باسمه تعالی

پروژه درس معماری و سازمان کامپیوتر

نیمسال تحصیلی ۴۰۳۲

استاد درس: دکتر امیر خورسندی

دستیاران آموزشی: محمد مهدی اسدی، محمد رضائیان

مقدمه

هدف این پروژه، طراحی و پیاده‌سازی یک معماری و شبیه‌ساز کامپیوتری برای پردازنده‌ای مبتنی بر معماری RISC-V [1] است. به طور کلی معماری RISC-V یک معماری مجموعه دستورالعمل (ISA) استاندارد و باز است که بر اساس اصول طراحی سیستم‌های با مجموعه دستورالعمل کاهش یافته (RISC) توسعه یافته و امروزه مبنای بسیاری از پردازنده‌های جدید و کارآمد است.

RISC-V به دلیل ساختار ساده، قابل توسعه بودن، و رایگان بودن مجوز استفاده از آن، در سال‌های اخیر به یکی از محبوب‌ترین معماری‌ها در حوزه‌های آموزشی، تحقیقاتی و حتی صنعتی تبدیل شده است. این معماری با ارائه مجموعه‌ای از دستورالعمل‌های پایه، در کنار افزونه‌های اختیاری مانند M (ضرب و تقسیم)، F (اعداد اعشاری) و C (مجموعه دستورهای فشرده شده)، به طراحان اجازه می‌دهد معماری را با توجه به نیاز خود گسترش دهند. سادگی در طراحی سخت‌افزار، امکان سفارشی‌سازی، وجود مستندات رسمی و ابزارهای متن‌باز از جمله مزایای کلیدی RISC-V هستند. بنابراین ما هم یک نمونه ساده شده از آن را برای این پروژه آموزشی در نظر گرفته ایم.

بدیهی است که با توجه به محدودیت زمانی پروژه، امکان بررسی کامل تمام قابلیت‌های RISC-V وجود ندارد. بنابراین تنها بخشی از دستورات که در این فایل تعیین شده الزامی خواهند بود. البته علاوه بر دستورات ضروری، بخشی نیاز به صورت اختیاری و برای کسب نمره بیشتر مشخص شده‌اند. البته در صورتی که دستورات بخش ضروری پروژه به‌طور کامل پیاده‌سازی نشده باشند، هیچ‌گونه نمره‌ای به بخش‌های امتیازی پروژه تعلق نخواهد گرفت.

هم چنین دستیاران آموزشی در کنار شما خواهند بود تا با برگزاری جلسات توجیهی[2]، ارائه مطالب تکمیلی و پاسخ به سؤالات، روند پیشرفت پروژه را تسهیل کنند.

شرح کلی پروژه

بنابراین، گام های پروژه به سه بخش اصلی زیر تقسیم می شود. بدیهی است که کسب امتیاز پایه پروژه مشروط به صحت عملکرد همه این بخش‌ها خواهد بود. البته علاوه بر امتیاز پایه، امتیازهای تشویقی هم وجود دارد که در بخش‌های بعدی شرح داده خواهند شد.

---

[1] وب سایت رسمی RISC-V

[2] شرکت در جلسات عمومی اعلام شده از سوی دستیاران آموزشی درس برای دریافت امتیاز پروژه الزامی خواهد بود.

۱. طراحی معماری مبتنی بر RISC-V

در مرحله نخست، شما با بهره‌گیری از ابزارها و دانش یک معمار سیستم‌های کامپیوتری، موظف به طراحی یک معماری ساده و البته کارآمد مبتنی بر مجموعه‌دستورالعمل‌های تعیین شده RISC-V هستید. این معماری باید شامل بلوک دیاگرام کلی واحدهای عملیاتی، مسیر داده (Data Path) و واحد کنترل به همراه نحوه اتصال آن‌ها باشد.

هم چنین، با استفاده از زبان RTL اید به‌طور دقیق مشخص کنید که به ازای هر یک از دستورالعمل‌های تعیین شده در هر پالس ساعت ، چه ریزعملیات‌هایی در معماری پیشنهادی شما انجام می‌شود.

تمامی جزئیات طراحی، فرضیات اتخاذشده، نمودارهای مربوط به معماری و مسیر داده و توضیحات فنی مرتبط باید به‌صورت کامل مستند شده و در قالب فایلی با فرمت مناسب (PDF یا Markdown) در پوشه‌ی doc پروژه قرار گیرد.

۲. پیاده‌سازی اسمبلر

در مرحله دوم، انتظار می‌رود با استفاده از هر زبان برنامه‌نویس دلخواه، یک اسمبلر طراحی و پیاده‌سازی شود که یک فایل شامل کدهای اسمبلی با قالب RISC-V را به‌عنوان ورودی دریافت کرده و آن را به کد ماشین معادل ترجمه کند. خروجی این فرآیند باید به‌صورت یک فایل باینری با پسوند bin. ذخیره شود تا در مرحله‌ی بعد، به‌منظور راه‌اندازی فرایند اجرای برنامه در سخت‌افزار (یا شبیه‌ساز)، مورد استفاده قرار گیرد.

۳. هسته شبیه ساز

در مرحله نهایی، یک نرم‌افزار شبیه‌ساز طراحی و پیاده‌سازی خواهد شد که به‌صورت مرحله‌به‌مرحله (به ازای هر پالس ساعت)، رفتار سخت‌افزار RISC-V را در اجرای ریزدستورالعمل‌ها شبیه‌سازی می‌کند.

فرآیند کار این هسته به این صورت است که در ابتدا فایل باینری تولیدشده در مرحله‌ی دوم را به‌عنوان ورودی دریافت کرده و محتوای آن را در بلوک حافظه‌ی شبیه‌سازی‌شده بارگذاری می‌نماید. سپس، با آغاز چرخه‌ی اجرای دستورالعمل‌ها، از اولین دستور تا انتها را به‌صورت گام‌به‌گام، از طریق شبیه‌سازی ریزعملیات‌های مربوطه، اجرا می‌کند و جزییات آن (محتوای ثبات‌ها، حافظه، ورودی/خروجی و سایر اجزای سیستم) را با گرافیک مناسب به کاربر نمایش می‌دهد.

لازم به توجه است که با توجه به نرم‌افزاری بودن فرآیند شبیه‌سازی، باید تمهیداتی برای تقلید دقیق رفتار سخت‌افزار در نظر گرفته شود. به‌عنوان مثال، در سخت‌افزار ممکن است دو ثبات بتوانند در یک پالس ساعت مقدار خود را به‌صورت مستقیم مبادله کنند، در حالی‌که در پیاده‌سازی نرم‌افزاری این عملیات باید با استفاده از مکانیزم‌های واسطه انجام شود.

