Politechnika Świętokrzyska w Kielcach Wydział Elektroniki, Automatyki i Informatyki

Projekt: Architektura Systemów Komputerowych 2

Grupa: **2ID13A**	Temat: **6**
Ocena:	Rok akademicki: **2022/2023-S1**

Ocena z projektu

Imię i nazwisko	Numer albumu	1 termin Data: 30.01.2023	2 termin Data: 02.02.2023	3 termin Data:
Hubert Futrzyński	---------			

Spis treści

1 Temat projektu

Wykonać projekt mikroprocesora oraz układów towarzyszących zgodnie z założeniami projektowymi przedstawionymi w zasadach zaliczenia projektu. Ponadto mikroprocesor musi:

- mieć możliwość zaadresowania 4096 słów pamięci operacyjnej,
- wspierać adresowania: domyślne, natychmiastowe, bezpośrednie, bazowe z przesunięciem,
- wspierać segmentację pamięci z podziałem na segment kodu programu i segment danych,
- posiadać odpowiednią liczbę rejestrów segmentowych,
- posiadać rejestr licznika rozkazów (tylko do odczytu),
- posiadać 3 rejestry uniwersalne 16 bitowe,
- obsługiwać stos,
- wykonywać rozkazy:
    - przesyłanie danych rej-nat, rej-rej, rej-pam,
    - dodawanie/odejmowanie 16 bitowe rej-nat, rej-rej,
    - porównywanie rej-rej, rej-nat,
    - wywołanie podprogramu dla adresu podanego jako liczba lub rejestr,
    - wykonywanie skoku bezwarunkowego,
    - wykonywanie skoków warunkowych gdy większe, mniejsze, równe,
    - warunkowe wyliczanie wartości funkcji logicznych dla rej - rej dla warunków: gdy większe, mniejsze, równe.

Podstawową długością słowa mikroprocesora jest 8 bitów. Rejestr znaczników musi być aktualizowany po wykonaniu odpowiednik rozkazów. Słowo rozkazu mikroprocesora MUSI posiadać zmienną długość. Długość słowa na magistrali danych mikroprocesora ma wynosić 8 bitów. W pamięci należy przygotować program, który będzie demonstrował możliwości mikroprocesora (treść pseudokodu wraz z treścią assemblera należy zamieścić w sprawozdaniu)

2 Wstęp teoretyczny omawianego projektu

Projekt obsługuje wszystkie założenia projektowe wszystkie adresowania. Wspiera segmentacje pamięci na segment kodu i danych. Posiada wszystkie potrzebne rejestry segmentowe takie jak SS-Rejestr Stosu, SP-Rejestr przetrzymujący wskaźnik na szczyt stosu, BP-Rejestr Bazowy oraz CS-Segment kodu i DS-Segment danych. Rejestr Licznika programu został nazwany IP od Instruction Pointer. Procesor zawiera 3 Rejestry 16 bitowe które zostały stworzony za pomocą 2 rejestrów 8 bitowych, długość słowa magistrali danych wynosi 8bitów więc takie połączenie rejestrów zostało wymuszone. Procesor obsługuje Stos który znajduję się w bloku segmentu. Wszystkie rozkazy które są wymienione w temacie projektu są realizowane przez stworzony w tym projekcie procesor.

3 Założenia projektowe

1: Wspierać adresowania: domyślne, natychmiastowe, bezpośrednie oraz bazowe z przesunięciem.
2: Wspierać segmentację pamięci z podziałem na segment kodu i danych.
3: Posiadać specjalny rejestr licznika oraz 3 rejestry uniwersalne 16 bitowe.
4: Obsługiwać stos
5: Wykonywać rozkazy przesyłania danych, dodawanie i odejmowanie, wykonywanie skoków warunkowych i bezwarunkowych, wyliczanie wartości funkcji logicznych, wywoływanie podprogramu oraz porównywanie danych.

3.1 Lista rejestrów uniwersalnych/specjalnego przeznaczenia

Tabela 1 Tabela z rejestrami uniwersalnymi

LP.	Numer rejestru	Nazwa rejestru	Opis rejestru
1	00	R0	Rejestr uniwersalny 16 bitowy
2	01	R1	Rejestr uniwersalny 16 bitowy
3	10	R2	Rejestr uniwersalny 16 bitowy

3.2 Lista rejestrów segmentowych

Tabela 2 Lista z rejestrami segmentowymi

LP.	Numer rejestru	Nazwa rejestru	Opis rejestru
1	000	CS	Segment Kodu
2	001	DS	Segment Danych
3	010	SS	Segment Stosu
4	011	SP	Wskaźnik na szczyt Stosu
5	100	BP	Rejestr Bazowy
6	101	IP	Rejestr Licznika Rozkazów

3.3 Lista rozkazów

Tabela 3 Tabela rozkazów

LP.	Rozkaz	Opis rozkazu
1	JMP	skok bezwarunkowy
2	JG	skok warunkowy, gdy >
3	JL	skok warunkowy, gdy <
4	JE	skok warunkowy, gdy =
5	NOP	Nic nie rób
6	RET	Powrót z procedury
7	RESET	Restart programu
8	CALL	Wywołanie podprogramu
9	PUSH	Odłożenie wartości z rejestru na stos
10	POP	Zdjęcie wartości ze stosu i zapisanie w rejestrze
11	TOP	Sczytanie wartości ze szczytu stosu
12	NOT	Negacja Rejestru
13	AND	Koniunkcja
14	OR	Alternatywa
15	XOR	Alternatywa wykluczająca
16	NOR	Negacja sumy logicznej
17	NAND	Negacja koniunkcji
18	CMP	Porównywanie danych
19	ADD16B	Dodawanie 16 bitowe
20	SUB16B	Odejmowanie 16 bitowe
21	MOV	Adresowanie danych
22	LEA	Odczyt licznika rozkazów

