

به نام خدا



آزمایش شماره ۴

آز معماری - دکتر سربازی آزاد

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

نیمسال تابستان ۰۲-۰۳

اعضای گروه:

میترا قلی پور چنانی - ۰۱۱۰۶۳۶۳

نیکا قادری - ۰۱۱۰۶۳۲۸

ملیکا علیزاده - ۰۱۱۰۶۲۵۵



۱ هدف و ایده کلی

هدف از این آزمایش طراحی یک واحد محاسباتی برای اعداد ممیز شناور با تعداد بیت ۳۲ می‌باشد. این واحد محاسبات تنها اعمال جمع و تفریق را انجام می‌دهد که با یک بیت ورودی کنترل می‌شود. جزئیات پیاده‌سازی به صورت زیر می‌باشد:

در این آزمایش (طی دو جلسه)، مدار یک جمع/تفریق‌کننده ممیز شناور را طراحی کرده و با استفاده ابزار Proteus شبیه‌سازی می‌کنیم. پس از اطمینان از صحت عملکرد در شبیه‌ساز، آن را روی بورد پیاده‌سازی می‌کنیم. مدار اولیه برای شبیه‌سازی را مطابق استاندارد IEEE-754 بستی طراحی کنید. برای سهولت در پیاده‌سازی روی بورد، تعداد بیت‌ها را از ۳۲ به ۱۲ کاهش دهید.

مشخصات مدار مورد نظر به قرار زیر است:

عملوند اول (ورودی): A

عملوند دوم (ورودی): B

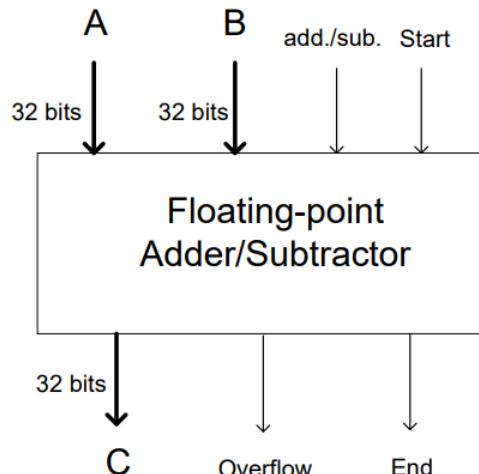
حاصل جمع/تفریق (خروجی): C

شروع عملیات (ورودی): Start

پایان عملیات (خروجی): End

Overflow (خروجی): Overflow

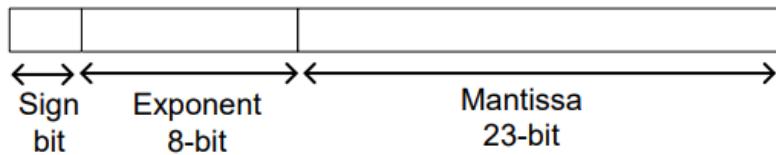
مشخص کننده جمع/تفریق (ورودی): add/sub



شکل ۴: جمع/تفریق‌کننده ممیز شناور



فرمت نمایش اعداد به صورت IEEE – 754 می‌باشد. یعنی بیت اول، بیت علامت، ۸ بیت بعدی نمایانگر توان و ۲۳ بیت بعدی، قسمت فرکشن یا همان مانتیسا می‌باشند:



شرح آزمایش

ابتدا مدار جمع و تفريط کننده دو عدد ممیز شناور (شکل ۴) را با فرمت استاندارد IEEE-754 مطابق شکل ۵ طراحی کرده و با ابزار Proteus شبیه‌سازی نمایید. پس از اطمینان از صحت

عملکرد، طراحی انجام شده را روی بورد پیاده‌سازی نمایید. برای سهولت در پیاده‌سازی، تعداد بیت‌های مدار طراحی شده را مطابق شکل ۶ از ۳۲ بیت به ۱۲ بیت کاهش دهید. با فعال شدن سیگنال Start، مدار شروع به کار کرده و اگر سیگنال add/sub برابر صفر باشد، مقدار $A+B$ و اگر این سیگنال برابر یک باشد، مقدار $A-B$ را محاسبه کرده و روی خط C قرار می‌دهد و سیگنال End را به منزله اتمام عملیات فعل می‌کند. ورودی‌های A و B نرمالیزه بوده و خروجی C نیز باید نرمالیزه باشد. در صورت بروز سرریز، سیگنال Overflow فعل می‌شود. استفاده از شمارنده با قابلیت شمارش رو به بالا و پایین برای نگهداری نما در طراحی می‌تواند حجم مدار را کاهش دهد.

نتایج مورد انتظار

در این آزمایش، جمع یا تفريط دو عدد دودویی ممیز شناور با فعال شدن سیگنال Start محاسبه می‌شود. انتظار می‌رود نتیجه صحیح بعد از چند سیکل ساعت، بسته به تفاوت دو نما، همزمان با فعال شدن سیگنال End در خروجی مشاهده شود.

در ادامه، ابتدا الگوریتم اصلی را بررسی می‌کیم، سپس بخش‌های مختلف و جزئیات پیاده‌سازی آن‌ها را توضیح داده و در آخر به تست مدار در محیط Proteus می‌پردازیم.



۲ الگوریتم

راه حل کلی ما برای جمع یا تفریق اعداد ممیز شناور همان است که در کلاس معماری کامپیوتر ذکر شده است. تصویر زیر استراتژی مطرح شده در کلاس دکتر ارشدی را نشان می‌دهد:



Floating-Point Addition

۱ Align binary points

- Shift number with smaller exponent (**why?**)

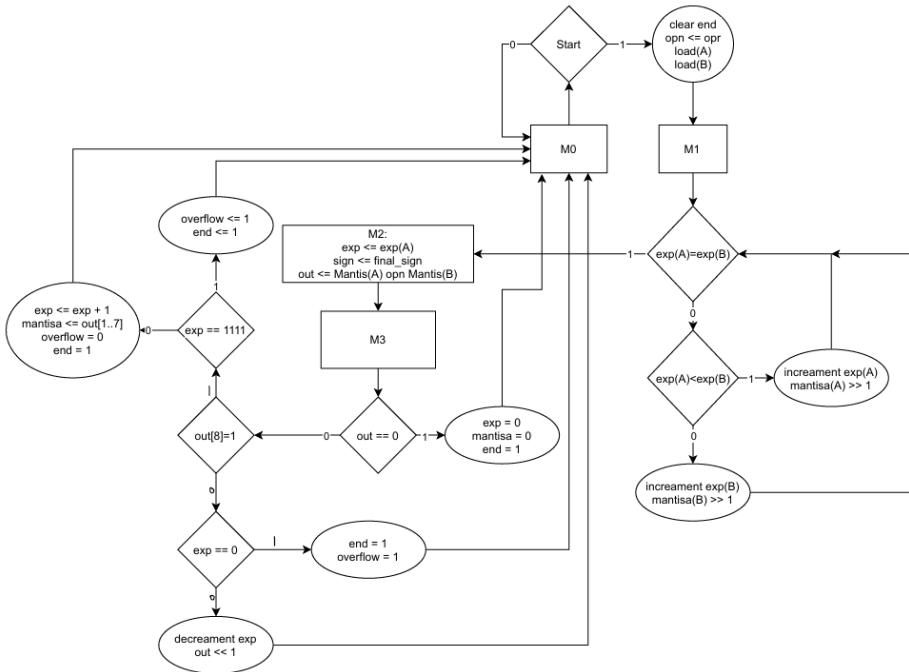
۲ Add significands

۳ Normalize result & check for over/underflow

۴ Round and renormalize if necessary



مدار دارای چهار حالت اصلی است، که در چارت صفحه بعد آورده شده‌اند.



مراحل مختلف به ترتیب به این شکل می‌باشند:

۱. M_0 : در این استیت منتهی سیگنال استارت می‌مانیم و زمانی که سیگنال استارت رسید ورودی‌ها را در رجیسترها متناظر لود می‌کنیم تا بتوانیم از آنها در آینده استفاده کنیم. بنابراین با استفاده از رجیسترها، تغییراتی که در وسط انجام محاسبات در ورودی‌ها داده می‌شوند روی جواب تاثیری نمی‌گذارند.

۲. M_1 : برای اینکه بتوانیم دو عدد ممیز شناور را با یکدیگر جمع کنیم و یا از هم کم کنیم باید توان برابر داشته باشند. برای همین عددی که توان کمتر دارد را انقدری شیفت می‌دهیم و به توان آن می‌فرماییم تا توانها یکسان شوند (عددی که توان بیشتر دارد بازرسش‌تر است).

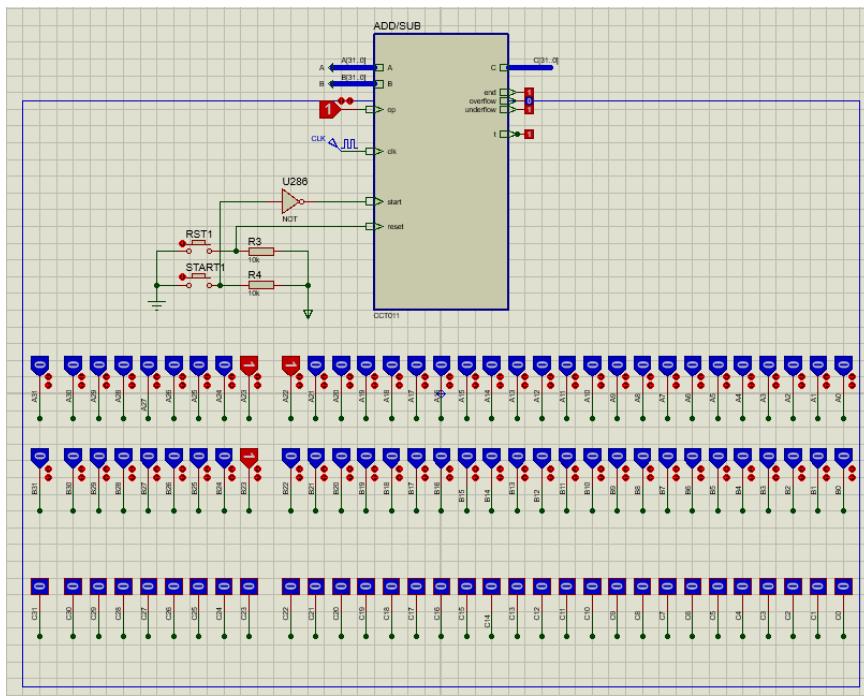
۳. M_2 : زمانی که توان‌ها برابر شدند با توجه به اینکه چه عملیاتی باید انجام شود (با توجه به اندازه عددها و علامت آنها و عملیات درخواست شده) عملیات را روی مانتیس‌ها انجام می‌دهیم و توان برابر شده در مرحله قبل خواهد بود.

۴. M_3 : ممکن است که جواب تولید شده نرمال نباشد که باید نرمال شود. برای این کار تا جایی که پرازش‌ترین بیت مانتیس یک شود، از توان کم می‌کنیم و مانتیس را به چپ شیفت می‌دهیم. البته اگر در این حین اکسپوننت صفر شود و هنوز بیت پرازش صفر باشد، یعنی underflow داریم.



