تمرین تئوری ششم

سیدمحمدرضا جوادی 402105868

ساختار و زبان کامپیوتر، دکتر اسدی

پاییز 1403

فهرست مطالب

سوال1)

الف)

این کد ارقام شماره فرد را باهم جمع کرده و ارقام زوج را نیز جمع می کند. اگر b یک باشد جمع فرد هارا از زوج ها کم کرده و اگر صفر باشد جمع زوج هارا منهای جمع فرد ها می کند.

کد سی:

```c
#include <stdio.h>

int weird_func(int a,int b){
    if (a==0)
        return 0;
    int result = weird_func(a/10, b ^ 1);
    int remainder = a %10;
    if (b == 0)
        return result - remainder;
    return result + remainder;
}

int main(){
    printf("%d\n", weird_func(135,1));
}
```

ب)

در خط base_case: 42 به اشتباه sp را 12 تا بالا برده درصورتی که قبل از آنکه به آنجا برویم اصلا sp تغییر نکرده بود پس به ساختار برنامه آسیب می زند و باید حذف شود. همچنین وقتی به آنجا برویم بعد از آن در ادامه به لیبل exit می رسد و آنجا هم دوباره sp را 12 تا تغییر می دهد پس باید قبل آن یک jr $ra می گذاشت.

سوال2)

الف)

نحوه ذخیره شدن در لیتل اندین به این صورت است که بیت با ارزش بالاتر در آدرس با ارزش بالاتر است، و در بیگ اندین برعکس در جدول های زیر حافظه از چپ به راست (0 به 3) هست و 2 جدول این بخش برای حالت اول هستند.

num_1 که استرینگ است، پس هر کاراکتر آن در یک بایت ذخیره می شود، مقادیر صحیح درواقع کد asci کاراکتر ها هستند:

"4" یا 52	"3" یا همان عدد 51	"2" یا همان عدد 50	"1" یا همان عدد 49

num_2 یک عدد صحیح است، پس عدد 32 بیتی است و 8 بیت 8بیت جدا می شود و بایت با ارزش بالاتر به سمت چپ تر می رود.

0000 0000 0000 0000 0000010011010010

210	4	0	0

حال به توجه به مطالب بالا و کد می توان خط به خط توضیح داد چه مقداری در ثبات ریخته می شود:

```
.data
num_1: .asciiz "1234"
num_2: .word 1234
.text
main:
       lw $a0, num_1
```

```
                                خود مقدار رشته "1234" در ثبات ریخته می شود، به نوعی مقدار 825373492 که معادل
                                00110000100110010000110011001101000 است
        lb $a0, num_1
                                0 تا بایت بعد رشته، که طبق جدول می شود 49
        lb $a0, num_1+1
                                1 تا بایت بعد رشته، 50
        lb $a0, num_1+2
                                2تا بایت بعد رشته 51
        lb $a0, num_1+3
                                3 بایت بعد، که می شود 52
        lh $a0, num_1
                                2 بایت اول که 0011000100110010 است پس ثبات 12594می شود
        lh $a0, num_1+1
این مورد به دلیل آدرس دهی غلط ارور خواهد داد، چون مضرب 2 نیست آدرس. ولی با فرض اینکه چنین نوع آدرس دهی داشته باشیم:
                                بایت دوم و سوم که 0011001000110011است یعنی می شود مقدار ثبات 12851
        lh $a0, num_1+2
                                2 بایت آخر که 0011001100110100 است یعنی13108
        lw $a0, num_2
                                1234 را در ثبات می ریزد
        lb $a0, num_2
                                بایت اول که0 می شود.
        lb $a0, num_2+1
                                بایت دوم که می شود 0 باز
        lb $a0, num_2+2
                                در بایت سوم 04 هگزادسیمال یعنی 4 هست که در ثبات می ریزد
        lb $a0, num_2+3
                                D2 هگزادسیمال که مقدار بایت آخر است را در ثبات می ریزد یعنی می شود 210
        lh $a0, num_2
                                2 بایت اول که 0x0000است پس ثبات 0 می شود
        lh $a0, num_2+1
این مورد به دلیل آدرس دهی غلط ارور خواهد داد، چون مضرب 2 نیست آدرس. ولی با فرض اینکه چنین نوع آدرس دهی داشته باشیم:
                                بایت دوم و سوم که 0x0004 است یعنی می شود مقدار ثبات 4
        lh $a0, num_2+2
                                2 بایت آخر که 0x04D2 است یعنی 1234.
```

ب)

در دستوراتی که بایت به بایت دریافت می کردند مانند lhو lb ثبات دچار تغییر می شود. چرا که عینا بایتی در حافظه را می خواهیم استخرج کنیم.

