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:

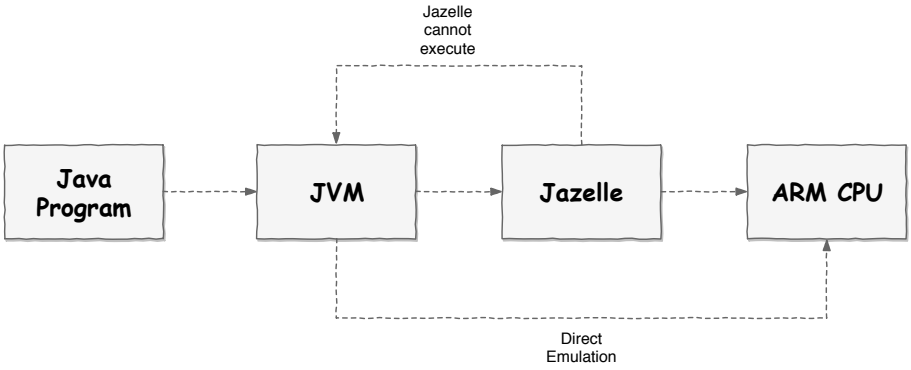
- یادگیری کار با ماشین JVM
- یادگیری کار با ماشین ARM-7
- ساخت مبدل برای تبدیل دستورات میان این دو

[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 ماژول‌های پیاده‌سازی‌شده با verilog

فایل‌های memory

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 خواندنِ 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.

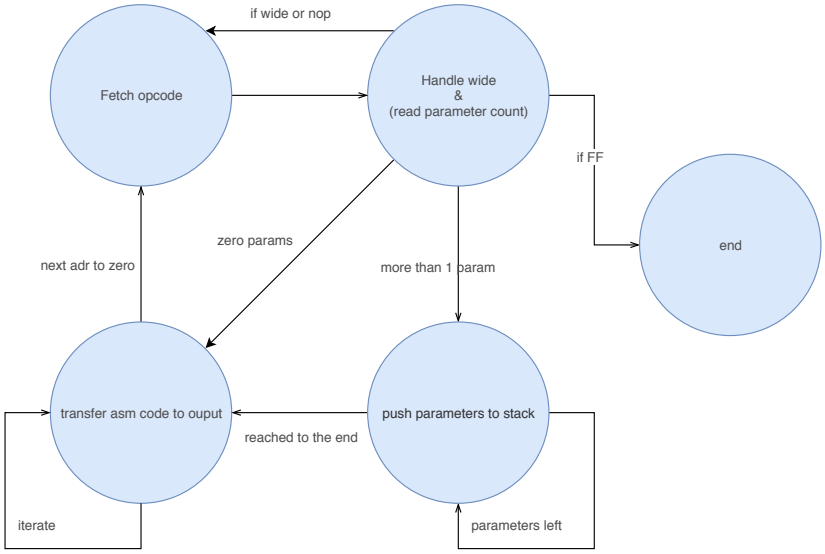
فایل Count ROM

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.

⚠ **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]

حالات ماشین حالت را می‌توانید در شکل زیر مشاهده نمایید:



شکل ۲: ماشین حالت

FETCH_INSTRUCTION

در این استیت، آپکد دستور JVM، از رم مربوط به آن خوانده می‌شود. و بعد برای بررسی نوع دستور و گرفتن تعداد پارامترهای آن، به استیت (۲) می‌رویم.

CHECK_WIDE_AND_READ_COUNTER

با بررسی آپکد دستور که در مرحله‌ی قبل خوانده‌ایم، در این‌جا با بررسی نوع دستور، به یکی از استیت‌های زیر می‌رویم:

[8] State Machine

END

برای پایان برنامه از یک بایت که در دستورات JVM نیست [9] مانند 0xFF استفاده کرده‌ایم. در صورتی که آپکد برابر این کلمه باشد، برنامه‌ی داخل رم JVM تمام شده‌است و کار این ماژول به پایان می‌رسد.

