Veryl で作る CPU

— 基本編 (の第 I 部) —

［著］kanataso

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 / はじめに

本書を手に取っていただき、ありがとうございます。

本書は、OS を実行できる程度の機能を持った RISC-V の CPU を、新しめのハードウェア記述言語である **Veryl** で記述する方法について解説した本です。

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。例えば本文に「後の章」と書かれていても、本書には含まれない場合があります。

本書の pdf, web 版は無料で配布されており、`https://github.com/nananapo/veryl-riscv-book` でダウンロード, 閲覧することができます。

続きが気になったり、誤植を見つけた場合は、GitHub をご確認ください。

本書の対象読者

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

前提とする知識

次の要件を満たしていると良いです。

- C, C++, C#, JavaScript, Python, Ruby Rust のような一般的なプログラミング言語をある程度使いこなすことができる
- 論理演算を知っている

問い合わせ先

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

- リポジトリ: `https://github.com/nananapo/veryl-riscv-book`

サポートページも用意しています。

- サポートページ：**??**

謝辞

本書は TODO 氏と TODO 氏にレビューしていただきました。また、本書の一部はサイボウズ・ラボ株式会社のサイボウズ・ラボユースの支援を受けて執筆されたものです。この場を借りて感謝します。

凡例

本書では、プログラムコードを次のように表示します。太字は強調を表します。

```
print("Hello, world!\n");          ←太字は強調
```

プログラムコードの差分を表示する場合は、追加されたコードを太字で、削除されたコードを取り消し線で表します。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。

```
~~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_
```

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

```
$ echo Hello          ←行頭の「$ 」はプロンプト、それ以降がユーザ入力
```

本文に対する補足情報や注意・警告は、次のようなノートや囲み枠で表示します。

ノートタイトル

ノートは本文に対する補足情報です。

タイトル

本文に対する補足情報です。

タイトル

本文に対する注意・警告です。

Intro

こんにちは! あなたは CPU を作成したことがありますか? 作成したことがあってもなくても大歓迎、この本は CPU 自作の面白さと Veryl を世に広めるために執筆されました。実装を始める前に、まずは RISC-V や使用する言語、本書の構成について簡単に解説します。RISC-V や Veryl のことを知っているという方は、本書の構成だけ読んでいただければ OK です。それでは始めましょう。

CPU の自作

TODO

- できるのか?できます
- VLSI お高い
- Efabless
- OpenMPW, TinyTapeout
- FPGA

CPU のテストはシミュレーションと実機 (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。ただし、実機がなくても実装を進めることができるので所有していなくても構いません。

RISC-V

RISC-V は、カリフォルニア大学バークレー校で開発された RISC の ISA(命令セットアーキテクチャ) です。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、RISC-V は仕様がオープンでカスタマイズ可能であるという特徴もあって、研究で利用されたり既に何種類もマイコンが市販されているなど、着実に広まっていっています。

インターネット上には多くの RISC-V の実装が公開されています。例として、rocket-chip(Chisel による実装), Shakti(Bluespec SV による実装), rsd(SystemVerilog による実装) が挙げられます。

本書では、RISC-V のバージョン riscv-isa-release-87edab7-2024-05-04 TODO を利用します。RISC-V の最新の仕様は、GitHub の riscv/riscv-isa-manual[*1] で確認することができます。

RISC-V には、基本整数命令セットとして RV32I, RV64I, RV32E, RV64E[*2]が定義されています。RV の後ろにつく数字はレジスタの長さ (XLEN) が何ビットかです。数字の後ろにつく文字が I の場合、XLEN ビットのレジスタが 32 個存在します。E の場合はレジスタの数が 16 個になります。

[*1] https://github.com/riscv/riscv-isa-manual/

[*2] RV128I もありますが、まだ Draft 段階です

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, CSR などの追加の命令や機能は拡張として定義されています。CPU が何を実装しているかを示す表現に ISA String というものがあり、例えばかけ算と割り算, 不可分操作ができる RV32I の CPU は `RV32IMA` と表現されます。

本書では、まず、 `RV32I` の CPU を作成します。これを `RV64IMACFD_Zicond_Zicsr_Zifencei` に進化させることを目標に実装を進めます。

使用する言語

ハードウェア記述言語とは、文字通り、ハードウェアを記述するための言語です。ハードウェアとは論理回路のことで、ハードウェア記述言語を使うと論理回路を記述, 生成することができます。これ以降、ハードウェア記述言語のことを HDL (Hardware Description Language) と書くことがあります。

有名な HDL としては Verilog HDL, SystemVerilog, VHDL が挙げられますが、本書では、CPU の実装に Veryl という HDL を使用します。Veryl は SystemVerilog の構文を書きやすくしたような言語で、Veryl のプログラムは SystemVerilog に変換することができます。そのため、SystemVerilog を利用できる環境で Veryl を使用することができます。

Veryl の構文や機能は SystemVerilog と似通っており、SystemVerilog が分かる人は殆ど時間をかけずに Veryl を書けるようになると思います。本書は SystemVerilog を知らない人を対象にしているため、SystemVerilog を知っている必要はありません。HDL や Veryl の記法は第 2 章「ハードウェア記述言語 Veryl」で解説します。

他には、回路のシミュレーションやテストのために C++, Python を利用します。プログラムがどのような意味や機能を持つかについては解説しますが、言語の仕様や書き方、ライブラリなどについては説明しません。

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. RV64I を実装する (第 6 章)
4. パイプライン化する (第 7 章)
5. M, A, C 拡張を実装する
6. UART と割り込みを実装する
7. OS を実行するために必要な CSR を実装する
8. OS を実行する

本書 (基本編の第 I 部) では、上の 1 から 4 までを実装, 解説します。

目次

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。
次のいずれかの環境を用意してください。筆者は Windows を推奨します。

- WSL が使える Windows
- Mac
- Linux

1.1 Veryl

1.1.1 Veryl のインストール

Veryl は本書で利用する HDL です。まず、Veryl のトランスパ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 をインストールすることができます。

Verylup は cargo, または GitHub の Release ページから入手することができます。cargo が入っている方は `cargo install verylup` でインストールしてください。cargo が入っていない場合は、veryl-lang/verylup[*1] から入手方法を確認することができます。

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: 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

veryl はまだ開発途上の言語であり、頻繁にバージョンが更新されます。最新の Veryl に更新す

[*1] https://github.com/veryl-lang/verylup

るには、次のようなコマンドを実行します。

▼リスト 1.2: Veryl の更新

```
$ verylup update
```

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: Veryl のバージョン TODO をインストールする

```
$ verylup install TODO
```

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: 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
--------------------
TODO
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: veryl のバージョン確認

```
$ veryl --version
veryl TODO
```

特定のバージョンの Veryl を使用するには、次のように veryl コマンドを実行します。

▼リスト 1.6: Veryl のバージョン TODO を使用する

```
$ veryl +TODO ← +でバージョンを指定する
```

本書で利用する Veryl のバージョン

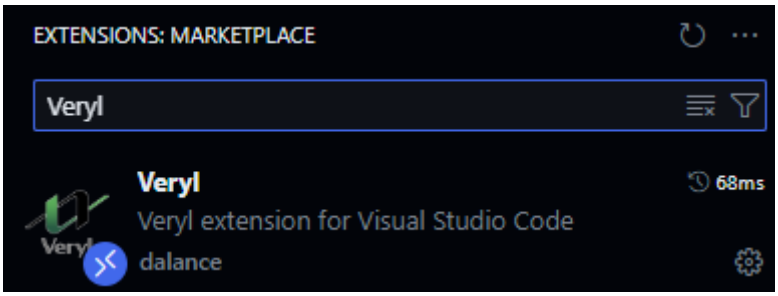
本書ではバージョン TODO を利用しますが、Veryl 側の問題によりプログラムをビルドできないことがあります。これの対処方法についてはサポートページを確認してください。

- サポートページ：**??**

1.1.2　VSCode の拡張のインストール

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ことをお勧めします。

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



▲図 1.1: Veryl の VSCode 拡張

1.2 Verilator

Verilator[*2]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。

`apt` 、または `brew` を利用してインストールすることができます。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`

1.3 riscv-gnu-toolchain

riscv-gnu-toolchain には、RISC-V 向けのコンパイラなどの toolchain が含まれています。

riscv-collab/riscv-gnu-toolchain[*3] 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。

FPGA を利用する方へ

TangMega138K を利用する人は GOWIN EDA, PYNQ-Z1 を利用する人は Vivado のインストールが必要です。

[*2] `https://github.com/verilator/verilator`
[*3] `https://github.com/riscv-collab/riscv-gnu-toolchain`

第 2 章
ハードウェア記述言語 Veryl

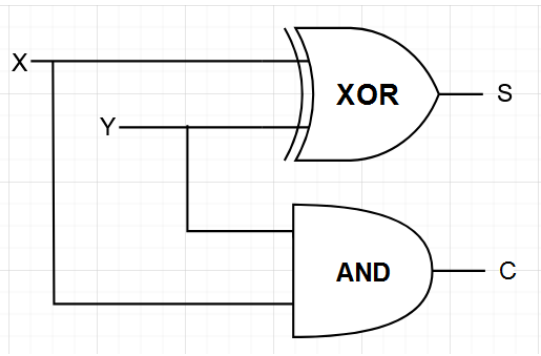
2.1 Veryl とは何か?

