Raport Implementare Procesor MIPS 16

Vlad-Alexandru Bartolomei

April 26, 2023

Contents

1 Descrierea procesorului implementat **3**

2 Cele patru instrucțiuni alese **3**

3 Tabelul cu valori pentru instrucțiunile alese **4**

4 Programul executat de către procesor **5**

5 Tracing **8**

6 Corectitudinea în VHDL, a.k.a. Schematic **9**

7 Mențiune **10**

1 Descrierea procesorului implementat

Proiectul de față implementează un procesor tip MIPS-16, adică un Microprocessor without Interlocked Pipelined Stages pe 16 biți (instrucțiunile, i.e. codul mașină, și rezultatele ocupă 16 biți, adică 2 bytes). Ținând seama de codificarea consacrată a codului mașină (again, pe 16 biți), ne "bucurăm" de posibilitatea de a adresa doar 8 registre ($0 to $7) în blocul nostru de registre (RegFile).

2 Cele patru instrucțiuni alese

Din cele patru cerute în îndrumător, am considerat că două instrucțiuni îmi sunt suficiente pentru a-mi desfășura activitatea:

- Logical XOR, având identificatorul xor → Instrucțiune de tip R.
- Branch Greter, având identificatorul bgt → Instrucțiune de tip I.

Următorul tabel va reda complet instrucțiunile folosite pe parcursul execuției întregului cod mașină:

Tipul instrucțiunii	**Numele Instrucțiunii**	**Identificator**
Instrucțiune de tip R	Addition (Adunare)	add
	Substraction (Scădere)	sub
	Shift Left Logical (cu shift amount)	sll
	Shift Right Logical (cu shift amount)	srl
	Logical AND	and
	Logical XOR	xor
Instrucțiune de tip I	Add immediate	addi
	Load Word	lw
	Store Word	sw
	Branch on Equal	beq
	Branch on Greater	bgt
Instrucțiune de tip J	Jump	j

3 Tabelul cu valori pentru instrucțiunile alese

Instrucțiune	Opcode (instr (15..13))	RegDst	ExtOp	ALUSrc	Branch on Equal	Branch on Greater	Jump	MemWrite	MemtoReg	RegWrite	ALUOp (2 downto 0) (coincide cu Opcode)	Func (instr (2..0))	ALUCtrl (2 downto 0)
add	000	1	x	0	0	0	0	0	0	1	000	000	000
sub	000	1	x	0	0	0	0	0	0	1	000	001	001
sll	000	1	x	0	0	0	0	0	0	1	000	010	010
srl	000	1	x	0	0	0	0	0	0	1	000	011	011
and	000	1	x	0	0	0	0	0	0	1	000	100	100
xor	000	1	x	0	0	0	0	0	0	1	000	110	110
addi	001	1	1	1	0	0	0	0	0	1	001	xxx	000
beq	100	0	1	0	1	0	0	0	x	0	100	xxx	001
bgt	101	0	1	0	0	1	0	0	x	0	101	xxx	101
lw	010	1	1	1	0	0	0	0	1	1	010	xxx	000
sw	011	0	1	1	0	0	0	1	x	0	011	xxx	000
j	111	0	x	x	0	0	1	0	x	0	111	xxx	xxx

