Projekt SYKO

RISC-V Implementacja wirtualnego procesora

w semestrze zimowym 2020

Jakub Szymański
Nr albumu *300387*

Jakub Jończyk
Nr albumu *284336*

Mikołaj Grajeta
Nr albumu *299119*

Prowadzący: mgr inż. Aleksander Pruszkowski

Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Włocławek 2021

Oświadczenie o samodzielnie wykonanej pracy.

Oświadczamy, że niniejsza praca, stanowiąca podstawę do uznania osiągnięcia efektów uczenia się z przedmiotu SYKO została wykonana przeze mnie samodzielnie.

Wpisanie w linii poniżej imienia, nazwiska i numeru albumu oraz daty jest jednoznaczne z podpisaniem oświadczenia

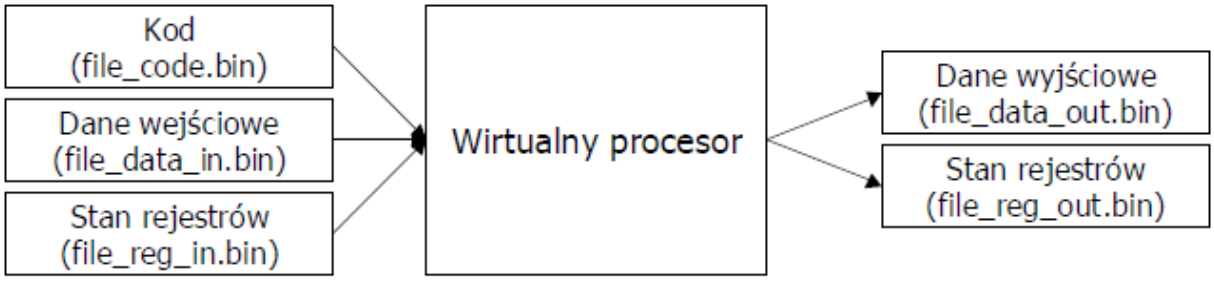
Jakub Szymański 300387 Włocławek, 26.01.2021

1. Implementacja wirtualnego procesora.

Implementacja wirtualnego procesora została zapisana w języku C. Na podstawie zdobytej wiedzy wykładowej oraz dokumentacji architektury RISC-V zostały zaimplementowane następujące funkcje:

- **SW**
- **SLL**
- **LUI**
- **JAL**
- **AUIPC**
- **ADD**
- **SLTI**
- **SB**
- **JALR**
- **BNE**
- **ADDI**

Kompilacja odbywała się za pomocą GCC pod systemem LINUX MINT 19.1. Nie zostały użyte konstrukcje typowe dla języków C++ czy też C#. Poniższy rysunek przedstawia przykładowy informacji wejściowych i wyjściowych.



Wirtualny procesor przyjmuje trzy pliki:

- Kod – file_code.bin
- Dane wejściowe – file_data_in.bin
- Stan rejestrów file_reg_in.bin

Później aplikacja wykonuje zadane instrukcje napisane w plikach asemblerowych i zapisuje zawartości pamięci do plików:

- Dane wyjściowe – file_data_out.bin
- Stan rejestrów – file_reg_out.bin

Na bazie tych wyników stwierdzaliśmy czy implementacja funkcji jest poprawna i procesor działa tak jak zamierzono. Implementowany procesor to RV32I, czyli 32 bitowa wersja RISC-V. Oznacza to, że mamy do dyspozycji stałoprzecinkowe 32 rejestry 32 bitowe.

2. Implementacja – postać źródłowa.

Wirtualny procesor składa się z plików źródłowych:

- **main.c:**
  W tym pliku znajduje się pętla aplikacji , w której znajduje się funkcja switch, dzięki której implementowany procesor może wybrać jakiej funkcji aktualnie użyć zgodnie z danymi wejściowymi. Są tu również wywoływane funkcje odczytywania i zapisu do pliku.

- **mem_abs.c:**
  Plik zawiera implementacje funkcji potrzebnych do obsługi odczytu, zapisu do plików oraz obsługi wirtualnego procesora, m. in. Zapis do pamięci, licznik PC. Zostały również napisane funkcje do wyłuskiwania argumentów RD, RS1 i RS2.

- **opcodes.h:**
  Znajdują się tu wzorce opcodów poszczególnych funkcji oraz wzorce opcodów + func3 i funct 7. Plik zawiera również nagłówki funkcji dla poszczególnych opcodów.

- **types.h:**
  Definiowane są tutaj używane typy danych.

- **merror.h:**
  Zdefiniowane numery dla poszczególnych errorów i sytuacji wyjątkowych.

- **Pliki źródłowe implementowanych funkcji, np. f_sw.c:**
  W plikach tych znajdują się implementacje funkcji, które zostały przydzielone do naszego zespołu.

  Kompilacja odbywa się za pomocą pliku makefile i kompilatora GCC:

```
CC=gcc
CFLAGS=-c -Wall

syko.out: mem_abs.c f_add.c f_addi.c f_auipc.c f_bne.c f_jal.c f_jalr.c f_lui.c f_sb.c f_sll.c f_slti.c f_sw.c main.c
	$(CC) $(CFLAGS) mem_abs.c -o mem_abs.o
	#make functions
	$(CC) $(CFLAGS) f_add.c -o f_add.o
	$(CC) $(CFLAGS) f_addi.c -o f_addi.o
	$(CC) $(CFLAGS) f_auipc.c -o f_auipc.o
	$(CC) $(CFLAGS) f_bne.c -o f_bne.o
	$(CC) $(CFLAGS) f_jal.c -o f_jal.o
	$(CC) $(CFLAGS) f_jalr.c -o f_jalr.o
	$(CC) $(CFLAGS) f_lui.c -o f_lui.o
	$(CC) $(CFLAGS) f_sb.c -o f_sb.o
	$(CC) $(CFLAGS) f_sll.c -o f_sll.o
	$(CC) $(CFLAGS) f_slti.c -o f_slti.o
	$(CC) $(CFLAGS) f_sw.c -o f_sw.o
	#main
	$(CC) $(CFLAGS) main.c -o main.o
	$(CC)  mem_abs.o f_add.o f_addi.o f_auipc.o f_bne.o f_jal.o f_jalr.o f_lui.o f_sb.o f_sll.o f_slti.o f_sw.o main.o -o syko.out
	#done

test: syko.out
	syko.out

clean:
	rm -f mem_abs.o
	rm -f f_add.o
	rm -f f_addi.o
	rm -f f_auipc.o
	rm -f f_bne.o
	rm -f f_jal.o
	rm -f f_jalr.o
	rm -f f_lui.o
	rm -f f_sb.o
	rm -f f_sll.o
	rm -f f_slti.o
	rm -f f_sw.o
	rm -f main.o
	#done
```

Program kompiluje się poprawnie bez błędów i ostrzeżeń, jak również wczytuje poprawnie pliki wejściowe i zapisuje wynik w plikach wyjściowych.

Opis poszczególnych funkcji:

- **void checkR0(DataType RD)**
  Funkcja ta sprawdza czy podany rejestr wynikowy nie jest rejestrem x0. Rejestr ten, alias Zero, ma zawsze zawartość zero, w RISC-V nic nie można wpisać do jego treści. Funkcja przyjmuje argument DataType RD, czyli uint32_t, natomiast nic nie zwraca, jeśli RD jest równe zero wtedy ten rejestr ustawiany jest na wartość 0.

```
void checkR0(DataType RD){
        if(RD == 0){
                setRegister(RD, 0);
    }
}
```

- **DataType getRD(void)**
  Funkcja ta zwraca argument wynikowy z treści otrzymanej z plików wejściowych, nakładana jest maska na odpowiednie bity i przesunięta jest ta wartość w prawo o 7 bitów. Jak wiemy z dokumentacji procesora RISC-V argument wynikowy RD w instrukcjach typu R, I, U, J znajduje się na bitach o numerach od 7 do 11. Stąd użyte przesunięcie o 7 bitów w prawo i maska $\& \, 0xF80$.

```
DataType getRD(void){
   return (getMEMC(getPC()) & 0xF80) >> 7;
}
```

- **DataType getRS1(void)**
  Funkcja ta zwraca argument źródłowy 1 z treści otrzymanej z plików wejściowych, nakładana jest maska na odpowiednie bity i przesunięta jest ta wartość w prawo o 15 bitów. Jak wiemy z dokumentacji procesora RISC-V argument źródłowy 1 w instrukcjach typu R, I, S, B znajduje się na bitach o numerach od 15 do 19. Stąd użyte przesunięcie o 15 bitów w prawo i maska $\& \, 0xF8000$.

```
DataType getRS1(void){
    return (getMEMC(getPC()) & 0xF8000) >>15;
}
```

- **DataType getRS2(void)**
  Funkcja ta zwraca argument źródłowy 2 z treści otrzymanej z plików wejściowych, nakładana jest maska na odpowiednie bity i przesunięta jest ta wartość w prawo o 15 bitów. Jak wiemy z dokumentacji procesora RISC-V argument źródłowy 2 w instrukcjach typu R, S, B znajduje się na bitach o numerach od 19 do 24. Stąd użyte przesunięcie o 19 bitów w prawo i maska $\& \, 0xF8000$.

```
DataType getRS2(void){
    return (getMEMC(getPC()) & 0x1F00000) >> 20;
}
```

3. Testowanie wirtualnego procesora.

Dla każdej badanej funkcji został napisany plik w języku assembler, funkcje zostały zbadane w trybie normalnym pracy procesora – sprawdzone zostały podstawowe funkcje, czyli wywołanie tych funkcji i poprawny zapis – oraz zachowanie aplikacji w warunkach brzegowych. Na każdym etapie testowania instrukcje testujące zostały skompilowane i jako pliki wejściowe zostały pobrane przez program.

