دانشکده مهندسی کامپیوتر

واحد محاسباتی استاندارد اعشاری
بخش‌های تقسیم و ریشه دوم

پروژه نهایی درس طراحی سیستم‌های دیجیتال

امین اسدی، حسین انتظاری، فراز شاهسون، ژیوار صورتی، محسن فیاض، زهرا موسوی

دکتر فرشاد بهاروند

بهمن ۱۳۹۸

پ

تقسیم وظایف تیم		
اعضای گروه	وظیفه	درصد مشارکت
محسن فیاض	پیاده سازی ماژول تقسیم و جذر به روش نیوتن تست با زبان c++ برای ۴ ماژول (گلدن مدل)، تست بنچ ۴ ماژول، تست یکپارچه سازی دوم ماژول wrapper، سنتز	۱۶/۶۶٪
حسین انتظاری	نوشتن wrapper برای ۴ ماژول، تست بنچ wrapper، پیدا کردن کد تقسیم دابل	۱۶/۶۶٪
ژیوار صورتی	نوشتن و قالب بندی گزارش، نوشتن بررسی دقیق استاندارد floating point سینگل و دابل نوشتن مقدمه و چکیده	۱۶/۶۶٪
فراز شاهسون	تحلیل ماژول تقسیمگر با روش شیفت، الگوریتم و عملکرد، نوشتن داکیومنت مربوط به ماژول تقسیمگر، تحلیل حالت‌های خاص special cases تقسیم و جذر، نوشتن حالت‌های خاص تست (corner case)	۱۶/۶۶٪
زهرا موسوی	ساختن ارائه کلاسی پروژه، نوشتن و قالب بندی گزارش، کشیدن شماتیک کامپیوتری ماژول ها، نوشتن بررسی دقیق استاندارد floating point سینگل و دابل	۱۶/۶۶٪
امین اسدی	تحلیل روش نیوتن رافسون برای جذر، تحلیل روش نیوتن رافسون برای تقسیم، پیدا کردن روش بهینه انتخاب مقدار اولیه	۱۶/۶۶٪

فهرست مطالب

فهرست تصاویر

فهرست برنامه‌ها

فصل ۱

مقدمه

۱ـ۱ چکیده

عملیات نقطه شناور [1] به طور گسترده در مجموعه بزرگی از پردازش‌های علمی و سیگنال‌ها مورد استفاده قرار می گیرد. IEEE استانداردی را برای این عملیات با نام IEEE-754 تعیین کرده است. در این پروژه، واحد عملکرد حسابی شناور، مطابق استانداردهای IEEE توسعه داده و آزمایش شد. مشخصات فنی این واحد، ترکیبی از یک منطق کنترل و مسیر داده است. عملیات، لازم است بر روی اعداد و نقطه‌های شناور با دقت یک و دو انجام شود. با توجه به اینکه مفهوم اعداد floating point باید به خوبی درک شود و همینطور به درستی عملیات آن‌ها پیاده سازی شوند، باید واحدهای عملیاتی به خوبی تست شوند. به این منظور نیز از مدل طلایی که به زبان C++ است استفاده می‌شود. گفتنی است که سخت افزار مورد استفاده در این پروژه در زبان verilog پیاده سازی شده است.

[1] floating point

۱ـ۲ تاریخچه و کاربرد

انجمن IEEE در سال ۱۹۸۵ نمایش اعداد باینری ممیز شناور را با IEEE 754 استاندارد سازی کرد. از آن پس، در اکثر معماری های مدرن رایانه از این استاندارد پیروی می‌شود. در گذشته، واحد محاسباتی ممیز شناور به عنوان یک پردازنده کمکی در کنار پردازنده اصلی قرار می‌گرفت اما امروزه با پیشرفت تکنولوژی و اهمیت بالای محاسبات اعشاری در کارهای گرافیکی یا سه بعدی، این واحد حتی به صورت کاملا جدا، در کارت های گرافیکی قرار می‌گیرد.

از اوایل دهه ۱۹۹۰، بسیاری از ریز پردازنده‌ها برای desktop ها و سرورها بیش از یک FPU دارند.

۱ـ۳ نحوه کلی عملکرد

مدار طراحی شده می‌تواند بین ۴ حالت تقسیم و جذر ۳۲ بیتی یا ۶۴ بیتی اعداد اعشاری تغییر کاربرد دهد. پس از انتخاب عمل مورد نظر و گذاشتن ورودی های لازم، پس از مدتی، مدار آماده شدن نتیجه‌ی عملیات را اعلام می‌کند و این نتیجه قابل استفاده خواهد بود.

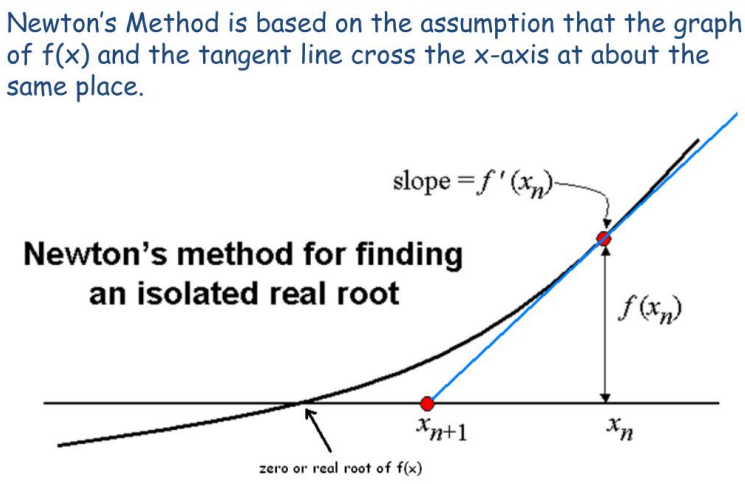
برای ساخت این مدار ابتدا تحقیقات در مورد خود استاندارد اعداد ممیز شناور و سپس درباره انتخاب روش مناسب تقسیم و جذر با توجه به ماهیت سخت‌افزاری پروژه آغاز شد و پس از بررسی‌ها، روش نیوتن انتخاب، طراحی، پیاده سازی و تست شد. سپس ماژولی برای در بر گرفتن ۴ ماژول مورد نظر طراحی، پیاده سازی و تست شد.

در انتها با استفاده از ابزار vivado سنتز و پیاده سازی مدار انجام گرفت که با موفقیت انجام شد و نتایج حاصله در هر مرحله مستندسازی شد.

۱ـ۴ پایه ریاضی

۱ـ۴ـ۱ روش نیوتن

روش نیوتن که به روش "نیوتن_رافسون" نیز معروف است یکی از روش‌های تقریب ریشه توابع حقیقی در محاسبات عددی است. در این روش از یک نقطه اولیه به عنوان "حدس اولیه" استفاده می‌شود و در طی



شکل ۱ـ۱: روش نیوتن در تقریب جذر

مراحل متوالی تقریب‌های دقیق‌تری برای ریشه حاصل می‌شوند؛ به این صورت که اگر تقریب بدست آمده در مرحله n اُم برابر x_n باشد، تقریب در مرحله $n+1$ اُم از رابطه‌ی زیر حاصل می‌گردد:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

که x_{n+1} تقریب بهتری نسبت به x_n برای ریشه است.

در واقع این روش از شیب نمودار تابع که به سمت محل تلاقی آن با محور افقی اشاره دارد کمک می‌گیرد.

صحت رابطه‌ی بالا از شکل ۱ـ۱ پیداست:

$$y = f'(x_n)(x - x_n) + f(x_n),$$
$$O = f'(x_n)(x_{n+1} - x_n) + f(x_n).$$
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

۱ـ۴ـ۲ کاربرد روش نیوتن در تقریب جذر

فرض کنید c یک عدد مثبت دلخواه باشد. می‌خواهیم حاصل جذر این عدد را بدست آوریم؛ یعنی عدد x را به گونه‌ای بدست آوریم که:

$$x^2 = c \Rightarrow x^2 - c = 0$$

در نتیجه کافیست ریشه‌ی تابع $f(x) = x^۲ - c$ را بدست‌اوریم:

$$f'(x) = 2x$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n{}^۲ - S}{2x_n} = \frac{۱}{۲}(2x_n - (x_n - \frac{S}{x_n})) = \frac{۱}{۲}(x_n + \frac{S}{x_n})$$

روش انتخاب حدس اولیه: برای انتخاب حدس اولیه در این قسمت با توجه به شکل نمودار تابع $f(x) = x^۲ - c$ دشواری وجود ندارد و با انتخاب هرمقدارمثبت برای حدس اولیه، ریشه‌ی مثبت این این تابع به درستی بدست خواهد آمد؛ از این رو از نصف بخش توانی عدد به عنوان حدس اولیه استفاده شده است که در بخش ۲ـ۳ـ۴ به علت این موضوع پرداخته شده است.

۱ـ۴ـ۳ کاربرد روش نیوتن در تقریب تقسیم

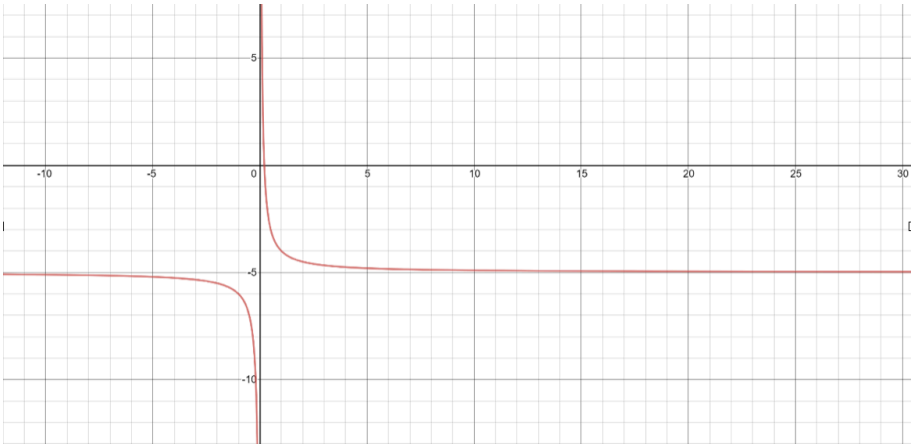
برای محاسبه حاصل تقسیم $Q = \frac{N}{D}$ ابتدا حاصل $X = \frac{۱}{D}$ را با تقریبات متوالی بدست آورده و سپس Q را با استفاده از حاصل ضرب X و N بدست می آورد؛ یعنی:

$$Q = \frac{N}{D} = N \times (۱ \times D) = N \times X$$

برای محاسبه تقریب $\frac{۱}{D}$ با استفاده از روش نیوتن باید یک تابع $f(x)$ پیدا کنیم که یک ریشه در نقطه $x = \frac{۱}{D}$ داشته باشد. به وضوح یک تابع با چنین شرایطی $g(x) = Dx - ۱$ است، ولی با استفاده از این تابع تقریب نیوتن بدون داشتن مقدار $\frac{۱}{D}$ قابل محاسبه نیست. در واقع روش نیوتن با استفاده از این تابع سعی می‌کند مقدار دقیق $\frac{۱}{D}$ را در یک مرحله بدست آورد. تابع مناسبی که در اینجا قابل استفاده است تابع $f(x) = (\frac{۱}{x}) - D$ است. با استفاده از فرمول تقریب نیوتن داریم:

$$X_i + ۱ = X_i - \frac{f(X_i)}{f'(x)} = X_i - \frac{\frac{۱}{X_i} - D}{-\frac{۱}{X_i^۲}} = X_i + X_i(۱ - DX_i) = X_i(۲ - DX_i)$$

روش انتخاب حدس اولیه: برای انتخاب حدس اولیه در این قسمت با توجه به شکل نمودار تابع $f(x) = \frac{۱}{x} - c$ در صورتی که یک حدس اولیه بین ۰ و ریشه مورد نظر انتخاب شود و اینکه شیب تابع در این ناحیه همواره منفی است، پس از چند مرحله به دقت مناسبی برای ریشه دست خواهیم یافت. برای یافتن یک حدس اولیه با این ویژگی قسمت توانی عدد $\frac{۱}{D}$ را که عددی منفی است، قرینه می‌کنیم و سپس یک واحد از آن کم می‌کنیم که در بخش ۲ـ۳ـ۱ به علت این موضوع پرداخته شده است.



شکل ۱ـ۲: Newton Method for Divide

۱ـ۵ کاربردها

برتری نمایش ممیز شناور به ممیز ثابت و عدد صحیح، در توانایی پشتیبانی آن از دامنهٔ گسترده‌تری از مقادیر است. نمایش ممیز ثابت که ۷ رقم ده‌دهی و دو رقم اعشاری داشته باشد می‌تواند برای نمایش اعدادی چون ۶۷/۱۲۳۴۵، ۴۵/۱۲۳، ۲۳/۱ و... به کار رود، در حالی که در نمایش ممیز شناور (همانند قالب ممیز شناور دهدهی۳۲ بیتی در IEEE 754) با هفت رقم دهدهی، می‌توان افزون بر موارد قبلی، ۲۳۴۵۶۷/۱، ۷/۱۲۳۴۵۶، ۰۰۰۰۱۲۳۴۵۶۷/۰، ۱۲۳۴۵۶۷۰۰۰۰۰۰۰۰۰ و... را نیز نمایش داد. نمایش ممیز شناور به کمی حافظهٔ بیشتر نیاز دارد (برای کدبندی محل ممیز)، بنابراین اگر قرار باشد اعداد در فضایی برابر با ممیز ثابت ذخیره شوند، ممیزهای شناور دامنهٔ بیشتری را به بهای دقت کمتر پشتیبانی خواهند کرد.

سرعت عملیات ممیز شناور، که معمولاً با عنوان اندازه‌گیری کارایی فلاپس شناخته می‌شود، از ویژگی‌های مهم ماشین‌ها است، مخصوصاً در نرم‌افزارهایی که عملیات ریاضی را در مقیاس گسترده انجام می‌دهند. توانایی انجام میلیارد محاسبه در هر ثانیه بر روی اعداد واقعی، کامپیوترهای مدرن را بسیار مفید می‌کند. گرافیک‌های سه بعدی، دنیاهای مجازی برای بازی، شبیه سازی‌های علمی و ابزار طراحی به کمک رایانه (CAD) فقط زیرمجموعه‌ی برنامه‌هایی هستند که با چنین قابلیت‌هایی امکان پذیر هستند. رایانه‌ها از اعداد ممیز شناور به عنوان تقریب محدود برای نمایش علمی عادی اعداد واقعی استفاده می‌کنند. در گذشته از چندین قالب با دقت و کنوانسیون‌های مختلف استفاده شده است، اما خوشبختانه قالب و عملکردهای تعریف شده بر روی آنها توسط IEEE 754 در سال ۱۹۸۵ استاندارد شده و به طور جهانی توسط صنعت پذیرفته شده است.

متداول ترین فرمت‌ها عبارتند از ۳۲ بیتی تک دقتی (float) و ۶۴ بیتی با دقت دو برابر .(double)

۱ ـ ۶ استانداردها

استاندارد IEEE برای محاسبات شناور (IEEE 754) یک استاندارد فنی برای محاسبات نقطه شناور است که در سال ۱۹۸۵ توسط انستیتو مهندسان برق و الکترونیک (IEEE) تاسیس شده است. این استاندارد بسیاری از مشکلات موجود در اجرای متنوع نقاط شناور را نشان می‌دهد که استفاده از آنها را غیرمطمئن و دشوار کرده و قابلیت حمل آنها را کاهش داده است. IEEE Standard 754 floating point رایج ترین نمایش امروز برای شماره‌های واقعی در رایانه‌ها، از جمله رایانه‌های شخصی مبتنی بر اینتل، مکینتاش و اکثر سیستم عامل‌های یونیکس است.

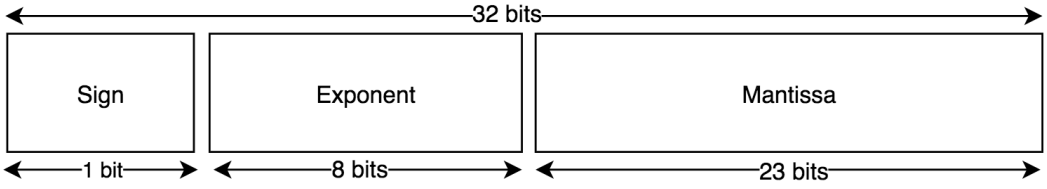
روش‌های مختلفی برای نشان دادن تعداد نقطه شناور وجود دارد اما IEEE 754 در بیشتر موارد کارآمدترین است. IEEE 754 دارای ۳ مؤلفه اساسی است:

۱. The Sign of Mantissa

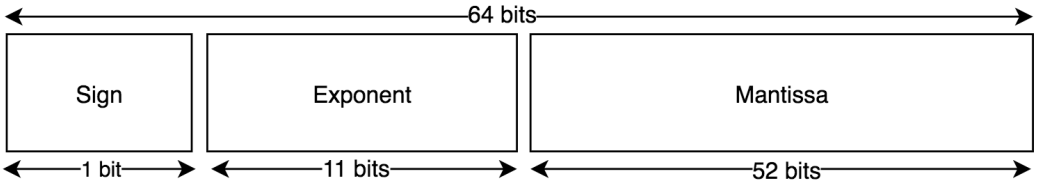
۲. The Biased exponent

۳. The Normalised Mantisa

IEEE 754 بر اساس اندازه‌ی سه مؤلفه‌ی فوق به دو بخش تقسیم می شود:



شکل ۱ ـ ۳: Single Precision IEEE 754 Floating Point Standard



شکل ۱ ـ ۴: Double Precision IEEE 754 Floating Point Standard

مقادیر ویژه [۲] : IEEE برخی از مقادیر را که می تواند ابهام داشته باشد رزرو کرده است.

- صفر: $0-$ و $0+$ مقادیر مشخصی دارند، گرچه هر دو برابر هستند.
- مخدوش شده [۳]
- بی نهایت
- غیر عدد [۴] مقدار NAN برای نشان دادن یک مقدار که یک خطاست استفاده می شود.

Special Values		
Exponent	Mantisa	Value
0	0	exact 0
255	0	Infinity
0	not 0	denormalised
255	not 0	NAN

اعداد دِنرمال: [۵] اعداد دنرمال به حالت خاصی از نمایش اعداد، در قرارداد استاندارد IEEE به دو صورت ۳۲ بیتی و ۶۴ بیتی اطلاق میشود که به منظور نمایش اعداد بسیار کوچک تعبیه شده است و محاسبه‌ی آن با حالت عادی برای بدست آوردن یک عدد از روی نمایش IEEE آن عدد متفاوت است. در حالت عادی،

[2] Special Values
[3] Denormalised
[4] Not A Number (NAN)
[5] Denormal Numbers

برای بدست آوردن یک عدد از روی فرم IEEE ، طبق فرمول زیر عمل میکنیم:

- اعداد ۳۲ بیتی :

$$(-1)^{sign} \times 1.[mantissa] \times 2^{exponent-127}$$

- اعداد ۶۴ بیتی :

$$(-1)^{sign} \times 1.[mantissa] \times 2^{exponent-1023}$$

این در حالی است که اگر طبق قرارداد، تمام بیتهای exponent در نمایش IEEE برابر صفر قرار داده شوند و mantissa مقداری غیر از صفر به خود گیرد، اعداد در دو حالت ۳۲ بیتی و ۶۴ بیتی به صورت زیر محاسبه می‌شوند:

- اعداد ۳۲ بیتی :

$$(-1)^{sign} \times 0.[mantissa] \times 2^{-126}$$

- اعدا ۶۴ بیتی :

$$(-1)^{sign} \times 0.[mantissa] \times 2^{-1022}$$

که در این حالت به آنها اعداد دِنرمال گفته می‌شود. در جدول زیر، دامنه‌ی اعداد قابل نمایش در هر حالت نرمال و دنرمال، ۳۲ بیتی و ۶۴ بیتی آمده است.

Floating Point Range

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23})\times 2^{127}$	$\pm \approx 10^{-44.85}$ to $\approx 10^{38.53}$
Double Precision	$\pm 2^{-1074}$ to $(1-2^{-52})\times 2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$	$\pm \approx 10^{-323.3}$ to $\approx 10^{308.3}$

شکل ۱ ـ ۵: Floating Point Range

مراجع

[1] *Babylonian method*. URL: https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_method.

[2] *Newton–Raphson division*. URL: https://en.wikipedia.org/wiki/Division_algorithm#Newton%5C%E2%5C%80%5C%93Raphson_division.

[3] *Newton–Raphson method*. URL: https://en.wikipedia.org/wiki/Newton%5C%27s_method.

[4] *Newton–Raphson method*. URL: https://en.wikipedia.org/wiki/Newton%5C%27s_method#Square_root_of_a_number.

[5] *Square Root and Divide Golden Model*. URL: http://www.cs.utsa.edu/~wagner/CS3343/newton/divroot.html.

[6] *Square root binary estimates*. URL: https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Binary_estimates.

[7] *Xilinx Vivado Design Suite*. URL: http://users.wpi.edu/~rjduck/MMCM%5C%20Vivado%5C%20example%5C%20Verilog.pdf.

فصل ۲

توصیف معماری سیستم

سیستم به طور کلی از چهار ماژول محاسبه گر و یک واحد controller تشکیل شده است، که این واحد به عنوان یک لایه wrapper روی بقیه چهار ماژول محاسبه گر عمل می کند.

این چهار واحد محاسبه‌گر که هر کدام وظیفه انجام محاسبه خاصی را به عهده دارند، به شرح زیر است:

- single precision DIVIDER
- double precision DIVIDER
- single precision SQRT
- double precision SQRT

controller وظیفه وصل کردن ورودی های ماژول اصلی به ورودی های این چهار واحد محاسبه‌گر و وصل کردن خروجی های این چهار واحد به خروجی های ماژول اصلی دارد، تا عملیات hand shake مانند همان hand shake موجود در واحد های محاسبه گر باشد، همچنین controller بر اساس مقدار ورودی process ورودی های ماژولی که باید محاسبات را انجام دهد، تغییر می دهد.

۲ـ۱ اینترفیس‌های سیستم

۲ـ۱ـ۱ مشخصات کلی

در این بخش به توصیف اختصاری interface یا همان wrapper که در واحد کلی استفاده شده، می‌پردازیم. wrapper در اینجا در حقیقت همان interface برای کل سیستم است. wrapper به طور کلی یک subroutine در برنامه‌ها و همینطور واحدهای محاسباتی است که وظیفه اصلی آن فراخوانی یک تابع دیگر است بدون هیچ هزینه محاسباتی دیگر.

این واحد در حقیقت ورودی‌ها و خروجی‌های واحد محاسبه گر را control می‌کند. به این شکل که ورودی و خروجی‌های سیستم در ابتدا از این واحد یعنی wrapper رد می‌شوند تا به واحد اصلی sqrt یا division برسند. به جز control ورودی و خروجی‌های سیستم wrapper با استفاده از Signal هایی که از واحد کنترلی می‌گیرد، واحد اصلی یا همان واحد محاسبه گر را Control می‌کند. کاری که این واحد انجام می‌دهد به بیان بهتر، شامل منتظر ماندن برای گرفتن ورودی‌ها، اعلام دیده شدن ورودی‌ها، گرفتن Mode سیستم (اینکه System چه کاری را انجام بدهد را Control می‌کند) ، دادن ورودی‌ها و همینطور mode سیستم به واحد اصلی، گرفتن خروجی واحد اصلی، اعلام آماده بودن خروجی، دادن خروجی به بیرون واحد اصلی و همینطور منتظر ماندن برای اعلام دیده شدن خروجی توسط Controller ، می‌شود.

۲ـ۱ـ۲ ورودی‌ها

این واحد ورودی‌های سیستم را تحت عنوان input_as و Input_bs می‌گیرد که این ورودی‌ها ۳۲ بیتی هستند. همینطور برای گرفتن ورودی‌های ۶۴ بیتی از سیم‌های input_ad و Input_bd استفاده می‌شود. همینطور برای اینکه مشخص شود این ورودی‌ها پایدار شده اند، این واحد Signal هایی را تحت عنوان Input_a_stb و Input_b_stb می‌گیرد. زمانی که ورودی‌ها روی Input_a و Input_b قرار دارند، Wrapper با دیدن Input_a_stb و Input_b_stb ، آن ورودی‌ها را پایدار محسوب می‌کند. کارایی قسمت گفته شده این است که میتوانیم کنترل کنیم که تا زمانی که ورودی‌ها پایدار نشده اند، کار محاسبه را شروع نکنیم. کنترل

کردن Mode سیستم نیز از طریق سیمی تحت عنوان Process که به واحد کنترلی وارد می‌شود، انجام می‌شود. این ورودی از طریق یک Signal کنترلی که از طریق Controller به Wrapper وارد می‌شود mode سیستم را مشخص می‌کند.

این Signal با Signal ای که مشخص کننده‌ی پایدار بودن ورودی است and می‌شود و خروجی آن به یکی از Pin های واحدهای محاسباتی اصلی که وظیفه Enable کردن واحد را دارند وارد می‌شود. به این صورت تنها یکی از ۴ واحد محاسباتی در یک زمان فعال می‌باشند. دو ورودی دیگر نیز به Wrapper وارد می‌شوند که وظیفه اعلام اینکه خروجی واحدهای محاسباتی پایدار شده، دیده شده است را دارند.

۲ـ۱ـ۳ خروجی‌ها

حال به توصیف خروجی‌های این Wrapper می‌پردازیم که خروجی واحدهای محاسباتی را Control می‌کند. همان طور که گفته‌شد Wrapper باید خروجی‌های واحد اصلی که درون آن است را بگیرد و به بیرون از خود انتقال دهد؛ البته این انتقال با توجه به Signal های کنترلی صادر شده از واحد کنترلی انجام می‌گیرد.

یکی از خروجی‌های Wrapper همان خروجی واحد محاسباتی داخل Wrapper است که با Output_z نشان داده می‌شود. خروجی دیگر این واحدهای محاسباتی که پایدار شدن خروجی این واحدها را نشان می‌دهد نیز تحت عنوان Output_stb وجود دارد. در زمانی که خروجی واحد محاسباتی پایدار شود ، خروجی Output_z که Wrapper آن را به بیرون انتقال می‌دهد، به عنوان خروجی سیستم شناخته می‌شود. دلیل وجود این قسمت از Wrapper این است که در زمان آماده بودن خروجی یا حتی مقدار داشتن خروجی واحدهای محاسباتی بتوانیم صحیح بودن این خروجی‌ها را تشخیص دهیم و از آن‌ها استفاده کنیم. نکته دیگری که در Wrapper مشخص است این است که با توجه به Signal های کنترلی صادرشده از طرف واحد کنترلی، خروجی مورد نظر با استفاده از Mux هایی که در قسمت آخر Wrapper قرار دارند، به بیرون انتقال پیدا می‌کند.

نکته مهم که در آخر باید آن را ذکر کنیم این است که Acknowledge signal گفته شده در ابتدا یک مقدار Initial دارد و پس از مدتی که پایدار شدن ورودی دیده‌شد، تغییر خواهد کرد. مکانیزم گفته شده با استفاده

از یک And پیاده سازی می‌شود که عبارتست از And شده همه Signal های Acknowledge که از واحدهای محاسباتی گرفته می‌شود. همینطور در نظر داشته باشید که در ابتدا دیده شدن و گرفتن اولین ورودی یعنی input_a بررسی می‌شود و پس از آن ورودی دیگر بررسی می‌شود.

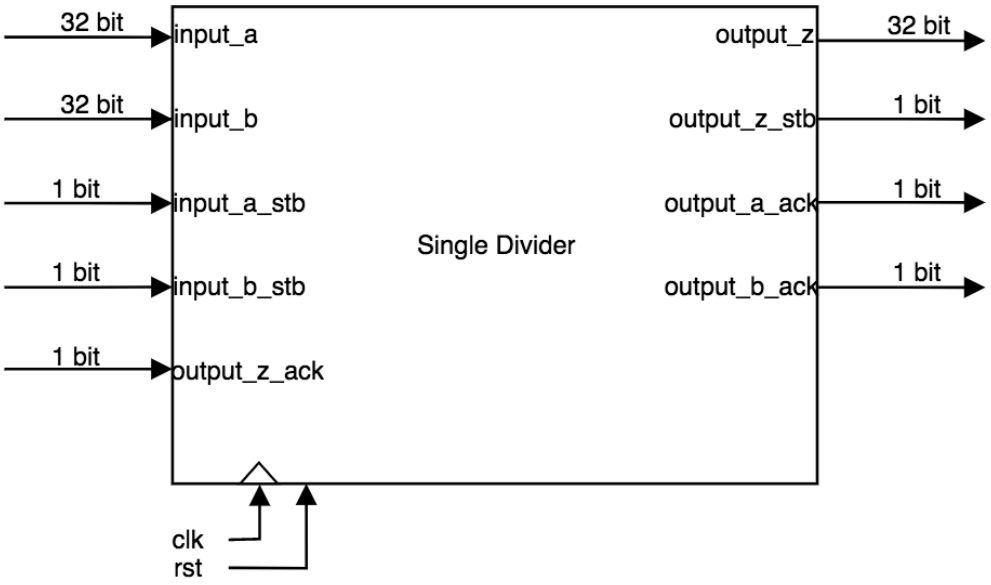
به طور کلی تر مکانیزم دیده شدن ورودی اول با استفاده از And شدن همه Signal های acknowledge واحدها و ورودی دوم با استفاده از Or شدن همه Signal های acknowledge واحدها بدست می آید. به طور دقیق‌تر برای اعلام دیده شدن ورودی اول، ملاک، صفر شدن acknowledge signal است که در ابتدا یک بوده و برای ورودی دوم، ملاک، یک شدن acknowledge signal است که در ابتدا صفر بوده‌است.

