# Projekt SYKO

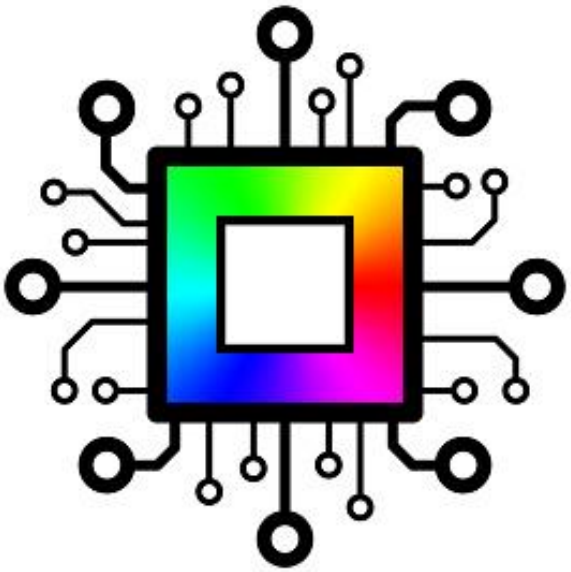
w Instytucie Systemów Elektronicznych

# System procesorowy

---

**Wykonawcy:** **Bartosz Ostrowski**
**Jacek Martyniak**

**Prowadzący:** **prof. nzw. dr hab. Jan Ogrodzki**



**Warszawa, 26 czerwiec 2020**

## 1. Założenia projektowe.

Naszym celem jest zaprojektowanie procesora opartego na architekturze akumulatorowej. Powinien on wykonywać instrukcje określone w treści zadania. Powinien zawierać również dwa rejestry (nie wliczając niezbędnych do działania procesora). Procesor musiał mieć zaimplementowaną obsługę kilku instrukcji z danymi trybami adresowania widocznymi poniżej.

| Instrukcja | Tryby adresowania |
|---|---|
| ADD | N/R/B/P |
| LOAD | N/R/B/P |
| JNOF | N/R |

Wykorzystane tryby adresowania:

- N - natychmiastowy
- R - rejestrowy
- B - bazowy
- P - przemieszczeniowy

Założenia czasowe procesora:

- f = 33.333MHz, zatem okres zegara równy 30 ns,
- opóźnienie pamięci MR 5ns,
- opóźnienie pozostałych bloków 3ns.

Projektowanie zaczynamy od stworzenie schematu blokowego, następnie realizujemy graf automatu sterującego jednostki sterującej. Mając powyższe informacje można przejść do pisania kodu poszczególnych komponentów. Należy zacząć od mniejszych komponentów, przewidując odpowiednie wejścia i wyjścia oraz sygnały sterujące, tak, aby potem była możliwość połączenia wszystkich komponentów w kompletny współpracujący system procesorowy.
Po przejściu testów poszczególnych komponentów można spróbować przetestować cały system zapisując instrukcje w pamięci i sprawdzając, czy operacje wykonują się prawidłowo.

## 2. Instrukcje procesora

| **Instrukcja** | LOAD - 00 | ADD - 01 | JNOF - 10 | - |
|---|---|---|---|---|
| **Numer rejestru** | 0-REG_1 | 1-REG_2 | - | - |
| **Tryb adresowania** | 00-Rejestrowy | 01-Bazowy | 10-Przemieszczeniowy | 11-instrukcja jednoargumentowa |

Kody instrukcji są 8-bitowe i opisane zgodnie z tabelą powyżej.
Poniżej znajduje się rozpisana tabela wszystkich możliwych do obsłużenia instrukcji wraz z odpowiadającymi im zdekodowanymi kodami dla jednostki sterującej.

| Instrukcje jednoargumentowe | | | | |
|---|---|---|---|---|
| **Instrukcja** | **Tryb adresowania** | **Rejestr** | **Kod zakodowany** | **Kod zdekodowany** |
| LOAD | rejestrowy | REG_1 | 00000000 | 00000 |
| LOAD | rejestrowy | REG_2 | 00000100 | 00100 |
| LOAD | bazowy | REG_1 | 00000001 | 01000 |
| LOAD | bazowy | REG_2 | 00000101 | 01100 |
| LOAD | przemieszczeniowy | - | 00000-10 | 10-00 |
| LOAD | natychmiastowy | - | 00000-11 | 11-00 |
| ADD | rejestrowy | REG_1 | 00001000 | 00001 |
| ADD | rejestrowy | REG_2 | 00001100 | 00101 |
| ADD | bazowy | REG_1 | 00001001 | 01001 |
| ADD | bazowy | REG_2 | 00001101 | 01101 |
| ADD | przemieszczeniowy | - | 00001-10 | 10-01 |
| ADD | natychmiastowy | - | 00001-11 | 11-01 |
| JNOF | rejestrowy | REG_1 | 00010000 | 00010 |
| JNOF | rejestrowy | REG_2 | 00010100 | 00110 |
| JNOF | bazowy | REG_1 | 00010001 | 01010 |
| JNOF | bazowy | REG_2 | 00010101 | 01110 |
| JNOF | przemieszczeniowy | - | 00010-10 | 10-10 |
| JNOF | natychmiastowy | - | 00010-11 | 11-10 |
| BŁĄD | | | INNY | 11111 |

## 3. Bloki funkcjonalne systemu.

| ELEMENT | SYGNAŁY STERUJĄCE | IN | OUT | BIDIRECTIONAL |
|---|---|---|---|---|
| **MAR** | clk, rst, lae | mar_in (8) | mar_out (8) | - |
| **MEMORY** | mw, mr, | address (8) | - | data (8) |
| **MBR** | clk,rst, re, we | - | - | mbr_data (8), mbr_mem (8) |
| **CU** | clk, RESET,<br>re, we, mw, rw,lae,<br>incr, jump, cag (3),<br><br>oe: Acc, buf, REG_1, REG_2, IR, flags, IMR<br><br>ie: Acc, buf, REG_1, REG_2, IR, IMR | flags (5),<br>ird (5) | start_adr (8),<br>increment (8) | - |
| **Clock** | | - | clk | - |
| **IR** | clk, rst, ie | ir_in (8) | ir_out (8) | - |
| **IR_DECODER** | | ir_out (8) | ird_out (5) | - |
| **AG** | cag (3) | imr (8),<br>reg1 (8),<br>reg2 (8),<br>pc (8) | ag_out (8) | pc_ag (8) |
| **PC** | clk, rst, incr, jump | jump_adr (8),<br>start_adr (8),<br>increment (8) | - | pc_ag (8) |
| **ALU** | - | x (8), y (8) | flags (5), z(8) | - |
| **buf** | clk, rst, ie, oe | buf_in (8) | buf_out (8) | - |
| **flags** | clk, rst | ie (5) | - | - |
| **REG_1, REG_2, IMR, ACC** | ie, oe, clk, rst | | reg_out | reg_io |

| Component | Signal | Typ |
|---|---|---|
| SYSTEM | rst, clk, RESET | std_logic |
| BUSES | addressBus | std_logic_vector(7 downto 0) |
| | dataBus | std_logic_vector(7 downto 0) |
| ALU | y, z | std_logic_vector(7 downto 0) |
| | flags | std_logic_vector(4 downto 0) |
| ACC | ie_ACC, oe_ACC | std_logic |
| buf | oe_buf, ie_buf | std_logic |
| MEMORY | lae, re_MBR, we_MBR, mw, mr | std_logic |
| | mar_mem, mem_mbr | std_logic_vector(7 downto 0) |
| AG | r1_ag, r2_ag, pc_ag, imr_ag | std_logic_vector(7 downto 0) |
| | cag | std_logic_vector(2 downto 0) |
| REG_1 | oe_REG_1, ie_REG_1 | std_logic |
| REG_2 | oe_REG_2, ie_REG_2 | std_logic |
| IR | oe_IR, ie_IR | std_logic |
| IR_DECODER | ir_ird | std_logic_vector(7 downto 0) |
| | ird_cu | std_logic_vector(4 downto 0) |
| IMR | oe_IMR, ie_IMR | std_logic |
| PC | increment, start_adr | std_logic_vector(7 downto 0) |
| | jump, incr | std_logic |

## 4. Implementacja w języku opisu sprzętu VHDL

Dalszym etapem projektu jest implementacja zaprojektowanej architektury w języku VHDL. Dokonany został widoczny niżej podział zadań.