3.1 Funkcja ADDI

a) Implementacja

ADDI jest to instrukcja typu I, argument wbudowany IMM jest dodawany do argumentu źródłowego RS1 i zapisywany w RD.

Instrukcja: **`ADDI RD, RS1, IMM12`**

```c
#include <stdio.h>
#include "types.h"
#include "mem_abs.h"

void F_ADDI(void)
{
        //writeOpcode();
        DataType RD = getRD();
        DataType RS1 = getRS1();
        DataType IMM12=(getMEMC(getPC()) & 0xFFF00000) >>20;

        printf("0x%04x: ADDI R%d, R%d, R%d\n", getPC(), RD, RS1, IMM12);

    setRegister(RD, (getRegister(RS1) + IMM12));
    checkR0(RD);
    incPC();
}
```

b) Przygotowanie plików testowych

```
.section .text
        .align 4
        .global _start
_start:
                addi x0, x0, 0x001 #przewidywanie w x0 0
                addi x1, x0, 0x001 #przewidywanie w x1 1
                addi x2, x0, 0x002 #przewidywanie w x2 2
                addi x3, x1, -0x001 #przewidywanie w x3 0
                addi x4, x1, -0x005 #przewidywanie w x4 -5
                addi x5, x4, 0x005 #przewidydywanie w x5 0
                addi x6, x0, 0x7FF #przewidydywanie w x6 MAX_WARTOSC
                addi x7, x0, -0x800 #przewidydywanie w x7 MIN_WARTOSC
            addi x8, x6, 0x7FF #0 MAX MAX
            addi x9, x8, 0x001
            addi x10, x7, -0x800 #0 MIN MIN
            addi x11, x10, 0x001
            addi x25, x0, 0x001
            addi x26, x25, 0x001
            addi x27, x26, 0x001
            addi x28, x27, 0x001
            addi x29, x28, 0x001
            addi x30, x29, 0x001
                addi x31, x0, 0x7FF

```

Kompilacja pliku .s oraz przygotowanie plików wejściowych przebiegło pomyślnie.

Po wykonanej kompilacji pliku z kodem testowym sprawdzono plik test.lst w celu sprawdzenia statusu po deasemblacji i rodzaju procesora.

```
test:     file format elf32-littleriscv
test
architecture: riscv:rv32, flags 0x00000012:
EXEC_P, HAS_SYMS
start address 0x00000000

Program Header:
    LOAD off    0x00000060 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000054 memsz 0x00000054 flags r-x

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000054  00000000  00000000  00000060  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .riscv.attributes 0000001a  00000000  00000000  000000b4  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    d  .text  00000000 .text
00000000 l    d  .riscv.attributes      00000000 .riscv.attributes
00000000 g       .text  00000000 _start
```

Jak widać architektura RISC-V zgadza się z projektem, jest to RV32I. Adres startowy PC to 0x00000000. Funkcje po deasemblacji:

```
00000000 <_start>:
   0:   00100013                li      zero,1
   4:   00100093                li      ra,1
   8:   00200113                li      sp,2
   c:   fff08193                addi    gp,ra,-1
  10:   ffb08213                addi    tp,ra,-5
  14:   00520293                addi    t0,tp,5 # 5 <_start+0x5>
  18:   7ff00313                li      t1,2047
  1c:   80000393                li      t2,-2048
  20:   7ff30413                addi    s0,t1,2047
  24:   00140493                addi    s1,s0,1
  28:   80038513                addi    a0,t2,-2048
  2c:   00150593                addi    a1,a0,1
  30:   00100c93                li      s9,1
  34:   001c8d13                addi    s10,s9,1
  38:   001d0d93                addi    s11,s10,1
  3c:   001d8e13                addi    t3,s11,1
  40:   001e0e93                addi    t4,t3,1
  44:   001e8f13                addi    t5,t4,1
  48:   7ff00f93                li      t6,2047
        ...

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0:   1941                    addi    s2,s2,-16
   2:   0000                    unimp
   4:   7200                    flw     fs0,32(a2)
   6:   7369                    lui     t1,0xffffa
   8:   01007663                bgeu    zero,a6,14 <_start+0x14>
   c:   0000000f                fence   unknown,unknown
  10:   7205                    lui     tp,0xfffe1
  12:   3376                    fld     ft6,376(sp)
  14:   6932                    flw     fs2,12(sp)
  16:   7032                    flw     ft0,44(sp)
  18:   0030                    addi    a2,sp,8
```

Skok do kolejnej instrukcji odbywa się co 4 bity, po deasemblacji można zauważyć, że rejestry x0, x1, … x31 zmieniły nazwy na aliasy, np. x0 – zero, x2 – ra (Return Address). Wszystkie instrukcje ADDI z argumentem źródłowym 1 jako x0 zostały zdeasemblowane na instrukcje LI.

c) Kompilacja i wykonanie aplikacji.

Kompilacja programu udała się, wynik wykonania aplikacji:

```
jszyman:glowny$ ./syko.out
T: 0x00100013
0x0000: ADDI R0, R0, R1
T: 0x00100093
0x0004: ADDI R1, R0, R1
T: 0x00200113
0x0008: ADDI R2, R0, R2
T: 0xfff08193
0x000c: ADDI R3, R1, R4095
T: 0xffb08213
0x0010: ADDI R4, R1, R4091
T: 0x00520293
0x0014: ADDI R5, R4, R5
T: 0x7ff00313
0x0018: ADDI R6, R0, R2047
T: 0x80000393
0x001c: ADDI R7, R0, R2048
T: 0x7ff30413
0x0020: ADDI R8, R6, R2047
T: 0x00140493
0x0024: ADDI R9, R8, R1
T: 0x80038513
0x0028: ADDI R10, R7, R2048
T: 0x00150593
0x002c: ADDI R11, R10, R1
T: 0x00100c93
0x0030: ADDI R25, R0, R1
T: 0x001c8d13
0x0034: ADDI R26, R25, R1
T: 0x001d0d93
0x0038: ADDI R27, R26, R1
T: 0x001d8e13
0x003c: ADDI R28, R27, R1
T: 0x001e0e93
0x0040: ADDI R29, R28, R1
T: 0x001e8f13
0x0044: ADDI R30, R29, R1
T: 0x7ff00f93
0x0048: ADDI R31, R0, R2047
T: 0x00000000
Wykryto nieznana instrukcje (PC=0x0000004c, T=0x00000000)
```

Jak widać aplikacja zadziałała poprawnie i rejestry zostały wczytane zgodnie z napisanym kodem testowym.

d) Sprawdzenie działania procesora

W celu sprawdzenia czy procesor poprawnie wykonuje zadane instrukcje został otworzony plik file_reg_out.bin komendą hexdump.

```
jszyman:test$ hexdump file_reg_out.bin
0000000 0000 0000 0001 0000 0002 0000 1000 0000
0000010 0ffc 0000 1001 0000 07ff 0000 0800 0000
0000020 0ffe 0000 0fff 0000 1000 0000 1001 0000
0000030 0000 0000 0000 0000 0000 0000 0000 0000
*
0000060 0000 0000 0001 0000 0002 0000 0003 0000
0000070 0004 0000 0005 0000 0006 0000 07ff 0000
0000080 004c 0000
0000084
```

Jak widać w rejestrze zero nie mamy żadnych wartości – jest tylko zero, co jest zgodne z oczekiwaniami. Kolejne analizy rejestrów potwierdzają działanie tej funkcji w wirtualnym procesorze.

```
addi x1, x0, 0x001 #przewidywanie w x1 1
addi x2, x0, 0x002 #przewidywanie w x2 2
```

3.2 Funkcja ADD

a) Implementacja

ADD jest to instrukcja typu R – funkcja dodaje zawartości 2 rejestrów i zapisuje wynik pod trzecim rejestrem (RD):

`RD = RS1 + RS2`

Instrukcja: **`ADD RD, RS1, RS2`**

```c
#include <stdio.h>

#include "types.h"
#include "mem_abs.h"

void F_ADD(void){
    writeOpcode();
    DataType RS1 = getRS1();
    DataType RS2 = getRS2();
    DataType RD = getRD();

    printf("0x%04x: ADD R%d, R%d, R%d\n", getPC(), RD, RS1, RS2);

    setRegister(RD, (getRegister(RS1) + getRegister(RS2)));    //właściwe obliczenie
    checkR0(RD);
    incPC();                                                   //zwiększenie licznika rozkazów
}

