تمرین تئوری سوم

سیدمحمدرضا جوادی 402105868

ساختار کامپیوتر ، دکتر اسدی، پاییز 1403

سوال اول)

الف)

در این روش به منظور کوتاه تر کردن طول کد و بهینه شدن سرعت اجرای کد، در هنگام اجرای آن، خود برنامه خودش را تغییر می دهد. البته این تغییر عملکرد (functionality) کد را تغییر نمی دهد. توجه کنید که تغییر رندوم یا تصادفی نیست بلکه تمام از قبل برنامه ریزی شده و عمدی است. گاهی می دانیم اتفاق خاصی مکررا رخ می دهد و می خواهیم سربار اضافی را برداریم و کد مستقیما به پرداختن به کار به روش سریعتری شود. در این هنگام می توانیم از قبل کدرا به طوری تغییر دهیم تا از اعمال اضافه ای بپرهیزد یا کد خود که درحال اجراست را به نوعی overwrite کند. پدر این روش ، پرچم گذاری است که در آن هردفعه براساس رخداد های پس از اجرا باید شرطی را چک می کردیم و مناسب همان کد مربوطه اجرا می شد. اما در کد خوداصلاح این چک شدن های اضافه از میان می روند.یکی از مثال های واقعی کرنل لینکوس است که با توجه به کاری که می کنیم برخی امکانات را فعال و برخی را غیر فعال می کند. مثالی ساده در رابطه با اسمبلی منبع(https://stackoverflow.com/questions/29262391/self-modifying-mips-code) این کد دائما 2 عدد می گیرد و باهم جمع می کند، اما هنگامی که جمعشان 13 بود برنامه تغییر کرده و از حالا بجای جمع، And منطقی استفاده می کند. در حقیقت خط

add $t2, $t0, $t1

به

and $t2, $t0, $t1

تبدیل می شود

```
# MIPS Program to demonstrate self-modifying code

.data
num1:         .word      0
num2:         .word      0
addOut:       .asciiz    "ADD: "
andOut:       .asciiz    "AND: "

.text
main:
sumLoop:
   li $v0, 5      #Call code to read an int
   syscall
   move $t0, $v0      #Moving read int to $t1

   li $v0, 5      #Call code to read an int
   syscall
   move $t1, $v0      #Moving read int to $t2

   add $t2, $t0, $t1   #Adding num1 and num2 together

   la $a0, addOut
   li $v0, 4
   syscall

   move $a0, $t2
   li $v0, 1
   syscall

   beq $t2, 13, instMod    #Calling method to modify add instruction if sum = 13
   bne $t2, 0, sumLoop #If result is not yet 0, ask for new sum

endSumLoop:
   li $v0, 10
   syscall

instMod: #Method to change add instruction to an and instruction
```

```
  lw $t1, instruction_to_replace
  sw $t1, instruction_to_be_replaced
  j sumLoop  # go back to your sumLoop

instruction_to_replace:
  and $t2, $t0, $t1
```

ب)

فرخوان های سیستمی به کاربر این اجازه را می دهد تا بتواند از کارکرد های سیستم استفاده کند. هنگام فراخوانی اجرای برنامه متوقف شده، و اجرای درخواست در کرنل سیستم ادامه می یابد و پس از پایان عمل به ادامه برنامه برمی گردیم با تاخیر کمی.

برای برخی عملیات ها مانند ورودی خروجی، کار با فایل، شبکه، مواقعی که به دسترسی سیستم عامل نیاز داریم و ... توانایی انجام عملیات صرفا با فراخوانی سیستم عامل ممکن است. در این مواقع برای آن کار ها نیاز به عمیات هایی داریم که در سیستم عامل پیاده سازی شده اند.

ج)

منظور مقداری است که در $v0 قرار می گیرد. خروج 10، خواندن 5 برای عدد صحیح و 6 عدد اعشاری (برای کاراکتر و استرینگ هم داریم همینطور انواع دیگر داده ها) و برای پرینت کردن 1 عدد صحیح، 2 اعشاری 3 دابل و 4 رشته است.

	in $v0		
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of	*See note below table*

د)

```
1 .data
2
3 .text
4 .globl main
5
6 main:
7 li $a0, 5
8 li $a1, 8
9
10 jal func
11
12 jal exit
13
14 exit:
15 li $v0, 10
```

```
16 syscall
17
18 go_to_exit:
19 j exit
20
21 func2:
22 la $t6, go_to_exit
23 lw $t4, 0($t6)
24 la $t6, jump_section
25 sw $t4, 0($t6)
26 func:
27 add $a0, $a0, $a1
28 la $t9, func
29 lw $t8, 0($t9)
30 li $t7, 2048
31 xor $t8, $t7, $t8
32 sw $t8, 0($t9)
33 jump_section:
34 j func2
```

خط 1 که بخش دیتا را مشخص می گند ولی خالی است بعد از آن به تکست یا قسمت دستورات می رسیم.

