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

本書を手に取っていただき、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ことをお勧めします。

- 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, 西山悠太朗，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!\n");          ←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!\n");~~          ←取り消し線は削除したコード
print("Hello, "+name+"!\n");      ←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_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。

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。

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、着実に広まっていっています。

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。

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、「2.1 あああ」(p.3) 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。

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)。

▼表 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 2)。

▼表 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

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。

目次

まえがき / はじめに **i**

Introduction **iv**

RISC-V iv

使用する言語 iv

本書の構成 v

第 I 部　基本編 **1**

第 1 章　環境構築 **2**

1.1 Veryl 2

1.2 Verilator 2

1.3 riscv-gnu-toolchain 2

1.4 参考実装 2

第 2 章　ハードウェア記述言語 Veryl **3**

2.1 あああ 3

第 3 章　RV32I の実装 **4**

3.1 プロジェクトの作成 4

3.2 定数の定義 5

3.3 メモリ 5

3.3.1 メモリのインターフェースの定義 5

3.3.2 メモリの定義 6

3.4 top モジュールの作成 6

3.5 命令フェッチ 7

3.5.1 命令フェッチの実装 7

3.5.2 動作の確認 7

3.6 命令のデコードと即値の生成 8

3.7 レジスタの定義と読み込み 8

3.8 ALU 8

3.9 ロード、ストア命令 9

3.9.1 LW, SW 命令 9

3.9.2 LH[U], LB[U], SH, SB 命令 9

3.10 レジスタに値を書き込む . . . 10
3.10.1 ライトバックの実装 . . . 10
3.10.2 ライトバックのテスト . . . 10
3.11 分岐, ジャンプ . . . 10
3.11.1 JAL, JALR 命令 . . . 10
3.11.2 分岐命令 . . . 11
3.12 riscv-tests でテストする . . . 11
3.12.1 最小限の CSR 命令の実装 . . . 11
3.12.2 終了検知 . . . 11
3.12.3 テストの実行 . . . 11

第 4 章 RV64I の実装 13
4.1 メモリの幅を広げる . . . 13
4.2 LW, LWU, LD 命令の実装 . . . 14
4.3 SD 命令の実装 . . . 14
4.4 LUI, AUIPC 命令の実装 . . . 14
4.5 ADDW, ADDIW, SUBW 命令の実装 . . . 14
4.6 シフト命令の実装 . . . 14
4.7 riscv-tests . . . 14

第 II 部 基本的な拡張とトラップの実装 15

第 5 章 M 拡張の実装 16
5.1 概要 . . . 16
5.2 MUL[W] 命令 . . . 16
5.3 MULH 命令 . . . 16
5.4 MULHU 命令 . . . 16
5.5 MULHSU 命令 . . . 16
5.6 DIV[W] 命令 . . . 16
5.7 DIVU[W] 命令 . . . 17
5.8 REM[W] 命令 . . . 17
5.9 REMU[W] 命令 . . . 17

第 6 章 例外の実装 18
6.1 例外とは何か？ . . . 18
6.2 illegal instruction . . . 18
6.3 メモリのアドレスのやつ . . . 18

第 7 章　A 拡張の実装　19
7.1　概要 . . . 19
7.2　AMO 系 . . . 19
7.3　LR / SC . . . 19
7.4　例外 . . . 19

第 8 章　C 拡張の実装　20
8.1　概要 . . . 20
8.2　実装方針 . . . 20
8.3　圧縮命令の変換 . . . 20

第 9 章　MMIO の実装　21
9.1　概要 . . . 21
9.2　実装方針 . . . 21

第 10 章　割り込みの実装　22
10.1　概要 . . . 22
10.2　UART RX . . . 22
10.3　タイマ割り込み . . . 22

第 III 部　privilege mode の実装　23

第 11 章　M-mode の実装　24

第 12 章　S-mode の実装　25

第 13 章　ページングの実装　26
13.1　ページングとは何か . . . 26
13.2　PTW の実装 . . . 26
13.3　Sv32 . . . 26
13.4　Sv39 . . . 26
13.5　Sv48 . . . 26
13.6　Sv54 . . . 26

