طراحی سیستم‌های دیجیتال

گزارش نهایی پروژه‌ی سوم

مهبد مجید	۹۵۱۰۹۳۷۲
سبحان محمدپور	۹۵۱۰۶۶۰۷
کیمیا حمیدیه	۹۵۱۰۹۴۳۴
روژین نوبهاری	۹۵۱۰۵۲۳۸
کوشا جافریان	۹۵۱۰۵۴۵۴

۲۰ تیر ۱۳۹۷

توصیف اولیه

مقدمه

امروزه با توجه به کاربرد گسترده‌ی Java و بالطبع JVM ، در صنعت و جهان مدرن امروزی منطقی به نظر می‌رسد که فرآیند اجرای کدهای جاوا را سریع‌تر کنیم. یکی از راه‌های خوب برای رسیدن به این مهم، می‌تواند پیاده‌سازی سخت‌افزاری JVM که در واقع هسته‌ی جاواست باشد.

اهداف

در این پروژه می‌خواهیم برای پردازنده‌ی ARM-7 (صبای ۲) یک شتاب‌دهنده [1] ی سخت‌افزاری JVM بسازیم. نحوه‌ی کار این شتاب‌دهنده به این شکل است که پردازنده opcodeهای JVM را دریافت می‌کند و به شتاب‌دهنده می‌دهد و شتاب‌دهنده دستورات معادل پردازنده را تولید می‌کند.

مراحل انجام پروژه

به طور کلی با توجه به اهداف پروژه ما باید ۳ کار را برای انجام پروژه انجام‌دهیم:

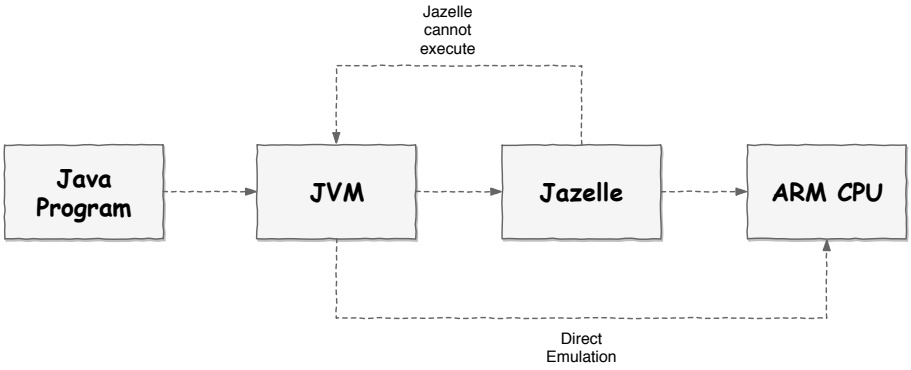
- یادگیری کار با ماشین JVM
- یادگیری کار با ماشین ARM-7
- ساخت مبدل برای تبدیل دستورات میان این دو

[1] accelerator

تقسیم‌بندی پروژه

انتخاب opcodeها

ابتدا لیست opcodeهای JVM را پیداکردیم. از آنجایی که قرار بود این پروژه برای ۵ تیم باشد، نیاز بود تا تعدادی از آپکدها را جداکنیم که پروژه برای ۴ تیم مناسب شود. برای جداکردن تعدادی از این opcodeها از Jazelle الگو می‌گیریم. Jazelle به این صورت عمل می‌کند که JVM دستوراتش را به Jazelle می‌فرستد و اگر Jazelle از آن‌ها پشتیبانی کرد، آن‌ها را اجرا می‌کند، و اگر هم پشتیبانی نمی‌کرد آن‌ها را به JVM باز می‌گرداند تا آن‌ها را به دستوراتی که Jazelle از آن‌ها پشتیبانی می‌کند تبدیل کند.



شکل ۱: روند اجرای کار jazelle

ما نیز به این صورت عمل می‌کنیم که عده‌ای از opcodeها که پیاده‌سازی نرم‌افزاریشان ساده‌تر است را جدا می‌کنیم و الباقی opcodeها را در پیاده‌سازیمان می‌آوریم.

