Projekt Mikroprocesora Dokumentacja

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,**
- **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 2 rejestry uniwersalne 16 bitowe z możliwością adresowania części starszej i młodszej rejestru,**
- **obsługiwać stos,**
- **posiadać wydzielony blok ALU,**
- **wykonywać rozkazy:**
  - **przesyłanie danych rej-nat, rej-rej, rej-pam,**
  - **dodawanie/odejmowanie rej-nat, rej-rej, rej-pam,**
  - **dodawanie/odejmowanie 16 bitowe rej-nat, rej-rej,**
  - **porównywanie rej-rej, rej-nat, rej-pam,**
  - **wywołanie podprogramu,**
  - **wykonywanie skoku bezwarunkowego do adresu podanego jako liczba lub rejestr,**
  - **wykonywanie skoków warunkowych gdy większe, mniejsze, równe,**
  - **wyliczanie wartości funkcji logicznych dla rej-rej, rej-pam.**

Podstawową długością słowa mikroprocesora jest 8 bitów. Rejestr znaczników musi być aktualizowany po wykonaniu odpowiednich 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

Mikroprocesor jest układem cyfrowym sekwencyjnym, wykonującym polecenia. CPU jest podstawową jednostką obliczeniową komputera.

2.1 Moduły procesora

Niezależnie od typu i rodzaju większość procesorów posiada modułową konstrukcję, w skład której wchodzą:

2.1.1 Moduły ścieżki danych

- **blok rejestrów ogólnego przeznaczenia**
- **pamięci podręczne pierwszego poziomu**
- **rejestry adresowe**
- **pamięć stronicowania i translacji adresów TLB**
- **układ arytmetyczno logiczny**

2.1.2 Moduły kontrolera

- **sterownik magistral**
- **układy sterujące**
- **układ adresowy**
- **blok pobierania rozkazów**
- **dekoder instrukcji**

2.2 Moduły konstrukcyjne

2.2.1 Magistrale zewnętrzne

Procesor posiada trzy rodzaje magistral:

- **danych - zazwyczaj o długości słowa maszynowego procesora**
- **adresową - determinującą rozmiar pamięci jaką może procesor zaadresować**
- **kontrolną - służącą do: przesyłania poleceń dla procesora, komunikacji z różnymi układami komputera, sterowania zapisem do i odczytem z pamięci oraz urządzeń we/wy**

2.2.2 Rejestry

Rejestry służą do przechowywania wartości, jakie są produktami wykonania instrukcji procesora. Konstrukcyjnie rejestr jest pamięcią statyczną, wykonaną z przerzutników (typu D).

Procesor w rejestrach przechowuje:

- **argumenty i wyniki operacji arytmetycznych i logicznych**
- **wartości stałe (tylko do odczytu)**
- **adresy pamięci**
- **dane aplikacji**
- **dane kontrolne**

Zazwyczaj rejestry łączy się w bloki, posiadające wspólne wejścia i wyjścia oraz wejścia sterujące.

2.2.3 Układ arytmetyczny i rejestr stanu

Układ arytmetyczno-logiczny (ALU) jest modułem wykonującym w procesorze obliczenia. Nowoczesne procesory posiadają od kilku do kilkunastu ALU na jeden rdzeń, dzięki czemu możliwe jest realizowanie zadań optymalizacyjnych.

Zazwyczaj ALU wykonuje operacje:

- **arytmetyczne: dodawanie, odejmowanie, mnożenie…**
- **logiczne: sumę logiczną, iloczyn logiczny, negację…**
- **inkrementację/dekrementację**
- **wyliczanie na żądanie bitów stanu**

W typowych konstrukcjach układ ALU oblicza również pewne dane, charakteryzujące wynik, np. informujące, że wynikiem jest wartość zero. Takie informacje znajdują się w rejestrze stanu.

Informacje z rejestru stanu wpływają na decyzje podejmowane przez procesor odnośnie wykonania instrukcji z różnych rozkazów, w szczególności skoków warunkowych.

2.3 Asembler

Termin informatyczny związany z programowaniem i tworzeniem kodu maszynowego dla procesorów. Oznacza on program tworzący kod maszynowy na podstawie kodu źródłowego wykonanego w niskopoziomowym języku programowania bazującym na podstawowych operacjach procesora zwanym językiem asemblera. Program tworzony w innych językach programowania niż asembler jest zwykle kompilowany do języka asemblera w wyniku pracy kompilatora, a następnie zamieniany na kod binarny przez program asemblera.

3 Założenia projektowe

W ramach tego projektu zostanie zaprojektowany mikroprocesor i układy towarzyszące zgodnie z wymaganiami projektowymi. Mikroprocesor będzie wspierać adresowanie 4096 słów pamięci operacyjnej przy użyciu różnych sposób adresowania, takich jak adresowanie bezpośrednie, domyślne i natychmiastowe. Ponadto mikroprocesor będzie wspierać segmentację pamięci, z podziałem na segment kodu programu i segment danych.

Rejestry mikroprocesora będą obejmować 2 rejestry uniwersalne 16-bitowe oraz odpowiednią liczbę rejestrów segmentowych. Mikroprocesor będzie wspierać obsługę stosu oraz będzie posiadał rejestr licznika rozkazów. Ponadto mikroprocesor będzie wykonywał rozkazy zawarte w temacie projektu.

Blok ALU będzie wydzielony i odpowiedzialny za wykonywanie różnych rozkazów, takich jak dodawanie, odejmowanie, porównywanie, wyliczanie funkcji logicznych.

Podstawową długością słowa mikroprocesora będzie 8 bitów, rejestr znaczników będzie aktualizowany po wykonaniu odpowiednich rozkazów, słowo rozkazu mikroprocesora będzie posiadać zmienną długość, długość słowa na magistrali danych będzie wynosić 8 bitów. Zostanie także przygotowany program demonstrujący możliwości mikroprocesora w postaci pseudokodu wraz z treścią assemblera. Zostaną także przygotowane kody HDL użytych komponentów.

3.1 Lista rejestrów uniwersalnych/specjalnego przeznaczenia

Według założeń projektu, posiadamy 2 rejestry 16 bitowe z możliwością adresacji części młodszej oraz starszej, uzyskaliśmy to przez zastosowanie oddzielnej adresacji młodszych 8 bitów rejestru oraz starszych 8 bitów rejestru. (Odpowiednio LDA_H, LDA_L)

Z powodu takiego podziału, zmuszeni jesteśmy do kopiowania wartości pomiędzy rejestrami 8 bitowymi w 2 cyklach, natomiast pomiędzy 16 bitowy w 4 cyklach (oddzielnie część starsza i młodsza)

Tabela 1: Tabela z rejestrami uniwersalnymi

LP.	Numer rejestru	Nazwa rejestru	Opis rejestru
1	0	A	16 Bitowy rejestr ogólnego przeznaczenia, z możliwością adresacji części młodszej oraz starszej (A_L, A_H)
2	1	B	16 Bitowy rejestr ogólnego przeznaczenia, z możliwością adresacji części młodszej oraz starszej (B_L, B_H)

Tabela 2: Tabela z rejestrami specjalnego przeznaczenia/dodatkowych

LP.	Numer rejestru	Nazwa rejestru	Opis rejestru
1	2	SP	W połączeniu z SS, tworzy efektywny adres ostatniego dodanego słowa do stosu.
2	3	IP	Rejestr służący do obliczenia adresu fizycznego następnego słowa rozkazu w pamięci.
3	4	MBR	8 Bitowy rejestr danych pamięci
4	5	MAR	12 Bitowy rejestr adresowy pamięci
4	6	FLAG	8 Bitowy rejestr przechowujący flagi po przeprowadzonych operacjach
5	7	IR	Rejestr rozkazu (kod rozkazu)
6	8	ALU	8 Bitowy rejestr przechowujący starsze 8 bitów operacji na ALU
7	9	ALU_L	8 Bitowy rejestr przechowujący młodsze 8 bitów operacji na ALU
8	10	STOS	8 Bitowy rejestr przechowujący ostatnią dodaną wartość na stos

3.2 Lista rejestrów segmentowych

Poniższe rejestry służą do wyliczenia adresu fizycznego pamięci. Ze względu na to iż, pamięć mamy podzieloną na segment kodu, programu i danych (w tym stosu) to potrzebujemy adres początkowy każdego zakresu.

Tabela 3: Lista z rejestrami segmentowymi

LP.	Numer rejestru	Nazwa rejestru	Opis rejestru
1	11	CS	Rejestr segmentowy programu (CodeSegment)
2	12	DS	Rejestr segmentowy danych (DataSegment)
3	13	SS	Rejestr segmentowy stosu (StackSegment)

3.3 Lista rozkazów

Spis listy rozkazów według założeń projektu. Wszystkie rozkazy udało się tutaj zamieścić.

