# CS202 RISCV Instruction Based CPU

## 团队分工

| 姓名 | 学号 | 分工 | 贡献比 |
| --- | --- | --- | --- |
| 王恺勋 | 12310803 | 总体设计，顶层整合，调试修改，decode和alu实现，汇编器开发 | 38% |
| 王扬皓 | 12310802 | riscv测试样例编写，调试修改，IO输入设计 | 31% |
| 张泰玮 | 12310801 | ifetch和dmem实现 | 31% |

## 计划进程

| 进度 | 日期 |
| --- | --- |
| 开始进行总体设计和需要的instruction确定 | 5月6日 |
| 各个模块编写完毕 | 5月8日 |
| testcase1汇编编写完毕 | 5月10日 |
| 中期答辩，测试完毕testcase1，修改了一些时序bug | 5月13日 |
| testcase2汇编编写完毕，增加了auipc的实现 | 5月15日 |
| testcase2汇编debug完毕，ecall实现，开始设计自己的assembler解决rars问题 | 5月17日 |
| 上板测试调试，修改时序设计bug，assembler设计完毕 | 5月18日 |
| 基本设计完毕 | 5月19日 |

## CPU Architecture

### ISA design

| R-Type | opcode | rs1 | rs2 | rd | funct7 | funct3 | usage | example |
|---|---|---|---|---|---|---|---|---|
| add | 0110011 | rs1 | rs2 | rd | 0000000 | 000 | rd <- rs1 + rs2 | `add t0, x0, t1` |
| sub | 0110011 | rs1 | rs2 | rd | 0100000 | 000 | rd <- rs1 - rs2 | `sub t0, x0, t1` |
| and | 0110011 | rs1 | rs2 | rd | 0000000 | 111 | rd <- rs1 & rs2 | `and t0, x0, t1` |
| or | 0110011 | rs1 | rs2 | rd | 0000000 | 110 | rd <- rs1 \| rs2 | `or t0, x0, t1` |
| xor | 0110011 | rs1 | rs2 | rd | 0000000 | 100 | rd <- rs1 ^ rs2 | `xor t0, x0, t1` |
| slt | 0110011 | rs1 | rs2 | rd | 0000000 | 010 | rd <- (rs1 < rs2) ? 1 : 0 | `slt t0, x0, t1` |
| sll | 0110011 | rs1 | rs2 | rd | 0000000 | 001 | rd <- rs1 << rs2 | `sll t0, x0, t1` |
| srl | 0110011 | rs1 | rs2 | rd | 0000000 | 101 | rd <- rs1 >> rs2 | `srl t0, x0, t1` |
| sltu | 0110011 | rs1 | rs2 | rd | 0000000 | 011 | rd <- (rs1 < rs2) ? 1 : 0 | `sltu t0, x0, t1` |
| mul | 0110011 | rs1 | rs2 | rd | 0000001 | 000 | rd <- rs1*rs2[31:0] | `mul t0, x0, t1` |
| div | 0110011 | rs1 | rs2 | rd | 0000001 | 100 | rd <- rs1/rs2 | `div t0, x0, t1` |

| I-Type1 | opcode | rs1 | rs2 | rd | funct7 | funct3 | usage | example |
|---|---|---|---|---|---|---|---|---|
| addi | 0010011 | rs1 | imm | rd | NULL | 000 | rd <- rs1 + imm | `addi t0, t1, 2` |
| xori | 0010011 | rs1 | imm | rd | NULL | 100 | rd <- rs1 ^ imm | `xori t0, t1, 3` |
| ori | 0010011 | rs1 | imm | rd | NULL | 110 | rd <- rs1 \| imm | `ori t0, t1, 4` |
| andi | 0010011 | rs1 | imm | rd | NULL | 111 | rd <- rs1 & rs2 | `andi t0, t1, 5` |
| slli | 0010011 | rs1 | imm | rd | imm[11:5]=0x00 | 001 | rd <- rs1 << imm[4:0] | `slli t0, t1, 6` |
| srli | 0010011 | rs1 | imm | rd | imm[11:5]=0x00 | 101 | rd <- rs1 >> imm[4:0] | `srli t0, t1, 7` |
| slti | 0010011 | rs1 | imm | rd | NULL | 010 | rd <- (rs1 < imm) ? 1 : 0 | `slti t0, t1, 9` |

