DEPARTAMENTUL CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI

UNIVERSITATEA POLITEHNICA TIMIȘOARA

Universitatea Politehnica Timișoara

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Membrii echipei: Novacovici Ioana, Novacovici Sonia, Panduru Marius, Paulescu Andrei, Pegulescu Bogdan, Privantu Gabriel-Claudiu, Alexandru-Cristian Avram

Numele echipei: NEED 4 POWER

GitHub link: https://github.com/Sonianv/PCP

Coordonator: as. dr. ing. Bozdog Alexandru

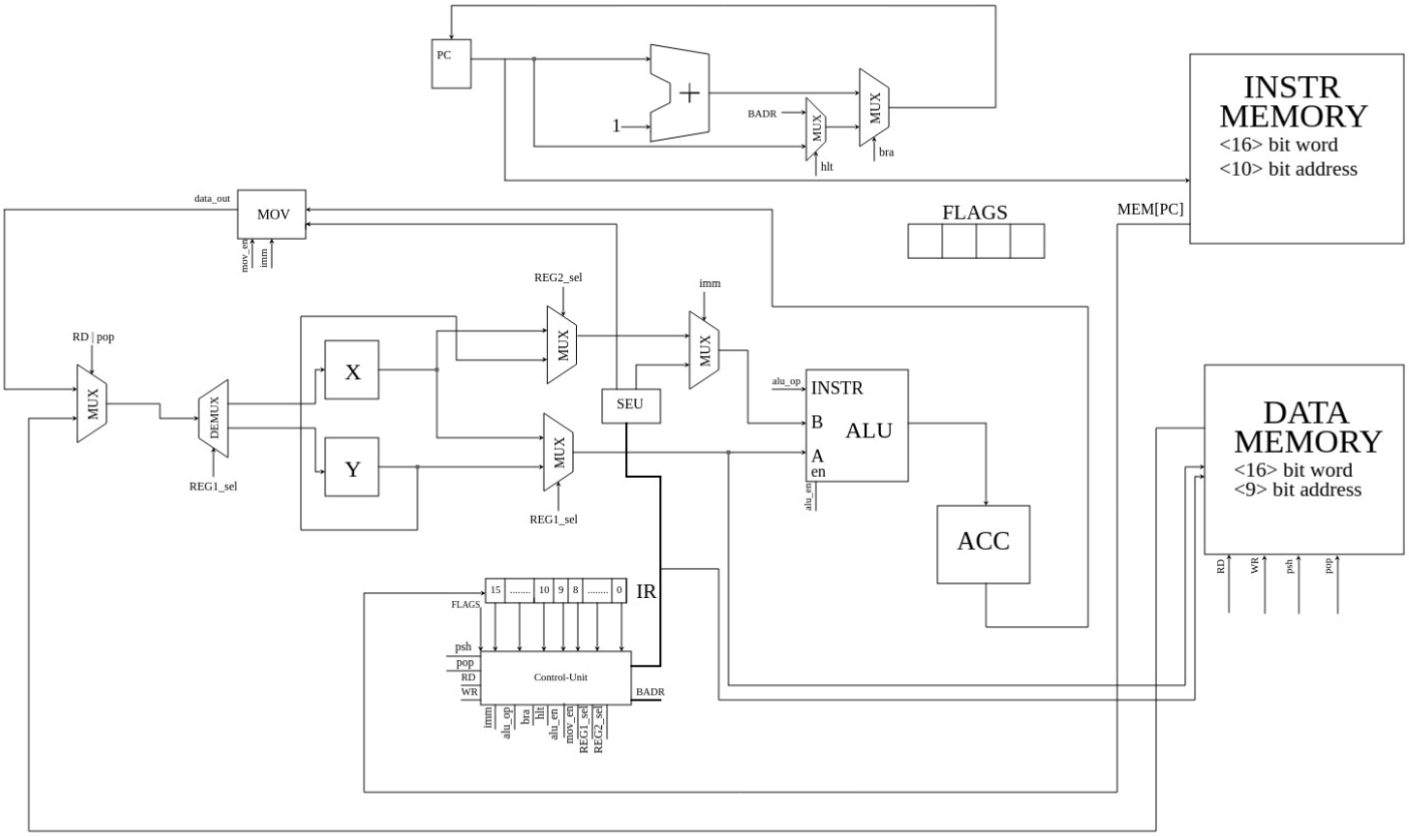
FUNDAMENTE DE INGINERIA CALCULATOARELOR

(F.I.C.)

