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。

目次

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。

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.5) 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 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

5.13 合成する

5.14 シフト命令の実装の検討

5.15 シフト命令の複数サイクル化

5.16　効果の確認

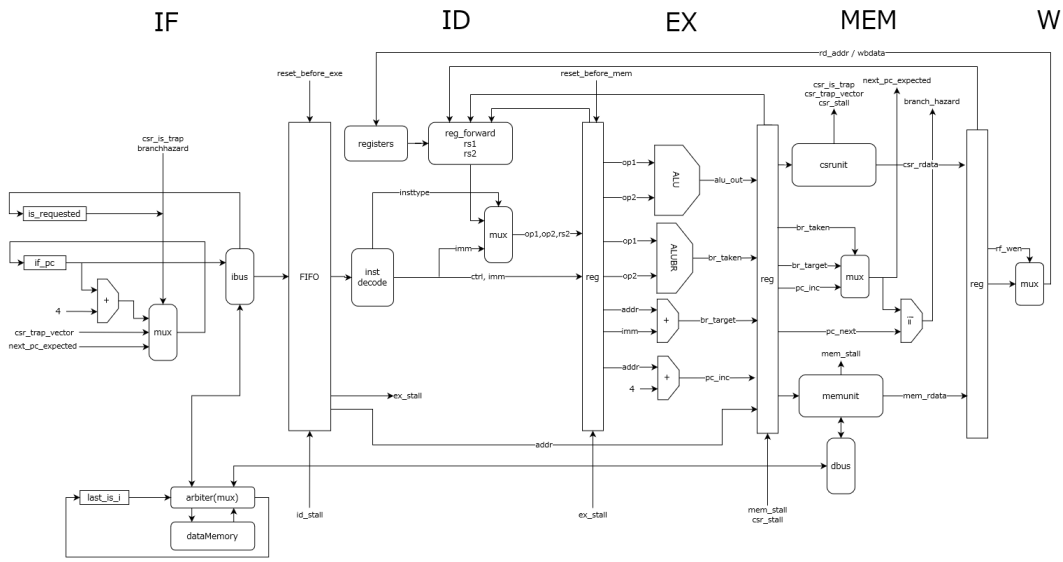
5.17　実機での動作確認

付録 A
参考実装 (bluecore) の解説

TODO 供養

本書では、参考実装に新しい機能を実装していくことで、RISC-V と CPU の実装について学んでいきます。本書では筆者が作成した RISC-V の CPU である bluecore を参考実装として利用します。この章では、bluecore がどのような構成になっているか、どのように命令が実行されていくかについて解説します。

A.1 概要



▲図 A.1: bluecore のアーキテクチャ図

bluecore はハードウェア記述言語 Veryl で記述されている 32bit の RISC-V の CPU です。EBREAK 以外の RV32I の命令に対応していて、他に CSRRW, CSRRS, CSRRC, CSRRWI,

CSRRSI, CSRRCI, MRET 命令を実行することができます。

RISC-V のテストスイートである riscv-tests の RV32I の命令のテスト (rv32ui-p からはじまるテスト) をすべて正常に実行することができます。CSR は mtvec, mepc, mcause のみ実装しています。これは riscv-tests を実行するのに必要な最低限のレジスタです。

命令は 5 段のパイプライン処理で実行されます。ステージは IF(命令フェッチ), ID(命令デコード), EX(実行), MEM(メモリ操作), WB(ライトバック) です。データハザードの解決は ID ステージで行います。また、EX, MEM, WB ステージから ID ステージへのフォワーディングが実装されています。

MEM ステージでは、メモリ操作だけではなく CSR 命令の処理と分岐ハザードの判定を行います。分岐の成否を求めるのは EX ステージですが、分岐ハザードを発生させるのは MEM ステージになります。そのため、分岐による PC の遷移処理と CSR 命令 (ecall や mret) によって発生する PC の遷移処理が同じステージで行われることになり、コードが少し単純になっています。

▼リスト A.1: ディレクトリ構成

```
bluecore
├── Makefile
├── README.md
├── core ← Verylのプロジェクトディレクトリ
│   ├── Makefile
│   ├── Veryl.lock
│   ├── Veryl.toml
│   └── src
│       ├── common ←汎用のVerylプログラムをまとめたディレクトリ
│       ├── packages ←パッケージをまとめたディレクトリ
│       ├── *.veryl ← Verylのプログラム
│       └── svconfig.sv ← coreの設定
├── src
│   └── tb.cpp ← verilatorでのテスト実行用のC++プログラム
├── synth ←合成用のプロジェクトディレクトリ
└── test
    ├── bin2hex.py ←バイナリをverilogが読める形式に変換するプログラム
    ├── riscv-tests-bin ←カスタマイズされたriscv-tests
    └── riscv-tests.py ← riscv-testsを実行するプログラム
```

A.2　最上位のモジュール (top.veryl)

最上位のモジュール (トップモジュール) は top(core/src/top.veryl) です。top では、memory と core のインスタンス化 (どちらも後述)、riscv-tests の終了判定を行っています。

