فاز اول پروژه طراحی همروند

مقدمه

در فاز و در وهله ی اول قصد تولید یک کلمه ی ۳۲ بیتی در فرمت ریز دستورالعمل[1]، از هر خط دستوری که از فایل ورودی دریافت می شود را خواهیم داشت که طبق استاندارد ها و معماری (Microprogrammed Architecture) از پیش تعیین شده در کتاب مرجع [1] معرفی می شوند و سپس کلمات را در یک فایل obj استخراج می کنیم.

در این فاز از نسخه ۳ پایتون[2] استفاده شده است. در ادامه جداول پیش فرض را درج خواهیم نمود:

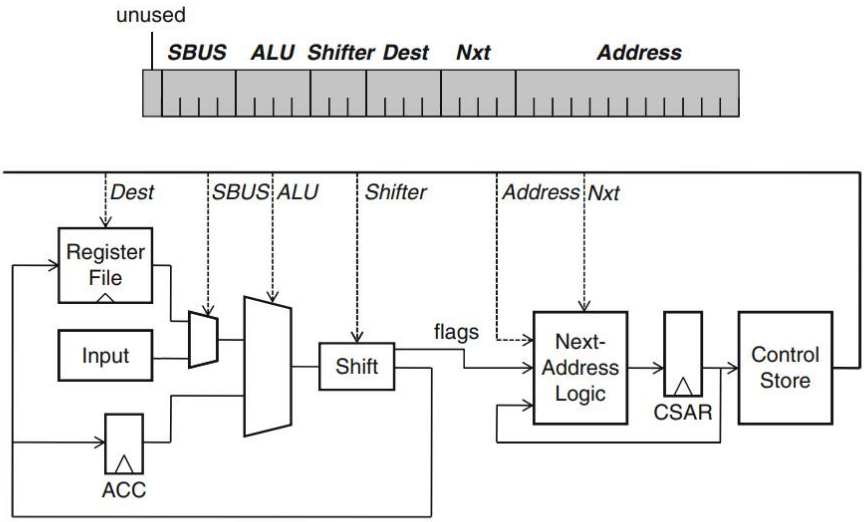

Fig. 5.7 A microprogrammed datapath

[1] Micro Instruction
[2] Python V3

Table 5.1 Microinstruction encoding of the example machine

Field	Width	Encoding			
SBUS	4	Selects the operand that will drive the S-Bus			
		0000	R0	0101	R5
		0001	R1	0110	R6
		0010	R2	0111	R7
		0011	R3	1000	Input
		0100	R4	1001	Address/Constant
ALU	4	Selects the operation performed by the ALU			
		0000	ACC	0110	ACC — S-Bus
		0001	S-Bus	0111	not S-Bus
		0010	ACC + SBus	1000	ACC + 1
		0011	ACC − SBus	1001	SBus − 1
		0100	SBus − ACC	1010	0
		0101	ACC & S-Bus	1011	1
Shifter	3	Selects the function of the programmable shifter			
		000	logical SHL(ALU)	100	arith SHL(ALU)
		001	logical SHR(ALU)	101	arith SHR(ALU)
		010	rotate left ALU	111	ALU
		011	rotate right ALU		
Dest	4	Selects the target that will store S-Bus			
		0000	R0	0101	R5
		0001	R1	0110	R6
		0010	R2	0111	R7
		0011	R3	1000	ACC
		0100	R4	1111	unconnected
Nxt	4	Selects next-value for CSAR			
		0000	CSAR + 1	1010	cf ? CSAR + 1 : Address
		0001	Address	0100	zf ? Address : CSAR + 1
		0010	cf ? Address : CSAR + 1	1100	zf ? CSAR + 1 : Address

شرح گزارش[]

در حالت کلی در کد ۳ تابع اصلی خواهیم داشت و ۵ بخش را بررسی خواهیم نمود:

- اقدامات اولیه
- تابع "seq_ins_parser"
- تابع "RunCommand"
- تابع "RunJMP"
- فراخوانی توابع و استخراج نتیجه

