به نام خدا

فاز چهارم پروژه معماری کامپیوتر

نیمسال دوم سال تحصیلی ۱۴۰۰-۱۴۰۱

دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

توضیحات فاز:

در این فاز ما به پردازنده‌ای که ساختیم، امکان کار با اعداد اعشاری استاندارد IEEE754 را دادیم. این کار را به کمک یه ALU جداگانه و یک رجیستر فایل مخصوص اعداد اعشاری انجام دادیم. دستوراتی که در این فاز پیاده شدند عبارتند از: انتقال از و به رجیستر فایل مخصوص اعداد اعشاری، جمع، تفریق، ضرب، تقسیم، وارون عدد، گرد کردن، تبدیل عدد صحیح به اعشاری و برعکس. برای هر یک از دستورات اعداد اعشاری opcodeهای جدیدی تعریف شده است که در ادامه لیست آنها آمده است. همچنین ALU که طراحی شده است سیگنال‌های خروجی Divide by Zero, qNaN, sNaN, Underflow, Overflow را دارا است.

اعضای تیم:

- *سهیل نظری مندجین*
- *بنیامین ملکی*
- *هیربد بهنام*
- *هیراد داوری*

FPU:

در ابتدا به توضیح FPU و سیگنال‌های ورودی و خروجی آن می‌پردازیم. تعریف سیگنال‌های خروجی به صورت زیر است:

- Division by Zero: تقسیم بر ۰ اتفاق افتاده است.
- Quiet Not a Number و Signaling: عددی که به صورت `?111111111?????????????????????` است qNaN نامیده می‌شود. qNaN در واقعیت با sNaN فرق دارد چرا که می‌توان sNaN را trap کرد. ولی qNaN سیگنالی را فعال نمی‌کند و باعث رخ دادن expectation در alu نمی‌شود. sNaN به صورت `?1111111101?????????????????????` تعریف شده است.
- Inexact: درصورتی که یک عدد خیلی بزرگ را بعلاوه‌ی یک عدد خیلی کوچک کنیم ممکن است که از دقت یکی از اعداد مجبور شویم که صرف نظر کنیم. این سیگنال زمانی فعال می‌شود که عدد خروجی جواب دقیق نیست.
- Overflow و Underflow: در زمان‌هایی جواب نهایی برابر منفی یا مثبت بی‌نهایت می‌شود (مثلا ضرب دو عدد خیلی بزرگ). در این حالت Overflow یا Underflow فعال می‌شود.

در خود FPU از تمامی ماژول‌هایی که در زیر آمده شده instance گرفته شده است و سپس ورودی‌های یکسان (a و b) به تمامی آنها داده می‌شود. در نهایت بین خروجی‌های آنها یک mux قرار می‌دهیم و با توجه به fpu opcode بین خروجی‌های ماژول‌ها انتخاب می‌کنیم.

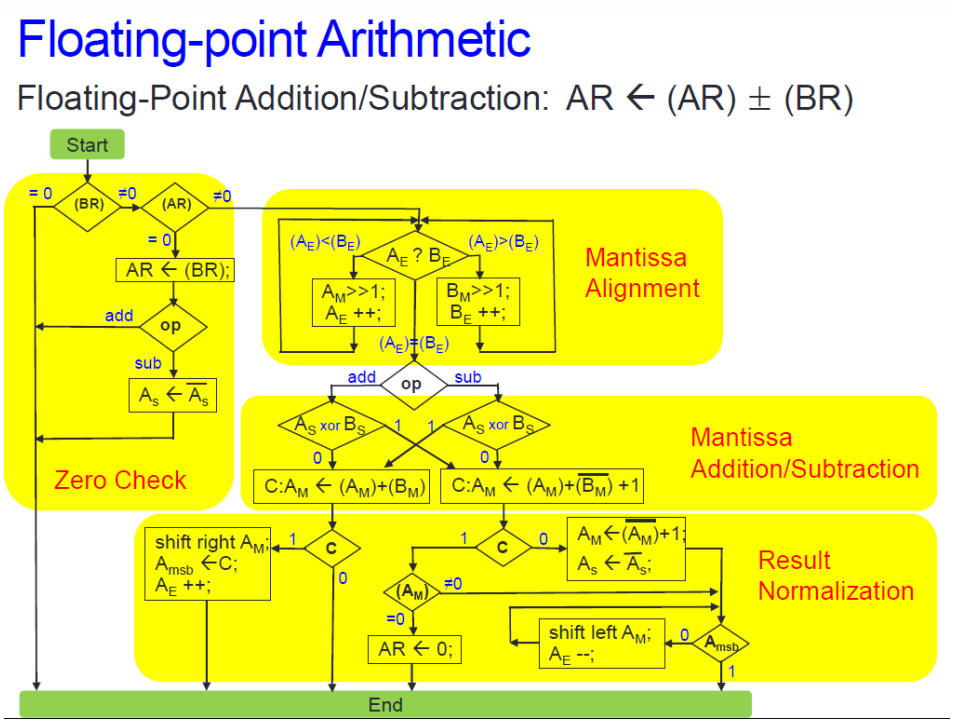
حال به معرفی ماژول‌های FPU می‌پردازیم.

