Układy cyfrowe i systemy wbudowane 2

PROJEKT

prowadzący: dr inż. Jarosław Sugier

Termin zajęć:

czwartek TN 8:00 – 11:00

Skład grupy:

Piotr Markiewicz

Bartosz Pogoda

Spis treści

1. Wprowadzenie

1.1. Cel i zakres

Celem projektu było zaprojektowanie oraz zaimplementowanie przy użyciu języka VHDL prostej gry działającej na układzie Spartan-3E (XC3S500E). Sterowanie w grze miało odbywać się za pomocą myszki komputerowej, a wyświetlanie obrazu gry przy użyciu monitora VGA.

Projekt został przeprowadzony w sposób iteracyjny, z podziałem na etapy mające na celu stopniowe poznawanie potrzebnych technologii i ich zastosowanie w projekcie. Można wyróżnić następujące etapy:

1. Nawiązanie komunikacji z monitorem VGA,
2. nawiązanie komunikacji z myszą komputerową (PS/2),
3. określenie oraz implementacja mechaniki gry.

Każdemu etapowi towarzyszyły testy, mające na celu sprawdzenie, czy kolejne funkcjonalności zostały zrealizowane poprawnie oraz weryfikację grywalności gry.

1.2. Mechanika gry

Podczas projektu powstała gra zręcznościowa, w której gracz za pomocą ruchów myszy steruje „duszkiem” umieszczonym w prostokątnym obszarze w centralnej części ekranu.

1.2.1. Cel gry

Gracz ma na celu omijanie zbliżających się ku niemu pocisków, które są losowo generowane na krawędziach ekranu.

1.2.2. Życia

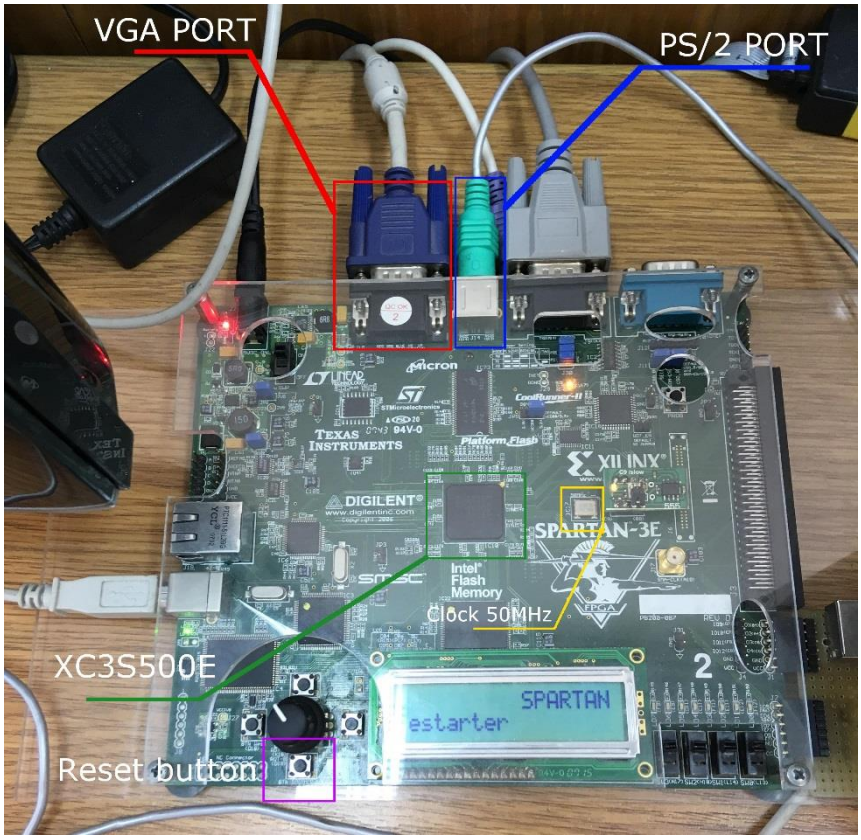
Gracz ma do dyspozycji cztery życia, których ilość jest prezentowana bezpośrednio na awatarze duszka – utrata kolejnych żyć powoduje jego stopniową dekonstrukcję.

1.2.3. Umiejętność specjalna

Duszek posiada odnawialną umiejętność specjalną w postaci skoku / teleportacji w bok lub w górę. Umiejętność tą gracz może użyć poprzez wciśnięcie lewego, prawego bądź środkowego przycisku myszy – każdemu przyciskowi podporządkowany jest inny kierunek skoku. Po użyciu skoku jest on niedostępny przez kolejne pięć sekund. Podczas ładowania skoku awatar gracza jest w kolorze białym, a gdy skok jest gotowy – w kolorze zielonym.

1.3. Sprzęt

Projekt został zaimplementowany na płycie Spartan-3E (UG230) [2]. Z punktu widzenia projektu istotnymi układami umieszczonymi na tej płycie były: układ FPGA XC3S500E [1], zegar 50MHz, port VGA, port PS/2.



Rysunek 1. Płyta Spartan-3E (UG230)

Sprzętem służącym do interakcji z użytkownikiem był monitor działający w standardzie VGA oraz mysz z interfejsem PS/2.

Rysunek 2. Mysz PS/2

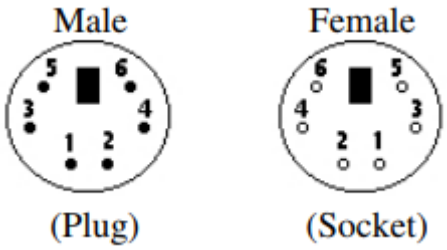
Rysunek 3. Monitor VGA

1.4. Wstęp teoretyczny

W celu realizacji oraz zrozumienia działania zaimplementowanych modułów odpowiedzialnych za komunikację z myszą komputerową oraz monitorem VGA niezbędna jest wiedza na temat interfejsów PS/2 oraz mechanizmu synchronizacji VGA.

1.4.1. Interfejs PS/2

Interfejs PS/2 jest 6-pinowym portem używanym do połączenia myszki lub klawiatury z komputerem. Konstrukcje portów dla klawiatury i myszki są elektrycznie podobne oraz korzystają z tego samego protokołu komunikacji [4].