با توجه به این‌که این بخش ویترین نهایی پروژه محسوب می‌شود، سادگی استفاده از آن برای کاربر و نیز جذابیت بصری آن از اهمیت بالایی برخوردار است. همچنین، پیاده‌سازی قابلیت‌های تکمیلی مانند محیط توسعه کد (با امکان پیشنهاد خودکار کد)، ادغام با اسمبلر (با قابلیت نمایش خروجی و خطاهای احتمالی موجود در کد و ارائه پیشنهاد برای رفع آن‌ها) و نیز ارائه ابزارهای دیباگ (اجرای گام‌به‌گام، تعیین نقاط توقف و ...) می‌تواند به افزایش امتیاز پروژه کمک کند.

این بخش از پروژه را هم می‌توانید با استفاده از هر زبان برنامه نویسی و فریمورک GUI پیاده سازی کنید . اما بایستی داخل یک فایل ReadMe [۳] توضیحات روند بیلد و اجرای پروژه به صورت دقیق و کامل ذکر شود. به صورت اکید توصیه می‌گردد که کدهای خود را به صورت ماژولار پیاده سازی و اصول کدنویسی تمیز و خوانا[٤] را رعایت کنید :) هم‌چنین پیشنهاد می‌شود از ابزار های کنترل ورژن مثل گیت و گیت هاب برای بک آپ گرفتن از روند توسعه پروژه استفاده کنید.

گروه بندی

شما می‌توانید این پروژه را در قالب گروه حداقل دو و یا حداکثر چهار نفره انجام بدهید. البته برای گروه‌های چهار نفره لازم است که دستورات ممیز شناور را علاوه بر مجموعه دستورات پایه پیاده‌سازی کنند. توجه داشته باشید که نمره هر شخص بر اساس میزان فعالیت در پروژه مشخص خواهد شد. جزئیات گروه‌بندی ظرف حداکثر یک هفته از تاریخ اعلام پروژه (تا پایان روز پنجشنبه ۱ خردادماه) باید به دستیاران آموزشی درس اطلاع داده شود.

تحویل پروژه

شما تا پایان روز ۳ تیر ماه ۱۴۰۴ فرصت دارید که فایل های مورد نظر به شرح زیر را در قالب یك فایل فشرده که به صورت CA_Project_stdNum.zip نامگذاری شده در تکلیفی با عنوان پروژه نهایی درس درون سامانه ی یکتا بارگذاری کنید. این کار باید توسط تمامی اعضای گروه به صورت مجزا انجام پذیرد.

- تمامی فایل های کد و فایل های وابسته مربوط به بخش های مختلف پروژه، به تفکیک داخل پوشه‌ای به نام src قرار گیرد.
- خروجی اجرایی پروژه به همراه فایل نصب نیازمندی‌های احتمالی در پوشه‌ای به نام build قرار گیرد.
- یک فایل ReadMe شامل توضیحات متنی در رابطه با نحوه توسعه، چگونگی نصب و اجرای پروژه تهیه و ذخیره شود.
- یک یا چند فایل در برگیرنده توضیح دقیق معماری طراحی شده به همراه توضیح مختصر و RTL هر دستور که داخل پوشه doc قرار می گیرند.
- یک یا چند ویدیو (حداقل ۵ دقیقه و حداکثر ۱۵ دقیقه) جهت معرفی پروژه، نحوه کار و اجرای برنامه که داخل پوشه video دخیره می‌شوند.

ارائه پروژه

این پروژه یك ارائه نیز خواهد داشت که جزییات آن متعاقباً اعلام خواهد گردید (به احتمال زیاد به صورت حضوری D:).

---

[۳] چگونه یک ReadMe خوب برای پروژه بنویسیم (freecodecamp)

[٤] بهترین نکات مربوط به کدنویسی تمیز (clean code)

راه های ارتباطی با دستیاران آموزشی درس

شما می‌توانید سوالات و مشکلات خودت را داخل گروه تلگرامی درس مطرح کنید (لینک گروه تلگرام).
همچنین می‌تواند به صورت مستقیم با دستیاران آموزشی در ارتباط باشید:

- آقای محمد مهدی اسدی: @mm00asady
- آقای محمد رضاییان: @Mr_MRF

نکات مهم در طراحی معماری مبتنی بر RISC-V

همان‌طور که در بخش مقدمه توضیح داده شد، در مرحله‌ی اول پروژه، لازم است یک دیاگرام کلی از معماری پیشنهادی خود ترسیم کنید. در این بلوک دیاگرام صرفاً نمایش کلیات مربوط به بلوک‌های اصلی (مانند واحد کنترل، ALU، فایل ثبات‌ها، حافظه و مسیرهای داده) و نحوه ارتباط آن‌ها با یکدیگر کافی است. **دقت شود که معماری طراحی‌شده باید مبتنی بر مدل Von Neumannباشد، به این معنا که داده و دستورالعمل‌ها در یک فضای حافظه‌ی مشترک ذخیره می‌شوند. هم‌چنین به منظور کاهش هزینه‌ها از گذرگاه مشترک به منظور ایجاد ارتباط بین بخش‌های مختلف استفاده نمایید.** برای طراحی این دیاگرام می‌توانید از ابزارهای متنوع کمک بگیرید.[5]

در ادامه این مرحله برای هر دستورالعمل RISC-V که توسط معماری شما پشتیبانی می‌شود، باید مشخص کنید که اجرای آن در هر سیکل پالس ساعت چگونه انجام می‌شود. به‌عبارت دیگر، فازهای اجرای هر دستور (خواندن، رمزگشایی، اجرا، دسترسی به حافظه و نوشتن نتیجه) را برای معماری طراحی‌شده به‌صورت گام‌به‌گام مستند کنید.

نکات مهم در طراحی اسمبلر RISC-V

همان گونه که می‌دانید اسمبلر یک برنامه است که وظیفه دارد کد نوشته‌شده به زبان اسمبلی (یک زبان سطح پایین نمادین و نزدیک به سخت‌افزار) را به کد زبان ماشین (کدی در قالب صفر و یک که توسط پردازنده قابل فهم است) تبدیل کند.

در فرآیند اسمبل کردن، اسمبلر ابتدا دستورات اسمبلی را تجزیه (Parse) کرده و با توجه به جدول کدهای عملیاتی (Opcode Table) و فرمت دستورالعمل‌های پشتیبانی شده در سخت‌افزار، معادل دودویی (Binary) هر دستور را تولید می‌کند.

