**2022年全国大学生信息安全竞赛**

**作品报告**

**作品名称： 针对LoongArch的反编译器**

**电子邮箱： 1370723334@qq.com**

**提交日期： 2022.6.20**

填写说明

1. 所有参赛项目必须为一个基本完整的设计。作品报告书旨在能够清晰准确地阐述（或图示）该参赛队的参赛项目（或方案）。

2. 作品报告采用A4纸撰写。除标题外，所有内容必需为宋体、小四号字、1.5倍行距。

3. 作品报告中各项目说明文字部分仅供参考，作品报告书撰写完毕后，请删除所有说明文字。(本页不删除)

4. 作品报告模板里已经列的内容仅供参考，作者可以在此基础上增加内容或对文档结构进行微调。

5. 为保证网评的公平、公正，作品报告中应避免出现作者所在学校、院系和指导教师等泄露身份的信息。

**目 录**

[摘要 4](#_Toc22795)

[第一章 作品概述 5](#_Toc6293)

[第二章 作品设计与实现 6](#_Toc26871)

[2.1 LoongArch反汇编器 6](#_Toc31170)

[2.1.1 划分指令 6](#_Toc29882)

[2.1.2 识别指令 6](#_Toc8765)

[2.1.3 指令结构化 7](#_Toc2848)

[2.2 elf分析器 8](#_Toc22047)

[2.2.1 检测elf文件的合法性 9](#_Toc5727)

[2.2.2 分析elf文件的各个节 9](#_Toc7004)

[2.2.3 分析本地符号 10](#_Toc32477)

[2.2.4 分析动态符号 10](#_Toc29151)

[2.2.5 记录其他重要节的内容 10](#_Toc8567)

[2.2.6 分析函数范围 10](#_Toc5449)

[2.3 反编译器 10](#_Toc7345)

[2.3.1. 数据流分析 11](#_Toc24011)

[2.3.2 控制流分析 17](#_Toc21687)

[2.3.3 抽象语法树生成 29](#_Toc16567)

[2.3.4 代码优化 31](#_Toc19481)

[第三章 作品测试与分析 40](#_Toc14014)

[3.1 测试方案 40](#_Toc8097)

[3.2 测试环境与设备 40](#_Toc15602)

[3.3 视图与基本功能 41](#_Toc9663)

[3.4 与其他反编译器比较 44](#_Toc3847)

[3.4.1 测试数据类型与指针 44](#_Toc30294)

[3.4.2 测试简单循环 46](#_Toc27750)

[3.4.3 测试非结构语句组成的顺序控制流 48](#_Toc1520)

[3.4.3 测试多入口循环控制流 51](#_Toc1807)

[3.5 结果分析 53](#_Toc6059)

[第四章 创新性说明 54](#_Toc29281)

[第五章 总结 55](#_Toc9040)

[参考文献 56](#_Toc22844)

# 摘要

在CPU的设计过程中，为CPU选择一个指令集是CPU设计的基础工作。在当前对国产化CPU的呼声越来越高的当下，国产指令集LoongArch也横空出世。当目前为止使用LoongArch指令集的cpu主要为龙芯3a5000、龙芯3b5000、龙芯3c5000系列的CPU，并且相较于个人计算机，它们更多用于服务器中。

当龙芯CPU使用规模变大后势必会面临各种各样的安全威胁，在这其中计算机病毒便是其中重要的一部分。而当前LoongArch相关的软件生态还没有成熟，安全软件缺失，因此该软件是一个针对LoongArch指令集的反汇编器与反编译器。

在对反编译器的设计中控制流分析是重要的一步。在控制流分析中反编译器将汇编指令的跳转语句翻译成具有一定层次结构的高级控制流语句，如if...else语句、while语句等。对于某些无法翻译成高级控制语句的部分，一般是使用goto语句来表示。因此一个反编译器最终生成的goto语句的数目也在一定程度上反应反编译算法的好坏。

本作品在控制流分析对于这些goto语句将采用重构控制流的方式进行消除，最终在代码生成时不产生goto语句。

# 第一章 作品概述

LoongArch指令集是龙芯中科推出的自研CPU指令集，包括基础架构部分和向量指令、虚拟化、二进制翻译等扩展部分，近2000条指令。龙芯指令系统具有较好的自主性、先进性与兼容性。龙芯指令系统从整个架构的顶层规划，到各部分的功能定义，再到细节上每条指令的编码、名称、含义，在架构上进行自主重新设计，具有充分的自主性。

LoongArch指令集手册共有三部，分别是基础架构、虚拟化指令集、二进制翻译指令集。在2021年4月，龙芯中科发布了基础架构指令集的手册，因此本次作品也是围绕这这一卷手册制作的。

二进制反编译器是逆向工程中的难点，当前比较优秀的反编译器有hex-rays、cutter、ghidra等。在反编译中最重要的两个步骤为数据流分析与控制流分析。这两部分析也直接决定了反编译代码生成的优劣。

在本次作品中我将展示对LoongArch指令集的反编译器。到目前位置以LoongArch指令集为指令集的计算机数量还不是很多，相关的计算机病毒等威胁还没有显现，但是我们未雨绸缪，先尝试设计了一款反编译器。

WebAssembly 是一种新的编码方式，可以在现代的网络浏览器中运行 － 它是一种低级的类汇编语言，具有紧凑的二进制格式，可以接近原生的性能运行，并为诸如 C / C ++等语言提供一个编译目标，以便它们可以在 Web 上运行。它也被设计为可以与 JavaScript 共存，允许两者一起工作。

目前将rust语言编译为wasm的工具链最为完整和成熟。在作品中也将会使用rust作为后端语言并将其编译为wasm，在浏览器中被调用可以将在使用LoongArch指令集的未经任何处理二进制可执行文件反编译，整个过程将在浏览器中完成，用户在浏览器分析结束后可以在结果窗口中得到目标文件的elf信息、反汇编代码、反编译伪c代码以及中间生成的数据流中间代码、控制流图、控制流树、抽象语法树等调试信息。

# 第二章 作品设计与实现

本次反编译器的设计主要分为三大部分，分别是LoongArch反汇编器、elf文件分析器、反编译器。

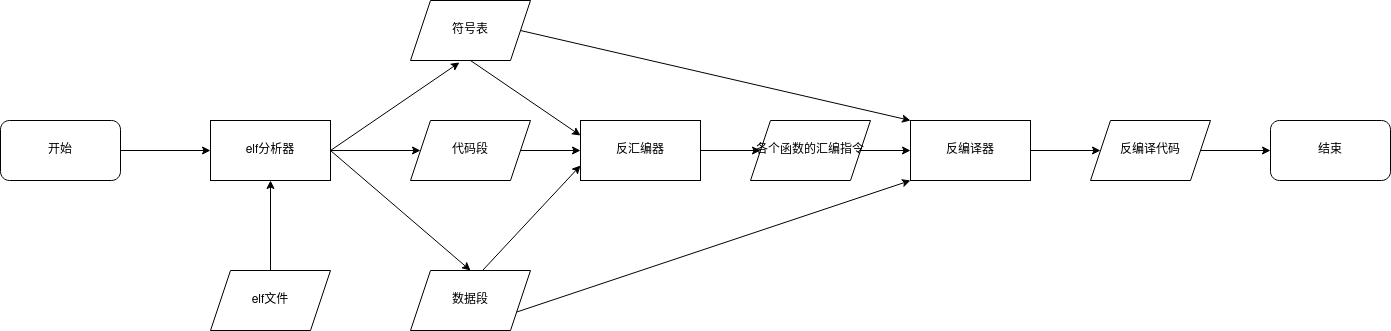


图 1 反编译器流程图

**2.1 LoongArch反汇编器**

截止目前龙芯公司只对外发布了三卷指令集手册中的第一卷，因此反编译器只能反编译属于第一卷中汇编指令。

与当前大多数反汇编引擎一样，这个反编译器接收一串二进制汇编代码并将这串二进制代码输出为汇编指令。

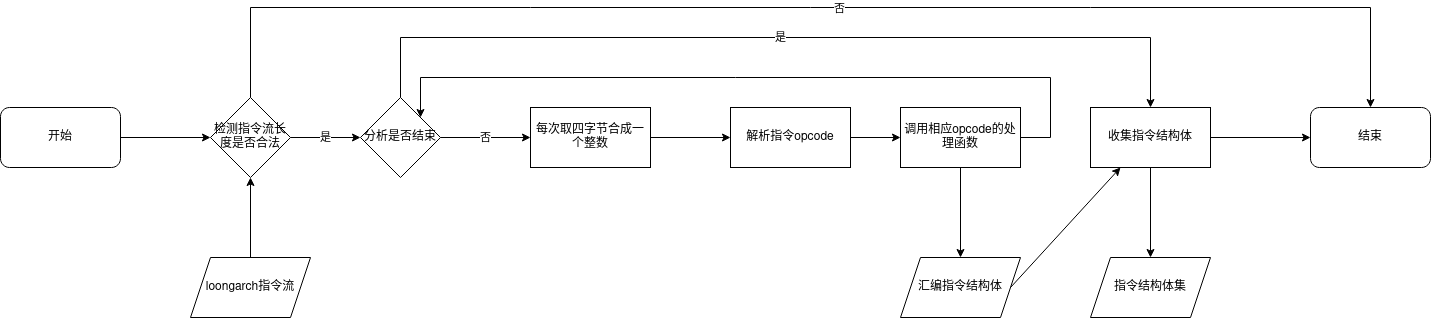


图 2 反汇编器流程图

**2.1.1 划分指令**

LoongArch指令集是精简指令集，因此每条指令占4字节。在反编译指令时需要先判断整个二进制串的长度，若不是4的倍数则拒绝反汇编。

**2.1.2 识别指令**

指令的编码风格是所有寄存器操作数都从0比特开始从低到高一次摆放。操作码都是从第31比特开始从高到低一次摆放。如果指令中包含有立即数操作数，那么立即数域位于寄存器域和操作码域之间，根据指令不同类型有不同长度。具体来说，包含9中典型的指令编码格式，即3中不含立即数的格式2R，3R，4R以及6中含立即数的编码格式2RI8，2RI14，2RI16，1RI21，I26

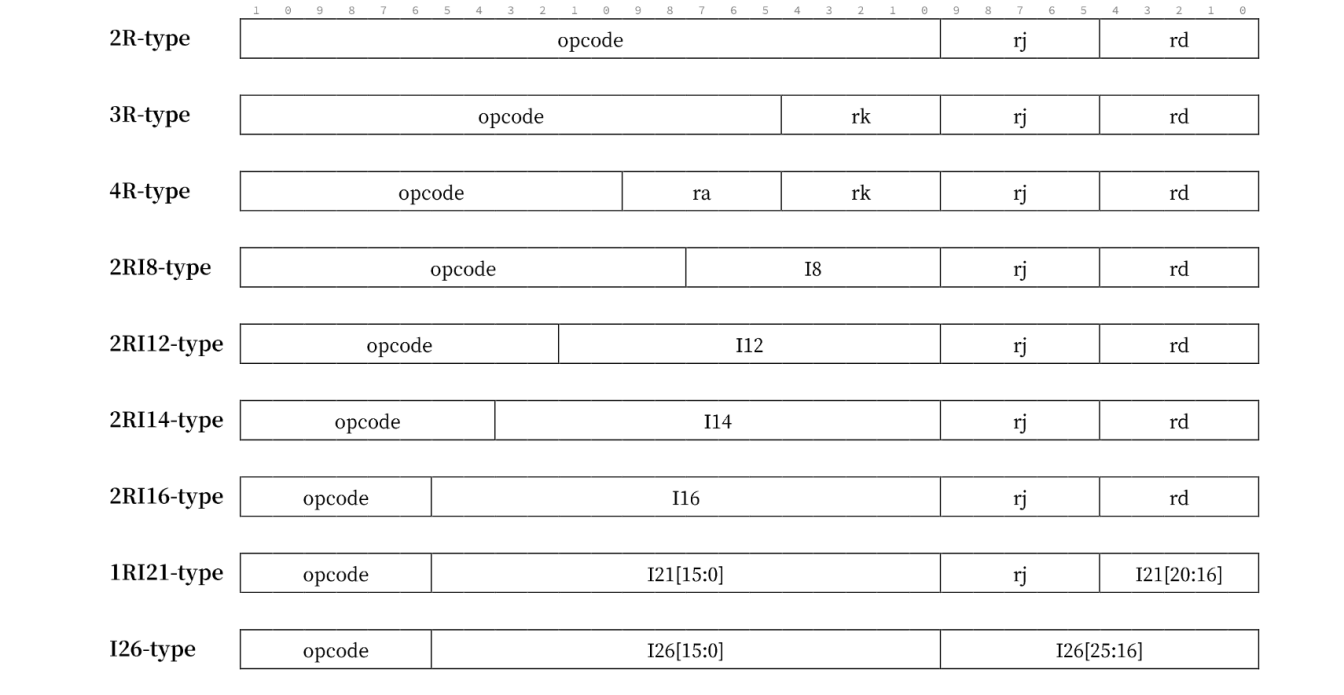


图 3 LoongArch汇编指令分类

由于各条指令的操作码是不会重合的，因此识别指令的主要方法是识别指令的操作码。但是由于指令的操作码的长度会有区别，因此在识别各条指令是我给指令识别设置了一定的顺序，这个顺序是操作码较短的指令先进行匹配，操作码较长的指令后进行匹配。这样设置匹配顺序的原因是若较长的操作码的前段与某个较短操作码重合则会发生匹配错误，因此顺序的由短到长。

识别的流程就是用各条指令的操作码与当前输入指令的相应部分进行匹配，若匹配成功，则本条指令的余下部分也可以从操作码中知晓了。将余下的立即数域、寄存器域进行分别识别即可将当前指令识别。

**2.1.3 指令结构化**

当指令识别完毕后，应当将指令存放起来，而存放的形式应当便于后续的分析，因此我将整条指令存放与一个结构体中。

|  |
| --- |
| struct AssemblyInstruction {  address  label  bytes  opcode  operand1  operand2  operand3  operand4  regs\_write  regs\_read  } |

以上便是存放一条汇编指令所定义的结构体，其中address于表示的是该指令的预期地址，label表示该条指令是否存在一个标签，如果该指令所在地址有一个标签则这个域填上标签的名称，否则这个域填上空。bytes表示该指令的4字节二进制码。opcode表示指令的操作码。operand1到operand4表示指令的操作数，若指令操作数没有4个则多余部分填上空。regs\_write表示在本条指令中被写的寄存器。regs\_read表示在本条指令中被读的寄存器。

将整条指令结构化后则后续操作可以从一个结构体中知道一条指令的所有细节，这样就不需要再去分析字符串形式的指令了。

**2.2 elf分析器**

到目前为止在兼容LoongArch指令集的操作系统主要是linux操作系统的衍生版本，因此操作系统上的可执行文件是elf文件，对elf文件的分析是非常重要的一步。本次实验只对gcc生成的未经任何修饰、优化的elf文件进行分析。

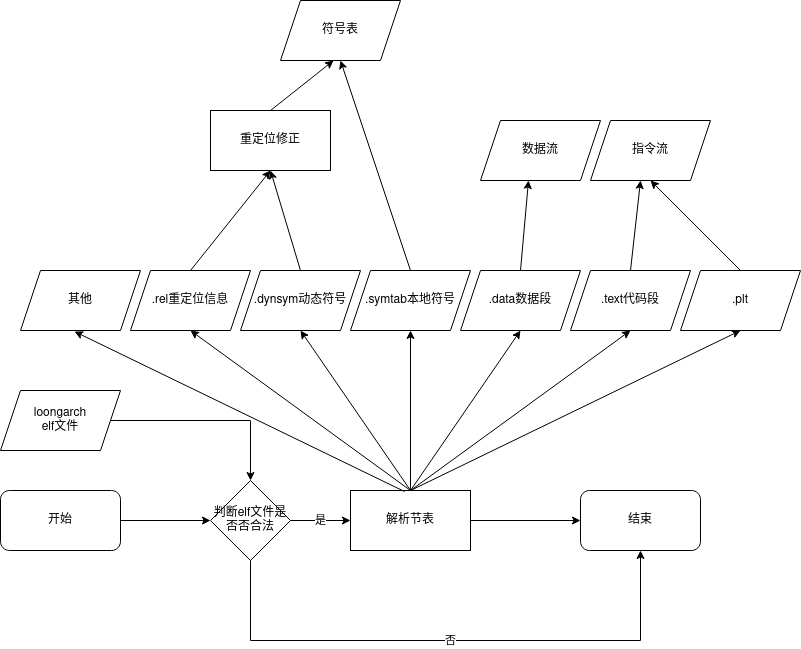


图 4 解析elf文件流程

**2.2.1 检测elf文件的合法性**

读入一个elf文件后，无论文件的位数的多少，elf header部分的大小都是不变的。先读入一个elf header的区域，先检测elf header的域e\_ident是否与标志elf文件的一致。在检测e\_ident中的EI\_CLASS位置用来判断elf文件的位数。其中最重要的检测是检测e\_machine域，在LoongArch指令集下的elf文件这个域的值是258。若检查没有出错则可以判断文件的位数。

**2.2.2 分析elf文件的各个节**

在没有进过任何处理的elf文件中，elf的节表也存放了描述整个elf文件的所有信息，因此可以从节中去分析elf文件。

.dynsym、.dynstr、.rela.dyn、.plt、.text、.data、.rodata、.dynamic、.symtab、.strtab等节是我主要找的节，将这些节的内容分别存放起来以便后续分析

