Proiectarea unei unitati centrale de executie

Nume : Nistor Cristian-Emil

Grupa : 30231 / Semigrupa : 2

Cuprins

1. Introducere

Proiectarea unei unități centrale reprezintă un proces complex și esențial în dezvoltarea sistemelor informatice moderne. O unitate centrală bine concepută joacă un rol crucial în funcționarea eficientă a unui sistem de calcul. Acest proiect se axează pe proiectarea unei astfel de unități centrale, care să integreze aproximativ 20 de instrucțiuni esențiale, acoperind operațiuni aritmetice, logice, de transfer și de salt. În plus, unitatea centrală proiectată va include elemente cheie, precum un acumulator și 8 registre generale, pentru a facilita manipularea și stocarea datelor în cadrul sistemului.

Una dintre caracteristicile distinctive ale acestei unități centrale este implementarea sa utilizând bistabile și porti logice, cu ajutorul tehnologiei Xilinx și limbajului de descriere a hardware-ului VHDL (VHSIC Hardware Description Language).

Unitatea centrală proiectată va oferi suport pentru diferite moduri de adresare, inclusiv adresare imediată, adresare directă și adresare prin intermediul unui registru dedicat.

2. Planificare

Laborator1: Alegere proiect
Laborator2: Documentare
Laborator 3: Program assembly + cod masina
Laborator 4: Unitatea de control + Registre
Laborator 5: ALU + Unitatea de memorie
Laborator 6: Legarea tuturor componentelor + Testare
Laborator 7: Predare

3. Studiu bibliografic

3.1. *Registre Arhitectura x86* :

3.1.1. Registre de date:

EAX	Accumulator
EBX	Base register
ECX	Counter register
EDX	Data register - can be used for I/O port access and arithmetic functions

3.1.2. Registre de adresa:

ESI	Source index register
EDI	Destination index register
EBP	Base pointer register
ESP	Stack pointer

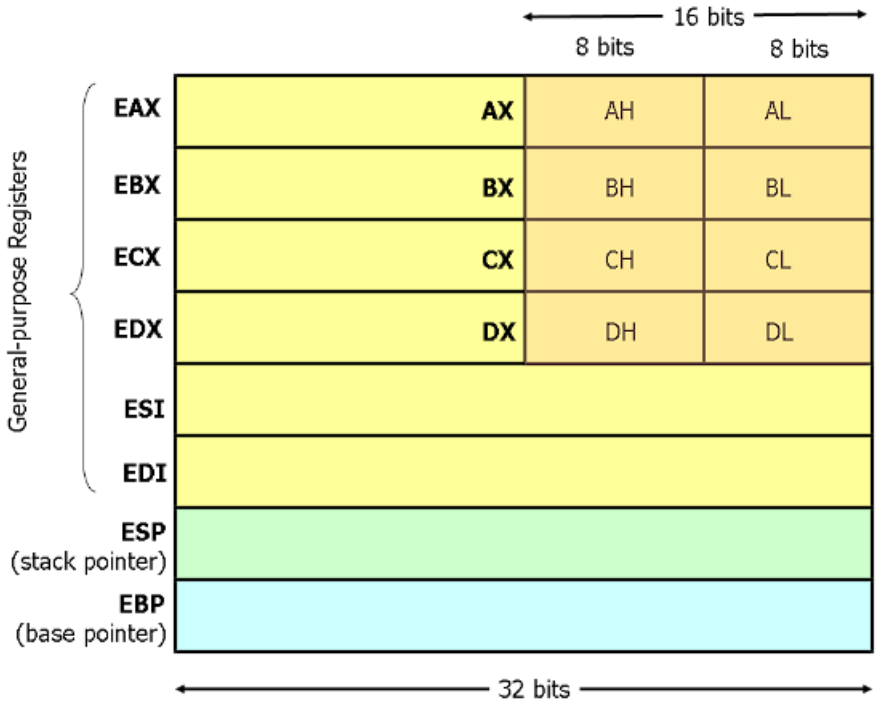


Figure 1. x86 Registers

3.1.3. Registre de segment: **CS, DS, ES, FS, GS, SS**

Acestea sunt registrele de segment, care conțin adresele de bază ale segmentelor

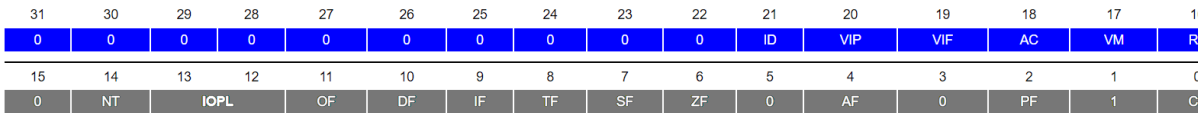
corespunzătoare din memoria principala

3.1.4. Registre de control:

Alti doi registri sunt importanti pentru starea curentă a procesorului:

EIP	instruction pointer
EFLAGS	flags

EFLAGS este un registru de 32 de biți utilizat ca o colecție de biți reprezentând valori booleene pentru a stoca rezultatele operațiilor și starea procesorului.



Printre cele mai folosite flaguri se numara:

CF: Carry flag. Seteaza dacă ultima operație aritmetică a efectuat (pt adunare) sau a împrumutat (pt scădere) puțin peste dimensiunea registrului. Acest lucru este apoi verificat atunci când operația este urmată de o adunare cu transport sau scădere cu împrumut pentru a trata valori prea mari pentru a le tine intr-un singur registru.

