ΕΡΓΑΣΤΉΡΙΟ ΑΡΧΙΤΕΚΤΟΝΙΚΉ Η/Υ

2Η ΕΡΓΑΣΊΑ ΕΡΓΑΣΤΗΡΊΟΥ

x32 MIPS ASSEMBLY

Ορφέας- Άγγελος Νικολάου

AM: 2792

E-MAIL: int02792@uoi.gr

Άρτα, 2023 / 11 / 11

Table of contents

Άσκηση 1 – MIPS

Περίληψη

Σε αυτή την άσκηση ζητιέται η υλοποίηση ενός κώδικα σε x32 MIPS asm όπου θα εκτυπώνει το ονοματεπώνυμο, ΑΜ, και εξάμηνο ενός μαθητή (το στοιχεία αυτά θα είναι στο .data και θα είναι ήδη αρχικοποιημένα).

Μετά από αυτό, scanf() δύο ακέραιους αριθμούς με κατάλληλα μηνύματα και να κάνει printf() το άθροισμά τους.

Υλοποίηση

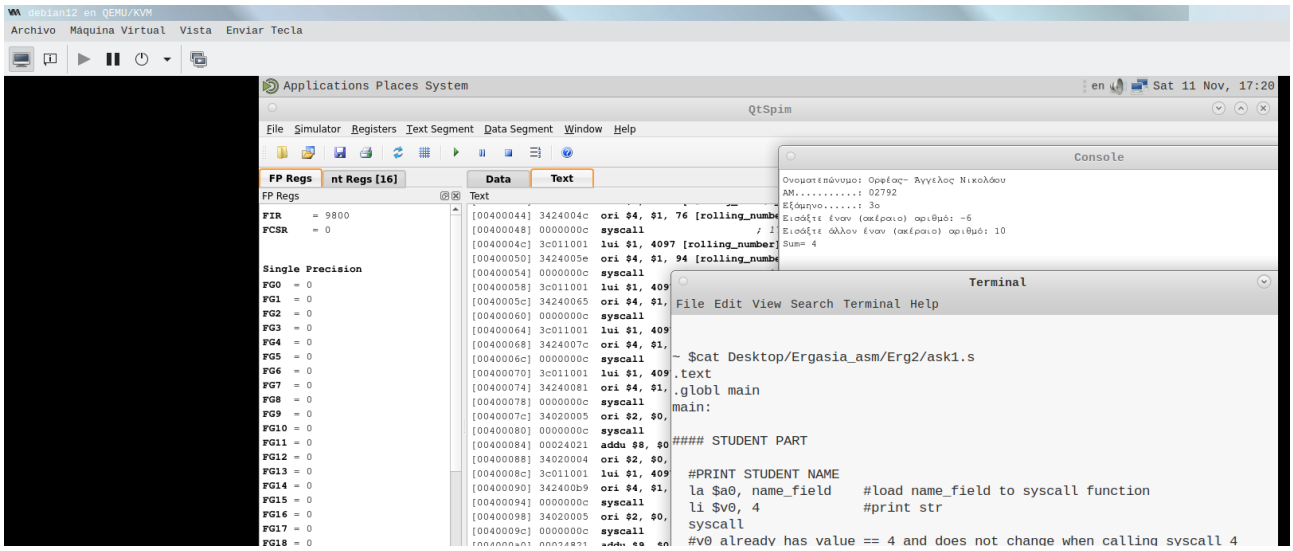
Κώδικας

Ο κώδικας βρίσκεται στα αρχεία της εργασίας στον κατάλληλο φάκελο (Ασκ1), και στο ακόλουθο pastebin και screenshot.