اسمبلر همچنین ممکن است خطاهای نحوی (Syntax Errors) یا معنایی (Semantic Errors) را در کد اسمبلی شناسایی و گزارش کند تا برنامه‌نویس آن‌ها را اصلاح کند.

مطابق آن چه در درس گفته شده، اسمبلر ها معمولاً در دو مرحله عمل می‌کنند. در ادامه، خلاصه ای از کل فرایند را با توضیح هر مرحله آورده شده:

---

[5] نرم افزار های پیشنهادی برای طراحی دیاگرام معماری: Logisim Evolution, Draw.io

۱. تحلیل اولیه و ساخت جدول نمادها: در این مرحله، اسمبلر کد را به لحاظ موارد زیر تحلیل می‌کند.

- تشخیص برچسب‌ها (Labels) و ثبت آن‌ها در Symbol Table
- شمارش اندازه‌ی دستورالعمل‌ها و داده‌ها برای تعیین موقعیت هر خط
- بررسی اصول نگارش دستورات (Syntax Check) برای اطمینان از صحت ساختار دستورات

۲. ترجمه و تولید کد ماشین: در این مرحله، اسمبلر از اطلاعات مرحله‌ی اول برای تولید کد استفاده می‌کند.

- ترجمه‌ی هر دستور به کد باینری معادل
- جایگزینی برچسب ها با آدرس واقعی با استفاده از Symbol Table
- ساخت فایل خروجی باینری با پسوند **`.bin`**

توجه داشته باشید که حتماً باید در داخل فایل **README** توضیح مختصری درباره روند عملکرد اسمبلر و مراحل آن ارائه دهید. این توضیحات باید شامل مراحل مختلف فرآیند اسمبل کردن، از جمله نحوه تحلیل دستورات، تبدیل به کد باینری و تولید فایل خروجی باشد.

نکات مهم در طراحی شبیه ساز

در مرحله آخر پروژه، لازم است یک نرم‌افزار شبیه‌ساز طراحی و پیاده‌سازی شود که بتواند با دریافت یک فایل باینری با پسوند `.bin`، رفتار سخت‌افزار را در پردازش دستورات شبیه‌سازی کند.
برای شروع، هنگامی که شبیه‌ساز راه‌اندازی می‌شود، مقدار تمامی ثبات‌ها صفر است و مقدار ثبات برنامه‌شمار (PC) به آدرس **`0x1000`** تنظیم می‌شود.
در ادامه، شبیه‌ساز باید فازهای مختلف اجرای هر دستور را به‌طور گام‌به‌گام شبیه‌سازی کند. این فازها شامل مراحل زیر خواهند بود:

۱. واکشی دستور: بارگذاری دستور از آدرس تعیین شده درون حافظه.
۲. رمزگشایی: تجزیه دستور برای استخراج اجزای مختلف آن شامل کد دستور، عملوندها و ...
۳. اجرا: انجام عملیات محاسباتی یا منطقی به صورت پالس به پالس.
۴. دسترسی به حافظه: اگر دستور نیاز به دسترسی به حافظه (خواندن یا نوشتن) داشته باشد، این مرحله انجام می‌شود.
۵. نوشتن نتیجه: نوشتن نتیجه‌ی عملیات در ثبات‌ها یا حافظه.

در طول زمان شبیه‌سازی، باید همواره وضعیت سیستم (شامل ثبات‌ها، حافظه و وضعیت پردازنده) در هر مرحله نمایش داده شود.
برای کنترل زمان درشبیه‌ساز، حداقل باید دو حالت پیاده‌سازی شود. حالت اول به صورت توقف‌ناپذیر که کل اجرا از ابتدا تا انتها با سرعت تعیین شده توسط کاربر پیش می رود. اما در حالت دوم با هر بار فشردن یک کلید مخصوص توسط کاربر تعداد مشخصی پالس ساعت تولید و اجرا به همان اندازه پیش خواهد رفت.

نمونه نرم‌افزارهای اسمبلر و شبیه‌ساز

برای دید گرفتن در رابطه با نحوه پیاده‌سازی یک اسمبلر و شبیه ساز، می‌توانید نسخه آماده شده برای کامپیوتر پایه مانو را بررسی کنید:

- mano simulator

هم‌چنین برای مشاهده نحوه کار با اسمبلر و شبیه‌ساز RISC-V، می‌توانید به وب‌سایت‌های زیر مراجعه کنید:

- ripes.me
- RISC-V Interpreter

نکات پایانی

در پایان توصیه می‌گردد در انجام پروژه به موارد زیر دقت کنید:

- مطالعه دقیق مستندات **RISC-V**: قبل از هر چیز، مروری بر مستندات رسمی ISA و افزونه‌های مورد نیاز (مثلاً بخش M برای ضرب و تقسیم) داشته باشید تا از دامنه پروژه و فرمت دستورالعمل‌ها مطمئن شوید. این اطلاعات در ادامه همین فایل، فایل‌های ضمیمه پروژه و نیز بر روی اینترنت قابل دسترس هستند.
- تنظیم ساختار پروژه: پیش از آغاز کدنویسی، ساختار فایل‌ها و پوشه‌ها (به‌ویژه doc، src و bin) را بر اساس استاندارد تیم مشخص کنید تا از بی‌نظمی در ایجاد فایل‌ها جلوگیری شود.
- تعریف محدوده کار: لیستی از دستورالعمل‌های اجباری و اختیاری پروژه تهیه کنید و با اعضا تیم روی آن توافق کنید تا از اتلاف وقت بر سر قابلیت‌های اضافی پیش از اتمام بخش‌های اصلی و پایه جلوگیری شود.
- تقسیم وظایف و زمان‌بندی: وظایف مختلف شامل طراحی معماری، نوشتن RTL، پیاده‌سازی اسمبلر، شبیه‌ساز، رابط کاربری و مستندسازی را بین اعضای تیم تقسیم کنید و جدول زمانی ساده‌ای برای هر فاز تعریف کنید. سعی کنید همه اعضا طبق این جدول کار را پیش ببرند تا در موعد مقرر کار به انتها برسد.
- توسعه تدریجی و تست مداوم: قابلیت‌ها را به بخش‌های کوچک مجزا تقسیم کنید (مثلاً بخش بارگذاری دستور، رمزگشایی، اجرای دستورات با قالب مشخص و ...) و پیش از تجمیع همه بخش‌ها برای هر قسمت یک تست واحد یا مثال ساده بنویسید تا از صحت عملکرد آن مطمئن شوید.
- مستندسازی هم‌زمان: هر تغییر در معماری یا کد را در اولین فرصت در فایل‌های doc مستندسازی کنید. تهیه تاریخچه تغییرات (Changelog) می‌تواند برای ثبت تصمیمات و فرضیات مفید باشد.
- استفاده از کنترل نسخه: با توجه به انجام گروه پروژه سعی نمایید از Git یا ابزار مشابه استفاده کنید و قبل از هر تغییر بزرگ، شاخه (branch) مجزا ایجاد و پس از آزمون، ادغام (merge) کنید.

