

# Veryl で作る CPU

— 基本編 —

[著] 阿部奏太

2025 年 6 月 1 日 基本編 第 II 部、第 III 部 ver 1.0

## ■免責

本書は情報の提供のみを目的としています。

本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

## ■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。

また本書では、™、®、©などのマークは省略しています。

# まえがき

こんにちは! あなたは CPU を自作したことがありますか? 自作したことがあってもなくても大歓迎、この本は CPU 自作の面白さを世に広めるために執筆されました。

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

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

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

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

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

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

## 本書を読むとわかるここと

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

## 対象読者

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

## 必要な知識

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

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

## 本書のソースコード / 問い合わせ先

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

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

## CPU の自作

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

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

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

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

CPU のテストはシミュレータと FPGA で行います。本書では、Tang Nano 9K と PYNQ-Z1 という FPGA を利用します。FPGA を持っていると、自作 CPU によって LED を制御したり、手持ちのパソコンと直接通信したりして楽しむことができます。

## RISC-V

**RISC-V** は、カリフォルニア大学バークレー校で開発された ISA です。仕様書の初版は 2011 年に公開されました。ISA としての歴史はまだ浅いですが、仕様が広く公開されていてカスタマイズ可能であるという特徴もあって、着実に広がりつつあります。

インターネット上には多くの RISC-V の実装が公開されています。例として、RocketChip<sup>2</sup> (Chisel による実装)、Shakti<sup>3</sup> (Bluespec SystemVerilog による実装)、RSD<sup>4</sup> (SystemVerilog による実装) が挙げられます。

本書では、RISC-V のバージョン RISC-V ISA Manual, version 20240411 を利用します。RISC-V の最新の仕様は、GitHub の riscv/riscv-isa-manual<sup>5</sup> で確認できます。

RISC-V には、基本整数命令セットとして RV32I、RV64I、RV32E、RV64E<sup>6</sup> が定義されています。RV の後ろにつく数字はレジスタの長さが何ビットかです。基本整数命令セットには最低限の命令しか定義されていません。それ以外のかけ算や割り算、不可分操作などの命令や機能は拡張

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

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

<sup>3</sup> <https://shakti.org.in/>

<sup>4</sup> <https://github.com/rsd-devel/rsd>

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

<sup>6</sup> RV128I もありますが、まだ Draft 段階（準備中）です

として定義されています。

どの拡張を実装しているかを示す文字列では、例えばかけ算と割り算、不可分操作ができる RV32I の CPU は `RV32IMA` と表現されます。本書では、まず、`RV32I` の CPU を実装します。これを、OS を実行できる程度までに進化させることを目標に実装を進めます。

## 本書の構成

本シリーズ(基本編)では、次のように CPU を実装していきます。

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

## 凡例

プログラムコードの差分を表示する場合は、追加されたコードを太字で、削除されたコードを取り消し線で表します。ただし、リスト内のコードが全て新しく追加されるときは太字を利用しません。コードを置き換えるときは太字で示し、削除されたコードを示さない場合もあります。

```
print("Hello, world!\n");      ←取り消し線は削除したコード
print("Hello, "+name+"!\n");  ←太字は追加したコード
```

長い行が右端で折り返されると、折り返されたことを表す小さな記号がつきます(pdf版のみ)。

```
123456789_123456789_123456789_123456789_123456789_123456789_123456789_123456789_123456789_>
_9_123456789_123456789_123456789_123456789_123456789_123456789_123456789_123456789_123456789_
```

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

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

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

# 目次

まえがき	i
<b>第1部 RV32I/RV64I の実装</b>	<b>1</b>
<b>第1章 環境構築</b>	<b>2</b>
1.1 Veryl . . . . .	2
1.2 Verilator . . . . .	4
1.3 riscv-gnu-toolchain . . . . .	4
<b>第2章 ハードウェア記述言語 Veryl</b>	<b>5</b>
2.1 ハードウェア記述言語 . . . . .	5
2.1.1 論理回路の構成 . . . . .	5
2.1.2 ハードウェア記述言語 . . . . .	6
2.1.3 Veryl . . . . .	7
2.2 Veryl の基本文法、機能 . . . . .	8
2.2.1 コメント . . . . .	8
2.2.2 値、リテラル . . . . .	9
2.2.3 module . . . . .	10
2.2.4 ユーザー定義型 . . . . .	16
2.2.5 式、文、宣言 . . . . .	18
2.2.6 interface . . . . .	24
2.2.7 package . . . . .	25
2.2.8 ジェネリクス . . . . .	26
2.2.9 その他の機能、文 . . . . .	27
<b>第3章 RV32I の実装</b>	<b>30</b>
3.1 CPUは何をやっているのか? . . . . .	30
3.2 プロジェクトの作成 . . . . .	32
3.3 定数の定義 . . . . .	33
3.4 メモリ . . . . .	34
3.4.1 メモリのインターフェースを定義する . . . . .	34
3.4.2 メモリモジュールを実装する . . . . .	35
3.4.3 メモリの初期化、環境変数の読み込み . . . . .	37
3.5 最上位モジュールの作成 . . . . .	37
3.6 命令フェッチ . . . . .	39