تقسیم‌بندی opcodeها

برای افزایش بازدهی گروه‌ها opcodeهای انتخابی به شانزده بسته‌ی ۱۰تایی تقسیم‌بندی شدند و با یک کد R که به صورت رندوم این ۱۶ بسته را به گروه‌ها تخصیص می‌داد، تقسیم‌بندی کردیم.(سید را هم با حضور اعضای سایر گروه‌ها تعیین‌کردیم.)

کد

```r
library(dplyr)
set.seed(1919)
s = sample(seq(from = 1, to = 16), replace = F)
c("jaferian", "asadi", "hoseini", "ghafarloo") %>%
cbind(t(apply(matrix(s, nrow = 4), 1, sort)))
```

خروجی کد

Group Name				
jaferian	4	6	7	8
asadi	3	9	10	12
hoseini	1	5	11	14
ghafarloo	2	13	15	16

لیست دستورات

لیست دستورات را می‌توانید در صفحه‌گسترده‌ی ۱ و ۲ مشاهده‌کنید.

برخی مd[اژول‌های پیاده‌سازی‌شده با verilog

فایل memory.v

در این فایل دو ماژول حافظه طراحی شده‌است. ماژول اول memory_r است که به عنوان ورودی سیگنال‌های کلاک [۲] ، ریست [۳] ، شروع [۴] و آدرس [۵] را می‌گیرد. در ضمن دو سیگنال خروجی داده‌ی مورد نظر [۶] و آماده‌بودن رم و جواب [۷] را هم داریم. حال در این ماژول، با فعال‌شدن سیگنال شروع، منتظر می‌مانیم تا سینگال ready حافظه فعال شود. در اصل وجود start و ready به شکل پیاده‌سازی‌شده برای پیاده‌سازی تاخیر حافظه بوده‌است. به محض فعال‌شدن ready، از آدرس مورد نظر، محتوا را خوانده و در data_out خروجی می دهیم. ماژول دوم نیز memory_w است که برای نوشتن در حافظه استفاده می‌شود. توجه‌کنید که در این جا دیگر لازم نیست data_out را خروجی‌دهیم و تنها هنگام فعال‌شدنِ start در حالت نوشتن، صبر می‌کنیم تا سیگنال ready حافظه فعال‌شود و به محض فعال‌شدن در address مورد نظر محتوای مربوطه را می‌نویسیم. توجه‌کنید که تنها تفاوت اساسی با memory_r این است که به جای خروجی data_out یک ورودی data_in داریم که محتوایی که باید در حافظه نوشته‌شود را مشخص می‌کند.

فایل next_byte_gen.v

همان‌طور که می‌دانیم؛ در پردازنده‌هایِ واقعیِ JVM، هنگام خواندن و نوشتن در حافظه، با بایت سر و کار نداریم؛ بلکه برای مثال موقع خواندن یکَ word ۴ بایتی از حافظه خوانده می‌شود. در بسیاری از مواقع این word، شامل چندین بایت است که برای دَسترسی به مواردی مانند opcode یا offset باید این بایت‌ها را جداجدا بخوانیم. برای این منظور ماژول next_byte_gen طراحی شده‌است که یک memory_r را instantiate کرده و در صورتی که هر دو سیگنال start و ready فعال باشند؛ PC که برابر address حافظه‌ی instantiate شده‌است را یک واحد اضافه می‌کند که معادل یک بایت جلورفتن یا رفتن به بایتَ بعدی است و در صورت فعال‌بودنِ reset نیز، مقداری پیش‌فرض را برابر PC قرار خواهد داد.

[2] clk
[3] reset
[4] start
[5] address
[6] data_out
[7] ready

فایل instruction_ram.v

این ماژول نیز کار پیچیده‌ای انجام نمی‌دهد و تنها یک word ۴ بایتی را از ورودی دریافت‌کرده و درون یک حافظه‌ی نوشتنی (memory_w) می‌نویسد. برای این کار کافیست تا هنگام instantiate کردن این حافظه درون ماژول، data_in آن را برابر word خوانده‌شده از ورودی قراردهیم. بدیهی‌است که سایر پارامترها نیز باید به درستی تنظیم شوند.

