UKŁADY CYFROWE I SYSTEMY WBUDOWANE

Zadanie projektowe: Gra wykorzystująca potencjometr

Autorzy:
Wojciech Ormaniec, 226181
Bartosz Rodziewicz, 226105

Prowadzący: dr inż. Jarosław Sugier

Grupa: Czwartek, TN, 8:00

Wprowadzenie

Temat

Tematem naszego projektu było zrealizowanie gry wykorzystującej potencjometr jako input od gracza.

Zrealizowaliśmy go w formie prostej gry zręcznościowej polegającej na ruszaniu platformy gracza w lewo i w prawo, celem uniknięcia spadających obiektów. Zderzenie ze spadającym obiektem kończy grę.

Sprzęt

Gra została napisana w języku VHDL na układ logiczny typu FPGA z rodziny Xilinx Spartan 3E FPGA Starter Kit Board, model XC3S500E.

Do gry wykorzystujemy input od gracza za pomocą potencjometru podłączonego do portu ADC płytki. Do obsługi ADC wykorzystujemy moduł *ADC_Ctrl* przygotowany przez dr inż. Jarosława Sugiera.

Wyjście z gry wykorzystuje oczywiście obraz puszczany przez port VGA na monitor. Obsługę VGA napisaliśmy od zera. Obraz wyjściowy jest w rozdzielczości 800x600px, 72Hz.

Teoria

Potencjometr – ADC

Dokładny opis działania portu ADC znajduje się w User Guidzie, wypisanym w spisie literatury.

Port ADC na płytce, na której pracowaliśmy działa pod zegarem 50MHz. Posiada dwa kanały umożliwiając pracę potencjometru (jeden kanał) lub joysticka (dwa kanały). Nasz projekt wykorzystuje potencjometr, czyli jeden kanał – kanał A.

Są dwa moduły odpowiedzialne za jego pracę – „LTC 6912-1 AMP" i „LTC 1407A-1 ADC".

Cała obsługa tego portu w naszym projekcie odbywa się za pomocą modułu czarnej skrzynki *ADC_Ctrl*.

Aby wystartować port ADC należy skrzynce *ADC_Ctrl* podać sygnał *AMP_WE* i 8 bitowy sygnał *AMP_DI* odpowiedzialny za czułość portu ADC na kanale A i B (jest on wykorzystywany przez moduł „LTC 6912-1 AMP").

Mimo wykorzystania tylko jednego kanału nasz sygnał *AMP_DI* to 00010001, czyli maksymalna czułość dla obu kanałów (dokładny opis innych wartości w User Guidzie).

Teoretycznie sygnał *AMP_WE* można podać tylko raz na działanie programu jednak korzystając z rady dr Sugiera podajemy ten sygnał co klatkę obrazu, gdy przetwarzany jest piksel (0,0), ponieważ zapewnia to stabilniejszą pracę.

Aby odczytać aktualną wartość kanałów ADC należy skrzynce *ADC_Ctrl* podać sygnał *ADC_Start*. Skrzynka *ADC_*Ctrl podaje wtedy sygnał *CONV*. Jest on wykorzystywany przez moduł „LTC 1407A-1 ADC" i powoduje on odczyt wartości i zwrócenie jej na sygnały *ADC_DOA* i *ADC_DOB* skrzynki *ADC_Ctrl*.

Sygnał podawany na wyjściach *ADC_DOA* i *ADC_DOB* jest podawany w formie 14 bitowej wartości typu signed w przedziale [0x2000, 0x1FFF], czyli w dziesiętnym [-8192, 8191].

Sygnał *ADC_Start* podajemy pod koniec klatki, w trakcie tzw. „retrace time”, w momencie, gdy rysowany byłby piksel (801, 801) (taki piksel nie istnieje w naszej rozdzielczości).

VGA

Obraz VGA podawany jest liniami, od lewej do prawej, z góry na dół.

Co „tick” zegara na wyjściach *R*, *G* i *B* podaje się wartość jaki kolor powinien przyjąć piksel (jeden piksel na „tick” zegara).

Jednak poza podawaniem danych poszczególnych pikseli w każdej linii i pomiędzy końcem jednej klatki obrazu, a początkiem drugiej jest tzw. „retrace time”, połączony z sygnałami *HorizontalSync* i *VerticalSync*, który zapewnia odpowiednią synchronizacje obrazu na monitorze.

W trakcie „retrace-time” wartości RGB powinny być ustawione na 000.

Więcej szczegółów jest opisane przy opisie procesów odpowiedzialnych za VGA.

Dokładny opis jak to działa, na przykładzie poziomej linii i rozdzielczości 640x480px, 60Hz podany jest w User Guidzie do naszej płytki. Wartości dla naszej rozdzielczości 800x600px, 72Hz znaleźliśmy na stronie VGA Signal Timings, załączonej w spisie literatury.

Kod

Projekt został napisany w całości w jednym module (plus moduł *ADC_Ctrl*). Może być on nie do końca czytelny i zawierać różne tymczasowe, mało mówiące nazwy, jak i pliki, które nie są przez nas już w żaden sposób używane.