---

3.6.1	命令フェッチを実装する . . . . .	39
3.6.2	memory モジュールと core モジュールを接続する . . . . .	40
3.6.3	命令フェッチをテストする . . . . .	41
3.6.4	フェッチした命令を FIFO に格納する . . . . .	46
3.7	命令のデコードと即値の生成 . . . . .	51
3.7.1	デコード用の定数と型を定義する . . . . .	53
3.7.2	制御フラグと即値を生成する . . . . .	54
3.7.3	デコーダをインスタンス化する . . . . .	56
3.8	レジスタの定義と読み込み . . . . .	57
3.8.1	レジスタファイルを定義する . . . . .	57
3.8.2	レジスタの値を読み込む . . . . .	58
3.9	ALU による計算の実装 . . . . .	60
3.9.1	ALU モジュールを作成する . . . . .	60
3.9.2	ALU モジュールをインスタンス化する . . . . .	61
3.9.3	ALU モジュールをテストする . . . . .	63
3.10	レジスタに結果を書き込む . . . . .	65
3.10.1	ライトバック処理を実装する . . . . .	65
3.10.2	ライトバック処理をテストする . . . . .	66
3.11	ロード命令とストア命令の実装 . . . . .	67
3.11.1	LW、SW 命令を実装する . . . . .	67
3.11.2	LB、LBU、LH、LHU 命令を実装する . . . . .	76
3.11.3	SB、SH 命令を実装する . . . . .	77
3.11.4	LB、LBU、LH、LHU、SB、SH 命令をテストする . . . . .	81
3.12	ジャンプ命令、分岐命令の実装 . . . . .	82
3.12.1	JAL、JALR 命令を実装する . . . . .	82
3.12.2	条件分岐命令を実装する . . . . .	86
<b>第 4 章</b>	<b>Zicsr 拡張の実装</b>	<b>90</b>
4.1	CSR とは何か? . . . . .	90
4.2	CSR 命令のデコード . . . . .	91
4.3	csruni モジュールの実装 . . . . .	92
4.3.1	csruni モジュールを作成する . . . . .	92
4.3.2	mtvec レジスタを実装する . . . . .	94
4.3.3	csruni モジュールをテストする . . . . .	96
4.4	ECALL 命令の実装 . . . . .	97
4.4.1	ECALL 命令とは何か? . . . . .	97
4.4.2	トラップを実装する . . . . .	98
4.4.3	ECALL 命令をテストする . . . . .	102
4.5	MRET 命令の実装 . . . . .	104
4.5.1	MRET 命令を実装する . . . . .	104

---

4.5.2	MRET 命令をテストする . . . . .	105
<b>第 5 章</b>	<b>riscv-tests によるテスト</b>	<b>106</b>
5.1	riscv-tests とは何か? . . . . .	106
5.2	riscv-tests のビルド . . . . .	106
5.2.1	riscv-tests をビルドする . . . . .	106
5.2.2	成果物を\$readmemh で読み込める形式に変換する . . . . .	107
5.3	テスト内容の確認 . . . . .	109
5.4	テストの終了検知 . . . . .	111
5.5	テストの実行 . . . . .	112
5.6	複数のテストの自動実行 . . . . .	112
<b>第 6 章</b>	<b>RV64I の実装</b>	<b>117</b>
6.1	XLEN の変更 . . . . .	118
6.1.1	SLL[I]、SRL[I]、SRA[I] 命令を変更する . . . . .	118
6.1.2	LUI、AUIPC 命令を変更する . . . . .	118
6.1.3	CSR を変更する . . . . .	118
6.1.4	LW 命令を変更する . . . . .	119
6.1.5	riscv-tests でテストする . . . . .	119
6.2	ADD[I]W、SUBW 命令の実装 . . . . .	121
6.2.1	ADD[I]W、SUBW 命令をデコードする . . . . .	121
6.2.2	ALU に ADDW、SUBW を実装する . . . . .	122
6.2.3	ADD[I]W、SUBW 命令をテストする . . . . .	123
6.3	SLL[I]W、SRL[I]W、SRA[I]W 命令の実装 . . . . .	124
6.3.1	SLL[I]W、SRL[I]W、SRA[I]W 命令をテストする . . . . .	124
6.4	LWU 命令の実装 . . . . .	125
6.4.1	LWU 命令をテストする . . . . .	126
6.5	LD、SD 命令の実装 . . . . .	126
6.5.1	メモリの幅を広げる . . . . .	126
6.5.2	命令フェッチ処理を修正する . . . . .	126
6.5.3	SD 命令を実装する . . . . .	127
6.5.4	LD 命令を実装する . . . . .	128
6.5.5	LD、SD 命令をテストする . . . . .	129
<b>第 7 章</b>	<b>CPU のパイプライン化</b>	<b>130</b>
7.1	CPU の速度 . . . . .	130
7.1.1	CPU の性能を考える . . . . .	130
7.1.2	実行速度を上げる方法を考える . . . . .	131
7.1.3	パイプライン処理のステージを考える . . . . .	132
7.2	パイプライン処理の実装 . . . . .	134

