LLVM Testing Infrastructure Guide

PLCT Lab
Molly Chen
xiaoou@iscas.ac.cn

Content

1. LLVM Testing Infrastructure Organization
2. Quick Start
3. Regression Test Structure
4. Lit (LLVM Integrated Tester)

5. Practice：Add a testcase for zfh extension

(refer to https://www.llvm.org/docs/TestingGuide.html)

LLVM Testing Infrastructure Organization

Category	Location	Code	Target	Required before commit
Unit tests	llvm/unittests	Using *Google Test* And *Google Mock*	support library, generic data structure	Yes
Regression Tests	**llvm/test**	**Small pieces of code**	**Transformations and analysis on IR**	**Yes**
Test-suite (Nightly tests)	test-suite Subversion module (https://github.com/llvm/llvm-test-suite.git)	Whole programs	Program compiling and executing; Performance benchmarking	No
Debugging information tests	debuginfo-tests Subversion module	C based language, LLVM assembly language	Quality of debugging information	No

Table1 three major categories of tests and debugging information tests

Quick Start (1/2)

- Requirements
  - Need all of the software required to build LLVM, and Python2.7 or later.
- How to run test
  - First, build LLVM.

(Refer to http://llvm.org/docs/GettingStarted.html)

  - Commands for unit tests and regression tests

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`

Quick Start (2/2)

- How to run test (cont.)
  - Commands for unit tests and regression tests (cont.)

To get reasonable testing performance, build LLVM in release mode, i.e.

```
% cmake -DCMAKE_BUILD_TYPE="release" -DLLVM_ENABLE_ASSERTIONS=On
```

  - Commands for test-suite

(refer to https://www.llvm.org/docs/TestSuiteGuide.html)

  - Commands for Debugging Information tests

To run debugging information tests simply add the debuginfo-tests project to your LLVM_ENABLE_PROJECTS define on the cmake command-line.

Regression Test Structure (1/3)

- Driver: lit (llvm integrated tester, see lit section)
- Structure of llvm/test
  - Config file: lit.site.cfg (each dir mush have)
  - Sub directory

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.

```
|-- Analysis
|-- Assembler
|-- Bindings
|-- Bitcode
|-- BugPoint
|-- CodeGen
|-- DebugInfo
|-- Demangle
|-- Examples
|-- ExecutionEngine
|-- Feature
|-- FileCheck
|-- Instrumentation
|-- Integer
|-- JitListener
|-- LTO
|-- Linker
|-- MC
|-- MachineVerifier
|-- Object
|-- ObjectYAML
|-- Other
|-- Reduce
|-- SafepointIRVerifier
|-- Support
|-- SymbolRewriter
|-- TableGen
|-- ThinLTO
|-- Transforms
|-- Unit
|-- Verifier
|-- YAMLParser
`-- tools
```

Figure1 Sub-directory in llvm/test

Regression Test Structure (2/3)

- Writing new regression tests
  - General
    - Config file (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
```

figure2 RUN lines in a test file

(FileCheck tool refer to https://www.llvm.org/docs/CommandGuide/FileCheck.html)

    - Put related tests into a single file, or adding your code in existing file.

Regression Test Structure (3/3)

- Writing new regression tests (cont.)
  - General
  - Extra files
    - If small, put in the same file, use separator;
    - If large, put them in a subdirectory.
  - Fragile tests
    - To make tests robust, always use `opt … < %s` in the RUN line.
  - Platform-specific tests
    - Add specific triple, Test with specific FileCheck, Go in its own directory
  - Constraining test execution
    - REQUIRES, UNSUPPORTED, XFAIL
  - Substitutions (see in Lit Section)

Lit (LLVM Integrated Tester) (1/2)

- Usage

```
% llvm-lit ~/llvm/test/Integer/BitPacked.ll
```

- Test Status Results

PASS	succeeded.
FLAKYPASS	succeeded after being re-run more than once
XFAIL	failed, but expected
XPASS	succeeded, but expected to fail
FAIL	failed
UNRESOLVED	could not be determined
UNSUPPORTED	not supported
TIMEOUT	timed out, considered a failure

Lit (LLVM Integrated Tester) (2/2)

- Substitutions

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)
%%	%

Figure3 part of lit substitutions

- Test Run Output Format

