General Purpose Processor
Proiect
Fundamente de ingineria calculatoarelor

Echipa "Carry Home"

- Project Manager: Efrem Dragoș-Sebastian-Mihaly
- Hardware Desgin: Gostian Loredana și Bărbosu Bianca-Florentina
- Software: Gabor Sebastian, Farcaș Adrian-Tiberiu și Goșa Gheorghe-Cristian
- Testing: Gheorghe Iuliana

Link către proiect pe GitHub: `https://github.com/Carry-Home/gpp_calc`

Cuprins

Listă de tabele

Listă de figuri

1 Introducere

Un procesor de uz general este un procesor care nu are un scop anume. El poate fi folosit în diverse aplicații și scopuri și poate fi particularizat pentru o anumită aplicație. Proiectul nostru constă în sintetizarea unui procesor de uz general care să permită operațiile găsite într-un calculator de buzunar obișnuit.

2 Setul de instrucțiuni

Numele instrucțiunii	***Descriere scurtă***	***Opcode <6>***	***Biții rămași <10>***
TRX	Transferă valoarea din ACC în X	000000	0000000000
TRY	Transferă valoarea din ACC în Y	000001	0000000000

Tabela 1: Operații de transfer acumulator - registru

Numele instrucțiunii	***Descriere scurtă***	***Opcode <6>***	***Adresă registru <1>***	***Immediate <9>***
LDR	Încarcă din memorie de la adresa specificată la Immediate în registrul specificat de adresa (0 - X, 1- Y)	000010	<1>	<9>
STR	Încarcă în memorie la adresa specificată la Immediate valoarea registrului specificat de adresa (0 - X, 1 - Y)	000011	<1>	<9>

Tabela 2: Operații care folosesc memoria

Numele instrucțiunii	***Descriere scurtă***	***Opcode <6>***	***Adresă registru <2>***	***Immediate <8>***
PSH	Încarcă în memorie la adresa specificată de Stack Pointer valorea registrului specificat la adresa (00 - X, 01- Y, 10 - ACC, 11 - PC)	000100	<2>	0000 0000
POP	Încarcă din memorie de la adresa specificată de Stack Pointer în registrului specificat de adresa (00 - X, 01 - Y, 10 - ACC, 11 - PC)	000101	<2>	0000 0000

Tabela 3: Operații care folosesc stiva

Numele instrucțiunii	***Descriere scurtă***	***Opcode <6>***	***Adresa de branch <10>***
BRZ	Se actualizează Program Counter cu adresa de branch specificată de cei mai nesemnificativi 10 biți doar dacă indicatorul Zero este activ	000110	<10>
BRN	Se actualizează Program Counter cu adresa de branch specificată de cei mai nesemnificativi 10 biți doar dacă indicatorul Negative este activ	000111	<10>
BRC	Se actualizează Program Counter cu adresa de branch specificată de cei mai nesemnificativi 10 biți doar dacă indicatorul Carry este activ	001000	<10>
BRO	Se actualizează Program Counter cu adresa de branch specificată de cei mai nesemnificativi 10 biți doar dacă indicatorul Overflow este activ	001001	<10>
BRA	Se actualizează Program Counter cu adresa de branch specificată de cei mai nesemnificativi 10 biți	001010	<10>

Numele instrucțiunii	*Descriere scurtă*	*Opcode <6>*	*Adresa de branch <10>*
JMP	Se apelează procedura aflată la adresa specificată de cei mai nesemnificativi 10 biți.	2*	JMP și RET vor fi implementate ca pseudoinstrucțiuni. Un JMP [adr] va genera PSH PC și BRA [adr] iar RET va genera un POP PC. Doar instrucțiunea RET va putea face POP PC, iar un POP PC va actualiza Program Counter dar îl va și incrementa cu 2 (POP incrementează cu 2 doar PC)
RET	Se revine din procedură executându-se instrucțiunea următoare celei de după apelul procedurii		

Tabela 4: Operații de salt condiționat și necondiționat

Numele instrucțiunii	*Descriere scurtă*	*Opcode <6>*	*Adresă registru <1>*	*Immediate <9>*
ADD	Dacă Immediate este 0, atunci se realizează operația ACC = ACC + X sau ACC = ACC + Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația X = X + Immediate sau Y = Y + Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	001101	<1>	<9>
SUB	Dacă Immediate este 0, atunci se realizează operația ACC = ACC - X sau ACC = ACC - Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația X = X - Immediate sau Y = Y - Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	001110	<1>	<9>
LSR	Dacă Immediate este 0, atunci se realizează operația ACC = ACC >>X sau ACC = ACC >> Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația X = X >>Immediate sau Y = Y >>Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	001111	<1>	<9>
LSL	Dacă Immediate este 0, atunci se realizează operația ACC = ACC <<X sau ACC = ACC <<Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația X = X <<Immediate sau Y = Y <<Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	010000	<1>	<9>
MUL	Dacă Immediate este 0, atunci se realizează operația ACC = ACC * X sau ACC = ACC * Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația X = X * Immediate sau Y = Y * Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	010001	<1>	<9>
DIV	Dacă Immediate este 0, atunci se realizează operația ACC = ACC / X sau ACC = ACC / Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația X = X / Immediate sau Y = Y / Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	010010	<1>	<9>