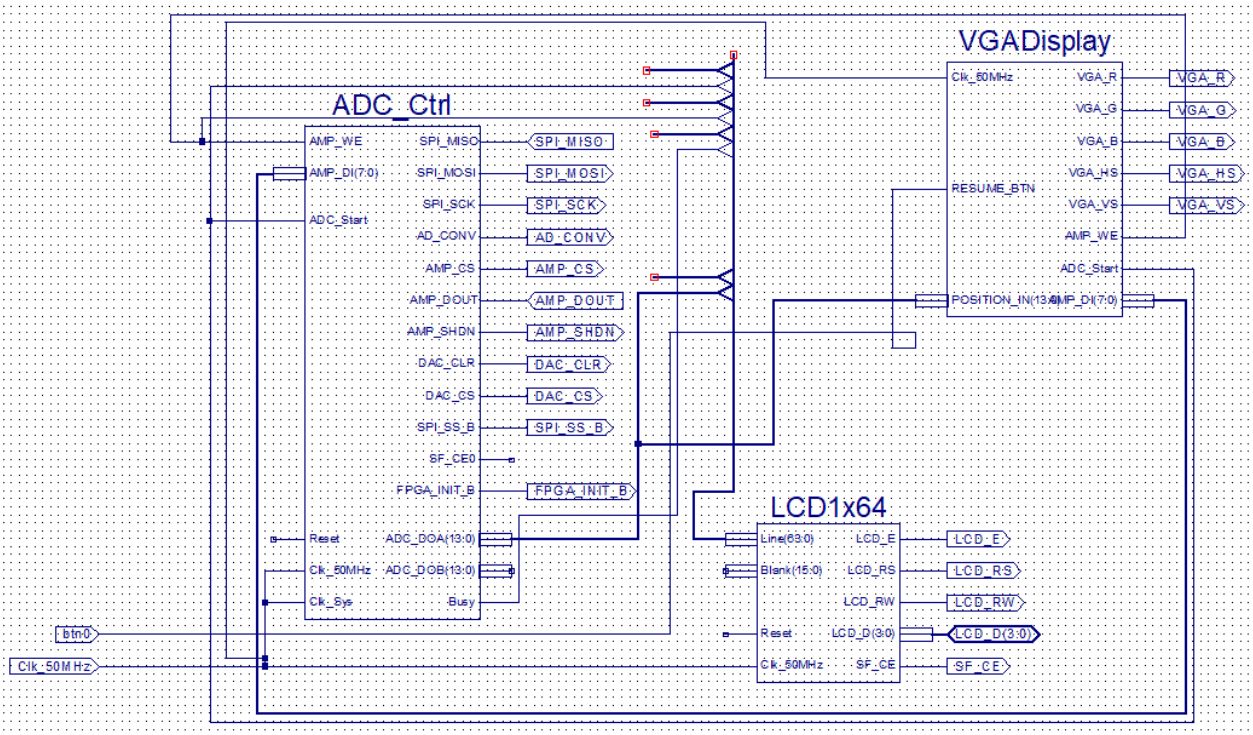
Zdarzyło się to z tego powodu, że pod koniec prac nad projektem planowane było posprzątanie kodu i pozbycie się nie potrzebnych rzeczy. Niestety nie starczyło nam czasu by to dokończyć. Na repozytorium w branchu *clean-up* znajduje się to co udało nam się zrobić w tej kwestii. Niestety z jakiegoś powodu przestał działać odczyt z potencjometru i nie byliśmy tego w stanie naprawić.

Bardzo za to przepraszamy, jeśli ktoś zdecydował się kontynuować pracę nad naszym projektem.

Projekt

Hierarchia źródeł

Naszym głównym źródłem w projekcie jest plik schematu (*adc_test.sch*), który wygląda następująco:



Jak widać kod składa się z trzech modułów. Głównym modułem naszej gry, jest moduł *VGADisplay*. Jest to jedyny moduł napisany przez nas i zawiera on całą logikę gry, jak i obsługę VGA.

Moduł *ADC_Ctrl* odpowiada za obsługę portu ADC, a *LCD1x64* za wyświetlanie liczb w formacie szesnastkowym na wbudowanym w płytkę wyświetlaczu. Ekran LCD był wykorzystywany przez nas do debugowania portu ADC i miał być usunięty w końcowym sprzątaniu kodu. Oba moduły są autorstwa dr Sugiera.

Dodatkowo poza schematem i modułami w projekcie znajdują się pliki UCF, które mapują I/O z naszego kodu na I/O płytki. Łącznie znajdują się trzy pliki UCF:

- *GenIO.ucf* – do obsługi zegara i przycisku restartu gry
- *ADC_DAC.ucf* – do obsługi ADC
- *LCD.ucf* – do obsługi LCD (miał być usunięty)

Sygnały wejściowe i wyjściowe są w miarę łatwo zrozumiałe patrząc na schemat:

- *Clk_50MHz* – główny zegar
- *Btn0* – sygnał z lewego przycisku, które resetuje grę
- Sygnały wyjściowe dla VGA – potrzebne do obsługi VGA (więcej o nich w opisie procesów odpowiedzialnych za obsługę i generowanie VGA)
- Sygnały dla ADC i LCD – obsługiwane przez moduły

Sygnały wewnętrzne pomiędzy modułowe to tylko sygnały służące do obsługi ADC napisane przez nas (wyjaśnione w opisie działania protokołu ADC).

Moduły

Cała nasza gra została napisana w jednym module, więc ten podpunkt jest bardziej wyjaśnieniem działania poszczególnych procesów znajdujących się w module *VGADisplay*.

Sygnały wejściowe i wyjściowe

`entity VGADisplay is`	
`Port ( Clk_50MHz : in STD_LOGIC;`	Sygnał zegara
`POSITION_IN : in signed (13 downto 0);`	Nieprzetworzony sygnał z wejścia ADC, z kanału A
`RESUME_BTN : in STD_LOGIC;`	Sygnał z lewego przycisku, resetujący grę
`VGA_R : out STD_LOGIC;`	Sygnał koloru czerwonego do wyjścia VGA
`VGA_G : out STD_LOGIC;`	Sygnał koloru zielonego do wyjścia VGA
`VGA_B : out STD_LOGIC;`	Sygnał koloru niebieskiego do wyjścia VGA
`VGA_HS : out STD_LOGIC;`	Sygnał HorizontalSync do wyjścia VGA
`VGA_VS : out STD_LOGIC;`	Sygnał VerticalSync do wyjścia VGA
`AMP_WE : out STD_LOGIC;`	Sygnał uruchamiający ADC
`AMP_DI : out STD_LOGIC_VECTOR (7 downto 0);`	Sygnał czułości ADC
`ADC_Start : out STD_LOGIC );`	Sygnał wysyłający żądanie pobrania wartości ADC
`end VGADisplay;`	

