Αναφορά Project

Περιγραφή Ζητούμενων

Ο κώδικας που μας ζητήθηκε να υλοποιήσουμε ήταν το παιχνίδι διάσχισης του λαβυρίνθου. Αναλυτικότερα, υλοποιήσαμε δύο λειτουργικότητες. Αρχικά το παιχνίδι διάσχισης του λαβυρίνθου, κατά το οποίο ο χρήστης μετακινεί τον παίκτη με τα πλήκτρα S,D,A,W μέχρι να φτάσει στην έξοδο κι έπειτα η επίλυση του λαβυρίνθου με χρήση αναδρομικής συνάρτησης.

Σε πρώτη φάση, υλοποιήσαμε τον κώδικα σε C κι έπειτα σε Assembly Mips. Σε δεύτερη φάση, εφαρμόσαμε την τεχνική polling και σε τρίτη φάση κάναμε χρήση interrupts.

Περιγραφή της Εκτέλεσης

Πρώτα απ' όλα, ορίζουμε global μεταβλητές και πίνακες, ακολουθώντας τις οδηγίες της εκφώνησης. Στην Assembly, αρχικοποιούμε τον πίνακα σε ascii. και την τελευταία γραμμή σε asciiz, για να διαβάζεται ο τελικός χαρακτήρας.

Κώδικας C:

```
int step = 0;
int W = 21;
int H = 11;
int startX = 1;
int playerPos = 1;
int totalElements = 231;
char temp[100];
char map[232] = "I.IIIIIIIIIIIIIIIIIII"
                "I....I....I.......I.I"
                "III.IIIII.I.I.III.I.I"
                "I.I.....I..I..I.....I"
                "I.I.III.II...II.I.III"
                "I...I...III.I...I...I"
                "IIIII.IIIII.III.III.I"
                "I.............I.I...I"
                "IIIIIIIIIIIIIII.I.III"
                "@.............I..II"
                "IIIIIIIIIIIIIIIIIIII";
```

Κώδικας Assembly:

```
.data
        .align 2
temp: .space 100

message1: .asciiz "Welcome to The aMAZEing Game!\n"
message2: .asciiz "----------Menu----------\n"
message3: .asciiz "Your moves are:\n"
message4: .asciiz "  W to go up. \n"
message5: .asciiz "  S to go down. \n"
message6: .asciiz "  A to go left. \n"
message7: .asciiz "  D to go right. \n"
message8: .asciiz "  E to have the best route. \n"
message9: .asciiz "\nHere is your Labyrinth:\n"
message10: .asciiz "\nPlease enter your move:\t"
message11: .asciiz "\nNot possible move.\n"
theend: .asciiz "\nWinner Winner Chicken Dinner!\n"

map:            .ascii  "I.IIIIIIIIIIIIIIIIIII"
                .ascii  "I....I....I.......I.I"
                .ascii  "III.IIIII.I.I.III.I.I"
                .ascii  "I.I.....I..I..I.....I"
                .ascii  "I.I.III.II...II.I.III"
                .ascii  "I...I...III.I...I...I"
                .ascii  "IIIII.IIIII.III.III.I"
                .ascii  "I.............I.I...I"
                .ascii  "IIIIIIIIIIIIIII.I.III"
                .ascii  "@.............I..II"
                .asciiz "IIIIIIIIIIIIIIIIIIII"
```

- Κύρια συνάρτηση (main)

Στη main, ορίζουμε τις επιλογές του χρήστη με πέντε switch cases. Αρχικοποιούμε πάντα μία μεταβλητή step, που υπολογίζει κάθε φορά πόσο πρέπει να μετακινηθεί ο χρήστης για την επόμενή του κίνηση. Για παράδειγμα, αν ο χρήστης επιλέξει το πλήκτρο 'S' για να πάει κάτω, τότε θέτουμε το step = +W, όπου W είναι η global μεταβλητή που ορίζει τον αριθμό των στηλών. Έπειτα, προσθέτουμε το step στο playerPos, κι έτσι έχουμε το επόμενο βήμα του χρήστη. Σε αυτό το σημείο κάνουμε έλεγχο ώστε να εξασφαλίσουμε ότι το βήμα δεν είναι εκτός ορίων ή πάνω σε τοίχο, κι έπειτα καλούμε όπως θα δούμε παρακάτω την συνάρτηση που υλοποιήσαμε για να κινείται ο χρήστης. Στο τέλος κάθε case εκτυπώνουμε τον λαβύρινθο. Ενδεικτικά, παρατίθεται ο κώδικας σε C και Assembly για το case W.