▼リスト A.2: top のポート定義 (top.veryl)

```
module top (
    clk : input  logic     ,
    rst : input  logic     ,
    gpio: output logic<32>,
```

```
) {
```

top のポートは、clk, rst, gpio と定義されています。それぞれ、クロック信号, リセット信号, 32bit の汎用の IO です。top では、テストの状態 (成功, 失敗, 実行中) に応じて、gpio に設定する値を変更しています。

FPGA 向けに合成するときは、さらに上位のモジュールを用意します。例えば GOWIN の TangMega 138K 向けに合成する場合は top_gowin.veryl を利用します。top_gowin モジュールのポート定義には led があり、top モジュールの gpio 信号の下位 6bit を led の値としています。

▼リスト A.3: core のインスタンス化 (top.veryl)

```
    // coreのインスタンス化
    inst c: core (
        clk       ,
        rst       ,
        ibus_if   ,
        dbus_if   ,
    );
```

top モジュールでは、memory と core のインスタンス化と相互接続を行っています。memory は RAM のモジュールです。RAM には命令とデータが格納されます。core は命令を処理するモジュールです。core は、命令フェッチ, ロードストアのために RAM にアクセスするため、top モジュールで接続されています。

リスト A.3 は、core をインスタンス化している部分のコードです。 `ibus_if` , `dbus_if` はそれぞれ命令, データ用の RAM とのインターフェースです。

A.3　メモリ, メモリバス

メモリ (RAM) とメモリバス (メモリと core とのインターフェース) の構成について詳しく説明します。メモリから一度に読み込める, 書き込めるビット幅は 32bit です。これは eei パッケージ (packages/eei.veryl) に `MEMBUS_WIDTH` として定義されています (リスト A.4)。また、バスのビット幅のデータを表す型 (`MemBusData`) と、書き込み時に使用するマスクを表す型 (`MemBusMask`) も定義されています。

▼リスト A.4: メモリバスの設定 (packages/eei.veryl)

```
    local MEMBUS_WIDTH: u32                      = 32;
    type MemBusData   = logic<MEMBUS_WIDTH>    ;
    type MemBusMask   = logic<MEMBUS_WIDTH / 8>;
```

▼リスト A.5: メモリのインスタンス化 (top.veryl)

```
    // メモリのアドレスの幅
    local DATA_ADDR_WIDTH: u32 = 14;
```

```
    // アドレスのオフセット
    local DATA_ADDR_OFFSET: u32 = $clog2(MEMBUS_WIDTH / 8);

    local DATA_ADDR_MSB: u32 = DATA_ADDR_WIDTH + DATA_ADDR_OFFSET - 1;
    local DATA_ADDR_LSB: u32 = DATA_ADDR_OFFSET;

    // メモリのインスタンス化
    inst datamemory: memory #(
        DATA_WIDTH: MEMBUS_WIDTH        ,
        ADDR_WIDTH: DATA_ADDR_WIDTH     ,
        FILE_PATH : MEMORY_INITIAL_FILE,
    ) (
        clk                                             ,
        rst                                             ,
        ready : mem_ready                               ,
        valid : mem_valid                               ,
        wen   : mem_wen                                 ,
        addr  : mem_addr[DATA_ADDR_MSB:DATA_ADDR_LSB],
        wdata : mem_wdata                               ,
        wmask : mem_wmask                               ,
        rvalid: mem_rvalid                              ,
        rdata : mem_rdata                               ,
    );
```

リスト A.5 は、top モジュールでメモリをインスタンス化している部分のコードです。memory モジュールは、 `DATA_WIDTH` bit の幅のデータを 2 の `ADDR_WIDTH` 乗個並べたメモリを生成します。`DATA_WIDTH` は `MEMBUS_WIDTH` なので 32bit、 `ADDR_WIDTH` は `DATA_ADDR_WIDTH` なので 14 になります。したがって、メモリの大きさは 32bit(= 4byte) * 2 の 14 乗 (4096) = 16384byte = 16KB になります。

memory モジュールはアドレス 0 からマッピングされます。bluecore には MMIO(Memory Mapped IO) などは存在しないため、すべてのメモリ空間が memory モジュールへのアクセスにマップされます。

メモリの初期値は、`FILE_PATH` で指定されたファイルに記述された値になります。メモリはビッグエンディアン形式で値を格納します。

`config::MEMORY_INITIAL_FILE` パラメータは config パッケージ (packages/config.veryl) に定義された文字列型の定数です。 `MEMORY_INITIAL_FILE` パラメータの値は svconfig パッケージ (svconfig.sv) で定義された定数の値になっています。svconfig パッケージでは、定数にマクロの値を割り当てています。本書の執筆時点 (2024/05) では Veryl にマクロ機能が存在しないため、SystemVerilog を利用することによってマクロの値を取得しています。