---

7.2.1	ステージに分割する準備をする . . . . .	134
7.2.2	FIFO を作成する . . . . .	135
7.2.3	IF ステージを実装する . . . . .	138
7.2.4	ID ステージを実装する . . . . .	139
7.2.5	EX ステージを実装する . . . . .	139
7.2.6	MEM ステージを実装する . . . . .	141
7.2.7	WB ステージを実装する . . . . .	144
7.2.8	デバッグのために情報を表示する . . . . .	145
7.2.9	パイプライン処理をテストする . . . . .	146
7.3	データ依存の対処 . . . . .	146
7.3.1	正しく動かないプログラムを確認する . . . . .	146
7.3.2	データ依存とは何か？ . . . . .	147
7.3.3	データ依存に対処する . . . . .	148
7.3.4	パイプライン処理をテストする . . . . .	149

**第 II 部 RV64IMAC の実装****151**

<b>第 8 章</b>	<b>M 拡張の実装</b>	<b>152</b>
8.1	概要 . . . . .	152
8.2	命令のデコード . . . . .	154
8.3	muldivunit モジュールの実装 . . . . .	155
8.3.1	muldivunit モジュールを作成する . . . . .	155
8.3.2	EX ステージを変更する . . . . .	156
8.4	符号無しの乗算器の実装 . . . . .	158
8.4.1	mulunit モジュールを実装する . . . . .	158
8.4.2	mulunit モジュールをインスタンス化する . . . . .	160
8.5	MULHU 命令の実装 . . . . .	161
8.6	MUL、MULH 命令の実装 . . . . .	161
8.6.1	符号付き乗算を符号無し乗算器で実現する . . . . .	161
8.6.2	符号付き乗算を実装する . . . . .	162
8.6.3	MULHSU 命令の実装 . . . . .	163
8.6.4	MULW 命令の実装 . . . . .	164
8.7	符号無し除算の実装 . . . . .	166
8.7.1	divunit モジュールを実装する . . . . .	166
8.7.2	divunit モジュールをインスタンス化する . . . . .	169
8.8	DIVU、REMU 命令の実装 . . . . .	169
8.9	DIV、REM 命令の実装 . . . . .	170
8.9.1	符号付き除算を符号無し除算器で実現する . . . . .	170
8.9.2	符号付き除算を実装する . . . . .	170
8.10	DIVW、DIVUW、REMW、REMUW 命令の実装 . . . . .	172

---

<b>第 9 章 例外の実装</b>	<b>174</b>
9.1 例外とは何か? . . . . .	174
9.2 例外情報の伝達 . . . . .	175
9.2.1 Environment call from M-mode 例外を IF ステージで処理する . . . . .	175
9.2.2 mtval レジスタを実装する . . . . .	177
9.3 Breakpoint 例外の実装 . . . . .	179
9.4 Illegal instruction 例外の実装 . . . . .	180
9.4.1 不正な命令ビット列で例外を起こす . . . . .	180
9.4.2 読み込み専用の CSR への書き込みで例外を起こす . . . . .	182
9.5 命令アドレスのミスアライン例外 . . . . .	185
9.6 ロードストア命令のミスアライン例外 . . . . .	186
<b>第 10 章 Memory-mapped I/O の実装</b>	<b>188</b>
10.1 Memory-mapped I/O とは何か? . . . . .	188
10.2 定数の定義 . . . . .	189
10.3 mmio_controller モジュールの作成 . . . . .	192
10.4 RAM の接続 . . . . .	196
10.4.1 mmio_controller モジュールに RAM を追加する . . . . .	196
10.4.2 RAM と mmio_controller モジュールを接続する . . . . .	198
10.4.3 PC の初期値の変更 . . . . .	200
10.5 ROM の実装 . . . . .	201
10.5.1 mmio_controller モジュールに ROM を追加する . . . . .	201
10.5.2 ROM の初期値のパラメータを作成する . . . . .	203
10.5.3 ROM と mmio_controller モジュールを接続する . . . . .	204
10.5.4 ROM から RAM にジャンプする . . . . .	206
10.6 デバッグ用の入出力デバイスの実装 . . . . .	206
10.6.1 デバイスのアドレスを設定する . . . . .	207
10.6.2 mmio_controller モジュールにデバイスを追加する . . . . .	207
10.6.3 出力を実装する . . . . .	209
10.6.4 出力をテストする . . . . .	210
10.6.5 riscv-tests に対応する . . . . .	213
10.6.6 入力を実装する . . . . .	215
10.6.7 入力をテストする . . . . .	217
<b>第 11 章 A 拡張の実装</b>	<b>219</b>
11.1 アトミック操作 . . . . .	219
11.1.1 アトミック操作とは何か? . . . . .	219
11.1.2 Zaamo 拡張 . . . . .	220
11.1.3 Zalrsc 拡張 . . . . .	220

