FPGA

Technika cyfrowa

Sprawozdanie 4

Natalia Brzozowska

ZADANIE 1.

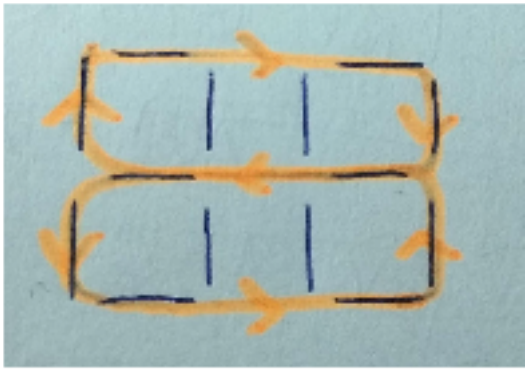
Polecenie:

Korzystając z programu Quartus II firmy Intel (www.intel.com), należy zrealizować praktycznie w układzie FPGA (np. na zestawie UP2 dostępnym w laboratorium) projekt wyświetlający na dwóch wyświetlaczach siedmiosegmentowych efekt węża, pokazany na filmie.
Projekt należy wykonać dwoma sposobami:

- w języku SytemVerilog
- w języku VHDL

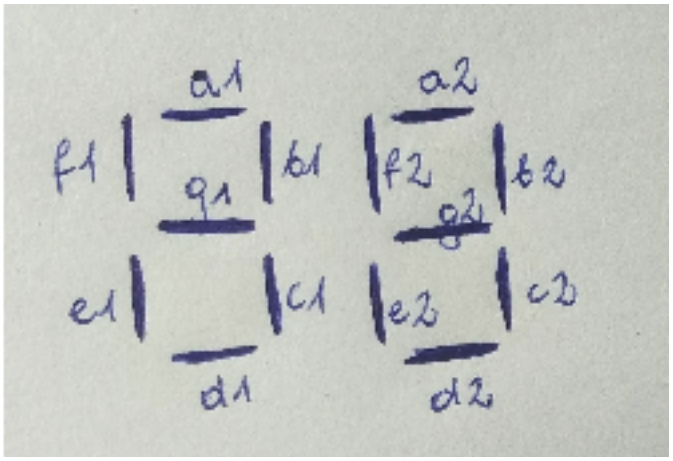
Projekt:

Wąż pokazany na filmie "porusza" się tak jak pokazano na rysunku 1.



Rysunek 1. Kierunek "poruszania" się węża po wyświetlaczach.

W jednym momencie (w jednej jednostce czasu) świecą się trzy segmenty z czternastu segmentów, dwóch wyświetlaczy siedmiosegmentowych. Litery oznaczające segmenty pierwszego wyświetlacza można przedstawić z cyfrą 1, a drugiego z cyfrą 2 (rysunek 2.).



Rysunek 2. Oznaczenia segmentów dwóch wyświetlaczy siedmiosegmentowych.

Wtedy powinny być zapalane kolejno segmenty:

a1,a2,b2,g2,g1,e1,d1,d2,c2,g2,g1,f1…

Po segmencie f1 powinny być zapalone znowu wszystkie segmenty od początku i tak ciągle. Wąż nie będzie w ogóle korzystał z segmentów b1,c1,f2,e2 - będą one ciągle zgaszone.

W jednym momencie mają być wyświetlane tylko trzy segmenty, oznaczające węża i przy każdym "ruchu" jeden z segmentów będzie gasł (ogon węża), a segment sąsiedni do "głowy" węża zapalał się. Segmenty zapalone w danym momencie można przedstawić tak jak w tabeli 1.

Nr	Zapalone segmenty
0	a1,a2,b2
1	a2,b2,g2
2	b2,g2,g1
3	g2,g1,e1
4	g1,e1,d1
5	e1,d1,d2
6	d1,d2,c2
7	d2,c2,g2
8	c2,g2,g1
9	g2,g1,f1
10	g1,f1,a1
11	f1,a1,a2

Tabela 1. Kolejność zapalania segmentów w danej chwili.

Do realizacji funkcji opisanej powyższą tabelą można wykorzystać licznik modulo 12, którego wyjścia będą sterować układ kombinacyjny odpowiedzialny za sterowanie wyświetlaczami. Jednak w języku SystemVerilog, jak i w języku VHDL, można skorzystać z behawioralnego opisu układu, przez co program sam dobierze odpowiednie komponenty, które będą wykonywały opisane funkcje. Nie trzeba zatem ręcznie projektować układu kombinacyjnego.