3.4 Zakodowana lista rozkazów

Tabela 4 Zakodowana Tabela rozkazów

Kod grupy	Kod rozkazu	Opis rozkazu
Operacje jednoargumentowe		
00	**M - kod podgrupy , CC - kod operacji, RR – kod rejestru**	
	dla M=0	**M CC --- [nat8]**
	0 000 -- [nat8]	jmp
	0 001 -- [nat8]	jg
	0 010 -- [nat8]	jl
	0 011 -- [nat8]	je
	1 100 --	nop
	1 101 --	ret
	1 110 --	reset
	dla M=1	**M CCC RR**
	1 000 RR	jmp
	1 001 RR	call RR
	1 010 RR	push RR
	1 011 RR	pop RR
	1 100 RR	top RR
	1 101 RR	not RR
Operacje dwuargumentowe		
01	**M - kod podgrupy, CCCC - kod operacji, RR - kod rejestru I, RR -kod rejestru II**	
	dla M=0	**M CCCC- ---- RR RR**
	0 0000 - ----RR RR	add16B RR, RR
	0 0001 - - ---RR RR	sub16B RR, RR
	0 0010 - ----RR RR	cmp RR, RR
	0 0011 - ----RR RR	and RR, RR
	0 0100 - ----RR RR	or RR, RR
	0 0101 - ----RR RR	xor RR, RR
	0 0110 - ----RR RR	nor RR, RR
	0 0111 - ----RR RR	nand RR, RR
	dla M=1	**M CC RR - [nat8]**
	1 00 RR - [nat8]	add16B RR, nat8
	1 01 RR - [nat8]	sub16B RR, nat8
	1 10 RR - [nat8]	cmp RR, nat8
Operacje przesyłania danych rej-rej, rej-nat, rej-pam		
10	**M - kod podgrupy, RR - kod rejestru I, RR -kod rejestru II**	
	dla M=0	**M RR RR**
	0 RR RR -	mov RR, RR
11	**M - kod podgrupy, CC – kod operacji, RR - kod rejestru**	
	dla M=0	**M -- RR [nat8] 11 dla: [BP]**
	0 00 RR - [nat8]	mov RR, nat8
	0 01 RR - [nat8]	mov RR, [BP+nat8]
	0 10 RR - [nat8]	mov RR, [nat8]
	dla M=1	**M -- RR**
	1 00 RR -	lea RR, [IP]

4 Opis sprzętowy

Układ wykonawczy zbudowany jest z czterech demultiplekserów, posiada wejście danych 8bitowe oraz wyjście danych 8 bitowe. Układ posiada także wyjście rejestru flag które wychodzi ze zbudowanego komponentu ALU.
Układ ALU czyli Jednostka Arytmetyczno Logiczna zbudowana jest z trzech demultiplekserów i dwóch multiplekserów. Nasza jednostka posiada także sumator 8 bitowy który przyjmuje daną Cin która odpowiada za zmianę operacji z dodawania na odejmowanie i na odwrót. W ALU znajduję się także nasz własny komparator stworzony na potrzeby projektu by wyliczał wartości funkcji logicznych gdy porównywane liczby są mniejsze, większe lub równe, dla każdego warunku mogą być wyliczone odpowiednie funkcje logiczne. W Jednostce Arytmetyczno Logicznej znajduje się także rejestr flag w którego skład wchodzą flagi wychodzące z sumatora oraz komparatora takie jak Cout- Flaga Carry- przeniesienia najstarszego bitu, Oout- Flaga Overflow - przepełnienia, Zout- Flaga Zero - zera, oraz wym – Flaga mniejszości, wyr – Flaga równości oraz wyw – Flaga większości.
Układ Banku Rejestrów zbudowany jest z jednego demultipleksera oraz pięciu multiplekserów posiada 3 rejestry uniwersalne 16 bitowe umożliwiające adresowanie starszej i młodszej części rejestru. Posiada jedno wejście danych oraz 4 wyjścia które umożliwiają dodawanie i porównywanie danych 16 bitowych na magistrali 8 bitowej.
Cały układ posiada pin resetu który zeruje wszystkie rejestry uniwersalne oraz rejestr flag, dzięki czemu procesor zaczyna pracę od nowa.
Segment Stosu zrobiony jest osobno, nie znajduje się w środku jednostki wykonawczej składa się on z pięciu rejestrów segmentowych, jednego multipleksera i jednego demultipleksera oraz bloku ATU (Jednostka tłumaczenia adresów) który tłumaczy adres logiczny na fizyczny który jest potem podawany jako MAR (magistrala adresowa) i MBR (magistrala danych). Jako wejście są podawane dane które mają być zapisane na stos lub też informacja o tym że chcemy je sczytać. Jako wyjście podawane są dane które sczytaliśmy ze stosu.