ITERATE

با استفاده از count_rom که همان‌طور که قبلاً توضیح داده شده‌است، ماژولی است که با توجه به آپکد دستورات ،JVM تعداد بایت پارامترهایی که بعد از آن آپکد قرار می‌گیرند را مشخص می‌کند ــ می‌توان تعداد پارامترهای هر آپکد را در این‌جا داشت. بنابراین اگر پارامتری پس از این آپکد نداشته باشیم، مستقیماً به این استیت می‌رویم.

تست‌بنچ‌های مربوط به ماژول‌های ساخته‌شده

در نهایت پس از ساخت تمامی این ماژول‌ها تست‌بنچ‌هایی نوشتیم و به کمك آن‌ها، پیش از Integration از صحت عملکرد آن‌ها، اطمینان حاصل کردیم. این تست‌بنچ‌ها شامل تست‌بنچ‌های زیر هستند:

- **next_byte_gen_tb**: که از آن برای بررسی صحت ماژول‌های memory_r و next_byte_gen استفاده می‌کنیم.
- **read_count_tb**: که از آن برای بررسی صحت ماژول‌های read_count و countr_rom استفاده می‌کنیم.
- **write_tb**: که از آن برای بررسی صحت ماژول‌های memory_w و write استفاده می‌کنیم.

پیاده‌سازی دستورات پردازنده به‌کمک اسمبلی ARM

در این مرحله تمامی دستورات پردازنده را با کمک زبان اسمبلی ARM پیاده‌سازی کردیم. این دستورات شامل موارد زیر می‌شوند:

دستورات ستون دوم شامل کار با استک و اعداد صحیح

برای پیاده‌سازی دستورات dup و مشابه آن‌ها تنها از دو دستور push و pop استفاده شده‌است. به این شکل که به‌ترتیب ذکرشده در مراجع، موارد مربوطه را push و pop می‌کنیم.
دستورات pop و pop2 نیز با استفاده از یک خط دستور pop قابل پیاده‌سازی هستند و برای دستور swap نیز مشابه دستورات قبلی عمل می‌کنیم.

دستورات سه ستون دیگر شامل دستورات کار با float و double

برای کار با اعداد floating point در اسمبلی ARM دستورات مشخصی وجود دارد اما این دستورات توسط coprocessor اجرا می‌شوند. برای این که به coprocessor متصل شویم، از چند خط کد در ابتدای فایل s. ارسالی استفاده کرده‌ایم. این خطوط شامل فعال کردن مواردی مانند fpu و اتصال به coprocessor می‌شود.

[9] reserved word

چالش‌های اجرای کدهای اسمبلی نیازمند fpu

در ابتدا می‌خواستیم شبیه‌سازی دستورات اسمبلی را به کمک Keil انجام‌دهیم، اما پس از اندکی تلاش مشاهده‌کردیم که پردازنده‌های دارای fpu، از خانواده‌ی cortex-m هستند که تنها دستورات Thumb را پشتیبانی می‌کنند اما ما می‌خواستیم که از دستورات ۳۲بیتی ARM در این پروژه بهره‌ببریم. . بنابراین تصمیم‌گرفتیم که در این بخش از نرم‌افزار DS-5 استفاده‌کنیم. پس از نصب نرم‌افزار DS-5 ابتدا سعی‌کردیم تا به کمک Fast Model شبیه‌سازی را انجام دهیم و با جست‌وجو میان Fast Modelها متوجه‌شدیم که باید از پردازنده‌های خانواده‌ی Cortex-A استفاده‌کنیم و در بین این پردازنده‌ها ، پردازنده‌ی Cortex-A7 را انتخاب‌کردیم زیرا این پردازنده هم دارای fpu است و هم از دستورات ARM پشتیبانی می‌کند.

در کامپایلر مربوطه، تنظیمات را به گونه‌ای انجام دادیم تا به جای استفاده از عملیات اعشاری soft-fp، از hard-fp استفاده‌کند که در این‌جا، به این مشکل برخوردیم که کد ما وارد Trapهای CPU می‌شد.

