel a representar por pantalla
BLU	Salida que indica con 4 bits la cantidad de color azul que contendrá el píxel a representar por pantalla

- *Descripción de la arquitectura:*

Declaramos todos los componentes del proyecto y las señales intermedias necesarias para interconectar las instancias asociadas a dichos componentes. En el bloque *Top_proyecto* por tanto usamos prácticamente en su totalidad la metodología de diseño estructural.

Además de la interconexión de instancias encontramos un proceso cuya función es la de seleccionar los datos de la memoria de *Fases* o de la memoria *Tablero* para que sean entrada del *Gestor de juego* (durante el desarrollo del juego serán los datos del tablero los que se necesiten, pero cuando se producen cambios de fase es necesario copiar el nuevo mapa de la memoria de *Fases* a la de *Tablero*):

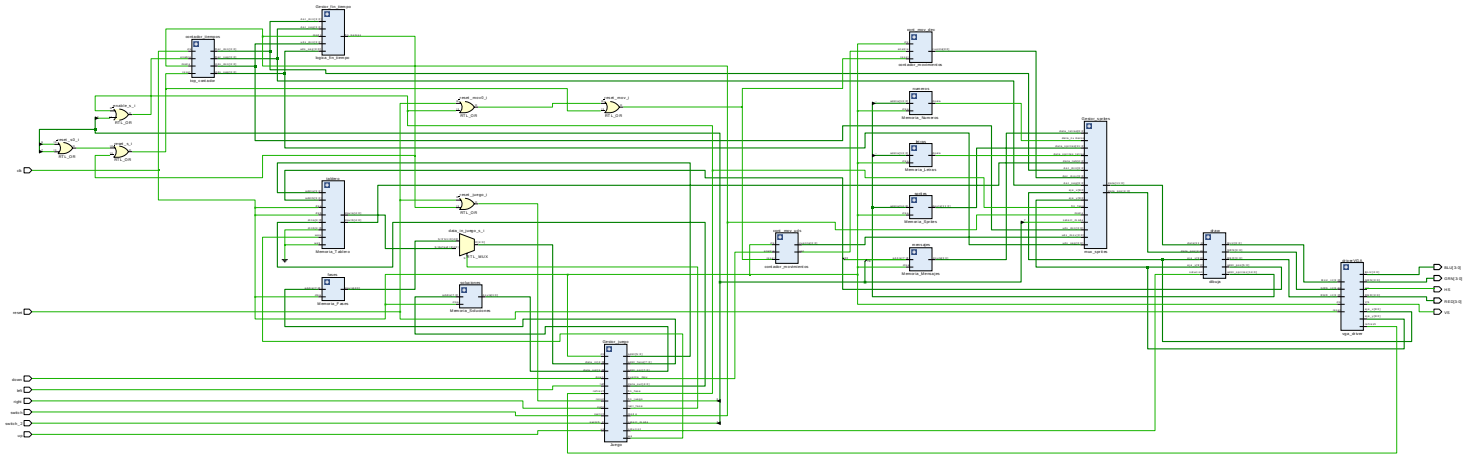
```
multiplexor_juego: process(lee_fase_s, data_fase_s, data_tab_juego_s)
begin
if(lee_fase_s='1') then
    data_in_juego_s <= data_fase_s;
else data_in_juego_s <= data_tab_juego_s;
end if;
end process;
```

También tenemos un conjunto de secuencias concurrentes que dan lugar a puertas lógicas simples necesarias para el correcto funcionamiento del juego:

```
enable_s <= fin_fase_s or select_mode_s;
reset_s <= fin_juego_s or select_mode_s or fin_tiempo_s;
reset_juego <= reset or fin_tiempo_s;
reset_mov <= reset or fin_fase_s or reset_s;
addr_letras_s <= select_mode_s & fin_juego_s & addr_pos_s;
```

- *Interconexiones de componentes:*

(se adjunta archivo top_proyecto.sch)



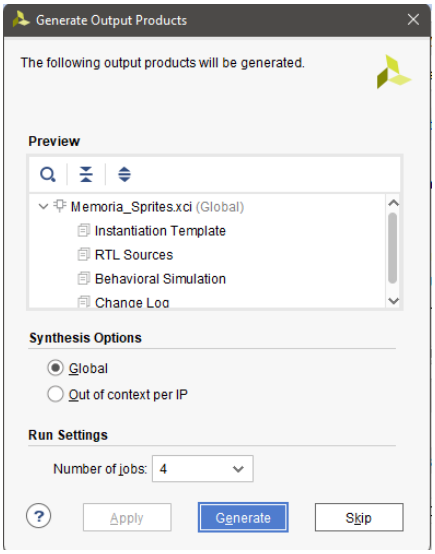
4. OPTIMIZACIÓN Y ADAPTACIÓN DE RECURSOS

