# 第2回勉強会

RISC-Vのプロジェクトについて

2025-07-24

# Table of Contents

# Preamble

RISC-Vのプロセッサを設計する際に必要となるであろうプロジェクトについて説明をしていきます.

# RISC-V Technical Specifications

RISC-V Ratified Specifications からRISC-Vの内容が確定した仕様書を閲覧することができます.

## The RISC-V Instruction Set Manual Volume I: Unprivileged ISA

最も基本的なのは， The RISC-V Instruction Set Manual Volume I: Unprivileged ISA だと思います． ここにRISC-V ISAの基礎が書いてあります. 全部読む意味はあまりないと思いますが， **Chapter 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. また， **Chapter 35. RV32/64G Instruction Set Listings** の命令一覧はプロセッサの実装の際に非常に役に立ちます.

RISC-Vは基本整数命令セットとして， RV32I, RV32E, RV64E, RV64Iの中から1つ選ぶ必要があります. これらを同時に選択することはできません． 多くのプロジェクトではRV32IかRV64Iが使用されている印象です. Eは省電力向けの仕様ですが，あまり見たことないです. そもそも，後述のABIがDraftの段階です.

これらの基本整数命令セットに好きな拡張命令を追加できるようにRISC-Vは設計されています. 例えばM拡張は乗算/除算命令ですが，これを追加するとプロセッサはRV32IMと表されます. よく見る拡張命令を表にまとめました.

表 *1.* よく見る拡張命令

| Extension | desctiption |
|---|---|
| Zicsr | CSRレジスタを操作するための拡張命令です．CSRを実装しないなら必要ありません. |
| M | 整数乗算除算用の拡張命令です. |
| A | アトミック命令の拡張命令です．排他制御を実現します. |
| F | 単精度浮動小数演算の拡張命令です. |
| D | 倍精度浮動小数演算の拡張命令です. |
| C | 圧縮命令です．addやswが16ビットで表現できる拡張命令です. |
| V | ベクトル拡張命令です. |

## RISC-V ABIs Specification

RISC-V ABIs Specification も知っておくと良いかもしれません． RISC-VのABI (Application Binary Interface) は， レジスタの使い方の規約などを定めています（ちゃんと読んだことありません）． その中でも特に重要なのが，データ型のサイズや関数呼び出し時のレジスタの使い方です． 現在定義されているABIをまとめましたが， `ILP32` 以外を使うことはないと思います.

表 *2.* 現在定義されている*ABI*

| ABI | desctiption |
|---|---|
| ILP32 | RV32IのABI． Integer, Long, Pointerが32ビットになります. |
| ILP32F | F拡張をサポートするRV32IのABI. |
| ILP32D | D拡張をサポートするRV32IのABI. |
| ILP32E | RV32EのABI． まだDraft． （RV32EはRatified） |

| ABI | desctiption |
|---|---|
| LP64 | RV64IのABI．Long, Pointerが64ビットになります． |
| LP64F | F拡張をサポートするRV64IのABI． |
| LP64D | D拡張をサポートするRV64IのABI． |
| LP64Q | Q拡張をサポートするRV64IのABI． |

## RISC-V N-Trace (Nexus-based Trace)/Efficient Trace for RISC-V

N-Trace と E-Trace 知っておくと良いかもしれません． これらは，RISC-Vプロセッサのプログラムカウンタの系列をトレースするための仕様書です． 例えば，100MHzのプロセッサがあるとすると，毎秒100,000,000回PCが変化します． RV32IのPCは32ビットなので，これをそのまま送信すると，400MByte/sのトラフィックが発生します． この莫大なトラフィックを軽減することを目的とした仕様書がこの2つです． これを使うと酷くても1命令1ビット程度まで圧縮できます．

# RISC-V GNU Toolchain

RISC-V GNU Toolchainは RISC-Vのクロスコンパイラです．

## Build

ビルド方法を説明します．

研究室のサーバーを利用する場合は `/tools/cad/riscv/rv32ima` にあるので，ビルドする必要はありません．

ソースコードは GitHub で公開されています．

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

configureで使いたいRISC-Vのアーキテクチャとabiを選択します． 例えば，RV32IMを使用したい場合は次のようなオプションを指定します．

```
--with-arch=rv32im --with-abi=ilp32
```

*LINUX*を動作させるようなプロセッサを使用する場合は，

```
--with-arch=rv32ima_Zicsr_Zifencei_Zicntr --with-abi=ilp32
```

