

Veryl で作る RISC-V CPU

— 基本編 —

[著] kanataso

技術書典 11（2024 年秋）新刊

2024 年 11 月 2 日 ver 1.0

■免責

本書は情報の提供のみを目的としています。

本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。

また本書では、™、®、©などのマークは省略しています。

まえがき / はじめに

TODO : 大幅に書き換える

本書を手にとっていただき、ありがとうございます。

本書は、OS を実行できる程度の機能を持った高速な RISC-V の CPU を実装する方法についてわかりやすく解説した本です。この本を読めば、RISC-V の基本的な拡張、世の中の CPU がどのように実装されているかについて、ある程度理解することができます。

本書の目的

本書の目的は、OS を実行できる程度の CPU を実装することで、OS や CPU の動作について深く理解することです。

本書の対象読者

本書は次のような人を対象としています。

- 入門書を読むことで RISC-V の CPU を実装したことがある
- 自作の RISC-V CPU に高度な機能を実装したい / 高速化したい

本書には参考実装があるため、自前の RISC-V CPU 実装を用意しなくても本書を読み進めることができます。最初から挫折しないためには RISC-V のパイプライン処理の CPU を実装したことがあることが望ましいです。

前提とする知識

本書を読むにあたり、次のような知識が必要となります。

- ハードウェア記述言語の書き方 (SystemVerilog か Verilog)
- RV32I の実装に関する知識
- パイプライン処理の実装方法

本書では、RISC-V(RV32I) の実装方法やパイプライン処理についてほとんど解説していません。RV32I のパイプライン処理の CPU を実装する方法については次の書籍を参考にするをお勧めします。

- 新・標準プログラマーズライブラリ RISC-V で学ぶコンピュータアーキテクチャ 完全入門, 吉瀬謙二, ISBN 978-4-297-14008-3
- RISC-V と Chisel で学ぶ はじめての CPU 自作 ——オープンソース命令セットによるカスタム CPU 実装への第一歩, 西山悠太郎, 井田健太, ISBN 978-4-297-12305-5

ほとんど難なく本書を読むためには、SystemVerilog を利用している「RISC-V で学ぶコンピュータアーキテクチャ 完全入門」を読むことをお勧めします。

問い合わせ先

本書に関する質問やお問い合わせは、次のページまでお願いします。正誤表はここにあります。

- URL: <https://www.example.com/mybook/>

謝辞

本書は XXXX 氏と XXXX 氏にレビューしていただきました。この場を借りて感謝します。ありがとうございました。

凡例

本書では、プログラムコードを次のように表示します。太字は強調を表します。

```
print("Hello, world!");
```

 ←太字は強調

プログラムコードの差分を表示する場合は、追加されたコードを太字で、削除されたコードを取り消し線で表します。

```
print("Hello, world!");  
print("Hello,  "+name+"!");
```

 ←取り消し線は削除したコード
←太字は追加したコード

長い行が右端で折り返されると、折り返されたことを表す小さな記号がつきます。

```
123456789_123456789_123456789_123456789_123456789_123456789_123456789_123456789_
```

ターミナル画面は、次のように表示します。行頭の「\$」はプロンプトを表し、ユーザが入力するコマンドには薄い下線を引いています。

```
$ echo Hello
```

 ←行頭の「\$」はプロンプト、それ以降がユーザ入力

本文に対する補足情報や注意・警告は、次のようなノートや囲み枠で表示します。

.....

ノートタイトル

ノートは本文に対する補足情報です。

.....



タイトル

本文に対する補足情報です。



タイトル

本文に対する注意・警告です。

目次

まえがき / はじめに	i
第 1 章 Introduction	1
1.1 RISC-V	1
1.2 使用する言語	2
1.3 本書の構成	2
第 I 部 基本編	4
第 2 章 環境構築	5
2.1 Veryl	5
2.2 Verilator	5
2.3 riscv-gnu-toolchain	5
2.4 参考実装	5
第 3 章 ハードウェア記述言語 Veryl	6
3.1 あああ	6
第 4 章 RV32I の実装	7
4.1 概要	7
4.2 実装方針	7
4.3 実装	7
4.3.1 メモリ	7
4.3.2 デコーダ	7
4.3.3 レジスタを読む	7
4.3.4 ALU	7
4.3.5 メモリアクセス	7
4.3.6 レジスタに書き込む	7
4.3.7 CSR (ecall mret)	7
4.4 テスト	8
4.4.1 riscv-tests	8
4.4.2 終了検知	8
4.4.3 テストの実行	8

