Raport PBL

Christopher Lozinski
Paweł Orszulik
Łukasz Fuss

1. Temat projektu PBL

Projektem realizowanym w ramach PBL było stworzenie modelu procesora zgodnego z zestawem instrukcji normy *IEC 61131-3: Programming Industrial Automation Systems*. Przeprowadzono weryfikację, syntezę oraz przetestowano działanie zaproponowanego rozwiązania na układzie FPGA Altera CycloneV DE1-SoC.

2. Założenia projektowe

Podczas projektowania procesora szczególną uwagę poświęcono poniższym wymaganiom:

- instrukcje procesora są trójargumentowe,
- instrukcje są zgodne z normą IEC 61131-3,
- procesor ma być jednocyklowy(jeśli ograniczenia technologiczne na to pozwolą).

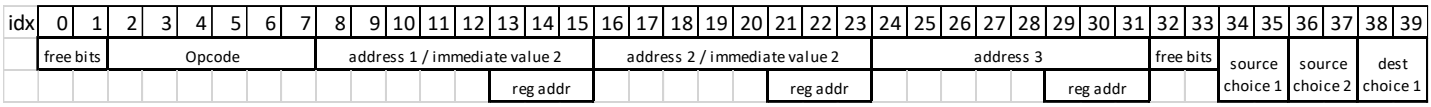
3. Lista instrukcji procesora

Pierwszym krokiem podczas projektowania tej jednostki było stworzenie listy instrukcji z godnej z normą IEC 61131-3. Poniższa lista przedstawia dostępne instrukcje w zaprojektowanym procesorze wraz z opisem operandów.

op_code	Instruction	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h00	AND	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h01	ANDN	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h02	OR	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h03	ORN	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h04	XOR	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h05	XORN	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h06	NOT	arg1 (source1) 8bit	-	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	-	Destination_choice 2 bits
8'h07	ADD	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h08	SUB	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h09	MUL	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h0A	DIV	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h0B	MOD	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h0C	GT	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h0D	GE	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h0E	EQ	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h0F	NE	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h10	LE	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h11	LT	arg1 (source1) 8bit	arg2(source2) 8 bits	arg3(destination) 8bits	2 free bits	Source1_choice 2bits	Source2_choice 2bits	Destination_choice 2 bits
8'h12	JMP	jmp_addr	-	-	2 free bits	-	-	-
8'h13	JMP0	jmp_addr	-	-	2 free bits	-	-	-
8'h14	JMP1	jmp_addr	-	-	2 free bits	-	-	-
8'h15	CAL	jmp_addr	-	-	2 free bits	-	-	-
8'h16	CAL0	jmp_addr	-	-	2 free bits	-	-	-
8'h17	CAL1	jmp_addr	-	-	2 free bits	-	-	-
8'h18	RET	-	-	-	2 free bits	-	-	-
8'h19	RET0	-	-	-	2 free bits	-	-	-
8'h1A	RET1	-	-	-	2 free bits	-	-	-
8'h1B	S	-	-	arg3(destination) 8bits	2 free bits	-	-	Destination_choice 2 bits
8'h1C	R	-	-	arg3(destination) 8bits	2 free bits	-	-	Destination_choice 2 bits
8'h1D	ST	-	-	arg3(destination) 8bits	2 free bits	-	-	Destination_choice 2 bits
8'h1E	STN	-	-	arg3(destination) 8bits	2 free bits	-	-	Destination_choice 2 bits
8'h1F	LD	arg1 (source1) 8bit	-	-	2 free bits	Source1_choice 2bits	-	-
8'h20	LDN	arg1 (source1) 8bit	-	-	2 free bits	Source1_choice 2bits	-	-
8'h21	NOP	-	-	-	2 free bits	-	-	-

4. Format instrukcji

Naszym zadaniem było zbudowanie 3-argumentowego procesora, w którym każdy argument może być adresem rejestru, adresem pamięci bitowej, adresem pamięci o szerokości słowa lub zmienną natychmiastową. Przyjęliśmy, że pojedyncza instrukcja będzie miała szerokość 40 bitów.



