Eksperymenty z FPGA (18)

Monitor interfejsu VGA

W poprzednim odcinku uruchomiliśmy wyświetlanie obrazu poprzez interfejs VGA. Uzyskane wyniki możemy sobie przypomnieć z filmu [1]. Jednak jedynym sposobem weryfikacji było uruchomienie kodu w sprzęcie i podłączenie wyświetlacza. Dzisiaj rozwiążemy tę niedogodność, tworząc monitor interfejsu VGA. Będzie on odczytywał generowany obraz i zapisywał w postaci plików graficznych. Tak jak poprzednio przed przystąpieniem do wykonywania eksperymentów zachęcam do aktualizacji repozytorium z przykładami (na przykład poprzez wywołanie polecenia git pull).

Najpierw musimy poznać sposób tworzenia plików graficznych. Użyjemy najprostszego formatu, czyli BMP [1]. Korzystając z programu MS Paint, utworzyłem testowy rysunek (*16_PONG/test.bmp*). W powiększeniu został pokazany na **rysunku 1**. W oryginale jest on bardzo mały – ma wymiary 4 na 2 piksele, ponieważ ułatwi nam to analizę jego zawartości.

Format BMP

Utworzony rysunek to plik binarny, więc możemy go otworzyć za pomocą dowolnego edytora tego typu danych (na przykład w konsoli rozkazem hexdump). Ponieważ jednak docelowo chcemy tworzyć pliki graficzne w języku System Verilog, skorzystamy od razu z niego. Na **listingu 1** został pokazany krótki kod, którego zadaniem jest otworzenie pliku i wypisanie kolejnych bajtów jego zawartości. Możemy go uruchomić w symulatorze ModelSim poleceniem:

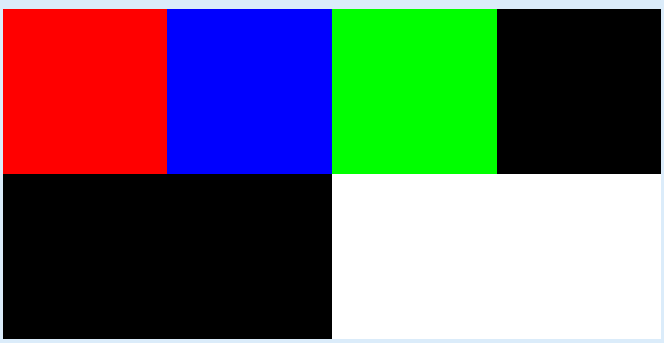
```
do ./read_hex.do
```

Jego wykonanie spowoduje wyświetlenie odczytanych bajtów, po jednym w wierszu.

Uzyskany wynik (już ręcznie sformatowany do trochę innej postaci) został pokazany na **listingu 2**. Zaraz dowiemy się, dlaczego bajty zostały pogrupowane w taki sposób. Pierwsze 14 bajtów stanowi nagłówek pliku. Ich znaczenie zostało opisane w **tabeli 1**. Pierwsze dwa z nich to tak zwany magiczny numer (*magic number*). Pozwalają one rozpoznać rodzaj pliku. Dla bitmapy są to 0x42 i 0x4d, co w kodzie ASCII oznacza litery BM.

Listing 1. Odczytywanie pliku binarnego (16_PONG/read_hex.sv)

```
10 module read_hex;
11   initial begin
12     int f, r;
13     logic [7:0]b;
14     f = $fopen("test.bmp", "rb");
15     while ($fread(b, f))
16       $display("%x", b);
17     $fclose(f);
18   end
19 endmodule
```



Rysunek 1. Testowy plik 16_PONG/test.bmp

Kolejne cztery to rozmiar pliku. Jednak jak widzimy w tabeli, czeka tu na nas mała pułapka. Kolejność bajtów to: od najmniej znaczącego do najbardziej znaczącego. Oznacza to, że musimy czytać je od końca (bajty nie bity). Z tego powodu rozmiar to nie 0x4e000000, co dałoby 1308622848, ale 0x4e, czyli 78. Zasada ta dotyczy wszystkich wielobajtowych pól w naszym pliku.