第 5 章	RV64I の実装	9
5.1	実装方針	9
5.2	メモリの実装	9
5.3	メモリバスの実装	9
5.4	IF ステージの実装	9
5.5	riscv-tests の確認	9
5.6	LWU, LD 命令の実装	9
5.7	SD 命令の実装	10
5.8	LUI, AUIPC 命令の実装	10
5.9	ADDW 命令の実装	10
5.10	ADDIW 命令の実装	10
5.11	SUBW 命令の実装	10
5.12	シフト命令の実装	10
第 II 部	基本的な拡張とトラップの実装	11
第 6 章	M 拡張の実装	12
6.1	概要	12
6.2	MUL[W] 命令	12
6.3	MULH 命令	12
6.4	MULHU 命令	12
6.5	MULHSU 命令	12
6.6	DIV[W] 命令	12
6.7	DIVU[W] 命令	13
6.8	REM[W] 命令	13
6.9	REMU[W] 命令	13
第 7 章	例外の実装	14
7.1	例外とは何か?	14
7.2	illegal instruction	14
7.3	メモリのアドレスのやつ	14
第 8 章	A 拡張の実装	15
8.1	概要	15
8.2	AMO 系	15
8.3	LR / SC	15
8.4	例外	15

第 9 章	C 拡張の実装	16
9.1	概要	16
9.2	実装方針	16
9.3	圧縮命令の変換	16
第 10 章	MMIO の実装	17
10.1	概要	17
10.2	実装方針	17
第 11 章	割り込みの実装	18
11.1	概要	18
11.2	UART RX	18
11.3	タイマ割り込み	18
第 III 部	privilege mode の実装	19
第 12 章	M-mode の実装	20
第 13 章	S-mode の実装	21
第 14 章	ページングの実装	22
14.1	ページングとは何か	22
14.2	PTW の実装	22
14.3	Sv32	22
14.4	Sv39	22
14.5	Sv48	22
14.6	Sv54	22
第 IV 部	OS を動かす	23
第 15 章	virtio の実装	24
第 16 章	xv6 の実行	25
付録 A	デバッグのための環境の整備	26
A.1	riscv-tests の実行	26
A.1.1	実行する	26
A.1.2	riscv-tests-bin ディレクトリ	29

A.1.3	riscv-tests の実行の流れ	30
A.1.4	riscv-tests の結果を確認するための仕組み	34
A.1.5	テストのためのオプションを実装する (book-sec5-impl-test-option)	34
A.2	プログラムを作成して実行する	37
A.3	ログの整備	37
A.4	Spike との比較	37
A.5	メモリ操作の追跡	37
A.6	ストールの検知	37
A.7	ビジュアライズ	37
あとがき / おわりに		38

第 1 章

Introduction

こんにちは！ あなたは CPU を作成したことがありますか？ 作成したことがあってもなくても大歓迎、この本は CPU 自作の面白さを世に広めるために執筆されました。実装を始める前に、まずは RISC-V や使用する言語、本書の構成について簡単に解説します。RISC-V や Veryl のことを知っているという方は、本書の構成だけ読んでいただければ OK です。それでは始めましょう。

1.1 RISC-V

RISC-V はカリフォルニア大学バークレー校で開発された RISC の ISA(命令セットアーキテクチャ) です。ISA としての歴史はまだ浅く、仕様書の初版は 2011 年に公開されました。それにも関わらず、RISC-V は仕様がオープンでカスタマイズ可能であるという特徴もあって、研究目的で利用されたり既に何種類もマイコンが市販されているなど、着実に広まっていきます。

インターネット上には多くの RISC-V の実装が公開されています。例として、rocket-chip(Chisel による実装)、Shakti(Bluespec SV による実装)、rsd(SystemVerilog による実装) が挙げられます。これらを参考にして実装するのもいいと思います。

本書では、RISC-V のバージョン riscv-isa-release-87edab7-2024-05-04 を利用します。RISC-V の最新の仕様については、riscv/riscv-isa-manual (<https://github.com/riscv/riscv-isa-manual/>) で確認することができます。

RISC-V には基本整数命令セットとして RV32I, RV64I, RV32E, RV64E が定義されています。RV の後ろにつく数字はレジスタの長さ (XLEN) が何ビットかです。数字の後ろにつく文字が I の場合、XLEN ビットのレジスタが 32 個存在します。E の場合はレジスタの数が 16 個になります。

基本整数命令セットには最低限の命令しか定義されていません。その代わり、RISC-V ではかけ算や割り算、不可分操作、CSR などの追加の命令や機能が拡張として定義されています。CPU が何を実装しているかを示す表現に ISA String というものがあり、例えばかけ算と割り算、不可分操作ができる RV32I の CPU は RV32IMA と表現されます。

本書では、まず RV32I の CPU を作成し、これを RV64IMACFD_Zicond_Zicsr_Zifencei に進化させることを目標に実装を進めます。

1.2 使用する言語

本書では、CPU の実装に Veryl というハードウェア記述言語を使用します。Veryl は SystemVerilog の構文を書きやすくしたような言語で、Veryl のプログラムは SystemVerilog に変換することができます。構文や機能はほとんど SystemVerilog と変わらないため、SystemVerilog が分かる人は殆どノータイムで Veryl を書けるようになると思います。Veryl の詳細については、「3.1 あああ」(p.6) で解説します。なお、SystemVerilog の書き方については本書では解説しません。

他にはシミュレーションやテストのために C++, Python を利用します。プログラムがどのような意味かについては解説しますが、SystemVerilog と同じように基本的な書き方については解説しません。