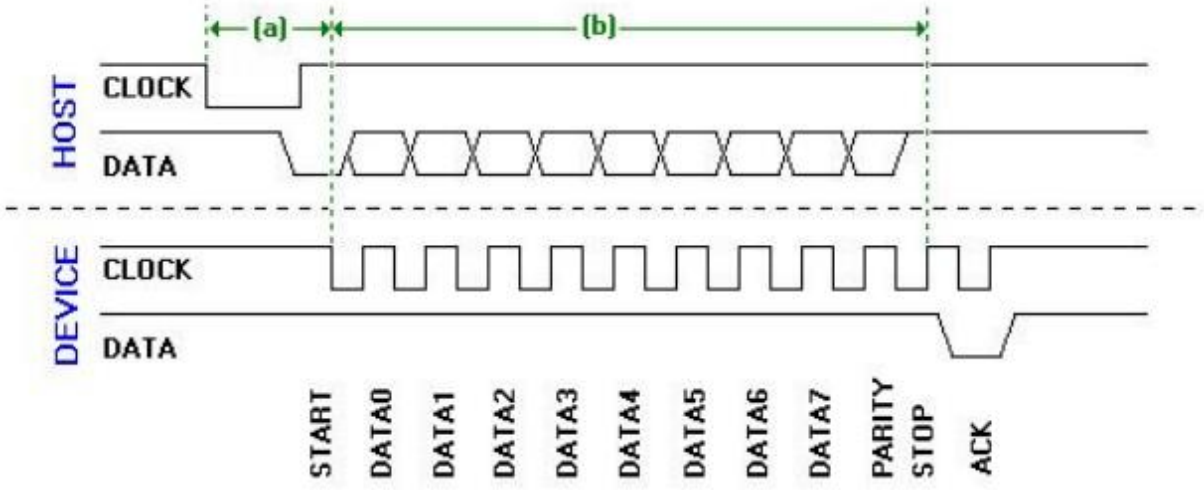
Rysunek 4. Złącze interfejsu PS/2

PIN	SYGNAŁ	FUNKCJA
Pin 1	+DATA	Dane
Pin 2	-	Nie używane
Pin 3	GND	Masa
Pin 4	VCC	Zasilanie (+5V)
Pin 5	+CLK	Zegar
Pin 6	-	Nie używane

Tabela 1. Opis pinów złącza interfejsu PS/2

Protokół komunikacyjny charakteryzuje się tym, że jest serialny, synchroniczny oraz dwukierunkowy. Podłączone urządzenie odpowiedzialne jest za generowanie sygnału zegara w zakresie 10.0kHz – 16.7kHz. Host kontroluje komunikacje przy użyciu linii zegara, stan niski oznacza, że komunikacja z urządzeniem jest zablokowana.

Przesyłanie danych rozpoczyna się wysłaniem jednego bitu startu, a następnie wysyłanych jest 8 bitów danych (rozpoczynając od najmniej znaczącego bitu - LSB), bit parzystości i bit stopu. Urządzenie odczytuje dane na zboczu opadającym zegara.



Rysunek 5. Szczegółowy opis komunikacji host-urządzenie

Mysz wysyła dane w 3-bajtowych pakietach danych, w kolejności pokazanej na poniższej tabeli.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	X overflow	Y sign	X sign	1	Middle Button	Right Button	Left Button
Byte 2	X movement							
Byte 3	Y movement							

Tabela 2. Dane wysyłane przez mysz PS/2

1.4.2. Synchronizacja VGA

VGA to analogowy standard wideo o wysokiej rozdzielczości używany w miejscach, gdzie potrzebna jest możliwość przesyłania ostrego, dokładnego obrazu. VGA wykorzystuje trzy oddzielne przewody do przesyłania, składających się z trzech kolorów (R, G i B), pionowo i poziomo synchronizujących sygnałów 0V/5V. Wideo to nic innego jak strumień klatek, w której każda klatka składa się z poziomych linii, gdzie każda linia to seria pikseli, noszących informacje o kolorze oraz jego jasności 0V-0.7V.

W celu poprawnej synchronizacji sygnałów z monitorem, VGA kontroler wymaga podania zegara pikseli, który składa się z dwóch liczników. Każdy licznik, zaczynając od 0, odlicza sygnał przesyłany do monitora, który możemy podzielić na cztery części:

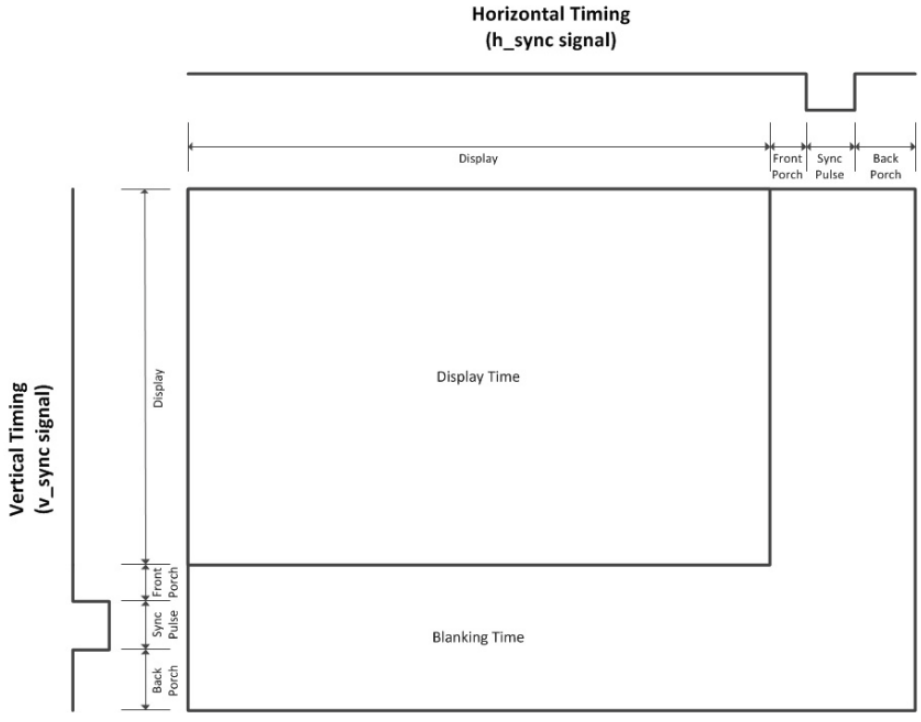
1. Display – okres, w którym może być wyświetlane wideo
2. Front Porch
3. Sync Pulse
4. Back Porch

Każda część, trwa stały określony okres czasu, zależny od szybkości odświeżania obsługiwanej przez monitor. Dokładne wartości znajdują się w tabeli na stronie [7].

Pierwszy licznik rośnie wraz z zboczem dolnym zegara i kontroluje h_sync, poziomą synchronizację. Po poziomym czasie wyświetlania, występuje czas wygaszania, który zawiera front porch, h_sync i poziomy back porch, każdy o ściśle ustalonej długości. Na końcu każdego wiersza, licznik resetuje się.

Drugi licznik, rośnie wraz z końcem każdego wiersza, kontrolując przy tym v_sync, pionową synchronizację. Tak jak wcześniej, po pionowym czasie wyświetlania występuje czas wygaszania, który składa się z pionowego front porch'a, v_sync'u oraz pionowego back porch'a. Wraz z końcem pionowego czasu wygaszania, licznik resetuje się.

Czas wyświetlania definiowany jest poprzez logiczną bramkę AND, pionowego i poziomego czasu wyświetlania.



