软件所智能软件中心PLCT实验室 王鹏 实习生

2020//04/29

目 录

• 01 解决rvv-llvm问题

llvm-mc -disassemble 的定义在llvm/tools/llvm-mc.cpp中有，主要是做为llvm-mc -help时的一种option。

具体的定义在Disassembler.cpp中，主要功能是实现从标准输入或文件中以十六进制写入的字节字符串的反汇编程序。

static bool PrintInsts（）函数

```
int Disassembler::disassemble(const Target &T,
 const std::string &Triple, MCSubtargetInfo &STI,
 MCStreamer &Streamer, MemoryBuffer &Buffer,
 SourceMgr &SM, raw_ostream &Out) {}
```

```
# RUN: llvm-mc %s -triple=riscv32 --mattr=+v -riscv-no-aliases -show-encoding \
# RUN:     | FileCheck -check-prefixes=CHECK-ASM,CHECK-ASM-AND-OBJ %s


# CHECK-ASM-AND-OBJ: vsetvl a3, a3, a2
# CHECK-ASM: encoding: [0xd7,0xf6,0xc6,0x80]
vsetvl a3, a3, a2

# CHECK-ASM-AND-OBJ: vsetvli a0, a1, e16,m2,d4
# CHECK-ASM: encoding: [0x57,0xf5,0x55,0x04]
vsetvli a0, a1, e16,m2,d4

# CHECK-ASM-AND-OBJ: vsetvli a0, a1, e1024,m8,d8
# CHECK-ASM: encoding: [0x57,0xf5,0xf5,0x07]
vsetvli a0, a1, e1024, m8, d8
```

test/MC/RISCV/rvv-valid.s

```
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/clang/include/clang/Basic$ echo "0xd7,0xf6,0xc6,0x80" | llvm-mc  -disassemble  -triple=riscv32
        .text
<stdin>:1:1: warning: invalid instruction encoding
0xd7,0xf6,0xc6,0x80
^
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/clang/include/clang/Basic$ echo "0xd7,0xf6,0xc6,0x80" | llvm-mc  -disassemble  -triple=riscv32 --mattr=+v
        .text
        vsetvl  a3, a2, Stack dump:
0.      Program arguments: llvm-mc -disassemble -triple=riscv32 --mattr=+v
#0 0x0000559e64ffc72a llvm::sys::PrintStackTrace(llvm::raw_ostream&) (/opt/llvm/bin/llvm-mc+0x1f272a)
#1 0x0000559e64ffa504 llvm::sys::RunSignalHandlers() (/opt/llvm/bin/llvm-mc+0x1f0504)
#2 0x0000559e64ffa642 SignalHandler(int) (/opt/llvm/bin/llvm-mc+0x1f0642)
#3 0x00007f0ed626c890 __restore_rt (/lib/x86_64-linux-gnu/libpthread.so.0+0x12890)
#4 0x0000559e64f48413 EvaluateSymbolicAdd(llvm::MCAssembler const*, llvm::MCAsmLayout const*, llvm::DenseMap<llvm::MCSection const*, unsigned long, llvm:
:DenseMapInfo<llvm::MCSection const*>, llvm::detail::DenseMapPair<llvm::MCSection const*, unsigned long> > const*, bool, llvm::MCValue const&, llvm::MCSy
mbolRefExpr const*, llvm::MCSymbolRefExpr const*, long, llvm::MCValue&) (/opt/llvm/bin/llvm-mc+0x13e413)
#5 0x00007ffc69b91af0
Segmentation fault (core dumped)
```

echo "0xd7,0xf6,0xc6,0x80" | llvm-mc -disassemble -triple=riscv32 --mattr=+v 反汇编测试用例

不加属性会报错

```
# RUN: llvm-mc %s -triple=riscv32 --mattr=+v -riscv-no-aliases -show-encoding \
# RUN:     | FileCheck -check-prefixes=CHECK-ASM,CHECK-ASM-AND-OBJ %s


# CHECK-ASM-AND-OBJ: vsetvli a0, a1, e8,m1,d1
# CHECK-ASM: encoding: [0x57,0xf5,0x05,0x00]
vsetvli a0, a1, 0

# CHECK-ASM-AND-OBJ: vsetvli a0, a1, e1024,m8,d8
# CHECK-ASM: encoding: [0x57,0xf5,0xf5,0x7f]
vsetvli a0, a1, 0x7ff

# CHECK-ASM-AND-OBJ: vlb.v v0, 0(a1)
# CHECK-ASM: encoding: [0x07,0x80,0x05,0x12]
vlb.v v0, (a1)
```

rvv-pseudo-valid.s中vsetvli和vsetvli指令的CHECK-ASM-AND-OBJ不对，vlb指令也前两者disassemble结果不同。

```
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/clang/include/clang/Basic$ echo "0x57,0xf5,0x05,0x00" | llvm-mc  -disassemble  -triple=riscv32 --mattr=+v
        .text
        vsetvli a1, 0, e256,m4,d4
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/clang/include/clang/Basic$ echo "0x57,0xf5,0xf5,0x7f" | llvm-mc  -disassemble  -triple=riscv32 --mattr=+v
        .text
        vsetvli a1, 2047, e256,m4,d4
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/clang/include/clang/Basic$ echo "0x07,0x80,0x05,0x12" | llvm-mc  -disassemble  -triple=riscv32 --mattr=+v
        .text
        vlb.v   v0, Stack dump:
0.      Program arguments: llvm-mc -disassemble -triple=riscv32 --mattr=+v
#0 0x000055e0bc0e072a llvm::sys::PrintStackTrace(llvm::raw_ostream&) (/opt/llvm/bin/llvm-mc+0x1f272a)
#1 0x000055e0bc0de504 llvm::sys::RunSignalHandlers() (/opt/llvm/bin/llvm-mc+0x1f0504)
#2 0x000055e0bc0de642 SignalHandler(int) (/opt/llvm/bin/llvm-mc+0x1f0642)
#3 0x00007f923302e890 __restore_rt (/lib/x86_64-linux-gnu/libpthread.so.0+0x12890)
#4 0x000055e0bc02c413 EvaluateSymbolicAdd(llvm::MCAssembler const*, llvm::MCAsmLayout const*, llvm::DenseMap<llvm::MCSection const*, unsigned long, llvm:
:DenseMapInfo<llvm::MCSection const*>, llvm::detail::DenseMapPair<llvm::MCSection const*, unsigned long> > const*, bool, llvm::MCValue const&, llvm::MCSy
mbolRefExpr const*, llvm::MCSymbolRefExpr const*, long, llvm::MCValue&) (/opt/llvm/bin/llvm-mc+0x13e413)
#5 0x000055e0bc24bd17 _fini (/opt/llvm/bin/llvm-mc+0x35dd17)
Segmentation fault (core dumped)
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/clang/include/clang/Basic$
```