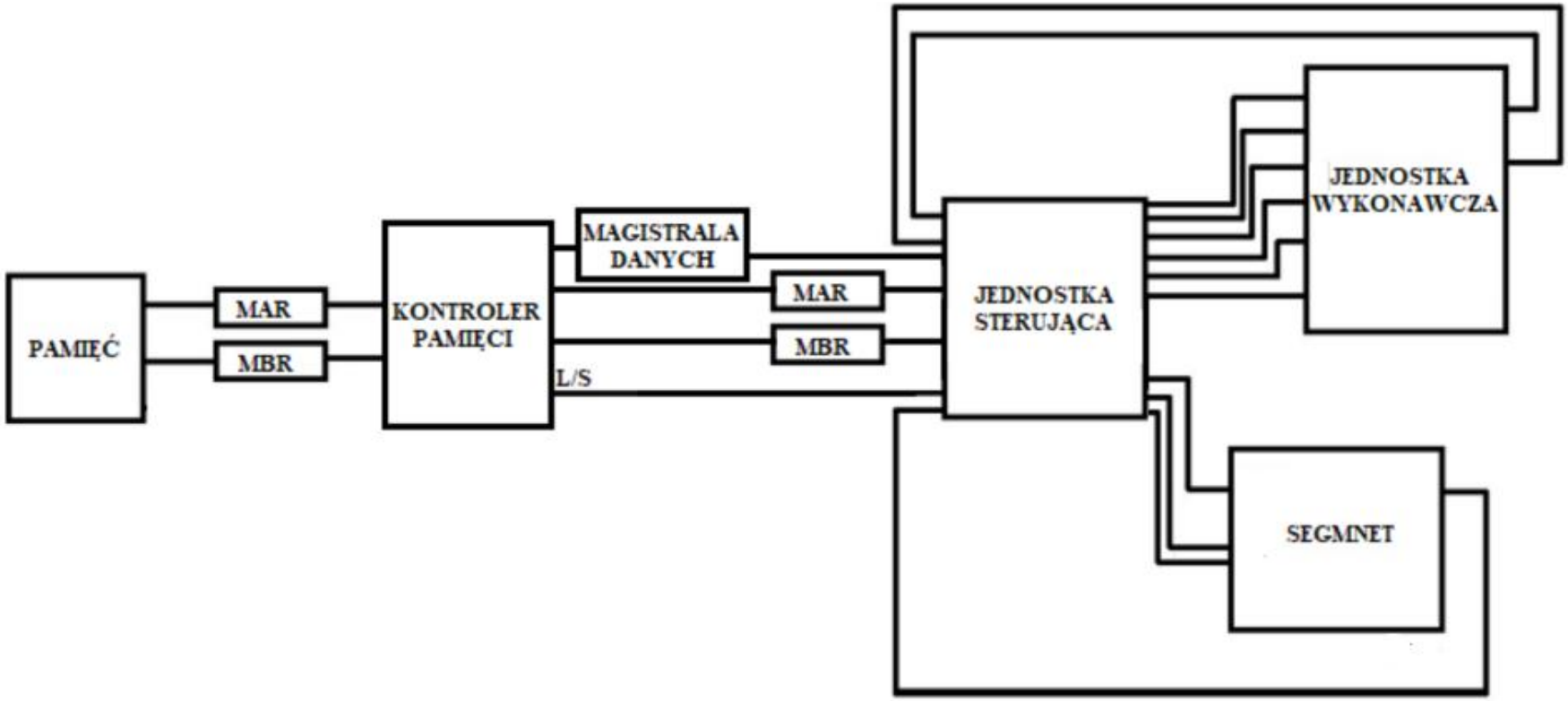
4.1 Jednostka sterująca

Jednostka sterująca przyjmuje jako wejście dane wyjściowe oraz dane rejestru flag które wychodzą z jednostki wykonawczej.
Do naszej jednostki jako dane wejściowe wchodzą także kody instrukcji zawarte w zakodowanej liście rozkazów.
Jako dane wyjściowe jednostka sterująca przekazuje do układu wykonawcze dane 8bitowe oraz przekazuje odpowiednie wartości dla multiplekserów oraz rejestrów aby na naszych danych mogły być wykonywane operacje zgodne z założeniami projektowymi.
Z naszej jednostki do osobnego segmentu stosu wychodzą dane które chcemy zapisać na stosie bądź też informacja że chcemy je sczytać ze stosu. A wracają jako dane wejściowe właśnie sczytane informacje.

5 Schemat mikroprocesora

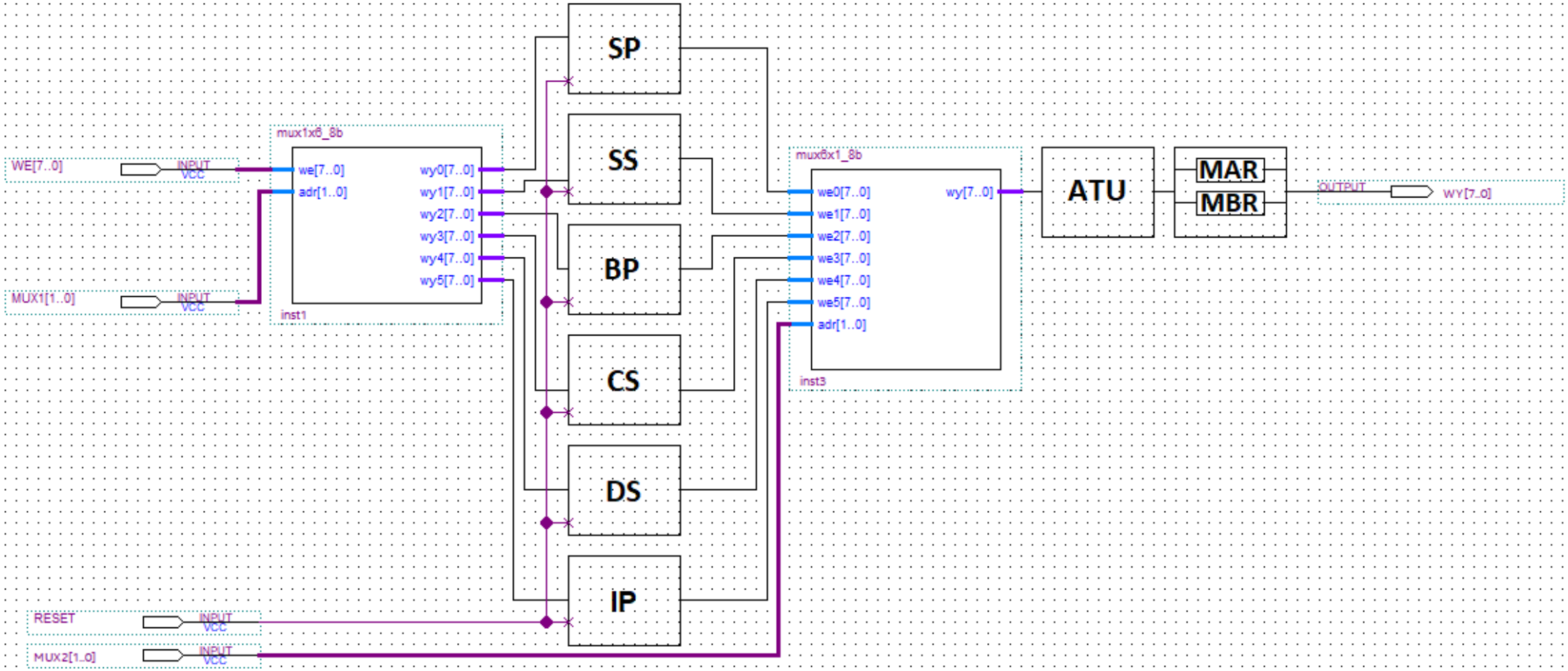
Poniżej przedstawione uproszczone schematy poszczególnych elementów mikroprocesora.

