SCK – PROJEKT INWIDUALNY nr.1

Wykonawca : Julia Polak

Numer Indeksu: 310 965

1. WEJŚCIA I WYJŚCIA JEDNOSTKI EXE_UNIT

```
module exe_unit (i_argB,
    i_argA,
    i_oper,
    o_result,
    o_VF,
    o_ZF,
    o_PF

);
```

```
input logic [N-1:0] i_argA, i_argB;
input logic [3:0] i_oper;
output logic [M-1:0] o_result;
logic [M-1:0] s_gray, s_crc4, s_u1intou2, s_licz1, s_term, s_priorytet, s_crc3, s_BF1;
output logic o_VF, o_ZF, o_PF;
```

RODZAJ WEJŚCIA/WYJŚCIA	NAZWA	ZAKRES
Wejścia jednostki exe_unit	i_argA I_argB	N bitów
Wyjście jednostki exe_unit	o_result	M bitów
Wejście sterujące jednostki exe_unit	i_oper	4 bity
Wyjścia “wewnętrzne” (służą do implementacji modułów w jednostce exe_unit)	s_gray s_crc4 s_u1intou2 s_licz1 s_term s_priorytet s_crc3 s_BF1	
Wyjścia znaczników	o_VF o_ZF o_PF	

2. PARAMETRY JEDNOSTKI EXE_UNIT

```
parameter M = 32;
parameter N = 32;
```

Parametr M odpowiada liczbie bitów wyjścia jednostki exe_unit, natomiast parametr N odpowiada liczbie bitów wyść jednostki exe_unit.

3. SPOSÓW URUCHOMIANIA JEDNOSTKI EXE_UNIT

Aby uruchomić jednostkę exe_unit należy wpisać w terminal komendę make. Poprzez plik run.ys oraz plik makefile jednostka syntezuje się wraz z zaimplementowanymi w niej modułami. Po syntezie wyświetla się schemat jednostki oraz powstaje plik exe_unit_rtl.sv.

Plik run.ys

```
read_verilog -sv exe_unit_1.sv
read_verilog -sv u2intoGray.sv
read_verilog -sv crc4.sv
read_verilog -sv crc3.sv
read_verilog -sv u1.sv
read_verilog -sv zliczanie.sv
read_verilog -sv termometr.sv
read_verilog -sv priorytet.sv
read_verilog -sv znacznik_PF.sv
read_verilog -sv BF1.sv

synth

abc -g AND,OR,XOR
opt_clean -purge

show exe_unit
```

Plik makefile

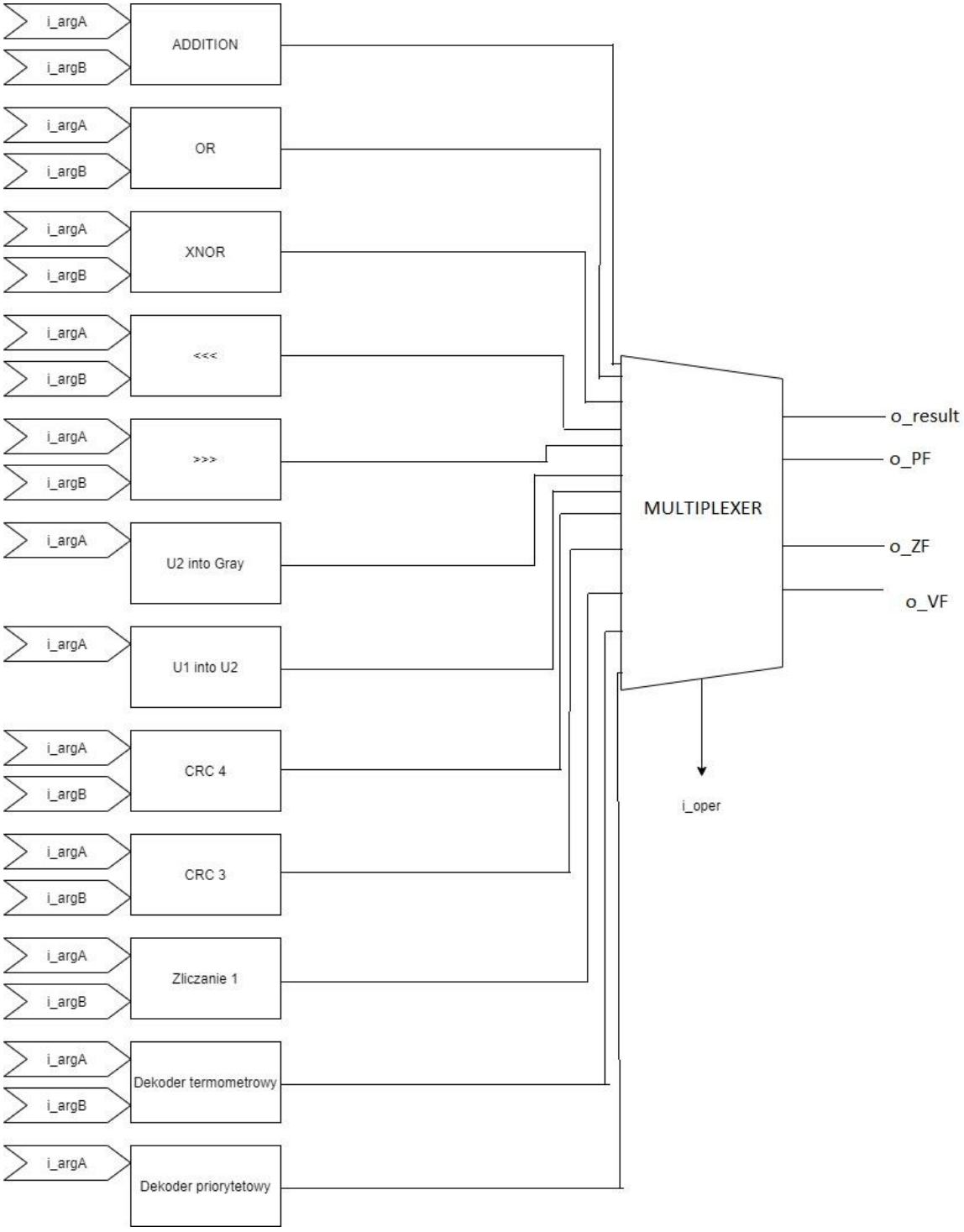
```
SYNTH = yosys

rtl:
	${SYNTH} -s run.ys
```