Instrukcja składa się z:

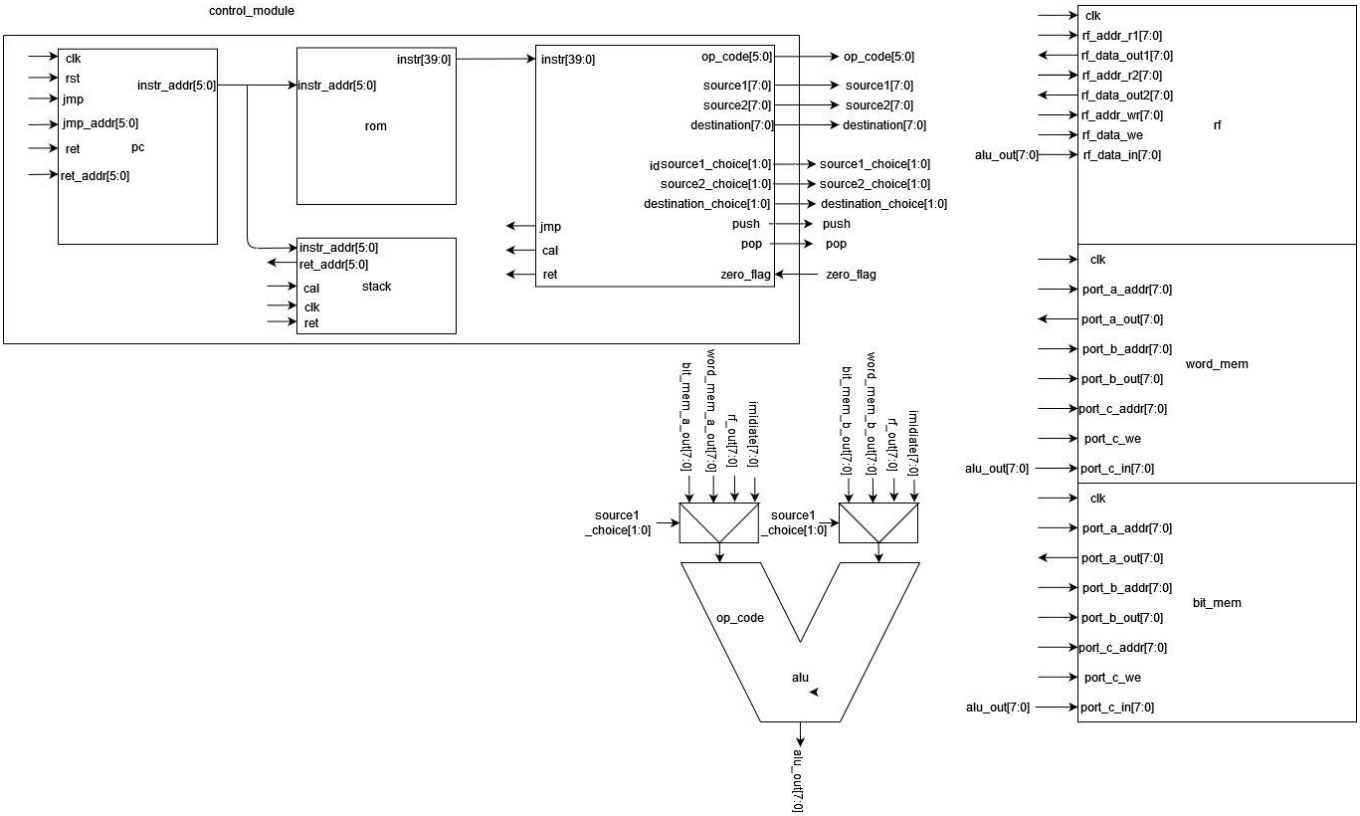
- 8 bitowego kodu operacji,
- 8 bitowego adresu pamięci, rejestru lub wartości zmiennej natychmiastowej (źródło 1),
- 8 bitowego adresu pamięci, rejestru lub wartości zmiennej natychmiastowej (źródło 2),
- 8 bitowego adresu pamięci lub rejestru (adres zapisu wyniku operacji),
- 2 niewykorzystanych bitów,
- 2 bitów kodujących miejsce odczytu źródła 1,
- 2 bitów kodujących miejsce odczytu źródła 2,
- 2 bitów kodujących miejsce zapisu wyniku operacji.

Warto zauważyć, że kod operacji potrzebuje tylko 6 bitów. Dodanie 2 bitów, które nie są wykorzystywane w żaden sposób jest podyktowane formatem pliku hexadecymalnego(hex file, .hex). Hex file jest podzielony w taki sposób, że jego pojedynczy element składa się z 8 bitów. Zawartość tego pliku wgrywana jest później bezpośrednio do pamięci ROM, z której procesor odczytuje kolejne instrukcje.

Kodowanie miejsca zapisu i odczytu przedstawia poniższa tabela. W jednostce przyjęto szerokość słowa równą 8 bitom.

