

LoongArch汇编教学系统设计文档

1. 目标描述
2. 比赛题目分析和相关资料调研
 - 2.1 比赛题目分析
 - 2.2 相关资料调研
3. 系统框架设计
 - 3.1 设计逻辑框架
 - 3.2 设计开发框架
 - 3.3 设计物理框架
4. 开发计划
 - 4.1 需求分析和设计阶段
 - 4.2 核心功能开发阶段
 - 4.2.1 环境搭建
 - 4.2.1.1 JDK版本
 - 4.2.1.2 IDE环境
 - 4.2.2 LoongArch32精简指令集支持
 - 4.2.2.1 指令格式
 - 4.2.2.2 指令语句
 - 4.2.2.3 指令模拟
 - 4.2.2.4 伪指令实现
 - 4.2.3 中断和例外
 - 4.2.3.1 控制状态寄存器(CSR)
 - 4.2.3.1.1 CSR.ESAT
 - 4.2.3.1.2 CSR.ERA
 - 4.2.3.1.3 CSR.BADV
 - 4.2.3.2 中断例外前端显示
 - 4.2.4 用户界面开发
 - 4.3 调试和性能优化阶段
 - 4.3.1 调试
 - 4.3.1.1 单元调试
 - 4.3.1.2 集成调试
 - 4.3.1.3 系统调试
 - 4.3.1.4 用户验收调试
 - 4.3.2 性能优化
 - 4.3.2.1 代码优化
 - 4.3.2.2 并发与并行
 - 4.4 扩展功能开发阶段
 - 4.4.1 BHT simulator
 - 功能分析
 - 实现
 - 调试
 - 4.4.2 Instruction counter
 - 功能分析
 - 实现
 - 4.4.3 Instruction statistics
 - 功能分析
 - 实现
 - 4.4.4 LoongArch X-ray
 - 功能分析
 - 实现
5. 比赛过程中的重要进展
 - 5.1 项目确立
 - 5.2 主要指令集的实现
 - 5.3 LoongArch程序的可执行
 - 5.4 伪指令实现

5.5 扩展功能实现	
5.6 README和项目文档编写	
6. 系统测试情况	
6.1.指令集测试	
6.1.1 算术运算测试	
6.1.2 逻辑运算测试	
6.1.3 移位运算测试	
6.1.4 无条件跳转测试	
6.1.5 有条件跳转测试	
6.1.6 存储和读取测试	
6.1.7 伪指令测试	
2.程序编写测试	
6.2.1 斐波那契数列测试	
6.2.2 插入排序算法测试	
6.3 前端显示测试	
6.4 拓展功能测试	
6.4.1 BHT simulator	
6.4.2 LoongArch X-ray	
7. 遇到的主要问题和解决方法	
7.1 LoongArch指令的不熟悉	
问题描述:	
解决方法	
7.2 Mars编写时代码较为过时的语法修改	
问题描述	
解决方法	
7.3 Mips和LoongArch特性不同导致的冲突	
问题描述	
解决方法	
7.4 调试和性能优化的挑战	
问题描述	
解决方法	
8. 分工和协作	
8.1 指令集相关代码编写	
8.2 前端界面实现	
8.3 帮助文档的编写	
8.4 伪指令拓展	
8.5 扩展功能实现	
9. 提交仓库目录和文件描述	
9.1 仓库目录	
9.2 文件描述	
9.2.1 根目录文件	
9.2.2 lars目录文件	
9.2.3 docs目录文件	
10. 比赛收获	

LoongArch汇编教学系统设计文档

1. 目标描述

- **兼容性扩展：** 为Mars增添对LoongArch架构的支持，使其能够模拟和执行LoongArch汇编语言编写的程序。满足LoongArch架构教学和开发需求。
- **指令集实现：** 实现LoongArch指令集的核心部分，包含约50条基础指令。这些指令应该覆盖基本操作，以使用户能够编写和执行基础的汇编程序。
- **交互式开发体验：** 确保LoongArch架构下的Mars能够提供与MIPS相似的交互式开发体验。这包括代码编辑、调试、性能分析和即时反馈等功能。

- **教学适用性：**考虑到Mars在教育领域的广泛应用，新增添的LoongArch支持应该易于理解和使用，以适应不同层次的教学需求。提供相应的教学资源 and 示例代码，帮助学生更好地学习和掌握LoongArch架构。
- **代码可移植性：**鼓励开发者在实现LoongArch支持时考虑到代码的可移植性，使得从MIPS迁移到LoongArch的程序转换尽可能简单，降低学习曲线和迁移成本。
- **文档和指南：**提供全面的文档和使用指南，详细描述如何在Mars中使用LoongArch架构，包括快速入门、API参考和常见问题解答等。

2. 比赛题目分析和相关资料调研

2.1 比赛题目分析

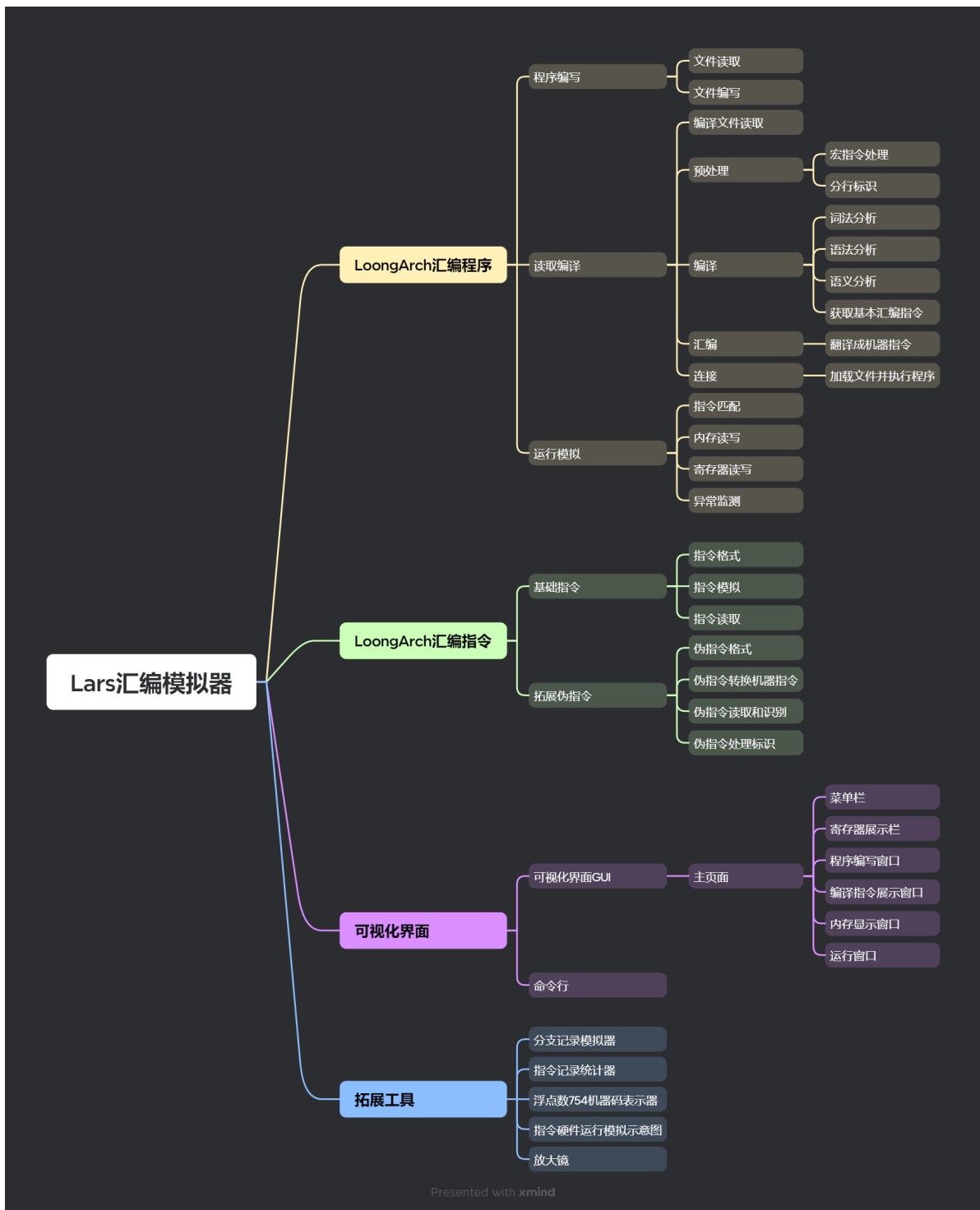
本次比赛的目标是为Mars增添对LoongArch架构的支持，并实现部分或全部LoongArch指令集，以支持Mars的大部分特性。

