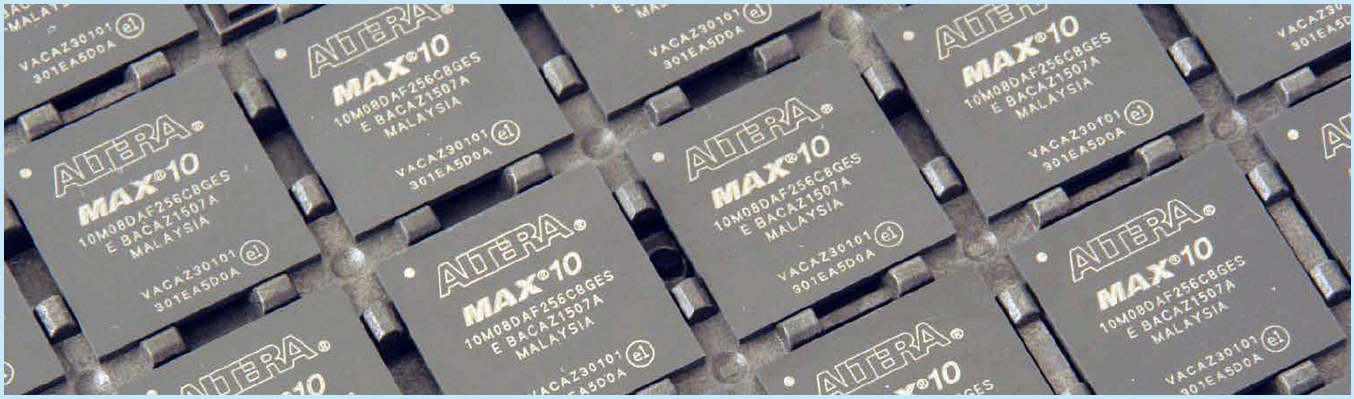


Eksperymenty z FPGA (16)

Pamięć RAM

W poprzednim odcinku rozważaliśmy, w jaki sposób można w układzie FPGA zrealizować obliczenia iteracyjne. Jako przykład użyliśmy transformaty Fouriera. Dzisiaj przygotujemy implementację, a następnie przetestujemy ją w symulacji i sprzęcie. Przy okazji dowiemy się, jak modelować pamięć RAM. Wszystkie przykłady można pobrać z repozytorium https://gitlab.com/rysy_core/fpga-experiments.

Na początku przygotujemy jedne z głównych bloków nowego projektu, czyli dwuportową pamięć RAM. Tego typu blok nosi po angielsku nazwę *simple dual-port*. Oznacza to, że możliwy jest równoczesny zapis i odczyt. Nie można jednak wykonać dwóch zapisów albo dwóch odczytów. Pamięć o takich możliwościach nazywa się *true dual-port*. Moduły pamięci M9K, w które wyposażony jest nasz układ FPGA, są właśnie tego drugiego typu, jednak nam obecnie nie jest potrzebna jego pełna funkcjonalność.

Kod SystemVerilog z **listingu 1** modeluje zachowanie naszej pamięci. Mamy dwa parametry:

- K – liczbę bitów w słowie,
- N – liczbę słów pamięci.

Dodatkowo obliczamy zmienną `LOG_N`, która oznacza liczbę bitów potrzebnych do przechowania adresu.

Dalej znajdziemy wejścia i wyjścia. Pierwsze z nich (clk) to zegar. Dalej znajdziemy port odpowiedzialny za zapis:

- `addr_wr` – adres słowa w pamięci,
- `data_wr` – dane do zapisania,
- `wr` – flaga oznaczająca, że dane są poprawne i mają zostać wpisane.

Pozostałe tworzą port pozwalający na odczyt:

- `addr_rd` – adres w pamięci,
- `data_rd` – odczytane dane.

