lab4实验报告

组长：

全映菱 PB17111644

小组成员：

陈炳楠 PB17111650

张陶竞 PB17111637

实验要求

1.配置LLVM使其支持RISCV，下载编译riscv-gnu-toolchain与spike模拟器，学习生成riscv可执行文件并使用spike模拟器来运行

2.阅读RegAllocFast.cpp源代码（为github中新版本），理解实际中LLVM进行寄存器分配的思想、方式、流程，与龙书中寄存器分配部分的理论知识进行比较

报告内容

1. RISC-V 机器代码的生成和运行

- LLVM 8.0.1适配RISC-V

  进入`llvm-build`文件夹，运行指令

  ```
  cmake -DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=RISCV -DLLVM_TARGETS_TO_BUILD=X86
  ../llvm
  make -j4
  ```

  等待编译完成，则LLVM8.0.1就适配了RISC-V。

  这个过程是参考了`issue#253`中的方式，在lab3的基础（我使用的是release版本）上进行增量编译，添加额外的编译选项。

  ```
  ylquan@ubuntu:~/CompilingLab/lab3/llvm-build$ cmake -DLLVM_EXPERIMENTAL_TARGETS_
  TO_BUILD=RISCV -DLLVM_TARGETS_TO_BUILD=X86 ../llvm
  -- Could NOT find LibXml2 (missing:  LIBXML2_LIBRARIES LIBXML2_INCLUDE_DIR)
  -- Native target architecture is X86
  -- Threads enabled.
  -- Doxygen disabled.
  -- Go bindings disabled.
  -- Could NOT find OCaml (missing:  OCAMLFIND OCAML_VERSION OCAML_STDLIB_PATH)
  -- Could NOT find OCaml (missing:  OCAMLFIND OCAML_VERSION OCAML_STDLIB_PATH)
  ```

  ```
  -- Generating done
  -- Build files have been written to: /home/ylquan/CompilingLab/lab3/llvm-build
  ylquan@ubuntu:~/CompilingLab/lab3/llvm-build$ make -j4
  [  1%] Built target LLVMDemangle
  [  1%] Built target LLVMTableGen
  [  3%] Built target LLVMBinaryFormat
  [  3%] Built target LLVMHello_exports
  [  6%] Built target obj.llvm-tblgen
  [  6%] Built target llvm_vcsrevision_h
  [ 11%] Built target LLVMSupport
  [ 11%] Built target LLVMMCParser
  [ 11%] Built target LLVMMCDisassembler
  [ 13%] Built target LLVMMCA
  ```

```
[100%] Linking CXX executable ../../bin/llvm-lto
[100%] Building CXX object tools/clang/tools/driver/CMakeFiles/clang.dir/cc1gen_
reproducer_main.cpp.o
[100%] Built target llvm-lto
[100%] Linking CXX executable ../../../../bin/clang
[100%] Linking CXX executable ../../bin/llvm-exegesis
[100%] Built target llvm-exegesis
[100%] Built target clang
ylquan@ubuntu:~/CompilingLab/lab3/llvm-build$
```

- lab3-0 GCD样例 LLVM IR 生成 RISC-V源码的过程

  ①为了使用RISC-V相关指令，首先需要下载riscv-gnu-toolchain，我使用的是：

```
git clone https://github.com/riscv/riscv-gnu-toolchain.git
cd riscv-gnu-toolchain
git submodule update --init --recursive
```

  i.`git clone`过程中由于网络情况或其他原因常常出现下载中断的情况，所以需要进入文件夹，使用`git submodule update --init --recursive`指令继续下载；

  ii.下载过程中可能由于代理问题中断下载，出现报错信息：

  `Failed to connect to github.com port 443: Connection refused`

  此时输入以下指令，并继续`git clone`即可：

```
git config --global --unset http.proxy
git config --global --unset https.proxy
```