rvv-pseudo-valid.s中三条指令测试结果

rvv-mask-valid.s和rvv-pseudo-mask-valid.s也存在Stack dump的情况

```
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/clang/include/clang/Basic$ echo "0x73,0x23,0x20,0xc2" | llvm-mc  -disassemble  -triple=riscv32 --mattr=+v
        .text
        csrr    t1, vlenb
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/clang/include/clang/Basic$ echo "0xf3,0x23,0x20,0xc2" | llvm-mc  -disassemble  -triple=riscv32 --mattr=+v
        .text
        csrr    t2, vlenb
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/clang/include/clang/Basic$ echo "0x73,0x23,0x10,0xc2" | llvm-mc  -disassemble  -triple=riscv32 --mattr=+v
        .text
        csrr    t1, vtype
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/clang/include/clang/Basic$ echo "0xf3,0x23,0x10,0xc2" | llvm-mc  -disassemble  -triple=riscv32 --mattr=+v
        .text
        csrr    t2, vtype
```

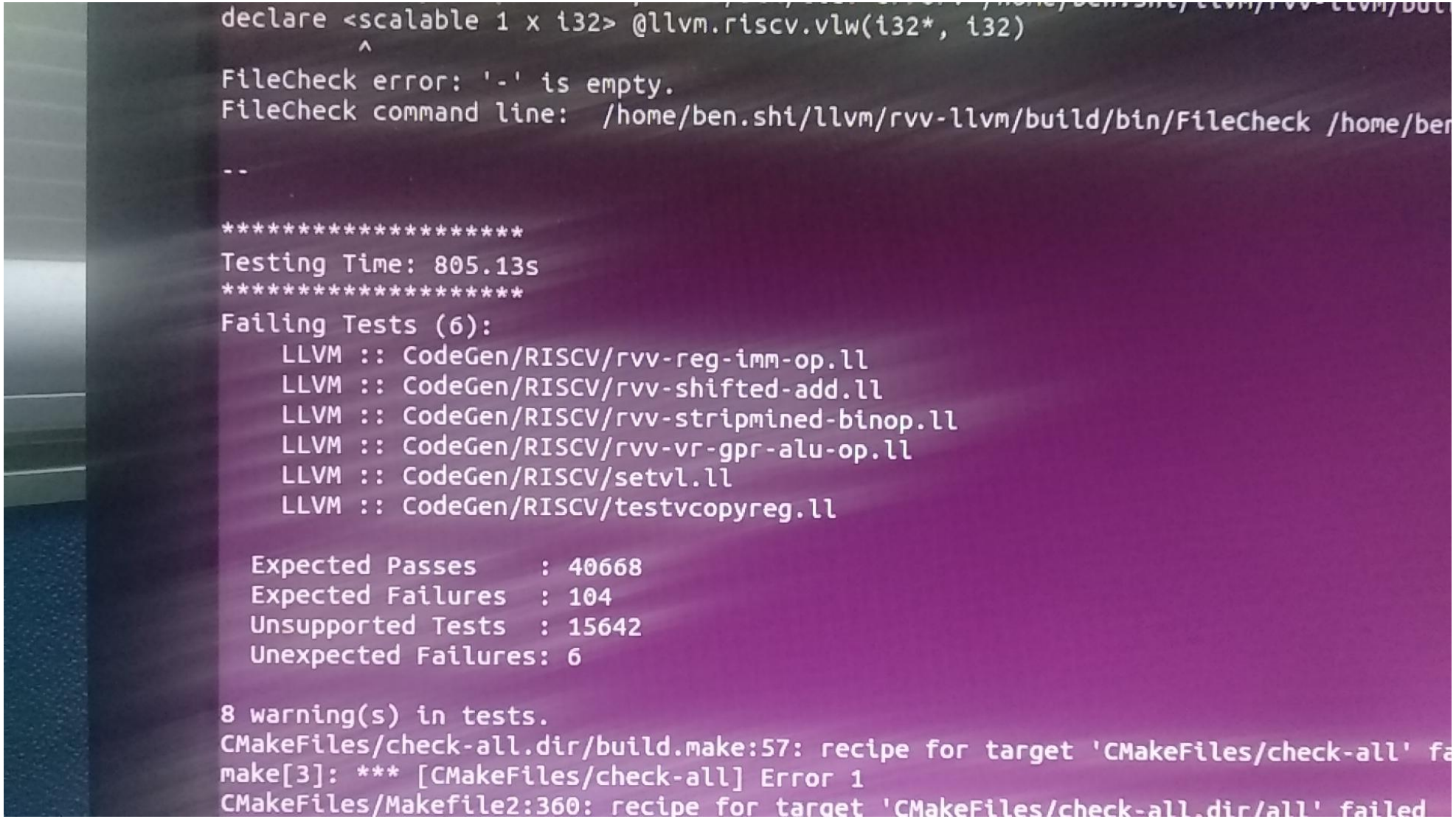
user-csr-names.s

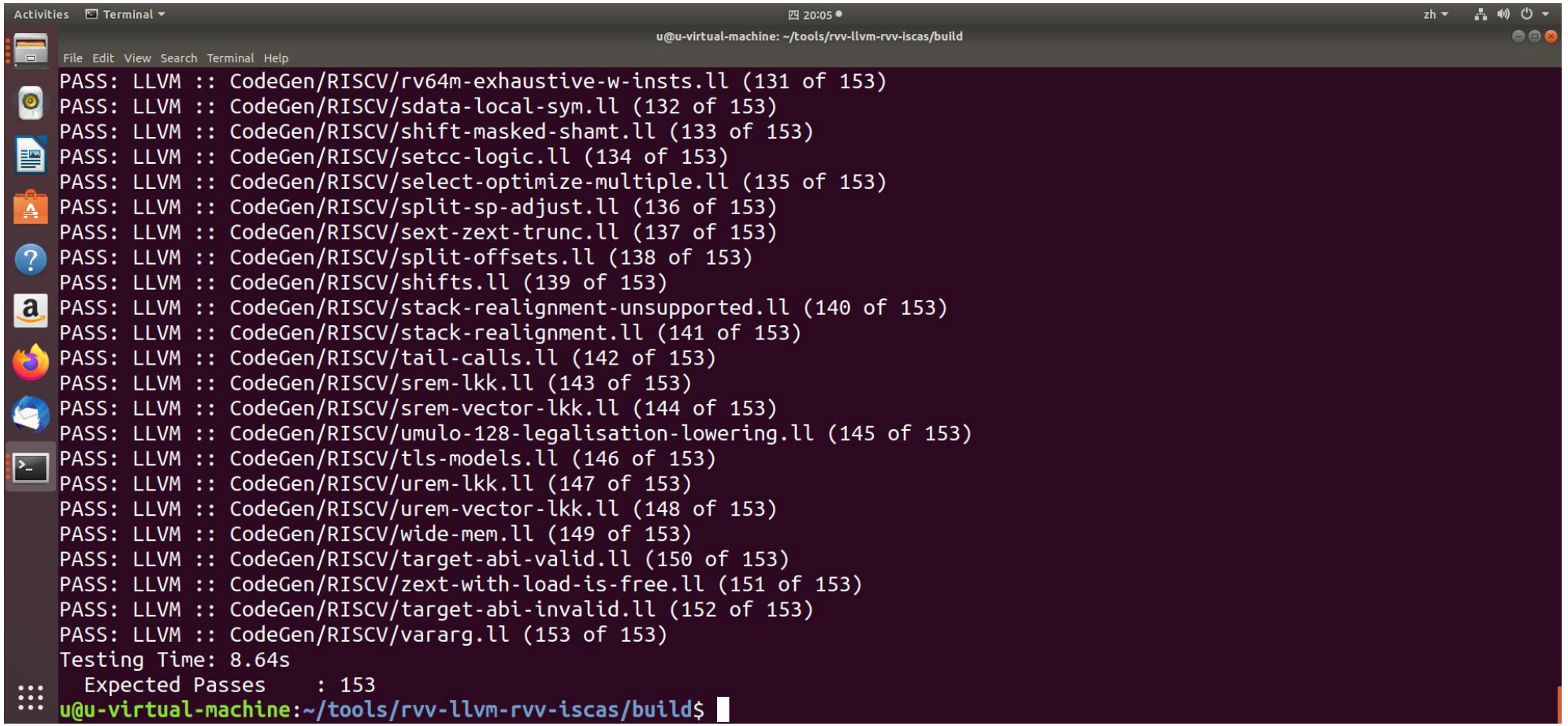
disassemble以后没有问题

```
# vtype
# name
# CHECK-INST: csrrs t1, vtype, zero
# CHECK-ENC:  encoding: [0x73,0x23,0x10,0xc2]
# CHECK-INST-ALIAS: csrr t1, vtype
# uimm12
# CHECK-INST: csrrs t2, vtype, zero
# CHECK-ENC:  encoding: [0xf3,0x23,0x10,0xc2]
# CHECK-INST-ALIAS: csrr t2, vtype
# name
csrrs t1, vtype, zero
# uimm12
csrrs t2, 0xC21, zero

# vlenb
# name
# CHECK-INST: csrrs t1, vlenb, zero
# CHECK-ENC:  encoding: [0x73,0x23,0x20,0xc2]
# CHECK-INST-ALIAS: csrr t1, vlenb
# uimm12
# CHECK-INST: csrrs t2, vlenb, zero
# CHECK-ENC:  encoding: [0xf3,0x23,0x20,0xc2]
# CHECK-INST-ALIAS: csrr t2, vlenb
# name
csrrs t1, vlenb, zero
# uimm12
csrrs t2, 0xC22, zero
```