فایل‌های مربوط به Decoder

دیکُدر طراحی‌شده در این پروژه به صورت چند ماژول Read Only Memory(ROM) طراحی شده‌است. این ROMها عبارتند از:

Address ROM

این ROM یک آدرس به عنوان ورودی گرفته و آدرس بعدی که پس از این آدرس باید به آن برویم را برمیگرداند.

Convert ROM

این ROM یک آدرس را به عنوان ورودی گرفته و به عنوان خروجی ID دستور مربوطه را به ما تحویل می‌دهد.

Instruction ROM

این ROM، ID دستور را گرفته و خود دستور را به ما می دهد. منظور از خروجی‌دادن خود دستور، پیاده‌سازی آن به صورت ۰ و ۱ درون ROM است. توجه‌کنید که برای پیاده‌سازی این ROM، ابتدا دستورات پردازنده را با زبان اسمبلی ARM نوشتیم و سپس به کمک یک قطعه‌کد پایتون به صورت خودکار آن‌ها را به فرمت کدشده ۰ و ۱ که باید درون این ROM نوشته‌شود؛ در می‌آوریم.

توضیحی درباره توالی آدرس‌ها

توجه‌کنید که هنگامی که یک دستور را می خوانیم؛ ابتدا در آدرس مربوط به Opcode آن دستور قرار داریم، اما پس از آن با موارد تعیین شده در Address ROM، به صورت زنجیره‌ای (مانند یک لیست پیوندی) جلورفته و به ترتیب مجموعه عملیات مشخصی را انجام خواهیم داد. (توجه‌کنید که ممکن است یک دستور JVM به چندین دستور ARM تبدیل شود بنابراین باید زنجیره‌ای از دستورات را به ترتیب اجرا کنیم!) توجه کنید که Convert ROM نیز ورودی آدرس را گرفته و یک ID را تحویل Instruction ROM می‌دهد و این ROM وظیفه اجرای دستور را خواهد داشت.

فایل Count ROM

این ROM برای این پیاده‌سازی شده‌است که مشخص‌کند پس از خواندن Opcode یک دستور، چند بایت آینده مربوط به ادامه این دستور خواهد بود. توجه‌کنید که برخی از دستورات ممکن است تنها از یک بایت که همان Opcode است تشکیل‌شده باشند مثلا Pop ولی بسیاری از دستورات هستند که مواردی مانند یک Offset ۲ بایتی یا مشابه آن دارند. بنابراین Count ROM با گرفتن Opcode مشخص می‌کند که دستور مربوطه چند بایت اضافی دارد. توجه‌کنید که در فاز اول پروژه که شامل ٪۴۰ کار

می‌شود؛ مواردی که شامل حداکثر ۲ بایت اضافه باشد را Handle کرده‌ایم و تمامی موارد در فاز نهایی پروژه پیاده‌سازی خواهند شد.

⚠ **توجه مهم** همان‌طور که ذکر شد؛ دستورات ممکن است پس از Opcode تعدادی immediate داشته باشند که پارامترهایی مانند index، varnum یا offset را مشخص کنند. برای راحتی کار، در پیاده‌سازی خود، این پارامترها را درون یکی از ثبات‌های پردازنده ARM می‌ریزیم و به درون استک Push ، ARM می‌کنیم. این کار سبب می‌شود که دیگر نیازی به انجام تغییر در Instruction ROM نباشد و عملا پیاده‌سازی ما به مراتب راحت‌تر خواهد شد.

ماشین حالت [۸]

ابتدا توجه‌کنید که این ماژول یک ورودی Reset دارد که هنگام خاموش‌کردن آن، بلافاصله ثبات‌ها و پارامترهای مربوطه برای آغاز کار set می‌شوند. علاوه بر این، در ماژول State Machine، ماژول next_byte_gen را به‌عنوان رابطی با RAM مربوط به JVM، ماژول Instruction RAM را به عنوان رابطی با RAM مربوط به ARM و هم‌چنین ROM های پیاده‌سازی‌شده در Decoder به‌همراه ROM Count را قرار می‌دهیم. توجه کنید که محتوای این هROMا چگونگی جابه جایی بین هStateا را مشخص خواهد کرد. حال پیرامون استیت‌های State Machine و کاربرد هر یک