**2.2.3 分析本地符号**

elf文件的本地符号存放在.symtab中，每个符号的名称字符串存放在.strtab中。对于本地符号来说每个存放一个符号记录的结构体中的value字段便是这个符号的预期地址，st\_info字段则可以推断这个符号的属性是变量还是一个函数，之后在strtab中将这个符号的字符串名称，将地址与这个符号的信息绑定起来，这样就可以保存号一个本地符号的信息了。当之后分析时遇到一个地址，可以将遇到的地址与所有符号记录进行比对，若比对成功则表示该地址是一个符号。

**2.2.4 分析动态符号**

elf文件中还会使用各种各样的动态符号。动态符号存放的位置是.dynsym节，对应动态符号的名称的字符串则存放在.dynstr中。去分析这两个位置即可获取到动态符号的信息。具体的分析方式与本地符号相似，但是动态符号存在一个重定位，在记录动态符号时还需要将这个符号的重定位方式记录下来，在后面的分析中需要对重定位进行一些修正才能与代码中的地址相互匹配。

**2.2.5 记录其他重要节的内容**

在elf文件中我们还需要将.text节、.data节（这里泛指各种数据节）、.plt节的内容单独存放出来。

**2.2.6 分析函数范围**

在之前的本地符号分析中，我们可以将本地的所有函数符号的信息获取到，在函数信息中存放有函数地址，即这个函数的起始地址，函数的大小，这样我们就可以将函数的范围获取到。使用一个结构体将函数的名称、起始地址、大小存放起来，并用其起始地址与它对应上，挡在代码中遇见一串函数地址时，可以将地址与各个函数的起始地址相互匹配以判断函数是哪一个函数。

**2.3 反编译器**

本次反编译器的反编译单位是函数，也就是说我们是通过输入一个个函数的信息然后输出对应函数的反编译结果的。

反编译器的设计主要分成部分数据流分析、控制流分析、抽象语法树生成、代码优化四个部分

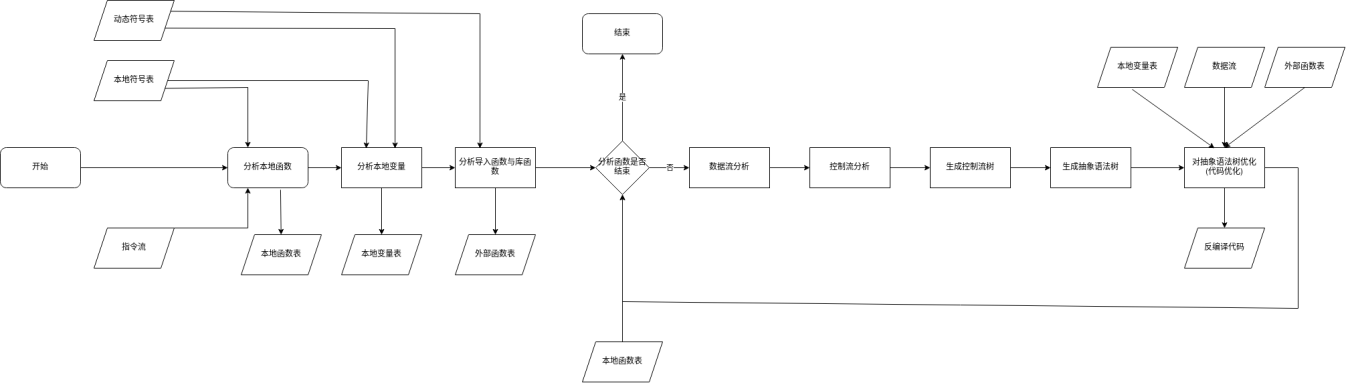


图 5 符号解析流程

**2.3.1. 数据流分析**

数据流分析的主要目的是识别在函数内的变量和变量的类型、函数传入的参数、函数的返回值等。

**2.3.1.1 划分变量类型**

在汇编语言中一个变量可以存放的位置是一个内存地址或者是寄存器内。更加具体地区分可以区分为存放在函数栈中的变量、存放在全局内存中的变量、存放在寄存器中的变量，我将这三类变量划分成三种地址类型的变量，分别是局部变量、全局变量、临时变量。这三类变量还可以划分成参数、返回值、普通变量。区分参数与普通变量的方法是参数不需要声明便可以使用、而普通变量需要声明后才可以使用。

**2.3.1.2 唯一地区分不同变量**

变量的存放的方式是使用一个结构体去存放。

|  |
| --- |
| struct DFISymbolRecord {  address,  sym\_type  id,  size,  value,  } |

每一个这样的结构体是一个变量符号记录，存放了一个变量符号的信息，address是一个枚举，枚举的类型有Stack(stack)、Memory(memory)、GR(gr)，分别表示地址是在栈内、全局内存内、寄存器内，后面括号附加的值则指示了具体位置。在分析属于栈内的变量时，因为无法知道当前栈基址基址寄存器中的值，因此我将其假设为0，又因为栈基址寄存器的值在函数返回前是不会改变的因此可以用指令中的立即树偏移来表示这个变量在栈中的位置。而在表示存放在全局内存中的变量时，因为每个这个类型的变量都会有一个自己的预期存放地址，将这个地址附加在括号中即可唯一识别每个全局变量。对于存放在寄存器中的临时变量来说，寄存器后面附加的值便是寄存器的序号，但是用于同一个寄存器在不同的语境下可以表示不同的变量，因此只是通过寄存器的序号还不足以区分不同的临时变量，因此在结构体中的id值是用于唯一区分临时变量的值，这个值是逐渐增长的，每识别一个临时变量就增长1，这样临时变量也可以被唯一的标识出来了。id的值对于本地变量与全局变量来说是无效的，因此它们都设置为0。

**2.3.1.3 变量的存储**

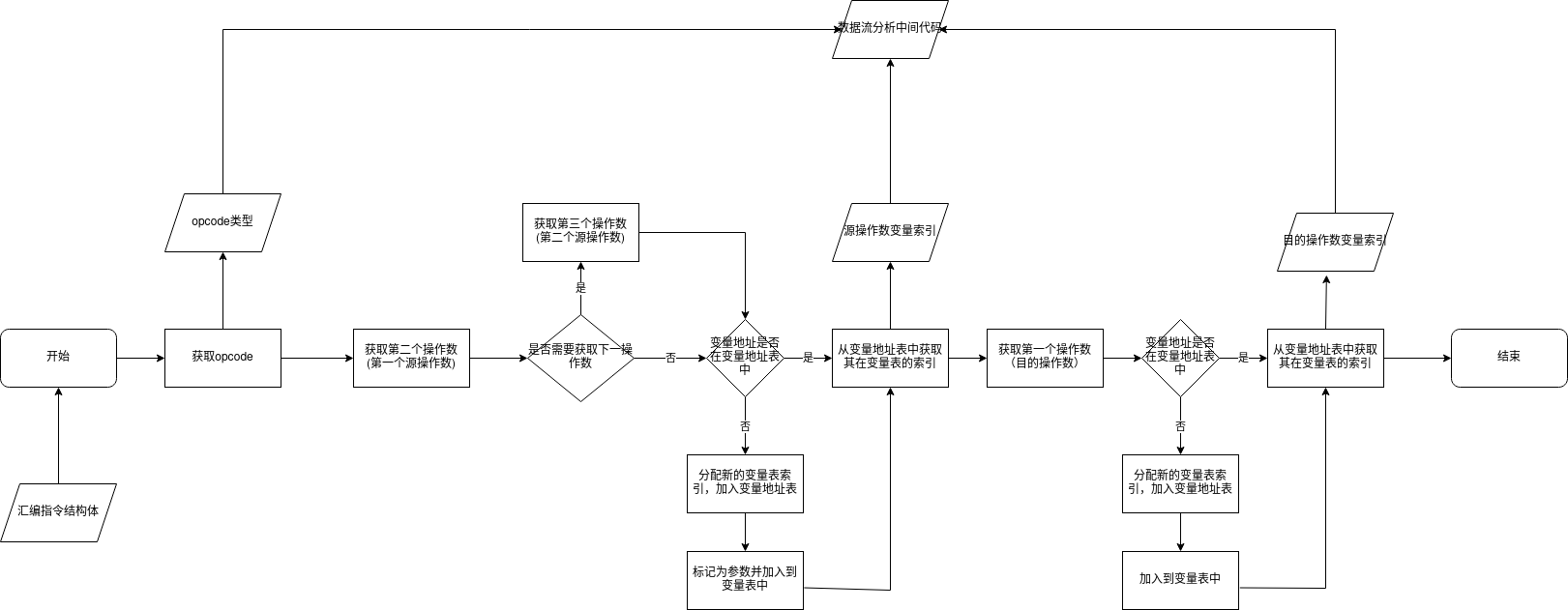


图 6 生成新变量记录流程

在上面的步骤中使用了一个结构体来表示所有的变量，但是在分析函数时需要快速地识别一个地址所代表的变量是否是已经记录下来了的变量，于是我们采用哈希表与存放在分析函数过程中产生的所有变量。这个表也就是函数变量表。函数变量表的索引是变量符号记录。当分析一条指令时，将各个操作数转化成相应的变量符号记录的形式，这样去函数变量表中查找是否存在这个变量即可将判断这个变量是否被记录下来，若没有被记录则将它加入到函数变量表中。

对于在过程中产生的临时变量，即有些即有些指令的目的操作对象是一个寄存器，但是这类指令中的寄存器作用是作为一个中转，这时可以不为它产生一个临时变 量，但是我们又要将它记录下来，于是我再建立了一张哈希表，名字叫做函数寄 存器表，这张表的索引的寄存器的编号，内容是寄存器当前存放的内容，内容如 果是一个变量，则存放的是一个变量符号记录，如果是一个数字，则存放数字常 量记录。当指令需要写入寄存器时将这张表对应的项目的值改变。当指令需要读 取寄存器时，可以从这张表中获取寄存器当前存放的值。

**3.1.4 识别本地变量**

由上述划分规则可以知道，本地变量的存放地址是函数内的栈。LoongArch汇编指令与其他指令类似，需要一个栈基地址寄存器和立即数偏移一起指示一个栈内变量的地址，通过这个特性我们便可以找出栈中的所有本地变量

**2.3.1.5 临时变量的识别**

临时变量存放在寄存器中，因此只需要将寄存器操作数转变为变量符号记录，再在函数变量表中查找就可以判断是否存在这个变量。

**2.3.1.6 全部变量的识别**

当全局变脸被使用时，它一定是存放在一个寄存器中的，寄存器的值便是全局变量的地址，这也意味着全局变量的出现与store、load指令的出现相互关联。当出现store\load指令并且相关寄存器中存放的是一个整数类型的值时便可以判断这个整数的值便是一个去哪聚变量的地址，将它转换成变量记录的形式，就可去表中存放了

**2.3.1.7 在调用函数内识别被调用函数的参数**

在调用者函数内去调用一个函数，可以向这个函数内传递无限制数量个数参数，这些参数有些是存放在数的栈在函数开辟新的占空间时就已经开辟好了，也就是说函寄存器中的，有些存放在栈中的，具体的存放方式需要参照LoongArch指令的调用约定。

LoongArch指令集的调用约定是前8个参数都会一一放入一个对应的寄存器去存放，剩下的参数在放入栈中，存放参数的栈空间可以同时容纳下所有的存放在栈中变量和所有存放在栈中的变量的参数且它们的空间没有重叠。从栈传递的参数从左到右一次从低地址位置开始到高地址放入用于存放参数的栈空间中。

另外在每次根据编译器的行为可以知道在每次开始调用一个函数之前都会对 函数重新设置参数，也就是说相邻的两个函数调用若存在变量完全一致，存 放变量的地址也完全一致的情况，生成的汇编代码依旧会对后一个函数的参 数进行重新填充。

通过上述特性，我设计了如下方式去分析一个被调用函数的参数的方法。首先建立一张与变量符号表结构一致的哈希表和一张与函数寄存器表一致的表，将它们分别称为参数变量符号表和参数寄存器表，在分析这指令时分析到变量时对这张表的操作与函数变量符号表的操作是一致，在分析到寄存

时的操作与函数寄存器表的操作也是一致的。也就是在发现新的变量时将变量加入到这张表中，将发生值改变的寄存器的值更新。但是在发生函数调用的指令时的操作与函数变量表不一致。

当分析函数调用指令结束后，会将这两张表的内容清空中的项目全部清空。这样做的原因是，在两个函数调用之间程序会为下一个被调用的函数重新 填充参数，因为表被清空了，被填充的参数一定不会在这张表内部，这样就 可以将这个疑似参数的变量加入到参数变量表中。最后在分析函数调用指令 时，只需要从这张表中分析参数即可。当然这只通过从调用者的角度去分析 产生的结果只是说明这些值有资格成为参数，不代表它们一定就是参数，最 终确定参数还需要从被调用者的角度去分析。进行这一不的原因是可以缩小 分析范围，简化分析步骤。

具体的分析过程如下。已知前8个参数将会通过寄存器传递，而且这8个寄存器是固定的，因此可以先在参数寄存器表中按照传参的顺序去查找是否存在相关的记录，若在查找到最后一个寄存器之前变存在一个寄存器的项目缺失，假设只找到前n个用于传递参数的寄存器的记录，那么表明本次传参最多只能传递n个参数并且不存在通过栈传递的参数。将这n个寄存器的记录称为疑似参数记录。若这8个项目在表中都被找到了，那么将这8个记录都加入疑似参数记录中。因为此时如果还有参数要传递，则只能通过栈传递了，因此接下来去分析参数变量表的内容。在之前的分析中我们已经知道通过栈传递的参数的顺序是传参从左到右，则在栈中是从低地址到高地址的方式去传递，而且因为这个最低的地址是函数栈空间开辟时就已经生成了的，这样这个最低的栈地址是不会改变的，我们只需要在进入函数时分析函数初始化的操作，就可以知道函数栈的最低地址，而这个地址是第一个通过栈传递的数的地址，并且其中的参数无论位数是多少，它们都将按照8字节的方式对其，这样我们就可以每次移动8字节查找相应的记录，若存在记录则将其加入疑似参数记录中，直到查找到空记录。由于函数栈内的本地变量的地址增长方向是从高地址到低地址的，而参数的地址增长方向是从低地址到高地址的，因此它们中间一般会存在一段空记录，通过这段空记录将它们划分开来。当然也会有刚好无法划分的情况，此时会将整个栈都加入到疑似参数记录中，这时就需要惨被调用这的角度进行区分。

**2.3.1.8 分析当前函数的参数**

在划分变量类型的部分已经讨论过了传入一个函数的参数是未声明的情况下就被使用了，因此要在分析时区分什么情况是变量已经声明了，什么情况是变量没有被声明。在之前的分析中我引入了一张表示寄存器的表即函数寄存器表，在每次分析一个函数的开始时，将寄存器的状态都设置为没有声明的状态。当分析到一条指令时，将指令的目的操作数设置为声明状态。而源操作数是一个没有声明状态的寄存器则可以判断这个寄存器中存放的值便是传入函数的参数。分析程序会收集这类寄存器的记录，将它们加入参数记录中。

以上的分析值涵盖了通过寄存器传递的参数，接下来介绍分析通过栈传递的参数的方法。因为栈的空间是向低地址拓展的，即被调用函数的起始栈地址（最高栈地址）与调用它的函数的最低栈地址（开始存放栈传递参数的起始地址）是相接的，因为栈传递参数的地址在调用这的栈空间中，因此它的地址都会比被调用函数的栈地址要高。因为之前我们已经假定栈的基地址指针寄存器的值是0了，因此函数的变量的偏移是一个负数，函数的参数的偏移就是一个正数，通过这样的方式我们便可以知道那些存放在栈中的变量是函数的参数了。将这些函数的参数也加入到参数记录中。

**2.3.1.9 确定函数的参数**

到目前未知，我们已经从调用者和被调用这的角度分析出来了疑似参数记录和实际使用的参数记录，取两者的交集即可表示实际的参数。

**2.3.1.10 分析函数的返回值**

根据函数的调用约定，函数的返回值将存放在一个固定的寄存器中，只要记录下这个寄存的的值，便可以将这个值作为函数的返回值。但是函数本身没有返回值，在程序中这个寄存器也会被设置，具体分析函数有没有返回值，返回值的类型是什么的方法将会在后面的步骤中介绍。

**2.3.1.11 生成中间代码**

反汇编引擎生成中间代码的作用是将汇编语言变成一种更加高级、更加通用、便于分析的语言。我将用一个结构体去表示中间代码。

|  |
| --- |
| struct DataFlowIr{  address,  opcode,  operand1,  operand2,  operand3,  } |

中间代码的格式类似与三地址码，address表示的这条指令产生的地址，opcode表示的是操作码，operand1表示的是目的操作数，剩下的是表示源操作数。相比于原来的汇编指令结构体它的操作码会更少。操作数可以是一个数字常量记录或者是一个变量记录。

**2.3.1.12 判断一条汇编指令是否需要生成中间代码**

并不是所有的汇编指令都需要生成对应的中间代码。不需要生成中间代码的汇编指令有如下特征

(1) 目的操作数是寄存器，源操作数都是是常量

(2) 赋值语句，将一个变量记录或常量记录赋值到另一个寄存器中

**2.3.1.13 有关指针的操作**

|  |
| --- |
| (1) int a;  int b = a;  (2) int a;  int \*b = &a;  (3) int \*a;  int b = \*a; |

上述三种类型的代码是这步需要进行区分的。

产生上述代码的关键是一个寄存器或者栈中的地址存放的变量可能是一个值也可能是一个指针。因此在变量的记录中需要标注上这个变量是一个值还是一个指针。对于一个指着来说，识别它的关键是找store\load汇编指令，这两条指令的操作数中存在指针，因此在寻找到这两类指令是将变量的类型标注一下，以便识别它们是值还是一个指针。