Rysunek 6. Dokładny przebieg sygnałów synchronizacji VGA

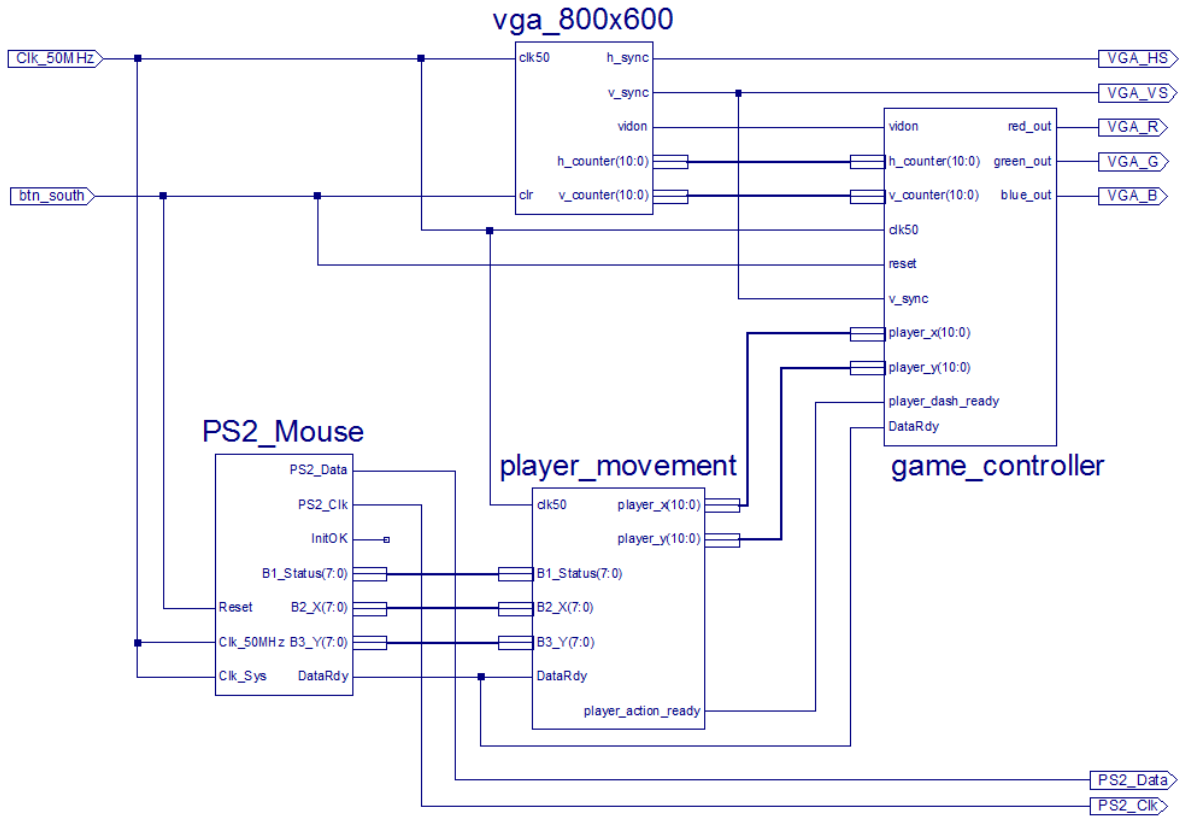
Na potrzeby projektu, aby uzyskać rozdzielczość 800x600 pikseli oraz częstotliwość odświeżania wynoszącą 72Hz, użyliśmy następujących wartości parametrów.

Resolution (px)	Refresh Rate (Hz)	Pixel Clock (MHz)	Horizontal (pixel clocks)				Vertical (rows)			
			Display	Front Porch	Sync Pulse	Back Porch	Display	Front Porch	Sync Pulse	Back Porch
800x600	72	50	800	56	120	64	600	37	6	23

Tabela 3. Użyte parametry VGA

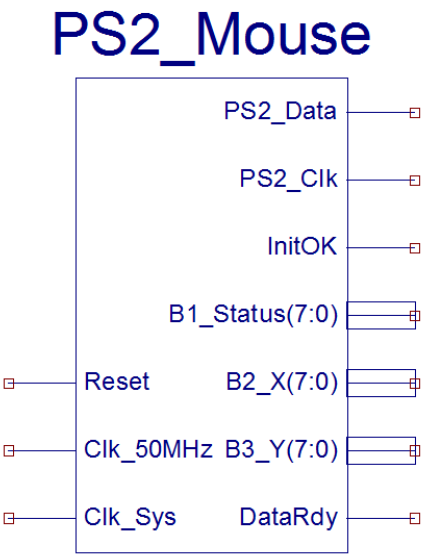
2. Projekt

2.1 Główny schemat układu (Top module)



Rysunek 7. Schemat układu

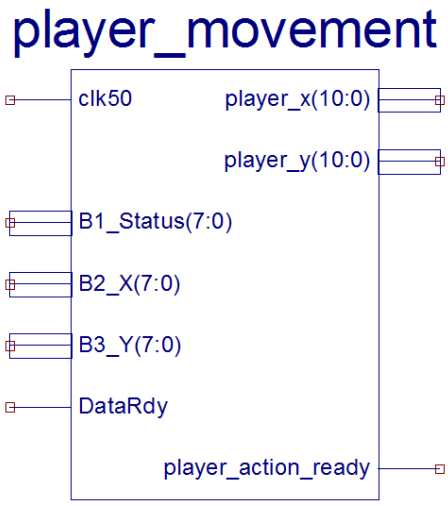
2.1.1 Moduł PS2_Mouse



Rysunek 8. Symbol modułu PS2_Mouse

Moduł PS2_Mouse został pobrany ze strony kursu [3]. W projekcie pełni funkcje odbierania kodów wysyłanych przez mysz podłączoną do portu PS/2 oraz ich wyprowadzanie na wyjścia podłączone do modułu obsługującego ruchy gracza.

2.1.2 Moduł player_movement



Rysunek 9. Symbol modułu player_movement

Moduł Player_Movement jest odpowiedzialny za logikę sterowania duszkiem. Na jego wejścia podłączone są wyprowadzenia modułu PS2_Mouse. Ruchy myszą powodują przesuwanie się duszka – w obrębie ograniczonego miejsca na środku ekranu. Przyciski myszy powodują użycie specjalnej umiejętności – o ile jest ona w stanie gotowości. Wyjściem układu są współrzędne x oraz y będące pozycją gracza na ekranie oraz flaga gotowości umiejętności specjalnej (player_action_ready).

Szczegóły implementacji

```
45    -- player position internals
46    signal in_player_x : SIGNED(10 downto 0) := "00100101100"; -- 300
47    signal in_player_y : SIGNED(10 downto 0) := "00100101100"; -- 300
48    signal in_player_action : SIGNED(2 downto 0) := "000";
49
50    -- player action
51    signal player_action_counter : UNSIGNED(27 downto 0) := (others => '0');
52    constant player_action_counter_max : UNSIGNED(27 downto 0) := to_unsigned(250000000, 28);
53    signal in_player_action_ready : std_logic := '1';
54
55    -- player movement area boundaries
56    constant player_x_lower_bound : SIGNED(10 downto 0) := "00011001000"; --200
57    constant player_x_upper_bound : SIGNED(10 downto 0) := "01001011000"; --600
58    constant player_y_lower_bound : SIGNED(10 downto 0) := "00001100100"; --100
59    constant player_y_upper_bound : SIGNED(10 downto 0) := "00111110100"; --500
60
61    constant dash_distance : SIGNED(6 downto 0) := "0110010"; -- 50
```