Floating Point Negator:

این ماژول عدد ورودی را قرینه (وارون) می‌کند. این کار صرفا با قرینه کردن بیت آخر که sign bit است صورت می‌گیرد.

Floating Point Adder/Subtractor:

این قطعه دو عدد اعشاری را جمع/تفریق می‌کند. در ابتدا برای راحت تر شدن کار در صورتی که باید تفریق صورت بگیرد، عدد دوم را قرینه می‌کنیم و سپس جمع را انجام می‌دهیم. فلوچارت جمع از اسلایدها برداشته شده است و به صورت زیر است. دقت کنید که صرفا op=sub وجود ندارد چرا که ما b را در صورت نیاز معکوس کردیم.



برای تشخیص دادن inexact باید زمانی که fraction را به چپ یا راست شیف می‌دهیم چک کنیم که ۱ای را دور می‌اندازیم یا خیر. در این صورت باید سیگنال inexact فعال باشد. برای overflow/underflow می‌توان از توان کمک گرفت. در صورتی که توان overflow شد باید با توجه به sign bit سیگنال overflow یا underflow را فعال کنیم. در صورتی که underflow در توان رخ داد خروجی برابر ۰ می‌شود.

Floating Point Multiplier:

در این ماژول ما دو ورودی a و b را دریافت می‌کنیم و سپس حاصل ضرب آن‌ها را در خروجی result قرار می‌دهیم. البته این محاسبه، حالت‌های خاص نیز دارد که به همین جهت ابتدا آن‌ها ر ا بررسی می‌کنیم:

اگر هر کدام از اعداد a یا b صفر (ZERO) باشند و عدد دیگر NaN یا Infinity نباشد، حاصل ضرب برابر ZERO خواهد شد.

اگر یکی از اعداد a یا b صفر و دیگری NaN یا Infinity باشد، خروجی QNAN خواهد بود.

اگر شرایط قبلی را نداشته باشیم ولی یکی از اعداد a یا b Infinity باشد، خروجی Infinity خواهد بود و علامت آن از xor علامت‌های ورودی‌ها بدست می‌آید.

در غیر از شرایط بالا الگوریتم نرمال ضرب اعداد اعشاری را اجرا می‌کنیم؛ علامت حاصل را از xor علامت‌های ورودی‌ها بدست می‌آوریم. برای بدست آوردن توان حاصل‌ضرب، توان‌های ورودی‌ها را جمع می‌کنیم، اما چون توان‌های موجود در سخت‌افزار ما biased هستند، لازم است که حاصل‌جمع توان‌ها را منهای ۱۲۷ کنیم تا این مقدار ۲ بار کم نشده باشد. برای محاسبه مانتیس حاصل‌ضرب نیز مانتیس‌های ورودی‌ها را در هم ضرب می‌کنیم و آن حاصل‌ضرب را در یک 48 reg بیتی می‌ریزیم. (زیرا در حقیقت علاوه بر ۲۳ بیت موجود در نمایش‌های اعداد، یک یک در ابتدای این مانتیس‌ها به صورت نرمالایزد باید در نظر بگیریم و در حقیقت ۲ عدد ۲۴ بیتی را در هم ضرب کنیم که حاصل حداکثر با ۴۸ بیت قابل نمایش خواهد بود. سپس با توجه به اینکه حاصل ضرب ما به چه صورت است، توان result را بازنگری کرده تا عدد اعشاری نتیجه نیز به