**2.3.2 控制流分析**

当前主流的反编译的在分析控制流时是使用结构化控制流分析的方法，使用这种方法时是需要按照判断条件分支的控制流图和循环控制流图的形态去对待分析的控制流图进行匹配，可以完全匹配的则可以很好的生成结构化的控制流，遇到无法匹配的分支时则为这个分支生成一条goto语句。

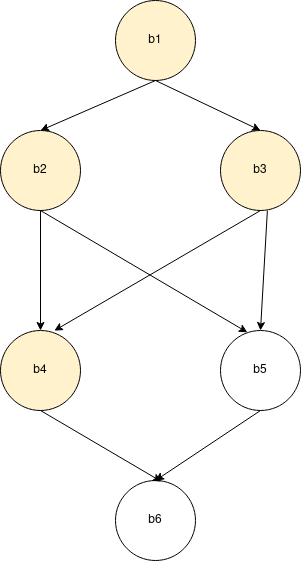


图 7 控制流图

如图7所示在一般的结构化分析中，分析程序会将b1,b2,b3,b4分析为一个if else 结构，而对于b2到b5，b3到b5这两条边无法直接放入这个if else 结构中，因此在b2和b3所对应的基本快中会产生goto语句用于跳转到b5中。

但本次设计的反编译器将不会使用上述算法，而采用通过语义等价的重构控制流的方式进行分析控制流，这样分析的好处是将不会在控制语句中产生goto语句。

在进行分析之前需要明确一些概念

(1) 顺序控制流：指控制流中不存在回边，即没有循环。

(2) 可结构化的控制流区域：指控制流中的一个区域可以在不改变控制流本身的情况下就可以用不含goto语句的控制流语句表示出来。

(3) 单起始点单结束点区域：只从一个起始点出发，无论怎么走最终都会走向同一个结束点的区域。

(4) 单入口单出口区域：指只有一个入口点和一个出口点的循环区域。

(5) 结构化变量：为了进行语义重构控制流而产生的变量，作用是指导控制流的走向。

(6) 结构化的控制流：没有goto语句的控制流

(7) 基本快：由一系列语句形成的代码块，若一个基本快中任意一条语句被执行，则基本快中的其他语句也一定会被执行。

(8) 控制流树：由控制流图生成的树状结构，树中的每一个节点代表一个基本快。

(9) 循环的入口点：循环的入口点是指一个属于循环的节点，它的父节点不全是循环内的节点

(10) 循环的出口点：循环的出口点是指一个不属于循环的节点，它的父节点中存在属于循环的节点。

(11) 回边：一条属于循环内的边，它的子节点是循环的入口点，父节点是循环的一节点。

**2.3.2.1 顺序控制流分析**

在顺序控制流中没有循环，因此我们可以断定，在程序执行到顺序控制

区域时，程序会从当前若干条通向最终节点的路径中选择其中的一条执行。 而我们可以计算出从起始节点到任意节点需要满足的条件

下面介绍顺序控制流的分析方法。

从判断分支中可以看出来一个节点如果它的父节点大与2个，那么这个节点一定无法被直接结构化，必须存在goto语句指向这个节点，若节点的父节点的个数为2个节点，则它也可能无法被结构化。当节点的父节点为1个时，它一定可以被结构化。

首先现将整个控制流划分为若干个单起始点单结束点的区域。这一步的目的是，对于在在同一个单起始点单结束点区域内的节点，它的到达条件不需要考虑到这个区域外面的节点的条件，只需要考虑内部节点的条件即可。而对于结束节点来说，无论它的父节点有多少个，都不需要考虑它的到达条件，只要进入到头结点，那么结束节点也必然经过。

下面介绍如何找出控制流中的所有单起始点单结束点区域。我们发现单起始点单结束点区域有如下特征

(1) 除起始节点外的任意节点的父节点以及父节点的父节点等等最终都会指向起始节点。

(2) 除结束节点外的任意节点的子节点以及子节点的子节点等等最终都和指向结束节点。

由以上特征我总结出了如下算法，在给定起始节点与结束节点的时候判断段这两个节点之间是否能形成一个单起始点单结束点区域。

首先进行初始化操作，初始化一个target数组，这个数组记录了暂时还没有找到但是需要被找到的节点的标号，因此初始化时需要将结束节点加入其中。region数组用于存放已经找到了的节点的标号。还有marked哈希表，用于快速查找一个节点是否已经被查找到。最后是finish标志，用来表示算法是否已经结束。start记录开始节点的标号，end记录结束节点的标号。

算法是一个递归算法，递归结束的标志的target数组为空或者finish的值为真。算法开始前将start的值传递给now。

算法开始先判断now的值是否等于end的值，若不等于且该节点没有被标记则将now的子节点加入target中。若now的值不等于start的值且该节点没有被标记则将now的父节点加入到target数据当中，并标记该节点。在判断now节点是否在target节点中，若在其中，则将它从target中移除并加入region中，并标记该节点。在判断若当前节点不是结束节点则取当前节点的子节点在整个控制流中的拓扑排序的值，若结束节点的拓扑排序在子节点之前，则表明走到了区域外部，并设置finish为真值。这样算法就结束了。若算法没有在上一步结束则将当前节点的子节点一次作为参数now送入函数中递归执行。当算法结束后需要判断target中是否还存在值，若还存在值，则表明不是一个单起始点单结束点区域，否则就找到一个单起始点单结束点区域。

经过上述算法后若产生一个单起始点单结束点区域，则将整个区域用有个节点进行替代，之后在对整个控制流图进行上述操作，知道无法分析出新的单起始点单结束点区域为止。

之后程序将对各个单起始点单结束点进行分析，分析方法如下。由于在这个区域内结束点是一定会到达的，因此我们不需要去分析它的到达条件。由之前所述，一个节点的的父节点若是有2个，则它可能不可以被结构化，但是在一个单起始点单结束点的区域中，除了结束节点外一个节点的父节点若不止1个，则它一定不可以被结构化。这样我们便可以将所有父节点个数不是1的节点的连接这父节点的变全部去除，此时所有节点都有0个或1个父节点，但这时候这个区域可能会被分割为多个区域，由之前的叙述可知，这些每个区域都是一个可结构化的区域。因此我们可以现将这些区域结构化，在考虑将结构化后的区域连接起来。

连接各个结构化区域的方法如下。首先找到各个可以结构化区域的头节点，用这个头结点作为整个区域的代表，各个区域的排列顺序就是区域的头结点在整个控制流图中的拓扑排序的顺序。这样我们就可以将它们按顺序排列出来了。

之后可以计算从头结点到达各个可结构化区域的头结点的条件即可。

这样一个单起始点单结束点区域的分析就结束了，只要从内部向外部分析，就可以将整个顺序控制流图结构化。

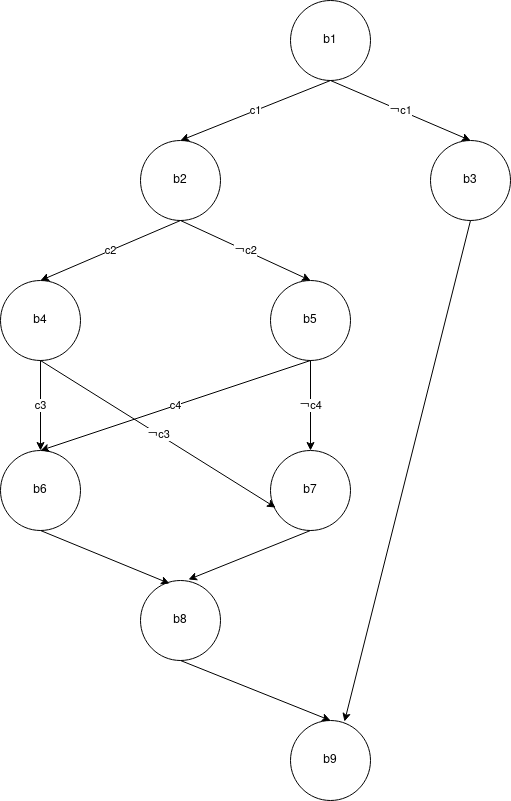


图 8

当需要结构化控制流图8时。

首先将其分割为若干个单起始点单结束点区域。

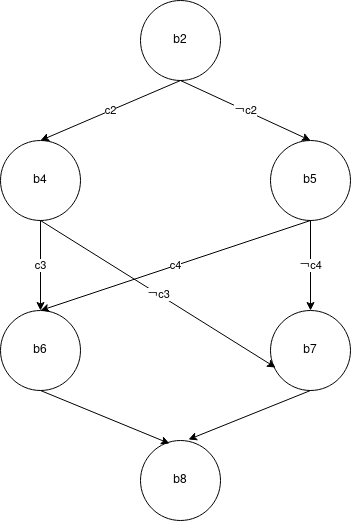
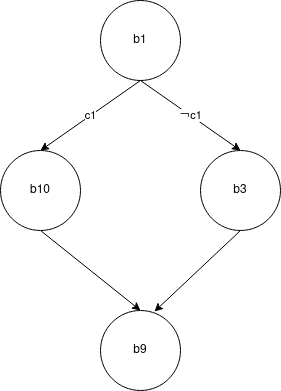
 

图 9

可以将其分为两个区间，并记录下b10对应着头结点为b2的区间。

在将各个区间结构化，首先先去所有拥有一个以上父节点的节点的入边。

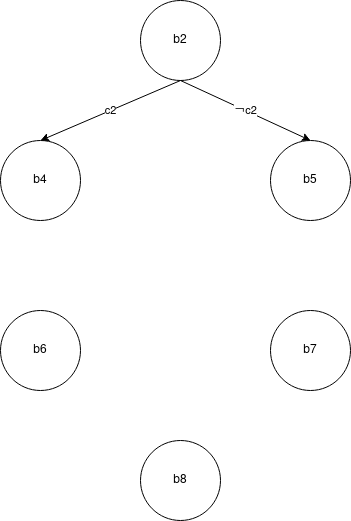
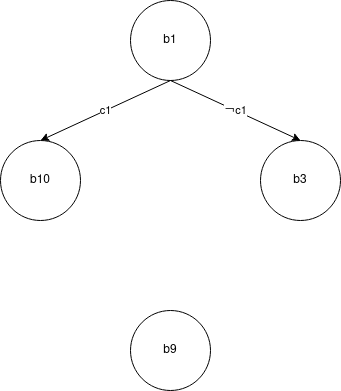
 

图 10

这步后上述所有单独的区域都是可结构化的控制流区域，之后便是将这些单独的区域连接到一起。连接的顺序为各个区域的头结点的拓扑排序。显然拓扑排序的顺序为1、2、3、4、5、6、7、8、9，各个头结点的拓扑排序为1、2、6、7、8、9。

除了每个单起始点单结束点区域的尾节点，所有之前拥有一个以上父节点的节点都需要为其计算到达条件。

|  |
| --- |
| b1  if(c1) {  b2  if(c2) {  b4  } else {  b5  }  if(c3 || c4) {  b6  }  if(!c3 || !c4) {  b7  }  b8  } else {  b3  }  b9 |

**2.3.2.2 单入口单出口循环控制流分析**

在分析一个循环之前需要去识别循环。下面介绍循环识别算法。

循环识别算法是一个递归算法，它输入个控制流图，输出控制流图中所有的环，这些环可以被看作循环。

算法开始前先进行初始化。path数组是一个二维数组，它记录了这个控制流图分析过程中产生的所有循环路径。path\_stack是记录已经走过的节点的标号，但这些节点又无法构成循环并且它是有序的，instack是一个哈希表，内容与path\_stack一致，使用来快速查找一个点是否在path\_stack中的。marked是一个哈希表，用来快速查找一个节点是否已经经过。算法开始时现将头结点作为参数输入到函数中。

假设输入的节点标号为v，则将v加入instack、marked、path\_stack中，之后遍历v的子节点若子节点不在instack中则将子节点作为函数的参数再次进行递归调用。否则否则检测子节点是否已经在instck中，若已经在里面了则表明找到了一个循环，循环的路径是从子节点开始到当前节点，将这条路径从path\_stack中弹出，并将它加入到paths数组中，最后将将v节点从path\_stack中弹出。

等到递归算法结束后我们便可以从paths数组中找到循环。

对于上述循环的处理，首先需要判断循环的入口点与结束点是否唯一。

首先遍历循环节点的父节点与子节点以便求出这个循环的入口点与出口点。若循环的入口点与出口点都是唯一的，这可以开始对这个单入口点单出口点循环的控制流分析。

这一步的主要目的是将循环转换为类似顺序结构的形式。由于我们已经知道了循环的入口点，那么首先现将循环的入口点用一个节点代替，这个节点的类型被标注为循环，为这个节点新增加一条变，这条边是指向原循环开始节点的边。循环之所以是一个环的原因主要是存在一条回边，回边原本是指向循环的开始节点的我们将其重定向到另一个新的节点上去，这个节点被我们标记为back，表示到这里循环便要回到最初的开头。对于出口的处理是，由于只有一个出口，那么就将之前标记为loop的节点的变指向这个出口，原来指向出口节点的变被重定向到另一节点，这个节点被标记为break，代表循环的终止。最后将所有的break、back节点都让它们指向一个sink汇聚点。这样原来的环状结构被重新构造为顺序结构了，并且从loop节点开始到sink节点中的节点正好的一个单起始节点单结束节点的区域，这样表可以对这个区域进行结构化处理了。

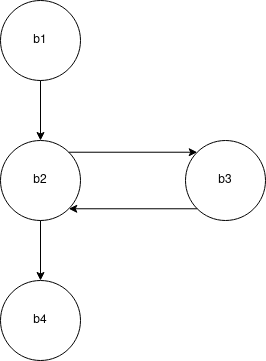


图 11 循环控制流图

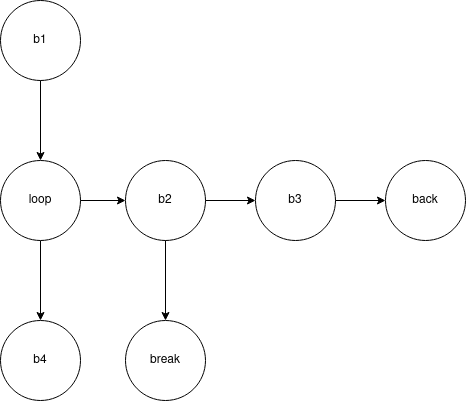


图 12 处理后的循环控制流图

|  |
| --- |
| b1  loop {  b2  if(c1) {  b3  continue  } else {  break  }  }  b4 |

**2.3.2.3 单入口多出口循环的控制流分析**

对于单入口多出口的循环来说，在有多个出口的情况下，我们为每个出口位置的节点添加上一个被赋了不同值的结构化变量，根据这个结构化变量来控制在出口位置到底要进过那些基本快

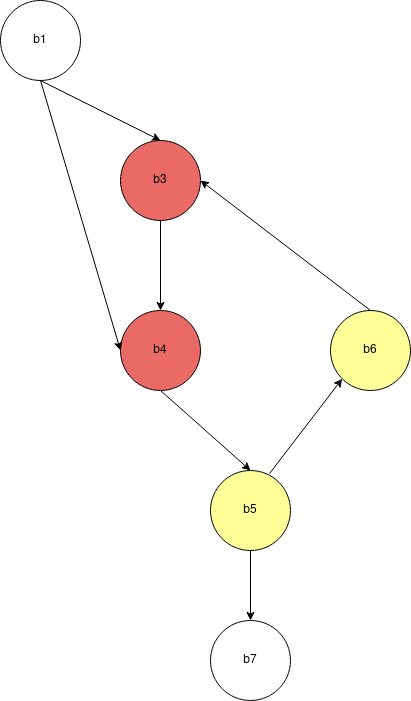


图 13 多入口循环控制流图

如上图所示，b3、b4、b5、b6形成了一个循环，但是程序控制流可以从b1进入循环或者从b2进入循环，这样这个循环变包括了两个入口点。

下图是重构后的循环。

在重构这个控制图的过程中引入了一个结构化变量i，在每个入口点基本快前加上一条控制语句，即 i = x，这样通过i的值可以判断控制流是从哪一个入口点进入的。在回边之前的基本快也添加一条语句，一般是i = 0。其余的重构方式与简单循环一致。