```

b) Przygotowanie plików testowych

```
.data
    .align 4
.text
    .align  4
    .global _start
_start:
    li x1, 0x111
    li x2, 0x7FF
    add x0, x1, x2 #expect 0
    li x3, 0x001
    li x4, 0x001
    add x5, x3, x4 #expect 2
    li x6, 0x7FF
    li x7, 0x7FF
    add x8, x6, x7 #expect 0xFFE
    li x9, -0x800
    li x10, -0x800
    add x11, x9, x3
    add x12, x10, x9
    add x13, x7, x0
    add x14, x7, x3
    li x28, 0x002
    li x29, 0x13D
    add x30, x29, x28
    add x31, x29, x10

```

Zaimplementowane testy znajdują się w plikach
z rozszerzeniem `.s`

Funkcje po deasemblacji:

```

00000000 <_start>:
   0:   11100093                li      ra,273
   4:   7ff00113                li      sp,2047
   8:   00208033                add     zero,ra,sp
   c:   00100193                li      gp,1
  10:   00100213                li      tp,1
  14:   004182b3                add     t0,gp,tp
  18:   7ff00313                li      t1,2047
  1c:   7ff00393                li      t2,2047
  20:   00730433                add     s0,t1,t2
  24:   80000493                li      s1,-2048
  28:   80000513                li      a0,-2048
  2c:   003485b3                add     a1,s1,gp
  30:   00950633                add     a2,a0,s1
  34:   000386b3                add     a3,t2,zero
  38:   00338733                add     a4,t2,gp
  3c:   00200e13                li      t3,2
  40:   13d00e93                li      t4,317
  44:   01ce8f33                add     t5,t4,t3
  48:   00ae8fb3                add     t6,t4,a0
        ...

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0:   1941                    addi    s2,s2,-16
   2:   0000                    unimp
   4:   7200                    flw     fs0,32(a2)
   6:   7369                    lui     t1,0xffffa
   8:   01007663                bgeu    zero,a6,14 <_start+0x14>
   c:   0000000f                fence   unknown,unknown
  10:   7205                    lui     tp,0xfffe1
  12:   3376                    fld     ft6,376(sp)
  14:   6932                    flw     fs2,12(sp)
  16:   7032                    flw     ft0,44(sp)
  18:   0030                    addi    a2,sp,8

```

c) Komplilacja i wykonanie programu

Kompilacja powiodła się, po uruchomieniu otrzymujemy:

```
jszyman:glowny$ ./syko.out
T: 0x11100093
0x0000: ADDI R1, R0, R273
T: 0x7ff00113
0x0004: ADDI R2, R0, R2047
T: 0x00208033
0x0008: ADD R0, R1, R2
T: 0x00100193
0x000c: ADDI R3, R0, R1
T: 0x00100213
0x0010: ADDI R4, R0, R1
T: 0x004182b3
0x0014: ADD R5, R3, R4
T: 0x7ff00313
0x0018: ADDI R6, R0, R2047
T: 0x7ff00393
0x001c: ADDI R7, R0, R2047
T: 0x00730433
0x0020: ADD R8, R6, R7
T: 0x80000493
0x0024: ADDI R9, R0, R2048
T: 0x80000513
0x0028: ADDI R10, R0, R2048
T: 0x003485b3
0x002c: ADD R11, R9, R3
T: 0x00950633
0x0030: ADD R12, R10, R9
T: 0x000386b3
0x0034: ADD R13, R7, R0
T: 0x00338733
0x0038: ADD R14, R7, R3
T: 0x00200e13
0x003c: ADDI R28, R0, R2
T: 0x13d00e93
0x0040: ADDI R29, R0, R317
T: 0x01ce8f33
0x0044: ADD R30, R29, R28
T: 0x00ae8fb3
0x0048: ADD R31, R29, R10
T: 0x00000000
Wykryto nieznana instrukcje (PC=0x0000004c, T=0x00000000)
```

Efekt działania wygląda poprawnie

d) Sprawdzenie działania procesora

W celu sprawdzenia czy procesor poprawnie wykonuje zadane instrukcje został otworzony plik `file_reg_out.bin` komendą *hexdump*:

```
jszyman:test$ hexdump file_reg_out.bin
0000000 0000 0000 0111 0000 07ff 0000 0001 0000
0000010 0001 0000 0002 0000 07ff 0000 07ff 0000
0000020 0ffe 0000 0800 0000 0800 0000 0801 0000
0000030 1000 0000 07ff 0000 0800 0000 0000 0000
0000040 0000 0000 0000 0000 0000 0000 0000 0000
*
0000070 0002 0000 013d 0000 013f 0000 093d 0000
0000080 004c 0000
0000084
```

Rejestr zerowy zawiera same zera, co jest zgodne z oczekiwaniami. W x0 otrzymaliśmy 0, w x5 otrzymaliśmy 2, zatem efekt pokrywa się z przewidywaniami.

3.3 Funkcja AUIPC

a) Implementacja

Funkcja AUIPC (Add Upper Immediate to PC) wykunuje instrukcję typu U, która dodaje 20-bitową wartość z pamięci wewnętrznej IMM do 12 najstarszych bitów PC i zapisuje 32-bitowy wynik pod rejestrem RD.

Instrukcja: `AUIPC RD, IMM20`

Ciało funkcji:

```c
#include <stdio.h>

#include "types.h"
#include "mem_abs.h"

void F_AUIPC(void){
    writeOpcode();
    DataType RD = getRD();
    DataType IMM20=(getMEMC(getPC()) & 0xFFFFF000) >>12;

    printf("0x%04x: AUIPC R%d, R%d\n", getPC(), RD, IMM20);

    setRegister(RD, (getPC() + (IMM20 << 12)));
    checkR0(RD);
    incPC();
}

```

b) Przygotowanie plików testowych

```
.section .text
        .align 4
        .global _start
_start:
        # We expect 0 as result of every operation on x0
        auipc x0, 0xFFFFF
    # Max positive value
        auipc x1, 0xFFFFF
    # Positive value
        auipc x2, 0xF1D6C
        auipc x3, 0x00200
        auipc x4, 0x0125D

```

Zaimplementowane testy znajdują się w plikach
z rozszerzeniem `.s`

Funkcje po deasemblacji:

```
00000000 <_start>:
   0:   fff7d017                auipc   zero,0xfff7d
   4:   00001097                auipc   ra,0x1
   8:   00002117                auipc   sp,0x2
   c:   fffff197                auipc   gp,0xfffff
  10:   0125d217                auipc   tp,0x125d
  14:   00212297                auipc   t0,0x212
  18:   00011e17                auipc   t3,0x11
  1c:   fffffe97                auipc   t4,0xfffff
  20:   12122f17                auipc   t5,0x12122
  24:   00000f97                auipc   t6,0x0
        ...

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0:   1941                    addi    s2,s2,-16
   2:   0000                    unimp
   4:   7200                    flw     fs0,32(a2)
   6:   7369                    lui     t1,0xffffa
   8:   01007663                bgeu    zero,a6,14 <_start+0x14>
   c:   0000000f                fence   unknown,unknown
  10:   7205                    lui     tp,0xfffe1
  12:   3376                    fld     ft6,376(sp)
  14:   6932                    flw     fs2,12(sp)
  16:   7032                    flw     ft0,44(sp)
  18:   0030                    addi    a2,sp,8
```

c) Komplilacja i wykonanie programu

Kompilacja powiodła się, po uruchomieniu otrzymujemy:

```
jszyman:glowny$ ./syko.out
T: 0xfff7d017
0x0000: AUIPC R0, R1048445
T: 0x00001097
0x0004: AUIPC R1, R1
T: 0x00002117
0x0008: AUIPC R2, R2
T: 0xfffff197
0x000c: AUIPC R3, R1048575
T: 0x0125d217
0x0010: AUIPC R4, R4701
T: 0x00212297
0x0014: AUIPC R5, R530
T: 0x00011e17
0x0018: AUIPC R28, R17
T: 0xfffffe97
0x001c: AUIPC R29, R1048575
T: 0x12122f17
0x0020: AUIPC R30, R74018
T: 0x00000f97
0x0024: AUIPC R31, R0
T: 0x00000000
Wykryto nieznana instrukcje (PC=0x00000028, T=0x00000000)
```

Efekt działania wygląda poprawnie

d) Sprawdzenie działania procesora

W celu sprawdzenia czy procesor poprawnie wykonuje zadane instrukcje został otworzony plik `file_reg_out.bin` komendą *hexdump*:

```
jszyman:test$ hexdump file_reg_out.bin
0000000 0000 0000 1004 0000 2008 0000 f00c ffff
0000010 d010 0125 2014 0021 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
*
0000070 1018 0001 f01c ffff 2020 1212 0024 0000
0000080 0028 0000
0000084
```

Rejestr zerowy zawiera same zera, co jest zgodne z oczekiwaniami. Dane zawarte w kolejnych rejestrach pokrywają się z przewidywanym działaniem testów.