https://pastebin.com/dukwQxqm

```
1   .text
 1 .globl main
 2 main:
 3
 4 #### STUDENT PART
 5
 6   #PRINT STUDENT NAME
 7   la $a0, name_field    #load name_field to syscall function
 8   li $v0, 4             #print str
 9   syscall
10   #v0 already has value == 4 and does not change when calling syscall 4
11   #so no reason to load it again as 4
12   la $a0, name
13   syscall
14   #PRINT STUDENT ROLLING NUMBER
15   la $a0, rolling_number_field
16   syscall
17   la $a0, rolling_number
18   syscall
19   #PRINT STUDENT SEMESTER
20   la $a0, semester_field
21   syscall
22   la $a0, semester
23   syscall
24
25 ####ADD NUMBERS PART
26
27   # x
28   #printf enter x
29   la $a0, enter_x_msg
30   syscall
31   #scanf("%d", &x);
32   li $v0, 5
33   syscall
34   move $t0, $v0 #return value from syscall 5 is $v0
35   #y
36   #printf enter y
37   li $v0, 4
38   la $a0, enter_y_msg
39   syscall
```

```
 22   #printf enter y
 21   li $v0, 4
 20   la $a0, enter_y_msg
 19   syscall
 18   #scanf("%d", &y);
 17   li $v0, 5
 16   syscall
 15   move $t1, $v0
 14   #printf("Sum= ");
 13   li $v0, 4
 12   la $a0, result_msg
 11   syscall
 10   # ήθελα να κάνω
  9   # add $s0, $t0, $t1
  8   # και το s0 --> a0 με lw/sw
  7   # αλλά δεν με άφηνε το interpreter/compiler/whatever
  6   #printf("%d", sum);
  5   li $v0, 1
  4   add $a0, $t0, $t1
  3   syscall
  2   # exit
  1   li $v0, 10
 59   syscall
  1 .data
  2   # για καλύτερο modularity εβαλα 1 για το σταθερο και 1 για το καθε μαθητη
  3   name_field:
  4     .asciiz "Ονοματεπώνυμο: "
  5   name:
  6     .asciiz "Ορφέας- Άγγελος Νικολάου\n"
  7   rolling_number_field:
  8     .asciiz "AM..........: "
  9   rolling_number:
 10     .asciiz "02792\n"
 11   semester_field:
 12     .asciiz "Εξάμηνο.....: "
 13   semester:
 14     .asciiz "3ο\n"
 15   # printf sum part
 16   enter_x_msg:
 17     .asciiz "Εισάξτε έναν (ακέραιο) αριθμό: "
 18   enter_y_msg:
 19     .asciiz "Εισάξτε άλλον έναν (ακέραιο) αριθμό: "
 20   result_msg:
 21     .asciiz "Sum= "
~
```



Το qtSPIM μόνο έχει .deb πακέτο, άρα έπρεπε να το βάλω σε VM debian,

Και δεν ήθελα να το κάνω compile.

Λειτουργία του κώδικα

Στο .data έθεσα τους αλφαριθμητικούς σε κάθε tag του. Χρησιμοποιώ asciiz για να έχει το ‘\0’ στο τέλος (δοκίμασα με ascii και βάλω εγώ το ‘\ 0’, δυστυχώς δεν δούλεψε).

```
7   la $a0, name_field
8   li $v0, 4
9   syscall
```

Θέτω σε parameter για syscall το name_field (από το .data), και λέω ότι θέλω να καλέσω την ‘4’ (ισοδυναμεί με puts() στην C);

Μέχρι Line30 δεν χρειάζεται να εξηγήσω κάτι αφού είναι η ίδια διαδικασία, απλά με διαφορετικά tags.

```
31  #scanf("%d", &x);
32  li $v0, 5
33  syscall
34  move $t0, $v0 #return value from syscall 5 is $v0
```

Syscall 5 είναι scanf () int, και κάνει return στο $v0, άρα το θέτω στο $t0.

```
4   #printf enter y
3   li $v0, 4
2   la $a0, enter_y_msg
1   syscall
    #scanf("%d", &y);
1   li $v0, 5
2   syscall
3   move $t1, $v0
```

Ίδια λογική με τον δεύτερο ακέραιο.

```
3   #printf("Sum= ");
2   li $v0, 4
1   la $a0, result_msg
    syscall
1   # ήθελα να κάνω
```

puts (“Sum= “) διότι δεν υπάρχουν variadic functions όπως την printf() στην asm ώστε να μπορώ να κάνω printf (“Sum= %d”, sum (k, y)).

```
6   li $v0, 1
7   add $a0, $t0, $t1
8   syscall
```

Θέτω $a0 (για παράμετρο για syscall) το άθροισμα μεταξύ τον δύο αριθμών που διαβάστηκαν.