- پشتیبانی از مکانیزم خطایابی: اگرچه پیاده‌سازی این بخش در اسمبلر و شبیه‌ساز اختیاری است اما انجام آن در صورت بروز مشکل، می‌تواند به افزایش سرعت در تشخیص محل خطا به خود شما کمک نماید.
- طراحی مجزای هسته شبیه‌ساز از رابط گرافیکی: پیشنهاد می‌شود که در ابتدا هسته شبیه‌ساز را در ساده‌ترین حالت و با ورودی/خروجی حداقلی (صرفاً متن) پیاده کنید و پس از تکمیل و اطمینان از صحت عملکرد آن، در ادامه GUI را به آن اضافه کنید.
- تعامل مستمر با دستیاران آموزشی: در صورت بروز مشکل با دستیار آموزشی در ارتباط باشید تا مطمئن شوید مسیر پیشرفت پروژه منطبق بر انتظارات است.
- بررسی مجدد این دستورالعمل: هر از گاهی فهرست قابلیت‌های اجباری (MVP) تعیین شده در این فایل را مرور کنید و از اتمام آن‌ها پیش از زمان مقرر مطمئن شوید همگی را به درستی انجام داده‌اید.

موفق باشید :)

کلیات معماری RISC-V

معماری RISC-V دارای طراحی مدولاری است که در نسخه حداقلی از بخش‌های پایه تشکیل شده و در نسخه‌های کامل‌تر افزونه‌های اختیاری به آن اضافه می‌شود. بر این اساس در این بخش مروری بر نسخه پایه و مجموعه دستورات پشتیبانی شده توسط آن که در انجام این پروژه ضروری هستند خواهیم داشت.

ثبات های پایه

معماری RISC-V شامل ۳۲ ثبات پایه مندرج در جدول زیر است:

General Purpose Registers

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

در جدول بالا ستون اول نام ثبات را که بر اساس شماره‌گذاری ترتیبی بین x0 تا x31 تعیین شده نشان می‌دهد، در ستون دوم، نام قراردادی مستعار ثبات ها که بر اساس عملکرد خاص آن‌ها تعیین شده آمده است. بدیهی است که در حین کدنویسی اسمبلی، این نام‌ها ساده‌تر و قابل‌فهم‌تر هستند. ستون سوم نیز وظیفه یا کاربرد خاص هر ثبات را مشخص می‌نماید. در ستون آخر مشخص شده است که هنگام فراخوانی تابع، چه کسی مسئول ذخیره مقدار فعلی هر ثبات است:

- **Caller**: تابع فراخوان (کدی که تابع را صدا می‌زند) مسئول ذخیره مقدار فعلی است.
- **Callee**: خود تابع فراخوانی‌شده باید مقدار را ذخیره کند.
- —: نیازی به ذخیره‌سازی نیست.

توضیحات زیر در رابطه با کارکرد خاص این ثبات‌ها حایز اهمیت هستند:
در معماری RISC-V، ثبات **x0** همواره مقدار صفر دارد و نوشتن در آن بی‌اثر است.
ثبات (**x1 (ra**: این ثبات برای نگهداری آدرس بازگشت از توابع استفاده می‌شود. به عبارت دیگر، پس از اجرای یک تابع، این ثبات آدرس بازگشتی را که پردازنده باید به آن بازگردد، ذخیره می‌کند.
ثبات (**x2 (sp**: این ثبات به‌عنوان اشاره‌گر پشته (Stack Pointer) استفاده می‌شود.
ثبات (**x3 (gp**: این ثبات به‌عنوان اشاره‌گر به داده‌های سراسری (Global Pointer) مورد استفاده قرار می‌گیرد. این داده‌ها معمولاً داده‌هایی هستند که در سراسر برنامه قابل دسترسی‌اند.
ثبات (**x4 (tp**: این ثبات که به‌عنوان **Thread Pointer** نیز شناخته می‌شود، برای نگهداری اطلاعات مربوط به نخ‌ها (threads) در برنامه‌های چندنخی استفاده می‌شود.
ثبات‌های **x5** تا **x7** و **x28** تا **x31**: این ثبات‌ها به‌عنوان ثبات‌های موقتی در نظر گرفته می‌شوند. این ثبات‌ها برای ذخیره مقادیر موقتی در محاسبات و برنامه‌نویسی استفاده می‌شوند و در نتیجه نیازی به حفظ مقدار آن‌ها پس از فراخوانی توابع نیست.
ثبات **x8**: این ثبات به‌عنوان اشاره‌گر فریم یا ثبات ذخیره‌شده در زمان استفاده از سیستم‌عامل استفاده می‌شود.
ثبات‌های **x10** تا **x17**: این ثبات‌ها برای ارسال آرگومان‌ها به توابع و برخی از آن‌ها برای دریافت مقادیر بازگشتی از آن‌ها استفاده می‌شوند.
ثبات‌های **x9** و **x18** تا **x27**: این ثبات‌ها به‌عنوان ثبات‌های ذخیره‌سازی پایدار برای حفظ مقادیر بین توابع در نظر گرفته شده‌اند. این ثبات‌ها معمولاً برای نگهداری مقادیری که باید در طول اجرای چندین تابع حفظ شوند، به‌کار می‌روند.