اقدامات اولیه:

ورودی های در نظر گرفته شده در قالب یک فایل با پسوند txt با نام ('MicroInstruction.txt') در مسیر (./Phase1/io/) ذخیره شده اند و محتوای آن ها دستوراتیست به سبک و سیاق Listing 5.1 کتاب مرجع :

Listing 5.1 Micro-program to evaluate a GCD

```
1  ;    Command Field          || Jump Field
2            IN -> R0
3            IN -> ACC
4  Lcheck:  (R0 - ACC)         || JUMP_IF_Z Ldone
5           (R0 - ACC) << 1    || JUMP_IF_C LSmall
6           R0 - ACC -> R0     || JUMP Lcheck
7  Lsmall:  ACC - R0 -> ACC    || JUMP Lcheck
8  Ldone:                        JUMP Ldone
```

در ابتدا با استفاده از دستوراتی فایل را خوانده و هر خط از آن که یک رشته دستور محسوب می شود را در لیستی با نام (microins) ذخیره می کنیم. سپس متغیر های سراسری را مقدار دهی اولیه می کنیم و یک متغیر از نوع دیکشنری با نام DicTable برای پیاده سازی داده های Table 5.1 معرفی کرده و مقادیر مربوطه را در آن قرار می دهیم.

متغیر های SBUS,ALU,Shifter,Dest,Nxt,Address را نیز برای نگهداری مقادیر باینری کد شده هر خط دستور معرفی کرده و مقدار پیشفرض به آن ها اختصاص می دهیم.

علاوه بر آن با معرفی متغیر (mild) از نوع لیست می خواهیم پس از parse کردن هر خط دستور، بخش های مختلف دستور را در یک دیکشنری قرار داده و آن را به این لیست (mild) اضافه کنیم. در بخش بعدی این عمل بیشتر شرح داده خواهد شد.

تابع seq_ins_parser :

هدف این تابع جدا کردن بخش های یک خط دستور به بخش های زیر است. برای مثال جدول زیر بخش های مختلف یک خط دستور را نشان خواهد داد:

آدرس	لیبل	دستور اصلی	شرط پرش به همراه لیبل پرش
7	LSmall	ACC - R6 -> ACC	JUMP Lcheck
7	LSmall :	ACC - R6 -> ACC ||	JUMP Lcheck

در نهایت پس از جدا سازی بخش های مختلف هر کدام در متغیری ذخیره شده اند:

- آدرس (tmpaddress)
- لیبل (tmplabel)
- دستور اصلی (tmpcommandfield)
- شرط پرش به همراه لیبل پرش (tmpjumpfield)

و طبق توضیحاتی که در بخش قبل ذکر شد این بخش ها را در یک متغیر از نوع دیکشنری و با نام (midic) ذخیره کردیم و هر یک از midic ها را در هر خط به لیستی با نام (mild) اضافه و در نهایت این لیست را return کردیم.

تابع RunCommand :

دستور ها را در نمای کلی می توان به دو دسته تقسیم نمود:

- **دستوراتی که پرش شرطی دارند (یعنی Assignment ندارند):**
  این دسته با فلگ های zf و cf سر و کار دارند و حداکثر باید عملیات محاسباتی و شیفت در این دسته انجام شود.
- **دستوراتی که یا پرشی ندارند یا پرش غیر شرطی دارند:**
  این دسته پس از انجام محاسبات و شیفت باید حاصل را در رجیستری ذخیره کرده و با فلگ ها کاری ندارد.

حال می توان دریافت، دسته ی اول (Dest) ثابتی دارند و مقدار 1111(unconnected) برای آن ها مناسب است.

تفاوت این دو دسته را می توان در وجود کاراکتر انتسابی "<-" آشکار نمود. به عبارتی اگر این رشته در دستور وجود داشت در دسته دوم و در غیر این صورت در دسته اول جای می گیرند.