Tabela 4: Tabela rozkazów

LP.	Rozkaz	Opis rozkazu
1	RESET	RESTART PROGRAMU
2	NOP	NIE RÓB NIC
3	RET	POWRÓT Z PROCEDURY
4	MOV RR, N	PRZESYŁ DANYCH REJ-NAT
5	MOV RR, RR	PRZESYŁ DANYCH REJ-REJ
6	MOV RR,[N]	PRZESYŁ DANYCH REJ-PAM
7	ADD RR,N	DODAWANIE REJ-NAT
8	ADD RR, RR	DODAWANIE REJ-REJ
9	ADD RR,[N]	DODAWANIE REJ-PAM
10	ADC RR,RR	DODAWANIE Z FLAGA REJ-REJ
11	ADC RR,N	DODAWANIE Z FLAGA REJ-NAT
12	ADC RR, [N]	DODAWANIE Z FLAGA REJ-PAM
13	SUB RR,RR	ODEJMOWANIE REJ-REJ
14	SUB RR,N	ODEJMOWANIE REJ-NAT
15	SUB RR,[N]	ODEJMOWANIE REJ-PAM
16	SBB RR,RR	ODEJMOWANIE Z FLAGA REJ-REJ
17	SBB RR,N	ODEJMOWANIE Z FLAGA REJ-NAT
18	SBB RR, [N]	ODEJMOWANIE Z FLAGA REJ-PAM

19	ADD R,N	DODAWANIE 16 bitowe REJ-NAT
20	ADD R,R	DODAWANIE 16 bitowe REJ-REJ
21	SUB R,N	ODEJMOWANIE 16 bitowe REJ-NAT
22	SUB R,R	ODEJMOWANIE 16 bitowe REJ-REJ
23	CMP RR,RR	PORÓWNANIE REJ-REJ
24	CMP RR,N	PORÓWNANIE REJ-NAT
25	CMP RR,[N]	PORÓWNANIE REJ-PAM
26	JMP RR	SKOK BEZWARUNKOWY DO INSTRUKCJI O WARTOŚCI REJESTR
27	JMP N	SKOK BEZWARUNKOWY DO INSTRUKCJI O WARTOŚCI NAT
28	JG N	SKOK JEŚLI WIĘKSZE
29	JL N	SKOK JEŚLI MNIEJSZE
30	JE N	SKOK JEŚLI RÓWNE
31	CALL N	WYWOŁANIE PODPROGRAMU NAT
32	CALL RR	WYWOŁANIE PODPROGRAMU REJ
33	CALL [N]	WYWOŁANIE PODPROGRAMU PAMIĘĆ
34	AND RR, RR	OPERACJA AND REJ-REJ
35	AND RR, [N]	OPERACJA AND REJ-PAM
36	NAND RR, RR	OPERACJA NAND REJ-REJ
37	NAND RR, [N]	OPERACJA NAND REJ-PAM
38	OR RR, RR	OPERACJA OR REJ-REJ
39	OR RR, [N]	OPERACJA OR REJ-PAM
40	NOR RR,RR	OPERACJA NOR REJ-REJ
41	NOR RR, [N]	OPERACJA NOR REJ-PAM
42	XOR RR,RR	OPERACJA XOR REJ-REJ
43	XOR RR, [N]	OPERACJA XOR REJ-PAM
44	XNOR RR,RR	OPERACJA XNOR REJ-REJ
45	XNOR RR, [N]	OPERACJA XNOR REJ-PAM
46	NOT RR	OPERACJA NOT REJ
47	NOT [N]	OPERACJA NOT PAM
48	PUSH RR	WYSŁANIE DANYCH Z REJESTRU DO STOSU
49	POP RR	POZYSKANIE DANYCH Z STOSU DO REJESTRU

50	TOP RR	ZWRACA WARTOŚĆ Z SZCZYTU STOSU DO REJ
51	TOP_2 RR	MODYFIKUJE WARTOŚĆ SZCZYTU STOSU NA WARTOŚĆ REJ
52	PC_RD RR	ODCZYT LICZNIKA ROZKAZÓW DO REJESTRU

3.4 Zakodowana lista rozkazów

Tabela 5: Tabela zakodowana lista rozkazów

NR	--	KOD GRUPY	KOD OPERACJI	NIE ISTOTNE	ROZKAZ
1		0000	00	- -	RESET
2			01	- -	NOP
3			10	- -	RET

RR - rejestr 8 bitowy

NR	RR,RR	KOD GRUPY	KOD OPERACJI	NIE ISTOTNE	Rejestr I	Rejestr II	FLAGS	ROZKAZ
5		0001	0000	- - -	RR	RR	0	MOV RR, RR
8			0001	- - -	RR	RR	0	ADD RR, RR
13			0010	- - -	RR	RR	0	SUB RR, RR
23			0011	- - -	RR	RR	0	CMP RR, RR
34			0100	- - -	RR	RR	0	AND RR, RR
36			0101	- - -	RR	RR	0	NAND RR, RR
38			0110	- - -	RR	RR	0	OR RR, RR
40			0111	- - -	RR	RR	0	NOR RR, RR
42			1000	- - -	RR	RR	0	XOR RR, RR
44			1001	- - -	RR	RR	0	XNOR RR, RR
10			1100	- - -	RR	RR	1	ADC RR, RR
16			1101	- - -	RR	RR	1	SBB RR, RR

NR	RR,N	KOD GRUPY	KOD OPERACJI	NIE ISTOTNE	Rejestr I	Wart. Natych	FLAGS	ROZKAZ
4		0010	000	- - - - - -	RR	12345678	0	MOV RR, N

7			001	- - - - - -	RR	12345678	0	ADD RR, N
14			010	- - - - - -	RR	12345678	0	SUB RR, N
24			011	- - - - - -	RR	12345678	0	CMP RR, N
12			100	- - - - - -	RR	12345678	1	ADC RR, N
17			101	- - - - - -	RR	12345678	1	SBB RR, N

NR	RR,[N]	KOD GRUPY	KOD OPERACJI	NIE ISTOTNE	Rejestr I	ADR Pamieci	ROZKAZ
6		0011	0000	- - - - - -	RR	12345678	MOV RR, [N]
9			0001	- - - - - -	RR	12345678	ADD RR, [N]
15			0010	- - - - - -	RR	12345678	SUB RR, [N]
25			0011	- - - - - -	RR	12345678	CMP RR, [N]
35			0100	- - - - - -	RR	12345678	AND RR, [N]
37			0101	- - - - - -	RR	12345678	NAND RR, [N]
39			0110	- - - - - -	RR	12345678	OR RR, [N]
41			0111	- - - - - -	RR	12345678	NOR RR, [N]
43			1000	- - - - - -	RR	12345678	XOR RR, [N]
45			1001	- - - - - -	RR	12345678	XNOR RR, [N]
12			1100	- - - - - -	RR	12345678	ADC RR, [N]
18			1101	- - - - - -	RR	12345678	SBB RR, [N]

NR	RR	KOD GRUPY	KOD OPERACJI	NIE ISTOTNE	Rejestr I	ROZKAZ
26		0100	000	- - - - - - -	RR	JMP RR
32			001	- - - - - - -	RR	CALL RR
46			010	- - - - - - -	RR	NOT RR
48			011	- - - - - - -	RR	PUSH RR
49			100	- - - - - - -	RR	POP RR
50			101	- - - - - - -	RR	TOP RR
51			110	- - - - - - -	RR	TOP_2 RR
52			111	- - - - - - -	RR	PC RR

NR	N	KOD GRUPY	KOD OPERACJI	NIE ISTOTNE	Wart. Natych	ROZKAZ
27		0101	000	-	12345678	JMP N
28			001	-	12345678	JG N
29			010	-	12345678	JL N
30			011	-	12345678	JE N
31			100	-	12345678	CALL N

NR	[N]	KOD GRUPY	KOD OPERACJI	NIEISTOTNE	Wart. Natych	ROZKAZ
33		0110	0	- - -	12345678	CALL [N]
47			1	- - -	12345678	NOT [N]

R0 -> starsze 8 bitów 16 bitowego rejestru

R1 -> Młodsze 8 bitów 16 bitowego rejestru

NR	R,R	KOD GRUPY	KOD OPERACJI	NIEISTOTNE	Rejestr 16b_1	Rejestr 16b_2	Rejestr 16b_1	Rejestr 16b_2	ROZKAZ
20		1000	0	- - -	R0	R1	R0	R1	ADD R, R
22			1	- - -	R0	R1	R0	R1	SUB R, R

NR	R,N	KOD GRUPY	KOD OPERACJI	NIEISTOTNE	Rejestr 16b I	Rejestr 16b I	Wart. Natych	Wart. Natych_2	OPIS
19		1001	0	- - - - - - -	R0	R1	00000000	12345678	ADD R, NN
21			1	- - - - - - -	R0	R1	00000000	12345678	SUB R, NN

4 Opis sprzętowy

4.1 Jednostka sterująca

Wejścia (8-bitowe):

- **DATA - dane z pamięci, bezpośrednio transportowane do układu sterującego.**

Wejścia (1-bitowe):

- **CLK - bezpośredni dostęp do zegara procesora.**
- **RESET - Wejście resetujące jednostkę sterującą i innych elementów.**

Wyjścia:

- **35 wyjść sterujących o różnych rozmiarach bitowych. Zostaną one wyjaśnione podczas omawiania konkretnego elementu, którym sterują.**

4.2 Jednostka arytmetyczno-logiczna

Jednostka arytmetyczno-logiczna składa się z 4 wejść oraz 2 wyjść.

