

Autores: Francisco Javier Álvarez Ropero, José Araujo González y

Alberto González Isorna.

Memoria del Proyecto Snake

Proyecto Sistemas Electrónicos

Grado en Ingeniería en Electrónica, Robótica

y Mecatrónica

Dep. de Ingeniería Electrónica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2017

Índice

[1 Introducción 1](#_Toc473361049)

[2 Estructura del Proyecto 3](#_Toc473361050)

[2.1. Joystick 3](#_Toc473361051)

[2.1.1 Preamplificador de ganancia 4](#_Toc473361052)

[2.1.2 Control del Interfaz SPI (Preamplificador de Ganancia) 5](#_Toc473361053)

[2.1.3 Convertidor Analógico Digital 5](#_Toc473361054)

[2.1.4 Control del Interfaz SPI (Convertidor Analógico Digital) 6](#_Toc473361055)

[2.2. Control 7](#_Toc473361056)

[2.2.1 Desplazamiento 7](#_Toc473361057)

[2.2.2 Lugar de desplazamiento 7](#_Toc473361058)

[2.2.3 Situaciones del juego 7](#_Toc473361059)

[2.3 Representación por Vga 8](#_Toc473361060)

[2.3.1 Introduccion VGA 8](#_Toc473361061)

[2.3.2 Descripción del bloque: “Dibuja” 8](#_Toc473361062)

[2.3.3 Memorias RAM y ROM 9](#_Toc473361063)

[3 Solución a nivel de bloque 11](#_Toc473361064)

[3.1 Joystick 11](#_Toc473361065)

[3.1.1 Máquina de Estados (Comunicación con el Circuito de Captura Analógica) 11](#_Toc473361066)

[3.1.2 Máquina de Estados (Decisión de Dirección de la Snake) 14](#_Toc473361067)

[3.2 Control 15](#_Toc473361068)

[3.2.1 Máquina de estados 15](#_Toc473361069)

[3.3. Representación por VGA 17](#_Toc473361070)

[3.3.1 Representacion de tablero (RAM) 17](#_Toc473361071)

[3.3.2 Representación de las memorias ROM 18](#_Toc473361072)

[3.3.3 Representación de seta aleatoria 19](#_Toc473361073)

[4 Anexos 20](#_Toc473361074)

[4.1. Anexo I: código Matlab 20](#_Toc473361075)

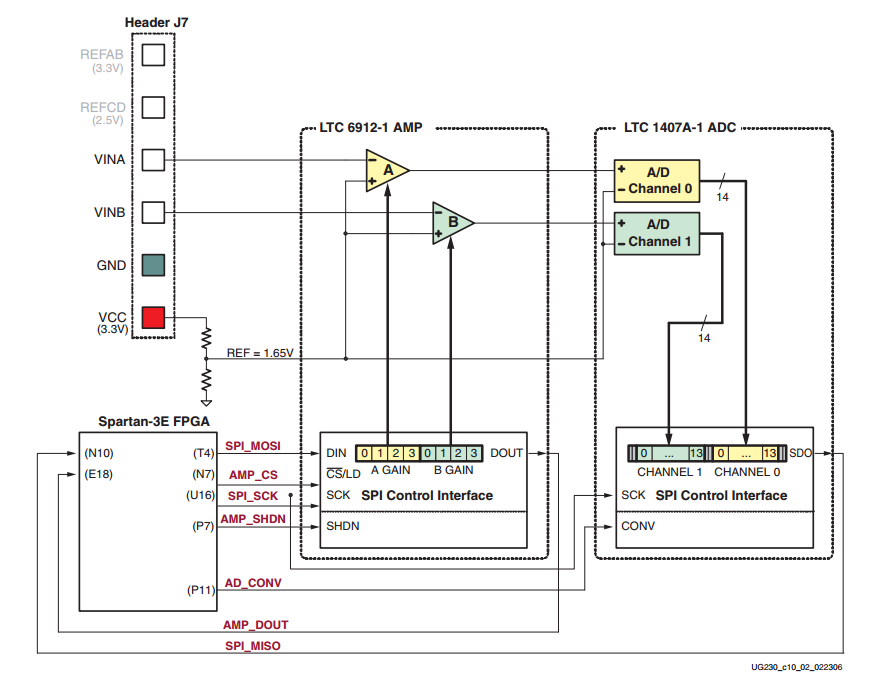
[4.2. Anexo II: Sumario de diseño del proyecto 21](#_Toc473361076)

# Introducción

# Estructura del Proyecto

## Joystick

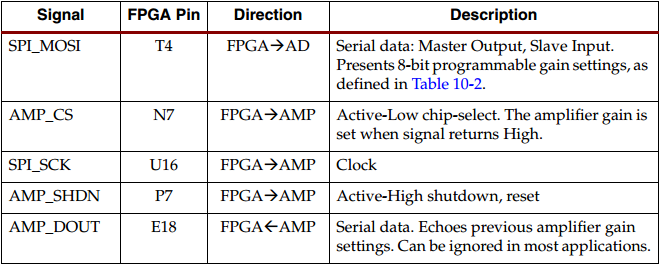
En este primer bloque, se encontrará toda la documentación e información necesaria para comprender el funcionamiento del circuito de captura de analógica *(****Analog Circuit Capture****)*, el cual consiste en la conjunción de un **Preamplificador de Ganancia** *(LTC 6912-1 AMP)*y un **Convertidor Analógico Digital** *(LTC 1407A-1 ADC)*. En la siguiente figurá se verá el interconexiado entre ambas partes.

****

Asimismo se explicará cual es la forma de comunicación con dicho circuito, a través de un protocolo **SPI** (*Serial Peripheral Interface*), cuya documentación irá implícita en cada apartado

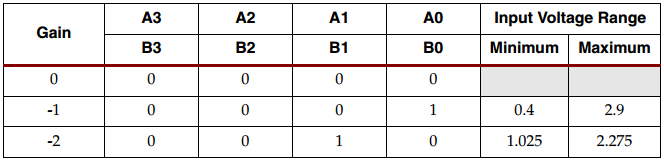
### Preamplificador de ganancia

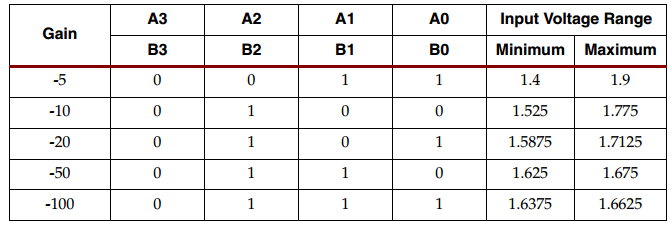
Estos amplificadores inversores de ganancia programable escalan el valor analógico de entrada para posteriormente mandarlo al convertidor analógico digital. En primer lugar, se verá el interfaz entre la FPGA y el amplificador.



Notar que, las señales *SPI\_MOSI* y *SPI\_SCK* están compartidas con otros dispositivos en el bus SPI y que *AMP\_CS* es una señal de entrada activa a nivel bajo.

La ganancia programable es de utilidad para darle un rango a la entrada máximo y mínimo a cada canal de entrada del circuito de captura analógico, dicho valor es programable desde -1 a -100 como se figura en la siguiente tabla.

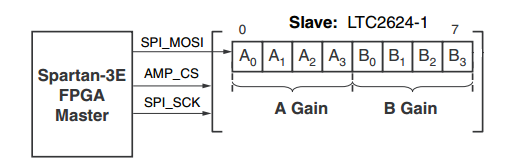




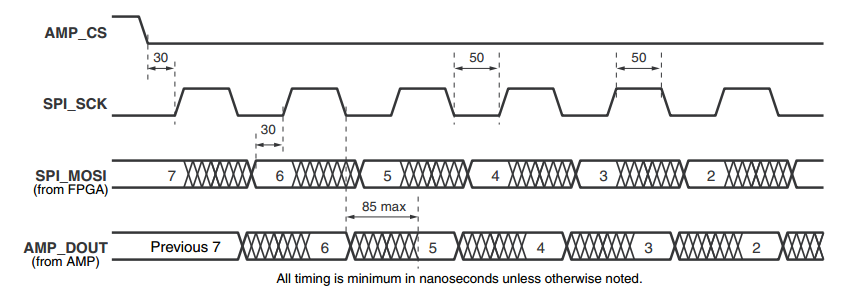
Siendo un vector de 8 bits mandado a través del *SPI\_MOSI,* que llevará el valor de la ganancia a amplificar.