5.1 Ścieżka danych



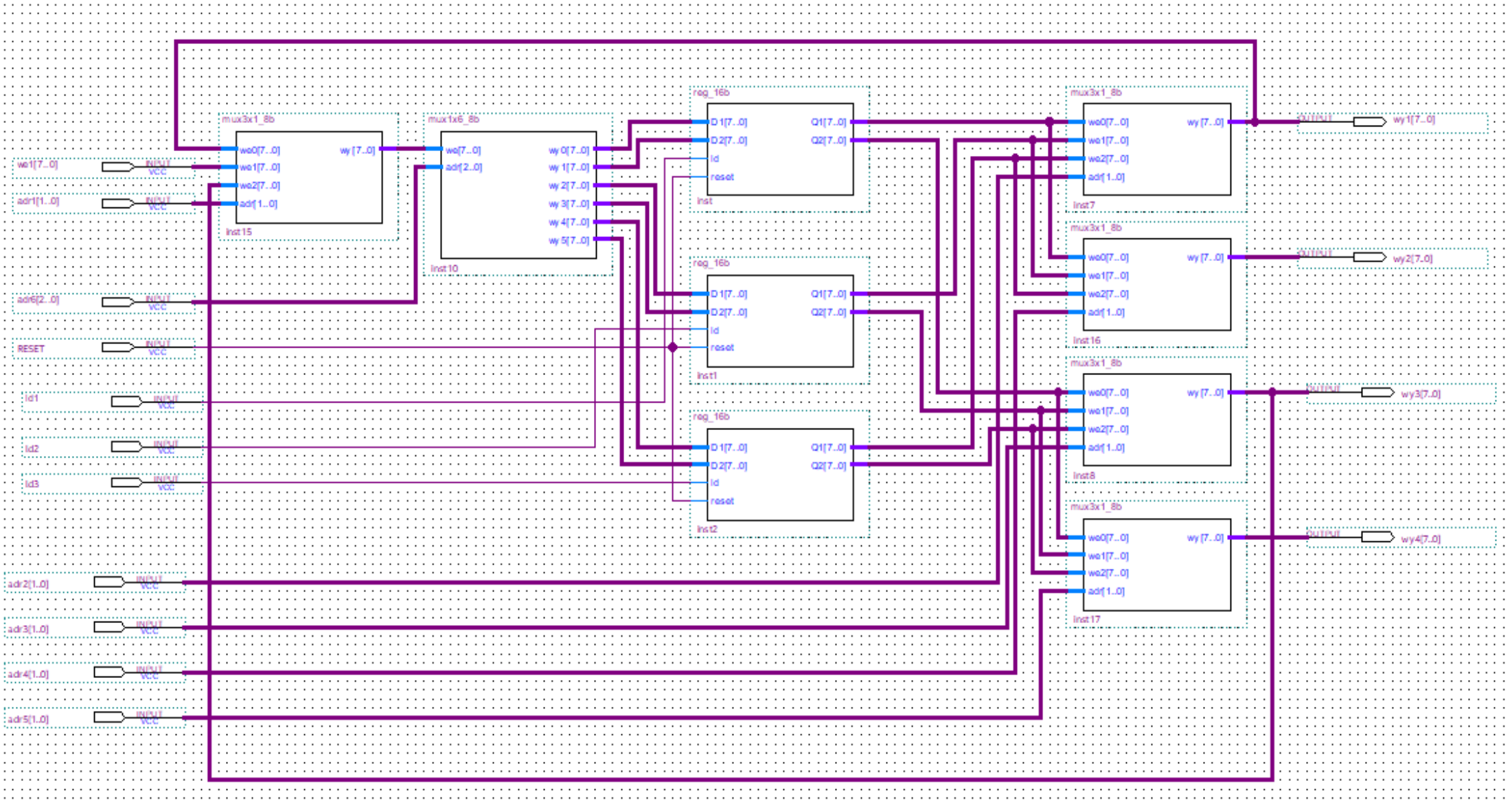
Rysunek 1 Ścieżka danych

5.2 Segment



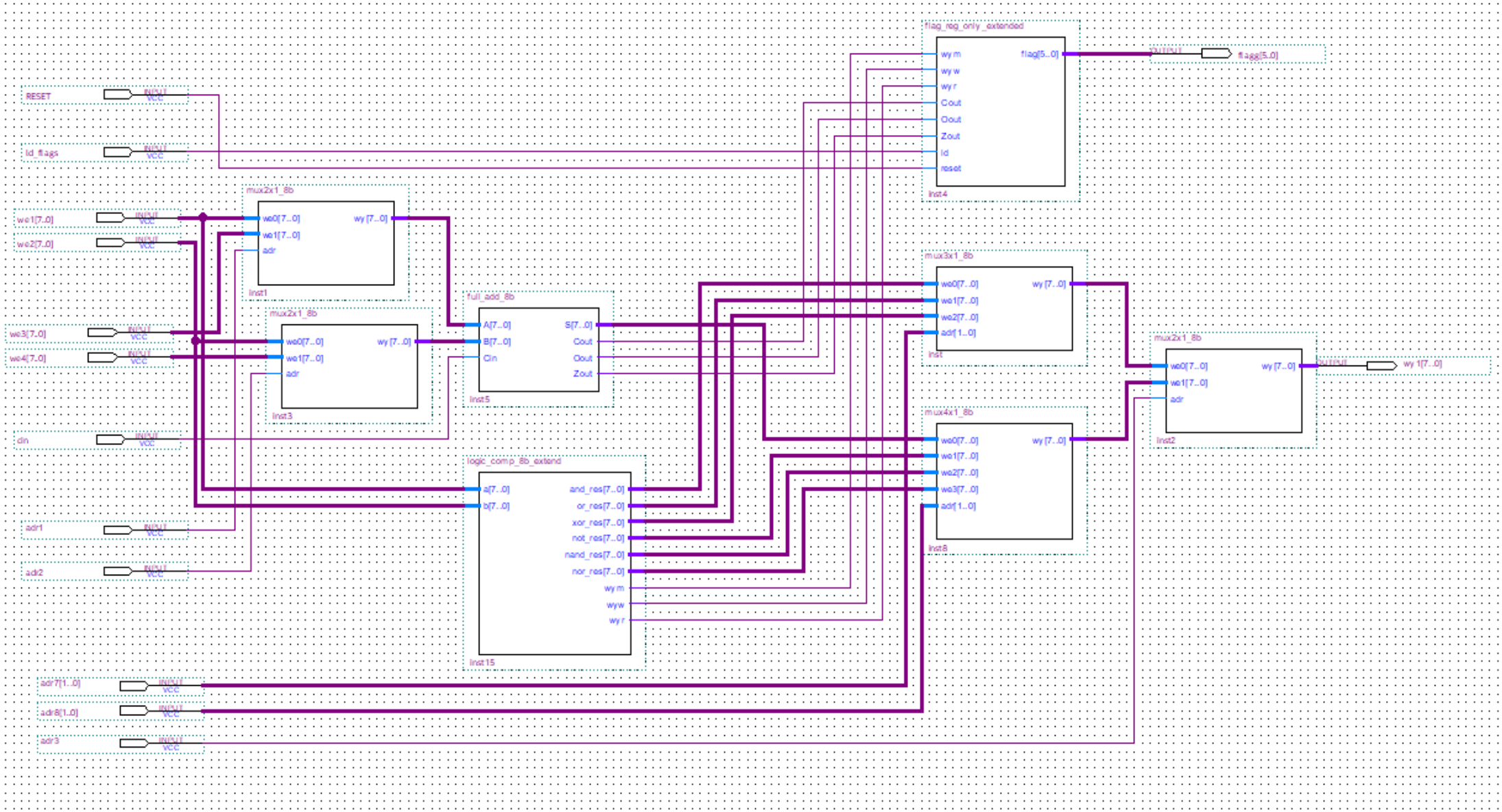
Rysunek 2 Segment

5.3 Bank rejestrów



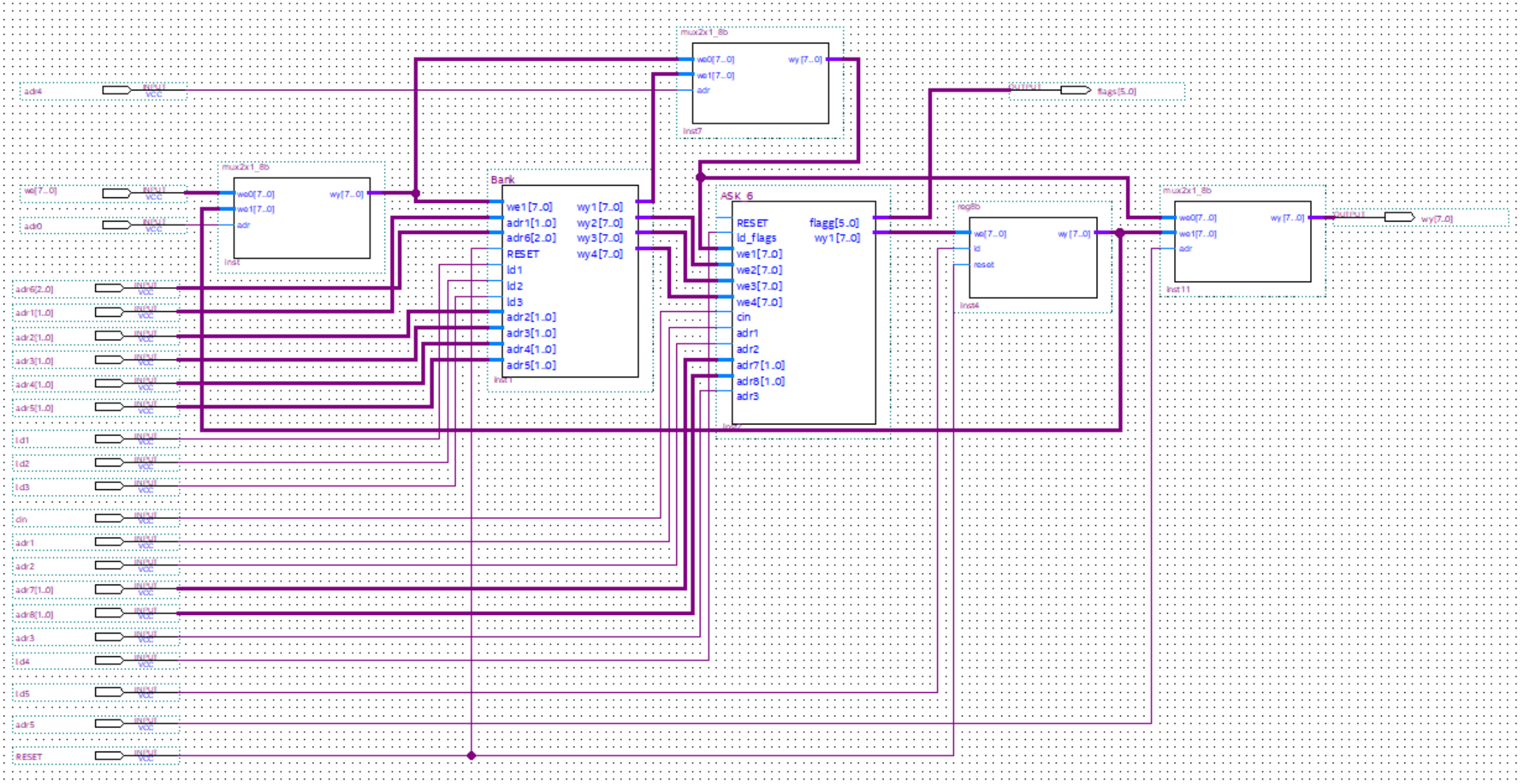
Rysunek 3 Bank rejestrów

5.4 ALU – jednostka arytmetyczno-logiczna



Rysunek 4 ALU

5.5 Układ wykonawczy



Rysunek 5 Układ wykonawczy

6 Mikroinstrukcje mikroprocesora

Lp.	Cykl	Mikroinstrukcja	Opis
PRZESYŁANIE DANYCH/ODCZYT LICZNIKA			
1.	**Raa=WE**		Ładowanie wartości z wejścia układu do rejestrów.
	t0:	ADR0=0, ADR1=01, ADR6=000	Adresowanie natychmiastowe
	t1:	LD1=1	
2.	**Rbb=Raa**		Ładowanie wartości z jednego rejestru do drugiego.
	t0:	ADR2=00, ADR1=00, ADR6=010	Adresowanie domyślne
	t1:	LD2=1	
3.	**Raa=[nat8]**		Ładowanie wartości z pamięci do rejestru.
	t0:	ADR2=00, ADR1=00, ADR6=000	Adresowanie bezpośrednie (nat8 jako adres pamięci)
	t1:	LD1=1	
4.	**Raa=[BP+nat8]**		Ładowanie wartości z rejestru bazowego do rejestru.
	t0:	ADR2=00, ADR1=00, ADR6=000	Adresowanie bazowe z przesunięciem
	t1:	LD1=1	
5.	**Rcc=[IC]**		Ładowanie wartości rejestru licznika rozkazów do rejestru.
	t0:	ADR1=00, ADR6=100	Odczyt licznika rozkazów