Następne cztery bajty są zarezerwowane (niektóre programy mogą ich używać). U nas są one równe 0. Pozostałe bajty to odstęp pomiędzy początkiem pliku a początkiem tablicy pikseli – u nas jest to 54. To oznacza, że pomiędzy nagłówkiem a danymi jest jeszcze 40 bajtów. Zajmiemy się nimi za chwilę. Na razie spójrzmy na **listing 3**, gdzie znajdziemy strukturę opisującą nagłówek. Użyjemy jej później podczas odczytywania i zapisywania plików. Aby móc jej używać w różnych projektach, została ona umieszczona w pakiecie (*package*) bmpPkg.

Wróćmy do analizowania zawartości pliku. Kolejne 40 bajtów to tak zwany nagłówek DIB. Przechowuje on dodatkowe informację o danych. Istnieje wiele jego wersji. W naszym przypadku jest to BITMAPINFOHEADER. Listę wszystkich jego pól znajdziemy w **tabeli 2**. Dla nas najważniejsze to biWidth i biHeight, które opisują szerokość i wysokość obrazu. Kolejnym polem, które będziemy modyfikować, jest biSizeImage, czyli rozmiar danych. Dla pozostałych zastosujemy te same wartości, które znaleźliśmy w pliku testowym.

Listing 2. Zawartość pliku 16_PONG/test.bmp

```
42 4d 4e 00 00 00 00 00 00 00 36 00 00 00

28 00 00 00 04 00 00 00 02 00 00 00 01 00 18 00 00 00 00 00
18 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00 00 00 00 00 00 ff ff ff ff ff ff
05 0a ff ff 05 0a 0a ff 05 00 00 00
```

Tabela 1. Elementy nagłówka pliku

Element	Rozmiar	W danych	Znaczenie
„Magic number”	2 bajty	42 4d	BM
Rozmiar pliku	4 bajty	4e 00 00 00	0x4e = 78
Zarezerwowane	4 bajty	00 00 00 00	
Przesunięcie do tablicy pikseli	4 bajty	36 00 00 00	0x36 = 54

Tak jak wcześniej, dla drugiego nagłówka także tworzymy odpowiadającą mu strukturę. Znajdziemy ją na **listingu 4**.

Zostało nam jeszcze ostatnie 24 bajty. Jak łatwo się domyślić to muszą w końcu być nasze dane. Trzy kolejne bajty opisują kolor jednego piksela: pierwszy z nich oddaje kolor niebieski, drugi zielony, a trzeci czerwony. Pierwsza trójka odpowiada pikselowi z lewego dolnego rogu obrazu. Następnie znajdziemy kolejne piksele z dolnego wiersza, aby później przejść do początku wiersza, który znajduje się nad nim. Opis mógł wyjść zawiły, ale wszelkie wątpliwości powinien rozwiać **rysunek 2**.

Tabela 2. Nagłówek bitmap info header

Nazwa	Funkcja	Rozmiar [bajty]	W danych	Znaczenie
biSize	Wilekość nagłówka	4	28 00 00 00	0x28 = 40
biWidth	Szerokość obrazu, px	4	04 00 00 00	0x04 = 4
biHeight	Wysokość obrazu, px	4	02 00 00 00	0x02 = 2
biPlanes	Liczba warstw kolorów	2	01 00	0x01 = 1
biBitCount	Liczba bitów na px	2	18 00	0x18 = 24
biCompression	Kompresja	4	00 00 00 00	Brak
biSizeImage	Rozmiar rysunku	4	18 00 00 00	24
biXPelsPerMeter	Rozdzielczość pozioma	4	00 00 00 00	Brak
biYPelsPerMeter	Rozdzielczość pionowa	4	00 00 00 00	Brak
biClrUsed	Liczba kolorów w palecie	4	00 00 00 00	Brak
biClrImportant	Ważne kolory w palecie, stosowane przy animacji	1	00	Brak
biClrRotation	Rotacja pakiety, stosowane przy animacji	1	00	Brak
biReserved	Zarezerwowane	2	00 00	