Κώδικας C:

```
switch (userMove){

case 'W': case 'w':

    step = -W;
    if((playerPos+step) <=0 || map[playerPos+step] == 'I'){
        printf("\nNot possible move...");
    }else{
        playerPos += step;

        if(makePlayerMove(playerPos-step)==1){
            return(0);
        }
        printLabyrinth();
    }
    break;
```

Κώδικας Assembly:

Όπως βλεπουμε, η Assembly είναι πιστή μετατροπή της C.

```
caseW:
        bne $s0, 'W', caseS

        addi $t0, $0, -1
        mul $s5, $s1, $t0  #kanoyme pollaplasiasmo gia na apokthsoyme to $s1 = - $s1 (w=-w) kai
                           # to vasoyme sto step

        add $t1, $s3, $s5  #$t1 = playerPos + step

        la $s4, map
        add $s4, $s4, $t1
        lb $t5, ($s4)

        bgt $t1, $0, other_or1_if
                j if_cond_1

        other_or1_if:
        bne $t5, 'I', else_caseW_if
                j if_cond_1

                if_cond_1:
                addi $v0, $0, 4
                la $a0, message11
                syscall
                j while_loop

        else_caseW_if:
        #add $s3, $s3, $s5
        move $s3, $t1

        sub $a1, $s3, $s5 #opoy $a1 exei to mesa to orisma playerPos-step gia th makeplayermove
        jal makePlayerMove
        move $t0, $v0 #h v0 tha exei thn timh poy theloyme na epistrefei h move

        bne $t0, 1, go_outside
        jal end_main

        go_outside:
        li $t2, 46
        la $s4, map
        add $s4, $s4, $a1
        sb $t2, ($s4)

        move $a3, $s3
        jal printLabyrinth
```

Αντίστοιχα, παραθέτουμε τον κώδικα για το case E, όπου καλούμε την αναδρομική συνάρτηση.

Κώδικας C:

```
case 'E': case 'e':

    makeMove(startX);
    printLabyrinth();
    break;
```

Κώδικας Assembly:

```
caseE:
        bne $s0, 'E', while_loop

        addi $a1, $0, 1 # $a1 to starX poy exoyme pei sth c oti einai iso me 1 arxika
        #exoyme to orisma na pernaei sto $a1
        jal makeMove

        move $a3, $s3
        jal printLabyrinth
```

- printLabyrinth()

Στη C, η printLabyrinth είναι ίδια με την εκφώνηση. Η μετατροπή της στην Assembly είναι ένα-προς-ένα με τη C. Επειδή, όμως, καλούμε τη usleep (αναλύεται αργότερα), είναι απαραίτητη η χρήση της στοίβας, διότι είναι αναγκαίο να έχουμε αποθηκεύσει σε αυτήν την τιμή επιστροφής, έτσι ώστε να γυρίζει στο σωστό σημείο του κώδικα από το οποίο καλέστηκε.

- makePlayerMove()

Με αυτή τη συνάρτηση κινούμε τον παίκτη. Εάν η επόμενη κίνηση του παίκτη είναι η έξοδος, τότε κάνουμε τις απαραίτητες αλλαγές στον λαβύρινθο, έτσι ώστε να εμφανιστεί '%' στην έξοδο, 'P' στην θέση πριν από την έξοδο και '.' στη θέση πριν από το P.