  iii.下载过程中还可能由于无法连接外网导致无法下载，出现报错信息：

  `Failed to connect to boringssl.googlesource.com port 443: Connection refused`

  此时在github网站搜索对应的boringssl镜像，将对应仓库`clone`下来即可

  ②下载完成后，需要安装textinfo，否则会报warning无法编译；可以在当前文件夹中新建build文件夹，在build文件夹中进行编译（我此处直接在riscv-gnu-toolchian文件夹中进行编译，尝试过在build文件夹中编译，但由于pk的编译过程报错，又重新编译了riscv-gnu-toolchian，是在riscv-gnu-toolchain中直接编译的）：检查环境并生成当前环境使用的MakeFile，安装编译所需要的依赖

```
sudo apt-get install textinfo
mkdir build //可选
cd build //可选
../configure --prefix=$RISCV --enable-multilib
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool
patchutils bc zlib1g-dev libexpat-dev
vim ~/.bashrc
在最后增加两行：
export  RISCV="~/CompilingLab/lab3/riscv/riscv-gnu-toolchian"
export  PATH=$PATH:$RISCV/bin
source ~/.bashrc
sudo make
```

  i.我在第一次编译时并不清楚需要安装对应的依赖，所以在`make`过程中会报错：`configure: error: Building GCC requires GMP 4.2+, MPFR 2.4.0+ and MPC 0.8.0+.`，我又下载安装了`gmp-6.1.0`、`mpc-1.0.3`、`mpfr-3.1.4`，但不清楚是否由于没有配置好，再次`make`时依然报错；后来我运行了如上指令进行依赖的安装编译，才顺利完成了`make`

- 安装 Spike模拟器并运行上述生成的RISC-V源码

  ①安装spike，克隆spike对应的仓库并进行编译安装

```
git clone https://github.com/riscv/riscv-isa-sim.git
sudo apt-get install device-tree-compiler
cd riscv-isa-sim
mkdir build
cd build
../configure --prefix=$RISCV
make
sudo make install
```

  ②安装pk，克隆pk对应的仓库并进行编译安装

```
git clone --recursive https://github.com/riscv/riscv-pk.git
cd riscv-pk
mkdir build
cd build
../configure --prefix=$RISCV
make
sudo make install
```

  i.我在编译pk的过程中曾碰到报错`gcc: error: unrecognized argument in option '-mcmodel=medany'`，无法完成pk的编译，参考了网络上一篇教程（参考资料2-②），输入`vim ~/.bashrc`，在最后增加了路径信息，并重新进行riscv-gnu-toolchain的make，成功编译pk

  ③输入以下指令分步生成RISCV可执行文件，通过spike进行模拟执行

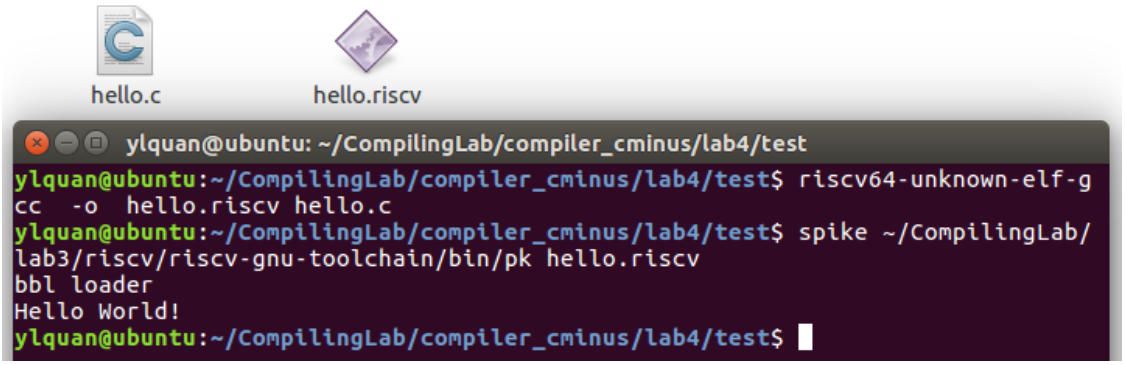
```
~/CompilingLab/lab3/llvm-build/bin/clang -emit-llvm --target=riscv64 -
march=rv64gc gcd.c -S -o gcd.ll -I/opt/riscv/riscv64-unknown-elf/include
~/CompilingLab/lab3/llvm-build/bin/llc -march=riscv64 gcd.ll -o gcd.s
riscv64-unknown-elf-gcc gcd.s -o gcd.riscv
spike ~/CompilingLab/lab3/riscv/riscv-gnu-toolchain/bin/pk gcd.riscv
```