1.3 本書の構成

本書では、単純な RISC-V のパイプライン処理の CPU を高速化, 高機能化するために実装を進めていきます。まず OS を実行できる程度に CPU を高機能化したら、高速にアプリケーションを実行できるように CPU を高速化します。そのため、本書は大きく分けて高機能化編と高速化編の 2 つで構成されています。

高機能化編では、CPU で xv6 と Linux を実行できるようにします。OS を実行するために、かけ算, 不可分操作, 圧縮命令, 例外, 割り込み, ページングなどの機能を実装します (表 1.1)。

▼ 表 1.1: 実装する機能：高機能化編

章	実装する機能
あ	あ
あ	あ
あ	あ
あ	あ
あ	あ
あ	あ

高速化編では、CPU に様々な高速化手法を取り入れます。具体的には、分岐予測, TLB, キャッシュ, マルチコア化, アウトオブオーダー実行などです (表 1.2)。

本書では、筆者が作成したパイプライン処理の RV32I の参考実装 (bluecore) に機能を追加し、テストを記述し実行するという方法で解説を行っています。テストはシミュレーションと実機 (FPGA) で行います。本書で使用している FPGA は、Gowin 社の TangMega 138K というボードです。これは 3 万円程度で AliExpress で購入することができます。ただし、実機がなくても実装を進めることができるので所有していなくても構いません。

▼ 表 1.2: 実装する機能：高速化編

章	実装する機能
あ	あ
あ	あ
あ	あ
あ	あ
あ	あ
あ	あ
あ	あ
あ	あ



▲ 図 1.1: 使用する FPGA(TangMega138K)

第Ⅰ部

基本編

第 2 章

環境構築

2.1 Veryl

rustup cargo veryl vscode の拡張

2.2 Verilator

インストールするだけ

2.3 riscv-gnu-toolchain

2.4 参考実装

clone

第 3 章

ハードウェア記述言語 Veryl

3.1 あああ

あああ

第 4 章

RV32I の実装

4.1 概要

4.2 実装方針

4.3 実装

4.3.1 メモリ

4.3.2 デコーダ

4.3.3 レジスタを読む

4.3.4 ALU

4.3.5 メモリアクセス

4.3.6 レジスタに書き込む

4.3.7 CSR (ecall mret)

4.4 テスト

4.4.1 riscv-tests

古いのを appendix にする

4.4.2 終了検知

4.4.3 テストの実行

第 5 章

RV64I の実装

RISC-V には、64bit 環境向けの基本整数命令セットとして RV64I が用意されています。

5.1 実装方針

5.2 メモリの実装

5.3 メモリバスの実装

5.4 IF ステージの実装

5.5 riscv-tests の確認

5.6 LWU, LD 命令の実装

5.7 SD 命令の実装

5.8 LUI, AUIPC 命令の実装

5.9 ADDW 命令の実装

5.10 ADDIW 命令の実装

5.11 SUBW 命令の実装

5.12 シフト命令の実装

SLLIW, SRLIW, SRAIW, SLL, SRL, SRA, SLLW, SRLW, SRAW

第Ⅱ部

基本的な拡張とトラップの実装

第 6 章

M 拡張の実装

6.1 概要

6.2 MUL[W] 命令

6.3 MULH 命令

6.4 MULHU 命令

6.5 MULHSU 命令

6.6 DIV[W] 命令

6.7 DIVU[W] 命令

6.8 REM[W] 命令

6.9 REMU[W] 命令

第 7 章

例外の実装

7.1 例外とは何か？

7.2 illegal instruction

7.3 メモリのアドレスのやつ

いまのところこれだけ？

第 8 章

A 拡張の実装

8.1 概要

シングルコアなので超簡単テストを通すことだけを考える

8.2 AMO 系

8.3 LR / SC

8.4 例外

第 9 章

C 拡張の実装

9.1 概要

9.2 実装方針

フロントエンド

9.3 圧縮命令の変換

第 10 章

MMIO の実装

10.1 概要

UART TX/RX を作ります

10.2 実装方針

第 11 章

割り込みの実装

11.1 概要

11.2 UART RX

11.3 タイマ割り込み

第Ⅲ部

privilege mode の実装

第 12 章

M-mode の実装

第 13 章

S-mode の実装

第 14 章

ページングの実装

14.1 ページングとは何か

14.2 PTW の実装

14.3 Sv32

14.4 Sv39

14.5 Sv48

14.6 Sv54

第Ⅳ部

OS を動かす

第 15 章

virtio の実装

どうするか

第 16 章

xv6 の実行

付録 A

デバッグのための環境の整備

TODO 供養

CPU を記述するとき、バグを生まずに実装するのは不可能です。シミュレーション時にバグを見つけることができればいいのですが、CPU を製造した後にバグが見つかることもあります。一度製造したハードウェアを修正するのは非常に難しいため、CPU のベンダーはバグのことをエラッタ (errata) としてリスト化しており、CPU の上で動くソフトウェアはエラッタを回避するようにプログラムしなければいけません。そのため、バグのあるコードを記述してしまったときにバグの存在にできるだけ早く気付けるようにしておくことが重要です。また、バグがあることが判明したときはバグの原因を速く見つけれられるような仕組みがあるといいでしょう。