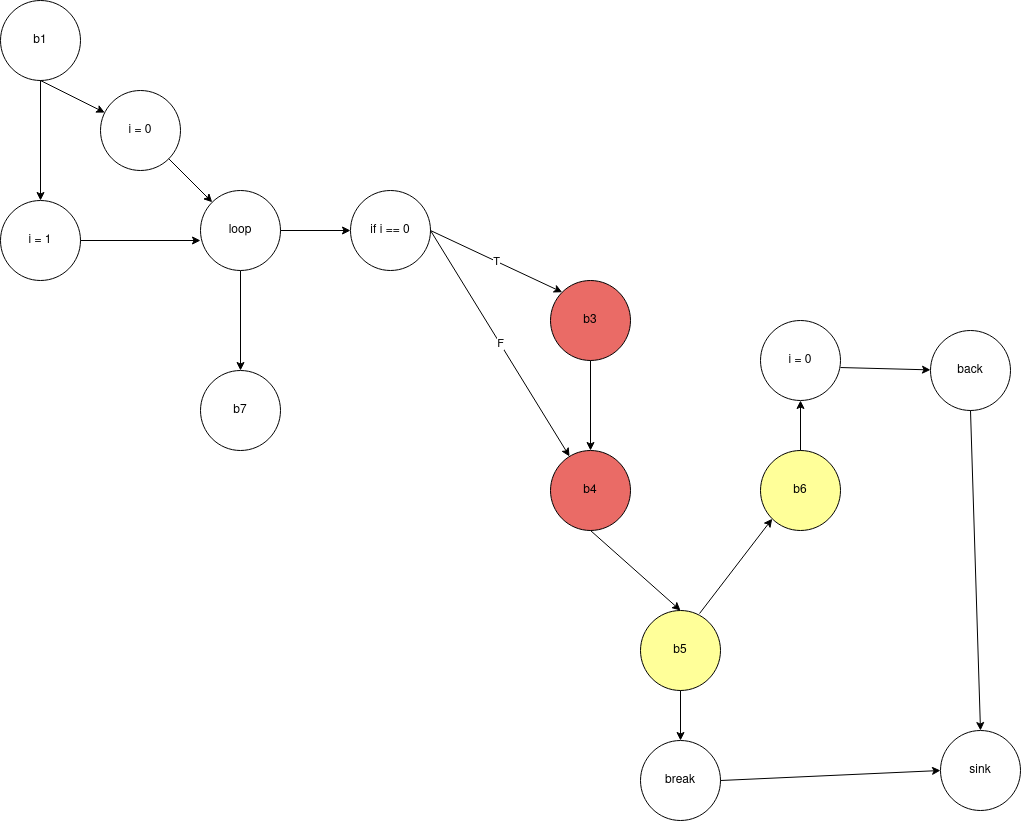


图 14 重构后的多入口循环控制流图

**2.3.2.4 多入口单出口循环的控制流分析**

对与多入口单出口的循环，我们为它在不用入口的末尾附上不同值的结构化变量，在将这些入口全部重定向到同一个节点，所有的控制流都将从这个节点进入，而这个节点也就成为了这个循环的唯一入口点。在到达入口点后需要进入循环的哪一个基本快这是由结构化变量决定的。整个过程类似与在控制流平坦化中的变量在控制控制流的走向。

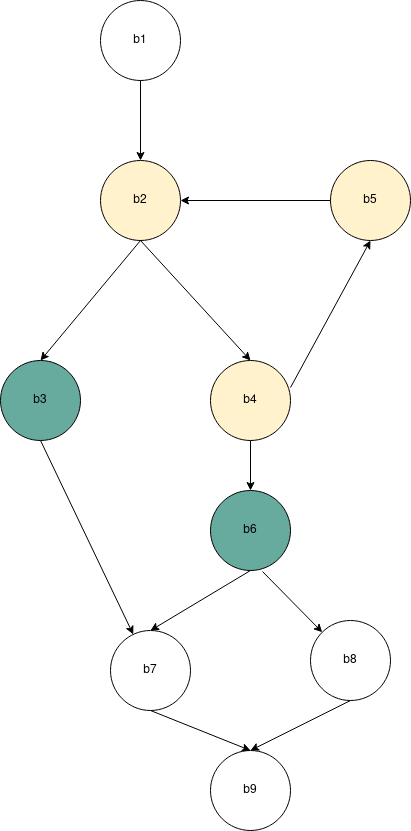


图 15 多出口循环控制流图

在图中循环是b2、b4、b5，出口点有b3、b6。重构后的控制图如下图所示。

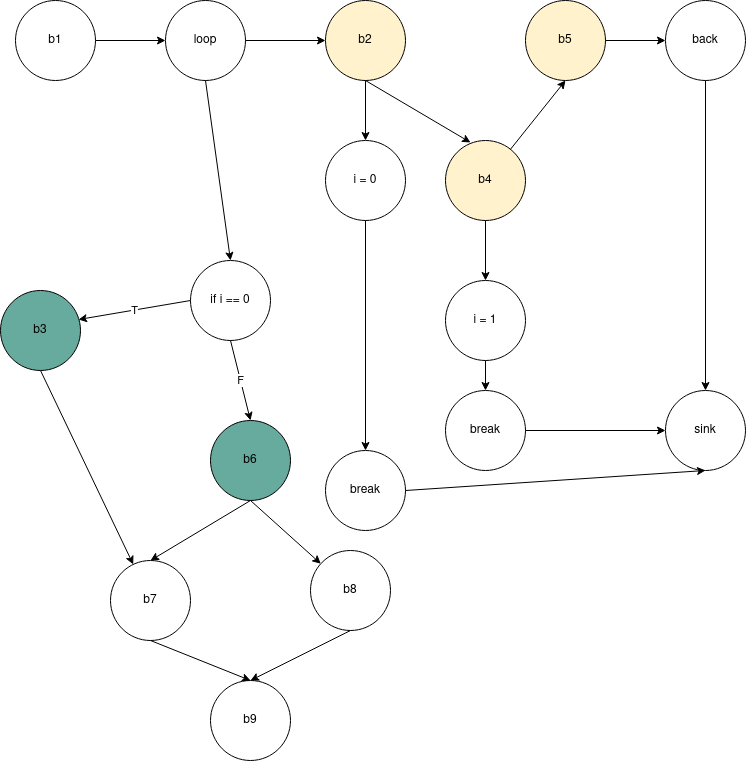


图 16 重构后的多出口循环控制流图

**2.3.2.5 多入口多出口循环的控制流分析**

单入口多出口的控制流分析，之后在进行单出口多入口的控制流分析。

**2.3.2.6 控制流树的生成**

我们可以从重构后的控制流图中生成相应的控制流树。控制流树是具有层次结构。

**2.3.3 抽象语法树生成**

根据上面的叙述，控制流树已经具有一定的层次结构了。抽象语法树则是从控制流树中生成的。其中的主要步骤是将控制流树中的基本块中德玛每一条语句展开。

**2.3.3.1 抽象语法树的构成**

抽象语法树主要由一个个节点连接而成。每个节点是一个结构体。结构体的定义如下。

|  |
| --- |
| struct AbstractSyntaxTree {  ast\_type,  value,  next,  } |

在这个结构体中ast\_type表示的节点的属性。value表示的意义是根据ast\_type决定的，例如若ast\_type表示的是一个变量，则value存放的是这个变量在变量表中的标号；若ast\_type表示的是一个整形数字则它表示的就是数字的值。next是这个树的下一个节点，但这个是一个数组，我们让它的next节点不唯一，next是有先后顺序的，这样做是为了方便编程实现。

**2.3.3.2 语句的抽象语法树表示**

对于常见的赋值语句来说，语句的右侧可能会有若干个操作数和若干个操作符。赋值语句的左侧的的节点的类型用Assign来表示，因为赋值语句的左侧只可能是一个变量，因此节点的value值表示的便是变量标号，在在赋值语句右侧的表示是用一串二叉树来表示，这样可以将同时将运算优先级表示出来。

对于if语句，可以用一个节点类型为If的节点表示，它的子节点有一个类型为Cindictions的节点表示，还有一个类型为True的节点和一个类型为False的节点表示。

对于条件的表示，使用类型为Condictions的节点进行表示。由于一条条件并不一定只有以一个简单条件组成，因此节点下也连接了一串二叉树用来表示多个条件的集合，二叉树的父节点都是操作符节点，操作符节点表示了这些简单条件节点的衔接方式。

简单条件节点的表示，简单条件节点就是只有一个关系运算符和两个操作数的条件。用一个类型为Condiction(relation)的节点表示，节点类型中的relation表示关系运算符是什么。它还有两个子节点，分别是两个变量节点。

变量节点的表示使用一个类型为Variable(is\_ptr)的节点表示，它的value值就是变量在变量表中的标号。is\_ptr表示了这个变量在这条语句中是否作为指针。

常数节点的表示使用一个类型为Integer(is\_signed)的节点表示，它value值是这个节点的真正的值转变为无符号64位值后的数值，is\_signed指示了这个数是否的有符号数。

循环节点的表示是用一个类型为loop的节点表示，节点的next中的节点便是循环内部的语句序列

if为真或假的节点用节点类型为True或者False的节点进行表示，在节点的next中的语句就是其对应的语句序列。

**2.3.3.3 分析基本块中的指令**

在先前生成的中间代码已经比较接近高级语言了，但是其中的变量类型分析还没有做好。本次分析主要是分析各个变量的具体类型。

对于一条中间代码如果它的操作码不是load或者store那么它的类型就已经可以确定下来了。当操作码的类型是load或者store的时候，则会产生一个指针。这个指针指向的数据的类型也可以从load或者store语句中的其他操作树中知晓。

**2.3.4 代码优化**

**2.3.4.1 常量折叠**

在编译器生成汇编代码时在源代码中的常量变被折叠了，在这个分析程序中存在常量折叠的地方是对结构化变量赋值的时候产生的，因为指令集中不存在将一个数直接赋值给一个寄存器的指令，因此在为结构化变量赋值时实际上采用的指令是i = 0 + x。在此产生了可以进行常量折叠优化的情况。

常量折叠优化的过程为输入一条赋值语句，递归地进入类型为Operator的节点，直到没有类型为Operator的子节点，将这个节点与这个节点的子节点形成的抽象语法树送入解析函数，若在解析函数中发现其子节点都是常数，怎返回这两个常数的结果，否则返回空。若返回结果不为空，则用返回的结果节点代替当前节点。

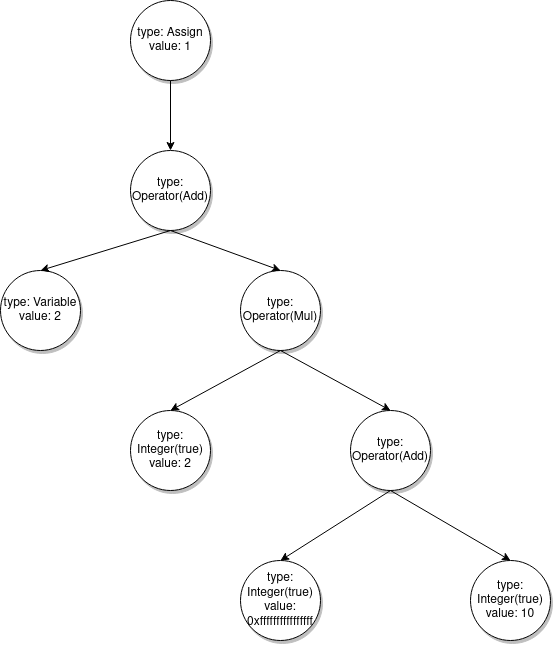


图 17 常量折叠前的抽象语法树

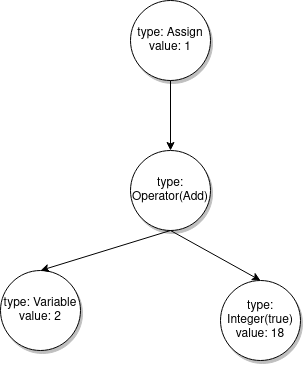


图 18 常量折叠后的抽象语法树

**2.3..4.2 变量传播**

变量传播的过程是先从前到后遍历程序中出现的赋值语句，每找到一条赋值语句时，记录下赋值语句被赋值的变量，在从后向前遍历所有的赋值语句，若赋值语句等号右侧中出现的变量中存在记录下的变量，则可以将这个变量替换为记录下来的赋值语句等号右侧的内容。当所有语句都遍历完成后，在从前到后遍历，若一条赋值语句被赋值的变量在后面的语句中不在出现，则可以将这条语句删除。

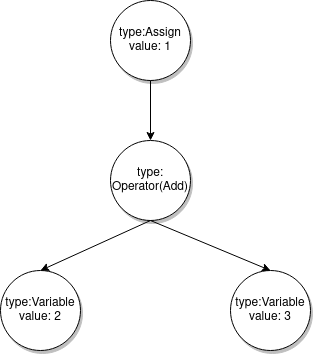


图 19 抽象语法树1

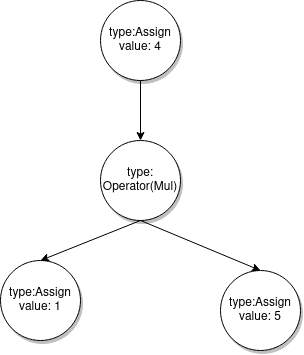


图 20 抽象语法树2

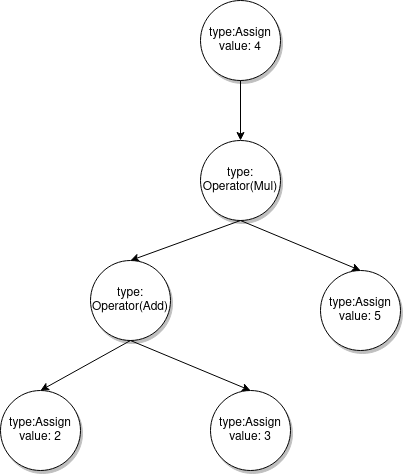


图 21 抽象语法树1、2经过变量传播后的抽象语法树

**2.3.4.3 条件合并**

多个可以合并的条件，可以将这些条件合并到同一个if中

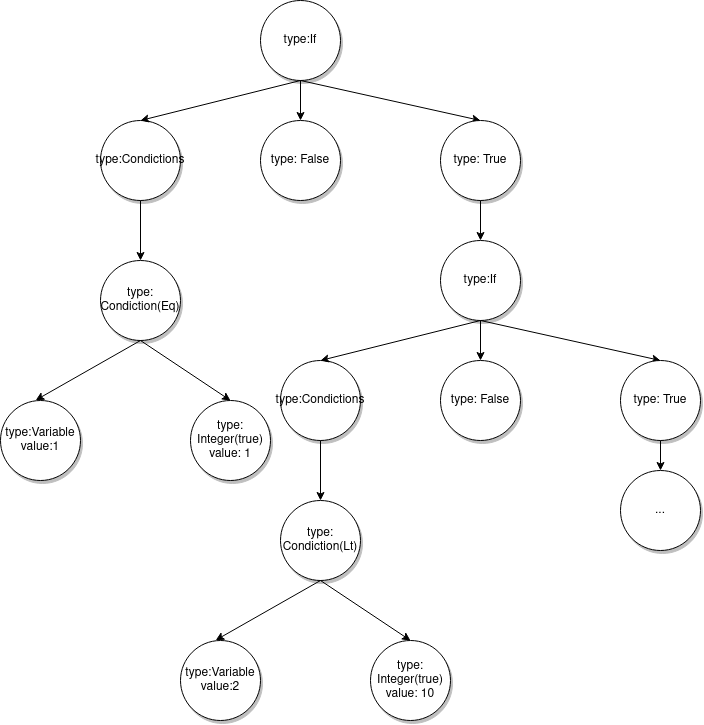


图 22 存在可以合并条件的抽象语法树

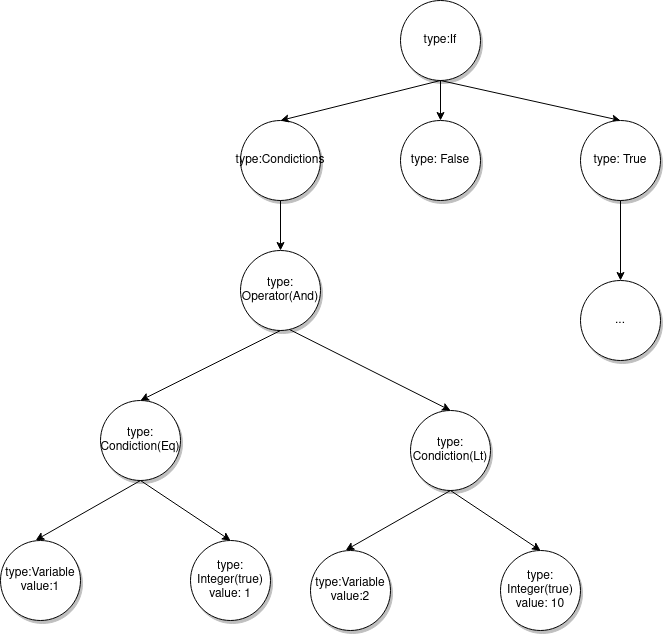


图 23 条件合并后的抽象语法树

**2.3.4.4 重构while循环**

代码生成的原始循环是使用Loop进行表示的，若在Loop节点下存在若干条语句及一个类型为If的节点，并且这个节点里面存在一个类型为Break的节点，则可以将这个Loop节点转为While节点。

首先判断在开头的若干条语句是否为循环的初始化语句。这步只需要判断这若干条语句中赋值语句等号右侧的变量在循环体内部是否被改变。若没有改变，则等号右侧的值在循环体中也没有改变，说明为初始化语句并将这些语句提到循环外，并将If节点去除，将Loop节点改为While节点。若改变了，则将其改变为do...while结构。

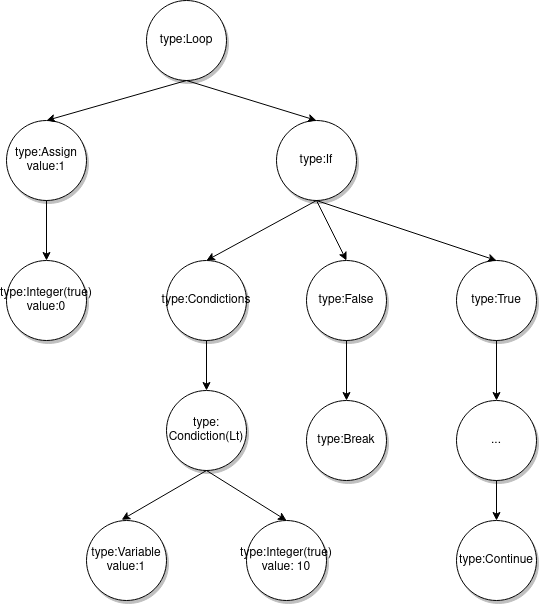


图 24 重构前的循环

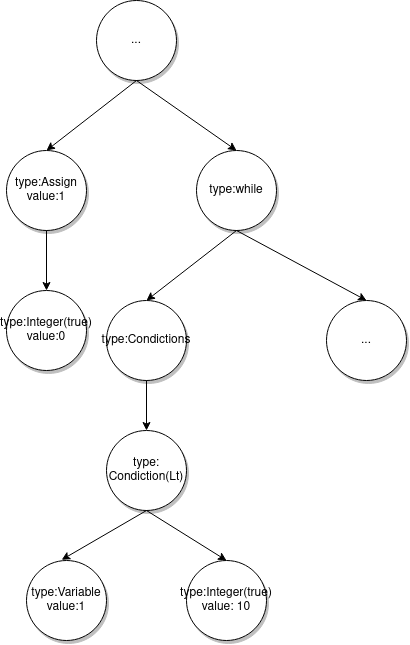


图 25 重构后的while循环

# 第三章 作品测试与分析

（建议包括测试方案、测试环境搭建、测试设备、测试数据、结果分析等）

**3.1 测试方案**

在浏览器中部署反编译器，反编译使用LoongArch指令集的程序。生成反编译代码与主流反编译器生成的反编译代码对比。当前最流行的反编译器为ida的hex-rays，hex-ray将反编译使用intel指令集的测试程序。由于LoongArch是精简指令集，因此还将使用ghidra反编译使用mips指令集的测试程序。