```
PASS: A (1 of 4)
PASS: B (2 of 4)
FAIL: C (3 of 4)
******************** TEST 'C' FAILED ********************
Test 'C' failed as a result of exit code 1.
********************
PASS: D (4 of 4)
```

Figure4 output demo of lit

(From lit - https://www.llvm.org/docs/CommandGuide/lit.html)

Practice：Add a testcase for zfh extension (1/3)

- To verify Zfh extension,

add llvm/test/MC/RISCV/rv64zfh-valid.s

- How test case look like?

Refer to existing test case, like rv64zbb-valid.s

Test case contains:

- **Run lines**
- **CHECK tag**
  - **Conventional assembly**
  - **Object code**
- **Code to be tested (here should be assembly code)**

```
# 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
```

Figure5 text in rv64zbb-valid.s

Practice：Add a testcase for zfh extension (2/3)

- Code to test

Should cover special value like 0,

and boundary value like -2048/2047;

Cover all rounding modes: RNE, RTZ, RDN, RUP, RMM, DYN

```
flh f0, 12(a0)
flh f1, +4(ra)
flh f2, -2048(x13)
flh f3, %lo(2048)(s1)
flh f4, 2047(s2)
flh f5, 0(s3)
fsh f6, 2047(s4)
fsh f7, -2048(s5)
fsh f8, %lo(2048)(s6)
fsh f9, 999(s7)
fmadd.h f10, f11, f12, f13, dyn
fmsub.h f14, f15, f16, f17, dyn
fnmsub.h f18, f19, f20, f21, dyn
fnmadd.h f22, f23, f24, f25, dyn
```

Figure6 examples of riscv zfh assembly

- Conventional assembly

Figure7 Register name following calling convention

31 0	
x0 / zero	Hardwired zero
x1 / ra	Return address
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporary
x6 / t1	Temporary
x7 / t2	Temporary
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Function argument, return value
x11 / a1	Function argument, return value
x12 / a2	Function argument
x13 / a3	Function argument
x14 / a4	Function argument
x15 / a5	Function argument
x16 / a6	Function argument
x17 / a7	Function argument
x18 / s2	Saved register
x19 / s3	Saved register
x20 / s4	Saved register
x21 / s5	Saved register
x22 / s6	Saved register
x23 / s7	Saved register
x24 / s8	Saved register
x25 / s9	Saved register
x26 / s10	Saved register
x27 / s11	Saved register
x28 / t3	Temporary
x29 / t4	Temporary
x30 / t5	Temporary
x31 / t6	Temporary
32	

31 0	
pc	
32	

63 32	31 0	
	f0 / ft0	FP Temporary
	f1 / ft1	FP Temporary
	f2 / ft2	FP Temporary
	f3 / ft3	FP Temporary
	f4 / ft4	FP Temporary
	f5 / ft5	FP Temporary
	f6 / ft6	FP Temporary
	f7 / ft7	FP Temporary
	f8 / fs0	FP Saved register
	f9 / fs1	FP Saved register
	f10 / fa0	FP Function argument, return value
	f11 / fa1	FP Function argument, return value
	f12 / fa2	FP Function argument
	f13 / fa3	FP Function argument
	f14 / fa4	FP Function argument
	f15 / fa5	FP Function argument
	f16 / fa6	FP Function argument
	f17 / fa7	FP Function argument
	f18 / fs2	FP Saved register
	f19 / fs3	FP Saved register
	f20 / fs4	FP Saved register
	f21 / fs5	FP Saved register
	f22 / fs6	FP Saved register
	f23 / fs7	FP Saved register
	f24 / fs8	FP Saved register
	f25 / fs9	FP Saved register
	f26 / fs10	FP Saved register
	f27 / fs11	FP Saved register
	f28 / ft8	FP Temporary
	f29 / ft9	FP Temporary
	f30 / ft10	FP Temporary
	f31 / ft11	FP Temporary
32	32	