  i.助教提供的tutorial中为相对路径，我此处为了避免路径配置错误，直接使用了绝对路径

  ii.我在安装pk之后运行`spike pk gcd.riscv`之后依然出现错误`terminate called after throwing an instance of 'std::runtime_error'`
  `what(): could not open pk (did you misspell it? If VCS, did you forget +permissive/+permissive-off?)`，同样参考教程后重新编译riscv-gnu-toolchian，并将安装编译好的pk可执行文件复制到`riscv-gnu-toolchain/bin`中，使用绝对路径，才成功使用模拟器：

iii.使用`spike pk`的测试，测试文件代码：

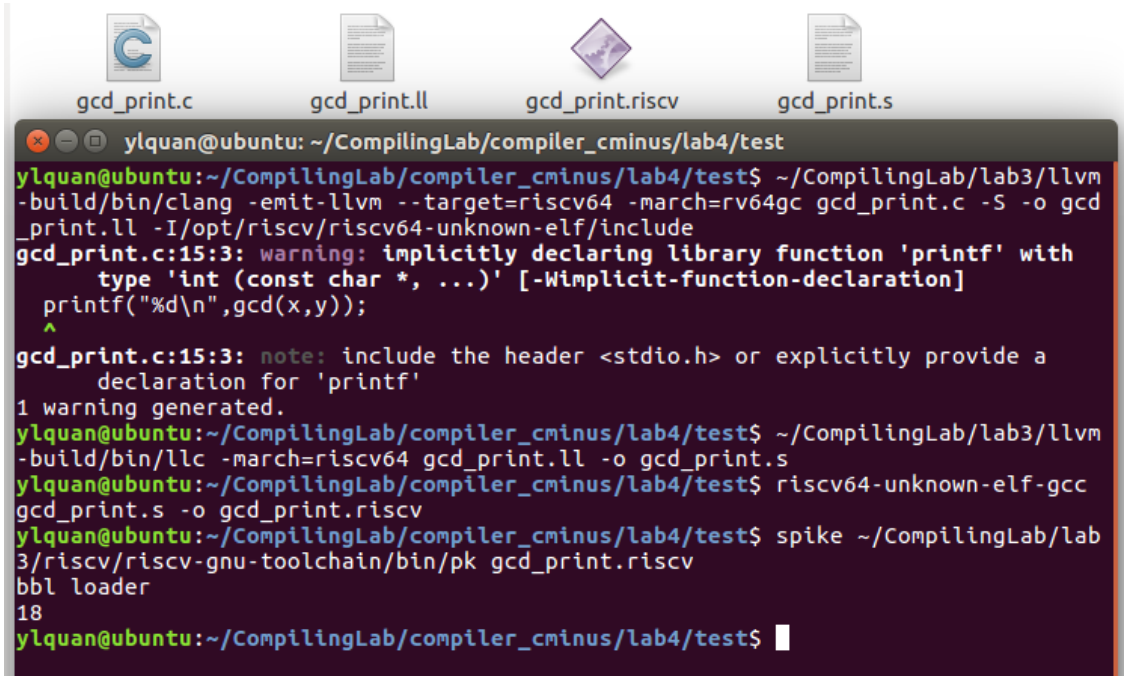
```
#include <stdio.h>
int main(void)
{
        printf("Hello World!\n");
        return 0;
}
```



iv.对于包含如`#include<stdio.h>`代码的.c文件，无法进行分步生成，但可以去掉`#include<stdio.h`，同时也可以使用`printf`函数进行输出（此时会报`warning`如图），以查看程序是否正确执行：

gcd_print.c文件代码：

```
/*#include <stdio.h>*/
int gcd (int u, int v) { /* calculate the gcd of u and v */
  if (v == 0) return u;
  else return gcd(v, u - u / v * v); /* v,u-u/v*v is equals to u mod v*/
}
int main() {
  int x; int y; int temp;
  x = 72;
  y = 18;
  if (x<y) {
    temp = x;
    x = y;
    y = temp;
  }
  printf("%d\n",gcd(x,y));
  return 0;
}
```



2. LLVM源码阅读与理解

- RegAllocFast.cpp 中的几个问题
  - *RegAllocFast* 函数的执行流程?

