



دانشگاه تهران  
پردیس دانشکده‌های فنی  
دانشکده مهندسی برق و کامپیوتر

## طراحی افزونه و مُد Debugger برای پردازنده RISC-V

پایان‌نامه برای دریافت درجه کارشناسی  
در رشته مهندسی برق گرایش سیستم‌های دیجیتال

نام  
محمد تقی زاده گیوری

شماره دانشجویی  
۸۱۰۱۹۸۳۷۳

استاد راهنما:  
دکتر زین العابدین نوابی شیرازی

تیرماه ۱۴۰۳

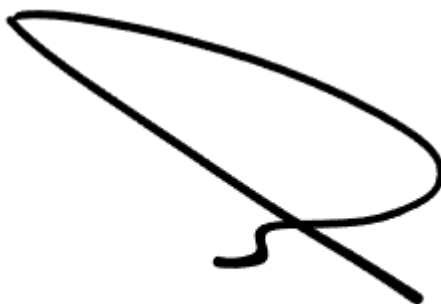
بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

تعهدنامه اصالت اثر  
باسمه تعالی

اینجانب محمد تقی زاده گیوری تأیید می کنم که مطالب مندرج در این پایان نامه حاصل تلاش اینجانب است و به دستاوردهای پژوهشی دیگران که در این نوشته از آنها استفاده شده است مطابق مقررات ارجاع گردیده است. این پایان نامه قبلاً برای احراز هیچ مدرک هم سطح یا بالاتر ارائه نشده است. کلیه حقوق مادی و معنوی این اثر متعلق به دانشکده فنی دانشگاه تهران می باشد.

نام و نام خانوادگی دانشجو :  
محمد تقی زاده گیوری

امضای دانشجو :



## تشکر و قدردانی

با سپاس فراوان از استاد ارجمند جناب آقای دکتر نوابی، استاد راهنمای این پژوهش و خانم مریم رجبعلی پناه و خانم زهرا جهان پیمما، دانشجویان دکتری دانشگاه تهران که از راهنمایی های مفید ایشان بهره مند گردیدم.

با سپاس فراوان از استاد ارجمند جناب آقای دکتر علیزاده، که زحمات نقد و داوری این پژوهش را با نهایت لطف پذیرفتند.

## چکیده<sup>۱</sup>

یکی از بزرگ ترین مشکلاتی که برنامه نویسان هنگام برنامه نویسی با آن مواجه می شوند، پیدا کردن علتِ نادرست بودن خروجی برنامه است. به عبارتی، زمانی که خروجی کد نوشته شده، اشتباه است، دلایل متعددی می تواند باعث خروجی نادرست شود که پیدا کردن علت نادرست بودن خروجی، باتوجه به تعداد زیاد فرضیات ممکن، دشوار است. برای تسهیل این امر، نیاز به ابزاری است که روند اجرای برنامه را به برنامه نویسان نشان دهد و از طریق آن بتوان اجرا شدن تک تک خطوط کد و تاثیری که هر کدام بر مقادیر متغیرها و... می گذارد را مشاهده کرد تا از این طریق، سریع تر بتوان به علتِ عدم کارکرد مناسب برنامه پی برد.

Debugger ابزاری است که به فرآیند پیدا کردن علت خرابی کد (Debugging) کمک می کند. هدف از این پروژه، ایجاد قابلیتِ مشاهدهٔ روند اجرا برنامه در پردازنده می باشد تا با ایجاد این امکان، قابلیت debug کردن و مشاهدهٔ مقادیر ذخیره شده در پردازنده و حافظه در هر قسمت از برنامه ای که در پردازنده اجرا می شود، وجود داشته باشد. در این صورت تشخیص منشأ خطا در برنامه، تسهیل و تسريع می گردد.

برای رسیدن به این هدف و پیاده سازی آن، ابتدا با ساختار و نحوه کارکرد Debugger آشنا می شویم. سپس به طراحی RTL<sup>۲</sup> و توصیف سخت افزاری آن در محیط Verilog پرداخته و تغییراتی نیز در ساختار پردازنده RISC-V اعمال می کنیم. در نهایت به آزمودن Debugger در حضور پردازنده که با یک برنامه خاص برنامه نویسی شده است، می پردازیم. خواهیم دید که Debugger طراحی شده قادر است اجرا برنامه بر روی پردازنده را متوقف کند، مقادیر ذخیره شده در register file پردازنده و هم چنین حافظه خارجی را بخواند و در صورت نیاز، مقداری در آنان ذخیره کند و پس از آن، اجرا برنامه را از جایی که متوقف شده بود، ادامه دهد.

**کلمات کلیدی: register file, RISC-V, Verilog, RTL, debugging**

---

<sup>1</sup> Abstract

<sup>2</sup> Register Transfer Level

## فهرست مطالب

فصل ۱: مقدمه و بیان مساله .....	۱
۱-۱- مقدمه .....	۲
۲-۱- تاریخچه‌ای از موضوع تحقیق .....	۲
۳-۱- شرح مسئله تحقیق .....	۳
۴-۱- تعریف موضوع تحقیق .....	۳
۵-۱- اهداف و آرمان‌های کلی تحقیق .....	۳
۶-۱- روش انجام تحقیق .....	۴
۷-۱- ساختار پایان‌نامه .....	۴
فصل ۲: مفاهیم اولیه و پیش زمینه پروژه .....	۵
۱-۲- مقدمه .....	۶
۲-۲- بخش اول : پردازنده RISC-V .....	۶
۱-۲-۲- مقدمه‌ای بر پردازنده .....	۶
۲-۲-۲- مقدمه‌ای بر RISC-V .....	۷
۳-۲-۲- ساختار پردازنده RISC-V .....	۸
۳-۲- بخش دوم : Debugger .....	۱۶
۱-۳-۲- مقدمه‌ای بر Debugger .....	۱۶
۲-۳-۲- تعامل Debugger با پردازنده RISC-V .....	۱۷
۴-۲- خلاصه و جمع بندی .....	۱۸
فصل ۳: طراحی Debugger .....	۱۹
۱-۳- مقدمه .....	۲۰

۲۱	۲-۳- روش پیشنهادی برای طراحی Debugger
۲۵	۱-۲-۳- ساختار Debugger
۲۹	۲-۲-۳- پاسخ دادن پردازنده به درخواست Debugger
۴۵	۳-۳- ابزارهای مورد نیاز برای طراحی Debugger
۴۵	۴-۳- معیار ارزیابی
۴۷	۵-۳- نتایج بدست آمده از طراحی Debugger
۵۲	۶-۳- تحلیل نتایج
۵۲	۷-۳- خلاصه و جمع‌بندی
۵۳	فصل ۴: پیاده سازی Debugger
۵۴	۱-۴- مقدمه
۵۴	۲-۴- نحوه پیاده سازی
۵۹	۳-۴- خلاصه و جمع‌بندی
۶۰	فصل ۵: جمع‌بندی و نتیجه‌گیری
۶۱	۱-۵- جمع‌بندی
۶۱	۲-۵- نتیجه‌گیری
۶۲	فصل ۶: مراجع

## فهرست شکل‌ها

شکل (۲-۲) ثبات در پردازنده AFTAB	۱۰
شکل (۳-۲) جمع کننده PC در پردازنده AFTAB	۱۰
شکل (۴-۲) ساختار دستور در پردازنده AFTAB	۱۱
شکل (۵-۲) ماژول DARU در پردازنده AFTAB	۱۲
شکل (۶-۲) ماژول DAWU در پردازنده AFTAB	۱۳
شکل (۷-۲) ماژول Comparator در پردازنده AFTAB	۱۴
شکل (۸-۲) ماژول Adder/Subtractor Unit (ASU) در پردازنده AFTAB	۱۴
شکل (۹-۲) ماژول Logical Logic Unit (LLU) در پردازنده AFTAB	۱۴
شکل (۱۰-۲) ماژول Barrel Shifter Unit (BSU) در پردازنده AFTAB	۱۵
شکل (۱۱-۲) ماژول Attached Arithmetic Unit (AAU) در پردازنده AFTAB	۱۵
شکل (۱۲-۲) سیستم Debug پردازنده RISC-V (به صورت کلی)	۱۷
شکل (۱۳-۳) سیستم Debug پردازنده RISC-V (همراه با جزئیات کامل)	۲۱
شکل (۱۴-۳) مسیر داده Debug Module	۲۶
شکل (۱۵-۳) کنترل کننده Debug Module	۲۷
شکل (۱۶-۳) ماژول Debugger Command Decoder	۳۰
شکل (۱۷-۳) ساختار درخواست Debug Module	۳۱
شکل (۱۸-۳) تغییرات data path پردازنده جهت خواندن از register file توسط Debug Module	۳۲
شکل (۱۹-۳) تغییرات کنترلر پردازنده جهت پاسخ به درخواست های Debug Module	۳۳
شکل (۲۰-۳) تغییرات data path پردازنده جهت نوشتن به register file توسط Debug Module	۳۶
شکل (۲۱-۳) تغییرات data path پردازنده جهت خواندن از حافظه خارجی توسط Debug Module	۳۹
شکل (۲۲-۳) تغییرات data path پردازنده جهت نوشتن به حافظه خارجی توسط Debug Module	۴۲
شکل (۲۳-۳) متوقف شدن اجرا برنامه با ارسال درخواست core_halt_request به Debugger	۴۷
شکل (۲۴-۳) خواندن از register file با ارسال درخواست خواندن از register file به Debugger	۴۸
شکل (۲۵-۳) نوشتن به register file با ارسال درخواست نوشتن به register file به Debugger	۴۹
شکل (۲۶-۳) خواندن از حافظه خارجی با ارسال درخواست خواندن از حافظه خارجی به Debugger	۵۰
شکل (۲۷-۳) نوشتن به حافظه خارجی با ارسال درخواست نوشتن به حافظه خارجی به Debugger	۵۱
شکل (۲۸-۴) پیاده سازی Data Path ماژول Debug Module در Verilog	۵۵
شکل (۲۹-۴) پیاده سازی کنترلر ماژول Debug Module در Verilog	۵۶
شکل (۳۰-۴) پیاده سازی ماژول Debugger Command Decoder در Verilog	۵۶
شکل (۳۱-۴) تغییرات اعمالی به Data Path پردازنده RISC-V در Verilog	۵۷
شکل (۳۲-۴) تغییرات اعمالی به کنترلر پردازنده RISC-V در Verilog	۵۸



## فهرست جدول‌ها

## فهرست علائم اختصاری

DDT	Dynamic Debugging Tool
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
PC	Program Counter
opcode	operation code
DARU	Data Adjustment Read Unit
DAWU	Data Adjustment Write Unit
ASU	Adder/Subtractor Unit
LLU	Logical Logic Unit
BSU	Barrel Shifter Unit
AAU	Attached Arithmetic Unit
DTM	Debug Transport Module
DMI	Debug Module Interface
DM	Debug Module

# فصل ۱

## مقدمه و بیان مساله

---

در این فصل نخست به بیان مقدمات کار و تاریخچه‌ای کوتاه از مساله تحقیق پرداخته، سپس مساله و موضوع مورد بررسی در این پایان‌نامه، اهداف و روش کلی تحقیق را بیان می‌کنیم و در نهایت به ساختار پایان‌نامه‌ی پیش رو اشاره خواهیم کرد.

## ۱-۱- مقدمه

با پیشرفت فن آوری در طی دهه های گذشته، به تدریج سطح انتظارات مردم از صنعت بالا رفته است. این سطح از انتظارات سبب شده است تا در میان تولید کنندگان رقابت برای برآورده کردن نیاز های مردم شدت گیرد. در این میان صنعت نرم افزار نیز بیش از پیش رشد کرده و رقابت شدیدی در این حوزه شکل گرفته است. در نتیجه تولید کنندگان نرم افزار برای اینکه بتوانند با دیگر تولید کنندگان رقابت کنند، نیاز دارند تا محصول خود را سریع تر عرضه کرده و محصول با کیفیت تری تولید کنند. برای سرعت دادن به توسعه نرم افزار نیاز است تا برنامه نویسان بتوانند سریع تر عیب برنامه ای که نوشتند را تشخیص دهند تا زمان کمتری صرف عیب یابی شده و در نتیجه سرعت رشد محصول بالا رود. بنابراین ایجاد بستری که بتواند به عیب یابی برنامه کمک کند، به شدت مورد توجه است.

## ۱-۲- تاریخچه ای از موضوع تحقیق

تاریخچه دیباگرها به روزهای اولیه کامپیوترها باز میگردد. اولین دیباگرها ابزارهای ابتدایی بودند که به برنامه نویسان اجازه می دادند محتویات حافظه و رجیسترها را بررسی کنند. با پیچیده تر شدن کامپیوترها، دیباگرها برای ارائه ویژگی های پیچیده تر، مانند اجرای گام به گام برنامه و بازرسی متغیرها، تکامل یافتند. دیباگرهای اولیه که اغلب "ردیاب" نامیده می شوند، اغلب بخشی از سیستم عامل یا محیط زمان اجرا بودند. دهه ۱۹۶۰ شاهد ظهور ابزارهای اشکال زدایی اختصاصی مانند (Dynamic Debugging Tool) DDT (ابزار اشکال زدایی پویا) برای IBM System/360 و بعداً، اشکال زدایی نمادین بود که به برنامه نویسان اجازه می داد تا با کد منبع به جای دستورالعمل های ماشین تعامل داشته باشند. توسعه رابط های کاربری گرافیکی در دهه ۱۹۸۰ باعث ایجاد انقلابی در اشکال زدایی شد و آن را برای برنامه نویسان بصری تر و در دسترس تر کرد. امروزه، دیباگرها ابزارهای ضروری برای توسعه نرم افزار هستند و طیف وسیعی از ویژگی ها و قابلیت ها را برای کمک به برنامه نویسان برای شناسایی و رفع خطاهای کد خود ارائه می دهند.

### ۱-۳- شرح مسئله تحقیق

همان طور که پیش تر توضیح داده شد، تولیدکنندگان نرم افزار نیاز جدی دارند که سرعت تولید محصولات خود را افزایش دهند تا بتوانند با دیگر تولیدکنندگان رقابت کنند. در نتیجه نیاز به بستری برای عیب یابی برنامه است که برنامه نویسان بتوانند در آن روند اجرا برنامه ای که نوشته اند را مشاهده کنند تا سریع تر بتوانند منشأ خطا در کدی که نوشته اند را پیدا کنند و در نتیجه با کاهش مدت زمان عیب یابی، سرعت توسعه نرم افزار نیز بالا رود. بنابراین مسئله تحقیق را ایجاد بستری برای مشاهده روند اجرا برنامه در پردازنده تعریف می کنیم تا به این موضوع که از اهمیت زیادی برخوردار است پاسخ دهیم.

### ۱-۴- تعریف موضوع تحقیق

در این تحقیق، قصد داریم به مسئله تحقیق که در بخش قبل گفته شد، پرداخته و بستری فراهم کنیم که از طریق آن بتوان روند اجرا برنامه را مشاهده کرده و در نتیجه عیب یابی برنامه تسریع یابد. بنابراین در این تحقیق به این موضوع می پردازیم که این بستر که Debugger نام دارد را برای پردازنده RISC-V طراحی و توسعه دهیم که قابلیت امکان توقف برنامه و مشاهده مقادیر ذخیره شده در پردازنده را داشته باشد تا با متوقف کردن برنامه و مشاهده مقادیر ذخیره شده در پردازنده تا به اینجا از برنامه، روند اجرا برنامه قابل مشاهده باشد.

### ۱-۵- اهداف و آرمان های کلی تحقیق

هدف از این تحقیق، ایجاد قابلیت مشاهده روند اجرا برنامه در پردازنده می باشد تا با ایجاد این امکان، قابلیت مشاهده مقادیر ذخیره شده در پردازنده در هر قسمت از برنامه ای که در پردازنده اجرا می شود، وجود داشته باشد تا در این صورت قابلیت تشخیص منشأ خطا در برنامه تسهیل گردد.

## ۱-۶- روش انجام تحقیق

برای انجام این تحقیق:

- ابتدا به طراحی Debugger می پردازیم.
- سپس در ساختار پردازنده RISC-V تغییراتی که لازم است ایجاد می کنیم.
- در ادامه Debugger طراحی شده را در یک زبان توصیف سخت افزار پیاده سازی می کنیم.
- در آخر Debugger پیاده سازی شده را در حضور پردازنده تغییر یافته تست کرده و می آزمایشیم تا مطمئن شویم که Debugger طراحی شده به درستی کار می کند.

## ۱-۷- ساختار پایان نامه

در فصل دوم، مقدمات، مفاهیم اولیه و پیش زمینه هایی را که جهت درک هر چه بهتر موضوع های مطرح شده در این پایان نامه مورد نیاز است، از مفاهیم مربوط به پردازنده RISC-V تا debugger ارائه می شود. در فصل سوم توضیحات مربوط به ساختار و طراحی Debugger برای پردازنده RISC-V ارائه می شود. در فصل چهارم پس از طراحی های انجام شده در فصل قبل، به جزئیات پیاده سازی Debugger در محیط Verilog می پردازیم. در نهایت، در فصل پنجم، نتیجه گیری های کلی حاصل شده در این تحقیق مورد بحث قرار می گیرد.

## فصل ۲

### مفاهیم اولیه و پیش زمینه پروژه

---

در فصل پیش رو مقدمات، مفاهیم اولیه و پیش‌زمینه‌هایی را که جهت درک هر چه بهتر موضوع‌های مطرح شده در این پایان‌نامه مورد نیاز است، از مفاهیم مربوط به پردازنده RISC-V تا debugger ارائه خواهد شد.

## ۲-۱- مقدمه

در این فصل به طور خلاصه، به مفاهیم و پیش زمینه هایی که برای درک و اجرای این پروژه الزامی هستند اشاره می کنیم. ابتدا ساختار کلی پردازنده RISC-V مورد استفاده در این پروژه را بررسی می کنیم و سپس به Debugger اشاره می کنیم. در آخر، پردازنده و Debugger را در کنار هم مورد بررسی قرار داده و نحوه کارکرد Debugger را به صورت کلی بیان می کنیم.

## ۲-۲- بخش اول : پردازنده RISC-V

## ۲-۲-۱- مقدمه ای بر پردازنده

ابتدا لازم است تا با پردازنده و نحوه کارکرد آن آشنا شویم:

برنامه هایی که بر روی کامپیوتر اجرا می شوند، عموماً نیاز به انجام یک سری عملیات بر روی مجموعه ای از داده ها دارند. برای انجام این کار، از یک مدار دیجیتالی به عنوان "پردازنده" استفاده می شود. پردازنده ابتدا عملیاتی که باید انجام دهد را از یک حافظه به صورت مجموعه ای از دستورات خوانده و متناسب با دستور خوانده شده، داده هایی را از یک حافظه خارجی خوانده و بر روی آنان عملیاتی را انجام می دهد و در آخر در صورت نیاز، نتایج را در قسمتی از آن حافظه ذخیره می کند. عملیاتی که پردازنده انجام می دهد عموماً ۳ نوع است:

## ۱. محاسباتی

این نوع دستورات، عملیات ریاضی بر روی داده ها انجام می دهند. به طور مثال دو داده را با هم جمع می کند یا دو داده را در هم ضرب می کند و نتیجه حاصل شده را در حافظه داخلی پردازنده (register file) ذخیره می کند.

## ۲. خواندن یا نوشتن به حافظه

این نوع دستورات برای تعامل با حافظه خارجی استفاده می شوند. به طور کلی پردازنده از حافظه خارجی داده می خواند و آن را در حافظه داخلی خود (register file) ذخیره می کند که این کار با دستور Load انجام می شود و نتایج حاصل از عملیات انجام شده که در حافظه داخلی پردازنده



(register file) است را در حافظه خارجی ذخیره می کند که این کار با استفاده از دستور Store صورت می گیرد.