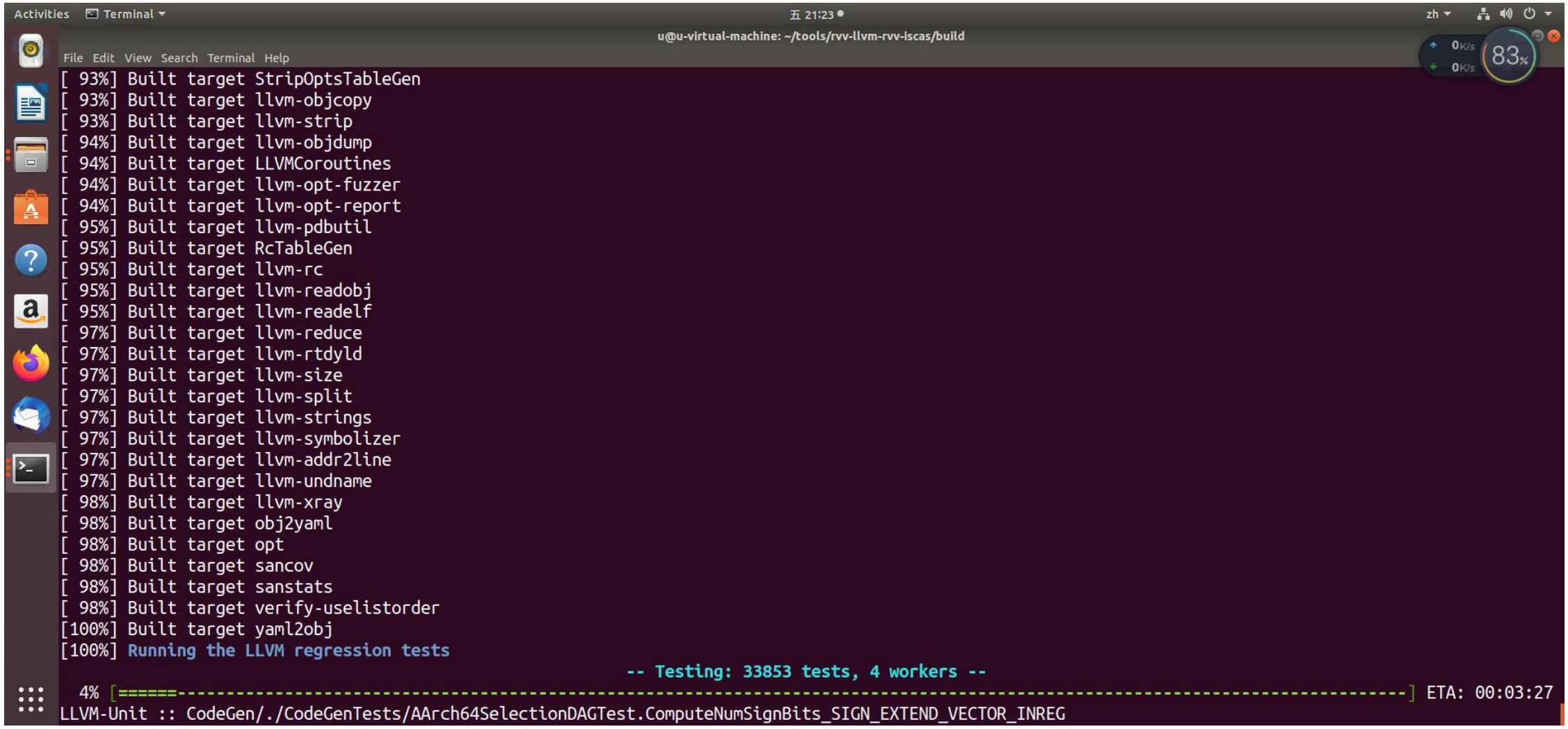
user-csr-names.s
中的vtyoe和vlenb
两个系统CSR

issue #19

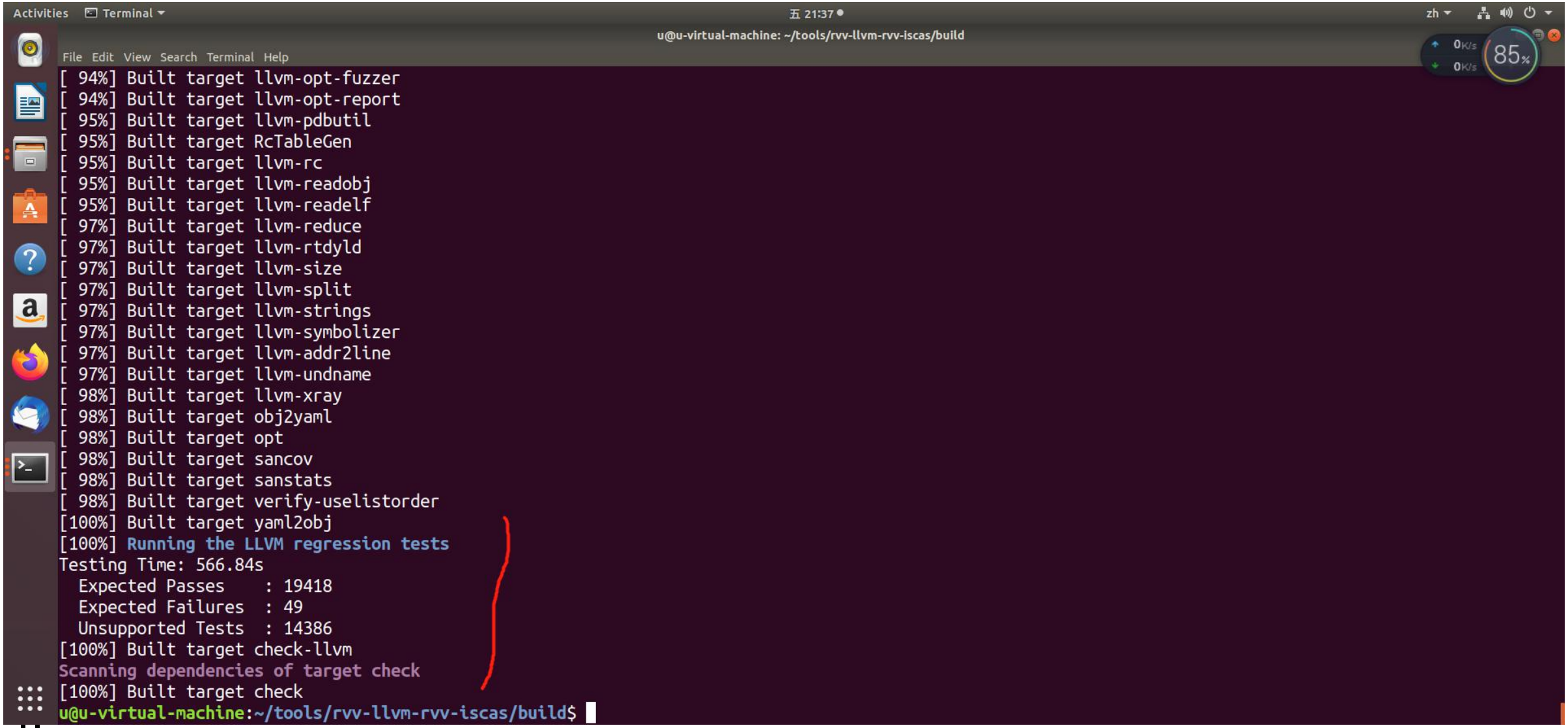

declare <scalable 1 x i32> @llvm.riscv.vlw(i32*, i32)
^
FileCheck error: '-' is empty.
FileCheck command line: /home/ben.shi/llvm/rvv-llvm/build/bin/FileCheck /home/ben
--
****
Testing Time: 805.13s
****
Failing Tests (6):
LLVM :: CodeGen/RISCV/rvv-reg-imm-op.ll
LLVM :: CodeGen/RISCV/rvv-shifted-add.ll
LLVM :: CodeGen/RISCV/rvv-stripmined-binop.ll
LLVM :: CodeGen/RISCV/rvv-vr-gpr-alu-op.ll
LLVM :: CodeGen/RISCV/setvl.ll
LLVM :: CodeGen/RISCV/testvcopyreg.ll
Expected Passes : 40668
Expected Failures : 104
Unsupported Tests : 15642
Unexpected Failures: 6
8 warning(s) in tests.
CMakeFiles/check-all.dir/build.make:57: recipe for target 'CMakeFiles/check-all' fa
make[3]: *** [CMakeFiles/check-all] Error 1
CMakeFiles/Makefile2:360: recipe for target 'CMakeFiles/check-all.dir/all' failed



在测试用例的注释命令行里改成总是通过？


Activities
Terminal
五 21:23
zh
u@u-virtual-machine: ~/tools/rvv-llvm-rvv-iscas/build
File Edit View Search Terminal Help
0K/s