Listing 3. Struktura opisująca nagłówek pliku BMP (16_PONG/bmp_pkg.sv)

```
15 typedef struct packed {
16   logic [15:0]mn;
17   logic [31:0]size;
18   logic [31:0]res;
19   logic [31:0]offset;
20 } bmp_header;
```

Listing 4. Struktura opisująca nagłówek pliku bitmap info header (16_PONG/bmp_pkg.sv)

```
22 typedef struct packed {
23   logic [31:0] biSize;
24   logic [31:0] biWidth;
25   logic [31:0] biHeight;
26   logic [15:0] biPlanes;
27   logic [15:0] biBitCount;
28   logic [31:0] biCompression;
29   logic [31:0] biSizeImage;
30   logic [31:0] biXPelsPerMeter;
31   logic [31:0] biYPelsPerMeter;
32   logic [31:0] biClrUsed;
33   logic [ 7:0] biClrImportant;
34   logic [ 7:0] biClrRotation;
35   logic [15:0] biReserved;
36 } bmp_info_header;
```

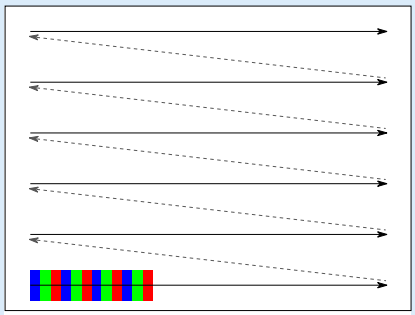
Odczytywanie pliku BMP

Rozłożyliśmy już bitmapę na części pierwsze. Teraz możemy pokusić się o utworzenie zadania (*task* – można rozumieć jako funkcję, która nie zwraca wartości), które załaduje i zinterpretuje plik BMP. Jej fragment został pokazany na **listingu 5**. Przyjmuje on jeden argument: fname, czyli ścieżkę do pliku, który ma zostać załadowany. Następnie w liniach 39...43 definiujemy zmienne. Najciekawsze z nich to nasze struktury reprezentujące nagłówki bh oraz bih. Trzybajtowa tablica c posłuży do odczytywania pojedynczego piksela.

W wierszu 46 odczytujemy nagłówek i zapisujemy go w zmiennej bh. Następnie w kolejnych pięciu liniach wyświetlamy jego kolejne pola. Do zmiany kolejności bajtów w słowach używamy operatora strumieniowego {<<8{}}. Niestety nie jest on jeszcze wspierany w używanej przeze mnie wersji Quartus, więc obecnie nie możemy go użyć do syntezy. Jednak ModelSim sobie z nim radzi. Nic nie stoi na przeszkodzie użycia go w symulacji. Odczyt drugiego nagłówka jest analogiczny, dlatego nie znajduje się na listingu. Do odczytania tablicy pikseli używamy dwóch pętli for. Zewnętrzna iteruje po wierszach, a wewnętrzna po kolumnach. Odczytujemy po 3 bajty naraz i zapisujemy w zmiennej c. Następnie uzyskane dane są wypisywane na ekran.

Aby przetestować nasz kod, przygotowałem bardzo krótki moduł bmp_open. Jak widzimy na **listingu 6**, składa się on jedynie z bloku initial, w którym wywołujemy nasze zadanie. Ponieważ blok składa się z pojedynczej linii, nie musimy dodawać słów begin end. Uruchamiamy go poleceniem: `do ./bmp_open.do`.

Fragment uzyskanego wyniku prezentuje **listing 7**. Warto porównać uzyskany efekt z tym z listingu 2. Cenne będą także eksperymenty z własnymi plikami BMP.



