

Veryl で作る CPU

— 基本編 第 I 部 RV32I/RV64I の実装 —

[著] 阿部奏太

技術書典 17 (2024 年秋) 新刊

2024 年 11 月 3 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

こんにちは! あなたは CPU を自作したことがありますか? 自作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。

パソコンの主要な部品である CPU は、とても規模が大きくて複雑な電子回路で構成されています。現代的で、高速で、ゲームができるような CPU を作るのは非常に難しいです。ですが、ちょっと遅くて機能が少ない CPU なら、誰でも簡単に作ることができます。

本書では、**Veryl** という「ハードウェアを記述するための言語」で CPU を自作する方法を解説しています。Veryl を使うと、例えば 32 ビットの足し算をする回路を次のように記述できます。

```
module Adder (
    x  : input  logic<32>,
    y  : input  logic<32>,
    sum: output logic<32>,
) {
    always_comb {
        sum = x + y;
    }
}
```

`x` と `y` を受け取って、`sum` に `x + y` を割り当てるだけ.... 簡単ですね。これと同様に、入出力を書いて、足し算や AND、OR、NOT 演算などを書くだけで、CPU を書くことができます。

CPU は大きな論理回路です。でも、Veryl で書いた小さな部品を組み合わせていけば、簡単に作ることができます。

あなたも CPU を自作してみませんか? 自分好みの CPU を作る第一歩を踏み出しましょう。

本書を読むとわかるこ

- CPU の仕組み、動作、実装
- Veryl の基本文法
- Veryl での CPU の実装方法
- RISC-V の基本整数命令セット

対象読者

- 自作 CPU に興味がある人
- コンピュータアーキテクチャに興味がある人
- Veryl が気になっている人

必要な知識

- 基本的な論理演算 (AND、OR、NOT くらいしか使いません)
- C、C++、JavaScript、Python、Ruby、Rust のような一般的なプログラミング言語の経験

本書では、Veryl の他に C++, Python, Makefile, シェルスクリプトを使用します。Veryl については詳細を解説しています。他の言語については、動作とどのような機能を持つかは解説しますが、言語の仕様や書き方、ライブラリなどは説明しません。

本書のソースコード / 問い合わせ先

本書で利用するソースコードは、以下のサポートページから入手できます。質問やお問い合わせ方法についてもサポートページを確認してください。

- <https://github.com/nananapo/veryl-riscv-book/wiki/techbookfest17-support-page>

注意

本書は「Veryl で作る CPU 基本編」の第 I 部のみを発行したものです。本文に「後の章」と書かれても、本書には含まれない場合があります。完全版および電子版は無料で配布されており、<https://github.com/nananapo/veryl-riscv-book> で入手できます。

CPU の自作

CPU って自作できるのでしょうか？ そもそも CPU の自作って何でしょうか？ CPU の自作の一般的な定義はありませんが、筆者は「命令セットアーキテクチャの設計」「論理設計」「物理的に製造する」に分類できると考えています。

命令セットアーキテクチャ (Instruction Set Architecture, ISA) とは、CPU がどのような命令を実行できるかを定めたもの (仕様) です。論理設計とは、簡単に言うと、仕様の動作を実現する論理回路を設計することです。CPU は論理回路で構成されているため、CPU の設計には論理設計が必要になります。最近の CPU は、物理的には VLSI (Very Large Scale Integration, 超大規模集積回路) によって実装されています。VLSI の製造には莫大なお金が必要です¹。

ISA と実装 (設計と製造) には深い関わりがあるため、ISA を知らずして実装はできないし、実装を知らずして ISA を作ることはできません。本書では、RISC-V という ISA の CPU を実装することで、一般的な CPU の作り方やアーキテクチャについて学びます。

物理的な製造のハードルは高いですが、FPGA を使うことで簡単にお試しできます。FPGA (Field Programmable Gate Array) とは、任意の論理回路を実現できる集積回路のことです [1]。最近では、安価 (数千～数万円) で FPGA を入手できます。

CPU のテストはシミュレータと FPGA で行います。本書では、TangMega 138K と PYNQ-Z1 という FPGA を利用します。FPGA を持っていると、自作 CPU によって LED を制御したり、手持ちのパソコンと直接通信したりして楽しむことができます。

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 ISA です。仕様書の初版は 2011 年に公開されました。ISA としての歴史はまだ浅いですが、仕様が広く公開されていてカスタマイズ可能であるという特徴もあって、着実に広がりつつあります。

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、RocketChip² (Chisel による実装)、Shakti³ (Bluespec SystemVerilog による実装)、RSD⁴ (SystemVerilog による実装) が挙げられます。

本書では、RISC-V のバージョン RISC-V ISA Manual, version 20240411 を利用します。RISC-V の最新の仕様は、GitHub の [riscv/riscv-isa-manual](https://github.com/riscv/riscv-isa-manual)⁵ で確認できます。

RISC-V には、基本整数命令セットとして RV32I、RV64I、RV32E、RV64E⁶ 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が何ビットかです。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。それ以外のかけ算や割り算、不可分操作などの命令や機能は拡張

¹ 小さいチップなら安く (数万～数百万円で) 製造できます。OpenMPW や TinyTapeout で検索してください

² <https://github.com/chipsalliance/rocket-chip>

³ <https://shakti.org.in/>

⁴ <https://github.com/rsd-devel/rsd>

⁵ <https://github.com/riscv/riscv-isa-manual/>

⁶ RV128I もありますが、まだ Draft 段階 (準備中) です

として定義されています。

どの拡張を実装しているかを示す ISA String という表現では、例えばかけ算と割り算、不可分操作ができる RV32I の CPU は **RV32IM** と表現されます。本書では、まず、**RV32I** の CPU を実装します。これを、OS を実行できる程度までに進化させることを目標に実装を進めます。

本書の構成

本シリーズ (基本編) では、次のように CPU を実装していきます。

1. RV32I の CPU を実装する (第 3 章)
2. Zicsr 拡張を実装する (第 4 章)
3. CPU をテストする (第 5 章)
4. RV64I を実装する (第 6 章)
5. パイプライン化する (第 7 章)
6. 実機でテストする (第 8 章)
7. M 拡張、A 拡張、C 拡張を実装する
8. UART と割り込みを実装する
9. OS を実行するために必要な CSR を実装する
10. OS を実行する

本書 (基本編の第 I 部) では、上の 1 から 6 までを実装、解説します。

凡例

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。ただし、リスト内のコードが全て新しく追加されるときは太字を利用しません。コードを置き換えるときは太字で示し、削除されたコードを示さない場合もあります。

```
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_123456789_>9_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
```

プログラムコードやターミナル画面は、**...**などの複数の点で省略することができます。

目次

まえがき	i
第1部 RV32I/RV64I の実装	1
第1章 環境構築	2
1.1 Veryl	2
1.2 Verilator	4
1.3 riscv-gnu-toolchain	4
第2章 ハードウェア記述言語 Veryl	5
2.1 ハードウェア記述言語	5
2.1.1 論理回路の構成	5
2.1.2 ハードウェア記述言語	6
2.1.3 Veryl	7
2.2 Veryl の基本文法、機能	8
2.2.1 コメント	8
2.2.2 値、リテラル	9
2.2.3 module	10
2.2.4 ユーザー定義型	16
2.2.5 式、文、宣言	18
2.2.6 interface	23
2.2.7 package	25
2.2.8 ジェネリクス	26
2.2.9 その他の機能、文	27
第3章 RV32I の実装	30
3.1 CPUは何をやっているのか?	30
3.2 プロジェクトの作成	32
3.3 定数の定義	33
3.4 メモリ	34
3.4.1 メモリのインターフェースを定義する	34
3.4.2 メモリモジュールを実装する	35
3.4.3 メモリの初期化、環境変数の読み込み	37
3.5 最上位モジュールの作成	37
3.6 命令フェッチ	38

3.6.1	命令フェッチを実装する	39
3.6.2	memory モジュールと core モジュールを接続する	40
3.6.3	命令フェッチをテストする	41
3.6.4	フェッチした命令を FIFO に格納する	46
3.7	命令のデコードと即値の生成	51
3.7.1	デコード用の定数と型を定義する	53
3.7.2	制御フラグと即値を生成する	54
3.7.3	デコーダをインスタンス化する	56
3.8	レジスタの定義と読み込み	57
3.8.1	レジスタファイルを定義する	57
3.8.2	レジスタの値を読み込む	57
3.9	ALU による計算の実装	59
3.9.1	ALU モジュールを作成する	60
3.9.2	ALU モジュールをインスタンス化する	61
3.9.3	ALU モジュールをテストする	63
3.10	レジスタに結果を書き込む	65
3.10.1	ライトバック処理を実装する	65
3.10.2	ライトバック処理をテストする	65
3.11	ロード命令とストア命令の実装	67
3.11.1	LW、SW 命令を実装する	67
3.11.2	LB、LBU、LH、LHU 命令を実装する	76
3.11.3	SB、SH 命令を実装する	77
3.11.4	LB、LBU、LH、LHU、SB、SH 命令をテストする	81
3.12	ジャンプ命令、分岐命令の実装	82
3.12.1	JAL、JALR 命令を実装する	82
3.12.2	条件分岐命令を実装する	86
第4章	Zicsr 拡張の実装	89
4.1	CSR とは何か?	89
4.2	CSR 命令のデコード	90
4.3	csrunit モジュールの実装	91
4.3.1	csrunit モジュールを作成する	91
4.3.2	mtvec レジスタを実装する	93
4.3.3	csrunit モジュールをテストする	95
4.4	ECALL 命令の実装	96
4.4.1	ECALL 命令とは何か?	96
4.4.2	トラップを実装する	97
4.4.3	ECALL 命令をテストする	101
4.5	MRET 命令の実装	103
4.5.1	MRET 命令を実装する	103

4.5.2	MRET 命令をテストする	104
第 5 章	riscv-tests によるテスト	105
5.1	riscv-tests とは何か?	105
5.2	riscv-tests のビルド	105
5.2.1	riscv-tests をビルドする	105
5.2.2	成果物を\$readmemh で読み込める形式に変換する	106
5.3	テスト内容の確認	108
5.4	テストの終了検知	110
5.5	テストの実行	110
5.6	複数のテストの自動実行	111
第 6 章	RV64I の実装	115
6.1	XLEN の変更	116
6.1.1	SLL[I]、SRL[I]、SRA[I] 命令を変更する	116
6.1.2	LUI、AUIPC 命令を変更する	116
6.1.3	CSR を変更する	116
6.1.4	LW 命令を変更する	117
6.1.5	riscv-tests でテストする	117
6.2	ADD[I]W、SUBW 命令の実装	119
6.2.1	ADD[I]W、SUBW 命令をデコードする	119
6.2.2	ALU に ADDW、SUBW を実装する	120
6.2.3	ADD[I]W、SUBW 命令をテストする	121
6.3	SLL[I]W、SRL[I]W、SRA[I]W 命令の実装	122
6.3.1	SLL[I]W、SRL[I]W、SRA[I]W 命令をテストする	122
6.4	LWU 命令の実装	123
6.4.1	LWU 命令をテストする	124
6.5	LD、SD 命令の実装	124
6.5.1	メモリの幅を広げる	124
6.5.2	命令フェッチ処理を修正する	124
6.5.3	SD 命令を実装する	125
6.5.4	LD 命令を実装する	126
6.5.5	LD、SD 命令をテストする	127
第 7 章	CPU のパイプライン化	128
7.1	CPU の速度	128
7.1.1	CPU の性能を考える	128
7.1.2	実行速度を上げる方法を考える	129
7.1.3	パイプライン処理のステージを考える	130
7.2	パイプライン処理の実装	132

7.2.1	ステージに分割する準備をする	132
7.2.2	FIFO を作成する	133
7.2.3	IF ステージを実装する	136
7.2.4	ID ステージを実装する	137
7.2.5	EX ステージを実装する	137
7.2.6	MEM ステージを実装する	139
7.2.7	WB ステージを実装する	142
7.2.8	デバッグのために情報を表示する	143
7.2.9	パイプライン処理をテストする	144
7.3	データ依存の対処	144
7.3.1	正しく動かないプログラムを確認する	144
7.3.2	データ依存とは何か？	145
7.3.3	データ依存に対処する	146
7.3.4	パイプライン処理をテストする	147
第8章 CPU を合成する		149
あとがき		150
参考文献		151

第Ⅰ部

RV32I/RV64I の実装

第 1 章

環境構築

本書で使用するソフトウェアをインストールします。WSL が使える Windows、Mac、Linux のいずれかの環境を用意してください。

1.1 Veryl

Veryl のインストール

本書では Veryl という言語で CPU を記述します。まず、Veryl のトランスペイラをインストールします。Veryl には、Verylup というインストーラが用意されており、これを利用することで Veryl をインストールできます。

Verylup は GitHub の Release ページから入手できます。veryl-lang/verylup^{*1} で入手方法を確認してください^{*2}。Verylup を入手したら、次のように Veryl の最新版をインストールします（リスト 1.1）。

▼ リスト 1.1: Veryl のインストール

```
$ verylup setup
[INFO ]  downloading toolchain: latest
[INFO ]  installing toolchain: latest
[INFO ]    creating hardlink: veryl
[INFO ]    creating hardlink: veryl-ls
```

Veryl の更新

最新の Veryl に更新するには、次のようなコマンドを実行します（リスト 1.2）。

▼ リスト 1.2: Veryl の更新

```
$ verylup update
```

*1 <https://github.com/veryl-lang/verylup>

*2 cargo が入っている方は、`cargo install verylup` でもインストールできます。

インストールするバージョンの指定

特定のバージョンの Veryl をインストールするには、次のようなコマンドを実行します（リスト 1.3）。

▼リスト 1.3: Veryl のバージョン 0.13.1 をインストールする

```
$ verylup install 0.13.1
```

インストールされているバージョン一覧は次のように確認できます（リスト 1.4）。

▼リスト 1.4: インストール済みの Veryl のバージョン一覧を表示する

```
$ verylup show
installed toolchains
-----
0.13.1
0.13.2
latest (default)
```

使用するバージョンの指定

バージョンを指定しない場合は、最新版の Veryl が使用されます（リスト 1.5）。

▼リスト 1.5: veryl のバージョン確認

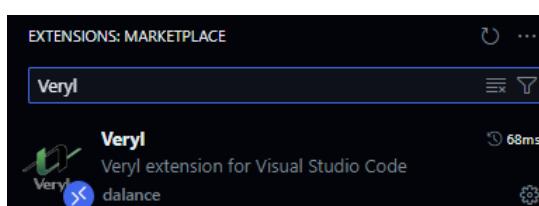
```
$ veryl --version
veryl 0.13.2
```

特定のバージョンの Veryl を使用するには `+` でバージョンを指定します（リスト 1.6）。

▼リスト 1.6: Veryl のバージョン 0.13.2 を使用する

```
$ veryl +0.13.2 ← +でバージョンを指定する
```

エディタ拡張のインストール



▲図 1.1: Veryl の VSCode 拡張

エディタに VSCode を利用している方は、図 1.1 の拡張をインストールするとシンタックスハイライトなどの機能を利用できます。

- <https://marketplace.visualstudio.com/items?itemName=dalance.vscode-veryl>

エディタに Vim を利用している方は、GitHub の [veryl-lang/veryl.vim](https://github.com/veryl-lang/veryl.vim)^{*3} でプラグインを入手できます。

1.2 Verilator

Verilator^{*4}は、SystemVerilog のシミュレータを生成するためのソフトウェアです。

パッケージマネージャ (apt、Homebrew など) を利用してインストールできます。パッケージマネージャが入っていない場合は、以下のページを参考にインストールしてください。

- <https://verilator.org/guide/latest/install.html>



本書で利用する Verilator のバージョン

2024/10/28 時点の最新バージョンは v5.030 ですが、Verilator の問題によりシミュレータをビルドできない場合があります。対処方法はサポートページを確認してください。

1.3 riscv-gnu-toolchain

riscv-gnu-toolchain は、RISC-V 向けのコンパイラやシミュレータなどが含まれているツールチェーン (ソフトウェア群) です。

GitHub の [riscv-collab/riscv-gnu-toolchain](https://github.com/riscv-collab/riscv-gnu-toolchain)^{*5} の README にインストール方法が書かれています。README の `Installation (Newlib)` を参考にインストールしてください。

^{*3} <https://github.com/veryl-lang/veryl.vim>

^{*4} <https://github.com/verilator/verilator>

^{*5} <https://github.com/riscv-collab/riscv-gnu-toolchain>

第 2 章

ハードウェア記述言語 Veril

CPU (Central Processing Unit, 中央演算処理装置) は、コンピュータを構成する主要な部品の1つであり、電気で動くとても複雑な回路で構成されています。

本書では「ハードウェア記述言語」によって CPU をの回路を記述します。回路を記述するといつても、いったい何をどうやって記述するのでしょうか?

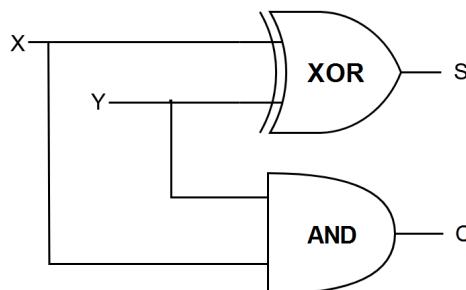
まずは、論理回路を構成する方法から考えます。

2.1 ハードウェア記述言語

2.1.1 論理回路の構成

論理回路とは、デジタル(例えば0と1だけ)なデータを利用して、データを加工、保持する回路のことです。論理回路は、組み合わせ回路と順序回路に分類できます。

組み合わせ回路とは、入力に対して、一意に出力の決まる回路[2]のことです。例えば、1ビット同士の加算をする回路は図2.1、表2.1のよう表されます。この回路は半加算器と呼ばれていて、1ビットのXとYを入力として受けとり、1ビットの和Sと桁上げCを出力します。入力(X, Y)が決まると出力(C, S)が一意に決まるため、半加算器は組み合わせ回路です。



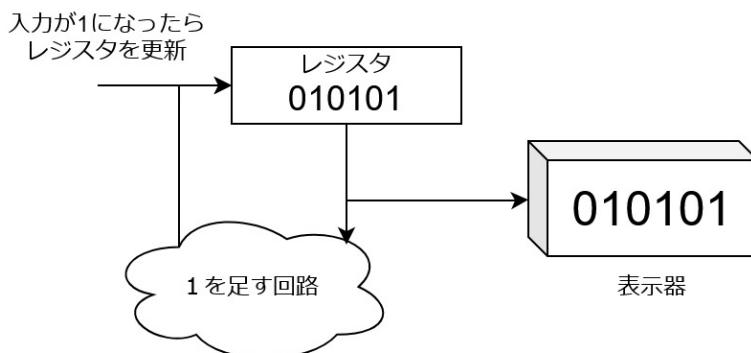
▲図2.1: 半加算器(MIL記法の回路図)

▼表 2.1: 半加算器 (真理値表)

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

順序回路とは、入力と回路自身の状態によって一意に出力の決まる回路 [2] です。例えば、入力が 1 になるたびにカウントアップして値を表示するカウンタを考えます (図 2.2)。カウントアップするためには、今のカウンタの値 (状態) を保持する必要があります。そのため、このカウンタは入力と状態によって一意に出力の決まる順序回路です。

1 ビットの値はフリップフロップ (flip-flop, FF) という回路によって保持できます。フリップフロップを N 個並列に並べると、N ビットの値を保持できます。フリップフロップを並列に並べた記憶装置のことをレジスタ (register, 置数器) と呼びます。基本的に、レジスタの値はリセット信号 (reset signal, reset) によって初期化し、クロック信号 (clock signal, clock) に同期したタイミングで変更します。



▲図 2.2: カウンタ (順序回路の例)

論理回路を設計するには、真理値表を作成し、それを実現する論理演算を構成します。入力数や状態数が数十個ならどうにか人力で設計できるかもしれません、数千、数万の入力や状態があるとき、手作業で設計するのはほとんど不可能です。これを設計するために、ハードウェア記述言語を利用します。

2.1.2 ハードウェア記述言語

ハードウェア記述言語 (Hardware Description Language, HDL) とは、デジタル回路を設計するための言語です。例えば HDL である SystemVerilog を利用すると、半加算器はリスト 2.1 のように記述できます。

▼リスト 2.1: SystemVerilog による半加算器の記述

```
module HalfAdder(
    input logic x,          // 入力値X
    input logic y,          // 入力値Y
    output logic c,         // 出力値C
    output logic s          // 出力値S
);
    assign c = x & y; // &はAND演算
    assign s = x ^ y; // ^はXOR演算
endmodule
```

半加算器 (HalfAdder) モジュールは、入力として `x` と `y` を受け取り、出力 `c` と `s` に `x` と `y` を使った演算を割り当てています。

また、レジスタを利用した回路をリスト 2.2 のように記述できます。レジスタの値を、リセット信号 `rst` が `0` になったタイミングで `0` に初期化し、クロック信号 `clk` が `1` になったタイミングでカウントアップします。

▼リスト 2.2: SystemVerilog によるカウンタの記述

```
module Counter(
    input logic clk, // クロック信号
    input logic rst // リセット信号
);
    // 32ビットのレジスタの定義
    logic [31:0] count;

    always_ff @(posedge clk, negedge rst) begin
        if (!rst) begin
            // rstが0になったとき、countを0に初期化する
            count <= 0;
        end else begin
            // clkが1になったとき、countの値をカウントアップする
            count <= count + 1;
        end
    end
endmodule
```

HDL を使用した論理回路の設計は、レジスタの値と入力値を使った組み合わせ回路と、その結果をレジスタに格納する操作の記述によって行えます。このような、レジスタからレジスタに、組み合わせ回路を通したデータを転送する抽象度のことをレジスタ転送レベル (Register Transfer Level, RTL) と呼びます。

HDL で記述された RTL を実際の回路のデータに変換することを合成と呼びます。合成するソフトウェアのことを合成系と呼びます。

2.1.3 Veril

メジャーな HDL といえば、Verilog HDL、SystemVerilog、VHDL などが挙げられます。

Verilog HDL(Verilog) と VHDL は 1980 年代に開発された言語であり、最近のプログラミング

言語と比べると機能が少なく、冗長な記述が必要です。SystemVerilog は Verilog のスーパーセットです。言語機能が増えて便利になっていますが、スーパーSETTであることから、あまり推奨されない古い書き方が可能だったり、バグの原因となるような良くない仕様^{*1}を受け継いでいます。

本書では、CPU の実装に Veryl という HDL を使用します。Veryl は 2022 年 12 月に公開された言語です。Veryl の抽象度は、Verilog と同じくレジスタ転送レベルです。Veryl の文法や機能は、Verilog や SystemVerilog に似通ったものになっています。しかし、if 式や case 式、クロックとリセットの抽象化、ジェネリクスなどの痒い所に手が届く機能が提供されており、高い生産性を発揮します。

Veryl のソースコードはコンパイラ（トランスペイラ）によって、自然で読みやすい SystemVerilog のソースコードに変換されます。そのため、Veryl は旧来の SystemVerilog の環境と共に存でき、SystemVerilog の資産を利用できます。



注意

本書は 2024/11/3 時点の Veryl(バージョン 0.13.2) を、本書で利用する範囲の文法と機能を解説しています。Veryl はまだ開発中(安定版がリリースされていない) 状態の言語であるため、破壊的変更が入り、記載しているコードが使えなくなる可能性があります。

2.2 Veryl の基本文法、機能

それでは、Veryl の書き方を学んでいきましょう。Veryl のドキュメントは <https://doc.veryl-lang.org/book/ja/> に存在します。また、Veryl Playground^{*2}では、Veryl の SystemVerilog へのトランスペイルをウェブブラウザ上でお試しできます。

2.2.1 コメント

Veryl では次のようにコメントを記述できます（リスト 2.3）。

▼リスト 2.3: コメント

```
// 1行のコメント
/* 範囲コメント */
/*
    範囲コメントは改行してもOK
*/
```

^{*1} 例えば、未定義の変数が 1 ビット幅の信号線として解釈される仕様があります

^{*2} <https://doc.veryl-lang.org/playground/>

2.2.2 値、リテラル

論理回路では、デジタルな値を扱います。デジタルな値は `0` と `1` の二値 (2-state) で表現されますが、一般的なハードウェア記述言語では、`0` と `1` に `x` と `z` を加えた四値 (4-state) が利用されます (表 2.2)。

▼表 2.2: 4-state の値

値	意味	真偽
0	0	偽
1	1	真
x	不定値	偽
z	ハイインピーダンス	偽

不定値 (unknown value, `x`) とは、`0` か `1` のどちらか分からぬ値です。不定値は、未初期化のレジスタの値の表現に利用されたり、不定値との演算の結果として生成されます。**ハイインピーダンス** (high-impedance, `z`) とは、どのレジスタや信号とも接続されていないことを表す値です。物理的なハードウェア上では、全ての値は `0` か `1` の二値として解釈されますが、信号の状態としてハイインピーダンスを持ちます。不定値はシミュレーションのときに利用します。

1 ビットの四値を表現するための型は `logic` です。N ビットの `logic` 型は `logic<N>` と記述できます。1 ビットの二値を表現する型は `bit` です。基本的に、レジスタや信号の定義に `bit` 型は利用せず、`logic` 型を利用します。

`logic` 型と `bit` 型は、デフォルトで符号が無い型として扱われます。符号付き型として扱いたいときは、型名の前に `signed` キーワードを追加します (リスト 2.4)。

▼リスト 2.4: 符号付き型

```
signed logic<4> // 4ビットの符号付きlogic型
signed bit<2> // 2ビットの符号付きbit型
```

32 ビットと 64 ビットの `bit` 型を表す型が定義されています (表 2.3)。

▼表 2.3: 整数型

型名	等価な型
u32	<code>bit<32></code>
u64	<code>bit<64></code>
i32	<code>signed bit<32></code>
i64	<code>signed bit<64></code>

数値はリスト 2.5 のように記述できます。

▼リスト 2.5: 数値リテラル

```

4'b0101 // 4ビットの数値 (2進数表記)
4'bxxzz // 4ビットの数値 (2進数表記)

12'o34xz // 12ビットの数値 (8進数表記)
32'h89abcdef // 32ビットの数値 (16進数表記)

123 // 10進数の数値
32'd12345 // 32ビットの数値 (10進数表記)

// 数値リテラルの好きな場所に_を挿入できる
1_2_34_567

// xとzは大文字でも良い
4'bxXzZ

// 全ビット0、1、x、zにする
'0
'1
'x
'z

// 指定したビット幅だけ0、1、x、zにする
8'0 // 8ビット0
8'1 // 8ビット1
8'x // 8ビットx
8'z // 8ビットz

// 幅を指定しない場合、幅が自動で推定される
'hffff // 16ビット
'h1fff // 13ビット (13'b1_1111_1111_1111)

```

文字列は **string** 型で表現できます。文字列の値はリスト 2.6 のように記述できます。

▼リスト 2.6: 文字列リテラル

```

"Hello World!" // 文字列リテラル
"abcdef\nabc" // エスケープシーケンスを含む文字列リテラル

```

2.2.3 module

論理回路はモジュール (Module) というコンポーネントで構成されます。例えば、半加算器のモジュールは次のように定義できます (リスト 2.7)。

▼リスト 2.7: 半加算器 (HalfAdder) モジュール

```

module HalfAdder (
    x: input logic, // 1ビットのlogic型の入力
    y: input logic, // 1ビットのlogic型の入力
    s: output logic, // 1ビットのlogic型の出力
    c: output logic, // 1ビットのlogic型の出力
) {

```

```

always_comb {
    s = x ^ y; // sにx XOR yを代入
    c = x & y; // cにx AND yを代入
}

```

HalfAdder モジュールには、入力変数として `x` と `y`、出力変数として `s` と `c` が宣言されています。入出力の変数のことを接続ポート、または単にポートと呼びます。

入力ポートを定義するとき、モジュール名の後の括弧の中に、`変数名 : input 型名` と記述します。出力ポートを宣言するときは `input` の代わりに `output` と記述します。複数のポートを宣言するとき、宣言の末尾にカンマ (,) を記述します。

変数のブロッキング代入

HalfAdder モジュールでは、`always_comb` ブロックの中で出力変数 `s` と `c` に値を代入しています。変数への代入は `変数名 = 式;` で行います。`always_comb` ブロック内での代入のことを、ブロッキング代入 (blocking assignment) と呼びます。

通常のプログラミング言語での代入とは、スタック領域やレジスタに存在する変数に値を格納することです。これに対して `always_comb` ブロック内での代入は、式が評価 (計算) された値が変数に 1 度だけ代入されるのではなく、変数の値は常に式の計算結果になります。

具体例で考えます。例えば `always_comb` ブロックの中で、1 ビットの変数 `x` に 1 ビットの変数 `y` を代入します (リスト 2.8)。

▼リスト 2.8: x に y を割り当てる

```

always_comb {
    x = y;
}

```

`y` の値が時間経過により `0 → 1 → 0 → 1 → 0` と変化したとします。このとき、`x` の値は `y` が変わると同時に変化します (図 2.3)。図 2.3 は、時間を横軸、`x` と `y` の値を線の高低で表しています。図 2.3 のような図を波形図 (waveform)、または単に波形と呼びます。

`x` に `y` ではなく `a + b` を代入すると、`a` か `b` の変化をトリガーに `x` の値が変化します。



▲図 2.3: x は y の値の変化に追従する

always_comb ブロックには複数の代入文を記述できます。このとき、代入文は上から順番に実行(逐次実行)されます。

▼リスト 2.9: ブロッキング代入は逐次実行される

```
always_comb {
    s = X;
    a = s; // a = X
    s = Y;
    b = a; // b = Y
}
```

例えばリスト 2.9 では、`a` には `X` が代入されますが、`b` には `Y` が代入されます。変数 `a` と `b` と `s` は、変数 `X` か `Y` の変化をトリガーに値が更新されます。

1 つの変数にしかブロッキング代入しないとき、`assign` 文でもブロッキング代入できます(リスト 2.10)。

▼リスト 2.10: `assign` 文によるブロッキング代入

```
// assign 変数名 = 式;
assign a = b + 100;
```

always_comb ブロック内の代入と同じように、リスト 2.10 では `b` の変化をトリガーに `a` の値が変化します。

ブロッキング代入は論理回路の状態(レジスタ)を変更しません。そのため、ブロッキング代入文は組み合わせ回路になります。

変数の宣言

モジュールの中では、`var` 文によって新しく変数を宣言できます(リスト 2.11)。

▼リスト 2.11: 変数の宣言

```
// var 変数名 : 型名;
var value : logic<32>;
```

`var` 文で宣言した変数に対してブロッキング代入できます。

`let` 文を使うと、変数の宣言とブロッキング代入を同時にできます(リスト 2.12)。

▼リスト 2.12: 変数の宣言とブロッキング代入

```
// let 変数名 : 型名 = 式;
let value : logic<32> = 100 + a;
```

レジスタの定義と代入

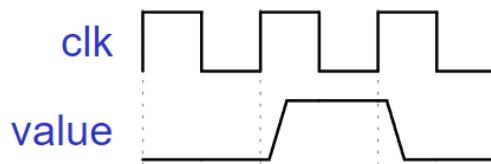
変数を宣言するとき、変数に式がブロッキング代入されない場合、変数はレジスタとして解釈できます(リスト 2.13)。

▼リスト 2.13: レジスタの定義

```
// var レジスタ名 : 型名;
var reg_value : logic<32>;
// reg_valueにブロッキング代入しない
```

本書ではレジスタのことを変数、または変数のことをレジスタと呼ぶことがあります。

レジスタの値はクロック信号に同期したタイミングで変更し、リセット信号に同期したタイミングで初期化します(図 2.4)。本書では、クロック信号が立ち上がる(0 から 1 に変わる)タイミングでレジスタの値を変更し、リセット信号が立ち下がる(1 から 0 に変わる)タイミングでレジスタの値を初期化することとします。



▲図 2.4: レジスタ (value) の値はクロック信号 (clk) が立ち上がるタイミングで変わる

レジスタの値は、`always_ff` ブロックで初期化、変更します(リスト 2.14)。`always_ff` ブロックには、値の変更タイミングのためのクロック信号とリセット信号を指定します。

▼リスト 2.14: レジスタの値の初期化と変更

```
// レジスタの定義
var value : logic<32>;
// always_ff(クロック信号, リセット信号)
always_ff(clk, rst) {
    if_reset {
        // リセット信号のタイミングで0に初期化する
        value = 0;
    } else {
        // クロック信号のタイミングでカウントアップする
        value = value + 1;
    }
}
```

`if_reset` 文の中の文は、リセット信号のタイミングで実行されます。`if_reset` 文に `else` 文を付けることで、クロック信号のタイミングで処理を実行できます。レジスタの値をリセットしない場合、リセット信号と `if_reset` 文を省略することができます。逆に、リセット信号を指定する場合は必ず `if_reset` 文を書かなければいけません。

クロック信号は `clock` 型、リセット信号は `reset` 型で定義します。モジュールのポートに1組のクロック信号とリセット信号が定義されているとき、`always_ff` ブロックのクロック信号とリセット信号の指定を省略できます(リスト 2.15)。

▼リスト 2.15: クロック信号とリセット信号の省略

```
module ModuleA(
  clk: input clock,
  rst: input reset,
){
  // always_ff(clk, rst)と等しい
  always_ff {}
}
```

レジスタの値は、同じタイミングで動く always_ff ブロックの中の全ての代入文の右辺を評価した後に変更されます（リスト 2.16）。この代入はブロッキング代入と違って逐次実行されないので、ノンブロッキング代入（non-blocking assignment）と呼びます。

2つ以上の always_ff ブロックで、1つの同じレジスタの値を変更することはできません。

▼リスト 2.16: 複数のレジスタの値を同じタイミングで変更する

```
// 全ての代入文の右辺を評価した後に、AとBが変更される
// その結果、AとBの値が入れ替わる
always_ff(clk, rst) {
  A = B;
}
always_ff(clk, rst) {
  B = A;
}
```

リスト 2.16 の A と B の代入文は、1つの always_ff ブロックにまとめて記述できます（リスト 2.17）。この場合もリスト 2.16 と同様に、A と B の代入文の右辺を評価した後に、レジスタの値が変更されます。

▼リスト 2.17: ノンブロッキング代入の更新タイミングは同じ

```
always_ff {
  // AとBの値を入れ替える
  A = B;
  B = A;
}
```

本書ではブロッキング代入とノンブロッキング代入を区別せず、どちらも代入と呼ぶことがあります。

変数への代入方法と動作を表 2.4 にまとめます。大変間違えやすいため、気を付けてください。

▼表 2.4: 変数への代入方法と動作の違い

代入場所	代入文の名称	更新タイミング
always_comb	ブロッキング代入	ブロック内の式で参照されている変数が更新されたとき。 上から順に実行される。
always_ff	ノンブロッキング代入	クロック信号、リセット信号のタイミング。 同じタイミングで実行される全ての代入文の右辺を評価した後にレジスタの値が変更される。

モジュールのインスタンス化

あるモジュールを利用したいとき、モジュールをインスタンス化 (instantiate) することにより、モジュールの実体を宣言できます。

モジュールは、**inst** キーワードによってインスタンス化できます (リスト 2.18)。

▼リスト 2.18: ModuleA モジュール内で HalfAdder モジュールをインスタンス化する

```
module ModuleA {
    // モジュールと接続するための変数の宣言
    let x : logic = 0;
    let y : logic = 1;
    var s : logic;
    var c : logic;

    // inst インスタンス名 : モジュール名(ポートとの接続);
    inst ha1 : HalfAdder(
        x: x, // ポートxに変数xを接続する
        y: y,
        s, // ポート名と変数名が同じとき、ポート名の指定を省略できる
        c,
    );
}
```

インスタンス名が違えば、同一のモジュールを 2 つ以上インスタンス化できます。

パラメータ、定数

モジュールには、インスタンス化するときに変更可能な定数 (パラメータ) を用意できます。

モジュールのパラメータは、ポート宣言の前の **#()** の中で **param** キーワードによって宣言できます (リスト 2.19)。

▼リスト 2.19: モジュールのパラメータの宣言

```
module ModuleA #(
    // param パラメータ名 : 型名 = デフォルト値
    param WIDTH : u32 = 100, // u32型のパラメータ
    param DATA_TYPE : type = logic, // type型のパラメータには型を指定できる
) (
    // ポートの宣言
)
```

モジュールをインスタンス化するとき、ポートの割り当てと同じようにパラメータの値を割り当てられます (リスト 2.20)。

▼リスト 2.20: パラメータの値を指定する

```
inst ma : ModuleA #(
    // パラメータの割り当て
    WIDTH: 10,
    DATA_TYPE: logic<10>
) /* ポートの接続 */;
```

パラメータに指定する値は、合成時に確定する値(定数)である必要があります。モジュール内では、変更不可能なパラメータ(定数)を定義できます。定数を定義するには `const` キーワードを使用します(リスト 2.21)。

▼リスト 2.21: 定数の定義

```
// const 定数名 : 型名 = 式;
// 式に変数が含まれてはいけない
const SECRET : u32 = 42;
```

2.2.4 ユーザー定義型

構造体型

構造体(struct)とは、複数のデータから構成される型です。例えば、リスト 2.22 のように記述すると、`logic<32>` と `logic<16>` の 2 つのデータから構成される型を定義できます。

▼リスト 2.22: 構造体型の定義

```
// struct 型名 { フィールドの定義 }
struct MyPair {
    // 名前 : 型
    word: logic<32>,
    half: logic<16>,
}
```

構造体の要素(フィールド, field)には`.`を介してアクセスできます(リスト 2.23)。

▼リスト 2.23: フィールドへのアクセス、割り当て

```
// 構造体型の変数の宣言
var pair: MyPair;

// フィールドにアクセスする
let w : logic<32> = pair.word;

// フィールドに値を割り当てる
always_comb {
    pair.word = 12345;
}
```

列挙型

複数の値の候補から値を選択できる型を作りたいとき、**列挙型**(enumerable type)を利用できます。列挙型の値の候補のことを**バリアント**(variant)と呼びます。

例えば、A、B、C、D のいずれかのバリアントをとる型は次のように定義できます(リスト 2.24)。

▼リスト 2.24: 列挙型の定義

```
// enum 型名 : logic<バリアント数を保持できるだけのビット数> { バリアントの定義 }
enum abc : logic<2> {
```

```
// バリアント名 : バリアントを表す値,
A = 2'd0,
B = 2'd1,
C = 2'd2,
D = 2'd3,
}
```

バリアントを表す値や、バリアントを保持できるだけのビット数は省略できます（リスト 2.25）。

▼ リスト 2.25: 列挙型の省略した定義

```
enum abc {
    A, B, C, D
}
```

配列

`<>` を使用することで、多次元の型を定義できます（リスト 2.26）。`<>` を使用して構成される型の要素は、連続した領域に並ぶことが保証されます（図 2.5）。

▼ リスト 2.26: 多次元の型

```
logic<N>      // Nビットのlogic型
logic<A, B>    // BビットのlogicがA個並ぶ型
```

```
var v : logic<3, 4>;
```

v[2]	v[1]	v[0]
v[2][3] v[2][2] v[2][1] v[2][0]	v[1][3] v[1][2] v[1][1] v[1][0]	v[0][3] v[0][2] v[0][1] v[0][0]

▲ 図 2.5: `<>` の型の要素は連続した領域に並ぶ（例：v[1][0] と v[0][3] が隣り合う）

`[]` を使用することでも、多次元の型を定義できます（リスト 2.27）。ただし、`[]` を使用して構成される型の要素は、連続した領域に並ぶことが保証されません。

▼ リスト 2.27: 配列型

```
// 型名[個数] で、"型名"型が"個数"個の配列になる
logic[32]      // 要素数が32のlogicの配列型
logic[4, 8]    // logicが8個の配列が4個ある配列型
```

型に別名をつける

`type` キーワードを使うと、型に別名を付けられます（リスト 2.28）。

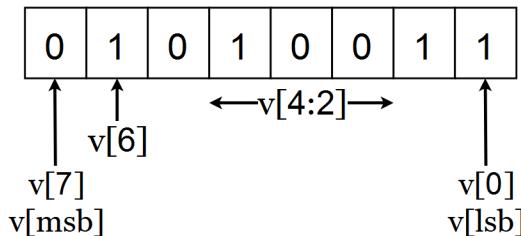
▼ リスト 2.28: 型に別名を付ける

```
// type 名前 = 型;
type ptr = logic<32>;
type ptr_array = ptr<32>;
```

2.2.5 式、文、宣言

ビット選択

```
let v : logic<8> = 8'b01010011;
```



▲図 2.6: ビット選択

変数の任意のビットを切り出すには `[]` を使用します (図 2.6)。範囲の選択には `[:]` を使用します。最上位ビット (most significant bit, MSB) は **msb** キーワード、最下位ビット (least significant bit, LSB) は **lsb** キーワードで指定できます。選択する場所の指定には式を使えます。

よく使われる範囲の選択には、別の書き方が用意されています (リスト 2.29)。

▼リスト 2.29: 範囲の選択の別の記法

```
v[s +: w] // = v[s+w-1 : s]
v[s -: w] // = v[s : s-w+1]
v[i step w] // = v[i*(w+1) : i*w] = v[i*w +: w]
```

演算子

Veril では、表 2.5 の演算子を使用できます。ほとんどの演算子と優先度は通常のプログラミング言語と同じですが、ビット演算の種類が多かったり、`x` と `z` を考慮した演算があるなどの違いがあります。

SystemVerilog との差異を説明すると、`++`、`--`、`:=`、`:/`、`<=` (代入)、`?:` (三項演算子) が無く、`<` と `>` がそれぞれ `<:` と `>:` に変更されています。また、`inside` と `{}` の形式が変更され、if 式、case 式、switch 式が追加されています。

単項、二項演算子の使用例は次の通りです (リスト 2.30)。

▼リスト 2.30: 単項、二項演算子 (Veril のドキュメントの例 [4] を改変)

```
// 単項算術演算
a = +1;
a = -1; // 正負を反転させる

// 単項論理演算
a = !1; // 否定 (真偽を反転させる)
a = ~1; // ビット反転 (0を1、1を0にする)

// 単項集約演算
```

```

// 集約: 左のビットから順に1ビットずつビット演算する
// 例: k = 3'b110のとき、&k = 0
//      &k = 1 & 1 & 0
//      まず、k[msb]とk[1]をANDして1を得る。
//      次に、その結果とk[0]をANDして0を得る。
//      この値が&kの結果になる。
a = &1; // AND
a = |1; // OR
a = ^1; // XOR
a = ~&1; // NAND
a = ~|1; // NOR
a = ~^1; // XNOR
a = ^~1; // XNOR

// 二項算術演算
a = k ** p; // kのp乗
a = 1 * 1; //かけ算
a = 1 / 1; //割り算
a = 1 % 1; // 剰余
a = 1 + 1; // 足し算
a = 1 - 1; // 引き算

// シフト演算
// 注意：右オペランド(シフト数)は符号無しの数として扱われる
a = k << n; // kをnビット左シフトする。空いたビットは0で埋める
a = k <<< n; // <<と同じ
a = k >> n; // kをnビット右シフトする。空いたビットは0で埋める
a = k >>> n; // kが符号無しのとき>>と同じ。符号付きのとき、空いたビットはMSBで埋める

// 比較演算
a = n <: m; // nはm未満
a = n <= m; // nはm以下
a = n >: m; // nはmよりも大きい (mを含まない)
a = n >= m; // nはm以上 (mを含む)
a = n == m; // nはmと等しい (xかzを含む場合、x)
a = n != m; // nはmと等しくない (xかzを含む場合、x)
a = n === m; // nはmと等しい (xとzを含めて完全に一致)
a = n !== m; // nはmと等しくない (xとzを含めて完全に一致)
a = n ==? m; // ==?と同じ。ただし、mに含まれるx,zはワイルドカードになる
a = n !=? m; // !=?と同じ

// ビット演算 (ビット単位, bitwise)
a = 1 & 1; // ビット単位AND
a = 1 ^ 1; // ビット単位XOR
a = 1 ~^ 1; // ビット単位XNOR
a = 1 ~^ 1; // ビット単位XNOR
a = 1 | 1; // ビット単位OR

// 二項論理演算
a = x && y; // xとyの両方が真のとき真
a = x || y; // xまたはyが真のとき真

```

▼表 2.5: 演算子と優先度 [3]

演算子	結合性	優先順位
<code>() [] :: .</code>	左	高い
<code>+ - ! ~ & ~& ~ ^ ~^ ~^ (単項)</code>	左	
<code>**</code>	左	
<code>* / %</code>	左	
<code>+ - (二項)</code>	左	
<code><< >> <<< >>></code>	左	
<code><: <= >: >=</code>	左	
<code>== != === !== ==? !=?</code>	左	
<code>& (二項)</code>	左	
<code>^ ~^ ^~ (二項)</code>	左	
<code> (二項)</code>	左	
<code>&&</code>	左	
<code> </code>	左	
<code>= += -= *= /= %= &= ^= = <=< >>= <<<= >>>=</code>	なし	
<code>{}</code> inside outside if case switch	なし	低い

if、switch、case

条件によって動作や値を変えたいとき、**if** 文を使用します (リスト 2.31)。if 文は式にできます。if 式は必ず値を返す必要があり、else が必須です。

▼リスト 2.31: if 文、if 式

```
var v : logic<32>;
always_comb {
    if WIDTH == 0 {
        // WIDTH == 0のとき
        v = 0;
    } else if WIDTH == 1 {
        // WIDTH != 0かつWIDTH == 1のとき
        v = 1;
    } else {
        // WIDTH != 0かつWIDTH != 1のとき
        v = if WIDTH == 3 { // ifは式にもなる
            3
        } else {
            // if式はelseが必須
            4
        };
    }
}
```

always_comb ブロック内で変数に代入するとき、if 文の全ての場合で代入する必要があることに注意してください (`v` は常に代入されています)。

リスト 2.31 と同じ意味の文を **switch** 文で書けます (リスト 2.32)。どの条件にも当てはまらな

いときの動作は **default** で指定します。switch は式にできます。switch 式は必ず値を返す必要があり、default が必須です。

▼リスト 2.32: switch 文、switch 式

```
var v : logic<32>;
always_comb {
    switch {
        // WIDTH == 0のとき
        WIDTH == 0: {
            v = 0;
        }
        // WIDTH != 0かつWIDTH == 1のとき
        WIDTH == 1: v = 1; // 要素が1つの文のとき、{}は省略できる
        // WIDTH != 0かつWIDTH != 1のとき
        default:
            // switch式
            v = switch {
                WIDTH == 3: 3, // カンマで区切る
                default : 4, // switch式はdefaultが必須
            };
    }
}
```

リスト 2.31 のように 1 つの要素 (`WIDTH`) の一致のみが条件のとき、同じ意味の文を **case** 文で書けます (リスト 2.33)。式にできたり、式に **default** が必須なのは switch 文と同様です。

▼リスト 2.33: case 文、case 式

```
var v: logic<32>;
always_comb {
    case WIDTH {
        // WIDTH == 0のとき
        0: {
            v = 0;
        }
        // WIDTH != 0かつWIDTH == 1のとき
        1: v = 1; // 要素が1つの文のとき、{}は省略できる
        // WIDTH != 0かつWIDTH != 1のとき
        default:
            // case式
            v = case WIDTH {
                3: 3, // カンマで区切る
                default : 4, // case式はdefaultが必須
            };
    }
}
```

連結、repeat

ビット列や文字列を連結したいとき、`{}` を使用できます (リスト 2.34)。`+` では連結できない (値の足し算になる) ことに注意してください。同じビット列、文字列を繰り返して連結したいときは **repeat** キーワードを使用します (リスト 2.35)。

▼リスト 2.34: 連結

```
{12'h123, 32'habcd0123} // 44'h123_abcde0123になる
{"Hello", " ", "World!"} // "Hello World!"になる
```

▼リスト 2.35: repeat を使って連結を繰り返す

```
// {繰り返したい要素 repeat 繰り返す回数}
{4'b0011 repeat 3, 4'b1111} // 16'b0011_0011_0011_1111になる
{"Happy" repeat 3} // "HappyHappyHappy"になる
```

for

for 文はループを実現するための文です。for 文はリスト 2.36 のように記述できます。例えばループ変数が 0 から 31 になるまで (32 回) 繰り返すなら、範囲に `0..32`、または `0..=31` と記述します。範囲には定数のみ指定できます。

▼リスト 2.36: for 文の記法

```
// for ループ変数名: 型 in 範囲 { 処理 }
for i: u32 in 0..32 { ... }
```

break 文を使うとループから抜け出せます。例えばリスト 2.37 では `x` の値は 256 になります。

▼リスト 2.37: always_comb ブロック内で for 文を記述する例

```
var x: u32;
always_comb {
    x = 0;
    for _: u32 in 0..1024 {
        if x == 256 {
            break;
        }
        x += 1;
    }
}
```

inside、outside

値がある範囲に含まれているかという条件を記述したいとき、**inside** 式を利用できます。

inside 式 {範囲} で、式の結果が範囲内にあるかという条件を記述できます (リスト 2.38)。逆に、範囲外にあるという条件は **outside** 式で記述できます。

▼リスト 2.38: inside、outside

```
inside n {0..10} // nが0以上10未満のとき1
inside n {0..=10} // nが0以上10以下のとき1
inside n {0, 1, 3} // nが0、1、3のいずれかのとき1
inside n {0, 2..10} // nが0、または2以上10未満のとき1

// outsideはinsideの逆
outside n {0..10} // nが0未満、または10より大きいとき1
outside n {0, 1, 3} // nが0、1、3以外の値のとき1
```

function

何度も記述する操作や計算は、関数 (function) を使うことでまとめて記述できます (リスト 2.39)。関数は値を引数で受け取り、return 文で値を返します。値を返さないとき、戻り値の型の指定を省略できます。

引数には向きを指定できます。function の実行を開始するとき、input として指定されている実引数の値が仮引数にコピーされます。function の実行が終了するとき、output として指定されている仮引数の値が実引数の変数にコピーされます。output を使用することで、変数に値を割り当てるることができます。

▼リスト 2.39: 関数

```
// べき乗を返す関数
function get_power(
    a : input u32,
    b : input u32,
) -> u32 {
    return a ** b;
}

val v1 : logic<32>;
val v2 : logic<32>;

always_comb {
    v1 = get_power(2, 10); // v1 = 1024
    v2 = get_power(3, 3); // v2 = 27
}

// a + 1をbに代入する関数
function assign_plus1(
    a : input logic<32>,
    b : output logic<32>,
) { // 戻り値はないので省略
    b = a + 1;
}

val v3 : logic<32>;

always_comb {
    assign_plus1(v1, v3); // v3 = v1 + 1
}
```

2.2.6 interface

モジュールに何個もポートが存在するとき、ポートの接続は非常に手間のかかる作業になります。例えばリスト 2.40 では、向きが対になっているポートが ModuleA と ModuleB に定義されており、これを一つ一つ接続しています。

▼リスト 2.40: モジュールのポートの相互接続

```
module ModuleA (
    req_a: output logic,
    req_b: output logic,
    req_c: output logic,
) {}
module ModuleB (
    resp_a: input logic,
    resp_b: input logic,
    resp_c: input logic,
) {}
module Top{
    var a: logic;
    var b: logic;
    var c: logic;
    inst ma : ModuleA (
        req_a:a,
        req_b:b,
        req_c:c,
    );
    inst mb : ModuleB (
        resp_a:a,
        resp_b:b,
        resp_c:c,
    );
}
```

モジュール間のポートの接続を簡単に行うために、インターフェース (**interface**) という機能が用意されています。リスト 2.40 の ModuleA と ModuleB を相互接続するようなインターフェースは次のように定義できます（リスト 2.41）。

▼リスト 2.41: インターフェースの定義

```
// interface インターフェース名 { }
interface iff_ab {
    var a : logic;
    var b : logic;
    var c : logic;

    modport req {
        a: input,
        b: input,
        c: input,
    }
    modport resp {
        a: output,
        b: output,
        c: output,
    }
}
```

iff_ab インターフェースを利用すると、リスト 2.40 を簡潔に記述できます（リスト 2.42）。

▼リスト 2.42: インターフェースによる接続

```
module ModuleA (
    req : modport iff_ab::req,
){}

module ModuleB (
    resp : modport iff_ab::resp,
){}

module Top{
    // インターフェースのインスタンス化
    inst iab : iff_ab;
    inst ma : ModuleA (req: iab);
    inst mb : ModuleB (resp: iab);
}
```

インターフェースはポートの宣言と接続を抽象化します。インターフェース内に変数を定義すると、**modport** 文によってポートと向きを宣言できます。モジュールでのポートの宣言は、**ポート名 : modport インターフェース名::modport 名** と記述できます。modport で宣言されたポートにインターフェースのインスタンスを渡すことにより、ポートの接続を一気に行えます。モジュールと同じように、インターフェースにはパラメータを宣言できます（リスト 2.43）。

▼リスト 2.43: パラメータ付きのインターフェース

```
// interface インターフェース名 #( パラメータの定義 ) { }
interface iff_params #(
    param PARAM_A : u32 = 100,
    param PARAM_B : u64 = 200,
){ }
```

インターフェース内には関数の定義や `always_comb` ブロック、`always_ff` ブロックなどの文を記述できます。

2.2.7 package

複数のモジュールやインターフェースにまたがって使用したいパラメータや型、関数はパッケージ（package）に定義できます（リスト 2.44）。

▼リスト 2.44: パッケージの定義

```
package PackageA {
    const WIDTH : u32 = 1234;
    type foo = logic<WIDTH>;
    function bar () -> u32 {
        return 1234;
    }
}
```

パッケージに定義した要素には、**パッケージ名::要素名** でアクセスできます（リスト 2.45）。

▼リスト 2.45: パッケージの要素にアクセスする

```
module ModuleA {
    const W : u32 = PackageA::WIDTH;
    var value1 : PackageA::foo;
    let value2 : u32 = PackageA::bar();
}
```

import 文を使用すると、要素へのアクセス時にパッケージ名の指定を省略できます（リスト 2.46）。

▼リスト 2.46: パッケージを import する

```
import PackageA::WIDTH; // 特定の要素をimportする
import PackageA::*; // 全ての要素をimportする
```

2.2.8 ジェネリクス

関数やモジュール、インターフェース、パッケージ、構造体はジェネリクス (generics) によってパラメータ化できます。

例えば、要素に任意の型 T や W ビットのデータを持つ構造体は、次のようにジェネリックパラメータ (generic parameter) を使うことで定義できます（リスト 2.47）。ジェネリックパラメータに渡される値は、ジェネリクスの定義位置からアクセスできる定数である必要があります。

▼リスト 2.47: パラメータ化された構造体

```
module ModuleA {
    // ::<>でジェネリックパラメータを定義する
    // constで数値を受け取る
    struct StructA::<W: const> {
        A: logic<W>;
    }

    // 複数のジェネリックパラメータを定義できる
    // typeで型を受け取る
    // デフォルト値を設定できる
    struct StructB::<W: const, T: type, D:const = 100> {
        A: logic<W>;
        B: T;
        C: logic<D>;
    }

    // ::<>でジェネリックパラメータを指定する
    type A = StructA::<16>;
    type B = StructB::<17, A>;
    type C = StructB::<18, B, 19>;
}
```

2.2.9 その他の機能、文

initial、final

initial ブロックの中の文はシミュレーションの開始時に実行されます。**final** ブロックの中の文はシミュレーションの終了時に実行されます(リスト 2.48)。

▼リスト 2.48: initial、final ブロック

```
module ModuleA {
    initial {
        // シミュレーション開始時に実行される
    }
    final {
        // シミュレーション終了時に実行される
    }
}
```

SystemVerilog との連携

SystemVerilog のモジュールやパッケージ、インターフェースを利用できます。SystemVerilog のリソースにアクセスするには `$sv::` を使用します(リスト 2.49)。

▼リスト 2.49: SystemVerilog の要素を利用する

```
module ModuleA {
    // SystemVerilogでsvpackageとして
    // 定義されているパッケージを利用する
    let x = $sv::svpackage::X;
    let y = $sv::svpackage::Y;

    var s: logic;
    var c: logic;

    // SystemVerilogでHalfAdderとして
    // 定義されているモジュールをインスタンス化する
    inst ha : $sv::HalfAdder(
        x, y, s, c
    );

    // SystemVerilogでsvinterfaceとして
    // 定義されているインターフェースをインスタンス化する
    inst c: $sv::svinterface;
}
```

SystemVerilog のソースコードを直接埋め込み、展開できます(リスト 2.50)。

▼リスト 2.50: SystemVerilog 記述を埋め込む

```
// SystemVerilog記述を直接埋め込む
embed (inline) sv{{{
    module ModuleA(
        output logic a
    );
}}}
```

```

        assign a = 0;
      endmodule
    }

// SystemVerilogのソースファイルを展開する
// パスは相対パス
include.inline, "filename.sv");

```

システム関数、システムタスク

SystemVerilog に標準で用意されている関数 (システム関数、システムタスク) を利用できます。システム関数 (system function) とシステムタスク (system task) の名前は \$ から始まります。本書で利用するシステム関数とシステムタスクを表 2.6 に例挙します。

▼表 2.6: 本書で使用するシステム関数、システムタスク

関数名	機能	戻り値
\$clog2	値の log2 の ceil を求める	数値
\$size	配列のサイズを求める	数値
\$bits	値の幅を求める	数値
\$signed	値を符号付きとして扱う	符号付きの値
\$readmemh	レジスタにファイルのデータを代入する	なし
\$display	文字列を出力する	なし
\$error	エラー出力する	なし
\$finish	シミュレーションを終了する	なし

それぞれの使用例は次の通りです (リスト 2.51)。システム関数やシステムタスクを利用するとときは、通常の関数呼び出しのように使用します。

▼リスト 2.51: システム関数、システムタスクの使用例

```

const w1 : u32 = $clog2(32); // 5
const w2 : u32 = $clog2(35); // 6

var array : logic<4,8>;
const s1 : u32 = $size(array); // 4
const s2 : u32 = $bits(array); // 32

var uvalue : u32;
let svalue : i32 = $signed(uvalue) + 1;

initial {
    $readmemh("file.hex", array);
    $display("Hello World!");
    $error("Error!");
    $finish();
}

```

アトリビュート

アトリビュートを使うと、宣言に注釈をつけられます。例えばリスト 2.52 は、リスト 2.53 にトランスパイルされます。

▼ リスト 2.52: アトリビュートを使った Veril コード

```
#[sv("keep=\"true\"")]
var aaa : logic;

#[ifdef(IS_DEBUG)]
var bbb : logic;

#[ifndef(TEST)]
var ccc : logic;
```

▼ リスト 2.53: 同じ意味の SystemVerilog コード

```
(* keep="true" *)
logic aaa;

`ifdef IS_DEBUG
logic bbb;
`endif

`ifndef TEST
logic ccc;
`endif
```

`#[sv()]` は、宣言に SystemVerilog の属性を付けられます。属性は使用するときに説明します。
`#[ifdef(マクロ名)]` をつけられた宣言は、マクロが存在するときにのみ定義されるようになります。
`#[ifndef(マクロ名)]` はその逆で、マクロが存在しないときにのみ定義されるようになります。

アトリビュートはポートやパラメータ、ブロック、モジュール、インターフェース、パッケージなど、どの宣言にも付けることができます。

標準ライブラリ

Veril には、よく使うモジュールなどが標準ライブラリとして準備されています。標準ライブラリは <https://std.veril-lang.org/> で確認できます。

本書では標準ライブラリを使用していないため、説明は割愛します。

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 の CPU は基本整数命令セットを必ず実装して、他の命令や機能は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、まずは最小限の命令を実行できる CPU を実装しましょう。

3.1 CPU は何をやっているのか？

CPU を実装するには何が必要でしょうか？まずは CPU とはどのような動作をするものなのを考えます。**プログラム内蔵方式** (stored-program computer) と呼ばれるコンピュータの CPU は、次の手順でプログラムを実行します。

1. メモリ (memory, 記憶装置) 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」とは一体何を指しているのでしょうか？普通のプログラムが書くのは 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 が読み込んで実行できる形式のプログラムです。これはよく**機械語** (machine code) と呼ばれ、0 と 1 で表される 2 進数のビット列^{*1}で記述されています。

メモリから機械語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。

^{*1} その昔、Setun という 3 進数のコンピュータが存在したらしく、機械語は 3 進数のトリット (trit) で構成されていたようです

まず、機械語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, address)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 は機械語を実行しますが、一気にすべての機械語を読み込んだり実行するわけではなく、機械語の最小単位である**命令**(instruction)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呼びます。

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呼びます。

命令をデコードすると、いよいよ計算やメモリの読み書きを行います。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、次のいずれかで指定されます。

- レジスタ(=CPU 内に存在する計算データ用のレジスタ列)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ちら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。

フォンノイマン型アーキテクチャ(von Neumann architecture)と呼ばれるコンピュータの構成方式では、メモリのデータの読み書きを、機械語が格納されているメモリと同じメモリに対して行います。

計算やメモリの読み書きが終わると、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、メモリから値を読み込む命令なら読み込まれた値を格納します。

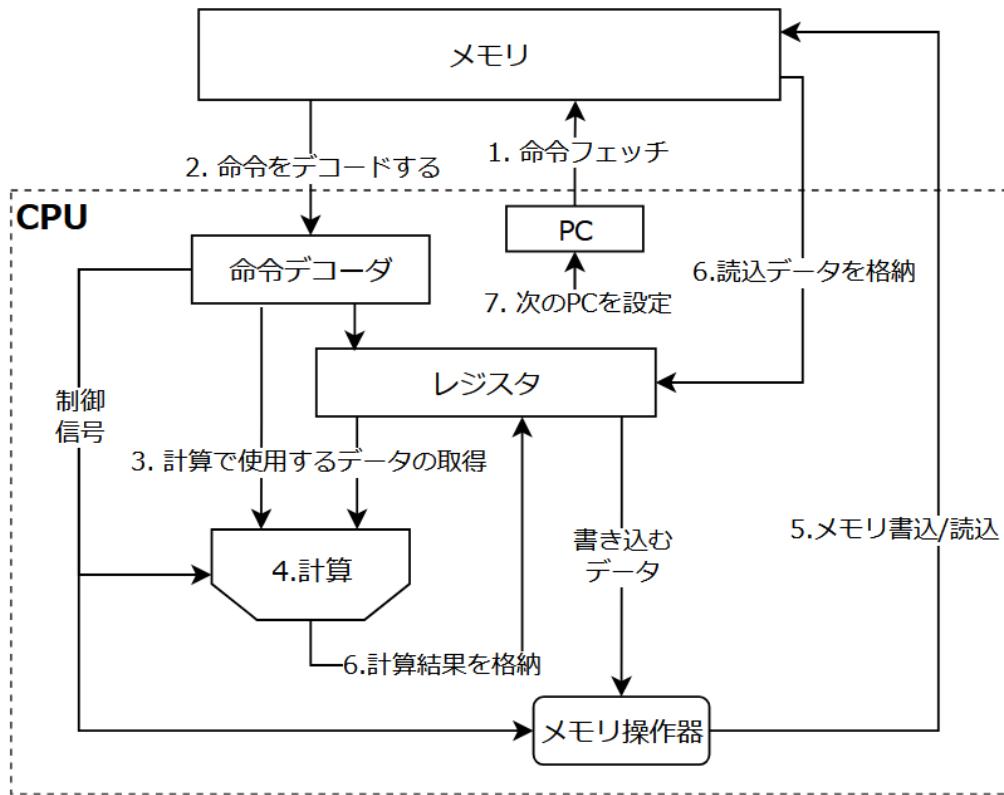
これで命令の実行は終わりですが、CPU は次の命令を実行する必要があります。今現在実行している命令のアドレスを格納しているレジスタのことを**プログラムカウンタ**(program counter, PC)と呼びます。CPU は PC の値をメモリに渡すことで命令をフェッチしています。

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、まず、分岐の成否を判定します。分岐が成立する場合は PC の値を分岐先のアドレスに設定します。分岐が成立しない場合は通常の命令と同じです。

ここまで話をまとめると、CPU の動作は次のようになります(図 3.1)。

1. PC に格納されたアドレスにある命令をフェッチする
2. 命令を取得したらデコードする
3. 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)
4. 計算する命令の場合、計算を行う
5. メモリにアクセスする命令の場合、メモリ操作を行う
6. 計算やメモリアクセスの結果をレジスタに格納する
7. PC の値を次に実行する命令のアドレスに設定する

CPU が一体どんなものなのかが分かりましたか? 実装を始めましょう。



▲図 3.1: CPU の動作

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 (リスト 3.1)。プロジェクトは 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、プロジェクト名のディレクトリと、その中に `Veryl.toml` が作成されます。`Veryl.toml` を次のように変更してください (リスト 3.2)。

▼リスト 3.2: `Veryl.toml`

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

```
[build]
sourcemap_target = {type ="none"}
```

Veryl のソースファイルを格納するために、プロジェクトのディレクトリ内に src ディレクトリを作成してください (リスト 3.3)。

▼リスト 3.3: src ディレクトリを作成する

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

3.3 定数の定義

いよいよコードを記述します。まず、CPU 内で何度も使用する定数や型を書いておくための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 3.4)。

▼リスト 3.4: 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、この名前を使用しています。

eei パッケージには、次の定数を定義します。

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、type 文によって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の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。そのため、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 (ILEN) です。また、メモリからの読み込み命令、書き込み命令の最大の幅も 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を定義する

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表 3.1 のようなポートを定義する必要があります。これを一つ一つ定義して接続するのは面倒なため、interface を定義します。

▼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
valid	logic	input	メモリアクセスを要求しているかどうか
ready	logic	output	メモリアクセス要求を受容するかどうか
addr	logic<ADDR_WIDTH>	input	アクセス先のアドレス
wen	logic	input	書き込みかどうか (1 なら書き込み)
wdata	logic<DATA_WIDTH>	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	logic<DATA_WIDTH>	output	受容した読み込み命令の結果

`src/membus_if.veryl` を作成し、次のように記述します (リスト 3.5)。

▼リスト 3.5: インターフェースの定義 (membus_if.veryl)

```
interface membus_if::<DATA_WIDTH: const, ADDR_WIDTH: const> {
    var valid : logic          ;
    var ready : logic          ;
```

```

var addr  : logic<ADDR_WIDTH>;
var wen   : logic           ;
var wdata : logic<DATA_WIDTH>;
var rvalid: logic           ;
var rdata : logic<DATA_WIDTH>;

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,
}
}

```

`membus_if` はジェネリックインターフェースです。ジェネリックパラメータとして、`ADDR_WIDTH` と `DATA_WIDTH` が定義されています。`ADDR_WIDTH` はアドレスの幅、`DATA_WIDTH` は1つのデータの幅です。

interface を利用することで変数の定義が不要になり、ポートの相互接続を簡潔にできます。

3.4.2 メモリモジュールを実装する

メモリを作る準備が整いました。`src/memory.veryl` を作成し、次のように記述します（リスト3.6）。

▼リスト 3.6: メモリモジュールの定義 (memory.veryl)

```

module memory:<DATA_WIDTH: const, ADDR_WIDTH: const> #(
    param FILEPATH_IS_ENV: logic  = 0 , // FILEPATHが環境変数名かどうか
    param FILEPATH        : string = "", // メモリの初期化用ファイルのパス、または環境変数名
) (
    clk    : input  clock           ,
    rst    : input  reset           ,
    membis: modport membis_if:<DATA_WIDTH, ADDR_WIDTH>::slave,
) {
    type DataType = logic<DATA_WIDTH>;
    var mem: DataType [2 ** ADDR_WIDTH];
}

```

```

initial {
    // memを初期化する
    if FILEPATH != "" {
        if FILEPATH_IS_ENV {
            $readmemh(util::get_env(FILEPATH), mem);
        } else {
            $readmemh(FILEPATH, mem);
        }
    }
}

always_comb {
    membus.ready = 1;
}

always_ff {
    if_reset {
        membus.rvalid = 0;
        membus.rdata  = 0;
    } else {
        membus.rvalid = membus.valid;
        membus.rdata  = mem[membus.addr[ADDR_WIDTH - 1:0]];
        if membus.valid && membus.wen {
            mem[membus.addr[ADDR_WIDTH - 1:0]] = membus.wdata;
        }
    }
}
}

```

memory モジュールはジェネリックモジュールです。次のジェネリックパラメータを定義しています。

DATA_WIDTH

メモリのデータの単位の幅を指定するためのパラメータです。

この単位ビットでデータを読み書きします。

ADDR_WIDTH

データのアドレスの幅(メモリの容量)を指定するためのパラメータです。

メモリの容量は `DATA_WIDTH * (2 ** ADDR_WIDTH)` ビットになります。

ポートには、クロック信号とリセット信号と `membus_if` インターフェースを定義しています。
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` は対象アドレスのデータになります。

書き込み

書き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。

3.4.3 メモリの初期化、環境変数の読み込み

memory モジュールのパラメータには、`FILEPATH_IS_ENV` と `FILEPATH` を定義しています。memory モジュールをインスタンス化するとき、`FILEPATH` には、メモリの初期値が格納されたファイルのパスか、ファイルパスが格納されている環境変数名を指定します。初期化は `$readmemh` システムタスクで行います。

`FILEPATH_IS_ENV` が 1 のとき、環境変数の値を取得して、初期化用のファイルのパスとして利用します。環境変数は util パッケージの `get_env` 関数で取得します。

util パッケージと `get_env` 関数を作成します。`src/util.veryl` を作成し、次のように記述します（リスト 3.7）。

▼ リスト 3.7: util.veryl

```
embed (inline) sv{{{
    package svutil;
        import "DPI-C" context function string get_env_value(input string key);
        function string get_env(input string name);
            return get_env_value(name);
        endfunction
    endpackage
}}}

package util {
    function get_env (
        name: input string,
    ) -> string {
        return $sv::svutil::get_env(name);
    }
}
```

util パッケージの `get_env` 関数は、コード中に埋め込まれた SystemVerilog の `svutil` パッケージの `get_env` 関数の結果を返しています。`svutil` パッケージの `get_env` 関数は、C(C++) で定義されている `get_env_value` 関数の結果を返しています。`get_env_value` 関数は後で定義します。

3.5 最上位モジュールの作成

次に、最上位のモジュール（Top Module）を作成して、memory モジュールをインスタンス化します。

最上位のモジュールとは、設計の階層の最上位に位置するモジュールのことです。論理設計では、最上位モジュールの中に、あらゆるモジュールやレジスタなどをインスタンス化します。

memory モジュールはジェネリックモジュールであるため、1つのデータのビット幅とメモリのサイズを指定する必要があります。これらを示す定数を eei パッケージに定義します（リスト 3.8）。メモリのアドレス幅（サイズ）には、適当に 16 を設定しています。これによりメモリ容量は $32\text{ ビット} * (2^{16}) = 256\text{KiB}$ になります。

▼リスト 3.8: メモリのデータ幅とアドレスの幅の定数を定義する (eei.veryl)

```
// メモリのデータ幅
const MEM_DATA_WIDTH: u32 = 32;
// メモリのアドレス幅
const MEM_ADDR_WIDTH: u32 = 16;
```

それでは、最上位のモジュールを作成します。 `src/top.veryl` を作成し、次のように記述します（リスト 3.9）。

▼リスト 3.9: 最上位モジュールの定義 (top.veryl)

```
import eei::*;

module top (
    clk: input clock,
    rst: input reset,
) {
    inst membus: membus_if::<MEM_DATA_WIDTH, MEM_ADDR_WIDTH>;

    inst mem: memory::<MEM_DATA_WIDTH, MEM_ADDR_WIDTH> #(
        FILEPATH_IS_ENV: 1
        FILEPATH      : "MEMORY_FILE_PATH",
    ) (
        clk      ,
        rst      ,
        membus   ,
    );
}
```

top モジュールでは、先ほど作成した memory モジュールと、membus_if インターフェースをインスタンス化しています。

memory モジュールと membus インターフェースのジェネリックパラメータには、`DATA_WIDTH` に `MEM_DATA_WIDTH`、`ADDR_WIDTH` に `MEM_ADDR_WIDTH` を指定しています。メモリの初期化は、環境変数 `MEMORY_FILE_PATH` で行うようにパラメータで指定しています。

3.6 命令フェッチ

メモリを作成したので、命令フェッチ処理を作れるようになりました。

いよいよ、CPU のメインの部分を作成します。

3.6.1 命令フェッチを実装する

`src/core.veryl` を作成し、次のように記述します（リスト 3.10）。

▼リスト 3.10: core.veryl

```
import eei::*;

module core (
    clk    : input    clock           ,
    rst    : input    reset           ,
    membus: modport membus_if:<ILEN, XLEN>::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 && membus.valid;
                    if membus.ready && membus.valid {
                        if_pc          = if_pc_next;
                        if_pc_requested = if_pc;
                    }
                }
            } else {
                if membus.ready && membus.valid {
                    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);
        }
    }
}

```

core モジュールは、クロック信号とリセット信号、membus_if インターフェースをポートに持っています。membus_if インターフェースのジェネリックパラメータには、データ単位として `ILEN` (1 つの命令のビット幅)、アドレスの幅として `XLEN` を指定しています。

`if_pc` レジスタは PC(プログラムカウンタ) です。ここで `if_` という接頭辞は instruction fetch(命令フェッチ) の略です。`if_is_requested` は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。どのレジスタも `0` で初期化しています。

`always_comb` ブロックでは、アドレス `if_pc` にあるデータを常にメモリに要求しています。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` にしています。

上から 1 つめの `always_ff` ブロックでは、フェッチ中かどうかとメモリが ready(要求を受け入れる) 状態かどうかによって、`if_pc` と `if_is_requested`、`if_pc_requested` の値を変更しています。

メモリにデータを要求するとき、`if_pc` を次の命令のアドレス (4 を足したアドレス) に変更して、`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、命令フェッチが完了し、データが `membus.rdata` に供給されています。メモリが ready 状態なら、すぐに次の命令フェッチを開始します。この状態遷移を繰り返すことによって、アドレス `0 → 4 → 8 → c → 10 ...` の命令を次々にフェッチします。

上から 2 つめの `always_ff` ブロックは、デバッグ用の表示を行うコードです。命令フェッチが完了したとき、その結果を `$display` システムタスクによって出力します。

3.6.2 memory モジュールと core モジュールを接続する

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 インターフェースでメモリと接続します。

core モジュールが指定するアドレスは 1 バイト単位のアドレスです。それに対して、memory モジュールは 32 ビット (=4 バイト) 単位でデータを整列しているため、データは 4 バイト単位のアドレスで指定する必要があります。

まず、1 バイト単位のアドレスを、4 バイト単位のアドレスに変換する関数を作成します (リスト 3.11)。これは、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実現できます。

▼リスト 3.11: アドレスを変換する関数を作成する (top.veryl)

```

// アドレスをメモリのデータ単位でのアドレスに変換する
function addr_to_memaddr (
    addr: input logic<XLEN>           ,
) -> logic<MEM_ADDR_WIDTH> {

```

```
    return addr[$clog2(MEM_DATA_WIDTH / 8)+:MEM_ADDR_WIDTH];
}
```

addr_to_memaddr 関数は、`MEM_DATA_WIDTH` (=32) をバイトに変換した値 (=4) の `log2` をとった値 (=2) を使って、`addr[17:2]` を切り取っています。範囲の選択には `+:` を利用しています。

次に、core モジュール用の membus_if インターフェースを作成します（リスト 3.12）。ジェネリックパラメータには、core モジュールのインターフェースのジェネリックパラメータと同じく、`ILEN` と `XLEN` を割り当てます。

▼ リスト 3.12: core モジュール用の membus_if インターフェースをインスタンス化する (top.veryl)

```
inst membus      : membus_if::<MEM_DATA_WIDTH, MEM_ADDR_WIDTH>;
inst membus_core: membus_if::<ILEN, XLEN>;
```

`membus` と `membus_core` を接続します。アドレスには `addr_to_memaddr` 関数で変換した値を割り当てます（リスト 3.13）。

▼ リスト 3.13: membus と membus_core を接続する (top.veryl)

```
always_comb {
    membus.valid      = membus_core.valid;
    membus_core.ready = membus.ready;
    // アドレスをデータ幅単位のアドレスに変換する
    membus.addr       = addr_to_memaddr(membus_core.addr);
    membus.wen        = 0; // 命令フェッチは常に読み込み
    membus.wdata       = 'x;
    membus_core.rvalid = membus.rvalid;
    membus_core.rdata  = membus.rdata;
}
```

最後に core モジュールをインスタンス化します（リスト 3.14）。メモリと CPU が接続されました。

▼ リスト 3.14: core モジュールをインスタンス化する (top.veryl)

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

3.6.3 命令フェッチをテストする

ここまでコードが正しく動くかを検証します。

Veryl で記述されたコードは `veryl build` コマンドで SystemVerilog のコードに変換できます。変換されたソースコード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、Veryl のプロジェクトをビルドします（リスト 3.15）。

▼リスト 3.15: Veryl のプロジェクトのビルド

```
$ 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 ファイルと同名の `sv` ファイルと `core.f` ファイルが生成されます。拡張子が `sv` のファイルは SystemVerilog のファイルで、`core.f` には生成された SystemVerilog のファイルのリストが記載されています。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 のコードを 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、次のような C++ プログラムを書きます*2。

`src/tb_verilator.cpp` を作成し、次のように記述します（リスト 3.16）。

▼リスト 3.16: tb_verilator.cpp

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

namespace fs = std::filesystem;

extern "C" const char* get_env_value(const char* key) {
    const char* value = getenv(key);
    if (value == nullptr)
        return "";
    return value;
}

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;
    }
}
```

*2 Verilog のソースファイルだけでビルドすることもできます

```

}

// 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;
    }
}

// 環境変数でメモリの初期化用ファイルを指定する
const char* original_env = getenv("MEMORY_FILE_PATH");
setenv("MEMORY_FILE_PATH", memory_file_path.c_str(), 1);

// top
Vcore_top *dut = new Vcore_top();

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

// 環境変数を元に戻す
if (original_env != nullptr){
    setenv("MEMORY_FILE_PATH", original_env, 1);
}

// loop
dut->rst = 1;
for (long long i=0; !Verilated::gotFinish() && (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、そのクロック信号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

実行時に環境変数 MEMORY_FILE_PATH として渡されます。

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、top モジュールのクロック信号を更新して 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信号によりリセットする必要があります。そのため、top モジュールの rst を 1 にしてから 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 3.17）。

▼ リスト 3.17: 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 ディレクトリに指定しています。

リスト 3.17 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 3.18）。

▼ リスト 3.18: sample.hex

```
01234567
89abcdef
deadbeef
cafebabe
```

←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。シミュレータを実行すると、memory モジュールは `$readmemh` システムタスクで `sample.hex` を読み込みます。それにより、メモリは次のように初期化されます (表 3.2)。

▼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
0x000000000	01234567
0x000000004	89abcdef
0x000000008	deadbeef
0x00000000c	cafebebe
0x000000010～	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 3.19)。

▼リスト 3.19: 命令フェッチの動作チェック

```
$ obj_dir/sim src/sample.hex 5
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を確認できます。

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非常に面倒です。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 3.20)。

▼リスト 3.20: Makefile

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

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

build:
    veryl fmt
    veryl build
```

```

clean:
    veryl clean
    rm -rf $(OBJ_DIR)

sim:
    verilator --cc $(VERILATOR_FLAGS) -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)

.PHONY: build clean sim

```

これ以降、次のように Verilator のソースコードのビルド、シミュレータのビルド、成果物の削除ができるようになります（リスト 3.21）。

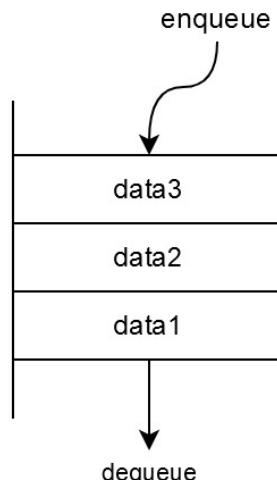
▼ リスト 3.21: Makefile によって追加されたコマンド

```

$ make build ← Verilator のソースコードのビルド
$ 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

```

3.6.4 フェッチした命令を FIFO に格納する



▲ 図 3.2: FIFO

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りません。

複数クロックかかる命令に対応するために、命令の処理が終わってから次の命令をフェッチするように変更する場合、命令の実行の流れは次のようになります。

1. 命令の処理が終わる
2. 次の命令のフェッチ要求をメモリに送る
3. 命令がフェッチされ、命令の処理を開始する

このとき、命令の処理が終わってから次の命令をフェッチするため、次々にフェッチするよりも多くのクロック数が必要です。これは CPU の性能を露骨に悪化させるので許容できません。

FIFO の作成

そこで、**FIFO**(First In First Out, ファイフオ) を作成して、フェッチした命令を格納します。 FIFO とは、先に入れたデータが先に出されるデータ構造のことです(図 3.2)。命令をフェッチしたら FIFO に格納 (enqueue) し、命令を処理するときに FIFO から取り出し (dequeue) ます。

Veryl の標準ライブラリ^{*3}には FIFO が用意されていますが、FIFO は簡単なデータ構造なので自分で作ってみましょう。`src/fifo.veryl` を作成し、次のように記述します(リスト 3.22)。

▼リスト 3.22: FIFO モジュールの実装 (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 as Ptr;
    let tail_plus2: Ptr = tail + 2 as Ptr;
    always_comb {
        rvalid = head != tail;
        rdata  = mem[head];
    }
    assign wready = if WIDTH == 1 {
        head == tail || rready
    } else {
        tail_plus1 != head
    };
}
```

^{*3} <https://std.veryl-lang.org/>

```

// 2つ以上空きがあるかどうか
let wready_two: logic = wready && tail_plus2 != head;

always_ff {
    if_reset {
        head = 0;
        tail = 0;
    } else {
        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 $2^{** \text{WIDTH}} - 1$ 個格納できる 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せます。

データを取り出す

`rvalid` が 1 のとき、データを取り出せます。データを取り出せるとき、`rdata` にデータが供給されています。`rready` を 1 にすることで、FIFO にデータを取り出したことを通知できます。

データの格納状況は、`head` レジスタと `tail` レジスタで管理します。データを追加するとき、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。

データを追加できる状況とは、`tail` に 1 を足しても `head` を超えないとき、つまり、`tail` が指す場所が一周しまわないときです。この制限から、FIFO には最大でも $2^{** \text{WIDTH}} - 1$ 個しかデータを格納できません。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。`WIDTH` が 1 のときは特別で、既にデータが 1 つ入っていても、`rready` が 1 のときはデータを追加できるようにしています。

命令フェッチ処理の変更

fifo モジュールを使って、命令フェッチ処理を変更します。

まず、FIFO に格納する型を定義します（リスト 3.23）。`if_fifo_type` には、命令のアドレス（`addr`）と命令のビット列（`bits`）を格納するためのフィールドを含めます。

▼リスト 3.23: FIFO で格納する型を定義する (core.veryl)

```
// ifのFIFOのデータ型
struct if_fifo_type {
    Addr: Addr,
    Inst: Inst,
}
```

次に、FIFO と接続するための変数を定義します（リスト 3.24）。

▼リスト 3.24: FIFO と接続するための変数を定義する (core.veryl)

```
// 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;
```

FIFO モジュールをインスタンス化します（リスト 3.25）。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できるようにします。WIDTH パラメータには `3` を指定することで、サイズを $2^{** 3 - 1} = 7$ にしています。このサイズは適当です。

▼リスト 3.25: FIFO をインスタンス化する (core.veryl)

```
// 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 ,
);
```

fifo モジュールをインスタンス化したので、メモリへデータを要求する処理を変更します（リスト 3.26）。

▼リスト 3.26: フェッチ処理の変更 (core.veryl)

```
// 命令フェッチ処理
always_comb {
    // FIFOに2個以上空きがあるとき、命令をフェッチする
    membus.valid = if_fifo.wready_two;
    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;
}

```

リスト 3.26 では、メモリに命令フェッチを要求する条件を FIFO に 2つ以上空きがあるという条件に変更しています^{*4}。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、FIFO から常にデータを取り出すようにしています。

命令をフェッチできたら FIFO に格納する処理を always_ff ブロックの中に追加します（リスト 3.27）。

▼ リスト 3.27: FIFO へのデータの格納 (core.veryl)

```

// 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;
    }
}

```

`if_fifo_wvalid` と `if_fifo_wdata` を `0` に初期化します（リスト 3.28）。

▼ リスト 3.28: 変数の初期化 (core.veryl)

```

if_reset {
    if_pc          = 0;
    if_is_requested = 0;
    if_pc_requested = 0;
    if_fifo_wvalid = 0;
    if_fifo_wdata  = 0;
} else {

```

命令をフェッチできたとき、`if_fifo_wvalid` の値を `1` にして、`if_fifo_wdata` 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。

それ以外のとき、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 2つ以上空きがあるときに行うため、まだ追加されていないデータが `if_fifo_wdata` 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。

^{*4} 1つ空きがあるという条件だとあふれてしまいます。FIFO が容量いっぱいのときにどうなるか確認してください

FIFO のテスト

FIFO をテストする前に、命令のデバッグ表示を行うコードを変更します (リスト 3.29)。

▼ リスト 3.29: 命令のデバッグ表示を変更する (core.veryl)

```
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);
    }
}
```

シミュレータを実行します (リスト 3.30)。命令がフェッチされて表示されるまでに、FIFO に格納してから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 3.30: FIFO をテストする

```
$ make build
$ make 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 (decode) と呼びます。

RISC-V の命令のビット列には次のような要素が含まれています。

オペコード (opcode)

5 ビットの値です。命令を区別するために使用されます。

funct3、funct7

funct3 は 3 ビット、funct7 は 7 ビットの値です。命令を区別するために使用されます。

即値 (Immediate, imm)

命令のビット列の中に直接含まれる数値です。

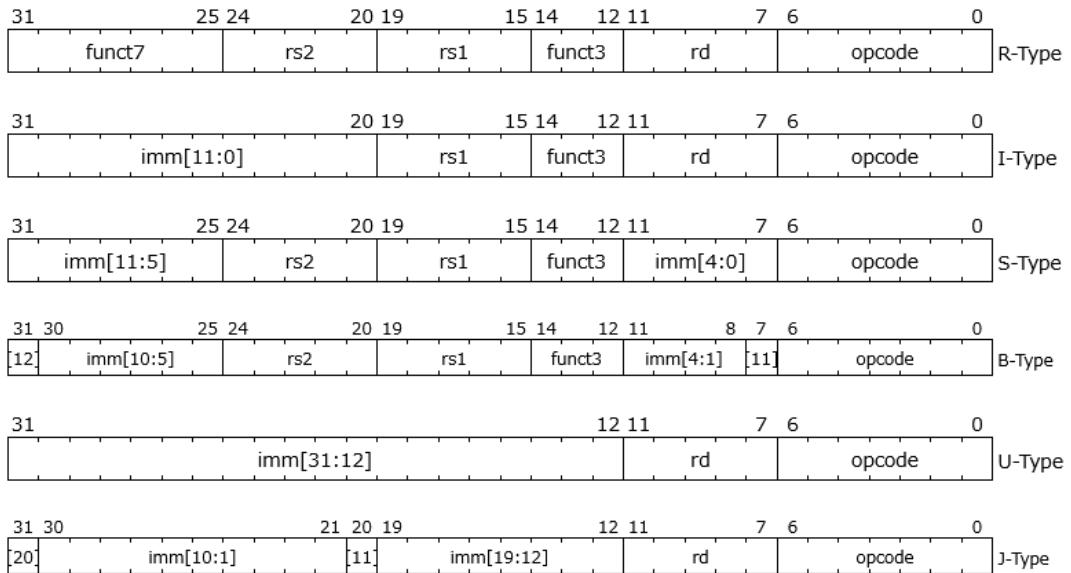
ソースレジスタ (Source Register) の番号

計算やメモリアクセスに使う値が格納されているレジスタの番号です。レジスタは 32 個あるため 5 ビットの値になっています。

デスティネーションレジスタ (Destination Register) の番号

命令の結果を格納するためのレジスタの番号です。ソースレジスタと同様に 5 ビットの値になっています。

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 3.3)。



▲図 3.3: RISC-V の命令形式 [5]

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(足し算)、SUB(引き算) 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(即値を使った足し算)、ANDI(即値を使った AND 演算) 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、即値と rs1 を足し合わせたメモリのアドレスに、rs2 を書き込みます。例えば SW 命令 (メモリに 32 ビット書き込む命令) 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、PC に即値を足したアドレスにジャンプします。

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 に即値を足したアドレスにジャンプします。

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 デコード用の定数と型を定義する

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 3.31)。

▼ リスト 3.31: 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
        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ひとまとめにした構造体です。`itype` には命令の形式、`funct3` と `funct7` にはそれぞれ命令の `funct3` と `funct7` フィールドを格納します。これ以外の構造体のフィールドは、使用するときに説明します。

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 3.32）。

▼ リスト 3.32: `opcode` の定数を定義する (`eei.veryl`)

```
// opcode
const OP_LUI    : logic<7> = 7'b0110111;
const OP_AUIPC : logic<7> = 7'b0010111;
const OP_OP     : logic<7> = 7'b0110011;
const OP_OP_IMM: logic<7> = 7'b0010011;
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;
```

これらの値とそれぞれの命令の対応は、仕様書 [6] を確認してください。

3.7.2 制御フラグと即値を生成する

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 3.33）。

▼ リスト 3.33: `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_i: UIntX = {bits[31] repeat XLEN - $bits(imm_i_g), imm_i_g};
```

```

let imm_s: UIntX = {bits[31] repeat XLEN - $bits(imm_s_g), imm_s_g};
let imm_b: UIntX = {bits[31] repeat XLEN - $bits(imm_b_g) - 1, imm_b_g, 1'b0};
let imm_u: UIntX = {bits[31] repeat XLEN - $bits(imm_u_g) - 12, imm_u_g, 12'b0};
let imm_j: UIntX = {bits[31] repeat XLEN - $bits(imm_j_g) - 1, imm_j_g, 1'b0};

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: imm_i,
        OP_OP_IMM : imm_i,
        OP_BRANCH : imm_b,
        OP_STORE : imm_s,
        default : 'x,
    };
    ctrl = {case op {
        OP_LUI : {InstType::U, T, T, F, F, F, F},
        OP_AUIPC : {InstType::U, T, F, F, F, F, F},
        OP_JAL : {InstType::J, T, F, F, T, F, F},
        OP_JALR : {InstType::I, T, F, F, T, F, F},
        OP_BRANCH: {InstType::B, F, F, F, F, F, F},
        OP_LOAD : {InstType::I, T, F, F, F, F, T},
        OP_STORE : {InstType::S, F, F, F, F, F, F},
        OP_OP : {InstType::R, T, F, T, F, F, F},
        OP_OP_IMM: {InstType::I, T, F, T, F, F, F},
        default : {InstType::X, 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 形式の命令を考えます。まず、命令のビット列から即値部分を取り出して変数 `imm_b_g` 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。

`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供給しています。

制御フラグの生成

opcode が OP-IMM な命令、例えば ADDI 命令を考えます。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 (OP-IMM) のとき、次のように制御信号 `ctrl` を設定します。1 ビットの `1'b0` と `1'b1` を入力する手間を省くために、`F` と `T` という定数を用意していることに注意してください。

- 命令形式 `itype` を `InstType::I` に設定します
- 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(計算を実行する部品) を利用するため、`is_aluop` を `1` に設定します
- `funct3`、`funct7` に命令中のビットをそのまま設定します
- それ以外のフィールドは `0` に設定します

3.7.3 デコーダをインスタンス化する

`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 3.34)。

▼ リスト 3.34: `inst_decoder` モジュールのインスタンス化 (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。

デバッグ用の `always_ff` ブロックに、デコードした結果をデバッグ表示するコードを記述します (リスト 3.35)。

▼ リスト 3.35: デコード結果のデバッグ表示 (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);
    }
}
```

`src/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 3.36)。

▼リスト 3.36: デコーダをテストする

```
$ make build
$ make 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     : fffffdb5ea
0000000c : cafebebe
  itype   : 000000
  imm     : 00000000
```

例えば `32'h01234567` 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 `32'h00000010`、`imm` が `32'h00000012` になっており、正しくデコードできていることを確認できます。

3.8 レジスタの定義と読み込み

RV32I には、32 ビット幅のレジスタが 32 個用意されています。ただし、0 番目のレジスタの値は常に `0` です。

3.8.1 レジスタファイルを定義する

core モジュールにレジスタを定義します。レジスタの幅は XLEN(=32) ビットであるため、`UIntX` 型のレジスタの配列を定義します (リスト 3.37)。

▼リスト 3.37: 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 (register file) と呼ぶため、`regfile` という名前をつけています。

3.8.2 レジスタの値を読み込む

レジスタを定義したので、命令が使用するレジスタの値を取得します。

図 3.3 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 つのレジスタの値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値と即

値を利用します。

命令のビット列の中のソースレジスタの番号の場所は、命令形式が違っても共通の場所にあります。コードを簡単にするために、命令がレジスタの値を利用するかどうかに関係なく、常にレジスタの値を読み込むことにします（リスト 3.38）。

▼ リスト 3.38: 命令が使うレジスタの値を取得する (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 式を使うことで、0 番目のレジスタが指定されたときは、値が常に 0 になるようにします。

レジスタの値を読み込んでいることを確認するために、デバッグ表示にソースレジスタの値を追加します（リスト 3.39）。\$display システムタスクで、命令のレジスタ番号と値をデバッグ表示します。

▼ リスト 3.39: レジスタの値をデバッグ表示する (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);
    }
}
```

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値が初期化されておらず、0 番目のレジスタの値以外は不定値^{*5}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。always_ff ブロックの if_reset で、i 番目 ($0 < i < 32$) のレジスタの値を i + 100 で初期化します（リスト 3.40）。

^{*5} Verilator はデフォルト設定では不定値に対応していないため、不定値は 0 になります

▼リスト 3.40: レジスタを適当な値で初期化する (core.veryl)

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

レジスタの値を読み込んでいることを確認します（リスト 3.41）。

▼リスト 3.41: レジスタ読み込みのデバッグ

```
$ make build
$ make sim
$ obj_dir/sim sample.hex 7
00000000 : 01234567
    itype   : 000010
    imm     : 00000012
    rs1[ 6] : 0000006a
    rs2[18] : 00000076
00000004 : 89abcdef
    itype   : 100000
    imm     : ffffb09a
    rs1[23] : 0000007b
    rs2[26] : 0000007e
00000008 : deadbeef
    itype   : 100000
    imm     : fffffb5ea
    rs1[27] : 0000007f
    rs2[10] : 0000006e
0000000c : cafebebe
    itype   : 000000
    imm     : 00000000
    rs1[29] : 00000081
    rs2[15] : 00000073
```

`32'h01234567` は `jalr x10, 18(x6)` です。JALR 命令は、ソースレジスタ `x6` を使用します。`x6` はレジスタ番号が 6 であることを表しており、値は 106 に初期化しています。これは 16 進数で `32'h0000006a` です。

シミュレーションと結果が一致していることを確認してください。

3.9 ALU による計算の実装

レジスタと即値が揃い、命令で使用するデータが手に入るようになりました。基本整数命令セットの命令では、足し算や引き算、ビット演算などの簡単な整数演算を行います。それでは、CPU の計算を行う部品である **ALU**(Arithmetic Logic Unit) を作成します。

3.9.1 ALU モジュールを作成する

レジスタと即値の幅は 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。符号付き整数を利用するため、eei モジュールに XLEN ビットの符号付き整数型を定義します（リスト 3.42）。

▼ リスト 3.42: XLEN ビットの符号付き整数型を定義する (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 3.43）。

▼ リスト 3.43: 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 sll: UIntX = op1 << op2[4:0];
    let srl: UIntX = op1 >> op2[4:0];
    let sra: SIntX = $signed(op1) >>> op2[4:0];

    let slt : UIntX = {1'b0 repeat XLEN - 1, $signed(op1) <: $signed(op2)};
    let sltu: UIntX = {1'b0 repeat XLEN - 1, op1 <: op2};

    always_comb {
        if ctrl.is_aluop {
            case ctrl.funct3 {
                3'b000: result = if ctrl.itype == InstType::I | ctrl.funct7 == 0 {
                    add
                } else {
                    sub
                };
                3'b001: result = sll;
                3'b010: result = slt;
                3'b011: result = sltu;
                3'b100: result = op1 ^ op2;
                3'b101: result = if ctrl.funct7 == 0 {
                    srl
                } else {
                    sra
                };
                3'b110 : result = op1 | op2;
            }
        }
    }
}
```

```
        3'b111 : result = op1 & op2;
        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)。

▼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

仕様書で整数演算命令として定義されている命令 [7] は、funct3 と funct7 フィールドによって計算の種類を特定できます (表 3.4)。

▼表 3.4: 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	符号付き <=
3'b011	符号なし <=
3'b100	ビット単位 XOR
3'b101	右論理、右算術シフト
3'b110	ビット単位 OR
3'b111	ビット単位 AND

それ以外の命令は、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（リスト 3.33）。

always_comb ブロックでは、funct3 の case 文によって計算を選択します。funct3 だけでは選択できないとき、funct7 を使用します。

3.9.2 ALU モジュールをインスタンス化する

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 (リスト 3.44)。

▼ リスト 3.44: 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.iotype {
        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;
        }
    }
}

```

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 形式と B 形式は、レジスタの値とレジスタの値の演算を行います。 `op1` と `op2` は、レジスタの値 `rs1_data` と `rs2_data` になります。

I 形式、S 形式

I 形式と S 形式は、レジスタの値と即値の演算を行います。 `op1` と `op2` は、それぞれレジスタの値 `rs1_data` と即値 `inst_imm` になります。 S 形式はメモリの書き込み命令に利用されており、レジスタの値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 形式と 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` と `op2` は、それぞれ PC `inst_pc` と即値 `inst_imm` になります。 J 形式は JAL 命令に利用されており、PC に即値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、PC に即値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。

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 (リスト 3.45)。 結果を受け取る用の変数として、 `alu_result` を指定します。

▼リスト 3.45: ALU のインスタンス化 (core.veryl)

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

3.9.3 ALU モジュールをテストする

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` と `op2`、`alu_result` をデバッグ表示します (リスト 3.46)。

▼リスト 3.46: ALU の結果をデバッグ表示する (core.veryl)

```
always_ff {
    if if_fifo_rvalid {
        $display("%h : %h", inst_pc, inst_bits);
        $display("  type : %b", inst_ctrl.type);
        $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);
        $display("  op2 : %h", op2);
        $display("  alu res : %h", alu_result);
    }
}
```

`src/sample.hex` を、次のように書き換えます (リスト 3.47)。

▼リスト 3.47: 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 (表 3.5)。

▼表 3.5: 命令の意味

アドレス	命令	命令形式	意味
0x00000000	addi x1, x0, 32	I 形式	$x1 = x0 + 32$
0x00000004	auipc x2, 256	U 形式	$x2 = pc + 256$
0x00000008	add x3, x1, x2	R 形式	$x3 = x1 + x2$

シミュレータを実行し、結果を確かめます (リスト 3.48)。

▼リスト3.48: ALUのデバッグ

```
$ make build
$ make sim
$ obj_dir/sim src/sample.hex 6
00000000 : 02000093
  itype   : 000010
  imm    : 00000020
  rs1[ 0] : 00000000
  rs2[ 0] : 00000000
  op1    : 00000000
  op2    : 00000020
  alu res : 00000020
00000004 : 00100117
  itype   : 010000
  imm    : 00100000
  rs1[ 0] : 00000000
  rs2[ 1] : 00000065
  op1    : 00000004
  op2    : 00100000
  alu res : 00100004
00000008 : 002081b3
  itype   : 000001
  imm    : 00000000
  rs1[ 1] : 00000065
  rs2[ 2] : 00000066
  op1    : 00000065
  op2    : 00000066
  alu res : 000000cb
```

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、命令を実行しても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、`32'h00000000` と表示されています。`op2` は即値です。即値は 32 であるため、`32'h00000020` 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 `32'h00000020` が表示されています。

auipc x2, 256

`op1` は PC です。`op1` には、命令のアドレス `0x00000004` が表示されています。`op2` は即値です。256 を 12bit 左にシフトした値 `32'h00100000` 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 `32'h00100004` 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、`32'h00000065` と表示されています。`op2` 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、`32'h00000066` 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 `32'h000000cb` 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、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、値をライトバック (write-back) すると呼びます。

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 ライトバック処理を実装する

書き込む対象のレジスタ (デスティネーションレジスタ) は、命令の rd フィールドによって番号で指定されます。デコード時に、レジスタに結果を書き込む命令かどうかを `InstCtrl.rwb_en` に格納しています (リスト 3.33)。

LUI 命令のときは即値をそのまま、それ以外の命令のときは ALU の結果をライトバックします (リスト 3.49)。

▼リスト 3.49: 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      = if inst_ctrl.is_lui {
    inst_imm
} else {
    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 ライトバック処理をテストする

デバッグ表示用の always_ff ブロックで、ライトバック処理の概要をデバッグ表示します (リスト 3.50)。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` システムタスクを呼び出します。

▼リスト 3.50: ライトバックのデバッグ表示 (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、結果を確かめます（リスト 3.51）。

▼リスト 3.51: ライトバックのデバッグ

```
$ make build
$ make sim
$ obj_dir/sim sample.hex 6
00000000 : 02000093
  itype   : 0000010
  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 を足した値（`32'h00000020`）を格納しています。

auipc x2, 256

x2 に、256 を 12 ビット左にシフトした値（`32'h00100000`）と PC（`32'h00000004`）を足した値（`32'h00100004`）を格納しています。

add x3, x1, x2

x1 は 1 つ目の命令で `32'h00000020` に、x2 は 2 つ目の命令で `32'h00100004` にされています。x3 に、x1 と x2 を足した結果 `32'h00100024` 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！

最後に、テストのためにレジスタの値を初期化していたコードを削除します（リスト 3.52）。

▼リスト 3.52: レジスタの初期化をやめる (core.veryl)

```
always_ff {
    if if_fifo_rvalid && inst_ctrl.rwb_en {
        regfile[rd_addr] = wb_data;
    }
}
```

3.11 ロード命令とストア命令の実装

RV32I には、メモリのデータを読み込む、書き込む命令として次の命令があります（表 3.6）。データを読み込む命令のことをロード命令、データを書き込む命令のことをストア命令と呼びます。2つを合わせてロードストア命令と呼びます。

▼表 3.6: RV32I のロード命令、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 で拡張する
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 で拡張する
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 3.44）。ストア命令は、rs2 の値をメモリに格納します。

3.11.1 LW、SW 命令を実装する

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 命令と 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、`src/memunit.veryl` に記述します（リスト 3.53）。

▼リスト 3.53: 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?
>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
>タ
rdata : output UIntX     , // ロード命令の結果 (stall = 1)
>0のときに有効)
stall : output logic     , // メモリアクセス命令が完了したか?
>ていない
membus: modport membus_if:<MEM_DATA_WIDTH, XLEN>::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.iotype == 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: logic<MEM_DATA_WIDTH>;

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とき、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 (リスト 3.33)。

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` は 1 になっています。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

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` \rightarrow `WaitReady` \rightarrow `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止める必要があります。

CPU の実行が止まることを、CPU がストール (Stall) すると呼びます。メモリアクセス中のストールを実現するために、memunit モジュールには処理中かどうかを表す `stall` フラグを実装しています。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` の値を決定します (表 3.7)。

▼表 3.7: `stall` の値の決定方法

状態	<code>stall</code> が 1 になる条件
<code>Init</code>	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
<code>WaitReady</code>	常に 1
<code>WaitValid</code>	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

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 命令か 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。この問題は後の章で対応するため、全てのロードストア命令は、アクセスするビット幅で割り切れるアドレスにしかアクセスしないということにしておきます。

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 3.54)。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。

▼リスト 3.54: `inst_valid` と `inst_is_new` の定義 (core.veryl)

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

次に、`inst_is_new` の値を更新します (リスト 3.55)。命令が現在のクロックで供給されたかどうかは、FIFO の `rvalid` と `rready` を観測することでわかります。`rvalid` が 1 のとき、`rready` 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。`rready` 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 かどうかは考えません)。

▼ リスト 3.55: `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;
        }
    }
}
```

`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を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に使用されているため、`memunit` モジュールのために使用できません。また、`memory` モジュールは同時に 2 つの操作を受け付けられません。

この問題を、`core` モジュールにメモリとのインターフェースを 2 つ用意して `top` モジュールで調停することにより回避します。

まず、`core` モジュールに命令フェッチ用のポート `i_membus` と、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 (リスト 3.56)。

▼ リスト 3.56: `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::<ILEN, XLEN>::master ,
    d_membus: modport membus_if::<MEM_DATA_WIDTH, XLEN>::master ,
) {
```

命令フェッチ用のポートが `membus` から `i_membus` に変更されたため、既存の `membus` を `i_membus` に置き換えてください (リスト 3.57)。

▼ リスト 3.57: `membus` を `i_membus` に置き換える (core.veryl)

```
// FIFOに2個以上空きがあるとき、命令をフェッチする
i_membus.valid = if_fifo.wready_two;
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3.58）。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。

▼リスト3.58: メモリへのアクセス要求の調停 (top.veryl)

```
inst membus : membus_if::<MEM_DATA_WIDTH, MEM_ADDR_WIDTH>;
inst i_membus: membus_if::<ILEN, XLEN>; // 命令フェッチ用
inst d_membus: membus_if::<MEM_DATA_WIDTH, XLEN>; // ロードストア命令用

var memarb_last_i: logic;

// メモリアクセスを調停する
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 = addr_to_memaddr(d_membus.addr);
        membus.wen = d_membus.wen;
        membus.wdata = d_membus.wdata;
    } else {
        membus.addr = addr_to_memaddr(i_membus.addr);
        membus.wen = 0; // 命令フェッチは常に読み込み
        membus.wdata = 'x;
    }
}
```

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ロードストア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 3.59）。

▼ リスト 3.59: 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 モジュール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 3.60）。

▼ リスト 3.60: 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

memunit モジュールが処理中のときは命令を FIFO から取り出すのを止める処理と、ロード命令で読み込んだデータをレジスタにライトバックする処理を実装します。

memunit モジュールが処理中のとき、FIFO から命令を取り出すのを止めます（リスト 3.61）。

▼ リスト 3.61: 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;
```

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。

次に、ロード命令の結果をレジスタにライトバックします（リスト 3.62）。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`）、`memu_rdata` を `wb_data` に設定します。

▼ リスト 3.62: 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_lui {
    inst_imm
} else if inst_ctrl.is_load {
    memu_rdata
} else {
    alu_result
};
```

ところで、現在のコードでは memunit の処理が終了していないときも値をライトバックし続けています。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、次のようにコードを変更します（リスト 3.63）。

▼ リスト 3.63: 命令の実行が終了したときにのみライトバックする (core.veryl)

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

デバッグ表示も同様で、ライトバックするときにのみデバッグ表示します（リスト 3.64）。

▼ リスト 3.64: ライトバックするときに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 命令と SW 命令が正しく動作していることを確認するために、デバッグ表示に次のコードを追加します（リスト 3.65）。

▼ リスト 3.65: メモリモジュールの状態をデバッグ表示する (core.veryl)

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

ここからのテストは実行するクロック数が多くなります。そこで、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 3.66）。

▼ リスト 3.66: 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、`src/sample.hex` を次のように変更します（リスト 3.67）。

▼ リスト 3.67: 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
deadbeef // 0x20

```

プログラムは次のようになっています（表 3.8）。

▼ 表 3.8: メモリに格納する命令

アドレス	命令	意味
0x00000000	lw x10, 0x20(x0)	x10 に、アドレスが 0x20 のデータを読み込む
0x00000004	addi x11, x0, 0x400	x11 = 0x400
0x00000008	sw x11, 0x20(x0)	アドレス 0x20 に x11 の値を書き込む
0x0000000c	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 `0x00000020` には、データ `32'hdeadbeef` を格納しています。1 つ目の命令で `32'hdeadbeef` が読み込まれ、3 つ目の命令で `32'h00000400` を書き込み、4 つ目の命令で `32'h00000400` が読み込まれます。

シミュレータを実行し、結果を確かめます（リスト 3.68）。

▼ リスト 3.68: LW、SW 命令のテスト

```

$ make build
$ make sim
$ obj_dir/sim src/sample.hex 13

#
#          4
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
...
#                         6
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
...
#                         13
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 命令を実装する

LB と LBU と SB 命令は 8 ビット単位、LH と LHU と 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できます。

LB、LBU、LH、LHU、LW 命令は、funct3 の値で区別できます (表 3.9)。funct3 の上位 1 ビットが 1 のとき、符号拡張を行います。

▼表 3.9: ロード命令の funct3

funct3	命令
3'b000	LB
3'b100	LBU
3'b001	LH
3'b101	LHU
3'b010	LW

まず、何度も記述することになる値を短い名前 (`W`、`D`、`sext`) で定義します (リスト 3.69)。`sext` は、符号拡張を行うかどうかを示す変数です。

▼リスト 3.69: W、D、sext の定義 (memunit.veryl)

```
const W  : u32          = XLEN;
let D  : logic<MEM_DATA_WIDTH> = membus.rdata;
let sext: logic          = ctrl.funct3[2] == 1'b0;
```

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 3.70）。

▼リスト 3.70: 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.funct3[1:0] {
    2'b00 : case addr[1:0] {
        0      : {sext & D[7] repeat W - 8, D[7:0]},
        1      : {sext & D[15] repeat W - 8, D[15:8]},
        2      : {sext & D[23] repeat W - 8, D[23:16]},
        3      : {sext & D[31] repeat W - 8, D[31:24]},
        default: 'x,
    },
    2'b01 : case addr[1:0] {
        0      : {sext & D[15] repeat W - 16, D[15:0]},
        2      : {sext & D[31] repeat W - 16, D[31:16]},
        default: 'x,
    },
    2'b10 : D,
    default: 'x,
};
```

ロードした値の拡張を行うとき、値の最上位ビットと `sext` を AND 演算した値を使って拡張します。これにより、符号拡張するときは最上位ビットの値が、ゼロで拡張するときは `0` が拡張に利用されます。

3.11.3 SB、SH 命令を実装する

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、一部のみの書き込みを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（リスト 3.71、リスト 3.72、リスト 3.73）。

▼リスト 3.71: wmask の定義 (membus_if.veryl)

```
var wmask : logic<DATA_WIDTH / 8>;
```

▼リスト 3.72: modport master に wmask を追加する (membus_if.veryl)

```
modport master {
    ...
    wmask : output,
    ...
}
```

▼リスト 3.73: modport slave に wmask を追加する (membus_if.veryl)

```
modport slave {
    ...
    wmask : input ,
    ...
}
```

`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。バイト単位で指定するため、`wmask` の幅は `DATA_WIDTH / 8` 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 3.74）。

▼リスト 3.74: 書き込みマスクをサポートする memory モジュール (memory.veryl)

```
module memory::<DATA_WIDTH: const, ADDR_WIDTH: const> #(
    param FILEPATH_IS_ENV: logic = 0, // FILEPATHが環境変数名かどうか
    param FILEPATH : string = "", // メモリの初期化用ファイルのパス、または環境変数名
) (
    clk : input clock ,
    rst : input reset ,
    membus: modport membus_if::<DATA_WIDTH, ADDR_WIDTH>::slave,
) {
    type DataType = logic<DATA_WIDTH> ;
    type MaskType = logic<DATA_WIDTH / 8>;

    var mem: DataType [2 ** ADDR_WIDTH];

    // 書き込みマスクをDATA_WIDTHに展開した値
    var wmask_expand: DataType;
    always_comb {
        for i: u32 in 0..DATA_WIDTH {
            wmask_expand[i] = wmask_saved[i / 8];
        }
    }

    initial {
        // memを初期化する
        if FILEPATH != "" {
            if FILEPATH_IS_ENV {
                $readmemh(util::get_env(FILEPATH), mem);
            } else {
                $readmemh(FILEPATH, mem);
            }
        }
    }
}
```

```

}

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

var addr_saved : logic  <ADDR_WIDTH>;
var wdata_saved: DataType          ;
var wmask_saved: MaskType         ;
var rdata_saved: DataType          ;

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

always_ff {
    if state == State::WriteValid {
        mem[addr_saved[ADDR_WIDTH - 1:0]] = wdata_saved & wmask_expand | rdata_saved & ~wmask_saved;
    }
}

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[membus.addr[ADDR_WIDTH - 1:0]];
                addr_saved    = membus.addr[ADDR_WIDTH - 1:0];
                wdata_saved   = membus.wdata;
                wmask_saved   = membus.wmask;
                rdata_saved   = mem[membus.addr[ADDR_WIDTH - 1:0]];
                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格納し、状態を `State::WriteValid` に移動する。

State::WriteValid

書き込みマスクつ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格納します。また、指定されたアドレスのデータを `rdata_saved` に格納します。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` も調停します (リスト 3.75)。

▼ リスト 3.75: wmask の調停 (top.veryl)

```

membus.valid = i_membus.valid | d_membus.valid;
if d_membus.valid {
    membis.addr  = addr_to_memaddr(d_membus.addr);
    membis.wen   = d_membus.wen;
    membis.wdata = d_membus.wdata;
    membis.wmask = d_membus.wmask;
} else {
    membis.addr  = addr_to_memaddr(i_membus.addr);
    membis.wen   = 0; // 命令フェッチは常に読み込み
    membis.wdata = 'x;
    membis.wmask = 'x;
}
```

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したので、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 3.76、リスト 3.77)。

▼ リスト 3.76: req_wmask の定義 (memunit.veryl)

```
var req_wmask: logic<MEM_DATA_WIDTH / 8>;
```

▼ リスト 3.77: membis に wmask を設定する (memunit.veryl)

```

// メモリアクセス
membus.valid = state == State::WaitReady;
membus.addr  = req_addr;
membus.wen   = req_wen;
membus.wdata = req_wdata;
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 3.78、リスト 3.79）。

▼ リスト 3.78: `if_reset` で `req_wmask` を初期化する (memunit.veryl)

```
if_reset {
    state      = State::Init;
    req_wen   = 0;
    req_addr  = 0;
    req_wdata = 0;
    req_wmask = 0;
} else {
```

▼ リスト 3.79: 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.funct3[1:0] {
    2'b00 : 4'b1 << addr[1:0], ← SB命令のとき、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] { ← SH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, ← SW命令のとき、全体に書き込む
    default: 'x,
};
```

3.11.4 LB、LBU、LH、LHU、SB、SH 命令をテストする

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 3.80: src/sample_lbh.hex

```
02000083 // lb x1, 0x20(x0) : x1 = ffffffef
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 3.81: 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 ジャンプ命令、分岐命令の実装

まだ重要な命令を実装できていません。プログラムで分岐やループを実現するためにはジャンプや分岐をする命令が必要です。RV32I には、仕様書 [8] に次の命令が定義されています (表 3.10)。

▼表 3.10: ジャンプ命令、分岐命令

命令	形式	動作
JAL	J 形式	PC+ 即値に無条件ジャンプする。rd に PC+4 を格納する
JALR	I 形式	rs1+ 即値に無条件ジャンプする。rd に PC+4 を格納する
BEQ	B 形式	rs1 と rs2 が等しいとき、PC+ 即値にジャンプする
BNE	B 形式	rs1 と rs2 が異なるとき、PC+ 即値にジャンプする
BLT	B 形式	rs1(符号付き整数) が rs2(符号付き整数) より小さいとき、PC+ 即値にジャンプする
BLTU	B 形式	rs1(符号なし整数) が rs2(符号なし整数) より小さいとき、PC+ 即値にジャンプする
BGE	B 形式	rs1(符号付き整数) が rs2(符号付き整数) より大きいとき、PC+ 即値にジャンプする
BGEU	B 形式	rs1(符号なし整数) が rs2(符号なし整数) より大きいとき、PC+ 即値にジャンプする

ジャンプ命令は、無条件でジャンプするため、**無条件ジャンプ** (Unconditional Jump) と呼びます。分岐命令は、条件付きで分岐するため、**条件分岐** (Conditional Branch) と呼びます。

3.12.1 JAL、JALR 命令を実装する

まず、無条件ジャンプを実装します。

JAL (Jump And Link) 命令は、PC+ 即値でジャンプ先を指定します。Link とは、rd レジスタに PC+4 を記録しておくことで、分岐元に戻れるようにしておく操作のことです。即値の幅は 20 ビットです。PC の下位 1 ビットは常に 0 なため、即値を 1 ビット左シフトして符号拡張した値を PC に加算します (即値の生成はリスト 3.33 を確認してください)。JAL 命令でジャンプ可能な範囲は、PC ± 1MiB です。

JALR (Jump And Link Register) 命令は、rs1+ 即値でジャンプ先を指定します。即値は I 形式の即値です。JAL 命令と同様に、rd レジスタに PC+4 を格納 (link) します。JALR 命令でジャンプ可能な範囲は、rs1 レジスタの値 ± 4KiB です。

inst_decoder モジュールは、JAL 命令か JALR 命令のとき、`InstCtrl.rwb_en` を 1、`InstCtrl.is_aluop` を 0、`InstCtrl.is_jump` を 1 としてデコードします。

無条件ジャンプであるかどうかは `InstCtrl.is_jump` で確かめられます。また、`InstCtrl.is_aluop` が 0 なため、ALU は常に加算を行います。加算の対象のデータが、JAL 命令 (J 形式) なら PC と即値、JALR 命令 (I 形式) なら rs1 と即値になっていることを確認してください (リスト 3.44)。

無条件ジャンプの実装

それでは、無条件ジャンプを実装します。まず、ジャンプ命令を実行するときにライトバックする値を `inst_pc + 4` にします (リスト 3.82)。

▼リスト 3.82: pc + 4 を書き込む (core.veryl)

```
let wb_data: UIntX    = if inst_ctrl.is_lui {
    inst_imm
} else if inst_ctrl.is_jump {
    inst_pc + 4
} else if inst_ctrl.is_load {
    memu_rdata
} else {
    alu_result
};
```

次に、次にフェッチする命令をジャンプ先の命令に変更します。フェッチ先の変更が発生を示す信号 `control_hazard` と、新しいフェッチ先を示す信号 `control_hazard_pc_next` を作成します（リスト 3.83、リスト 3.84）。

▼リスト 3.83: control_hazard と control_hazard_pc_next の定義 (core.veryl)

```
var control_hazard      : logic;
var control_hazard_pc_next: Addr ;
```

▼リスト 3.84: control_hazard と control_hazard_pc_next の割り当て (core.veryl)

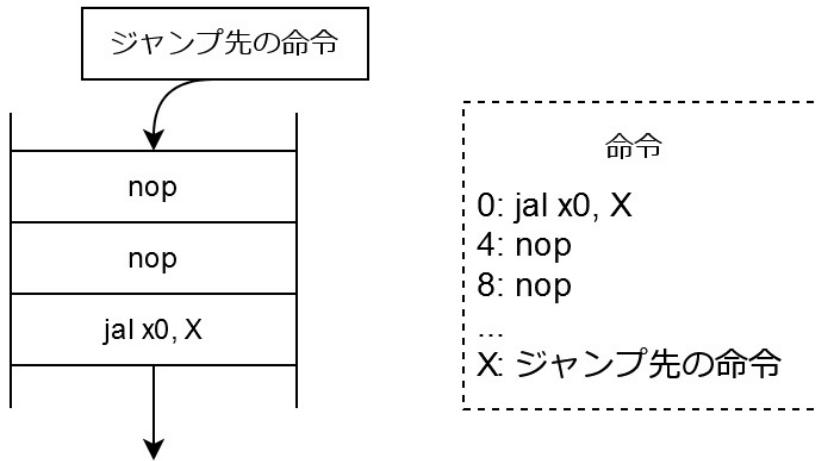
```
assign control_hazard      = inst_valid && inst_ctrl.is_jump;
assign control_hazard_pc_next = alu_result;
```

`control_hazard` を利用して `if_pc` を更新し、新しく命令をフェッチしなおすようにします（リスト 3.85）。

▼リスト 3.85: PC をジャンプ先に変更する (core.veryl)

```
always_ff {
    if_reset {
        ...
    } else {
        if control_hazard {
            if_pc      = control_hazard_pc_next;
            if_is_requested = 0;
            if_fifo_wvalid = 0;
        } else {
            if if_is_requested {
                ...
            }
            // IFのFIFOの制御
            if if_is_requested && i_membus.rvalid {
                ...
            }
        }
    }
}
```

ここで、新しく命令をフェッチしなおすようにしても、ジャンプ命令によって実行されることがなくなった命令が FIFO に残っていることがあることに注意する必要があります（図 3.4）。



▲図 3.4: ジャンプ命令とジャンプ先の間に余計な命令が入ってしまっている

実行すべきではない命令を実行しないようにするために、ジャンプ命令を実行するときに、FIFO をリセットします。

FIFO に、中身をリセットするための信号 `flush` を実装します (リスト 3.86)。

▼リスト 3.86: ポートに `flush` を追加する (fifo.vervyl)

```
module fifo #(
    param DATA_TYPE: type = logic,
    param WIDTH      : u32    = 2      ,
) (
    clk      : input  clock      ,
    rst      : input  reset      ,
    flush   : input  logic      ,
    wready: output logic      ,
```

`flush` が 1 のとき、`head` と `tail` を 0 に初期化することで FIFO を空にします (リスト 3.87)。

▼リスト 3.87: `flush` が 1 のとき、FIFO を空にする (fifo.vervyl)

```
always_ff {
    if_reset {
        head = 0;
        tail = 0;
    } else {
        if flush {
            head = 0;
            tail = 0;
        } else {
            if wready && wvalid {
                mem[tail] = wdata;
```

```
        tail      = tail + 1;
    }
    if rready && rvalid {
        head = head + 1;
    }
}
```

core モジュールで、`control_hazard` と `flush` を接続し、FIFO をリセットします（リスト 3.88）。

▼リスト 3.88: ジャンプ命令のとき、FIFO をリセットする (core.veryl)

```
inst if_fifo: fifo #(
  DATA_TYPE: if_fifo_type,
  WIDTH     : 3
) (
  clk      ,
  rst      ,
  flush : control_hazard,
  ...
);
```

無条件ジャンプのテスト

簡単なテストを作成し、動作をテストします（リスト 3.89、リスト 3.90）。

▼ リスト 3.89: sample_jump.hex

```
0100006f // 0: jal x0, 0x10 : 0x10にジャンプする
deadbeef // 4:
deadbeef // 8:
deadbeef // c:
01800093 // 10: addi x1, x0, 0x18
00808067 // 14: jalr x0, 8(x1) : x1+8=0x20にジャンプする
deadbeef // 18:
deadbeef // 1c:
fe1fff06f // 20: jal x0, -0x20 : 0にジャンプする
```

▼リスト 3.90: テストの実行

```
$ make build
$ make sim
$ obj_dir/sim src/sample_jump.hex 17
#
#                                     4
00000000 : 0100006f
  reg[ 0] <= 00000004 ← rd = PC + 4
#
#                                     8
00000010 : 01800093 ← 0x00 → 0x10にジャンプしている
  reg[ 1] <= 00000018
#
#                                     9
00000014 : 00808067
  reg[ 0] <= 00000018 ← rd = PC + 4
#
#                                     13
```

```

00000020 : fe1ff06f ← 0x14 → 0x20にジャンプしている
  reg[ 0] <= 00000024 ← rd = PC + 4
#
#                                17
00000000 : 0100006f ← 0x20 → 0x00にジャンプしている
  reg[ 0] <= 00000004

```

3.12.2 条件分岐命令を実装する

条件分岐命令はすべて B 形式で、PC+ 即値で分岐先を指定します。それぞれの命令は、命令の funct3 フィールドで判別できます (表 3.11)。

▼表 3.11: 条件分岐命令と funct3

funct3	命令	演算
3'b000	BEQ	==
3'b001	BNE	!=
3'b100	BLT	符号付き <=
3'b101	BGE	符号付き >
3'b110	BLTU	符号なし <=
3'b111	BGEU	符号なし >

条件分岐の実装

分岐の条件が成立するかどうかを判定するモジュールを作成します。`src/brunit.veryl` を作成し、次のように記述します (リスト 3.91)。

▼リスト 3.91: brunit.veryl

```

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

module brunit (
  funct3: input logic<3>,
  op1   : input UIntX   ,
  op2   : input UIntX   ,
  take  : output logic  , // 分岐が成立するか否か
) {
  let beq : logic = op1 == op2;
  let blt : logic = $signed(op1) <: $signed(op2);
  let bltu: logic = op1 <: op2;

  always_comb {
    case funct3 {
      3'b000 : take = beq;
      3'b001 : take = !beq;
      3'b100 : take = blt;
      3'b101 : take = !blt;
      3'b110 : take = bltu;
      3'b111 : take = !bltu;
    }
  }
}

```

```
        default: take = 0;  
    }  
}  
}
```

brunit モジュールは、`funct3` に応じて `take` の条件を切り替えます。分岐が成立するときに `take` が 1 になります。

brunit 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 3.92）。命令が B 形式のとき、`op1` は `rs1_data`、`op2` は `rs2_data` になっていることを確認してください（リスト 3.44）。

▼リスト 3.92: brunit モジュールのインスタンス化 (core.veryl)

```
var brunit_take: logic;  
  
inst bru: brunit (  
    funct3: inst_ctrl.funct3,  
    op1  
    op2  
    take  : brunit_take  
)
```

命令が条件分岐命令で `brunit_take` が 1 のとき、次の PC を PC + 即値にします (リスト 3.93、リスト 3.94)。

▼リスト 3.93: 命令が条件分岐命令か判定する関数 (core.veryl)

```
// 命令が分岐命令かどうかを判定する
function inst_is_br (
    ctrl: input InstCtrl,
) -> logic
{
    return ctrl.itype == InstType::B;
}
```

▼リスト 3.94: 分岐成立時の PC の設定 (core.vver1)

```
    assign control_hazard      = inst_valid && (inst_ctrl.is_jump || inst_is_br(inst_ctrl) &  
& brunit_take);  
    assign control_hazard_pc_next = if inst_is_br(inst_ctrl) {  
        inst_pc + inst_imm  
    } else {  
        alu_result  
    };
```

`control_hazard` は、命令が無条件ジャンプ命令か、命令が条件分岐命令かつ分岐が成立するときに 1 になります。 `control_hazard_pc_next` は、無条件ジャンプ命令のときは `alu_result`、条件分岐命令のときは PC + 即値になります。

条件分岐命令のテスト

条件分岐命令を実行するとき、分岐の成否をデバッグ表示します。デバッグ表示を行っている always_ff ブロック内に、次のコードを追加します（リスト 3.95）。

▼ リスト 3.95: 分岐判定のデバッグ表示 (core.veryl)

```
if inst_is_br(inst_ctrl) {
    $display("  br take  : %b", brunut_take);
}
```

簡単なテストを作成し、動作をテストします（リスト 3.96、リスト 3.97）。

▼ リスト 3.96: sample_br.hex

```
00100093 //  0: addi x1, x0, 1
10100063 //  4: beq x0, x1, 0x100
00101863 //  8: bne x0, x1, 0x10
deadbeef //  c:
deadbeef // 10:
deadbeef // 14:
0000d063 // 18: bge x1, x0, 0
```

▼ リスト 3.97: テストの実行

```
$ make build
$ make sim
$ obj_dir/sim src/sample_br.hex 15
#          4
00000000 : 00100093 ← x1に1を代入
#          5
00000004 : 10100063
  op1      : 00000000
  op2      : 00000001
  br take  : 0 ← x0 != x1なので不成立
#          6
00000008 : 00101863
  op1      : 00000000
  op2      : 00000001
  br take  : 1 ← x0 != x1なので成立
#          10
00000018 : 0000d063 ← 0x08 → 0x18にジャンプ
  br take  : 1 ← x1 > x0なので成立
#          14
00000018 : 0000d063 ← 0x18 → 0x18にジャンプ
  br take  : 1
```

BLT、BLTU、BGEU 命令は後の章で紹介する riscv-tests でテストします。



実装していない RV32I の命令

メモリフェンス命令、ECALL 命令、EBREAK 命令は後の章で実装します。

第 4 章

Zicsr 拡張の実装

4.1 CSR とは何か?

前の章では、RISC-V の基本整数命令セットである RV32I を実装しました。既に簡単なプログラムを動かせますが、例外や割り込み、ページングなどの機能がありません^{*1}。このような機能は CSR を介して提供されます。

RISC-V には、CSR(Control and Status Register) というレジスタが 4096 個存在しています。例えば `mtvec` というレジスタは、例外や割り込みが発生したときのジャンプ先のアドレスを格納しています。RISC-V の CPU は、CSR の読み書きによって制御 (Control) や状態 (Status) の読み取りを行います。

CSR の読み書きを行う命令は、Zicsr 拡張によって定義されています (表 4.1)。本章では、Zicsr に定義されている命令、RV32I に定義されている ECALL 命令、MRET 命令、`mtvec` レジスタ、`mepc` レジスタ、`mcause` レジスタを実装します。

▼表 4.1: Zicsr 拡張に定義されている命令

命令	作用
CSRRW	CSR に rs1 を書き込み、元の CSR の値を rd に書き込む
CSRRWI	CSRRW の rs1 を、即値をゼロ拡張した値に置き換えた動作
CSRRS	CSR と rs1 をビット OR した値を CSR に書き込み、元の CSR の値を rd に書き込む
CSRRSI	CSRRS の rs1 を、即値をゼロ拡張した値に置き換えた動作
CSRRC	CSR と ~rs1(rs1 のビット NOT) をビット AND した値を CSR に書き込み、元の CSR の値を rd に書き込む
CSRRCI	CSRRC の rs1 を、即値をゼロ拡張した値に置き換えた動作

^{*1} それぞれの機能は実装するときに解説します

4.2 CSR 命令のデコード

まず、Zicsr に定義されている命令（表 4.1）をデコードします。

これらの命令の opcode は `SYSTEM` (`7'b1110011`) です。この値を eei パッケージに定義します（リスト 4.1）。

▼リスト 4.1: opcode 用の定数の定義 (eei.veryl)

```
const OP_SYSTEM: logic<7> = 7'b1110011;
```

次に、`InstCtrl` 構造体に、CSR を制御する命令であることを示す `is_csr` フラグを追加します（リスト 4.2）。

▼リスト 4.2: is_csr を追加する (corectrl.veryl)

```
// 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_csr   : logic       , // CSR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
}
```

これでデコード処理を書く準備が整いました。`inst_decoder` モジュールの `InstCtrl` を生成している部分を変更します（リスト 4.3）。

▼リスト 4.3: OP_SYSTEM と is_csr を追加する (inst_decoder.veryl)

```
is_csrを追加
ctrl = {case op {
    OP_LUI    : {InstType::U, T, T, F, F, F, F},
    OP_AUIPC : {InstType::U, T, F, F, F, F, F},
    OP_JAL   : {InstType::J, T, F, F, T, F, F},
    OP_JALR  : {InstType::I, T, F, F, T, F, F},
    OP_BRANCH: {InstType::B, F, F, F, F, F, F},
    OP_LOAD  : {InstType::I, T, F, F, F, T, F},
    OP_STORE : {InstType::S, F, F, F, F, F, F},
    OP_OP    : {InstType::R, T, F, T, F, F, F},
    OP_OP_IMM: {InstType::I, T, F, T, F, F, F},
    OP_SYSTEM: {InstType::I, T, F, F, F, F, T},
    default   : {InstType::X, F, F, F, F, F, F},
}, f3, f7};
```

リスト 4.3 では、opcode が `OP_SYSTEM` な命令を、I 形式、レジスタに結果を書き込む、CSR を操作する命令であるということにしています。他の opcode の命令は CSR を操作しない命令であ

るということにしています。

CSRRW、CSRRS、CSRRC 命令は、rs1 レジスタの値を利用します。CSRRWI、CSRRSI、CSRRCI 命令は、命令のビット列中の rs1 にあたるビット列(5ビット)を 0 で拡張した値を利用します。それぞれの命令は funct3 で区別できます(表 4.2)。

▼表 4.2: Zicsr に定義されている命令 (funct3 による区別)

funct3	命令
3'b001	CSRRW
3'b101	CSRRWI
3'b010	CSRRS
3'b110	CSRRSI
3'b011	CSRRC
3'b111	CSRRCI

操作対象の CSR のアドレス(12ビット)は、命令のビットの上位 12 ビット(I 形式の即値)をそのまま利用します。

4.3 csrunit モジュールの実装

CSR を操作する命令のデコードができたので、CSR 関連の処理を行うモジュールを作成します。

4.3.1 csrunit モジュールを作成する

src/csrunit.veryl を作成し、次のように記述します(リスト 4.4)。

▼リスト 4.4: csrunit.veryl

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

module csrunit (
    clk      : input  clock      ,
    rst      : input  reset      ,
    valid    : input  logic      ,
    ctrl     : input  InstCtrl  ,
    csr_addr: input  logic  <12>,
    rs1      : input  UIntX     ,
    rdata    : output UIntX     ,
) {
    // CSRR(W|S|C)[I]命令かどうか
    let is_wsc: logic = ctrl.is_csr && ctrl.funct3[1:0] != 0;
}
```

csrunit モジュールの主要なポートの定義は表 4.3 のとおりです。まだ csrunit モジュールには CSR が一つもないため、中身が空になっています。

▼表4.3: csrunitモジュールのポート定義

ポート名	型	向き	意味
valid	logic	input	命令が供給されているかどうか
ctrl	InstCtrl	input	命令のInstCtrl
csr_addr	logic<12>	input	命令が指定するCSRのアドレス(命令の上位12ビット)
rs1	UIntX	input	CSRR(W S C)のときrs1の値、CSRR(W S C)Iのとき即値(5ビット)をゼロで拡張した値
rdata	UIntX	output	CSR命令によるCSR読み込みの結果

csrunitモジュールを、coreモジュールの中でインスタンス化します(リスト4.5)。

▼リスト4.5: csrunitモジュールのインスタンス化(core.veryl)

```
var csru_rdata: UIntX;

inst csru: csrunit (
    clk           ,
    rst           ,
    valid        : inst_valid        ,
    ctrl         : inst_ctrl         ,
    csr_addr: inst_bits[31:20],
    rs1          : if inst_ctrl.funct3[2] == 1 && inst_ctrl.funct3[1:0] != 0 {
        {1'b0 repeat XLEN - $bits(rs1_addr), rs1_addr} // rs1を0で拡張する
    } else {
        rs1_data
    },
    rdata: csru_rdata,
);
```

CSR命令の結果の受け取りのために変数 `csru_rdata` を作成し、csrunitモジュールをインスタンス化しています。

`csr_addr` ポートには命令の上位12ビットを設定しています。`rs1` ポートには、即値を利用する命令(CSRR(W|S|C)I)の場合は `rs1_addr` を `0` で拡張した値を、それ以外の命令の場合は `rs1` のデータを設定しています。

次に、CSRを読み込んだデータをレジスタにライトバックします。具体的には、`InstCtrl.is_csr` が `1` のとき、`wb_data` が `csru_rdata` になるようにします(リスト4.6)。

▼リスト4.6: CSR命令の結果がライトバックされるようにする(core.veryl)

```
let rd_addr: logic<5> = inst_bits[11:7];
let wb_data: UIntX    = if inst_ctrl.is_lui {
    inst_imm
} else if inst_ctrl.is_jump {
    inst_pc + 4
} else if inst_ctrl.is_load {
    memu_rdata
} else if inst_ctrl.is_csr {
    csru_rdata
}
```

```

} else {
    alu_result
};
```

最後に、デバッグ用の表示を追加します。デバッグ表示用の always_ff ブロックに、次のコードを追加してください（リスト 4.7）。

▼リスト 4.7: rdata をデバッグ表示する (core.veryl)

```

if inst_ctrl.is_csr {
    $display("  csr rdata : %h", csr_u_rdata);
}
```

これらのテストは、csrunit モジュールに CSR を追加してから行います。

4.3.2 mtvec レジスタを実装する

csrunit モジュールには、まだ CSR が定義されていません。1 つ目の CSR として、mtvec レジスタを実装します。

mtvec レジスタ、トラップ



▲図 4.1: mtvec のエンコーディング [9]

mtvec レジスタは、仕様書 [10] に定義されています。mtvec は、MXLEN ビットの WARL レジスタです。mtvec のアドレスは `12'h305` です。

MXLEN は misa レジスタに定義されていますが、今のところは XLEN と等しいという認識で問題ありません。WARL は Write Any Values, Reads Legal Values の略です。その名の通り、好きな値を書き込めますが読み出すときには合法な値^{*2}になっているという認識で問題ありません。

mtvec は、トラップ (Trap) が発生したときのジャンプ先 (Trap-Vector) の基準となるアドレスを格納するレジスタです。トラップとは、例外 (Exception)、または割り込み (Interrupt) により、CPU の制御を変更することです^{*3}。トラップが発生するとき、CPU は CSR を変更した後、mtvec に格納されたアドレスにジャンプします。

例外とは、命令の実行によって引き起こされる異常な状態 (unusual condition) のことです。例えば、不正な命令を実行しようとしたときには Illegal Instruction 例外が発生します。CPU は、例外が発生したときのジャンプ先 (対処方法) を決めておくことで、CPU が異常な状態に陥ったままにならないようにしています。

mtvec は BASE と MODE の 2 つのフィールドで構成されています。MODE はジャンプ先の

^{*2} 合法な値とは実装がサポートしている有効な値のことです

^{*3} トラップや例外、割り込みは Volume I の 1.6Exceptions, Traps, and Interrupts に定義されています

決め方を指定するためのフィールドですが、簡単のために常に `2'b00` (Direct モード) になるようになります。Direct モードのとき、トラップ時のジャンプ先は `BASE << 2` になります。

mtvec レジスタの実装

それでは、mtvec レジスタを実装します。まず、CSR のアドレスを表す列挙型を定義します (リスト 4.8)。

▼リスト 4.8: CsrAddr 型を定義する (csrunit.veryl)

```
// CSRのアドレス
enum CsrAddr: logic<12> {
    MTVEC = 12'h305,
}
```

次に、mtvec レジスタを作成します。MXLEN=XLEN としているので、型は `UIntX` にします (リスト 4.9)。

▼リスト 4.9: mtvec レジスタの定義 (csrunit.veryl)

```
// CSR
var mtvec: UIntX;
```

MODE は Direct モード (`2'b00`) しか対応していません。mtvec は WARL なレジスタなので、MODE フィールドには書き込めないようにする必要があります。これを制御するために mtvec レジスタの書き込みマスク用の定数を定義します (リスト 4.10)。

▼リスト 4.10: mtvec レジスタの書き込みマスクの定義 (csrunit.veryl)

```
// wmask
const MTVEC_WMASK: UIntX = 'hffff_fff0;
```

次に、書き込むデータ `wdata` の生成と、mtvec レジスタの読み込みを実装します (リスト 4.11)。

▼リスト 4.11: レジスタの読み込みと書き込むデータの作成 (csrunit.veryl)

```
var wmask: UIntX; // write mask
var wdata: UIntX; // write data

always_comb {
    // read
    rdata = case csr_addr {
        CsrAddr::MTVEC: mtvec,
        default: 'x,
    };
    // write
    wmask = case csr_addr {
        CsrAddr::MTVEC: MTVEC_WMASK,
        default: 0,
    };
    wdata = case ctrl.funct3[1:0] {
        2'b01: rs1,
```

```

    2'b10 : rdata | rs1,
    2'b11 : rdata & ~rs1,
    default: 'x,
  } & wmask;
}

```

always_comb ブロックで、`rdata` ポートに `csr_addr` に応じた CSR の値を割り当てます。`wdata` には、CSR に書き込むデータを割り当てます。CSR に書き込むデータは、書き込む命令 (CSRRW[I]、CSRRS[I]、CSRRC[I]) によって異なります。`rs1` ポートには `rs1` の値か即値が供給されているため、これと `rdata` を利用して `wdata` を生成しています。funct3 と演算の種類の関係は表 4.2 を参照してください。

最後に、mtvec レジスタへの書き込み処理を実装します。mtvec への書き込みは、命令が CSR 命令である場合 (`is_wsc`) にのみ行います (リスト 4.12)。

▼リスト 4.12: CSR への書き込み処理 (csrunit.veryl)

```

always_ff {
  if_reset {
    mtvec = 0;
  } else {
    if valid {
      if is_wsc {
        case csr_addr {
          CsrAddr::MTVEC: mtvec = wdata;
          default : {}
        }
      }
    }
  }
}

```

mtvec の初期値は `0` です。mtvec に `wdata` を書き込むとき、MODE が常に `2'b00` になります。

4.3.3 csrunit モジュールをテストする

mtvec レジスタの書き込み、読み込みができるることを確認します。

`test/sample_csr.hex` を作成し、次のように記述します (リスト 4.13)。

▼リスト 4.13: sample_csr.hex

```

305bd0f3 // 0: csrrwi x1, mtvec, 0b10111
30502173 // 4: csrrs x2, mtvec, x0

```

テストでは、CSRRWI 命令で mtvec に `32'b10111` を書き込んだ後、CSRRS 命令で mtvec の値を読み込みます。CSRRS 命令で読み込むとき、rs1 を x0(ゼロレジスタ) にすることで、mtvec の値を変更せずに読み込みます。

シミュレータを実行し、結果を確かめます (リスト 4.14)。

▼リスト4.14: mtvecの読み込み/書き込みテストの実行

```
$ make build
$ make sim
$ ./obj_dir/sim test/sample_csr.hex 5
#
#          4
00000000 : 305bd0f3 ← mtvecに32'b10111を書き込む
  itype      : 000010
  rs1[23]    : 00000000 ← CSRRWIなので、mtvecに32'b10111(=23)を書き込む
  csr rdata : 00000000 ← mtvecの初期値(0)が読み込まれている
  reg[ 1] <= 00000000
#
#          5
00000004 : 30502173 ← mtvecを読み込む
  itype      : 000010
  csr rdata : 00000014 ← mtvecに書き込まれた値を読み込んでいる
  reg[ 2] <= 00000014 ← 32'b10111のMODE部分がマスクされて、32'b10100 = 14になっている
```

mtvecのBASEフィールドにのみ書き込みが行われ、`32'h00000014`が読み込まれることを確認できます。

4.4 ECALL命令の実装

せっかくmtvecレジスタを実装したので、これを使う命令を実装します。

4.4.1 ECALL命令とは何か？

RV32Iには、意図的に例外を発生させる命令としてECALL命令が定義されています。ECALL命令を実行すると、現在の権限レベル(Privilege Level)に応じて表4.4のような例外が発生します。

権限レベルとは、権限(特権)を持つソフトウェアを実装するための機能です。例えばOS上で動くソフトウェアは、セキュリティのために、他のソフトウェアのメモリを侵害できないようにする必要があります。権限レベル機能があると、このような保護を、権限のあるOSが権限のないソフトウェアを管理するという風に実現できます。

権限レベルはいくつか定義されていますが、本章では最高の権限レベルであるMachineレベル(M-mode)しかなものとします。

▼表4.4: 権限レベルとECALLによる例外

権限レベル	ECALLによって発生する例外
M	Environment call from M-mode
S	Environment call from S-mode
U	Environment call from U-mode

mcause、mepc レジスタ

ECALL命令を実行すると例外が発生します。例外が発生すると mtvec にジャンプし、例外が発生した時の処理を行います。これだけでもいいのですが、例外が発生したときに、どこで(PC)、どのような例外が発生したのかを知りたいことがあります。これを知るために、RISC-Vには、どこで例外が発生したかを格納する mepc レジスタと、例外の発生原因を格納する mcause レジスタが存在しています。

CPUは例外が発生すると、mtvecにジャンプする前に、mepcに現在のPC、mcauseに発生原因を格納します。これにより、mtvecにジャンプしてから例外に応じた処理を実行できるようになります。

例外の発生原因は数値で表現されており、Environment call from M-mode例外には11が割り当てられています。

4.4.2 トランプを実装する

それでは、ECALL命令とトランプの仕組みを実装します。

定数の定義

まず、mepcとmcauseのアドレスを `CsrAddr` 型に追加します(リスト4.15)。

▼リスト4.15: mepcとmcauseのアドレスを追加する(csrunit.veryl)

```
// CSRのアドレス
enum CsrAddr: logic<12> {
    MTVEC = 12'h305,
    MEPC = 12'h341,
    MCAUSE = 12'h342,
}
```

次に、トランプの発生原因を表現する型 `CsrCause` を定義します。今のところ、発生原因は ECALL命令による Environment Call From M-mode例外しかありません(リスト4.16)。

▼リスト4.16: CsrCause型の定義(csrunit.veryl)

```
enum CsrCause: UIntX {
    ENVIRONMENT_CALL_FROM_M_MODE = 11,
}
```

最後に、mepcとmcauseの書き込みマスクを定義します(リスト4.17)。mepcに格納されるのは例外が発生した時の命令のアドレスです。命令は4バイトに整列して配置されているため、mepcの下位2ビットは常に `2'b00` になるようにします。

▼リスト4.17: mepcとmcauseの書き込みマスクの定義(csrunit.veryl)

```
const MTVEC_WMASK : UIntX = 'hffff_fff0;
const MEPC_WMASK : UIntX = 'hffff_fff0;
const MCAUSE_WMASK: UIntX = 'hffff_ffff;
```

mepcとmcauseレジスタの実装

mepcとmcauseレジスタを作成します。サイズはMXLEN(=XLEN)なため、型は UIntXとします(リスト4.18)。

▼リスト4.18: mepcとmcauseレジスタの定義(csrunit.veryl)

```
// CSR
var mtvec : UIntX;
var mepc : UIntX;
var mcause: UIntX;
```

次に、mepcとmcauseの読み込み処理と、書き込みマスクの割り当てを実装します。どちらもcase文にアドレスと値のペアを追加するだけです(リスト4.19、リスト4.20)。

▼リスト4.19: mepcとmcauseの読み込み(csrunit.veryl)

```
rdata = case csr_addr {
    CsrAddr::MTVEC : mtvec,
    CsrAddr::MEPC : mepc,
    CsrAddr::MCAUSE: mcause,
    default : 'x,
};
```

▼リスト4.20: mepcとmcauseの書き込みマスクの設定(csrunit.veryl)

```
wmask = case csr_addr {
    CsrAddr::MTVEC : MTVEC_WMASK,
    CsrAddr::MEPC : MEPC_WMASK,
    CsrAddr::MCAUSE: MCAUSE_WMASK,
    default : 0,
};
```

最後に、mepcとmcauseの書き込みを実装します。if_resetで値を0に初期化し、case文にmepcとmcauseの場合を実装します(リスト4.21)。

▼リスト4.21: mepcとmcauseの書き込み(csrunit.veryl)

```
always_ff {
    if_reset {
        mtvec = 0;
        mepc = 0;
        mcause = 0;
    } else {
        if valid {
            if is_wsc {
                case csr_addr {
                    CsrAddr::MTVEC : mtvec = wdata;
                    CsrAddr::MEPC : mepc = wdata;
                    CsrAddr::MCAUSE: mcause = wdata;
                    default : {}
                }
            }
        }
    }
}
```

```

    }
}
}
```

例外の実装

ECALL命令と、それによって発生するトラップを実装します。まず、csrunitモジュールにポートを追加します（リスト4.22）。

▼リスト4.22: csrunitモジュールにポートを追加する (csrunit.veryl)

```

module csrunit (
    clk      : input  clock      ,
    rst      : input  reset      ,
    valid    : input  logic      ,
    pc       : input  Addr       ,
    ctrl     : input  InstCtrl  ,
    rd_addr  : input  logic <5> ,
    csr_addr : input  logic <12>,
    rs1      : input  UIntX     ,
    rdata    : output UIntX     ,
    raise_trap: output logic   ,
    trap_vector: output Addr   ,
) {
```

それぞれの用途は次の通りです。

pc

現在処理している命令のアドレスを受け取ります。

例外が発生するとき、mepcにPCを格納するために使います。

rd_addr

現在処理している命令のrdの番号を受け取ります。

命令がECALL命令かどうかを判定するために使います。

raise_trap

例外が発生するとき、値を1にします。

trap_vector

例外が発生するとき、ジャンプ先のアドレスを出力します。

csrunitモジュールの中身を実装する前に、coreモジュールに例外発生時の動作を実装します。csrunitモジュールと接続するための変数を定義してcsrunitモジュールと接続します（リスト4.23、リスト4.24）。

▼リスト4.23: csrunitモジュールのポートの定義を変更する ① (core.veryl)

```

var csru_rdata      : UIntX;
var csru_raise_trap : logic;
var csru_trap_vector: Addr ;
```

▼リスト4.24: csrunitモジュールのポートの定義を変更する②(core.veryl)

```

inst csrunit: csrunit (
    clk           ,
    rst           ,
    valid        : inst_valid        ,
    pc           : inst_pc           ,
    ctrl         : inst_ctrl         ,
    rd_addr      ,
    csr_addr: inst_bits[31:20],
    rs1          : if inst_ctrl.funct3[2] == 1 && inst_ctrl.funct3[1:0] != 0 {
                    {1'b0 repeat XLEN - $bits(rs1_addr), rs1_addr} // rs1を0で拡張する
                } else {
                    rs1_data
                },
    rdata        : csrunit_rdata,
    raise_trap  : csrunit_raise_trap,
    trap_vector: csrunit_trap_vector,
);

```

次に、トラップするときにジャンプさせます。

例外が発生するとき、`csrunit_raise_trap`が1になります。`csrunit_trap_vector`がトラップ先になります。トラップするときの動作には、ジャンプと分岐命令の仕組みを利用します。`control_hazard`の条件に`csrunit_raise_trap`を追加して、トラップするときに`control_hazard_pc_next`を`csrunit_trap_vector`に設定します(リスト4.25)。

▼リスト4.25: 例外の発生時にジャンプさせる(core.veryl)

```

assign control_hazard = inst_valid && (
    csrunit_raise_trap ||
    inst_ctrl.is_jump ||
    inst_is_br(inst_ctrl) && brunit_take
);
assign control_hazard_pc_next = if csrunit_raise_trap {
    csrunit_trap_vector ←トラップするとき、trap_vectorに飛ぶ
} else if inst_is_br(inst_ctrl) {
    inst_pc + inst_imm
} else {
    alu_result
};

```

000000000000	00000	000	00000	1110011	ECALL
--------------	-------	-----	-------	---------	-------

▲図4.2: ECALL命令のフォーマット[6]

それでは、csrunitモジュールにトラップの処理を実装します。

ECALL命令は、I形式、即値は0、rs1とrdは0、funct3は0、opcodeはSYSTEMな命令です(図4.2)。これを判定するための変数を作成します(リスト4.26)。

▼リスト4.26: ECALL命令かどうかの判定(csrunit.veryl)

```
// ECALL命令かどうか
let is_ecall: logic = ctrl.is_csr && csr_addr == 0 && rs1[4:0] == 0 && ctrl.funct3 == 0 && >
>rd_addr == 0;
```

次に、例外が発生するかどうかを示す `raise_expt` と、例外の発生の原因を示す `expt_cause` を作成します。今のところ、例外は ECALL 命令によってのみ発生するため、`expt_cause` は実質的に定数になっています（リスト 4.27）。

▼リスト4.27: 例外とトラップの判定(csrunit.veryl)

```
// Exception
let raise_expt: logic = valid && is_ecall;
let expt_cause: UIntX = CsrCause::ENVIRONMENT_CALL_FROM_M_MODE;

// Trap
assign raise_trap = raise_expt;
let trap_cause: UIntX = expt_cause;
assign trap_vector = mtvec;
```

トラップが発生するかどうかを示す `raise_trap` には、例外が発生するかどうかを割り当てます。トラップの原因を示す `trap_cause` には、例外の発生原因を割り当てます。また、トラップ先には `mtvec` を割り当てます。

最後に、トラップに伴う CSR の変更を実装します。トラップが発生するとき、mepc レジスタに PC、mcause レジスタにトラップの発生原因を格納します（リスト 4.28）。

▼リスト4.28: トラップが発生したらCSRを変更する(csrunit.veryl)

```
always_ff {
    if_reset {
        ...
    } else {
        if valid {
            if raise_trap { ←トラップ時の動作
                mepc = pc;
                mcause = trap_cause;
            } else {
                if is_wsc {
                    ...
                }
            }
        }
    }
}
```

4.4.3 ECALL命令をテストする

ECALL命令をテストする前に、デバッグのために `$display` システムタスクで、例外が発生したかどうかと、トラップ先を表示します（リスト 4.29）。

▼リスト4.29: トラップの情報をデバッグ表示する(core.veryl)

```
if inst_ctrl.is_csr {
    $display(" csr rdata : %h", csru_rdata);
    $display(" csr trap : %b", csru_raise_trap);
```

```

    $display(" csr vec : %h", csru_trap_vector);
}

```

`test/sample_ecall.hex`を作成し、次のように記述します（リスト4.30）。

▼リスト4.30: `sample_ecall.hex`

```

30585073 // 0: csrrwi x0, mtvec, 0x10
00000073 // 4: ecall
00000000 // 8:
00000000 // c:
342020f3 // 10: csrrs x1, mcause, x0
34102173 // 14: csrrs x2, mepc, x0

```

CSRRWI命令でmtvecレジスタに値を書き込み、ECALL命令で例外を発生させてジャンプします。ジャンプ先では、mcauseレジスタとmepcレジスタの値を読み取ります。

シミュレータを実行し、結果を確かめます（リスト4.31）。

▼リスト4.31: ECALL命令のテストの実行

```

$ make build
$ make sim
$ ./obj_dir/sim test/sample_ecall.hex 10
#
#          4
00000000 : 30585073 ← CSRRWIでmtvecに書き込み
  rs1[16] : 00000000 ← 10(=16)をmtvecに書き込む
  csr trap : 0
  csr vec : 00000000
  reg[ 0 ] <= 00000000
#
#          5
00000004 : 00000073
  csr trap : 1 ← ECALL命令により、例外が発生する
  csr vec : 00000010 ← ジャンプ先は0x10
  reg[ 0 ] <= 00000000
#
#          9
00000010 : 342020f3
  csr rdata : 0000000b ← CSRRSでmcauseを読み込む
  reg[ 1 ] <= 0000000b ← Environment call from M-modeなのでb(=11)
#
#          10
00000014 : 34102173
  csr rdata : 00000004 ← CSRRSでmepcを読み込む
  reg[ 2 ] <= 00000004 ← 例外はアドレス4で発生したので4

```

ECALL命令によって例外が発生し、mcauseとmepcに書き込みが行われてからmtvecにジャンプしていることを確認できます。

ECALL命令の実行時にレジスタに値がライトバックされてしまっていますが、ECALL命令のrdは常に0番目のレジスタであり、0番目のレジスタは常に値が0になるため問題ありません。

4.5 MRET命令の実装

MRET命令は、トラップ先からトラップ元に戻るための命令です。MRET命令を実行すると、mepcレジスタに格納されたアドレスにジャンプします^{*4}。例えば、権限のあるOSから権限のないユーザー空間に戻るために利用します。

4.5.1 MRET命令を実装する

0011000	00010	00000	000	00000	1110011	MRET
---------	-------	-------	-----	-------	---------	------

▲図4.3: MRET命令のフォーマット [11]

まず、csrunitモジュールに供給されている命令がMRET命令かどうかを判定する変数`is_mret`を作成します(リスト4.32)。MRET命令は、上位12ビットは`12'b001100000010`、`rs1`は`0`、`funct3`は`0`、`rd`は`0`です(図4.3)。

▼リスト4.32: MRET命令の判定(csrunit.veryl)

```
// MRET命令かどうか
let is_mret: logic = ctrl.is_csr && csr_addr == 12'b0011000_00010 && rs1[4:0] == 0 && ctrl.function == "MRET" && funct3 == 0 && rd_addr == 0;
```

次に、csrunitモジュールにMRET命令が供給されているときにmepcにジャンプする仕組みを実装します。ジャンプするための仕組みには、トラップによってジャンプする仕組みを利用します(リスト4.33)。`raise_trap`に`is_mret`を追加し、トラップ先も変更します。

▼リスト4.33: MRET命令によってジャンプさせる(csrunit.veryl)

```
// Trap
assign raise_trap = raise_expt || (valid && is_mret);
let trap_cause : UIntX = expt_cause;
assign trap_vector = if raise_expt {
    mtvec
} else {
    mepc
};
```

例外が優先

`trap_vector`には、`is_mret`のときに`mepc`を割り当てるのではなく、`raise_expt`のときに`mtvec`を割り当てています。これは、MRET命令によって発生する例外があるからです。MRET命令の判定を優先すると、例外が発生するのに`mepc`にジャンプしてしまいます。

^{*4}他のCSRや権限レベルが実装されている場合は、他にも行うことがあります

4.5.2 MRET命令をテストする

mepcに値を設定してからMRET命令を実行することでmepcにジャンプするようなテストを作成します(リスト4.34)。

▼リスト4.34: sample_mret.hex

```
34185073 // 0: csrrwi x0, mepc, 0x10
30200073 // 4: mret
00000000 // 8:
00000000 // c:
00000013 // 10: addi x0, x0, 0
```

シミュレータを実行し、結果を確かめます(リスト4.35)。

▼リスト4.35: MRET命令のテストの実行

```
$ make build
$ make sim
$ ./obj_dir/sim test/sample_mret.hex 9
#          4
00000000 : 34185073 ← CSRRWIでmepcに書き込み
  rs1[16]   : 00000000 ← 0x10(=16)をmepcに書き込む
  csr trap  : 0
  csr vec   : 00000000
  reg[ 0] <= 00000000
#          5
00000004 : 30200073
  csr trap  : 1 ← MRET命令によってmepcにジャンプする
  csr vec   : 00000010 ← 10にジャンプする
#          9
00000010 : 00000013 ← 10にジャンプしている
```

MRET命令によってmepcにジャンプすることを確認できます。

MRET命令はレジスタに値をライトバックしていますが、ECALL命令と同じく0番目のレジスタが指定されるため問題ありません。

第 5 章

riscv-tests によるテスト

第 3 章では、RV32I の CPU を実装しました。簡単なテストを作成して動作を確かめましたが、まだテストできていない命令が複数あります。そこで、riscv-tests というテストを利用することで、CPU がある程度正しく動いているらしいことを確かめます。

5.1 riscv-tests とは何か?

riscv-tests は、RISC-V のプロセッサ向けのユニットテストやベンチマークテストの集合です。命令や機能ごとにテストが用意されており、これを利用することで簡単に実装を確かめられます。すべての命令のすべての場合を網羅するようなテストではないため、riscv-tests をパスしても、確実に実装が正しいとは言えないことに注意してください^{*1}。

GitHub の riscv-software-src/riscv-tests^{*2} からソースコードをダウンロードできます。

5.2 riscv-tests のビルド



riscv-tests のビルドが面倒、もしくはよく分からなくなってしまった方へ

<https://github.com/nananapo/riscv-tests-bin/tree/bin4>

完成品を上記の URL においておきます。core/test にコピーしてください。

5.2.1 riscv-tests をビルドする

まず、riscv-tests を clone します (リスト 5.1)。

^{*1} 実装の正しさを完全に確かめるには形式的検証 (formal verification) を行う必要があります

^{*2} <https://github.com/riscv-software-src/riscv-tests>

▼リスト 5.1: riscv-tests の clone

```
$ git clone https://github.com/riscv-software-src/riscv-tests
$ cd riscv-tests
$ git submodule update --init --recursive
```

riscv-tests は、プログラムの実行が `0x80000000` から始まると仮定した設定になっています。しかし、CPU はアドレス `0x00000000` から実行を開始するため、リンクにわたす設定ファイル `env/p/link.ld` を変更する必要があります（リスト 5.2）。

▼リスト 5.2: riscv-tests/env/p/link.ld

```
OUTPUT_ARCH( "riscv" )
ENTRY(_start)

SECTIONS
{
    . = 0x00000000; ←先頭を0x00000000に変更する
```

riscv-tests をビルドします。必要なソフトウェアがインストールされていない場合、適宜インストールしてください（リスト 5.3）。

▼リスト 5.3: riscv-tests のビルド

```
$ cd riscv-testsをcloneしたディレクトリ
$ autoconf
$ ./configure --prefix=core/testへのパス
$ make
$ make install
```

core/test に share ディレクトリが作成されます。

5.2.2 成果物を\$readmemh で読み込める形式に変換する

riscv-tests をビルドできましたが、これは `$readmemh` システムタスクで読み込める形式（以降 HEX 形式と呼びます）ではありません。これを CPU でテストを実行できるように、ビルドしたテストのバイナリファイルを HEX 形式に変換します。

まず、バイナリファイルを HEX 形式に変換する Python プログラム `test/bin2hex.py` を作成します（リスト 5.4）。

▼リスト 5.4: test/bin2hex.py

```
import sys

# 使い方を表示する
def print_usage():
    print(sys.argv[1])
    print("Usage:", sys.argv[0], "[bytes per line] [filename]")
    exit()
```

```

# コマンドライン引数を受け取る
args = sys.argv[1:]
if len(args) != 2:
    print_usage()
BYTES_PER_LINE = None
try:
    BYTES_PER_LINE = int(args[0])
except:
    print_usage()
FILE_NAME = args[1]

# バイナリファイルを読み込む
allbytes = []
with open(FILE_NAME, "rb") as f:
    allbytes = f.read()

# 値を文字列に変換する
bytestrs = []
for b in allbytes:
    bytestrs.append(format(b, '02x'))

# 00を足すことでBYTES_PER_LINEの倍数に揃える
bytestrs += ["00"] * (BYTES_PER_LINE - len(bytestrs) % BYTES_PER_LINE)

# 出力
results = []
for i in range(0, len(bytestrs), BYTES_PER_LINE):
    s = ""
    for j in range(BYTES_PER_LINE):
        s += bytestrs[i + BYTES_PER_LINE - j - 1]
    results.append(s)
print("\n".join(results))

```

このプログラムは、第二引数に指定されるバイナリファイルを、第一引数に与えられた数のバイト毎に区切り、16進数のテキストで出力します。

HEX ファイルに変換する前に、ビルドした成果物を確認する必要があります。例えば `test/share/riscv-tests/isa/rv32ui-p-add` は ELF ファイル^{*3}です。CPU は ELF を直接に実行する機能を持っていないため、`riscv64-unknown-elf-objcopy` を利用して、ELF ファイルを余計な情報を取り除いたバイナリファイルに変換します（リスト 5.5）。

▼ リスト 5.5: ELF ファイルをバイナリファイルに変換する

```
$ find share/ -type f -not -name ".*.dump" -exec riscv32-unknown-elf-objcopy -O binary {} {}.bin \;
```

最後に、objcopy で生成されたバイナリファイルを、Python プログラムで HEX ファイルに変換します（リスト 5.6）。

^{*3} ELF(Executable and Linkable Format) とは実行可能ファイルの形式です

▼ リスト 5.6: バイナリファイルを HEX ファイルに変換する

```
$ find share/ -type f -name "*.bin" -exec sh -c "python3 bin2hex.py {} > {}.hex" \;
```

5.3 テスト内容の確認

riscv-tests には複数のテストが用意されていますが、本章では、名前が `rv32ui-p-` から始まる RV32I 向けのテストを利用します。

例えば、ADD 命令のテストである `test/share/riscv-tests/isa/rv32ui-p-add.dump` を読んでみます（リスト 5.7）。`rv32ui-p-add.dump` は、`rv32ui-p-add` のダンプファイルです。

▼ リスト 5.7: `rv32ui-p-add.dump`

```
Disassembly of section .text.init:
00000000 <_start>:
 0: 0500006f          j      50 <reset_vector>

00000004 <trap_vector>:
 4: 34202f73          csrr   t5,mcause ← t5 = mcause
 ...
18: 00b00f93          li     t6,11
1c: 03ff0063          beq   t5,t6,3c <write_tohost>
 ...
0000003c <write_tohost>: ← 0x1000にテスト結果を書き込む
3c: 00001f17          auipc  t5,0x1
40: fc3f2223          sw     gp,-60(t5) # 1000 <tohost>
 ...
00000050 <reset_vector>:
50: 00000093          li     ra,0
 ...
c8: 00000f93          li     t6,0
 ...
130: 00000297          auipc t0,0x0
134: ed428293          addi   t0,t0,-300 # 4 <trap_vector>
138: 30529073          csrw   mtvec,t0 ← mtvecにtrap_vectorのアドレスを書き込む
 ...
178: 00000297          auipc t0,0x0
17c: 01428293          addi   t0,t0,20 # 18c <test_2>
180: 34129073          csrw   mepc,t0 ← mepcにtest_2のアドレスを書き込む
 ...
188: 30200073          mret   ← mepcのアドレス=test_2にジャンプする

0000018c <test_2>: ← 0 + 0 = 0 のテスト
18c: 00200193          li     gp,2 ← gp = 2
190: 00000593          li     a1,0
194: 00000613          li     a2,0
```

```

198: 00c58733          add    a4,a1,a2
19c: 00000393          li     t2,0
1a0: 4c771663          bne   a4,t2,66c <fail>
...
0000066c <fail>: ←失敗したときのジャンプ先
...
674: 00119193          sll    gp, gp, 0x1 ← gpを1ビット左シフトする
678: 0011e193          or    gp, gp, 1 ← gpのLSBを1にする
...
684: 00000073          ecall
...
00000688 <pass>: ←すべてのテストに成功したときのジャンプ先
...
68c: 00100193          li     gp,1 ← gp = 1
690: 05d00893          li     a7,93
694: 00000513          li     a0,0
698: 00000073          ecall
69c: c0001073          unimp

```

命令のテストは次の流れで実行されます。

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` にジャンプする。最後まで正常に実行できたら `pass` にジャンプする。
4. `fail`、`pass` : テストの成否をレジスタに書き込み、`trap_vector` にジャンプする。
5. `trap_vector` : `write_tohost` にジャンプする。
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。

`_start` から実行を開始し、最終的に `write_tohost` に移動します。テスト結果はメモリの `.tohost` に書き込まれます。`.tohost` のアドレスは、リンクの設定ファイルに記述されています(リスト 5.8)。プログラムのサイズは `0x1000` よりも小さいため、`.tohost` のアドレスは `0x1000` になります。

▼ リスト 5.8: riscv-tests/env/p/link.ld

```

OUTPUT_ARCH( "riscv" )
ENTRY(_start)

SECTIONS
{
    . = 0x00000000;
    .text.init : { *(.text.init) }
    . = ALIGN(0x1000);
    .tohost : { *(.tohost) }
}

```

5.4 テストの終了検知

テストを実行するとき、テストの終了を検知して、成功か失敗かを報告する必要があります。

riscv-tests はテストの終了を示すために、`.tohost` に LSB が `1` な値を書き込みます。書き込まれた値が `32'h1` のとき、テストが正常に終了したことを表しています。それ以外のときは、テストが失敗したことを表しています。

riscv-tests が終了したことを検知する処理を `top` モジュールに記述します。`top` モジュールでメモリへのアクセスを監視し、`.tohost` に LSB が `1` な値が書き込まれたら、`test_success` に結果を書き込んでテストを終了します。(リスト 5.9)。

▼ リスト 5.9: メモリアクセスを監視して終了を検知する (top.veryl)

```
// riscv-testsの終了を検知する
#[ifdef(TEST_MODE)]
always_ff {
    let RISCVTESTS_TOHOST_ADDR: Addr = 'h1000 as Addr;
    if d_membus.valid && d_membus.ready && d_membus.wen == 1 && d_membus.addr == RISCVTESTS_TOHOST_ADDR && d_membus.wdata[lsb] == 1'b1 {
        test_success = d_membus.wdata == 1;
        if d_membus.wdata == 1 {
            $display("riscv-tests success!");
        } else {
            $display("riscv-tests failed!");
            $error ("wdata : %h", d_membus.wdata);
        }
        $finish();
    }
}
```

`test_success` はポートとして定義します(リスト 5.10)。

▼ リスト 5.10: テスト結果を報告するためのポートを宣言する (top.veryl)

```
module top (
    clk: input clock,
    rst: input reset,
    #[ifdef(TEST_MODE)]
    test_success: output bit,
)
```

アトリビュートによって、終了検知のコードと `test_success` ポートは `TEST_MODE` マクロが定義されているときにのみ存在するようになっています。

5.5 テストの実行

試しに ADD 命令のテストを実行してみましょう。ADD 命令のテストの HEX ファイル

は `test/share/riscv-tests/isa/rv32ui-p-add.bin.hex` です。

TEST_MODE マクロを定義してシミュレータをビルドし、正常に動くことを確認します（リスト 5.11）。

▼ リスト 5.11: ADD 命令の riscv-tests を実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE" ← TEST_MODE マクロを定義してビルドする
$ ./obj_dir/sim test/share/riscv-tests/isa/rv32ui-p-add.bin.hex 0
# 4
00000000 : 0500006f
# 8
00000050 : 00000093
...
# 593
00000040 : fc3f2223
  itype    : 000100
  imm     : ffffffc4
  rs1[30]  : 0000103c
  rs2[ 3]  : 00000001
  op1     : 0000103c
  op2     : ffffffc4
  alu res  : 00001000
  mem stall : 1
  mem rdata : ff1ff06f
riscv-tests success!
- ~/core/src/top.sv:26: Verilog $finish
```

`riscv-tests success!` と表示され、テストが正常終了しました*4。

5.6 複数のテストの自動実行

ADD 命令以外の命令もテストしたいですが、わざわざコマンドを手打ちしたくありません。自動でテストを実行して、その結果を報告するプログラムを作成しましょう。

`test/test.py` を作成し、次のように記述します（リスト 5.12）。

▼ リスト 5.12: test.py

```
import argparse
import os
import subprocess

parser = argparse.ArgumentParser()
parser.add_argument("sim_path", help="path to simulator")
parser.add_argument("dir", help="directory includes test")
```

*4 実行が終了しない場合はどこかしらにバグがあります。`rv32ui-p-add.dump` と実行ログを見比べて、頑張って原因を探してください

```
parser.add_argument("files", nargs='*', help="test hex file names")
parser.add_argument("-r", "--recursive", action='store_true', help="search file recursively")
parser.add_argument("-e", "--extension", default="hex", help="test file extension")
parser.add_argument("-o", "--output_dir", default="results", help="result output directory")
parser.add_argument("-t", "--time_limit", type=float, default=10, help="limit of execution time. >
>set 0 to nolimit")
args = parser.parse_args()

# run test
def test(file_name):
    result_file_path = os.path.join(args.output_dir, file_name.replace(os.sep, "_") + ".txt")
    cmd = args.sim_path + " " + file_name + " 0"
    success = False
    with open(result_file_path, "w") as f:
        no = f.fileno()
        p = subprocess.Popen("exec " + cmd, shell=True, stdout=no, stderr=no)
        try:
            p.wait(None if args.time_limit == 0 else args.time_limit)
            success = p.returncode == 0
        except:
            pass
        finally:
            p.terminate()
            p.kill()
    print(("PASS" if success else "FAIL") + " : " + file_name)
    return (file_name, success)

# search files
def dir_walk(dir):
    for entry in os.scandir(dir):
        if entry.is_dir():
            if args.recursive:
                for e in dir_walk(entry.path):
                    yield e
            continue
        if entry.is_file():
            if not entry.name.endswith(args.extension):
                continue
            if len(args.files) == 0:
                yield entry.path
            for f in args.files:
                if entry.name.find(f) != -1:
                    yield entry.path
                    break

if __name__ == '__main__':
    os.makedirs(args.output_dir, exist_ok=True)

    res_strs = []
    res_statuses = []

    for hexpath in dir_walk(args.dir):
        f, s = test(os.path.abspath(hexpath))
        res_strs.append(("PASS" if s else "FAIL") + " : " + f)
```

```

res_statuses.append(s)

res_strs = sorted(res_strs)
statusText = "Test Result : " + str(sum(res_statuses)) + " / " + str(len(res_statuses))

with open(os.path.join(args.output_dir, "result.txt"), "w", encoding='utf-8') as f:
    f.write(statusText + "\n")
    f.write("\n".join(res_strs))

print(statusText)

if sum(res_statuses) != len(res_statuses):
    exit(1)

```

この Python プログラムは、第 2 引数で指定したディレクトリに存在する、第 3 引数で指定した文字列を名前に含むファイルを、第 1 引数で指定したシミュレータで実行し、その結果を報告します。

次のオプションの引数が存在します。

-r

第 2 引数で指定されたディレクトリの中にあるディレクトリも走査します。デフォルトでは走査しません。

-e 拡張子

指定した拡張子のファイルのみを対象にテストします。HEX ファイルをテストしたい場合は、`-e hex` にします。デフォルトでは `hex` が指定されています。

-o ディレクトリ

指定したディレクトリにテスト結果を格納します。デフォルトでは `result` ディレクトリに格納します。

-t 時間

テストに時間制限を設けます。0 を指定すると時間制限はなくなります。デフォルト値は 10(秒) です。

テストが成功したか失敗したかの判定には、シミュレータの終了コードを利用しています。テストが失敗したときに終了コードが 1 になるように、Verilator に渡している C++ プログラムを変更します (リスト 5.13)。

▼ リスト 5.13: tb_verilator.cpp

```

#ifndef TEST_MODE
    return dut->test_success != 1;
#endif

```

それでは、RV32I のテストを実行しましょう。riscv-tests の RV32I 向けのテストの接頭辞である `rv32ui-p-` を引数に指定します (リスト 5.14)。

▼ リスト 5.14: rv32ui-p から始まるテストを実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv32ui-p-
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-lh.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sb.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sltiu.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sh.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-bltu.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-or.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sra.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-xor.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-addi.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-srai.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-srl.i.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-auipc.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-slli.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-slti.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-lb.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-lw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-bge.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sub.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-xori.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-beq.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-fence_i.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-jal.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-and.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-lui.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-bgeu.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-slt.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sll.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-jalr.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-add.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-simple.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-andi.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv32ui-p-ma_data.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-lhu.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-lbu.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sltu.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-ori.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-blt.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-bne.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-srl.bin.hex
Test Result : 39 / 40
```

rv32ui-p- から始まる 40 個のテストの内、39 個のテストに成功しました。テストの詳細な結果は results ディレクトリに格納されています。

rv32ui-p-ma_data は、ロードストアするサイズに整列されていないアドレスへのロードストア命令のテストです。これは後の章で例外として対処するため、今は無視します。

第 6 章

RV64I の実装

これまでに、RISC-V の 32 ビットの基本整数命令セットである RV32I の CPU を実装しました。RISC-V には 64 ビットの基本整数命令セットとして RV64I が定義されています。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？まず、RV64I では XLEN が 32 ビットから 64 ビットに変更され、レジスタの幅や各種演算命令の演算の幅が 64 ビットになります。それに伴い、32 ビット幅での整数演算を行う命令、64 ビット幅でロードストアを行う命令が追加されます（表 6.1）。また、演算の幅が 64 ビットに広がるだけではなく、一部の命令の動作が少し変わります（表 6.2）。

▼表 6.1: RV64I で追加される命令

命令	動作
ADD[I]W	32 ビット単位で加算を行う。結果は符号拡張する
SUBW	32 ビット単位で減算を行う。結果は符号拡張する
SLL[I]W	レジスタの値を 0 ~ 31 ビット左論理シフトする。結果は符号拡張する
SRL[I]W	レジスタの値を 0 ~ 31 ビット右論理シフトする。結果は符号拡張する
SRA[I]W	レジスタの値を 0 ~ 31 ビット右算術シフトする。結果は符号拡張する
LWU	メモリから 32 ビット読み込む。結果はゼロで拡張する
LD	メモリから 64 ビット読み込む
SD	メモリに 64 ビット書き込む

▼表 6.2: RV64I で変更される命令

命令	変更後の動作
SLL[I]	0 ~ 63 ビット左論理シフトする
SRL[I]	0 ~ 63 ビット右論理シフトする
SRA[I]	0 ~ 63 ビット右算術シフトする
LUI	32 ビットの即値を生成する。結果は符号拡張する
AUIPC	32 ビットの即値を符号拡張したものに pc を足し合わせる
LW	メモリから 32 ビット読み込む。結果は符号拡張する

実装のテストには riscv-tests を利用します。RV64I 向けのテストは `rv64ui-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ます。

6.1 XLEN の変更

レジスタの幅が 32 ビットから 64 ビットに変わることを、XLEN が 32 から 64 に変わることです。eei パッケージに定義している `XLEN` を 64 に変更します（リスト 6.1）。RV64I にあっても命令の幅（ILEN）は 32 ビットのままでです。

▼ リスト 6.1: XLEN を変更する (eei.veryl)

```
const XLEN: u32 = 64;
```

6.1.1 SLL[I]、SRL[I]、SRA[I] 命令を変更する

RV32I では、シフト命令は `rs1` の値を 0 ~ 31 ビットシフトする命令として定義されています。これが RV64I では、`rs1` の値を 0 ~ 63 ビットシフトする命令に変更されます。

これに対応するために、ALU のシフト演算する量を 5 ビットから 6 ビットに変更します（リスト 6.2）。I 形式の命令（SLLI、SRLI、SRAI）のときは即値の下位 6 ビット、R 形式の命令（SLL、SRL、SRA）のときはレジスタの下位 6 ビットを利用します。

▼ リスト 6.2: シフト命令でシフトする量を変更する (alu.veryl)

```
let sll: UIntX = op1 << op2[5:0];
let srl: UIntX = op1 >> op2[5:0];
let sra: SIntX = $signed(op1) >>> op2[5:0];
```

6.1.2 LUI、AUIPC 命令を変更する

RV32I では、LUI 命令は 32 ビットの即値をそのままレジスタに格納する命令として定義されています。これが RV64I では、32 ビットの即値を 64 ビットに符号拡張した値を格納する命令に変更されます。AUIPC 命令も同様で、即値に PC を足す前に、即値を 64 ビットに符号拡張します。

この対応ですが、XLEN を 64 に変更した時点ですでに完了しています（リスト 6.3）。そのため、コードの変更の必要はありません。

▼ リスト 6.3: U 形式の即値は XLEN ビットに拡張されている (inst_decoder.veryl)

```
let imm_u: UIntX = {bits[31] repeat XLEN - $bits(imm_u_g) - 12, imm_u_g, 12'b0};
```

6.1.3 CSR を変更する

`MXLEN` (=XLEN) が 64 ビットに変更されると、CSR の幅も 64 ビットに変更されます。そのため、`mtvec`、`mepc`、`mcause` レジスタの幅を 64 ビットに変更する必要があります。

しかし、mtvec、mepc、mcause レジスタは XLEN ビットのレジスタ (`UIntX`) として定義しているため、変更の必要はありません。また、mtvec、mepc、mcause レジスタは MXLEN を基準に定義されており、RV32I から RV64I に変わってもフィールドに変化はないため、対応は必要ありません。

唯一、書き込みマスクの幅を広げる必要があります (リスト 6.4)。

▼ リスト 6.4: CSR の書き込みマスクの幅を広げる (`csrunit.veryl`)

```
const MTVEC_WMASK : UIntX = 'hffff_ffff_ffff_ffffc;
const MEPC_WMASK : UIntX = 'hffff_ffff_ffff_ffffc;
const MCAUSE_WMASK: UIntX = 'hffff_ffff_ffff_ffff;
```

6.1.4 LW 命令を変更する

LW 命令は 32 ビットの値をロードする命令です。RV64I では、LW 命令の結果が 64 ビットに符号拡張されるようになります。これに対応するため、memunit モジュールの `rdata` の割り当ての LW 部分を変更します (リスト 6.5)。

▼ リスト 6.5: LW 命令のメモリの読み込み結果を符号拡張する (`memunit.veryl`)

```
2'b10 : {D[31] repeat W - 32, D[31:0]},
```

また、XLEN が 64 に変更されたことで、幅を `MEM_DATA_WIDTH` (=32) として定義している `req_wdata` の代入文のビット幅が左右で合わなくなってしまっています。ビット幅を合わせるために、rs2 の下位 `MEM_DATA_WIDTH` ビットだけを切り取ります (リスト 6.6)。

▼ リスト 6.6: 左辺と右辺でビット幅を合わせる (`memunit.veryl`)

```
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[MEM_DATA_WIDTH - 1:0] << {addr[1:0], 3'b0};
```

6.1.5 riscv-tests でテストする

RV32I 向けのテストの実行

まず、RV32I 向けのテストが正しく動くことを確認します (リスト 6.7)。

▼ リスト 6.7: RV32I 向けのテストを実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv32ui-p-
...
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-srl.bin.hex
Test Result : 40 / 40
```

RV32I 向けのテストにすべて成功しました。しかし、`rv32ui-p-ma_data` は失敗するはずです（リスト 5.14）。これは、riscv-tests の RV32I 向けのテストは、XLEN が 64 のときはテストを実行せずに成功とするためです（リスト 6.8）。

▼リスト 6.8: `rv32ui-p-add` は XLEN が 64 のときにテストせずに成功する (`rv32ui-p-add.dump`)

```
00000050 <reset_vector>:
...
13c: 00100513      li      a0,1 ← a0 = 1
140: 01f51513      slli    a0,a0,0x1f ← a0を31ビット左シフト
144: 00054c63      bltz   a0,15c <reset_vector+0x10c> ← a0が0より小さかったらジンプ
>ジャンプ
148: 0ff0000f      fence
14c: 00100193      li      gp,1 ← gp=1（テスト成功）にする
150: 05d00893      li      a7,93
154: 00000513      li      a0,0
158: 00000073      ecall ← trap_vectorにジャンプして終了
```

riscv-tests は、a0 に 1 を代入した後、a0 を 31 ビット左シフトします。XLEN が 32 のとき、a0 の最上位ビット（符号ビット）が 1 になり、a0 は 0 より小さくなります。XLEN が 64 のとき、a0 の符号は変わらないため、a0 は 0 より大きくなります。これを利用して、XLEN が 32 ではないときは `trap_vector` にジャンプして、テスト成功として終了しています。

RV64I 向けのテストの実行

それでは、RV64I 向けのテストを実行します（リスト 6.9）。RV64I 向けのテストは名前が `rv64ui-p-` から始まります、

▼リスト 6.9: RV64I 向けのテストを実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-
...
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-add.bin.hex
...
Test Result : 14 / 52
```

ADD 命令のテストを含む、ほとんどのテストに失敗してしまいました。これは riscv-tests のテストが、まだ未実装の命令を含むためです（リスト 6.10）。

▼リスト 6.10: ADD 命令のテストは未実装の命令 (ADDIW 命令) を含む (`rv64ui-p-add.dump`)

```
0000000000000000208 <test_7>:
208: 00700193      li      gp,7
20c: 800005b7      lui    a1,0x80000
210: ffff8637      lui    a2,0xfffff8
214: 00c58733      add   a4,a1,a2
218: ffff03b7      lui    t2,0xfffff0
21c: fff3839b      addiw t2,t2,-1 # ffffffffffffff <_end+0xfffffffffffffedff>
>f>
220: 00f39393      slli   t2,t2,0xf
```

224: 46771063	bne a4, t2, 684 <fail>
---------------	------------------------

ということで、失敗していることを気にせずに実装を進めます。

6.2 ADD[I]W、SUBW 命令の実装

RV64I では、ADD 命令は 64 ビット単位で演算する命令になり、32 ビットの加算をする ADDW 命令と ADDIW 命令が追加されます。同様に、SUB 命令は 64 ビット単位の演算になり、32 ビットの減算をする SUBW 命令が追加されます。32 ビットの演算結果は符号拡張します。

6.2.1 ADD[I]W、SUBW 命令をデコードする

imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW

▲図 6.1: ADDW、ADDIW、SUBW 命令のフォーマット [6]

ADDW 命令と SUBW 命令は R 形式で、opcode は OP-32 (7'b0111011) です。ADDIW 命令は I 形式で、opcode は OP-IMM-32 (7'b0011011) です。

まず、eei パッケージに opcode の定数を定義します (リスト 6.11)。

▼リスト 6.11: opcode を定義する (eei.veryl)

```
const OP_OP_32    : logic<7> = 7'b0111011;
const OP_OP_IMM_32: logic<7> = 7'b0011011;
```

次に、InstCtrl 構造体に、32 ビット単位で演算を行う命令であることを示す is_op32 フラグを追加します (リスト 6.12)。

▼リスト 6.12: is_op32 を追加する (corectrl.veryl)

```
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_op32  : logic     , // OP-32またはOP-IMM-32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_csr   : logic     , // CSR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
}
```

inst_decoder モジュールの `InstCtrl` と即値を生成している部分を変更します（リスト 6.13、リスト 6.14）。これでデコードは完了です。

▼ リスト 6.13: OP-32、OP-IMM-32 の InstCtrl の生成 (inst_decoder.veryl)

```
is_op32を追加
ctrl = {case op {
    OP_LUI      : {InstType::U, T, T, F, F, F, F, F, F},
    OP_AUIPC    : {InstType::U, T, F, F, F, F, F, F, F},
    OP_JAL      : {InstType::J, T, F, F, F, T, F, F, F},
    OP_JALR     : {InstType::I, T, F, F, F, T, F, F, F},
    OP_BRANCH   : {InstType::B, F, F, F, F, F, F, F, F},
    OP_LOAD     : {InstType::I, T, F, F, F, F, T, F, F},
    OP_STORE    : {InstType::S, F, F, F, F, F, F, F, F},
    OP_OP      : {InstType::R, T, F, T, F, F, F, F, F},
    OP_OP_IMM   : {InstType::I, T, F, T, F, F, F, F, F},
    OP_OP_32    : {InstType::R, T, F, T, T, F, F, F, F},
    OP_OP_IMM_32: {InstType::I, T, F, T, T, F, F, F, F},
    OP_SYSTEM   : {InstType::I, T, F, F, F, F, F, T, F},
    default     : {InstType::X, F, F, F, F, F, F, F, F},
}, f3, f7};
```

▼ リスト 6.14: OP-IMM-32 の即値の生成 (inst_decoder.veryl)

```
imm = case op {
    OP_LUI, OP_AUIPC      : imm_u,
    OP_JAL                : imm_j,
    OP_JALR, OP_LOAD      : imm_i,
    OP_OP_IMM, OP_OP_IMM_32: imm_i,
    OP_BRANCH             : imm_b,
    OP_STORE              : imm_s,
    default               : 'x,
};
```

6.2.2 ALU に ADDW、SUBW を実装する

制御フラグを生成できたので、それに応じて 32 ビットの ADD と SUB を計算します。

まず、32 ビットの足し算と引き算の結果を生成します（リスト 6.15）。

▼ リスト 6.15: 32 ビットの足し算と引き算をする (alu.veryl)

```
let add32: UInt32 = op1[31:0] + op2[31:0];
let sub32: UInt32 = op1[31:0] - op2[31:0];
```

次に、フラグによって演算結果を選択する関数 `sel_w` を作成します（リスト 6.16）。この関数は、`is_op32` が 1 なら `value32` を 64 ビットに符号拡張した値、0 なら `value64` を返します。

▼ リスト 6.16: 演算結果を選択する関数を作成する (alu.veryl)

```
function sel_w (
    is_op32: input logic ,
    value32: input UInt32,
```

```

    value64: input UInt64,
) -> UInt64 {
    if is_op32 {
        return {value32[msb] repeat 32, value32};
    } else {
        return value64;
    }
}

```

sel_w 関数を使用し、alu モジュールの演算処理を変更します。case 文の足し算と引き算の部分を次のように変更します（リスト 6.17）。

▼リスト 6.17: 32 ビットの演算結果を選択する (alu.veryl)

```

3'b000: result = if ctrl.itype == InstType::I | ctrl.funct7 == 0 {
    sel_w(ctrl.is_op32, add32, add)
} else {
    sel_w(ctrl.is_op32, sub32, sub)
};

```

6.2.3 ADD[I]W、SUBW 命令をテストする

RV64I 向けのテストを実行して、結果ファイルを確認します（リスト 6.18、リスト 6.19）。

▼リスト 6.18: RV64I 向けのテストを実行する

```

$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-

```

▼リスト 6.19: テストの実行結果 (results/result.txt)

```

Test Result : 42 / 52
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ld.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-lwu.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ma_data.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-sd.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-slliw.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-sllw.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-sraiw.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-sraw.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-srliw.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-srlw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-add.bin.hex
...
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-addiw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-addw.bin.hex
...
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-subw.bin.hex
...

```

ADDIW、ADDW、SUBW だけでなく、未実装の命令以外のテストにも成功しました。

6.3 SLL[I]W、SRL[I]W、SRA[I]W 命令の実装

RV64I では、SLL[I]、SRL[I]、SRA[I] 命令は rs1 を 0 ~ 63 ビットシフトする命令になり、rs1 の下位 32 ビットを 0 ~ 31 ビットシフトする SLL[I]W、SRL[I]W、SRA[I]W 命令が追加されます。32 ビットの演算結果は符号拡張します。

000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

▲図 6.2: SLL[I]W、SRL[I]W、SRA[I]W 命令のフォーマット [6]

SLL[I]W、SRL[I]W、SRA[I]W 命令のフォーマットは、RV32I の SLL[I]、SRL[I]、SRA[I] 命令の opcode を変えたものと同じです。SLLW、SRLW、SRAW 命令は R 形式で、opcode は OP-32 です。SLLIW、SRLIW、SRAIW 命令は I 形式で、opcode は OP-IMM-32 です。どちらの opcode の命令も、ADD[I]W 命令と SUBW 命令の実装時にデコードが完了しています。

alu モジュールで、32 ビットのシフト演算の結果を生成します (リスト 6.20)。

▼リスト 6.20: 32 ビットのシフト演算をする (alu.veryl)

```
let sll32: UInt32 = op1[31:0] << op2[4:0];
let srl32: UInt32 = op1[31:0] >> op2[4:0];
let sra32: SInt32 = $signed(op1[31:0]) >>> op2[4:0];
```

生成したシフト演算の結果を sel_w 関数で選択します。case 文のシフト演算の部分を次のように変更します (リスト 6.21)。

▼リスト 6.21: 32 ビットの演算結果を選択する (alu.veryl)

```
3'b001: result = sel_w(ctrl.is_op32, sll32, sll);
...
3'b101: result = if ctrl.funct7 == 0 {
    sel_w(ctrl.is_op32, srl32, srl)
} else {
    sel_w(ctrl.is_op32, sra32, sra)
};
```

6.3.1 SLL[I]W、SRL[I]W、SRA[I]W 命令をテストする

RV64I 向けのテストを実行し、結果ファイルを確認します (リスト 6.22、リスト 6.23)。

▼リスト 6.22: RV64I 向けのテストを実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-
```

▼リスト 6.23: テストの実行結果 (results/result.txt)

```
Test Result : 48 / 52
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ld.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-lwu.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ma_data.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-sd.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-add.bin.hex
...
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sll.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-slli.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-slliw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sllw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sra.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-srai.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sraiw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sraw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-srl.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-srli.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-srliw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-srlw.bin.hex
...
...
```

SLLW、SLLIW、SRLW、SR LIW、SRAW、SRAIW 命令のテストに成功していることを確認できます。

6.4 LWU命令の実装

LB、LH命令は、ロードした値を符号拡張した値をレジスタに格納します。これに対して、LBU、LHU命令は、ロードした値をゼロで拡張した値をレジスタに格納します。

同様に、LW命令は、ロードした値を符号拡張した値をレジスタに格納します。これに対して、RV64Iでは、ロードした32ビットの値をゼロで拡張した値をレジスタに格納するLWU命令が追加されます。

imm[11:0]	rs1	110	rd	0000011	LWU
-----------	-----	-----	----	---------	-----

▲図 6.3: LWU命令のフォーマット [6]

LWU命令はI形式で、opcodeはLOADです。ロードストア命令はfunct3によって区別でき、LWU命令のfunct3は3'b110です。デコード処理に変更は必要なく、メモリにアクセスする処理を変更する必要があります。

memunit モジュールの、ロードする部分を変更します。32 ビットを `rdata` に割り当てるとき、`sext` によって符号かゼロで拡張するかを選択します (リスト 6.24)。

▼ リスト 6.24: LWU 命令の実装 (memunit.veryl)

```
2'b10 : {sext & D[31] repeat W - 32, D[31:0]},
```

6.4.1 LWU 命令をテストする

LWU 命令のテストを実行します (リスト 6.25)。

▼ リスト 6.25: LWU 命令をテストする

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-lwu
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-lwu.bin.hex
Test Result : 1 / 1
```

6.5 LD、SD 命令の実装

RV64I には、64 ビット単位でロードストアを行う LD 命令と SD 命令が定義されています。

imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD

▲ 図 6.4: LD、SD 命令のフォーマット

LD 命令は I 形式で、opcode は `LOAD` です。SD 命令は S 形式で、opcode は `STORE` です。どちらの命令も `funct3` は `3'b011` です。デコード処理に変更は必要ありません。

6.5.1 メモリの幅を広げる

現在のメモリの 1 つのデータの幅 (`MEM_DATA_WIDTH`) は 32 ビットですが、このままだと 64 ビットでロードやストアを行うときに、最低 2 回のメモリアクセスが必要です。これを 1 回のメモリアクセスで済ませるために、データの幅を 32 ビットから 64 ビットに広げます (リスト 6.26)。

▼ リスト 6.26: `MEM_DATA_WIDTH` を 64 ビットに変更する (eei.veryl)

```
const MEM_DATA_WIDTH: u32 = 64;
```

6.5.2 命令フェッチ処理を修正する

`XLEN`、`MEM_DATA_WIDTH` が変わっても、命令の長さ (`ILEN`) は 32 ビットのままです。そのため

め、top モジュールの `i_membus.rdata` の幅は 32 ビットなのに対し、`membus.rdata` は 64 ビットになり、ビット幅が一致しません。

ビット幅を合わせて正しく命令をフェッチするために、64 ビットの読み出しデータの上位 32 ビット、下位 32 ビットをアドレスの下位ビットで選択します。アドレスが 8 の倍数のときは下位 32 ビット、それ以外のときは上位 32 ビットを選択します。

まず、命令フェッチの要求アドレスをレジスタに格納します（リスト 6.27、リスト 6.28）。

▼ リスト 6.27: アドレスを格納するためのレジスタの定義 (top.veryl)

```
var memarb_last_i : logic;
var memarb_last_iaddr: Addr ;
```

▼ リスト 6.28: レジスタに命令フェッチの要求アドレスを格納する (top.veryl)

```
// メモリアクセスを調停する
always_ff {
    if_reset {
        memarb_last_i      = 0;
        memarb_last_iaddr = 0;
    } else {
        if membus.ready {
            memarb_last_i      = !d_membus.valid;
            memarb_last_iaddr = i_membus.addr;
        }
    }
}
```

このレジスタの値を利用し、`i_membus.rdata` に割り当てる値を選択します（リスト 6.29）。

▼ リスト 6.29: アドレスによってデータを選択する (top.veryl)

```
i_membus.rdata = if memarb_last_iaddr[2] == 0 {
    membus.rdata[31:0]
} else {
    membus.rdata[63:32]
};
```

6.5.3 SD 命令を実装する

SD 命令の実装のためには、書き込むデータ（`wdata`）と書き込みマスク（`wmask`）を変更する必要があります。

書き込むデータはアドレスの下位 2 ビットではなく下位 3 ビット分シフトします（リスト 6.30）。

▼ リスト 6.30: 書き込むデータの変更 (memunit.veryl)

```
req_wdata = rs2 << {addr[2:0], 3'b0};
```

書き込みマスクは 4 ビットから 8 ビットに拡張されるため、アドレスの下位 2 ビットではなく下位 3 ビットで選択します（リスト 6.31）。

▼ リスト 6.31: 書き込みマスクの変更 (memunit.veryl)

```

req_wmask = case ctrl.funct3[1:0] {
    2'b00 : 8'b1 << addr[2:0],
    2'b01 : case addr[2:0] {
        6      : 8'b11000000,
        4      : 8'b00011000,
        2      : 8'b00001100,
        0      : 8'b00000011,
        default: 'x,
    },
    2'b10 : case addr[2:0] {
        0      : 8'b00001111,
        4      : 8'b11110000,
        default: 'x,
    },
    2'b11 : 8'b11111111,
    default: 'x,
};
```

6.5.4 LD 命令を実装する

メモリのデータ幅が 64 ビットに広がるため、`rdata` に割り当てる値を、アドレスの下位 2 ビットではなく下位 3 ビットで選択します (リスト 6.32)。

▼ リスト 6.32: rdata の変更 (memunit.veryl)

```

rdata = case ctrl.funct3[1:0] {
    2'b00 : case addr[2:0] {
        0      : {sext & D[7] repeat W - 8, D[7:0]},
        1      : {sext & D[15] repeat W - 8, D[15:8]},
        2      : {sext & D[23] repeat W - 8, D[23:16]},
        3      : {sext & D[31] repeat W - 8, D[31:24]},
        4      : {sext & D[39] repeat W - 8, D[39:32]},
        5      : {sext & D[47] repeat W - 8, D[47:40]},
        6      : {sext & D[55] repeat W - 8, D[55:48]},
        7      : {sext & D[63] repeat W - 8, D[63:56]},
        default: 'x,
    },
    2'b01 : case addr[2:0] {
        0      : {sext & D[15] repeat W - 16, D[15:0]},
        2      : {sext & D[31] repeat W - 16, D[31:16]},
        4      : {sext & D[47] repeat W - 16, D[47:32]},
        6      : {sext & D[63] repeat W - 16, D[63:48]},
        default: 'x,
    },
    2'b10 : case addr[2:0] {
        0      : {sext & D[31] repeat W - 32, D[31:0]},
        4      : {sext & D[63] repeat W - 32, D[63:32]},
        default: 'x,
    },
    2'b11 : D,
    default: 'x,
};
```

```
};
```

6.5.5 LD、SD命令をテストする

LD、SD命令のテストを実行する前に、メモリのデータ単位が4バイトから8バイトになったため、テストのHEXファイルを4バイト単位の改行から8バイト単位の改行に変更します（リスト6.33）。

▼リスト6.33: HEXファイルを8バイト単位に変更する

```
$ cd test
$ find share/ -type f -name "*.bin" -exec sh -c "python3 bin2hex.py 8 {} > {}.hex" \;
```

riscv-testsを実行します（リスト6.34）。

▼リスト6.34: RV32I、RV64Iをテストする

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share/rv32ui-p-
...
Test Result : 40 / 40
$ python3 test/test.py -r obj_dir/sim test/share/rv64ui-p-
...
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ma_data.bin.hex
...
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-ld.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sd.bin.hex
...
Test Result : 51 / 52
```

RV64IのCPUを実装できました。

第 7 章

CPU のパイプライン化

これまでの章では、同時に 1 つの命令を実行する CPU を実装しました。高機能な CPU を実装するのは面白いですが、プログラムの実行が遅くてはいけません。機能を増やす前に、一度性能のことを考えてみましょう。

7.1 CPU の速度

CPU の性能指標は、例えば消費電力や実行速度が考えられます。本章では、プログラムの実行速度を考えます。

7.1.1 CPU の性能を考える

性能の比較にはクロック周波数やコア数などが用いられますが、プログラムの実行速度を比較する場合、プログラムの実行にかかる時間のみが絶対的な指標になります。プログラムの実行時間は、次のような式で表せます(図 7.1)

$$CPU \text{ 時間} = \frac{\text{実行命令数} \times CPI}{\text{クロック周波数}}$$

▲図 7.1: CPU 性能方程式 [12]

それぞれの用語の定義は次の通りです。

CPU 時間 (CPU time)

プログラムの実行のために CPU が費やした時間

実行命令数

プログラムの実行で実行される命令数

CPI (Clock cycles Per Instruction)

プログラム全体またはプログラムの一部分の命令を実行した時の 1 命令当たりの平均クロック

ク・サイクル数

クロック周波数 (clock rate)

クロック・サイクル時間 (clock cycle time) の逆数

クロック・サイクル時間は、クロックが $0 \rightarrow 1 \rightarrow 1$ になる周期のこと

今のところ、CPU は命令をスキップしたり無駄に実行することはありません。そのため、実行命令数は、プログラムを 1 命令ずつ順に実行していった時の実行命令数になります。

CPI を計測するためには、何の命令にどれだけのクロック・サイクル数がかかるかと、それぞれの命令の割合が必要です。今のところ、メモリにアクセスする命令は 3 ~ 4 クロック、それ以外の命令は 1 クロックで実行されます。命令の割合は考えないでおきます。

クロック周波数は、CPU の回路のクリティカルパスの長さによって決まります。クリティカルパスとは、組み合わせ回路の中で最も大きな遅延を持つ経路のことです。

7.1.2 実行速度を上げる方法を考える

CPU 性能方程式の各項に注目すると、CPU 時間を減らすためには、実行命令数を減らすか、CPI を減らすか、クロック周波数を増大させる必要があります。

実行命令数に注目する

実行命令数を減らすためには、コンパイラによる最適化でプログラムの命令数を減らすソフトウェア的な方法と、命令セットアーキテクチャ (ISA) を変更することで必要な命令数を減らす方法が存在します。どちらも本書の目的とするところではないので、検討しません^{*1}。

CPI に注目する

CPI を減らすためには、例えどの命令も 1 クロックで実行してしまうという方法が考えられます。しかし、そのために論理回路を大きくすると、その分クリティカルパスが長くなってしまう場合があります。また、1 クロックに 1 命令しか実行しない場合、どう頑張っても CPI は 1 より小さくなりません。

CPI をより効果的に減らすためには、1 クロックで 1 つ以上の命令を実行開始し、1 つ以上の命令を実行完了すればいいです。これを実現する手法として、スーパースカラやアウトオブオーダー実行が存在します。これらの手法はずっと後の章で解説、実装します。

クロック周波数に注目する

クロック周波数を増大させるには、クリティカルパスの長さを短くする必要があります。

今のところ、CPU は計算命令を 1 クロック (シングルサイクル) で実行します。例えば ADD 命令を実行するとき、FIFO に保存された ADD 命令をデコードし、命令のビット列をもとにレジスタの値を選択し、ALU で足し算を実行し、その結果をレジスタにライトバックします。これらを 1 クロックで実行するということは、命令が保存されている 32 ビットのレジスタと 32*64 ビット

^{*1} 他の方法として、関数呼び出しやループを CPU 側で検知して結果を保存して利用することで実行命令数を減らす手法があります。この手法はずっと後の章で検討します。

のレジスタファイルを入力に、64 ビットの ADD 演算の結果を出力する組み合わせ回路が存在するということです。この回路は大変に段数の深い組み合わせ回路を必要とし、長いクリティカルパスを生成する原因になります。

クロック周波数を増大させるもっとも単純な方法は、命令の処理をいくつかのステージ(段)に分割し、複数クロックで1つの命令を実行することです。複数のクロック・サイクルで命令を実行することから、この形式の CPU はマルチサイクル CPU と呼びます。

\時間(t)	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6
命令1	ステージ1	ステージ2	ステージ3			
命令2				ステージ1	ステージ2	ステージ3

▲図 7.2: 命令の実行 (マルチサイクル)

命令の処理をいくつかのステージに分割すると、それに合わせて回路の深さが軽減され、クロック周波数を増大させられます。

図 7.2 では、1つの命令を3クロック(ステージ)で実行しています。3クロックもかかるのであれば、CPI が3倍になり、CPU 時間が増えてしまいそうです。しかし、処理を均等な3ステージに分割できた場合、クロック周波数は3分の1になる^{*2}ため、それほど CPU 時間は増えません。

しかし、CPI がステージ分だけ増大してしまうのは問題です。この問題は、命令の処理を、まるで車の組立のように流れ作業で行うことで緩和できます(図 7.3)。このような処理のことを、パイプライン処理と呼びます。

\時間(t)	t = 1	t = 2	t = 3	t = 4	t = 5
命令1	ステージ1	ステージ2	ステージ3		
命令2		ステージ1	ステージ2	ステージ3	
命令3			ステージ1	ステージ2	ステージ3

▲図 7.3: 命令の実行 (パイプライン処理)

本章では、CPU をパイプライン化することで性能の向上を図ります。

7.1.3 パイプライン処理のステージを考える

具体的に処理をどのようなステージに分割してパイプライン処理を実現すればいいでしょうか?これをるために、第3章の最初で検討した CPU の動作を振り返ります。第3章では、CPU の動作を次のように順序付けしました。

^{*2} 実際のところは均等に分割することはできないため、Nステージに分割してもクロック周波数はN分の1になります

1. PC に格納されたアドレスにある命令をフェッチする
2. 命令を取得したらデコードする
3. 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)
4. 計算する命令の場合、計算を行う
5. メモリにアクセスする命令の場合、メモリ操作を行う
6. 計算やメモリアクセスの結果をレジスタに格納する
7. PC の値を次に実行する命令のアドレスに設定する

もう少し大きな処理単位に分割しなおすと、次の 5 つの処理 (ステージ) を構成できます。ステージ名の後ろに、それぞれ対応する上のリストの処理の番号を記載しています。

IF (Instruction Fetch) ステージ (1)

メモリから命令をフェッチします。

フェッチした命令を ID ステージに受け渡します。

ID (Instruction Decode) ステージ (2、3)

命令をデコードし、制御フラグと即値を生成します。

生成したデータを EX ステージに渡します。

EX (EXecute) ステージ (3、4)

制御フラグ、即値、レジスタの値を利用し、ALU で計算します。

分岐判定やジャンプ先の計算も行い、生成したデータを MEM ステージに渡します。

MEM (MEMory) ステージ (5、7)

メモリにアクセスする命令と CSR 命令を処理します。

分岐命令かつ分岐が成立する、ジャンプ命令である、またはトラップが発生するとき、IF、ID、EX ステージにある命令を無効化して、ジャンプ先を IF ステージに伝えます。メモリのロード、CSR の読み込み結果を WB ステージに渡します。

WB (WriteBack) ステージ (6)

ALU の演算結果、メモリや CSR の読み込み結果など、命令の処理結果をレジスタに書き込みます。

MEM ステージではジャンプするときに IF、ID、EX ステージにある命令を無効化します。これは、IF、ID、EX ステージにある命令は、ジャンプによって実行されない命令になるためです。パイプラインのステージにある命令を無効化することを、パイプラインをフラッシュ (flush) すると呼びます。

IF、ID、EX、MEM、WB の 5 段の構成を、**5 段パイプライン** (Five Stage Pipeline) と呼ぶことがあります。

CSR を MEM ステージで処理する

上記の 5 段のパイプライン処理では、CSR の処理を MEM ステージで行っています。これはいったいなぜでしょうか?

CPU には ECALL 命令による例外しか実装してしないため、EX ステージで CSR の処理を行ってしまっても問題ありません。しかし、他の例外、例えばメモリアクセスに伴う例外を実装するとき、問題が生じます。

メモリアクセスに起因する例外が発生するのは MEM ステージです。このとき、EX ステージで CSR の処理を行っていて、EX ステージに存在する命令が mtvec レジスタに書き込む CSRRW 命令だった場合、本来は MEM ステージで発生した例外によって実行されないはずである CSRRW 命令によって、既に mtvec レジスタが書き換えられているかもしれません。これを復元する処理を書くことはできますが、MEM ステージ以降で CSR を処理することでもこの事態を回避できるため、MEM ステージで CSR を処理しています。

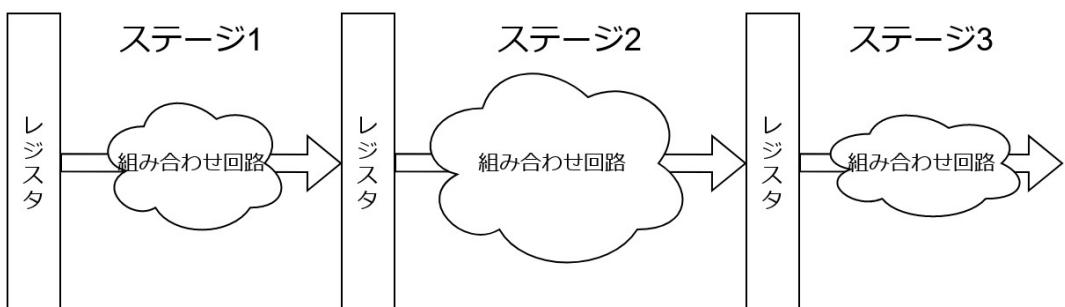
.....

7.2 パイプライン処理の実装

7.2.1 ステージに分割する準備をする

それでは、CPU をパイプライン化します。

パイプライン処理では、複数のステージがそれぞれ違う命令を処理します。そのため、それぞれのステージのために、現在処理している命令を保持するためのレジスタ（パイプラインレジスタ）を用意します。



▲図 7.4: パイプライン処理の概略図

まず、処理を複数ステージに分割する前に、既存の変数の名前を変更します。

core モジュールでは、命令をフェッチする処理に使う変数の名前の先頭に `if_` 、 FIFO から取り出した命令の情報を表す変数の名前の先頭に `inst_` をつけています。

命令をフェッチする処理は IF ステージに該当するため、`if_` から始まる変数はこれまで問題ありません。しかし、`inst_` から始まる変数は、CPU の処理を複数ステージに分けたとき、どのステージの変数か分からなくなります。IF ステージの次は ID ステージであるため、変数が ID ステージのものであることを示す名前に変えてしまいます。

▼リスト 7.1: 変数名を変更する (core.veryl)

```

let ids_valid      : logic      = if_fifo_rvalid;
var ids_is_new    : logic      ; // 命令が現在のクロックで供給されたかどうか
let ids_pc        : Addr       = if_fifo_rdata.addr;
let ids_inst_bits: Inst       = if_fifo_rdata.bits;
var ids_ctrl      : InstCtrl;
var ids_imm       : UIntX     ;

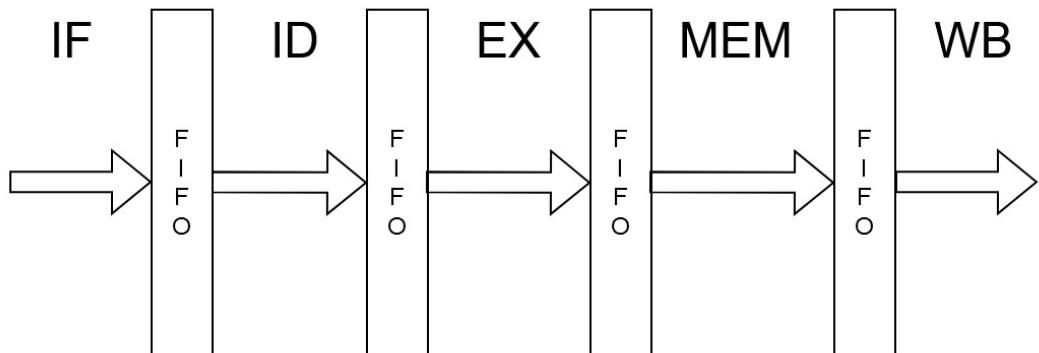
```

`inst_valid`、`inst_is_new`、`inst_pc`、`inst_bits`、`inst_ctrl`、`inst_imm` の名前をリスト 7.1 のように変更します。定義だけではなく、変数を使用しているところもすべて変更してください。

7.2.2 FIFO を作成する

命令フェッチ処理とそれ以降の処理は、それぞれ独立して動作しています。実は既に CPU は、IF と ID ステージ (命令フェッチ以外の処理を行うステージ) の 2 ステージのパイプライン処理を行っています。

IF ステージと ID ステージは FIFO で区切られており、FIFO のレジスタを経由して命令の受け渡しを行います。これと同様に、5 ステージのパイプライン処理の実装では、それぞれのステージを FIFO で接続します (図 7.5)。ただし、FIFO のサイズは 1 とします。この場合、FIFO はただの 1 つのレジスタです。

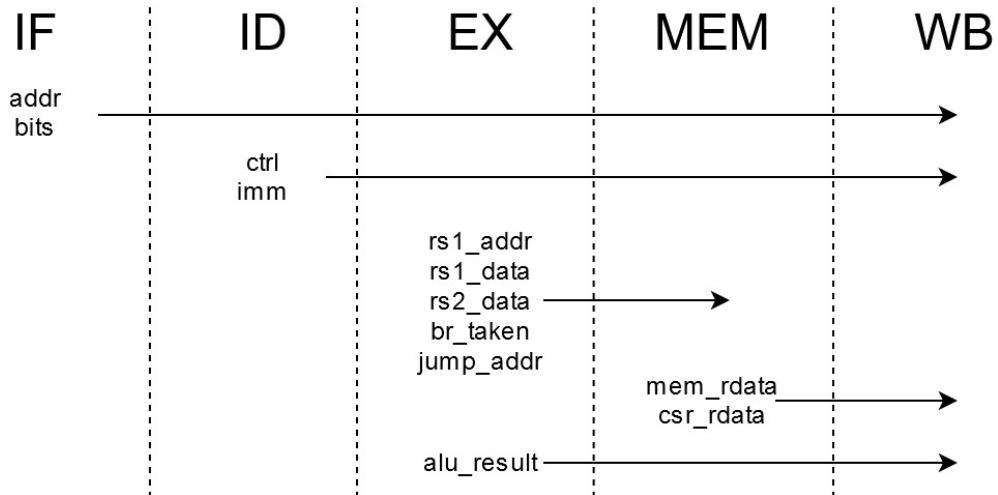


▲図 7.5: FIFO を利用したパイプライン処理

IF から ID への FIFO は存在するため、ID から EX、EX から MEM、MEM から WB への FIFO を作成します。

構造体の定義

まず、FIFO に格納するデータの型を定義します。それぞれのフィールドが存在する区間は図 7.6 の通りです。



▲図 7.6: 構造体のフィールドの生存区間

▼リスト 7.2: ID → EX の間の FIFO のデータ型 (core.veryl)

```
struct exq_type {
    addr: Addr      ,
    bits: Inst      ,
    ctrl: InstCtrl ,
    imm : UIntX    ,
}
```

ID ステージは、IF ステージから命令のアドレスと命令のビット列を受け取ります。命令のビット列をデコードして、制御フラグと即値を生成し、EX ステージに渡します（リスト 7.2）。

▼リスト 7.3: EX → MEM の間の FIFO のデータ型 (core.veryl)

```
struct memq_type {
    addr      : Addr      ,
    bits      : Inst      ,
    ctrl      : InstCtrl ,
    imm       : UIntX    ,
    alu_result: UIntX    ,
    rs1_addr : logic <5>,
    rs1_data : UIntX    ,
    rs2_data : UIntX    ,
    br_taken : logic    ,
    jump_addr: Addr      ,
}
```

EX ステージは、ID ステージで生成された制御フラグと即値を受け取ります。整数演算命令のとき、レジスタの値を使って計算します。分岐命令のとき、分岐判定を行います。CSR やメモリアクセスで rs1 と rs2 を利用するため、演算の結果とともに MEM ステージに渡します（リスト 7.3）。

▼リスト 7.4: MEM → WB の間の FIFO のデータ型 (core.veryl)

```
struct wbq_type {
    addr      : Addr      ,
    bits      : Inst      ,
    ctrl      : InstCtrl ,
    imm       : UIntX    ,
    alu_result: UIntX    ,
    mem_rdata : UIntX    ,
    csr_rdata : UIntX    ,
}
```

MEM ステージは、メモリのロード結果と CSR の読み込みデータを生成し、WB ステージに渡します（リスト 7.4）。

WB ステージでは、命令がライトバックする命令のとき、即値、ALU の計算結果、メモリのロード結果、CSR の読み込みデータから 1 つを選択し、レジスタに値を書き込みます。

構造体のフィールドの生存区間が図 7.6 のようになっている理由が分かったでしょうか？

FIFO のインスタンス化

FIFO と接続するための変数を定義し、FIFO をインスタンス化します（リスト 7.5、リスト 7.6）。

`DATA_TYPE` パラメータには先ほど作成した構造体を設定します。FIFO のデータ個数は 1 であるため、`WIDTH` パラメータには `1` を設定します³。`mem_wb_fifo` の `flush` は `0` にしています。

▼リスト 7.5: FIFO と接続するための変数を定義する (core.veryl)

```
// ID -> EXのFIFO
var exq_wready: logic      ;
var exq_wvalid: logic      ;
var exq_wdata : exq_type;
var exq_rready: logic      ;
var exq_rvalid: logic      ;
var exq_rdata : exq_type;

// EX -> MEMのFIFO
var memq_wready: logic      ;
var memq_wvalid: logic      ;
var memq_wdata : memq_type;
var memq_rready: logic      ;
var memq_rvalid: logic      ;
var memq_rdata : memq_type;

// MEM -> WBのFIFO
var wbq_wready: logic      ;
var wbq_wvalid: logic      ;
var wbq_wdata : wbq_type;
var wbq_rready: logic      ;
var wbq_rvalid: logic      ;
var wbq_rdata : wbq_type;
```

³ FIFO のデータ個数は `2 ** WIDTH - 1` です

▼リスト 7.6: FIFO のインスタンス化 (core.veryl)

```

inst id_ex_fifo: fifo #(
    DATA_TYPE: exq_type,
    WIDTH    : 1      ,
) (
    clk           ,
    rst           ,
    flush : control_hazard,
    wready: exq_wready  ,
    wvalid: exq_wvalid  ,
    wdata : exq_wdata   ,
    rready: exq_rready  ,
    rvalid: exq_rvalid  ,
    rdata : exq_rdata   ,
);

inst ex_mem_fifo: fifo #(
    DATA_TYPE: memq_type,
    WIDTH    : 1      ,
) (
    clk           ,
    rst           ,
    flush : control_hazard,
    wready: memq_wready  ,
    wvalid: memq_wvalid  ,
    wdata : memq_wdata   ,
    rready: memq_rready  ,
    rvalid: memq_rvalid  ,
    rdata : memq_rdata   ,
);

inst mem_wb_fifo: fifo #(
    DATA_TYPE: wbq_type,
    WIDTH    : 1      ,
) (
    clk           ,
    rst           ,
    flush : 0      ,
    wready: wbq_wready,
    wvalid: wbq_wvalid,
    wdata : wbq_wdata ,
    rready: wbq_rready,
    rvalid: wbq_rvalid,
    rdata : wbq_rdata ,
);

```

7.2.3 IF ステージを実装する

まず、IF ステージを実装します。… といっても、既に IF ステージ (=命令フェッチ処理) は独立に動くものとして実装されているため、手を加える必要はありません。

リスト 7.7 のようなコメントを挿入すると、ステージの処理を書いている区間が分かりやすくな

ります。ID、EX、MEM、WB ステージを実装するときにも同様のコメントを挿入し、ステージの処理のコードをまとまった場所に配置しましょう。

▼ リスト 7.7: IF ステージが始まることを示すコメントを挿入する (core.veryl)

```
////////////////////////////// IF Stage /////////////////////////////// /  
var if_pc : Addr ;  
...
```

7.2.4 ID ステージを実装する

ID ステージでは、命令をデコードします。既に `ids_ctrl` と `ids_imm` には、デコード結果の制御フラグと即値が割り当てられているため、既存のコードの変更は必要ありません。

デコード結果は EX ステージに渡します。EX ステージにデータを渡すには、`exq_wdata` にデータを割り当てます (リスト 7.8)。

▼ リスト 7.8: EX ステージに値を渡す (core.veryl)

```
always_comb {  
    // ID -> EX  
    if_fifo_rready = exq_wready;  
    exq_wvalid = if_fifo_rvalid;  
    exq_wdata.addr = if_fifo_rdata.addr;  
    exq_wdata.bits = if_fifo_rdata.bits;  
    exq_wdata.ctrl = ids_ctrl;  
    exq_wdata.imm = ids_imm;  
}
```

ID ステージにある命令は、EX ステージが命令を受け入れられるとき (`exq_wready`)、ID ステージを完了して EX ステージに処理を進められます。この仕組みは、`if_fifo_rready` に `exq_wready` を割り当てることで実現できます。

最後に、命令が現在のクロックで供給されたかどうかを示す変数 `id_is_new` は必要ないため削除します (リスト 7.9)。

▼ リスト 7.9: `ids_is_new` を削除する (core.veryl)

```
var ids_is_new : logic ;
```

7.2.5 EX ステージを実装する

EX ステージでは、整数演算命令のときは ALU で計算し、分岐命令のときは分岐判定を行います。

まず、EX ステージに存在する命令の情報を `exq_rdata` から取り出します (リスト 7.10)。

▼ リスト 7.10: 変数の定義 (core.veryl)

```
let exs_valid : logic = exq_rvalid;  
let exs_pc : Addr = exq_rdata.addr;
```

```

let exs_inst_bits: Inst      = exq_rdata.bits;
let exs_ctrl      : InstCtrl = exq_rdata.ctrl;
let exs_imm       : UIntX    = exq_rdata.imm;

```

次に、EX ステージで扱う変数の名前を変更します。変数の名前に `exs_` をつけます（リスト 7.11）。

▼リスト 7.11: 変数名を変更する (core.veryl)

```

// レジスタ番号
let exs_rs1_addr: logic<5> = exs_inst_bits[19:15];
let exs_rs2_addr: logic<5> = exs_inst_bits[24:20];

// ソースレジスタのデータ
let exs_rs1_data: UIntX = if exs_rs1_addr == 0 {
    0
} else {
    regfile[exs_rs1_addr]
};

let exs_rs2_data: UIntX = if exs_rs2_addr == 0 {
    0
} else {
    regfile[exs_rs2_addr]
};

// ALU
var exs_op1      : UIntX;
var exs_op2      : UIntX;
var exs_alu_result: UIntX;

always_comb {
    case exs_ctrl.iotype {
        InstType::R, InstType::B: {
            exs_op1 = exs_rs1_data;
            exs_op2 = exs_rs2_data;
        }
        InstType::I, InstType::S: {
            exs_op1 = exs_rs1_data;
            exs_op2 = exs_imm;
        }
        InstType::U, InstType::J: {
            exs_op1 = exs_pc;
            exs_op2 = exs_imm;
        }
        default: {
            exs_op1 = 'x;
            exs_op2 = 'x;
        }
    }
}

inst alum: alu (
    ctrl  : exs_ctrl      ,

```

```

op1  : exs_op1      ,
op2  : exs_op2      ,
result: exs_alu_result,
);

var exs_brunit_take: logic;

inst bru: brunit (
  funct3: exs_ctrl.funct3,
  op1  : exs_op1      ,
  op2  : exs_op2      ,
  take  : exs_brunit_take,
);

```

最後に、MEM ステージに命令とデータを渡します。MEM ステージにデータを渡すために、

`memq_wdata` にデータを割り当てます（リスト 7.12）。

▼ リスト 7.12: MEM ステージにデータを渡す (core.veryl)

```

always_comb {
  // EX -> MEM
  exq_rready      = memq_wready;
  memq_wvalid     = exq_rvalid;
  memq_wdata.addr = exq_rdata.addr;
  memq_wdata.bits = exq_rdata.bits;
  memq_wdata.ctrl = exq_rdata.ctrl;
  memq_wdata.imm  = exq_rdata.imm;
  memq_wdata.rs1_addr = exs_rs1_addr;
  memq_wdata.rs1_data = exs_rs1_data;
  memq_wdata.rs2_data = exs_rs2_data;
  memq_wdata.alu_result = exs_alu_result;
  ←ジャンプ命令、または、分岐命令かつ分岐が成立するとき、1にする
  memq_wdata.br_taken = exs_ctrl.is_jump || inst_is_br(exs_ctrl) && exs_brunit_take;
  memq_wdata.jump_addr = if inst_is_br(exs_ctrl) {
    exs_pc + exs_imm ←分岐命令の分岐先アドレス
  } else {
    exs_alu_result ←ジャンプ命令のジャンプ先アドレス
  };
}

```

`br_taken` には、ジャンプ命令かどうか、または分岐命令かつ分岐が成立するか、という条件を割り当てます。`jump_addr` には、分岐命令、またはジャンプ命令のジャンプ先アドレスを割り当てます。MEM ステージではこれを用いてジャンプと分岐を処理します。

EX ステージにある命令は、MEM ステージが命令を受け入れられるとき（`memq_wready`）、EX ステージを完了して MEM ステージに処理を進められます。この仕組みは、`exq_rready` に `memq_wready` を割り当てることで実現できます。

7.2.6 MEM ステージを実装する

MEM ステージでは、メモリにアクセスする命令と CSR 命令を処理します。また、ジャンプ命

令、分岐命令かつ分岐が成立、またはトランプが発生するとき、次に実行する命令のアドレスを変更します。

ロードストア命令でメモリにアクセスしているとき、EX ステージから MEM ステージに別の命令の処理を進めることはできず、パイプライン処理は止まってしまいます。パイプライン処理を進められない状態のことをパイプラインハザード (pipeline hazard) と呼びます。

まず、MEM ステージに存在する命令の情報を `memq_rdata` から取り出します (リスト 7.13)。MEM ステージでは、csrunit モジュールに、命令が現在のクロックで MEM ステージに供給されたかどうかの情報を渡します。そのため、変数 `mem_is_new` を定義しています。

▼ リスト 7.13: 変数の定義 (core.veryl)

```
var mems_is_new : logic      ;
let mems_valid  : logic      = memq_rvalid;
let mems_pc    : Addr        = memq_rdata.addr;
let mems_inst_bits: Inst    = memq_rdata.bits;
let mems_ctrl   : InstCtrl  = memq_rdata.ctrl;
let mems_rd_addr : logic <5> = mems_inst_bits[11:7];
```

`mem_is_new` には、`id_is_new` の更新に利用していたコードを利用します (リスト 7.14)。

▼ リスト 7.14: mem_is_new の更新 (core.veryl)

```
always_ff {
    if_reset {
        mems_is_new = 0;
    } else {
        if memq_rvalid {
            mems_is_new = memq_rready;
        } else {
            mems_is_new = 1;
        }
    }
}
```

次に、MEM モジュールで使う変数に合わせて、memunit モジュールと csrunit モジュールのポートに割り当てる変数名を変更します (リスト 7.15)。

▼ リスト 7.15: 変数名を変更する (core.veryl)

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

inst memu: memunit (
    clk           ,
    rst           ,
    valid : mems_valid      ,
    is_new: mems_is_new     ,
    ctrl  : mems_ctrl       ,
    addr  : memq_rdata.alu_result,
    rs2   : memq_rdata.rs2_data ,
```

```

rdata : memu_rdata      ,
stall : memu_stall      ,
membus: d_membus        ,
);

var csru_rdata      : UIntX;
var csru_raise_trap : logic;
var csru_trap_vector: Addr ;

inst csru: csrunit (
    clk          ,
    rst          ,
    valid       : mems_valid      ,
    pc          : mems_pc        ,
    ctrl        : mems_ctrl      ,
    rd_addr    : mems_rd_addr   ,
    csr_addr   : mems_inst_bits[31:20],
    rs1        : if mems_ctrl.funct3[2] == 1 && mems_ctrl.funct3[1:0] != 0 {
        {1'b0 repeat XLEN - $bits(memq_rdata.rs1_addr), memq_rdata.rs1_addr} // rs1を0で拡張する
    } else {
        memq_rdata.rs1_data
    },
    rdata      : csru_rdata,
    raise_trap : csru_raise_trap,
    trap_vector: csru_trap_vector,
);

```

フェッチ先が変わったことを表す変数 `control_hazard` と、新しいフェッチ先を示す信号 `control_hazard_pc_next` では、EX ステージで計算したデータと CSR ステージのトラップ情報を利用します（リスト 7.16）。

▼リスト 7.16: ジャンプの判定処理 (core.veryl)

```

assign control_hazard      = mems_valid && (csru_raise_trap || mems_ctrl.is_jump || memq_rdata.br_taken);
assign control_hazard_pc_next = if csru_raise_trap {
    csru_trap_vector
} else {
    memq_rdata.jump_addr
};

```

ジャンプ命令の後ろの余計な命令を実行しないために、`control_hazard` が 1 になったとき、ID、EX、MEM ステージに命令を供給する FIFO をフラッシュします。`control_hazard` が 1 になるとき、MEM ステージの処理は完了しています。後述しますが、WB ステージの処理は必ず 1 クロックで終了します。そのため、フラッシュするとき、MEM ステージにある命令は必ず WB ステージに移動します。

最後に、WB ステージに命令とデータを渡します（リスト 7.17）。WB ステージにデータを渡すために、`wbq_wdata` にデータを割り当てます

▼ リスト 7.17: WB ステージにデータを渡す (core.veryl)

```
always_comb {
    // MEM -> WB
    memq_rready      = wbq_wready && !memu_stall;
    wbq_wvalid       = memq_rvalid && !memu_stall;
    wbq_wdata.addr   = memq_rdata.addr;
    wbq_wdata.bits   = memq_rdata.bits;
    wbq_wdata.ctrl   = memq_rdata.ctrl;
    wbq_wdata.imm    = memq_rdata.imm;
    wbq_wdata.alu_result = memq_rdata.alu_result;
    wbq_wdata.mem_rdata = memu_rdata;
    wbq_wdata.csr_rdata = csru_rdata;
}
```

MEM ステージにある命令は、memunit モジュールが処理中ではなく (`!memu_stall`)、WB ステージが命令を受け入れられるとき (`wbq_wready`)、MEM ステージを完了して WB ステージに処理を進められます。この仕組みは、`memq_rready` と `wbq_wvalid` を確認してください。

7.2.7 WB ステージを実装する

WB ステージでは、命令の結果をレジスタにライトバックします。WB ステージが完了したら命令の処理は終わりなので、命令を破棄します。

まず、WB ステージに存在する命令の情報を `wbq_rdata` から取り出します (リスト 7.18)。

▼ リスト 7.18: 変数の定義 (core.veryl)

```
let wbs_valid    : logic    = wbq_rvalid;
let wbs_pc       : Addr     = wbq_rdata.addr;
let wbs_inst_bits: Inst     = wbq_rdata.bits;
let wbs_ctrl     : InstCtrl = wbq_rdata.ctrl;
let wbs_imm      : UIntX    = wbq_rdata.imm;
```

次に、WB ステージで扱う変数名を変更します。変数名に `wbs_` をつけます (リスト 7.19)。

▼ リスト 7.19: 変数名を変更する (core.veryl)

```
let wbs_rd_addr: logic<5> = wbs_inst_bits[11:7];
let wbs_wb_data: UIntX     = if wbs_ctrl.is_lui {
    wbs_imm
} else if wbs_ctrl.is_jump {
    wbs_pc + 4
} else if wbs_ctrl.is_load {
    wbq_rdata.mem_rdata
} else if wbs_ctrl.is_csr {
    wbq_rdata.csr_rdata
} else {
    wbq_rdata.alu_result
};

always_ff {
    if wbs_valid && wbs_ctrl.rwb_en {
```

```

        regfile[wbs_rd_addr] = wbs_wb_data;
    }
}

```

最後に、命令を FIFO から取り出します。WB ステージでは命令を複数クロックで処理することはなく、WB ステージの次のステージを待つ必要もありません。`wbq_rready` に 1 を割り当てることで、常に FIFO から命令を取り出します(リスト 7.20)。

▼リスト 7.20: 命令を FIFO から取り出す (core.veryl)

```

always_comb {
    // WB -> END
    wbq_rready = 1;
}

```

これで、IF、ID、EX、MEM、WB ステージを作成できました。

7.2.8 デバッグのために情報を表示する

今まででは同時に 1 つの命令しか処理していませんでしたが、これからは全てのステージで別の命令を処理することになります。デバッグ表示を変更しておきましょう。

リスト 7.21 のように、デバッグ表示の `always_ff` ブロックを変更します。

▼リスト 7.21: 各ステージの情報をデバッグ表示する (core.veryl)

```

//////////////////////////// DEBUG //////////////////////////////
var clock_count: u64;

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

        $display("");
        $display("# %d", clock_count);

        $display("IF -----");
        $display("    pc : %h", if_pc);
        $display(" is req : %b", if_is_requested);
        $display(" pc req : %h", if_pc_requested);
        $display("ID -----");
        if ids_valid {
            $display("    %h : %h", ids_pc, if_fifo_rdata.bits);
            $display("    itype : %b", ids_ctrl.itype);
            $display("    imm   : %h", ids_imm);
        }
        $display("EX -----");
        if exs_valid {
            $display("    %h : %h", exq_rdata.addr, exq_rdata.bits);
            $display("    op1    : %h", exs_op1);
        }
    }
}

```

```

\$display(" op2      : %h", exs_op2);
\$display(" alu      : %h", exs_alu_result);
if inst_is_br(exs_ctrl) {
    \$display(" br take : ", exs_brunit_take);
}
\$display("MEM ----");
if mems_valid {
    \$display(" %h : %h", memq_rdata.addr, memq_rdata.bits);
    \$display(" mem stall : %b", memu_stall);
    \$display(" mem rdata : %h", memu_rdata);
    if mems_ctrl.is_csr {
        \$display(" csr rdata : %h", csru_rdata);
        \$display(" csr trap  : %b", csru_raise_trap);
        \$display(" csr vec   : %h", csru_trap_vector);
    }
    if memq_rdata.br_taken {
        \$display(" JUMP TO   : %h", memq_rdata.jump_addr);
    }
}
\$display("WB ----");
if wbs_valid {
    \$display(" %h : %h", wbq_rdata.addr, wbq_rdata.bits);
    if wbs_ctrl.rwb_en {
        \$display(" reg[%d] <= %h", wbs_rd_addr, wbs_wb_data)
    }
}
}

```

7.2.9 パイプライン処理をテストする

それでは、riscv-tests を実行してみましょう。試しに、RV64I の ADD のテストを実行します。

▼リスト 7.22: パイプライン処理のテスト

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-add.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-add.bin.hex
Test Result : 0 / 1
```

おや？ テストに失敗してしまいました。一体何が起きているのでしょうか？

7.3 データ依存の対処

7.3.1 正しく動かないプログラムを確認する

実は、ただ IF、ID、EX、MEM、WB ステージに処理を分割するだけでは、正しく命令を実行できません。例えば、リスト 7.23 のようなプログラムは正しく動きません。

`test/sample_datahazard.hex` を作成し、次のように記述します（リスト 7.23）。

▼ リスト 7.23: `sample_datahazard.hex`

```
0010811300100093 // 0: addi x1, x0, 1      4: addi x2, x1, 1
```

このプログラムでは、`x1` に $x0 + 1$ を代入した後、`x2` に $x1 + 1$ を代入します。シミュレータを実行し、どのように実行されるかを確かめます（リスト 7.24）。

▼ リスト 7.24: `sample_datahazard.hex` を実行する

```
$ make build
$ make sim
$ ./obj_dir/sim test/sample_datahazard.hex 7
...
#
#           5
ID -----
0000000000000004 : 00108113
  itype : 000010
  imm  : 0000000000000001
EX -----
0000000000000000 : 00100093
  op1    : 0000000000000000 ← x0
  op2    : 0000000000000001 ← 即値
  alu    : 0000000000000001 ← ゼロレジスタ + 1 = 1

#
#           6
ID -----
0000000000000008 : 00000000
  itype : 000000
  imm  : 0000000000000000
EX -----
0000000000000004 : 00108113
  op1    : 0000000000000000 ← x1
  op2    : 0000000000000001 ← 即値
  alu    : 0000000000000001 ← x1 + 1 = 2のはずだが1になっている

MEM -----
0000000000000000 : 00100093
...
```

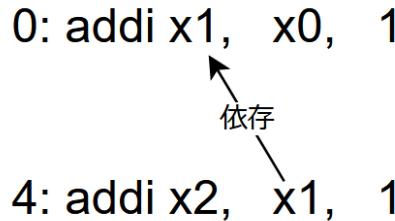
ログを確認すると、アドレス 0 の命令で `x1` が 1 になっているはずですが、アドレス 4 の命令で `x1` を読み込むときに `x1` は 0 になっています。

この問題は、まだアドレス 0 の命令の結果がレジスタファイルに書き込まれていないのに、アドレス 4 の命令でレジスタファイルで結果を読み出しているために発生しています。

7.3.2 データ依存とは何か？

ある命令 A の実行結果の値を利用する命令 B が存在するとき、命令 A と命令 B の間にはデータ依存（data dependence）があると呼びます。データ依存に対処するためには、命令 A の結果がレジスタに書き込まれるのを待つ必要があります。データ依存があることにより発生するパイプラ

インハザードのことをデータハザード (data hazard) と呼びます。



▲図 7.7: データ依存関係のあるプログラム

7.3.3 データ依存に対処する

レジスタの値を読み出すのは EX ステージです。データ依存に対処するために、データ依存関係があるときに EX ステージをストールさせます。

まず、MEM と EX か、WB と EX ステージにある命令の間にデータ依存があることを検知します (リスト 7.25)。例えば MEM ステージとデータ依存の関係にあるとき、MEM ステージの命令はライトバックする命令で、rd が EX ステージの rs1、または rs2 と一致しています。

▼リスト 7.25: データ依存の検知 (core.veryl)

```

// データハザード
let exs_mem_data_hazard: logic = mems_valid && mems_ctrl.rwb_en && (mems_rd_addr == exs_rs1_
>addr || mems_rd_addr == exs_rs2_addr);
let exs_wb_data_hazard : logic = wbs_valid && wbs_ctrl.rwb_en && (wbs_rd_addr == exs_rs1_add_
>r || wbs_rd_addr == exs_rs2_addr);
let exs_data_hazard : logic = exs_mem_data_hazard || exs_wb_data_hazard;
  
```

次に、データ依存があるときに、データハザードを発生させます (リスト 7.26)。データハザードを起こすためには、EX ステージの FIFO の `rready` と MEM ステージの `wvalid` に、データハザードが発生していないという条件を加えます。

▼リスト 7.26: データ依存があるときにデータハザードを起こす (core.veryl)

```

always_comb {
    // EX -> MEM
    exq_rready      = memq_wready && !exs_data_hazard;
    memq_wvalid     = exq_rvalid && !exs_data_hazard;
  
```

最後に、データハザードが発生しているかどうかをデバッグ表示します (リスト 7.27)。

▼リスト 7.27: データハザードが発生しているかをデバッグ表示する (core.veryl)

```

$display("EX -----");
if exs_valid {
    $display("  %h : %h", exq_rdata.addr, exq_rdata.bits);
    $display("  op1      : %h", exs_op1);
  
```

```
$display("  op2      : %h", exs_op2);
$display("  alu      : %h", exs_alu_result);
$display("  dhazard : %b", exs_data_hazard);
```

7.3.4 パイプライン処理をテストする

test/sample_datahazard.hex が正しく動くことを確認します。

▼リスト 7.28: sample_datahazard.hex が正しく動くことを確認する

```
$ make build
$ make sim
$ ./obj_dir/sim test/sample_datahazard.hex 7
...
#                      5
...
ID -----
 0000000000000004 : 00108113
  type : 000010
  imm  : 0000000000000001
EX -----
 0000000000000000 : 00100093
  op1   : 0000000000000000
  op2   : 0000000000000001
  alu   : 0000000000000001
  dhazard : 0
...
#                      6
...
EX -----
 0000000000000004 : 00108113
  op1   : 0000000000000000
  op2   : 0000000000000001
  alu   : 0000000000000001
  dhazard : 1 ←データハザードが発生している
MEM -----
 0000000000000000 : 00100093
  mem stall : 0
  mem rdata : 0000000000000000
WB -----
#                      7
...
EX -----
 0000000000000004 : 00108113
  op1   : 0000000000000000
  op2   : 0000000000000001
  alu   : 0000000000000001
  dhazard : 1
MEM -----
WB -----
 0000000000000000 : 00100093
```

```
reg[ 1] <= 0000000000000001 ← 1が書き込まれる

#
#          8
...
EX ----
0000000000000004 : 00108113
op1      : 0000000000000001 ← x1=1が読み込まれた
op2      : 0000000000000001
alu      : 0000000000000002 ← 正しい計算が行われている
dhsazard : 0 ← データハザードが解消された
MEM ----
WB ----
```

アドレス 4 の命令が、6 クロック目と 7 クロック目に EX ステージでデータハザードが発生し、アドレス 0 の命令が実行終了するのを待っているのを確認できます。

RV64I の riscv-tests も実行します。

▼ リスト 7.29: riscv-tests を実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-
...
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ma_data.bin.hex
...
Test Result : 51 / 52
```

正しくパイプライン処理が動いていることを確認できました。

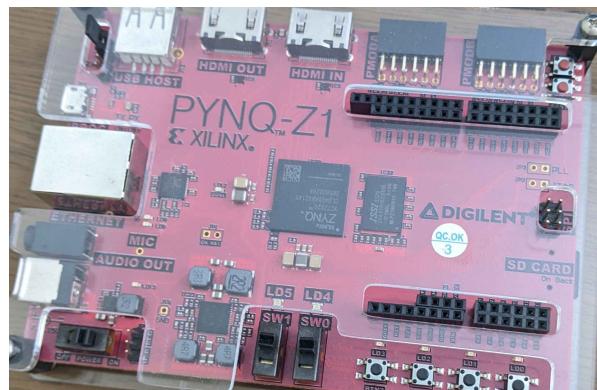
第 8 章

CPU を合成する

本章では FPGA で CPU を動作させます。

本章は Web 版で提供します。以下のサポートページを確認してください。

- <https://github.com/nananapo/veryl-riscv-book/wiki/techbookfest17-support-page>



▲ 図 8.1: PYNQ-Z1



▲ 図 8.2: TangMega138K

あとがき

いかがだったでしょうか。本書が自作 CPU の助けになれば幸いです。

著者について



阿部奏太 (kanataso) (kanapipopipo@X/Twitter, nananapo@GitHub)

いつの間にか自作 CPU の沼に沈んでいました。

カラオケまねきねこ (のまねっこ) とコメダ珈琲 (のエッグサンド) が好き。

計算機と法律に興味があります。

謝辞

本書は次の方々にレビューしていただきました。

- 石谷太一 (@taichi-ishitani^{*1})
- 井田健太 (@ciniml^{*2})
- 内田公太 (@uchan_nos^{*3})
- 初田直也 (@dalance^{*4})

筆者が CPU を作り始めたのは、井田さんの「RISC-V と Chisel で学ぶ はじめての CPU 自作^{*5}」を読んだのがきっかけでした。この本が無ければ、筆者は CPU を作ろうとは思わなかったかと思います。

その半年後から約一年間、サイボウズ・ラボ株式会社のサイボウズ・ラボユースの支援を受けることで、自作 CPU に集中して取り組むことができました (本書の一部はラボユースの期間に執筆されました)。メンターの内田さんにはとても感謝しています。

Veryl の作者の初田さんには、筆者が Veryl で CPU を作るにあたって見つけた不具合を迅速に修正していただきました。初田さんと石谷さんにはレビューでとても多くの指摘をいただき、本書の品質を向上できました。

執筆にあたって関わったすべての方に、この場をお借りしてお礼申し上げます。

^{*1} <https://github.com/taichi-ishitani>

^{*2} <https://github.com/ciniml>

^{*3} https://x.com/uchan_nos

^{*4} <https://github.com/dalance>

^{*5} <https://gihyo.jp/book/2021/978-4-297-12305-5>

参考文献

- [1] 天野 英晴, FPGA の原理と構成, オーム社
- [2] 坂井 修一, 論理回路入門, 培風館
- [3] 演算子の優先順位 - The Veryl Hardware Description Language, https://doc.veryl-lang.org/book/ja/05_language_reference/04_expression/01_operator_precedence.html
- [4] 演算子 - The Veryl Hardware Description Language, https://doc.veryl-lang.org/book/ja/05_language_reference/02_lexical_structure/01_operator.html
- [5] The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411
- 2.3. Immediate Encoding Variants
- [6] The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411
- 37. RV32/64G Instruction Set Listings
- [7] The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411
- 2.4. Integer Computational Instructions
- [8] The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411
- 2.5. Control Transfer Instructions
- [9] The RISC-V Instruction Set Manual Volume II: Privileged Architecture version 20240411
- Figure 10. Encoding of mtvec MODE field.
- [10] The RISC-V Instruction Set Manual Volume II: Privileged Architecture version 20240411
- 3.1.7. Machine Trap-Vector Base-Address Register
- [11] The RISC-V Instruction Set Manual Volume II: Privileged Architecture version 20240411
- 15. RISC-V Privileged Instruction Set Listings
- [12] David Patterson, John Hennessy(著), 成田 光彰 訳, コンピュータの構成と設計 MIPS Edition 第6版 [上] ~ハードウェアとソフトウェアのインタフェース~, 日経BP

VeryI で作る CPU

基本編 第 I 部 RV32I/RV64I の実装

2024 年 11 月 3 日 ver 1.0 (技術書典 17)

著 者 阿部奏太

発行者 阿部奏太

連絡先 kanastudio@oekaki.chat

印刷所 株式会社栄光

© 2024 ミ一ミミ研究室