۳ جزئیات پیاده‌سازی و توضیحات

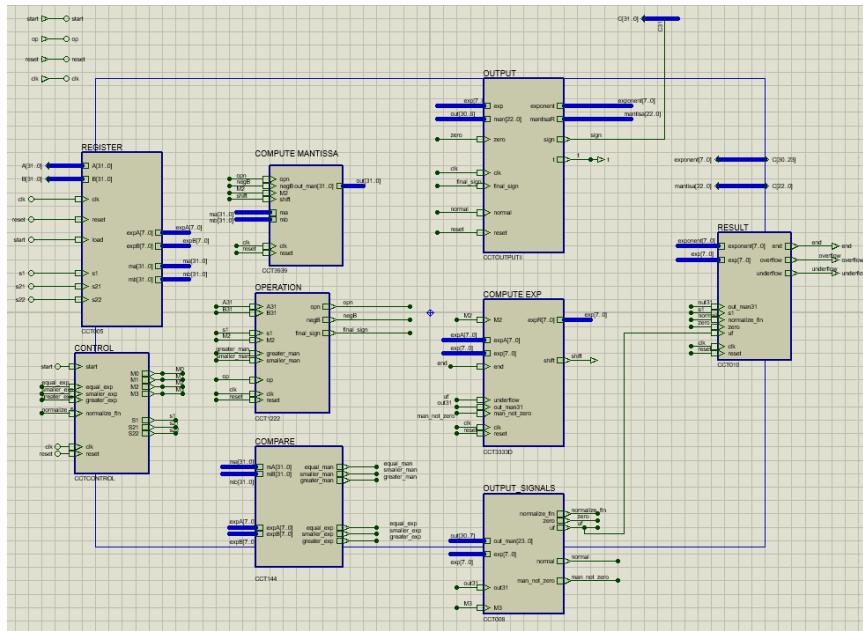
نمای کلی به صورت زیر می‌باشد:



در اینجا دکمه‌های استارت و ریست به صورت پوش باتن پیاده‌سازی شده‌اند و ورودی و خروجی‌ها همانند دستور کار قابل مقداردهی و مشاهده می‌باشند.



این مدل، متشکل از بخش‌های مختلفی است که در کنار هم عملیات مورد نظر را انجام می‌دهند:

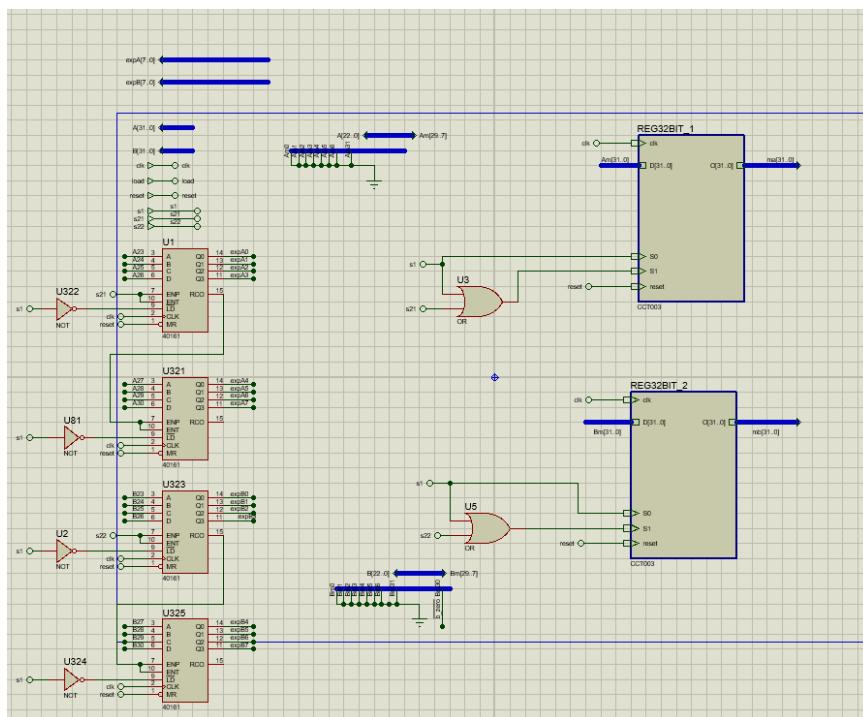


هر کدام از این مراحل‌ها با دقت بیشتری مورد بررسی قرار خواهد گرفت.



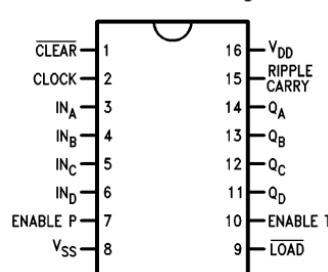
Register ۱.۳

این واحد مسؤولیت جدا کردن و نگه داشتن بخش توان، و همچنین بخش مانتیس را بر عهده دارد.



برای نگه داشتن توان‌ها از چهار واحد استفاده می‌کنیم که دو شمارنده را تشکیل می‌دهند. با این کار توانی که کمتر است را به توان بزرگ‌تر نزدیک می‌کنیم و به جای آن مانتیس را کوچک‌تر می‌کنیم، چراکه برای انجام جمع یا تفریق نیاز هست که توان‌ها یکی باشند. ساختار این آی‌اسی به صورت زیر می‌باشد:

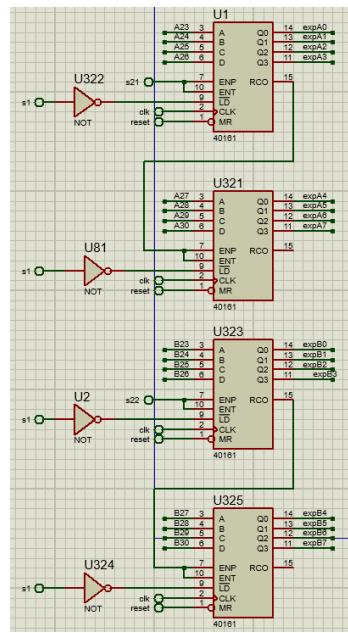
Dual-In-Line Package



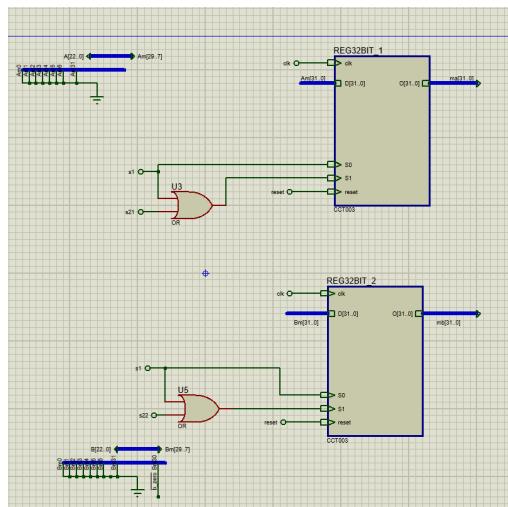
دو ورودی هر کدام می‌باشد که ent و enp همان carry out را هم فعال می‌کند. بنابراین هردو این ورودی‌ها را به ورودی کنترلی مناسب متصل می‌کنیم. همچنین هنگامی که استارت بزنیم و در حالت اول باشیم باید ورودی‌ها لود شوند. توجه داشته باشید که اگر سیگنال s21 فعال باشد، یعنی در حالت M1 هستیم و توان A کمتر از B است. در نتیجه توان A باید یک واحد افزایش یابد، همچنین مانتیس آن باید تقسیم بر دو شود، یعنی یک بیت به راست شیفت بخورد (بنابراین باید شمارندهای متناظر با توان A فعال شوند). برای همین ورودی‌های دو شمارنده اول S21 فعال باشد یعنی همین اعمال باید برای عملوند دوم انجام شود. بنابراین



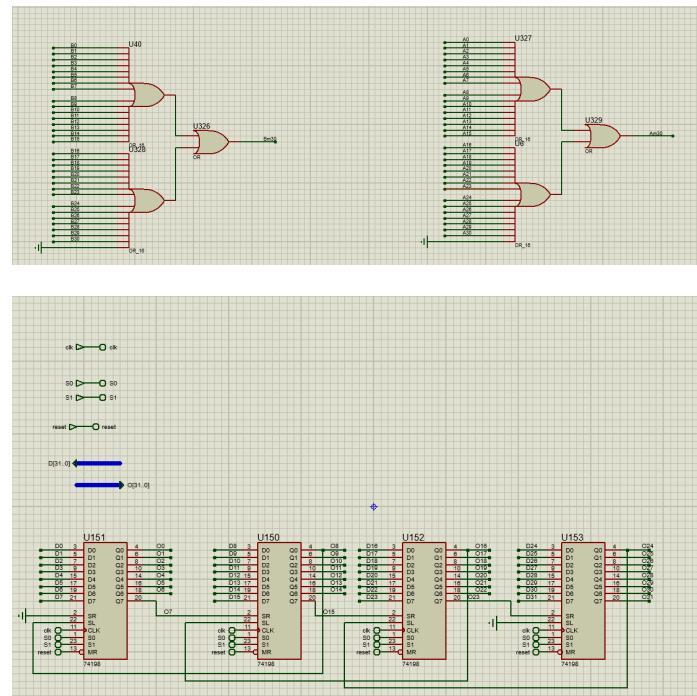
ورودی‌های چهار قطعه شمارنده تعیین می‌شوند:



حال به سراغ مانیپس‌ها می‌رومیم. در این پیاده‌سازی، تعداد بیت‌های مانیپس را ۳۲ در نظر گرفتیم. بنابراین نیاز است تا ۲۳ بیت ورودی - که پردازش هستند - در جای خود مقداردهی شوند و بقیه بیت‌ها نیز با صفر گسترش داده شوند. البته بیت شماره ۳۰ تمام بیت‌های ورودی است که مشخص می‌کند عدد ورودی صفر است یا نه. پس تا اینجا چنین سیستمی برای ذخیره مانیپس‌ها داریم:



همان طور که مشخص است برای ذخیره بخش مانیپس از یک مازول به نام *REG32BIT* استفاده شده است که ساختار آن به صورت زیر می‌باشد:



از چهار آی‌اسی 74198 استفاده شده است که هر کدام ۸ بیت ماتیس را پوشش می‌دهند. عملکرد این قطعه به صورت زیر می‌باشد:

MODE SELECT TABLE				
INPUTS				RESPONSE
M̄R	OP	S0*	S1*	
L	X	X	X	Asynchronous Reset; Outputs = LOW
H	/	H	H	Parallel Load; P0 → Q0
H	/	L	H	Shift Right; DSR → Q0, Q0 → Q1, etc.
H	/	H	L	Shift Left; DSL → Q7, Q7 → Q6, etc.
H	X	L	L	Hold

*Select Inputs should be changed only while CP is HIGH

H = HIGH Voltage Level

L = LOW Voltage Level

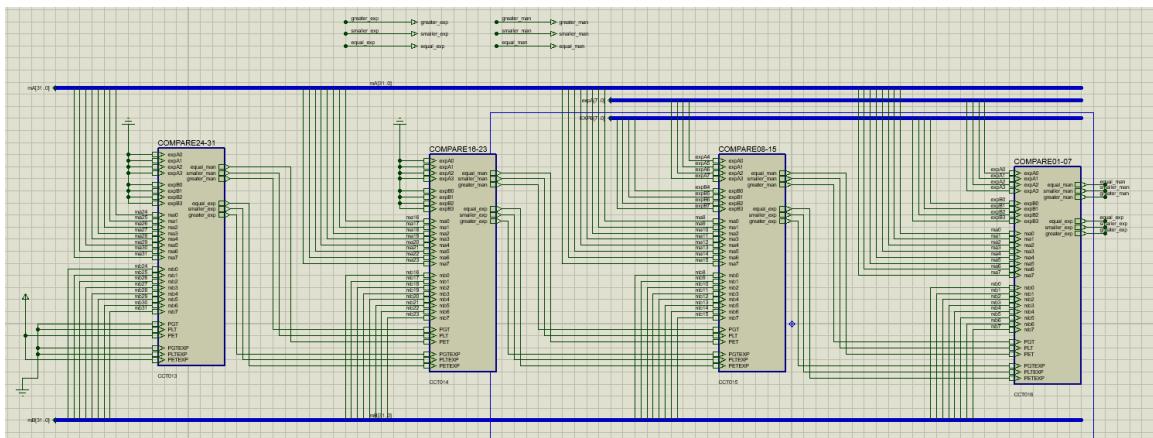
X = Immaterial

با توجه به جدول بالا و ورودی‌های S0 و S1 در شکل ۱.۳، مشخص است که اگر S1 فعال باشد - که سیگنال لود اولیه می‌باشد - دیتا از ورودی‌ها لود می‌شود و اگر قسمت اکسپوننت را افزایش داده‌ایم، در اینجا دیتا باید به راست شیفت بخورد. واحدهای متواالی نیز به صورت ripple متصل می‌شوند تا عملیات شیفت به صورت یکنواخت انجام شود. بدین صورت بخش Register به طور کامل بررسی شد.