4 Programul executat de către procesor

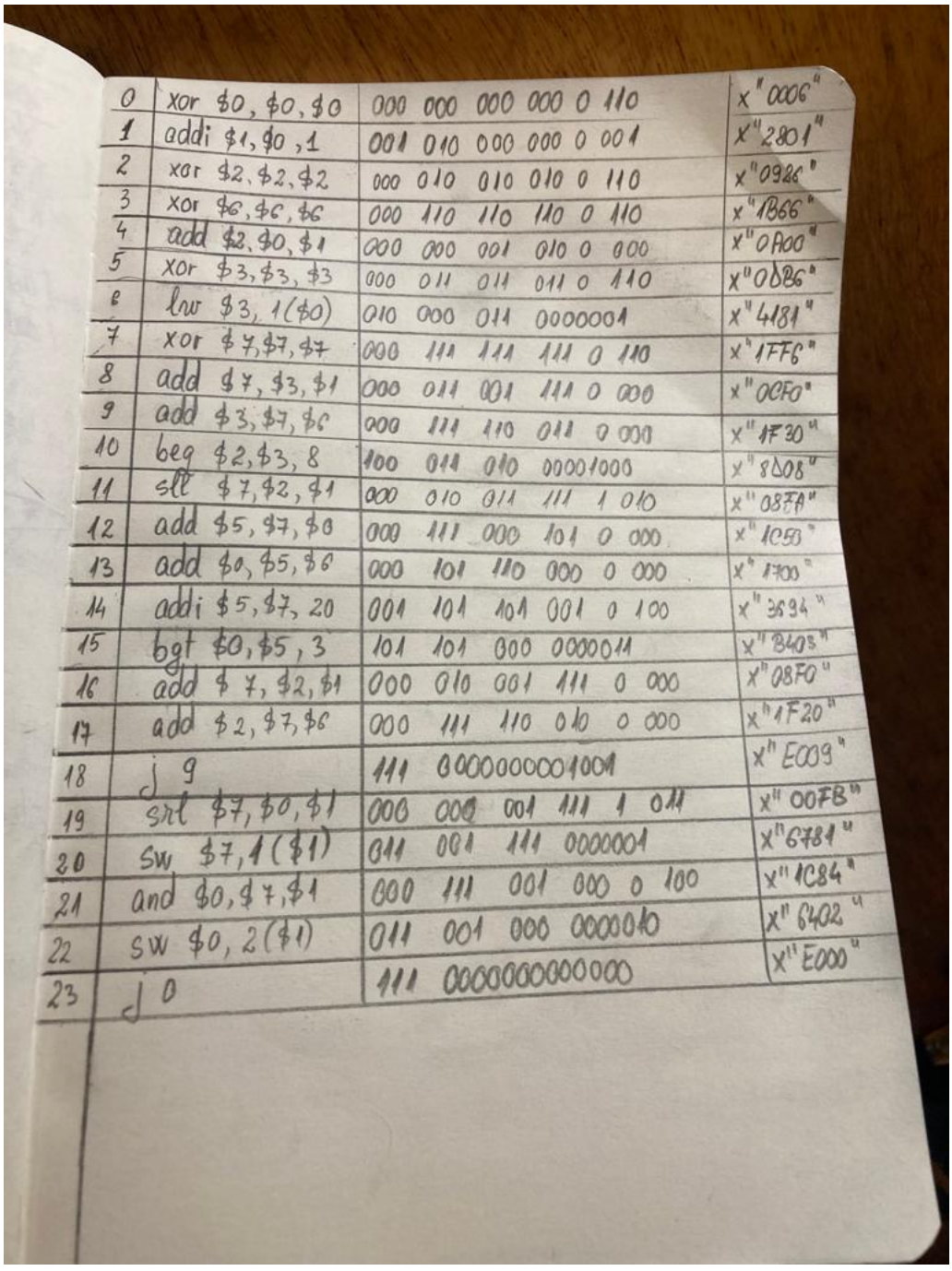
Programul propus pentru acest procesor iterează cu o variabilă în intervalul [1, N] (ocazie cu care citim N din memorie, i.e. facem lw) și adună într-o sumă (pe care o inițializează cu 0 prin operația xor) valorile iteratorului, fiecare înmulțită cu 2 (shiftare sau deplasare logică left).

Pentru a parcurge toate numerele de la 1 la N, se implementează o buclă cu număr necunoscut de pași, tip while, cu ajutorul instrucțiunilor beq și j. Pentru a vedea utilitatea instrucțiunii bgt, am decis ca, în buclă, în caz că suma depășește valoarea 20 (impusă direct în codul mașină), să iasă forțat din while.

După while, suma "își revine la normal" , adică o împărțim cu 2 prin shiftare sau deplasare logică la dreapta. O particularitate a procesorului MIPS 16, rezultată tot din codificarea instrucțiunilor, este că îi revine un singur bit câmpului shift amount (shiftăm sau nu, deci cu o poziție).

```c
#include <stdio.h>
#define N 20

int main() {
    short int i = 1;
    int sum = 0;
    while(i <= N) {
        sum = sum +(i << 1);
        printf( format: "sum <-- %hi << 1, i.e. %hi\n", i, i << 1);
        printf( format: "%d\n", sum);
        if(sum > 9998)
            goto out;
        i++;
    }
    out: sum = sum >> 1;
    printf( format: "Final sum is = %x\n", sum);
    if(sum&00000001 == 1)
        printf( format: "numar impar");
    else printf( format: "numar par");
    return 0;
}
```



0	xor $0, $0, $0	000 000 000 000 0 110	x"0006"
1	addi $1, $0, 1	001 010 000 000 0 001	x"2801"
2	xor $2, $2, $2	000 010 010 010 0 110	x"0926"
3	xor $6, $6, $6	000 110 110 110 0 110	x"1B66"
4	add $2, $0, $1	000 000 001 010 0 000	x"0A00"
5	xor $3, $3, $3	000 011 011 011 0 110	x"0DB6"
6	lw $3, 1($0)	010 000 011 0000001	x"4181"
7	xor $7, $7, $7	000 111 111 111 0 110	x"1FF6"
8	add $7, $3, $1	000 011 001 111 0 000	x"0CF0"
9	add $3, $7, $6	000 111 110 011 0 000	x"1F30"
10	beq $2, $3, 8	100 011 010 00001000	x"8D08"
11	sll $7, $2, $1	000 010 011 111 1 010	x"09FA"
12	add $5, $7, $0	000 111 000 101 0 000	x"1C50"
13	add $0, $5, $6	000 101 110 000 0 000	x"1700"
14	addi $5, $7, 20	001 101 101 001 0 100	x"3694"
15	bgt $0, $5, 3	101 101 000 0000011	x"B403"
16	add $7, $2, $1	000 010 001 111 0 000	x"08F0"
17	add $2, $7, $6	000 111 110 010 0 000	x"1F20"
18	j 9	111 00000000001001	x"E009"
19	srl $7, $0, $1	000 000 001 111 1 011	x"00FB"
20	sw $7, 1($1)	011 001 111 0000001	x"6781"
21	and $0, $7, $1	000 111 001 000 0 100	x"1C84"
22	sw $0, 2($1)	011 001 000 0000010	x"6402"
23	j 0	111 00000000000000	x"E000"