2.2 相关资料调研

- 《CPU 设计实战：LoongArch 版》¹ 教材结合 LoongArch 指令集，从计算机系统的角度阐述了汇编语言的理解和应用以及描述了计算机系统的组成及硬、软件的关系。
- 龙芯开源社区² 和相关Gitee项目为开发者提供了交流和合作的平台，促进了LoongArch生态的健康发展。
- LoongArch指令集具有RISC特征，采用定长指令和规整的编码格式，融合了MIPS和RISC-V的优点。
- 为实现LoongArch支持，开发者可以参考LoongArch指令集手册，了解指令集的详细信息和使用方法。

3. 系统框架设计

3.1 设计逻辑框架



3.2 设计开发框架

Lars沿用Mars的java.swing的图形库进行设计，依据该图形库，Lars的开发框架为MVC（Model View Controller）设计框架，即将Lars主要分成模型，视图，控制器三部分进行开发。

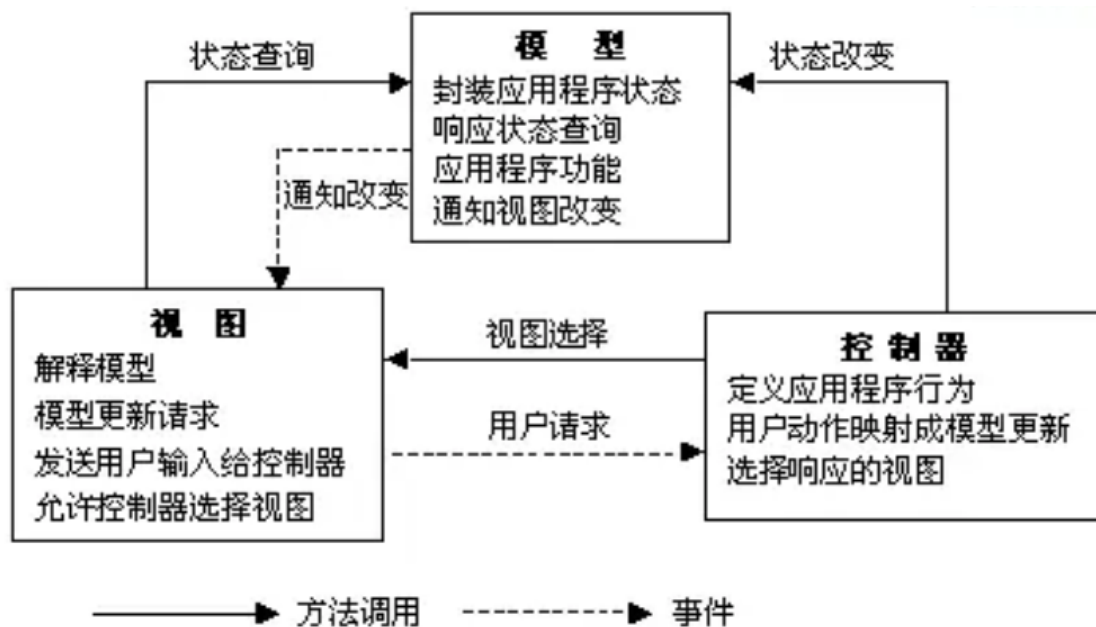


图1 MVC组件类型的关系和功能

3.3 设计物理框架

Lars作为本地运行的汇编模拟器，只涉及到用户本身的计算机。

4. 开发计划

4.1 需求分析和设计阶段

4.1.1 将支持MIPS的汇编模拟器Mars进行底层代码的修改，改成支持LoongArch32位精简指令集的Lars汇编模拟器。

4.1.2 对Mars的部分如数据通路模拟，分支跳转记录等拓展工具进行修改，将其修改成支持LoongArch32位精简指令集的拓展工具。

4.1.3 结合LoongArch32位精简指令集，修改Mars的帮助文档，注释等，便于其对LoongArch的汇编教学。

4.2 核心功能开发阶段

4.2.1 环境搭建

4.2.1.1 JDK版本

```
java version "20.0.1" 2023-04-18
Java(TM) SE Runtime Environment (build 20.0.1+9-29)
Java HotSpot(TM) 64-Bit Server VM (build 20.0.1+9-29, mixed mode, sharing)
```

4.2.1.2 IDE环境

```
IntelliJ IDEA 2023.1.3
```

4.2.2 LoongArch32精简指令集支持

4.2.2.1 指令格式

指令集的格式根据LoongArch32精简指令集文档信息进行实现，分为9种典型的指令集格式，但对部分指令格式与典型的指令集格式有所不同，所以在编码的过程中，为其增加额外的指令格式符。

表 1-1 龙芯架构 32 位精简版典型指令编码格式

	3 3 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0																																											
2R-type	opcode																rj				rd																							
3R-type	opcode												rk				rj				rd																							
4R-type	opcode								ra				rk				rj				rd																							
2RI8-type	opcode												I8								rj				rd																			
2RI12-type	opcode								I12																rj				rd															
2RI14-type	opcode						I14																								rj				rd									
2RI16-type	opcode				I16																										rj				rd									
1RI21-type	opcode						I21[15:0]																								rj				I21[20:16]									
I26-type	opcode						I26[15:0]																		I26[25:16]																			

开发过程中的指令格式代码实现：

```
public static final BasicInstructionFormat TWO_R_TYPE = new
BasicInstructionFormat();
public static final BasicInstructionFormat THREE_R_TYPE= new
BasicInstructionFormat();
public static final BasicInstructionFormat FOUR_R_TYPE= new
BasicInstructionFormat();

public static final BasicInstructionFormat TWO_R_I8_TYPE= new
BasicInstructionFormat();
public static final BasicInstructionFormat TWO_R_I12_TYPE= new
BasicInstructionFormat();
public static final BasicInstructionFormat TWO_R_I14_TYPE= new
BasicInstructionFormat();
public static final BasicInstructionFormat TWO_R_I16_TYPE= new
BasicInstructionFormat();

public static final BasicInstructionFormat TWO_R_I16_BRANCH_TYPE= new
BasicInstructionFormat();
public static final BasicInstructionFormat JUMP_TYPE= new
BasicInstructionFormat();

public static final BasicInstructionFormat ONE_R_I21_TYPE= new
BasicInstructionFormat();
public static final BasicInstructionFormat ONE_R_I20_TYPE= new
BasicInstructionFormat();
public static final BasicInstructionFormat I26_TYPE= new
BasicInstructionFormat();
```

```
public static final BasicInstructionFormat SYSCALL= new
BasicInstructionFormat();
```

4.2.2.2 指令语句

对于不同的指令集语句，Lars实现了对其：样例描述、机器码、语句作用、语句格式等性质的描述。

4.2.2.3 指令模拟

Lars沿用了Mars的编译和程序模拟运行的过程，并在其基础上对LoongArch的支持作了修改。

在编译的过程中，编译器会对文本进行拆分，区分.text域的指令和.data域的变量。然后对指令语句进行拆分，对语句的变量，编译器助记符等进行具体意义的转述，将其转换为初步的汇编语句，然后再对初步的汇编语句转换为二进制机器码，最后进行程序的组装和执行。

在指令的模拟过程中，单个指令在编写时就赋予了其功能的模拟过程，例如下图所示的指令 *add.w*

```
new BasicInstruction("add.w $t1,$t2,$t3",//指令语句描述
    "Addition without overflow : set $t1 to ($t2 add $t3) no
overflow",//指令功能描述
    BasicInstructionFormat.THREE_R_TYPE,//指令格式
    "00000000000100000 ccccc bbbbbb aaaaa",//机器码格式
    new SimulationCode() { //模拟实现
        public void simulate(ProgramStatement statement) throws
ProcessingException {
            int[] operands = statement.getOperands();
            int t2 = RegisterFile.getValue(operands[2]);
            int t1 = RegisterFile.getValue(operands[1]);
            int sum = t1 + t2;
            RegisterFile.updateRegister(operands[0], sum);
        }
    })
```

其获取原指令中的rk,rj中的值，然后将rk,rj中的值求和用以更新rd的值。实现了3R求和操作指令。

4.2.2.4 伪指令实现

基于Mars对MIPS的伪指令实现，Lars为了便于程序的开发和LoongArch汇编的教学，也对Lars添加了对伪指令的支持。

Lars的伪指令的具体实现方式是：

- 1.在编译阶段识别伪指令
- 2.对伪指令的操作进行实际指令集指令的拆分
- 3.实现拆分后的指令集指令以实现伪指令

Lars实现了部分汇编程序编写过程中常用的一些伪指令，如

la(Load Address) **li**(Load Immediate) **move**(Move value of register 1 to register 2)

nop(null instruction) **jirl**(Return the function call) ...

