RISCV GNU Toolchain和LLVM 回归测试的介绍

PLCT实验室

xiaoou@iscas.ac.cn

2021.6.27

目录

回归测试

- 测试用例主要是编译器开发过程中出错小程序的积累或者是针对特定功能添加的小代码片段
- 用于验证编译器功能的正确性
- 用于验证新的commit没有引入新的错误或导致其他代码产生错误

RISCV GNU Toolchain构建和回测

1. 安装必要软件

```
$ sudo apt-get install git build-essential tcl expect flex texinfo bison \
libpixman-1-dev libglib2.0-dev pkg-config zlib1g-dev ninja-build
```

2. 下载`RISCV GNU Toolchain`源码

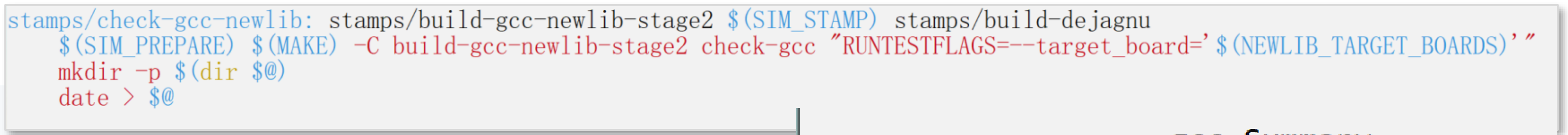
```
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

如果下载太慢，可以加上`--depth=1`只下载当前最新的提交

3. configure & make

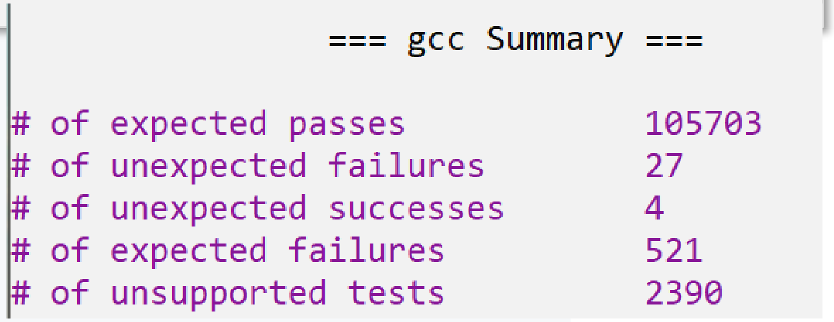
```
$ cd riscv-gnu-toolchain
$ mkdir build && cd build
$ ./configure --prefix=$RISCV/rv64/ --with-arch=rv64gc --with-abi=lp64d --with-multilib-generator="rv64gc-lp64d--"
$ make $nproc
```

4. `make check`运行回归测试



```
stamps/check-gcc-newlib: stamps/build-gcc-newlib-stage2 $(SIM_STAMP) stamps/build-dejagnu
	$(SIM_PREPARE) $(MAKE) -C build-gcc-newlib-stage2 check-gcc "RUNTESTFLAGS=--target_board='$(NEWLIB_TARGET_BOARDS)'"
	mkdir -p $(dir $@)
	date > $@
```

```
## 'make check' depends on 'make check-gcc-newlib', so default runs gcc's testsuite.
##  check targets: gcc, dhrystone, binutils, gdb
##  check-gcc-newlib/check-gcc-linux
##  check-dhrystone-newlib/check-dhrystone-linux
##  check-binutils-newlib/check-binutils-linux
##  check-gdb-newlib/check-gdb-newlib
$ make check
```