**3.2 测试环境与设备**

(1) CPU：Loongson 3a5000

操作系统：loongnix

反编译目标指令集： 精简指令集LoongArch

反编译器：本程序

编译器：gcc

(2) CPU：Intel i7 9th

操作系统：Arch Linux x86\_64

反编译目标指令集：精简指令集Mips64el

反编译器：Ghidra

编译器：mips64el-buildroot-linux-uclibc-gcc

(3) CPU：Intel i7 9th

操作系统：Windows10

反编译目标指令集： 复杂指令集x86\_64

反编译器：Hex-rays

编译器：gcc

**3.3 视图与基本功能**

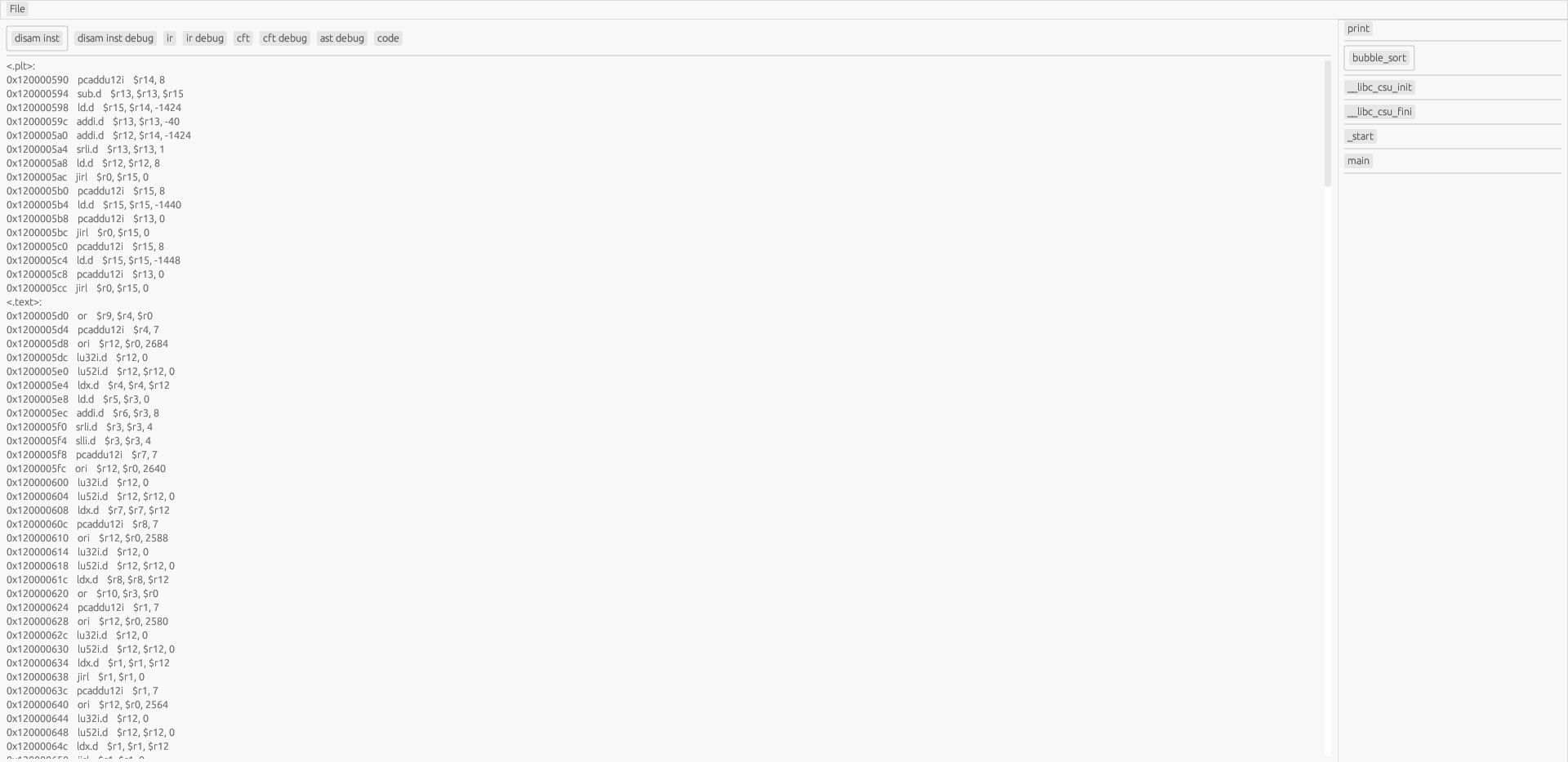


图 26 分析后生成的汇编代码

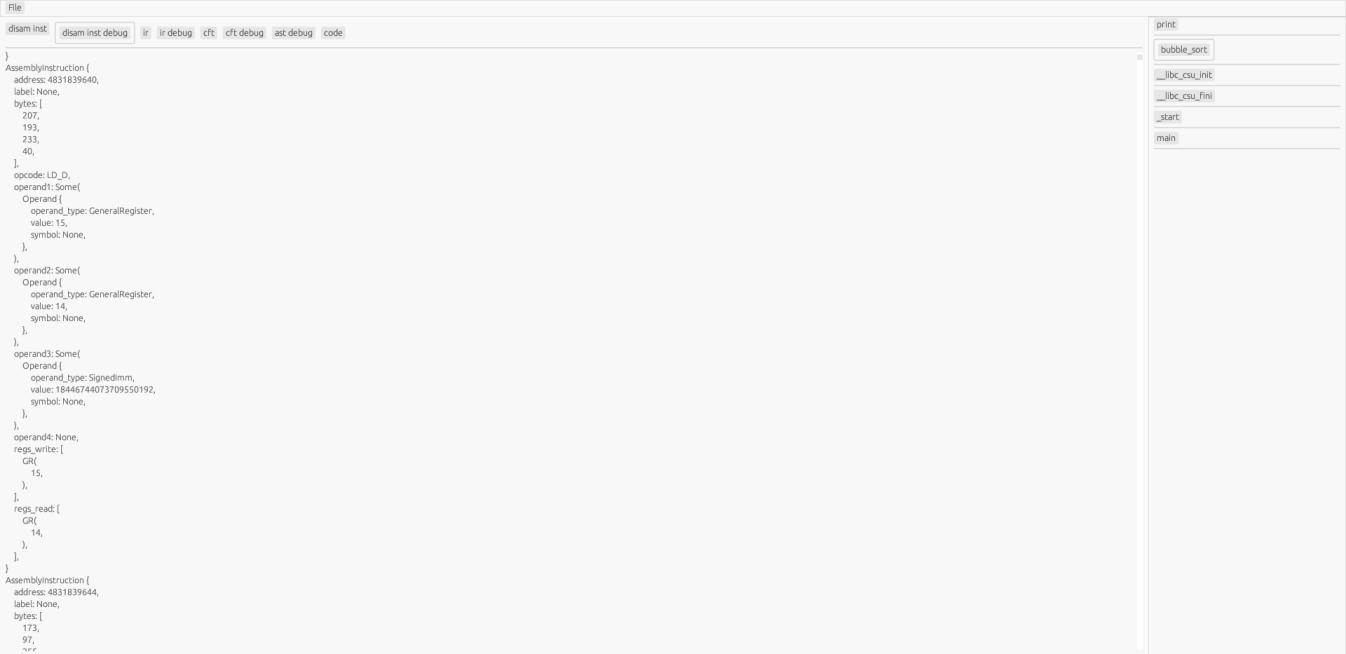


图 27 汇编代码结构体

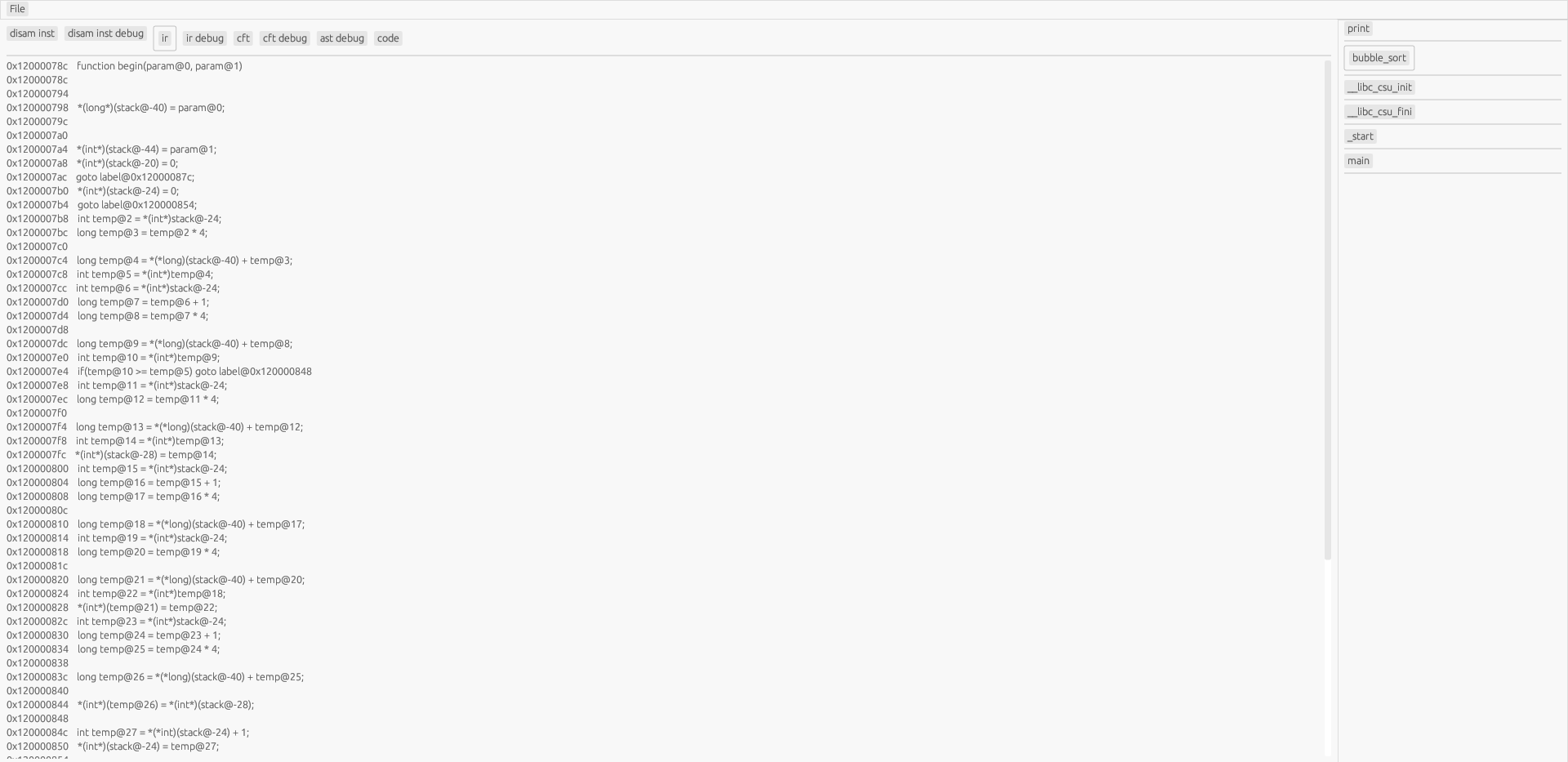


图 28 数据流分析后的中间代码

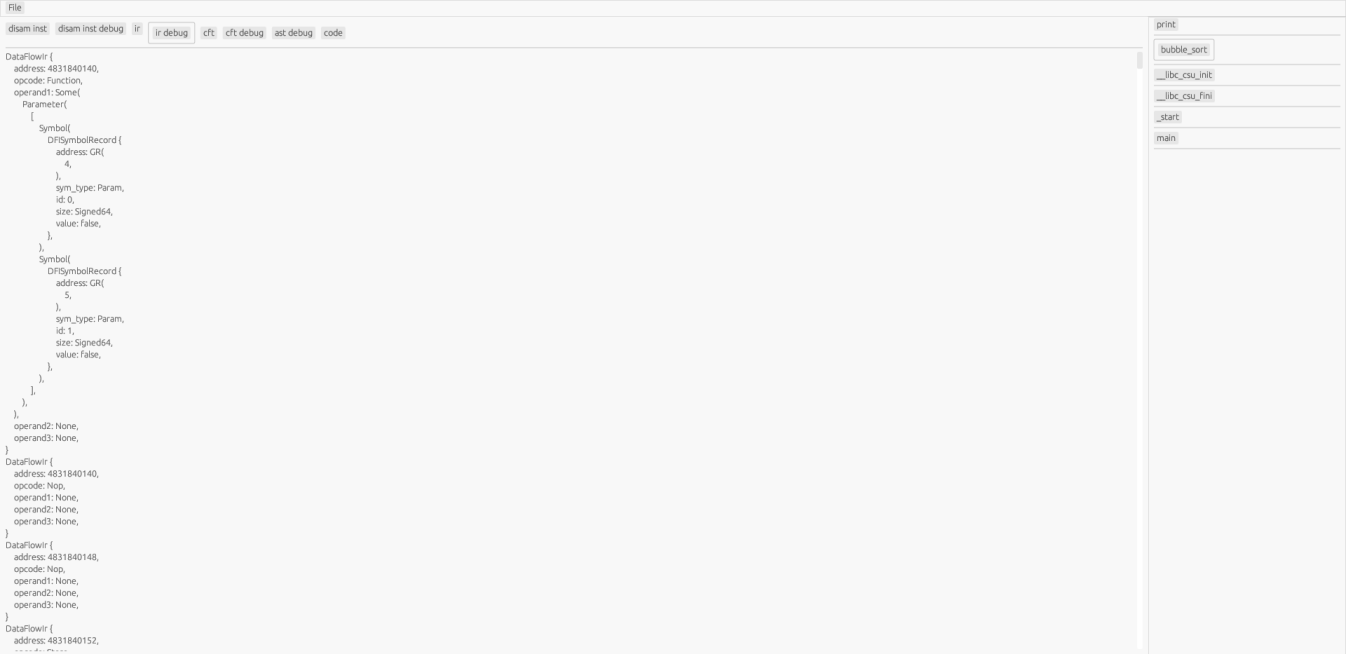


图 29 中间代码结构体

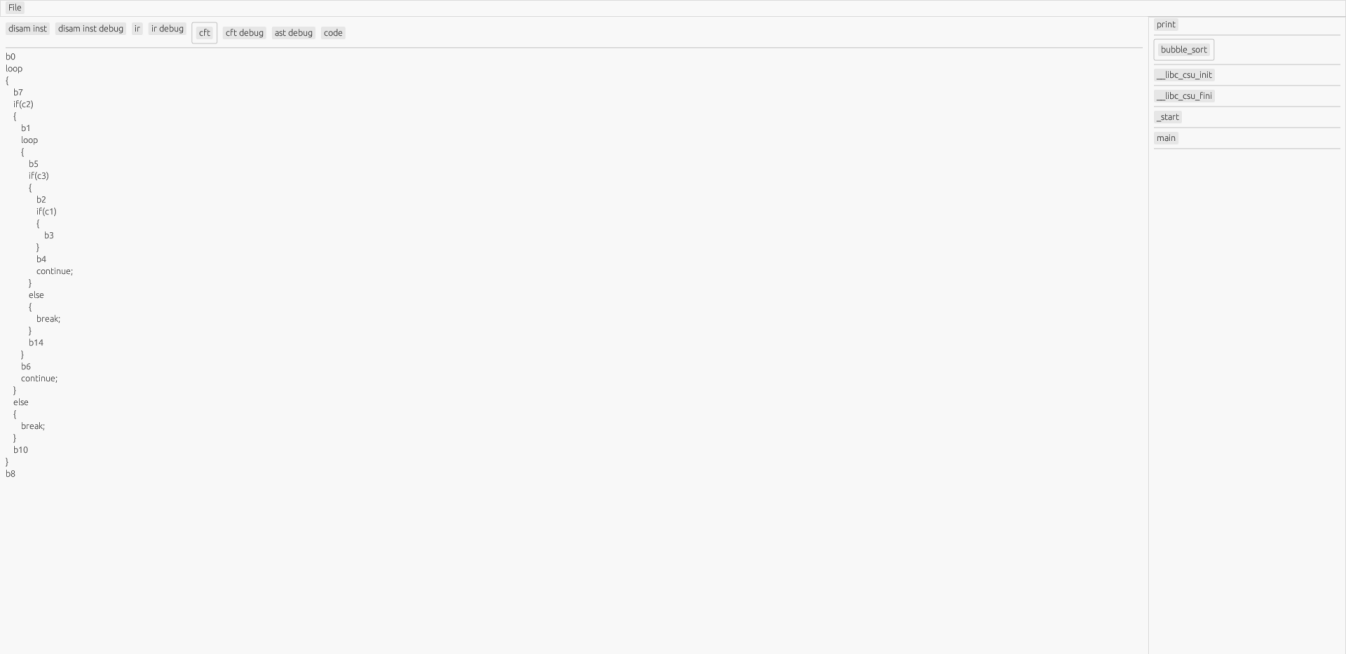


图 30 控制流树结果，即各个基本块的分布

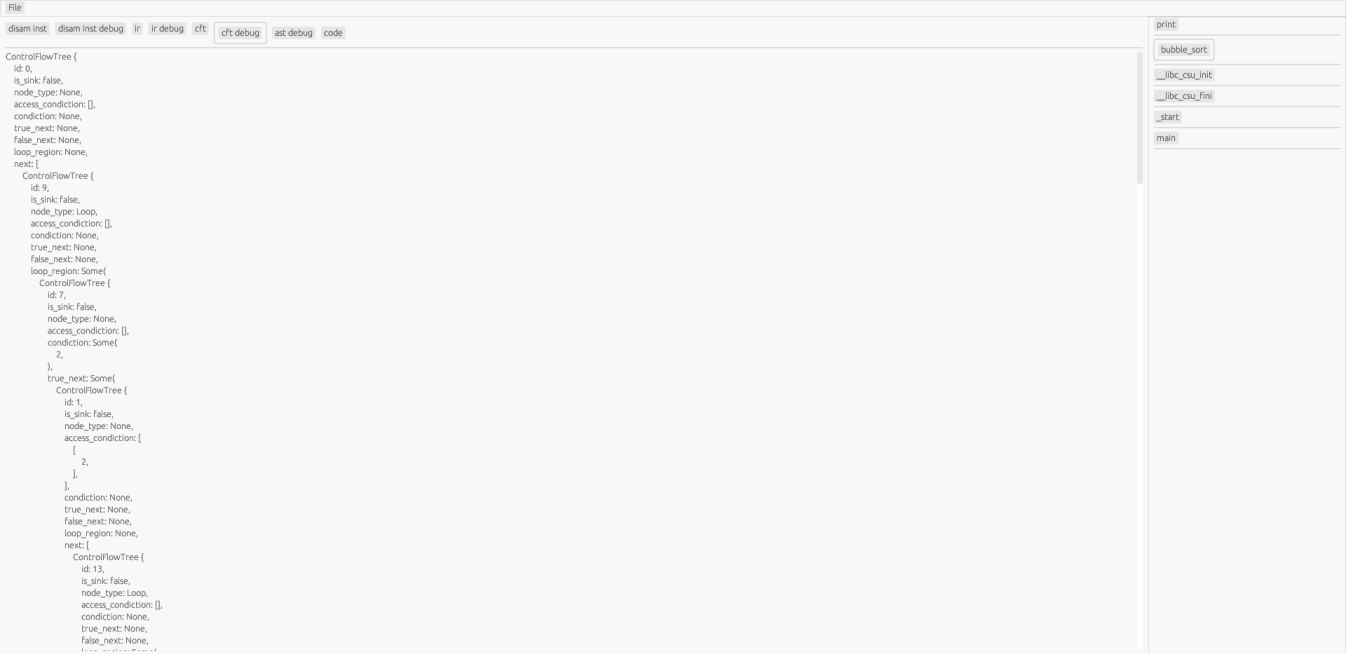