Sygnały wewnętrzne

`architecture Behavioral of VGADisplay is`	
`Signal vs_counter : INTEGER;`	Sygnał zliczający linijkę, która aktualnie jest rysowana na ekranie. Przyjmuje wartości z przedziału szerszego niż [0, 599] z uwagi na „retrace time”.
`Signal hs_counter : INTEGER;`	Sygnał zliczający aktualnie rysowany piksel w linijce (tak samo uwzględnia „retrace time”, a więc wartości ujemne stąd typ integer. *vs_counter* i *hs_counter* to główne sygnały odpowiadające za synchronizację czasową wszystkich działań w module, zaraz po zegarze.
`Signal playerPositionX : INTEGER := 400;`	Sygnał przechowujący aktualną pozycję X gracza.
`-- playerPositionY := 500;`	Pozycja Y gracza jest ustawiona na stałe w kodzie na wartość 500px
`type Point is array (1 downto 0) of INTEGER;`	Nasz typ danych – punkt/piksel
`type BombArray is array (4 downto 0) of Point;`	Nasz typ – tablica 5 punktów

`Signal bombsPosition : BombArray := ( (50, -2850), (200, -2650), (400, -2450), (600, -2250), (750, -2050) ); -- -2000 to get few seconds before first bomb, 200 difrence between them to not get them falling all in the same time -- bombsPosition(x)(1) -> x position; bombsPosition(x)(0) -> y position;`	Sygnał będący tablicą, przechowujący pozycję 5 punktów – pięciu bomb, które aktualnie spadają na gracza. Przypisywane wartości to początkowe wartości gry. Wartości X są rozłożone w miarę równomiernie po całym ekranie, wartości Y mają duża wartość, by nie spadały przez chwilę od uruchomienia gry i dodatkowo mają pomiędzy sobą odstęp, by nie spadały w jednej linii.
`Signal rand800 : unsigned (9 downto 0);`	Sygnał przechowujący aktualną „pseudolosową” wartość modulo 800, która jest wykorzystywana do losowej pozycji bomby po upadku
`Signal colision : STD_LOGIC := '0';` `begin`	Sygnał przechowujący stan końca gry, gdy ‘1’ gra jest skończona i trzeba zresetować przyciskiem.

Procesy odpowiedzialne za działanie VGA

```
	HorizontalSync : process ( Clk_50MHz, hs_counter ) is
	begin
		if ( rising_edge(Clk_50MHz) ) then
			if ( hs_counter < -64 ) then
				VGA_HS <= '0';
			else
				VGA_HS <= '1';
			end if;
		end if;
	end process;
```

Proces *HorizontalSync* odpowiada za nadawanie sygnału *HorizontalSync* dla VGA. Nadawany jest on, gdy *hs_counter* jest w przedziale [-184, -65].

```
	VerticalSync : process ( Clk_50MHz, vs_counter ) is
	begin
		if ( rising_edge(Clk_50MHz) ) then
			if ( vs_counter < -23 ) then
				VGA_VS <= '0';
			else
				VGA_VS <= '1';
			end if;
		end if;
	end process;
```

Proces *VerticalSync* odpowiada za nadawanie sygnału *VerticalSync* dla VGA. Nadawany jest on, gdy *vs_counter* jest w przedziale [-29, -24].

```
	PixelCounters : process ( Clk_50MHz, hs_counter, vs_counter ) is
	begin
		if ( falling_edge(Clk_50MHz) ) then
			if ( hs_counter = 855 ) then
				hs_counter <= -184;
				if ( vs_counter = 636 ) then
```

```
                              vs_counter <= -29;
                        else
                              vs_counter <= vs_counter + 1;
                        end if;
                  else
                        hs_counter <= hs_counter + 1;
                  end if;
            end if;
      end process;
```

PixelCounters to główny proces zarządzający timingiem w naszej grze. Dba on by *hs_counter* i *vs_counter* były w swoich zakresach.

Z tego co widzimy *hs_counter* przyjmuje wartości w przedziale [-184, 855]. Okres ten dzieli się na:

- Sygnał Sync [-184, -65]
- Front Porch [-64, -1]
- Czas wyświetlania [0, 799]
- Back porch [800, 855]

Sygnał *vs_counter* działa analogicznie, w przedziałach: [-29, -24], [-23, -1], [0, 599] i [600, 636].

Proces odpowiedzialny za działanie ADC

```
      ADCSync : process ( Clk_50MHz, hs_counter, vs_counter ) is
      begin
            if ( rising_edge(Clk_50MHz) ) then
                  if ( hs_counter = 0 and vs_counter = 0 ) then
                        AMP_DI <= X"11";
                        AMP_WE <= '1';
                  elsif ( hs_counter = 800 and vs_counter = 600 ) then
                        ADC_Start <= '1';
                  else
                        ADC_Start <= '0';
                        AMP_WE <= '0';
                  end if;
            end if;
      end process;
```

Proces ten generuje sygnały startujące ADC i sygnał odczytujący ADC.

Start następuje w pikselu (0, 0), odczyt (800, 600) (już poza zakresem rysowania klatki). Poprzez resztę pikseli sygnały są ustawione na 0.

Proces odpowiedzialny za rysowanie

```
      Print : process ( vs_counter, hs_counter, playerPositionX, bombsPosition ) is
      begin
      if ( rising_edge(Clk_50MHz) ) then
            if ( hs_counter > 0 and hs_counter < 799 and vs_counter > 0 and
vs_counter < 599 ) then
                  if (hs_counter > playerPositionX - 20 and hs_counter <
playerPositionX + 20 and vs_counter > 490 and vs_counter < 510) then
                        -- print player
                        VGA_R <= '1';
                        VGA_G <= '0';
```

```
                        VGA_B <= '1';
                    elsif ( hs_counter > bombsPosition(0)(1) - 5 and hs_counter <
bombsPosition(0)(1) + 5 and vs_counter > bombsPosition(0)(0) - 10 and vs_counter <
bombsPosition(0)(0) + 10 ) then
                        -- print bomb 0
                        VGA_R <= '0';
                        VGA_G <= '1';
                        VGA_B <= '0';
                    elsif ( hs_counter > bombsPosition(1)(1) - 5 and hs_counter <
bombsPosition(1)(1) + 5 and vs_counter > bombsPosition(1)(0) - 10 and vs_counter <
bombsPosition(1)(0) + 10 ) then
                        -- print bomb 1
                        VGA_R <= '0';
                        VGA_G <= '1';
                        VGA_B <= '0';
                    elsif ( hs_counter > bombsPosition(2)(1) - 5 and hs_counter <
bombsPosition(2)(1) + 5 and vs_counter > bombsPosition(2)(0) - 10 and vs_counter <
bombsPosition(2)(0) + 10 ) then
                        -- print bomb 2
                        VGA_R <= '0';
                        VGA_G <= '1';
                        VGA_B <= '0';
                    elsif ( hs_counter > bombsPosition(3)(1) - 5 and hs_counter <
bombsPosition(3)(1) + 5 and vs_counter > bombsPosition(3)(0) - 10 and vs_counter <
bombsPosition(3)(0) + 10 ) then
                        -- print bomb 3
                        VGA_R <= '0';
                        VGA_G <= '1';
                        VGA_B <= '0';
                    elsif ( hs_counter > bombsPosition(4)(1) - 5 and hs_counter <
bombsPosition(4)(1) + 5 and vs_counter > bombsPosition(4)(0) - 10 and vs_counter <
bombsPosition(4)(0) + 10 ) then
                        -- print bomb 4
                        VGA_R <= '0';
                        VGA_G <= '1';
                        VGA_B <= '0';
                        VGA_B <= '0';
                    elsif ( colision = '1' ) then
                        -- print red game over background
                        VGA_R <= '1';
                        VGA_G <= '0';
                        VGA_B <= '0';
                    else
                        -- print black background
                        VGA_R <= '0';
                        VGA_G <= '0';
                        VGA_B <= '0';
                    end if;
                else
                    -- set 0 to secure vga output and not get random lines
                    VGA_R <= '0';
                    VGA_G <= '0';
                    VGA_B <= '0';
```

```
            end if;
      end if;
      end process;
```