Compare ۲.۳

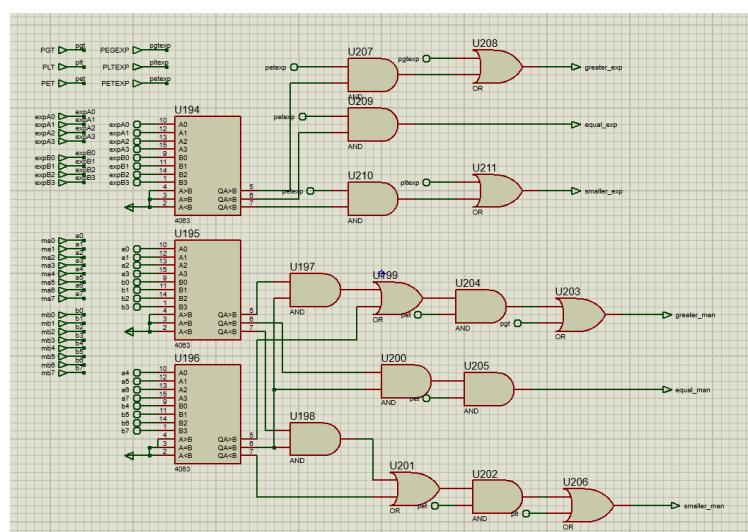
این بخش از خروجی‌های register استفاده می‌کند تا مانتیس و توان دو عدد ورودی را با هم مقایسه کند. در پیاده‌سازی این بخش از چهار قطعه مقایسه‌گر خاص استفاده شده که به صورت متوالی به هم متصل شده‌اند:



هر واحد مقایسه‌گر شش ورودی دارد که از بیت‌های خروجی مقایسه‌گر قبلی گرفته شده‌اند و باعث می‌شوند ارزش مکانی حفظ شود. این ورودی‌ها به صورت زیر هستند:

پیت‌های پارازش‌تر ورودی ma بزرگتر از mb بوده است. PLT, PET به همین صورت برای کوچکتر یا مساوی بودن بیت‌های مانتیس A می‌باشند. $.PGTEXP, PLTEXP, PETEXP$ به همین صورت برای بخش اکسپوننت این ورودی‌ها را داریم. عادی می‌باشد با این تفاوت که در هر واحد دو بار عمل مقایسه انجام می‌شود. ساختار درونی هر واحد مقایسه‌گر نیز مانند یک *Comparator* می‌باشد با این تفاوت که در هر واحد نتیجه فعلی *overwrite* می‌شود. یک بار برای مانتیس و یک بار برای اکسپوننت. همچنین اگر بیت‌های پارازش‌تر مساوی نباشند نتیجه قبلی نتیجه فعلی را *overwrite* می‌کند.

ساختار هر کدام از واحدها به صورت زیر است که در آن از واحدهای 4063 به عنوان مقایسه‌گر بیس استفاده شده است:





Control ۳.۳

سیگنال‌های کنترلی در این قسمت تولید می‌شوند. ابتدا، با توجه به نمودار و با استفاده از روش‌های ساده‌سازی همچون جدول کارنو، برای مراحل مختلف عملیات به حالت‌های زیر می‌رسیم:

$$M_0^+ = \overline{(M_0 + M_1 + M_2 + M_3)} + \overline{Start}M_0 + NormalizeFin$$

$$M_1^+ = M_0Start + (equal\ exp)M_1$$

$$M_2^+ = M_1(equal\ exp)$$

$$M_3^+ = M_2 + M_3NormalizeFin$$

این حالت‌ها در سراسر مدار مورد استفاده قرار می‌گیرند و سنکرون هستند، بنابراین آن‌ها را در یک شیفت‌رجیستر 74198 ذخیره می‌کنیم که در هر سیگنال کلاک لود می‌کند.

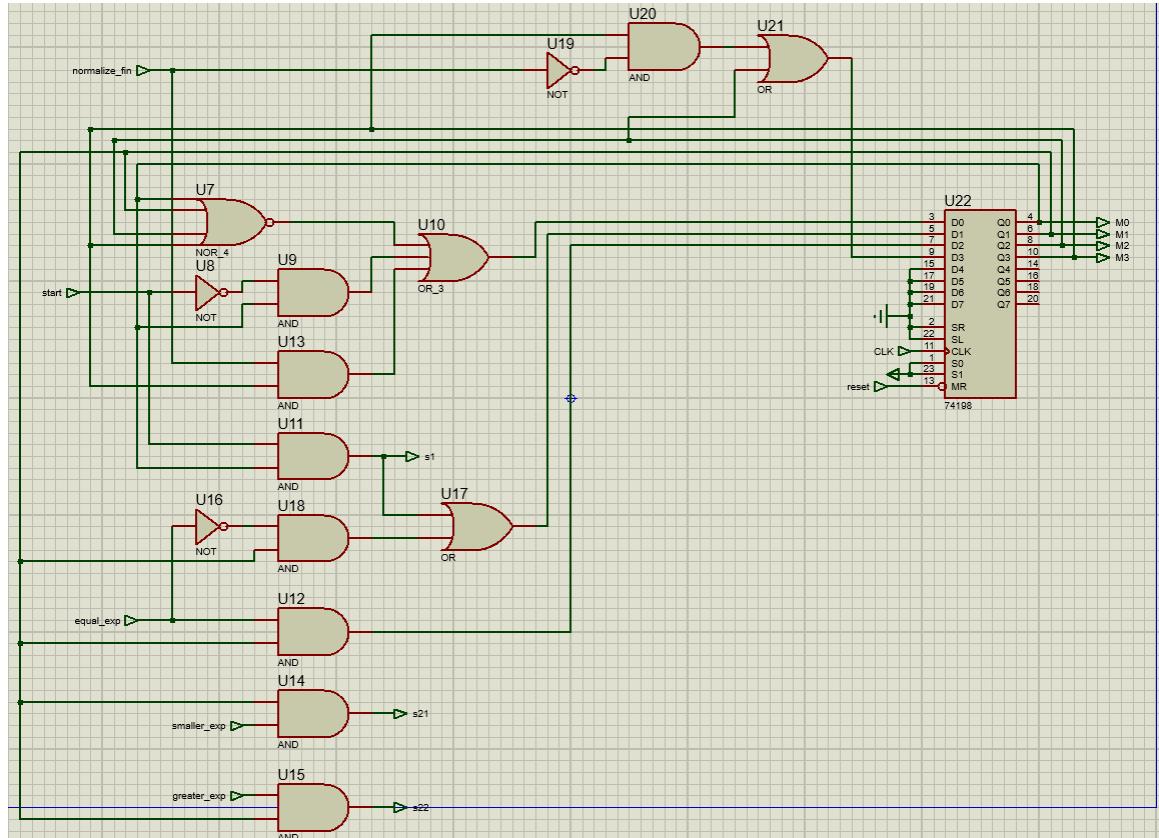
همچنین یک سری سیگنال ورودی نیز برای قسمت *operation register* و *register* که در اینجا تولید می‌شوند:

$$s_1 = M_0Start$$

$$s_{21} = exp\ biggerM_1$$

$$s_{22} = exp\ smallerM_1$$

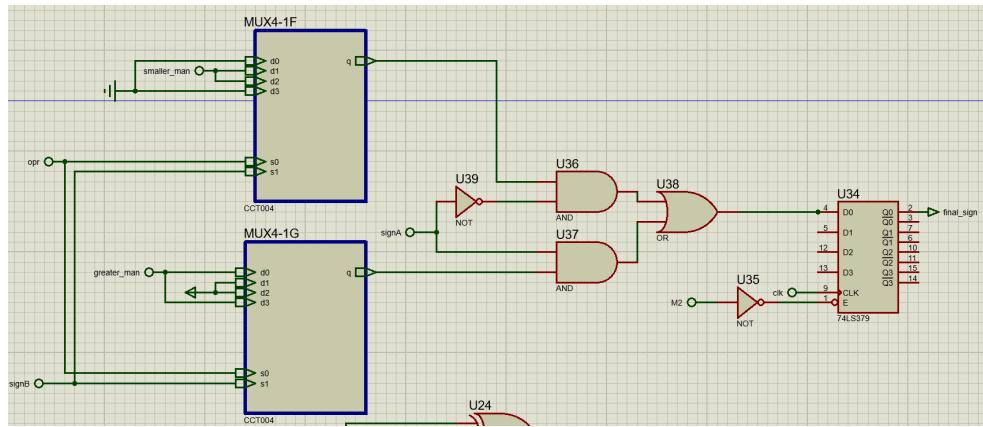
در واقع مشخص می‌کند که عملیات شروع شده است و $S21$ و $S22$ مشخص می‌کنند کدام یک از توان‌ها کوچک‌تر است و باید به دیگری برسد.



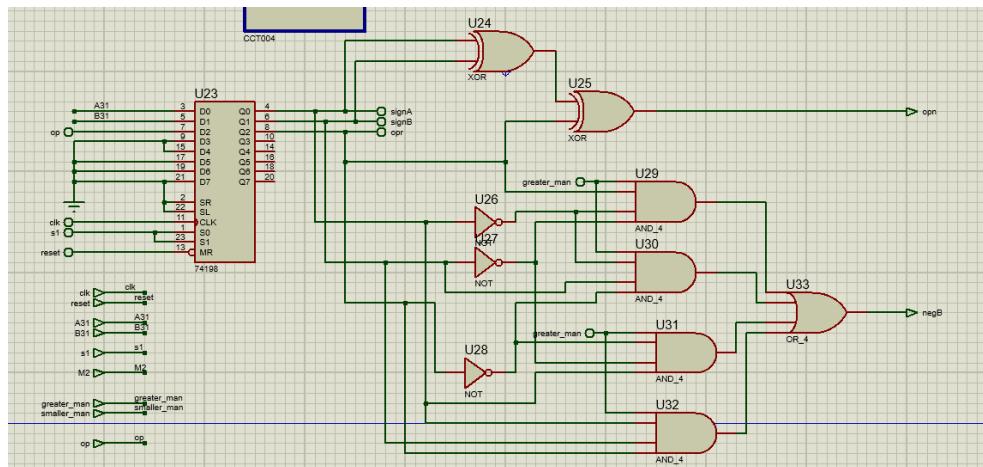


Operation ۴.۳

این بخش دو وظیفه بر عهده دارد، یکی تولید سیگنال‌های ورودی برای Compute Mantisa، و دیگری محاسبه علامت جواب نهایی. با استفاده از مدار زیر به دست می‌آید:



برای اطمینان از درستی این قسمت می‌توان جدول درستی کشید یا حالت‌های مختلف را بررسی کرد. برای مثال هنگامی که در حال جمع کردن باشیم، عدد A مثبت باشد و B هم منفی باشد و $A < B$ نتیجه منفی و بیت علامت یک خواهد شد.