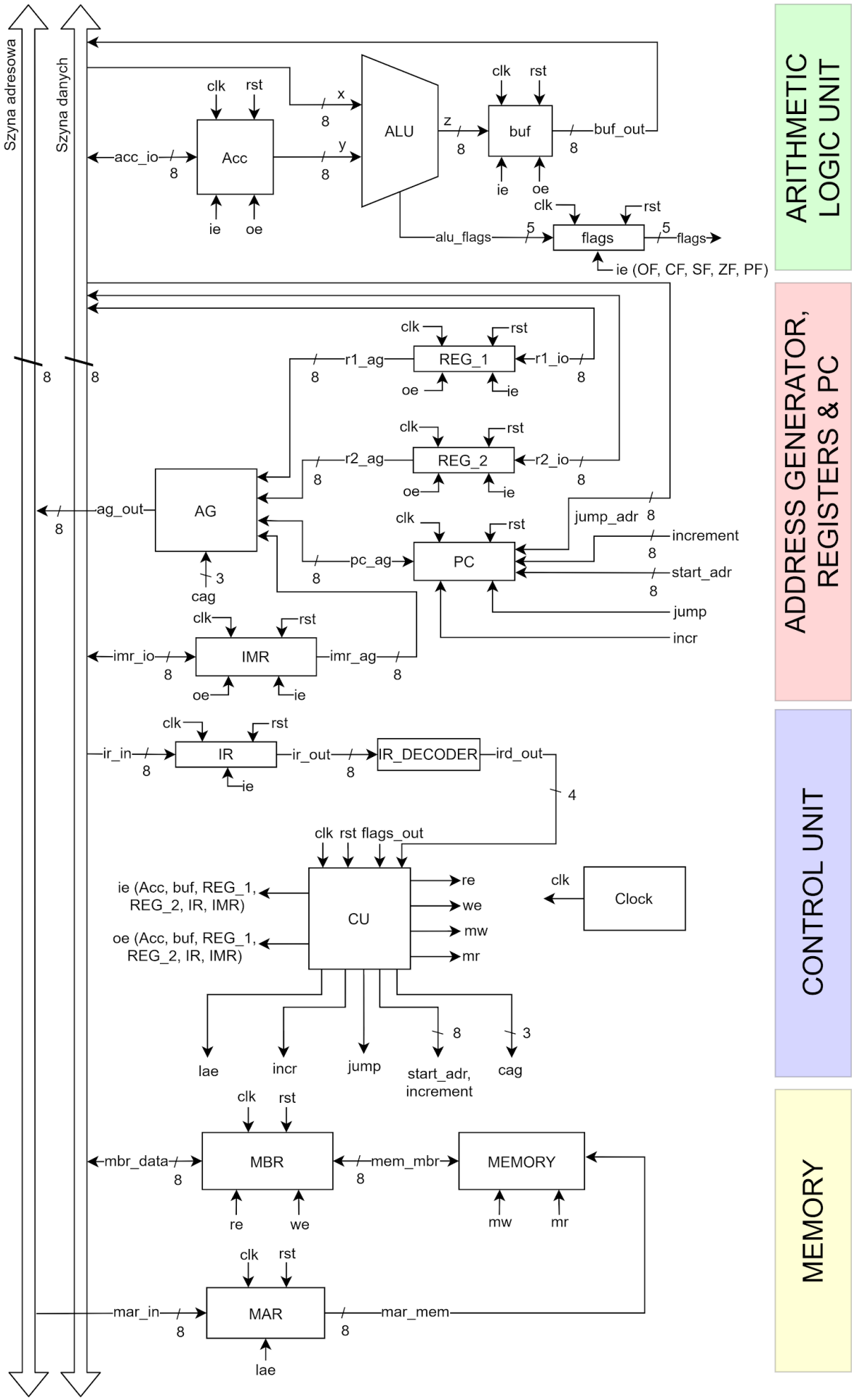
| Nr | Bartosz Ostrowski | Jacek Martyniak |
|---|---|---|
| 1. | ADDRESS GENERATOR, CLOCK & PC | ARITHMETIC LOGIC UNIT |
| 2. | REGISTERS | MEMORY |
| 3. | CONTROL UNIT, IR DECODER | |

## 4. Pliki zawierające implementacje komponentów.

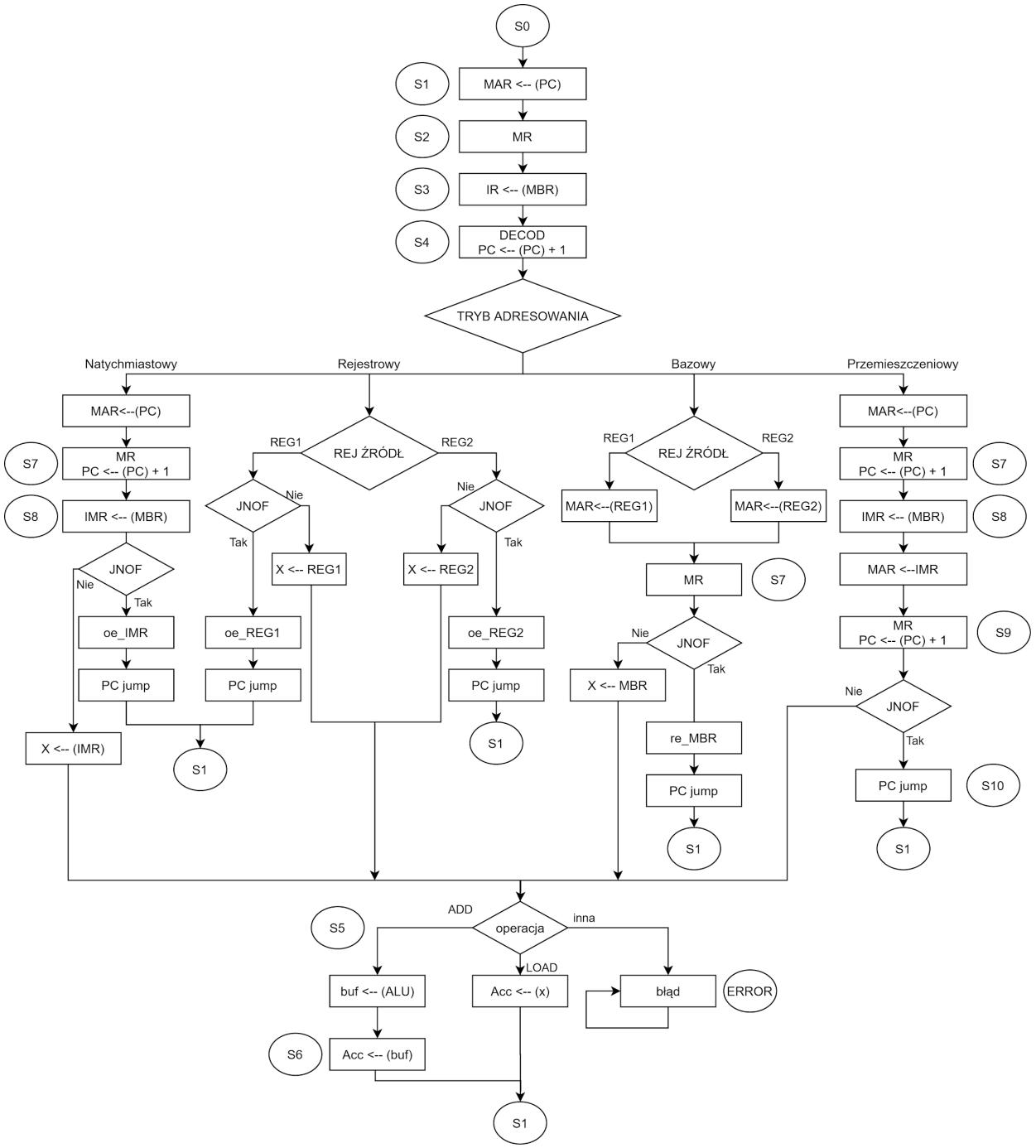
| Nazwa komponentu | Nazwa pliku |
|---|---|
| MAR | MAR.vhd |
| MEMORY | MEMORY.vhd |
| MBR | MBR.vhd |
| CU | CU.vhd |
| Clock | System.vhd |
| IR | IR.vhd |
| IR_DECODER | IR_DECODER.vhd |
| AG | AG.vhd |
| REG_1 | REG.vhd |
| REG_2 | REG.vhd |
| PC | PC.vhd |
| IMR | REG.vhd |
| Acc | REG.vhd |
| ALU | ALU.vhd |
| buf | buf.vhd |
| główny plik procesora | System.vhd |

## 5. Architektura systemu procesorowego.

Na następnej stronie widoczna jest architektura systemu procesorowego. Została ona wykonana przy użyciu internetowego narzędzia diagrams.net.


Szyna adresowa
Szyna danych
ARITHMETIC LOGIC UNIT
ADDRESS GENERATOR, REGISTERS & PC
CONTROL UNIT
MEMORY
clk
rst
Acc
acc_io
ie
oe
x
y
ALU
z
buf
buf_out
alu_flags
flags
ie (OF, CF, SF, ZF, PF)
REG_1
r1_ag
r1_io
REG_2
r2_ag
r2_io
AG
ag_out
cag
PC
pc_ag
jump_adr
increment
start_adr
jump
incr
IMR
imr_io
imr_ag
IR
ir_in
ir_out
IR_DECODER
ird_out
flags_out
CU
ie (Acc, buf, REG_1, REG_2, IR, IMR)
oe (Acc, buf, REG_1, REG_2, IR, IMR)
re
we
mw
mr
Clock
lae
incr
jump
start_adr, increment
cag
MBR
mbr_data
mem_mbr
MEMORY
MAR
mar_in
mar_mem

## 6. Graf automatu sterującego przepływem instrukcji




Po pojawieniu się sygnału RESET na wejściu CU, automat przechodzi do stanu początkowego S0.

# 7. Opis funkcjonalny bloków

## 7.1. Rejestry: REG_1, REG_2, Acc, IMR

Rejestry działają synchronicznie. Na zboczu narastającym zegara sprawdzany jest stan resetu. Gdy wynosi on 0, zawartość rejestru jest zerowana.
W przeciwnym wypadku rejestr sprawdza stan sygnału ie, czyli input enable. Gdy wynosi on 1, rejestr zapisuje dane znajdujące się w danym momencie na szynie danych. Jeśli wynosi 0, to dane w rejestrze pozostają niezmienione.
Podczas zbocza opadającego sprawdzane jest sygnał oe czyli output enable. Gdy wynosi 1, to na szynę danych ładowana jest zawartość rejestru. W przeciwnym przypadku wyjście reg_io przyjmuje wysoką impedancję. Na zboczu opadającym niezależnie od sygnałów sterujących na wyjście reg_out wystawiania jest zawartość rejestru. Wyjście to trafia na AG, aby wystawić adres z rejestru.

