

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

本書を手にとっていただき、ありがとうございます。

本書は、OS を実行できる程度の機能を持った 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!");
```

←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!");
```

←取り消し線は削除したコード

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

←太字は追加したコード

長い行が右端で折り返されると、折り返されたことを表す小さな記号がつきます。

```
123456789_123456789_123456789_123456789_123456789_123456789_123456789_123456789_
```

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

```
$ echo Hello
```

←行頭の「\$」はプロンプト、それ以降がユーザ入力

本文に対する補足情報や注意・警告は、次のようなノートや囲み枠で表示します。

.....

ノートタイトル

ノートは本文に対する補足情報です。

.....

**タイトル**
本文に対する補足情報です。

**タイトル**
本文に対する注意・警告です。

Intro

WIP

目次

まえがき / はじめに	i
Intro	iii
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
第 2 章 ハードウェア記述言語 Veryl	3
第 3 章 RV32I の実装	4
3.1 CPU は何をやっているのか?	4
3.2 プロジェクトの作成	5
3.3 定数の定義	6
3.4 メモリ	7
3.4.1 メモリのインターフェースの定義	7
3.4.2 メモリの実装	8
3.5 top モジュールの作成	10
3.6 命令フェッチ	11
3.6.1 命令フェッチの実装	11
3.6.2 命令フェッチのテスト	12
3.6.3 フェッチした命令を FIFO に格納する	17
3.7 命令のデコードと即値の生成	20
3.7.1 定数と型の定義	22
3.7.2 デコードと即値の生成	23
3.7.3 デコーダのインスタンス化	25
3.8 レジスタの定義と読み込み	26
3.9 ALU を作り、計算する	28
3.9.1 ALU の作成	28
3.9.2 ALU のテスト	31
3.10 レジスタに結果を書き込む	32
3.10.1 ライトバックの実装	32

3.10.2	ライトバックのテスト	33
3.11	ロード命令とストア命令の実装	34
3.11.1	LW, SW 命令の実装	35
3.11.2	LB, LBU, LH, LHU 命令の実装	43
3.11.3	SB, SH 命令の実装	44
3.11.4	LB, LBU, LH, LHU, SB, SH 命令のテスト	48
3.12	分岐, ジャンプ	48
3.12.1	JAL, JALR 命令	48
3.12.2	分岐命令	49
3.13	riscv-tests でテストする	49
3.13.1	最小限の CSR 命令の実装	49
3.13.2	終了検知	49
3.13.3	テストの実行	49
第 4 章	RV64I の実装	51
4.1	メモリの幅を広げる	51
4.2	LW, LWU, LD 命令の実装	52
4.3	SD 命令の実装	52
4.4	LUI, AUIPC 命令の実装	52
4.5	ADDW, ADDIW, SUBW 命令の実装	52
4.6	シフト命令の実装	52
4.7	riscv-tests	52
第 II 部	基本的な拡張とトラップの実装	53
第 5 章	M 拡張の実装	54
5.1	MUL[W] 命令	54
5.2	MULH 命令	54
5.3	MULHU 命令	54
5.4	MULHSU 命令	54
5.5	DIV[W] 命令	55
5.6	DIVU[W] 命令	55
5.7	REM[W] 命令	55
5.8	REMU[W] 命令	55
第 6 章	例外の実装	56
6.1	例外とは何か?	56
6.2	illegal instruction	56
6.3	メモリのアドレスのやつ	56

第 7 章	A 拡張の実装	57
7.1	概要	57
7.2	AMO 系	57
7.3	LR / SC	57
7.4	例外	57
第 8 章	C 拡張の実装	58
8.1	概要	58
8.2	実装方針	58
8.3	圧縮命令の変換	58
第 9 章	MMIO の実装	59
9.1	概要	59
9.2	実装方針	59
第 10 章	割り込みの実装	60
10.1	概要	60
10.2	UART RX	60
10.3	タイマ割り込み	60
第 III 部	privilege mode の実装	61
第 11 章	M-mode の実装	62
第 12 章	S-mode の実装	63
第 13 章	ページングの実装	64
13.1	ページングとは何か	64
13.2	PTW の実装	64
13.3	Sv32	64
13.4	Sv39	64
13.5	Sv48	64
13.6	Sv54	64
第 IV 部	OS を動かす	65

第 14 章 virtio の実装	66
第 15 章 xv6 の実行	67
あとがき / おわりに	68

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

TODO

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。

計算を実行するユニット (部品) のことを、ALU(Arithmetic Logic Unit) といいます。

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



```

var addr : Addr ;
var wen  : logic ;
var wdata : UInt32;
var rvalid: logic ;
var rdata : UInt32;

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
) (
    clk      : input  clock          ,
    rst      : input  reset          ,
    membus   : modport membus_if::slave,
    FILE_PATH: input  string          , // メモリの初期値が格納されたファイルのパス
) {

    var mem: UInt32 [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 $32 \text{ ビット} * (2 ** \text{MEMORY_WIDTH})$ 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。初期化は\$readmemh システムタスクで行います。(ポートとして定義していますが、本書ではパラメータとして扱います。)

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 (
    clk          : input clock ,
    rst          : input reset ,
    MEM_FILE_PATH: input string,
) {
    inst membus: membus_if;

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

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。ここで `if_` という prefix は instruction fetch の略です。 `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,...` という順番のアドレスの命令を次々にフェッチするようになっていきます。

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 命令フェッチのテスト

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。

シミュレータのビルドには Verilator を利用します。Verilator は与えられた SystemVerilog プログラムを C++ プログラムに変換することでシミュレータを生成します。verilator を利用するために、次のような C++ プログラムを書く必要があります。

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

▼リスト 3.10: tb_verilator.cpp

```
#include <iostream>
#include <filesystem>
#include <verilated.h>
#include "Vcore_top.h"

namespace fs = std::filesystem;

int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);

    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " MEMORY_FILE_PATH [CYCLE]" << std::endl;
        return 1;
    }

    // メモリの初期値を格納しているファイル名
    std::string memory_file_path = argv[1];
    try {
        // 絶対パスに変換する
        fs::path absolutePath = fs::absolute(memory_file_path);
        memory_file_path = absolutePath.string();
    } catch (const std::exception& e) {
        std::cerr << "Invalid memory file path : " << e.what() << std::endl;
        return 1;
    }

    // シミュレーションを実行するクロックサイクル数
    unsigned long long cycles = 0;
    if (argc >= 3) {
        std::string cycles_string = argv[2];
        try {
            cycles = stoull(cycles_string);
        } catch (const std::exception& e) {
```

```

        std::cerr << "Invalid number: " << argv[2] << std::endl;
        return 1;
    }
}

Vcore_top *dut = new Vcore_top();
dut->MEM_FILE_PATH = memory_file_path;

// reset
dut->clk = 0;
dut->rst = 1;
dut->eval();
dut->rst = 0;
dut->eval();

// loop
dut->rst = 1;
for (long long i=0; cycles == 0 || i / 2 < cycles; i++) {
    dut->clk = !dut->clk;
    dut->eval();
}

dut->final();
}

```

このC++プログラムはtopモジュール（プログラム中ではVtop_core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。

このプログラムはコマンドライン引数として次の2つの値を受け取ります。

MEMORY_FILE_PATH

メモリの初期値のファイルへのパス。実行時にtopモジュールのMEM_FILE_PATHパラメータに渡されます。

CYCLE

何クロックで実行を終了するかを表す値。0のときは終了しません。デフォルト値は0です。

Verilatorによるシミュレーションは、トップモジュールのクロック信号を変更してeval関数を呼び出すことにより実行します。プログラムではclkを反転させてevalするループの前にtopモジュールをリセットする必要があるため、topモジュールのrstを1にしてevalを実行し、rstを0にしてまたevalを実行し、rstを1にもどしてからclkを反転しています。

シミュレータのビルド

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 --cc -f core.f --exe src/tb_verilator.cpp --top-module top --Mdir obj_dir
$ make -C obj_dir -f Vcore_top.mk ←シミュレータをビルドする

```

```
$ mv obj_dir/Vcore_top obj_dir/sim ←シミュレータの名前をsimに変更する
```

`verilator --cc` コマンドに次のコマンドライン引数を渡して実行することで、シミュレータを生成するためのプログラムが `obj_dir` に生成されます。

-f

SystemVerilog プログラムのファイルリストを指定します。今回は `core.f` を指定しています。

--exe

実行可能なシミュレータの生成に使用する、main 関数が含まれた C++ プログラムを指定します。今回は `src/tb_verilator.cpp` を指定しています。