    答:

    1.RegAllocfast所用算法思想

    虚拟寄存器：由于LLVM是静态赋值的，所以虚拟寄存器是SSA的形式。
    fast allocator使用本地分配策略。它以线性方式从上到下在基本块级别扫描程序，在每一个基本块内按序遍历指令及其操作数。当虚拟寄存器出现时将它分配到物理寄存器。在基本块之间没有活跃的寄存器。在每个块结束时执行溢出操作。这种方法对未优化的生存期短的代码很有效，该分配器用于LLVM中的调试目的。

    RegAllocfast主要思想为线性扫描算法。

    在线性扫描算法中，我们以虚拟寄存器的live interval为单位进行分配，每个live interval的范围表示该虚拟寄存器在这个区间被使用了。首先，编译器要将控制流图CFG线性化并给每条指令编号。在LLVM里，是通过深度优先序来线性化CFG的，同时通过slot index 来给每条指令编号。其次，需要对IR做后向数据流分析，就可以获得每个变量的活跃信息，并进一步得到live interval。

    首先遍历没有处理的变量列表中的live interval,依次为其分配寄存器。首先，遍历一次active列表（active列表存储的是活跃且已分配物理寄存器的live interval，其中的live interval以结束点升序排列），判断在当前的interval的开始位置是否存在已经结束的interval，如果有则将其删除并将分配给它的寄存器回收用于后续分配。然后判断当前可用的寄存器数目是否为0,如果没有可用的寄存器，执行溢出。比较active列表的最后一个interval和当前interval哪一个的结束位置更迟，将结束位置更迟的interval溢出到stack slot。如果是当前的interval被溢出，需要为其分配一个栈上的位置；否则把分配给active列表中的寄存器分配给当前的interval并将它溢出到栈。