PF: Parity flag. Seteaza dacă numărul de biți cu valoarea 1 din octetul cel mai puțin semnificativ este un multiplu de 2.

ZF: Zero flag. Seteaza dacă rezultatul unei operații este zero.

SF: Sign flag. Seteaza dacă rezultatul unei operații este negativ

DF: Direction flag. Dacă se setează, operațiunile cu șir își vor decrementa pointerul în loc să-l incrementeze, citind memoria în sens invers.

OF: Overflow flag. Seteaza dacă operațiile aritmetice cu semn au ca rezultat o valoare prea mare pentru ca registrul să o poata conține.

3.2. Tipuri de adresare :

1. **Adresare imediata** :

Adresarea imediată este un mod de adresare în care operandul instrucțiunii este specificat direct în cadrul instrucțiunii. Instructiunea conține însăși valoarea operandului și această valoare este utilizată direct în operația specificată de instrucțiune. Acest tip de adresare este util atunci când se lucrează cu valori constante sau cu valori mici care pot fi încorporate direct în codul instrucțiunii.

Exemplu :

```
MOV AX, 5       ; Încarcă valoarea 5 în registru AX
```

2. **Adresare directa :**

Adresarea directă este un mod de adresare în care operandul instrucțiunii este situat la o anumită adresă de memorie. Adresa operandului este specificată direct în cadrul instrucțiunii și valoarea operandului este luată din acea adresă specificată. Adresarea directă este utilă atunci când se lucrează cu variabile sau cu date stocate la anumite adrese de memorie.

Exemplu :

```
MOV AX, [1234]          ; Încarcă valoarea stocată la adresa de memorie 1234 în
                          registru AX
```

3. **Adresare indirecta:**

Adresarea indirecta implică specificarea unui registru în cadrul instrucțiunii care conține adresa de memorie a operandului. În loc să specifici direct adresa de memorie, specifici un registru care conține această adresă. Operandul este apoi luat de la adresa de memorie conținută în registru.

Exemplu :

```
MOV AX, [BX]        ; Se încarcă valoarea de la adresa de memorie stocată în registru
                      BX în registru AX
```

4. **Adresarea tip registru**:

Adresarea tip registru este un mod de adresare în care ambii operanzi sunt registri, fiind rapidă și eficientă, deoarece elimină necesitatea accesului la memorie pentru a obține sau a stoca date.

Exemplu:

```
SUB CX, DX          ; Scade conținutul registrelor CX și DX și stochează rezultatul în
                    registru CX
```

4. Analiza

In primul rand vom analiza instructiunile folosite in programul implementat pentru a sesiza ce resurse hardware vom avea nevoie pentru implementarea unitatii centrale.

Pentru instructiunile add, inc o sa avem nevoie de un sumator ce va fi inclus in ALU. De asemenea instructiunea add va influenta registrul EFLAGS, mai exact OF si CF (pentru Carry Overflow), respective ZF, SF, AF sau PF, iar instructiunea inc va influenta OF, SF, ZF, AF si PF.

Pentru instructiunea xor o sa avem nevoie de o componenta ce va realize xor intre 2 operanzi, component introdusa in ALU. Flaguri afectate: OF și CF sunt șterse; SF, ZF și PF sunt setate în funcție de rezultat. Starea steagului AF este nedefinită.

Instructiunea cmp presupune existenta unui scazator inclus in ALU, iar flagurile CF, OF, SF, ZF, AF și PF sunt setate în funcție de rezultat.

Instructiunea test presupune existenta unui AND logic inclus in ALU, iar flagurile afectate vor fi SF, ZF si PF (OF si CF primesc 0, iar AF este nedefinit).

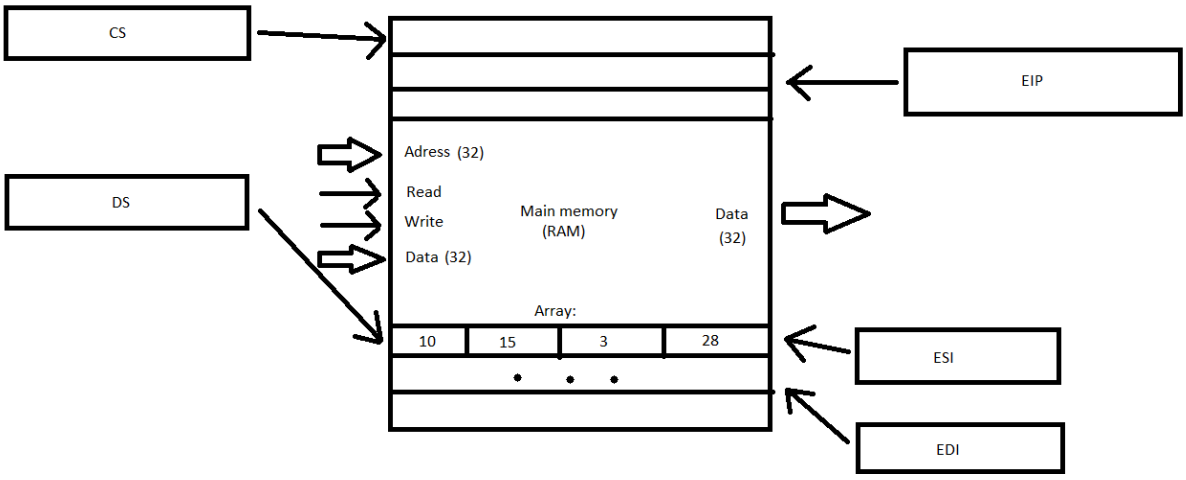
Instructiunea jump presupune folosirea unui sumator inclus in ALU pentru calcularea adresei lui EIP.