其中部分伪指令来自于LoongArch官方文档

官方文档所提供的伪指令：

Pseudo Instruction	Machine Instruction	Meaning
jr \$rd	jirl \$zero, \$rd, 0	Direct register jump
ret (jr \$ra)	jirl \$zero, \$ra, 0	Function return
bgt \$rj, \$rd, (si18 symbol)	blt \$rd, \$rj, (si18 symbol)	if(\$rj > \$rd)jump (si18 symbol)
bgtu \$rj, \$rd, (si18 symbol)	bltu \$rd, \$rj, (si18 symbol)	if(\$rj > \$rd)jump (si18 symbol)
ble \$rj, \$rd, (si18 symbol)	bge \$rd, \$rj, (si18 symbol)	if(\$rj <= \$rd)jump (si18 symbol)
bleu \$rj, \$rd, (si18 symbol)	bgeu \$rd, \$rj, (si18 symbol)	if(\$rj <= \$rd)jump (si18 symbol)
bltz \$rd, (si18 symbol)	blt \$rd, \$zero, (si18 symbol)	if(\$rd < 0)jump (si18 symbol)
bgtz \$rd, (si18 symbol)	blt \$zero, \$rd, (si18 symbol)	if(\$rd > 0)jump (si18 symbol)
blez \$rd, (si18 symbol)	bge \$zero, \$rd, (si18 symbol)	if(\$rd <= 0)jump (si18 symbol)
bgez \$rd, (si18 symbol)	bge \$rd, \$zero, (si18 symbol)	if(\$rd >= 0)jump (si18 symbol)
move \$rd, \$rj	or \$rd, \$rj, \$zero	Assign the value of \$rj to \$rd

Pseudo Instruction	Machine Instruction	Meaning
li.w \$rd, imm32	lu12i.w \$rd, si20	Load a 32-bit immediate
	ori \$rd, \$rd, si12	

部分指令沿用Mars的写法，但对其进行LoongArch化的修改。

4.2.3 中断和例外

4.2.3.1 控制状态寄存器(CSR)

由于LARS是软件层面的应用，所以只对汇编程序中可能出现的中断和例外进行处理。因此只选择三个CSR进行中断和例外的处理。

4.2.3.1.1 CSR.ESAT

记录进入中断和异常后的状态，利用Ecode和EsubCode记录中断和例外的类型。

例外状态寄存器定义

表 7-6 例外状态寄存器定义

位	名字	读写	描述
1:0	IS[1:0]	RW	两个软件中断的状态位。比特 0 和 1 分别对应 SWI0 和 SWI1。 软件中断的设置也是通过这两位完成，软件写 1 置中断写 0 清中断。
9:2	IS[9:2]	R	8 个硬中断（HWI0~HWI7）的中断状态位。高电平有效。 在线中断模式下，硬件仅是逐拍采样各个中断源并将其状态记录与此。此时对于所有中断须为电平中断的要求，是由中断源负责保证，并不在此处维护。
10	0	R0	保留域。读返回 0，且软件不允许改变其值。
11	IS[11]	R	定时器中断（TI）的中断状态位。高电平有效。在线中断模式下，硬件仅是逐拍采样各个中断源并将其状态记录与此。此时对于所有中断须为电平中断的要求，是由中断源负责保证，并不在此处维护。
12	IS[12]	R	核间中断（IPI）的中断状态位。高电平有效。在线中断模式下，硬件仅是逐拍采样各个中断源并将其状态记录与此。此时对于所有中断须为电平中断的要求，是由中断源负责保证，并不在此处维护。
15:13	0	R0	保留域。读返回 0，且软件不允许改变其值。
21:16	Ecode	R	例外类型一级编码。触发例外时，硬件会根据例外类型将表 7-7 中 Ecode 栏定义的数值写入该域。
30:22	EsubCode	R	例外类型二级编码。触发例外时，硬件会根据例外类型将表 7-7 中 EsubCode 栏定义的数值写入该域。
31	0	R0	保留域。读返回 0，且软件不允许改变其值。

Ecode编码定义

Ecode	EsubCode	例外代号	例外类型
0x0	0	INT	中断。
0x1	0	PIL	load 操作页无效例外

Ecode	EsubCode	例外代号	例外类型
0x2	0	PIS	store 操作页无效例外
0x3	0	PIF	取指操作页无效例外
0x4	0	PME	页修改例外
0x7	0	PPI	页特权等级不合规例外
0x8	0	ADEF	取指地址错例外
	1	ADEM	访存指令地址错例外
0x9	0	ALE	地址非对齐例外
0xB	0	SYS	系统调用例外
0xC	0	BRK	断点例外
0xD	0	INE	指令不存在例外
0xE	0	IPE	指令特权等级错例外
0xF	0	FPD	浮点指令未使能例外
0x12	0	FPE	基础浮点指令例外
0x1A-0x3E			保留编码
0x3F	0	TLBR	TLB 重填例外

4.2.3.1.2 CSR.ERA

在发生例外后，将发生的指令PC记录到CSR.ERA。

例外返回地址寄存器定义

表 7-8 例外返回地址寄存器定义

位	名字	读写	描述
GRLEN-1:0	PC	RW	触发例外时，硬件会将触发例外的指令的 PC 记录到这里。

4.2.3.1.3 CSR.BADV

在与地址相关的中断发生后，将错误的地址位置放入CSR.BADV寄存器。

出错虚地址寄存器定义

表 7-9 出错虚地址寄存器定义

位	名字	读写	描述
GRLEN-1:0	VAddr	RW	当触发 TLB 重填例外和地址错误相关例外时，硬件将出错的虚地址记录于此。

4.2.3.2 中断例外前端显示

在中断和例外发生的时候，更新CSR的内容，并且显示。

如下图是一个ADEM例外。

Name	Number	Value
\$7 (BADV)	7	0x00000002
\$5 (ESTAT)	5	0x00480001
\$6 (ERA)	6	0x00400000

同时在运行终端同时显示错误指令位置和中断类型。

```
Assemble: assembling C:\Users\18316\IdeaProjects\Lars\XraysTest.asm
Assemble: operation completed successfully.
Go: running XraysTest.asm
Error in C:\Users\18316\IdeaProjects\Lars\XraysTest.asm line 7: Runtime exception at 0x00400000: fetch address not aligned on word boundary 0x00000002
Go: execution terminated with errors.
```

4.2.4 用户界面开发

开发用户友好的界面，包括代码编辑器和模拟器控制面板。

4.3 调试和性能优化阶段

4.3.1 调试

4.3.1.1 单元调试

对Lars的程序最小可测单位进行单元的调试，对主要修改代码功能的函数方法和类定义进行分析调试，测试代码的鲁棒性和可行性。

4.3.1.2 集成调试

在单元调试完成验收的基础上，对各单元的集成单位进行调试。将部分模块进行组合的测试，比如在Lars中，对指令集所相关的指令执行模拟，指令编译，指令编写等模块进行集成的测试，从而发现不同模块中是否存在接口的错误和冲突。

4.3.1.3 系统调试

在集成调试完成验收的基础上，对各模块进行组合测试。在本项目中，主要分成前端和后端的系统测试：后端中主要测试汇编程序编译，执行，输出以及硬件的模拟过程代码的测试；前端中主要测试GUI的显示以及前后端接口的测试。