### ۳. کنترل روند برنامه

این نوع دستورات برای کنترل روند اجرا برنامه هستند. برنامه در حالت عادی از اولین آدرس حافظه ای که دستورات در آن ذخیره شده است، شروع می شود و دستورات را یکی یکی از حافظه خوانده و اجرا می کند و تا آخرین دستور که در آخرین آدرس حافظه هست، این روند ادامه می یابد. برای تغییر روند اجرا برنامه کافیسست آدرسی که از حافظه می خوانیم را تغییر دهیم تا روند اجرا برنامه از حالت عادی تغییر کرده و ترتیب اجرا دستورات متفاوت شود. به تغییر دادن آدرس "پرش" گفته می شود، زیرا برنامه از آدرسی که هست به یک آدرس دیگر پرش می کند. حال این تغییر آدرس (پرش) یا بدون شرط خاصی انجام می شود که با دستور Jump انجام می شود، یا فقط در صورت برقرار بودن یک شرط خاص، روند برنامه باید تغییر کند که با استفاده از دستور Branch، این کار صورت می گیرد.

## ۲-۲-۲- مقدمه ای بر RISC-V

بعد از آشنایی با کلیت پردازنده، با مفهوم RISC-V آشنا می شویم:

RISC مخفف Reduced Instruction Set Computer است. در واقع پردازنده ای که RISC است، ساختار دستوراتی که پشتیبانی می کند یا Instruction set آن، ساده است و به عبارتی میزان پیچیدگی دستورات آن کاهش (Reduced) یافته است. در مقابل RISC، پردازنده های CISC هستند که Instruction set و دستوراتی که پشتیبانی می کنند پیچیده (Complex) است. به طور مثال یک پردازنده CISC، دستور sin را پشتیبانی می کند که برای محاسبه سینوس، نیاز به مدار دیجیتالی پیچیده است، در حالی که پردازنده RISC، چنین دستوری را پشتیبانی نمی کند و برنامه نویس موظف است که با استفاده از دستورات ساده ای که RISC پشتیبانی می کند، برنامه ای بنویسد که سینوس را محاسبه کند. RISC-V در واقع پنجمین نسخه (V) از پردازنده های RISC است که دارای Instruction set و مجموعه دستورات ساده ای هستند. RISC-V تنها فرمت دستورات را مشخص می کند و پیاده سازی آن را به طراح واگذار کرده است. به همین دلیل پیاده سازی های مختلفی برای این نوع پردازنده ارائه شده است که در این پروژه از پردازنده RISC-V، به نام آفتاب (AFTAB) که توسط CINI (آزمایشگاه ملی امنیت سایبری) و دانشگاه تهران طراحی شده

است، استفاده می کنیم.

## ۲-۲-۳- ساختار پردازنده RISC-V

ابتدا به ساختار کنترلی پردازنده AFTAB که همان طور که پیش تر گفتیم نوعی پیاده سازی برای پردازنده RISC-V است، می پردازیم تا با روند اجرا دستورات در پردازنده AFTAB آشنا شویم. مراحل زیر به ترتیب طی می شود تا یک دستور از مجموعه دستورات، اجرا شود:

۱. ابتدا دستور از حافظه ای که دستورات در آن ذخیره شده است، خوانده می شود. به این مرحله

getInstr می گویند که در آن، دستورات از حافظه خوانده می شود.

۲. حال دستور خوانده شده را رمزگشایی (decode) می کنیم تا از روی بخش های مختلف دستور، به

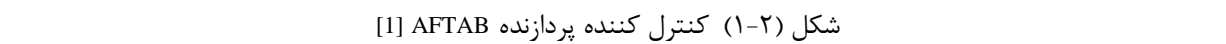
نوع دستور، oprand های آن (آرگومان هایی که دستور دارد) و... پی ببریم.

۳. حال که فهمیدیم دستور چیست و چه آرگومان هایی دارد، کافیهست که باتوجه به دستور، از یک

واحد محاسباتی برای اجرا دستور استفاده کنیم. ممکن است اجرا برخی از دستورات بیش از چند

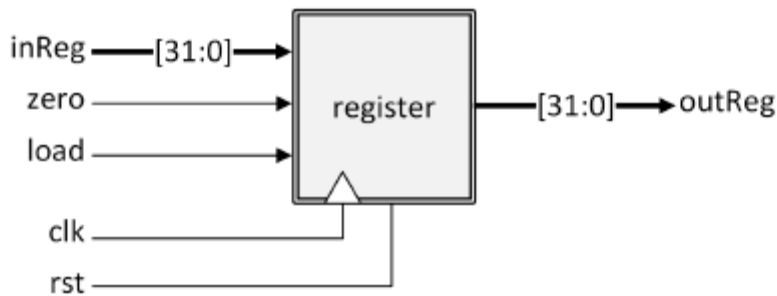
مرحله صورت گیرد. از این جهت به این نوع پردازنده ها multi-cycle (چند مرحله ای) می گویند.

این فرآیند برای تمامی دستورات تکرار می شود.



۱. برای اجرا مرحله `getInstr`، نیاز به تعیین کردن آدرس حافظه داریم تا متناسب با آدرس، دستور موجود در خانه ای از حافظه که آدرس به آن اشاره می کند، از حافظه حاوی دستورات خوانده شود. برای این کار، از یک ثبات (`register`) که حاوی آدرس حافظه است استفاده می کنیم. به این ثبات، شمارنده برنامه یا `Program Counter (PC)` می گویند. مقدار این ثبات، از آدرس اولین دستور در حافظه شروع شده و تا آدرس آخرین دستور در حافظه، ادامه می یابد، به همین خاطر به آن شمارنده برنامه گویند.

شماتیک ثابت در شکل زیر نشان داده شده است.



شکل (۲-۲) ثابت در پردازنده AFTAB [2]

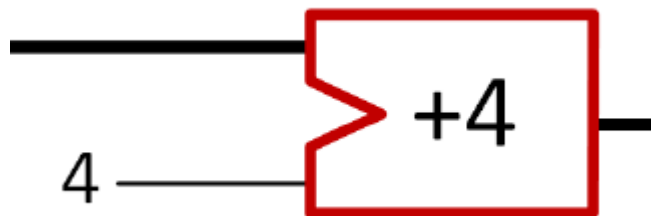
rst: اگر این سیگنال فعال شود، ثابت reset شده و مقدار ذخیره شده در ثابت پاک و مقدار آن برابر صفر می شود.

load: اگر این سیگنال فعال شود و همزمان سیگنال clk در لبه بالارونده (posedge) خود باشد، ۳۲ بیتی که در ورودی ثابت (inReg) هست عیناً در ثابت ذخیره می شود.

zero: اگر این سیگنال فعال شود، مقدار ذخیره شده در ثابت برابر صفر می شود.

outReg: خروجی ثابت است و ۳۲ بیتی که در ثابت ذخیره شده است را نشان می دهد.

برای اینکه آدرس ذخیره شده در ثابت Program Counter یا به اختصار PC از آدرس اولین دستور در حافظه شروع شده و تا آدرس آخرین دستور در حافظه، ادامه یابد، نیاز است تا مقدار این ثابت پس از اجرا هر دستور اضافه شود تا PC به آدرس دستور بعدی اشاره کرده و در نتیجه دستور بعدی اجرا شود. برای اینکار نیاز به یک جمع کننده است که شماتیک آن در شکل زیر نشان داده شده است:



شکل (۳-۲) جمع کننده PC در پردازنده AFTAB [3]

هر خانه حافظه ۱ بایت (۸ بیت) گنجایش دارد و هر سطر حافظه (word) حاوی ۴ خانه حافظه یا ۴ بایت (۳۲ بیت) است، در نتیجه پس از خواندن یک دستور که ۳۲ بیت و معادل ۴ بایت است، برای رفتن به دستور بعدی نیاز است تا آدرس حافظه را ۴ تا اضافه کنیم تا PC به سطر بعدی حافظه که حاوی دستور بعدی است، اشاره کند. به همین دلیل ورودی دوم جمع کننده را برابر ۴ قرار دادیم.

۲. برای رمزگشایی (decode) کردن دستور، نیاز به مدار دیجیتالی خاصی نیست. کافیتست باتوجه به ساختار دستور که در شکل زیر نشان داده شده است، بخش های مختلف دستور شامل نوع دستور، آرگومان های آن و... را از دستور خوانده شده از حافظه، استخراج کنیم.

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7				rs2		rs1	funct3		rd			opcode		R-type	
imm[11:0]						rs1	funct3		rd			opcode		I-type	
imm[11:5]				rs2		rs1	funct3		imm[4:0]			opcode		S-type	
imm[12]	imm[10:5]			rs2		rs1	funct3		imm[4:1]	imm[11]	opcode		B-type		
imm[31:12]										rd			opcode		U-type
imm[20]	imm[10:1]			imm[11]		imm[19:12]			rd			opcode		J-type	

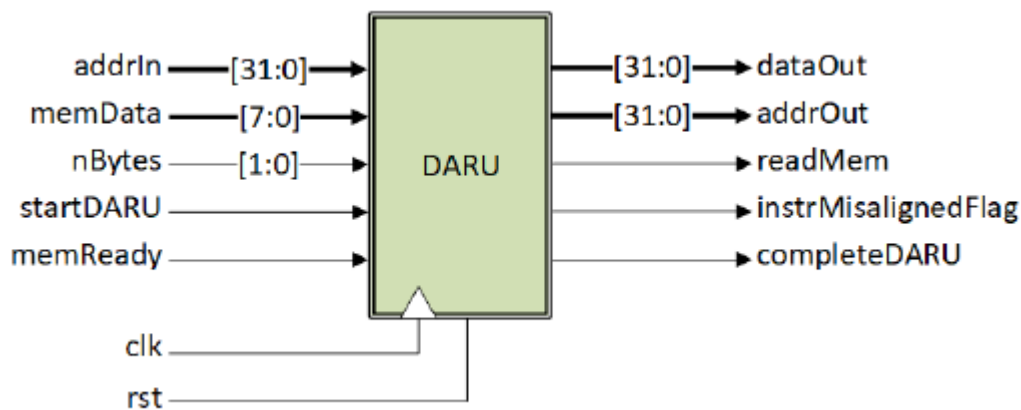
شکل (۲-۴) ساختار دستور در پردازنده AFTAB [4]

مثلا نوع دستور را می توان از ۷ بیت ابتدایی دستور (بیت های ۰ تا ۶) که بیانگر opcode (مخفف operation code که نشان دهنده نوع دستور است) استخراج کرد.

۳. حال پس از پی بردن به نوع دستور و آرگومان های آن، لازم است متناسب با نوع دستور، از یک یا چند مدار سخت افزاری، برای اجرا دستور استفاده کنیم:

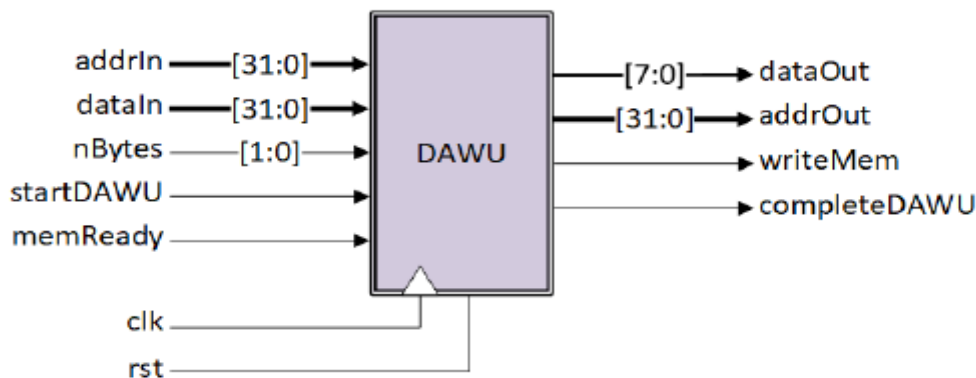
۱. اگر دستور load باشد، برای خواندن داده از حافظه خارجی کافیتست آدرسی که از حافظه قرار است بخوانیم را تعیین کنیم. برای این کار نیاز به مدار خاصی نیست، اما از جایی که هر سطر از حافظه دارای ۴ خانه است که هر خانه ۸ بیت یا یک بایت گنجایش دارد و هر داده در حافظه، دارای ۳۲ بیت یا ۴ بایت است که هر بایت آن به ترتیب در خانه های یک سطر از حافظه قرار می گیرد، نیاز است تا فرآیند خواندن در ۴ سیکل متوالی صورت گیرد تا در هر سیکل، محتوای یک خانه از سطر حافظه خوانده شده و آدرسی که از آن می خوانیم، یک واحد اضافه شود تا در

سیکل بعدی ۸ بیت موجود در خانه بعدی از همان سطر از حافظه خارجی خوانده شود تا در پایان ۴ سیکل، تمام داده ۳۲ بیتی که در یک سطر از حافظه خارجی قرار دارد، خوانده و در حافظه داخلی پردازنده (file register) ذخیره شود. برای اینکه این فرآیند انجام شود از ماژول Data Adjustment Read Unit (DARU) استفاده می‌کنیم. این ماژول که شماتیک آن در ادامه نشان داده شده است، ابتدا منتظر می‌ماند تا دستور خواندن از حافظه با فعال شدن سیگنال startDARU داده شود. سپس به تعداد ۴ سیکل (که با قرار دادن ۱۱ باینری (۳) در مبنای ۱۰) در دو بیت nBytes تعیین می‌شود، داده از حافظه می‌خواند. این واحد در هر سیکل برای خواندن از حافظه، سیگنال readMem را فعال می‌کند تا به حافظه خارجی دستور خواندن از حافظه دهد. سپس صبر می‌کند تا داده واقع در خانه ای از حافظه که آدرس آن توسط addrOut ۳۲ بیتی (که در اولین سیکل برابر آدرس ورودی (addrIn) است) تعیین می‌شود، توسط حافظه خارجی، در memData نوشته شود. این واحد به محض فعال شدن سیگنال memReady توسط حافظه خارجی، متوجه آماده بودن داده ای ارسالی از سمت حافظه خارجی شده و محتوای memData را در ثبات های داخلی خود ذخیره می‌کند. این فرآیند به تعداد nBytes که در واقع بیانگر تعداد بایتی است که باید از حافظه خوانده شود، تکرار می‌شود و در هر مرحله آدرس addrOut یکی زیاد می‌شود تا محتوای خانه بعدی حافظه خوانده شود. در آخر پس از پایان ۴ سیکل، ماژول DARU، ۴ بایت خوانده شده از حافظه خارجی، که در ثبات های خود ذخیره کرده است را در کنار هم قرار داده و به صورت یک داده ۳۲ بیتی در dataOut نوشته و فرآیند خواندن از حافظه خارجی در این مرحله پایان یافته و سیگنال completeDARU فعال می‌شود.



شکل (۲-۵) ماژول DARU در پردازنده AFTAB [5]

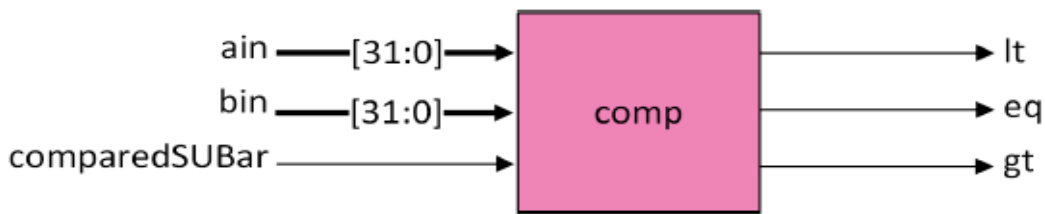
۲. اگر دستور store باشد همانند دستور load، نیاز است ۴ سیکل، داده بین پردازنده و حافظه خارجی رد و بدل شود، با این تفاوت که برای این دستور، بجای خواندن از حافظه، داده در حافظه ذخیره می شود. در نتیجه برای اجرا دستور store، به مداری سخت افزاری مانند DARU نیاز داریم که بجای ورودی memData، ورودی داده ۳۲ بیتی dataIn که در حافظه قرار است نوشته شود را داشته باشد. بجای سیگنال readMem، سیگنال writeMem داشته باشد که دستور نوشتن به حافظه خارجی را با فعال کردن آن، به حافظه خارجی دهد، و همچنین یک خروجی ۸ بیتی (۱ بایتی) به نام dataOut داشته باشد که در هر مرحله ۸ بیت از داده ای که قرار است در حافظه نوشته شود را در آن خروجی بنویسد و با فعال شدن سیگنال memReady متوجه شود که ۸ بیت در حافظه با موفقیت نوشته شده است. در نتیجه برای اجرا دستور store، از ماژولی به نام Data Adjustment Write Unit (DAWU) استفاده می کنیم که شماتیک آن در شکل زیر نشان داده شده است.



شکل (۲-۶) ماژول DAWU در پردازنده AFTAB [6]

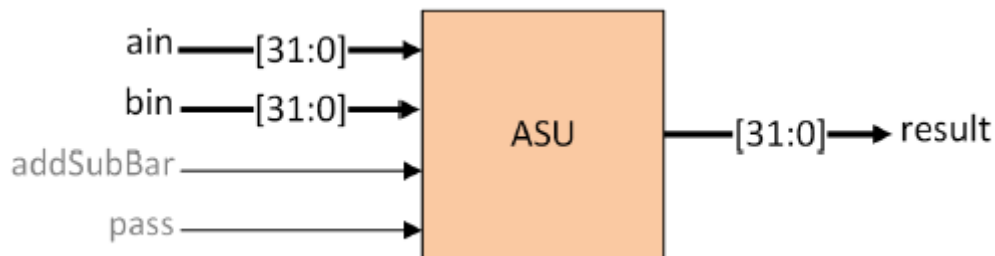
۳. اگر دستور از نوع محاسباتی باشد، از مدارات محاسباتی برای اجرا آن استفاده می کنیم:

۱. اگر دستور، مقایسه یا compare کردن دو عدد ۳۲ بیتی باشد از ماژول comparator که شماتیک آن در ادامه نشان داده شده است، استفاده می کنیم.



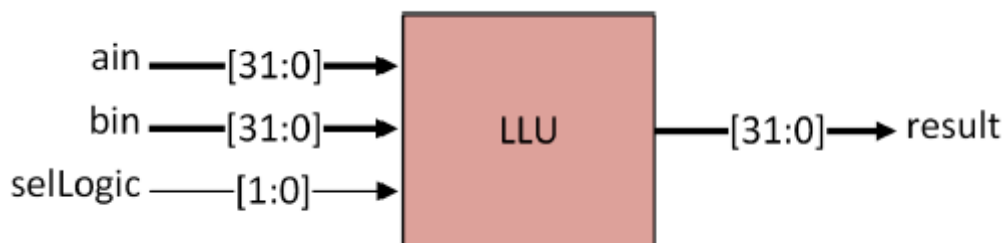
شکل (۷-۲) ماژول Comparator در پردازنده AFTAB [7]

۲. اگر دستور، جمع یا تفریق کردن دو عدد ۳۲ بیتی باشد از ماژول Adder/Subtractor Unit یا ASU که شماتیک آن در شکل زیر نشان داده شده است، استفاده می کنیم.



شکل (۸-۲) ماژول Adder/Subtractor Unit (ASU) در پردازنده AFTAB [8]

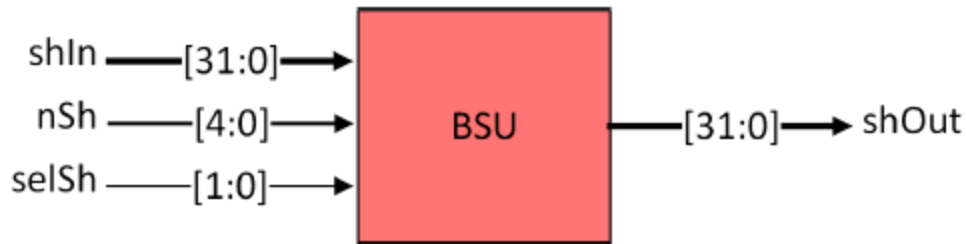
۳. اگر دستور، XOR، AND یا OR کردن دو عدد ۳۲ بیتی باشد از ماژول Logical Logic Unit یا LLU که شماتیک آن در شکل زیر نشان داده شده است، استفاده می کنیم. که مقدار selLogic تعیین می کند که ماژول، کدام یک از عملیات XOR، AND یا OR کردن را به ورودی ها اعمال کند.