Jest to proces rysujący wszystkie elementy na ekranie.

Jego działanie jest następujące:

1. Jeśli wartości *hs* i *vs_counter* nie są w przedziale (0, 799) i (0, 599) to ma ustawić RGB na 000, aby zapobiec dziwnym liniom. W trakcie analizy kodu do dokumentacji zauważyliśmy, że warunek powinien być w przedziale [0, 799] i [0, 599], więc w tym momencie zostawiamy po jednym czarnym pikselu na obrzeżach.
2. Jeśli się w tym przedziale znajdują to ma rysować obiekty w kolejności:
    a. Gracz
    b. Bomba 0
    c. …
    d. Bomba 4
    e. Czerwone tło gry (gdy wykryto kolizję i gra się skończyła)
    f. Czarne tło

Kolejność ma znaczenie tylko jeśli obiekty by na siebie nachodziły (co oczywiście czasem się zdarza, zwłaszcza jeśli chodzi o tło).

Warunki do rysowania poszczególnych obiektów są tak skomplikowane, ponieważ pozycja obiektów jest przechowywana jako punkt, a obiekty są rysowane jako prostokąty (a nie pojedyncze piksele).

Rozmiary obiektów:

- Gracz – prostokąt 39x18px
- Bomba – prostokąt 9x19px

Tak dziwne i nierówne rozmiary obiektów są spowodowane tym samym błędem co to, że zostawiamy 1px krawędzi.

Procesy logiki gry

```
      CalculatePlayerPos : process ( POSITION_IN ) is
            variable temp : INTEGER;
      begin
            if ( rising_edge(Clk_50MHz) and colision = '0' ) then
                  temp := to_integer( POSITION_IN );
                  temp := temp / 64;
                  temp := temp * 3;

                  playerPositionX <= 399 + temp;
            end if;
      end process;
```

Proces odpowiedzialny za ustawienie aktualnej pozycji gracza.

Do tymczasowej zmiennej przypisuje on aktualną wartość zwróconą przez ADC. ADC zwraca wartość w przedziale [-8192, 8191], więc musimy ją „obrobić”, aby nadawała się do użycia. Biorąc jej 3/64

otrzymujemy wartość w przedziale [-384, 383], która praktycznie idealnie nadaje się nam do poruszania 39px platformą po 800px ekranu.

Po zmniejszeniu zakresu zmiennej temp ustawiamy gracza na środek ekranu i dodajemy ten „offset”.

W trakcie analizy kodu do dokumentacji zauważyliśmy, że proces ten, mógłby wykonywać się raz na klatkę, ponieważ raz na klatkę odczytywana jest nowa wartość z ADC. Zmniejszyło by to znacząco ilość obliczeń układu w trakcie pracy programu.

```
	CalculateBombsPos : process ( hs_counter, vs_counter, bombsPosition ) is
	begin
		if ( rising_edge(Clk_50MHz) ) then
			if ( colision = '0' ) then
				if ( hs_counter = 855 and vs_counter = 636 ) then
					if ( bombsPosition(0)(0) >= 650 ) then --despawn
below the screen
						bombsPosition(0)(1) <= to_integer( rand800
);
						bombsPosition(0)(0) <= -12; --spawn above
screen
					else
						--move 7 pixels down every frame
						bombsPosition(0)(0) <= bombsPosition(0)(0) +
7;
					end if;
				end if;
				if ( hs_counter = 855 and vs_counter = 636 ) then
					if ( bombsPosition(1)(0) >= 650 ) then --despawn
below the screen
						bombsPosition(1)(1) <= to_integer( rand800
);
						bombsPosition(1)(0) <= -12; --spawn above
screen
					else
						--move 7 pixels down every frame
						bombsPosition(1)(0) <= bombsPosition(1)(0) +
7;
					end if;
				end if;
				if ( hs_counter = 855 and vs_counter = 636 ) then
					if ( bombsPosition(2)(0) >= 650 ) then --despawn
below the screen
						bombsPosition(2)(1) <= to_integer( rand800
);
						bombsPosition(2)(0) <= -12; --spawn above
screen
					else
						--move 7 pixels down every frame
						bombsPosition(2)(0) <= bombsPosition(2)(0) +
7;
					end if;
				end if;
				if ( hs_counter = 855 and vs_counter = 636 ) then
```

```
                                    if ( bombsPosition(3)(0) >= 650 ) then --despawn
below the screen
                                          bombsPosition(3)(1) <= to_integer( rand800
);
                                          bombsPosition(3)(0) <= -12; --spawn above
screen
                                    else
                                          --move 7 pixels down every frame
                                          bombsPosition(3)(0) <= bombsPosition(3)(0) +
7;
                                    end if;
                              end if;
                              if ( hs_counter = 855 and vs_counter = 636 ) then
                                    if ( bombsPosition(4)(0) >= 650 ) then --despawn
below the screen
                                          bombsPosition(4)(0) <= -12; --spawn above
screen
                                          bombsPosition(4)(1) <= to_integer( rand800
);
                                    else
                                          --move 7 pixels down every frame
                                          bombsPosition(4)(0) <= bombsPosition(4)(0) +
7;
                                    end if;
                              end if;
                        end if;
                        -- reset bomb pos if button pressed
                        if ( RESUME_BTN = '1' ) then
                              bombsPosition <= ( (50, -2850), (200, -2650), (400, -
2450), (600, -2250), (750, -2050) );
                        end if;
                end if;
        end process;
```

Jest to proces, który dla każdej bomby wylicza jej nową pozycję, co klatkę obrazu.