4. LISTA REALIZOWANYCH FUNKCJI

Nazwa funkcji	Argumenty funkcji	Wejście i_oper
Dodawanie	-Wejścia: i_argA, i_argB -Wyjście: o_result	4’b0000
OR	-Wejścia: i_argA, i_argB -Wyjście: o_result	4’b0001
XNOR	-Wejścia: i_argA, i_argB -Wyjście: o_result	4’b0010
Arytmetyczne przesunięcie argumentów w prawo	-Wejścia: i_argA, i_argB -Wyjście: o_result	4’b0011
Arytmetyczne przesunięcie argumentów w lewo	-Wejścia: i_argA, i_argB -Wyjście: o_result	4’b0100
Konwersja danej wejściowej z kodu U2 na kod GRAY	-Parametr: BITS = 4 -Wejścia: i_argA (4 bity) -Wyjście: o_result (4 bity)	4’b0101
Konwersja danej wejściowej z kodu U1 na kod U2	-Parametr: BITS = 2 -Wejścia: i_argA (2 bity) -Wyjście: o_result (2 bity)	4’b0110
Wyznaczanie kodu CRC4	Parametr WCODE = 5 Parametr WPOLY = 4 -Wejścia: i_data (2 bity), i_poly (2 bity) , i_crc (3 bity) -Wyjście: o_crc (3 bity)	4’b0111
Wyznaczanie kodu CRC3	Parametr WCODE = 4 Parametr WPOLY = 3 -Wejścia: i_data (2 bity), i_poly (2 bity) , i_crc (3 bity) -Wyjście: o_crc (3 bity)	4’b1000
Zliczanie sumarycznej liczby jedynek w obu argumentach wejściowych	-Parametr: BITS = 2 -Parametr: LEN = BITS*2 -Wejścia: i_argA, i_argB (2 bity) -Wyjście: o_result (2 bity)	4’b1001
Dekoder termometrowy	-Parametr LEN = 2 -Wejście: i_argA (2 bity) -Wyjście: i_argB (2 bity)	4’b1010
Dekoder priorytetowy	-Parametr: BITS = 3 -Wejście: i_argA (3 bity) -Wyjście: o_result (2 bity)	4’b1100