| I-Type2 | opcode | rs1 | rs2 | rd | funct7 | funct3 | usage | example |
|---|---|---|---|---|---|---|---|---|
| lb | 0000011 | rs1 | imm | rd | NULL | 000 | rd = {{24{M[rs1+imm][7]}}, M[rs1+imm][7:0]}; | `lb t0, 0(t1)` |
| lw | 0000011 | rs1 | imm | rd | NULL | 010 | rd = M[rs1+imm][31:0] | `lw t0, 0(t1)` |
| lbu | 0000011 | rs1 | imm | rd | NULL | 100 | rd = {24'b0, M[rs1+imm][7:0]} | `lbu t0, 0(t1)` |

| S-Type | opcode | rs1 | rs2 | rd | funct7 | funct3 | usage | example |
|---|---|---|---|---|---|---|---|---|
| sw | `0100011` | rs1 | imm | rd | NULL | `010` | M[rs1+imm][31:0] = rs2[31:0] | `sw t0, 0(t1)` |

| B-Type | opcode | rs1 | rs2 | rd | funct7 | funct3 | usage | example |
|---|---|---|---|---|---|---|---|---|
| beq | `1100011` | rs1 | rs2 | rd | NULL | `000` | PC += (rs1 == rs2) ? {imm,1'b0} : 4 | `beq t0, t1, label` |
| bne | `1100011` | rs1 | rs2 | rd | NULL | `001` | PC += (rs1 != rs2) ? {imm,1'b0} : 4 | `bne t0, t1, label` |
| blt | `1100011` | rs1 | rs2 | rd | NULL | `100` | PC += (rs1 < rs2) ? {imm,1'b0} : 4 | `blt t0, t1, label` |
| bge | `1100011` | rs1 | rs2 | rd | NULL | `101` | PC += (rs1 >= rs2) ? {imm,1'b0} : 4 | `bge t0, t1, label` |
| bltu | `1100011` | rs1 | rs2 | rd | NULL | `110` | PC += (rs1(unsigned) < rs2(unsigned)) ? {imm,1'b0} : 4 | `bltu t0, t1, label` |
| bgeu | `1100011` | rs1 | rs2 | rd | NULL | `111` | PC += (rs1(unsigned) >= rs2(unsigned)) ? {imm,1'b0} : 4 | `bgeu t0, t1, label` |

| JAL-Type | opcode | rs1 | rs2 | rd | funct7 | funct3 | usage | example |
|---|---|---|---|---|---|---|---|---|
| jal | `1101111` | NULL | imm | rd | NULL | NULL | rd = PC + 4; PC += imm | `jal t0, label` |

| JALR-Type | opcode | rs1 | rs2 | rd | funct7 | funct3 | usage | example |
|---|---|---|---|---|---|---|---|---|
| jalr | `1100111` | rs1 | imm | rd | NULL | NULL | rd = PC + 4; PC = rs1 + imm | `jalr t0, 0(t1)` |

| U-Type | opcode | rs1 | rs2 | rd | funct7 | funct3 | usage | example |
|---|---|---|---|---|---|---|---|---|
| lui | `0110111` | NULL | NULL | rd | NULL | NULL | rd = imm << 12 | `lui t0, 0x80000` |
| auipc | `0010111` | NULL | NULL | rd | NULL | NULL | rd = PC + imm << 12 | `auipc t0, 0x80000` |

| System | opcode | rs1 | rs2 | rd | funct7 | funct3 | usage | example |
|---|---|---|---|---|---|---|---|---|
| ecall | `1110011` | NULL | NULL | NULL | `0000000` | `000` | a7 = 0 output register a0 to led<br>a7 = 1 store swtich data to a0 | `ecall` |

| Pseudo(assembler support) | Description |
|---|---|
| la(support both data and label) | auipc, addi |
| li | addi |
| mv | addi |
| beqz | beq |
| bnez | bne |