Wejścia (8-bitowe):

- **L_A - najmłodsze bity liczby A**
- **H_A - najstarsze bity liczby A**
- **L_B - najmłodsze bity liczby B**
- **H_B - najstarsze bity liczby B**

Wyjścia (8-bitowe):

- **FLAGI - wyjście rejestru flag**
- **WY - wyjście ALU, przekazujące wynik wykonanych działań**

Funkcjonalności:

- **porównywanie wartości 8-bitowych**
- **dodawanie i odejmowanie wartości 8 i 16-bitowych bez flagi**
- **dodawanie i odejmowanie wartości 8-bitowych z flagą**
- **wykonywanie operacji logicznych na wartościach 8-bitowych**

W rejestrze flag przechowywane są informacje o wyniku operacji komparatora oraz informacje czy wystąpiło przeniesienie (carry out) lub nadmiar (overflow).

Operacje 8-bitowe są wykonywane poprzez wybranie 2 z 4 dostępnych wejść.

Operacje 16-bitowe takie jak odejmowanie i dodawanie, są wykonane jako operacje 8-bitowe, na przykład dodawanie odbywa się poprzez wykonania dodawania między wartościami najmłodszych bitów liczby A i B, następnie zostaje wykonane dodawanie najstarszych bitów liczby A i B a do otrzymanego wyniku zostaje również dodane przeniesienie (carry out) z dodawania najmłodszych bitów jeśli takowe wystąpiło, wynik najmłodszych zostaje zapisany w osobnych rejestrze LD_ALU_L, natomiast część starsza wyniku zostaje zapisana w rejestrze LD_ALU, analogicznie wykonywane jest odejmowanie.

4.3 Bank rejestrów

Bank rejestrów posiada 1 wejście i 4 wyjścia.

Wejście (8-bitowe):

- **WART - najmłodsze lub najstarsze bity liczby**

Wyjścia (8-bitowe):

- **WYREJ0 - najstarsze bity liczby A**
- **WYREJ1 - najmłodsze bity liczby A**
- **WYREJ0_2 - najstarsze bity liczby B**
- **WYREJ1_2 - najmłodsze bity liczby B**

Funkcjonalności:

- **zapisywanie liczb w rejestrach 16-bitowych**
- **wczytywanie liczb z rejestrów 16-bitowych**

Operacje zapisu 8-bitowych liczb w rejestrach 16-bitowych wymagają dwukrotnego pobrania z wejścia bitów. Pierwsza część najmłodszych lub najstarszych bitów jest

ustawiana na wybranym rejestrze, a potem zostają one przesłane dalej pętlą zwrotną do multipleksera i demultipleksera. Dalej zostaje pobrana druga część najmłodszych lub najstarszych bitów, które zostają razem z pierwszą częścią (czekającą w demultiplekserze) zostają zapisane jednocześnie do rejestru.

Operacja wczytania wymaga jedynie wybrania odpowiedniej części najmłodszych lub najstarszych bitów z wybranego rejestru za pomocą 1 z 4 multiplekserów, które przesyłają bity na wyjście.

4.4 SEG_manager

Moduł zarządzania segmentami dostępnymi w naszej architekturze, składający się z 4 wejść i 3 wyjść. Na wejściu posiada 1 ścieżkę adresującą, umożliwiającą ustawienie początkowe konkretnego segmentu oraz 3 ścieżki jednobitowe oferujące zapis dostarczanych bitów do konkretnego segmentu.

Wejście (8-bitowe):
- **ADR - ścieżka adresująca**

Wejścia (1-bitowe):
- **SC_LD - ścieżka jednobitowa umożliwiająca ustawienie segmentu kodu**
- **SD_LD - Ścieżka jednobitowa umożliwiająca ustawienie segmentu danych**
- **SS_LD - Ścieżka jednobitowa umożliwiająca ustawienie segmentu stosu**

Wyjścia (8-bitowe):
- SEG_SC - Ścieżka 8 bitowa zawierająca adres początkowy segmentu kodu
- SEG_SD - Ścieżka 8 bitowa zawierająca adres początkowy segmentu danych
- SEG_SS - Ścieżka 8 bitowa zawierająca adres początkowy segmentu stosu

4.5 Converter

Moduł konwertujący dane wejściowe oraz dany segment w fizyczny adres pamięci. Ze względu na to iż korzystaliśmy z dostępnych elementów 16 bitowych, ostatnie 4 bity są zawsze zerowane aby uzyskać 12 bitów adresujących, czyli 4096 słów.

Wejścia (8-bitowe):
- **SEG - Adres bazowy konkretnego segmentu**
- **DATA - Offset adresu od adresu bazowego**

Wyjście (15 bitowe):
- **SP_ADR - Wyliczony fizyczny adres 15 bitowy (ucięte najstarsze 4 bity)**

4.6 SP_MANAGER

Moduł zarządzania STACK_POINTER'em czy wskaźnikiem na stos. Umożliwia on odczyt obecnej wartości wskaźnika, zwiększenie o 1, zmniejszenie oraz posiada wbudowany Converter który został już omówiony wcześniej.

Wejście (15-bitowe):
- **ADR - Wejście to umożliwia ustawienie początkowe wskaźnika.**

Wejście (8- bitowe):
- **SEG_SS - Ścieżka zawierająca adres początkowy segmentu stosu**

Wejście (2 -bitowe):
- **SP_MUX - Wejście to, umożliwia wybór pomiędzy zwróceniem obecnego wskaźnika, inkrementacją go oraz dekrementacją.**

Wejście (1- bitowe):
- **LDSP - Ścieżka umożliwiająca zapisanie nowego wskaźnika, także po inkrementacji/dekrementacji.**

Wyjście(15-bitowe):
- **SP_ADR - Adres fizyczny elementu na szczycie stosu.**

4.7 MEMORY

Moduł zarządzania pamięcią, w tym też stosem. Umożliwia on odczytanie wartości z pamięci oraz zapisanie do niej, posiada osobny rejestr dla wartości z pamięci oraz stosu.

WEJŚCIA(15-bitowe):
- **ADR - Wejście zawierające fizyczny adres pamięci do którego się odwołujemy.**

Wejścia(8- bitowe):
- **Data - Dane które są zapisywane w pamięci**

Wejścia (1-bitowe):
- **MEM_LD - Dla wartości równej 0, korzystamy z odczytu natomiast dla wartości 1 zapisujemy obecnie przyłączone dane.**
- **LD_OUTPUT - Zapis do rejestru danych**
- **LD_STOS - Zapis do rejestru stosu**

4.8 LICZNIK_ROZKAZOW

Moduł przechowywujący rejestr z obecnym adresem fizycznym rozkazu, umożliwia on wykonywanie podstawowych skoków, powrót z procedury, wywołanie procedury/podprogramu.

Wejścia(15-bitowe):
- **NEW_ADR - Nowy adres kolejnych rozkazów, wykorzystywany podczas skoków lub wywołań procedur.**

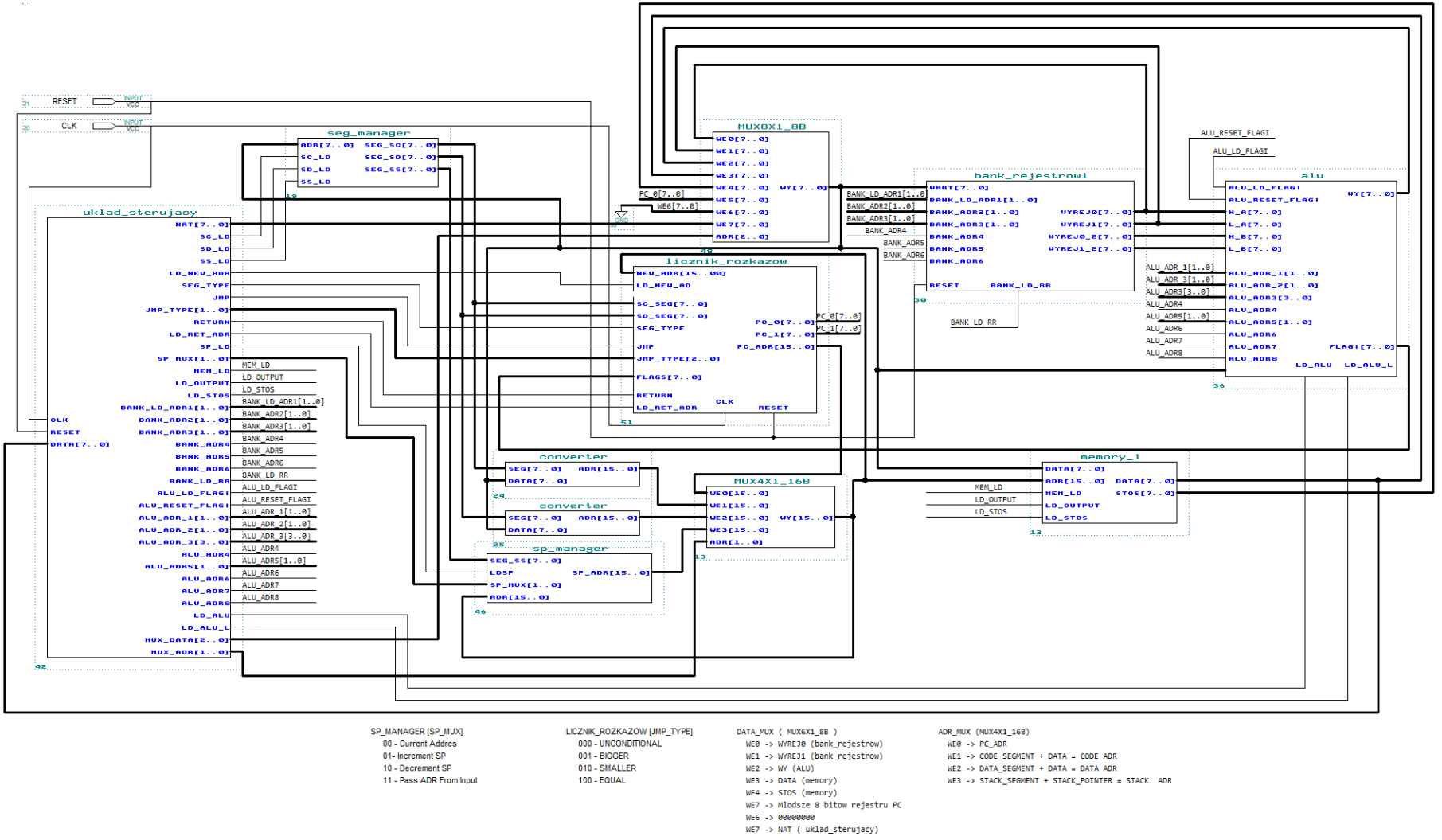
Wejścia(8- bitowe):
- **SC_SEG - Adres bazowy segmentu kodu, potrzebny do obliczania adresu fizycznego dla pamięci.**
- **FLAGS - Dostep do flag dla skoków warunkowych.**

