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

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 RISC-V の CPU を、新しめのハードウェア記述言語である Veryl で記述する方法について解説した本です。本書は無料で、pdf 版は https://github.com/nananapo/veryl-riscv-book で入手することができます。

本書の対象読者

本書はコンピュータアーキテクチャに興味があり、何らかのプログラミング言語を習得している人を対象としています。

前提とする知識

未定

問い合わせ先

本書に関する質問やお問い合わせは、以下のリポジトリに issue を立てて行ってください。

- URL: https://github.com/nananapo/veryl-riscv-book/issues

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

TODO 大幅に書き換える

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

第Ⅰ部

基本編

第 1 章

環境構築

1.1 Veryl

rustup cargo vscode の拡張

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 1.1: 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
```

▼リスト 1.2: veryl がインストールされているかの確認

```
$ veryl --version
veryl 0.12.0
```

1.2 Verilator

インストールするだけ

1.3 riscv-gnu-toolchain

clone

第 2 章
ハードウェア記述言語 Veryl

2.1 あああ

あああ
パッケージパラメータ使い方

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。基本整数命令という名前の通り、整数の足し引きやビット演算、ジャンプ、分岐命令などの最小限の命令しか実装されていません。また、32 ビット幅の汎用レジスタが 32 個定義されています。ただし、0 番目のレジスタの値は常に 0 です。RISC-V は基本整数命令セットに新しい命令を拡張として実装します。複雑な機能を持つ CPU を実装する前に、まずは最小の機能を持つ CPU を実装しましょう。

3.1 CPU は何をやっているのか？

上に書かれている文章の意味が分からなくても大丈夫。詳しく説明します。

CPU を実装するには何が必要でしょうか？ まずは CPU がどのような動作をするかについて考えてみます。一般的に、汎用のプログラムを実行する CPU は次の手順でプログラムを実行していきます。

1. メモリからプログラムを読み込む
2. プログラムを実行する
3. 1, 2 の繰り返し

ここで、メモリから読み込まれる「プログラム」とは一体何を示しているのでしょうか？ 普通のプログラマが書くのは C 言語や Rust などのプログラミング言語のプログラムですが、通常の CPU はそれをそのまま解釈して実行することはできません。そのため、メモリから読み込まれる「プログラム」とは、CPU が読み込んで実行することができる形式のプログラムです。これはよく「機械語」と呼ばれ、0 と 1 で表される 2 進数のビット列で記述されています。

メモリからプログラムを読み込んで実行するのが CPU の仕事ということが分かりました。これをもう少し掘り下げます。

まず、プログラムをメモリから読み込むためには、メモリのどこを読み込みたいのかという情報 (アドレス) をメモリに与える必要があります。また、当然ながらメモリが必要です。

CPU はプログラムを実行しますが、一気にすべてのプログラムを読み込んだり実行するわけで

はなく、プログラムの最小単位である「命令」を一つずつ読み込んで実行します。命令をメモリに要求、取得することを、命令をフェッチするといいます。

命令が CPU に供給されると、CPU は命令のビット列がどのような意味を持っていて何をすればいいかを判定します。このことを、命令をデコードするといいます。

命令をデコードすると、いよいよ計算やメモリアクセスを行います。しかし、例えば足し算を計算するにも何と何を足し合わせればいいのか分かりません。この計算に使うデータは、次のように指定されます。

- レジスタ (= CPU に存在する小さなメモリ) の番号
- 即値 (= 命令のビット列から生成される数値)

計算対象のデータにレジスタと即値のどれを使うかは命令によって異なります。レジスタの番号は命令のビット列の中に含まれています。

計算やメモリアクセスが終わると、その結果をレジスタに格納します。例えば足し算を行う命令なら足し算の結果が、メモリから値を読み込む命令なら読み込まれた値が格納されます。

これで命令の実行は終わりですが、CPU は次の命令を実行する必要があります。今現在実行している命令のアドレスを格納しているメモリのことをプログラムカウンタ (PC) と言い、CPU は PC の値をメモリに渡すことで命令をフェッチしています。CPU は次の命令を実行するために、PC の値を次の命令のアドレスに設定します。ジャンプ命令の場合は、PC の値をジャンプ先のアドレスに設定します。分岐命令の場合は、分岐の成否を計算で判定し、分岐が成立する場合は分岐先のアドレスを PC に設定します。分岐が成立しない場合は、通常の命令と同じように次の命令のアドレスを PC に設定します。

ここまでの話をまとめると、CPU の動作は次のようになります。

- PC に格納されたアドレスにある命令をフェッチする
- 命令を取得したらデコードする
- 計算で使用するデータを取得する (レジスタの値を取得したり、即値を生成する)
- 計算する命令の場合、計算を行う
- メモリにアクセスする命令の場合、メモリ操作を行う
- 計算やメモリアクセスの結果をレジスタに格納する
- PC の値を次に実行する命令に設定する

CPU が何をするものなのかが分かりましたか？ 実装を始めましょう。

3.2　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。ここでは適当に core という名前にしています。

▼リスト 3.1: 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 が作成されます。

TODO ソースマップがいらないので消す

▼リスト 3.2: 作成された Veryl.toml