# CPU design

| CPU_clk | frequency | description |
|---|---|---|
| fpga_clk | 100MHz | input from EGO1 |
| clk | 6MHz | for most cpu components |
| clk_mem | 48MHz | for dmem to read and write |

本设计为单周期处理器，理论CPI接近1，不支持pipeline，32个32位寄存器

This cpu design is single-cycle cpu, theoretical CPI approaches 1, do not support pipeline, 32 32bit register.

本设计为哈佛架构，寻址单位为byte（字节），指令空间大小为64KB，数据空间64KB，栈空间的基地址为0000_FFFF。

采用MMIO和轮询的方式访问IO。

This design is Harvard architecture, address unit is byte, instruction capacity is 64KB, data capacity is 64KB, the base address of stack space is 0000_FFFF.

Using MMIO and polling to access IO.

| IO address | Function |
| --- | --- |
| 0xFFFFFE50 | 开关读取sw_input（8bit） |
| 0xFFFFFE54 | 开关读取确认键状态（1bit） |
| 0xFFFFFE44 | 开关读取testcase（3bit） |
| 0xFFFFFE08 | LED左8位显示 |
| 0xFFFFFE84 | 显示管16进制显示 |
| 0xFFFFFE88 | 显示管10进制显示 |

## CPU ports

| PORT | Description |
| --- | --- |
| fpga_clk | 100MHz from EGO1 board |
| fpga_rst | rst from EGO1 board |
| switch_16 | 16 bit switch signal from EGO1 board |
| led_16 | 16 bit led control signal to EGO1 board |
| seg_sel_8 | 7 segment display select signal(one-hot) |
| seg_tube0_8 | right 4 segment display |
| seg_tube1_8 | left 4 segment display |

# 方案分析说明

## 浮点数运算

### 硬件

在verilog中实现一条fadd指令，通过软件指令读取输入的8bitIEEE754标准浮点数，通过一条fadd指令实现两数加法（或减法）。但是在实验过程中发现，原本的23MHz时钟下，该fadd指令有时无法在1 cycle内计算完毕两数加法，因为其涉及多个加法器、乘法器、位移操作和除法器，尤其是乘法器和除法器对时间消耗大，会得到随机错误结果。

### 软件

在多个cycle内实现浮点数加法，不会有时序问题，但是实现较麻烦，还缺少复用性。

### 结论

使用软件实现。

## 哈佛架构下data段地址设计方案

### 软件rars

因为rars有两种方案，一种是ins从地址0开始，一种是data从0开始，其余指令都正常，但如果涉及到了auipc这种需要pc寄存器和某个地址之间相对距离的指令，rars的指令没有办法实现读取对应的指令，因为rars使用的是冯·诺依曼架构，我们实现的是哈佛架构。

在软件实现，可以把pc寄存器设计成从0开始，然后在（auipc或者la）相关指令执行时，将data（包括.data声明的和label）地址添加0x00002000的负偏移量，这样并不优雅。

### 手动添加空地址

或者在rars输出的data.txt前面手动添加 $\frac{0\times00002000_{16}}{4_{10}}=2048_{10}$ 条00000000数据，之后就可以正常使用rars的data读取了，这样也不优雅。

