DOCUMENTAȚIE DE PROIECT DESPRE

IMPLEMENTAREA MICROCONTROLERULUI PICOBLAZE PE 8 BIȚI

STUDENT

AIERIZER SAMUEL

Grupa

30212

PROFESOARĂ

DIANA POP

DATA

20.05.2019

1. Specificaţie proiect

Proiectul constă din implementarea microcontrolerului PicoBlaze pe 8-biţi conform documentaţiei sale. Proiectul implementează toate instrucţiunile de control al programului, de logică, de aritmetică, de şiftare şi rotire şi de input/output. Singurele instrucţiuni ce nu au fost implementate sunt cele legate de modulul de întrerupere.

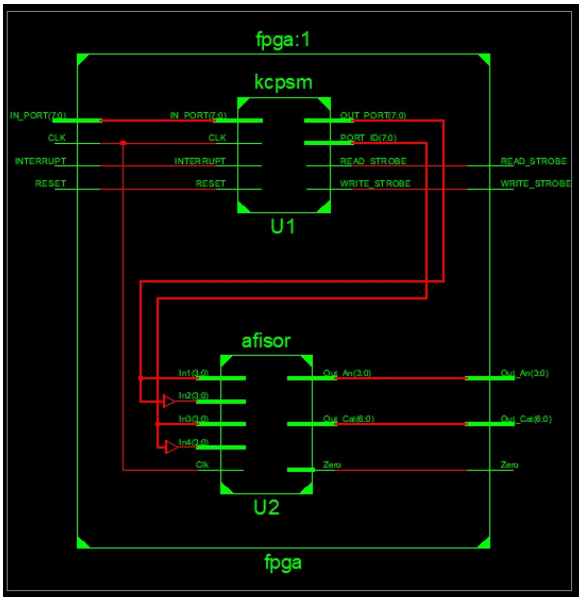
Instrucţiunile de control al programului sunt: Jump, Call şi Return. Acestea pot fi necondiţionate sau condiţionate de flag-ul de Carry sau de Zero. Instrucţiunile logice sunt: Load, And, Or şi Xor, acestea putând fi executate între conţinutul unuia dintre cele 16 registre interne şi o constantă sau cu conţinutul unui alt registru. Instrucţiunile aritmetice sunt: Add, Add cu Carry, Subb şi Subb cu Carry. Acestea pot fi executate în mod similar cu cele logice. Instrucţiunile de şiftare şi rotire sunt: şiftare cu 0, şiftare cu 1, şiftare cu cel mai semnificativ bit, şiftare cu cel mai nesemnificativ bit şi şiftare cu Carry. Acestea pot fi executate în ambele direcţii. Instrucţiunile şi funcţionarea lor specifică se găseşte în documentaţie.

Microcontrolerul stă din 5 părţi majore: memoria de instrucţiuni, blocul de registre, unitate aritmetică-logică, cel de control al programului şi unitatea de comandă. Memoria de instrucţiunii are maxim 256 de instrucţiuni de mărime Word. Blocul de registre conţine 16 registre de 8 biţi. Unitatea aritmetică-logică este responsabilă pentru efectuarea instrucţiunilor logice, aritmetice şi cele de şiftare şi rotire. Unitatea de control al programului este responsabilă pentru schimbarea adreselor pe 8 biţi conform instrucţiunilor de control sau a incrementării în cazul celorlalte instrucţiuni. Unitatea de comandă decodifică instrucţiunile şi trimite semnale de comandă către celelalte componente.

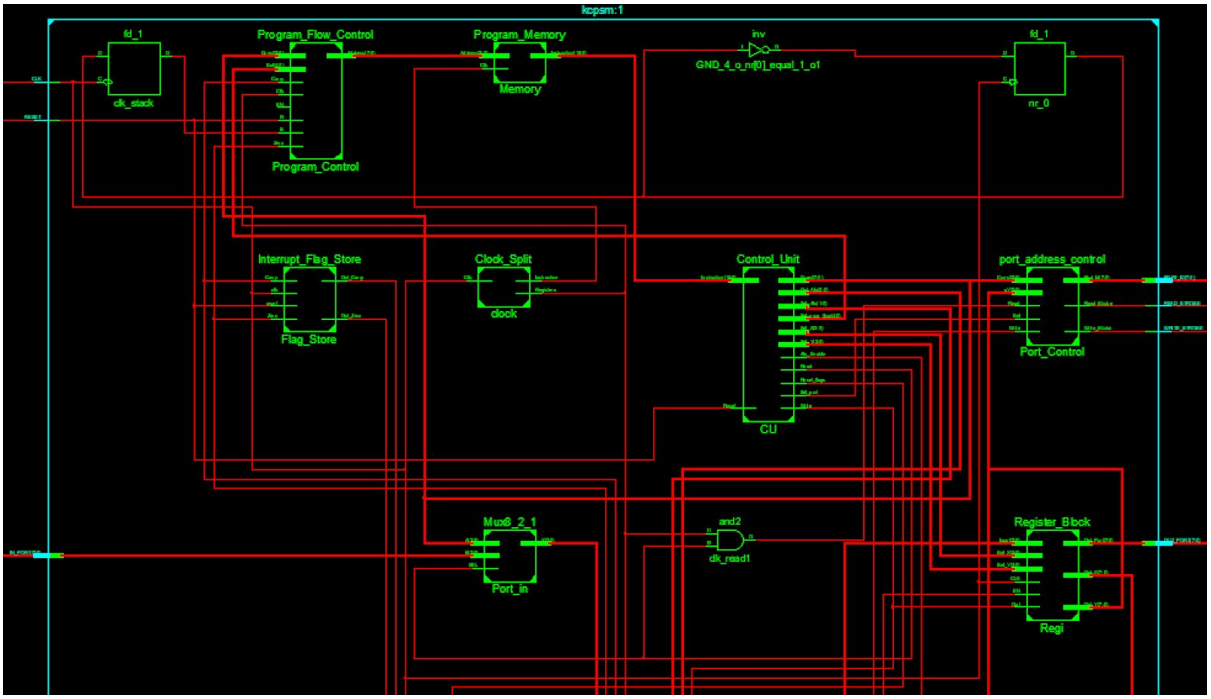
La pornire, microcontrolerul porneşte cu adresa 0 şi după un semnal de tact intră instrucţiunea de pe adresa 0 în unitatea de comandă. Folosim semnalul de tact intern a plăci FPGA. Fiecare instrucţiune este executată în două semnale de tact, astfel asigurând o funcţiune corespunzătoare documentaţiei.