Następnie znajdziemy pomocnicze sygnały. Najważniejsza z nich jest tablica data, w której będą przechowywane dane. Dalej znajdziemy bloki logiki sekwencyjnej. Pierwszy z nich (linie 28...33) realizuje opóźnienie wszystkich wejść o jeden takt zegara. Drugi (wiersze 35...37) realizuje zapis. Jak widzimy, wpis do pamięci zajmuje dwa cykle zegara.

Na samym końcu mamy odczyt. Odpowiednie słowo zostaje wybrane z pamięci i wpisane na wyjście. Widzimy więc, że wybrane dane znajdują się na wyjściu na drugim cyklu zegara po pojawieniu się adresu na wejściu.

Ostatnim problemem, który tylko zasygnalizuję, jest zachowanie, gdy następuje odczyt i zapis pod ten sam adres (*read-during-write*). Inofrmację o wybranym trybie znajdziemy w logach programu Quartus:

```
Info (286033): Parameter READ_DURING_WRITE_MODE_
MIXED_PORTS set to OLD_DATA
```

Więcej informacji na temat bloków M9K znajdziemy w [1].

Aby sprawdzić działanie naszego kodu, powstał krótki testbench. Jego fragmenty znajdziemy w **listingu 2**. W liniach 27...35 znajdziemy generowanie danych wejściowych. Do każdej komórki pamięci wpiszemy wartość jej adresu. Następnie w drugiej części (45...55) odczytamy każdą z zapisanych wartości. Za pomocą polecenia assert (linia 51) sprawdzamy, czy uzyskaliśmy poprawny wynik. Jeżeli nie, zostanie wypisany odpowiedni komunikat o błędzie.

Listing 1. Implementacja pamięci RAM (*14_iterative_fft/ram.sv*)

```
10 module ram #(
11  parameter N = 32,
12  parameter LOG_N = $clog2(N),
13  parameter K = 9
14 ) (
15  input wire clk,
16  input wire wr,
17  input wire [LOG_N-1:0]addr_wr,
18  input wire [K-1:0]data_wr,
19  input wire [LOG_N-1:0]addr_rd,
20  output logic [K-1:0]data_rd
21 );
22  logic wr_r;
23  logic [LOG_N-1:0]addr_wr_r;
24  logic [K-1:0]data_wr_r;
25  logic [LOG_N-1:0]addr_rd_r;
26  logic [K-1:0]data[N-1:0];
27
28  always_ff @(posedge clk) begin
29      wr_r <= wr;
30      addr_wr_r <= addr_wr;
31      data_wr_r <= data_wr;
32      addr_rd_r <= addr_rd;
33  end
34
35  always_ff @(posedge clk)
36      if (wr_r)
37      data[addr_wr_r] <= data_wr_r;
38
39  always_ff @(posedge clk)
40      data_rd <= data[addr_rd_r];
41
42 endmodule
```

Jak zwykle symulację możemy uruchomić w programie ModelSim za pomocą polecenia:

```
do ./ram.do
```

Dzięki temu, że na jego końcu została dodania linia:

```
coverage report -assert
```

W logach symulacji dostaniemy raport na temat asercji. Jego treść znajdziemy w **listingu 3**.

Skupmy się jednak na **rysunku 1**, który prezentuje przebiegi czasowe. Sygnały zostały podzielone na cztery grupy. Na samej górze mamy zegar, następnie port zapisujący, dane, a na końcu odczyt. Zachęcam do samodzielnego przeanalizowania, na których zboczach zegara następuję zapisanie, a kiedy odczyt danych.

Aby zobaczyć, jak pamięć RAM jest realizowana w sprzęcie, przygotujemy nowy projekt w środowisku Quartus (*14_iterative_fft / ram.qpf*). Zawiera on tylko nasz nowy moduł. Nie pełni więc żadnej sensownej funkcji. Pokaże nam tylko, jak nasz moduł zostanie zaimplementowany.

