Raport

PBL

### Christopher Lozinski

Paweł Orszulik

Łukasz Fuss

1. Projektem realizowanym w ramach PBL było stworzenie modelu procesora opierającego się na instrukcjach podanych w normie ( IEC 61131-3: Programming Industrial Automation Systems ) oraz przeprowadzenie jego weryfikacji, syntezy oraz przetestowanie jego działania na DE1-SoC.

**2. Założenia projektowe:**

* instrukcje są trójargumentowe
* zaprojektowany procesor ma być jednocyklowy jeśli ograniczenia technologiczne na to pozwolą

3. **Lista instrukcji zaprojektowanego procesora**





# Instruction Format

Our assignment was to build a 3 argument CPU where each argument could be a register number, a bit memory address, a word memory address, or an immediate value. That calls for 5 fields, a six bit opcode field, an 8 bit first argument field, a second 8 bit argument field, an 8 bit output field, and a six bit type field. Initially we did not have an assembler, so it was most readable to use hex values, which led to 5 fields each of 2 hex or 8 bits wide, for a total of 40 bits. The extra two bits in the choice field proved useful for allowing iteration.

* 8 bitowy kod operacji
* 8 bitowy adres pamięci, rejestru lub wartość danej natychmiastowej (źródło 1)
* 8 bitowy adres pamięci, rejestru lub wartość danej natychmiastowej (źródło 2)
* 8 bitowy adres pamięci lub rejestru (adres zapisu wyniku operacji)
* 2 wolne bity
* 2 bity kodujące miejsce odczytu źródła 1
* 2 bity kodujące miejsce odczytu źródła 2
* 2 bity kodujące miejsce zapisu wyniku operacji

Możliwe kody miejsca odczytu lub zapisu:

* 00 rejestry
* 01 pamięć bitowa
* 10 pamięć słów
* 11 dana natychmiastowa (tylko w przypadku odczytu)

4. Schemat procesora

Obraz zawierający tekst, diagram, Rysunek techniczny, Plan

Opis wygenerowany automatycznie

**5.** **Opis poszczególnych modułów**  
  
**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:

|  |  |
| --- | --- |
| sourceX\_choice | Źródło |
| 00 | Rejestry |
| 01 | Pamięć RAM Bit |
| 10 | Pamięć 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 odpowiednio szybko 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**

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.

**ROM (moduł memory)**

Moduł zawierający instrukcje procesora realizujące zadany program. Instrukcje są zapisany 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 (do zapisu adresów powrotu)**

Stos zachowujący adres powrotu w przypadku wykonania instrukcji call.

**6. Schemat blokowy pierwszego programu testowego.**

Obraz zawierający tekst, zrzut ekranu, wyświetlacz, oprogramowanie

Opis wygenerowany automatycznie

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 for egde detection

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

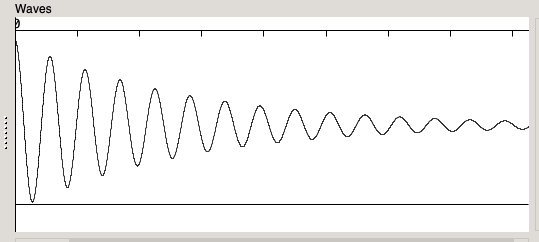
IF1JUMP 0B // if couter not 0 then start new cycle

SET b\_max 1 // if counter 0 set max to 1

JUMP 0B // start new cycle

# PID Controller

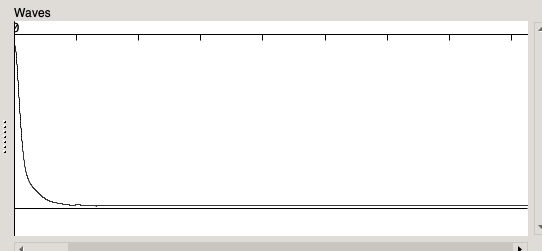
To learn about real time control, an 8 bit PID controller was built. First the 32 bit oscillator to be controlled was built. Here is the output of the oscillator.



This is being display in GTKWave. Of course GTKWave has its limitations, It only displays positive values. Meaning that any negative value would look terrible. But the CPU is an 8 bit CPU, so 32 bit to 8 bit data conversion was needed.

assign positionOut = position[31:24];

Then a PID controller was built. Below is the assembly code. Of course it is really hard to build a PID controller with an 8 bit range. That only gives us +127 to -128. Worse yet, because of the gtkwave limitations, we need to center it on 64 giving not much range of control. Specifically the multipliers for the Integral term has to be 0, or it experiences overflow after 2 or 3 iterations. Similarly the Proportional term is limited to a 1, or again it would experience overflow. Only the derivative term can be a meaningful number.



# PID Controller Assembly

//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

LD positionIn position 00 //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

LD position previousPosition 00

JUMP 0

# Conclusion

It was lots of fun working on this processor. 3 argument instructions make coding very easy. But we advise against it. Most modern cpus are 32 bits wide , and most memories are 16/32/64 bits wide.. With this CPU, one would need 3\*32 bits of instruction just for the addresses. Plus more for the opcodes and choice fields. That is over 108 bit wide instructions, much too wide for most memories.

And while an 8 bit instruction set was fine for the first assignment, it is recommended to use at least 24 bit instructions for PID controllers.

**Podsumowanie**

Każda z zaimplementowanych operacji procesora zajmuje 1 cykl zegara. Przy zaproponowanej przez oprogramowanie Quartus częstotliwości zegara równej 75 MHz. Jedna operacja zajmuje około 13,4 ns (zaokrąglając w górę). Jeden cykl pierwszego programu testowego wymaga wykonania 32 operacji, które zajmą 428.8 ns. Drugi program testowy wymaga wykonania 14 operacji które zajmą 187.6 ns.

# Iterative Output Application

An alternative decentralized design was also written. It allowed for the memory address to be read from a specified register, allowing for itteration 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.

Obraz zawierający tekst, Czcionka, linia, zrzut ekranu

Opis wygenerowany automatycznie

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

.

.

.

# Iterative Output Assembly

LD 1 2 030 //Load a 1 into register 2 the accumulator

LD 1 3 030 //Load a 1 into register 3

LD 1 1 031 //Load a 1 into bitMemory addr 1

LD 1 1 032 //Load a 1 into wordMemory addr 1

MUL 1 1 2 120 //Multiply bit memory 4 by word memory 6 into reg 2 accumulator

LD 2 2 030 //Load a 2 into register 2 the accumulator

LD 2 3 030 //Load a 2 into register 3

LD 1 1 031 //Load a 1 into bitMemory addr 1

LD 2 1 032 //Load a 2 into wordMemory addr 1

MUL 2 2 2 120 //Multiply bit memory 4 by word memory 6 into reg 2 accumulator

LD 3 2 030 //Load a 3 into register 2 the accumulator

LD 3 3 030 //Load a 3 into register 3

LD 0 1 031 //Load a 1 into bitMemory addr 1

LD 3 1 032 //Load a 3 into wordMemory addr 1

MUL 3 3 2 120 //Multiply bit memory 4 by word memory