Kod 1. Lista wewnętrznych sygnałów modułu Player_Movement

Powyższy listing przedstawia listę wewnętrznych sygnałów modułu. W pierwszej sekcji zdefiniowane zostały wewnętrzne sygnały przetrzymujące pozycję duszka oraz aktualnie używaną akcję. W drugiej sekcji znajdują się sygnały umożliwiające kontrolę stanu akcji specjalnej oraz jego ładowanie. Maksymalna wartość licznika (player_action_counter_max) została ustawiona na wartość 250mln. Jest to ilość cykli zegara 50MHz potrzebna do odliczenia 5 sekund. W dolnej sekcji zostały zdefiniowane stałe tj. współrzędne ograniczające zakres poruszania się duszka oraz odległość jego skoku – dasha.

```
66      process_mouse_movement : process(clk50, DataRdy)
67      begin
68
69        if rising_edge(clk50) then
70           if in_player_action_ready = '1' then
71              if in_player_action(0) = '1' then --left click
72                 in_player_x <= in_player_x - dash_distance;
73                 in_player_action_ready <= '0';
74              end if;
75
76              if in_player_action(1) = '1' then -- right click
77                 in_player_x <= in_player_x + dash_distance;
78                 in_player_action_ready <= '0';
79              end if;
80
81              if in_player_action(2) = '1' then -- mouse3 click
82                 in_player_y <= in_player_y - dash_distance;
83                 in_player_action_ready <= '0';
84              end if;
85           else
86              -- player action not ready - loading logic
87              player_action_counter <= player_action_counter + 1;
88
89              if player_action_counter = player_action_counter_max then
90                  player_action_counter <= to_unsigned(0, 28);
91                  in_player_action_ready <= '1';
92              end if;
93
94           end if;
```

Kod 2. Akcje specjalne cz1

Część procesu przedstawiona na powyższym listingu odpowiada za wykonywanie akcji specjalnych oraz jej ładowanie, jeżeli nie jest gotowa.

```
96          if DataRdy = '1' then
97              -- Apply mouse movement deltas to temp variables
98              in_player_x <= in_player_x + signed(B2_X);
99              in_player_y <= in_player_y - signed(B3_Y);
100
101             -- Keep the player in horizontal boundaries
102             if in_player_x < player_x_lower_bound then
103                 in_player_x <= player_x_lower_bound;
104             elsif in_player_x > player_x_upper_bound then
105                 in_player_x <= player_x_upper_bound;
106             end if;
107
108             -- Keep the player in vertical boundaries
109             if in_player_y < player_y_lower_bound then
110                 in_player_y <= player_y_lower_bound;
111             elsif in_player_y > player_y_upper_bound then
112                 in_player_y <= player_y_upper_bound;
113             end if;
114
115         end if;
116       end if;
117
118     end process process_mouse_movement;
```

Kod 3. Akcje specjalne cz2

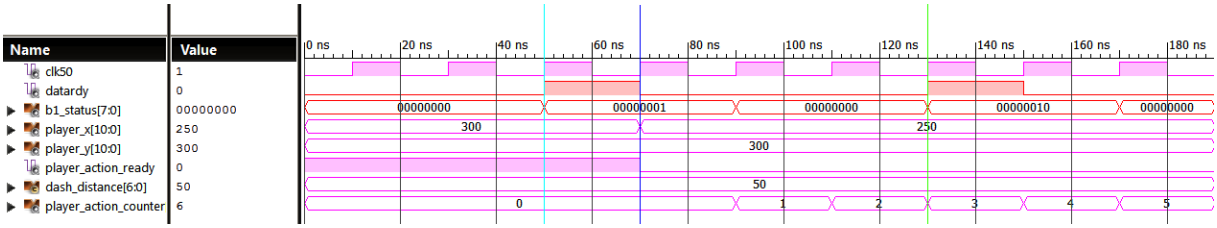
Następnie w przypadku, gdy ustawiony jest bit DataRdy, ma miejsce dodanie do pozycji gracza przesunięć myszki. W przypadku, jeśli nowo obliczone współrzędne mieszczą się po za granicami obszaru ruchu zostają one odpowiednio wyrównane do granicznych wartości.

```
120     process_button_status : process(clk50, DataRdy, B1_Status)
121     begin
122         if DataRdy = '1' and rising_edge(clk50) then
123 
124             in_player_action(0) <= B1_Status(0);
125             in_player_action(1) <= B1_Status(1);
126             in_player_action(2) <= B1_Status(2);
127 
128         end if;
129     end process process_button_status;
130 
131     -- output internals
132     player_x <= std_logic_vector(in_player_x);
133     player_y <= std_logic_vector(in_player_y);
134 
135     player_action_ready <= in_player_action_ready;
136 
```

Kod 4. Proces process_button_status

W module znajduje się również proces przypisujący wewnętrznemu sygnałowi wartości odczytane z wektora B1_Status, a także instrukcje przypisania współbieżnego wewnętrznych sygnałów do wyjściowych.

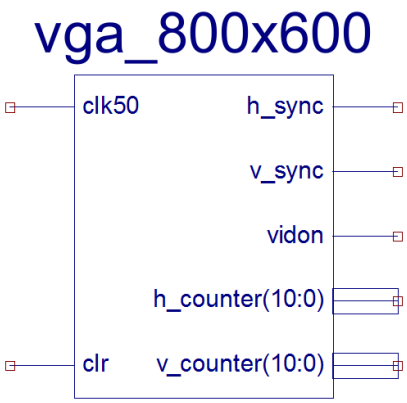
Przykładowe symulacje modułu



Rysunek 10. Symulacja działania modułu player_movement

Symulacja na rysunku 10. przedstawia zachowanie układu podczas próby użycia jednego z wariantów umiejętności specjalnej. W momencie 50ns (błękitna kreska) zostaje zasymulowane wciśnięcie lewego przycisku myszy podczas gdy akcja zawodnika jest gotowa (player_action_ready = ‘1’). Jak widać przy następnym zboczu rosnącym zegara (niebieska kreska) odpowiednio zmienia się współrzędna X postaci (player_x) – następuje skok, a flaga gotowości akcji zostaje ustawiona w stan ‘0’. Kolejne zbocza rosnące zegara powodują inkrementacje licznika, który odlicza czas (5s) do kolejnej możliwości użycia umiejętności. W stanie ładowania się umiejętności specjalnej zostaje zasymulowane kolejne wciśnięcie przycisku w momencie 130ns (zielona kreska). Widać brak reakcji układu – umiejętność nie może zostać użyta.

2.1.3 Moduł vga_800x600



Rysunek 11. Symbol modułu vga_800x600

Zadaniem modułu vga_800x600 było generowanie sygnałów synchronizujących komunikację z monitorem VGA na podstawie parametrów zalecanych przez producenta dla trybu 800x600 72Hz (Tabela 3). Moduł na wyjście wyprowadza również aktualne współrzędne „rysującego" punktu na ekranie (h_counter i v_counter) oraz informację czy punkt ten znajduje się w obszarze dozwolonym do rysowania – vidon. Wymienione trzy sygnały wyjściowe są niezbędne a zarazem wystarczające do prawidłowej realizacji rysowania obiektów gry poprzez moduł game_controller.