Kod	Miejsce odczytu lub zapisu
00	Rejestry
01	Pamięć bitowa – RAM Bit
10	Pamięć o szerokości słowa – RAM Word
11	Zmienna natychmiastowa(możliwy tylko odczyt)

5. Schemat procesora



Stworzony przez nas schemat procesora jest prosty w zrozumieniu oraz pozwala na łatwe rozróżnienie ról poszczególnych modułów. Architektura procesora pozwoliła nam na ponowne wykorzystanie modułów napisanych przez nas podczas innych przedmiotów. Zaletą takiego rozwiązania jest to, że nasze moduły zostały już gruntowanie przetestowane i pozwoliło na stworzenie działającego prototypu w bardzo krótkim czasie. Prototyp był wykorzystywany przy testowaniu zachowania bloków

pamięci RAM dostępnych w układzie Altera DE1-SoC. Dzięki tym eksperymentom udało się potwierdzić, że technologia umożliwi nam stworzenie procesora jednocyklowego.

6. Opis modułów procesora

Jednostka arytmetyczno-logiczna – ALU

Zadaniem tego układu kombinacyjnego jest wykonywanie operacji arytmetycznych, logicznych i zarządzania pamięcią RAM. Wewnętrzna logika wypracowuje odpowiedni wynik dla dwóch argumentów wejściowych i wybranej operacji(przez sygnał op_code). ALU przesyła stan flag Carry(C) i Borrow(B) do rejestru flag. Wartym podkreślenia jest sposób wyboru źródła argumentów wejściowych, dla wejścia in_a i in_b możemy odczytać wartość zmiennej z czterech źródeł zależnie od sygnałów source1_choice i source2_choice. Poniżej opisana jest zależność między sygnałem sourceX_chioce(X może być równe 1 albo 2) a źródłem zmiennej – tożsama z kodowaniem miejsca zapisu i odczytu przedstawionym w 4. punkcie sprawozdania.

sourceX_choice	Źródło odczytu
00	Rejestry
01	Pamięć bitowa – RAM Bit
10	Pamięć o szerokości słowa – RAM Word
11	Zmienna natychmiastowa

Rejestr Flag – Flag Reg

W tym rejestrze zatrzaskiwany jest stan flag Zero(Z), Carry(C) i Borrow(B). Flaga Z informuje o wystąpieniu zera w rejestrze R2. Zatrzaskiwanie stanu flagi Z przy opadającym zboczu zegara pozwala na zaktualizowanie stanu flagi pół cyklu zegara wcześniej. Reszta układu jest zsynchronizowana z narastającym zboczem. Takie rozwiązanie jest konieczne do realizacji skoków w programie opartych o zawartość akumulatora(rejestr R2) w mikroprocesorze jednocyklowym. W naszym mikroprocesorze każda instrukcja potrzebuje dokładnie jednego taktu zegara do poprawnego wykonania się. Na przykład instrukcja IF0JUMP wykona skok do podanego adresu kiedy flaga Z jest równa 1 - zero w akumulatorze. Pod uwagę brany

jest stan flagi Z przed narastającym zboczem z zachowaniem czasów setup i hold. Jeśli w instrukcji poprzedzającej IF0JUMP doszło do zmiany stanu flagi Z to stan tej flagi w rejestrze flag zostanie zaktualizowany tylko wtedy, kiedy rejestr flag jest zsynchronizowanym z opadającym zboczem. Użycie narastającego zbocza wymagało by odczekania jednego taktu zegara w celu odczytania przez IF0JUMP oczekiwanej przez programistę wartości flagi Z. Flagi C i B są aktualizowane tylko jeśli ALU wysteruje sygnał flag_cb_valid i wystąpi narastające zbocze zegara. Pozwala to na podtrzymanie stanu flag C i B na czas dłuższy niż jeden takt zegara. Wszystkie flagi można wyzerować sygnałem flag_rst.

Pamięć operacyjna bitowa - RAM Bit

Pamięć operacyjna bitowa została dodana do naszego mikroprocesora w celu realizacji obsługi wejść i wyjść. Rozmiar pamięci to 256 bitów. Posiada dwie bramy pozwalające na odczyt pamięci i jedną bramę pozwalającą na zapis. W symulacji używany jest model tej pamięci napisany w Verilogu. Do syntezy użyto dedykowanej, dla układu FPGA Intel Altera Cyclone V, pamięci RAM z katalogu IP Core. Ta pamięć bazuje na blokach pamięci MLAB, które znajdują się fizycznie w układzie FPGA. Niestety w katalogu IP Core Intela nie występuje wersja trójbramowa pamięci RAM. Dlatego do skonstruowania jednej pamięci RAM z dwoma bramami czytającymi i jedną zapisującą, użyto dwóch bloków pamięci MLAB. W celu zapewnienia spójności danych, do obu bloków wykonywany jest jednoczesny zapis. Zewnętrzny układ jest odpowiedzialny za odczyt stanu wejść fizycznych i zapisanie tych stanów w konkretnych adresach pamięci RAM Bit. Drugim zadaniem tego układu jest odczytanie stanu komórek pamięci RAM Bit, przeznaczonych do ustawiania stanu wyjść przez mikroprocesor, i ustawienie wyjść fizycznych zgodnie ze stanem tych komórek.

