تمرین تئوری هفتم

ساختار و زبان کامپیوتر

سیدمحمدرضا جوادی
402105868
پاییز 1403

فهرست مطالب

سوال اول)

آ)

چون لیتل اندین است، هر کلمه بایت با ارزش پایین تر در خانه با ارزش پایین تر می رود،

به نوعی اگر از بالا به پایین ارزش موقعیت حافظه افزایش یابد بایت ها اینگونه ذخیره می شوند:

00110100

00010010

11111110

11111111

هنگام لود شدن در رجیستر به صورت extended 0اضافه می شوند پس:

```
1 segment .data
2 data: dw 4660, -2
3 segment .text
4 mov ax, [data]
                                                                    52
5 mov ax, [data+1]
                                                                    18
6 mov ax, [data+2]
                                                                   254
```

ب)

مقادیر صحیح قرار گرفته در رجیستر بعد از هر دستور:

```
1 mov al, '8'
                                                                    56
2 add al, '7'
                                                                 111=56+55
3 daa
این دستور مقدار عددی (مثلا 8 اینجا کاراکتر پاس داده شده، ولی این دستور میاد میگه تو اون دو مورد عدد bcd تبدیل می کنه و به اون صورت
جمع می کنه حاصلشو میریزه تو ax، یعنی عدد 15 به صورت bcd ذخیره می شود:
                                                               0001 0101
که مقدار راستی در al و چپی در  ah ذخیره می شود(al=5,ah=1)
4 mov ah, '1'
                                                          49 در ah می رود

5 add ah, '9'
                                                              49+57=106
6 aaa
مشابه daa ولی خب با رجیستر ah کار میکند. مقدار جمع به صورت bcd ذخیره می شود یعنی:
```

```
0000 1010
پس مقدار al=10 و مقدار ah=0
```

ج)

اولی)

چک می کند آیا عدد منفی است یا نه

```
cwd dx,ax
```

دومی)

بقیه در یک دستور جا نمی شوند.

سوال2)

مراحل برنامه‌نویسی و بارگذاری DLL

1. کامپایل کردن

کد سی به کد اسمبلی ترجمه می شود

2. اسمبل کردن

کد اسمبلی مرحله قبل،به کد ماشین تبدیل می‌شود.

3. لینک کردن

در این مرحله، فایل‌های شی با کتابخانه‌های SLL که استفاده می‌کنید، ترکیب می‌شوند. وقتی به توابعی در SLL نیاز دارید، کامپایلر باید آدرسی برای آنها پیدا کند. اینجا جداولی به نام جدول نشانه‌ها Symbol Tableو جدول جابه‌جایی Relocation Table ایجاد می‌شود. جدول نشانه‌ها شامل نام توابع و مکان‌های مربوط به آنهاست، در حالی که جدول جابه‌جایی به ماشین می‌گوید که چه چیزهایی باید تغییر کند(مانند دستورات پرش یا کار با حافظه)

4. بارگذاری

زمانی که شما برنامه را اجرا می‌کنید، سیستم‌عامل مسئول بارگذاری برنامه به حافظه است. اینجا، آدرس‌های توابع موجود در DLL به آدرس‌های واقعی تبدیل می‌شوند و همه‌چیز آماده می‌شود تا برنامه اجرا شود.

فرآیند حل آدرس‌ها

وقتی که برنامه اجرا می‌شود و یک تابع را از DLL می‌خواهد، باید آدرس واقعی آن تابع مشخص شود. اینجا دو مفهوم مهم به کار می‌آید:

1. PLT (Procedure Linkage Table)

این جدول برای مدیریت فراخوانی توابع به کار می‌رود. هر بار که شما به یک تابع در DLL دسترسی پیدا می‌کنید، سیستم به PLT نگاه می‌کند. اگر تابع برای اولین بار فراخوانی شده باشد، PLT آدرس واقعی این تابع را مشخص می‌کند و در دفعات بعدی، از این آدرس استفاده می‌شود.

2. GOT (Global Offset Table)

این جدول شامل آدرس‌های واقعی توابع و داده‌های دیگه‌ای است که در برنامه استفاده می‌شوند. GOT به PLT کمک می‌کند تا اجزای مختلف برنامه را درست بارگذاری کند.