انواع قالب‌های دستورات پایه

در معماری RISC-V، تمامی دستورالعمل‌ها دارای طول ثابت ۳۲ بیت هستند و بر اساس نوع عملکرد به چند قالب اصلی تقسیم می‌شوند که هر قالب ساختار بیت‌بندی مشخصی دارد. وجود این قالب‌های استاندارد، کدگذاری ساده و گسترش‌پذیر را در معماری RISC-V ممکن می‌سازد.
این قالب‌ها شامل انواع R-type، I-type، S-type، B-type، U-type و J-type هستند که تفاوت آن‌ها توسط opcode مشخص می گردد و هر قالب برای دسته‌ای خاص از دستورالعمل‌ها به کار می‌رود. به عنوان مثال قالب R برای انجام عملیات محاسباتی بین دو ثبات، قالب I برای کار با عملوند بلافصل، قالب S برای دستورات ذخیره‌سازی، قالب B برای پرش‌های شرطی، قالب‌های U و J برای آدرس‌دهی‌های سطح بالا و پرش‌های غیرشرطی به کار می‌روند. قالب FENCE-type هم برای هماهنگی دستورات حافظه و جلوگیری از اجرای خارج از ترتیب طراحی شده‌اند که در اجرای موازی دستورات به کار می‌رود و پیاده سازی آن از اهداف این پروژه نیست.

31 27	26 25	24 20	19 15	14 12	11 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|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type
31 28	27 24	23 20	19 15	14 12	11 7	6 0	
fm	pred	succ	rs1	funct3	rd	opcode	FENCE

دستورالعمل‌های پایه محاسبات اعداد صحیح RV32I

این مجموعه شامل دستوراتی برای انجام عملیات محاسباتی پایه (مانند جمع، تفریق، و شیفت)، عملیات بارگذاری و ذخیره‌سازی داده در حافظه، دستورالعمل‌های پرش و انشعاب و برخی عملیات منطقی نظیر AND، OR و XOR بر روی اعداد صحیح می‌باشد. RV32I به عنوان پایه‌ای ترین مجموعه دستورات موجود در معماری RISC-V، بر روی سادگی و کارایی بالا تمرکز دارد. این مجموعه مبنای توسعه نسخه‌های پیشرفته‌تر RISC-V مانند RV64I یا RV128I نیز محسوب می‌شود. بخشی از این دستورات در جدول زیر قابل مشاهده است. در فایل ضمیمه می‌توانید لیست کامل این دستورات و عملکردهای آنها را مشاهده کنید.

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-ext
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	signed
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	unsigned
addi	ADD (Immediate)	I	0010011	0x0		rd = rs1 + imm	
xori	XOR (Immediate)	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR (Immediate)	I	0010011	0x6		rd = rs1 | imm	
andi	AND (Immediate)	I	0010011	0x7		rd = rs1 & imm	

همان گونه که مشاهده می‌شود در همه دستورات این جدول `opcode = 0110011` می‌باشد، یعنی دستور از نوع R-type است. در معماری RISC-V، دستوراتی که ساختار R-type دارند (مانند add، sub، and و ...) به دلیل استفاده از یک قالب یکسان، باید به کمک فیلدهای کمکی از یکدیگر متمایز شوند. این فیلدها عبارت‌اند از:

- **`funct3`**: برای شناسایی دسته‌ی عملیات (مثلاً جمع، and، or و ...)
- **`funct7`**: برای تمایز بین دستوراتی که `funct3` یکسان دارند (مثلاً `add` و `sub`)

در ادامه، دستورات پایه‌ای که پیاده‌سازی آن‌ها برای این پروژه الزامی هستند را بررسی خواهیم کرد.

- add: جمع دو مقدار موجود در ثبات‌های rs1 و rs2 و ذخیره نتیجه در ثبات rd.

funct7	rs2	rs1	funct3	rd	opcode
0000000	xxxxx	xxxxx	000	xxxxx	0110011

مثال: `add x5, x6, x7` → `x5 = x6 + x7`

- sub: کم کردن مقدار ثبات rs2 از مقدار ثبات rs1 و ذخیره نتیجه در ثبات rd.

funct7	rs2	rs1	funct3	rd	opcode
0100000	xxxxx	xxxxx	000	xxxxx	0110011

مثال: `sub x5, x6, x7` → `x5 = x6 - x7`

- xor: اجرای عملگر منطقی XOR بیت به بیت بین مقادیر rs1 و rs2 و ذخیره نتیجه در rd.

funct7	rs2	rs1	funct3	rd	opcode
0000000	xxxxx	xxxxx	100	xxxxx	0110011

مثال: `xor x5, x6, x7` → `x5 = x6 ^ x7`

- or: اجرای عملگر منطقی OR بیت به بیت بین rs1 و rs2 و ذخیره نتیجه در rd.

funct7	rs2	rs1	funct3	rd	opcode
0000000	xxxxx	xxxxx	110	xxxxx	0110011

مثال: `or x5, x6, x7` → `x5 = x6 | x7`

- and: اجرای عملگر منطقی AND بیت به بیت بین rs1 و rs2 و قرار دادن نتیجه در rd.

funct7	rs2	rs1	funct3	rd	opcode

0000000	xxxxx	xxxxx	111	xxxxx	0110011

مثال: `and x5, x6, x7` → `x5 = x6 & x7`

- sll: شیفت منطقی به چپ مقدار rs1 به اندازه مشخص‌شده توسط ۵ بیت کم ارزش rs2 که نتیجه در rd ذخیره می‌شود.

funct7	rs2	rs1	funct3	rd	opcode
0000000	xxxxx	xxxxx	001	xxxxx	0110011

مثال: `sll x5, x6, x7` → `x5 = x6 << (x7 & 0x1F)`

- srl: شیفت منطقی به راست مقدار rs1 به اندازه مشخص‌شده توسط ۵ بیت کم ارزش rs2 و ذخیره نتیجه در rd.

funct7	rs2	rs1	funct3	rd	opcode
0000000	xxxxx	xxxxx	101	xxxxx	0110011

مثال: `srl x5, x6, x7` → `x5 = x6 >> (x7 & 0x1F)`

- sra: شیفت حسابی به راست مقدار rs1 به اندازه مقدار مشخص‌شده توسط ۵ بیت کم ارزش rs2 با حفظ بیت علامت که نتیجه در rd قرار می‌گیرد.

funct7	rs2	rs1	funct3	rd	opcode
0100000	xxxxx	xxxxx	101	xxxxx	0110011

مثال: `sra x5, x6, x7` → `x5 = signed(x6) >> (x7 & 0x1F)`

- slt: اگر مقدار عدد علامتدار درون rs1 از مقدار عدد علامتدار درون rs2 کمتر باشد مقدار ۱ و در غیر این صورت مقدار ۰ در rd قرار می‌گیرد.

funct7	rs2	rs1	funct3	rd	opcode

0000000	xxxxx	xxxxx	010	xxxxx	0110011

مثال: `slt x1, x2, x3` → `x1 = (x2 < x3) ? 1 : 0`

sltu: عملیاتی مشابه slt را به صورت بدون علامت (unsigned) انجام می‌دهد.