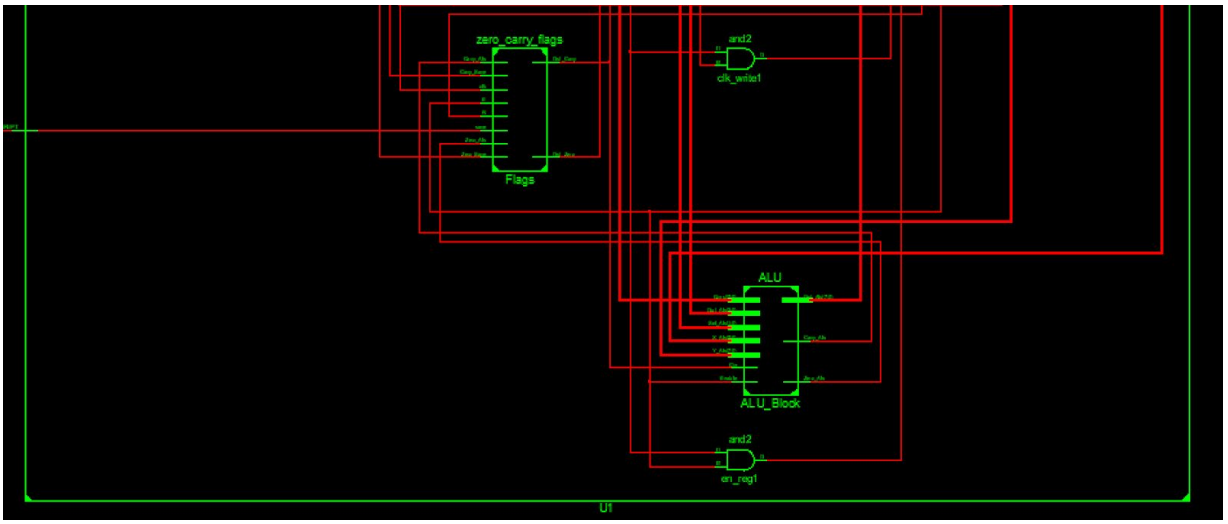
2. Schemă bloc

Schema bloc top-level constă din microcontrolerul PicoBlaze care este prescurtat drept kcpsm (Constant (k) Coded Programmable State Machine) și din blocul de afișare, care este responsabil pentru afișarea ieșirilor pe afișoarele pe 7 segmente.



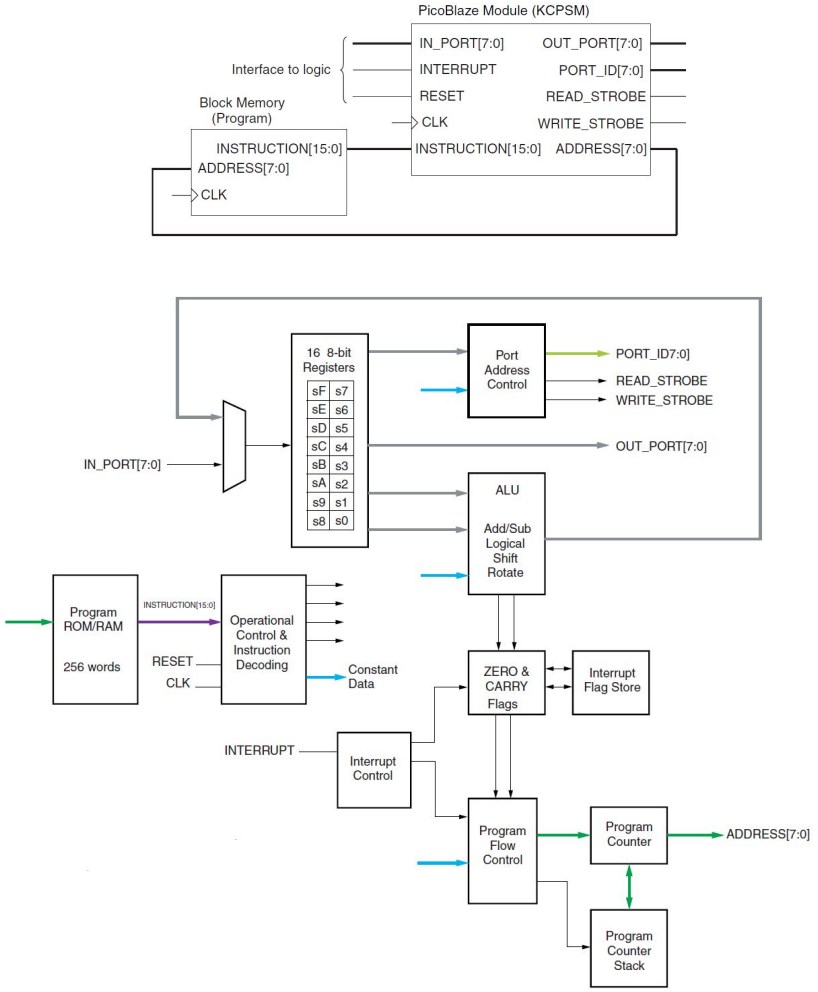
Schema bloc a microcontrolerului este următoarea:





3. Proiectare și implementare

Schema bloc a microcontrolerului este o implementare specializată a celei găsite în documentație.

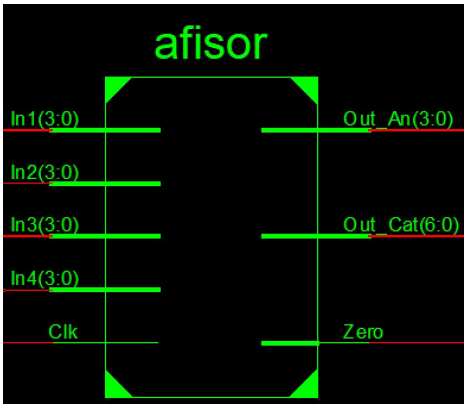


Implementarea mea este diferită prin implementarea controlului de program ca find o singură componentă în loc de 3. Această decizie a fost pentru a face mai uşoară utilizarea celor trei componente şi reduce logica separată din top-level.

