UNIVERSITATEA POLITEHNICA TIMIȘOARA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
SECȚIA CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI

Pocket Calculator with General Purpose Processor

Fundamente de Ingineria Calculatoarelor

Team: 0xcHa0s

Podean Roxana-Andreea, Ploscaru Carla, Plăvăț Vlad,
Pop Adrian, Pop Tudor-Antoniu, Pop Alexandru

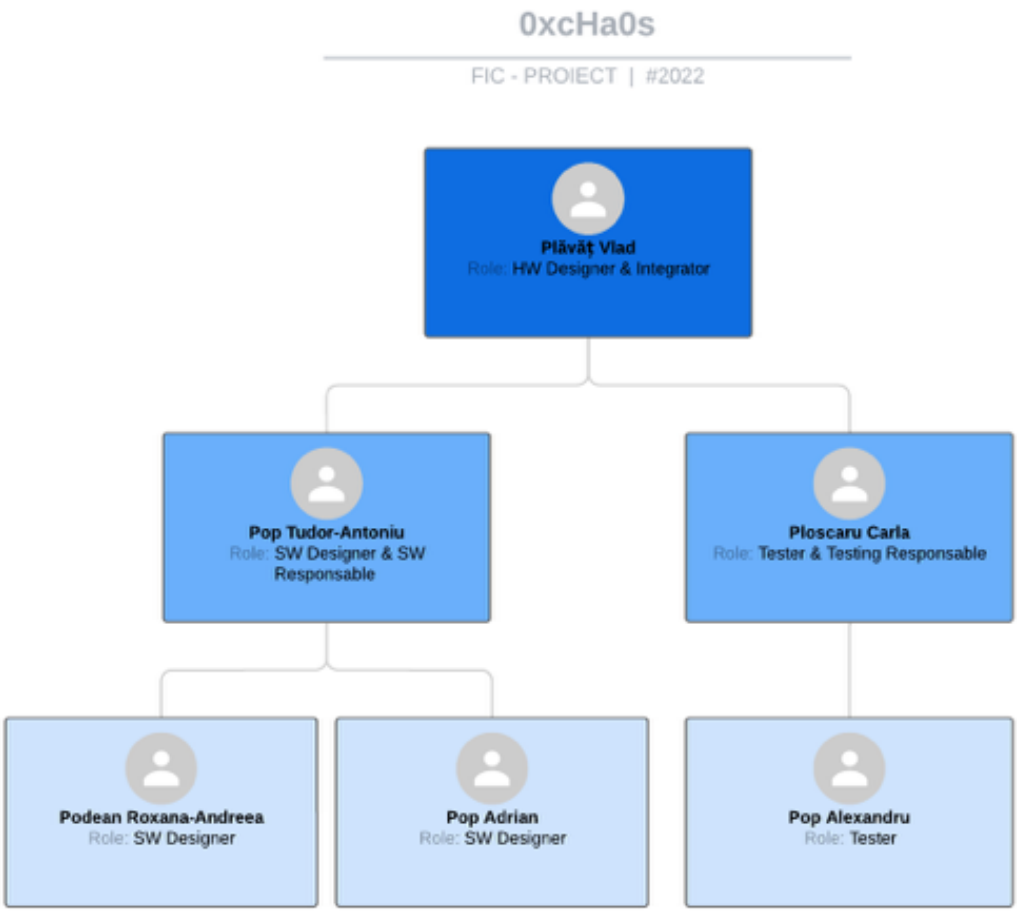
Subgrupa 5.1

An universitar: 2022-2023

Cuprins

CAPITOLUL 1

1.1 Prezentarea echipei



1.2 Tema Proiectului

Implementarea unui procesor cu următoarele componente:

- ⇒ Două registre de uz general: X, Y
- ⇒ Registru acumulator: A
- ⇒ Unitate aritmetică-logică
- ⇒ Unitate de control
- ⇒ Memorie de instrucțiuni și date
- ⇒ Unitate de extindere a semnului
- ⇒ Registerele procesorului: PC, SP, Flag

