تمرین تئوری چهارم

سید محمدرضا جوادی 402105868

ساختار و زبان کامپیوتر، دکتر اسدی

پاییز 1403

سوال اول)

عینا کار حلقه هارا (حتی **i++** را) را پیاده سازی می کنیم:

```
# c code:
# int a[3], b[3], c[3];
# for (int i = 0; i != 3; i++) {
# c[i] = 4*a[i] - 8*b[i];
# }
# for (int i = 1; i != 3; i++) {
# c[i] = 2*c[i-1] + c[i];
# }
# assembly code:
.data
    a: .word 1, 2, 3 # values are not important
    b: .word 1, 2, 3 # values are not important
    c: .word 1, 2, 3 # values are not important
.text
.globl main

main:
    la $s0, a #s0 is a
    la $s1, b #s1 is b
    la $s2, c #s2 is c

    #--------------------implementing first loop without loop
    add $t4, $zero, $zero #counter
    lw $t3, 0($s0)
    add $t0 , $t3, $t3
    add $t0 , $t0, $t0 # 4a

    lw $t3, 0($s1)
    add $t1 , $t3, $t3
    add $t1 , $t1, $t1
    add $t1 , $t1, $t1 # 8b
    sub $t4, $t0, $t1
    sw $t4, 0($s2) #c[0]
    addi $t4, $t4, 1 #i++

    lw $t3, 4($s0)
    add $t0 , $t3, $t3
    add $t0 , $t0, $t0 # 4a

    lw $t3, 4($s1)
    add $t1 , $t3, $t3
    add $t1 , $t1, $t1
    add $t1 , $t1, $t1 # 8b
    sub $t4, $t0, $t1
```

```
sw $t4, 4($s2) #c[1]
addi $t4, $t4, 1 #i++


lw $t3, 8($s0)
add $t0 , $t3, $t3
add $t0 , $t0, $t0 # 4a


lw $t3, 8($s1)
add $t1 , $t3, $t3
add $t1 , $t1, $t1
add $t1 , $t1, $t1 # 8b
sub $t4, $t0, $t1
sw $t4, 8($s2) #c[2]
addi $t4, $t4, 1 #i++


#--------------------implementing second loop without loop
addi $t4, $zero, 1 #counter
lw $t0, 0($s2) #c0
lw $t1, 4($s2) #c1
lw $t2, 8($s2) #c2

add $t3, $t0,$t0
add $t3, $t3, $t1
sw $t3, 4($s2) #c1 done
addi $t4, $t4, 1 #i++

move $t1, $t3 # new value of c1 loaded in t1


add $t3, $t1,$t1
add $t3, $t3, $t2
sw $t3, 8($s2)
addi $t4, $t4, 1 #i++
```

سوال دوم)

الف)

شبه دستورات مجموعه ای از دستورات هستند که در واقع از یک یا چند دستور دیگر تشکیل شده اند(به دلیل افزایش خوانایی یا مفهومی یا زیاد بودن استفاده از چند دستور کنار هم و ...). این ها برخلاف دستورات واقعی در ISA تعریف نشده و کامپیوتر آن را نمی فهمد و هنگامی

ترجمه به کد ماشین توسط اسمبلر به دستورات ریزتر تبدیل می شوند. برای انواع اقسام توضیحات بسیاری از این ها می توانید به لینک زیر سر بزنید ولی ما طبق گفته سوال فقط 2 مثال می زنیم:

https://matthews.sites.truman.edu/files/2019/11/pseudoinstructions.pdf

مثال اول دستور li که ازori (یای با مقدار ثابت) و lui(لود قسمت بیت های بالاتر یعنی 16 بیت مقدار ثابت)ساخته شده و مثال دوم دستورmove که از add (جمع بدون علامت) تشکیل شده اند.