を使用します．

記述全体は次のようになります．

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

だいたい，5分から10分程度で終わります． ビルドが完了すると `/home/fujino/rv32im/bin` に `riscv32-unknown-elf-` から始まるコンパイラが生成されています．

`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． `riscv32-unknown-elf-g++` がないとかライブラリが見つからないなどの問題があったら， 失敗しています． この場合は，シングルスレッドで実行してみてください． 時間はかかりますが，確実です．

## Useage

使い方はx86のgccとあまり変わらないです． よく使うオプションを次にまとめました．

表 3. よく使うオプション

| Option | Describe |
|---|---|
| -march | 生成する命令セットを指定します． アーキテクチャが `RV32IMA_Zicsr` の場合， `-march=rv32ima_zicsr` のように指定します． |
| -mabi | ABIを指定します． `-mabi=ilp32` 以外使うことはないかと思います． |
| -nostartfiles | エントリーポイント `_start` をユーザー定義のものに変更します． このオプションなしで，初期化ルーチンをリンクすると，標準ライブラリの `_start` と衝突します． |
| -O | 最適化オプションです． `-Os` か `-O2` を使うことが多いです． |
| -T | リンカを指定します． `-Tsrc/link.ld` のように使います． |

CFU Proving Groundでは次のようにコンパイルしています．

```
$ riscv32-unknown-elf-gcc -Os -march=rv32im -mabi=ilp32 -nostartfiles -Iapp
-Tapp/link.ld -o build/main.elf app/crt0.s app/*.c *.c
```

| Option | Describe |
|---|---|
| -Os | 生成する実行可能ファイルのサイズを最小化する方向に最適化します． |
| -march=rv32im | RVProcがサポートするISAはRV32IMです． |
| -nostartfiles | RVProcは `crt0.s` から命令を実行したいので，GCC標準のスタートアップルーチンを生成しないオプションです． このオプションを使用する場合は，リンカに `_start` というエントリを教える必要があります． RVProcはこれを，`crt0.s` で行っています． |
| -Iapp | インクルードするヘッダーを `./app` ディレクトリから探してくれというオプションです． |
| -Tapp/link.ld | リンカを指定します．ここでは `./app/link.ld` を指定しています． |
| -o build/main.elf | 出力先を指定します． |

# Verilator

シミュレーションには Verilator を使うことが多いです． 高速で，波形が見れるのでVivado付属のXSIMよりこっちが便利です． `apt` でインストールできます．

```
sudo apt install verilator
```

*Ubuntu 24.04*以前を使っていると，*apt*は古い*verilator*をインストールします． 最新の版はビルドが必要です． この古い*verilator*では後述する `--binary` オプションが使えません． また，`always #5 clk = ~clk;` といった `#` を使った記述をサポートしていないので， テストベンチを*C*で記述する必要があります．

## Build

研究室のサーバーを利用する場合は `/tools/cad/bin/verilator` があるので，ビルドする必要はありません．

ソースコードは GitHub で公開されています．

```
$ git clone https://github.com/verilator/verilator -b v5.036
$ cd verilator
$ autoconf
$ ./configure
$ make -j $(nproc)
```

`verilator/build/bin` にverilatorが生成されます．

## Usage

使えるオプション はたくさんありますが，よく使うオプションをまとめておきます． Verilator はWarningが厳しすぎてすぐコンパイルを中止するので， `-Wno-` 系のオプションをめちゃくちゃ使います．

表 4. よく使うオプション

| option | desctiption |
|---|---|
| --binary | 実行可能ファイルを生成するオプションです． |
| --trace | `$dumpvars` を使いたいときは追加します． |
| --top < *module name* > | 階層の一番上のモジュール名を指定します． |
| -Wno-WIDTHEXPAND | 幅の違うwireをつなげるとwarningが表示されるので無効にします． |
| --timing | `initial forever \#5 clk ⇐ ~clk;` のような遅延記述を使う場合はこのオプションが必要です． |
| -I< *dirname* > | verilogで `` `include `` を使用している場合，ヘッダーのあるディレクトリを指定します． |

`top.v` にテストベンチを記述しているとすると，次のように記述します．

```
$ verilator --binary --trace -Isrc --top top \
      -Wno-WIDTHEXPAND --timing src/top.v
```

実行可能ファイルは `obj_dir/Vtop` になります． `./obj_dir/Vtop +memfile=/fujino/home/addi.mem` のような記述で，引数を与えることが可能で，

```
string str;
initial $value$plusargs("memfile=%s", str);
```

というように，テストベンチ側で受け取ることができます．

# RISC-V Tests

RISC-V Tests はRISC-Vの検証用のテストベンチ集です. 作ったプロセッサがRISC-Vの仕様を満たしているか検証できます.