Wejścia(3-bitowe):
- **JMP_TYPE - Typ obecnie przeprowadzanego skoku warunkowego.**

Wejścia(1-bitowe);
- **LD_NEW_AD - Zapisanie nowego adresu rozkazów**
- **JMP - Wartość określająca chęć wykonania skoku, w tym bezwarunkowego.**
- **RETURN - Wartość umożliwiająca powrót z procedury/podprogramu**
- **LD_RET_ADR - Zapisanie wartości rejestru prze wykonaniem skosku, aby umożliwić powrót.**

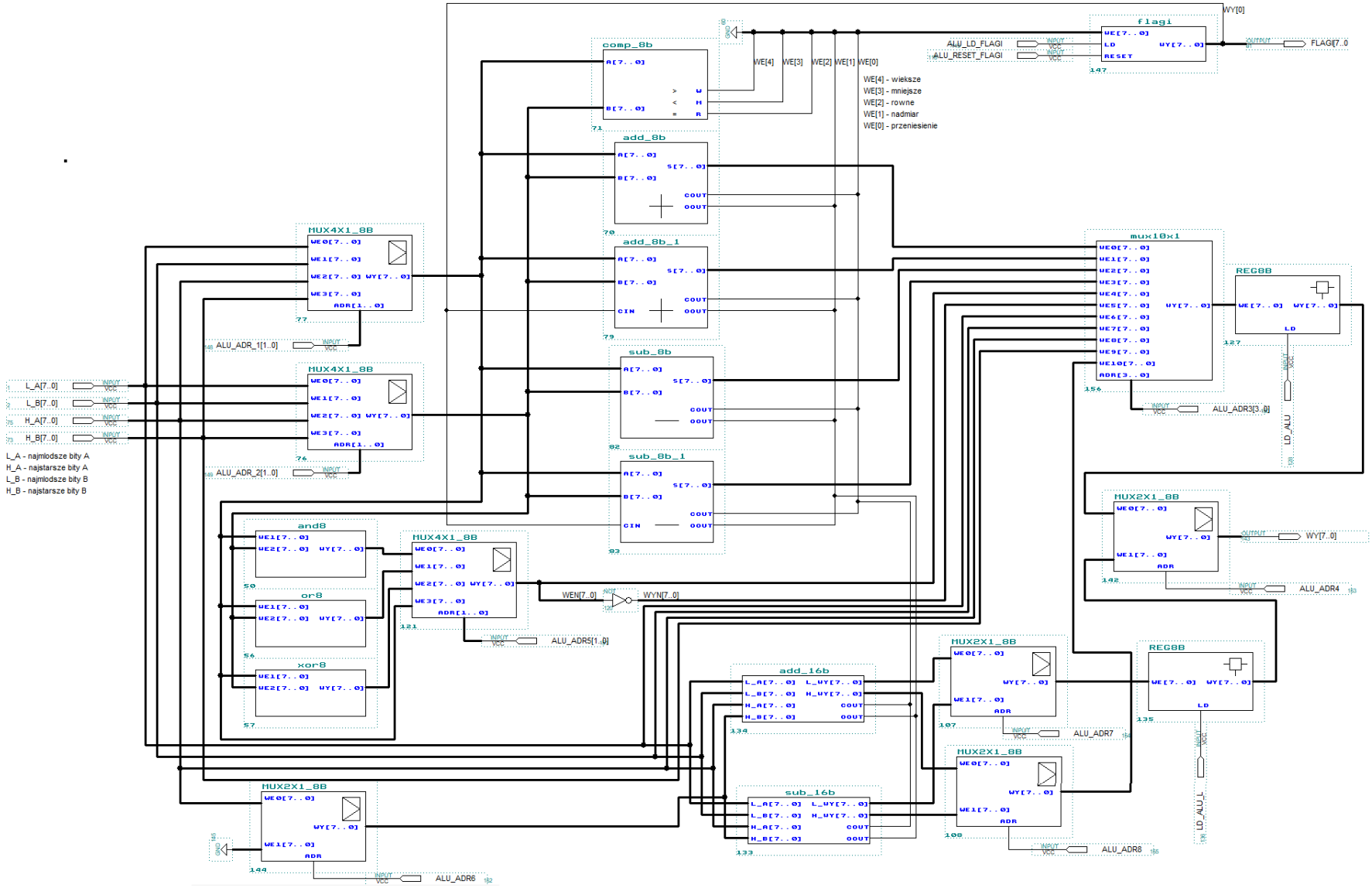
5 Schemat mikroprocesora

5.1 Schemat mikroprocesora



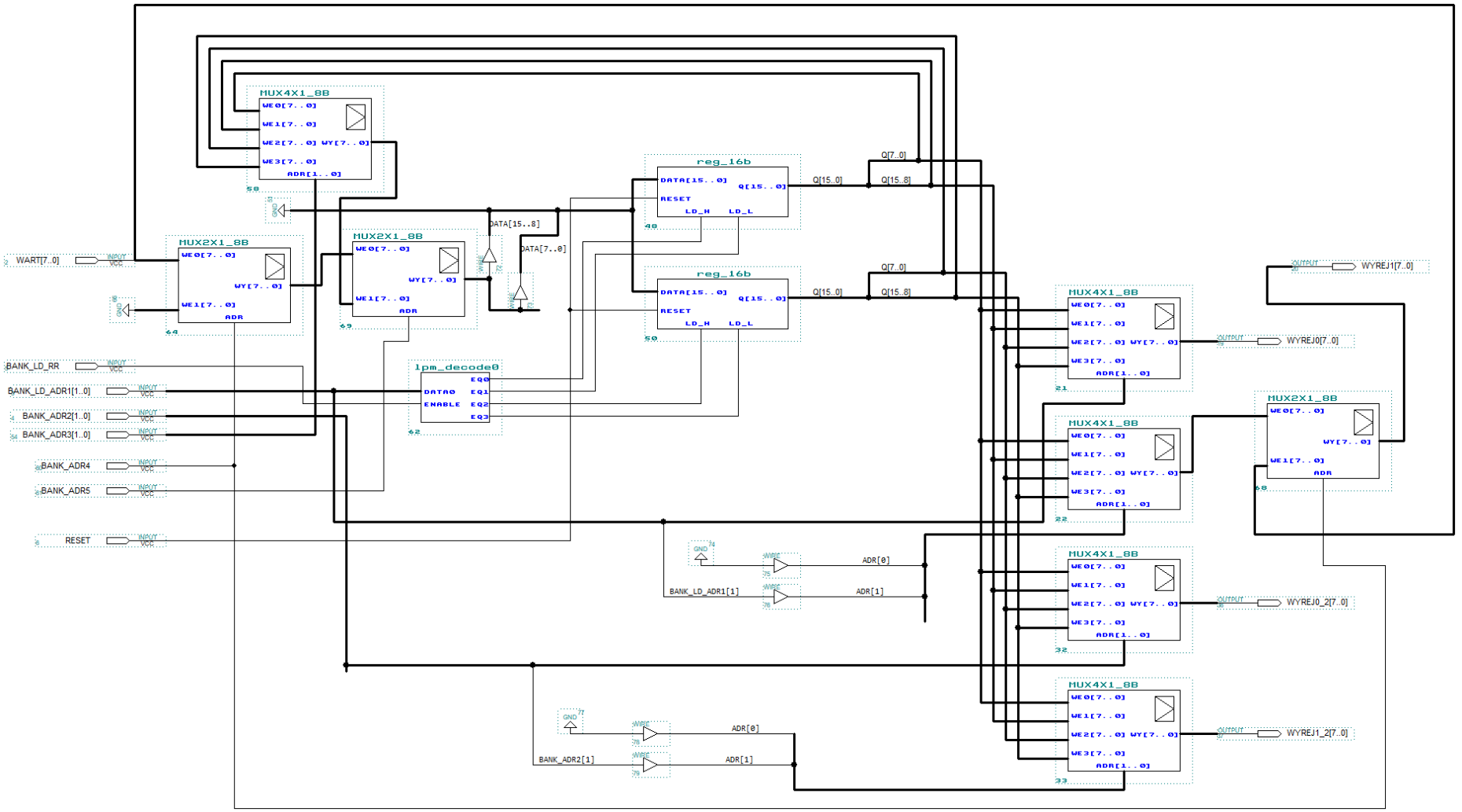
Rysunek 1: Schemat mikroprocesora

5.2 Schemat ALU



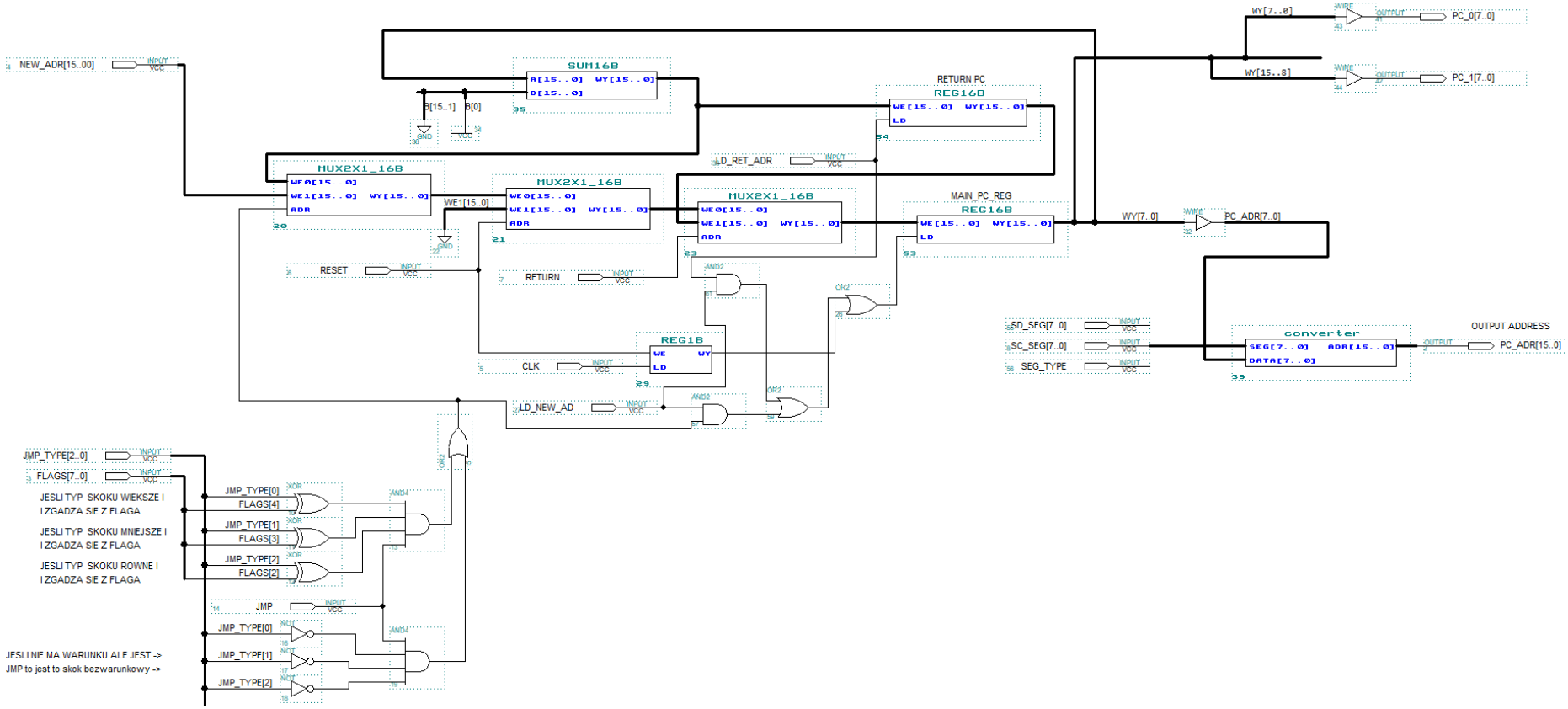
Rysunek 2: Schemat ALU

5.3 Schemat banku rejestrów



Rysunek 3: Schemat banku rejestrów

5.4 Schemat licznika rozkazów



Rysunek 4: Schemat licznika rozkazów

6 Mikroinstrukcje mikroprocesora

LP.	ROZKAZ	Cykle
1	RESET	t0: RESET = 1 t1: NOP t2: NOP t3: NOP
2	NOP	t0: NOP

3	RET	t0: RETURN = 1
		t1: LD_NEW_AD = 1;
		t2: LD_RET_ADR = 1;
		t3: NOP

4	MOV RR,N	t0: DATA_MUX = 7; BANK_ADR4 = 0; BANK_ADR5 = 0;
		t1: BANK_LD_ADR1 = RR;
		t2: BANK_LD_RR = 1;
		t3: NOP

5	MOV RR,RR	t0: BANK_ADR5 = 1; BANK_ADR3 = RR2; BANK_LD_ADR1 = RR1;
		t1: BANK_LD_RR = 1;
		t2: NOP
		t3: NOP

6	MOV RR, [N]	t0: ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3;
		t2: BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR;

7	ADD RR,N	t0: MUX_DATA = 7, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 3, BANK_LD_ADR1 = 0, BANK_ADR2 = 2 t1: ALU_ADR_1 = 2, ALU_ADR_2 = 3, ALU_ADR_3 = 0, LD_ALU = 1, ALU_LD_FLAGI = 1 t2: ALU_ADR4 = 0, MUX_DATA = 2, BANK_LD_ADR1 = 3, BANK_LD_RR = 1 t3: nop
8	ADD RR, RR	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_2 = 1, ALU_ADR_3 = 0 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1 t4: nop

9	ADD RR,[N]	t0: ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1
		ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR3 = 0;
		t2: LD_ALU = 1; ALU_ADR4 = 0; DATA_MUX = 2; BANK_ADR4 = 0;
		BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR;

10	ADC RR,RR	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_2 = 1, ALU_RESET_FLAGI = 1, ALU_ADR_3 = 1 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1 t4: nop