- makemove()

Πρόκειται για την αναδρομική συνάρτηση που επιλύει τον λαβύρινθο. Στην C, χρησιμοποιήσαμε τον κώδικα της εκφώνησης, με έναν επιπλέον έλεγχο (map[index]=='P'). Το πράγμα που αλλάξαμε μόνο στον κώδικα που μας δόθηκε ήταν να αφαιρέσουμε την εντολή που καλούσε την printLabyrinth() η οποία μέσα στην συνάρτηση αυτή εμφάνιζε κάθε αλλαγή που γινόταν στον πίνακα μέσα στην αναδρομή. Στην Assembly ο κώδικας είναι ένα-προς-ένα με τη C.

- usleep()

Την usleep την χρησιμοποιήσαμε για χρονική καθυστέρηση. Την υλοποιήσαμε με δύο for loops και ως όρισμα εισάγουμε ένα μεγάλο αριθμό (στην περίπτωσή μας 200).

Κώδικας C:

```
void usleep(int index){

 for(int i=0;i<index;i++){
    for(int j=0;j<index;j++){
    }
 }
}
```

Κώδικας Assembly:

```
usleep:
        move $t1, $a0
        addi $t2, $0, 0  # i

        first_sleep_for:
                bge $t2, $t1, end_sleep

                addi $t3, $0, 0  # j
                second_sleep_for:
                        bge $t3, $t1, end_for_sleep
                        addi $t3, $t3, 1
                        j second_sleep_for

                end_for_sleep:
                        addi $t2, $t2, 1
                        j first_sleep_for

        end_sleep:
        jr $ra
```

Polling

Για τη δεύτερη φάση της υλοποίησης του εργαστηρίου, όπως υπόθηκε και παραπάνω, μας ζητήθηκε η χρήση της τεχνικής Polling για να διαβαστεί ο χαρακτήρας, δηλαδή η κίνηση στα πλαίσια της άσκησης, που πληκτρολογεί ο χρήστης. Για την εφαρμογή αυτής της τεχνικής, αρχικά έπρεπε να ενεργοποιήσουμε στο περιβάλλον που γράφουμε σε κώδικα Assembly, δηλαδή το QtSpim, την ρύθμιση Mapped I/O για να μπορεί να δεχθεί το πρόγραμμα τη νέα μας υλοποίηση. Ουσιαστικά, με την χρήση της τεχνικής Polling αντικαταστήσαμε την εντολή syscall, με την οποία θα παίρναμε από το πληκτρολόγιο την επιλογή του χρήστη, με ένα άλλο τύπο κώδικα με τον οποίο επιτυγχάνεται καλύτερη επικοινωνία του επεξεργαστή με τις περιφερειακές συσκευές εισόδου/ εξόδου. Όσον

αφορά τον κώδικα που χρησιμοποιήσαμε, όπως ανέφερε στην εκφώνηση, δημιουργήσαμε αρχικά μία νέα συνάρτηση που την ονομάσαμε read_ch στην οποία θα χρησιμοποιούσαμε τους καταχωρητές receiver control και receiver data για να λάβουμε τον χαρακτήρα που ήθελε ο χρήστης κάθε φορά. Συγκεκριμένα κάθε φορά που στην main καλούταν η συνάρτηση αυτή, δημιουργόταν ένας επαναληπτικός βρόχος στον οποίο γινόταν έλεγχος για το αν η τιμή του receiver control ήταν μηδεν. Αν ίσχυει αυτό τότε η επανάληψη θα συνέχιζε, που θα σήμαινε ότι ακόμη δεν έχει υπάρξει επικοινωνία με το πληκτρολόγιο, δηλαδή ο χρήστης δεν έχει εισάγει ακόμη την τιμή που θέλει. Όταν ο χρήστης θα πληκτρολογουσα ουσιαστικά θα τελείωνε η επανάληψη και θα πήγαινε στις υπόλοιπες γραμμές του κώδικα που χρησιμοποιείται ο καταχωρητής receiver data, στον οποίο έχει αποθηκευτεί η τιμή που επιθυμούμε. Παρακάτω δίνεται το κομμάτι του κώδικα που αναλύσαμε:

```
read_ch:
        la $t1, 0xffff0000 #receiver control mono giati ayta kanoyn read
        lw      $t0, 0($t1)
        andi    $t0, $t0, 1
        beq     $t0, $0, read_ch

        la $t2, 0xffff0004 #receiver data
        lw $t3, 0($t2)
        move $v0, $t3
        jr $ra
```

