به نام او

گزارش پروژه درس معماری کامپیوتر 99-98

اعضای گروه :

محمد رضا عبدی 97110285

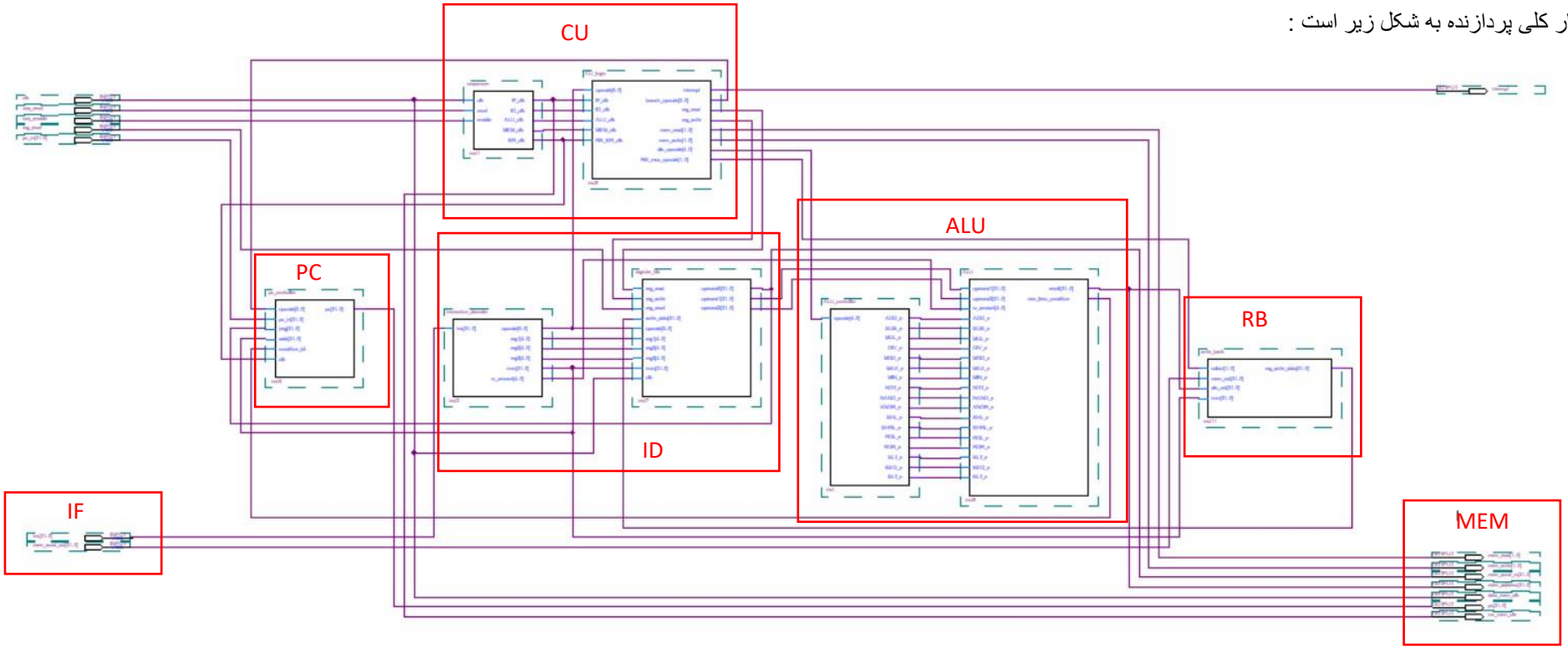
امیرحسین عباسی 97102044

امیرحسین حسن پور 97103208

علی شفیعی 97110122

1) CPU :

ساختار کلی پردازنده به شکل زیر است :



CU :

این ساختار از دو بخش sequencer و logic تشکیل شده است :

Sequencer : این بخش یک one-hot counter میباشد که به ترتیب لبه های فعالی که در خروجی خود میدهد, به بخش logic اعلام میکند که الان در چه stage ای از اجرای دستور (instruction fetch , instruction decode , arithmetic unit , memory read/write و write back) قرار داریم و بخش logic به واسطه آن کار مورد نظر را انجام میدهد.

** این مدار یک reset آسنکرون و یک enable حساس به سطح مثبت دارد.

** کد وریلاگ این بخش در sequencer.v قرار دارد.

LOGIC : این بخش با توجه به لبه های مثبت دریافت شده از sequencer متوجه میشود که در چه stage ای قرار دارد و چه اتفاقی باید در cpu رخ بدهد.

** کد وریلاگ این بخش در CU_logic.v قرار دارد.

** معماری پردازنده به صورت Multi-cycle بوده و با توجه به سیگنال های لبه مثبت در هر خروجی sequencer کار مورد نظر را انجام میدهد.

IF : در این مرحله دستور از memory خوانده میشود و وارد ماژول instruction_decode میشود.

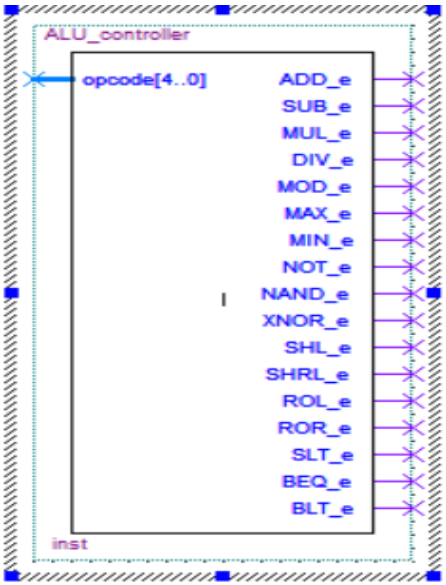
ID : در این مرحله ماژول instruction_decode دستور داده شده را میشکند و مقادیری مانند opcode , اندیس رجیستر های درگیر در دستور , immediate data که به 32 بیت sign extend شده و sr_amount که برای دستورات شیف کاربرد دارد را از دستور استخراج کرده و به CU میدهد, ضمنا اندیس رجیستر ها و دیتای immediate را هم به register file میدهد. (کد وریلاگ در instruction_decoder.v قرار دارد.)