۲ـ۱ـ۴ کلاک سیستم

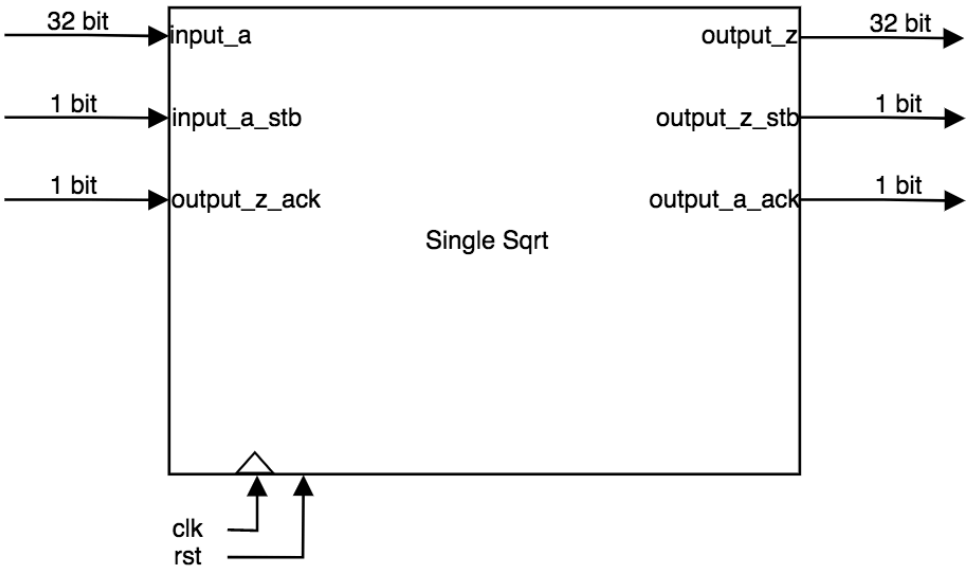
این سیستم، از یک کلاک واحد برای تمام ماژول‌ها استفاده می کند که به علت وجود نداشتن CDC [۱] باعث عدم لزوم استفاده از همگام ساز [۲] در داخل سیستم می شود.
در بخش سنتز از یک MMCM نیز استفاده شده است که در همان بخش توضیح داده شده است.

۲ـ۲ دیاگرام بلوکی سخت‌افزار

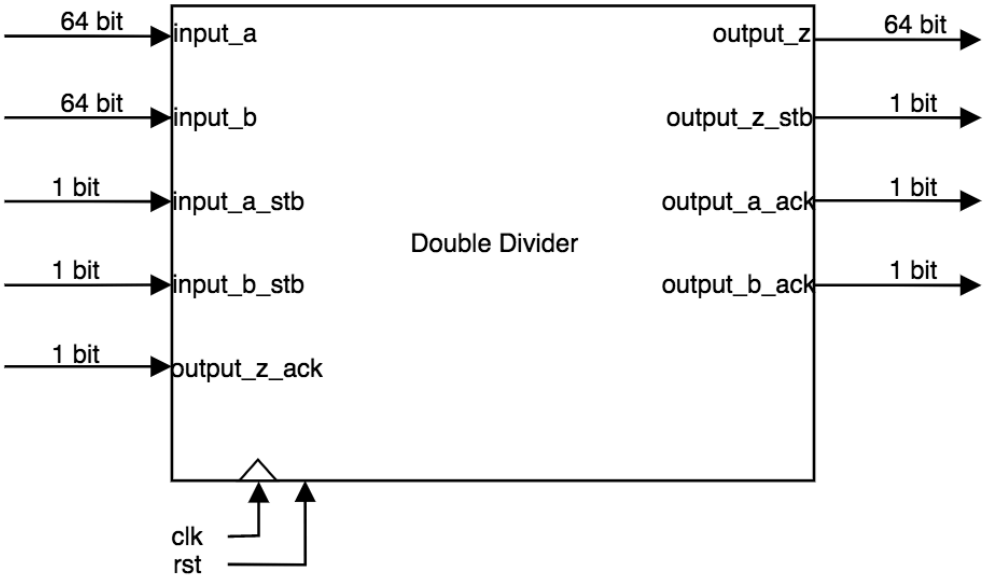
[1] Clock Domain Crossing
[2] Synchronizer



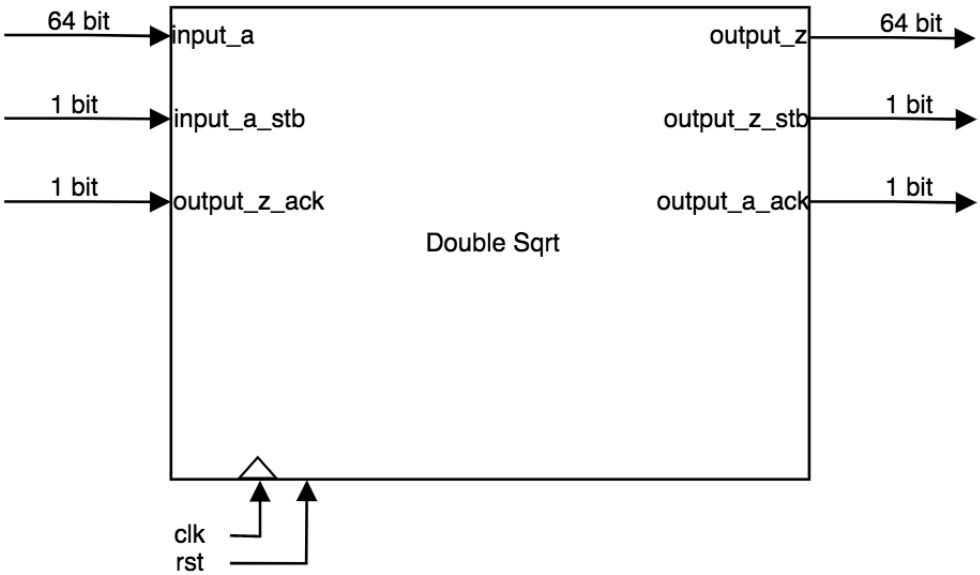
شکل ۲ ـ ۱ : Single Divider Module



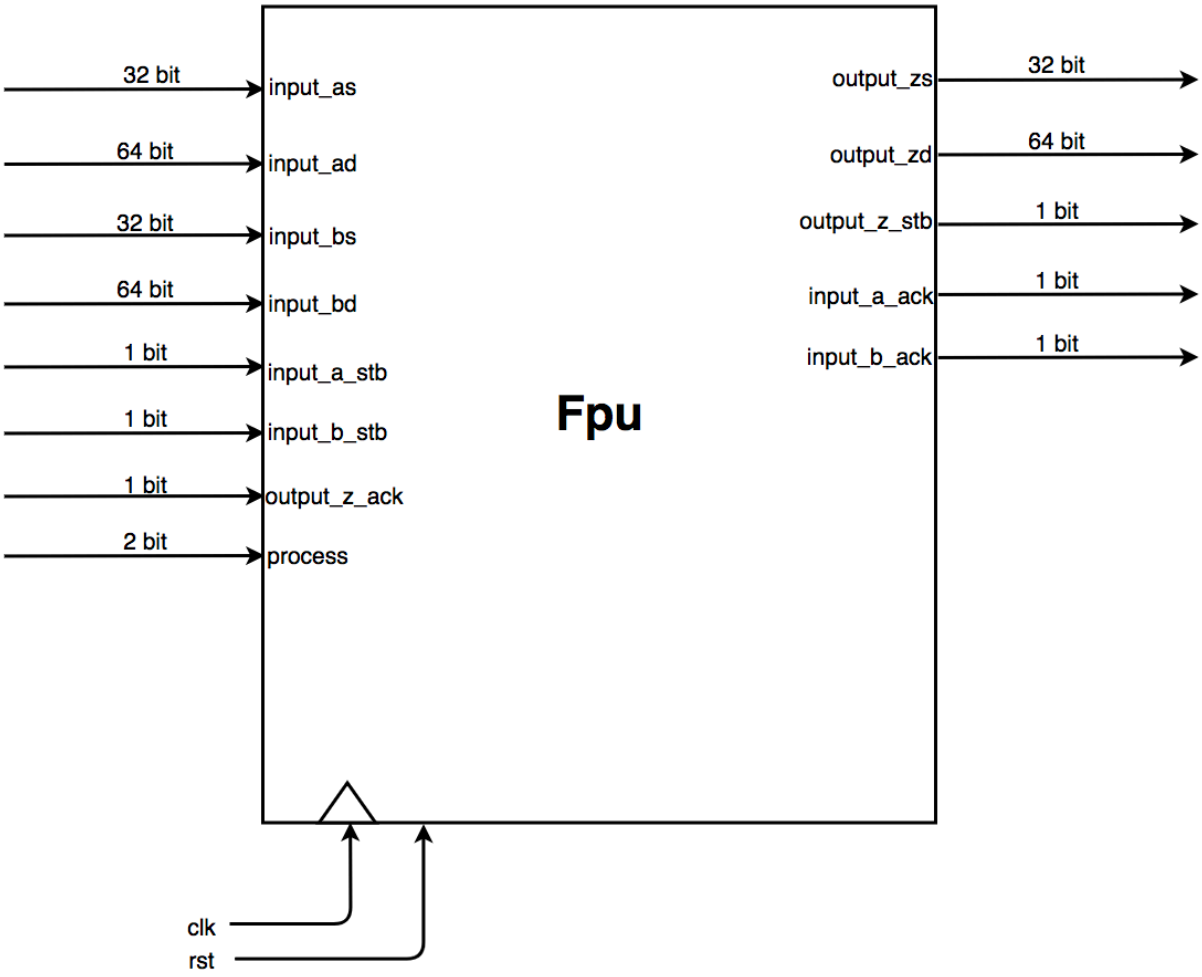
شکل ۲ ـ ۲ : Single Sqrt Module



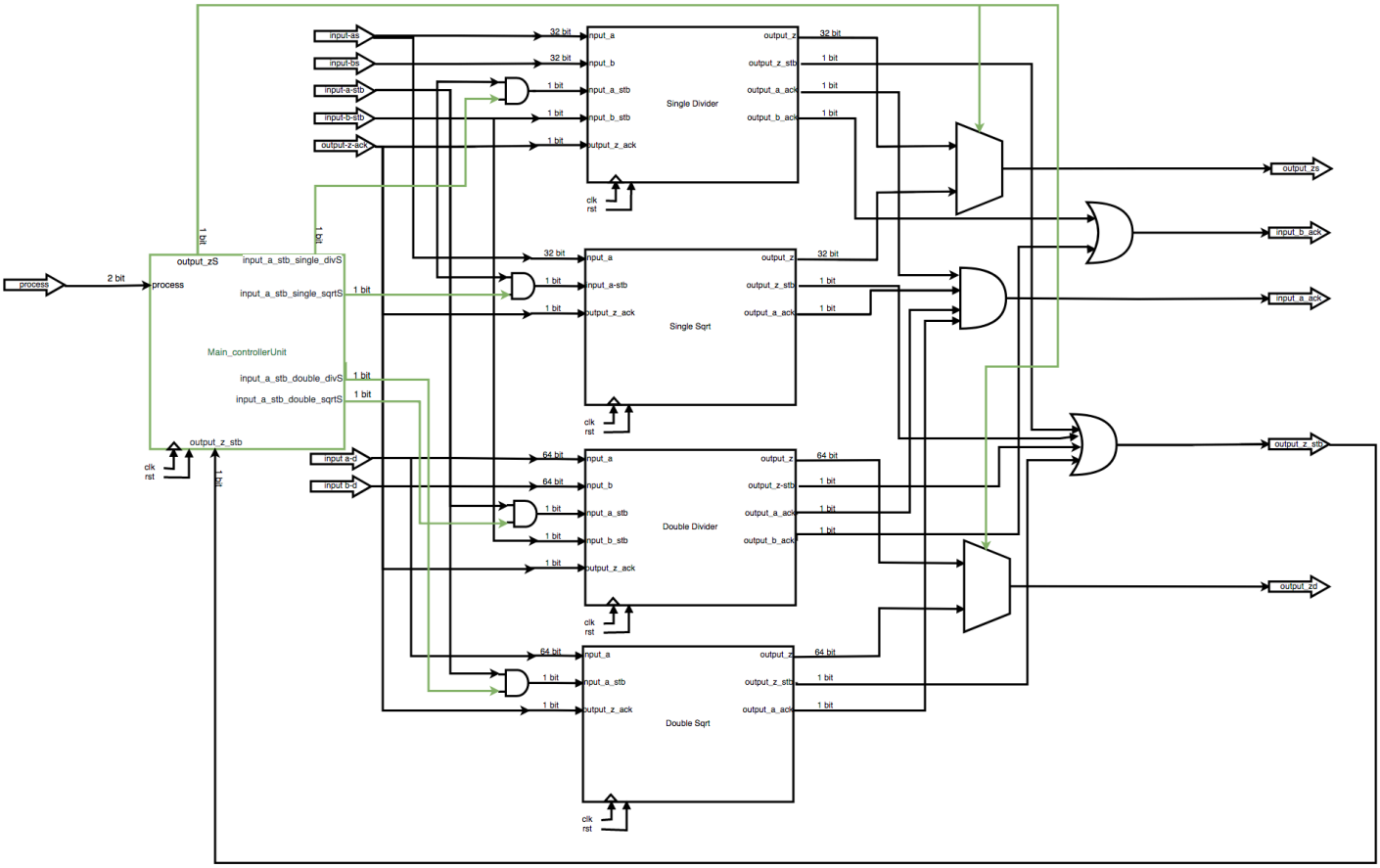
شکل ۲ـ۳: Double Divider Module



شکل ۲ـ۴: Double Sqrt Module



شکل ۲ـ۵: Main module



شکل ۲ـ۶: Inside of Main Module

۲ـ۳ توصیف هر **module** به صورت جداگانه

۲ـ۳ـ۱ **Single Divider Newton** (آـ۳)

وظیفه این ماژول تقسیم دو عدد Single Precision Floating Point است.

این ماژول از روش نیوتن رافسون که قبلا توضیح داده شده استفاده می‌کند.

ابتدا طبق معمول با استفاده از handshake مناسب، دو عدد a و b ورودی گرفته می‌شوند و محاسبه تقسیم $\frac{a}{b}$ آغاز می‌گردد.

سپس حالت‌های خاص تقسیم بررسی می‌شوند، و در صورت وقوع هر کدام، پاسخ سریعاً در خروجی قرار می‌گیرد.

Division Special Cases	
Operation	Result
$NAN \div n$	NAN
$n \div NAN$	NAN
$NAN \div NAN$	NAN
$\pm inf \div \pm inf$	NAN
$0 \div 0$	NAN
$\pm inf \div n$	$\pm inf$
$n \div 0$	$\pm inf$
$n \div \pm inf$	0
$0 \div n$	0

در صورتی که هیچکدام از حالت‌های بالا رخ نداده بود باید عملیات تقسیم شروع شود. چون روش تقسیم توضیح داده شده است در اینجا روی انتخاب حدس اولیه مناسب برای $\frac{۱}{b}$ تمرکز می‌کنیم که در روش نیوتن محاسبه می‌شود. برای این منظور بهترین کار استفاده از قابلیت‌هایی است که ذخیره سازی به روش floating point در اختیارمان می‌گذارد.

ساختار عدد ورودی b به شکل زیر است.

$$b = (-۱)^{Sign} \times (۱.Mantissa) \times ۲^{e-۱۲۷}$$

ابتدا برای همگرایی و جلوگیری از مشکلات محاسباتی، علامت b را ذخیره می‌کنیم تا در آخر کار علامت درست را خروجی دهیم و در روش نیوتن از قدر مطلق عدد استفاده می‌کنیم. طبق شکلِ ساختاری عدد ورودی، می توان معکوس آن را به شکل زیر نوشت.

$$\frac{۱}{|b|} = \frac{۱}{(۱.Mantissa)} \times ۲^{-(e-۱۲۷)}$$

با توجه به این ساختار عددی، برای انتخاب حدس اولیه به راحتی می توان بخش توانی عدد را قرینه کرد. البته برای اینکه مطمئن باشیم روش نیوتن برای محاسبه تقسیم همگرا می‌شود، باید حدسمان از اصل عدد کمتر باشد، بنابراین علاوه بر قرینه کردن توان ۲، یکی هم از آن کم می‌کنیم تا اطمینان حاصل شود که حدس کمتر از مقدار واقعی $\frac{۱}{|b|}$ است و جواب نیوتن همگرا می‌شود. عدد حدس زده شده به شکل زیر است.

$$\frac{۱}{|b|} \simeq (۱.Mantissa) \times ۲^{-(e-۱۲۷)-۱}$$

محاسبه $x_n \times (۲ - b \times x_n)$ خودش به سه بخش ضرب کردن، تفریق و سپس ضرب کردن، تقسیم می‌شود. شرط خاتمه محاسبات این است که خروجی یک مرحله‌ی محاسبه با خروجی دفعه قبل برابر باشد و یا تعداد باری که محاسبه در حال انجام است به تعداد مشخصی برسد که در فایل defines.v (آـ۱) مشخص شده است.

برای انجام محاسبات ضرب و جمع مورد نیاز در این ماژول از ماژول‌هایی استفاده شده است که به صورت تمام و کمال، هر حالتی از اعداد ممیز شناور از جمله حالات غیرنرمال را پشتیبانی می‌کنند. بنابراین این ماژول نیز می تواند تقسیم را روی هر نوع عدد ممیز شناور از جمله غیرنرمال انجام دهد که تست این موارد در بخش شبیه سازی آمده است.

البته در موارد خاص، برای بعضی اعداد غیر نرمال که بسیار کوچک هستند، نمی توان $\frac{۱}{b}$ را محاسبه کرد، زیرا توزیع اعداد در ساختار ممیز شناور به کمک تعریف اعداد غیر نرمال، روی اعداد کوچک تمرکز بیشتری دارد و نمی تواند معادل آن مقدار کوچک را در اعداد بزرگ داشته باشد. به عنوان مثال در اعداد ممیز شناور ۳۲ بیتی کوچکترین عدد قابل نمایش حدود $۱/۴ \times ۱۰^{-۴۵}$ است اما بزرگترین عدد قابل نمایش در این استاندارد حدود $۳/۴ \times ۱۰^{+۳۸}$ است. به همین دلیل روش نیوتن برای تقسیم که سعی می‌کند ابتدا $\frac{۱}{b}$ را تخمین بزند، برای بعضی از اعداد غیر نرمالی که معکوسشان قابل نمایش نباشد و در مخرج کسر باشند، قابلیت محاسبه ندارد.

۲ـ۳ـ۲ Double Divider Newton (آـ۴)

پیاده سازی این بخش مانند بخش قبل است و تنها تفاوتش در تعداد بیت‌ها است.
ورودی b به شکل زیر است.

$$b = (-۱)^{Sign} \times (۱.Mantissa) \times ۲^{e-۱۰۲۳}$$

و عدد حدس زده شده به شکل زیر است.

$$\frac{۱}{|b|} \simeq (۱.Mantissa) \times ۲^{-(e-۱۰۲۳)-۱}$$

۲ـ۳ـ۳ Divider (Shift and Subtract) (آـ۷)

در این قسمت، به بحث درباره تقسیم‌گر اعشاری می‌پردازیم. ابتدا پورت‌های ورودی و خروجی این تقسیم‌گر را توضیح می‌دهیم. سپس خواص و ویژگی‌های این تقسیم‌گر را بیان می‌کنیم. بعد، الگوریتم مورد استفاده برای عملیات تقسیم را شرح می‌دهیم. بعد از آن، کد شبیه سازی این تقسیم‌گر را مورد بررسی قرار داده و درمورد اجزای آن صحبت می‌کنیم.
این تقسیم‌گر دو ورودی ۳۲ بیتی Input_a و Input_b را دریافت میکند تا نتیجه تقسیم Input_a بر Input_b را محاسبه کند و در خروجی ۳۲ بیتی Output_z آن را تحویل دهد. تقسیم‌گر از ۲ سیگنال ورودی ۱ بیتی Input_a_stb و Input_b_stb استفاده می‌کند تا خبردار شود که چه موقع، به ترتیب ورودی‌های روی پورتهای

متناظر Input_a و Input_b صحیح و آماده‌ی خواندن است. همینطور در هنگام خواندن، تقسیم‌گر از دو سیگنال خروجی ۱ بیتی Input_a_ack و Input_b_ack استفاده می‌کند تا خوانده شدن این دو ورودی را به مرجع آنها اطلاع دهد. بعد از محاسبات مربوط به عمل تقسیم، تقسیم‌گر با استفاده از سیگنال خروجی ۱ بیتی Input_z_stb آماده شدن جواب را اطلاع می‌دهد و در صورت خوانده شدن جواب ذخیره شده در Output_z توسط ماژولی دیگر، یک سیگنال در پورت ورودی ۱ بیتی Output_z_ack دریافت می‌کند که بعد از آن می‌تواند مقدار روی خروجی را تغییر دهد زیرا مقدار قبلی توسط ماژول هدف دریافت شده است. همچنین این تقسیم‌گر از ورودی ۱ بیتی clk برای کلاک محاسباتش و از ورودی ۱ بیتی rst برای ریست همزمان با کلاک استفاده میکند.

تقسیمگر مورد بحث، تقسیم‌گر اعشاری اعداد نمایش داده شده طبق استاندارد IEEE و single-precision است که برای ذخیره سازی این اعداد، ۳۲ بیت مورد نیاز است. تعداد کلاک مورد نیاز در این تقسیم‌گر برای هر بار اجرای عمل تقسیم بر روی دو ورودی a و b از زمانی که این دو ورودی روی پورتهای Input_a و Input_b آماده باشند تا زمانی که خروجی روی Output_z آماده استفاده باشد، حداقل ۷ سیکل کلاک و حداکثر ۱۸۱ سیکل کلاک می‌باشد. علت این اختلاف زیاد بین بیشترین و کمترین زمان عملکرد تقسیمگر این است که این تقیسمگر تمام حالات استثنایی ورودی‌ها را در نمایش استاندارد IEEE از قبیل صفر، بینهایت و اعداد دِنرمال را تشخیص داده و به درستی در محاسبات خود لحظ می‌کند. همچنین این تقسیم‌گر، نتیجه خروجی را به صورت کامل به فرم استاندارد نمایش IEEE در می‌آورد و در صورت لزوم قادر به نمایش اعداد استثنایی مانند بینهایت و اعداد دِنرمال نیز هست. تقسیم‌گر برای محاسبه نتیجه تقسیم دو عدد از یک ماشین حالت با ۱۵ حالت مجزا از هم استفاده می‌کند که در ذیل، به توضیح از هر کدام از این حالات می‌پردازیم:

- تقسیم‌گر کارش را از حالت get_a شروع می‌کند. در این حالت تقسیم‌گر منتظر می‌شود تا ورودی Input_a آن مقدار صحیحی دریافت کند تا آن را برای ادامه‌ی پردازش در رجیستر a بریزد. بعد از این مرحله، پردازش به حالت شماره ۲ می‌رسد.

- تقسیم‌گر در حالت get_b مشابه کار بالا را برای ورودی Input_b انجام می‌دهد و این ورودی را در رجیستر b ذخیره می‌کند. سپس تقسیمگر تغییر حالت می‌دهد به حالت شماره ۳.

- در مرحله unpack تقسیم‌گر، اجزای تشکیل دهنده‌ی هر کدام از اعداد a و b را که شامل علامت عدد، توان آن عدد و مقدار اعشاری آن عدد می‌شوند، از یکدیگر جدا کرده و در رجیسترهای متناظر با هرکدام

که به ترتیب a_s، b_s، a_e، b_e، a_m و b_m هستند نگه می‌دارد تا بوسیله‌ی آن‌ها، محاسبات را انجام دهد.در رجیسترهای بالا، حرف s مخفف ،sign حرف e مخفف exponent و حرف ،m کوتاه شده‌ی mantissa هستند. هرکدام از رجیسترهای sign در بالا تنها ۱ بیت برای نمایش علامت عدد دارند. رجیسترهای exponent ذکر شده در بالا، ۸ بیت برای نمایش مقادیر توان دارند. همچنین، رجیسترهای mantissa از ۲۳ بیت برای اعشار و یک بیت ۲۴ام برای نگهداری رقم یکان عدد استفاده می‌کنند. بعد از این حالت، تقسیم‌گر با پیشروی به حالت ۴ ادامه می‌دهد.

- در حالت special cases تقسیم‌گر چک می‌کند تا اگر اعداد ورودی، جزو استثناها بودند، اقدام مناسب را انجام دهد. در این مرحله حالات استثنایی ذیل به ترتیب و اولویت ذکر، کنترل می‌شوند:
  - اگر a و b هر دو NaN باشند که معادل not a number است و یعنی a و b تعریف نشده باشند که خروجی نیز NaN خواهد بود،
  - اگر a و b هر دو inf باشند که معادل اعداد خیلی بزرگ یا بینهایت است که در این صورت خروجی NaN خواهد بود،
  - اگر a مقدار inf داشته باشد که در این صورت اگر b مقدار صفر داشته باشد مقدار خروجی NaN و در غیر این صورت خروجی نیز مقدار inf خواهد داشت،
  - اگر b مقدار inf داشته باشد که خروجی در این صورت صفر خواهد بود،
  - اگر a مقدار صفر داشته باشد که در این حالت، اگر b نیز صفر است خروجی مقدار NaN و در غیر این صورت مقدار صفر خواهد داشت،
  - اگر b حامل مقدار صفر است، خروجی مقدار inf را به خود بگیرد.

در صورت رخداد هر کدام از حالات بالا، مستقیم به حالت put_z میرویم تا خروجی را در حال، روی رجیستر ۳۲ بیتی خروجی Output_z قرار دهیم. در نهایت، اگر هیچ کدام از حالات خاص بالا نبود، با چک کردن این که اعداد a و b دِنرمال هستند یا خیر ادامه می‌دهیم و به حالت normalise_a می‌رویم.

- در حالت normalise_a عدد a را در در صورت نرمال نبودن، تبدیل به یک عدد نرمال در نمایش IEEE می‌کنیم و سپس به حالت normalise_b می‌رویم.

- در حالت normalise_b همان کار بالا را برای عدد b انجام می‌دهیم و هنگامی که نمایش عدد b نیز نرمال بود به حالت divide_0 می‌رویم.

- در حالت divide_0 که اولین گام از الگوریتم تقسیم است، رجیسترهای مورد استفاده در عمل تقسیم را مقداردهی اولیه می‌کنیم. این رجیسترها عبارت اند از z_s که ۱ بیتی است و نشان دهنده‌ی علامت جواب است، z_e که ۸ بیتی است و نشان دهنده‌ی بخش توان جواب است، quotient که یک رجیستر ۵۱ بیتی است و نشان دهنده‌ی خارج قسمت تقسیم است، remainder که یک رجیستر ۵۱ بیتی است و نشان دهنده‌ی باقی مانده‌ی تقسیم است، رجیستر ۶ بیتی count که تعداد مراحل تقسیم را می‌شمارد، رجیستر ۵۱ بیتی dividend که در هر لحظه، مقدار مقسوم فعلی در عملیات تقسیم را نگه میدارد و رجیستر ۵۱ بیتی divisor که مقدار مقسوم علیه را در طول عملیات تقسیم نگهداری می‌کند. تقسیمگر بعد از این مقداردهی این رجیسترها در این مرحله، به مرحله‌ی بعدی تقسیم، یعنی divide_1 می‌رود.

- در حالت divide_1، رجیستر quotient هر دفعه یک بار به سمت چپ شیفت می‌خورد. رجیستر remainder نیز یک بار به سمت چپ شیفت می‌خورد و با ریتم یک بیت در هر مرتبه، از سمت راست آن، divider از طرف بیت پرارزشش وارد می‌شود. سپس تقسیمگر وارد حالت divide_2 می‌شود.

- در حالت divide_2، چک می‌شود اگر مقدار remainder از divisor بزرگتر یا با آن مساوی شد، در بیت سمت راست quotient مقدار یک را می‌ریزد و مقدار divisor را از remainder کم می‌کند. در غیر این صورت مقدار صفر را در بیت سمت راست quotient می‌ریزد. مجموعه عملیات مراحل divide_1 و divide_2 ۵۰ بار تکرار می‌شوند و بعد از اتمام این ۵۰ دفعه، تقسیمگر به حالت divide_3 می‌رود.

- در حالت divide_3 جواب عملیات تقسیم در رجیسترهای مربوطه برای خروجی z قرار داده می‌شوند و رجیسترهای کمکی‌ای که برای مرحله گرد کردن عدد بدست آمده از آن‌ها استفاده می‌شود، مقداردهی می‌شوند. بعد از این مرحله، به حالت normalise_1 می‌رسیم.

- حالت normalise_1 و حالت بعد آن، برای نرمال کردن عدد z خروجی بدست آمده در صورت دِنرمال

بودن آن بکار می‌روند. روند محاسبه‌ی خروجی تقسیمگر از این دو حالات یک بار عبور می‌کند. حالت بعد همان مرحله‌ی normalise_2 است.

- در مرحله‌ی normalise_2 فرایند نرمال کردن عدد خروجی تکمیل می‌شود و سپس تقسیم‌گر به حالت round میرسد.
- در حالت round، تقس‌یمگر با توجه به نتایجی که از مرحله divide_3 نگهداری کرده است، تصمیم می‌گیرد که آیا دقیقتر این است که عدد بدست آمده به بالا گرد شود یا همانطور که هست بماند. بعد از این حالت، تقسیم‌گر وارد حالت pack میشود.
- در حالت pack، تقسیمگر اجزای خروجی بدست آمده را که عبارت اند از z_s، z_e و z_m، دوباره در کنار هم قرار می‌دهد تا به یک عدد واحد در استاندارد نمایش اعداد IEEE تبدیل شوند و این عدد قابل نمایش روی رجیستر Output_z به عنوان خروجی می‌باشد. حالت نهایی تقسیم‌گر، put_z است.
- در مرحله‌ی put_z، که مرحله‌ی نهایی کار تقسیم‌گر است، خروجی بدست آمده روی Output_z قرار می‌گیرد و همینطور سیگنال Output_z_stb برای خبر دادن آماده شدن خروجی ماژول تقسیم‌گر، فعال می‌شود. تقسیمگر در همین حالت می‌ماند تا خروجی تولید شده توسط ماژول دیگری دریافت شود و سیگنال Output_z_ack توسط آن ماژول فعال شود. در این صورت، تقسیم‌گر دوباره به حالت get_a می‌رود برای شروع عملیات تقسیمی جدید.

۲ـ۳ـ۴ Single SQRT (آـ۵)

وظیفه این ماژول محاسبه کردن ریشه دوم یک عدد Single Precision Floating Point است.
این ماژول از روش نیوتن رافسون که قبلا توضیح داده شده استفاده می‌کند.
ابتدا طبق معمول با استفاده از handshake مناسب، عدد a ورودی گرفته می‌شود و محاسبه ریشه دوم $\sqrt{a}$ آغاز می‌گردد. سپس حالت‌های خاص جذر بررسی می‌شوند، و در صورت وقوع هر کدام، پاسخ سریعاً در خروجی قرار می‌گیرد.