```
[project]
name = "core"
version = "0.1.0"
```

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

```
$ cd core
$ mkdir src
```

3.3　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記述するパッケージを作成します。

`src/eei.veryl` を作成し、次のように記述します。

▼リスト 3.3: eei.veryl

```
package eei {
    const XLEN: u32 = 32;
    const ILEN: u32 = 32;

    type UIntX  = logic<XLEN>;
    type UInt32 = logic<32>  ;
    type UInt64 = logic<64>  ;
    type Inst   = logic<ILEN>;
    type Addr   = logic<XLEN>;
}
```

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。

eei パッケージには、次のパラメータを定義します。

XLEN

XLEN は、RISC-V において整数レジスタの長さを示す数字として定義されています。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。

ILEN

ILEN は、RISC-V において CPU の実装がサポートする命令の最大の幅を示す値として定義されています。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 にしています。

また、何度も使用することになる型に別名を付けています。

UIntX, UInt32, UInt64

幅がそれぞれ XLEN, 32, 64 の符号なし整数型

Inst

命令のビット列を格納するための型

Addr

メモリのアドレスを格納するための型。RISC-V で使用できるメモリ空間の幅は XLEN なので UIntX でもいいですが、アドレスであることを明示するために別名を定義しています。

3.4　メモリ

CPU はメモリに格納された命令を実行します。よって、CPU の実装のためにはメモリの実装が必要です。RV32I において命令の幅は 32 ビットです。また、メモリからのロード命令、ストア命令の最大の幅も 32 ビットです。

これを実現するために、次のような要件のメモリを実装します。

- 読み書きの単位は 32 ビット
- クロックに同期してメモリアクセスの要求を受け取る
- 要求を受け取った次のクロックで結果を返す

3.4.1　メモリのインターフェースの定義

このメモリモジュールには、クロックとリセット信号の他に 7 個のポートを定義する必要があります (表 3.1)。これを一つ一つ定義、接続するのは面倒なため、次のような interface を定義します。

`src/membus_if.veryl` を作成し、次のように記述します。

▼リスト 3.4: インターフェースの定義 (membus_if.veryl)

```
import eei::*;

interface membus_if {
    type DataType = UInt32;

    var valid : logic   ;
    var ready : logic   ;
```

```
    var addr  : Addr    ;
    var wen   : logic   ;
    var wdata : DataType;
    var rvalid: logic   ;
    var rdata : DataType;

    modport master {
        valid : output,
        ready : input ,
        addr  : output,
        wen   : output,
        wdata : output,
        rvalid: input ,
        rdata : input ,
    }

    modport slave {
        valid : input ,
        ready : output,
        addr  : input ,
        wen   : input ,
        wdata : input ,
        rvalid: output,
        rdata : output,
    }
}
```

▼表 3.1: メモリモジュールに必要なポート

ポート名	型	向き	意味
clk	clock	input	クロック信号
rst	reset	input	リセット信号
valid	logic	input	メモリアクセスを要求しているかどうか
ready	logic	output	メモリアクセスを受容するかどうか
addr	Addr	input	アクセスするアドレス
wen	logic	input	書き込みかどうか (1 なら書き込み)
wdata	UInt32	input	書き込むデータ
rvalid	logic	output	受容した要求の処理が終了したかどうか
rdata	UInt32	output	受容した読み込み命令の結果

interface を利用することで、レジスタやワイヤの定義が不要になり、さらにポートの相互接続を簡潔にすることができます。

3.4.2　メモリの実装

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 3.5: memory.veryl

```
import eei::*;

module memory #(
    param MEMORY_WIDTH: u32    = 20, // メモリのサイズ
    param FILE_PATH   : string = "" // メモリの初期値が格納されたファイルのパス
    ,
) (
    clk   : input   clock            ,
    rst   : input   reset            ,
    membus: modport membus_if::slave,
) {

    var mem: membus_if::DataType [2 ** MEMORY_WIDTH];