** در این مرحله register file با توجه با اندیس هایی که از ماژول instruction_decode دریافت میکند , اگر دستور نیاز به مقدار رجیستر داشته باشد CU پایه read آن را فعال میکند و مقادیر متناظر با اندیس ها و یا دیتای immediate از Register file خارج خواهند شد. (کد وریلاگ در register_file.v قرار دارد.)

** در ماژول register file ترتیب خروجی به ترتیب اندیس های رجیستر داده شده در دستور است. در دستوراتی که immediate data یا immediate address دارند مقدار operand2 برابر با آن دیتا و در دستوراتی که تنها اندیس register ورودی میگیرند, operand2 برابر با مقدار موجود در اندیس آخرین رجیستر خواهد بود.

ALU : در این مرحله اگر دستور به عملیات محاسباتی نیاز داشته باشد, با توجه با opcode فرستاده شده از سوی CU مقادیر operand1 و operand2 حاصل از register file و sr_amount حاصل از ماژول instruction_decoder یک عملیات محاسباتی را انجام میدهد.

** ماژول alu controller در این قسمت opcode ورودی را دریافت می کند و سپس خروجی به صورت one hot می دهد که به clk enable مدار های موجود در alu متصل می شود : (کد وریلاگ موجود در فایل ALU_controller.v)



قسمت محاسبات منطقی (ALU)

با توجه به قسمت قبل از ALU که alu_controller است، سیم‌های فعالسازی از آن قسمت به قسمت ALU می‌آیند. ورودی ALU علاوه بر سیم‌های فعالسازی، سیم‌های Operand1,2 و Sr_Amount هستند. این ورودی ها حاوی مقادیر register و Immediate می‌باشند.

فایل‌های این قسمت در پوشه alu در قسمت اصلی پروژه قرار دارند. مدار کلی alu در همان ابتدای پوشه alu و هر کدام از مدارهای جمع و ضرب و ... در پوشه /main/format1,4 قرار دارند. پوشه block هم برای symbol هاست.

علت اینکه دستورات تنها در دو فرمت پیاده سازی شده اند این است که با همین دو نوع دستور میتوانستیم دستورات فرمت دوم و سوم را پیاده کنیم. مثلاً برای دستورات immediate، در مرحله قبل از ALU، مقادیر immediate تبدیل به کلمه 32 بیتی می شدند و از طریق operand ها به ALU می آمدند و دیگر لازم نبود که یکبار دیگر عملیات های جمع و ضرب و ... برای آنها به صورت جداگانه پیاده سازی شود. برای فرمت سوم نیز تنها نیاز بود که index مورد نیاز تولید شود و در مراحل بعدی مورد استفاده قرار گیرد. این مقدار نیز با جمع کردن operand ها قابل انجام بود و نیاز به مدار جداگانه نداشت. همچنین در فرمت چهارم نیز از 5 دستور تنها دو دستور مدار میخواست و باقی دیگر نیاز به مدار نداشتند.

مدار های هر کدام از عملیات ها به صورت جداگانه پیاده سازی شده و برخی نیز با هم ادغام شدند؛ برای مثال جمع و ضرب را با یک مدار هم میتوان انجام داد و شیفت ها و چرخش های چپ و راست نیز همینطور.

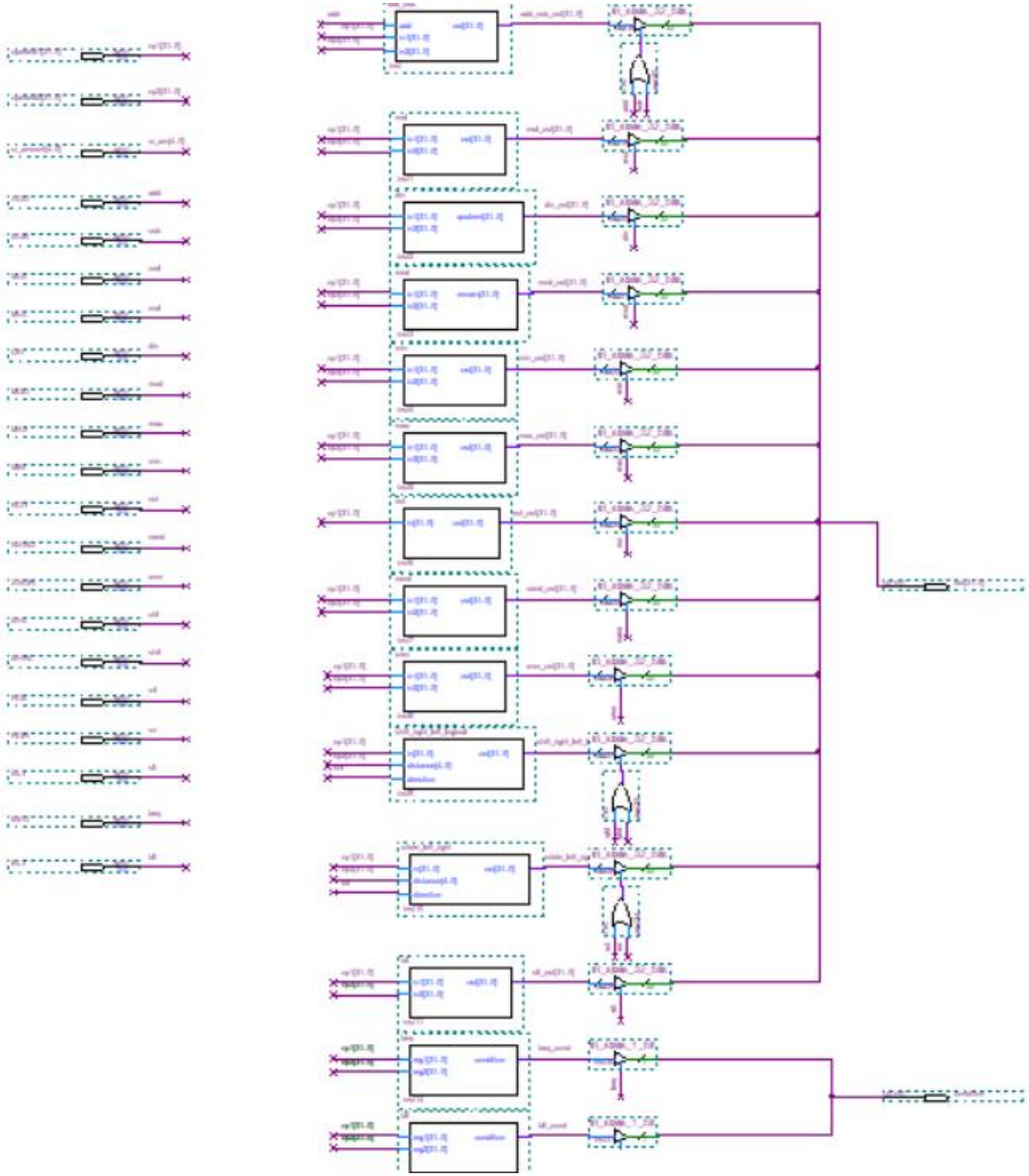
سیمبول همه مدار ها در بلاک ها قرار داده شد و در ALU اصلی مورد استفاده قرار گرفت. همانطور که در آن میبینید برای راحتی و عدم شلوغی کار ورودی ها و سیم ها به صورت جداگانه به وسیله نامگذاری پیاده سازی شده اند. مدار های فاز اول به صورت ترکیبی (دارای تاخیر) پیاده سازی شده و همه مدار ها ورودی ها را در هر زمان و با هر دستور دریافت می کنند و عملیات مربوطه را انجام می دهند اما اینکه این خروجی محاسبه از کدام مدار باید گرفته شود و به بیرون ALU داده شود را سیم های فعالسازی و Tristate ها تعیین می کنند. پشت سر هر مداری یک Tristate قرار داده شده و آن مدار های مشترک نیست عنصر سوم Tristateشان به وسیله or مورد استفاده قرار گرفته است.