```
                === gcc Summary ===

# of expected passes            105703
# of unexpected failures        27
# of unexpected successes       4
# of expected failures          521
# of unsupported tests          2390
```

LLVM构建和回测

1. 安装必要软件

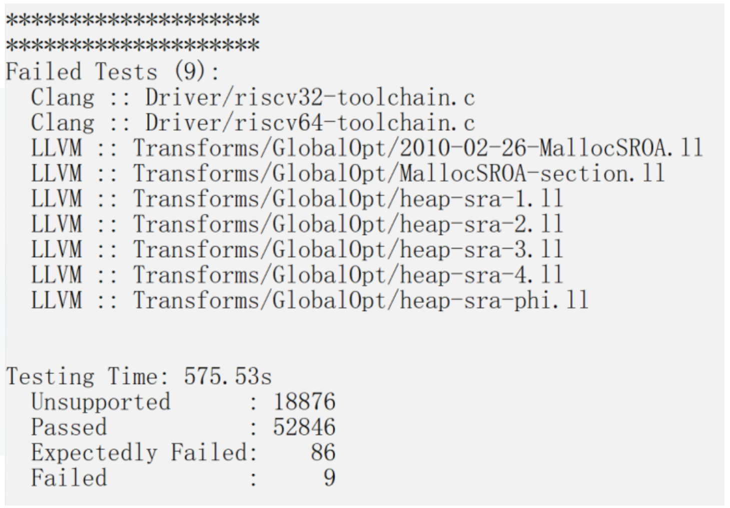
```
apt-get update apt-get install cmake ninja-build
```

2. 下载llvm源码

```
git clone --recursive https://github.com/llvm/llvm-project.git
```

3. configure & make

```
$ cd llvm-project
$ mkdir build && cd build
$ cmake -DLLVM_TARGETS_TO_BUILD="X86;RISCV" -DLLVM_ENABLE_PROJECTS="clang" -G Unix Makefiles ../llvm
$ make -j $nproc

## or you can use ninja to build:
## cmake -DLLVM_PARALLEL_LINK_JOBS=3 -DLLVM_TARGETS_TO_BUILD="X86;RISCV" -DLLVM_ENABLE_PROJECTS="clang"
## ninja
```



```
********************
********************
Failed Tests (9):
  Clang :: Driver/riscv32-toolchain.c
  Clang :: Driver/riscv64-toolchain.c
  LLVM :: Transforms/GlobalOpt/2010-02-26-MallocSROA.ll
  LLVM :: Transforms/GlobalOpt/MallocSROA-section.ll
  LLVM :: Transforms/GlobalOpt/heap-sra-1.ll
  LLVM :: Transforms/GlobalOpt/heap-sra-2.ll
  LLVM :: Transforms/GlobalOpt/heap-sra-3.ll
  LLVM :: Transforms/GlobalOpt/heap-sra-4.ll
  LLVM :: Transforms/GlobalOpt/heap-sra-phi.ll


Testing Time: 575.53s
  Unsupported      : 18876
  Passed           : 52846
  Expectedly Failed:    86
  Failed           :     9
```

在编译过程中，ld程序比较消耗内存，可以使用 `-DLLVM_PARALLEL_LINK_JOBS` 来限制并行的ld数量。

4. `make check` 运行回归测试