1.3 Aspecte Generale

Registre:

- ⇒ Acumulator: 16-bit
- ⇒ Două registre de uz general: X & Y 16-bit
- ⇒ Patru registre Flag: Zero, Negativ, Carry, Overflow 4-bit
- ⇒ Stivă: 16-bit
- ⇒ Program counter: 10-bit

Dimensiunea cuvântului: 16-bit.
Stiva crește invers în memorie, iar memoria este adresabilă la nivel de cuvânt.
Program counter-ul pornește de la 0 și este incrementat la fiecare instrucțiune.
Instrucțiunile CALL și JMP pun program counter-ul pe stivă, iar instrucțiunea RET elimină program counter-ul de pe stivă.
Toți parametrii procedurilor și valoarea de returnare sunt pasate folosind stiva.
Doar instrucțiunile aritmetice și logice au dreptul de a schimba registrul Flag

Dimensiunea instrucțiunii este de 16-bit: 6-bit opcode, 1-bit register address, 9-bit immediate size.

CAPITOULUL 2

2.1 Setul de Instrucțiuni

În tabelul de mai jos sunt reprezentate opcode-urile instrucțiunilor, biții mai puțin semnificativi fiind reprezentați pe prima linie iar cei mai semnificativi pe prima coloană.

	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
000	NOP	PUSH/POP	LOAD	STORE	LOADB	STOREB	MOVI	MOVR
001	RAD	POW						
010	BRA	BRE/BRZ	BNE	BRN/BLT	BRP/BGT	BLE	BGE	BRC
011	BRO						JMP	RET
100	ADDRI	ADDAI	ADDRR		SUBRI	SUBAI	SUBRR	
101	SHIFT		INC	DEC	MULRR	MULAI	DIVRR	DIVAI
110	MODRR	MODAI	ANDRR	ANDAI	ORRR	ORAI	XORRR	XORAI
111	NOT	NEG	CMPRI	CMPAI	CMPRR	TSTRI	TSTAI	TSTRR

Instrucțiunea NOP:

Bits	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
NOP	0	0	0	0	0	0	-									

Instrucțiunea **NOP** nu modifică starea procesorului.
Sintaxă: NOP

Instrucțiunile PUSH & POP:

Bits	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	0	0	0	0	0	1	C	D		-						

Pentru C egal cu 0 se va executa instrucțiunea **POP** iar pentru C egal cu 1 instrucțiunea **PUSH**. În funcție de valoarea lui D se va selecta registrul destinație/sursă dintre X (00), Y (01) sau ACC (10).

Instrucțiunea **PUSH** decrementează stack pointer-ul și salvează pe stivă valoarea unui registru, în timp ce instrucțiunea **POP** citește de pe stivă și salveaza în registru, incrementând apoi stack pointer-ul.
Sintaxă: `POP X/Y/ACC, PUSH X/Y/ACC`

Instrucțiunile LOAD & STORE:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Load	0	0	0	0	1	0	D	ADDRDAT								
Store	0	0	0	0	1	1	D	ADDRDAT								

În funcție de valoarea lui D se va selecta unul din registrele X (0) sau Y (1).

LOAD și **STORE** sunt două instrucțiuni care transferă date între registre și memorie de la adresa (variabila) specificata. Instrucțiunea **LOAD** citește datele din memorie în registre iar instrucțiunea **STORE** scrie datele din registre în memorie.
Sintaxă: `LOAD X/Y VarSCR, STORE X/Y VarDST`

Instrucțiunile LOADB & STOREB:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LOADB	0	0	0	1	0	0	D	ADDRB								
STOREB	0	0	0	1	0	1	D	ADDRB								

Sintaxă: În funcție de D se va selecta una dintre variantele de mai jos.