0K/s
83%
[93%] Built target StripOptsTableGen
[93%] Built target llvm-objcopy
[93%] Built target llvm-strip
[94%] Built target llvm-objdump
[94%] Built target LLVMCoroutines
[94%] Built target llvm-opt-fuzzer
[94%] Built target llvm-opt-report
[95%] Built target llvm-pdbutil
[95%] Built target RcTableGen
[95%] Built target llvm-rc
[95%] Built target llvm-readobj
[95%] Built target llvm-readelf
[97%] Built target llvm-reduce
[97%] Built target llvm-rtdyld
[97%] Built target llvm-size
[97%] Built target llvm-split
[97%] Built target llvm-strings
[97%] Built target llvm-symbolizer
[97%] Built target llvm-addr2line
[97%] Built target llvm-undname
[98%] Built target llvm-xray
[98%] Built target obj2yaml
[98%] Built target opt
[98%] Built target sancov
[98%] Built target sanstats
[98%] Built target verify-uselistorder
[100%] Built target yaml2obj
[100%] Running the LLVM regression tests
-- Testing: 33853 tests, 4 workers --
4% [======----------] ETA: 00:03:27
LLVM-Unit :: CodeGen/./CodeGenTests/AArch64SelectionDAGTest.ComputeNumSignBits_SIGN_EXTEND_VECTOR_INREG