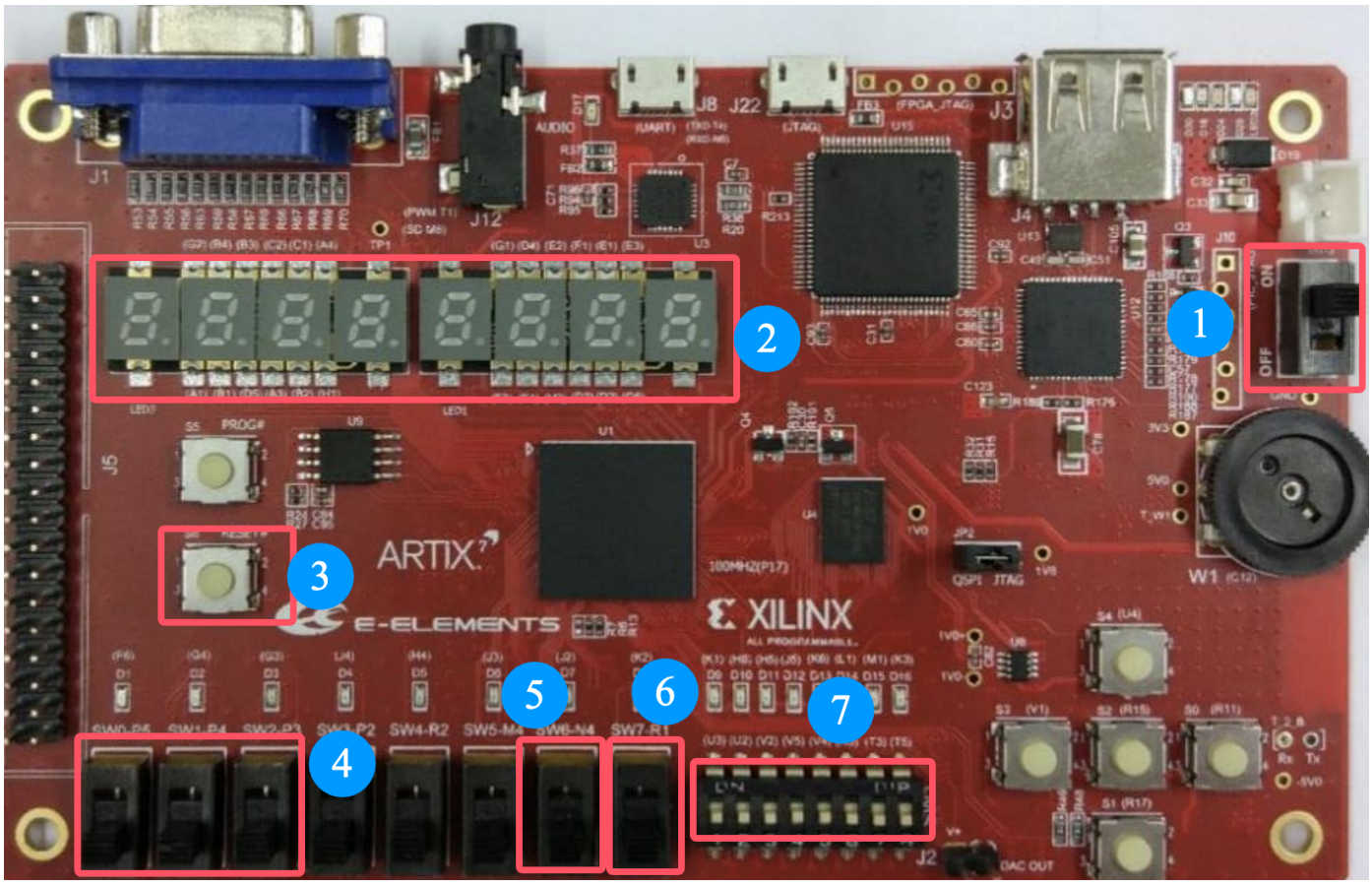
### 自行编写assembler软件

最后我们自己用java编写了**适配哈佛架构**的编译器，输入asm文件（支持.data，label寻址，自定义伪指令（单条和多条都可），注释），输出自定义长度的ins.coe和data.coe文件。