RAM の初期値

bluecore の初期状態では FPGA への合成を前提としているため、RAM の値を$readmemh システムタスクによって初期化しています。普通の CPU では CPU の実行を ROM に格納された命令から開始し、補助記憶から RAM に命令やデータを読み込んで実行します。本書では、

bluecore が命令をそのような順で実行するように実装を変更していきます。

memory モジュールのポートは、名前が mem_から始まるワイヤを介して接続されます。リスト A.6 は、メモリとインターフェースを接続している部分のコードです。 `ibus_if` は命令フェッチ用, `dbus_if` はロードストア用としてインターフェースが定義されています。

▼リスト A.6: mem_*とインターフェースの接続 (top.veryl)

```
// メモリとinterfaceの接続
always_comb {
    // load/storeを優先する
    ibus_if.ready      = mem_ready & !dbus_if.valid;
    ibus_if.resp_valid = memarb_last_is_i & mem_rvalid;
    ibus_if.resp_rdata = mem_rdata;

    dbus_if.ready      = mem_ready;
    dbus_if.resp_valid = !memarb_last_is_i & mem_rvalid;
    dbus_if.resp_rdata = mem_rdata;

    mem_valid = ibus_if.valid | dbus_if.valid;
    if dbus_if.valid {
        mem_wen   = dbus_if.wen;
        mem_addr  = dbus_if.addr;
        mem_wdata = dbus_if.wdata;
        mem_wmask = dbus_if.wmask;
    } else {
        mem_wen   = 0;
        mem_addr  = ibus_if.addr;
        mem_wdata = ibus_if.wdata;
        mem_wmask = ibus_if.wmask;
    }
}
```

ibus_if.ready は、命令フェッチの要求をメモリが受け付けることができるかを示すワイヤです。これは、メモリモジュールが要求を受け付けることができる (`mem_ready`)、 `dbus_if` がロードストアの要求をしようとしていない (`!dbus_if.valid`) という 2 つの条件が満たされるときにのみ `1` になります。一方、ロードストアの要求を受け付けるかどうか (`dbus_if.ready`) は、`mem_ready` のみを条件としています。したがって、メモリが利用できるときは命令フェッチではなくロードストアが優先されることになります。

メモリへの指令は、 `dbus_if.valid` が `1` のときに `dbus_if` の値になり、 `0` のときには `ibus_if` の値になります。 `ibus_if` の値が採用されるとき、命令フェッチではメモリの値を書き換えないため、書き込み命令かどうか (`mem_wen`) は常に `0` になります。

A.4　パイプライン処理

前述のとおり、bluecore は 5 ステージのパイプライン処理を行う CPU です。パイプライン処理とは、「各処理がステージ (段) に分割されていて、先行する処理全体の完了を待たずにステージごとに次の処理を開始する」[*1] ような処理のことです。CPU においては、1 クロックで 1 つの命令を実行するのではなく、例えば IF, ID, EX, MEM, WB といった段階に分割し、複数クロックで実行することによってパイプライン処理が実現されます。パイプライン処理で命令を処理すると、回路のクリティカルパスが短くなることで最大周波数 (fMax) が向上し、命令のスループットが上がることが期待されます。

bluecore では命令を次の 5 つのステージで処理します。

1. IF：命令をメモリからフェッチする
2. ID：命令を解析し、どのように実行するかを求める
3. EX：計算する
4. MEM: ロードストアをおこなう
5. WB：レジスタに結果をライトバックする

core モジュールには、各ステージの処理や接続、制御が記述されています。

A.4.1　IF ステージ (Instruction Fetch Stage)

IF ステージは、メモリから命令をフェッチし、フェッチした命令を ID ステージに渡します。

リスト A.7 は、メモリバスに対して命令フェッチ要求を送る部分のコードです。命令フェッチの要求は、IF ステージと ID ステージの間の FIFO に命令を追加することができる (`idq_wready_next`) かつ、すでに要求済みの場合 (`if_is_requested`) はフェッチ結果が取得できた (`ibus_if.resp_valid`) という条件を満たしたときにのみ有効になります (`ibus_if.valid`)。要求するアドレス `ibus_if.addr` は、 `if_pc` レジスタの値になります。

▼リスト A.7: IF ステージのメモリバスとの接続 (core.veryl)

```
    // IFステージ
    var if_pc           : Addr ; // PC
    var if_is_requested: logic; // フェッチ中かどうか
    var if_requested_pc: Addr ; // フェッチ中のPC

    // フェッチ命令
    always_comb {
        ibus_if.valid = idq_wready_next & (!if_is_requested | ibus_if.resp_valid);
        ibus_if.addr  = if_pc;
        ibus_if.wen   = 0;
        ibus_if.wdata = 'x; // 命令フェッチは書き込まない
        ibus_if.wmask = 0;
    }
```