Na początku sprawdza, czy nie nastąpiła kolizja. Jeśli nastąpiła to nie wylicza nowych wartości, a w momencie zresetowania gry przywraca bomby na pozycje startowe.

Jeśli kolizja nie nastąpiła to dla każdej bomby sprawdza, czy wypadła z ekranu i jeśli tak to przesuwa ją na górę w nowej losowej pozycji (korzystając z sygnału *rand800*, który działa jak aktualna wartość pseudolosowa modulo 800). Bomba chwilę leci poza ekranem, by gracz nie miał odczucia, że ta nowa bomba to dokładnie ta sama co właśnie udało mu się ominąć, a nowa nadlatująca w niego.

Jeśli nie wypadła z ekranu, to po prostu przesuwa ją w dół, o 7px, co klatkę. Stwierdziliśmy, że jest to w miarę optymalna prędkość do gry.

```
        CheckIfColiosion : process ( Clk_50MHz, hs_counter, vs_counter,
bombsPosition, playerPositionX ) is
        begin
                if ( rising_edge(Clk_50MHz) and hs_counter = 801 and vs_counter = 601
) then
                        -- once per frame check if colision occurs
                        if ( hs_counter = 801 and vs_counter = 601 ) then
```

```
                        if ( abs( playerPositionX - bombsPosition(0)(1) ) < 25
and abs( 500 - bombsPosition(0)(0) ) < 20 ) then
                                colision <= '1';
                        elsif ( abs( playerPositionX - bombsPosition(1)(1) ) < 25
and abs( 500 - bombsPosition(1)(0) ) < 20 ) then
                                colision <= '1';
                        elsif ( abs( playerPositionX - bombsPosition(2)(1) ) < 25
and abs( 500 - bombsPosition(2)(0) ) < 20 ) then
                                colision <= '1';
                        elsif ( abs( playerPositionX - bombsPosition(3)(1) ) < 25
and abs( 500 - bombsPosition(3)(0) ) < 20 ) then
                                colision <= '1';
                        elsif ( abs( playerPositionX - bombsPosition(4)(1) ) < 25
and abs( 500 - bombsPosition(4)(0) ) < 20 ) then
                                colision <= '1';
                        end if;
                end if;
                -- resume game if button pressed
                if ( RESUME_BTN = '1' ) then
                        colision <= '0';
                end if;
        end if;
    end process;
```

Jest to proces sprawdzający czy nastąpiła kolizja gracza z bombą.

Sprawdzenie polega na wyliczeniu odległości pomiędzy pozycjami gracza i każdej bomby po kolei uwzględniając ich faktyczne rozmiary (a nie, że są pikselami). Również tutaj jest możliwość, że całkiem skrajne wartości, gdy bomba tylko leciutko dotyka gracza nie zostaną wykryte, jednak z uwagi na tempo gry i niedokładność potencjometru, jest to nie odczuwalne w trakcie gry.

Gdy kolizja zostanie wykryta następuje ustawienie flagi kolizji, co zmienia działanie procesów opisanych wyżej. Gdy wciśnięty zostaje przycisk restartu gry, flaga jest kasowana i gra zaczyna działać dalej (inne procesy resetują jej stan do stanu początkowego).

Procesy pomocnicze

```
    CalculateRand : process ( Clk_50MHz, POSITION_IN ) is
    begin
        if ( rising_edge(Clk_50MHz) ) then
            if ( rand800 < 800 ) then
                rand800 <= rand800 + 1 + unsigned( POSITION_IN(1 downto
0) );
            else
                rand800 <= (others => '0');
            end if;
        end if;
    end process;
```

Jest to generator liczby pseudolosowej modulo 800.

Co „tick” zegara do sygnału dodawana jest liczba od 1 do 4 (1 zawsze, plus wartość dwóch najmłodszych bitów ADC, które mogą mieć wartość [0, 3]).

Gdy 800 zostanie przekroczone następuje wyzerowanie sygnału.

Odwołując się do tego sygnału później mamy za każdym razem inną losową wartość, jeśli odwołania te następują w jakiejś odległości czasowej od siebie.

