



**DEPARTAMENTO
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Final: Super Mario en FPGA

5/12/2014

Diseño de Sistemas con FPGA

Integrante	LU	Correo electrónico
Rodriguez, Leticia Lorena	10/03	letyrodri@gmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

Índice

1. Introducción	3
2. Objetivo	3
3. Diseño	6
3.1. Top Level: Mario	6
3.1.1. Módulo vga_sync	6
3.1.2. Módulo bitmap_gen	6
4. Implementación	7
4.1. Preparación de los datos	7
4.1.1. Sprites	7
4.1.2. Mapas	7
4.2. Programación del código	8
4.2.1. MAP_ADDRESS_LOGIC	8
4.2.2. SPRITE_ADDRESS_LOGIC	8
4.2.3. PIXEL_LOGIC	9
5. APENDICE 1: Preparado de BlockRAM con imágenes usando GIMP	10
5.1. Imagen fuente	10
5.2. Paso a paso: Conversión de JPG a binario (ASCII) de 256 colores	11
5.3. Paso a paso: Conversión de binario (ASCII) de 256 colores a archivo de texto con 0 y 1 para BlockRAM (bin)	16
6. APENDICE 2: Organización de archivos adjuntos	18



Figura 1: Super Mario Bros. Pantalla del juego original.

1. Introducción

El 13 de Septiembre de 1985 se produce el lanzamiento del Super Mario Bros., un videojuego de plataformas de desplazamiento lateral que corría en las consolas de videojuegos Family Computers Famicom y Nintendo Entertainment System (NES).

El presente trabajo implementa en FPGA niveles del legendario juego.(Figura 1)

2. Objetivo

Se presenta a continuación un diseño y una implementación en FPGA de niveles ficticios del Super Mario Bros utilizando los gráficos originales del juego y la técnica de sprites.

El trabajo está preparado para que la placa FPGA sea conectada a un monitor LCD y utilice una resolución de 640 x 480 píxeles (elegida arbitrariamente).

Se busca como segundo objetivo, hacer un ahorro de memoria, por lo cual, en lugar de almacenar en la placa FPGA la imagen completa del nivel, se utilizan la técnica de sprites usadas en los videojuegos. Por lo tanto, el nivel completo se irá generando en tiempo real en base a la composición de pequeñas imágenes de 32 x 32 píxeles (llamadas sprites).

Así la pantalla de 640 x 480 píxeles quedaría virtualmente dividida en bloques de 32 x 32 píxeles, de la siguiente manera:

15 Filas de 32 píxeles de alto	Total: 480 píxeles																																	
	1	16	31	46	61	76	91	106	121	136	151	166	181	196	211	226	241	256	271	286														
	2	17	32	47	62	77	92	107	122	137	152	167	182	197	212	227	242	257	272	287														
	3	18	33	48	63	78	93	108	123	138	153	168	183	198	213	228	243	258	273	288														
	4	19	34	49	64	79	94	109	124	139	154	169	184	199	214	229	244	259	274	289														
	5	20	35	50	65	80	95	110	125	140	155	170	185	200	215	230	245	260	275	290														
	6	21	36	51	66	81	96	111	126	141	156	171	186	201	216	231	246	261	276	291														
	7	22	37	52	67	82	97	112	127	142	157	172	187	202	217	232	247	262	277	292														
	8	23	38	53	68	83	98	113	128	143	158	173	188	203	218	233	248	263	278	293														
	9	24	39	54	69	84	99	114	129	144	159	174	189	204	219	234	249	264	279	294														
	10	25	40	55	70	85	100	115	130	145	160	175	190	205	220	235	250	265	280	295														
	11	26	41	56	71	86	101	116	131	146	161	176	191	206	221	236	251	266	281	296														
	12	27	42	57	72	87	102	117	132	147	162	177	192	207	222	237	252	267	282	297														
	13	28	43	58	73	88	103	118	133	148	163	178	193	208	223	238	253	268	283	298														
	14	29	44	59	74	89	104	119	134	149	164	179	194	209	224	239	254	269	284	299														
	15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255	270	285	300														
Total: 640 píxeles																																		
20 Columnas de 32 píxeles de ancho																																		

Figura 2: División lógica de la pantalla en bloques de 32 x 32 píxeles

10001	17	00001	1
10010	18	00010	2
10011	19	00011	3
10100	20	00100	4
10101	21	00101	5
10110	22	00110	6
10111	23	00111	7
11000	24	01000	8
11001	25	01001	9
11010	26	01010	10
11011	27	01011	11
11100	28	01100	12
11101	29	01101	13
11110	30	01110	14
11111	31	01111	15
inválido		10000	16
SPRITE 2		SPRITE 1	

Figura 3: Sprites usados indexados virtualmente en forma secuencial

La selección de que imagen le corresponde a cada bloque de la pantalla, será proveída por un archivo de configuración (al cual se lo llama mapa) y que contiene un valor para cada uno de los 300 bloques de la pantalla.

Dicho valor, indica el número de sprite a usar de la imagen de sprites que se encuentra virtualmente indexada en forma secuencial. (Figura 3)

1	16	31	46	61	76	91	106	121	136	151	166	181	196	211	226	241	256	271	286
2	17	32	47	62	77	92	107	122	137	152	167	182	197	212	227	242	257	272	287
3	18	33	48	63	78	93	108	123	138	153	168	183	198	213	228	243	258	273	288
4	19	34	49	64	79	94	109	124	139	154	169	184	199	214	229	244	259	274	289
5	20	35	50	65	80	95	110	125	140	155	170	185	200	215	230	245	260	275	290
6	21	36	51	66	81	96	111	126	141	156	171	186	201	216	231	246	261	276	291
7	22	37	52	67	82	97	112	127	142	157	172	187	202	217	232	247	262	277	292
8	23	38	53	68	83	98	113	128	143	158	173	188	203	218	233	248	263	278	293
9	24	39	54	69	84	99	114	129	144	159	174	189	204	219	234	249	264	279	294
10	25	40	55	70	85	100	115	130	145	160	175	190	205	220	235	250	265	280	295
11	26	41	56	71	86	101	116	131	146	161	176	191	206	221	236	251	266	281	296
12	27	42	57	72	87	102	117	132	147	162	177	192	207	222	237	252	267	282	297
13	28	43	58	73	88	103	118	133	148	163	178	193	208	223	238	253	268	283	298
14	29	44	59	74	89	104	119	134	149	164	179	194	209	224	239	254	269	284	299
15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255	270	285	300