- Calculatorul de buzunar -

An universitar: 2022-2023

Schema hardware:



Descrierea componentelor:

1. Control Unit:

- produce semnale de control, pe baza opcode-ului instrucțiunii, care commandă diferite componente ale CPU
- RD/WR pentru citire/scriere din memorie
- alu_op pentru diferențierea între operații ALU și celelalte tipuri, plus alu_en, semnal de enable pentru operațiile ALU
- bra și hlt ca selectori pentru multiplexoarele care încarcă valoarea în PC în funcție de tipul instrucțiunii
- imm pentru semnalizarea operațiilor cu valori immediate

2. **Program Counter (PC):**

   - Registru pe 10 biți
   - Indica spre instructiunea curenta
   - Iesirea selectata de multiplexor va indica adresa urmatoarei instructiuni pasate PC-ului

3. **Instruction Memory:**

   - Are un singur port de citire (memorie rom)
   - Adresele sunt pe 10 biți, iar datele sunt instrucțiuni pe 16 biți

4. **Acumulator (ACC):**

   - Registru pe 16 biți, în care se stochează rezultatul din ALU

5. **Sign Extend Unit (SEU):**

   - Modul pentru extinderea numărului pe 16 biți
   - Folosit pentru valorile de tip immediate, care sunt pe 9 biți

6. **Data Memory:**

   - Are un singur port de citire si scriere (wdata/rdata), pe 16 biti
   - Daca WE e activ, atunci se scriu datele din wdata la adresa indicată de immediate
   - Daca RE e activ, atunci se citesc datele de la adresa immediate în data bus, rdata
   - În cazul unei citiri din memorie, data va fi demultiplexată și încărcată în unul din regiștrii X sau Y, în funcțtie de valoarea lui IR[9] – register adress

7. **Multiplexor (MUX):**

   - Circuite logice combinaţionale cu m intrări şi o singură ieşire, care permit transferul datelor de la una din intrări spre ieşirea unică

- Selecţia intrării de la care se transferă datele se face prin intermediul unui cuvânt de cod de selecţie numit adresă, cuvânt care are n biţi

8. Arithmetic-Logic Unit (ALU):

- Este o parte din Central Processing Unit care contine operatii aritmetice si logice
- Are ca input cei doi operanzi si output rezultatul, toate pe 16 biți
- Codurile de operatii ajuta ALU sa stie ce operatie trebuie sa execute

9. Mover (MOV):

- Folosit pentru a încărca în unul din regiștrii X/Y valoarea din acumulator/immediate
- Are ca intrări de selecție semnalele imm și IR[9]

10. Demux (DEMUX):

- Folosit pentru a demultiplexa datele din memorie și a le încărca doar în unul dintre registrii X/Y

11. Flags:

- Câmp pe 4 biți, activați de operațiile ALU
- 4 flag-uri pentru cazurile de: zero, overflow, negative, carry

12. IR:

- Registru pe 16 biți, în care se încarcă instrucțiunea curentă din memoria de instrucțiuni

Instruction set of the processor:

I. **Memory Access Instructions**

1) **Load:**
LDR R, #imm

2) **Store:**
STR R, #imm

II. Data Processing Instructions

1) **Add:**
ADD x, y => ACC:=x + y
ADD x, #imm => ACC:=x + #imm

2) **Subtract:**
SUB x, y => ACC:=x - y
SUB x, #imm => ACC:=x - #imm

3) **Logical Shift Right:**
LSR x, y => ACC:=x >> y
LSR x, #imm => ACC:=x >> #imm

4) **Logical Shift Left:**
LSL x, y => ACC:=x << y
LSL x, #imm => ACC:=x << #imm

5) **Rotate Shift Right:**
RSR ws, x => ws:=x/2,
RSR ws, y => ws:=y/2

6) **Rotate Shift Left:**
RSL ws, x, y

7) **Move:**
MOV R, ACC => R := ACC
MOV R, #imm => R := #imm
(R = X/Y)

8) **Multiply:**
MUL x, y => ACC:=x * y
MUL x, #imm => ACC:=x * #imm

9) **Divide:**
DIV x, y => ACC:=x / y
DIV x, #imm => ACC:=x / #imm

10) **Modulo:**
MOD x, y => ACC:=x % y
MOD x, #imm => ACC:=x % #imm

11) **Bitwise AND:**
AND x, y => ACC:=x & y
AND x, #imm => ACC:=x & #imm

12) **Bitwise OR:**
OR x, y => ACC:=x | y
OR x, #imm => ACC:=x | #imm

13) **Bitwise XOR:**
XOR x, y => ACC:= x ^ y
XOR x, #imm => ACC:= x ^ #imm

14) **NOT:**
NOT ws, x => ws := ~x
NOT ws, y => ws := ~y

15) **Compare: CMP**
CMP x, y => x – y, fără store la rezultat in ACC
CMP x, #imm => x - #imm, fără store la rezultat in ACC