همراه خروجی 32 بیتی ALU یک condition bit نیز قرار دارد که مخصوص دستورات شرطی است و این را به سی پی یو مخابره می کند که آیا شرط برقرار شده یا نه. ساختار این شرط ها هم شبیه قبل است.

برای کشیدن مدار ها از Mega wizard ها استفاده شده که در فولدر اصلی پروژه موجود است. در ضمن باید کلاک سی پی یو از بزرگترین تاخیر مدار ها بزرگ تر باشد که مشکلی بوجود نیاید که ایگونه نیز هست.

** برای دستورات blt و beq در alu تنها مقادیر دو رجیستر با هم مقایسه میشوند و نتیجه به صورت یک تک بیت به ماژول pc_controller رفته و همچنین محاسبه ادرس پرش در ماژول pc_controller صورت میگیرد.

** تمامی مدارات این بخش به صورت ترکیبی پیاده سازی شده اند.



MEM : در این مرحله اگر دستور نیاز به کار با حافظه داشته باشد, CU سیگنال متناظر با خواندن یا نوشتن از حافظه را به حافظه میفرستد.

** به دلیل تنوع در نوع خواندن و نوشتن از حافظه (نظیر خواند و نوشتن یک بایت یا یک کلمه) پایه های خواندن و نوشتن پنهایی بیش از یک بیت دارند.

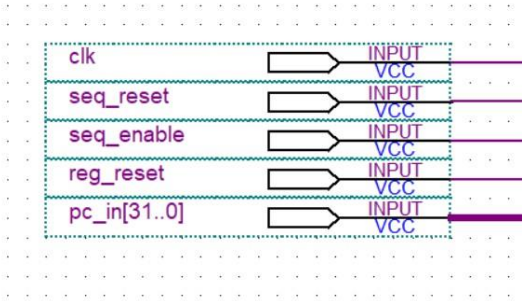
RB : در این مرحله اگر دیتای بدست آمده از مراحل قبلی نیاز به ذخیره سازی در register file داشته باشد, با توجه به opcode فرستاده شده از CU به ماژول write_back برای انتخاب مقداری که میخواهد نوشته شود و فعال شدن پایه write توسط CU صورت میگیرد.

** ماژول write_back در اصل یک مالتی پلکسر میباشد و کد وریلاگ این ماژول در write_back.v قرار دارد.

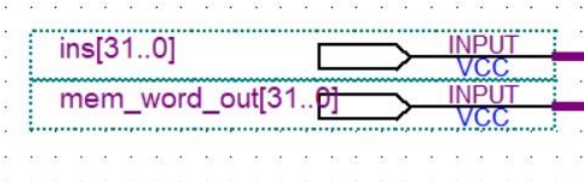
** در این مرحله program counter نیز آپدیت میشود و این کار در ماژول pc_controller انجام میشود. این ماژول با توجه به نوع آپدیت کردن pc از طریق دریافت آپکدی از CU و دریافت مابقی اطلاعات از ماژول های ALU و register file این کار را انجام میدهد. (کد وریلاگ درpc_controller.v قرار دارد.)

ورودی های پردازنده :

** پایه های کلاک , ریست sequencer و register file , فعال بودن sequencer و مقدار program counter ورودی و اولیه.

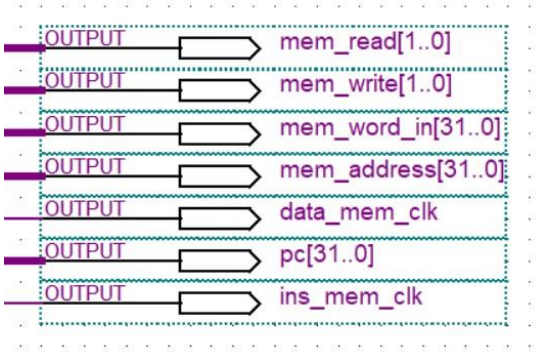


** ورودی های موردی نیاز در مرحله ID و MEM که به ترتیب برابر با دستور ورودی و مقدار خوانده شده از حافظه هستند.



خروجی های پردازنده :