Szczegóły implementacji

```
43  architecture Behavioral of vga_800x600 is
44
45      -- breakpoint values with porches applied
46      constant H_pixels : std_logic_vector(10 downto 0) := "10000010000"; -- 1040 (64 + 120 + 800 + 56)
47      constant H_before_sync :std_logic_vector(10 downto 0) := "01101011000"; --(800 + 56)
48      constant H_after_sync :std_logic_vector(10 downto 0) := "01111010000"; --(800 + 56 + 120)
49
50      constant V_pixels : std_logic_vector(10 downto 0) := "01010011010"; -- 666 (23 + 6 + 600 + 37)
51      constant V_before_sync :std_logic_vector(10 downto 0) := "01001111101"; --(600 + 37)
52      constant V_after_sync :std_logic_vector(10 downto 0) := "01010000011"; --(600 + 37 + 6)
53
54      signal hcs : std_logic_vector(10 downto 0) := "00000000000"; --Horizontal and Vertical counters
55      signal vcs : std_logic_vector(10 downto 0) := "00000000000";
56      signal vsenable : std_logic := '0'; -- enable vertical counter
57
58  begin
```

Kod 5. Architektura modułu vga_800x600

Powyżej zdefiniowane stałe obejmują wartości graniczne liczników, w których, zgodnie z zaleceniami dla synchronizacji 800x600x72Hz, następują zmiany wartości sygnałów synchronizujących v_sync, h_sync.

```
60    --Counter for the horiziontal sync signal
61     counter_horizontal : process(clk50, clr)
62     begin
63         if clr = '1' then
64             hcs <= "00000000000";
65         elsif (clk50'event and clk50 = '1') then
66             if hcs = H_pixels - 1 then
67                 -- counter has reached the end of pixel count
68                 hcs <= "00000000000";
69                 vsenable <= '1'; -- enable vertical counter
70             else
71                 hcs <= hcs + 1; -- increment horizontal counter
72                 vsenable <= '0'; -- leave vsenable off
73             end if;
74         end if;
75     end process counter_horizontal;
76     h_sync <= '0' when (hcs >= H_before_sync and hcs < H_after_sync) else '1';
77
78     --Counter for the vertical sync signal
79     counter_vertical : process(clk50, clr)
80     begin
81         if clr = '1' then
82             vcs <= "00000000000";
83         elsif (clk50'event and clk50 = '1' and vsenable = '1') then
84             if vcs = V_pixels - 1 then
85                 vcs <= "00000000000";
86             else
87                 vcs <= vcs + 1; -- increment vertical counter
88             end if;
89         end if;
90     end process counter_vertical;
91     v_sync <= '0' when (vcs >= V_before_sync and vcs < V_after_sync) else '1';
```

Kod 6. Proces inkrementacji liczników

Powyższy listing przedstawia procesy odpowiedzialne za inkrementację liczników, moment zerowania liczników oraz zmianę wartości sygnałów synchronizujących na podstawie wartości liczników.

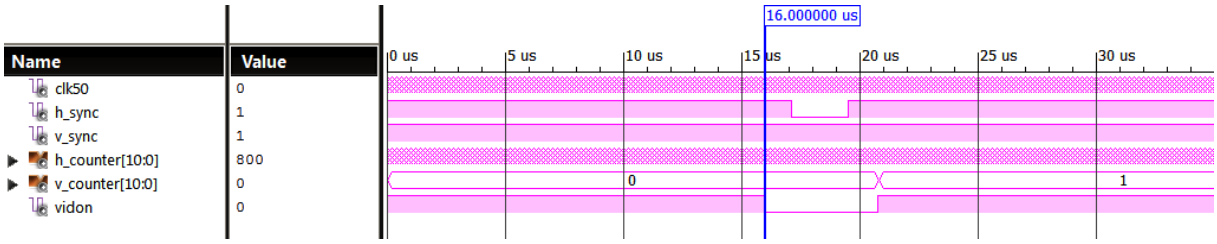
```
93      --Enable video out when withing the porches
94      vidon <= '1' when (((hcs >= 0) and (hcs < 800))
95                    and ((vcs >= 0)  and (vcs < 600))) else '0';
96
97      -- output counters
98      h_counter <= hcs;
99      v_counter <= vcs;
```

Kod 7. Vidon oraz sygnały wyjściowe modułu vga_800x600

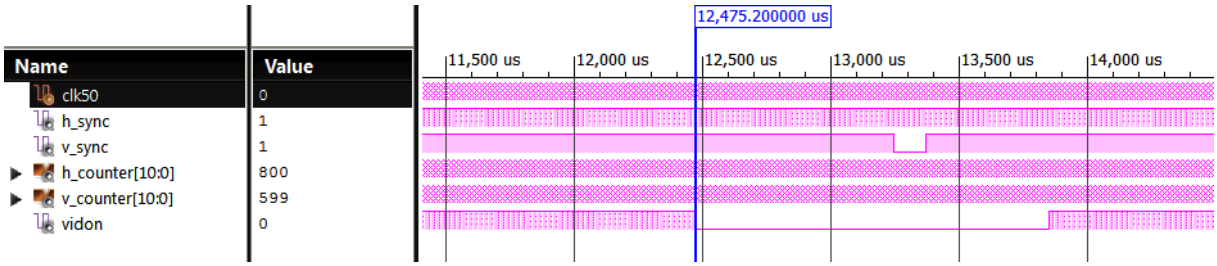
Obliczenie wartości oraz jej przypisanie dla sygnału vidon na podstawie aktualnego stanu liczników. Jeżeli znajdują się one w obrębie docelowego obszaru rysowania 800x600, sygnał zostaje ustawiony na ‘1’.

Przykładowe symulacje modułu

Poprzez symulację układu sprawdzono poprawność generowania sygnałów synchronizujących.

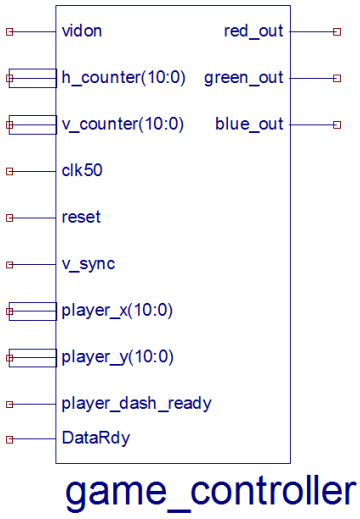


Rysunek 12. Horizontal timing



Rysunek 13. Vertical timing

2.1.4 Moduł game_controller



Rysunek 14. Symbol modułu game_controller

Moduł game_controller, jako najwyższy w hierarchii, agregował wszystkie sygnały z modułów player_movement oraz vga_800_600 i decydował o kolorze piksela wyświetlanego na monitorze, wyprowadzając na wyjście trzy bity R, G oraz B. To tutaj, odbywała się, detekcja kolizji, resetowanie położeń przeciwników, jak i zliczanie opóźnienia umiejętności specjalnej.