本章では、テストを記述して実行する方法を確認した後、バグの発生個所を速くを見つけるための仕組みを作成します。

CPU でプログラムが正しく動くことを確認するには、実装が正しいことを数学的に証明する、テストプログラムが正しく動くことを確認する、ランダムなプログラムを実行して問題なく動くことを確認するなど様々な手法がありますが、この中で最も簡単な手法はテストプログラムを実行することです。

CPU はプログラムを動かすために存在します。まずはプログラムを動かしてみることから始めましょう。

A.1 riscv-tests の実行

A.1.1 実行する

riscv-tests (<https://github.com/riscv-software-src/riscv-tests>) とは、RISC-V のテストスイートです。riscv-tests を実行することで、各命令がある程度正しく動くことを確認することができます。

プログラムを動かすのに一番手っ取り早いのは、すでに用意されたプログラムを動かすことです。bluecore にはコンパイル済みの riscv-tests のバイナリが含まれています。試しに ADD 命令のテスト (rv32ui-p-add) を実行してみましょう。

▼ リスト A.1: riscv-tests(rv32ui-p-add) を実行する

```
$ make build
$ make verilator MEMFILE=test/riscv-tests-bin/rv32ui-p-add.bin.hex CYCLE=0
(省略)
MEM ---
00000040:fc3f2223
stall      : 1
is_mem_op  : 1
is_csr_op  : 0
funct3     : 2
mem_out    : 00000093
csr_out    : 00000000
WB ---
wdata: 00000001
test: Success
- /core/src/top.sv:128: Verilog $finish
- /core/src/top.sv:128: Second verilog $finish, exiting
```

最終的に `test: Success` という文字が出力されたでしょうか？ もし Success ではなく Fail が出力されたりいつまでも出力されない場合、環境構築に失敗している可能性があります。正しい手順を踏んでいるか確認してください。

make build

Veryl のプログラムをそのままシミュレーションできる環境は今のところ存在しません。そのため、まずは `make build` を実行して veryl のプログラムを SystemVerilog に変換 (コンパイル) します。Makefile には `build` を実行したら、core/で `make build` を実行するように記述されています (リスト A.2)。

▼ リスト A.2: build (Makefile)

```
build:
    make -C ${core} build
```

core の Makefile の build を実行すると `veryl build` が実行される (リスト A.3) ため、これによって SystemVerilog プログラムが作成されます。

▼ リスト A.3: build (core/Makefile)

```
build:
    veryl build
```

veryl プログラムをコンパイルしたあとに `ls` コマンドを実行すると、リスト A.4 のように拡張子が veryl のファイルと同じ名前の sv ファイルが存在する状態になります。

▼ リスト A.4: make build を実行した後の core/src

```
$ ls -R core/src
core/src:
alu.sv      csrunit.sv      reg_forward.sv
alu.veryl   csrunit.veryl   reg_forward.veryl
```

```

alubr.sv      inst_decode.sv      svconfig.sv
alubr.veryl   inst_decode.veryl   top.sv
common       memunit.sv          top.veryl
core.sv       memunit.veryl       top_gowin.sv
core.veryl    packages           top_gowin.veryl

core/src/common:
fifo.sv       membus_if.sv        memory.sv
fifo.veryl    membus_if.veryl     memory.veryl

core/src/packages:
config.sv     corectrl.sv         csr.sv      eei.sv
config.veryl  corectrl.veryl      csr.veryl   eei.veryl

```

コンパイル結果の SystemVerilog ファイルを削除したい場合は、`make clean`、または `veryl clean` コマンドを実行します。

make verilator MEMFILE=~ CYCLE=~

`make verilator` コマンドは、Verilator でシミュレーションを実行するためのコマンドです。MEMFILE にメモリの初期値として読み込むファイルを指定し、CYCLE でシミュレーションの最長実行サイクル数 (0 で無制限) を指定します。実行した `make verilator MEMFILE=test/riscv-tests-bin/rv32ui-p-add.bin.hex CYCLE=0` では、メモリの初期値として `test/riscv-tests-bin/rv32ui-p-add.bin.hex`、最長実行サイクル数として `0` を指定しています。

riscv-tests.py

riscv-tests を実行するときに毎回オプションを指定するのは面倒です。これを楽にするために自動でテストを実行するプログラム (`test/riscv-tests.py`) を用意してあります。

▼ リスト A.5: rv32ui-p-から始まるテストをすべて実行する

```

$ cd test
$ make -C .. build
$ python3 riscv-tests.py -j 8 rv32ui-p-
(省略)
PASS : rv32ui-p-xor.bin.hex
PASS : rv32ui-p-sw.bin.hex
PASS : rv32ui-p-xori.bin.hex
Test Result : 39 / 39