Kod architektury:

```
architecture arch of REG is
begin

	process (clk, ie, oe, rst)

	variable store : std_logic_vector (7 downto 0);
	begin
		if rising_edge(clk) then
			if rst='0' then
				store := (others=>'0');
			elsif ie='1' then
				store := reg_io;
			end if;
		elsif falling_edge(clk) then
			if oe='1' then
				reg_io <= store after delay;
			else
				reg_io <= (others=>'Z') after delay;
			end if;
				reg_out	<= store after delay;
		end if;
	end process;
end arch;
```

### 7.2. Pamięć: Memory

Pamięć jest elementem działającym asynchronicznie, tj. stany wystawiane są niezależnie od zboczy sygnału zegarowego. Posiada ona wejście adresowe oraz jedno wejście/wyjście danych (odczyt i zapis danych odbywa się poprzez ten sam port). Sterowanie odbywa się za pomocą dwóch sygnałów: mw (memory write) oraz mr (memory read). Pamięć jest reprezentowana jako zmienna - tablica bajtów. Zapis i odczyt danych następuje do zmiennej pod adresem podanym na wejście adresowe. W przypadku, gdy nie jest wystawiony żaden sygnał sterujący, wyjście danych przyjmuje wysoką impedancję. Adres wymaga konwersji z std_logic_vector do odpowiedniego formatu.
Widoczna jest zawartość pamięci z rozkazami i danymi, które wykorzystaliśmy podczas symulacji.

Kod architektury:

```
 architecture arch of Memory is

  type ROM_type is array (0 to 4 ) of std_logic_vector(7 downto 0);

 begin
      process(mw, mr)

       variable rom_data: ROM_type:=(
      "00000011", --LOAD N
      "00000001", --liczba
      "00001010", --ADD P
      "00000111", --liczba 7
      "00010000", --JNOF REG1
      "00000000", --EMPTY
      "00000000", --EMPTY
      "00000010"  --liczba 2
   );
      begin
         if mw='1' then
            rom_data(to_integer(unsigned(address))) := data;
         elsif mr='1' then
            data  <=  rom_data(to_integer(unsigned(address)))  after
delay;
         else
            data <= (others=>'Z') after delay;
         end if;

      end process;

 end arch;
```

### 7.3. Generator adresu: AG

Generator adresu jest multiplekserem wystawiającym adres na szynę adresową. Za pomocą sygnału sterującego *cag* wystawia on na wyjście adres ze wskazanego źródła: REG 1, REG 2, PC, IMR. Działanie AG jest asynchroniczne.

```
architecture arch of AG is

begin

 with cag select
       ag_out  <=  imr              after delay when "000",
                   reg1             after delay when "001",
                   reg2             after delay when "010",
                   pc               after delay when "011",
                   (others=>'Z')    after delay when others;
end arch;
```

### 7.4. Rejestr adresu:  MAR

Rejestr adresu jest synchronicznym komponentem służącym do przechowywania adresu pamięci dla memory. Jest podłączona do szyny adresowej.
Zbocze narastające - dla sygnału resetu (rst) równego 0, rejestr zostaje zerowany, w przeciwnym wypadku badany jest warunek sygnału sterującego *lae,* gdy wynosi on 1, następuje wczytanie do rejestru bajtu z szyny adresowej.
Zbocze opadające - na każdym zboczu opadającym, na wyjście MAR zostaje wystawiona zawartość tego rejestru.

Kod architektury:

```
 architecture arch of MAR is
 begin

       process (clk, rst, lae)

       variable store : std_logic_vector (ND downto 0);
       begin
             if rising_edge(clk) then
                   if rst='0' then
                         store := (others=>'0');
                   elsif lae='1' then
                         store := mar_in;
                   end if;
             elsif falling_edge(clk) then
                   mar_out <= store after delay;
             end if;

       end process;
 end arch;
```

### 7.5. Rejestr danych: MBR

Rejestr danych MBR jest rejestrem synchronicznym pośredniczącym w wymianie informacji między pamięcią, a szyną danych.
Zbocze narastające - dla sygnału resetu (rst) rejestr jest zerowany, następnie sprawdzane są kolejno warunki:

- re (read enable) - zapisywane do rejestru są dane wystawione przez pamięć,
- we (write enable) - zapisywane do rejestru są dane z szyny danych.

Zbocze opadające:

- re - na szynę danych wystawiane są dane z rejestru,
- we - do pamięci wystawiane są dane z rejestru,
- w przeciwnym przypadku zwalniana jest szyna danych (wysoka impedancja).

Kod architektury:

```
architecture arch of MBR is
begin

	process (clk, re, we, rst)

	variable store : std_logic_vector (7 downto 0);
	begin
		if rising_edge(clk) then
			if rst='0' then
				store := (others=>'0');
			elsif re='1' then
				store := mbr_mem;
			elsif we='1' then
				store := mbr_data;
			end if;

		elsif falling_edge(clk) then
			if re='1' then
				mbr_data <= store after delay;
			elsif we='1' then
				mbr_mem <= store after delay;
			else
				mbr_data <= (others=>'Z') after delay;
			end if;
		end if;

	end process;
end arch;
```

### 7.6. Rejestr instrukcji: IR

Rejestr instrukcji jest asynchronicznym rejestrem wykorzystywanym tylko do przechowywania instrukcji. Jako element przechowujący zawartość rejestru wykorzystana została zmienna.
Zbocze narastające, badanie warunków:

- rst = 0 - wykonanie czyszczenia rejestru
- ie = 1 (input enable) - zapisanie danych z wejścia ( szyny danych ) do rejestru.

Zbocze opadające:

- na wyjście ir_out wystawiana jest zawartość rejestru.

Kod architektury:

```
architecture arch of IR is
begin

 process (clk, ie, oe, rst)

 variable store : std_logic_vector (ND downto 0);
 begin
        if rising_edge(clk) then
               if rst='0' then
                      store := (others=>'0');
               elsif ie='1' then
                      store := ir_in;
               end if;
        elsif falling_edge(clk) then
               ir_out <= store after delay;
        end if;
 end process;
end arch;
```

### 7.7. Bufor - buf

Bufor jest rejestrem przechowującym wyniki działania ALU. Jest on niezbędny w architekturze akumulatorowej. Aby zapisać w miejscu docelowym wynik operacji ALU, należy go tymczasowo zapisać w buforze, aby zwolnić szybę danych, by móc następnie wystawić zawartość bufora na nią.
Podczas zbocza narastającego sprawdzane jest, czy sygnał rst = 0, wtedy następuje zerowanie bufora, dla sygnału ir (input enable) = 1 - wynik operacji ALU wpisywany jest do bufora.
Na zboczu opadającym sprawdzany jest sygnał oe (output enable), dla równego 1 - na szynę danych wystawiana jest zawartość bufora, w przeciwnym wypadku zwalniana jest szyna danych (stan wysokiej impedancji).
Kod architektury:

```
architecture arch of buf is
begin

	process (clk, ie, oe, rst)

	variable store : std_logic_vector (ND downto 0);
	begin
		if (rising_edge(clk)) then
			if rst='0' then
				store := (others=>'0');
		elsif ie='1' then
				store := buf_in;
			end if;
		elsif falling_edge(clk) then
			if oe='1' then
				buf_out <= store after delay;
			else
				buf_out <= (others=>'Z') after delay;
			end if;
		end if;

	end process;
end arch;
```

### 7.8. Licznik rozkazów - PC

Licznik rozkazów jest rejestrem procesora przechowującym informację o tym, w którym obecnie miejscu sekwencji instrukcji znajduje się procesor. Jest on rejestrem synchronicznym posiadającym specjalne wejścia:

Wejścia 8 - bitowe:

- start_adr - wejście adresu początkowego (po resecie)
- increment - wejście wskazujące o ile PC ma zwiększyć
- jump_adr

Wejścia sterujące:

- jump - wpisanie adresu skoku z wejścia jump_adr
- incr - inkrementacja
- rst - zapisanie do PC wartości początkowej

Zbocze narastające:

- dla rst = 0 - wpis adresu start_adr do PC i wystawienie go na wyjście.

Zbocze opadające:

- dla incr = 1 - inkrementacja rejestru wartością zadaną przez wejście increment,
- dla jump = 1 - wystawienie adresu skoku (jump_adr) na wyjście PC

Kod architektury:

```
architecture arch of PC is
begin
	process(clk, rst, incr, start_adr, jump)
		variable ch : std_logic_vector(ND downto 0);
		variable ces: std_logic;
	begin
		if (rst='0') then
			pc_io <= start_adr;
		elsif falling_edge(clk) and incr='1' then
			ces := '0';
			for i in 0 to ND loop
				ch(i) := (ces xor pc_io(i)) xor increment(i);
				ces := (pc_io(i) and increment(i)) or
					((pc_io(i) xor increment(i)) and ces);
			end loop;
			pc_io <= ch after delay;
		elsif falling_edge(clk) and jump='1' then
			pc_io <= jump_adr after delay;
		end if;
	end  process;
end arch;
```

### 7.9. Dekoder rozkazów - IR DECODER

Zadaniem dekodera rozkazów jest translacja instrukcji zakodowanych w pamięci na sygnały dla jednostki sterującej, tj. “wyciągnięcie” potrzebnych informacji o instrukcji: rozkazie, trybie adresowania oraz rejestru. Badany jest także przypadek inny, który jest rozpoznawany jako błąd.

Kod architektury:

```
architecture arch of IR_DECODER is
begin

--WDOBYWAMY:
--     tryb adresowania          "------**"
--     nr rejestry               "-----*--" R1-0, R2-1
--     instrukcja                "---**---" LOAD-00, ADD-01, JNOF-10

process(ird_in) is
begin

if ird_in(4 downto 3)="11" or (not ird_in(7 downto 6)="000") then
      ird_out <= (others => '1');     --ERROR
else
      ird_out <= ird_in(1 downto 0)&ird_in(3)&ird_in(4 downto 3);
end if;

end process;
end arch;
```

### 7.10. Jednostka arytmetyczno-logiczna - ALU

Jednostka arytmetyczno-logiczna służy do wykonywania operacji arytmetycznych i logicznych. W naszym przypadku jest wykorzystywana tylko do wykonywania operacji ADD, czyli dodawania arytmetycznego.

ALU posiada dwa wejścia x, y oraz wyjście wyniku obliczeń danej operacji - z.