Square Root Special Cases	
Operation	Result
$\sqrt{NAN}$	NAN
$\sqrt{inf}$	inf
$\sqrt{a<0}$	NAN
$\sqrt{0}$	0

در صورتی که هیچکدام از حالت‌های بالا رخ نداده بود باید عملیات جذر شروع شود. چون روش جذر توضیح داده شده است در اینجا روی انتخاب حدس اولیه مناسب برای $\sqrt{a}$ تمرکز می کنیم که در روش نیوتن محاسبه می‌شود. برای این منظور بهترین کار استفاده از قابلیت‌هایی است که ذخیره سازی به روش floating point در اختیارمان می‌گذارد.

ساختار عدد ورودی a به شکل زیر است.

$$a = (-۱)^{Sign} \times (۱.Mantissa) \times ۲^{e-۱۲۷}$$

طبق شکل ساختاری عدد ورودی، می توان جذر آن را به شکل زیر نوشت.

$$\sqrt{a} = \sqrt{(۱.Mantissa)} \times ۲^{\frac{(e-۱۲۷)}{۲}}$$

با توجه به این ساختار عددی، برای انتخاب حدس اولیه به راحتی می توان بخش توانی عدد را با شیفت دادن تقسیم بر ۲ کرد.

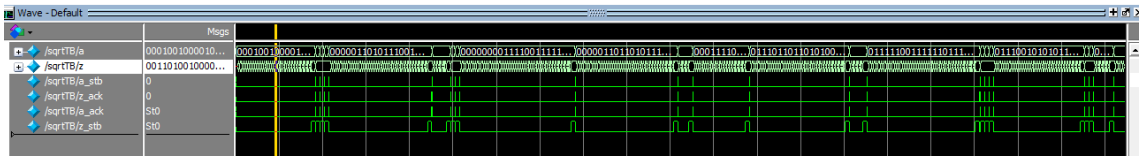
عدد حدس زده شده به شکل زیر است:

$$\sqrt{a} \simeq (۱.Mantissa) \times ۲^{\frac{(e-۱۲۷)}{۲}}$$

اجرای $\frac{۱}{۲}(x_n + \frac{a}{x_n})$ خودش به دو بخش تقسیم کردن و سپس جمع کردن، تقسیم می‌شود. بخش تقسیم بر دو کردن انتهای کار هزینه‌ای ندارد و به راحتی قابل انجام است. شرط خاتمه محاسبات این است که

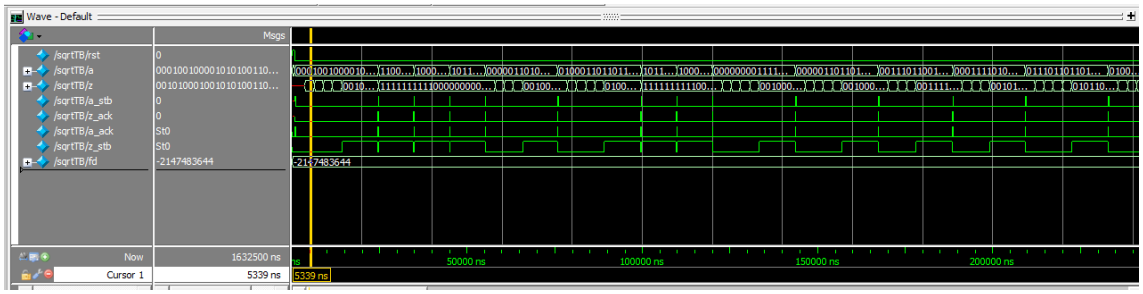
خروجی یک بار محاسبه با خروجی دفعه قبل برابر باشد و یا تعداد باری که محاسبه در حال انجام است به تعداد مشخصی برسد که در فایل defines.v (آـ۱) مشخص شده است.

در حالت بدون استفاده از حدس اولیه مناسب و شروع با یک عدد ثابت، شکل موج مدار به صورت زیر بود.



شکل ۲ـ۷: bad x_0 wave

در شکل مشخص است که به ازای محاسبه هر جذر، به تعداد بالایی تکرار الگوریتم نیوتون نیاز داشتیم که حدود ۶۰ بار بود. اما با تغییر حدس اولیه به حالتی که بالاتر توضیح داده شد به شکل موج زیر رسیدیم.



شکل ۲ـ۸: smart x_0 wave

در شکل بالا مشخص است که برای هر عدد، تنها حدود ۴ تا ۵ بار اجرای الگوریتم کافی است.

برای انجام محاسبات تقسیم و جمع مورد نیاز در این ماژول از ماژول‌هایی استفاده شده است که به صورت تمام و کمال، هر حالتی از اعداد ممیز شناور از جمله حالات غیر نرمال را پشتیبانی می‌کنند. بنابراین این ماژول نیز می‌تواند جذر را روی هر نوع عدد ممیز شناور از جمله غیر نرمال انجام دهد که تست این موارد در بخش شبیه سازی آمده است.

۲ـ۳ـ۵ Double SQRT (آـ۶)

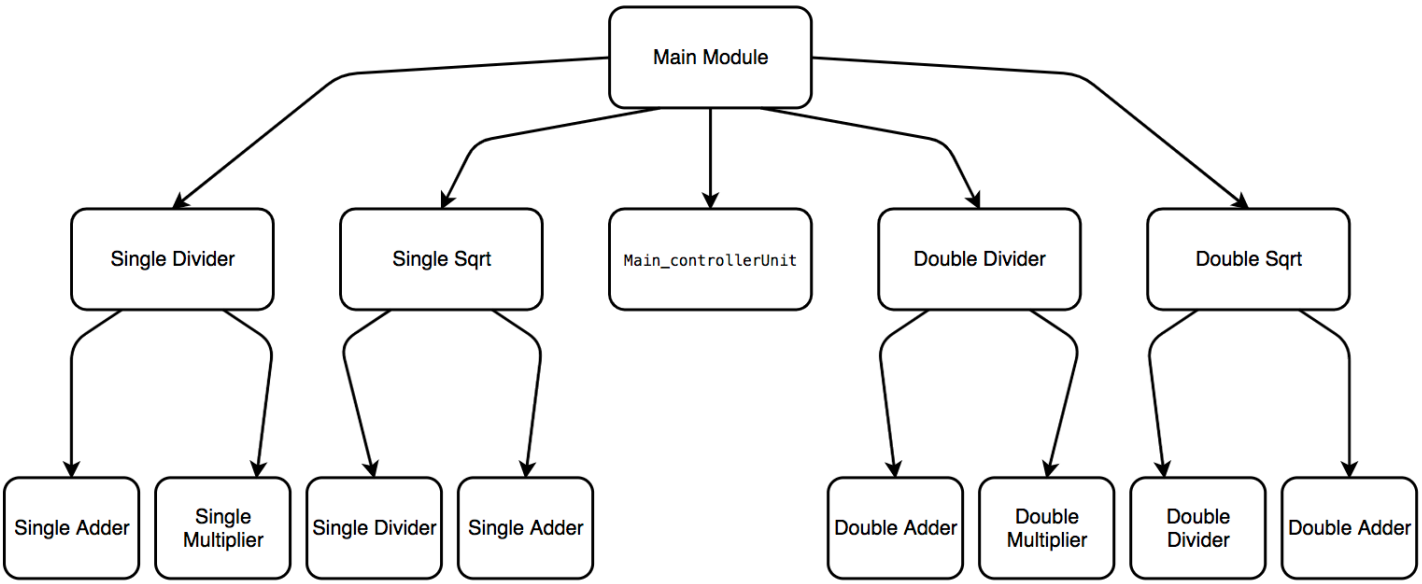
پیاده سازی این بخش مانند بخش قبل است و تنها تفاوتش در تعداد بیت‌ها است.
ساختار عدد ورودی a به شکل زیر است.

$$a = (-۱)^{Sign} \times (۱.Mantissa) \times ۲^{e-۱۰۲۳}$$

عدد حدس زده شده به شکل زیر است.

$$\sqrt{a} \simeq (۱.Mantissa) \times ۲^{\frac{(e-۱۰۲۳)}{۲}}$$

۲ ـ ۴ ساختار درختی سیستم و **Design Hierarchy**



شکل ۲ ـ ۹: Design Hierarchy

فصل ۳

روند شبیه سازی و نتایج حاصله

۳ـ۱ توصیف کلی Test-bench و مقایسه

۳ـ۱ـ۱ Test-bench ماژول‌ها (آـ۱۱)

برای هر ماژول پس از پیاده سازی، یک تست بنچ خاص خود ماژول نوشته شد که حدود ۱۰۰ عدد تصادفی را تولید می‌کند و به ماژول می‌دهد. در آخر عدد ورودی تصادفی و خروجی در فایلی که آدرس آن در defines.v (آـ۱) مشخص شده نوشته می‌شود تا با استفاده از مدل طلایی مقایسه انجام گیرد.

۳ـ۱ـ۲ Test-bench رابط برنامه (آـ۹)

ماژول اینترفیس برنامه که Main نام دارد نیز پس از پیاده سازی برایش دو تست متفاوت توسط دو نفر نوشته شد. اولی توسط خود نویسنده کد، و دومی توسط شخصی دیگر.

این Test-bench برای کل ماژول‌ها است. یعنی به ازای هر ۴ ماژول درون ماژول اصلی، ۱۰۰ عدد تست تصادفی ایجاد می‌کند و آنها را ذخیره می‌کند تا بعداً با مدل طلایی مقایسه شود. همچنین تستی دیگر نیز برای بررسی حالت های خاص در نظر گرفته شده است. (آـ۱۰)

۳ـ۱ـ۳ مدل طلایی آـ۳

مدل طلایی تمام ۴ ماژول به زبان c++ نوشته شده است. مدل طلایی می‌تواند فایلی که توسط ماژول‌ها در شبیه سازی کد verilog ساخته شد را بخواند، خودش با مدل طلایی مقایسه کند و نتیجه را نمایش دهد.
برای تبدیل رشته خوانده شده از ورودی به عدد، آن را با استفاده از تابع strtoull یا to ulong به عدد تبدیل می‌کنیم.
برای آنکه این عدد را بتوانیم درون یک متغییر float برای تست ماژول‌های single precision و درون یک متغییر double برای تست ماژول‌های double precision بریزیم باید عملیات خاصی انجام دهیم. برای این کار از عملیات زیر استفاده می‌کنیم.

```
*(int*)&input
```

این کار ابتدا اشاره گر به متغیر را می‌گیرد، سپس آن را به اشاره گر int تبدیل می‌کند و در آخر با گرفتن * روی کل این قسمت، به مقدار عددی داخل آن خانه حافظه دسترسی دارد که می‌توان آن را خواند یا رویش نوشت. باید توجه داشت که واحد خواندن از حافظه بایت است، به همین دلیل نمی‌توان این مقدار را به صورت باینری نمایش داد. پس برای حل این موضوع نیز تابعی نوشته شد تا با گرفتن متغیر، و شیفت دادن آن بتواند پس از به تعداد بیت ورودی، مرحله، مقدار داخل آدرس فیزیکی متغیر را به رشته تبدیل کند.
عمل تقسیم را با استفاده از تقسیم عادی زبان c++ و عمل جذر را با استفاده از تابع sqrt در کتابخانه cmath انجام دادیم. نتایج این بررسی‌ها در ادامه خواهد آمد.

۳ـ۱ـ۴ تست **corner case** ها (آـ۱۰)

علاوه بر تعداد زیادی تست تصادفی، حالتهای گوشه، مانند اعداد خیلی بزرگ و خیلی کوچک یا اعداد غیر نرمال و همچنین حالتهای خاص مانند تقسیم بر ۰ یا استفاده از بی‌نهایت در محاسبات نیز به ماژول‌ها داده و تست شد.

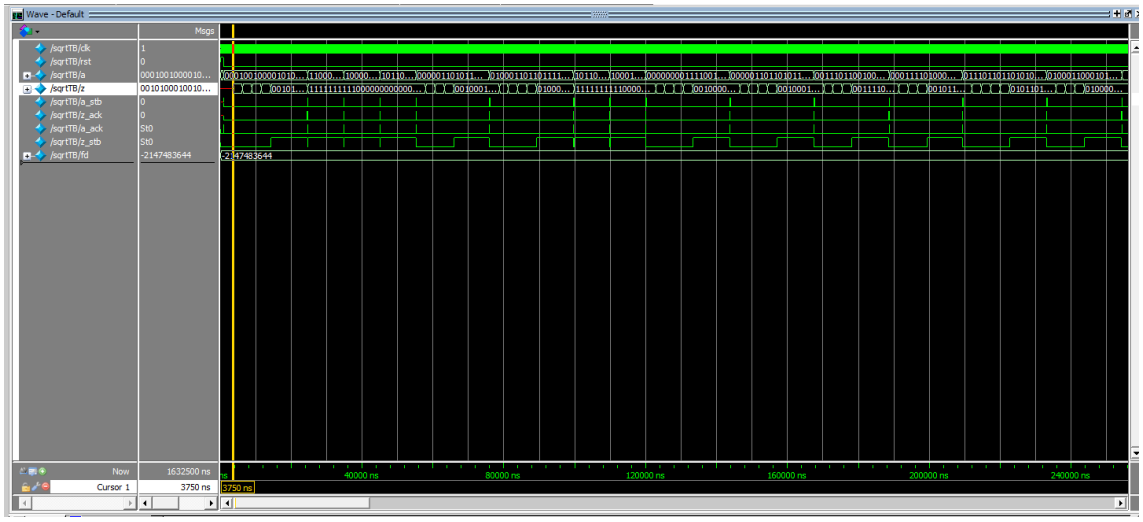
۳ـ۲ توصیف روند شبیه سازی سخت افزار

برای شبیه سازی سخت افزار از نرم افزار ModelSim استفاده شد. ابتدا ماژول‌ها به صورت تکی تست شدند و سپس ماژول در برگیرنده تمام آنها به صورت جداگانه تست شد.

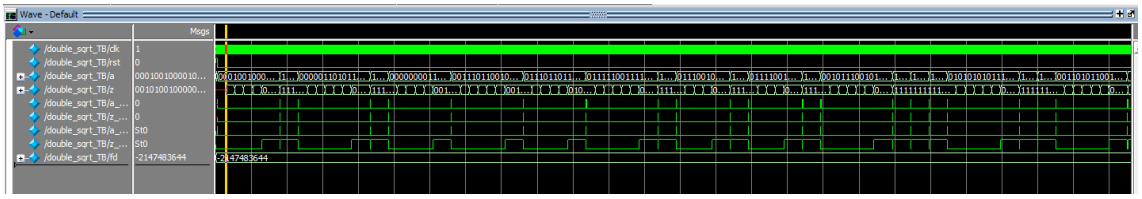
۳ـ۲ـ۱ مشاهده شکل موج‌ها

۳ـ۲ـ۱ـ۱ شکل موج تست تصادفی ماژول‌ها

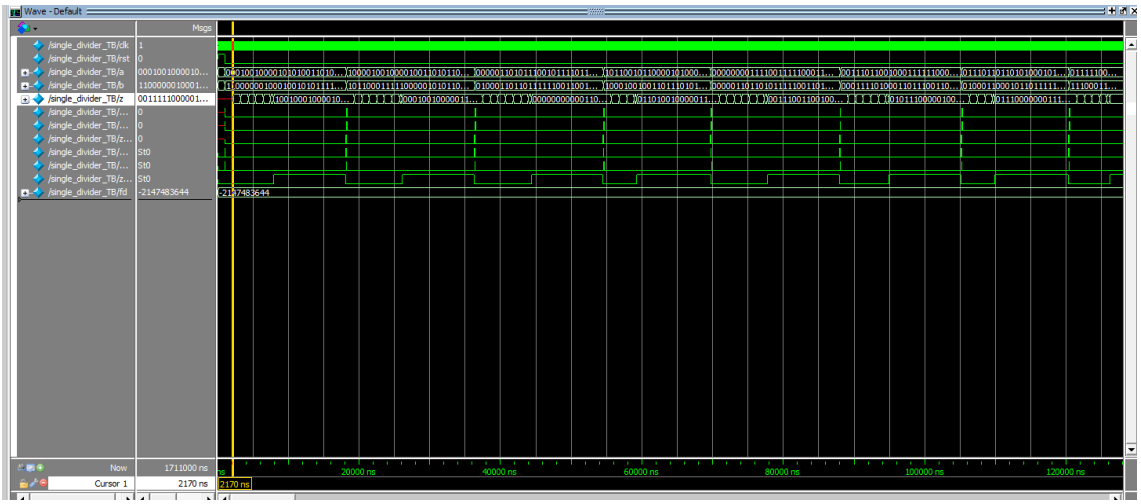
در ادامه شکل موج تست ۴ ماژول به صورت تکی با ورودی‌های تصادفی می‌آید.



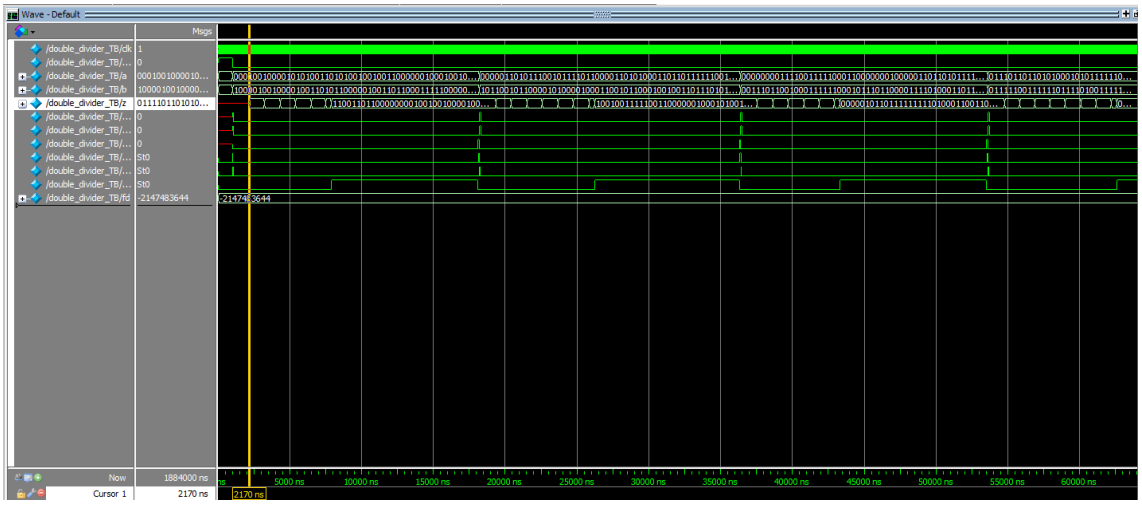
شکل ۳ـ۱: single square root wave



شکل ۳ـ۲: double square root wave



شکل ۳ـ۳: single divider using newton method wave

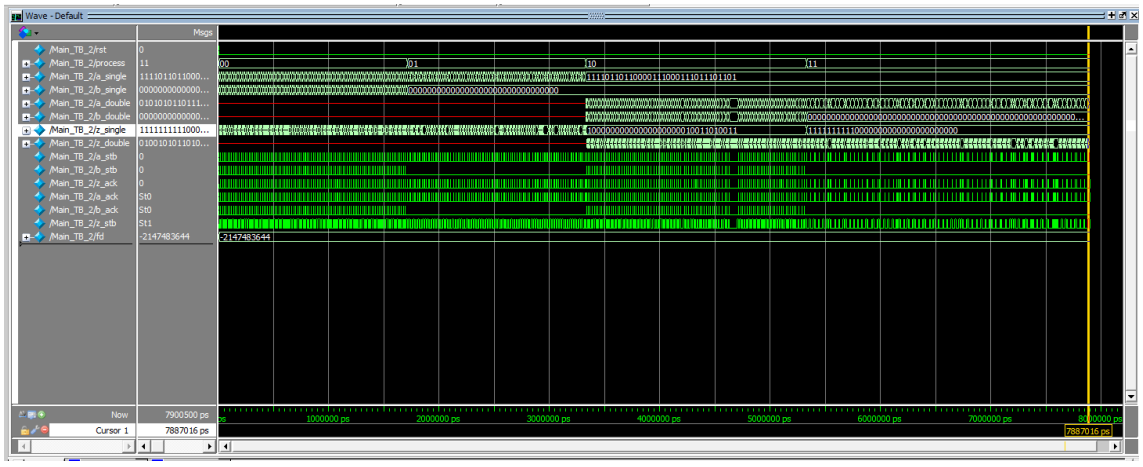


شکل ۳ـ۴: double divider using newton method wave

۳ـ۲ـ۱ـ۲ شکل موج تست یکپارچگی برای ورودی‌های تصادفی

در ادامه شکل موج تست ماژول Main که ماژول در برگیرنده ۴ ماژول ذکر شده است می‌آید.

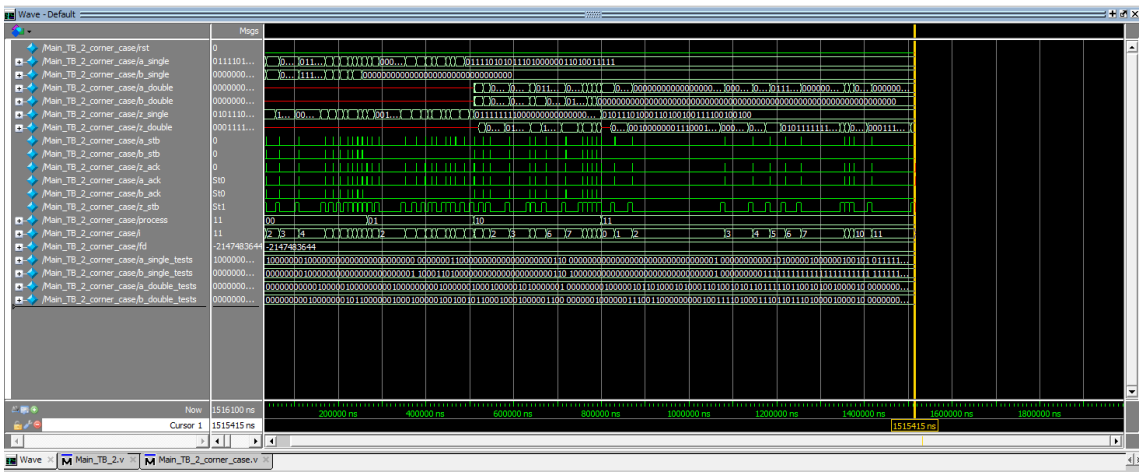
در این تست به هر ۴ ماژول داخل آن حدود ۱۰۰ تست تصادفی داده شد و نتایج آن بررسی شد که مقایسه در بخش‌های بعدی می‌آید.



شکل ۳ـ۵: Integration test wave

۳ـ۲ـ۱ـ۳ شکل موج تست یکپارچگی برای **corner case** ها

همانطور که مشخص است تعداد تست‌های خاص کمتر است اما مواردی را پوشش می‌دهند که ممکن است در تست‌های تصادفی دیده نشوند.



شکل ۳ـ۶: Integration test corner cases wave

۳ـ۳ مشاهده ورودی و خروجی‌ها و مقایسه با مدل طلایی

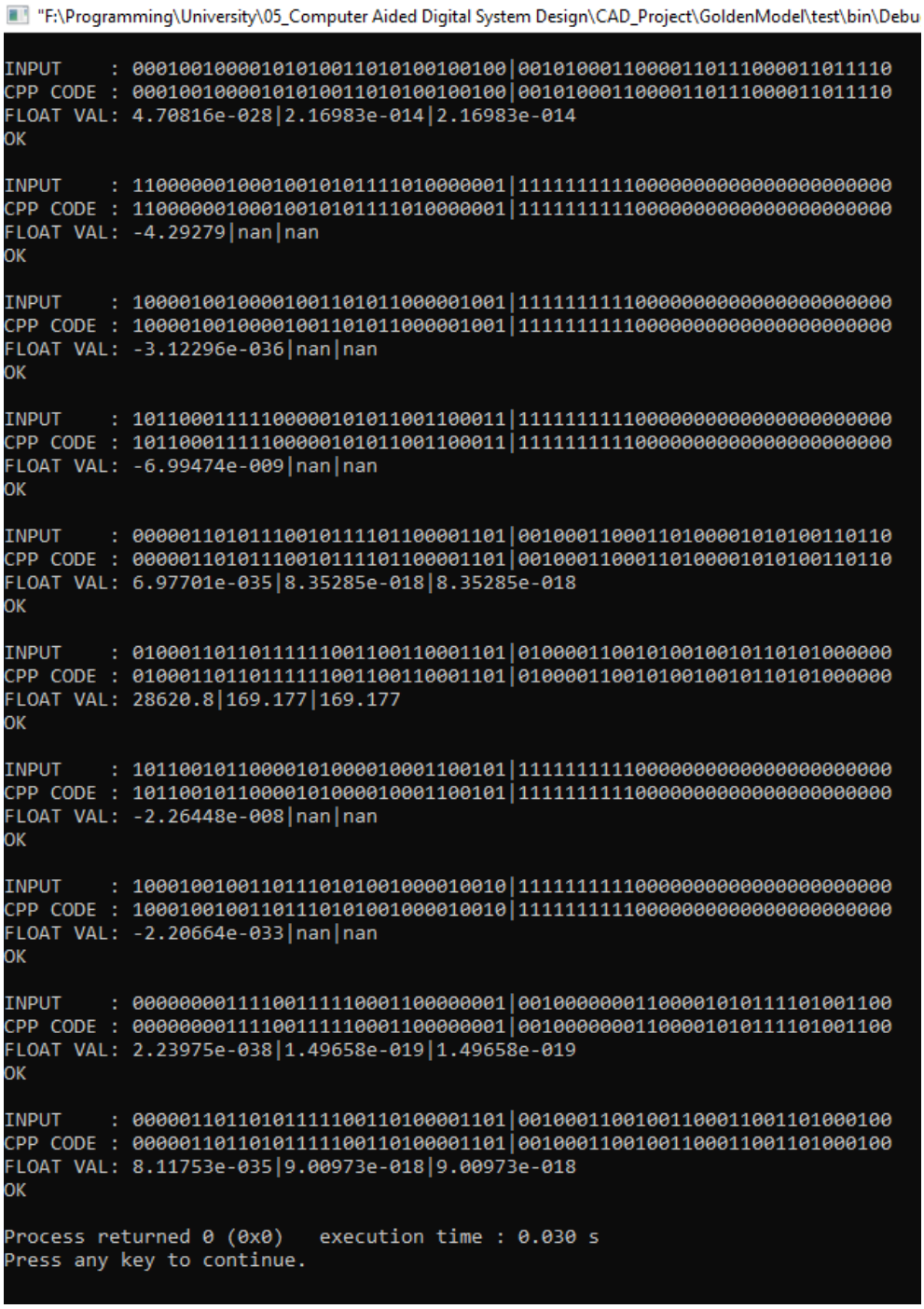
همانطور که گفته شد، کد وریلاگ تست بنچ‌ها، ورودی‌های خود را به صورت تصادفی یا حالت‌های خاص می‌سازد و سپس آن ورودی‌ها را به همراه خروجی در فایلی می‌نویسد. شکل‌های زیر، نتیجه مقایسه این خروجی‌ها با خروجی گرفته شده از مدل طلایی است.

اعداد مقابل خط INPUT ورودی و خروجی کد سخت افزاری است و اعداد مقابل CPP CODE ورودی داده شده و خروجی گرفته شده از مدل طلایی است که دقیقا زیر ورودی و خروجی سخت افزاری آمده تا قابلیت مقایسه بیت به بیت فراهم باشد. در خط بعد مقابل FLOAT VAL یا DOUBLE VAL مقدار عددی مبنای ۱۰ ورودی داده شده، سپس خروجی سخت افزار و سپس خروجی مدل طلایی آورده شده است. در خط آخر نیز اگر خروجی سخت افزاری با خروجی مدل طلایی یکی باشد OK و در غیر این صورت ERROR چاپ می شود.

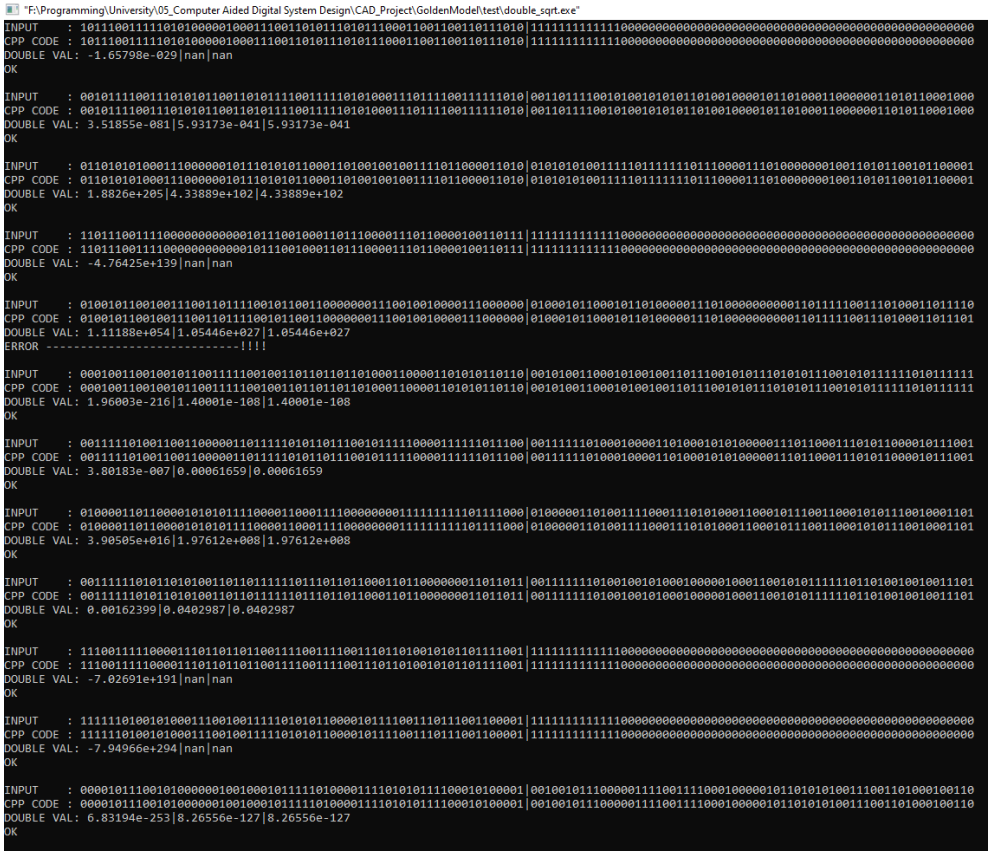
مواردی که OK چاپ شده یعنی هیچ تفاوتی در بیت‌های خروجی و چیزی که در مدل طلایی محاسبه شده نبوده است.