在该调试过程中，事先编写一定数量的LoongArch汇编程序进行系统的测试，在测试的过程中关注指令的执行情况，程序的运行状态，实时关注前后端是否有接口冲突和错误。

4.3.1.4 用户验收调试

在所有代码测试完成的基础上，程序进行试运行，邀请一些学习汇编语言的学生和教师进行试用。在用户的验收反馈中获取程序在代码测试过程中未发现的问题，从而进行及时的修改和更正。

4.3.2 性能优化

4.3.2.1 代码优化

对程序进行代码层面的优化，减少不必要的运算，运用算法和数据结构以减少内存的分配和时间的占用。

比如在指令集的遍历中，可以事先对指令数组进行哈希化，在哈希表中查询指令，可以优化指令的编译速度。

4.3.2.2 并发与并行

对程序进行并发和并行的优化

4.4 扩展功能开发阶段

扩展功能的开发基于Java的swing图形库，并且沿用了AbstractMarsToolAndApplication抽象类作为此类功能与用户编写的LoongArch汇编程序进行接口连接，监听程序执行，使拓展功能工具能够实时根据指令运行而进行相应功能的运行。

4.4.1 BHT simulator

功能分析

在程序执行的过程中，记录分支指令的执行，对每一条分支指令进行跳转目标和分支的次数、准确性的记录。

实现

在汇编程序运行的过程中，利用指令集的格式判断当前指令是否属于分支指令，若是分支指令，则读取该指令的目标地址，并且记录跳转地址；进而判断跳转地址是否被占用，若未被占用，则跳转成功。

调试

4.4.2 Instruction counter

功能分析

读取当前指令，根据指令的类型进行计数，并且计算指令在已执行指令中数目的占比，然后显示。

实现

在汇编程序运行的过程中，利用指令集的格式判断当前指令属于何种指令，然后将对应的指令数目增加并显示。

4.4.3 Instruction statistics

功能分析

读取当前指令，根据指令所调用的硬件方式进行计数。

实现

在汇编程序运行的过程中，根据当前指令所调用的主要器件进行分析，然后将对应的器件调用增加并显示。

4.4.4 LoongArch X-ray

功能分析

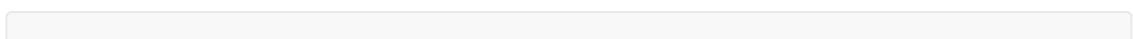
根据不同的指令的数据通路，实现指令数据通路的动画模拟；并且拆分指令机器码，显示指令的具体实现和编码。

实现

1. 数据通路动画：

数据通路动画由以下部分组成：在xml中定义线段顶点；读取线段顶点的坐标和绘图颜色数据；根据线段顶点的坐标数据，绘制在背景图相应坐标绘制相应的线段。

顶点的实现样例：



```

<num_vertex>1</num_vertex>
<name>*inst</name>
<init>512</init>
<end>470</end>
<color_Itype>255#255#0</color_Itype>
<color_Rtype>255#255#0</color_Rtype>
<color_Jtype>255#255#0</color_Jtype>
<color_LOADtype>255#255#0</color_LOADtype>
<color_STOREtype>255#255#0</color_STOREtype>
<color_ONE_R_I20_type>255#255#0</color_ONE_R_I20_type>
<color_B>255#255#0</color_B>
<color_BL>255#255#0</color_BL>
<color_BRANCHtype>255#255#0</color_BRANCHtype>
<other_axis>85</other_axis>
<isMovingXaxis>false</isMovingXaxis>
<target_vertex>2#3</target_vertex>
<is_text>false</is_text>

```

2. 指令集拆分:

指令集拆分游由以下部分组成: 对指令集拆分成九种格式(Load,Store,Branch,Jump,3_R等);根据不同类型指令的机器码拆分成操作码段和立即数段或寄存器段等; 根据相应的拆分, 显示该码段所对应的具体含义(操作码、立即数、寄存器等); 并且翻译指令的具体实现作用。

5. 比赛过程中的重要进展

5.1 项目确立

在LoongArch架构的学习过程中, 我们发现LoongArch指令集暂时没有一个方便用户教学学习, 编写LA汇编程序的软件。对此, 我们小组决定开发一个LoongArch 32汇编教学系统, 用于高校日常的汇编教学。并且提供相应的拓展功能辅助教学和学习。

5.2 主要指令集的实现

经过几周的工作后, 我们初步完成了指令集的编写。实现的指令集主要为算术运算指令、逻辑运算指令、移位运算指令、转移指令和存储读取指令。这些指令基本囊括我们日常教学中的所有指令。指令的实现通过Java代码编写。以下是部分指令实现代码:

```

//I12指令addi.w实现
instructionList.add(
    new BasicInstruction("addi.w $t1,$t2,-1000",
        "Addition without overflow : set $t1 to ($t2 add $t3) no overflow",
        BasicInstructionFormat.TWO_R_I12_TYPE,
        "0000001001 cccccccccc bbbbbb aaaaa",
        new SimulationCode() {
            public void simulate(ProgramStatement statement) throws
ProcessingException {
                int[] operands = statement.getOperands();
                int t2 = operands[2];
                int t1 = RegisterFile.getValue(operands[1]);
                int sum = t1 + t2;
                RegisterFile.updateRegister(operands[0], sum);
            }
        }
    )

```

```
});
```

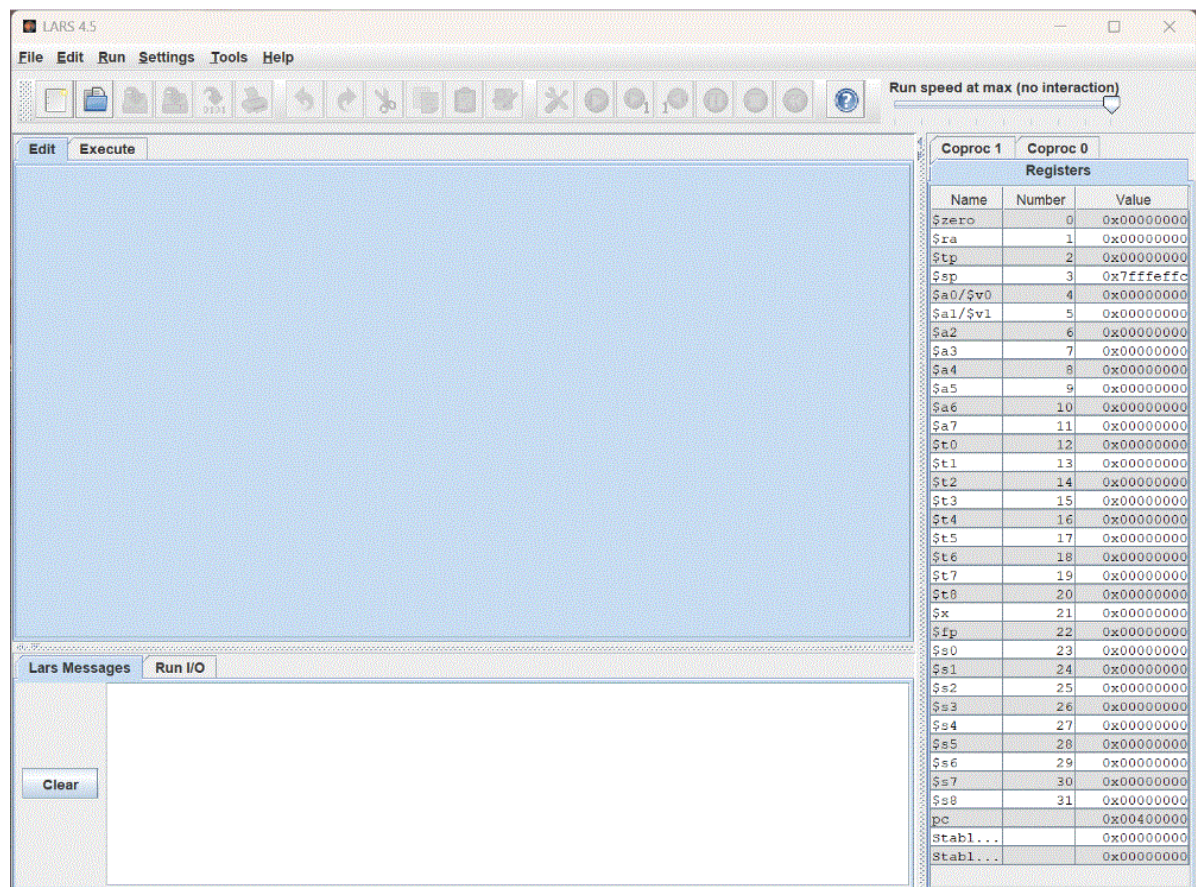
```
//Branch指令beq实现
instructionList.add(
    new BasicInstruction("beq $t1,$t2,label",
        "Branch if equal : Branch to statement at label's address if
        $t1 and $t2 are equal",
        BasicInstructionFormat.TWO_R_I16_BRANCH_TYPE,
        "010110 cccccccccccccccc aaaaa bbbbbb",
        new SimulationCode() {
            public void simulate(ProgramStatement statement) throws
ProcessingException {
                int[] operands = statement.getOperands();

                if (RegisterFile.getValue(operands[0]) ==
RegisterFile.getValue(operands[1])) {
                    processBranch(operands[2]);
                }
            }
        });
```