- ⇒ D = 0 | LOADB X var - X = Mem[var + Y]
- ⇒ D = 1 | LOADB Y var - Y = Mem[var + X]
- ⇒ D = 0 | STOREB X var - Mem[var + X] = Y
- ⇒ D = 1 | STOREB Y var - Mem[var + Y] = X

Instrucțiunea MOV:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVI	0	0	0	1	1	0	D	IMM								
MOVR	0	0	0	1	1	1	D		S		-					

Instrucțiunea MOV mută informație dintr-o parte în cealaltă. În cazul **MOVI** se mută un immediate în X sau Y iar în cazul **MOVR** se mută un registru într-un alt registru.
Sintaxă: `MOV X/Y Imm`, `MOV X/Y/ACC X/Y/ACC`

Instrucțiunea RAD:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Flag
RAD	0	0	1	0	0	0	-										CF, OF

Instrucțiunea **RAD** calculează radicalul numărului format din registrul X concatenat cu registrul acumulator. CF va fi setat in cazul calculului cu număr negativ iar OF va fi setat în cazul în care rezultatul nu încape in registrul destinație pe 16 biți.
Sintaxă: `RAD`

Instrucțiunea POW:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Flag
POW	0	0	1	0	0	1	E		-								CF, OF

În funcție de valoarea lui E se va selecta registrul destinație/sursă dintre X (00), Y (01) sau ACC (10).

Instrucțiunea **POW** calculează rezultatul numărului aflat în registrul acumulator ridicat la puterea dată de registrul E, stocând rezultatul în registrul X concatenat cu registrul acumulator. CF va fi setat în cazul ridicării lui 0 la puterea 0 sau in cazul calculului cu numere negative iar OF va fi setat în cazul în care rezultatul nu încape in registrul destinație pe 32 de biți.
Sintaxă: `POW`

Instrucțiunile Branch:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Condition
BRA	0	1	0	0	0	0	ADDRINST										mereu
BRE/BRZ	0	1	0	0	0	1	ADDRINST										ZF = 1
BNE	0	1	0	0	1	0	ADDRINST										ZF = 0
BRN/BLT	0	1	0	0	1	1	ADDRINST										NF != OF
BRP/BGT	0	1	0	1	0	0	ADDRINST										ZF = 0 & NF = OF
BLE	0	1	0	1	0	1	ADDRINST										ZF = 1 & NF != OF
BGE	0	1	0	1	1	0	ADDRINST										NF = OF
BRC	0	1	0	1	1	1	ADDRINST										CF = 1
BRO	0	1	1	0	0	0	ADDRINST										OF = 1

Instrucțiunile de branch modifică program counter-ul în funcție de condiția îndeplinită.

Instrucțiunea JMP:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	1	1	1	1	0	ADDRINST									

Instrucțiunea **JMP** apelează o subrutină, punând adresa de revenire pe stivă.
Sintaxă: `JMP`

Instrucțiunea RET:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RET	0	1	1	1	1	1	ADDRINST									

Instrucțiunea **RET** este apelată la finalul unei subrutine și citește adresa de revenire de pe stivă și o scrie în program counter.
Sintaxă: `RET`

Instrucțiunea ADD:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDRI	1	0	0	0	0	0	X/Y	IMM								
ADDAI	1	0	0	0	0	1	-	IMM								
ADDRR	1	0	0	0	1	0	X/Y/ACC		X/Y/ACC		-					

Instrucțiunea **ADD** execută operația de adunare**.** Pentru cazul **ADDRI** se execută adunare între registrul X (0) sau Y (1) și un Immediate. **ADDAI** execută operația de adunare între registrul ACC și un Immediate. În timp ce în cazul **ADDRR** se execută operația de adunare între registrele X (00), Y (01) sau ACC (10).
Sintaxă: `ADD X/Y/ACC X/Y/ACC/IMM`

Instrucțiunea SUB:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUBRI	1	0	0	1	0	0	X/Y	IMM								
SUBAI	1	0	0	1	0	1	-	IMM								
SUBRR	1	0	0	1	1	0	X/Y/ACC		X/Y/ACC		-					