Pamięć operacyjna(8-bit) - RAM Word

Pamięć operacyjna przechowująca zmienne o szerokości 8 bitów. Rozmiar pamięci to 256 8-bitowych słów. Została tak samo skonstruowana jak pamięć RAM Bit, także posiada dwie bramy czytające i jedną bramę zapisującą zmienną do pamięci.

Rejestry - Register File

Jest to zbiór rejestrów operacyjnych. Rejestrów jest 11, każdy o szerokości 8 bitów, z czego programista ma pełny dostęp tylko do 9 z nich. Rejestr R0 na stałe jest równy 8’h00 i pozwala tylko na odczyt tej wartości. Rejestr R1 na stałe przechowuje

8’h11 i zachowuje się jak R0. Rejestr R2 to akumulator(ACC, CR – Current Result), na jego podstawie ustawiana jest flaga Z. W celu spełnienia jednocyklowości tego mikroprocesora, koniecznym było wprowadzenie przewidywania kolejnego stanu flagi Z. Dzieje się to przez analizę czy dana, która przyszła do rejestrów będzie zapisana do R2 i czy jest równa 8’h00. Jeśli tak jest to flaga Z jest ustawiana w stan wysoki mimo, że w R2 może jeszcze nie znajdować się wartość 8’h00. Tym sposobem udało się uniknąć konieczności czekania jednego taktu zegara po zmianie stanu flagi Z jeśli programista chciał wykonać instrukcję np. IF0JUMP. Układ ten posiada także zaimplementowany stos. Rejestry R3-R10 są ogólnego przeznaczenia. Zbiór rejestrów posiada dwie bramy do odczytywania poszczególnego rejestru i jedną bramę do zapisu do konkretnego rejestru(poza R0 i R1). Wybór konkretnego rejestru odbywa się przez podanie jego adresu(adresy dostępne: 0 - 11 przy odczycie, 2 - 11 przy zapisie).

Licznik programu – Program Counter (PC)

Licznik programów to rejestr procesora zawierający adres następnej instrukcji. Moduł posiada możliwość resetu licznika, skoku na zadany adres instrukcji (adres bezpośredni) oraz skoku na adres powrotu.

Pamięć ROM – pamięć programu

Moduł zawierający instrukcje procesora realizujące zadany program. Instrukcje są zapisane w pliku .hex i odczytywane w czasie kompilacji modułu.

Dekoder

Moduł dekodujący instrukcje i sterujący skokami warunkowymi jak i bezwarunkowymi oraz skokiem na adres powrotu. Moduł steruje również stosem zapisując stan rejestrów w przypadku skoku z śladem. Ponadto ważną funkcją dekodera jest zdekodowanie instrukcji na kod operacji, argumenty oraz kody miejsc odczytu źródeł oraz zapisu.

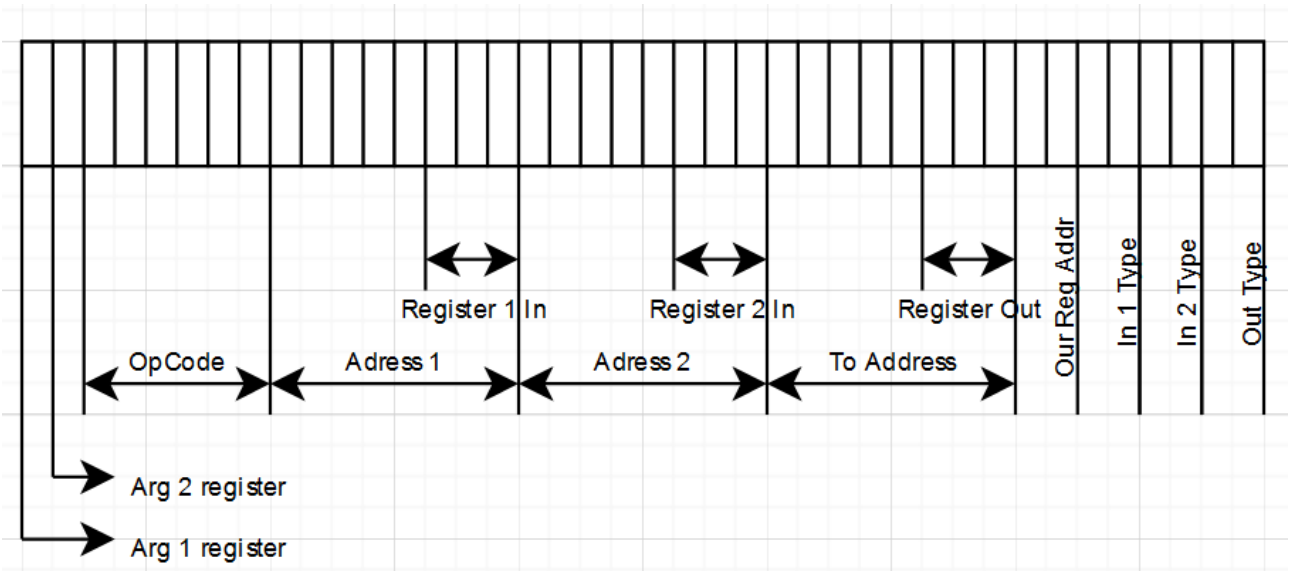
Stos

Stos zachowujący adres powrotu w przypadku wykonania instrukcji Call.