Numele instrucțiunii	*Descriere scurtă*	*Opcode <6>*	*Adresă registru <1>*	*Immediate <9>*
MOD	Dacă Immediate este 0, atunci se realizează operația ACC = ACC % X sau ACC = ACC % Y, în funcție de adresa registrului (0 - X , 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația X = X % Immediate sau Y = Y % Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	010011	<1>	<9>
CMP	Dacă Immediate este 0, atunci se compară ACC cu X sau cu Y și se setează flagurile corespunzătoare. Dacă Immediate este diferit de 0, atunci se compară X sau Y cu Immediate și se setează flagurile corespunzătoare.	010100	<1>	<9>
INC	Se incrementează valoarea din registrul specificat prin adresă (0 - X, 1 - Y).	010101	<1>	000000000
DEC	Se decrementează valoarea din registrul specificat prin adresă (0 - X, 1 - Y).	010110	<1>	000000000
AND	Dacă Immediate este 0, atunci se realizează operația ACC = ACC & X sau ACC = ACC & Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația X = X & Immediate sau Y = Y & Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	010111	<1>	<9>
OR	Dacă Immediate este 0, atunci se realizează operația ACC = ACC — X sau ACC = ACC — Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația X = X — Immediate sau Y = Y — Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	011000	<1>	<9>
XOR	Dacă Immediate este 0, atunci se realizează operația ACC = ACC ^X sau ACC = ACC ^Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația X = X ^Immediate sau Y = Y ^Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	011001	<1>	<9>
NOT	Se inversează biții registrului specificat prin adresă (0 - X, 1 - Y).	011010	<1>	0000000000
RSR	Dacă Immediate este 0, atunci se realizează operația de rotire la dreapta al lui ACC cu atâtea poziții cât este specificat in X sau în Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația de rotire la dreapta al lui X sau Y cu atâtea poziții cât este specificat în Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	011011	<1>	<9>
RSL	Dacă Immediate este 0, atunci se realizează operația de rotire la stânga al lui ACC cu atâtea poziții cât este specificat in X sau în Y, în funcție de adresa registrului (0 - X, 1 - Y). Dacă Immediate este diferit de 0, atunci se realizează operația de rotire la stânga al lui X sau Y cu atâtea poziții cât este specificat în Immediate, în funcție de adresa registrului (0 - X, 1 - Y).	011100	<1>	<9>
FCT	Se realizează operația de factorial a valorii din Immediate și rezultatul este pus în X sau Y, în funcție de adresa registrului (0 - X, 1 - Y).	011101	<1>	<9>

Tabela 5: Operații aritmetice și logice

Numele instrucțiunii	*Descriere scurtă*	*Opcode <6>*	*Adresă registru <1>*	*Immediate <9>*
MOV	Mută valorea dată prin Immediate în registrul X sau Y în funcție de adresă (0 - X, 1 - Y).	011110	<1>	<9>

Tabela 6: Operații de mutare în registru

Numele instrucțiunii	*Descriere scurtă*	*Opcode <6>*	*Biții rămași <1>*
NOP	Generează un delay de un ciclu de clock	111111	0000000000

Tabela 7: Instrucțiune care nu execută nicio operație

3 Design hardware

3.1 Schema procesorului

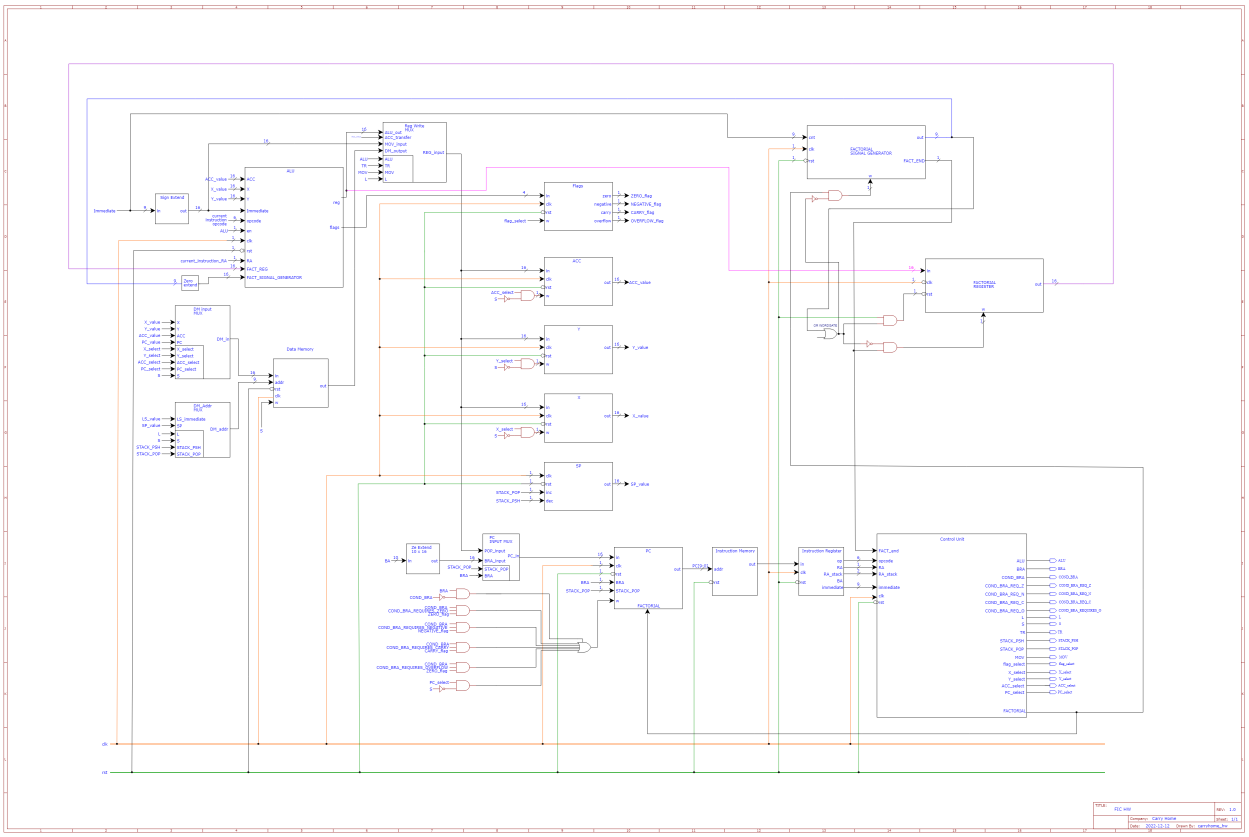

Figura 1: Schema procesorului

3.2 Detalierea componentelor

Procesorul nostru este alcătuit din 17 module care urmează să fie detaliate.

3.2.1 Unitatea de control

Unitatea de control reprezintă creierul operațiunilor pe care procesorul le efectuează. Aceasta trebuie să asigure coerența celor 19 semnale de control pe care le transmite la ieșire.

Are o intrare de opcode pe care o decodifică. Împreună cu intrări de decizie cum ar fi RA, RA_stack, respectiv Immediate, activează semnalele de control corespunzătoare.