En todo momento se ha intentado priorizar la optimización del circuito generado. Esto es, multiplexando señales en el top, preparando en estados anteriores las direcciones de memoria a leer, configurando estrategia del sintetizador de Vivado…

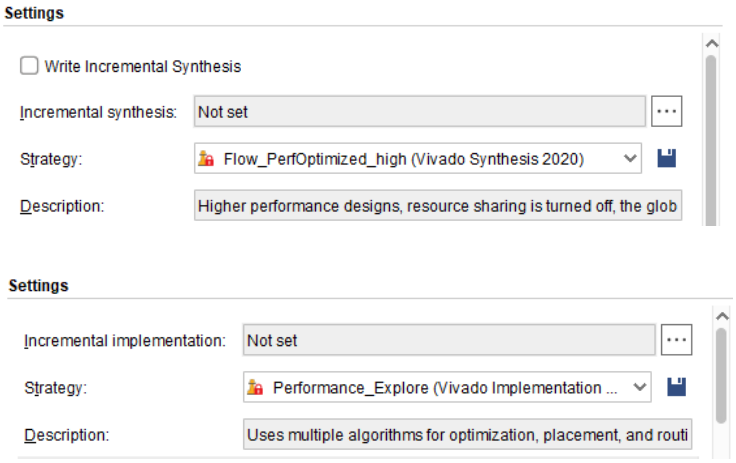
Por ello, tras el desarrollo de un código funcional, se han llevado a cabo diversas etapas de depuración y mejora de eficiencia del circuito, principalmente basándose en la información obtenida en los avisos:

- Aunque primeramente se planteó el uso de reset asíncrono en todos los bloques del circuito, los avisos del sintetizador mostraban que no era buena idea debido a posibles fallos que podían producirse al estar dichos bloques conectados a memorias síncronas.

- En un primer momento se intentó realizar la síntesis de los bloques de memoria por IP Generator a través del modo 'Out of context', ya que esto permitía un menor tiempo de 'runtime' sintetizando. No obstante, se obtuvo un aviso que informaba que la síntesis del circuito se generaba para una frecuencia de reloj de 20 MHz, cuando la del proyecto es de 10 MHz. Esto hacía que el circuito inferido no estuviera del todo adaptado a nuestra aplicación, por lo que tras cambiar la opción de síntesis a 'Global', donde se analiza el resto del proyecto para tener en cuenta la señal de reloj real que se utiliza, este problema desaparecía.



Por otro lado, buscando optimizar todo lo posible, se ha configurado Vivado para que su sintetizador e implementador utilicen una estrategia enfocada al rendimiento y uso de área mínimo, permitiéndole incrementar el tiempo de 'runtime' a costa de usar múltiples algoritmos que potencialmente generen mejores resultados según nuestros criterios. La configuración final elegida fue:.



Tras todo ello, se ha conseguido que el circuito generado en la placa tenga las siguientes características, según los reportes obtenidos:

Site Type	Used	Fixed	Available	Util%
Slice LUTs	408	0	20800	1.96
LUT as Logic	408	0	20800	1.96
LUT as Memory	0	0	9600	0.00
Slice Registers	176	0	41600	0.42
Register as Flip Flop	176	0	41600	0.42
Register as Latch	0	0	41600	0.00
F7 Muxes	9	0	16300	0.06
F8 Muxes	0	0	8150	0.00

Site Type	Used	Fixed	Available	Util%
Slice	137	0	8150	1.68
SLICEL	96	0		
SLICEM	41	0		
LUT as Logic	408	0	20800	1.96
using O5 output only	0			
using O6 output only	338			
using O5 and O6	70			
LUT as Memory	0	0	9600	0.00
LUT as Distributed RAM	0	0		
LUT as Shift Register	0	0		
Slice Registers	176	0	41600	0.42
Register driven from within the Slice	159			
Register driven from outside the Slice	17			
LUT in front of the register is unused	5			
LUT in front of the register is used	12			
Unique Control Sets	18		8150	0.22

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	11	0	50	22.00
RAMB36/FIFO*	6	0	50	12.00
RAMB36E1 only	6			
RAMB18	10	0	100	10.00
RAMB18E1 only	10			

Se puede observar cómo las unidades básicas CLB (bloque lógico configurable) utilizadas suponen un porcentaje de utilización mínimo del disponible. De igual manera, con la memoria RAM también se dispone de un gran porcentaje sin uso para la ampliación del proyecto.

Con todo esto, hemos conseguido un proyecto funcional, completo y eficiente, añadiendo nuevas mecánicas y optimizando los recursos disponibles.