```
$ make check

## if you use ninja build, you also use 'ninja check'
```

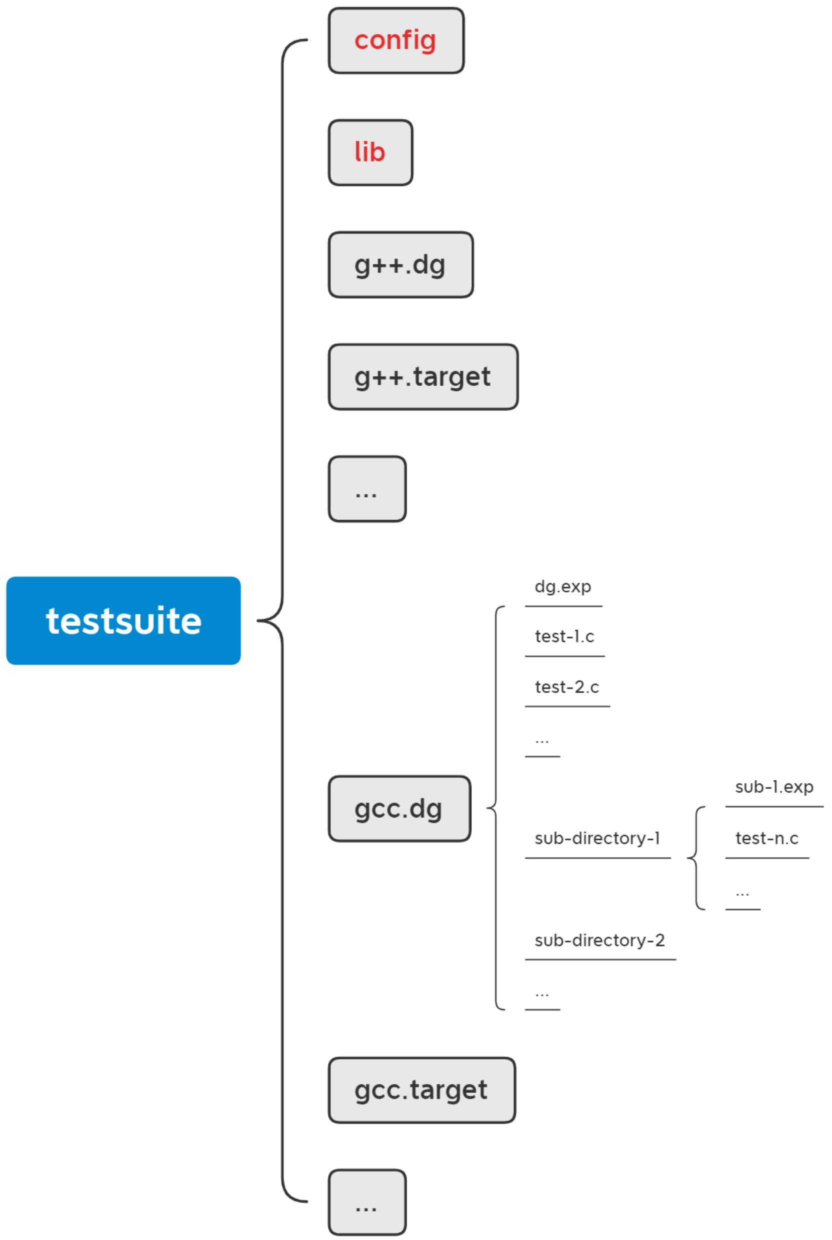
Run all unit tests	`$ make check-llvm-unit`
Run all regression tests	`$ make check-llvm`
Run regression individual test	`$ llvm-lit ~/llvm/test/Integer/BitPacked.ll`
Run regression subsets	`$ llvm-lit ~/llvm/test/CodeGen/ARM`
Run LLVM and Clang	`$ make check-all`

目录

GNU回归测试的目录结构

riscv-gnu-toolchain/riscv-gcc/gcc/testsuite （以gcc源码为例）：

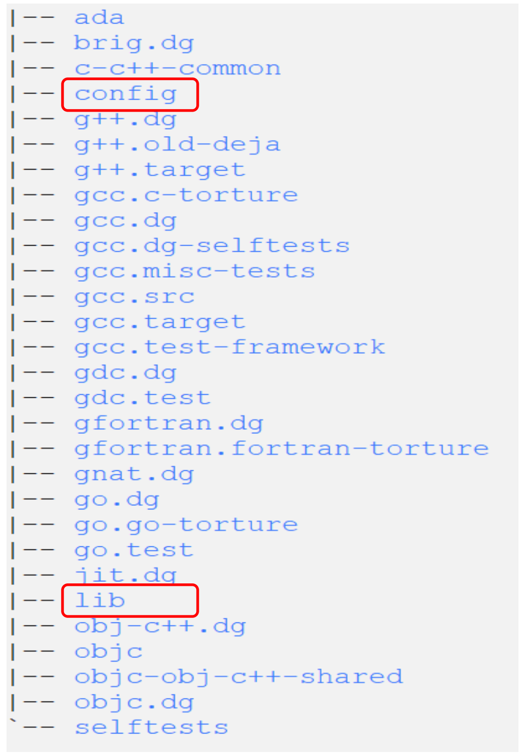
- config：该目录包含一个`default.exp`，用于设定默认的配置信息。
- lib：在lib中存放的也是配置文件，针对不同的工具或语言。
- 测试目录：
  - 目录名：toolname[.type][.tests]
  - [.type]将同一类测试分成了多个文件夹，例如：gcc.dg, gcc.target
  - 测试目录 = .exp + testcases + 子目录
  - 每个测试目录必须至少有一个.exp文件来作为driver来运行所有的测试用例。
  - DejaGNU测试框架会按字母顺序遍历测试目录并执行.exp文件。



GNU回归测试的目录结构

- gcc/testsuite中的测试目录和描述 （部分）

测试目录	描述
ada	the Ada language testsuites.
brig.dg	BRIG(HSALL) frontend test case.
c-c++-common	for both c and c++ test
g++.dg	All **new G++ tests** should be placed here.
g++.target/gcc.target	test for different processor **architecture**.
gcc.dg	**correctness tests** for various compiler features should go here if possible.
gcc.c-torture	contains code fragments broken easily historically. Run with **multiple optimization options**, so tests for features which only break at some optimization levels belong here.