Tabelę 1 można zapisać także za pomocą tabeli prawdy:

Nr	a1	a2	b2	g2	g1	e1	d1	d2	c2	f1	b1	c1	f2	e2
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	1	0	0	0	0	0	0	0	0	0
3	0	0	0	1	1	1	0	0	0	0	0	0	0	0
4	0	0	0	0	1	1	1	0	0	0	0	0	0	0
5	0	0	0	0	0	1	1	1	0	0	0	0	0	0
6	0	0	0	0	0	0	1	1	1	0	0	0	0	0
7	0	0	0	1	0	0	0	1	1	0	0	0	0	0
8	0	0	0	1	1	0	0	0	1	0	0	0	0	0
9	0	0	0	1	1	0	0	0	0	1	0	0	0	0
10	1	0	0	0	1	0	0	0	0	1	0	0	0	0
11	1	1	0	0	0	0	0	0	0	1	0	0	0	0

Tabela 2. Tabela prawdy układu węża.

Pobrałam program Quartus II w wersji 9.0, w którym zaprojektowałam podany układ. Wybrałam moduł FPGA FLEX10K o numerze EPF10K70RC240-4.

1. Język SystemVerilog

Opisałam układ w języku SystemVerilog w sposób behawioralny - opisałam jak ma się zachowywać układ, w danych sytuacjach. W tym celu utworzyłam moduł snake przyjmujący dwa wejścia - clk (wejście zegarowe) i reset oraz czternaście wyjść - dla każdego segmentu wyświetlaczy. Zmienne określające stan na wyjściu są typu reg - zapamiętują wartość. Utworzyłam jeszcze jedną zmienną czterobitową (num), która będzie się zmieniała od 0 do 11 (licznik). W zależności od wartości tej zmiennej będą się zapalały odpowiednie segmenty.

Kod programu:

```
module snake(clk, reset,a1,a2,b1,b2,c1,c2,d1,d2,e1,e2,f1,f2,g1,g2);
input clk, reset;
output reg a1,a2,b1,b2,c1,c2,d1,d2,e1,e2,f1,f2,g1,g2;

reg [3:0] num;
```

W module snake zaimplementowałam dwa bloki proceduralne typu always. W pierwszym bloku, który będzie wykonywał się niejako w pętli(ponieważ tak działa blok always) znajduje się jedna instrukcja case. W tej instrukcji case, w zależności od wartości parametru num wykonają się odpowiednie instrukcje i do odpowiednich wyjść zostaną przekazane sygnały o określonej wartości. W wyniku tego zostaną zapalone odpowiednie segmenty wyświetlaczy. Za każdym razem trzy zmienne wyjściowe otrzymują wartość 1, a reszta wyjść przyjmuje wartość 0.