图 31 控制流树结构体

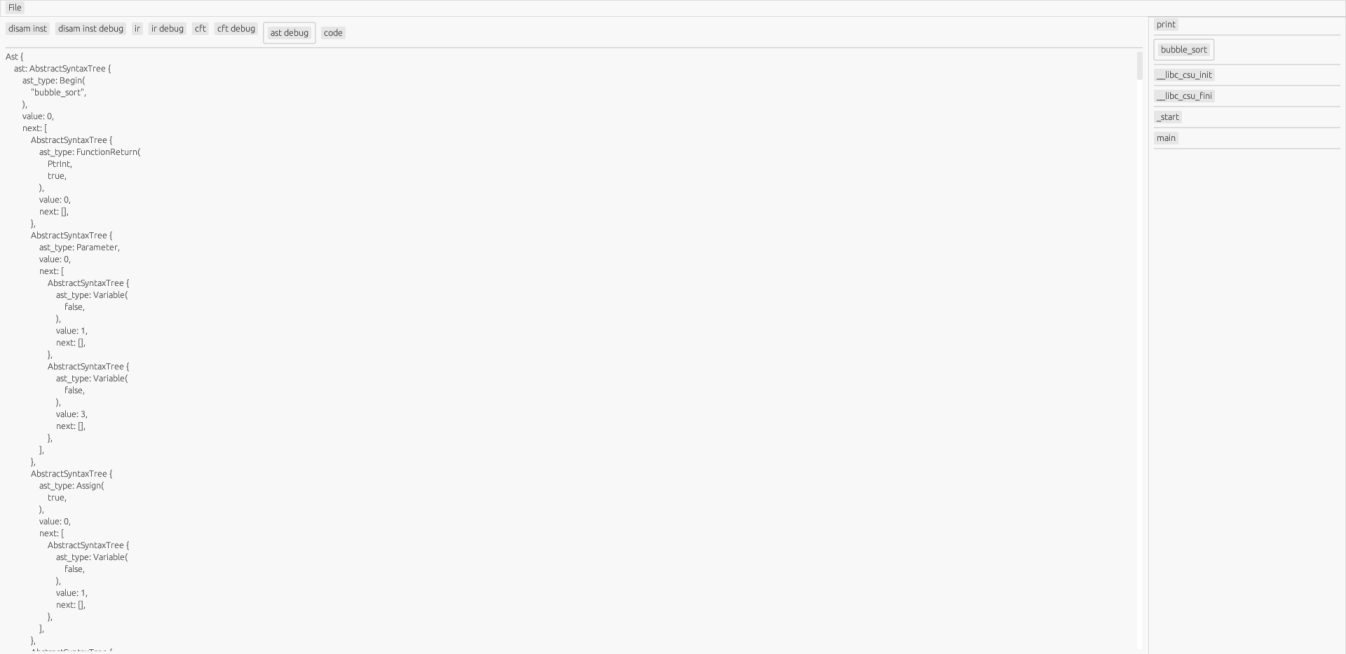


图 32 抽象语法树结构体

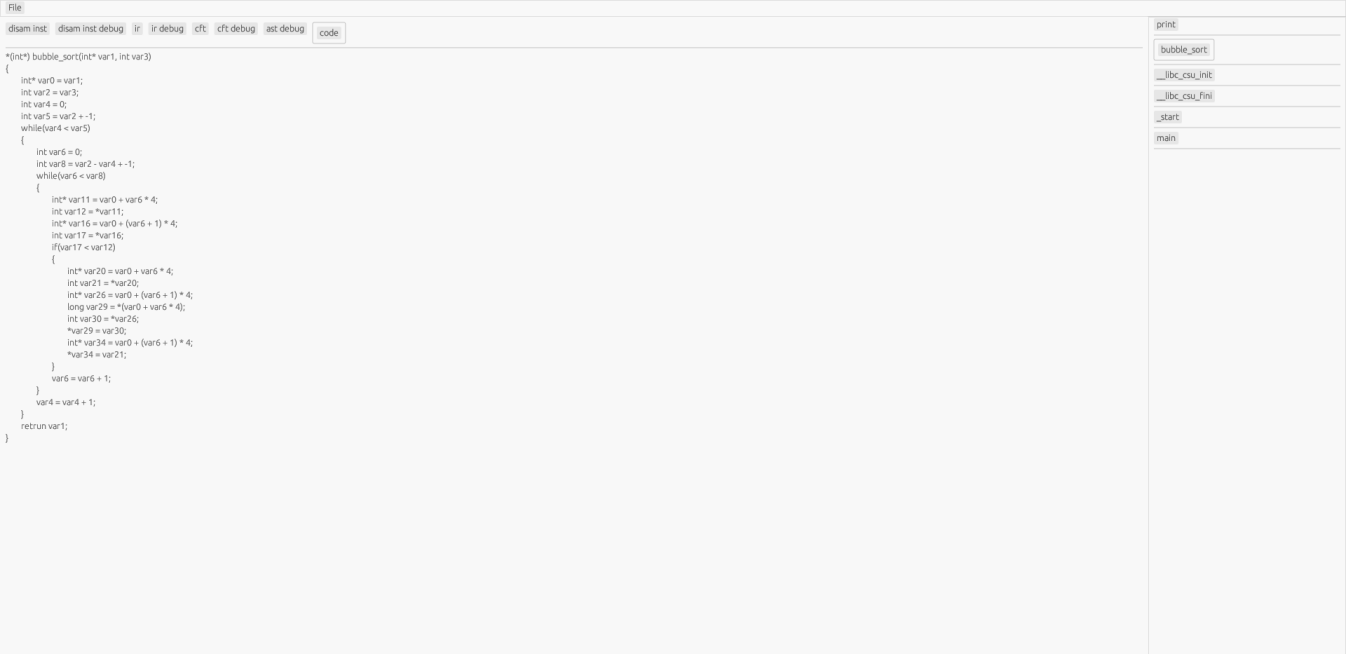
**3.4 与其他反编译器比较**

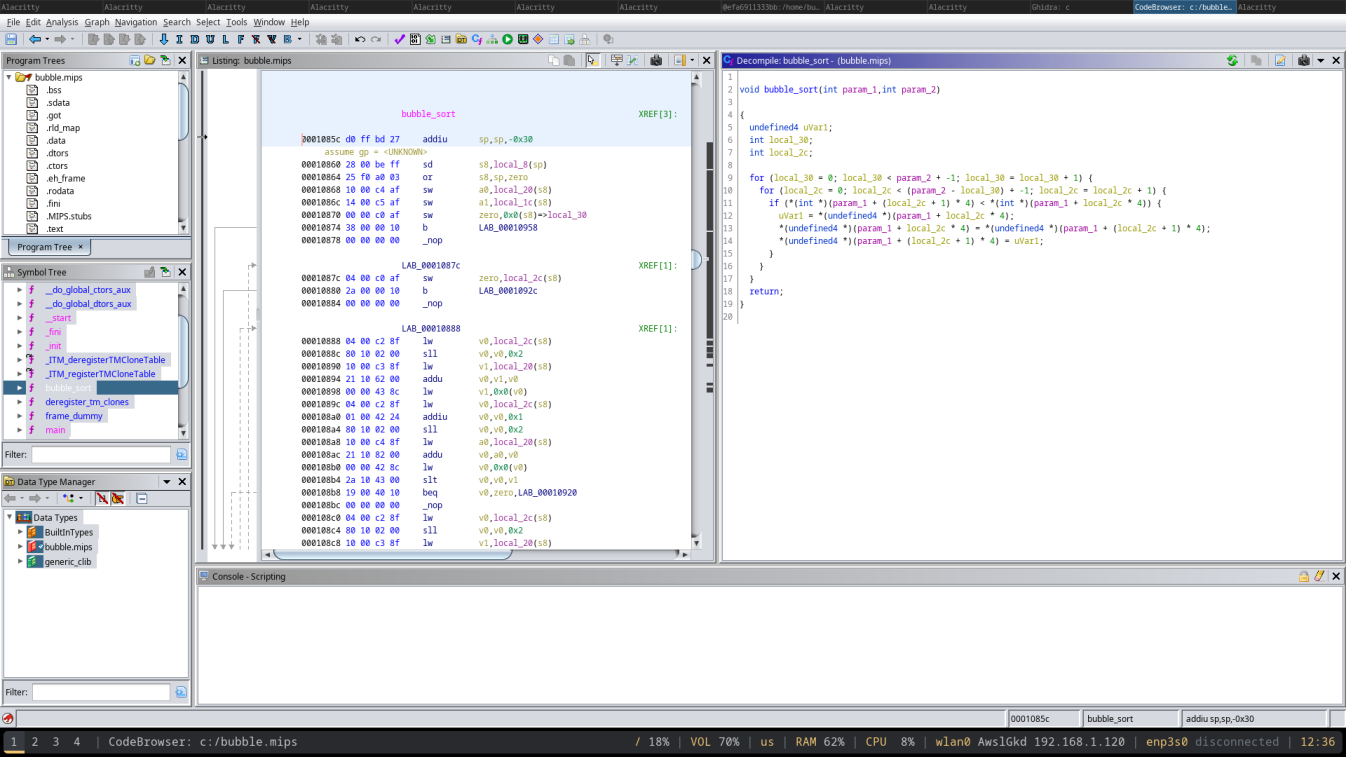
在本次测试中选取了四个源文件进行比较，并将它们分别编译为不同的架构的可执行文件。下面的测试展示顺序分别为测试程序源代码、LoongArch反编译器反编译结果、ghidra反编译结果、hex-rays反编译结果。

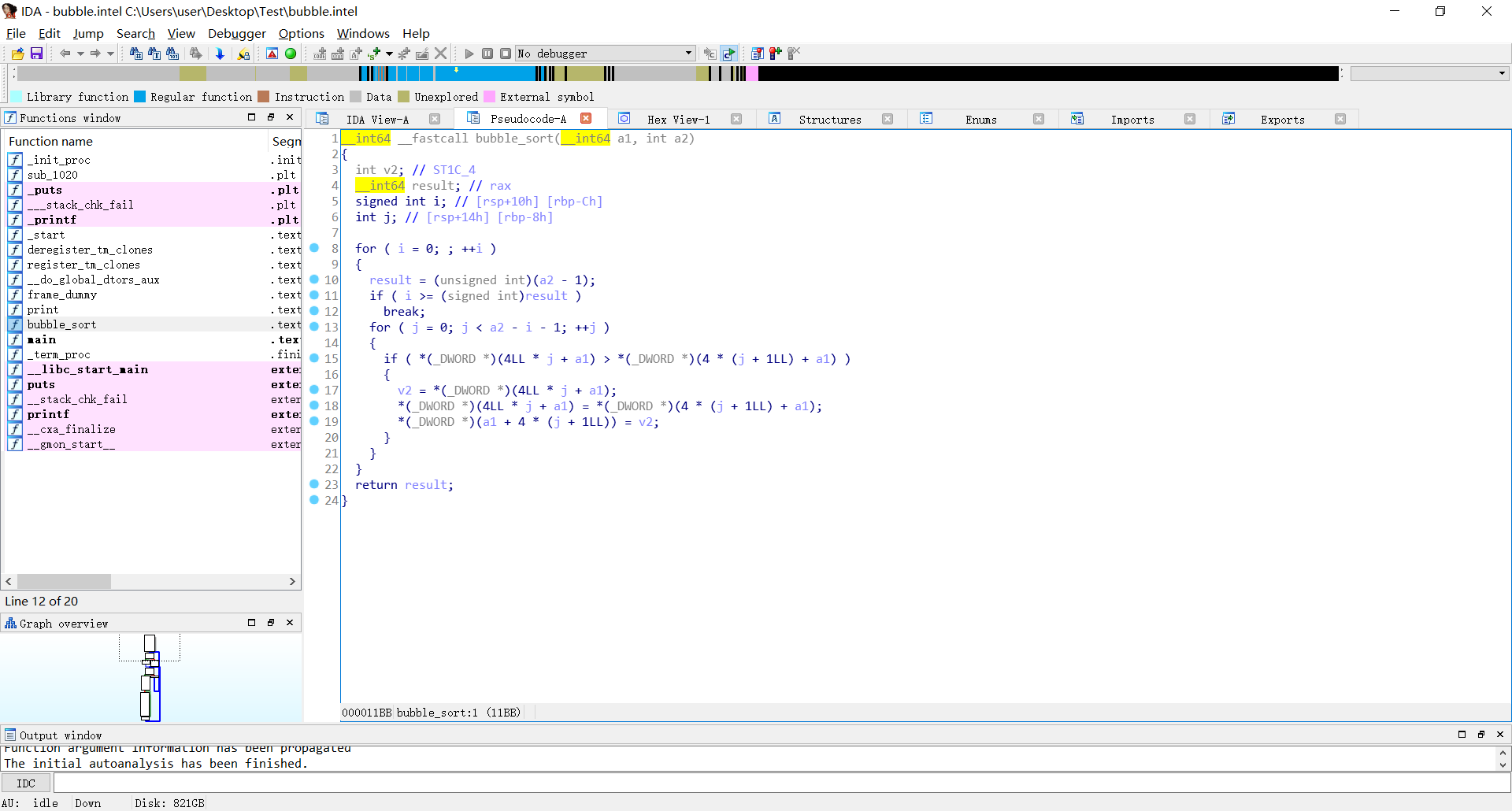
**3.4.1 测试数据类型与指针**

源文件名为bubble.c，在源代码中存在一个void bubble\_sort(int \*array, int n)函数，函数将array中的数进行冒泡排序，n是数组的大小。

|  |
| --- |
| void bubble\_sort(int \*array, int n) {  for(int i = 0; i < n - 1; i++) {  for(int j = 0; j < n - i - 1; j++) {  if(array[j] > array[j + 1]) {  int tmp = array[j];  array[j] = array[j + 1];  array[j + 1] = tmp;  }  }  }  } |