3.4 Funkcja JALR

a) Implementacja

Funkcja JALR (Jump and Link Register) wykonuje instrukcję typu J, bezwarunkowy skok pośredni, gdzie:

- IMM jest tylko 12-bitowy ze znakiem

- adres skoku wyznacza: RS1 + IMM

Instrukcja: `JALR RD, RS1, IMM12`:

```c
#include <stdio.h>

#include "types.h"
#include "mem_abs.h"

void F_JALR(void)
{
    writeOpcode();

    // 5-bit; 7-11
    DataType RD = getRD();
    // 5-bit; 15-19
    DataType RS1 = getRS1();
    // 12-bit; 20-31
    DataType IMM12=(getMEMC(getPC()) & 0xFFF00000) >>20;

    setRegister(RD, getPC() + 4);
    checkR0(RD);
   // setPC(IMM_WORD_ALIGNMENT(IMM12) + getRegister(RS1));

    printf("0x%04x: JALR R%d, IMM12%d(R%d)\n", getPC(), RD, IMM12, RS1);
}

```

b) Kompilacja i komentarz

Funkcja kompiluje się, niestety nie udało nam się wykonać testów

W celu sprawdzenia czy procesor poprawnie wykonuje zadane instrukcje został otworzony plik **`file_reg_out.bin`** komendą *hexdump*:

3.5 Funkcja SB

a) Implementacja

Funkcja SB (Store Binary) wykonuje instrukcję typu S - zapisuje w pamięci spod rejestru RS1 wartość 8-bitową.

Instrukcja: `SB RS1, RS2, IMM12`:

f_sb.c 〉 No Selection

```c
#include <stdio.h>

#include "types.h"
#include "mem_abs.h"

void F_SB(void)
{
//git   writeOpcode();
    // 5-bit; 15-19
    DataType RS1 = getRS1(); //stary zapis JonczykPRO (getMEMC(getPC()) & 0x7C000) >>15;
    // 5-bit; 20-24
    DataType RS2 = getRS2(); //stary zapis JonczykPRO (getMEMC(getPC()) & 0xF80000) >>20;
    // 5-bit IMM5 + 7-bit IMM7
    DataType IMM12 = ((getMEMC(getPC()) & 0x7C0) >> 7) + ((getMEMC(getPC()) & 0xFE000000) >> 25);
    printf("0x%04x: SB R%d, IMM12%d(R%d)\n", getPC(), RS2, IMM12, RS1);

    setRegister(RS2, getRegister((int8_t)IMM12 + (RS1)));
    // IMM12 jako offset - adres RS2+IMM
    checkR0(RS2);
    incPC();
}

```

b) Przygotowanie plików testowych

sb.s 〉 No Selection

```s
.section .text
         .align 4
         .global _start
_start:
        li x1, 0x111
        li x2, 0x222
        li x3, 0x333
        li x4, 0x444
        sb x0, 1(x2) # test0
        sb x5, 1(x1)
        li x10, 0x3DAB
        sb x11, 0(x10)
        sb x10, 0(x0)
        sb x30, 3(x1)
        sb x31, 10(x1)

```

Zaimplementowane testy znajdują się w plikach
z rozszerzeniem `.s`

Funkcje po deasemblacji:

```
Disassembly of section .text:

00000000 <_start>:
   0:   11100093                li      ra,273
   4:   22200113                li      sp,546
   8:   33300193                li      gp,819
   c:   44400213                li      tp,1092
  10:   000100a3                sb      zero,1(sp)
  14:   005080a3                sb      t0,1(ra)
  18:   00004537                lui     a0,0x4
  1c:   dab50513                addi    a0,a0,-597 # 3dab <_start+0x3dab>
  20:   00b50023                sb      a1,0(a0)
  24:   00a00023                sb      a0,0(zero) # 0 <_start>
  28:   01e081a3                sb      t5,3(ra)
  2c:   01f08523                sb      t6,10(ra)
  30:   0000                    unimp
        ...

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0:   1941                    addi    s2,s2,-16
   2:   0000                    unimp
   4:   7200                    flw     fs0,32(a2)
   6:   7369                    lui     t1,0xffffa
   8:   01007663                bgeu    zero,a6,14 <_start+0x14>
   c:   0000000f                fence   unknown,unknown
  10:   7205                    lui     tp,0xfffe1
  12:   3376                    fld     ft6,376(sp)
  14:   6932                    flw     fs2,12(sp)
  16:   7032                    flw     ft0,44(sp)
  18:   0030                    addi    a2,sp,8
```

c) Komplilacja i wykonanie programu

Kompilacja powiodła się, po uruchomieniu otrzymujemy:

```
jszyman:glowny$ ./syko.out
T: 0x11100093
0x0000: ADDI R1, R0, R273
T: 0x22200113
0x0004: ADDI R2, R0, R546
T: 0x33300193
0x0008: ADDI R3, R0, R819
T: 0x44400213
0x000c: ADDI R4, R0, R1092
T: 0x000100a3
0x0010: SB R0, IMM121(R2)
T: 0x005080a3
0x0014: SB R5, IMM121(R1)
T: 0x00004537
0x0018: LUI R10, R4
T: 0xdab50513
0x001c: ADDI R10, R10, R3499
T: 0x00b50023
0x0020: SB R11, IMM120(R10)
T: 0x00a00023
0x0024: SB R10, IMM120(R0)
T: 0x01e081a3
0x0028: SB R30, IMM123(R1)
T: 0x01f08523
0x002c: SB R31, IMM1210(R1)
T: 0x00000000
Wykryto nieznana instrukcje (PC=0x00000030, T=0x00000000)
jszyman:glowny$
```

Efekt działania wygląda poprawnie

d) Sprawdzenie działania procesora

W celu sprawdzenia czy procesor poprawnie wykonuje zadane instrukcje został otworzony plik `file_reg_out.bin` komendą *hexdump*:

```
jszyman:test$ hexdump file_reg_out.bin
0000000 0000 0000 0111 0000 0222 0000 0333 0000
0000010 0444 0000 0222 0000 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 4dab 0000
0000030 0000 0000 0000 0000 0000 0000 0000 0000
*
0000070 0000 0000 0000 0000 0444 0000 4dab 0000
0000080 0030 0000
0000084
jszyman:test$
```

Rejestr zerowy zawiera same zera, co jest zgodne z oczekiwaniami.
Funkcja zwraca poprawne wyniki

3.6 Funkcja SW

a) Implementacja

Funcja SW (Store Word) wykonuje instrukcję typu S - zapisuje w pamięci pod adres rejestru RS2+IMM wartość 32 bitową z RS1.

Instrukcja: `SW RS1, RS2, IMM17`:

```c
#include <stdio.h>

#include "types.h"
#include "mem_abs.h"

void F_SW(void){
    //writeOpcode();

    DataType RS1 = getRS1();
    DataType RS2 = getRS2();
    DataType IMM17 =((getMEMC(getPC()) & 0xF80) >> 7) + ((getMEMC(getPC()) & 0xFE000000) >> 20);

    // store value from RS1 to (R2 + IMM)
    setRegister((RS1) + (int32_t)IMM17, getRegister(RS2));

    printf("0x%04x: SW R%d, R%d, R%d\n", getPC(), RS1, RS2, IMM17);
    checkR0((RS1) + (int32_t)IMM17);
    // Increment Program Counter
    incPC();
}
```

b) Przygotowanie plików testowych

```
.section .text
        .align 4
        .global _start
_start:
        li x1, 0x001
        li x2, 0x002
        li x3, 0x003
        li x4, 0x004
        sw x1, 0(x0) #test Zero
        sw x0, 0(x2)
        sw x1, 4(x3)
        sw x2, 1(x5)
        sw x4, 28(x2)
        li x5, 0XFFFFFFF
        sw x5, 10(x5)
        sw x3, 31(x3)
```

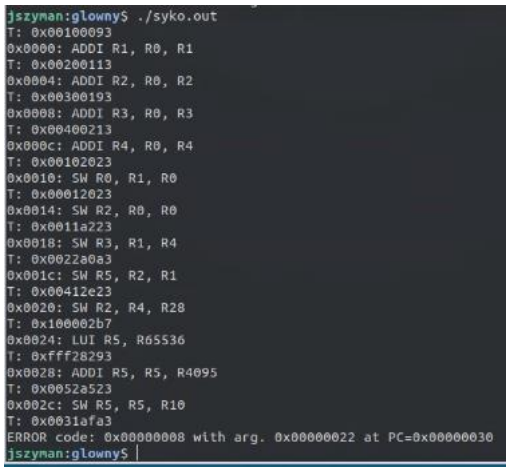
Zaimplementowane testy znajdują się w plikach
z rozszerzeniem `.s`