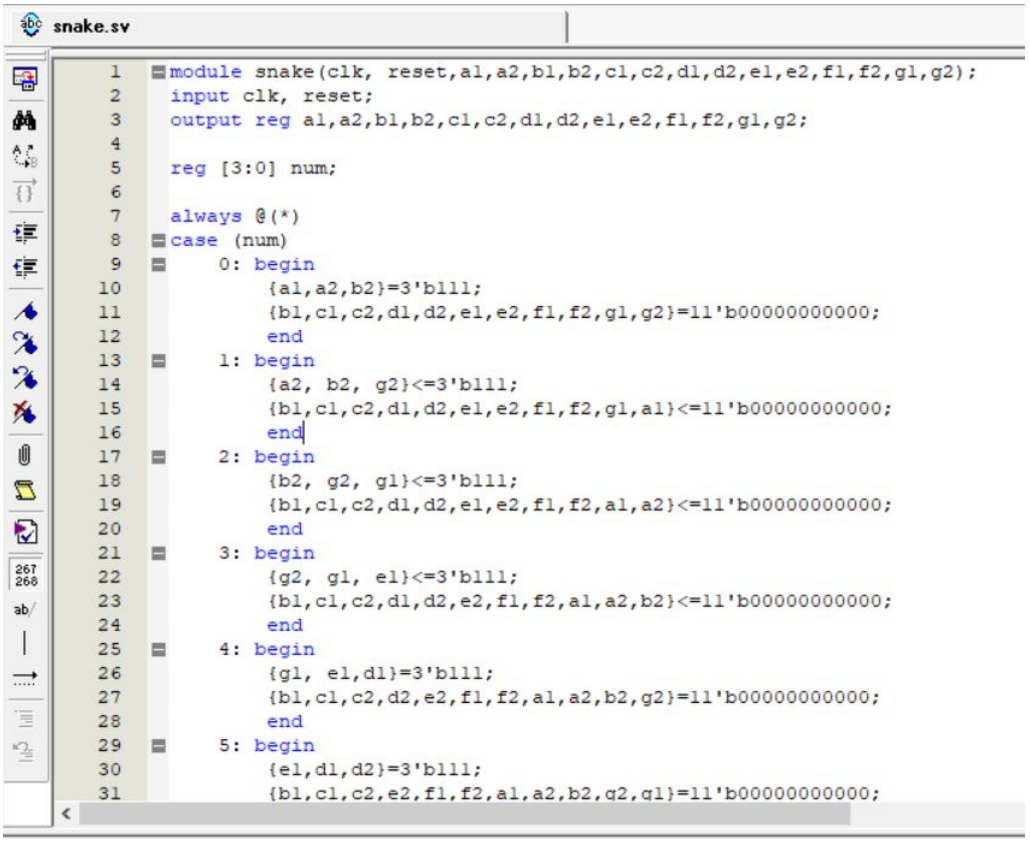
```
always @(*)
case (num)
      0: begin
            {a1,a2,b2}=3'b111;
            {b1,c1,c2,d1,d2,e1,e2,f1,f2,g1,g2}=11'b00000000000;
            end
      1: begin
            {a2, b2, g2}=3'b111;
            {b1,c1,c2,d1,d2,e1,e2,f1,f2,g1,a1}=11'b00000000000;
            end
      2: begin
            {b2, g2, g1}=3'b111;
            {b1,c1,c2,d1,d2,e1,e2,f1,f2,a1,a2}=11'b00000000000;
            end
      3: begin
            {g2, g1, e1}=3'b111;
            {b1,c1,c2,d1,d2,e2,f1,f2,a1,a2,b2}=11'b00000000000;
            end
      4: begin
            {g1, e1,d1}=3'b111;
            {b1,c1,c2,d2,e2,f1,f2,a1,a2,b2,g2}=11'b00000000000;
            end
      5: begin
```

```
            {e1,d1,d2}=3'b111;
            {b1,c1,c2,e2,f1,f2,a1,a2,b2,g2,g1}=11'b00000000000;
            end
      6: begin
            {d1,d2,c2}=3'b111;
            {b1,c1,e2,f1,f2,a1,a2,b2,g2,g1,e1}=11'b00000000000;
            end
      7: begin
            {d2,c2,g2}=3'b111;
            {b1,c1,e2,f1,f2,a1,a2,b2,g1,e1,d1}=11'b00000000000;
            end
      8: begin
            {c2,g2,g1}=3'b111;
            {b1,c1,e2,f1,f2,a1,a2,b2,e1,d1,d2}=11'b00000000000;
            end
      9: begin
            {g2,g1,f1}=3'b111;
            {b1,c1,e2,f2,a1,a2,b2,e1,d1,d2,c2}=11'b00000000000;
            end
      10: begin
            {g1,f1,a1}=3'b111;
            {b1,c1,e2,f2,a2,b2,e1,d1,d2,c2,g2}=11'b00000000000;
            end
      11: begin
            {f1,a1,a2}=3'b111;
            {b1,c1,e2,f2,b2,e1,d1,d2,c2,g2,g1}=11'b00000000000;
            end
      default:
            {a1,a2,b1,b2,c1,c2,d1,d2,e1,e2,f1,f2,g1,g2}=14'b00000000000000;
endcase
```

Drugi blok proceduralny typu always będzie się wykonywał, wtedy gdy sygnał na zboczu zegara będzie narastający. Wtedy jeśli reset będzie równy 1, czyli układ zostanie włączony, num przyjmie wartość 0 - ustawienie wartości początkowej układu. Na każdym kolejnym zboczu narastającym zegara, num będzie się zwiększał, aż do 11. Gdy osiągnie wartość 11 znów zostanie wyzerowany. Ten blok opisuje licznik modulo 12.

```
always @(posedge clk)
      if(reset) num<=0;
      else
            if(num==11) num<=0;
            else num<=num+1;
endmodule
```

Zaprojektowany układ można rozumieć jako automat, w którym pierwszy blok always realizuje funkcje wyjść, a drugi funkcje przejść. Zmienna num reprezentuje stan automatu. Jest to automat Moore'a, ponieważ wyjście zależy od stanu automatu.



Okno 1. Kod w języku SystemVerilog w programie Quartus II.

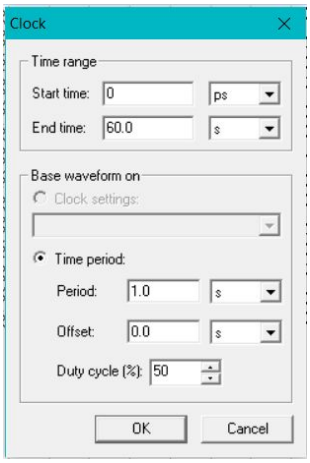
Powyżej zamieściłam zrzut ekranu z fragmentem kodu w programie Quartus II.

