| 小组成员 | 队长/队员 | 班级 | 学号 |
| --- | --- | --- | --- |
| 崔庭瑀 | 队长 | 无04 | 2019011083 |
| 蒋嘉铭 | 队员 | 计01 | 2020010885 |
| 袁樱 | 队员 | 智00 | 2020010930 |
| 王文琦 | 队员 | 计03 | 2020010915 |
| 吴玮妍 | 队员 | Cornell | 2020403531 |

小组人员

## 基本功能完成情况

### 导论

本次大作业是基于RISC-V指令集并对其做出精简形成的自定义Extremely Reduced Instruction Set(ERISC)指令集。本组将模拟实际汇编语言的编译，把内存控制指令全部翻译为16进制指令的机器码储存于本组自定义的栈中，根据此栈执行指令并进行可视化。我们可以执行的指令不仅包含了基本要求中的内存访问指令、栈操作指令、寄存器计算指令、控制指令、操作指令的记录和可视化，还在寄存器计算指令中加入了浮点数运算，使得我们的程序更加精准并多功能化。之后的段落将详细介绍我们对于每一个要求的施行结果。

### 程序翻译介绍

本组将模拟实际汇编语言的编译，将控制指令均翻译成为16进制指令存储于自定义的栈InstructionStack中。InstructionStack的大小为0x400010（4M），他是一个单独的struct，里面包含了stack和stacktop两个fields来代表我们自定义的栈和追踪栈顶位置的变量。程序中全部跳转与函数调用指令将直接在编译阶段被翻译为8位地址码，指示跳转后应执行的程序行在原给定指令集（instructionSet）中的位置。我们将根据instructionStack编译出来的机器码进行指令运行。指令翻译展示如图2.1。翻译功能在InstructionParser.cpp和InstructionParser.h中实施。

![img](file:///C:/Users/Lenovo/AppData/Local/Temp/msohtmlclip1/01/clip_image002.gif)

（图2.1)

以add指令为例，我们将这个指令用16进制数0x40代替，将其需要的寄存器rd，rs1和rs2也用1字节的16进制数表示。比如，如果rd1对应的寄存器为x10，他被翻译后对应的16进制数为0x0A。x0至x31分别对应0x00至0x1F，如用到别名，别名翻译后对应的数与其对应的寄存器一样。在之后的段落中我们将翻译过后的16进制数称之为指令/寄存器的“代号”。

当运行一段指令时，以“寻找素数”为例，给定的指令在图2.2展示，被翻译过后的机器码在图2.3展示，翻译出来的机器码每一行所对应16个字节，在行初有指令集所在的行数，比如第一行为0x000000，第二行为0x000010，以此类推。在运行时，给定指令集的第一个指令为`mov(imm)`因为`mov(imm)`对应的代号为0x31，所以如图2.2所展示，第一行的第一位为31。（图2.2只是翻译过后机器码的可视化，其中所有的1字节16进制数都被忽略掉了“0x”）。因为mov有0x30和0x31两个版本，分别对应寄存器之间赋值和将立即数赋值给寄存器的两种情况，所以当指令为`mov, rd, imm`的形式时，mov将自动被赋予代号0x31。当指令涉及到立即数或者地址的时候，因为instructionStack中的4字节大小的立即数和地址均为小端存储, 因此，31之后接着的00和00|11|11|11分别对应寄存器x0和立即数0x11111100。对于0x65的40|07|07|05，分别表示`add,rd,rs1,rs2`.

![img](file:///C:/Users/Lenovo/AppData/Local/Temp/msohtmlclip1/01/clip_image004.gif)

**ERISC指令完成情况 + Implementation Deep Dive**

我们实现了所有基本要求的操作指令：内存访问指令、栈操作指令、寄存器计算指令、控制指令。我们还对指令操作进行了记录和可视化。我们将实际进行操作的寄存器、内存和栈都包含在Storage.cpp和Storage.h的struct SimulatorStorageType中。

在程序的实现中, 我们充分利用了C++语言指针的特性, 通过指针类型的强制转化, 减小了编码量. 四个连续的字节既可以视为一个`uint`, 也可以视为 `char[4]`. 而在C语言中, 其理解的方式只取决于其指针的类型. 而C语言中, 任何类型的指针都可以通过 `void*` 这个桥梁进行转化. 当需要将一个`uint`写进栈里时, 可以将

在内存访问指令中，我们实现了内存数据加载`load`和内存数据存入`store`功能，这部分功能在MemoryInstruction.cpp和MemoryInstruction.h中实施。在`load`指令（标准形态为`load [rd], [rs]`）中，我们会将内存中地址为寄存器[rs]中值的数据加载到寄存器 [rd]中。根据InstructionStack，我们首先通过自定义变量variable `next`获取我们执行到的指令的位置，然后根据InstructionStack中包含的代号找到对应的rd（因为寄存器有32个，我们的代号可以直接作为索引找到对应的寄存器）。接着我们用"[ ]"将rd中间的地址提取出来作为需读取内存的位置。我们用同样的方式找到rs，从而可以将memory中的数据加载到寄存器当中。在store指令中，我们用了相同的原理实现将寄存器[rs]的值存入内存地址为寄存器[rd]中值的功能。

在栈操作指令中，我们实现了入栈push和出栈pop两个功能，这部分功能在MemoryInstruction.cpp和MemoryInstruction.h中实施。在push中（标准形态为`push [rs]`），我们会将寄存器[rs]的值压入栈。 我们同样用next找到了寄存器rs的位置并将其中的值读出。在压入栈的过程中，可以直接使用前述bithack的方法, 直接读出栈顶四个char对应的uint, pop 同理.