بنابراین پس از کمی جست‌وجو در منابع مختلف، متوجه‌شدیم که ابتدا باید fpu را فعال‌کنیم و بعد در ادامه به سراغ کدهای مربوطه برویم زیرا در غیر این صورت دچار مشکل خواهیم‌شد. لذا در ابتدای کدهای اسمبلی، کدی نوشتیم که fpu را فعال‌کند و به کمک آن کد، می‌توانستیم که دستورات مربوط به fp را اجرا کنیم.

کد اسمبلی مذکور به شکل زیر است:

```
area start, code
export StartHere
StartHere
MRC p15, 0, r0, c1, c1, 2
ORR r0, r0, #2_11<<10 ; enable fpu
MCR p15, 0, r0, c1, c1, 2
LDR r0, =(0xF << 20)
MCR p15, 0, r0, c1, c0, 2
MOV r3, #0x40000000
VMSR FPEXC, r3
import __main
b __main
```

حال در هر مرحله پس از اجرای هر قطعه کد مربوطه، برای بررسی درستی آن، ثبات‌ها را pop می‌کردیم و مشاهده می‌کردیم که آیا نتیجه دلخواه در درون آن‌ها ذخیره‌شده‌است یا خیر.

پیاده‌سازی دستورات

حال به نحوه‌ی پیاده‌سازی دستورات این بخش می‌پردازیم:

- **دستورات fconst و dconst**

برای پیاده‌سازی این دستورات تنها کافیست عدد مربوطه را به درون یک ثبات mov کرده و آن را به درون استک push کنیم. توجه‌کنید که برای دستورات floating point پیش از هر دستور باید کاراکتر v را قرار دهیم و برای اعداد float از f32. و برای اعداد double از .f64 استفاده‌کنیم. نکته دیگر پیاده‌سازی این دستورات این است که مقدار صفر را نمی‌توانیم به درون یک ثبات مشخص movکنیم و برای این کار از عملیات sub و کم‌کردن مقدار یک ثبات از خودش برای تولید عدد صفر استفاده کرده‌ایم.

- **دستورات ضرب و جمع و تفریق و تقسیم**

برای چنین عملیاتی در دستوراتی مانند dsub یا fdiv، ابتدا دو مقدار را از استک pop کرده و سپس عملیات مربوطه را انجام می‌دهیم و حاصل را به درون استک push می‌کنیم. این دستورات نکته خاصی ندارند و تمامی عملیات ضرب، جمع، تفریق یا تقسیم توسط coprocessor انجام می‌شوند.

- **دستورات مربوط به compare**

برای ۴ دستور مربوط به compare ابتدا دو عدد را با vpop از استک pop کرده و با vcmp مقایسه می‌کنیم. توجه کنید که باید به مقایسه‌کننده مخصوص دستورات floating point متصل شویم و برای این کار از یک خط دستور زیر استفاده می‌شود.

```
VMRS APSR_nzcv, FPSCR
```

پس از آن از سه بلوک مختلف استفاده می‌کنیم: بلوک eq، بلوک gt و بلوک lt که هر یک شامل دو خط کد است و عملکرد برنامه را در صورت تساوی، بزرگ‌تر یا کوچک‌تر بودن مقایسه تعیین خواهد کرد.

- **دستورات load و store**

در دستورات daload و faload ابتدا دو عدد از استک می‌خوانیم که تعیین‌کننده محل خواندن از حافظه است. یکی از آن‌ها را به‌عنوان مبدا گرفته و دیگری را در ۴ ضرب کرده و به آن می‌افزاییم تا محل خواندن عدد مربوطه به‌دست آید. سپس از محل به‌دست‌آمده در حافظه یکی از ثبات‌ها را loadکرده و حاصل را به درون استک pushمی‌کنیم. در دستورات store نیز کار مشابه است با این تفاوت که عددی که باید ذخیره‌شود را در ابتدا از استک خوانده و پس از آن دو عدد دیگر می‌خوانیم که به شکل ذکرشده در بالا، محل ذخیره‌سازی در حافظه را مشخص می‌کنند. در نهایت عددی که در ابتدای کار خوانده‌شد را در حافظه ذخیره می‌کنیم.

