# مستندات پروژه

# طراحی و پیاده‌سازی اسمبلر و شبیه‌ساز پردازنده RISC-V

## ۱. مقدمه و نگاه کلی به پروژه

این پروژه به عنوان بخشی از درس معماری کامپیوتر با هدف درک عمیق‌تر مفاهیم بنیادی طراحی پردازنده و نحوه اجرای دستورالعمل‌ها در سطح ماشین انجام شده است. هدف اصلی، طراحی و پیاده‌سازی یک زنجیره ابزار کامل برای معماری RISC-V (نسخه RV32IM) بود که شامل دو بخش اصلی است:

1. **اسمبلر (Assembler):** برنامه‌ای که کد اسمبلی نوشته شده توسط کاربر را به کد ماشین ۳۲ بیتی قابل فهم برای پردازنده تبدیل می‌کند.
2. **شبیه‌ساز پردازنده (CPU Simulator):** برنامه‌ای که یک پردازنده RISC-V را به صورت نرم‌افزاری شبیه‌سازی کرده و قادر است کد ماشین تولید شده توسط اسمبلر را خط به خط اجرا کند.

این پروژه به زبان C++ و با رویکرد شیءگرا پیاده‌سازی شده تا هر بخش از پردازنده و فرآیند اسمبل به صورت ماژولار و مستقل توسعه داده شود. در ادامه، ساختار کلی پروژه و وظیفه هر یک از فایل‌ها به تفصیل شرح داده می‌شود.

## ۲. ساختار کلی و جریان کار

جریان کار در این پروژه به این صورت است که ابتدا یک فایل با پسوند `asm.` حاوی کد اسمبلی RISC-V به **اسمبلر** داده می‌شود. اسمبلر در دو مرحله (Two-Pass) این کد را پردازش کرده و یک فایل باینری خروجی تولید می‌کند. سپس، این فایل باینری در حافظه **شبیه‌ساز پردازنده** بارگذاری شده و پردازنده شبیه‌سازی شده، دستورالعمل‌ها را یکی پس از دیگری واکشی (Fetch)، کدگشایی (Decode) و اجرا (Execute) می‌کند.

اجزای اصلی پروژه را می‌توان به سه دسته تقسیم کرد: **اجزای اسمبلر**، **اجزای شبیه‌ساز** و **اجزای مشترک**.

## ۳. تشریح فایل‌ها و ماژول‌ها

در این بخش، وظیفه هر ماژول و فایل‌های مرتبط با آن ( `h.` و `cpp.` ) توضیح داده می‌شود.

### ۳.۱. اجزای مشترک (Shared Components)

### `common.h`

این فایل به عنوان یک "جعبه ابزار" برای کل پروژه عمل می‌کند. تمام تعاریف، ثوابت، ساختارهای داده‌ای و توابع کمکی که در چندین ماژول مختلف استفاده می‌شوند، در این فایل قرار گرفته‌اند تا از تکرار کد جلوگیری شود و یکپارچگی پروژه حفظ گردد.

- **ثوابت:** مانند `MEMORY_SIZE` که حجم کل حافظه را مشخص می‌کند.
- **ساختارهای داده:** مانند `InstructionInfo` که اطلاعات مربوط به هر دستورالعمل (opcode, funct3, ...) را برای استفاده در اسمبلر نگهداری می‌کند.
- **توابع کمکی (`Utils`):** شامل توابعی مانند `trim` برای حذف فضاهای خالی از رشته‌ها، `split` برای جدا کردن بخش‌های یک رشته و `register_to_int` برای تبدیل نام رجیستر (مانند `t0`) به شماره آن (مانند `5`).

## ۳.۲. اجزای اسمبلر (Assembler Components)

وظیفه این مجموعه از فایل‌ها، تبدیل کد اسمبلی به کد ماشین است.