### 结论

自行编写assembler适配哈佛架构，改进工具链。

# 系统上板使用说明



| 序号 | 名称 | 说明 |
| --- | --- | --- |
| 1 | 电源开关 | 打开后开发板上电 |
| 2 | 7段数码显示管 | 有16进制和10进制显示两种模式 |
| 3 | 系统复位键 | 按下后初始化 |
| 4 | 输入当前testcase number | 3位testcase 000-111 |
| 5 | uart传输模式开关 | （未实现） |
| 6 | 回车（确认输入键） | 做确认以及步进效果 |
| 7 | 数据输入 | 8bit输入 |

# 自测试说明

| Testcase | 测试方法 | 测试类型 | 测试用例描述 | 结果 |
| --- | --- | --- | --- | --- |
| 1-0 | 仿真+上板 | 集成 | 输入`a`, 输出`a`, 输入`b`, 输出`b` | 通过 |
| 1-1 | 仿真+上板 | 集成 | 输入`a`, `lb`存入, `sw`到MEM和数码管 | 通过 |
| 1-2 | 仿真+上板 | 集成 | 输入`b`, `lbu`存入, `sw`到MEM和数码管 | 通过 |
| 1-3 | 仿真+上板 | 集成 | a和b是否相等，相等点亮8个灯 | 通过 |
| 1-4 | 仿真+上板 | 集成 | a是否小于b，小于点亮8个灯 | 通过 |
| 1-5 | 仿真+上板 | 单元+集成 | a是否小于b (unsigned)，小于点亮8个灯 | 通过 |
| 1-6 | 仿真+上板 | 单元+集成 | a是否小于b，小于点亮1个灯 | 通过 |
| 1-7 | 仿真+上 | 单元+集 | a是否小于b (unsigned)，小于点亮1个灯 | 通过 |

| | | | | |
|---|---|---|---|---|
| | 板 | 成 | | |
| 2-0 | 仿真+上板 | 单元+集成 | 输入8bit，led输出8bit倒序 | 通过 |
| 2-1 | 仿真+上板 | 单元+集成 | 输入8bit，是回文点亮1个led | 通过 |
| 2-2 | 仿真+上板 | 单元+集成 | 输入2个8bit浮点数，存入MEM，十进制显示整数部分 | 通过 |
| 2-3 | 仿真+上板 | 单元+集成 | 计算2-2两个浮点数加起来结果的整数部分，十进制显示 | 通过 |
| 2-4 | 仿真+上板 | 单元+集成 | 输入4bit，生成CRC-4检验，拼接元数据后在led上显示 | 通过 |
| 2-5 | 仿真+上板 | 单元+集成 | 输入8bit，与2-4生成的8bit数据比较，相同点亮1个灯 | 通过 |
| 2-6 | 仿真+上板 | 单元+集成 | 测试了auipc, la(data段和label段), mul, div, ecall(两种)，用ecall输入输出，乘2除4，最终输出输入值除2 | 通过 |
| 2-7 | 仿真+上板 | 单元+集成 | 测试了jalr，如果jalr工作，则点亮一盏灯 | 通过 |

`lui`测试见每个样例开头的计算IO地址部分。

# Reference List

Debouncer, segment_display设计借鉴了本人去年数字逻辑的项目：https://github.com/KaixunWang/SUSTech-CS207Project--Rangehood-105points

头文件管理灵感来自：https://github.com/OctCarp/SUSTech_CS202-Organization_2023s_Project-CPU/blob/main/CPU_Verilog/vga/vhead.svh

Assembler设计借鉴了：https://github.com/TheThirdOne/rars, https://github.com/Thewbi/riscvasm, https://github.com/kcelebi/riscv-assembler