سوال3)

آ)

آرایه)

روش indexed زیرا راحت تر می توانیم با داشتن نقطه شروع یا پایان نسبت به آن آدرس بقیه را بگوییم. البته اگر لیبلی در data. داشته باشد روش displacement نیز مشابها مفید خواهید بود.

ماشین حساب)

روش direct تا مستقیما نتایج یا محاسبات را در نقطه ای مشخص و محدود بریزیم.

چندلایه)

روش scaled تا بتوانیم به هرلایه دسترسی داشته باشیم و در هرلایه iterate کنیم.

ب)

منبع من https://stackoverflow.com/questions/38966919/do-complex-addressing-modes-have-extra-overhead-for-loads-from-memory است.

در حقیقت در جواب به سوال کلی تری مانند "آیا روش های آدرس دهی های پیچیده تر هزینه ی بیشتری دارند" صحبت کنیم که در حالت کلی به پردازنده بستگی دارد، ولی در مورد اینتل بهینه سازی هایی انجام می شود و در کل هزینه کمتر از mips می شود. اگر دستوری مانند scaled addressingرا بیشتر باز کنیم و به 3 یا 4زیر دستور تقسیم کنیم، اینطور به نظر می رسد که یک دستور معادل 5 تا 6 دستور ریز تر شده پس کند تر است ولی در حقیقت بیشتر این ها مستقلند و در پردازنده هنگامی که با چنین چیزی مواجه می شود به صورت موازی پردازش را انجام می دهد پس اگر فکر می کردیم 6 دستور جمعا 6 سایکل می خواهند، این به 1 یا 2 تقلیل می یابد و به نوعی اندازه یک دستور عادی زمان می برد. در آنسو در پردازنده ای مانند میپس ما واقعا مجبوریم آن چندخط را پیاده سازی کنیم و در کل هزینه همان دستورات را می گیرد. اینگونه اینتل با پشتیبانی این دستورات پرفورمنس بهتری هم در این نوع می گیرد.

ج)

حوزه بررسی/نام	optimizing	JIT	Multi-pass	Single-pass
Performance	بسیار بالا چون در چند زمان در چند لایه این بهینه سازی می کند و حتی خود سورس کد برنامه نویس را به نفع سرعت تغییر می دهد.	نسبت به مفسر های دیگر سرعت بسیار بهتری دارد زیرا می تواند توابع را inlineکند، کد های مرده را حذف کند حلقه هارا باز کند و ... کلی بهینه سازی در حین اجرا انجام دهد	سرعت کامپایل کمتر می شود زیرا چند دور بر روی کد می زند و بهینه می کند. ولی در اجرای آن کد سریع تر اجرا می شود	از منظر سرعت کامپایل سریعتر ولی معمولا سرعت اجرای کد کامپایل شده کمتر از بقیه است ولی در کد های کوچک مناسب تر است.

Complexity	پیاده سازی این از همه سخت تر است زیرا عملا سورس کد هم دوبار چه قبل اجرا و چه در حین اجرا بررسی می کند و حتی ساختار را به نفع سرعت تغییر می دهد	نسبتا پیچیدگی هایی درپیاده سازی دارد زیرا عملا یک ماشین مجازی بر روی سیستم نصب می کند و کد را دومرحله ترجمه می کند. متوسط است.	پیاده سازی آن سخت تر است چون چند بار هردفعه تغییراتی را اعمال می کند و کند تر است ولی برای زبان های پیچیده تر مناسب تر است زیرا می تواند در چند دور آن هارا ترجمه کند.	کمتر از بقیه، فقط کد را مستقیما تبدیل می کند بدون هیچ بازدید مجددی و خط به خط. البته زبان های پیچیده تر را سخت تر کامپایل می کند.
Embedded	در سیستم هایی که متحرک اند و سرعت در عین مصرف کم سخت افزار مهم است کاربرد دارد زیرا کد به شدت بهینه سازی شده و بهترین حالت خود است حتی به قیمت تغییر کد برنامه نویس	در بسیاری از موارد که به اتصال بین دستگاه ها مربوط می شود و ماشین مجازی نیاز است کاربرد دارند(مانند وب) می توان به جاوا اشاره کرد که در بسیاری از صنایع جای دارد	کاربرد های فراوانی در همه حوزه های صنعت دارند، زیرا در کد های بسیاری نیاز داریم تا کد یکبار کامپایل شود و بعد از آن بار ها اجرا پس هزینه زمانی موقع کامپایل مهم نیست در عوض سرعت اجرا بالاتر است	نسبتا مناسب، یعنی شاید ایده آل ترین حالت نباشد ولی می توان از آن استفاده کرد اگر کد ترجمه شده همچنان realtime بودن را حفظ کند مثلا در کد های سایز پایین کاربرد دارد.