Instrucțiunea **SUB** execută operația de scădere**.** Pentru cazul **SUBRI** se execută scădere între registrul X (0) sau Y (1) și un Immediate. **SUBAI** execută operația de scădere între registrul ACC și un Immediate. În timp ce în cazul **SUBRR** se execută operația de scădere între registrele X (00), Y (01) sau ACC (10).
Sintaxă: `SUB X/Y/ACC X/Y/ACC/IMM`

Instrucțiunea SHIFT:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHIFT	1	0	1	0	0	0	Operation(M)			X/Y/ACC		I	QQQQ			

Operation	Assembly Syntax	Bits		
Right Shift Logic	RSL	0	0	0
Left Shift Logic	LSL	0	0	1
Right Shift Arithmetic	RSA	0	1	0
Left Shift Arithmetic	LSA	0	1	1
Rotate Right	RSR	1	0	0
Rotate Left	LSR	1	0	1
Rotate Carry Right	RSC	1	1	0
Rotate Carry Left	LSC	1	1	1

Instrucțiunea **SHIFT** execută una din operațiile enumerate în tabelul de mai sus în funcție de combinția biților dați de Operation. Dacă operația se termină în **M** *(ex. RSLM)***,** valoarea cu care se șiftează este transformată în modulo 16. Operația aleasă poate fi executată cu registrele X (00), Y (01) sau ACC (10). Dacă **I** este 1, următorii 4 biți vor reprezanta un Immediate, în caz contrar următorii 2 biți fac selectia între X (00), Y (01) sau ACC (10) iar penultimul bit decide dacă se va șifta cu valoarea aflată în registru sau cu valoarea transformată în modulo 16.
Sintaxă: `Operation(M) X/Y/ACC X/Y/ACC/IMM`

Instrucțiunile INC & DEC:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INC	1	0	1	0	1	0	X/Y/ACC		-							
DEC	1	0	1	0	1	1	X/Y/ACC		-							

Instrucțiunea **INC** va incrementa unul dintre registrele X (00), Y (01) sau ACC (10).
Instrucțiunea **DEC** va decrementa unul dintre registrele X (00), Y (01) sau ACC (10).
Sintaxă: `INC X/Y/ACC, DEC X/Y/ACC`

Instrucțiunea MUL:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULRR	1	0	1	1	0	0	X/Y/ACC		X/Y/ACC		-					
MULAI	1	0	1	1	0	1	-	IMM								

Instrucțiunea **MUL** va executa operația de înmulțire. În cazul **MULRR** se va executa înmulțirea între registrele X (00), Y (01) sau ACC (10). Pentru cazul **MULAI** se se va executa înmulțirea între registrul acumulator și un Immediate.
Sintaxă: `MUL X/Y/ACC X/Y/ACC, MUL ACC IMM`

Instrucțiunea DIV:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIVRR	1	0	1	1	1	0	X/Y/ACC		X/Y/ACC		-					
DIVAI	1	0	1	1	1	1	-	IMM								

Instrucțiunea **DIV** va executa operația de împărțire. În cazul **DIVRR** se va executa împărțire între registrele X (00), Y (01) sau ACC (10). Pentru cazul **DIVAI** se se va executa împărțire între registrul acumulator și un Immediate.
Sintaxă: `DIV X/Y/ACC X/Y/ACC, DIV ACC IMM`

Instrucțiunea MOD:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODRR	1	1	0	0	0	0	X/Y/ACC		X/Y/ACC		-					
MODAI	1	1	0	0	0	1	-	IMM								

Instrucțiunea **MOD** va executa operația de modulo. În cazul **MODRR** se va executa modulo între registrele X (00), Y (01) sau ACC (10). Pentru cazul **MODAI** se se va executa modulo între registrul acumulator și un Immediate.
Sintaxă: `MOD X/Y/ACC X/Y/ACC, MOD ACC IMM`

Instrucțiunea AND:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ANDRR	1	1	0	0	1	0	X/Y/ACC		X/Y/ACC		-					
ANDAI	1	1	0	0	1	1	-	IMM								