### Control del Interfaz SPI (Preamplificador de Ganancia)

En este apartado trataremos como realizar la comunicación con el amplificador, siendo la única misión enviar el vector de ganancia una sola vez, comenzando por el bit , tal como se muestra en la siguiente figura:



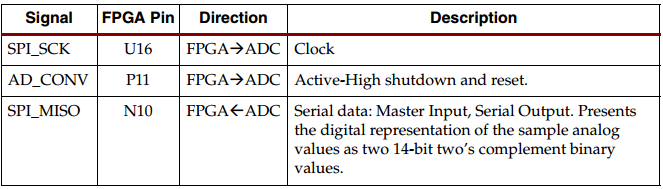
La comunicación empieza cuando la FPGA activa a nivel bajo *AMP\_CS*, el amplificador captura los datos en serie en *SPI\_MOSI* en el flanco de subida de la señal de reloj *SPI\_SCK.*



La frecuencia máxima de la comunicación SPI es de 10 MHz.

### Convertidor Analógico Digital

Estos convertidores se activan al mismo tiempo con la señal *AD\_CONV*, de forma simultánea además realizan la conversión. Los valores en digital tendrán una extensión de 14 bits por cada conversor, todos ellos serán enviado en serie. El interfaz entre la FPGA y los convertidores se condensan en la siguiente tabla.

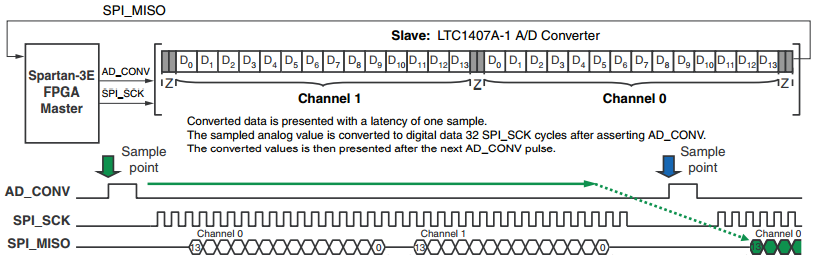


Notar que las señales *SPI\_SCK* y *SPI\_MISO* están compartidas con otros dispositivos en el bus SPI, la señal *AD\_CONV* deberá ser activada en cada conversión pues en caso contrario tendrá el último valor guardado y se leería constantemente ese último. La frecuencia máxima de conversión es de 1.5 MHz.

Detallar además que por *SPI\_MISO* será por donde lleguen los valores digitales de 14 bits por canal, en serie.

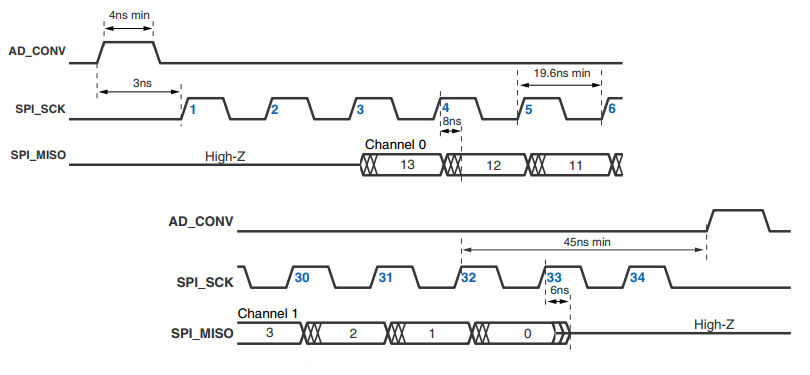
### Control del Interfaz SPI (Convertidor Analógico Digital)

La disposición de como vamos a recibir los bits se ve claramente en la siguiente figura:

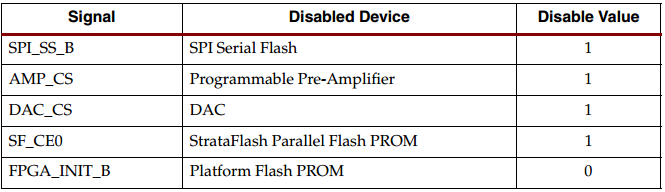


Comenzará de esta forma a enviar por el bit del canal 0 y todos los restantes, con un espacio de dos ciclos de reloj *SPI\_SCK* entre ambos canales.

Así pues, la comunicación comienza con la activación de *AD\_CONV* a nivel alto, tras ello damos comienzo al reloj *SPI\_SCK,* tras los dos primeros ciclos el convertidor empezará a enviar cada bit del canal 0 por cada ciclo, comenzando desde el más significativo. Una vez terminado los 14 bits, hay un espacio de dos ciclos de *SPI\_SCK* para más tarde enviar los otros 14 bits del canal 1 de la misma manera y otros dos ciclos al final, en total la transmisión tardará 34 ciclos.



Subrayar también que los demás dispositivos deben de estar deshabilitados del bus SPI para evitar que varios escriban al mismo tiempo en el canal *SPI\_MISO*, los cuales vienen dados en la siguiente tabla:



## Control

En este bloque se explicará el sentido físico del control de la serpiente durante todo el tiempo de juego.

Para ello hay se piensa en el juego como si fuera una matriz a la cual se rellena cada posición según nos corresponda.

### Desplazamiento

Al empezar, se tendrá a la serpiente en una posición específica a través de dos variables denominadas Ynext y Xnext. Las cuales ocupan cinco bits y las dos concatenadas accederemos a la dirección de memoria de la RAM deseada. Este movimiento dependerá de hacia dónde queramos ir.

Además dicho movimiento no puede ser contradictorio, es decir, al estar moviéndose hacia arriba no debe ir hacia abajo aunque el jugador pulse dicha dirección.

### Lugar de desplazamiento

Una vez que se ha decidido hacia donde irá la serpiente, esa posición tendrá en la memoria un valor asignado.

Dicho valor dependerá de si hay una seta, un muro, un hueco o una serpiente en ella. Esto hace que según se halle un valor u otro suceda lo siguiente: la serpiente crezca, se mueva o muera.

### Situaciones del juego

Si en la siguiente posición a la cual se moverá la serpiente se encuentra una parte de la serpiente o un muro, el fin será el mismo, el juego termina porque la serpiente choca con una parte sólida.

Al haber una seta, la serpiente debe incrementar su cuerpo una unidad y hacer aparecer una seta en un lugar distinto, a la vez que aleatorio.

En cambio, si la posición se encuentra vacía la serpiente debe mantener el tamaño. Para ello, la solución propuesta es hacer lo mismo que en el crecimiento y posteriormente borrar la cola.