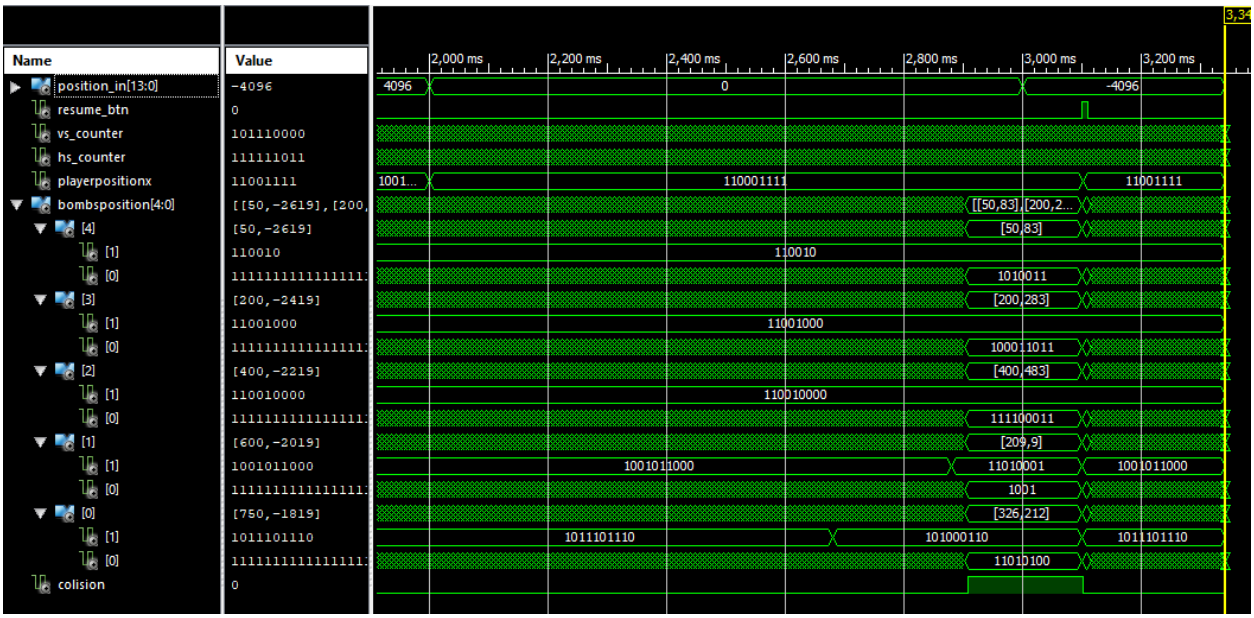
Symulacja

Wykonanie poprawnej symulacji do naszego kodu jest bardzo trudne. Napisanie całej gry w jednym module skutkuje, że nawet ograniczając liczbę śledzonych sygnałów bardzo szybko trafiamy na „tracing limit”. Poniżej spróbujemy wykazać ciekawsze momenty z symulacji, które udało nam się wykazać.

Wykrycie kolizji

Symulacja wykrycia kolizji zostaną napisana w taki sposób, że pozycja gracza (399) nie zmieniała się przez cały czas, podczas, gdy jedna z domyślnych pozycji bomb (bomba #2) ma pozycję 400.

Na powyższej symulacji widzimy, że gdy bomba #2 osiągnęła pozycję y 483 nastąpiło wykrycie kolizji. Od tego momentu pozycje bomb już się nie ruszyły. Program nie reagowałby również na zmianę pozycji gracza na wejściu *position_in*, jednak nie zostało to zasymulowane.

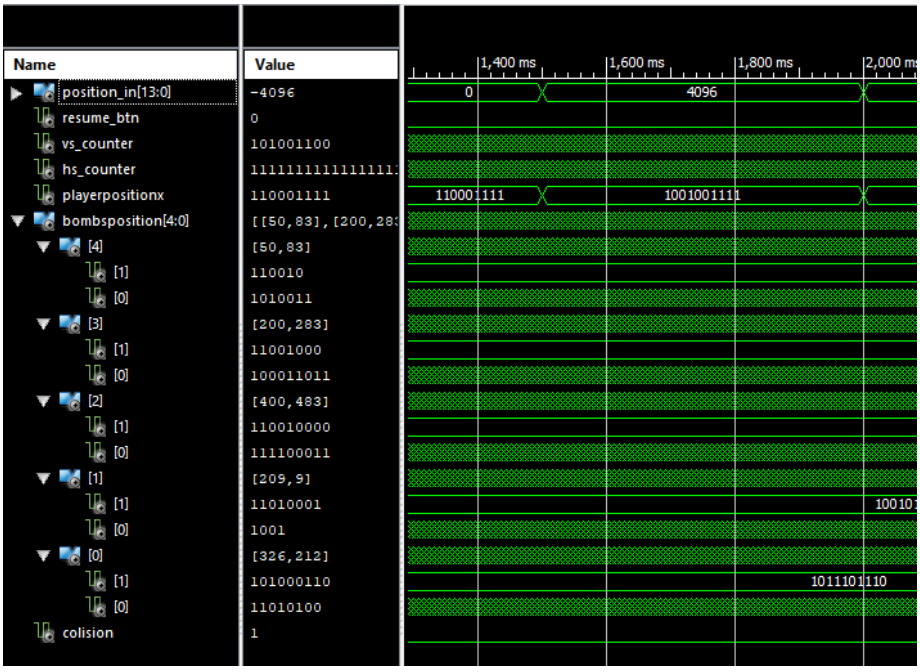


Jest to druga symulacja, w podobny sposób symulująca wykrycie kolizji. Zasymulowana została zmiana pozycji gracza po wykryci kolizji i widzimy, że gracz zmienia swoją pozycję dopiero po restarcie gry przyciskiem.

Restart gry

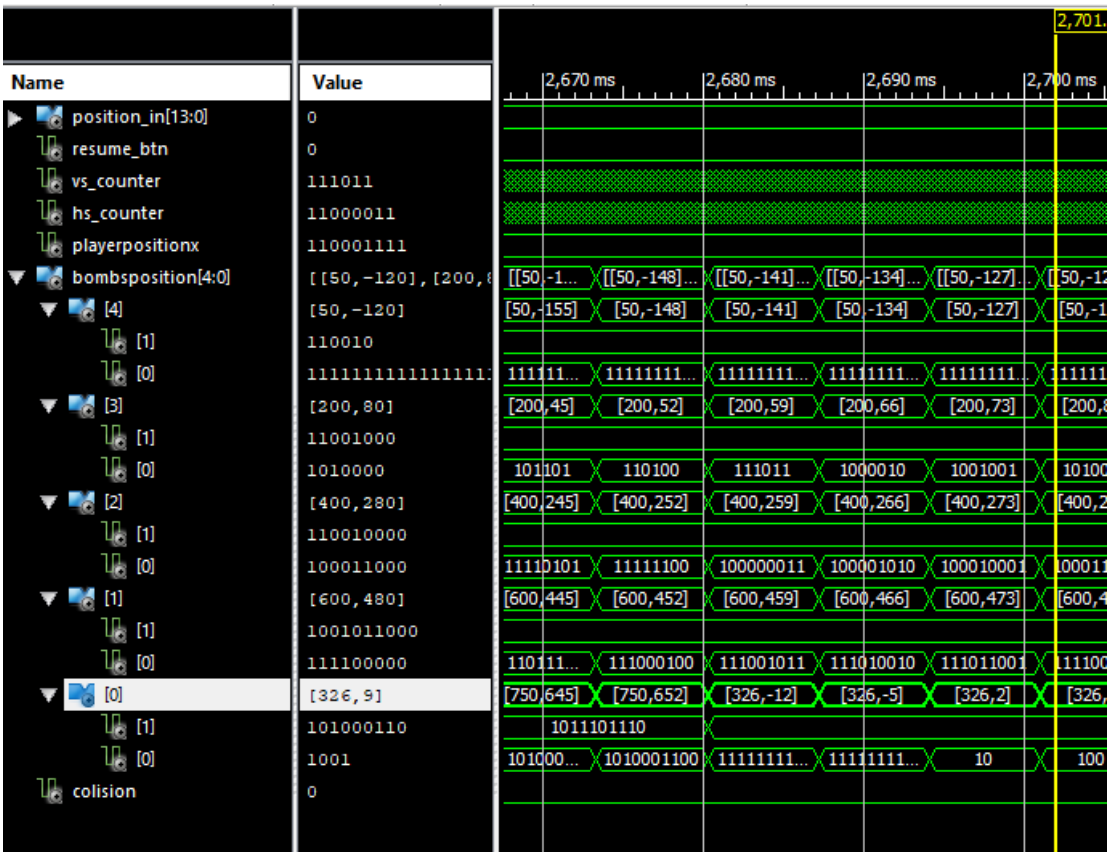
Na drugiej symulacji widzimy, że gra restartuje się poprawnie i bomby wracają na swoje pierwotne pozycje.