Figura 4: Pantalla con sprites reemplazados

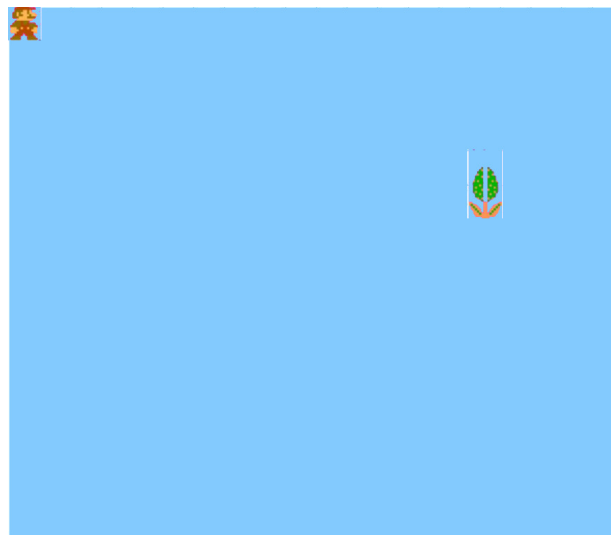


Figura 5: Pantalla con sprite y cielo reemplazados

Por ejemplo, si un mapa indica en su primer posición el valor 00010 (binario), en la posición 230 el valor 11000 y 231 el valor 11001. Lo que haría el código es reemplaza los 3 cuadrados en la pantalla. (Figura 4)

El valor 00000 en el mapa fue reservado como color del cielo y se exceptua de la técnica de sprites. Así también se ha reservado el color: 11010111 (fucsia) como transparente, es decir, cada vez que aparezca ese color dentro de un sprite será interpretado como cielo. (Figura 5)

3. Diseño

3.1. Top Level: Mario

El módulo **Mario** está compuesto por dos módulos. (Figura 6)

El primero **vga_sync** encargado de implementar la sincronización VGA y hacer posible que un pixel RGB sea transferido al puerto VGA para ser mostrado en pantalla.

Y el segundo, llamado **bitmap_gen** que se encarga de determinar el color del pixel a mostrarse en función de la posición x e y del barrido de la pantalla.

3.1.1. Módulo vga_sync

Implementación extraída del libro: FPGA Prototyping by Verilog Examples autor Pong P. Chu.

3.1.2. Módulo bitmap_gen

El módulo recibe del módulo de video el **pix_x** y **pix_y** que le corresponden a la posición actual de x e y en el barrido de la pantalla VGA, un valor **video_on** que indica si el barrido se encuentra en una parte visible de la pantalla, dos valores btn correspondientes a los botones de derecha e izquierda, clk el reloj del sistema y por último, reset.

En la Figura 7, se muestra el diseño del módulo. Vale aclarar que los submodulos **map_address_log**, **sprite_address_logic** y **pixel_logic** no son módulos de verilog, sino que las funcionalidades son implementadas dentro del módulo bitmap_gen utilizando lógica cableada (wired).

El objetivo global del módulo es emitir como salida 1 byte con el valor RGB (RRRGGBB) correspondiente, según el mapa y los sprites, para a los x e y que recibe como entrada.

Para determinar que RGB corresponde deberá:

1. Encontrar qué sprite que debe mostrar en esa posición x e y según lo que indique el mapa
2. Una vez que tiene el ID del sprite (valor de 5 bits), encontrar que pixel le corresponde dentro del sprite.

Adicionalmente, existe una entrada btn que posee 2 bits, el primer indica si se presionó el boton de la izquierda y el segundo, el de la derecha. Estos botones se encuentran conectados a un debounce. y sirven para desplazar el mapa hacia la izquierda o derecha respectivamente.

4. Implementación

4.1. Preparación de los datos

La implementación incluye tres memorias: dos correspondientes a los sprites y una correspondiente al mapa. Los valores que poseen dichas memorias son proveídos por los archivos `sprite1.bin`, `sprite2.bin` y `mapa.bin`.

Las memorias de sprite se tratan de 2 KB cada una. Cada posición de memoria tiene 8 bits y esta referenciada por una dirección de 14 bits.

En cambio, el mapa, posee en cada posición 5 bits y , dado que hay espacio para 5 mapas, el tamaño de la dirección es de 11 bits.

A continuación se explica brevemente como se confeccionaron dichos archivos.

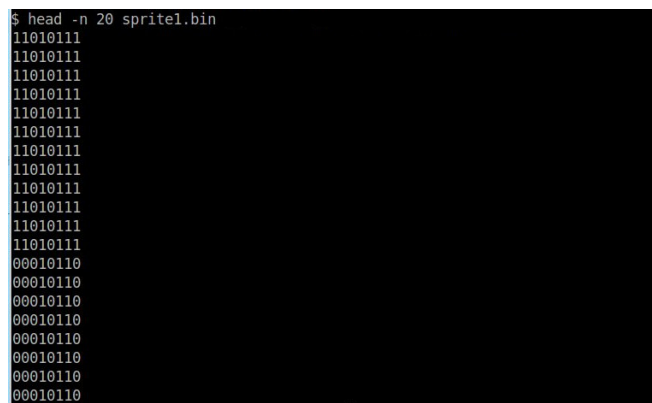
4.1.1. Sprites

Los dibujos del juego Super Mario Bros se obtuvieron desde Internet en formato de imagen de 16 bits de color.