```

リスト A.5 のように `riscv-tests.py` を実行すると、名前が `rv32ui-p-` から始まるテストを並列実行数が 8 (`-j 8`) で実行します。RV32I 向けのテストは 39 個あるため、それらがすべて実行され、結果として 39 個中 39 個のテストにパスしたという結果が表示されます。

それぞれのテストの実行時のログは `test/results` ディレクトリに格納されます。また、すべてのテストの成否についての情報は `test/result/results.txt` に記録されます。

A.1.2 riscv-tests-bin ディレクトリ

test/riscv-tests-bin ディレクトリには、次のファイルが含まれています。

- テストプログラムのバイナリ (*.bin)
- バイナリを SystemVerilog の \$readmemh タスクで読める形式に変換したファイル (*.bin.hex)
- バイナリのダンプファイル (*.dump)

hex ファイルはリスト A.6 のような形式になっています。これに対応する dump ファイルはリスト A.7 です。命令が一致しているのを確認できます。

▼ リスト A.6: hex ファイルの冒頭 10 行 (rv32ui-p-add.bin.hex)

```
0500006f
34202f73
00800f93
03ff0863
00900f93
03ff0463
00b00f93
03ff0063
00000f13
000f0463
```

▼ リスト A.7: dump ファイルの冒頭 10 命令 (rv32ui-p-add.dump)

```
rv32ui-p-add:      file format elf32-littleriscv

Disassembly of section .text.init:

00000000 <_start>:
   0:  0500006f          j      50 <reset_vector>

00000004 <trap_vector>:
   4:  34202f73          csrr   t5,mcause
   8:  00800f93          li     t6,8
  c:  03ff0863          beq    t5,t6,3c <write_tohost>
 10:  00900f93          li     t6,9
 14:  03ff0463          beq    t5,t6,3c <write_tohost>
 18:  00b00f93          li     t6,11
 1c:  03ff0063          beq    t5,t6,3c <write_tohost>
 20:  00000f13          li     t5,0
 24:  000f0463          beqz   t5,2c <trap_vector+0x28>
```

リスト A.8 で bin ファイルを確認すると、最初の命令 0500006f が 157 000 000 005 のように、リトルエンディアン形式で格納されていることが分かります。bluecore のメモリは 4byte のデータを 1byte 単位のビッグエンディアン形式で値を格納しているため、リトルエンディアン形式をビッグエンディアン形式に変換したファイルを作成する必要があります。

▼ リスト A.8: 1byte 単位で hexdump する

```
$ hexdump -b riscv-tests-bin/rv32ui-p-add.bin | head -1
00000000 157 000 000 005 163 057 040 064 223 017 200 000 143 010 377 003
```

bluecore にはリトルエンディアン形式からビッグエンディアン形式への変換を行うために test/bin2hex.py が用意されています。bin2hex.py の使い方はリスト A.9 の通りです。8byte 単位でエンディアンを変換したい場合は、4 を 8 に変更します。

▼ リスト A.9: リトルエンディアン形式をビッグエンディアン形式に変換する

```
$ python3 bin2hex.py 4 ファイル名 > 結果の保存先のファイル名
$ # 例 : rv32ui-p-add.bin -> rv32ui-p-add.hex
$ python3 bin2hex.py 4 riscv-tests-bin/rv32ui-p-add.bin > riscv-tests-bin/rv32ui-p-add.hex
```

A.1.3 riscv-tests の実行の流れ

さて、riscv-tests を実行できることを確かめたので、riscv-tests はどのようなプログラムを実行して CPU の確からしさ確かめているのかを確認します。

riscv-tests は基本的に次のような流れで実行されていきます。

1. `_start` : `reset_vector` にジャンプする
2. `reset_vector` : 各種状態を初期化する
3. `test_*` : テストを実行する。命令の結果がおかしかったら fail に飛ぶ。最後まで正常に実行できたら pass に飛ぶ。
4. fail, pass : テストの成否をレジスタに書き込み、`trap_vector` に飛ぶ
5. `trap_vector` : `write_tohost` に飛ぶ
6. `write_tohost` : テスト結果をメモリに書き込む。ここでループする

プログラムを参照しながら流れを確認していきましょう。test/riscv-tests-bin/rv32ui-p-add.dump を開いてください。

1. `_start`

bluecore はリセットされるとアドレス 0 からプログラムの実行を開始します。そのため、まずはアドレス 0 の `_start` から riscv-tests の実行がスタートします。`_start` には、`reset_vector` にジャンプする命令のみが記述されています。

▼ リスト A.10: `_start` (rv32ui-p-add.dump)

```
00000000 <_start>:
0: 0500006f          j          50 <reset_vector>
```

.....
疑似命令 (pseudo instruction)

j 命令は RISC-V の仕様書には定義されていません。それでは一体 `_start` に出てきた j 命令とは何でしょうか？ これはアセンブラでの記述を楽にするための疑似的な命令です。実際には

j 命令は jal 命令にコンパイルされます。j 命令の機械語 0500006f を RISC-V のデコーダー (<https://luplab.gitlab.io/rvcodecs/#q=0500006f>) で確認してみると jal x0, 80 と解釈されます。このことから j 命令は jal の PC の保存先レジスタ (rd) が x0 である、つまりジャンプだけしてリンクしない命令であることが分かります。

.....