5 Tracing

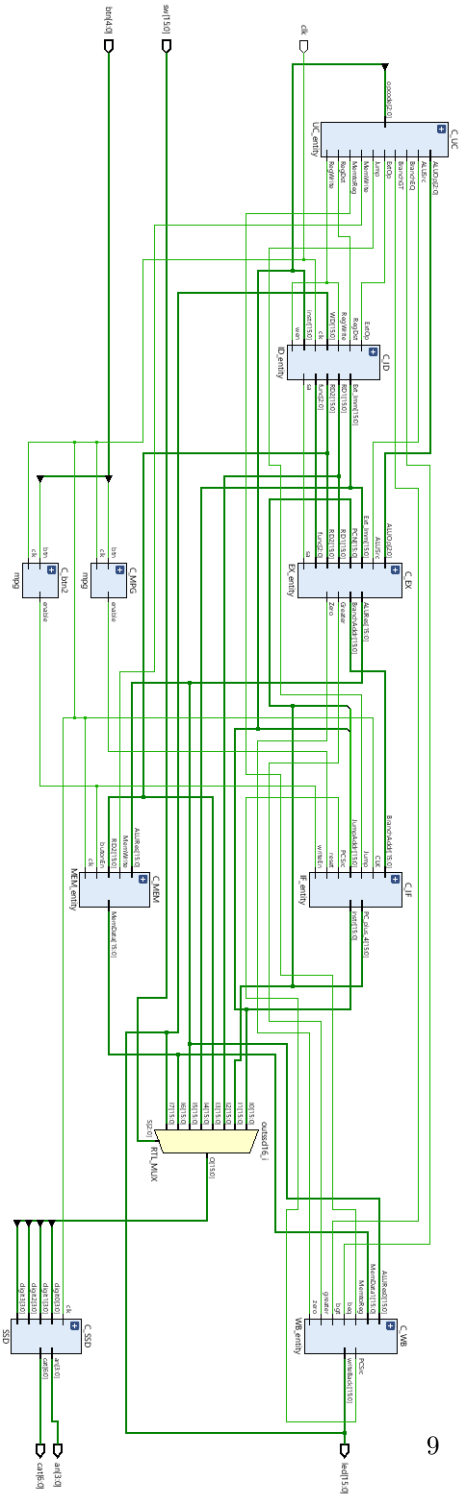
În încercarea de a urmări codul assembly-mips de mai sus, vom ține cont de următoarele alegeri:

- $0 = suma
- $1 = 1 (aici vom ține constant valoarea 1)
- $2 = i (iteratorul nostru)
- $3 = N
- $7 = registru auxiliar, folosit în operații precum add
- $6 = 0 (aici vom ține constant valoarea 0)

La liniile 0-5 se inițializează unii dintre regiștrii necesari și se stochează în $2 (iterator) valoarea 1. La liniile 6-9 se aduce din memoria de date în registrul 3 valoarea lui N dorită și se incrementează. La linia 10 intrăm în buclă, aspect marcat fiind de instrucțiunea beq. Bucla ține până la linia 18 (j 0). În buclă, se ia fiecare valoare a iteratorului în registrul auxiliar $7 și se înmulțește cu 2, iar apoi se adună la sumă ($0). De remarcat este linia 15, cu instrucțiunea bgt, care, în caz că suma curentă depășește valoarea 20, are efect de break asupra buclei (iese forțat din ea).

La linia 19 împărțim înapoi cu 2 suma finală. La linia 20 salvăm acest rezultat în memorie, la adresa 2. La linia 21 extragem ultimul bit al sumei, pentru a testa paritatea; acesta este salvat în memorie la adresa 3. La final programul sare la început, iar programul se poate reexecuta

6 Corectitudinea în VHDL, a.k.a. Schematic



7 Mențiune

Instrucțiunea add nu suportă următorul gen de operație: add $2, $2, $0 din motive, consider eu, de bun simț. Pentru că ar încerca în mod concomitent să acceseze registrul sursă 2 și să și suprascrie în registrul destinație, tot 2! Tocmai de aceea apare nevoia folosirii registrului auxiliar $7.