16) **Test: TST**
TST x, y => x & y, fără store la rezultat in ACC
TST x, #imm => x & #imm, fără store la rezultat in ACC

17) **Increment:**
INC R => ACC:=R+1
(R = x/y)

18) **Decrement:**
DEC R => ACC:= R – 1

III. Control Flow Instructions

1) **Branch if Zero:** BRZ
2) **Branch if Negative:** BRN
3) **Branch if Carry:** BRC
4) **Branch if Overflow:** BRO

5) **Branch Always (unconditional branch):** BRA
6) **Jump:** JMP
7) **Ret:** RET
8) **Push:** PSH
9) **POP:** POP
10) **HLT:** HLT

- **Opcodes:**

OPERATIONS	Codes
BRZ	000000 (0)
BRN	000001 (1)
BRC	000010 (2)
BRO	000011 (3)
BRA	000100 (4)
JMP	000101 (5)
RET	000110 (6)
LDR R, #immediate	000111 (7)
STR R, #immediate	001000 (8)

OPERATIONS	Codes	
	R, R Operations	**R, #immediate Operations**
ADD	001001 (9)	011000 (24)
SUB	001010 (10)	011001 (25)
LSR	001011 (11)	011010 (26)
LSL	001100 (12)	011011 (27)
RSR	001101 (13)	011100 (28)
RSL	001110 (14)	011101 (29)
MUL	001111 (15)	011110 (30)
DIV	010000 (16)	011111 (31)
MOD	010001 (17)	100000 (32)

AND	010010 (18)	100001 (33)
OR	010011 (19)	100010 (34)
XOR	010100 (20)	100011 (35)
CMP	010101 (21)	100100 (36)
TST	010110 (22)	100101 (37)
NOT	010111 (23)	100110 (38)

Operations	R, R Operations Codes
INC	100111 (39)
DEC	101000 (40)
LOG2	101001 (41)
MODULE	101010(42)
FACTRL	101011(43)
MOVR	111010(58)
MOVI	111011(59)
PSH	111100(60)
POP	111101(61)
NOP	111110(62)
HLT	111111(63)

Application-Specific Instruction Set Processor (ASIP)

1. **MODUL → |n|**
2. **FATORIAL → n!**
3. **LOG2 → $\log_2 n$**

Assembler

Am implementat 2 variante de assembler: unul pentru Windows(Assembler_Windows), unul pentru Linux(Assembler_Linux). Ambele variante sunt scrise în limbajul C.

Mod de lucru: Programul preia liniile de cod assembly scrise în fișierul program.txt și le convertește pe rând în codurile hexa corespunzătoare tabelelor de mai sus + convențiilor specificate. Codurile sunt scrise în fișierul instructions.mem care e citit în Verilog în memoria de instrucțiuni. Pentru Linux, programul trebuie compilat în C, iar apoi rulat în

terminal(dacă este rulat cu opțiunea -output, în terminal se vor afișa valorile finale din regiștrii X și Y și din acumulator). Pentru Windows programul își crează executabil care odată rulat scrie în instructions.mem.

Exemple

a^b

```
MOV X, #2
      MOV Y, #7
      PSH Y
      PSH X
      PSH X
      CMP Y, #0
      BRZ REZULTAT1
      CMP Y, #1
      BRZ REZULTAT2
LOOP:  POP Y
      POP X
      MUL X, Y
      POP Y
      PSH X
      MOV X, ACC
      SUB Y, #1
      MOV Y, ACC
      ADD X, #0
      POP X
      PSH Y
      PSH X
      MOV X, ACC
      PSH X
      CMP Y, #1
      BRZ DONE
      BRA LOOP
REZULTAT1: ADD Y, #1
BRA DONE
REZULTAT2: ADD X, #0
DONE:  HLT
```

Numărul de biți de 1 dintr-un număr

```
MOV X, #255
      MOV Y, #0
      PSH Y
LOOP:  AND X, #1
      POP Y
      PSH X
      MOV X, ACC
      ADD X, Y
      MOV Y, ACC
      POP X
      PSH Y
      LSR X, #1
      MOV X, ACC
      CMP X, #0
      BRZ DONE
      BRA LOOP
DONE:  ADD Y, #0
       HLT
```

Suma cifrelor unui număr

```
        MOV X, #199 //număr
        MOV Y, #0 //suma
LOOP:   CMP X, #0
        BRZ DONE
        MOD X, #10
        PSH X
        PSH Y
        MOV Y, ACC
        POP X
        ADD Y, X
        MOV Y, ACC
        POP X
        DIV X, #10
        MOV X, ACC
        BRA LOOP
DONE:   ADD Y, #0
        HLT
```