Funkcje po deasemblacji:

```
Disassembly of section .text:

00000000 <_start>:
   0: 00100093           li  ra,1
   4: 00200113           li  sp,2
   8: 12def1b7           lui gp,0x12def
   c: 00118193           addi  gp,gp,1 # 12def001 <_start+0x12def001>
  10: efff2237           lui tp,0xefff2
  14: 00f20213           addi  tp,tp,15 # efff200f <_start+0xefff200f>
  18: 00012023           sw  zero,0(sp)
  1c: 0050a023           sw  t0,0(ra)
  20: 00612023           sw  t1,0(sp)
  24: 0071a023           sw  t2,0(gp)
  28: 00822023           sw  s0,0(tp) # 0 <_start>
  2c: 0090a0a3           sw  s1,1(ra)
  30: 00a1a1a3           sw  a0,3(gp)
  34: 012222a3           sw  s2,5(tp) # 5 <_start+0x5>
  38: 01d22123           sw  t4,2(tp) # 2 <_start+0x2>
  3c: 01f120a3           sw  t6,1(sp)
  40: 0000                 unimp
  ...

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0: 1941                 addi  s2,s2,-16
   2: 0000                 unimp
   4: 7200                 flw fs0,32(a2)
   6: 7369                 lui t1,0xffffa
   8: 01007663           bgeu  zero,a6,14 <_start+0x14>
   c: 0000000f           fence unknown,unknown
  10: 7205                 lui tp,0xfffe1
  12: 3376                 fld ft6,376(sp)
  14: 6932                 flw fs2,12(sp)
  16: 7032                 flw ft0,44(sp)
  18: 0030                 addi  a2,sp,8
```

c) Komplilacja i wykonanie programu

Kompilacja powiodła się, po uruchomieniu otrzymujemy:



Efekt działania wygląda poprawnie

d) Sprawdzenie działania procesora

W celu sprawdzenia czy procesor poprawnie wykonuje zadane instrukcje został otworzony plik `file_reg_out.bin` komendą *hexdump*:

```
jszyman:test$ hexdump file_reg_out.bin
0000000 0000 0000 0001 0000 0001 0000 0003 0000
0000010 0004 0000 0fff 1000 0001 0000 0001 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0000 0000 0000 0fff 1000
0000040 0000 0000 0000 0000 0000 0000 0000 0000
*
0000070 0000 0000 0000 0000 0004 0000 0000 0000
0000080 0030 0000
0000084
jszyman:test$
```

Rejestr zerowy zawiera same zera, co jest zgodne z oczekiwaniami.
Instrukcja `sw x1, 4(x3)` zapisuje wartość z x1 (czyli 1) pod rejestr x3+4, czyli x7.

```
.section .text
        .align 4
        .global _start
_start:
        li x1, 0x001
        li x2, 0x002
        li x3, 0x003
        li x4, 0x004
        sw x1, 0(x0) #test Zero
        sw x0, 0(x2)
        sw x1, 4(x3)
        sw x2, 1(x5)
        sw x4, 28(x2)
        li x5, 0XFFFFFFFF
        sw x5, 10(x5)
        sw x3, 31(x3)
```

3.7 Funkcja SLTI

a) Implementacja

Funkcja SLTI (Set Less than Immediate) wykonuje instrukcję typu I – wstawia wartość 1 do RD jeżeli RS1 jest większy od IMM, w przeciwnym przypadku wstawia 0 pod rejestr RD.

Instrukcja: `SLTI RD, RS1, IMM12`:

```c
#include <stdio.h>

#include "types.h"
#include "mem_abs.h"

void F_SLTI(void){
    //writeOpcode();
    DataType RS1 = getRS1();
    DataType IMM12=(getMEMC(getPC()) & 0xFFF00000) >>20;
    DataType RD = getRD();

    printf("0x%04x: SLTI R%d, R%d, R%d\n", getPC(), RD, RS1, IMM12);

    if(getRegister(RS1) > IMM12){
        setRegister(RD, 0x00000001);
    }   else{
        setRegister(RD, 0x00000000);
    }
    checkR0(RD);
    incPC();
}
```

b) Przygotowanie plików testowych

```
.section .text
        .align 4
        .global _start
_start:
        slti x0, x1, 1
        li x1, 0x02
        slti x2, x1, 0x01
        slti x3, x1, 0x03
        li x4, 0x1F
        li x5, 0x00
        slti x6, x4, 0x1F
        slti x7, x4, 0x00
        slti x8, x5, 0x1F
        slti x9, x5, 0x00
        li x25, 0xEFFFFFFF
        li x26, 0xFFFFFFFF
        slti x27, x25, 0xFFFFFFFF
        slti x28, x25, 0x00
        slti x29, x26, 0x011
        slti x30, x26, 0x00
        slti x31, x26, 0xFFFFFFFF

```

Zaimplementowane testy znajdują się w plikach
z rozszerzeniem `.s`

Funkcje po deasemblacji:

```
00000000 <_start>:
   0:   0010a013                slti    zero,ra,1
   4:   00200093                li      ra,2
   8:   0010a113                slti    sp,ra,1
   c:   0030a193                slti    gp,ra,3
  10:   01f00213                li      tp,31
  14:   00000293                li      t0,0
  18:   01f22313                slti    t1,tp,31
  1c:   00022393                slti    t2,tp,0
  20:   01f2a413                slti    s0,t0,31
  24:   0002a493                slti    s1,t0,0
  28:   f0000cb7                lui     s9,0xf0000
  2c:   fffc8c93                addi    s9,s9,-1 # effffff <_start+0xefffffff>
  30:   fff00d13                li      s10,-1
  34:   fffcad93                slti    s11,s9,-1
  38:   000cae13                slti    t3,s9,0
  3c:   011d2e93                slti    t4,s10,17
  40:   000d2f13                slti    t5,s10,0
  44:   fffd2f93                slti    t6,s10,-1
        ...

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0:   1941                    addi    s2,s2,-16
   2:   0000                    unimp
   4:   7200                    flw     fs0,32(a2)
   6:   7369                    lui     t1,0xffffa
   8:   01007663                bgeu    zero,a6,14 <_start+0x14>
   c:   0000000f                fence   unknown,unknown
  10:   7205                    lui     tp,0xfffe1
  12:   3376                    fld     ft6,376(sp)
  14:   6932                    flw     fs2,12(sp)
  16:   7032                    flw     ft0,44(sp)
  18:   0030                    addi    a2,sp,8

```

c) Komplilacja i wykonanie programu

Kompilacja powiodła się, po uruchomieniu otrzymujemy:

```
jszyman:glowny$ ./syko.out
T: 0x0010a013
0x0000: SLTI R0, R1, R1
T: 0x00200093
0x0004: ADDI R1, R0, R2
T: 0x0010a113
0x0008: SLTI R2, R1, R1
T: 0x0030a193
0x000c: SLTI R3, R1, R3
T: 0x01f00213
0x0010: ADDI R4, R0, R31
T: 0x00000293
0x0014: ADDI R5, R0, R0
T: 0x01f22313
0x0018: SLTI R6, R4, R31
T: 0x00022393
0x001c: SLTI R7, R4, R0
T: 0x01f2a413
0x0020: SLTI R8, R5, R31
T: 0x0002a493
0x0024: SLTI R9, R5, R0
T: 0xf0000cb7
0x0028: LUI R25, R983040
T: 0xfffc8c93
0x002c: ADDI R25, R25, R4095
T: 0xfff00d13
0x0030: ADDI R26, R0, R4095
T: 0xfffcad93
0x0034: SLTI R27, R25, R4095
T: 0x000cae13
0x0038: SLTI R28, R25, R0
T: 0x011d2e93
0x003c: SLTI R29, R26, R17
T: 0x000d2f13
0x0040: SLTI R30, R26, R0
T: 0xfffd2f93
0x0044: SLTI R31, R26, R4095
T: 0x00000000
Wykryto nieznana instrukcje (PC=0x00000048, T=0x00000000)
```

Efekt działania wygląda poprawnie

d) Sprawdzenie działania procesora

W celu sprawdzenia czy procesor poprawnie wykonuje zadane instrukcje został otworzony plik `file_reg_out.bin` komendą *hexdump*:

```
jszyman:test$ hexdump file_reg_out.bin
0000000 0000 0000 0002 0000 0001 0000 0000 0000
0000010 001f 0000 0000 0000 0000 0000 0001 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
*
0000060 0000 0000 0fff f000 0fff 0000 0001 0000
0000070 0001 0000 0001 0000 0001 0000 0000 0000
0000080 0048 0000
0000084
```

Rejestr zerowy zawiera same zera, co jest zgodne z oczekiwaniami. W pierwszym przypadku RS1=0 nie jest większy od IMM=1, więc wpisano ‘0’. W drugim przypisujemy RS1=2, jest to wartość większa od IMM=1 (bez zmian), więc w rejestrze RD znalazła się wartość ‘1’.

3.8 Funkcja LUI

a) Implementacja

LUI (Load Upper Immediate) jest to instrukcja typu U. W miejsce górnych 20 bitów RD (Destination Register) ładowane jest 20 górnych bitów z IMM (12 dolnych bitów ustawione są na zero). Instrukcja ta jest używana zazwyczaj, gdy rejestr musi być zapełniony jakąś dużą wartością. Rejestrem docelowym do którego ładujemy wartość może być każdy z rejestrów od x1 do x31. Rejestr x0 może być tylko źródłem wartości, nie można do niego nic wpisać.