Szczegóły implementacji

```
191    -- random counters
192    signal x_pos_random : SIGNED(10 downto 0) := "01100100000";
193    signal y_pos_random : SIGNED(10 downto 0) := "01100100000";
194    signal c_random : SIGNED(10 downto 0) := "01100100000";

271    --random counters
272    process(clk50, DataRdy)
273    begin
274       if rising_edge(clk50)then
275
276          x_pos_random <= x_pos_random + "00000000001"; -- + 1
277          y_pos_random <= y_pos_random + "00000000001"; -- + 1
278          c_random <= c_random + "00000000001"; -- + 1
279
280          if DataRdy = '1' then
281             x_pos_random <= x_pos_random + "00000000011"; -- + 3
282             y_pos_random <= y_pos_random + "00000000111"; -- + 7
283             c_random <= c_random + "000000001111"; -- + 15
284          end if;
285       end if;
286    end process;
```

Kod 8. Fragmenty odpowiedzialne za pseudolosowość

Pseudolosowość w naszym programie została zaimplementowała poprzez zwiększanie się 11-sto bitowego licznika o stałą wartość wraz z zboczem rosnącym zegara, oraz dodatkowo w momencie akcji myszy wykonanej przez gracza. W ten sposób otrzymaliśmy niedeterministyczny, dla człowieka licznik, którego wartość mogła zostać wykorzystana, np. do wylosowania pozycji przeciwnika.

```
function RESPAWN_ENEMY( current_enemy : ENEMY_TYPE ; x_pos_random : SIGNED(10 downto 0) ; y_pos_random : SIGNED(10 downto 0) ; c_random : SIGNED(10 downto 0))
   return ENEMY_TYPE is
variable enemy : ENEMY_TYPE;
begin
   enemy.width := current_enemy.width;
   enemy.height := current_enemy.height;

   enemy.X := x_pos_random;
   enemy.Y := y_pos_random;

   enemy.X_velocity := current_enemy.X_velocity;
   enemy.Y_velocity := current_enemy.Y_velocity;

   if x_pos_random mod 4 = 1 then
      enemy.X_velocity := -enemy.X_velocity;
   end if;

   if y_pos_random mod 8 = 1 then
      enemy.Y_velocity := -enemy.Y_velocity;
   end if;

   -- going right bottom corner direction
   if enemy.X_velocity > 0 and enemy.Y_velocity > 0 then
      if c_random mod 2 = 1 then
         enemy.Y := "00000000001"; -- 1
      else
         enemy.X := "00000000001"; -- 1
      end if;
   end if;

   -- going right top corner direction
   if enemy.X_velocity > 0 and enemy.Y_velocity < 0 then
      if c_random mod 2 = 1 then
         enemy.Y := "01001011000"; -- 600
      else
         enemy.X := "00000000001"; -- 1
      end if;
   end if;

   -- going left bottom corner direction
   if enemy.X_velocity < 0 and enemy.Y_velocity > 0 then
      if c_random mod 2 = 1 then
         enemy.Y := "00000000001"; -- 1
      else
         enemy.X := "01100011111"; -- 799
      end if;
   end if;

   -- going left top corner direction
   if enemy.X_velocity < 0 and enemy.Y_velocity < 0 then
      if c_random mod 2 = 1 then
         enemy.Y := "01001011000"; -- 600
      else
         enemy.X := "01100011111"; -- 799
      end if;
   end if;

   return enemy;

end RESPAWN_ENEMY;
```

Kod 9. Funkcja odradzająca przeciwnika

Powyższa funkcja, przedstawia proces resetowania pozycji przeciwnika, która używana jest po tym jak przeciwnik przemieści się poza widoczną część ekranu lub po wystąpieniu kolizji z graczem. Jej zadaniem jest, wylosowanie krawędzi ekranu, na który przeniesie się obiekt przeciwnika oraz zmiana jego przyspieszenia w pozycji pionowej i poziomej tak, aby przechodził przez możliwy obszar pozycji gracza.

```
369   move_enemies : process (clk50, v_sync, v_counter, enemies, reset)
370      variable current_enemy : ENEMY_TYPE;
371   begin
372         if rising_edge(clk50) then
373            if reset = '1' then
374               game_state.player_lives <= "11";
375               game_state.IS_GAME_OVER <= false;
376            elsif not game_state.IS_GAME_OVER and v_sync = '1' and h_counter = "01100011111" and v_counter = "01001010111" then -- 799x599
377
378               for I in enemies'range loop
379                  current_enemy := enemies(I);
380
381                  current_enemy.X := current_enemy.X + current_enemy.X_velocity;
382                  current_enemy.Y := current_enemy.Y + current_enemy.Y_velocity;
383
384                  if current_enemy.X < 0 OR current_enemy.Y < 0 then -- przekrecil sie
385                     current_enemy := RESPAWN_ENEMY(current_enemy, x_pos_random, y_pos_random, c_random);
386
387                  end if; -- end przekrecil sie
388
389                   -- check collision
390                  if (IS_BETWEEN(current_enemy.X, signed(player_x), signed(player_x) + signed(player_size))
391                        or IS_BETWEEN(current_enemy.X + current_enemy.width, signed(player_x), signed(player_x) + signed(player_size))) AND
392                     (IS_BETWEEN(current_enemy.Y, signed(player_y), signed(player_y) + signed(player_size))
393                        or IS_BETWEEN(current_enemy.Y + current_enemy.height, signed(player_y), signed(player_y) + signed(player_size))) then
394
395                     game_state.player_lives <= game_state.player_lives - "01";
396                     current_enemy := RESPAWN_ENEMY(current_enemy, x_pos_random, y_pos_random, c_random);
397
398                     if game_state.player_lives = "00" then
399                        -- player died, game over
400                        game_state.IS_GAME_OVER <= true;
401                     end if;
402
403                  end if;
404
405                  enemies(I) <= current_enemy;
406
407               end loop;
408            end if;
409
410         end if;
411
412   end process move_enemies;
```

Kod 10. Proces move_enemies

Powyższy proces move_enemies miał na celu, zasymulowanie przesunięcia wszystkich przeciwników. W trakcie tej operacji wykrywana zostawała również kolizja pomiędzy przeciwnikiem, a graczem (linie 390-393), podczas takiej kolizji gracz tracił jedno życie, a pozycja przeciwnika zostawała resetowana.

Aby zapewnić mechanizm resetu gry, również w tym miejscu, po wciśnięciu przycisku SOUTH na płytce spartan, następowało przywrócenie podstawowego poziomu zdrowia gracza.