```
|-- ada
|-- brig.dg
|-- c-c++-common
|-- config
|-- g++.dg
|-- g++.old-deja
|-- g++.target
|-- gcc.c-torture
|-- gcc.dg
|-- gcc.dg-selftests
|-- gcc.misc-tests
|-- gcc.src
|-- gcc.target
|-- gcc.test-framework
|-- gdc.dg
|-- gdc.test
|-- gfortran.dg
|-- gfortran.fortran-torture
|-- gnat.dg
|-- go.dg
|-- go.go-torture
|-- go.test
|-- jit.dg
|-- lib
|-- obj-c++.dg
|-- objc
|-- objc-obj-c++-shared
|-- objc.dg
`-- selftests
```

LLVM回归测试的目录结构

- llvm/test 目录结构
    - config文件：lit.site.cfg （每个目录需要有一个）
    - 子目录：

子目录	描述
Analysis	checks Analysis passes.
Assembler	checks Assembly reader/writer functionality.
Bitcode	checks Bitcode reader/writer functionality.
CodeGen	checks code generation and each target.
Features	checks various features of the LLVM language.
Linker	tests bitcode linking.
Transforms	tests each of the scalar, IPO, and utility transforms to ensure they make the right transformations.
Verifier	tests the IR verifier.

目录

- 回归测试的流程和命令
- 回归测试的目录结构
- 测试用例的写法

添加一个GNU的测试用例

- 两种方法
    - 在已有的testsuite子目录下添加测试用例，例如在`gcc/testsuite/gcc.dg/`下添加；
    - 创建新的测试子目录，并在其中添加`.exp`文件和测试用例。

（这里的测试用例指testsuite子目录下存放的`.c .s`等代码片段，里面包含了DejaGnu定义的指示符，DejaGnu的测试框架会根据这些指示符来执行和判断）

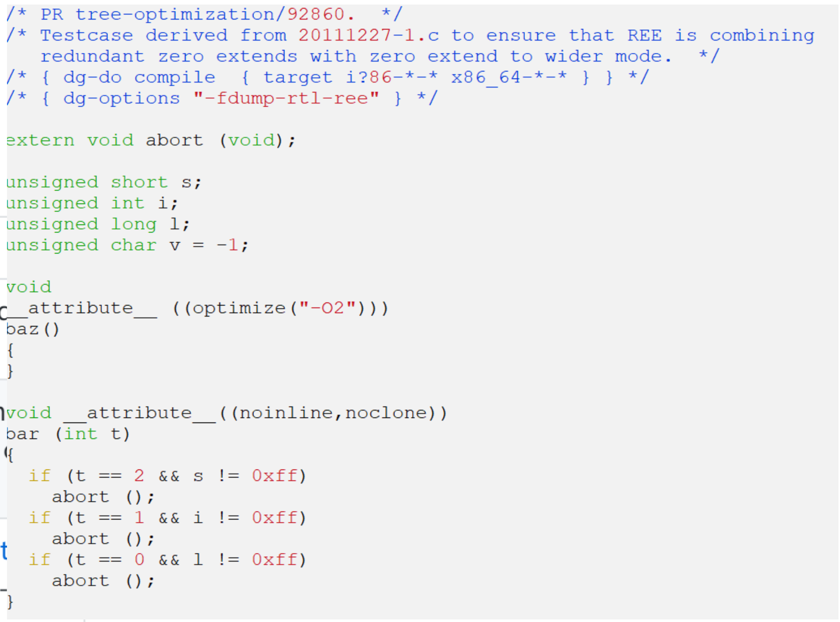
其中，每个子目录下.exp文件作为该子目录的driver，来定义怎样执行该目录下的测试用例，一般的格式如下：