Καλώ syscall 1 (printf (“%d”, some_int)).

```
7   # exit
8   li $v0, 10
9   syscall
10 .data
```

Syscall 10 είναι για exit το πρόγραμμα (αλλιώς συνεχίζει και διαβάζει διευθύνσεις μνήμης σαν να ήταν operations και δεν γίνεται implicitly όπως στην C ή C++).

2η Άσκηση – MIPS (μέρος 1ο)

Περίληψη

Ζητιέται η μετατροπή από κώδικα C σε κώδικα x32 MIPS assembly.

Υλοποίηση

Το κάθε “block” εντολών στην δίπλα εικόνα αντιστοιχεί σε μία εντολή σε κώδικα της C.

Έστω ότι υπάρχει ο καταχωρητής $s8 και το register είναι 26.

```
47 // CONVERT C CODE TO x32 MIPS asm
46
45 /* C code */
44
43 όλα είναι i32
42
41 x = 0;
40 a = 7 + B[16];
39 c = a + x;
38 d = 10 + 8;
37 f = e + a;
36 B[16] = c - 5;
35 B[8] = d + a;
34 B[12] = a - A[4];
33
32 /* Διευκρινησεις */
31
30 a...f == $s0...$s5
29 A = $s6, B = $s7
28
27 Έστω ότι υπάρχει ο καταχωρητής $s8, και η θέση του είναι 26
26 x = $s8
25
24 /* ASM code */
23
22 # x  = 0;
21 addi $s8, $zero, 0 #this is so ugly alla den eimai sigouros an gia lw $s8, $zero prepei na balw dieu8hnhsh sto $zero
20
19 # a = 7 + B[16];
18 lw $t0, 64($s7)  #temp0 = B[16];
17 addi $s0, t0, 7  #a = temp0 + 7
16
15 # c = a + x
14 add $s2, $s0, $s8
13
12 # d = 10 + 8
11 addi $s3, $zero, 10 # d = 0 + 10   mporw kai li alla den kseroume ta opcodes
10 addi $s3, $s3, 8    # d += 8
 9
 8 # f = e + a;
 7 add $s5, $s4, $s0
 6
 5 # B[16] = c - 5
 4 addi $t0, $s2, -5  #temp0 = c - 5
 3 sw $t0, 64($s7)   #B[16] = temp0
 2
 1 # B[8] = d + a;
48  add $t0, $s3, $s0 #temp = d + a
 1 sw $t0, 32($s7)   #B[8] = temp0
 2
 3 # B[12] = a - A[4];
 4 lw $t0, 16($s6)    #temp0 = A[4];
 5 sub $t1, $s0, $t0  #temp1 = a - temp0;
 6 sw $t1, 48($s7)    #B[12] = temp1
```

2η Άσκηση – MIPS (μέρος 2ο)

Περίληψη

Θέσαμε πριν στον καταχωρητή $s8 το register 26.

X32 MIPS asm → Machine code (base 10) → Περιγραφή πίνακα →

→ Machine code (base 2).

Υλοποίηση

Κώδικας μηχανής δεκαδικό

Στις σειρές με Format R το type είναι κόκκινο διότι δεν υπάρχει λόγος να συμπληρωθεί.

		6 bytes	5 bytes	5 bytes	5 bytes	5 bytes	6 bytes	
N. of Instruction	Format	OP	Rs	Rt	Rd	Shamt	Funct	Type
1	I	8	0	26		0		const
2	I	35	8	23		64		adress
3	I	8	8	16		7		const
4	R	0	16	26	18	0	32	
5	I	8	0	19		10		const
6	I	8	19	19		8		const
7	R	0	20	16	21	0	32	
8	I	8	18	8		-5		const
9	I	43	8	23		64		adress
10	R	0	19	16	8	0	32	
11	I	43	8	23		32		adress
12	I	35	22	8		16		adress
13	R	0	16	8	9	0	34	
14	I	43	9	23		48		adress

Περιγραφή πίνακα

Για συντόμευση, εξηγώ το I και R από τώρα:

- add, sub πάντα έχουν format R.