	t1:	LD3=1, ADR1=10, ADR4=1, ADR5=0	

DODAWANIE, ODEJMOWANIE, PORÓWNANIE			
6.	**NAT cmp RRbb**		Porównanie wartości natychmiastowej i rejestru i zapisanie w rejestrze flag
	t0:	ADR4=0, ADR4=01, LD_FLAGS=1	
7.	**RRaa cmp Rbb**		Porównanie wartości z rejestru 1 i rejestru 2 i zapisanie w rejestrze flag
	t0:	ADR2=00, ADR4=01, LD_FLAGS=1	
8.	**Raa = Raa AND Rbb**		Koniunkcja
	t0:	ADR2=00, ADR4=01	Zapis do flag warunku mniejsze , równe czy większe
	t1:	LD_FLAGS=1, ADR7=00, ADR3=0	
	t2:	LD5=1, ADR0=1, ADR1=00, ADR6=000	
	t3:	LD1=1	
9.	**Raa = Raa OR Rbb**		Alternatywa
	t0:	ADR2=00, ADR4=01	Zapis do flag warunku mniejsze , równe czy większe
	t1:	LD_FLAGS=1, ADR7=01, ADR3=0	
	t2:	LD5=1, ADR0=1, ADR1=00, ADR6=000	
	t3:	LD1=1	

10.	**Raa = Raa XOR Rbb**		Alternatywa wykluczająca
	t0:	ADR2=00, ADR4=01	Zapis do flag warunku mniejsze , równe czy większe
	t1:	LD_FLAGS=1, ADR7=10, ADR3=0	
	t2:	LD5=1, ADR0=1, ADR1=00, ADR6=000	
	t3:	LD1=1	
11.	**Raa = Raa NOR Rbb**		Negacja sumy logicznej
	t0:	ADR2=00, ADR4=01	Zapis do flag warunku mniejsze , równe czy większe
	t1:	LD_FLAGS=1, ADR8=11, ADR3=0	
	t2:	LD5=1, ADR0=1, ADR1=00, ADR6=000	
	t3:	LD1=1	