شکل (۹-۲) ماژول Logical Logic Unit (LLU) در پردازنده AFTAB [9]

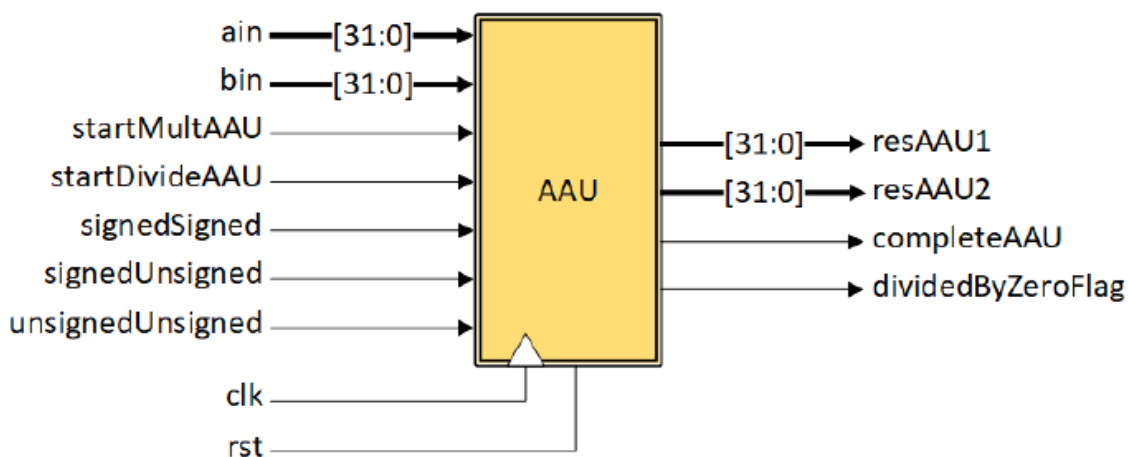


۴. اگر دستور، نوعی شیفت مانند (SLL) Shift Left Logical، (SRL) Shift Right Logical، یا (SRA) Shift Right Arithmetic کردن دو عدد ۳۲ بیتی باشد از ماژول Barrel Shifter Unit یا BSU که شماتیک آن در شکل زیر نشان داده شده است، استفاده می کنیم. که مقدار selShift تعیین می کند که ماژول، به چه شکلی ورودی را شیفت دهد.



شکل (۲-۱۰) ماژول Barrel Shifter Unit (BSU) در پردازنده AFTAB [10]

۵. اگر دستور، ضرب یا تقسیم کردن دو عدد ۳۲ بیتی باشد از ماژول Attached Arithmetic Unit یا AAU که شماتیک آن در شکل زیر نشان داده شده است، استفاده می کنیم. که می تواند همزمان در دو خروجی خود، نتیجه ضرب و تقسیم دو ورودی را قرار دهد که مقدار هر کدام از ورودی ها می تواند به صورت عدد علامت دار ( که بیت اول آن نشان دهنده علامت عدد است) یا به صورت بدون علامت ( که کل ۳۲ بیت بیانگر مقدار عدد است) محاسبه شود.



شکل (۲-۱۱) ماژول Attached Arithmetic Unit (AAU) در پردازنده AFTAB [11]

## ۲-۳- بخش دوم : Debugger

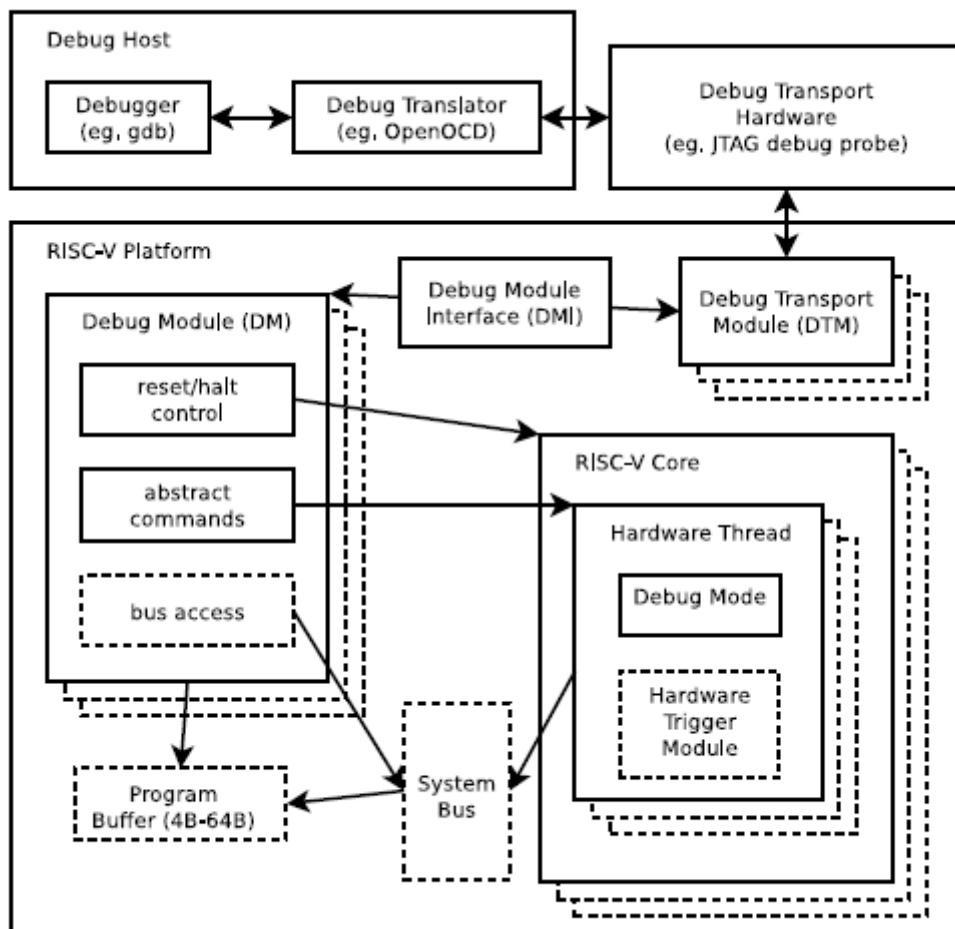
### ۲-۳-۱- مقدمه‌ای بر Debugger

پس از اینکه با پردازنده RISC-V و نحوه کارکرد آن آشنا شدیم. لازم است کمی راجع به Debugger و انتظاراتی که از یک Debugger می رود آشنا شویم. هدف کلی Debugger آن است که به برنامه نویسان این امکان را دهد که بتوانند روند اجرا برنامه در پردازنده را به خوبی مشاهده کنند تا سریع تر بتوانند دلیل نادرست بودن خروجی برنامه را پیدا و بتوانند به اصطلاح debug کنند. برای مشاهده روند اجرا برنامه، برنامه نویسان نیاز دارند که بتوانند اجرا برنامه را در هر قسمتی از برنامه متوقف کنند و مقادیر متغیرها تا جایی که برنامه اجرا شده است را ببینند. در این صورت می توانند تاثیر اجرا شدن هر خط از کد بر مقادیر متغیرها و... را به وضوح دیده و منشأ نادرست بودن خروجی را با سختی کمتری پیدا کنند. پس بر این اساس، Debugger باید بتواند اجرا برنامه را در هر قسمتی از برنامه متوقف کرده و در حین اینکه اجرا برنامه متوقف شده و Program Counter پردازنده ثابت مانده و دستور جدیدی اجرا نمی شود، مقادیری که تا این بخش از برنامه در حافظه خارجی و همچنین حافظه داخلی پردازنده (register file) ذخیره شده است را از پردازنده، دریافت و آنان را به برنامه نویس نمایش دهد. در مواقعی، برنامه نویسان نیاز دارند تا مقدار یک متغیر را دستی، در جایی مشخصی از برنامه مقداردهی کنند تا تاثیر مقدار آن متغیر بر اجرا برنامه را مشاهده کنند. بدین منظور Debugger باید بتواند علاوه بر خواندن مقادیر ذخیره شده در حافظه خارجی و حافظه داخلی پردازنده، مقدار مشخصی که توسط برنامه نویس مشخص می شود را در آنان ذخیره کند تا مقدار متغیری که در حافظه ذخیره شده است را به این صورت تغییر داده و تاثیر مقدار آن متغیر بر روند اجرا برنامه، با از سر گرفتن اجرا برنامه، از جایی که متوقف شده بود، مشخص شود. در نتیجه Debugger باید بتواند:

۱. اجرا برنامه را در هر قسمتی از برنامه، متوقف (pause) کند.
۲. در حین اینکه برنامه متوقف شده است، مقادیر ذخیره شده در حافظه خارجی و ثبات های حافظه داخلی پردازنده را خوانده و به برنامه نویس نمایش دهد.
۳. در حین اینکه برنامه متوقف شده است، مقدار مشخصی که از سوی برنامه نویس تعیین شده است را در حافظه خارجی یا ثبات های حافظه داخلی پردازنده ذخیره کند.
۴. اجرا برنامه را، از جایی که متوقف (pause) شده بود، ادامه داده و به اصطلاح resume کند.

## ۲-۳-۲- تعامل Debugger با پردازنده RISC-V

Debugger برای اینکه بتواند پردازنده را متوقف کند، مقادیر حافظه خارجی و داخلی پردازنده را بخواند یا مقداری در آنان ذخیره کند، و در آخر، اجرا برنامه را از سر گیرد، نیاز دارد تا درخواست خود را به پردازنده اعلام کند تا پردازنده در پاسخ به Debugger، پردازنده را متوقف کند، مقادیر را خوانده و در اختیار Debugger قرار دهد یا مقداری در حافظه خارجی یا داخلی خود ذخیره کند، و یا اجرا برنامه را از جایی که متوقف شده بود ادامه دهد. بنابراین Debugger و پردازنده نیاز دارند تا با هم به صورت مستقیم در تعامل باشند. نحوه تعامل Debugger و پردازنده در شکل زیر نشان داده شده است.



شکل (۲-۱۲) سیستم Debug پردازنده RISC-V (به صورت کلی) [12]

برنامه نویس برای اینکه بتواند گدی که نوشته است را debug کند، در قسمت هایی از برنامه ای که نوشته است breakpoint قرار می دهد تا با رسیدن اجرا برنامه به خطی که breakpoint قرار دارد، اجرا برنامه در آن خط متوقف شده و مقادیر متغیر ها تا به آنجا برنامه، برای برنامه نویس نمایش داده شود. در نتیجه وقتی برنامه به خطی که breakpoint دارد می رسد، لازم است اجرا برنامه بر روی پردازنده متوقف (pause) شده و مقادیر متغیر ها از حافظه خارجی یا داخلی پردازنده خوانده شود. برای اینکار، ابتدا درخواست متوقف کردن پردازنده، از محیطی که در آن برنامه debug می شود (Debug Host)، خارج شده و از طریق سیم و ارتباطات سخت افزاری (Debug Transport Hardware) مانند JTAG به سخت افزاری که در آن برنامه اجرا می شود (RISC-V Platform)، منتقل می شود. در سخت افزاری که برنامه در آن اجرا می شود، بخش Debug Transport Module (DTM) درخواست را از ارتباطات سخت افزاری دریافت کرده و درخواست را به Debug Module Interface (DMI) منتقل می کند. این واحد مانند واسطه بین Debug Host و Debugger عمل کرده و درخواست ارسال شده از Debug Host را به Debugger منتقل می کند. در آخر Debugger یا Debug Module (DM) درخواست خود را به پردازنده RISC-V اعلام کرده و نتیجه درخواست (مقدار خوانده شده از حافظه خارجی یا داخلی پردازنده) از پردازنده به Debugger، و از Debugger به DMI، و از DMI از طریق DTM به Debug Host ارسال شده و نتیجه درخواست یا همان مقدار خوانده شده از حافظه خارجی یا داخلی پردازنده که در واقع مقدار یک متغیر است برای برنامه نویس نمایش داده می شود.

## ۲-۴ - خلاصه و جمع بندی

در این فصل با مفاهیم اولیه و پیش زمینه هایی جهت درک هرچه بهتر پردازنده RISC-V، Debugger و نحوه تعامل آن با پردازنده RISC-V جهت debug کردن آشنا شدیم.

## فصل ۳

### طراحی Debugger

---

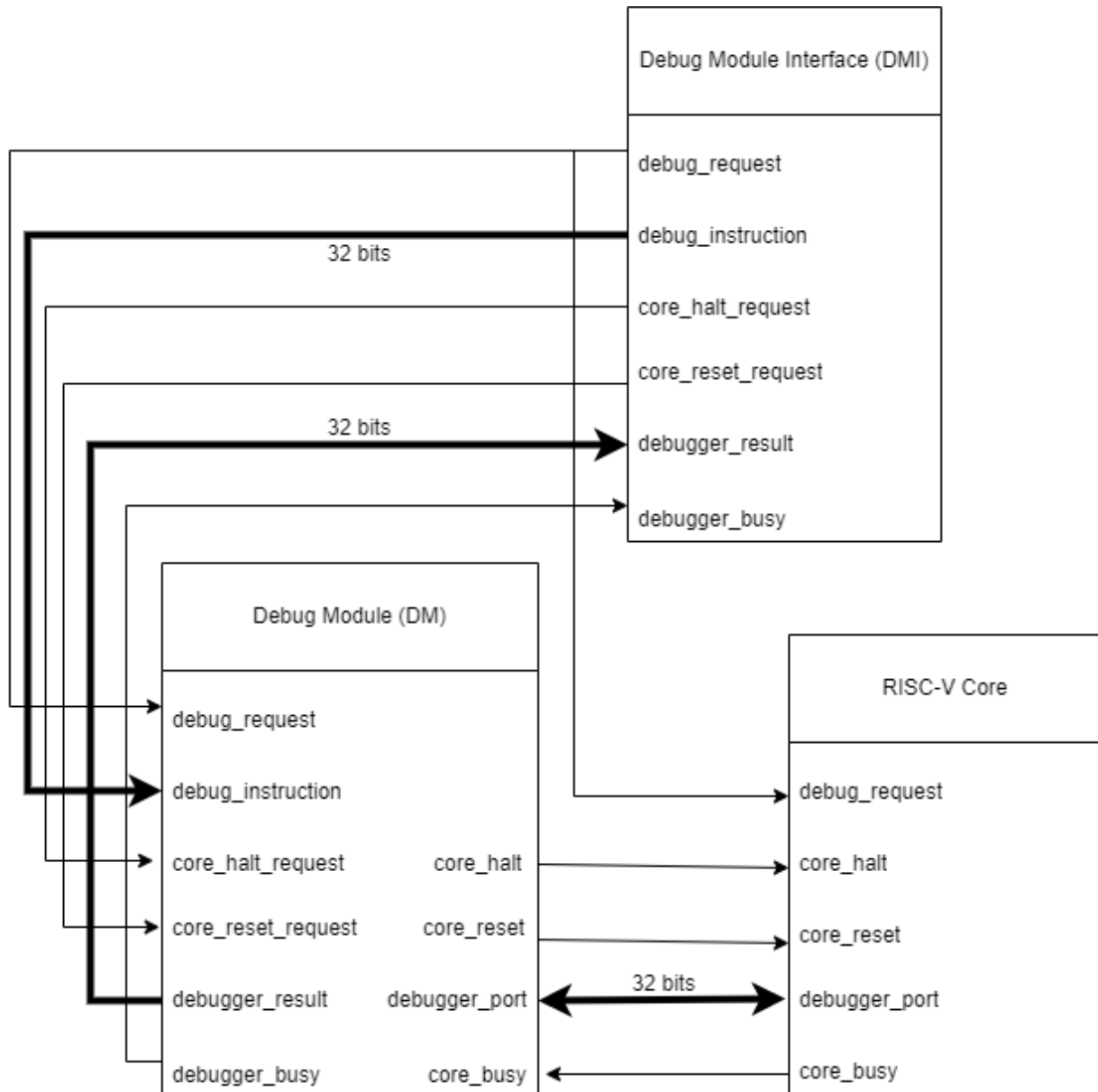
فصل سوم در برگیرنده‌ی توضیح مربوط به ساختار و طراحی Debugger برای پردازنده RISC-V می باشد.

### ۳-۱ - مقدمه

در این فصل نخست به معرفی فرآیندی که طی می شود تا Debugger، درخواست آمده از سوی (DMI) Debug Module Interface را، به پردازنده ارسال، و پاسخ درخواست ارسالی را، از پردازنده RISC-V، دریافت و پاسخ را به DMI منتقل کند، می پردازیم. در ادامه بر اساس این فرآیند، ساختار Debugger را تشریح کرده و در آخر به روندی که در پردازنده طی می شود تا به درخواست آمده از سوی DMI، پاسخ داده شود خواهیم پرداخت.

### ۳-۲- روش پیشنهادی برای طراحی Debugger

برای طراحی Debugger، ابتدا لازم است تا با فرآیند ارسال درخواست از DMI به DM، و سپس ارسال درخواست از سوی Debug Module (DM) به پردازنده RISC-V، دریافت پاسخ از پردازنده توسط DM و ارسال پاسخ درخواست از DM به Debug Module Interface یا DMI آشنا شویم:



شکل (۳-۱۳) سیستم Debug پردازنده RISC-V (همراه با جزئیات کامل) [13]

## ۱. در سمتِ Debug Module Interface یا DMI:

ابتدا DMI سیگنال `debug_request` را فعال کرده و درخواستی که از پردازنده دارد را بر روی باس ۳۲ بیتی `debug_instruction` به DM اعلام می کند. اگر دستور، دستوری باشد که می خواهد داده ای را در حافظه خارجی یا داخلی پردازنده ذخیره کند، پس از این سیکل، در سیکل بعدی، DMI بر روی این باس، داده ای را که قرار است در بخشی از حافظه نوشته شود، قرار می دهد و اگر دستور، دستوری باشد که با حافظه خارجی سروکار داشته و نیاز به تعیین آدرس برای خواندن یا نوشتن به آن آدرس از حافظه خارجی دارد، پس از قرار گرفتن آرگومان داده، در سیکل بعدی آن، DMI بر روی باس `debug_instruction`، آدرسی را که قرار است به آن آدرس از حافظه خارجی، داده ای نوشته یا از آن خوانده شود، قرار می دهد. در نتیجه هر درخواست طی ۳ سیکل از DMI به DM منتقل می شود:

- در سیکل اول DMI، سیگنال `debug_request` را فعال کرده و درخواستی که از پردازنده دارد را بر روی باس ۳۲ بیتی `debug_instruction` قرار می دهد.
- در سیکل دوم، آرگومان داده دستور که داده ای است که قرار است به حافظه خارجی یا داخلی پردازنده نوشته شود، بر روی باس `debug_instruction` توسط DMI قرار داده می شود.
- در سیکل سوم، آرگومان آدرس دستور که آدرسی است که قرار است به آن آدرس از حافظه خارجی داده ای نوشته یا از آن خوانده شود، بر روی باس `debug_instruction` توسط DMI قرار داده می شود.

اگر درخواست DMI، آرگومان داده یا آدرس نداشته باشد، DMI در سیکل متناظر با آرگومان داده و آدرس، چیزی بر روی باس ۳۲ بیتی `debug_instruction` قرار نمی دهد و به عبارتی باس `debug_instruction` برابر ۳۲ بیت z می شود.