funct7	rs2	rs1	funct3	rd	opcode
0000000	xxxxx	xxxxx	011	xxxxx	0110011

مثال: `sltu x1, x2, x3` → `x1=(unsigned(x2) < unsigned(x3)) ? 1 : 0`

- addi: جمع مقدار ثبات rs1 با یک عدد ثابت بلافصل (Immediate) و ذخیره نتیجه در ثبات rd. همان گونه که قابل مشاهده است، قالب این دستور با دستورات قبلی متفاوت می‌باشد.

imm[11:0]	rs1	funct3	rd	opcode
xxxxxxxxxxx	xxxxx	000	xxxxx	0010011

مثال: `addi x5, x6, 10` → `x5 = x6 + 10`

قابل ذکر است که این دستور امکان بارگذاری داده در ثبات‌ها را بدون نیاز به دسترسی به حافظه فراهم می‌کند. به عبارت دقیق‌تر اگر در مثال ارائه‌شده، مقدار ثبات x6 برابر صفر باشد، عدد ۱۰ در ثبات x5 بارگذاری خواهد شد.

- lh: ۱۶ بیت کم ارزش محتوای حافظه با آدرس `rs1 + imm` را به صورت sign-extende شده درون ثبات `rd` بارگذاری می‌کند.

imm[11:0]	rs1	funct3	rd	opcode
xxxxxxxxxxx	xxxxx	001	xxxxx	0000011

مثال: `lh x5, 4(x10)` → `x5 = sign-extend(M[x10+4][15:0])`

- lw: محتوای ۳۲ بیتی درون حافظه با آدرس تعیین شده را در `rd` بارگذاری می‌کند.

imm[11:0]	rs1	funct3	rd	opcode
xxxxxxxxxxx	xxxxx	010	xxxxx	0000011

مثال: `lw x5, 4(x10)` → `x5 = M[x10+4][31:0]`

- sh: مقدار ۱۶ بیت کم ارزش درون rs2 را در حافظه به آدرس [rs1 + imm] ذخیره می‌کند.

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
xxxxxxx	xxxxx	xxxxx	001	xxxxx	0100011

مثال: `sh x6, 2(x10)` → `M[x10 + 2] = x6[15:0]`

- sw: مقدار ۳۲ بیتی درون rs2 را درون حافظه به آدرس [rs1 + imm] ذخیره می‌کند.

imm [11:5]	rs2	rs1	func t3	imm [4:0]	opco de
xxxx xxx	xxxx x	xxxx x	010	xxxx x	0100 011

مثال: `sw x6, 2(x10)` → `M[x10 + 2] = x6[31:0]`

- beq: اگر رابطه rs1 == rs2 بین محتوای دو ثبات برقرار بود، ادامه اجرای دستورات به آدرس جدید PC + imm منتقل خواهد شد.

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
xxxxxxx	xxxxx	xxxxx	000	xxxxx	1100011

مثال: `beq x5, x6, label` → `if x5==x6 then jump to label`

- bne: اگر rs1 != rs2 بود به PC + imm پرش کن.

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
xxxxxxx	xxxxx	xxxxx	001	xxxxx	1100011

مثال: if x5!=x6 then jump to label → bne x5, x6, label

- blt: اگر rs1 < rs2 بود پرش انجام شود. در این مقایسه اعداد به صورت علامتدار در نظر گرفته می‌شوند.

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
xxxxxxx	xxxxx	xxxxx	100	xxxxx	1100011

مثال: if x5 < x6 (signed) then jump to label → blt x5, x6, label

- bge: اگر rs1 >= rs2 پرش انجام شود. در این مقایسه اعداد به صورت علامتدار در نظر گرفته می‌شوند.

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
xxxxxxx	xxxxx	xxxxx	101	xxxxx	1100011

مثال: if x5 >= x6 (signed) then jump to label → bge x5, x6, label

- bltu: اگر rs1 < rs2 پرش انجام شود. در این مقایسه اعداد به صورت بدون علامت در نظر گرفته می‌شوند.

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
xxxxxxx	xxxxx	xxxxx	110	xxxxx	1100011

مثال: if x5<x6 (unsigned) then jump to label → bltu x5, x6, label

- bgeu: اگر rs1 ≥ rs2 پرش انجام شود. در این مقایسه اعداد به صورت بدون علامت در نظر گرفته می‌شوند.

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
xxxxxxx	xxxxx	xxxxx	111	xxxxx	1100011

مثال: `bgeu x5, x6, label` → `if x5>=x6 (unsigned) then jump to label`

- jal: ابتدا `rd = PC + 4` می‌شود و سپس ادامه اجرا به آدرس `PC + imm` منتقل می‌گردد. در حقیقت این دستور برای فراخوانی توابع استفاده می‌شود.

imm[20|10:1|11|19:12]	rd	opcode
xxxxxxxxxxxxxxxxxxxx	xxxxx	1101111

مثال: `jal x1, 100` → `x1 = PC+4, PC += 100`

- jalr: مشابه دستور قبلی است با این تفاوت که پرش به آدرس `rs1 + imm` انجام خواهد شد (آدرس-دهی غیرمستقیم).

imm[11:0]	rs1	funct3	rd	opcode
xxxxxxxxxxx	xxxxx	000	xxxxx	1100111

مثال: `jalr x1, 0(x5)` → `x1 = PC+4; PC = x5 + imm`

- lui: بارگذاری مقدار بلافصل ۲۰ بیتی در بخش پرارزش rd (`rd = imm << 12`).

imm[31:12]	rd	opcode
xxxxxxxxxxxxxxxxxxxx	xxxxx	0110111

مثال: `lui x5, 0x12345` → `x5 = 0x12345000`

- auipc: بارگذاری به صورت `rd = PC + ( imm << 12 )` که برای آدرس‌دهی نسبی استفاده می‌شود.

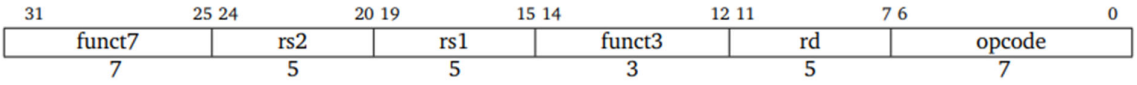
imm[31:12]	rd	opcode
xxxxxxxxxxxxxxxxxxxx	xxxxx	0010111

مثال: `auipc x5, 0x12` → `x5 = PC + 0x12000`

مجدداً تاکید می‌گردد که از مجوعه کلیه دستورات پایه، برای این پروژه فقط موارد ذکر شده در بالا ضروری هستند.