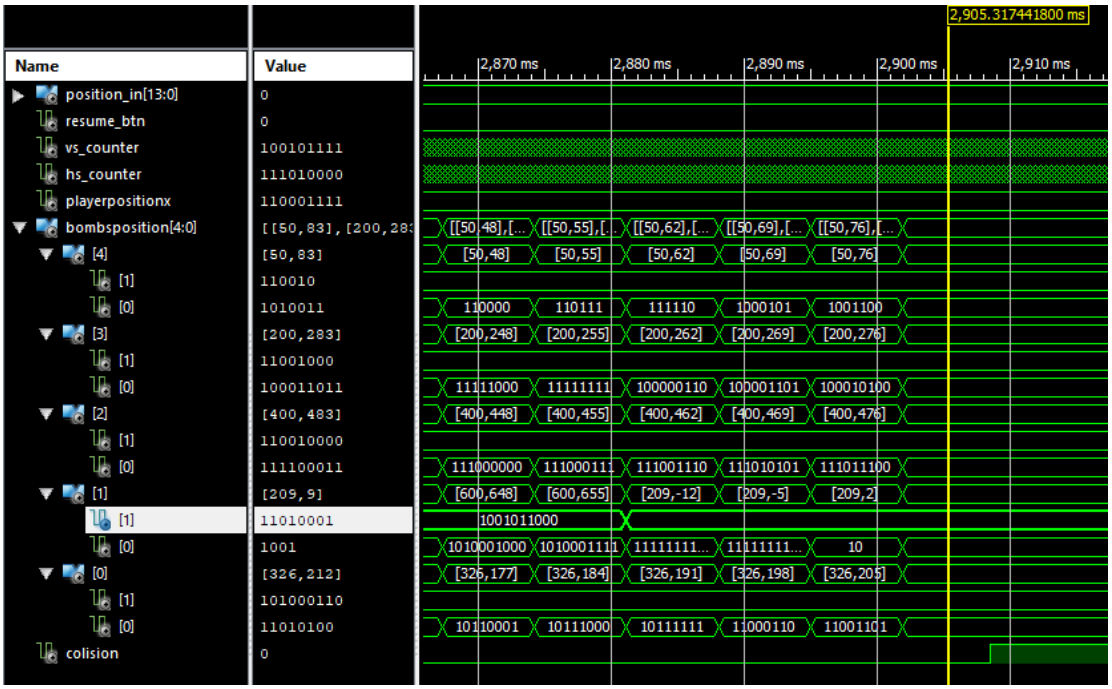
Zmiana pozycji gracza



Na tym fragmencie symulacji drugiej widzimy, że pozycja gracza ulega zmianie w zależności od wartości *position_in*. (0b110001111 to 399, 0b1001001111 to 591).

Zmiana pozycji y bomby przy spadnięciu na dół ekranu





Inne fragmenty symulacji pokazują, że bomby poprawnie zmieniają swoją pozycję y w momencie wypadnięcia z ekranu.

Implementacja

Rozmiar

Użycie konkretnych zasobów układu reprezentuje tabelka poniżej.

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	464	9,312	4%
Number of 4 input LUTs	2,326	9,312	24%
Number of occupied Slices	1,545	4,656	33%
Number of Slices containing only related logic	1,545	1,545	100%
Number of Slices containing unrelated logic	0	1,545	0%
Total Number of 4 input LUTs	2,876	9,312	30%
Number used as logic	2,325		
Number used as a route-thru	550		
Number used as Shift registers	1		
Number of bonded IOBs	26	232	11%
Number of IDDR2s used	1		
Number of ODDR2s used	2		
Number of BUFGMUXs	1	24	4%
Number of MULT18X18SIOs	1	20	5%
Average Fanout of Non-Clock Nets	2.25		