## ۲. در سمتِ Debug Module یا DM:

- ابتدا با مشاهده فعال شدن سیگنال `debug_request`، DM متوجه درخواست DMI شده و داده موجود روی باس `debug_instruction` که در این سیکل، برابر با درخواست DMI از پردازنده است را در ثبات (register) داخلی خود (Debugger\_Command\_Register) ذخیره می کند.
- سپس در سیکل دوم، DM، داده موجود روی باس `debug_instruction` که در این سیکل، برابر با آرگومان داده درخواست DMI از پردازنده است را در ثبات (register) داخلی خود (Debugger\_Command\_Data\_Argument\_Register) ذخیره کرده و همزمان درخواست DMI، که در سیکل قبلی در ثبات Debugger\_Command\_Register ذخیره شده بود را، از



- طریق باس ۳۲ بیتی `debugger_port` به پردازنده RISC-V منتقل می کند.
- در سیکل سوم، DM، داده موجود روی باس `debug_instruction` که در این سیکل، برابر با آگومان آدرس درخواست DMI از پردازنده است را در ثبات (register) داخلی خود (`Debugger_Command_Address_Argument_Register`) ذخیره کرده و همزمان آگومان داده درخواست DMI، کسه در سیکل قبلی در ثبات `Debugger_Command_Data_Argument_Register` ذخیره شده بود را، از طریق باس ۳۲ بیتی `debugger_port` به پردازنده RISC-V منتقل می کند.
  - در چهارمین سیکل، DM، آگومان آدرس درخواست DMI، که در سیکل قبلی در ثبات `Debugger_Command_Address_Argument_Register` ذخیره شده بود را، از طریق باس ۳۲ بیتی `debugger_port` به پردازنده RISC-V منتقل می کند. در این سیکل و پس از ۳ سیکل فوق، درخواست DMI و آگومان های آن، به DM به صورت کامل، منتقل شده و DM، سیگنال `debugger_busy` را فعال می کند تا به DMI اعلام کند که درخواست و آگومان های آن کاملاً دریافت شده و مشغول بررسی درخواست است.
  - پس از این ۴ سیکل، درخواست DMI و آگومان های آن، به پردازنده به صورت کامل، منتقل شده، پردازنده شروع به پردازش درخواست DMI کرده و سیگنال `core_busy` را فعال می کند تا به DM اعلام کند که درخواست و آگومان های آن کاملاً دریافت شده و مشغول بررسی درخواست است.
  - پس از گذشت چند سیکل، با غیرفعال شدن سیگنال `core_busy`، DM متوجه اتمام پردازش دستور توسط پردازنده شده و نتیجه درخواست که بر روی باس `debugger_port` توسط پردازنده قرار داده شده است، را در ثبات (register) داخلی خود (`Debugger_Result_Register`) ذخیره می کند.
  - در سیکل بعدی، DM، سیگنال `debugger_busy` را غیرفعال کرده و نتیجه درخواست DMI را بر روی باس ۳۲ بیتی `debugger_result` قرار می دهد تا DMI با غیرفعال شدن سیگنال `debugger_busy`، متوجه آماده بودن نتیجه درخواست شده و نتیجه درخواست را از روی این باس بردارد.

### ۳. در سمت پردازنده RISC-V:

برای اینکه درخواست DMI و آگومان های آن در پردازنده قرار گیرد و سپس درخواست DMI

باتوجه به بخش های مختلفی که دارد، رمزگشایی یا decode شده و به آن پاسخ داده شود، در سمت پردازنده، ماژولی به نام `debugger_command_decoder` قرار می دهیم که درخواست ارسالی از سمت DMI و آرگومان های آن را در ثبات های داخلی خود ذخیره کرده و سپس باتوجه به درخواست و بخش های مختلف آن، درخواست را رمزگشایی کرده و متوجه نوع دستور و... شود:

- ابتدا با مشاهده فعال شدن سیگنال `debug_request`، پردازنده متوجه درخواست DM می شود.
- در سیکل دوم، DM درخواست ارسالی از سوی DMI را بر روی باس `debugger_port` قرار داده و پردازنده نیز آن را از روی این باس خوانده و در ثبات داخلی `debugger_command_decoder` به نام `Debugger_Command_Register` ذخیره می کند.

- در سیکل سوم، DM آرگومان داده درخواست ارسالی از سوی DMI را بر روی باس `debugger_port` قرار داده و پردازنده نیز آن را از روی این باس خوانده و در ثبات داخلی `debugger_command_decoder` به نام `Debugger_Command_Data_Argument_Register` ذخیره می کند.

- در سیکل چهارم، DM آرگومان آدرس درخواست ارسالی از سوی DMI را بر روی باس `debugger_port` قرار داده و پردازنده نیز آن را از روی این باس خوانده و در ثبات داخلی `debugger_command_decoder` به نام `Debugger_Command_Address_Argument_Register` ذخیره می کند.

- پس از این ۴ سیکل، درخواست DMI و آرگومان های آن، به صورت کامل به پردازنده منتقل شده و پردازنده، سیگنال `core_busy` را فعال می کند تا به DM اعلام کند که درخواست و آرگومان های آن کاملاً دریافت شده و مشغول بررسی درخواست است. سپس ماژول `debugger_command_decoder` باتوجه به درخواست DMI که در ثبات داخلی `Debugger_Command_Register` ذخیره شده است، متوجه نوع دستور شده و نوع دستور را به کنترلر پردازنده اعلام می کند. در آخر پردازنده باتوجه به نوع دستور، طی چند سیکل، به درخواست پاسخ داده، سیگنال `core_busy` را غیر فعال کرده و پاسخ را بر روی باس `debugger_port` قرار می دهد تا DM با غیر فعال شدن سیگنال `core_busy`، متوجه آماده بودن پاسخ درخواست شده و آن را از روی باس `debugger_port` دریافت کند.

لازم به ذکر است که اگر DMI، صرفاً قصد متوقف (pause) کردن پردازنده را داشته باشد، سیگنال `core_halt_request` را فعال می کند. سپس DM با فعال شدن این سیگنال، سیگنال `core_halt` که از DM

خارج و به پردازنده وارد شده است را فعال کرده و در نتیجه، پردازنده، درخواست `core_halt_request` ارسال شده از سوی DMI را دریافت کرده و اجرا برنامه را با ثابت نگه داشتن Program Counter (PC)، متوقف و pause می کند. مشابه همین روند برای حالتی که DMI صرفاً قصد reset کردن پردازنده را داشته باشد، طی می شود با این تفاوت که در این حالت، سیگنال `core_reset_request` توسط DMI فعال شده و DM هم سیگنال `core_reset` را فعال کرده و پردازنده با صفر کردن Program Counter (PC)، reset می شود.

### ۳-۲-۱ - ساختار Debugger

بر اساس فرآیند ارسال درخواست، از سوی DMI به DM، سپس ارسال شدن درخواست از DM به پردازنده، دریافت کردن پاسخ از پردازنده و ارسال پاسخ از DM به DMI که پیش تر توضیح داده شد، DM یا همان Debugger Module، دارای ۳ ثبات داخلی می باشد که ورودی آنان، باس ۳۲ بیتی `debug_instruction` است تا در سیکل اول، درخواست DMI، در سیکل دوم آرگومان داده درخواست و در سیکل سوم، آرگومان آدرس درخواست در این ثبات ها ذخیره شود:

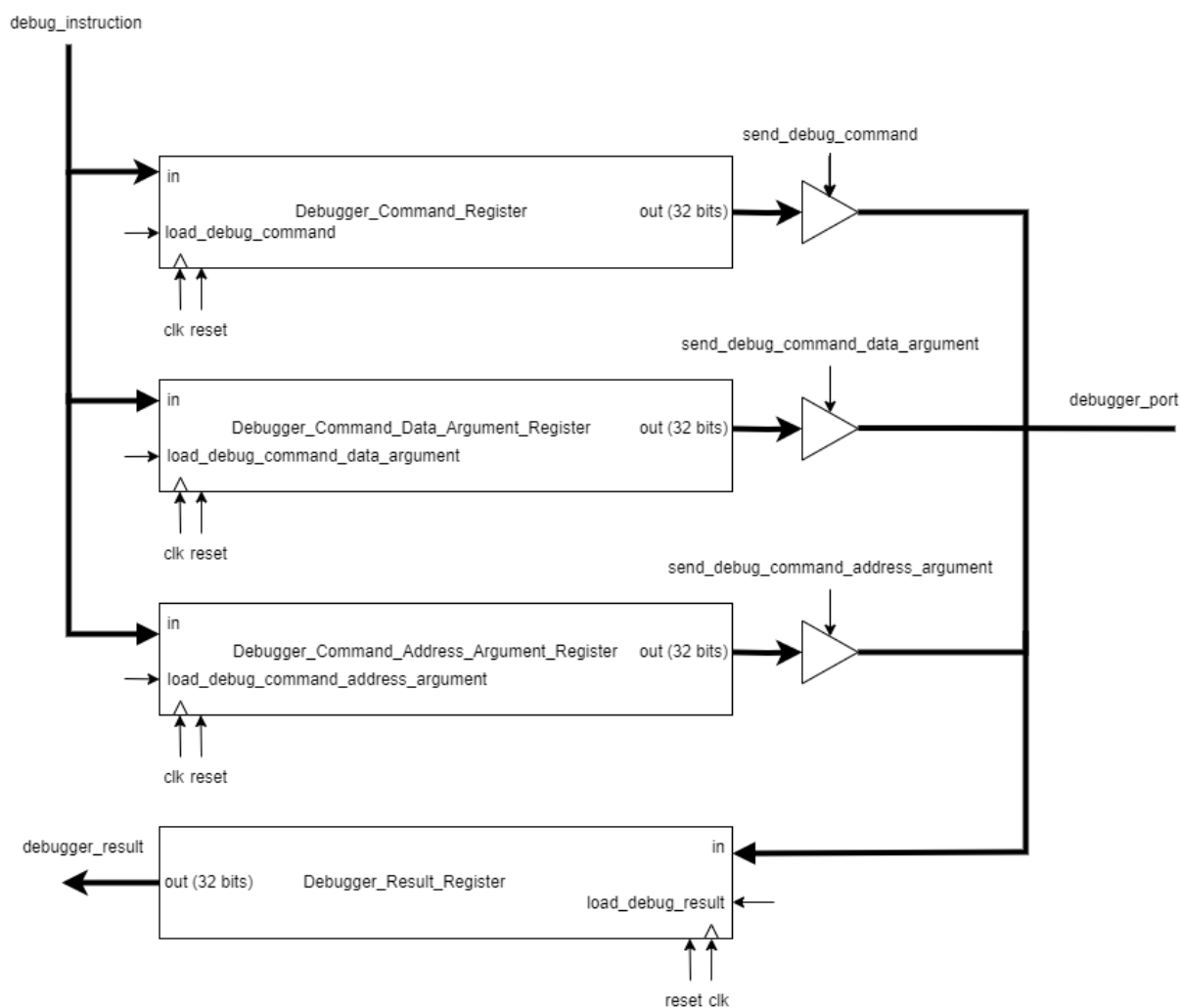
Debugger\_Command\_Register

Debugger\_Command\_Data\_Argument\_Register

Debugger\_Command\_Address\_Argument\_Register

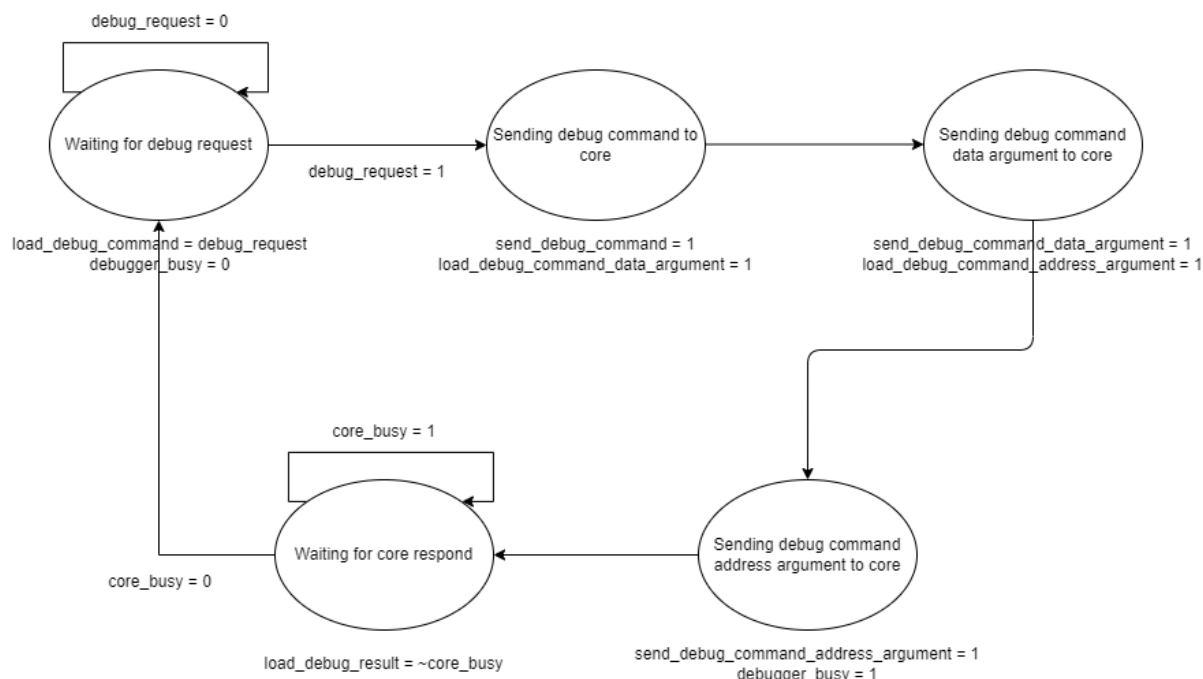
و باید بتوان به هر کدام داده ای نوشته شود، در نتیجه هر کدام از این ثبات ها، سیگنالی برای load کردن دارند تا بتوان مقداری که بر روی باس `debug_instruction` قرار دارد، در این ثبات ها ذخیره شود. همچنین از جایی که فقط یک باس ۳۲ بیتی به نام `debugger_port` بین DM و پردازنده است، در هر سیکل، فقط یک ثبات می تواند داده بر روی این باس قرار دهد. اگر چند ثبات، همزمان داده روی باس قرار دهند، داده ها مخدوش شده و داده ای اشتباه به پردازنده ارسال می شود. در نتیجه برای اینکه بتوان کاری کرد که در هر سیکل فقط یک ثبات، داده بر روی باس `debugger_port` قرار دهد، خروجی هر ثبات را به یک tristate متصل کرده و کنترلر DM، در هر سیکل فقط یکی از tristate ها را فعال می کند. علاوه بر این ۳ ثبات، Debugger نیاز به یک ثبات به نام `Debugger_Result_Register` دارد که ورودی آن باس `debugger_port` باشد تا پاسخ پردازنده را از باس `debugger_port` برداشته و در خود ذخیره کند.

بنابراین مسیر داده یا همان Data Path در ماژول DM به صورت زیر می باشد:



شکل (۳-۱۴) مسیر داده (DM) Debug Module [14]

و مرحله‌ای که در کنترلر DM طی می‌شود تا درخواست DMI به پردازنده منتقل شده و پاسخ درخواست، از پردازنده به DMI منتقل شود به صورت زیر است:



شکل (۳-۱۵) کنترل کننده Debug Module (DM) [15]

۱. در ابتدا، DM منتظر می‌ماند تا با فعال شدن سیگنال `debug_request`، درخواستی از سوی DMI ارسال شود. با فعال شدن این سیگنال، سیگنال `load_debug_command` نیز فعال شده و در نتیجه درخواست DMI که بر روی بوس `debug_instruction` توسط DMI قرار داده شده است، در ثبات `Debugger_Command_Register` ذخیره می‌شود.

۲. پس از ذخیره شدن درخواست DMI در DM، سیگنال `load_debug_command_data_argument` فعال شده و آرجومان داده درخواست DMI که بر روی بوس `debug_instruction` توسط DMI قرار داده شده است، در ثبات `Debugger_Command_Data_Argument_Register` ذخیره می‌شود. همچنین، با فعال شدن سیگنال `send_debug_command`، tristate متناظر با ثبات `Debugger_Command_Register` فعال شده و درخواست DMI بر روی بوس `debugger_port` قرار داده می‌شود تا پردازنده، آن را از روی این بوس دریافت کند.

۳. پس از ذخیره شدن آرجومان داده درخواست DMI در DM، سیگنال `load_debug_command_address_argument` فعال شده و آرجومان آدرس درخواست DMI که بر

روی باس debug\_instruction توسط DMI قرار داده شده است، در ثبات  
 Debugger\_Command\_Address\_Argument\_Register ذخیره می شود. همچنین، با فعال شدن  
 سیگنال tristate send\_debug\_command\_data\_argument، متناظر با ثبات  
 Debugger\_Command\_Data\_Argument\_Register فعال شده و آرگومان داده درخواست DMI  
 بر روی باس debugger\_port قرار داده می شود تا پردازنده، آرگومان داده را از روی این باس دریافت  
 کند.

۴. پس از ذخیره شدن آرگومان آدرس درخواست DMI در DM، با فعال شدن سیگنال  
 tristate send\_debug\_command\_address\_argument، متناظر با ثبات  
 Debugger\_Command\_Address\_Argument\_Register فعال شده و آرگومان آدرس درخواست  
 DMI بر روی باس debugger\_port قرار داده می شود تا پردازنده، آرگومان آدرس را از روی این باس  
 دریافت کند. در این مرحله درخواست DMI و آرگومان های آن به پردازنده به صورت کامل منتقل  
 شده و DM، سیگنال debugger\_busy را فعال می کند تا به DMI اعلام کند که درخواست و  
 آرگومان های آن کاملاً دریافت شده و مشغول بررسی درخواست است.

۵. در این مرحله DM صبر می کند تا با غیر فعال شدن سیگنال core\_busy، پردازش دستور توسط  
 پردازنده خاتمه یابد و نتیجه درخواست را با فعال کردن سیگنال load\_debug\_result از روی باس  
 debugger\_port در ثبات Debugger\_Result\_Register ذخیره کند. با پرسش از این مرحله به مرحله  
 ۱، سیگنال debugger\_busy غیر فعال شده تا DMI، با غیر فعال شدن این سیگنال، متوجه آماده  
 بودن نتیجه درخواست شده و نتیجه درخواست را از روی باس ۳۲ بیتی debug\_result که در  
 واقع خروجی ثبات Debugger\_Result\_Register است، بردارد.

### ۳-۲-۲- پاسخ دادن پردازنده به درخواست Debugger

بر اساس فرآیند ارسال درخواست، از سوی DMI به DM، سپس ارسال شدن درخواست از DM به پردازنده، دریافت کردن پاسخ از پردازنده و ارسال پاسخ از DM به DMI که پیش تر توضیح داده شد، پردازنده برای پاسخ دادن به درخواست آمده از سوی DM، مراحل زیر را به ترتیب طی می کند:

۱. ابتدا پردازنده، از یک سیکل پس از فعال شدن سیگنال `debug_request`، شروع به ذخیره کردن درخواست و آرگومان های آن، که هر سیکل، توسط DM، بر روی `debugger_port` گذاشته می شود کرده و آنان را در ثبات های داخلی ماژول `debugger_command_decoder` می کند. پس از اتمام این فرآیند، پردازنده سیگنال `core_busy` را فعال می کند تا به DM اعلام کند که درخواست و آرگومان های آن کاملاً دریافت شده و مشغول بررسی درخواست است.
۲. سپس ماژول `debugger_command_decoder`، بر اساس درخواست ذخیره شده در ثبات داخلی خود، درخواست را رمزگشایی (decode) کرده و نوع دستور را به کنترلر پردازنده اطلاع می دهد.
۳. پس از فهمیدن نوع درخواستی که DM از پردازنده دارد، پردازنده طی چند مرحله، به آن درخواست پاسخ داده و سیگنال `core_busy` را غیرفعال می کند تا DM متوجه اتمام پردازش دستور شده و پاسخ درخواست را از روی باس `debugger_port` بردارد.