با توجه به توضیحات بالا از هر خط کد بخش مربوط به دستور را جدا می کنیم و آن را در متغیر (CurrentCommand) ذخیر کرده و به شرح زیر عمل می کنیم:

CurrentCommand		
LoS	->	RoS

(if): با استفاده از تابع split() به دنبال "<-" می گردیم و سمت چپ آن را در LoS و سمت راست آن را در RoS ذخیره می کنیم.

- در این صورت در RoS باید (R0…R7 و یا ACC) وجود داشته باشد. پس هر کدام از آن ها بود فیلد Dest برابر با آن می شود. برای این کار محتویات RoS را در دیکشنری DicTable جستجو می کنیم و مقدار باینری آن را ر فیلد Dest قرار می دهیم.
-

(else): حال اگر این کاراکتر وجود نداشت یعنی هیچ Assignment ای در این دستور وجود ندارد، پس تمام محتویات CurrentCommand را در LoS ذخیره می کنیم.

- در این صورت همانطور که گفته شد فیلد (Dest) مقدار ثابتی دارد و 1111(unconnected) برای آن مناسب است.

*** توجه شود برای تمام مقادیر رشته ای با استفاده از تابع strip() کاراکتر های space اضافه را حذف کردیم.**

*** تا به اینجا تکلیف فیلد Dest مشخص شد.**

حال محتویات LoS چیزی شبیه به موارد زیر خواهد شد:

- IN
- Constant
- (R5 - ACC) <<1
- (R6 - ACC)
- ACC – R2

(if): حال اگر در LoS کاراکتر ")" موجود بود:

یعنی یک عملگر و دو عملوند درون پرانتز داریم و تا زمانی که به کاراکتر "(" برسیم باید رشته را پیمایش کنیم و عبارت درون پرانتز را در متغیر tmpAlu قرار داده و ممکن است در شرایطی بعد از پرانتز عملگر شیفت داشته باشیم پس:

(if): اگر کاراکتر ">>" بعد از پرانتز نمایان شد فلگی با نام LSF را برابر True قرار می دهیم.

(if): اگر کاراکتر "<<" بعد از پرانتز نمایان شد فلگی با نام RSF را برابر True قرار می دهیم.

توجه شود مقادیر پیشفرض هر دو فلگ False است.

(else): حال اگر پرانتزی وجود نداشت تمام محتویات LoS را به عنوان یک operator (عملیات) یا یک عملوند تکی درون متغیر tmpAlu قرار می دهیم.

حال برای متغیر tmpAlu سه حالت بوجود می آید:

	tmpAlu		
حالت اول	**not**	**Op1**	**Op2**
حالت دوم	**IN , R0.....R7**		
حالت سوم	**ACC**		
حالت چهارم	**Op1**	**>>,<<**	**Op2**
حالت پنجم	**Op1**	**- , + , &**	**Op2**

در حالت اول و دوم و سوم بدون مشکلی مقادیر باینری ALU و SBUS بدست می آیند و برای حالت چهارم همانطور ذکر شده بود فلگ های مربوط به شیفت را ست می کنیم **و اما در حالت پنجم**:

ابتدا باید مشخص کنیم کدام یک از op1,op2 ACC است و کدام یک رجیستری از Sbus است و با علم بر این موضوع مقدار ALU و SBUS را مشخص می کنیم.

*** تا به اینجا تکلیف فیلد ALU و SBUS مشخص شد.**

حال با توجه به مقادیر فلگ های شیفت فیلد Shifter را مشخص می کنیم:

(if): اگر LSF فعال بود مقدار باینری مربوط به شیفت منطقی چپ (۰۰۰) را در فیلد Shifter قرار می دهیم.

(if): اگر RSF فعال بود مقدار باینری مربوط به شیفت منطقی راست (۰۰۱) را در فیلد Shifter قرار می دهیم.

(if): اگر LSF و RSF هر دو **غیرفعال** بودند مقدار باینری مربوط به عملیات بدون شیفت (۱۱۱) را در فیلد Shifter قرار می دهیم.