Interrupts

Για τη τρίτη φάση του εργαστηρίου μας ζητήθηκε η χρήση της τεχνικής των interrupts αντί αυτή του polling για να διαβαστεί ένας χαρακτήρας από το πληκτρολόγιο. Για την εφαρμογή τους, αρχικά έπρεπε να κάνουμε ορισμένες ρυθμίσεις στο simulator του QtSpim. Συγκεκριμένα, έπρεπε να ενεργοποιήσουμε την ρύθμιση Mapped I/O και να αλλάξουμε τον κώδικα του interrupt handler για να βάλουμε ένα συγκεκριμένο κομμάτι κώδικα που ήταν απαραίτητο για την σωστή λειτουργία του προγράμματος.

```
# Interrupt-specific code goes here!
        la $a0, cflag
        addi $v0, $0, 1
        sw $v0, 0($a0)

        la $a0, 0xffff0004  #receiver data
        lb $v0, 0($a0)

        la $a0, cdata
        sb $v0, 0($a0)
# Don't skip instruction at EPC since it has not executed.
```

Όπως φαίνεται στην εικόνα παραπάνω, καλούμε τις global μεταβλητές cflag, cdata που ανέφερε η εκφώνηση της εργαστηριακής άσκησης και εκχωρούμε στην πρώτη την τιμή 1 ενώ στην δεύτερη εκχωρούμε την τιμή που παίρνουμε από τον receiver data. Όσον αφορά τον κώδικα της main στην assembly, αρχικά ενεργοποιήσαμε με κατάλληλη εντολή όλα τα interrupts και στη συνέχεια θέταμε την τιμή του cflag μηδενική. Μέσα στη

```
################## Interrupt ####################
mfc0 $a0, $12
ori $a0, $a0, 0xff11
mtc0 $a0, $12

li $t0, 0xffff0000
#lw $t1, 0($t0)
ori $a0, $0, 2
sw $a0, 0($t0)

la $a0, cflag
addi $t0, $0, 0
sw $t0, ($a0)

loop_label:
        la $t1, cflag
        lw      $t0, 0($t1)
        beq     $t0, $0, loop_label

        la $t2, cdata
        lw $t3, 0($t2)

        la $t1, cflag
        addi $t0, $0, 0
        sw $t0, ($t1)

#########################################
move $s0, $t3
```

δομή επανάληψης ελέγχω συνεχώς την τιμή του cflag. Αν είναι μηδενική η τιμή του δεν βγαίνει από την επανάληψη. Στην περίπτωση που είναι ένα παίρνουμε την τιμή cdata την οποία χρησιμοποιούμε πλέον ως το $s0, userMove, και ξαναμηδενίζουμε το cflag.

```
        ################## Interrupt ####################
        la $t1, cflag
        addi $t0, $0, 0
        sw $t0, ($t1)

loop_label:
        li $a0, 0x801
        mtc0 $a0, $12

        la $t0, 0xffff0000
        li $a0, 0x2
        sw $a0, 0($t0)

                la $t1, cflag
                lw      $t0, 0($t1)
                beq     $t0, $0, loop_label

                la $t2, cdata
                lw $t3, 0($t2)

                la $t1, cflag
                addi $t0, $0, 0
                sw $t0, ($t1)

        ########################################
        move $s0, $t3
```

*****Ο κώδικας παραπάνω είναι αυτός που αλλάξαμε μετά το εργαστήριο για να δούμε αν μπορέσουμε να κάνουμε τον κώδικα των interrupts να δουλέψει. Οπότε κάναμε κάποιες μετατροπές στο κομμάτι που δείξαμε στο εργαστήριο αλλά και αλλάξαμε στην αρχή του κώδικα τις μεταβλητές cflag, cdata όπου αποθηκεύσαμε την μνήμη με align, κάτι που δεν είχαμε κάνει πριν. ******

```
.align 2
.globl cflag
.globl cdata
cflag: .space 4
cdata: .space 4
```