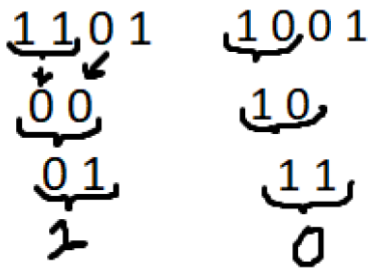
5. SCHEMAT BLOKOWY STRUKTURY JEDNOSTKI



6. FLAGI I ZNACZNIKI

a.) PF – znacznik uzupełnienia do parzystej liczby jedynek

Flaga PF ma za zadanie “uzupełniać” wynik do liczby parzystej jedynek. Czyli także ma za zadanie odczytywać liczbę jedynek w wyniku końcowym na o_result i uzupełniać gdy będzie nieparzysta liczba do parzystej. Flagę tę zrobiłam poprzez ^o_result, a następnie zanegowanie go. Mimo, iż jest to tylko zanegowany XOR to spełnia swą funkcję co udowodniłam na dole. Zrobiłam dwa przypadki dla parzystej liczby jedynek i nieprzystej.



Dla parzystej liczby jedynek zwraca zero natomiast dla nieparzystej 1, więc po zanegowaniu wychodzi odwrotnie i dobrze.

Wejście: o_PF

Implementacja:

```
//znacznik_PF
  begin
   o_PF = ~(^o_result);
   end
```

b.) BF – znacznik informujący, że w wyniku jest tylko jedna jedynka

Znacznik to ma za zadanie prześledzić wynik na o_result i wyświetlić się gdy będzie tylko jedna jedna jedynka. Polega na tym, że przegląda każdą liczbę wyniku pojedyńczo i gdy napotka jedynkę dodaje do licznika 1 i tak po kolei. Gdy stan końcowy licznika jest równy 1 to znacznik zwraca 1, natomiast jest inaczej niż jeden to zwraca zero.

Wejście: o_BF

Implementacja:

```
module wskaznik_BF (i_argA, o_BF1);

    parameter BITS = 2;
    input logic [BITS-1:0] i_argA;
    output logic o_BF1;
    integer i;
    localparam integer s_BF1 = '0;
    always_comb
    begin
        for (i=1; i<BITS; i++)
        begin
            if (i_argA[i] == 1)
                s_BF1=s_BF1+1'b1;

        end
        if(s_BF1 == 1'b1)
            o_BF1 = s_BF1;
        else
            o_BF1 = '0;
    end
endmodule
```

c.) VF – znacznik przepełnienia, informujący, że wynik operacji nie może się pomieścić w żadnej operacji

Flaga VF wyświetla się w sytuacji przepełnienia taką sytuacją jest np. Gdy parametry wejściowe będą miały po 2 bity. W sytuacji dodawania takich bitów wynik może wyjść poza możliwości zapisu na dwóch bitach. Gdy w zapisie bitowym mamy 11+11 to jest to trzy plus trzy w zapisie dziesiętnym co równa się sześć. Takiej liczby nie możemy zapisać na dwóch bitach więc wyświetla się flaga o przepełnieniu.

Wejście: o_VF

Implementacja:

```
4'b0000: {o_VF, o_result} = i_argA + i_argB;
```

d.) ZF- znacznik informujący, że wynikiem operacji jest ‘0

Znacznik ZF ma za zadanie wyświetlić zero gdy na o_result dostaniemy ‘0. Po prostu zrobiłam pętlę if, w której warunkiem było porównanie o_result do ‘0.

Wejście: o_ZF

Implementacja:

```
//znacznik_ZF
    begin
    if (o_result=='0)
    o_ZF=1;
    else o_ZF=0;
    end
```

7. MODUŁY I ICH UŻYCIE

a.) Arytmetyczne przesunięcie w lewo/ prawo

```
4'b0011: o_result = (argA >>> i_argB);
4'b0100: o_result = (argA <<< i_argB);
```

Moduł ten polega, iż przesuwa wszystkie argumenty wejścia i_argA w lewo/ prawo o liczbę miejsc wskazaną przez argument i_argB, który użytkownik sam ustala (przesunięcie w prawo o 1 to podzielenie przez 2, o 2 to przez 4....)

b.) Konwersja danej wejściowej z kodu U1 na kod U2

```
module u1intou2 (i_argA,
                //i_argB,
                o_result);

parameter BITS = 2;
input logic signed [BITS-1:0] i_argA;
//input logic signed [BITS 1:0] i_argB;
output logic signed [BITS-1:0] o_result;
//integer ;
always_comb
    begin
        if(i_argA[BITS-1] == 0)
        begin
            o_result = i_argA;
        end
        else    //(i_argA[BITS == 1])
        begin
            o_result = (i_argA >> 1);
        end
    end
endmodule
```