llv


Activities
Terminal
五 21:37
zh
u@u-virtual-machine: ~/tools/rvv-llvm-rvv-iscas/build
File Edit View Search Terminal Help
0K/s
0K/s
85%
[94%] Built target llvm-opt-fuzzer
[94%] Built target llvm-opt-report
[95%] Built target llvm-pdbutil
[95%] Built target RcTableGen
[95%] Built target llvm-rc
[95%] Built target llvm-readobj
[95%] Built target llvm-readelf
[97%] Built target llvm-reduce
[97%] Built target llvm-rtdyld
[97%] Built target llvm-size
[97%] Built target llvm-split
[97%] Built target llvm-strings
[97%] Built target llvm-symbolizer
[97%] Built target llvm-addr2line
[97%] Built target llvm-undname
[98%] Built target llvm-xray
[98%] Built target obj2yaml
[98%] Built target opt
[98%] Built target sancov
[98%] Built target sanstats
[98%] Built target verify-uselistorder
[100%] Built target yaml2obj
[100%] Running the LLVM regression tests
Testing Time: 566.84s
Expected Passes : 19418
Expected Failures : 49
Unsupported Tests : 14386
[100%] Built target check-llvm
Scanning dependencies of target check
[100%] Built target check
u@u-virtual-machine:~/tools/rvv-llvm-rvv-iscas/build$

llv

```
llvm/build/bin$ make check

[100%] Running the LLVM regression tests
Testing Time: 566.84s
  Expected Passes    : 19418
  Expected Failures  : 49
  Unsupported Tests  : 14386
[100%] Built target check-llvm
Scanning dependencies of target check
[100%] Built target check
```

通过网上查找相关解决方案，突然想起来了自己编译release版本时候没有出现这个问题，但是debug版本的时候就会出现这个问题，发现原来是swap空间不够了。
接下来就创建更多的swap空间（这里创建了20G）：

```
sudo mkdir swapfile
cd /swapfile
sudo dd if=/dev/zero of=swap bs=1024 count=20000000
sudo mkswap -f  swap
sudo swapon swap
```

如果需要卸载这个 swap 文件，可以进入建立的 swap 文件目录。执行下列命令。

```
sudo swapoff swap
```

• 02 LLVM实例

```
u@u-virtual-machine:~/tools/test$ cat add.cc

int add(int a, int b)
{
    return (a+b);
}
```

1. 在编译LLVM的时候，需要使用cmake -DLLVM_ENABLE_ASSERTIONS=ON才会启用view-dags等选项，这些选项下面会被用于查看输出的DAG图。
2. 安装xdot等dot查看工具，Ubuntu可以使用sudo apt-get install xdot命令安装。

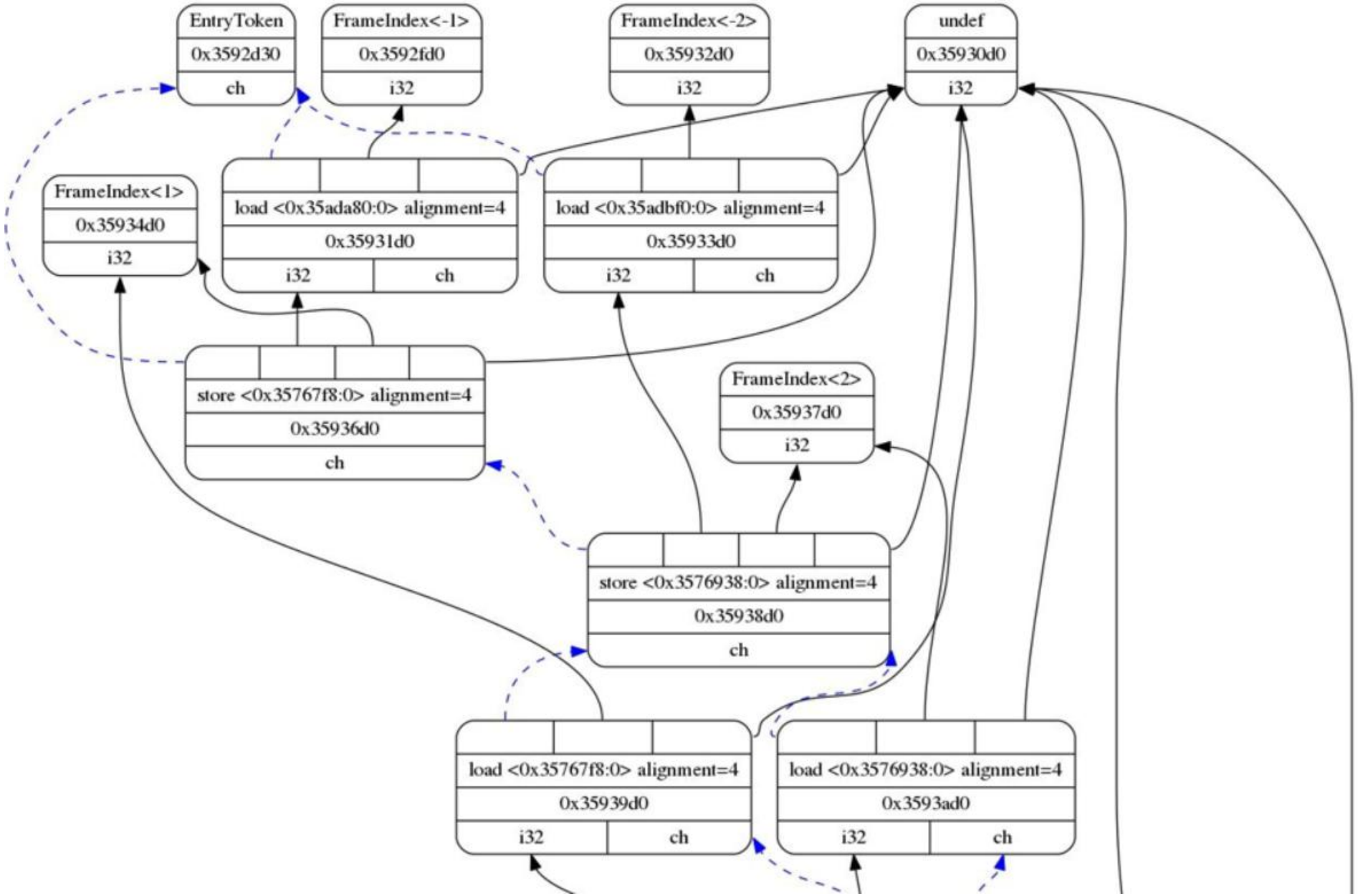
```
u@u-virtual-machine:~/tools/test$ cat add.ll
; ModuleID = 'add.cc'
source_filename = "add.cc"
target datalayout = "e-m:e-p:64:64-i64:64-i128:128-n64-S128"
target triple = "riscv64"

; Function Attrs: noinline nounwind optnone
define signext i32 @_Z3addii(i32 signext %a, i32 signext %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}

attributes #0 = { noinline nounwind optnone "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "frame-pointer"="none" "less-pre
cise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="
false" "no-trapping-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0 "}
```