Rysunek 2. Kolejność danych w pliku bmp

Tworzenie pliku BMP

To, co nas najbardziej interesuje, to zapisywanie zebranych danych do plików graficznych. Poprzednie eksperymenty miały nas jedynie zapoznać ze strukturą pliku. Dopiero teraz przygotujemy to, co jest nam najbardziej potrzebne. Najciekawsze fragmenty kodu znajdziemy na **listingu 8**. Jak widzimy w linii 79, nasze zadanie przyjmuje tym razem cztery parametry: ścieżkę do pliku, wymiary obrazu (szerokość i wysokość) oraz referencję do dynamicznej tablicy pikseli. Następnie deklarujemy zmienne pomocnicze. Poza naszymi strukturami tworzymy tu także tablicę bajtów o odpowiadającej im długości. Zanim ją wykorzystamy, najpierw wypełnimy nagłówki. W wierszu 87 obliczamy rozmiar pliku, następnie w liniach 89...92 uzupełniamy nagłówek. W wierszu 94 wyjaśnia się,

Listing 5. Odczytywanie zawartości pliku BMP (16_PONG/bmp_pkg.sv)

```
38 task show(string fname);
39   int a1, f, i;
40   logic [7:0]b;
41   bmp_header bh;
42   bmp_info_header bih;
43   logic [2:0][7:0]c;
44   a1 = $fopen(fname, "rb");
45
46   f = $fread(bh, a1);
47   $display("BMP header:");
48   $display("Magic number: 0x%x", bh.mn);
49   $display("size: %d", {<<8{bh.size}});
50   $display("res: %d", {<<8{bh.res}});
51   $display("offset: %d", {<<8{bh.offset}});

69   for (int y=0; y<{<<8{bih.biHeight}}; y++) begin
70     for (int x=0; x<{<<8{bih.biWidth}}; x++) begin
71       f = $fread(c, a1);
72       $display("x: %d, y: %d, r: %d, g: %d, b: %d", x, y, c[0], c[1], c[2]);
73     end
74   end
75   $fclose(a1);
77 endtask
```

Listing 6. Testowy kod odczytujący plik test.bmp (16_PONG/bmp_open.sv)

```
10 module bmp_open;
11
12 initial
13   bmpPkg::show("test.bmp");
14
15 endmodule
```

Listing 7. Fragment odczytanego pliku BMP

```
# BMP header:
# Magic number: 0x424d
# size:         78
# res:          0
# offset:         54

# x:           0, y:           0, r:   0, g:   0, b:   0
# x:           1, y:           0, r:   0, g:   0, b:   0
# x:           2, y:           0, r: 255, g: 255, b: 255
# x:           3, y:           0, r: 255, g: 255, b: 255
# x:           0, y:           1, r: 255, g:  10, b:   5
# x:           1, y:           1, r:  10, g:   5, b: 255
# x:           2, y:           1, r:   5, g: 255, b:  10
# x:           3, y:           1, r:   0, g:   0, b:   0
```

Listing 8. Zapisywanie tablicy pikseli do pliku BMP (16_PONG/bmp_pkg.sv)

```
079 task automatic save(string fname, int width,
        int height, ref logic [2:0][7:0]c[][]);
080   int f;
081   bmp_header bh;
082   bmp_info_header bih;
083   logic [31:0] size;
084   logic [7:0] bh_b[13:0];
085   logic [7:0] bih_b[39:0];
086
087   size = $bits(bh)/8 + $bits(bih)/8 + width*height*3;
088
089   bh.mn = 16'h424d;
090   bh.size = {<<8{size}};
091   bh.res = '0;
092   bh.offset = {<<8{32'd54}};
093
094   {>>{bh_b}} = bh;

112   f = $fopen(fname, "wb");
113   for (int i=0; i < 14; i++)
114     $fwrite(f, "%c", bh_b[13-i]);

117
118   for (int y = 0; y < height; y++)
119     for (int x = 0; x < width; x++)
120       for (int z = 0; z < 3; z++)
121         $fwrite(f, "%c", c[y][x][2-z]);
122
123   $fclose(f);
124 endtask
```