```
load_lib ${tool}-dg.exp
dg-init
dg-runtest [lsort [glob —nocomplain $srcdir/$subdir/foo*]] ...
dg-finish
```

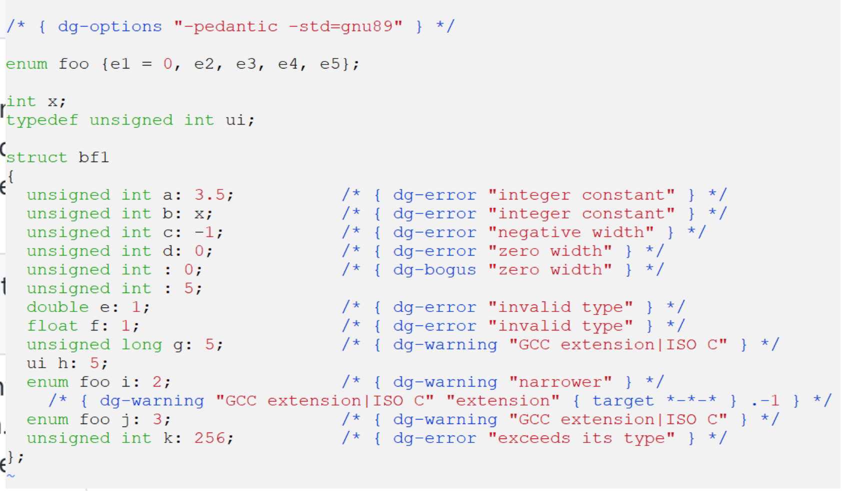
添加一个GNU的测试用例

- 测试用例中的指示符

header 1	header 2
`{ dg-do do-what-keyword [{ target/xfail selector }]}`	**Specify how to build the test** : do-what-keyword specifies how the test is compilec d whether it is executed. It is one of: preprocess, compile, assemble, link, run.
`{ dg-options options [{ target selector }] }`	**Specify additional compiler options** : This DejaGnu directive provides a list of com r options, to be used if the target system matches selector, that replace the default ons used for this set of tests.
`{ dg-skip-if comment { selector } [{ include-opts } [{ exclude-opts }]] }`	**Skip a test for some targets** : Skip the test if all of the following conditions are met skip a test if either -O2 or -O3 is used with -g but not if -fpic is also present: /* { dg- -if "" { --* } { "-O2 -g" "-O3 -g" } { "-fpic" } } */
`{ dg-xfail-if comment { selector } [{ include-opts } [{ exclude-opts }]] }`	**Expect a test to fail for some targets**: Expect the test to fail if the conditions (which are the same as for dg-skip-if) are met. This does not affect the execute step.
`{ dg-error regexp [comment [{ target/xfail selector } [line] ]] }`	**Verify compiler messages** : This DejaGnu directive appears on a source line that is ected to get an error message, or else specifies the source line associated with the r age. If there is no message for that line or if the text of that message is not matchec y regexp then the check fails and comment is included in the FAIL message. The che does not look for the string 'error' unless it is part of regexp.
`{ dg-output regexp [{ target/xfail selector }] }`	**Verify output of the test executable** : This DejaGnu directive compares regexp to t ombined output that the test executable writes to stdout and stderr.
`{ dg-final { local-directive } }`	**Add checks at the end of a test** :This DejaGnu directive is placed within a commen ywhere in the source file and is processed after the test has been compiled and run. ltiple 'dg-final' commands are processed in the order in which they appear in the urce file. See Final Actions, for a list of directives that can be used within dg-final.



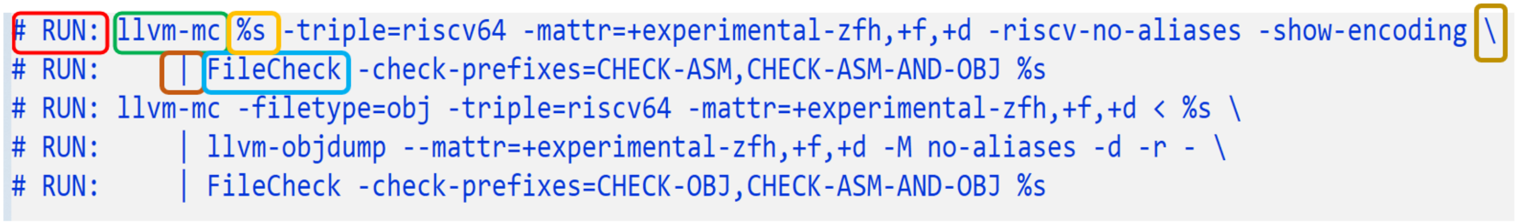
```
/* PR tree-optimization/92860.  */
/* Testcase derived from 20111227-1.c to ensure that REE is combining
   redundant zero extends with zero extend to wider mode.  */
/* { dg-do compile  { target i?86-*-* x86_64-*-* } } */
/* { dg-options "-fdump-rtl-ree" } */