    // Addrをmemのインデックスに変換する関数
    function addr_to_memaddr (
        addr: input Addr                ,
    ) -> logic<MEMORY_WIDTH> {
        return addr[MEMORY_WIDTH - 1 + 2:2];
    }

    initial {
        // memをFILE_PATHに格納されているデータで初期化
        if FILE_PATH != "" {
            $readmemh(FILE_PATH, mem);
        }
    }

    always_comb {
        membus.ready = 1;
    }

    always_ff {
        membus.rvalid = membus.valid;
        membus.rdata  = mem[addr_to_memaddr(membus.addr)];
        if membus.valid && membus.wen {
            mem[addr_to_memaddr(membus.addr)] = membus.wdata;
        }
    }
}
```

memory モジュールには次のパラメータが定義されています。

MEMORY_WIDTH

メモリのサイズを指定するためのパラメータです。メモリのサイズは 32 ビット * (2 ** MEMORY_WIDTH) になります。

FILE_PATH

メモリの初期値が格納されたファイルのパスです。指定しない場合は""になり初期化されません。初期化は$readmemh システムタスクで行います。

読み込み、書き込み時の動作は次の通りです。

読み込み

読み込みが要求されるとき、`membus.valid` が `1` 、`membus.wen` が `0` 、`membus.addr` が対象アドレスになっています。次のクロックで、`membus.rvalid` が `1` になり、`membus.rdata` はメモリのデータになります。

書き込み

読み込みが要求されるとき、`membus.valid` が `1` 、`membus.wen` が `1` 、`membus.addr` が対象アドレスになっています。`always_ff` ブロックでは、`membus.wen` が `1` であることを確認し、`1` の場合は対象アドレスに `membus.wdata` を書き込みます。次のクロックで `membus.rvalid` が `1` になります。

Addr 型では 1 バイト単位でアドレスを指定しますが、mem レジスタは 32 ビット (=4 バイト) 単位でデータを整列しています。そのため、Addr 型のアドレスをそのまま mem レジスタのインデックスとして利用することはできません。`addr_to_memaddr` 関数は、1 バイト単位のアドレスの下位 2 ビットを切り詰めることによって、mem レジスタにおけるインデックスに変換しています。

3.5　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 3.6: top.veryl

```
import eei::*;

module top #(
    param MEM_FILE_PATH: string = "",
) (
    clk: input clock,
    rst: input reset,
) {
    inst membus: membus_if;

    inst mem: memory #(
        FILE_PATH: MEM_FILE_PATH,
    ) (
        clk     ,
        rst     ,
        membus  ,
    );
}
```

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。また、memory モジュールのポートに接続するための membus_if インターフェースもインスタンス化しています。

3.6　命令フェッチ

メモリを作成したため、命令フェッチ処理を作る準備が整いました。いよいよ CPU のメイン部分を作成していきます。

3.6.1　命令フェッチの実装

`src/core.veryl` を作成し、次のように記述します。

▼リスト 3.7: core.veryl

```
import eei::*;

module core (
    clk   : input   clock             ,
    rst   : input   reset             ,
    membus: modport membus_if::master,
) {

    var if_pc           : Addr ;
    var if_is_requested: logic; // フェッチ中かどうか
    var if_pc_requested: Addr ; // 要求したアドレス

    let if_pc_next: Addr = if_pc + 4;

    // 命令フェッチ処理
    always_comb {
        membus.valid = 1;
        membus.addr  = if_pc;
        membus.wen   = 0;
        membus.wdata = 'x; // wdataは使用しない
    }

    always_ff {
        if_reset {
            if_pc            = 0;
            if_is_requested = 0;
            if_pc_requested = 0;
        } else {
            if if_is_requested {
                if membus.rvalid {
                    if_is_requested = membus.ready;
                    if membus.ready {
                        if_pc            = if_pc_next;
                        if_pc_requested = if_pc;
                    }
                }
            } else {
                if membus.ready {
                    if_is_requested = 1;
                    if_pc            = if_pc_next;
```

```
                    if_pc_requested = if_pc;
                }
            }
        }
    }

    always_ff {
        if if_is_requested && membus.rvalid {
            $display("%h : %h", if_pc_requested, membus.rdata);
        }
    }
}
```

`if_pc` レジスタは PC(プログラムカウンタ) です。`if_is_requested` で現在フェッチ中かどうかを管理しており、フェッチ中のアドレスを `if_pc_requested` に格納しています。

`always_comb` ブロックでは、常にメモリにアドレス `if_pc` にある命令を要求しています。命令フェッチではメモリの読み込みしか行わないため、`membus.wen` は `0` になっています。

上から 1 つめの `always_ff` ブロックでは、フェッチ中かどうか、メモリは ready(要求を受け入れる) 状態かどうかによって、`if_pc`, `if_is_requested`, `if_pc_requested` の値を変更しています。メモリに新しくフェッチを要求する時、`if_pc` を次の命令のアドレス (`4` を足したアドレス) に、`if_is_requested` を `1` に変更しています。フェッチ中かつ `membus.rvalid` が `1` のときは命令フェッチが完了しています。その場合は、メモリが ready ならすぐに次の命令フェッチを開始します。

これにより、0,4,8,c,10,... という順番で次々に命令をフェッチするようになっています。

上から 2 つめの `always_ff` ブロックはデバッグ用のプログラムです。命令フェッチが完了したときにその結果を `$display` システムタスクによって出力します。

次に、top モジュールで core モジュールをインスタンス化し、membus_if 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。

▼リスト 3.8: top.veryl 内で core モジュールをインスタンス化する