برای اجرا مرحله ۱:

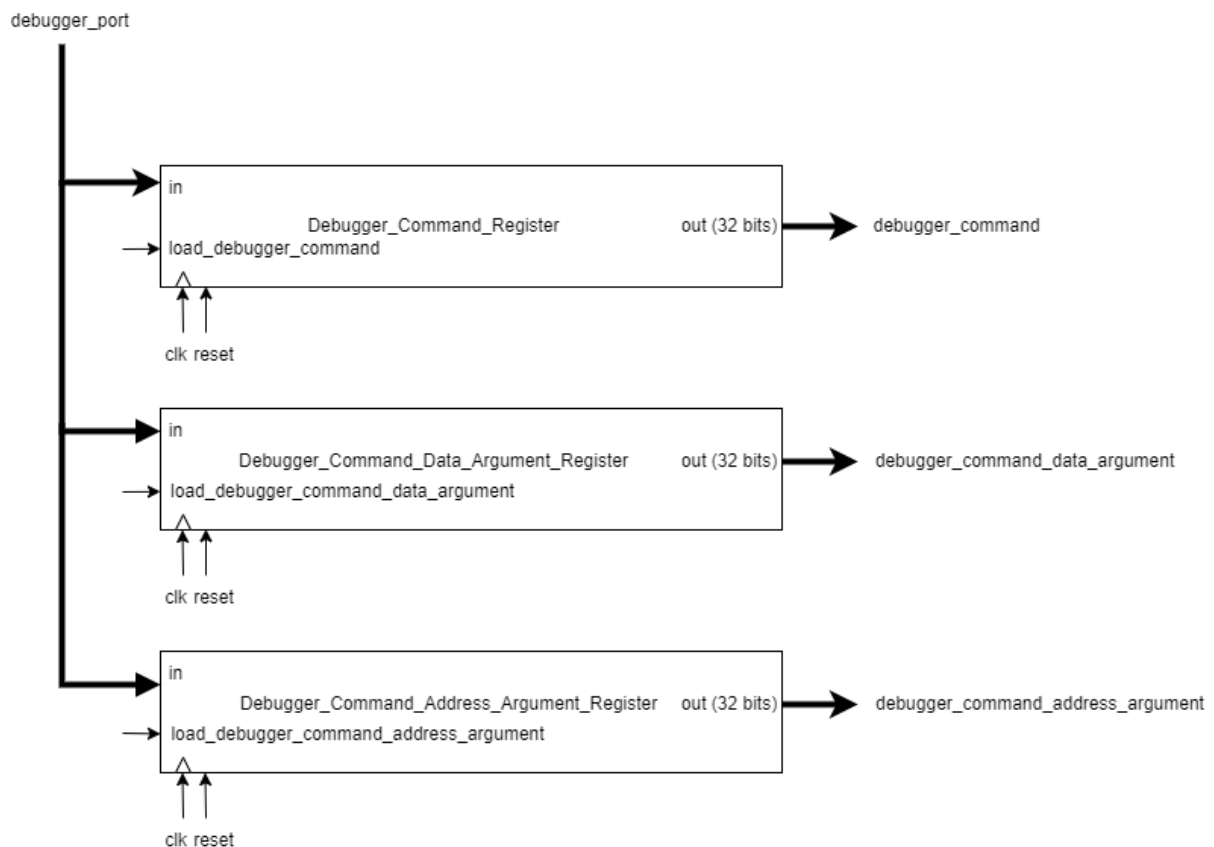
ماژول `debugger_command_decoder`، باید دارای ۳ ثبات داخلی باشد که درخواست، آرگومان داده و آرگومان آدرس درخواست را بتواند ذخیره کند تا بعد از آن بر اساس محتوای درخواست و بخش های آن، درخواست DM رمزگشایی شود:

`Debugger_Command_Register`

`Debugger_Command_Data_Argument_Register`

`Debugger_Command_Address_Argument_Register`

و باید بتوان به هر کدام داده ای نوشته شود، در نتیجه هر کدام از این ثبات ها، سیگنالی برای `load` کردن دارند که توسط کنترلر پردازنده فعال می شود تا بتوان مقداری که بر روی بوس `debugger_port` قرار دارد، در این ثبات ها ذخیره شود. بنابراین ساختار ماژول `debugger_command_decoder` به صورت زیر می باشد:



شکل (۳-۱۶) ماژول Debugger Command Decoder [16]



برای اجرا مرحله ۲:

ماژول `debugger_command_decoder`، برای اینکه بتواند نوع درخواست را تشخیص دهد، با توجه به تصویر زیر که بیانگر ساختار درخواستی است که DM به پردازنده ارسال می کند، بیت های ۲۴ تا ۳۱ (`cmdtype`) را مورد بررسی قرار می دهد:

- اگر این ۸ بیت صفر باشد، درخواست از نوع Register Access است که به این معنا است که DM، درخواست خواندن یا نوشتن به ثباتی در حافظه داخلی پردازنده (Register File) را دارد.

- اگر این ۸ بیت معادل ۲ در مبنای ۱۰ باشد، درخواست از نوع Memory Access است که به این معنا است که DM، درخواست خواندن یا نوشتن به خانه ای از حافظه خارجی را دارد.

حال بعد از اینکه نوع درخواست مشخص شد، `debugger_command_decoder` با توجه به ساختار درخواست DM، بیت ۱۶ (`write`) از درخواست را مورد بررسی قرار می دهد تا تشخیص دهد که DM قصد کدام یک از عملیات نوشتن یا خواندن را دارد:

- اگر این بیت صفر باشد، به این معنا است که DM، قصد خواندن دارد.

- اگر این بیت یک باشد، به این معنا است که DM، قصد نوشتن دارد.

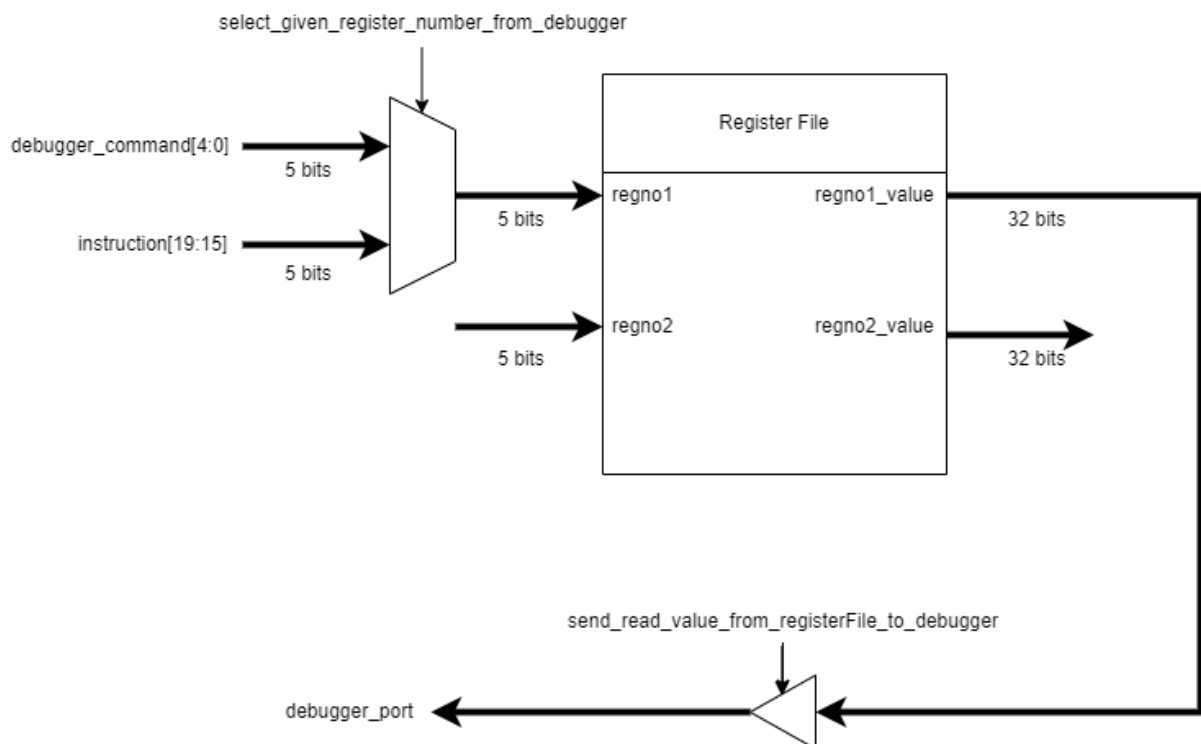
اگر دستور از نوع Register Access باشد، مطابق با تصویر زیر، `debugger_command_decoder` از بیت های ۰ تا ۴ (`regno`) تشخیص می دهد که DM قصد نوشتن یا خواندن به کدام ثبات را دارد. از جایی که ۳۲ ثبات در حافظه داخلی پردازنده قرار دارد، برای اینکه بتوان ۳۲ مقدار مختلف جهت آدرس دهی ثبات ها داشت، نیاز به ۵ بیت است تا در نتیجه ۲<sup>۵</sup> یا ۳۲ حالت مختلف برای تعیین ثبات وجود داشته باشد.

31	24	23	22	20	19	18	17	16	15	0
cmdtype	0	aarsize	aarpostincrement	postexec	transfer	write	regno			
8	1	3	1	1	1	1	16			

شکل (۳-۱۷) ساختار درخواست (DM) [17]

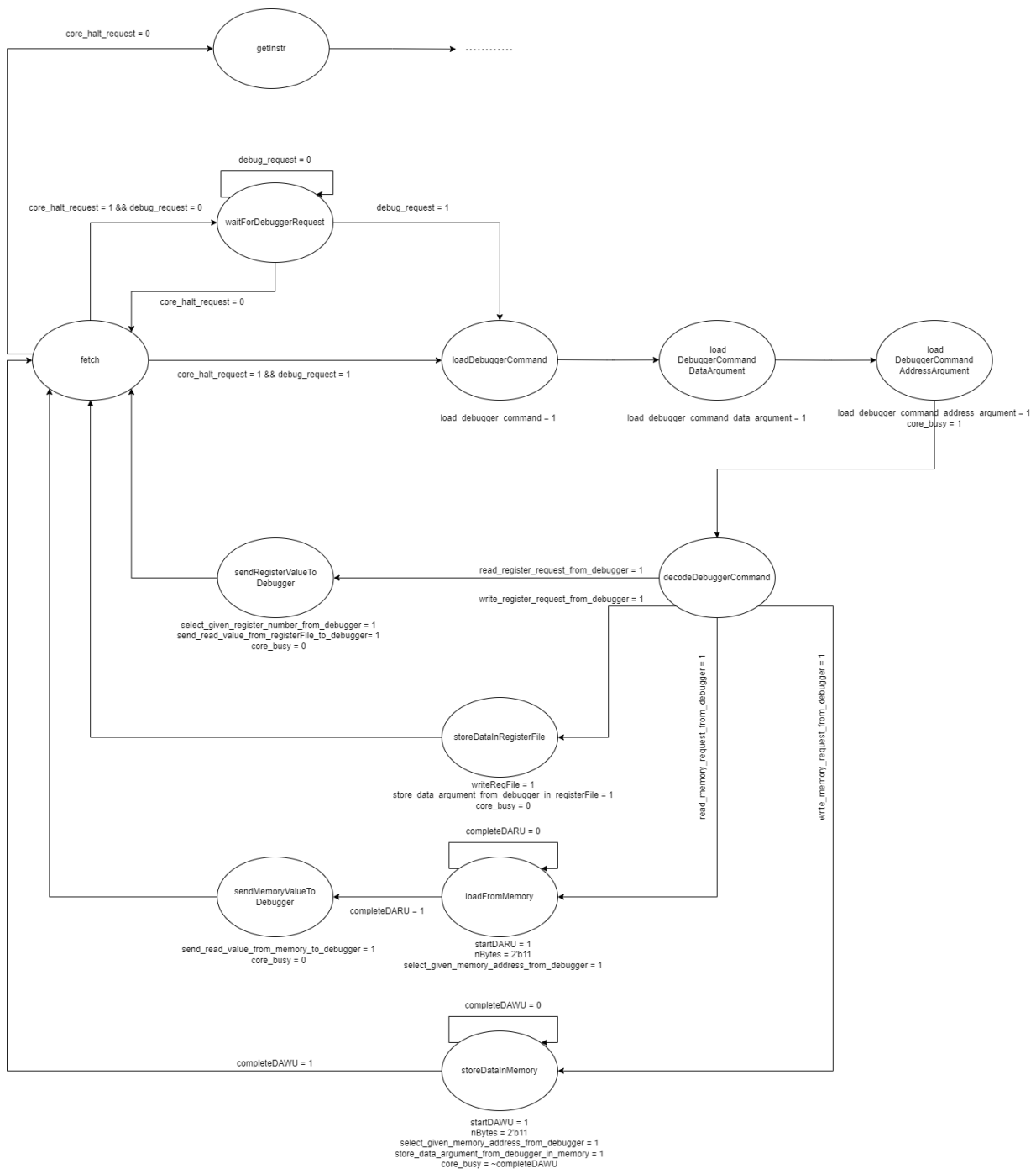
برای اجرا مرحله ۳:

۱. اگر درخواست DM، خواندن مقدار ثباتی از حافظه داخلی پردازنده (register file) باشد: کافست ورودی register file که تعیین کننده آن است که مقدار کدام ثبات از ثبات های داخل register file خوانده شده و در خروجی register file قرار گیرد را برابر بیت های ۰ تا ۴ (regno) درخواست DM قرار دهیم. در نتیجه برای این ورودی register file، یک multiplexer قرار می دهیم که اگر درخواست DM، خواندن مقدار ثباتی از ثبات های register file بود، این ورودی توسط درخواست DM تعیین شود و در نتیجه در خروجی register file مقدار ثباتی که DM قصد دانستن آن را دارد، قرار گیرد. در آخر، خروجی register file را به باس debugger\_port متصل می کنیم تا مقدار ثبات که در خروجی register file قرار دارد، از طریق باس debugger\_port به DM منتقل شود:



شکل (۳-۱۸) تغییرات data path پردازنده جهت خواندن از register file توسط Debug Module (DM) [18]

بنابراین مراحل که در کنترلر پردازنده طی می شود تا به این درخواست DM پاسخ داده شود به ترتیب زیر است:



شکل (۳-۱۹) تغییرات کنترلر پردازنده جهت پاسخ به درخواست های Debug Module (DM) [19]

۱. ابتدا در مرحله اولیه (fetch) قرار داریم. اگر سیگنال `core_halt` فعال شود، متوجه می شویم که قرار است DM بعدا درخواستی را به پردازنده ارسال کند، در نتیجه در صورتی که `core_halt_request` فعال شود، به

مرحله `waitForDebuggerRequest` پرش می کنیم. اگر علاوه بر این سیگنال، `debug_request` هم فعال شده باشد، متوجه می شویم که در سیکل بعدی DM قرار است درخواست خود را بر روی `debugger_port` قرار دهد، در نتیجه در این صورت، به مرحله `loadDebuggerCommand` پرش می کنیم.

۲. در مرحله `waitForDebuggerRequest` منتظر می شویم تا سیگنال `debug_request` فعال شود تا پس از آن به مرحله `loadDebuggerCommand` پرش کنیم. در صورتی که در این مرحله سیگنال `core_halt_request` غیر فعال شود، متوجه می شویم که DM دیگر قصد متوقف (pause) کردن پردازنده را ندارد و اجرا پردازنده باید از جایی که متوقف شده بود ادامه یابد، در نتیجه در این صورت، به مرحله اولیه (fetch) پرش می کنیم تا دستورات بعدی از حافظه خوانده و `fetch` شود و اجرا دستورات، مطابق با مرحله‌ای که در کنترلر پردازنده است، ادامه یابد.

۳. در مرحله `loadDebuggerCommand`، با فعال کردن سیگنال `load_debugger_command`، درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. پس از این مرحله، به مرحله `loadDebuggerCommandDataArgument` پرش می کنیم.

۴. در مرحله `loadDebuggerCommandDataArgument`، با فعال کردن سیگنال `load_debugger_command_data_argument`، آرگومان داده درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. پس از این مرحله، به مرحله `loadDebuggerCommandAddressArgument` پرش می کنیم.

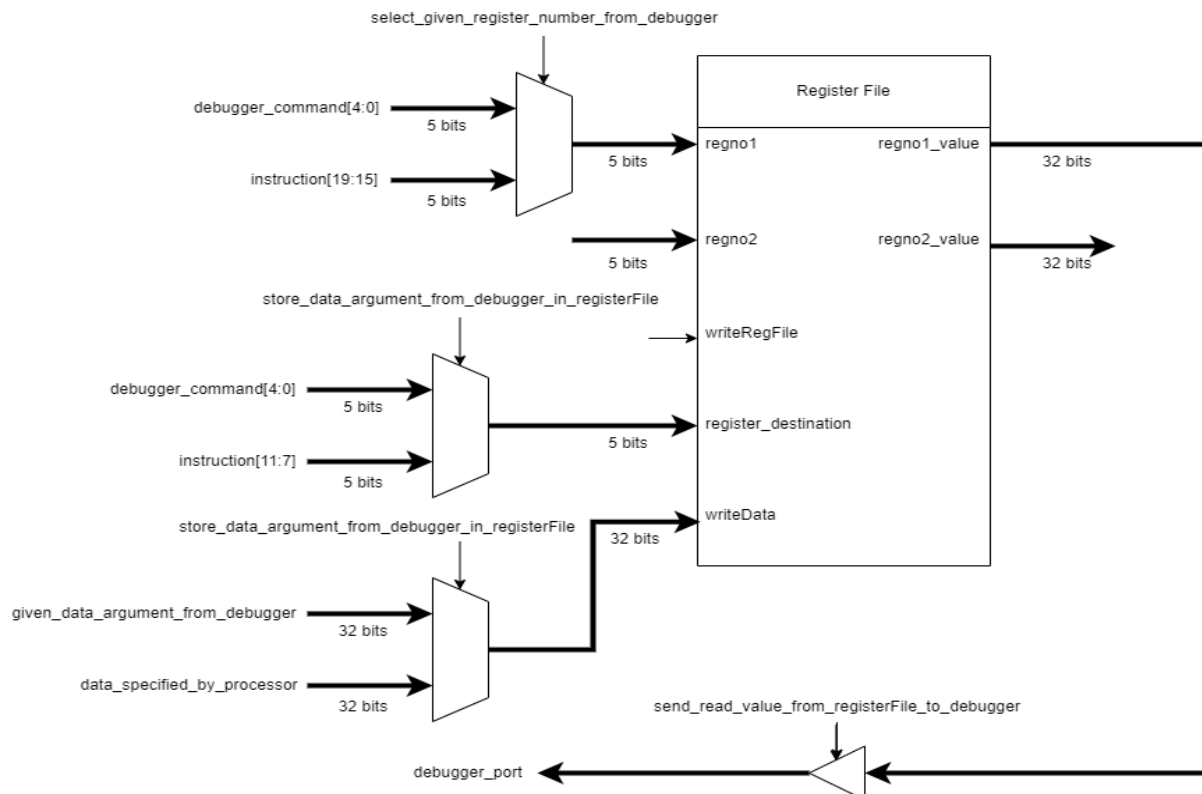
۵. در مرحله `loadDebuggerCommandAddressArgument`، با فعال کردن سیگنال `load_debugger_command_address_argument`، آرگومان آدرس درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. همچنین، با فعال کردن سیگنال `core_busy`، به DM اعلام کنیم که درخواست و آرگومان های آن کاملاً دریافت شده و پردازنده مشغول بررسی درخواست است. پس از این مرحله، به مرحله `decodeDebuggerCommand` پرش می کنیم.

۶. در مرحله `decodeDebuggerCommand`، بر اساس سیگنال هایی که از ماژول `debugger_command_decoder` به پردازنده وارد شده و تعیین کننده نوع درخواست است، به مرحله ای که آن درخواست را اجرا می کند پرش می کنیم. در این صورت، اگر درخواست DM، `read_register_request_from_debugger` باشد، برای پاسخ به آن به مرحله `sendRegisterValueToDebugger` پرش می کنیم.

۷. در مرحله `sendRegisterValueToDebugger`، با فعال شدن سیگنال `select_given_register_number_from_debugger`، مقدار ثابتی که DM قصد دانستن مقدار آن را دارد، در خروجی `file register` قرار می گیرد و سپس با فعال شدن سیگنال `send_read_value_from_registerFile_to_debugger`، مقدار ثابت بر روی باس `debugger_port` قرار می گیرد و همزمان سیگنال `core_busy` غیرفعال می شود تا DM متوجه اتمام پردازش درخواست خود شده و مقدار ثابت را از روی باس `debugger_port` بردارد. پس از این مرحله پاسخ به درخواست DM پایان یافته و پردازنده به مرحله اولیه (fetch) پرش می کند.