Para armar los sprites se utilizó el software de Linux GIMP. El procesamiento que se hizo fue el siguiente:

1. seleccionar y recortar las imágenes en 32 x 32 pixeles
2. acomodar los sprites en una única imagen, situándolos uno debajo del otro hasta completar 16 sprites
3. convertir la imagen a 256 colores con la paleta de colores utilizada por Xilinx
4. guardar la imagen en datos crudos (raw)
5. procesarla con un script de python que convierta el byte en un string de 0s y 1s para que pueda ser leído por la implementación `spritex.v`.

Los archivos resultantes son `sprite1.bin` y `sprite2.bin`, y la memoria es accesible mediante los módulos `sprite1.v` y `sprite2.v`.



```
$ head -n 20 sprite1.bin
11010111
11010111
11010111
11010111
11010111
11010111
11010111
11010111
11010111
11010111
11010111
11010111
11010111
11010111
11010111
11010111
00010110
00010110
00010110
00010110
```

4.1.2. Mapas

Se armó un único archivo CSV de 20 columnas y 75 líneas (5 mapas de 15 líneas c/u) donde en cada posición se indica un valor entre 0 y 31 correspondiente al índice del sprite que corresponda que vaya en esa posición.

Luego, mediante un script elaborado en python, se convirtió este archivo en un formato que pueda ser leído por la memoria `mapa.v`.

Dicho script toma cada posición del mapa, recorriéndolo por columnas, y lo convierte a un string que representa el número binario de 5 bits.

El archivo resultante se llama: mapa.bin y los datos en memoria son accesibles desde el módulo mapa.v.

```

game: bash - Konsole <2>
$ head -n 17 mapa.bin
00000
00000
00000
00000
00000
00000
00000
00000
00000
00000
00000
00000
00000
00010
00110
00110
00110
00000
00000

```

4.2. Programación del código

4.2.1. MAP_ADDRESS_LOGIC

Dentro del módulo, bitmap_gen.v, figura la lógica para el cálculo de la dirección de memoria del mapa que se tiene que leer según la posición x e y.

Dicho cálculo es el siguiente:

```

assign addr_map_next = shift_map*15+((pix_x >>5)*15)+(pix_y >>5);

mapa mapa(.clk(clk), .en(1'b1), .addr(addr_map_next), .dataout(sprite_block));

```

donde pix_x y pix_y representan la posición actual del barrido de la pantalla (dado por el módulo vga_sync.v) y shift_map representa un registro que contiene el desplazamiento. Este desplazamiento está calculado en función a los botones de derecha e izquierda que se vayan presionando.

Para determinar el índice del mapa que corresponde a ese X e Y, los divide por 32 dado que la pantalla esta dividida en cuadrados de 32x32. Multiplica por 15 el resultado de X, que es la cantidad de líneas para posicionarse en la columna correcta. Suma ambos valores porque el mapa esta representado como una posición de memoria contigua.

Finalmente, agrega el desplazamiento de columna, X, (multiplicado por 15).

4.2.2. SPRITE_ADDRESS_LOGIC

Esta sección explica el cálculo de la dirección de memoria del sprite incluido en el módulo bitmap_gen.v.

Vale aclarar, que tanto para los sprites, como para el mapa, las direcciones de memoria calculadas se emite por el cable en forma constante y que la lógica determina cuando y cómo son usadas dichas salidas.

La salida de la memoria de MAPA se hace por el cable sprite_block en forma constante. Esta lectura de memoria, indica el número binario de 5 bits a utilizar como sprite.

Por eso, para acceder a la memoria de SPRITE, se utiliza el valor emitido por la línea sprite_block para seleccionar el sprite a utilizar.


```

assign addr_next = (pix_x % 32) + ((pix_y % 32) << 5) + (sprite_block - 1) * 32 * 32;

sprite1 sprite1(.clk(clk), .en(1'b1), .addr(addr_next), .dataout(sprite1_out));
sprite2 sprite2(.clk(clk), .en(1'b1), .addr(addr_next), .dataout(sprite2_out));

```

La fórmula muestra el cálculo que se realiza para acceder a la memoria de sprite.

Dicha fórmula, utiliza el `sprite_block` para desplazarse hasta el inicio del sprite que estamos buscando dentro de la memoria de SPRITES. Se calcula mediante el módulo de los X e Y, el desplazamiento dentro de cuadrado de 32x32 píxeles para encontrar el píxel RGB exacto a emitir en el monitor VGA para ese `pix_x` y `pix_y`.

Queda emitiendo finalmente, en los cables `sprite1_out` y `sprite2_out`, el RGB obtenido desde la memoria de SPRITE para esas posiciones X e Y según lo indicado por la memoria del MAPA.

4.2.3. PIXEL LOGIC

La última parte de la lógica, va a resolver los casos excepcionales. Por ejemplo, que se muestre el pixel del `sprite1` o `sprite2`, que se muestre cielo dado que vino el color fucsia (transparente) o que se muestre cielo porque el `sprite_block` es 00000.

```

// Devuelta del pixel
wire is_sprite2, is_sky;

assign is_sprite2      = (sprite_block > 16);
assign is_sky = (sprite_block == 5'b00000) ||
    (is_sprite2 && (sprite2_out == TRANSPARENT.COLOR)) ||
    (!is_sprite2) && sprite1_out == TRANSPARENT.COLOR);

always @*
    begin
        bit_rgb = 8'b00000000;
        if (video_on)
            begin
                if (is_sky)
                    // Bloque valor 0 muestra el celeste del cielo
                    bit_rgb = SKY_COLOR;
                else
                    if (is_sprite2)
                        bit_rgb = sprite2_out;
                    else
                        bit_rgb = sprite1_out;
                end
            end
    end

```

5. APENDICE 1: Preparado de BlockRAM con imágenes usando GIMP

Las imágenes utilizadas en el presente trabajo fueron obtuvidas desde Internet.