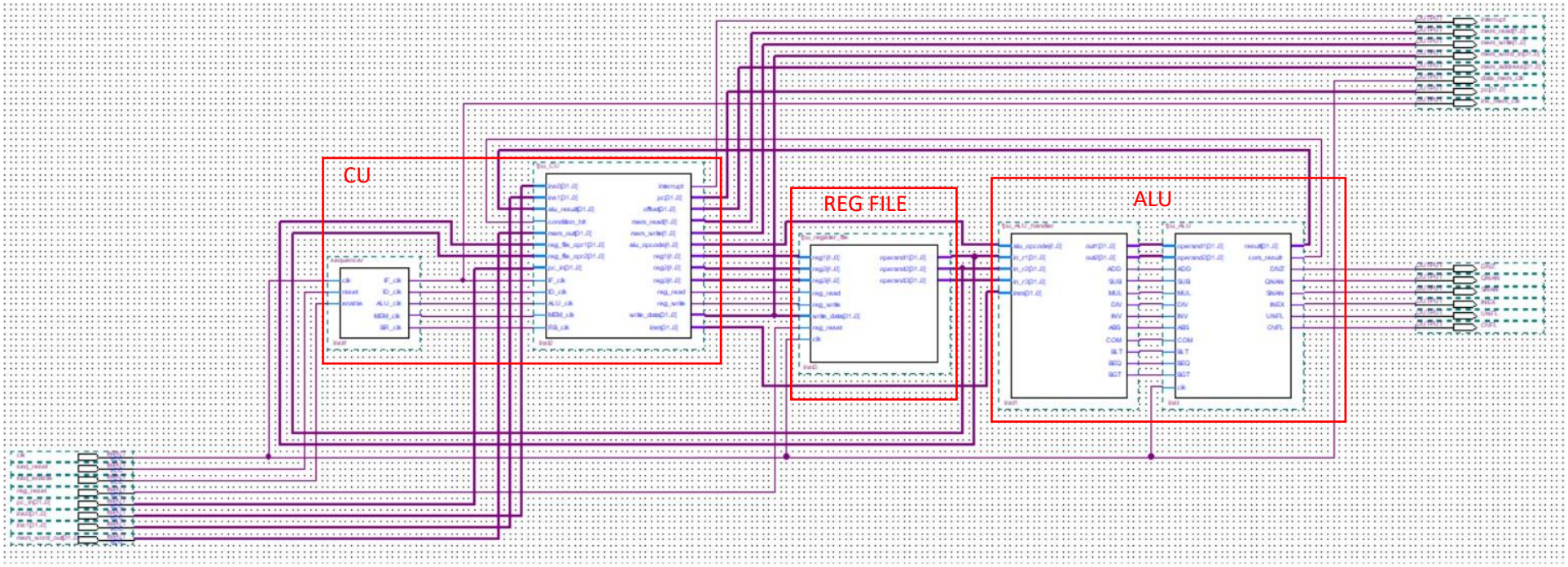
** پایه های کنترلی برای حافظه (شامل پایه های کنترلی خواندن و نوشتن , آدرس مورد نظر, دیتای نوشتن و کلاک ورودی به حافظه) و پایه های کنترلی برای حافظه دستورات (شامل شماره دستور و کلاک برای خواندن دستور).



** پایه های کنترلی برای interrupt که برای سوییچ کردن حافظه بین پردازنده و کمک پردازنده استفاده میشود.

2) FPU :

ساختار کمک پردازنده (برای محاسبات اعشاری fpu) به شکل زیر میباشد :



CU :

این ساختار همانند پردازنده از دو بخش sequencer و logic تشکیل شده است :

Sequencer : دقیقا همانند sequencer موجود در پردازنده.

LOGIC : این بخش با توجه به لبه های مثبت دریافت شده از sequencer متوجه میشود که در چه stage ای قرار دارد و چه اتفاقی باید در fpu رخ بدهد.

** کد وریلاگ این بخش در fpu_CU.v قرار دارد.

** معماری کمک پردازنده به صورت Multi-cycle بوده و با توجه به سیگنال های لبه مثبت در هر خروجی sequencer کار مورد نظر را انجام میدهد.

IF : در این مرحله دستور از memory خوانده میشود و وارد CU میشود.

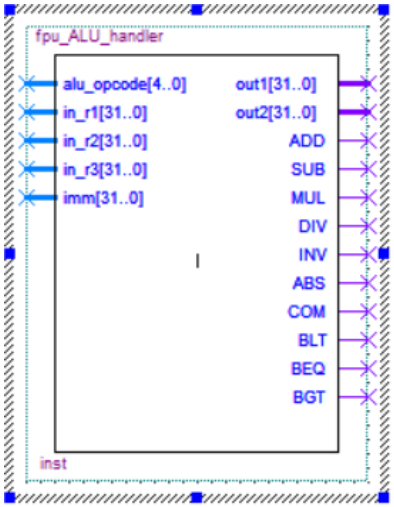
ID : در این مرحله CU دستور گرفته شده را میشکند و مقادیری مانند opcode , اندیس رجیستر های درگیر در دستور , immediate data که مربوط به 32 بیت در خط بعد از دستور fetch شده قرار میگیرد را تعیین میکند.

** در این مرحله register file با توجه با اندیس هایی که از CU دریافت میکند , اگر دستور نیاز به مقدار رجیستر داشته باشد CU پایه read آن را فعال میکند و مقادیر متناظر با اندیس ها و یا دیتای immediate از Register file خارج خواهند شد. (کد وریلاگ در fpu_register_file.v قرار دارد.)