1.در خط 7 و 8 مقادیر 5 و 8 در a0و a1 ذخیره می شوند و در 10 به تابع funcمی رویم. آنجا a0وa1 جمع و در a0می ریزیم پس می شود13. بعد آدرس فانک درt9 قرار می گیرد. بعد در t8، مقدار در آدرس در حافظه با آدرس فانک را می ریزد. بعد عدد 2048 در t7قرار می گیرد. بعد t7وt8 ایکسور شده و در t8 می ریزیم(در واقع بیت 11ام آدرس در t8 با یک ایکسور می شود). حال t8را در آدرس t9یعنی جایی که قبلا اولین خط فانک در آن ذخیره شده بود می ریزیم. 33 که لیبل است، به 34 می رویم در آنجا به تابع دوم پرش می کنیم. قبل از ادامه توجه کنید که در آن دستور بیتی که تغییر می کرد در rd بود که a0 از جدول نگاه کنیم با 00100معادل است. الان با این تغییر بیت اول این تغییر کند می شود 00101که همان a1 است. پس در حقیقت rdما عوض شد.

2.آدرس لیبل go_to_exitدر t6قرار می گیرد و خود دستور در حافظه در t4ریخته می شود. همینطور آدرسjump_section در t6 قرار می گیرد و بعد t4 که آدرس لیبل go_to_exitرا داشت در حافظه با آدرس t6 میریزیم که قبلا جای لیبل jump_section بود. در حقیقت با اینکار دفعه بعدی که به jump برسیم دستورات exitاجرا خواهند شد. حال دوباره به func رسیدیم و همانطور که در خط آخر پاراگراف قبل توضیح دادیم این دفعه جمع a1, a0را در a1 می ریزد پسa1 می شود جمع 13 و 8 یعنی 21. ادامه همان پاراگراف قبل است با این تفاوت که این دفعه با ایکسور شدن دوباره ، خط اول فانک به حالت پیشین بر میگردد و همینطور با رسیدن به jumpسابق در حقیقت به لیبل go_to_exit می رویم با و با کد 10 سیسکال کرده برنامه تمام می شود

13 و 21 مقادیر نهایی به ترتیب a0 و a1 است

ه)

بجای خط 7:

```
Li $v0,5

Syscall

Move $a0, $v0
```

بجای خط 8:

```
Li $v0,5
```

Syscall

Move $a1, $v0

سوال2)

توضیح خط به خط بعد از کد

```
1 .data
2 a: .word 2
3
4 .text
5 main:
6 la $s0, a
7 la $s1, op
8 sub $s0, $s1, $s0
9 sub $s1, $zero , $s0
10 lw $s0, a($s0)
11 lw $s1, op($s1)
12 add $s0, $s0, $s1
13 sw $s0, op($zero)
14 la $s0, label
15 la $s1, op
16 sub $s0, $s1, $s0
17 sw $s0, a($zero)
18 label:
19 li $s0, 10
20 li $s1, 15
21 op:
22 add $a0, $s0, $s1
23 jal print_int
24 lw $a0, a($zero)
25 jal print_int
26 jal end
27
28 print_int:
29 li $v0, 1
30 syscall
31 jr $ra
32
33 end:
34 li $v0, 10
35 syscall
```

در دو خط اول داده ها تعریف می شوند و a ، آدرس خانه ای حاوی عدد 2ست.

در ادامه به بخش دستورات می رسیم در خط 6 آدرس a را در s0 و در 7 آدرس خط اول دستور op را در s1 می ریزیم. خط بعد(8) در s0 ما s1-s0یا آدرس aمنهای آدرس op را می ریزیم. در 9 0 منهای آن می کنیم یعنی به نوعی s0را قرینه می کنیم (a-op) و در s1می ریزیم. در خط 9 مقدار در پرانتز به علاوه آفست می شود s0 + aو چون s0=op-a پس op-a+a=op یعنی داده در

آدرس لیبل opرا در s0 می ریزیم. در خط 11 هم مشابه همین اتفاق ولی برعکس می افتد یعنی چون s1=a-op پس op+s1=a و داده در خانه با آدرسaرا در s1می ریزیم یعنی مقدار 2. حال s1را با s0 جمع کرده در s0 می ریزیم پس s0می شود جمع مقدار خود آدرس opکه باینری دستور add $a0, $s0, $s1 بود به علاوه 2 که قسمت 5 بیت سمت راست که func هست 2 تا تغییر می کند و این دستور به subتبدیل می شود. در خط 13 این دستور تغییر یافته که در s0 بود در خط اول op ذخیره می شود یعنی کد خودش را تغییر می دهد.