Dado que se encontraban en un formato distinto al requerido para ser importadas dentro de la placa Nexys Spartan 3, fueron procesadas mediante el aplicativo GIMP de Linux y scripts en python. El archivo resultante se guardó con extensión .bin y puede ser utilizado en la inicialización de BlockRAM.

A continuación se describe el mecanismo utilizado para el armado de los sprites que puede ser replicado en cualquier otro trabajo que requiera de imágenes convertidas a 256 colores guardadas en memoria.

5.1. Imagen fuente

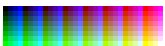
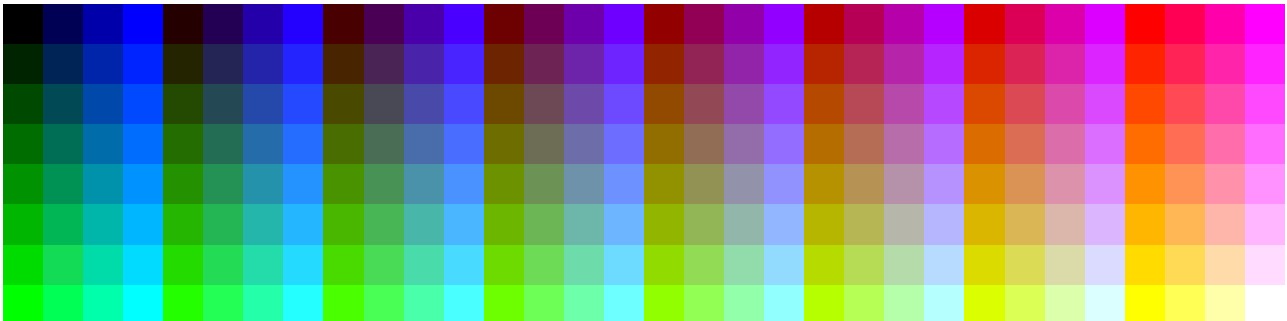
Se puede utilizar cualquier imagen, en cualquier formato que pueda ser abierto por el programa GIMP de Linux.

Lo que se quiere lograr es convertir la imagen descargada de Internet en:

- que cada pixel sea de 8 bits 3-3-2 RGB. (256 colores)
- que utilice la paleta de colores VGA usada por Nexys
- que sea exportada en dígitos binarios para ser leída por la blockRAM
- que la conversión se haga respetando los colores, es decir, la conversión de las paletas de colores tiene que respetar los colores (no tendría que pasar que se codifique un verde como rojo, por ejemplo).

La paleta VGA que utiliza la Nexys 3 es la MSX2 Screen8, obtuvida desde wikipedia en Internet:

http://en.wikipedia.org/wiki/List_of_monochrome_and_RGB_palettes



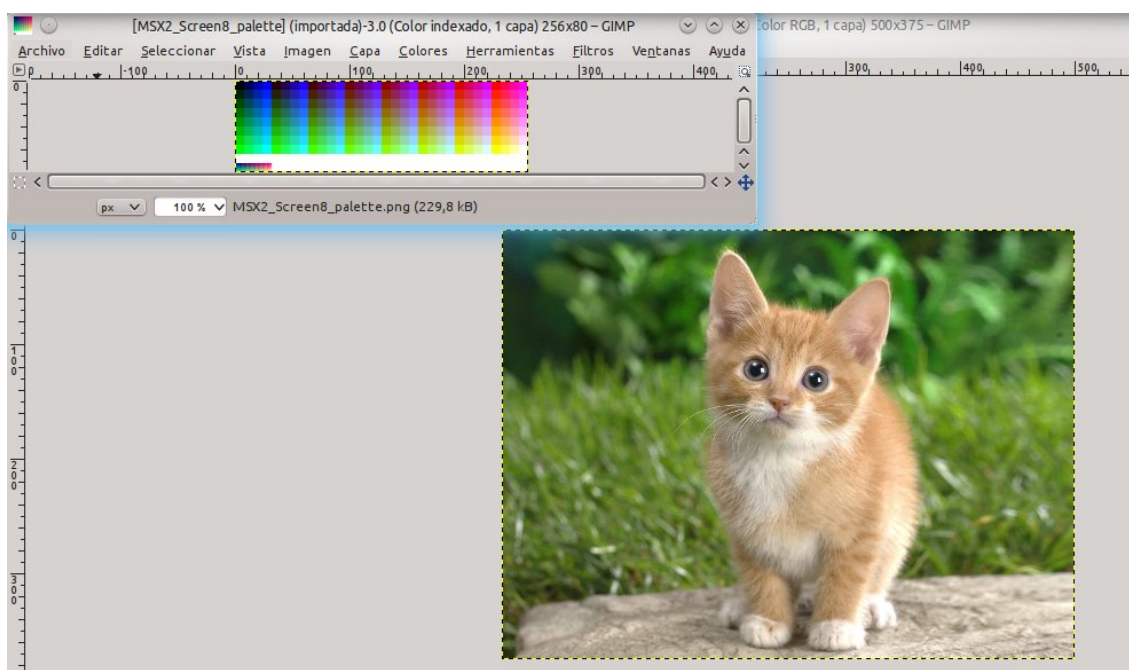
Y como ejemplo para esta guía vamos a utilizar una imagen arbitraria descargada:



Esta imagen ocupa 57,8 Kb en formato JPG y tiene una resolución de 500 x 375 píxeles.