سوال3)

کد سی:

```c
#include <stdio.h>

int func2(int n, int m) {
    return 2 * m * (n/2);
}
```

```c
int func1(int n, int m) {
    if (n == 0 || m == 0)
        return 1;
    return n + func1(n--, --m) + func2(n, m--);
}

int main() {
    int n = 5;
    int m = 6;
    func1(n, m);
    return 0;
}
```

کد اسمبلی متناظر:

```asm
.text
.globl main
main:
    li      $a0, 5      # $a0 = 5
        li      $a1, 6      # $a1 = 6
    jal func1
        li      $v0, 10     # $v0 = 10
        syscall


func1:
    bne $a0,$0,case0
    li $v0,1
    jr $ra
case0:
    bne $a1,$0,case1
    li $v0,1
    jr $ra
case1:
    subi $sp,$sp,16
    sw $a0, 0($sp) #n
    subi $a1,$a1,1
    sw $a1, 4($sp) #--m
    sw $ra, 8($sp)

    jal func1
    sw $v0, 12($sp)

    lw $a0, 0($sp) #n
    subi $a0,$a0,1 # n--

    lw $a1, 4($sp) #m -1

    jal func2
```

```
    lw $a1, 4($sp) #m -1

    lw $a0, 0($sp) #n
    add $v0,$v0,$a0
    lw $t0, 12($sp)
    add $v0,$v0,$t0

    subi $a1,$a1,1 # m--

    lw $ra, 8($sp)
    addi $sp,$sp,16
    jr $ra

func2:
    sll $t1,$a1,1 # 2 * m
    sra $t0,$a0,1 # n/2
    mul $v0, $t1,$t0# 2 * m * (n/2)
    jr $ra
```

ماشین کد(همگی هگزادسیمال هستند):

24040005

24050006

0c100005

2402000a

0000000c

14800002

24020001

03e00008

14a00002

24020001

03e00008

20010010

03a1e822

afa40000

20010001

00a12822

afa50004

afbf0008

0c100005

afa2000c

8fa40000

20010001

00812022

8fa50004

0c100023

8fa50004

8fa40000

00441020

8fa8000c

00481020

20010001

00a12822

8fbf0008

23bd0010

03e00008

00054840

00044043

71281002

03e00008

سوال4)

کد اسمبلی داده که

```
1 .data
2 .align 2
3 data:
4 .word 0x12345678
5 .half 0x9ABC
```

```
6 .byte 0xDE
7 .byte 0xF0
```

است.

الف)

در صورت عدم استفاده داده ها را طبق رهنمود پیشفرض ذخیره می کند اما پس از تعیین رهنمود 2 آدرس ها فقط مضرب چهار می توانند باشند. در این مورد خاص شاید به نظر برسد تفاوتی حاصل نمی شود ولی اگر قبل از این داده دیگری ذخیره کرده بودیم دیگر ذخیره سازی بایت ها طبق خواسته ما نبود و بهم می ریخت.

ب)

کلمه 0x12345678 خط 4 در 0x1000 تا 0x1003 (4 بایت) نیم کلمه0x9ABC خط 5 در0x1004ت0x1005، بایت 0xde در 0x1006 و 0xf0 در 0x1007

ج)

درصورت عدم موفقیت که پیغام ارور misaligned را بر می گرداند زیرا ما به طور درست آدرس دهی نکردیم، ولی اگر فرض کنیم خطایی نمی داد و 4 بایت بعدی آدرس را در نظر می گرفت و چون میپس بیگ اندین است، کلمه را به شکل0x56789ABC می دید. در جدول زیر آدرس از 0x1002 تا 0x1005 است.

0xBC	0x9A	Ox78	0x56

د)

```
    # a0 contains address
    lb $t1, 0($a0)
    sll $t1,$t1,24

    lb $t2, 1($a0)
    sll $t2,$t2,16
    or $t1,$t2,$t1

    lb $t2, 2($a0)
    sll $t2,$t2,8
    or $t1,$t2,$t1

    lb $t2, 3($a0)
    or $t1,$t2,$t1
```

کد بالا را می توان برای دستور lw که خطا ندهد استفاده کرد. بله این کدی که من نوشتم به شدت وابسته است چون دستور lb خود به نحوه ذخیره سازی بیگ اندین یا لیتل اندین وابسته است(یعنی اگر مقداری را ذخیره کردیم و بعد بایت به بایت برداشت کنیم مهم است)

مثلا در مورد دستوری که در مورد قبل بود، اگر بیگ اندین باشد باشد مشابه قسمت قبل بر می گرداند ولی اگر لیتل اندین باشد جدول بالا به شکل زیر در می آید و اطلاعات برعکس ذخیره می شوند:

0x9A	0xBC	0x12	0x34

ه)

```
1 la $t0, data
                                    مقدار 1000 هگز در ثبات ریخته می شود
2 lh $t1, 4($t0)
  مقدار 1004 و 1005 هگز حافظه یعنی نیم کلمه 0x9abc در ثبات ریخته می شود شود  اما چون دستور signed است مقدار  منفی25924
                                    ذخیره می شود
3 lb $t2, 6($t0)
```

```
بایت 1006 هگز حافظه یعنی 0xde در ثبات ریخته می شود  اما چون دستور signed است مقدار منفی 34 ذخیره می شود
4 lbu $t3, 7($t0)
مقدار 0xf0 ذخیره می شود ( مثبت، دستور آنسایند است)
5 lhu $t4, 4($t0)
مقدار 1004 و 1005 هگز حافظه یعنی نیم کلمه ox9abc در ثبات ریخته می شود شود
```

سوال5)

الف)

```
int fun(int a){
    if (a<2)
        return 1;
    return 2 + fun(a-1) + fun(a-2) + fun(a-3);
}
```

ب)

709

ج)

532 بار

د)

یک راه محاسبه و حل خود رابطه بازگشتی است . ولی مشکل این کار در این است که در n های بزرگ به شدت نادقیق می شویم.