5.3 LoongArch程序的可执行

在基础指令集实现后，我们开始了对指令集的系统性测试，为此用LoongArch指令集编写了斐波那契数列和插入排序这两个小程序进行测试，并且成功在LARS上运行成功。

以下是斐波那契数列的实现示例图：



5.4 伪指令实现

在完成指令集的编写之后，我们打算更进一步。众所周知，汇编指令记忆起来是有一定困难的，并且汇编指令偏向底层，往往一个简单的功能就需要数条指令来编写实现，这大大提高了LoongArch 32汇编程序的编写难度，不利于汇编的开发学习。所以，我们沿袭MARS伪指令的做法，对LARS也提供了伪指令的支持，用户可以根据自身需要来打开或关闭伪指令支持。通过伪指令，我们可以很方便的记忆指令，实现同一功能，需要的汇编代码行数也大大减少，是程序编写更加的便捷。

5.5 扩展功能实现

为了让LARS作为教学工具能更加方便地提供教学和学习，我们对LARS的拓展功能进行了开发。实现了诸如分支记录，指令统计，浮点机器码转换，内存访问可视化，以及数据通路展示等多项拓展功能。[数据通路展示 6.4.2节](#)

5.6 README和项目文档编写

经过许久的开发后，LARS的功能和操作逐渐完善。为了帮助用户快速上手，了解LARS的功能，我们开始了README的编写，并且编写了更详细的项目文档来介绍LARS的功能。

6. 系统测试情况

6.1.指令集测试

6.1.1 算术运算测试

```
addi.w $a0,$a0,2047  #a0 = 0x000007ff = 2047
addi.w $a1,$a1,-2048 #a1 = 0xffffffff800
add.w $a2,$a1,$a0    #a2 = 0xfffffffffff
addi.w $a3,$a3,2046  #a3 = 0x000007fe = 2046
sub.w $a4,$a0,$a3    #a4 = 0x00000001 = 2047 - 2046 = 1
mul.w $a5,$a0,$a4    #a5 = 0x000007ff = 2047 * 1 = 2047
addi.w $a6,$a6,1023  #a6 = 0x000003ff = 1023
div.w $a7,$a3,$a6    #a7 = 0x00000002 = 2046 / 1023 = 2
```

6.1.2 逻辑运算测试

```
#首先设置$a0 = 0x0000f0f0
#首先设置$a1 = 0x00f00ff0
and $a2,$a0,$a1    #a2 = a0 and a1 = 0x000000f0
or $a3,$a0,$a1     #a3 = a0 or a1 = 0x00f0fff0
nor $a4,$a2,$a3    #a2 = a0 nor a1 = 0xff0f000f
xor $a5,$a0,$a1    #a2 = a0 xor a1 = 0x00f0ff00
andi $a6,$a0,2047  #a2 = a0 and 0x000007ff = 0x000000f0
ori $a7,$a0,2047   #a2 = a0 or 0x000007ff = 0x0000f7ff
```

6.1.3 移位运算测试

```

addi.w $t0,$t0,2047    #设置t0的初值t0 = 0x000007ff
addi.w $a0,$a0,3       #设置偏移量a0 = 3
sll.w $t1,$t0,$a0      #逻辑左移3位, 得t1 = 0x00003ff8
srl.w $t2,$t1,$a0      #逻辑右移3位, 得t2 = 0x000007ff
sra.w $t3,$t0,$a0      #算术右移3位, 得t1 = 0x000000ff

slli.w $t4,$t0,3       #逻辑左移3位, 得t1 = 0x00003ff8
srli.w $t5,$t4,3       #逻辑右移3位, 得t2 = 0x000007ff
srai.w $t6,$t0,3       #算术右移3位, 得t1 = 0x000000ff

```

6.1.4 无条件跳转测试

```

.text
addi.w $t0,$t0,1    #设置t0的初值t0 = 1
addi.w $t1,$t1,1    #设置t1的初值t1 = 1
b part1             #跳转到part1

part3:
    addi.w $t0,$t0,3    #t0 = t0 + 3 = 5
    b exit1             #跳转到exit1

part2:
    addi.w $t1,$t1,2    #t1 = t1 + 2 = 3
    jirl $t2,$ra,0      #返回part1

part1:
    addi.w $t0,$t0,1    #t0 = t0 + 1 = 2
    bl part2
    addi.w $t0,$t0,1    #t0 = t0 + 1 = 3
    addi.w $t0,$t0,1    #t0 = t0 + 1 = 4
exit1:

```

6.1.5 有条件跳转测试

```

.text
addi.w $t0,$t0,1    #设置t0的初值t0 = 1
addi.w $t1,$t1,1    #设置t1的初值t1 = 1
beq $t0,$t1,part1   #t0和t1相等就跳转到part1

part3:
    addi.w $t0,$t0,3    #t0 = t0 + 3 = 5
    bne $t0,$t1,exit1   #t0和t1不相等就跳转到exit1

part1:
    addi.w $t0,$t0,1    #t0 = t0 + 1 = 2
    bge $t0,$t1,part2   #如果t0大于t1就跳转到part2

part2:
    addi.w $t1,$t1,2    #t1 = t1 + 2 = 3
    blt $t0,$t1,part3   #如果t0小于t1就跳转到part3

exit1:

```


6.1.6 存储和读取测试

```
.data
arrays:          #定义一个数组arrays[2]
.word 0:2

.text
la $a0, arrays    #将数组的地址加载到a0
addi.w $t0, $t0, 1 #设置t0的初值t0 = 1
addi.w $t1, $t1, 2 #设置t1的初值t1 = 2

st.w $t0, $a0, 0   #将t0中的值存储到arrays[0]中
st.w $t1, $a0, 4   #将t0中的值存储到arrays[1]中

ld.w $t2, $a0, 0   #将arrays[0]中的值加载到t2中 t2 = 1
ld.w $t3, $a0, 4   #将arrays[1]中的值加载到t3中 t3 = 2
```

6.1.7 伪指令测试

```
.data
fibs: .word 0 : 12

.text
li.w $t2, 0x0f0f0f0f    #加载32位立即数 t2 = 0x0f0f0f0f

la $t3, ($t0)            #将t0中的地址加载到t3中          t3= 0xffffffffc
la $t4, -4               #将有符号立即数看作一个地址加载到t4中 t4 = 0xffffffffc
la $t5, 0x0f0f0f0f       #将32位立即数看作一个地址加载到t5中 t5 = 0x0f0f0f0f
la $t6, fibs             #将fibs的地址加载到t6中          t6 = 0x10010000

st.w $t0, ($t6)          #将t0寄存器中的数据储存在t6的地址中 t6中是fibs第一个地址 fibs[0] =
0xffffffffc
st.w $t0, -256           #将t0寄存器中的数据储存在-256对应的地址中 (-256) =
0xffffffffc
st.w $t1, 0x10010004     #将t1寄存器中的数据储存在fibs第二个地址中 fibs[1] =
0x00000f0f
st.w $t2, 8($t6)         #将t2寄存器中的数据储存在fibs第三个地址中 fibs[2] =
0x0f0f0f0f

ld.w $a0, ($t6)          #将fibs[0]地址处的值赋给a0        a0 = 0xffffffffc
ld.w $a1, -256           #将-256对应的地址中数据赋给a1     a1 = 0xffffffffc
ld.w $a2, 4($t6)         #将fibs[1]中的值赋给a2            a2 = 0x00000f0f
```