در خط 14 آدرس labelدر s0 و در خط 15 آدرس opدر s1 ذخیره می شود. بعد در 16، s1منهای s0 می شود که همان فاصله دو دستور است، دو خط با هم فاصله دارند و باتوجه به 4 بایتی بودن کلمه ها می شود 8 بایت اختلاف. مقدار s0 8 خواهد بود. بعد در خط 17 مقدار این ثبات را در آدرس aمی ریزیم پس aهم از 2 به 8 تغییر می کند

با رسیدن به لیبل label 10 را در s0 و 15 را در s1 می ریزیم.

به opرسیدیم ولی توجه کنید طبق صحبت های بالا الان addبه subتبدیل شده و حاصله s1-s0در a0 ریخته می شود (15-10) پس مقدار در a0 می شود منفی 5. همانطور که در راهنمایی آمده در خط 23 عدد صحیح در a0 که منفی 5 است توسط تابع چاپ می شود و به خطی که آن را فراخوانده بود با jr $ar باز می گیردد.

بعد در در خط 24 مقدار در خانه a که 8 بود در a0 می ریزیم و بعد باز تابع پرینت را کال می کنیم، که این بار مقدار 8 را چاپ می کند. در خط 26 به لیبل خروج می رویم، آنجا با قرار دادن کد 10 در v0 که نشانه خاتمه ی اجرای برنامه است و فراخوانی سیستمی (syscall) برنامه پایان می یابد.

خروجی:

-58

یعنی ابتدا منفی 5 و بعد 8 چاپ می شود، و البته newline چاپ نکرد برنامه.

سوال3)

الف)

عدد در A:

123456789ABCDEF0

اعداد بیان شده صرفا به ترتیب خوانش آن هستند.

دستگاه Aبه گره اول: چون A بیگ اندین و دومی لیتل اندین است اطلاعات را برعکس پردازش می کند و عدد مارا به صورت روبرو معنی می کند :

F0DEBC9A78563412

گره اول به دوم: چون اولی لیتل اندین و دومی بیگ اندین است اطلاعات برعکس پردازش شده را همانطور که ذخیره شده معنا می کند و عدد مارا به صورت روبرو در نظر می گیرد :

F0DEBC9A78563412

گره دوم و سوم: دومین گره بیگ اندین و سومین لیتل اندین است پس بار دیگر اطلاعات را برعکس می خواند:

123456789ABCDEF0

گره سوم و چهارم: سومی لیتل اندین و چهارمی بیگ اندین، اطلاعات همینطور می مانند:

123456789ABCDEF0

در نهایت در ارسال گره چهارم به دستگاه Bچون گره بیگ اندین و دستگاه لیتل اندین است باز اطلاعات معکوس پردازش شده و به شکل زیر تغییر می یابد:

F0DEBC9A78563412

ب)

از آنجا که هریک اطلاعات را به روش خودش می خواند، اگر مانند الف جلو برویم می بینیم یک مرحله کمتر اطلاعات برعکس می شود و اتفاقا این دفعه اطلاعات به درستی به دستگاه Bخواهد رسید. با اینکه اینجا دستگاه ها و گره ها از یکدیگر بی اطلاع بودند و شانسی این خرابی موجب درست شدن عملیات شد، ولی در کل این تفسیر اشتباه در سیستم هایی که نتایج آن ها حیاتی است مانند معاملات تجاری و پردازش های مالی می تواند موجب خسران عظیمی شود.

ج)

این پروتکل ها مخصوصا برای همین مشکل طراحی شدند، و ایده اصلی بیان یک استاندارد کلی در ارسال و دریافت و دستور عمل هنگام تبدیل داده دستگاه به استاندارد یا بالعکس است.

در این پروتکل ها همه داده به شکل big endianارسال و دریافت می شود تا مطمئن شویم اطلاعات از یک استاندارد پیروی می کنند. حال هنگام دریافت گیرنده می تواند با توابع پیاده سازی شده داده را به زبان خود تبدیل کند و همینطور ارسال کننده هم با توابع استاندارد دیگر می تواند داده های ارسالی خود را استاندارد کند، با استفاده از توابعمانند:

Htonl :

از هاست به شبکه برای long variables

Ntohl

از شبکه به هاست برای long variables

مدیریت در این دولایه انتقال و استفاده تفاوت هایی باهم دارند:

در انتقال تمامی بایت های شبکه به شکل بیگ اندین تبدیل و ارسال می شوند و هردستگاه پس از دریافت باید به روش خود آن را سازگار کند (مثلا لیتل اندین کند).