11	ADC RR,N	t0: ALU_RESET_FLAGI = 1; MUX_DATA = 7, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 3, BANK_LD_ADR1 = 0, BANK_ADR2 = 2 t1: ALU_ADR_1 = 2, ALU_ADR_2 = 3, ALU_ADR_3 = 1, LD_ALU = 1, ALU_LD_FLAGI = 1 t2: ALU_ADR4 = 0, MUX_DATA = 2, BANK_LD_ADR1 = 3, BANK_LD_RR = 1 t3: nop

12	ADC RR, [N]	t0: ALU_RESET_FLAGI = 1; ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1
		ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR3 = 1;
		t2: LD_ALU = 1; ALU_LD_FLAGI = 1 ;ALU_ADR4 = 0; DATA_MUX = 2; BANK_ADR4 = 0; BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR;

13	SUB RR,RR	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_2 = 1, ALU_ADR_3 = 2 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1 t4: nop
14	SUB RR,N	t0: MUX_DATA = 7, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 3, BANK_LD_ADR1 = 0, BANK_ADR2 = 2 t1: ALU_ADR_1 = 2, ALU_ADR_2 = 3, ALU_ADR_3 = 2, LD_ALU = 1, ALU_LD_FLAGI = 1 t2: ALU_ADR4 = 0, MUX_DATA = 2, BANK_LD_ADR1 = 3, BANK_LD_RR = 1 t3: nop

15	SUB RR,[N]	t0: ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1
		ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR3 = 2;
		t2: LD_ALU = 1; ALU_ADR4 = 0; DATA_MUX = 2; BANK_ADR4 = 0; BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR;

16	SBB RR,RR	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, ALU_RESET_FLAGI = 1, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_2 = 1, ALU_ADR_3 = 3 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1 t4: nop
17	SBB RR,N	t0: ALU_RESET_FLAGI = 1; MUX_DATA = 7, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 3, BANK_LD_ADR1 = 0, BANK_ADR2 = 2 t1: ALU_ADR_1 = 2, ALU_ADR_2 = 3, ALU_ADR_3 = 3, LD_ALU = 1, ALU_LD_FLAGI = 1 t2: ALU_ADR4 = 0, MUX_DATA = 2, BANK_LD_ADR1 = 3, BANK_LD_RR = 1 t3: nop

18	SBB RR, [N]	t0: ALU_RESET_FLAGI = 1; ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1
		ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR3 = 3;
		t2: LD_ALU = 1; ALU_LD_FLAGI = 1 ; ALU_ADR4 = 0; DATA_MUX = 2; BANK_ADR4 = 0; BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR;

19	ADD R,N	t0: DATA_MUX = 7; BANK_ADR2 = R0; BANK_ADR4 = 1; ALU_ADR6 = 1;
		t1: LD_ALU = 1; LD_ALU_L;ALU_ADR3 = 10; ALU_ADR4 = 0;
		DATA_ MUX = 2; BANK_ADR4 = 0; BANK_ADR5 = 0; BANK_LD_ADR = R0;

		t2: BANK_LD_RR = 1; BANK_LD_RR = 0; BANK_LD_ADR = R1;
		ALU_ADR4 = 1;
		t3: BANK_LD_RR = 1;