Moduł ten mam zadanie zmieniać daną wejściową z kodu U1 na U2. Zaimplementowałam to za pomocą pętli if... else... . U1 i U2 dla bitów ujemnych są takie same. Więc najpierw pętla sprawdza czy najstarszy bit jest równy 0 (co oznacza ,że jest dodatni). Jeżeli tak jest to nic się nie zmienia. Następnie porównuje najstarszy bit do jedynki (co oznacza, że jest ujemny). Jeżeli tak jest to aby zmienić U1 na U2 wystarczy arytmetycznie przesunąć wejście o 1 w prawo.

c.) Konwersja danej wejściowej z kodu U2 na kod Gray

```
module u2intoGray (i_argA, o_result);

parameter BITS = 4;
input logic signed[BITS-3:0] i_argA;
//input logic signed [BITS 1:0] i_argB;
output logic signed[BITS-3:0] o_result;
//integer ;

//logic [BITS-1:0] s_argA;

always_comb
    begin

        if(i_argA[BITS-1] == 0)
        begin
            o_result  = i_argA ^ (i_argA >> 1);
        end

    end
endmodule
```

Moduł ten ma za zadanie zmienienie danej wejściowej z kodu U2 na kod Gray. U2 w przeciwieństwie do kodu graya ma wartości ujemne. Więc na początek za pomocą pętli if... sprawdzamy najstarszy bit w celu zweryfikowania czy dana jest ujemna czy dodatnia (1- ujemna, 0- dodatnia). Jeżeli jest dodatnia to następuje XOR bitu(n) oraz bitu arytmetycznie przesuniętego w prawo.

8. LISTA PLIKÓW .SV I DOPASOWANIE DO MODUŁÓW

NAZWA PLIKU	CO TEN MODUŁ ROBI
u1.sv	Konwersja danej wejściowej z kodu U1 na kod U2
u2intoGray.sv	Konwersja danej wejściowej z kodu U2 na kod Gray
crc4.sv	Wyznaczanie kodu CRC-4

crc3.sv	Wyznaczanie kodu CRC-4
zliczanie.sv	Zliczanie sumarycznej liczby jedynek w obu argumentach wejściowych
termometr.sv	Dekoder termometrowy
priorytet.sv	Dekoder priorytetowy
BF1.sv	Znacznik BF

9. IMPLEMENTACJA MODUŁÓW W EXE_UNIT

Pierwsze kilka funkcji typu: dodawanie, XOR itp. zaimplementowałam od razu w jednostce exe_unit, ponieważ szybciej było niż implementowanie później osobnych modułów w exe_unit.

```
case (i_oper)
    4'b0000: {o_VF, o_result} = i_argA + i_argB;
    4'b0001: o_result = i_argA ^ i_argB;
    4'b0010: o_result = ~(argA ^ i_argB);
    4'b0011: o_result = (argA >>> i_argB);
    4'b0100: o_result = (argA <<< i_argB);
```

Dłuższe funkcje zrobiłam w oddzielnych funkcjach, a następnie zaimplementowałam w jednostce exe_unit

```
u2intoGray#(.BITS(N)) u2intoGray(.i_argA(i_argA), .o_result(s_gray));
u1intou2#(.BITS(N)) u1intou2(.i_argA(i_argA), .o_result(s_u1intou2));
crc4_eval#(.WCODE(5), .WPOLY(4)) crc4_eval(.i_data(i_argA), .i_poly(i_argB), .i_crc( 3'b0), .o_crc(s_crc4));
crc3_eval#(.WCODE(5), .WPOLY(4)) crc3_eval(.i_data(i_argA), .i_poly(i_argB), .i_crc( 3'b0), .o_crc(s_crc3));
zliczanie_1#(.BITS(N)) zliczanie(.i_argA(i_argA), .i_argB(i_argB), .o_result(s_licz1));
thermometer2U2#(.LEN(N)) termometr(.i_argA(i_argA), .o_result(s_term));
dekoder_piorytetowy#(.BITS(N)) dekoder_piorytetowy(.i_argA(i_argA), .o_result(s_priorytet));
wskaznik_BF#(.BITS(N)) wskaznik_BF(.i_argA(o_result), .o_BF1(s_BF1));
```

```
  4'b0101: o_result = s_gray;
  4'b0110: o_result = s_u1intou2;
  4'b0111: o_result = s_crc4;
  4'b1000: o_result = s_crc3;
  4'b1001: o_result = s_licz1;
  4'b1010: o_result = s_term;
  4'b1011: o_result = s_priorytet;
  default: o_result  = '0;
```