## 2.3 Representación por Vga

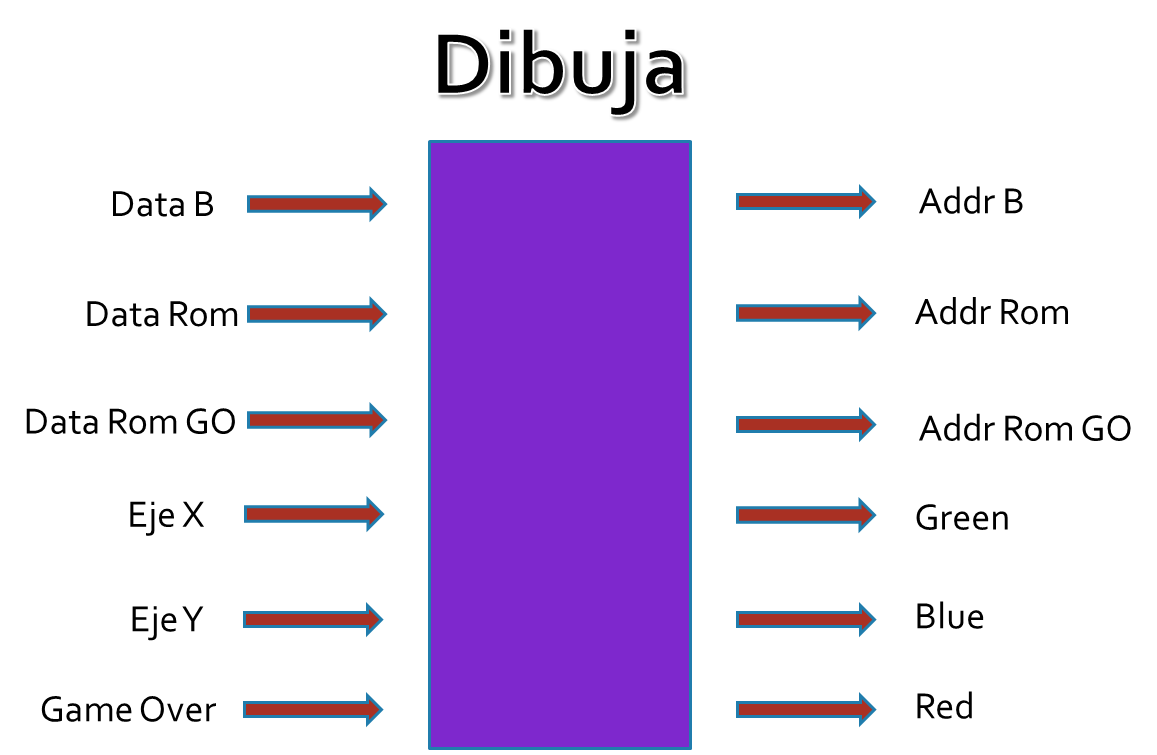
### Introduccion VGA

Una vez que tenemos una idea de cómo va a ser el procedimiento del juego, tenemos que plasmarlo adecuadamente a través del estándar de gráficos VGA. Afortunadamente, los parámetros y bloques internos los tenemos realizados en prácticas, así que lo único que nos interesará será modificar adecuadamente el bloque que se encarga de dibujar por pantalla para que represente el tablero del juego entre otras cosas.

La idea es simple, convertir los ejes (X e Y) que van recorriendo toda la pantalla, en una posición de memoria determinada, de forma que el primer pixel corresponda con la primera posición de memoria y el último con la última. Como no nos interesa que el tablero coja toda la pantalla y queremos añadir un fondo, se explicará más adelante cual ha sido el procedimiento elegido para adecuar el tablero a la pantalla.

### Descripción del bloque: “Dibuja”

Como hemos dicho anteriormente, el bloque dibuja es el principal a la hora de representar el juego por pantalla. A nivel de caja negra sería de la siguiente forma:



|  |  |  |  |
| --- | --- | --- | --- |
| ENTRADAS | FUNCION | SALIDAS | FUNCION |
| Data B | Dato que hay en una determinada posición de la memoria RAM. | **Addr B** | Direción de la RAM |
| Data ROM | Dato en la ROM ue se ocupa del fondo | **Addr ROM** | Dirección de la ROM del fondo |
| Data ROM GO | Dato en la ROM que se ocupa de la pantalla de fin de juego | **Addr ROM GO** | Direción de la ROM de fin de juego |
| Eje X, Eje Y | Ejes que va recorriendo la pantalla | **RGB** | Señales Para cada color |
| Game Over | Señal que indica que el juego ha acabado |  |  |

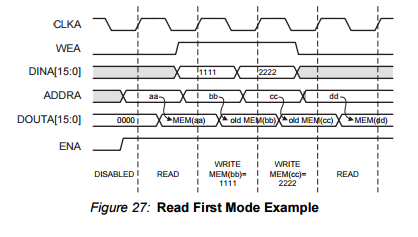
### Memorias RAM y ROM

La memoria RAM la escogimos del tamaño del tablero, 32x32. La ventaja de elegir un múltiplo de dos es la fácil asinación de ejes, como se explicará más adelante. La desventaja que no cuadra con las dimensiones de la pantalla ya que ninguna es múltiplo de dos. Como tampoco queremos que el tablero ocupe toda la pantalla en principio no es un problema.

Por lo tanto la memoria RAM tiene direcciones de la 0 a la 1023, siendo necesarios 10 bits para direccionarla. La ram es de doble puerto, es decir, que podemos extraer resultados y poner resultados desde dos señales diferentes. Estonos permite dedicar una parte a la máquina de estados y otra a la parte de representar. Para evitar conflictos de escritura, se elige que solo la máquina de estados pueda escribir, lo que es lógico ya que es la que va a ir realizando cambios en el juego y por tanto en la memoria. Po otro lado, en situaciones como representar la seta en diferentes posiciones aleatoria, hubiera sido útil poder escribir en ambos lados. La solución ha sido añadir este tipo de cosas a la máquina de estados, que en términos de tiempo de ejecución no se ve afectada por añadir más estados.

En segundo Lugar se introducen dos memorias ROM, solo lectura, para el fondo, y la pantalla de fin de juego. Desgraciadamente hay que optar por memorias más grandes ya que hemos calculado que para distinguir bien una letra hace falta de promedio un cuadrado de 5x5 es decir, 25 píxeles o 25 direcciones de memoria aunque no se ocupen todas más los espacios entre caracteres. Lo que haciendo con el mismo método que en la RAM podríamos incluir unas 5 letras en el ejex.

Por último aclarar que el acceso tanto a RAM como a ROM no es siempre inmediato, para lectura si apuntamos a una dirección si podemos saber el dato en el mismo ciclo de reloj, sin embargo para escritura hay que esperar un ciclo entero de reloj, a que carge la dirección y se pueda escribir. Estos modos se pueden configurar en el IPcore y decidimos que este funcionamiento era el más adecuado, (READ FIRST).



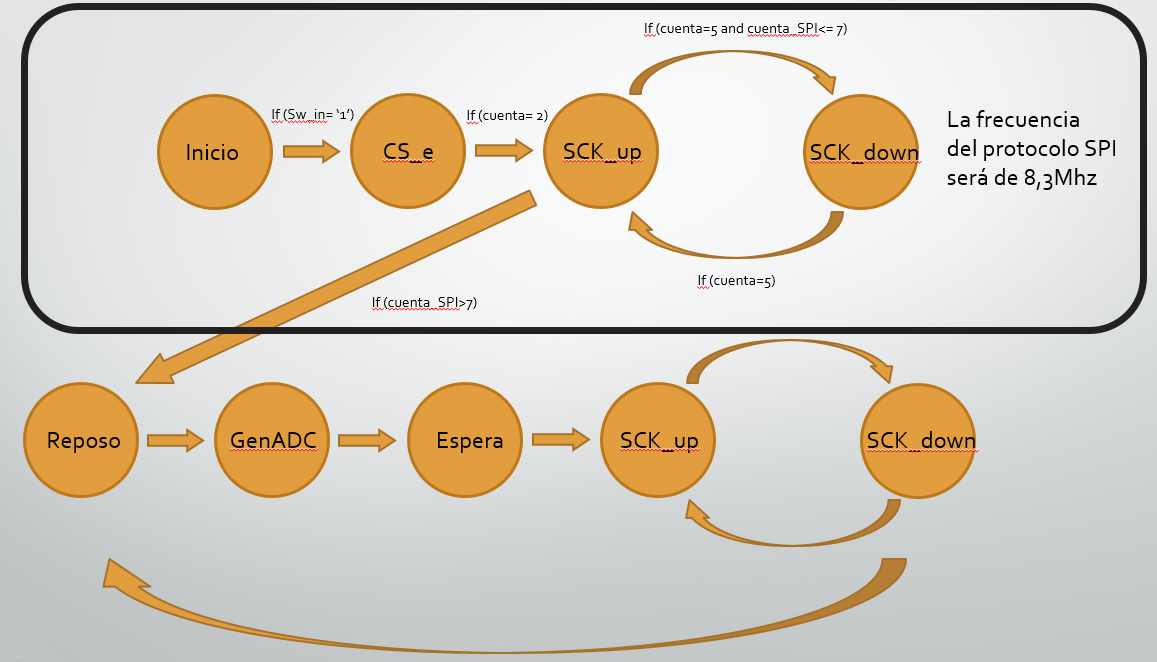
# Solución a nivel de bloque

## Joystick

En este apartado, se abordará la solución propuesta para llevar a cabo la conversión y comunicación con el circuito de captura analógica. La solución adoptada se dividirá en dos partes: la primera constará de la comunicación con el circuito y la segunda parte tomará las decisiones para decidir a que dirección irá la serpiente y cada cuánto tiempo se hará las lecturas del convertidor analógico digital.