Schema a fost dată în proiect, astfel partea mea a fost doar proiectarea funcţionalităţii acestor componente şi sincronizarea lor conform modului de funcţionare specificat.

4. Lista componentelor utilizate și Semnificația semnalelor

1. Afişor

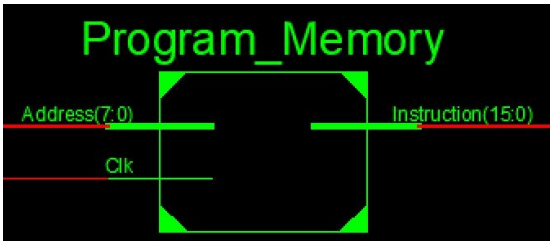


Intrări: In1, In2, In3, In4 sunt intrări std_logic_vector(3 downto 0). Acestea vor fi difuzate pe afişorul pe 7 segmente. Clk este std_logic, este semnalul de tact pentru sincronizarea componentei.

Ieşiri: Out_Cat este ieşirea std_logic_vector(6 downto 0) pentru cei 7 catozi. Out_An este ieşirea std_logic_vector(3 downto 0) pentru anodul comun.

2. Kcpsm

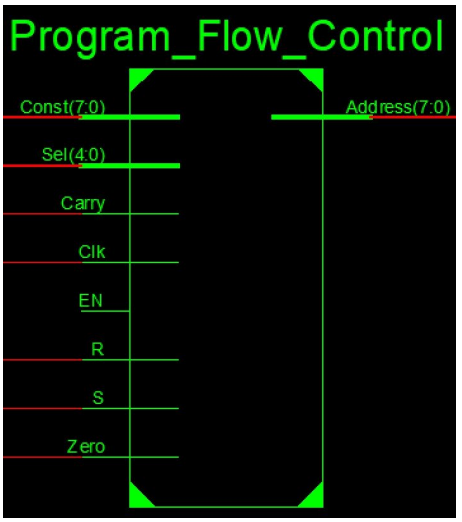
2.1. Program Memory



Intrări: Address este std_logic_vector(7 downto 0), ce arată adresa ce urmează. Clk este semnalul de sincronizare a componentei.

Ieşiri: Are o singură ieşire, aceasta este std_logic_vector(15 downto 0), ce este instrucţiunea ce urmează să fie executată.

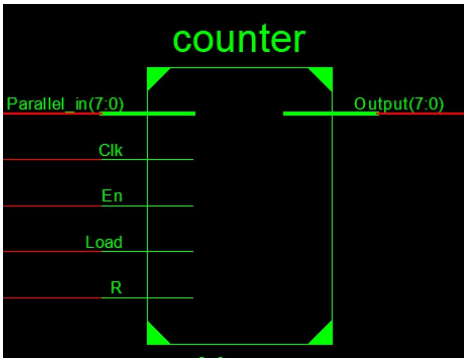
2.2. Program Flow Control



Intrări: Intrarea Const std_logic_vector(7 downto 0) este constantă ce specifică adresa la care va rezulta instrucţiunea call sau jump. Semnalul Sel care este std_logic_vector(4 downto 0) specifică ce comandă se execută. Cel mai semnificativi 2 biţi sunt pentru cele 4 feluri de găsire a adresei: incrementare, jump, call şi return. Bitul mijlociu arată dacă instrucţiunea este condiţionată, sau nu, restul arătând dacă este Z, NZ, C, NC. Carry şi Zero intră din componenta Zero Carry Flags. Enable, Reset, S şi Clock sunt necesare pentru sincronizarea şi resetarea componentei.

Ieşiri: Singura adresă este cea validă cu care va continua execuţia instrucţiunilor.

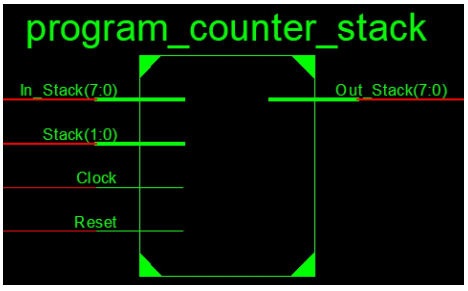
2.2.1. Counter



Intrări: Parralel_in este std_logic_vector(7 downto 0) pentru a putea executa comenzile mai complexe ale componentei de control a programului. Clk, En, Load şi R sunt folosite pentru sincronizarea componentei.

Ieşiri: Output va fi adresa instrucţiunii următoare şi în caz de call, valoare incrementată a acesteia va fi pusă pe stivă

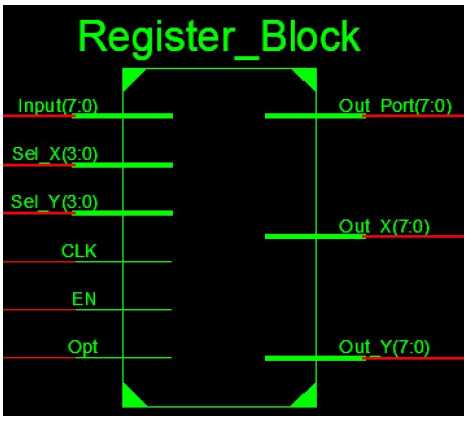
2.2.2. Program Counter Stack



Intrări: In_Stack este vector de std_logic şi este valoarea ceva fi stocată pe stivă. Stack este şi enable-ul componentei (Stack(1)) şi arată şi dacă se va face push sau pop (Stack(0)). Restul componentelor sunt pentru sincronizare.

Ieşiri: Out_Stack este valoarea elementului scos de pe stivă ce va fi pus în numărător cu parallel load.