12.	**Raa = Raa NAND Rbb**		Negacja koniunkcji
	t0:	ADR2=00, ADR4=01	Zapis do flag warunku mniejsze , równe czy większe
	t1:	LD_FLAGS=1, ADR8=10, ADR3=0	
	t2:	LD5=1, ADR0=1, ADR1=00, ADR6=000	
	t3:	LD1=1	
13.	**Raa = NOT Raa**		Negacja Rejestru
	t0:	ADR2=00	Zapis do flag warunku mniejsze , równe czy większe
	t1:	LD_FLAGS=1, ADR8=00, ADR3=0	
	t2:	LD5=1, ADR0=1, ADR1=00, ADR6=000	
	t3:	LD1=1	
14.	**Raa = Raa + Rbb**		Dodawanie wartości dwóch rejestrów i zapis wyniku do rejestru
	t0:	ADR2=00, ADR3=01, ADR1=0, ADR2=0	Dodanie młodszych bitów
	t1:	CIN=0	
	t2:	LD_FLAGS=1, ADR8=00, ADR3=1	
	t3:	LD5=1, ADR0=1, ADR1=00, ADR6=000	
	t4:	ADR4=00, ADR5=01, ADR1=1, ADR2=1	Dodanie starszych bitów
	t5:	CIN=0	
	t6:	LD_FLAGS=1, ADR8=00, ADR3=1	
	t7:	LD5=1, ADR0=1, ADR1=00, ADR6=001	
15.	**Raa = Raa -Rbb**		Odejmowanie wartości dwóch rejestrów i zapis wyniku do rejestru
	t0:	ADR2=00, ADR3=01, ADR1=0, ADR2=0	Dodanie młodszych bitów
	t1:	CIN=1	
	t2:	LD_FLAGS=1, ADR8=00, ADR3=1	
	t3:	LD5=1, ADR0=1, ADR1=00, ADR6=000	
	t4:	ADR4=00, ADR5=01, ADR1=1, ADR2=1	Dodanie starszych bitów
	t5:	CIN=1	
	t6:	LD_FLAGS=1, ADR8=00, ADR3=1	
	t7:	LD5=1, ADR0=1, ADR1=00, ADR6=001	