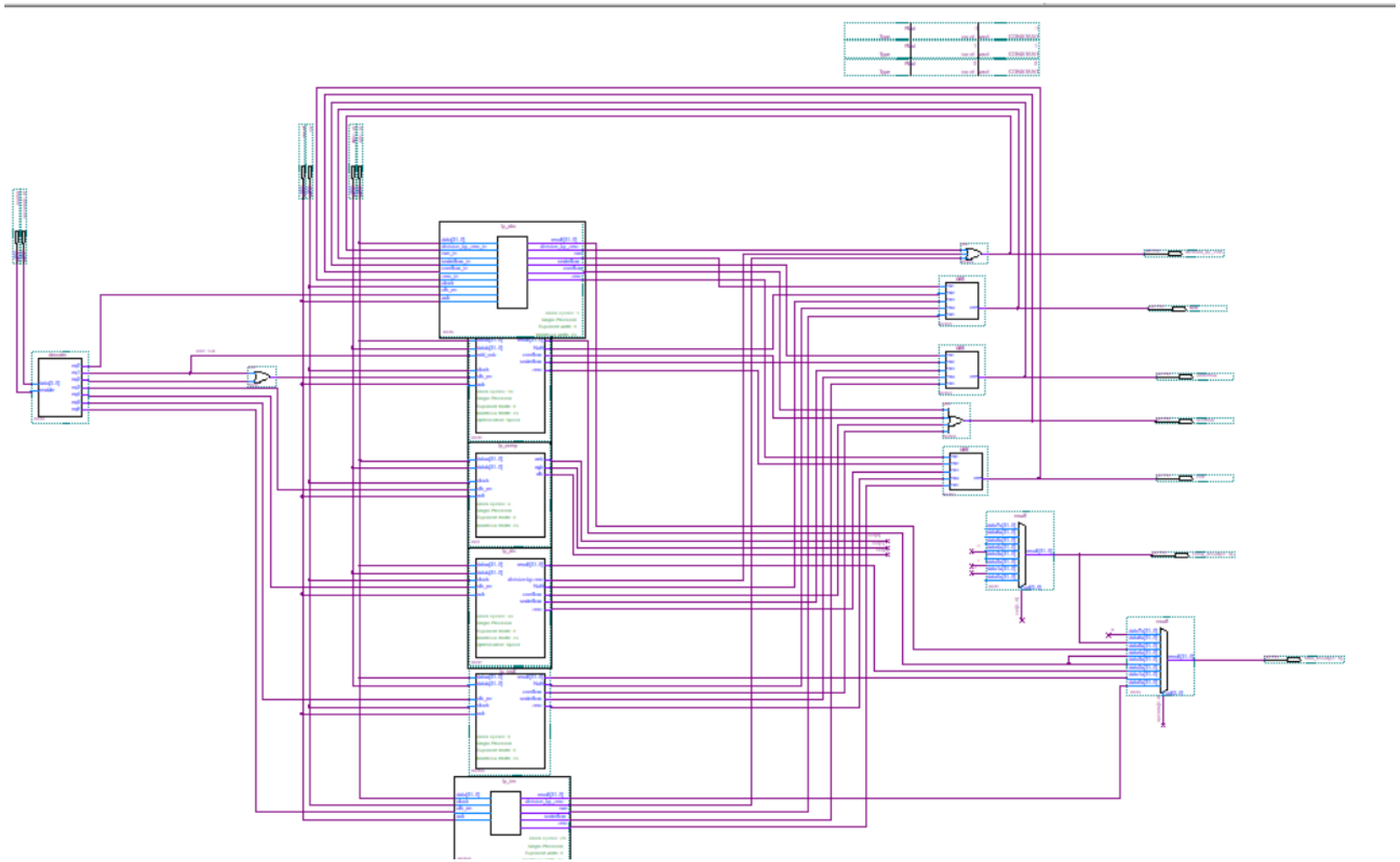
** در ماژول register file ترتیب خروجی به ترتیب اندیس های رجیستر داده شده در دستور است. در دستوراتی که immediate data یا immediate address دارند مقدار آن برخلاف register file موجود در cpu مستقیما به alu controller داده میشود.

ALU : در این مرحله اگر دستور به عملیات محاسباتی نیاز داشته باشد, با توجه با opcode و immediate data فرستاده شده از سوی CU مقادیر operand1 و operand2 و operand3 حاصل از register file یک عملیات محاسباتی را انجام میدهد.

** ماژول fpu alu handler به مانند alu controller به عنوان ورودی opcode را دریافت می کند و clk enable مدار را فعال می کند و همچنین ورودی ها را به آن مدار انتقال می دهد : (کد وریلاگ آن در فایل fpu_ALU_handler موجود است.)



(فایل fp_ alu) برای فاز دوم floating point را انتخاب کردیم و برای قسمت های مختلف آن به صورت ترتیبی مدار هایی طراحی کردیم و در نهایت مداری به شکل زیر به دست آمد (هر کدام از مدار های داخلی دارای clk enable و clk و reset به صورت آسنکرون و ورودی و خروجی هستند و در نهایت خروجی را برحسب opcode ورودی تعیین می کنیم و به branch handler می دهیم):



MEM : در این مرحله اگر دستور نیاز به کار با حافظه داشته باشد, CU سیگنال متناظر با خواندن یا نوشتن از حافظه را به حافظه میفرستد.

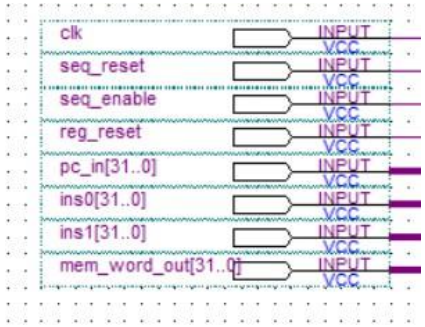
** به دلیل تنوع در نوع خواندن و نوشتن از حافظه (نظیر خواند و نوشتن یک بایت یا یک کلمه) پایه های خواندن و نوشتن پنهایی بیش از یک بیت دارند ولی در fpu تنها سیگنال های مربوط به خواندن و نوشتن ورد ارسال میشوند.

RB : در این مرحله اگر دیتای بدست آمده از مراحل قبلی نیاز به ذخیره سازی در register file داشته باشد, با توجه به opcode فرستاده شده از CU و فعال شدن پایه write آن صورت میگیرد.

** در این مرحله program counter نیز آپدیت میشود و این کار در CU انجام میشود. در کل دستورات branch در fpu تنها بخش مقایسه مقدار دو ثبات در ALU صورت میگیرد و مابقی آن نیز در خود CU انجام میشود.

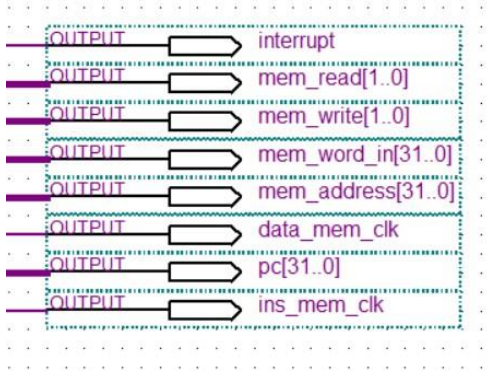
ورودی های کمک پردازنده :

** دقیقا همانند cpu

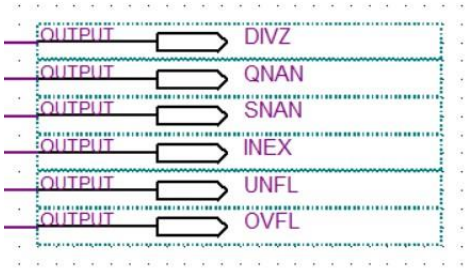


خروجی های پردازنده :

** یک بخش از خروجی ها دقیقا همانند خروجی های cpu است



** پایه های استثنائات در محسبات اعشاری بنابر IEE 754 در alu



** طول دستورات fpu همواره 32 بیت میباشند اما در مورد دستوراتی که immediate floating point میگیرند طول دستور 64 بیت میباشد که 32 بیت دوم , دیتای immediate میاید.