7. Testy jednostki

Iterative Output Application

To test our initial design we wrote program that iterates through memory. It allowed for the memory address to be read from a specified register, allowing for iteration through arrays. In the end it was not required. Maybe this functionality is less needed for real time control, than in general computing. All of the opcodes were the same between the two designs. Most of the modules were also the same. In this approach, rather than a centralized controller, logic was devolved to the specific modules. Those modules received copies of the opCode, and were responsible for figuring out how to respond to it. They are described here. Stack responded to RET, CALL and RST opcodes. Parser only parses the input instructions and does not operate on the input. Program Counter responds to RET, CALL, JMP, IF0JUMP, IF1JUMP and RST q
This design also supports iteration by incrementing a register as the memory address. The instruction layout is presented below.



Basically the first two bits on the left, and bit [6] from the right determined if the memory address should be taken from a register. If arg 1 register is TRUE, then the address for argument 1 comes from register [arg1]. If the arg 2 register is TRUE, then the address for argument 2 comes from register [arg2]. If the arg out register is TRUE, then the result is written to the address given in register [argOut]. It turns out that the final project did not need this functionality.

Iterative Output Results

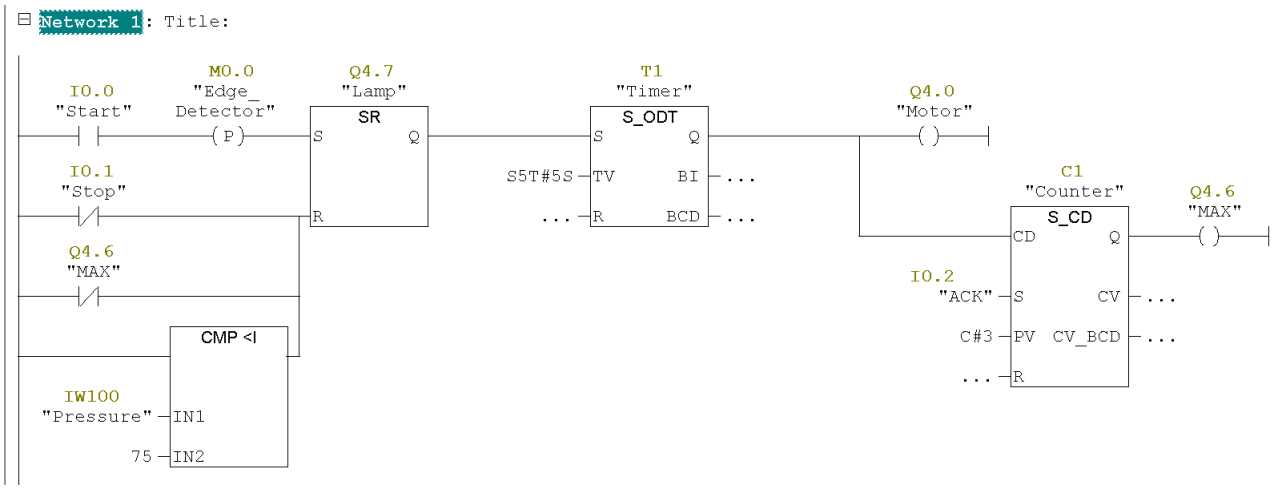
INITIALIZING BIT MEMORY
ADDRESS DATA
0 1
1 1
2 0
3 1
.
.
.

INITIALIZING WORD MEMORY
ADDRESS DATA
0 0
1 1
2 2
3 3
.
.
.

BitA WordB Product
1 0 x
1 1 0
0 2 1
1 3 0
1 4 3
.
.
.

Program testowy nr 1

Poniższy schemat przedstawia program testowy napisany w języku LAD. Do testów naszej jednostki program w języku LAD został przetłumaczony na instrukcje naszego procesora.