Semnalul $FACT\_END$ este folosit ca intrare în Control Unit pentru a dezactiva modulul factorial, respectiv a pune semnalul de control FACT pe 0, atunci când instrucțiunea factorial ce s-a desfășurat și-a terminat execuția, aceasta fiind singura instrucțiune care se desfășoară pe mai mult de un ciclu de clock.

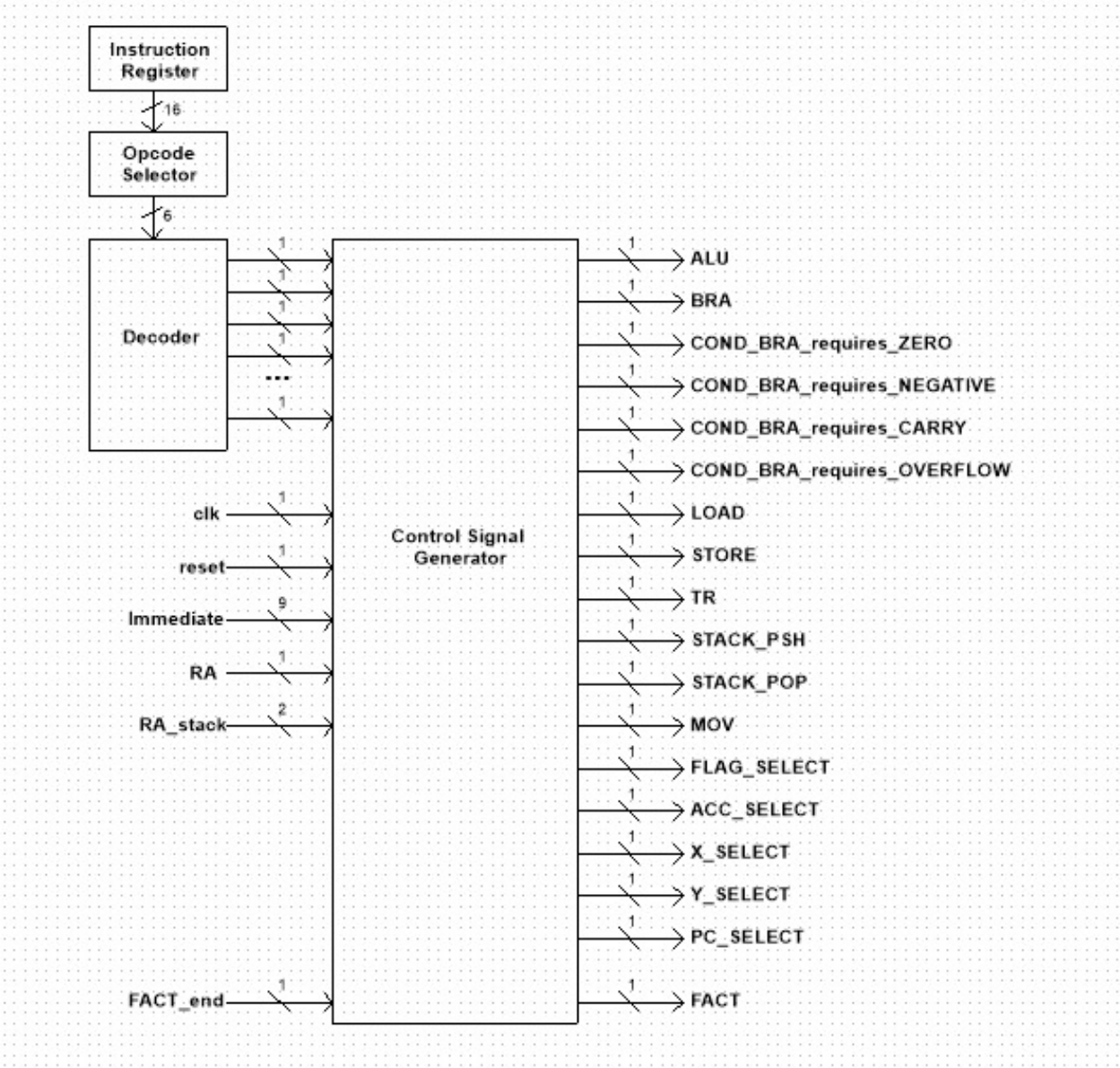

Figura 2: Schema de bază a unității de control

Așadar, următoarele semnale de control se activează în cazurile:

- ALU: se pune pe 1 atunci când se execută o instrucțiune aritmetică sau logică, respectiv reprezintă intrarea de enable de la ALU;
- BRA: se pune pe 1 atunci cand se execută o instrucțiune de branching (condiționat sau necondiționat);
- COND_BRA: se pune pe 1 atunci când se execută o instrucțiune de branching condiționat (oricare);
- COND_BRA_REQUIRES_ZERO: se pune pe 1 atunci când se execută o instrucțiune de branching condiționat cu flag-ul ZERO (de ex: BRZ);
- COND_BRA_REQUIRES_NEGATIVE: se pune pe 1 atunci când se execută o instrucțiune de branching condiționat cu flag-ul NEGATIVE (de ex: BRN);
- COND_BRA_REQUIRES_CARRY: se pune pe 1 atunci când se execută o instrucțiune de branching condiționat cu flag-ul CARRY (de ex: BRC);
- COND_BRA_REQUIRES_OVERFLOW: se pune pe 1 atunci când se execută o instrucțiune de branching condiționat cu flag-ul OVERFLOW (de ex: BRO);
- L (Load): se pune pe 1 atunci când instrucțiunea care se execută preia date din Data Memory (inclusiv în urma instrucțiunii de POP se pune L pe 1);
- S (Store): se pune pe 1 atunci când instrucțiunea care se execută încarcă date în Data Memory (inclusiv în urma instrucțiunii de PSH se pune S pe 1);
- TR: se pune pe 1 atunci când se execută o instrucțiune de transfer acumulator - registru (de ex: TRX, TRY);
- STACK_PSH: se pune pe 1 atunci când se execută o instrucțiune de PSH;
- STACK_POP: se pune pe 1 atunci când se execută o instrucțiune de POP;
- MOV: se pune pe 1 atunci când se execută instrucțiunea de MOV;
- flag_select: se pune pe 1 atunci când instrucțiunea care se execută actualizează flagurile (în acest caz, doar instrucțiunile artimetice și logice);
- ACC_select: se pune pe 1 atunci când instrucțiunea care se execută ori actualizează acumulatorul, ori preia valoarea de la acumulator pentru a o încărca în Data Memory (prin multiplexorul de selecție al intrării în Data Memory). Deosebirea dintre cele două situații este aceea că în prima situație S = 0, pe când în a doua situație S = 1. Pentru instrucțiunile aritmetice și logice, doar dacă Immediate = 0, ACC_select = 1;
- X_select: se pune pe 1 atunci când instrucțiunea care se execută ori actualizează registrul X, ori preia valoarea de la registrul X pentru a o încărca în Data Memory (prin multiplexorul de selecție al intrării în Data Memory). Deosebirea dintre cele două situații este aceea că în prima situație S = 0, pe când în a doua situație S = 1. Pentru instrucțiunile aritmetice și logice, doar dacă Immediate e diferit de 0 și RA = 0, X_select = 1;
- Y_select: se pune pe 1 atunci când instrucțiunea care se execută ori actualizează registrul Y, ori preia valoarea de la registrul Y pentru a o încărca în Data Memory (prin multiplexorul de selecție al intrării în Data Memory). Deosebirea dintre cele două situații este aceea că în prima situație S = 0, pe când în a doua situație S = 1. Pentru instrucțiunile aritmetice și logice, doar dacă Immediate e diferit de 0 și RA = 1, Y_select = 1;