## Build

RISC-V Testsはプロセッサ毎にビルドする必要があります． GitHub で公開されているのでクローンしてきます.

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

ここでは，RV32I向けにRISC-V Testsを修正する方法を説明します.

### riscv-tests/isa/Makefile

RV32Iで必要なテストは， `rv32ui` です． `riscv-tests/isa/Makefile` で次を変更します.

```
42|    default: rv32ui
```

### riscv-tests/env/p/link.ld

ハードウェア固有のメモリマップを定義するリンカです． 今回は命令メモリとデータメモリが統合されたメモリ( `ram` )を考えます.

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

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

SECTIONS
{
  .text.init : { *(.text.init) } > ram
  .text : { *(.text) } > ram
  .data : { *(.data) } > ram
  .bss : { *(.bss) } > ram
  _end = .;
}
```

### riscv-tests/env/p/riscv-test.h

このヘッダに初期化ルーチンを記述します． ですので，RISC-V Testsでは `crt0.S` は不要です． すべてのテストは `RVTEST_CODE_BEGIN` から実行されます．

```
183|    #define RVTEST_CODE_BEGIN        \
```

デフォルトの初期化ルーチンでは `Zicsr` 拡張や例外処理をサポートしていないと使えない命令が使用されているので， RV32I用に修正していきます． `_start:` が1番最初に実行される命令です． 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．

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

`RVTEST_PASS` と `RVTEST_FAIL` は テストに通ったときと落ちたときの処理を定義します． 例外処理をサポートしないプロセッサでは `fence` や `ecall` は使えないので，別のルーチンに変更します． 今回は，プロセッサが `0x40008000` に `777` を書き込んだらPASS, `0` を書き込んだらFAILという約束にします． 実装は次のようになります．

```
#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;
```

`tohost` はホストへデータを送信するアドレスを指定します． 一般的にはUARTのアドレスを指定します． 今回は `tohost` は `0x40008000` ですが，先ほどの `RVTEST_PASS` と `RVTEST_FAIL` で決め打ちしたので，今回は使いません．ですので削除します．

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

使用する場合は，リンカに

```
. = 0x40008000
.tohost : { *(.tohost) }
```

のように記述します．

umimp は例外処理をサポートしないプロセッサでは意味がないので，削除します．

```
#define RVTEST_CODE_END
```

以上の変更でもって， 次のコマンドでテストベンチを生成します．

```
$ make isa
```

riscv-tests/isa にオブジェクトダンプと実行可能ファイルが生成されます． ここで， rv32ui-p- と rv32ui-v- が生成されていると思います． rv32ui-v- は仮想アドレスをサポートするプロセッサ向けのテストです． 今回はサポートしない方針なので，rv32ui-p- のみを使用します． ここでは， tests ディレクトリにコピーします．

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

## Usage

実行可能ファイルは用意できたので，Verilogのシミュレーションで使える形に変換します． 変換には objcopy を使うことができます． hex がこの実装です． -O verilog で $readmemh で読み取れる形式のファイルを生成できます．

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

今回はバイトアライメントされたメモリを想定しています． ワードアライメントの場合は次の方法を使う方が確実です．

```
$ riscv32-unknown-elf-objcopy -O binary main.elf main.bin
$ dd if=main.bin of=mem.bin conv=sync bs=1KB
$ hexdump -v -e '1/4 "%08x\n"' main.bin > main.hex
```

テストベンチは次のように記述できるかと思います．

```
module top;
    reg clk = 1; initial forever #5 clk = ~clk; // 100MHz clock
    reg [63:0] cc = 0; always @(posedge clk) cc <= cc+1; // clock cycle counter

    string hex_file;
    initial begin
        if ($value$plusargs("hex_file=%s", hex_file)) begin
            $display("Loading hex file: %s", hex_file);
            $readmemh(hex_file, top.dut.ram);
        end else begin
            $display("No hex file specified, using default values.");
        end
    end

    reg done = 0;
    always @(posedge clk) begin
        if (top.dut.dbus_en == 4'b1111 && top.dut.dbus_write_addr ==
32'h40008000) begin
            if (top.dut.dbus_write_data == 777) $finish;
            else $fatal;
        end
        else if (cc == 1000) $fatal;
    end

    main dut(
        .clk_i(clk),
        .rx_i(1'b1),
        .tx_o()
    );
endmodule
```

検証は次のようなルールで自動化してしまうのが，楽です．