5.2. Paso a paso: Conversión de JPG a binario (ASCII) de 256 colores

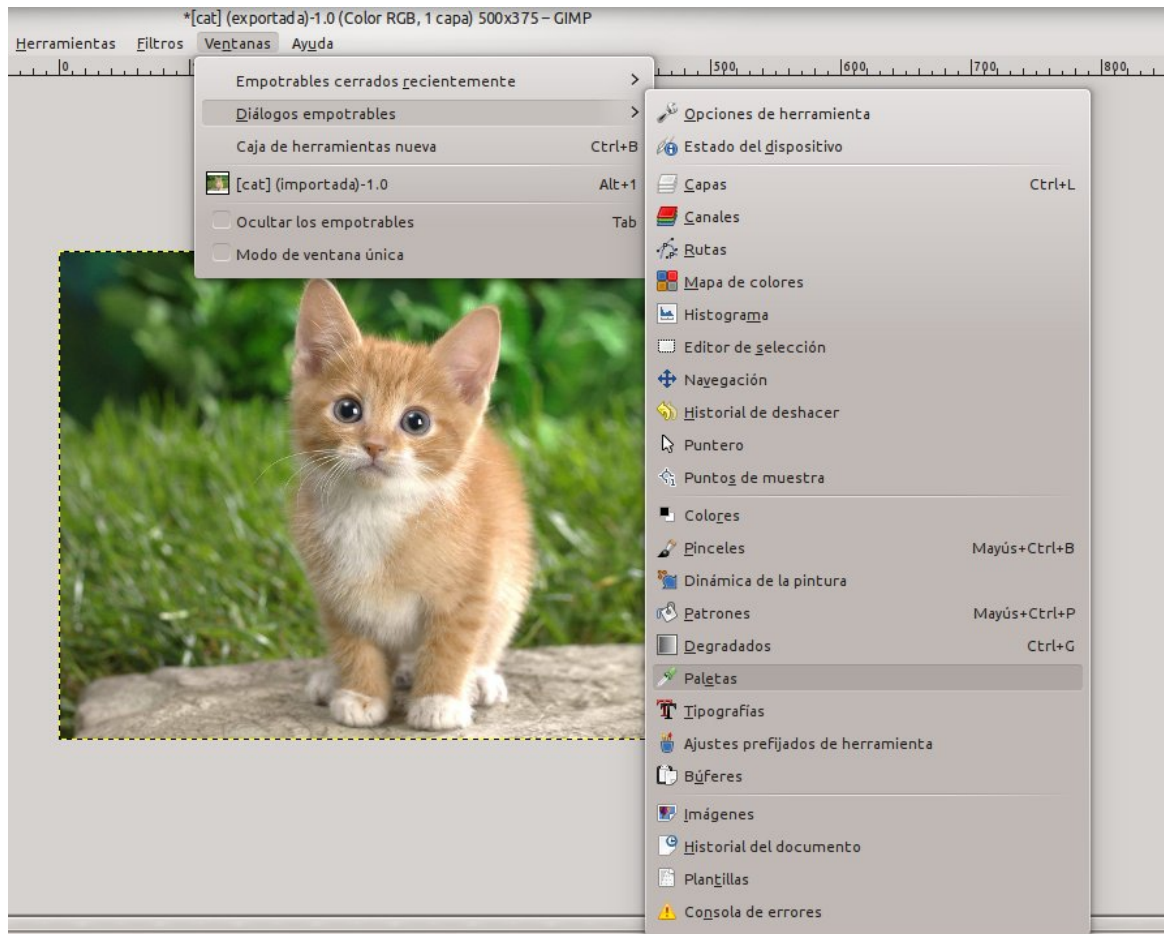
1. Abrir la imagen de paleta y la imagen del gato en gimp en GIMP



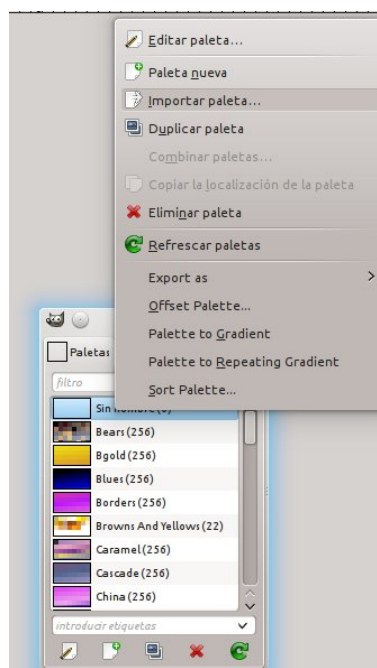
2. Importar la Paleta

Hay que trabajar sobre la imagen del gato. Vamos a usar la paleta en la conversión de la imagen del gato.

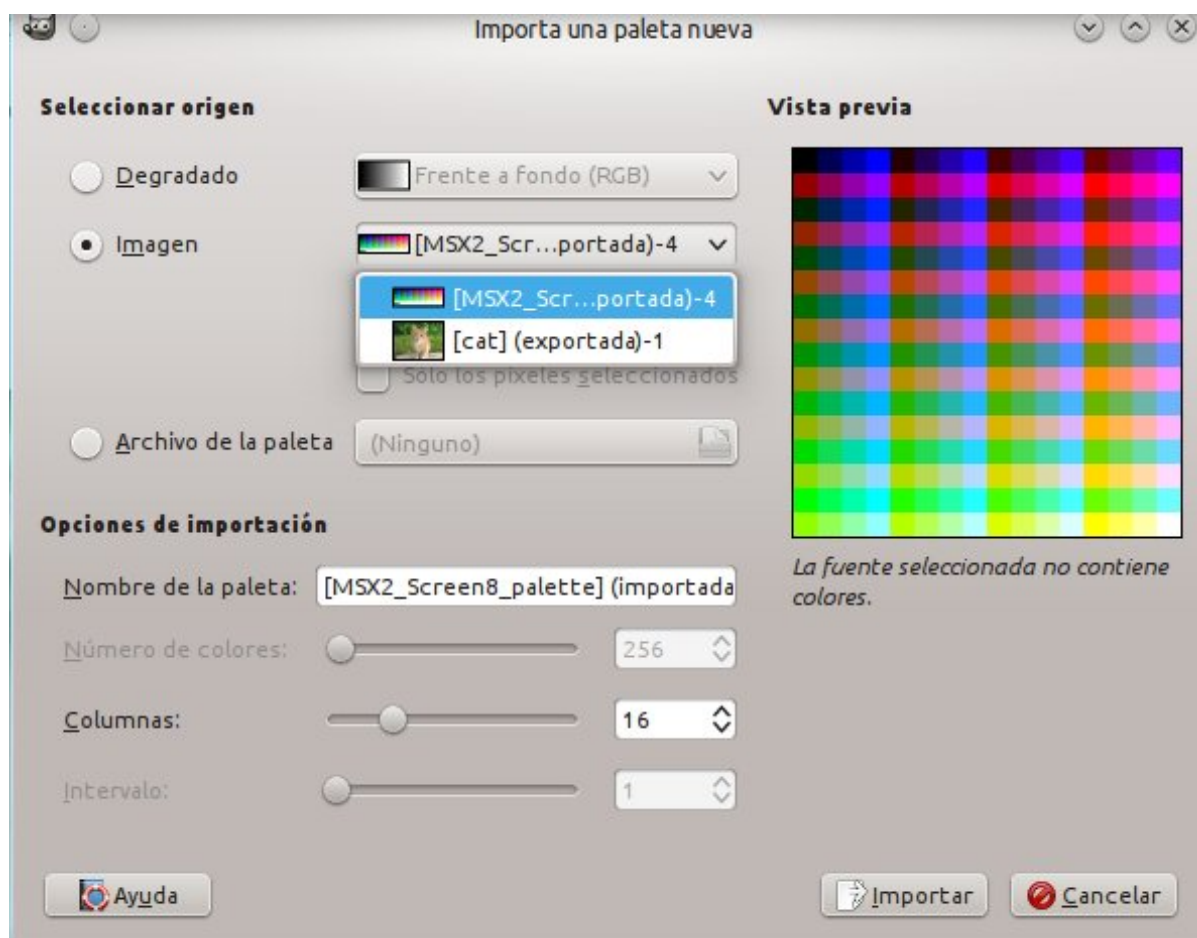
Para eso, en GIMP ir a Paletas:



Va a salir una ventana de paletas, haciendo boton derecho seleccionar Importar Paleta.



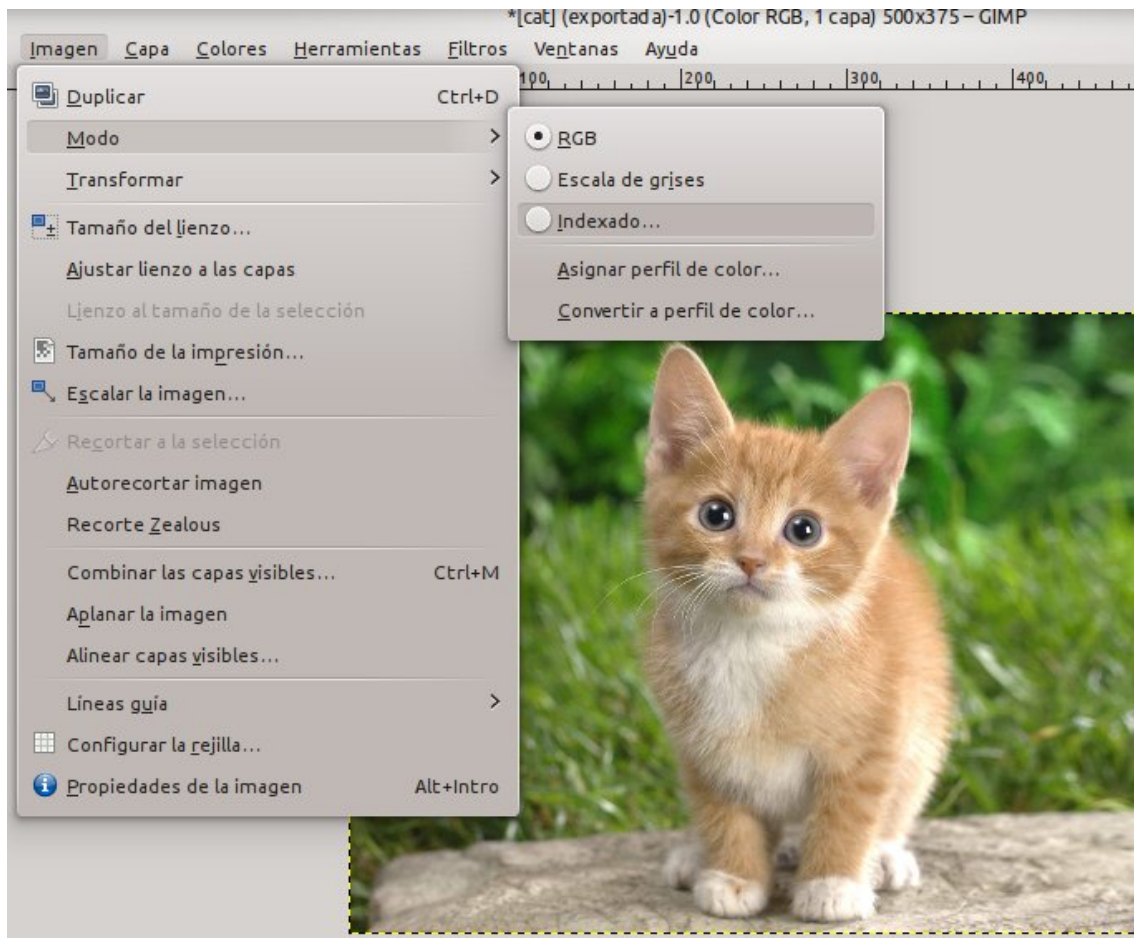
En la siguiente pantalla, seleccionar como imagen la de la paleta:



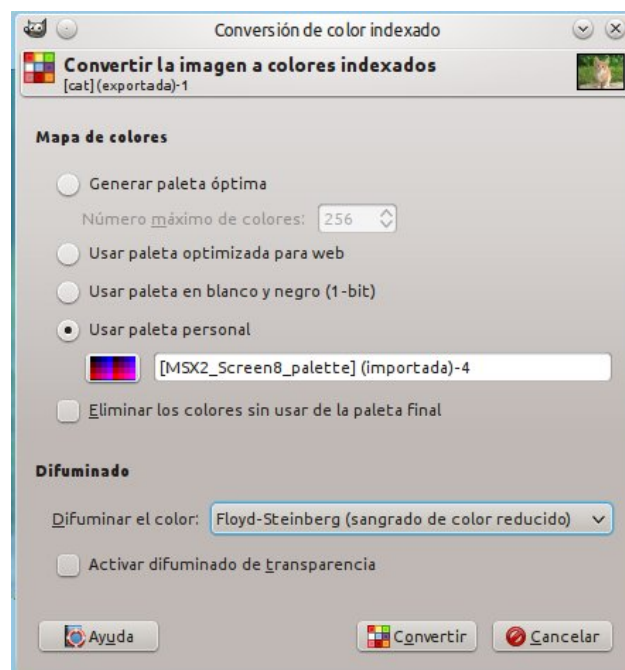
Con estos pasos, ya tenemos la paleta disponible para la conversión.