[*1] FPGA の原理と構成, 天野英晴

▼リスト A.8: IF ステージのレジスタの制御 (core.veryl)

```
    always_ff(clk, rst) {
            (省略)
            // フラッシュ時の動作
            if flush_before_memstage {
                if_is_requested = 0; // 次のクロックでフェッチ開始する
                if_pc           = if mem_csr_is_trap {
                    mem_csr_trap_vector // トラップ
                } else {
                    mem_next_pc_expected // 分岐ハザード
                };
            } else {
                if ibus_if.ready & ibus_if.valid {
                    if_pc           = if_pc + 4;
                    if_is_requested = 1;
                    if_requested_pc = if_pc;
                } else {
                    if if_is_requested & ibus_if.resp_valid {
                        if_is_requested = 0;
                    }
                }
            }
```

リスト A.8 は、`if_pc` レジスタなどを更新する always_ff ブロックです。IF ステージがフラッシュされる (`flush_before_memstage`) とき、フラッシュの原因に基づいて `if_pc` レジスタを更新します。フラッシュの原因は次の 2 種類があります

- CSR 命令 (ecall や mret) によるトラップ
- 分岐ハザードによるトラップ

IF ステージがフラッシュされないときは、命令をフェッチします。命令をフェッチできるとき、`if_pc` をインクリメントして、フェッチを要求したかどうかを `1` にし、要求中のアドレスを `if_pc` (インクリメントされる前の値) に設定します。命令をフェッチできないとき、すでにフェッチ要求済みでフェッチ結果が取得できた場合は、フェッチを要求したかどうかを `0` にします。

CPU がリセットされたとき、プログラムは実行をアドレス 0 から開始します。

A.4.2　ID ステージ (Instruction Decode Stage)

ID ステージは、IF ステージから命令を受け取り、命令を実行するために必要な情報を生成して EX ステージに渡します。ID ステージには、命令のデコード、データハザードの解決の大きく分けて 2 つの役割があります。

命令のデコード

命令を実行するために必要な情報の生成 (デコード処理) は、inst_decode.veryl で行っています。デコード処理では、命令の下位 7bit(`op`) を case 文で分岐することによって、制御用の値 (`ctrl`) と即値 (`imm`) を生成しています (リスト A.9)。

▼リスト A.9: デコード処理の一部 (inst_decode.veryl)

```
    always_comb {
        case op {
            OP_LUI: {
                        imm  = imm_u;
                        ctrl = {T, InstType::U, T, T, F, F, F, F, F, f3, f7};
                    }
            OP_AUIPC: {
                          imm  = imm_u;
                          ctrl = {T, InstType::U, T, F, F, F, F, F, F, f3, f7};
                      }
```

▼リスト A.10: 制御用の値の型の定義 (packages/corectrl.veryl)

```
    enum InstType: logic<6> {
        X = 6'b000000,
        R = 6'b000001,
        I = 6'b000010,
        S = 6'b000100,
        B = 6'b001000,
        U = 6'b010000,
        J = 6'b100000,
    }

    struct InstCtrl {
        is_legal : logic      , // 命令が合法である
        insttype : InstType   , // 命令のType
        rf_wen   : logic      , // レジスタに書き込むかどうか
        is_lui   : logic      , // LUI命令である
        is_aluop : logic      , // ALUを利用する命令である
        is_jump  : logic      , // ジャンプ命令である
        is_load  : logic      , // ロード命令である
        is_system: logic      , // CSR命令である
        is_fence : logic      , // フェンス命令である
        funct3   : logic   <3>, // 命令のfunct3フィールド
        funct7   : logic   <7>, // 命令のfunct7フィールド
    }
```

リスト A.10 は、制御用の値を表す型の定義 (`InstCtrl`) です。ID ステージ以降のステージでは、InstCtrl のメンバーの値をみることで命令を実行していきます。それぞれのメンバーの意味についてはコメントと inst_decode モジュールを参照してください。

RISC-V の命令フォーマット

TODO OP の表と Type 分類

データハザードの解決

ID ステージでは、EX ステージや MEM ステージで利用するソースレジスタ (rs1, rs2) の値を解決する処理を行います。

ここにデータハザードを説明するいい感じの図

ID ステージにある命令が使用するレジスタが、EX, MEM, WB ステージに存在する命令のデスティネーションレジスタ (rd) である場合、ID ステージにある命令は先のステージにある命令の結果に依存しています。このとき、依存している命令の結果がレジスタに書き込まれるのを待つ (ストールする) 必要があります。しかし、レジスタを介してデータを受け取るのではなく、依存している命令があるステージから直接データを受け取るようにすることで、ストールする必要があるサイクル数を減らすことができます。この手法のことをフォワーディングといいます。