extern void abort (void);

unsigned short s;
unsigned int i;
unsigned long l;
unsigned char v = -1;

void
__attribute__ ((optimize("-O2")))
baz()
{
}

void __attribute__((noinline,noclone))
bar (int t)
{
  if (t == 2 && s != 0xff)
    abort ();
  if (t == 1 && i != 0xff)
    abort ();
  if (t == 0 && l != 0xff)
    abort ();
}
```



```
/* { dg-options "-pedantic -std=gnu89" } */

enum foo {e1 = 0, e2, e3, e4, e5};

int x;
typedef unsigned int ui;

struct bf1
{
  unsigned int a: 3.5;          /* { dg-error "integer constant" } */
  unsigned int b: x;            /* { dg-error "integer constant" } */
  unsigned int c: -1;           /* { dg-error "negative width" } */
  unsigned int d: 0;            /* { dg-error "zero width" } */
  unsigned int : 0;             /* { dg-bogus "zero width" } */
  unsigned int : 5;
  double e: 1;                  /* { dg-error "invalid type" } */
  float f: 1;                   /* { dg-error "invalid type" } */
  unsigned long g: 5;           /* { dg-warning "GCC extension|ISO C" } */
  ui h: 5;
  enum foo i: 2;                /* { dg-warning "narrower" } */
    /* { dg-warning "GCC extension|ISO C" "extension" { target *-*-* } .-1 } */
  enum foo j: 3;                /* { dg-warning "GCC extension|ISO C" } */
  unsigned int k: 256;          /* { dg-error "exceeds its type" } */
};
~
```

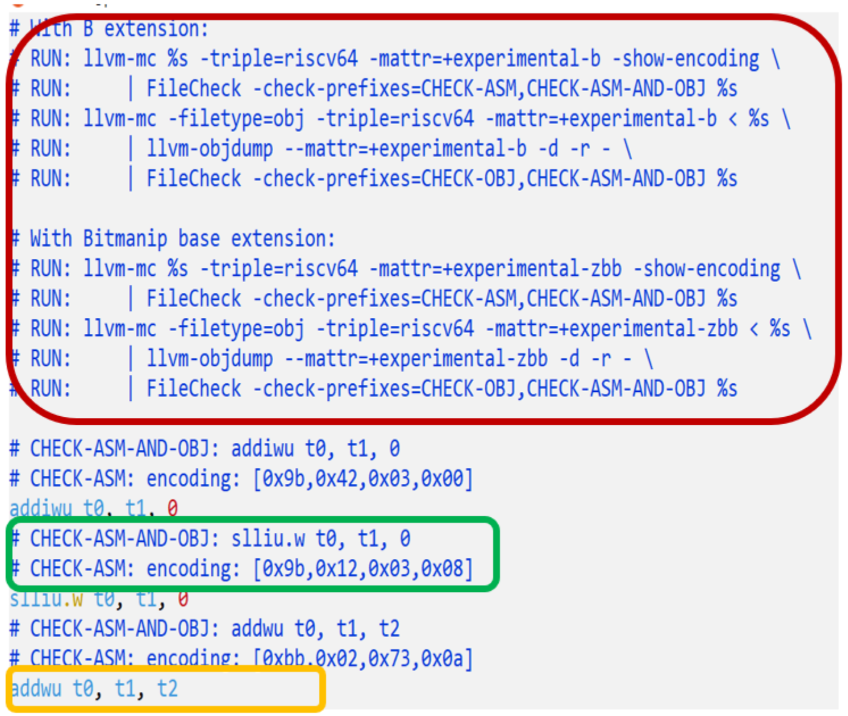
添加一个LLVM的测试用例

- 添加一个新的测试用例
  - 新测试子目录下需要创建lit.local.cfg
  - RUN lines