20	ADD R,R	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, , ALU_ADR_6 = 0, ALU_ADR_7 = 0, ALU_ADR_8 = 0 t1: LD_ALU_L = 1, ALU_ADR_3 = 10, LD_ALU = 1, ALU_ADR_4 = 0, ALU_LD_FLAGI = 1, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 0 t2: BANK_LD_RR = 1, ALU_ADR_4 = 1, BANK_LD_RR = 0, BANK_LD_ADR1 = 1 t3: BANK_LD_RR = 1

21	SUB R,N	t0: DATA_MUX = 7; BANK_ADR2 = R0; BANK_ADR4 = 1; ALU_ADR6 = 1; ALU_ADR_7 = 1, ALU_ADR_8 = 1;
		t1: LD_ALU = 1; LD_ALU_L;ALU_ADR3 = 10; ALU_ADR4 = 0;
		DATA_ MUX = 2; BANK_ADR4 = 0; BANK_ADR5 = 0; BANK_LD_ADR = R0;
		t2: BANK_LD_RR = 1; BANK_LD_RR = 0; BANK_LD_ADR = R1;
		ALU_ADR4 = 1;
		t3: BANK_LD_RR = 1;

22	SUB R,R	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, , ALU_ADR_6 = 0, ALU_ADR_7 = 1, ALU_ADR_8 = 1 t1: LD_ALU_L = 1, ALU_ADR_3 = 10, LD_ALU = 1, ALU_ADR_4 = 0, ALU_LD_FLAGI = 1, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 0 t2: BANK_LD_RR = 1, ALU_ADR_4 = 1, BANK_LD_RR = 0, BANK_LD_ADR1 = 1 t3: BANK_LD_RR = 1

23	CMP RR,RR	t0: ALU_RESTART_FLAGI = 1; BANK_LD_ADR1 = RR1; BANK_ADR2 = RR2; ALU_ADR_1 = 2; ALU_ADR_2 = 3;
		t1: ALU_LD_FLAGI = 1;
		t2: NOP
		t3: NOP

24	CMP RR,N	t0:ALU_RESTART_FLAGI = 1; BANK_ADR2 = RR; BANK_LD_ADR1 = BANK_ADR4 = 1; ALU_ADR_1 = 3; ALU_ADR_2 = 0;
		t1: ALU_LD_FLAGI = 1;
		t2: NOP;
		t3: NOP;

25	CMP RR, [N]	t0: ALU_RESET_FLAGI = 1; ADR_MUX = 2; DATA_MUX = 7; t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1 ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR3 = 3; t2: ALU_LD_FLAGI = 1 ; t3: NOP

26	JMP RR	t0: BANK_LD_ADR1 = RR; DATA_MUX = 0; ADR_MUX = 1;JMP = 1;
		t1: LD_RET_ADR = 1
		t2: LD_NEW_AD = 1;
		t3: NOP

27	JMP N	t0: DATA_MUX = 7; ADR_MUX = 1;JMP = 1;
		t1: LD_RET_ADR = 1
		t2: LD_NEW_AD = 1;
		t3: NOP

28	JG N	t0: DATA_MUX = 7; ADR_MUX = 1;JMP = 1; JMP_TYPE =1; t1: LD_RET_ADR = 1

		t2: LD_NEW_AD = 1; t3: NOP

29	JL N	t0: DATA_MUX = 7; ADR_MUX = 1;JMP = 1; JMP_TYPE =2;
		t1: LD_RET_ADR = 1
		t2: LD_NEW_AD = 1;
		t3: NOP

30	JE N	t0: DATA_MUX = 7; ADR_MUX = 1;JMP = 1; JMP_TYPE =3;
		t1: LD_RET_ADR = 1
		t2: LD_NEW_AD = 1;
		t3: NOP

31	CALL N	t0: DATA_MUX = 7; ADR_MUX = 1;
		t1: LD_RET_ADR = 1;
		t2: LD_NEW_AD = 1;
		t3: NOP;

32	CALL RR	t0: DATA_MUX = 0; BANK_LD_ADR1 = RR;
		t1: ADR_MUX = 1;
		t2: LD_RET_ADR = 1;
		t3: LD_NEW_AD = 1;

33	CALL [N]	t0: DATA_MUX = 7; ADR_MUX = 2;
		t1: LD_OUTPUT; DATA_MUX = 3; ADR_MUX = 1;
		t2: LD_RET_ADR = 1;
		t3:LD_NEW_AD = 1;

34	AND RR, RR	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_2 = 1, ALU_ADR_5 = 0, ALU_ADR_3 = 4 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1 t4: nop

35	AND RR, [N]	t0: ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1
		ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR5 = 0; ALU_ADR3 = 4;
		t2: LD_ALU = 1; ALU_ADR4 = 0; DATA_MUX = 2; BANK_ADR4 = 0;
		BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR = 1;

36	NAND RR, RR	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_2 = 1, ALU_ADR_5 = 0, ALU_ADR_3 = 5 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1 t4: nop

37	NAND RR, [N]	t0: ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1
		ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR5 = 0; ALU_ADR3 = 5;
		t2: LD_ALU = 1; ALU_ADR4 = 0; DATA_MUX = 2; BANK_ADR4 = 0;
		BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR = 1;

38	OR RR, RR	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_2 = 1, ALU_ADR_5 = 1, ALU_ADR_3 = 4 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1

		t4: nop

39	OR RR, [N]	t0: ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1
		ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR5 = 1; ALU_ADR3 = 4;
		t2: LD_ALU = 1; ALU_ADR4 = 0; DATA_MUX = 2; BANK_ADR4 = 0;
		BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR = 1;

40	NOR RR,RR	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_2 = 1, ALU_ADR_5 = 1, ALU_ADR_3 = 5 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1 t4: nop

41	NOR RR, [N]	t0: ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1
		ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR5 = 1; ALU_ADR3 = 5;
		t2: LD_ALU = 1; ALU_ADR4 = 0; DATA_MUX = 2; BANK_ADR4 = 0;
		BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR = 1;

42	XOR RR,RR	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_2 = 1, ALU_ADR_5 = 2, ALU_ADR_3 = 4 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1 t4: nop

43	XOR RR, [N]	t0: ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1
		ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR5 = 2; ALU_ADR3 = 4;
		t2: LD_ALU = 1; ALU_ADR4 = 0; DATA_MUX = 2; BANK_ADR4 = 0;
		BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR = 1;

44	XNOR RR,RR	t0: BANK_LD_ADR1 = 1, BANK_ADR2 = 3, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_2 = 1, ALU_ADR_5 = 2, ALU_ADR_3 = 5 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1 t4: nop

45	XNOR RR, [N]	t0: ADR_MUX = 2; DATA_MUX = 7;
		t1: LD_OUTPUT = 1; DATA_MUX = 3; BANK_LD_ADR1 = RR; BANK_ADR4 = 1
		ALU_ADR_1 = 2; ALU_ADR_2 = 0; ALU_ADR5 = 2; ALU_ADR3 = 5;
		t2: LD_ALU = 1; ALU_ADR4 = 0; DATA_MUX = 2; BANK_ADR4 = 0;
		BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR = 1;

46	NOT RR	t0: BANK_LD_ADR1 = 0, BANK_ADR4 = 0, ALU_ADR_1 = 0, ALU_ADR_5 = 3, ALU_ADR_3 = 5 t1: LD_ALU = 1, ALU_ADR_4 = 0, MUX_DATA = 2, BANK_ADR4 = 0, BANK_ADR5 = 0, BANK_LD_ADR1 = 2, t3: BANK_LD_RR = 1 t4: nop

47	NOT [N]	t0: DATA_MUX = 7; ADR_MUX = 2; MEM_LD = 0;
		t1: LD_OUTPUT; DATA_MUX = 3; BANK_ADR4 = 1; ALU_ADR_1 = 0; ALU_ADR5 = 3; ALU_ADR3 = 5;
		t2: LD_ALU = 1; DATA_MUX = 0; BANK_LD_ADR1 = 0;

		t3: BANK_LD_RR = 1;

48	PUSH RR	t0: BANK_LD_ADR = RR; DATA_MUX = 0; SP_MUX = 1; ADR_MUX = 3;
		t1: LDSP = 1;
		t2: MEM_LD = 1; LD_STOS = 1;
		t3: NOP

49	POP RR	t0: SP_MUX = 0; ADR_MUX = 3; MEM_LD = 0;
		t1: LD_OUTPUT = 1;
		t2: LD_STOS = 1; DATA_MUX = 4; BANK_ADR4 = 0; BANK_ADR5 = 0;
		BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR = 1;

50	TOP RR	t0: SP_MUX = 0; ADR_MUX = 3; MEM_LD = 0;
		t1: LD_OUTPUT = 1;
		t2: LD_STOS = 1; DATA_MUX = 4; BANK_ADR4 = 0; BANK_ADR5 = 0;
		BANK_LD_ADR1 = RR;
		t3: BANK_LD_RR = 1;

51	TOP_2 RR	t0: BANK_ADR4 = 0; BANK_ADR5 = 0; BANK_LD_ADR1 = RR; DATA_MUX = 0;
		t1: SP_MUX = 0; ADR_MUX = 3;
		t2: MEM_LD = 1;
		t3: LD_STOS = 1;