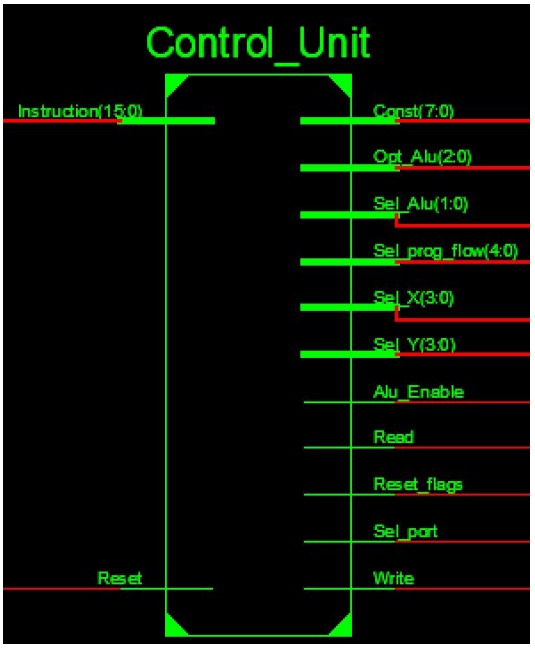
2.3. Register Block



Intrări: Intrarea Input este cea care se va scrie în registrul selectat de Sel_X. Sel_X şi Sel_Y specifică care dintre registre vor fi folosite. Opt arată dacă valoarea din registru va intra în ALU sau va fi scrisă pe Out_Port. Clk şi EN sunt folosite la sincronizare.

Ieşiri: Ieşirea Out_Port este semnal de ieşire din microcontroler. Ieşirile Out_X şi Out_Y sunt valori de pe registrele X şi Y selectate.

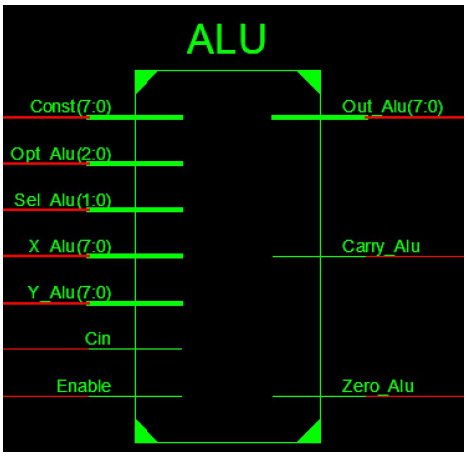
2.4. Control Unit



Intrări: Are ca intrare doar instrucţiunea pe 16 biţi şi semnalul de Reset.

Ieşiri: Const este ieşirea de constantă, Sel_X şi Sel_Y sunt selecţii pentru blocul de registre. Sel_Alu şi Opt_Alu specifică ce fel de instrucţiune se va executa şi Alu_Enable este folosit la blocarea valorilor din ALU când aceasta nu este folosită. Sel_prog_flow este pentru controlul de program. Reset_flags este folosită la interrupt. Write, Read şi Sel_port sunt pentru Port Address Control, ca Port_Id să afişeze corect informaţia.

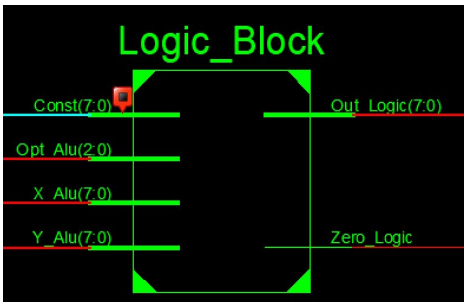
2.5. ALU



Intrări: Const, X_Alu şi Y_Alu sunt valorile cu ce se vor executa instrucţiunile din ALU. Sel_Alu specifică dacă este folosită componenta logică, aritmetică, şiftare dreapta sau şiftare stânga. Opt_Alu specifică ce se va folosi din acestă componentă. Cin este Carry-ul precedent şi Enable pentru sincronizare.

Ieşiri: Out_Alu este rezultatul operaţiei executate. Carry_Alu şi Zero_Alu sunt valori noi ale flag-urilor.

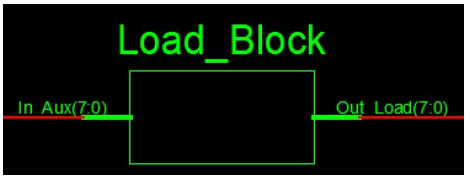
2.5.1. Bloc Logic



Intrări: Const, X_Alu şi Y_Alu sunt valorile cu ce se vor executa instrucţiunile din ALU. Opt_Alu specifică în acest caz dacă se va executa Load, And, Or sau Xor.

Ieşiri: Out_Logic este rezultatul operaţiei executate. Zero_Logic este noua valoare a flagului.

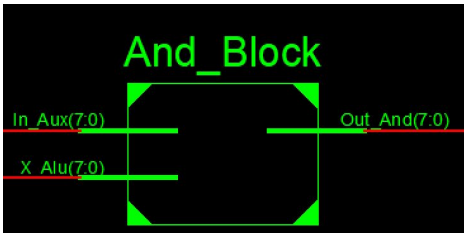
2.5.1.1. Load



Intrări: In_Aux va fi valoarea ce va fi pusă pe ieşire.

Ieşiri: Out_Load este valoarea ce se trimite la registru.

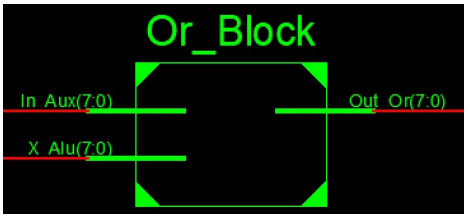
2.5.1.2. And



Intrări: In_Aux şi X_Alu sunt valorile între care se va executa And pe biţi.

Ieşiri: Out_And este valoarea ce se trimite la registru.

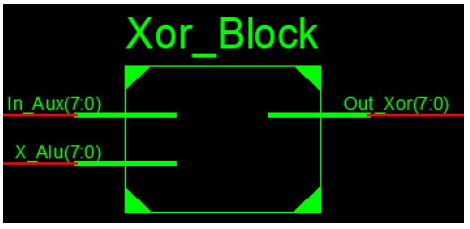
2.5.1.3. Or



Intrări: In_Aux şi X_Alu sunt valorile între care se va executa Or pe biţi.

Ieşiri: Out_Or este valoarea ce se trimite la registru.

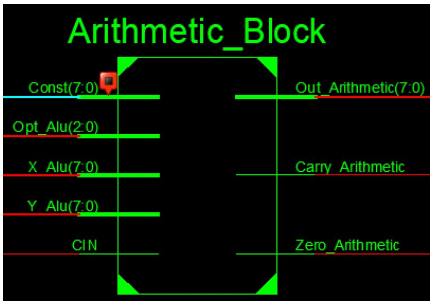
2.5.1.4. Xor



Intrări: In_Aux şi X_Alu sunt valorile între care se va executa Xor pe biţi.

