# RISC-Vエコシステムの使用方法

Aoba Fujino

2025-07-11

# Table of Contents

# RISC-V GNU Toolchain

RISC-V GNU Toolchainは GCCコンパイラを含むアセットです． とにもかくにも，コンパイラがないと始まりません. これがRISC-Vのクロスコンパイラになります. コンパイルする前に，使いたい拡張命令を選択できます. この研究室では基本的には `RV32IM` なので，次のコマンドになるかと思います. デフォルトだと `RV64GC` になっています. これはRV32IMのスーパーセットなので，`--march=rv32im --mabi=ilp32` というオプションでRV32IMようにコンパイルできます.

## ビルド方法

ビルドしていきます． [INFO]: 研究室のサーバーを利用する場合は `/cad/tool/bin/` にあるので，ビルドする必要はありません.

READMEの内容に従うだけです.

```
$ git clone https://github.com/riscv/riscv-gnu-toolchain
$ sudo apt-get install autoconf automake autotools-dev curl python3 python3-pip
python3-tomli libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex
texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git
cmake libglib2.0-dev libslirp-dev
```

configureで使いたいRISC-Vのアーキテクチャ とABIを選択します． 吉瀬研究室では基本的にRV32IMで十分です． たまに，LINUXを動作させるようなプロセッサを使用する場合は，RV32IMAを使用します． 64ビットはおそらく作っていません． 32ビットなので，ABIはilp32かilp32f, ilp32dから選ぶことになります． ilp32fはF拡張をサポートしている場合，ilp32dはD拡張をサポートしている場合に選択します． 吉瀬研究室のプロセッサには浮動小数演算ユニットはついていないので，ABIはilp32を使用します.

以上を踏まえて，ここではrv32im, ilp32でビルドします.

```
$ ./configure --prefix=/home/fujino/rv32im --with-arch=rv32im --with-abi=ilp32
$ make -j $(nproc)
```

だいたい，5分から10分ぐらいで終わります.

たまにjオプションを使うとエラーを見落とすことがあります． うまくいってそうなのに，binにg++がないとかgccはあるけどライブラリが見つからないの問題があったら， 失敗しています． シングルスレッドで実行してみてください（めちゃくちゃ時間かかります）.

# RISC-V Tests

RISC-V Testsは RISC-Vの検証用のテストベンチ集です. 自分の作ったプロセッサがRISC-Vの仕様を満たしているかで使います. めちゃくちゃ重要です.

## How to USE

使ってみましょう． クローンからです.

```
$ git clone https://github.com/riscv-software-src/riscv-tests
$ cd riscv-tests
$ git submodule update --init --recursive
$ autoconf
$ ./configure --with-xlen=32
```

単純なRV32IMで必要なテストは，rv32uiとrv32umです． ですが，ここではrv32uiのみのテストを想定します． riscv-tests/isa/Makefileで次を変更します.

```
  42    default: rv32ui
```

rv32uiにも，特権モードと仮想アドレスモードのテストがあります． 通常仮想アドレスモードは使わないので，次の記述を削除します.

```
  74    $$($(1)_v_tests): $(1)-v-%: $(1)/%.S
  75        $$(RISCV_GCC) $(2) $$(RISCV_GCC_OPTS) -DENTROPY=0x$$(shell echo
\$$@ | md5sum | cut -c 1-7) -std=gnu99 -O2 -I$(src_dir)/../env/v
-I$(src_dir)/macros/scalar -T$(src_dir)/../env/v/link.ld
$(src_dir)/../env/v/entry.S $(src_dir)/../env/v/*.c $$< -o $$@
  76    $(1)_tests += $$($(1)_v_tests)
```

また，ハードウェア固有の環境に合わせるためにriscv-tests/env/pのファイルを変更します． まず，リンカです． テンプレートはこんな感じです． 今回はdmemとimemが統合されたメモリを考えます.

```
OUTPUT_ARCH("riscv")
ENTRY(_start)

MEMORY {
    ram : ORIGIN = 0x00000000, LENGTH = 0x00001000
}

SECTIONS
{
    .text : {
        *(.text.init)
        *(.text.*)
```

```
        *(.text)
    } > ram

    .data : {
        *(.data.*)
        *(.sdata*)
        *(.data)
    } > ram

    .rodata : {
        *(.rodata.*)
        *(.srodata.*)
        *(.rodata)
    } > ram

    .bss : {
        *(.bss.*)
        *(.sbss)
        *(.bss)
        *(COMMON)
        _end = .;
    } > ram

    /DISCARD/ : {
        *(.debug*)
    }
}
```

riscv-tests/env/p/riscv_test.h に初期化ルーチンを記述します.

```
  183    #define RVTEST_CODE_BEGIN
\
```

から始まる部分が初期化ルーチンです. CSR命令をサポートしていないと使えないような命令がたくさん使用されているので，アーキテクチャ向けに改良していきます. _start:というのが，先ほどのリンカで指定したエントリーポイントです. ここからごっそり消していきます.

特にreset_vectorというラベルが初期化をしている部分です. いろいろ書いてありますが，XREGの初期化以外不要です. 全部削除します.