52	PC RR	t0: DATA_MUX = 5; BANK_ADR4 = 0; BANK_ADR5 = 0 ;
		t1: BANK_LD_ADR1 = RR;
		t2: BANK_LD_RR = 1;
		t3: NOP

6.1 Przykład kodu wykonawczego w języku Asembler

Kod wykonujący wybrane operacje w języku Asembler, takie jak przesył danych między rejestrami, rejestrem i wartością natychmiastową, dodawanie i odejmowanie 8-bitowe, dodawanie 16-bitowe, porównywanie dwóch wartości i wykonywanie na tej podstawie skoku warunkowego, wywołanie procedury, operacje logiczny or i and oraz operacje na stosie.

```
mov al, 5 ;przesłanie danych rej-nat
mov bl, al ;przesłanie danych rej-rej, bl = 5

adc al, 2 ;dodawanie z flagą rej-nat, al = 7
sub bl, 1; odejmowanie rej-nat, bl = 4

mov ah, 1 ;przesłanie danych rej-nat, ah = 1, ax = 263
mov bh, 1 ;przesłanie danych rej-nat, bh = 1, bx = 260

add ax, bx ;dodawanie 16-bitowe rej-rej, ax = 523
;ah = 2, al = 11, bh = 1, bl = 4

cmp al, bl ;porównanie rej-rej al > bl, ustawienie flagi większości
jg wiekszy ;skok warunkowy jeśli większe do etykiety "wiekszy"

wiekszy:
    call procedura ;wywołanie nat procedury

or al, bl ;operacja logiczna or rej-rej, al = 15
and al, 12 ;operacja logiczna and rej-nat, al = 12

push al ;wysłanie danych al do stosu
push bl ;wysłanie danych bl do stosu
; stos: 4, 12
TOP_2 ah ;modyfikuje szczyt stosu na wartość z rejestru ah
; stos: 2, 12
TOP al ;zwraca wartość szczytu bez usuwania, al = 2
; stos: 2, 12
POP bl ;zdjęcie wartości ze stosu, bl = 2
; stos: 12

procedura:
    nop ;nie rób nic
    ret ;powrót z procedury
```

Listing 1: Przykładowy program w języku Asembler

6.2 Przykład pseudokodu wykonawczego

Pseudokod wykonujący wybrane operacje, takie jak przesył danych między rejestrami, rejestrem i wartością natychmiastową, dodawanie i odejmowanie 8-bitowe, dodawanie 16-bitowe, porównywanie dwóch wartości i wykonywanie na tej podstawie skoku warunkowego, wywołanie procedury, operacje logiczny or i and oraz operacje na stosie.

```
Prześlij wartość 5 do rejestru al
Prześlij zawartość rejestru al do rejestru bl
Wykonaj dodawanie z flagą rejestru al i wartości 2
Wykonaj odejmowanie bez flagi rejestru bl i wartości 1
Prześlij wartość 1 do rejestru ah
Prześlij wartość 1 do rejestru bh
Wykonaj dodawanie rejestrów 16-bitowych ax i bx
Porównaj wartości rejestrów al i bl
Wykonaj skok warunkowy do etykiety “wiekszy” jeśli flaga większe jest ustawiona
na 1
Etykieta “wiekszy”: wywołaj procedurę “procedura”
Wykonaj operację logiczną or na rejestrach al i bl
Wykonaj operację logiczną and na rejestrze al i wartości 12
Wyślij na stos wartość rejestru al
Wyślij na stos wartość rejestru bl
Modyfikuj wartość na szczycie stosu wartością rejestru ah
Pobierz wartość szczytu stosu bez zdejmowania do rejestru al
Zdejmij wartość ze stosu do rejestru bl

Procedura “procedura”:
nie rób nic
powrót z procedury
```

Listing 2: Przykładowy program w pseudokodzie

7 Kod HDL

Kod przedstawia podstawowe komponenty użyte w projekcie takie jak multipleksery, demultipleksery, sumatory, subtraktory, komparator, bramki logiczne oraz rejestry.

7.1 Kod multiplekserów

Kod VHDL multiplekserów użytych w projekcie.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2x1_8b is
  port(we0, we1: in std_logic_vector(7 downto 0);
       adr: in std_logic;
       wy: out std_logic_vector(7 downto 0));
end mux2x1_8b;

architecture behavioral_mux2x1_8b of mux2x1_8b is
begin
  with adr select
       wy <= we0 when '0', we1 when others;
end behavioral_mux2x1_8b ;
```

Listing 3: Kod układu multipleksera 2x1 8-bitowego

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

entity mux3x1_8b is
  port(we0, we1, we2: in std_logic_vector(7 downto 0);
       adr: in std_logic_vector(1 downto 0);
       wy: out std_logic_vector(7 downto 0));
end mux3x1_8b;

architecture mux3x1_8b_arch of mux3x1_8b is
begin
  with adr select
       wy <= we0 when "00", we1 when "01", we2 when others;
end mux3x1_8b_arch;
```

Listing 4: Kod układu multipleksera 3x1 8-bitowego

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux4x1_8b is
  port(we0, we1, we2, we3: in std_logic_vector(7 downto 0);
        adr: in std_logic_vector(1 downto 0);
        wy: out std_logic_vector(7 downto 0));
end mux4x1_8b;

architecture behavioral_mux4x1_8b of mux4x1_8b is
begin
  with adr select
    wy <= we0 when "00", we1 when "01", we2 when "10", we3 when others;
end behavioral_mux4x1_8b ;
```

Listing 5: Kod układu multipleksera 4x1 8-bitowego

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

entity mux6x1_8b is
  port(we0, we1, we2, we3, we4, we5: in std_logic_vector(7 downto 0);
       adr: in std_logic_vector(2 downto 0);
       wy: out std_logic_vector(7 downto 0));
end mux6x1_8b;

architecture behavioral_mux6x1_8b of mux6x1_8b is
begin
  with adr select
       wy <= we0 when "000", we1 when "001", we2 when "010", we3 when "011", we4
when "100", we5 when others;
end behavioral_mux6x1_8b;
```

Listing 6: Kod układu multipleksera 6x1 8-bitowego

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux10x1_8b is
    port(we0, we1, we2, we3, we4, we5, we6, we7, we8, we9, we10: in
std_logic_vector(7 downto 0);
       adr: in std_logic_vector(3 downto 0);
       wy: out std_logic_vector(7 downto 0));
end mux2x1_8b;

architecture behavioral_mux10x1_8b of mux10x1_8b is
begin
  with adr select
    wy <= we0 when '0000', we1 when "0001", we2 when "0010", we3 when "0011",
we4 when "0100", we5 when "0101", we6 when "0110", we7 when "0111", we8 when
"1000", we9 when "1001", we10 when others;
end behavioral_mux10x1_8b ;
```

Listing 7: Kod układu multipleksera 10x1 8-bitowego

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

entity mux4x1_16b is
  port(ADR: in std_logic_vector(1 downto 0);
  WE0, WE1, WE2, WE3: in std_logic_vector(15 downto 0);
  WY: out std_logic_vector(15 downto 0));
end mux4x1_16b;

architecture behavioral_mux4x1_16b of mux4x1_16b is
begin
  with ADR select
      WY <= WE0 when "00", WE1 when "01", WE2 when "10", WE3 when others;
end behavioral_mux4x1_16b;
```

Listing 8: Kod układu multipleksera 4x1 16-bitowego

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

entity mux2x1_16b is
  port(we0, we1: in std_logic_vector(15 downto 0);
       adr: in std_logic;
       wy: out std_logic_vector(15 downto 0));
end mux2x1_16b;

architecture behavioral_mux2x1_16b of mux2x1_16b is
begin
  with adr select
       wy <= we0 when '0', we1 when others;
end behavioral_mux2x1_16b;
```

Listing 9: Kod układu multipleksera 2x1 16-bitowego

7.2 Kod demultiplekserów

Kod VHDL demultiplekserów użytych w projekcie.

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

entity mux1x2_8b is
  port(we: in std_logic_vector(7 downto 0);
       adr: in std_logic;
       wy0, wy1: out std_logic_vector(7 downto 0));
end mux1x2_8b;

architecture behavioral_mux1x2_8b of mux1x2_8b is
begin
  process(we, adr)
  begin
       if adr = '0' then
         wy0 <= we;
         wy1 <= "00000000";
       else
         wy0 <= "00000000";
         wy1 <= we;
       end if;
  end process;
end behavioral_mux1x2_8b;
```

Listing 10: Kod układu demultipleksera 1x2 8-bitowego

7.3 Kod komparatora

Kod VHDL komparatora 8-bitowego użytego w projekcie.