Dodatkowo ALU dla każdej operacji wylicza flagi:

- OF - overflow (przepełnienie),
- CF - carry flag ( przeniesienie),
- SF - sign flag ( flaga znaku ),
- ZF - zero flag (flaga zer),
- PF - partity flag (flaga parzystości).

Kod architektury:

```
architecture arch of alu is
begin
process(x, y)
    variable result : std_logic_vector(ND downto 0);
    variable i : integer; --for FOR
    variable carry : std_logic;
    variable flags_buf : std_logic_vector(4 downto 0);
    variable parity_v : std_logic;

begin
    carry := '0';
    flags_buf:=(others=>'0');
    parity_v := '1';

    ADD_LOOP: for i in 0 to ND loop
        result(i) := (carry xor x(i)) xor y(i);
        carry := (x(i) and y(i)) or ((x(i) xor y(i)) and carry);
    end loop;

    --OF
    if (x(ND)=y(ND) and x(ND)/=result(ND)) then
        flags_buf(0) := '1';
    else
        flags_buf(0) := '0';
    end if;

    --CF
    flags_buf(1) := carry;

    --ZF
    flags_buf(3) := result(ND);

    --PF
    for i in 0 to ND loop
        parity_v := parity_v xor result(i);
    end loop;
    flags_buf(4) := parity_v;

    --flags_out
    flags <= flags_buf after delay;

    --OUT
    z <= result after delay;

end process;
end arch;
```

### 7.10 Jednostka sterująca - CU

Jednostka sterująca wysyła sygnały do poszczególnych bloków, by aktywować ich wyjścia lub wejścia. Dostaje ona zdekodowaną instrukcję, w której określona jest operacja, tryb adresowania oraz rejestr. Znajduje się w niej implementacja automatu stanów, które opisane są poniżej

## 8. Instrukcje

**S0:** Stan początkowy. Ten stan uaktywnia się po wysłaniu sygnału reset do jednostki sterującej. Wszystkie sygnały sterujące zostają wyłączone, a zawartości rejestrów wyczyszczone. Następnym stanem jest S1.

```
when s0 =>
        stateX <= 0;
        if r_e = '1' then
                start_adr <= "00000001";        --first instruction
                rst <= '0';                --wpisanie start adr
                cag <= "011";
                oe_ACC <='0';
                ie_ACC <='0';
                ie_REG_1 <= '0';
                oe_REG_1 <= '0';
                ie_REG_2 <= '0';
                oe_REG_2 <= '0';
                ie_IR <= '0';
                ie_IMR <= '0';
                oe_IMR <= '0';
                re_MBR <= '0';
                we_MBR <= '0';
                oe_buf <= '0';
                ie_buf <= '0';
                mr <= '0';
                mw <= '0';
                lae <= '0';
                jump <= '0';
                incr <= '0'; --PC address
        else --POBRANIE 1 INSTRUKCJ
                rst <= '1';
                next_state <= s1;
        end if;
```

**S1:** Pierwszym krokiem jest przesłanie z PC adresu instrukcji. W tym celu CU wysyła sygnał CAG do AG w celu ustawienia wyjścia z PC na szynę adresową. Przy okazji zerowane są sygnały sterujące potrzebne w kolejnych stanach. W tym kroku również wysyłany jest sygnał lae do MAR w celu wysterowania konkretnego adresu pamięci. Potem następuje przejście do kolejnego stanu.

```
when s1 => --
	stateX <= 1;
	if r_e = '1' then
		oe_buf <= '0';
		ie_buf <= '0';
		cag <= "011";
		lae<= '1';
		ie_ACC <='0';
		oe_REG_1 <= '0';
		oe_REG_2 <= '0';
		oe_IMR <= '0';
		oe_IR <= '0';
		jump<='0';
	else
		next_state<= s2;
	end if;
```

**S2:** W tym stanie następuje odczyt z pamięci. CU wysyła do Memoy sygnał mr, dzięki czemu zawartość komórki, której adres odczytany został w poprzednim stanie wysyłana jest do rejestru MBR. W tym stanie również MBR otrzymuje sygnał z CU, by odebrać dane z pamięci oraz wystawić na szynę danych swoją zawartość. Następuje wyłączenie odczytu adresu z szyny adresowej do MAR i ustalenie stanu następnego czyli S3.

```
when s2 => --odczyt z pamięci pierwszy
	stateX <= 2;
	if r_e = '1' then
		mr <= '1';
		re_MBR <= '1';
	else
		mr <= '0';
		re_MBR<= '1';
		lae <= '0';
		next_state<= s3;
	end if;
```

**S3:** Tu następuje przekazanie instrukcji z szyny danych do rejestru IR. W tym celu Cu wysyła sygnał ie do rejestru. Następuje w tym stanie również zwiększenie licznika PC o jeden poprzez sygnał incr. Na zboczu opadającym instrukcja, za pośrednictwem dekodera, trafia do jednostki sterującej. Po zapisie tej instrukcji do rejestru można już wyłączyć jej transmisję z szyny danych. W tym stanie następuje również przypisanie do zmiennych odpowiednich części sygnału z dekodera. Przy okazji sprawdzany jest warunek, czy odebrana instrukcja jest poprawna. Jeżeli tak to następny stan to S4. Jeśli nie to procesor blokuje się na stanie błędu.

```
when s3 => --zapis do rejestru instrukcji
	stateX <= 3;
	if r_e = '1' then
		increment <= "00000001";
		ie_IR<= '1';
	else
		re_MBR <='0';
		incr <= '1';
		ie_IR <= '0';
		a_mode := ird(4 downto 3); --addressing mode
		reg := ird(2);                  --register
		instr := ird(1 downto 0); --instruction

		next_state <= s4;

		if instr="11" then       --ERROR
			next_state <= ERROR;
		end if;
	end if;
```

**S4:** W tym stanie na początku następuje wyłączenie sygnału incr, a następnie sprawdzenie trybu adresowania.

1. a_mode = ”00” Tryb rejestrowy. Dla tego trybu sprawdzana jest instrukcja. Gdy jest JNOF czyli instr = "10" oraz nie ma flagi OF, następnym stanem będzie stan s10, odpowiadający za skok. Jeśli jednak flaga pojawi się, następnym stanem będzie S1. Gdy ma nastąpić instrukcja LOAD lub ADD, następnym stanem w tych dwóch przypadkach będzie s5. Dla wszystkich trzech instrukcji następuje wybór rejestru, z którego dane mają być odczytane. Dzieje się to poprzez wysłanie sygnału oe do odpowiedniego rejestru.
2. a_mode = ”01” Tryb bazowy. W tym przypadku sprawdzany jest rejestr, w którym zapisany jest adres w pamięci, gdzie przechowywana jest instrukcja. Dla odpowiedniego rejestru generator adresu wystawia jego zawartość na szynę adresową. Następny stan to s7.
3. a_mode = ”10” i a_mode = ”11” Tryby natychmiastowy i przemieszczeniowy. W tych stanach następuje przygotowanie do wczytania kolejnej komórki pamięci. W tym celu CU wystawia sygnał lae, w celu wczytania adresu pamięci z PC. AG jest ustawione na PC. Następny stan to S7.

```
when s4 =>
	stateX <= 4;
	if r_e = '1' then
		incr <= '0';
		case a_mode is
```

```
                when "00" =>        --rejestrowy

                      if reg = '0' then  --reg 1
                            oe_REG_1 <= '1';
                      else                        --reg 2
                            oe_REG_2 <= '1';
                      end if;

                when "01" => --bazowy
                      if reg = '0' then  --reg_1
                            cag <= "001";
                      else                        --reg_2
                            cag <= "010";
                      end if;
                when others  => lae <= '1';
                --natychmiastowy/przemieszczenie
                end case;
          else
                case a_mode is
                      when "00" =>              --rejestrowy
                            if instr = "10" then
                                  if flags(0)='0' then
                                  --sprawdzanie flagi OF
                                        next_state <= s10;
                                  else
                                        next_state <= s1;  --jnof
                                  end if;
                            else
                                  next_state <= s5;  --load, add
                            end if;
                      when others =>
                      --bazowy/natychmiast/przemieszczeniowy
                            next_state <= s7;
                end case;
          end if;
```

**S5:** Stan ten obsługuje instrukcje ADD i LOAD:

1. instr = ”00”. Instrukcja LOAD. W celu wykonania tej instrukcji trzeba załadować akumulator (rejestr Acc) tym co w danym momencie znajduje się na szynie danych. Dla trybu rejestrowego będzie to zawartość rejestru, dla bazowego - zawartość komórki pamięci, której adres był w rejestrze, dla natychmiastowego - zawartość rejestru natychmiastowego, a w przemieszczeniowym - zawartość komórki pamięci, na którą wskazuje adres pobrany z instrukcji (rejestr natychmiastowy). Następny stan to s1.
2. instr = ”01”. Instrukcja ADD. Instrukcja polega na wykonaniu operacji arytmetycznej dodawania. W tym celu dana zapisana w Acc po przejściu przez ALU trafia dodana do zawartości na szynie danych zostaje i zapisana do bufora.