### `parser.h` / `parser.cpp`

این ماژول "کارگردان" فرآیند اسمبل است. فرآیند اسمبل را در دو گذر (Pass) مدیریت می‌کند:

- **گذر اول (`first_pass`):** در این مرحله، کل کد اسمبلی یک بار خوانده می‌شود تا تمام برچسب‌ها (Labels) و آدرس حافظه متناظر با آن‌ها شناسایی و در `SymbolTable` ذخیره شوند. این کار برای محاسبه آدرس پرش‌ها و انشعاب‌ها ضروری است.
- **گذر دوم (`second_pass`):** در این مرحله، کد اسمبلی دوباره خوانده می‌شود. این بار با داشتن آدرس تمام برچسب‌ها، هر خط از کد به ماژول `Encoder` فرستاده شده تا به کد ماشین ۳۲ بیتی تبدیل شود و در نهایت در یک فایل باینری نوشته شود.

### `encoder.h` / `encoder.cpp`

این ماژول "مترجم" پروژه است. وظیفه اصلی آن، دریافت نام یک دستورالعمل (مثلاً `add`) و عملوندهای آن (مثلاً `x1, x2, x3`) و تبدیل آن به کد ماشین ۳۲ بیتی باینری بر اساس فرمت‌های استاندارد RISC-V (مانند R-Type, I-Type, S-Type و ...) است. یکی از چالش‌های اصلی در پیاده‌سازی این بخش، مدیریت نحوه کدگذاری متفاوت مقادیر فوری (Immediate) در فرمت‌های مختلف بود.

### `symbol_table.h` / `symbol_table.cpp`

این کلاس یک "دفترچه آدرس" ساده اما حیاتی است. یک جدول درهم‌سازی (Hash Table) برای نگاشت نام برچسب‌ها (Labels) به آدرس حافظه‌شان نگهداری می‌کند. `Parser` در گذر اول این جدول را پر کرده و در گذر دوم برای محاسبه آفست‌های دستورات پرش و انشعاب از آن استفاده می‌کند.

### `pseudo.h` / `pseudo.cpp`

بسیاری از دستورات رایج در اسمبلی RISC-V، در واقع دستورات شبه (Pseudo-instructions) هستند که برای راحتی برنامه‌نویس تعریف شده‌اند. برای مثال، دستور `mv rd, rs` (انتقال مقدار یک رجیستر به دیگری) یک دستور شبه است که به دستور واقعی `addi rd, rs, 0` تبدیل می‌شود. این ماژول وظیفه شناسایی این دستورات و گسترش (Expand) آن‌ها به یک یا چند دستورالعمل واقعی RISC-V را بر عهده دارد تا `Encoder` بتواند آن‌ها را پردازش کند.

## ۳.۳. اجزای شبیه‌ساز پردازنده (CPU Simulator Components)

این بخش، قلب پروژه است و یک پردازنده واقعی را شبیه‌سازی می‌کند.

### `cpu.h` / `cpu.cpp`

این کلاس، پردازنده مرکزی (CPU) را به طور کامل شبیه‌سازی می‌کند. من در این پروژه یک مدل **چند چرخه‌ای (Multi-Cycle)** را پیاده‌سازی کردم تا فرآیند اجرای دستورالعمل‌ها به واقعیت نزدیک‌تر باشد.

- **حالت پردازنده:** شامل شمارنده برنامه ( `pc` )، فایل رجیستر ( `reg_file` ) و دسترسی به حافظه ( `memory` ).
- **پایپ‌لاین شبیه‌سازی شده:** وضعیت فعلی پردازنده در یکی از مراحل `FETCH` ، `DECODE` ، `EXECUTE` ، `MEMORY` ، `WRITE_BACK` قرار دارد.
- **حلقه اصلی ( `clock_tick` ):** با هر فراخوانی این تابع، پردازنده یک کلاک پالس جلو می‌رود و عملیات مربوط به مرحله فعلی پایپ‌لاین را انجام می‌دهد. این تابع، تمام منطق جابجایی بین مراحل و اجرای دستورات را مدیریت می‌کند.
- **بارگذاری برنامه:** تابع `load_program` فایل باینری را از دیسک خوانده و در حافظه شبیه‌سازی شده بارگذاری می‌کند.