۲. اگر درخواست DM، نوشتن داده به ثباتی از حافظه داخلی پردازنده (register file) باشد:

برای اینکه بتوان در ثباتی از ثبات های register file، داده ای نوشت، ابتدا اینکه کدام ثبات مد نظر است با ورودی ۵ بیتی register\_destination تعیین می شود و سپس اینکه چه داده ای قرار است در این ثبات نوشته شود توسط ورودی ۳۲ بیتی writeData مشخص می گردد. بنابراین کافیت ورودی register\_destination را برابر بیت های ۰ تا ۴ (regno) درخواست DM قرار دهیم تا داده موجود در writeData در ثباتی که DM قصد نوشتن به آن را دارد، ذخیره شود. در نتیجه برای این ورودی register file، یک multiplexer قرار می دهیم که اگر درخواست DM، نوشتن داده به ثباتی از ثبات های register file بود، این ورودی توسط درخواست DM تعیین شود. سپس ورودی writeData را برابر مقدار ثبات Register\_Command\_Data\_Argument\_Register که در ماژول debugger\_command\_decoder ذخیره شده است، قرار می دهیم تا داده ای را که DM قصد نوشتن آن به register file دارد در register file ذخیره شود. بنابراین برای این ورودی register file، یک multiplexer قرار می دهیم که اگر درخواست DM، نوشتن داده ای به ثباتی از ثبات های register file بود، این ورودی توسط آرگومان داده درخواست DM تعیین شود تا آرگومان داده درخواست DM در register file ذخیره شود:



شکل (۳-۲۰) تغییرات data path پردازنده جهت نوشتن به register file توسط (DM) [20]

بنابراین مراحل که در کنترلر پردازنده طی می شود تا به این درخواست DM پاسخ داده شود به ترتیب زیر است:

۱. ابتدا در مرحله اولیه (fetch) قرار داریم. اگر سیگنال `core_halt` فعال شود، متوجه می شویم که قرار است DM بعداً درخواستی را به پردازنده ارسال کند، در نتیجه در صورتی که `core_halt_request` فعال شود، به مرحله `waitForDebuggerRequest` پرش می کنیم. اگر علاوه بر این سیگنال، سیگنال `debug_request` هم فعال شده باشد، متوجه می شویم که در سیکل بعدی DM قرار است درخواست خود را بر روی باس `debugger_port` قرار دهد، در نتیجه در این صورت، به مرحله `loadDebuggerCommand` پرش می کنیم.
۲. در مرحله `waitForDebuggerRequest` منتظر می شویم تا سیگنال `debug_request` فعال شود تا پس از آن به مرحله `loadDebuggerCommand` پرش کنیم. در صورتی که در این مرحله سیگنال `core_halt_request` غیر فعال شود، متوجه می شویم که DM دیگر قصد متوقف (pause) کردن پردازنده را ندارد و اجرا پردازنده باید از جایی که متوقف شده بود ادامه یابد، در نتیجه در این صورت، به مرحله اولیه (fetch) پرش می کنیم تا دستورات بعدی از حافظه خوانده و `fetch` شود و اجرا دستورات، مطابق با مراحل که در کنترلر پردازنده است، ادامه یابد.
۳. در مرحله `loadDebuggerCommand`، با فعال کردن سیگنال `load_debugger_command`، درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. پس از این مرحله، به مرحله `loadDebuggerCommandDataArgument` پرش می کنیم.
۴. در مرحله `loadDebuggerCommandDataArgument`، با فعال کردن سیگنال `load_debugger_command_data_argument`، آرگومان داده درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. پس از این مرحله، به مرحله `loadDebuggerCommandAddressArgument` پرش می کنیم.
۵. در مرحله `loadDebuggerCommandAddressArgument`، با فعال کردن سیگنال `load_debugger_command_address_argument`، آرگومان آدرس درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. همچنین، با فعال کردن سیگنال `core_busy`، به DM اعلام کنیم که درخواست و آرگومان های آن کاملاً دریافت شده و پردازنده مشغول بررسی درخواست است. پس از این مرحله، به مرحله `decodeDebuggerCommand` پرش می کنیم.
۶. در مرحله `decodeDebuggerCommand`، بر اساس سیگنال هایی که از ماژول

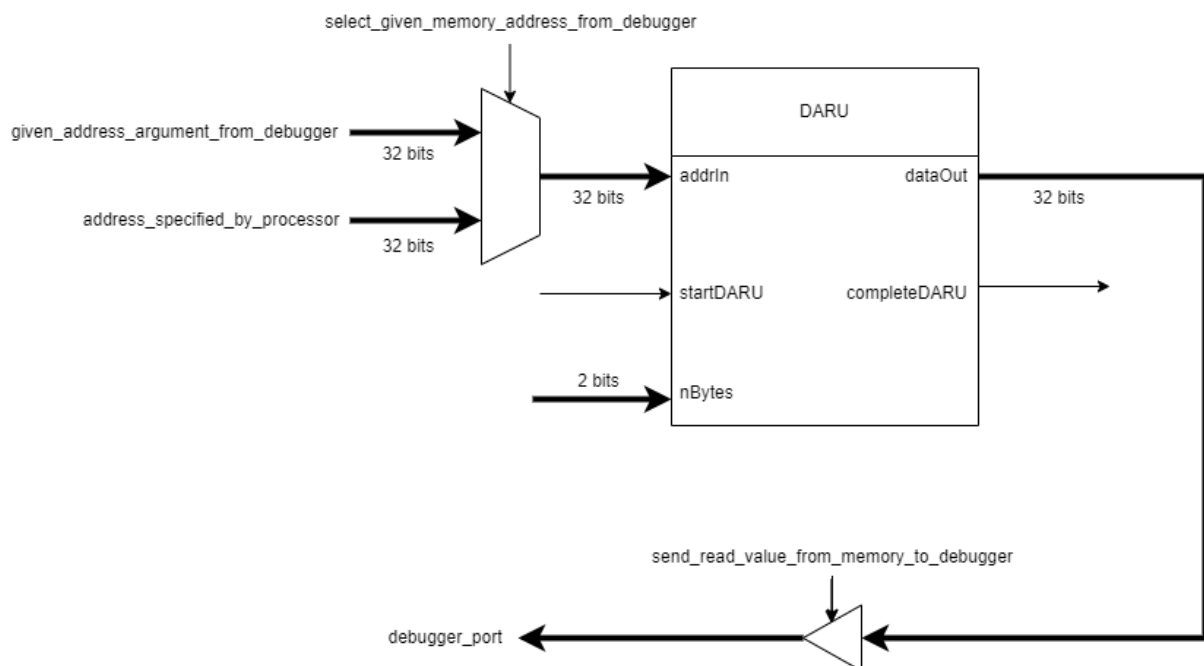
debugger\_command\_decoder به پردازنده وارد شده و تعیین کننده نوع درخواست است، به مرحله ای که آن درخواست را اجرا می کند پرش می کنیم. در این صورت، اگر درخواست DM، write\_register\_request\_from\_debugger باشد، برای پاسخ به آن به مرحله storeDataInRegisterFile پرش می کنیم.

۷. در مرحله storeDataInRegisterFile، ابتدا سیگنال writeRegFile را فعال می کنیم تا به register file اعلام کنیم که قصد ذخیره کردن داده در آن را داریم. سپس با فعال شدن سیگنال store\_data\_argument\_from\_debugger\_in\_registerFile، در ورودی register\_destination بیت های ۰ تا ۴ (regno) درخواست DM قرار می گیرد تا داده موجود در ورودی writeData در ثباتی که DM قصد نوشتن به آن را دارد، ذخیره شود و همزمان با فعال شدن این سیگنال، در ورودی writeData، داده ای را که DM قصد نوشتن آن به ثباتی از register file دارد، قرار گرفته و در نتیجه آرگومان داده درخواست DM در ثباتی که DM قصد نوشتن به آن را دارد، ذخیره می شود. در آخر سیگنال core\_busy غیرفعال می شود تا DM متوجه اتمام پردازش درخواست خود شود. پس از این مرحله پاسخ به درخواست DM پایان یافته و پردازنده به مرحله اولیه (fetch) پرش می کند.



۳. اگر درخواست DM، خواندن مقدار ذخیره شده در خانه ای از حافظه خارجی باشد:

برای اینکه بتوان مقدار ذخیره شده در خانه ای از حافظه خارجی را خواند، از واحد DARU که در پردازنده قرار دارد و برای خواندن از حافظه خارجی از آن استفاده می شود استفاده می کنیم. این واحد که در بخش ساختار پردازنده RISC-V (بخش ۲-۲-۳) توضیح داده شد، برای خواندن، نیاز دارد تا بداند از کدام آدرس از حافظه خارجی بخواند، بنابراین کافیسست ورودی `addrIn`، که بیانگر آدرسی است که باید مقدار آن سطر از حافظه، خوانده و `load` شود را برابر آرگومان آدرسی که در ثبت `Debugger_Command_Address_Argument_Register` در `_____` ثاب شده است. `debugger_command_decoder` هست، قرار دهیم. در نتیجه برای این ورودی `DARU`، یک `multiplexer` قرار می دهیم که اگر درخواست DM، خواندن از حافظه خارجی بود، این ورودی توسط آرگومان آدرس درخواست DM تعیین شود. در آخر، خروجی `DARU` را به `debugger_port` متصل می کنیم تا مقدار خوانده شده از حافظه خارجی که در خروجی `DARU` قرار دارد، از طریق `debugger_port` به DM منتقل شود:



شکل (۲۱-۳) تغییرات data path پردازنده جهت خواندن از حافظه خارجی توسط (DM) Debug Module [21]

بنابراین مراحل که در کنترلر پردازنده طی می شود تا به این درخواست DM پاسخ داده شود به ترتیب زیر است:

۱. ابتدا در مرحله اولیه (fetch) قرار داریم. اگر سیگنال `core_halt` فعال شود، متوجه می شویم که قرار است DM بعداً درخواستی را به پردازنده ارسال کند، در نتیجه در صورتی که `core_halt_request` فعال شود، به مرحله `waitForDebuggerRequest` پرش می کنیم. اگر علاوه بر این سیگنال، سیگنال `debug_request` هم فعال شده باشد، متوجه می شویم که در سیکل بعدی DM قرار است درخواست خود را بر روی بوس `debugger_port` قرار دهد، در نتیجه در این صورت، به مرحله `loadDebuggerCommand` پرش می کنیم.
۲. در مرحله `waitForDebuggerRequest` منتظر می شویم تا سیگنال `debug_request` فعال شود تا پس از آن به مرحله `loadDebuggerCommand` پرش کنیم. در صورتی که در این مرحله سیگنال `core_halt_request` غیر فعال شود، متوجه می شویم که DM دیگر قصد متوقف (pause) کردن پردازنده را ندارد و اجرا پردازنده باید از جایی که متوقف شده بود ادامه یابد، در نتیجه در این صورت، به مرحله اولیه (fetch) پرش می کنیم تا دستورات بعدی از حافظه خوانده و fetch شود و اجرا دستورات، مطابق با مراحل که در کنترلر پردازنده است، ادامه یابد.
۳. در مرحله `loadDebuggerCommand` با فعال کردن سیگنال `load_debugger_command`، درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. پس از این مرحله، به مرحله `loadDebuggerCommandDataArgument` پرش می کنیم.
۴. در مرحله `loadDebuggerCommandDataArgument` با فعال کردن سیگنال `load_debugger_command_data_argument`، آرگومان داده درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. پس از این مرحله، به مرحله `loadDebuggerCommandAddressArgument` پرش می کنیم.
۵. در مرحله `loadDebuggerCommandAddressArgument` با فعال کردن سیگنال `load_debugger_command_address_argument`، آرگومان آدرس درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. همچنین، با فعال کردن سیگنال `core_busy`، به DM اعلام کنیم که درخواست و آرگومان های آن کاملاً دریافت شده و پردازنده مشغول بررسی درخواست است. پس از این مرحله، به مرحله `decodeDebuggerCommand` پرش می کنیم.
۶. در مرحله `decodeDebuggerCommand`، بر اساس سیگنال هایی که از ماژول

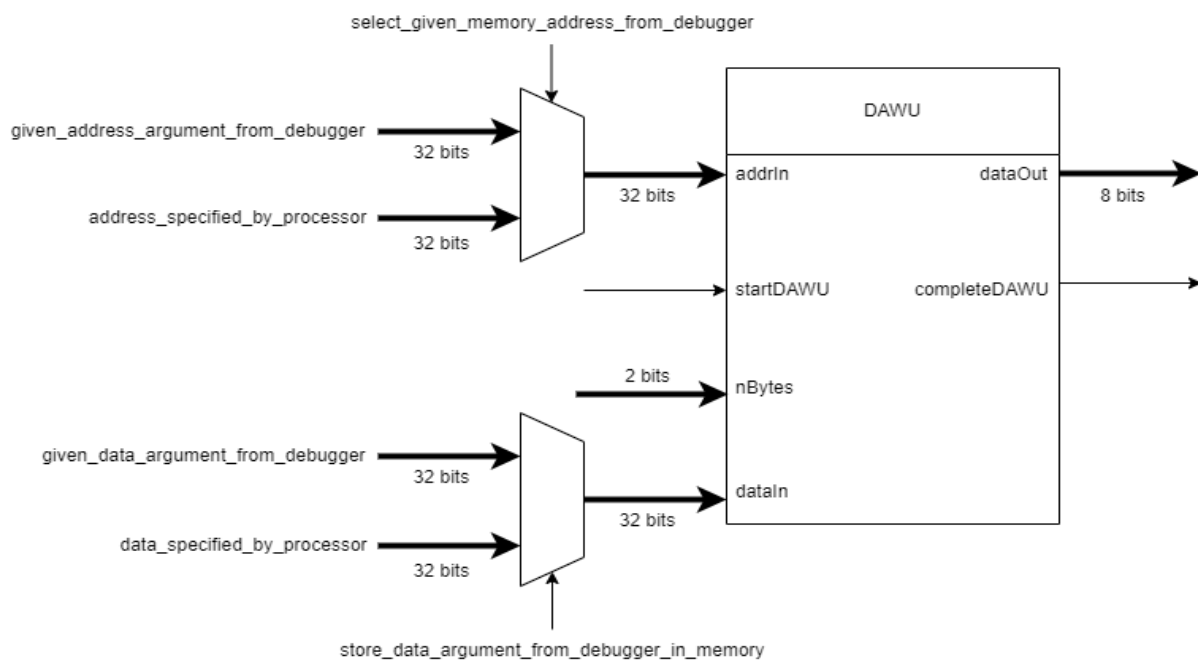
debugger\_command\_decoder به پردازنده وارد شده و تعیین کننده نوع درخواست است، به مرحله ای که آن درخواست را اجرا می کند پرسش می کنیم. در این صورت، اگر درخواست DM، read\_memory\_request\_from\_debugger باشد، برای پاسخ به آن به مرحله loadFromMemory پرسش می کنیم.

۷. در مرحله loadFromMemory، ابتدا با فعال کردن سیگنال select\_given\_memory\_address\_from\_debugger، در ورودی addrIn مازول DARU، آرگومان آدرس درخواست DM که در مازول debugger\_command\_decoder در Debugger\_Command\_Address\_Register هست، قرار می گیرد تا DARU، مقدار ذخیره شده در سطری از حافظه را بخواند، که DM قصد خواندن آن را دارد. سپس برای اینکه کل محتوای داده ۳۲ بیتی، که در یک سطر از حافظه ذخیره شده است، خوانده و load شود، مقدار ورودی nBytes را برابر ۱۱ باینری (۳ در مبنا ۱۰) قرار می دهیم تا DARU از حافظه خارجی به تعداد ۴ بایت با شروع از آدرس addrIn خوانده و در نتیجه کل محتوای سطری از حافظه که داده ۳۲ بیتی در آن قرار دارد، خوانده شود. پس از این، با فعال کردن سیگنال startDARU، فرآیند خواندن از حافظه آغاز می شود. پس از ۴ سیکل با فعال شدن سیگنال completeDARU، فرآیند خواندن از حافظه خاتمه یافته و محتوای سطری از حافظه که DM قصد خواندن آن را دارد، در خروجی DARU قرار می گیرد. در آخر پس از این مرحله به مرحله sendMemoryValueToDebugger پرسش می کنیم.

۸. در مرحله sendMemoryValueToDebugger، با فعال کردن سیگنال send\_read\_value\_from\_memory\_to\_debugger، مقدار خوانده شده از حافظه خارجی که در خروجی DARU قرار دارد، از طریق باس debugger\_port به DM منتقل شده و در نتیجه محتوای سطری از حافظه خارجی که DM قصد خواندن آن را داشت، به DM منتقل می شود. در آخر سیگنال core\_busy غیرفعال می شود تا DM متوجه اتمام پردازش درخواست خود شود. پس از این مرحله پاسخ به درخواست DM پایان یافته و پردازنده به مرحله اولیه (fetch) پرسش می کند.

۴. اگر درخواست DM، نوشتن داده به خانه ای از حافظه خارجی باشد:

برای اینکه بتوان داده ای در خانه ای از حافظه خارجی ذخیره کرد، از واحد DAWU که در پردازنده قرار دارد و برای نوشتن به حافظه خارجی از آن استفاده می شود استفاده می کنیم. این واحد که در بخش ساختار پردازنده RISC-V (بخش ۲-۲-۳) توضیح داده شد، برای نوشتن، نیاز دارد تا بدانند به کدام آدرس از حافظه خارجی بنویسد، بنابراین کافیسیت ورودی `addrIn`، که بیانگر آدرسی است که باید به آن سطر از حافظه، مقداری نوشته و `store` شود را برابر با آرگومان آدرسی که در `debugger_command_decoder` که در `Debugger_Command_Address_Argument_Register` ذخیره شده است، قرار دهیم. در نتیجه برای این ورودی DAWU، یک multiplexer قرار می دهیم که اگر درخواست DM، نوشتن داده به حافظه خارجی بود، این ورودی توسط آرگومان آدرس درخواست DM تعیین شود. سپس ورودی `dataIn` را برابر با مقدار ثبات `Debugger_Command_Data_Argument_Register` که در multiplexer قرار می دهیم تا داده ای را که DM قصد نوشتن آن به حافظه خارجی دارد، در حافظه خارجی ذخیره شود. بنابراین برای این ورودی DAWU، یک multiplexer قرار می دهیم که اگر درخواست DM، نوشتن داده به خانه ای از حافظه خارجی بود، این ورودی توسط آرگومان داده درخواست DM تعیین شود تا آرگومان داده درخواست DM در حافظه خارجی ذخیره شود:



شکل (۳-۲۲) تغییرات data path پردازنده جهت نوشتن به حافظه خارجی توسط (DM) Debug Module [22]

بنابراین مراحل که در کنترلر پردازنده طی می شود تا به این درخواست DM پاسخ داده شود به ترتیب زیر است:

۱. ابتدا در مرحله اولیه (fetch) قرار داریم. اگر سیگنال `core_halt` فعال شود، متوجه می شویم که قرار است DM بعداً درخواستی را به پردازنده ارسال کند، در نتیجه در صورتی که `core_halt_request` فعال شود، به مرحله `waitForDebuggerRequest` پرش می کنیم. اگر علاوه بر این سیگنال، سیگنال `debug_request` هم فعال شده باشد، متوجه می شویم که در سیکل بعدی DM قرار است درخواست خود را بر روی باس `debugger_port` قرار دهد، در نتیجه در این صورت، به مرحله `loadDebuggerCommand` پرش می کنیم.
۲. در مرحله `waitForDebuggerRequest` منتظر می شویم تا سیگنال `debug_request` فعال شود تا پس از آن به مرحله `loadDebuggerCommand` پرش کنیم. در صورتی که در این مرحله سیگنال `core_halt_request` غیر فعال شود، متوجه می شویم که DM دیگر قصد متوقف (pause) کردن پردازنده را ندارد و اجرا پردازنده باید از جایی که متوقف شده بود ادامه یابد، در نتیجه در این صورت، به مرحله اولیه (fetch) پرش می کنیم تا دستورات بعدی از حافظه خوانده و `fetch` شود و اجرا دستورات، مطابق با مراحل که در کنترلر پردازنده است، ادامه یابد.
۳. در مرحله `loadDebuggerCommand` با فعال کردن سیگنال `load_debugger_command`، درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. پس از این مرحله، به مرحله `loadDebuggerCommandDataArgument` پرش می کنیم.
۴. در مرحله `loadDebuggerCommandDataArgument` با فعال کردن سیگنال `load_debugger_command_data_argument`، آرگومان داده درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. پس از این مرحله، به مرحله `loadDebuggerCommandAddressArgument` پرش می کنیم.
۵. در مرحله `loadDebuggerCommandAddressArgument` با فعال کردن سیگنال `load_debugger_command_address_argument`، آرگومان آدرس درخواست DM در ثبات داخلی `debugger_command_decoder` ذخیره می شود. همچنین، با فعال کردن سیگنال `core_busy`، به DM اعلام کنیم که درخواست و آرگومان های آن کاملاً دریافت شده و پردازنده مشغول بررسی درخواست است. پس از این مرحله، به مرحله `decodeDebuggerCommand` پرش می کنیم.
۶. در مرحله `decodeDebuggerCommand`، بر اساس سیگنال هایی که از ماژول

debugger\_command\_decoder به پردازنده وارد شده و تعیین کننده نوع درخواست است، به مرحله ای که آن درخواست را اجرا می کند پرش می کنیم. در این صورت، اگر درخواست DM، write\_memory\_request\_from\_debugger باشد، برای پاسخ به آن به مرحله storeDataInMemory پرش می کنیم.