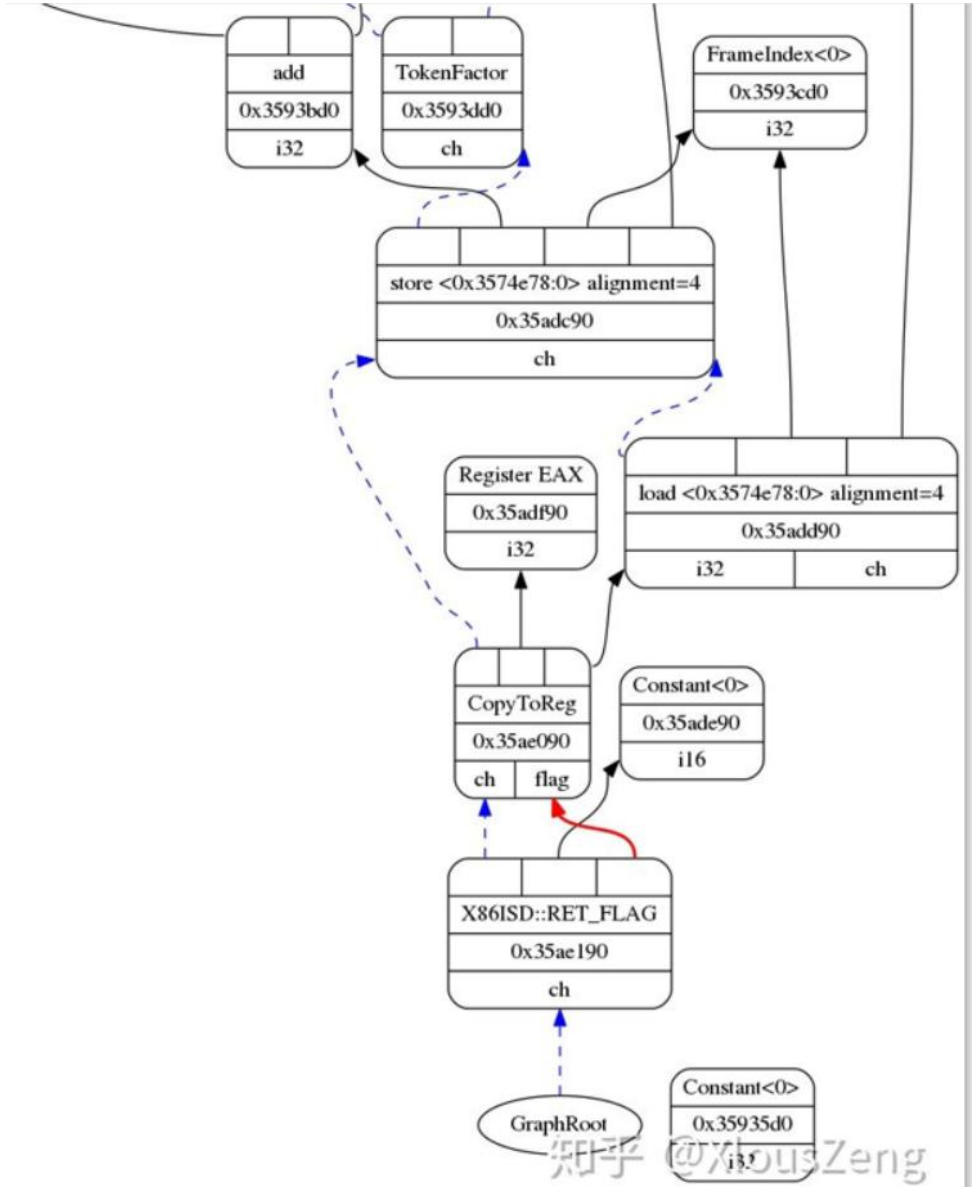
rvv-llvm中，用clang，target选择riscv64，生成上述add.cc的LLVM IR 中间表示add.ll

然后调用SelectionDAGBuild类以基本块为单位构造一个Selection DAG图，所有的alloca会被分配一个函数栈索引FrameIndex，然后构造一个对应的FrameIndexSDNode结点。store和load指令分别转换为ISD::store和ISD::load，add操作转换为ISD::add，ret会转换为RISCVISD::RET_FLAG。

