به نام خدا

دانشگاه تهران
دانشکدهٔ مهندسی برق و کامپیوتر

پروژهٔ نهایی درس

طراحی کامپیوتری سیستم های دیجیتال

Floating Point Arithmetic Unit

مدرس: دکتر بهاروند

اعضای گروه:

علیرضا سالمی، رضا قنبری، امید بداقی،

امیرحسین عبادی، مبینا شاه بنده

پاییز ۹۸

فهرست مطالب

۱ مقدمه ۸

۱ـ۱ چکیده ۸

۱ـ۲ تاریخچه و کاربردها ۸

۱ـ۳ نحوه عملکرد کلی ۱۲

۱ـ۳ـ۱ ماژول های اصلی ۱۲

۱ـ۳ـ۲ مدل طلایی ۱۳

۱ـ۴ پایهٔ ریاضی ۱۵

۱ـ۴ـ۱ عملیات تقسیم با استفاده از الگوریتم Long Division ۱۸

۱ـ۴ـ۲ عملیات جذر با استفاده از الگوریتم Shifting nth Root ۲۰

۲ توصیف معماری سیستم ۲۳

۲ـ۱ Interface ۲۳

۲ـ۱ـ۱ Input Interface ۲۳

۲ـ۱ـ۲ Output Interface ۲۵

۲ـ۲ ماژول تقسیم ۲۶

۲ـ۲ـ۱ ماژول div_unit ۲۶

۲ـ۲ـ۲ ماژول combinational_div ۲۷

۲ـ۲ـ۳ ماژول divider ۲۷

۲ـ۲ـ۴ تست قسمت های تقسیم ۲۹

۲ـ۲ـ۵ تقسیم Wrapper ۲۹

۲ـ۲ـ۶ تعداد مراحل اجرای الگوریتم تقسیم ۳۰

۲ـ۲ـ۷ محاسبه جواب نهایی تقسیم ۳۰

۲ـ۲ـ۸ محاسبه توان نهایی ۳۱

۲ـ۲ـ۹ تعیین Overflow و Underflow ۳۲

۲ـ۲ـ۱۰ واحد کنترل ۳۳

۲ـ۲ـ۱۱ ماژول Wrapper ۳۴
۲ـ۳ ماژول جذر ۳۵
۲ـ۳ـ۱ پیاده‌سازی کلی ۳۵
۲ـ۳ـ۲ ماژول Input Wrapper ۳۷
۲ـ۳ـ۳ ماژول Exponent Handler ۴۲
۲ـ۳ـ۴ ماژول Output Wrapper ۴۴
۲ـ۳ـ۵ ماژول SQRT ۴۵
۲ـ۳ـ۶ واحد کنترل ۴۷
۲ـ۳ـ۷ ماژول First One Finder ۴۸
۲ـ۳ـ۸ ماژول sqrt_unit ۴۹
۲ـ۴ رسیدگی به حالات استثناء ۵۲
۲ـ۵ ماژول FPU ۵۳
۲ـ۵ـ۱ ماژول Input Interface ۵۳
۲ـ۵ـ۲ ماژول‌های جذر و تقسیم ۵۴
۲ـ۵ـ۳ مالتی‌پلکسرهای خروجی ۵۴
۲ـ۵ـ۴ ماژول Output Interface ۵۴
۲ـ۶ واسط کاربری ۵۷
۲ـ۶ـ۱ اسکریپت پایتون ۵۷
۲ـ۶ـ۲ اسکریپت TCL ۵۸
۲ـ۶ـ۳ فایل ورودی ۵۸
۲ـ۶ـ۴ فایل خروجی ۵۹
۳ شبیه سازی و تست ۶۰
۳ـ۱ Test Bench ۶۰
۳ـ۱ـ۱ ورودی و خروجی ۶۰
۳ـ۱ـ۲ ریست و خاتمه عملیات ۶۰

۳ـ۱ـ۳ دادن ورودی‌ها به ماژول اصلی ۶۰
۳ـ۱ـ۴ شمارنده کلاک ۶۱
۳ـ۲ نمونه هایی از تست مدار روی Wave ۶۱
۳ـ۳ درستی آزمایی ۶۳
۴ سنتز ۶۵
۴ـ۱ گزارش زمانی مدار ۶۶
۴ـ۲ گزارش کلاک ۶۷
۴ـ۳ گزارش Clock Interaction ۶۸
۴ـ۴ گزارش بهره‌برداری ۶۸
۴ـ۵ گزارش توان ۶۹
۴ـ۶ سایر گزارش‌های موجود در فایل Log ۷۱
۴ـ۶ـ۱ FSM های تشخیص داده شده و کدینگ جدید آن‌ها ۷۱
۴ـ۶ـ۲ گزارش سیم‌هایی با چند Driver ۷۲
۴ـ۶ـ۳ گزارش Component های مصرف شده ۷۲
۴ـ۶ـ۴ گزارش فضای مصرف شده توسط هر ماژول ۷۳
۵ نتیجه گیری ۷۵
مراجع ۷۶

فهرست تصاویر

شکل ۱ : بخش های مختلف یک عدد ممیز شناور ۱۱
شکل ۲: شکل کلی مدار ۱۳
شکل ۳ : div_unit ۲۷
شکل ۴: قسمت های ترکیبی و ترتیبی ماژول تقسیم ۲۸
شکل ۵ : واحد کنترل ماژول تقسیم ۳۳
شکل ۶: ماژول SQRT ۳۶
شکل ۷: سیگنال های ورودی Input Wrapper برای ماژول جذر ۳۸
شکل ۸ : دریافت توان و اعشار در ماژول جذر ۴۱
شکل ۹ : واحد کنترل Input Wrapper ماژول جذر ۴۲
شکل ۱۰: Exponent Handler ماژول جذر ۴۳
شکل ۱۱ : Output Wrapper ماژول جذر ۴۴
شکل ۱۲ : مسیر داده ماژول SQRT ۴۷
شکل ۱۳ : واحد کنترل ماژول جذر ۴۸
شکل ۱۴ : ماژول first_one_finder ۴۹
شکل ۱۵ : ماژول sqrt_unit ۵۱
شکل ۱۶ : مسیر داده ماژول FPU ۵۵
شکل ۱۷ : واحد کنترل ماژول FPU ۵۶
شکل ۱۸ : نتیجه شبیه سازی – عملیات تقسیم ـ ۱ ..... ۶۱
شکل ۱۹ : نتیجه شبیه سازی – عملیات جذر ـ ۱ ..... ۶۲
شکل ۲۰ : نتیجه شبیه سازی – عملیات تقسیم ـ ۲ ..... ۶۲
شکل ۲۱: نتیجه شبیه سازی ـ عملیات جذر ـ ۲ ..... ۶۳
شکل ۲۲ : نقشه قرار گیری مدار روی بورد ۶۵

فهرست جداول

جدول ۱ : مقادیر مختلف Operation ها ۲۳
جدول ۲ : مقادیر مختلف Flag ها ۲۴
جدول ۳: حالت های خاص برای تقسیم ۵۲
جدول ۴ : حالت های خاص برای جذر ۵۲
جدول ۵ : گزارش زمانی سنتز ۶۶
جدول ۶ : گزارش زمانی پیاده سازی ۶۶
جدول ۷ : گزارش کلاک ۶۷
جدول ۸ : گزارش Clock Interaction ۶۸
جدول ۹ : گزارش بهره برداری ۶۸
جدول ۱۰ : گزارش توان ۶۹
جدول ۱۱ : FSM های فایل Log ۷۱
جدول ۱۲ : گزارش سیم های با چند Driver ۷۲
جدول ۱۳ : گزارش Component های مصرف شده ۷۲
جدول ۱۴ : گزارش فضای مصرفی هر ماژول ۷۳

جدول فعالیت اعضاء گروه

	ارزیابی اولیه	پیاده‌سازی	تست	سنتز	گزارش
مبینا شاه بنده	یافتن الگوریتم تقسیم	ماژول تقسیم	اسکریپت پایتون و TCL و تست کلی	بررسی شماتیک	تدوین صفحات و توضیح بخش تست و اسکریپت
امید بداقی	یافتن الگوریتم تقسیم	ماژول تقسیم	واحد تقسیم و تست کلی	بررسی شماتیک	نگارش بخش پیاده‌سازی
امیرحسین عبادی	یافتن مدل طلایی	ماژول جذر	اتصال مدل طلایی به فرایند تست و تست کلی	بررسی شماتیک	نگارش مقدمه و توضیح مدل طلایی
رضا قنبری	یافتن الگوریتم جذر	ماژول جذر	واحد جذر و تست کلی	انجام سنتز و اجرای تست فانکشنال پس از سنتز	نگارش بخش پیاده‌سازی
علیرضا سالمی	یافتن الگوریتم جذر	رابط‌های ورودی خروجی	نوشتن کد ارزیابی کننده برای ماژول اصلی و تست کلی	انجام سنتز و اجرای تست فانکشنال پس از سنتز	نگارش بخش ارزیابی ریاضی و نتایج سنتز

۱ مقدمه

۱ـ۱ چکیده

در این پروژه یک واحد پردازش محاسبات اعشاری با استفاده از زبان های توصیف سخت افزار ، دارای قابلیت های تقسیم و جذر پیاده سازی شده است. کاربر با وارد کردن اعداد اعشاری با دقت واحد[۱] یا دقت مضاعف مورد نظر خود و هم چنین تعیین کردن عملیات منتخب خویش، پاسخ دقیق عملیات را بدست می آورد. در این پروژه برای آزمودن صحت عملکرد پاسخ های خروجی از کتابخانه mpmath که به زبان پایتون پیاده سازی شده است، استفاده شده است. الگوریتم های بهینه جذر و تقسیم انتخاب شده در **بخش ۱ـ۴** به تفصیل توضیح داده شده است.

۱ـ۲ تاریخچه و کاربردها

در گذشته سیستم های کامپیوتری شامل یک بخش جداگانه پردازش محاسبات اعشاری نبودند بلکه عملیات های ریاضیاتی خود را توسط یک پردازنده مکمل [۲]انجام میداند. برای مثال حضور یک واحد پردازشی اعشاری در داخل واحد پردازنده گرافیکی[۳] الزامی بود و تمامی عملیات های اعشاری توسط این واحد صورت میگرفت. روش دیگر انجام عملیات های وابسته به ممیز شناور[۴] توسط نرم افزار ها بود که در این حالت سربار کاری واحد پردازش مرکزی[۵] افزوده گردیده و از سخت افزار خاص منظور برای محاسبات اعشاری استفاده نمی گشت. یکی از روش های محاسبات نرم افزاری استفاده از توابع CORDIC[۶] است که با استفاده از توابع مثلثاتی و هذلولوی قادر به انجام عملیات های وابسته به ممیز شناور در سطح نرم افزار می شوند. در سری های ,Intel 8087 80287, 80387 تا سری 80486 (حدود سال ۱۹۸۹) از این روش ها برای محاسبات اعشاری استفاده می شد. از دهه ۱۹۹۰ به بعد بیشتر ریزپردازنده ها از یک یا چند واحد محاسباتی اعشاری استفاده میکردند.

---

[1] Single precision

[2] Co-processor

[3] Graphical processing unit

[4] Floating point

[5] Central processing unit

[6] Coordinate rotation digital computer

در معماری های فعلی کامپیوتر ها بخش ممیز شناور و صحیح از یکدیگر جدا شده اند و برای مثال رجیستر های خاص منظوره برای ممیز شناور در نظر گرفته شده است. در معماری x87 اینتل که زیرمجموعه ای از دستورات معماری معروف x86 اینتل است، از یک واحد پردازش اعشاری برای اعداد اعشاری استفاده شده است که همراه واحد پردازش مرکزی های دیگر x86 کار میکند و با مجموعه دستورالعمل[1] خاصی که در دسترس میگذارد، محاسبات اعشاری را تسریع می کند. البته در دستورالعمل های این پردازنده قابلیت های اندک محاسباتی مانند جمع، تفریق و مقایسه به همراه برخی توابع پیچیده تر ریاضیاتی مانند محاسبه تانژانت صورت میگیرد. پیش از اضافه شدن این واحد محاسباتی به پردازنده ها، کامپایلرها می بایست از فراخوانی های کند کتابخانه ای استفاده میکردند.

در دهه 1980 در معماری های ارائه شده توسط IBM مکان سوکتی برای یک واحد پردازش اعشاری دیده میشد که کاملا جدا از ساختار واحد پردازش مرکزی تعبیه شده بود. اینگونه واحد پردازش اعشاری ها امروزه نیز در واحد های ریزپردازنده و کامپیوتر های تک بردی[2] دیده می شود که در صورت اضافه کردن آنها به بورد مورد نظر و استفاده از محیط های یکپارچه سازی نرم افزاری[3] خاص خودشان، می توان سرعت پردازش اعشاری را در این سیستم ها بالا برد.

همانطور که مکرراً ذکر شد، از نقاط مثبت و کاربرد های واحد پردازش اعشاری تسریع عملیات هایی است که بر پایه ی محاسبات ریاضی اعشاری هستند. همچنین امروزه در واحدهای پردازش اعشاری با استفاده از خط انتقال[4]، سرعت بخشی به دریافت[5] دستورالعمل ها و بازدهی بیشتر اتفاق افتاده است. اما از طرفی به هنگام برنامه ریزی کردن[6] این دستورات در خط انتقال به علت اینکه مانند بقیه دستورات عادی نیستند، باید عملیات های برنامه ریزی بسیار با دقت انجام شوند و هر دستور بعد از مدت زمان معین (به عنوان مثال ۱۴ سیکل زمانی) از خط انتقال خارج شود و جواب را به باس خروجی بدهد. در غیر اینصورت دچار تداخل بین دستورات می شویم و سیستم به حالت غیرمطلوبی می رود.

---

[1] Instruction set architecture
[2] Single-board computer
[3] IDE
[4] Pipeline
[5] Load
[6] Scheduling

انواع مدل های مختلف واحد پردازش اعشاری را می توان در مدار های مختلف دید اما همانطور که ذکر شد در حالت کلی به سه دسته تقسیم میشوند. دسته اول همان کتابخانه های موجود نرم افزاری هستند که به اصطلاح به عنوان واحد پردازش اعشاری عمل میکنند. این کتابخانه ها با استفاده از واحد پردازش مرکزی به ترتیب دستورات معادل این واحد را ایجاد کرده و با سرعت کندتری نسبت به حالت فیزیکی واحد پردازش اعشاری محاسبات خود را انجام می دهد. البته از طرفی از لحاظ هزینه و مکان روی بورد دارای مزیت نسبت به واحدهای فیزیکی است. دسته دوم این پردازنده ها بصورت قابل اضافه شدن هستند که می توان آنها را با اتصال به بورد کامپیوتر خود و نصب درایور مخصوص در پردازش های اعشاری استفاده نمود. دسته سوم نیز بصورت مشخص از ابتدا در روی بورد قرار گرفته اند و درکنار سایر واحد های پردازنده عملیات اعشاری را با مجموعه دستورات مشخص خود انجام می دهند. امروزه بیشتر دسته سوم این پردازنده ها مورد توجه طراحان سیستم های کامپیوتری قرار گرفته که علت عمده آن بهینه بودن و سرعت بالا این مدل طراحی است.

با توجه به اینکه محاسبات اعشاری اهمیت بسیار زیادی پیدا کرده است، شرکت هایی مانند Intel با قرار دادن یک واحد پردازش اعشاری ــ که از محاسبات تجاری[1] بهره می گیرد ــ روی بورد خود استفاده می کنند. این پردازنده میتواند اعداد صحیح تا دقت 10^{18} را بصورت کاملا دقیق و بدون خطا محسابه کند و اعداد بزرگتر از این مقدار را رند کند. هم چنین این واحد پردازشی از قابلیت های دیگری مانند رند کردن جهت دار، پشتیبانی از اعداد بسیار کوچک[2] و برنامه های جدا برای کنترل کردن استثنائات[3] بهره می برد.

استاندارد IEEE ممیز شناور

در استانداردهای IEEE استاندارد ۷۵۴ مباحث مربوط به ممیز شناور را توضیح می دهد. این استاندارد مباحث زیر را پوشش میدهد :

۱. فرمت های معماری اعداد: مجموعه ای از اعداد دودویی[4] و دهدهی[5] که شامل اعداد متناهی، نامتناهی و NaN می باشد

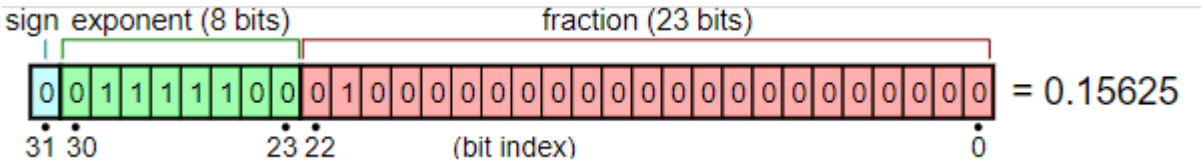
---

[1] Commercial computing
[2] Gradual underflow
[3] Exception handlers
[4] Binary
[5] Decimal

۲. فرمت تبدیل اعداد: تبدیل های مورد استفاده برای تغییر داده از یک encoding به دیگری برای بهینه شدن فرآیند ها
۳. قواعد رند کردن
۴. عملیات ها: عملیات های حسابی و مثلثاتی استاندارد
۵. کنترل کردن حالات استثناء

فرمت اعداد

اعداد ممیز شناور به دو حالت دقت واحد و مضاعف هستند که به ترتیب ۳۲ بیت و ۶۴ بیت میباشند. در شکل زیر حالت ۳۲ بیت را مشاهده می کنید. این عدد ۳۲ بیتی شامل ۱ بیت علامت، ۸ بیت توان و ۲۳ بیت اعشار است.



شکل ۱ : بخش های مختلف یک عدد ممیز شناور

اعداد ممیز شناور مضاعف که ۶۴ بیتی هستند دارای ۱ بیت علامت، 11 بیت توان و ۵۳ بیت اعشار است.

نحوه محاسبه اعداد ممیز شناور دقت واحد به صورت زیر است:

$$value = (-1)^{sign} * 2^{exponent-127} * (1 + \sum_{1}^{23} frac_{23-i} * 2^{-i})$$

نحوه محاسبه اعداد ممیز شناور دقت مضاعف به صورت زیر است:

$$value = (-1)^{sign} * 2^{exponent-1023} * (1 + \sum_{1}^{52} frac_{52-i} * 2^{-i})$$

در این پروژه عملیات های مربوط به اعداد به فرمت نرمال پشتیبانی می شوند. هم چنین اعداد INF و NaN و صفر نیز پشتیبانی می شود و تنها اعداد غیر نرمال[1] که درواقع بخش توان آنها صفر است، پشتیبانی نمی شود و عملیات با آنها باعث ایجاد استثناء می گردد.

۱ـ۳ نحوه عملکرد کلی

۱ـ۳ـ۱ ماژول های اصلی

ماژول های اصلی از دید کلی در زیر آمده اند:

۱. **ماژول ورودی**

این ماژول ورودی های مورد نظر کاربر را دریافت کرده و هم چنین از کاربر عملیات مورد نیاز را نیز دریافت میکند. علاوه بر آن شرایط دقت واحد یا مضاعف بودن ورودی ها را نیز دریافت می‌کند. حال بر اساس اینکه عملیات، جذر یا تقسیم است و اینکه اعداد ورودی به کدام فرمت اعداد اعشاری هستند، ورودی های متناظر برای هر ماژول را ایجاد می‌کند.

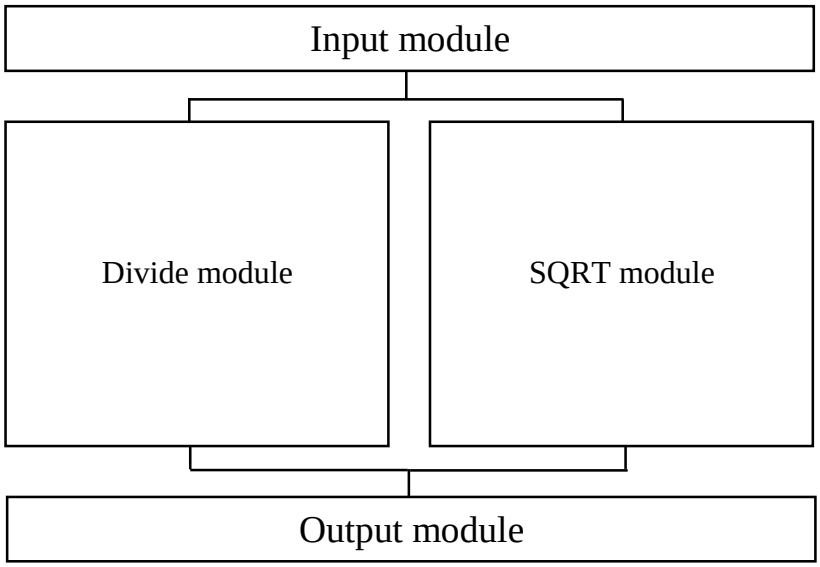
۲. **ماژول های عملیات**

این بخش شامل دو ماژول اصلی است که عملیات های متناظر را انجام می‌دهند. ماژول تقسیم دو عدد را دریافت می‌کند و بعد از انجام عملیات لازم بر روی ورودی داده شده توسط ماژول ورودی (بدست آوردن توان و عدد داده شده در بیت های ورودی)، محاسبات مربوطه به عملیات متناظر را آغاز می‌کند.

۳. **ماژول خروجی**

این ماژول نیز پاسخ حاصل عملیات های تقسیم و جذر را دریافت می‌کند و آنها را به فرمت مورد پسند خروجی در می آورد. بعبارتی دیگر خروجی حاصله از ماژول های عملیاتی را بصورت استاندارد اعشاری در ۶۴ بیت به نمایش می‌گذارد.

[1] De-normal



شکل ۲ : شکل کلی مدار

۱ـ۳ـ۲ مدل طلایی

همانطور که گفته شده از کتابخانه mpmath بعنوان مدل طلایی در این پروژه استفاده شده است. این کتابخانه دارای بخش های متعدد محاسباتی ریاضی برای اعداد اعشاری با اعشار نامحدود است. این کتابخانه از داده ها با نوع های حقیقی و مختلط اعشاری پشتیبانی می کند. این داده ها را با شی هایی با نوع mpf[1] و mpc[2] به ترتیب بازمی‌گرداند. هم چنین این کتابخانه ماتریس ها را نیز پشتیبانی میکند. مقادیر خاص نیز با دو نوع NaN[3] و INF[4] مشخص می‌شوند.

برپا کردن[5] کتابخانه در پایتون

برای این کار بایستی ابتدا در فایل پایتون موردنظر تمامی کتابخانه های mpmath را شامل کنیم[6] و سپس در رابط کاربری تحت فرمان از تابع sqrt برای جذر و از تقسیم معمولی برای تقسیم استفاده کنیم. در این حالات اگر اعداد ما حقیقی باشند، خروجی نمایش داده شده یک mpf است که بایستی

---

[1] Mpmath floating point
[2] Mpmath complex
[3] Not a number
[4] Infinite
[5] Setup
[6] Include

مقدار ممیز شناور آن را از این شی خارج کنیم. برای این کار نیز کافی است حاصل خروجی تابع صدا زده شده را به تابع float دهیم تا خروجی مورد نظر ممیز شناور را بازگرداند.

توابع موردنیاز برای تقسیم و جذر

توابع مورد نیاز برای محاسبه جذر در فایل *libintmath.py* موجود است که با استفاده تابع *isqrt_small_python* و تابع *isqrt_fast_python* عملیات را انجام می دهد. در تابع اولی در صورتیکه عدد اعشاری از حدی کوچکتر باشد از جذر پایتون استفاده می کند و در غیر اینصورت با استفاده از عملیات های بیتی و الگوریتم نیوتن محاسبه جذر را انجام می دهد. تابع دیگر که با سرعت بیشتر این عملیات را انجام می دهد در پیاده سازی از الگوریتم نیوتن استفاده می کند. تفاوت این دو مدل پیاده سازی این است که در تابع اولی با استفاده از حلقه while شروع به پیمایش و نزدیک شدن به عددی می کند که حاصل جذر می باشد ولی در تابع دومی با استفاده از تابع *giant_step* که در همین فایل پیاده سازی شده، با سرعت بیشتری به سمت عدد موردنیاز حرکت می کنیم. لذا الگوریتم اصلی پیاده سازی جذر در این مدل طلایی، الگوریتم نیوتن است.

هم چنین برای پیاده سازی عملیات تقسیم در فایل *libmpf.py* تابع *mpf_div* این عملیات را انجام میدهد. در این تابع نیز عملیات بیتی اعمال شده و بعد از بدست آوردن حاصل برای جواب از تابع *normalize* استفاده شده است که به ترتیب علامت و خارج قسمت و باقی مانده را بصورت ممیز شناور بدست آورده و این تابع یک چندتایی[1] نرمال شده را به عنوان جواب نهایی باز می گرداند.

توابع ذکر شده در بالا دارای یک سری توابع رابط هستند که به وسیله آنها به رابط کاربری خط فرمان[2] متصل میشود. این توابع عملیات های چک کردن صحیح بودن فرمت عدد و تقسیم بخش های مختلف آن را در دستور کار خود دارند.

[1] Tuple

[2] Command line interface

۱ـ۴ پایهٔ ریاضی

در ۳ مورد بعد، اعمال را با نتیجه عدد صحیح فرض کنید. علت آن است که در دنیای دیجیتال، نمایش اعداد صحیح ساده‌تر است و در صورت نیاز به اعداد حقیقی، از پروتکل‌هایی استفاده می‌کنیم که اعداد اعشاری را به صورت اعداد صحیح مدل کنند.

۱. اگر عددی باینری در رقم 2n ام (با شروع از یک) یا بالاتر خود مقدار یک داشته باشد، پس از اجرای الگوریتم جذر در رقم n ام یا بزرگتر از آن مقدار یک خواهد داشت.

اثبات: اگر عدد مورد نظر را به صورت جمع توان هایی از ۲ بنویسیم داریم:

$$input = \sum_{p=0}^{\log(input)} 2^p * input[p]$$

بنابراین اگر از عبارت بالا جذر بگیریم در کوچکترین حالت برای input فقط رقم 2n ام آن مقدار یک دارد و سایر ارقام صفر هستند؛ پس $input = 2^{2n-1}$ خواهد بود و با اعمال جذر، داریم $\sqrt{input} = \sqrt{2^{2n-1}} = \frac{2^n}{\sqrt{2}}$ که چون مقدار جذر ۲ از خود ۲ کمتر است ولی از یک بیشتر است، به معنای آن است که در رقم n ام خود مقدار یک دارد و نه n+1 ام. حال هر عدد بزرگتری از کمترین مقدار input را جذر بگیریم، مقدار آن از جذر کمترین مقدار input بیشتر است؛ پس در رقمی برابر یا بزرگتر از n مقدار یک دارد. در صورتی که عدد باینری در رقم 2n-1 ام خود (با شروع از یک) مقدار یک داشته باشد، به وضوح پس از اجرای الگوریتم جذر در رقم n ام مقدار یک خواهد داشت.

۲. اگر بزرگترین رقمی که عددی باینری در آن مقدار یک دارد در بیت 2n ام یا کمتر باشد، پس از اجرای الگوریتم جذر در رقم n ام یا کمتر از آن مقدار یک خواهد داشت.

اثبات: اگر عدد مورد نظر را به صورت جمع توان هایی از ۲ بنویسیم داریم:

$$input = \sum_{p=0}^{\log(input)} 2^p * input[p]$$

بنابراین اگر از عبارت بالا جذر بگیریم در بزرگترین حالت input، تمام رقم‌های 2n ام تا اول مقدار یک خواهند داشت. از طرفی می‌دانیم که:

$$2^{2n} - 1 = 2^{2n-1} + 2^{2n-2} + \cdots + 2 + 1$$

پس input در بزرگترین حالت خود برابر $2^{2n} - 1$ خواهد بود و به وضوح مقدار آن از 2^{2n} کمتر است. میدانیم جذر 2^{2n} کوچکترین عددی است که در رقم n+1 ام خود مقدار یک دارد پس $\sqrt{input}$ در رقم n ام یا کمتر مقدار یک خواهد داشت. در صورتی که عدد باینری در رقم 2n-1 ام خود (با شروع از یک) مقدار یک داشته باشد، به وضوح پس از اجرای الگوریتم جذر در رقم n ام مقدار یک خواهد داشت.

۳. اگر عددی در رقم 2n ام خود مقدار یک داشته باشد و همه ارقام بزرگتر از 2n صفر باشند، جذر آن در رقم n ام خود مقدار یک دارد.

اثبات: با در نظر گرفتن موارد اثبات شده در قسمت یک و دو، درمی‌یابیم که اشتراک این دو بخش رقم n ام است. بنابرین عبارت فوق ثابت می‌شود.

۴. اگر اپراتور صحیح L (اپراتوری که مقدار خروجی آن عددی صحیح است) داشته باشیم و بخواهیم به کمک آن اپراتور حقیقی S (خروجی آن عدد حقیقی است) را پیاده سازی کنیم، داریم:

$$S(input) = L(input * 2^k) * 2^r$$

که r و k به دقتی که میخواهیم تبدیل انجام شود بستگی دارد. می‌توان عبارت فوق را ثابت کرد اما به دلیل پیچیدگی زیاد این عمل، در اینجا فقط این عبارت را برای اپراتور های تقسیم و جذر ثابت می‌کنیم.

اثبات برای جذر

فرض کنید بخش صحیح جذر عدد ورودی باینری input برابر inp_z باشد و قسمت بعد از ممیز آن نیز inp_f باشد. فرض کنید بخواهیم با دقت m رقم اعشار را محاسبه کنیم. در این صورت داریم:

$$\sqrt{input * 2^{2m}} = inp_z.inp_f * 2^m$$

این عبارت به این معناست که قسمت صحیح حاصل فوق، شامل m رقم پر ارزش inp_f نیز خواهد بود. با این تفاسیر، اگر اپراتور L جذر صحیح عدد $input * 2^{2m}$ را محاسبه کند، m رقم از inp_f را نیز محاسبه خواهد کرد. بنابراین می‌توان با دقت m رقم، ارقام قسمت اعشاری را نیز محاسبه کرد. پس اپراتور S کافیست که حاصل $L(input * 2^{2m})$ را بر 2^m تقسیم کند تا m رقم اعشاری به سرجای خود برگردند.

اثبات برای تقسیم

فرض کنید در تقسیم عدد input1 به عدد input2 قسمت صحیح جواب برابر res_z و قسمت بعد از ممیز برابر res_f باشد. فرض کنید بخواهیم با دقت m رقم اعشار را محاسبه نمائیم. در این صورت داریم:

$$\frac{input1 * 2^m}{input2} = res_z.res_f * 2^m$$

این عبارت به این معناست که قسمت صحیح حاصل فوق، شامل m رقم پر ارزش res_f نیز خواهد بود. با این تفاسیر، اگر اپراتور L تقسیم صحیح عدد $input1 * 2^m$ بر input2 را محاسبه کند، m رقم از res_f را نیز محاسبه خواهد کرد. بنابرین میتوان با دقت m رقم، ارقام قسمت اعشاری را نیز محاسبه کرد. پس اپراتور S کافیست که حاصل $L(input1 * 2^m, input2)$ را بر 2^m تقسیم کند تا m رقم اعشاری به سرجای خود برگردند.

۱ـ۴ـ۱ عملیات تقسیم با استفاده از الگوریتم Long Division

در ریاضیات روش long division به عنوان روشی استاندارد برای انجام تقسیم اعداد بر یک‌دیگر کاربرد دارد و به اندازه‌ای ساده است که در محاسبات دستی نیز مورد استفاده قرار می‌گیرد. در این روش تقیسم به مراحل ساده‌تری شامل تفریق و جمع تبدیل می‌گردد اما بر خلاف روش‌های دیگر تقسیم به وسیله تفریق، هزینه آن از مرتبه $o((\log n)^2)$ است. برای دریافت اطلاعات بیشتر در باره این روش می‌توانید به این **لینک** مراجعه فرمایید.

۱ـ۴ـ۱ـ۱ مراحل اجرای الگوریتم Long Division

اگر بخواهیم عدد N را بر D تقیسیم کنیم به صورت زیر عمل می کنیم:

```
if D = 0 then error (Division By Zero)
Q := 0
R := 0
for i := n - 1 .. 0 do
    R := R << 1
    R(0) := N(i)
    if R ≥ D then
        R := R - D
        Q(i) := 1
    end
end
```

پس از کامل شدن اجرای الگوریتم فوق، حاصل خارج قسمت تقسیم در Q و مقدار باقی‌مانده در R قرار دارد.

۱ـ۴ـ۱ـ۲ بررسی پیچیدگی زمانی الگوریتم Long Division

اگر عدد N را بر M تقسیم کنیم، در این صورت تعداد بیت‌های لازم برای نمایش آن‌ها برابر log N و log M است. برای انجام هر تقسیم، چون از جمع کننده استفاده می‌شود

و بیت نقلی منتقل می‌شود، هزینه تفریق عدد N از M برابر o(log(max(M,N))) است. در هر حلقه از اجرای الگوریتم، یک بار تفریق رخ خواهد داد و الگوریتم در مجموع، به اندازه log M دفعه اجرا می‌شود. بنابرین هزینه اجرای الگوریتم فوق به صورت زیر است.

با فرض اینکه $M = o(n)$ و $N = o(n)$ بنابراین:

$$T(n) = o(logn) * o(logn) = o((log\, n)^2)$$

۱ـ۴ـ۱ـ۳ اثبات درستی الگوریتم Long Division

به منظور مشاهده جزئیات مربوط به اثبات ریاضی کامل مربوط به الگوریتم، می‌توانید به این **لینک** مراجعه فرمایید.

۱ـ۴ـ۱ـ۴ الگوریتم Long Division برای اعداد Floating Point

برای پیاده سازی الگوریتم تقسیم دو عدد به فرمت floating point به کمک long division کار را به دو بخش تقسیم می‌کنیم.

در بخش اول به محاسبه حاصل تقسیم بخش اعشاری به یکدیگر می‌پردازیم. الگوریتم long division فقط جواب قسمت صحیح (بعد از ممیز) تقسیم دو عدد بر یکدیگر را بر می‌گرداند. برای محاسبه قسمت اعشاری از روش اسثبات شده در بخش ۴ ام قسمت پیش‌نیازهای اثبات الگوریتم که توضیح داده شد، استفاده می‌کنیم. پس داریم:

اگر بخواهیم عملیات را روی دو عدد با دقت واحد باشند انجام دهیم، عددی را که تقسیم می‌شود را به اندازه ۲۴ واحد به چپ شیفت می‌دهیم. در این صورت ۲۴ رقم از قسمت اعشاری حاصل تقسیم نیز محاسبه خواهد شد. چون برای بازسازی و تبدیل دوباره عدد حاصل به فرمت دقت واحد به ۲۳ رقم اعشار نیاز است، به این شکل ارقام مورد نظر تولید خواهد شد.

اگر بخواهیم عملیات را روی دو عدد دقت مضاعف انجام دهیم، عددی را که تقسیم می‌شود را به اندازه ۵۳ واحد به چپ شیفت می‌دهیم. در این صورت ۵۳ رقم از قسمت اعشاری حاصل تقسیم نیز محاسبه خواهد شد. چون برای بازسازی و تبدیل دوباره عدد حاصل به فرمت دقت مضاعف به ۵۲ رقم اعشار نیاز است، به این شکل ارقام مورد نظر تولید خواهد شد.

در بخش دوم به محاسبه توان عدد حاصل از تقسیم می‌پردازیم. وقتی دو عدد به یکدیگر تقسیم شوند، توان‌های آن‌ها از هم کم خواهند شد و با مقداری ثابت جمع می‌شود تا به فرمت مورد نیاز برای اعداد floating point دریاید. در اینجا صرفا الگوریتم مورد استفاده و شیوه استفاده از آن را بیان کردیم. جزئیات پیاده سازی‌ها و نقاط مرزی در قسمت توضیح پیاده سازی مفصلا بیان خواهد شد.

۱ـ۴ـ۲ عملیات جذر با استفاده از الگوریتم Shifting nth Root

در ریاضیات ریشه n ام یک عدد از حل معادله $x^n = A$ به دست می‌آید و مقدار آن برابر $x = \sqrt[n]{A}$ است. در این روش، مقدار ریشه n ام یک عدد، به صورت رقم به رقم و هر رقم در یک مرحله محاسبه می‌شود و هنگامی که یک رقم محاسبه شود مقدار آن قطعی است و در محاسبات ارقام بعدی، مقدار آن عوض نخواهد شد. الگوریتم ذکر شده برای هر پایه (base) قابل محاسبه است و روش پیاده سازی آن برای هر پایه، با تغییر اندکی نسبت به پایه قبلی قابل انجام است. همچنین در شرایطی خاص، میتوان داده‌های سریال را نیز به وسیله این الگوریتم پیاده سازی کرد. دقت کنید این الگوریتم فقط قادر به محاسبه ریشه صحیح اعداد است. برای مثال در صورتی که عدد ۲۳ به عنوان ورودی داده شود مقدار خروجی آن برابر ۴ خواهد بود. در ادامه فرایند تبدیل این الگوریتم به الگوریتمی که ریشه را به صورت حقیقی محاسبه کند بیان خواهیم کرد. برای دریافت اطلاعات بیشتر راجع به این الگوریتم به این **لینک** مراجعه فرمایید.

۱ـ۴ـ۲ـ۱ مراحل اجرای الگوریتم Shifting nth Root

اگر بخواهیم ریشه دوم عدد num را محاسبه کنیم داریم:

```
res = 0
bit = 1 << N-2
while (bit > num) begin
     bit >>= 2;
end
while (bit != 0) begin
```

```
        if (num ≥ res + bit) begin
            num -= res + bit;
            res = (res ≫ 1) + bit;
        end
        else
            res ≫= 1;
        bit ≫= 2;
end
```

در پایان حاصل جذر در متغیر *res* قرار دارد.

۱ـ۴ـ۲ـ۲ بررسی پیچیدگی زمانی الگوریتم Shifting nth Root

اگر عدد N را بر M تقسیم کنیم، در این صورت تعداد بیت‌های لازم برای نمایش آن‌ها برابر log N و log M است. برای انجام هر جمع، چون از جمع کننده استفاده می‌شود و بیت نقلی منتقل می‌شود، هزینه جمع عدد N و M برابر o(log(max(M,N))) است. قسمت حلقه while به صورت کاملا combinational محاسبه می‌شود و هزینه آن به اندازه یک جمع کننده است (در قسمت مربوط به پیاده‌سازی راجع به این مورد بیشتر توضیح داده شده است). از طرفی در پیاده سازی های رایج این الگوریتم از جمع کننده برای انجام جمع استفاده می‌شود اما در پیاده سازی انجام شده در این پروژه، از مجموعه ای از گیت‌های or استفاده شده است که به صورت موازی با هزینه o(1) عمل می‌کنند. تعداد اجراهای حلقه اصلی الگوریتم o(log(bit)) است که در بدترین حالت ممکن o(log n) هزینه دارد. هزینه مقایسه دو عدد نیز o(log n) است. بنابرین هزینه الگوریتم به صورت زیر است:

با فرض اینکه $M = o(n)$ و $N = o(n)$ بنابراین:

$$T(n) = o(logn) * o(logn) = o((\log n)^2)$$

توجه: در لینک ارائه شده در قسمت قبل هزینه این الگوریتم با پیاده سازی نرم‌افزاری امده است و به همین دلیل با محاسبات فوق متفاوت است.

۱ـ۴ـ۲ـ۳ اثبات درستی الگوریتم

به منظور مشاهده جزئیات مربوط به اثبات ریاضی کامل مربوط به الگوریتم، می‌توانید به این **لینک** مراجعه فرمایید.

۱ـ۴ـ۲ـ۴ الگوریتم Shifting nth Root برای اعداد Floating Point

برای پیاده سازی الگوریتم جذر یک عدد به فرمت floating point به کمک shifting nth root کار را به دو بخش تقسیم می‌کنیم.

در بخش اول به محاسبه حاصل جذر بخش اعشاری می‌پردازیم. الگوریتم shifting nth root فقط جواب قسمت صحیح (بعد از ممیز) جذر عدد را بر می‌گرداند. برای محاسبه قسمت اعشاری از روش اثبات شده در بخش ۴ ام قسمت پیش‌نیازهای اثبات الگوریتم که توضیح داده شد، استفاده می‌کنیم. پس داریم:

I. اگر بخواهیم عملیات را روی عدد با دقت واحد انجام دهیم، اگر توان عدد ورودی زوج باشد، عدد ورودی را به اندازه ۲۳ بیت و اگر توان فرد باشد، به اندازه ۲٤ بیت به چپ شیفت می‌دهیم. به این ترتیب در ازای هر دو حالت زوج یا فرد بودن توان، تعداد ۲۳ بیت رقم بعد از اعشار برای حاصل جذر محاسبه می‌شود. برای بازسازی عدد به صورت دقت واحد، به ۲۳ بیت اعشاری نیاز است که داریم؛ پس می‌توان عدد را بازسازی کرد.

II. اگر بخواهیم عملیات را روی عدد با دقت مضاعف انجام دهیم، اگر توان عدد ورودی زوج باشد، عدد ورودی را به اندازه ۵۲ بیت و اگر توان فرد باشد، به اندازه ۵۳ بیت به چپ شیفت می‌دهیم. به این ترتیب در ازای هر دو حالت زوج یا فرد بودن توان، تعداد ۵۲ بیت رقم بعد از اعشار برای حاصل جذر محاسبه می‌شود. برای بازسازی عدد به صورت دقت واحد، به ۵۲ بیت اعشاری نیاز است که داریم؛ پس می‌توان عدد را بازسازی کرد.

در بخش دوم به محاسبه توان عدد حاصل از جذر می‌پردازیم. اگر توان عدد زوج باشد به دو تقسیم می‌شود. اگر توان عدد فرد باشد، ابتدا با یک جمع شده و سپس به دو تقسیم می‌شود. در هر دو حالت، عدد حاصل از تقسیم بر دو با مقداری ثابت جمع می‌شود تا به فرمت مورد نیاز برای اعداد floating point دریاید. در اینجا صرفا الگوریتم مورد استفاده و شیوه استفاده از آن را بیان کردیم. جزئیات پیاده سازی‌ها و نقاط مرزی در **بخش ۲** مفصلا بیان شده است.

۲ توصیف معماری سیستم

Interface ۱_۲

Input Interface ۱_۱_۲

این ماژول کاملا combinational است و ورودی‌های خود را به شکلی واحد به قطعات با معنا (مانند توان و اعشار و علامت) تبدیل می‌کند تا سایر ماژول‌ها بتوانند راحت تر از ورودی داده شده استفاده کنند. همچنین این ماژول وضعیت اعداد ورودی و نوع آن‌هارا مشخص کرده و در اختیار سایر ماژول‌ها قرار می‌دهد.

ورودی

۱. *inpA,inpB*: عدد ورودی به فرمت IEEE754

۲. *operation*: نوع عملیاتی که باید روی داده ورودی انجام شود که مطابق جدول زیر است:

جدول ۱ : مقادیر مختلف Operation ها

نوع عملیات	ورودی
تقسیم با دقت واحد	00
جذر با دقت واحد	01
تقسیم با دقت مضاعف	10
جذر با دقت مضاعف	11

خروجی

I. *flagsA,flagsB*:

برای ساخت سیگنال های فوق از جدول و تعاریف زیر کمک گرفتیم:

a. De-normal: اگر بیت‌های 23 تا 30 برای دقت واحد و بیت‌های 52 تا 62 برای دقت مضاعف صفر باشد.
b. Zero: اگر بیت های 0 تا 30 برای دقت واحد و بیت‌های 0 تا 62 برای دقت مضاعف صفر باشد.
c. Inf: برای دقت واحد اگر بیت های 0 تا 22 صفر و بیت های 23 تا 30 یک باشد. برای دقت مضاعف اگر بیت‌های 0 تا 51 برابر صفر و بیت‌های 52 تا 62 برابر یک باشد.
d. Nan: برای دقت واحد اگر بیت های 0 تا 22 صفر نباشد و بیت های 23 تا 30 یک باشد. برای دقت مضاعف اگر بیت‌های 0 تا 51 برابر صفر نباشد و بیت‌های 52 تا 62 برابر یک باشد.
e. Normal: ورودی که هیچ یک از شرایط بالا را نداشته باشد نرمال است.

جدول ۲ : مقادیر مختلف Flag ها

وضعیت	flagsA,flagsB
De-normal	000
Zero	001
Inf	010
Nan	011
Normal	100

II. *signA,signB*:
a. اگر دقت عملیات واحد باشد: در این صورت این مقادیر برابر بیت 31 از *inpA* و *inpB* است.
b. اگر دقت عملیات مضاعف باشد: در این صورت این مقادیر برابر بیت 63 از *inpA* و *inpB* است.

III. *expA,expB*:
a. اگر دقت عملیات واحد باشد: در این صورت این مقادیر برابر بیت 23 تا 30 از *inpA* و *inpB* است.
b. اگر دقت عملیات مضاعف باشد: در این صورت این مقادیر برابر بیت های 52 تا 62 از *inpA* و *inpB* است.

IV. $outA, outB$:

a. اگر دقت عملیات واحد باشد: در اینصورت این مقادیر برابر با بیت های 0 تا 22 از $inpA$ و $inpB$ است که در اینتهای آن یک بیت یک قرار داده شده است (FPU طراحی شده فقط اعداد نرمال را پشتیبانی می‌کند).
b. اگر دقت عملیات مضاعف باشد: در اینصورت این مقادیر برابر با بیت های 0 تا 51 از $inpA$ و $inpB$ است که در انتهای آن یک بیت یک قرار داده شده است (FPU طراحی شده فقط اعداد نرمال را پشتیبانی می‌کند).

۲ـ۱ـ۲ Output Interface

این ماژول کاملا combinational است که ورودی های خود را به شکلی کنار یکدیگر قرار می‌دهد که به فرمت IEEE754 باشد و دقت واحد و مضاعف را پشتیبانی نماید. این ماژول خروجی‌های ماژول جذرگیرنده و تقسیم‌کننده را که به وسیله مالتی‌پلکسر انتخاب شده را دریافت می‌نماید.

ورودی

۱. $intA$: مقدار اعشار ساخته شده توسط ماژول‌های تقسیم یا جذرکننده است.
۲. $expA$: توان ساخته شده توسط ماژول‌های تقسیم یا جذرکننده است.
۳. $signA$: علامت ساخته شده توسط ماژول‌های تقسیم یا جذرکننده است.
۴. $mode$: نوع عملیات کنار هم قرار دادن داده‌ها به صورت دقت واحد یا مضاعف است.

خروجی

$outA$:

I. اگر دقت واحد باشد: در این صورت بیت‌های 0 تا 22 از اعشار ورودی را به بیت‌های 0 تا 7 از توان ورودی وصل کرده و بعد از آن بیت علامت را قرار می‌دهد و سایر بیت‌های بعدی را صفر می‌گذارد. در این حالت فقط بیت های 0 تا 31 مقدار خروجی شامل مقادیر ارزشمند هستند.
II. اگر دقت مضاعف باشد: در این صورت بیت‌های 0 تا 51 از ورودی را به توان ورودی وصل کرده و بعد از آن بیت علامت را قرار می‌دهد. در این حالت از تمام بیت های خروجی استفاده خواهد شد.

۲ـ۲ ماژول تقسیم

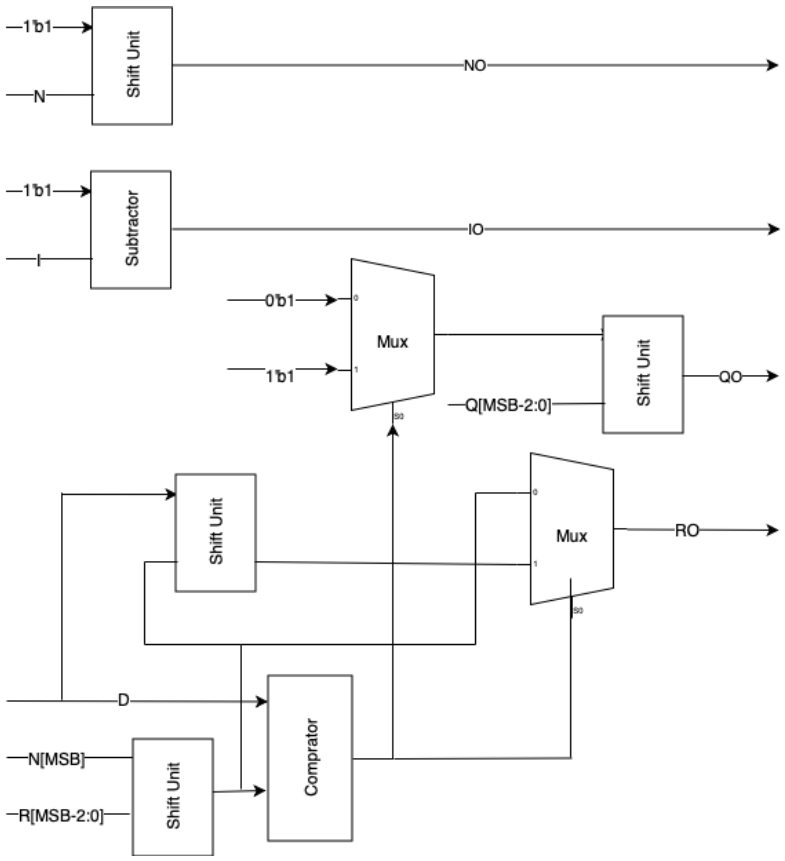
۲ـ۲ـ۱ ماژول div_unit

این ماژول، وظیفه پیاده سازی الگوریتم شرح داده شده در **بخش ۱ـ۴ـ۱ـ۱** را بر عهده دارد.

یک نکته پیاده سازی در اینجا مهم و قابل توجه می باشد؛ در وریلاگ N(i) و Q(i) وجود ندارند. اما با توجه به الگوریتم توضیح داده شده، میدانیم که مقدار اولیه i برابر است با تعداد بیت های مقسوم و بعد از هربار اجرای این کد، یکی از آن کم می شود. پس می توان به جای N(i)، هربار از بیت پر ارزش آن استفاده کرد و N را یک واحد به چپ شیفت داد. برای پیاده سازی Q(i) نیز، می توانیم هربار Q را به چپ شیفت بدهیم.

اگر R<D آنگاه بیت کم ارزش آن را ۰ و در غیر این صورت ۱ می گذاریم.

همچنین یک ورودی به عنوان i به این ماژول می دهیم. در صورتی که i به ۰ برسد، عملیات به اتمام می رسد و دیگر این محاسبات انجام نمی شود.



شکل ۳ : *div_unit*

۲ـ۲ـ۲ ماژول combinational_div

با توجه به توضیحات داده شده، برای کاهش کلاک مدار، در این طراحی، قسمت ترکیبی[1] به صورت چند مرحله ای طراحی شده است. برای رسیدن به محدودیت های صورت پروژه، قسمت ترکیبی ماژول تقسیم، توسط ۴ ماژول متوالی div_unit پیاده سازی شده است.

۲ـ۲ـ۳ ماژول divider

یک ماژول کاملا معنی دار طراحی شده است که به ازای گرفتن دو ورودی، باقی مانده و خارج قسمت را محاسبه می کند.

این ماژول شامل دو قسمت می باشد. قسمت کاملا ترکیبی که چهار بار حلقه ی for را به صورت متوالی انجام می دهد و قسمتی که مقادیر را رجیستر می کند.

[1] Combinational

در اینجا خروجی های combinational_div و مقسوم و مقسوم علیه رجیستر می شوند و سپس داده های رجیستر شده را به combinational_div می دهد.

تا زمانی که شمارنده به ۰ نرسیده است، سیگنال خروجی *ready* ، ۱ نمی شود.

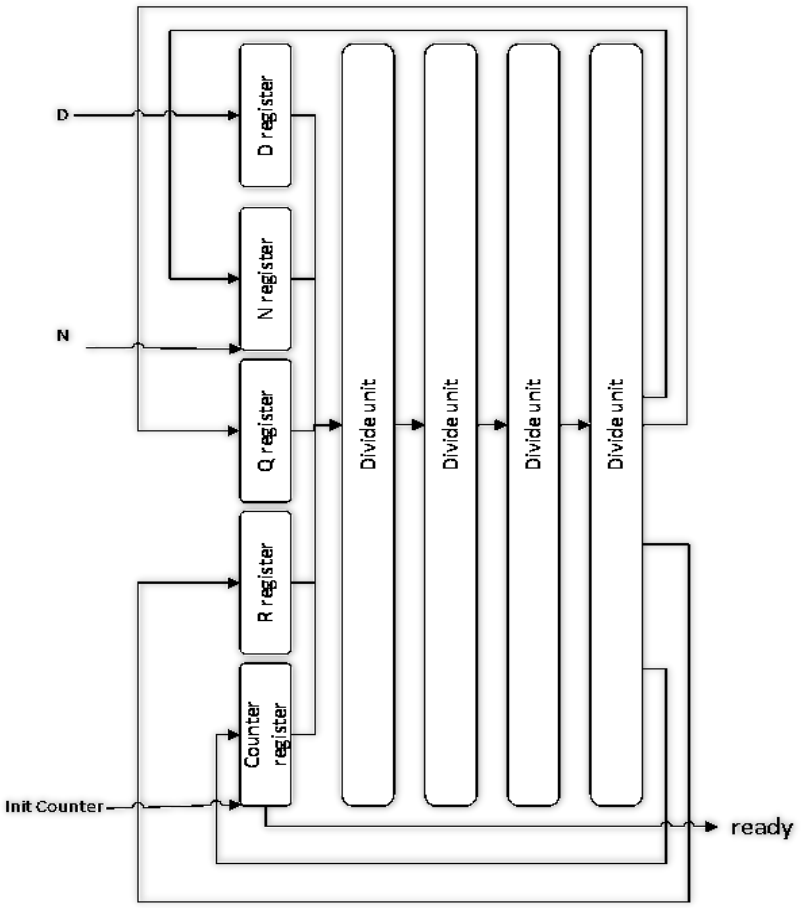
باقی مانده و خارج قسمت به همراه یک سیگنال *ready* خروجی های این ماژول می باشند.

مقسوم و مقسوم علیه و *enable* و شمارنده نیز ورودی های آن می باشند.

شمارنده تعداد مراحل اجرای الگوریتم را کنترل می کند. برای مثال اگر ۳۰ باشد، در کد مطابق **الگوریتم**، i مقدار اولیه ۳۰ را می گیرد و هر بار یکی کم می شود و پس از ۰ شدن، سیگنال خروجی *ready*، ۱ می شود.

Resetting

در صورتی که سیگنال *reset* فعال شود، خارج قسمت و باقی مانده برابر ۰ و i برابر با تعداد اجرای مراحل الگوریتم (۴۸ برای دقت واحد و ۱۰۶ برای دقت مضاعف) خواهند شد.



شکل ۴ : قسمت های ترکیبی و ترتیبی ماژول تقسیم

۲ـ۲ـ۴ تست قسمت های تقسیم

برای تمام ماژول های نام‌برده به صورت جداگانه test bench های جامعی نوشته شده است. تلاش بر این بوده است تمام حالات ممکن در تست ها در نظر گرفته شود. تست های زیادی با مقادیر اتفاقی[1] انجام شده است.

همچنین حالاتی که ممکن است مشکل آفرین باشند نیز مورد توجه قرار گرفته شده است تا تمام ماژول های تقسیم بی نقص باشند. تمام آن ها خروجی ها را در فایل های csv[2] ذخیره می کنند.

۲ـ۲ـ۵ Wrapper تقسیم

دلایل استفاده

ماژول تقسیم صرفا دو ورودی می گیرد و نتیجه تقسیم را به ما می دهد. برای اینکه تقسیم، مخصوص عملیات تقسیم FPU بشود، یک wrapper برای آن قرار می دهیم.

در اینجا عملیاتی نظیر شیفت دادن و آماده کردن ورودی، محاسبه علامت، حالات خاص و ... انجام می شوند که در بخش های بعدی هرکدام شرح داده می شوند.

wrapper از دو قسمت کنترلر و مسیر داده تشکیل شده است.

۲ـ۲ـ۵ـ۱ مسیر داده

محاسبه علامت

می دانیم در تقسیم دو عدد بر هم، اگر دو علامت یکسان باشند نتیجه مثبت و در غیر این صورت منفی می باشد.

تحقق سخت افزاری آن، XOR کردن علامت دو ورودی می باشد.

آماده کردن ورودی

---

[1] Random

[2] Comma-separated values

طبق توضیحات داده شده پیرامون الگوریتم، مقسوم به تعدادی بیت، شیفت می شود.

همچنین برای اینکه دقت واحد و دقت مضاعف با یک ماژول بتوانند کار کنند، نیاز به مقسوم و مقسوم علیه به صورت ۱۰۶ بیتی (۵۲ بیت اعشار و ۱ بیت ۱ ضمنی و ۵۳ بیت نیز برای شیفت) داریم.

دقت واحد

در اینجا، ۲۴ بیت پر ارزش در سیم مقسوم، متعلق به مقسوم ورودی می باشد.

۲۴ بیت کم ارزش در سیم مقسوم علیه نیز متعلق به مقسوم علیه ورودی می باشد.

باقی بیت های دو سیم، ۰ می باشد.

دقت مضاعف

در اینجا، ۵۳ بیت پر ارزش در سیم مقسوم، متعلق به مقسوم ورودی می باشد.

۵۳ بیت کم ارزش در سیم مقسوم علیه نیز متعلق به مقسوم علیه ورودی می باشد.

باقی بیت های دو سیم، ۰ می باشد.

به طور کلی آماده کردن دو ورودی مقسوم و مقسوم علیه برای تقسیم در wrapper انجام می شود.

۲ـ۲ـ۶ تعداد مراحل اجرای الگوریتم تقسیم

این عدد که با نام *initCounter* در کد موجود می باشد، تعداد مراحل اجرای الگوریتم را نشان می دهد.

برای مثال، در حالت دقت واحد، مقسوم را ۲۴ بیت شیفت می دهیم. پس تعداد مراحل اجرای الگوریتم، باید ۴۸ بار باشد و *initCounter* را ۴۸ قرار می دهیم.

۲ـ۲ـ۷ محاسبه جواب نهایی تقسیم

ماژول تقسیم به ما خارج قسمت تقسیم را می دهد. حال ما از خارج قسمت، برای ساختن نتیجه تقسیم در FPU استفاده می کنیم. می دانیم برای دقت بیشتر، ما داده را ۲۴ بیت (در حالت دقت مضاعف) شیفت دادیم.

$$divide = \frac{\{1'b1, mantissa1\} * 2^{24}}{\{1'b1, mantissa2\}}$$

عبارت داخل {} به معنای الحاق می باشد.

mantissa1 قسمت اعشار مقسوم و mantissa2 قسمت اعشار مقسوم علیه می باشد.

حال دو حالت را در نظر می گیریم

۱. mantissa1 از mantissa2 بسیار کوچک تر باشد.

برای مثال:

$$divide = \frac{\{1'b1,23'b0\}*2^{24}}{\{1'b1,23'b1\}} = \frac{2^{23}*2^{24}}{2^{24}-1} = 2^{23} + e$$

این عدد را می توان با ۲۴ بیت نشان داد.

۲. mantissa1 از mantissa2 بسیار بزرگتر باشد.

برای مثال:

$$divide = \frac{\{1'b1,23'b1\}*2^{24}}{\{1'b1,23'b0\}} = \frac{2^{24}*(2^{24}-1)}{2^{23}} = 2^{25} - e$$

این عدد را می توان با ۲۵ بیت نشان داد.

همان طورکه مشاهده شد، همواره بیت ۲۴ یا ۲۵ ام ۱ می باشد.

حال اگر بیت ۲۵ ام یک بود، بیت های ۵۳ تا ۱ را انتخاب می کنیم و در غیر این صورت بیت های ۵۲ تا ۰ را.

برای حالت دقت مضاعف نیز به طور مشابه اثبات می شود. در حالت دقت مضاعف ملاک انتخاب، بیت ۵۴ و ۵۳ می باشد.

۲ـ۲ـ۸ محاسبه توان نهایی

برای محاسبه ی توان کل، می دانیم در تقسیم دو عدد داریم:

توان مخرج ـ توان صورت = توان کل

اما برای نمایش در فرمت IEEE 754 باید توان را به علاوه یک پایه کنیم که مقدار آن ۱۲۷ برای حالت دقت واحد و ۱۰۲۳ برای حالت دقت مضاعف می باشد. تنها نکته اینجاست که ما در قسمت محاسبه ی نتیجه ی نهایی، با توجه به دو بیت خاص، یا بیت های ۵۲ تا ۰ و یا بیت های ۵۳ تا ۱ را محاسبه کردیم.

بنابر این، برای خنثی کردن اثر تفاوت این دو حالت (آنی که ۵۳ تا ۱ را شامل می شود یک بیت در واقع از توان استفاده کرده است)، بر همان اساس دو بیت خاص که در قسمت محاسبه ی نتیجه نهایی استفاده کردیم، اینجا هم یا با ۱۲۶ یا با ۱۲۷ (برای حالت دقت مضاعف با ۱۰۲۳ یا ۱۰۲۲) جمع می کنیم.

۲ـ۲ـ۹ تعیین Overflow و Underflow

underflow زمانی رخ می دهد که توان نهایی از ۰ کمتر باشد (در حالت استاندارد IEEE754).

overflow زمانی رخ می دهد که توان نهایی بیشتر از ۲۵۵ برای حالت واحد و ۲۰۲۳ برای حالت مضاعف باشد.

برای تعیین این دو، ابتدا در قسمت محاسبه ی توان نهایی، 2'b0 را به بیت پر ارزش تمام سیم های مورد استفاده در آن اضافه می کنیم.

دلیل آن این است که اگر منفی شود، بیت پر ارزش ۱ می شود. (در وریلاگ تفاضل عدد بزرگتر از عدد کوچکتر باعث می شود بیت پر ارزش ۱ بشود) با توجه به محدوده ی توان ها (هر کدام ۱۲ بیت) امکان ندارد جمع آنها بتواند بیت ۱۴ ام را ۱ کند؛ پس ۱ شدن بیت پر ارزش قطعا به معنای underflow می باشد.

همچنین برای underflow، باید توان صورت کوچکتر از توان مخرج باشد. پس وجود این دو شرط در کنار هم، باعث فعال شدن سیگنال underflow می شود.

برای overflow نیز داریم:

۱ـ دقت مضاعف: اگر بیت ۱۲ ام ۱ باشد یعنی توان از 2^{11} بیشتر می باشد و overflow رخ می دهد.

برخی شروط را مانند اینکه تمام بیت ها ۱ نباشند (اگر باشند حالت خاص می شود) و اینکه underflow رخ نداده باشد نیز چک می کنیم.

۲ـ دقت واحد: اگر بیت ۹ ام ۱ باشد یعنی توان از 2^8 بیشتر می باشد و overflow رخ می دهد.

همچنین برای overflow، باید توان صورت بزرگتر از توان مخرج باشد. پس وجود این دو شرط در کنار هم، باعث فعال شدن سیگنال overflow می شود.

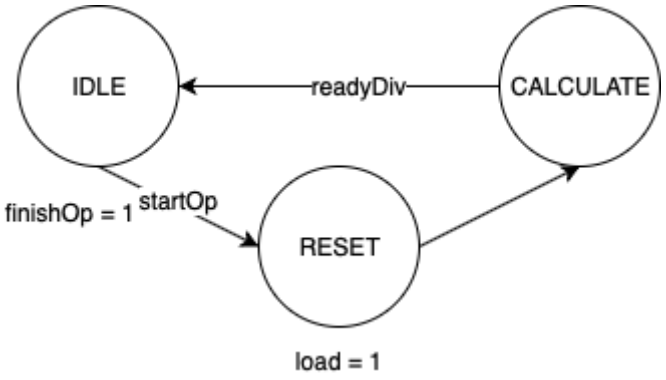
۲ـ۲ـ۱۰ واحد کنترل

دارای سه حالت می باشد:

IDLE: منتظر می ماند تا دستور شروع عملیات ارسال شود.

RESET: در این حالت، reset رجیستر ها فعال می شود و آن ها مقادیر مناسب را برای شروع کار می گیرند.

CALCULATE: در این حالت، محاسبه تقسیم انجام می شود. هنگامی که تقسیم عملیات خود را به اتمام برساند، یک سیگنال *readyDiv* ارسال می کند. تا زمانی که این سیگنال ارسال نشده است منتظر می مانیم. هنگامی که آن را دریافت کردیم، به حالت *IDLE* می رویم.



شکل ۵ : واحد کنترل ماژول تقسیم

۲ـ۲ـ۱۱ ماژول Wrapper

از اتصال دو ماژول کنترلر و مسیر داده حاصل می شود. بجز تمام ورودی های شرح داده شده مانند مقسوم و مقسوم علیه و خروجی هایی مانند حاصل تقسیم و حالات خاص، یک ورودی برای شروع عملیات و یک خروجی برای اتمام عملیات تقسیم دارد.

۲ـ۳ ماژول جذر

۲ـ۳ـ۱ پیاده‌سازی کلی

در این بخش از ماژول کلی به پیاده‌سازی جذر گیرنده‌ی ممیز شناور پرداخته‌ایم. این ماژول با مقادیر جداشده‌ی عدد ممیز شناور سر و کار دارد؛ یعنی ورودی‌های آن عددهای جداشده‌ی توان[1] و اعشار[2] هستند و خروجی‌های آن نیز توان و اعشار عدد خروجی هستند. ماژول‌های اصلی این بخش عبارتند از:

ماژول Input Wrapper

این ماژول ورودی‌ها را دریافت می‌کند و مقادیر توان و اعشار را برای ماژول‌های بعدی آماده می‌کند. توضیحات بیشتر در **بخش ۲ـ۳ـ۲** آمده است.

ماژول SQRT

این ماژول جذر یک عدد صحیح را می‌گیرد و یک عدد صحیح را به عنوان خروجی جذر می‌دهد؛ توضیحات بیشتر در **بخش ۲ـ۳ـ۵** آمده است.

ماژول Exponent Handler

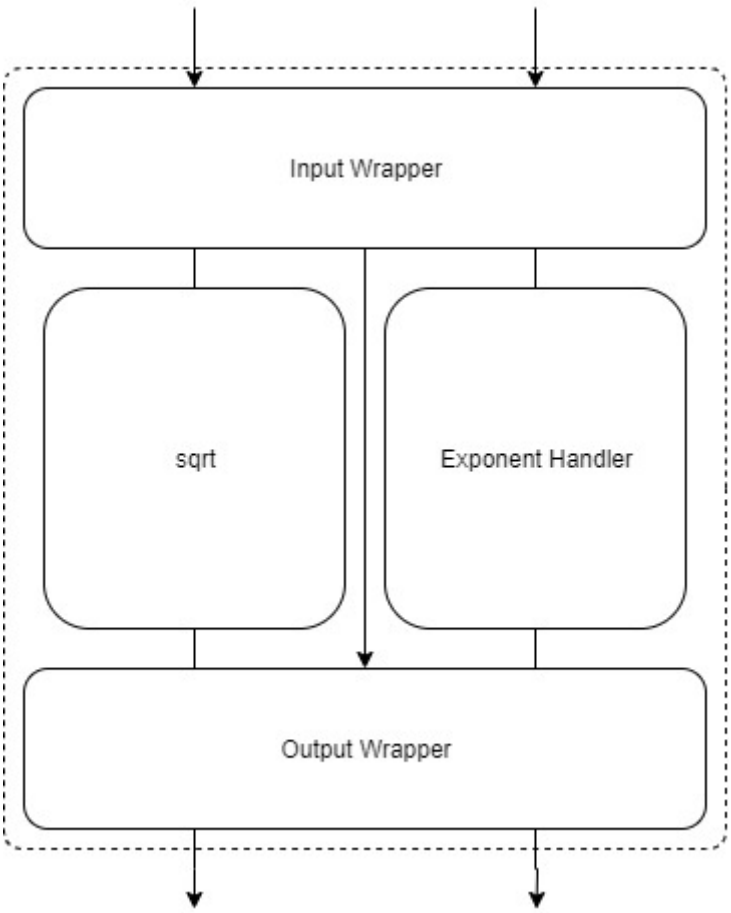
این ماژول توان ورودی را می‌گیرد و با توجه به نوع عدد(دقت واحد یا دقت مضاعف) توان عدد خروجی را به دست می‌دهد. توضیحات بیشتر در **بخش ۲ـ۳ـ۳** آمده است.

ماژول Output Wrapper

این ماژول، خروجی‌های ماژول‌های میانی را می‌گیرد و پس از انجام تغییراتی (برای نمونه خروجی‌های حالات خاص در این‌جا آماده می‌شود) آن را به عنوان خروجی می‌دهد. توضیحات بیشتر در بخش ۲ـ۳ـ۴ آمده است.

[1] Exponent

[2] Mantissa



شکل ۶: ماژول SQR T

به منظور کاهش مساحت مدار و با توجه به یکسان بودن عملکرد مدار برای هر دو نوع دقت واحد و دقت مضاعف، در این ماژول تنها یک مدار برای هر دو حالت پیاده‌سازی شده‌است. با توجه به این مورد گذرگاه‌های داخل ماژول باید به اندازه‌ی کافی بزرگ باشند تا داده‌های هر دو نوع در آن انتقال داده شوند. به همین منظور گذرگاه اعشار ۵۳ بیتی و گذرگاه توان ورودی ۱۱ بیتی است. از آنجا که این اندازه‌ها متناسب با داده‌های دقت مضاعف هستند، در حالت دقت واحد کم ارزش‌ترین بیت داده روی کم ارزش‌ترین بیت گذرگاه قرار می‌گیرد و به همین ترتیب سایر بیت‌ها روی گذرگاه قرار می‌گیرند.

به طور کلی برای انجام عملیات جذر، توان و اعشار را به طور جداگانه محاسبه می‌کنیم؛ این کار بدین صورت است که با استفاده از ماژول sqrt، جذر اعشار را می‌گیریم و به طور موازی توان خروجی را هم محاسبه می‌کنیم. با توجه به آنچه که در **قضیهٔ دوم اثبات ریاضی برای جذر** اثبات شده باید برای محاسبه بیت‌های خروجی، اعشار مقداری شیفت داده شود و به تبع آن توان نیز به گونه‌ای

محاسبه شود که شیفت دادن اعشار تاثیری در مقدار نهایی خروجی نداشته باشد. عملیات شیفت اعشار توسط input wrapper و عملیات محاسبه توان توسط ماژول exponent handler انجام می‌شود.

۲ـ۳ـ۲ ماژول Input Wrapper

این ماژول ورودی‌ها را از ماژول input interface دریافت می‌کند و با راه‌اندازی ماژول‌های میانی منجر به آغاز عملیات جذرگیری می‌شود. به منظور کاستن از پیچیدگی‌های ماژول‌های میانی، این ماژول وظیفه‌ی یکسان‌سازی داده‌ها را برعهده دارد به طوری که داده‌هایی که به ماژول‌های exponent handler و sqrt می‌روند مستقل از نوع داده (دقت واحد و دقت مضاعف) است. همچنین بخشی وظیفه‌ی رسیدگی به استثناها (مانند منفی بودن ورودی) هم توسط این ماژول انجام می‌گیرد.

ورودی‌های ماژول به شرح زیرند:

clk: کلاک کلی مدار

rst: ریست مدار

in_type: نوع عدد ورودی؛ در صورتی که عدد ورودی دقت واحد باشد مقدار این ورودی برابر با صفر و در صورتی که دقت مضاعف باشد، برابر با ۱ است.

start: سیگنالی که از کنترلر مرکزی به منظور آغاز فرایند جذر صادر می‌شود.

ready: سیگنالی که از ماژول sqrt صادر می‌شود و این پیام را می‌رساند که ماژول اشاره شده می‌تواند ورودی دریافت کند و در حال انجام عملیات نیست.

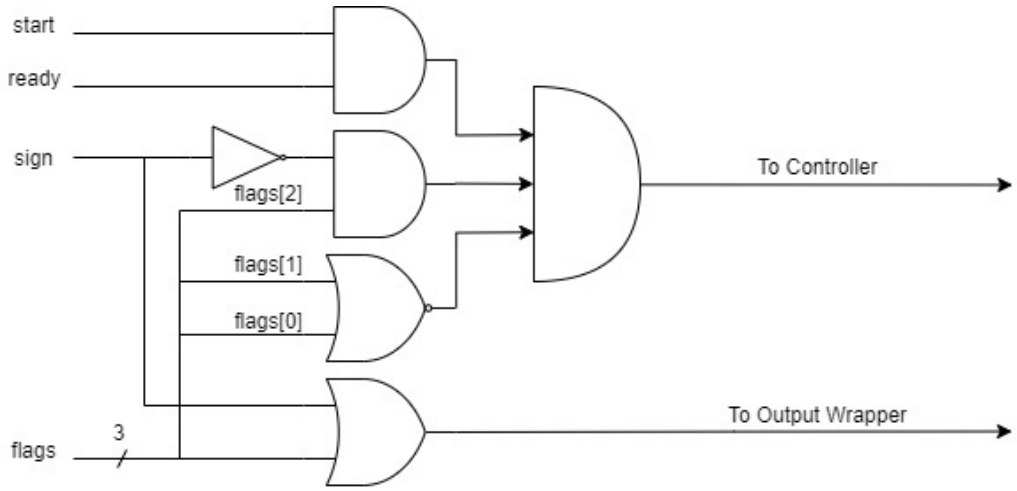
sign: علامت عدد ورودی

ld_reg: سیگنالی کنترلی است که به رجیسترهای توان و اعشار دستور نمونه‌گیری را می‌دهد.

in_exp: مقدار توان ورودی

in_flags: سیگنال‌هایی که موارد استثناء را مشخص می‌کنند. برای توضیحات دقیق‌تر به **جدول ۲** مراجعه شود.

in_mantissa: اعشار ورودی



شکل ۷: سیگنال های ورودی Input Wrapper برای ماژول جذر

عملکرد ماژول

دریافت توان و اعشار

در این بخش ماژول توان و اعشار ورودی را از input interface می‌گیرد و در رجیسترها ذخیره می‌کند. برای ذخیره کردن و استفاده‌ی اعشار با توجه به تفاوت‌های اعداد دقت واحد و دقت مضاعف باید به نحوی اعشار را ذخیره کنیم که پس از انجام عملیات جذر به دقت مورد نیاز برسیم که این کار در input wrapper با شیفت دادن اعشار انجام می‌شود. بدین منظور برای ۴ حالت زیر تعداد بیت‌هایی را که باید شیفت بدهیم، محاسبه می‌کنیم (در محاسبات زیر، M' خروجی ماژول sqrt و M ورودی این ماژول است و همچنین exp ورودی ماژول exponent handler و exp' خروجی آن است):

۱) عدد دقت واحد با توان زوج:

می‌دانیم در این حالت باید تعداد بیت‌های اعشار پس از عملیات جذر، ۲۳ بیت باشد.

$$Ans = \sqrt{1.M \times 2^{exp-127}} = \sqrt{1.M \times 2^{23} \times 2^{exp-127-23}}$$
$$= \sqrt{1M \times 2^{24} \times 2^{exp-127-23-24}}$$
$$= \sqrt{1M \times 2^{24}} \times 2^{\left(\frac{1}{2}\right)(exp-174)+127} = 1M' \times 2^{\left(\frac{1}{2}\right)(exp-174)+127}$$
$$= 1.M' \times 2^{\left(\frac{1}{2}\right)(exp-174)+23+127} = 1.M' \times 2^{\left(\frac{1}{2}\right)exp+63}$$

$$\rightarrow 1M' = \sqrt{1M \times 2^{24}}, exp' = \frac{exp}{2} + 63$$

از آنجا که اعداد de-normal توسط ماژول محاسبه نمی‌شوند، بنابراین همواره بزرگترین بیت عدد ورودی، بیت ۱ ضمنی است که در حالت دقت واحد بیت ۲۴ام عدد است؛ در نتیجه عدد $1M'$ نیز با توجه به **قضیهٔ دوم اثبات ریاضی برای جذر** ۲۴ بیتی می‌شود و بیت ۲۴ام آن نیز برابر با ۱ می‌گردد. پس دقت ۲۳ بیت دلخواه را به دست می‌دهد. دلیل افزوده شدن ۱۲۷ به توان هم این است در فرمت IEEE 754 Floating Point به توان واقعی عدد ۱۲۷ واحد افزوده می‌شود.

همانگونه که از اثبات دیده می‌شود، برای این که به ۲۳ بیت دقت در خروجی ماژول sqrt برسیم باید اعشار ورودی آن را ۲۴ بیت شیفت دهیم.

۲) عدد دقت واحد با توان فرد:

می‌دانیم در این حالت هم باید تعداد بیت‌های اعشار پس از عملیات جذر، ۲۳ بیت باشد.

$$\begin{aligned} Ans &= \sqrt{1.M \times 2^{exp-127}} = \sqrt{1.M \times 2^{23} \times 2^{exp-127-23}} \\ &= \sqrt{1M \times 2^{23} \times 2^{exp-127-23-23}} \\ &= \sqrt{1M \times 2^{23}} \times 2^{\left(\frac{1}{2}\right)(exp-173)+127} = 1M' \times 2^{\left(\frac{1}{2}\right)(exp-173)+127} \\ &= 1.M' \times 2^{\left(\frac{1}{2}\right)(exp-173)+23+127} = 1.M' \times 2^{\left(\frac{1}{2}\right)(exp+1)+63} \end{aligned}$$

$$\rightarrow 1M' = \sqrt{1M \times 2^{23}}\,, exp' = \frac{exp+1}{2} + 63$$

موارد استفاده شده در این اثبات مانند مورد ۱ است.

همانگونه که از اثبات دیده می‌شود، برای این که به ۲۳ بیت دقت در خروجی ماژول sqrt برسیم باید اعشار ورودی آن را ۲۳ بیت شیفت دهیم.

۳) عدد دقت مضاعف با توان زوج:

می‌دانیم در این حالت باید تعداد بیت‌های اعشار پس از عملیات جذر، ۵۲ بیت باشد.

$$\begin{aligned} Ans &= \sqrt{1.M \times 2^{exp-1023}} = \sqrt{1.M \times 2^{52} \times 2^{exp-1023-52}} \\ &= \sqrt{1M \times 2^{53} \times 2^{exp-127-52-53}} \\ &= \sqrt{1M \times 2^{53}} \times 2^{\left(\frac{1}{2}\right)(exp-1128)+1023} = 1M' \times 2^{\left(\frac{1}{2}\right)(exp-1128)+1023} \\ &= 1.M' \times 2^{\left(\frac{1}{2}\right)(exp-1128)+52+1023} = 1.M' \times 2^{\left(\frac{1}{2}\right)exp+511} \end{aligned}$$

$$\rightarrow 1M' = \sqrt{1M \times 2^{53}}\,, exp' = \frac{exp}{2} + 511$$

از آنجا که اعداد de-normal توسط ماژول محاسبه نمی‌شوند بنابراین همواره بزرگترین بیت عدد ورودی، بیت ۱ ضمنی است که در حالت دقت واحد بیت ۵۳ام عدد است؛ در نتیجه عدد $1M'$ نیز با توجه به **قضیهٔ دوم اثبات ریاضی برای جذر** ۵۳ بیتی می‌شود و بیت ۵۳ام آن نیز برابر با ۱ می‌گردد. پس دقت ۵۲ بیت دلخواه را به دست می‌دهد. دلیل افزوده شدن ۱۰۲۳ به توان هم این است در فرمت IEEE 754 Floating Point به توان واقعی عدد ۱۰۲۳ واحد افزوده می‌شود.

همانگونه که از اثبات دیده می‌شود، برای این که به ۵۲ بیت دقت در خروجی ماژول sqrt برسیم باید اعشار ورودی آن را ۵۳ بیت شیفت دهیم.

۴) عدد دقت مضاعف با توان فرد:

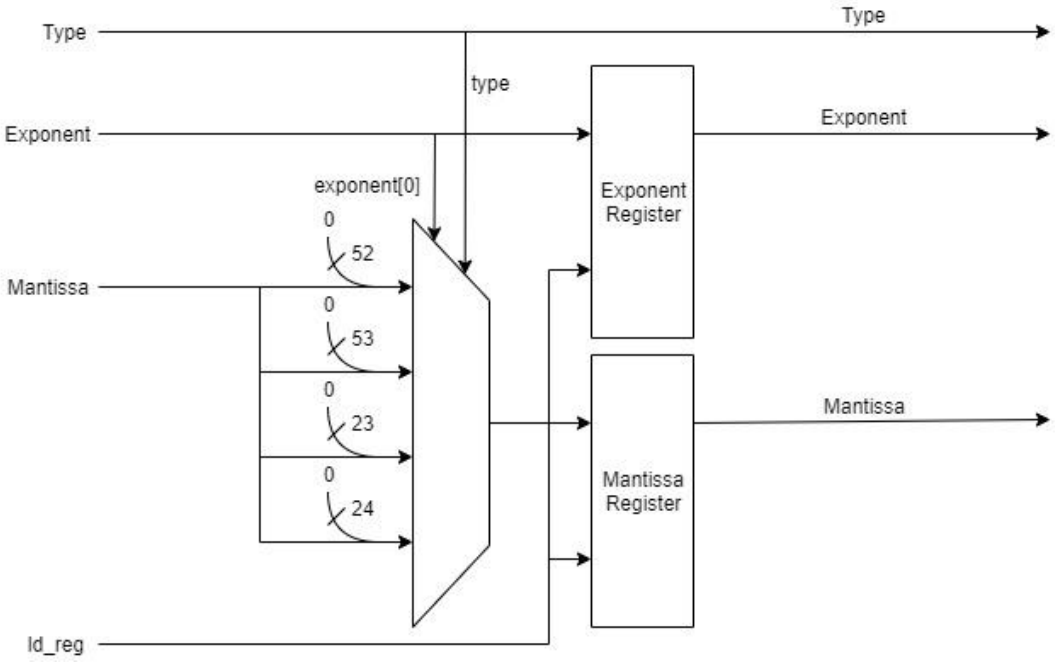
می‌دانیم در این حالت باید تعداد بیت‌های اعشار پس از عملیات جذر، ۵۲ بیت باشد.

$$Ans = \sqrt{1.M \times 2^{exp-1023}} = \sqrt{1.M \times 2^{52} \times 2^{exp-1023-52}}$$
$$= \sqrt{1M \times 2^{52} \times 2^{exp-127-52-52}}$$
$$= \sqrt{1M \times 2^{52}} \times 2^{\left(\frac{1}{2}\right)(exp-1127)+1023} = 1M' \times 2^{\left(\frac{1}{2}\right)(exp-1127)+1023}$$
$$= 1.M' \times 2^{\left(\frac{1}{2}\right)(exp-1127)+52+1023} = 1.M' \times 2^{\left(\frac{1}{2}\right)(exp+1)+511}$$

$$\rightarrow 1M' = \sqrt{1M \times 2^{52}}\,, exp' = \frac{exp+1}{2} + 511$$

موارد استفاده شده در این اثبات مانند مورد ۳ است.

همانگونه که از اثبات دیده می‌شود، برای این که به ۵۲ بیت دقت در خروجی ماژول sqrt برسیم باید اعشار ورودی آن را ۵۲ بیت شیفت دهیم.



شکل ۸ : دریافت توان و اعشار در ماژول جذر

راه اندازی ماژول اصلی و سیگنال‌های حالت استثناء

در این بخش با استفاده از سیگنال *run* ماژول sqrt را راه‌اندازی می‌کنیم. برای این که ماژول اصلی راه‌اندازی شود، باید شرایط زیر برقرار باشد:

۱. سیگنال *start* از کنترلر مرکزی صادر شده باشد.

۲. مقدار *sign* منفی نباشد.

۳. سیگنال *ready* از ماژول sqrt صادر شده باشد.

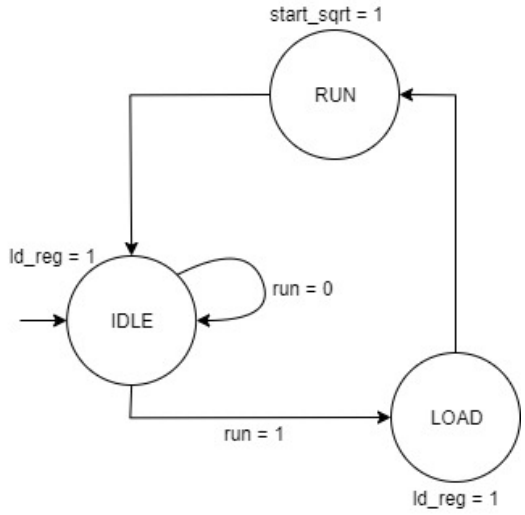
۴. مقدار ورودی normal باشد.

برای رعایت همه‌ی این موارد سیگنال‌ها به صورت مشخص شده در **شکل ۷** با هم ترکیب شده‌اند.

همچنین برای مشخص شدن این که عدد منفی است، سیگنال *sign* را بیت‌های *flags*، OR می‌کنیم تا در صورت منفی بودن ورودی، مقدار *flags*، ۱۱۱ می‌شود.

کنترلر

کنترلر این ماژول سیگنال‌های ذخیره کردن رجیستر و همچنین سیگنال آغاز عملیات جذر را راه‌اندازی می‌کند. برای آغاز عملیات باید سیگنال *run*، ۱ شود. سپس در اولین لبه بالا رونده کلاک ورودی‌ها در رجیستر ذخیره می‌شوند. از طرفی در کلاک بعدی سیگنال آغاز sqrt از سوی کنترلر صادر می‌شود.



شکل ۹ : واحد کنترل Input Wrapper ماژول جذر

۲ـ۳ـ۳ ماژول Exponent Handler

این ماژول با استفاده از سیگنال *type* مقدار توان خروجی را از توان ورودی محاسبه می‌کند و به صورت کاملا ترکیبی پیاده‌سازی شده است. ورودی‌های ماژول عبارتند از:

type: نوع عدد ورودی؛ در صورتی که عدد ورودی دقت واحد باشد مقدار این ورودی برابر با صفر و در صورتی که دقت مضاعف باشد ۱ است.

exp: توان ورودی

با توجه به اثبات‌های انجام شده در **بخش ۲ـ۳ـ۲** داریم (exp' خروجی ماژول و exp ورودی آن است):

۱) عدد دقت واحد با توان زوج

$$exp' = \frac{exp}{2} + 63$$

۲) عدد دقت واحد با توان فرد

$$exp' = \frac{exp+1}{2} + 63 = \frac{exp-1}{2} + 63 + 1$$

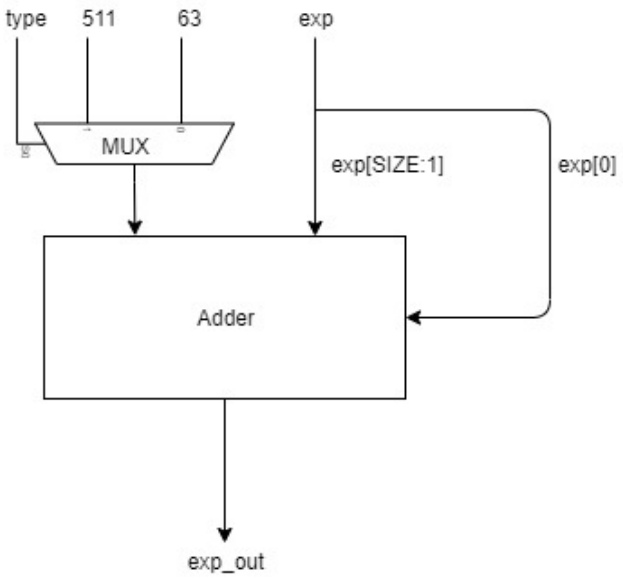
۳) عدد دقت مضاعف با توان زوج

$$exp' = \frac{exp}{2} + 511$$

۴) عدد دقت مضاعف با توان فرد

$$exp' = \frac{exp+1}{2} + 511 = \frac{exp-1}{2} + 511 + 1$$

در نتیجه مشاهده می‌شود که در حالاتی که توان فرد است یک واحد از آن کاسته می‌شود و سپس بر۲ تقسیم می‌شود و یک واحد نیز در نهایت به ورودی داده می‌شود. پس می‌توان آن را بدین صورت پیاده کرد که کم ارزش‌ترین بیت عدد را از آن جدا می‌کنیم و به عنوان carry in به ماژول می‌دهیم. این کار باعث می‌شود که هم carry in مورد نیاز ساخته شود و هم جدا کردن بیت کم ارزش باعث شیفت خوردن سایر بیت‌ها می‌گردد که همان نقش تقسیم عدد بر ۲ را ایفا می‌کند. همچنین از آنجا که در توان‌های زوج نیز کم ارزش‌ترین بیت، صفر است بنابراین این کار را برای توان‌های زوج نیز می‌توان انجام داد. پس برای پیاده‌سازی این بخش کافی است از یک مالتی پلکسر برای تعیین عدد ثابت جمع شونده (۶۳ برای دقت واحد و ۵۱۱ برای دقت مضاعف) استفاده شود و خروجی آن نیز با *exp* آماده شده جمع شود.



شکل ۱۰: ماژول جذر Exponent Handler

۲ـ۳ـ۴ ماژول Output Wrapper

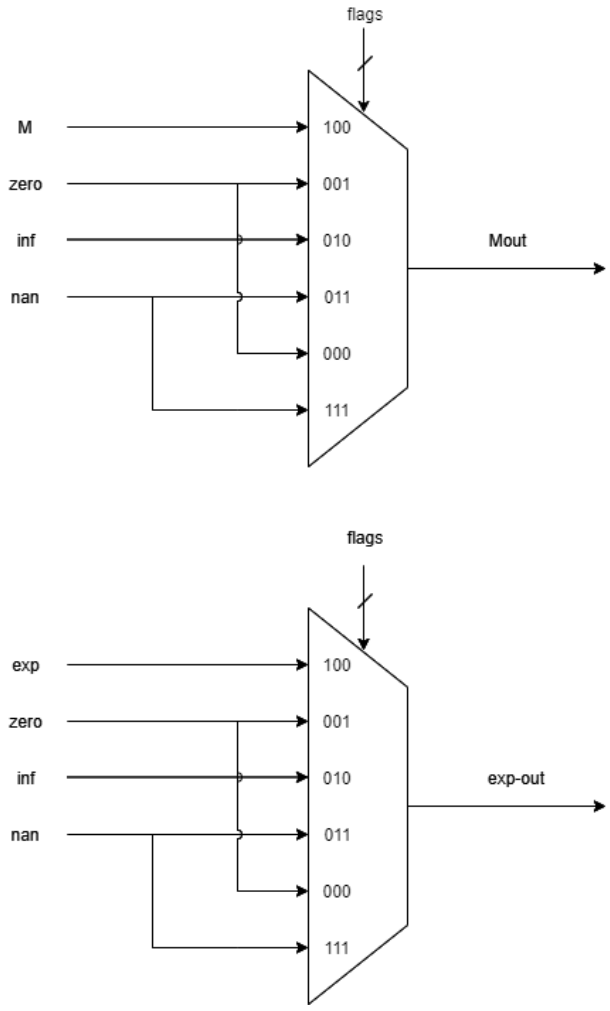
این ماژول با استفاده از سیگنال‌های *flags* به صورت کاملا ترکیبی به مدیریت استثناها می‌پردازد و مقادیر خروجی متناسب با هر *flags* را به عنوان خروجی می‌دهد. ورودی‌های این ماژول عبارتند از:

in_mantissa: اعشار ورودی

in_exp: توان ورودی

in_flags: سیگنال‌هایی که برای مدیریت استثناها استفاده می‌شود. مقادیر این سیگنال‌ها و مفاهیم آن در **جدول ۲** آمده است و مقادیری که به عنوان خروجی به ازای هر یک از آن‌ها صادر می‌شود در **بخش ۲ـ۴** آمده است.

برای پیاده‌سازی آن‌ها از مالتی پلکسرهایی برای توان و اعشار استفاده می‌شود که با توجه به مقادیر *flags* مقدار درست را بر روی خروجی قرار می‌دهد.



شکل ۱۱ : Output Wrapper ماژول جذر

۲ـ۳ـ۵ ماژول SQRT

این ماژول یک عدد صحیح را دریافت کرده و جذر آن را به صورت یک عدد صحیح به خروجی می‌دهد. در واقع عدد خروجی ماژول جذر بزرگترین عدد مربع کامل کوچک‌تر از ورودی است. برای نمونه به ازای ورودی ۵۱ در خروجی ۷ را بازمی‌گرداند و به ازای ورودی ۴۹ عدد ۷ را بازمی‌گراند. ورودی‌های آن به شرح زیرند:

clk: کلاک کلی مدار

rst: ریست مدار

start: سیگنالی است که ماژول را راه‌اندازی می‌کند.

num: عدد ورودی که باید از آن جذر گرفته شود.

خروجی این ماژول جذر عدد و سیگنال *ready* است که مشخص می‌کند خروجی آماده شده است و ماژول در حال محاسبه نیست. به طور کلی نحوه عملیات این ماژول بدین صورت است که در هر کلاک ۴ بیت از خروجی را محاسبه می‌کند و در رجیستر معادل ذخیره می‌کند. همچنین مقادیر مورد نیاز در هر مرحل را از مرحله قبل آماده می‌کند و رجیستر می‌کند. مسیرداده‌ی این ماژول چهار بخش عمده را شامل می‌شود که به شرح زیرند:

۱) ماژول first_one_finder

این ماژول بزرگ‌ترین توان ٤ کوچک‌تر از عدد ورودی را محاسبه می‌کند. توضیحات بیشتر در **بخش ۲ـ۳ـ۷** آمده است.

۲) مالتی پلکسرهای قبل از رجیسترهای ذخیره‌سازی

در این قسمت با استفاده از سیگنال کنترلی صادر شده از کنترلر مشخص می‌شود که کدام مورد باید در رجیسترها ذخیره شود. در صورتی که این سیگنال صفر باشد ورودی‌ها در رجیستر ذخیره می‌شوند و در صورتی که ۱ باشد خروجی مرحله قبلی فرایند جذر را ذخیره می‌کند.

۳) رجیسترهای ذخیره کننده

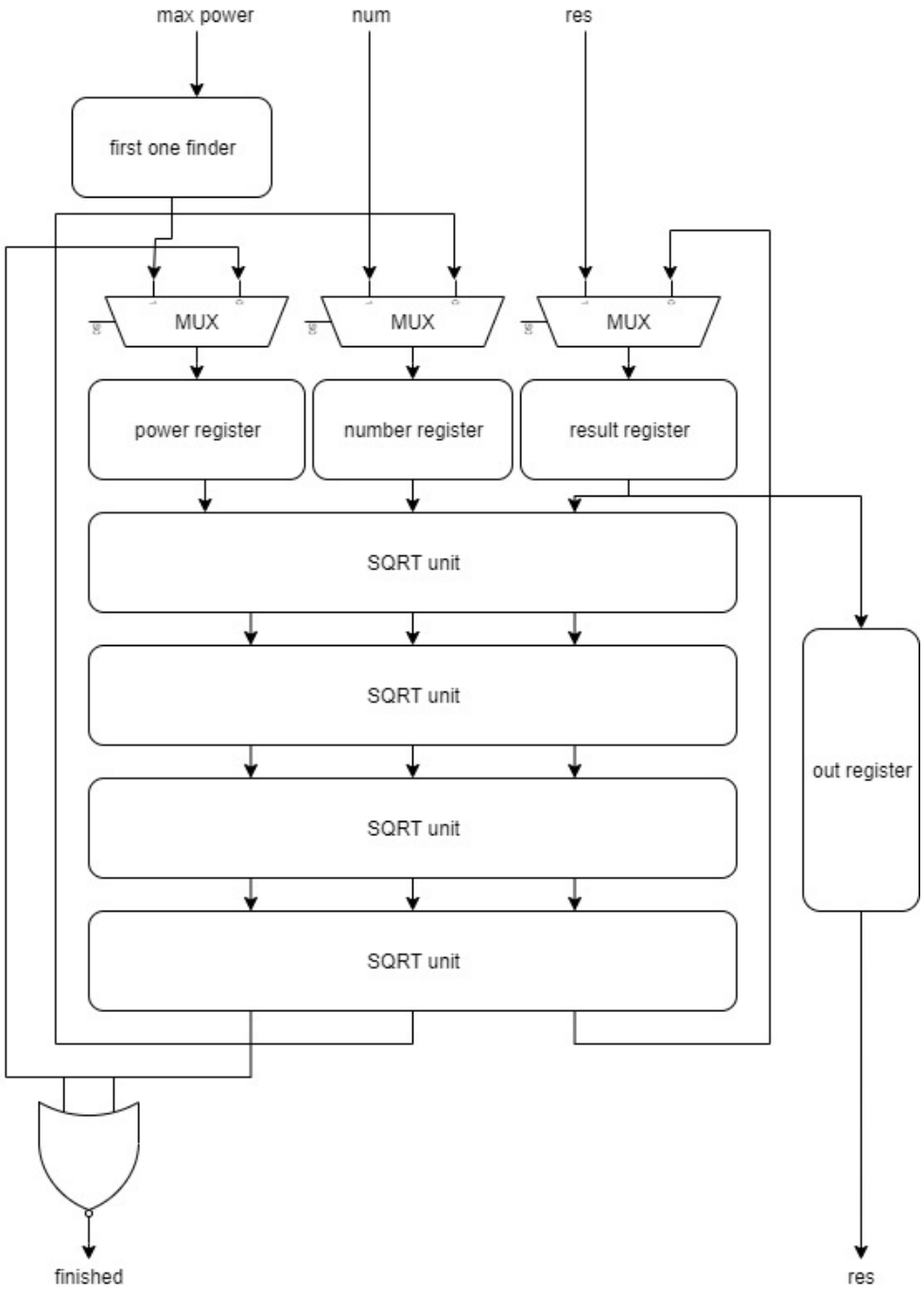
در این بخش سه مقدار ذخیره می‌شود؛ بیت‌های محاسبه شده از خروجی، مقدار باقیمانده از ورودی و همچنین بزرگترین توان ۴ که کوچک‌تر از عدد مورد نظر است.

با توجه به الگوریتم اثبات شده در **بخش ۱ـ۴ـ۲** نیاز به نگه‌داری این مقادیر داریم. همچنین یک رجیستر نیز برای نگه‌داری مقدار نهایی استفاده می‌شود.

۴) بدنه‌ی اصلی ماژول

این قسمت شامل ۴ ماژول sqrt_unit است که به صورت سری قرار گرفته‌اند. ماژول sqrt_unit عملیات جذرگیری را در پایین‌ترین سطح انجام میدهد و یک بیت از مقدار حاصل را به خروجی می‌دهد. جزییات بیشتر در **بخش ۲ـ۳ـ۸** آمده است.

این عملیات تا زمانی ادامه می‌یابد که مقدار *max_power4* (که خروجی آخرین ماژول sqrt_unit است) صفر شود. در این صورت تعداد بیت‌های محاسبه شده در خروجی برابر با دقت مورد نیاز ما خواهد شد. اگر در هر یک از ماژول‌های میانی sqrt_unit این مقدار صفر شود مقدار جذر در سایر ماژول‌های sqrt_unit تغییری نمی‌کند و به خروجی آخرین ماژول می‌رسد. در نهایت، پایان عملیات با استفاده از سیگنال *finished* به اطلاع کنترلر می‌رسد.

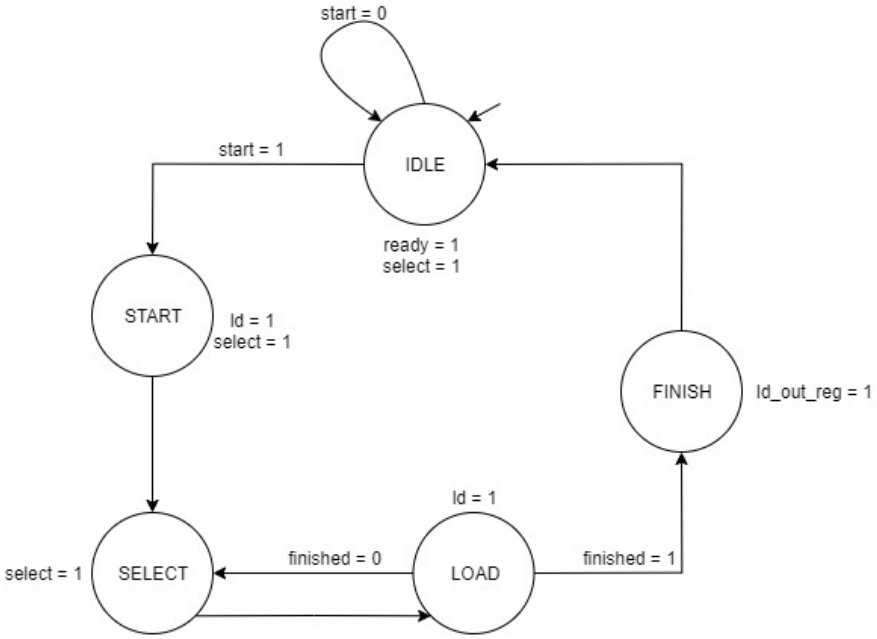


شکل ۱۲ : مسیر داده ماژول SQRT

۲ـ۳ـ۶ واحد کنترل

کنترلر ماژول سیگنال‌های سلکتور مالتی پلکسرها و همچنین دستور ذخیره کردن داده در رجیستر را صادر می‌کند. برای محاسبه جذر کنترلر بین حالات *LOAD* و *SELECT* جابجا می‌شود تا جایی که سیگنال *finished* از مسیرداده صادر شود که منجر به خاتمه عملیات می‌شود. این کار باعث

می‌شود که اتمام عملیات ماژول وابسته به سیگنال *finished* باشد که آن نیز وابسته به مقدار *max_power* است که در هر کلاک ۲ واحد شیفت می‌خورد. در نتیجه کلاک‌های مصرف شده در مدار با توجه به مقدار ورودی تعیین می‌شود و متناسب با اندازه داده ورودی است.

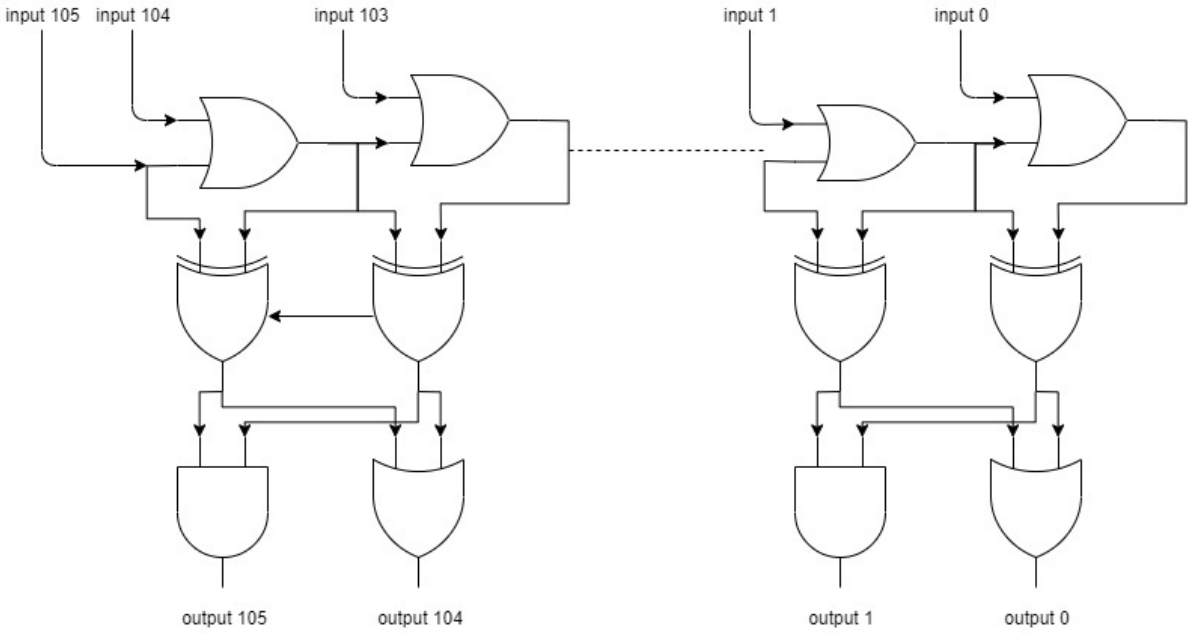


شکل ۱۳ : واحد کنترل ماژول جذر

۲ـ۳ـ۷ ماژول First One Finder

این ماژول یک ورودی می‌گیرد و در خروجی بزرگ‌ترین توان ۴ کوچک‌تر از عدد مورد نظر را می‌دهد. ورودی این ماژول تنها یک عدد است که میخواهیم بزرگ‌ترین توان ۴ را در آن بیابیم. می‌دانیم اگر بیت‌های عدد باینری را از بیت کم ارزش و از صفر شماره‌گذاری نماییم، آنگاه بیت‌هایی که توان‌های ۴ هستند در خانه‌های با شماره‌های فرد قرار دارند. پس برای پیاده کردن این ماژول ابتدا بزرگ‌ترین توان ۲ی آن را می‌یابیم. برای یافتن بزرگ‌ترین توان عدد، بیت‌ها را یکی یکی از سمت بیت پرارزش با هم OR می‌کنیم. در اینصورت هرجا که بیتی ۱ باشد از آنجا به بعد خروجی‌ها همه ۱ می‌شوند پس با دو به دو XOR کردن موارد می‌توان آن‌ها را صفر کرد و تنها بیتی که ۱ می‌ماند بزرگ‌ترین توان است.

حال اگر این بیت در جایگاه فرد بود آن را تغییر نمی‌دهیم و اگر در جایگاه زوج بود آن را صفر کرده و در جایگاه سمت چپ آن ۱ قرار می‌دهیم. برای این‌کار از دو گیت OR و AND به صورتی که در شکل زیر آمده است استفاده می‌کنیم.



شکل ۱۴ : ماژول *first_one_finder*

۲ـ۳ـ۸ ماژول sqrt_unit

این ماژول الگوریتم جذر را پیاده‌سازی می‌کند و به گونه‌ای پیاده شده است که قابلیت آبشاری کردن را داشته باشد. همچنین این ماژول هربار یک بیت از خروجی را محاسبه می‌کند. ورودی این ماژول به صورت زیر است:

prev_result : مقدار نتیجه‌ی ماژول قبلی

prev_num : عدد ورودی به ماژول

prev_biggest_power : بزرگ‌ترین عدد توان ۴ کوچک‌تر عدد ورودی

بخش‌های اصلی این ماژول در ادامه شرح داده شده است.

۲ـ۳ـ۸ـ۱ بدنه اصلی

با توجه به شبه کد داده شده در **بخش ۱ـ۴ـ۲** بدنه‌ی ماژول شامل یک مقایسه کننده و یک تفریق کننده است.

۲ـ۳ـ۸ـ۲ جمع کننده‌های res و bit

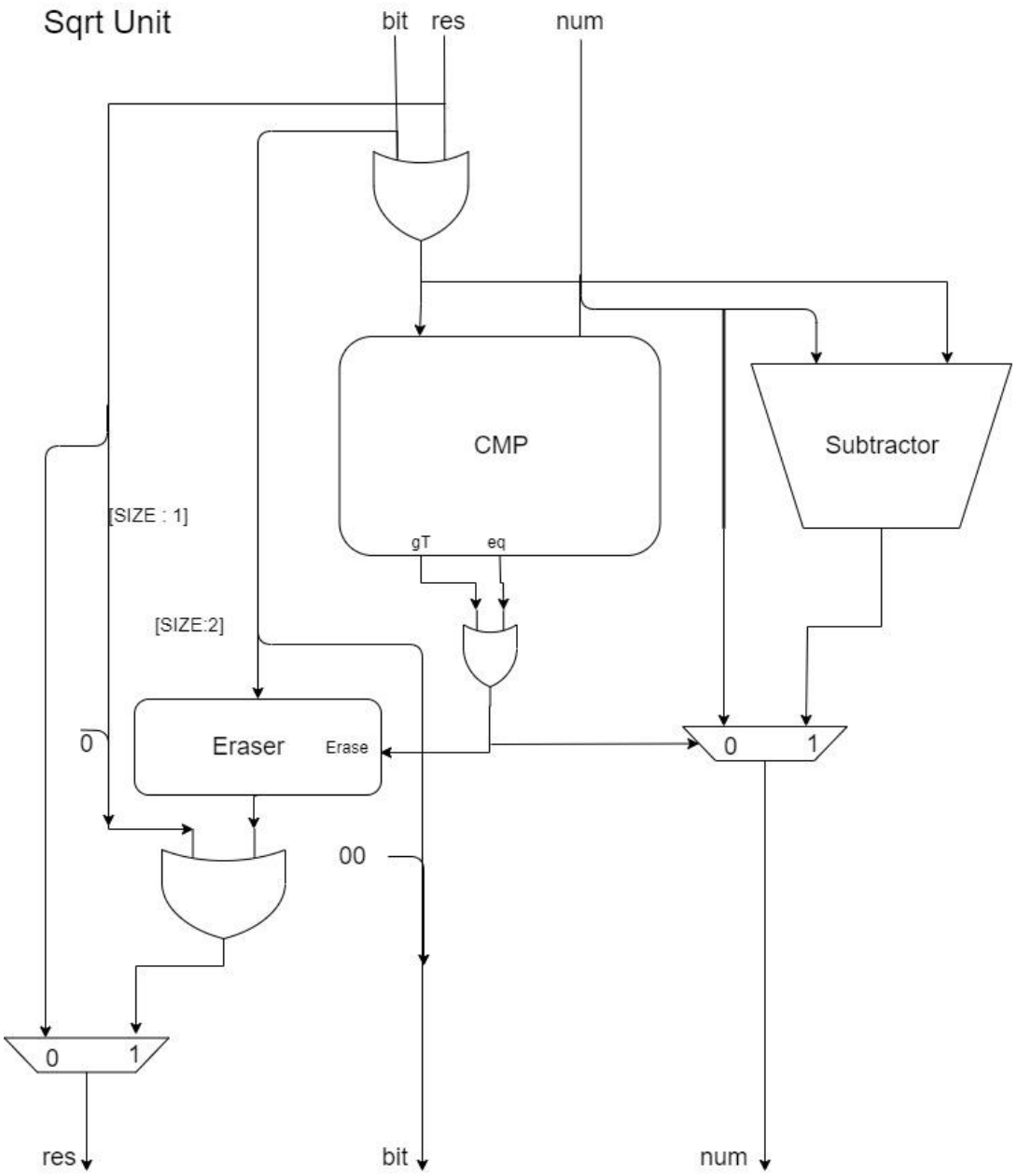
در هر جایی که عملیات جمع بین *res* و *bit* انجام شده است، به جای آن این دو عدد به صورت بیت به بیت با هم OR می‌شوند. زیرا عدد *bit* همواره در یک بیت مقدار ۱ دارد و در سایر بیت‌ها مقدارش صفر است (*bit* برابر با بزرگ‌ترین توان ۴ کوچک‌تر از عدد ورودی است). از طرفی در اولین باری که مقدار *res* عددی غیر صفر می‌شود، مقدارش با *bit* برابر است و از آن به بعد در هر بار تکرار حلقه مقدار *bit* دو بیت شیفت می‌خورد ولی مقدار *res* یک بیت شیفت می‌خورد. در نتیجه در هر جایی که این دو مقدار با هم جمع می‌شوند مقدار *bit* در جایی غیرصفر است که مقدار *res* در آنجا صفر است؛ در نتیجه در جمع بین بیت‌های این دو عدد carry ایجاد نمی‌شود و کافیست بیت‌ها دوبدو OR شوند.

۲ـ۳ـ۸ـ۳ ماژول eraser

این ماژول بدین صورت است است یک سیگنال *erase* دریافت می‌کند و اگر این سیگنال ۰ باشد آنگاه خروجی صفر می‌شود و اگر ۱ باشد ورودی را روی خروجی قرار می‌دهد.

۲ـ۳ـ۸ـ۴ سایر بخش ها

در این‌جا یک مالتی پلکسر استفاده می‌شود که براساس خروجی مقایسه‌کننده مقدار عدد ورودی یا حاصل تفریق *bit* + *res* از عدد ورودی را در خروجی ماژول قرار می‌دهد. همچنین مقادیر *bit* و *res* نیز شیفت داده می‌شوند.



شکل ۱۵ : ماژول sqrt_unit

۲ـ۴ رسیدگی به حالات استثناء

در جداول زیر تمام حالات خاص، و نتیجه آن نوشته شده است. جدول اول مربوط به تقسیم و جدول دوم مربوط به جذر است. کد وریلاگ متناظر با این جدول، برای رسیدگی به حالات خاص پیاده سازی شده است.

جدول ۳: حالت های خاص برای تقسیم

DIVIDEND	DIVISOR	RESULT
NaN	every thing	NaN
every thing	NaN	NaN
zero	zero	NaN
infinty	infinity	NaN
zero	not zero	zero
not infinity	infinity	zero
infinity	not infinity	infinity
not zero	zero	infnity

جدول ۴ : حالت های خاص برای جذر

INPUT	RESULT
NaN	NaN
infinity	infinity
zero	zero
negative	NaN
denormal	NaN

۲ـ۵ ماژول FPU

این ماژول تمام مدار را در بر می‌گیرد و عملیات‌های خواسته شده در سطح بالا توسط این ماژول انجام می‌شود. ورودی‌های آن به شرح زیرند:

clk: کلاک مدار

rst: ریست مدار

start: سیگنالی که آغاز عملیات را به اطلاع ماژول می‌رساند.

inpA & inpB: ورودی‌های مدار؛ در عملیات تقسیم، *inpA* مقسوم و *inpB* مقسوم علیه است و در عملیات جذر، *inpA* عددی است که جذر آن محاسبه می‌شود و *inpB* در جذر نادیده انگاشته می‌شود.

operation: مشخص می‌کند که عملیات خواسته شده چیست و همچنین تعیین کننده‌ی دقت واحد یا دقت مضاعف بودن عدد است.

مقدار خروجی در *res* قرار داده می‌شود و با استفاده از سیگنال *ready*، پایان عملیات و درست بودن مقدار روی خروجی اطلاع داده می‌شود.

مسیر داده این ماژول شامل ۴ قسمت اصلی می‌شود که به صورت زیر شرح داده شده است.

۲ـ۵ـ۱ ماژول Input Interface

ماژول input interface در ابتدای مسیر ورود داده‌ها به ماژول قرار دارد و به صورت کاملا ترکیبی طراحی شده است. در اینجا، داده های ورودی به input interface داده می‌شود تا فرمت مناسبی برای استفاده توسط ماژول های تقسیم و جذر ساخته بشود و همچنین در این بخش، توان، اعشار و علامت عدد از هم جدا می‌شوند و تا خروجی ماژول به صورت جداگانه پردازش می‌شوند. عملیات دیگری که در این قسمت انجام می‌گیرد، ایجاد سیگنال‌های *flags* است که برای مدیریت استثناها استفاده می‌شوند.

۲ـ۵ـ۲ ماژول‌های جذر و تقسیم

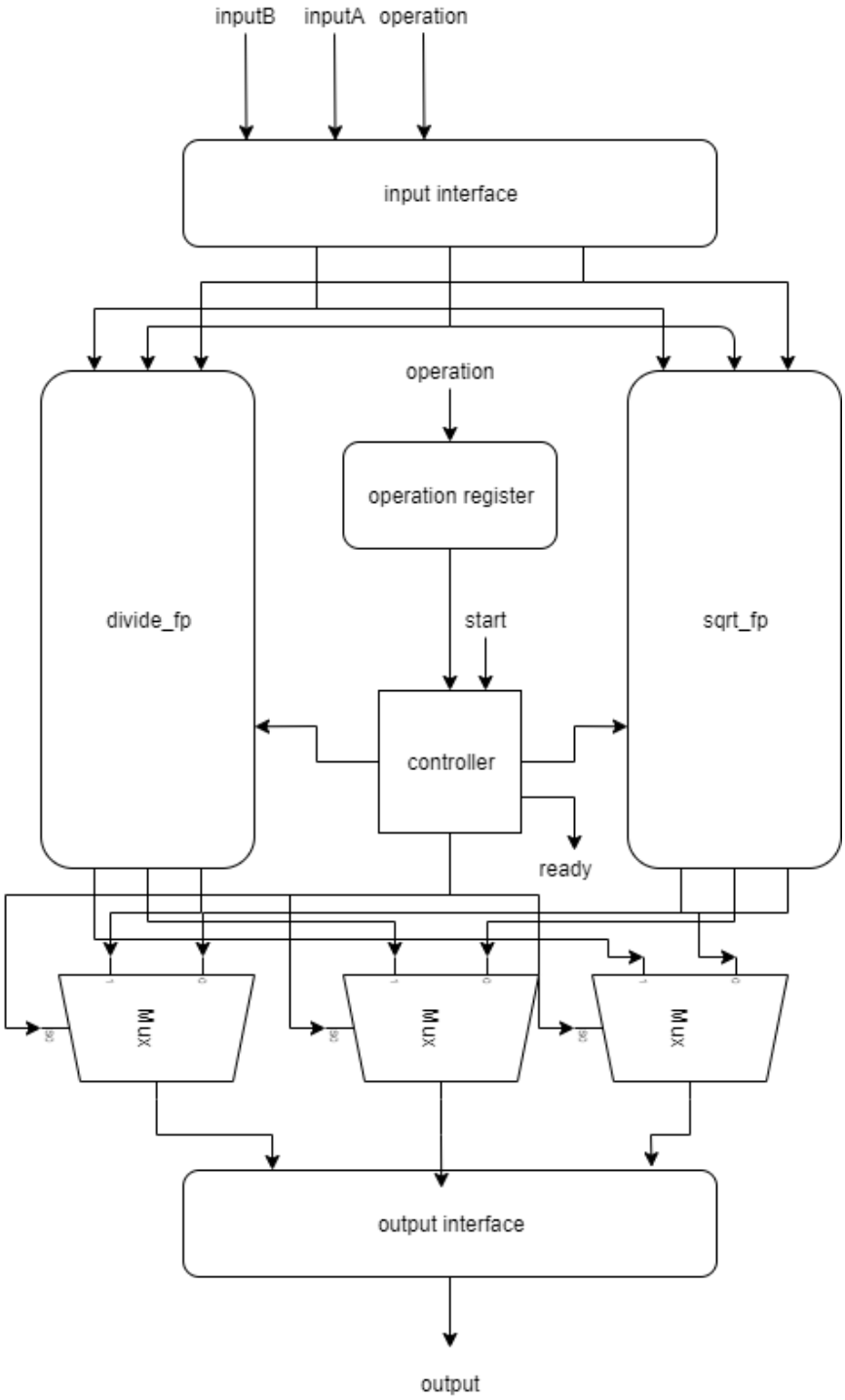
این ماژول‌ها بدنه اصلی مدار را تشکیل می‌دهند و ورودی‌ها پس از عبور از input interface برای پردازش به این ماژول‌ها سپرده می‌شوند. انتخاب این که کدام ماژول اجرا شود توسط ورودی *operation* انجام می‌گیرد که پس از ورود به مدار توسط یک رجیستر ذخیره می‌شود.

۲ـ۵ـ۳ مالتی‌پلکسرهای خروجی

پس از اینکه ماژول فعال شده خروجی را محاسبه کرد، از تعدادی مالتی پلکسر استفاده می کنیم تا با توجه به عملیاتی که کاربر درخواست کرده است، خروجی تقسیم یا جذر را (هم اعشار و هم توان و هم علامت محاسبه شده) به output interface بدهیم تا خروجی مناسب برای نمایش دادن به کاربر را تولید کند.

۲ـ۵ـ۴ ماژول Output Interface

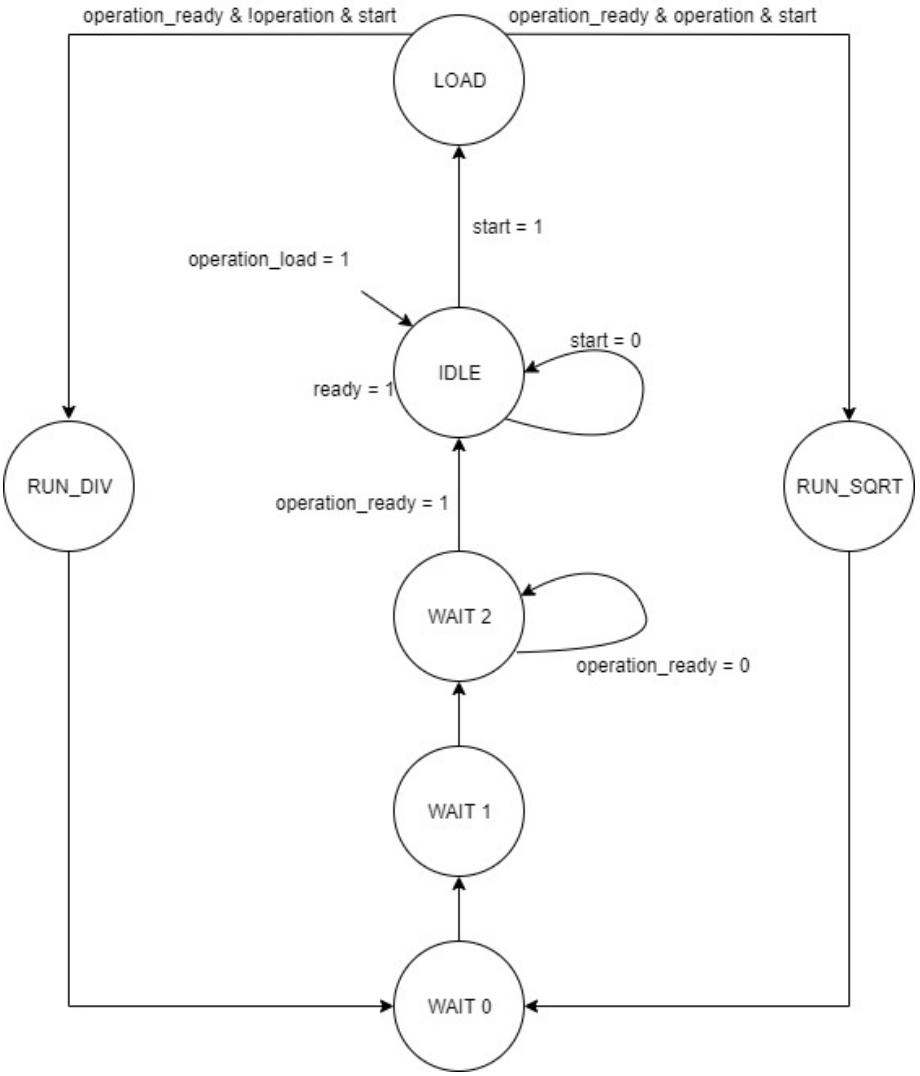
این ماژول علامت، اعشار و توان را به صورت جداگانه از مالتی‌پلکسرها دریافت می‌کند و با بسته کردن آن‌ها، پاسخ را در خروجی قرار می‌دهد.



شکل ۱۶ : مسیر داده ماژول FPU

کنترلر این ماژول نیز سیگنال *start* را دریافت می‌کند و با توجه به نوع عملیات درخواست شده‌ی کاربر، یکی از ماژول‌های جذر یا تقسیم را راه‌اندازی می‌کند. برای راه‌اندازی این ماژول‌ها، هیچ یک از آن‌ها نباید در حال انجام محاسبه بر روی داده باشند و در نتیجه در یک زمان، تنها یکی از این دو

ماژول به انجام عملیات می‌پردازد. همچنین با توجه به وجود کنترلرهای داخلی در هر یک از ماژول‌های جذر و تقسیم، کنترلر مرکزی تعدادی کلاک (که برابر با تعداد کلاک‌های لازم برای آغاز کردن عملیات‌های تقسیم یا جذر است) می‌ایستد و پس از آن منتظر سیگنال *ready* می‌ماند که از یکی از این ماژول‌ها صادر شود.



شکل ۱۷ : واحد کنترل ماژول FPU

۲ـ۶ واسط کاربری[1]

یک واسط کاربری به زبان پایتون نوشته شده است که در ارتباط با اسکریپت[2] دیگری است که به زبان TCL[3] نوشته شده است.

۲ـ۶ـ۱ اسکریپت پایتون

همان طور که در **بخش ۱ـ۳ـ۲** گفته شد، برای استفاده از مدل طلایی به کتاب خانه ی mpmath نیاز داریم؛ بنابراین برای این که کد ما روی هر کامپیوتری قابلیت اجرا داشته باشد، باید این کتاب خانه در هنگام اجرا نصب شود. برای این منظور در کد واسط کاربری، ابتدا این شرط که کتاب خانه روی کامپیوتر نصب شده است یا نه، بررسی می شود و در صورتی که نصب نباشد، یک فایل batch به نام *mpmath.bat* را اجرا می کند که این فایل batch پس از ایجاد directory های مناسب، فایل zip مربوط به mpmath را که در *Test/libs/mpmath-1.1.0.zip* قرار دارد، unzip و نصب می کند.

سپس فایل هایی را که برای ساختن تست و بررسی آن هستند import می کنیم. توابع در بخش بعد شرح داده شده اند. به عنوان خروجی این اسکریپت، علاوه بر پاسخ عملیات ها در یک فایل، درصد دقت این پاسخ ها نشان داده می شود. نحوه بدست آوردن این درصد دقت در **بخش ۳ـ۳** شرح داده شده است.

۲ـ۶ـ۱ـ۱ توابع

getDirectoryFromSource یک آدرس کامل را می گیرد و آن را نسبت به آدرس ریشه بر می گرداند

getListOfFiles لیست فایل های داخل یک directory را بر می گرداند

getModTimes زمان آخرین ویرایش فایل ها را بر می گرداند

isCompiled بررسی می کند که آیا فایل های پروژه compile شده اند یا نه

writeCompileTime زمان آخرین compile را ذخیره می کند

---

[1] User Interface

[2] Script

[3] Tool Command Language

۲ـ۶ـ۱ـ۲ طریقهٔ اجرا

برای حالت تست دستور زیر را در خط فرمان[1] وارد می کنیم

```
python run.py -t <number_of_tests>
```

<number_of_tests> تعداد نمونه هایی است که می خواهیم به صورت خودکار برای تست کردن مدار ساخته شوند

برای حالتی که می خواهیم عملیات های دلخواه روی ورودی های دلخواه انجام شوند، دستور زیر را وارد می کنیم

```
python run.py <input_file_name>
```

<input_file_name> نام فایل ورودی است که توضیحات مربوط به آن در **بخش ۲ـ۶ـ۳** آمده است.

۲ـ۶ـ۲ اسکریپت TCL

نام فایل ورودی و تعداد نمونه های آن از طریق فایل پایتون به این اسکریپت داده می شود. در صورتی که فایل ها از قبل compile شده باشند و پس از آن ویرایش نشده باشند، دوباره compile نمی شوند و در غیر این صورت compile می شوند. پس از compile، عملیات شبیه سازی[2] صورت می گیرد و پس از آن هم تا انتهای کد اجرا می شود. هم چنین در مرحله ی compile، مسیر های فایل های پروژه اضافه می شوند.

۲ـ۶ـ۳ فایل ورودی

هر خط از فایل ورودی به صورت زیر باید باشد:

```
<operation>, <input1>, <input2_if_exists>
```

[1] Command line

[2] Simulation

اگر عملیات تقسیم باشد باید دو ورودی داشته باشیم که اولی بر دومی تقسیم می شود، و اگر جذر باشد یک ورودی دارد که از آن جذر گرفته می شود. توضیحات مربوط به حالات مختلف <operation> در **جدول ۱** آمده است.

۲ـ۶ـ۴ فایل خروجی

در مرحله ی اجرای کد، پس از اتمام عملیات محاسباتی، نتایج در فایلی به نام *output.csv* نوشته می شوند. هر خط از این فایل متناظر با همان خط در فایل ورودی است.

هر خط از این فایل به صورت زیر نوشته می شود:

```
<operation>, <input1>, <input2_if_exists>, <result>, <number_of_clocks>
```

در صورتی که عملیات تقسیم باشد، هر دو ورودی به ترتیب نوشته می شوند و در صورتی که عملیات جذر باشد، تنها ورودی نوشته می شود. فرمت همه این اعداد باینری است؛ ورودی و خروجی طبق استاندارد IEEE 754 نوشته می شوند. هم چنین، <number_of_clocks> تعداد کل کلاک هایی را که عملیات طول کشیده است نشان می دهد.

۳ شبیه سازی و تست

روی **۷ میلیون نمونه ی تصادفی** تست انجام شده است و تمامی آن ها با دقت 100% درستی آزمایی [1] شده است.

نحوه تست کلی مدار در بخش های زیر شرح داده شده است.

۳ـ۱ Test Bench

علاوه بر Test Bench های ماژول های درونی، یک Test Bench کلی هم برای مدار نوشته شده که در آن یک نمونه [2] از ماژول FPU را تست می‌کنیم. بخش‌های مختلف آن به شرح زیر است.

۳ـ۱ـ۱ ورودی و خروجی

برای این که بتوان این ماژول را در تعداد بالا تست کرد به گونه‌ای که نیاز به تغییر زیادی در test bench نباشد و همچنین برای بررسی خودکار تست‌ها ورودی‌ها را از فایل می‌خوانیم و خروجی‌ها را نیز در فایل می‌نویسیم (برای تغییر تعداد تست‌ها کافیست عدد REPEAT را تغییر دهیم). برای خواندن و نوشتن نیز از فایل csv استفاده می‌کنیم که هم فرمت ساده‌ای دارد و هم با توجه به پشتیبانی نرم‌افزار excel عملیات‌های تست و بررسی درستی خروجی‌ها ساده‌تر انجام می‌شود. برای خواندن و نوشتن ورودی‌ها از ماکروهای زبان C استفاده می‌شود.

۳ـ۱ـ۲ ریست و خاتمه عملیات

این بخش در یک بلوک initial جداگانه پیاده می‌شود و سیگنال‌های ریست و همچنین خاتمه مدار را مشخص می‌کند.

۳ـ۱ـ۳ دادن ورودی‌ها به ماژول اصلی

در بلاک اصلی initial ورودی‌ها از فایل خوانده شده و بسته به نوع عملیات خروجی‌ها را می‌نویسیم. از سویی برای این که خروجی‌های ماژول آماده شده باشند منتظر دیدن سیگنال *ready* مدار می‌مانیم و پس از آن نیز یک کلاک صبر می‌کنیم تا خروجی به حالت پایدار درآید. پس از آن نیز

[1] Verify
[2] Instance

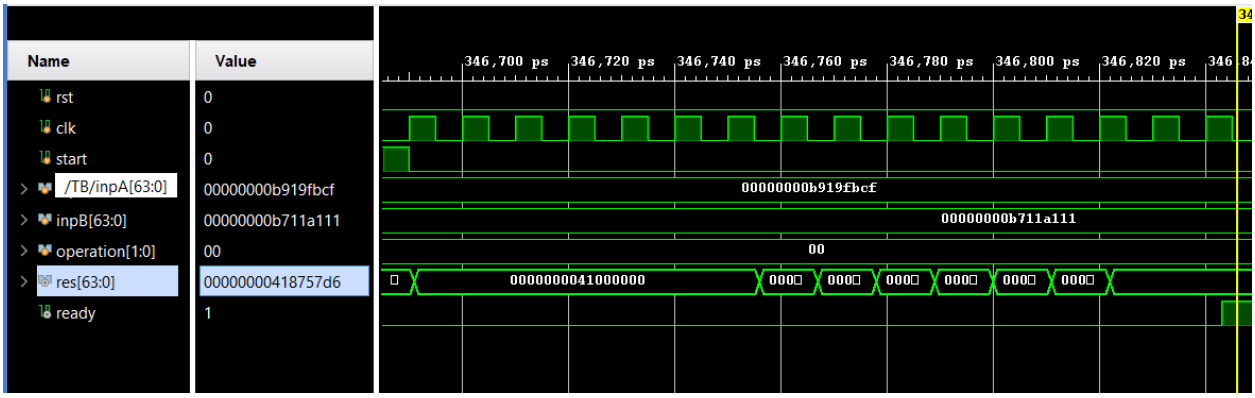
بسته به نوع عملیات، خروجی را در فایل می‌نویسیم (زیرا بسته به نوع عملیات خروجی‌های متفاوتی در فایل نوشته می‌شود).

۳ـ۱ـ۴ شمارنده کلاک

برای شمارش تعداد کلاک‌های مصرف شده در هر بار اجرای عملیات، از یک شمارنده استفاده می‌کنیم که باعث می‌شود که بررسی تعداد کلاک‌های مصرف شده در هر بار عملیات ساده‌تر شود. شمارنده‌ای که این عملیات را انجام می‌دهد، با مشاهده سیگنال *start* صفر می‌شود و تا زمانی که سیگنال *ready* صادر نشود در هر لبه بالا رونده کلاک یک واحد افزوده می‌شود.

۳ـ۲ نمونه هایی از تست مدار روی Wave

همان طور که در بخش ۲ـ۶ گفته شد، پس از compile کد ها، شبیه سازی صورت می گیرد اما این شبیه سازی توسط vsim در پس زمینه انجام می شود؛ ما برای این که نتایج را ببینیم شبیه سازی را در محیط Vivado انجام دادیم. **توجه کنید که تعداد کلاک های اجرایی شامل زمان های مربوط به دریافت ورودی و بازگردانی خروجی نیز می باشد.**



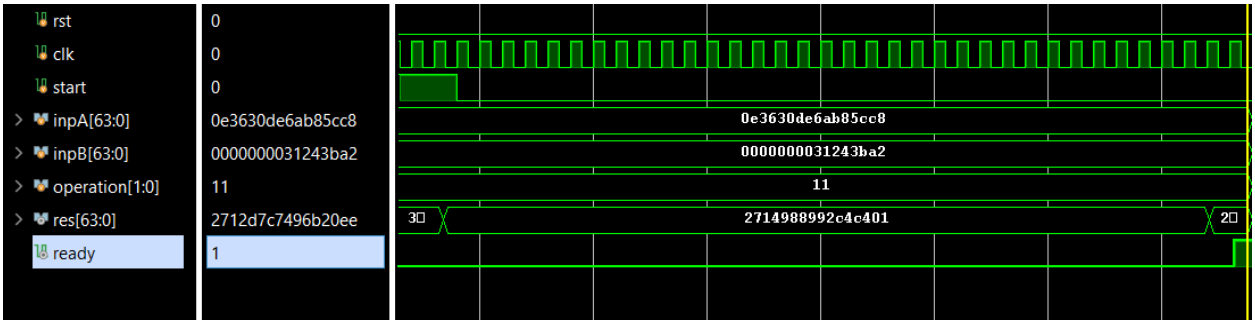
شکل ۱۸ : نتیجه شبیه سازی– عملیات تقسیم ـ ۱

همانگونه که در تصویر مشاهده می‌کنید عدد

$-1.468502305215224623680114746 09*10^{-4}$

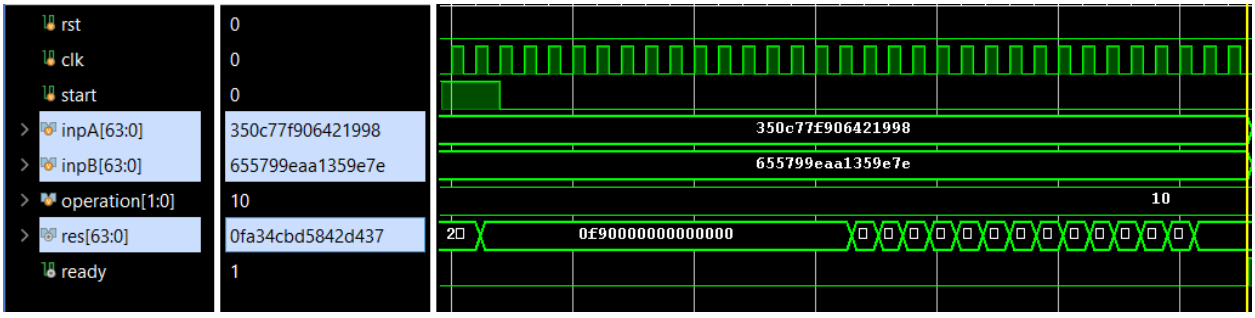
بر عدد $-8.680174687469843775033950805 66*10^{-6}$

تقسیم گردیده است و حاصل آن که عدد $1.69178886413574218 75*10^{1}$ است در تعداد ۱۶ کلاک اماده شده است. این عملیات با دقت واحد است.



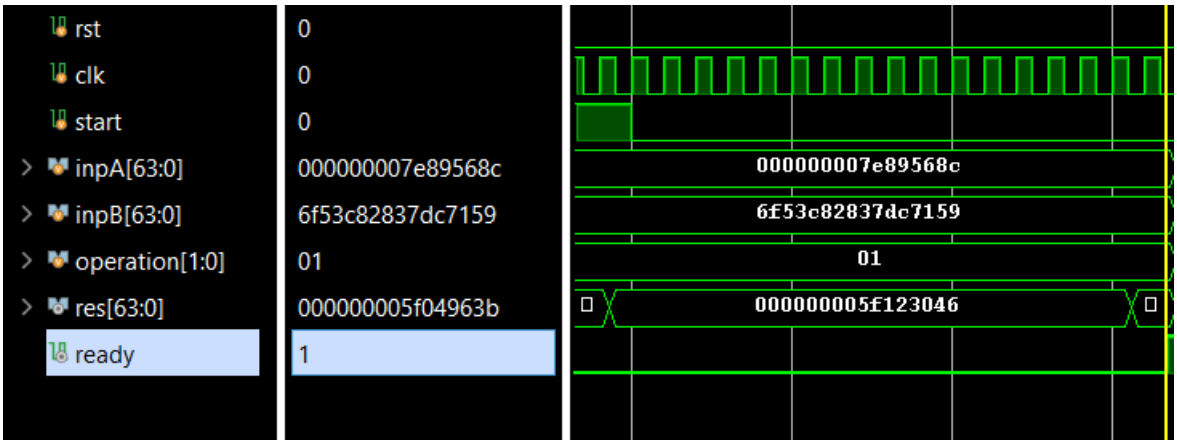
شکل ۱۹ : نتیجه شبیه سازی– عملیات جذر ـ ۱

همانگونه که در تصویر مشاهده می‌کنید عدد $3.32796127476367806509331758208*10^{-240}$ جذر گرفته شده است که و حاصل آن که عدد $1.82427006629053613574618211598*10^{-120}$ است در تعداد ۳۶ کلاک آماده شده است. این عملیات با دقت مضاعف است.



شکل ۲۰ : نتیجه شبیه سازی– عملیات تقسیم ـ ۲

همانگونه که در تصویر مشاهده می‌کنید عدد $3.71533134059355382513869500292*10^{-53}$ بر عدد $1.53021403048194713683931405726*10^{180}$ تقسیم گردیده است و حاصل آن که عدد $2.42798148924526235730505157598*10^{-233}$ است در تعداد ۳۳ کلاک اماده شده است. این عملیات با دقت مضاعف است.



شکل ۲۱ : نتیجه شبیه سازی ـ عملیات جذر ـ ۲

همانگونه که در تصویر مشاهده می‌کنید عدد $9.12768062448559904988672712 79*10^{37}$ جذر گرفته شده است که و حاصل آن که عدد $9.553888530699124736*10^{18}$ است در تعداد ۳۶ کلاک آماده شده است. این عملیات با دقت واحد است.

۳ـ۳ درستی آزمایی

همانطور که پیش تر توضیح داده شده است، نتایج حاصل از شبیه‌سازی در فایل csv نوشته خواهد شد. برای تست اکثر ماژول‌ها در فایل testHelper تابعی برای تولید تست و درستی‌آزمایی نتایج حاصل از خروجی قرار داده شده است. در اینجا فقط به توضیح شیوهٔ انجام تست ماژول اصلی (FPU) می‌پردازیم. هنگامی که نتایج حاصل از شبیه سازی در فایل csv قرار گرفت، تابع FPUTestVerifier موجود در کتابخانه testHelper با دریافت آدرس فایل خروجی شبیه‌سازی به راستی آزمایی تست‌ها می‌پردازد. فایل خروجی حاصل از شبیه‌سازی ماژول FPU، نتایج حاصل را به صورت زیر در فایل خروجی خود قرار می دهد:

تقسیم	Operation	Input1	Input2	Result	Num of cycles
جذر	Operation	Input	-	Result	Num of cycles

تابع FPUTestVerifier

این تابع با توجه به نوع *operation* یکی از دو مسیر لازم برای ارزیابی جذر یا تقسیم را انتخاب می‌نماید. پس از استخراج داده های ورودی، آن‌ها را به توابع SQRT یا Divide می‌دهد. این دو

تابع به طور مستقیم از توابع موجود در رابط مدل طلایی استفاده کرده و نتایج را به صورت نرم افزاری شبیه سازی می‌کند. سپس نتیجه حاصل از ماژول FPU از فایل csv استخراج شده و به کمک تابع check_answer با نتیجه شبیه‌سازی با استفاده از مدل طلایی مقایسه می‌شود. در صورتی که نتیجه غلط باشد، نتیجه مورد انتظار و نتیجه حاصل شده نمایش داده خواهد شد. در پایان نیز درصد دقت تست نمایش داده می‌شود.

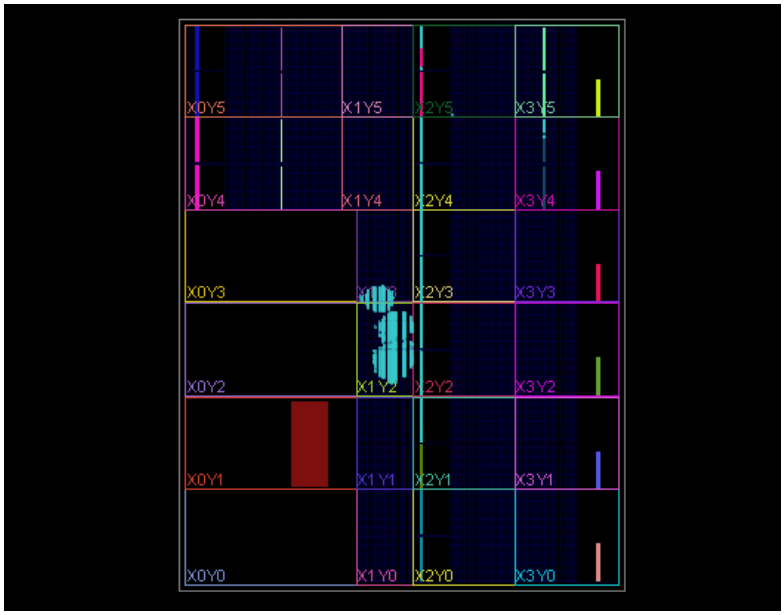
تابع check_answer

در زبان‌های نرم‌افزاری مانند Python یا C++، با بهره‌گیری از توانایی CPU های اینتل، برای محاسبات floating point محاسبات را با دقت بالاتر از ۳۲ یا ۶۴ بیت انجام می‌دهند و در پایان وقتی بخواهند اعداد را به فرمت ۳۲ یا ۶۴ بیتی تبدیل کنند، مقدار *mantissa* را به بالا یا پایین گرد می‌کند. چون در FPU همچین مکانیزمی وجود ندارد، فرض بر آن شد که همیشه مقدار *mantissa* را همان ۲۳ یا ۵۲ بیت در نظر بگیریم. لذا تفاوت نتیجه محاسبه شده توسط FPU و کد python تنها در جمع یک مقدار یک با *mantissa* است. بنابراین در تابع check_answer ابتدا بررسی می کنیم اگر مقدار محاسبه شده FPU و کد Python برابر باشد جواب درست بر می گرداند. اگر برابر نباشند، نتیجه حاصل از FPU را با یک جمع کرده و با نتیجه کد Python مقایسه می‌کند. در صورتی که برابر باشد جواب درست برمیگرداند و در غیر این‌صورت جواب نادرست برمیگرداند.

۴ سنتز

سنتز مدار به کمک نرم افزار vivado صورت گرفته است. ابتدا برای این منظور باید include path در پروژه vivado مطابق با کامپیوتر تنظیم شود. به عبارت دیگر باید در آدرس tools/setting/Verilog options/included path تمام زیر پوشه های موجود در پروژه وارد شود. در فرایند سنتز تعدادی هشدار[۱] وجود داشت که با بررسی آن‌ها دریافتیم ایرادی جدی وجود ندارد. در هشدارهای داده شده، فقط یک مورد سیم وجود دارد که به هیچ کجا وصل نشده است (در output_interface) که این عمل عامدانه و از روی آگاهی صورت گرفته است لذا مشکلی وجود ندارد.

در تست functional بعد از سنتز، عملکرد مدار به درستی صورت نمی‌گرفت و در FSM ساخته شده، وضعیت تغییر نمی‌کرد. با بررسی‌های انجام شده و تجزیه مدار سنتز شده برای FSM دریافتیم که در شماتیک ایجاد شده مشکلی وجود ندارد و عدم اجرای صحیح مدار، به شکلی است که رجیستر مربوط به تغییر وضعیت، مقدار صحیح ورودی را دریافت می‌کند، کلاک دارد و load آن فعال است، اما نمونه برداری صورت نمی‌گیرد. در پایان این ایراد عجیب در شبیه‌سازی بعد از سنتز با وجود تلاش‌های بسیار زیاد، رفع نگردید. در زیر به گزارشات مربوط به سنتز می‌پردازیم:



شکل ۲۲ : نقشه قرار گیری مدار روی بورد

[۱] Warning

۴ـ۱ گزارش زمانی مدار[1]

در تصاویر زیر، نتیجه بعد از سنتز و بعد از پیاده‌سازی[2] را مشاهده می‌کنید. بعد از پیاده‌سازی، hold به طور کامل برطرف شده است.

جدول ۵ : گزارش زمانی سنتز

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.785 ns	Worst Hold Slack (WHS):	-0.075 ns	Worst Pulse Width Slack (WPWS):	2.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	-13.034 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	416	Number of Failing Endpoints:	0
Total Number of Endpoints:	1203	Total Number of Endpoints:	1203	Total Number of Endpoints:	826

Timing constraints are not met.

جدول ۶ : گزارش زمانی پیاده سازی

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.916 ns	Worst Hold Slack (WHS):	0.058 ns	Worst Pulse Width Slack (WPWS):	2.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1203	Total Number of Endpoints:	1203	Total Number of Endpoints:	826

All user specified timing constraints are met.

[1] Design timing summary

[2] Implementation

۴ـ۲ گزارش کلاک[1]

فرکانس ماکسیمم: ۹۰ مگاهرتز

جدول ۷ : گزارش کلاک

Name	Waveform	Period (ns)	Frequency (MHz)
clk	{0.000 5.000}	10.000	100.000
clk_out1_clk_wiz_0	{0.000 5.556}	11.111	90.000

Timing Check	Count 1	Worst Severity
no_input_delay	131	High
no_output_delay	65	High
no_clock	0	
constant_clock	0	
pulse_width_clock	0	
unconstrained_internal_endpoints	0	
multiple_clock	0	
generated_clocks	0	
loops	0	
partial_input_delay	0	
partial_output_delay	0	
latch_loops	0	

[1] Clock Report Summary

۴ـ۳ گزارش Clock Interaction

جدول ۸ : گزارش *Clock Interaction*

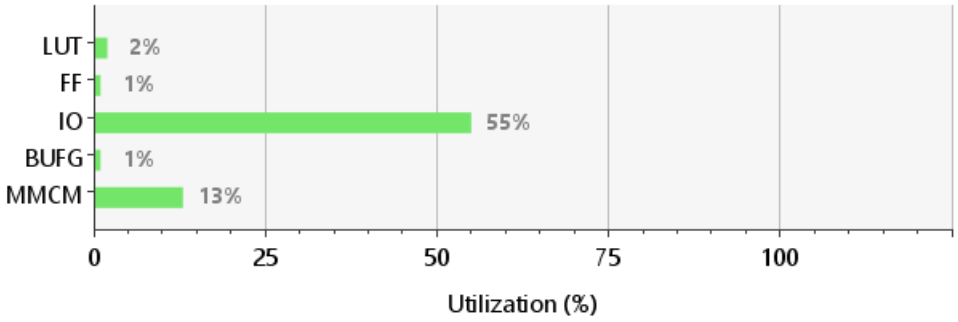
Source Clock	Destination Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)	Path Req (WNS)
clk_out1_clk_wiz_0	clk_out1_clk_wiz_0	rise - rise	0.916	0.000	0	1203	11.111

Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints (THS)	Total Endpoints (THS)	Path Req (WHS)	Clock Pair Classification	Inter-Clock Constraints
rise - rise	0.058	0.000	0	1203	0.000	Clean	Timed

۴ـ۴ گزارش بهره‌برداری[1]

جدول ۹ : گزارش بهره برداری

Resource	Utilization	Available	Utilization %
LUT	3945	230400	1.71
FF	823	460800	0.18
IO	198	360	55.00
BUFG	1	544	0.18
MMCM	1	8	12.50



[1] Utilization

Ref Name	Used	Functional Category
LUT5	1514	CLB
LUT3	1146	CLB
LUT6	981	CLB
FDCE	823	Register
LUT4	584	CLB
LUT2	431	CLB
LUT1	267	CLB
CARRY8	253	CLB
INBUF	133	I/O
IBUFCTRL	133	Others
OBUF	65	I/O
MUXF7	1	CLB
MMCME4_ADV	1	Clock
BUFGCE	1	Clock

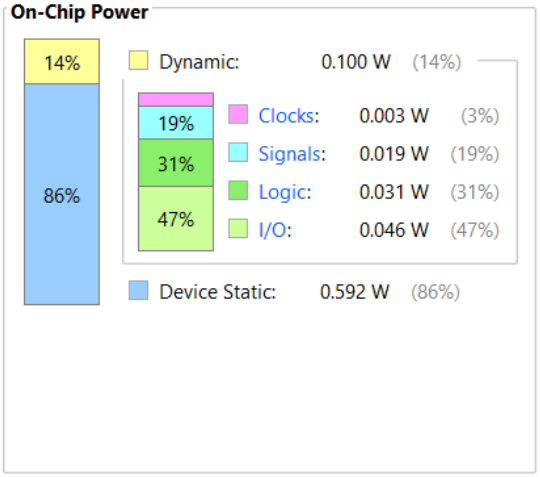
۴ـ۵ گزارش توان[1]

جدول ۱۰ : گزارش توان

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.692 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.7°C**
Thermal Margin:	74.3°C (75.2 W)
Effective ϑJA:	1.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



[1] Power report

Supply Source	Voltage (V)	Total (A)	Dynamic (A)	Static (A)
Vccint	0.850	0.199	0.064	0.136
Vccint_io	0.850	0.072	0.002	0.070
Vccbram	0.850	0.002	0.000	0.002
Vccaux	1.800	0.148	0.000	0.148
Vccaux_io	1.800	0.069	0.014	0.055
Vcco33	3.300	0.000	0.000	0.000
Vcco25	2.500	0.000	0.000	0.000
Vcco18	1.800	0.012	0.012	0.000
Vcco15	1.500	0.000	0.000	0.000
Vcco135	1.350	0.000	0.000	0.000
Vcco12	1.200	0.000	0.000	0.000
Vcco10	1.000	0.000	0.000	0.000
Vccadc	1.800	0.008	0.000	0.008
VCC_PSINTFP	0.850	0.000	0.000	0.000
VCC_PSINTLP	0.850	0.005	0.000	0.005
VPS_MGTRAVCC	0.850	0.000	0.000	0.000
VCC_PSINTFP_DDR	0.850	0.000	0.000	0.000
VCC_PSPLL	1.200	0.002	0.000	0.002
VPS_MGTRAVTT	1.800	0.000	0.000	0.000
VCCO_PSDDR_504	1.200	0.000	0.000	0.000
VCC_PSAUX	1.800	0.002	0.000	0.002
VCC_PSBATT	1.200	0.000	0.000	0.000
VCC_PSDDR_PLL	1.800	0.000	0.000	0.000
VCCO_PSIO0_500	3.300	0.000	0.000	0.000
VCCO_PSIO1_501	3.300	0.000	0.000	0.000
VCCO_PSIO2_502	3.300	0.000	0.000	0.000
VCCO_PSIO3_503	3.300	0.000	0.000	0.000
VCC_PSADC	1.800	0.001	0.000	0.001
VCCINT_VCU	0.900	0.025	0.000	0.025
MGTAVcc	0.900	0.000	0.000	0.000
MGTAVtt	1.200	0.000	0.000	0.000
MGTVccaux	1.800	0.000	0.000	0.000

۴ـ۶ سایر گزارش‌های موجود در فایل Log

۴ـ۶ـ۱ FSM های تشخیص داده شده و کدینگ جدید آن‌ها

جدول ۱۱ : FSM های فایل Log

```
---------------------------------------------------------------------------
INFO: [Synth 8-802] inferred FSM for state register 'ps_reg' in module 'input_wrapper_CU'
INFO: [Synth 8-802] inferred FSM for state register 'ps_reg' in module 'sqrt_CU'
INFO: [Synth 8-802] inferred FSM for state register 'ps_reg' in module 'div_wrapper_CU'
INFO: [Synth 8-802] inferred FSM for state register 'ps_reg' in module 'FPU_CU'
---------------------------------------------------------------------------------------------------
|  |  |  |  |  State |                   New Encoding |               Previous Encoding
---------------------------------------------------------------------------------------------------
|  |  |  |  |   IDLE |                             00 |                             00
|  |  |  |  |   LOAD |                             01 |                             01
|  |  |  |  |    RUN |                             10 |                             10
---------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'ps_reg' using encoding 'sequential' in module 'input_wrapper_CU'
---------------------------------------------------------------------------------------------------
|  |  |  |  |  State |                   New Encoding |               Previous Encoding
---------------------------------------------------------------------------------------------------
|  |  |  |  |   IDLE |                            000 |                            000
|  |  |  |  |  START |                            001 |                            001
|  |  |  |  | SELECT |                            010 |                            010
|  |  |  |  |   LOAD |                            011 |                            011
|  |  |  |  | FINISH |                            100 |                            100
---------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'ps_reg' using encoding 'sequential' in module 'sqrt_CU'
---------------------------------------------------------------------------------------------------
|  |  |  |  |  State |                   New Encoding |               Previous Encoding
---------------------------------------------------------------------------------------------------
|  |  |  |  |   IDLE |                             00 |                             00
|  |  |  |  |  RESET |                             01 |                             01
|  |  |  |  |CALCULATE |                             10 |                             10
---------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'ps_reg' using encoding 'sequential' in module 'div_wrapper_CU'
INFO: [Synth 8-6159] Found Keep on FSM register 'ps_reg' in module 'FPU_CU', re-encoding will not be performed
---------------------------------------------------------------------------------------------------
|  |  |  |  |  State |                   New Encoding |               Previous Encoding
---------------------------------------------------------------------------------------------------
*
|  |  |  |  |   IDLE |                            000 |                            000
|  |  |  |  |   LOAD |                            110 |                            110
|  |  |  |  | RUN_DIV |                            001 |                            001
|  |  |  |  |RUN_SQRT |                            010 |                            010
|  |  |  |  |  WAIT0 |                            011 |                            011
|  |  |  |  |  WAIT1 |                            100 |                            100
|  |  |  |  |  WAIT2 |                            101 |                            101
---------------------------------------------------------------------------------------------------
```

۴ـ۶ـ۲ گزارش سیم‌هایی با چند Driver

جدول ۱۲ : گزارش سیم های با چند Driver

```
Report Check Netlist:
+------+------------------+-------+---------+-------+------------------+
|      |Item              |Errors |Warnings |Status |Description       |
+------+------------------+-------+---------+-------+------------------+
|1     |multi_driven_nets |      0|        0|Passed |Multi driven nets |
+------+------------------+-------+---------+-------+------------------+
```

۴ـ۶ـ۳ گزارش Component های مصرف شده

جدول ۱۳ : گزارش Component های مصرف شده

```
Detailed RTL Component Info :
+---Adders :
      3 Input    106 Bit       Adders := 8
      2 Input    106 Bit       Adders := 4
      4 Input     13 Bit       Adders := 1
      3 Input     12 Bit       Adders := 1
+---XORs :
      2 Input      1 Bit         XORs := 159
+---Registers :
                 106 Bit    Registers := 10
                  11 Bit    Registers := 3
                   2 Bit    Registers := 1
+---Muxes :
      2 Input    106 Bit        Muxes := 43
      3 Input    106 Bit        Muxes := 1
      2 Input     64 Bit        Muxes := 1
      2 Input     53 Bit        Muxes := 8
      3 Input     53 Bit        Muxes := 2
      4 Input     53 Bit        Muxes := 1
      2 Input     13 Bit        Muxes := 1
      2 Input     11 Bit        Muxes := 9
      3 Input     11 Bit        Muxes := 1
      3 Input      7 Bit        Muxes := 1
      4 Input      3 Bit        Muxes := 4
      2 Input      3 Bit        Muxes := 10
      5 Input      3 Bit        Muxes := 1
      8 Input      3 Bit        Muxes := 1
      3 Input      2 Bit        Muxes := 2
      2 Input      2 Bit        Muxes := 5
      2 Input      1 Bit        Muxes := 5
      3 Input      1 Bit        Muxes := 4
      5 Input      1 Bit        Muxes := 8
```

۴ـ۶ـ۴ گزارش فضای مصرف شده توسط هر ماژول

جدول ۱۴ : گزارش فضای مصرفی هر ماژول

	Instance	Module	Cells
1	top		6199
2	fpu	FPU	5955
3	CU	FPU_CU	10
4	DP	FPU_DP	5945
5	Fp_sqrt	fp_sqrt	2248
6	Input_wrapper	input_wrapper	242
7	CU	input_wrapper_CU	17
8	DP	input_wrapper_DP	225
9	exponent_register	register_sqrt__parameterized1_18	30
10	mantis_register	register_sqrt__parameterized0_19	195
11	Main_module	sqrt	2006
12	Controller	sqrt_CU	11
13	Datapath	sqrt_DP	1993
14	max_power_mux	multiplexer	38
15	num_mux	multiplexer_5	106
16	number_reg	register_sqrt__parameterized0	384
17	out_reg	register_sqrt__parameterized0_6	53
18	power_reg	register_sqrt__parameterized0_7	929
19	result_reg	register_sqrt__parameterized0_8	243
20	sqrt_level1	sqrt_unit	22
21	Comparator	comparator_16	7
22	Subtactor	subtractor_17	15
23	sqrt_level2	sqrt_unit_9	23
24	Comparator	comparator_14	8
25	Subtactor	subtractor_15	15
26	sqrt_level3	sqrt_unit_10	173
27	Comparator	comparator_12	8
28	Subtactor	subtractor_13	165
29	sqrt_level4	sqrt_unit_11	22
30	Comparator	comparator	8
31	Subtactor	subtractor	14
32	fp_divide	wrapper	3523
33	DP	div_wrapper_DP	3487
34	M1	divider	3376
35	r1	register_div	239
36	r2	register_div_1	1490
37	r3	register_div_2	1378
38	r4	register_div_3	199
39	r5	register_div_4	65
40	expA_reg	register_sqrt__parameterized1	43
41	expB_reg	register_sqrt__parameterized1_0	45
42	cu	div_wrapper_CU	36
43	operationRegister	register_sqrt	174

Report Cell Usage:

	Cell	Count
1	clk_wiz_0	1
2	CARRY8	253
3	LUT1	267
4	LUT2	431
5	LUT3	1146
6	LUT4	584
7	LUT5	1514
8	LUT6	981
9	MUXF7	1
10	FDCE	823
11	IBUF	132
12	OBUF	65

۵ نتیجه گیری

در ابتدا درباره تاریخچه و کاربردهای ماژول FPU در صنعت توضیحاتی ارائه گردید. سپس درباره مدل طلایی و خواص و پیاده‌سازی‌های آن به صورت مفصل بحث گردید. در ادامه پایهٔ ریاضی و جبری اعمال انجام شده و برخی اثبات‌های ریاضی برای استفاده از الگوریتم‌های مورد استفاده آورده شد. در ادامه بحث، هر یک از ماژول‌ها استفاده شده و کاربرد آن‌ها و تناظر آن‌ها با الگوریتم‌های مورد استفاده آورده شد. سپس به تفصیل راجع به اسکریپت TCL و رابط پایتون برای سهولت تست و شبیه‌سازی موضوعاتی بیان شد و شیوه های تست به طور دقیق مورد بحث قرار گرفت. در پایان نیز شرح عمل سنتز و گزارشات حاصل از آن مطرح گردید که حاکی از موفقیت این عملیات است.

تعداد کلاک های اجرایی عملیات ها در مدار طراحی شده، برای دقت واحد حداکثر ۱۳ کلاک و برای دقت مضاعف حداکثر ۲۷ کلاک می باشد.

برخی از دست‌آورد ها

۱. سادگی پیاده‌سازی الگوریتم‌های انتخاب شده و کارایی آن‌ها
۲. مصرف انرژی کم
۳. ماژولار بودن ساختار پروژه و قابلیت استفاده مجدد از ماژول‌ها در سایر پروژه‌ها
۴. سهولت شبیه‌سازی به کمک اسکریپت‌های نوشته شده

مراجع

[1] *Wikipedia - nth root algorithm*

https://en.m.wikipedia.org/wiki/Nth_root_algorithm

[2] *Wikipedia - long division algorithm*

https://en.m.wikipedia.org/wiki/Long_division

[3] *Wikipedia - computing square roots*

https://en.m.wikipedia.org/wiki/Methods_of_computing_square_roots

[4] *Wikipedia - shifting nth root algorithm*

https://en.m.wikipedia.org/wiki/Shifting_nth_root_algorithm

[5] *Wolfram Mathworld – square root algorithms*

http://mathworld.wolfram.com/SquareRootAlgorithms.html

[6] *View Point Systems – product testing methods*

https://www.viewpointusa.com/TM/wp/product-testing-methods-industrial-hardware-products/

[7] *Wikipedia – IEEE754*

https://en.m.wikipedia.org/wiki/IEEE_754

[8] *Wikipedia – floating point unit*

https://en.m.wikipedia.org/wiki/Floating-point_unit

[9] *MPmath – MPmath documentation*

http://mpmath.org/doc/current/