صورت normalised در بیاید و فرمت اعداد اعشاری ما بهم نریزد. برای overflow و underflow نیز پیش از مرحله نرمالایزیشن، چک می‌کنیم که آیا توان result علامت متفاوتی نسبت به xor علامت‌های توان‌های ورودی‌ها دارد یا خیر و در صورت تفاوت داشتن overflow یا underflow (بسته به منفی یا مثبت بودن توان result) فعال می‌کنیم.

Floating Point Divisor:

این قطعه با ورودی گرفتن دو عدد a و b، خارج قسمت تقسیم a بر b را به عنوان خروجی روی result می‌نویسد. در این تقسیم حالت‌های خاصی ممکن است رخ دهد که آن‌ها در ابتدا جداگانه بررسی می‌شوند:

اگر یکی از ورودی‌ها NaN باشد، خروجی Qnan خواهد بود.

در غیر این صورت، اگر تقسیم بر صفر داشته باشیم و مقسوم صفر باشد، خروجی QNAN می‌شود و اگر تقسیم بر صفر داشته باشیم و مقسوم مخالف صفر باشد، خروجی Infinity می‌شود و علامت آن با توجه به علامت‌های ورودی‌ها تعیین می‌شود.

در غیر این صورت اگر تقسیم بر Infinity داشته باشیم و مقسوم نیز Infinity باشد، خروجی QNAN می‌شود. اگر مقسوم بی‌نهایت نباشد، خروجی صفر می‌شود.

در غیر شرایط بالا، اگر مقسوم صفر باشد، خروجی صفر شده و اگر مقسوم Infinity باشد، خروجی Infinity می‌شود و علامت آن با xor گرفتن از علامت ورودی‌ها مشخص می‌شود.

اگر هیچ‌کدام از موارد بالا رخ ندهد، حالت خاص نداریم و الگوریتم عادی تقسیم fp را اجرا می‌کنیم؛ علامت نتیجه از xor گرفتن علامت ورودی‌ها مشخص می‌شود. توان a باید منهای توان b شود ولی چون biased هستند، لازم است که ۱۲۷ را به نتیجه تفریق بالا اضافه کنیم تا یک بار bias ما بر توان خروجی اثر گذاشته باشد. برای محاسبه مانتیس خروجی نیز، مانتیس نرمالایز شده a را ۲۴ واحد به چپ شیفت می‌دهیم و در یک reg ۴۸ بیتی می‌گذاریم. این reg را به extend شده مانتیس b به ۴۸ بیت، تقسیم می‌کنیم. حال اگر مانتیس a از مانتیس b کوچکتر بود، ۲۳ بیت LSB حاصل تقسیم را در مانتیس result قرار می‌دهیم و توان نتیجه را نیز منهای ۱ می‌کنیم. اگر این مانتیس‌ها برابر بودند، ۲۳ بیت صفر را در مانتیس نتیجه قرار می‌دهیم. اگر مانتیس مقسوم از مانتیس مقسوم علیه بزرگ‌تر بود، بیت‌های دوم تا ۲۴ام حاصل تقسیم را در مانتیس result قرار می‌دهیم. برای تشخیص over/underflow نیز پیش از انجام محاسبه مانتیس بررسی می‌کنیم که آیا توان نتیجه، نسبت به توان‌های ورودی‌ها وضعیت غیرنرمالی دارد یا خیر و اگر چنین بود، با توجه به علامت توان result، اعلام overflow/underflow می‌کنیم.