### Máquina de Estados (Comunicación con el Circuito de Captura Analógica)

En primer lugar, la máquina de estados que nos ayuda a resolver el problema de la comunicación es la siguiente:

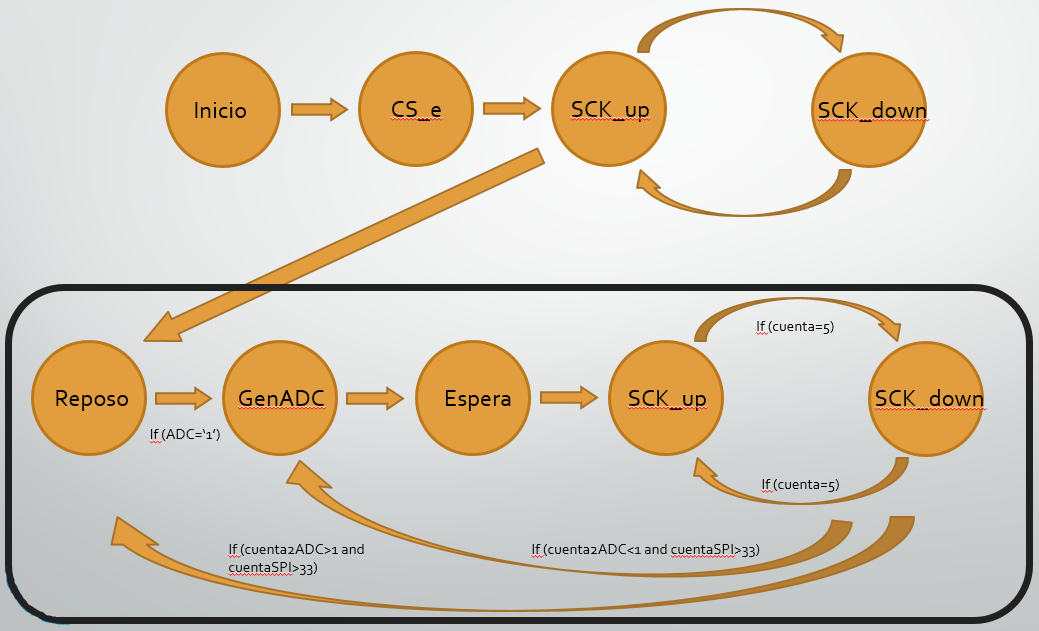


La parte recuadrada es la responsable de mandar el vector de ganancia a través de *SPI\_MOSI,* además de generar el reloj *SPI\_SCK*. El reloj tendrá una frecuencia de 8,3 MHz, la cual es menor que los 10 MHz necesarios para la comunicación con el preamplificador.

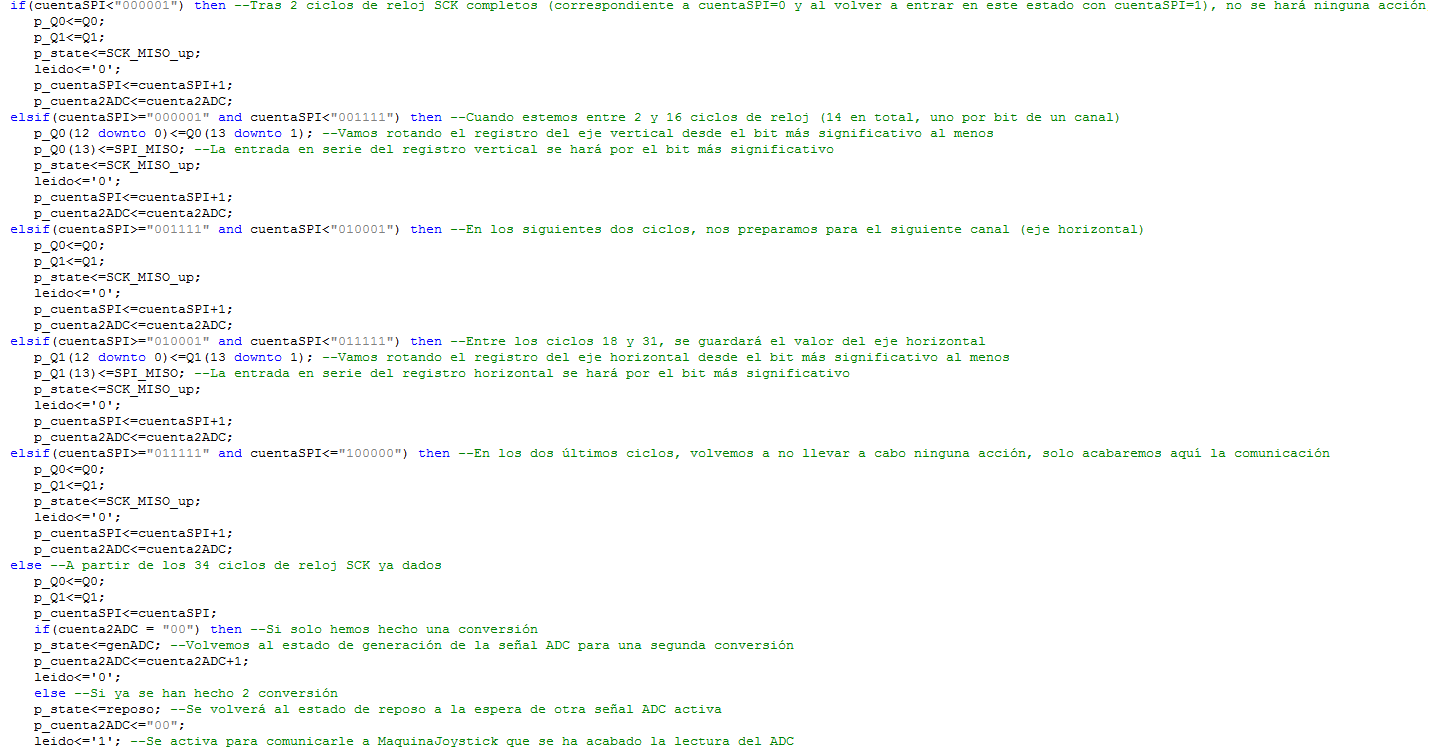
Cabe destacar la sencillez del envío a través de un registro de desplazamiento con salida en serie y carga en paralelo (en cada reinicio, inicializaremos ese registro *GainVector* en “00010001”), en código VHDL queda así de sencillo:



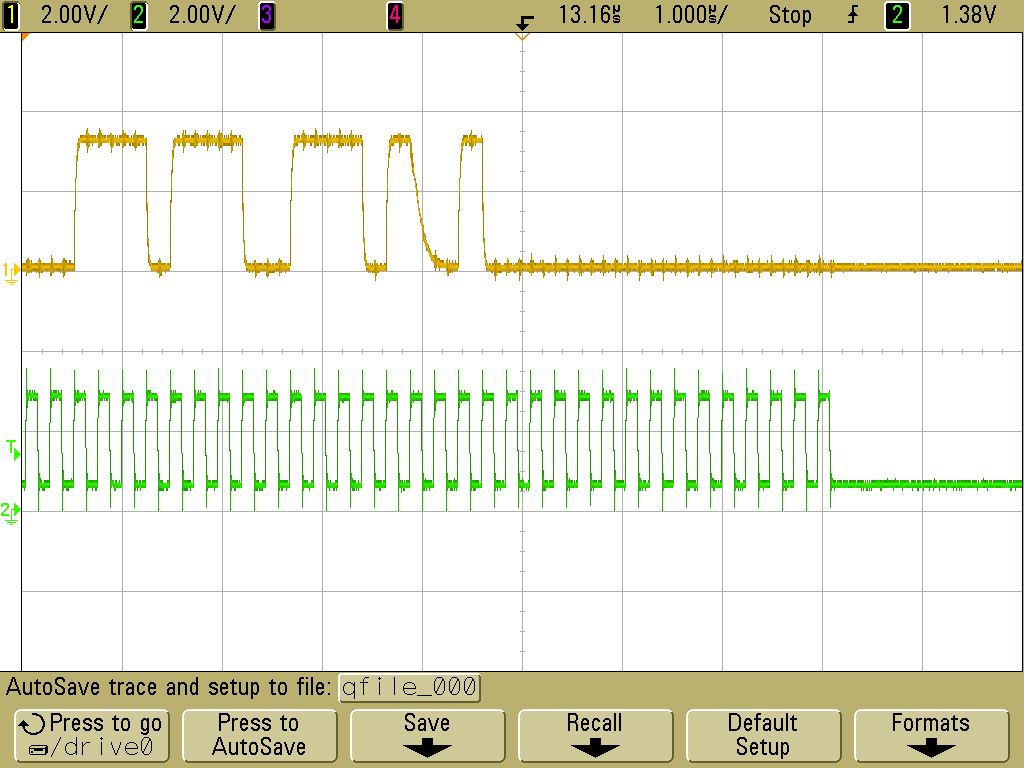
Se hace rotar el registro hacia la izquierda mientras se mandan el bit más significativo de *GainVector*, en el momento que se haya enviado los 8 bits de ganancia, la máquina de estados irá a la parte que de forma cíclica, lee los convertidores.