第 IV 部　OS を動かす　27

第 14 章　virtio の実装　28

第 15 章　xv6 の実行　29

付録 A　デバッグのための環境の整備　30

A.1　riscv-tests の実行 30

A.1.1　実行する 30

A.1.2　riscv-tests-bin ディレクトリ 33

A.1.3　riscv-tests の実行の流れ 34

A.1.4　riscv-tests の結果を確認するための仕組み 38

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) 38

A.2　プログラムを作成して実行する 41

A.3　ログの整備 41

A.4　Spike との比較 41

A.5　メモリ操作の追跡 41

A.6　ストールの検知 41

A.7　ビジュアライズ 41

あとがき / おわりに　42

第 I 部

基本編

第 1 章

環境構築

1.1 Veryl

rustup cargo veryl vscode の拡張

1.2 Verilator

インストールするだけ

1.3 riscv-gnu-toolchain

1.4 参考実装

clone

第 2 章
ハードウェア記述言語 Veryl

2.1 あああ

あああ

第 3 章
RV32I の実装

本章では RISC-V の基本整数命令セットである RV32I を実装します。基本整数命令であるから、整数の足し算やビット演算、シフト命令などの最低限の命令しか実装されていません。RV32I には 32bit 幅のレジスタが 32 個用意されていて、0 番目のレジスタの値は常に 0 です。

本書では CPU の高速化のことは考えず、マルチサイクルで同時に一つの命令のみを実行する CPU を実装します。

3.1 プロジェクトの作成

まず、Veryl のプロジェクトを作成します。Veryl には、verylup という toolchain が用意されており、これを利用することで veryl をインストールすることができます。

▼リスト 3.1: verylup のインストール

```
$ cargo install verylup ← verylupのインストール
$ verylup setup ← verylupのセットアップ
[INFO ]  downloading toolchain: latest
[INFO ]   installing toolchain: latest
[INFO ]     creating hardlink: veryl
[INFO ]     creating hardlink: veryl-ls
```

Veryl をインストールできたらプロジェクトを作成します。ここでは適当に core という名前にしています。

▼リスト 3.2: 新規プロジェクトの作成

```
$ veryl new core
[INFO ]      Created "core" project
```

すると、プロジェクト名のフォルダと、その中に Veryl.toml が作成されています。

▼リスト 3.3: 作成された Veryl.toml

```
TODO 極性の設定とかについて書く
```

Veryl のプログラムを格納するために、src フォルダを作成しておいてください。

```
$ mkdir src
```

3.2　定数の定義

いよいよプログラムを記述していきます。まず、CPU 内で何度も使用する定数や型をパッケージに定義します。

`src/eei.veryl` を作成し、次のように記述します。

▼リスト 3.4: eei.veryl

```
```

EEI とは、RISC-V execution environment interface の略です。RISC-V のプログラムの実行環境とインターフェースという広い意味があり、ISA の定義も EEI に含まれているため名前を使用しています。

XLEN は、レジスタの長さを示す数字です。RV32I のレジスタの長さは 32 ビットであるため、値を 32 にしています。

3.3　メモリ

3.3.1　メモリのインターフェースの定義

CPU はメモリに格納されたプログラムを実行します。よって、CPU の実装のためにはメモリの実装が必要です。メモリの定義をするために、次の定数を `eei` パッケージに定義します。

▼リスト 3.5: 定数の定義

```
```

ILEN は、実装がサポートする命令の最大の幅を示す値です。RISC-V の命令の幅は、後の章で説明する圧縮命令を除けばすべて 32 ビットです。そのため、値を 32 をにしています。

また、何度も使用することになる型に名前を付けています。UIntX は幅が XLEN の符号なし整数、Inst は命令を格納するための型、Addr はメモリアドレスを格納するための型です。