** همواره قبل و بعد از استفاده از دستورات fpu بایستی به ترتیب دستورات mtc و mfc را برای دسترسی به حافظه و فعال کردن fpu sequencer صدا زد.

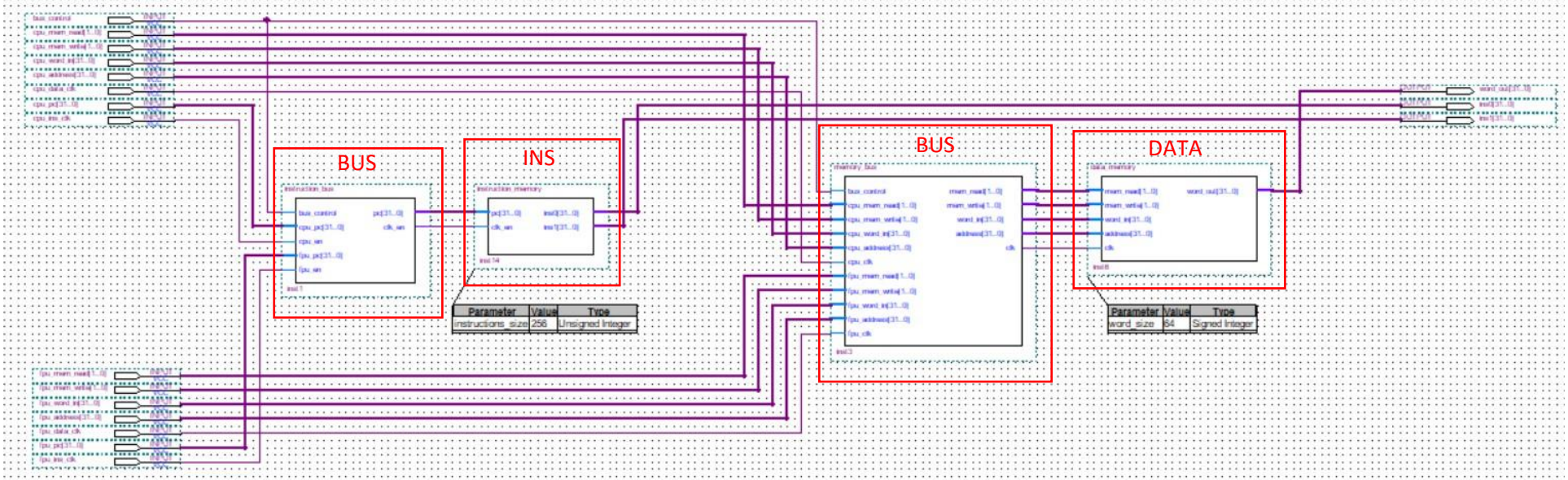
لیست دستورات قابل استفاده در fpu هنگام فعال بودن :

** (دستور mtc هنگامی که cpu فعال است صدا زده میشود).

instruction	job	Opc(6)	R0(5)	R1(5)	R2(5) addr(16) off(16)	Imm(32)	length
MTC	Move to float coprocessor	100000	-	-	-	-	32 bits
MFC	Move from float coprocessor	111111	-	-	-	-	32 bits
ADDF	DST ← SRC1 + SRC2	100001	5	5	5	-	32 bits
SUBF	DST ← SRC1 - SRC2	100010	5	5	5	-	32 bits
MULF	DST ← SRC1 * SRC2	100011	5	5	5	-	32 bits
DIVF	DST ← SRC1 / SRC2	100100	5	5	5	-	32 bits
INVF	DST ← 1 / SRC	100101	5	5	-	-	32 bits
ABSF	DST ← int32(SRC)	100110	5	5	-	-	32 bits
COMF	SRC1 > SRC2 : DST = 1 , SRC1 < SRC2 : DST = -1 , SRC1 == SRC2 : DST = 0	100111	5	5	5	-	32 bits
MOVIF	DST ← IMM	110000	5	5	-	32	64 bits
ADDIF	DST ← SRC + IMM	110001	5	5	-	32	64 bits
SUBIF	DST ← SRC - IMM	110010	5	5	-	32	64 bits
MULIF	DST ← SRC * IMM	110011	5	5	-	32	64 bits
DIVIF	DST ← SRC / IMM	110100	5	5	-	32	64 bits
INVIF	DST ← 1 / IMM	110101	5	5	-	32	64 bits
ABSIF	DST ← int32(IMM)	110110	5	5	-	32	64 bits
LF = LW	VR ← MEM [$AR+ SIGN EXTEND (Offset)]	111000	5	5	16	-	32 bits
SF = SW	MEM [$AR+ SIGN EXTEND (Offset)] ← VR	111001	5	5	16	-	32 bits
BEQF	REG1 == REG2 : PC ← PC + SIGN EXTEND (Address | “00”)	111100	5	5	16	-	32 bits
BLTF	REG1 < REG2 : PC ← PC + SIGN EXTEND (Address | “00”)	111101	5	5	16	-	32 bits
BGTF	REG1 > REG2 : PC ← PC + SIGN EXTEND (Address | “00”)	111110	5	5	16	-	32 bits
HLT	STOP PC	000000	-	-	-	-	32 bits

3) MEMORY :

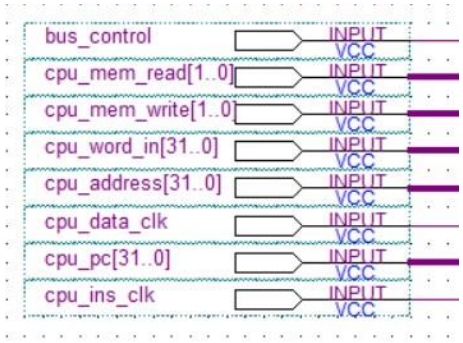
ساختار حافظه به شکل زیر میباشد :



حافظه دارای دو بخش data memory و instruction memory میباشد. که برای راحتی کار سایز data memory را 64 بایت و سایز instruction memory را 256 ورد درنظر گرفتیم. (کد وریلاگ آنها به ترتیب در فایل های data_memory.v و instruction_memory.v قرار دارند.)

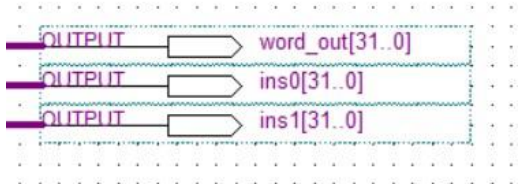
** از آنجایی که بین cpu و fpu حافظه share میشود bus هایی داریم که همانند مالتی پلکسر عمل میکنند که پایه کنترل آنها دست ماژول interrupt_handler میباشد. (کد وریلاگ آنها به ترتیب در فایل های memory_bus.v و instruction_bus.v قرار دارند.)

ورودی های حافظه : سیگنال مربوط به کنترل bus ها و سیگنال های کنترلی حافظه مورد نیاز از طرف cpu و fpu :



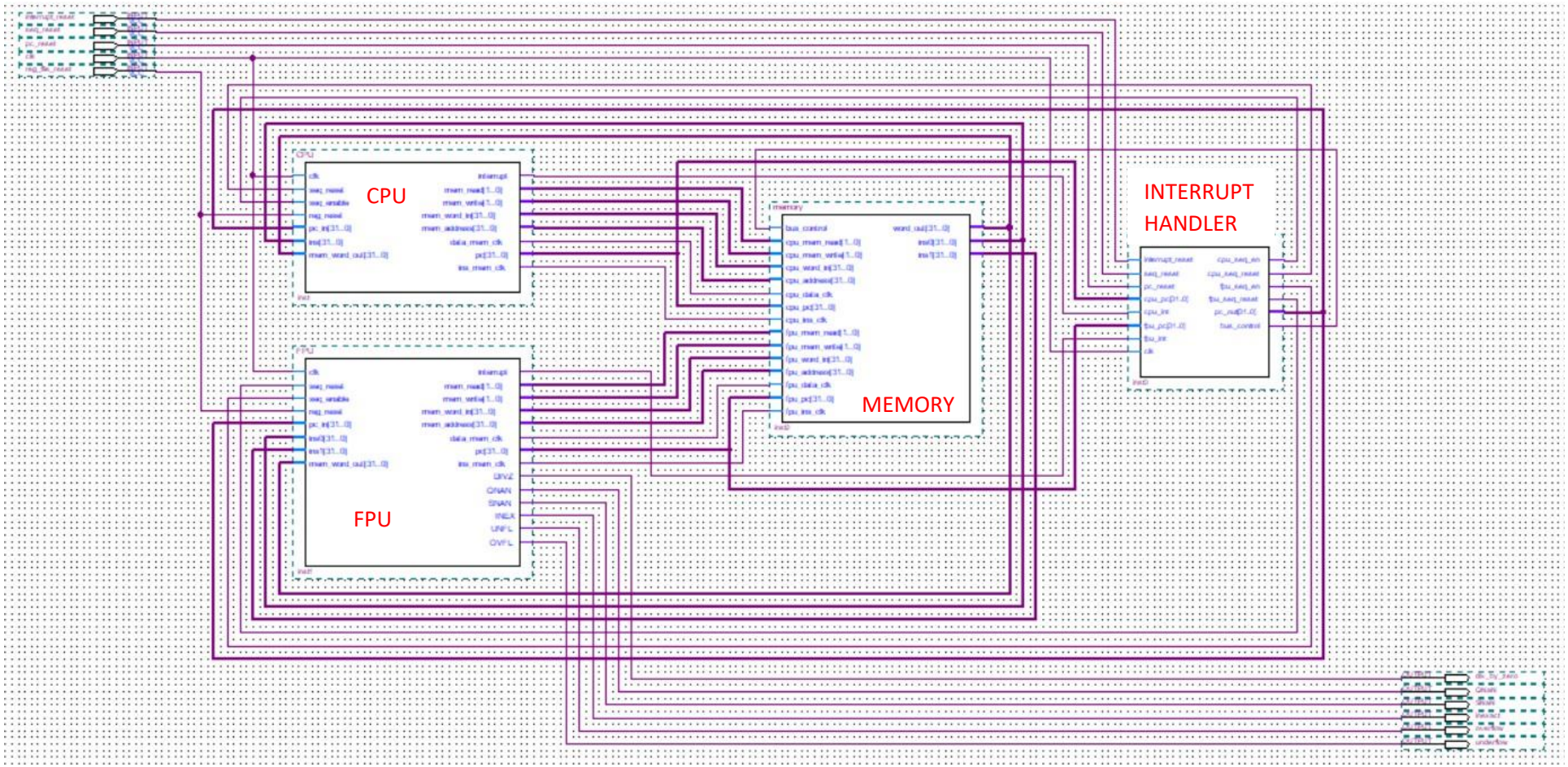
خروجی های حافظه : خروجی مربوط به دستورات خوانده شده از instruction memory برای fpu و cpu و همچنین خروجی ورد مورد نیاز از data memory در صورت نیاز.

** ins1 دستور بلافصله بعد از ins0 بوده و برای دستورات نوع immediate در fpu کاربرد دارد و درکل دستوری که در fpu و cpu اجرا میشود ins0 است.



4) MACHINE :

ساختار نهایی ماشین طراحی شده به شکل زیر میباشد :



Interrupt handling :

** ماژول interrupt handler وظیفه تخصیص حافظه به cpu یا fpu و فعال نگاه داشتن cpu یا fpu را برعهده دارد. (کد وریلاگ این ماژول در interrupt_handler.v موجود است.)

نحوه عملکرد: هنگامی که cpu به دستور mtc در مرحله ID برمیخورد, پایه interrupt خود را یک کرده و ماژول interrupt_handler پس از دریافت آن کار های زیر را انجام میدهد:

1) bus control حافظه را تغییر میدهد (یعنی با تغییر آن, حافظه به fpu اختصاص پیدا میکند.)

2) sequencer مربوط به cpu را reset کرده و آن را متوقف میکند ← با این کار عملا cpu هیچ کاری انجام نمیدهد.