نحوه نوشتن دستور و مثالی از کاربرد و همینطور ترجمه به ماشین را در جدول زیر می بینید:

نحوه نوشتن سینتکس دستور	چیزی که در ترجمه به آن تبدیل می شود
Li $reg, Number	Lui $reg, upperPartOfNumber Ori $reg, lowerPartOfNumber
Move $r1, $r2	Add $r1, $r2, $zero

ب)

مفید بودن آن ها بسیار مشهود است، بسیاری از وقت ها برای یک کار خاص که در ISA تعریف نشده اند ولی بسیار پرکاربردند ما مجبوریم از ساختار های نسبتا پیچیده تر استفاده کنیم که هم زحمت مارا زیاد می کند و هم ناخواناست و هم ممکن است منجر به اشتباه شود. به طور مثال بسیاری از مواقع مقدار یک رجیستر را در دیگری می خواهیم بریزیم فکر کردن به منطق و ساختار پیچیده دستوری که معادل این کار باشد ممکن است طی دفعات زیاد اذیت کند و عملی تکراری و بیهوده را بار ها انجام دهیم ولی برای این مجموعه دستور های بسیار پرکاربرد و با ساختار پیچیده می توانیم سودو اینستراکشن هارا تعریف کنیم اینطوری هم پیاده سازی آن دستور برای همه یکسان می شود هم خوانایی افزایش می یابد و هم نگران پیچیدگی پشت دستور نیستیم.

از طرفی به دلیل انتزاعی سازی این بخش برنامه نویس تسلط خود بر برنامه را کمی از دست می دهد زیرا بخشی از دستورات به اسمبلر منتقل می شود و شاید مسائلی مانند بهینگی یا امنیت برنامه با چالش مواجه شوند.

ج)

با پیاده سازی یکسان و با بهینه سازی آن تمام برنامه نویس های استفاده کننده از آن برنامه را به یک شکل و بهترین حالت استفاده می کنند و هنگام استفاده از کد دیگران مانند پروژه های گروهی کد در مجموع بهتر عمل می کند.

از آنجا که پیاده سازی این دستورات از نظر کاربر (برنامه نویس) خارج است، ممکن است به بهینه ترین حالت پیاده سازی نشده باشد یا به تعداد دستورات خیلی بیشتری تبدیل شوند که برای ما محدودیت ایجاد می کند و کد در کل سنگین تر و کندتر می شود.

د)

همانطور که اشاره شد با انتزاعی سازی پیاده سازی دستورات ما دیگر به جزییات پیاده سازی کد دسترسی نداریم و نمی دانیم اسمبلر در پشت پرده چکار می کند. با این شرایط عدم درک برنامه نویس از عملکرد و پیاده سازی شبه دستور می تواند منجر به اشتباهات ناخواسته ای شود. در آن سو از آنجا که از پیاده سازی آن ها بی اطلاعیم این عدم آگاهی می تواند حساسیت های امنیتی را ایجاد کند.

ه)

این در مثال های بخش اول هم بود. این دستور مقدار موجود در یک ثبات را به ثبات دیگر منتقل می کند. مثلا رجیستر دارای مقداری را $register1 و رجیستری که می خواهیم در آن مقدار را بریزیم $register2 فرض کنید در این صورت:

```
move $register2, $register1   دستور مجازی، چیزی که ما بینیم←
```

```
add $register2, $register1, $zero  دستور واقعی، چیزی که دستور مجازی به آن تبدیل می شود.
```

که می دانیم $zero رجیستری است که به صورت پیشفرض مقدار ثابت 0 دارد.

و)

فرض کنید عدد ما 0x12345678 و رجیستری که می خواهیم در آن بریزیم $regاست. در این صورت:

```
li $reg, 0x12345678  دستور مجازی که ما از آن استفاده می کنیم

دستوراتی که دستور مجازی ما به آن ها تبدیل می شوند:

lui $reg, 0x1234
ori $reg, $reg, 0x5678
```

سوال سوم)

0x00842026

0000 0000 1000 0100 0010 0000 0010 0110

این دستور opcode صفر دارد پس rفرمت است و 6 بیت آخر فانک هست، 100110 برای xor است. همچنین هر 3 رجیستر 00100 رجیستر چهارم یعنی a0 بعد این دستور a0 با خودش xor می شود یعنی مقدارش به صفر تغییر می کند.

