



دانشگاه صنعتی شریف

دانشکده مهندسی برق

ساختار کامپیوتر و ریزپردازنده و آزمایشگاه

نیمسال دوم ۹۳-۹۴

آزمایش شماره ۳

شبیه‌ساز اسمبلی MIPS

تهیه کننده:

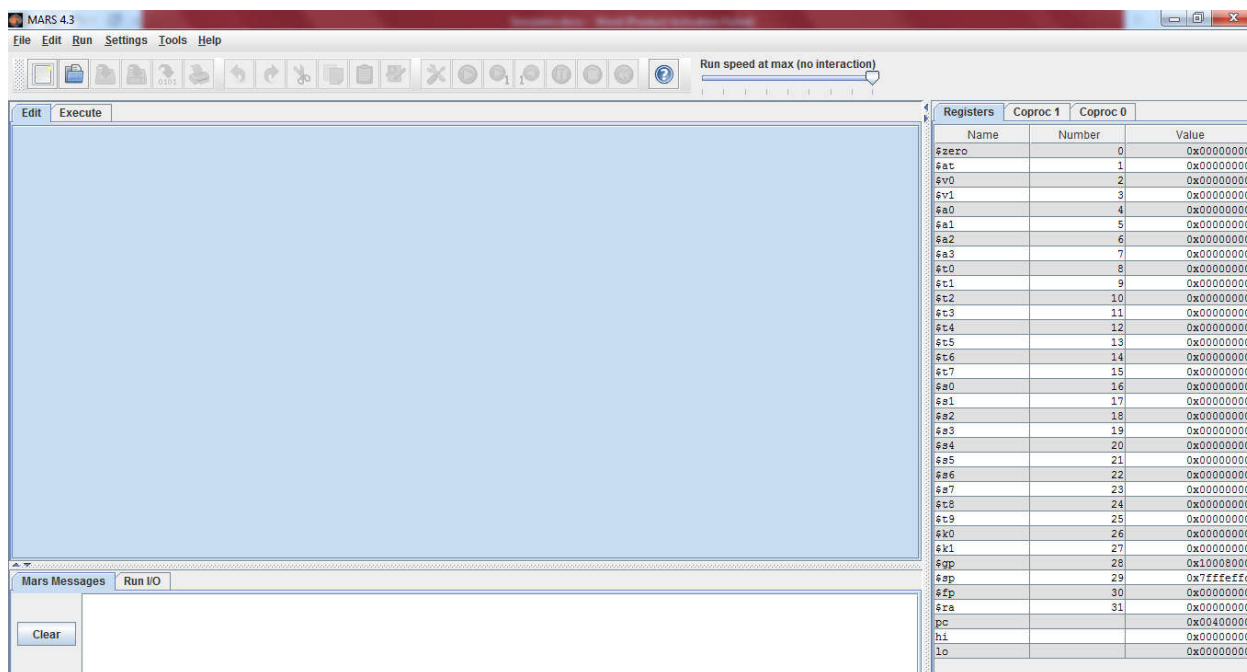
سید مهدی حسینی

در این آزمایش قصد داریم شما را با محیط یکی از شبیه‌سازهای پردازنده‌های مبتنی بر معماری MIPS آشنا کنیم. نرم‌افزاری که برای این کار انتخاب شده است، شبیه‌ساز MARS است که توسط محققین دانشگاه “Missouri State University” و بر اساس کتاب “Computer Organization and Design” طراحی و ارتقا داده شده است. در ابتدا با محیط این نرم‌افزار آشنا می‌شویم، سپس چند کد اسمبلی MIPS را به کمک آن شبیه‌سازی می‌کنیم.

۱- شبیه ساز MARS:

برای اینکه بتوانید این شبیه‌ساز را بر روی سیستم خود اجرا کنید در ابتدا لازم است که نرم‌افزار “Java Runtime” را بر روی سیستم خود نصب کرده باشد (فایل‌های نصب نسخه ۳۲ و ۶۴ بیتی Java Runtime بر روی لوح فشرده‌ای که در اختیارتان قرار گرفته موجود است). سپس کافی است که بر روی فایل Mars4_3.jar دبل کلیک کنید (دقت کنید که پسوند این فایل jar است و نباید از حالت فشرده خارج شود).