メモリの読み書きの単位を 32 ビットとします。

メモリモジュールを定義するにあたって、次のようなポートが必要になります。

表

これらのポートを一つ一つ接続するのは面倒になるため、先に interface を定義しておきます。

▼リスト 3.6: インターフェースの定義 (memif.veryl)

interface を利用することで、ポートの相互接続を簡潔にすることができます。

3.3.2　メモリの定義

メモリを作る準備が整いました。`src/memory.veryl` を作成し、その中にメモリモジュールを記述します。

▼リスト 3.7: memory.veryl

memory モジュールには次のパラメータが定義されています。

表

memory モジュールのポートは次の通りです。

表

memory モジュールは `addr` ポートで受け取ったアドレスのデータを読み出し、次のクロックでデータを `rdata` レジスタに出力します。

メモリの初期値を$readmemh システムタスクでファイルに記述された値に設定します。ファイルには、次のアセンブリを示す、次の値を格納してください。

▼リスト 3.8: あ

▼リスト 3.9: あ

3.4　top モジュールの作成

次に、最上位のモジュールを定義します。

▼リスト 3.10: top.veryl

top モジュールには次のポートが定義されています。

表 clock, reset

先ほど作った memory モジュールをインスタンス化しています。memory モジュールのポートに接続するための memif インターフェースもインスタンス化しています。

3.5　命令フェッチ

3.5.1　命令フェッチの実装

CPU のメイン部分を作成していきます。

`src/core.veryl` を作成し、次のように記述してください。

▼リスト 3.11: core.veryl

```
```

`var pc : Addr = 0;` は、現在実行している命令のアドレスを示すレジスタの定義です。このようなレジスタのことをプログラムカウンタ (PC) と呼びます。単位はバイトで、初期値を 0 として定義しています。

core モジュールは、メモリにアドレスが PC のデータを要求します。メモリが要求を受け入れたら PC を 4 インクリメントします。また、次のクロック以降に返答が valid になるのを待ち、valid になったらまた PC のアドレスのデータを要求します。

次に、top モジュールで core モジュールをインスタンス化し、memif インターフェースを接続します。これによって、メモリと CPU が接続されました。

3.5.2　動作の確認

ここまでのプログラムが正しく動くか検証しましょう。

Veryl のプログラムは、次のようにビルドすることができます。

▼リスト 3.12: プログラムのビルド / フォーマット

```
$ veryl fmt ←プログラムのフォーマット
$ veryl build ←ビルド
```

ここでビルドとは `veryl` ファイルを SystemVerilog に変換することを指します。ビルドすると、`ファイル名.sv` と `ファイル名.sv.map` 、 `ファイルリストのやつ` が生成されます。

プログラムを実行 (シミュレーション) するために、verilator を利用します。

▼リスト 3.13: verilator

```
$ verilator にゃ
```

生成されたシミュレータを実行します。

▼リスト 3.14: 命令フェッチの動作チェック

```
```

メモリファイルのデータが 1 行ずつ読み込まれていることが確認できます。これらのビルド、シミュレータのビルドを一つのコマンドで済ますために、Makefile を作成しておきましょう。

▼リスト 3.15: Makefile

3.6　命令のデコードと即値の生成

次に各命令がどのような意味を持つのかを、命令のビットをチェックすることで取得します。

RV32I では、次の形の命令フォーマットが定義されています。ここに各形式の簡単な説明。

デコード処理を書く前に、デコードの結果生成する列挙子と構造体を `src/ctrl.veryl` に定義します。

まず、形式を示す enum を作成します。

次に、命令がどのような操作を行うかを示す構造体を作成します。

追加で、構造体を引数にとって、それがどのような命令であるかを判別する関数を作成しておきます。

それでは、命令のデコード処理を書きます。デコーダとして、`src/decode.veryl` を定義します。

decode モジュールでは、受け取った命令の OP ビットを確認し、その値によって InstType, InstCtrl, 即値を設定しています。処理の振り分けには case 文を使用しています。