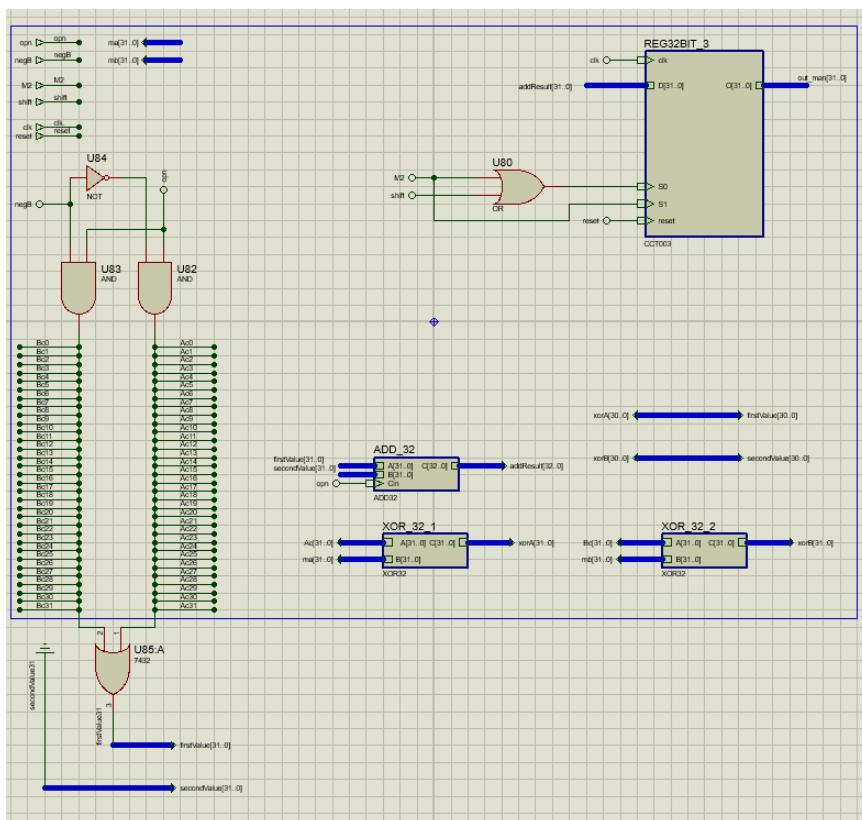


برای مشخص کردن نوع عملیات انجام شده - op - کافی است تا از سه ورودی تاثیر گذار xor , opr , $signB$, $signA$ و opr بگیریم. این عملوندها در یک شیفت رجیستر 74198 ذخیره شده‌اند که ورودی‌ها را ثابت نگه می‌دارد. $negB$ هم مشخص می‌کند در صورت فعال شدن op , $open$ ، بهتر است B قرینه شود یا A . در واقع تضمین می‌کند که خروجی عملیات همواره مثبت باشد و تنها با استفاده از $final\ sign$ علامت جواب نهایی مشخص شود.



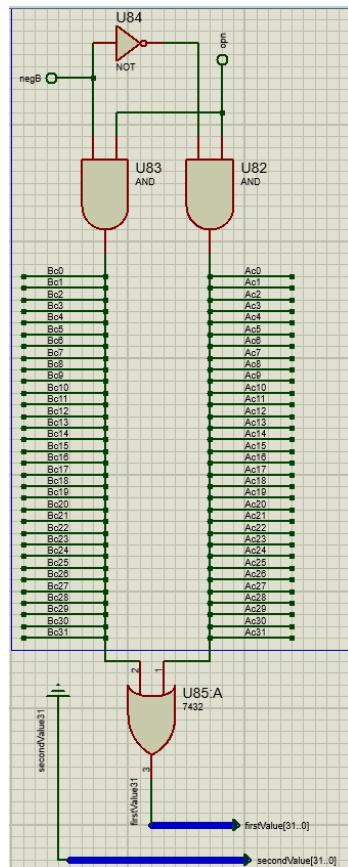
Compute Mantisa ۵.۳

این بخش مسئول انجام عملیات حسابی مانتیس‌ها، ذخیره جواب و در صورت نیاز، شیفت دادن آن می‌باشد. ساختار کلی آن در زیر آمده است:

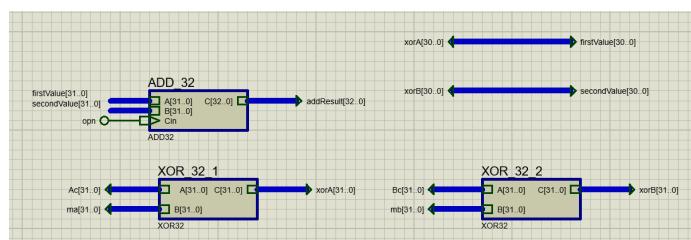


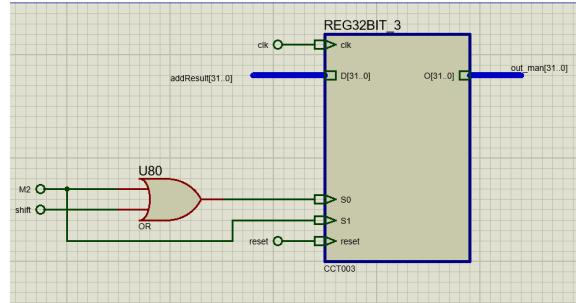


در این بخش قبل از جمع کردن دو عملوند، ابتدا هر کدام از آن‌ها را با دو عدد Bc و Ac xor می‌گیریم، در واقع این کار باعث می‌شود تا اگر قرار است تفاضل کنیم یا یکی از عدها منفی می‌باشد، این در محاسبات لحاظ شود و عدد مورد نظر قرینه شود.



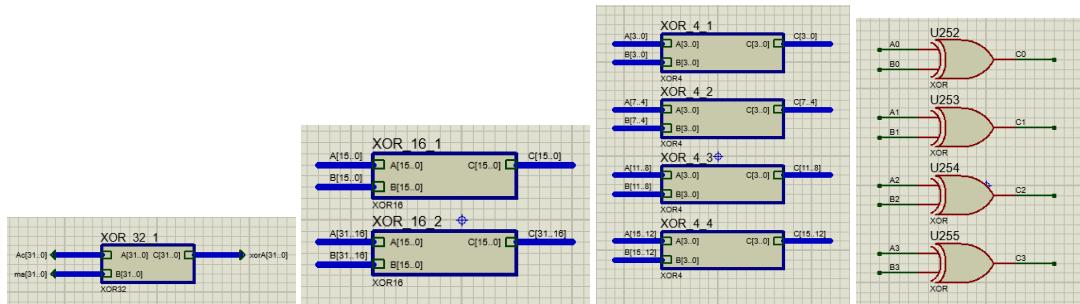
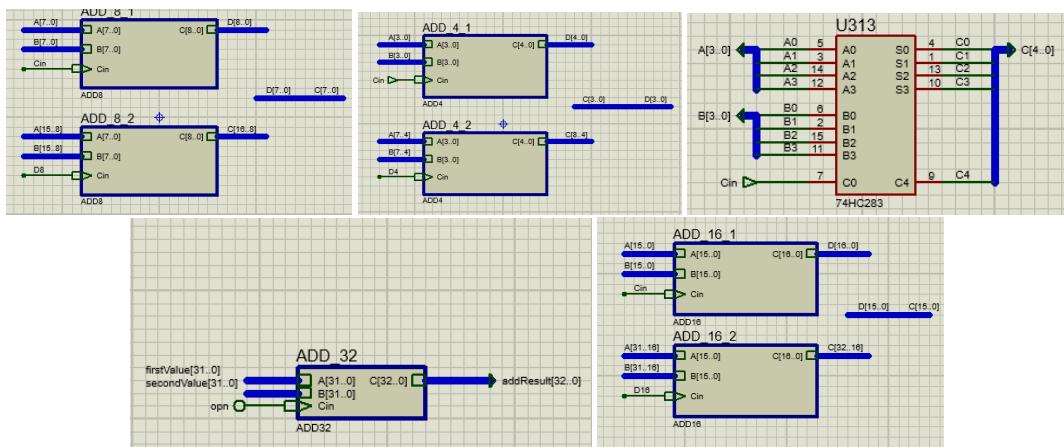
همچنین ورودی‌های کنترلی این بخش *opm* و *negB* می‌باشند. اگر *opm* یک باشد، یعنی باید برای دو عدد مثبت عمل تفاضل انجام شود. مشخص می‌کند برای این کار *B* باید مکمل شود یا *A*. از بیت آخر نیز برای بررسی *overflow* استفاده می‌شود. حال که عملوندها آماده شده‌اند جمع را انجام می‌دهیم و خروجی در *add result* قرار می‌گیرد:





برای ذخیره و شیفت جواب، از یک واحد *REG32BIT* استفاده می‌کنیم که عملکردش در بخش قبلی بررسی شد. ورودی‌های کنترلی به این صورت هستند که اگر در مرحله محاسبات $M2 = 1$ باشد، خروجی جمع کننده در شیفت رجیستر لود شود. همچنین اگر ورودی *shift* باشد، یعنی در حال نرمال‌سازی هستیم و جواب باید به چپ شیفت بخورد.

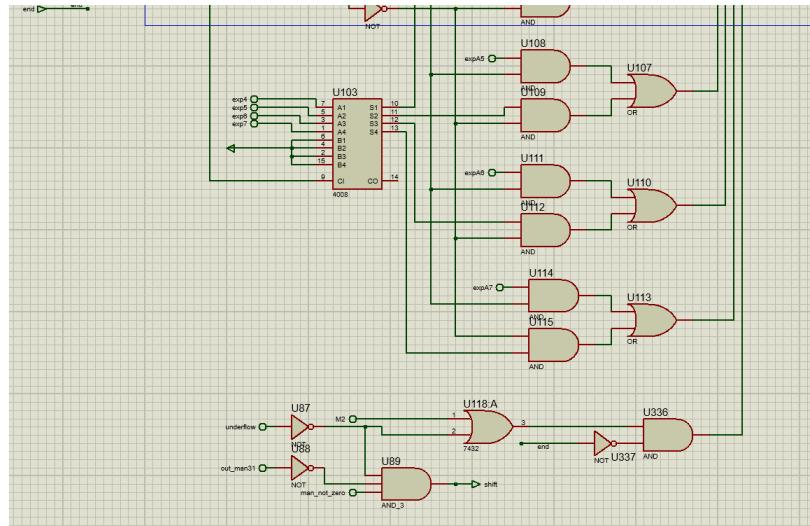
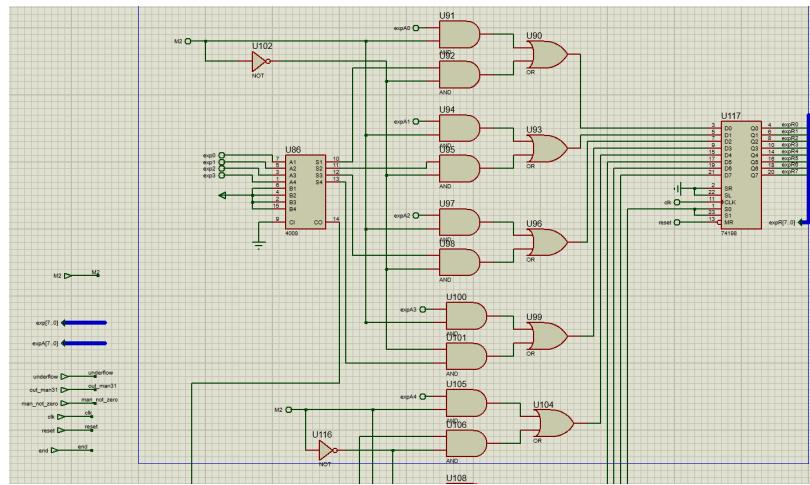
در آخر نیز، جمع کننده ۳۲ بیتی و xor ۳۲ بیتی نیز به صورت زیر، و با کار هم قرار دادن تعدادی از واحدها کنار هم ساخته می‌شوند:
(برای ساخت کوچک‌ترین واحد جمع کننده از یک واحد 74283 استفاده شده است که یک جمع کننده چهار بیتی می‌باشد)