RD = IMM[32:12] | zero[11:0]

Instrukcja: **`LUI RD, IMM`**:

```c
#include <stdio.h>

#include "types.h"
#include "mem_abs.h"

void F_LUI(void){
     //writeOpcode();

    DataType IMM12=(getMEMC(getPC()) & 0xFFFFF000);
    DataType RD = getRD();

    printf("0x%04x: LUI R%d, R%d\n", getPC(), RD, IMM12 >> 12);

    setRegister(RD, IMM12);
    checkR0(RD);
    incPC();                                          //zwiększenie licznika rozkazów
    // jest git MG

}
```

b) Przygotowanie plików testowych

lui.s 〉 No Selection

```
.section .text
         .align 4
         .global _start
_start:
         lui x0, 0x7F7DF
         lui x1, 0x00001
         lui x2, 0x00003
         lui x3, 0xFFFFF
         lui x4, 0x005
         lui x19, 0xADCFE
         lui x28, 0x00000
         lui x29, 0x11111
         lui x30, 0xF2F2F
         lui x31, 0XFFFFF
```

Kompilacja pliku .s oraz przygotowanie plików wejściowych przebiegło pomyślnie.

Testujemy działanie rejestrów skrajnych oraz co się wydarzy, gdy argumentem funkcji będzie maksymalna wartość.

Po wykonanej kompilacji pliku z kodem testowym sprawdzono plik test.lst w celu sprawdzenia statusu po deasemblacji i rodzaju procesora.

```
test:     file format elf32-littleriscv
test
architecture: riscv:rv32, flags 0x00000012:
EXEC_P, HAS_SYMS
start address 0x00000000

Program Header:
    LOAD off    0x00000060 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000034 memsz 0x00000034 flags r-x

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000034  00000000  00000000  00000060  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .riscv.attributes 0000001a  00000000  00000000  00000094  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    d  .text  00000000 .text
00000000 l    d  .riscv.attributes      00000000 .riscv.attributes
00000000 g       .text  00000000 _start
```

Jak widać architektura RISC-V zgadza się z projektem, jest to RV32I. Adres startowy PC to 0x00000000. Funkcje po deasemblacji:

```
Disassembly of section .text:

00000000 <_start>:
   0:   7f7df037                lui     zero,0x7f7df
   4:   000010b7                lui     ra,0x1
   8:   00003137                lui     sp,0x3
   c:   fffff1b7                lui     gp,0xfffff
  10:   00005237                lui     tp,0x5
  14:   adcfe9b7                lui     s3,0xadcfe
  18:   00000e37                lui     t3,0x0
  1c:   11111eb7                lui     t4,0x11111
  20:   f2f2ff37                lui     t5,0xf2f2f
  24:   ffffffb7                lui     t6,0xfffff
        ...

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0:   1941                    addi    s2,s2,-16
   2:   0000                    unimp
   4:   7200                    flw     fs0,32(a2)
   6:   7369                    lui     t1,0xffffa
   8:   01007663                bgeu    zero,a6,14 <_start+0x14>
   c:   0000000f                fence   unknown,unknown
  10:   7205                    lui     tp,0xfffe1
  12:   3376                    fld     ft6,376(sp)
  14:   6932                    flw     fs2,12(sp)
  16:   7032                    flw     ft0,44(sp)
  18:   0030                    addi    a2,sp,8
```

Skok do kolejnej instrukcji odbywa się co 4 bity, po deasemblacji można zauważyć, że rejestry x0, x1, … x31 zmieniły nazwy na aliasy, np. x0 – zero, x2 – ra (Return Address).

c) Kompilacja i wykonanie aplikacji.

Kompilacja programu udała się, wynik wykonania aplikacji:

```
jszyman:glowny$ ./syko.out
T: 0x7f7df037
0x0000: LUI R0, R522207
T: 0x000010b7
0x0004: LUI R1, R1
T: 0x00003137
0x0008: LUI R2, R3
T: 0xfffff1b7
0x000c: LUI R3, R1048575
T: 0x00005237
0x0010: LUI R4, R5
T: 0xadcfe9b7
0x0014: LUI R19, R711934
T: 0x00000e37
0x0018: LUI R28, R0
T: 0x11111eb7
0x001c: LUI R29, R69905
T: 0xf2f2ff37
0x0020: LUI R30, R995119
T: 0xffffffb7
0x0024: LUI R31, R1048575
T: 0x00000000
Wykryto nieznana instrukcje (PC=0x00000028, T=0x00000000)
```

Jak widać aplikacja zadziałała poprawnie i rejestry zostały wczytane zgodnie z napisanym kodem testowym.

d) Sprawdzenie działania procesora

W celu sprawdzenia czy procesor poprawnie wykonuje zadane instrukcje został otworzony plik file_reg_out.bin komendą hexdump.

```
jszyman:test$ hexdump file_reg_out.bin
0000000 0000 0000 1000 0000 3000 0000 f000 ffff
0000010 5000 0000 0000 0000 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
*
0000040 0000 0000 0000 0000 0000 0000 e000 adcf
0000050 0000 0000 0000 0000 0000 0000 0000 0000
*
0000070 0000 0000 1000 1111 f000 f2f2 f000 ffff
0000080 0028 0000
0000084
```

Jak widać w rejestrze zero nie mamy żadnych wartości – jest tylko zero, co jest zgodne z oczekiwaniami. Kolejne analizy rejestrów potwierdzają działanie tej funkcji w wirtualnym procesorze.

3.9 Funkcja BNE

a) Implementacja

BNE (Branch If Not Equal) jest to instrukcja typu B. Zawartość rejestru RS1 i rejestru RS2 są ze sobą porównywane. Jeżeli są różne to wykonywany jest skok w nowe miejsce – nowe miejsce w kodzie wyznacza PC + rozszerzone_ze_znakiem(IMM 13bitowe). Zerowy bit imm jest domyślnie ustawiany na wartość 0.

IMM[0] = 0

Gdy warunek „nierówności” RS1 i RS2 nie jest spełniony to program normalnie przechodzi do kolejnego PC = PC + 4.

Instrukcja: **`BNE RS1, RS2, IMM`:**

f_bne.c 〉 No Selection

```c
#include <stdio.h>

#include "types.h"
#include "mem_abs.h"


DataType IMM_WORD_ALIGNMENT_BNE(DataType BEFORE_WA){  //funkcja do rozszerzenia ze znakiem

 if(((BEFORE_WA & 0x1000) >> 12) == 1) //jeli najstarszy bit z 13 bitowego wyrazenia IMM jest jedynka to powiela jedynke, jesli jest zerem to powiela zero od 13 do 31 bitu
     return 0xFFFFE000 + BEFORE_WA; // 0xFFFFE000 to 19 jedynek i 13 zer + BEFORE_WA - to wartosci pierwszych 13 bitow
 else return BEFORE_WA;
}

void F_BNE(void){
    writeOpcode();

    DataType RS1 = getRS1();
    DataType RS2 = getRS2();
    DataType IMM_B_type = ((getMEMC(getPC()) & 0xF00) >> 7) + ((getMEMC(getPC()) & 0x7E000000) >> 20 ) +
        ((getMEMC(getPC()) & 0x80) << 4 ) + ((getMEMC(getPC()) & 0x80000000) >> 19 );

    //pojedyncze przesuniecia
    //imm[0]=0 domyslnie bit zerowy jest zerem - zatem imm ma 13 bitow
    //DataType IMM_B_type_4_1 = ((getMEMC(getPC()) & 0xF00) >> 7) ;
    //DataType IMM_B_type_10_5 =((getMEMC(getPC()) & 0x7E000000) >> 20 );
    //DataType IMM_B_type_11 =((getMEMC(getPC()) & 0x80) << 4 );
    //DataType IMM_B_type_12 =((getMEMC(getPC()) & 0x80000000) >> 19 );

    if (getRegister(RS1) != getRegister(RS2)){
        setPC(IMM_WORD_ALIGNMENT_BNE(IMM_B_type) + getPC());
    } else {
        incPC();                                        //zwiększenie licznika rozkazów
    }

}

```

b) Przygotowanie plików testowych

bne.s 〉 No Selection

```
1  .section .text
2           .align 4
3           .global _start
4  _start:
5          li x1, 0x01
6          li x2, 0x02
7          li x3, 0xFF
8          li x4, 0xFF
9          bne x1, x2, _funct1
10         nop
11 _funct1:
12         nop
13         addi x1, x1, 0x02
14         nop
15         bne x3, x4, _funct2
16         bne x1, x3, _funct3
17 _funct2:
18         li x5, 0x2
19 _funct3:
20         li x5, 0x1
21         li x29, -0x800
22         li x30, 0xFFF
23         nop
24         bne x30, x29, _funct4
25 _funct4:
26         li x31, 0x01
27
```

Kompilacja pliku .s oraz przygotowanie plików wejściowych przebiegło pomyślnie.