Zapisane są również flagi do rejestru flags zgodne z wynikiem dodawania. Potem zawartość bufora wystawiona zostaje na szynę danych po jej zwolnieniu. Następny stan to s6, gdzie odbędzie się dalsza część wykonywania instrukcji ADD.

```
when s5 =>      --OPERACJE LOAD i ADD
        stateX <= 5;
        if r_e = '1' then
              incr <= '0';
              mr <= '0';
              case instr is
                    when "00" => ie_ACC <= '1';     --LOAD
                    when "01" =>
                          ie_buf <= '1';     --ADD
                          ie_flags <= '1';
                    when others => next_state <= ERROR;
              end case;
        else
              case instr is
                    when "00" =>
                          next_state <= s1; --LOAD
                          ie_ACC <= '0';
                    when "01" =>
                          re_MBR <= '0';
                          oe_buf <= '1';
                          ie_flags <= '0';
                          ie_buf <= '0';
                          oe_IMR <= '0';
                          oe_REG_1 <= '0';
                          oe_IMR <= '0';
                          oe_REG_2 <= '0';
                          next_state <= s6;
                    when others =>
                          next_state <= ERROR;
              end case;
        end if;
```

**S6:** W tym stanie następuje dalszy ciąg operacji dodawania ADD. W tym miejscu wysyłany jest sygnał ie do akumulatora, gdzie docelowo ma być zapisany wynik dodawania. Następny stan to S1.

```
when s6 =>   --OPERACJE CZ2 - ADD
      stateX <= 6;
      if r_e = '1' then
            ie_ACC <= '1';
      elsif r_e = '0' then
            ie_ACC <= '0';
            next_state <= s1;
```

```
        end if;
```

**S7:** Ten stan jest bardzo podobny do drugiego, gdyż w tym miejscu również następuje odczyt z pamięci. Sprawdzany jest tutaj tryb adresowania. Jeśli jest to tryb przemieszczeniowy lub natychmiastowy to licznik PC zwiększany jest o jeden. Ten stan obsługuje również instrukcje w trybie bazowym. Adres skoku w przypadku instrukcji JNOF jest czytany z komórki pamięci, której adres zapisany jest w rejestrze. Po odczycie pamięci następuje wystawienie jej zawartości na szynę danych za pomocą sygnału re wysyłanego do MBR. Następnie dla instrukcji JNOF następuje wysłanie sygnału skoku do PC i przejście do stanu S1, a dla pozostałych dwóch instrukcji, następuje przejście do stanu S5, gdzie owe instrukcje zostaną wykonane. Dla trybu natychmiastowego i przemieszczeniowego następny stan to S8.

```
when s7 =>   --memory read
        stateX <= 7;
        if r_e = '1' then
                mr <= '1';
                re_MBR <= '1';
        else
                if a_mode = "11" or a_mode = "10" then
                --tryb natychmiastowy/przemiesczeniowy
                        incr <= '1';
                end if;
                mr <= '0';
                re_MBR<= '1';
                lae <= '0';
                if a_mode = "11" or a_mode = "10" then
                --tryb natychmiastowy/przemiesczeniowy
                        next_state <= s8;
                elsif instr = "10" then --JNOF tryb bazowy
                        if flags(0)='0' then --sprawdzanie flagi OF
                                jump <= '1';
                        end if;
                        next_state <= s1;
                else
                        next_state <= s5; --add, load tryb bazowy
                end if;
        end if;
```

**S8:** W tym trybie następuje dalsza obsługa instrukcji w trybie natychmiastowym i przemieszczeniowym. Dane odczytane z pamięci w poprzednim stanie zapisywane są tutaj do rejestru natychmiastowego. Po zapisaniu następuje w przypadku trybu natychmiastowego wystawienie zawartości rejestru na szynę danych. Dla Instrukcji JNOF sprawa jest prosta, gdyż dla obu trybów wysyłany jest sygnał jump (pod warunkiem, że nie ma OF) do PC i przejście do stanu pierwszego. Dla trybu przemieszczeniowego następuje ustawienie AG na rejestr IMR, dzięki czemu na szynie adresowej pojawi się jego zawartość. Potrzeba odebrać teraz adres z szyny danych więc CU wysyła sygnał lae do MAR. Następuje przejście do stanu S9 gdzie nastąpi odebranie danych z pamięci. W przypadku trybu natychmiastowego dane z rejestru IMR zostaną wykorzystane do załadowania akumulatora lub zsumowane z jego zawartością. W tym celu następuje przejście do stanu S5.

```
when s8 => -- tryb natychmiastowy.2/przemieszczeniowy.2
	stateX <= 8;
	if r_e = '1' then
		incr <= '0';
		ie_IMR <= '1';
	else
		ie_IMR <= '0';
		re_MBR<= '0';
		if instr = "10" then --jnof
			if flags(0)='0' then --sprawdzanie flagi OF
				jump <= '1';
			end if;
			next_state <= s1;
		elsif a_mode = "10" then
			cag <= "000";
			lae <= '1';
			next_state <= s9; --memory_read
		else
			oe_IMR <= '1';
			next_state <= s5; --add, load
		end if;
	end if;
```

**S9:** W tym stanie odczytywane są dane z pamięci dla trybu przemieszczeniowego. Adres podany w poprzednim stanie wskazuje na komórkę pamięci, z której dzięki sygnałowi mr dane są przekazywane na wyjście pamięci. Sygnał re dla MBR wysyła dane na szynę danych, gdzie odebrane i odpowiednio wykorzystane w zależności od instrukcji zostaną w stanie S5. Następuje wyłączenie sygnału lae oraz oe z IMR.

```
	when s9 => -- tryb przemieszczeniowy.3
		stateX <= 9;
		if r_e = '1' then
		else
			mr <= '1';
			re_MBR<= '1';
			lae <= '0';
			next_state <= s11;
		end if;
```

**S10:** W tym stanie obsługiwana jest instrukcja skoku dla trybu rejestrowego. Wysłany zostaje sygnał skoku. Następny stan to s1.

```
when s10 =>
	stateX <= 10;
	if r_e='1' then
	elsif r_e='0' then
		jump <= '1';
		next_state <= s1;
	end if;
```

**S11:** Stan ten powstał podczas problemów z załadowaniem danych do bufora podczas instrukcji dodawania. Bufor próbował zapisać sumę zawartości akumulatora i szyny danych, która ze względu na opóźnienia w wystawianiu danej z pamięci była pusta. W celu rozwiązania problemu dodany został jeden takt zegara na wyprowadzenie danych z MBR.

```
when s11 =>
      stateX <= 11;
      if r_e = '1' then
      else
            next_state <= s5;
      end if;
```

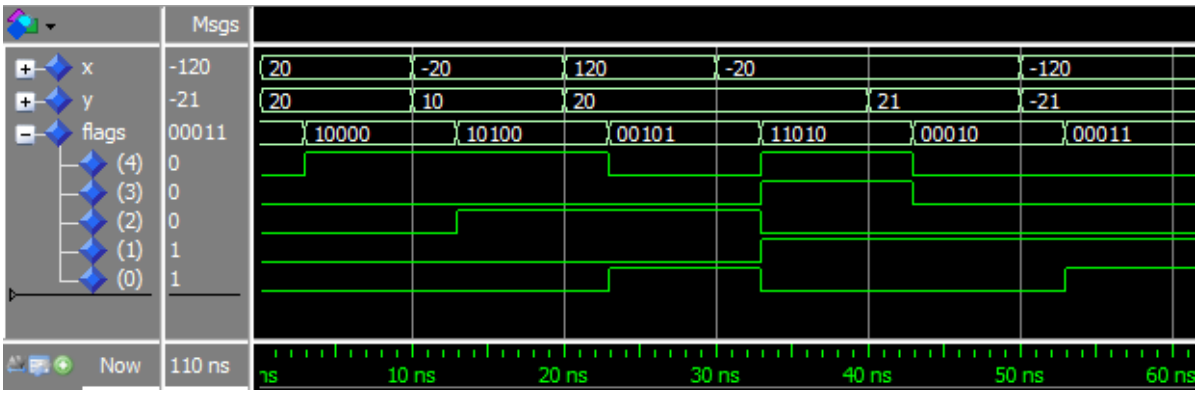
**ERROR:** Stan, w którym następuje zapętlenie pod wpływem nieprawidłowości w odczycie instrukcji.

```
when ERROR =>
      stateX <= 10;
      incr <= '0';
      next_state <= ERROR;
```

## 11. Testowanie komponentów

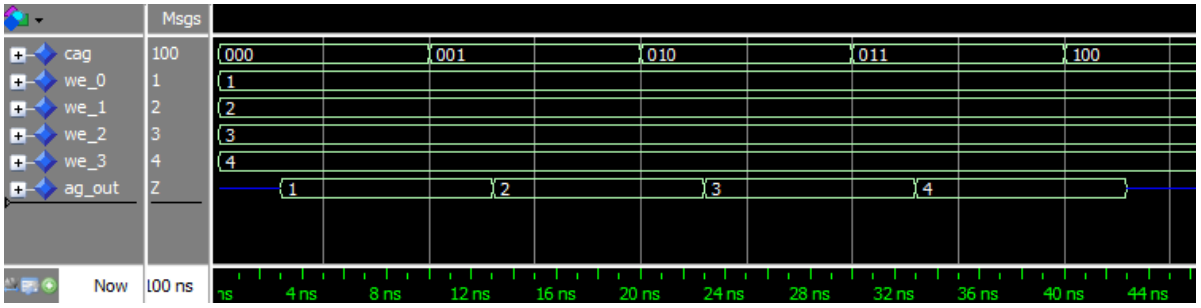
W tym punkcie prezentujemy wyniki symulacji przeprowadzonych przy pomocy napisanych przez nas testbenchów sprawdzających działanie każdego bloku procesora. Sprawdzały one poprawność reakcji bloków na sygnały wejściowe oraz poprawność czasów opóźnień.

### 11.1. ALU



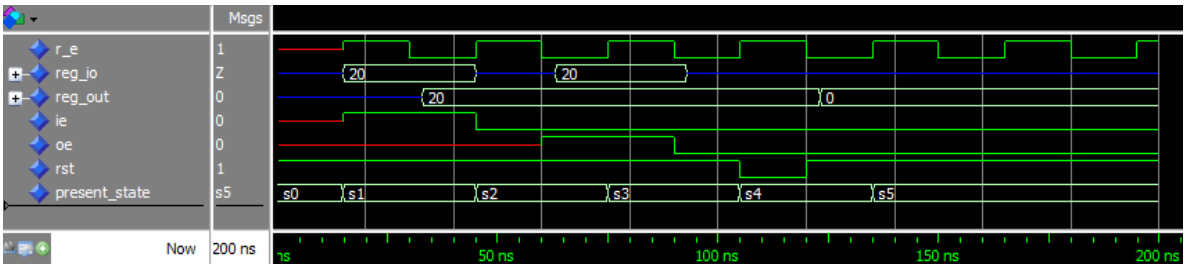

Na wejścia *x* oraz *y* jednostki arytmetycznej w odstępach 10 ns podawane są pewne liczby. Jak widać na wyjściu *z* pojawia się suma danych wejściowych mająca zakres od -128 do 127. Jeżeli liczba wykracza poza zakres to licznik “przekręca się”, a na najmłodszym bicie sygnału *flags* pojawia się ‘1’, co oznacza flagę OF. Drugi bit oznacza CF i jego wartość to 1 gdy wynik jest zbyt duży by zmieścić się na 8 bitach. Trzeci bit to SF, jest ustawiany, gdy wynik jest ujemny. Czwarty bit to ZF i pojawia się, gdy wynik jest równy 0.