--top-module

トップモジュールを指定します。今回は `top` モジュールを指定しています。

--Mdir

成果物の生成先を指定します。今回は `obj_dir` フォルダに指定しています。

上記のコマンドの実行により、シミュレータが `obj_dir/sim` に生成されました。

メモリの初期化用ファイルの作成

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
←必ず末尾に改行をいれてください
```

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みます。それにより、メモリは次のように初期化されます。

▼ 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

シミュレータの実行

生成されたシミュレータを実行し、アドレスが 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。

▼ リスト 3.13: 命令フェッチの動作チェック

```
$ obj_dir/sim src/sample.hex 4
00000000 : 01234567
00000004 : 89abcdef
00000008 : deadbeef
0000000c : cafebebe
```

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。

Makefile の作成

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

```
PROJECT = core
FILELIST = $(PROJECT).f

TOP_MODULE = top
TB_PROGRAM = src/tb_verilator.cpp
OBJ_DIR = obj_dir/
SIM_NAME = sim

build:
    veryl fmt
    veryl build

clean:
    veryl clean
    rm -f src/*.sv.map
    rm -rf $(OBJ_DIR)

sim:
    verilator --cc -f $(FILELIST) --exe $(TB_PROGRAM) --top-module $(PROJECT)_$(TOP_MODULE) >
--Mdir $(OBJ_DIR)
    make -C $(OBJ_DIR) -f V$(PROJECT)_$(TOP_MODULE).mk
    mv $(OBJ_DIR)/V$(PROJECT)_$(TOP_MODULE) $(OBJ_DIR)/$(SIM_NAME)
```

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 ← Verylプログラムのビルド
$ 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
```

3.6.3 フェッチした命令を FIFO に格納する

FIFO の作成

フェッチした命令は次々に実行されますが、その命令が何クロックで実行されるかは分かりません。命令が常に1クロックで実行される場合は現状の常にフェッチし続けるようなコードで問題ありませんが、例えばメモリにアクセスする命令は実行に何クロックかかるか分からないため、フェッチされた次の命令を保持しておくバッファを用意しておく必要があります。

そこで、FIFO を作成して、フェッチした命令を格納します。 `src/fifo.veryl` を作成し、次のように記述します。

▼ リスト 3.16: fifo.veryl

```
module fifo #(
    param DATA_TYPE: type = logic,
    param WIDTH      : u32 = 2 ,
) (
    clk : input  clock ,
    rst : input  reset ,
    wready: output logic ,
    wvalid: input  logic ,
    wdata : input  DATA_TYPE,
    rready: input  logic ,
    rvalid: output logic ,
    rdata : output DATA_TYPE,
) {
    type Ptr = logic<WIDTH>;

    var mem : DATA_TYPE [2 ** WIDTH];
    var head: Ptr ;
    var tail: Ptr ;

    let tail_plus1: Ptr = tail + 1;

    always_comb {
        rvalid = head != tail;
        rdata  = mem[head];
        wready = tail_plus1 != head;
    }

    always_ff {
        if wready && wvalid {
            mem[tail] = wdata;
            tail      = tail + 1;
        }
        if rready && rvalid {
            head = head + 1;
        }
    }
}
```

fifo モジュールは、 `DATA_TYPE` 型のデータを `2 ** WIDTH - 1` 個格納することができる FIFO で

す。操作は次のように行います。

データを追加する

`wready` が 1 のとき、データを追加することができます。データを追加するためには、追加したいデータを `wdata` に格納し、`wvalid` を 1 にします。追加したデータは次のクロック以降に取り出すことができます。

データを取り出す

`rready` が 1 のとき、データを取り出すことができます。データを取り出すことができるとき、`rdata` にデータが出力されています。`rvalid` を 1 にすることで、FIFO にデータを取り出したことを通知することができます。

`head` レジスタと `tail` レジスタによってデータの格納状況を管理しています。データを書き込むとき、つまり `wready && wvalid` のとき、`tail = tail + 1` しています。データを取り出すとき、つまり `rready && rvalid` のとき、`head = head + 1` しています。

データを書き込める状況とは、`tail` に 1 を足しても `head` を超えない、つまり、`tail` が指す場所が一周してしまわないときです。この制限から、FIFO には最大でも $2 \times \text{WIDTH} - 1$ 個しかデータを格納することができません。データを取り出せる状況とは、`head` と `tail` の指す場所が違うときです。

命令フェッチ処理の変更

fifo モジュールを使って、次のように命令フェッチ処理を変更します。

まず、fifo モジュールをインスタンス化します。

▼ リスト 3.17: fifo モジュールのインスタンス化

```
// ifFIFOのデータ型
struct if_fifo_type {
    addr: Addr,
    bits: Inst,
}

// FIFOの制御用レジスタ
var if_fifo_wready: logic      ;
var if_fifo_wvalid: logic     ;
var if_fifo_wdata : if_fifo_type;
var if_fifo_rready: logic      ;
var if_fifo_rvalid: logic     ;
var if_fifo_rdata : if_fifo_type;

// フェッチした命令を格納するFIFO
inst if_fifo: fifo #(
    DATA_TYPE: if_fifo_type,
    WIDTH      : 3
) (
    clk
    ,
    rst
    ,
    wready: if_fifo_wready,
```

```

wvalid: if_fifo_wvalid,
wdata : if_fifo_wdata ,
rready: if_fifo_rready,
rvalid: if_fifo_rvalid,
rdata : if_fifo_rdata ,
);

```

まず、FIFO に入れるデータの型として `if_fifo_type` という構造体を定義します。`if_fifo_type` には、命令のアドレス (`addr`) と命令のビット列 (`bits`) を格納するためのメンバーが含まれています。

次に、fifo モジュールとデータの受け渡しをするための変数を定義し、fifo モジュールを `if_fifo` という名前でインスタンス化しています。`DATA_TYPE` パラメータに `if_fifo_type` を渡すことでアドレスと命令のペアを格納することができるようにし、`WIDTH` に 3 と指定することで、サイズを $2 \times 3 - 1 = 7$ にしています。このサイズは適当です。

fifo モジュールを用意したので、メモリヘフェッチ指令を送る処理を変更します。

▼ リスト 3.18: フェッチ処理の変更

```

// 命令フェッチ処理
always_comb {
    // FIFOに空きがあるとき、命令をフェッチする
    membus.valid = if_fifo_wready; ← 1をif_fifo_wreadyに変更
    membus.addr  = if_pc;
    membus.wen   = 0;
    membus.wdata = 'x; // wdataは使用しない

    // 常にFIFOから命令を受け取る
    if_fifo_rready = 1;
}

```

上のコードでは、メモリに命令フェッチを要求する条件を、FIFO に空きがあるという条件に変更しています。これにより、FIFO があふれてしまうことがなくなります。また、とりあえず FIFO から常にデータを取り出すようにしています。

次に、命令をフェッチできたら FIFO に格納するようにします。

▼ リスト 3.19: FIFO へのデータの格納

```

always_ff {
    ...
    // IFのFIFOの制御
    if if_is_requested && membus.rvalid { ←フェッチできた時
        if_fifo_wvalid    = 1;
        if_fifo_wdata.addr = if_pc_requested;
        if_fifo_wdata.bits = membus.rdata;
    } else {
        if if_fifo_wvalid && if_fifo_wready { ← FIFOにデータを格納できる時
            if_fifo_wvalid = 0;
        }
    }
}

```

上のコードを `always_ff` ブロックの中に追加します。また、`if_fifo_wvalid` と `if_fifo_wdata` を `if_reset` 内で 0 に初期化してください。

フェッチができた時、`if_fifo_wvalid` レジスタの値を 1 にして、`if_fifo_wdata` レジスタにフェッチした命令とアドレスを格納します。これにより、次のクロック以降の FIFO に空きがあるタイミングでデータが追加されます。

それ以外の時、FIFO にデータを格納しようとしていて FIFO に空きがあるとき、`if_fifo_wvalid` を 0 にすることでデータの追加を完了します。

命令フェッチは FIFO に空きがあるときにのみ行うため、まだ追加されていないデータが `if_fifo_wdata` レジスタに格納されていても別のデータに上書きされてしまうことはありません。

▼ リスト 3.20: 命令を表示する

```
let inst_pc : Addr = if_fifo_rdata.addr;
let inst_bits: Inst = if_fifo_rdata.bits;

always_ff {
    if if_fifo_rvalid {
        $display("%h : %h", inst_pc, inst_bits);
    }
}
```

命令を表示するコードを上のように変更し、シミュレータを実行しましょう。命令がフェッチされて表示されるまでに、FIFO に格納して取り出すクロック分だけ遅延があることに注意してください。

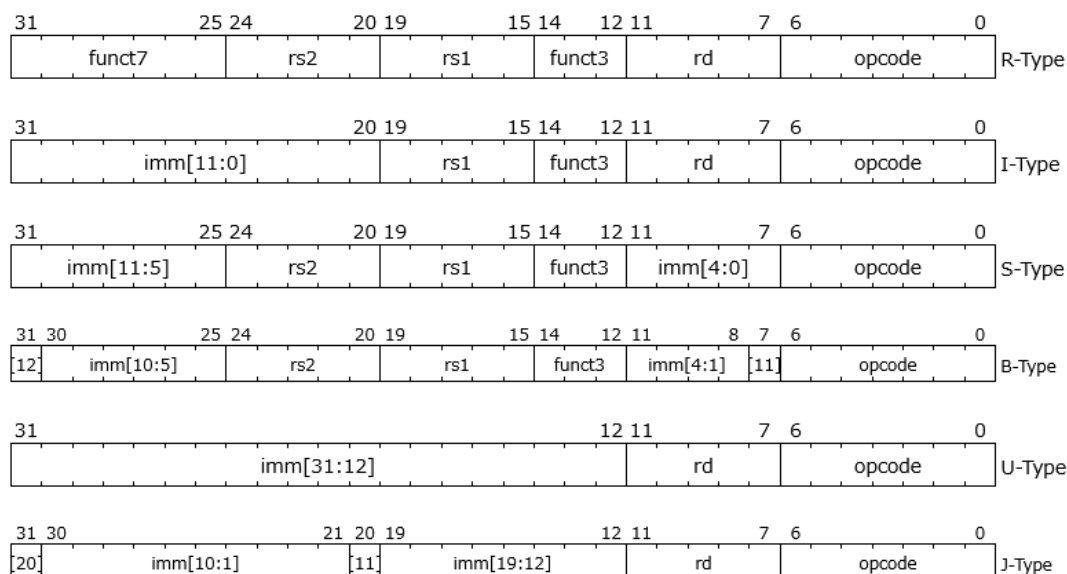
▼ リスト 3.21: FIFO をテストする

```
$ make build sim
$ obj_dir/sim src/sample.hex 7
00000000 : 01234567
00000004 : 89abcdef
00000008 : deadbeef
0000000c : cafebebe
```

3.7 命令のデコードと即値の生成

命令をフェッチすることができたら、フェッチした命令がどのような意味を持つかをチェックし、CPU が何をすればいいかを判断するためのフラグや値を生成します。この作業のことを、命令のデコードと呼びます。

RISC-V にはいくつかの命令の形式がありますが、RV32I には R, I, S, B, U, J の 6 つの形式の命令が存在しています。



▲ 図 3.1: RISC-V の命令形式 (引用元: The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411 2.3. Immediate Encoding Variants)

R 形式

ソースレジスタ (rs1, rs2) が 2 つ、デスティネーションレジスタ (rd) が 1 つの命令形式です。2 つのソースレジスタの値を使って計算し、その結果をデスティネーションレジスタに格納します。例えば ADD, SUB 命令に使用されています。

I 形式

ソースレジスタ (rs1) が 1 つ、デスティネーションレジスタ (rd) が 1 つの命令形式です。12 ビットの即値 (imm[11:0]) が命令中に含まれており、これと rs1 を使って計算し、その結果をデスティネーションレジスタに格納します。例えば ADDI, SUBI 命令に使用されています。

S 形式

ソースレジスタ (rs1, rs2) が 2 つ、デスティネーションレジスタ (rd) が 1 つの命令形式です。12 ビットの即値 (imm[11:5], imm[4:0]) が命令中に含まれており、これとソースレジスタを使って計算やメモリにアクセスし、その結果をデスティネーションレジスタに格納します。例えば SW 命令 (メモリにデータを格納する命令) に使用されています。

B 形式

ソースレジスタ (rs1, rs2) が 2 つの命令形式です。12 ビットの即値 (imm[12], imm[11], imm[10:5], imm[4:1]) が命令中に含まれています。分岐命令に使用されており、ソースレジスタの計算の結果が分岐を成立させる場合、即値を使ってジャンプします。

U 形式

デスティネーションレジスタ (rd) が 1 つの命令形式です。20 ビットの即値 (`imm[31:12]`) が命令中に含まれています。例えば LUI 命令 (レジスタの上位 20 ビットを設定する命令) に使用されています。

J 形式

デスティネーションレジスタ (rd) が 1 つの命令形式です。20 ビットの即値 (`imm[20]`, `imm[19:12]`, `imm[11]`, `imm[10:1]`) が命令中に含まれています。例えば JAL 命令 (ジャンプ命令) に使用されており、PC に即値を足した相対位置にジャンプします。

全ての命令形式には `opcode` が共通して存在しています。命令の判別には `opcode`、`funct3`、`funct7` を利用します。

3.7.1 定数と型の定義

デコード処理を書く前に、デコードに利用する定数と型を定義します。 `src/corectrl.veryl` を作成し、次のように記述します。

▼ リスト 3.22: `corectrl.veryl`

```
import eei::*;

package corectrl {
    // 命令形式を表す列挙型
    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,
    }

    // 制御に使うフラグ用の構造体
    struct InstCtrl {
        itype      : InstType    , // 命令の形式
        rwb_en     : 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
    }
}
```

`InstType` は、命令の形式を表すための列挙型です。 `InstType` の幅は 6 ビットで、それぞれの

ビットに1つの命令形式が対応しています。どの命令形式にも対応しない場合、すべてのビットが0の `InstType::X` を対応させます。

`InstCtrl` は、制御に使うフラグを列挙するための構造体です。 `itype` には命令の形式、 `funct3` , `funct7` には、それぞれ命令の `funct3` , `funct3` フィールドを格納します。これ以外の構造体のメンバーについては、使用するときの説明します。

命令をデコードするとき、まず `opcode` を使って判別します。このために、デコードに使う定数を `eei` パッケージに記述します。

▼ リスト 3.23: `eei.veryl` に追加で記述する

```
// opcode
const OP_OP_IMM : logic<7> = 7'b0010011;
const OP_LUI    : logic<7> = 7'b0110111;
const OP_AUIPC  : logic<7> = 7'b0010111;
const OP_OP     : logic<7> = 7'b0110011;
const OP_JAL    : logic<7> = 7'b1101111;
const OP_JALR   : logic<7> = 7'b1100111;
const OP_BRANCH : logic<7> = 7'b1100011;
const OP_LOAD   : logic<7> = 7'b0000011;
const OP_STORE  : logic<7> = 7'b0100011;
const OP_MISC_MEM : logic<7> = 7'b0001111;
const OP_SYSTEM : logic<7> = 7'b1110011;
```

これらの値とそれぞれの命令の対応については、仕様書 Volume I の 37. RV32/64G Instruction Set Listings を確認してください。

3.7.2 デコードと即値の生成

デコード処理を書く準備が整いました。 `src/inst_decoder.veryl` を作成し、次のように記述します。

▼ リスト 3.24: `inst_decoder.veryl`

```
import eei::*;
import corectrl::*;

module inst_decoder (
    bits: input  Inst    ,
    ctrl: output InstCtrl,
    imm : output UIntX   ,
) {
    // 即値の生成
    let imm_i_g: logic<12> = bits[31:20];
    let imm_s_g: logic<12> = {bits[31:25], bits[11:7]};
    let imm_b_g: logic<12> = {bits[31], bits[7], bits[30:25], bits[11:8]};
    let imm_u_g: logic<20> = bits[31:12];
    let imm_j_g: logic<20> = {bits[31], bits[19:12], bits[20], bits[30:21]};
    let imm_z_g: logic<17> = bits[31:15]; // {csr address, uimm}

    let imm_i: UIntX = {imm_i_g[msb] repeat XLEN - $bits(imm_i_g), imm_i_g};
```



```

let imm_s: UIntX = {imm_s_g[msb] repeat XLEN - $bits(imm_s_g), imm_s_g};
let imm_b: UIntX = {imm_b_g[msb] repeat XLEN - $bits(imm_b_g) - 1, imm_b_g, 1'b0};
let imm_u: UIntX = {imm_u_g[msb] repeat XLEN - $bits(imm_u_g) - 12, imm_u_g, 12'b0};
let imm_j: UIntX = {imm_j_g[msb] repeat XLEN - $bits(imm_j_g) - 1, imm_j_g, 1'b0};
let imm_z: UIntX = {1'b0 repeat XLEN - $bits(imm_z_g), imm_z_g};

let op: logic<7> = bits[6:0];
let f7: logic<7> = bits[31:25];
let f3: logic<3> = bits[14:12];

const T: logic = 1'b1;
const F: logic = 1'b0;

always_comb {
    imm = case op {
        OP_LUI, OP_AUIPC                : imm_u,
        OP_JAL                          : imm_j,
        OP_JALR, OP_LOAD, OP_OP_IMM, OP_MISC_MEM: imm_i,
        OP_BRANCH                       : imm_b,
        OP_STORE                        : imm_s,
        OP_SYSTEM                       : imm_z,
        default                          : 'x,
    };
    ctrl = {case op {
        OP_LUI      : {InstType::U, T, T, F, F, F, F, F},
        OP_AUIPC    : {InstType::U, T, F, F, F, F, F, F},
        OP_JAL      : {InstType::J, T, F, F, T, F, F, F},
        OP_JALR     : {InstType::I, T, F, F, T, F, F, F},
        OP_BRANCH   : {InstType::B, F, F, F, F, F, F, F},
        OP_LOAD     : {InstType::I, T, F, F, F, T, F, F},
        OP_STORE    : {InstType::S, F, F, F, F, F, F, F},
        OP_OP       : {InstType::R, T, F, T, F, F, F, F},
        OP_OP_IMM   : {InstType::I, T, F, T, F, F, F, F},
        OP_MISC_MEM : {InstType::I, F, F, F, F, F, F, T},
        OP_SYSTEM   : {InstType::I, T, F, F, F, F, T, F},
        default     : {InstType::X, F, F, F, F, F, F, F},
    }, f3, f7};
}
}

```

inst_decoder モジュールは、命令のビット列 **bits** を受け取り、制御信号 **ctrl** と即値 **imm** を出力します。

即値の生成

B 形式の命令について考えます。まず、命令のビット列から即値部分を取り出して、**imm_b_g** ワイヤを生成します。B 形式の命令内に含まれている即値は 12 ビットで、最上位ビットは符号ビットです。最上位ビットを繰り返す (符号拡張する) ことによって、32 ビットの即値 **imm_b** を生成します。

imm_z は CSR 命令で使用する即値をまとめたものです。これについては後の章で説明します。

always_comb ブロックでは、opcode を case 式で分岐することにより **imm** ポートに適切な即値

を出力しています。

制御フラグの生成

opcode が OP-IMM な命令、例えば ADDI 命令について考えます。ADDI 命令は、即値とソースレジスタの値を足し、デスティネーションレジスタに結果を格納する命令です。

`always_comb` ブロックでは、opcode が `OP_OP_IMM` のとき、次のように制御信号 `ctrl` を設定します。

- 命令形式 `itype` を `InstType::I` に設定します
- `funct3` , `funct7` を命令中のビットをそのまま設定します
- 結果をレジスタに書き込むため、`rwb_en` を 1 に設定します
- ALU(計算を実行するユニット)を利用するため、`is_aluop` を 1 に設定します。
- それ以外のメンバーは 0 に設定します。

3.7.3 デコーダのインスタンス化

`inst_decoder` モジュールを、`core` モジュールでインスタンス化します。

▼ リスト 3.25: `inst_decoder` のインスタンス化 (`core.veryl`)

```
let inst_pc : Addr      = if_fifo_rdata.addr;
let inst_bits: Inst     = if_fifo_rdata.bits;
var inst_ctrl: InstCtrl;
var inst_imm : UIntX    ;

inst_decoder: inst_decoder (
  bits: inst_bits,
  ctrl: inst_ctrl,
  imm : inst_imm ,
);
```

まず、デコーダと `core` モジュールを接続するために `inst_ctrl` と `inst_imm` を定義します。次に、`inst_decoder` モジュールをインスタンス化します。`bits` ポートに `inst_bits` を渡すことで、フェッチした命令をデコードします。

▼ リスト 3.26: デコード結果の表示プログラム (`core.veryl`)

```
always_ff {
  if if_fifo_rvalid {
    $display("%h : %h", inst_pc, inst_bits);
    $display(" itype : %b", inst_ctrl.itype);
    $display(" imm   : %h", inst_imm);
  }
}
```

デバッグ用の `always_ff` ブロックに、デコードした結果を表示するプログラムを記述します。

`sample.hex` をメモリの初期値として使い、デコード結果を確認します。

▼ リスト 3.27: デコーダのテスト

```
$ make build sim
$ obj_dir/sim_src/sample.hex 7
00000000 : 01234567
  itype   : 000010
  imm     : 00000012
00000004 : 89abcdef
  itype   : 100000
  imm     : fffbc09a
00000008 : deadbeef
  itype   : 100000
  imm     : fffdb5ea
0000000c : cafebebe
  itype   : 000000
  imm     : 00000000
```

例えば `01234567` は、`jalr x10, 18(x6)` という命令のビット列になります。命令の種類は JALR で、命令形式は I 形式、即値は 10 進数で `18` です。デコード結果を確認すると、`itype` が `0000010`、`imm` が `00000012` になっており、正しくデコードできていることが確認できます。

3.8 レジスタの定義と読み込み

RV32I の仕様では、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。

命令を実行するとき、実行に使うデータをレジスタ番号で指定することがあります。実行に使うデータとなるレジスタのことを、ソースレジスタと呼びます。また、命令の結果を、指定された番号のレジスタに格納することがあります。このために使われるレジスタのことを、デスティネーションレジスタと呼びます。

core モジュールに、レジスタを定義します。RV32I のレジスタの幅は XLEN(=32) ビットです。よって、サイズが 32 の `UIntX` 型のレジスタの配列を定義します。

▼ リスト 3.28: レジスタの定義 (core.veryl)

```
// レジスタ
var regfile: UIntX<32>;
```

レジスタをまとめたもののことをレジスタファイルと呼ぶため、`regfile` という名前をつけています。

図 3.1 を見るとわかるように、RISC-V の命令は形式によってソースレジスタの数が異なります。例えば、R 形式はソースレジスタが 2 つで、2 つのレジスタのデータを使って実行されます。それに対して、I 形式のソースレジスタは 1 つです。I 形式の命令の実行には、ソースレジスタのデータと即値を利用します。

レジスタを定義したので、命令が使用するレジスタのデータを取得します。命令のビット列の中のソースレジスタの番号の場所は、命令形式が違っていても共通の場所にあります。

ここで、プログラムを簡単にするために、命令中のソースレジスタの番号にあたる場所に、常にソースレジスタの番号が書かれていると解釈します。更に、命令がレジスタのデータを利用するかどうかに関係なく、常にレジスタのデータを読み込むことにします。

▼ リスト 3.29: 命令が使うレジスタのデータを取得する (core.veryl)

```
// レジスタ番号
let rs1_addr: logic<5> = inst_bits[19:15];
let rs2_addr: logic<5> = inst_bits[24:20];

// ソースレジスタのデータ
let rs1_data: UIntX = if rs1_addr == 0 {
    0
} else {
    regfile[rs1_addr]
};
let rs2_data: UIntX = if rs2_addr == 0 {
    0
} else {
    regfile[rs2_addr]
};
```

if 式により、0 番目のレジスタが指定されたときは、常に 0 になるようにします。レジスタの値を読み込めていることを確認するために、次のように記述します。

▼ リスト 3.30: レジスタの値を表示する (core.veryl)

```
always_ff {
    if if_fifo_rvalid {
        $display("%h : %h", inst_pc, inst_bits);
        $display(" itype : %b", inst_ctrl.itype);
        $display(" imm : %h", inst_imm);
        $display(" rs1[%d] : %h", rs1_addr, rs1_data);
        $display(" rs2[%d] : %h", rs2_addr, rs2_data);
    }
}
```

\$display システムタスクで、命令のレジスタ番号とデータを表示します。早速動作のテストをしたいところですが、今のままだとレジスタのデータが初期化されておらず、0 番目のレジスタのデータ以外は不定 (0 か 1 か分からない) になってしまいます。

これではテストする意味がないため、レジスタの値を適当な値に初期化します。

▼ リスト 3.31: レジスタの値を初期化する (core.veryl)

```
// レジスタの初期化
always_ff {
    if_reset {
        for i: i32 in 0..32 {
            regfile[i] = i + 100;
        }
    }
}
```

```
    }
  }
}
```

上のコードでは、`always_ff` ブロックの `if_reset` で、 n 番目 ($32 > n > 0$) のレジスタの値を `n + 100` で初期化しています。

▼ リスト 3.32: レジスタ読み込みのデバッグ

```
$ make build sim
$ obj_dir/sim sample.hex 7
TODO にや
```

`01234567` は `jalr x10, 18(x6)` です。JALR 命令は、2つのソースレジスタ `x10` と `x6` を使用します。それぞれ、レジスタ番号が `10` , `6` であることを表しており、値は `110` , `106` になります。それぞれ 16 進数で `TODO` , `TODO` です。これが、シミュレーションと一致していることを確認してください。

3.9 ALU を作り、計算する

命令は足し算や引き算、ビット演算などの計算を行います。計算の対象となるデータが揃ったので、ALU(計算する部品)を作成します。

3.9.1 ALU の作成

データの幅は `XLEN` です。計算には、符号付き整数と符号なし整数向けの計算があります。これに利用するために、`eei` モジュールに `XLEN` ビットの符号あり整数型を定義します。

▼ リスト 3.33: XLEN ビットの符号付き整数を定義する (`eei.veryl`)

```
type SIntX  = signed logic<XLEN>;
type SInt32 = signed logic<32> ;
type SInt64 = signed logic<64> ;
```

次に、`src/alu.veryl` を作成し、次のように記述します。

▼ リスト 3.34: `alu.veryl`

```
import eei::*;
import corectrl::*;

module alu (
  ctrl : input  InstCtrl,
  op1  : input  UIntX ,
  op2  : input  UIntX ,
  result: output UIntX ,
) {
```

```

let add: UIntX = op1 + op2;
let sub: UIntX = op1 - op2;

let srl: UIntX = op1 >> op2[4:0];
let sra: SIntX = $signed(op1) >>> op2[4:0];

always_comb {
  if ctrl.is_aluop {
    case ctrl.funct3 {
      3'b000: result = if ctrl.itype == InstType::I | ctrl.funct7 == 0 {
        add // ADD, ADDI
      } else {
        sub // SUB
      };
      3'b001: result = op1 << op2[4:0]; // SLL, SLLI
      3'b010: result = {1'b0 repeat XLEN - 1, $signed(op1) <: $signed(op2)}; // SLT, >
      3'b011: result = {1'b0 repeat XLEN - 1, op1 <: op2}; // SLTU, SLTUI
      3'b100: result = op1 ^ op2; // XOR, XORI
      3'b101: result = if ctrl.funct7 == 0 {
        srl // SRL, SRLI
      } else {
        sra // SRA, SRAI
      };
      3'b110 : result = op1 | op2; // OR, ORI
      3'b111 : result = op1 & op2; // AND, ANDI
      default: result = 'x;
    }
  } else {
    result = add;
  }
}
}

```

alu モジュールには、次のポートを定義します。

▼表 3.3: alu モジュールのポート定義

ポート名	方向	型	用途
ctrl	input	InstCtrl	制御用信号
op1	input	UIntX	1 つ目のデータ
op2	input	UIntX	2 つ目のデータ
result	output	UIntX	結果

命令が ALU でどのような計算を行うかは命令の種別によって異なります。RV32I では、仕様書の 2.4. Integer Computational Instructions (整数演算命令) に定義されている命令は、命令の funct3, funct7 フィールドによって計算の種類を特定することができます。

それ以外の命令は、CSR 命令を除いて足し算しか行いません。そのため、デコード時に整数演算命令とそれ以外の命令を **InstCtrl.is_aluop** で区別し、整数演算命令以外は常に足し算を行うよ

うにしています。具体的には、`opcode` が OP か OP-IMM の命令の `InstCtrl.is_aluop` を 1 にしています。(inst_decoder モジュールを確認してください)

`always_comb` ブロックでは、case 文で funct3 によって計算を区別します。それだけでは区別できないとき、funct7 を使用します。

▼ リスト 3.35: ALU に渡すデータの用意 (core.veryl)

```
// ALU
var op1      : UIntX;
var op2      : UIntX;
var alu_result: UIntX;

always_comb {
  case inst_ctrl.i_type {
    InstType::R, InstType::B: {
      op1 = rs1_data;
      op2 = rs2_data;
    }
    InstType::I, InstType::S: {
      op1 = rs1_data;
      op2 = inst_imm;
    }
    InstType::U, InstType::J: {
      op1 = inst_pc;
      op2 = inst_imm;
    }
    default: {
      op1 = 'x;
      op2 = 'x;
    }
  }
}
```

次に、ALU に渡すデータを用意します。`UIntX` 型の変数 `op1` , `op2` , `alu_result` を定義し、`always_comb` ブロックで値を割り当てます。割り当てるデータは命令形式によって次のように異なります。

R 形式, B 形式

R 形式, B 形式は、レジスタのデータとレジスタのデータの演算を行います。`op1` , `op2` は、レジスタのデータ `rs1_data` , `rs2_data` になります。

I 形式, S 形式

I 形式, S 形式は、レジスタのデータと即値の演算を行います。`op1` , `op2` は、それぞれレジスタのデータ `rs1_data` , 即値 `inst_imm` になります。S 形式はメモリのストア命令に利用されており、レジスタのデータと即値を足し合わせた値がアクセスするアドレスになります。

U 形式, J 形式

U 形式, J 形式は、即値と PC を足した値、または即値を使う命令に使われています。`op1` ,

`op2` は、それぞれ PC `inst_pc` , 即値 `inst_imm` になります。J 形式は JAL 命令に利用されており、即値と PC を足した値がジャンプ先になります。U 形式は AUIPC 命令と LUI 命令に利用されています。AUIPC 命令は、即値と PC を足した値をデスティネーションレジスタに格納します。LUI 命令は、即値をそのままデスティネーションレジスタに格納します。

▼ リスト 3.36: ALU のインスタンス化 (core.veryl)

```
inst alu: alu (
    ctrl : inst_ctrl ,
    op1   ,
    op2   ,
    result: alu_result,
);
```

ALU に渡すデータを用意したので、alu モジュールをインスタンス化します。結果を受け取る用の変数として、`alu_result` を指定します。

3.9.2 ALU のテスト

最後に ALU が正しく動くことを確認します。`always_ff` ブロックで、`op1` , `op2` , `alu_result` を表示します。

▼ リスト 3.37: ALU の結果表示 (core.veryl)

```
always_ff {
    if if_fifo_rvalid {
        $display("%h : %h", inst_pc, inst_bits);
        $display(" itype : %b", inst_ctrl.itype);
        $display(" imm : %h", inst_imm);
        $display(" rs1[%d] : %h", rs1_addr, rs1_data);
        $display(" rs2[%d] : %h", rs2_addr, rs2_data);
        $display(" op1 : %h", op1); ←追加
        $display(" op2 : %h", op2); ←追加
        $display(" alu res : %h", alu_result); ←追加
    }
}
```

`sample.hex` を次のように書き換えます。

▼ リスト 3.38: sample.hex を書き換える

```
02000093 // addi x1, x0, 32
00100117 // auipc x2, 256
002081b3 // add x3, x1, x2
```

それぞれの命令の意味は次のとおりです。
シミュレータを実行し、結果を確かめます。

▼表 3.4: 命令の意味

アドレス	命令	意味
00000000	<code>addi x1, x0, 32</code>	$x1 = x0 + 32$
00000004	<code>auipc x2, 256</code>	$x2 = pc + 256$
00000008	<code>add x3, x1, x2</code>	$x3 = x1 + x2$

▼リスト 3.39: ALU のデバッグ

```
$ make build sim
$ obj_dir/sim src/sample.hex 5
TODO
```

まだ結果をディスティネーションレジスタに格納する処理を作成していません。そのため、レジスタの値は変わらないことに注意してください

addi x1, x0, 32

`op1` は 0 番目のレジスタの値です。0 番目のレジスタの値は常に 0 であるため、`00000000` と表示されています。`op2` は即値です。即値は 32 であるため、16 進数で `00000020` と表示されています。ALU の計算結果として、0 と 32 を足した結果 `00000020` が表示されています。

auipc x2, 256

`op1` は 2 番目のレジスタの値です。2 番目のレジスタは `102` として初期化しているので、`TODO` と表示されています。`op2` は PC です。命令のアドレス `00000004` が表示されています。ALU の計算結果として、これを足した結果 `TODO` が表示されています。

add x3, x1, x2

`op1` は 1 番目のレジスタの値です。1 番目のレジスタは `101` として初期化しているので、`TODO` と表示されています。2 番目のレジスタは `102` として初期化しているので、`TODO` と表示されています。ALU の計算結果として、これを足した結果 `TODO` が表示されています。

3.10 レジスタに結果を書き込む

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。

ライトバックする値は、計算やメモリアクセスの結果です。まだメモリにアクセスする処理を実装していませんが、先にライトバック処理を実装します。

3.10.1 ライトバックの実装

書き込む対象のレジスタは、命令の `rd` フィールドによって番号で指定します。デコード時に、

ライトバックする命令かどうかを `InstCtrl.rwb_en` に格納しています。(inst_decoder モジュールを確認してください)

▼ リスト 3.40: ライトバック処理の実装 (core.veryl)

```
let rd_addr: logic<5> = inst_bits[11:7];
let wb_data: UIntX    = alu_result;

always_ff {
  if_reset {
    for i: i32 in 0..32 {
      regfile[i] = i + 100;
    }
  } else {
    if if_fifo_rvalid && inst_ctrl.rwb_en {
      regfile[rd_addr] = wb_data;
    }
  }
}
```

3.10.2 ライトバックのテスト

`always_ff` ブロックに、ライトバック処理の概要を表示するプログラムを記述します。処理している命令がライトバックする命令のときにのみ、`$display` システムコールを呼び出します。

▼ リスト 3.41: 結果の表示 (core.veryl)

```
if inst_ctrl.rwb_en {
  $display(" reg[%d] <= %h", rd_addr, wb_data);
}
```

シミュレータを実行し、結果を確認めます。

▼ リスト 3.42: ライトバックのデバッグ

```
$ make build sim
$ obj_dir/sim sample.hex 6
00000000 : 02000093
  itype      : 000010
  imm        : 00000020
  rs1[ 0]    : 00000000
  rs2[ 0]    : 00000000
  op1        : 00000000
  op2        : 00000020
  alu res    : 00000020
  reg[ 1] <= 00000020
00000004 : 00100117
  itype      : 010000
  imm        : 00100000
  rs1[ 0]    : 00000000
  rs2[ 1]    : 00000020
  op1        : 00000004
  op2        : 00100000
```

```

alu res : 00100004
reg[ 2] <= 00100004
00000008 : 002081b3
itype   : 000001
imm     : 00000000
rs1[ 1] : 00000020
rs2[ 2] : 00100004
op1     : 00000020
op2     : 00100004
alu res : 00100024
reg[ 3] <= 00100024

```

addi x1, x0, 32

x1 に、0 と 32 を足した結果を格納しています。

auipc x2, 256

x2 に、PC と 256 を足した結果を格納しています。

add x3, x1, x2

x1 は 1 つ目の命令で 00000020 に、x2 は 2 つ目の命令で 00100004 にされています。x3 に、x1 と x2 を足した結果 00100024 を格納しています。

おめでとうございます！ この CPU は整数演算命令の実行ができるようになりました。

3.11 ロード命令とストア命令の実装

RV32I には、メモリのデータをロードする (読み込む)、ストアする (書き込む) 命令として次の命令があります。

▼表 3.5: ロード命令, ストア命令

命令	作用
LB	8 ビットのデータを読み込む。上位 24 ビットは符号拡張する
LBU	8 ビットのデータを読み込む。上位 24 ビットは 0 とする
LH	16 ビットのデータを読み込む。上位 16 ビットは符号拡張する
LHU	16 ビットのデータを読み込む。上位 16 ビットは 0 とする
LW	32 ビットのデータを読み込む
SB	8 ビットのデータを書き込む
SH	16 ビットのデータを書き込む
SW	32 ビットのデータを書き込む

ロード命令は I 形式、ストア命令は S 形式です。これらの命令で指定するメモリのアドレスは、rs1 と即値の足し算です。ALU に渡すデータが rs1 と即値になっていることを確認してください (リスト 3.29)。ストア命令は、rs2 の値をメモリに格納します。

3.11.1 LW, SW 命令の実装

8ビット、16ビット単位で読み書きを行う命令の実装は少し大変です。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。

memunit モジュールの作成

メモリ操作を行うモジュールを `memunit.veryl` に記述します。

▼リスト 3.43: memunit.veryl

```
import eei::*;
import corectrl::*;

module memunit (
    clk      : input  clock          ,
    rst      : input  reset          ,
    valid    : input  logic          ,
    is_new    : input  logic          , // 命令が新しく供給されたかどうか
    ctrl     : input  InstCtrl        , // 命令のInstCtrl
    addr     : input  Addr            , // アクセスするアドレス
    rs2      : input  UIntX           , // ストア命令で書き込むデータ
    rdata    : output UIntX           , // ロード命令の結果 (stall = 0のときに有効)
    stall    : output logic          , // メモリアクセス命令が完了していない
    membus   : modport membus_if::master, // メモリとのinterface
) {

    // 命令がメモリにアクセスする命令か判別する関数
    function inst_is_memop (
        ctrl: input InstCtrl,
    ) -> logic {
        return ctrl.itype == InstType::S || ctrl.is_load;
    }

    // 命令がストア命令か判別する関数
    function inst_is_store (
        ctrl: input InstCtrl,
    ) -> logic {
        return inst_is_memop(ctrl) && !ctrl.is_load;
    }

    // memunitの状態を表す列挙型
    enum State: logic<2> {
        Init, // 命令を受け付ける状態
        WaitReady, // メモリが操作可能になるのを待つ状態
        WaitValid, // メモリ操作が終了するのを待つ状態
    }

    var state: State;

    var req_wen : logic ;
    var req_addr : Addr ;
    var req_wdata: UInt32;
```

```

always_comb {
    // メモリアクセス
    membus.valid = state == State::WaitReady;
    membus.addr  = req_addr;
    membus.wen   = req_wen;
    membus.wdata = req_wdata;
    // loadの結果
    rdata = membus.rdata;
    // stall判定
    stall = valid & case state {
        State::Init      : is_new && inst_is_memop(ctrl),
        State::WaitReady: 1,
        State::WaitValid: !membus.rvalid,
        default          : 0,
    };
}

always_ff {
    if_reset {
        state      = State::Init;
        req_wen    = 0;
        req_addr   = 0;
        req_wdata  = 0;
    } else {
        if valid {
            case state {
                State::Init: if is_new & inst_is_memop(ctrl) {
                    state      = State::WaitReady;
                    req_wen    = inst_is_store(ctrl);
                    req_addr   = addr;
                    req_wdata  = rs2;
                }
                State::WaitReady: if membus.ready {
                    state = State::WaitValid;
                }
                State::WaitValid: if membus.rvalid {
                    state = State::Init;
                }
                default: {}
            }
        }
    }
}
}

```

memunit モジュールでは、命令がメモリにアクセスする命令の時、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。

命令がメモリにアクセスする命令かどうかは、`inst_is_memop` 関数で判定します。ストア命令のとき、命令の形式は S 形式です。ロード命令のとき、デコーダは `InstCtrl.is_load` を 1 にしています。

memunit モジュールには、次の状態が定義されています。初期状態は `State::Init` です。

State::Init

memunit モジュールに新しく命令が供給されたとき、`valid` と `is_new` が 1 になります。新しく命令が供給されて、それがメモリにアクセスする命令のとき、状態を `State::WaitReady` に移動します。その際、`req_wen` にストア命令かどうか、`req_addr` にアクセスするアドレス、`req_wdata` に `rs2` を格納します。

State::WaitReady

この状態の時、命令に応じた要求をメモリに送り続けます。メモリが要求を受け付ける (`ready`) とき、状態を `State::WaitValid` に移動します。

State::WaitValid

メモリに送信した要求の処理が終了した (`rvalid`) とき、状態を `State::Init` に移動します。

メモリにアクセスする命令のとき、memunit モジュールは `Init` , `WaitReady` , `WaitValid` の順で状態を移動するため、実行には少なくとも 3 クロックが必要です。その間、CPU はレジスタのライトバック処理や FIFO からの命令の取り出しを待つ必要があります。

これを実現するために、memunit モジュールには処理中かどうかを表す `stall` フラグが存在します。有効な命令が供給されているとき、`state` やメモリの状態に応じて、次のように `stall` を決定します。

▼ 表 3.6: stall の値の決定方法

状態	stall が 1 になる条件
Init	新しく命令が供給されて、それがメモリにアクセスする命令のとき
WaitReady	常に 1
WaitValid	処理が終了していない (<code>!membus.rvalid</code>) とき



アドレスが 4 バイトに整列されていない場合の動作

今のところ、memory モジュールはアドレスの下位 2 ビットを無視するため、`addr` の下位 2 ビットが `00` ではない、つまり、4 で割り切れないアドレスに対して `LW`, `SW` 命令を実行する場合、memunit モジュールは正しい動作をしません。2 で割り切れないアドレスに対する `LH`, `LHU`, `SH` 命令についても同様です。これらの問題については後の章で対策するため、今は無視します。

memunit モジュールのインスタンス化

core モジュール内に memunit モジュールをインスタンス化します。

まず、命令が供給されていることを示す信号 `inst_valid` と、命令が現在のクロックで供給されたことを示す信号 `inst_is_new` を作成します。

▼ リスト 3.44: inst_valid, inst_is_new の定義 (core.veryl)

```
let inst_valid : logic    = if_fifo_rvalid;
var inst_is_new: logic    ; // 命令が今のクロックで供給されたかどうか
```

▼ リスト 3.45: inst_is_new の実装 (core.veryl)

```
always_ff {
  if_reset {
    inst_is_new = 0;
  } else {
    if if_fifo_rvalid {
      inst_is_new = if_fifo_rready;
    } else {
      inst_is_new = 1;
    }
  }
}
```

命令が供給されているかどうかは、`if_fifo_rvalid` と同値です。これを機に、`if_fifo_rvalid` を使用しているところを `inst_valid` に置き換えましょう。

命令が現在のクロックで供給されたかどうかは、FIFO の `rvalid` , `rready` を観測することでわかります。`rvalid` が 1 のとき、`ready` が 1 なら、次のクロックで供給される命令は新しく供給される命令です。`ready` が 0 なら、次のクロックで供給されている命令は現在のクロックと同じ命令になります。`rvalid` が 0 のとき、次のクロックで供給される命令は常に新しく供給される命令になります。(次のクロックで `rvalid` が 1 かどうかについては考えません)

さて、memunit モジュールをインスタンス化する前に、メモリとの接続方法について考える必要があります。

core モジュールには、メモリとの接続点として membus ポートが存在します。しかし、これは命令フェッチ用に使用されているため、memunit モジュール用に使用することができません。また、memory モジュールは同時に 2 つの操作を受け付けることができません。

この問題を、core モジュールにメモリとの接続点を 2 つ用意し、それを top モジュールで調停することにより回避します。

▼ リスト 3.46: core モジュールのポート定義 (core.veryl)

```
module core (
  clk      : input  clock      ,
  rst      : input  reset      ,
  i_membus : modport membus_if::master,
  d_membus : modport membus_if::master,
) {
```

まず、core モジュールに、命令フェッチ用のポート `i_membus` , ロードストア命令用のポート `d_membus` の 2 つのポートを用意します。命令フェッチ用のポートが `membus` から `i_membus` に変更されるため、既存の `membus` を `i_membus` に置き換えてください。

▼ リスト 3.47: membus を i_membus に置き換える (core.veryl)

```
// FIFOに空きがあるとき、命令をフェッチする
i_membus.valid = if_fifo_wready;
i_membus.addr  = if_pc;
i_membus.wen   = 0;
i_membus.wdata = 'x; // wdataは使用しない
```

次に、top モジュールでの調停を実装します。

▼ リスト 3.48: メモリへのアクセス要求の調停 (top.veryl)

```
inst membus : membus_if;
inst i_membus: membus_if; // 命令フェッチ用
inst d_membus: membus_if; // ロードストア命令用

var memarb_last_i: logic;

// メモリアccessを調停する
always_ff {
    if_reset {
        memarb_last_i = 0;
    } else {
        if membus.ready {
            memarb_last_i = !d_membus.valid;
        }
    }
}

always_comb {
    i_membus.ready  = membus.ready && !d_membus.valid;
    i_membus.rvalid = membus.rvalid && memarb_last_i;
    i_membus.rdata  = membus.rdata;

    d_membus.ready  = membus.ready;
    d_membus.rvalid = membus.rvalid && !memarb_last_i;
    d_membus.rdata  = membus.rdata;

    membus.valid = i_membus.valid | d_membus.valid;
    if d_membus.valid {
        membus.addr  = d_membus.addr;
        membus.wen   = d_membus.wen;
        membus.wdata = d_membus.wdata;
    } else {
        membus.addr  = i_membus.addr;
        membus.wen   = i_membus.wen;
        membus.wdata = i_membus.wdata;
    }
}
```

新しく、`i_membus` と `d_membus` をインスタンス化し、それを `membus` と接続します。

調停の仕組みは次のとおりです。

- `i_membus` と `d_membus` の両方の `valid` が 1 のとき、`d_membus` を優先する

- `memarb_last_i` レジスタに、受け入れた要求が `i_membus` からのものだったかどうかを記録する
- メモリが要求の結果を返すとき、`memarb_last_i` を見て、`i_membus` と `d_membus` のどちらか片方の `rvalid` を 1 にする

命令フェッチを優先していると命令の処理が進まないため、`i_membus` よりも `d_membus` を優先します。

core モジュールとの接続を次のように変更します。

▼ リスト 3.49: membus を 2 つに分けて接続する (top.veryl)

```
inst c: core (
    clk      ,
    rst      ,
    i_membus ,
    d_membus ,
);
```

memory モジュールと memunit を接続する準備が整ったので、memunit モジュールをインスタンス化します。

▼ リスト 3.50: memunit モジュールのインスタンス化 (core.veryl)

```
var memu_rdata: UIntX;
var memu_stall: logic;

inst memu: memunit (
    clk      ,
    rst      ,
    valid : inst_valid ,
    is_new: inst_is_new,
    ctrl  : inst_ctrl ,
    addr  : alu_result ,
    rs2   : rs2_data  ,
    rdata : memu_rdata ,
    stall : memu_stall ,
    membus: d_membus  ,
);
```

memunit モジュールの処理待ちとライトバック

最後に、memunit モジュールが処理中は命令を FIFO から取り出すのを止める処理と、LW 命令で読み込んだデータがレジスタにライトバックする処理を実装します。

▼ リスト 3.51: memunit モジュールの処理が終わるのを待つ (core.veryl)

```
// memunitが処理中ではないとき、FIFOから命令を取り出していい
if_fifo_rready = !memu_stall;
```

▼ リスト 3.52: memunit モジュールの結果をライトバックする (core.veryl)

```
let rd_addr: logic<5> = inst_bits[11:7];
let wb_data: UIntX    = if inst_ctrl.is_load {
    memu_rdata
} else {
    alu_result
};
```

memunit モジュールが処理中のとき、`memu_stall` が 1 になっています。そのため、`memu_stall` が 1 のときは、`if_fifo_rready` を 0 にすることで、FIFO からの命令の取り出しを停止します。

ライトバック処理では、命令がロード命令のとき (`inst_ctrl.is_load`)、`alu_result` ではなく `memu_rdata` を `wb_data` に設定します。

ところで、現在のプログラムでは、memunit の処理が終了していないときもライトバックを続けています。レジスタへのライトバックは命令の実行が終了したときのみで良いため、次のようにプログラムを変更します。

▼ リスト 3.53: 命令の実行が終了したときのみライトバックする (core.veryl)

```
if inst_valid && if_fifo_rready && inst_ctrl.rwb_en {
    regfile[rd_addr] = wb_data;
}
```

▼ リスト 3.54: ライトバックするときのみデバッグ表示する (core.veryl)

```
if if_fifo_rready && inst_ctrl.rwb_en {
    $display(" reg[%d] <= %h", rd_addr, wb_data);
}
```

LW, SW 命令のテスト

LW, SW 命令が正しく動作していることを確認するために、デバッグ出力を次のように変更します。

▼ リスト 3.55: メモリモジュールの状態を出力する (core.veryl)

```
$display(" mem stall : %b", memu_stall);
$display(" mem rdata : %h", memu_rdata);
```

また、ここからのテストは実行するクロック数が多くなるため、ログに何クロック目かを表示することで、ログを読みやすくします。

▼ リスト 3.56: 何クロック目かを出力する (core.veryl)

```
var clock_count: u64;

always_ff {
    if_reset {
        clock_count = 1;
    } else {
```

```

clock_count = clock_count + 1;
if inst_valid {
    $display("# %d", clock_count);
    $display("%h : %h", inst_pc, inst_bits);
    $display(" itype      : %b", inst_ctrl.itype);
}

```

LW, SW 命令のテストのために、sample.hex を次のように変更します。

▼ リスト 3.57: テスト用のプログラムを記述する (sample.hex)

```

02002503 // lw x10, 0x20(x0)
40000593 // addi x11, x0, 0x400
02b02023 // sw x11, 0x20(x0)
02002603 // lw x12, 0x20(x0)
00000000
00000000
00000000
00000000
00000000
deadbeef // 0x20

```

プログラムは次のようになっています。

▼ 表 3.7: メモリに格納するデータ

アドレス	命令	意味
00000000	lw x10, 0x20(x0)	x10 に、アドレスが 0x20 のデータを読み込む
00000004	addi x11, x0, 0x400	x11 = 0x400
00000008	sw x11, 0x20(x0)	アドレス 0x20 に x11 の値を書き込む
0000000c	lw x12, 0x20(x0)	x12 に、アドレスが 0x20 のデータを読み込む

アドレス 0x20 には、データ deadbeef を格納しています。

シミュレータを実行し、結果を確認めます。

▼ リスト 3.58: LW, SW 命令のテスト

```

$ make build sim
$ obj_dir/sim src/sample.hex 13
#                               3
00000000 : 02002503
itype      : 000010
imm        : 00000020
rs1[ 0]    : 00000000
rs2[ 0]    : 00000000
op1        : 00000000
op2        : 00000020
alu res     : 00000020
mem stall  : 1 ← LW命令でストールしている
mem rdata  : 02b02023
(省略)
#                               5
00000000 : 02002503

```

```

itype      : 000010
imm        : 00000020
rs1[ 0]    : 00000000
rs2[ 0]    : 00000000
op1        : 00000000
op2        : 00000020
alu res    : 00000020
mem stall  : 0 ← LWが終わったので0になった
mem rdata  : deadbeef
reg[10] <= deadbeef ← 0x20の値が読み込まれた
(省略)
#                               12
0000000c : 02002603
itype      : 000010
imm        : 00000020
rs1[ 0]    : 00000000
rs2[ 0]    : 00000000
op1        : 00000000
op2        : 00000020
alu res    : 00000020
mem stall  : 0
mem rdata  : 00000400
reg[12] <= 00000400 ←書き込んだ値が読み込まれた

```

3.11.2 LB, LBU, LH, LHU 命令の実装

LB, LBU, SB 命令は 8 ビット単位、LH, LHU, SH 命令は 16 ビット単位でロード/ストアを行う命令です。

まずロード命令を実装します。ロード命令は 32 ビット単位でデータを読み込み、その結果の一部を切り取ることで実装することができます。

まず、何度も記述することになる定数と変数を短い名前 (**W** , **D**) で定義します。

▼ リスト 3.59: W と D の定義 (memunit.veryl)

```

const W: u32      = 32;
let D: UInt32 = membus.rdata;

```

LB, LBU, LH, LHU, LW 命令は、funct3 の値で区別することができます。

▼ 表 3.8: ロード命令の funct3

funct3	命令
000	LB
100	LBU
001	LH
101	LHU
010	LW

funct3 を case 文で分岐し、アドレスの下位ビットを見ることで、命令とアドレスに応じた値を

rdata に設定します。

▼ リスト 3.60: rdata をアドレスと読み込みサイズに応じて変更する (memunit.veryl)

```
// loadの結果
rdata = case ctrl.func3 {
  3'b000 : case addr[1:0] {
    0      : {D[7] repeat W - 8, D[7:0]},
    1      : {D[15] repeat W - 8, D[15:8]},
    2      : {D[23] repeat W - 8, D[23:16]},
    3      : {D[31] repeat W - 8, D[31:24]},
    default: 'x,
  },
  3'b100 : case addr[1:0] {
    0      : {1'b0 repeat W - 8, D[7:0]},
    1      : {1'b0 repeat W - 8, D[15:8]},
    2      : {1'b0 repeat W - 8, D[23:16]},
    3      : {1'b0 repeat W - 8, D[31:24]},
    default: 'x,
  },
  3'b001 : case addr[1] {
    0      : {D[15] repeat W - 16, D[15:0]},
    1      : {D[31] repeat W - 16, D[31:16]},
    default: 'x,
  },
  3'b101 : case addr[1] {
    0      : {1'b0 repeat W - 16, D[15:0]},
    1      : {1'b0 repeat W - 16, D[31:16]},
    default: 'x,
  },
  3'b010 : D,
  default: 'x,
};
```

3.11.3 SB, SH 命令の実装

次に、SB, SH 命令を実装します。

memory モジュールで書き込みマスクをサポートする

memory モジュールは、32 ビット単位の読み書きしかサポートしておらず、一部の書き込みもサポートしていません。本書では、一部のみ書き込む命令を memory モジュールでサポートすることで、SB, SH 命令を実装します。

まず、membus_if インターフェースに、書き込む場所をバイト単位で示す信号 `wmask` を追加します。`wmask` には、書き込む部分を 1、書き込まない部分を 0 で指定します。このような挙動をする値を、書き込みマスクと呼びます。

▼ リスト 3.61: wmask の定義 (membus_if.veryl)

```
var wmask : logic <4>;
```

▼ リスト 3.62: modport master に wmask を追加する (membus_if.veryl)

```
wmask : output,
```

▼ リスト 3.63: modport slave に wmask を追加する (membus_if.veryl)

```
wmask : input ,
```

バイト単位で指定するため、wmask の幅は 4 ビットです。

次に、memory モジュールで書き込みマスクをサポートします。

▼ リスト 3.64: 書き込みマスクをサポートする memory モジュール (memory.veryl)

```
import eei::*;

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

    var mem: UInt32 [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];
    }

    // 書き込みマスクをUInt32に展開した値
    var wmask_expand: UInt32;
    for i in 0..32 :wm_expand_block {
        assign wmask_expand[i] = wmask_saved[i / 8];
    }

    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);
        }
    }

    // 状態
    enum State {
        Ready,
        WriteValid,
    }
    var state: State;

    var addr_saved : Addr          ;
```

```

var wdata_saved: UInt32 ;
var wmask_saved: logic <4>;
var rdata_saved: UInt32 ;

always_comb {
    membus.ready = state == State::Ready;
}

always_ff {
    if state == State::WriteValid {
        mem[addr_to_memaddr(addr_saved)] = wdata_saved & wmask_expand | rdata_saved & ~wmask_expand;
    }
}

always_ff {
    if_reset {
        state          = State::Ready;
        membus.rvalid = 0;
        membus.rdata  = 0;
        addr_saved    = 0;
        wdata_saved   = 0;
        wmask_saved   = 0;
        rdata_saved   = 0;
    } else {
        case state {
            State::Ready: {
                membus.rvalid = membus.valid & !membus.wen;
                membus.rdata  = mem[addr_to_memaddr(membus.addr)];
                addr_saved    = membus.addr;
                wdata_saved   = membus.wdata;
                wmask_saved   = membus.wmask;
                rdata_saved   = mem[addr_to_memaddr(membus.addr)];
                if membus.valid && membus.wen {
                    state = State::WriteValid;
                }
            }
            State::WriteValid: {
                state          = State::Ready;
                membus.rvalid = 1;
            }
        }
    }
}
}

```

書き込みマスクをサポートする memory モジュールは、次の 2 つの状態を持ちます。

State::Ready

要求を受け付ける。読み込み要求のとき、次のクロックで結果を返す。書き込み要求のとき、要求の内容をレジスタに保存し、状態を `State::WriteValid` に移動する。

State::WriteValid

書き込みマスク付きの書き込みを行う。状態を **State::Ready** に移動する。

memory モジュールは、書き込み要求が送られてきた場合、名前が **_saved** で終わるレジスタに要求の内容を保存します。また、**rdata_saved** に、指定されたアドレスのデータを保存します。次のクロックで、書き込みマスクを使った書き込みを行い、要求の処理を終了します。

top モジュールの調停処理で、**wmask** も調停するようにします。

▼ リスト 3.65: wmask の設定 (top.veryl)

```
membus.valid = i_membus.valid | d_membus.valid;
if d_membus.valid {
    membus.addr  = d_membus.addr;
    membus.wen   = d_membus.wen;
    membus.wdata = d_membus.wdata;
    membus.wmask = d_membus.wmask; ←追加
} else {
    membus.addr  = i_membus.addr;
    membus.wen   = i_membus.wen;
    membus.wdata = i_membus.wdata;
    membus.wmask = i_membus.wmask; ←追加
}
```

memunit モジュールの実装

memory モジュールが書き込みマスクをサポートするようになったので、memunit モジュールで wmask を設定します。

req_wmask レジスタを作成し、**membus.wmask** と接続します。

▼ リスト 3.66: req_wmask の定義 (memunit.veryl)

```
var req_wmask: logic<4>;
```

▼ リスト 3.67: membus に wmask を設定する (memunit.veryl)

```
membus.wmask = req_wmask;
```

always_ff の中で、req_wmask の値を設定します。それぞれの命令のとき、wmask がどうなるかを確認してください。

▼ リスト 3.68: if_reset で req_wmask を初期化する (memunit.veryl)

```
req_wmask = 0;
```

▼ リスト 3.69: メモリにアクセスする命令のとき、wmask を設定する (memunit.veryl)

```
req_wmask = case ctrl.func3[1:0] {
    2'b00 : 4'b1 << addr[1:0], ← LB, LBUのとき、アドレス下位2ビット分だけ1を左シフトする
    2'b01 : case addr[1:0] { ← LH, LHU命令のとき
        2      : 4'b1100, ←上位2バイトに書き込む
        0      : 4'b0011, ←下位2バイトに書き込む
    }
}
```



```

        default: 'x,
    },
    2'b10 : 4'b1111, ← LW命令のとき、全体に書き込む
    default: 'x,
};

```

3.11.4 LB, LBU, LH, LHU, SB, SH 命令のテスト

簡単なテストを作成し、動作をテストします。

2つテストを記載するので、正しく動いているか確認してください。

▼ リスト 3.70: src/sample_lbh.hex

```

02000083 // lb x1, 0x20(x0) : x1 = ffffffffef
02104083 // lbu x1, 0x21(x0) : x1 = 000000be
02201083 // lh x1, 0x22(x0) : x1 = ffffdead
02205083 // lhu x1, 0x22(x0) : x1 = 0000dead
00000000
00000000
00000000
00000000
deadbeef // 0x0

```

▼ リスト 3.71: src/sample_sbsh.hex

```

12300093 // addi x1, x0, 0x123
02101023 // sh x1, 0x20(x0)
02100123 // sb x1, 0x22(x0)
02200103 // lb x2, 0x22(x0) : x2 = 00000023
02001183 // lh x3, 0x20(x0) : x3 = 00000123

```

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 Vexl 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

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

あとがき / おわりに

いかがだったでしょうか。感想や質問は随時受け付けています。

著者紹介

kanataso

X(Twitter) : @kanapipopipo GitHub : @nananapo

Veryl で作る RISC-V CPU

基本編

2024 年 11 月 2 日 ver 1.0 (技術書典 11)

著 者 kanataso

印刷所 日光企画

© 2024 カウプラン機関極東支部