16.	**RESET**		Zresetowanie programu
	t0:	RESET=1	
SKOKI: BEZWARUNKOWE I WARUNKOWE			
17.	**JMP, [nat8]**		Skok bezwarunkowy do adresu pamięci podanego jako wartość natychmiastowa
	t0:	ADR0=0	Skok
18.	**JMP, RR**		Skok bezwarunkowy do adresu pamięci podanego jako rejestr
	t0:	ADR0=1	Skok
19.	**JG**		Skok warunkowy (jump if greater) skacze, jeśli flaga większości jest ustawiona.
	t0:	LD_FLAGS=1	Jeśli poprzednia operacja arytmetyczna większa od 0 to może skoczyć
	t1:	nop	
20.	**JL**		Skok warunkowy (jump if less) skacze, jeśli flaga mniejszości jest ustawiona
	t0:	LD_FLAGS=1	Jeśli poprzednia operacja arytmetyczna mniejsza od 0 to może skoczyć
	t1:	nop	
21.	**JE**		Skok warunkowy (jump if equal) skacze, jeśli flaga równości jest ustawiona
	t0:	LD_FLAGS=1	Jeśli poprzednia operacja arytmetyczna jest równa 0 to może skoczyć
	t1:	nop	
OPERACJE NA STOSIE			
22.	**SS=Raa**		Odłożenie wartości z rejestru na stos
	t0:	ADR2=00, ADR4=1, ADR5=0	
	t1:	SS=0	
23.	**Raa=SP**		Zdjęcie wartości ze stosu i zapisanie w rejestrze
	t0:	SP=1 , SADR=01, ADR0=0, ADR1=01, ADR6=000	
	t1:	LD1=1	
24.	**Raa=SP**		Sczytanie wartości ze szczytu stosu
	t0:	SP=1 , SADR=01, ADR0=0, ADR1=01, ADR6=000	
	t1:	LD1=1	
25.	**CALL**		Wywołanie podprogramu dla adresu podanego jako rejestr
	t0:	SS=0	Zapisanie adresu powrotu na stosie
	t1:	nop	
25.	**RET**		Powrót z procedury
	t0:	SS=1	Odczytanie adresu powrotu ze stosu
	t1:	nop	

6.1 Przykład kodu wykonawczego w języku Asembler

```
1: MOV A, 6;      //Wczytaj liczbę natychmiastową do rej. A
2: ADD16B A, 5;   //Dodaj do liczby 5, wynik zapisz w A
3: PUSH A;        //Umieść wartość rejestru A na szczycie stosu
4: MOV B, 8;      //Wczytaj liczbę natychmiastową do rej. B
5: ADD16B B, 2;   //Dodaj do liczby 2, wynik zapisz w B
6: PUSH B;        //Umieść wartość rejestru B na szczycie stosu
7: TOP A;         //Zapisz adres wierzchołka stosu do rejestru A
8: CALL A;        //Skocz do adresu podanego przez rejestr A i zapisz adres powrotu na stos
9: POP A;         //Pobierz wartość z wierzchołka stosu i przypisz do rejestru A
10: POP B;        //Pobierz wartość z wierzchołka stosu i przypisz do rejestru B
11: CMP A, B;     //Porównaj rejestr A z rejestrem B
12: JG 14;        //Jeżeli A > B skocz do instrukcji 14
13: JMP 15;       //Skocz bezwarunkowo do instrukcji 15
14: NAND A, B;    //Wykonaj operację NAND na rejestrach A i B, wynik zapisz w A
15: NOP;          //Instrukcja nop
16: MOV C, 18;    //Wczytaj liczbę natychmiastową do rej. C
17: JMP C;        //Skocz do adresu podanego przez rejestr C
18: CMP A, 10;    //Porównaj rejestr A z 10
19: JE 19;        //Jeżeli rejestr A jest równy 10 skocz do instrukcji 19
20: RET;          //Powrót do adresu zapisanego na stosie
21: RESET;        //Resetuje stos i wszystkie rejestry
```

Listing 1 Przykładowy program w języku Asembler

7 Kod HDL

Kody VHDL tworzone były poprzez analizę kodów zawartych na stronie hector.tu.kielce.pl pod własne potrzeby założeń projektowych.

7.1 Kod multipleksera

Porty wejścia to WE jako dane 8bitowe,
Port adr ustawia port wyjścia na 00, 01 lub inny czyli ten trzeci
Porty wyjścia wy0,wy1,wy2 jako dane 8bitowe
Niżej opisane wszystkie przypadki dla warunków wejściowych

```
library ieee;
use ieee.std_logic_1164.all;

entity mux1x3_8b is
  port(we: in std_logic_vector(7 downto 0);
       adr: in std_logic_vector(1 downto 0);
       wy0, wy1, wy2: out std_logic_vector(7 downto 0));
end mux1x3_8b;

architecture mux1x3_8b_arch of mux1x3_8b is
begin
  process(we, adr)
  begin
    if adr = "00" then
      wy0 <= we;
      wy1 <= "00000000";
      wy2 <= "00000000";
    elsif adr = "01" then
      wy0 <= "00000000";
      wy1 <= we;
      wy2 <= "00000000";
    else
      wy0 <= "00000000";
      wy1 <= "00000000";
      wy2 <= we;
    end if;
  end process;
end mux1x3_8b_arch;
```