decode モジュールを core モジュールでインスタンス化します。命令のデコード結果を表示し、次のように表示されているか確認してください。

3.7　レジスタの定義と読み込み

最初に説明した通り、RV32I では 32 ビット幅のレジスタが 32 個用意されています。0 番目のレジスタの値は常に 0 です。

core モジュールに、レジスタを定義します。初期値を 0 に設定しておきます。

RV32I の命令は、最大で 2 個のレジスタの値を同時に読み出します。命令の中のレジスタのアドレスを示すビットの場所は共通で、rs1, rs2, rd で示されています。このうち、rs1, rs2 はソースレジスタ、rd はディスティネーションレジスタ (結果の書き込み先) です。

簡単のために、命令がレジスタを使用するか否かにかかわらず、常にレジスタの値を読み出すことにします。0 番目のレジスタが指定されたときは、レジスタを読み込まずに 0 を読み込んでいます。

3.8　ALU

ALU とは、Arithmetic Logic Unit の略で、CPU の計算を行う部分です。ALU は足し算や引き算、シフト命令などの計算を行います。ALU でどの計算を行うかは、funct3, funct5 によって

判別します。

`alu.veryl` を作成し、次のように記述します。

プログラム

ポート定義

core モジュールで alu モジュールをインスタンス化します。

3.9　ロード、ストア命令

3.9.1　LW, SW 命令

RISC-V にはメモリのデータを読み込む/書き込む命令として次の命令があります。

表

これらの命令で指定するメモリのアドレスは足し算です。先ほど作った ALU は、ALU を使用する命令ではない場合は常に足し算を行うため、ALU の結果をアドレスとして利用できます。

まず 32 ビット単位で読み書きを行う LW, SW 命令を実装します。

メモリ操作を行うモジュールを `memunit.veryl` に定義します。

プログラム

memunit モジュールでは、命令がメモリ命令の時、ALU から受け取ったアドレスをメモリに渡して操作を実行します。書き込み命令の時は、書き込む値を memif.wdata に設定し、memif.wen を 1 に設定します。

memunit モジュールを core モジュールにインスタンス化します。ここで、memunit モジュールとメモリの接続は、命令フェッチ用のインターフェースとは別にしなくてはいけません。そのため、core モジュールに新しく memif_data を定義し、これを memunit モジュールと接続します。

これで top モジュールにはロードストア命令と命令フェッチのインターフェースが 2 つ存在します。しかし、メモリは同時に 1 つの読み込みまたは書き込みしかできないため、これを調停する必要があります。

top モジュールに、ロードストアと命令フェッチが同時に要求した場合は、ロードストアを優先するプログラムを記述します。

ロードストアには複数クロックかかるため、これが完了していないことを示すワイヤがあります。これを見て、core は処理を進めます。

アラインの例外について注記を入れる

3.9.2　LH[U], LB[U], SH, SB 命令

ロード、ストア命令には、2 バイト単位, 1 バイト単位での読み書きを行う命令も存在します。

まずロード命令を実装します。ロード命令は 32bit 単位での読み込みをしたものの一部を切り取ってあげればよさそうです。

プログラム

次に、ストア命令を実装します。ここで 32 ビット単位で読み込んだ後に一部を書き換えて書き込んであげる方法、またはメモリモジュール側で一部のみを書き込む操作をサポートする方法が考えられます。本書では後者を採用します。

memif インターフェースに、どこの書き込みを行うかをバイト単位で示すワイヤを追加します。

プログラム

これを利用して、読み込みして加工して書き込みという操作をサポートさせます。

プログラム

3.10　レジスタに値を書き込む

CPU はレジスタから値を読み込み、これを計算して、レジスタに結果の値を書き戻します。レジスタに値を書き戻すことを、ライトバックと言います。

3.10.1　ライトバックの実装

計算やメモリアクセスが終わったら、その結果をレジスタに書き込みます。書き込む対象のレジスタは rd 番目のレジスタです。書き込むかどうかは InstCtrl.reg_wen で表されます。