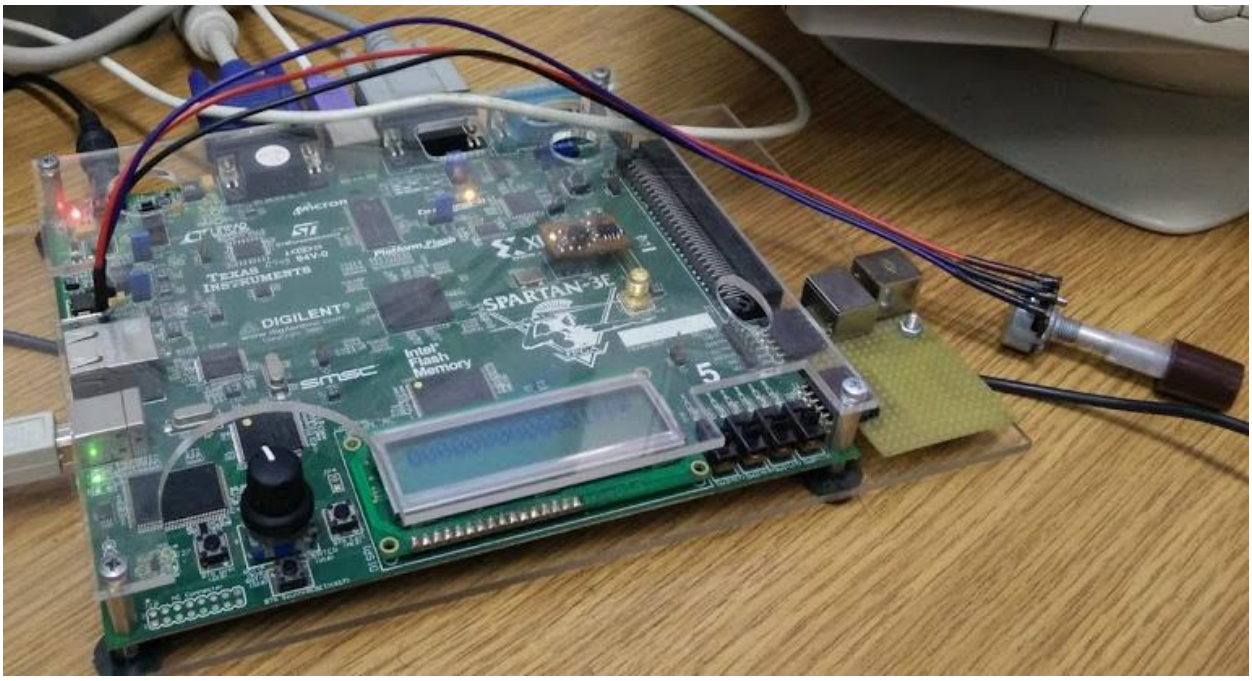
Prędkość

	Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
1	No	NET "Clk_50MHz_BUFGP/IBUFG" PERIOD = 20 ns HIGH 50%	SETUP HOLD	-4.594ns 0.927ns	29.188ns	199 0	279217 0

Nasz program jest taktowany zegarem 50MHz, czyli okresem 20ns. Z uwagi na „timing constraint” nasz rzeczywisty okres wynosi 29.188ns, co przekłada się na zegar rzeczywisty 34.261MHz.

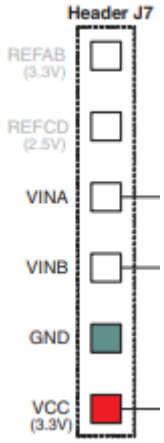
Podręcznik użytkownika

Podłączenie układu



Poza podstawowym podłączeniem płytki Spartan, należy podłączyć monitor do złącza VGA i potencjometr do portu ADC.

Podłączenie potencjometru



Potencjometr posiada trzy kabelki – zasilanie (czerwony), sygnał (czarny) i masę (niebieski). Kolory mogą się różnić dla innych potencjometrów.

Kabel zasilania i masy podłączamy odpowiednio do zasilania i masy, a kabel sygnałowy do wejścia kanału A.

Zaprogramowanie układu

Po poprawnym podłączeniu układu kompilujemy projekt („*Generate Programming File*”) (bądź bierzemy gotową binarkę z naszego GitHuba) i wgrywamy gotowy plik poprzez iMPACT. Wgrywamy tylko plik *.bit* i pomijamy wszystkie inne wyskakujące okienka.

Właściwa gra

Po poprawnym zaprogramowaniu układu na ekranie LCD powinna wyświetlić się jakaś liczba. Jest to aktualna wartość zwracana przez ADC i powinna się zmieniać, gdy kręcimy potencjometrem.

Po zaprogramowaniu również powinien włączyć się monitor i wyświetlić platformę gracza w kolorze fuksji. Powinna poruszać się lewo prawo, gdy kręcimy potencjometrem.

Po dwóch-trzech sekundach powinniśmy zobaczyć pierwsze spadające bomby. Bomby spadają cały czas z tą samą prędkością. Gra może trwać w nieskończoność – kończy się, gdy jedna z bomb uderzy w platformę gracza.

Ekran wtedy robi się czerwony i gra się wstrzymuje.

Aby wtedy ponownie uruchomić grę należy nacisnąć lewy przycisk. Restart gry jest możliwy również w trakcie gry, nie trzeba być w trybie „game over”.

Więcej zdjęć z gameplay’u


Film prezentujący krótki gameplay znajduje się tutaj: https://youtu.be/aLSygoTT584

Podsumowanie

Uwagi krytyczne

Projekt wykonany został dość słabo i pokrył małą część funkcjonalności, które można by tu wprowadzić. Stanowczo zabrakło nam czasu na uprzątniecie kodu projektu. Dodatkowo wykonanie takiej analizy kodu, jak w tej dokumentacji powinniśmy zrobić wcześniej, aby wyłapać, już na etapie pisania, nieścisłości, jakie nam się wkradły.

Możliwości dalszego rozwoju

- Poprawienie nieścisłości znalezionych w trakcie analizy kodu
- Posprzątanie kodu
- Podzielenie kodu na moduły
  Bez problemu można by podzielić ten kod na odpowiednie moduły odpowiedzialne za np. obsługę VGA, obsługę ADC, logikę gry i rysowanie na monitorze.
- Wprowadzenie zmiany trudności gry stopniowo, im dłużej gracz gra
  Zmiana trudności mogłaby polegać na stopniowym przyśpieszaniu bomb.
- Wykorzystanie ekranu LCD do wyświetlania punktów zależnych od tego jak długo gra gracz
  Najprościej licząc ilość klatek, ile gracz już gra.
- Wprowadzenie kilku trybów rozgrywki
  Parametrem rozgrywki mógłby być rozmiar gracza i czas po jakim gra przyśpiesza. Zmiana parametrów mogła by się odbywać za pomocą 3 wolnych przycisków.

Spis literatury

- Repozytorium z naszym projektem - https://github.com/baatochan/WAGDGame
- Strona producenta - https://www.xilinx.com/
- Podręcznik użytkownika (Spartan-3E FPGA Starter Kit Board User Guide, User Guide, ug230.pdf) - https://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf
- Strona z modułami dr Sugiera - http://www.zsk.ict.pwr.wroc.pl/zsk_ftp/fpga/
- VGA Signal Timings - http://tinyvga.com/vga-timing/800x600@72Hz