Po wykonanej kompilacji pliku z kodem testowym sprawdzono plik test.lst w celu sprawdzenia statusu po deasemblacji i rodzaju procesora.

bne — nano test.lst — 114×43

```
 GNU nano 2.0.6                          File: test.lst

test:     file format elf32-littleriscv
test
architecture: riscv:rv32, flags 0x00000012:
EXEC_P, HAS_SYMS
start address 0x00000000

Program Header:
    LOAD off    0x00000060 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000054 memsz 0x00000054 flags r-x

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000054  00000000  00000000  00000060  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .riscv.attributes 0000001a  00000000  00000000  000000b4  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    d  .text  00000000 .text
00000000 l    d  .riscv.attributes      00000000 .riscv.attributes
00000000 l    df *ABS*  00000000 bne.o
00000018 l       .text  00000000 _funct1
0000002c l       .text  00000000 _funct2
00000030 l       .text  00000000 _funct3
00000048 l       .text  00000000 _funct4
00000000 g       .text  00000000 _start
```

Jak widać architektura RISC-V zgadza się z projektem, jest to RV32I. Adres startowy PC to 0x00000000. Funkcje po deasemblacji:

```
Disassembly of section .text:

00000000 <_start>:
   0:   00100093                li      ra,1
   4:   00200113                li      sp,2
   8:   0ff00193                li      gp,255
   c:   0ff00213                li      tp,255
  10:   00209463                bne     ra,sp,18 <_funct1>
  14:   00000013                nop

00000018 <_funct1>:
  18:   00000013                nop
  1c:   00208093                addi    ra,ra,2
  20:   00000013                nop
  24:   00419463                bne     gp,tp,2c <_funct2>
  28:   00309463                bne     ra,gp,30 <_funct3>

0000002c <_funct2>:
  2c:   00200293                li      t0,2

00000030 <_funct3>:
  30:   00100293                li      t0,1
  34:   80000e93                li      t4,-2048
  38:   00001f37                lui     t5,0x1
  3c:   ffff0f13                addi    t5,t5,-1 # fff <_funct4+0xfb7>
  40:   00000013                nop
  44:   01df1263                bne     t5,t4,48 <_funct4>

00000048 <_funct4>:
  48:   00100f93                li      t6,1
        ...

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0:   1941                    addi    s2,s2,-16
   2:   0000                    unimp
   4:   7200                    flw     fs0,32(a2)
   6:   7369                    lui     t1,0xffffa
   8:   01007663                bgeu    zero,a6,14 <_start+0x14>
   c:   0000000f                fence   unknown,unknown
  10:   7205                    lui     tp,0xfffe1
  12:   3376                    fld     ft6,376(sp)
  14:   6932                    flw     fs2,12(sp)
  16:   7032                    flw     ft0,44(sp)
  18:   0030                    addi    a2,sp,8
```

Skok do kolejnej instrukcji odbywa się co 4 bity, po deasemblacji można zauważyć, że rejestry x0, x1, … x31 zmieniły nazwy na aliasy, np. x0 – zero, x2 – RA (Return Address).

3.10 Funkcja SLL

a) Implementacja

SLL (Shift Logical Left) jest to funkcja typu R. Jest to operacja logicznego przesunięcia w lewo wartości z rejestru RS1. Liczba przesunięć jest określona w rejestrze RS2. Wynik przesunięcia zapisywany jest do RD. Młodsze bity przy przesunięciu zapełniane są zerami.

Instrukcja: `SLL RD, RS1, RS2`

f_sll.c 〉 No Selection

```c
#include <stdio.h>

#include "types.h"
#include "mem_abs.h"

void F_SLL(void){
//  writeOpcode();
    DataType RS1 = getRS1();
    DataType RS2 = getRS2();
    DataType RD = getRD();

    printf("0x%04x: SLL R%d, R%d, R%d\n", getPC(), RD, RS1, RS2);

    setRegister(RD, (getRegister(RS1) << getRegister(RS2)));    //właściwe obliczenie
    checkR0(RD);
    incPC();                                                    //zwiększenie licznika rozkazów
}
```

b) Przygotowanie plików testowych

sll.s 〉 No Selection

```s
.section .text
         .align 4
         .global _start
_start:
        li x1, 0x001
        li x2, 0x002
        sll x0, x2, x1
        li x3, 0x010
        sll x4, x3, x1
        li x5, 0xFFF
        li x6, 0x12D
        sll x7, x5, x1
        sll x8, x6, x2
        sll x9, x2, x6
        li x14, 0x123
        sll x28, x1, x5
        sll x29, x14, x1
        sll x30, x13, x5
        sll x31, x5, x5
```

Kompilacja pliku .s oraz przygotowanie plików wejściowych przebiegło pomyślnie.

Po wykonanej kompilacji pliku z kodem testowym sprawdzono plik test.lst w celu sprawdzenia statusu po deasemblacji i rodzaju procesora.

```
test:     file format elf32-littleriscv
test
architecture: riscv:rv32, flags 0x00000012:
EXEC_P, HAS_SYMS
start address 0x00000000

Program Header:
    LOAD off    0x00000060 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000044 memsz 0x00000044 flags r-x

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000044  00000000  00000000  00000060  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .riscv.attributes 0000001a  00000000  00000000  000000a4  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    d  .text  00000000 .text
00000000 l    d  .riscv.attributes      00000000 .riscv.attributes
00000000 g       .text  00000000 _start
```

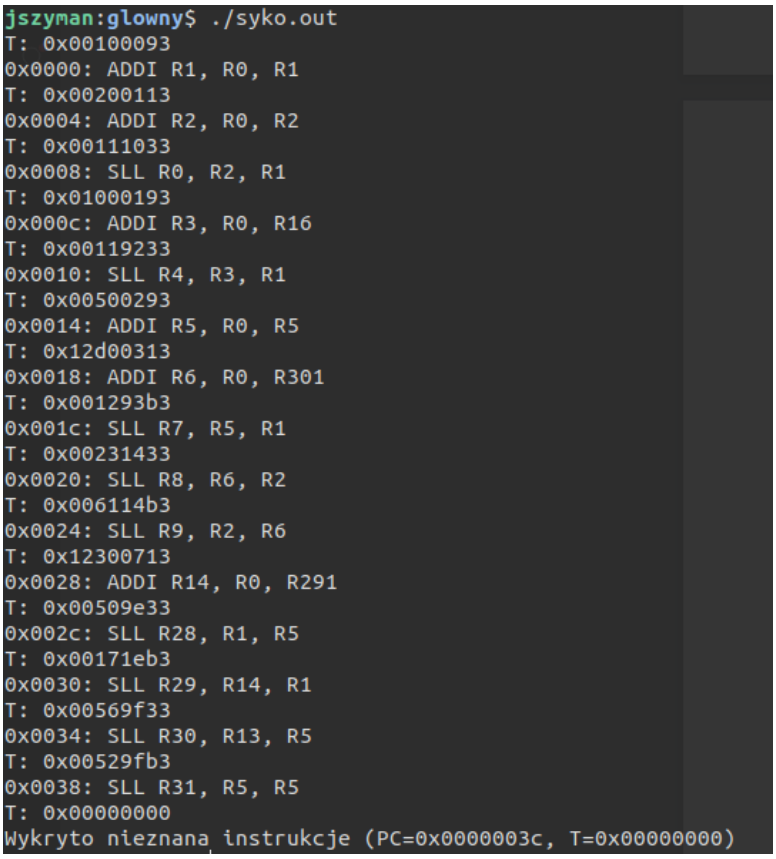
Jak widać architektura RISC-V zgadza się z projektem, jest to RV32I. Adres startowy PC to 0x00000000. Funkcje po deasemblacji:

```
00000000 <_start>:
   0:   00100093                li      ra,1
   4:   00200113                li      sp,2
   8:   00111033                sll     zero,sp,ra
   c:   01000193                li      gp,16
  10:   00119233                sll     tp,gp,ra
  14:   00500293                li      t0,5
  18:   12d00313                li      t1,301
  1c:   001293b3                sll     t2,t0,ra
  20:   00231433                sll     s0,t1,sp
  24:   006114b3                sll     s1,sp,t1
  28:   12300713                li      a4,291
  2c:   00509e33                sll     t3,ra,t0
  30:   00171eb3                sll     t4,a4,ra
  34:   00569f33                sll     t5,a3,t0
  38:   00529fb3                sll     t6,t0,t0
        ...

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0:   1941                    addi    s2,s2,-16
   2:   0000                    unimp
   4:   7200                    flw     fs0,32(a2)
   6:   7369                    lui     t1,0xffffa
   8:   01007663                bgeu    zero,a6,14 <_start+0x14>
   c:   0000000f                fence   unknown,unknown
  10:   7205                    lui     tp,0xfffe1
  12:   3376                    fld     ft6,376(sp)
  14:   6932                    flw     fs2,12(sp)
  16:   7032                    flw     ft0,44(sp)
  18:   0030                    addi    a2,sp,8
```

Skok do kolejnej instrukcji odbywa się co 4 bity, po deasemblacji można zauważyć, że rejestry x0, x1, … x31 zmieniły nazwy na aliasy, np. x0 – zero, x2 – RA (Return Address).

c) Kompilacja i wykonanie aplikacji.