۷. در مرحله storeDataInMemory ابتدا با فعال کردن سیگنال select\_given\_memory\_address\_from\_debugger، در ورودی addrIn مازول DAWU، آرگومان آدرس درخواست DM که در مازول debugger\_command\_decoder در Debugger\_Command\_Address\_Argument\_Register هست، قرار می گیرد تا DAWU، در سطری از حافظه بنویسد که DM قصد نوشتن به آن را دارد. همزمان با فعال شدن سیگنال store\_data\_argument\_from\_debugger\_in\_memory، در ورودی dataIn، داده ای که DM قصد نوشتن آن به حافظه خارجی دارد، قرار گرفته تا آرگومان داده درخواست DM در حافظه ذخیره شود. سپس برای اینکه کل محتوای داده ۳۲ بیتی یا ۴ بیتی که قرار است DM در حافظه بنویسد، در یک سطر از حافظه ذخیره شده و store شود، مقدار ورودی nBytes را برابر ۱۱ باینری (۳ در مبنا ۱۰) قرار می دهیم تا DAWU به حافظه خارجی به تعداد ۴ بایت با شروع از آدرس addrIn نوشته و در نتیجه کل ۴ بایت محتوای آرگومان داده درخواست DM به سطری از حافظه که آدرس آن از addrIn شروع می شود، نوشته شود. پس از این، با فعال کردن سیگنال startDAWU، فرآیند نوشتن به حافظه آغاز می شود. پس از ۴ سیکل با فعال شدن سیگنال completeDAWU، فرآیند نوشتن به حافظه خاتمه یافته و آرگومان داده درخواست DM در سطری از حافظه که DM قصد نوشتن به آن را دارد، ذخیره و نوشته می شود و همزمان با فعال شدن این سیگنال، سیگنال core\_busy نیز غیرفعال می شود تا DM متوجه اتمام پردازش درخواست خود شود. پس از این مرحله پاسخ به درخواست DM پایان یافته و پردازنده به مرحله اولیه (fetch) پرش می کند.

### ۳-۳- ابزارهای مورد نیاز برای طراحی Debugger

برای طراحی Debugger، از زبان توصیف سخت افزار Verilog استفاده می کنیم تا در محیط Verilog بتوانیم برای پردازنده RISC-V، Debugger پیاده سازی کرده و آن را تست و بیازماییم.

### ۳-۴- معیار ارزیابی<sup>۱</sup>

همان طور که در فصل ۲ در بخش Debugger گفته شد، Debugger باید بتواند:

۱. اجرا برنامه را در هر قسمتی از برنامه، متوقف (pause) کند.
۲. در حین اینکه برنامه متوقف شده است، مقادیر ذخیره شده در حافظه خارجی و ثبات های حافظه داخلی پردازنده را خوانده و به برنامه نویس نمایش دهد.
۳. در حین اینکه برنامه متوقف شده است، مقدار مشخصی که از سوی برنامه نویس تعیین شده است را در حافظه خارجی یا ثبات های حافظه داخلی پردازنده ذخیره کند.
۴. اجرا برنامه را، از جایی که متوقف (pause) شده بود، ادامه داده و به اصطلاح resume کند.

در نتیجه برای ارزیابی Debugger طراحی شده از معیار های زیر استفاده می کنیم:

۱. با ارسال درخواست core\_halt\_request به Debugger، اجرای برنامه در پردازنده متوقف (pause) شده و با غیر فعال کردن این سیگنال، اجرا برنامه در پردازنده از جایی که متوقف شده بود ادامه یابد.
۲. در حین اینکه سیگنال core\_halt\_request فعال است و اجرا برنامه در پردازنده متوقف شده است، با ارسال درخواست خواندن از register file و آرجومان های آن به Debugger، پس از چند سیکل با غیر فعال شدن سیگنال debugger\_busy، در خروجی debugger\_result، محتوای ثباتی از ثبات های register file که در درخواست ارسالی محتوای آن درخواست شده بود، قرار گیرد.
۳. در حین اینکه سیگنال core\_halt\_request فعال است و اجرا برنامه در پردازنده متوقف شده است، با ارسال درخواست نوشتن به register file و آرجومان های آن به Debugger، پس از چند سیکل با غیر

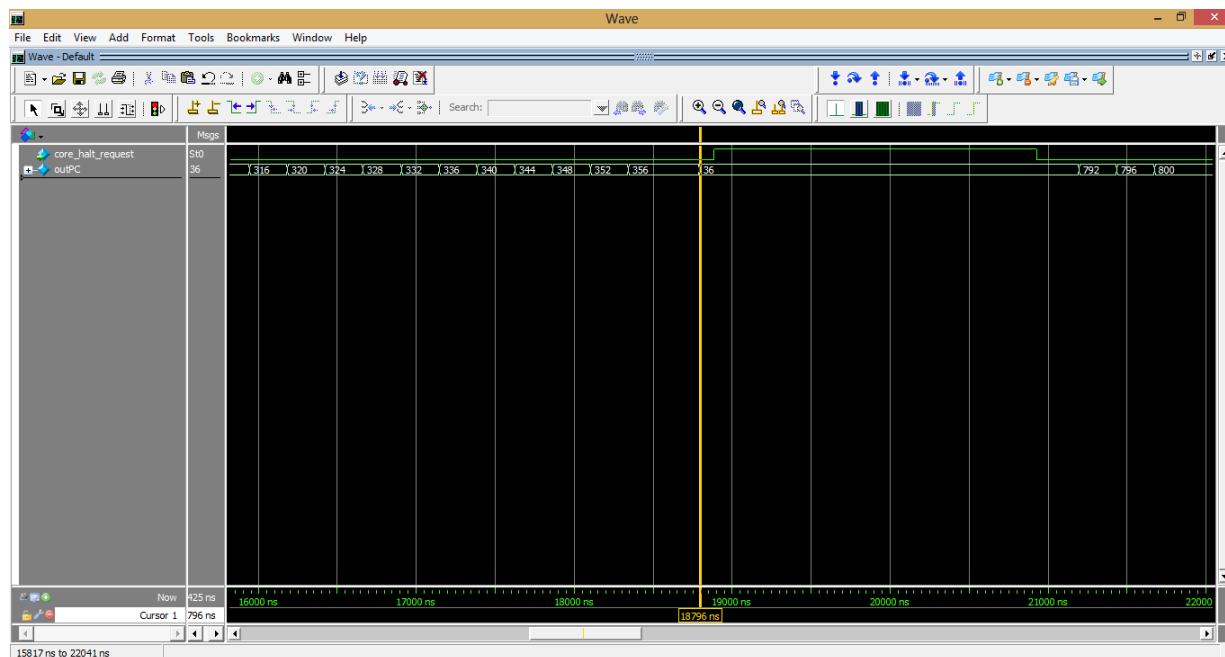
<sup>1</sup> Evaluation metric

- فعال شدن سیگنال `debugger_busy`، در ثباتی از ثبات های `register file` که در درخواست ارسالی نوشتن به آن درخواست شده بود، محتوایی که در درخواست ارسالی بود، نوشته و ذخیره شود.
۴. در حین اینکه سیگنال `core_halt_request` فعال است و اجرا برنامه در پردازنده متوقف شده است، با ارسال درخواست خواندن از حافظه خارجی و آرگومان های آن به `Debugger`، پس از چند سیکل با غیر فعال شدن سیگنال `debugger_busy`، در خروجی `debugger_result`، محتوای سطری از حافظه خارجی که در درخواست ارسالی محتوای آن درخواست شده بود، قرار گیرد.
۵. در حین اینکه سیگنال `core_halt_request` فعال است و اجرا برنامه در پردازنده متوقف شده است، با ارسال درخواست نوشتن به حافظه خارجی و آرگومان های آن به `Debugger`، پس از چند سیکل با غیر فعال شدن سیگنال `debugger_busy`، در سطری از حافظه خارجی که در درخواست ارسالی نوشتن به آن درخواست شده بود، محتوایی که در درخواست ارسالی بود، نوشته و ذخیره شود.



### ۳-۵- نتایج بدست آمده از طراحی Debugger

متوقف کردن اجرا برنامه در پردازنده با ارسال درخواست `core_halt_request` به Debugger، و ادامه یافتن اجرا برنامه با غیر فعال کردن این سیگنال:

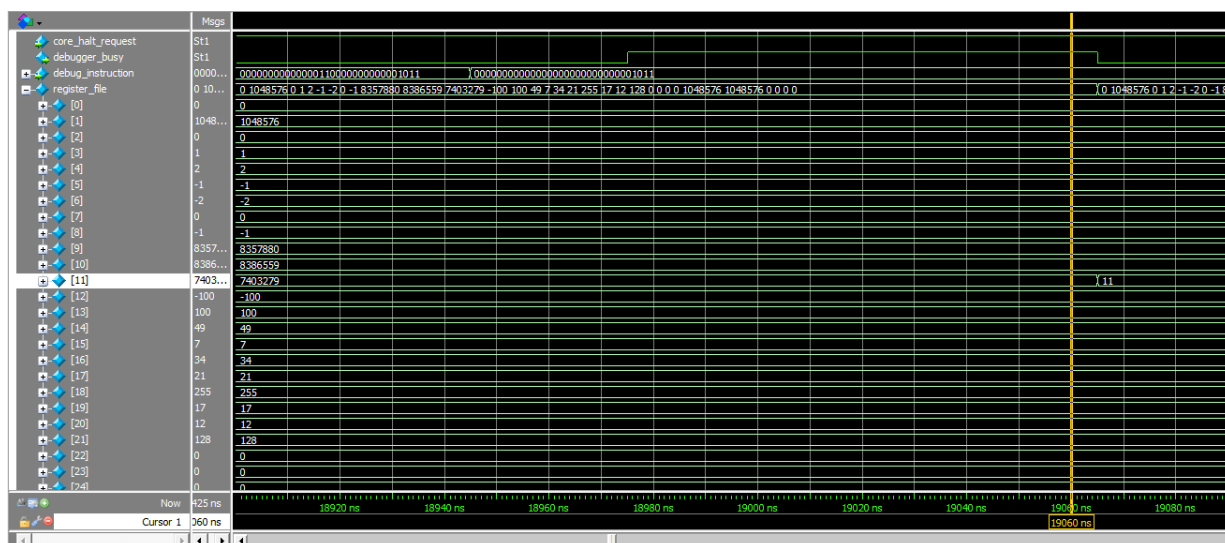


شکل (۳-۲۳) متوقف شدن اجرا برنامه با ارسال درخواست `core_halt_request` به Debugger [23]

همان طور که در تصویر فوق مشاهده می شود، با فعال کردن سیگنال `core_halt_request`، اجرا برنامه با ثابت ماندن مقدار Program Counter که مقدار آن در ثبات `outPC` در تصویر فوق قابل مشاهده است، متوقف و pause شده و در آخر با غیر فعال کردن این سیگنال، اجرا برنامه با افزایش PC، ادامه یافته است که نشان از درستی عملکرد Debugger است و در نتیجه Debugger معیار ارزیابی شماره ۱ که در بخش قبل (۳-۴) گفته شد را پاس می کند.



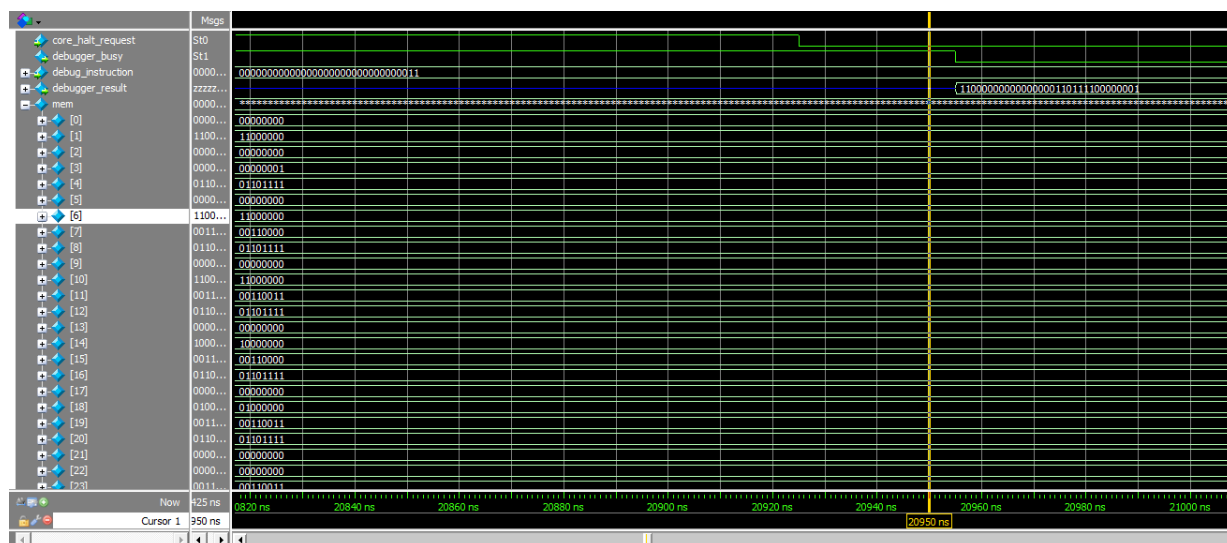
نوشتن به register file به ارسال درخواست نوشتن به register file به Debugger:



شکل (۳-۲۵) نوشتن به register file به ارسال درخواست نوشتن به register file به Debugger [25]

همان طور که در تصویر فوق مشاهده می شود، در حین اینکه سیگنال core\_halt\_request فعال است، ابتدا درخواست نوشتن به register file بر روی باس debug\_instruction قرار می گیرد که ۵ بیت انتهایی آن که بیانگر آدرس ثابتی است که می خواهیم در آن بنویسیم برابر 01011 یا ۱۱ است تا آرگومان داده درخواست که در ادامه بر روی باس debug\_instruction قرار گرفته و مقدار آن برابر ۱۱ در مبنای ۱۰ است، در ثبات ۱۱ از register file نوشته شود. مشاهده می شود که با غیر فعال شدن سیگنال debug\_busy، همان طور که انتظار میرفت، در ثبات ۱۱ از register file، مقدار ۱۱ نوشته شده که نشان از درستی عملکرد Debugger بوده و در نتیجه Debugger معیار ارزیابی شماره ۳ که در بخش قبل (۳-۴) گفته شد را پاس کرده است.

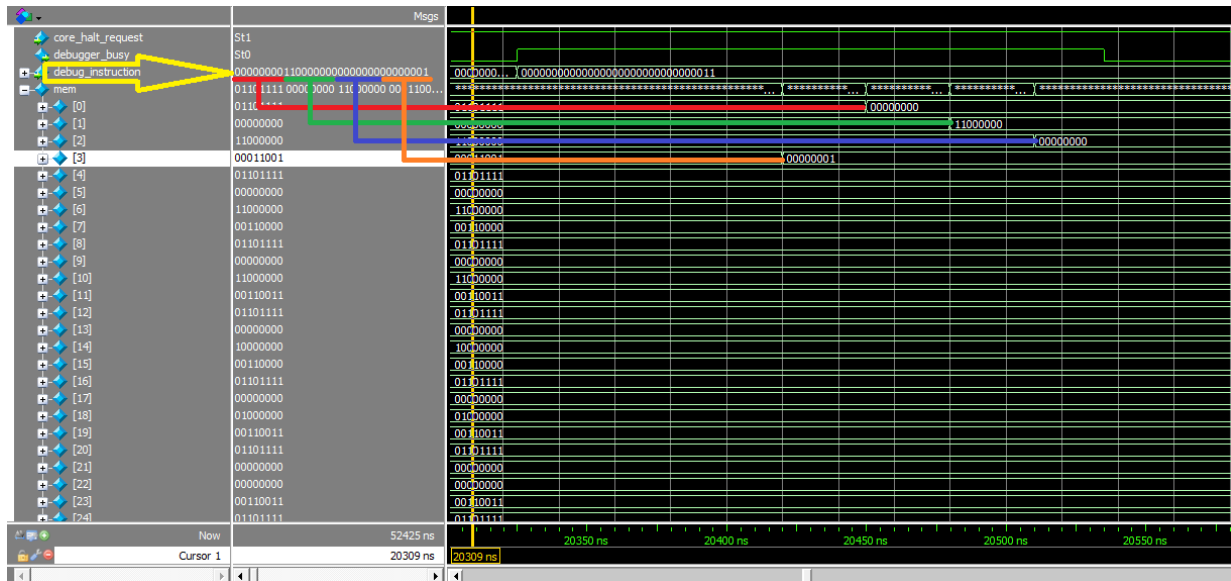
خواندن از حافظه خارجی با ارسال درخواست خواندن از حافظه خارجی به Debugger:



شکل (۳-۲۶) خواندن از حافظه خارجی با ارسال درخواست خواندن از حافظه خارجی به Debugger [26]

همان طور که در تصویر فوق مشاهده می شود، در حین اینکه سیگنال `core_halt_request` فعال است، پس از قرار گرفتن درخواست خواندن از حافظه خارجی بر روی باس `debug_instrucion`، آنگومان آدرس بر روی این باس قرار می گیرد که مقدار آن باتوجه به تصویر فوق برابر ۳ است تا محتوای سطری از حافظه که با آدرس ۳ شروع می شود، خوانده شده و بر روی باس `debugger_result` قرار گیرد. مشاهده می شود که با غیر فعال شدن سیگنال `debugger_busy`، همان طور که انتظار میرفت، محتوای سطری از حافظه که با آدرس ۳ شروع می شود که به ترتیب معادل محتوای خانه های ۶ تا ۳ از حافظه است، بر روی باس `debugger_result` قرار گرفته که نشان از درستی عملکرد Debugger بوده و در نتیجه Debugger معیار ارزیابی شماره ۴ که در بخش قبل (۳-۴) گفته شد را پاس کرده است.