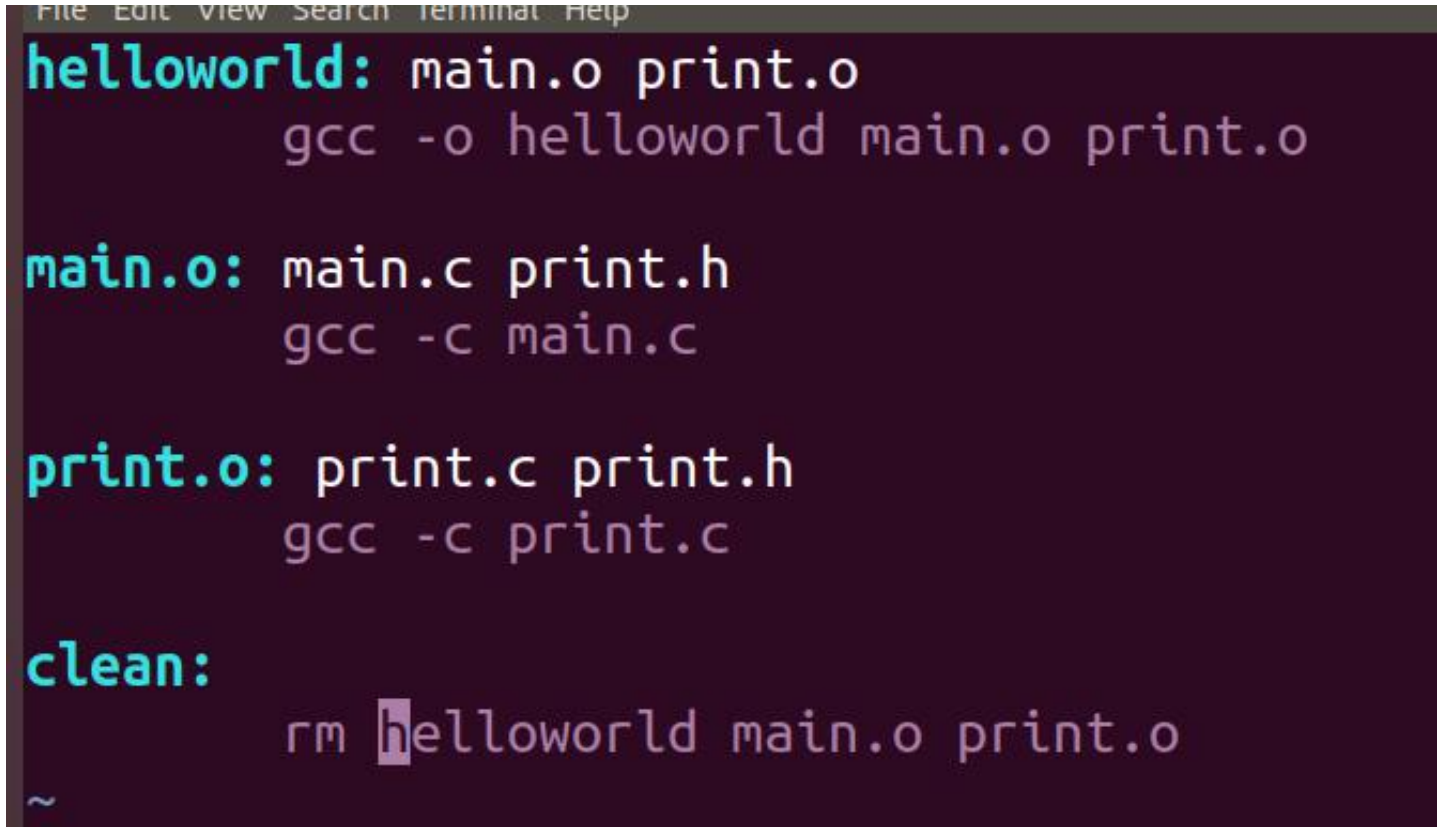
SelectionDAG变换

clang-cc -S -triple=i386-linux-gnu add.c -view-dag-combine1-dags 查看输出的DAG


EntryToken
0x3592d30
ch
FrameIndex<-1>
0x3592fd0
i32
FrameIndex<-2>
0x35932d0
i32
undef
0x35930d0
i32
FrameIndex<1>
0x35934d0
i32
load <0x35ada80:0> alignment=4
0x35931d0
i32
ch
load <0x35adbf0:0> alignment=4
0x35933d0
i32
ch
store <0x35767f8:0> alignment=4
0x35936d0
ch
FrameIndex<2>
0x35937d0
i32
store <0x3576938:0> alignment=4
0x35938d0
ch
load <0x35767f8:0> alignment=4
0x35939d0
i32
ch
load <0x3576938:0> alignment=4
0x3593ad0
i32
ch


add
0x3593bd0
i32
TokenFactor
0x3593dd0
ch
FrameIndex<0>
0x3593cd0
i32
store <0x3574e78:0> alignment=4
0x35adc90
ch
Register EAX
0x35adf90
i32
load <0x3574e78:0> alignment=4
0x35add90
i32
ch
CopyToReg
0x35ae090
ch
flag
Constant<0>
0x35ade90
i16
X86ISD::RET_FLAG
0x35ae190
ch
GraphRoot
Constant<0>
0x35935d0
i32

```
u@u-virtual-machine:~/tools/test/linking-test$ cat main.c
#include "print.h"

int main(void){

    printhello();

    return 0;

}
u@u-virtual-machine:~/tools/test/linking-test$ cat print.h
#include<stdio.h>

void printhello();
u@u-virtual-machine:~/tools/test/linking-test$ cat print.c
#include"print.h"

void printhello(){

    printf("Hello, world\n");

}
```



1 Makefile:5: *** missing separator. Stop. (solved)

2 make: *** No rule to make target ' ', needed by 'main.o'. Stop.

• 03 LLVM测试框架

1. LLVM测试框架简介

在LLVM编译器后端开发过程中，针对特定平台必然要定义特定指令集及其指令格式，并对后端流程的各个阶段或pass做相应修改。根据需求编写测试用例的测试驱动开发（Test-Driven Development，简称TDD）是推动开发进行的有效方法，可以在出现问题时实现可回溯，有助于编写简洁可用和高质量的代码。LLVM测试框架是在LLVM编译器后端开发过程中实现测试驱动开发的有效手段，有很高的灵活性和健壮性，可保证加速开发过程稳步进行。

本文涵盖LLVM测试框架，需要用到的工具，以及如何增加和运行测试用例。LLVM测试框架主要包括两类：回归测试和整体程序测试。回归测试代码位于LLVM代码库llvm/test路径下。这些用例应在每次提交前运行通过。整体程序测试也称为LLVM测试套件（LLVM test suite）。

回归测试

回归测试是测试LLVM特定功能或触发LLVM特定bug的一小段代码。这些代码用什么语言依赖于被测试LLVM部分用什么语言。这些测试用例由LLVM lit测试工具驱动，用例代码位于llvm/test。

测试套件

测试套件包含完整程序，程序代码通常用高级语言如C/C++，可以编译链接进某个可执行程序。这些程序有用户特定编译器编译，然后执行捕获程序输出和时序信息。这些输出和参考输出比较以确保程序编译正确。

除了编译和执行程序，完整程序测试还可作为LLVM性能基准，包括衡量程序生成效率和LLVM编译速度、优化和代码生成。

调试信息测试

测试套件包括检查调试信息质量的用例。这些用例用C或LLVM汇编语言写成。这些测试用例在某调试器下编译或运行。通过检查调试器输出可评估调试信息。更详细信息参见测试套件下的Readme.txt。

本文重点介绍回归测试。若要运行所有的LLVM回归测试用例，可使用check-llvm：

```
% make check-llvm
```

若要运行单个测试用例，或某个测试子集，可以使用llvm-lit脚本，llvm-lit是作为LLVM一部分编译生成。例如，若要运行AMDGPU的add.i16.ll测试用例，可以执行如下命令：

```
user@user:~/project/llvm/llvm/build$ bin/llvm-lit ../test/CodeGen/AMDGPU/add.i16.ll
```

输出如下：

```
-- Testing: 1 tests, 1 threads --
PASS: LLVM :: CodeGen/AMDGPU/add.i16.ll (1 of 1)
Testing Time: 0.64s
Expected Passes : 1
```

也可以运行整个AMDGPU的CodeGen测试用例：

```
user@user:~/project/llvm/llvm/build$ bin/llvm-lit ../test/CodeGen/AMDGPU
```