2. reset_vector

reset_vector ではテストを実行するための準備を整えます。ここで今のところ注目する必要があるのは、レジスタの初期化 (リスト A.11) とテストへジャンプするためのコード (リスト A.12) です。レジスタの初期化部分では、0 番目のレジスタ以外のレジスタの値を 0 に設定しています*1。

▼ リスト A.11: レジスタの 0 初期化 (rv32ui-p-add.dump)

```
00000050 <reset_vector>:
50: 00000093          li    ra,0
54: 00000113          li    sp,0
(省略)
c4: 00000f13          li    t5,0
c8: 00000f93          li    t6,0
```

▼ リスト A.12: テストにジャンプするためのコード (rv32ui-p-add.dump)

```
174: 30005073          csrw   mstatus,0
178: 00000297          auipc  t0,0x0
17c: 01428293          add    t0,t0,20 # 18c <test_2>
180: 34129073          csrw   mepc,t0
184: f1402573          csrr   a0,mhartid
188: 30200073          mret
```

テストにジャンプするためのコードでは次のようになっています。

1. 174: CSR の mstatus レジスタを 0 に設定する
2. 180: mepc をテスト開始場所 (test_2) に設定する
3. 188: MRET 命令でテスト開始場所にジャンプ

ジャンプするための命令は MRET 命令です。M-mode のときに MRET 命令が実行されると、モードを mstatus レジスタの MPP ビット (幅は 2bit) に保存されている数値で表される権限レベルのモードに設定し、PC を mepc レジスタに設定された値に設定 (ジャンプ) します。

riscv-tests の rv32ui-p-add では、テストを U-mode で実行することを想定しています。そのため、mstatus レジスタに 0 を設定することで mstatus の MPP ビットを 0 に設定し、U-mode に遷移するようにしています。また、mepc レジスタに test_2 のアドレスを設定することで、テスト開始場所にジャンプするようにしています。

なお、今のところ bluecore は U-mode や mstatus レジスタをサポートしていないため、mret 命令は常に M-mode から S-mode にモードを変更し、mepc にジャンプするような命令になっています。

*1 RISC-V の 0 番目のレジスタ (x0, zero) は常に 0 であることが保証されています

3. test_*

test_*では命令のテストを実行します。リスト A.13 は ADD 命令のテストの 1 つです。このテストでは、ra レジスタに 0, sp レジスタに 0 をロードし、ra と sp を足した値を a4 レジスタに格納しています。足し算の結果が正しいことを確認するために、0 と一致しない場合には fail に遷移します。

▼ リスト A.13: test_2 (rv32ui-p-add.dump)

```
0000018c <test_2>:
18c: 00200193      li    gp,2
190: 00000093      li    ra,0
194: 00000113      li    sp,0
198: 00208733      add   a4,ra,sp
19c: 00000393      li    t2,0
1a0: 4c771663      bne   a4,t2,66c <fail>
```

最後のテストでは、fail に遷移しなかった場合には pass に遷移します (リスト A.14)。よって、テストに失敗した場合には fail に遷移し、成功した場合には pass に遷移します。

▼ リスト A.14: test_38(最後のテスト) (rv32ui-p-add.dump)

```
00000650 <test_38>:
650: 02600193      li    gp,38
654: 01000093      li    ra,16
658: 01e00113      li    sp,30
65c: 00208033      add   zero,ra,sp
660: 00000393      li    t2,0
664: 00701463      bne   zero,t2,66c <fail>
668: 02301063      bne   zero,gp,688 <pass>
```

各テストでは、テスト開始前に gp レジスタにテストごとに固有の値を格納しています。例えば test_2 では 2 を、test_38 では 38 を格納しています。この値は 0 以外になっています。

4. fail, pass

fail, pass では、gp レジスタに成功したか失敗したかを示す値を格納した後、ECALL 命令で trap_vector に遷移します。ECALL 命令は例外を発生させることで現在の権限レベルよりも高い権限のモードに移動するための命令^{*2}です。

RISC-V には、例外が発生して M-mode に遷移するとき、mcause レジスタに例外の発生原因を格納し、mepc レジスタに格納されている PC に遷移すると定義されています。bluecore は例外が発生するときに常に M-mode に遷移します。reset_vector で mepc に trap_vector のアドレスを設定しているため、fail, test は ECALL 命令によって trap_vector に遷移します。

▼ リスト A.15: fail (rv32ui-p-add.dump)

```
0000066c <fail>:
66c: 0ff0000f      fence
```

^{*2} 現在の権限レベルと同じ権限のままの場合もあります

```

670: 00018063      beqz    gp,670 <fail+0x4>
674: 00119193      sll     gp,gp,0x1
678: 0011e193      or      gp,gp,1
67c: 05d00893      li      a7,93
680: 00018513      mv      a0,gp
684: 00000073      ecall