プログラム

3.10.2　ライトバックのテスト

ここで、プログラムをテストしましょう。

メモリに格納されている命令は〜なので、結果が〜になることを確認できます。

3.11　分岐, ジャンプ

まだ、重要な命令を実装できていません。分岐命令とジャンプ命令を実装します。

3.11.1　JAL, JALR 命令

JAL(Jump And Link) 命令は相対アドレスでジャンプ先を指定し、ジャンプします。ジャンプ命令である場合は PC の次の値を PC + 即値に設定するようにします。Link とあるように、rd レジスタに現在の PC+4 を格納します。

プログラム

JALR(Jump And Link Register) 命令は、レジスタに格納されたジャンプ先にジャンプします。レジスタの値と即値を加算し、次の PC に設定します。JAL 命令と同様に、rd レジスタに現在の PC+4 を格納します。

プログラム

3.11.2　分岐命令

分岐命令には次の種類があります。全ての分岐命令は相対アドレスで分岐先を指定します。

分岐するかどうかの判定を行うモジュールを作成します。

プログラム

alubr モジュールの＊が 1 かつ、分岐命令である場合、PC を PC+ 即値に指定します。分岐しない場合はそのままです。

3.12　riscv-tests でテストする

古いのを appendix にする。

riscv-tests は、RISC-V の CPU が正しく動くかどうかを検証するためのテストセットです。これを実行することで CPU が正しく動いていることを確認します。

riscv-tests のビルド方法については付録を参考にしてください。

3.12.1　最小限の CSR 命令の実装

riscv-tests を実行するためには、いくつかの制御用のレジスタ (CSR) と、それを読み書きする命令 (CSR 命令) が必要になります。それぞれの命令やレジスタについて、本章では深く立ち入りません。

mtvec

ecall 命令

mret 命令

3.12.2　終了検知

riscv-tests が終了したことを検知し、それが成功か失敗かどうかを報告する必要があります。

riscv-tests は終了したことを示すためにメモリのあああ番地に値を書き込みます。この値が 1 のとき、riscv-tests が正常に終了したことを示します。それ以外の時は、riscv-tests が失敗したことを示します。

riscv-tests の終了の検知処理を top モジュールに記述します。

プログラム

3.12.3　テストの実行

試しに add のテストを実行してみましょう。add 命令のテストは rv32ui-p-add.bin.hex に格納されています。これを、メモリの readmemh で読み込むファイルに指定します。

プログラム

ビルドして実行し、正常に動くことを確認します。

複数のテストを自動で実行する

add 以外の命令もテストしたいですが、そのために readmemh を書き換えるのは大変です。これを簡単にするために、readmemh にはマクロで指定する定数を渡します。

プログラム

自動でテストを実行し、その結果を報告するプログラムを作成します。

プログラム

この Python プログラムは、riscv-tests フォルダにある hex ファイルについてテストを実行し、結果を報告します。引数に対象としたいプログラムの名前の一部を指定することができます。

今回は RV32I のテストを実行したいので、riscv-tests の RV32I 向けのテストの接頭辞である rv32ui-p-引数に指定すると、次のように表示されます。

第 4 章
RV64I の実装

前章では RISC-V の 32bit 環境である RV32I の CPU を実装しました。RISC-V には 64bit 環境の基本整数命令セットとして RV64I が用意されています。本章では RV32I の CPU を RV64I にアップグレードします。

では、具体的に RV32I と RV64I は何が違うのでしょうか? RV64I では、レジスタの幅が 32bit から 64bit に変わり、各種演算命令の演算の幅も 64 ビットになります。