حوزه بررسی/نام	*Ahead-Of-Time*	*Incremental*
Performance	تفاوت عمده با JIT این است که قبل از اجرا به اصطلاح پیش کامپایلی انجام می دهد و کارهایی که در آینده باید بکند و متغیر ها و حلقه ها و ... را بررسی اجمالی می کند و بهینه سازی هارا در نظر می گیرد. در کد هایی که عملکرد سنگینی در حین اجرا دارند مانند فعالیت های علمی فوق العاده است، می توانیم در دستگاهی کامپایل کنیم و به دستگاه های دیگر کد را بدهیم (در انواع معماری عملکرد یکسان دارند)	از آنجا که هردفعه به بخش جدید می پردازد ممکن است برخی بهینه سازی های میان بخش های قدیم و جدید را از دست بدهد و کلا سرعت بیشتر با single pass قابل مقایسه است. در آنسو سرعت کامپایل برنامه بسیار زیاد است بیشتر از همه زیر حتی وقتی کد بزرگی داریم فقط نسبت به کد قبلی تغییرات را می سنجد و کامپایل می کند/
Complexity	بسیار زیاد چون عملا پیچیدگی های JIT را که داریم هیچ، باید قبل اجرا نیز کد را بتوانیم بررسی کنیم و بهینه سازی ها هارا انجام دهیم. که این بهینه سازی ها علاوه بر طراحی چنین ماشین مجازی پیچیدگی پیاده سازی را بسیار زیاد می کند(هرچند هنگام اجرا برنامه سبکتر است)	بسیار زیاد چون دیگر نه تنها وظیفه ترجمه و بهینه سازی را داریم بلکه باید تغییرات، دپندنسی ها و ... را در کد تشخیص دهیم و بخش های جدید را لینک کنیم و حتی اگر مورد کوچکی تغییر را نتوانیم شناسایی کنیم و تغییرات را کامپایل کنیم هنگام اجرا ممکن است ارور بخوریم. تجهیزات سخت افزاری بیشتری نیز می خواهد.
Embedded	به شدت مناسب است زیرا مناسب سیستم های با شرایط سفت و ددلاین های مهم هستند. در یک بار ترجمه به صورتی کدی بهینه سازی شده برای ماشین مجازی درمی آید و در همه معماری ها ماشین کد تولید شده قابل اجرا است.	کاربرد خیلی کمی دارد. بیشتر در محصولات نرم افزاری که نیازمند تست دائم اند مناسب است زیرا بعد هرتغییر باید تست کنند و اگر هربار از اول کد بزرگ ما کامپایل شود مدت زمان طولانی صرف کامپایل شدن هدر می رود درحالی که چندخط کد اضافه کردیم و تست آن مهم بود.

د)

سرعت)

طبیعتا مفسر ها زمان اجرای بیشتری می گیرند زیرا صرف اجرا دستور نیست بلکه همزمان باید ترجمه هم کنند. در حالی که اورهد ترجمه به قبل از اجرا می رود در کامپایلر ها

اندازه کد)

از آنجا که کامپایلر ها زمان بیشتری قبل اجرا دارند بهینه سازی هایی می کنند و حجم کد می تواند کمتر باشد("گاهی" به این خاطر است که ممکن است در یک حلقه که به طور ثابت 100 بار اجرا می شود برای کم کردن 100 دستور مقایسه حلقه را باز کند. ولی باز حجم کد اجرایی قطعا کمتر است)

نیاز به منابع سخت افزاری)

در هنگام اجرا مفسر ها منبع بیشتری نیاز دارند تا همزمان ترجمه و ... هم انجام دهند ولی برای کامپایلر ها در حین اجرا فقط سخت فزار باید دستورات را اجرا کند نه بیشتر.