En este caso, la zona recuadrada genera la señal *AD\_CONV* y posteriormente la señal *SPI\_SCK*, en ella estaremos guardando tal y como se ha visto en el apartado de **Control de Interfaz SPI** los valores de cada canal, siendo su código VHDL el siguiente:



De esta forma se podrá guardar en dos registros de desplazamientos distintos de entrada serie y salida paralelo para posteriormente enviarlo a la segunda parte. Sin embargo, vamos a ver de qué forma, visto desde un analizador digital, quedan las señales recibidas a través de *SPI\_MISO* y *SPI\_SCK*.

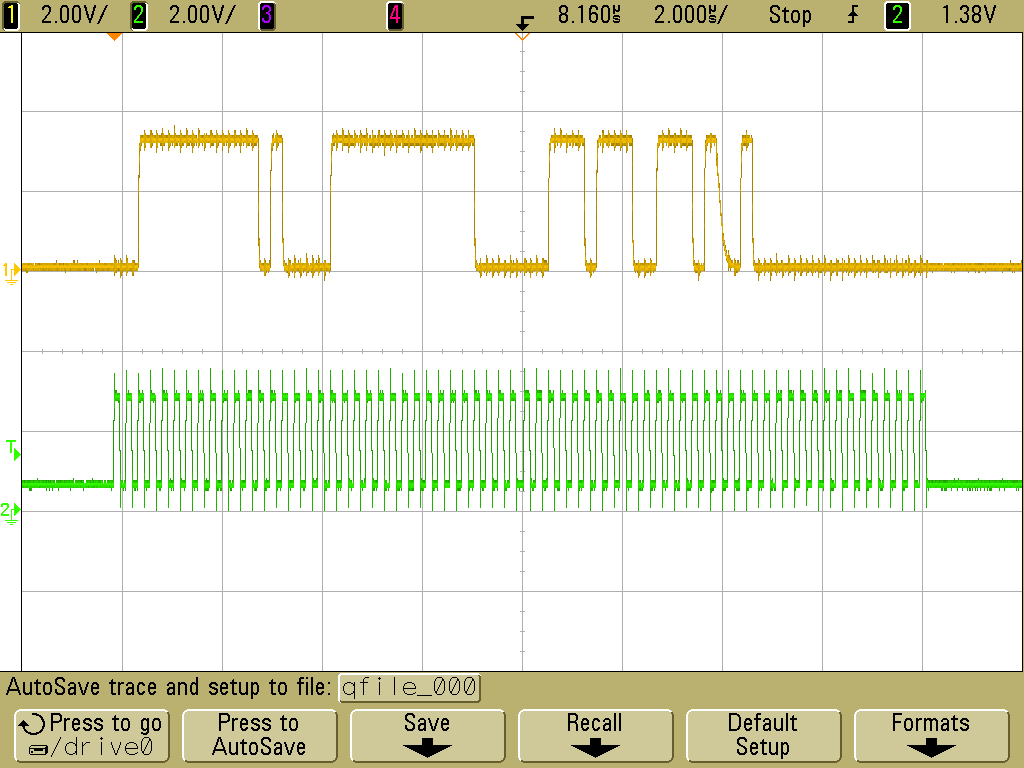


Horizontal

Vertical

Aquí se ve claramente como la señal verde es el reloj generado a través de la máquina de estados (*SPI\_SCK*)y la señal anaranjada, la que sale desde el convertidor hasta la FPGA (*SPI\_MISO*), en el primer canal obtendremos el valor del eje vertical del Joystick y en el segundo, el eje horizontal del mismo.

Cabe subrayar un problema que surgió durante las pruebas, a pesar que la señal *AD\_CONV* estaba siendo bien activada, el valor que da tras una lectura del convertidor, es el último medido en este mismo; es decir, de alguna forma se queda guardado en el registro del ADC el último valor digital, este error ha sido sencillamente subsanado realizando dos lecturas seguidas por cada iteración, de la siguiente manera visto desde el analizador digital:



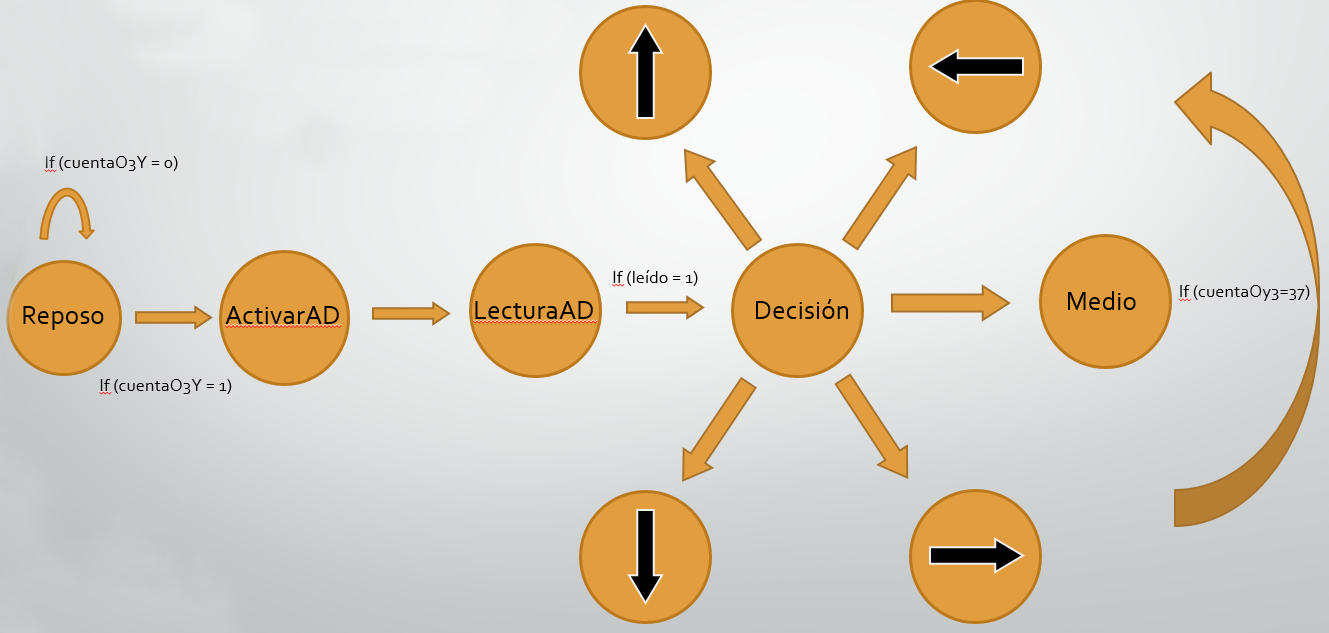
2º conversión

1º conversión

La segunda conversión es la correcta, por ello se desecha la primera directamente.

### Máquina de Estados (Decisión de Dirección de la Snake)

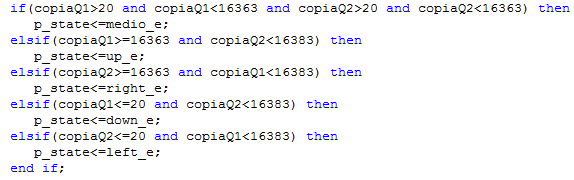
En este apartado se verá la máquina de estados destinada a cómo decidir dependiendo del valor ya digitalizado del Joystick, hacia donde deberá ir.



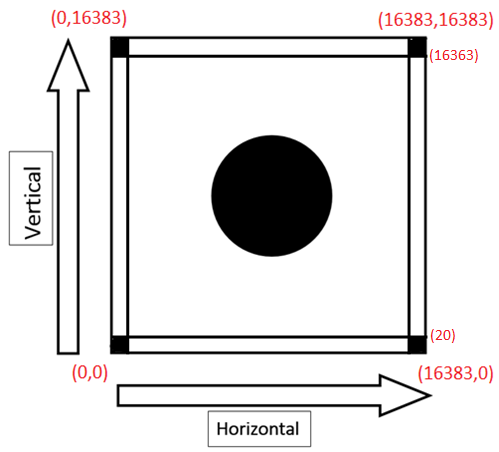
De esta forma tan sencilla, tenemos la solución al problema de cómo decidir la dirección de la Snake, esta máquina de estados consta de algunos detalles que se procederán a explicar seguidamente:

La frecuencia a la que leemos los convertidores (dando por hecho ya que se leen dos veces en cada iteración) vendrá dada por la señal O3Y cuya prodecendia radica en el bloque **Representación por VGA**, la cual será contada 17 veces en cualquiera de los estados (*arriba, abajo, izquierda, derecha o medio*) pues el bloque **Control** lee sus entradas cada 15 veces O3Y y así nos aseguramos que captará las señales de los estados correctamente.