Kod programu testowego

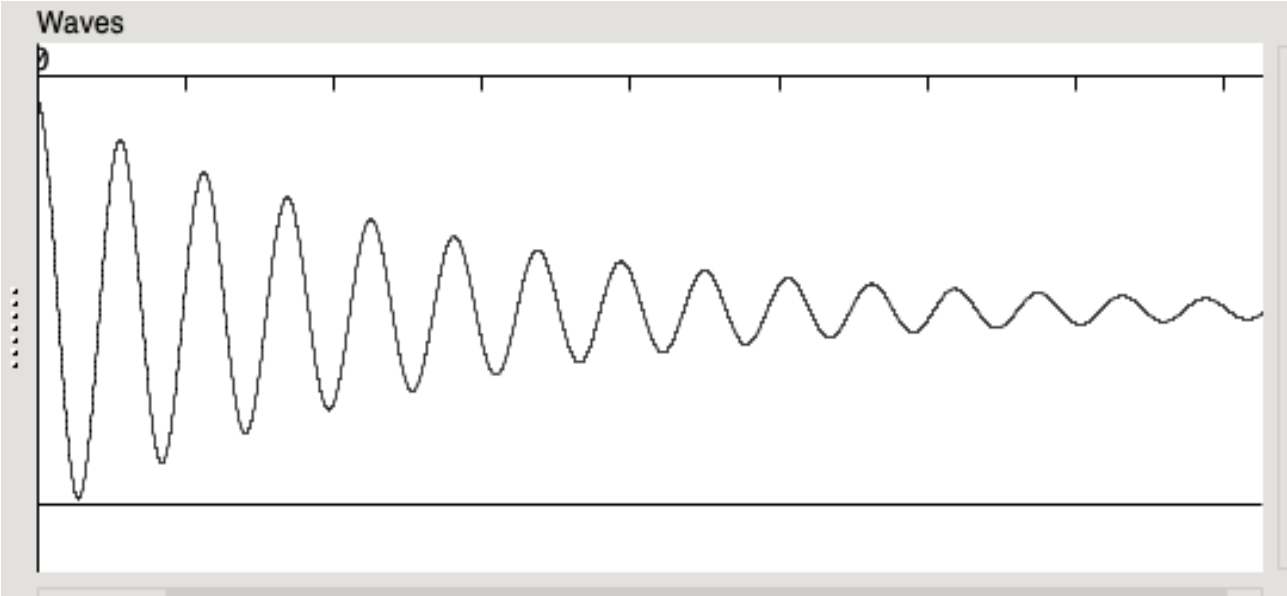
```
//  assemby code for PBL
//  compilator ommits empty lines
//  to comment line add "//" in front !!IMPORTANT!! NO SPACES BEFORE "//"
//   to add comment at the end of line add "//" followed by your comment
//  use def keyword followed by name followed by value
//  it works like defines in C so for example:
//  def start 00
//  AND start 00 02 300
//  start in AND operation will be replaced in compilation by 00
///////////////// explanation for current code /////////////////
//  r_ prefix register addr
//  b_ prefix bit memory addr
//  w_ prefix word memory addr
//  i_ data_imidiate value in hex
////////////////////// DEFINE SECTION ////////////////////////////////
def reg_0       00
def reg_1       01
def r_acc       02 // reg addr of acc
```

```
def b_start     00 // mem bit addr
def b_stop      01 // mem bit addr ~b_stop resets lamp and timer
def b_max       02 // mem bit addr
def b_lamp      03 // mem bit addr lamp
def b_motor     04 // mem bit addr motor
def b_ack       05 // ack motor started 3 times
def w_pressure  00 // mem word addr
def w_timer     01 // timer register
def w_cnt       02 // counter for counting down motor tests
def w_temp      03 // addr temp
def w_pres_cond 04 // addr checking pressure condition
def w_sub       05 // addr for substraction from timer
def w_sub2      06 // addr for substraction fron counter
def i_80        50 // data imidiate 80%
def i_wait_time 02 // time in clk cycles to count
def i_75        4B // data imidiate 75%
///////////////// setting memory values /////////////////
RST  b_ack      1          //set ack to 0
RST  b_lamp     1          //set initial lamp value to 0
SET  b_start    1           //set start button to 1
SET  b_stop     1          //set stop value to 1 (active state = 0)
SET  b_max      1          //set max value  to 1 (active state = 0)
RST  b_motor    1          //set motor driver to 0
OR   reg_0  i_wait_time w_timer  032 //set timer = wait time
OR   reg_0 i_80 w_pressure 032 //set pressure to i_80%
OR   reg_0 02   w_timer    032 //set timer for to cycles
OR   reg_0 03   w_cnt      032 //set motor startup counter to 3 times
///////////////Program////////////////////////////////
RST  w_temp 2        //reset registers
RST  r_acc  0        //reset registers
AND  b_stop b_max r_acc 110     //Evaluate reset condition
NOT  r_acc r_acc 00   //Evaluate reset condition
LT   w_pressure i_75  w_pres_cond 232
OR   r_acc w_pres_cond r_acc 020
```

```
OR    b_lamp b_start  b_lamp 111   //if start pressed set lamp to 1
ANDN  b_lamp r_acc b_lamp 101  //if reset codition met then set lamp to 0
LD    b_lamp 1
IF1JUMP 16 //if lamp not active reset timer
OR   reg_0 i_wait_time w_timer 032
LD    w_timer 2
IF0JUMP 1A //if timer is zero jump over decrementation and set motor to 1
SUB   w_timer b_lamp w_timer 212      //decrement timer
JUMP  0B //start new cycle
SET   b_motor                                          1
///////////////// edge detection ////////////////////////
RST   w_sub2 2   // reset memory to avoid accidental usage of wrong data
RST   r_acc 0    // reset acc for same reason as above
XOR   b_motor w_temp w_sub2 122   // check for edge
AND   w_sub2 b_motor w_sub2 212  // check for posegde
OR    b_motor reg_0 w_temp 102 //save previous motor state, egde detec.
LD    b_ack 1
IF0JUMP 23
OR    reg_0  03 w_cnt 032   // if ack is 1 then reset counter
LD    w_cnt 2
IF0JUMP 26 // if count is not 0 decrement
SUB   w_cnt w_sub2 w_cnt 222  // decrement couter
LD    w_cnt 2
IF1JUMP 0B // if couter not 0 then start new cycle
SET   b_max 1    // if counter 0 set max to 1
JUMP 0B     // start new cycle
```