---

11.1.4 命令の順序 . . . . .	221
<b>11.2 命令のデコード . . . . .</b>	<b>222</b>
11.2.1 is_amo フラグを実装する . . . . .	223
11.2.2 アドレスを変更する . . . . .	224
11.2.3 ライトバックする条件を変更する . . . . .	225
<b>11.3 amounit モジュールの作成 . . . . .</b>	<b>225</b>
11.3.1 インターフェースを作成する . . . . .	225
11.3.2 amounit モジュールの作成 . . . . .	226
<b>11.4 Zalrsc 拡張の実装 . . . . .</b>	<b>231</b>
11.4.1 LR.W、LR.D 命令を実装する . . . . .	231
11.4.2 SC.W、SC.D 命令を実装する . . . . .	233
<b>11.5 Zaamo 拡張の実装 . . . . .</b>	<b>235</b>
<b>第 12 章 C 拡張の実装</b>	<b>239</b>
12.1 概要 . . . . .	239
12.2 IALIGN の変更 . . . . .	240
12.3 実装方針 . . . . .	241
12.4 命令フェッチモジュールの実装 . . . . .	242
12.4.1 インターフェースを作成する . . . . .	242
12.4.2 core モジュールの IF ステージを削除する . . . . .	243
12.4.3 inst_fetcher モジュールを作成する . . . . .	244
12.4.4 inst_fetcher モジュールと core モジュールを接続する . . . . .	248
12.5 16 ビット境界に配置された 32 ビット幅の命令のサポート . . . . .	249
12.6 RVC 命令の変換 . . . . .	251
12.6.1 RVC 命令フラグの実装 . . . . .	251
12.6.2 32 ビット幅の命令に変換する . . . . .	253
12.6.3 RVC 命令を発行する . . . . .	258
<b>第 III 部 特権/割り込みの実装</b>	<b>261</b>
<b>第 13 章 M-mode の実装 (1. CSR の実装)</b>	<b>262</b>
13.1 概要 . . . . .	262
13.1.1 特権レベルとは何か? . . . . .	262
13.1.2 特権レベルの実装順序 . . . . .	263
13.1.3 XLEN の定義 . . . . .	264
13.2 misa レジスタ (Machine ISA) . . . . .	264
13.3 mimpid レジスタ (Machine Implementation ID) . . . . .	265
13.4 mhartid レジスタ (Hart ID) . . . . .	266
13.5 mstatus レジスタ (Machine Status) . . . . .	266

---

13.6	ハードウェアパフォーマンスマニタ . . . . .	268
13.6.1	mcycle レジスタ . . . . .	268
13.6.2	minstret レジスタ . . . . .	269
13.7	mscratch レジスタ (Machine Scratch) . . . . .	270
<b>第 14 章</b>	<b>M-mode の実装 (2. 割り込みの実装)</b>	<b>272</b>
14.1	概要 . . . . .	272
14.1.1	割り込みとは何か? . . . . .	272
14.1.2	RISC-V の割り込み . . . . .	272
14.1.3	割り込みの優先順位 . . . . .	273
14.1.4	割り込みの原因 (cause) . . . . .	274
14.1.5	ACLINT (Advanced Core Local Interruptor) . . . . .	274
14.2	ACLINT モジュールの作成 . . . . .	275
14.2.1	インターフェースを作成する . . . . .	275
14.2.2	aclint_memory モジュールを作成する . . . . .	276
14.2.3	mmio_controller モジュールに ACLINT を追加する . . . . .	276
14.2.4	ACLINT と mmio_controller、csrunit モジュールを接続する . . . . .	278
14.3	ソフトウェア割り込みの実装 (MSWI) . . . . .	279
14.3.1	MSIP レジスタを実装する . . . . .	280
14.3.2	mip、mie レジスタを実装する . . . . .	281
14.3.3	mstatus の MIE、MPIE ビットを実装する . . . . .	283
14.3.4	割り込み処理の実装 . . . . .	284
14.3.5	ソフトウェア割り込みをテストする . . . . .	286
14.4	mtvec の Vectored モードの実装 . . . . .	287
14.5	タイマ割り込みの実装 (MTIMER) . . . . .	288
14.5.1	タイマ割り込み . . . . .	288
14.5.2	MTIME、MTIMECMP レジスタを実装する . . . . .	288
14.5.3	mip.MTIP、割り込み原因を設定する . . . . .	290
14.5.4	タイマ割り込みをテストする . . . . .	290
14.6	WFI 命令の実装 . . . . .	291
14.7	time、instret、cycle レジスタの実装 . . . . .	292
<b>第 15 章</b>	<b>U-mode の実装</b>	<b>294</b>
15.1	misa.Extensions の変更 . . . . .	294
15.2	mstatus.UXL の実装 . . . . .	294
15.3	mstatus.TW の実装 . . . . .	295
15.4	mstatus.MPP の実装 . . . . .	295
15.5	CSR のアクセス権限の確認 . . . . .	297
15.6	mcouteren レジスタの実装 . . . . .	298
15.7	MRET 命令の実行を制限する . . . . .	299