Instructiunile jge, jz, jnz se folosesc de flaguri pentru a determina daca salturile si realizeaza sau nu(daca schimbam valoarea lui EIP sau nu). Jge verifica daca SF = OF, jz daca ZF=1 iar jnz daca ZF=0.

Sign / Zero extension : pentru instructiunile care au ca si operand un imediat, vom folosi o unitate de extindere a imediatului astfel incat sa poata fi folosit in ALU.

Memoria externa (Main memory) : aceasta memorie va fi una de tipul RAM si va fi impartita in 2 componente : memoria codului (ce contine instructiunile programului) si memoria de date (ce contine datele de intrare, respective iesire necesare pentru rularea programului).

De asemenea aceasta memorie se va folosi de registri CS, DS, EIP. CS va indica adresa din memorie de unde incepe memoria codului, DS va indica adresa memoriei de date iar EIP va indica instructiunea ce urmeaza a fi executata.



Tot odata din punct de vedere al formatului instructiunilor, se pot observa 3 tipuri de instructiuni:

Tip1 : Opcode + ModR/M

Tip2 : Opcode + ModR/M + Immediate

Tip3 : Opcode + Immediate (salt)

Formatul Instructiunilor :

Opcode : Pentru a determina lungimea opcode–ului o sa adaugam ca si prin byte al opcode-ului byte-ul 0FH care o sa ne semnaleze faptul ca avem un opcode mai mare. De asemenea acesta poate folosi 3 biti din byte-ul ModR/M.

ModR/M : Reg / Opcode contine un registru folosit ca si operand sau o extensie de opcode.

R/M : Alături de 2 biți R/M, cei 5 biți specifică un operand, fie un registru, fie o memorie.

Immediate : Octeții imediatului sunt datele care sunt transmise direct cu instrucțiunea în loc să preluăm date din registre sau din memorie.

Deplasarea/secțiunea imediată poate fi de 1, 2, 3 sau 4 octeți.

5. Proiectare

5.1. Instructiuni folosite :

add : Instrucțiunea add adună cei doi operanzi ai săi, stocând rezultatul în primul său operand. In timp ce ambii operanzi pot fi registri, cel mult un operand poate fi o locație de memorie.

Inst:	opcode:
ADD r/m32, imm32	81
ADD r/m8, r8	00

xor: Acesta instrucțiune efectuează operația logica xor pe biți asupra operanzilor lor, plasând rezultatul în locația primului operand.

Inst:	opcode:
XOR r32, r32	33

mov : Instrucția "mov" copiază elementul de date referit de al doilea operand (adică conținutul unui registru, conținutul din memorie sau o valoare constantă) în locația referită de primul operand (adică un registru sau o locație din memorie). Deși sunt posibile mutări de la registru la registru, mutările directe de la memorie la memorie nu sunt permise. În cazurile în care sunt necesare transferuri de memorie, conținutul memoriei sursă trebuie mai întâi încărcat într-un registru, după care poate fi stocat la adresa destinație din memorie.

Inst:	opcode:
MOV r32, r/m32	8B
MOV r32, imm32	B8

jmp: Transferă controlul programului la instrucțiunea aflată la locația de memorie indicată de operand.

Inst:	opcode:
JMP imm8	EB

cmp: Compara valorile celor doi operanzi specificați, setând corespunzător valorile in registrul EFLAGS. Această instrucțiune este echivalentă cu instrucțiunea de scadere, cu excepția faptului că rezultatul scăderii este eliminat în loc să înlocuiască primul operand.

Inst:	opcode:

CMP r32, r/m32	3B

jge, jnz, jn: Aceste instrucțiuni sunt sărituri condiționate care se bazează pe starea unui set de coduri de condiție care sunt stocate într-un registru special numit EFLAGS. Conținutul acestui registru include informații despre ultima operație aritmetică efectuată. De exemplu, un bit din acest cuvânt indică dacă ultimul rezultat a fost zero. Un altul indică dacă ultimul rezultat a fost negativ. Pe baza acestor coduri de condiție, pot fi efectuate o serie de sărituri condiționate. De exemplu, instrucțiunea jz efectuează un salt la eticheta de operand specificată dacă rezultatul ultimei operații aritmetice a fost zero. În caz contrar, controlul trece la următoarea instrucțiune în secvență.

Inst:	opcode:	conditon:
JGE imm8	7D	if GE = 1
JNZ imm8	75	if Z = 0
JZ imm8	74	if Z = 1

inc: Crește conținutul operandului său cu 1 (in cadrul programului meu cresterea se face cu 4).

Inst:	opcode:
CMP r32, r/m32	40

test: Instrucțiunea TEST efectuează un AND pe biți pe doi operanzi. Flag-urile SF, ZF, PF sunt modificate în timp ce rezultatul AND-ului este ignorat. Flag-urile OF și CF sunt setate la 0, în timp ce flag-ul AF este nedefinit.