Program testowy nr 2 – Regulator PID

Aby poznać sterowanie w czasie rzeczywistym, zbudowano 8-bitowy regulator PID. Najpierw zbudowano 32-bitowy oscylator, który ma być sterowany. Oto wyjście oscylatora.

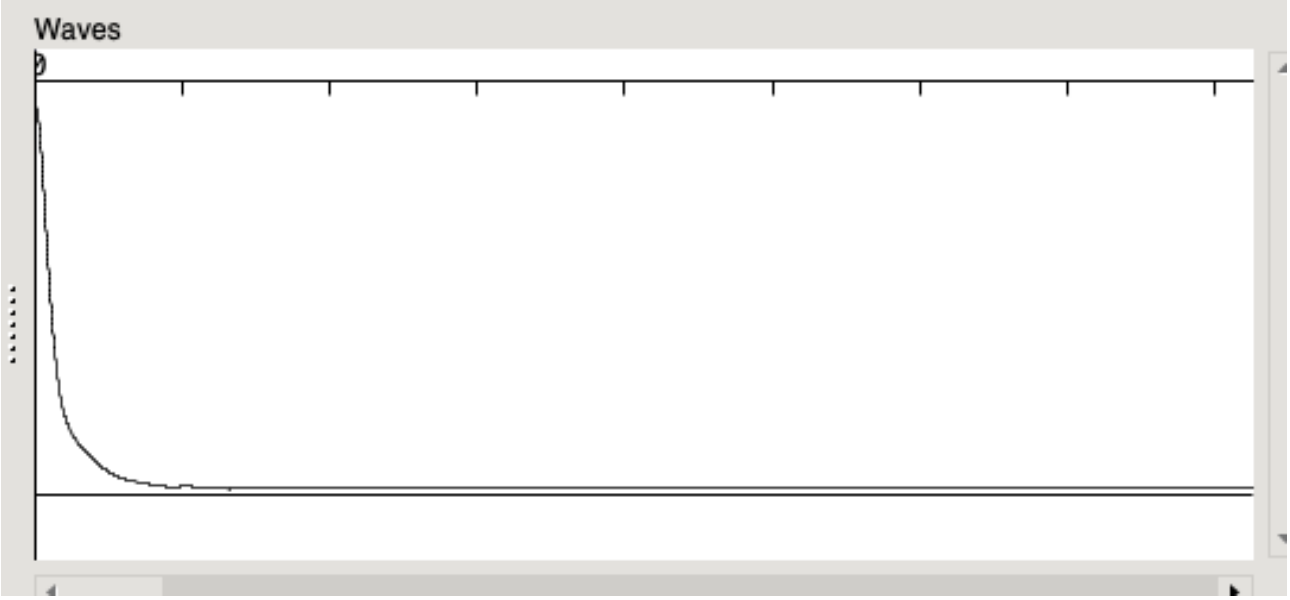


Powyższy wykres jest wyświetlany w GTKWave. Warto zaznaczyć, że GTKWave nie potrafi wyświetlać wartości ujemnych o czym musieliśmy pamiętać przy tworzeniu programu testowego za pomocą naszych instrukcji. Nasz procesor jest procesorem 8-bitowym, więc konieczna była konwersja danych z 32-bitów na 8-bitów w sposób przedstawiony poniżej.

```
assign positionOut = position[31:24];
```