- PC_select: se pune pe 1 atunci când instrucțiunea care se execută ori actualizează PC (de ex: instrucțiunile de branch sau POP), ori preia valoarea de la PC pentru a o încărca în Data Memory (prin multiplexorul de selecție al intrării în Data Memory). Deosebirea dintre cele două situații este aceea că în prima situație S = 0, pe când în a doua situație S = 1;
- FACT: se pune pe 1 atunci când se execută instrucțiunea factorial și pe 0 atunci când nu se execută instrucțiunea factorial;

Listing 1: Interfața unității de control

```
module CU(
  input [5:0] opcode,
  input RA, clk, rst,
  input [1:0] RA_stack,
  input [8:0] Immediate,
  output reg ALU, BRA, COND_BRA, COND_BRA_REQUIRES_ZERO,
  COND_BRA_REQUIRES_NEGATIVE, COND_BRA_REQUIRES_CARRY,
  COND_BRA_REQUIRES_OVERFLOW, L, S, TR, STACK_PSH,STACK_POP,
  MOV, flag_select, ACC_select, X_select, Y_select, PC_select
);
```

3.2.2 Unitatea aritmetică-logică

În această unitate intră valorile regiștrilor ACC, X, Y și Immediate. De asemenea, intră și valoarea curentă a timer-ului și registrului asociat modului factorial. Conține o unitate de control locală pentru operații aritmetice și selectează pe baza opcode-ului, Immediate și RA formatul instrucțiunii pe care să o execute. Astfel, nu toți regiștri de la intrare vor fi folosiți la un moment dat. Se va alege, pe baza criterilor, regiștri folosiți în operațiile aritmetice sau logice.