Inst:	opcode:
CMP AL, imm8	A8

5.2. Program assembly: Suma numerelor pare de pe pozitiile impare dintr-un sir

```
XOR ECX, ECX        ; ECX va fi utilizat pentru a itera prin șir
ADD EDX, 0          ; EDX va stoca suma numerelor pare de pe pozițiile impare
```

```
        MOV ESI, array                  ; ESI va arăta către începutul șirului
calculate_sum:
        MOV EBX, n
        CMP ECX, [EBX]          ; Verificăm dacă am parcurs întregul șir
        JGE done                ; Dacă da, ne oprim
        ADD ESI, ECX
        MOV AL, [ESI]           ;Încarcăm numărul curent din șir în AL
        TEST AL, 1              ; Verificăm dacă numărul este impar
        JNZ next_number         ; Dacă da, trecem la următorul număr în șir
        TEST AL, 1              ; Dacă numărul este par, verificăm cel mai puțin semnificativă bit
        JZ add_to_sum           ; Dacă este par, adăugăm numărul la suma
next_number:
        INC ECX                 ; Trecem la următorul element din șir
        JMP calculate_sum       ; Continuăm să calculăm suma
add_to_sum:
        ADD DL, AL              ; Adăugăm numărul par la suma (DL stochează suma)
        INC ECX                 ; Trecem la următorul element din șir
        JMP calculate_sum       ; Continuăm să calculăm suma
done:
                ; La acest punct, suma numerelor pare de pe pozițiile impare este stocată în DL
```

5.3. Cod masina:

```
0:      0b00110011 0b11 001 001     : XOR ECX, ECX
1:      0b10000001  0b11 010 000 0x00000000       : ADD EDX, 0
```

```
2:      0b10111000 0b11 110 000 0x00000008 :      MOV ESI, array
        calculate_sum:
3:      0b10111000 0b11 011 000 0x00000000        : MOV EBX, n
4:      0b00111011 0b00 001 011     : CMP ECX, [EBX]
5:      0b01111101 0b00010010       : JGE done
6:      0b00000001 0b11 110 001     : ADD ESI, ECX
7:      0b10001010 0b00 000 110     : MOV AL, [ESI]
8:      0b10111000 0b11 110 000 0x00000008        : MOV ESI, array
9:      0b10101000 0b11 000 000 0b000000001       : TEST AL, 1
10:     0b01110101 0b00001101       : JNZ next_number
11:     0b10101000 0b11 000 000 0b000000001       : TEST AL, 1
12:     0b01110100 0b00001111       : JZ add_to_sum
        next_number:
13:     0b01000000 0b11 001 000     : INC ECX
14:     0b11101011 0b00000011       : JMP calculate_sum
        add_to_sum:
15:     0b00000000 0b11 010 000     : ADD DL, AL
16:     0b01000000 0b11 001 000     : INC ECX
17:     0b11101011 0b00000011       : JMP calculate_sum
        done:
18:
```

5.4. Proiectare Componente:

5.4.1. Instruction Fetch:

Primul pas al ciclului este preluarea unei instrucțiuni din memorie. Registrul Instruction Pointer (IP) al procesorului deține adresa de memorie a următoarei instrucțiuni care urmează să fie executată. Procesorul citește conținutul acestei locații de memorie și stochează instrucțiunea într-o zonă de stocare temporară numită registru de instrucțiuni (IR).

In functie de opcode-ul instructiunii curente, componenta Instruction Fetch va scoate pe iesire adresa urmatoarei instructiuni.

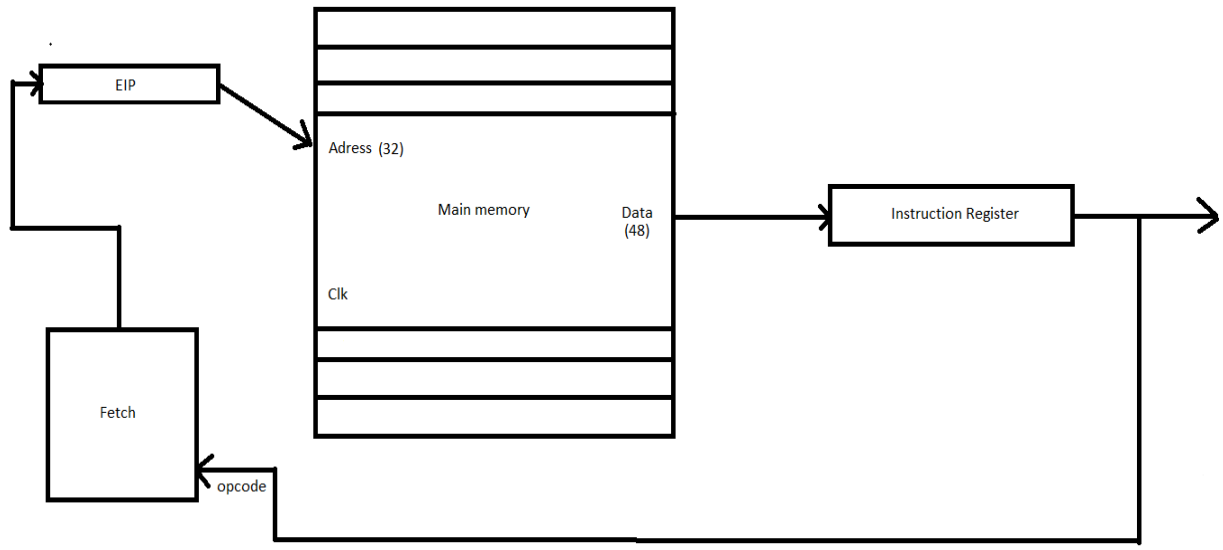


Figura 5.4.1

5.4.2. Instruction Decode:

Odată ce instrucțiunea a trecut prin etapa de fetch si a fost preluata, procesorul trebuie să o decodifice pentru a specifica operatia ce urmeaza a fi facuta. Procesul de decodificare implică interpretarea diferitelor câmpuri ale instrucțiunii, cum ar fi Opcode, ModR/M, Immediate. Opcode-ul specifică tipul de operație ce trebuie efectuată precum si lungimea operanzilor, în timp ce ModR/M si Immediate scot pe iesire operanzii.