---

15.8	ECALL 命令の cause を変更する . . . . .	300
15.9	割り込み条件の変更 . . . . .	301
<b>第 16 章 S-mode の実装 (1. CSR の実装)</b>		<b>302</b>
16.1	misa.Extensions、mstatus.SXL、mstatus.MPP の実装 . . . . .	303
16.2	scounteren レジスタの実装 . . . . .	303
16.3	sstatus レジスタの実装 . . . . .	305
16.4	トラップの委譲 . . . . .	306
16.4.1	トラップの委譲 . . . . .	306
16.4.2	トラップに関連するレジスタを作成する . . . . .	307
16.4.3	stvec レジスタの実装 . . . . .	308
16.4.4	トラップで sepc、scause、stval レジスタを変更する . . . . .	309
16.4.5	mstatus の SIE、SPIE、SPP ビットを実装する . . . . .	309
16.4.6	SRET 命令を実装する . . . . .	310
16.4.7	SEI、SSI、STI を実装する . . . . .	312
16.4.8	割り込み条件、トラップの動作を変更する . . . . .	315
16.5	ソフトウェア割り込みの実装 (SSWI) . . . . .	316
<b>第 17 章 S-mode の実装 (2. 仮想記憶システム)</b>		<b>318</b>
17.1	概要 . . . . .	318
17.1.1	仮想記憶システム . . . . .	318
17.1.2	ページング方式 . . . . .	318
17.1.3	RISC-V の仮想記憶システム . . . . .	318
17.2	satp レジスタ . . . . .	319
17.3	Sv39 のアドレス変換 . . . . .	320
17.3.1	ページングが有効になる条件 . . . . .	321
17.3.2	PTE のフェッチ . . . . .	321
17.4	実装順序 . . . . .	323
17.5	メモリで発生する例外の実装 . . . . .	324
17.5.1	例外を伝達する . . . . .	325
17.5.2	ページフォルトが発生した正確なアドレスを特定する . . . . .	330
17.6	satp レジスタの作成 . . . . .	331
17.7	mstatus の MXR、SUM、MPRV ビットの実装 . . . . .	332
17.8	アドレス変換モジュール (PTW) の実装 . . . . .	333
17.8.1	CSR のインターフェースを実装する . . . . .	333
17.8.2	Bare だけに対応するアドレス変換モジュールを実装する . . . . .	334
17.8.3	ptw モジュールをインスタンス化する . . . . .	337
17.9	Sv39 の実装 . . . . .	340
17.9.1	定数の定義 . . . . .	340
17.9.2	PTE の定義 . . . . .	341

17.9.3 ptw モジュールの実装 . . . . .	343
<b>17.10 SFENCE.VMA 命令の実装 . . . . .</b>	<b>347</b>
17.10.1 SFENCE.VMA 命令をデコードする . . . . .	347
17.10.2 特権レベルの確認、mstatus.TVM を実装する . . . . .	347
<b>17.11 パイプラインをフラッシュする . . . . .</b>	<b>348</b>
17.11.1 CSR の変更 . . . . .	348
17.11.2 FENCE.I 命令の実装 . . . . .	349
<b>第 18 章 PLIC の実装</b>	<b>350</b>
<b>第 19 章 Linux を動かす</b>	<b>351</b>
<b>あとがき (第 I 部)</b>	<b>352</b>
<b>あとがき (第 II 部、第 III 部)</b>	<b>353</b>
<b>このプロジェクトに貢献する</b>	<b>354</b>
<b>参考文献</b>	<b>355</b>

## **第Ⅰ部**

# **RV32I/RV64I の実装**

# 第 1 章

## 環境構築

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

### 1.1 Veryl

#### Veryl のインストール

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

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

##### ▼ リスト 1.1: Veryl のインストール

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

#### Veryl の更新

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

##### ▼ リスト 1.2: Veryl の更新

```
$ verylup update
```

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

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

## インストールするバージョンの指定

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

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

```
$ verylup install 0.13.1
```