Fetch Instruction

این State برای خواندن یک دستور JVM از حافظه JVM طراحی شده‌است. توجه کنید که Waiting تعریف شده در ماژول State Machine برای Handle کردن تاخیر حافظه قرار داده شده‌است. به این شکل که سیگنال Ready حافظه نشان می‌دهد که آیا حافظه برای Fetch کردن دستور آماده شده‌است یا خیر. توجه کنید که قبل از فعال شدن سیگنال Ready، عملیات Fetch صورت نمی‌پذیرد و باید تا فعال‌شدن این سیگنال صبر کنیم. در صورت پایان انتظار برای حافظه، به استیت بعدی خواهیم‌رفت.

Check Wide

در این State بررسی می‌کنیم که دستور مورد نظر Wide است یا خیر. با مطالعه instruction Set کامل JVM مشاهده می‌کنیم که دستوری به نام Wide وجود دارد که به‌عنوان پیشوند قبل از برخی دستورها می‌آید و سبب می‌شود که index دستورات به جای ۱ بایت، ۲ بایت باشد. البته در دستورات ما چنین دستوری وجود ندارد اما با توجه به این که می‌خواستیم این مورد نیز در آینده بر روی این State Machine قابل پیاده‌سازی باشد؛ از یک ثبات در این استیت استفاده کردیم که در صورت Wide بودن دستور مقدار آن را یک می‌کنیم. پس از بررسی Wide بودن یا نبودن دستور، به State بعدی خواهیم رفت.

Read Counter

این دستور به کمک ROM Count بررسی می‌کند که تعداد بیت‌های مربوط به پارامترهای پس از Opcode دستور مورد نظر چند تاست. توجه‌کنید که برای افزایش سرعت کار از آنجایی که تعداد دستوراتی که اصلا پارامتری ندارند بسیار زیاد است، اگر این تعداد صفر بود،(مثلا در دستوراتی مانند DUP) بلافاصله به State بعدی خواهیم رفت. در غیر این صورت متغیر byte_params را برابر تعداد بایت‌ها قرار داده و در State بعدی به سراغ Fetch کردن پارامترهای مربوط به دستور از حافظه خواهیم رفت.

[8] State Machine

Fetch Params

fetch کردن پارامترها از نظر نحوه پیاده‌سازی و شیوه‌ی انجام کار بسیار به Fetch کردن دستورات شبیه است. تنها تفاوت اساسی‌ای که وجود دارد این است که پارامترهای Fetch شده را درون ثباتی به نام Push Register می‌ریزیم. این ثبات جهت مواردی مانند Push کردن پارامترها به درون استک ARM طراحی شده‌است که در صفحات قبلی توضیح دادیم.

Push to Stack

این استیت دو مرحله دارد، ابتدا باید immediateی که می‌خواهد به درون استک Push شود را Load کنیم و در مرحله دوم آن را به درون استک Push کنیم. این موضوع که در کدام مرحله قرار داریم را نیز با یک بیت کنترل می‌کنیم. این بیت push_state نام دارد. توجه کنید که چنانچه push_state برابر صفر باشد؛ باید عملیات Load کردن immediate را انجام دهیم و پس از پایان آن push_state را یک خواهیم‌کرد و دوباره به همین استیت می‌رویم. در بازگشت به همین State چون push_state برابر با ۱ است؛ عملیات push کردن مقدارِ لود شده به درون استک را انجام خواهیم‌داد.

Read Next

به کمک Address ROM در این State، آدرس بعدی را به دست می‌آوریم. توجه‌کنید که وقتی می‌خواهیم دستور بعدی را Fetch کنیم؛ باید پیش از آن به آدرس مربوطه رفته‌باشیم که این آدرس از روی Address ROM تعیین خواهدشد.

توجه کنید که برخی از ماژول‌های پیاده‌سازی شده در بالا، هنوز test نشده‌اند و به عبارتی Testbench آن‌ها نوشته نشده‌است. نوشتن این Testbenchها و تست ماژول‌ها در Integration نهایی را به ٪۶۰ پایانی پروژه موکول کرده‌ایم.

مراجع

- داک JVM در سایت ORACLE موجود در

JSR-000924 Java® Virtual Machine Specification