Procesem odpowiedzialnym za wybór koloru aktualnego piksela jest draw_display_area. To w tym miejscu przypisywane są wartości R G B, na podstawie aktualnego stanu gry.

```
288  draw_display_area : process(clk50, vidon, h_counter, v_counter, player_x, player_y)
289     variable current_enemy : ENEMY_TYPE;
290  begin
291
292     if rising_edge(clk50) and vidon = '1' then
293        if game_state.IS_GAME_OVER then
294           in_red_out <= '1';
295           in_green_out <= '0';
296           in_blue_out <= '0';
297        else
298           in_red_out <= '0';
299           in_green_out <= '0';
300           in_blue_out <= '0';
301         -- draw player
302         if h_counter > player_x and signed(h_counter) < signed(player_x) + signed(player_size) then
303            if v_counter > player_y and signed(v_counter) < signed(player_y) + signed(player_size) then
304
305               if   -- nose
306               (game_state.player_lives > "10"
307               and IS_INSIDE(signed(player_x) + player_detail_2base_size,
308                     signed(player_y) + signed(player_size) - player_detail_2base_size,
309                     player_detail_2base_size,
310                     player_detail_base_size,
311                     signed(h_counter),
312                     signed(v_counter)))
313               -- right eye
314               or (game_state.player_lives > "01"
315               and IS_INSIDE(signed(player_x) + player_detail_base_size,
316                     signed(player_y) + player_detail_base_size,
317                     player_detail_base_size,
318                     player_detail_base_size,
319                     signed(h_counter),
320                     signed(v_counter)))
321               -- left eye
322               or (game_state.player_lives > "00"
323               and IS_INSIDE(signed(player_x) + signed(player_size) -player_detail_2base_size,
324                     signed(player_y) + player_detail_base_size,
325                     player_detail_base_size,
326                     player_detail_base_size,
327                     signed(h_counter),
328                     signed(v_counter)))
329               then
330                  in_red_out <= '0';
331                  in_green_out <= '0';
332                  in_blue_out <= '0';
333               else
334                  -- if player has collision draw red;
335                  if player_dash_ready = '1' then
336                     in_green_out <= '1';
337                  else
338                     in_red_out <= '1';
339                     in_green_out <= '1';
340                     in_blue_out <= '1';
341                  end if;
342               end if;
343
344            end if;
```

Kod 11. Proces draw_display_area cz1

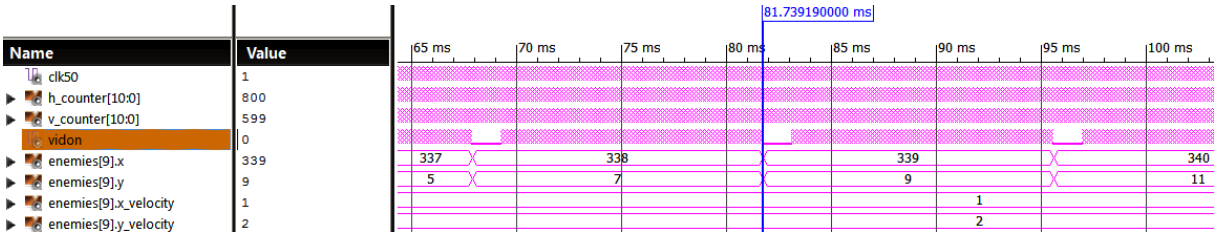
Powyższa część procesu draw_display_area odpowiedzialna jest za rysowanie modelu gracza, biorąc pod uwagę aktualną ilość jego życia. Widzimy to w linijkach 305, 313 oraz 322, gdzie na podstawie ilości punktów zdrowia rysują się poszczególne części modelu gracza: lewe oko, prawe oko, nosek.

Kolejna, a zarazem ostatnia część procesu draw_display_area miała za zadanie narysowanie modeli przeciwników, biorąc pod uwagę ich aktualne pozycje.

```
347             -- draw enemies
348             for I in enemies'range loop
349                current_enemy := enemies(I);
350
351                if signed(h_counter) > current_enemy.X and signed(h_counter) < current_enemy.X + current_enemy.width then
352                   if signed(v_counter) > current_enemy.Y and signed(v_counter) < current_enemy.Y + current_enemy.height then
353                      in_blue_out <= '1';
354                   end if;
355                end if;
356
357             end loop;
358
359          end if;
360
361          red_out <= in_red_out;
362          green_out <= in_green_out;
363          blue_out <= in_blue_out;
364       end if;
365    end process draw_display_area;
```

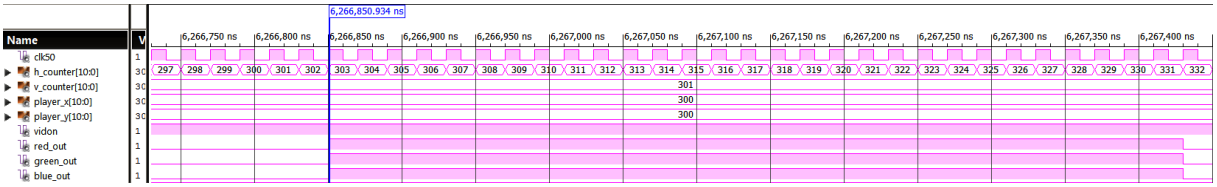
Kod 12. Proces draw_display_area cz2

Przykładowe symulacje modułu



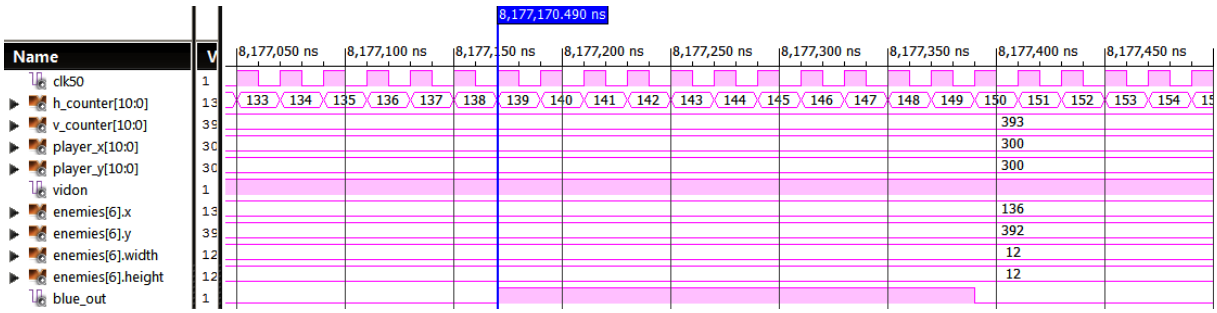
Rysunek 15. Symulacja klatki animiacji

Symulacja przedstawia moment przeliczenia – „klatkę” animacji. Przykładowo, jak widać na przebiegu czasowym, pozycja x, y jednego z pocisków zostaje przeliczona zgodnie z jego prędkością. Warto zaznaczyć, że animacja odbywała się zawsze po wyrysowaniu całego ekranu.



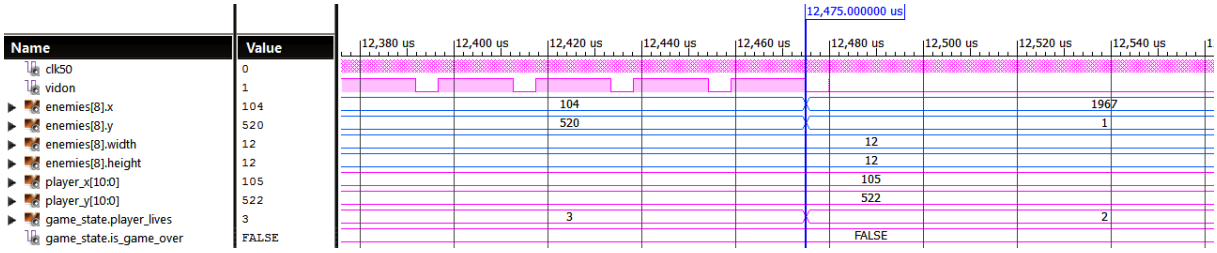
Rysunek 16. Symulacja pierwszego fragmentu gracza

Wyrysowanie pierwszego fragmentu gracza – w kolorze białym (umiejętność specjalna nie gotowa) w pozycji startowej 300, 300.