    注：下图来自Linear Scan Register Allocation（参考资料6.）

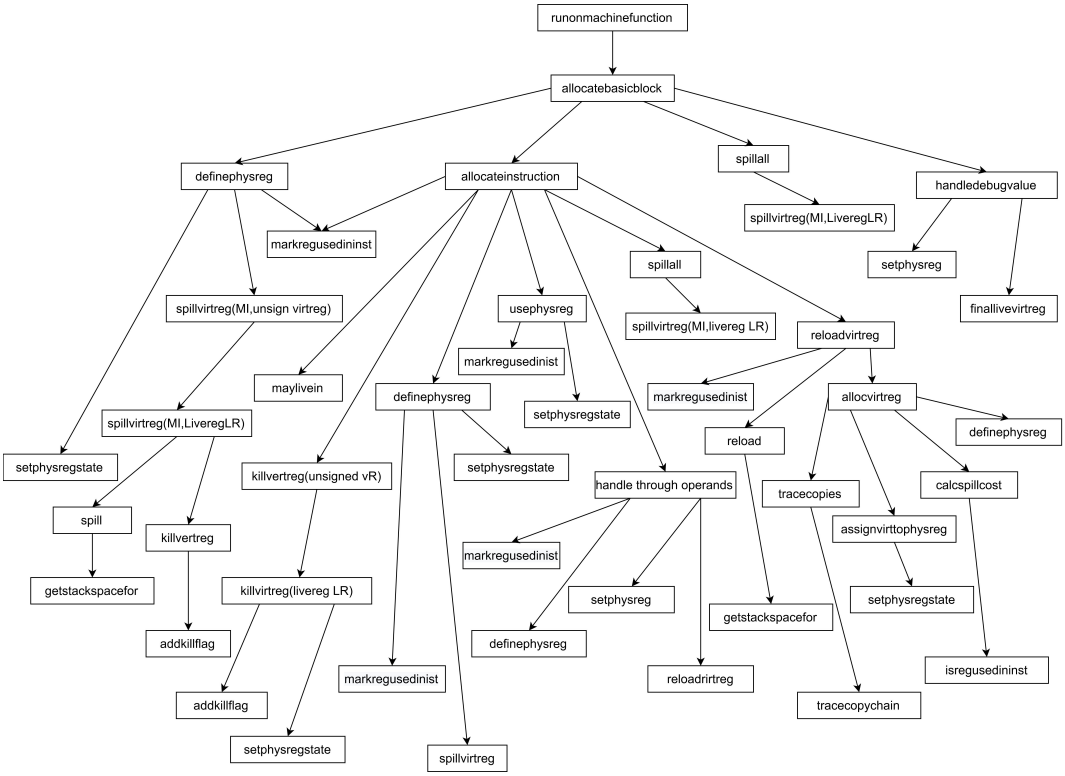
```
LinearScanRegisterAllocation
    active ← {}
    foreach live interval i, in order of increasing start point
        ExpireOldIntervals(i)
        if length(active) = R then
            SpillAtInterval(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

ExpireOldIntervals(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```

2.调用关系

首先调用`runonmachinefunction`函数，做了一些寄存器分配相关的初始化工作，之后调用`allocatebasicblock`对每一个基本块分析。在对每一个基本块的处理中，我们首先对活跃区间内可分配的寄存器的状态设置为regreserved,调用函数`definephysreg`，之后调用`allocateinstruction`来对每一条指令做出分析。在每一个basicblock结束时，将活跃的寄存器spill到栈。在对每一条指令的处理中，我们会通过4次扫描，对物理寄存器的状态进行标记，查找虚拟寄存器的末端的查找，对一些特殊的操作数进行处理，对约束条件做出设置，并且我们会根据设置条件和活跃性分析对寄存器的做出分配。`allocateinstruction`详见第三题的分析。



- *allocateInstruction* 函数有几次扫描过程以及每一次扫描的功能?

  答:

  1.有四次扫描过程

  2.每一次扫描的功能:

  ①第一次扫描:运行在可以分配来使用的物理寄存器上,标记物理寄存器的使用和earlyclobber的使用,找到虚拟寄存器操作数的结尾,并对它们调用`usephysreg`函数。在这一阶段,物理寄存器状态必须是regdisabled,regfree和regreserved,而不能被分配给虚拟寄存器;否则,如果这条指令是earlyclobber,我们就调用`definephysreg`函数在溢出虚拟寄存器之后将物理寄存器标记为regreserved或regfree,并将hasearlyclobber标记为true,否则标记hasphysdefs为true。之后,当有多个定义和相关的操作数,我们必须确保相同的寄存器分配,这时我们会调用`handlethroughoperands`函数处理有特殊限制的操作数。

  ②第二次扫描:运行在虚拟寄存器的使用上,即分配了虚拟寄存器的使用。我们通过`reloadvirtreg`函数将寄存器添加到livevirtregs中,并根据此时拿到的物理寄存器对应的虚拟寄存器来更改copysourcereg变量。之后调用`setphysreg`函数,它会更新物理寄存器和虚拟寄存器的映射关系,如果调用返回true,调用`setphysreg`函数,标记虚拟寄存器不再有效。之后,如果存在没有定义的使用,我们调用`allocvirtregundef`函数,将给它分配寄存器。之后,跟踪这条指令定义的寄存器查看是否有earlyclobbers,如果存在,调用`markregusedininst`函数,标记物理寄存器在这条指令中使用过。最后,如果这条指令是一个call类型的指令,我们会调用`spillall`函数将所有的寄存器溢出,这是为了使得当异常发生时,landingpad期待能在它们的溢出槽中找到一个寄存器。

  ③第三次扫描:运行在已定义的物理寄存器上,在分配是Defs的虚拟寄存器之前标记所有的Defs物理寄存器为使用过(Defs表示定值)。首先调用`definephysreg`函数,并将该物理寄存器标记为regreserved,除非`MO.isdead()`函数返回true,这种状况会将物理寄存器标记为regfree。

  ④第四次扫描:运行在已定义的虚拟寄存器上,分配Defs并收集死的Defs。首先调用`definevirtreg`函数,为虚拟寄存器分配物理寄存器并标记它为dirty(dirty指示寄存器需要spill)。之后,调用`setphysreg`函数,如果该函数返回true,就将虚拟寄存器压入到virtdead向量中,并将copydstreg标记为0;否则根据新分配的物理寄存器来对它做出设置。