2.程序编写测试

6.2.1 斐波那契数列测试

```
.data
fibs: .word 0:12        # 定义一个存储斐波那契数的数组，初始化为12个元素的0
size: .word 12          # 定义一个存储数组大小的变量，值为12

.text
la $t0, fibs            # 将fibs数组的起始地址加载到寄存器$t0
la $t5, size            # 将size变量的地址加载到寄存器$t5
ld.w $t5, $t5, 0        # 将size变量的值加载到寄存器$t5（数组的小）
li.w $t2, 1             # 将立即数1加载到寄存器$t2
```

```

        st.w $t2, $t0, 0      # fibs[0] = 1
        st.w $t2, $t0, 4      # fibs[1] = 1
        addi.w $t1, $t5, -2    # 将size减2存储到寄存器$t1（作为循环计数器）

loop:
    ld.w $t3, $t0, 0          # $t3 = fibs[n]
    ld.w $t4, $t0, 4          # $t4 = fibs[n+1]
    add.w $t2, $t3, $t4        # $t2 = fibs[n] + fibs[n+1]
    st.w $t2, $t0, 8          # Store fibs[n+2] = fibs[n] + fibs[n+1]
    addi.w $t0, $t0, 4         # 将$t0的值加4，指向数组的下一个元素
    addi.w $t1, $t1, -1        # 将循环计数器$t1减1
    bgtz $t1, loop            # 如果循环计数器$t1大于0，跳转到loop继续循环
    la $a0, fibs               # 将fibs数组的起始地址加载到寄存器$a0
    add.w $a1, $zero, $t5      # 将$t5（数组大小）加载到寄存器$a1
    b print                    # 跳转到print标签

# print标签下的代码用于打印斐波那契数的标题
.data
    head: .asciz "The Fibonacci numbers are:"    # 定义斐波那契数的标题字符串
    space: .asciz " /n"                          # 定义一个字符串，包含一个空格
.text

print:
    add.w $t0, $zero, $a0      # 将$a0的值（fibs数组的地址）加载到寄存器$t0
    add.w $t1, $zero, $a1      # 将$a1的值（数组大小）加载到寄存器$t1

# out标签下的代码用于逐个打印斐波那契数
out:
    la $a0, head               # 将标题字符串的地址加载到寄存器$a0
    li.w $a7, 4                # 设置系统调用号为4（打印字符串）
    syscall 0
    ld.w $a0, $t0, 0           # 将$t0指向的数组元素的值加载到寄存器$a0
    li.w $a7, 1                # 设置系统调用号为1（打印整数）
    syscall 0
    la $a0, space              # 将空格字符串的地址加载到寄存器$a0
    li.w $a7, 4                # 设置系统调用号为4（打印字符串）
    syscall 0
    addi.w $t0, $t0, 4         # 将$t0的值加4，指向数组的下一个元素
    addi.w $t1, $t1, -1        # 将$t1的值减1
    bgtz $t1, out              # 如果$t1大于0，跳转到out继续循环
    b exit                     # 如果所有斐波那契数都已打印完毕，则跳转到exit标签退出程序

exit:
    li.w $a7, 10               # 设置系统调用号为10（退出程序）
    syscall 0                  # 执行系统调用

```

6.2.2 插入排序算法测试

```

.data
    arrays: .word 0:12        # 定义数组arrays，包含12个4字节的元素
.text
    la $a6, arrays             # 将数组arrays的地址加载到寄存器$a6

getLength:
    li.w $a7, 5                # 设置系统调用号为5，这里假设5是读取数组长度的系统调用
    syscall 0                  # 执行系统调用

```

```

    move $t0, $a0          # 将数组长度存储到$t0
    li.w $a2 0             # 设置计时器初始值
    move $a3 $a6           # 设置偏移量初始值

getNumber:
    beq $a2, $t0, deliverParam # 如果读取完成 就进入排序
    li.w $a7 5             # 读取数据
    syscall 0
    st.w $a0, $a3, 0       # 将数组存储到内存中
    addi.w $a3, $a3, 4     # 偏移量+4
    addi.w $a2, $a2, 1     # 计数+1
    b getNumber           # 继续循环

deliverParam:
    move $a0, $a6          # 将数组的地址加载到$a0
    move $a1, $t0          # 将数组长度加载到$a1
    b sort                # 跳转到sort

#s0==i, s1==j, s2==v 地址初始值, s3==Length
sort:
    addi.w $sp, $sp, -20   # 为存储寄存器值在栈上分配空间
    st.w $ra, $sp, 16     # 将返回地址$ra压栈保存
    st.w $s3, $sp, 12     # 将$s3压栈保存
    st.w $s2, $sp, 8
    st.w $s1, $sp, 4
    st.w $s0, $sp, 0

    move $s2, $a0          # 将数组地址$a0保存到$s2
    move $s3, $a1
    li.w $s0, 0            # 初始化外层循环变量i为0

for1tst:    #第一层循环
    slt $t0, $s0, $s3      # if $s0 < $s3 (i < n) $t0 = 1
    beq $t0, $zero, exit1  # if $t0 == 0, 即外层循环结束, 则跳转到exit1
    addi.w $s1, $s0, -1    # j = i - 1
    #记录当前的key=array[i]
    slli.w $t5, $s0, 2     # $t5 = $s0 * 4
    add.w $t6, $t5, $s2
    ld.w $t7, $t6, 0

for2tst:    #第二层循环
    slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
    bne $t0, $zero, exit2  # if $s1 < 0, 即内层循环结束, 则跳转到exit2
    slli.w $t1, $s1, 2     # $t1 = j * 4
    add.w $t2, $s2, $t1    # $t2 = v + (j * 4)
    ld.w $t3, $t2, 0       # $t3 = v[j]
    ld.w $t4, $t2, 4       # $t4 = v[j + 1]
    slt $t0, $t4, $t3      # if $t4 < $t3 (v[j + 1] < v[j]) $t0 = 1
    beq $t0, $zero, exit2  # if $t4 >= $t3, 即无需交换, 则跳转到exit2
    bl swap                # 调用swap过程进行交换
    addi.w $s1, $s1, -1    # j--
    b for2tst             # 继续内层循环

swap:
    st.w $t3, $t2, 4       # 将$t3存储到v[j + 1]
    st.w $t4, $t2, 0       # 将$t4存储到v[j]
    jr $ra                 # 返回到调用swap的地方