راه پیشنهادی من برنامه نویسی پویا است همچنین فرمول بازگشتی را ساده تر هم می کنیم.

رابطه اولیه:

$$f(n) = 2 + f(n-1) + f(n-2) + f(n-3)$$

- ➔ $f(n) = f(n-1) + 2 + f(n-2) + f(n-3) + f(n-4) - f(n-4)$
- ➔ $f(n) = 2f(n-1) – f(n-4)$

اینگونه هم تعداد کال ها کاهش پیدا می کند هم با استفاده از برنامه نویسی پویا تعداد کال ها کمتر هم می شود(یعنی آرایه ای درست کنیم که نتایج محاسبه در آن ذخیره می شود، هر بار که عضوی را می خواهیم ابتدا بررسی می کند در آن ارایه وجود دارد یا نه یعنی درصورتی که قبل محاسبه شده باشد در O(1) آن را بر می گرداند) طبیعتا چون اینجا رابطه بازگشتی از درجه 4 شد باید در شرط پایه هم تغییراتی دهیم تا 3 یا 4 عضو اول را از قبل داشته باشد و برگرداند.

سوال6)

تابع اول:

```
f1:
        addiu   $sp,$sp,-8
        sw      $fp,4($sp)
        move    $fp,$sp
        sw      $4,8($fp)
        sw      $5,12($fp)
        sw      $6,16($fp)
```

```
        sw      $7,20($fp)
        lw      $3,8($fp)
        lw      $2,12($fp)
        mul     $3,$3,$2
        lw      $2,16($fp)
        mul     $3,$3,$2
        lw      $2,20($fp)
        mul     $3,$3,$2
        lw      $2,24($fp)
        mul     $3,$3,$2
        lw      $2,28($fp)
        mul     $2,$3,$2
        move    $sp,$fp
        lw      $fp,4($sp)
        addiu   $sp,$sp,8
        jr      $31
        nop
```

بهینه سازی ها: برای اینکه از رجیستر های بیشتر استفاده نشود و همینطورمجبور به ذخیره سازی موقت در حافظه نشویم (که بسیار کند است) نتایج ضرب هارا مستقیما در رجیستر های خروجی ذخیره می کند و به ادامه عملیات می پردازد تا سرعت بالا برود و حافظه مورد نیاز کمتر شود. همینطور استفاده اضافی از استک ندارد و فقط طبق قرارداد های میپس استفاده کرد. همچنین برای اینکه خود کد در حافظه بهینه تر ذخیره شود و آدرس دهی سریعتر از nop استفاده کرده.

تابع دوم:

```
f2:
        addiu   $sp,$sp,-16
        sw      $fp,12($sp)
        move    $fp,$sp
        sw      $4,16($fp)
        li      $2,1                        # 0x1
        sw      $2,0($fp)
        sw      $0,4($fp)
.L3:
        lw      $2,4($fp)
        slt     $2,$2,6
        beq     $2,$0,.L2
        nop

        lw      $2,4($fp)
        sll     $2,$2,2
        lw      $3,16($fp)
        addu    $2,$3,$2
        lw      $2,0($2)
        lw      $3,0($fp)
        mul     $2,$3,$2
        sw      $2,0($fp)
        lw      $2,4($fp)
```

```
        addiu   $2,$2,1
        sw      $2,4($fp)
        b       .L3
        nop

.L2:
        lw      $2,0($fp)
        move    $sp,$fp
        lw      $fp,12($sp)
        addiu   $sp,$sp,16
        jr      $31
        nop
```

اجتناب از محاسبات اضافی، برای مثال بجای ضرب در توان های 2 از شیفت استفاده کرده، مانند قسمت قبل از رجیستر های اضافی سعی کرده استفاده نکند و با خود رجیستر های خروجی حدالامکان کار کرده. برای کاهش حجم کد در مواقعی دستورات خود ISA را استفاده کرده نه شبه دستورات (مثلا برای جمع با یک) همینطور دستوراتی که پیچیدگی کمتری داشتند را ترجیح داده (مثلا unsigned ها) در حلقه بجای گذاشتن لیبل ها یا beq های اضافی با کمترین مقایسه و دستورات پرش حلقه را پیاده سازی کرده. همچنین برای اینکه خود کد در حافظه بهینه تر ذخیره شود و آدرس دهی سریعتر از nop استفاده کرده.