BCD码计算使用了AI设计实现“加三移位法”，减少除法使用：https://en.wikipedia.org/wiki/Excess-3

在设计Assembler时，使用了AI进行寻址查找扫描设计：具体见Assembler

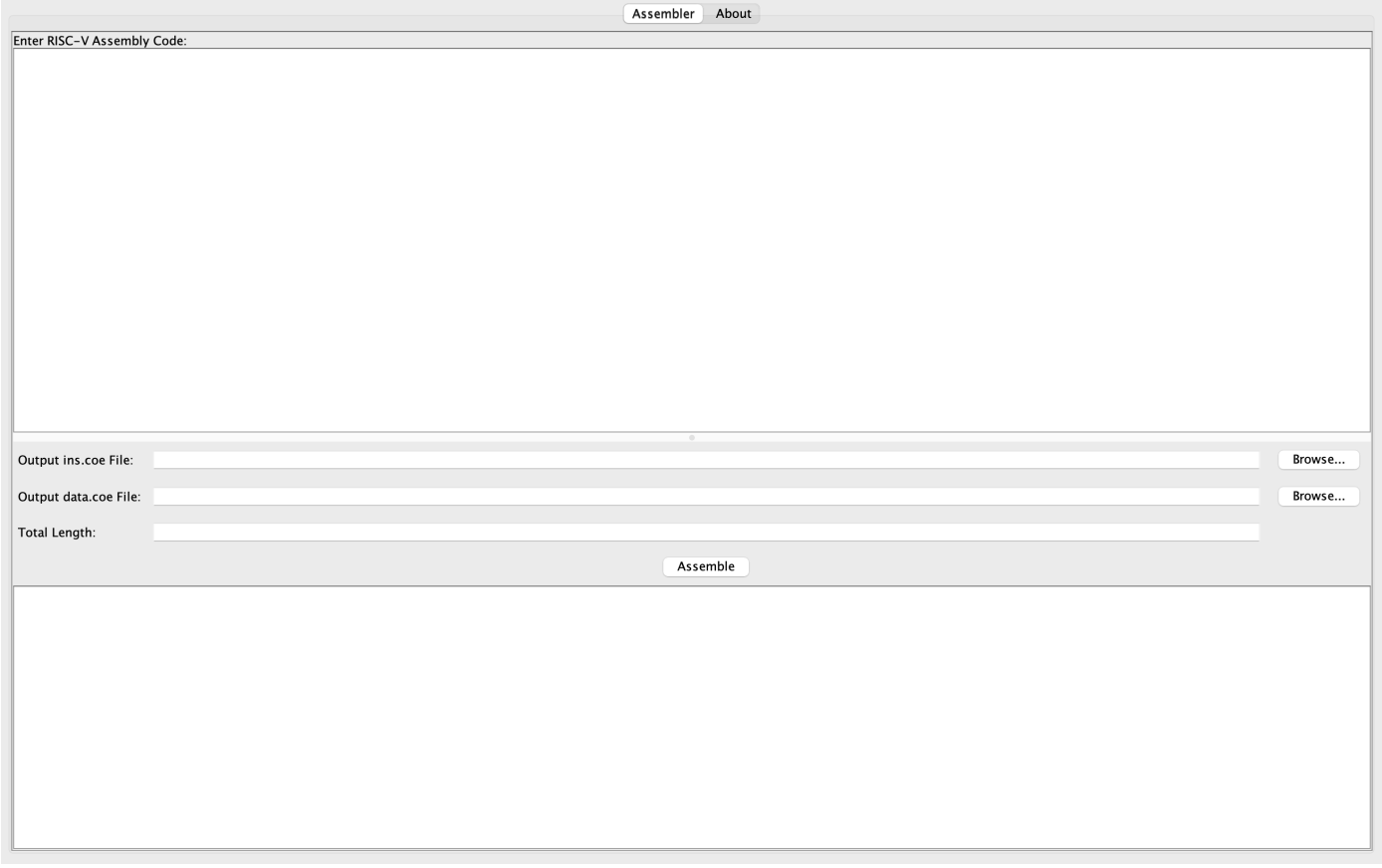
# Bonus相关

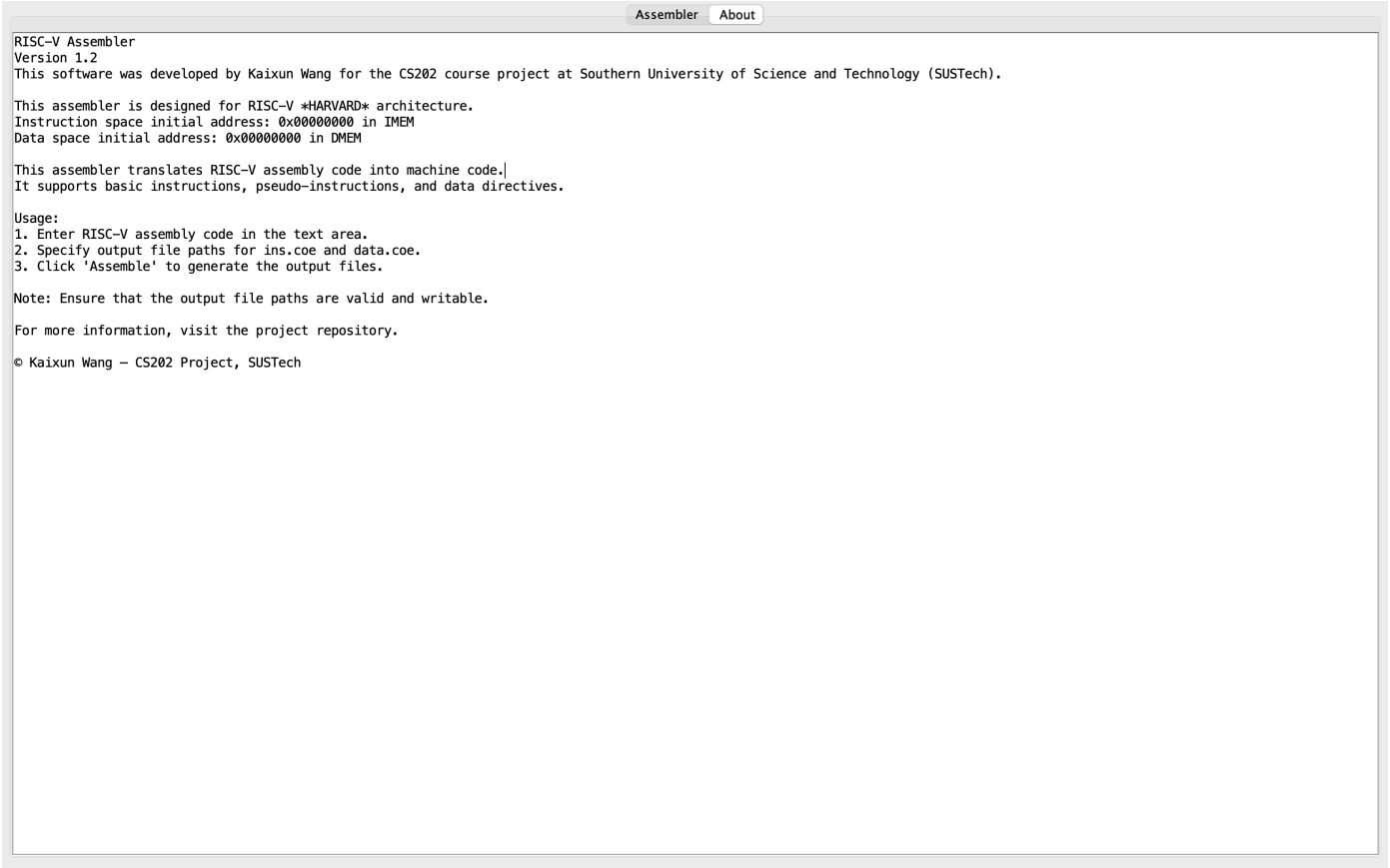
# ISA指令扩展

| Instruction | opcode | rs1 | rs2 | rd | funct7 | funct3 | usage | example |
|---|---|---|---|---|---|---|---|---|
| ecall | `1110011` | NULL | NULL | NULL | `0000000` | `000` | a7 = 1 output register a0 to led<br>a7 = 0 store swtich data to a0 | `ecall` |
| mul | `0110011` | rs1 | rs2 | rd | `0000001` | `000` | rd <- (rs1*rs2)[31:0] | `mul t0, x0, t1` |
| div | `0110011` | rs1 | rs2 | rd | `0000001` | `100` | rd <- rs1/rs2 | `div t0, x0, t1` |