*** تا به اینجا تکلیف فیلد Shifter مشخص شد.**

تابع Runjump :

ابتدا بخش مربوط به شرط پرش و لیبل در هر خط دستور را در متغیر CurrentJump قرار داده و قسمت اول را از قسمت دوم جدا می کنیم.

قسمت اول یکی از موارد زیر است که هر کدام اگر باشد در دیکشنری DicTable مقدار باینری معادل را برای فیلد Nxt قرار می دهیم:

- JUMP
- JUMP_IF_Z
- JUMP_IF_C
- JUMP_IF_NZ
- JUMP_IF_NC

در صورتی که CurrentJump یک رشته خالی بدست آید آنگاه یعنی پرش نداریم و مقدار باینری عدم پرش را در فیلد Nxt قرار می دهیم.

حال اگر CurrentJump خالی نباشد قسمت دوم حاوی محتویات لیبل مربوطه است. باید فیلد Address را به گونه ای مقدار دهی کنیم و باید دریابیم که این لیبل به چه آدرسی اشاره می کند. برای این کار در تک تک عناصر لیست mild جستجو را انجام می دهیم و آدرس مربوطه را بدست آورده و در فیلد Address ذخیره می کنیم.

*** تا به اینجا تکلیف فیلد Nxt و Address مشخص شد.**

فراخوانی توابع:

حال در حلقه ای هر بار یک عنصر از لیست mild را به عنوان یک خط دستور به توابع RunCommand و RunJump ارسال می کنیم و پس از اینکه مقادیر تمام فیلد ها بدست آمد آن ها را در کنار هم قرار داده و یک کلمه ۳۲ بیتی تولید می کنیم.

استخراج نتیجه در فایل :

در مد bw (binary write) با فایل ها کار می کنیم و فایلی به نام EndFile.txt در مسیر (./Phase1/io/) ایجاد می کنیم تا محتویات خروجی را در آن ذخیره کنیم.

مشکلی که در حین انجام این عمل بوجود آمد این بود که داده ها در این مد حداکثر در بلوک های هشت بیتی یا همان یک بایتی می توانند نوشته شوند در صورتی که هر word که ایجاد خواهد شد ۳۲ بیتی است.

پس هر word را به چهار بایت که هر کدام هشت بیت هستند تقسیم نموده و بایت به بایت در فایل نوشتیم.

حال نتایج تولید کد برای دو خط تصادفی از دستورات وروردی را خواهیم دید:

```
_____________________________________________________________________________________________
Command Number 3 ---            5 (R4 - ACC) << 1 || JUMP_IF_C LSmall

This Line saved in Dictionary:  {'address': '5', 'label': '', 'command': '(R4 - ACC) << 1', 'jmp': ' JUMP_IF_C LSmall\n'}

The Command Successfully Encoded!

SBUS: 0b100             ALU: 0b100              Shifter: 0b0            Dest: 0b1111            Nxt: 0b10               Address: 0b111

Generated 32 bit Word is :  0  0100  0100  000  1111  0010  000000000111
_____________________________________________________________________________________________
Command Number 4 ---            6 R0 - ACC -> R0 || JUMP Lcheck

This Line saved in Dictionary:  {'address': '6', 'label': '', 'command': 'R0 - ACC -> R0', 'jmp': ' JUMP Lcheck\n'}

The Command Successfully Encoded!

SBUS: 0b0               ALU: 0b100              Shifter: 0b0            Dest: 0b0               Nxt: 0b1                Address: 0b100

Generated 32 bit Word is :  0  0000  0100  000  0000  0001  000000000100
```

و در پایان فرمت ذخیره سازی دستورات در فایل EndFile.txt به صورت باینری را خواهیم دید:

```
That's Binary format saved in EndFile.txt :      b'@\xf0\x06P@\xf8\x06P\x02\x7f@\x08"\x0f \x07\x02\x00\x10\x041\x88\x10\x041\x8f\x10\x08'
All Done!
```

مراجع و ماخذ:

1. Patrick R.Schaumont , A Practical Introduction to Hardware/Software Codesign