```
    inst c: core (
        clk     ,
        rst     ,
        membus  ,
    );
```

3.6.2　フェッチのテスト

ここまでのプログラムが正しく動くかを検証します。

Veryl で記述されたプログラムは `veryl build` コマンドで SystemVerilog のプログラムに変換することができます。変換されたプログラムをオープンソースの Verilog シミュレータである Verilator で実行することで、命令フェッチが正しく動いていることを確認します。

まず、プログラムをビルドします。

▼リスト 3.9: Veryl プログラムのビルド

```
$ veryl fmt ←フォーマットする
$ veryl build ←ビルドする
```

上記のコマンドを実行すると、veryl プログラムと同名の `.sv` ファイルと `core.f` ファイルが生成されます。 `core.f` は生成された SystemVerilog のプログラムファイルのリストです。これをシミュレータのビルドに利用します。

Verilator でのシミュレーションの実行のために C++ プログラムを作成します。 `src/test_verilator.cpp` を作成し、次のように記述します。

▼リスト 3.10: test_verilator.cpp

```
にゃ
```

この C++ プログラムでは、top モジュール (プログラム中では Vtop クラス) をインスタンス化し、そのクロックを反転して実行するのを繰り返しています。

利用できるパラメータは次の通りです。

-time, -t

何クロックで実行を終了するか。0 のときは終了しない。デフォルト値は 0。

-memory, -m

メモリの初期値のファイルへのパス。デフォルト値は""。

`verilator` コマンドを実行し、シミュレータをビルドします。

▼リスト 3.11: シミュレータのビルド

```
$ verilator にゃ
```

上記のコマンドの実行により、シミュレータが `にゃ` に生成されました。シミュレータを実行する前にメモリの初期値となるファイルを作成します。 `src/sample.hex` を作成し、次のように記述します。

▼リスト 3.12: sample.hex

```
01234567
89abcdef
deadbeef
cafebebe
```

値は 16 進数で 4 バイトずつ記述されています。シミュレーションを実行すると、このファイルは memory モジュールの `readmemh` システムタスクによって読み込まれます。それにより、メモリは次のように初期化されます。

生成されたシミュレータを実行し、0, 4, 8, c のデータが正しくフェッチされていることを確認します。

▼表 3.2: sample.hex によって設定されるメモリの初期値

アドレス	値
00000000	01234567
00000004	89abcdef
00000008	deadbeef
0000000c	cafebebe
00000010~	不定

▼リスト 3.13: 命令フェッチの動作チェック

```
$ Vtop -t 6 -m sample.hex
実行結果
```

メモリファイルのデータが 4 バイトずつ読み込まれていることが確認できます。

ビルド、シミュレータのビルドのために一々コマンドを打つのは面倒です。これらの作業を一つのコマンドで済ますために、`Makefile` を作成し、次のように記述します。

▼リスト 3.14: Makefile

```
```

これ以降、次のようにビルドやシミュレータのビルドができるようになります。

▼リスト 3.15: Makefile によって追加されたコマンド

```
$ make build ←ビルド
$ make sim ←シミュレータのビルド
$ make clean ←ビルドした成果物の削除
```

3.7　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。

3.8　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。

3.9　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。

3.10　ロード、ストア命令

3.10.1　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

3.10.2　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

3.11　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。

3.11.1　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

3.11.2　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。

3.12　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。

3.12.1　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

3.12.2　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。

3.13　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。

3.13.1　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

3.13.2　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

3.13.3　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

前章では RV64I を実装しました。RV64I は 64 ビットの基本整数命令セットであり、基本的な演算しか実装されていません。M 拡張はこれにかけ算と割り算の命令を実装します。

M 拡張には、かけ算をおこなう MUL 命令、割り算をおこなう DIV 命令、剰余を求める REM 命令があります。これらの計算は Veryl に用意されている*, /, % 演算子で実装することができますが、これによって自動で実装される回路は 1 クロックで計算を完了させる非常に大きなものになってしまい、CPU の最大周波数を大幅に低下させてしまいます。これを回避するために、複数クロックでゆっくり計算を行うモジュールを作成します。

5.1 MUL[W] 命令

5.2 MULH 命令

5.3 MULHU 命令

5.4 MULHSU 命令

5.5 DIV[W] 命令

引き放し法でやる

5.6 DIVU[W] 命令

5.7 REM[W] 命令

5.8 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

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

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

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