Listing 2 Kod układu multipleksera

7.2 Kod rejestru flag

Porty wejścia to dane uzyskane w komparatorze takie jak większe, mniejsze, równe oraz dane uzyskane z sumatora takie jak flaga Carry, Overfloat i flaga Zero
Port ld ustawiony na 1 pozwala na zapis tych flag w rejestrze flag
Port Reset resetuje rejestr flag jak i także pozostałe inne rejestry zawarte w programie.

```
library ieee;
use ieee.std_logic_1164.all;

entity flag_reg_only_extended is
port(wym, wyw, wyr: in std_logic;
Cout, Oout, Zout: in std_logic;
ld: in std_logic;
reset: in std_logic;
flag: out std_logic_vector(5 downto 0));
end flag_reg_only_extended;

architecture flag_reg_only_extended_arch of flag_reg_only_extended is
begin
process(ld,reset)
begin
if reset = '1' then
flag <= (others => '0');
elsif rising_edge(ld) then
flag(0) <= wym;
flag(1) <= wyw;
flag(2) <= wyr;
flag(3) <= Cout;
flag(4) <= Oout;
flag(5) <= Zout;
end if;
end process;
end flag_reg_only_extended_arch;
```

Listing 3 Kod układu rejestru flag

7.3 Kod komparatora

**Porty wejścia to a oraz b czyli dane 8 bitowe przekazane do porównania.
Dane wyjściowe zostają przekazane na wyjście dopiero po ich porównaniu i sprawdzeniu czy równe, mniejsze lub większe. Pod każdym tym warunkiem obliczane są wybrane funkcje logiczne, tak jak zostało to napisane w temacie i założeniach projektu.
Dane wyjściowe wyw–większe, wym–mniejsze i wyr–równe zostają przekazane do rejestru flag.**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity logic_comp_8b_extend is
    port(a, b: in std_logic_vector(7 downto 0);
         and_res, or_res, xor_res, not_res, nand_res, nor_res:
         out std_logic_vector(7 downto 0);
         wym, wyw, wyr: out std_logic);
end logic_comp_8b_extend;

architecture logic_comp_8b_extend_arch of logic_comp_8b_extend is
begin
    process(a, b)
    begin
        wyw <= '0';
        wym <= '0';
        wyr <= '0';
        if a > b then
            and_res <= a and b;
            not_res <= not a;
            nand_res <= not (a and b);
            wyw <= '1';
        elsif a < b then
            or_res <= b or a;
            not_res <= not b;
            nor_res <= not (b or a);
            wym <= '1';
        else
            xor_res <= a xor b;
            not_res <= not a;
            not_res <= not b;
            wyr <= '1';
        end if;
    end process;
end logic_comp_8b_extend_arch;
```

Listing 4 Kod układu komparatora

7.4 Kod rejestru 16bitowego

Porty wejścia A,A2 oraz B,B2 oznaczane jako młodsza i starsza część danej 16 bitowej którą mamy dodawać/odejmować. S i SS to wyniki wyjścia. Cin na 0 oznacza dodawanie a na 1 odejmowanie. Kod opracowany na postawie 8 bitowego sumatora tylko dodane kolejne 8bitów.

```
library ieee;
use ieee.std_logic_1164.all;
entity full_add_16b is
  port(A, A2, B, B2: in std_logic_vector(7 downto 0);
       Cin: in std_logic;
       S: out std_logic_vector(7 downto 0);
       SS: out std_logic_vector(7 downto 0);
       Cout, Oout, Zout: out std_logic);
end full_add_16b;
library ieee;
use ieee.std_logic_1164.all;
entity full_add is
  port(A, B: in std_logic;
       Cin: in std_logic;
       S: out std_logic;
       Cout: out std_logic);
end full_add;
architecture full_add_16b_arch of full_add_16b is
component full_add
  port(A, B: in std_logic;
       Cin: in std_logic;
       S: out std_logic;
       Cout: out std_logic);
end component;
signal Cr, LS, Cr1, LS1: std_logic_vector(7 downto 0);
begin
  -- 8-bitowy sumator dla pierwszych 8 bitów
  s0: full_add port map (A1(0), B1(0), Cin,   LS(0), Cr(0));
  s1: full_add port map (A1(1), B1(1), Cr(0), LS(1), Cr(1));
  s2: full_add port map (A1(2), B1(2), Cr(1), LS(2), Cr(2));
  s3: full_add port map (A1(3), B1(3), Cr(2), LS(3), Cr(3));
  s4: full_add port map (A1(4), B1(4), Cr(3), LS(4), Cr(4));
  s5: full_add port map (A1(5), B1(5), Cr(4), LS(5), Cr(5));
  s6: full_add port map (A1(6), B1(6), Cr(5), LS(6), Cr(6));
  s7: full_add port map (A1(7), B1(7), Cr(6), LS(7), Cr(7));
  -- 8-bitowy sumator dla drugich 8 bitów
  ss0: full_add port map (A2(0), B2(0), Cin,   LS1(0), Cr1(0));
  ss1: full_add port map (A2(1), B2(1), Cr(0), LS1(1), Cr1(1));
  ss2: full_add port map (A2(2), B2(2), Cr(1), LS1(2), Cr1(2));
  ss3: full_add port map (A2(3), B2(3), Cr(2), LS1(3), Cr1(3));
  ss4: full_add port map (A2(4), B2(4), Cr(3), LS1(4), Cr1(4));
  ss5: full_add port map (A2(5), B2(5), Cr(4), LS1(5), Cr1(5));
  ss6: full_add port map (A2(6), B2(6), Cr(5), LS1(6), Cr1(6));
  ss7: full_add port map (A2(7), B2(7), Cr(6), LS1(7), Cr1(7));
  Cout <= Cr(7);
  Oout <= Cr(7) xor Cr(6);
  Zout <= not (LS(0) or LS(1) or LS(2) or LS(3) or LS(4) or LS(5) or LS(6) or LS(7));
  S <= LS(7 downto 0);
  SS <= LS1(7 downto 0);
end full_add_16b_arch;
```

Listing 5 Kod układu rejestru 16bitowego

8 Podział pracy w projekcie

Imię i nazwisko	Numer albumu	Podział procentowy
Hubert Futrzyński	--------	**100%**

9 Spis rysunków

10 Spis listingów

11 Spis tabel