Flow Status	Successful - Wed Jun 17 16:02:11 2020
Quartus II Version	9.0 Build 184 04/29/2009 SP 1 SJ Web Edition
Revision Name	snake
Top-level Entity Name	snake
Family	FLEX10K
Device	EPF10K70RC240-4
Timing Models	Final
Met timing requirements	Yes
Total logic elements	16 / 3,744 (< 1 %)
Total pins	16 / 189 (8 %)
Total memory bits	0 / 18,432 (0 %)

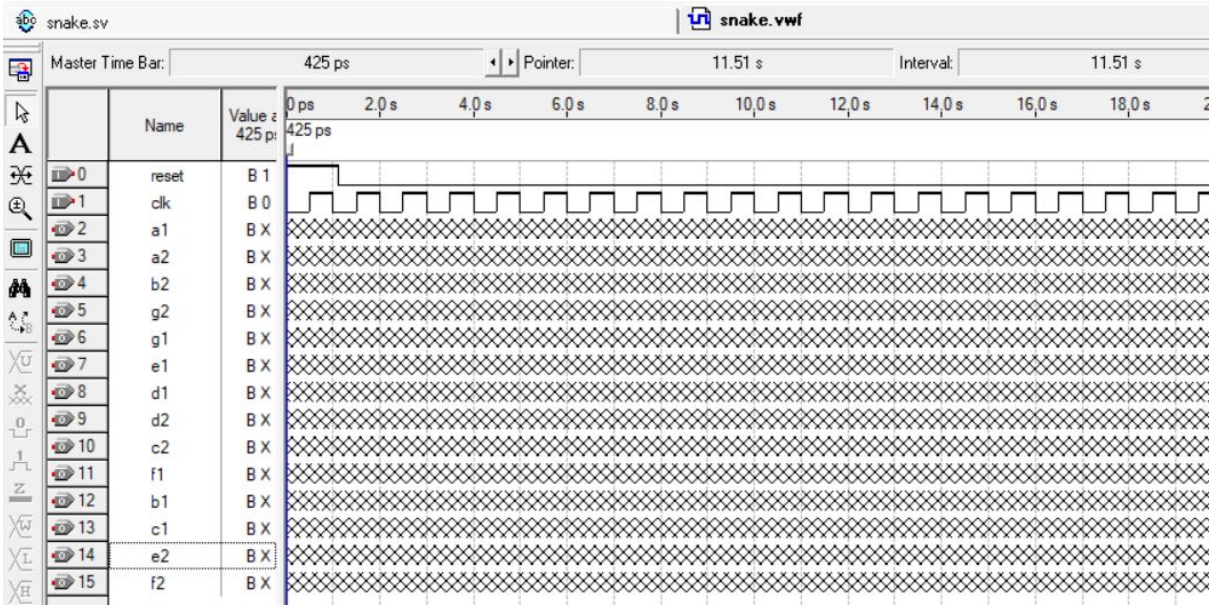
Okno 2. Informacja dotycząca kompilacji.

Program kompiluje się poprawnie (bez error'ów).

Utworzyłam również plik typu Vector Waveform, w celu przeprowadzenia symulacji układu. Zmieniłam End Time (60 s) symulacji oraz Grid Size(1 s). Dodałam do symulacji wszystkie wyjścia i wejścia programu. Do wejścia clk (zegar) przyporządkowałam opcję Clock i nadałam odpowiednie wartości (zostały pokazane na obrazie Okno 2). Wejście reset jest równe 1 dla pierwszego zbocza narastającego zegara (clk), później przez cały czas jest równe zero.



Okno 3. Ustawienie wartości zegara dla symulacji.



Okno 4. Plik snake.vwf zawierający konfigurację symulacji.

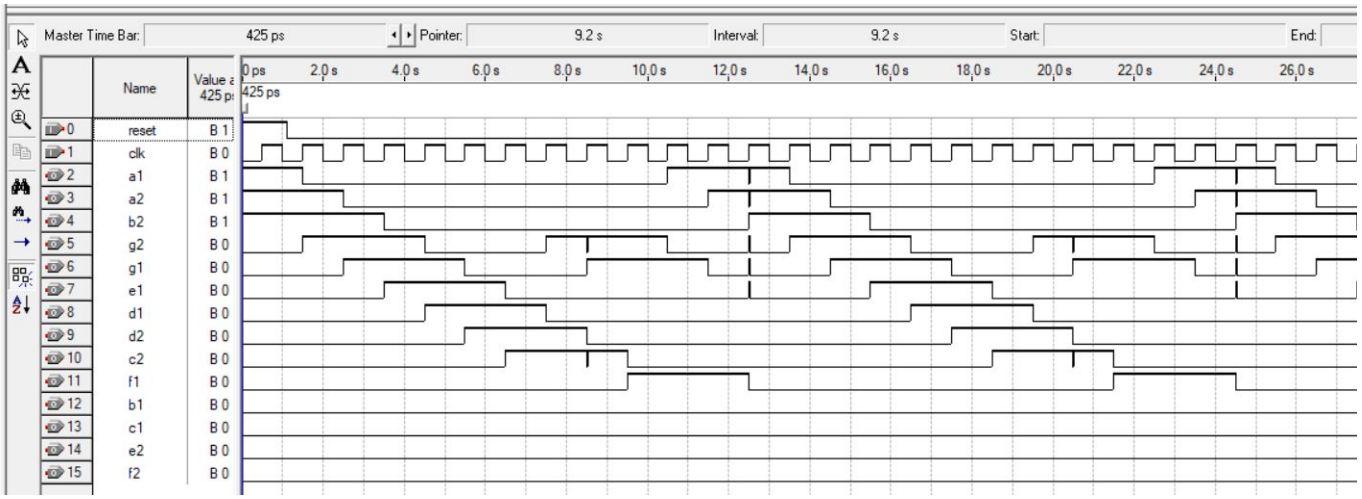
Do każdego wejścia i wyjścia dopasowałam odpowiedni PIN w urządzeniu.

Named: * Edit:

	Node Name	Direction	Location	I/O Bank
1	a1	Output	PIN_6	
2	a2	Output	PIN_17	
3	b1	Output	PIN_7	
4	b2	Output	PIN_18	
5	c1	Output	PIN_8	
6	c2	Output	PIN_19	
7	clk	Input	PIN_83	
8	d1	Output	PIN_9	
9	d2	Output	PIN_20	
10	e1	Output	PIN_11	
11	e2	Output	PIN_21	
12	f1	Output	PIN_12	
13	f2	Output	PIN_23	
14	g1	Output	PIN_13	
15	g2	Output	PIN_24	
16	reset	Input	PIN_28	
17	<<new node>>			

Okno 5. Wejścia i wyjścia układu dopasowane do odpowiednich PIN-ów.

Po przeprowadzeniu symulacji uzyskałam wyniki przedstawione na poniższym obrazie.



Okno 6. Wyniki symulacji - przebiegi na wyjściach i wejściach układu.

Na powyższym zrzucie ekranu widać dwa pełne "obiegi" węża - zmienna num w kodzie wzrosła od 0 do 11 dwa razy. Jeden obieg, zgodnie z założeniami, trwał jedenaście taktów zegara. Sygnał na wyjściach zmienia się tak jak powinien - z każdym kolejnym zboczem narastającym zegara sygnał na jednym z wejść zmienia się z wysokiego na niski, a na innym z niskiego na wysoki. Gdzieniegdzie występują hazardy - niewielkie zmiany sygnałów na przeciwny, wtedy gdy zbocze sygnału zegara jest narastające. Nie powinny one jednak zaburzyć działania programu. Symulacja pokazuje jak wąż powinien "iść" po wyświetlaczu.

2. Język VHDL

Utworzyłam nowy projekt i napisałam program analogiczny do tego zaimplementowanego w języku SystemVerilog. Działa na tej samej zasadzie, co wcześniej opisany program. Zmieniła się jednak składnia języka, przez co kod wygląda nieco inaczej.
Do pierwszego procesu (analogicznego do bloku proceduralnego always w języku SystemVerilog) jest teraz przekazywana zmienna typu signal. Jeśli zmienna ulegnie zmianie to blok się wykona. Wartość tej zmiennej jest zmieniana w drugim procesie liczącym od 0 do 11.
Zmienna num, której wartość jest zmienną licznikową, również zmienianą w drugim procesie, jest zdefiniowana jako zmienna shared, po to by oba procesy mogły z niej korzystać.

Poniżej zamieszczam cały kod programu węża.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity snake_vhdl is
port(
	reset, clk: in std_logic;
	a1,a2,b1,b2,c1,c2,d1,d2,e1,e2,f1,f2,g1,g2: out std_logic
	);
end snake_vhdl;

architecture arch1 of snake_vhdl is
shared variable num : integer;
signal sig : std_logic := '0';
begin

	process(sig)
	begin
	case num is
		when 0 =>
			a1<='1';
			a2<='1';
			b2<='1';
			g2<='0';
			g1<='0';
			e1<='0';
			d1<='0';
			d2<='0';
			c2<='0';
			f1<='0';
			b1<='0';
			c1<='0';
			f2<='0';
			e2<='0';
		when 1 =>
			a1<='0';
			a2<='1';
```

```
                b2<='1';
                g2<='1';
                g1<='0';
                e1<='0';
                d1<='0';
                d2<='0';
                c2<='0';
                f1<='0';
                b1<='0';
                c1<='0';
                f2<='0';
                e2<='0';
        when 2 =>
                a1<='0';
                a2<='0';
                b2<='1';
                g2<='1';
                g1<='1';
                e1<='0';
                d1<='0';
                d2<='0';
                c2<='0';
                f1<='0';
                b1<='0';
                c1<='0';
                f2<='0';
                e2<='0';
        when 3 =>
                a1<='0';
                a2<='0';
                b2<='0';
                g2<='1';
                g1<='1';
                e1<='1';
                d1<='0';
                d2<='0';
                c2<='0';
                f1<='0';
                b1<='0';
                c1<='0';
                f2<='0';
                e2<='0';
        when 4 =>
                a1<='0';
                a2<='0';
                b2<='0';
                g2<='0';
                g1<='1';
                e1<='1';
                d1<='1';
```

```
                d2<='0';
                c2<='0';
                f1<='0';
                b1<='0';
                c1<='0';
                f2<='0';
                e2<='0';
        when 5 =>
                a1<='0';
                a2<='0';
                b2<='0';
                g2<='0';
                g1<='0';
                e1<='1';
                d1<='1';
                d2<='1';
                c2<='0';
                f1<='0';
                b1<='0';
                c1<='0';
                f2<='0';
                e2<='0';
        when 6 =>
                a1<='0';
                a2<='0';
                b2<='0';
                g2<='0';
                g1<='0';
                e1<='0';
                d1<='1';
                d2<='1';
                c2<='1';
                f1<='0';
                b1<='0';
                c1<='0';
                f2<='0';
                e2<='0';
        when 7 =>
                a1<='0';
                a2<='0';
                b2<='0';
                g2<='1';
                g1<='0';
                e1<='0';
                d1<='0';
                d2<='1';
                c2<='1';
                f1<='0';
                b1<='0';
                c1<='0';
```

```
                f2<='0';
                e2<='0';
            when 8 =>
                a1<='0';
                a2<='0';
                b2<='0';
                g2<='1';
                g1<='1';
                e1<='0';
                d1<='0';
                d2<='0';
                c2<='1';
                f1<='0';
                b1<='0';
                c1<='0';
                f2<='0';
                e2<='0';
            when 9 =>
                a1<='0';
                a2<='0';
                b2<='0';
                g2<='1';
                g1<='1';
                e1<='0';
                d1<='0';
                d2<='0';
                c2<='0';
                f1<='1';
                b1<='0';
                c1<='0';
                f2<='0';
                e2<='0';
            when 10 =>
                a1<='1';
                a2<='0';
                b2<='0';
                g2<='0';
                g1<='1';
                e1<='0';
                d1<='0';
                d2<='0';
                c2<='0';
                f1<='1';
                b1<='0';
                c1<='0';
                f2<='0';
                e2<='0';
            when 11 =>
                a1<='1';
                a2<='1';
```

```
                    b2<='0';
                    g2<='0';
                    g1<='0';
                    e1<='0';
                    d1<='0';
                    d2<='0';
                    c2<='0';
                    f1<='1';
                    b1<='0';
                    c1<='0';
                    f2<='0';
                    e2<='0';
             when others =>
                    a1<='0';
                    a2<='0';
                    b2<='0';
                    g2<='0';
                    g1<='0';
                    e1<='0';
                    d1<='0';
                    d2<='0';
                    c2<='0';
                    f1<='0';
                    b1<='0';
                    c1<='0';
                    f2<='0';
                    e2<='0';
      end case;
end process;


process (clk)
begin
      if rising_edge(clk) then
            sig<=not(sig);
            if(reset='1')then num:=0;
            else
                  if(num=11) then num:=0;
                  else num:=num+1;
                  end if;
            end if;
      end if;
end process;
end arch1;
```

snake_vhdl.vhd | Compilation Report - Flow

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.all;
3
4   entity snake_vhdl is
5   port(
6       reset, clk: in std_logic;
7       a1,a2,b1,b2,c1,c2,d1,d2,e1,e2,f1,f2,g1,g2: out std_logic
8       );
9   end snake_vhdl;
10
11  architecture arch1 of snake_vhdl is
12  shared variable num : integer;
13  signal sig : std_logic := '0';
14  begin
15
16      process(sig)
17      begin
18      case num is
19          when 0 =>
20              a1<='1';
21              a2<='1';
22              b2<='1';
23              g2<='0';
24              g1<='0';
25              e1<='0';
26              d1<='0';
27              d2<='0';
28              c2<='0';
29              f1<='0';
30              b1<='0';
```

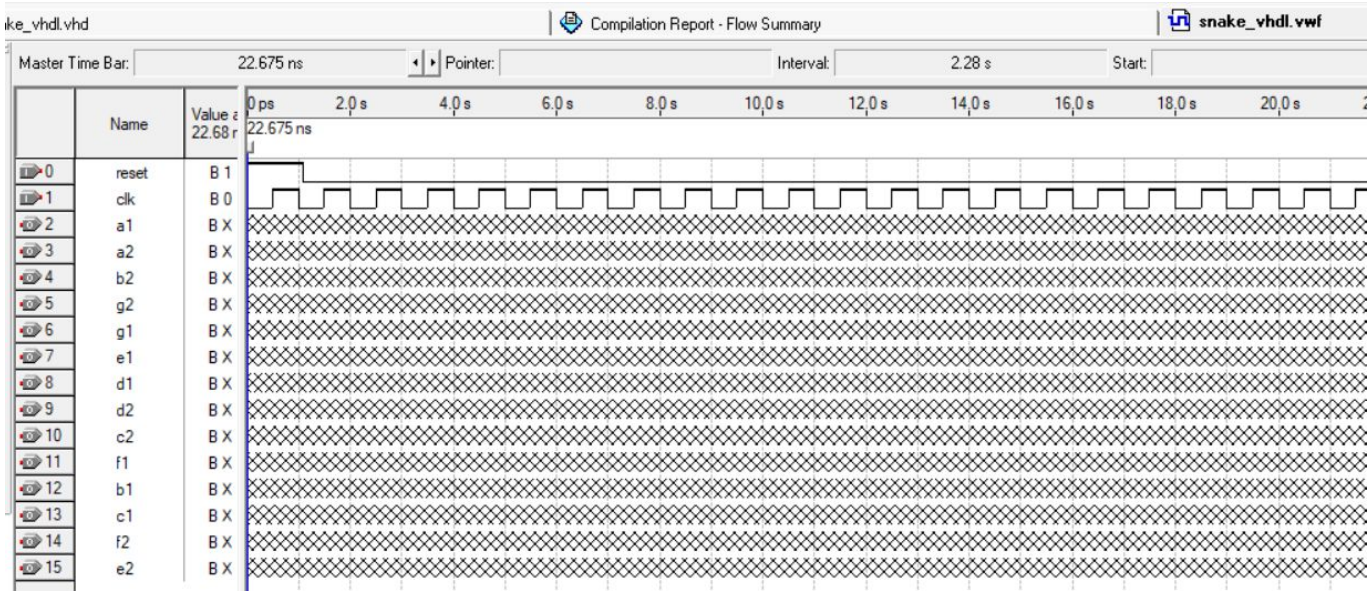
Okno 7. Fragment kodu w języku VHDL w programie Quartus II.

Kompilacja przebiegła pomyślnie:

Flow Status	Successful - Wed Jun 17 17:22:58 2020
Quartus II Version	9.0 Build 184 04/29/2009 SP 1 SJ Web Edition
Revision Name	snake_vhdl
Top-level Entity Name	snake_vhdl
Family	FLEX10K
Device	EPF10K70RC240-4
Timing Models	Final
Met timing requirements	Yes
Total logic elements	60 / 3,744 (2 %)
Total pins	16 / 189 (8 %)
Total memory bits	0 / 18,432 (0 %)

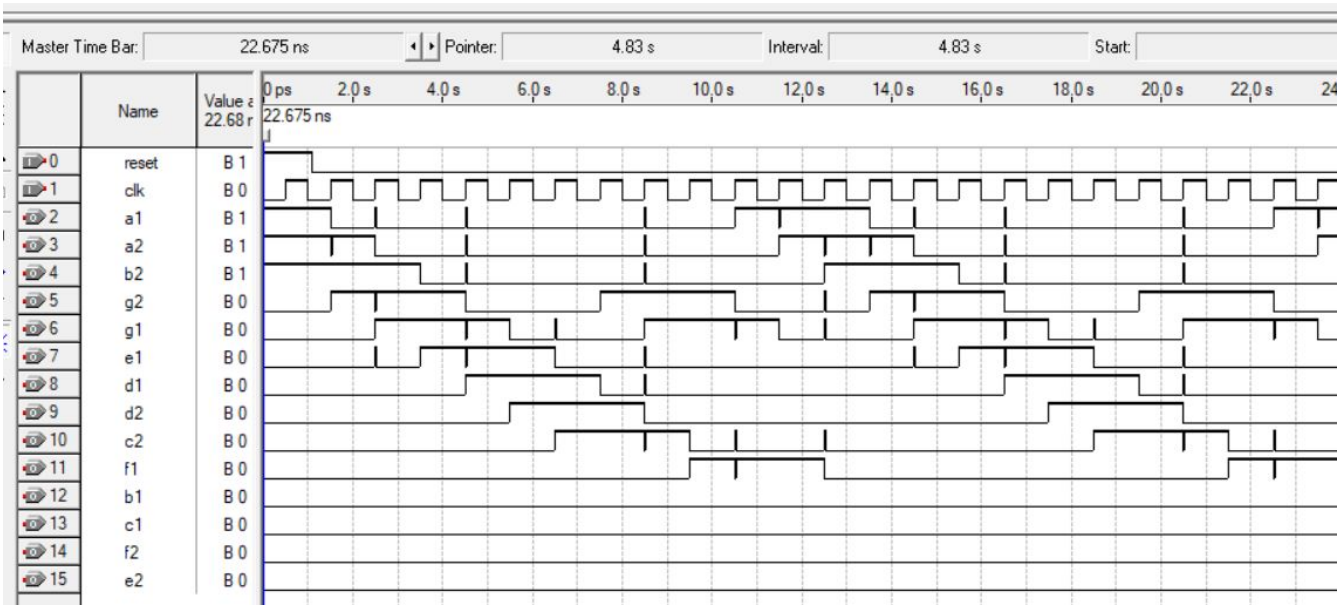
Okno 8. Informacja dotycząca kompilacji programu w języku VHDL.