▼リスト A.11: フォワーディングの制御を行う処理 (reg_forward.veryl)

```
    let ex_hazard : logic = ex_valid & ex_addr == addr;
    let mem_hazard: logic = mem_valid & mem_addr == addr;
    let wb_hazard : logic = wb_valid & wb_addr == addr;

    always_comb {
        data_hazard = valid & addr != 0 & (if ex_hazard {
            !ex_can_fw
        } else if mem_hazard {
            !mem_can_fw
        } else if wb_hazard {
            !wb_can_fw
        } else {
            0
        });
        result = if addr == 0 {
            0
        } else if ex_hazard {
            ex_data
        } else if mem_hazard {
            mem_data
        } else if wb_hazard {
            wb_data
        } else {
            reg_data
        };
    }
```

bluecore では、EX, MEM, WB ステージから ID ステージへのフォワーディングを実装しています。リスト A.11 は、フォワーディングの制御を行う reg_forward モジュールの一部です。 `data_hazard` は、データハザードが発生して待つ必要があるかどうかを示すワイヤです。 `data_hazard` は、ソースレジスタのアドレス (`addr`) が 0 ではなく、 `addr` と EX, MEM, WB ステージのデスティネーションレジスタのアドレスが一致するとき、一致したステージからフォワーディングができない場合に `1` になります。 `result` は、フォワーディングした結果の値のワイヤです。依存性がある場合は依存性があるステージから提供される値が設定され、依存性がない場合はレジスタから読みだした値が設定されます。

ID ステージでライトバックする値が確定する命令

bluecore では、LUI 命令, 分岐命令, ジャンプ命令のライトバックする値を ID ステージで確定

させます。それぞれの命令のライトバックする値は表 A.1 のようになります。

▼表 A.1: ID ステージでライトバックする値が確定する命令

命令	ライトバックする値
LUI 命令	即値 (を 12bit 左にシフトした値)
分岐命令	PC + 4
ジャンプ命令	PC + 4

リスト A.12 は、ライトバックする値を確定させる処理です。EX ステージに渡すデータにはライトバックする値を管理する `wbctx` という構造体が含まれています。 `valid` にはすでに値が確定したかどうか、 `addr` にはデスティネーションレジスタのアドレス、 `data` には確定した値が格納されます。 `valid` で使用している inst_is_から始まる function は、引数に制御用の値 (`InstCtrl`) を受け取り、それが何の命令かを判定する関数です。これらの function はすべて corectrl パッケージに記述されています。

▼リスト A.12: ID ステージでライトバックする値を確定させる処理 (core.veryl)

```
        // LUI, BRANCH, JUMPは、IDステージでライトバックする値が確定する
        exq_wdata.wbctx.valid = inst_is_lui(id_ctrl) | inst_is_branch(id_ctrl) | inst_is_jump(id
›_ctrl);
        exq_wdata.wbctx.addr  = id_rd_addr;
        exq_wdata.wbctx.data  = if inst_is_lui(id_ctrl) {
            id_imm
        } else {
            idq_rdata.addr + 4
        };
```

A.4.3　EX ステージ (EXecution Stage)

EX ステージは、ID ステージから制御用の値と演算用の値 (ソースレジスタの値や即値) を受け取り、演算を行うステージです。演算には、ライトバックする値を求めるための演算、MEM ステージで利用するアドレスを求めるための演算、分岐が発生するか判定するための演算の 3 種類があります。core モジュールでは演算のためのモジュールとして、alu, alubr をインスタンス化しています (リスト A.13)。

▼リスト A.13: alu, alubr モジュールのインスタンス化 (core.veryl)

```
    inst ex_alu: alu (
        ctrl  : exq_rdata.ctrl,
        op1   : exq_rdata.op1 ,
        op2   : exq_rdata.op2 ,
        result: ex_alu_out    ,
    );

    inst ex_alubr: alubr (
        funct3: exq_rdata.ctrl.funct3,
        op1   : exq_rdata.op1         ,
```

```
        op2   : exq_rdata.op2        ,
        take  : ex_br_taken          ,
    );
```

alu モジュール (alu.veryl)

alu モジュールは、様々な演算を実行した結果を制御用の値に応じて選択するモジュールです。`InstCtrl.is_aluop` が `1` のとき、`InstCtrl.funct3` の値に応じて加算や減算、シフトやビット演算などの結果が選択されます。`0` のときは、常に加算の結果が選択されます[*2]。

▼表 A.2: alu の演算の種類

funct3	演算
3'b000	加算、または減算
3'b001	左シフト
3'b010	符号あり <=
3'b011	符号なし <=
3'b100	ビット単位 XOR
3'b101	右 (論理|算術) シフト
3'b110	ビット単位 OR
3'b111	ビット単位 AND

alubr モジュール (alubr.veryl)

alubr モジュールは、分岐の成否を判定するための演算を行うモジュールです。`InstCtrl.funct3` の値に応じて、`==`, `!=`, `>`, `>=` などの結果が選択されます。分岐の成否は `taken` に割り当てられています。bluecore では分岐によってパイプラインをフラッシュする操作を MEM ステージで行うため、EX ステージは MEM ステージに分岐の成否を渡します。

▼表 A.3: alubr の演算の種類

funct3	演算
3'b000	==
3'b001	!=
3'b100	符号あり <=
3'b101	符号あり >
3'b110	符号なし <=
3'b111	符号なし >

EX ステージでライトバックする値が確定する命令

EX ステージでは、メモリ命令, CSR 命令以外のすべての命令でライトバックする値が確定しま

[*2] RISC-V では、演算した結果を利用して操作を行う命令 (例えばロードストア命令) は加算しか利用しません。bluecore では、そのような命令の is_aluop を 0 としています。