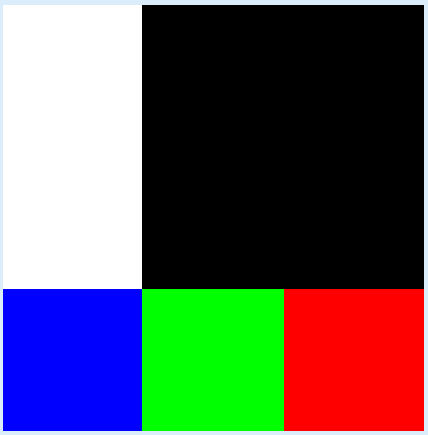
po co nam dodatkowa tablica bajtów. Zapisujemy w nim nasz nagłówek w odwróconej kolejności bitów. Tu znowu z pomocą przychodzi nam operator strumieniowy. Na końcu, w pętli for, bajt po bajcie wpisujemy nagłówek do pliku. Używamy tu opcji %c polecenia fwrite. Obsługa drugiego nagłówka jest analogiczna. Wpisywanie tablicy pikseli także następuje bajt po bajcie. Użyte są tu 3 pętle: wysokość, szerokość oraz kolory (wiersze 118...121). Na zakończenie plik jest zamykany (wiersz 123).

Do pierwszych testów użyjemy modułu bmp_save. Jego fragmenty zostały pokazane na **listingu 9**. Na początku definiujemy zmienne, następnie ustalamy wymiary rysunku na 4×3 piksele. Ciekawe rzeczy dzieją się w liniach 19...21. Tutaj tworzymy dynamiczną

Listing 9. Tworzenie pliku BMP (16_PONG/bmp_save.sv)

```
10 module bmp_save;
11
12   initial begin
13     int width, height;
14     logic [2:0][7:0]c[][];
15
16     width = 4;
17     height = 3;
18
19      c = new[height];
20      for (int y = 0; y < height; y++)
21        c[y] = new[width];
22
23      c[0][0] = {8'd255, 8'd0, 8'd0};
24      c[0][1] = {8'd0, 8'd255, 8'd0};
25      c[0][2] = {8'd0, 8'd0, 8'd255};
26      c[0][3] = {8'd255, 8'd255, 8'd255};

38      bmpPkg::save("save_test.bmp", width, height, c);
39    end
40  endmodule
```



Rysunek 3. Wygenerowana grafika

dwuwymiarową tablicę. Jest to tak naprawdę tablica tablic, dlatego jej tworzenie wymaga kilku kroków. Najpierw operatorem new tworzymy tablice wierszy. Następnie w pętli for inicjujemy każdy wiersz jako tablicę pikseli. Kolejną częścią jest jej wypełnienie. Jak widzimy, dostęp do niej jest typowy. Używamy po prostu operatorów []. Tak docieramy do linii 38, gdzie wywołujemy nasze zadanie. Cały program można uruchomić poleceniem:

```
do ./bmp_save.do
```

Uzyskana grafika powinna być podobna do tej z **rysunku 3**.

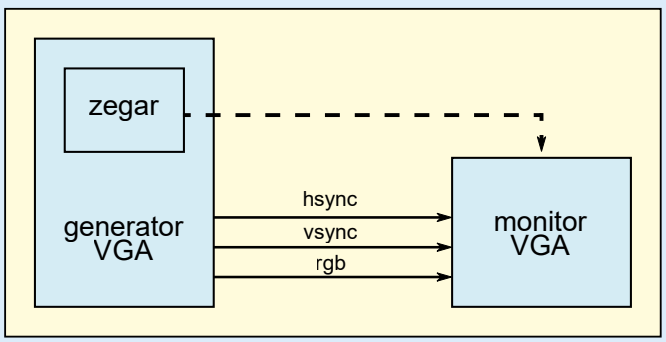
Monitor interfejsu VGA

Jesteśmy już gotowi do implementacji naszego dzisiejszego projektu – monitora VGA. Na **rysunku 4** zostało pokazuje jego połączenie z generatorem sygnału VGA (przygotowanym w poprzednim odcinku). Stosujemy tu małe oszustwo. Oryginalny sygnał VGA nie ma zegara, numer kolumny należy odzyskać na podstawie odstępów pomiędzy kolejnymi sygnałami hsync. My jednak możemy po prostu wyciągnąć zegar z generatora i na jego podstawie ustalić położenie odbieranego piksela. Implementację monitora prezentuje **listing 10**.