在寄存器计算指令中，我们实现了寄存器赋值，寄存器加、减、乘、除、取余、位与、位或（`add,sub,mul,div,rem,and,or`）。寄存器赋值在MemoryInstruction.cpp和MemoryInstruction.h中实施，寄存器的运算在arithmeticOperation.cpp和arithmeticOperation.h中实施。在寄存器赋值mov(标准形式为`mov [rd],[rs/imm]`)中，我们利用InstructionStack找到应被赋值的寄存器并赋值。

在控制指令中，我们实现了无条件跳转`jal`，条件跳转`beq/bne/blt/bge`，函数调用`call`，函数返回`ret`七个功能。这些功能在MemoryInstruction.cpp和MemoryInstruction.h中实施。在函数跳转中，以jal为例，我们也用int类型的指针将InstructionStack中的地址读取出来作为我们需要跳转到的指令的地址。条件跳转的跳转部分和jal是一样的原理，只不过我们会先对两个寄存器中的值进行相等、不等、大于等于或者小与等判断决定跳转与否。如果不进行跳转则直接根据InstructionStack运行下一个指令。函数调用 `call` 与`jal` 类似, 直接跳转到对应的行号即可, 同时对调用栈进行修改. 在函数返回ret中，我们会用int指针提取出实际操作的栈的行号作为程序下一轮运行的位置，也就是对行号进行了出栈。

在指令操作可视化的实现中，我们实施了程序结束符end和绘图指令`draw`，这两部分指令分别在outputTxt.cpp/outputTxt.h和drawBitmapImage.cpp/drawBitmapImage.h中体现。在绘图指令draw中，我们在内存访问指令、栈操作指令、寄存器计算指令、控制指令的实施中用drawBitmapImage.h里的struct record的`reg_write,reg_read,memory,stack`记录了程序从程序开始或上一次 draw 指令执行后到此次 draw 指令之间，程序对寄存器、内存、栈空间的访问情况，从而在drawBitmapImage.cpp具体实施。每次执行 draw 指令时，我们都会输出一个.bmp

文件。我们通过call outputTxt.cpp中的函数`outputTxt()`来实现程序结束符end，并输出一个result.txt 文件，其内容为整个程序运行结束后 32 个寄存器和内存里所有值的结果。

我们还对程序进行了可视化。文档中 5.1~5.4 的输入输出文件分别在附件 Task_1~Task_4 四个文件夹中, Task_5 是利用泰勒展开计算 $e$ 的近似值的程序及其输出.

综上，我们不仅完成了大作业要求中二、三、四、五节规定的全部任务，还实现了寄存器之间的浮点数运算。

优化

程序设计中很重要的一个理念是用有限的时间和空间解决问题。因此，本组在设计的过程中不仅对ERISC的功能全部进行了实现，还对复杂度进行了优化从而达到更短的运行时间和更高效的运行过程。原先我们在MemoryInstruction中用switch来判断每一种指令，但是由于种类过多，效率较差。我们首先把比较常用的指令放到前面，一定程度上减少了时间消耗。之后我们发现，可以利用函数指针，在常数时间内将指令代码转化为函数，大大减小了时间开销。

分工合作情况汇总：

| 负责文档/人员 | 崔庭瑀 | 蒋嘉铭 | 袁樱 | 王文琦 | 吴玮妍 |
|---|---|---|---|---|---|
| **CMakeList.txt** | main developer + debug | — | — | — | — |
| **GeneralFunctions.cpp** | main developer + debug | — | — | — | — |
| **GeneralFunctions.h** | main developer + debug | — | — | — | — |
| **InstructionParser.cpp** | main developer + debug | — | — | — | — |
| **InstructionParser.h** | main developer + debug | — | — | — | — |
| **MemoryInstruction.cpp** | Help debug | help debug +optimization | — | — | main developer + debug |
| **MemoryInstruction.h** | Help debug | — | — | — | main developer + debug |
| **Settings.h** | main developer + debug | — | — | — | — |

| 负责文档/人员 | 崔庭瑀 | 蒋嘉铭 | 袁樱 | 王文琦 | 吴玮妍 |
|---|---|---|---|---|---|
| **Storage.cpp** | main developer + debug | — | — | — | — |
| **Storage.h** | main developer + debug | — | — | — | — |
| **arithmeticOperation.cpp** | Help debug | main developer +help debug +optimization | — | — | — |
| **arithmeticOperation.h** | Help debug | main developer +help debug +optimization | — | — | — |
| **drawBitmapImage.cpp** | Help debug | Optimization | main developer + debug | Help debug | — |
| **drawBitmapImage.h** | Help debug | Optimization | main developer + debug | Help debug | Help debug |
| **in** | — | — | — | main developer(s) + debug | — |
| **input.risc** | main developer(s) + debug | Help debug | — | main developer(s) + debug | — |
| **main.cpp** | main developer(s) + debug | Help debug | — | — | main developer(s) + debug |
| **outputTxt.cpp** | — | — | main developer + debug | — | — |
| **outputTxt.h** | — | — | main developer + debug | — | — |
| **result3.txt** | main developer + debug | — | — | — | — |

| 负责文档/人员 | 崔庭瑀 | 蒋嘉铭 | 袁樱 | 王文琦 | 吴玮妍 |
| --- | --- | --- | --- | --- | --- |
| **文档** | 文档素材提供 | 帮忙检查 | — | — | 主要负责人 |
| **视频** | — | 主要负责人 | — | — | 主要负责人 |

总结

本组成员对C++语言中的指针了解比较深入, 利用指针类型转换和 `union` 语法, 不仅极大的减小了编码量, 同时利用 C++ 语言的 float, 实现了浮点数的运算, 并利用其计算出了 $e$ 的近似值. 但是还有一些不足, 例如常数因子比较大, 作图时文件IO占据的时间较多, 今后可以在这些方面继续研究优化方案.