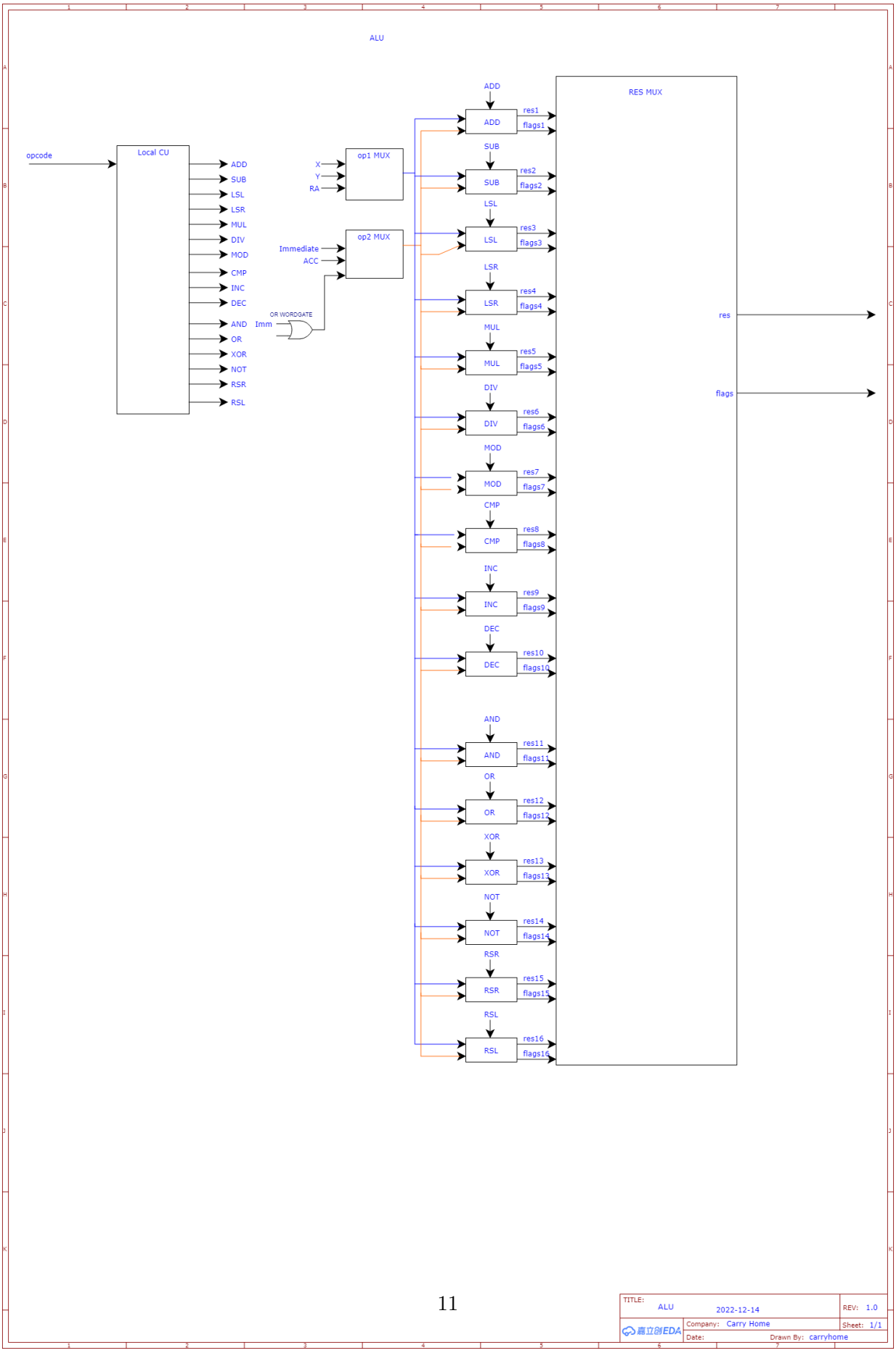

Figura 3: Schema de bază a unității aritmetice și logice

Spre exemplu, dacă avem instrucțiunea ADD #reg, cum Immediate = 0 se va alege utilizarea regiștrilor ACC și X sau Y, în funcție de #reg. Dacă în schimb se folosește instrucțiunea ADD #reg #immediate, se va alege utilizarea regiștrilor X sau Y, în funcție de #reg, respectiv valoarea din Immediate.

Instrucțiunile aritmetice și logice modifică flagurile la fiecare operație. Astfel, sunt prezente următoarele flaguri:

- ZERO: dacă în urma efectuării operației aritmetice sau logice rezultatul este 0, acest flag este setat pe 1;
- NEGATIVE: dacă în urma efectuării operației aritmetice sau logice rezultatul are bitul de semn 1, acest flag este setat pe 1;
- CARRY: dacâ se consideră operanzii numere fără semn și rezultatul este unul eronat (se adună două numere iar rezultatul depășește valoarea maximă alocată pe acel număr de biți), acest flag este setat pe 1. Acest flag nu se setează în cazul operațiilor de șiftare și împărțire, el râmâne pe 0;
- OVERFLOW: dacă se consideră operanzii numere cu semn și rezultatul este unul eronat (se adună două numere de același semn și rezultatul e de semn contrar), acest flag este setat pe 1. Acest flag nu se setează în cazul operațiilor de șiftare și împărțire, el râmâne pe 0;

Listing 2: Interfața unității aritmetice și logice

```
1  module ALU(
2    input [15:0] ACC,X,Y,Immediate,fact_reg,fact_val,
3    input [5:0] opcode,
4    input en,clk,rst,RA,
5    output reg [15:0] res,
6    output reg [3:0] flags);
7
8    reg [15:0] value_before_operation;
9    reg sign_before_operation, same_sign;
10   reg [15:0] rotate_cnt;
```

3.2.3 Regiștri ACC, X, Y

Sunt regiștri pe 16 biți care vor fi folosiți în operațiile internele ale procesorului. Au un semnal de intrare prin care se poate actualiza valoarea din registru, un semnal de tact activ pe frontul descrescător și un semnal de reset activ pe frontul descrescător.

Motivul pentru care registrele sunt active pe frontul descrescător al semnalului de tact este acela că s-a ales ca etapele de fetch și execute să se execute pe nivelul pozitiv al semnalului de tact, pe când actualizarea regiștrilor și a PC-ului să se facă pe frontul descrescător al său. Dacă regiștri s-ar actualiza pe nivelul logic pozitiv al semnalului de tact, atunci există riscul în care operațiile aritmetice și logice să nu funcționeze corect. Ținând cont de faptul că semnalul de selecție pentru respectivul registru este activ, atunci pe toată perioada pozitivă a tactului ALU scrie în registru, preia noua valoare din regiștru, o scrie din nou ș.a.m.d, prin urmare se evită o buclă de nedorit, respectiv un comportament neașteptat în folosirea instrucțiunilor aritmetice și logice.

Listing 3: Interfața regiștrilor ACC, X, Y

```
1  module reg(
```

```
2     input [15:0] in,
3     input clk,rst,w,
4     output reg [15:0] out);
```

3.2.4 Multiplexorul de selecție al intrării în ACC, X, Y

Acest modul este utilizat din motivul ca exista mai multe surse care pot actualiza regiștri ACC, X, Y la un moment dat. Spre exemplu, regiștri pot fi actualizați atât din instrucțiuni aritmetice și logice (spre exemplu ADD X #immediate care face X = X + #immediate) sau prin alte instructiuni (de exemplu MOV X #immediate încarcă în X valoarea #immediate, LDR X #immediate încarcă în X valoarea din Data Memory de la locația #immediate, TRX încarcă în X valoarea din acumulator). Prin urmare, mai multe ieșiri vor trebui să comande aceeași intrare la un anumit registru (intrarea prin care actualizează valoarea stocată în registru). Acest lucru nu este posibil, deci este necesară selecția liniei căreia să i se acorde prioritate să modifice valoarea din registru atunci când ea are nevoie. Deci, ne vom folosi de semnalele de control generate de Control Unit pentru a efectua aceasta decizie, mai exact semnalele ALU, MOV, AL, TR.

Listing 4: Interfața multiplexorului de selecție al intrării în ACC, X, Y

```
1  module REG_WR_MUX(
2     input [15:0] in_ALU,
3     input [15:0] in_MOV,
4     input [15:0] in_DM,
5     input [15:0] in_ACC_TRANSFER,
6     input ALU,MOV,L,TR,
7     output reg [15:0] in
8  );
```

3.2.5 Registrul de indicatori / flags

Este un registru pe 4 biți. Valoarea din acest registru poate fi actualizată numai de ALU, iar acest fapt presupune o intrare care să faciliteze această actualizare. Are o intrare in pe 4 biți, unde:

- Bitul 0 reprezintă bitul de overflow;
- Bitul 1 reprezintă bitul de carry;
- Bitul 2 reprezintă bitul de negative;
- Bitul 3 reprezintă bitul de zero;

Acest registru se actualizează pe nivelul pozitiv al semnalului de tact (neexistând probleme ca la ACC, X, Y sau PC), respectiv se resetează pe frontul descrescător al semnalului de reset (pune ieșirile ZERO, NEGATIVE, CARRY, OVERFLOW pe 0).