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

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

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

## 使用するバージョンの指定

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

### ▼ リスト 1.5: veryl のバージョン確認

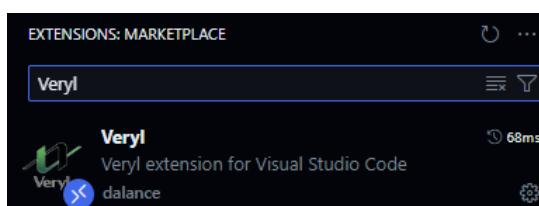
```
$ veryl --version  
veryl 0.13.2
```

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

### ▼ リスト 1.6: Veryl のバージョン 0.13.2 を使用する

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

## エディタ拡張のインストール



▲図 1.1: Veryl の VSCode 拡張

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

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

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

## 1.2 Verilator

Verilator<sup>\*4</sup>は、SystemVerilog のシミュレータを生成するためのソフトウェアです。

パッケージマネージャ (apt、Homebrew など) を利用してインストールできます。パッケージマネージャが入っていない場合は、ドキュメント<sup>\*5</sup>を参考にインストールしてください。



### 本書で利用する Verilator のバージョン

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

## 1.3 riscv-gnu-toolchain

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

GitHub の [riscv-collab/riscv-gnu-toolchain](https://github.com/riscv-collab/riscv-gnu-toolchain)<sup>\*6</sup> 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

GOWIN の FPGA 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。

<sup>\*3</sup> <https://github.com/veryl-lang/veryl.vim>

<sup>\*4</sup> <https://github.com/verilator/verilator>

<sup>\*5</sup> <https://verilator.org/guide/latest/install.html>

<sup>\*6</sup> <https://github.com/riscv-collab/riscv-gnu-toolchain>

## 第 2 章

# ハードウェア記述言語 Veril

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

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

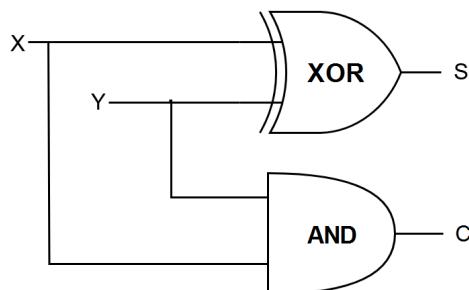
まずは、論理回路を構成する方法から考えます。

## 2.1 ハードウェア記述言語

### 2.1.1 論理回路の構成

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

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



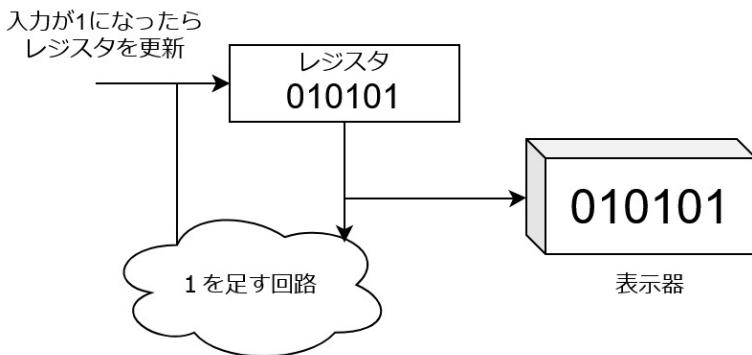
▲図 2.1: 半加算器 (MIL 記法の回路図)

▼表 2.1: 半加算器 (真理値表)

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

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

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



▲図 2.2: カウンタ (順序回路の例)

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

## 2.1.2 ハードウェア記述言語

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

## ▼リスト 2.1: SystemVerilog による半加算器の記述

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

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

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

## ▼リスト 2.2: SystemVerilog によるカウンタの記述

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

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

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

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

### 2.1.3 Veril

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

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

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

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

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



### 注意

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

## 2.2 Veryl の基本文法、機能

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

### 2.2.1 コメント

Veryl では次のようにコメントを記述できます（リスト 2.3）。

#### ▼リスト 2.3: コメント

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

<sup>\*1</sup> 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

<sup>\*2</sup> <https://doc.veryl-lang.org/playground/>

## 2.2.2 値、リテラル

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

▼表 2.2: 4-state の値

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

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

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

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

▼リスト 2.4: 符号付き型

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

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

▼表 2.3: 整数型

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

数値はリスト 2.5 のように記述できます。

▼リスト 2.5: 数値リテラル

```

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

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

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

// 数値リテラルの好きな場所に_を挿入できる
1_2_34_567

// xとzは大文字でも良い
4'bxXzZ

// 全ビット0、1、x、zにする
'0
'1
'x
'z

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

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