- *calcSpillCost* 函数的执行流程?

答：

1.函数功能：

返回spill物理寄存器和它的所有alias的代价。
alias：与目标机器相关，以x86为例，AH，AX,EAX有相同的物理地址，LLVM将多个这样的物理寄存器作为units，其中每个unit是一个alias。

2.函数流程：

若一个物理寄存器或者它的任一alias正在这条指令中使用，不能spill到内存，返回spillImpossible。

否则，判断该物理寄存器的状态，若为
(1)regDisabled:表示物理寄存器不能用于分配，但是它的alias可能正在被使用
操作：无操作，之后对物理寄存器的alias操作
(2)regFree:表示物理寄存器当前没有被使用，且可以立即进行分配而不需要检查它的alias
操作：则返回0
(3)regReserved：表示物理寄存器被明确地分配（如设置调用的参数等），则在使用前保持保留状态
操作：不能spill，返回spillImpossible
(4)default：表示物理寄存器被映射到一个虚拟寄存器
操作：若寄存器需要spill，返回spillDirty（是enum类型的，值为100u);
若寄存器不需要spill，返回spillClean(是enum类型的，值为50u)。

对于disabled register，spill cost为它所有别名的spill cost之和
遍历该disabled registe的所有别名，对每个别名状态进行判断，若为
(1)regDisable：继续找下一个别名
(2)regFree：物理寄存器的spill cost自增1，继续找下一个别名
(3)regReserved：不能spill，返回spillImpossible
(4)default：
若寄存器需要spill，spill cost在原值上增加spillDirty
若寄存器不需要spill，spill cost在原值上增加spillClean

返回spill cost

- *hasTiedOps*，*hasPartialRedefs*，*hasEarlyClobbers* 变量的作用?

答：

1.hasearlyclobber：early-clobber是一种输入输出限制情况下的，用来使寄存器分配更加安全的一个标志，考虑以下情况：LLVM会将同一个寄存器分配给一个output和一个input，但这样可能不够安全，比如有两条汇编指令，第一条写了一个output，第二条读了一个input和第二个output，这时必须明确指出output是一个“early-clobber"的output。将一个output标记为"early-clobber"确保LLVM不将这个寄存器用于任一input，除非这个input与output关联。如果在所有输入寄存器被读前，定值操作数寄存器被机器指令写了，那么将hasearlyclobber值设置为true。之后会调用`handlethroughoperands`函数来处理这一特殊情况，在这个函数中，我们会调用`definevirtreg`函数,为虚拟寄存器分配一个物理寄存器并标记它为dirty。之后会调用`setphysreg`函数来更改物理寄存器对应的操作数，返回true的情况可能为:（1）该指令没有子寄存器并且这条指令iskill或isdead成立（2）该指令iskill成立，如果该函数返回true会将该寄存器压入virtdead向量中，会在所有扫描结束之后调用`killvirtreg`函数，标记virtdead向量中所有寄存器不再有效，来确保同一寄存器的多个定值分配相同。通过上述操作，我们可以解决寄存器存在earlyclobber时的寄存器分配问题。

2.haspartialredef：当对某个寄存器的子寄存器有了使用并且对该虚拟寄存器进行了读写操作，这时我们需要设置haspartialredef的值为true,说明它对某个大寄存器的一部分进行了定义和使用，为了保证不产生冲突，我们调用`handlethroughoperands`函数将部分定义中的所有物理寄存器标记为使用过来避免对它们的再次分配。通过上述操作，我们可以解决寄存器存在部分重定义情况下的寄存器分配问题。