Ieșirile ZERO, NEGATIVE, CARRY, OVERFLOW reprezintă starea registrului la un moment dat.

Listing 5: Interfața registrului de indicatori / flags

```
1  module FLAGS(
```

```
2     input [3:0] in,
3     input clk,rst,w,
4     output reg ZERO,NEGATIVE,CARRY,OVERFLOW
5   );
```

3.2.6 Instruction Memory

Instruction Memory este un ROM de 1024 locații x 16 biți în care se încarcă în prealabil instrucțiunile înainte ca procesorul să pornească. Când procesorul pornește, el începe să execute instrucțiuni de la adresa 0, prin urmare instrucțiunile trebuie plasate în Instruction Memory începând cu adresa 0.

Are un semnal de input addr prin care se poate adresa locația din memorie. În Instruction Memory există 1024 locații, deoarece adresa de branching pentru instrucțiunile de branch este pe 10 biți. Acest semnal de input va fi comandat de Program Counter.

Singurul semnal de output reprezintă chiar instrucțiunea de la adresa respectivă.

Mai mult, Instruction Memory se resetează pe frontul descrescător al semnalului de reset, iar resetarea presupune plasarea pe linia de output a instrucțiunii de la adresa 0.

Listing 6: Interfața memoriei de instrucțiuni

```
1  module IM(
2    input [9:0] addr,
3    input rst,
4    output reg [15:0] out
5  );
6
7  reg [15:0] rom [1023:0];
```

3.2.7 Instruction Register

Instruction Register este un registru care preia ieșirea dată de Instruction Memory și o formatează astfel încat următoarele date sunt extrase din instrucțiune: opcode, RA (Register Address), RA_STACK (Register Address pentru cazul instrucțiunilor ce folosesc stiva), BA (Branching Address) și IMM (Immediate).

Sunt calculate în felul următor:

- Opcode este reprezentat de biții 15-10 din instrucțiune;
- RA este bitul 9 din instrucțiune;
- RA_STACK sunt biții 9-8 din instrucțiune;
- BA sunt biții 9-0 din instrucțiune;
- IMM sunt biții 8-0 din instrucțiune;

În mod clar, pentru o anumită instrucțiune nu sunt valide toate ieșirile. Asta pentru că nu toate instrucțiunile dispun atât de RA, RA_STACK, BA cât și de IMM. Prin urmare, prin intermediul semnalelor de control, se cunoaște tipul instrucțiunii respectiv care dintre ieșirile lui Instruction Register sunt valide sau nu.

Listing 7: Interfața registrului de instrucțiuni

```
module IR(
  input [15:0] in,
  input clk,rst,w,
  output reg [15:0] out,
  output reg [5:0] opcode,
  output reg RA,
  output reg [9:0] BA,
  output reg [8:0] IMM,
  output reg [1:0] RA_stack
);
```

3.2.8 Program Counter

Program Counter este un registru pe 16 biți folosit pentru indexarea Instruction Memory (se vor folosi cei mai nesemnificativi 9 biți pentru indexare).

Trebuie să poată fi actualizat (prin operațiile POP, respectiv branching). Dacă se face POP PC iar la adresa din Data Memory conform Stack Pointer se află valoarea X, în Program Counter ar trebui să se depoziteze valoarea X + 2 (deoarece POP PC va fi folosit doar de pseudoinstrucțiunea RET, făcându-se saltul astfel către adresa instrucțiunii imediat următoare celei de JMP), pe când dacă se face BRA X, în Program Counter se va depozitva valoarea X, reluându-se execuția programului de la adresa X.

Dacă nu se scrie prin nicio operație (POP sau branching), atunci Program Counter-ul se incrementează la fiecare front descrescator de tact, atâta timp cât nu se execută instrucțiunea factorial. Acest lucru se datorează faptului ca instrucțiunea factorial e singura instrucțiune care se execută pe mai multe cicluri de clock. De asemenea, dacă nu s-ar face scrierea pe frontul descrescător al semnalului de tact, ar exista riscul în care semnalele de control ar fi invalide la un moment dat. În acest caz, odată activă o instrucțiune, PC ar trece imediat la următoarea, activându-se implicit și semnalele de control aferente următoarei instrucțiuni. Așadar, se evită această problemă.

De asemenea el se resetează doar la frontul descrescător al semnalului de reset.

Listing 8: Interfața PC

```
module PC(
  input [15:0] in,
  input clk,rst,w,BRA,STACK_POP,FACT,
  output reg [15:0] out
);
```

3.2.9 Multiplexorul de selecție al intrării în Program Counter

În Program Counter se pot încărca valori în anumite cazuri (spre exemplu, atunci se folosește instrucțiunea RET sau, în caz uzual, prin instrucțiunile de branch). Prin urmare, două surse nu pot comanda aceeași intrare în același timp, deci este nevoie de un multiplexor sa selecteze linia corespunzătoare la momentul potrivit, care va intra ca input în Program Counter și îl va actualiza. Se vor folosi semnalele de control $STACK\_POP$ și BRA pentru a realiza această selecție.

Listing 9: Interfața multiplexorului de selecție al intrării în Program Counter

```
module PCInputDecider(
    input [15:0] POP_input,
    input [15:0] BRA_input,
    input STACK_POP, BRA,
    output reg [15:0] PC_in
);
```

3.2.10 Data Memory

Data Memory este un RAM de 512 locații x 16 biți în care procesorul poate să scrie (prin operații de STR sau PSH) sau din care procesorul poate să citească (prin operații de LDR sau POP).

Intrările sunt următoarele:

- in: intrare pe 16 biți, prin care se specifică valoarea de încarcat în memorie;
- addr: intrare pe 9 biți, prin care se specifică locația din memorie. Este pe 9 biți deoarece instrucțiunile LDR și STR au Immediate pe 9 biți și deci pot adresa maxim 512 locații;
- w: intrare de scriere - dacă este pe 1 logic înseamnă scriere la adresa specificată de addr a lui in;
- clk: semnalul de clock. Acest circuit funcționează pe nivelul pozitiv al semnalului de clock; rst: semnalul de reset. Acest circuit se resetează pe frontul descrescător al semnalului de reset;

Acest modul are o singură ieșire reprezentând valoarea stocată în memorie (pe 16 biți) de la locația specificată prin addr.