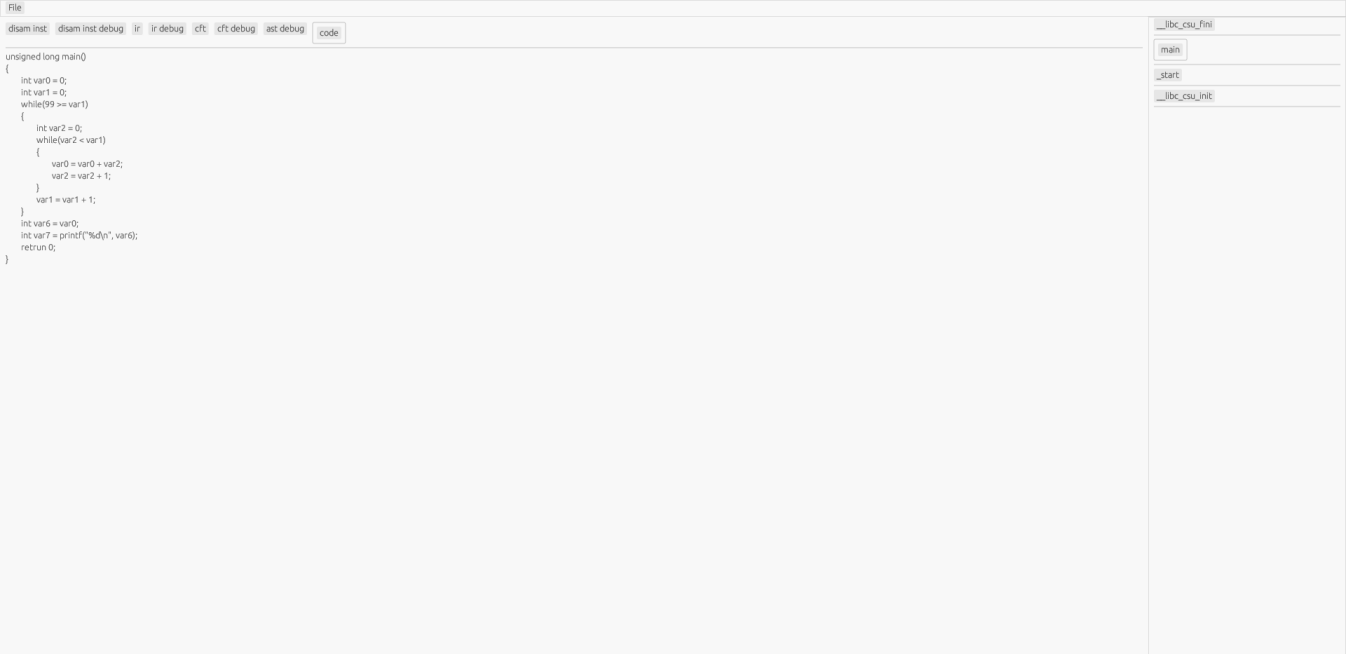


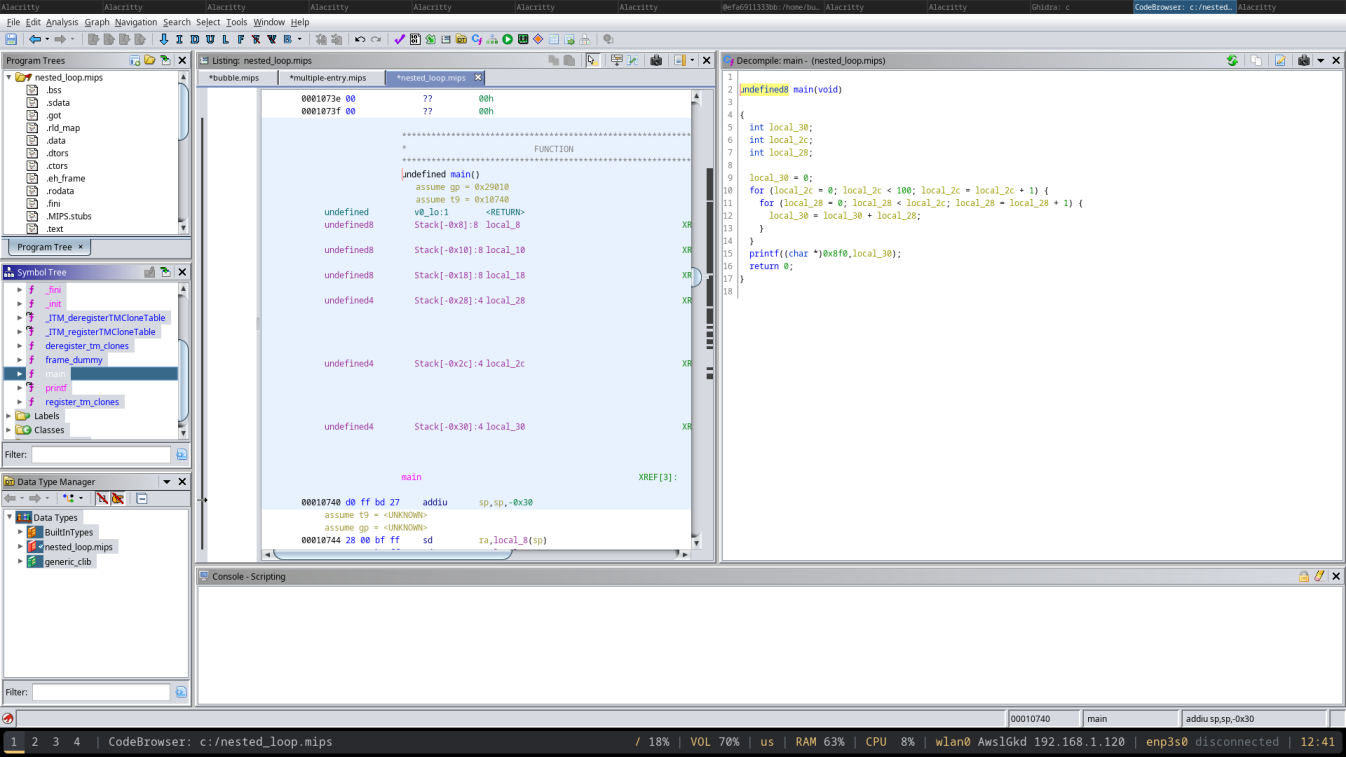


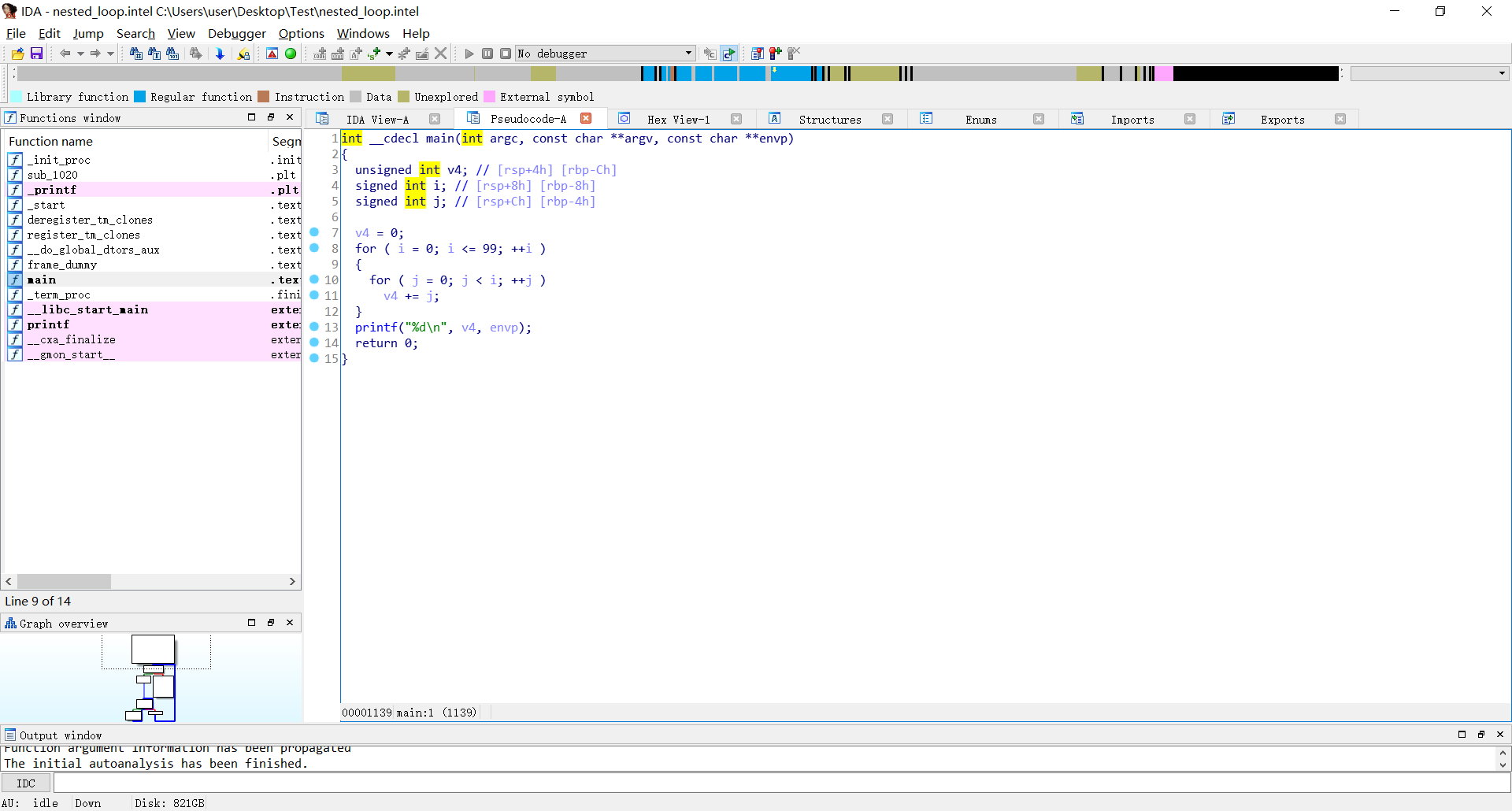
**3.4.2 测试简单循环**

本测试的源文件，名为nested-loop.c。源文件中存在一个嵌套的循环结构并且程序都是由结构化语句组成的。

|  |
| --- |
| int main() {  int sum = 0;  for(int i = 0; i < 100; i++) {  for(int j = 0; j < i; j++) {  sum += j;  }  }  printf("%d**\n**", sum);  } |



