

# Implementarea unor Algoritmi de Verificare Folosind Limbaj de Asamblare MIPS

## Introducere

Pentru acest proiect, au fost implementate:

- Un algoritm de Cyclic Redundancy Check pe 16b.
- Un algoritm de sumă de control pentru şiruri de cifre zecimale.

Algoritmii şi implementările lor în limbajul C sunt descrise în [1].

Pentru înţelegerea maşinii MIPS şi a setului de instrucţiuni s-au folosit [2] şi [3].

Mediul de simulare folosit pentru maşina MIPS este [MARS](#). Pentru diverse informaţii despre apeluri sistem şi macro-uri în MARS s-au folosit [4] şi [5].

## Implementare

### Subrutinele implementate

Subrutinele şi semnăturile lor sunt în cea mai mare parte conforme cu cele din implementările din [1].

### Algoritm CRC

Generarea codului CRC de 16 biţi se face cu ajutorul subrutinei `_icrc`. Ca argumente, vom avea:

- **INPUT** - a0 - crc - valoarea de return a unui apel `_icrc()` anterior
- **INPUT** - în stivă - bufptr - adresa şirului de octeţi pentru care calculăm CRC-ul
- **INPUT** - a1 - len - lungimea şirului de octeţi
- **INPUT** - a2 - jinit -  $\geq 0$  - CRC-ul se iniţializează cu fiecare octet pe valoarea jinit;  $< 0$  - CRC-ul se iniţializează cu valoarea din argumentul `crc`.
- **INPUT** - a3 - jrev -  $< 0$  - caracterele prelucrate şi codul final vor fi inversate din punct de vedere al biţilor.
- **OUTPUT** - v1 - va conţine valoarea returnată de subrutină apelantului, CRC-ul.

Reprezentarea zecimală a configuraţiei binare asociate polinomului generator folosit în acest algoritm este 4129.

Pentru accelerarea operaţiei, algoritmul iterează şirul la nivel de octet, nu de bit, folosind o tabelă care conţine valori precalculate pentru „zaparea” din procedeul CRC. Tabela se generează doar la primul apel al lui `_icrc`, astfel încât apelurile succesive sunt mai rapide.

Se foloseşte şi o tabelă statică, hardcodată, pentru inversarea nibbles-ilor.

Generarea valorilor din tabela precalculată se face folosind subrutina auxiliar `_icrc1`. Aceasta va fi apelată de 256 de ori, pentru a genera efectul de zapare al fiecărui caracter ASCII posibil. Ca argumente, i se vor pasa:

- **INPUT** - a0 – crc - caracterul pentru care precalculez efectul „zapării”.
- **OUTPUT** - v1 - va conține valoarea returnată de subrutină, care va fi stocată în tabela de valori precalculate.

Față de varianta din carte, a fost eliminat argumentul **onech** al acestei subrutine, deoarece nu era necesar.

#### Algoritm sumă de control pentru șiruri de cifre zecimale

Generarea sumei de control pentru șiruri de cifre zecimale se face cu ajutorul subrutinei **\_decchk**. Ca argumente, i se vor pasa:

- **INPUT** a0 – string - adresa șirului de cifre pentru care calculez suma de control
- **INPUT** a1 – len - lungimea șirului
- **OUTPUT** v0 – ch - cifra rezultat al sumei de control
- **OUTPUT** v1 - valoare binară (0, 1), valabilă dacă șirul din a0 are pe ultima poziție deja calculată valoarea sumei de control. În acest caz, conținutul lui v1 va fi 0 dacă suma de control este invalidă pentru șirul dat, și 1 altfel. Se poate folosi pentru verificarea integrității șirurilor.

#### Probleme întâmpinate

- Inițial, am încercat implementarea în limbaj de asamblare RISC, dar din cauza capacității reduse de memorie a simulatorului RISC, conținutul tabelor precalculate pentru CRC nu putea fi generat și conținut. Pentru rezolvare, am trecut pe MIPS, care suportă adrese și date pe 32 de biți.

#### Decizii de implementare

- Pentru ușurarea scrierii codului în MIPS, m-am folosit de pseudo instrucțiuni precum **move**, **bgt** („branch if greater than”), **blt** („branch if less than”). [2]
- Pentru facilitarea scrierii codului repetitiv, am folosit capacitatea simulatorului MARS de a prelucra MACRO-uri [4], pentru:
  - Operații utilitare care folosesc apeluri sistem – **print\_char**, **print\_string**, **print\_int** etc.

```
.macro print_hex (%reg)
push_word ($a0)
move $a0, %reg
li $v0, 34
syscall
pop_word ($a0)
.end_macro

.macro print_string (%addr)
push_word ($a0)
la $a0, %addr
li $v0, 4
syscall
pop_word ($a0)
.end_macro
```

- Operații pe biți mai complexe, repetitive în contextul CRC – **lobyte**, **hibyte** etc.

```
# pastreaza doar al doilea octet al valorii dintr-un registru dat ca parametru
# intentionat pentru valori half-word(16b)
.macro lobyte (%reg)
andi %reg, %reg, 0x000000FF
.end_macro

# pastreaza doar al doilea octet al valorii dintr-un registru dat ca parametru
# intentionat pentru valori half-word(16b)
.macro hibyte (%reg)
srl %reg, %reg, 8
andi %reg, %reg, 0x000000FF
.end_macro
```

- Operații de **push** și **pop** pentru lucrul cu stiva

```
# pusheaza un word in stiva din registrul corespunzator
# si modifica stack pointerul in mod corespunzator
.macro push_word (%reg)
addi $sp, $sp, -4
sw %reg, ($sp)
.end_macro

# pop-uie un word din stiva in registrul specificat
# si modifica stack pointerul in mod corespunzator
.macro pop_word (%reg)
lw %reg, ($sp)
addi $sp, $sp, 4
.end_macro
```