これで十分です.

```
#define RVTEST_CODE_BEGIN                                               \
        .section .text.init;                                            \
        .globl _start;                                                  \
_start:                                                                 \
        INIT_XREG;
```

```
#define RVTEST_PASS                                                     \
```

と

```
#define RVTEST_FAIL                                                     \
```

でテストに通った時と落ちた時の処理を定義します． やはりここでもfenceやecallが使用されているので，変更していきます． とりあえず．`0x40008000`に777を書き込んだらPASS,0だったらFAILにしましょう． こんな感じに変更できます．

```
#define RVTEST_PASS                                                     \
        lui t0, 0x40008;                                                  \
        li t1, 777;                                                     \
        sw t1, 0(t0);                                                   \
1:      j 1b;
```

```
#define RVTEST_FAIL                                                     \
        lui t0, 0x40008;                                                  \
        li t1, 0;                                                       \
        sw t1, 0(t0);                                                   \
1:      j 1b;
```

あとtohostも決め打ちしたので，いりません．

```
#define RVTEST_DATA_BEGIN                                               \
        EXTRA_DATA                                                      \
        .align 4; .global begin_signature; begin_signature:
```

あとumimpが嫌いなので，消します．

```
#define RVTEST_CODE_END
```

出力は.elfにしたいので `riscv-tests/isa/Makefile` をこんな感じに修正します．

```
%.dump: %
    $(RISCV_OBJDUMP) $<.elf > $@

define compile_template

$$($(1)_p_tests): $(1)-p-%: $(1)/%.S
    $$(RISCV_GCC) $(2) $$(RISCV_GCC_OPTS) -I$(src_dir)/../env/p
-I$(src_dir)/macros/scalar -T$(src_dir)/../env/p/link.ld $$< -o $$@.elf
```

次のコマンドでテストベンチが生成されます.

```
$ make isa
```

riscv-tests/isaにオブジェクトダンプとELFファイルが生成されます． さすがに，出力先をしているオプションがあるかと思いますが，また見つけれていないです． とりあえず，testディレクトリにコピーしましょう.

```
$ mkdir tests
$ cp riscv-tests/isa/rv32ui-p-*.elf tests/
```

## テストする

先ほど作ったベンチマークを元にテストしていきます． まず，Verilatorでシミュレーション可能な形式に変更していきます． ただ，命令とデータを取り出すだけです． 厄介なことに，今回はmissalignedアクセスを有効にするために， バイトアライメントのRAMを用意しました． 何を使ってもいいのですが，あんまりスクリプトがバラバラするとは嫌いなので， 記述はこんな感じになるかと思います.

`-O verilog` で `$readmemh` で読み取れる形式のファイルがダンプされる． が，今回はバイトアライメントのメモリを使用しているから使えているだけで，ワードアライメントだと 使い方がわからん.

```
tests:
    $(MAKE) -C riscv-tests isa
    mkdir -p tests
    cp riscv-tests/isa/rv32ui-p-*.elf tests/

hex: tests
    for elf in $(wildcard tests/*.elf); do \
        riscv32-unknown-elf-objcopy -O verilog $$elf tests/$$(basename $$elf
.elf).hex; \
    done
```

これでVerilog用の準備は完了

テストベンチはこんな感じで記述する.

```
```

# RISC-V ISA Specifications

https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications

RISC-Vの仕様書です. 年1回ぐらい更新があります. Volume IにRISC-Vの基礎がすべて書いてあります. 全部読む必要はないと思いますが， `2. RV32I Base Integer Instruction Set, Version 2.1` ぐらいは目を通しておくとよいかもしれません.

RISC-VはRV32I, RV32E, RV64E, RV64Iの中から最低1つ選ぶ必要があります. 通常多くの人はRV32IかRV64Iを選択します. Eを使ったプロセッサはあまり主流ではないと思います.

この基本整数命令セットに好きな拡張命令を追加して育てられるようにRISC-Vは設計されています. 例えばMは乗算/除算命令を表しますが，これを追加するとプロセッサはRV32IMに進化します. よく使うのはRV32I, M, A, Cです.

ABIは（） 通常 ilp32` を使います. これはintが32bit longが32bit pointerが32bitになります. `lp64` もあり，これはintが32bit longが64bit pointerが64bitになります. 研究室のプロセッサは32ビットなので，ilp32じゃないと動きません.

# [RISC-V GNU Toolchain]

# [Verilator](https://github.com/verilator/verilator)

Verilogのシミュレーションはこれでやりましょう. Verilatorです. aptでも落とせますが，24.04以前のバージョンだと古いものが落とされるので，ビルドしたほうがいいです. 使い方はこんな感じです.

# [Coremark](https://github.com/eembc/coremark)

自分の設計したプロセッサの性能を確かめたくなったらまずはCoremarkです. プロセッサの性能を測定することができまる.

# embedded-bench

タスク別のプロセッサの性能を調査するベンチマークです.

# Spike

RISC-Vのシミュレータです.

# RISC-V Opcodes

https://github.com/riscv/riscv-opcodes