Następnie zbudowano regulator PID. Poniżej znajduje się kod asemblera naszej jednostki. Oczywiście naprawdę trudno jest zbudować regulator PID z zakresem 8 bitów. Ten zakres daje nam tylko +127 do -128. Co gorsza, ze względu na ograniczenia GTKWave, musimy utrzymywać wykres w okolicach wartości 64, co nie daje dużego zakresu kontroli. W szczególności mnożniki składnika całkującego muszą wynosić 0, w przeciwnym razie nastąpi przepełnienie po 2 lub 3 iteracjach. Podobnie człon proporcjonalny jest ograniczony do 1, w przeciwnym razie ponownie doszłoby do przepełnienia. Tylko składnik pochodnej może mieć duży wpływ na zachowanie systemu.

Poniższy wykres przedstawia działanie regulatora PID na zbudowanym wcześniej oscylatorze.



Kod regulatora PID

```
//FIRST DEFINE THE REGISTERS
def positionIn 09
def feedbackOut 08
def position 07
def previousPosition 06
def error 05
def integral 04
def working 03
def result 02
def velocity 01


//NOW DEFINE THE CONSTANTS
def timeStep  01
def setPoint 50  //HEX
def kDerivative FC   //THIS IS -4
def kProportional 01
def kIntegral  00 //0 STEP 0

OR positionIn 00 position 000 //1 STEP 1

//CALCULATE THE PROPORTIONAL TERM
SUB position setPoint error 030   //2 08
MUL error kProportional result 030   //3
```

```
//CALCULATE THE INTEGRAL TERM
MUL integral kIntegral working 030  //4
ADD working result result 000  //5

//CALCULATE THE DERIVATIVE TERM
SUB position previousPosition velocity 000 //6 08 IS THIS CORRECT
DIV velocity timeStep velocity 030     //7 0A
MUL velocity kDerivative  working 030    //8 09
ADD working result result 000    //9 07

//APPLY THE CONTROL SIGNAL
MUL result 01  feedbackOut 030  //10 09

//UPDATE THE INTEGRAL TERM
ADD error integral integral 000

OR position 00 previousPosition 000
JUMP 0
```

8. Wyniki testów

Podczas syntezy za pomocą narzędzia Quartus udało osiągnąć się maksymalną częstotliwość taktowania równą 75 MHz. Przy takim taktowaniu jedna operecja w naszej jednostce zajmuje około 13.4 ns. Jedno cyklowość naszego procesora pozwala na łatwą estymację szybkości wykonywania programów, czasy wykonania programów testowych umieszczono w tabeli poniżej.

Program	Ilość operacji	Czas [ns]	Uwagi
Test nr 1	32	428.8	Czas dla jednego przebiegu pętli
Test nr 2	14	187.6	-

Trzeba podkreślić, że program testowy nr 1 działa w pętli podobnej do tej w jakiej działają sterowniki PLC. Pozwala to na odczyt wejść, wykonanie operacji w odniesieniu do zmian na wejściach i w końcu na ustawienie wyjść. Powyższy czas dotyczy pojedynczej iteracji tej pętli.

9. Podsumowanie

Założenia jakie przyjęliśmy na początku zostały spełnione, procesor ma trójargumentowe instrukcje, które potrzebują dokładnie jednego cyklu zegara na wykonanie. Także instrukcje tej jednostki są zgodne z normą IEC 61131-3.

Wykorzystanie trójargumentowych instrukcji sprawia pisanie programów łatwym i szybkim. Okupione jest to wzrostem wymagań przy doborze pamięci dla jednostki. W naszym procesorze słowo jest wielkości 8 bit. Pojedyncza instrukcja zajmuje w pamięci 40 bitów. Dla obu tych wielkości nie ma problemów ze znalezieniem odpowiednio szerokiej pamięci. Jednakże podczas zwiększania szerokości słowa przyjęte przez nas rozwiązanie może stać się problematyczne. W nowoczesnych procesorach szerokość słowa to przynajmniej 32 bity, zwiększenie słowa do tej wartości w naszej architekturze spowodowało by wydłużenie pojedynczej instrukcji do 108 bitów. Może dojść do sytuacji, że instrukcja będzie zbyt długa aby zmieścić ją w dostępnych modułach pamięci. Przy tworzeniu programu testującego nr 2, ograniczenia słowa 8 bitowego szybko wyszły na jaw. Do optymalnej pracy regulatora PID dobrze było by rozszerzyć słowo jednostki do przynajmniej 24 bitów.

Podsumowując, przyjęte założenia sprawdziły się dobrze dla jednostki o 8 bitowym słowie. Użycie instrukcji z trzema argumentami pozwoliło na zmniejszenie ilości operacji i uproszczenie programów napisanych dla tej architektury.