- Pentru utilitățile de afișare și de ieșire de program m-am folosit de apeluri sistem. Aceste utilități au ajutat la debug și la validarea funcționării algoritmilor.
- Pentru transmiterea parametrilor și preluarea rezultatelor, am încercat să folosesc cât mai mult convențiile pentru folosirea registrelor, după cum sunt descrise în figura următoare din [2]:

0	zero	constant 0
1	at	reserved for assembler
2	v0	results from callee
3	v1	returned to caller
4	a0	arguments to callee
5	a1	from caller: caller saves
6	a2	
7	a3	
8	t0	temporary
...		
15	t7	
16	s0	callee saves
...		
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return Address
		caller saves

- În majoritatea operaţiilor din subrutine am folosit registrele t0-t9.
- Pentru trimiterea de parametri, am folosit stiva şi a0-a3
- Valorile returnate au fost trimise prin v0 şi v1 (chiar şi din macro-uri).
- Valorile refolosite au fost salvate în s0-s7.

## Exemple de funcţionare

### Algoritm CRC

#### Exemplul 1

- Date de intrare: „sunt smecheer”
- Valoarea returnată de programul C:

```
sir = sunt smecheer
crc = 80db
```

- Valoare returnată de programul scris în limbaj de asamblare

```
0x000080db
-- program is finished running --
```

#### Exemplul 2

- Date de intrare: „Piatra crapa capul caprei in patru, cum a crapat si capra piatra in patru.”
- Valoarea returnată de programul C:

```
sir = Piatra crapa capul caprei in patru, cum a crapat si capra piatra in patru.
crc = afef
```

- Valoarea returnată de programul scris în limbaj de asamblare:

```
0x0000afef
-- program is finished running --
```

### Algoritm sumă de control pentru şiruri de cifre zecimale

#### Exemplu 1

- Date de intrare: „9714558”
- Valoarea returnată de programul C:

```
sir 9714558
9 57: cifra! indecsi ((9, 0), (0, 9)) 9
7 55: cifra! indecsi ((7, 1), (9, 0)) 9
1 49: cifra! indecsi ((1, 2), (9, 8)) 1
4 52: cifra! indecsi ((4, 3), (1, 0)) 1
5 53: cifra! indecsi ((5, 4), (1, 2)) 3
5 53: cifra! indecsi ((5, 5), (3, 7)) 5
8 56: cifra! indecsi ((8, 6), (5, 1)) 9
checksum: 4
```

- Valoare returnată de programul scris în limbaj de asamblare

(\*) pentru amândouă pozele, doar ultimele rânduri sunt de interes, restul sunt indecși afișați la etapele intermediare ale algoritmului, pentru verificare

### Exemplu 2

- Date de intrare: „192355641”
- Valoarea returnată de programul C:

```

sir 192355641
1 49: cifra! indecsi ((1, 0), (0, 1)) 1
9 57: cifra! indecsi ((9, 1), (1, 4)) 0
2 50: cifra! indecsi ((2, 2), (0, 0)) 0
3 51: cifra! indecsi ((3, 3), (0, 6)) 6
5 53: cifra! indecsi ((5, 4), (6, 2)) 9
5 53: cifra! indecsi ((5, 5), (9, 7)) 2
6 54: cifra! indecsi ((6, 6), (2, 6)) 8
4 52: cifra! indecsi ((4, 7), (8, 9)) 4
1 49: cifra! indecsi ((1, 0), (4, 1)) 0
checksum: 7
-- program is finished running --

```

- Valoare returnată de programul scris în limbaj de asamblare:

```

1
0
0
6
9
2
8
4
0
7
-- program is finished running --

```

(\*) pentru amândouă pozele, doar ultimele rânduri sunt de interes, restul sunt indecși afișați la etapele intermediare ale algoritmului, pentru verificare.

### Corectitudinea rezultatelor

Amândoi algoritmi au fost testați, și produc aceleași valori ca și algoritmi în C pe care sunt bazați. Codul complet este disponibil la <https://github.com/ionhedes/proiectFIC>.

## Concluzii

### Probleme

Deoarece acești algoritmi nu au fost bazați pe operații matematice complexe, sau în virgulă flotantă, nu au existat probleme în implementare. Singura problemă inițială a fost gestionarea memoriei, care a dus la schimbarea limbajului de asamblare suport pentru proiect din RISC în MIPS.

### Comentarii despre implementări

Implementările propuse urmăresc îndeaproape algoritmi propuși în [1]. Cu o înțelegere mai bună a conceptelor matematice, aş fi putut scrie algoritmi mai eficienți.

### Ce am învățat?

- Ce este o sumă de control și la ce folosește?
- Cum funcționează CRC-ul? Puțin despre matematica din spatele lui.
- ISBN-ul din cărți și numărul de card au ultima cifră o sumă de control.

### Bibliografie

- [1] Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., 1992, *Numerical Recipes in C*, 2nd ed. (New York, Cambridge University Press), §20.3.
- [2] Akoglu, A., 2012, *Learning MIPS and SPIM* (University of Arizona, College of Engineering, Computer Architecture and Design Lecture Notes), [link](#) accesat la 29.11.2021
- [3] Anonim, *MIPS Instruction Reference* (Lund University, Faculty of Engineering, Electrical and Information Technology, Computer Architecture Lecture Notes), [link](#) accesat la 29.11.2021
- [4] Volmar, K.R., *Writing and Using Macros* (Missouri State University), [link](#) accesat la 30.11.2021
- [5] Volmar, K.R., *SYSCALL functions available in MARS* (Missouri State University), [link](#) accesat la 01.12.2021