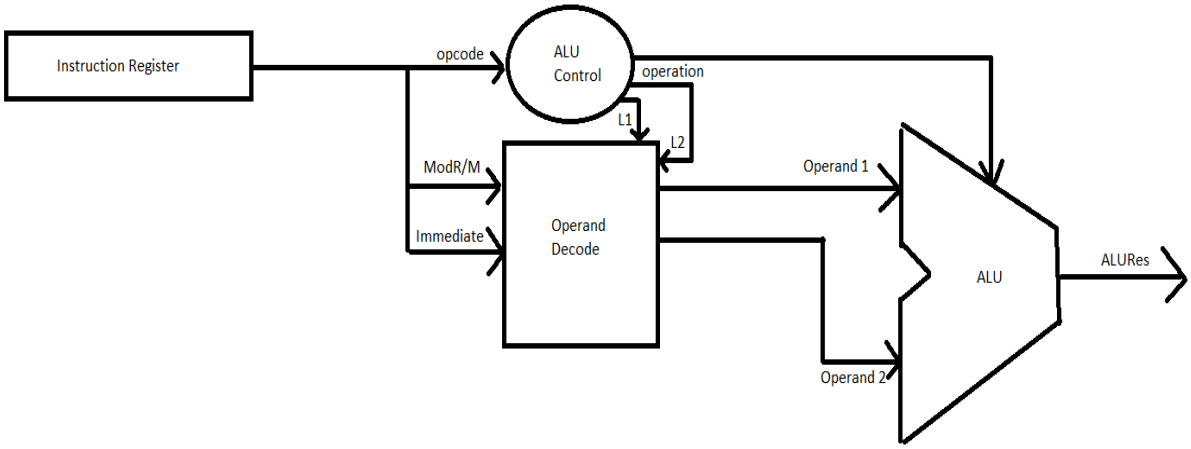


Figura 5.4.2.1

Operand Decode Logic:

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX] [ECX] [EDX] [EBX] [--][--][1] disp32[2] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[EAX]+disp8[3] [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

Figura 5.4.2.2

5.4.3.Unitatea Aritmetica si Logica (ALU):

Din moment ce am extras operatia si operanzii din instructiune, procesorul urmeaza sa foloseasca ALU pentru a obtine rezultatul dorit. Unitatea Aritmetica – Logica efectueaza urmatoarele operatii: adunare pe numere intregi, scadere pe numere intregi, xor logic, and logic si noOp (no operation) in cazul instructiunilor de salt sau de MOV.

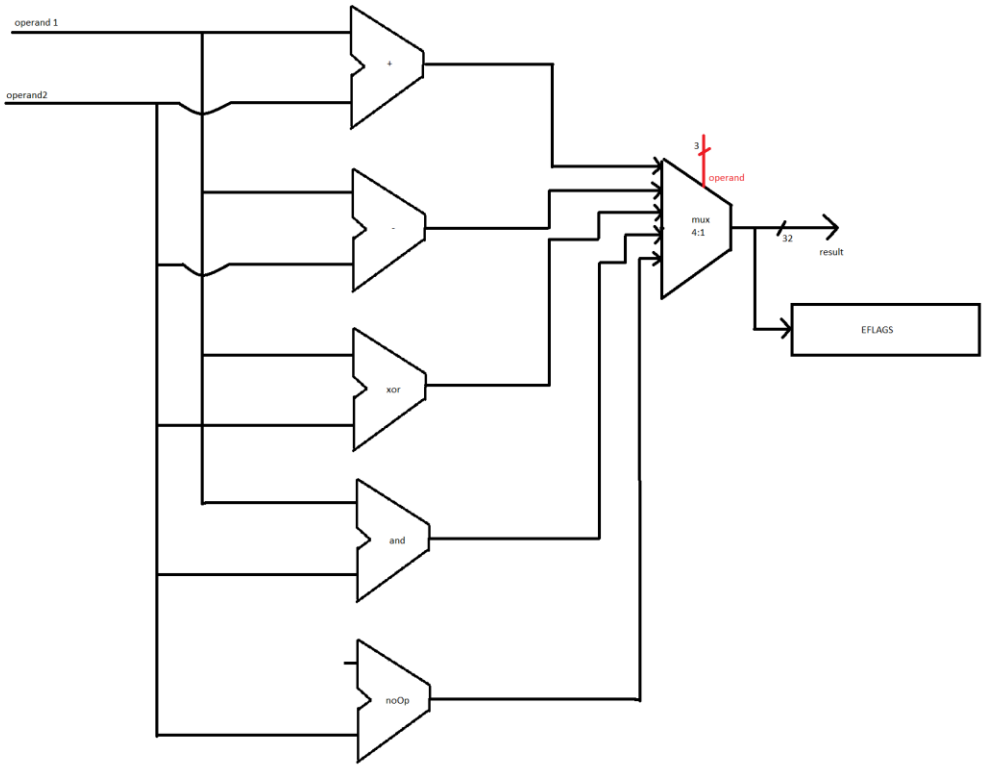
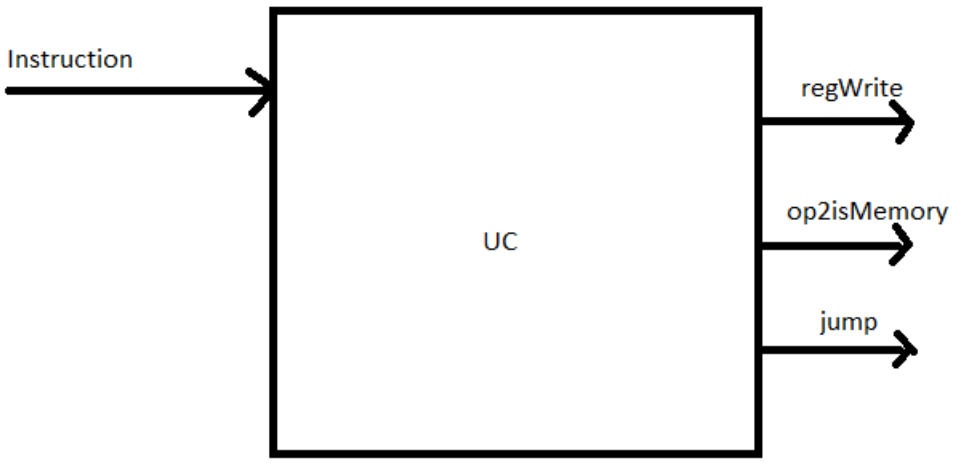