دستورالعمل‌های ضرب RV32M

در معماری پایه RISC-V دستورهای ضرب (mul)، تقسیم (div) و باقیمانده (rem) به‌صورت پیش‌فرض وجود ندارند. برای استفاده از این دستورات، لازم است از افزونه‌ی اختیاری M به سخت‌افزار اضافه شود. تمامی دستورها در این افزونه از قالب **R-type** پیروی می‌کنند و کد عملیاتی (opcode) آن‌ها برابر با **0110011** است. بنابراین برای تمایز میان آن‌ها، از فیلدهای **funct3** و **funct7** استفاده می‌شود.



در ادامه، دستوراتی از این مجموعه که برای انجام این پروژه الزامی هستند را بررسی خواهیم کرد.

- mul: ضرب اعداد ۳۲ بیتی علامتدار به صورت **rs1 × rs2** که فقط ۳۲ بیت پایین (بیت‌های [31:0]) ذخیره می‌شود.

funct7	rs2	rs1	funct3	rd	opcode
0000001	xxxxx	xxxxx	000	xxxxx	0110011

مثال: `mul x10, x5, x6` → `x10 = x5 * x6`

- mulh: مقدار ۳۲ بیت پرارزش حاصل از ضرب دو عدد ۳۲ بیتی علامتدار (بیت‌های [63:32]) را برمی‌گرداند.

funct7	rs2	rs1	funct3	rd	opcode
0000001	xxxxx	xxxxx	001	xxxxx	0110011

مثال: `mulh x11, x5, x6` → `x11 = (x5 * x6) >> 32`

- div: تقسیم صحیح علامتدار را به صورت rd = rs1 / rs2 انجام می‌دهد. در این عمل خارج‌قسمت ذخیره می‌شود.

funct7	rs2	rs1	funct3	rd	opcode
0000001	xxxxx	xxxxx	100	xxxxx	0110011

مثال: `div x14, x6, x5` → `x14 = x6 / x5`

- rem: باقیمانده تقسیم علامتدار را به صورت rd = rs1 % rs2 ذخیره می‌نماید.

funct7	rs2	rs1	funct3	rd	opcode
0000001	xxxxx	xxxxx	110	xxxxx	0110011

مثال: `rem x14, x6, x5` → `x14 = x6 % x5`

دستورات محاسبات ممیز شناور (ضروری برای گروه‌های چهارنفره)

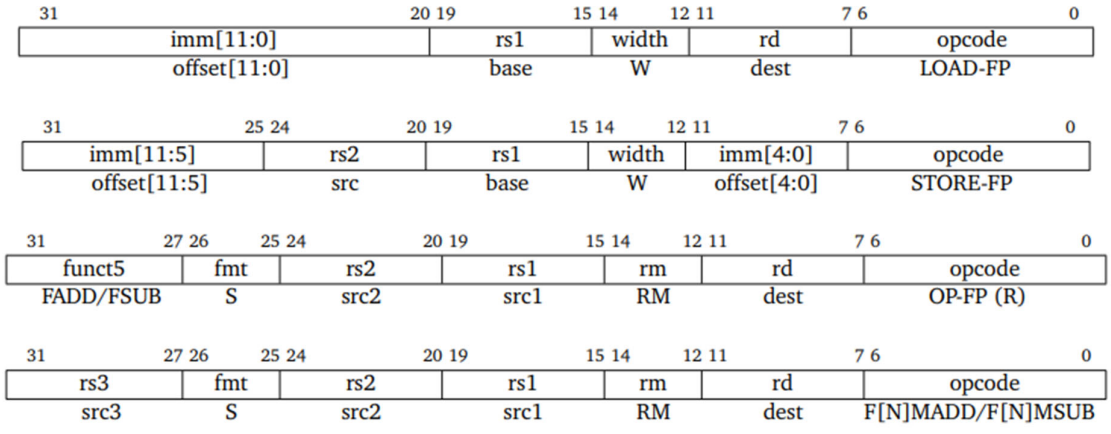
گروه‌های چهار نفره، علاوه بر دستورات مشخص‌شده، موظفند که طراحی و پیاده‌سازی دستورات مرتبط با ممیز شناور را نیز انجام دهند. انجام این بخش برای سایر گروه‌ها امتیاز اضافی در برخواهد داشت.

ابتدا لازم است که ثبات‌های اختصاصی جهت انجام محاسبات ممیز شناور را بشناسیم.

Floating Point Registers

f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

ساختار دستور ممیز شناور هم به صورت زیر است.



از مجموع دستورات موجود جهت انجام محاسبات اعشاری موارد زیر ضروری هستند. برای آشنایی با جزئیات بیشتر در این رابطه، پیشنهاد می‌شود که به منابع معتبر [۶] و [۷] مراجعه کرده یا در اینترنت به جستجو بپردازید.

- دستورات پایه:

flw	fadd.s	fmul.s
fsw	fsub.s	fdiv.s

- دستورات تبدیل:

fcvt.s.w	fcvt.w.s
fcvt.s.wu	fcvt.wu.s

- دستورات شرطی:

fle.s	flt.s	feq.s

دستورات اختیاری

پیاده‌سازی دستورات تعیین شده در این بخش به صورت اختیاری است و تنها در صورتی مشمول نمره خواهند بود که پیاده‌سازی دستورات اصلی بخش های قبلی به‌طور کامل و صحیح انجام شده باشند. فهرست این دستورات در ادامه ارائه شده است که برای آشنایی با جزییات آن‌ها باید به منابع و اینترنت رجوع نمایید.

- دستورات اختیاری پایه RV32I:

ori	slli	lb	lhu
andi	srli	lbu	sb

[۶] لینک پیشنهادی؛ RV32F, RV64D Instructions

[۷] لینک پیشنهادی؛ F Standard Extension for Single-Precision Floating-Point

- دستورات اختیاری ضرب و تقسیم RV32M:

mulsu	mulu	divu	remu

نکاتی در رابطه با زبان اسمبلی RISC V

در پیاده‌سازی اسمبلر بایستی موارد زیر را پیاده سازی نمایید. تنها یک بخش که به صورت صریح اعلام شده است اختیاری و مشمول امتیاز تشویقی است و سایر موارد الزامی هستند.