Ieşiri: Out_Xor este valoarea ce se trimite la registru.

2.5.2. Bloc Arithmetic

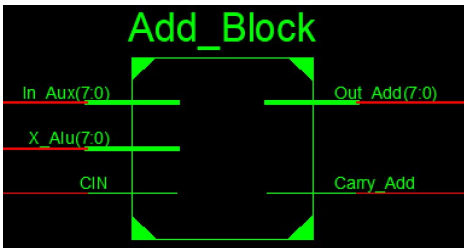


Intrări: Const, X_Alu şi Y_Alu sunt valorile cu ce se vor executa instrucţiunile din blocul aritmetic. Opt_Alu

specifică dacă se va folosi Add, AddCy, Subb, SubbCy. Cin este Carry-ul precedent.

Ieşiri: Out_Arithmetic este rezultatul operaţiei executate. Carry_Arithmetic şi Zero_Arithmetic devin noile valori ale flag-urilor.

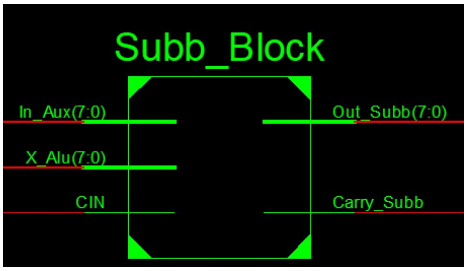
2.5.2.1. Add



Intrări: In_Aux şi X_Alu sunt valorile ce vor fi însumate. Cin este Carry-ul precedent ce se foloseşte doar la însumarea cu Carry.

Ieşiri: Out_Add este rezultatul însumării şi Carry_Add este noul Carry.

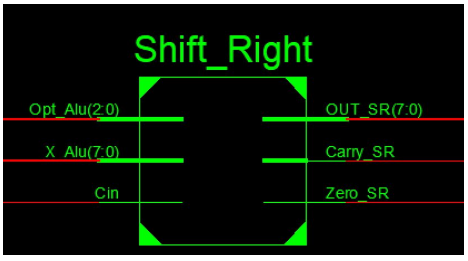
2.5.2.2. Subb



Intrări: In_Aux şi X_Alu sunt valorile ce vor fi scăzute. Cin este Carry-ul precedent ce se foloseşte doar la scăderea cu Carry.

Ieşiri: Out_Subb este rezultatul însumării şi Carry_Subb este noul Carry.

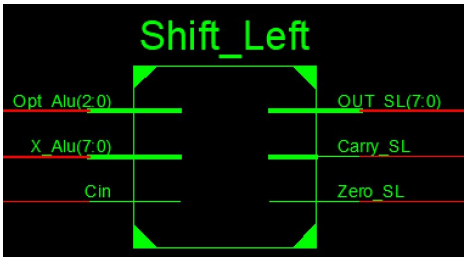
2.5.3. Şiftare Dreapta



Intrări: X_Alu este valoarea pe care se execută instrucţiunea. Cin este Carry-ul precedent. Opt_Alu specifică ce fel de şiftare sau rotire se va executa.

Ieşiri: Out_SR este rezultatul şiftării/rotirii şi Carry_SR şi Zero_SR devin noile valori de flag.

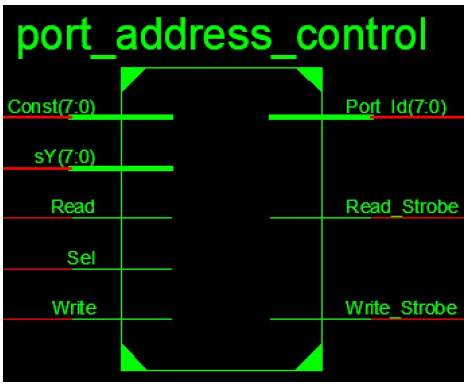
2.5.4. Şiftare Stânga



Intrări: X_Alu este valoarea pe care se execută instrucţiunea. Cin este Carry-ul precedent. Opt_Alu specifică ce fel de şiftare sau rotire se va executa.

Ieşiri: Out_SL este rezultatul şiftării/rotirii şi Carry_SL şi Zero_SL devin noile valori de flag.

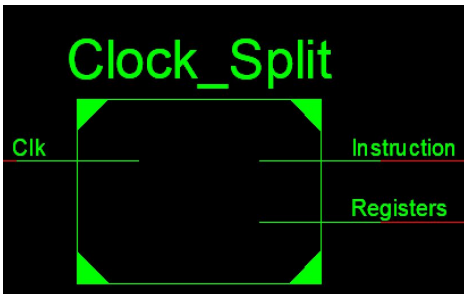
2.6. Port Address Control



Intrări: Const şi sY sunt valorile ce vor fi scrise pe Port_Id şi Sel este semnalul de select ce va determina care dintre cele două se scrie. Read şi Write sunt pentru activarea ieşirilor în moment potrivit.

Ieşiri: Port_Id este una dintre cele două intrări pe 8 biţi. Read_Strobe este '1' pe durata al doilea clock când se execută Input, iar Write_Strobe este '1' pe durata al doilea clock când se execută Output. Altfel aceste ieşiri sunt '0'.

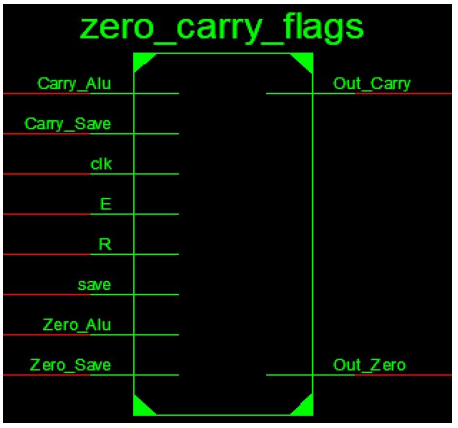
2.7. Clock Split



Intrări: Parte foarte importantă pentru buna funcţiune a microcontrolerului. Intrarea Clk este semnalul de tact global ce urmează să fie "despărţit" în două.