Instrucțiunea **AND** va executa operația logică ȘI. În cazul **ANDRR** se va executa operația logică ȘI între registrele X (00), Y (01) sau ACC (10). Pentru cazul **ANDAI** se se va executa operația logică ȘI între registrul acumulator și un Immediate.
Sintaxă: `AND X/Y/ACC X/Y/ACC, AND ACC IMM`

Instrucțiunea OR:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ORRR	1	1	0	1	0	0	X/Y/ACC		X/Y/ACC		-					
ORAI	1	1	0	1	0	1	-	IMM								

Instrucțiunea **OR** va executa operația logică SAU. În cazul **ORRR** se va executa operația logică SAU între registrele X (00), Y (01) sau ACC (10). Pentru cazul **ORAI** se se va executa operația logică SAU între registrul acumulator și un Immediate.
Sintaxă: `OR X/Y/ACC X/Y/ACC, OR ACC IMM`

Instrucțiunea XOR:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XORRR	1	1	0	1	1	0	X/Y/ACC		X/Y/ACC		-					
XORAI	1	1	0	1	1	1	-	IMM								

Instrucțiunea **XOR** va executa operația logică SAU-EXCLUSIV. În cazul **XORRR** se va executa operația logică SAU-EXCLUSIV între registrele X (00), Y (01) sau ACC (10). Pentru cazul **XORAI** se se va executa operația logică SAU-EXCLUSIV între registrul acumulator și un Immediate.
Sintaxă: `XOR X/Y/ACC X/Y/ACC, XOR ACC IMM`

Instrucțiunile NOT & NEG:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOT	1	1	0	1	1	0	X/Y/ACC		-							
NEG	1	1	0	1	1	1	X/Y/ACC		-							

Instrucțiunea **NOT** va nega toți biții registrului X (00), Y (01) sau ACC (10) în timp ce instrucțiunea **NEG** va calcula opusul numărului din registrul X (00), Y (01) sau ACC (10).
Sintaxă: `NOT X/Y/ACC, NEG X/Y/ACC`

Instrucțiunea CMP:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMPRI	1	1	1	0	1	0	X/Y	IMM								
CMPAI	1	1	1	0	1	1	-	IMM								
CMPRR	1	1	1	1	1	0	X/Y/ACC		X/Y/ACC		-					

Instrucțiunea **CMP** compară două valori**.** Pentru cazul **CMPRI** se va compara valoarea din registrul X (0) sau Y (1) cu valoarea unui Immediate. **CMPAI** va compara valoarea din registrul ACC cu valoarea unui Immediate. În timp ce în cazul **CMPRR** se compară valorile dintre două registre X (00), Y (01) sau ACC (10).
Sintaxă: `CMP X/Y/ACC X/Y/ACC/IMM`