```

▼ リスト A.16: test (rv32ui-p-add.dump)

```

00000688 <pass>:
688: 0ff0000f      fence
68c: 00100193      li      gp,1
690: 05d00893      li      a7,93
694: 00000513      li      a0,0
698: 00000073      ecall

```

リスト A.15 では、gp レジスタに格納された値を左に 1bit シフトし、最下位ビットを 1 にしています。リスト A.16 では、gp レジスタに 1 を格納しています。これにより、gp レジスタによってテストに成功したか失敗したかを区別することができます。

5. trap_vector

trap_vector では、mcause レジスタの値をロードし、その値が不正なものではないことを確認したら write_tohost にジャンプします。

bluecore で riscv-tests を実行するときは、fail,pass の ECALL 命令によって S-mode から M-mode に遷移します。そのため、mcause レジスタには S-mode からの Environment Call が原因であるとして 9 が格納されています。これがアドレス 14 で確かめられ、write_tohost にジャンプします。

▼ リスト A.17: trap_vector (rv32ui-p-add.dump)

```

00000004 <trap_vector>:
 4: 34202f73      csrr    t5,mcause
 8: 00800f93      li      t6,8
 c: 03ff0863      beq     t5,t6,3c <write_tohost>
10: 00900f93      li      t6,9
14: 03ff0463      beq     t5,t6,3c <write_tohost>
18: 00b00f93      li      t6,11
1c: 03ff0063      beq     t5,t6,3c <write_tohost>
20: 00000f13      li      t5,0
24: 000f0463      beqz    t5,2c <trap_vector+0x28>
28: 000f0067      jr      t5
2c: 34202f73      csrr    t5,mcause
30: 000f5463      bgez    t5,38 <handle_exception>
34: 0040006f      j       38 <handle_exception>

```

6. write_tohost

write_tohost では、gp レジスタの値を tohost(0x1000) に書き込む命令を実行することによって riscv-tests の結果を報告します。

tohost のアドレスは riscv-tests をコンパイルする際に設定します。riscv-tests のコンパイルに

については別の章で解説します。

▼ リスト A.18: write_tohost (rv32ui-p-add.dump)

```
0000003c <write_tohost>:
3c: 00001f17          auipc    t5,0x1
40: fc3f2223          sw       gp,-60(t5) # 1000 <tohost>
44: 00001f17          auipc    t5,0x1
48: fc0f2023          sw       zero,-64(t5) # 1004 <tohost+0x4>
4c: ff1ff06f          j        3c <write_tohost>
```

A.1.4 riscv-tests の結果を確認するための仕組み

riscv-tests は特定のアドレスに値を書き込むことによって結果を確認することを説明しました。bluecore ではメモリと core を接続する top モジュールで riscv-tests の結果を確認しています。

▼ リスト A.19: riscv-tests の結果を確認するコード (top.veryl)

```
always_ff (clk, rst) {
    if_reset {
        test_state = TestState::Reset;
    } else {
        // riscv-tests tohostでの書き込みを検知する
        if dbus_if.valid & dbus_if.wen & dbus_if.addr == RISCVTES_EXIT_ADDR {
            $display("wdata: %h", dbus_if.wdata);
            // 成功したかどうかを出力する
            if dbus_if.wdata == RISCVTES_WDATA_SUCCESS {
                $display ("test: Success");
                test_state = TestState::Success;
            } else {
                $error    ("test: Fail");
                test_state = TestState::Fail;
            }
            // テスト終了
            $finish();
        } else {
            if test_state == TestState::Reset {
                test_state = TestState::Running;
            }
        }
    }
}
```

`dbus_if` を監視し、書き込み要求かつアドレスが `RISCVTES_EXIT_ADDR` のとき、書き込む値をチェックします。`wdata` が `RISCVTES_WDATA_SUCCESS` のときは成功、それ以外のときは失敗とし、`finish` システムタスクでシミュレーションを終了します。`RISCVTES_` から始まる値は eei パッケージで定義されています。

A.1.5 テストのためのオプションを実装する (book-sec5-impl-test-option)

さて、top モジュールで riscv-tests の終了チェックを行っているわけですが、チェック処理は

riscv-tests ではないプログラムを実行しているときも動いてしまいます。このままでは普通のプログラムを実行するときにシミュレーションが止まってしまうため不便です。これをテストを実行しているというオプションを追加することで、チェック処理を動かさないようにします。