سوال4)

به ترتیب:

Register indirect

lw $r1,0($r2)

Direct

lw $r1, ox2000($0)

Memory indirect

توضیح: رجیستر حاوی آدرس خانه ای است که در آن خانه آدرس مقدار مورد نظر در حافظه وجود دارد

Lw $at, 0($r2)

Lw $at, 0($at)

Add $r1,$r1,$at

Scaled

توضیح: به شکل displacement+r3*scale+r2 است.

move $at,scale

mul $r3,$at

mflo $at

add $at,$r2,$at

lw $at, 8($at)

sub $r1,$r1,$at

Auto-Increment

توضیح:بعد عملیات رجیستر مورد استفاده را اندازه یک کلمه افزایش می دهد.

Lw $at, 0($r1)

Subi $sp,$sp,4

Sw $at, ($sp)

Addi $r1,$r1,4

Indexed

add $at,$r1,$r2

sw $r1,0($at)

سوال5)

الف)

مانند بخش آ سوال3 است. Indexed مانند پیمایش یک آرایه، scaled مانند حافظه های چند لایه مانند ماتریس ها، و auto decrement مانند وقتی از بالای استک می آییم یا به طور کلی از انتهای لیستی به اولش می آییم.

مزایای این مود ها در سوال 3 هم ذکر شده اند، سرعت اجرا، کوتاه و موجز شدن کد، آسان تر شدن کار برنامه نویس، همینطور رها شدن از نگرانی alignment ها. این قالب مثال ها هم بسیار متدوال اند و پرداختن به دستور مناسب در ISA هم سرعت را بالا برده هم کار برنامه نویس راحت تر می شود.

ب)

بالطبع پردازنده هایی که پشتیبانی می کنند مانند x86 از امکانات سخت افزاری بیشتر به طور متوسط برخوردارند (مثلا اینتل در رایانه های شخصی یا سرور ها) پس فراهم کردن نیاز های سخت افزاری چالش بزرگی برای آن ها نیست. کاربرد عمده میپس در نهفته هاست جایی که مشخصا ارزانی سحت افزار برایمان اولویت دارد زیرا هم قیمت مهم است هم حجم و هم مصرف پردازنده و عمر آن و افزودن پیچیدگی های سخت افزاری اگر ما به هدف real time بودن رسیده باشیم مورد تایید نیست. در مورد افزایش کارایی درصورت پیاده سازی indexed به تفضیل در الف و 3 آ شرح داده ایم ولی محدودیتی که ایجاد می کند را هم به تفضیل حالت کلی تر را توضیح دادیم.

سوال6)

الف)

در خط 10 مقدار مستقیم(immediate) داریم. در 11 هم مقدار مستقیم یا immediate و در ادامه دو دستور داریم: خط 13 مستقیم(immediate) و در 14 register indirect ولی درمجموع یک دستور را پی می گیرند که مقدار داخل حافظه N در رجیستر بریزد . خط 12 هم مستقیم(immediate) بود. در 17 یک auto increment داریم که مقدار آن خانه در حافظه را در رجیستر ریخته می شد و بعد رجیستر دومی را به اندازه یک کلمه (4بایت) افزایش می دهد. خط 22 هم یک register indirectداریم.

ب)

تی ای ما(پوریا) تو گروه گفتن که نیاز نیست همه را پیاده کنیم.

پیاده سازی حلقه با scaled:

```
Mov r10,#0

LOOP:

Ldr r4, #0[r2,r10,#4]

Add r0,r0,r4

adds r3,r3,#-1

inc r10

bne loop

str r0,[r1]

B end
```

ج)

خب این همان قبلی است صرفا بخش حلقه را باید جایگزین کنیم.

```
1 .data
2 SUM: .word 0
```

```
3 DATA: .word 5, 8, 3, 12
4 N: .word 4
5
6 .text
7 .global _start
8
9 _start:
10 LDR R1, =SUM ; Load address of SUM
11 MOV R0, #0 ; Initialize R0 (temporary sum) to 0
12 LDR R2, =DATA
13 LDR R3, =N
14 LDR R3, [R3]
15
16 Mov r10,#0
17 LOOP:
18 Ldr r4, #0[R2, R10,#4]
19 Add R0, R0 , R4
20 adds R3, R3,#-1
21 inc R10
22 bne loop
23 str R0,[ R1]
24
25 END:
26 NOP ; Placeholder for program termination
```