```

```

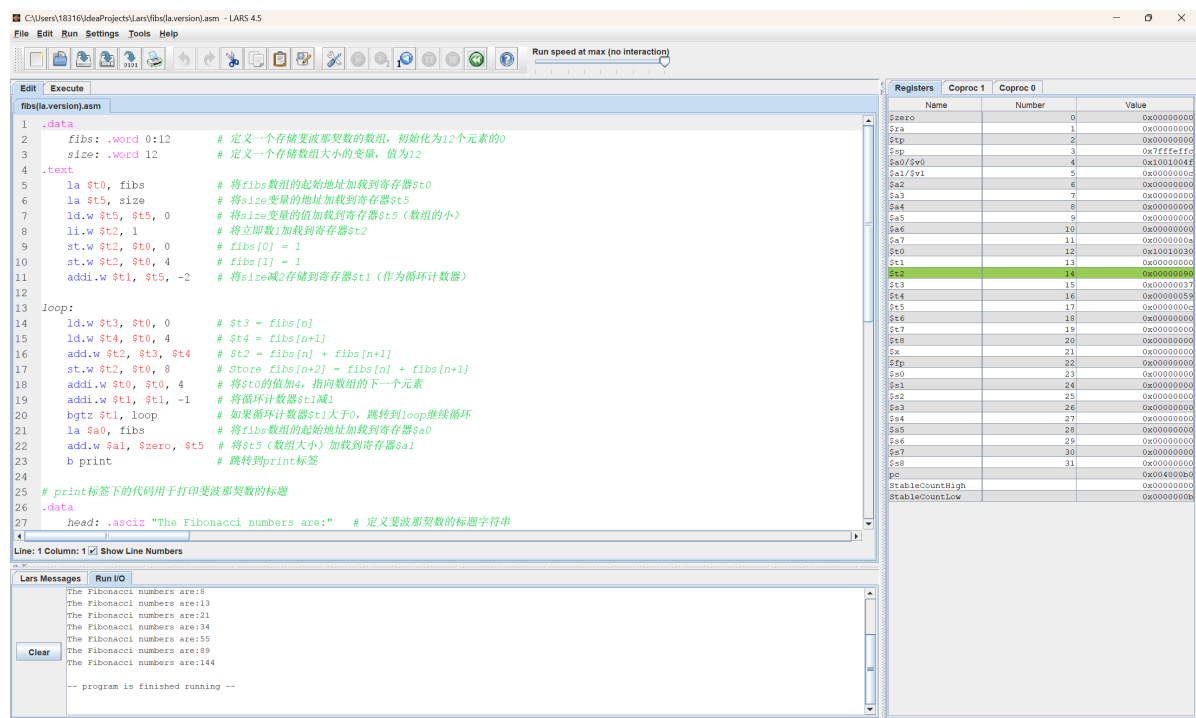
exit2:
    addi.w $s0, $s0, 1      # i++
    slli.w $t5, $s1, 2
    add.w $t5, $t5, $s2
    st.w $t7, $t5, 4        # 将key存储到array[j+1]
    b for1tst              # 继续外层循环

exit1:
    #还原现场
    ld.w $s0, $sp, 0        # 从栈中恢复$s0
    ld.w $s1, $sp, 4
    ld.w $s2, $sp, 8
    ld.w $s3, $sp, 12
    ld.w $ra, $sp, 16
    addi.w $sp, $sp, 20
    #jirl $zero,$ra, 0      # return to calling routine

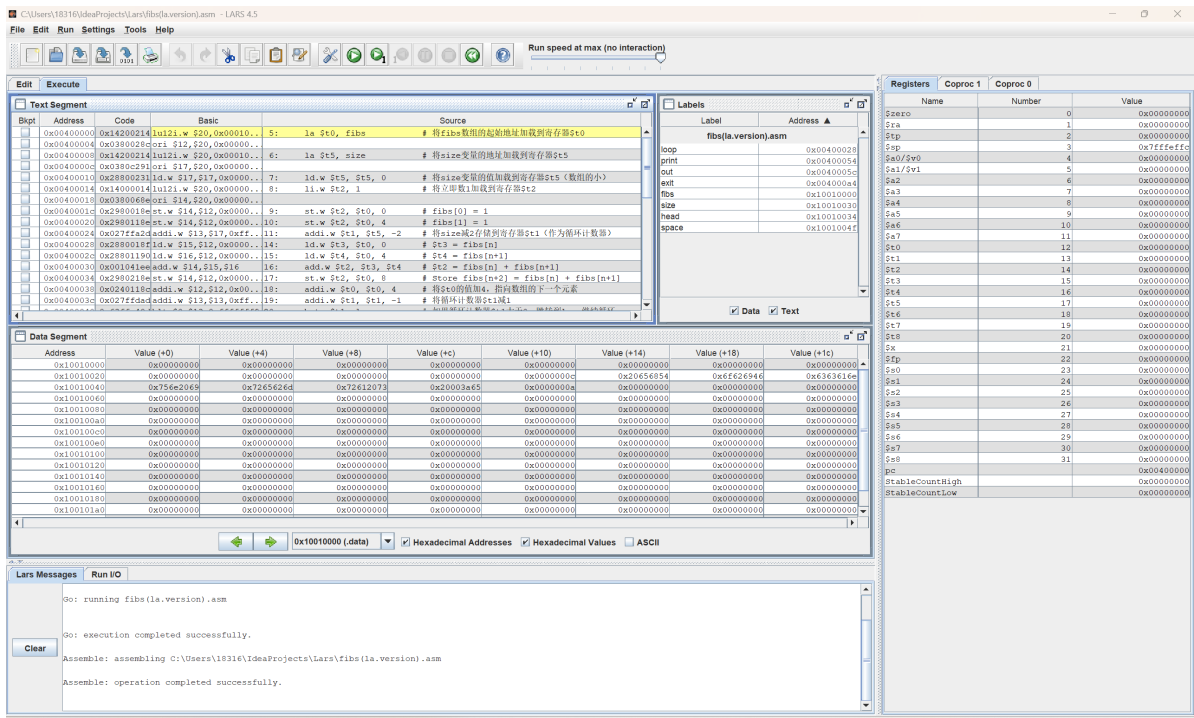
```