پس از اجرا پنجره‌ای به شکل زیر باز می‌شود:

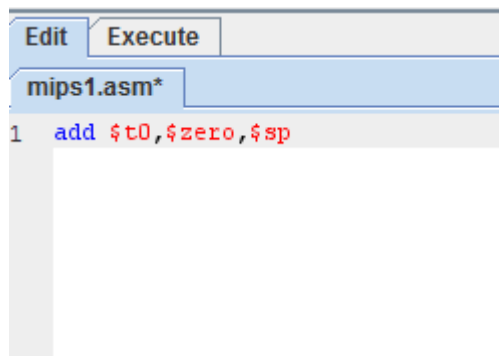


رجیسترها:

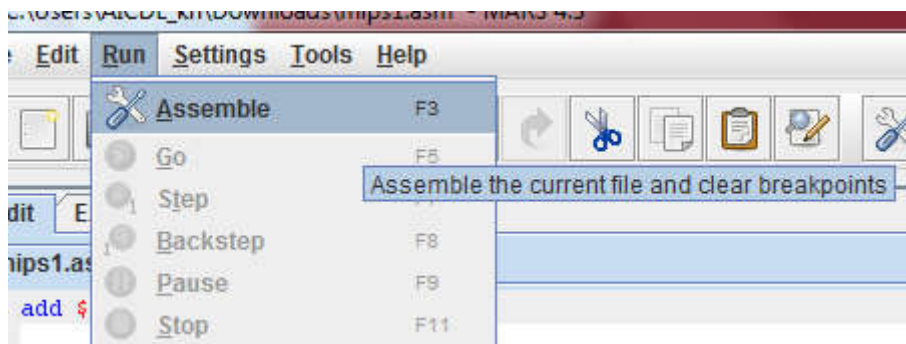
در قسمت سمت راست رجیسترهای پردازنده قرار گرفته‌اند که دارای سه بخش رجیسترهای عمومی، رجیسترهای ممیز شناور و رجیسترهای مربوط به رسیدگی به حالات استثنا قرار دارند که به ترتیب با نام‌های Registers, Coproc1 و Coproc0 مشخص شده‌اند. در هر بخش نام اسمبلی رجیستر (مانند \$a0)، آدرس رجیستر (مانند 4) و محتوای کنونی آن (مانند 0x00000000) قرار دارد. در این قسمت می‌توان با دبل کلیک بر روی محتوای هر رجیستر مقدار آن را تغییر داد.

محیط ویرایشگر:

قسمت آبی‌رنگ محیط ویرایشگر را نشان می‌دهد، این قسمت دارای دو زیر بخش *Edit* و *Execute* است که از بخش *Edit* آن برای نوشتن کد اسمبلی و اصلاح آن استفاده می‌کنیم. برای ایجاد یک فایل اسمبلی جدید از منوی *File* گزینه *New* را انتخاب کنید. همانطور که مشاهده می‌کنید یک فایل جدید به نام *mips1.asm* و با پسوند *.asm* باز می‌شود که قرار است کد اسمبلی را در آن بنویسیم:

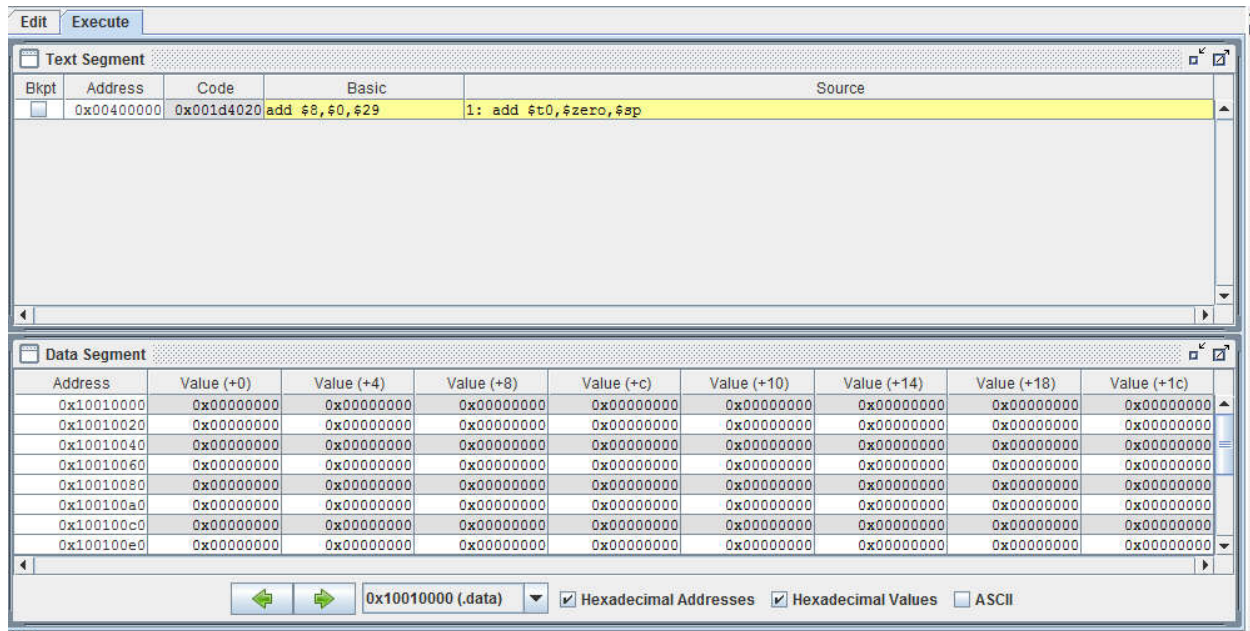


فراموش نکنید پس از نوشتن کد، فایل ایجاد شده را ذخیره کنید. برای مثال، در شکل بالا یک خط کد اسمبلی نوشته شده است که محتوای رجیستر *\$sp* را به وسیله جمع با رجیستر صفر به رجیستر *\$t0* منتقل می‌کند. همانطور که مشاهده می‌کنید در کد بالا ما از نام‌های قراردادی رجیسترها (مانند *\$sp*) استفاده کرده‌ایم که برای پردازنده نامفهوم است چرا که پردازنده رجیسترها را با آدرسشان می‌شناسد، همچنین موارد دیگری همچون نشانه‌گذاری حلقه‌ها، وارد کردن اعداد ثابت به شکل ده دهی، استفاده از دستوراتی که در مجموعه دستورات پردازنده وجود ندارد!! (که با نام *Pseudo assembly* شناخته می‌شوند) و ... که برای راحتی اسمبلر ایجاد شده‌اند باعث می‌شود که نیاز به ترجمه کد اسمبلی به زبان اسمبلی قابل فهم برای پردازنده و کد ماشین متناظر با آن باشد. این ترجمه را نرم افزار شبیه‌ساز برای ما انجام می‌دهد. برای این کار از منوی *Run* گزینه *Assemble* را انتخاب می‌کنیم:



در این مرحله اگر کد نوشته شده *Error* نداشته باشد، به زبان اسمبلی نهایی ترجمه شده و در زیر بخش *Execute* نمایش داده می‌شود. در صورتی که کد شما خطا داشته باشد در پنجره *MARS Messages* شماره خط مربوط به خطا و توضیحاتی در رابطه با علت آن نمایش داده می‌شود.

محیط *Execute* به صورت زیر است:

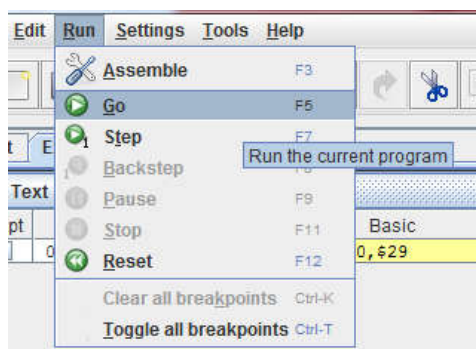


در بخش بالایی کد ترجمه شده شما قرار دارد. همانطور که مشاهده می‌کنید در اینجا رجیسترها با آدرس معادلشان جایگذاری شده‌اند و کد اسمبلی نهایی نوشته شده است (*Basic*) و کد ماشین متناظر با آن‌ها نیز تولید شده است (*Code*). بخش *Address* محل ذخیره شدن دستورالعمل در حافظه را نشان می‌دهد و بخش *Source* کد نوشته شده توسط شما را مشخص می‌کند تا متوجه شوید که این دستورالعمل به طور دقیق متناظر با کدام بخش از کد شماست.

در قسمت پایین محتوای حافظه نمایش داده شده است که در اینجا نیز با دبل کلیک بر روی هر بخش می‌توانید مقدار درون هر خانه از حافظه را جهت انجام شبیه‌سازی تغییر دهید.

اجرای کد:

حال می‌خواهیم کد ترجمه شده را اجرا کنیم. برای این کار از منوی *Run* گزینه *Go* را انتخاب می‌کنیم.



این گزینه تمام کد را اجرا می‌کند. گزینه *Step* فقط دستورالعمل بعد را اجرا می‌کند و گزینه *Reset* نیز به تبع تمام شبیه‌سازی را ریست می‌کند.