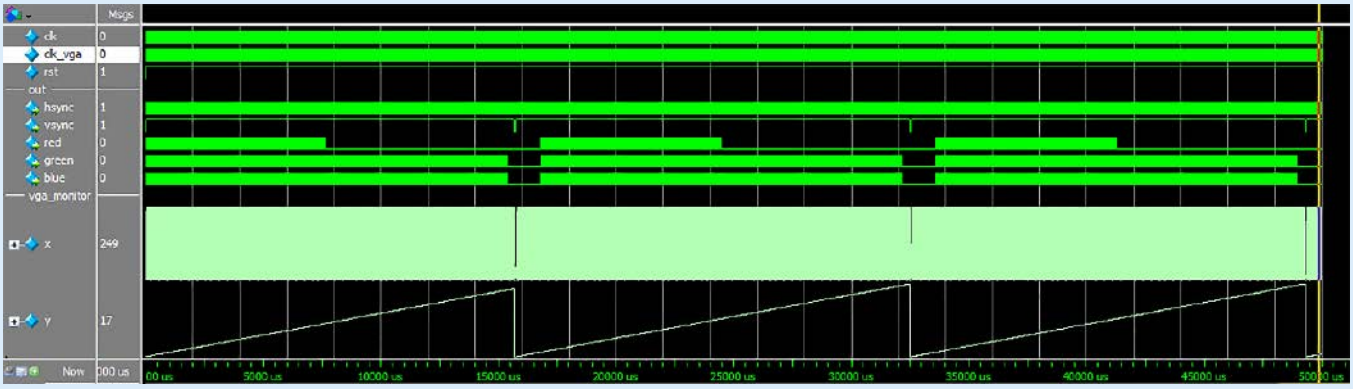
Nasz moduł przyjmuje jeden parametr: NAME. Określa on nazwę tworzonego pliku. Wejścia są standardowe: zegar, reset oraz sygnał VGA: synchronizacja i dane. W liniach 19...22 zdefiniujemy pomocnicze parametry. Nasz monitor będzie obsługiwał jedynie rozdzielczość 640 na 480, ale właśnie poprzez modyfikację tych parametrów możemy to zmienić. Parametry V i H oznaczają wysokość i szerokość, a V_offset i H_offset odległość początku danych od końca sygnału synchronizującego.

W bloku initial (28...32) inicjujemy tablicę, w której będziemy zapisywać kolejne piksele. Natomiast cała logika modułu znajduje się w bloku always. Na początku (linie 35...41) obsługiwany jest reset. Następnie sprawdzamy, czy sygnał synchronizacji pionowej jest aktywny. Jeżeli tak, to sprawdzamy, czy w poprzednim kroku nie był jeszcze nieaktywny (stan wysoki). Jeżeli wykryjemy zbocze opadające, wywołamy funkcję zapisującą ramkę do pliku i inkrementujemy jej numer. Następnie wyzerujemy licznik wierszy i kolumn.

Kolejny krok to wykrycie synchronizacji poziomej. Tutaj wykrycie zbocza opadającego inkrementuje licznik wierszy. Ostatni przypadek to taki, gdy żaden z sygnałów synchronizujących nie jest aktywny. Wtedy po prostu inkrementujemy licznik kolumn. W wierszach 58...61 następuje zapis stanu piksela do tablicy. Dzieje się to tylko



Rysunek 4. Połączenie monitora z generatorem sygnału



Rysunek 5. Przebiegi czasowe z symulacji monitora

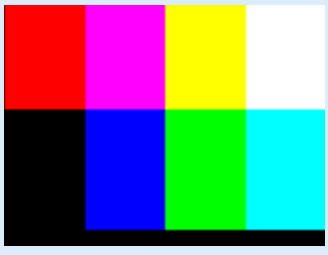
wtedy, gdy oba liczniki znajdują się wewnątrz obszaru ekranu. Ponieważ w interfejsie VGA wiersze są przesyłane od góry, a w plikach BMP są zapisywane od dołu, musimy zamienić ich kolejność.