3.hastiedoperands：表明此时出现了一种对寄存器的限制，这个寄存器操作数与另一个寄存器操作数关联了，即在定义时分配了相同的寄存器。我们需要调用`handlethroughoperands`函数对这种特殊的指令操作做出相应的处理。我们还需要调用`reloadvirtreg`函数，确保该操作数对应的虚拟寄存器在对应的物理寄存器中有效，之后它对应的物理寄存器调用`setphysreg`函数来更改这个操作数对应的寄存器。通过上述操作，我们可以解决寄存器操作数相关联情况下的寄存器分配问题。

- 书上所讲的算法与LLVM源码中的实现之间的不同点

  答：

  为虚拟寄存器分配物理寄存器，龙书算法中，当指令x=y为复制指令时,先分配Ry,再让Rx=Ry，当没有可供分配的物理寄存器时,需要计算所有已被分配的物理寄存器的spill cost，即spill时需生成的sw指令条数，选择其中cost的最小的，spill后分配；而fast alloator对Ry的选择先检查是否可以使用hint，若可以，就直接选择hint，若不能使用hint，才通过迭代选择spill cost最小的物理寄存器。在`allocateinstruction`函数中我们判断机器指令的`iscopy()`是否返回true，若为true，则我们将机器指令Opnum为0的操作数对应的寄存器赋给copydstreg，将copydstreg作为`reloadvirtreg`函数的Hint参数的值，`reloadvirtreg`里又将Hint作为`allocatevirtreg`函数的Hint0参数的值，在该函数中对Hint0判断，若Hint0是物理寄存器且可分配且在待分配的虚拟寄存器的类中，计算Hint0的spill cost，若spill cost<spillDirty(spillDirty值为100u),我们调用`definephysreg`函数来spill Hint0并将Hint0标记为regFree，再调用`assignvirttophysreg`函数将Hint0分配给虚拟寄存器，返回，即在Hint0可用的情况下，无需在所有物理寄存器中选择spill cost最小的物理寄存器，而直接将Hint0分配给虚拟寄存器。若Hint0不可用，调用`traceCopies`函数，该函数虚拟寄存器的定义是否为copy,若是则返回一个可合并的物理寄存器。在`allocatevirtreg`函数中为它尝试的第二个hint，设为Hint1。Hint1是否可用判断方法同Hint0。
  fast allocator在hint的spill cost小于某个特定值（这里为spillDirty=100u),就将hint分配给虚拟寄存器，而不迭代选择spill cost最小的物理寄存器，这样在spill cost小幅增大情况下，可以提高寄存器分配速度。

组内讨论内容

1.讨论1

时间：2019年12月14日19:00~20:00

地点：西区图书馆4楼

内容：展开lab4第一次讨论，主要给各小组成员分配任务：

①各小组成员均进行riscv编译环境的配置

②小组成员进行ResAllocFast.cpp源码阅读，任务分配如下（以助教提供的github链接中文件为准），其中数字为CPP文件中行号

全映菱：50~452

张陶竞：453~883

陈炳楠：886~1328

2.讨论2

时间：2019年12月16日20:00~23:00

地点：西区图书馆4楼

内容：展开第二次讨论，主要开始进行源码阅读的整体理解，各小组成员讲解自己负责阅读部分的代码操作与功能，尝试解决小组成员在配置riscv编译环境时遇到的问题：

①小组成员尽管都进行了两天的尝试，但依然没有完成riscv编译环境的配置，需要继续进行

②全映菱同学负责的源码阅读部分基本完成说明，张陶竞、陈炳楠同学的部分留待第二天讨论进行讲解

3.讨论3

时间：2019年12月17日19:00~23:00

地点：西区图书馆4楼

内容：展开第三次讨论，主要为继续进行ResAllocFast.cpp文件的代码阅读，张陶竞、陈炳楠同学讲解自己负责部分代码的操作与功能；由于此时全映菱同学已完成riscv-gnu-toolchain文件的git clone，其余两位同学尚未完成，因此全映菱同学继续进行环境配置，完成实验的第一部分；陈炳楠、张陶竞同学继续进行源代码理解：

①全映菱同学基本完成实验第一部分，分步生成了可执行文件

②陈炳楠、张陶竞同学进行源代码理解，初步完成源代码梳理