3. Convertir a RGB usando la paleta

En esta sección, cambiamos la imagen a modo indexado. Esto significa dos cosas: primero que cada pixel va a estar compuesto por un byte (va a haber 256 colores posibles) y por otro lado, los colores van a estar codificados con la paleta que importamos anteriormente.



Cuando salga la pantalla de modo indexado, debemos seleccionar la paleta que cargamos anteriormente y el algoritmo de Floyd-Steinberg.



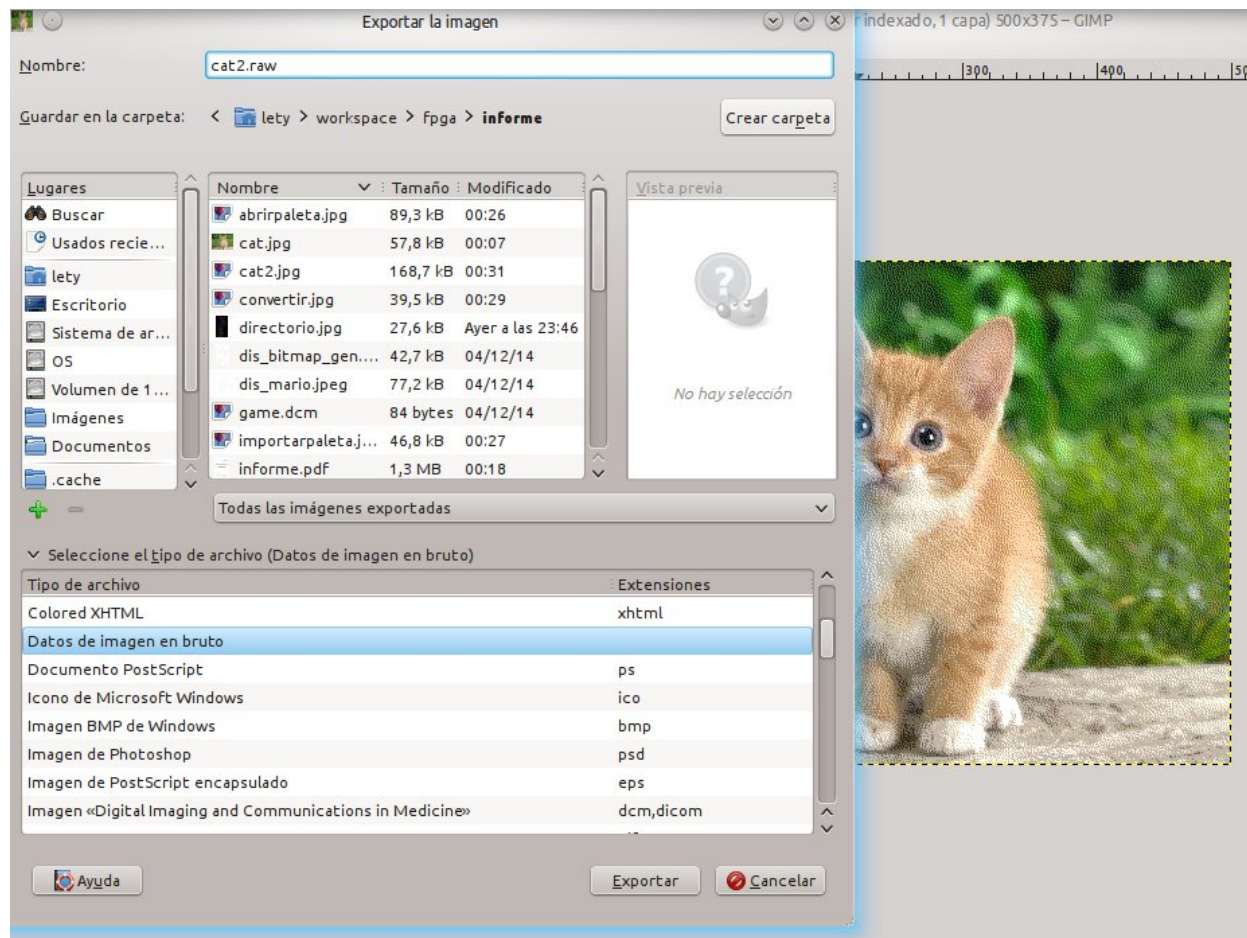
Luego de la conversión, la imagen quedará:



Notar que se percibe levemente que esta nueva imagen incluye menos colores que la original sin perder del todo la definición.

4. Salvar como datos en bruto (raw)

Por último, para obtener los bytes, el binario en forma de ASCII debemos salvar la imagen en bruto (crudo - raw). Para eso:



5.3. Paso a paso: Conversión de binario (ASCII) de 256 colores a archivo de texto con 0 y 1 para BlockRAM (bin)

La imagen en bruto para que pueda ser importada en la blockRAM usando Verilog, deberá estar convertida en texto de 0s y 1s.