Ieşiri: Instruction este ieşirea ce va fi tact pentru memoria de instrucţiuni şi Registers devine tact pentru blocul de registre.

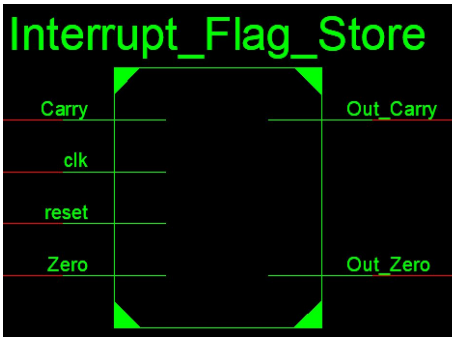
2.8. Zero Cary Flags



Intrări: Carry_Alu şi Zero_Alu sunt valorile noi ce se vor scrie peste cele vechi cînd save este '0'. Altfel Zero_Save şi Carry_Save vor fi folosite. R este pentru resetare, Clk şi E sunt pentru sincronizare.

Ieşiri: Out_Carry şi Out_Zero sunt noile valori de flag.

2.9. Interrupt Flag Store



Intrări: Carry şi Zero sunt salvate la frontul crescător de clk. Reset este pentru resetare

Ieşiri: Out_Carry şi Out_Zero sunt noile valori de flag salvate pentru interrupt.

5. Justificarea soluției alese

Am ales să implementez majoritatea componentelor într-un mod structural. Unde era posibil am încercat să implementez structural sau cu flux de date. Acest lucru se vede şi la numărul de slice-uri utilizate la care am putut reduce implementare.

Din cauza lipsei unui partener la proiect nu am fost în stare să implementez instrucţiunile de interrupt şi în anumite locuri (exemplu numărătorul din Program Flow Control) aş fi putut face structural. Dat find resursele limitate am încercat să implementez cât mai bine posibil acest microcontroler.

Am ales ca stiva să meargă pe un alt tact ca toate celelalte componente. Motivul este că la instrucţiunea de return se scotea valoarea din stivă în acelaşi moment când ar fi trebuit să fiu pus în numărător. Asta a dus la setarea numărătorului într-un mod neadecvat. Aducerea tactului de stivă la frontul descrescător precedent tactului de schimbare a valorii numărătorului a rezolvat problema. Astfel valoarea din stivă este scoasă mai repede şi ea ajunge pe semnalul Parallel_in al numărătorului ca să intre în acesta corespunzător.

Decizia de a folosi doar un semnal de constantă şi de a folosi enable-uri şi semnale de tact diferite a condus la un număr redus de resurse utilizate. Numărul de semnale trimise la ALU, blocul de registre sau controlul de program sunt strictul necesar pentru a codifica fiecare instrucţiune posibilă.

La controlul de program am ales codificare pe 5 biţi, în ciuda faptului că s-ar fi putut codifica doar pe 4, astfel folosind un bit de semnal mai puţin. Însă asta ar fi presupus un efort mai mare din punctul de vedere a logici folosite pentru codificarea stărilor. Decizia de a folosi 5 biţi a dus la o logică mult simplificată, astfel a redus şi efortul depus şi a redus "consumul" codificării în acelaşi timp.

Din păcate nu am reuşit să rezolv fiecare warning ce îmi dă programul ISE Design Suit, însă pe parcursul proiectării şi depanării am încercat să reduc pe cât posibil numărul acestora. Alte alegeri pe care le-am făcut şi nu le-am menţionat aici, au fost făcute luând în vedere costul şi complexitatea implementării.

6. Utilizare și rezultate

Resurse folosite

Documentaţia microcontrolerului PicoBlaze pe 8 biţi:
http://users.utcluj.ro/~baruch/resources/PicoBlaze/xapp213.pdf

Resurse necesare

Pentru utilizarea acestui proiect este nevoie de un FPGA Nexys4. Alte modele pot funcţiona, însă trebuie modificat fişierul fpga.ufc în mod corespunzător.

Descrierea utilizării

În primul rând se deschide fiţierul Program_Memory.vhdl din folderul /src. Se modifică memoria de instrucţiuni în modul dorit şi se salvează schimbările. Pentru a efectua aceste schimbări în mod corespunzător, utilizatorul are nevoie de documentaţie pentru a vedea forma corectă a instrucţiunilor.

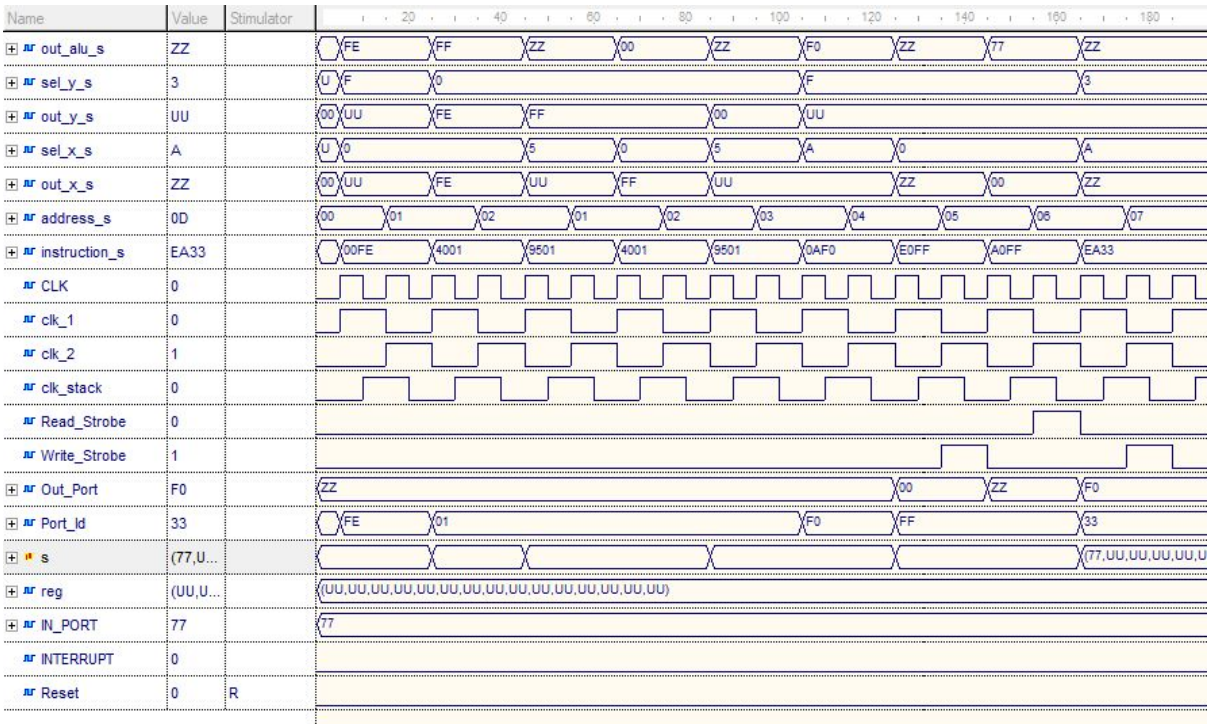
Utilizatorul are nevoie de programul ISE Design Suit sai Vivado şi de o placă FPGA. Va trebui modificat fpga.ufc în cazul folosirii la FPGA diferit de Nexys4. Se va programa dispozitivul FPGA şi se va folosi microcontrolerul.