す。リスト A.14 は、EX ステージでライトバックする値を確定させる処理です。 `wbctx.valid` , `wbctx.data` は、EX ステージよりも前で値が確定していた場合には値をそのまま引き継いでいます。新しくライトバックする値が確定する場合、 `data` には ALU の演算結果 (`ex_alu_out`) が格納されます。

▼リスト A.14: EX ステージでライトバックする値を確定させる処理 (core.veryl)

```
        // メモリ命令, CSR命令以外は、EXEステージでライトバックする値が確定する
        memq_wdata.wbctx.valid = exq_rdata.wbctx.valid | !(inst_is_memory_op(exq_rdata.ctrl) | i›
›nst_is_csr_op(exq_rdata.ctrl));
        memq_wdata.wbctx.addr  = exq_rdata.wbctx.addr;
        memq_wdata.wbctx.data  = if exq_rdata.wbctx.valid {
            exq_rdata.wbctx.data
        } else {
            ex_alu_out
        };
```

EX ステージがストールする条件

EX ステージの次の MEM ステージでは、分岐, ジャンプ命令の次に実行される命令が正しい遷移先のものかどうかを判定しています。そのためには、分岐, ジャンプ命令の次の命令のアドレスが必要になります。EX ステージは次の命令のアドレスを取得するために、ID ステージに命令が供給されて次の命令のアドレスが判明するまでストールします。

A.4.4　MEM ステージ (MEMory Stage)

MEM ステージは、EX ステージから制御用の値と演算結果を受け取って、ロードストア命令の場合はメモリアクセスを行うステージです。また、CSR 命令の処理や、分岐ハザードを発生させる処理も行います。

分岐, ジャンプの判定

分岐やジャンプ命令の次の PC は、EX ステージまで実行しないと分かりません。しかし、EX ステージで次の PC が確定するまで次の命令をフェッチしないでいるのは効率が悪いです。bluecore では、分岐が常に成立しないことを仮定して投機的に命令をフェッチ, 実行します。ただし、EX ステージで分岐の成否が確定してから MEM ステージで PC の遷移先を確かめることで、間違った命令が実行されないようにしています。

本来の遷移先と違う命令がパイプラインに取り込まれてしまっている場合、EX ステージ以前のステージを無効化 (フラッシュ) し、新しく命令をフェッチしなおします。このような状況のことを分岐ハザードといいます。

▼リスト A.15: 分岐ハザードの判定 (core.veryl)

```
    // 正しい次のPC
    let mem_next_pc_expected: Addr = if !memq_rdata.br_taken {
        memq_rdata.pc_inc
    } else {
        memq_rdata.br_target
```

```
};

// 分岐ハザードが発生するかどうか
let mem_is_branch_hazard: logic = memq_rvalid & mem_next_pc_expected != memq_rdata.pc_next;
```

リスト A.15 は、分岐ハザードの判定を行っている部分のコードです。分岐の成否 (br_taken) によって、正しい次の PC(mem_next_pc_expected) を切り替えています。分岐が成立しない場合は PC をインクリメントした値 (pc_inc)、成立する場合は分岐先の PC(br_target) を mem_next_pc_expected に設定しています。分岐ハザードが発生するかどうか (mem_is_branch_target) は、正しい次の PC が投機的に選択された遷移先と一致しないかどうかによって判定されます。

ロードストア命令の処理

ロード命令, ストア命令は、memunit モジュールによって処理されます (リスト A.16)。

▼リスト A.16: memunit モジュールのインスタンス化 (core.veryl)

```
inst mem_memunit: memunit (
    clk                          ,
    rst                          ,
    dbus_if                      ,
    valid   : memq_rvalid        ,
    is_new  : mem_is_new         ,
    ctrl    : memq_rdata.ctrl    ,
    rs2     : memq_rdata.op      ,
    addr    : memq_rdata.alu_out,
    is_stall: mem_mem_stall      ,
    rdata   : mem_mem_rdata      ,
);
```

memunit モジュールは、core モジュールの dbus_if インターフェースを介してメモリにアクセスします。MEM ステージに供給された命令がロードストア命令の時、MEM ステージをストールし、状態を Init から WaitReady に移動します。状態が WaitReady のとき、dbus_if に対してロードかストアを要求します (リスト A.19)。 dbus_if が要求を受け付ける (dbus_if.ready が 1) とき、状態を WaitValid に移動します。状態が WaitReady のとき、ロードの結果が返ってくるかストアが完了した (dbus_if.resp_valid) ら状態を Init に移動し、MEM ステージのストールを終了します。MEM ステージの状態の遷移はリスト A.18 のように定義されています。

▼リスト A.17: memunit モジュールの状態の定義 (memunit.veryl)

```
enum State: logic<2> {
    Init,
    WaitReady,
    WaitValid,
}
```