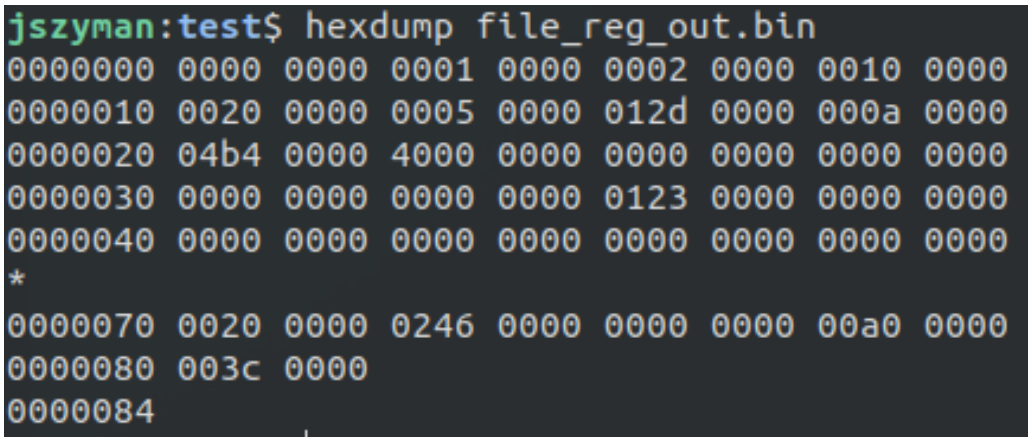
Kompilacja programu udała się, wynik wykonania aplikacji:



Jak widać aplikacja zadziałała poprawnie i rejestry zostały wczytane zgodnie z napisanym kodem testowym.

d) Sprawdzenie działania procesora

W celu sprawdzenia czy procesor poprawnie wykonuje zadane instrukcje został otworzony plik file_reg_out.bin komendą hexdump.



Jak widać w rejestrze zero nie mamy żadnych wartości – jest tylko zero, co jest zgodne z oczekiwaniami. Kolejne analizy rejestrów potwierdzają działanie tej funkcji w wirtualnym procesorze.

3.11 Funkcja JAL

a) Implementacja

JAL (Jump And Link) jest to instrukcja typu J. Są to funkcje sterowania transferem, czyli skoki. JAL to bezwarunkowy skok relatywny. Używany jest do wywołania podprogramu.

Instrukcja: **`JAL RD, offset`**

f_jal.c 〉 No Selection

```c
#include <stdio.h>

#include "types.h"
#include "mem_abs.h"

DataType IMM_WORD_ALIGNMENT_jal(DataType BEFORE_WA){  //funkcja do rozszerzenia ze znakiem

     if(((BEFORE_WA & 0x100000) >> 20) == 1) //jeli najstarszy bit z 21 bitowego wyrazenia IMM jest jedynka to
         powiela jedynke, jesli jest zerem to powiela zero od 20 do 32 bitu
         return 0xFFE00000 + BEFORE_WA; // 0xFFF00000 to 12 jedynek i 20 zer + BEFORE_WA - to wartosci
             pierwszych 20 bitow
     else return BEFORE_WA;
}

void F_JAL(void){
    writeOpcode();

    DataType RD = getRD();

    //składowe IMM - bit zerowy IMM jest domniemany i zostawiamy tam wartosc 0
    //DataType IMM_10_1 =  (getMEMC(getPC()) & 0x7FE00000) >> 20;
    //DataType IMM_11_ =  (getMEMC(getPC()) & 0x100000) >> 9;
    //DataType IMM_19_12 =  (getMEMC(getPC()) & 0xFF000);
    //DataType IMM_20_ =  (getMEMC(getPC()) & 0x80000000) >> 11;

    DataType IMM20 = ((getMEMC(getPC()) & 0x7FE00000) >> 20) + ((getMEMC(getPC()) & 0x100000) >> 9) +
        (getMEMC(getPC()) & 0xFF000) + ((getMEMC(getPC()) & 0x80000000) >> 11);

    setRegister(RD, getPC() + 4);
    checkR0(RD);
    setPC(IMM_WORD_ALIGNMENT_jal(IMM20) + getPC());
    printf("wynik JAL   RD%d    IMM20%d",RD ,IMM20 );
}

```

b) Przygotowanie plików testowych

jal.s 〉 No Selection

```
1  .section .text
2           .align 4
3           .global _start
4   _start:
5           li x1, 0x0
6           jal x11, _funct2
7   _funct1:
8           li x2, 0x1
9           jal x12, _funct5
10  _funct2:
11          li x3, 0x2
12          jal x13, _funct4
13  _funct3:
14          li x4, 0x3
15          jal x14, _funct1
16  _funct4:
17          li x5, 0x4
18          jal x15, _funct3
19  _funct5:
20          nop
21
```

Kompilacja pliku .s oraz przygotowanie plików wejściowych przebiegło pomyślnie.

Po wykonanej kompilacji pliku z kodem testowym sprawdzono plik test.lst w celu sprawdzenia statusu po deasemblacji i rodzaju procesora.

```
test:     file format elf32-littleriscv
test
architecture: riscv:rv32, flags 0x00000012:
EXEC_P, HAS_SYMS
start address 0x00000000

Program Header:
    LOAD off    0x00000060 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000034 memsz 0x00000034 flags r-x

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000034  00000000  00000000  00000060  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .riscv.attributes 0000001a  00000000  00000000  00000094  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    d  .text  00000000 .text
00000000 l    d  .riscv.attributes      00000000 .riscv.attributes
00000000 l    df *ABS*  00000000 jal.o
00000010 l       .text  00000000 _funct2
00000008 l       .text  00000000 _funct1
00000028 l       .text  00000000 _funct5
00000020 l       .text  00000000 _funct4
00000018 l       .text  00000000 _funct3
00000000 g       .text  00000000 _start
```

Jak widać architektura RISC-V zgadza się z projektem, jest to RV32I. Adres startowy PC to 0x00000000. Funkcje po deasemblacji:

```
Disassembly of section .text:

00000000 <_start>:
   0:   00000093                li      ra,0
   4:   00c005ef                jal     a1,10 <_funct2>

00000008 <_funct1>:
   8:   00100113                li      sp,1
   c:   01c0066f                jal     a2,28 <_funct5>

00000010 <_funct2>:
  10:   00200193                li      gp,2
  14:   00c006ef                jal     a3,20 <_funct4>

00000018 <_funct3>:
  18:   00300213                li      tp,3
  1c:   fedff76f                jal     a4,8 <_funct1>

00000020 <_funct4>:
  20:   00400293                li      t0,4
  24:   ff5ff7ef                jal     a5,18 <_funct3>

00000028 <_funct5>:
  28:   00000013                nop
        ...

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0:   1941                    addi    s2,s2,-16
   2:   0000                    unimp
   4:   7200                    flw     fs0,32(a2)
   6:   7369                    lui     t1,0xffffa
   8:   01007663                bgeu    zero,a6,14 <_funct2+0x4>
   c:   0000000f                fence   unknown,unknown
  10:   7205                    lui     tp,0xfffe1
  12:   3376                    fld     ft6,376(sp)
  14:   6932                    flw     fs2,12(sp)
  16:   7032                    flw     ft0,44(sp)
  18:   0030                    addi    a2,sp,8
```

Skok do kolejnej instrukcji odbywa się co 4 bity, po deasemblacji można zauważyć, że rejestry x0, x1, … x31 zmieniły nazwy na aliasy, np. x0 – zero, x2 – ra (Return Address).

c) Kompilacja i wykonanie aplikacji.

Kompilacja programu udała się, wynik wykonania aplikacji:

```
jszyman:glowny$ ./syko.out
T: 0x00000093
0x0000: ADDI R1, R0, R0
T: 0x00c005ef
wynik JAL   RD11    IMM2012T: 0x00200193
0x0010: ADDI R3, R0, R2
T: 0x00c006ef
wynik JAL   RD13    IMM2012T: 0x00400293
0x0020: ADDI R5, R0, R4
T: 0xff5ff7ef
wynik JAL   RD15    IMM202097140T: 0x00300213
0x0018: ADDI R4, R0, R3
T: 0xfedff76f
wynik JAL   RD14    IMM202097132T: 0x00100113
0x0008: ADDI R2, R0, R1
T: 0x01c0066f
wynik JAL   RD12    IMM2028T: 0x00000013
0x0028: ADDI R0, R0, R0
T: 0x00000000
Wykryto nieznana instrukcje (PC=0x0000002c, T=0x00000000)
jszyman:glowny$
```

Jak widać aplikacja zadziałała poprawnie i rejestry zostały wczytane zgodnie z napisanym kodem testowym.

d) Sprawdzenie działania procesora

W celu sprawdzenia czy procesor poprawnie wykonuje zadane instrukcje został otworzony plik file_reg_out.bin komendą hexdump.

```
jszyman:test$ hexdump file_reg_out.bin
0000000 0000 0000 0000 0000 0001 0000 0002 0000
0000010 0003 0000 0004 0000 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0008 0000
0000030 0010 0000 0018 0000 0020 0000 0028 0000
0000040 0000 0000 0000 0000 0000 0000 0000 0000
*
0000080 002c 0000
0000084
jszyman:test$
```

Jak widać w rejestrze zero nie mamy żadnych wartości – jest tylko zero, co jest zgodne z oczekiwaniami. Kolejne analizy rejestrów potwierdzają działanie tej funkcji w wirtualnym procesorze.