Najstarszy bit oznacza flagę PF i pojawia się przy parzystych wynikach, 11.2. Generator adresu (AG)



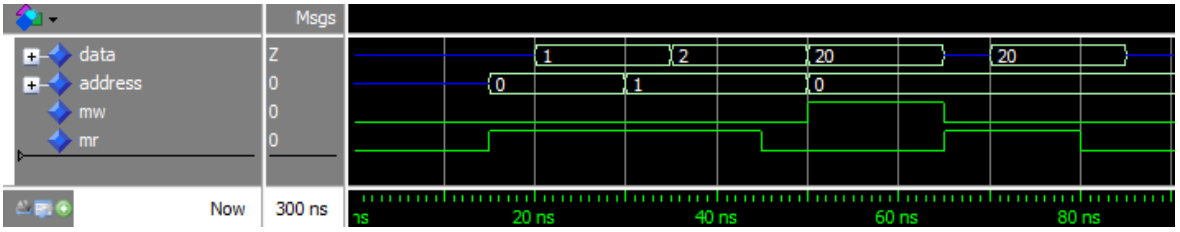

Na kolejne wejścia generatora adresu wpisujemy na stałe wartości 1,2,3,4.
Za pomocą wejścia sterującego cag kolejno zmieniamy źródło skąd pobierany jest adres, który jak widać zostaje wystawiony po ustalonym opóźnieniu.

## 11.3. Rejestr




Sygnał *r_e* zmienia się w takt zegara przyjmując stany odpowiednio od stanu zegara.
Na wejściu *reg_io* wpisywana jest wartość decymalna “20” oraz ustawiany jest sygnał *ie* i wartość ta wpisywana jest do pamięci. Na zboczu opadającym po opóźnieniu zostaje ta wartość wystawiona na wyjście *reg_out*. Następnie linia wejściowa jest zwalniana i sygnał *ie* jest wyłączany. Na kolejnym zboczu opadającym ustawiany jest sygnał *oe*, co powoduje wystawienie zawartości rejestru na port *reg_io*.
Testowany jest także reset, co prawidłowo skutkuje wyczyszczeniem pamięci - wpisanie do rejestru “0”.

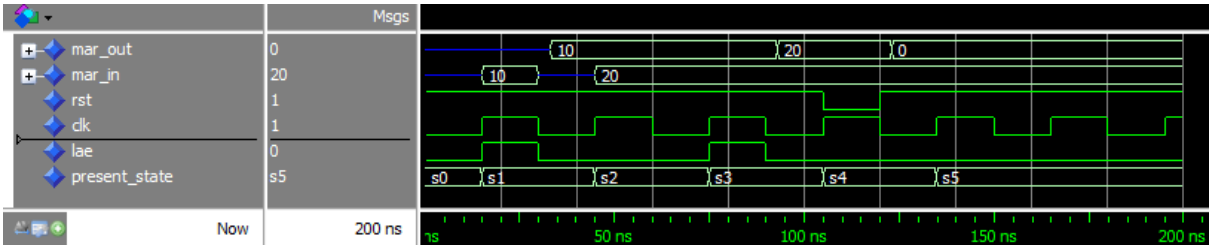
## 11.4. Pamięć (Memory)




Do testów kolejne komórki pamięci mają wartości decymalne “1”, “2”. Aktywowany jest sygnał *mr* i ustawiane są kolejno adresy “0”, “1”, co skutkuje pojawieniem się na porcie i/o odpowiednio przypisanych wartości. Test odczytu jest prawidłowy.

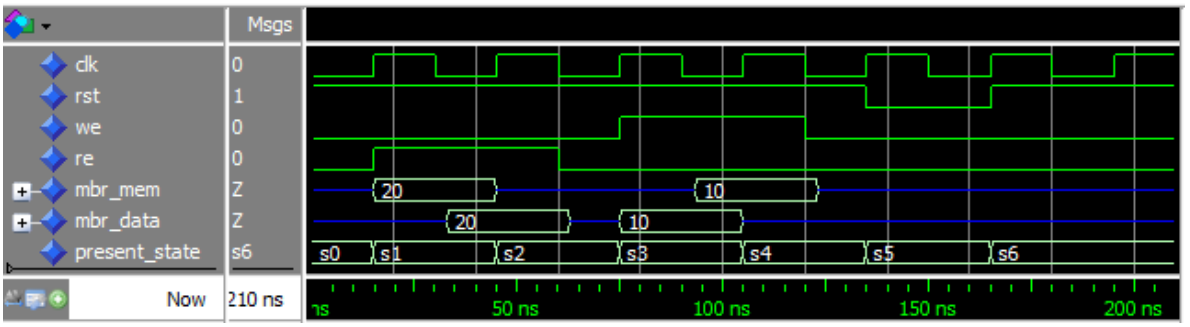
Zdejmowany jest sygnał *mr*, a następnie: na port data wystawiana jest wartość “20”, adres wynosi “0”. Dokonuje się próba nadpisania poprzedniej wartości tej komórki pamięci, co skutkuje sukcesem.

## 11.5. MAR



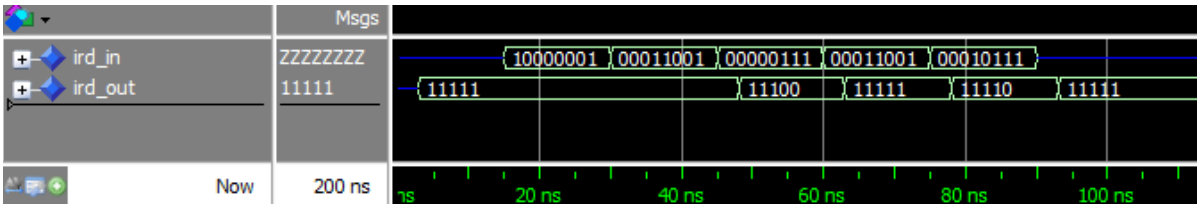

Na wejście podawana jest wartość “10” wraz z sygnałem *lae*. Co skutkuje zapisanie tej wartości w pamięci. Po wyłączeniu sygnału lae, dane przestają być wpisywane i moduł nie reaguje na zmianę, zareaguje dopiero przy kolejnym zboczu narastającym, gdy będzie wystawiony tej sygnał.

## 11.6. MBR



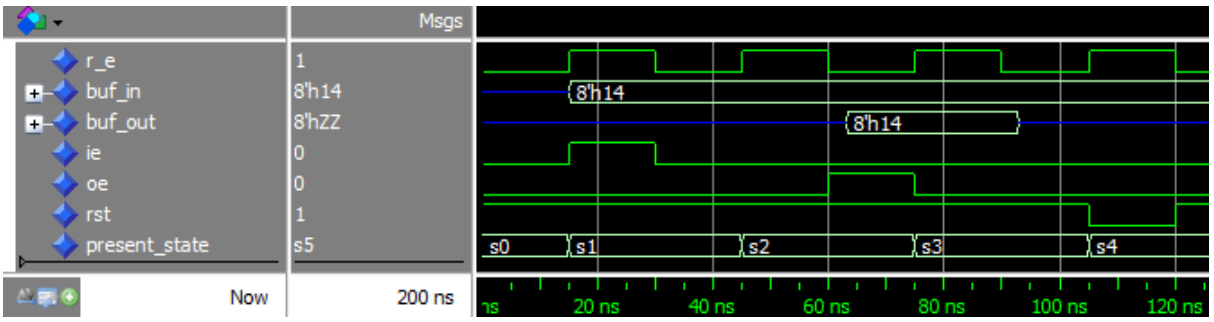

Na wejście *mbr_mem* wystawiane są dane “20” wraz z sygnałem *re*, co skutkuje wypisaniem tych danych na *mbr_data*. Analogicznie dzieje się dla sygnału sterującego *we*, gdzie dane z *mbr_data* przepisywane są na port *mbr_mem*.

## 11.7. Dekoder rozkazów (IR_DECODER)



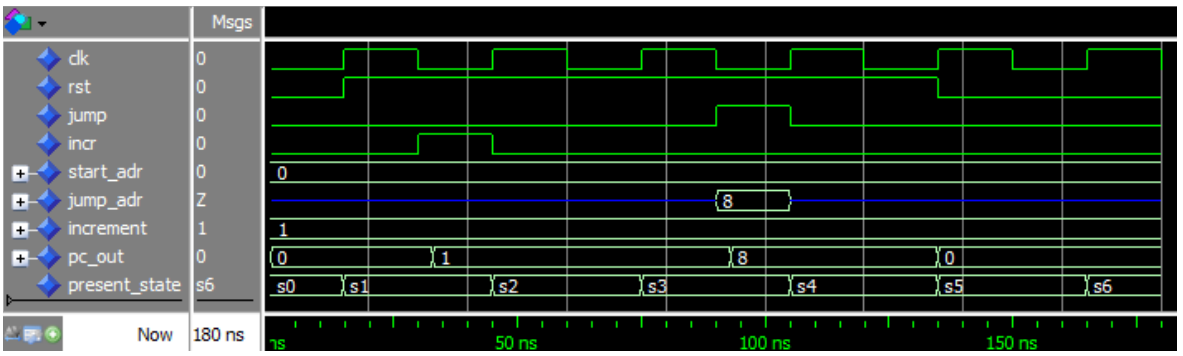

Dekoder rozkazów prawidłowo zamienia kod instrukcji na kod zgodny z tabelą znajdującą się w początkowej części sprawozdania. Dla nieprawidłowych danych wystawia kod błędu “11111”, wprowadzamy dla testu nieprawidłowy kod operacji “11”, co skutkuje wystawieniem informacji o błędzie. Nieprawidłowa instrukcja to też taka, która nie ma zer na pierwszych trzech bitach, gdyż są one wykorzystywane.