### `instruction.h` / `instruction.cpp`

این ماژول نقش "کدگشا" (Decoder) را ایفا می‌کند. یک دستورالعمل ۳۲ بیتی خام را از حافظه دریافت کرده و آن را بر اساس فرمت‌های RISC-V به اجزای سازنده‌اش تجزیه می‌کند: `opcode` ، `rd` ، `rs1` ، `rs2` ، `funct3` ، `funct7` و مقدار `imm` . همچنین نام دستور (Mnemonic) و نوع آن را مشخص می‌کند. این اطلاعات برای مراحل بعدی، به خصوص `EXECUTE` ، حیاتی است.

### `register_file.h` / `register_file.cpp`

این کلاس، فایل رجیستر پردازنده را شبیه‌سازی می‌کند. یک آرایه ۳۲ عضوی برای نگهداری مقادیر ۳۲ رجیستر عمومی (x0 تا x31) دارد. توابع `read` و `write` را برای دسترسی به رجیسترها فراهم می‌کند و منطق مهمی را پیاده‌سازی می‌کند: **نوشتن در رجیستر `x0` هیچ اثری ندارد** و مقدار آن همیشه صفر باقی می‌ماند.

### `memory.h` / `memory.cpp`

این کلاس حافظه اصلی (Main Memory) سیستم را شبیه‌سازی می‌کند. در این پروژه، یک حافظه ۶۴ کیلوبایتی به صورت یک آرایه از بایت‌ها (`<std::vector<uint8_t>`) پیاده‌سازی شده است. این ماژول توابعی برای خواندن و نوشتن مقادیر با اندازه‌های مختلف (بایت، نیم‌کلمه و کلمه) فراهم می‌کند و ترتیب بایت‌ها به صورت **Little-Endian** (همانند استاندارد RISC-V) مدیریت می‌شود.

## ۴. روند توسعه و چالش‌ها

روند توسعه پروژه به صورت ماژولار و از پایه‌ای‌ترین بخش‌ها شروع شد:

1. ابتدا کلاس‌های `Memory` و `RegisterFile` به عنوان زیرساخت‌های اصلی پیاده‌سازی شدند.
2. سپس، کار بر روی **اسمبلر** آغاز شد. ابتدا `Encoder` برای تبدیل دستورات به کد ماشین و سپس `Parser` برای مدیریت فرآیند دو گذره توسعه یافت. داشتن یک اسمبلر کارا در مراحل اولیه به من اجازه داد تا برنامه‌های تستی کوچک نوشته و فایل‌های باینری برای تست شبیه‌ساز تولید کنم.
3. در مرحله بعد، **شبیه‌ساز** توسعه یافت. ابتدا ماژول `Instruction` برای کدگشایی و سپس کلاس اصلی `CPU` با منطق پایپ‌لاین چند چرخه‌ای پیاده‌سازی شد.
4. یکی از بخش‌های چالش‌برانگیز، **پیاده‌سازی صحیح منطق ALU و به‌روزرسانی PC** در مرحله `EXECUTE` بود. مدیریت پرش‌ها، انشعاب‌ها و محاسبه آدرس‌ها نیازمند دقت بالایی بود تا مطابق با استاندارد RISC-V عمل کند.
5. **تست و اشکال‌زدایی** بخش بزرگی از زمان پروژه را به خود اختصاص داد. برای هر دستورالعمل، یک تست کیس کوچک در اسمبلی نوشته، به باینری تبدیل و سپس در شبیه‌ساز اجرا می‌شد تا از صحت عملکرد ALU، دسترسی به حافظه و به‌روزرسانی رجیسترها اطمینان حاصل شود.