Compute Exp ۶.۳

این بخش به صورت زیر می‌باشد:

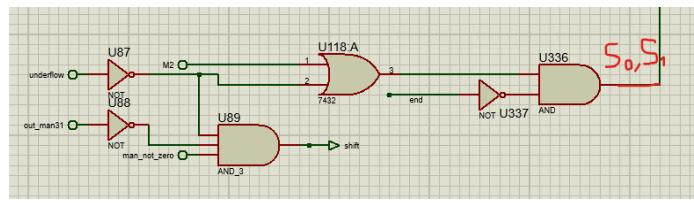


کاربرد این بخش، نگه داری از بخش اکسپوننت جواب و در صورت نیاز، نرمال سازی آن می‌باشد. مرحله M2 وقتی هست که مقدار مانتیس جواب مشخص می‌شود. بنابراین در این مرحله باید بخش اکسپوننت جواب را هم لود کنیم. از آنجایی که اکسپوننت نهایی برابر با توان A می‌باشد (اگر توان A کمتر باشد، زیاد می‌شود و اگر بیشتر از B باشد، B زیاد می‌شود و در نهایت توان A همان توان جواب نهایی می‌باشد) آن را در مرحله A لود می‌کنیم.

در صورتی که underflow داشته باشیم نیاز هست تا وقتی که این فلگ فعال باشد و سیگنال end هم صفر باشد، از توان کم کنیم و مانتیس را به چپ شیفت دهیم. برای قسمت توان، کافی است توان فلی را با 1111 که مکمل دوی عدد یک می‌باشد جمع کنیم (در واقع یک واحد از توان کم می‌شود و آن را در رجیستر لود کنیم. این کار با استفاده از دو واحد 4008 که جمع کننده چهاربیتی است انجام می‌شود. برای قسمت مانتیس، کافی است سیگنال shift که در بخش Compute Mantisa استفاده می‌شود و مانتیس را به چپ شیفت می‌دهد، فعال شود. پس تا وقتی underflow داریم، مانتیس صفر نیست و بیتی به ارزش یک را از دست نمی‌دهیم - یعنی بیت آخر یک نیست - سیگنال shift باید



فعال باشد، که با استفاده از یک گیت *and* با سه ورودی پیاده‌سازی شده است. اکسپوننت نیز در یک واحد 74198 نگه داری می‌شود که عملکردش قبل از آنچایی که هرگز شیفت نمی‌دهیم ورودی‌های *sr* و *sl* مهم نبیستند و *s0* و *s1* نیز هم‌مان به بخش کنترلی متصل‌اند که در صورت فعال شدن، ورودی‌ها را لود می‌کنند و در غیر این صورت خروجی‌های قبلی را ذخیره می‌کنند. در پایان این بخش، بخش کنترلی با توجه به توضیحات فوق به صورت زیر ساخته می‌شود:



Output Signals ۷.۳

این بخش، ۵ سیگنال وضعیت را تولید می‌کند:

۱. استفاده می‌شود و باعث می‌شود فقط وقتی مانتیس را شیفت نمی‌کند. *Compute Exp man not zero*.

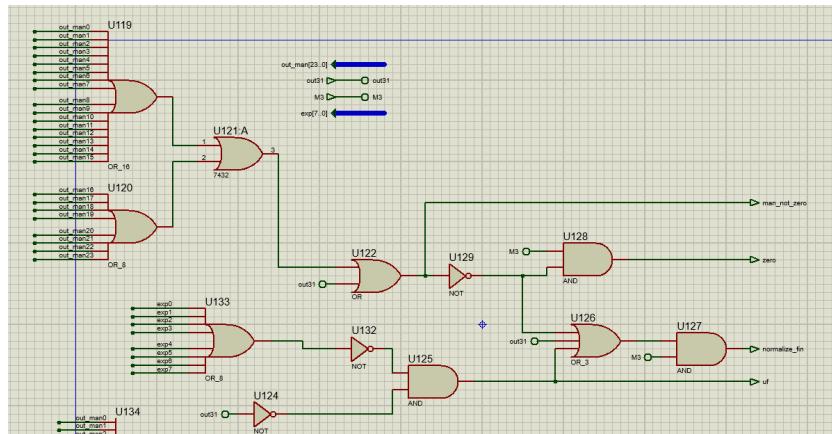
۲. *uf*: سیگنال *underflow* می‌باشد که در دو بخش *Compute Exp* و *Result* استفاده می‌شود. وقتی داریم که توان کاملاً صفر شده باشد اما *out31* - بیت آخر جواب که نقش یک در نمایش استاندارد ۰...۰۱۰۰۰ را دارد - هنوز صفر باشد. به عبارتی دیگر، نتوانیم با شیفت مانتیس به چپ، به نمایش استاندارد برسیم.

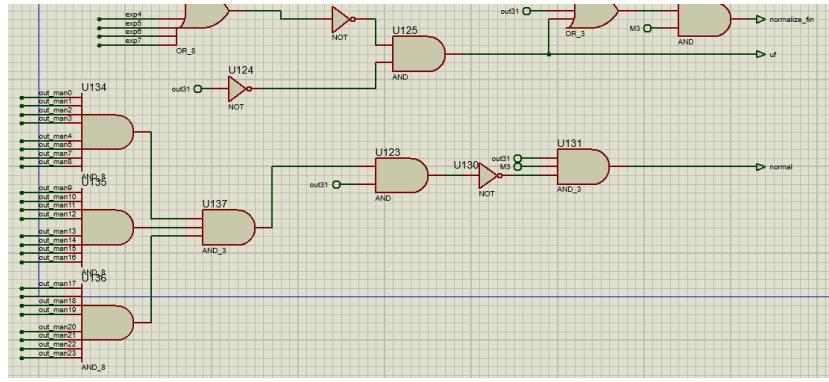
۳. در مازول *output normal* استفاده می‌شود. این سیگنال زمانی فعال می‌شود که در حالت نهایی باشیم و بیت شماره سی و یک خروجی فعال شده باشد و تمام مقادیر *out* یک نباشد.

۴. در مازول‌های *result* و *control* و *normalize fin* استفاده می‌شود. این خروجی زمانی فعال می‌شود که عملیات نرم‌افزاری تمام شده باشد، یعنی بیت شماره سی و یک مانتیس یک شده باشد، یا اصلاً جواب خروجی صفر باشد یا *underflow* رخ داده باشد. در هر صورت در اینجا دیگر جواب به دست آمده است.

۵. در *result* و *output zero* استفاده می‌شود. در صورتی ۱ می‌شود که حاصل مانتیس صفر باشد و در حالت نهایی - *M3* - باشیم.

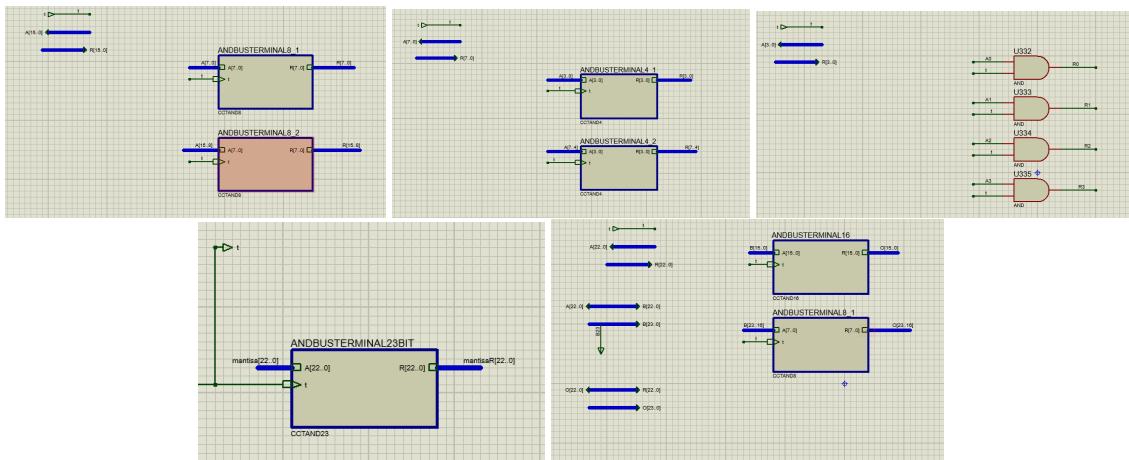
با توجه به توضیحات بالا، این بخش به صورت زیر پیاده‌سازی می‌شود:





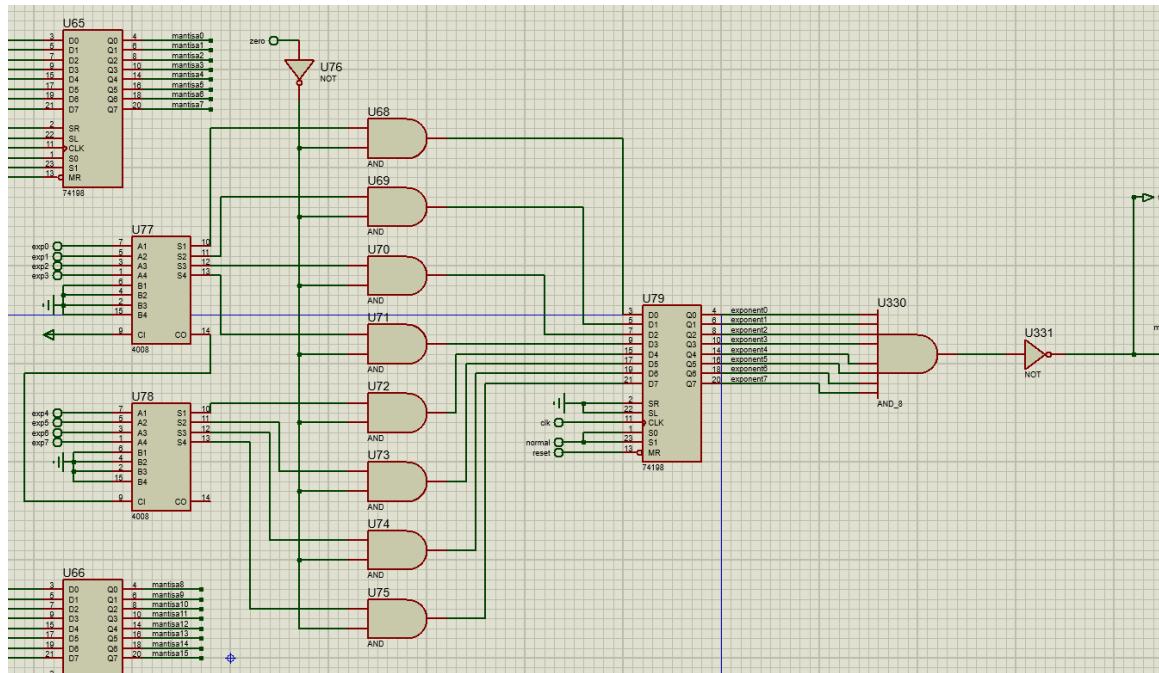
Output ۸.۳

این واحد خروجی نهایی را لود می‌کند. در صورتی که بیت zero فعال باشد، تمام بیت‌های خروجی باید صفر شود. همچنین هنگامی که normal فعال باشد، ورودی‌ها در رجیسترها 74198 لود می‌شوند. البته قبل از لود اکسپوننت‌ها، با استفاده از دو جمع کننده 4008 توان را ابتدا با یک جمع می‌کنیم. این کار به این دلیل است که توان برای اولین بار در M_2 داخل بخش Compute Exp لود می‌شود، و بعد از آن در هر کلاک به طور خودکار یک واحد کم می‌شود تا به نتیجه نهایی برسیم. در حالی که مثلاً در بهترین حالت، در یک کلاک بعدی که وارد حالت M_3 می‌شویم نتیجه حاصل شده و $normal$ فعال می‌شود. در پایان این کلاک، اکسپوننت یک واحد کمتر از مقدار واقعی است. سپس در ابتدای کلاک بعدی لود می‌شود و بنابراین اکسپوننت همیشه یک واحد کمتر از مقدار واقعی اش هست و باید با یک جمع شود. این اتفاق در مورد بخش مانتیس نمی‌فتد چراکه ورودی کنترلی آن مستقیم از بیت سی و یکم و فلگ‌هایی مثل $zero$ به دست می‌آید و سنکرون نیست. قبل از اعلام نتیجه نهایی، باید بررسی کنیم که آیا اکسپوننت بیشترین مقدار ممکن (11111111) را دارد یا خیر. این کار را با and گرفتن از تمام بیت‌های توان انجام می‌دهیم. با توجه به کاربرد آن در واحد بعدی، بهتر است از not آن استفاده کنیم؛ آن را t می‌نامیم. در نتیجه وقتی t صفر است یعنی اکسپوننت بیشترین مقدار را دارد. به عبارتی این کار مشخص می‌کند آیا خروجی بی نهایت هست یا خیر. از آنچهی که در دستور کار ذکر شده خروجی باید نرمالیزه باشد، حالت Nan نداریم و در حالت بی نهایت کافی است تمام بیت‌های مانتیس را صفر کنیم. این کار با استفاده از واحدی به نام $ANDBUSTERMINAL23BIT$ انجام می‌شود، که در واقع ۲۳ واحد دو ورودی، یک ورودی و ورودی دیگر بیت متناظر مانتیس می‌باشد. ساختار این واحد به صورت زیر است:

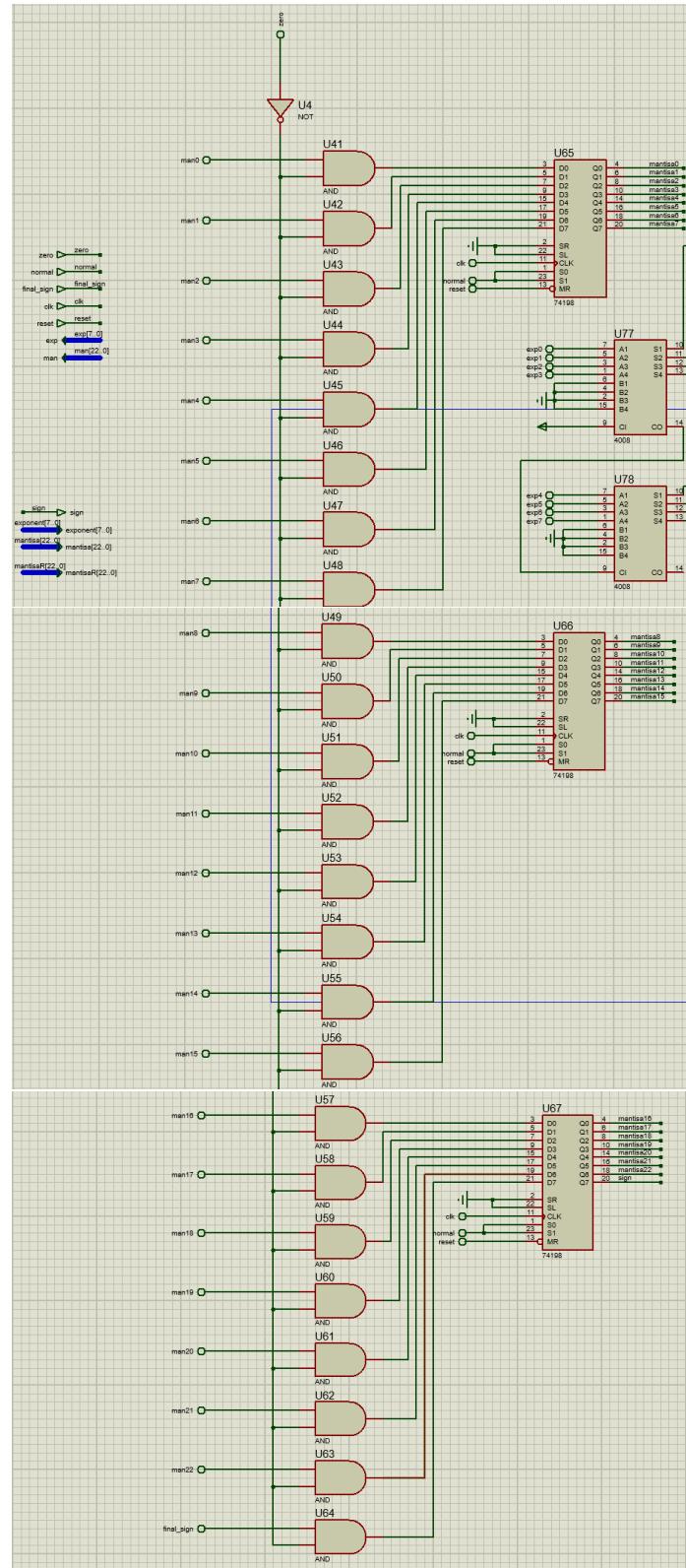




طبق توضیحات فوق، بخش توان به شکل زیر ساخته می‌شود:



همچنین بخش مانتیس نیز همانند مدل صفحه بعد ساخته می‌شود و به این ترتیب این مازول هم به طور کامل مورد بررسی قرار گرفت.





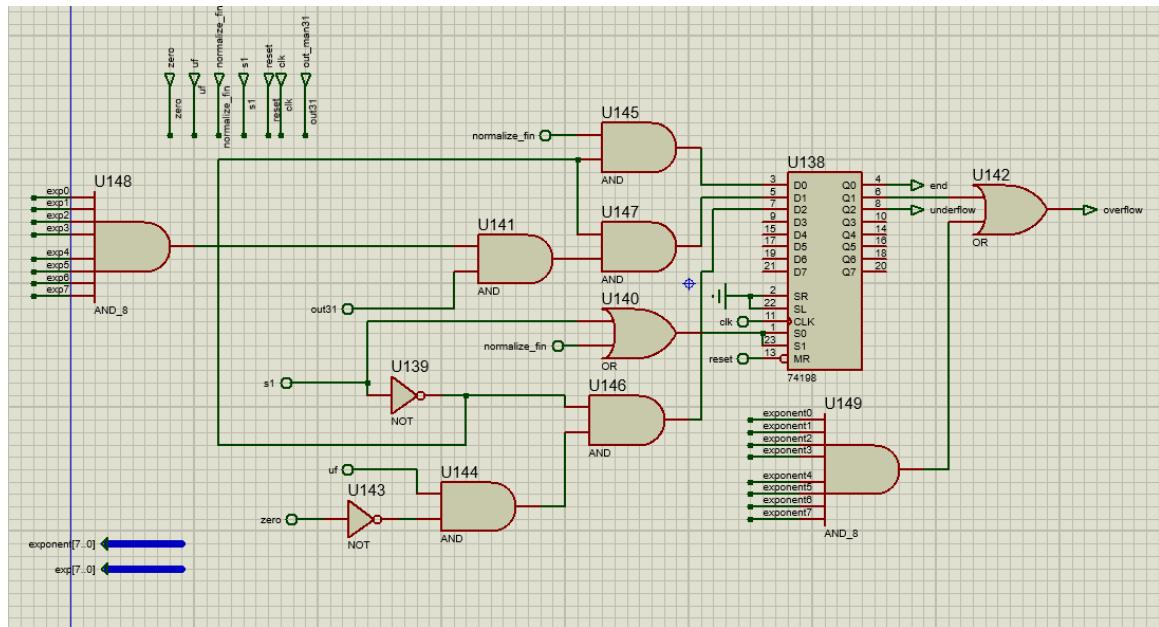
Result ۹.۳

وظیفه این مژول ساخت سه سیگنال مهم $underflow$, end , $overflow$ می‌باشد.

۱. end : موقعی به پایان عملیات می‌رسیم که محاسبه انجام شده باشد یعنی در حالت $M3$ باشیم؛ همچنین خروجی نرمال باشد، یعنی بیت سی و یکم آن یک باشد. البته ممکن است خروجی کلا صفر یا $underflow$ هم باشد که باید آن‌ها را در نظر گرفت. تمام این حالات در $normalize_fin$ بررسی شده‌اند و تنها کافی است اگر این سیگنال فعال باشد و در حالت درست باشیم این سیگنال فعال شود.

۲. $underflow$: اگر uf فعال باشد و عدد نهایی هم صفر نباشد، یعنی نمی‌توانیم خروجی را به درستی نشان دهیم و $underflow$ اتفاق می‌افتد. البته در صورتی که کاربر دوباره استارت زده باشد این نتایج ولید نیست و باید با $s1not$ بگیریم.

۳. $overflow$: وقتی اتفاق می‌افتد که آخرین بیت مانتیس یک باشد اما توان exp بیشترین مقدار ممکن را داشته باشد که ولید نیست؛ یعنی وقتی دیگر نتوانیم مانتیس را به چپ شیفت بدھیم. همچنین اگر بیت‌های توان نهایی ($exponent$) نیز ۱۱۱۱۱۱۱۱ باشند هم $overflow$ داریم. دلیل بررسی جداینه این دو حالت این می‌باشد که ممکن است هنگام جمع exp با یک، $Cout$ پاسخ خروجی تماماً صفر باشد؛ یعنی اگر exp $exponent$ یا در نظر بگیریم یک حالت را از دست خواهیم داد. همچنین برای نگه داری این سیگنال‌ها از یک واحد ۷۴۱۹۸ استفاده می‌کنیم. در نهایت شکل کلی این بخش هم به صورت زیر در می‌آید:





شبیه‌سازی و تست ۴

Test 1: +

Basic addition/subtraction of normalized floating-point numbers.