Floating Point Comparator:

این ماژول دو ورودی میگیرد و سه ورودی lt, eq, gt را به ما می‌دهد. در ابتدا دقت کنید که در صورتی که یکی از ورودی‌های ما NaN بود باید تمامی سیگنال‌ها را ۰ کنیم. حال در صورتی که بیت‌های دو عدد برابر بودند باید سیگنال eq را ۱ کنیم. حال باید چک کنیم که آیا عددی بی نهایت است یا خیر. این حالت‌های خاص رو جداگانه حساب می‌کنیم.

در غیر این صورت در ابتدا چک می‌کنیم که آیا a یا b هر کدام ۰ هستند یا خیر. در این حالت فقط بیت علامت را نگاه می‌کنیم. در صورتی که اعداد غیر صفر بودند در ابتدا sign bit آنها را نگاه می‌کنیم. در صورتی که فرق داشتند که جواب مشخص است و صرفا با نگاه کردن به sign بیت مشخص می‌شود. در غیر این صورت در ابتدا توان و سپس fraction را نگاه می‌کنیم که برای کدام یک از اعداد بزرگتر است.

Binary To Floating Point:

برای تبدیل یک عدد در مبنای دو به یک عدد در floating point در ابتدا چک می‌کنیم که آیا عدد ورودی ۰ است یا خیر. در صورتی که عدد حاصل صفر بود، تمام بیت‌های خروجی را ۰ می‌کنیم و به کار خاتمه می‌دهیم. در غیر این صورت اندیس پر ارزش ترین ۱ را پیدا می‌کنیم و اندیس آنرا برابر exponent عدد قرار می‌دهیم. سپس مابقی بیت‌های عدد را در قسمت fraction قرار می‌دهیم. در صورتی که بیتی بعد از آخرین بیت قرار داده شده در fraction برابر ۱ باشد سیگنال inexact را فعال می‌کنیم.

Floating Point To Binary:

مشخص است بعضی از اعداد floating point بزرگتر از آن هستند که در ۳۲ بیت عدد صحیح جا شوند. پس در اینجا نیاز به سیگنال underflow و overflow داریم. در ابتدا در صورتی که عدد ورودی برابر qNaN یا sNaN بود خروجی را برابر ۰ قرار می‌دهیم (این را قرارداد می‌کنیم). اگر عدد ورودی نیز مثبت یا منفی بی نهایت بود سیگنال‌های overflow یا underflow را با توجه به sign فعال می‌کنیم. همچنین در صورتی که توان بیشتر از ۳۲ بود همین کار را انجام می‌دهیم. حال در صورتی که توان کمتر از ۰ بود، عدد حاصل را برابر ۰ قرار می‌دهیم و کار را به اتمام می‌رسانیم.

در غیر این صورت بیتی که اندیس آن توان است را برابر ۱ قرار می‌دهیم. سپس از پر ارزش ترین بیت fraction شروع به حرکت می‌کنیم و بیت‌های قبلی در جواب را برابر بیت‌های fraction قرار می‌دهیم. این کار را تا زمانی ادامه می‌دهیم که به تعداد exponent بیت کپی کرده باشیم یا اینکه به آخر هر یک از اعداد برسیم.

گرد کردن عدد اعشاری:

یکی از روش‌های گرد کردن اعداد با توجه به کف است! بدین صورت که داریم: $round(x) = \lfloor x + 0.5 \rfloor$. پس کافی است که به کمک جمع کننده، ورودی بعلاوه‌ی ۰.۵ را حساب کنیم و به ماژول Floating Point to Binary بدهیم.