- οι άλλες εντολές που ξέρουμε (εντός της εμβέλειας της εργασίας) είναι format I.

Οι αριθμοί στη ακόλουθη λίστα αντιστοιχούν στο N. of Instruction.

1. Rt είναι το destination για τις εντολές μορφής I, και αφού έχουμε addi, τότε είναι ο καταχωρητής $s8, ο οποίος δεν υπάρχει και για την εργασία θέσαμε τον αριθμό 26. Η σταθερά είναι 0 και το Rs 0 επειδή και τα δύο είναι $zero ή ‘0’.
2. Κάνω lw άρα το lhs είναι source και το rhs είναι destination, άρα ουσιαστικά είναι στην σειρά. 64 adress επειδή έχω M[$s7 + 64] (η διεύθυνση στις εντολές lw και sw είναι rhs μόνο).
3. Διεύθυνση προορισμού είναι $s0/16 άρα μπαίνει στο Rt. Ίδια λογική με N. 1.
4. Αφού είναι εντολή τύπου R τότε OP == 0 και το constant/adress διαιρείται σε 3 κομμάτια, Rd, Shamt, Funct. Τα δύο τελευταία είναι προκαθορισμένα από την add. add x, y, z. Rd → x, Rs → y, Rt → z.
5. καταχωρητής προορισμού $s3 άρα το βάζω στο Rt. Rs τον $zero άρα πρακτικά κάνω li $s3, 10.
6. $s3 += 8.
7. Καταχωρητής $s5 είναι 21 και προορισμού, άρα το βάζω στο Rd αφού η εντολή είναι μορφή R. Τα δύο source registers μπαίνουν Rs, Rt αριστερό δεξί αντίστοιχα.
8. Στο $t0 αφαιρώ το $s3 με το -5. Αφού είναι εντολή I, ο προορισμός πάλι είναι Rt (8).
9. Αυτή την φορά έχουμε sw (43) και ευτυχώς για την ψυχολογία μου είναι μια από τις λίγες εντολές που lhs → Rs και rhs → Rt, άρα τους καταχωρητές τους βάζω στην σειρά έτσι όπως είναι. Αφού είναι sw, ο 16bit αριθμός είναι adress.

10. Πάλη add, lhs → Rd, rhs → Rt, middle - → Rs.

11. sw άρα 23/$s7 Rt αφού είναι rhs και 8/$t0 Rs αφού είναι lhs.

12. lw άρα αντίθετα με το sw, Rt → 8/$t0/destination/lhs και Rs → 22/$s6/source/rhs. Adress αυτή την φορά είναι 16 αφού θέλω την 4η θέση.

13. Επιτέλους κάτι διαφορετικό, έχουμε sub. Δουλεύει (στα OP codes) ακριβώς σαν το add (δηλαδή Rd → destination κτλ) αλλά το Funct είναι 34 αντί για 32.

14. sw άρα η διεύθυνση προορισμού είναι αυτή που είναι δεξιά, δηλαδή ο B[12] (48($s7)).

Κώδικάς μηχανής δυαδικό

		6 bytes	5 bytes	5 bytes	5 bytes	5 bytes	6 bytes	
N. of Instruction	Format	OP	Rs	Rt	Rd	Shamt	Funct	Type
1	I	001000	00000	11010	0000000000000000			const
2	I	100011	01000	10111	0000000001000000			adress
3	I	001000	01000	10000	0000000000000111			const
4	R	000000	10000	11010	10010	00000	100000	
5	I	001000	00000	10011	0000000000001010			const
6	I	001000	10011	10011	0000000000001000			const
7	R	000000	10100	10000	10101	00000	100000	
8	I	001000	10010	01000	1111111111111011			const
9	I	101011	01000	10111	0000000001000000			adress
10	R	000000	10011	10000	01000	00000	100000	
11	I	101011	01000	10111	0000000000100000			adress
12	I	100011	10110	01000	0000000000010000			adress
13	R	000000	10000	01000	01001	00000	100010	
14	I	101011	01001	10111	0000000000110000			adress