6.3 前端显示测试

对斐波那契数列测试程序的编写界面前端



对斐波那契数列程序的编译后界面前端

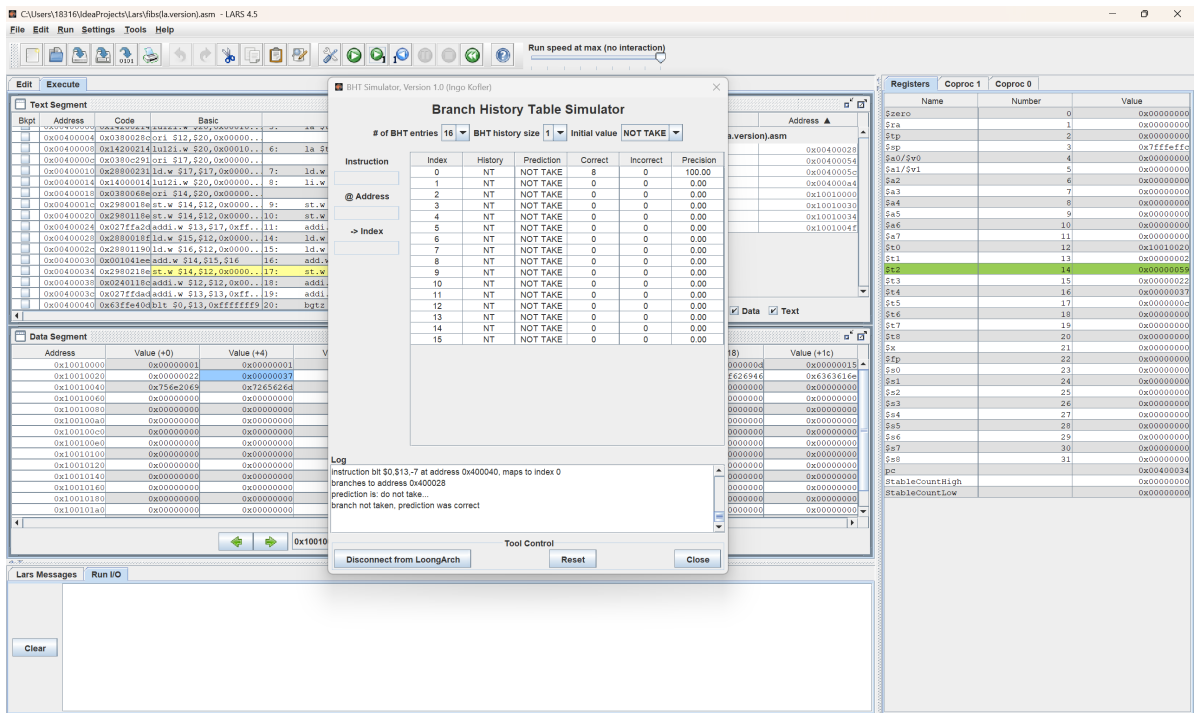


6.4 拓展功能测试

仅对实现和展示较为复杂的拓展功能进行测试展示。

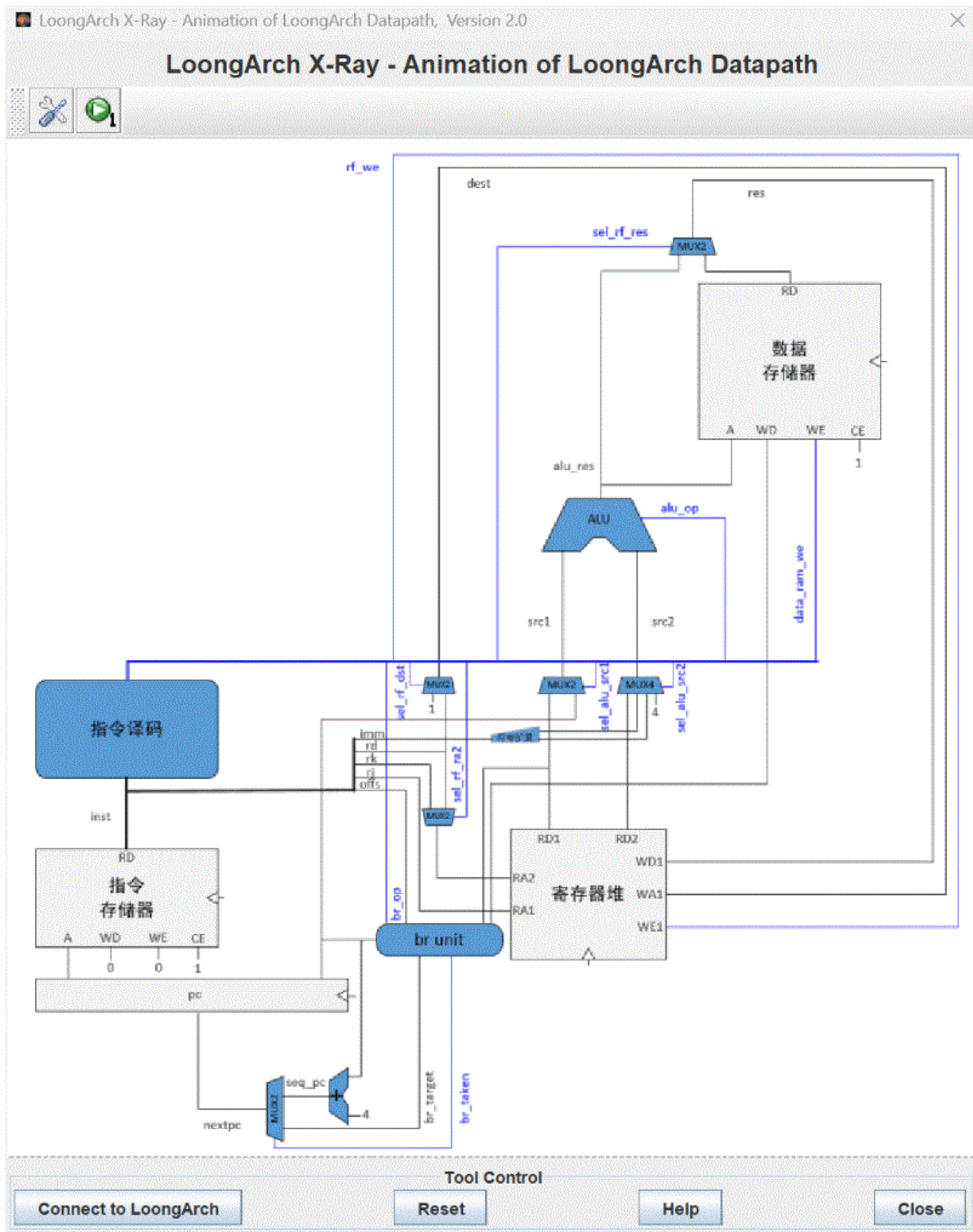
6.4.1 BHT simulator

实现了对分支跳转的记录。



6.4.2 LoongArch X-ray

实现对不同硬件调用指令的数据通路展示。



7. 遇到的主要问题和解决方法

7.1 LoongArch指令的不熟悉

问题描述：

LoongArch作为一种新的处理器架构，开发团队对于LoongArch指令集不够熟悉，这影响了开发进度和代码编写的准确性。

解决方法

- **深入学习**：团队成员投入时间深入学习LoongArch指令集架构，包括：
 - 阅读官方文档
 - 参与在线课程
- **实践操作**：通过编写和测试简单的汇编程序来加深对LoongArch指令集的理解。

7.2 Mars编写时代码较为过时的语法修改

问题描述

Mars模拟器的原代码基于较早版本的Java编写，部分语法和库可能不再被现代Java版本所支持。

解决方法

- **代码审查**：对现有代码进行详细的审查，明确哪些部分需要更新。
- **逐步更新**：逐步替换过时的语法和库，以确保兼容性和性能。
- **测试**：更新后进行详尽的测试，以确保模拟器的功能和性能没有受到影响。

7.3 Mips和LoongArch特性不同导致的冲突

问题描述

Mars原本支持MIPS架构，而LoongArch与MIPS是两种不同的处理器架构，在某些特性上存在差异，这导致了代码冲突和功能不兼容的问题。

解决方法

- **架构对比**：详细对比MIPS和LoongArch的架构差异，列出二者不兼容的特性。
- **代码处理**：对冲突的代码进行模块化处理，实现LoongArch架构特定的代码分支。

7.4 调试和性能优化的挑战

问题描述

随着Lars对LoongArch架构功能的持续增加，导致代码调试逐渐困难，性能逐渐降低。

解决方法

- **模块化调试**：对指令集所相关的指令进行模块化集成测试，对扩展功能进行系统测试，实时关注前端是否有接口冲突和错误。
- **代码优化**：对代码进行优化，通过改进算法和数据结构以提高性能。

8. 分工和协作

项目以队伍Mars2Lars_NUAAers的成员为主，在指导老师的支持和帮助下分工协作完成。

8.1 指令集相关代码编写

初步编写与调试：熊浩然、杨余松

验收测试与修改：冼俊杰

8.2 前端界面实现

核心代码分析与编写：冼俊杰、熊浩然

优化与测试：冼俊杰

8.3 帮助文档的编写

帮助文档编撰：杨余松

帮助文档验收与矫正：冼俊杰

8.4 伪指令拓展

伪指令编写：杨余松

伪指令测试与修改：冼俊杰

8.5 扩展功能实现

拓展功能代码编写：冼俊杰

拓展功能相关文档编写：熊浩然

拓展功能测试与修改：冼俊杰

9. 提交仓库目录和文件描述

9.1 仓库目录

<https://gitlab.eduxiji.net/T202410287992756/project2210132-236065.git>

9.2 文件描述

9.2.1 根目录文件

表1:

文件/目录	描述
/.idea	存放idea项目配置文件
/help	存放LARS帮助文件中的文档
/images	存放LARS所用到的图片、图标资源
/lars	LARS的主要实现代码
/simulator	对LARS汇编程序进行运行模拟的实现
/tools	LARS拓展工具功能的实现
/META-INF	打包Jar包时自动生成的配置文件
/out	运行文件
/test	存放指令集测试汇编程序

表2(续表1):

文件/目录	描述
/docs	存放项目文件
Lars.jar	Lars打包的Jar包文件
Lars.java	Lars的启动主类
LoongArchXRayOpcode.xml	存放X-ray工具相关的信息
PseudoOps.txt	定义伪指令的文件

9.2.2 lars目录文件

文件/目录	描述
/assembler	实现LoongArch 32汇编程序编译过程的代码
/loongarch	实现LoongArch 32架构的核心代码，包含硬件模拟，指令集模拟
/simulator	实现LoongArch 32汇编程序运行过程的代码
/tools	实现拓展工具的代码
/util	自定义便于代码编写的方法类代码
/venus	前端GUI实现代码

9.2.3 docs目录文件

文件/目录	描述
/picture	项目文件所需图片
proj_docs.md	项目文件

10. 比赛收获

在本次比赛中，我们小组学习了许多关于LoongArch架构指令集的相关知识，深入了解和研究了LoongArch指令集的实现和作用，这让我们对国产CPU架构有了更深一层的理解，作为计算机类专业的大学生，我们对国产CPU和国产架构有了更深入的认识，也让我们对CPU的自主化研究有了自信和动力，使我们越发想要投入到中国计算机领域的建设中。

同时，本次的项目开发是小组内的成员协作完成的，这既锻炼了我们的团队合作能力，也让我们对协作分工有了更深入的认识，提高了合作效率。

我们在比赛中收获累累硕果，这将是未来学习和工作生活中重要的一次锻炼和学习的机会。

1. 汪文祥 邢金璋 等著 [🔗](#)

2. 龙芯开源社区网址: <http://www.loongnix.cn/> [🔗](#)