- **دستورات frem و fneg**

دستور fneg بسیار ساده است. یک عدد را از درون استک pop کرده و با دستور vneg.f32 منفی کرده و در نهایت به‌درون استک push می‌کنیم. دستور frem نیز طبق فرمول ذکرشده در مراجع پیاده‌سازی شده‌است اما توجه کنید که هنگام انجام تقسیم اول که مربوط به اعداد floating point است؛ باید حاصل را به یک integer تبدیل کنیم. برای این امر از دستور vcvt استفاده می‌کنیم که بعد از آن باید فرمت مبدا و فرمت مقصد را بنویسیم. برای مثال برای تبدیل حاصل floating point تقسیم به عدد صحیح از دستور زیر بهره می‌گیریم که s32 نشان‌دهنده فرمت مقصد و f32 نشان‌دهنده فرمت مبدا خواهد بود.

```
vcvt.s32.f32 s2,s2
```

- **دستورات convert ساده**

دستوراتی مانند d2f و f2i و f2d و d2i دستورات convert ساده هستند. رای این دستورات تنها کافی‌ست از استک عدد مورد نظر را بخوانیم، به‌کمک دستور vcvt توضیح داده‌شده در بالا آن را به فرمت مورد نظر تبدیل کرده و در نهایت حاصل را به درون استک push کنیم.

- **دو دستور پیچیده تر convert**

برای پیاده‌سازی دو دستور f2l و d2l به این شکل عمل می‌کنیم که از دو خط کد آماده یافت‌شده در اینترنت برای انجام این تبدیل استفاده می‌کنیم که این دو خط کد به شکل زیر خواهد بود:

```
import __aeabi_d2lz
bl __aeabi_d2lz
```

بدیهی است که در دستورات مربوط به float به جای d از f استفاده می‌شود. سایر خطوط کد نیز نکته جدیدی ندارد.

- **دستورات fload**

برای پیاده‌سازی دستور fload، ابتدا یک عدد را از استک می‌خوانیم و به‌کمک شیفت چپ آن را در ۴ ضرب می‌کنیم. سپس حاصل را با frame pointer یا همان fp جمع‌کرده تا محل خواندن از حافظه به‌دست آید. سپس از محل مربوطه در حافظه خوانده و در یک ثبات ذخیره می‌کنیم و در نهایت حاصل را به درون استک، push می‌کنیم.

برای دستورات fload_n نیز فرمت کلی زیر را داریم:

```
vldr.f32 s0,[fp, #n*4]
vpush.f32 {s0}
```

که به این معناست که از fp به اندازه ۴ برابر n جلو می‌رویم و محتوا را از محل حافظه می‌خوانیم و در نهایت محتوای خوانده‌شده را به درون استک push می‌کنیم.

اضافه‌کردن کدهای اسمبلی به Instruction ROM

حال دستورات پیاده‌سازی‌شده را هر یک به فرمت زیر می‌نویسیم:

```
;#<instr_name> <ARM Code>
```

و هدف این است که دستورات نوشته‌شده با این فرمت را به کد پایتونی که در بالا ذکر شد بدهیم تا Instruction ROM را به صورت خودکار برای ما تولید کند.

کد پایتون و Assembler

در کد پایتون ذکرشده، هر بار Assembler را صدا می‌کنیم و رشته‌های باینری مربوطه را به کمک اجرای Assembler از آن استخراج می‌کنیم تا برای ROM استفاده‌شود.

توجه‌کنید که در فاز نهایی پروژه کد پایتون مذکور را بهینه‌کرده‌ایم،سس بنابراین این کد علاوه بر این که محتوای ROM را همان‌طور که ذکر شد، تولید می‌کند؛ اگر هر کدی به آن به عنوان ورودی بدهیم، خیلی سریع کامپایل خواهد کرد و تست‌کیس مربوطه را برای ما تولید خواهدکرد. این بهینه‌سازی و فرآیند کامپایل سریع، سبب گرفتن خروجی در کمترین زمان ممکن خواهدشد و به‌وضوح کار بسیار مفیدی خواهد بود.