در مواردی که ERROR چاپ شده نیز می توان دید که تنها به اندازه یک بیت کم ارزش، تفاوت بین خروجی سخت افزاری و خروجی مدل طلایی وجود دارد. علت وجود این اختلاف آن است که در الگوریتم‌های اصلی محاسبات ممیز شناور، با تعداد بیت بیشتری محاسبات را انجام می دهند و سپس عدد گرد شده را خروجی می دهند، اما در محاسبات ما، صرفا از تعداد بیت مشخص شده استفاده کرده ایم نه بیشتر. این تفاوت بسیار کم است و حتی نمایش مبنای ۱۰ اعداد که چاپ شده اند هیچ تفاوتی ندارند. از طرفی عدد مدل طلایی نیز عدد دقیق نیست و همانطور که گفته شد تقریب زده شده به نزدیک ترین عدد است. بنابراین می‌توان از این اختلاف اندک چشم پوشی کرد.

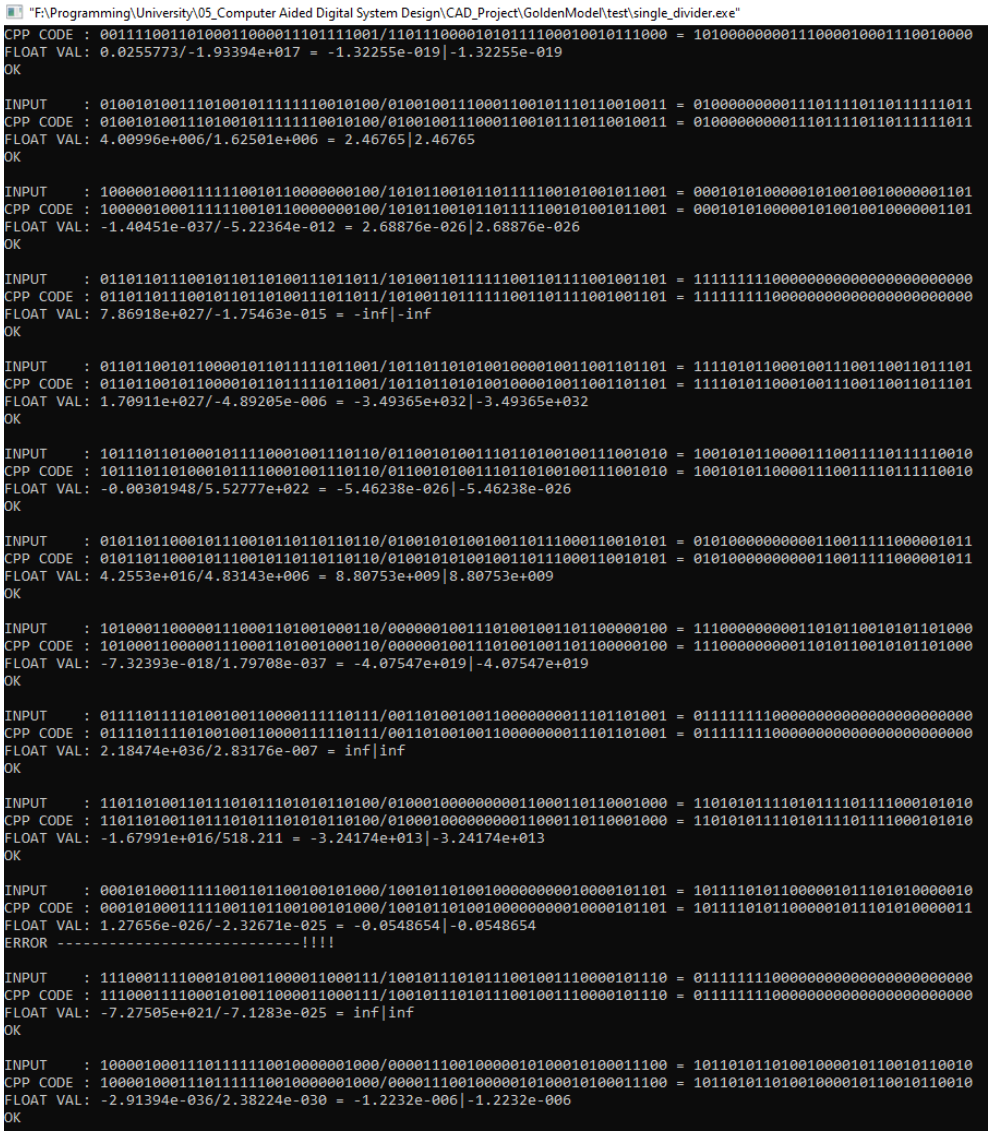
۳ـ۳ـ۰ـ۱ مقایسه نتایج ماژول‌ها روی تست‌های تصادفی



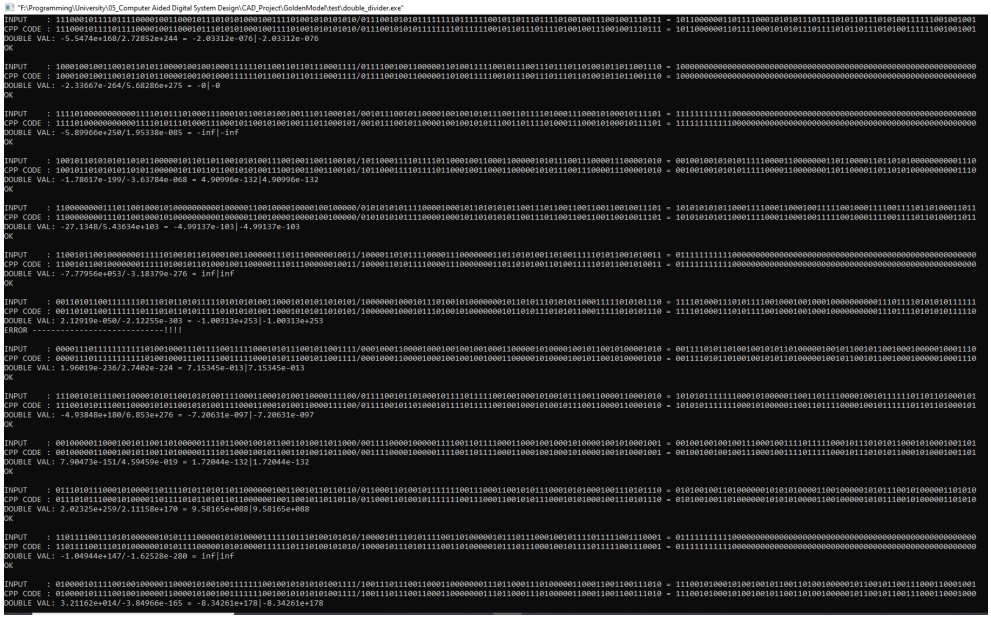
شکل ۳ـ۷: Single Square Root Test



شکل ۳ـ۸: Double Square Root Test

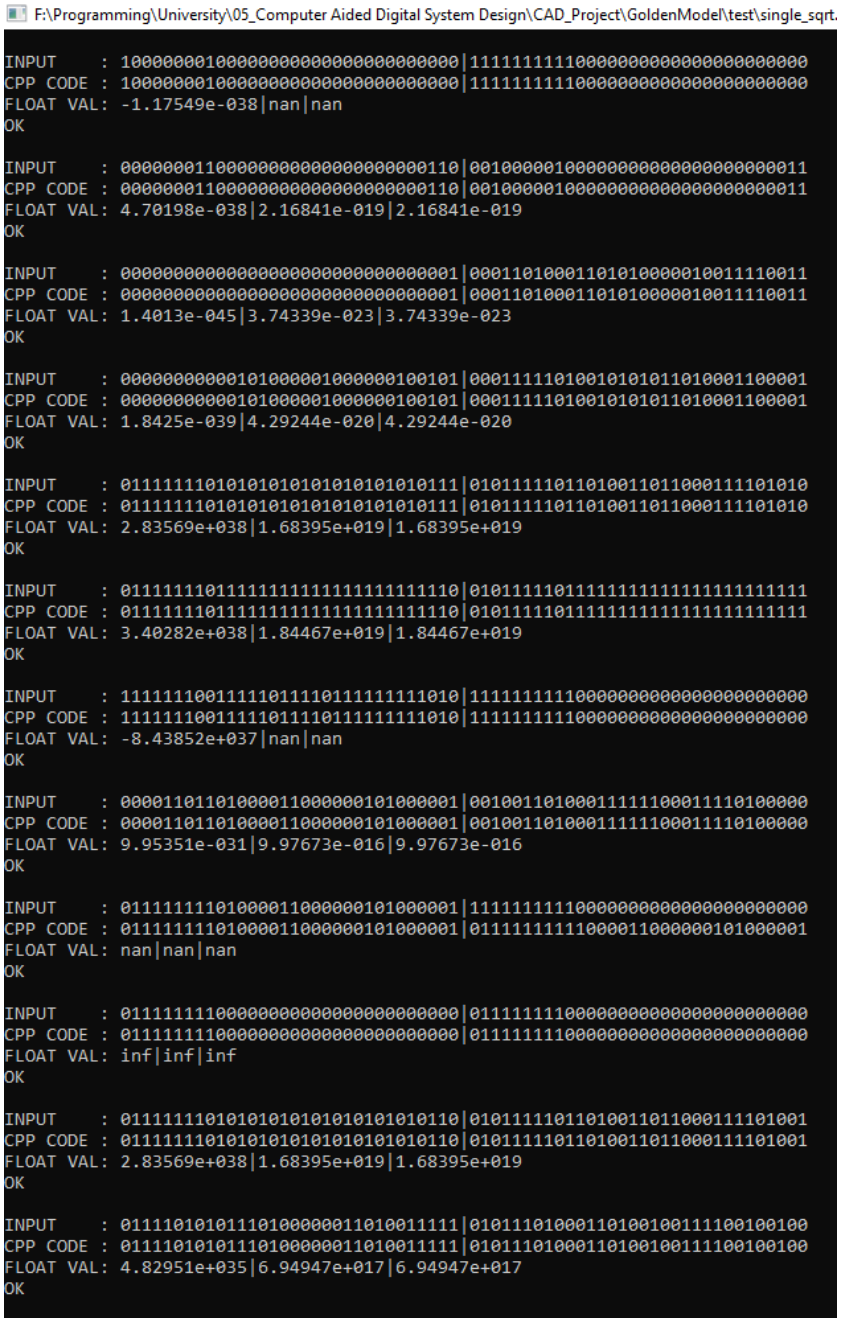


شکل ۳ـ۹: Single Divider Using Newton Method Test

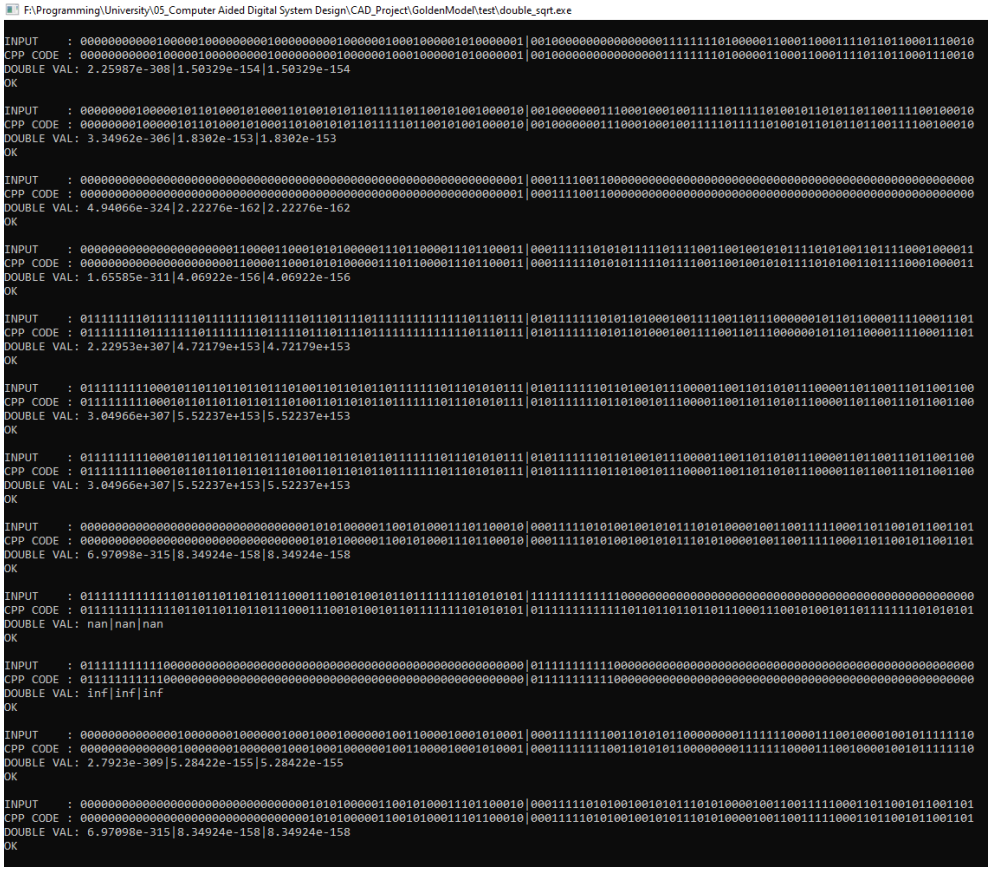


شکل ۳ـ۱۰ : Double Divider Using Newton Method Test

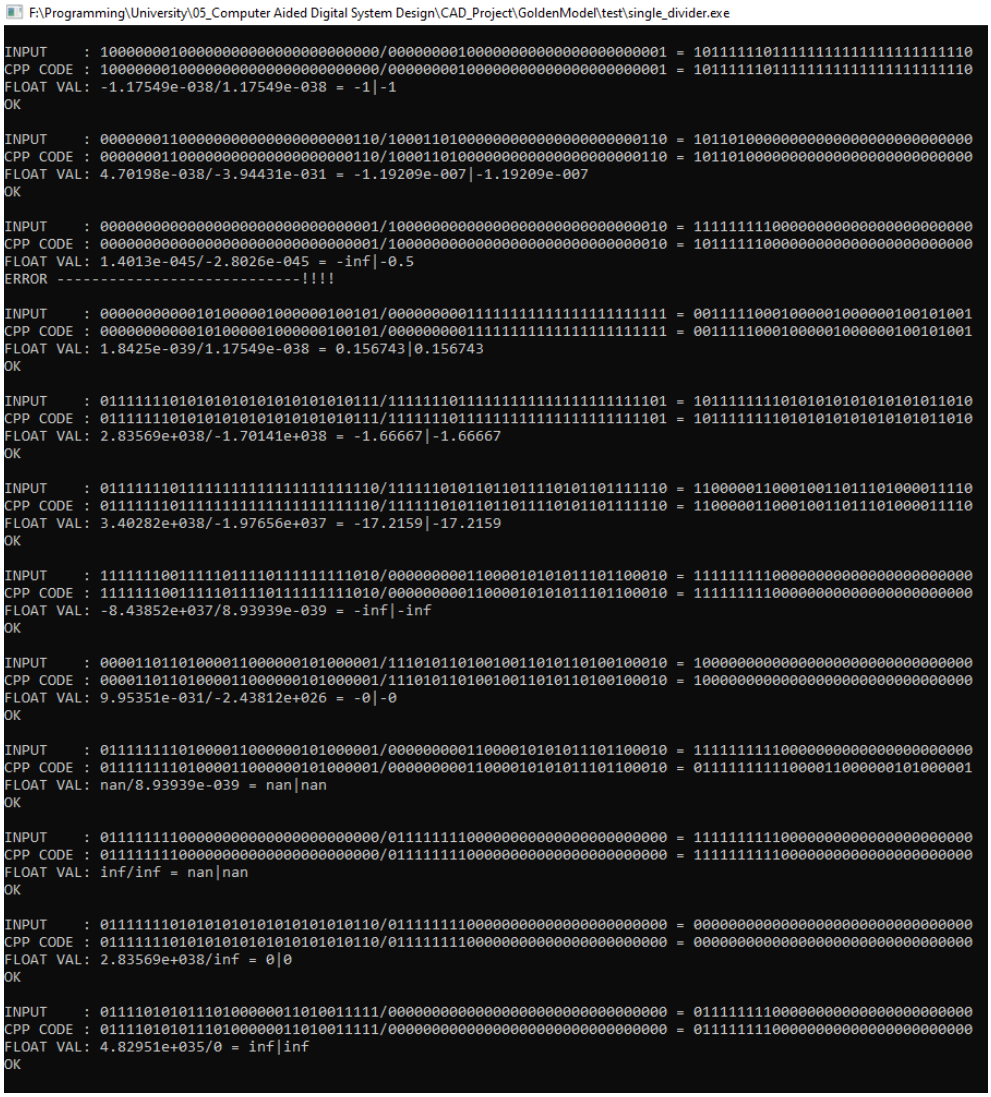
۳ـ۳ـ۰ـ۲ مقایسه نتایج ماژول‌ها روی تست‌های **corner case**



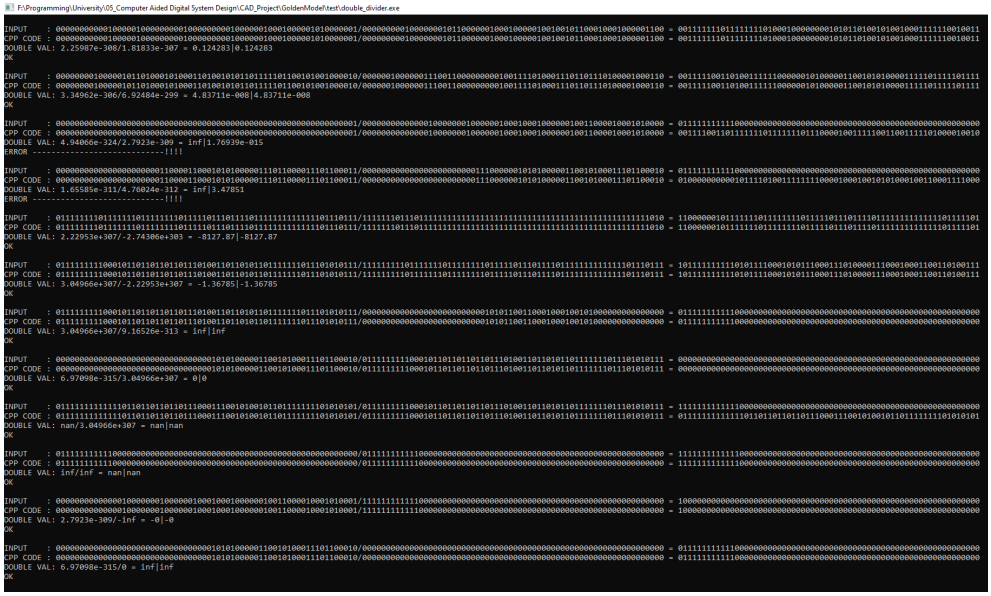
شکل ۳ـ۱۱ : Single Square root Corner Case Test



شکل ۳ـ۱۲ : Double Square root Corner Case Test



شکل ۳ـ۱۳ : Single Divider Using Newton Method Corner Case Test



شکل ۳ـ۱۴ : Double Divider Using Newton Method Corner Case Test

در تست‌های تقسیم همانطور که دیده می‌شود، تست‌هایی که مخرجشان عدد بسیار کوچکی است به درستی عمل نمی‌کنند. علت این موضوع همان است که محاسبه تقسیم به روش نیوتن نیازمند محاسبه معکوس عدد مخرج است و متاسفانه معکوس اعداد غیر نرمال ممیز شناور اصلا قابلیت نمایش در این استاندارد را ندارند. این موضوع در آخر بخش ۲ـ۳ـ۱ به تفصیل توضیح داده شده است.

اما قابل توجه است که صورت کسر می تواند غیر نرمال باشد و این محدودیت فقط روی مخرج است.

در مورد تست‌های جذر مشخص است که حتی روی کوچک‌ترین عدد غیر نرمال که در تست هم آمده درست جواب می‌دهد و ورودی آن هیچگونه محدودیتی از نظر کوچکی یا بزرگی ندارد.

۳ـ۴ آموزش نحوه‌ی تست برنامه

برای تست مدار انجام دو مرحله کافی است.

- شبیه سازی Main_TB_2.v در پوشه‌ی ModelSim
- صحت سنجی نتایج حاصله، با اجرای کدهای با پسوند exe. در پوشه‌ی GoldenModel/test

عمل اول ورودی‌ها و خروجی‌های ماژول سخت افزاری را در پوشه‌ی ModelSim/output می‌ریزد. و عمل دوم ورودی‌ها را از این پوشه می‌خواند، مدل طلایی را روی ورودی‌ها اجرا می‌کند و در آخر نتیجه‌ی مقایسه خروجی های مدل طلایی و خروجی های ماژول را نمایش می‌دهد.

فصل ۴

پیاده سازی و نتایج حاصله (Implementation)

۴ـ۱ سنتز سیستم روی FPGA با استفاده از ابزار CAD

با استفاده از ابزار Vivado و روی دستگاه Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit [۱] سنتز و implementation کل سیستم انجام گرفت و همچنین Behavioral Simulation ، Functional Simulation و Timing Simulation انجام شد که نتایج آن در ادامه می‌آید.

همچنین برای کلاک سیستم، یک MMCM [۲] از ip های موجود در vivado استفاده شد و پس از تست‌های مختلف، کلاک خروجی بیشینه که سیستم را دچار مشکلات زمان بندی نکند به دست آمد.

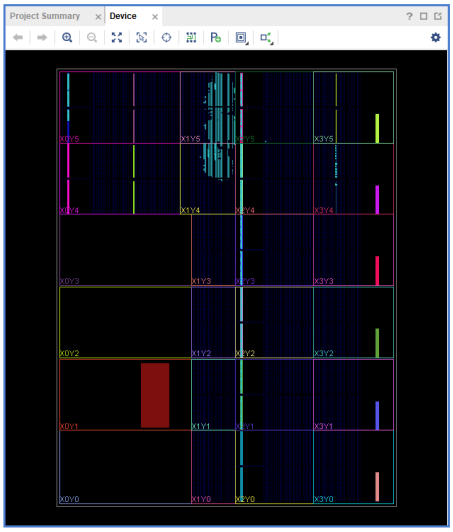
---

[1] www.xilinx.com/zcu104

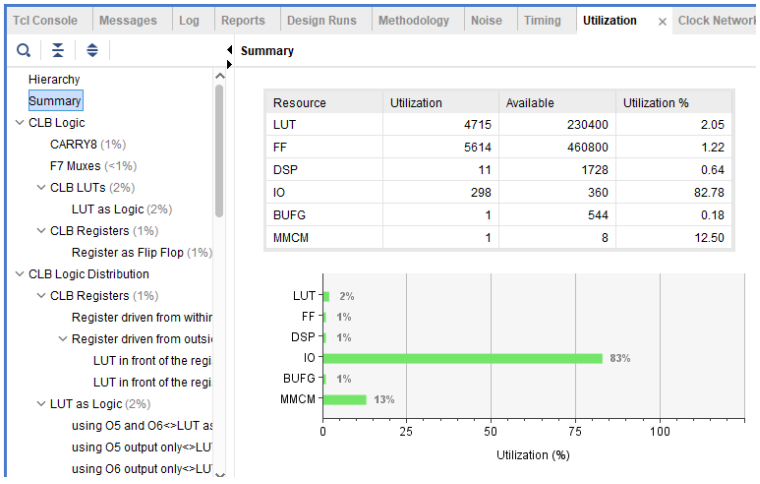
[2] Mixed-Mode Clock Manager

۴ـ۲ گزارش پیاده سازی

۴ـ۲ـ۱ مساحت و بهره برداری

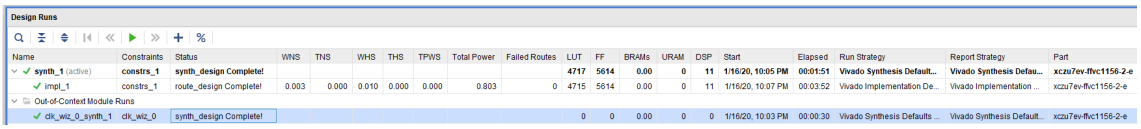


شکل ۴ـ۱ : Device Schematic



شکل ۴ـ۲ : Utilization Summary

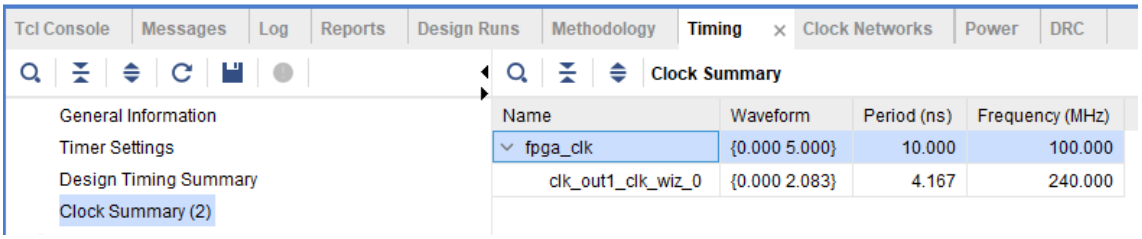
۴ـ۲ـ۲ تعداد فلیپ فلاپها و LUTها



شکل ۴ـ۳: LUT, FF

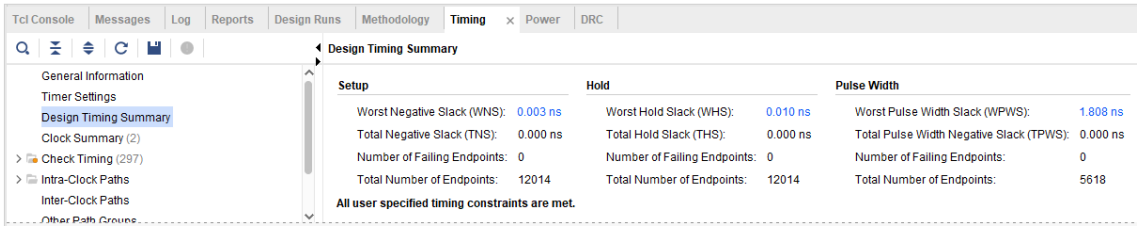
#LUT	۴۷۱۵
#FF	۵۶۱۴

۴ـ۲ـ۳ زمان بندی، کلاک و حداکثر فرکانس



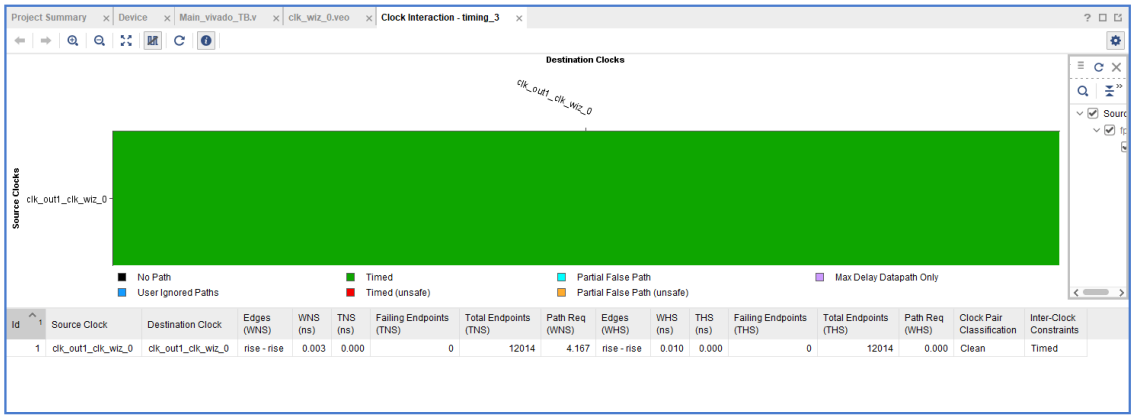
شکل ۴ـ۴: Clock Summary

حداکثر فرکانس کلاک که بدون مشکل قابل پیاده سازی است، مقدار 240MHz است.



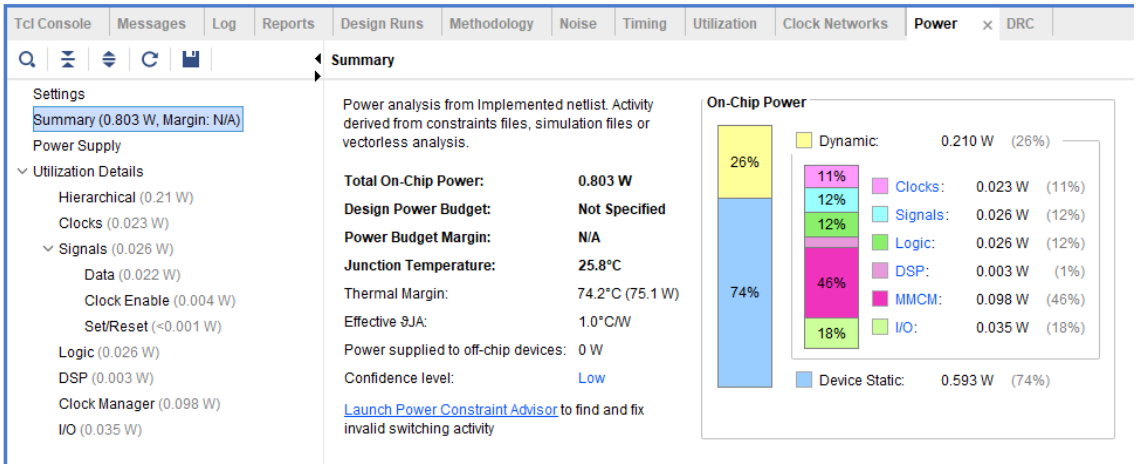
شکل ۴ـ۵: Timing Summary

در بخش سنتز و نتایج زمانی آن، مقدار THS صفر نبوده و مشکل نقض hold time وجود داشت که چنین مشکلی در بخش پیاده سازی دیده نمی‌شود. طبق تحقیقات انجام گرفته، این موضوع در سنتز حل نمی‌شود و به علت اینکه در بخش پیاده سازی، تاخیر سیم‌ها به صورت دقیق قرار داده می‌شوند، باعث افزایش تاخیر کلی بخش combinational و رفع خطای hold time می‌شود. (کاری که با گذاشتن بافر هم می‌توان انجام داد.)



شکل ۴ـ۶: Clock Interaction

۴ـ۲ـ۴ توان

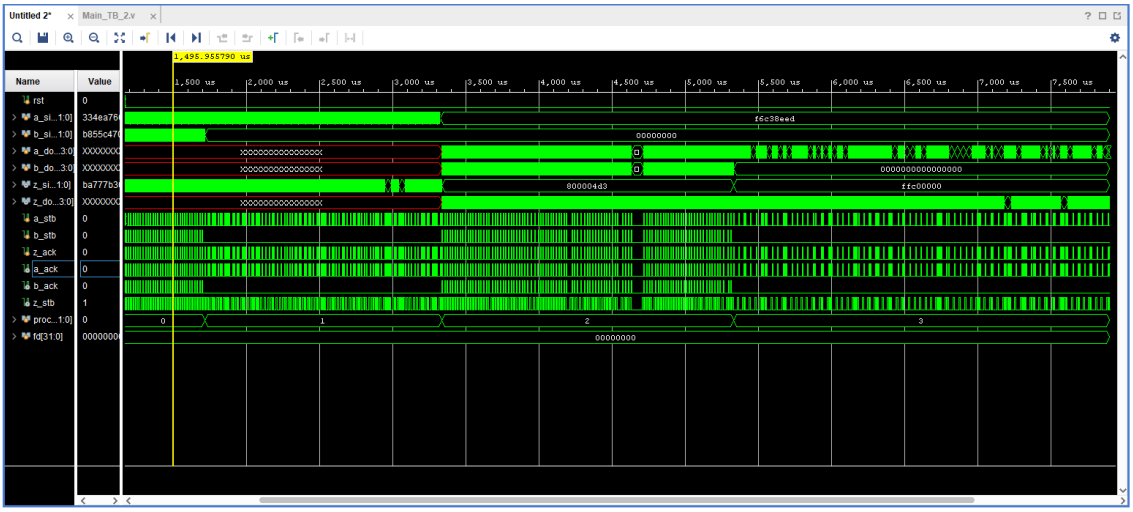


شکل ۴ـ۷: Power Summary

۴ـ۳ گزارش شبیه سازی در vivado

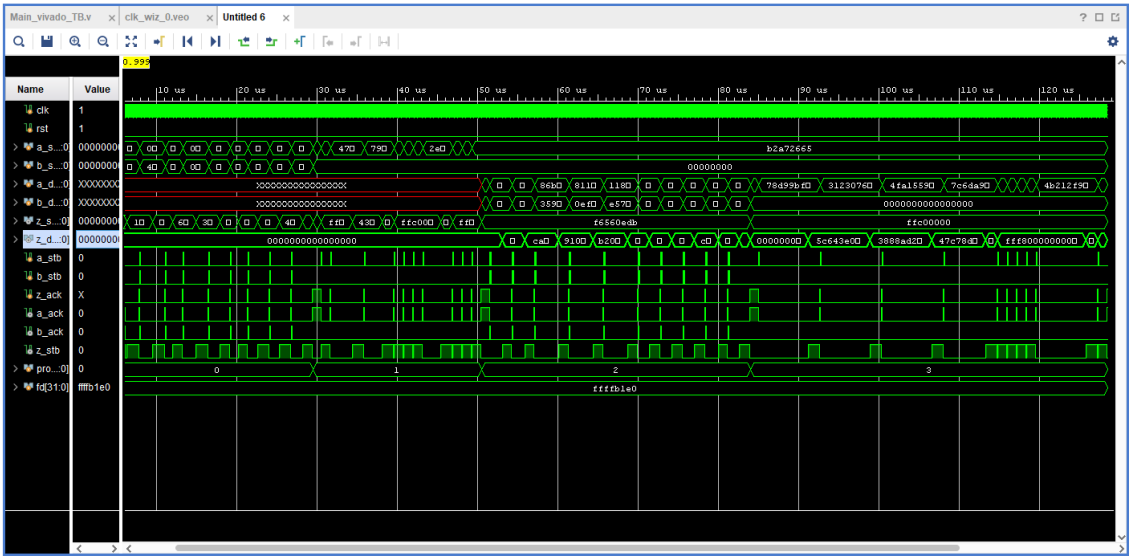
در چند مرحله برای اطمینان از صحت عملکرد سخت افزار پس از سنتز، نتایج را از شبیه سازی‌های نرم افزار vivado به مدل طلایی دادیم و با نتایج حاصل از مدل طلایی مقایسه کردیم که نتایج آن در ادامه می‌آید.

۴ـ۳ـ۱ گزارش شبیه سازی Behavioral

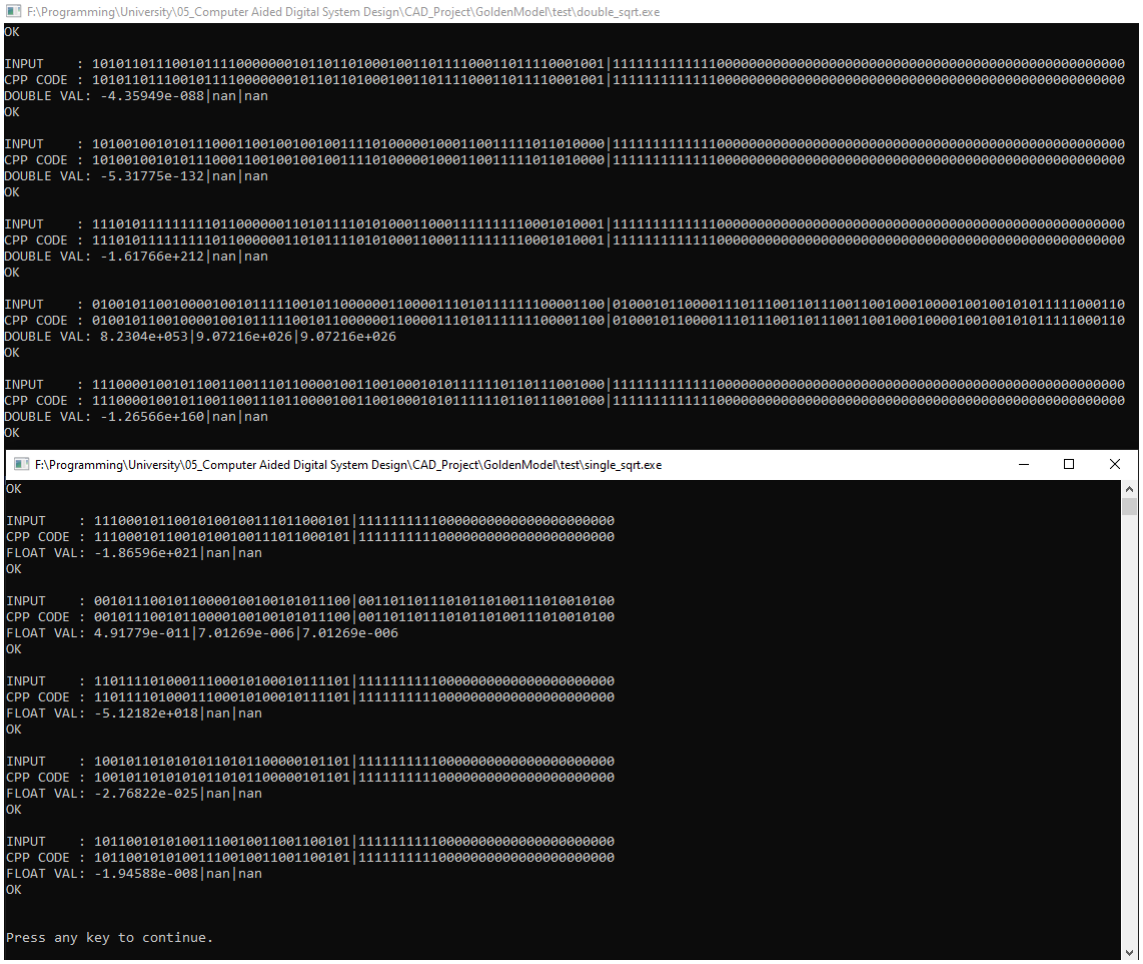


شکل ۴ـ۸: Vivado Behavioral Simulation Waveform

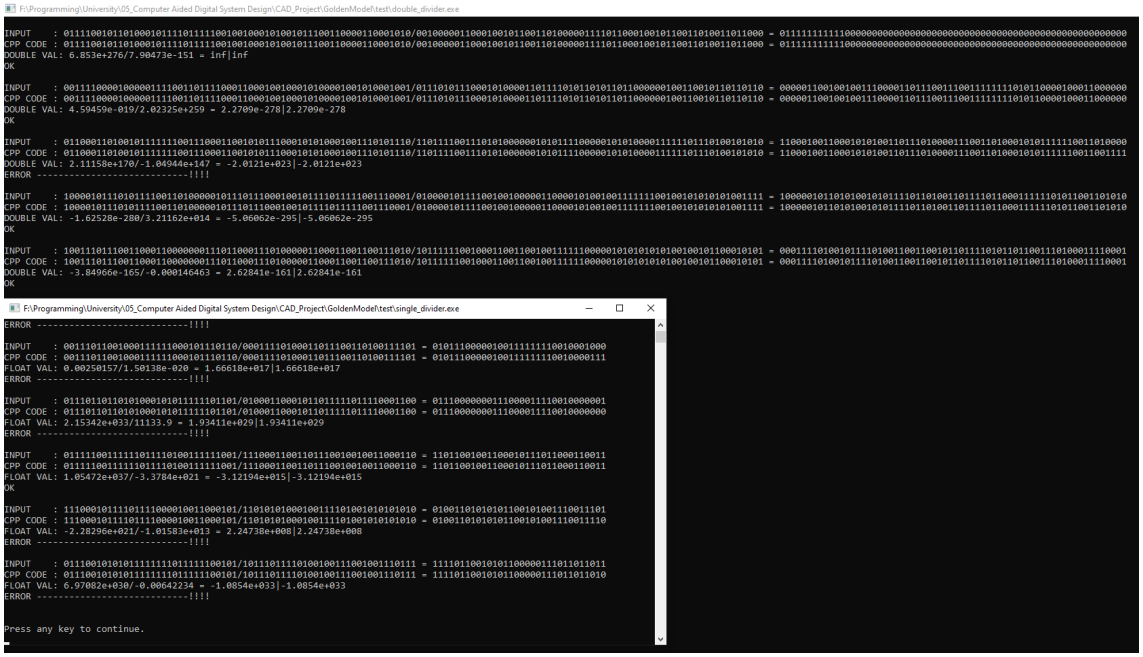
۴ـ۳ـ۲ گزارش شبیه سازی Functional



شکل ۴ـ۹: Vivado Functional Simulation Waveform

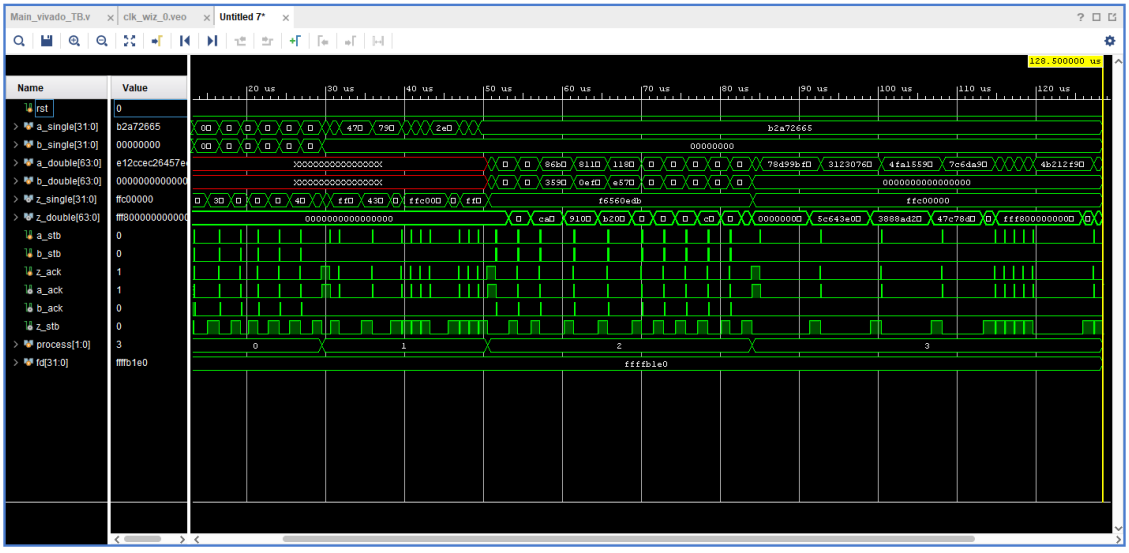


شکل ۴ـ۱۰ : Vivado Functional Simulation Square Root Test

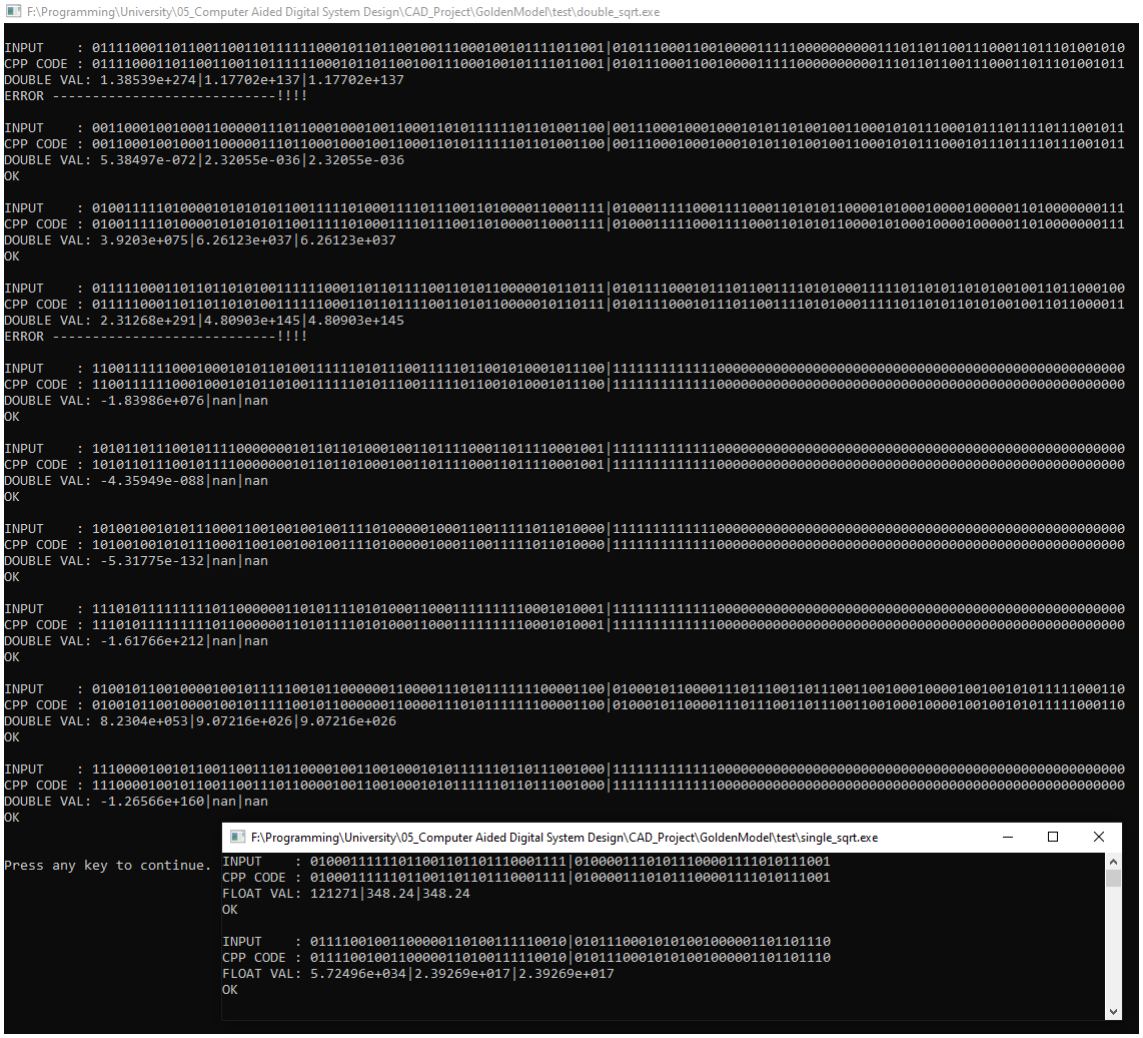


شکل ۴ـ۱۱ : Vivado Functional Simulation Divider Test

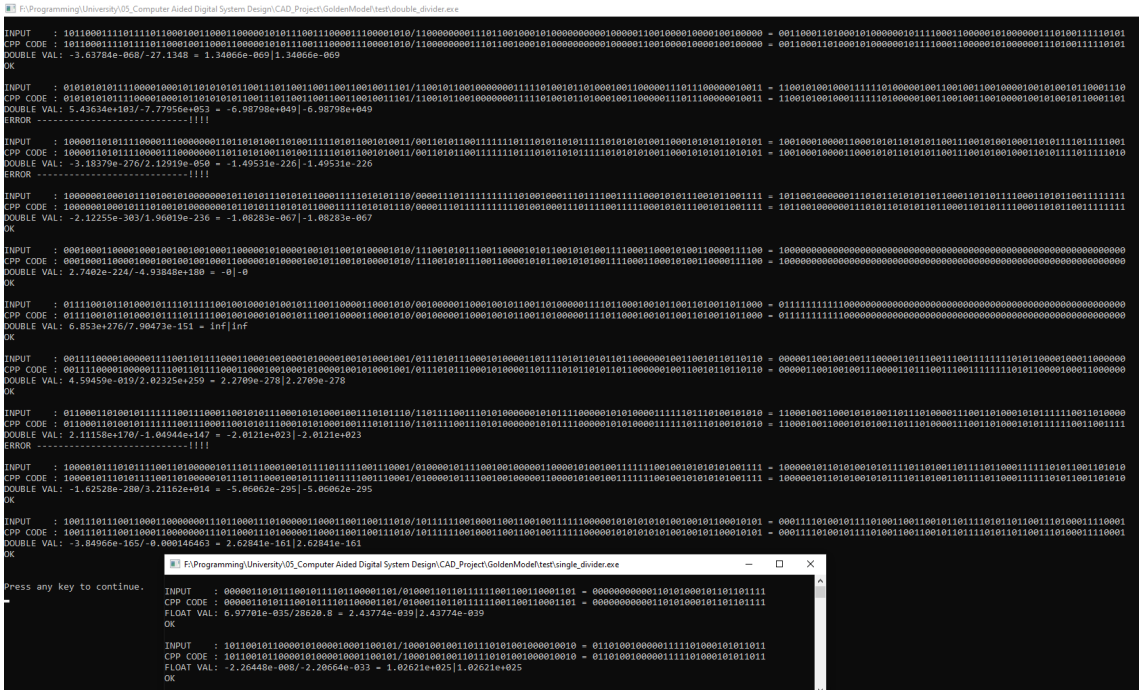
۴ـ۳ـ۲ گزارش شبیه سازی Timing



شکل ۴ـ۱۲ : Vivado Timing Simulation Waveform



شکل ۴ـ۱۳ : Vivado Timing Simulation Square Root Test



شکل ۴ـ۱۴ : Vivado Timing Simulation Divider Test

همانطور که مشخص است در تمام شبیه سازی‌ها، خروجی سخت افزار با نتایج مدل طلایی همخوانی دارد.

فصل ۵

نتیجه‌گیری

در این گزارش ابتدا به توضیح استاندارد ممیز شناور برای ذخیره سازی اعداد اعشاری پرداختیم. سپس طی بررسی‌هایی که داشتیم، به این نتیجه رسیدیم که برای پیاده سازی تقسیم و جذر این اعداد، روش نیوتن روشی سریع و مناسب است. این روش و محاسبات ریاضیاتی آن انجام شد و سپس با استفاده از آنها ماژول‌های لازم ساخته شدند. برای تست صحت عملکرد ماژول‌ها مدل طلایی به زبان c++ طراحی شد و از توابع این زبان به عنوان استاندارد اصلی اعمال ریاضیاتی روی اعداد ممیز شناور استفاده شد.

برای اجرای الگوریتم نیوتن نیاز به یک حدس اولیه داریم، که با استفاده از ساختار ذخیره سازی ممیز شناور، توانستیم حدس بسیار دقیقی بزنیم، که طبق شکل موج‌ها می‌توان دید که در تعداد تکرار الگوریتم بسیار کم، سریعاً به جواب نهایی می‌توان رسید.

لازم به ذکر است که برای تقسیم، دو ماژول با دو روش متفاوت وجود دارد که هر دو نیز توضیح داده شده اند.

در مقایسه‌های انجام شده با مدل طلایی به دقت بسیار بالایی رسیدیم به طوری که تنها تفاوت خروجی ماژول‌ها با نتایج مدل طلایی در حداکثر یک بیت کم ارزش بود، که آن هم تنها در زمان‌هایی که نمی‌توان محاسبه را با دقت کامل انجام داد پیش می‌آید که در گرد کردن عدد، مدل طلایی دقت بالاتری داشت.

از نظر تست حالت‌های خاص، تمام ماژول‌ها از تمام اعداد بسیار بزرگ و کوچک پشتیبانی می‌کنند و حتی اعداد غیرنرمال نیز به خوبی محاسبه می‌شوند.

تنها در ماژول تقسیم و آن هم به علت عدم تقارن توزیع اعداد ممیز شناور در توان‌های بسیار کوچک و بزرگ،

نمی‌توانیم معکوس اعداد بسیار کوچک را محاسبه کنیم. به همین دلیل تنها ماژول تقسیم و آن هم مخرج آن نباید عدد غیرنرمال باشد و غیر از آن هیچ محدودیتی روی ورودی‌های سیستم نیست.

در انتها این سیستم را با استفاده از نرم افزار vivado سنتز، پیاده سازی و شبیه سازی کردیم، که توانایی رسیدن به فرکانس حدودی $۲۴۰ MHz$ نشان از بهینه سازی کل سیستم برای فرکانس‌های بالا و عدم وجود مسیرهای بحرانی غیرمتعارف دارد.

پیوست آ

ضمیمه

آ ـ ۱ Modules

آ ـ ۱ ـ ۱ defines.v

```
`define DEBUGGING 0
`define FLOATING_POINT_SINGLE_2 32'b01000000000000000000000000000000
`define FLOATING_POINT_DOUBLE_2 64'b0100000000000000000000000000000000000000000000000000000000000000
`define SINGLE_SQRT_MAX_LOOP 50
`define DOUBLE_SQRT_MAX_LOOP 200
`define SINGLE_DIVIDER_MAX_LOOP 30
`define DOUBLE_DIVIDER_MAX_LOOP 60

`define SINGLE_DIVIDER_OUTPUT_FILE_NAME "output/single_divider_output.txt"
`define DOUBLE_DIVIDER_OUTPUT_FILE_NAME "output/double_divider_output.txt"
`define SINGLE_SQRT_OUTPUT_FILE_NAME "output/single_sqrt_output.txt"
`define DOUBLE_SQRT_OUTPUT_FILE_NAME "output/double_sqrt_output.txt"

`define VIVADO_SINGLE_DIVIDER_OUTPUT_FILE_NAME "single_divider_output.txt"
`define VIVADO_DOUBLE_DIVIDER_OUTPUT_FILE_NAME "double_divider_output.txt"
`define VIVADO_SINGLE_SQRT_OUTPUT_FILE_NAME "single_sqrt_output.txt"
`define VIVADO_DOUBLE_SQRT_OUTPUT_FILE_NAME "double_sqrt_output.txt"

// Comment this to use default divider
`define NEWTON_MODE

`define PROCESS_SINGLE_DIVIDER 2'b00
`define PROCESS_SINGLE_SQRT 2'b01
`define PROCESS_DOUBLE_DIVIDER 2'b10
```

```
`define PROCESS_DOUBLE_SQRT 2'b11
```

Listing :1-آ defines.v

آ_۱_۲ Main.v

```
//IEEE Floating Point Unit (Main module)
//Copyright (C) Hossein Entezari 2020
//2020-01-01
`include "defines.v"

module Main(
        input_as,  // single
        input_bs,  // single
        input_ad,  // double
        input_bd,  // double
        input_a_stb,
        input_b_stb,
        output_z_ack,
        process,
        clk,
        rst,
        output_zs,
        output_zd,
        output_z_stb,
        input_a_ack,
        input_b_ack);

  input clk;
  input rst;
  input[1:0] process;

  input     [31:0] input_as;
  input     [63:0] input_ad;
  input     input_a_stb;
  output    input_a_ack;

  input     [31:0] input_bs;
  input     [63:0] input_bd;
  input     input_b_stb;
  output    input_b_ack;

  output    [31:0] output_zs;
  output    [63:0] output_zd;
  output    output_z_stb;
  input     output_z_ack;

  wire input_a_stb_single_divS, input_a_stb_single_sqrtS, input_a_stb_double_divS, input_a_stb_double_sqrtS, output_zS
      ;

  wire input_a_stb_single_div, output_z_stb_single_div, input_a_ack_single_div, input_b_ack_single_div;
```

```
wire[31:0] output_z_single_div;
assign input_a_stb_single_div = input_a_stb & input_a_stb_single_divS;


`ifndef NEWTON_MODE
divider single_divider_instance(
      .input_a(input_as),
      .input_b(input_bs),
      .input_a_stb(input_a_stb_single_div),
      .input_b_stb(input_b_stb),
      .output_z_ack(output_z_ack),
      .clk(clk),
      .rst(rst),
      .output_z(output_z_single_div),
      .output_z_stb(output_z_stb_single_div),
      .input_a_ack(input_a_ack_single_div),
      .input_b_ack(input_b_ack_single_div));
`else
divider_newton single_divider_newton_instance(
      .input_a(input_as),
      .input_b(input_bs),
      .input_a_stb(input_a_stb_single_div),
      .input_b_stb(input_b_stb),
      .output_z_ack(output_z_ack),
      .clk(clk),
      .rst(rst),
      .output_z(output_z_single_div),
      .output_z_stb(output_z_stb_single_div),
      .input_a_ack(input_a_ack_single_div),
      .input_b_ack(input_b_ack_single_div));
`endif

wire input_a_stb_single_sqrt, output_z_stb_single_sqrt, input_a_ack_single_sqrt;
wire[31:0] output_z_single_sqrt;
assign input_a_stb_single_sqrt = input_a_stb & input_a_stb_single_sqrtS;
sqrt single_sqrt_instance(
      .input_a(input_as),
      .input_a_stb(input_a_stb_single_sqrt),
      .output_z_ack(output_z_ack),
      .clk(clk),
      .rst(rst),
      .output_z(output_z_single_sqrt),
      .output_z_stb(output_z_stb_single_sqrt),
      .input_a_ack(input_a_ack_single_sqrt));

wire input_a_stb_double_div, output_z_stb_double_div, input_a_ack_double_div, input_b_ack_double_div;
wire[63:0] output_z_double_div;
assign input_a_stb_double_div = input_a_stb & input_a_stb_double_divS;

`ifndef NEWTON_MODE
double_divider double_divider_instance(
      .input_a(input_ad),
      .input_b(input_bd),
```

```
98        .input_a_stb(input_a_stb_double_div),
99        .input_b_stb(input_b_stb),
100       .output_z_ack(output_z_ack),
101       .clk(clk),
102       .rst(rst),
103       .output_z(output_z_double_div),
104       .output_z_stb(output_z_stb_double_div),
105       .input_a_ack(input_a_ack_double_div),
106       .input_b_ack(input_b_ack_double_div));
107 `else
108   double_divider_newton double_divider_newton_instance(
109       .input_a(input_ad),
110       .input_b(input_bd),
111       .input_a_stb(input_a_stb_double_div),
112       .input_b_stb(input_b_stb),
113       .output_z_ack(output_z_ack),
114       .clk(clk),
115       .rst(rst),
116       .output_z(output_z_double_div),
117       .output_z_stb(output_z_stb_double_div),
118       .input_a_ack(input_a_ack_double_div),
119       .input_b_ack(input_b_ack_double_div));
120 `endif
121
122 wire input_a_stb_double_sqrt, output_z_stb_double_sqrt, input_a_ack_double_sqrt;
123 wire[63:0] output_z_double_sqrt;//////////////////////////
124 assign input_a_stb_double_sqrt = input_a_stb & input_a_stb_double_sqrtS;
125 double_sqrt double_sqrt_instance(
126       .input_a(input_ad),
127       .input_a_stb(input_a_stb_double_sqrt),
128       .output_z_ack(output_z_ack),
129       .clk(clk),
130       .rst(rst),
131       .output_z(output_z_double_sqrt),
132       .output_z_stb(output_z_stb_double_sqrt),
133       .input_a_ack(input_a_ack_double_sqrt));
134
135
136 Main_controllerUnit Controller_unit_instance(
137       .process(process),
138       .clk(clk),
139       .rst(rst),
140       .input_a_stb(input_a_stb),
141       .output_z_stb(output_z_stb),
142       .input_a_stb_single_divS(input_a_stb_single_divS),
143       .input_a_stb_single_sqrtS(input_a_stb_single_sqrtS),
144       .input_a_stb_double_divS(input_a_stb_double_divS),
145       .input_a_stb_double_sqrtS(input_a_stb_double_sqrtS),
146       .output_zS(output_zS));
147
148 assign output_zs = (output_zS == 1'b0) ? output_z_single_div : output_z_single_sqrt;
149 assign output_zd = (output_zS == 1'b0) ? output_z_double_div : output_z_double_sqrt;
```

```
150   assign input_a_ack = &({input_a_ack_single_div, input_a_ack_single_sqrt, input_a_ack_double_div,
          input_a_ack_double_sqrt});
151   assign input_b_ack = |({input_b_ack_single_div, input_b_ack_double_div});
152   assign output_z_stb = |({output_z_stb_single_div, output_z_stb_single_sqrt, output_z_stb_double_div,
          output_z_stb_double_sqrt});
153 endmodule
154
155 module Main_controllerUnit(
156         process,
157         clk,
158         rst,
159         input_a_stb,
160         output_z_stb,
161         input_a_stb_single_divS,
162         input_a_stb_single_sqrtS,
163         input_a_stb_double_divS,
164         input_a_stb_double_sqrtS,
165         output_zS);
166
167   input[1:0] process;
168   input clk;
169   input rst;
170   input input_a_stb;
171   input output_z_stb;
172
173   output reg input_a_stb_single_divS;
174   output reg input_a_stb_single_sqrtS;
175   output reg input_a_stb_double_divS;
176   output reg input_a_stb_double_sqrtS;
177   output output_zS;
178
179   reg[2:0] ps, ns;
180   reg[1:0] process_tmp;
181   parameter idle=0, fetch=1, wait_single_div=2, wait_single_sqrt=3, wait_double_div=4, wait_double_sqrt=5, wait_z_stb
          =6;
182
183   assign output_zS = process_tmp[0];
184
185   always@(posedge clk, posedge rst) begin
186         if(rst == 1'b1) process_tmp <= 2'b0;
187         else if(ps == idle) process_tmp <= process;
188         else process_tmp <= process_tmp;
189   end
190
191   always@(posedge clk, posedge rst) begin
192         if(rst == 1'b1) {ps} <= 6'b0;
193         else ps <= ns;
194   end
195
196   always@(ps) begin
197         input_a_stb_single_divS = 1'b0;
198         input_a_stb_single_sqrtS = 1'b0;
199         input_a_stb_double_divS = 1'b0;
```

```
200       input_a_stb_double_sqrtS = 1'b0;
201
202       case(ps)
203       idle: begin
204
205       end
206       fetch: begin
207
208       end
209       wait_single_div: input_a_stb_single_divS = 1'b1;
210       wait_single_sqrt: input_a_stb_single_sqrtS = 1'b1;
211       wait_double_div: input_a_stb_double_divS = 1'b1;
212       wait_double_sqrt: input_a_stb_double_sqrtS = 1'b1;
213       default: begin
214
215       end
216       endcase
217 end
218
219 always@(ps, input_a_stb, process, output_z_stb, process_tmp) begin
220       {ns} = idle;
221       case(ps)
222       idle: begin
223             if(input_a_stb == 1'b1) ns = fetch;
224             else ns = ps;
225       end
226       fetch: begin
227             if(process_tmp == `PROCESS_SINGLE_DIVIDER)      ns = wait_single_div;
228             else if(process_tmp == `PROCESS_SINGLE_SQRT) ns = wait_single_sqrt;
229             else if(process_tmp == `PROCESS_DOUBLE_DIVIDER) ns = wait_double_div;
230             else if(process_tmp == `PROCESS_DOUBLE_SQRT) ns = wait_double_sqrt;
231       end
232       wait_single_div: begin
233             if(output_z_stb == 1'b1) ns = wait_z_stb;
234             else ns = ps;
235       end
236       wait_single_sqrt: begin
237             if(output_z_stb == 1'b1) ns = wait_z_stb;
238             else ns = ps;
239       end
240       wait_double_div: begin
241             if(output_z_stb == 1'b1) ns = wait_z_stb;
242             else ns = ps;
243       end
244       wait_double_sqrt: begin
245             if(output_z_stb == 1'b1) ns = wait_z_stb;
246             else ns = ps;
247       end
248       wait_z_stb: begin
249             if(output_z_stb == 1'b0) ns = idle;
250             else ns = ps;
251       end
252       default: begin
```

```
253             ns = idle;
254         end
255         endcase
256   end
257
258 endmodule
```

Listing :2-آ Main.v

single_divider_newton.v ۳_۱_آ

```
1  //IEEE Floating Point Divider (Single Precision) using Newton-Raphson method
2  //Copyright (C) Mohsen Fayyaz 2019 "mohsenfayyaz.ir"
3  //2019-12-12
4  //
5  `include "defines.v"
6
7  module divider_newton(
8         input_a,
9         input_b,
10        input_a_stb,
11        input_b_stb,
12        output_z_ack,
13        clk,
14        rst,
15        output_z,
16        output_z_stb,
17        input_a_ack,
18        input_b_ack) ;
19
20   input     clk;
21   input     rst;
22
23   input     [31:0] input_a;
24   input     input_a_stb;
25   output    input_a_ack;
26
27   input     [31:0] input_b;
28   input     input_b_stb;
29   output    input_b_ack;
30
31   output    [31:0] output_z;
32   output    output_z_stb;
33   input     output_z_ack;
34
35   reg       s_output_z_stb;
36   reg       [31:0] s_output_z;
37   reg       s_input_a_ack;
38   reg       s_input_b_ack;
39
40   reg       [3:0] state;
```

```
41  parameter get_a         = 4'd0,
42            get_b         = 4'd1,
43            unpack        = 4'd2,
44            special_cases = 4'd3,
45            normalise_a   = 4'd4,
46            normalise_b   = 4'd5,
47            divide_0      = 4'd6,
48            divide_1      = 4'd7,
49            divide_2      = 4'd8,
50            divide_3      = 4'd9,
51            divide_4      = 4'd10,
52            divide_5      = 4'd11,
53            round         = 4'd12,
54            pack          = 4'd13,
55            put_z         = 4'd14;
56
57  reg       [31:0] a, b, z;
58  reg       [23:0] a_m, b_m, z_m;
59  reg       [9:0] a_e, b_e, z_e;
60  reg       a_s, b_s, z_s;
61  reg       guard, round_bit, sticky;
62  reg       [50:0] quotient, divisor, dividend, remainder;
63  reg       [5:0] count;
64
65  reg   [31:0] adder_a, adder_b;
66  wire   [31:0] adder_z;
67  reg adder_a_stb, adder_b_stb, adder_z_ack;
68  wire adder_a_ack, adder_b_ack, adder_z_stb;
69  adder adder_0(
70    .clk(clk),
71    .rst(rst),
72    .input_a(adder_a),
73    .input_a_stb(adder_a_stb),
74    .input_a_ack(adder_a_ack),
75    .input_b(adder_b),
76    .input_b_stb(adder_b_stb),
77    .input_b_ack(adder_b_ack),
78    .output_z(adder_z),
79    .output_z_stb(adder_z_stb),
80    .output_z_ack(adder_z_ack));
81
82  reg   [31:0] multiplier_a, multiplier_b;
83  wire   [31:0] multiplier_z;
84  reg multiplier_a_stb, multiplier_b_stb, multiplier_z_ack;
85  wire multiplier_a_ack, multiplier_b_ack, multiplier_z_stb;
86  multiplier multiplier_0(
87    .clk(clk),
88    .rst(rst),
89    .input_a(multiplier_a),
90    .input_a_stb(multiplier_a_stb),
91    .input_a_ack(multiplier_a_ack),
92    .input_b(multiplier_b),
93    .input_b_stb(multiplier_b_stb),
```

```
94    .input_b_ack(multiplier_b_ack),
95    .output_z(multiplier_z),
96    .output_z_stb(multiplier_z_stb),
97    .output_z_ack(multiplier_z_ack));
98
99  function [31:0] abs_invert_exponent_function;
100   input[31:0] in;
101   abs_invert_exponent_function = {1'b0, 254 - in[30:23] - 1, in[22:0]};  // and a minus 1 so that the approximation
        becomes less than real division to converge
102 endfunction
103
104 reg[9:0] temp_dist;
105 task normalize_exponent;
106   input [31:0] a;
107   input [31:0] b;
108   output[31:0] a_out;
109   output[31:0] b_out;
110   begin
111     //Denormalised Number
112     if ({|b[30:23]} == 0) begin
113       b_e <= -126;
114     end else begin
115       temp_dist = 126 - b[30:23];  // set the e = -1 = 126; so that 0.5 <= D=b <= 1
116       a[30:23] = a[30:23] + temp_dist;
117       b[30:23] = 126;
118     end
119
120   end
121 endtask
122
123 reg[31:0] x_n, x_n_new;
124
125 always @(posedge clk)
126 begin
127
128   case(state)
129
130     get_a:
131     begin
132       s_input_a_ack <= 1;
133       if (s_input_a_ack && input_a_stb) begin
134         a <= input_a;
135         s_input_a_ack <= 0;
136         state <= get_b;
137       end
138     end
139
140     get_b:
141     begin
142       s_input_b_ack <= 1;
143       if (s_input_b_ack && input_b_stb) begin
144         b <= input_b;
145         s_input_b_ack <= 0;
```

```
146         state <= unpack;
147       end
148     end
149
150     unpack:
151     begin
152       a_m <= a[22 : 0];
153       b_m <= b[22 : 0];
154       a_e <= a[30 : 23] - 127;
155       b_e <= b[30 : 23] - 127;
156       a_s <= a[31];
157       b_s <= b[31];
158       state <= special_cases;
159     end
160
161     special_cases:
162     begin
163       //if a is NaN or b is NaN return NaN
164       if ((a_e == 128 && a_m != 0) || (b_e == 128 && b_m != 0)) begin
165         z[31] <= 1;
166         z[30:23] <= 255;
167         z[22] <= 1;
168         z[21:0] <= 0;
169         state <= put_z;
170         //if a is inf and b is inf return NaN
171       end else if ((a_e == 128) && (b_e == 128)) begin
172         z[31] <= 1;
173         z[30:23] <= 255;
174         z[22] <= 1;
175         z[21:0] <= 0;
176         state <= put_z;
177       //if a is inf return inf
178       end else if (a_e == 128) begin
179         z[31] <= a_s ^ b_s;
180         z[30:23] <= 255;
181         z[22:0] <= 0;
182         state <= put_z;
183          //if b is zero return NaN
184         if ($signed(b_e == -127) && (b_m == 0)) begin
185           z[31] <= 1;
186           z[30:23] <= 255;
187           z[22] <= 1;
188           z[21:0] <= 0;
189           state <= put_z;
190         end
191       //if b is inf return zero
192       end else if (b_e == 128) begin
193         z[31] <= a_s ^ b_s;
194         z[30:23] <= 0;
195         z[22:0] <= 0;
196         state <= put_z;
197       //if a is zero return zero
198       end else if (($signed(a_e) == -127) && (a_m == 0)) begin
```

```
199         z[31] <= a_s ^ b_s;
200         z[30:23] <= 0;
201         z[22:0] <= 0;
202         state <= put_z;
203          //if b is zero return NaN
204         if (($signed(b_e) == -127) && (b_m == 0)) begin
205           z[31] <= 1;
206           z[30:23] <= 255;
207           z[22] <= 1;
208           z[21:0] <= 0;
209           state <= put_z;
210         end
211       //if b is zero return inf
212       end else if (($signed(b_e) == -127) && (b_m == 0)) begin
213         z[31] <= a_s ^ b_s;
214         z[30:23] <= 255;
215         z[22:0] <= 0;
216         state <= put_z;
217       end else begin
218         //Denormalised Number
219         if ($signed(a_e) == -127) begin
220           a_e <= -126;
221         end else begin
222           a_m[23] <= 1;
223         end
224         //Denormalised Number
225         if ($signed(b_e) == -127) begin
226           b_e <= -126;
227         end else begin
228           b_m[23] <= 1;
229         end
230         state <= divide_0;
231       end
232     end
233
234     divide_0:
235     begin
236       //z_s <= a_s ^ b_s;
237       //z_e <= a_e - b_e;
238       //a[31] <= a_s ^ b_s;  // To multiply in the end and correct the sign
239
240       // 0.5 <= d0 <= 1
241
242       x_n <= abs_invert_exponent_function(b);  // b with inverted power of 2 as X0
243       b <= {1'b0, b[30:0]};
244       //$display("%b -> %b", b, abs_invert_exponent_function(b));
245       count <= 0;
246
247       state <= divide_1;
248     end
249
250     divide_1:  // (D=b) * Xi
251     begin
```

```
252         multiplier_a <= b;
253         multiplier_b <= x_n;
254         multiplier_a_stb <= 1;
255         multiplier_b_stb <= 1;
256         multiplier_z_ack <= 0;
257         if(multiplier_z_stb)
258         begin
259           state <= divide_2;
260           x_n_new <= multiplier_z;
261           //$display("->x_n %b", x_n);
262           //$display("->b %b", b);
263           //$display("->x_n_new_multiply %b", multiplier_z);
264           multiplier_z_ack <= 1;
265           multiplier_a_stb <= 0;
266           multiplier_b_stb <= 0;
267         end
268       end
269
270       divide_2:  // 2 - (DXi=x_n_new)
271       begin
272         adder_a <= `FLOATING_POINT_SINGLE_2;
273         adder_b <= {1'b1, x_n_new[30:0]};
274         adder_a_stb <= 1;
275         adder_b_stb <= 1;
276         adder_z_ack <= 0;
277         if(adder_z_stb)
278         begin
279           state <= divide_3;
280           x_n_new <= adder_z;
281           //$display("->x_n_new_adder %b", adder_z);
282           adder_z_ack <= 1;
283           adder_a_stb <= 0;
284           adder_b_stb <= 0;
285         end
286       end
287
288       divide_3:  // Xi * (2-DXi = x_n_new)
289       begin
290         multiplier_a <= x_n;
291         multiplier_b <= x_n_new;
292         multiplier_a_stb <= 1;
293         multiplier_b_stb <= 1;
294         multiplier_z_ack <= 0;
295         if(multiplier_z_stb)
296         begin
297           state <= divide_4;
298           x_n_new <= multiplier_z;
299           multiplier_z_ack <= 1;
300           multiplier_a_stb <= 0;
301           multiplier_b_stb <= 0;
302
303           if(`DEBUGGING)
304             s_output_z <= x_n;  //Show results before finsihing calculation
```

```
305       end
306     end
307
308     divide_4: // Wait for multiplier to acknowledge
309     begin
310       if(!multiplier_z_stb)
311         if(count == `SINGLE_DIVIDER_MAX_LOOP || x_n == x_n_new)
312           begin
313             state <= divide_5;
314           end else begin
315             state <= divide_1;
316             count <= count + 1;
317             x_n <= x_n_new;
318         end
319     end
320
321     divide_5:  // (N=a) * ((1/D)=x_n_new)
322     begin
323       multiplier_a <= a;
324       multiplier_b <= x_n_new;
325       multiplier_a_stb <= 1;
326       multiplier_b_stb <= 1;
327       multiplier_z_ack <= 0;
328       if(multiplier_z_stb)
329       begin
330         state <= put_z;
331         z <= {a_s ^ b_s, multiplier_z[30:0]};  // Correct the sign because we used abs(b) for 1/b
332         multiplier_z_ack <= 1;
333         multiplier_a_stb <= 0;
334         multiplier_b_stb <= 0;
335       end
336     end
337
338     put_z:
339     begin
340       s_output_z_stb <= 1;
341       s_output_z <= z;
342       if (s_output_z_stb && output_z_ack) begin
343         s_output_z_stb <= 0;
344         state <= get_a;
345       end
346     end
347
348   endcase
349
350   if (rst == 1) begin
351     state <= get_a;
352     s_input_a_ack <= 0;
353     s_input_b_ack <= 0;
354     s_output_z_stb <= 0;
355   end
356
357 end
```

```
358  assign input_a_ack = s_input_a_ack;
359  assign input_b_ack = s_input_b_ack;
360  assign output_z_stb = s_output_z_stb;
361  assign output_z = s_output_z;
362
363 endmodule
```

Listing :3-آ single_divider_newton.v

آ_۱_۴ double_divider_newton.v

```
1  //IEEE Floating Point Divider (Double Precision)
2  //Copyright (C) Mohsen Fayyaz 2020 "mohsenfayyaz.ir"
3  //2020-01-10
4  //
5  `include "defines.v"
6
7  module double_divider_newton(
8          input_a,
9          input_b,
10         input_a_stb,
11         input_b_stb,
12         output_z_ack,
13         clk,
14         rst,
15         output_z,
16         output_z_stb,
17         input_a_ack,
18         input_b_ack);
19
20   input     clk;
21   input     rst;
22
23   input     [63:0] input_a;
24   input     input_a_stb;
25   output    input_a_ack;
26
27   input     [63:0] input_b;
28   input     input_b_stb;
29   output    input_b_ack;
30
31   output    [63:0] output_z;
32   output    output_z_stb;
33   input     output_z_ack;
34
35   reg       s_output_z_stb;
36   reg       [63:0] s_output_z;
37   reg       s_input_a_ack;
38   reg       s_input_b_ack;
39
40   reg       [3:0] state;
```

```
parameter get_a         = 4'd0,
          get_b         = 4'd1,
          unpack        = 4'd2,
          special_cases = 4'd3,
          normalise_a   = 4'd4,
          normalise_b   = 4'd5,
          divide_0      = 4'd6,
          divide_1      = 4'd7,
          divide_2      = 4'd8,
          divide_3      = 4'd9,
          divide_4      = 4'd10,
          divide_5      = 4'd11,
          round         = 4'd12,
          pack          = 4'd13,
          put_z         = 4'd14;

reg       [63:0] a, b, z;
reg       [52:0] a_m, b_m, z_m;
reg       [12:0] a_e, b_e, z_e;
reg       a_s, b_s, z_s;
reg       guard, round_bit, sticky;
reg       [108:0] quotient, divisor, dividend, remainder;
reg       [6:0] count;

reg   [63:0] adder_a, adder_b;
wire   [63:0] adder_z;
reg adder_a_stb, adder_b_stb, adder_z_ack;
wire adder_a_ack, adder_b_ack, adder_z_stb;
double_adder adder_0(
  .clk(clk),
  .rst(rst),
  .input_a(adder_a),
  .input_a_stb(adder_a_stb),
  .input_a_ack(adder_a_ack),
  .input_b(adder_b),
  .input_b_stb(adder_b_stb),
  .input_b_ack(adder_b_ack),
  .output_z(adder_z),
  .output_z_stb(adder_z_stb),
  .output_z_ack(adder_z_ack));

reg   [63:0] multiplier_a, multiplier_b;
wire   [63:0] multiplier_z;
reg multiplier_a_stb, multiplier_b_stb, multiplier_z_ack;
wire multiplier_a_ack, multiplier_b_ack, multiplier_z_stb;
double_multiplier multiplier_0(
  .clk(clk),
  .rst(rst),
  .input_a(multiplier_a),
  .input_a_stb(multiplier_a_stb),
  .input_a_ack(multiplier_a_ack),
  .input_b(multiplier_b),
  .input_b_stb(multiplier_b_stb),
```

```
94    .input_b_ack(multiplier_b_ack),
95    .output_z(multiplier_z),
96    .output_z_stb(multiplier_z_stb),
97    .output_z_ack(multiplier_z_ack));
98
99  function [63:0] abs_invert_exponent_function;
100   input[63:0] in;
101   abs_invert_exponent_function = {1'b0, 2046 - in[62:52] - 1, in[51:0]};  // and a minus 1 so that the approximation
        becomes less than real division to converge
102 endfunction
103
104 reg[63:0] x_n, x_n_new;
105
106 always @(posedge clk)
107 begin
108
109   case(state)
110
111     get_a:
112     begin
113       s_input_a_ack <= 1;
114       if (s_input_a_ack && input_a_stb) begin
115         a <= input_a;
116         s_input_a_ack <= 0;
117         state <= get_b;
118       end
119     end
120
121     get_b:
122     begin
123       s_input_b_ack <= 1;
124       if (s_input_b_ack && input_b_stb) begin
125         b <= input_b;
126         s_input_b_ack <= 0;
127         state <= unpack;
128       end
129     end
130
131     unpack:
132     begin
133       a_m <= a[51 : 0];
134       b_m <= b[51 : 0];
135       a_e <= a[62 : 52] - 1023;
136       b_e <= b[62 : 52] - 1023;
137       a_s <= a[63];
138       b_s <= b[63];
139       state <= special_cases;
140     end
141
142     special_cases:
143     begin
144       //if a is NaN or b is NaN return NaN
145       if ((a_e == 1024 && a_m != 0) || (b_e == 1024 && b_m != 0)) begin
```

```
146         z[63] <= 1;
147         z[62:52] <= 2047;
148         z[51] <= 1;
149         z[50:0] <= 0;
150         state <= put_z;
151         //if a is inf and b is inf return NaN
152       end else if ((a_e == 1024) && (b_e == 1024)) begin
153         z[63] <= 1;
154         z[62:52] <= 2047;
155         z[51] <= 1;
156         z[50:0] <= 0;
157         state <= put_z;
158       //if a is inf return inf
159       end else if (a_e == 1024) begin
160         z[63] <= a_s ^ b_s;
161         z[62:52] <= 2047;
162         z[51:0] <= 0;
163         state <= put_z;
164          //if b is zero return NaN
165         if ($signed(b_e == -1023) && (b_m == 0)) begin
166           z[63] <= 1;
167           z[62:52] <= 2047;
168           z[51] <= 1;
169           z[50:0] <= 0;
170           state <= put_z;
171         end
172       //if b is inf return zero
173       end else if (b_e == 1024) begin
174         z[63] <= a_s ^ b_s;
175         z[62:52] <= 0;
176         z[51:0] <= 0;
177         state <= put_z;
178       //if a is zero return zero
179       end else if (($signed(a_e) == -1023) && (a_m == 0)) begin
180         z[63] <= a_s ^ b_s;
181         z[62:52] <= 0;
182         z[51:0] <= 0;
183         state <= put_z;
184          //if b is zero return NaN
185         if (($signed(b_e) == -1023) && (b_m == 0)) begin
186           z[63] <= 1;
187           z[62:52] <= 2047;
188           z[51] <= 1;
189           z[50:0] <= 0;
190           state <= put_z;
191         end
192       //if b is zero return inf
193       end else if (($signed(b_e) == -1023) && (b_m == 0)) begin
194         z[63] <= a_s ^ b_s;
195         z[62:52] <= 2047;
196         z[51:0] <= 0;
197         state <= put_z;
198       end else begin
```

```
199         //Denormalised Number
200         if ($signed(a_e) == -1023) begin
201           a_e <= -1022;
202         end else begin
203           a_m[52] <= 1;
204         end
205         //Denormalised Number
206         if ($signed(b_e) == -1023) begin
207           b_e <= -1022;
208         end else begin
209           b_m[52] <= 1;
210         end
211         state <= divide_0;
212       end
213     end
214
215     divide_0:
216     begin
217       z_s <= a_s ^ b_s;
218       z_e <= a_e - b_e;
219
220       x_n <= abs_invert_exponent_function(b);  // b with inverted power of 2 as X0
221       b <= {1'b0, b[62:0]};  // abs of b
222       count <= 0;
223
224       state <= divide_1;
225     end
226
227     divide_1:  // (D=b) * Xi
228     begin
229       multiplier_a <= b;
230       multiplier_b <= x_n;
231       multiplier_a_stb <= 1;
232       multiplier_b_stb <= 1;
233       multiplier_z_ack <= 0;
234       if(multiplier_z_stb)
235       begin
236         state <= divide_2;
237         x_n_new <= multiplier_z;
238         //$display("->x_n %b", x_n);
239         //$display("->b %b", b);
240         //$display("->x_n_new_multiply %b", multiplier_z);
241         multiplier_z_ack <= 1;
242         multiplier_a_stb <= 0;
243         multiplier_b_stb <= 0;
244       end
245     end
246
247     divide_2:  // 2 - (DXi=x_n_new)
248     begin
249       adder_a <= `FLOATING_POINT_DOUBLE_2;
250       adder_b <= {1'b1, x_n_new[62:0]};  // minus
251       adder_a_stb <= 1;
```

```
252         adder_b_stb <= 1;
253         adder_z_ack <= 0;
254         if(adder_z_stb)
255         begin
256           state <= divide_3;
257           x_n_new <= adder_z;
258           //$display("->x_n_new_adder %b", adder_z);
259           adder_z_ack <= 1;
260           adder_a_stb <= 0;
261           adder_b_stb <= 0;
262         end
263       end
264
265       divide_3:  // Xi * (2-DXi = x_n_new)
266       begin
267         multiplier_a <= x_n;
268         multiplier_b <= x_n_new;
269         multiplier_a_stb <= 1;
270         multiplier_b_stb <= 1;
271         multiplier_z_ack <= 0;
272         if(multiplier_z_stb)
273         begin
274           state <= divide_4;
275           x_n_new <= multiplier_z;
276           multiplier_z_ack <= 1;
277           multiplier_a_stb <= 0;
278           multiplier_b_stb <= 0;
279
280           if(`DEBUGGING)
281             s_output_z <= x_n;  //Show results before finsihing calculation
282         end
283       end
284
285       divide_4: // Wait for multiplier to acknowledge
286       begin
287         if(!multiplier_z_stb)
288           if(count == `DOUBLE_DIVIDER_MAX_LOOP || x_n == x_n_new)
289             begin
290               state <= divide_5;
291             end else begin
292               state <= divide_1;
293               count <= count + 1;
294               x_n <= x_n_new;
295           end
296       end
297
298       divide_5:  // (N=a) * ((1/D)=x_n_new)
299       begin
300         multiplier_a <= a;
301         multiplier_b <= x_n_new;
302         multiplier_a_stb <= 1;
303         multiplier_b_stb <= 1;
304         multiplier_z_ack <= 0;
```

```
305         if(multiplier_z_stb)
306         begin
307           state <= put_z;
308           z <= {a_s ^ b_s, multiplier_z[62:0]};  // Correct the sign because we used abs(b) for 1/b
309           multiplier_z_ack <= 1;
310           multiplier_a_stb <= 0;
311           multiplier_b_stb <= 0;
312         end
313       end
314
315       put_z:
316       begin
317         s_output_z_stb <= 1;
318         s_output_z <= z;
319         if (s_output_z_stb && output_z_ack) begin
320           s_output_z_stb <= 0;
321           state <= get_a;
322         end
323       end
324
325     endcase
326
327     if (rst == 1) begin
328       state <= get_a;
329       s_input_a_ack <= 0;
330       s_input_b_ack <= 0;
331       s_output_z_stb <= 0;
332     end
333
334   end
335   assign input_a_ack = s_input_a_ack;
336   assign input_b_ack = s_input_b_ack;
337   assign output_z_stb = s_output_z_stb;
338   assign output_z = s_output_z;
339
340 endmodule
```

Listing :4-آ double_divider_newton.v

آ ـ ۱ ـ ۵ single_sqrt.v

```
1  //IEEE Floating Point Square Root (Single Precision)
2  //Copyright (C) Mohsen Fayyaz 2019 "mohsenfayyaz.ir"
3  //2019-12-12
4  //
5  `include "defines.v"
6
7  module sqrt(
8         input_a,
9         input_a_stb,
10        output_z_ack,
```

```
      clk,
      rst,
      output_z,
      output_z_stb,
      input_a_ack);

input     clk;
input     rst;

input     [31:0] input_a;
input     input_a_stb;
output    input_a_ack;

output    [31:0] output_z;
output    output_z_stb;
input     output_z_ack;

reg       s_output_z_stb;
reg       [31:0] s_output_z;
reg       s_input_a_ack;

reg       [3:0] state;
parameter get_a         = 4'd0,
          //get_b         = 4'd1,
          unpack        = 4'd2,
          special_cases = 4'd3,
          normalise_a   = 4'd4,
          //normalise_b   = 4'd5,
          sqrt_0      = 4'd6,
          sqrt_1      = 4'd7,
          sqrt_2      = 4'd8,
          sqrt_3      = 4'd9,
          normalise_1   = 4'd10,
          normalise_2   = 4'd11,
          round         = 4'd12,
          pack          = 4'd13,
          put_z         = 4'd14;

reg       [31:0] a, z;
reg       [23:0] a_m, z_m;
reg       [9:0] a_e, z_e;
reg       a_s, z_s;
reg       guard, round_bit, sticky;

reg       [50:0] quotient, divisor, dividend, remainder;
reg       [7:0] count;

reg   [31:0] divider_a, divider_b;
wire  [31:0] divider_z;
reg divider_a_stb, divider_b_stb, divider_z_ack;
wire divider_a_ack, divider_b_ack, divider_z_stb;
divider divider_0(
  .clk(clk),
```

```
64    .rst(rst),
65    .input_a(divider_a),
66    .input_a_stb(divider_a_stb),
67    .input_a_ack(divider_a_ack),
68    .input_b(divider_b),
69    .input_b_stb(divider_b_stb),
70    .input_b_ack(divider_b_ack),
71    .output_z(divider_z),
72    .output_z_stb(divider_z_stb),
73    .output_z_ack(divider_z_ack));
74
75  reg   [31:0] adder_a, adder_b;
76  wire   [31:0] adder_z;
77  reg adder_a_stb, adder_b_stb, adder_z_ack;
78  wire adder_a_ack, adder_b_ack, adder_z_stb;
79  adder adder_0(
80    .clk(clk),
81    .rst(rst),
82    .input_a(adder_a),
83    .input_a_stb(adder_a_stb),
84    .input_a_ack(adder_a_ack),
85    .input_b(adder_b),
86    .input_b_stb(adder_b_stb),
87    .input_b_ack(adder_b_ack),
88    .output_z(adder_z),
89    .output_z_stb(adder_z_stb),
90    .output_z_ack(adder_z_ack));
91
92  function [31:0] divide_exponent_function;
93    input[31:0] in;
94    divide_exponent_function = {in[31], ((in[30:23]-127) >> 1) + 127, in[22:0]};
95  endfunction
96
97  reg[31:0] sqrt_n, sqrt_n_new;
98  always @(posedge clk)
99  begin
100
101   case(state)
102
103     get_a:
104     begin
105       s_input_a_ack <= 1;
106       if (s_input_a_ack && input_a_stb) begin
107         a <= input_a;
108         s_input_a_ack <= 0;
109         state <= unpack;
110       end
111     end
112
113     unpack:
114     begin
115       a_m <= a[22 : 0];
116       a_e <= a[30 : 23] - 127;
```

```
117       a_s <= a[31];
118       state <= special_cases;
119     end
120
121     special_cases:
122     begin
123       //if a is less than 0 return NaN
124       if (a_s == 1) begin
125         z[31] <= 1;
126         z[30:23] <= 255;
127         z[22] <= 1;
128         z[21:0] <= 0;
129         state <= put_z;
130       //if a is NaN or b is NaN return NaN
131       end else if (a_e == 128 && a_m != 0) begin
132         z[31] <= 1;
133         z[30:23] <= 255;
134         z[22] <= 1;
135         z[21:0] <= 0;
136         state <= put_z;
137       //if a is inf return inf
138       end else if (a_e == 128) begin
139         z[31] <= a_s;
140         z[30:23] <= 255;
141         z[22:0] <= 0;
142         state <= put_z;
143       //if a is zero return zero
144       end else if (($signed(a_e) == -127) && (a_m == 0)) begin
145         z[31] <= a_s;
146         z[30:23] <= 0;
147         z[22:0] <= 0;
148         state <= put_z;
149       end else begin
150         //Denormalised Number
151         if ($signed(a_e) == -127) begin
152           a_e <= -126;
153         end else begin
154           a_m[23] <= 1;
155         end
156         state <= normalise_a;
157       end
158     end
159
160     normalise_a:
161     begin
162       if (a_m[23]) begin
163         state <= sqrt_0;
164       end else begin
165         a_m <= a_m << 1;
166         a_e <= a_e - 1;
167       end
168     end
169
```

```
170     sqrt_0:
171     begin
172       z_s <= a_s;
173       z_e <= a_e >> 1; // a_e/2
174       sqrt_n <= divide_exponent_function(a);  // a with half power of 2 as X0
175       //sqrt_n <= a;
176       count <= 0;
177
178       state <= sqrt_1;
179     end
180
181     sqrt_1:  // a/x_n
182     begin
183       divider_a <= a;
184       divider_b <= sqrt_n;
185       divider_a_stb <= 1;
186       divider_b_stb <= 1;
187       divider_z_ack <= 0;
188       if(divider_z_stb)
189       begin
190         state <= sqrt_2;
191         sqrt_n_new <= divider_z;
192         divider_z_ack <= 1;
193         divider_a_stb <= 0;
194         divider_b_stb <= 0;
195       end
196     end
197
198     sqrt_2:
199     begin
200       adder_a <= sqrt_n_new;
201       adder_b <= sqrt_n;
202       adder_a_stb <= 1;
203       adder_b_stb <= 1;
204       adder_z_ack <= 0;
205       if(adder_z_stb)
206       begin
207         adder_a_stb <= 0;
208         adder_b_stb <= 0;
209         adder_z_ack <= 1;
210         if(`DEBUGGING)
211           s_output_z <= sqrt_n;  //Show results before finsihing calculation
212
213         if(count == `SINGLE_SQRT_MAX_LOOP || sqrt_n == {adder_z[31], adder_z[30:23] - 1, adder_z[22:0]})
214         begin
215           z <= {adder_z[31], adder_z[30:23] - 1, adder_z[22:0]};
216           state <= put_z;
217         end else begin
218           state <= sqrt_1;
219           count <= count + 1;
220           sqrt_n <= {adder_z[31], adder_z[30:23] - 1, adder_z[22:0]};  // 1/2(x+a/x)
221         end
222       end
```

```
223       end
224
225       put_z:
226       begin
227         s_output_z_stb <= 1;
228         s_output_z <= z;
229         if (s_output_z_stb && output_z_ack) begin
230           s_output_z_stb <= 0;
231           state <= get_a;
232         end
233       end
234
235     endcase
236
237     if (rst == 1) begin
238       state <= get_a;
239       s_input_a_ack <= 0;
240       s_output_z_stb <= 0;
241     end
242
243   end
244   assign input_a_ack = s_input_a_ack;
245   assign output_z_stb = s_output_z_stb;
246   assign output_z = s_output_z;
247
248 endmodule
```

Listing :5-آ single_sqrt.v

آ ـ ۱ ـ ۶ double_sqrt.v

```
1  //IEEE Floating Point Square Root (Double Precision)
2  //Copyright (C) Mohsen Fayyaz 2020 "mohsenfayyaz.ir"
3  //2020-01-10
4  //
5  `include "defines.v"
6
7  module double_sqrt(
8         input_a,
9         input_a_stb,
10        output_z_ack,
11        clk,
12        rst,
13        output_z,
14        output_z_stb,
15        input_a_ack);
16
17   input     clk;
18   input     rst;
19
20   input     [63:0] input_a;
```

```
21  input     input_a_stb;
22  output    input_a_ack;
23
24  output    [63:0] output_z;
25  output    output_z_stb;
26  input     output_z_ack;
27
28  reg       s_output_z_stb;
29  reg       [63:0] s_output_z;
30  reg       s_input_a_ack;
31
32  reg       [3:0] state;
33  parameter get_a         = 4'd0,
34            //get_b         = 4'd1,
35            unpack        = 4'd2,
36            special_cases = 4'd3,
37            normalise_a   = 4'd4,
38            //normalise_b   = 4'd5,
39            sqrt_0      = 4'd6,
40            sqrt_1      = 4'd7,
41            sqrt_2      = 4'd8,
42            sqrt_3      = 4'd9,
43            normalise_1   = 4'd10,
44            normalise_2   = 4'd11,
45            round         = 4'd12,
46            pack          = 4'd13,
47            put_z         = 4'd14;
48
49  reg       [63:0] a, b, z;
50  reg       [52:0] a_m, b_m, z_m;
51  reg       [12:0] a_e, b_e, z_e;
52  reg       a_s, b_s, z_s;
53  reg       [6:0] count;
54
55  reg   [63:0] divider_a, divider_b;
56  wire  [63:0] divider_z;
57  reg divider_a_stb, divider_b_stb, divider_z_ack;
58  wire divider_a_ack, divider_b_ack, divider_z_stb;
59  double_divider divider_0(
60    .clk(clk),
61    .rst(rst),
62    .input_a(divider_a),
63    .input_a_stb(divider_a_stb),
64    .input_a_ack(divider_a_ack),
65    .input_b(divider_b),
66    .input_b_stb(divider_b_stb),
67    .input_b_ack(divider_b_ack),
68    .output_z(divider_z),
69    .output_z_stb(divider_z_stb),
70    .output_z_ack(divider_z_ack));
71
72  reg   [63:0] adder_a, adder_b;
73  wire   [63:0] adder_z;
```

```
74  reg adder_a_stb, adder_b_stb, adder_z_ack;
75  wire adder_a_ack, adder_b_ack, adder_z_stb;
76  double_adder adder_0(
77    .clk(clk),
78    .rst(rst),
79    .input_a(adder_a),
80    .input_a_stb(adder_a_stb),
81    .input_a_ack(adder_a_ack),
82    .input_b(adder_b),
83    .input_b_stb(adder_b_stb),
84    .input_b_ack(adder_b_ack),
85    .output_z(adder_z),
86    .output_z_stb(adder_z_stb),
87    .output_z_ack(adder_z_ack));
88
89  function [63:0] divide_exponent_function;
90    input[63:0] in;
91    divide_exponent_function = {in[63], ((in[62:52]-1023) >> 1) + 1023, in[51:0]};
92  endfunction
93
94  reg[63:0] sqrt_n, sqrt_n_new;
95  always @(posedge clk)
96  begin
97
98    case(state)
99
100     get_a:
101     begin
102       s_input_a_ack <= 1;
103       if (s_input_a_ack && input_a_stb) begin
104         a <= input_a;
105         s_input_a_ack <= 0;
106         state <= unpack;
107       end
108     end
109
110     unpack:
111     begin
112       a_m <= a[51 : 0];
113       a_e <= a[62 : 52] - 1023;
114       a_s <= a[63];
115       state <= special_cases;
116     end
117
118     special_cases:
119     begin
120       //if a is less than 0 return NaN
121       if (a_s == 1) begin
122         z[63] <= 1;
123         z[62:52] <= 2047;
124         z[51] <= 1;
125         z[50:0] <= 0;
126         state <= put_z;
```

```
127         //if a is NaN or b is NaN return NaN
128         end else if (a_e == 1024 && a_m != 0) begin
129           z[63] <= 1;
130           z[62:52] <= 2047;
131           z[51] <= 1;
132           z[50:0] <= 0;
133           state <= put_z;
134         //if a is inf return inf
135         end else if (a_e == 1024) begin
136           z[63] <= 0;
137           z[62:52] <= 2047;
138           z[51:0] <= 0;
139           state <= put_z;
140         //if a is zero return zero
141         end else if (($signed(a_e) == -1023) && (a_m == 0)) begin
142           z[63] <= 0;
143           z[62:52] <= 0;
144           z[51:0] <= 0;
145           state <= put_z;
146         end else begin
147           //Denormalised Number
148           if ($signed(a_e) == -1023) begin
149             a_e <= -1022;
150           end else begin
151             a_m[52] <= 1;
152           end
153           state <= normalise_a;
154         end
155       end
156
157       normalise_a:
158       begin
159         if (a_m[52]) begin
160           state <= sqrt_0;
161         end else begin
162           a_m <= a_m << 1;
163           a_e <= a_e - 1;
164         end
165       end
166
167       sqrt_0:
168       begin
169         z_s <= a_s;
170         z_e <= a_e >> 1; // a_e/2
171         sqrt_n <= divide_exponent_function(a);  // a with half power of 2 as X0
172         //sqrt_n <= a;
173         count <= 0;
174
175         state <= sqrt_1;
176       end
177
178       sqrt_1:  // a/x_n
179       begin
```

```
180         divider_a <= a;
181         divider_b <= sqrt_n;
182         divider_a_stb <= 1;
183         divider_b_stb <= 1;
184         divider_z_ack <= 0;
185         if(divider_z_stb)
186         begin
187           //$display("%b/%b = %b", a, sqrt_n, divider_z);
188           state <= sqrt_2;
189           sqrt_n_new <= divider_z;
190           divider_z_ack <= 1;
191           divider_a_stb <= 0;
192           divider_b_stb <= 0;
193         end
194       end
195
196       sqrt_2:  // 1/2(x+a/x)
197       begin
198         adder_a <= sqrt_n_new;
199         adder_b <= sqrt_n;
200         adder_a_stb <= 1;
201         adder_b_stb <= 1;
202         adder_z_ack <= 0;
203         if(adder_z_stb)
204         begin
205           //$display("%b+%b = %b", sqrt_n_new, sqrt_n, {adder_z[63], adder_z[62:52] - 1, adder_z[51:0]});
206           //$display("%b+%b/2 = %b", sqrt_n_new, sqrt_n, adder_z);
207           adder_a_stb <= 0;
208           adder_b_stb <= 0;
209           adder_z_ack <= 1;
210           if(`DEBUGGING)
211             s_output_z <= sqrt_n;  //Show results before finsihing calculation
212
213           if(count == `DOUBLE_SQRT_MAX_LOOP || sqrt_n == {adder_z[63], adder_z[62:52] - 1, adder_z[51:0]})  // -1 will
      divide by 2 // 1/2(x+a/x)
214           begin
215             z <= {adder_z[63], adder_z[62:52] - 1, adder_z[51:0]};
216             state <= put_z;
217           end else begin
218             state <= sqrt_1;
219             count <= count + 1;
220             sqrt_n <= {adder_z[63], adder_z[62:52] - 1, adder_z[51:0]};  // 1/2(x+a/x)
221           end
222         end
223       end
224
225       put_z:
226       begin
227         s_output_z_stb <= 1;
228         s_output_z <= z;
229         if (s_output_z_stb && output_z_ack) begin
230           s_output_z_stb <= 0;
231           state <= get_a;
```

```
232         end
233       end
234
235     endcase
236
237     if (rst == 1) begin
238       state <= get_a;
239       s_input_a_ack <= 0;
240       s_output_z_stb <= 0;
241     end
242
243   end
244   assign input_a_ack = s_input_a_ack;
245   assign output_z_stb = s_output_z_stb;
246   assign output_z = s_output_z;
247
248 endmodule
```

Listing آ-6: double_sqrt.v

آ ـ ۱ ـ ۷ single_divider.v (using shift and subtract method)

```
1  //IEEE Floating Point Divider (Single Precision)
2  //Copyright (C) Jonathan P Dawson 2013
3  //2013-12-12
4  //
5  module divider(
6          input_a,
7          input_b,
8          input_a_stb,
9          input_b_stb,
10         output_z_ack,
11         clk,
12         rst,
13         output_z,
14         output_z_stb,
15         input_a_ack,
16         input_b_ack);
17
18   input     clk;
19   input     rst;
20
21   input     [31:0] input_a;
22   input     input_a_stb;
23   output    input_a_ack;
24
25   input     [31:0] input_b;
26   input     input_b_stb;
27   output    input_b_ack;
28
29   output    [31:0] output_z;
```

```
30  output     output_z_stb;
31  input      output_z_ack;
32
33  reg        s_output_z_stb;
34  reg        [31:0] s_output_z;
35  reg        s_input_a_ack;
36  reg        s_input_b_ack;
37
38  reg        [3:0] state;
39  parameter  get_a          = 4'd0,
40             get_b          = 4'd1,
41             unpack         = 4'd2,
42             special_cases  = 4'd3,
43             normalise_a    = 4'd4,
44             normalise_b    = 4'd5,
45             divide_0       = 4'd6,
46             divide_1       = 4'd7,
47             divide_2       = 4'd8,
48             divide_3       = 4'd9,
49             normalise_1    = 4'd10,
50             normalise_2    = 4'd11,
51             round          = 4'd12,
52             pack           = 4'd13,
53             put_z          = 4'd14;
54
55  reg        [31:0] a, b, z;
56  reg        [23:0] a_m, b_m, z_m;
57  reg        [9:0] a_e, b_e, z_e;
58  reg        a_s, b_s, z_s;
59  reg        guard, round_bit, sticky;
60  reg        [50:0] quotient, divisor, dividend, remainder;
61  reg        [5:0] count;
62
63  always @(posedge clk)
64  begin
65
66    case(state)
67
68      get_a:
69      begin
70        s_input_a_ack <= 1;
71        if (s_input_a_ack && input_a_stb) begin
72          a <= input_a;
73          s_input_a_ack <= 0;
74          state <= get_b;
75        end
76      end
77
78      get_b:
79      begin
80        s_input_b_ack <= 1;
81        if (s_input_b_ack && input_b_stb) begin
82          b <= input_b;
```

```
83          s_input_b_ack <= 0;
84          state <= unpack;
85        end
86      end
87
88      unpack:
89      begin
90        a_m <= a[22 : 0];
91        b_m <= b[22 : 0];
92        a_e <= a[30 : 23] - 127;
93        b_e <= b[30 : 23] - 127;
94        a_s <= a[31];
95        b_s <= b[31];
96        state <= special_cases;
97      end
98
99      special_cases:
100     begin
101       //if a is NaN or b is NaN return NaN
102       if ((a_e == 128 && a_m != 0) || (b_e == 128 && b_m != 0)) begin
103         z[31] <= 1;
104         z[30:23] <= 255;
105         z[22] <= 1;
106         z[21:0] <= 0;
107         state <= put_z;
108         //if a is inf and b is inf return NaN
109       end else if ((a_e == 128) && (b_e == 128)) begin
110         z[31] <= 1;
111         z[30:23] <= 255;
112         z[22] <= 1;
113         z[21:0] <= 0;
114         state <= put_z;
115       //if a is inf return inf
116       end else if (a_e == 128) begin
117         z[31] <= a_s ^ b_s;
118         z[30:23] <= 255;
119         z[22:0] <= 0;
120         state <= put_z;
121          //if b is zero return NaN
122         if ($signed(b_e == -127) && (b_m == 0)) begin
123           z[31] <= 1;
124           z[30:23] <= 255;
125           z[22] <= 1;
126           z[21:0] <= 0;
127           state <= put_z;
128         end
129       //if b is inf return zero
130       end else if (b_e == 128) begin
131         z[31] <= a_s ^ b_s;
132         z[30:23] <= 0;
133         z[22:0] <= 0;
134         state <= put_z;
135       //if a is zero return zero
```

```
136       end else if (($signed(a_e) == -127) && (a_m == 0)) begin
137         z[31] <= a_s ^ b_s;
138         z[30:23] <= 0;
139         z[22:0] <= 0;
140         state <= put_z;
141          //if b is zero return NaN
142         if (($signed(b_e) == -127) && (b_m == 0)) begin
143           z[31] <= 1;
144           z[30:23] <= 255;
145           z[22] <= 1;
146           z[21:0] <= 0;
147           state <= put_z;
148         end
149       //if b is zero return inf
150       end else if (($signed(b_e) == -127) && (b_m == 0)) begin
151         z[31] <= a_s ^ b_s;
152         z[30:23] <= 255;
153         z[22:0] <= 0;
154         state <= put_z;
155       end else begin
156         //Denormalised Number
157         if ($signed(a_e) == -127) begin
158           a_e <= -126;
159         end else begin
160           a_m[23] <= 1;
161         end
162         //Denormalised Number
163         if ($signed(b_e) == -127) begin
164           b_e <= -126;
165         end else begin
166           b_m[23] <= 1;
167         end
168         state <= normalise_a;
169       end
170     end
171
172     normalise_a:
173     begin
174       if (a_m[23]) begin
175         state <= normalise_b;
176       end else begin
177         a_m <= a_m << 1;
178         a_e <= a_e - 1;
179       end
180     end
181
182     normalise_b:
183     begin
184       if (b_m[23]) begin
185         state <= divide_0;
186       end else begin
187         b_m <= b_m << 1;
188         b_e <= b_e - 1;
```

```
189       end
190     end
191
192     divide_0:
193     begin
194       z_s <= a_s ^ b_s;
195       z_e <= a_e - b_e;
196       quotient <= 0;
197       remainder <= 0;
198       count <= 0;
199       dividend <= a_m << 27;
200       divisor <= b_m;
201       state <= divide_1;
202     end
203
204     divide_1:
205     begin
206       quotient <= quotient << 1;
207       remainder <= remainder << 1;
208       remainder[0] <= dividend[50];
209       dividend <= dividend << 1;
210       state <= divide_2;
211     end
212
213     divide_2:
214     begin
215       if (remainder >= divisor) begin
216         quotient[0] <= 1;
217         remainder <= remainder - divisor;
218       end
219       if (count == 49) begin
220         state <= divide_3;
221       end else begin
222         count <= count + 1;
223         state <= divide_1;
224       end
225     end
226
227     divide_3:
228     begin
229       z_m <= quotient[26:3];
230       guard <= quotient[2];
231       round_bit <= quotient[1];
232       sticky <= quotient[0] | (remainder != 0);
233       state <= normalise_1;
234     end
235
236     normalise_1:
237     begin
238       if (z_m[23] == 0 && $signed(z_e) > -126) begin
239         z_e <= z_e - 1;
240         z_m <= z_m << 1;
241         z_m[0] <= guard;
```

```
242         guard <= round_bit;
243         round_bit <= 0;
244       end else begin
245         state <= normalise_2;
246       end
247     end
248
249     normalise_2:
250     begin
251       if ($signed(z_e) < -126) begin
252         z_e <= z_e + 1;
253         z_m <= z_m >> 1;
254         guard <= z_m[0];
255         round_bit <= guard;
256         sticky <= sticky | round_bit;
257       end else begin
258         state <= round;
259       end
260     end
261
262     round:
263     begin
264       if (guard && (round_bit | sticky | z_m[0])) begin
265         z_m <= z_m + 1;
266         if (z_m == 24'hffffff) begin
267           z_e <=z_e + 1;
268         end
269       end
270       state <= pack;
271     end
272
273     pack:
274     begin
275       z[22 : 0] <= z_m[22:0];
276       z[30 : 23] <= z_e[7:0] + 127;
277       z[31] <= z_s;
278       if ($signed(z_e) == -126 && z_m[23] == 0) begin
279         z[30 : 23] <= 0;
280       end
281       //if overflow occurs, return inf
282       if ($signed(z_e) > 127) begin
283         z[22 : 0] <= 0;
284         z[30 : 23] <= 255;
285         z[31] <= z_s;
286       end
287       state <= put_z;
288     end
289
290     put_z:
291     begin
292       s_output_z_stb <= 1;
293       s_output_z <= z;
294       if (s_output_z_stb && output_z_ack) begin
```

```
295           s_output_z_stb <= 0;
296           state <= get_a;
297         end
298       end
299
300     endcase
301
302     if (rst == 1) begin
303       state <= get_a;
304       s_input_a_ack <= 0;
305       s_input_b_ack <= 0;
306       s_output_z_stb <= 0;
307     end
308
309   end
310   assign input_a_ack = s_input_a_ack;
311   assign input_b_ack = s_input_b_ack;
312   assign output_z_stb = s_output_z_stb;
313   assign output_z = s_output_z;
314
315 endmodule
```

Listing :7-آ single_divider.v

آ ـ ۱ ـ ۸ double_divider.v (using shift and subtract)

```
1  //IEEE Floating Point Divider (Double Precision)
2  //Copyright (C) Jonathan P Dawson 2014
3  //2014-01-11
4  //
5  module double_divider(
6          input_a,
7          input_b,
8          input_a_stb,
9          input_b_stb,
10         output_z_ack,
11         clk,
12         rst,
13         output_z,
14         output_z_stb,
15         input_a_ack,
16         input_b_ack);
17
18   input      clk;
19   input      rst;
20
21   input      [63:0] input_a;
22   input      input_a_stb;
23   output     input_a_ack;
24
25   input      [63:0] input_b;
```

```
26  input     input_b_stb;
27  output    input_b_ack;
28
29  output    [63:0] output_z;
30  output    output_z_stb;
31  input     output_z_ack;
32
33  reg       s_output_z_stb;
34  reg       [63:0] s_output_z;
35  reg       s_input_a_ack;
36  reg       s_input_b_ack;
37
38  reg       [3:0] state;
39  parameter get_a         = 4'd0,
40            get_b         = 4'd1,
41            unpack        = 4'd2,
42            special_cases = 4'd3,
43            normalise_a   = 4'd4,
44            normalise_b   = 4'd5,
45            divide_0      = 4'd6,
46            divide_1      = 4'd7,
47            divide_2      = 4'd8,
48            divide_3      = 4'd9,
49            normalise_1   = 4'd10,
50            normalise_2   = 4'd11,
51            round         = 4'd12,
52            pack          = 4'd13,
53            put_z         = 4'd14;
54
55  reg       [63:0] a, b, z;
56  reg       [52:0] a_m, b_m, z_m;
57  reg       [12:0] a_e, b_e, z_e;
58  reg       a_s, b_s, z_s;
59  reg       guard, round_bit, sticky;
60  reg       [108:0] quotient, divisor, dividend, remainder;
61  reg       [6:0] count;
62
63  always @(posedge clk)
64  begin
65
66    case(state)
67
68      get_a:
69      begin
70        s_input_a_ack <= 1;
71        if (s_input_a_ack && input_a_stb) begin
72          a <= input_a;
73          s_input_a_ack <= 0;
74          state <= get_b;
75        end
76      end
77
78      get_b:
```

```
79      begin
80        s_input_b_ack <= 1;
81        if (s_input_b_ack && input_b_stb) begin
82          b <= input_b;
83          s_input_b_ack <= 0;
84          state <= unpack;
85        end
86      end
87
88      unpack:
89      begin
90        a_m <= a[51 : 0];
91        b_m <= b[51 : 0];
92        a_e <= a[62 : 52] - 1023;
93        b_e <= b[62 : 52] - 1023;
94        a_s <= a[63];
95        b_s <= b[63];
96        state <= special_cases;
97      end
98
99      special_cases:
100     begin
101       //if a is NaN or b is NaN return NaN
102       if ((a_e == 1024 && a_m != 0) || (b_e == 1024 && b_m != 0)) begin
103         z[63] <= 1;
104         z[62:52] <= 2047;
105         z[51] <= 1;
106         z[50:0] <= 0;
107         state <= put_z;
108         //if a is inf and b is inf return NaN
109       end else if ((a_e == 1024) && (b_e == 1024)) begin
110         z[63] <= 1;
111         z[62:52] <= 2047;
112         z[51] <= 1;
113         z[50:0] <= 0;
114         state <= put_z;
115       //if a is inf return inf
116       end else if (a_e == 1024) begin
117         z[63] <= a_s ^ b_s;
118         z[62:52] <= 2047;
119         z[51:0] <= 0;
120         state <= put_z;
121          //if b is zero return NaN
122         if ($signed(b_e == -1023) && (b_m == 0)) begin
123           z[63] <= 1;
124           z[62:52] <= 2047;
125           z[51] <= 1;
126           z[50:0] <= 0;
127           state <= put_z;
128         end
129       //if b is inf return zero
130       end else if (b_e == 1024) begin
131         z[63] <= a_s ^ b_s;
```

```
132         z[62:52] <= 0;
133         z[51:0] <= 0;
134         state <= put_z;
135       //if a is zero return zero
136       end else if (($signed(a_e) == -1023) && (a_m == 0)) begin
137         z[63] <= a_s ^ b_s;
138         z[62:52] <= 0;
139         z[51:0] <= 0;
140         state <= put_z;
141          //if b is zero return NaN
142         if (($signed(b_e) == -1023) && (b_m == 0)) begin
143           z[63] <= 1;
144           z[62:52] <= 2047;
145           z[51] <= 1;
146           z[50:0] <= 0;
147           state <= put_z;
148         end
149       //if b is zero return inf
150       end else if (($signed(b_e) == -1023) && (b_m == 0)) begin
151         z[63] <= a_s ^ b_s;
152         z[62:52] <= 2047;
153         z[51:0] <= 0;
154         state <= put_z;
155       end else begin
156         //Denormalised Number
157         if ($signed(a_e) == -1023) begin
158           a_e <= -1022;
159         end else begin
160           a_m[52] <= 1;
161         end
162         //Denormalised Number
163         if ($signed(b_e) == -1023) begin
164           b_e <= -1022;
165         end else begin
166           b_m[52] <= 1;
167         end
168         state <= normalise_a;
169       end
170     end
171
172     normalise_a:
173     begin
174       if (a_m[52]) begin
175         state <= normalise_b;
176       end else begin
177         a_m <= a_m << 1;
178         a_e <= a_e - 1;
179       end
180     end
181
182     normalise_b:
183     begin
184       if (b_m[52]) begin
```

```
185         state <= divide_0;
186       end else begin
187         b_m <= b_m << 1;
188         b_e <= b_e - 1;
189       end
190     end
191
192     divide_0:
193     begin
194       z_s <= a_s ^ b_s;
195       z_e <= a_e - b_e;
196       quotient <= 0;
197       remainder <= 0;
198       count <= 0;
199       dividend <= a_m << 56;
200       divisor <= b_m;
201       state <= divide_1;
202     end
203
204     divide_1:
205     begin
206       quotient <= quotient << 1;
207       remainder <= remainder << 1;
208       remainder[0] <= dividend[108];
209       dividend <= dividend << 1;
210       state <= divide_2;
211     end
212
213     divide_2:
214     begin
215       if (remainder >= divisor) begin
216         quotient[0] <= 1;
217         remainder <= remainder - divisor;
218       end
219       if (count == 107) begin
220         state <= divide_3;
221       end else begin
222         count <= count + 1;
223         state <= divide_1;
224       end
225     end
226
227     divide_3:
228     begin
229       z_m <= quotient[55:3];
230       guard <= quotient[2];
231       round_bit <= quotient[1];
232       sticky <= quotient[0] | (remainder != 0);
233       state <= normalise_1;
234     end
235
236     normalise_1:
237     begin
```

```
238       if (z_m[52] == 0 && $signed(z_e) > -1022) begin
239         z_e <= z_e - 1;
240         z_m <= z_m << 1;
241         z_m[0] <= guard;
242         guard <= round_bit;
243         round_bit <= 0;
244       end else begin
245         state <= normalise_2;
246       end
247     end
248
249     normalise_2:
250     begin
251       if ($signed(z_e) < -1022) begin
252         z_e <= z_e + 1;
253         z_m <= z_m >> 1;
254         guard <= z_m[0];
255         round_bit <= guard;
256         sticky <= sticky | round_bit;
257       end else begin
258         state <= round;
259       end
260     end
261
262     round:
263     begin
264       if (guard && (round_bit | sticky | z_m[0])) begin
265         z_m <= z_m + 1;
266         if (z_m == 53'hffffff) begin
267           z_e <=z_e + 1;
268         end
269       end
270       state <= pack;
271     end
272
273     pack:
274     begin
275       z[51 : 0] <= z_m[51:0];
276       z[62 : 52] <= z_e[10:0] + 1023;
277       z[63] <= z_s;
278       if ($signed(z_e) == -1022 && z_m[52] == 0) begin
279         z[62 : 52] <= 0;
280       end
281       //if overflow occurs, return inf
282       if ($signed(z_e) > 1023) begin
283         z[51 : 0] <= 0;
284         z[62 : 52] <= 2047;
285         z[63] <= z_s;
286       end
287       state <= put_z;
288     end
289
290     put_z:
```

```
      begin
        s_output_z_stb <= 1;
        s_output_z <= z;
        if (s_output_z_stb && output_z_ack) begin
          s_output_z_stb <= 0;
          state <= get_a;
        end
      end

    endcase

    if (rst == 1) begin
      state <= get_a;
      s_input_a_ack <= 0;
      s_input_b_ack <= 0;
      s_output_z_stb <= 0;
    end

  end
  assign input_a_ack = s_input_a_ack;
  assign input_b_ack = s_input_b_ack;
  assign output_z_stb = s_output_z_stb;
  assign output_z = s_output_z;

endmodule
```

Listing :8-آ double_divider.v

آ ـ ۲ Test Benches

آ ـ ۲ ـ ۱ Main_TB_2.v (Random test cases for all 4 modules)

```
//IEEE Floating Point Integration Test (Random Testing)
//Copyright (C) Mohsen Fayyaz 2019 "mohsenfayyaz.ir"
//2019-1-15
//
`timescale 1ns/1ns
`include "defines.v"

module Main_TB_2();
  reg clk=0, rst=1;
  reg[31:0] a_single, b_single;
  reg[63:0] a_double, b_double;
  wire[31:0] z_single;
  wire[63:0] z_double;
  reg a_stb, b_stb, z_ack;
  wire a_ack, b_ack, z_stb;
  reg[1:0] process;

```

```
18  integer fd;
19
20  always #10 clk = ~clk;  // 25MHz
21
22  Main main(
23        .input_as(a_single),
24        .input_bs(b_single),
25        .input_ad(a_double),
26        .input_bd(b_double),
27        .input_a_stb(a_stb),
28        .input_b_stb(b_stb),
29        .output_z_ack(z_ack),
30        .process(process),
31        .clk(clk),
32        .rst(rst),
33        .output_zs(z_single),
34        .output_zd(z_double),
35        .output_z_stb(z_stb),
36        .input_a_ack(a_ack),
37        .input_b_ack(b_ack));
38
39  initial begin
40    process = `PROCESS_SINGLE_DIVIDER;
41    fd = $fopen(`SINGLE_DIVIDER_OUTPUT_FILE_NAME,"w");
42    {a_single, b_single} = 0;
43    rst = 1;
44    #1000
45    rst = 0;
46    repeat(100) begin
47    z_ack = 0;
48    //a = 32'b10111111100000000000000000000000;  //-1
49    //b = 32'b01000000000000000000000000000000;  //2
50    a_single = $random;
51    a_stb = 1;
52    b_single = $random;
53    b_stb = 1;
54    #100;
55    a_stb = 0;
56    b_stb = 0;
57    while(!z_stb)
58      #500;
59    #10000;
60    $fwrite(fd, "%b %b %b\n", a_single, b_single, z_single);  //Unsigned Integer
61    z_ack = 1;
62    # 100;
63    end
64    $fclose(fd);
65    $display("File Created: %s", `SINGLE_DIVIDER_OUTPUT_FILE_NAME);
66
67
68    process = `PROCESS_SINGLE_SQRT;
69    fd = $fopen(`SINGLE_SQRT_OUTPUT_FILE_NAME,"w");
70    {a_single, b_single} = 0;
```

```
71    #1000
72    repeat(100) begin
73    z_ack = 0;
74    //a = 32'b10111111100000000000000000000000;  //-1
75    //b = 32'b01000000000000000000000000000000;  //2
76    a_single = $random;
77    a_stb = 1;
78    #100;
79    a_stb = 0;
80    while(!z_stb)
81      #500;
82    #10000;
83    $fwrite(fd, "%b %b\n", a_single, z_single);  //Unsigned Integer
84    z_ack = 1;
85    # 100;
86    end
87    $fclose(fd);
88    $display("File Created: %s", `SINGLE_SQRT_OUTPUT_FILE_NAME);
89
90
91    process = `PROCESS_DOUBLE_DIVIDER;
92    fd = $fopen(`DOUBLE_DIVIDER_OUTPUT_FILE_NAME,"w");
93    {a_double, b_double} = 0;
94    #1000
95    repeat(100) begin
96    z_ack = 0;
97    //a = 32'b10111111100000000000000000000000;  //-1
98    //b = 32'b01000000000000000000000000000000;  //2
99    a_double[63:32] = $random;  // $random gives at most 32 bit
100   a_double[31:0] = $random;
101   a_stb = 1;
102   b_double[63:32] = $random;  // $random gives at most 32 bit
103   b_double[31:0] = $random;
104   b_stb = 1;
105   #100;
106   #100;
107   a_stb = 0;
108   b_stb = 0;
109   while(!z_stb)
110     #500;
111   #10000;
112   $fwrite(fd, "%b %b %b\n", a_double, b_double, z_double);  //Unsigned Integer
113   z_ack = 1;
114   # 100;
115   end
116   $fclose(fd);
117   $display("File Created: %s", `DOUBLE_DIVIDER_OUTPUT_FILE_NAME);
118
119
120   process = `PROCESS_DOUBLE_SQRT;
121   fd = $fopen(`DOUBLE_SQRT_OUTPUT_FILE_NAME,"w");
122   {a_double, b_double} = 0;
123   #1000
```

```
    repeat(100) begin
    z_ack = 0;
    //a = 32'b10111111100000000000000000000000;  //-1
    //b = 32'b01000000000000000000000000000000;  //2
    a_double[63:32] = $random;  // $random gives at most 32 bit
    a_double[31:0] = $random;
    a_stb = 1;
    #100;
    a_stb = 0;
    while(!z_stb)
      #500;
    #10000;
    $fwrite(fd, "%b %b\n", a_double, z_double);  //Unsigned Integer
    z_ack = 1;
    # 100;
    end
    $fclose(fd);
    $display("File Created: %s", `DOUBLE_SQRT_OUTPUT_FILE_NAME);


    $stop;
  end

endmodule
```

Listing :9-آ Main_TB_2.v

Main_TB_2_corner_case.v ۲ـ۲ـآ

```
//IEEE Floating Point Integration Test (Corner Cases Test)
//Copyright (C) Mohsen Fayyaz 2019 "mohsenfayyaz.ir"
//2019-1-15
//
`timescale 1ns/1ns
`include "defines.v"

module Main_TB_2_corner_case();
  reg clk=0, rst=1;
  reg[31:0] a_single, b_single;
  reg[63:0] a_double, b_double;
  wire[31:0] z_single;
  wire[63:0] z_double;
  reg a_stb, b_stb, z_ack;
  wire a_ack, b_ack, z_stb;
  reg[1:0] process;
  integer i;
  integer fd;

  always #10 clk = ~clk;  // 25MHz

  Main main(
```

```
      .input_as(a_single),
      .input_bs(b_single),
      .input_ad(a_double),
      .input_bd(b_double),
      .input_a_stb(a_stb),
      .input_b_stb(b_stb),
      .output_z_ack(z_ack),
      .process(process),
      .clk(clk),
      .rst(rst),
      .output_zs(z_single),
      .output_zd(z_double),
      .output_z_stb(z_stb),
      .input_a_ack(a_ack),
      .input_b_ack(b_ack));

reg[31:0] a_single_tests[0:20];
reg[31:0] b_single_tests[0:20];
reg[63:0] a_double_tests[0:20];
reg[63:0] b_double_tests[0:20];
initial begin
  // <SINGLE CORNER CASES
  // Extremely small normalised numbers
  a_single_tests[0] = 32'b10000000100000000000000000000000;
  b_single_tests[0] = 32'b00000000100000000000000000000001;
  a_single_tests[1] = 32'b00000001100000000000000000000110;
  b_single_tests[1] = 32'b10001101000000000000000000000110;
  // Extremely small denormalised numbers
  a_single_tests[2] = 32'b00000000000000000000000000000001;
  b_single_tests[2] = 32'b10000000000000000000000000000010;
  a_single_tests[3] = 32'b00000000000101000001000000100101;
  b_single_tests[3] = 32'b00000000011111111111111111111111;
  // Extremely large normalised numbers
  a_single_tests[4] = 32'b01111111010101010101010101010111;
  b_single_tests[4] = 32'b11111110111111111111111111111101;
  a_single_tests[5] = 32'b01111111011111111111111111111110;
  b_single_tests[5] = 32'b11111101011010111111010110111110;
  // Mixed Extreme cases
  a_single_tests[6] = 32'b11111110011111011111011111111010;
  b_single_tests[6] = 32'b00000000011000010101011101100010;
  a_single_tests[7] = 32'b00001101101000011000000101000001;
  b_single_tests[7] = 32'b11101011010010011010110100100010;
  // Special Cases
  a_single_tests[8] = 32'b01111111101000011000000101000001; //NAN
  b_single_tests[8] = 32'b00000000011000010101011101100010;
  a_single_tests[9] = 32'b01111111100000000000000000000000; //INF
  b_single_tests[9] = 32'b01111111100000000000000000000000; //INF
  a_single_tests[10] = 32'b01111111010101010101010101010110;
  b_single_tests[10] = 32'b01111111100000000000000000000000; //INF
  a_single_tests[11] = 32'b01111010101110100000011010011111;
  b_single_tests[11] = 32'b00000000000000000000000000000000; //0
  // SINGLE CORNER CASES/>

```

```
76    // <DOUBLE CORNER CASES
77    // Extremely small normalised numbers
78    a_double_tests[0] = 64'b0000000000010000010000000001000000000100000010001000001010000001;
79    b_double_tests[0] = 64'b0000000001000000010110000001000100000100100101100010001000001100;
80    a_double_tests[1] = 64'b0000000010000010110100010100011010010101101111101100101001000010;
81    b_double_tests[1] = 64'b0000001000000111001100000000100111101000111011011110100001000110;
82    // Extremely small denormalised numbers
83    a_double_tests[2] = 64'b0000000000000000000000000000000000000000000000000000000000000001;
84    b_double_tests[2] = 64'b0000000000000010000001000000100010001000000100110000010001010000;
85    a_double_tests[3] = 64'b0000000000000000000001100001100010101000001110110000011101100011;
86    b_double_tests[3] = 64'b0000000000000000000000011100000010101000001100101000111011000010;
87    // Extremely large normalised numbers
88    a_double_tests[4] = 64'b0111111110111111101111111101111101110111011111111111111101110111;
89    b_double_tests[4] = 64'b1111111011101111111111111111111111111111111111111111111111111010;
90    a_double_tests[5] = 64'b0111111111000101101101101101110100110110101101111111011101010111;
91    b_double_tests[5] = 64'b1111111110111111101111111101111101110111011111111111111101110111;
92    // Mixed Extreme cases
93    a_double_tests[6] = 64'b0111111111000101101101101101110100110110101101111111011101010111;
94    b_double_tests[6] = 64'b0000000000000000000000000010101100110001000100101000000000000000;
95    a_double_tests[7] = 64'b0000000000000000000000000000000010101000001100101000111011000010;
96    b_double_tests[7] = 64'b0111111111000101101101101101110100110110101101111111011101010111;
97    // Special Cases
98    a_double_tests[8] = 64'b0111111111111101101101101101110001110010100101101111111101010101; //NAN
99    b_double_tests[8] = 64'b0111111111000101101101101101110100110110101101111111011101010111;
100   a_double_tests[9] = 64'b0111111111110000000000000000000000000000000000000000000000000000; //INF
101   b_double_tests[9] = 64'b0111111111110000000000000000000000000000000000000000000000000000; //INF
102   a_double_tests[10] = 64'b0000000000000010000000100000010001000100000001001100001000101010001;
103   b_double_tests[10] = 64'b1111111111110000000000000000000000000000000000000000000000000000; //-INF
104   a_double_tests[11] = 64'b0000000000000000000000000000000001010100000110010100011101100010;
105   b_double_tests[11] = 64'b0000000000000000000000000000000000000000000000000000000000000000; //0
106   // SINGLE CORNER CASES/>
107
108
109   process = `PROCESS_SINGLE_DIVIDER;
110   fd = $fopen(`SINGLE_DIVIDER_OUTPUT_FILE_NAME,"w");
111   {a_single, b_single} = 0;
112   rst = 1;
113   #1000
114   rst = 0;
115   for (i = 0; i < 12; i = i + 1) begin
116     z_ack = 0;
117     a_single = a_single_tests[i];
118     a_stb = 1;
119     b_single = b_single_tests[i];
120     b_stb = 1;
121     #100;
122     a_stb = 0;
123     b_stb = 0;
124     while(!z_stb)
125       #500;
126     #10000;
127     $fwrite(fd, "%b %b %b\n", a_single, b_single, z_single);  //Unsigned Integer
128     z_ack = 1;
```

```
    # 100;
  end
  $fclose(fd);
  $display("File Created: %s", `SINGLE_DIVIDER_OUTPUT_FILE_NAME);


  process = `PROCESS_SINGLE_SQRT;
  fd = $fopen(`SINGLE_SQRT_OUTPUT_FILE_NAME,"w");
  {a_single, b_single} = 0;
  #1000
  for (i = 0; i < 12; i = i + 1) begin
    z_ack = 0;
    a_single = a_single_tests[i];
    a_stb = 1;
    #100;
    a_stb = 0;
    while(!z_stb)
      #500;
    #10000;
    $fwrite(fd, "%b %b\n", a_single, z_single);  //Unsigned Integer
    z_ack = 1;
    # 100;
  end
  $fclose(fd);
  $display("File Created: %s", `SINGLE_SQRT_OUTPUT_FILE_NAME);


  process = `PROCESS_DOUBLE_DIVIDER;
  fd = $fopen(`DOUBLE_DIVIDER_OUTPUT_FILE_NAME,"w");
  {a_double, b_double} = 0;
  #1000
  for (i = 0; i < 12; i = i + 1) begin
    z_ack = 0;
    a_double = a_double_tests[i];
    a_stb = 1;
    b_double = b_double_tests[i];
    b_stb = 1;
    #100;
    a_stb = 0;
    b_stb = 0;
    while(!z_stb)
      #500;
    #10000;
    $fwrite(fd, "%b %b %b\n", a_double, b_double, z_double);  //Unsigned Integer
    z_ack = 1;
    # 100;
  end
  $fclose(fd);
  $display("File Created: %s", `DOUBLE_DIVIDER_OUTPUT_FILE_NAME);


  process = `PROCESS_DOUBLE_SQRT;
  fd = $fopen(`DOUBLE_SQRT_OUTPUT_FILE_NAME,"w");
```

```
182     {a_double, b_double} = 0;
183     #1000
184     for (i = 0; i < 12; i = i + 1) begin
185       z_ack = 0;
186       a_double = a_double_tests[i];
187       a_stb = 1;
188       #100;
189       a_stb = 0;
190       while(!z_stb)
191         #500;
192       #10000;
193       $fwrite(fd, "%b %b\n", a_double, z_double);  //Unsigned Integer
194       z_ack = 1;
195       # 100;
196     end
197     $fclose(fd);
198     $display("File Created: %s", `DOUBLE_SQRT_OUTPUT_FILE_NAME);
199
200
201     $stop;
202   end
203
204 endmodule
```

Listing :10-آ Main_TB_2_corner_case.v

single_divider_TB.v ۳ـ۲ـآ

```
1  `timescale 1ns/1ns
2  `include "defines.v"
3
4  module single_divider_TB();
5    reg clk=0, rst=1;
6    reg   [31:0] a, b;
7    wire   [31:0] z;
8    reg a_stb, b_stb, z_ack;
9    wire a_ack, b_ack, z_stb;
10   integer fd;
11
12   always #10 clk=~clk;  // 25MHz
13
14   divider_newton divider_0(
15     .clk(clk),
16     .rst(rst),
17     .input_a(a),
18     .input_a_stb(a_stb),
19     .input_a_ack(a_ack),
20     .input_b(b),
21     .input_b_stb(b_stb),
22     .input_b_ack(b_ack),
23     .output_z(z),
```

```
    .output_z_stb(z_stb),
    .output_z_ack(z_ack));
  initial begin
    fd = $fopen(`SINGLE_DIVIDER_OUTPUT_FILE_NAME,"w");

    a = 0;
    b = 0;
    rst = 1;
    #1000
    rst = 0;

    repeat(100) begin
    z_ack = 0;
    //a = 32'b10111111100000000000000000000000;  //-1
    //b = 32'b01000000000000000000000000000000;  //2
    a = $random;
    a_stb = 1;
    b = $random;
    b_stb = 1;
    #100;
    a_stb = 0;
    b_stb = 0;
    while(!z_stb)
      #500;
    #10000;
    $fwrite(fd, "%b %b %b\n", a, b, z);  //Unsigned Integer
    z_ack = 1;
    # 100;

    end

    $fclose(fd);
    $stop;
  end
endmodule
```

Listing :11-آ single_divider_TB.v

آ ـ ۲ ـ ۴ double_divider_TB.v

```
`timescale 1ns/1ns
`include "defines.v"

module double_divider_TB();
  reg clk=0, rst=1;
  reg   [63:0] a, b;
  wire   [63:0] z;
  reg a_stb, b_stb, z_ack;
  wire a_ack, b_ack, z_stb;
  integer fd;

```

```
12   always #10 clk=~clk;  // 25MHz
13
14   double_divider_newton divider_0(
15     .clk(clk),
16     .rst(rst),
17     .input_a(a),
18     .input_a_stb(a_stb),
19     .input_a_ack(a_ack),
20     .input_b(b),
21     .input_b_stb(b_stb),
22     .input_b_ack(b_ack),
23     .output_z(z),
24     .output_z_stb(z_stb),
25     .output_z_ack(z_ack));
26   initial begin
27     fd = $fopen(`DOUBLE_DIVIDER_OUTPUT_FILE_NAME,"w");
28
29     a = 0;
30     b = 0;
31     rst = 1;
32     #1000
33     rst = 0;
34
35     repeat(100) begin
36     z_ack = 0;
37     //a = 32'b10111111100000000000000000000000;  //-1
38     //b = 32'b01000000000000000000000000000000;  //2
39     a[63:32] = $random;  // $random gives at most 32 bit
40     a[31:0] = $random;
41     a_stb = 1;
42     b[63:32] = $random;  // $random gives at most 32 bit
43     b[31:0] = $random;
44     b_stb = 1;
45     #100;
46     a_stb = 0;
47     b_stb = 0;
48     while(!z_stb)
49       #500;
50     #10000;
51     $fwrite(fd, "%b %b %b\n", a, b, z);  //Unsigned Integer
52     z_ack = 1;
53     # 100;
54
55     end
56
57     $fclose(fd);
58     $stop;
59   end
60 endmodule
```

Listing :12-آ double_divider_TB.v

آ ـ ۲ ـ ۵ single_sqrt_TB.v

```
1  `timescale 1ns/1ns
2  `include "defines.v"
3
4  module single_sqrt_TB();
5    reg clk=0, rst=1;
6    reg   [31:0] a;
7    wire   [31:0] z;
8    reg a_stb, z_ack;
9    wire a_ack, z_stb;
10   integer fd;
11
12   always #10 clk=~clk;  // 25MHz
13
14   sqrt sqrt_UUT(
15     .clk(clk),
16     .rst(rst),
17     .input_a(a),
18     .input_a_stb(a_stb),
19     .input_a_ack(a_ack),
20     .output_z(z),
21     .output_z_stb(z_stb),
22     .output_z_ack(z_ack));
23
24   initial begin
25     fd = $fopen(`SINGLE_SQRT_OUTPUT_FILE_NAME,"w");
26
27     a = 0;
28     rst = 1;
29     #1000
30     rst = 0;
31
32     repeat(100) begin
33     z_ack = 0;
34     //a = 32'b01000001110010000000000000000000;
35     //a = 32'b00111111101111110110111100000101;
36     a = $random;
37     a_stb = 1;
38     #100;
39     a_stb = 0;
40     while(!z_stb)
41       #500;
42     #10000;
43     $fwrite(fd, "%b %b\n", a, z);  //Unsigned Integer
44     z_ack = 1;
45     # 100;
46
47     end
48
49     $fclose(fd);
50     $stop;
51   end
```

```
endmodule
```

Listing :13-آ single_sqrt_TB.v

آ ـ ۲ ـ ۶ double_sqrt_TB.v

```
`timescale 1ns/1ns
`include "defines.v"

module double_sqrt_TB();
  reg clk=0, rst=1;
  reg   [63:0] a;
  wire   [63:0] z;
  reg a_stb, z_ack;
  wire a_ack, z_stb;
  integer fd;

  always #10 clk=~clk;  // 25MHz

  double_sqrt sqrt_UUT(
    .clk(clk),
    .rst(rst),
    .input_a(a),
    .input_a_stb(a_stb),
    .input_a_ack(a_ack),
    .output_z(z),
    .output_z_stb(z_stb),
    .output_z_ack(z_ack));

  initial begin
    fd = $fopen(`DOUBLE_SQRT_OUTPUT_FILE_NAME, "w");

    a = 0;
    rst = 1;
    #1000
    rst = 0;

    repeat(100) begin
    z_ack = 0;
    //a = 64'b0100000000010000000000000000000000000000000000000000000000000000;
    //a = 64'b0011111101111111011011111000000101;
    a[63:32] = $random;  // $random gives at most 32 bit
    a[31:0] = $random;
    a_stb = 1;
    #100;
    a_stb = 0;
    while(!z_stb)
      #500;
    #10000;
    $fwrite(fd, "%b %b\n", a, z);  //Unsigned Integer
    z_ack = 1;
```

```
    # 100;

    end

    $fclose(fd);
    $stop;
  end
endmodule
```

Listing :14-آ double_sqrt_TB.v

Golden Model ۳ـآ

single_divider.cpp ۱ـ۳ـآ

```
#include <iostream>
#include <stdio.h>
#include <math.h>
#include <sstream>
#include <fstream>
#include <string>
#include<bitset>
using namespace std;
string intToBinary(int n, int i=32)
{
    string result = "";
    // Prints the binary representation
    // of a number n up to i-bits.
    int k;
    for (k = i - 1; k >= 0; k--) {

        if ((n >> k) & 1)
            result += "1";
        else
            result += "0";
    }
    return result;
}

int main()
{
    /*cout << "Hello world!" << endl;
    float f = 1.0;
  unsigned int a, b, i;
    *(int*)&f = 0b00010010000010101001101010010010;
    cout << f << endl << sqrt(f) << endl;
    i = *(int*)&f;
    printf("%b\n", f);*/

```

```

   string in1, in2, out;
  float input1, input2, output, divideResult;
   string sample;
   ifstream file;
   file.open("../../ModelSim/output/single_divider_output.txt");
   while (getline(file, sample))
   {
       stringstream ss(sample);
       ss >> in1;
       ss >> in2;
       ss >> out;
       *(int*)&input1 = (int) bitset<64>(in1).to_ulong();  // String to float number
       *(int*)&input2 = (int) bitset<64>(in2).to_ulong();
       *(int*)&output = (int) bitset<64>(out).to_ulong();
       divideResult = input1/input2;

       cout << endl;
       cout << "INPUT    : " <<  in1 << "/" << in2 << " = " << out << endl;
       cout << "CPP CODE : " << intToBinary(*(int*)&input1) << "/" << intToBinary(*(int*)&(input2)) << " = " <<
    intToBinary(*(int*)&(divideResult)) << endl;
       cout << "FLOAT VAL: " << input1 << "/" << input2 << " = " << output << "|" << divideResult << endl;

       if(*(int*)&(divideResult) == *(int*)&(output) || (isnan(output) && isnan(divideResult)))
       {
           cout << "OK" << endl;
       }
       else
       {
           cout << "ERROR ----------------------------!!!!" << endl;
       }
   }


   cout << endl << endl << "Press any key to continue." << endl;
   getchar();
   return 0;
}
```

Listing :15-آ single_divider.cpp

آ ـ ۳ ـ ۲ double_divider.cpp

```
#include <iostream>
#include <stdio.h>
//#include <math.h>
#include <sstream>
#include <fstream>
#include <string>
#include<bitset>
#include <cmath>
```

```
#include <bits/stdc++.h>
#include <errno.h>
#include <cstdlib>
using namespace std;
string intToBinary(unsigned long long n, int i=32)
{
    string result = "";
    // Prints the binary representation
    // of a number n up to i-bits.
    int k;
    for (k = i - 1; k >= 0; k--) {
        /*if (i == 32 && k == 22){
            result += ".";
        }
        if(i == 64 && k == 51){
            result += ".";
        }*/
        if ((n >> k) & 1)
            result += "1";
        else
            result += "0";
    }
    return result;
}

string doubleToBinary(double doubleValue){
    uint8_t *bytePointer = (uint8_t *)&doubleValue;
    string result = "";
    for(size_t index = 0; index < sizeof(double); index++)
    {
        uint8_t byte = bytePointer[index];

        for(int bit = 0; bit < 8; bit++)
        {
            //printf("%d", byte&1);
            if (byte&1)
                result += "1";
            else
                result += "0";
            byte >>= 1;
        }
    }
    return result;
}


int main()
{
    /*cout << "Hello world!" << endl;
    float f = 1.0;
  unsigned int a, b, i;
    *(int*)&f = 0b00010010000101010011010100100100;
    cout << f << endl << sqrt(f) << endl;
```

```
    i = *(int*)&f;
    printf("%b\n", f);*/


    string in1, in2, out;
  double input1=1.0, input2=1.0, output=1.0, divideInput;
    string sample;
    ifstream file;
    file.open("../../ModelSim/output/double_divider_output.txt");
    while (getline(file, sample))
    {
        stringstream ss(sample);
        ss >> in1;
        ss >> in2;
        ss >> out;
        *(unsigned long long*)&input1 = (unsigned long long) strtoull(in1.c_str(), nullptr, 2);  // & gets the pointer to input memory, (int*) casts the address to int* and * at last gets it's value in int
        *(unsigned long long*)&input2 = (unsigned long long) strtoull(in2.c_str(), nullptr, 2);
        *(unsigned long long*)&output = (unsigned long long) strtoull(out.c_str(), nullptr, 2);
        divideInput = ((double)input1)/((double)input2);


        cout << endl;
        cout << "INPUT    : " <<  in1 << "/" << in2 << " = " << out << endl;
        cout << "CPP CODE : " << intToBinary(*(unsigned long long*)&input1, 64) << "/" << intToBinary(*(unsigned long long*)&input2, 64) << " = " << intToBinary(*(unsigned long long*)&(divideInput), 64) << endl;
        cout << "DOUBLE VAL: " << input1 << "/" << input2 << " = " << output << "|" << divideInput << endl;

        if(*(long int*)&(divideInput) == *(long int*)&(output) || (isnan(output) && isnan(divideInput)) )
        {
            cout << "OK" << endl;
        }
        else
        {
            cout << "ERROR ----------------------------!!!!" << endl;
        }
    }

    cout << endl << endl << "Press any key to continue." << endl;
    getchar();
    return 0;
}
```

Listing :16-آ double_divider.cpp

single_sqrt.cpp ۳ـ۳ـآ

```
#include <iostream>
#include <stdio.h>
//#include <math.h>
#include <sstream>
```

```
#include <fstream>
#include <string>
#include<bitset>
#include <cmath>
using namespace std;
string intToBinary(long int n, int i=32)
{
    string result = "";
    // Prints the binary representation
    // of a number n up to i-bits.
    int k;
    for (k = i - 1; k >= 0; k--) {
        /*if (i == 32 && k == 22){
            result += ".";
        }
        if(i == 64 && k == 51){
            result += ".";
        }*/
        if ((n >> k) & 1)
            result += "1";
        else
            result += "0";
    }
    return result;
}

int main()
{
    /*cout << "Hello world!" << endl;
    float f = 1.0;
  unsigned int a, b, i;
    *(int*)&f = 0b00010010000101010011010100100100;
    cout << f << endl << sqrt(f) << endl;
    i = *(int*)&f;
    printf("%b\n", f);*/


    string in, out;
  float input=1.0, output=1.0, sqrtInput;
    string sample;
    ifstream file;
    file.open("../../ModelSim/output/single_sqrt_output.txt");
    while (getline(file, sample))
    {
        stringstream ss(sample);
        ss >> in;
        ss >> out;
        *(int*)&input = (int) bitset<64>(in).to_ulong();  // & gets the pointer to input memory, (int*) casts the
      address to int* and * at last gets it's value in int
        *(int*)&output = (int) bitset<64>(out).to_ulong();
        sqrtInput = sqrt((double)input);


```

```
        cout << endl;
        double ff = sqrt((double)input);
        float f = ff;
        //cout << "RRRRR    : " << intToBinary(*(int*)&f) << endl;
        cout << "INPUT    : " <<  in << "|" << out << endl;
        cout << "CPP CODE : " << intToBinary(*(int*)&input) << "|" << intToBinary(*(int*)&(sqrtInput)) << endl;
        cout << "FLOAT VAL: " << input << "|" << output << "|" << sqrtInput << endl;

        if(*(int*)&(sqrtInput) == *(int*)&(output) || (isnan(output) && isnan(sqrtInput)))
        {
            cout << "OK" << endl;
        }
        else
        {
            cout << "ERROR ----------------------------!!!!" << endl;
        }
    }

    cout << endl << endl << "Press any key to continue." << endl;
    getchar();
    return 0;
}
```

Listing :17-آ single_sqrt.cpp

آ_۳_۴ double_sqrt.cpp

```
#include <iostream>
#include <stdio.h>
//#include <math.h>
#include <sstream>
#include <fstream>
#include <string>
#include<bitset>
#include <cmath>
#include <bits/stdc++.h>
#include <errno.h>
#include <cstdlib>
using namespace std;
string intToBinary(unsigned long long n, int i=32)
{
    string result = "";
    // Prints the binary representation
    // of a number n up to i-bits.
    int k;
    for (k = i - 1; k >= 0; k--) {
        /*if (i == 32 && k == 22){
            result += ".";
        }
        if(i == 64 && k == 51){
            result += ".";
```

```
        }*/
        if ((n >> k) & 1)
            result += "1";
        else
            result += "0";
    }
    return result;
}

string doubleToBinary(double doubleValue){
    uint8_t *bytePointer = (uint8_t *)&doubleValue;
    string result = "";
    for(size_t index = 0; index < sizeof(double); index++)
    {
        uint8_t byte = bytePointer[index];

        for(int bit = 0; bit < 8; bit++)
        {
            //printf("%d", byte&1);
            if (byte&1)
                result += "1";
            else
                result += "0";
            byte >>= 1;
        }
    }
    return result;
}


int main()
{
    /*cout << "Hello world!" << endl;
    float f = 1.0;
  unsigned int a, b, i;
    *(int*)&f = 0b00010010000010101001101010010100;
    cout << f << endl << sqrt(f) << endl;
    i = *(int*)&f;
    printf("%b\n", f);*/


    string in, out;
  double input=1.0, output=1.0, sqrtInput;
    string sample;
    ifstream file;
    file.open("../../ModelSim/output/double_sqrt_output.txt");
    while (getline(file, sample))
    {
        stringstream ss(sample);
        ss >> in;
        ss >> out;
        *(unsigned long long*)&input = (unsigned long long) strtoull(in.c_str(), nullptr, 2);  // & gets the pointer
      to input memory, (int*) casts the address to int* and * at last gets it's value in int
```

```
77          *(unsigned long long*)&output = (unsigned long long) strtoull(out.c_str(), nullptr, 2);
78          sqrtInput = sqrt((double)input);
79
80
81          cout << endl;
82          double ff = sqrt((double)input);
83          //float f = ff;
84          //cout << "RRRRR    : " << intToBinary(*(unsigned long long*)&f) << endl;
85          cout << "INPUT    : " <<  in << "|" << out << endl;
86          cout << "CPP CODE : " << intToBinary(*(unsigned long long*)&input, 64) << "|" << intToBinary(*(unsigned long
        long*)&(sqrtInput), 64) << endl;
87          cout << "DOUBLE VAL: " << input << "|" << output << "|" << sqrtInput << endl;
88
89          if(*(long int*)&(sqrtInput) == *(long int*)&(output)  || (isnan(output) && isnan(sqrtInput)))
90          {
91              cout << "OK" << endl;
92          }
93          else
94          {
95              cout << "ERROR ----------------------------!!!!" << endl;
96          }
97      }
98
99      cout << endl << endl << "Press any key to continue." << endl;
100     getchar();
101     return 0;
102 }
```

Listing :18-آ double_sqrt.cpp