Input A: 3.5 (0x40600000)

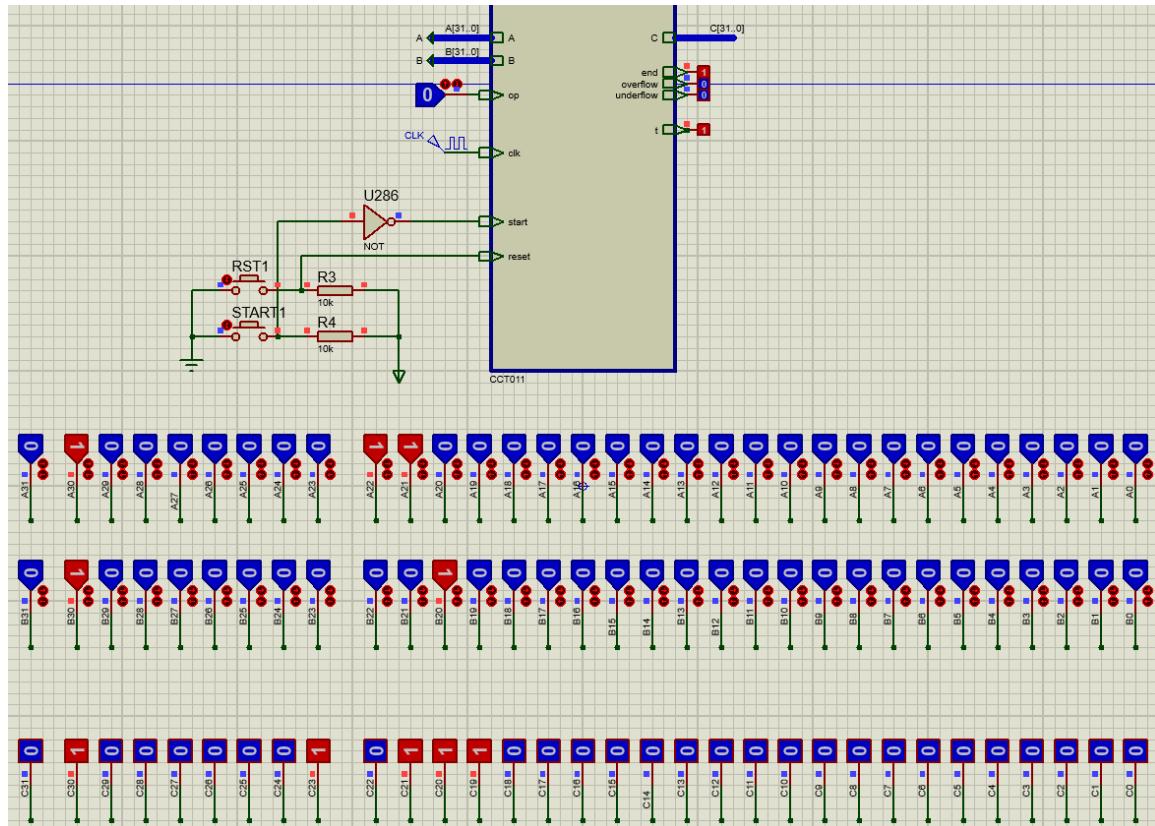
Binary: 0b01000000110000000000000000000000

Input B: 2.25 (0x40100000)

Binary: 0b01000000001000000000000000000000

Addition Output C: 5.75 (0x40B80000)

Binary: 0b01000001011100000000000000000000





Test 1: -

Basic addition/subtraction of normalized floating-point numbers.

Input A: 3.5 (0x40600000)

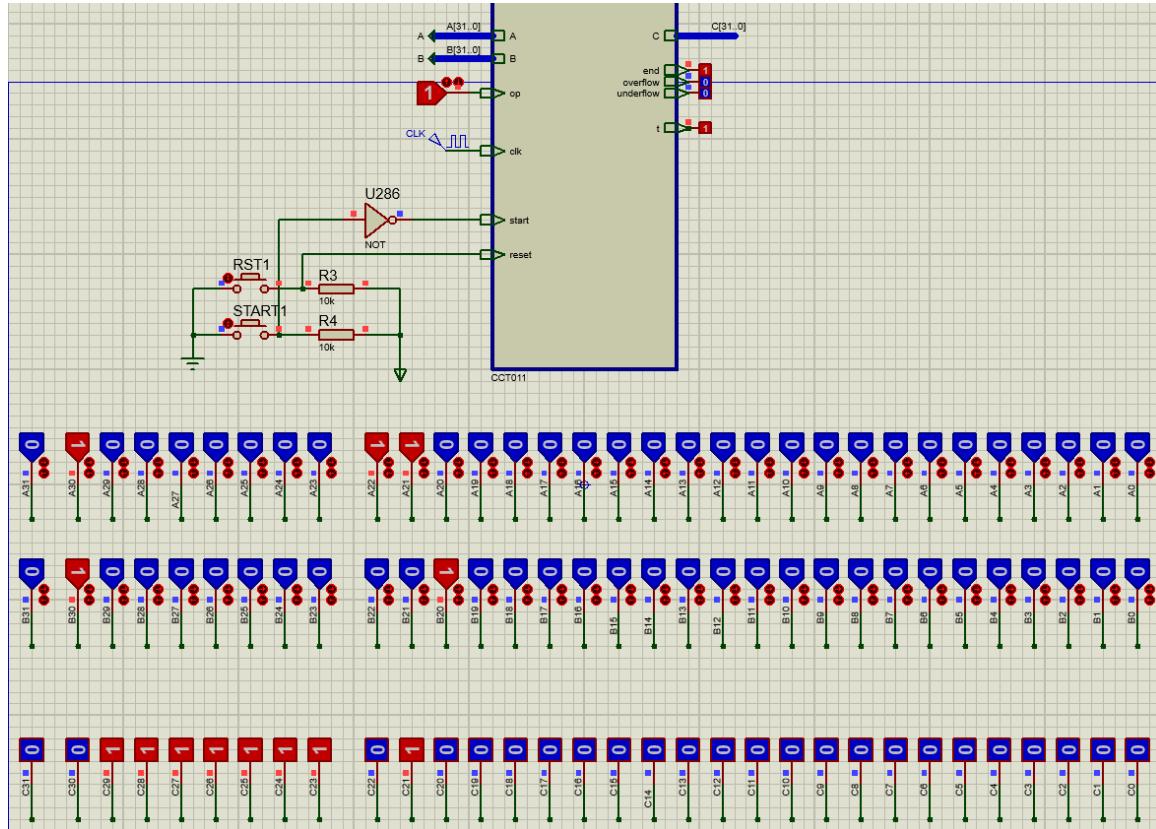
Binary: 0b01000000011000000000000000000000

Input B: 2.25 (0x40100000)

Binary: 0b01000000001000000000000000000000

Subtraction Output C: 1.25 (0x3FA00000)

Binary: 0b00111111010000000000000000000000





Test 2

Addition resulting in an overflow.

Input A: 3.4e38 (0x7F7FC99E)

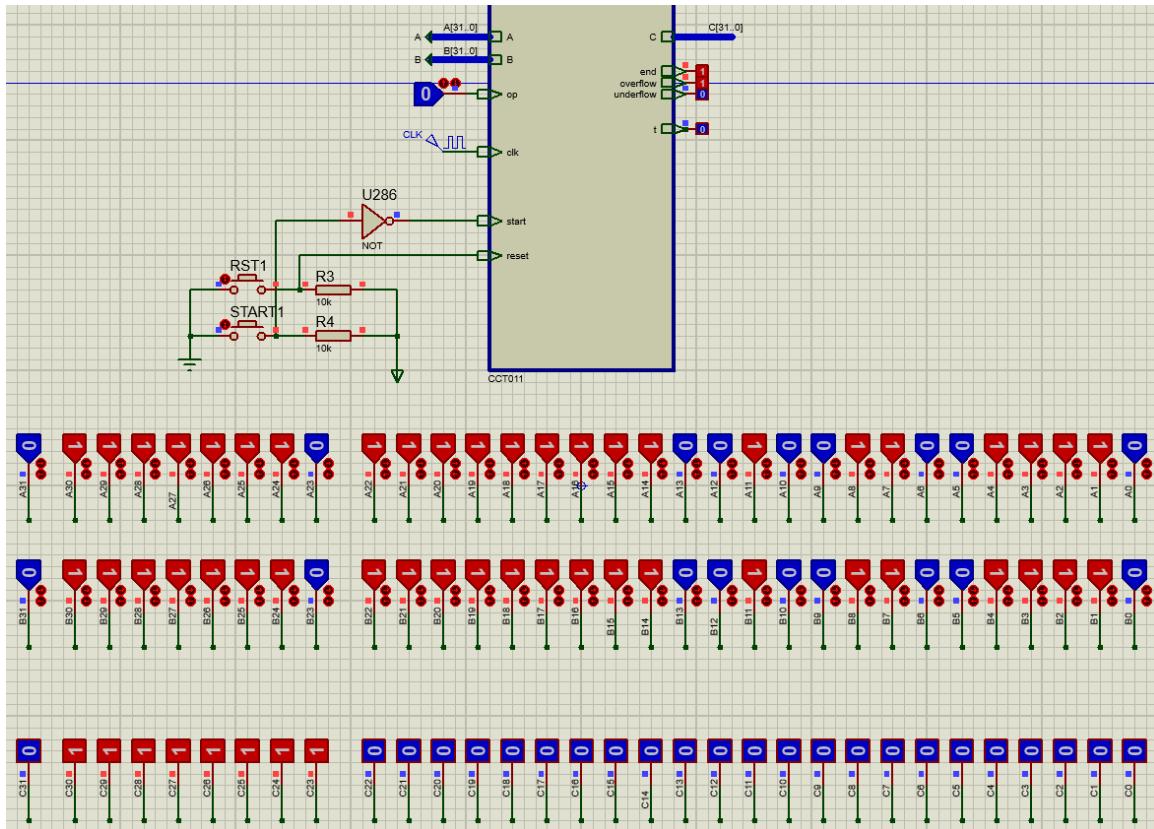
Binary: 0b01111110111111100100110011110

Input B: 3.4e38 (0x7F7FC99E)

Binary: 0b011111101111111100100110011110

Addition Output C: Overflow/Inf (0x7F800000)

Binary: 0b0111111100000000000000000000000000000000





Test 3

Subtraction resulting in an underflow.

Input A: 1.4e-45 (0x00000001)

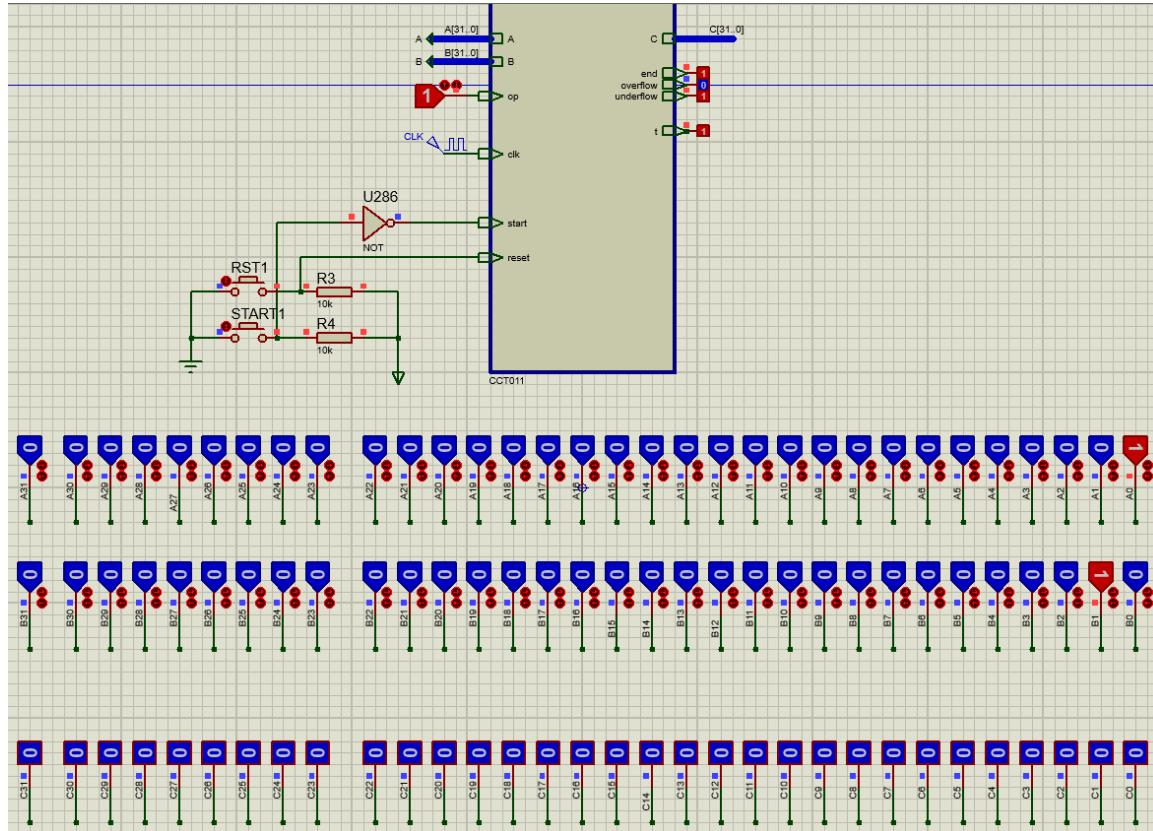
Binary: 0b00000000000000000000000000000001