```

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

#### ▼ リスト 2.6: 文字列リテラル

```

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

```

### 2.2.3 module

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

#### ▼ リスト 2.7: 半加算器 (HalfAdder) モジュール

```

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

```

```

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

```

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

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

### 変数のブロッキング代入

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

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

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

#### ▼リスト 2.8: x に y を割り当てる

```

always_comb {
    x = y;
}

```

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

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



▲図 2.3: x は y の値の変化に追従する

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

▼リスト 2.9: ブロッキング代入は逐次実行される

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

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

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

▼リスト 2.10: `assign` 文によるブロッキング代入

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

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

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

### 変数の宣言

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

▼リスト 2.11: 変数の宣言

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

`var` 文で宣言した変数に対してブロッキング代入できます。

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

▼リスト 2.12: 変数の宣言とブロッキング代入

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

### レジスタの定義と代入

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

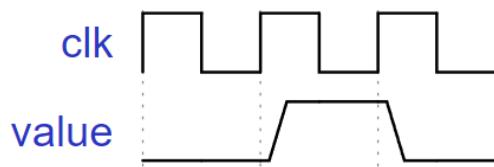
▼リスト 2.13: レジスタの定義

```
// var レジスタ名 : 型名;
var reg_value : logic<32>;
```

```
// reg_valueにブロッキング代入しない
```

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

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



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

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

#### ▼リスト 2.14: レジスタの値の初期化と変更

```
// レジスタの定義
var value : logic<32>;

// always_ff(クロック信号, リセット信号)
always_ff(clk, rst) {
    if_reset {
        // リセット信号のタイミングで0に初期化する
        value = 0;
    } else {
        // クロック信号のタイミングでカウントアップする
        value = value + 1;
    }
}
```

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

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

#### ▼リスト 2.15: クロック信号とリセット信号の省略

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

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

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

#### ▼リスト 2.16: 複数のレジスタの値を同じタイミングで変更する

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

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

#### ▼リスト 2.17: ノンブロッキング代入の更新タイミングは同じ

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

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

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

▼表 2.4: 変数への代入方法と動作の違い

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

## モジュールのインスタンス化

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

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

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

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

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

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

## パラメータ、定数

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

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

### ▼リスト 2.19: モジュールのパラメータの宣言

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

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

### ▼リスト 2.20: パラメータの値を指定する

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

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

#### ▼リスト 2.21: 定数の定義

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

### 2.2.4 ユーザー定義型

#### 構造体型

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

#### ▼リスト 2.22: 構造体型の定義

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

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

#### ▼リスト 2.23: フィールドへのアクセス、割り当て

```
// 構造体型の変数の宣言
var pair: MyPair;

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

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

#### 列挙型

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

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

#### ▼リスト 2.24: 列挙型の定義

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

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

enum型の値は `型名::バリアント名` で利用できます(リスト2.25)。

#### ▼リスト2.25: 列挙型の値

```
// enum型の変数の定義
let v : abcd = abcd::A;
```

バリアントを表す値や、バリアントを保持できるだけのビット数は省略できます(リスト2.26)。

#### ▼リスト2.26: 列挙型の省略した定義

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

### 配列

`<>`を使用することで、多次元の型を定義できます(リスト2.27)。`<>`を使用して構成される型の要素は、連続した領域に並ぶことが保証されます(図2.5)。

#### ▼リスト2.27: 多次元の型

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

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

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

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

`[]`を使用することでも、多次元の型を定義できます(リスト2.28)。ただし、`[]`を使用して構成される型の要素は、連続した領域に並ぶことが保証されません。

#### ▼リスト2.28: 配列型

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

## 型に別名をつける

`type` キーワードを使うと、型に別名を付けられます（リスト 2.29）。

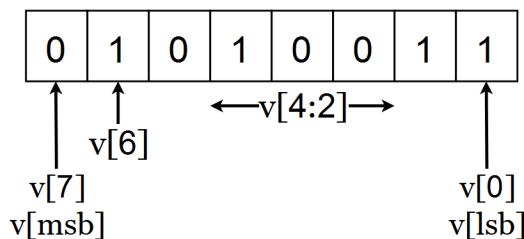
### ▼リスト 2.29: 型に別名を付ける

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

## 2.2.5 式、文、宣言

### ビット選択

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



▲図 2.6: ビット選択

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

よく使われる範囲の選択には、別の書き方が用意されています（リスト 2.30）。

### ▼リスト 2.30: 範囲の選択の別の記法

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

### 演算子

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

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

単項、二項演算子の使用例は次の通りです（リスト 2.31）。

## ▼リスト 2.31: 単項、二項演算子 (Veryl のドキュメントの例 [4] を改変)

```

// 単項算術演算
a = +1;
a = -1; // 正負を反転させる

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

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

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

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

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

// ビット演算 (ビット単位, bitwise)

```

```

a = 1 & 1; // ビット単位AND
a = 1 ^ 1; // ビット単位XOR
a = 1 ~^ 1; // ビット単位XNOR
a = 1 ~^ 1; // ビット単位XNOR
a = 1 | 1; // ビット単位OR