➢ فرمت عددنویسی در RISC-V

در کدهای اسمبلی، مقادیر عددی بلافصل (immediate values) می‌توانند در قالب‌های مختلفی نوشته شوند که مهم‌ترین آن‌ها عبارت‌اند از:

- **Decimal**: فرمت پیش‌فرض اعداد است. مثال: ۱۲، ۱۲-
- **Hexadecimal**: با پیشوند `0x` شروع می‌شود. مثال: `0xFF, 0x1A3`
- **Binary**: با پیشوند `0b` شروع می‌شود. مثال: `0b101010, 0b1111`

➢ فرمت حروف در RISC-V

در کدهای اسمبلی RISC-V، می‌توان به‌طور مستقیم حروف و سمبل‌ها را داخل حافظه ذخیره کرد. این کار به دو صورت معمول انجام می‌شود:

- تک حرف: مثلاً `'A'`
- رشته: مثلاً `"Hello World"`

در مرحله‌ی کامپایل، این حروف و رشته‌ها به مقادیر عددی متناظر خود (براساس جدول ASCII) تبدیل می‌شوند و در حافظه ذخیره می‌شوند.

➢ دستورات راهنما

دستوراتی که با نقطه (`.`) شروع می‌شوند، به نام Directive یا دستور راهنما شناخته می‌شوند. این‌ها دستورالعمل‌هایی برای راهنمایی اسمبلر هستند و نه برای اجرا بر روی CPU مقصد. در ادامه، فهرستی از مهم‌ترین دستورهای راهنما همراه با توضیحی مختصر ارائه شده است که باید در این پروژه پیاده‌سازی شوند.

- **.org**: این دستور به اسمبلر می‌گوید که موقعیت فعلی آدرس‌دهی را بر روی مقدار مشخصی تنظیم کند.
- **.word**: قراردادن یک مقدار ۳۲ بیتی (dword) درون حافظه.
- **.half**: قراردادن یک مقدار ۱۶ بیتی (word) درون حافظه.
- **.byte**: قراردادن یک مقدار ۸ بیتی (byte) درون حافظه.
- **.align n**: تراز کردن آدرس حافظه دستور بعدی. در حقیقت این راهنما به اسمبلر می‌گوید که آدرس بعدی داده یا کد باید مضرب 2^n باشد.

برای درک بهتر ساختار دستورات ویژه و برچسب های اسمبلی، به مثال زیر توجه کنید:

```
    .org 0x1000
my_word:
    .word 0xDEADBEEF

my_half:
    .half 0x1234

my_byte:
    .byte 0x7F

    # آدرس بعدی را به مضرب 2^2 = 4 تراز می‌کند
    .align 2
aligned_word:
    .word 0xCAFEBABE
```

➢ شبه دستورات

شبه دستورات یا Pseudo-Instructions در معماری RISC-V، دستوراتی هستند که به صورت مستقیم در مجموعه دستورالعمل‌های پایه (**ISA**) پشتیبانی نمی‌شوند اما برای راحتی برنامه‌نویس در سطح اسمبلی تعریف شده‌اند. این دستورها توسط اسمبلر به یک یا چند دستور واقعی تعریف شده در ISA تبدیل می‌شوند. در واقع شبه دستورات باعث افزایش خوانایی و ساده‌تر شدن برنامه‌نویسی می‌شوند، بدون این که نیاز به افزودن سخت‌افزار جدید در پردازنده باشد.
در ادامه، شبه دستوراتی که برای این پروژه انتخاب شده‌اند و پیاده‌سازی آن‌ها الزامی است را بررسی خواهیم کرد.

- nop: این دستور هیچ کاری انجام نمی‌دهد و برای ایجاد تأخیر و تنظیم زمان‌بندی یا به عنوان جایگاه موقت استفاده می‌شود.

دستور معادل: addi x0, x0, 0

- li: یک مقدار بلافصل (imm) را درون ثبات مقصد (rd) قرار می‌دهد.

دستور معادل:

- اگر imm کوچک (۱۲ بیت و کمتر) باشد: addi rd, x0, imm
- در غیر این صورت:

lui rd, upper(imm)
addi rd, rd, lower(imm)

* تابع upper ۲۰ بیت پرارزش عدد و تابع lower ۱۲ بیت کم ارزش عدد را بر می گرداند.

- mv: مقدار ثبات rs را درون ثبات rd کپی می‌کند.

دستور معادل: addi rd, rs, 0

- not: نقیض بیت‌به‌بیت (مکمل یک) مقدار درون ثبات rs را محاسبه و درون rd ذخیره می‌کند.

دستور معادل: xori rd, rs, -1

- neg: مقدار درون ثبات rs را در 1- ضرب می‌کند (مکمل دو) و درون rd می‌گذارد.

دستور معادل: sub rd, x0, rs

- شبه دستورات اختیاری: پیاده‌سازی این شبه دستورات اختیاری بوده و برای آشنایی با جزییات آن‌ها باید به منابع و اینترنت رجوع نمایید.

la	ret	j offset	call offset

➢ اندازه حافظه

پس از فرآیند اسمبل کردن کد اسمبلی، خروجی حاصل شامل مجموعه‌ای از دستورالعمل‌ها و داده‌هایی خواهد بود که درون حافظه‌ی اصلی (RAM) پردازنده قرار می‌گیرند. این خروجی به‌عنوان ورودی به برنامه‌ی شبیه‌ساز داده می‌شود تا اجرای برنامه شبیه‌سازی گردد.

با توجه به این که در این پروژه معماری ۳۲-بیتی (۳۲RV) در نظر گرفته شده، فضای آدرس‌دهی به صورت تئوری تا سقف ۴ گیگابایت (2^{32} بایت) قابل گسترش است. با این حال، به‌منظور کنترل حجم خروجی و جلوگیری از تخصیص بی‌مورد حافظه، توصیه می‌شود که محدوده‌ی آدرس‌دهی مورد استفاده را به میزان محدودتر درون اسمبلر مشخص و تعریف کنید. پیشنهاد ما بر اساس برنامه‌هایی که قرار است اجرا شوند، مقدار ۶۴ کیلو بایت است یعنی آدرس حافظه از **0x0** شروع و تا **0xFFFF** ادامه خواهد یافت. بنابراین چنان‌چه به آدرس‌های خارج از محدوده دسترسی صورت پذیرد، اسمبلر و شبیه‌ساز باید با صدور خطای مناسب به برنامه‌نویس هشدار دهند.

همچنین در این پیاده‌سازی، فرض بر آن است که مقدار اولیه‌ی ثبات شمارنده برنامه (PC) برابر با آدرس **0x1000** است.