```
# RUN: llvm-mc %s -triple=riscv64 -mattr=+experimental-zfh,+f,+d -riscv-no-aliases -show-encoding \
# RUN:     | FileCheck -check-prefixes=CHECK-ASM,CHECK-ASM-AND-OBJ %s
# RUN: llvm-mc -filetype=obj -triple=riscv64 -mattr=+experimental-zfh,+f,+d < %s \
# RUN:     | llvm-objdump --mattr=+experimental-zfh,+f,+d -M no-aliases -d -r - \
# RUN:     | FileCheck -check-prefixes=CHECK-OBJ,CHECK-ASM-AND-OBJ %s
```

Macro	Substitution
%s	source path (path to the file currently being run)
%S	source dir (directory of the file currently being run)
%p	same as %S
%{pathsep}	path separator
%t	temporary file name unique to the test
%basename_t	The last path component of %t but without the `.tmp` extension
%T	parent directory of %t (not unique, deprecated, do not use)
%%	%



```
# With B extension:
# RUN: llvm-mc %s -triple=riscv64 -mattr=+experimental-b -show-encoding \
# RUN:     | FileCheck -check-prefixes=CHECK-ASM,CHECK-ASM-AND-OBJ %s
# RUN: llvm-mc -filetype=obj -triple=riscv64 -mattr=+experimental-b < %s \
# RUN:     | llvm-objdump --mattr=+experimental-b -d -r - \
# RUN:     | FileCheck -check-prefixes=CHECK-OBJ,CHECK-ASM-AND-OBJ %s

# With Bitmanip base extension:
# RUN: llvm-mc %s -triple=riscv64 -mattr=+experimental-zbb -show-encoding \
# RUN:     | FileCheck -check-prefixes=CHECK-ASM,CHECK-ASM-AND-OBJ %s
# RUN: llvm-mc -filetype=obj -triple=riscv64 -mattr=+experimental-zbb < %s \
# RUN:     | llvm-objdump --mattr=+experimental-zbb -d -r - \
# RUN:     | FileCheck -check-prefixes=CHECK-OBJ,CHECK-ASM-AND-OBJ %s

# CHECK-ASM-AND-OBJ: addiwu t0, t1, 0
# CHECK-ASM: encoding: [0x9b,0x42,0x03,0x00]
addiwu t0, t1, 0
# CHECK-ASM-AND-OBJ: slliu.w t0, t1, 0
# CHECK-ASM: encoding: [0x9b,0x12,0x03,0x08]
slliu.w t0, t1, 0
# CHECK-ASM-AND-OBJ: addwu t0, t1, t2
# CHECK-ASM: encoding: [0xbb,0x02,0x73,0x0a]
addwu t0, t1, t2
```

THE END

Thanks For Your Watching !