## 11.8. Bufor (BUF)



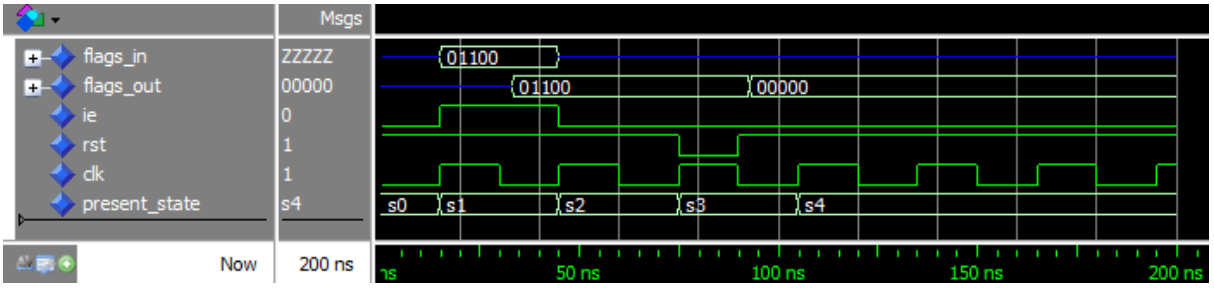

Zasada działania jest taka sama jak zwykłego rejestru. Na wejście wpisywana jest wartość “14”, pojawia się sygnał *ie*, co skutkuje zapisem do pamięci, odczyt odbywa się dla aktywnego sygnału *oe*. Bufor działa prawidłowo.

## 11.9. Licznik rozkazów (PC)




Na początku podawany jest adres startowy, w tym przypadku decymalne 0. Pod wpływem krótkiego sygnału *incr* licznik zostaje zwiększony o wartość sygnału increment. Po opóźnieniu wartość ta pojawia się na wyjściu *pc_out.* W późniejszej kolejności testowany jest skok. Po otrzymaniu na zboczu opadającym sygnału *jump,* licznik skacze do wartości podanej w tym samym czasie na *jump_adr*. Na końcu po otrzymaniu sygnału *rst,* licznik wskazuje adres początkowy.

## 11.10. Bufor flag (Flags_buf)




Na wejściu *flags_in* wpisywany jest ciąg bitów odpowiadających wartością flag będących wynikiem operacji arytmetycznej W tym przypadku jest to “01100”. Ustawiany jest także sygnał *ie*. Na zboczu opadającym, po opóźnieniu zostaje ten ciąg wystawiony na wyjście

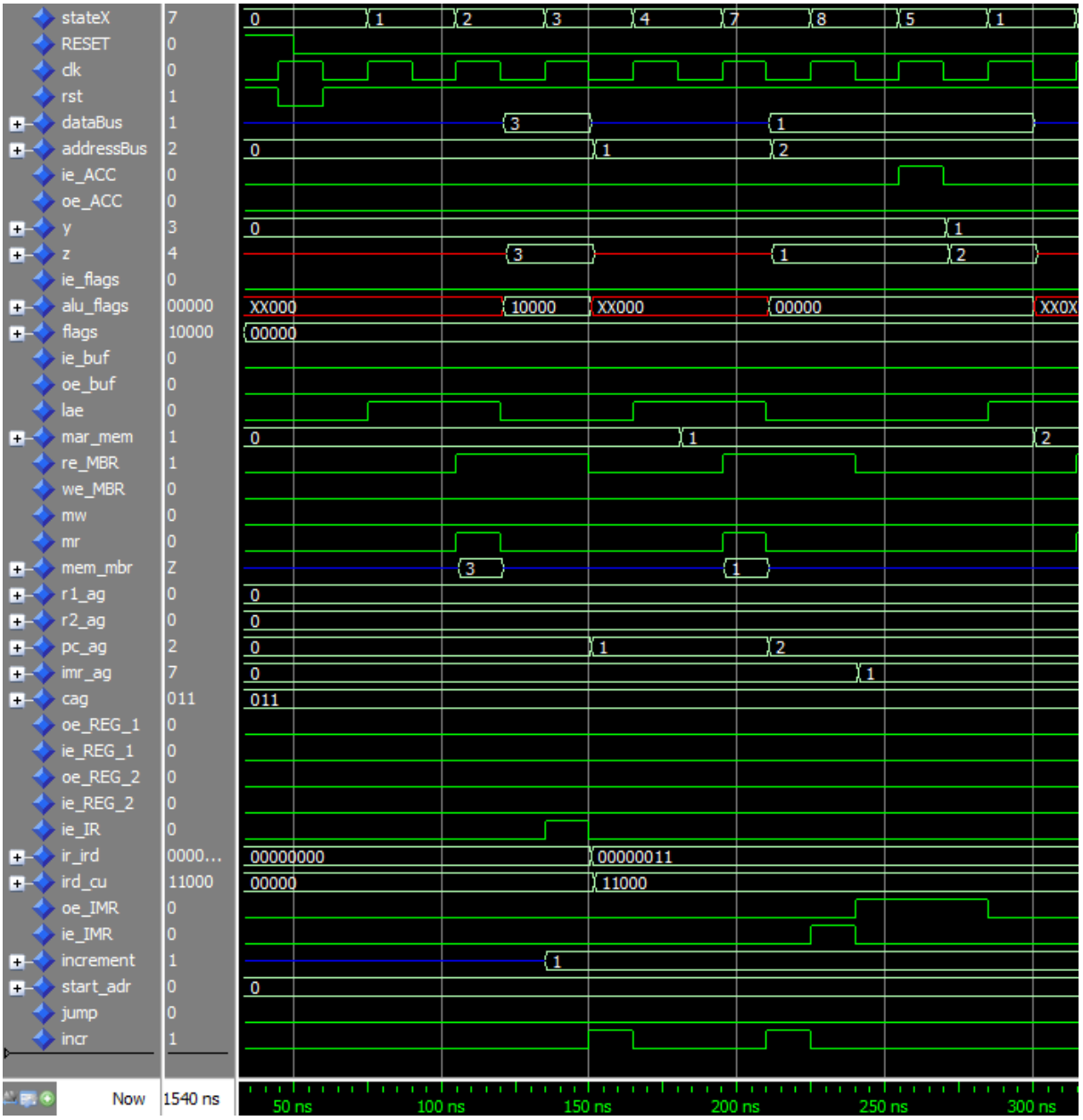
*flags_out,* co potwierdza prawidłowy zapis na zboczu narastającym. Następnie pod wpływem sygnału *rst* zawartość, rejestry zostaje wyzerowana.

## 12. Testowanie całego systemu

W celu ostatecznego przetestowania procesora wykonane zostaną testy poprawności działania instrukcji. Wszystkie sygnały z pliku *system.vhd* zostały przedstawione na zrzutach ekranu z symulacji. Wszystkie trzy przetestowane instrukcje następowały po sobie, jednakże dla lepszej widoczności przebiegów sygnałów podczas symulacji, wyniki zostały podzielone ze względu na poszczególne operacje.
Pomyślne testy następujących po sobie instrukcji doskonale pokazują, że cały układ prawidłowo wykonuje wcześniej ustalone założenia projektowe.

12.2. Test załadowania wartości do akumulatora trybem natychmiastowym i wykonanie operacji dodawania argumentu zaadresowanego trybem bazowym.