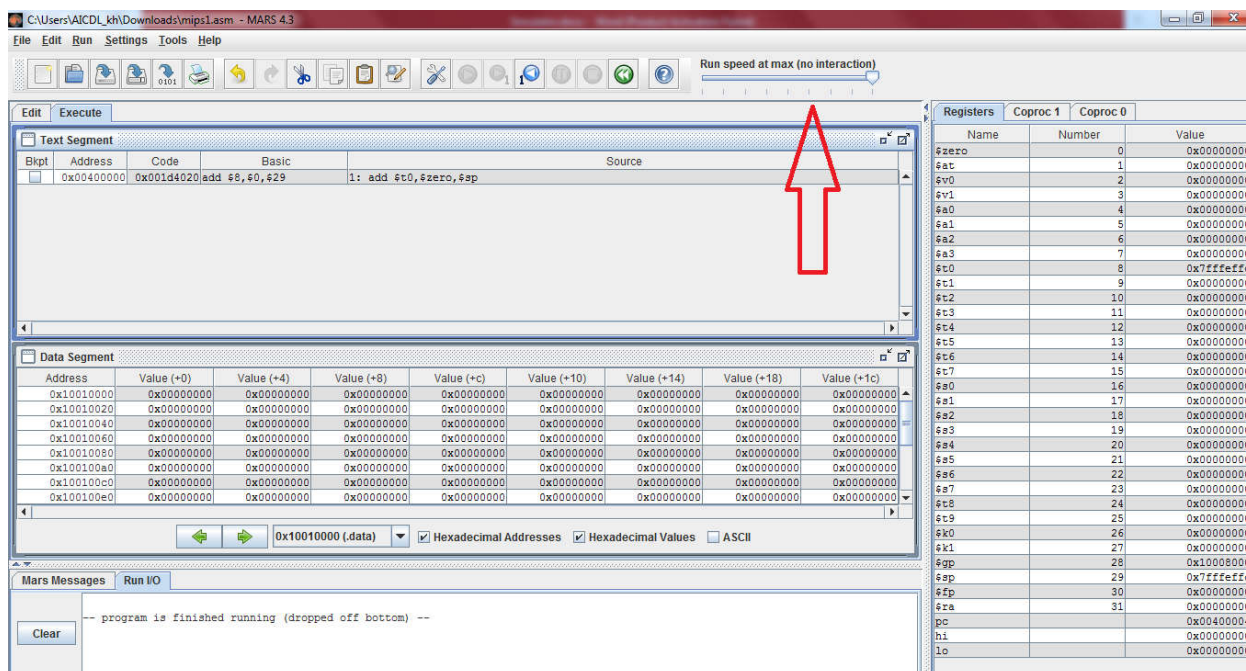
پس از انتخاب *Go* می‌توانیم نتیجه کار را در محتوای رجیسترها مشاهده کنیم.

Registers		
Coproc 1		
Coproc 0		
Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x7fffeffc
\$t1	9	0x00000000

همانطور که انتظار داشتیم محتوای رجیستر *\$sp* که مقدار *0x7fffeffc* بود درون رجیستر *\$t0* قرار گرفته است.

Run Speed

مثال بالا یک کد ساده یک خطی بود اما در موارد پیچیده‌تر، معمولاً علاقه‌مندیم که نتیجه اجرای خط به خط برنامه را ببینیم. شبیه ساز *MARS* گزینه‌ای را در اختیار ما قرار می‌دهد که به وسیله آن می‌توانیم انتخاب کنیم که سرعت اجرای خط به خط کد چه مقدار باشد تا به وسیله آن بتوانیم اجرا شدن خط به خط کد و نتایج حاصل از آن را مشاهده کنیم. این مشخصه توسط گزینه‌ی زیر قابل تنظیم است:



البته برای بررسی دقیق‌تر می‌توانید در زمان اجرا از گزینه *Step* استفاده کنید تا خودتان کد را خط به خط اجرا کنید.

- برای آشنایی بیشتر با شبیه‌ساز MARS می‌توانید به سایت توسعه‌دهندگان این نرم افزار مراجعه کنید:

<http://courses.missouristate.edu/KenVollmar/MARS/index.htm>

۲- یک مثال ساده

در این مثال قصد داریم برنامه‌ای را به زبان اسمبلی بنویسیم که دو عدد بدون علامت ۶۴ بیتی (!) را به وسیله دستورالعمل‌های ۳۲ بیتی MIPS با یکدیگر جمع کند و حاصل را درون دو رجیستر ۳۲ بیتی ذخیره کند. فرض کنید این اعداد مقادیر زیر را داشته باشند:

A=0x0154A8D0,EEE94560

B=0x0000102A,18002E00

می‌بایست حاصل برابر با S=0x0154B8FB06E97360 شود.

واضح است که به دلیل ۳۲ بیتی بودن پردازنده تمام دستورالعمل‌ها و رجیسترها ۳۲ بیتی است!