Στην 8. το const είναι τόσο μεγάλο επειδή είναι το -5. Η αρχιτεκτονική x32 MIPS τα ints έχει ως αρνητικά με συμπλήρωμα του δύο. (Κάνουμε bit flip όλα τα bits και αθροίζουμε μία μονάδα ή από δεξιά προς αριστερά κοιτάμε για έναν άσο, και όταν/άμα τον βρούμε κάνουμε bit flip όλα τα ψηφία στην αριστερή μεριά του). Δεν υπάρχει κάτι άλλο παράξενο, απλά μετατροπή από base10 σε base2.

Παρατηρείται ότι η κάθε γραμμή περιέχει 32bits.

2η Άσκηση – MIPS (μέρος 3ο)

Περίληψη

Απαντήσεις ερωτήσεων.

Υλοποίηση

Ερώτηση 1

Πέρα από τα διαφορετικά OP codes (το οποίο είναι λογικό, αφού είναι διαφορετικές εντολές format I), το lw διαβάζει τιμή από την διεύθυνσή μνήμης και την θέτει στον καταχωρητή (το δεξί βάζει τιμή στο αριστερό, και το δεξί μπορούμε να βάλουμε adress). Το sw διαβάζει από έναν καταχωρητή (πάλη αριστερά) και τον θέτει σε μια διεύθυνση μνήμης (δεξιά) (το δεξί παίρνει τιμή από το αριστερό, και το δεξί μπορούμε να βάλουμε adress).

Ερώτηση 2

Με τον καταχωρητή $zero ή με οποιονδήποτε άλλον που να έχει την ίδια τιμή .

Ερώτηση 3

Με την εντολή addi, και είναι ως εξής:

addi $some_register, $some_register, 10.

Ερώτηση 4

R format: περιέχει τα fields rd, shamt, funct. 5bit, 5bit, 6bit. Για κάθε πράξη το destination register είναι το rd, το shamt ορίζει πόσο left/right bit shift, και funct ως replacement για OP (αφού αυτό πρέπει να είναι 0 για να οριστεί το R format). Ο καταχωρητής πιο αριστερά είναι ο rd (προορισμού).

I format: τα 3 fields που ανέφερα πριν είναι 1 στο I format, άρα είναι και 16 bit. Αναλόγως την πράξη είναι constant ή adress αυτή η τιμή, και το destination register ΠΆΝΤΑ είναι Rt, όμως ανάλογα με την εντολή αυτός μπορεί να είναι ο rhs ή lhs καταχωρητής.

Ερώτηση 5

Εντολές R format:

add, sub, sll, slr, or, and, nor

Εντολές I format:

lw, sw, addi, la, li, ori, andi, not, move

Ερώτηση 6

Ναι, πχ. το $v0 για να ορίσουμε ποια syscall θέλουμε (κάποιες φορές είναι και τιμή return της syscall). $a0...$a3 για ορίσματα syscall. $t0...$t9 για temp (είναι συνήθεια να χρησιμοποιούνται αν και μόνο αν χρειάζονται για μια πράξη).

Υπάρχουν και $f0, $f12 όπου είναι για parameter και return αντίστοιχα για syscalls με float.

Ερώτηση 7

Ένα από τα πλεονεκτήματα είναι ότι το πρόγραμμα μπορεί να τρέξει στην ίδια αρχιτεκτονική (πχ. X86_64/amd64) με λίγες η μπορεί και καμία αλλαγή.

Μαζί με αυτό, το πρόγραμμα είναι πολύ πιο γρήγορο, διότι δεν μέσα από έναν interpreter ή ένα runtime virtual machine.

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

C/C++/Rust → Compiled

Java → Runtime VM

Python/Lua/JS → interpreted

ΠΗΓΉ