12.1. LOAD 1

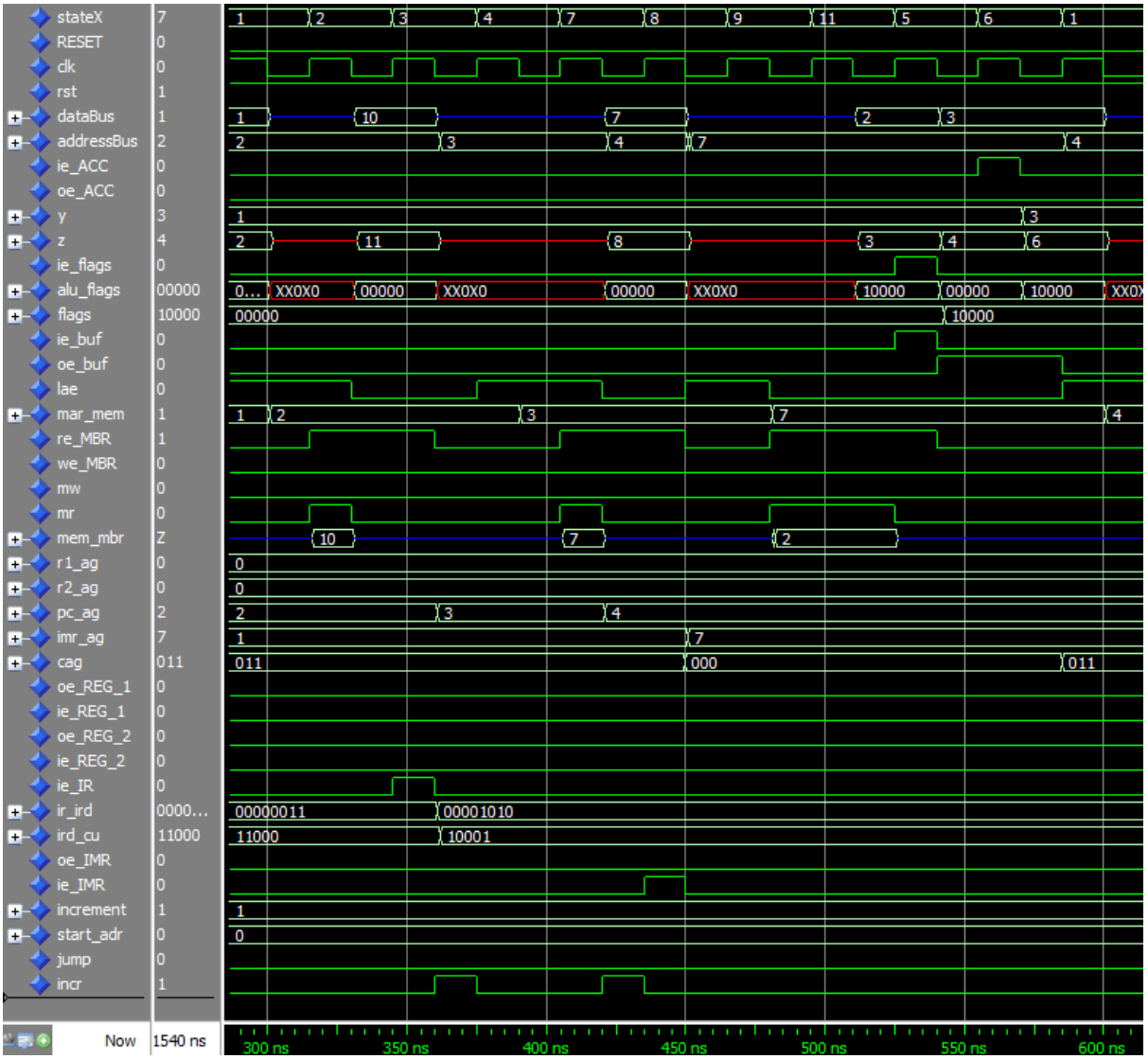



1) reset systemu: reset wszystkich komponentów, wpisanie początkowego adresu licznika rozkazów równego “0”, ustawienie generatora rozkazów na adres z PC i w konsekwencji wystawienie adresu początkowego licznika rozkazów równego “0” na szynę adresową,
2) Pobranie adresu początkowego z szyny adresowej do MAR przy pomocy sygnału sterującego *lae*,
3) Odczytanie z pamięci instrukcji spod adresu pamięci adresowej (MAR) przy pomocy sygnału *mr* (memory read) oraz wystawianie jej na szynę danych (sygnał sterujący *re_MBR*),

4) Zapisywanie instrukcji do rejestru IR, dekodowanie odebranej instrukcji, zwiększenie licznika procesów o wartość spod portu *increment* - “1”. Rozpoznanie adresowania natychmiastowego,
5) Pobranie adresu danej z PC (w tym przypadku adres to “1”) i zapisanie go w MAR,
6) Odczyt danej z pamięci spod adresu zapisanego w MAR oraz wystawienie go na szynę danych. Następuje zwiększenie licznika o “1”,
7) Zapis do IMR z szyny danych, a następnie wystawienie ich z IMR na szynę danych,
8) Zapisanie danych z IMR do akumulatora.

Zwolnienie linii danych. Koniec operacji.

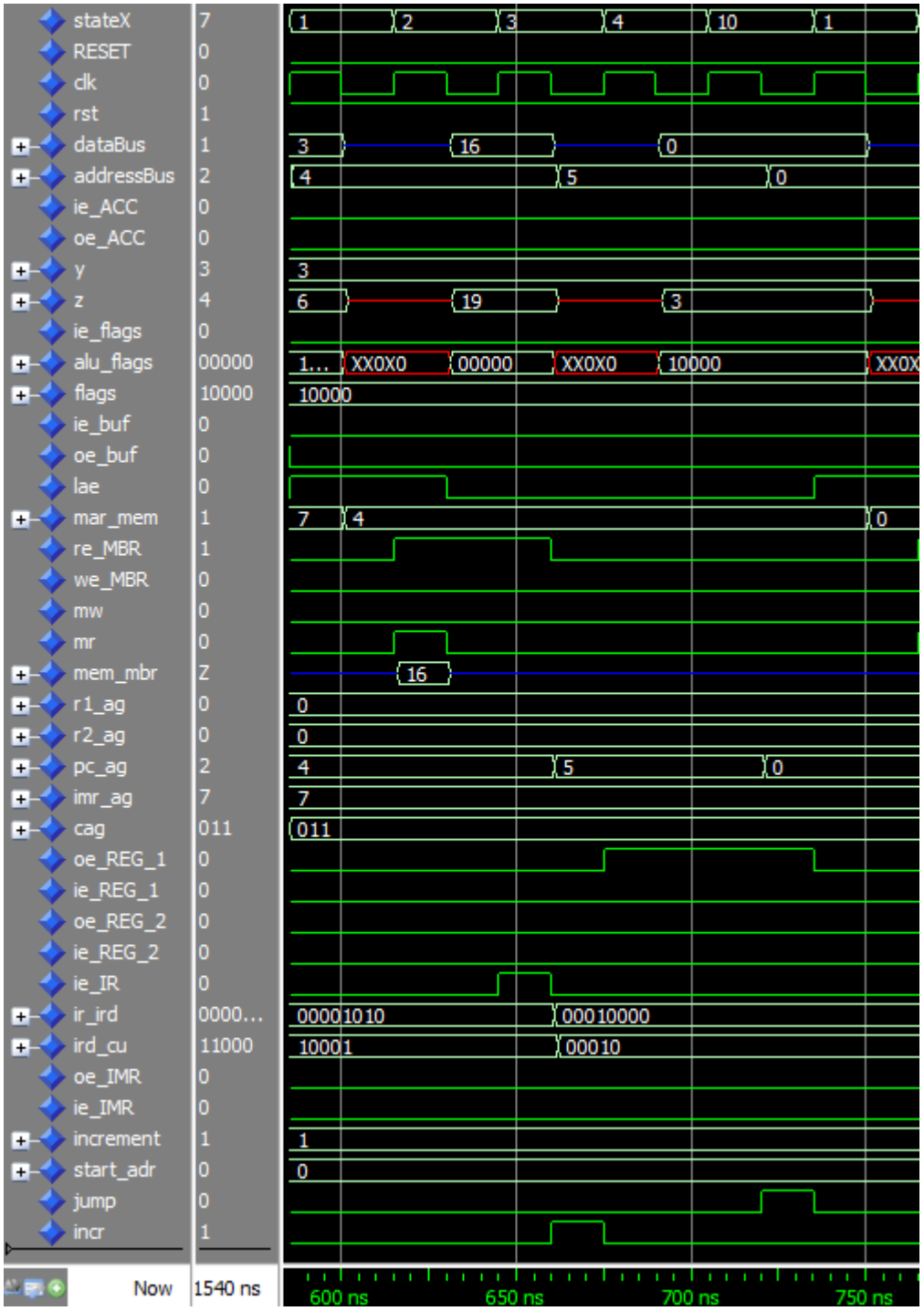
12.2. ADD [7], gdzie [7] = 2




1) wystawienie na szynę adresową adresu kolejnej instrukcji spod PC równego “2” i wpis adresu do MAR,
2) Odczytanie z pamięci instrukcji spod adresu MAR,- przy pomocy sygnału *mr* oraz wystawianie jej na szynę danych,
3) Zapisywanie instrukcji do rejestru IR, dekodowanie odebranej instrukcji, zwiększenie licznika procesów o “1”. Rozpoznanie trybu przemieszczeniowego.
4) Wpis do MAR adresu danej z PC, tj. kolejnej komórki pamięci - adres przemieszczeniowy.
5) Odczyt z pamięci adresu przemieszczeniowego równego “7”, zapis do MBR i wystawienie na szynę danych. Zwiększenie licznika o “1”.
6) Wpis zawartości szyny danych (adresu przemieszczeniowego) do IMR
7) Wystawienie adresu spod IMR za pomocą AG na szynę adresową

8) Wpis MAR z szyny adresowej
9) Odczyt danej spod adresu “7” z pamięci, zapis do MBR i wystawienie danej (w naszym przypadku “2”) na szynę danych.
10) Zapis do bufora sumy zawartości akumulatora oraz szyny danych (2+1=3)
11) Wystawienie zawartości bufora na szynę danych.
12) Zapis do akumulatora i zwolnienie szyny danych.

12.3. JNOF REG1




1) wystawienie na szynę adresową adresu kolejnej instrukcji spod PC równego “4” i wpis adresu do MAR,

2) Odczytanie z pamięci instrukcji spod adresu MAR przy pomocy sygnału *mr* oraz wystawianie jej na szynę danych,

3) Zapisywanie instrukcji do rejestru IR, dekodowanie odebranej instrukcji, zwiększenie licznika procesów o “1”. Rozpoznanie trybu rejestrowego,
4) Wystawienie zawartości danego rejestru na szynę danych, sprawdzenie flagi *Overflow*: jeśli równa jest ‘0’, czyli nie ma przepełnienia to realizacja skoku, w przeciwnym wypadku przejście do następnej instrukcji,
5) Flaga *OF* jest równa ‘0’, więc skok pod adres wystawiony na szynę danych, tj. wpisanie do PC adresu “0”, gdyż taka jest zawartość rejestru 1.

   W tym momencie następuje zapętlenie programu.

## 13. Wnioski

Udało się zaprojektować działający procesor zgodnie z danymi wytycznymi.
Dzięki temu projektowi lepiej zrozumieliśmy działanie prostego procesora o architekturze akumulatorowej. W zaprojektowanym procesorze przewidzieliśmy możliwości rozbudowy go o nowe operacje. Bardzo ważne jest wykonanie dobrego projekt schematu i diagramu automatu, aby na podstawie tych informacji można było zrealizować, zaprogramować zadane funkcjonalności.

Projekt został ukończony pomyślnie.