Para eso, se desarrolló un código python que hace la conversión. El código es `bramit.py` y se incluye el fuente a continuación.

```
import sys

filename = sys.argv[1]

f = open(filename, "rb")
f2 = open(filename+".out", "wrb")

try:
    byte = f.read(1)
    while byte != "":
        byteStr = "{0:b}".format(ord(byte)).rjust(8, '0')
        f2.write(byteStr)
        byte = f.read(1)
finally:
    f.close()
    f2.close()
```

Se incluye también el código del programa que invierte los bits y corrige que el proceso antes detallado ya que este

deja invertidos los bits R y G.

```
import sys

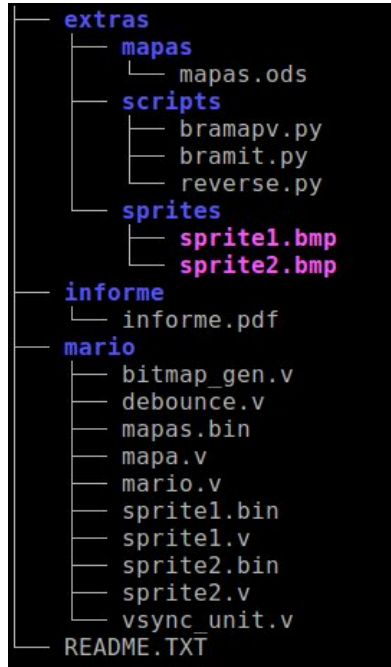
f = open(sys.argv[1], "r");
f2 = open(sys.argv[2], "w");

s=f.read(8);
while (s != ' '):
    x = s[6:8]+s[0:3]+s[3:6];
    print x
    f2.write(x+'\n');
    s = f.read(8);
f.close()
f2.close()
```

Luego de la ejecución de ambos scripts, se obtiene el binario listo para ser utilizado por la BlockRAM.

6. APENDICE 2: Organización de archivos adjuntos

Los archivos fuentes del presente trabajo se encuentran organizados de la siguiente manera:



- **mario**: archivos fuentes del código verilog y binarios para inicializar las memorias .bin.
- **informe**: el presente informe en formato PDF.
- **extras**: archivos no utilizados por el código.
 - mapas: planilla de OpenOffice para armar CSV con el mapa
 - sprites: archivos de imagen BMP fuentes que se usaron para armar los sprites
 - script: archivos de python que convierten los csv y archivos raw en archivos .bin listos para ser leídos con verilog.

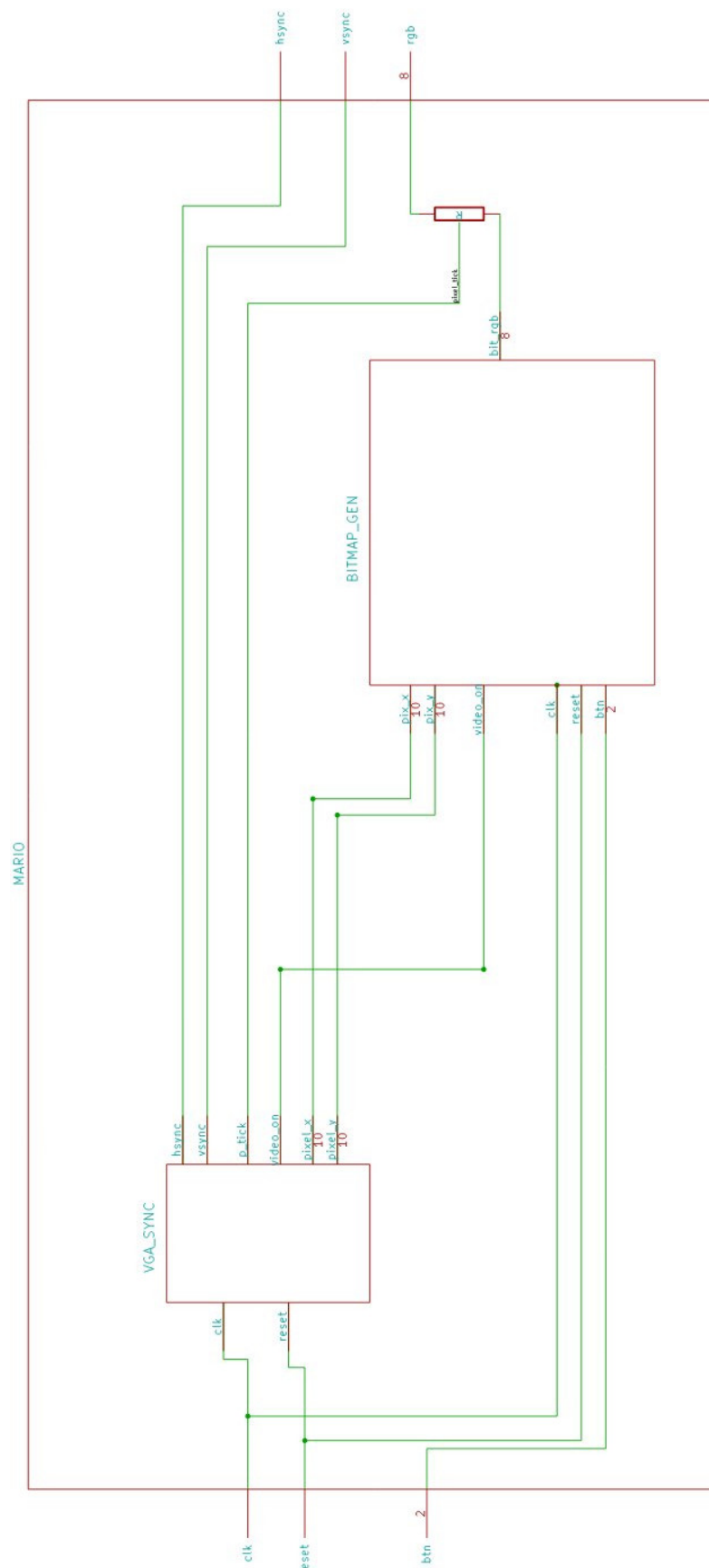


Figura 6: Diseño del Super Mario en FPGA: Módulo Mario