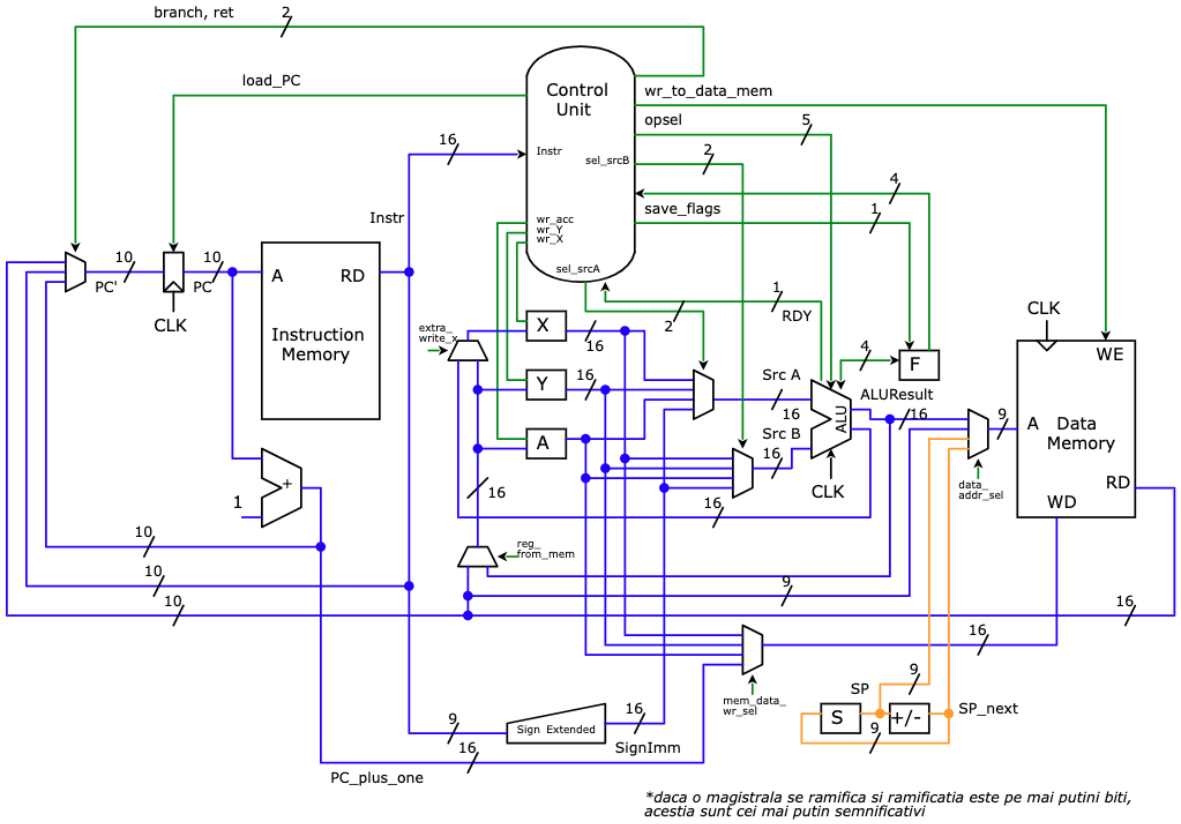
Instrucțiunea TST:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TSTRI	1	1	1	1	0	1	X/Y	IMM								
TSTAI	1	1	1	1	1	0	-	IMM								
TSTRR	1	1	1	1	1	1	X/Y/ACC		X/Y/ACC		-					

Instrucțiunea **TST** compară două valori folosind un ȘI-LOGIC**.** Pentru cazul **TSTRI** se va compara valoarea din registrul X (0) sau Y (1) cu valoarea unui Immediate folosind un ȘI-LOGIC. **TSTAI** va compara valoarea din registrul ACC cu valoarea unui Immediate folosind un ȘI-LOGIC. În timp ce în cazul **TSTRR** se compară valorile dintre două registre X (00), Y (01) sau ACC (10) folosind un ȘI-LOGIC.
Sintaxă: `TST X/Y/ACC X/Y/ACC/IMM`

CAPITOLUL 3

3.1 Descrierea Hardware



Schema hardware prezentată mai sus este alcătuită din următoarele componente:

- ⇒ Unitate de control
- ⇒ Memoria de date
- ⇒ ALU
- ⇒ Memoria de instrucțiuni
- ⇒ Sign Extend
- ⇒ Registrele X, Y, A
- ⇒ Registrul F

Unitatea de control este o componentă a unității centrale de procesare a unui computer care dirijează funcționarea procesorului. Aceasta activează semnalele de control necesare corespunzătoare instrucțiunii executate la momentul dat. Majoritatea instrucțiunilor sunt *single cycle* iar pentru instrucțiunile care nu se pot executa într-un singur ciclu de clock (pow & rad) se folosește semnalul *ready* (se pune semnalul pe 0, până când se termină de executat instrucțiunea).

Memoria de date este o memorie RAM în care se poate scrie și citi, nu este inițializată, stochează variabilele inițiale și stiva. Dimensiunea este de 512 de cuvinte. *(adresabil la nivel de cuvânt).* Citirea este *same cycle*, adică atunci când se primește adresa se transmit și datele.