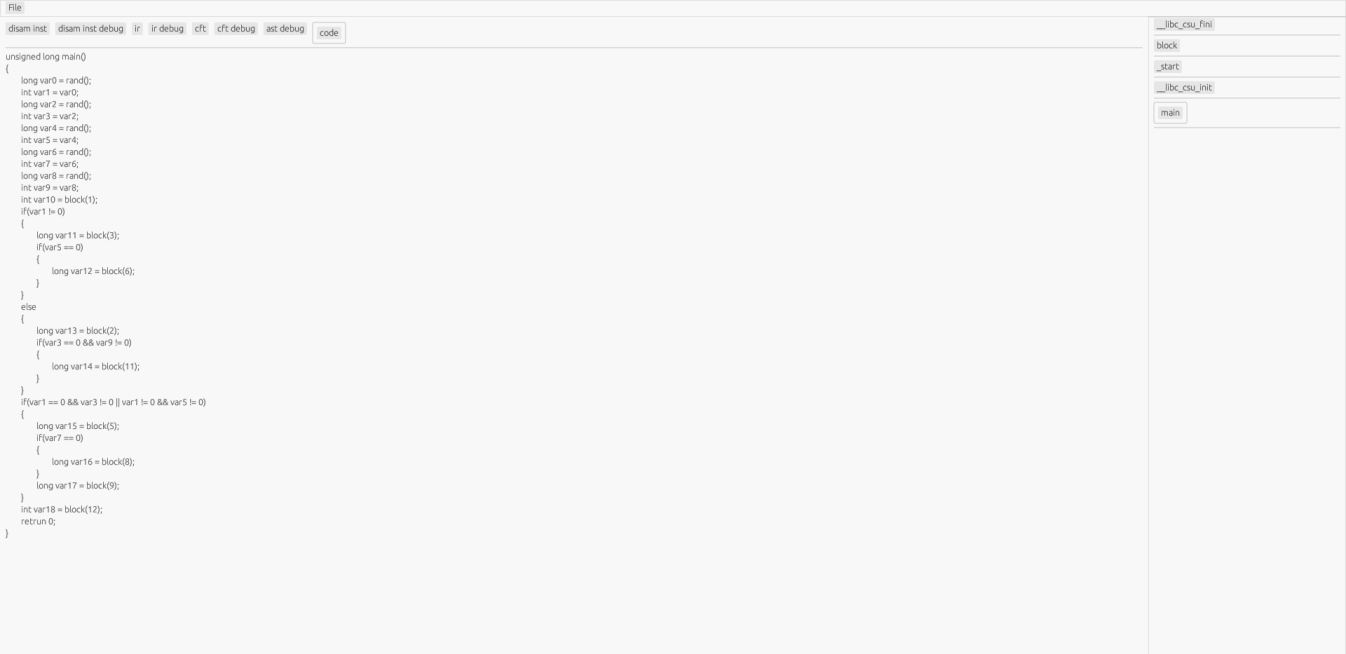


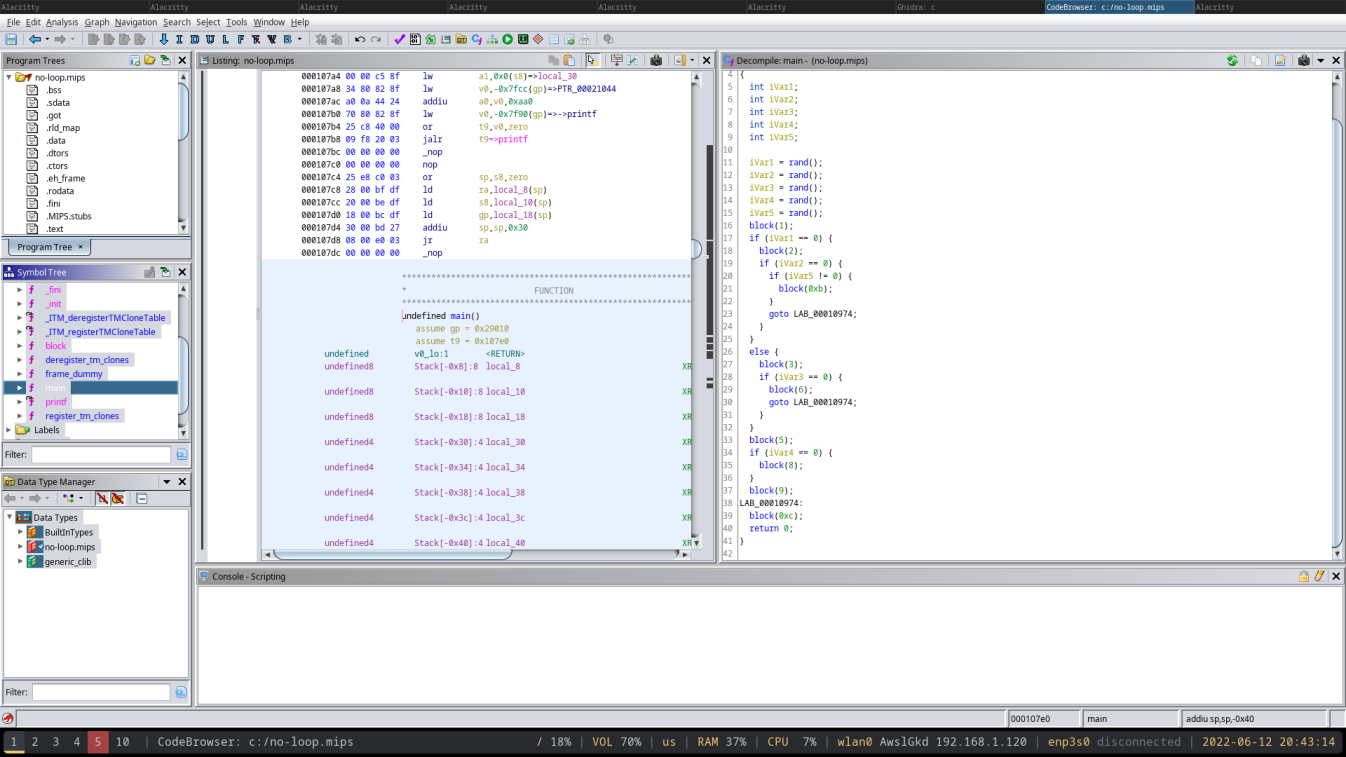


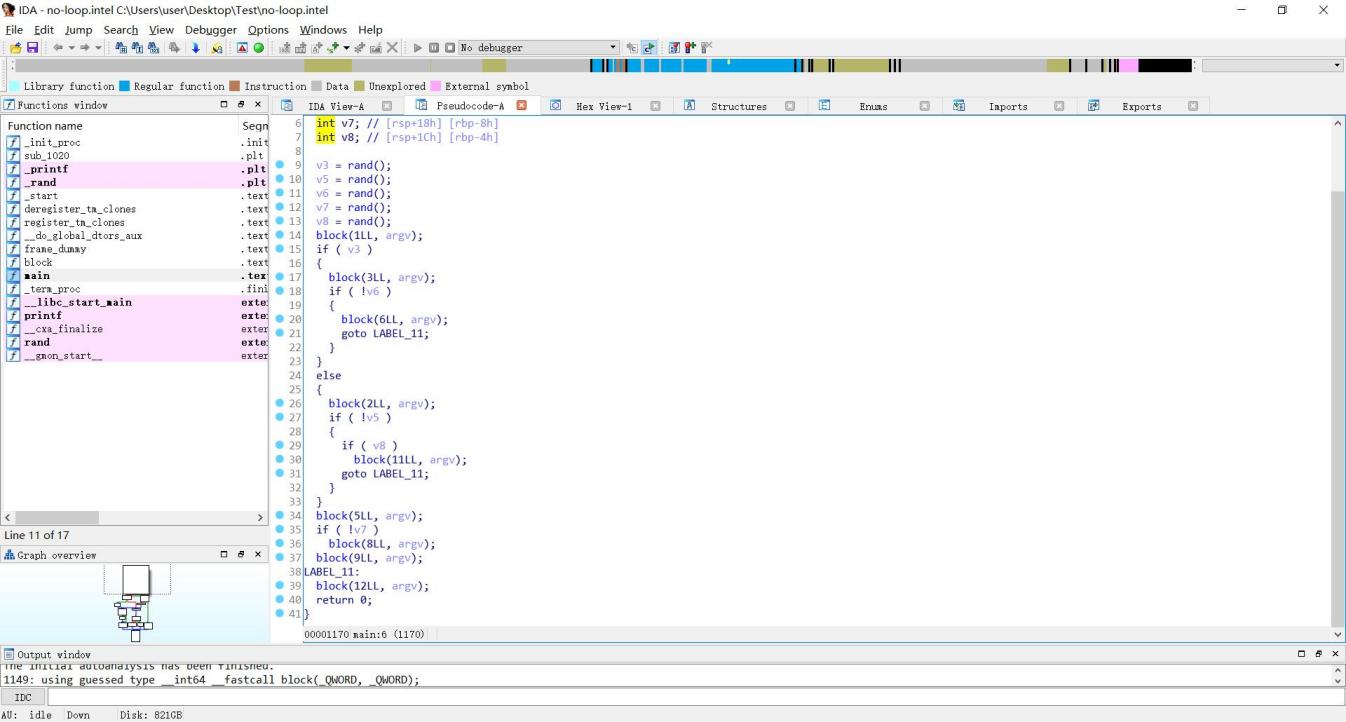
**3.4.3 测试非结构语句组成的顺序控制流**

本测试程序的源程序名为no-loop.c，在源程序中没有使用循环结构，只有分支结构，但是存在goto语句，因此本程序不是全部由结构化语句组成的。

|  |
| --- |
| void block(int i) {  printf("block %d**\n**", i);  }  int main() {  int a = rand(), b = rand(), c = rand(), d = rand(), e = rand();  block(1);  if(a == 0) {  block(2);  if(b == 0) {  goto B10;  } else {  goto B5;  }  } else {  block(3);  if(c == 0) {  goto B6;  } else {  goto B5;  }  }  B5:  block(5);  if(d == 0) {  block(8);  }  block(9);  goto B12;  B10:  if(e == 0) {  goto B12;  } else {  block(11);  }  goto B12;  B6:  block(6);  B12:  block(12);  } |



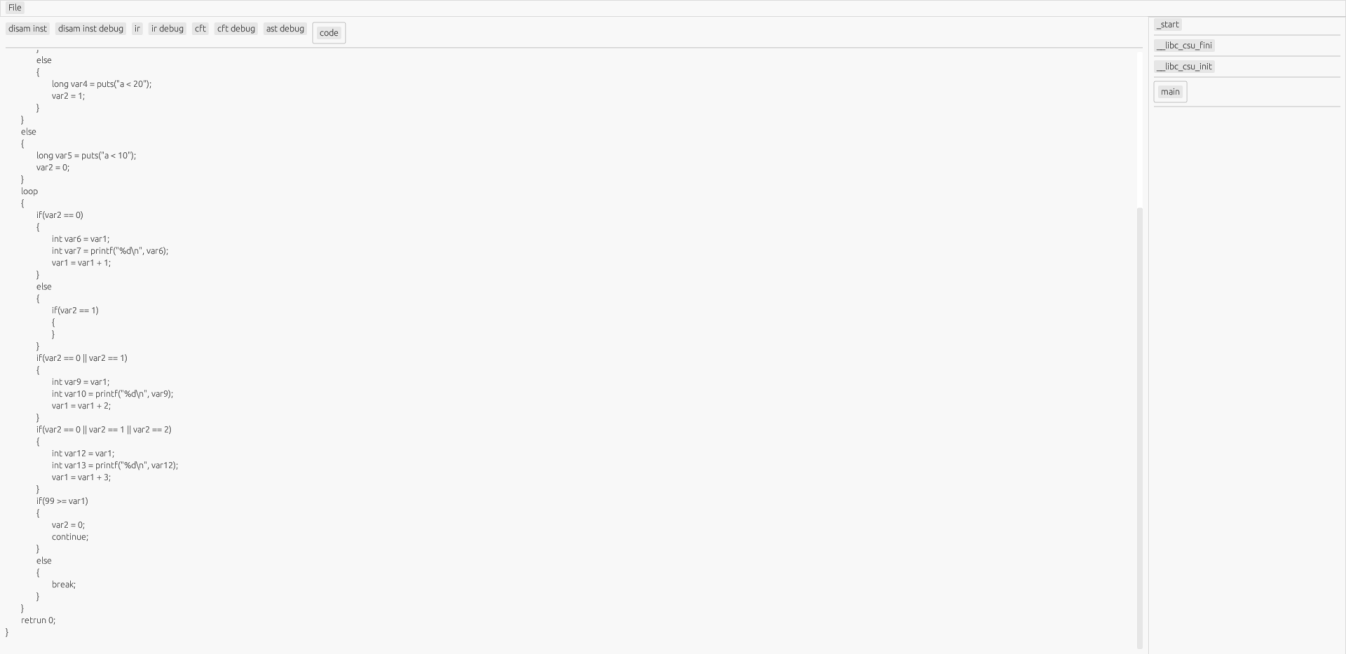
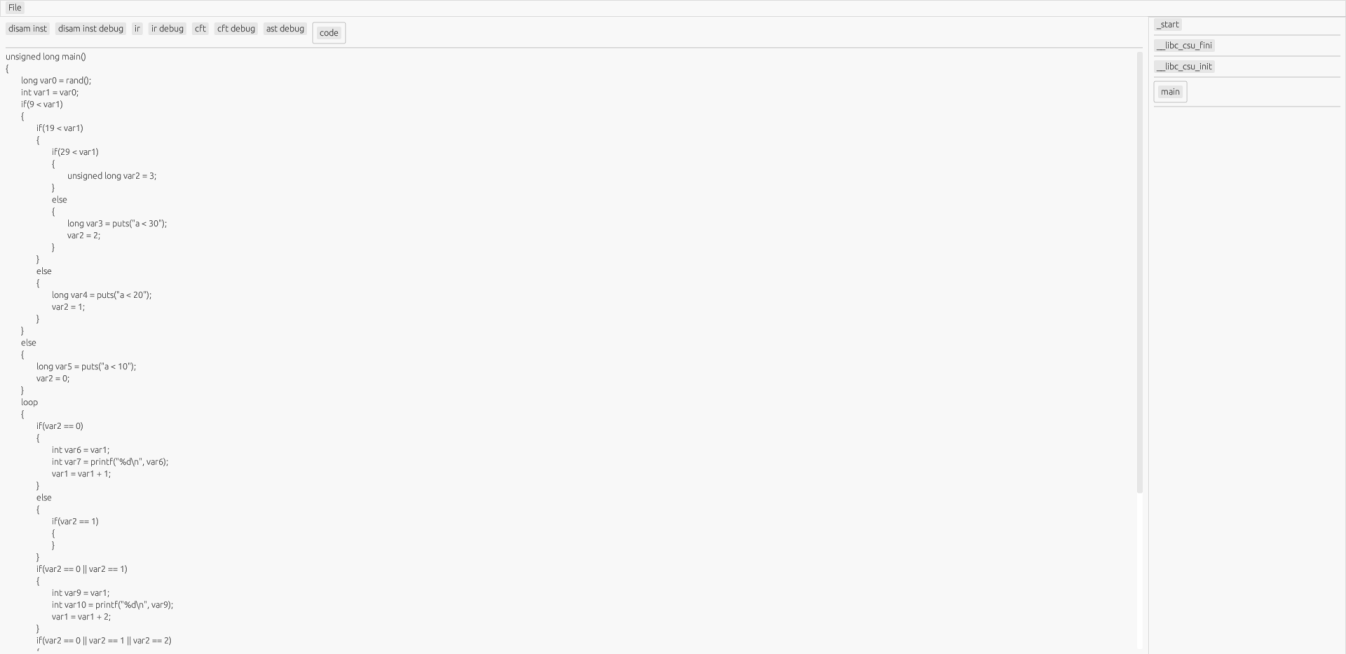


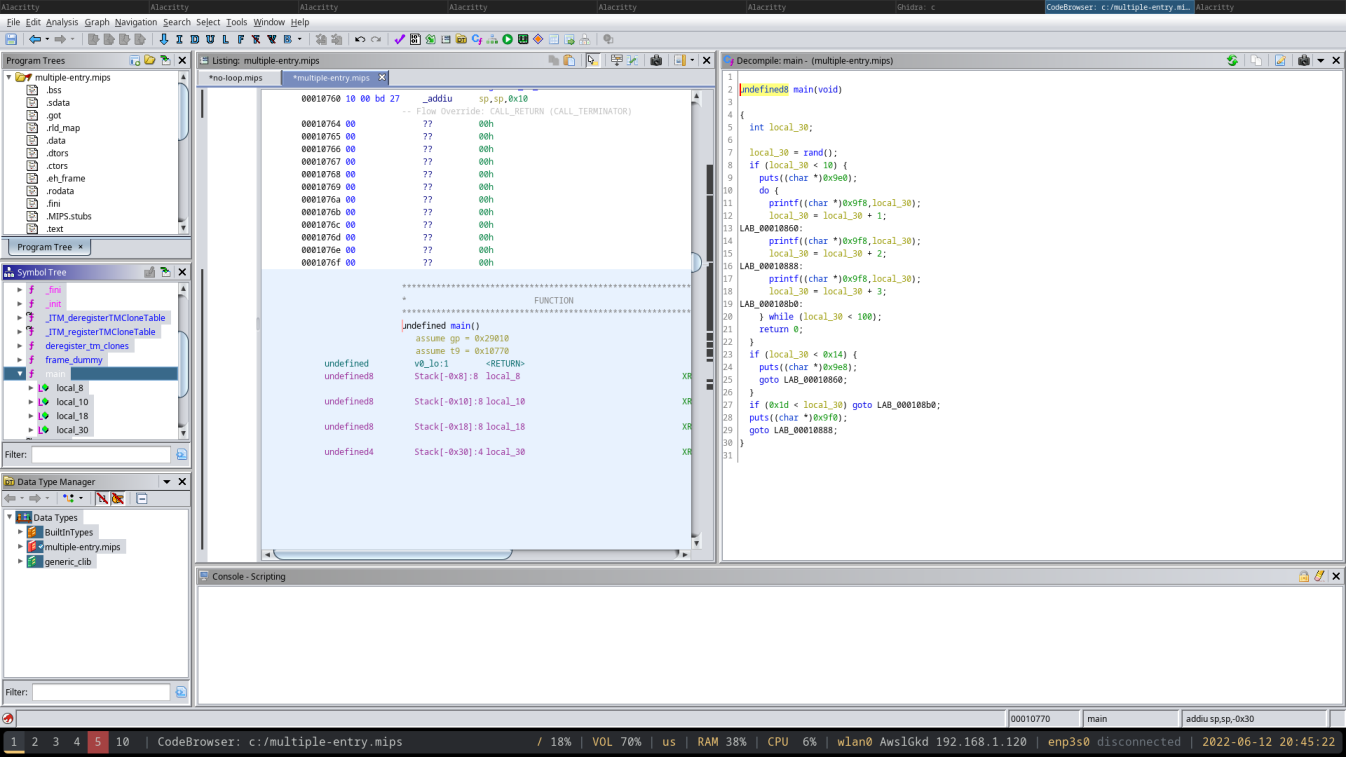


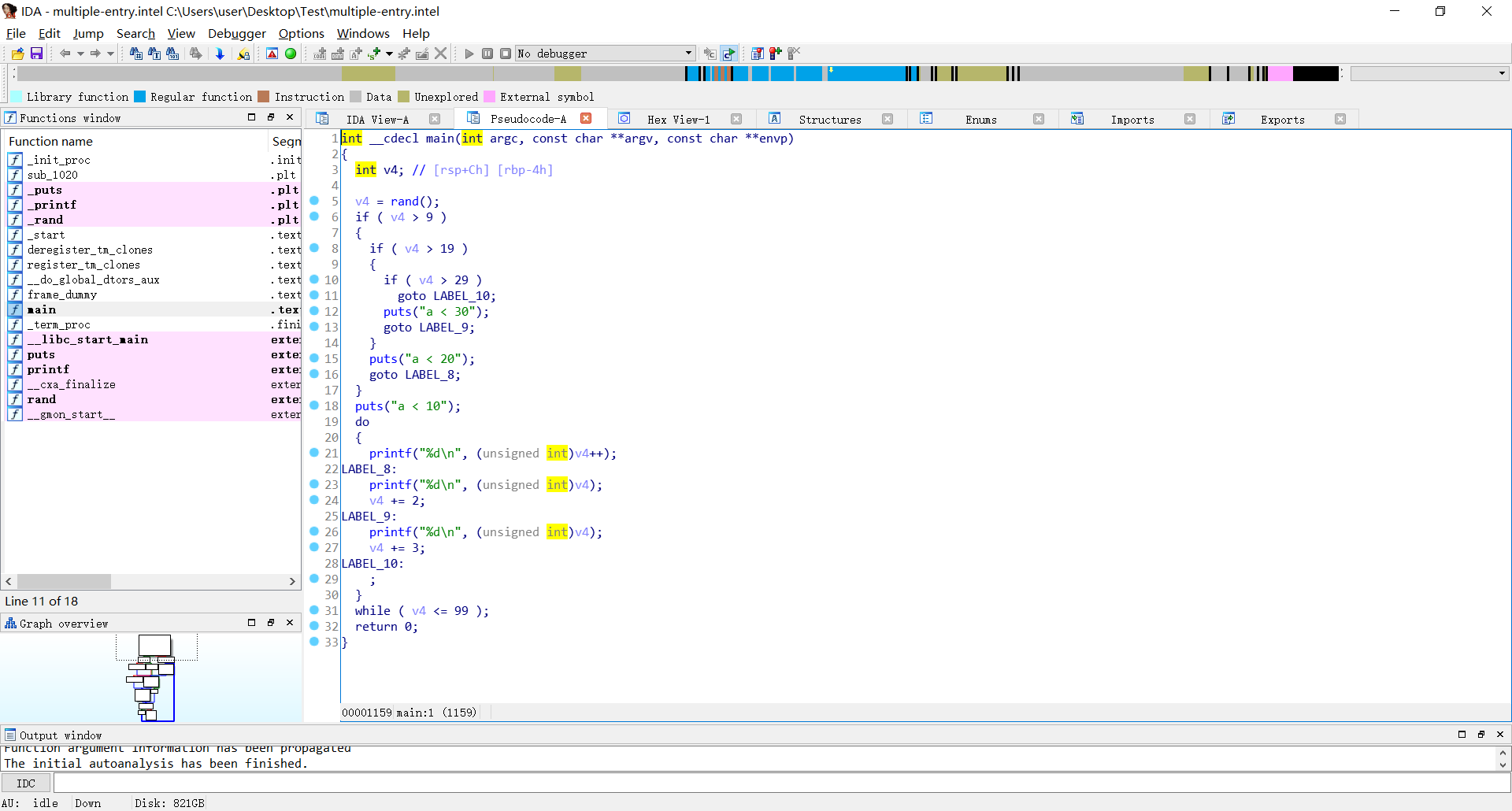
**3.4.3 测试多入口循环控制流**

源文件名为multiple-entry.c。文件中存在一个有两个入口的循环。

|  |
| --- |
| int main() {  int a = rand();  if(a < 10) {  puts("a < 10");  goto label1;  }  if(a < 20) {  puts("a < 20");  goto label2;  }  if(a < 30) {  puts("a < 30");  goto label3;  }  while(a < 100) {  label1:  printf("%d**\n**", a);  a += 1;  label2:  printf("%d**\n**", a);  a += 2;  label3:  printf("%d**\n**", a);  a += 3;  }  } |







**3.5 结果分析**

反编译器实现了基本的功能，正确分析了数据类型，分析出指针与指针操作、if..else结构、循环结构和while循环结构。

在上述例子中可以看出在这些情况下反编译器都没有生成goto语句并在特殊的循环中使用了结构化变量去重构循环。

# 第四章 创新性说明

1. 本反编译器的目标指令集是LoongArch，目前还没有相关的反编译器或反汇编引擎。

2.使用了不产生goto语句的控制流分析算法，在一定情况下生成的反编译代码可读性更好

3.由于当前在Loongson处理器相关的软件生态没有成熟，为了规避工具链与生态的缺陷，反编译器将编译为wasm并在浏览器中。

# 第五章 总结

在本作品中我们这对国产的LoongArch指令集设计了一个反编译器，并实现从解析可执行文件开始到反汇编最后到反编译这整套流程。

在反编译的过程中，我们也自己定义了数据流分析后产生的中间代码的格式，并使用了不产生goto语句的控制流分析算法。

在最后为了让反编译器实现跨平台，我们将反编译器编译为webassembly并让它在浏览器中部署。

# 参考文献

[1]Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, Matthew Smith.No More Gotos: Decompolation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations[R].[NDSS Symposium，2015](http://www.ndss-symposium.org/ndss2015)

[2]龙芯架构参考手册 - 卷一：基础架构

[3]龙芯架构 ELF psABI 规范