| auipc | `0010111` | NULL | NULL | rd | NULL | NULL | rd = PC + imm << 12 | `auipc t0, 0x80000` |
| la | pseudo-ins | pseudo-ins | pseudo-ins | pseudo-ins | pseudo-ins | pseudo-ins | automatically get address and `auipc` and `addi` | `la t0,label` |

# 自创汇编器、编译器

我们自己用java编写了**适配哈佛架构**的编译器（在bonus里有**两个得分点**，直接由asm文件编译到coe文件）（见Assembler源码和jar包），输入asm文件（支持.data，label寻址，**自定义伪指令**（编译后单条和多条都可），注释），直接输出编译后**自定义长度**的ins.coe和data.coe文件。






# 问题和总结

在本次 RISC-V CPU 的设计与实现过程中，我们从最初的架构选择到后期的调试和部署，遇到了诸多挑战，也积累了宝贵的经验。

首先，由于我们采用的是**哈佛架构**，而主流的 RISC-V 仿真环境（如 RARS）采用的是**冯·诺依曼架构**，两者在指令与数据的访问路径上存在本质差异。因此，直接使用现有工具链并不能满足我们的需求。为了解决这个兼容性问题，我们选择**自行编写一套新的编译器和汇编支持工具链**，使得程序可以适配我们的哈佛架构 CPU，这一过程加深了我们对指令集和编译原理的理解。