▼リスト A.18: memunit の状態遷移 (memunit.veryl)

```
    always_ff (clk, rst) {
        if_reset {
            state = State::Init;
        } else {
            if valid {
                case state {
                    State::Init: if is_new & is_memcmd {
                                        state = State::WaitReady;
                                    }
                    State::WaitReady: if dbus_if.ready {
                                            state = State::WaitValid;
                                        }
                    State::WaitValid: if dbus_if.resp_valid {
                                            state = State::Init;
                                        }
                    default: {}
                }
            }
        }
    }
```

▼リスト A.19: dbus_if への要求 (memunit.veryl)

```
    always_comb {
        dbus_if.valid = state == State::WaitReady;
        dbus_if.addr  = req_mem_addr;
        dbus_if.wen   = req_mem_wen;
        dbus_if.wdata = req_mem_wdata;
        dbus_if.wmask = req_mem_wmask;
    }
```

`dbus_if.wen` は、ストアを要求するかどうかです。ストアのとき `dbus_if.wmask` のビットが `1` の部分について、メモリの値を `dbus_if.wdata` に置き換えます。ただし、 `dbus_if.wmask` は 1bit に 1byte(8bit) が対応していることに注意してください。

ロードストアの幅 (byte, half word, word) やロードの符号拡張については、`InstCtrl.funct3` で選択しています (表 A.4, 表 A.5)。

▼表 A.4: 符号拡張の選択

funct3[2]	符号拡張の有無
0	符号拡張しない (LBU, LHU)
1	符号拡張する (LB, LH, LW)

CSR を利用する命令の処理

bluecore では、CSR を利用する命令を MEM ステージで処理します。bluecore が対応している CSR を利用する命令は、CSRRW(I), CSRRS(I), CSRRC(I), ECALL, MRET の 8 命令です。レジスタは、mtvec, mepc, mcause の 3 つのみ対応しています。CSR を利用する命令の処理は、

▼表 A.5: 幅の選択

funct3[1:0]	幅
2'b00	byte (SB, LB, LBU)
2'b01	half word (SH, LH, LHU)
2'b10	word (SW, LW)

csrunit モジュールで行います (リスト A.20)。

▼リスト A.20: csrunit モジュールのインスタンス化 (core.veryl)

```
    inst mem_csrunit: csrunit (
        clk                                ,
        rst                                ,
        valid       : memq_rvalid          ,
        rdata       : mem_csr_rdata        ,
        raise_trap : mem_csr_is_trap       ,
        trap_vector: mem_csr_trap_vector   ,
        pc          : memq_rdata.addr      ,
        ctrl        : memq_rdata.ctrl      ,
        rd_addr     : memq_rdata.wbctx.addr,
        csr_addr    : memq_rdata.imm[16:5] ,
        rs1         : if inst_is_csr_imm(memq_rdata.ctrl) {
            // uimmを符号拡張する
            {memq_rdata.imm[4] repeat XLEN - 5, memq_rdata.imm[4:0]}
        } else {
            // opにrs1の値が格納されている
            memq_rdata.op
        },
    );
```

CSR の値を編集する命令は、命令の上位 12bit を CSR のアドレスとして扱います。この 12bit は即値の 17 から 6 ビットの範囲 (`imm[16:5]`) に格納されています。即値 (uimm) を符号拡張した値を利用する命令 (CSRR(W|S|C)I) で利用する即値は、即値の 5 から 1 ビットの範囲 (`imm[4:0]`) に格納されています。この 5bit は符号拡張されて csrunit モジュールに渡されます。

▼リスト A.21: 例外の判定 (csrunit.veryl)

```
    // ECALLのとき、例外を発生させる
    let raise_expt: logic = cmd_is_ecall;
    let cause_expt: UIntX = if cmd_is_ecall {
        CsrCause::ENVIRONMENT_CALL_FROM_U_MODE + {1'b0 repeat XLEN - $bits(mode), mode}
    } else {
        0
    };
```

RISC-V では、例外、または割り込みが発生するとき、PC と権限レベル (モード) を移動させます。このことをトラップ (trap) といいます。RISC-V には権限が高い順に M, S, U というモードが用意されており、それぞれ 3, 1, 0 という数字が対応しています (表 A.6)。

▼表 A.6: RISC-V の権限レベル

レベル	名前	名前の略記
0	User/Application	U
1	Supervisor	S
2	予約済み	
3	Machine	M

bluecore は、ECALL による S-mode から M-mode への例外のみサポートしています。リスト A.21 は、例外が発生するかどうかを判定している部分のコードです。MEM ステージにある命令が ECALL 命令のとき (`cmd_is_ecall`)、例外が発生する (`raise_expt`) ことにします。このとき、例外の発生原因 (`cause_expt`) を、 `CsrCause::ENVIRONMENT_CALL_FROM_U_MODE` に現在の実行環境の権限レベル (`mode`) を足したものに設定します。RISC-V では、U-mode, S-mode, M-mode からの ECALL による例外であることを示す数値を、それぞれ 8, 9, 11 と定義しています。そのため、ECALL による例外の理由を示す値は、U-mode からの例外であることを示す 8 に権限レベルの数値を足したものと等しくなります。