4.讨论4

时间：2019年12月18日20:00~23:00

地点：西区图书馆4楼

内容：展开第四次讨论，继续进行源代码理解，学习龙书中寄存器分配的理论知识：

①陈炳楠、张陶竞同学查找LLVM寄存器分配函数相关的资料与文档

②全映菱同学阅读龙书中指定章节的内容

5.讨论5

时间：2019年12月19日20:00~23:00

地点：西区图书馆4楼

内容：展开第五次讨论，进行源代码流程梳理，开始写实验报告：

①陈炳楠、张陶竞同学梳理源代码的流程，

②全映菱同学基本完成实验第一部分的报告

6.学习记录

时间：2019年12月20日晚

由于西区图书馆4楼没有位置，陈炳楠、张陶竞同学在学生宿舍继续完成第二部分的报告，全映菱同学负责将已完成的报告部分合并

7.学习记录

时间：2019年12月21日上午至下午

陈炳楠、张陶竞同学继续完成第二部分的报告，全映菱同学将已完成的部分合并，完成报告。

实验总结

1.本次实验学习了如何配置LLVM使其支持RISCV，并通过下载安装riscv-gnu-toolchian、spike、pk等使用spike模拟器执行riscv可执行文件，简单学习了分步生成riscv可执行文件与执行的操作

2.本次实验了解了许多寄存器分配问题的相关算法，从图的着色算法到线性扫描算法再到基于ssa的线性扫描算法，它们有不同的机制和策略。在LLVM中，我们也关注到了其它的寄存器分配算法，regallocbasic，regallocgreedy和regallocPBQP，它们在不同的情况下各有优势，学习它们也让我们对寄存器分配问题有了更好的了解。

实验反馈

1.安装riscv-gnu-toolchain及其他所需可执行文件的过程中，实在不清楚碰到的问题到底要怎么解决，比如路径配置的问题，我不清楚应该怎么配置对应的路径，只能重新参考另一篇教程，重新进行长达两个小时的riscv-gnu-toolchain的编译，尝试是否能够不出现报错

2.`git clone`与安装riscv-gnu-toolchain所需内存较大，小组同学的虚拟机内存不一定足够，下载到一半就前功尽弃；而且编译安装的过程时间很长（我编译所需要的时间大概在两个小时），期间卡顿明显，甚至影响计算机的Windows系统，全方位卡死，不得不进行强制重启，损害硬件

3.阅读源码的过程开始时还是很痛苦的，由于对核心算法的实现、操作等不够熟悉，在阅读开始时无法将它们对应起来，需要查阅大量的资料，才能收集到足够的信息，部分概念只是我们自己的猜测与理解

参考资料

1.Christian Wimmer,Michael Franz. 2010. *Linear Scan Register Allocation on SSA Form*

2.Tiago Cariolano de Souza Xavier, George Souza Oliveira, Ewerton Daniel de Lima and Anderson Faustino da Silva. 2012. *A Detailed Analysis of the LLVM's Register Allocators*

3.MASSIMILIANO POLETTO, VIVEK SARKAR. 1999. *Linear Scan Register Allocation*

4.如何继续中断的`git clone`

5.如何安装RISCV-gnu-toolchain与模拟器

①如何安装RISCV-gnu-toolchain与模拟器i.

②如何安装RISCV-gnu-toolchain与模拟器ii.

6.如何解决github port:443 connection refused

7.inline asm

8.register allocation in llvm

9.Register_allocation

①register allocation_i

②register allocation_ii

10.寄存器分配问题

11.llvm里的寄存器分配

12.llvm里的寄存器分配：线性扫描算法

13.llvm里的寄存器分配：basic分配器

14.LLVM 的SSA介绍

15.inside a register allocator

16.early clobber参考的官方文档

17.[LLVMdev] Fast register allocation