Este posibilă şi ducerea proiectului la o fabrică de printat circuite şi implementarea microcontrolerului cu un set de instrucţiuni neschimbabile.

Rezultate obţinute

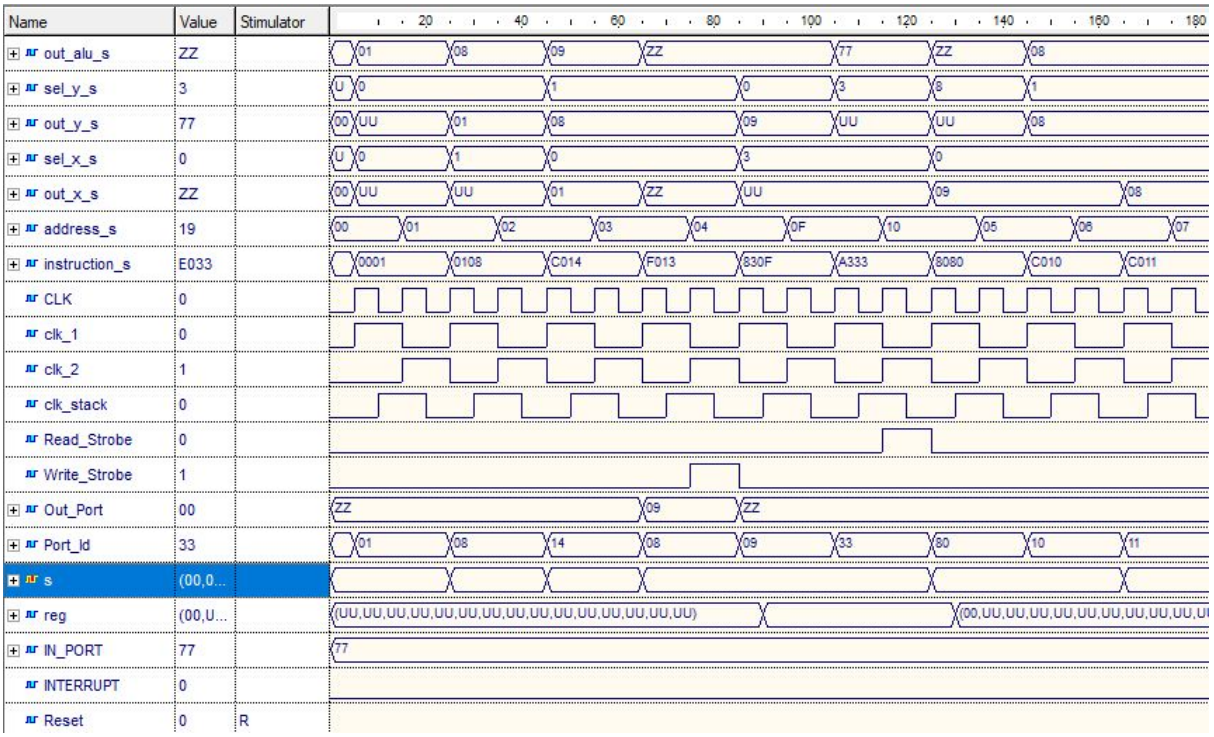
Simularea codului 1:

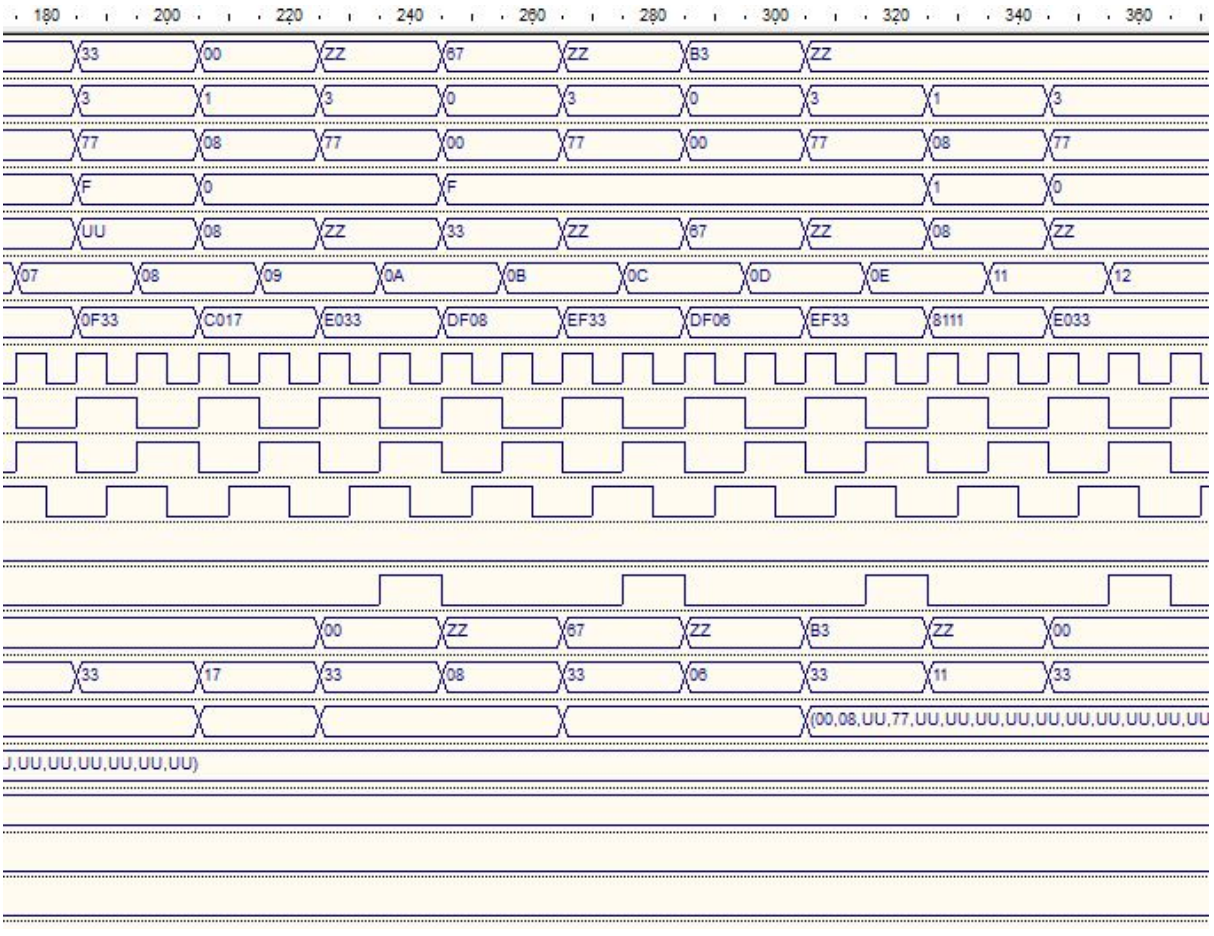
```
0 => "0000000011111110",          --  00      Load    s0, FE      0001
1 => "0100000000000001",          --  01      Add     s0, 01      40FE
2 => "1001010100000001",          --  02      Jump    NZ, 01      9901
3 => "0000101011110000",          --  03      Load    sA, F0      0AF0
4 => "1110000011111111",          --  04      Output  s0, FF      E0FF
5 => "1010000011111111",          --  05      Input   s0, FF      A0FF
others => "1110101000110011");    --          Output  sA, 33      EA33
```



Simularea codului 2:

```
0 => "0000000000000001",          -- 00      Load    s0, 01      0001
1 => "0000000100001000",          -- 01      Load    s1, 08      0108
2 => "1100000000010100",          -- 02      Add     s0, s1      C014
3 => "1111000000010011",          -- 03      Output  s0, s1      E013
4 => "1000001100001111",          -- 04      Call    0F          830F
5 => "1100000000010000",          -- 05      Load    s0, s1      C010
6 => "1100000000010001",          -- 06      And     s0, s1      C011
7 => "0000111100110011",          -- 07      Load    sF, 33      0F33
8 => "1100000000010111",          -- 08      Subcy   s0, s1      C017
9 => "1110000000110011",          -- 09      Output  s0, 33      E033
10 => "1101111100001000",         -- 10      Sra     sF          DF08
11 => "1110111100110011",         -- 11      Output  sF, 33      EF33
12 => "1101111100000110",         -- 12      S10     sF          DF06
13 => "1110111100110011",         -- 13      Output  sF, 33      EF33
14 => "1000000100010001",         -- 14      Jump    11          8111
15 => "1010001100110011",         -- 15      Input   s3, 33      A333
16 => "1000000010000000",         -- 16      Return              8080
others => "1110000000110011");    --         Output  s0, 33      E033
```


Name
Value
Stimulator
20
40
60
80
100
120
140
160
180
out_alu_s
ZZ
01
08
09
ZZ
77
ZZ
08
sel_y_s
3
U
0
1
0
3
6
1
out_y_s
77
00
UU
01
08
09
UU
UU
08
sel_x_s
0
U
0
1
0
3
0
out_x_s
ZZ
00
UU
UU
01
ZZ
UU
09
08
address_s
19
00
01
02
03
04
0F
10
05
06
07
instruction_s
E033
0001
0108
C014
F013
830F
A333
8080
C010
C011
CLK
0
clk_1
0
clk_2
1
clk_stack
0
Read_Strobe
0
Write_Strobe
1
Out_Port
00
ZZ
09
ZZ
Port_Id
33
01
08
14
08
09
33
80
10
11
s
(00,0...
reg
(00,U...
(UU,UU,UU,UU,UU,UU,UU,UU,UU,UU,UU,UU,UU,UU,UU,UU)
IN_PORT
77
77
INTERRUPT
0
Reset
0
R


180
200
220
240
260
280
300
320
340
360
33
00
ZZ
67
ZZ
B3
ZZ
3
1
3
0
3
0
3
1
3
77
08
77
00
77
00
77
08
77
F
0
F
1
0
UU
08
ZZ
33
ZZ
67
ZZ
08
ZZ
07
08
09
0A
0B
0C
0D
0E
11
12
0F33
C017
E033
DF08
EF33
DF06
EF33
8111
E033
00
ZZ
67
ZZ
B3
ZZ
00
33
17
33
08
33
06
33
11
33
(00,08,UU,77,UU,UU,UU,UU,UU,UU,UU,UU,UU,UU
J,UU,UU,UU,UU,UU,UU)

7. Posibilităţi de dezvoltare

Proiectul ar putea fi îmbunătăţit în mai multe feluri. Implementarea componentelor comportamentale într-un mod structural este o modalitate foarte bună de a îmbunătăţi funcţionarea şi costul microcontrolerului.

Implementarea instrucţiunilor de interrupt ar face mai utilizabilă acest dispozitiv şi ar face ca proiectul să fiu complet din punctul de vedere al instrucţiunilor.

S-ar putea lucra la reducerea warning-urilor la sinteza proiectului în ISE Design Suit, ce ar duce la o funcţionare mai bună.

Memoria ar putea fi mărită, astfel ar permite scrierea unor seturi de instrucţiuni mult mai complexe.

S-ar putea schimba structura microcontrolerului şi s-ar putea transforma într-unul de 16 biţi în loc de 8. Aceasta ar putea fi un proiect separat în sine.