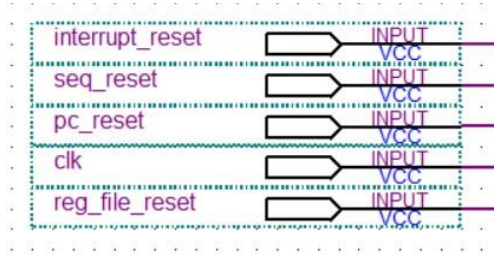
3) ثبات pc موجود در cpu را گرفته و ثبات pc موجود در fpu را آپدیت میکند (از آن pc به بعد را fpu اجرا میکند).

4) sequencer مربوط به fpu را reset کرده و آن را فعال میکند ← با این کار عملا fpu شروع به کار میکند.

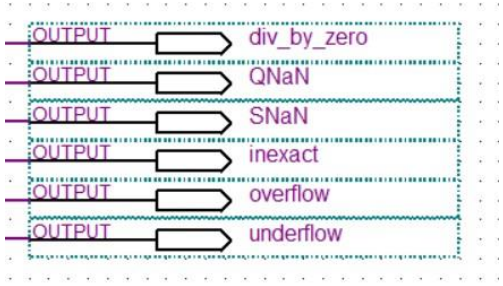
هنگامی که fpu به دستور mfc در مرحله ID برمیخورد دقیقا مراحل بالا رخ میدهد, فقط سویچ از fpu به cpu خواهد بود.

** کل فرایند switching یک کلاک هزینه دارد.

ورودی های مدار : ورودی های مربوط به کلاک کلی پردازنده ها و ریست کردن sequencer , register file , ثبات program counter مربوط به cpu و fpu و همچنین مربوط به ریست کردن interrupt bus مربوط به memory .



خروجی های مدار : خروجی های مربوط به fpu



5) TEST BENCH :

تمامی تست های انجام شده با modelsim بوده و فایل testbench.v ماژول top پروژه است که در آن ابتدا یکبار پایه های ریست ماژول machine فعال میشود و بعد از تعداد مشخصی سیکل زمانی شبیه سازی متوقف میشود.

** در تمامی تست ها کلاک ماشین 2 واحد زمانی در نظر گرفته شده است.

** اجرای تمامی دستورات به اندازه 5 کلاک طول میکشد (به جز دستورات mtc و mfc که به اندازه 3 کلاک پردازنده طول میکشند).

** در هر بار تست ابتدا data memory از روی فایل tests/in/initial_data_memory.mem و instruction memory از روی tests/in/instruction _memory.mem مقدار دهی اولیه شده و پس از آن برنامه موردنظر شروع به اجرا شدن میکند. و درنهایت در پوشه tests/out به ترتیب فایل های cpu_register_file.mem و fpu_register_file.mem و final_data_memory.mem تولید میشوند.

تولید تست ها :

برای آماده کردن تست‌های پروژه، یک کد پایتون نوشتیم که با گرفتن کد میپس، آن را به زبان ماشین تبدیل می‌کند و هم‌چنین خروجی‌های ممووری و رجیسترها را می‌سازد.

با اجرای برنامه main.py اتفاقات زیر می‌افتد:

- کد مپیس از فایل code.txt خوانده می‌شود
- از فایل input_memory.mem مقادیر اولیه حافظه خوانده می‌شود (حافظه به صورت هشت بیت هشت بیت است).
- کد زبان ماشین در فایل out.mem نوشته می‌شود.
- در فایل cpu_register_files.mem تعداد ۳۲ خط چاپ می‌شود که مقادیر رجیسترها را نشان می‌دهد.
- در فایل memory.mem مقادیر نهایی حافظه به صورت هشت بیت، هشت بیت نوشته می‌شوند.

تست های خاص :

این تست ها در پوشه tests و به ترتیب به نامهای fib و fpu قرار دارند.

1) برنامه ای که n را از حافظه خوانده و جمله n ام فیبوناچی را نمایش میدهد:

زمان اجرا بر اساس تعداد سیکل :

$(n+1)*25 = n*5*5 + 5*5$

در مثال تست شده n=10 بوده و زمان اجرا برابر با 275 سیکل پردازنده بوده و عدد نهایی که 55 یا 00110111 میباشد در آدرس 4 حافظه نوشته خواهد شد.

```
sub $r0 $r0 $r0
lw $r1 $r0 0
addi $r2 $r0 0
addi $r3 $r0 1
add $r4 $r2 $r3
addi $r2 $r3 0
addi $r3 $r4 0
subi $r1 $r1 1
blt $r0 $r1 -4
sw $r2 $r0 1
hlt
```

```
000010_00000_00000_00000_00000_000000
011000_00001_00000_0000000000000000
010010_00010_00000_0000000000000000
010010_00011_00000_0000000000000001
000001_00100_00010_00011_00000_000000
010010_00010_00011_0000000000000000
010010_00011_00100_0000000000000000
010011_00001_00001_0000000000000001
011111_00000_00001_1111111111111100
011001_00010_00000_0000000000000001
000000_00000_00000_0000000000000000
```

2) برنامه ای که در آن دو عدد ممیز شناور را از ورودی گرفته، عدد بزرگتر را بر عدد کوچکتر تقسیم کرده و نتیجه ی گرد شده را ذخیره کند.:

در مثال تست شده در حافظه اولیه عدد در خانه های 0 و 4 حافظه به ترتیب اعداد 5.5 و 4.5 قرار دارند که حاصل نهایی که برابر با نزدیک ترین مقدار صحیح به 5.5/4.5 یعنی 1 است در خانه 8 ام حافظه قرار میگیرد.

```
mtc
subf $f0 $f0 $f0
lf $f3 $f0 0
lf $f4 $f0 4
bltf $f3 $f4 3
lf $f3 $f0 4
lf $f4 $f0 0
divf $f4 $f4 $f3
absf $f4 $f4
sf $f4 $f0 8
mfc
```

```
100000_00000_00000_0000000000000000
100010_00000_00000_00000_00000000000
111000_00011_00000_0000000000000000
111000_00100_00000_0000000000000100
111101_00011_00100_0000000000000011
111000_00011_00000_0000000000000100
111000_00100_00000_0000000000000000
100100_00100_00100_00011_00000000000
100110_00100_00100_0000000000000000
111001_00100_00000_0000000000001000
111111_00000_00000_0000000000000000
000000_00000_00000_0000000000000000
```