```
valid:
    @for file in $(wildcard tests/*.hex); do \
        if ! ./obj_dir/Vtop +hex_file=$$file > /dev/null; then \
            echo "\033[31m[FAIL] $$file\033[0m"; \
        else \
            echo "\033[32m[PASS] $$file\033[0m"; \
        fi; \
    done
```

通らなくていいテストがあります． それが，`ma_data` と `fance_i` です． `ma_data` はミスアライメントを検証するテストです． 例えば， `0x01` に対して `lw` するようなケースが検証されます． このアライメントは*RISC-V ISA*の必須要件ではないので，サポートしない方針なら無視できます． `fence_i` は命令メモリを書き換えた際のハザードをチェックします． 命令メモリを*ROM*として実装する場合は，通らなくて*OK*です．

# CoreMark

CoreMarkはプロセッサの性能を検証するためのベンチマーク プログラムです． 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す．

## How to Use

GitHubで公開されています． クローンしましょう．

```
$ git clone https://github.com/eembc/coremark
```

`coremark.md5` のハッシュ値と一致させなけらばならないファイルがあります． 当然これらは変更が許されていません． 他は自由に変更できます． とりあえず，我々が独自のアーキテクチャ向けに変更しなければならないのは， `barebones` ディレクトリ内のファイルです． 他のディレクトリは使わないです．

| file | desctiption |
|---|---|
| core_portme.c | 時間とか測定する関数です． |
| core_portme.h | size_tを使っていて怒られるので， `ee_u32` に変更します． |
| core_portme.mak | Makefileです．ベアメタルルーチン固有のリンカディスクリプタや初期化ルーチンを使用する場合，使いにくいのでいりません． |
| cvt.c | なにかわかりませんが使わないとバグります． |
| ee_printf.c | ベンチマーク固有の文字の表示先を指定するための関数が定義されています． |

## ビルドしよう

ビルドしましょう．

## core_portme.c

まず `barebones_clock()` です． この関数からベンチマークの実行に何クロックサイクル要したか，取得できるようにしてあげます． 今回はとりあえず `0x80000000` から取得できるようにしてあげます．

実装はこんな感じでしょうか．

```
CORETIMETYPE
barebones_clock()
{
    unsigned int cycle_low, cycle_high;
    cycle_low = *((unsigned int*)0x80000000);
    cycle_high = *((unsigned int*)0x80000004);
    return (CORETIMETYPE)((((unsigned long long)cycle_high) << 32)
                          | cycle_low);
```

```
}
```

あと，ちょうど繰り上がった瞬間にcycle_highをキャプチャすると最悪なので， タイマーを `0x80000008` で制御できるようにします． とりあえず，0ストップ，1スタートでいいでしょうか．

```
/* Function : start_time
        This function will be called right before starting the timed portion of
   the benchmark.

        Implementation may be capturing a system timer (as implemented in the
   example code) or zeroing some system parameters - e.g. setting the cpu
clocks
   cycles to 0.
*/
void
start_time(void)
{
    GETMYTIME(&start_time_val);
    *(int*)0x80000008 = 0;
}
/* Function : stop_time
        This function will be called right after ending the timed portion of
the
   benchmark.

        Implementation may be capturing a system timer (as implemented in the
   example code) or other system parameters - e.g. reading the current value of
   cpu cycles counter.
*/
void
stop_time(void)
{
    *(int*)0x80000008 = 1;
    GETMYTIME(&stop_time_val);
}
```

タイマーのリセットは初期化ルーチンでやってしまいます．

```
void
portable_init(core_portable *p, int *argc, char *argv[])
{
    *(int *)0x80000008 = 0;
    *(int *)0x80000000 = 0;
    *(int *)0x80000004 = 0;
```

こんな感じでタイマーはよいかと．

## core_portme.h

なぜか `size_t` が怒られるので，次に変更します．