CPU を記述するといっても、いったい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。

2.1.1 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することができます。

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 [1] のことです。例えば、図 2.1 は半加算器です。半加算器は、入力 X, Y が決まると、出力 C, S が一意に決まります (表 2.1)。



▲図 2.1: 半加算器 (MIL 記法)

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 [1] です。例えば、入力が 1 になるたびにカウントダウンするカウンタを考えます。カウントダウンするためには、今のカウンタの値 (状態) を保持する必要があります。よって、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回路です。

▼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

1 ビットの値はフリップフロップという回路によって保持することができます。フロップフロップを N 個並列に並べると、N ビットの値を保持することができます。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** (置数器) といいます。基本的に、レジスタの値はリセット信号によって初期化し、クロック信号に同期したタイミングで変更します。

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設計できるかもしれませんが、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。有名な HDL としては、SystemVerilog が挙げられます。

SystemVerilog を利用すると、半加算器はリスト 2.1 のように記述することができます。

▼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,        // 出力地C
  output logic s         // 出力地S
);
  assign s = x ^ y; // ^はXOR演算
  assign c = x & y; // &はAND演算
endmodule
```

また、レジスタを利用した回路をリスト 2.2 のように記述することができます。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count + 1にする
      count <= count + 1;
    end
  end
endmodule
```

HDL を使用すると、論理回路を、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値を格納する操作として記述できます。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) といいます。

HDL で記述された論理回路は、**合成系**によって、RTL から実際の回路のデータに変換されます。

2.1.3　Veryl

メジャーな HDL といえば、Verilog, SystemVerilog などが挙げられます[*1]。

Verilog 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、スーパーセットであることから、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[*2]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, 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、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 プログラムは、コンパイラ (トランスパイラ) によって、自然で読みやすい SystemVerilog プログラムに変換されます。よって、Veryl は旧来の SystemVerilog の環境と共存することができ、SystemVerilog の資産を利用することができます。

2.2　基本

それでは、Veryl の書き方を簡単に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。

TODO

[*1] VHDL が無いじゃないかと思った方、すみません。VHDL のことを私はよく知らないので無いことにしました。

[*2] 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。ヤバすぎる

- リテラル
- 型 (logic, リテラル,struct,enum,clock,reset)
- type
- const
- 連接, repeat
- 変数 (var, let)
- 式 (演算子)
- if 式, case 式
- 文 (if, case, for)
- function
- SystemVerilog の機能 (システムタスク)

2.3 module

- module 宣言
- ポート
- パラメータ
- インスタンス化
- always_comb, assign
- always_ff, if_reset
- initial, final

2.4 interface

- パラメータ化
- modport
- インスタンス化

2.5 package

- import

2.6 ジェネリクス

- function
- module
- interface
- package

2.7 サンプルプログラム

- 半加算器
- 全加算器
- カウンタ

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?

上に書かれている文章の意味が分からなくても大丈夫です。詳しく説明します。

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

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

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

メモリからプログラムを読み込んで実行するのが CPU の仕事ということが分かりました。これ

[*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

をもう少し掘り下げます。

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

CPU はプログラムを実行しますが、一気にすべてのプログラムを読み込んだり実行するわけではなく、プログラムの最小単位である「命令」を一つずつ読み込んで実行します。命令をメモリに要求、取得することを、命令をフェッチするといいます。

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

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

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

計算対象のデータにレジスタと即値のど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。

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

計算やメモリアクセスが終わると、その結果をレジスタに格納します。例えば、足し算を行う命令なら足し算の結果、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。

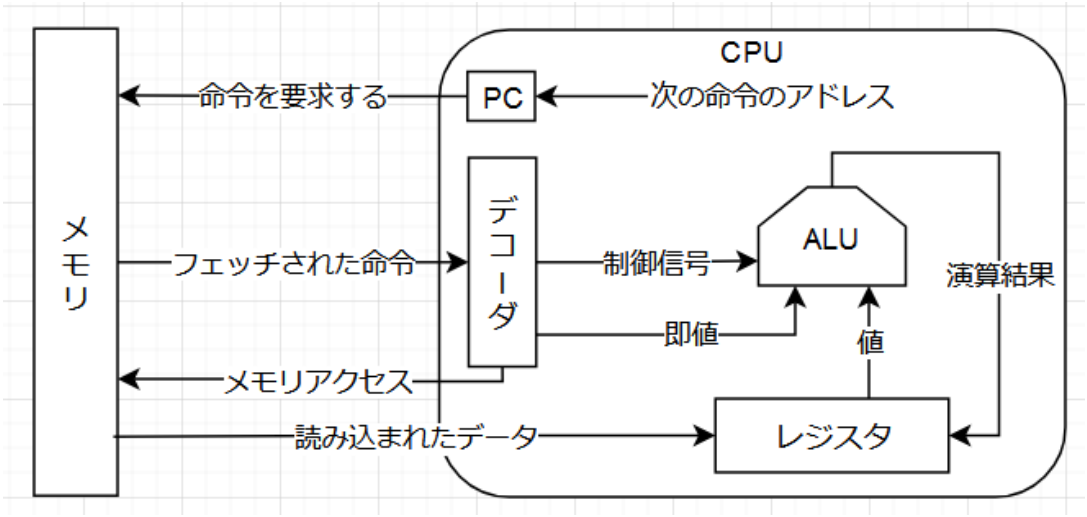
これで命令の実行は終わりですが、CPU は次の命令を実行する必要があります。今現在実行している命令のアドレスを格納しているレジスタのことをプログラムカウンタ (PC) と言い、CPU は PC の値をメモリに渡すことで命令をフェッチしています。

CPU は次の命令を実行するために、PC の値を次の命令のアドレスに設定します。ジャンプ命令の場合は、PC の値をジャンプ先のアドレスに設定します。分岐命令の場合は、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、通常の命令と同じように、PC の値を次の命令のアドレスに設定します。

ここまでの話をまとめると、CPU の動作は次のようになります。

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に設定する

CPU が何をするものなのかが分かりましたか? 実装を始めましょう。



▲図 3.1: 雑な CPU の図

3.2　プロジェクトの作成

まず、Veryl のプロジェクトを作成します。ここでは適当に core という名前にしています。

▼リスト 3.1: 新規プロジェクトの作成

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

すると、プロジェクト名のフォルダと、その中に `Veryl.toml` が作成されます。 `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 のプログラムを格納するために、プロジェクトのフォルダ内に src フォルダを作成しておいてください。

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

3.3　定数の定義

いよいよプログラムを記述していきます。まず、CPU 内で何度も使用する定数や型を記述するパッケージを作成します。

`src/eei.veryl` を作成し、次のように記述します (リスト 3.3)。

▼リスト 3.3: eei.veryl

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

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

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

3.4　メモリ

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

これを実現するために、次のような要件のメモリを実装します。

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

3.4.1　メモリのインターフェースを定義する

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

`src/membus_if.veryl` を作成し、次のように記述します (リスト 3.4)。

▼リスト 3.4: 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,
    }
```

```
}
```

▼表 3.1: メモリモジュールに必要なポート

ポート名	型	向き	意味
clk	clock	input	クロック信号
rst	reset	input	リセット信号
valid	logic	input	メモリアクセスを要求しているかどうか
ready	logic	output	メモリアクセスを受容するかどうか
addr	logic<ADDR_WIDTH>	input	アクセスするアドレス
wen	logic	input	書き込みかどうか (1 なら書き込み)
wdata	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

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`, `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、さらにポートの相互接続を簡潔にすることができます。

3.4.2　メモリ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 3.5)。

▼リスト 3.5: メモリモジュールの定義 (memory.veryl)

```
module memory::<DATA_WIDTH: const, ADDR_WIDTH: const> (
    clk       : input   clock                                          ,
    rst       : input   reset                                          ,
    membus    : modport membus_if::<DATA_WIDTH, ADDR_WIDTH>::slave,
    FILE_PATH: input   string                                          , // メモリの初期値が格納さ›
›れたファイルのパス
) {
    type DataType = logic<DATA_WIDTH>;

    var mem: DataType [2 ** ADDR_WIDTH];

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

    always_comb {
```

```
        membus.ready = 1;
    }

    always_ff {
        membus.rvalid = membus.valid;
        membus.rdata  = mem[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

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以外に、membus_if インターフェースと string 型の `FILE_PATH` を定義しています。memory モジュールを利用する時、`FILE_PATH` ポートには、メモリの初期値が格納されたファイルのパスを指定します。初期化は `$readmemh` システムタスクで行います。

読み込み、書き込み時の動作は次の通りです。

読み込み

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

書き込み

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

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、デザイン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。2 つの内、データのビット幅を示す定数を eei パッケージに定義します (リスト 3.6)。

▼リスト 3.6: 1 つのデータのビット幅を示す定数を定義する (eei.veryl)

```
    // メモリのデータ幅
    const MEM_DATA_WIDTH: u32 = 32;
```

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 3.7)。

▼リスト 3.7: 最上位モジュールの定義 (top.veryl)

```
import eei::*;

module top (
    clk           : input clock ,
    rst           : input reset ,
    MEM_FILE_PATH: input string,
) {

    inst membus: membus_if::<MEM_DATA_WIDTH, 20>;

    inst mem: memory::<MEM_DATA_WIDTH, 20> (
        clk                       ,
        rst                       ,
        membus                    ,
        FILE_PATH: MEM_FILE_PATH,
    );
}
```

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。

ジェネリックパラメータの `DATA_WIDTH` には、`eei::MEM_DATA_WIDTH` を指定しています。`membus` インターフェースのアドレスの幅と、memory モジュールのメモリ容量には、適当に 20 を指定しています。これにより、メモリ容量は 32 ビット * (2 ** 20) = 4 メビバイトになります。

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ることができ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ていきます。

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 3.8)。

▼リスト 3.8: 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, リセット信号, membus_if インターフェースをポートに持ちます。membus_if 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 prefix は instruction fetch(命令フェッチ) の略です。 `if_is_requested` は現在フェッチ中かどうかを管理しており、フェッチ中のアドレスを `if_pc_requested` に格納しています。

`always_comb` ブロックでは、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, 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` , `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時、 `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。このとき、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、0,4,8,c,10,... という順番のアドレスの命令を、次々にフェッチするようになっています。

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プログラムです。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 インターフェースを接続します。

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 3.9)。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 3.9: 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<20>   {
        return addr[20 + $clog2(MEM_DATA_WIDTH / 8) - 1:$clog2(MEM_DATA_WIDTH / 8)];
    }
```

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[21:2]` を切り取っています。

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 3.10)。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 3.10: 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, 20>;
    inst membus_core: membus_if::<ILEN, XLEN>;
```

`membus` と `membus_core` を接続します (リスト 3.11)。アドレスは、`addr_to_memaddr` 関数で変換した値を割り当てます。

▼リスト 3.11: 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。これによって、メモリと CPU が接続されました。

▼リスト 3.12: top.veryl 内で core モジュールをインスタンス化する

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

3.6.3　命令フェッチをテストする

ここまでのプログラムが正しく動くかを検証します。

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

まず、プログラムをビルドします。

▼リスト 3.13: Veryl プログラムのビルド

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

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

シミュレータのビルドには Verilator を利用します。Verilator は与えられた SystemVerilog プログラムを C++ プログラムに変換することでシミュレータを生成します。verilator を利用するために、次のような C++ プログラムを書く必要があります[*2]。

`src/tb_verilator.cpp` を作成し、次のように記述します (リスト 3.14)。

▼リスト 3.14: tb_verilator.cpp

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

namespace fs = std::filesystem;

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

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

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

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

[*2] Verilog プログラムだけでビルドすることもできます

```
            return 1;
        }
    }

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

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

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

    dut->final();
}
```

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

このプログラムは、コマンドライン引数として次の 2 つの値を受け取ります。

MEMORY_FILE_PATH

メモリの初期値のファイルへのパス

実行時に top モジュールの MEM_FILE_PATH ポートに渡されます。

CYCLE

何クロックで実行を終了するかを表す値

0 のときは終了しません。デフォルト値は 0 です。

Verilator によるシミュレーションは、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 3.15)。

▼リスト 3.15: シミュレータのビルド

```
$ verilator --cc -f core.f --exe src/tb_verialtor.cpp --top-module top --Mdir obj_dir
$ make -C obj_dir -f Vcore_top.mk ←シミュレータをビルドする
$ mv obj_dir/Vcore_top obj_dir/sim ←シミュレータの名前をsimに変更する
```

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

-f

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

--exe

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

--top-module

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

--Mdir

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

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

メモリの初期化用ファイルの作成

シミュレータを実行する前にメモリの初期値となるファイルを作成します。 `src/sample.hex` を作成し、次のように記述します (リスト 3.16)。

▼リスト 3.16: sample.hex

```
01234567
89abcdef
deadbeef
cafebebe
←必ず末尾に改行をいれてください
```

値は 16 進数で 4 バイトずつ記述されています。シミュレータを実行すると、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)。

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 3.17)。

▼リスト 3.17: 命令フェッチの動作チェック

```
$ obj_dir/sim src/sample.hex 5
00000000 : 01234567
00000004 : 89abcdef
```

▼表 3.2: sample.hex によって設定されるメモリの初期値

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

```
00000008 : deadbeef
0000000c : cafebebe
```

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

Makefile の作成

ビルド、シミュレータのビルドのために一々コマンドを打つのは非常に面倒です。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 3.18)。

▼リスト 3.18: Makefile

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

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

build:
	veryl fmt
	veryl build

clean:
	veryl clean
	rm -rf $(OBJ_DIR)

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

これ以降、次のように Veryl プログラムのビルド, 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 3.19)。

▼リスト 3.19: Makefile によって追加されたコマンド

```
$ make build ← Verylプログラムのビルド
$ make sim ←シミュレータのビルド
```

```
$ make clean ←ビルドした成果物の削除
```

3.6.4　フェッチした命令を FIFO に格納する

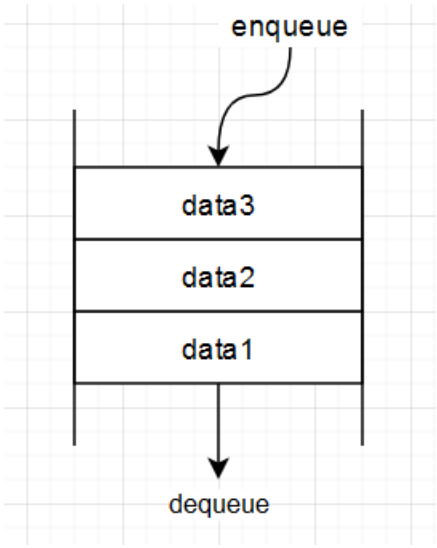
フェッチした命令は次々に実行されますが、その命令が何クロックで実行されるかは分かりません。命令が常に 1 クロックで実行される場合は、現状の常にフェッチし続けるようなコードで問題ありませんが、例えばメモリにアクセスする命令は実行に何クロックかかるか分かりません。

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たら次の命令をフェッチするようにします。すると、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

この場合、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サイクルが必要です。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



▲図 3.2: FIFO

そこで、**FIFO**(First In First Out, ファイフォ) 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。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。

`src/fifo.veryl` を作成し、次のように記述します (リスト 3.20)。

▼リスト 3.20: 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;

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

    if WIDTH == 1 :wready_block {
        assign wready = head == tail && rready;
    } else {
        assign wready = tail_plus1 != 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 ** WIDTH - 1` 個格納することができる FIFO です。操作は次のように行います。

データを追加する

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

データを取り出す

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

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てしまわないときです。この制限から、FIFO には最大でも `2 ** WIDTH - 1` 個しかデータを格納することができません。データを取り出せる状況とは、 `head` と `tail` の指す場所が違うときです。

`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することが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 3.21)。 `if_fifo_type` には、命令のアドレス (`addr`) と命令のビット列 (`bits`) を格納するためのメンバーを含めます。

▼リスト 3.21: 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,
    bits: Inst,
}
```

次に、FIFO と接続するための変数を定義します (リスト 3.22)。wdata と rdata のデータ型は `if_fifo_type` にしています。

▼リスト 3.22: 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 3.23)。`DATA_TYPE` パラメータに `if_fifo_type` を渡すことで、アドレスと命令のペアを格納することができるようにします。`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 3.23: 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 3.24)。

▼リスト 3.24: フェッチ処理の変更 (core.veryl)

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

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

リスト 3.24 では、メモリに命令フェッチを要求する条件を、FIFO に空きがあるという条件に変更しています。これにより、FIFO があふれてしまうことがなくなります。また、とりあえず FIFO から常にデータを取り出すようにしています。

最後に、命令をフェッチできたら FIFO に格納するコードを `always_ff` ブロックの中に追加します (リスト 3.25)。

▼リスト 3.25: 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;
        }
    }
```

また、`if_fifo_wvalid` と `if_fifo_wdata` を 0 に初期化します (リスト 3.26)。

▼リスト 3.26: 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時、`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。

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

命令フェッチは FIFO に空きがあるときにのみ行うため、まだ追加されていないデータが `if_fifo_wdata` 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。

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 3.27)。

▼リスト 3.27: 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);
        }
    }
```

それでは、シミュレータを実行します (リスト 3.28)。命令がフェッチされて表示されるまでに、FIFO に格納して取り出すクロック分だけ遅延があることに注意してください。

▼リスト 3.28: 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。

命令のビット列には、基本的に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)
: 命令のビット列の中に直接含まれる数値です。

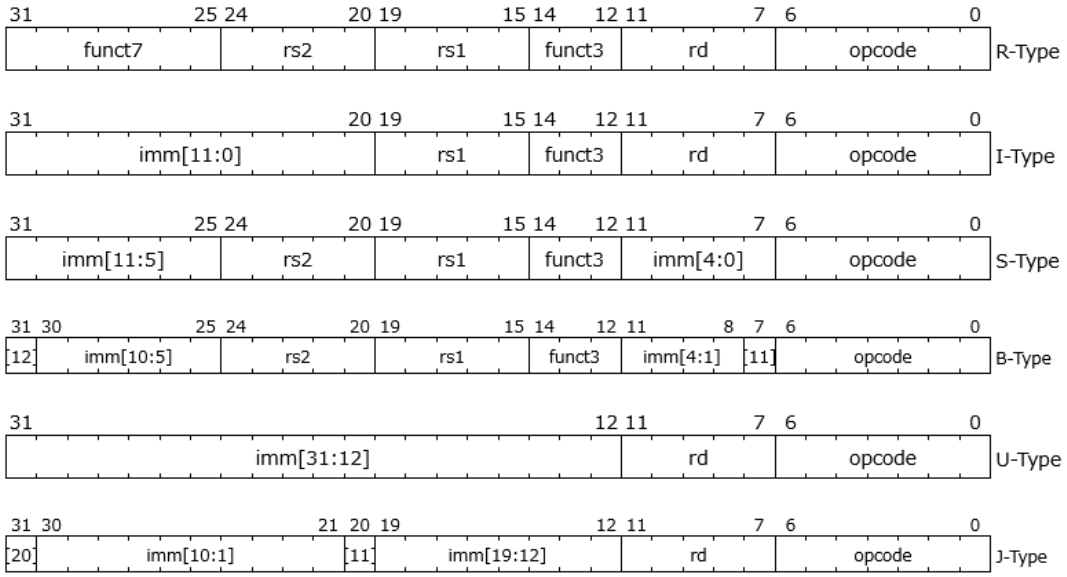
ソースレジスタ (Source Register) の番号
: 計算やメモリアクセスに使う値が格納されているレジスタの番号です。レジスタは 32 個あるため 5 ビットの値になっています。

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)。

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), SUBI(即値を使った引き算) 命令に使用されています。



▲図 3.3: RISC-V の命令形式 [2]

S 形式

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

B 形式

ソースレジスタ (rs1, rs2) が 2 つの命令形式です。12 ビットの即値 (imm[12], imm[11], imm[10:5], imm[4:1]) が命令中に含まれています。分岐命令に使用されており、ソースレジスタの計算の結果が分岐を成立させる場合、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　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 3.29)。

▼リスト 3.29: 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列挙するための構造体です。`itype` には命令の形式、`funct3`, `funct7` には、それぞれ命令の `funct3`, `funct3` フィールドを格納します。これ以外の構造体のメンバーについては、使用するときに説明します。

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 3.30)。

▼リスト 3.30: eei.veryl に追加で記述する (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については、仕様書 [3] 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 3.31)。

▼リスト 3.31: 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},
            OP_AUIPC : {InstType::U, T, F, F, F, F},
            OP_JAL   : {InstType::J, T, F, F, T, F},
            OP_JALR  : {InstType::I, T, F, F, T, F},
            OP_BRANCH: {InstType::B, F, F, F, F, F},
            OP_LOAD  : {InstType::I, T, F, F, F, T},
            OP_STORE : {InstType::S, F, F, F, F, F},
            OP_OP    : {InstType::R, T, F, T, F, F},
            OP_OP_IMM: {InstType::I, T, F, T, F, F},
            default  : {InstType::X, F, F, F, F, F},
        }, f3, f7};
    }
}
```

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

即値の生成

B 形式の命令について考えます。まず、命令のビット列から即値部分を取り出して、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 命令について考えます。ADDI 命令は、即値とソースレジスタの値を足し、デスティネーションレジスタに結果を格納する命令です。

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

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

3.7.3　デコーダのインスタンス化

inst_decoder モジュールを、 `core` モジュールでインスタンス化します (リスト 3.32)。

▼リスト 3.32: inst_decoder のインスタンス化 (core.veryl)

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

```
    var inst_imm : UIntX   ;

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

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

デバッグ用の `always_ff` ブロックに、デコードした結果を表示するプログラムを記述します (リスト 3.33)。

▼リスト 3.33: デコード結果の表示プログラム (core.veryl)

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

`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 3.34)。

▼リスト 3.34: 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     : fffdb5ea
0000000c : cafebebe
  itype   : 000000
  imm     : 00000000
```

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

3.8　レジスタの定義と読み込み

RV32I 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。

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

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、サイズが 32 の `UIntX` 型のレジスタの配列を定義します。

▼リスト 3.35: レジスタの定義 (core.veryl)

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

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

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。

ここで、プログラムを簡単にするために、命令中のソースレジスタの番号にあたる場所に、常にソースレジスタの番号が書かれていると解釈します。更に、命令がレジスタの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 3.36)。

▼リスト 3.36: 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めて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 3.37)。`$display` システムタスクで、命令のレジスタ番号と値を表示します。

▼リスト 3.37: レジスタの値を表示する (core.veryl)

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

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 (0 か 1 か分からない)[*3]になってしまいます。

これではテストする意味がないため、レジスタの値を適当な値に初期化します (リスト 3.38)。

▼リスト 3.38: レジスタの値を初期化する[*4] (core.veryl)

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

`always_ff` ブロックの `if_reset` で、n 番目 ($32 > n > 0$) のレジスタの値を `n + 100` で初期化します。

レジスタの値が読み込めていることを確認します (リスト 3.39)。

▼リスト 3.39: レジスタ読み込みのデバッグ

```
$ make build
$ make sim
```

[*3] 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

[*4] 「i は変数だから if_reset で使えません」のようなエラーが出る場合、申し訳ありませんが for 文を使わずに 1 つずつ初期化してください。

```
$ obj_dir/sim sample.hex 7
00000000 : 01234567
  itype   : 000010
  imm     : 00000012
  rs1[ 6] : 0000006a
  rs2[18] : 00000076
00000004 : 89abcdef
  itype   : 100000
  imm     : fffbc09a
  rs1[23] : 0000007b
  rs2[26] : 0000007e
00000008 : deadbeef
  itype   : 100000
  imm     : fffdb5ea
  rs1[27] : 0000007f
  rs2[10] : 0000006e
0000000c : cafebebe
  itype   : 000000
  imm     : 00000000
  rs1[29] : 00000081
  rs2[15] : 00000073
```

`01234567` 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` になります。これは 16 進数で `6a` 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 を作り、計算する

基本整数命令セットの命令は、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。レジスタと即値が揃い、計算の対象となるデータが手に入るようになりました。CPU の計算を行うユニットである **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, 即値の幅は `XLEN` です。計算には符号付き整数と符号無し整数向けの計算があります。

これに利用するために、eei モジュールに `XLEN` ビットの符号付き整数型を定義します (リスト 3.40)。

▼リスト 3.40: XLEN ビットの符号付き整数を定義する (eei.veryl)

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

次に、 `src/alu.veryl` を作成し、次のように記述します (リスト 3.41)。

▼リスト 3.41: 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)。

命令が ALU でどのような計算を行うかは命令によって異なります。仕様書で整数演算命令として定義されている命令 [4] は、命令の funct3 表 3.4, funct7 フィールドによって計算の種類を特定することができます。

▼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

▼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|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 3.31)。

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

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

I 形式, S 形式

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

U 形式, J 形式

U 形式, J 形式は、即値と PC を足した値、または即値を使う命令に使われています。 `op1` , `op2` は、それぞれ PC `inst_pc` , 即値 `inst_imm` になります。J 形式は JAL 命令に利用されており、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 3.43)。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 3.43: 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`, `op2`, `alu_result` を表示します (リスト 3.44)。

▼リスト 3.44: ALU の結果を表示する (core.veryl)

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

`sample.hex` を、次のように書き換えます (リスト 3.45)。

▼リスト 3.45: 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	意味
00000000	addi x1, x0, 32	x1 = x0 + 32
00000004	auipc x2, 256	x2 = pc + 256
00000008	add x3, x1, x2	x3 = x1 + x2

シミュレータを実行し、結果を確かめます (リスト 3.46)。

▼リスト 3.46: 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、`00000000` と表示されています。`op2` は即値です。即値は 32 であるため、16 進数で `00000020` と表示されています。ALU の計算結果として、0 と 32 を足した結果 `00000020` が表示されています。

auipc x2, 256

`op1` は PC です。`op1` には、命令のアドレス `00000004` 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 `00100000` 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 `00100004` 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、`00000065` と表示されています。2 番目のレジスタは `102` として初期化しているので、`00000066` 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 `000000cb` が表示されています。

3.10　レジスタに結果を書き込む

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

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

3.10.1　ライトバック処理を実装する

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します。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 3.31)。

とりあえず、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するようにします (リスト 3.47)。

▼リスト 3.47: 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表示します (リスト 3.48)。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 3.48: 結果の表示 (core.veryl)

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

シミュレータを実行し、結果を確かめます (リスト 3.49)。

▼リスト 3.49: ライトバックのデバッグ

```
$ make build
$ make sim
$ obj_dir/sim sample.hex 6
00000000 : 02000093
  itype   : 000010
  imm     : 00000020
```

```
  rs1[ 0] : 00000000
  rs2[ 0] : 00000000
  op1     : 00000000
  op2     : 00000020
  alu res : 00000020
  reg[ 1] <= 00000020
00000004 : 00100117
  itype   : 010000
  imm     : 00100000
  rs1[ 0] : 00000000
  rs2[ 1] : 00000020
  op1     : 00000004
  op2     : 00100000
  alu res : 00100004
  reg[ 2] <= 00100004
00000008 : 002081b3
  itype   : 000001
  imm     : 00000000
  rs1[ 1] : 00000020
  rs2[ 2] : 00100004
  op1     : 00000020
  op2     : 00100004
  alu res : 00100024
  reg[ 3] <= 00100024
```

addi x1, x0, 32

x1 に、0 と 32 を足した値 (`00000020`) 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 (`00100000`) と PC(`00000004`) を足した値 (`00100004`) を格納しています。

add x3, x1, x2

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

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

最後に、テストのためにレジスタの値を初期化するようにしていたコードを削除します (リスト 3.50)。

▼リスト 3.50: 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 3.42)。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, SW 命令を実装します。

memunit モジュールの作成

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

▼リスト 3.51: memunit.veryl

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

module memunit (
    clk   : input   clock                                   ,
    rst   : input   reset                                   ,
    valid : input   logic                                   ,
    is_new: input   logic                                   , // 命令が新しく供給されたかど›
›うか
    ctrl  : input   InstCtrl                                , // 命令のInstCtrl
    addr  : input   Addr                                    , // アクセスするアドレス
    rs2   : input   UIntX                                   , // ストア命令で書き込むデー›
›タ
    rdata : output  UIntX                                   , // ロード命令の結果 (stall = ›
›0のときに有効)
```

```
    stall : output  logic                                       , // メモリアクセス命令が完了し
ていない
    membus: modport membus_if::<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.itype == InstType::S || ctrl.is_load;
    }

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

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

    var state: State;

    var req_wen  : logic                 ;
    var req_addr : Addr                  ;
    var req_wdata: 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時、ALU から受け取ったアドレスをメモリに渡して操作を実行します。

命令がメモリにアクセスする命令かどうかは `inst_is_memop` 関数で判定します。ストア命令のとき、命令の形式は S 形式です。ロード命令のとき、デコーダは `InstCtrl.is_load` を `1` にしています (リスト 3.31)。

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

State::Init

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

State::WaitReady

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

State::WaitValid

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

メモリにアクセスする命令のとき、memunit モジュールは `Init` , `WaitReady` , `WaitValid` の順で状態を移動するため、実行には少なくとも 3 クロックが必要です。その間、CPU はレジスタ

のライトバック処理や FIFO からの命令の取り出しを待つ必要があります。

CPU の実行が止まることを、CPU が**ストール** (Stall) すると言います。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` フラグが存在します。有効な命令が供給されているとき、 `state` やメモリの状態に応じて、次のように `stall` の値を決定します (表 3.7)。

▼表 3.7: stall の値の決定方法

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

アドレスが 4 バイトに整列されていない場合の動作

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

memunit モジュールのインスタンス化

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

まず、命令が供給されていることを示す信号 `inst_valid` と、命令が現在のクロックで供給されたことを示す信号 `inst_is_new` を作成します (リスト 3.52)。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 3.52: inst_valid, inst_is_new の定義 (core.veryl)

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

`inst_is_new` の値を更新します (リスト 3.53)。

命令が現在のクロックで供給されたかどうかは、FIFO の `rvalid` , `rready` を観測することでわかります。 `rvalid` が `1` のとき、 `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` かどうかについては考えません)

▼リスト 3.53: 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;
            }
        }
    }
```

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

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

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

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 3.54)。

▼リスト 3.54: 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` に変更されるため、既存の `membus` を `i_membus` に置き換えてください (リスト 3.55)。

▼リスト 3.55: membus を i_membus に置き換える (core.veryl)

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

次に、top モジュールでの調停を実装します (リスト 3.56)。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 3.56: メモリへのアクセス要求の調停 (top.veryl)

```
    inst membus  : membus_if::<MEM_DATA_WIDTH, 20>;
    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どうかを記録する
- メモリが要求の結果を返すとき、 `memarb_last_i` を見て、 `i_membus` と `d_membus` のどちらか片方の `rvalid` を `1` にする

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

core モジュールとの接続を次のように変更します (リスト 3.57)。

▼リスト 3.57: membus を 2 つに分けて接続する (top.veryl)

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

memory モジュールと memunit を接続する準備が整ったので、memunit モジュールをインスタンス化します (リスト 3.58)。

▼リスト 3.58: memunit モジュールのインスタンス化 (core.veryl)

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

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

memunit モジュールの処理待ちとライトバック

最後に、memunit モジュールが処理中のときは命令を 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。

まず、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 3.59)。

▼リスト 3.59: 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 3.60)。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 3.60: 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、現在のプログラムでは、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、次のようにプログラムを変更します (リスト 3.61)。

▼リスト 3.61: 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するようにします (リスト 3.62)。

▼リスト 3.62: 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, SW 命令が正しく動作していることを確認するために、デバッグ出力を次のように変更します (リスト 3.63)。

▼リスト 3.63: メモリモジュールの状態を出力する (core.veryl)

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

また、ここからのテストは実行するクロック数が多くなるため、ログに何クロック目かを表示することで、ログを読みやすくします (リスト 3.64)。

▼リスト 3.64: 何クロック目かを出力する (core.veryl)

```
    var clock_count: u64;

    always_ff {
        if_reset {
            clock_count = 1;
```

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

LW, SW 命令のテストのために、sample.hex を次のように変更します (リスト 3.65)。

▼リスト 3.65: 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: メモリに格納するデータ

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

アドレス `0x20` には、データ `deadbeef` を格納しています。1 つ目の命令で `deadbeef` が読み込まれ、3 つ目の命令で `00000400` を書き込み、4 つ目の命令で `00000400` が読み込まれます。

シミュレーションを実行し、結果を確かめます (リスト 3.66)。

TODO

▼リスト 3.66: LW, SW 命令のテスト

```
$ make build
$ make sim
$ obj_dir/sim src/sample.hex 13
#                     3
00000000 : 02002503
  itype     : 000010
  imm       : 00000020
  rs1[ 0]   : 00000000
  rs2[ 0]   : 00000000
  op1       : 00000000
  op2       : 00000020
  alu res   : 00000020
```

```
  mem stall : 1 ← LW命令でストールしている
  mem rdata : 02b02023
(省略)
#                     5
00000000 : 02002503
  itype     : 000010
  imm       : 00000020
  rs1[ 0]   : 00000000
  rs2[ 0]   : 00000000
  op1       : 00000000
  op2       : 00000020
  alu res   : 00000020
  mem stall : 0 ← LWが終わったので0になった
  mem rdata : deadbeef
  reg[10] <= deadbeef ← 0x20の値が読み込まれた
(省略)
#                    12
0000000c : 02002603
  itype     : 000010
  imm       : 00000020
  rs1[ 0]   : 00000000
  rs2[ 0]   : 00000000
  op1       : 00000000
  op2       : 00000020
  alu res   : 00000020
  mem stall : 0
  mem rdata : 00000400
  reg[12] <= 00000400 ←書き込んだ値が読み込まれた
```

3.11.2　LB, LBU, LH, LHU 命令の実装

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

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

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

▼リスト 3.67: W と D 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;
```

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

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。

▼表 3.9: ロード命令の funct3

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

▼リスト 3.68: 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,
};
```

3.11.3　SB, SH 命令の実装

次に、SB, SH 命令を実装します。

memory モジュールで書き込みマスクをサポートする

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

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

▼リスト 3.69: wmask の定義 (membus_if.veryl)

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

▼リスト 3.70: modport master に wmask を追加する (membus_if.veryl)

```
modport master {
    ...
```

```
        wmask : output,
        ...
    }
```

▼リスト 3.71: modport slave に wmask を追加する (membus_if.veryl)

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

バイト単位で指定するため、 `wmask` の幅は 4 ビットです。

次に、memory モジュールで書き込みマスクをサポートします。

▼リスト 3.72: 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> (
    clk       : input   clock                                          ,
    rst       : input   reset                                          ,
    membus    : modport membus_if::<DATA_WIDTH, ADDR_WIDTH>::slave,
    FILE_PATH: input   string                                          , // メモリの初期値が格納さ›
›れたファイルのパス
) {
    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;
    for i in 0..DATA_WIDTH :wm_expand_block {
        assign wmask_expand[i] = wmask_saved[i / 8];
    }

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

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

    var addr_saved : 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 & ~wma›
›sk_expand;
        }
    }

    always_ff {
        if_reset {
            state          = State::Ready;
            membus.rvalid = 0;
            membus.rdata  = 0;
            addr_saved    = 0;
            wdata_saved   = 0;
            wmask_saved   = 0;
            rdata_saved   = 0;
        } else {
            case state {
                State::Ready: {
                                    membus.rvalid = membus.valid & !membus.wen;
                                    membus.rdata  = mem[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保存し、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

要求の内容を保存します。また、`rdata_saved` に、指定されたアドレスのデータを保存します。次のクロックで、書き込みマスクを使った書き込みを行い、要求の処理を終了します。

top モジュールの調停処理で、`wmask` も調停するようにします。

▼リスト 3.73: wmask の設定 (top.veryl)

```
    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;
        membus.wmask = d_membus.wmask;
    } 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;
        membus.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するようになったので、memunit モジュールで wmask を設定します。

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

▼リスト 3.74: req_wmask の定義 (memunit.veryl)

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

▼リスト 3.75: membus 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 3.76: 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 3.77: 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], ← LB, LBUのとき、アドレス下位2ビット分だけ1を左シフトする
        2'b01  : case addr[1:0] { ← LH, LHU命令のとき
            2      : 4'b1100, ←上位2バイトに書き込む
            0      : 4'b0011, ←下位2バイトに書き込む
            default: 'x,
        },
        2'b10  : 4'b1111, ← LW命令のとき、全体に書き込む
        default: 'x,
    };
```

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

簡単なテストを作成し、動作をテストします。

2 つテストを記載するので、正しく動いているか確認してください。

▼リスト 3.78: 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 3.79: 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 if 文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 [5] に次の命令が定義されています。

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

まず、無条件ジャンプを実装します。

▼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

JAL(Jump And Link) 命令は、PC+ 即値でジャンプ先を指定します。ここで 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を指しています。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。(即値の生成については inst_decoder モジュール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、JALR 命令を次のようにデコードしています。

- `InstCtrl.is_jump` = 1
- `InstCtrl.is_aluop` = 0

無条件ジャンプであるかどうかは `InstCtrl.is_jump` で確かめることができます。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 3.42)。

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、ライトバックする値を `inst_pc + 4` にします。

▼リスト 3.80: 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。そのために、フェッチ先の変更が発生したことを表す信号 `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 3.81: control_hazard と control_hazard_pc_next の定義 (core.veryl)

```
    let control_hazard        : logic = inst_valid && inst_ctrl.is_jump;
    let control_hazard_pc_next: Addr  = alu_result;
TODO
```

`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 3.82: PC を変更する (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。実行し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 をリセットするようにします。

FIFO に、内容をリセットするための信号 `flush` を追加します。

▼リスト 3.83: ポートに flush を追加する (fifo.veryl)

```
    flush : input  logic    ,
```

▼リスト 3.84: flush が 1 のとき、FIFO を空にする (fifo.veryl)

```
    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` が `1` のときに、FIFO をリセットするようにします。

▼リスト 3.85: 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, ←追加
        ...
    );
```

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 3.86: 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:
fe1ff06f // 20: jal x0, -0x20 : 0にジャンプする
```

▼リスト 3.87: テストの実行 (一部省略)

```
$ 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
```

無条件ジャンプを正しく実行できていることを確認することができます。

3.12.2　条件分岐命令

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 フィールドで判別することができます。

▼表 3.11: 条件分岐命令と funct3

funct3	命令	演算
000	BEQ	==
001	BNE	!=
100	BLT	符号付き <=
101	BGE	符号付き >
110	BLTU	符号なし <=
111	BGEU	符号なし >

条件分岐命令の実装

まず、分岐するかどうかの判定を行うモジュールを作成します。

`src/brunit.veryl` を作成し、次のように記述します。

▼リスト 3.88: 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、`take` は `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 3.89: brunit 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 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 3.42)。

命令が条件分岐命令で、`brunit_take` が `1` のとき、次の PC を PC + 即値にするようにします。

▼リスト 3.90: 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 3.91: 分岐成立時の PC の設定 (core.veryl)

```
    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、分岐の成否を表示するようにします。デバッグ表示を行っている `always_ff` ブロック内に、次のプログラムを追加します。

▼リスト 3.92: デバッグ表示 (core.veryl)

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

簡単なテストを作成し、動作をテストします。

▼リスト 3.93: 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 3.94: テストの実行 (一部省略)

```
$ make build
$ make sim
$ obj_dir/sim src/sample_br.hex 15
#                     4
00000000 : 00100093
  reg[ 1] <= 00000001 ← 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 命令についてはテストできていませんが、後の章で紹介する riscv-tests でテストします。

これで RV32I の実装は終わりです。お疲れ様でした。

実装していない RV32I の命令について

本章ではメモリフェンス命令, ECALL, EBREAK 命令などを実装していません。これらの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すことができますが、例外や割り込み, ページングなどの機能がありません。このような機能は CSR を利用して提供されます。

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/mepc/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

4.2　CSRR(W|S|C)[I] 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` (`1110011`) です。この値を eei パッケージに定義します。

▼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: 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 ←追加
    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: 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}, ←追加
    default  : {InstType::X, F, F, F, F, F, F},
}, f3, f7};
```

上のコードでは、opcode が `OP_SYSTEM` な命令を、I 形式で、レジスタに結果を書き込み、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 の命令については、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 レジスタのデータを利用します。CSRRWI, CSRRSI,

CSRRCI 命令は、命令のビット中の rs1 にあたるビット列 (5 ビット) をゼロ拡張した値を利用します。それぞれの命令は funct3 で区別することができます (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 csrunit モジュールを作成します。

4.3.1　csrunit モジュールの作成

`src/csrunit.veryl` を作成し、次のように記述します。

▼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次のとおりです。

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。

このままの状態で、とりあえず、csrunit モジュールを core モジュールの中でインスタンス化し

▼表 4.3: csrunit 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	CSRR(W|S|C)[I] による CSR 読み込みの結果

ます。

▼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,
);
```

上のコードでは、結果の受け取りのために `csru_rdata` レジスタを作成し、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、csrunit の結果をレジスタにライトバックするようにします。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: 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: デバッグ用に rdata を表示するようにする (core.veryl)

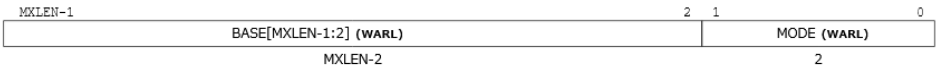
```
    if inst_ctrl.is_csr {
        $display("  csr rdata : %h", csru_rdata);
    }
```

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

4.3.2　mtvec レジスタの実装

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 レジスタとは何か?



▲図 4.1: mtvec のエンコーディング [6]

mtvec レジスタは、仕様書 [7] に定義されています。

mtvec は、MXLEN ビットの WARL なレジスタです。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 OK です。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るが、読み出すときには合法な値になるという認識で OK です。

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を言います[*1]。トラップが発生する時、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 0(Direct モード) になるようにしま

[*1] 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

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 レジスタを実装する

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: CsrAddr 型を定義する (csrunit.veryl)

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

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: mtvec レジスタの定義 (csrunit.veryl)

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

mtvec レジスタの 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。

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

```
// wmasks
const MTVEC_WMASK: UIntX = 'hffff_fffc;
```

次に、書き込むべきデータ `wdata` の生成と、mtvec レジスタの読み込みをします。

▼リスト 4.10: レジスタの読み込みと書き込みデータの作成 (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` に応じて CSR の値を割り当てます。`wdata` には、CSR に書き込むべきデータを割り当てます。CSR に書き込むべきデータは、書き込む命令 (CSRRW[I], CSRRS[I], CSRRC[I]) によって異なります。rs1 には rs1 の値か即値が格納されているため、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については、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記述します。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 4.11: 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 0 になるようにしています。

4.3.3　CSR のテスト

mtvec レジスタの書き込み、読み込みができることをテストします。

プロジェクトのフォルダに `test` ディレクトリを作成してください。`test/sample_csr.hex` を作成し、次のように記述します。

▼リスト 4.12: 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 `'b10111` を書き込んだ後、CSRRS 命令で mtvec の値を読み込んでいます。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んでいます。

シミュレータを実行し、結果を確かめます。

▼リスト 4.13: 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'b10111を書き込む
  itype     : 000010
  rs1[23]   : 00000000 ← CSRRWIなので、mtvecに'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 ←'b10111のMODE部分がマスクされて、'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、`00000014` が読み込まれることが確認できます。

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、CPU 上で動く権限 (特権, 機能) 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) しかないものとします。

▼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、例外が発生した時に、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。

例外が発生すると、CPU は mtvec にジャンプする前に、mepc に現在の PC を、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することが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　トラップの実装

それでは、ECALL 命令とトラッ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 4.14: mepc, 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,   ←追加
    MCAUSE = 12'h342, ←追加
}
```

次に、例外の原因を表現する型 `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 4.15: CsrCause 型の定義 (csrunit.veryl)

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

最後に、mepc, mcause の書き込みマスクを定義します。

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ので、mepc の下位 2 ビットは常に 0 になるようにします。

▼リスト 4.16: mepc, mcause の書き込みマスクの定義 (csrunit.veryl)

```
const MTVEC_WMASK : UIntX = 'hffff_fffc;
const MEPC_WMASK  : UIntX = 'hffff_fffc; ←追加
const MCAUSE_WMASK: UIntX = 'hffff_ffff; ←追加
```

mepc, mcause レジスタの実装

まず、mepc, 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 4.17: mepc, mcause レジスタの定義 (csrunit.veryl)

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

次に、mepc, mcause の読み込み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 4.18: mepc, 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 4.19: mepc, 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, mcause の書き込みを実装します。まず if_reset で値を 0 に初期化し、case 文に mepc, mcause の場合を追加します。

▼リスト 4.20: mepc, 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        : {}
                }
            }
        }
    }
}
```

例外を実装する

いよいよ ECALL 命令と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 4.21: 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        , ←追加
    ctrl        : input  InstCtrl    ,
    rd_addr     : input  logic   <5> , ←追加
    csr_addr    : input  logic   <12>,
    rs1         : input  UIntX       ,
    rdata       : output UIntX       ,
    raise_trap : output logic        , ←追加
    trap_vector: output Addr         , ←追加
) {
```

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。例外が発生した時、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 のアドレスを受け取ります。現在処理している命令が ECALL 命令かどうかを判定するために使います。

raise_trap

例外が発生する時、値を 1 にします。

trap_vector

例外が発生する時、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、ポートを接続します。

▼リスト 4.22: csrunit のポートの定義を変更する ① (core.veryl)

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

▼リスト 4.23: csrunit のポートの定義を変更する ② (core.veryl)

```
    inst csru: csrunit (
        clk                         ,
        rst                         ,
        valid   : inst_valid       ,
        pc      : inst_pc          , ←追加
```

```
        ctrl    : inst_ctrl       ,
        rd_addr                    , ←追加
        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,
        raise_trap : csru_raise_trap, ←追加
        trap_vector: csru_trap_vector,←追加
    );
```

次に、トラップするときに、トラップ先にジャンプするようにします。例外が発生する時、`csru_raise_trap` が `1` になり、 `csru_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、ジャンプや分岐命令の実装に利用したロジックを利用します。

`control_hazard` の条件に `csru_raise_trap` を追加し、トラップするときに `control_hazard_pc_next` を `csru_trap_vector` に設定します。

▼リスト 4.24: 例外の発生時にジャンプするようにする (core.veryl)

```
    assign control_hazard = inst_valid && (
        csru_raise_trap || ←追加
        inst_ctrl.is_jump ||
        inst_is_br(inst_ctrl) && brunit_take
    );
    assign control_hazard_pc_next = if csru_raise_trap {
        csru_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
    };
```

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。これを判定するための変数を作成します。

▼リスト 4.25: 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;
```

まず、例外が発生するかどうかを示す `raise_expt` と、例外が発生の原因を示す `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` は定数になっています。

▼リスト 4.26: 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、トラップ処理を記述します。トラップが発生する時、mepc レジスタに pc を、mcause レジスタにトラップの発生原因を格納します。

▼リスト 4.27: (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 命令のテスト

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 4.28: デバッグ用の表示を追加する (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);
    }
```

それでは簡単なテストを記述します。

CSRRW 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, mepc レジスタの値を読み取ります。

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

▼リスト 4.29: 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
```

シミュレータを実行し、結果を確かめます。

▼リスト 4.30: 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 ← 0x10(=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が確認できました。

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[*2]は、トラップ先からトラップ元に戻るための命令です。具体的には、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[*3]。

MRET 命令は、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。

[*2] MRET 命令は Volume II の 3.3.2. Trap-Return Instructions に定義されています
[*3] 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

4.5.1　MRET 命令を実装する

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ための変数 `is_mret` を作成します。MRET 命令は、上位 12 ビットが `001100000010` , rs1 は 0, funct3 は 0, rd は 0 です。

▼リスト 4.31: 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.f›
›unct3 == 0 && rd_addr == 0;
```

次に、MRET 命令が供給されているときに mepc にジャンプするようにするロジックを作成します。ジャンプするためのロジックは、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 4.32: 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
    };
```

トラップが発生しているかどうかの条件 `raise_mret` に `is_mret` を追加し、トラップ先を条件によって変更します。

ここで、 `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、将来的に MRET 命令によって例外が発生することがあるからです。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。

4.5.2　MRET 命令のテスト

MRET 命令が正しく動作するかテストします。

mepc に値を設定してから MRET 命令を実行し、mepc にジャンプするかどうかを確認します。

▼リスト 4.33: sample_mret.hex

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

▼リスト 4.34: 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が確認できます。

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

前の章で、RV32I の CPU を実装しました。簡単なテストを作成して操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、次の URL からソースコードをダウンロードすることができます。

riscv-software-src/riscv-tests : https://github.com/riscv-software-src/riscv-tests

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ることができます。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。

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 以下にコピーしてください。

5.2.1 riscv-tests のビルド

riscv-tests を clone します。

▼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、今のところ、CPU はアドレス `0x00000000` から実行を開始するため、リンカにわたす設定ファイル `env/p/link.ld` を変更します。

▼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: 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 をビルドすることができましたが、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。

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: core/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み
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 ファイルから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: ELF ファイルを変換する

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

objcopy で生成された bin 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: 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 4 {} > {}.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 `rv32ui-p-add.dump` を読んでみます。`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
 ...    ←レジスタ値のゼロ初期化
  c8:   00000f93                li      t6,0
 ...    ←↓mtvecにtrap_vectorのアドレスを書き込む
 130:   00000297                auipc   t0,0x0
 134:   ed428293                addi    t0,t0,-300 # 4 <trap_vector>
 138:   30529073                csrw    mtvec,t0
 ...    ←↓mepcにtest_2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
 ...    ←↓mretを実行し、mepcのアドレス=test_2にジャンプする
 188:   30200073                mret

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
 ... ←↓gpを1以外の値にする
```

```
 674:   00119193                sll     gp,gp,0x1
 678:   0011e193                or      gp,gp,1
 ...
 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
```

riscv-tests は、基本的に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 に飛ぶ。最後まで正常に実行できたら pass に飛ぶ。
4. fail, pass：テストの成否をレジスタに書き込み、trap_vector に飛ぶ
5. trap_vector：write_tohost に飛ぶ
6. write_tohost：テスト結果をメモリに書き込む。ここでループする

_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、リンカ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

テストを実行する場合、テストが終了したことを検知し、それが成功か失敗かどうかを報告する必要があります。

riscv-tests はテストが終了したことを示すために、`.tohost` に値を書き込みます。この値が 1

のとき、riscv-tests が正常に終了したことを示します。それ以外の時は、riscv-tests が失敗したことを示します。

riscv-tests 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` に値が書き込まれたら実行を終了します。

▼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
    const RISCVTESTS_TOHOST_ADDR: Addr = 32'h1000;
    always_ff {
        if membus.valid && membus.wen == 1 && membus.addr == RISCVTESTS_TOHOST_ADDR {
            if membus.wdata == 1 {
                $display("riscv-tests success!");
            } else {
                $display("riscv-tests failed!");
                $error  ("wdata : %h", membus.wdata);
            }
            $finish();
        }
    }
```

テストが失敗した場合、つまり 1 以外の値が書き込まれた場合、`$error` システムタスクを実行します。これにより、テスト失敗時のシミュレータの終了コードが 1 になります。

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。

シミュレータを実行し、正常に動くことを確認します。

▼リスト 5.10: ADD 命令の riscv-tests を実行する

```
$ make build
$ make sim
$ ./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!
- /home/kanataso/Documents/bluecore/core/src/top.sv:26: Verilog $finish
- /home/kanataso/Documents/bluecore/core/src/top.sv:26: Second verilog $finish, exiting
```

`riscv-tests success!` と表示され、テストが正常終了しました[*1]。

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。本書では、自動でテストを実行し、その結果を報告するプログラムを作成します。

`test/test.py` を作成し、次のように記述します。

▼リスト 5.11: test.py

```
import argparse
import os
import subprocess

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

[*1] 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...

```
            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。

今回は RV32I のテストを実行したいので、riscv-tests の RV32I 向けのテストの接頭辞である rv32ui-p-引数に指定します。

この Python プログラムには、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するようにします。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。

それでは、RV32I のテストを実行しましょう。

▼リスト 5.12: rv32ui-p から始まるテストを実行する

```
$ python3 test.py ../obj_dir/sim share rv32ui-p- -r
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-srli.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 個のテストにパスしました。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アライン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。これについて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、動作が少し変わる命令が存在します (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	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 `rv64i-p-` から始まるテストです。命令を実装するたびにテストを実行することで、命令が正しく実行できていることを確認します。

6.1　XLEN の変更

eei パッケージに定義している XLEN を 64 に変更します。RV64I になっても命令の幅 (`ILEN`) は 32 ビットのままです。

▼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] 命令の対応

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: 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];
```

I 形式の命令 (SLLI, SRLI, SRAI) のときは即値、R 形式の命令 (SLL, SRL, SRA) のときはレジスタの下位 6 ビットが利用されるようになります。

6.1.2　LUI, AUIPC 命令の対応

RV32I では、LUI 命令は 32 ビットの即値をそのまま保存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保存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。よって、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 の対応

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: CSR の書き込みマスクの幅を広げる (csrunit.veryl)

```
    const MTVEC_WMASK : UIntX = 'hffff_ffff_ffff_fffc;
    const MEPC_WMASK  : UIntX = 'hffff_ffff_ffff_fffc;
    const MCAUSE_WMASK: UIntX = 'hffff_ffff_ffff_ffff;
```

6.1.4　LW 命令の対応

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: 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]},
```

6.1.5　riscv-tests でテストする

TODO

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, ADDIW 命令が追加されます。同様に、SUB 命令は 64 ビッド単位の演算になり、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 [3]

ADDW, SUBW 命令は R 形式で、opcode は `OP-32` (`0111011`) です。ADDIW 命令は I 形式で、opcode は `OP-IMM-32` (`0011011`) です。

まず、eei パッケージに opcode の定数を定義します。

▼リスト 6.6: 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 6.7: 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 ←追加
    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。これでデコードは完了です。

▼リスト 6.8: 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},
    OP_AUIPC    : {InstType::U, T, F, F, F, F, F, F},
    OP_JAL      : {InstType::J, T, F, F, F, T, F, F},
    OP_JALR     : {InstType::I, T, F, F, F, T, F, F},
    OP_BRANCH   : {InstType::B, F, F, F, F, F, F, F},
    OP_LOAD     : {InstType::I, T, F, F, F, F, T, F},
    OP_STORE    : {InstType::S, F, F, F, F, F, F, F},
    OP_OP       : {InstType::R, T, F, T, F, F, F, F},
    OP_OP_IMM   : {InstType::I, T, F, T, F, F, F, F},
    OP_OP_32    : {InstType::R, T, F, T, T, F, F, F}, ←追加
    OP_OP_IMM_32: {InstType::I, T, F, T, T, F, F, F}, ←追加
    OP_SYSTEM   : {InstType::I, T, F, F, F, F, F, T},
    default     : {InstType::X, F, F, F, F, F, F, F},
}, f3, f7};
```

▼リスト 6.9: OP-32, 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, SUB を行うようにします。
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 6.10: 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。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を、 `0` なら `value64` を返します。

▼リスト 6.11: 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 6.12: 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

TODO

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 [3]

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, 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、シフト演算の結果を生成します。

▼リスト 6.13: 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` 関数で選択するようにします。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 6.14: 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

TODO

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 [3]

LWU 命令は I 形式で、opcode は `LOAD` です。ロード, ストア命令は funct3 によって区別することができます。LWU 命令の funct3 は `110` 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換します。32 ビットを `rdata` に割り当てるとき、`sext` によって符号拡張かゼロで拡張するかを選択するようにします。

▼リスト 6.15: 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

TODO

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, 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 `011` 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 (`eei::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になってしまいます。これを 1 回のメモリアクセスで済ませるために、データの幅を 32 ビットから 64 ビットに広げます。

▼リスト 6.16: MEM_DATA_WIDTH を 64 ビットに変更する (eei.veryl)

```
    const MEM_DATA_WIDTH: u32 = 64;
```

6.5.2　命令フェッチの対応

`XLEN` , `eei::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、top モジュールの `i_membus.rdata` の幅は 32 ビット、 `membus.rdata` は 64 ビットになり、 `i_membus.rdata` に `membus.rdata` の下位 32 ビットが接続されます。よって、今のコードのままだとアドレスの下位 3 ビットが `100` (=4) であっても、下位 3 ビットが `000` (=0) の命令が `i_membus.rdata` に割り当てられてしまいます。

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。PC[2] が 0 のときは下位 32 ビット、1 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のアドレスをレジスタに保存します。

▼リスト 6.17: アドレスを保存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 6.18: レジスタに命令フェッチの要求アドレスを保存する (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 6.19: 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　ストア命令を実装する

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。

▼リスト 6.20: 書き込みデータの変更 (memunit.veryl)

```
    req_wdata = rs2 << {addr[2:0], 3'b0};
```

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 ビット分だけシフトするようにします。

▼リスト 6.21: 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'b00110000,
            2      : @<b<|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,
    };
```

書き込みマスクは 8 ビットに拡張されます。それに伴い、アドレスの下位 2 ビットではなく下位 3 ビットで選択するようにするようにします。

6.5.4　ロード命令の実装

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 ビットで選択するようにします。

▼リスト 6.22: 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

TODO

6.6　RV64I のテスト

TODO

第 7 章
CPU のパイプライン処理化

これまでの章では、同時に 1 つの命令のみを実行する 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 の性能を考える

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について考えます。

7.1.1 CPU の性能指標

プログラムの実行速度を比較する時、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、簡単に、次のような式 [8] で表すことができます。

$$CPU \text{ 時間} = \frac{\text{実行命令数} \times CPI}{\text{クロック周波数}}$$

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・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

今のところ、CPU に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。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。命令の割合について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パスのことです。

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、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。

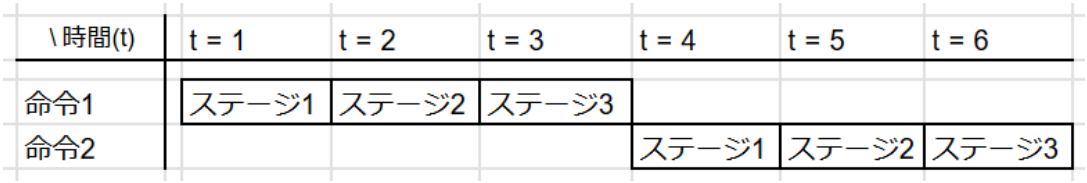
今のところ、CPU は計算命令を 1 クロック (**シングルサイクル**) で実行します。例えば ADD 命令を実行する時、FIFO に保存された ADD 命令をデコードし、命令のビット列をもとにレジスタのデータを選択し、ALU で足し算を実行し、その結果をレジスタにライトバックします。これらを 1 クロックで実行するということは、命令が保存されている 32 ビットのレジスタとレジスタファイルを入力に、64 ビットの ADD 演算の結果を出力する組み合わせ回路が存在するということです。この回路は大変に段数の深い組み合わせ回路を必要とし、長いクリティカルパスを生成する原因になります。

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 といいます。

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ることができます。

図 7.1 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

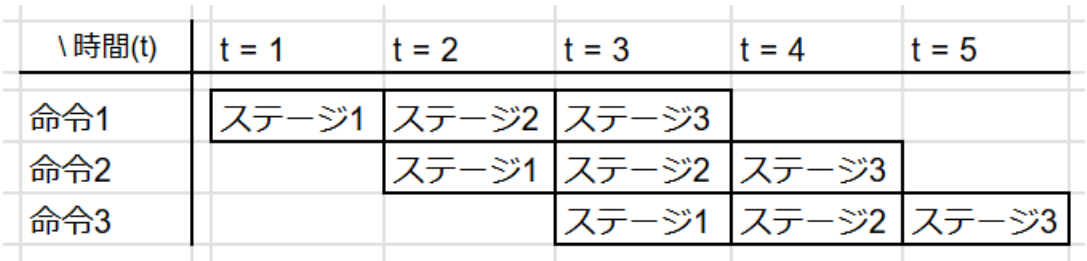
[*1] 他の方法として、関数呼び出しやループを CPU 側で検知して結果を保存, 利用することで実行命令数を減らす手法があります。この手法についてはずっと後の章で検討します。



▲図 7.1: 命令の実行 (マルチサイクル)

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。これは、命令の処理を車の組立のように流れ作業で行うことで緩和することができます。このような処理のことを、**パイプライン処理**と言います。



▲図 7.2: 命令の実行 (パイプライン処理)

本章では、CPU をパイプライン処理化することで、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について考える

では、具体的に処理をどのようなステージに分割し、パイプライン処理を実現すればいいでしょうか? これを考えるために、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。

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) を構成することができ

[*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 になりません

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フラッシュして、ジャンプ先を IF ステージに伝えます。メモリ, CSR の読み込み結果等を 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。

IF, ID, EX, MEM, WB の 5 段の構成を、5 段パイプライン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?

今のところ CPU には ECALL 命令による例外しか存在しないため、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。

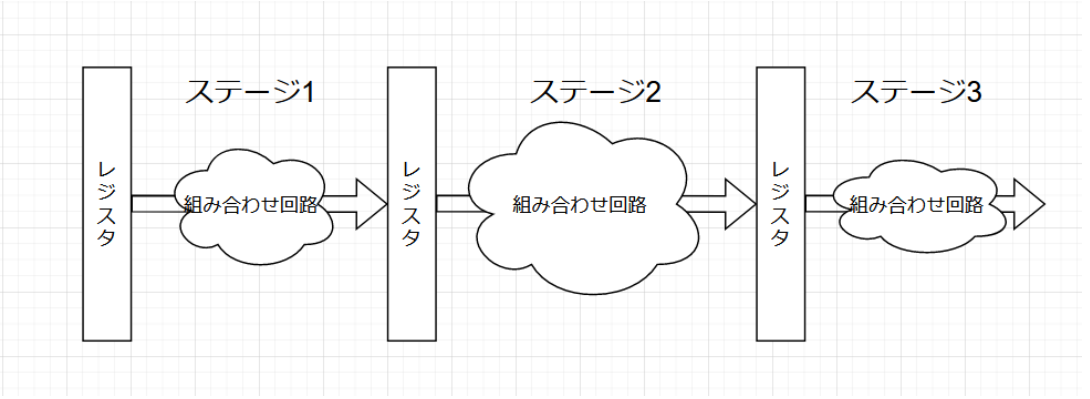
メモリアクセスに起因する例外が発生するのは MEM ステージです。このとき、EX ステージで CSR の処理を行っていて、EX ステージに存在する命令が mtvec レジスタに書き込む CSRRW 命令だった場合、本来は MEM ステージで発生した例外によって実行されないはずである CSRRW 命令によって、既に mtvec レジスタが書き換えられているかもしれません。これを復元する処理を書くことはできますが、MEM ステージ以降で CSR を処理することでもこの事態を回避できるため、無駄な複雑性を導入しないために、MEM ステージで CSR を処理しています。

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 をパイプライン処理化します。

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**) を用意してあげる必要があります。



▲図 7.3: 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、既存のレジスタの名前を変更します。

現状の 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 ステージに該当します。名前はこのま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、どのステージのレジスタ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、とりあえず、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しょう。

▼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   ; // 命令が今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   ;
```

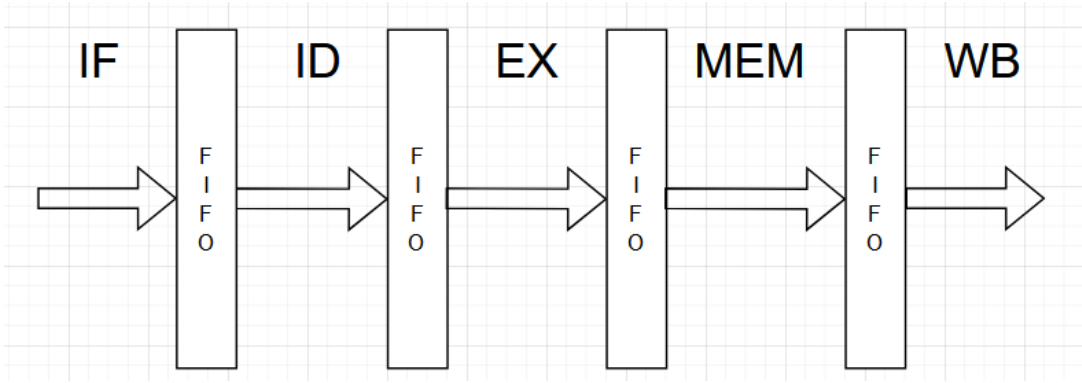
`inst_valid` , `inst_is_new` , `inst_pc` , `inst_bits` , `inst_ctrl` , `inst_imm` の名前をリスト 7.1 のように変更します。定義だけではなく、変数を使用しているところもすべて変更してください。

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, 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 7.4)。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 7.4: 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。

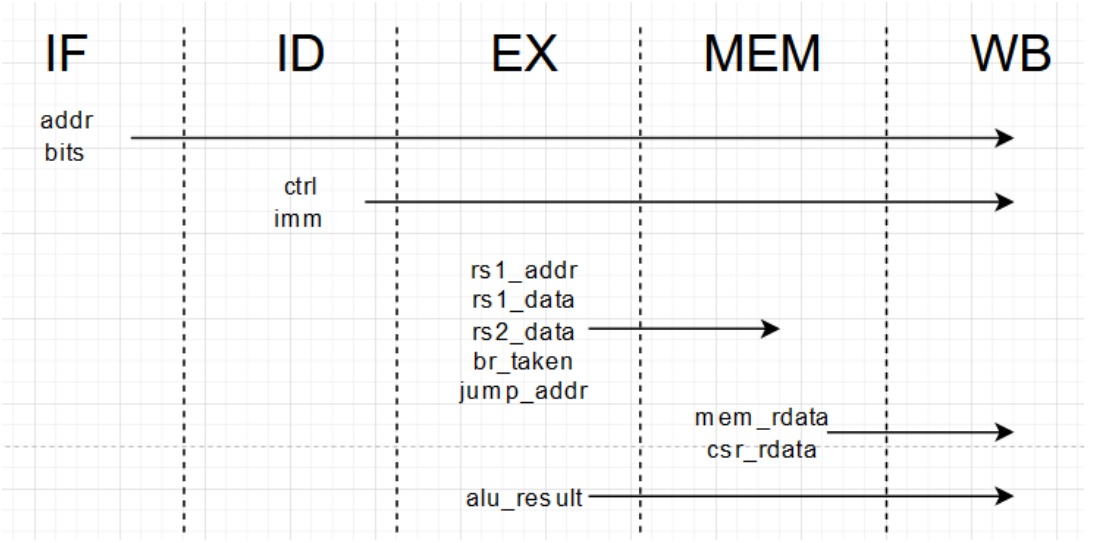
構造体の定義

まず、FIFO に格納するデータの型を定義します。それぞれのメンバーが存在する区間は図 7.5 の通りです。

▼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。



▲図 7.5: メンバーの生存区間

▼リスト 7.3: EX -> NEN 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 : logic       ,
}
```

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と受け取ります。整数演算命令の時、レジスタのデータを読み取り、ALU で計算します。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, rs2 のデータを利用するため、演算の結果とともに MEM ステージに渡します。

▼リスト 7.4: 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。

WB ステージでは、命令がライトバックする命令の時、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。

構造体のメンバーの生存区間が図 7.5 のようになっている理由が、なんとなく分かったでしょうか?

FIFO のインスタンス化

FIFO をインスタンス化します。`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 のデータの個数は 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` が `0` になっていることに注意してください。

▼リスト 7.5: 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        ,
    ) (
```

[*3] FIFO のデータ個数は 2 ** WIDTH - 1 です

```
        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。

ステージの区間を示すために、リスト 7.6 のようなコメントを挿入すると良いです。ID, EX, MEM, WB ステージを実装する時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 7.6: 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` , `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ます。そのため、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 ステージに渡す必要があります。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 7.7: 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付けることができるとき (`exq_wready`)、ID ステージを完了して EX ステージに処理を進めることができます。このロジックは、

`if_fifo_rready` に `exq_wready` を割り当てることで実現できます。

最後に、命令が今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 7.8: id_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、整数演算命令の時は ALU で計算し、分岐命令の時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 7.9: 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 7.10: 変数名の変更対応 (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.itype {
            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には、`memq_wdata` にデータを割り当てます。

▼リスト 7.11: 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_wvalid;
        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を割り当てます。これを利用することで、MEM ステージで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付けることができるとき (`memq_wready`)、EX ステージを完了して MEM ステージに処理を進めることができます。このロジックは、 `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, またはトラップが発生する時、次の命令のアドレスを変更します。

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。MEM ステージでは、csrunit モジュールに、命令が今のクロックで MEM ステージに供給されたかどうかの情報を渡す必要があります。そのため、変数 `mem_is_new` を定義しています。

▼リスト 7.12: 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、もともと `id_is_new` の更新に利用していたロジックを利用します。

▼リスト 7.13: 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、ポートなどに割り当てている変数名を変更します。

▼リスト 7.14: 変数名の変更対応 (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              ,
        rdaddr  : 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` には、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するようにします。

▼リスト 7.15: ジャンプ判定処理 (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
```

```
    };
```

`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

MEM ステージにある命令は、memunit が処理中ではなく (`!memy_stall`)、WB ステージが命令を受け付けることができるとき (`wbq_wready`)、MEM ステージを完了して WB ステージに処理を進めることができます。このロジックについては、 `memq_rready` と `wbq_wvalid` を確認してください。

▼リスト 7.16: 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;
    }
```

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書き込みます。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、MEM 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 7.17: 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の名前を変更します。変数の名前には `wbs_` をつけます。

▼リスト 7.18: 変数名の変更対応 (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ないため、 `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 7.19: 命令を FIFO から取り出す (core.veryl)

```
    always_comb {
        // WB -> END
        wbq_rready = 1;
    }
```

IF, ID, EX, MEM, WB ステージを作成できたので、5 段パイプラインの CPU は完成です。

7.2.8　デバッグ用に情報を表示する

今までは同時に 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。デバッグ用の表示を変更しておきましょう。

リスト 7.20 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 7.20: 各ステージの情報を表示する (core.veryl)

```
    //////////////////////////////// DEBUG ////////////////////////////////
    var clock_count: u64;

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

            $display("\n");
            $display("# %d", clock_count);

            $display("ID ------");
            if ids_valid {
                $display("  %h : %h", if_fifo_rdata.addr, if_fifo_rdata.bits);
                $display("  itype : %b", ids_ctrl.itype);
                $display("  imm   : %h", ids_imm);
```

```
            }
            $display("EX -----");
            if exq_rvalid {
                $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 memq_rvalid {
                $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);
                }
            }
            $display("WB ----");
            if memq_rvalid {
                $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　パイプライン処理のテスト

それでは、riscv-tests を実行してみましょう。RV32I, RV64I 向けのテストを実行します。

▼リスト 7.21: riscv-tests の実行

おや?テストにパスしません。一体何が起きているのでしょうか?

7.3　データハザードの対処

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、正しく命令を実行することができません。

7.3.1　正しく動かないプログラム

例えば、リスト 7.22 のようなプログラムは正しく動きません。`test/dh.hex` として、プログラ

ムを記述します。

▼リスト 7.22: 正しく動かないプログラムの例 (test/dh.hex)

```
```

7.3.2　データ依存

7.3.3　データ依存の対処

7.3.4　パイプライン処理をテストする

それでは、`test/dh.hex` を実行して、正しく動くことを確認します。

▼リスト 7.23: test/dh.hex が正しく動くことを確認する

```
```

riscv-tests も実行しましょう。

▼リスト 7.24: riscv-tests を実行する

```
```

テストにパスすることを確認できました。

第 8 章
CPU を合成する

TangMega138K と PYNQ-Z1
TODO

あとがき / おわりに

いかがだったでしょうか。質問は GitHub の issue にお願いします。

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。本書の pdf, web 版は無料で配布されており、`https://github.com/nananapo/veryl-riscv-book` でダウンロード, 閲覧することができます。

続きが気になったり、誤植を見つけた場合は、GitHub をご確認ください。

著者紹介

kanataso (kanapipopipo@X, nananapo@GitHub)
いつの間にか自作 CPU の沼に沈んでいました。
自ら外堀を埋めてしまい、もう抜け出せそうにありません。
計算機と法律に興味があります

参考文献

[1]
坂井 修一, 論理回路入門, 培風館
[2]
The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411 2.3. Immediate Encoding Variants
[3]
The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411 37. RV32/64G Instruction Set Listings
[4]
The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411 2.4. Integer Computational Instructions
[5]
The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411 2.5. Control Transfer Instructions
[6]
The RISC-V Instruction Set Manual Volume II: Privileged Architecture version 20240411 Figure 10. Encoding of mtvec MODE field.
[7]
The RISC-V Instruction Set Manual Volume II: Privileged Architecture version 20240411 3.1.7. Machine Trap-Vector Base-Address Register
[8]
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] 〜ハードウエアとソフトウエアのインタフェース〜, 日経 BP

Veryl で作る CPU

基本編 (の第 I 部)

2024 年 11 月 3 日　 ver 1.0 （技術書典 17）

著　者　kanataso
発行者　kanataso
連絡先　kanapipopipo@X
印刷所　日光企画