在硬件调试阶段，**时序问题是最大的难点之一**。即使使用了 FPGA 提供的 IP 核，并将其配置为最低支持频率，依然出现了数据无法稳定写入的情况。经过多次测试与波形分析，我们最终决定将数据内存（DMEM）的时钟频率提升至主频的 **8 倍**，以确保读写操作能够在一个 CPU 时钟周期内可靠完成。该设计虽然增加了系统复杂度，但有效避免了数据错乱的问题。

此外，项目中还频繁遇到**命名不一致、信号线连接错误**等问题，这类细节问题往往导致模块功能异常或综合失败。我们通过严格规范模块接口和命名规则，并借助 Vivado 的波形调试工具，逐步排查并解决了这些问题。

在项目协作方面，由于我们使用 Vivado 进行开发，其生成的大量临时文件和工程依赖对 GitHub 等版本控制工具的兼容性较差。这使得团队协作和版本管理一度陷入混乱。我们最终通过配置 `.gitignore` 文件，精简提交内容，仅保留核心源代码和约定的工程配置，改善了这一问题。

最后，我们还遇到了多种**随机性问题**，如时序竞争、信号未初始化等，这些问题往往难以复现，每次编译和烧录的表现都可能不同。这类问题虽然难以彻底避免，但也促使我们更加重视硬件设计的鲁棒性与验证流程的完整性。

综上所述，本项目的实现过程虽然充满挑战，但也极大地锻炼了我们对处理器设计、时序分析、系统架构与协同开发的实际能力。每一个问题的解决都让我们对底层硬件系统有了更深入的理解，为今后更复杂系统的开发打下了坚实的基础。