Listing 10: Interfața memoriei de date

```
module DM(
  input [15:0] in,
  input [8:0] addr,
  input clk,rst,w,
  output reg [15:0] out
);
```

3.2.11 Multiplexorul de selecție al intrării în Data Memory

El este folosit pentru a selecta ce dată ajunge in Data Memory. În Data Memory pot ajunge date din mai multe direcții din cauza operației de PSH. Când facem PSH, setul nostru de instrucțiuni poate face PSH la X,Y, ACC sau PC. Prin urmare, linia de OUT a fiecărui registru X, Y, ACC sau PC va trebui să comande intrarea de date de la Data Memory. Acest lucru nu este posibil simultan, prin urmare este nevoie de un multiplexor.

Selectarea intrării se face prin semnalele de control *X_select*, *Y_select*, *ACC_select* sau *PC_select*, care intră în multiplexor.

Listing 11: Interfața multiplexorului de selecție al intrării în Data Memory

```
module REG_DM_IN_MUX(
  input [15:0] in_X,
  input [15:0] in_Y,
  input [15:0] in_ACC,
  input [15:0] in_PC,
  input X_select,Y_select,ACC_select,PC_select,S,
  output reg [15:0] in
);
```

3.2.12 Multiplexorul de selecție al locației din Data Memory

El este folosit pentru a selecta ce locație este accesată în Data Memory. Există mai multe surse care pot seta câmpul de locație, cum ar fi instrucțiunile LDR/STR (caz în care valoare Immediate comandă intrarea) dar și Stack Pointer (pentru operațiile de PSH/POP). Astfel, trebuie folosit un multiplexor și aici. Se vor folosi semnalele de control L, S, $STACK\_PSH$ și $STACK\_POP$ pentru a realiza această selecție.

Listing 12: Interfața multiplexorului de selecție al locației în Data Memory

```
module REG_DM_ADDRESS_MUX(
  input [8:0] in_LS_immediate,
  input [8:0] in_SP_val,
  input L,S,STACK_PSH,STACK_POP,
  output reg [8:0] in
);
```

3.2.13 Stack Pointer

Stack Pointer este folosit pentru a adresa locația vârfului stivei în Data Memory. Stack Pointer-ul este pe 16 biți, însă în realitate noi vom folosi doar 9 biți ai lui. Aceasta se datorează faptului că Data Memory are 512 locații (deoarece Immediate la LDR/STR e pe 9 biți), iar 2^9 este 512, prin urmare doar 9 biți ai lui Stack Pointer vor fi folosiți.

Stiva crește invers în memorie. Asta înseamna că inițial, la un reset, valoarea din Stack Pointer este 01FFH, sau mai exact, ultimii 9 biți sunt 1, iar asta e chiar ultima poziție din Data Memory, adică locația 511. La un PSH se decrementează Stack Pointer-ul, iar la POP se incrementează.

De asemenea, și pentru acest modul reset-ul se face pe frontul descrescător al semnalului de tact.

Listing 13: Interfața SP

```
module SP(
  input clk,rst,inc,dec,
  output reg [15:0] out
);
```

3.2.14 Zero Extend 10x16 și Sign Extend 9x16

Zero Extend 10x16 este un modul cu o intrare pe 10 biți și o ieșire pe 16 biți. Ieșirea este egală cu valoarea intrării în care biții rămași sunt completați cu zero. Acest modul este folosit pentru extinderea adresei de branch care este pe 10 biți înainte ca aceasta să fie încarcată în Program Counter. Prin urmare, ieșirea acestui modul este intrare în multiplexorul de selecție al intrării în Program Counter.

Sign Extend 9x16 este un modul cu o intrare pe 9 biți și o ieșire pe 16 biți. Ieșirea este egală cu valoarea intrării în care biții rămași sunt completați cu bitul de semn al intrării. Acest modul este folosit pentru extinderea Immediate care este pe 9 biți înainte ca acesta să fie folosit în operații aritmetice. De asemenea, este folosit și pentru încarcarea într-un registru a unei valori prin MOV, deci această ieșire intră și în multiplexorul de selecție al întrării în registrele ACC, X, Y.

Listing 14: Interfața Zero Extend

```
1 module ZE10x16(
2   input [9:0] in,
3   output reg [15:0] out
4 );
```

Listing 15: Interfața Sign Extend

```
1 module SE9x16(
2   input [8:0] in,
3   output reg [15:0] out
4 );
```

3.2.15 Modulul factorial

Acesta este alcătuit din două submodule. Primul submodul este un signal generator cu rol de timer, iar al doilea submodul este un registru în care se memorează rezultatele intermediare ale factorialului la fiecare ciclu de clock.