Tak jak wcześniej, utworzyłam plik typu Vector Waveform potrzebny do symulacji. Dodałam do pliku wejścia i wyjścia układu. Do wejść przyporządkowałam sygnały takie jak do wcześniej opisywanego pliku.



Okno 9. Plik snake_vhdl.vwf zawierający konfigurację symulacji.

Podpiełam odpowiednie PINy do odpowiednich wejść i wyjść (tak jak wcześniej). Następnie uruchomiłam symulację.

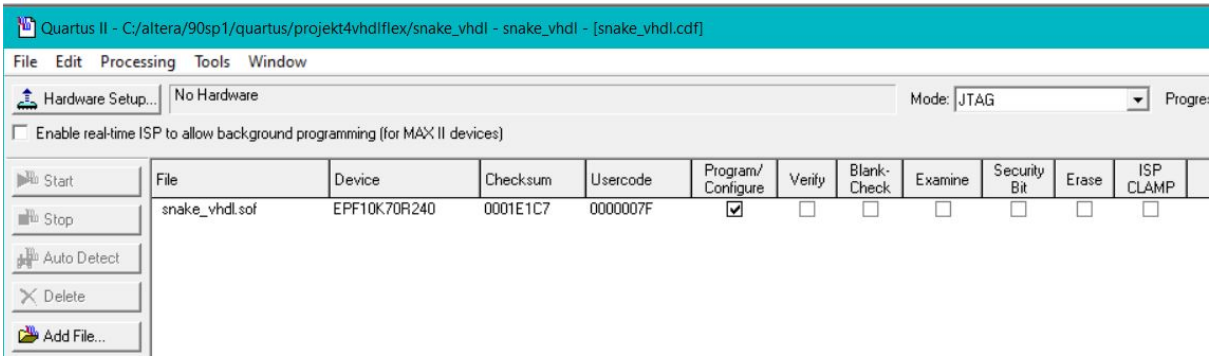


Okno 10. Wynik symulacji programu w języku VHDL.

Wyniki są prawie identyczne jak wcześniej. Sygnały na wyjściach zmieniają się tak jak powinny, za wyjątkiem zmian sygnału na przeciwny (tzw. hazardy) w czasie ok. 0.6 ns w niektórych momentach, gdy sygnał zegarowy zmienia się na wysoki (zbocze narastające). Hazardy występowały również przy wcześniejszej implementacji, jednak było ich mniej. Przyczyną hazardów są opóźnienia bramek w układzie kombinacyjnym. Jednak minimalny czas tych zmian, nie spowoduje widocznych zaburzeń w działaniu układu.

Programowanie układu FPGA:

Tak zaprojektowane układy można zaprogramować na płytce FPGA, korzystając z możliwości jakie daje oprogramowanie Quartus II. Po podpięciu FPGA do komputera i skonfigurowaniu wtyczek powinna pojawić się możliwość wyboru tego sprzętu, by go zaprogramować. Następnie powinna być możliwość wybrania przycisku Start, który rozpoczęłby programowanie układu.



Okno 11. Okno umożliwiające wybór FPGA i zaprogramowanie sprzętu .

Podsumowanie i wnioski:

Wykonane zadania pozwoliły mi "dotknąć" dziedziny jaką jest programowanie układów cyfrowych. Za pomocą języków SystemVerilog i VHDL można w dość łatwy sposób zaprogramować swój własny układ. Szczególnie przydatny i przyjazny wydaje się być opis behawioralny, dzięki któremu można zaprogramować układ nie znając (lub znając w niewielkim stopniu) poszczególne elementy elektroniczne. Język SystemVerilog wydał mi się przyjaźniejszy, dla moich potrzeb. W moim odczuciu język VHDL jest bardziej rozbudowany, przez co może być bardziej przydatny przy realizacji dużych projektów. Wtedy można by wykorzystać możliwości tego języka.

Programowanie układów cyfrowych wydaje się dość podobne do programowania komputera, z którym mamy najczęściej styczność (np. w języku C). Jednak trzeba pamiętać, że programując układ cyfrowy poszczególne bloki proceduralne (always/process) są wykonywane w pełni równolegle. Są realizowane przez odpowiednie elementy elektroniczne, których właściwością jest to że działają w sposób ciągły i równoległy (o ile tylko są zasilane).

Oprogramowanie Quartus II ułatwia i pomaga w zaprogramowaniu układu.

Zastosowania:

Układy FPGA można wykorzystać do utworzenia prawie każdego układu, który sobie wymyślimy. Przy pomocy języków SystemVerilog oraz VHDL można w dość łatwy sposób zaprogramować taki układ FPGA. Dodatkowo, zakup płytki FPGA jest możliwy i dostępny dla prawie każdej zainteresowanej osoby, tak jak oprogramowanie Quartus II, czy inne środowiska do programowania układów. Dzięki temu istnieje możliwość zaprogramowania własnego układu, właściwie w warunkach domowych. Dzięki temu można zrobić np. układ sterujący światłem w pokoju, czy radiem.