حل:

در ابتدا بایستی به وسیله دو دستور ori و lui مقادیر A و B را درون ۴ رجیستر قرار دهیم:

Edit	Execute
64bit_Add_Unsigned	
1	lui \$t0,0xeee9
2	ori \$t0,\$t0,0x4560
3	lui \$t1,0x0154
4	ori \$t1,\$t1,0xa8d0
5	lui \$t2,0x1800
6	ori \$t2,\$t2,0x2e00
7	lui \$t3,0x0000
8	ori \$t3,\$t3,0x102A

حال مقادیر عددی که دارای ارزش یکسان هستند را با یکدیگر جمع می‌کنیم، به عبارت دیگر ۳۲ بیت کم‌ارزش A با ۳۲ بیت کم‌ارزش B جمع می‌شود و همین‌طور ۳۲ بیت پرارزش (باید توجه داشت که جمع از نوع بدون علامت است):

9	addu \$t4,\$t0,\$t2
10	addu \$t5,\$t1,\$t3

اما اشکال کار اینجاست که جمع ۳۲ بیت کم‌ارزش ممکن است در بعضی موارد یک رقم نقلی تولید کند که در این صورت لازم است که نتیجه حاصل از جمع ۳۲ بیت‌های پرارزش به علاوه یک شود. در اینجا از این خاصیت استفاده می‌کنیم که "در صورت تولید رقم نقلی حاصل جمع از هر دو عدد ورودی کوچکتر است" (صحت این جمله را یک بار برای خودتان بررسی کنید). در نتیجه

کافی است حاصل جمع کم‌ارزش را با یکی از دو ورودی مقایسه کنیم و در صورتی که حاصل جمع کوچکتر بود، حاصل جمع پرارزش را با عدد یک جمع کنیم:

```
11 sltu $t6,$t4,$t0
12 addu $t5,$t5,$t6
```

توجه کنید که اگر مقدار حاصل، از مقدار ورودی کمتر باشد دستور sltu رجیستر \$t6 را یک می‌کند در غیر این صورت آن را برابر صفر قرار می‌دهد. در نتیجه، در انتها کافی است حاصل جمع پرارزش را با \$t6 جمع کنیم تا حاصل نهایی جمع در دو رجیستر {t5,t4} قرار بگیرد.

حال کد را اسمبل و اجرا می‌کنیم و محتوای رجیسترها را مشاهده می‌کنیم:

Registers	Coproc 1	Coproc 0	
Name	Number	Value	
\$zero	0	0x00000000	
\$at	1	0x00000000	
\$v0	2	0x00000000	
\$v1	3	0x00000000	
\$a0	4	0x00000000	
\$a1	5	0x00000000	
\$a2	6	0x00000000	
\$a3	7	0x00000000	
\$t0	8	0xee94560	
\$t1	9	0x0154a8d0	
\$t2	10	0x18002e00	
\$t3	11	0x0000102a	
\$t4	12	0x06e97360	
\$t5	13	0x0154b8fb	
\$t6	14	0x00000001	
\$t7	15	0x00000000	
\$t8	16	0x00000000	

نتیجه همان شد که انتظار آن را داشتیم. (کد اسمبلی این مثال در سامانه CW قرار داده شده است).

توجه ۱: در تمامی سوالات و پروژه‌های اختیاری بهینه بودن کد نیز در کنار عملکرد صحیح تابع، مورد بررسی و ارزشیابی قرار خواهد گرفت.

توجه ۲: در تمامی سوالات بالا دانشجویان به هیچ وجه مجاز به استفاده از دستورات *Pseudo assembly* نیستند و تمامی دستورات بایستی جزئی از *ISA* مربوط به *MIPS ۳۲* بیتی باشد. (*Pseudo assembly* دستوراتی است که در *ISA* وجود ندارند و برای راحتی اسمبلر اضافه شده‌اند و در نهایت به دستورات موجود در *ISA* ترجمه می‌شوند. برای مثال دستور *move* در *MIPS ISA* وجود ندارد و به وسیله جمع با رجیستر صفر انجام می‌شود، برای فهم بیشتر از دستور *move* در محیط *edit* استفاده کنید و ترجمه آن را در محیط *Execute* مشاهده کنید.) برای این که مطمئن شوید دستوراتی که نوشته‌اید *Pseudo assembly* نیست از منوی *settings* گزینه *Permit extended (pseudo) instruction and formats* را غیرفعال کنید. در این صورت استفاده از دستورات *Pseudo assembly* باعث ایجاد خطا خواهد شد.