```
typedef ee_u32          ee_size_t;
```

あと，ここでオプションをたくさん設定します． 大変です．

| define | desctiption |
|---|---|
| HAS_FLOAT | 浮動小数演算をサポートしていたら1にします．もちろん0です． |
| HAS_TIME_H | `time.h` をサポートしていたら1します．もちろん0です． |
| USE_CLOCK | `time.h` をサポートしていたら1します．もちろん0です． |
| HAS_STDIO | `stdio.h` をサポートしていたら1にします．もちろん0です． |
| HAS_PRINTF | `stdio.h` をサポートしていたら1にします．もちろん0です． |

## ee_printf.c

`uart_send_char()` なる関数を定義しなければなりません． とりあえずuartは0x40008000でいいですかね．

```
void
uart_send_char(char c)
{
    *(char *)0x40008000 = c;
}
```

これで完成です． コンパイルしましょう． リンカはさっきのを使います． 初期化ルーチンはレジスタリセットだけで良いでしょう．

ここで次の定数をdefineしてあげる必要があります．

| define | desctiption |
|---|---|
| FLAGS_STR | どんなオプションでコンパイルしたのか，ベンチマークの結果に表示してくれます |
| ITERATIONS | 何回カーネルを回すか指定します．10秒以上かかる回数にしないと怒られます． |
| CLOCKS_PER_SEC | 1秒当たりのクロック数を教えてあげることで，実時間と対応が取れるようになります． |

以上を踏まえるとこんな感じです． ここでは25MHzでプロセッサが動作すると仮定しています（ちゃんと確かめていません）． 最適化オプション使ってもOKです．

```
cmark:
    mkdir -p cmark
    riscv32-unknown-elf-gcc -O2 -static -nostartfiles -mcmodel=medany \
    -march=rv32i -mabi=ilp32 \
    -Tcoremark/link.ld -Icoremark -o cmark/coremark.elf \
    -DFLAGS_STR=\""-O2 -static -nostartfiles"\" \
    -DITERATIONS=1000 -DCLOCKS_PER_SEC=25000000 \
    coremark/*.c coremark/crt0.S
    riscv32-unknown-elf-objcopy -O verilog cmark/coremark.elf
cmark/coremark.hex
    riscv32-unknown-elf-objdump -D cmark/coremark.elf > cmark/coremark.dump
```

実行するとこんな感じの出力が得られます.

```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 720276619
Total time (secs): 28
Iterations/Sec   : 35
Iterations       : 1000
Compiler version : GCC15.1.0
Compiler flags   : -O2 -static -nostartfiles
Memory location  : STACK
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xd340
Correct operation validated. See README.md for run and reporting rules.
```

CoreMarkのスコアになるのは， `Iterations/Sec` です. ここでは，35になっています． めちゃくちゃ低いです． 普通に使うなら500は超えたいところです.

# Spike

Spike RISC-Vのシミュレータです. RISC-VのELFファイルを実行することができます.

```
git clone https://github.com/riscv-software-src/riscv-isa-sim
cd riscv-isa-sim
mkdir build
cd build
../configure --prefix=/home/fujino/riscv-sim
make -j $(nproc)
make install
```

すると `--prefix` で指定したディレクトリに `spike` が生成されます.

古すぎて*Releases*が使用できません. *master*ブランチをクローンしましょう.

proxy kernalというELFを実行する環境も必要です. ビルドしていきます.

```
$ git clone https://github.com/riscv-software-src/riscv-pk
$ mkdir build
$ cd build
$ ../configure --prefix=/home/fujino/rv32i --host=riscv32-unknown-elf --with
-arch=rv32i_zicsr_zifencei --with-abi=ilp32
$ make
$ make install
```

Hello, World!してみます.

```
int main() {
    printf("Hello, Spike!\n");
    return 0;
}
```

コマンドはこうです.

```
riscv32-unknown-elf-gcc main.c -o main
spike --isa=rv32i /home/fujino/rv32i/riscv32-unknown-elf/bin/pk main
```

```
Hello, Spike!
```

我々の先輩が*Spike*上で*Linux*を動かしたときの ログ もあります.

# RISC-V Opcodes

RISC-V Opcodes はRISC-Vのdefineを生成してくれるので便利.

```
make EXTENSIONS='rv_i rv32_i'
```

でRV32Iの命令をすべて生成できる． そんなに使う機会はないけど，Verilogのlocalparamとかtexの表とか自動で生成してくれるので便利.