Xor $a0, $a0, $a0

0x34900001

0011 0100 1001 0000 0000 0000 0000 0001

Opcode برابر 13 یعنی دستور ori و iفرمت است. در 16 بیت مقدار ثابت 1 است رجیستر 16 ام s0 است. این دستور a0 که صفر است را با یک or می کند و مقدار s0 یک می شود

Ori $s0, $a0,1

0x001087c0

0000 0000 0001 0000 1000 0111 1100 0000

Opcodeبرابر 0 یعنی دستور آر فرمت است و از مقدار 6 بیت آخر می فهمیم دستور sll است. رجیستر 16 ام s0 است. این دستور 31 تا یک را شیفت می دهد که جواب می شود منفی 2 به توان 31 و این را در s0 می ریزد.

Sll $s0, $s0, 31

0x00108043

0000 0000 0001 0000 1000 0000 0100 0011

Opcodeبرابر 0 یعنی دستور آر فرمت است و از مقدار 6 بیت آخر می فهمیم دستور sra است. چون شیفت محاسباتی علامت را حفظ می کند با یک شیفت به راست s0 منفی 2 به توان 30 می شود.

0x00102782

0000 00000001 0000 0010 0111 1000 0010

Opcodeبرابر ۰ یعنی دستور آر فرمت است و از مقدار ۶ بیت آخر می فهمیم دستور srl است. این می گوید s۰ را که ۰۰۰....۱۱۰۰۰۰ است۳۰ تا شیفت دهیم، یعنی ۱۱ که در بیت ۳۱ و ۳۲ بود به صورت لاجیکال ۳۰تا شیفت خورده به خانه اول و دوم میاید. مقدار نهایی a۰ برابر ۳ می شود

ب)

به صورت کلی در نهایت این کد دو عددs0 و s1 را از هم کم می کند و در a0می ریزد. از دستورات به طور ریاضیاتی اثبات می شود.

کامپایل شده به دودویی (هگزادسیمال نمایش می دهم):

0x02112024

0x02042026

0x02112825

0x00b02826

0x00852022

ج)

Xor می کند. اگر همه دستورات را در قابل دستورات بیتی قرار دهیم ریاضیاتی اثبات هم می شود. شهود دیگر مفهوم ایکسور به عنوان نوعی تفریق است.

.4

هر یک را بر اساس 6 بیت اول به فرمت های خود تقسیم می کنیم و بعد اگر rفرمت بود 3 رجیستر و فانکت را تعیین می کنیم در غیر اینصورت بر اساس داکیومنت ها نگاه می کنیم opcodeمال کدام دستور است. اینجا اکثرا r و Iفرمت بودند و نیاز بود تا rsوrtرا مشخص کنیم و آدرس نداشت

```
xor $s0,$s0,$s0
xor $s0,$s1,$s0
xor $s0, $s2,$s0
xor $s0, $s3,$s0
addi $s0,$s0, 1
addi $s1,$s1,1
add $s1, $s2,$s0
add $s2, $s3,$s1
add $s2,$s4,$s3
add $s4,$s1,$s0
add $s4, $s1, $s0
add $s0,$s2,$s1
```

```
add $s2,$s3,$s1
add $t1, $t2,$t1
```

سوال پنجم)

در بسیاری از مواقع کامپایلر کار هایی که برنامه نویس به دلایلی مانند راحتی نوشتن و خواندن یا ... انجام نداده را خود انجام می دهد. به طور مثال اگر حلقه قرار است ۱۰۰ بار اجرا شود و این همیشه ثابت است خود ۱۰۰ بار محتوای حلقه را می نویسد تا از شر ۱۰۰ بار چک کردن شرط راحت شود و برنامه بیهنه تر شود

در این جا هم حلقه را باز کرده(۳بار کپی پیست) اینطوری یک متغیر که باید increament می کرد را هم حذف کرده، یعنی هم از شر مقایسه راحت شده و بهینه تر کرده کدرا هم متغیر های اضافی را حذف کرده و برنامه حجم کمتری گرفته.

متغیر های غیر ضروری را تعریف نکرده ، آنهایی که هیچ استفاده ای نداشتند و صرفا یکبار تعریف شده.

همینطور توابع را باز کرده و به صورت درخط تابع add را پیاده سازی کرده. اینگونه زحمت اشغال استک یا فرایند کال کردن را نکرده و برنامه سبک تر و ساده تر شده.