Rysunek 17. Symulacja rysowania pocisku

Na symulacji widać fragment rysowania jednego z pocisków. Można tutaj zauważyć lekkie przesunięcie, które jednak z punktu widzenia gracza (ludzkiego oka) jest niezauważalne.



Rysunek 18. Symulacja wykrycia kolizji pocisku z graczem

Wykrycie kolizji pocisku z graczem w klatce animacji. Skutkiem kolizji jest przypisanie nowej pozycji dla pocisku (respawn) oraz utrata życia gracza – zmniejszenie wartości rekordu player_lives z 3 do 2.

3. Implementacja

3.1. Rozmiar implementacji

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	458	9,312	4%
Number of 4 input LUTs	3,235	9,312	34%
Number of occupied Slices	1,906	4,656	40%
Number of Slices containing only related logic	1,906	1,906	100%
Number of Slices containing unrelated logic	0	1,906	0%
Total Number of 4 input LUTs	3,544	9,312	38%
Number used as logic	3,234		
Number used as a route-thru	309		
Number used as Shift registers	1		
Number of bonded IOBs	9	232	3%
Number of BUFGMUXs	1	24	4%
Average Fanout of Non-Clock Nets	3.40		

```
Total memory usage is 281328 kilobytes
```

Rysunek 19. Wykorzystanie zasobów urządzenia

3.2. Prędkość implementacji

	Met	Constraint	Check	Worst Case Slack	**Best Case Achievable**	Timing Errors	Timing Score
1	Yes	NET "Clk_50MHz_BUFGP/IBUFG" PERIOD = 20 ns HIGH 50%	SETUP...	0.448ns0.611ns	19.552ns	00	00

Rysunek 20. Analiza czasowa

3.3. Podręcznik użytkownika

Przed uruchomieniem płytki z wgranym programem należy upewnić się, że do odpowiednich portów podłączone zostały mysz oraz monitor VGA. Po uruchomieniu sprzętu rozpoczyna się gra.

Rysunek 21. Widok po uruchomieniu gry

Gracz steruje duszkiem znajdującym się w środkowej części ekranu za pomocą ruchów myszy. Celem gry jest omijanie nadlatujących pocisków (niebieskie obiekty) poprzez odpowiednie poruszanie się postacią oraz wykorzystywanie umiejętności specjalnych.

Rysunek 22. Stany wpływu gotowości umiejętności specjalnej gracza na jego model

Użycie umiejętności specjalnej. Po lewej stronie widać sytuację, gdy umiejętność specjalna duszka jest gotowa – kolor zielony. Po prawej stronie widać zdjęcie tuż po użyciu skoku w prawo. Duszek oddalił się od nadciągającego pocisku tracąc jednocześnie możliwość wykonania skoku przez kolejne 5 sekund co obrazuje kolor biały.

Rysunek 23. Ilość żyć wpływająca na model gracza

Powyżej zostały przedstawione kolejne stany duszka po utracie kolejnych żyć (w momencie kolizji z pociskiem).

Rysunek 244. Ekran końca gry

W momencie utraty wszystkich czterech żyć gra zostaje zakończona. Wyświetlanie obiektów gry zostaje wstrzymane – widoczny jest czarny ekran. Gracz ma możliwość rozpoczęcia nowej gry poprzez wciśnięcie dolnego przycisku (BTN SOUTH) na płycie Spartan.

4. Podsumowanie

Cele postawione przed rozpoczęciem projektu zostały osiągnięte. Podczas projektu grupa projektowa nabyła wiedzę na temat interfejsów PS/2 oraz VGA, a także praktyczne umiejętności realizacji wytyczonych zadań poprzez tworzenie syntezowalnego opisu sprzętu w języku VHDL. Bardzo ważną umiejętnością okazało się korzystanie z dostępnych źródeł w postaci dokumentacji sprzętu, artykułów naukowych oraz treści udostępnianych w internecie.

Ze względu na niewielkie doświadczenie członków zespołu kluczowe również okazało się przyrostowe realizowanie zadań. Każdorazowo zaimplementowana nowa funkcjonalność, zmiana w opisie VHDL była testowana i weryfikowana w wyniku implementacji sprzętowej. Iteracyjne podejście pozwoliło grupie na szybkie wykrywanie problemów i ich źródeł.

Projekt przechowywany jest w systemie kontroli wersji GIT (repozytorium GitHub [5]), co również pozwoliło na dokładniejsze śledzenie kolejnych zmian oraz ewentualny powrót do poprzedniej, zweryfikowanej wersji projektu.

4.1. Uwagi krytyczne

Zachowanie się układu w momencie gdy gracz próbuje przesunąć się za krawędź dozwolonego obszaru jest niepoprawne. Można zauważyć, że gracz na ułamek sekundy przenosi się poza obszar ruchu, a dopiero w następnej klatce zostaje przywrócony do prawidłowej pozycji. Problem ten wynika z współbieżnego przypisania sygnału jeszcze przed sprawdzeniem poprawności pozycji oraz wyrównaniem gracza.

4.2. Dalsze prace

Podczas realizacji projektu niejednokrotnie pojawiały się pomysły, które mogłyby ukierunkować dalszy rozwój projektu.

- Urozmaicenie rozgrywki poprzez stopniowe zwiększanie poziomu trudności wraz z czasem rozgrywki. Coraz większe prędkości i/lub co raz większa ilość obiektów w grze wraz z upływem czasu,
- umożliwienie śledzenia aktualnego wyniku i postępów, poprzez, na przykład rysowanie zapełniającego się timera na ekranie,
- urozmaicenie mechaniki gry poprzez wprowadzenie nowego typu obiektu lecącego w stronę gracza – np. apteczki, po której złapaniu gracz odnawiałby jedno życie,
- rozbudowa startu oraz końca gry – ekran początkowy oraz ekran końcowy,
- dopasowanie balansu parametrów w grze – tj. poziomu trudności, ilości przeciwników, ich prędkości, kształtów, częstości odnawiania umiejętności specjalnej, odległości skoku poprzez udostępnienie gry testerom i ocenę jej grywalności dla różnych konfiguracji parametrów.

5. Spis ilustracji

6. Bibliografia

[1] XC3S500E (xilinix.com)

[2] UG230.PDF

[3] Spartan-3E Starter Kit: http://www.zsk.ict.pwr.wroc.pl/zsk_ftp/fpga/

[4] "The PS/2 Mouse/Keyboard Protocol" Adam Chapweske
www.avrfreaks.net/sites/default/files/PS2%20Keyboard.pdf

[5] Repozytorium projektu Github: https://github.com/twix20/ucisw2_projekt

[6] Opis myszy PS/2: https://wiki.osdev.org/PS/2_Mouse

[7] Opis VGA: https://eewiki.net/pages/viewpage.action?pageId=15925278