Por otra parte, los números escogidos y la toma de decisiones se verá mas claramente en el código VHDL que se adjunta:



Así pues, nos queda una zona de funcionamiento que se podría esquematizar de la siguiente forma, de la manera más sencilla posible. Se escogió esta división del espacio en el que trabaja el Joystick por simplificación del problema y hacer más rápida la toma de decisiones, de otra forma los problemas eran múltiples.

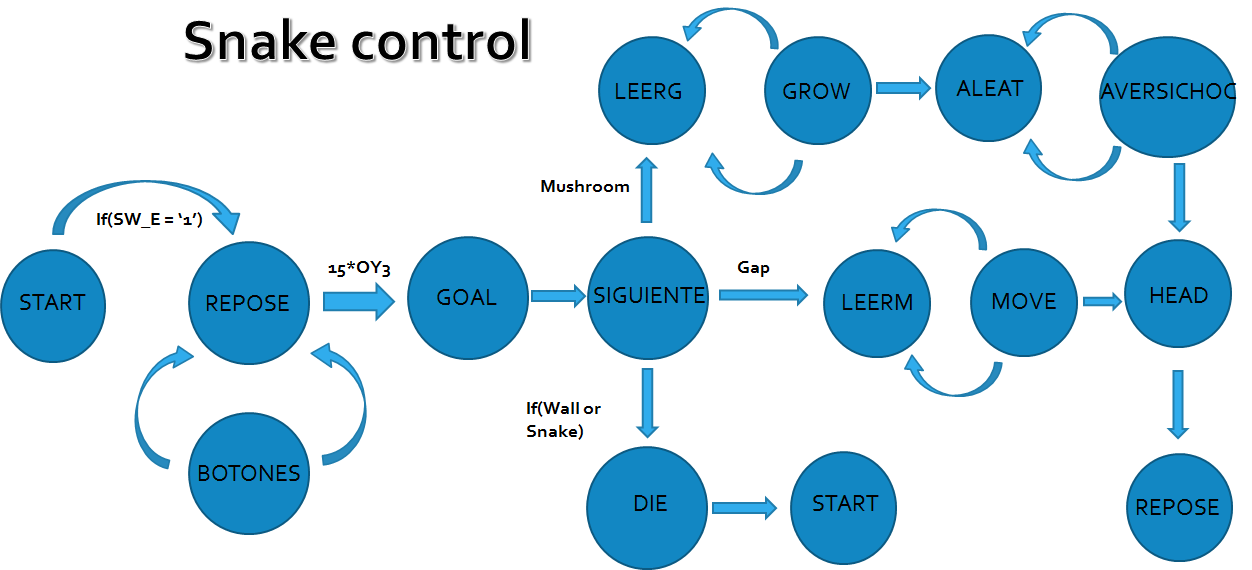
.

## Control

En este apartado, se proporcionará la solución propuesta para la máquina de estados que controla el movimiento de la serpiente a través de una máquina de estados.

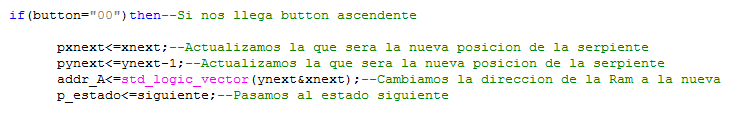
### Máquina de estados

La máquina de estados propuesta es la siguiente:

****

En ella, cabe destacar el uso de la señal OY3 como contador. En dicho estado *(Repose),* tras haber pulsado el joystick para pasar *Start*, hay una observación de la dirección que quiere el jugador que tome la serpiente.

Dicho movimiento corresponderá a sumar o restar una unidad al valor de Xnext o Ynext según corresponde con la imagen:



Tras pasar el tiempo de espera deseado para que el jugador tome la decisión de qué dirección desea tomar pasamos al siguiente estado, *Goal.*

Aquí se hará la comprobación de la siguiente posición de la cabeza de la serpiente para saber cual de las siguientes posibilidades se encuentra: seta, muro, hueco o la propia serpiente una vez que haya crecido lo suficiente.

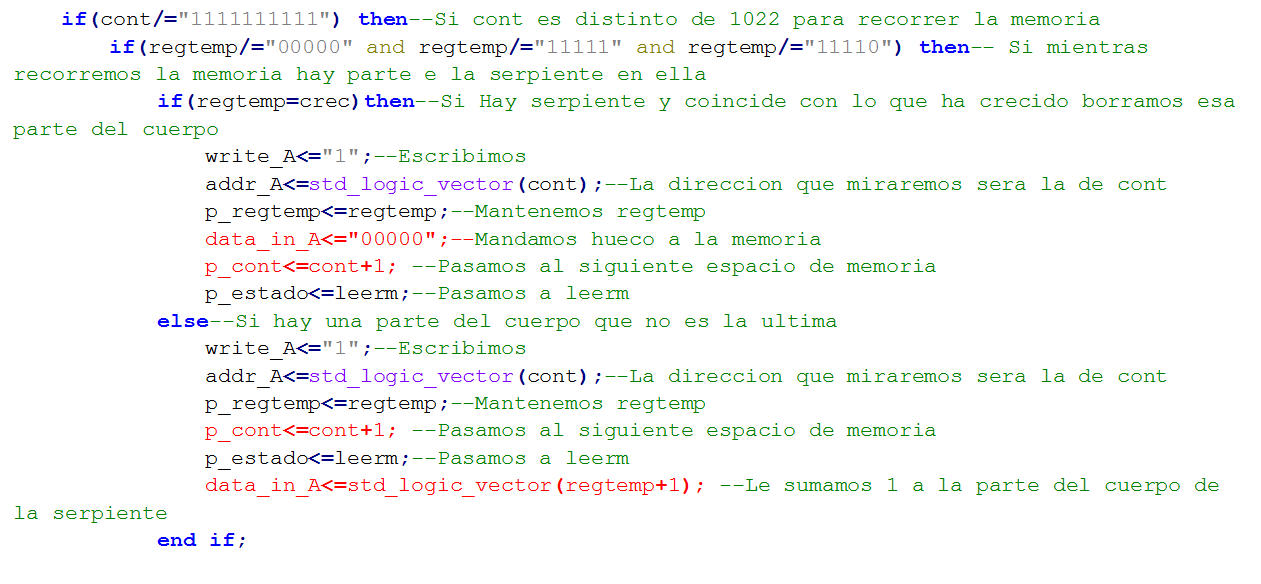
Una vez se sepa qué hay, en el estado *siguiente,* se toma la decisión de ir por la rama de crecimiento (*Grow*), movimiento (*Move*) o muerte (*Die*).

Tanto la rama de crecimiento y la de movimiento tienen la misma lectura de memoria ram, para la cual se usará un contador que designará la posición de memoria y un registro temporal que guardará el valor que halla en dicha posición de memoria como se adjunta en la imagen:

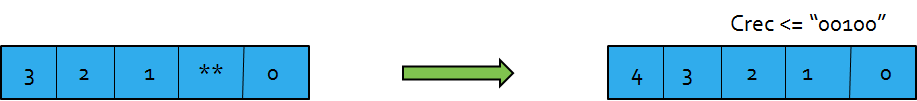


En la rama de *Move*, además deberemos de borrar la última parte del cuerpo de la serpiente. Así pues, se comparará el valor del cuerpo de la serpiente con la variable mencionada anteriormente.

La técnica usada será sumarles una unidad a todas las posiciones de la serpiente, contando como si hubiera crecido para así simplemente borrando la cola se consiga el propósito buscado como se muestra en la imagen, destacando el uso del contador y del registro temporal.



Respecto a la rama de crecimiento, se tendrá que tener un control de lo que ha crecido la serpiente. Para ello, se usará una variable llamada *crec* inicializada a uno, que indicará cual es la última parte de la serpiente.



Posteriormente, hay que buscar una posición aleatoria a una nueva seta, que debe ser aleatoria y no coincidir con la posición de ninguna parte de la serpiente.