Input B: 2.8e-45 (0x00000002)

Binary: 0b00000000000000000000000000000010

Subtraction Output C: Underflow

Binary: 0b00000000000000000000000000000000





Test 4

*Subtraction of very large numbers.***Input A:** 1.0e38 (0x7E967699)

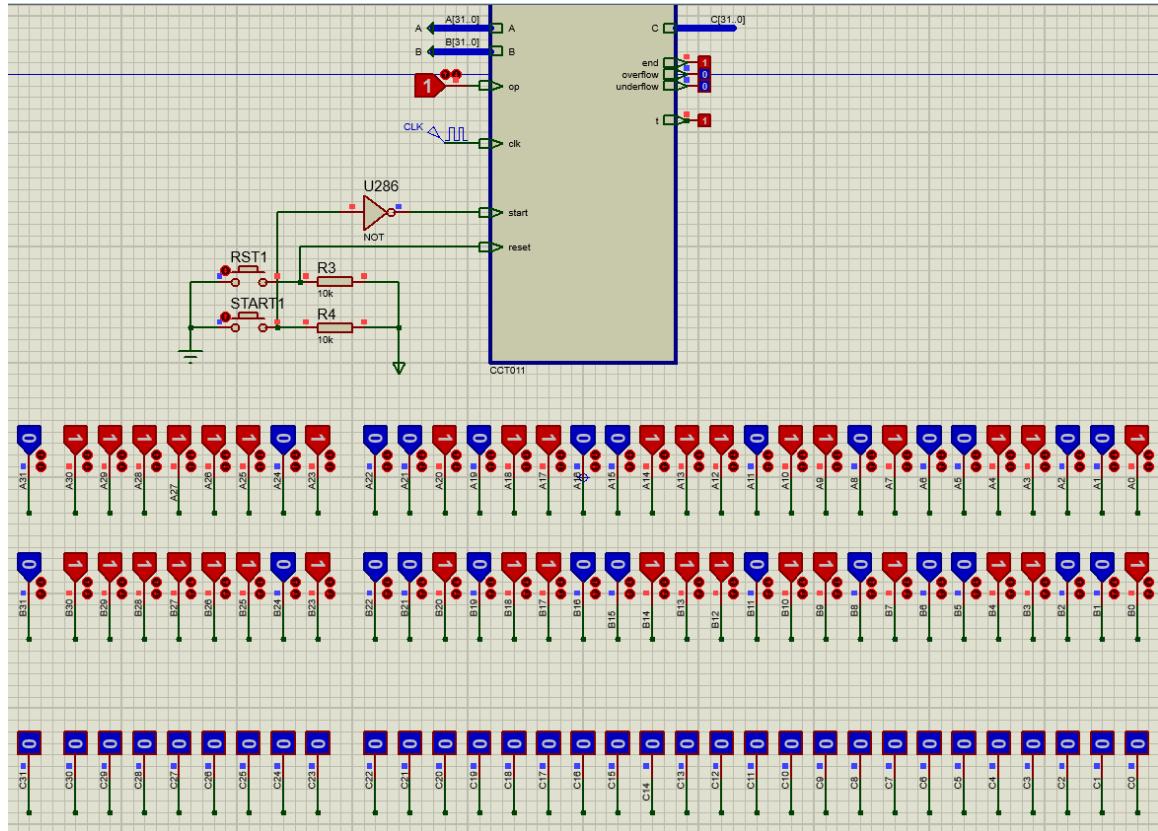
Binary: 0b0111110100101100111011010011001

Input B: 1.0e38 (0x7E967699)

Binary: 0b0111110100101100111011010011001

Subtraction Output C: 0 (0x00000000)

Binary: 0b000





Test 5: +

*Addition/subtraction involving zero.***Input A:** 0 (0x00000000)

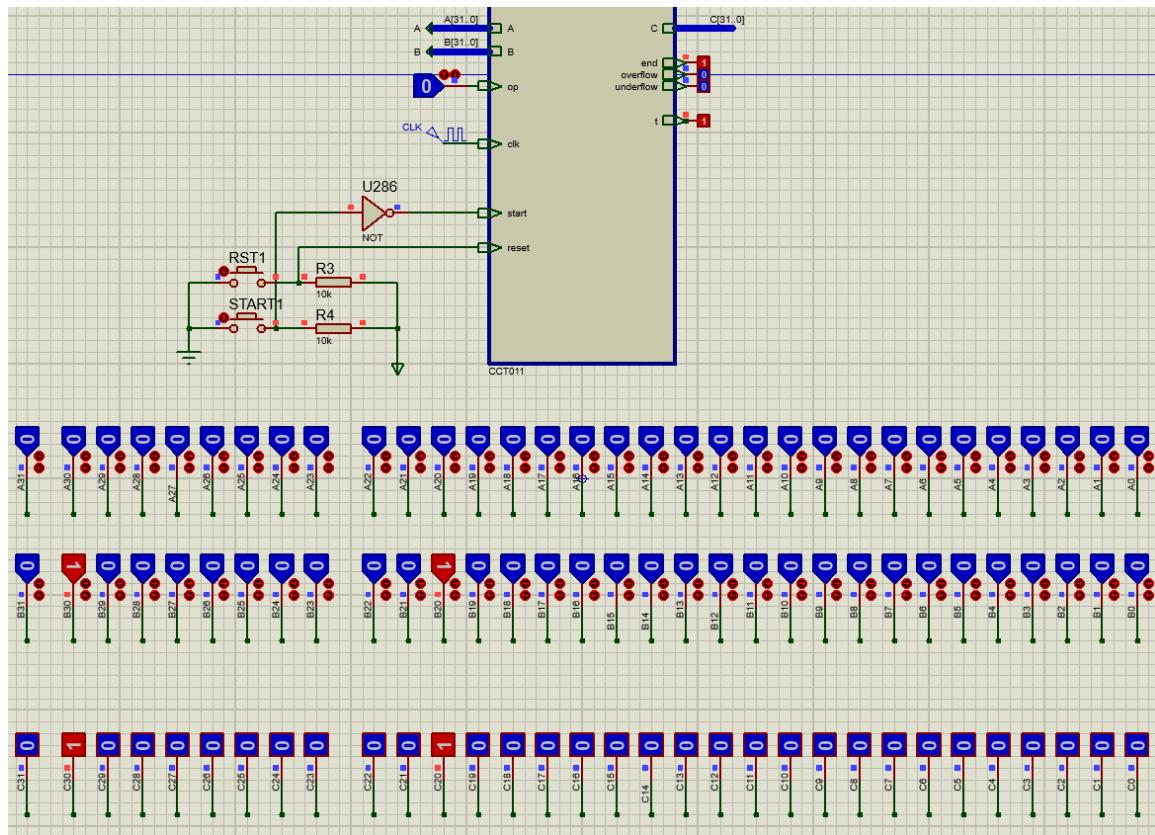
Binary: 0b00000000000000000000000000000000

Input B: 2.25 (0x40100000)

Binary: 0b01000000000100000000000000000000

Addition Output C: 2.25 (0x40100000)

Binary: 0b01000000000100000000000000000000





Test 5: -

*Addition/subtraction involving zero.***Input A:** 0 (0x00000000)

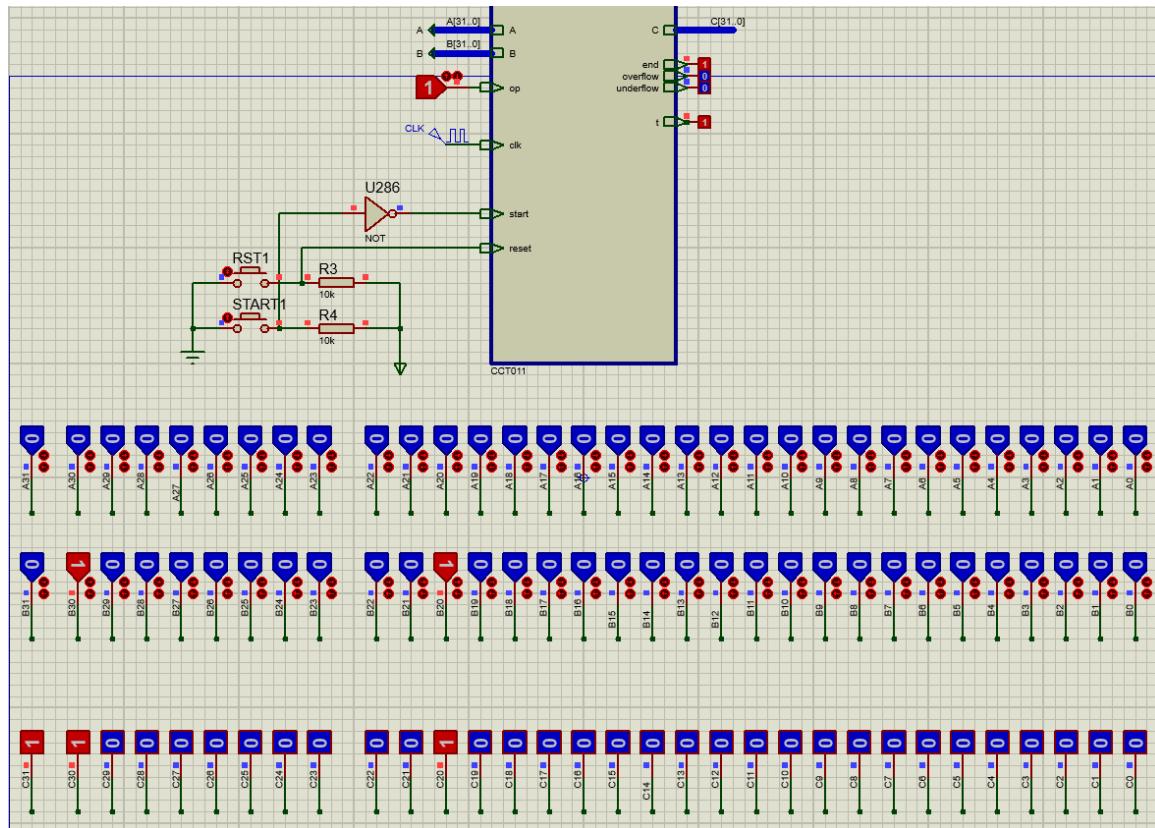
Binary: 0b00000000000000000000000000000000

Input B: 2.25 (0x40100000)

Binary: 0b01000000001000000000000000000000

Subtraction Output C: -2.25 (0xC0100000)

Binary: 0b11000000001000000000000000000000





Test 6

*Addition/subtraction involving infinity.***Input A:** Infinity (0x7F800000)

Binary: 0b0111111100000000000000000000000000000000

Input B: 2.25 (0x40100000)

Binary: 0b01000000001000000000000000000000

Addition Output C: Infinity (0x7F800000)

Binary: 0b01111111000000000000000000000000[0.08in]