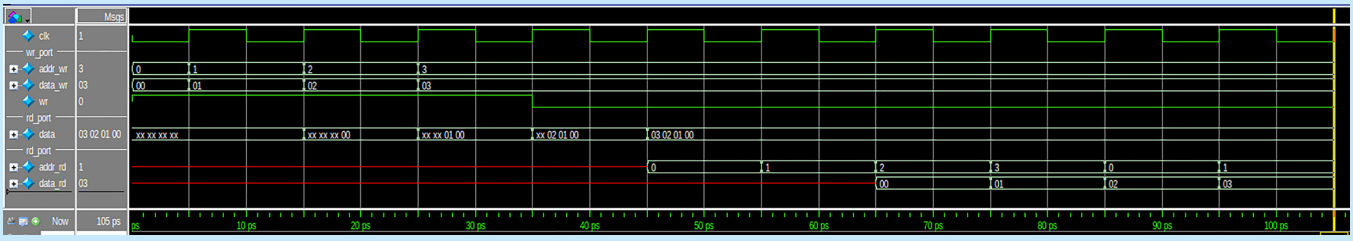
Na **rysunku 2** został zaprezentowany moduł RAM w widoku RTL. Widzimy tu rejestry na wejściach i wyjściach. Natomiast w środku mamy moduł SYNC_RAM. Widzimy więc, że środowisko poprawnie zinterpretowało nasz kod i wygenerowało blok pamięci RAM. Gdy przejdziemy do widoku Technology Map (Post-Fitting), zobaczymy już tylko jeden moduł. Oznacza to, że zostały wykorzystane jego wewnętrzne rejestry. Możemy się o tym przekonać, otwierając jego wewnętrzną konfigurację (**rysunek 4**). Widzimy na nim, że aby

Listing 2. Fragment testów pamięci RAM (*14_iterative_fft/ram_tb.sv*)

```
27 initial begin
28  wr = 1'b1;
29  for (int i=0; i<N; i++) begin
30      addr_wr = i;
31      data_wr = i;
32      @(posedge clk);
33  end
34  wr = 1'b0;
35 end
45 initial begin
46  repeat (5) @(posedge clk);
47  for (int i=0; i<N+LAT; i++) begin
48      addr_rd = i;
49      @(posedge clk);
50      if (i >= LAT)
51          assert (data_rd == i-LAT)
52      else $display("Wrong val %d != %d", data_rd, i-LAT);
53  end
54  $stop;
55 end
```

Listing 3. Raport z symulacji

```
# Coverage Report Summary Data by instance
#
# ================================================================================
# === Instance: /ram_tb
# === Design Unit: work.ram_tb
# ================================================================================
#     Enabled Coverage              Bins      Hits    Misses  Coverage
#     ----------------              ----      ----    ------  --------
#     Assertions                       1         1         0   100.00%
#
#
# TOTAL ASSERTION COVERAGE: 100.00%  ASSERTIONS: 1
#
# Total Coverage By Instance (filtered view): 100.00%
```



Rysunek 1. Przepływowa implementacja FFT

uzyskać pełną częstotliwość pracy, powinniśmy jeszcze dodać jeden przerzutnik na wyjściu z naszego modułu.

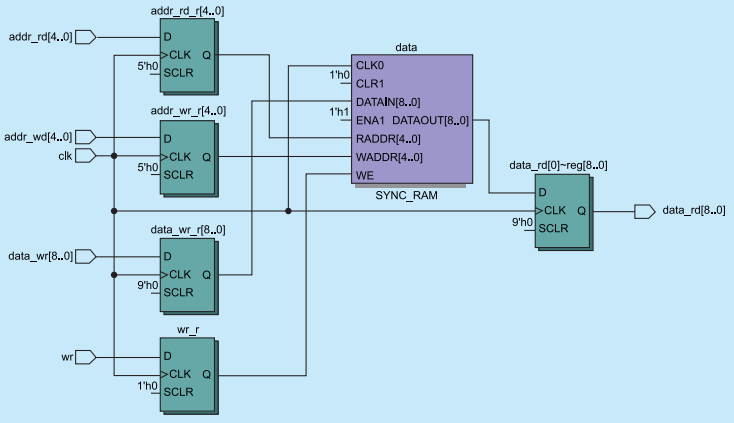
Implementacja FFT

Mamy już gotową pamięć RAM. Możemy więc przystąpić do składania naszego iteracyjnego procesora FFT. Cały kod znajduje się w pliku *14_iterative_fft/fft_iter.sv*. Tutaj przyjrzymy się tylko kilku jego fragmentom.