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

entity comp_8b is
  port(a, b: in std_logic_vector(7 downto 0);
       wym, wyw, wyr: out std_logic);
end comp_8b;

architecture behavioral_comp_8b of comp_8b is
begin
  process(a, b)
  begin
       wyw <= '0';
       wym <= '0';
       wyr <= '0';
       if a > b then
           wyw <= '1';
       elsif a < b then
           wym <= '1';
       else
           wyr <= '1';
       end if;
  end process;
end behavioral_comp_8b;
```

Listing 11: Kod układu komparatora 8-bitowego

7.4 Kod sumatora

Kod VHDL sumatorów 8 i 16-bitowych użytych w projekcie.

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

entity add_8b is
  port(A, B: in std_logic_vector(7 downto 0);
       Cin: in std_logic;
       S: out std_logic_vector(7 downto 0);
       Cout, Oout, Zout: out std_logic);
end add_8b;

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 behavioral_full_add of full_add is
begin
  S <= A xor B xor Cin;
  Cout <= (A and B) or (Cin and (A or B));
end behavioral_full_add;

architecture behavioral_add_8b of add_8b 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: std_logic_vector(7 downto 0);
begin
  s0: full_add port map (A(0), B(0), Cin,   LS(0), Cr(0));
  s1: full_add port map (A(1), B(1), Cr(0), LS(1), Cr(1));
  s2: full_add port map (A(2), B(2), Cr(1), LS(2), Cr(2));
  s3: full_add port map (A(3), B(3), Cr(2), LS(3), Cr(3));
  s4: full_add port map (A(4), B(4), Cr(3), LS(4), Cr(4));
  s5: full_add port map (A(5), B(5), Cr(4), LS(5), Cr(5));
  s6: full_add port map (A(6), B(6), Cr(5), LS(6), Cr(6));
  s7: full_add port map (A(7), B(7), Cr(6), LS(7), Cr(7));
  Cout <= Cr(7);
  Oout <= Cr(7) xor Cr(6);
  S <= LS;
end behavioral_add_8b;
```

Listing 12: Kod układu sumatora 8-bitowego

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

entity add_16b is
  port(L_A, L_B, H_A, H_B: in std_logic_vector(7 downto 0);
       L_WY, H_WY: out std_logic_vector(7 downto 0);
       Cout, Oout: out std_logic);
end add_16b;

architecture behavioral_add_16b of add_16b is
  component add_8b is
       port(A, B: in std_logic_vector(7 downto 0);
       Cin: in std_logic;
       S: out std_logic_vector(7 downto 0);
       Cout, Oout: out std_logic);
  end component;

  signal Cr, LS: std_logic_vector(7 downto 0);

begin
  s0: add_8b port map (L_A, L_B, "0", L_WY, Cr(0), Oout);
  s1: add_8b port map (H_A, H_B, Cr(0), H_WY, Cout, Oout);
end behavioral_add_16b;
```

Listing 13: Kod układu sumatora 16-bitowego o 4 wejściach 8 bitowych

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

entity sum16b is
  port(a, b: in std_logic_vector(15 downto 0);
       wy: out std_logic_vector(15 downto 0));
end sum16b;

architecture behavioral_sum16b of sum16b is
begin
  wy <= a+b;
end behavioral_sum16b;
```

Listing 14: Kod układu sumatora 16-bitowego o 2 wejściach

7.5 Kod subtraktor

Kod VHDL subtraktorów 8 i 16-bitowych użytych w projekcie.

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

entity sub_8b is
  port(A, B: in std_logic_vector(7 downto 0);
  Cin: in std_logic;
  S: out std_logic_vector(7 downto 0);
  Cout, Oout: out std_logic);
end sub_8b;

architecture behavioral_sub_8b of sub_8b is

component add_8b
  port(A, B: in std_logic_vector(7 downto 0);
  Cin: in std_logic;
  S: out std_logic_vector(7 downto 0);
  Cout, Oout: out std_logic);
end component;

signal Cr, LS: std_logic_vector(7 downto 0);
signal B_negation: std_logic_vector(7 downto 0);

begin
  B_negation <= NOT B;
  s0: add_8b port map (A(0), B_negation(0), '1', LS(0), Cr(0), Oout);
  s1: add_8b port map (A(1), B_negation(1), '1', LS(1), Cr(1), Oout);
  s2: add_8b port map (A(2), B_negation(2), '1', LS(2), Cr(2), Oout);
  s3: add_8b port map (A(3), B_negation(3), '1', LS(3), Cr(3), Oout);
  s4: add_8b port map (A(4), B_negation(4), '1', LS(4), Cr(4), Oout);
  s5: add_8b port map (A(5), B_negation(5), '1', LS(5), Cr(5), Oout);
  s6: add_8b port map (A(6), B_negation(6), '1', LS(6), Cr(6), Oout);
  s7: add_8b port map (A(7), B_negation(7), '1', LS(7), Cr(7), Oout);
  Cout <= Cr(7);
  Oout <= Cr(7) xor Cr(6);
  S <= LS;
end behavioral_sub_8b;
```

Listing 15: Kod układu subtraktora 8-bitowego

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

entity sub_16b is
  port(L_A, L_B, H_A, H_B: in std_logic_vector(7 downto 0);
       L_WY, H_WY: out std_logic_vector(7 downto 0);
       Cout, Oout: out std_logic);
end sub_16b;

architecture behavioral_sub_16b of sub_16b is
  component sub_8b is
       port(A, B: in std_logic_vector(7 downto 0);
       Cin: in std_logic;
       S: out std_logic_vector(7 downto 0);
       Cout, Oout: out std_logic);
  end component;

  signal Cr, LS: std_logic_vector(7 downto 0);

  begin
       s0: sub_8b port map (L_A, L_B, "0", L_WY, Cr(0), Oout);
       s1: sub_8b port map (H_A, H_B, Cr(0), H_WY, Cout, Oout);
end behavioral_sub_16b;
```

Listing 16: Kod układu subtraktora 16-bitowego

7.6 Kod bramek logicznych

Kod VHDL bramek logicznych użytych w projekcie.

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

entity gate_or is
  port(WE1, WE2: in std_logic_vector(7 downto 0);
  WY: out std_logic_vector(7 downto 0));
end gate_or;

architecture behavioral_gate_or of gate_or is
begin
  WY <= WE1 or WE2;
end behavioral_gate_or;
```

Listing 17: Kod układu bramki or

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

entity gate_and is
  port(WE1, WE2: in std_logic_vector(7 downto 0);
  WY: out std_logic_vector(7 downto 0));
end gate_and;

architecture behavioral_gate_and of gate_and is
begin
  WY <= WE1 and WE2;
end behavioral_gate_and;
```

Listing 18: Kod układu bramki and

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

entity gate_xor is
  port(WE1, WE2: in std_logic_vector(7 downto 0);
  WY: out std_logic_vector(7 downto 0));
end gate_xor;

architecture behavioral_gate_xor of gate_xor is
begin
  WY <= WE1 xor WE2;
end behavioral_gate_xor;
```

Listing 19: Kod układu bramki xor

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

entity gate_not is
  port(WE1, WE2: in std_logic_vector(7 downto 0);
  WY: out std_logic_vector(7 downto 0));
end gate_not;

architecture behavioral_gate_not of gate_not is
begin
  WY <= WE1 not WE2;
end behavioral_gate_not;
```

Listing 20: Kod układu bramki not

7.7 Kod rejestrów

Kod VHDL rejestrów 8 i 16-bitowych użytych w projekcie.

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

entity reg_8b is
  port(we: in std_logic_vector(7 downto 0);
       wy: out std_logic_vector(7 downto 0);
       ld: in std_logic);
end reg_8b;

architecture behavioral_reg_8b of reg_8b is
begin
  process(ld)
  begin
       if rising_edge(ld) then
       wy <= we;
       end if;
  end process;
end behavioral_reg_8b;
```

Listing 21: Kod układu rejestru 8-bitowego

```
entity reg_16b is
  port(DATA: in std_logic_vector(15 downto 0);
       Q: out std_logic_vector(15 downto 0);
       RESET: in std_logic;
       LD_H, LD_L: in std_logic);
end reg_16b;

architecture behavioral_reg_16b of reg_16b is
  component reg_8b is
       port(we: in std_logic_vector(7 downto 0);
       wy: out std_logic_vector(7 downto 0);
       ld: in std_logic);
  end component;

  signal najstarsze, najmlodsze: std_logic_vector(7 downto 0);

begin
  s0: reg_8b port map (najstarsze, najstarsze, LD_H);
  s1: reg_8b port map (najmlodsze, najmlodsze, LD_L);

  process(ld)
  begin
    if rising_edge(ld) then
       if LD_H = '1' then
         najstarsze <= we(15 downto 8);
       end if;
       if LD_L = '1' then
         najmlodsze <= we(7 downto 0);
       end if;
       wy <= we;
    end if;
  end process;
end behavioral_reg_16b;
```

Listing 22: Kod układu rejestru 16-bitowego użytego w banku rejestrów

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

entity reg16b is
  port(we: in std_logic_vector(15 downto 0);
       wy: out std_logic_vector(15 downto 0);
       ld: in std_logic);
end reg16b;

architecture behavioral_reg16b of reg16b is
begin
  process(ld)
  begin
       if rising_edge(ld) then
       wy <= we;
       end if;
  end process;
end behavioral_reg16b;
```

Listing 23: Kod układu rejestru 16-bitowego

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

entity reg1b is
  port(we: in std_logic;
       wy: out std_logic;
       ld: in std_logic);
end reg1b;

architecture behavioral_reg1b of reg1b is
begin
  process(ld)
  begin
       if rising_edge(ld) then
       wy <= we;
       end if;
  end process;
end behavioral_reg1b;
```

Listing 24: Kod układu rejestru 1-bitowego

9 Spis rysunków

10 Spis listingów

11 Spis tabel