```
# CHECK-ASM-AND-OBJ: flh ft0, 12(a0)
# CHECK-ASM: encoding: [0x07,0x10,0xc5,0x00]
flh f0, 12(a0)
```

Practice：Add a testcase for zfh extension (3/3)

- Object code

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Invalid. Reserved for future use.*
110		*Invalid. Reserved for future use.*
111	DYN	In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *Invalid.*

Table2 Rounding mode encoding

- Little-endian to Big-endian

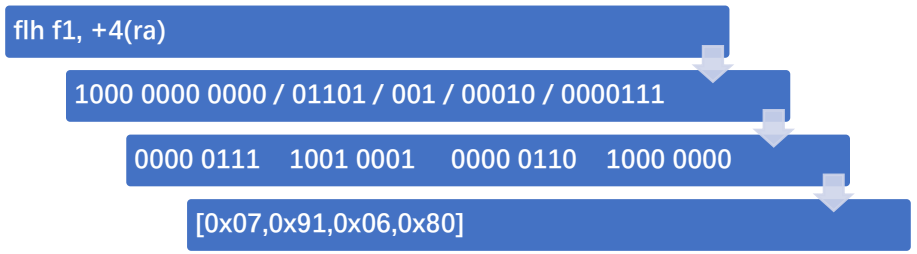

Figure9 example of transfer assembly to object

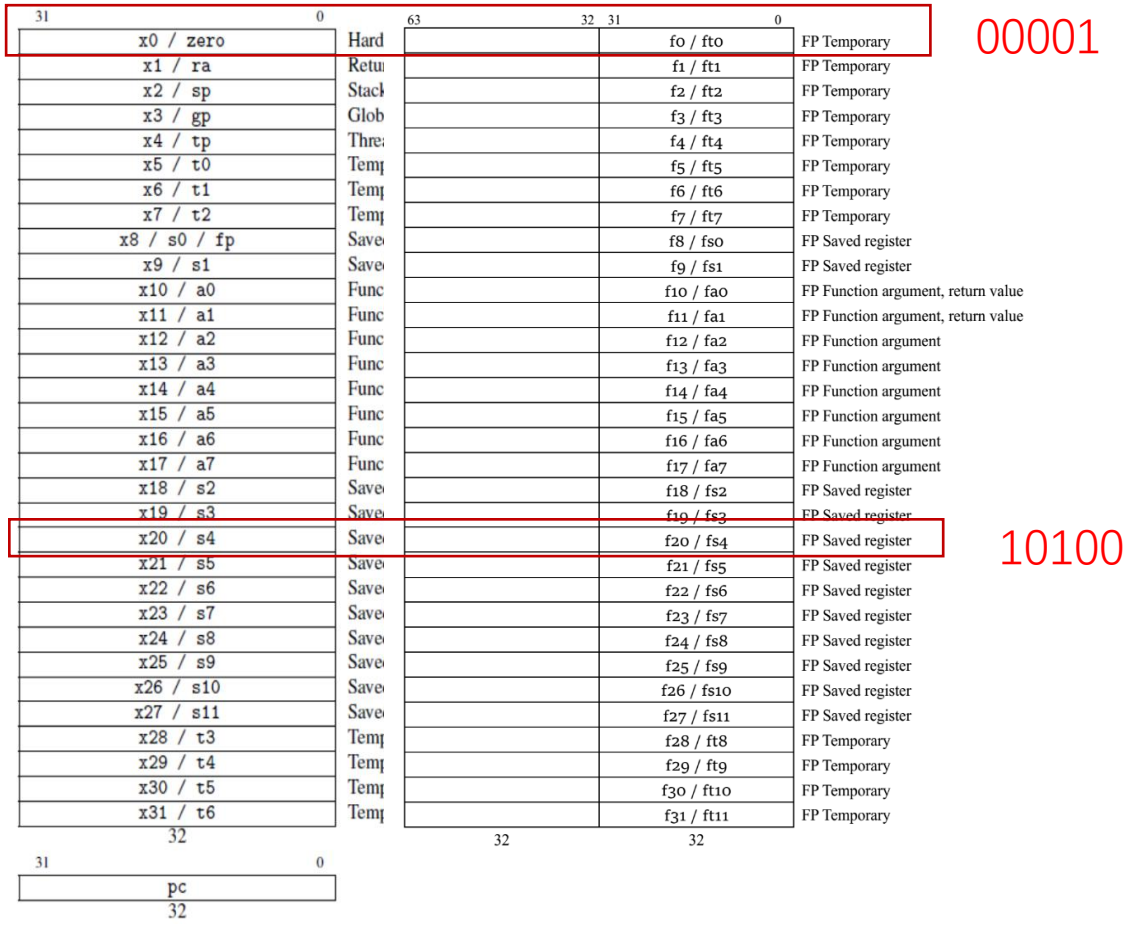

Figure8 register name and encoding

THE END

Thanks For Your Watching !