ALU este un circuit, care poate efectua operații aritmetice și logice. După ce se execută operația primită de la unitatea de control prin semmanlul *opsel,* ALU va pune semnalul *ready* pe 1.

Memoria de instrucțiuni este o memorie din care se poate doar citi, nu și scrie. Aici se stochează codul instrucțiunii *(opcode),* care urmează a fi executat.

Sign Extend este operația care replică bitul de semn.

Registrele X, Y sunt *general purpose registers* iar registrul **A** este cel în care se salvează marea majoritate a rezultatelor operațiilor.

Registrul F este format din ZF, NF, CF și OF. Acestea sunt setate în funcție de rezultatul operațiilor aritmetice sau logice.

Registrul PC este registrul care reține adresa instrucțiunii curente, pe care o transmite la memoria de instrucțiuni.

Registrul SP este un registru al cărui scop este reținâ adresa ultimei variabile adăugate în stivă.

CAPITOLUL 4

4.1 Descriere Software

Verilog:

Modulul *data_mem* face citirea datelor *(variabilelor)* din memorie și le stochează. Memoria este implementată ca un vector de 512 poziții cu lățimea de 16 biți. Datele se citesc în același ciclu de *clk* iar scrierea se face doar pe frontul acestuia.

Modulul *instr_mem* este memoria de instrucțiuni. Aceasta se iniţializează din fişierul *main.bin*.

Modulul *cpu* conține instanțe ale modulelor: *control unit, alu, registre* și *sign extend unit*. Sunt implementate multiplexoare prin always combinaţional. De asmenea se calculează și adresa următoarei instrucțiuni și valoarea următorului stack pointer pe baza unor semnale de control.

Modulul *SignExtend* replică bitul de semn.

Modulul *controlUnit* este implementat pur combinațional. Primeşte întreg opcode-ul pe 16 biţi de la memoria de instrucţiuni şi este implementat utilizând *case statement*. În fiecare *case* se activează semnalul corespunzător şi se setează semnalul de trecere la următoarea instrucţiune doar dacă e activ *ready*-ul, altfel nu se trece la următoarea instrucţiune. Semnalul pe care *control unit*-ul îl dă lui *alu* se numeşte *opsel* și este pe 5 biţi.

ALU conține instanțe ale modulelor pentru înmulțire, împărțire, ridicare la putere, radical. Acesta este implementat folosind *always combinațional* și un *case statement,* similar cu unitatea de control.

Pentru operațiile simple, în *case*-ul respectiv se determină rezultatul și se setează *flag*-urile. În cazul operațiilor mai complicate, se va prelua rezultatul, flag-urile și semnalul de *ready* de la submodulele respective.

Operația de înmulțire, împărțire și modulo au fost implementate folosind operatorii "*", "/", "%". Operația de ridicare la putere a fost implementată printr-un automat cu stări finite care efectuează înmulțiri repetate. Operația de extragere a radicalului a fost implementată printr-un automat cu stări finite care funcționează similar cu metoda de extragere a radicalului în baza 10.

Asamblor:

Asamblorul este un program care asigură traducerea limbajului simbolic în limbajul calculatorului. Acesta are ca date de intrare un fișier în limbaj de asamblare, specific procesorului, și ca fișier de ieșire, fisierul executabil, continând instrucțiunile în format binar.

Programul va primi în linia de comandă numele fisierului în limbaj de asamblare, iar fișierul de ieșire se va numi *main.bin*. În cazul în care asamblarea nu reușește, fișierul *main.bin* anterior va fi șters. La o compilare cu succes, se va mai crea și fișierul *mem_map.txt*, în care se va regăsi corespondenta între variabile, respectiv label-uri și adrelese la care se află fiecare. Asamblorul este implementat folosind limbajul de programare C.