结果就可以得出

2. 如何产生新测试用例

LLVM回归测试框架尽管简单，但仍需要设置一些信息。这些配置信息写在特定文件中。为了使回归测试工作，每个测试目录下都有一个lit.local.cfg文件. llvm-lit会根据这个文件决定如何运行测试。AMDGPU测试目录下的lit.local.cfg文件内容如下。如果用户要增加一个新的测试文件夹，只需要将lit.local.cfg文件拷贝到新文件夹下。

```
if not 'AMDGPU' in config.root.targets:
config.unsupported = True
```

下面以实例说明测试用例编写方法。这个例子非常简单，函数体中没有具体功能，直接返回。指定检查前缀为”FUNC”。

```
; RUN: llc -mcpu=gfx* -march=<target> --verify-
machineinstrs < %s | FileCheck -enable-var-scope -check-
prefixes=FUNC %s
; FUNC: {{^}}kernel0:
; FUNC: %bb.0:
; FUNC-NEXT: s_endpgm
define <target>_kernel void @kernel0(i32* %out, i32 %in)
align 256 {
entry:
ret void
}
```

每个测试文件的第一行都以“RUN:”开始，称为RUN指令行，这告诉lit如何运行这个文件。RUN指令行如果没有“RUN “，lit将报错。“RUN:”后的部分是lit运行测试要执行的脚本。RUN指令行的语法类似shell pipeline语法，包括I/O重定向和变量替换。RUN指令行由lit解释。每一行RUN指令行都单独执行，除非最后是“\”字符。lit会替换变量并安排pipeline执行。如果pipeline中的任何过程失败，整个用例就算失败。应该尽量使RUN指令行保持简单，只用来跑工具产生文本输出供检查。

RUN指令行中的“--verify-machineinstrs”用于启动机器码验证器（Machine Code Verifier）对指令的操作数数量、物理或虚拟寄存器操作数的寄存器类、寄存器生存期进行检查。

RUN指令行中的“%s”用于替换测试用例源码文件路径，即将如下命令行中的路径“../../test/MC /ELF/foo_test.s”作为输入传给LLVM工具：

bin/llvm-lit ../../test/MC/ELF/foo_test.s

除了%s，RUN指令行中还执行以下替换：

%%：用单个%替换，可以转义其它替换；

%S（大写S）：测试用例目录路径，例如：/home/user/llvm/test/MC/ELF。

%t：本测试用例中用到的临时文件名的文件路径。这个文件名不会和其它测试用例冲突。如果需要多个临时文件，可以加上%t，可用于表示重定向输出目的文件。

更多指令替换请参见文献[1]。

LLVM测试框架中检查输出的推荐方法是使用FileCheck工具看测试是否通过。RUN指令行中的“-check-prefixes=FUNC”是指定FileCheck工具的匹配模式前缀选项“-check-prefixes”，默认为“CHECK:”。此处通过“-check-prefixes=FUNC”可以选择自定义的前缀选项“FUNC”。FileCheck工具将在下文介绍。

用llc编译上例的.ll文件，产生的相应汇编代码如下：

```
kernel0: ; @kernel0

; %bb.0: ; %entry

s_endpgm
```

运行llvm-lit的结果是“FUNC:”和“FUNC-NEXT:”指令都可以通过，因为指令检查的内容与汇编代码相符。

但如果在IR函数中只增加一条指令“store volatile i32 0, i32* %out”，代码如下：

```
define <target>_kernel void @kernel0(i32* %out, i32 %in)
align 256 {
entry:
store volatile i32 0, i32* %out
ret void
}
```

由llc产生的相应汇编代码如下：

```
kernel0: ; @kernel0
; %bb.0: ; %entry
s_load_dwordx2 s[0:1], s[0:1], 0x9
v_mov_b32_e32 v2, 0
s_waitcnt lgkmcnt(0)
v_mov_b32_e32 v0, s0
v_mov_b32_e32 v1, s1
flat_store_dword v[0:1], v2
s_endpgm
```

运行llvm-lit结果将报错，因为以上“FUNC:”和“FUNC-NEXT:”指令不能通过。原因是“%bb.0: ”和“s_endpgm”之间增加了多条指令，“s_endpgm”已经不再是紧跟在“%bb.0: ”之后。

但是如果增加如下检查指令“; FUNC-NEXT: s_waitcnt lgkmcnt(0)”：

```
; RUN: llc -mcpu=gfx700 -march=<target> --verify-machineinstrs < %s | FileCheck -enable-var-scope -check-prefixes=FUNC %s
; FUNC: {{^}}kernel0:
; FUNC: v_mov_b32_e32 v{{[0-9]+}}, 0
; FUNC-NEXT: s_waitcnt lgkmcnt(0)
; FUNC: s_endpgm
```

再运行llvm-lit，这个case就可以通过了，因为从汇编代码中可以看到，指令s_waitcnt的确是紧接在指令 v_mov_b32_e32之后。

在实际编写测试用例时，LLVM开发文档文档中建议将相关的测试用例放到一个文件中，而不是每个测试都有一个单独的文件。但这带来的问题是出错时不易定位文件中的哪个用例出错导致整个文件测试失败。所以，文件的测试用例粒度需要根据实际需要把握。另外，在根据需求编写新用例前，应检查已有文件是否覆盖了新需求的特性。应首先考虑在已有测试用例文件中增加新代码而不是生成一个新文件。

3. FileCheck工具用法

FileCheck工具一般用于回归测试，由RUN指令行调用。FileCheck命令格式如下：

FileCheck match-filename [–check-prefix=XXX] [–strict-whitespace]

FileCheck读入两个文件，一个从标准输入，一个从命令行，并使用其中一个文件验证另一个文件。这个方式对test suite特别有用，test suite就是想验证某些工具的输出，例如llc。这类似于grep的功能，但对匹配一个文件中特定顺序的多个不同输入做了优化。如果FileCheck验证文件与期望内容匹配，返回0；否则，产生错误，返回非0值。

在此输入您的封面副标题

在此输入您的封面副标题

在此输入您的封面副标题

在此输入您的封面副标题

在此输入您的封面副标题

在此输入您的封面副标题

• 03 参考资料

```
; RUN: llc -mcpu=gfx* -march=<target> --verify-
machineinstrs < %s | FileCheck -enable-var-scope -check-
prefixes=FUNC %s
; FUNC: {{^}}kernel0:
; FUNC: %bb.0:
; FUNC-NEXT: s_endpgm
define <target>_kernel void @kernel0(i32* %out, i32 %in)
align 256 {
entry:
ret void
}
```

- 04 问题

llv

谢 谢

欢迎交流合作

2019/02/25