سنتزکردن کدها

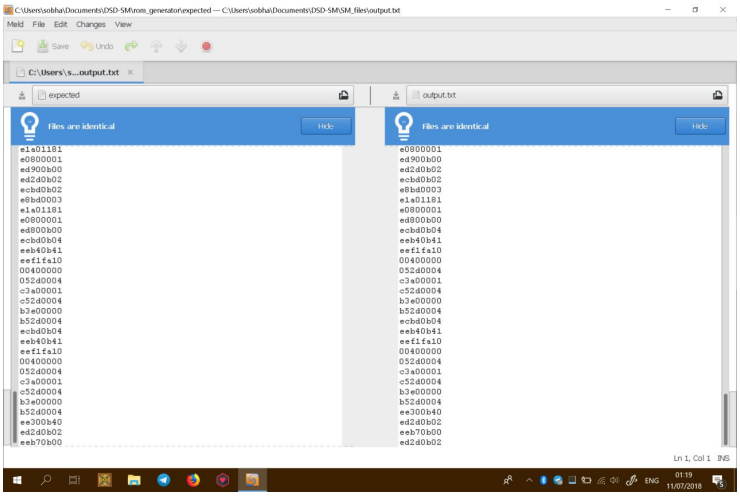
پس از این که به کمک دستورات نوشته‌شده با اسمبلی ARM و کد پایتون ذکرشده، ROMای که در قسمت‌های قبل توضیح‌داده‌شد را پرکردیم؛ مجموعه کدهای وریلاگمان کامل خواهدشد و حال باید این کدها را سنتزکنیم.

برای سنتز کدهای وریلاگ از نرم افزار Quartus محصول شرکت Altera استفاده‌کردیم و برای این کار در کوارتوس یک پروژه ساخته و تمامی فایل‌هایی که می‌خواهیم سنتزکنیم (تقریبا تمامی فایل‌ها به جز تست‌بنچ‌ها) را به آن پروژه اضافه‌نمودیم.

سپس با چندین بار تلاش برای سنتز کدها ، اشکالات موجود و بخش‌های غیرِ سنتزپذیر کد را از آن حذف‌کردیم و با قطعه کدهای سنتزپذیر جایگزین‌کردیم تا در نهایت سنتز ماژول اصلی یعنی ماژول top که در اصل مربوط به Integration تمامی مواردی بود که پیاده‌سازی کرده‌ایم، انجام‌شد.

برای راحت‌ترشدن اجرا و تست و import‌کردن فایل‌های سنتزشده به نرم‌افزار مادل‌سیم، از یک اسکریپت استفاده‌کردیم که با گرفتن ورودی، خود به خود نرم‌افزار مادل‌سیم را بازکرده و شبیه‌سازی را انجام‌داده و نتیجه را به ما ارائه می‌دهد. این اسکریپت سبب شد که سرعت تست کردن کدها و نتایج بسیار بالا رود و بتوانیم سریع‌تر مشکلات کدها را یافته و آن ها را برطرف کنیم.

در نهایت پس از رفع اشکالات کد مذکور، بالاخره خروجی تولیدشده توسط کد با خروجی مورد انتظار ما مشابه‌شد که شکل آن را در شکل ۳ می‌کنید.



شکل ۳: نتیجه‌ی سنتز

مشارکت اعضا در پروژه

کدهای اسمبلی: کوشا جافریان و روژین نوبهاری
کدهای وریلاگ: مهبد مجید و کیمیا حمیدیه
اشکال زدایی کدهای وریلاگ و سنتز آن: کوشا جافریان
کامنت گذاری کدهای اسمبلی: روژین نوبهاری
کامنت‌گذاری کدهای وریلاگ: کیمیا حمیدیه

برنامه‌ریزی پروژه و نظارت و کمک در تمامی بخش‌ها + تست نهایی پروژه: سبحان محمدپور
محتوای داک فاز نهایی پروژه: کوشا جافریان
تنظیم و طراحی داک پروژه: مهبد مجید و کیمیا حمیدیه

مراجع

- داک JVM در سایت ORACLE موجود در

JSR-000924 Java® Virtual Machine Specification