در کاربرد ولی معمولا نیاز نیست کسی نگران این باشد چون عموما پروتکل های این مورد بر اساس متن هستند که اکثرا UTF یا ASCII استفاده می کنند هرچند در کاربرد ها خیلی خیلی دقیق شویم ممکن است در برخی از مواقع برنامه نویس برای ارتباط بین چند مدیا مجبور باشد نوع داده را هم مشخص کند ولی در آن مورد هم رغبت به سمت بیگ اندین کردن داده است و ترجیح می دهند هردستگاه مبدل و کتابخانه خود را داشته باشد تا در ارتباط خللی پیش نیاید.

مثال:

یک مثال ملموس می‌تواند در هنگام پردازش داده‌ها از یک پایگاه داده باشد. اگر یک نرم‌افزار به داده‌ایی از یک سیستم-little endianنیاز داشته باشد، اما پروتکل شبکه به صورت big-endian داده‌ها را ارسال کند، نرم‌افزار می‌تواند با استفاده از توابع تبدیل، داده‌ها را به فرمت مورد نیاز تبدیل کند.

به این ترتیب، پروتکل‌های شبکه و توابع مربوطه با توجه به تفاوت endian در سطوح مختلف شبکه حل می‌شوند و این امر باعث تضمین درستی تبادل اطلاعات بین سیستم‌های مختلف با معماری متفاوت می‌شود.

سوال4)

الف)

در بیگ اندین بایت پرارزش تر در آدرس با ارزش کمتر می رود:

0x1000 → A1 0x1001→B2 0x1002→C3 0x1003→D4

ب)

این هم همچنان مانند قبل است و از چپ به راست حافظه پر می شود(از بایت های با ارزش کم به زیاد)

0x1000 → A1 0x1001→B2 0x1002→C3 0x1003→D4 0x1004→12 0x1005→34

ج)

بایت پر ارزش تر در بایت پر ارزش تر(ادامه صفحه بعد)

0x1000 →D4 0x1001→C3 0x1002→B2 0x1003→ A1

د)

بایت پر ارزش تر در بایت پر ارزش تر و بایت کم ارزش در کم ارزش تر

0x1004→34 0x1005→12

طبعیتا اگر لیتل اندین خانه هارا نخوانیم کلا سیستم به فنا رفته و بجای 0x1234، عدد 0x3412می خواند.

ه)

همه را برعکس می خواند و هیچکدام را نمی تواند درست استخراج کند. مثلا تو این مورد سیستم بیگ اندین 0xA1B2C3D4 را همینگونه به ترتیب در حافظه می ریزد ولی سیستم لیتل اندین هنگام خوانش این ، جایگاه هارا برعکس می کند و عدد به شکل 0xD4C3B2A1 خوانش می شود. این دنباله(توالی) به دلیل تفاوت ساختاری نحوه ذخیره سازی بایت هاست که موجب سوء تفاهم در خواندن و نوشتن شده.

سوال5)

تا قبل از رسیدن به خطوط دستور sub و and صرفا متغیر های number1وnumber2وnumber3را در t3وt4وt5 می ریزیم.

اول توضیح صرف این کد با بررسی 54:

ابتدا با 3 یعنی 00000000000000000000000000000011 andمی شود پس فقط دوبیت راست می ماند که 2 است. حال t5 که 2 در آن مانده را منهای t3 که یک است می کنیم که می شود 1 و در t5می ریزیم. این نشان می دهد بر 4 بخش پذیر نبوده.

توضیح کلی:

این andکردن دوبیت راست را می دهد، می دانیم اگر عدد مضرب 4 باشد دو بیت راستی اش باید 00 باشد، حال با این اند کردن دو بیت راست باقی می ماند که هرچه غیر 00 بود من جمله 1 و 2 و 3 وقتی منهای یک شود باز نامنفی می مانند ولی اگر 00 باشد منفی یک در عدد ذخیره می شود.

سوال6)

در big endian نحوه قرار گیری شبیه همان زبان آدمیزاد است از چپ به راست. پس بررسی می کنیم در حالت زبان طبیعی خودمان این عملیات ها درست بودند از نوع big endianاست در غیر اینصورت little endian.

سیستم اول:

7142 + 32c2 = a404

به زبان طبیعی درست است پس big endian

سیستم دوم:

e9b6 + 3acb = 2382

این اشتباه است پس little endian (دقت کنید که با برعکس کردن بایت ها به شکل b6e9-cb3a = 8223درست می شود)

سیستم سوم:

1dd5 - 8841 = 9593

در زبان طبیعی معنی نمی دهد و اشتباه است پس little endian(مانند قسمت قبل بایت هارا برعکس کنیم می بینیم درست می شود)

سیستم چهارم:

adee − 6002 = 4dec

در زبان طبیعی ما و ریاضیات درست است ←پس big endianاست