テストを実行するかどうかはシミュレーションの実行時に指定できると良いです。そのために、マクロの有無でオプションを指定できるようにします。

▼ リスト A.20: マクロと連動したパラメータを記述する (svconfig.sv)

```
// テストモードかどうか
`ifdef ENV_TEST
    localparam ENV_TEST = 1;
`else
    localparam ENV_TEST = 0;
`endif

// テストモードの時、結果を書き込むアドレス
`ifndef TEST_EXIT_ADDR
    `define TEST_EXIT_ADDR 'h1000
`endif
localparam TEST_EXIT_ADDR = `TEST_EXIT_ADDR;

// テストモードの時、結果が成功の時に書き込まれる値
`ifndef TEST_WDATA_SUCCESS
    `define TEST_WDATA_SUCCESS 1
`endif
localparam TEST_WDATA_SUCCESS = `TEST_WDATA_SUCCESS;
```

bluecore ではマクロを svconfig パッケージで利用しています。ここでリスト A.20 のように、マクロの有無によって値が変わるパラメータ (ENV_TEST) を定義します。加えて、結果を書き込むアドレスと成功を示す値をマクロで指定できるようにします。eei パッケージに定義されている RISCVTESST_EXIT_ADDR と RISCVTESST_WDATA_SUCCESS を削除し、代わりに TEST_EXIT_ADDR と TEST_WDATA_SUCCESS を定義します。

▼ リスト A.21: SystemVerilog のパッケージのパラメータをラップする (packages/config.veryl)

```
package config {
    local MEMORY_INITIAL_FILE: string      = $sv::svconfig::MEMORY_INITIAL_FILE;
    local ENV_TEST             : bit        = $sv::svconfig::ENV_TEST;
    local TEST_EXIT_ADDR       : logic <32> = $sv::svconfig::TEST_EXIT_ADDR;
    local TEST_WDATA_SUCCESS   : logic <32> = $sv::svconfig::TEST_WDATA_SUCCESS;
}
```

svconfig.veryl で定義したパラメータを\$sv キーワードを利用しないで利用するために、リスト A.21 のように config パッケージで\$sv キーワードを利用したパラメータを定義します。

▼ リスト A.22: チェック処理でパラメータを利用する (top.veryl)

```
if ENV_TEST :test_check {
    always_ff (clk, rst) {
        if_reset {
```

```

        test_state = TestState::Reset;
    } else {
        // テストの結果を書き込むアドレスへの書き込みを検知
        if dbus_if.valid & dbus_if.wen & dbus_if.addr == TEST_EXIT_ADDR {
            $display("wdata: %h", dbus_if.wdata);
            // 成功したかどうかを出力する
            if dbus_if.wdata == TEST_WDATA_SUCCESS {

```

そうしたら、top モジュールのチェック処理を if 文で囲いましょう。config パッケージは top.veryl の先頭でファイルスコープで import されているため、パッケージスコープを指定せずに使用することができます。

▼ リスト A.23: 新しく記述する変数 (riscv-tests.py)

```

TEST_EXIT_ADDR = "\\h1000"
TEST_WDATA_SUCCESS = "1"

```

▼ リスト A.24: コマンドのオプションを追加する (riscv-tests.py)

```

if len(args) == 0 or fileName.find(args[0]) != -1:
    mcmd = MAKE_COMMAND_VERILATOR
    options = []
    options.append("MEMFILE="+abpath)
    options.append("CYCLE=5000")
    options.append("MDIR="+fileName+"/")

    otherOptions = []
    otherOptions.append("-DENV_TEST")
    otherOptions.append("-DTEST_EXIT_ADDR="+TEST_EXIT_ADDR)
    otherOptions.append("-DTEST_WDATA_SUCCESS="+TEST_WDATA_SUCCESS)
    options.append("OPTION=\"\" + \"\".join(otherOptions) + \"\"")

    processes.append(executor.submit(test, mcmd + " " + "\".join(options), fileName))

```

最後に riscv-tests.py で make コマンドのオプションを指定するようにします。リスト A.23 のように、ファイルの先頭にテストの結果を書き込むアドレス (`TEST_EXIT_ADDR`) とテストが成功したときに書き込まれる値を示す変数を定義します。これをリスト A.23 のように `OPTION=` の中にマクロの定義として展開することで、 `make verilator` コマンドを実行するときにマクロが定義されるようになりました。

▼ リスト A.25: riscv-tests が正常に動くことを確認する

```

$ make build
$ cd test
$ python3 riscv-tests.py -j 8 rv32ui-p-
(省略)
PASS : rv32ui-p-xor.bin.hex
PASS : rv32ui-p-sw.bin.hex
PASS : rv32ui-p-xori.bin.hex
Test Result : 39 / 39

```

`make build` コマンドでビルドしなおしたら、`riscv-tests.py` を実行することで `riscv-tests` が正常に動作することを確認しましょう。通常のプログラムが動かせるかについては次の節で確認します。

この項での変更は `book-sec5-impl-test-option` タグで確認することができます。

A.2 プログラムを作成して実行する

A.3 ログの整備

ログの整備

A.4 Spike との比較

A.5 メモリ操作の追跡

A.6 ストールの検知

A.7 ビジュアライズ

Konata

あとがき / おわりに

いかがだったでしょうか。感想や質問は随時受け付けています。

著者紹介

ここに自己紹介を書きます

Veryl で作る RISC-V CPU

基本編

2024 年 11 月 2 日 ver 1.0 (技術書典 11)

著 者 kanataso

印刷所 日光企画

© 2024 カウプラン機関極東支部