▼表 A.7: RISC-V の例外の原因を示す数値

数値	例外の種類
8	Environment call from U-mode
9	Environment call from S-mode
10	予約済み
11	Environment call from M-mode

bluecore は、RISC-V のトラップに関する仕様の一部を実装しています。csrunit モジュールは例外が発生するとき、次のように動作します。

- モードを M-mode に設定する
- mcause レジスタに、例外の発生原因を格納する
- mepc レジスタに、例外が発生した命令の PC を格納する
- PC の遷移先として mtvec の値を output する

MRET 命令が実行されるときは、次のように動作します。

- モードを S-mode に設定する
- PC の遷移先として mepc の値を output する

CPU の起動 (リセット) 時の権限レベルは M-mode です。これを MRET 命令を実行することで S-mode に移動し、ECALL 命令を実行することで M-mode に戻ることができます。MRET 命令, ECALL 命令の動作は、リスト A.22 のように実装されています。

▼リスト A.22: CSR 命令の処理 (csrunit.veryl)

```
    always_ff (clk, rst) {
        if_reset {
            mode   = CsrMode::M;
            mcause = 0;
            mepc   = 0;
            mtvec  = 0;
        } else if valid {
            if raise_expt {
                mode   = CsrMode::M;
                mcause = cause_expt;
                mepc   = pc;
            } else {
                if cmd_is_mret {
                    mode = CsrMode::S;
                } else if cmd_is_write {
                    case csr_addr {
                        CsrAddr::MTVEC : mtvec  = wdata;
                        CsrAddr::MEPC  : mepc   = {wdata[XLEN - 1:2], 2'b00};
                        CsrAddr::MCAUSE: mcause = wdata;
                        default        : {}
                    }
                }
            }
        }
    }
```

MEM ステージでライトバックする値が確定する命令

MEM ステージでは、ロード命令, CSR 命令によるライトバックする値が確定し、すべての命令のライトバックする値が確定します。リスト A.23 は、MEM ステージでライトバックする値を確定させる処理です。 `wbctx.valid` , `wbctx.data` は、MEM ステージよりも前で値が確定していた場合には値をそのまま引き継いでいます。命令がロード命令のとき、memunit モジュールの出力値 (`mem_mem_rdata`) を `wbctx.data` に設定します。命令が CSR 命令のとき、csrunit モジュールの出力値 (`mem_csr_rdata`) を `wbctx.data` に設定します。

▼リスト A.23: MEM ステージでライトバックする値を確定させる処理 (core.veryl)

```
        // メモリ命令とCSR命令のライトバックする値を確定する
        wbq_wdata.wbctx.valid = 1;
        wbq_wdata.wbctx.addr  = memq_rdata.wbctx.addr;
        wbq_wdata.wbctx.data  = if memq_rdata.wbctx.valid {
            memq_rdata.wbctx.data
        } else if inst_is_memory_op(memq_rdata.ctrl) {
            mem_mem_rdata
        } else {
            mem_csr_rdata
        };
```

A.4.5　WB ステージ (WriteBack Stage)

WB ステージは、MEM ステージから制御用の値とライトバックする値を受け取って、ライトバックする場合はレジスタに値をライトバックするステージです。WB ステージはパイプライン処理の最後のステージであり、WB ステージで命令の実行は終了します。

リスト A.24 は、レジスタに命令の結果を書き込む処理です。レジスタに値を書き込む命令である (`InstCtrl.rf_wen`) 場合、レジスタ (`registers`) に命令の結果 (`wbctx.data`) を書き込みます。

▼リスト A.24: レジスタへのライトバック (core.veryl)

```
            if wbq_rvalid & wbq_rdata.ctrl.rf_wen {
                registers[wbq_rdata.wbctx.addr] = wbq_rdata.wbctx.data;
            }
```

A.5　命令実行の具体例

A.5.1　ADD 命令の実行

A.5.2　BGE 命令の実行

A.5.3　LH 命令の実行

A.5.4　CSRRW 命令の実行

A.6　合成

FPGA に合成する

付録 B
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。

B.1 riscv-tests の実行

B.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 B.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 B.2)。

▼リスト B.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 B.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 B.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 B.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 B.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 B.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 B.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。

B.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 B.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 B.7 です。命令が一致しているのを確認できます。

▼リスト B.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 B.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 B.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 B.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 B.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 B.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
```

B.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 B.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 B.11) とテストへジャンプするためのコード (リスト B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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>
```

B.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 B.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。

B.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 B.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 B.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 B.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$sv キーワードを利用しないで利用するために、リスト B.21 のように config パッケージで$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 B.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 B.23: 新しく記述する変数 (riscv-tests.py)

```
TEST_EXIT_ADDR = "\\'h1000"
TEST_WDATA_SUCCESS = "1"
```

▼リスト B.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 B.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 B.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 B.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。

B.2　プログラムを作成して実行する

B.3　ログの整備

ログの整備

B.4　Spike との比較

B.5　メモリ操作の追跡

B.6　ストールの検知

B.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