یونیت تست برای ماژول‌ها:

یکی از کارهایی که با اعضای گروه تصمیم به انجام آن گرفتیم ایجاد unit test برای تک تک ماژول‌های floating point بود. از آنجایی که verilator کد Verilog را تبدیل به یک کلاس C++ می‌کند، می‌توان آنرا به راحتی در C++ تست کرد. به عنوان مثال تابع زیر برای تست ماژول مقایسه کننده نوشته شده است:

```
void test_number(VFP_Comparator& module, float a, float b) {
    module.a = extract_float_bits(a);
    module.b = extract_float_bits(b);
    module.eval();
    uint32_t expected_pattern = (a > b) * 100 + (a == b) * 10 + (a < b);
    uint32_t got_pattern = (module.gt) * 100 + (module.eq) * 10 + (module.lt);
    if (expected_pattern != got_pattern)
        std::cout << "Invalid comparation on " << a << " and " << b << ": Expected " << expected_pattern << " but got " << got_pattern << std::endl;
}
```

سپس به ازای اعداد تصادفی این تابع را اجرا می‌کنیم و چک می‌کنیم نتیجه‌ی ماژول با نتیجه‌ی واقعی یکی باشد.

Control Unit:

دستور العمل‌ها:

add	000001
sub	000101
mult	000111
div	001001
negate	001010
round	001011
Float to binary	001101
Binary to float	001110
Comp_lt	001111
Comp_le	010000
Comp_eq	010001
Comp_nq	010010
Comp_gt	010011
Comp_ge	010100
Move to float	010101
Move from float	010110

ما دستورات Floating point را در داخل دستورات R_Type جا دادیم زیرا ۶۴ دستور را طبق توضیحات زیر میتوان در اپکد دستورات R_type جا داد ولی در پروژه ما تنها ۱۸ دستور R_Type استفاده کرده ایم پس ۱۶ دستور فلوتینگ پوینت را میتوان در بین دستورات آن جا داد و اینگونه سیم کشی نهایی ساده تر خواهد شد.

R (Register) Format

This divides the instruction into six fields as follows:

6	5	5	5	5	6
op	**rs**	**rt**	**rd**	**shamt**	**funct**

Where,

op = opcode

rs = identifier of first source register

rt = identifier of second source register

rd = identifier of destination register

shamt = shift amount indicating how many bits the contents of a register must be shifted left or right (only used in shift instructions – NOT used here)

funct = distinguishes among R-type instructions as all R-type instructions have op = 0.

برای انتقال دیتا بین رجیستر فایل‌ها از دستور جمع هر ALU و دادن صفر به عنوان دومین عدد در جفت ALUها استفاده می‌کنیم.

دیتافلو:

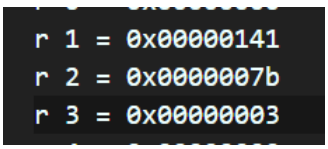
برای دیتافلو عملا یک رجیستر فایل در کنار (در همان stage) رجیستر فایل عادی قرار دارد. ورودی‌های آن دقیقا عین رجیستر فایل عادی است. Write enable و write register و write data نیز از مرحله‌ی آخر pipeline گرفته می‌شوند. FPU نیز در کنار ALU عادی است.

تست پردازنده:

برای تست پردازنده برنامه‌ی زیر نوشته شده است که مشابه مورد خواسته شده در دستور کار است.

```
# Define the numbers
addi $1, $0, 123
addi $2, $0, 321
# Move them to float register file
move.to.float $1, $1
move.to.float $2, $2
# Convert them to float
bin.to.float $1, $1
bin.to.float $2, $2
# Compare
float.lt $3, $1, $2
move.from.float $3, $3
beq $3, $0, DONE
# Swap if here
add $3, $0, $1
add $1, $0, $2
add $2, $0, $3
# Do it again
DONE:
move.to.float $1, $1
move.to.float $2, $2
bin.to.float $1, $1
bin.to.float $2, $2
float.div $3, $1, $2
float.round $3, $3
move.from.float $3, $3
```

این برنامه را تبدیل به ماشین کد می‌کنیم و در فایل‌های تست قرار می‌دهیم. سپس برای رجیستر فایل نهایی باید اعداد زیر را داشته باشیم:



رجیستر ۳ جواب گرد شده‌ی تقسیم ۳۲۱ بر ۱۲۳ است که برابر ۳ است. سپس تمامی تست‌ها را ران می‌کنیم و متوجه می‌شویم که تمام آنها پاس می‌شوند: