

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ



دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

پایان نامه کارشناسی

عنوان

یکپارچه سازی، صحت سنجی و ارزیابی یک ریزپردازنده نهفته تحمل پذیر اشکال

نگارش

محمد جواد دوستی

گرایش: سخت افزار

پوریا جولانی

گرایش: نرم افزار

استاد راهنما

دکتر سید قاسم میرعمادی

تیرماه ۱۳۸۹

به نام خدا

دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

پایان نامه کارشناسی

عنوان: یکپارچه سازی، صحت سنجی و ارزیابی یک ریزپردازنده نهفته تحمل پذیر اشکال

نگارش: پوریا جولانی، محمد جواد دوستی

استاد راهنما: دکتر سید قاسم میرعمادی

امضاء.....

تاریخ.....

## چکیده

سیستم‌های نهفته در اغلب وسایلی که بخشی از زندگی امروز را تشکیل می‌دهند وجود دارند. اهمیت و فراگیری این سیستم‌ها به حدی است که از کل ریزپردازنده‌هایی که هر ساله تولید می‌شود، ۷۹ تا ۹۹ درصد آن‌ها در سیستم‌های نهفته به کار می‌روند. به دلیل اهمیت ایمنی در بخشی از کاربردهای سیستم‌های نهفته، یکی از پارامترهای مهم در طراحی این سیستم‌ها و ریزپردازنده‌های آن‌ها، قابلیت اطمینان است. به همین دلیل از روش‌های مختلفی برای افزایش قابلیت اطمینان در ریزپردازنده‌های نهفته استفاده می‌شود. هدف این پژوهش، یکپارچه‌سازی، صحت‌سنجی عملکرد و ارزیابی قابلیت اطمینان یک ریزپردازنده نهفته تحمل‌پذیر اشکال است. در مرحله‌ی یکپارچه‌سازی، بخش‌های تحمل‌پذیر اشکال که قبلاً طراحی شده‌اند، شامل واحد محاسبه و منطق، واحد کنترل، بانک ثبات و وسایل جانبی، با هم یکپارچه شده و یک ریزپردازنده‌ی تحمل‌پذیر اشکال مبتنی بر معماری SPARC نسخه ۸ را برای مصارف بحرانی-امن تشکیل می‌دهند. نظر به این که یک واحد ممیز شناور برای این ریزپردازنده موجود نبود، یک واحد ممیز شناور سازگار با استاندارد IEEE-754 نیز طراحی، پیاده‌سازی و به ریزپردازنده افزوده شده و همچنین روش‌های تحمل‌پذیری اشکال برای آن ارائه شده است. برای سنجش صحت عملکرد این ریزپردازنده از برنامه‌های محک استاندارد MiBench استفاده شده است. در ادامه، ارزیابی روش‌های افزایش قابلیت اطمینان در این ریزپردازنده به کمک روش تزریق اشکال مبتنی بر مدل اشکال SEU ارزیابی شده‌اند.

**کلید واژه‌ها:** تحمل‌پذیری اشکال، تزریق اشکال، تکرخداد فیزیکی، صحت‌سنجی، یکپارچه‌سازی،

سیستم نهفته

۱	مقدمه .....	۱
۲	سیستم‌ها و ریزپردازنده‌های نهفته .....	۳
۲-۱	سیستم‌های نهفته .....	۳
۲-۲	اهمیت، تاریخچه و آینده سیستم‌های نهفته .....	۵
۲-۳	کاربردهای سیستم‌های نهفته .....	۷
۲-۴	قابلیت اطمینان در سیستم‌های نهفته .....	۸
۲-۵	ریزپردازنده‌های نهفته .....	۹
۲-۶	سیستم‌عامل‌های نهفته .....	۱۰
۳	تحمل‌پذیری اشکال در ریزپردازنده‌ی هدف .....	۱۳
۳-۱	معماری SPARC نسخه ۸ .....	۱۳
۳-۱-۱	ثبات‌ها .....	۱۴
۳-۱-۲	خط‌لوله .....	۱۸
۳-۱-۳	شیوه‌های آدرس‌دهی .....	۲۰
۳-۱-۴	قالب دستورالعمل‌ها .....	۲۱
۳-۱-۵	معماری واحد مدیریت حافظه .....	۲۱
۳-۱-۶	ریزپردازنده‌ی LEON 2 و نسخه تحمل‌پذیر اشکال آن .....	۲۳
۳-۲	واحدهای تحمل‌پذیر اشکال .....	۲۴

۳-۲-۱	واحد کنترل	۲۵
۳-۲-۲	واحد محاسبه و منطق	۲۶
۳-۲-۳	بانک ثبات	۲۶
۳-۲-۴	وسایل جانبی روی تراشه	۲۷
۴	طراحی و پیاده‌سازی واحد ممیز شناور ریزپردازنده‌ی هدف	۲۹
۴-۱	نیازمندی‌های یک واحد ممیز شناور	۲۹
۴-۲	واحد ممیز شناور موجود و واحد ممیز شناور جدید	۳۱
۴-۳	تغییرات انجام شده بر روی کد مبنا	۳۴
۴-۴	واحدهای مقایسه و تبدیلات	۳۷
۴-۵	تحمل‌پذیری اشکال در واحد ممیز شناور	۳۷
۴-۵-۱	روش‌های موجود	۳۸
۴-۵-۲	روش پیشنهادی برای حالت دقت مضاعف	۳۹
۴-۵-۳	روش پیشنهادی برای حالت دقت ساده	۴۱
۵	یکپارچه‌سازی واحدهای اصلی ریزپردازنده هدف	۴۴
۵-۱	روش انجام آزمون اولیه	۴۴
۵-۲	آزمون‌های مرحله‌ای	۴۵
۵-۳	تهیه نسخه مرجع ریزپردازنده جهت انجام آزمون	۴۷
۵-۴	یکپارچه‌سازی مرحله به مرحله	۴۸
۵-۵	جمع‌بندی	۴۸

۶	صحت‌سنجی ریزپردازنده‌ی هدف	۵۰
۶-۱	آزمون اولیه ریزپردازنده با برنامه‌های محک کوچک	۵۱
۶-۲	آزمون‌های کامل ریزپردازنده با بسته‌ی محک MiBench	۵۲
۶-۲-۱	بسته‌ی محک MiBench	۵۲
۶-۲-۲	اجرای برنامه‌های منتخب MiBench روی ریزپردازنده	۵۲
۶-۳	استفاده از واحد ممیز شناور برای تسریع در اجرای آزمون‌ها	۵۴
۶-۴	استفاده از سیستم‌عامل	۵۵
۶-۵	نحوه مقایسه نتایج	۵۶
۶-۶	شبیه‌ساز TSIM	۵۶
۶-۷	محاسبه پوشش کد	۵۷
۷	ارزیابی قابلیت اطمینان در ریزپردازنده‌ی هدف	۵۹
۷-۱	روش‌های ارزیابی قابلیت اطمینان	۶۰
۷-۱-۱	روش‌های تحلیلی	۶۰
۷-۱-۲	روش‌های تجربی	۶۱
۷-۲	مدل اشکال SEU	۶۴
۷-۳	برنامه‌های مقصد تزریق اشکال	۶۵
۷-۴	فرآیند تزریق اشکال	۶۷
۷-۵	تزریق اشکال بر روی ریزپردازنده اولیه	۷۲
۸	نتیجه‌گیری	۷۸

۸۲	.....مراجع
۸۵	.....پیوست‌ها
۸۶	..... ۱ پیوست ۱: نحوه‌ی اجرای سیستم‌عامل RTEMS روی ریزپردازنده SPARC
۸۶	..... ۱-۱ نصب
۸۷	..... ۱-۲ نحوه ایجاد Makefile و loader.c
۹۱	..... ۱-۳ استفاده از سیستم‌عامل در ModelSim
۹۳	..... ۲ پیوست ۲: آماده‌سازی پروژه LEON
۱۰۰	..... ۳ پیوست ۳: اسکریپت‌ها
۱۰۰	..... ۳-۱ اسکریپت‌های صحت‌سنجی
۱۰۰	..... ۳-۱-۱ اسکریپت gold.do
۱۰۲	..... ۳-۱-۲ اسکریپت allgolds.do
۱۰۲	..... ۳-۱-۳ اسکریپت check.do
۱۰۵	..... ۳-۱-۴ اسکریپت testall.do
۱۰۶	..... ۳-۲ اسکریپت lcompile.do
۱۰۸	..... ۳-۳ اسکریپت تزریق اشکال
۱۰۸	.....واژه‌نامه‌ی انگلیسی به فارسی



## فهرست جدول‌ها

### صفحه

جدول ۱-۲ انواع سیستم‌های نهفته .....	۸
جدول ۲-۲ مثال‌هایی از ریزپردازنده‌های نهفته .....	۱۰
جدول ۱-۴ دستورات مهم در واحد ممیز شناور معماری SPARC .....	۳۰
جدول ۱-۵ لیست فایل‌های تغییر یافته در واحدهای مختلف .....	۴۹
جدول ۱-۶ لیست برنامه‌های محک MiBench .....	۵۲
جدول ۲-۶ مقایسه سیستم‌عامل‌های نهفته .....	۵۵
جدول ۱-۷ پوشش کد واحد اعداد صحیح .....	۶۶
جدول ۲-۷ پوشش کد حافظه نهان داده .....	۶۶
جدول ۳-۷ نتایج تزریق اشکال برای Quicksort .....	۷۳
جدول ۴-۷ نتایج تزریق اشکال برای Basicmath .....	۷۳
جدول ۵-۷ نتایج تزریق اشکال برای Bitcount .....	۷۳
جدول ۶-۷ میانگین نتایج تزریق اشکال برای هر سه برنامه .....	۷۴
جدول پ-۱ تغییرات لازم در فایل device.vhd .....	۹۱

## فهرست شکل‌ها

## صفحه

شکل ۱-۲ ساختار کلی یک سیستم نهفته.....	۴
شکل ۲-۲ پیکربندی‌های مختلف نرم‌افزار در سیستم‌های نهفته.....	۵
شکل ۳-۲ مقایسه بازار ریزپردازنده‌ها.....	۶
شکل ۱-۳ طرز قرارگیری ثبات‌ها در ریزپردازنده SPARC نسخه ۸.....	۱۶
شکل ۲-۳ پنجره‌های ثبات.....	۱۷
شکل ۳-۳ پنجره‌های ثبات به صورت حلقوی.....	۱۸
شکل ۴-۳ مراحل خطلوله در ریزپردازنده SPARC.....	۱۹
شکل ۵-۳ قالب کلی دو نوع آدرس‌دهی در SPARC.....	۲۱
شکل ۶-۳ نمودار بلوکی اجزاء ریزپردازنده LEON.....	۲۴
شکل ۷-۳ شمایی از روش به کار رفته در واحد کنترل.....	۲۵
شکل ۸-۳ روش Partial Clock Gating.....	۲۷
شکل ۱-۴ شمایی از واحد ممیز شناور مرجع.....	۳۳
شکل ۲-۴ شمایی از واحد ممیز شناور نهایی.....	۳۶
شکل ۳-۴ شمایی از بخش درونی یک واحد محاسباتی در روش ارائه شده.....	۴۳
شکل ۱-۵ پایه‌های ریزپردازنده‌ی تحمل‌پذیر اشکال.....	۴۵
شکل ۱-۶ روند کلی مقایسه نتایج شبیه‌سازی.....	۵۶
شکل ۲-۶ مراحل طراحی تا اطمینان از جامعیت آزمون‌ها.....	۵۸
شکل ۱-۷ فرآیند کلی تزریق اشکال.....	۶۷
شکل ۲-۷ کشف خرابی زمانی بر حسب زمان برای حافظه نهان داده.....	۷۶
شکل ۳-۷ کشف خرابی زمانی بر حسب زمان برای حافظه نهان داده.....	۷۶
شکل پ-۱ تنظیم لازم برای شبیه‌سازی ریزپردازنده در ModelSim.....	۹۲

- شکل پ- ۲ ایجاد پروژه در ModelSim ..... ۹۴
- شکل پ- ۳ افزودن دو پوشه‌ی tbench و leon به پروژه ..... ۹۴
- شکل پ- ۴ افزودن فایل‌ها و ارجاع دادن به آن‌ها ..... ۹۴
- شکل پ- ۵ اجرای فرمان make برای ساخت پروژه ..... ۹۶
- شکل پ- ۶ شبیه‌سازی تنظیم tbleon در ModelSim ..... ۹۶
- شکل پ- ۷ ساختار نهایی فایل‌ها در پروژه‌ی شبیه‌سازی ..... ۹۷
- شکل پ- ۸ شمایی از سیگنال‌های اضافه شده پس از شبیه‌سازی ..... ۹۹

# فصل ۱

## ۱ مقدمه

کاربرد سیستم‌های نهفته روز به روز در حال افزایش است. این سیستم‌ها نقش بسیار مهمی در زندگی افراد ایفا می‌کنند. سالانه حدود ۷۹ تا ۹۹ درصد تولید ریزپردازنده‌ها را ریزپردازنده‌های نهفته تشکیل می‌دهند [Barr\_'06] [Marwede1\_'06]. این معیار نماینده‌ی آن است که کامپیوترهای همه منظوره، کاربردی به مراتب محدودتر از سیستم‌های نهفته – که امروزه در همه جا موجودند – دارند.

با افزایش استفاده از سیستم‌های نهفته و میزان وابستگی انسان به این سیستم‌ها، مؤلفه‌های دیگری نیز در انتخاب و طراحی مطرح شدند. یکی از این عوامل، قابلیت اطمینان است. سیستم‌هایی که در کاربردهای بحرانی – امن به کار می‌روند، با توجه به وظیفه‌ی مهمشان، باید قابلیت اطمینان بالایی داشته باشند. به عنوان نمونه می‌توان سیستم‌های نهفته‌ی یک بیمارستان، آسانسور یا یک فضاپیما را در نظر گرفت. از کار افتادن یا ایجاد وقفه در عملکرد این سیستم‌ها مشکلات بزرگی را ایجاد می‌کند.

در این پایان‌نامه، مراحل طی شده برای تهیه یک ریزپردازنده‌ی نهفته‌ی تحمل‌پذیر اشکال شرح داده شده‌اند. برای ساخت این ریزپردازنده، از واحدهای تحمل‌پذیر اشکال از پیش طراحی شده، استفاده شده است. همچنین واحد ممیز شناور سازگار با استاندارد IEEE-754 طراحی و پیاده‌سازی شده و دو روش تحمل‌پذیری اشکال برای عملیات دقت ساده و دقت مضاعف در این واحد ارائه شده است. این واحد با واحدهای قبلی یکپارچه شده‌اند. برای اطمینان از صحت یکپارچه‌سازی، از روش‌های صحت‌سنجی استفاده شده است. پس از آن، برای ارزیابی تحمل‌پذیری اشکال، آزمایش‌های تزریق اشکال به ریزپردازنده با استفاده از مدل اشکال SEU، انجام شده و میزان اشکالات پوشش داده شده، اشکالات منجر به خرابی و اشکالات متأخر اندازه‌گیری شده است.

در ادامه، مراحل مختلف پروژه شرح داده شده‌اند. در فصل دوم، سیستم‌های نهفته معرفی شده و اهمیت قابلیت اطمینان در آن‌ها بررسی شده است. فصل سوم، به معرفی ریزپردازنده هدف، معماری آن و واحدهای تحمل‌پذیر اشکال استفاده شده در آن می‌پردازد. واحد ممیز شناور ساخته شده همراه با دو روش تحمل‌پذیری اشکال آن، در فصل چهارم بررسی شده است. عملیات یکپارچه‌سازی ریزپردازنده و مراحل آن در فصل ۵ تشریح شده است. فصل ۶ نحوه صحت‌سنجی ریزپردازنده را به وسیله برنامه‌های محک استاندارد توضیح می‌دهد. در فصل ۷، مراحل ارزیابی قابلیت اطمینان برای ریزپردازنده هدف، مدل اشکال استفاده شده و نتایج حاصل از ارزیابی بیان شده‌اند.

## فصل ۲

### ۲ سیستم‌ها و ریزپردازنده‌های نهفته

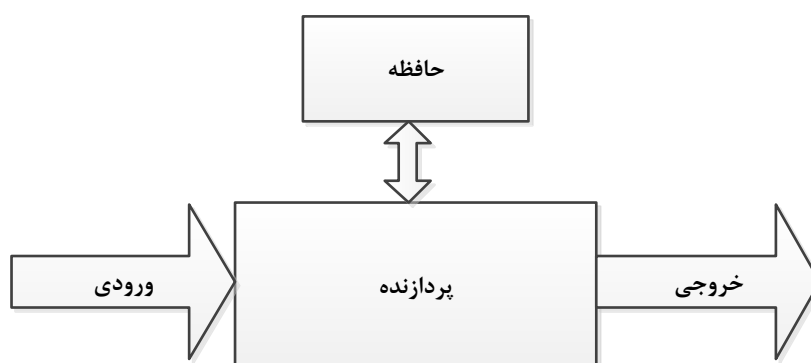
#### ۲-۱ سیستم‌های نهفته

یک سیستم نهفته، مجموعه‌ای است از سخت‌افزار و نرم‌افزار است که برای انجام وظیفه‌ای خاص بکار می‌رود. فلسفه طراحی سیستم نهفته با فلسفه طراحی کامپیوتر شخصی متفاوت است. سیستم نهفته برای انجام وظیفه‌ای خاص طراحی شده ولی کامپیوتر شخصی برای انجام وظیفه‌ای خاص طراحی نشده و قادر به انجام وظایف متعددی است [Barr\_'06]. همچنین این سیستم‌ها نسبت به سایر سیستم‌های کامپیوتری نیازمندی‌های کیفی و اتکاپذیری بالاتری دارند [Noergaard\_'05].

به عنوان مثالی از سیستم‌های نهفته می‌توان به ساعت مچی دیجیتال، پخش‌کننده‌های موسیقی، تلویزیون و گوشی‌های موبایل اشاره کرد. دو مثال دیگر از سیستم‌های نهفته، سیستم‌های فرود و کنترل

مریخ‌پیمای Pathfinder هستند که هر یک دارای نیازمندی‌های خاص خود و قابلیت‌های متناسب با آن نیازمندی‌ها است (سیستم فرود یک ریزپردازنده ۳۲-بیتی و ۱۲۸ مگابایت حافظه اصلی و سیستم کنترل یک ریزپردازنده ۸-بیتی و ۵۱۲ کیلوبایت حافظه اصلی داشت [Barr\_'06]).

اگرچه سیستم‌های نهفته می‌توانند ساختمان و کاربردهای بسیار متنوع و متفاوتی داشته باشند اما قسمتی از ساختار این سیستم‌ها مشترک است. یک سیستم برای آن که نهفته باشد باید حداقل از یک ریزپردازنده و نرم‌افزار تشکیل شده باشد [Barr\_'06]. به علاوه برای نگهداری برنامه، حافظه‌ای لازم است و حافظه‌ای نیز برای کارهای اجرایی برنامه لازم است که این نیاز به عنوان مثال می‌تواند توسط یک حافظه فقط خواندنی (ROM) و یک حافظه با دسترسی تصادفی (RAM) تأمین شود. در شکل ۱-۲ ساختار کلی یک سیستم نهفته مشاهده می‌شود.



شکل ۱-۲ ساختار کلی یک سیستم نهفته [Barr\_'06]

نرم‌افزار سیستم نهفته به شدت به کاربرد بستگی دارد و می‌تواند بسیار ساده و یا بسیار پیچیده باشد. در شکل ۲-۲ دو پیکربندی مختلف برای برنامه‌های یک سیستم نهفته دیده می‌شود. پیکربندی الف، یک سیستم نهفته‌ی ساده و حداقلی را نشان می‌دهد در حالی که سیستم ب، یک سیستم پیچیده را به تصویر کشیده است.

برنامه‌ی کاربردی		برنامه کاربردی	
پشته‌ی شبکه	سیستم‌عامل بی‌درنگ	راه‌اندازهای ابزار	
راه‌اندازهای ابزار		سخت‌افزار	
سخت‌افزار			

ب) سیستم نهفته پیچیده

الف) سیستم نهفته ساده

شکل ۲-۲ پیکربندی‌های مختلف نرم‌افزار در سیستم‌های نهفته [Barr\_'06]

## ۲-۲ اهمیت، تاریخچه و آینده سیستم‌های نهفته

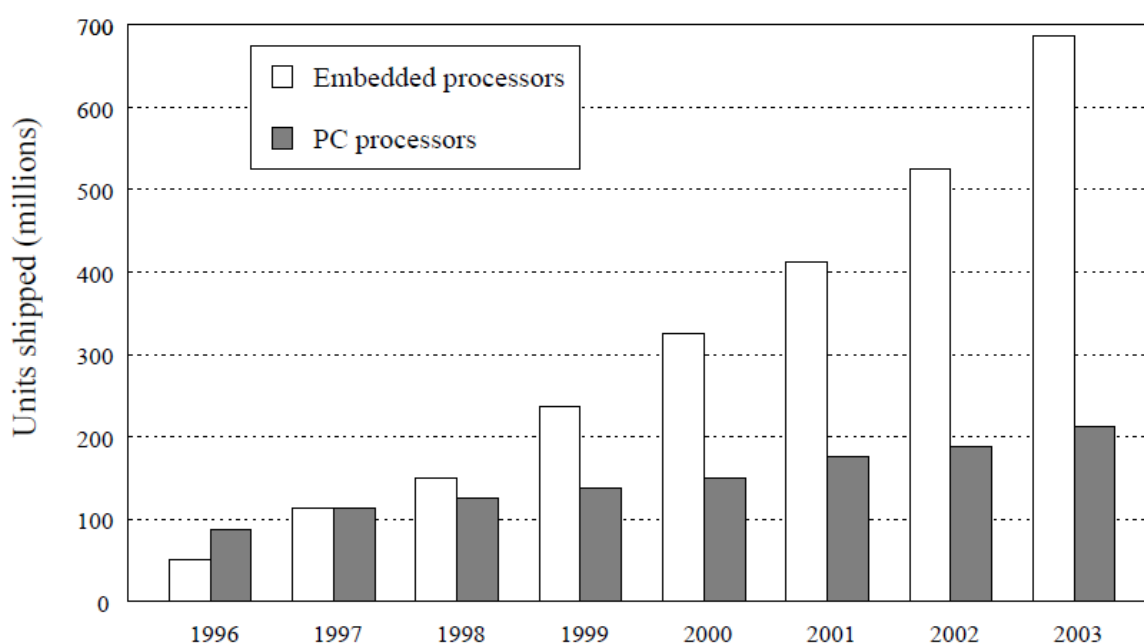
با توجه به تعریفی که برای سیستم‌های نهفته ذکر شد، ساخت و استفاده‌ی این سیستم‌ها قدمت زیادی ندارند. در سال ۱۹۷۱، اینتل اولین ریزپردازنده تک‌تراشه‌ی دنیا، یعنی تراشه ۴۰۰۴ را به سفارش شرکت Busicom تولید کرد. این ریزپردازنده عام‌منظوره بود و می‌توانست در انواع ماشین‌حساب‌ها استفاده شود. تفاوت ماشین‌حساب‌های مختلف در مقدار حافظه نصب شده روی آن‌ها و برنامه‌ی مربوط به مدل ماشین حساب بود [Barr\_'06].

موفقیت سیستم‌های نهفته به قدری بود که تولید آن در چند دهه بعد به طور نمایی افزایش یافت. امروزه این سیستم‌ها در همه جا هستند. در آشپزخانه (اجاق ماکروویو و ماشین لباس‌شوئی)، در اتاق نشیمن (تلویزیون، کنترل از راه‌دور، دستگاه‌های پخش فیلم و صدا، کنسول‌های بازی)، در محل کار (سیستم‌های حضور و غیاب، چاپگرها، دستگاه‌های پول‌شمار) نمونه‌های بسیاری از این سیستم‌ها بکار می‌رود.



مردم علاقه‌ی زیادی به ابزارهای دیجیتالی پیدا کرده‌اند. ابزارهای لوکسی که امروز مطرح هستند –مانند اتومبیل دوگانه سوز تویوتا پریوس که شبکه‌ای از ریزپردازنده‌های ۳۲-بیتی دارد– نشان می‌دهد که سیستم‌های نهفته آینده‌ی درخشانی خواهند داشت.

در شکل ۲-۳، روند رشد بازار ریزپردازنده‌های نهفته در مقایسه با ریزپردازنده‌های همه‌منظوره که در کامپیوترهای شخصی کاربرد دارند، آمده است.



شکل ۲-۳ مقایسه بازار ریزپردازنده‌ها [Wong\_'04]

همانطور که دیده می‌شود، روند استفاده از ریزپردازنده‌های نهفته، به طور نمایی در حال افزایش است. به طوری که در سال ۲۰۰۶، حدود ۶ میلیارد ریزپردازنده جدید تولید شدند. کمتر از ۲ درصد (حدود ۱۰۰ میلیون) از این ریزپردازنده‌ها مخصوص کامپیوترهای همه‌منظوره بودند [Barr\_'06].

## ۲-۳ کاربردهای سیستم‌های نهفته

گستره‌ی کاربرد سیستم‌های نهفته شامل تمام مواردی است که در آن‌ها لازم است سیستم‌های رایانه‌ای به شکلی خاص به کار گرفته شوند. در [Marwedel\_06] برخی از کاربردهای سیستم‌های نهفته به شکل زیر بیان شده است:

**صنعت خودرو:** خودروهای جدید دارای بخش‌های مختلف الکترونیکی مانند سیستم‌های کنترل موتور، کنترل کیسه هوا، ترمز ضد قفل، سیستم هشدار و تهنویه هستند.

**صنایع هوایی:** سیستم‌های الکترونیکی و واحدهای پردازشی زیادی در هواپیماها وجود دارند که همگی باید اتکاپذیر باشند.

**قطارها:** قطارها نیز مانند خودروها و هواپیماها دارای بخش‌های الکترونیکی زیادی هستند که همگی باید دارای ایمنی<sup>۱</sup> کافی باشند.

**مخابرات:** در مخابرات نیز بارزترین نمود سیستم‌های نهفته تلفن‌های همراه هستند.

**سیستم‌های پزشکی:** بسیاری از وسایل پیشرفته‌ی پزشکی به وسیله‌ی سیستم‌های پردازش اطلاعاتی که در درون آن‌ها نهفته شده است، امکان ارائه خدمات را دارند.

**کاربردهای نظامی:** یکی از اولین کاربردهای سیستم‌های کامپیوتری در پردازش اطلاعات نظامی بوده است.

**سیستم‌های احراز هویت:** در کاربردهای احراز هویت (مانند تشخیص اثر انگشت) نیز از سیستم‌های نهفته استفاده می‌شود.

---

<sup>۱</sup> Safety

**وسایل خانگی:** در وسایل مختلف خانگی مانند تلویزیون‌ها و سیستم‌های پخش صوت و نیز وسایل بازی الکترونیکی، از سیستم‌های نهفته استفاده می‌شود.

**ابزار تولید:** برای کنترل فرآیند تولید در کارخانجات به طور گسترده از سیستم‌های نهفته استفاده می‌شود.

**خانه‌های هوشمند:** برای افزایش ایمنی، امنیت، کنترل گرما و نور و کاهش مصرف انرژی از سیستم‌های نهفته در خانه‌های هوشمند استفاده می‌شود.

**رباتیک:** ربات‌ها نیز یکی از موارد قدیمی کاربرد سیستم‌های نهفته هستند که در آن‌ها جنبه کنترل مکانیکی بسیار مهم است.

## ۲-۴ قابلیت اطمینان در سیستم‌های نهفته

قابلیت اطمینان، بر اساس تعریف، احتمال آن است که سیستم تا زمان  $t$ ، به درستی کار کند. به عبارتی می‌توان این تعریف را با رابطه‌ی  $R(t) = Prob\{T > t\}$  نیز نمایش داد [Koren\_'07].

سیستم‌های نهفته مختلف، علاوه بر توان پردازشی، حافظه، تعداد تولید، توان مصرفی، هزینه توسعه و طول عمر، به قابلیت اطمینان متفاوتی احتیاج دارند. به عنوان مثال یک اسباب‌بازی قابلیت اطمینان بالایی نمی‌خواهد. از طرفی سیستم ضد قفل یک اتومبیل، باید کاملاً مطمئن باشد و در تمام شرایط کار کند. در نتیجه بسته به نوع سیستم، باید قابلیت‌های اطمینان مختلفی برای آن در نظر گرفت.

با توجه به نیازهای سیستم‌های نهفته مختلف، می‌توان آن‌ها را به سه گروه ضعیف، متوسط و قوی تقسیم کرد و آن‌ها را با توجه به نیازهایشان با هم مقایسه کرد. این مقایسه در جدول ۲-۱ آمده است:

جدول ۲-۱ انواع سیستم‌های نهفته [Barr\_'06]

معیار	ضعیف	متوسط	قوی
-------	------	-------	-----

ریزپردازنده	۴ یا ۸ بیتی	۱۶ بیتی	۳۲ یا ۶۴ بیتی
حافظه	کمتر از ۶۴ KB	۶۴ KB تا ۱ MB	بیشتر از ۱ MB
هزینه‌ی توسعه	کمتر از \$۱۰۰,۰۰۰	\$۱۰۰,۰۰۰ تا \$۱,۰۰۰,۰۰۰	بیشتر از \$۱,۰۰۰,۰۰۰
هزینه‌ی تولید	کمتر از \$۱۰	\$۱۰ تا \$۱,۰۰۰	بیشتر از \$۱,۰۰۰
تعداد دستگاه‌ها	کمتر از ۱۰۰	۱۰۰ تا ۱۰,۰۰۰	بیشتر از ۱۰,۰۰۰
توان مصرفی	بیشتر از ۱۰ mW/MIPS	۱۰ mW/MIPS تا ۱۰	کمتر از ۱۰ mW/MIPS
عمر	روزها، هفته‌ها یا ماه‌ها	سال‌ها	چندین دهه
قابلیت اطمینان	گاهی اوقات خراب می‌شود	باید مطمئن کار کند	مقاوم در برابر خرابی

همانطور که در جدول آمده، قابلیت اطمینان بالا، معمولاً برای سیستم‌هایی در رده‌ی قوی در نظر گرفته می‌شود. با این حال، بسیاری از طراحان سیستم‌های نهفته، برای حفظ اعتبار خود، ناچارند سیستم نهفته‌ی خود را مطمئن طراحی کنند.

## ۲-۵ ریزپردازنده‌های نهفته

پردازنده، اصلی‌ترین قسمت یک سیستم نهفته به حساب می‌آید. بر حسب پیچیدگی این پردازنده‌ها، می‌توان آن‌ها را به دو دسته ریزپردازنده‌ها و میکروکنترلرها تقسیم کرد. به طور کلی، ریزپردازنده‌ها حافظه‌ای اندک و تجهیزات I/O دارند؛ اما در میکروکنترلرها، تقریباً همه‌ی حافظه و تجهیزات I/O سیستم را روی تراشه جای قرار گرفته است [Noergaard\_'05].

صدها ریزپردازنده‌ی مختلف برای سیستم‌های نهفته تولید شده‌اند که هیچکدام به تنهایی بر بازار این ریزپردازنده‌ها مسلط نیستند. این ریزپردازنده‌ها را می‌توان به معماری دستورات عمل‌های<sup>۱</sup> مختلف تقسیم کرد. ریزپردازنده‌های با معماری مشابه باید بتوانند دستورات مشخصی را اجرا کنند. در جدول ۲-۲ لیستی از ریزپردازنده‌های نهفته رایج آورده شده است.

<sup>۱</sup> Instruction Set Architecture (ISA)

جدول ۲-۲ مثال‌هایی از ریزپردازنده‌های نهفته؛ برگرفته از [Noergaard\_05]

تولیدکننده	ریزپردازنده	معماری دستورالعمل
Advanced Micro Devices,...	Au1xxx	AMD
ARM,...	ARM7, ARM9...,	ARM
Infineon,...	C167CS, C165H, C164CI,...	C16X
Motorola/Freescale,...	5282, 5272, 5307, 5407,...	ColdFire
Vmetro,...	I960	I960
Renesas/Mitsubishi,...	32170, 32180, 32182, 32192,...	M32/R
Motorola/Freescale	MMC2113, MMC2114,...	M Core
MTI4kx, IDT, MIPS Technologies,...	R3K, R4K, 5K, 16, ...	MIPS32
NEC Corporation,...	Vr55xx, Vr54xx, Vr41xx	NEC
IBM, Motorola/Freescale,...	82xx, 74xx, 8xx, 7xx, 6xx, 5xx, 4xx	PowerPC
Motorola/Freescale,...	680x0 (68K, 68030, 68040, 68060,...), 683xx	68k
Hitachi,...	SH3 (7702, 7707, 7708, 7709), SH4 (7750)	SuperH (SH)
Analog Devices, Transtech DSP, Radstone,...	SHARC	SHARC
Intel,...	strongARM	strongARM
Sun Microsystems,...	UltraSPARC II, LEON 2,...	SPARC
Texas Instruments,...	TMS320C6xxx	TMS320C6xxx
Intel, Transmeta, National Semiconductor, Atlas,...	X86 [386, 486, Pentium (II, III, IV)...]	x86
Infineon,...	TriCore1, TriCore2,...	TriCore

در این پایان‌نامه، از بین ریزپردازنده‌های نهفته‌ی ذکر شده، ریزپردازنده SPARC نسخه ۸ انتخاب شده که در فصل بعد به علت این انتخاب پرداخته خواهد شد.

## ۲-۶ سیستم‌عامل‌های نهفته

استفاده از سیستم‌عامل در سیستم‌های نهفته با دو هدف انجام می‌شود. هدف نخست فراهم کردن لایه‌ای نرم‌افزاری است که به سایر نرم‌افزارها امکان می‌دهد که راحت‌تر ساخته شوند و همچنین از جزئیات سخت‌افزار آزادتر باشند. هدف دوم، مدیریت منابع سخت‌افزاری و نرم‌افزاری سیستم است به

نحوی که کارایی و قابلیت اطمینان لازم به دست آید [Noergaard\_'05]. با توجه به این که سیستم‌های نهفته غالباً در کاربردهای بی‌درنگ به می‌روند، سیستم‌عامل‌های نهفته، بی‌درنگ نیز هستند.

**استفاده یا عدم استفاده از سیستم‌عامل:** در بیشتر مواقع، پاسخ قطعی به این سؤال وجود ندارد. بسیاری از سیستم‌های نهفته می‌توانند کارشان را دقیقاً مطابق نیاز، با استفاده از یک حلقه بی‌پایان انجام دهند (و انجام هم می‌دهند). این سیستم‌های نهفته نیازی ندارند با افزودن نرم‌افزار اضافی (مانند یک سیستم‌عامل بی‌درنگ) پیچیده شوند. پیچیده‌تر کردن یک سیستم نهفته، هیچ سودی ندارد. هر سیستم بسته به مورد کاربرد آن، باید مستقلاً ارزیابی شود. اصولاً فرض اولیه آن است که وجود سیستم‌عامل لازم نیست. اما با بررسی نیازهای مازول‌های مختلف نرم‌افزاری، این فرض می‌تواند تغییر کند [Barr\_'06].

**فرآیند انتخاب سیستم‌عامل نهفته:** در هنگام انتخاب یک سیستم‌عامل نهفته باید عوامل متعددی را در نظر گرفت. در ادامه مهمترین این عوامل آمده است [Barr\_'06]:

- پشتیبانی ریزپردازنده: در این پروژه، سیستم‌عامل نهفته، باید از معماری SPARC نسخه ۸ پشتیبانی کند. همین عامل باعث می‌شود تا انتخاب سیستم‌عامل محدودتر شود.
- ویژگی‌های بی‌درنگ: سیستم‌عامل باید از ویژگی‌های بی‌درنگی که برنامه به آن نیاز دارد، پشتیبانی کند.
- محدودیت‌های مالی: قیمت سیستم‌عامل‌ها طیف وسیعی را شامل می‌شود: از سیستم‌عامل‌های منبع باز و رایگان تا سیستم‌عامل‌های ده‌ها هزار دلاری به ازای هر توسعه‌دهنده به اضافه‌ی حق تألیف هر واحدی که فروخته می‌شود. با توجه به محدودیت‌هایی که در این پروژه وجود داشت، از سیستم‌عامل‌های با منبع باز استفاده شد.

- **استفاده از حافظه:** سیستم‌عامل‌های معدودی می‌توانند در سیستم‌های نهفته‌ی کوچک جا شوند (مثلاً، با حذف قابلیت‌های آن برای کوچکتر کردن اندازه‌ی آن). سیستم‌عامل‌های دیگر به منابع اندکی (مشابه یک PC ضعیف) نیاز دارند. به علاوه چون در اینجا از شبیه‌سازی با دقت بالا (به ازای سیکل‌های کلاک) انجام می‌شود و منابع محدود است، بایستی گزینه‌ای را انتخاب کنیم که کمترین میزان حافظه را اشغال کند.
- **پشتیبانی فنی:** اغلب سیستم‌عامل‌های نهفته‌ی منبع باز<sup>۱</sup>، یک فهرست پستی<sup>۲</sup> یا انجمن<sup>۳</sup> دارند که مشکلات کاربران از طریق آن برطرف می‌شود. با توجه به رایگان بودن این سرویس‌ها در مقابل پشتیبانی فنی سیستم‌عامل‌های تجاری یا پشتیبانی تجاری، سیستم‌عامل‌های نهفته رایگان ترجیح داده می‌شوند.
- **سازگاری ابزار:** سیستم‌عامل استفاده شده باید با اسمبلر، کامپایلر، لینکر و اشکال زدایی که پیش از این استفاده می‌شده هماهنگ باشد و عملکرد مستقل یا کاملاً متفاوتی با آن نداشته باشد. در نتیجه می‌توان از روش‌هایی که قبلاً برای کامپایل و اجرای برنامه‌ها استفاده می‌شد، استفاده کرد.

---

<sup>۱</sup> Open Source

<sup>۲</sup> Mailing List

<sup>۳</sup> Forum

## فصل ۳

### ۳ تحمل پذیری اشکال در ریزپردازنده‌ی هدف

چنانکه گفته شد، معماری SPARC به عنوان معماری ریزپردازنده هدف انتخاب شد. در این فصل ابتدا این معماری معرفی می‌شود. سپس ریزپردازنده‌ی LEON 2 به عنوان یک پیاده‌سازی از این معماری که پایه‌ی انجام این پروژه قرار گرفته است، بررسی می‌شود. در نهایت روش‌های تحمل‌پذیری اشکال به کار رفته در اجزاء ریزپردازنده‌ی هدف، مرور شده‌اند.

#### ۳-۱ معماری SPARC نسخه ۸

معماری SPARC توسط شرکت Sun و بر اساس استاندارد RISC II که در دانشگاه برکلی طراحی شده بود، ساخته شد. این طراحی RISC بر مبنای حداقل‌سازی تعداد opcodeها صورت گرفته بود تا



اجرای هر دستورالعمل، به حدود یک چرخه<sup>۱</sup> نیاز داشته باشد. SPARC مخفف Scalable Processor ARChitecture به معنای «پردازنده با معماری مقیاس پذیر» می باشد. بر خلاف شرکت های دیگر، Sun معماری باز<sup>۲</sup> را برای SPARC برگزید و به تنهایی اقدام به تولید SPARC نکرد. در حقیقت SPARC استاندارد در سطح معماری دستورالعمل می باشد؛ در نتیجه دست تولیدکنندگان در ایجاد نوآوری و تغییر در پیاده سازی ریزپردازنده بر اساس SPARC باز است [Dandamudi\_'05].

در سال ۱۹۸۷ شرکت Sun مشخصات اولین ریزپردازنده ی خود با عنوان SPARC نسخه ۷ را منتشر ساخت. این ریزپردازنده ی ۳۲-بیتی توسط شرکت های Sun و Fujitsu پیاده سازی شد. مشخصات SPARC نسخه ۸ در سال ۱۹۹۰ منتشر شد. این ریزپردازنده هنوز ۳۲-بیتی بود و بر اساس مدل Big Endian کار می کرد. در نهایت SPARC نسخه ۹ در سال ۱۹۹۳ معرفی گردید. در سال ۱۹۹۵، Sun با ریزپردازنده ی UltraSPARC و Fujitsu با SPARC64 ایستگاه های کاری ۶۴-بیتی را عرضه کردند [Dandamudi\_'05].

مطالب ادامه ی این بخش از [SPARC International Inc.\_'92] استخراج شده اند و به معرفی ساختار ثبات ها، خطلوله، دستورات ریزپردازنده و شیوه های آدرس دهی ریزپردازنده ی SPARC نسخه ۸ می پردازند.

### ۳-۱-۱ ثبات ها

در هر لحظه برنامه ی کاربر به ۳۲ ثبات ۳۲-بیتی همه منظوره با نام های r0 تا r31 دسترسی دارد. ثبات های همه منظوره ی r0 تا r7 به عنوان ثبات های همگانی<sup>۳</sup> استفاده می شوند. با قرار دادن متغیرهای همگانی در ثبات های همگانی می توانیم از دسترسی به حافظه هنگام فراخوانی و بازگشت از پروسه ها<sup>۴</sup>

<sup>۱</sup> Cycle

<sup>۲</sup> Open Architecture

<sup>۳</sup> Global Registers

<sup>۴</sup> Procedures

جلوگیری کنیم. (پردازنده‌ی Itanium نیز از روشی مشابه این روش استفاده می‌کند)

[Dandamudi\_'05]

ثبات‌های ورودی<sup>۱</sup> (r24 تا r31) شامل آرگومان‌های ارسالی به توابع هستند. ثبات‌های خروجی<sup>۲</sup> (r8 تا r15) برای مصارف موقتی، ارسال آرگومان به تابع و نگهداری مقادیر بازگشتی به کار می‌روند. ثبات‌های محلی<sup>۳</sup> (r16 تا r23) برای نگهداری مقادیر متغیرهای محلی توابع استفاده می‌شوند. شکل ۱-۳ نحوه‌ی ساماندهی این ثبات‌ها را نشان می‌دهد.

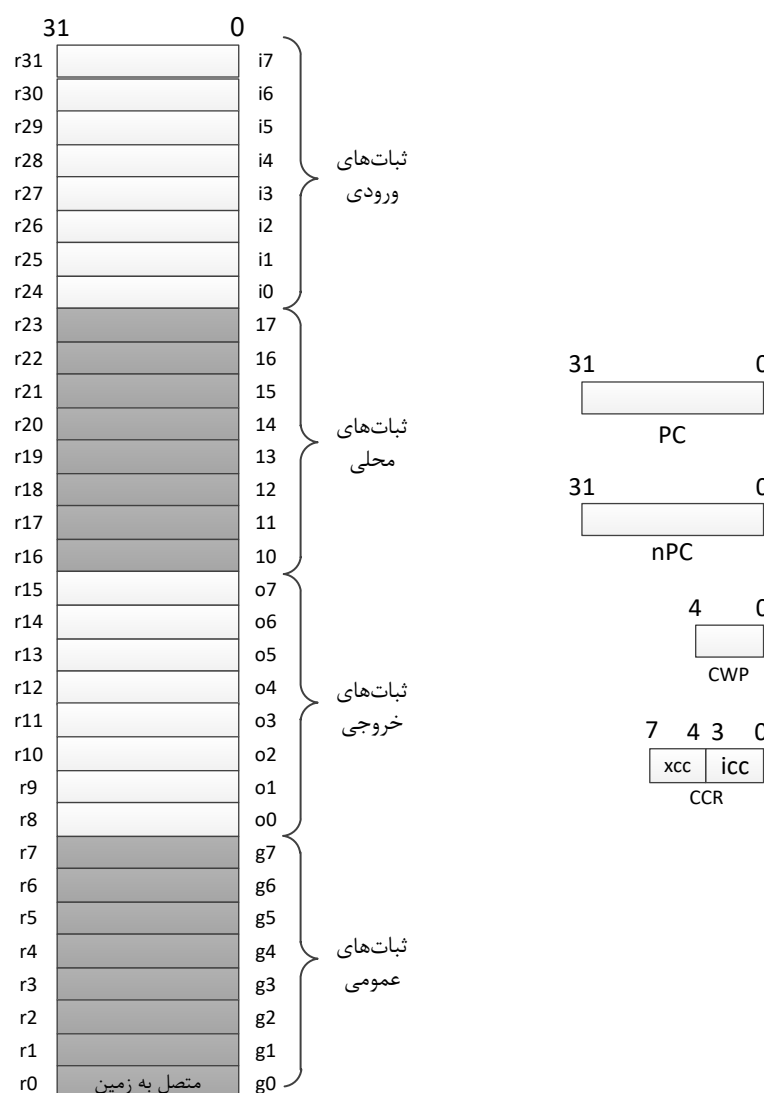
دو تا از ثبات‌های خروجی (o6 و o7) برای استفاده‌ی خاص رزرو شده‌اند و نباید از آن‌ها استفاده کرد. اولین ثبات عمومی نیز به طور سخت‌افزاری همیشه صفر می‌باشد. در نتیجه مقدار آن مستقل از عددی است که در آن می‌نویسیم. همه‌ی ثبات‌ها، عدد  $n$  را به صورت علامت‌دار ذخیره می‌کنند.

$$(-2^{31} \leq n \leq 2^{31})$$

<sup>1</sup> In Registers

<sup>2</sup> Out Registers

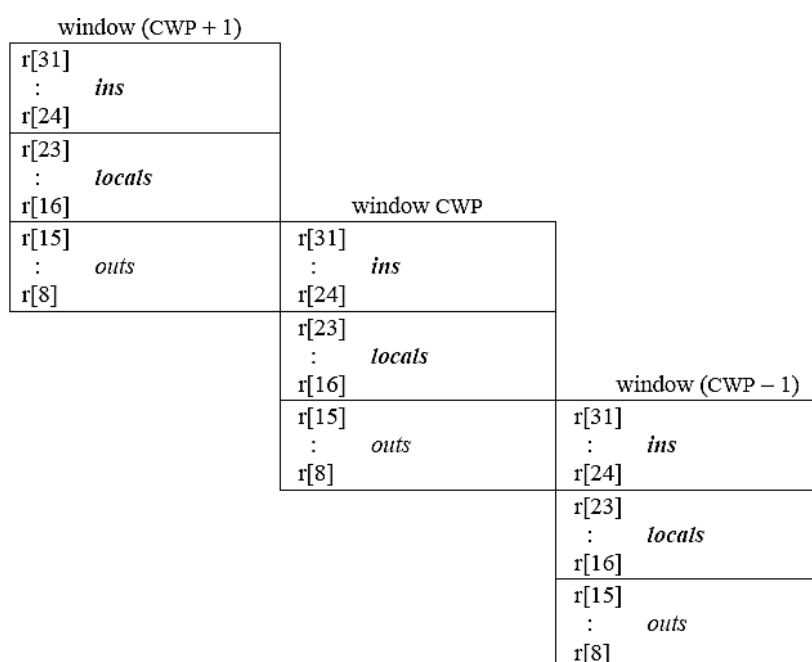
<sup>3</sup> Local Registers



شکل ۳-۱ طرز قرارگیری ثبت‌ها در ریزپردازنده SPARC نسخه ۸

ساختار SPARC به گونه‌ایست که پنجره‌های ثبت<sup>۱</sup> را بکار می‌برد. همانطور که در شکل ۳-۲ نشان داده شده است، هر پنجره، ۲۴ ثبت ورودی، خروجی و محلی دارد. ثبت‌های ورودی یک پنجره با ثبت‌های خروجی پنجره‌ی کناری هم‌پوشانی دارند. این کار باعث کاسته شدن از سربار مربوط به فراخوانی توابع می‌گردد. ثبت CWP (Current Window Pointer) اطلاعات مربوط به پنجره‌ی جاری را در خود نگه می‌دارد. با دو دستور restore و save مقدار CWP به ترتیب کاهش و افزایش می‌یابد.

<sup>۱</sup> Register Windows



r[7]
: <i>globals</i>
r[1]
r[0] 0

31 0

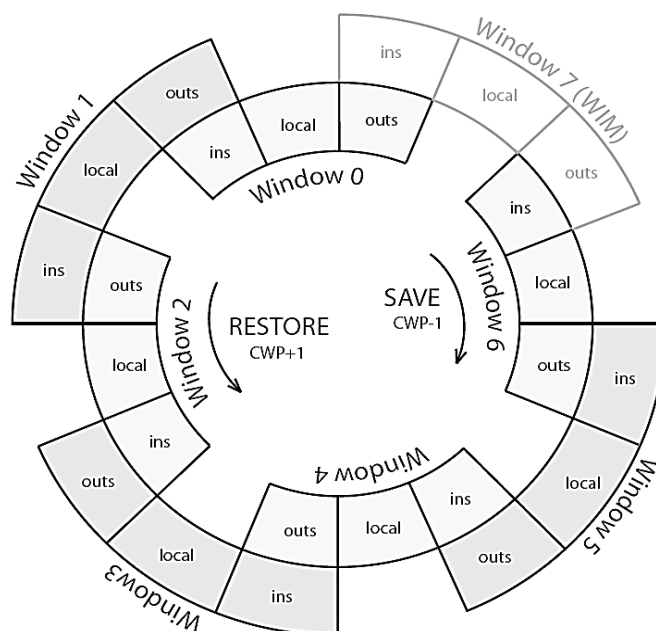
شکل ۳-۲ پنجره‌های ثابت

از مزایای پنجره‌های ثابت می‌توان به کاهش زحمت کامپایلرها در تخصیص ثابت‌ها، کاهش تعداد load و store لازم در کد تولید شده به وسیله کامپایلرها و امکان استفاده از حافظه‌ی خارجی اشاره کرد [Case\_'92]

تعداد پنجره‌ها بسته به نوع پیاده‌سازی می‌تواند ۲ تا ۳۲ باشد. تعداد کل ثابت‌ها برابر است با ۸ ثابت عمومی به اضافه‌ی تعداد پنجره‌ها  $\times ۱۶$ . در نتیجه تعداد ثابت‌ها می‌تواند بین ۴۰ (برای ۲ پنجره) تا ۵۲۰ ثابت (برای ۳۲ پنجره) باشد.

اگر این ثابت‌ها به صورت حلقوی کنار هم قرارگیرند (مشابه شکل ۳-۳) یک پنجره‌ی بی‌اعتبار ایجاد می‌گردد. (در این شکل، پنجره‌ی ۷) این پنجره توسط ثابت ۳۲-بیتی WIM (Window Invalid Mark) علامت‌گذاری می‌گردد. (هر بیت این ثابت وضعیت یک پنجره را معین

می‌سازد.) علت نامعتبر بودن پنجره‌ی هفتم آن است که ثابت‌های خروجی، ثابت‌های ورودی را رونویسی می‌کنند. با علامت‌گذاری این پنجره، در چنین مواقعی یک trap رخ می‌دهد.



شکل ۳-۳ پنجره‌های ثابت به صورت حلقوی

### ۳-۱-۲ خطلوله

پردازنده‌ی SPARC نیز مانند دیگر ریزپردازنده‌های RISC، از خطلوله استفاده می‌کند تا سرعت پردازش دستورات در آن افزایش یابد. برای نیل به این مقصود این ریزپردازنده، مراحل اجرای یک دستور را پنج قسمت می‌کند:

- ۱- **واکشی دستور:** اگر گش دستورالعمل‌ها فعال باشد، دستور از آن واکشی می‌شود. در غیر این صورت، درخواست واکشی به کنترل‌گر حافظه ارسال می‌شود. این دستور در پایان این مرحله آماده می‌شود و در داخل واحد کنترل، ذخیره می‌شود.

۲- **کدگشایی:** دستور کدگشایی شده و عملوندهای آن خوانده می‌شوند. عملوندها<sup>۱</sup> ممکن است از

بانک ثبات یا مقادیر داخلی ریزپردازنده باشند. آدرس‌های مقصد Call و دستورات انشعاب در

این مرحله تولید می‌شوند.

۳- **اجرا:** اجرای یک عملیات محاسباتی، محاسبه‌ی آدرس مقصد یک پرش یا محاسبه‌ی آدرس

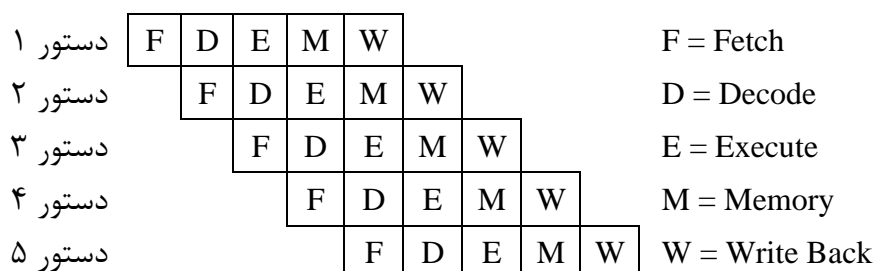
حافظه برای عمل خواندن یا ذخیره کردن.

۴- **دسترسی به حافظه:** به کش داده‌ها مراجعه می‌شود.

۵- **ذخیره‌ی نتیجه:** نتیجه هر عمل ALU، منطقی، شیفت یا خواندن از کش در ثبات‌ها نوشته

می‌شود [Gaisler\_'05].

شکل ۳-۴ مراحل خطلوله در SPARC را نشان می‌دهد.



شکل ۳-۴ مراحل خطلوله در ریزپردازنده SPARC

در عمل خطلوله دو جا به مشکل بر می‌خورد [Paul\_'00]:

• اگر دستور دوم وابسته به نتایجی باشد که دستور قبلی در حافظه قرار است ذخیره کند.

به عنوان مثال می‌توانید دو دستور زیر را در نظر بگیرید:

LOAD [%o0], %o1

ADD %o1, %o2, %o2

<sup>1</sup> Operand

خوشبختانه، در این صورت خود ریزپردازنده این حالت را تشخیص می‌دهد و یک سیکل انتظار بین دو دستور قرار می‌دهد. (برای جبران این زمان هدر رفته، می‌توان یک دستور غیروابسته را بین دو دستور بالا قرار داد.)

- هنگام به کار بردن دستورات پرش، دستور دوم هدر می‌رود. SPARC برای پوشش این حالت راه حل جالبی را بکاربرده است: استفاده از یک ثابت PC دیگر! این ریزپردازنده دو ثابت PC دارد. یکی با همان نام PC و دیگری با نام nPC (next Program Counter). nPC مقدار آدرس دستور بعدی را که قرار است اجرا گردد نگه می‌دارد. وقتی ریزپردازنده با دستور پرش مواجه شد، nPC به آدرس دستور مقصد اشاره می‌کند، اما با این حال همیشه دستور بعد از پرش اجرا می‌شود. (PC در اجرای آن به کار گرفته می‌شود) در نتیجه برای اجتناب از اتفاقات غیر منتظره، باید پس از هر دستور پرش، دستور  $\text{nop}^1$  را بکار برد.

### ۳-۱-۳ شیوه‌های آدرس‌دهی

ریزپردازنده‌ی SPARC تنها از دو شیوه‌ی آدرس‌دهی پشتیبانی می‌کند:

- **Register Indirect with Immediate**: آدرس مؤثر به صورت زیر محاسبه می‌شود:  

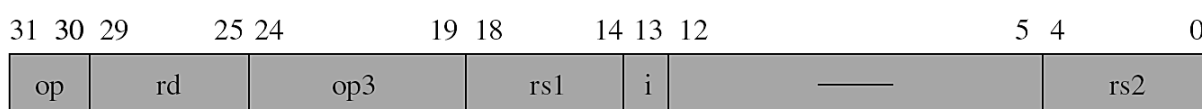
$$\text{مؤثر} = \text{Imm13} + \text{محتوای ثابت } R_x = \text{آدرس مؤثر}$$
 $R_x$  می‌تواند یکی از ثابت‌هایی که شرح داده شد، باشد. Imm13 نیز یک مقدار ۱۳-بیتی است که ابتدا sign extend می‌گردد.
- **Register Indirect with Index**: در این روش آدرس‌دهی آدرس مؤثر از جمع مقدار دو ثابت حاصل می‌شود.

<sup>1</sup> No Operation

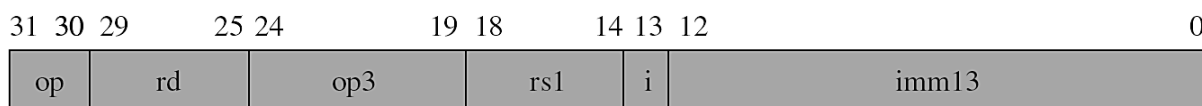
SPARC شیوهی آدرس‌دهی register indirect را پشتیبانی نمی‌کند. برای استفاده از این شیوه، باید در آدرس‌دهی نوع اول، صفر را به عنوان Imm13 استفاده نمود یا در آدرس‌دهی دوم، r0 را به عنوان یکی از دو ثبات، بکار برد. (دقت کنید که مقدار r0 همیشه صفر می‌باشد).

#### ۳-۱-۴ قالب دستورالعمل‌ها

همه‌ی دستورات SPARC، ۳۲-بیتی می‌باشند. در این دستورات از دو فیلد opcode استفاده می‌گردد: دو بیت پرارزش op (بیت‌های ۳۰ و ۳۱) گروهی را که دستور به آن تعلق دارد، مشخص می‌کند. فیلد دوم، یعنی op2 یا op3، خود دستورالعمل را معین می‌کند. در شکل ۳-۵ قالب کلی دو دستور، برای دو نوع آدرس‌دهی مختلف آمده است.



Register-register instructions (i = 0)



Register-immediate instructions (i = 1)

شکل ۳-۵ قالب کلی دو نوع آدرس‌دهی در SPARC [Dandamudi\_05]

#### ۳-۱-۵ معماری واحد مدیریت حافظه

بر طبق [SPARC International Inc.\_'92] پیاده‌سازی واحد مدیریت حافظه (Memory Management Unit) (MMU) اجباری نیست. اما در ضمیمه‌ی H این راهنما، SRMMU<sup>۱</sup> تعریف شده و توصیه شده که این واحد پیاده‌سازی گردد.

<sup>۱</sup> SPARC MMU



یکی از مزایای پیاده‌سازی سخت‌افزاری MMU، استانداردسازی است. اگر تولیدکنندگان از یک استاندارد MMU پیروی کنند، نوشتن و انتقال سیستم‌عامل از یک سیستم به سیستمی دیگر آسان‌تر می‌شود.

MMU از ترجمه‌ی آدرس سه مرحله‌ای استفاده می‌کند، مدخل‌های جداول صفحه نیز در MMU، cache شده تا مرحله‌ی ترجمه سریع‌تر گردد. این MMU ویژگی‌های زیر را دارد:

- آدرس مجازی ۳۲-بیتی
- آدرس فیزیکی ۳۶-بیتی
- اندازه‌ی ثابت 4KB برای هر صفحه
- آدرس‌دهی ۳-مرحله‌ای
- پشتیبانی از آدرس‌دهی خطی (با صفحات، برای 4KB، 256KB، 16MB و 4GB)
- پشتیبانی از چندین زمینه<sup>۱</sup>
- پردازش حالت miss به صورت سخت‌افزاری

وظائف MMU بدین شرح هستند:

۱- ترجمه‌ی آدرس مجازی به آدرس فیزیکی: این عمل با استفاده از صفحات 4KB انجام

می‌گیرد. در نتیجه، اگر برنامه‌ای 8MB فضا اشغال کند، نیازی نیست این 8MB به طور پیوسته در حافظه‌ی اصلی قرار داشته باشد.

۲- محافظت از حافظه: این عمل باعث می‌شود برنامه‌ها نتوانند در آدرس یکدیگر بنویسند. این

ویژگی در سیستم‌عامل‌های چندپردازشی مورد نیاز است.

۳- پیاده‌سازی حافظه‌ی مجازی: ابتدا به دنبال محتوای مطلوب در حافظه می‌گردد و در

صورتی که نتواند آن را بیابد، خطای صفحه را ایجاد می‌کند.

---

<sup>1</sup> Context

### ۳-۱-۶ ریزپردازنده‌ی LEON 2 و نسخه تحمل‌پذیر اشکال آن

LEON، مدل VHDL ریزپردازنده‌ای مبتنی بر SPARC نسخه ۸ است. این ریزپردازنده برای

کاربردهای نهفته طراحی شده است و ویژگی‌های زیر را دارد [Gaisler\_'05]:

- حافظه نهان جداگانه برای دستورات و داده‌ها
- ضرب‌کننده و تقسیم‌کننده سخت‌افزاری
- کنترل‌کننده وقفه
- واحد پشتیبان اشکال‌زدایی همراه با بافر ردیابی<sup>۱</sup>
- دو زمان‌سنج ۲۴-بیتی
- دو UART
- ویژگی کاهش توان مصرفی
- ریزپردازنده مراقب
- درگاه ورودی/خروجی ۱۶-بیتی
- کنترل‌گر منعطف حافظه
- واسط PCI و اترنت

ماژول‌های جدید، به سادگی به کمک گذرگاه‌های روی تراشه AMBA AHB/APB می‌توانند به

سیستم اضافه شوند. کد VHDL این ریزپردازنده بر مبنای استاندارد VHDL-87 نوشته شده و روی

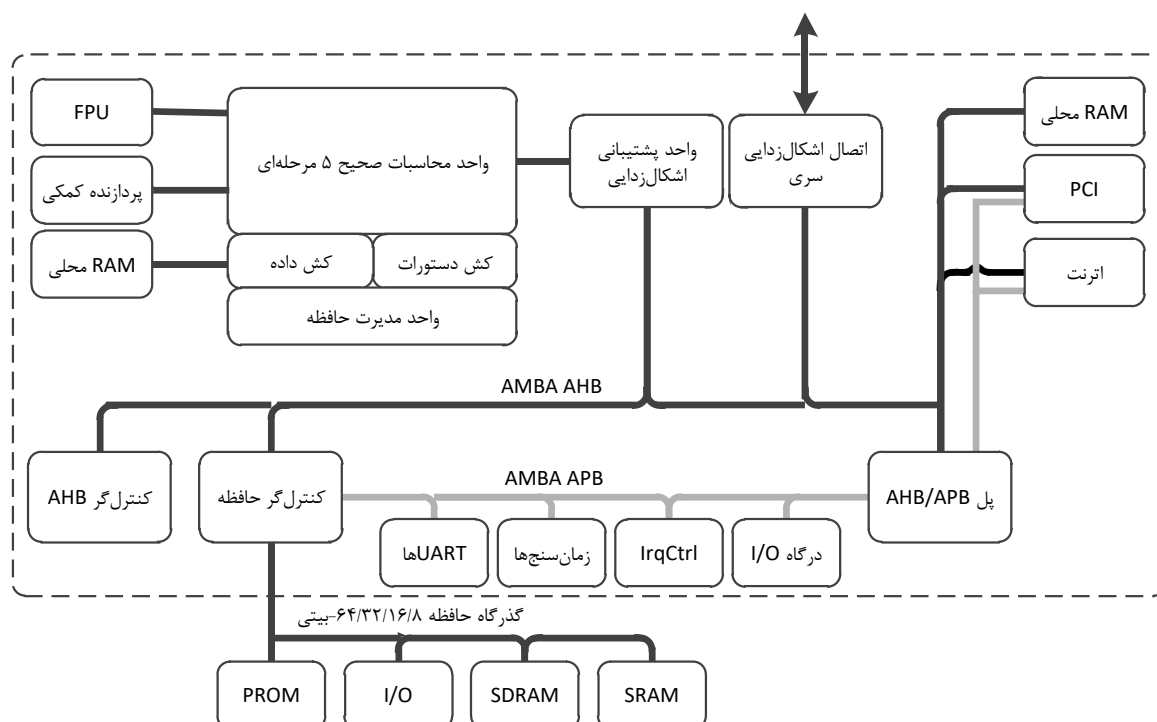
FPGA و ASIC، قابل سنتز می‌باشد. این کد بر اساس مجوز LGPL (برای خود مدل LEON) و GPL

(برای فایل‌های دیگر و testbench) منتشر شده است [Gaisler\_'05].

در شکل ۳-۶، اجزاء ریزپردازنده LEON و نحوه ارتباط آن‌ها را با هم، آمده است.

---

<sup>۱</sup> Trace Buffer



شکل ۳-۶ نمودار بلوکی اجزاء ریزپردازنده LEON [Gaisler\_'05]

ریزپردازنده LEON یک نسخه تحمل‌پذیر اشکال به نام LEON-FT دارد. این ریزپردازنده به گونه‌ای طراحی شده که در مقابل خطاهای SEU مقاوم باشد. برای نیل به این مقصود روش‌هایی نظیر زوجیت<sup>۱</sup>، کدهای BCH و افزونگی سه‌پیمانه‌ای<sup>۲</sup> استفاده شده است. این تکنیک‌ها در سطح کد VHDL پیاده‌سازی شده‌اند و در پروسه ساخت نیمه‌رسانای ریزپردازنده تغییری داده نشده است. کد این ریزپردازنده برخلاف LEON، باز نبوده و در دسترس نمی‌باشد [Gaisler\_'02].

## ۳-۲ واحدهای تحمل‌پذیر اشکال

در این بخش، مروری بر روش‌های تحمل‌پذیری اشکال در واحدهای از قبل طراحی شده که در ریزپردازنده‌ی هدف استفاده شده‌اند، انجام شده است. علاوه بر واحدهای موجود از قبل که در این فصل

<sup>1</sup> Parity

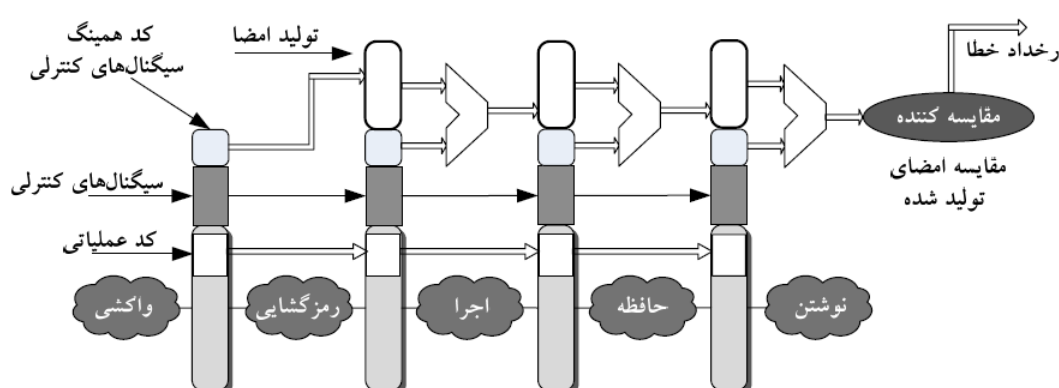
<sup>2</sup> Triple-Modular Redundancy (TMR)

بررسی می‌شوند، یک واحد ممیز شناور نیز طراحی و پیاده‌سازی شده و روش‌هایی برای تحمل‌پذیری اشکال آن ارائه شده است که در فصل بعد به تفصیل به آن پرداخته می‌شود.

### ۳-۲-۱ واحد کنترل

برای افزودن تحمل‌پذیری اشکال به واحد کنترل، از روش ارائه شده در [قاسم‌زاده محمدی\_۸۷] استفاده شده است. این روش با استفاده از افزونگی رفتاری و تولید امضا برای سیگنال‌های ورودی خطلوله، خطاهایی را که در این واحد رخ می‌دهند، کشف می‌کند.

روش کار به این صورت است که مطابق شکل ۳-۷ در هر طبقه از خطلوله کد همینگ مربوط به سیگنال‌های آن طبقه محاسبه می‌شود و در انتشار دستور از یک طبقه از خطلوله به طبقه بعد کد همینگ محاسبه شده برای طبقه قبل واریسی شده و نتیجه با کد تولید شده در طبقه قبلی جمع می‌گردد. به این ترتیب پس از رسیدن امضا به آخرین طبقه خطلوله مقدار این امضا باید صفر باشد، در غیر این صورت، اشکالی در سیستم رخ داده است؛ چراکه در صورت رخ ندادن اشکال حاصل واریسی کد همینگ در هر طبقه کد صفر خواهد بود. در صورت صفر نبودن این مقدار، حتماً خطایی در ریزپردازنده رخ داده است که به این ترتیب کشف و اعلام خواهد شد.



شکل ۳-۷ شمایی از روش به کار رفته در واحد کنترل [قاسم‌زاده محمدی\_۸۷]

### ۳-۲-۲ واحد محاسبه و منطق

روش‌های تحمل‌پذیری اشکال استفاده شده برای بخش‌های مختلف واحد محاسبه و منطق، به شرح زیر می‌باشند [نمازی\_۸۷]:

۱- **واحد منطقی و شیف‌ت:** این دو واحد بخش اندکی از توان مصرفی و مساحت ریزپردازنده را

به خود اختصاص داده‌اند، لذا می‌توان از روش سه‌پیمانه‌ای که قدرت تشخیص و تصحیح خوبی دارد - علی‌رغم سربار زیاد آن - استفاده کرد.

۲- **واحد جمع‌کننده:** در صورتی که برای انجام عمل جمع از جمع‌کننده‌ی پیشگوی رقم نقلی

(CLA<sup>۱</sup>) استفاده شود، نوعی از کدگذاری کم‌هزینه برای تشخیص خطا (LCED<sup>۲</sup>) پیشنهاد

شده است. اگر مدار جمع‌کننده از روش جمع‌کننده با انتخاب رقم نقلی (CSA<sup>۳</sup>) استفاده

کند، از یک روش رایج برای تشخیص خطا و سپس از روش AFT-CSA<sup>۴</sup> [Namazi\_'09]

برای تصحیح خطا می‌توان استفاده کرد. برای این پروژه از روش اول استفاده شده است.

۳- **واحد ضرب‌کننده:** با توجه به این مشاهده که تا اندازه ۶۴-بیت روش دوتائی بهتر از روش

پیشگویی توازنی عمل می‌کند، برای ایجاد تحمل‌پذیری اشکال، از روش دوتائی استفاده شده

است.

۴- **واحد تقسیم‌کننده:** برای این واحد نیز مانند واحد ضرب‌کننده استفاده از روش دوتائی

توصیه شده است.

### ۳-۲-۳ بانک ثبات

در [نمازی\_۸۷] دو روش ممکن برای ایجاد تحمل‌پذیری اشکال در بانک ثبات ارائه شده است.

روش اول استفاده از کد زوجیت است که کارایی را اندکی کاهش می‌دهد اما در عوض از لحاظ توان

<sup>۱</sup> Carry Look-ahead Adder

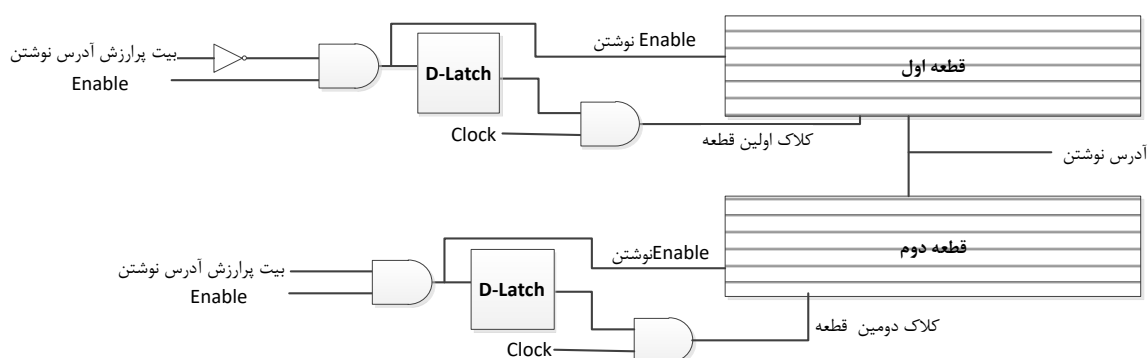
<sup>۲</sup> Low-cost Error Detection

<sup>۳</sup> Carry Select Adder

<sup>۴</sup> Adaptive Fault-Tolerant Carry Select Adder

مصرفی سربار کمی دارد. روش دوم استفاده از روش دوتایی<sup>۱</sup> است که سربار توان مصرفی زیادی دارد ولی موجب افت کارایی نخواهد شد. همچنین روش Partial Clock Gating برای کاهش توان مصرفی پیشنهاد شده است. باید دقت شود که این روش، فقط در کاهش توان مصرفی مدار مؤثر است و اثری روی میزان تحمل پذیری اشکال ندارد. (البته خود مدار مربوط به آن ممکن است میزان تحمل پذیری اشکال را کاهش دهد)

مدار استفاده شده برای روش Partial Clock Gating در شکل ۳-۸ آمده است.



شکل ۳-۸ روش Partial Clock Gating

از میان روش های پیشنهادی، در این پروژه، روش زوجیت همراه با Partial Clock Gating به کار رفته است.

#### ۳-۲-۴ وسائل جانبی روی تراشه

در [۸۷] روش هایی برای ایجاد تحمل پذیری اشکال برای وسائل جانبی ارائه شده است که در این پروژه از آن استفاده شده است. این روش ها را می توان بر حسب وسیله ی جانبی مورد بررسی به شکل زیر دسته بندی کرد:

۱- واحد فرستنده/گیرنده آسنکرون (UART): برای این واحد از روش های مختلف تشخیص

و تصحیح خطا استفاده شده است. برای ماشین حالت از روش تصحیح خطای کد همینگ

<sup>۱</sup> Duplication

استفاده شده است. دو شمارنده‌ی این واحد نیز با کد همینگ محافظت شده‌اند. برای واحد گیرنده روش نوینی به نام تصحیح توسط ذخیره بیت زوجیت [Razmkhah\_'09] ارائه شده است که خطاهایی را که در شیفت‌دهنده این واحد رخ می‌دهند، تشخیص می‌دهد و تصحیح می‌کند. برای فرستنده نیز از کد همینگ استفاده می‌شود.

۲- **شمارنده/زمان‌سنج:** برای این واحد از روش‌های معمول تشخیص خطا استفاده شده است که در میان آن‌ها، افزونگی سه‌تایی کارایی بهتری دارد.

۳- **کنترل‌کننده‌ی درگاه:** برای این واحد، کد همینگ روش مناسبی است که مصالحه‌ی مناسبی بین سربارهای توان مصرفی و مساحت و قابلیت کشف خطا است. با این حال برای این واحد از روش جدیدی استفاده شده است که بر اساس کشف خطا با استفاده از فلیپ‌فلاپ‌های دولبه کار می‌کند.

## فصل ۴

### ۴ طراحی و پیاده‌سازی واحد ممیز شناور ریزپردازنده‌ی هدف

با توجه به نبود یک واحد ممیز شناور کامل به همراه کد ریزپردازنده LEON2، یک واحد ممیز شناور تحمل‌پذیر اشکال به عنوان بخشی از این پروژه طراحی شده است. این فصل به توضیح ملزومات، طراحی، پیاده‌سازی و روش تحمل‌پذیری اشکال این واحد ممیز شناور می‌پردازد.

#### ۴-۱ نیازمندی‌های یک واحد ممیز شناور

معماری SPARC نسخه ۸ دستورات لازم برای یک واحد ممیز شناور را مشخص کرده است. مهم‌ترین این دستورات برای دقت‌های ساده و مضاعف، در جدول ۴-۱ مشاهده می‌شوند [ SPARC

92\_'[International Inc.



جدول ۴-۱ دستورات مهم در واحد ممیز شناور معماری SPARC

دستور	توضیح
عملیات حسابی	
FADDs / FADDd	جمع دو عدد دقت ساده / مضاعف
FSUBd / FSUBs	تفریق دو عدد دقت ساده / مضاعف
FMULs / FMULd	ضرب دو عدد دقت ساده / مضاعف
FDIVs / FDIVd	تقسیم دو عدد دقت ساده / مضاعف
FSQRTs / FSQRTd	یافتن جذر دو عدد دقت ساده / مضاعف
تبدیلات	
FiTOs / FiTOd	تبدیل عدد صحیح به دقت ساده / مضاعف
FsTOi / FdTOi	تبدیل دقت ساده / مضاعف به عدد صحیح
FsTOd / FdTOs	تبدیل دقت‌های ساده و مضاعف به یکدیگر
مقایسه	
FCMPs / FCMPd	مقایسه دو عدد دقت ساده / مضاعف

چنانکه در این جدول مشاهده می شود، واحد ممیز شناور در معماری SPARC نسخه ۸ دارای پنج عمل اصلی جمع، تفریق، ضرب، تقسیم و جذر می باشد که هر یک می توانند در دقت‌های ساده یا مضاعف استفاده شوند. حرف آخر نام دستور در تمامی دستورات ممیز شناور تعیین کننده دقت عملیات است که می تواند ساده (s) و یا مضاعف (d) باشد.

علاوه بر عملیات حسابی، این معماری شامل دستوراتی برای انجام تبدیلات بین اعداد و نیز مقایسه اعداد می باشد. تبدیلات می توانند بین دقت‌های مختلف اعداد اعشاری (از ساده به مضاعف و بالعکس) و یا بین اعداد صحیح و اعشاری (هر یک از دو دقت ساده یا مضاعف) رخ دهد. برای مقایسه دو عدد ممیز شناور، لازم است دقت نمایش این دو عدد یکسان باشد.

نمایش اعداد و نیز شیوه انجام تمام عملیات ممیز شناور در معماری SPARC نسخه ۸، مطابق با استاندارد IEEE-754 برای عملیات ممیز شناور صورت می گیرد [SPARC International Inc., '92]. در

این معماری اعداد ممیز شناور به صورت یک علامت<sup>۱</sup>، یک جزء اعشاری<sup>۲</sup> و یک نما<sup>۳</sup> ذخیره می‌شوند. به علاوه حالت‌های مختلفی برای انجام عمل گرد کردن و محاسبه نتایج عملیات وجود دارد که باید توسط واحدهای ممیز شناوری که این توصیف را پیاده‌سازی می‌کنند فراهم شود.

یک واحد ممیز شناور برای ریزپردازنده LEON2، توسط تولید کننده آن ارائه شده است که تحت لیسانسی غیر از لیسانس بقیه ریزپردازنده ارائه می‌شود و بنابراین در دسترس و رایگان نیست. با این حال در راهنمای کتابخانه IP CORE های Gaisler که این واحد نیز جزئی از آن است، شیوه برقراری ارتباط ریزپردازنده با واحد ممیز شناور و واسطی که باید توسط واحدهای ممیز شناور برای LEON پیاده‌سازی و رعایت شود به تفصیل توضیح داده شده است [Gaisler\_'07]. در این کار از این مستند به عنوان راهنمای پیاده‌سازی واسط ارتباط واحد ممیز شناور طراحی شده با ریزپردازنده استفاده گردیده است.

## ۴-۲ واحد ممیز شناور موجود و واحد ممیز شناور جدید

به همراه کد آزاد ریزپردازنده LEON2 یک واحد ممیز شناور بسیار بدوی برای پر کردن محل واحد ممیز شناور در مدل ریزپردازنده آورده شده است. این واحد نه تنها تمام دستورات SPARC نسخه ۸ را پیاده‌سازی نمی‌کند، بلکه با استاندارد IEEE-754 برای عملیات ممیز شناور نیز سازگار نیست [Gaisler\_'05].

این واحد از واسط کاملی که برای واحدهای ممیز شناور دیگر فراهم شده است استفاده می‌کند [Gaisler\_'05]. به این ترتیب و با توجه به این که ریزپردازنده و ارتباط آن با این واحد قبلاً به طور کامل پیاده‌سازی شده است، برای پیاده‌سازی واحد ممیز شناور جدید نیز از همین واسط استفاده شده است. با استفاده از جایگزینی کد واحد ممیز شناور قبلی با یک کد مبدل<sup>۴</sup> که واسط ریزپردازنده را به واسط کد

<sup>1</sup> Sign

<sup>2</sup> Fraction / Mantissa

<sup>3</sup> Exponent

<sup>4</sup> Adapter

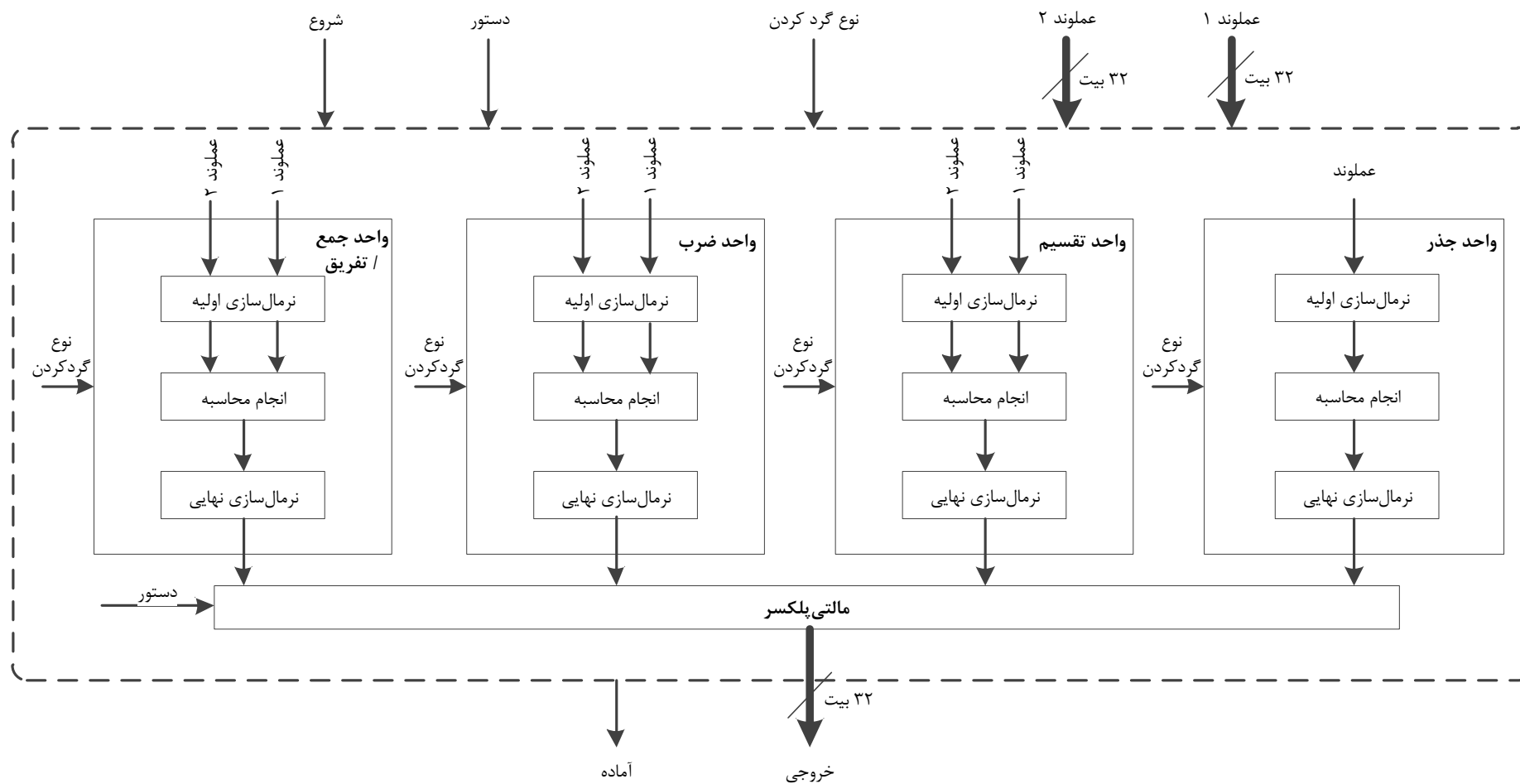
جدید تبدیل می‌کند و نیازمندی‌های زمانی ارتباط را فراهم می‌کند، بدون نیاز به تغییر کد در هسته ریزپردازنده اصلی، واحد ممیز شناور جدید به ریزپردازنده اضافه و جایگزین واحد ممیز شناور اولیه شده است.

واحد ممیز شناور جدید بر مبنای کدی از Jidan Al-Eryani که به صورت آزاد در سایت OpenCores.com در دسترس است (و از این پس آن را «کد مبنا» می‌نامیم) بنا شده است [Al-Eryani\_'06]. این واحد شامل ۴ عمل اصلی (جمع، تفریق، ضرب و تقسیم) و نیز عمل جذر برای دقت ساده است.

شمایی از این واحد ممیز شناور به شکل اولیه آن، در شکل ۴-۱ مشاهده می‌شود. هر یک از چهار بخش نشان داده شده در شکل به طور همزمان ورودی‌ها را دریافت می‌کنند و عملیات را انجام می‌دهند. خروجی با استفاده از یک Multiplexer و با توجه به این که دستور ورودی چه بوده است تعیین می‌شود. عملوندها ۳۲ بیتی هستند و خروجی نیز ۳۲ بیتی است (دقت ساده). هر یک از واحدهای داخلی دارای سه بخش نرمال‌سازی اولیه<sup>۱</sup>، محاسبه و نرمال‌سازی نهایی<sup>۲</sup> هستند که در گام اول داده برای انجام عملیات آماده می‌شود و عملیاتی نظیر تنظیم نما انجام می‌شود و در گام آخر خروجی تنشیم می‌شود و به قالب استاندارد در می‌آید با توجه به نیاز به پشتیبانی عملیات با دقت مضاعف و نیز دستورات دیگر SPARC برای واحد ممیز شناور، تغییراتی در این واحد داده شده است که در بخش بعد بیان می‌شود.

<sup>۱</sup> Pre-normalization

<sup>۲</sup> Post-normalization



شکل ۴-۱ شمایی از واحد ممیز شناور مرجع [Al-Eryani\_'06]

### ۴-۳ تغییرات انجام شده بر روی کد مبنا

تغییراتی که برای تولید واحد ممیز شناور سازگار با SPARC بر روی کد مبنا انجام گرفته است عبارتند از:

۱- افزودن پشتیبانی دقت مضاعف علاوه بر دقت ساده

۲- افزودن واحد مقایسه

۳- افزودن واحد تبدیلات

۴- افزودن سایر دستورات SPARC

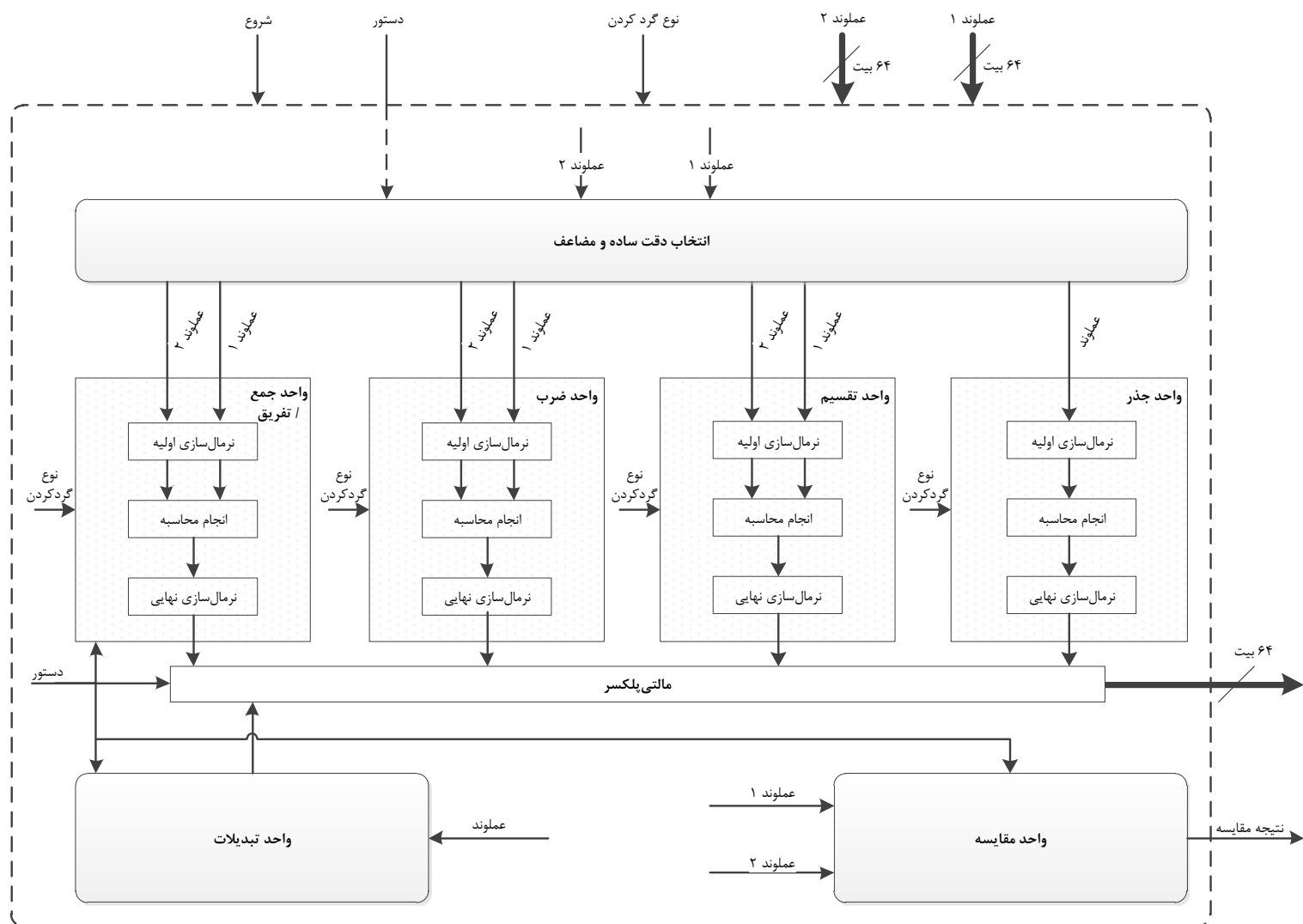
۵- افزودن واسط اتصال واحد ممیز شناور به ریزپردازنده

ابتدا تمام عملیات و داده‌های استفاده شده در کد مبنا به دقت مضاعف تبدیل گردیدند تا پشتیبانی این دقت نیز در سیستم وجود داشته باشد. پس از آن یک واحد مقایسه ساده برای انجام مقایسه بین اعداد دقت مضاعف به سیستم اضافه شد. یک واحد تبدیلات نیز برای انجام تبدیل اعداد صحیح به دقت ساده یا مضاعف و تبدیل بین اعداد با دقت ساده و مضاعف، افزوده شد. سایر دستورات SPARC که برای کار با حافظه‌های ممیز شناور هستند نیز پیاده‌سازی شدند و پس از اتمام این مراحل و پیاده‌سازی کامل دقت ساده، پشتیبانی دستورات دقت ساده نیز به واحد ممیز شناور جدید اضافه شد.

برای برگرداندن قابلیت دقت ساده، از همان سخت‌افزار بخش دقت مضاعف استفاده شده است. این کار که مبنای روش پیشنهادی برای تحمل‌پذیری اشکال در دقت ساده نیز هست، در بخش ۳-۵-۴ بیشتر توضیح داده شده است.

چنان‌که در بخش قبل گفته شد، برای سازگاری واسط ریزپردازنده با واسط واحد ممیز شناور از یک مبدل استفاده شد که به جای واحد ممیز شناور قبلی می‌نشیند و از همان واسط استفاده می‌کند.

شمای واحد ممیز شناور پس از اعمال تغییرات و افروندن واحدهای گفته شده، در شکل ۴-۲ دیده می‌شود. بخش انتخاب دقت، در صورتی که عملیات خواسته شده از نوع دقت ساده باشد، بیت‌های کم ارزش بخش اعشار و بیت‌های پرارزش بخش نما را برابر با صفر قرار می‌دهد تا داده‌های ورودی به معادل دقت مضاعف خود تبدیل شوند و همان عملیات دقت مضاعف بر روی آن‌ها انجام گیرد. تمام اعداد به قالب ۶۴-بیتی در آمده‌اند و سیگنال‌های نتایج مقایسه نیز به خروجی اضافه شده‌اند. مبدل واسط ارتباط با ریزپردازنده در این تصویر نشان داده نشده است.



شکل ۴-۲ شمایی از واحد ممیز شناور نهایی

#### ۴-۴ واحدهای مقایسه و تبدیلات

به کد مبنای واحد ممیزشناور، یک واحد مقایسه سازگار با واسط ریزپردازنده نیز اضافه شده است. این واحد به موازت سایر واحدها عمل می‌کند و در زمان لازم، مقدار سیگنال‌های CC<sup>۱</sup> را که وظیفه مشخص سازی نتیجه مقایسات را دارند، مطابق توصیف ارائه شده برای واسط واحد ممیز شناور، در خروجی تنظیم می‌کند [Gaisler\_'07].

برای پیاده‌سازی کامل دستورات مورد نیاز برای SPARC، به یک واحد تبدیلات بین اعداد نیز نیاز بود [SPARC International Inc.\_'92]. وظیفه این واحد، انجام تبدیلات بین دقت‌های ساده و مضاعف، بین اعداد صحیح و اعداد ممیز شناور دقت ساده، و بین اعداد صحیح و اعداد ممیز شناور دقت مضاعف است. کد این واحد از ابتدا و با توجه به توصیف عملیات مربوطه نوشته شده است و امکان انجام تبدیلات مورد نیاز واسط واحد ممیز شناور را برای هر دو دقت ساده و مضاعف و نیز اعداد صحیح، مطابق جدول ۴-۱ فراهم می‌کند.

#### ۴-۵ تحمل‌پذیری اشکال در واحد ممیز شناور

با توجه به آن که واحد ممیز شناور یک واحد محاسباتی مهم در ریزپردازنده است برای حفظ قابلیت اطمینان ریزپردازنده لازم است روشی نیز برای مقاوم‌سازی این واحد ارائه شود. در ادامه روش‌های موجود برای ایجاد تحمل‌پذیری اشکال در واحد ممیز شناور بررسی و روش‌های پیشنهادی برای عملیات دقت ساده و دقت مضاعف، به تفکیک ارائه شده‌اند.

---

<sup>۱</sup> Condition Codes



## ۴-۵-۱ روش‌های موجود

با توجه به حجم زیاد واحد ممیز شناور و نیز این که این ریزپردازنده برای کاربرد در سیستم‌های نهفته که پایین نگهداشتن مساحت و توان مصرفی در آن حائز اهمیت است طراحی شده است، برای ایجاد تحمل‌پذیری اشکال در واحد ممیز شناور سعی گردیده است که تا جای ممکن از روش‌هایی که سربار سخت‌افزار بسیار کمی دارند استفاده شود.

یک مجموعه روش موجود برای تحمل‌پذیری اشکال واحد ممیز شناور در [شکریان\_۸۷] ارائه شده است. این روش‌ها عمدتاً بر سرباز زمانی به وسیله محاسبه مجدد<sup>۱</sup> استوارند. یکی از روش‌ها نیز با محاسبه‌ای سریع، صحت تقسیم را با استفاده از ضرب مجدد خارج قسمت در مقسوم‌علیه پایش می‌کند. مشکل این روش‌ها این است که اولاً عمدتاً نیاز به افزودنی زمانی قابل توجهی دارند (به جز روش بررسی تقسیم که برای سرباز زمانی کم طراحی شده است). همچنین کد واحد ممیز شناور پیاده‌سازی شده برای این روش‌ها (که در آزمایشگاه موجود بود)، تنها ۴ دستور از دستورهای SPARC را پیاده‌سازی کرده است و با استاندارد IEEE-754 نیز سازگار نیست و در نهایت این که کد این واحد به زبان Verilog است که ادغام و شبیه‌سازی آمیخته<sup>۲</sup> آن با کد VHDL ریزپردازنده مشکلات عملی در پی دارد. به این دلایل از کدهای این مجموعه روش از پیش آماده استفاده نشده است.

یک دسته از روش‌های ایجاد تحمل‌پذیری اشکال، روش‌های مبتنی بر استفاده مجدد از سخت‌افزار<sup>۳</sup> هستند. این روش‌ها مدارهایی را که برای انجام سایر عملیات در ریزپردازنده حاضرند، در مواقعی که بدون استفاده هستند برای افزایش قابلیت اطمینان به کار می‌گیرند و به این ترتیب سربار سخت‌افزاری بسیار کمی (نزدیک به صفر) دارند [Fazeli\_'10]. در ادامه یکی از روش‌های تحمل‌پذیری

<sup>۱</sup> Re-computation

<sup>۲</sup> Mixed Simulation

<sup>۳</sup> Hardware Reuse

اشکال با استفاده مجدد از سخت‌افزار که برای واحد محاسبه و منطق<sup>۱</sup> ارائه شده است برای استفاده در دقت مضاعف در واحد ممیز شناور تطبیق داده یافته و تشریح شده و پس از آن نیز روش جدیدی با استفاده مجدد از سخت‌افزار برای دقت ساده ارائه شده است.

## ۴-۵-۲ روش پیشنهادی برای حالت دقت مضاعف

برای ایجاد تحمل‌پذیری اشکال در واحد دقت مضاعف، می‌توان از روش ارائه شده در گزارش [Fazeli\_'10] استفاده کرد. این روش بر اساس استفاده از داده‌های کم‌عرض<sup>۲</sup> کار می‌کند. به داده‌هایی که در نیمه بالای آن‌ها تمام بیت‌ها صفر یا تمام بیت‌ها یک باشند، داده‌های کم‌عرض گفته می‌شود. «مشاهده شده‌است که بخش زیادی از داده‌هایی که به مدارهای محاسباتی وارد یا از آن خارج می‌شوند، داده‌های کم‌عرض هستند» [Fazeli\_'10]. در هنگام انجام محاسبات با این داده‌ها، نیمی از توان محاسباتی صرف انجام عملیاتی می‌شود که لزومی به انجام آن نیست. به این ترتیب با استفاده مجدد از این بخش از سخت‌افزار در زمانی که عملیات روی داده‌های کم‌عرض صورت می‌گیرد، می‌توان از آن برای ایجاد افزونگی و افزایش قابلیت اطمینان استفاده کرد.

این روش برای مدارهای جمع‌کننده و ضرب‌کننده اعداد صحیح ارائه شده است. مدار جمع‌کننده به دو بخش شکسته می‌شود که یکی بر روی نیمه بالا و دیگری بر روی نیمه پایین داده کار می‌کند. در صورتی که حتی یکی از داده‌های ورودی کم‌عرض نباشند، این مدار مانند یک جمع‌کننده عادی عمل می‌کند، اما هنگامی که هر دو داده ورودی کم‌عرض باشند، هر دو نیمه بالا و پایین جمع‌کننده برای انجام عملیات روی نیمه پایین داده ورودی استفاده می‌شوند تا نتیجه عملیات دو بار محاسبه شده باشد. هم‌چنین با فرض این که همروندی در واحد محاسبه و منطق وجود نداشته باشد، به این معنی که در همین زمان عمل ضربی در این واحد صورت نگیرد، از جمع‌کننده موجود در واحد ضرب برای انجام یک محاسبه دیگر استفاده می‌شود. این سه محاسبه‌ی یکسان، نوعی از افزونگی سه تایی (TMR) را ایجاد

<sup>1</sup> Arithmetic Logic Unit (ALU)

<sup>2</sup> Narrow-width Values

می‌کنند که امکان تشخیص و تصحیح خطا در یکی از سه عملیات را فراهم می‌کند. به این ترتیب خروجی درست عمل جمع با مقایسه این سه نتیجه و گرفتن رای اکثریت به دست می‌آید.

برای استفاده از این روش در انجام عملیات ضرب، باید توجه کرد که اگر عملوندهای اول و دوم را به ترتیب با  $AB$  و  $CD$  نشان دهیم که  $A$  و  $C$  نیمه‌های بالای دو عملوند و  $B$  و  $D$  نیمه‌های پایین هستند، می‌توان عمل ضرب را با جمعی از چهار نتیجه میانی  $A \times C$ ،  $A \times D$ ،  $B \times C$  و  $B \times D$  که هر یک به اندازه لازم شیفت پیدا کرده‌اند انجام داد. اما در حالتی که هر دو عملوند کم‌عرض باشند، به غیر از نتیجه میانی  $B \times D$ ، سه نتیجه دیگر نیازی به انجام کامل عمل ضرب ندارند و مستقیماً قابل تشخیص هستند. به همین دلیل در این حالت می‌توان از هر چهار واحد تولید کننده نتایج میانی برای محاسبه  $B \times D$  استفاده کرد. در حالتی که تنها یکی از دو عملوند کم‌عرض باشد نیز تنها لازم است دو نتیجه از نتایج میانی محاسبه شود و دو واحد محاسبه کننده میانی آزاد خواهند ماند.

در حالتی که هر یک از داده‌های کم‌عرض ورودی به ضرب‌کننده منفی باشند (نیمه بالای آن تماماً یک باشد) برای استفاده از خاصیت بالا، لازم است آن را به داده‌ای تبدیل کنیم که در نیمه بالا تماماً صفر است. این کار با منفی کردن عدد مربوطه انجام می‌شود و علامت نتیجه به شکل مناسب منفی می‌شود یا (در حالتی که هر دو ورودی منفی باشند) بدون تغییر می‌ماند.

اگر تنها یک داده‌ی ورودی ضرب‌کننده کم‌عرض باشند، تنها دو واحد محاسبه میانی آزاد خواهند بود و بنابراین برای هر یک از دو نتیجه میانی، امکان انجام تنها یک عمل افزونه وجود دارد و نمی‌توان از روش TMR استفاده کرد. با این حال می‌توان از دو واحد میانی آزاد شده برای انجام عملیات مضاعف<sup>۱</sup> و تشخیص خطا استفاده کرد. البته در این حالت امکان تصحیح خطا را از دست خواهیم داد ولی با توجه به تعداد زیاد عملوندهایی از این دست، قابلیت تشخیص خطای به دست آمده ارزشمند خواهد بود.

---

<sup>۱</sup> Duplication

چنانکه گفته شده، این روش برای واحد محاسبه و منطق اعداد صحیح ارائه شده است. با این حال برای واحد ممیز شناور نیز می‌توان از این روش استفاده کرد، چرا که در عملیات ممیز شناور نیز اجزای تشکیل دهنده یک عدد ممیز شناور، اعدادی صحیح هستند و عملیات انجام شونده بر روی آن‌ها نیز همان عملیات صحیح است. به این ترتیب کافی است در هر یک از بخش‌های واحد ممیز شناور، واحدهای جمع و ضرب کننده موجود به روش گفته شده تغییر یابند تا اعمال جمع و ضرب ممیز شناور در سطح بالا تحمل‌پذیر اشکال شوند.

برای ایجاد تحمل‌پذیری اشکال در واحد تقسیم‌کننده، می‌توان از روش گفته شده در [شکریان\_۸۷] استفاده کرد. این روش به این ترتیب عمل می‌کند که ابتدا عمل تقسیم را انجام می‌دهد و سپس با انجام عمل ضرب خارج قسمت در مقسوم‌علیه، درستی تقسیم بررسی و اشکال در صورت وجود تشخیص داده می‌شود.

تحمل‌پذیری اشکال برای دقت ساده را می‌توان به شیوه‌ای دیگر با استفاده مجدد از سخت‌افزار دقت مضاعف به دست آورد. این روش در بخش بعد ارائه شده است.

### ۴-۵-۳ روش پیشنهادی برای حالت دقت ساده

روشی که برای ایجاد تحمل‌پذیری اشکال ارائه کرده‌ایم از سخت‌افزار موجود برای دقت مضاعف استفاده می‌کند. در این روش، سخت‌افزار دقت مضاعف به طور همزمان برای انجام محاسبات دقت ساده و نیز برای ایجاد تحمل‌پذیری اشکال در آن به کار گرفته می‌شود.

نکته قابل توجه در عملیات ممیز شناور آن است که تنها تفاروت عمده در دقت‌های ساده و مضاعف، تعداد بیستر بیت‌های هر یک از بخش‌های عدد در دقت مضاعف است. به عبارت دیگر کافی است تعدادی بیت صفر به نمایش دقت ساده‌ی عدد اضافه شود تا همان عدد در نمایش دقت مضاعف به دست آید. به این ترتیب می‌توان از همان سخت‌افزاری که برای محاسبات دقت مضاعف وجود دارد برای انجام محاسبات دقت ساده نیز استفاده کرد، اما در این صورت در زمان انجام عملیات دقت ساده، بخشی

از سخت‌افزار بلااستفاده خواهد ماند. در این روش سعی کرده ایم این بخش از سخت‌افزار دقت مضاعف را که در محاسبات دقت ساده بدون استفاده می‌ماند، برای ایجاد تحمل‌پذیری اشکال به کار گیریم. به این ترتیب هم امکان انجام محاسبات دقت ساده و هم تحمل‌پذیری اشکال آن را می‌توان با سربار اندک سخت‌افزاری به واحد ممیز شناور اضافه نمود.

نکته قابل توجه در عملیات ممیز شناور این است که عملیات ممیز شناور با استفاده از انجام اعمال صحیح بر روی هر یک از اجزای عدد ممیز شناور پیاده‌سازی می‌شود. به این ترتیب هر یک از اجزای تشکیل دهنده یک عدد ممیز شناور خود عددی صحیح است که عملیات اعداد صحیح بر روی آن انجام می‌گیرد. این بدان معناست که برای ایجاد تحمل‌پذیری اشکال در این اعمال، می‌توان از همان روش‌های تحمل‌پذیری اشکال در اعداد صحیح استفاده کرد.

از جمله روش‌های ایجاد تحمل‌پذیری اشکال برای اعمال محاسباتی صحیح، استفاده از کدهای حسابی<sup>۱</sup> است [Johnson\_'88]. این کدها نوع خاصی از کدهای تشخیص خطا هستند که عملوندهای اعمال حسابی مانند جمع و ضرب را با آن‌ها کدگذاری می‌کنند و اعمال حسابی را روی عملوندهای کد شده انجام می‌دهند. طراحی این کدها به صورتی است که نتیجه عملیات حسابی انجام شده روی عملوندهای کدگذاری شده، دارای خاصیت ویژه‌ای خواهد بود که امکان بررسی صحت عملکرد عملیات حسابی را فراهم می‌کند. به عنوان مثال یکی از کدهای حسابی که بسیار مورد قرار می‌گیرد، کد AN نام دارد که در آن عملوندها در عددی مثل A ضرب می‌شوند و پس از انجام عمل جمع، باقی‌مانده نتیجه بر A بررسی می‌شود. در صورتی که نتیجه بر A بخش‌پذیر نباشد، اشکالی در عملیات رخ داده است که به وسیله این کدها تشخیص داده شده است [Johnson\_'88].

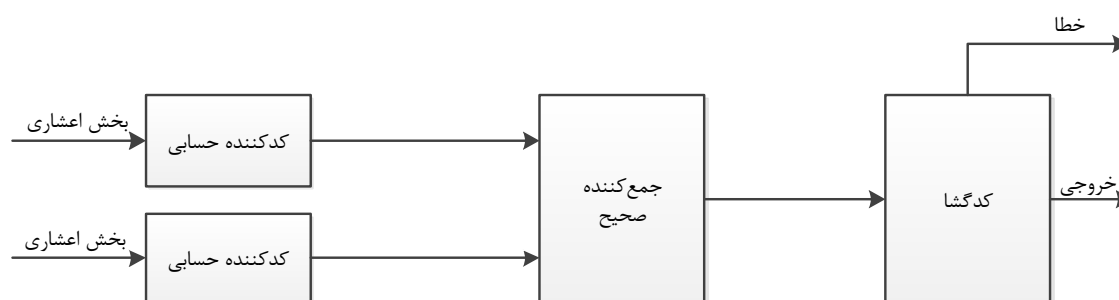
به این ترتیب، در عملیات دقت ساده نیز می‌توان از بیت‌های بدون‌استفاده‌ی واحد دقت مضاعف برای کد کردن عملوندها با کدهای حسابی استفاده کرد. به این ترتیب نیازی به ایجاد هیچ تغییری در

---

<sup>1</sup> Arithmetic Codes

زیرواحدهای عملیات صحیح بخش دقت مضاعف نیست و کافی است یک واحد کدکننده‌ی عملوندها قبل از واحد مربوطه قرار گیرد تا در مواقعی که دستورات دقت ساده باید اجرا شوند عملوندها را کدگذاری کند و یک واحد مقایسه‌کننده نیز بعد از آن قرار داده شود تا پس از پایان عملیات، درستی انجام آن را بررسی کند.

شمای کلی پیاده‌سازی این روش برای جمع‌کننده‌ای در درون یکی از واحدهای محاسباتی که بر روی بخش اعشاری داده‌ها عمل می‌کند، در شکل ۳-۴ مشاهده می‌شود. واحدهای کدکننده و کدگشا تنها در صورتی عمل می‌کنند که دستور مربوطه از نوع دقت ساده باشد و در غیر این صورت تنها داده ورودی را عبور می‌دهند تا عملیات دقت مضاعف به شکل عادی انجام شود.



شکل ۳-۴ شمایی از بخش درونی یک واحد محاسباتی در روش ارائه شده

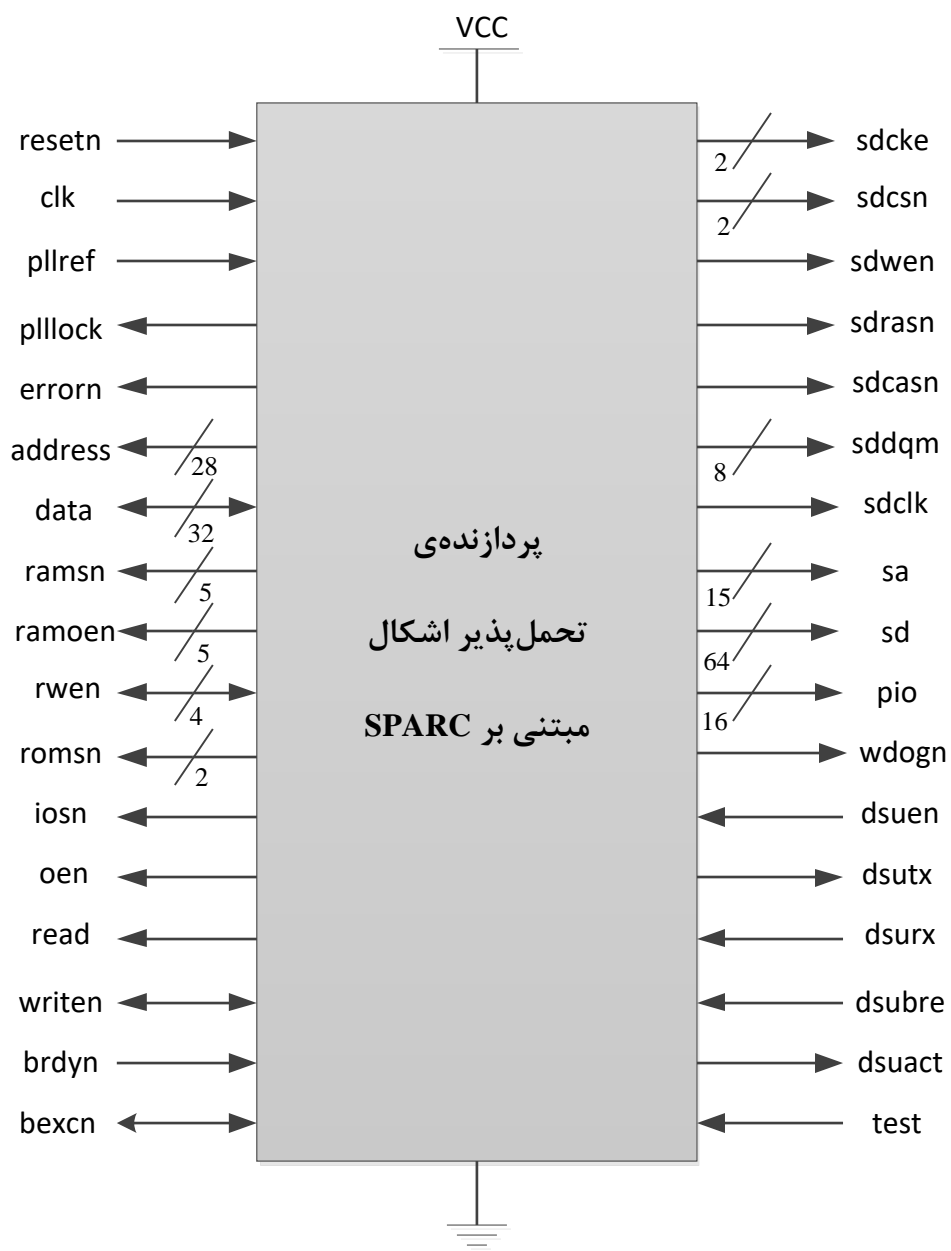
## فصل ۵

### ۵ یکپارچه‌سازی واحدهای اصلی ریزپردازنده هدف

#### ۵-۱ روش انجام آزمون اولیه

در ابتدا درستی هر واحد تحمل‌پذیر اشکال به عنوان جزئی از ریزپردازنده‌ی بدون تحمل‌پذیری اشکال بررسی شد. نحوه‌ی انجام کار به این صورت بوده است که ابتدا ریزپردازنده سنتز شده و سیگنال‌های ورودی و خروجی آن مشخص شدند. این سیگنال‌ها در شکل ۵-۱ آمده است.

سپس LEON به عنوان ریزپردازنده‌ی مرجع، شبیه‌سازی شده و مقادیر این سیگنال‌ها برای آن ضبط گردید. با فرض درستی عملکرد ریزپردازنده مرجع، و با توجه به عدم استفاده از المان تأخیر در کدهای جدید، کافی است نتایج شبیه‌سازی کدهای آن‌ها (مقادیر سیگنال‌های ورودی و خروجی ریزپردازنده) مشابه نتایج شبیه‌سازی ریزپردازنده مرجع باشند. در این حالت مطمئن خواهیم بود که ریزپردازنده‌ی تحمل‌پذیر اشکال درست کار می‌کند.



شکل ۵-۱ پایه‌های ریزپردازنده‌ی تحمل‌پذیر اشکال

## ۵-۲ آزمون‌های مرحله‌ای

قطعات واحد کنترل، واحد محاسبه و منطق، بانک ثبات و وسائل جانبی به وسیله هر یک از هفت

آزمون مورد بررسی درستی قرار گرفتند. در هر مرحله از یکپارچه‌سازی ریزپردازنده نیز، این آزمون‌ها



انجام می‌شوند تا از صحت یکپارچه‌سازی (به طور اجمالی) اطمینان حاصل شود. در نهایت، در فصل ۶، روش‌های کامل‌تری برای صحت‌سنجی ریزپردازنده‌ی یکپارچه معرفی و پیاده‌سازی خواهند شد.

برای بررسی صحت واحدها از هفت آزمون استفاده شد. شش آزمون اول، مربوط به آزمون‌های کوچک‌شده‌ای از برنامه‌های محک MiBench می‌باشند (توضیحات کامل‌تر در بخش ۱-۲-۶ آمده است). آزمون هفتم، آزمون ارائه شده همراه با بسته‌ی leon2-1.0.32-xst است. در ادامه، عملکرد هر کدام از آزمون‌ها توضیح داده می‌شوند [Guthaus\_'01]:

۱- **Basic Math**: این آزمون اعمال مختلف ریاضی نظیر جذر، سینوس، کسینوس، فاکتوریل، قدر مطلق و توان را روی تعدادی داده‌ی از پیش معین، انجام می‌دهد.

۲- **Bit Count**: این آزمون عملکرد ریزپردازنده را از لحاظ تغییر و کار با بیت‌ها می‌سنجد.

۳- **Bubble Sort**: این آزمون مرتب‌سازی حبابی را روی ۱۰ داده از پیش تعیین شده انجام می‌دهد.

۴- **Matrix Multiply**: این آزمون دو ماتریس  $5 \times 5$  را در هم ضرب می‌کند و حاصل را در یک ماتریس  $5 \times 5$  دیگر می‌ریزد.

۵- **Quick Sort**: این آزمون مرتب‌سازی سریع را روی ۱۱ داده انجام می‌دهد.

۶- **Queue**: این آزمون اعمال مختلف مربوط به صف را روی یک سری داده انجام می‌دهد.

۷- **آزمون همراه با LEON**: این آزمون، شامل آزمون‌های مختلفی است که قسمت‌های گوناگون ریزپردازنده را از لحاظ صحت عملکرد بررسی می‌کنند. لیست آزمون‌هایی که انجام می‌شوند از این قرار است:

✓ **MulTest**: آزمون ضرب‌کننده

✓ **DivTest**: آزمون تقسیم‌کننده

✓ **Mem Test و WP Test**: آزمون حافظه

✓ **FPU Test**: آزمون واحد ممیز شناور

✓ **EDAC Test**<sup>1</sup>: آزمون تشخیص و تصحیح خطا

✓ **Cache Test**: آزمون حافظه نهان

✓ **IRQ Test**: آزمون واحد ارسال تقاضا برای وقفه

✓ **UART Test**: با قرار دادن واحد UART در حالت دوری (loop-back) آن را می‌آزماید.

✓ **Timer Test**: آزمون زمان‌سنج

✓ **I/O Port Test**: آزمون درگاه‌های ورودی و خروجی با نوشتن و خواندن از ثبات‌های

مربوط به آن‌ها. به علاوه وقفه‌های درگاه‌ها را نیز می‌آزماید.

### ۵-۳ تهیه نسخه مرجع ریزپردازنده جهت انجام آزمون

برای تهیه نسخه مرجع برای هریک از برنامه‌های آزمون، ابتدا یک پروژه شامل نسخه مرجع LEON تهیه شده است و سپس با استفاده از دو اسکریپت Tcl (پیوست ۳)، شبیه‌سازی برای هریک از آزمون‌ها به عنوان برنامه اجراشونده بر روی ریزپردازنده انجام و سیگنال‌های خروجی ریزپردازنده ضبط و در کنار برنامه آزمون مربوطه، ذخیره می‌شوند.

به این ترتیب برای هر یک از برنامه‌های آزمون، نتایج شبیه‌سازی ریزپردازنده‌ی مرجع، به دست می‌آید. از این نتایج در هنگام بررسی درستی در هریک از مراحل یکپارچه‌سازی استفاده می‌شود؛ جزئیات انجام شبیه‌سازی در پیوست ۲ آمده است. بررسی درستی عملکرد هر واحد به وسیله اسکریپت‌های پیوست ۳ (check.do و testall.do) انجام شده است. این اسکریپت‌ها سیگنال‌های خروجی ریزپردازنده‌ی یکپارچه را با ریزپردازنده مرجع مقایسه می‌کنند.

<sup>1</sup> Error Detection And Correction

#### ۵-۴ یکپارچه‌سازی مرحله به مرحله

پس از بررسی اولیه‌ی درستی هریک از قسمت‌ها، درستی ریزپردازنده‌ی حاصل از کنار هم گذاشتن قطعات، مورد بررسی قرار گرفت. در این مرحله، واحد کنترل، واحد محاسبه و منطق، بانک ثبات، وسائل جانبی و واحد ممیز شناور به ترتیب به ریزپردازنده‌ی غیرتحمیل‌پذیر اشکال اضافه شدند و پس از هر مرحله، تمام برنامه‌های آزمون بر روی ریزپردازنده‌ی حاصل اجرا شده و نتایج بررسی شدند.

#### ۵-۵ جمع‌بندی

واحد کنترل، واحد محاسبه و منطق، بانک ثبات، وسائل جانبی و واحد ممیز شناور کنار هم گذاشته شده و آزمایش شدند. در تمام مراحل، نتایج شبیه‌سازی با نتایج شبیه‌سازی ریزپردازنده مرجع (LEON 2) یکسان بودند. در جدول ۵-۱ فایل‌های تغییر کرده در واحدهای مختلف به طور خلاصه ذکر شده‌اند.

جدول ۵-۱ لیست فایل‌های تغییر یافته در واحدهای مختلف

نام واحد	فایل‌های تغییر یافته
واحد کنترل	iu.vhd
واحد محاسبه و منطق	iu.vhd, multlib.vhd
بانک ثبات	leon.vhd, leon_eth.vhd, leon_eth_pci.vhd, leon_pci.vhd, mcore.vhd, proc.vhd, tech_map.vhd
UART	timers.vhd, uart.vhd, irqctrl.vhd, ioport.vhd
FPU	fpu_lth.vhd
	<p>فایل‌های اضافه شده:</p> <p>addsub_28.vhd convert.vhd mul_24.vhd post_norm_mul.vhd pre_norm_div.vhd serial_div.vhd single_pre.vhd cmp.vhd fpu.vhd post_norm_addsub.vhd post_norm_sqrt.vhd pre_norm_mul.vhd serial_mul.vhd sqrt.vhd comppack.vhd fpupack.vhd post_norm_div.vhd pre_norm_addsub.vhd pre_norm_sqrt.vhd single_post.vhd</p>

## فصل ۶

### ۶ صحت‌سنجی ریزپردازنده‌ی هدف

پس از یکپارچه‌سازی ریزپردازنده، بایستی از صحت عملکرد آن مطمئن شد. برای این کار، می‌توان از دو روش مختلف استفاده کرد [Gajski\_'09]:

- **روش نظری:** در این روش باید مدلی نظری از ریزپردازنده ارائه شود. این مدل — برای آنکه کارآمد باشد — بایستی تمام جزئیات ریزپردازنده در آن آمده باشد. در نهایت باید مدلی هم برای آزمون واقعی ریزپردازنده ارائه داد و به کمک ابزارهای ریاضیاتی ثابت کرد که ریزپردازنده، درست کار می‌کند. خوبی این روش، اطمینان کامل از صحت عملکرد ریزپردازنده است. با این حال، دو ایراد اساسی به این روش وارد است. اول آنکه ارائه‌ی یک مدل جامع و صحیح از ریزپردازنده کاری بسیار دشوار است. این مدل غالباً برای اجزاء ریزپردازنده ارائه می‌شود. به علاوه، این مدل باید تطابق کاملی با کد HDL ریزپردازنده داشته باشد و ایرادات

آن را -در صورت وجود- نیز مدل کند. در مرحله بعد، باید روشی کاملاً جامع از آزمون این مدل ارائه شود تا به کمک آن بتوان این ریزپردازنده را به کمک آن بررسی کرد.

- **روش عملی:** در این روش، کد HDL ریزپردازنده به کمک برنامه‌های محک بررسی می‌شود. خروجی کد HDL، باید دقیقاً با خروجی برنامه‌ی محک در یک سیستم مطمئن و یا خروجی استاندارد ارائه شده همراه با آن همخوانی داشته باشد. این روش، آزمون جامعی نمی‌باشد ولی در عمل، قابل پیاده‌سازی بوده و در اغلب مواقع، خطاهای طراحی را آشکار می‌کند. البته مواردی نظیر ایراد در واحد ممیز شناور ریزپردازنده Pentium نیز بوده‌اند که علی‌رغم یک تریلیون بردار آزمون، خطایی در آن باقی مانده بود [Chen\_'96].

اگرچه به کمک روش نظری، می‌توان از صحت پیاده‌سازی اطمینان پیدا کرد ولی با توجه به پیچیدگی‌های آن، امکان پیاده‌سازی آن برای یک ریزپردازنده‌ی بزرگ -نظیر SPARC- غیرممکن بود. به همین دلیل روش دوم -علی‌رغم ضعف آن در پیدا کردن ایرادات طراحی، انتخاب شد. به همین منظور میزان پوشش کد آزمون‌ها محاسبه شد و طوری آزمون‌ها انتخاب شدند که پوشش قابل قبولی داشته باشند.

## ۶-۱ آزمون اولیه ریزپردازنده با برنامه‌های محک کوچک

پس از یکپارچه‌سازی، آزمون اولیه‌ای که به ازای افزودن هر قطعه روی ریزپردازنده انجام شده بود، مجدداً روی ریزپردازنده‌ی یکپارچه تکرار گردید. خروجی مقایسه‌ای این آزمون‌ها روی ریزپردازنده‌ی نهایی، همانطور که انتظار می‌رفت، صحت یکپارچه‌سازی را تأیید نمود.

## ۶-۲ آزمون‌های کامل ریزپردازنده با بسته‌ی محک MiBench

### ۶-۲-۱ بسته‌ی محک MiBench

MiBench [Guthaus\_'01]، بسته‌ی محک رایگانی است که توسط دانشگاه میشیگان عرضه شده است.<sup>۱</sup> این بسته به صورت کد باز بوده و می‌توان کد آن را برای هرپردازنده‌ی مقصدی — با حداقل تغییر — کامپایل و اجرا نمود. آزمون‌های MiBench در ۶ گروه اصلی قرار دارند که در جدول ۶-۱ آمده‌اند.

جدول ۶-۱ لیست برنامه‌های محک MiBench [Guthaus\_'01]

ارتباطات	امنیتی	شبکه	اداری	خانگی	خودروسازی و صنعت
CRC32	blowfish enc.	dijkstra	ghostscript	jpeg	basicmath
FFT	blowfish dec.	patricia	ispell	lame	bitcount
IFFT	pgp sign	(CRC32)	rsynth	mad	qsort
ADPCM enc.	pgp verify	(sha)	sphinx	tiff2bw	susan (edges)
ADPCM dec.	rijndael enc.	(blowfish)	stringsearch	tiff2rgba	susan (corners)
GSM enc.	rijndael dec.			tiffdither	susan (smoothing)
GSM dec.	Sha			tiffmedian	
				typeset	

### ۶-۲-۲ اجرای برنامه‌های منتخب MiBench روی ریزپردازنده

با توجه به کاربرد در نظر گرفته شده برای این ریزپردازنده، از بین این گروه‌ها، برنامه‌های محک خودروسازی و صنعتی (به طور کامل)، آزمون stringsearch از گروه اداری و آزمون FFT از گروه ارتباطات برای آزمون ریزپردازنده انتخاب شدند.

این برنامه‌ها اغلب به دو صورت بزرگ و کوچک ارائه شده‌اند که آزمون‌های بزرگ برنامه‌ی محک را روی نمونه‌های بیشتری اجرا می‌کنند و زمان‌بر هستند. با این حال چون اجرای این آزمون‌ها یکبار

<sup>1</sup> <http://www.eecs.umich.edu/mibench>

زمان می‌برد و اطمینان از صحت عملکرد ریزپردازنده، بسیار مهم بودند، آزمون‌های بزرگ داوطلب مناسبی برای اجرا روی ریزپردازنده بودند.

برنامه‌های بزرگ، نمونه‌های بزرگتری دارند، که برای اجرا، به RAM بیشتری نیازمندند. به همین دلیل SRAM ریزپردازنده (۲x۶۴ KB)، برای اجرای آن‌ها کافی نیست و بایستی از SDRAM ریزپردازنده

(۲x۱۲۸ KB) استفاده نمود. استفاده از SDRAM باعث می‌شود SRAM دیگر کاربردی نداشته باشد و کنترل‌گر SDRAM به مدار اضافه شود. مشکلی که در اینجا پیش می‌آید آن است که با اجرای تنه‌های برنامه‌های بزرگ، SRAM و مدارهای مربوط به آن آزمون نمی‌شوند. به همین دلیل، برای آنکه آزمون‌های انجام شده جامع باشند، از هر دو نوع آزمون‌ها (کوچک و بزرگ) استفاده شد.

اغلب برنامه‌های محک، اطلاعاتی را به عنوان ورودی از یک فایل می‌خوانند و بعضی نیز خروجی را در فایل می‌نویسند. به عنوان مثال برنامه‌ی محک quicksort، داده‌هایی را که باید مرتب کند از فایل می‌خواند. چون سیستم شبیه‌ساز و آزمایش بسیار ابتدایی است، سیستم فایل<sup>۱</sup> وجود ندارد که بتواند فایل را از روی آن خواند. به عبارتی برنامه به طور مستقیم از حافظه‌ی روی ریزپردازنده اجرا می‌شود. برای برطرف کردن این مشکل دو راه حل وجود دارد:

یکی آنکه فایل ورودی را در به صورت داده‌های ایستا به کد برنامه اضافه کرد و خروجی را نیز در stdout به جای فایل نوشت. یکی از کاستی‌های این روش، آن است که برنامه‌ی محک استاندارد تغییر می‌کند که مطلوب ما نیست؛ چون دسترسی به فایل هم قسمتی از برنامه‌ی محک است که حذف می‌شود. به علاوه زمانی که فایل ورودی باینری است، افزودن اطلاعات آن به برنامه به صورت داده‌های ایستا، تقریباً غیرممکن است.

---

<sup>۱</sup> Filesystem



راه حل دیگر استفاده از سیستم عامل نهفته است. این سیستم عامل ها غالباً به صورت سیستم عامل بی درنگ عرضه می شوند. توضیحات در مورد این راه حل در بخش ۶-۲، موجود است.

### ۶-۳ استفاده از واحد ممیز شناور برای تسریع در اجرای آزمون ها

با توجه به معماری SPARC، پیاده سازی دستورات ضرب و تقسیم صحیح و دستورات کار با اعداد ممیز شناور (از قبیل جمع، تفریق، ضرب، تقسیم و جذر ممیز شناور) برای یک ریزپردازنده ی مبتنی بر SPARC اجباری نیستند [SPARC International Inc.\_'92]. به همین دلیل کامپایلرهای موجود، می توانند کد را طوری تولید کنند که از این دستورات استفاده نکند [Gaisler\_'09]؛ اما استفاده از این دستورات باعث می شود تا سرعت اجرای برنامه ها افزایش یابد. مثلاً اگر ریزپردازنده بخواهد جذر عددی را بدون دستور مربوط به آن حساب کند، باید دستورات زیادی را که معادل انجام این دستور است و توسط کامپایلر تولید شده است، اجرا کند. اگر مدار مربوط به این دستورات به ریزپردازنده اضافه شود، ریزپردازنده چون عملیات را به صورت سخت افزاری انجام می دهد، می تواند کار را با سرعت بیشتری انجام دهد.

به همین دلیل، با توجه به وقت گیر بودن آزمون ها و برای آزمون جامع ریزپردازنده، کد تولید شده، حاوی دستورات مربوط به واحد ممیز شناور و ضرب و تقسیم صحیح نیز بودند. به عنوان مثال، پس از فعال کردن واحد ممیز شناور، و اجرای کد basimath توسط آن، سرعت اجرا، بیش از شش برابر افزایش یافت. از همین جا می توان به این نتیجه رسید که اگر کارایی ریزپردازنده مد نظر باشد و از لحاظ سطح مدار مجتمع و تا حدودی توان مصرفی (واحد ممیز شناور، توان مصرفی ریزپردازنده را افزایش می دهد ولی با توجه به تسریع برنامه ها، باعث می شود در دراز مدت، توان مصرفی کاهش یابد) چندان مهم نباشد، می توان از SPARC با واحدهای مذکور استفاده کرد.

## ۶-۴ استفاده از سیستم عامل

در جدول ۶-۲، لیستی از سیستم عامل های نهفته ای که از ریزپردازنده SPARC نسخه ۸ پشتیبانی می کنند، آمده است.

جدول ۶-۲ مقایسه سیستم عامل های نهفته [Gaisler\_'10]

نام	مجوز استفاده	ویژگی بی درنگ	سازگاری ابزار
<b>RTEMS</b>	آزاد	دارد	دارد
<b>ECos</b>	آزاد	دارد	دارد
<b>Snapgear embedded linux</b>	آزاد	ندارد	دارد
<b>VxWorks</b>	تجاری	دارد	ندارد
<b>LynxOS</b>	تجاری	دارد	ندارد
<b>Nucleus</b>	تجاری	دارد	ندارد

با توجه به مقایسه ی بالا و اینکه RTEMS<sup>۱</sup> حافظه ی کمتری اشغال می کند، برای اجرا در نظر گرفته شد. یکی از کاربردهای مهمی که در بخش ۲-۲-۶ برای سیستم عامل نهفته ذکر شد، پشتیبانی از سیستم فایل است. سیستم عامل RTEMS از سیستم فایل های متعددی پشتیبانی می کند. با این حال، در سیستم شبیه سازی استفاده شده، تنها حافظه جانبی، RAM سیستم است. RTEMS از سیستم فایل درون حافظه<sup>۲</sup> پشتیبانی می کند [OAR Corporation\_'10] در این سیستم عامل، می توان فایل هایی را در قالب tar با برنامه اصلی لینک کرد و در هنگام بارگذاری سیستم عامل، آن ها را untar نمود. پس از آن برنامه می تواند به طور عادی، با فایل های باز شده در حافظه کار کند. به کمک این روش، می توان برنامه های محک MiBench را بدون تغییر روی ریزپردازنده اجرا کرد. حسن دیگری که استفاده از سیستم عامل نهفته دارد، آزمون بخش هایی از ریزپردازنده است که ممکن است برنامه های محک آن را نیازماید. به عنوان مثال، زمان سنج سیستم در برنامه های محک آزموده نمی شود ولی سیستم عامل برای

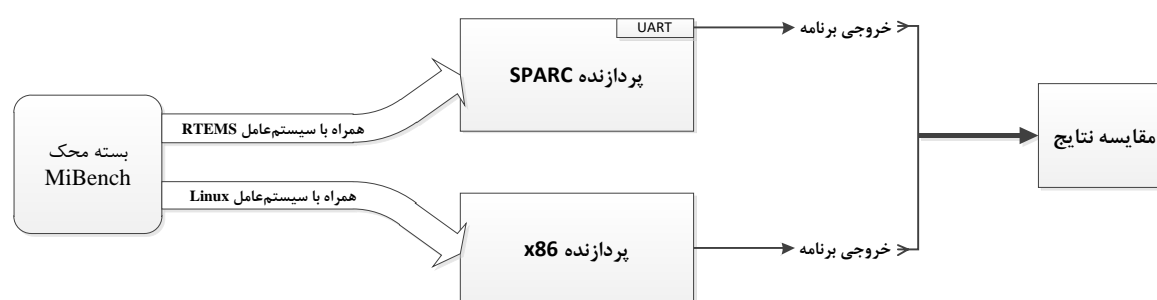
<sup>۱</sup> Real-Time Executive for Multiprocessor Systems

<sup>۲</sup> In-Memory File System (IMFS)

زمان‌بندی وظایف، از آن استفاده می‌کند. تنها ضعف استفاده از سیستم‌عامل، کند شدن شبیه‌سازی است، که چون آزمون‌ها یک بار باید انجام شوند، این ضعف قابل چشم‌پوشی است.

## ۶-۵ نحوه مقایسه نتایج

برنامه‌های محک هنگامی که روی یک کامپیوتر اجرا می‌شوند، خروجی خود را روی صفحه‌ی نمایش چاپ می‌کنند. کامپایلرهای استفاده شده برای SPARC، خروجی دستوراتی نظیر printf را به UART ارسال می‌کنند و برنامه‌ی شبیه‌ساز (ModelSim) نیز این خروجی‌ها را روی پنجره شبیه‌ساز نمایش می‌دهد. این خروجی‌ها را می‌توان با خروجی اصلی بسته‌ی محک MiBench که روی یک ریزپردازنده اینتل x86 اجرا شده است، مقایسه نمود. روند کلی کار در شکل ۶-۱ آمده است.



شکل ۶-۱ روند کلی مقایسه نتایج شبیه‌سازی

## ۶-۶ شبیه‌ساز TSIM

TSIM، ابزاری برای شبیه‌سازی برنامه‌های LEON 3 در سطح دستورالعمل می‌باشد [Gaisler\_'09] در مواردی برای بررسی عملکرد دقیق برنامه روی ریزپردازنده‌ی تحمل‌پذیر اشکال، می‌توان از این ابزار استفاده کرد. به عنوان مثال، شبیه‌سازی سیستم‌عامل در مواردی، بسیار طولانی خواهد بود و اگر در خروجی خطایی مشاهده شود، نمی‌توان مشخص کرد که این خطا مربوط به مراحل کامپایل و استفاده از سیستم‌عامل بوده یا خود ریزپردازنده ایراد دارد. به همین دلیل، برنامه‌های محک تولید شده برای اجرا روی سیستم شبیه‌ساز، ابتدا روی شبیه‌ساز TSIM اجرا شدند تا از خروجی برنامه اطمینان حاصل شود.

## ۶-۷ محاسبه پوشش کد

پس از انجام آزمون روی کدهای ادغام شده، بایستی از جامع بودن این آزمون‌ها اطمینان حاصل گردد. یکی از امکاناتی که ابزار شبیه‌ساز ModelSim فراهم می‌کند، پوشش کد (Code Coverage) است. پوشش کد معیاری است که مشخص می‌کند آزمون اجرا شده، چه میزان از کد HDL را اجرا کرده و بکار برده است. پوشش کدی که توسط ModelSim فراهم می‌شود، موارد زیر را شامل می‌شود [Mentor Graphics Inc., '10]:

۱- Statement

۲- Branch

۳- Condition

۴- Expression

۵- Toggle

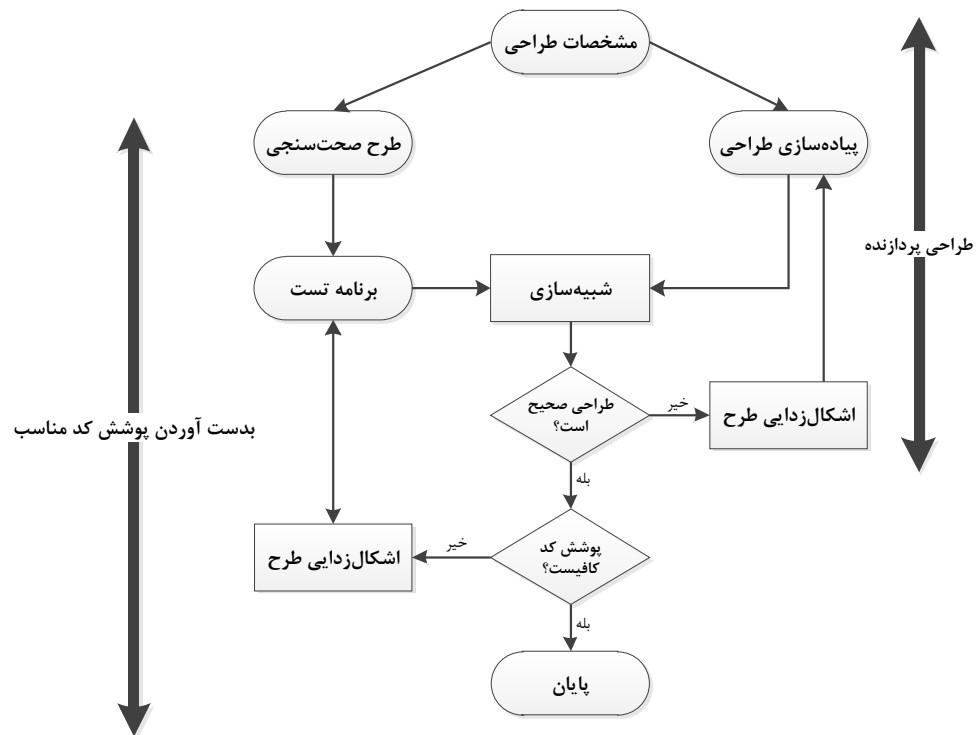
۶- FSM

افزایش میزان پوشش کد، نشان‌گر جامعیت آزمون اجرا شده می‌باشد. با این حال پوشش کد ۱۰۰٪ نمایانگر آزمون کامل کد HDL نیست؛ چرا که ممکن است قسمتی از کد تنها با داده‌ای خاص مشکل خود را بروز دهد. با این حال، پوشش کد معیار مناسبی است که می‌توان به کمک آن، تخمینی از جامعیت آزمون بدست آورد. از مزایای آزمون پوشش کد، محاسبه‌ی آن به صورت اتوماتیک توسط ابزار CAD است [Mentor Graphics Inc., '10].

مراحل طراحی تا اطمینان از جامعیت آزمون‌ها در شکل ۶-۲ رسم شده‌اند.

همانطور که مشاهده می‌شود باید پس از بدست آوردن میزان پوشش کد، سعی شود تا این میزان

بیشتر شده و به ۱۰۰٪ نزدیک شود.



شکل ۶-۲ مراحل طراحی تا اطمینان از جامعیت آزمون‌ها

## فصل ۷

### ۷ ارزیابی قابلیت اطمینان در ریزپردازنده‌ی هدف

در فصل دوم در مورد اهمیت قابلیت اطمینان در سیستم‌های نهفته صحبت شد. برای دست‌یابی به قابلیت اطمینان از روش‌های گوناگونی استفاده می‌شود. در این روش‌ها با ایجاد نوعی از افزونگی مانند افزونگی سخت‌افزاری (مثل استفاده از دو قطعه مشابه و مقایسه خروجی آن‌ها)، افزونگی زمانی (مانند تکرار عمل انجام شده و مقایسه نتایج)، افزونگی اطلاعاتی (مانند استفاده از کدهای تصحیح خطا<sup>۱</sup>) و یا افزونگی نرم‌افزاری (مثل استفاده از برنامه‌سازی N نسخه‌ای<sup>۲</sup>)، عملکرد سیستم بررسی و اشکالات احتمالی کشف و در صورت امکان تصحیح می‌شوند [Johnson\_88].

روش‌های تحمل‌پذیری اشکال که در طراحی یک سیستم به کار گرفته می‌شوند لازم است ارزیابی شوند تا کارایی آن‌ها و قابلیت اطمینان سیستم مشخص شود. روش‌های ارزیابی قابلیت اطمینان کمک

---

<sup>۱</sup> Error Correction Codes (ECC)

<sup>۲</sup> N-Version Programming

می‌کنند طراحان و سازندگان سیستم قابلیت اطمینان کنونی را با میزان مورد نیاز آن مقایسه و تغییرات و راهکارهای لازم را در حین طراحی و ساخت سیستم اعمال کنند. روش‌های ارزیابی قابلیت اطمینان را می‌توان به طور کلی به دو دسته روشهای تحلیلی و روشهای تجربی تفکیک کرد [Iyer\_'96]. در ادامه هر یک از این دو دسته کلی بیشتر بررسی شده‌اند.

## ۷-۱ روش‌های ارزیابی قابلیت اطمینان

### ۷-۱-۱ روش‌های تحلیلی

در روش‌های تحلیلی ارزیابی قابلیت اطمینان، مدلی ریاضی از سامانه تحت مطالعه ساخته و با استفاده از آن، ویژگی‌های مختلف مرتبط با قابلیت اطمینان در سامانه بررسی می‌شود. از جمله این مدل‌ها می‌توان به مدل‌های ترکیبیاتی و مدل‌های احتمالاتی اشاره کرد [Johnson\_'88].

در مدل‌های ترکیبیاتی ساختار سامانه متشکل از اجزای آن مدل‌سازی و قابلیت اطمینان سامانه با استفاده از قابلیت اطمینان هر یک از اجزا و با توجه به نسبت ساختاری اجزا با یکدیگر محاسبه می‌شود. از جمله این مدل‌ها می‌توان به سیستم‌های سری<sup>۱</sup>، سیستم‌های موازی<sup>۲</sup> و سیستم‌های غیر سری-موازی<sup>۳</sup> اشاره کرد [Johnson\_'88] [Koren\_'07]. از جمله اشکالات مدل‌های ترکیبیاتی عدم توانایی آن‌ها در مدل‌سازی سیستم‌های پیچیده به دلیل افزایش تعداد اجزا و پیچیدگی ساختار است [Johnson\_'88].

در روش احتمالاتی، حالت‌های<sup>۴</sup> سیستم (مثلاً ترکیبی از سلامت و یا خرابی هر یک از اجزا [Johnson\_'88]) یک مدل احتمالاتی را تشکیل می‌دهند. احتمال جابه‌جایی بین حالات با توجه به داده‌های قبلی در مورد سیستم و اجزای آن به دست می‌آید و قابلیت اطمینان با استفاده از میزان

<sup>1</sup> Series Systems

<sup>2</sup> Parallel Systems

<sup>3</sup> Non-series-parallel Systems

<sup>4</sup> States

احتمال حضور سیستم در حالات بدون خطا محاسبه می‌شود. از جمله مدل‌های احتمالاتی رایج، می‌توان به «مدل‌های مارکوف»<sup>۱</sup> و «مدل‌های پواسون»<sup>۲</sup> اشاره کرد [Koren\_'07].

استفاده صرف از روش‌های تحلیلی برای ارزیابی قابلیت اطمینان، دارای مشکلاتی است که از آن جمله عبارتند از [زرندی\_'۸۱]:

۱- لزوم انطباق کافی مدل با واقعیت که حصول آن در مواقعی که سیستم بزرگ باشد دشوار و یا غیرعملی است.

۲- محاسبات سنگین در صورت پیچیده بودن سیستم.

۳- نیاز به مشخص بودن پارامترهای سیستم و اجزای آن که ممکن است نیاز به اندازه‌گیری‌های تجربی داشته باشد.

به دلایل بالا، معمولاً از این روش در کنار روش‌های دیگر ارزیابی قابلیت اطمینان استفاده می‌شود [زرندی\_'۸۱].

## ۷-۱-۲ روش‌های تجربی

روش‌های تجربی ارزیابی قابلیت اطمینان را می‌توان برحسب مرحله‌ای از طول عمر سیستم که در آن اعمال می‌شوند طبقه‌بندی کرد. برای هر یک از مراحل «طراحی»<sup>۳</sup>، «نمونه‌سازی»<sup>۴</sup> و «عملیاتی»<sup>۵</sup>، روش‌های مختلف ارزیابی قابلیت اطمینان وجود دارند [Iyer\_'96]. عمده این روش‌ها بر اساس تزریق اشکال به سیستم به طرق مختلف و بررسی رفتار سیستم در اثر اشکال تزریق شده بنا شده‌اند.

<sup>1</sup> Markov Models

<sup>2</sup> Poisson Models

<sup>3</sup> Design Phase

<sup>4</sup> Prototype Phase

<sup>5</sup> Operational Phase



تزریق اشکال را به طور کلی می‌توان به سه دسته تزریق اشکال فیزیکی، تزریق اشکال نرم‌افزاری پیاده‌سازی شده<sup>۱</sup> و تزریق اشکال شبیه‌سازی شده<sup>۲</sup> تقسیم کرد [Zarandi\_'03]. در تزریق اشکال به روش فیزیکی، سیستم به صورت سخت‌افزاری پیاده‌سازی شده و با روش‌های مختلفی از جمله با استفاده از زنجیره پیمایش<sup>۳</sup>، اشکال مورد نظر در محل مورد نظر در آن تزریق می‌شود [Ejlali\_'03] و مورد ارزیابی قرار می‌گیرد. در تزریق اشکال نرم‌افزاری پیاده‌سازی شده، اشکال‌ها با استفاده از نرم‌افزار به یک سیستم فیزیکی تزریق می‌شوند [Ejlali\_'03]. به عنوان مثال بخشی از نرم‌افزار وظیفه تخریب داده مورد استفاده توسط بخش دیگر را بر عهده می‌گیرد. در تزریق اشکال شبیه‌سازی شده، سیستم به وسیله یک زبان مدل‌سازی مانند VHDL شبیه‌سازی می‌شود و اشکال‌ها به این مدل تزریق می‌شوند [Zarandi\_'03].

در فاز طراحی معمولاً از تزریق اشکال شبیه‌سازی شده برای ارزیابی قابلیت اطمینان استفاده می‌شود. برای این منظور، با به کار گیری ابزارهای طراحی به کمک کامپیوتر<sup>۴</sup> و تزریق اشکال به مدل شبیه‌سازی شده قابلیت اطمینان آن مورد آزمون قرار می‌گیرد تا بازخوردی به موقع برای انجام اصلاحات لازم در روش‌های تحمل‌پذیری اشکال استفاده شده به طراحان داده شود [Iyer\_'96]. به شیوه اعمال هر یک از انواع اشکالات در شبیه‌سازی، مدل آن اشکال گفته می‌شود. این مدل اشکال به سطح تجریدی که سیستم در آن شبیه‌سازی می‌شود بستگی زیادی دارد [Chen\_'03].

شبیه‌سازی سیستم می‌تواند در سطوح مختلف تجرید سیستم رخ دهد. در سطوح بالاتر مثل زمانی که مدل رفتاری<sup>۵</sup> سیستم شبیه‌سازی می‌شود، سرعت شبیه‌سازی بالاتر است اما مدل‌های رخداد اشکال باید با دقت بیشتری طراحی شوند تا به واقعیت نزدیک‌تر باشند. در این موارد معمولاً اثر اشکال در سطح انتزاع مربوطه برای مدل‌سازی آن اشکال به کار می‌رود. در سطوح پایین‌تر امکان شبیه‌سازی نزدیک‌تر به واقع مدل‌های اشکال وجود دارد اما جزئیاتی که باید شبیه‌سازی شوند و به همان نسبت

<sup>1</sup> Software Implemented Fault Injection (SWIFI)

<sup>2</sup> Simulated Fault Injection

<sup>3</sup> Scan Chain

<sup>4</sup> Computer-Aided Design (CAD)

<sup>5</sup> Behavioural

کندی شبیه‌سازی بیشتر می‌شود. به عنوان مثال شبیه‌سازی در سطح الکتریکی، امکان بررسی اثر علل فیزیکی منجر به رخداد یک اشکال را فراهم می‌کند [Iyer\_'96].

در فاز نمونه‌سازی، از تزریق اشکال فیزیکی به سیستم استفاده می‌شود [Ejlali\_'03]. در برخی از این روش‌ها از زنجیره پویش برای دستیابی به وضعیت درونی سخت‌افزار و تغییر آن استفاده می‌شود. در تعدادی دیگر از روش‌ها، اشکال از طریق پایه‌های یک مدار مجتمع<sup>۱</sup> به آن تزریق می‌شود. در بعضی از روش‌ها نیز از طریق ایجاد آشفتگی در دنیای خارج<sup>۲</sup>، مثلاً قرار دادن سخت‌افزار تحت تابش پرتوهای پرانرژی، شرایطی که در آن نوع خاصی از اشکال رخ می‌دهد در پیرامون سیستم ایجاد می‌شود [Ejlali\_'03].

در فاز عملیاتی، با جمع‌آوری داده از سیستم در حین کار آن در شرایط واقعی، شکل واقعی رخداد اشکال در عمل مشاهده و با تحلیل اندازه‌گیری‌های انجام شده قابلیت اطمینان سیستم محاسبه می‌شود. هم‌چنین این اندازه‌گیری‌ها مدل‌های واقعی رخداد خطا در عمل را مشخص می‌کنند و می‌توانند برای تعیین پارامترهای ورودی مدلسازی در سایر روش‌ها (مانند روش‌های تحلیلی) به کار [Ejlali\_'03].

تزریق اشکال در هر یک از سطوح تجرید و فازهای طول عمر سیستم مزایا و معایبی دارد که هریک مکمل دیگری است. تزریق اشکال شبیه‌سازی شده امکان آزمون سیستم قبل از ساخت آن را فراهم می‌کند، اما به مدل‌سازی‌های دقیق و انتخاب درست پارامترهای شبیه‌سازی وابسته است [Ejlali\_'03]. از سوی دیگر تزریق اشکال در سطح فیزیکی دقت بیشتری فراهم می‌کند اما هزینه بیشتری دارد و پیاده‌سازی آن دشوارتر است. در کنار دو روش فوق، روش‌های ارزیابی قابلیت اطمینان در

<sup>1</sup> Integrated Circuit (IC)

<sup>2</sup> External Disturbance

زمان عملیاتی بودن سیستم، امکان محاسبه پارامترهایی نظیر «زمان متوسط بین دو خرابی»<sup>۱</sup> را فراهم می‌کند که در روش‌های دیگر قابل محاسبه نیست [Iyer\_'96].

برای تزریق اشکال در این پروژه، از تزریق اشکال شبیه‌سازی شده استفاده شده است. برای این منظور از یک مدل اشکال رایج در تحقیقات اتکاپذیری استفاده شده است. فصل بعد به توصیف این مدل و نحوه شبیه‌سازی آن در سطح کد VHDL می‌پردازد.

## ۷-۲ مدل اشکال SEU

یک دسته از اشکال‌های گذرا که در سیستم‌های سخت‌افزاری رخ می‌دهد، خطاهای دارای اثر تک-رخداد (SEE<sup>۲</sup>) است که بر اثر تابش ذرات پر انرژی امکان اتفاق افتادن دارد. این ذرات می‌توانند موجب شوند که مقدار ولتاژ در یکی از گره‌های مدار و یا مقدار ذخیره شده در یک خانه حافظه تغییر کند. این تغییر در مدارهای ترتیبی بر عناصر حافظه تاثیر می‌گذارد و به شکل SEU<sup>۳</sup> ظاهر می‌شود و در مدارهای ترکیبی بر گره‌های مدار تاثیر می‌کند و به شکل SET<sup>۴</sup> نمود می‌یابد [قاسم‌زاده محمدی\_'۸۷].

مدل اشکال SEU در عمل کاربرد فراوانی دارد و بسیار مورد مطالعه قرار گرفته است. دلیل این امر این است که به علت زیاد بودن ذرات پرنرژی در فضا، در کاربردهای فضایی این نوع اشکال به وفور رخ می‌دهد و برای مقابله با این مدل اشکال نیز روشهای گوناگونی ارائه شده‌است [قاسم‌زاده محمدی\_'۸۷].

برای شبیه‌سازی مدل اشکال SEU در تزریق اشکال شبیه‌سازی شده در سطح کد VHDL، در زمان رخداد اشکال مقدار ذخیره شده در عنصری از حافظه که مقصد رخداد خطاست به مدت یک سیکل کلاک معکوس می‌شود. این زمانی است که برای مشاهده اثر تغییر مقدار آن عنصر حافظه توسط مدارهای متصل به آن لازم است. پس از این مدت مقدار عنصر حافظه به مقدار اصلی آن که توسط روند

<sup>۱</sup> Mean Time Between Failures (MTBF)

<sup>۲</sup> Single Event Effect

<sup>۳</sup> Single Event Upset

<sup>۴</sup> Single Event Transient

عادی شبیه‌سازی تعیین می‌شود بازگردانده می‌شود (زیرا پس از یک سیکل کلاک، مقدار موجود با مقدار ورودی عنصر حافظه جایگزین می‌شود). سپس شبیه‌سازی تا انتها ادامه می‌یابد تا اثر اشکال تزریق شده بر عملکرد سیستم مشاهده شود.

برای شبیه‌سازی این مدل لازم است اشکال به عناصر حافظه تزریق شود. برای این منظور چنانکه در ادامه گفته خواهد شد، سیگنال‌های متناظر با عناصر حافظه در مدل سطح بالا به دست می‌آیند و تزریق اشکال به آن‌ها صورت می‌گیرد.

### ۷-۳ برنامه‌های مقصد تزریق اشکال

با توجه به زمینه کاربرد این ریزپردازنده چنانکه در فصل ۶ اشاره شد، از برنامه‌های محک MiBench به عنوان برنامه‌های اجرا شونده در زمان تزریق اشکال استفاده شده است. برنامه‌های استفاده شده از این بسته محک برای تزریق اشکال عبارتند از:

۱- Basic Math

۲- Bitcount

۳- qsort

بر خلاف مورد سنجش صحت عملکرد که در آن به علت قطعی بودن شبیه‌سازی، یک بار اجرای برنامه کفایت می‌کند، در تزریق اشکال لازم است تا با توجه به مدل احتمالاتی توزیع رخداد اشکالات، تعداد زیادی اشکال در زمان‌های مختلف به برنامه تزریق شوند و برای جلوگیری از تداخل آثار اشکال‌های تزریق شده و نیز وجود امکان تحلیل آن‌ها، لازم است که به ازای هر بار تزریق اشکال، برنامه محک یک بار به طور کامل اجرا شود. بنابراین هر یک از برنامه‌های محک باید بارها اجرا شوند و به همین دلیل امکان استفاده از برنامه‌هایی که شبیه‌سازی آن‌ها مدت زیادی به طول می‌انجامد وجود ندارد.

از سوی دیگر با شبیه‌سازی برنامه‌های محک مورد استفاده مشخص شد که میزان پوشش برای بخش‌های مختلف ریزپردازنده توسط برنامه‌های محک، چنانکه در جداول ۷-۱ و ۷-۲ آمده است، با

کوچک شدن زیاد اندازه ورودی برنامه محک تغییر زیادی نمی‌کند. به این ترتیب می‌توان انتظار داشت که کاهش تعداد ورودی‌های برنامه تاثیر زیادی بر میزان کشف و پوشش خطا نداشته باشد، به خصوص که برخی برنامه‌های محک ماهیتاً برنامه‌های حلقه‌ای<sup>۱</sup> هستند و لذا دنباله دستوراتی که برای هر تعداد ورودی انجام می‌شوند یکسان‌اند و تنها تعداد دفعات تکرار متفاوت است. به همین دلیل، و با توجه به این که از نظر مفهومی نیز مشاهده اثر تزریق اشکال تا تعداد مشخصی پالس کلاک پس از رویداد اشکال معنادار و کافی است، اندازه ورودی برنامه‌های استفاده شده کاهش یافت تا زمان اجرای هر یک از برنامه‌ها، با حفظ پوشش برنامه اصلی، به حدود یک میلیون سیکل کلاک برسد که از نظر میزان اجرا برای مشاهده اثر خطا مناسب و از نظر زمان شبیه‌سازی نیز معقول و قابل دسترس است.

در جداول زیر میزان پوشش دستورات، شرط‌ها و عبارات در برنامه Quicksort برای اندازه‌های مختلف ورودی آمده است. پوشش‌ها در اکثر بخش‌ها کاملاً مشابه و بدون تغییر هستند و در واحدهای اعداد صحیح (IU) و حافظه نهان داده (Data Cache) تغییرات بسیار اندکی وجود دارد که قابل چشم‌پوشی است.

جدول ۷-۱ پوشش کد واحد اعداد صحیح

اندازه‌ی برنامه	دستورات	شرط‌ها	عبارت‌ها
برنامه‌ی اصلی	68.5%	40.2%	52.0%
با ۲۰۰ داده ورودی	68.5%	40.2%	52.4%
با ۱۰۰ داده ورودی	68.5%	40.2%	52.0%
با ۱۰ داده ورودی	68.3%	40.2%	51.7%

جدول ۷-۲ پوشش کد حافظه نهان داده

اندازه‌ی برنامه	دستورات	شرط‌ها	عبارت‌ها
برنامه‌ی اصلی	59.1%	43.7%	51.1%
با ۲۰۰ داده ورودی	59.1%	43.7%	51.1%

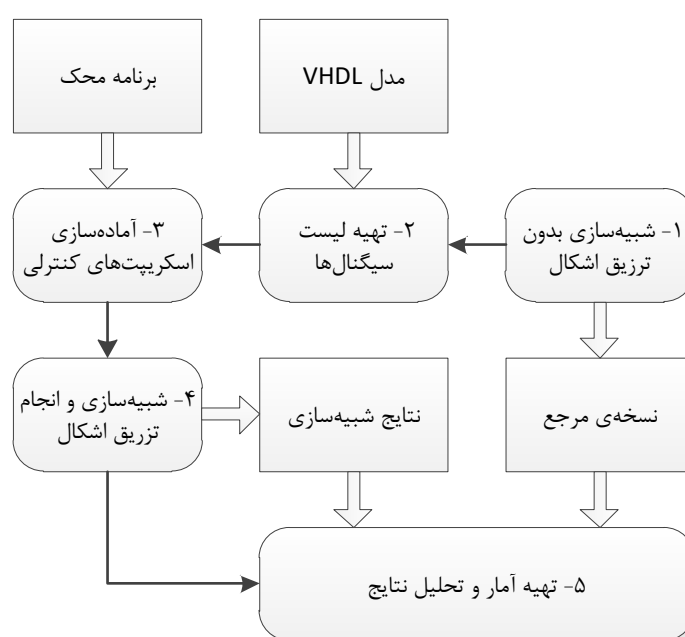
---

<sup>۱</sup> Iterative

با ۱۰۰ داده ورودی	60.1%	46.7%	53.5%
با ۱۰ داده ورودی	59.1%	43.2%	51.1%

#### ۷-۴ فرآیند تزریق اشکال

چنانکه در بخش ۷-۲ گفته شد، اشکال‌های تزریق شده دارای مدل SEU هستند. فرآیند کلی تزریق اشکال در شکل ۷-۱ مشاهده می‌شود. در ادامه هر یک از بخش‌های این فرآیند توضیح داده شده است.



شکل ۷-۱ فرآیند کلی تزریق اشکال

**شبیه‌سازی بدون تزریق اشکال:** برای تحلیل نتایج تزریق اشکال لازم است که خروجی‌های حاصل با یک نسخه مرجع<sup>۱</sup> مقایسه شوند. برای این منظور برنامه محکی که قرار است در شبیه‌سازی استفاده شود باید یک بار بدون تزریق اشکال بر روی ریزپردازنده اجرا شود تا علاوه بر به دست آمدن زمان شبیه‌سازی برنامه - که در تولید سناریوهای تزریق اشکال مورد استفاده قرار می‌گیرد - خروجی‌های اجرای سالم برنامه نیز به دست آید تا به عنوان نسخه مرجع از آن استفاده شود.

<sup>۱</sup> Golden Version

بخش‌هایی از خروجی سیستم که برای بررسی ذخیره می‌شوند عبارتند از:

۱- **خروجی برنامه:** خروجی برنامه که حاصل از دستورات نوشتن در کنسول (printf) می‌باشد

همان‌طور که در بخش ۵-۶ توضیح داده شد از طریق درگاه UART ریزپردازنده خارج و نیز

در کنسول ابزار شبیه‌سازی منعکس می‌شود. این خروجی برای بررسی صحت عملکرد

ریزپردازنده ذخیره می‌گردد.

۲- **محتوای حافظه‌ها:** در انتهای شبیه‌سازی محتوای حافظه‌های SRAM ریزپردازنده ذخیره

می‌گردند.

۳- **حالت ریزپردازنده<sup>۱</sup>:** مقادیر هر یک از رجیسترها و سیگنال‌های ریزپردازنده در انتهای

شبیه‌سازی ذخیره می‌گردد.

ذخیره سازی محتوای حافظه و حالت ریزپردازنده در پایان شبیه‌سازی کمک می‌کند تا اشکال‌های

متأخر (که در ادامه این بخش بررسی می‌شوند) شناسایی شوند. به دلیل تعداد زیاد سیگنال‌های

ریزپردازنده، امکان ثبت<sup>۲</sup> تغییرات تمام سیگنال‌ها وجود ندارد (فایل حاصل برای هر تزریق اشکال حجمی

بیش از چهار گیگابایت پیدا می‌کند). به همین ترتیب ذخیره تمام تغییرات خانه‌های حافظه نیز عملی

نیست. به این دلایل، به تشخیص خرابی سامانه و اشکال‌ها و خطاهای متأخر بسنده شده است و تنها

محتوای حافظه و حالت ریزپردازنده در انتهای شبیه‌سازی مورد مقایسه قرار گرفته‌است.

**تهیه لیست سیگنال‌ها:** ابتدا لازم است که مشخص شود تزریق اشکال به چه سیگنال‌هایی باید

انجام شود. با توجه به بحث مطرح شده در بخش ۲-۱-۷ تزریق اشکال پس از سنتز برای سیستمی با این

میزان پیچیدگی بسیار کند است. به همین دلیل لازم است تا تزریق اشکال به مدل VHDL که دارای

سطح انتزاع بالاتری است انجام شود. اما با توجه به آن که مدل تزریق اشکال استفاده شده SEU می‌باشد،

<sup>۱</sup> Processor State

<sup>۲</sup> Logging

لازم است که تزریق در سطح مدل VHDL به طریقی انجام گیرد که به اندازه کافی به مدل واقعی SEU نزدیک باشد. به عبارت دیگر لازم است مشخص شود چه سیگنال‌هایی از مدل VHDL پس از انجام عمل سنتز به عناصر حافظه تبدیل می‌شوند تا در سطح مدل VHDL نیز تزریق اشکال به همان سیگنال‌ها صورت گیرد.

برای شناسایی این سیگنال‌ها، ریزپردازنده سنتز می‌شود و سیگنال‌های مدل VHDL که متناظر با هر یک از عناصر حافظه در مدل سنتز هستند شناسایی و استخراج می‌شوند. برای انجام این عمل وجود دو ویژگی در ابزار سنتز لازم است. نخست آنکه خروجی ابزار سنتز امکان تشخیص واحدهای حافظه را به راحتی فراهم کند و دوم آن که ابزار از همان نام‌های سیگنال‌های مدل VHDL برای عناصر متناظرشان در مدل سنتز استفاده کند تا تشخیص تناظر بین این دو ممکن گردد. ابزار Design Compiler یکی از ابزارهای سنتز دارای این دو ویژگی است که در این کار استفاده شده است.

**آماده‌سازی اسکریپت‌های کنترلی:** در ابتدای هر رشته عملیات تزریق اشکال برای یک برنامه محک، لازم است سناریوهای تزریق اشکال برای آن برنامه آماده شوند. هر سناریو تزریق اشکال یک اسکریپت است که دستورات کنترل کننده ابزار شبیه‌سازی را برای تزریق یک اشکال در زمانی مشخص بر روی یک سیگنال مشخص به دست آمده از بخش قبل در حین اجرای یک برنامه محک مشخص، در خود دارد.

در بخش‌هایی از ریزپردازنده که تعداد سیگنال‌های متناظر با عناصر حافظه زیاد است، لازم است که تعدادی از آن‌ها به طور تصادفی برای تزریق اشکال انتخاب شوند. برای این کار سیگنال‌ها با احتمال مساوی انتخاب گردیده‌اند.

توزیع زمانی اشکال‌ها در طول اجرای برنامه یکنواخت در نظر گرفته شده است. برای استخراج زمان تزریق اشکال در هر یک از سناریوها، زمان کل شبیه‌سازی برنامه محک مربوطه (بدون تزریق



اشکال) به دست می‌آید و با تقسیم آن به بازه‌های مساوی به تعداد اشکال‌هایی که باید تزریق شوند، زمان هر تزریق اشکال به دست می‌آید.

به این ترتیب، هر سناریو شامل دستوراتی به ترتیب زیر است که شبیه‌ساز را برای تزریق یک اشکال SEU کنترل می‌کند:

- ۱- آماده سازی ابزار شبیه‌سازی
  - ۲- آغاز شبیه‌سازی برنامه و آماده‌سازی ریزپردازنده
  - ۳- شبیه‌سازی برنامه تا زمان تزریق اشکال
  - ۴- خواندن مقدار سیگنال هدف
  - ۵- قفل کردن<sup>۱</sup> مقدار سیگنال بر روی معکوس مقدار خوانده شده
  - ۶- ادامه شبیه‌سازی به اندازه یک پالس کلاک
  - ۷- آزاد کردن سیگنال برای برگشت آن به مقدار واقعی شبیه‌سازی
  - ۸- ادامه شبیه‌سازی تا زمان پایان برنامه اصلی
  - ۹- ذخیره نتایج خروجی، محتوای حافظه و حالت ریزپردازنده برای مقایسه با نسخه مرجع
- نمونه‌ای از یکی از اسکریپت‌های تهیه شده برای یک سناریوی تزریق اشکال به همراه توضیح دستورات آن در پیوست ۳ آمده است.

**شبیه‌سازی و انجام تزریق اشکال:** پس از آماده‌سازی نسخه مرجع و سناریوهای تزریق اشکال، این سناریوها در حلقه‌ای یکی پس از دیگری اجرا می‌شوند. با اجرای هر یک از سناریوها و ذخیره سازی نتایج تزریق اشکال، پایگاه داده‌ی نتایج خام تزریق اشکال ساخته می‌شود که در مراحل بعدی مورد

---

<sup>1</sup> Force

بررسی و تحلیل قرار گیرد. برای هر دنباله از تزریق اشکال‌ها، یک اسکرپیت مادر وجود دارد که اجرای سناریوهای مربوط به آن را کنترل می‌کند.

چنانکه پیش از این اشاره شد، در تزریق اشکال شبیه‌سازی شده یکی از مسائل مهم، سرعت شبیه‌سازی است. در تزریق اشکال‌های انجام شده در این پروژه مشکل سرعت به شکل پیش‌گفته، با کاهش تعداد دستورات برنامه محک و اجتناب از اجرای برنامه‌های بلند حل شده‌است. در صورتی که بتوان مفروضات خاصی (مانند کافی بودن شبیه‌سازی به اندازه تعداد معینی پالس کلاک) در مورد تزریق اشکال داشت، می‌توان از روش‌های دیگری نیز (نظیر تهیه نقاط کنترلی<sup>۱</sup> برای جلوگیری از انجام مجدد شبیه‌سازی تا آن نقاط و ادامه شبیه‌سازی از آن نقاط به اندازه معین) برای افزایش سرعت و ایجاد امکان شبیه‌سازی برنامه‌های بزرگ مانند سیستم‌عامل استفاده کرد.

**تهیه آمار و تحلیل نتایج:** پس از به دست آمدن نتایج تزریق اشکال، نوبت به بررسی نتایج و تهیه آمار می‌رسد. در این بخش خروجی هر تزریق انجام شده، محتوای حافظه آن و حالت ریزپردازنده با متناظر آن‌ها برای نسخه مرجع مقایسه می‌شوند. نتیجه هر تزریق اشکال در یکی از این سه دسته قرار دارد:

۱- **اجرای درست:** به این معنی که خروجی برنامه درست و به موقع بوده است و حافظه و حالت ریزپردازنده نیز در انتهای شبیه‌سازی با حافظه و حالت ریزپردازنده در نسخه مرجع مطابقت دارند.

۲- **خرابی:** که زمانی رخ می‌دهد که داده خروجی برنامه ناقص یا نادرست باشد.

۳- **اشکال متأخر:** که منظور از آن زمانی است که خروجی برنامه درست و کامل است، اما در محتوای حافظه یا حالت ریزپردازنده، نسبت به نسخه مرجع مغایرت وجود دارد که

---

<sup>۱</sup> Checkpoint

نشان‌دهنده اشکالی است که تزریق شده اما به خرابی تزریق نشده است و در ریزپردازنده به صورت نهان باقی مانده است.

هم‌چنین، امکان دارد که برخی از اشکال‌های تزریق شده، بر صحت نتایج تاثیر نگذارند بلکه تنها موجب به تعویق افتادن نمایش نتایج در خروجی شوند. با توجه به آن که برنامه‌ها تنها تا زمان پایان برنامه مرجع (در حالت نبود اشکال) اجرا شده اند، در صورت بروز چنین اتفاقی داده‌ها تا زمان پایان شبیه‌سازی به شکل ناقص در خروجی آمده‌اند که باعث می‌شود این اشکال‌ها در دسته‌بندی بالا در گروه «خرابی» طبقه‌بندی شوند. برای تفکیک این دسته از اشکال‌ها، تمام موارد خرابی کمی پس از زمان پایان برنامه مرجع نیز شبیه‌سازی شده‌اند. در صورتی که پس از گذشت این زمان، برنامه تمام خروجی‌ها را به درستی چاپ کند، این نوع خاص از خرابی را در دسته «خرابی زمانی» قرار می‌دهیم. این نوع از خرابی می‌تواند به ویژه برای سیستم‌های بی‌درنگ که دارای محدودیت‌های زمانی هستند، حائز اهمیت باشد. چنانکه در بخش ۵-۷ مشاهده می‌شود، این خرابی‌ها در نتایج تزریق اشکال واجد نکات قابل توجهی هستند که در آن بخش به تفصیل در مورد آن صحبت شده‌است.

پس از استخراج نتایج و آمار، این نتایج تحلیل و نقاط قوت و ضعف سیستم مشخص می‌شوند. از بازخورد این نتایج برای تصمیم‌گیری در مورد روش تحمل‌پذیری اشکال استفاده می‌شود.

## ۵-۷ تزریق اشکال بر روی ریزپردازنده اولیه

نتایج تزریق اشکال برای هر یک از برنامه‌های محک در هر یک از بخش‌های ریزپردازنده در جدول‌های ۷-۳ تا ۷-۶ آمده است. تعداد تزریق اشکال‌های منجر به هر یک از سه دسته نتیجه تشریح شده در بخش قبل، در این جدول مشخص شده است. همانطور که در بخش پیش گفته شد، پس از مشخص شدن تزریق‌های منجر به خرابی، شبیه‌سازی برای این تزریق‌ها را ادامه داده‌ایم تا خرابی‌های زمانی مشخص شوند. تعداد خرابی‌های زمانی و درصدی از خرابی‌ها که خرابی زمانی بوده‌اند، برای هر یک از برنامه‌های محک و هر یک از بخش‌های ریزپردازنده در این جدول آمده است.

جدول ۷-۳ نتایج تزریق اشکال برای Quicksort

	واحد ریزپردازنده	صحیح	متأخر	خرابی
QSort	Data Cache	59.6	38.8	1.6
	Data Cache Tags	39.7	59.1	1.2
	Instruction Cache	35	42.1	22.9
	Instruction Cache Tags	21.4	78.3	0.3
	Register File	64.65	29.85	5.5

جدول ۷-۴ نتایج تزریق اشکال برای Basicmath

	واحد ریزپردازنده	صحیح	متأخر	خرابی	نرخ خرابی زمانی	خرابی زمانی
BasicMath	Data Cache	36.1	58.8	5.1	1.9	36.25
	Data Cache Tags	27.4	54.1	18.5	18.3	98.92
	Instruction Cache	63.4	2.7	33.9	7.4	21.83
	Instruction Cache Tags	37.6	1.7	60.7	60.6	99.84
	Register File	77.75	12.9	9.35	5.9	63.1

جدول ۷-۵ نتایج تزریق اشکال برای Bitcount

	واحد ریزپردازنده	صحیح	متأخر	خرابی	نرخ خرابی زمانی	خرابی زمانی
BitCount	Data Cache	22	71.6	6.4	0.3	0.05
	Data Cache Tags	8	71.1	20.9	20.8	99.25
	Instruction Cache	52.7	23.6	23.7	4.1	17.3
	Instruction Cache Tags	43.9	12.3	43.8	42	95.89
	Register File	73.6	20.7	5.7	3.2	56.14

جدول ۶-۷ میانگین نتایج تزریق اشکال برای هر سه برنامه

Average	واحد ریزپردازنده	صحیح	متأخر	خرابی	نرخ خرابی زمانی	خرابی زمانی
	Data Cache	39.23	56.4	4.37	0.3	0.05
	Data Cache Tags	25.03	61.43	13.53	20.8	99.25
	Instruction Cache	50.37	22.8	26.83	4.1	17.3
	Instruction Cache Tags	34.3	30.77	34.93	42	95.89
	Register File	72	21.15	6.85	3.2	56.14

**تحلیل نتایج و نقاط قوت و ضعف:** از نکات قابل توجه در نتایج بالا، اشکال‌های متاخر هستند.

چنانکه مشاهده می‌شود، در اغلب موارد اشکال‌های تزریق شده به Cache ها و عناصر حافظه به صورت نهان باقی مانده‌اند. این مساله به این خاطر است که تنها بخش کمی از حافظه در برنامه استفاده می‌شود و بخش عمده‌ای از آن بدون استفاده می‌ماند و بنابراین بخش بزرگی از اشکال‌های تزریق شده در حافظه‌ها هرگز به خطا تبدیل نمی‌شوند.

از نکات جالب دیگر در این نتایج، خرابی‌های زمانی هستند. چنان‌که مشاهده می‌شود، در نزدیک به ۱۰۰ درصد مواردی که اشکال به Tag های داده‌ها در هر یک از Cache ها تزریق شده‌است، خرابی ایجاد شده به شکل خرابی زمانی است. این مطلب به این دلیل است که در صورت رخداد یک تغییر بیت در یکی از Tag ها، یا آن Tag پیش از این نماینده یک مقدار معتبر در Cache نبوده است (اصطلاحاً valid نبوده است) که در این صورت مقدار آن بی اهمیت است و بر روی برنامه تاثیری ندارد، و یا آن که معتبر بوده است که در این صورت نیز با تغییر یک بیت، احتمال این که عدد حاصل، عدد آدرسی از برنامه باشد که در ادامه به آن نیاز داریم و پیش از دسترسی مجدد برای همان داده قبلی یا داده‌ای دیگر، به آن خواهیم رسید، بسیار ناچیز است. به همین دلیل این اشکال‌ها تنها باعث غیر مفید شدن مقدار Tag می‌شوند که باعث می‌شود در دسترسی بعدی برنامه به آن مکان از حافظه، مقدار این Tag و داده

مربوط به آن دوباره از خانه حافظه برداشته شود. همچنین با توجه به این که حافظه نهان در این ریزپردازنده از نوع میان‌نویس<sup>۱</sup> است، تمام تغییراتی که باید در مقدار این خانه در حافظه اصلی داده شوند در همان زمان نوشتن مقادیر جدید در حافظه اصلی ثبت شده‌اند بنابراین داده‌های نوشته شده نیز از دست نرفته‌اند. به این ترتیب اثر این تزریق اشکال تنها عقب افتادن برنامه از زمان خود (به علت زمان تلف شده برای بازخوانی مقدار و بارگذاری مجدد آن در حافظه نهان) است.

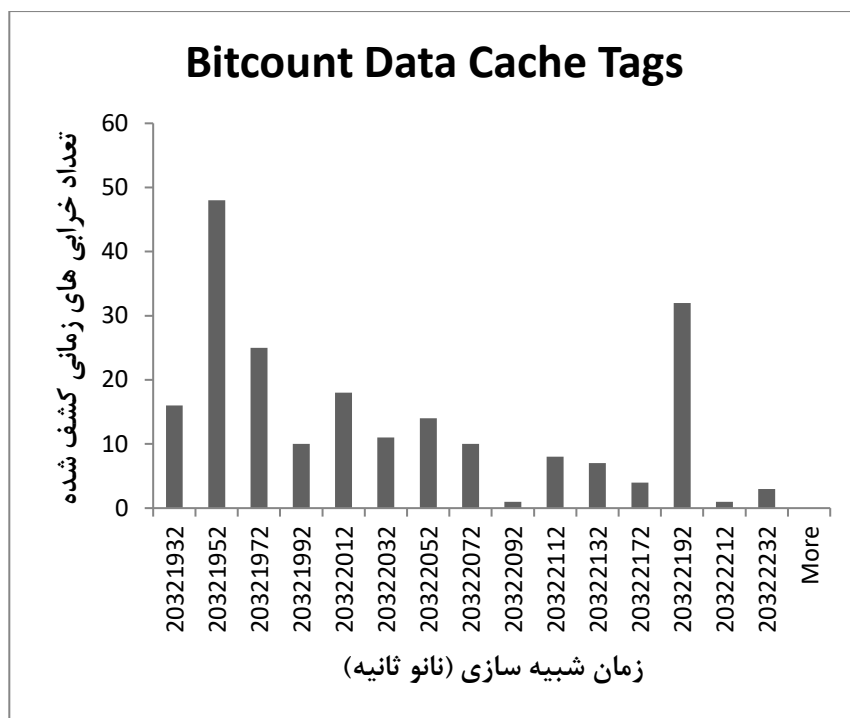
به این ترتیب می‌توان نتیجه گرفت که در سیستم‌های غیر بی‌درنگ که از حافظه نهان میان‌نویس استفاده می‌کنند، Tag‌های Cache‌ها به طور ذاتی در مقابل اشکال‌های از نوع SEU مقاوم هستند. از سوی دیگر این نتیجه برای سیستم‌های دارای حافظه نهان پس‌نویس<sup>۲</sup> معتبر نیست زیرا در این سیستم‌ها داده‌ی متناظر با خانه‌ای از حافظه که مقدار Tag آن از دست می‌رود، ممکن است تغییراتی داشته باشد که ذخیره نشده باشند و رخداد اشکال در Tag می‌تواند منجر به این شود که نه تنها مقدار تغییر داده شده در محل مناسب ذخیره نشود، بلکه مقدار محل دیگری از حافظه (که آدرس آن با Tag تغییر کرده و جعلی یکسان است) به علت نوشته شدن تغییرات روی آن، دچار خطا شود.

برای ادامه دادن برنامه جهت یافتن خرابی‌های زمانی، یک سوال مطرح این است که چه مقدار باید برنامه را ادامه داد تا خرابی‌های زمانی مشخص شوند. در نمودارهای ۷-۲ و ۷-۳، تعداد خرابی‌هایی که به عنوان خرابی زمانی شناسایی می‌شوند بر حسب زمانی که شبیه‌سازی را ادامه می‌دهیم، دیده می‌شود. زمان‌های داده شده، زمان مجازی شبیه‌سازی بر حسب نانو ثانیه هستند. زمان اجرای برنامه Bitcount که این دو نمودار برای آن رسم شده‌اند، در حدود ۲۰۳۲۰۰۰۰ نانو ثانیه است.

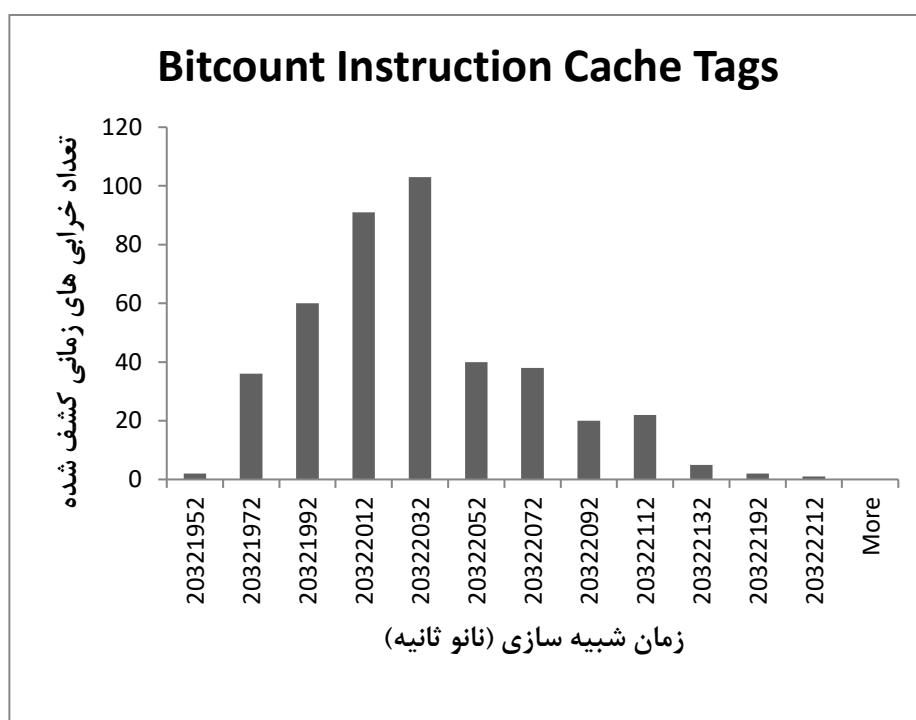
---

<sup>۱</sup> Write-through

<sup>۲</sup> Write-back



شکل ۷-۲ کشف خرابی زمانی بر حسب زمان برای حافظه نهان داده



شکل ۷-۳ کشف خرابی زمانی بر حسب زمان برای حافظه نهان دستور

با توجه به شکل مشاهده می‌شود که در مورد حافظه نهان داده، خرابی‌ها با شبیه‌سازی به اندازه چند کلاک معلوم می‌شوند در حالی که در مورد دستورات به مدت بیشتری شبیه‌سازی نیاز است. این مطلب می‌تواند به این دلیل باشد که در زمان واکنشی دستور، رخداد خطای حافظه نهان موجب نیاز به مراجعه به حافظه و توقف خط لوله<sup>۱</sup> می‌شود که سرشار زمانی بیشتری نسبت به دسترسی به داده (که در آن توقف خط لوله نداریم) دارد.

---

<sup>۱</sup> Pipeline Stall



## فصل ۸

### ۸ نتیجه‌گیری

در این پایان‌نامه، فرآیند تهیه یک ریزپردازنده تحمل‌پذیر اشکال برای سیستم‌های نهفته اجرا و یک ریزپردازنده با قابلیت‌های تحمل‌پذیری اشکال ایجاد شده است. ریزپردازنده ساخته شده با استفاده از برنامه‌های استاندارد محک، صحت‌سنجی شده است تا از عملکرد آن اطمینان حاصل شود. همچنین میزان قابلیت اطمینان ریزپردازنده با استفاده از تزریق اشکال شبیه‌سازی شده ارزیابی شده است.

سیستم‌های نهفته کاربرد فراوانی در عرصه‌های مختلف زندگی امروزه دارند. با توجه به این که بسیاری از این سیستم‌ها در کاربردهای بحرانی-امن مورد استفاده قرار می‌گیرند، قابلیت اطمینان یکی از نیازمندی‌های مهم آن‌هاست. برای ایجاد قابلیت اطمینان در این سیستم‌ها لازم است از روش‌های تحمل‌پذیری اشکال استفاده و کارایی این روش‌ها ارزیابی شوند.

ریزپردازنده تهیه شده در این پروژه بر مبنای معماری SPARC و با استفاده از ریزپردازنده LEON2 به عنوان ریزپردازنده مرجع، ساخته شده است و در آن از واحدهای تحمل‌پذیر اشکال موجود از قبل، شامل واحدهای محاسبه و منطق، بانک ثبات، واحد کنترل و وسائل جانبی، استفاده شده است. این واحدها به شکل مرحله به مرحله یکپارچه‌سازی شدند و در هر مرحله صحت یکپارچه‌سازی مورد آزمون قرار گرفت.

با توجه به این که بسیاری از کاربردهای سیستم‌های نهفته، مانند کاربردهای مخابراتی، نیاز به انجام محاسبات ممیز شناور دارند، یک واحد ممیز شناور نیز در این پروژه برای ریزپردازنده هدف طراحی و پیاده‌سازی شد. این واحد که از ابتدا در ریزپردازنده مرجع وجود ندارد، برای پشتیبانی دقت ساده و مضاعف مطابق دستورالعمل SPARC نسخه ۸ و استاندارد IEEE-754، طراحی و پیاده‌سازی و همراه با واحدهای دیگر در ریزپردازنده نهایی مجتمع شده است. همچنین برای ایجاد تحمل‌پذیری اشکال، روش‌هایی برای این واحد ارائه شد که هر دو بر مبنای استفاده مجدد از سخت‌افزار قرار دارند. برای دقت مضاعف، یک روش ایجاد تحمل‌پذیری اشکال برای واحد محاسبه و منطق اعداد صحیح، با عملیات ممیز شناور تطبیق داده شده است. برای دقت ساده روشی برای استفاده از سخت‌افزار موجود برای دقت مضاعف جهت ایجاد تحمل‌پذیری اشکال ارائه شده است.

برای صحت‌سنجی ریزپردازنده از بسته برنامه‌های محک استاندارد MiBench استفاده شد. این بسته که یکی از مجموعه برنامه‌های محک استاندارد برای سیستم‌های نهفته است، از کاربرد گسترده‌ای برخوردار است و به همین دلیل برای انجام صحت‌سنجی انتخاب گردید. با توجه به زمینه کاربرد ریزپردازنده هدف، مجموعه Automotive از این بسته آزمون اجرا شد. برای آزمون جامع‌تر و دقیق‌تر و رعایت محیط اجرای برنامه‌های محک و افزایش پوشش، از سیستم‌عامل در صحت‌سنجی ریزپردازنده استفاده و سیستم‌عامل بی‌درنگ RTEMS با موفقیت بر روی ریزپردازنده اجرا شد.

برای ارزیابی قابلیت اطمینان سیستم از تزریق اشکال شبیه‌سازی شده با مدل اشکال SEU استفاده شد. مدل SEU به علت رخداد زیاد این نوع اشکال در کاربردهای فضایی، یکی از مدل‌های بسیار رایج در ارزیابی قابلیت اطمینان است. شبیه‌سازی ریزپردازنده مرجع و تزریق اشکال بر روی آن با اجرای برنامه‌هایی از بسته MiBench انجام گرفت و میزان اشکال‌های بدون تاثیر، منجر به خرابی و متأخر اندازه‌گیری شد. همچنین مشاهده شد که تقریباً تمام از خرابی‌هایی که در اثر تزریق اشکال در Tag‌های حافظه نهان رخ می‌دهد از نوع خرابی زمانی است به این معنی که نتایج با تأخیر حاصل می‌شوند و در صورت ادامه شبیه‌سازی، نتایج به درستی در خروجی ظاهر خواهند شد. به علاوه دیده شد که تعداد زیادی از اشکال‌های تزریق شده در حافظه نهان، به صورت متأخر باقی می‌مانند.

کارهای آتی بر روی این ریزپردازنده می‌توانند در چند جهت متمرکز شوند. یکی از کارهای لازم، ایجاد تحمل‌پذیری اشکال در سایر واحدهای ریزپردازنده نظیر حافظه نهان، واحد مدیریت حافظه و گذرگاه داده است. همچنین با توجه به این که برخی از روش‌های به کار رفته برای ایجاد تحمل‌پذیری اشکال در این ریزپردازنده روش‌های کشف خطا هستند و قابلیت تصحیح خطا را ندارند، وجود یک واحد بازیابی (Recovery) برای این سیستم ضروری به نظر می‌رسد.

کار دیگری که باید در ادامه این پروژه انجام شود، پیاده‌سازی روش‌های تحمل‌پذیری اشکال ارائه شده برای واحد ممیز شناور و ارزیابی قابلیت اطمینان آن‌هاست. همچنین می‌توان این روش‌ها را با سایر روش‌های ارائه شده برای تحمل‌پذیری اشکال، به ویژه از نظر سربار زمانی و سخت‌افزاری تحلیل شده، مقایسه کرد.

برای ارزیابی بهتر تحمل‌پذیری اشکال می‌توان علاوه بر روش گفته شده، از تزریق اشکال فیزیکی مبتنی بر FPGA استفاده کرد. روش‌های متعددی برای انجام این کار وجود دارند که به برخی از آن‌ها در بخش ۲-۱-۷ اشاره شد. همچنین می‌توان مدل‌های دیگر اشکال نظیر اشکال چند بیتی (MBU) را برای ارزیابی تحمل‌پذیری اشکال استفاده کرد. با استفاده از روش‌های افزایش سرعت شبیه‌سازی و تزریق

---

اشکال، می‌توان شبیه‌سازی در سطوح پایین‌تر تجرید را نیز برای مدل‌سازی دقیق‌تر و افزایش تنوع اشکال‌هایی که تزریق می‌شوند، به کار گرفت.

1. Jidan Al-Eryani, "Floating Point Unit," 2006.
2. Michael Barr and Anthony Massa, *Programming Embedded Systems*: O'Reilly Media, 2006.
3. Brian Case, "SPARC Architecture," M. Slater, Ed., ed San Diego, CA: Academic Press, 1992, pp. 33-45.
4. Chien-in Henry Chen, "Behavioral test generation / fault simulation," *IEEE Potentials*, pp. 27-32, 2003.
5. Yirng-An Chen, Edmund Clarke, Pei-Hsin Ho, Yatin Hoskote, Timothy Kam, Manpreet Khaira, John O'Leary, and Xudong Zhao, " Verification of all circuits in a floating-point unit using word-level model checking " in *Lecture Notes in Computer Science*. vol. 1166/1996, ed: Springer Berlin / Heidelberg, 1996, pp. 19-33.
6. Sivarama P. Dandamudi, *Guide to RISC Processors: for Programmers and Engineers*: Springer, 2005.
7. Alireza Ejlali, Seyed Ghassem Miremadi, Hamid R. Zarandi, Ghazanfar Asadi, and Syavash, "A Hybrid Fault Injection Approach Based on Simulation and Emulation Co-operation," presented at the International Conference on Dependable Systems and Networks (DSN'03), 2003.
8. Mehdi Fazeli, Seyed Ghassem Miremadi, and Alireza Namazi, "OperandWidth Aware Hardware Reuse: A Low Cost Soft-Error Tolerant Technique to ALU Design in Embedded Processors," 2010.
9. Jiri Gaisler, "A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture," presented at the International Conference on Dependable Systems and Networks (DSN'02), 2002.
10. Jiri Gaisler, "LEON2 Processor User's Manual [Version 1.0.30]," 2005.
11. Jiri Gaisler, "BCC - Bare-C Cross-Compiler User's Manual [Version 1.0.32]," Aeroflex Gaisler AB2009.
12. Jiri Gaisler, "TSIM2 Simulator User's Manual: ERC32/LEON2/LEON3 [Version 2.0.13]," Aeroflex Gaisler AB2009.
13. Jiri Gaisler. (2010, 6/26/2010 8:07 PM). *Operating Systems* Available: [http://www.gaisler.com/cms/index.php?option=com\\_content&task=view&id=327&Itemid=218](http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=327&Itemid=218)
14. Jiri Gaisler, Edvin Catovic, Marko Isomäki, Kristoffer Glembo, and Sandi Habinc, "GRLIB IP Core User's Manual [Version 1.0.16]," ed: Gaisler Research, 2007.

15. Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Schirnerm Gunar, *Embedded System Design: Modeling, Synthesis and Verification*: Springer, 2009.
16. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," presented at the IEEE International Workshop on Workload Characterization (WWC 2001), 2001.
17. Ravishankar K. Iyer and Dong Tang, "Experimental Analysis of Computer System Dependability," D. K. Pradhan, Ed., ed Upper Saddle River, NJ, 282-392.: Prentice-Hall, 1996.
18. Barry W. Johnson, *Design & analysis of fault tolerant digital systems*: Addison-Wesley, 1988.
19. Israel Koren and C. Mani Krishna, *Fault-Tolerant Systems*. San Francisco, CA: Morgan-Kaufman Publishers, 2007.
20. Peter Marwedel, *Embedded System Design*: Springer, 2006.
21. Mentor Graphics Inc., "ModelSim® User's Manual: Software Version 6.6," Mentor Graphics Corporation 2010.
22. Alireza Namazi, Seyed Ghassem Miremadi, and Alireza Ejlali, "A High Speed and Low Cost Error Correction Technique for the Carry Select Adder," presented at the International Conference on Availability, Reliability and Security, 2009.
23. Tammy Noergaard, *Embedded Systems Architecture : A Comprehensive Guide for Engineers and Programmers*: Elsevier, 2005.
24. OAR Corporation, "RTEMS Filesystem Design Guide [Edition 4.9.99.0, for RTEMS 4.9.99.0]," 7/18/2010 2010.
25. Richard P. Paul, *SPARC Architecture, Assembly Language Programming, and C*, 2nd Edition ed.: Prentice-Hall, 2000.
26. Mohammad-Hamed Razmkhah, Seyed Ghassem Miremadi, and Alireza Ejlali, "A Micro-FT UART for Safety-Critical SoC-Based Applications," presented at the International Conference on Availability, Reliability and Security, 2009.
27. SPARC International Inc., "The SPARC Architecture Manual, Version 8," ed. USA: SPARC International, Inc., 1992.
28. Stephan Wong, Stamatis Vassiliadis, and Sorin Cotofana, "Embedded Processors: Characteristics and Trends," Computer Engineering Laboratory, Delft CE-TR-2004-03, 2004.
29. Hamid R. Zarandi, Seyed Ghassem Miremadi, and Alireza Ejlali, "Dependability Analysis Using a Fault Injection Tool Based on Synthesizability of HDL Models,"

presented at the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), 2003.

۳۰. محمد حامد رزمخواه، "وسائل جانبی بر روی تراشه با قابلیت تحمل پذیری اشکال برای پردازنده های نهفته"، پایان نامه ی کارشناسی ارشد، دانشکده ی مهندسی کامپیوتر، دانشگاه صنعتی شریف، ۱۳۸۷.

۳۱. حمید رضا زرندی، "ارزیابی تحمل پذیری خطا مبتنی بر شبیه سازی با استفاده از VHDL و Verilog"، پایان نامه ی کارشناسی ارشد، دانشکده ی مهندسی کامپیوتر، دانشگاه صنعتی شریف، ۱۳۸۱.

۳۲. سید محمد حسین شکریان، "طراحی و ارزیابی یک واحد پردازش اعداد حقیقی تحمل پذیر اشکال برای پردازنده های نهفته"، پایان نامه ی کارشناسی ارشد، دانشکده ی مهندسی کامپیوتر، دانشگاه صنعتی شریف، ۱۳۸۷.

۳۳. حسن قاسم زاده محمدی، "واحد کنترل تحمل پذیر اشکال برای ریزپردازنده های نهفته"، پایان نامه ی کارشناسی ارشد، دانشکده ی مهندسی کامپیوتر، دانشگاه صنعتی شریف، ۱۳۸۷.

۳۴. علیرضا نمازی، "طراحی، پیاده سازی و ارزیابی یک واحد محاسباتی / منطقی و بانک ثبات تحمل پذیر اشکال برای پردازنده های نهفته"، پایان نامه ی کارشناسی ارشد، دانشکده ی مهندسی کامپیوتر، دانشگاه صنعتی شریف، ۱۳۸۷.

پیوست‌ها





## ۱ پیوست ۱: نحوه‌ی اجرای سیستم‌عامل RTEMS روی ریزپردازنده

### SPARC

با توجه به اینکه RTEMS از ریزپردازنده SPARC پشتیبانی می‌کند و سربراندکی نیز دارد، برای اجرا و آزمون روی آن برگزیده شد. در ادامه فرایند اجرای RTEMS روی کد SPARC در شبیه‌ساز ModelSim شرح داده می‌شود.

### ۱-۱ نصب

در ابتدا باید محیط توسعه RTEMS آماده شود. به این منظور، باید RCC (RTEMS LEON/ERC32 Cross-Compiler System) را از سایت [www.gaisler.com](http://www.gaisler.com) دانلود و نصب نمود. برای استفاده از این ابزار در ویندوز توصیه می‌شود از GRTTools که از همان سایت قابل دریافت است، استفاده کرد. پس از نصب فایل‌های RCC (توسط GRTTools در ویندوز یا به طور عادی در

لینوکس) باید دایرکتوری bin آن در PATH سیستم قرار گیرد تا برنامه‌هایی نظیر sparc-rtems-gcc از خط فرمان به راحتی قابل دسترسی باشند.

## ۱-۲ نحوه ایجاد Makefile و loader.c

پس از نصب این ابزار می‌توان به کمک آن هر برنامه‌ای را تحت آن در ریزپردازنده نهفته اجرا کرد. برای این کار دو فایل make (Makefile) و loader.c را تهیه می‌کنیم. محتوای Makefile در ادامه آمده است:

```
1 CC=sparc-rtems-g++
2 LD=sparc-rtems-ld
3 CFLAGS=-O2 -mcpu=v8 -msoft-float -Wall -g -gleon2
4 LDFLAGS=-r
5 LIBS=-lm
6 TARGET=qsrt_small
7 INPUTFILE=input_small.dat
8
9 all: $(TARGET)
10
11 $(TARGET): tarfile loader.o $(TARGET).o
12     $(LD) $(LDFLAGS) -o temp.o loader.o $(TARGET).o -b binary tarfile
13     $(CC) $(CFLAGS) $(LIBS) temp.o -o $(TARGET)
14     sparc-rtems-strip $(TARGET)
15     sparc-rtems-objcopy --remove-section=.comment $(TARGET)
16     sparc-rtems-objcopy -O srec $(TARGET) sdram.rec
17     sparc-rtems-objdump -s $(TARGET) > ram.dat
18     sparc-rtems-size $(TARGET)
19
20 tarfile: $(INPUTFILE)
21     tar cf tarfile $(INPUTFILE)
22
23 clean:
24     rm -rf $(TARGET) *.o sdram.rec ram.dat tarfile
```

در خط ۱ و ۲، نوع کامپایلر و لینکر مشخص می‌شوند. در خط ۳ پارامترهایی که هنگام کامپایل و relocation استفاده خواهند شد تعریف می‌شوند:

- O2 : کد را بهینه (حدأکثر کارایی با حدأقل اندازه کد) می‌کند.
- mcpu=v8 : دستورات ضرب و تقسیم را به صورت سخت‌افزاری تولید می‌کند.
- msoft-float : دستورات ممیز شناور را به صورت نرم‌افزاری تولید می‌کند. با حذف این پارامتر، ریزپردازنده باید مجهز به یک FPU باشد و این FPU در تنظیمات آن فعال شده باشد.
- Wall : باعث می‌شود میزان warning دادن کامپایلر حدأکثر باشد. این پارامتر در این توسعه برنامه بسیار مناسب است.
- qleon2 : کد نهایی را برای leon2 تولید می‌کند. مشابه این پارامتر qleon3 و qleon3mp نیز برای LEON 3 نیز وجود دارند.

در خط ۴ پارامترهای مربوط به لینکر، تعریف می‌شوند:

- r : باعث می‌شود کد تولید شده relocatable باشد.
- در خط ۵، کتابخانه‌هایی که می‌خواهیم به برنامه اصلی لینک شوند ذکر می‌شوند. lm دسترسی به کتابخانه math را فراهم می‌کند. دقت شود، صرف قرار دادن `#include <math.h>` در ابتدای برنامه کافی نیست و باید کتابخانه مربوط به آن نیز با برنامه به این صورت لینک شود. در غیراینصورت با خطا مواجه خواهید شد.

خط ۶ اسم فایل اصلی که قرار است اجرا شود (`main()` در آن قرار دارد) ذکر می‌شود. INPUTFILE فایلی (یا فایل‌هایی) هستند که برنامه اصلی به آن‌ها احتیاج دارد و باید در حافظه به صورت IMFS (In-Memory File System) قرار بگیرند تا برنامه به آن‌ها دسترسی داشته باشد. در خط ۱۱ باید لیست فایل‌هایی که قرار است کامپایل شوند با پسوند `.o` قرار گیرند. این فایل‌ها باید در خط ۱۲ نیز ذکر شوند تا با loader لینک شوند.

## توجه:

- ۱- فایل‌هایی که می‌خواهیم به صورت IMFS در حافظه قرار دهیم را باید tar کرده و با نام tarfile (نام دیگر قابل قبول نیست!) با برنامه اصلی لینک کنیم. این کار را Makefile انجام می‌دهد. تنها کافیت لیست فایل‌ها را در متغیر INPUTFILE قرار دهیم.
- ۲- برای اینکه فرآیند لینک به درستی انجام شود، نباید از sparc-rtems-gcc استفاده نمود. در عوض باید از sparc-rtems-g++ استفاده کرد.

خط ۱۳ عملیات relocation را انجام می‌دهد. خط ۱۴ و ۱۵ قسمت‌های زائد برنامه را حذف می‌کنند. (البته این حذف اجباری است!) در نهایت در خطوط ۱۶ و ۱۷، فایل‌های sdram.rec و ram.dat ساخته می‌شوند. در نهایت نیز در خط ۱۸، اندازه این فایل‌ها نشان داده می‌شوند. (این خط اختیاری است.) خط ۲۰ و ۲۱ برای ساخت tarfile است. خط ۲۳ و ۲۴ برای پاک کردن فایل‌های تولید شده بکار می‌روند که با دستور make clean استفاده می‌شوند.

محتوای فایل loader.c به صورت زیر می‌باشد:

```

1  #include <rtems.h>
2
3  /* configuration information */
4  #define CONFIGURE_INIT
5
6  #include <bsp.h> /* for device driver prototypes */
7  /*forward declaration*/
8  rtems_task Init(rtems_task_argument argument); 9
10 /* configuration information */
11 #define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
12 #define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
13 #define CONFIGURE_MAXIMUM_TASKS                20
14 #define CONFIGURE_RTEMS_INIT_TASKS_TABLE
15 #define CONFIGURE_EXTRA_TASK_STACKS (3 * RTEMS_MINIMUM_STACK_SIZE)
16 #define CONFIGURE_INIT_TASK_STACK_SIZE          3000000
17 //Used for IMFS
18 #define CONFIGURE_USE_IMFS_AS_BASE_FILESYSTEM
19 #define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTOR 10
20
```

```

21  #include <rtems/confdefs.h>
22
23  #include <stdio.h>
24  #include <stdlib.h>
25  #include <string.h>
26  #include <unistd.h>
27  #include <errno.h>
28  #include <rtems.h>
29  #include <fcntl.h>
30  #include <rtems/error.h>
31  #include <rtems/dosfs.h>
32  #include <dirent.h>
33  #include <ctype.h>
34  #include <rtems/ide_part_table.h>
35  #include <rtems/libcsupport.h>
36  #include <rtems/fsmount.h>
37  #include <rtems/untar.h>
38  #include <rtems/imfs.h>
39  #include <math.h>
40
41  extern int _binary_tarfile_start;
42  extern int _binary_tarfile_size;
43  #define TARFILE_START _binary_tarfile_start
44  #define TARFILE_SIZE _binary_tarfile_size
45
46  extern int main(int argc, char *argv[]);
47
48  rtems_task Init( rtems_task_argument ignored ){
49      int argc=2;
50      char *argv[]={"qsort_small","input_small.dat"};
51      Untar_FromMemory ((unsigned char*)&TARFILE_START),(int)&TARFILE_SIZE);
52      main(argc,argv);
53      exit(0);
54  }

```

تنظیمات RTEMS ثابت هستند. در خط ۴۹ و ۵۰ پارامترهای تابع main اصلی تنظیم می‌شوند.

اگر تابع main دارای این پارامترها نباشد باید خط prototype (خط ۴۶) و خط فراخوانی آن (خط ۵۲)

اصلاح شوند. در خط ۵۱ محتوای فایل tarfile در حافظه باز می‌شود تا به صورت فایل سیستم در اختیار

برنامه اصلی باشد. در نهایت پس از آماده شدن، محیط سیستم عامل برنامه اصلی (که ورودی آن به صورت یک تابع main می باشد) در خط ۵۲ فراخوانده می شود. در خط ۵۳ پس از اتمام کار باید تابع exit فراخوانده شود که شبیه سازی به پایان برسد.

### ۱-۳ استفاده از سیستم عامل در ModelSim

فایل های تولید شده توسط فرآیند مذکور (sram.rec و ram.dat) به همراه romsd.dat (که از tsource اصلی برداشته شده و به نوعی محتوای prom برای راه اندازی اولیه ریزپردازنده به حساب می آید و ثابت نیز می باشد) باید در پروژه اصلی در شاخه tsource قرار داده شوند. به علاوه فایل در device.vhd باید دو تغییر صورت گیرد. این تغییرات در جدول پ-۱ آمده است.

جدول پ-۱ تغییرات لازم در فایل device.vhd

مقدار پس از تغییر	مقدار پیش از تغییر
sdramen => true	sdramen => false
uart => true	uart => fales

در نهایت پس از کامپایل، همانند **Error! Reference source not found.** باید تنظیم tb\_func\_sd برای شبیه سازی انتخاب و اجرا شود.

Library			
Name	Type	Path	
sw204420	Entity	C:\HW\HDS\FPU\leon\tech_fs90.vhd	
syncram	Entity	C:\HW\HDS\FPU\leon\tech_map.vhd	
target	Package	C:\HW\HDS\FPU\leon\target.vhd	
tb_full	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_full_disas	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_func8	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_func8_disas	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_func16	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_func16_disas	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_func32	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_func32_disas	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_func_sdrām	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_func_sdrām_disas	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_mem	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_mem_disas	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	
tb_mmu	Config	C:\HW\HDS\FPU\tbench\tbleon.vhd	

شکل پ- ۱ تنظیم لازم برای شبیه‌سازی ریزپردازنده در ModelSim



## ۲ پیوست ۲: آماده‌سازی پروژه LEON

در این پروژه از LEON 2 نسخه leon2-1.0.32-xst استفاده شد که تا زمان نگارش این گزارش آخرین نسخه منتشر شده LEON 2 به حساب می‌آید.<sup>۱</sup> به علاوه از sparc-elf-3.4.4 به عنوان کامپایلر (تحت لینوکس) استفاده شده است. برای شبیه‌سازی کد ریزپردازنده از نرم‌افزار Mentor Graphics QuestaSim 6.5 استفاده شد که علاوه بر قابلیت‌های ModelSim، قابلیت‌هایی برای صحت‌سنجی (verification) نیز ارائه می‌دهد.

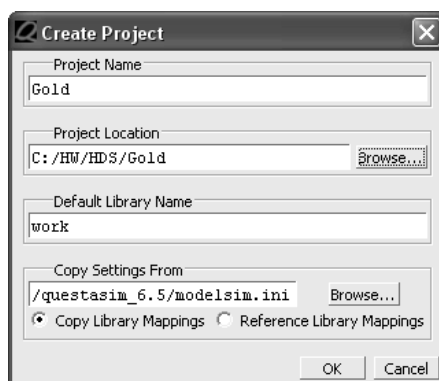
برای استفاده از کد LEON در QuestaSim باید مراحل زیر طی شود:

۱- پروژه‌ای در QuestaSim ایجاد شود. (در شکل پ-۲، Gold به عنوان نام پروژه انتخاب شده است.) دقت کنید که در ادامه فرض شده Default Library، *work* نام دارد.

---

<sup>۱</sup> در حال حاضر شرکت Aeroflex Gaisler روی LEON 3 تمرکز کرده است. ویژگی بارز LEON 3، پشتیبانی از چندپردازنده است.



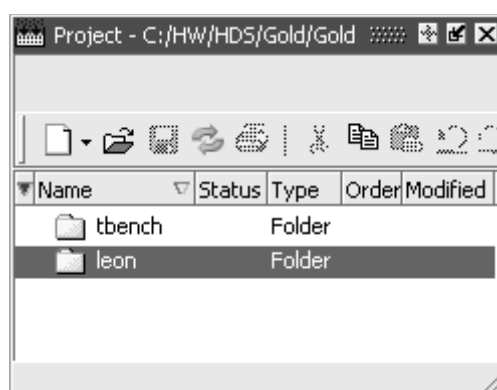


شکل پ-۲ ایجاد پروژه در ModelSim

۲- پوشه‌های leon، tbench و tsource مربوط به کد اصلی را در پوشه پروژه کپی کنید.

۳- پیش از افزودن فایل‌ها بایستی همانند شکل پ-۳ دو پوشه به نام‌های leon و tbench در

QuestaSim (داخل پروژه‌ای که ایجاد کردید) بسازید.

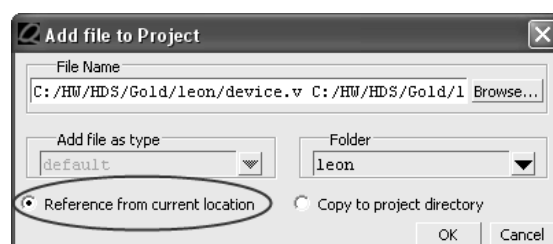


شکل پ-۳ افزودن دو پوشه‌ی tbench و leon به پروژه

۴- فایل‌های درون پوشه leon را به پوشه leon درون پروژه و نیز فایل‌های پوشه tbench را به پوشه

tbench بیفزایید. دقت کنید که بایستی فایل‌های افزوده شده به پروژه به فایل‌های اصلی اشاره

(reference) کنند. این مرحله در شکل پ-۴ آمده است.



شکل پ-۴ افزودن فایل‌ها و ارجاع دادن به آن‌ها

۵- برای کامپایل پروژه کافیسست از فایل compile.do که در پوشه

leon2-1.0.32-xst\sim\modelsim قرار دارد استفاده کنید. البته دقت کنید که باید خط اول

این فایل به صورت زیر تغییر کند:

```
vlib leon/work → vlib work
```

پس از اجرای این فایل (در مراحل کامپایل) هشدارهای زیر ممکن است داده شود:

- چون Library به نام work قبلاً ایجاد شده بود، این هشدار داده می‌شود.

```
# ** Warning: (vlib-34) Library already exists at "work".
```

- این هشدارها به خاطر وجود نداشتن resolution function برای سیگنال‌های مذکور می‌باشد.

```
# ** Warning: [5] leon/proc.vhd(43): Nonresolved signal 'ahbo' may have multiple sources.
```

```
#Drivers:
```

```
#leon/proc.vhd(180):Instantiation c0
```

```
#leon/proc.vhd(185):Instantiation c0
```

یا

```
# **Warning: [5] leon/proc.vhd(156): Nonresolved signal 'crami' may have multiple sources.
```

```
#Drivers:
```

```
#leon/proc.vhd(180):Instantiation c0
```

```
#leon/proc.vhd(185):Instantiation c0
```

- چون حالت پیش‌فرض نسخه فعلی QuestaSim، VHDL 1993 می‌باشد، در نتیجه

هشدارهای مبنی بر استفاده صرف از VHDL 1987 داده می‌شود. (در راهنمای LEON 2

آمده است که فقط از VHDL 1987 استفاده شده است).

```
# ** Warning: tbench/iram.vhd(186): (vcom-1194) FILE declaration was written using VHDL 1987 syntax.
```

```
# ** Warning: tbench/mt48lc16m16a2.vhd(493): (vcom-1194) FILE declaration was written using VHDL 1987 syntax.
```

```
# ** Warning: tbench/mt48lc16m16a2.vhd(494): (vcom-1194) FILE declaration was written using VHDL 1987 syntax.
```

```
# ** Warning: tbench/mspram.vhd(223): (vcom-1194) FILE declaration was written using VHDL 1987 syntax.
```

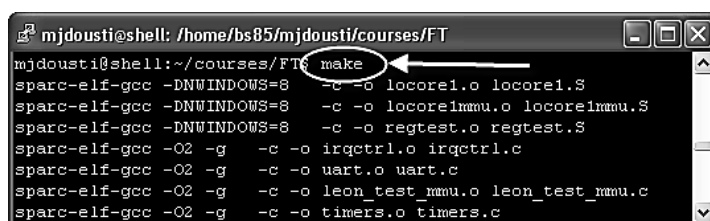
# \*\* Warning: tbench/mspram.vhd(231): (vcom-1194) FILE declaration was written using VHDL 1987 syntax.

۶- برای شبیه‌سازی پرازنده، بایستی برنامه‌ای روی آن اجرا کنید. به این منظور از فایل‌های موجود

در پوشه tsource (به خصوص Makefile) استفاده کنید. برای استفاده از کامپایلر sparc-elf-

3.4.4 بایستی آدرس sparc-elf-3.4.4/bin را در PATH قرار دهید. سپس مطابق شکل پ-۵

به شاخه tsource رفته و make را اجرا کنید.



```
mjdousti@shell: /home/bs85/mjdousti/courses/FT
mjdousti@shell:~/courses/FT$ make
sparc-elf-gcc -DNWINDOWS=8 -c -o locore1.o locore1.S
sparc-elf-gcc -DNWINDOWS=8 -c -o locore1mmu.o locore1mmu.S
sparc-elf-gcc -DNWINDOWS=8 -c -o regtest.o regtest.S
sparc-elf-gcc -O2 -g -c -o irqctrl.o irqctrl.c
sparc-elf-gcc -O2 -g -c -o uart.o uart.c
sparc-elf-gcc -O2 -g -c -o leon_test_mmio.o leon_test_mmio.c
sparc-elf-gcc -O2 -g -c -o timers.o timers.c
```

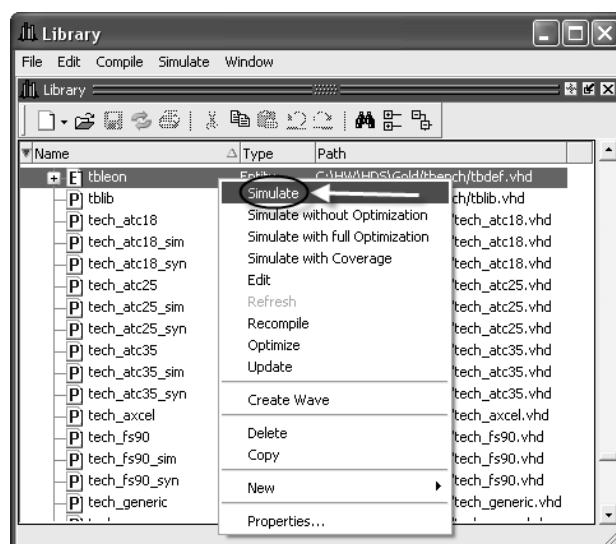
شکل پ-۵ اجرای فرمان make برای ساخت پروژه

در نهایت فایل‌های تولید شده را در پوشه مربوط به پروژه QuestaSim تحت عنوان tsource کپی

نمایید.

۷- برای شبیه‌سازی مطابق شکل پ-۶، tbleon را کتابخانه work را از library انتخاب کنید. روی

آن کلیک راست کرده و Simulate را انتخاب کنید.



شکل پ-۶ شبیه‌سازی تنظیم tbleon در ModelSim

۸- پیش از شبیه‌سازی، QuestaSim سعی می‌کند بهینه‌سازی‌هایی را روی طراحی‌های کامپایل

شده انجام دهد. اما این بهینه‌سازی به هشدارهایی می‌انجامد که در ۲ گروه جای می‌گیرند:

I. هشدارهای مربوط به نبودن کتابخانه unisim. این کتابخانه به علت استفاده از

تکنولوژی‌های شرکت Xilinx (FPGAهای<sup>۱</sup> vertex) می‌باشد. این کتابخانه همراه برنامه

Xilinx ISE وجود دارد و به طور رایگان در اینترنت نیز وجود دارد. از کل این کتابخانه

تنها ۳ فایل زیر لازم هستند:

- ۱- unisim\_VPKG.vhd
- ۲- unisim\_VCOMP.vhd
- ۳- unisim\_VITAL.vhd

این کتابخانه باید به پروژه اضافه شده و خطوط زیر به ابتدای فایل

compile.do (پس از خط vlib) اضافه شوند:

```
vcom -quiet unisim/unisim_VPKG.vhd
vcom -quiet unisim/unisim_VCOMP.vhd
vcom -quiet unisim/unisim_VITAL.vhd
```

در انتها ساختار فایل‌ها در پروژه بایستی به صورت شکل پ-۷ باشد.



شکل پ-۷ ساختار نهایی فایل‌ها در پروژه‌ی شبیه‌سازی

<sup>۱</sup> این فایل‌ها در پروژه به جای vertex, virtex نامیده شده‌اند!

به علاوه خطوط زیر باید در فایل‌های پروژه از حالت comment خارج شوند:

tech\_vertex.vhd

خط ۱۰۶۶ و ۱۰۶۷:

<pre>--library unisim;</pre>	→	<pre>library unisim;</pre>
<pre>--use unisim.vcomponents.all;</pre>		<pre>use unisim.vcomponents.all;</pre>

tech\_vertex2.vhd

خط ۸۸۶ و ۸۸۷:

<pre>--library unisim;</pre>	→	<pre>library unisim;</pre>
<pre>--use unisim.vcomponents.all;</pre>		<pre>use unisim.vcomponents.all;</pre>

با انجام این کار و کامپایل مجدد، (با compile.do جدید) اکثر هشدارها از بین می‌روند.

II. تعدادی هشدار نیز مربوط به نبود واحد FPU می‌باشند که بایستی با یکپارچه‌سازی واحد

تحمل‌پذیر اشکال شده این هشدارها نیز از بین بروند. دقت کنید که خود LEON دارای

FPU نمی‌باشد و در یک بسته جداگانه عرضه می‌شود. به علاوه می‌توان از FPUهای دیگر

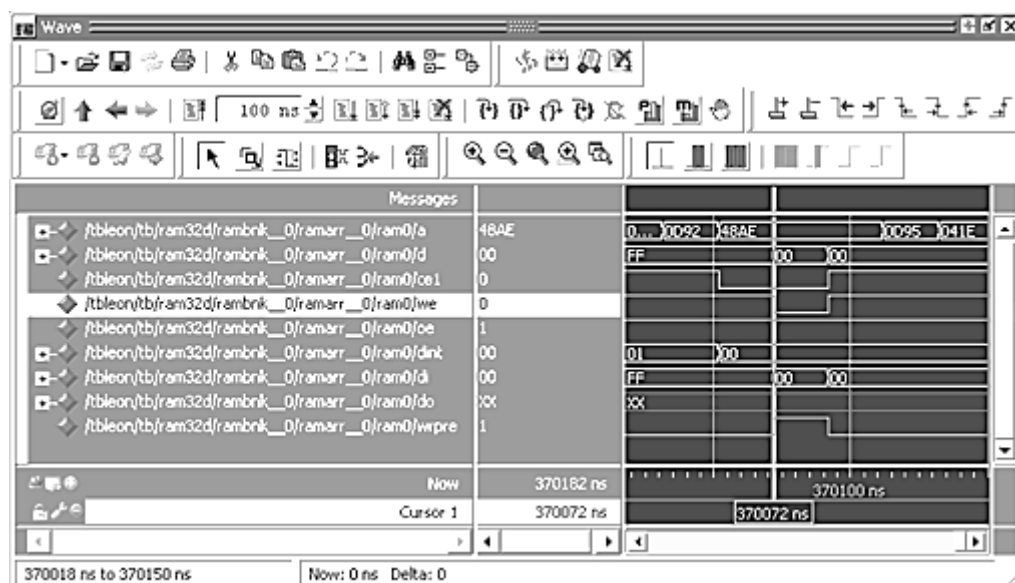
نظیر LTH یا Meiko نیز استفاده نمود. برای این کار باید LEON را با دستور make

xconfig برای این عمل تنظیم نمود.

۹- سیگنال‌هایی را که می‌خواهید حین شبیه‌سازی مشاهده کنید، به پنجره Wave اضافه کنید.

۱۰- با اجرای شبیه‌سازی مانند شکل پ-۸ می‌توانید اثر اجرای برنامه روی سیگنال‌های اضافه شده

به پنجره Wave را مشاهده کنید.



شکل پ- ۸ شمایی از سیگنال‌های اضافه شده پس از شبیه‌سازی



## ۳ پیوست ۳: اسکریپت‌ها

### ۳-۱ اسکریپت‌های صحت‌سنجی

#### ۳-۱-۱ اسکریپت gold.do

این اسکریپت، برای یک آزمون مشخص شده، نسخه Golden تولید می‌کند و اسکریپت allgolds.do برای تمام آزمون‌ها، اسکریپت gold.do را اجرا می‌کند تا نسخه مرجع برای تمام آزمون‌ها تولید شود. این اسکریپت‌ها در ضمیمه یک و دو آمده‌اند.

```
# Copy Test Bench Files to the appropriate place
# $testAddr is the address of the simulation files
# $benchAddr is the address of the test bench directory
set benchAddr $1
set testAddr $2
file delete -force $testAddr/tsource
file copy -force $benchAddr $testAddr/tsource
```

```

# Run the simulation
vsim -voptargs=+acc work.tbleon

# Add the processor's pinout signals
add wave sim:/tbleon/tb/p0/leon0/resetn
add wave sim:/tbleon/tb/p0/leon0/clk
add wave sim:/tbleon/tb/p0/leon0/pllref
add wave sim:/tbleon/tb/p0/leon0/plllock
add wave sim:/tbleon/tb/p0/leon0/errorn
add wave sim:/tbleon/tb/p0/leon0/address
add wave sim:/tbleon/tb/p0/leon0/data
add wave sim:/tbleon/tb/p0/leon0/ramsn
add wave sim:/tbleon/tb/p0/leon0/ramoen
add wave sim:/tbleon/tb/p0/leon0/rwen
add wave sim:/tbleon/tb/p0/leon0/romsn
add wave sim:/tbleon/tb/p0/leon0/iosn
add wave sim:/tbleon/tb/p0/leon0/oen
add wave sim:/tbleon/tb/p0/leon0/read
add wave sim:/tbleon/tb/p0/leon0/writen
add wave sim:/tbleon/tb/p0/leon0/brdyn
add wave sim:/tbleon/tb/p0/leon0/bexcn
add wave sim:/tbleon/tb/p0/leon0/sdcke
add wave sim:/tbleon/tb/p0/leon0/sdcsn
add wave sim:/tbleon/tb/p0/leon0/sdwen
add wave sim:/tbleon/tb/p0/leon0/sdrasn
add wave sim:/tbleon/tb/p0/leon0/sdcasn
add wave sim:/tbleon/tb/p0/leon0/sddqm
add wave sim:/tbleon/tb/p0/leon0/sdclk
add wave sim:/tbleon/tb/p0/leon0/sa
add wave sim:/tbleon/tb/p0/leon0/sd
add wave sim:/tbleon/tb/p0/leon0/pio
add wave sim:/tbleon/tb/p0/leon0/wdogn
add wave sim:/tbleon/tb/p0/leon0/dsuen
add wave sim:/tbleon/tb/p0/leon0/dsutx
add wave sim:/tbleon/tb/p0/leon0/dsurx
add wave sim:/tbleon/tb/p0/leon0/dsubre
add wave sim:/tbleon/tb/p0/leon0/dsuact
add wave sim:/tbleon/tb/p0/leon0/test
#add wave -r /*

# To continue script execution
onbreak {
resume
}

# First run
run -all

# Set onbreak to its default value
onbreak ""

# Save dataset as gold for future reuse
dataset save sim $benchAddr/gold.wlf;

echo Golden Version Made!

#Goodbye!

```



```
quit -sim;
```

## ۳-۱-۲ اسکریپت allgolds.do

```
set testadr "C:/HW/TestBenches"
set goldadr "C:/HW/HDS/Gold"
set scriptadr "C:/Documents and Settings/LIP/Desktop"

# Compile everything
project calculateorder

do $scriptadr/gold.do "$testadr/basic math" $goldadr
do $scriptadr/gold.do "$testadr/bitcount" $goldadr
do $scriptadr/gold.do "$testadr/bubble" $goldadr
do $scriptadr/gold.do "$testadr/matrix" $goldadr
do $scriptadr/gold.do "$testadr/qtsource" $goldadr
do $scriptadr/gold.do "$testadr/queue" $goldadr
do $scriptadr/gold.do "$testadr/original" $goldadr
```

## ۳-۱-۳ اسکریپت check.do

اسکریپت check.do نام پروژه، آدرس محل پروژه‌ها، نام آزمون و آدرس محل آزمون‌ها را دریافت و آزمون مربوطه را بر روی پروژه مشخص شده اجرا می‌کند. نتایج مقایسه در فایلی با پسوند compdiffs.sav برای هر یک از آزمون‌ها ذخیره می‌شود و در صورت نبود هیچ تفاوتی بین سیگنال‌ها، این فایل نیز به وجود نمی‌آید. بنابراین حالت درست زمانی است که پس از اجرا هیچ فایلی با پسوند compdiff.sav وجود نداشته باشد.

```
#Copy Test Bench Files to the appropriate place
#$testAddr is the address of the simulation files
#$benchAddr is the address of the test bench directory

set benchRoot $1
set testRoot $2

set testsAddr $3

set benchName $4
set testName $5

#The address in which results should be put
set destAddr $benchRoot
```

```

#Copy test bench specified on command line
file delete -force $testRoot/tsource
file copy -force $benchRoot/$benchName $testRoot/tsource

#Run the simulation

#vsim -coverage work.tbleon -voptargs="+acc +cover=bcesfx"
vsim work.tbleon -voptargs="+acc"

#add processor's pinout signals
add wave sim:/tbleon/tb/p0/leon0/resetn
add wave sim:/tbleon/tb/p0/leon0/clk
add wave sim:/tbleon/tb/p0/leon0/pllref
add wave sim:/tbleon/tb/p0/leon0/plllock
add wave sim:/tbleon/tb/p0/leon0/errorrn
add wave sim:/tbleon/tb/p0/leon0/address
add wave sim:/tbleon/tb/p0/leon0/data
add wave sim:/tbleon/tb/p0/leon0/ramsn
add wave sim:/tbleon/tb/p0/leon0/ramoen
add wave sim:/tbleon/tb/p0/leon0/rwen
add wave sim:/tbleon/tb/p0/leon0/romsn
add wave sim:/tbleon/tb/p0/leon0/iosn
add wave sim:/tbleon/tb/p0/leon0/oen
add wave sim:/tbleon/tb/p0/leon0/read
add wave sim:/tbleon/tb/p0/leon0/writen
add wave sim:/tbleon/tb/p0/leon0/brdyn
add wave sim:/tbleon/tb/p0/leon0/bexcn
add wave sim:/tbleon/tb/p0/leon0/sdcke
add wave sim:/tbleon/tb/p0/leon0/sdcsn
add wave sim:/tbleon/tb/p0/leon0/sdwen
add wave sim:/tbleon/tb/p0/leon0/sdrasn
add wave sim:/tbleon/tb/p0/leon0/sdcasn
add wave sim:/tbleon/tb/p0/leon0/sddqm
add wave sim:/tbleon/tb/p0/leon0/sdclk
add wave sim:/tbleon/tb/p0/leon0/sa
add wave sim:/tbleon/tb/p0/leon0/sd
add wave sim:/tbleon/tb/p0/leon0/pio
add wave sim:/tbleon/tb/p0/leon0/wdogrn
add wave sim:/tbleon/tb/p0/leon0/dsuen
add wave sim:/tbleon/tb/p0/leon0/dsutx
add wave sim:/tbleon/tb/p0/leon0/dsurx
add wave sim:/tbleon/tb/p0/leon0/dsubre
add wave sim:/tbleon/tb/p0/leon0/dsuact
add wave sim:/tbleon/tb/p0/leon0/test

# To continue script execution
onbreak {
resume
}

# First run
run -all

# Set onbreak to its default value
onbreak ""

```

```

# Saving coverage reports
file mkdir $destAddr/RESULTS/$testName/$testName-$benchName

#coverage save $destAddr/RESULTS/$testName/$testName-$
$benchName/cover.sav
#coverage save -instance /tbleon/tb/p0/leon0/mcore0/proc0/iu0 -code
bcefst -instance /tbleon/tb/p0/leon0/mcore0/uart2 -code bcefst -instance
/tbleon/tb/p0/leon0/mcore0/uart1 -code bcefst -instance
/tbleon/tb/p0/leon0/mcore0/proc0/wd0 -code bcefst
$destAddr/RESULTS/$testName/$testName-$benchName/cover.sav
#coverage open $destAddr/RESULTS/$testName/$testName-$
$benchName/cover.sav cover

#coverage report -html -htmldir $destAddr/RESULTS/$testName/$testName-$
$benchName/coverreport

#coverage save $destAddr/RESULTS/$testName-$benchName/cover.sav
#dataset close cover

# Open dataset as gold for comparison
dataset open $benchRoot/$benchName/gold.wlf
echo Golden Version Opened for comparison!

# Starting Compare
compare start -maxsignal 10000000 -maxtotal 10000000 gold sim

# add signals for comparison
compare add gold:/tbleon/tb/p0/leon0/resetn
compare add gold:/tbleon/tb/p0/leon0/clk
compare add gold:/tbleon/tb/p0/leon0/pllref
compare add gold:/tbleon/tb/p0/leon0/plllock
compare add gold:/tbleon/tb/p0/leon0/errorn
compare add gold:/tbleon/tb/p0/leon0/address
compare add gold:/tbleon/tb/p0/leon0/data
compare add gold:/tbleon/tb/p0/leon0/ramsn
compare add gold:/tbleon/tb/p0/leon0/ramoen
compare add gold:/tbleon/tb/p0/leon0/rwen
compare add gold:/tbleon/tb/p0/leon0/romsn
compare add gold:/tbleon/tb/p0/leon0/iosn
compare add gold:/tbleon/tb/p0/leon0/oen
compare add gold:/tbleon/tb/p0/leon0/read
compare add gold:/tbleon/tb/p0/leon0/writen
compare add gold:/tbleon/tb/p0/leon0/brdyn
compare add gold:/tbleon/tb/p0/leon0/bexcn
compare add gold:/tbleon/tb/p0/leon0/sdcke
compare add gold:/tbleon/tb/p0/leon0/sdcsn
compare add gold:/tbleon/tb/p0/leon0/sdwen
compare add gold:/tbleon/tb/p0/leon0/sdrasn
compare add gold:/tbleon/tb/p0/leon0/sdcasn
compare add gold:/tbleon/tb/p0/leon0/sddqm
compare add gold:/tbleon/tb/p0/leon0/sdclk
compare add gold:/tbleon/tb/p0/leon0/sa
compare add gold:/tbleon/tb/p0/leon0/sd
compare add gold:/tbleon/tb/p0/leon0/pio
compare add gold:/tbleon/tb/p0/leon0/wdogn
compare add gold:/tbleon/tb/p0/leon0/dsuen
compare add gold:/tbleon/tb/p0/leon0/dsutx

```

```

compare add gold:/tb/leon/tb/p0/leon0/dsurx
compare add gold:/tb/leon/tb/p0/leon0/dsubre
compare add gold:/tb/leon/tb/p0/leon0/dsuact
compare add gold:/tb/leon/tb/p0/leon0/test

# Running compare
compare run

# Saving comparison and differences
compare saverules $destAddr/RESULTS/$testName/$testName-
$benchName/comprules.sav

onerror {
echo Hurray! No Differences Found!!;
resume
}

compare savediffs $destAddr/RESULTS/$testName/$testName-
$benchName/compdiffs.sav
echo Comparison complete!

#Goodbye!

#echo Check Your Results!
quit -sim;
dataset close gold

```

#### ۳-۱-۴ اسکریپت testall.do

اسکریپت testall.do در هر یک از مراحل به تدریج اضافه شده است و اکنون شامل کد لازم برای آزمون هریک از قطعات یا مراحل مجتمع سازی می باشد. این آزمون اسکریپت check.do را با پارامترهای مناسب برای هریک از برنامه های آزمون برای پروژه تحت آزمون فراخوانی می کند.

```

# Compile all
do lcompile.do

# Call each test

#set projectName "iu"
#set projectName "uart"
#set projectName "regbank"
#set projectName "alu"
#set projectName "regalu"
#set projectName "watchdog"
#set projectName "iu+uart"
#set projectName "iu+uart+reg"
#set projectName "iu+uart+reg+alu"
#set projectName "iu+uart+reg+alu+wd"

# Test projectName

```

```
do check.do "C:/HW/TestBenches" "C:/HW/HDS/Gold" "C:/HW/HDS" "basic
math" $projectName
do check.do "C:/HW/TestBenches" "C:/HW/HDS/Gold" "C:/HW/HDS" "bitcount"
$projectName
do check.do "C:/HW/TestBenches" "C:/HW/HDS/Gold" "C:/HW/HDS" "bubble"
$projectName
do check.do "C:/HW/TestBenches" "C:/HW/HDS/Gold" "C:/HW/HDS" "matrix"
$projectName
do check.do "C:/HW/TestBenches" "C:/HW/HDS/Gold" "C:/HW/HDS" "original"
$projectName
do check.do "C:/HW/TestBenches" "C:/HW/HDS/Gold" "C:/HW/HDS" "qtsource"
$projectName
do check.do "C:/HW/TestBenches" "C:/HW/HDS/Gold" "C:/HW/HDS" "queue"
$projectName
```

## ۳-۲ اسکریپت lcompile.do

این اسکریپت، فایل‌های توصیف برنامه به زبان VHDL را به ترتیب کامپایل می‌کند.

```
vlib work
```

```
#Compile the FPU files
vcom -quiet fpu/fpupack.vhd
vcom -quiet fpu/single_pre.vhd
vcom -quiet fpu/single_post.vhd
vcom -quiet fpu/pre_norm_addsub.vhd
vcom -quiet fpu/addsub_28.vhd
vcom -quiet fpu/post_norm_addsub.vhd
vcom -quiet fpu/pre_norm_mul.vhd
vcom -quiet fpu/serial_mul.vhd
vcom -quiet fpu/post_norm_mul.vhd
vcom -quiet fpu/pre_norm_div.vhd
vcom -quiet fpu/serial_div.vhd
vcom -quiet fpu/post_norm_div.vhd
vcom -quiet fpu/pre_norm_sqrt.vhd
vcom -quiet fpu/sqrt.vhd
vcom -quiet fpu/post_norm_sqrt.vhd
vcom -quiet fpu/comppack.vhd
vcom -quiet fpu/convert.vhd
vcom -quiet fpu/cmp.vhd
vcom -quiet fpu/fpu.vhd
```

```
#Compiling the unisim library
#vcom -quiet unisim/unisim_VPKG.vhd
#vcom -quiet unisim/unisim_VCOMP.vhd
#vcom -quiet unisim/unisim_VITAL.vhd
```

```
#Compiling leon processor codes
vcom -quiet leon/amba.vhd
vcom -quiet leon/target.vhd
vcom -quiet leon/device.vhd
vcom -quiet leon/config.vhd
```

```

vcom -quiet leon/mmuconfig.vhd
vcom -quiet leon/sparcv8.vhd
vcom -quiet leon/iface.vhd
vcom -quiet leon/macro.vhd
vcom -quiet leon/debug.vhd
vcom -quiet leon/ambacomp.vhd
vcom -quiet leon/multlib.vhd
vcom -quiet leon/tech_generic.vhd
vcom -quiet leon/tech_proasic.vhd
vcom -quiet leon/tech_axcel.vhd
vcom -quiet leon/tech_atc18.vhd
vcom -quiet leon/tech_tsmc25.vhd
vcom -quiet leon/tech_atc35.vhd
vcom -quiet leon/tech_atc25.vhd
vcom -quiet leon/tech_atc18.vhd
vcom -quiet leon/tech_umc18.vhd
vcom -quiet leon/tech_fs90.vhd
vcom -quiet leon/bprom.vhd
vcom -quiet leon/tech_virtex.vhd
vcom -quiet leon/tech_virtex2.vhd
vcom -quiet leon/tech_map.vhd
vcom -quiet leon/fpulib.vhd
vcom -quiet leon/meiko.vhd
vcom -quiet leon/fpu_lth.vhd
vcom -quiet leon/fpu_core.vhd
vcom -quiet leon/grfpc.vhd
vcom -quiet leon/fpleu.vhd
vcom -quiet leon/mmu_icache.vhd
vcom -quiet leon/mmu_dcache.vhd
vcom -quiet leon/mmu_acache.vhd
vcom -quiet leon/mmutlbcam.vhd
vcom -quiet leon/mmulrue.vhd
vcom -quiet leon/mmulru.vhd
vcom -quiet leon/mmutlb.vhd
vcom -quiet leon/mmutw.vhd
vcom -quiet leon/mmu.vhd
vcom -quiet leon/mmu_cache.vhd

###Compiling wd.vhd for adding the watchdog part.
#vcom -quiet leon/wd.vhd

vcom -quiet leon/mul.vhd
vcom -quiet leon/div.vhd
vcom -quiet leon/rstgen.vhd
vcom -quiet leon/iu.vhd
vcom -quiet leon/icache.vhd
vcom -quiet leon/dcache.vhd
vcom -quiet leon/cachemem.vhd
vcom -quiet leon/acache.vhd
vcom -quiet leon/cache.vhd
vcom -quiet leon/proc.vhd
vcom -quiet leon/irqctrl2.vhd
vcom -quiet leon/apbmst.vhd
vcom -quiet leon/ahbarb.vhd
vcom -quiet leon/ahbram.vhd
vcom -quiet leon/lconf.vhd
vcom -quiet leon/wprot.vhd
vcom -quiet leon/ahbtest.vhd
vcom -quiet leon/ahbstat.vhd
vcom -quiet leon/timers.vhd

```

```

vcom -quiet leon/irqctrl.vhd
vcom -quiet leon/uart.vhd
vcom -quiet leon/ioport.vhd
vcom -quiet leon/sdmctrl.vhd
vcom -quiet leon/mctrl.vhd
vcom -quiet leon/ahbmst.vhd
vcom -quiet leon/dcom_uart.vhd
vcom -quiet leon/dcom.vhd
vcom -quiet leon/dma.vhd
vcom -quiet leon/dsu.vhd
vcom -quiet leon/dsu_mem.vhd
vlog -quiet +incdir+leon leon/ethermac.v
vcom -quiet leon/pci_arb.vhd
vcom -quiet leon/pci_gr.vhd
vcom -quiet leon/pci.vhd
vcom -quiet leon/eth_oc.vhd
vcom -quiet leon/mcore.vhd
vcom -quiet leon/leon.vhd
vcom -quiet leon/leon_pci.vhd
vcom -quiet leon/leon_eth_pci.vhd
vcom -quiet leon/leon_eth.vhd

```

```

#Compiling the testbench codes
vcom -quiet tbench/leonlib.vhd
vcom -quiet tbench/iram.vhd
vcom -quiet tbench/mt48lc16m16a2.vhd
vcom -quiet tbench/testmod.vhd
vcom -quiet tbench/mspram.vhd
vcom -quiet tbench/bprom.vhd
vcom -quiet tbench/tbgen.vhd
vcom -quiet tbench/tbllib.vhd
vcom -quiet tbench/tbdef.vhd
vcom -quiet tbench/tbleon.vhd
vcom -quiet tbench/tb_msp.vhd

```

### ۳-۳ اسکرپت تزریق اشکال

این اسکرپت ابتدا شبیه‌سازی را به مدت ۳۰۰us می‌کند. سپس در در زمان مشخص، (در اینجا

۱۱۵۶۹۵۰۰ns) روی سیگنال r.X(54) اشکال تزریق می‌کند. سپس به اندازه عرض یک کلاک اشکال را

پایدار نگه می‌دارد (تا نوع خطا SEU باشد) و سپس اجرا را تا انتها ادامه می‌دهد. در نهایت مقدار

سیگنال‌ها و محتوای حافظه را در فایل ذخیره می‌کند.

```

#SEU Injection in Main Core Testbeench Qsort_Large from MiBench
#Fault Injected in /tbleon/tb/p0/leon0/mcore0/proc0/iu0/dgen/div0/r.X(54)

```

```

transcript file FI/transcript
vsim work.tbleon
onerror {resume}

```

```
set StdArithNoWarnings 1

set NumericStdNoWarnings 1

run @300000 ns
transcript      file FI/FI_Results/FI_1000_Result.txt

run @11569500 ns
variable old_value [examine
/tbleon/tb/p0/leon0/mcore0/proc0/iu0/dgen/div0/r.X(54)]
if { $old_value == "0" } {
force -freeze /tbleon/tb/p0/leon0/mcore0/proc0/iu0/dgen/div0/r.X(54) 1
} else {
force -freeze /tbleon/tb/p0/leon0/mcore0/proc0/iu0/dgen/div0/r.X(54) 0
}

run 20 ns
noforce /tbleon/tb/p0/leon0/mcore0/proc0/iu0/dgen/div0/r.X(54)

run @20500000 ns

transcript file FI/transcript
add list -r sim:/tbleon/tb/p0/leon0/*
write list -events FI/FI_Results/FI_1000_list_event.txt

#Saving RAMs
mem save -o FI/FI_Results/FI_1000_ram00.mem -f mti -data hex -addr hex
/tbleon/tb/ram32d/rambnk__0/ramarr__0/ram0/ram/mema
mem save -o FI/FI_Results/FI_1000_ram01.mem -f mti -data hex -addr hex
/tbleon/tb/ram32d/rambnk__0/ramarr__1/ram0/ram/mema
mem save -o FI/FI_Results/FI_1000_ram02.mem -f mti -data hex -addr hex
/tbleon/tb/ram32d/rambnk__0/ramarr__2/ram0/ram/mema
mem save -o FI/FI_Results/FI_1000_ram03.mem -f mti -data hex -addr hex
/tbleon/tb/ram32d/rambnk__0/ramarr__3/ram0/ram/mema

quit -sim
```



## واژه‌نامه‌ی انگلیسی به فارسی

Adapter	مبدل
Arithmetic Codes	کدهای حسابی
Arithmetic Logic Unit (ALU)	واحد محاسبه و منطق
Behavioural Model	مدل رفتاری
Benchmark	برنامه‌ی محک
Carry Look-ahead Adder	جمع‌کننده‌ی پیشگوی رقم نقلی
Carry Select Adder	جمع‌کننده با انتخاب رقم نقلی
Computer-Aided Design (CAD)	طراحی به کمک کامپیوتر
Context	زمینه
Cycle	چرخه
Design Phase	طراحی
Duplication	دوتائی
Embedded Systems	سیستم نهفته
Error	خطا
Error Correction Codes	کدهای تصحیح خطا
Error Detection and Correction (EDAC)	آزمون تشخیص و تصحیح خطا
External Disturbance	آشفتگی در دنیای خارج
Fault	اشکال
Fault-tolerancy	تحمل‌پذیری اشکال
Force	قفل کردن
Global Register	ثبات همگانی
Golden Version	نسخه‌ی مرجع
Hardware Reuse	استفاده مجدد از سخت‌افزار
Input Register	ثباتی ورودی
Instruction-Set Architecture (ISA)	معماری دستورالعمل
Integrated Circuits (IC)	مدار مجتمع
Integration	یکپارچه‌سازی
Iterative Program	برنامه‌ی حلقه‌ای
Local Register	ثبات محلی
Logging	ثبت

Low Cost Error Detection	کدگذاری کم‌هزینه برای تشخیص خطا
Markov Model	مدل‌های مارکوف
Mean Time Between Failures (MTBF)	زمان متوسط بین دو خرابی
Mixed Simulation	شبیه‌سازی آمیخته
Narrow-width Values	داده‌های کم‌عرض
Non-Series-Parallel Systems	سیستم‌های غیرسری-موازی
N-Version Programming	برنامه‌سازی N نسخه‌ای
Open Architecture	معماری باز
Open Source	منبع باز
Operand	عملوند
Operational Phase	عملیاتی
Parallel Systems	سیستم‌های موازی
Parity	زوجیت
Poisson Model	مدل‌های پواسون
Procedure	پروسه
Processor State	حالت ریزپردازنده
Prototype Phase	نمونه‌سازی
Re-computation	محاسبه مجدد
Redundancy	افزونگی
Register Windows	پنجره‌های ثابت
Safety	ایمنی
Scan Chain	زنجیره پیمایش
Series Systems	سیستم‌های سری
Simulated Fault Injection	تزریق اشکال شبیه‌سازی شده
Software Implemented Fault Injection (SWIFI)	تزریق اشکال نرم‌افزاری پیاده‌سازی شده
States	حالت‌ها
Trace Buffer	بافر ردیابی
Triple-Modular Redundancy	افزونگی سه‌پیمانه‌ای
Verification	صحت‌سنجی

## **Abstract**

Embedded systems are used in most of the equipment that are part of our modern life. The importance and pervasiveness of these systems are so much that nearly 99 percent of all the processors produced annually are embedded processors. Due to the criticality of the applications in which embedded systems are used, reliability is one of the most important requirements for most embedded systems. In this thesis, three phases of the design process, namely integration, verification and evaluation of reliability, are performed in order to achieve an embedded fault-tolerant microprocessor. In the integration phase, previously designed fault-tolerant units of a microprocessor (Arithmetic-Logic Unit, Control Unit, Register Bank, and Peripherals) are integrated to form a fault-tolerant microprocessor based on the SPARC v.8 architecture for safety-critical applications. An IEEE-754 compliant floating point unit is also designed and implemented, and techniques have been devised for its fault-tolerance. To verify the correctness of the microprocessor's functionality, standard benchmark programs from the MiBench suit have been used. Finally, the capabilities of the fault-tolerance techniques used in the microprocessor are evaluated using conventional fault injection techniques, such as SEU-based fault injection.

**Keywords:** Fault-tolerance, Fault Injection, Single-Event Upset (SEU), Verification, Integration, Embedded Systems



**Sharif University of Technology**  
**Department of Computer Engineering**

**B.Sc. Thesis**

**Integration, Verification, and Evaluation of an  
Embedded Fault-Tolerant Microprocessor**

**By:**

**Mohammad Javad Dousti**

**Pooria Joulani**

**Supervisor:**

**Seyed-Ghassem Miremadi**

**June 2010**