نوشتن به حافظه خارجی با ارسال درخواست نوشتن به حافظه خارجی به Debugger:



شکل (۳-۲۷) نوشتن به حافظه خارجی با ارسال درخواست نوشتن به حافظه خارجی به Debugger [27]

همان طور که در تصویر فوق مشاهده می شود، در حین اینکه سیگنال `core_halt_request` فعال است، پس از قرار گرفتن درخواست نوشتن به حافظه خارجی بر روی باس `debug_instrucion`، آرجومان آدرس بر روی این باس قرار می گیرد که مقدار آن باتوجه به تصویر فوق برابر ۳ است تا محتوای آرجومان داده درخواست که قبل از آرجومان آدرس بر روی باس `debug_instruction` قرار گرفته و محتوای آن در تصویر فوق با فلش زرد رنگ مشخص شده است در سطری از حافظه که با آدرس ۳ شروع می شود، نوشته شود. مشاهده می شود که با غیر فعال شدن سیگنال `debugger_busy`، همان طور که انتظار میرفت، در سطری از حافظه که با آدرس ۳ شروع می شود، محتوای آرجومان داده درخواست به ترتیب از سمت چپ به راست در خانه های ۰ تا ۳ از حافظه نوشته شده است، که نشان از درستی عملکرد Debugger بوده و درنتیجه Debugger معیار ارزیابی شماره ۵ که در بخش قبل (۳-۴) گفته شد را پاس کرده است.

### ۳-۶- تحلیل نتایج

بر اساس نتایجی که در بخش قبل دیدیم، به این نتیجه می‌رسیم که Debugger طراحی شده مطابق با انتظارات ما عمل کرده و تمامی معیارهایی که برای ارزیابی آن مشخص کردیم را به درستی پاس کرده است. پس می‌توان گفت که Debugger به درستی طراحی و پیاده‌سازی شده است.

### ۳-۷- خلاصه و جمع‌بندی

فصل سوم به طور عمده در برگیرنده‌ی طراحی Debugger و جزئیات آن بود که در آخر، نتایج حاصل از Debugger پیاده‌سازی شده در محیط Verilog را مشاهده کردیم و دیدیم که Debugger طراحی شده به درستی کار می‌کند. در فصل بعد به جزئیات پیاده‌سازی Debugger در محیط Verilog می‌پردازیم.

## فصل ۴

### پیاده سازی Debugger

---

پس از طراحی های انجام شده در فصل قبل، در این فصل به جزئیات پیاده سازی Debugger در محیط Verilog می پردازیم.

#### ۴-۱- مقدمه

در این فصل Debugger را با استفاده از زبان توصیف سخت افزار Verilog پیاده سازی کرده و در محیط Verilog به پردازنده RISC-V، تغییرات لازم را اعمال می کنیم.

#### ۴-۲- نحوه پیاده سازی

در این بخش بر اساس طراحی صورت گرفته در فصل ۳ به جزئیات پیاده سازی Debugger و تغییرات اعمالی به پردازنده RISC-V در محیط Verilog می پردازیم.



## پیاده سازی Data Path ماژول (DM) Debug Module (DM) بر اساس بخش ۳-۲-۱ (ساختار Debugger):

```
module DataPath #(parameter size = 32) (
    input clk, rst, core_reset_request, core_halt_request, load_debug_result, load_debug_command, load_debug_command_data_argument, load_debug_command_address_argument, send_debug_cor
    output core_reset, core_halt,
    input [size-1:0] debug_instruction,
    inout [size-1:0] debugger_port,
    output [size-1:0] debugger_result
);

    wire [size-1:0] debugger_command, debugger_command_data_argument, debugger_command_address_argument;

    assign core_reset = core_reset_request;
    assign core_halt = core_halt_request;

    register #(size) Debugger_Command_Register (
        .clk(clk),
        .rst(rst),
        .ldR(load_debug_command),
        .in(debug_instruction),
        .out(debugger_command)
    );

    register #(size) Debugger_Command_Data_Argument_Register (
        .clk(clk),
        .rst(rst),
        .ldR(load_debug_command_data_argument),
        .in(debug_instruction),
        .out(debugger_command_data_argument)
    );

    register #(size) Debugger_Command_Address_Argument_Register (
        .clk(clk),
        .rst(rst),
        .ldR(load_debug_command_address_argument),
        .in(debug_instruction),
        .out(debugger_command_address_argument)
    );

    register #(size) Debugger_Result_Register (
        .clk(clk),
        .rst(rst),
        .ldR(load_debug_result),
        .in(debugger_port),
        .out(debugger_result)
    );

    tristate #(size) Debugger_Command_Tristate (
        .in(debugger_command),
        .enable(send_debug_command),
        .out(debugger_port)
    );

    tristate #(size) Debugger_Command_Data_Argument_Tristate (
        .in(debugger_command_data_argument),
        .enable(send_debug_command_data_argument),
        .out(debugger_port)
    );

    tristate #(size) Debugger_Command_Address_Argument_Tristate (
        .in(debugger_command_address_argument),
        .enable(send_debug_command_address_argument),
        .out(debugger_port)
    );

endmodule
```

شکل (۲۸-۲) پیاده سازی Data Path ماژول (DM) Debug Module (DM) در Verilog [28]

### پیاده سازی کنترلر ماژول (DM) Debug Module بر اساس بخش ۳-۲-۱ (ساختار Debugger):

```
module Controller (
    input clk, rst, debug_request, core_busy,
    output reg load_debug_result, load_debug_command, load_debug_command_data_argument, load_debug_command_address_argument, send_debug_command, send_debug_command_data_argument, send_debug_command_address_argument,
    output reg debugger_busy
);

    reg[2:0] ns, ps;
    parameter[2:0] waiting_for_debug_request = 3'b000, sending_debug_command_to_core = 3'b001, sending_debug_command_data_argument_to_core = 3'b010, sending_debug_command_address_argument_to_core = 3'b011;
    always @(ps, debug_request, core_busy) begin
        (load_debug_result, load_debug_command, load_debug_command_data_argument, load_debug_command_address_argument, send_debug_command, send_debug_command_data_argument, send_debug_command_address_argument) = case (ps)
            waiting_for_debug_request: begin ns = debug_request ? sending_debug_command_to_core : waiting_for_debug_request; load_debug_command = debug_request; debugger_busy = 0; end
            sending_debug_command_to_core: begin ns = sending_debug_command_data_argument_to_core; send_debug_command = 1; load_debug_command_data_argument = 1; end
            sending_debug_command_data_argument_to_core: begin ns = sending_debug_command_address_argument_to_core; send_debug_command_data_argument = 1; load_debug_command_address_argument = 1; end
            sending_debug_command_address_argument_to_core: begin ns = waiting_for_core_response; send_debug_command_address_argument = 1; debugger_busy = 1; end
            waiting_for_core_response: begin ns = core_busy ? waiting_for_core_response : waiting_for_debug_request; load_debug_result = ~core_busy; end
            default: ns = waiting_for_debug_request;
        endcase
    end

    always @(posedge clk, posedge rst) begin
        if(rst)
            ps <= waiting_for_debug_request;
        else
            ps <= ns;
        end
    end
endmodule
```

شکل (۲-۲۹) پیاده سازی کنترلر ماژول (DM) Debug Module در Verilog [29]

### پیاده سازی ماژول debugger\_command\_decoder بر اساس بخش ۳-۲-۲ (پاسخ دادن پردازنده به درخواست Debugger):

```
module debugger_command_decoder #(parameter size = 32) (
    input clk,
    input rst,
    input [size-1:0] debugger_port,
    input load_debugger_command,
    input load_debugger_command_data_argument,
    input load_debugger_command_address_argument,
    output read_register_request_from_debugger,
    output write_register_request_from_debugger,
    output read_memory_request_from_debugger,
    output write_memory_request_from_debugger,
    output [4:0] given_register_number_from_debugger,
    output [size-1:0] given_data_argument_from_debugger,
    output [size-1:0] given_address_argument_from_debugger
);

    wire [size-1:0] debugger_command, debugger_command_data_argument, debugger_command_address_argument;

    aifb_register #(size) Debugger_Command_Register (
        .clk(clk),
        .rst(rst),
        .zero(0),
        .ldR(load_debugger_command),
        .in(debugger_port),
        .out(debugger_command)
    );

    aifb_register #(size) Debugger_Command_Data_Argument_Register (
        .clk(clk),
        .rst(rst),
        .zero(0),
        .ldR(load_debugger_command_data_argument),
        .in(debugger_port),
        .out(debugger_command_data_argument)
    );

    aifb_register #(size) Debugger_Command_Address_Argument_Register (
        .clk(clk),
        .rst(rst),
        .zero(0),
        .ldR(load_debugger_command_address_argument),
        .in(debugger_port),
        .out(debugger_command_address_argument)
    );

    assign read_register_request_from_debugger = ~debugger_command[16] & debugger_command[17];
    assign write_register_request_from_debugger = debugger_command[16] & debugger_command[17];
    assign read_memory_request_from_debugger = ~debugger_command[16] & debugger_command[25];
    assign write_memory_request_from_debugger = debugger_command[16] & debugger_command[25];
    assign given_register_number_from_debugger = debugger_command[4:0];
    assign given_data_argument_from_debugger = debugger_command_data_argument;
    assign given_address_argument_from_debugger = debugger_command_address_argument;
endmodule
```

شکل (۲-۳۰) پیاده سازی ماژول Debugger Command Decoder در Verilog [30]

تغییرات اعمالی به Data Path پردازنده RISC-V بر اساس بخش ۳-۲-۲ (پاسخ دادن پردازنده به درخواست  
:(Debugger

```
debugger_command_decoder #(size) Debugger_Command_Decoder(
    .clk(clk),
    .rst(rst),
    .debugger_port(debugger_port),
    .load_debugger_command(load_debugger_command),
    .load_debugger_command_data_argument(load_debugger_command_data_argument),
    .load_debugger_command_address_argument(load_debugger_command_address_argument),
    .read_register_request_from_debugger(read_register_request_from_debugger),
    .write_register_request_from_debugger(write_register_request_from_debugger),
    .read_memory_request_from_debugger(read_memory_request_from_debugger),
    .write_memory_request_from_debugger(write_memory_request_from_debugger),
    .given_register_number_from_debugger(given_register_number_from_debugger),
    .given_data_argument_from_debugger(given_data_argument_from_debugger),
    .given_address_argument_from_debugger(given_address_argument_from_debugger)
);

assign debugger_port = (send_read_value_from_registerFile_to_debugger) ? p1 :
    (send_read_value_from_memory_to_debugger) ? dataDARU :
    (size){1'b0};

assign rsl = (select_given_register_number_from_debugger) ? given_register_number_from_debugger : inst[19:15];
assign rd = (store_data_argument_from_debugger_in_registerFile) ? given_register_number_from_debugger : inst[11:7];

aftab_registerFile #(size) registerFile (
    .clk(clk),
    .rst(rst),
    .setZero(setZero/* & ~selfFFPRegFile*/),
    .setOne(setOne/* & ~selfFFPRegFile*/),
    .rsl(rsl),
    .rs2(inst[24:20]),
    .rd(rd),
    .writeData(writeData),
    .writeRegFile(writeRegFile/* & ~selfFFPRegFile*/),
    .p1(p1/*_Integer*/),
    .p2(p2/*_Integer*/)
);

assign writeData = (selInc4PC == 1'b1) ? inc4PC :
    (selBSU == 1'b1) ? bsuResult :
    (selLLU == 1'b1) ? llusResult :
    (selASU == 1'b1) ? asuResult :
    (selIAU == 1'b1) ? auuResult :
    (selIARU == 1'b1) ? adjIARU :
    (selCSR == 1'b1) ? outCSR :
    (store_data_argument_from_debugger_in_registerFile == 1'b1) ? given_data_argument_from_debugger :
    (selPWriteData == 1'b1) ? p1 :
    (size){1'b0};

assign memAddrIn = (select_given_memory_address_from_debugger) ? given_address_argument_from_debugger : addrIn;
assign dawuDataIn = (store_data_argument_from_debugger_in_memory) ? given_data_argument_from_debugger : dataDAWU;

aftab_MEM_DAWU #(size) DAWU (
    .addrIn(memAddrIn),
    .dataIn(dawuDataIn),
    .nBytes(nBytes),
    .startDAWU(startDAWU),
    .memReady(memReady),
    .clk(clk),
    .rst(rst),
    .checkMisalignedDAWU(checkMisalignedDAWU),
    .addrOut(memAddrDAWU),
    .dataOut(memDataOut),
    .storeMisalignedFlag(),
    .completeDAWU(completeDAWU),
    .writeMem(writeMem)
);

aftab_MEM_DARU #(size) DARU (
    .clk(clk),
    .rst(rst),
    .startDARU(startDARU),
    .memReady(memReady),
    .dataInstrBar(dataInstrBar),
    .checkMisalignedDARU(checkMisalignedDARU),
    .addrIn(memAddrIn),
    .memData(memDataIn),
    .nBytes(nBytes),
    .completeDARU(completeDARU),
    .readMem(readMem),
    .instrMisalignedFlag(instrMisalignedFlag),
    .loadMisalignedFlag(),
    .dataOut(dataDARU),
    .addrOut(memAddrDARU)
);
```

شکل (۳۱-۲) تغییرات اعمالی به Data Path پردازنده RISC-V در Verilog [31]

تغییرات اعمالی به کنترلر پردازنده RISC-V بر اساس بخش ۳-۲-۲ (پاسخ دادن پردازنده به درخواست Debugger):

```

`define waitForDebuggerRequest 6'b100110
`define loadDebuggerCommand 6'b100111
`define loadDebuggerCommandDataArgument 6'b101000
`define loadDebuggerCommandAddressArgument 6'b101001
`define decodeDebuggerCommand 6'b101010
`define sendRegisterValueToDebugger 6'b101011
`define storeDataInRegisterFile 6'b101100
`define loadFromMemory 6'b101101
`define sendMemoryValueToDebugger 6'b101110
`define storeDataInMemory 6'b101111

case(p_state)
  `fetch: begin
    if(exceptionRaise || interruptRaise)
      n_state <= `checkDelegation;
    else if(instrMisalignedOut)
      n_state <= `fetch;
    else if(core_halt_request & ~debug_request)
      n_state <= `waitForDebuggerRequest;
    else if(core_halt_request & debug_request)
      n_state <= `loadDebuggerCommand;
    else
      n_state <= `getInstr;
    end

  `loadDebuggerCommand:
    n_state <= `loadDebuggerCommandDataArgument;
  `loadDebuggerCommandDataArgument:
    n_state <= `loadDebuggerCommandAddressArgument;
  `loadDebuggerCommandAddressArgument:
    n_state <= `decodeDebuggerCommand;
  `decodeDebuggerCommand: begin
    if(read_register_request_from_debugger)
      n_state <= `sendRegisterValueToDebugger;
    else if(write_register_request_from_debugger)
      n_state <= `storeDataInRegisterFile;
    else if(read_memory_request_from_debugger)
      n_state <= `loadFromMemory;
    else if(write_memory_request_from_debugger)
      n_state <= `storeDataInMemory;
    end
  `sendRegisterValueToDebugger:
    n_state <= `fetch;
  `storeDataInRegisterFile:
    n_state <= `fetch;
  `loadFromMemory: begin
    if (completeDARU)
      n_state <= `sendMemoryValueToDebugger;
    else
      n_state <= `loadFromMemory;
    end
  `sendMemoryValueToDebugger:
    n_state <= `fetch;
  `storeDataInMemory: begin
    if (completeDAWU)
      n_state <= `fetch;
    else
      n_state <= `storeDataInMemory;
    end
end

.

.

.

`loadDebuggerCommand: begin
  load_debugger_command <= 1'b1;
  core_busy <= 1'b1;
end
`loadDebuggerCommandDataArgument: begin
  load_debugger_command_data_argument <= 1'b1;
end
`loadDebuggerCommandAddressArgument: begin
  load_debugger_command_address_argument <= 1'b1;
end
`sendRegisterValueToDebugger: begin
  select_given_register_number_from_debugger <= 1'b1;
  send_read_value_from_registerFile_to_debugger <= 1'b1;
  core_busy <= 1'b0;
end
`storeDataInRegisterFile: begin
  writeRegFile <= 1'b1;
  store_data_argument_from_debugger_in_registerFile <= 1'b1;
  core_busy <= 1'b0;
end
`loadFromMemory: begin
  startDARU <= 1'b1;
  nBytes <= 2'b11;
  select_given_memory_address_from_debugger <= 1'b1;
end
`sendMemoryValueToDebugger: begin
  send_read_value_from_memory_to_debugger <= 1'b1;
  core_busy <= 1'b0;
end
`storeDataInMemory: begin
  startDAWU <= 1'b1;
  nBytes <= 2'b11;
  select_given_memory_address_from_debugger <= 1'b1;
  store_data_argument_from_debugger_in_memory <= 1'b1;
  core_busy <= ~completeDAWU;
end
end

```

شکل (۳۲-۲) تغییرات اعمالی به کنترلر پردازنده RISC-V در Verilog [32]

## ۴-۳- خلاصه و جمع بندی

در این فصل به جزئیات پیاده سازی پروژه خود و بخش های مختلف آن با نشان دادن تصاویری از کدهای نوشته شده در محیط Verilog پرداختیم.

## فصل ۵

### جمع‌بندی و نتیجه‌گیری

---

## ۵-۱- جمع‌بندی

در این تحقیق در گام نخست ما با مفهوم پردازنده آشنا شدیم. سپس به مفهوم RISC-V پرداخته و ساختار پردازنده AFTAB که نوعی پیاده‌سازی برای پردازنده RISC-V است را تشریح کردیم. در ادامه ابتدا چیهستی Debugger و وظایف آن را توضیح داده و پس از آن نحوه تعامل Debugger با پردازنده RISC-V را ارائه دادیم. سپس به طراحی Debugger پرداخته و ساختار Debug Module و تغییرات اعمال شده به پردازنده RISC-V را توضیح دادیم. در آخر Debugger طراحی شده را در محیط Verilog پیاده‌سازی کرده و آن را در حضور پردازنده RISC-V تغییر یافته تست کردیم و مطابق با معیارهای ارزیابی خود، صحت کارایی آن را بررسی کرده و نتیجه گرفتیم که Debugger طراحی شده به درستی کار می‌کند.

## ۵-۲- نتیجه‌گیری

در این پروژه هدف نهایی ما این بود که بتوانیم بستری برای مشاهده روند اجرا برنامه فراهم کنیم که قابلیت مشاهده مقادیر ذخیره شده در پردازنده در هر قسمت از برنامه‌ای که در پردازنده اجرا می‌شود را داشته باشد و پس از طی شدن بخش‌های قبل که طی فصل‌های ۱ تا ۴ گفته شد، به این هدف رسیدیم.

## فصل ٦

مراجع

---



- [1] RISC-V External Debug Support Version 0.13.2
- [2] The RISC-V Instruction Set Manual (Volume I: Unprivileged ISA)
- [3] The RISC-V Instruction Set Manual (Volume II: Privileged Architecture)

**Abstract:**

One of the biggest problems that programmers face when programming is to find the cause of incorrect program output. In other words, when the output of the written code is wrong, several reasons can cause the incorrect output, which is difficult to find the cause of the incorrect output, due to the large number of possible assumptions. To facilitate this, there is a need for a tool that shows the process of program execution to the programmers and make it possible to observe the execution of each line of the code and the effect that each one has on the values of variables, etc. By this way, the cause of the program not working properly can be discovered faster.

Debugging is a tool that helps in the process of finding the cause of code failure (Debugging). The purpose of this project is to make it possible to observe the process of program execution in the processor, so that there will be the ability to debug and view the values stored in the processor and memory in every part of the program that is executed by the processor. In this case, identifying the source of the error in the program will be facilitated and accelerated.

To achieve this goal and implement it, we first get to know the structure and operation of Debugger. Then we design RTL and its hardware description in Verilog environment and we also apply changes to the structure of RISC-V processor. Finally, we test the Debugger in the presence of the processor programmed with a specific program. We will see that the designed Debugger is able to pause the execution of the program on the processor, read the values stored in the register file of the processor as well as the external memory and, if necessary, store a value in them, and after that resume the execution from where it paused.

**Keywords: debugging, RTL, Verilog, RISC-V, register file**



University of Tehran



College of Engineering

School of Electrical and Computer Engineering

## **Debugger Extension Mode Design for RISC-V Processor**

A thesis submitted to the Undergraduate Studies Office

In partial fulfillment of the requirements for

The degree of Bachelor in

Electrical Engineering

**By:**

**Mohammad Taghizadeh Givari**

**Supervisor:**

**Dr. Zeinalabedin Navabi Shirazi**