Analizując powyższy kod należy mieć na uwadze, że jest to kod napisany jedynie do symulacji. Nie musimy tutaj zastanawiać się, czy będzie on syntezowalny. Aby wykonać symulację, potrzebujemy jeszcze testbenchu. Jego fragment został zaprezentowany na **listingu 11**. Jest on klasyczny: po prostu łączymy wyjścia z wejściem. Warto tu zwrócić uwagę na notację dut.clk_vga, za pomocą której uzyskujemy dostęp do sygnałów z wnętrza modułu dut. Symulację uruchamiamy rozkazem:

```
do ./vga_top.do
```

Trwa ona dłuższą chwilę. Gdy się zakończy powinniśmy zobaczyć wynik podobny do tego z **rysunku 5**. Widzimy, że wygenerowane zostały trzy ramki. W folderze projektu powinniśmy znaleźć trzy nowe pliki *fame0.bmp*, *frame1.bmp* oraz *frame2.bmp*. Pierwszy z nich został pokazany na **rysunku 6**, a drugi na **rysunku 7**.

Widzimy, że na pierwszej ramce brakuje początkowych wierszy. Wynika to z faktu, że sygnał synchronizujący generujemy dopiero po ramce. Dlatego ten błąd jest spodziewanym efektem. Dopiero druga (i kolejne) ramki złapane przez monitor są poprawne. Warto ją porównać do obrazu z ekranu uzyskanego w poprzednim odcinku.



Rysunek 6. Pierwsza grafika wygenerowana w symulacji

Rysunek 7. Druga grafika wygenerowana w symulacji

Podsumowanie

W tym odcinku było bardzo teoretycznie – same symulacje, bez eksperymentów w sprzęcie. Ale już w kolejnym odcinku użyjemy naszego monitora. Pomoże nam przy implementacji najbardziej klasycznej z gier: PONG-a.

Rafał Kozik
rafkozik@gmail.com

[1] https://bit.ly/3vC0waG
[2] https://bit.ly/3eOZeT4

Listing 10. Implementacja monitora VGA (16_PONG/vga_monitor.sv)

```
10 module vga_monitor #(
11   parameter NAME = "frame"
12 ) (
13   input wire clk,
14   input wire rst,
15   input wire [2:0]c,
16   input wire hsync,
17   input wire vsync
18 );
19   parameter V = 480;
20   parameter V_offset = 33;
21   parameter H = 640;
22   parameter H_offset = 48;
23   int frame;
24   logic [2:0][7:0]ct[][];
25   int x, y;
26   logic last_vsync, last_hsync;
27
28   initial begin
29     ct = new[V];
30     for (int iy = 0; iy < V; iy++)
31       ct[iy] = new[H];
32     end
33
34   always @(posedge clk) begin
35     if (!rst) begin
36       x = 0;
37       y = 0;
38       frame = 0;
39       last_vsync = 1'd0;
40       last_hsync = 1'd0;
41     end else begin
42       if (vsync == 1'd0) begin
43         if (last_vsync == 1'd1) begin
44           bmpPkg::save($sformatf("%s_%02d.bmp", NAME, frame), H, V, ct);
45           frame = frame + 1;
46         end
47         x = 0;
48         y = 0;
49       end else if (hsync == 1'd0) begin
50         x = 0;
51         if (last_hsync == 1'd1)
52           y = y + 1;
53       end else begin
54         x = x + 1;
55       end
56       last_vsync = vsync;
57       last_hsync = hsync;
58       if ((x >= H_offset && x < (H + H_offset)) &&
             (y >= V_offset && y < (V + V_offset))) begin
59         for (int cid = 0; cid < 3; cid++)
60           ct[V-1-(y-V_offset)][x-H_offset][cid] = c[2-cid] ? 8'd255 : 8'd0;
61       end
62     end
63   end
64
65 endmodule
```

Listing 11. Połączenie monitora z generatorem (16_PONG/vga_top_tb.sv)

```
11 module vga_top_tb;

28 vga_top dut (
29 .clk(clk),
30 .rst(rst),
31 .hsync(hsync),
32 .vsync(vsync),
33 .red(r),
34 .green(g),
35 .blue(b));
36
37 vga_monitor monitor (
38 .clk(dut.clk_vga),
39 .rst(rst),
40 .c({r, g, b}),
41 .hsync(hsync),
42 .vsync(vsync));
43
44 endmodule
```