それに伴い、次の命令が追加で定義されます。

これらの命令は 32 ビット幅での演算を行うものか、64 ビット幅でロードストアする命令です。

本章では、ロードストア命令を実装した後、それ以外の命令を実装します。

命令を実装したら、riscv-tests を実行することで、rv32ui-p-が正常に動くことを検証してください。64 ビット向けのテストは rv64i-p-から始まるテストです。命令を実装するたびにテストを実行することで、命令が正しく実行できていることを確認してください。

4.1 メモリの幅を広げる

ロードストア命令を実装するにあたって、メモリの幅を広げます。現在のメモリの幅は 32 ビットですが、このままだと 64 ビットでロードストアを行う場合に最低 2 回のメモリアクセスが必要になってしまいます。これを 1 回のメモリアクセスで済ませるために、メモリ幅を 32 ビットから 64 ビットに広げます。

プログラム

命令フェッチ部では、64 ビットの読み出しデータの上位 32 ビット, 下位 32 ビットを PC の下位 3 ビットで選択します。PC[2:0] が 0 のときは下位 32 ビット、4 のときは上位 32 ビットになります。

プログラム

メモリ命令を処理する部分では、LW 命令に新たに rdata の選択処理を追加します。LB[U], LH[U] については上位 32 ビットの場合について追加します。ストア命令では、マスクを変更し、アドレスに合わせて wdata を変更します。

プログラム

4.2 LW, LWU, LD 命令の実装

LW 命令は、符号拡張するように変更します。LWU 命令は、LHU, LBU 命令と同様に 0 拡張すればよいです。LD 命令は、メモリの rdata をそのまま結果に格納します。

4.3 SD 命令の実装

SD 命令は、マスクをすべて 1 で埋めて、wdata をレジスタの値をそのままにします。

4.4 LUI, AUIPC 命令の実装

なんか変わったっけ？？？

4.5 ADDW, ADDIW, SUBW 命令の実装

32 ビット単位で足し算、引き算をする命令が追加されています。これに対応するために ALU を変更します。

結果は符号拡張する必要があります。

4.6 シフト命令の実装

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

32 ビット単位に対してシフトする命令が追加されています。これに対応するために ALU を変更します。

4.7 riscv-tests

RV64I のテストがすべて正常に実行できることを確認してください。

第II部

基本的な拡張とトラップの実装

第 5 章
M 拡張の実装

5.1 概要

5.2 MUL[W] 命令

5.3 MULH 命令

5.4 MULHU 命令

5.5 MULHSU 命令

5.6 DIV[W] 命令

引き放し法でやる

5.7 DIVU[W] 命令

5.8 REM[W] 命令

5.9 REMU[W] 命令

第 6 章
例外の実装

6.1 例外とは何か？

6.2 illegal instruction

6.3 メモリのアドレスのやつ

いまのところこれだけ？

第 7 章
A 拡張の実装

7.1 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

7.2 AMO 系

7.3 LR / SC

7.4 例外

第 8 章
C 拡張の実装

8.1 概要

8.2 実装方針

フロントエンド

8.3 圧縮命令の変換

第 9 章
MMIO の実装

9.1 概要

UART TX/RX を作ります

9.2 実装方針

第 10 章
割り込みの実装

10.1 概要

10.2 UART RX

10.3 タイマ割り込み

第 III 部

privilege mode の実装

第 11 章
M-mode の実装

第 12 章

S-mode の実装

第 13 章
ページングの実装

13.1 ページングとは何か

13.2 PTW の実装

13.3 Sv32

13.4 Sv39

13.5 Sv48

13.6 Sv54

第 IV 部

OS を動かす

第 14 章
virtio の実装

どうするか

第 15 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
0000000 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 の確からしさを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/rvcodecjs/#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

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 == RISCVTESTS_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 == RISCVTESTS_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 `RISCVTESTS_EXIT_ADDR` のとき、書き込む値をチェックします。 `wdata` が `RISCVTESTS_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。 `RISCVTESTS_` 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 `RISCVTESTS_EXIT_ADDR` と `RISCVTESTS_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つけています。

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