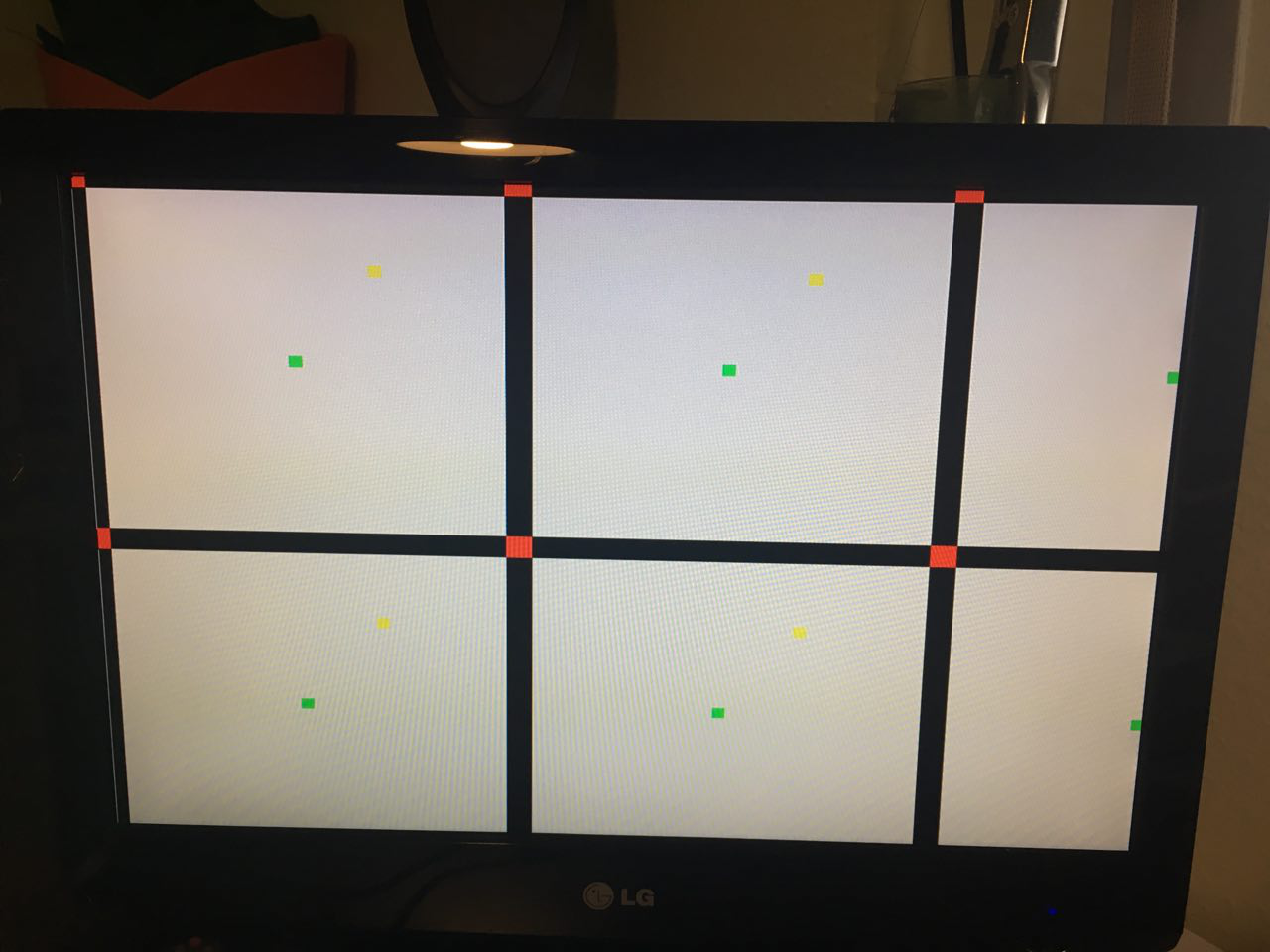
Una vez hecho esto, se escribirá la cabeza de la serpiente y se volverá al estado de *Repose* y así comenzar de nuevo eligiendo otra dirección a la que mover la serpiente.

## Representación por VGA

En esta parte se expondrá más detalladamente cuál ha sido el procedimiento para representar adecuadamente el juego por pantalla. Primeramente nos centraremos en el bloque dibuja, luego en las memorias y las imagines y por ultimo en la reprsentación aleatoria de la seta.

### Representacion de tablero (RAM)

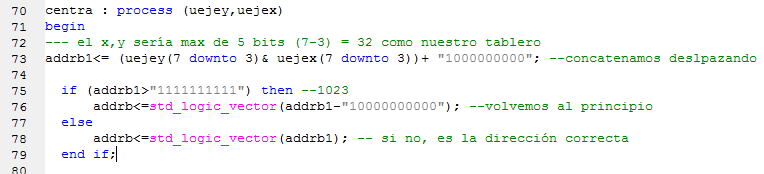
Primeramente nos marcamos como objetivo representar una posición de memoria por pantalla. El procedimiento inicial fue el explicado en clase, es decir concatenando los bits. De tal forma que la dirección a la que apunta un punto del tablero sería: . Esto conlleva el problema de que se representarían 640/32= 20 tableros en el ejex y 15 tableros en el ejey. Si vamos cogiendo solo los bits más sginificativos vamos multiplicando por dos dos el tablero. Si lo multiplicamos por 8 quedaría un tablero aceptable. Los ejes serían en vez de 4 downto 0, 7 downto 3. Seguimos teniendo el problema que aparecen más tableros y en las esquinas como se muestra en la figura:



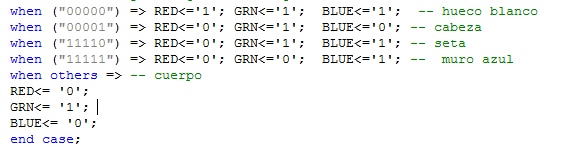
La solución adoptada en un primer momento fue hacer un bloque grande para poder representar el tamaño de tablero que quisieramos en la posición que quisieramos. Con esto podríamos igualmente representar las imágenes que quisiéramos. Cuando nos pusimos a ello nos dimos cuenta que era factible pero eran necesarios operaciones grandes de división y multiplicación. Entre otras cosas entre factores por 5 y por 3, lo que iba a suponer un aumento de la carga de cáculo.

Finalmente volvimos al planteamiento original, pero encontramos una forma más fácil de representar la pantalla a la mitad. Lo que hicimos fue desplazar las direcciones de memoria de forma que el primer pixel que se representa no es el primero si no que se encuentra a la mitad de la memoria. Hay que asegurarse que las direcciones que desborden vuelvan a cero, haciendo así una memoria cíclica. Para eliminar el resto de pantallas simplemente calculando los píxeles hacíamos una condición de poner el resto negro. Posteriormente en esta condición iría el fondo.

El código sería el siguiente:



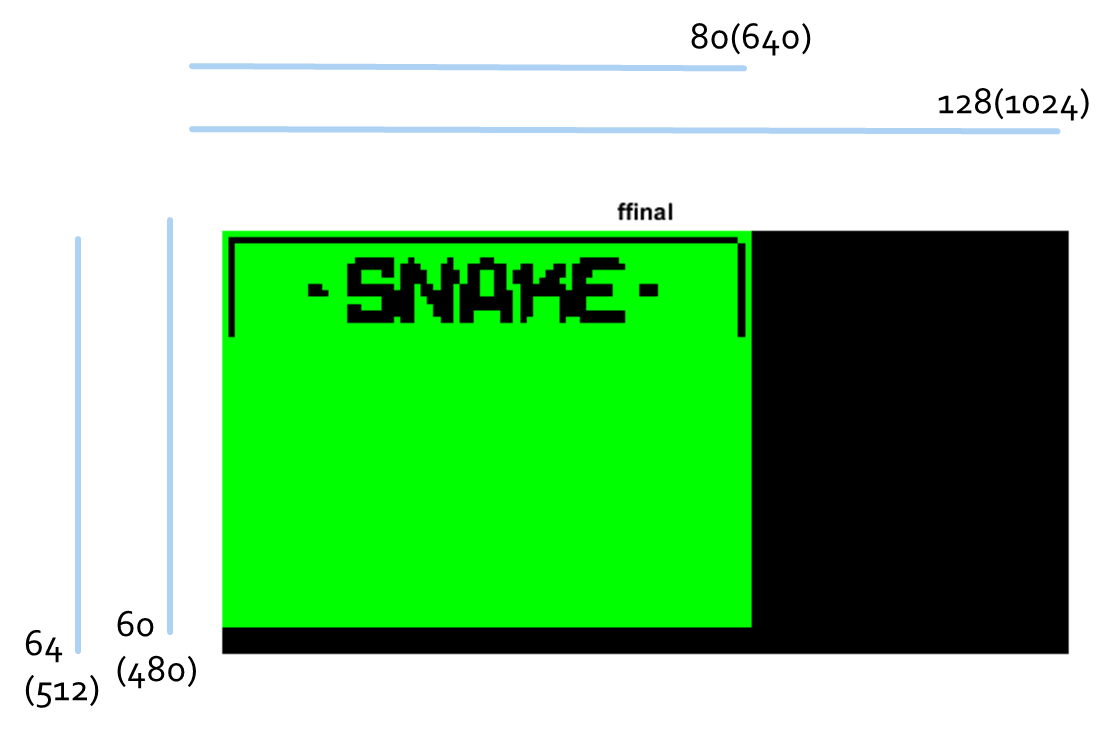
Nuestro ancho de ram era de 5 bits, lo que nos permitía alargar la serpiente hasta 30 posiciones, y los datos en la memoria estaban representados de la siguiente forma:



### Representación de las memorias ROM

Una vez que podiamos representar el tablero decidimos poner una imagen de fondo. Intentamos utilizer el coegen, proporcionado en clase. Tuvimos un problema con Java, la imagen no se cargaba. Una opción fue dibujar cuadrdito a cuadradito el fondo, cuando vimos lo que era de verdad desitimos. Por otro lado íbamos a seguir teniendo el problema de que la imagen no iba a representar exactamente lo que queríamos. O dejábamos partes en negro, o desaprovechábamos memoria. Optamos por la segunda opción ya que sólo íbamos a utilizar dos ROM.

El procedimiento era hacer una imagen con múltiplo de dos que pudiéramos representar concatenando pero teniendo en cuenta hasta donde se iba a representar realmente en la pantalla. Los múltiplos más cercanos eran 1024 para el ejex y 512 para el ejey. Esto lo dividimos entre 8 porque no es necesario representar pixel a pixel. Por lo tanto la memoria del fondo por ejemplo era de 128x64. Para el game over tuvimos que coger una de 128\*128 ya que la resolución de la imagen exigía más memoria. El procedimiento se ve más claro en la siguiente figura:



Debido a que reaizar este tipo de imagen no era muy factible con el coegen, optamos por realizar un programa en MATLAB que con la imagen. Este programa se incluye en el anexo y básicamente lo que hace es disminuir el tamaño de la imagen, rellenar de negro de 80 a 128 en horizontal y de 60 a 64 en vertical, hace un trheshold a cada canal mediante el método de Otsu y genera el fichero coe de 3 bits (RGB) ya que nuestra placa tiene solo un bit para cada color, por lo que solo podemos obtener 8 colores.

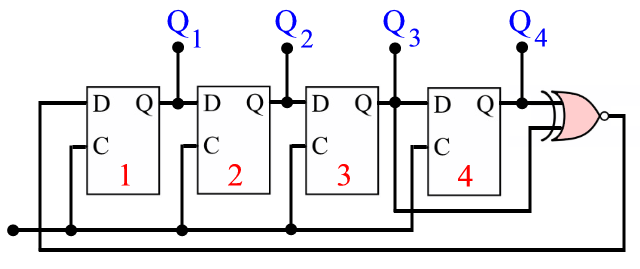
Para el caso de la pantalla de GAME OVER como hemos dicho anteriormente, utilizamos un poco más de memoria y el procedimiento es el mismo, utilizando el programa en Matlab generamos el coe. Para indicar que se ha acabado el juego y hay que cambiar la pantalla se crea un estado de game over en la máquina de estados, y se envía una señal (Fgame\_over) al bloque dibuja, que hace el cambio y representa la pantalla final.

### Representación de seta aleatoria

Para representar la seta de una forma aleatoria tuvimos varias opciones. En primer lugar, pensamos habilitar la escritua también por el lado de dibuja (lado B de la RAM), de forma que como el bloque dibuja sabe inmediatamente donde hay una seta porque la está pintando podría igualmente borrarla. Se nos ocurrió pasar una señal de la máquina de estado que indicara el momento de borrar la seta. Y ya tendríamos guardada la dirección para solo borrarla, calcular un número aleatorio y volver a pintarla. Esta vía no nos resultó muy práctica y escribir por los dos lados no nos parecía muy eficiente, así que desitimos.

En segundo lugar pensamos de incluirlo en la máquina de estados todo el proceso. Ya que había un estado en el que se comía la seta y por tanto solo quedaba incluir ahí el calculo aleatorio y pintar la seta.

Para el cálculo aleatorio nos hemos basado en la generación de combinaciones pseudo aleatorias con puertas XOR, el esquema es el siguiente:



Le incluimos una señal de habilitación para que mantenga el dato y calcule solo cuando nos sea necesario. La implementación en vhdl era mucho más sencilla de lo que nos planteamos en un principio, ya que al principio nos centramos en crear cada bloque (en nuestro caso 13) y luego nos dimos cuenta que era mucho más fácil describirlo como funcionaba el bloque entero. El código sería el siguiente:





Por último solo nos faltaba escribir otro estado en el que se cargara la dirección de la seta, se borre e ir a la nueva dirección calculada. Esperar un ciclo para ver que hay en esa posición y si era un muro o estaba en el límite del muro pues disminuir el ejex o ejey (ver siguiente figura). Si era serpiente, volver al estado de calcular el número aleatorio hasta que no lo sea.



# Anexos

## Anexo I: código Matlab

%% GENERA COE A PARTIR DE IMAGENES %%

% el tamaño queda definido en nrow,ncols

% el nombre en s\_out , acabando en .coe

% Flag = 1 threhold a cada color

% Flag = 0 threshold general (Por defecto)

% Flag = 2 threshold especifico

clear all; clc

flag=1

% tamaño recomendado para VGA / 8

ncols=80 %640/8

nrows=60 %480/8

% tamaño de pantalla

colpant=128

rowpant=64

% texto del coe

s\_out='gameover\_def.coe';

% leemos la imagen

f=imread('Snaketext.jpg');

% Re size

fres=imresize(f,[nrows ncols]);

for u=1:3

fs(:,:,u)=[fres(:,:,u) zeros(nrows,colpant-ncols)];

fs2(:,:,u)=[fs(:,:,u);zeros(rowpant-nrows,colpant)];

end

figure,imshow(fs2);

fd=double(fs2)./255;

switch flag

case 1

% Aquí se hace un threshold a cada color

level=zeros(1,3);

for i=1:3

level(i)=graythresh(fd(:,:,i));

end

fbin(:,:,1)=double(fd(:,:,1)>level(1)); % r

fbin(:,:,2)=double(fd(:,:,2)>level(2)); % g

fbin(:,:,3)=double(fd(:,:,3)>level(3)); % b

disp(255\*level);

case 0

%Aquí se hace un threshold general

level= graythresh(fd); disp(level);

fbin=double(fd>level);

case 2

level(1)=input('introduzca rlevel (0,1): ');

level(2)=input('introduzca glevel (0,1): ');

level(3)=input('introduzca blevel (0,1): ');

fbin(:,:,1)=double(fd(:,:,1)>level(1)); % r

fbin(:,:,2)=double(fd(:,:,2)>level(2)); % g

fbin(:,:,3)=double(fd(:,:,3)>level(3)); % b

% -------

end

r=fbin(:,:,1)'; rl=r(:);

g=fbin(:,:,2)'; gl=g(:);

b=fbin(:,:,3)'; bl=b(:);

Mcoe=[rl gl bl];

figure,imshow(fbin),shg

title ('ffinal');

fileID = fopen(s\_out,'w');

fprintf(fileID,';COE file generated by AGI CON MATLAB\n');

fprintf(fileID,'memory\_initialization\_radix=2;\n');

fprintf(fileID,'memory\_initialization\_vector=\n');

l=length(Mcoe);

for m=1:l

if m==l

fprintf(fileID,'%d%d%d;\n',rl(m),gl(m),bl(m));

else

fprintf(fileID,'%d%d%d,\n',rl(m),gl(m),bl(m));

end

end

fclose(fileID);

## Anexo II: Sumario de diseño del proyecto