assembler.c: identifică label-urile și declarările de variabile, citește din fișierul în limbaj de asamblare și scrie în fișierul rezultat în urma asamblării

opcode_behaviour.h: translatează liniile în cod masina

Funcționarea programului main: se parcurge fișierul de intrare pentru prima dată și se rețin adresele label-urilor, apoi se parcurge a doua oară și se traduc instrucțiunile**.** odata transalată se scrie în fișierul de ieșire rezultatul asamblării și adresele label-urilor și a variabilelor din fișierul mem_map.txt

Prima parcurgere constă în parcurgerea linie cu linie a fișierului, transformarea tuturor literelor în majusculă, eliminarea spaților de la începutul rândului și ignorarea linilor care nu încep cu literă, acestea fiind considerate comentariu. O linie care e formată dintr-un singur cuvânt și se termină în caract ":", e considerată label. Dacă un label cu același nume a mai fost declarat deja, se generează eroare. Altfel se va adauga în array-ul care reține label-urile și adresele corespunzatoare lor.

La a doua parcurgere, se vor analiza declarările de date. Variabilele se pot declara pe 1 word (16 biți), ex: data var1, sau ca vectori, care ocupă o zonă continuă de memorie de n cuvinte, ex. data vect 15 (pentru a rezerva 15 elemente). Variabilele sunt plasate în ordinea declarărilor, începând cu adresa 0. Primul cuvant este intrepretat ca numele instrcțiunii, care va fi căutat într-un vector cu nume de instrucțiuni cunoscute. Se va apela funcția de translatare corespunzătoare acelei instrucțiuni. Vectorul menționat conține numele instrucțiunii și funcția apelată pentru a o translata.

Pentru preluarea operanzilor instrucțiunii curente se folosește funcția analize_arguments. În mod generic, funcția de translatare a unei instrucțiuni apelează analize_arguments, dupa care verifică numărul de argumente, apoi pe baza lor va returna codul corespunzator instrucțiunii. De asemenea, poti fi generate diferite erori care opresc procesul de asamblare.

CAPITOLUL 5

5.1 Testare

Pentru a testa **Control Unit**, se va folosi fișierul *controlU.bin*, care va conține pe fiecare line câte un test. Din acest fișier, se citesc semnalele de intrare pt Control Unit, iar apoi, din fișierul de intrare, se citesc valorile semnalelor de control considerate corecte, care urmeaza a fi comparate cu rezultatul generat de instanța Control Unit. Fișierul de teste este generat manual.

Testarea **ALU** cuprinde testarea operațiilor. Pentru fiecare operație se vor genera 1000 de teste cu operanzi aleatori. Se verifică dacă rezultatul și flag-urile sunt corecte. După fiecare operație se afișeaza un mesaj în consolă.

Pentru testearea lui **POW** și **SQRT**, mai întâi se fac câteva teste pentru cazurile excepționale. Se testează apoi, pentru o plajă largă de valori întregi, ale operanzilor, toate combinațiile rezultate.

Testarea întregii funcționalități se face prin simularea modulului *top_module*. În prealabil vom fi generat fișierul main.bin, folosind asamblorul. *Main.bin* trebuie să se afle în același fișier cu *top_module*.

Pentru o verificare mai simplă, putem termina programul cu un loop infinit la aceeași instrucțiune. Acest lucru face ca program counter-ul să nu își mai modifice valoarea. Pentru a vedea rezultatele se poate analiza memoria de date.

CAPITOLUL 6

6.1 Bibliografie

⇒ https://www.sciencedirect.com/topics/computer-science/general-purpose-processor

⇒ https://en.wikibooks.org/wiki/Embedded_Control_Systems_Design/Processors

⇒ https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html

⇒ https://github.com/vprabhu28/16-Bit-CPU-using-Verilog

⇒ https://www.european-processor-initiative.eu/general-purpose-processor/

⇒ https://www.quora.com/What-is-a-difference-between-general-purpose-processor-and-single-purpose-processor

⇒ https://en.wikipedia.org/wiki/History_of_general-purpose_CPUs

⇒ https://www.cantorsparadise.com/the-square-root-algorithm-f97ab5c29d6d