Figura 5.4.3

5.4.4.Unitatea de Control(UC):

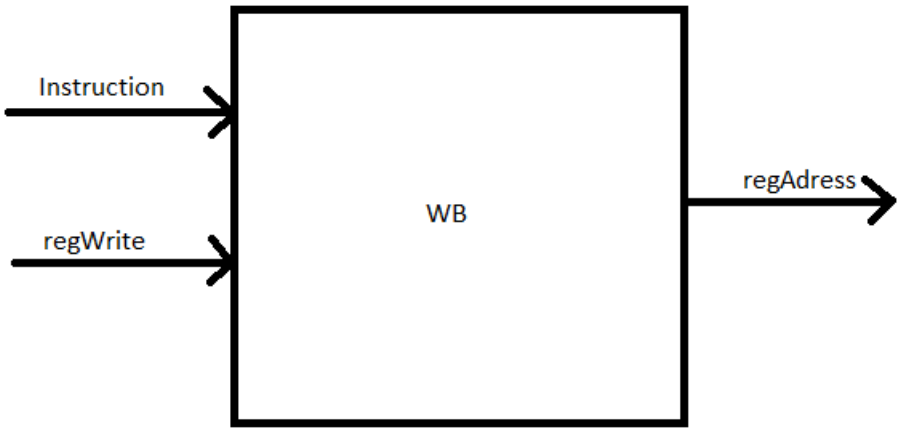
Unitatea de control primeste ca intrare instructiunea curenta si in functie de opcode-ul acesteia, scoate pe iesire flagurile necesare pentru realizarea instructiunii.

Semnalul regWrite permite scrierea in registri de date, jump semnalizeaza o instructiune de jump, iar op2isMemory imi spune ca trebuie sa accesez memoria pentru a scoate al doilea operand, inainte de a-l transmite la ALU.d



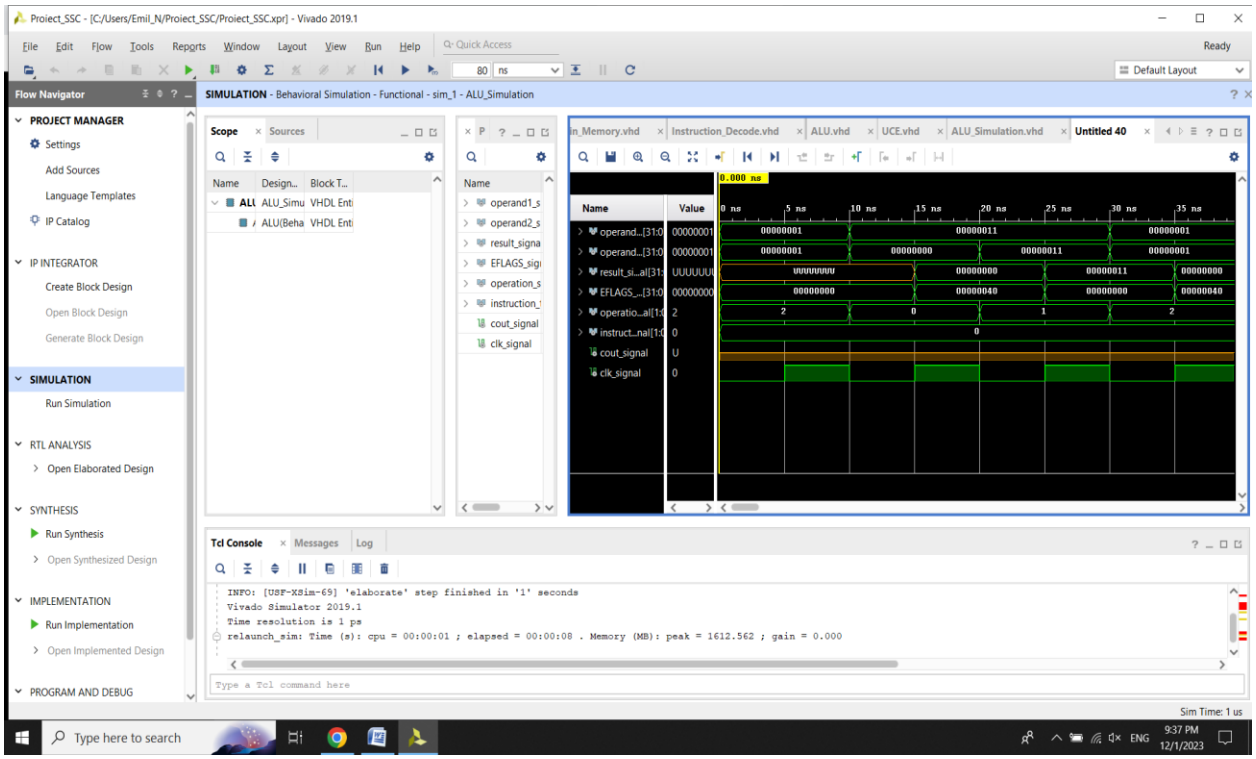
5.4.5.Write Back:

Unitatea de Write Back primeste instructiunea curenta si semnalul de regWrite, iar daca semnalul regWrite este 1, in functie de byte-ul ModR/M voi scoate pe iesire adresa registrului in care urmeaza sa scriu rezultatul oferit de ALU.

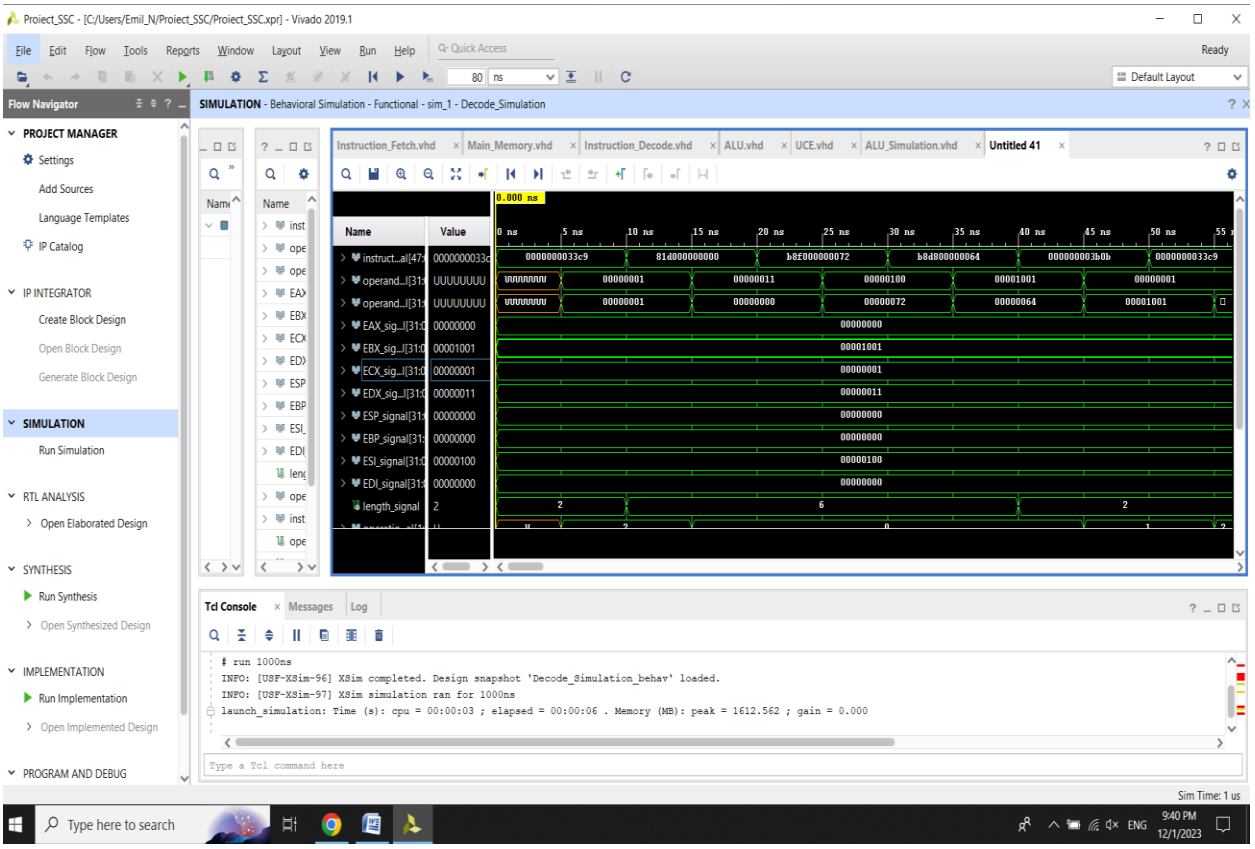


6. Simulari

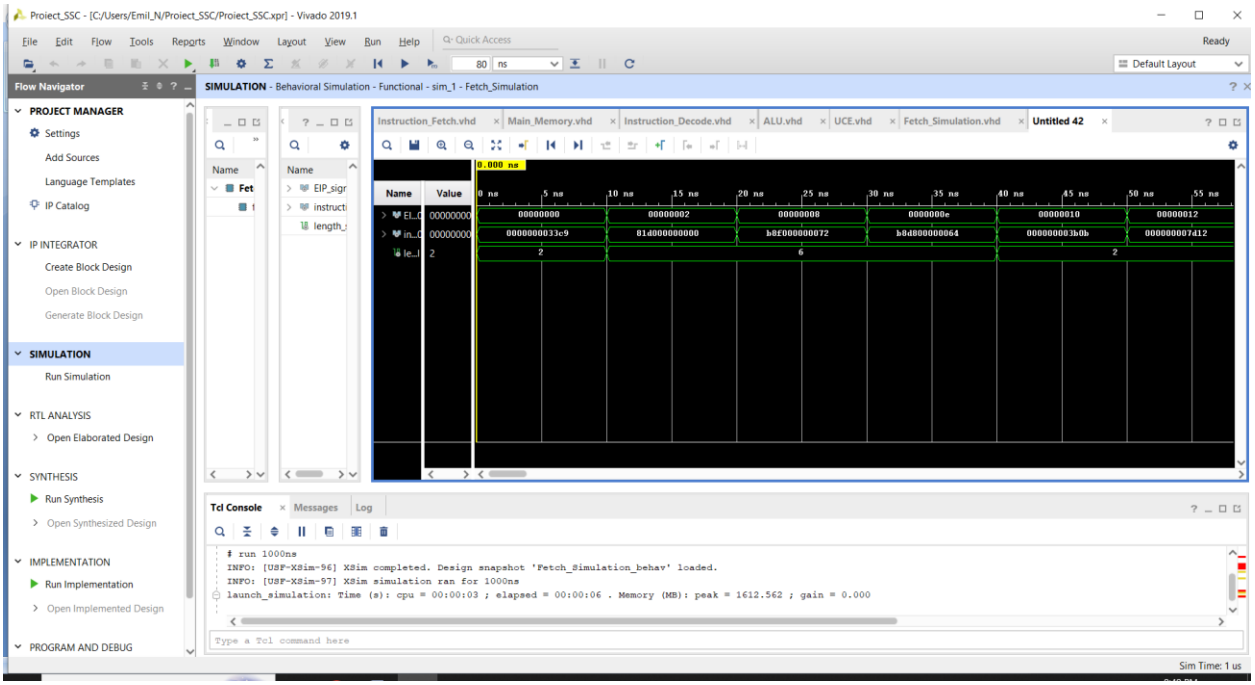
6.1. Simulare ALU:



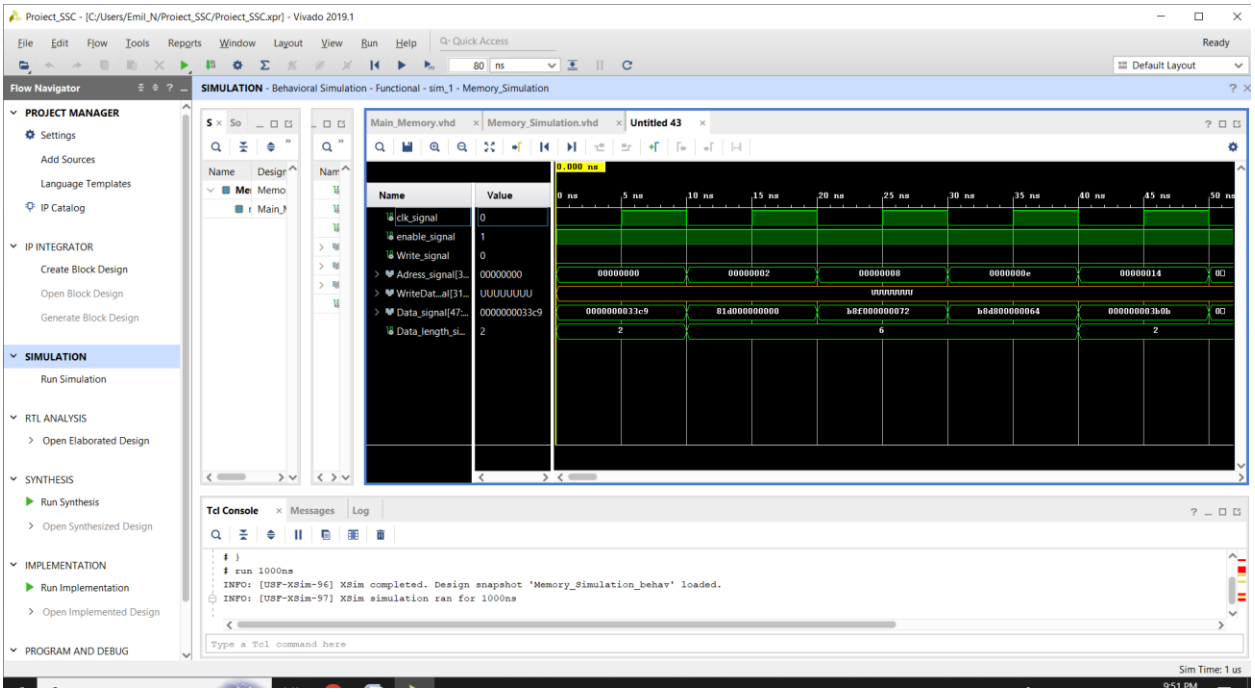
6.2. Simulare Instruction Decode



6.3. Simulare Instruction Fetch



6.4. Simulare Memorie



7. Bibliografie

https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-architecture

https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

https://medium.com/@g.c.dassanayake/an-introduction-to-intel-32-bit-instruction-decoding-9b3b0c15bebb

http://xxeo.com/single-byte-or-small-x86-opcodes