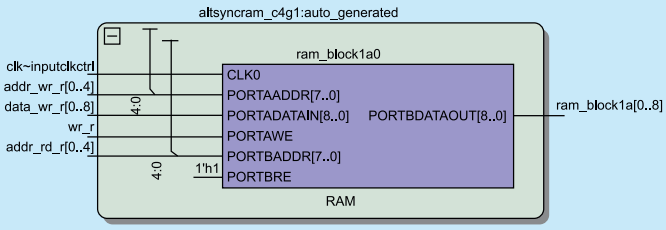
Na **listingu 4** widzimy interfejs naszego modułu. Tak jak w naszej przepływowej implementacji, przyjmuje on dwa parametry: `N` oznaczający długość trasformaty oraz `K`, czyli precyzję reprezentacji liczb. Dalej znajdziemy wejścia. Pierwsze dwa są standardowe: zegar `clk` i reset `rst`. Dalej znajdziemy liczbę zespoloną w postaci dwóch liczb `d_in_re` i `d_in_im`. Nowym elementem jest `addr_in`. Jest to adres, pod którym próbka zostanie zapisana w pamięci RAM. Procesowanie danych rozpocznie się przy każdej zmianie stanu sygnału `start` na przeciwny.

W przypadku wyjść dochodzi nam nowy sygnał ready, oznaczający, że przetwarzanie się skończyło i można rozpocząć zbieranie danych dla kolejnej transformaty. Dalej mamy po kolei zespolony sygnał `d_out_re`, `d_out_im`, numer próbki `addr_out` oraz sygnał poprawności danych `valid_out`.

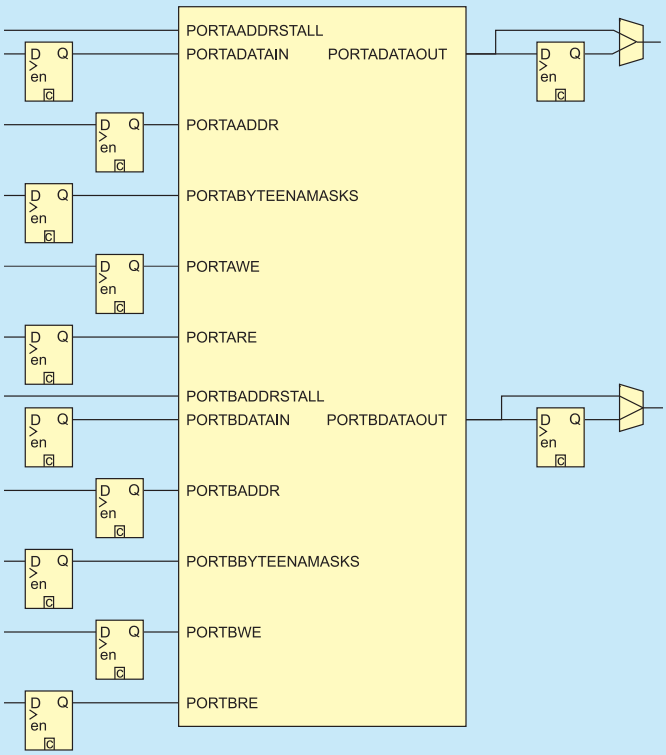
Na **listingu 5** znajdziemy sygnały kontrolne. W pierwszym bloku zatrzaskujemy w `start1` poprzedni stan sygnału `start`. Zmianę



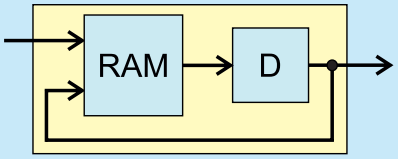
Rysunek 2. Moduł RAM w widoku RTL



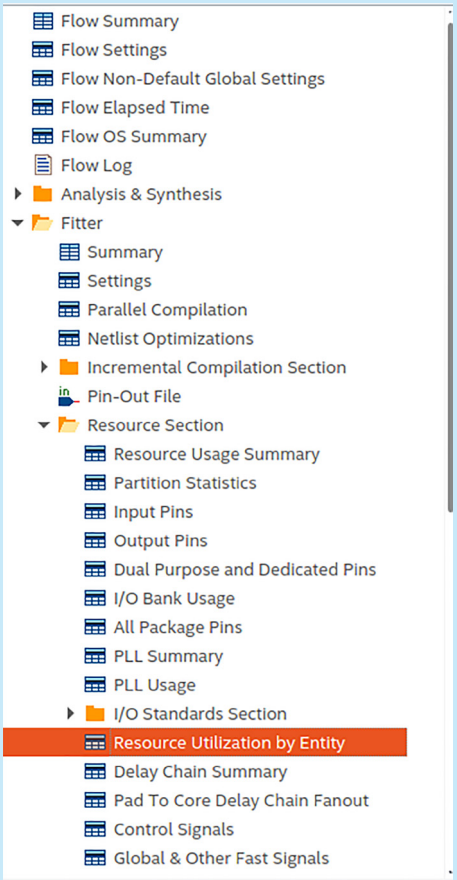
Rysunek 3. Blok RAM w widoku Technology Map (Post-Fitting)



Rysunek 4. Konfiguracja bloku M9K



Rysunek 5. Opóźnienie w pętli



Rysunek 6. Raport o zużyciu zasobów przez poszczególne moduły

Listing 4. Interfejs iteracyjnego procesora FFT (*14_iterative_fft/fft_iter.sv*)

```
10 module fft_iter #(
11  parameter K = 8,
12  parameter N = 2,
13  parameter LOG_N = $clog2(N),
14  parameter LLOG_N = $clog2(LOG_N)
15 ) (
16  input wire clk,
17  input wire rst,
18  input wire signed [K-1:0]d_in_re,
19  input wire signed [K-1:0]d_in_im,
20  input wire [LOG_N-1:0] addr_in,
21  input wire valid_in,
22  input wire start,
23  output logic ready,
24  output logic signed [K-1:0]d_out_re,
25  output logic signed [K-1:0]d_out_im,
26  output logic [LOG_N-1:0] addr_out,
27  output logic valid_out
28 );
```

Listing 5. Sygnały kontrolne (*14_iterative_fft/fft_iter.sv*)

```
56 always_ff @(posedge clk)
57  if (!rst)
58      start1 <= '0;
59  else
60      start1 <= start;
61
62 always_ff @(posedge clk)
63  if (!rst)
64      work <= '0;
65  else if (start != start1)
66      work <= 1'b1;
67  else if (work & mod_N_ov & mod_lN_ov)
68      work <= 1'b0;

72 counter #(.N(N)) mod_N (
73 .clk(clk),
74 .rst(rst),
75 .ce(work),
76 .q(i),
77 .ov(mod_N_ov));
78
79 counter #(.N(LOG_N)) mod_log_N (
80 .clk(clk),
81 .rst(rst),
82 .ce(work & mod_N_ov),
83 .q(n),
84 .ov(mod_lN_ov));
```

Listing 6. Wyznaczanie adresu odczytu pamięci (*14_iterative_fft/fft_iter.sv*)

```
86 always_comb
87  for (int ii = 0; ii < LOG_N; ii++)
88  if (ii > LOG_N-1-n)
89      ram_addr[ii] = i[ii];
90  else if (ii == LOG_N-1-n)
91      ram_addr[ii] = i[0];
92  else if (ii < LOG_N-1)
93      ram_addr[ii] = i[ii+1];
94  else
95      ram_addr[ii] = '0;
```

jego stanu wykrywamy za pomocą sprawdzenia, czy stan tych dwóch zmiennych jest różny (linia 65). Następuje wtedy ustawienie wewnętrznego sygnału `work`. Powoduje on, że liczniki `mod_N` i `mod_log_N` rozpoczną pracę. Sygnał `work` jest zerowany, gdy oba liczniki się przepełnią, czyli po NlogN cyklach.

Ostatni fragment kodu, który przeanalizujemy, to generowanie adresu odczytu z pamięci. Znajdziemy go na **listingu 6**. Implementuje on ideę zaprezentowaną w tablicy 3 z poprzedniego odcinka. Korzystamy z bloku `always_comb`, czyli tworzymy logikę kombinacyjną. Za pomocą pętli for dokonujemy przypisania kolejnych bitów adresu. Wynik naszej operacji znajdzie się w `ram_addr`.

Musimy jeszcze zastanowić się nad latencją. Uproszczony model naszego procesora został pokazany na **rysunku 5**. W poprzedniej części analizowaliśmy przypadek, gdy latencja obliczeń pomiędzy odczytem i zapisem do RAM-u była zerowa. W praktyce jednak zarówno dostęp do pamięci, jak i wykonywanie działań wymaga kilku cykli zegara. Na rysunku reprezentuje je blok z literą D. W naszym

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl

Tabela 1. Porównanie zużycia zasobów

	Przepływowa	Iteracyjna
Elementy logiczne	661	203
Rejestry	581	174
Bity pamieci	356	280
Bloki M9K	5	2
DSP 9x9	20	4

przypadku latencja jest równa 6. Oznacza to, że tyle cykli zegara minie, zanim nowa wartość zostanie zapisana do pamięci. Dla naszej implementacji oznacza to, że dopiero po tym czasie kolejna iteracja obliczeń będzie mogła sięgnąć ponownie do tej samej komórki pamięci. Aby to zapewnić, minimalna długość naszej transformaty wynosi 16. I taka też jest użyta w testbenchu *14_iterative_fft/fft_iter_tb.sv*. Uruchamiamy go rozkazem:

```
do ./fft_iter.do
```

W wyniku dostaniemy porównanie wektora testowego z otrzymanymi wynikami oraz błąd średniokwadratowy RMS. Dla zadanych danych wynosi on 1. Otrzymane przebiegi są dość duże i byłyby źle widoczne na wydrukowanym rysunku, dlatego zachęcam do uruchomienia symulacji i ich samodzielnej analizy.

Ostatnim krokiem jest podmienienie wersji przepływowej na iteracyjną. Odpowiedni kod znajdziemy w pliku *14_iterative_fft/fft_top.sv*. Aby go przetestować, w sprzęcie budujemy projekt *fft.qpf* i programujemy płytkę Rysino.

Na koniec porównamy jeszcze zużycie zasobów pomiędzy naszymi dwoma implementacjami FFT. Skorzystamy z raportu o liczbie elementów wykorzystywanych przez poszczególne komponenty. Jak widzimy na **rysunku 6** znajdziemy go w zakładce Fitter/Resource Section/Resource Utilization by Entity. Zebrane wyniki widzimy w **tabeli 1**. Wersja iteracyjna wymaga ponadtrzykrotnie mniej elementów logicznych. Zużycie pamięci jest mniejsze o ponad połowę. Największą oszczędność mamy jednak w mnożarkach, gdzie używamy ich tylko 4, zamiast 20.

Podsumowanie

Przekonaliśmy się, że nawet to samo zadanie można zrealizować na różne sposoby. Przygotowaliśmy dwie wersje FFT. Pierwsza przepływowa pozwala pracować z dużo wyższą częstotliwością, ale zużywa znacznie więcej zasobów niż podejście iteracyjne. Wybór odpowiedniego rozwiązania zależy od wymaganej szybkości oraz rozmiaru dostępnego układu FPGA.

Rafał Kozik
rafkozik@gmail.com

Źródła:

[1] Intel MAX 10 Embedded Memory User Guide
http://intel.ly/306j6K8