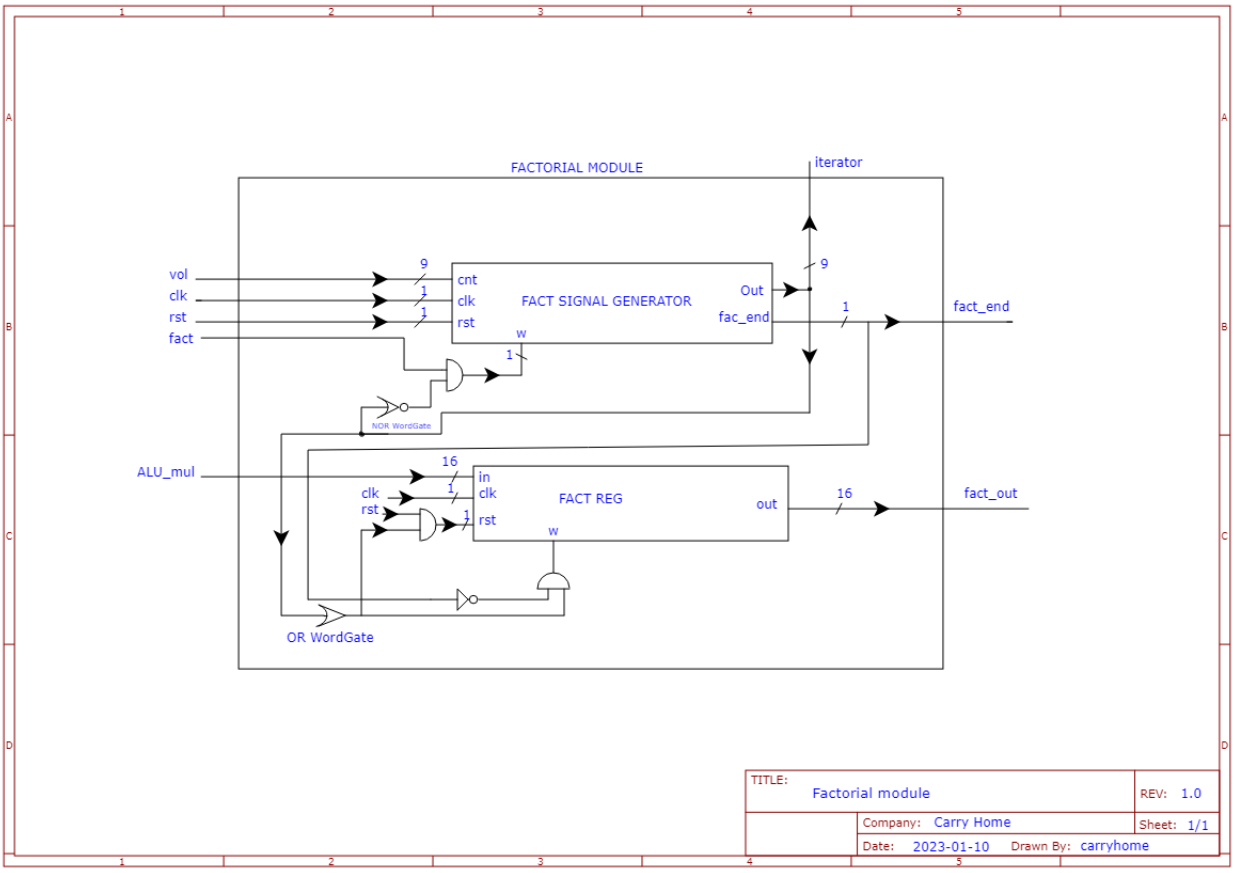

Figura 4: Schema de bază a modulului factorial

Listing 16: Interfața modulului factorial

```
module FACTORIAL(
  input [8:0] val,
  input clk, rst, FACT,
  input [15:0] ALU_mul,
  output reg FACT_END
  );
```

Inițial, în timer este încărcat valoarea pentru care se calculează factorialul. Spre exemplu, dacă se vrea să se calculeze 5!, în timer se încarcă valoarea 5. La fiecare ciclu de clock, timer-ul decrementează acea valoare. Atunci când timer-ul ajunge la 1, generează un semnal numit $FACT\_END$, prin care anunță Control Unit că factorialul s-a terminat de calculat.

Listing 17: Interfața timer-ului

```
module FACT_CNT(
  input [8:0] cnt,
  input clk, rst, w,
  output reg [8:0] out,
  output reg FACT_END);
```

Motivul existenței semnalului $FACT\_END$ este dat de faptul că, până acum, toate instrucțiunile noastre au fost single cycle, iar acum avem o instrucțiune factorial care are nevoie de un număr variabil de cicluri de clock ca să se execute. Din Control Unit va mai ieși un semnal FACT, care când este activ pe 1 desemnează faptul că factorialul rulează, iar dacă e pe 0 el nu rulează.

Atunci când semnalul $FACT\_END$ este generat, automat FACT se pune pe 0, prin urmare Control Unit știe să dezactiveze semnalul FACT atunci când factorialul s-a terminat de calculat. De asemenea, semnalul de control FACT, pe lângă că ne ajută să activăm modulul factorial, este folosit și ca input în PC. Motivul este acela că PC nu trebuie incrementat cât timp factorialul se execută. Până acum, PC se incrementa la fiecare sfârșit de ciclu de clock. Cum factorialul se calculează pe mai multe cicluri, acest lucru nu mai este valid, trebuie ca PC să fie halted până când factorialul s-a terminat de calculat și rezultatul său e depus în registrul corespunzător.

Al doilea submodul, registrul $FACT\_REG$, este actualizat la fiecare sfârșit de ciclu de clock cu valoarea factorialului pentru iterația curentă, dată de ALU. În ALU vor intra două noi intrări: valoarea curentă din timer care e zero extended și valoarea curentă din registrul $FACT\_REG$. Se realizează o operație de multiplicare, a cărui rezultat este depus din nou în $FACT\_REG$ pentru a putea fi folosit la iterații viitoare. De notat este faptul că la un reset acest registru este pus pe 1.

Listing 18: Interfața registrului FACT_REG

```
1 module FACT_reg(
2   input [15:0] in,
3   input clk,rst,w,
4   output reg [15:0] out);
```

Nu în ultimul rând, registrul $FACT\_REG$ trebuie să se reseteze de fiecare dată când se termină de calculat factorialul, pentru a putea fi folosit și pentru alte instrucțiuni de factorial, în cazul în care programul care este executat de procesor conține mai multe instrucțiuni prin care se calculează factorialul. Scrierea în regiștri X sau Y, după caz, se va face prin intermediul ieșirii de la ALU atunci când $FACT\_END$ este pus pe 1.

4 Sarcini

Designeri hardware:

- Bianca: ALU, schema finală, modul factorial;
- Lore: Control Unit, schema finală, modul factorial;

Software:

- Sebi: Asamblor, ALU, regiștri X, Y, ACC, IR, modulul final, modul factorial;
- Adrian: Control Unit, Sign Extend, PC, IM, multiplexoare pentru PC și regiștri, modul final, modul factorial;
- Cristi: Control Unit, Zero Extend, SP, DM, multiplexoare pentru DM, modul final, modul factorial;

Testing:

- Iulia: testare regiștri X, Y, ACC, PC, IR, Control Unit, ALU, Sign Extend, Zero Extend, modul final, modul factorial. S-a realizat o testare de $> 70\%$;

Project Manager s-a implicat în mai multe zone inclusiv design hardware (ALU, schemă finala, modul factorial și altele), Software (modul final, Control Unit, ALU și altele), testing. S-a ocupat de documentație, prezentări, timeline și gestionarea situațiilor.

Am avut în total 9 meeting-uri în care am discutat și rezolvat eventualele probleme.