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

```

▼表 2.5: 演算子と優先度 [3]

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

## if、switch、case

条件によって動作や値を変えたいとき、if 文を使用します（リスト 2.32）。if 文は式にできます。if 式は必ず値を返す必要があり、else が必須です。

▼リスト 2.32: if 文、if 式

```

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

```

```

        } else {
            // if式はelseが必須
            4
        };
    }
}

```

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

リスト 2.32 と同じ意味の文を switch 文で書けます（リスト 2.33）。どの条件にも当てはまらないときの動作は default で指定します。switch は式にできます。switch 式は必ず値を返す必要があり、default が必須です。

#### ▼リスト 2.33: switch 文、switch 式

```

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

```

リスト 2.32 のように 1 つの要素（WIDTH）の一一致のみが条件のとき、同じ意味の文を case 文で書けます（リスト 2.34）。式にできたり、式に default が必須なのは switch 文と同様です。

#### ▼リスト 2.34: case 文、case 式

```

var v: logic<32>;
always_comb {
    case WIDTH {
        // WIDTH == 0のとき
        0: {
            v = 0;
        }
        // WIDTH != 0かつWIDTH == 1のとき
        1: v = 1; // 要素が1つの文のとき、{}は省略できる
        // WIDTH != 0かつWIDTH != 1のとき
        default:
            // case式
    }
}

```

```
v = case WIDTH {
    3: 3, // カンマで区切る
    default : 4, // case式はdefaultが必須
};

}
```

### 連結、repeat

ビット列や文字列を連結したいとき、`{}` を使用できます（リスト 2.35）。`+` では連結できない（値の足し算になる）ことに注意してください。同じビット列、文字列を繰り返して連結したいときは `repeat` キーワードを使用します（リスト 2.36）。

#### ▼リスト 2.35: 連結

```
{12'h123, 32'habcd0123} // 44'h123_abcd0123になる
{"Hello", " ", "World!"} // "Hello World!"になる
```

#### ▼リスト 2.36: repeat を使って連結を繰り返す

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

### for

`for` 文はループを実現するための文です。`for` 文はリスト 2.37 のように記述できます。例えばループ変数が 0 から 31 になるまで（32 回）繰り返すなら、範囲に `0..32`、または `0..=31` と記述します。範囲には定数のみ指定できます。

#### ▼リスト 2.37: for 文の記法

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

`break` 文を使うとループから抜け出せます。例えばリスト 2.38 では `x` の値は 256 になります。

#### ▼リスト 2.38: always\_comb ブロック内で for 文を記述する例

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

### inside、outside

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

**inside** 式 {範囲} で、式の結果が範囲内にあるかという条件を記述できます (リスト 2.39)。逆に、範囲外にあるという条件は **outside** 式で記述できます。

#### ▼ リスト 2.39: inside、outside

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

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

## function

何度も記述する操作や計算は、関数 (**function**) を使うことでまとめて記述できます (リスト 2.40)。関数は値を引数で受け取り、**return** 文で値を返します。値を返さないとき、戻り値の型の指定を省略できます。

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

#### ▼ リスト 2.40: 関数

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

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

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

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

```

val v3 : logic<32>;
always_comb {
    assign_plus1(v1, v3); // v3 = v1 + 1
}

```

## 2.2.6 interface

モジュールに何個もポートが存在するとき、ポートの接続は非常に手間のかかる作業になります。例えばリスト 2.41 では、向きが対になっているポートが ModuleA と ModuleB に定義されており、これを一つ一つ接続しています。

### ▼リスト 2.41: モジュールのポートの相互接続

```

module ModuleA (
    req_a: output logic,
    req_b: output logic,
    req_c: output logic,
){}

module ModuleB (
    resp_a: input logic,
    resp_b: input logic,
    resp_c: input logic,
){}

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

```

モジュール間のポートの接続を簡単に行うために、インターフェース (**interface**) という機能が用意されています。リスト 2.41 の ModuleA と ModuleB を相互接続するようなインターフェースは次のように定義できます（リスト 2.42）。

### ▼リスト 2.42: インターフェースの定義

```

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

```

```

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

```

iff\_ab インターフェースを利用すると、リスト 2.41 を簡潔に記述できます（リスト 2.43）。

#### ▼ リスト 2.43: インターフェースによる接続

```

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

```

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

モジュールと同じように、インターフェースにはパラメータを宣言できます（リスト 2.44）。

#### ▼ リスト 2.44: パラメータ付きのインターフェース

```

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

```

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

## 2.2.7 package

複数のモジュールやインターフェースにまたがって使用したいパラメータや型、関数はパッケージ（package）に定義できます（リスト 2.45）。

## ▼リスト 2.45: パッケージの定義

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

パッケージに定義した要素には、`パッケージ名::要素名` でアクセスできます（リスト 2.46）。

## ▼リスト 2.46: パッケージの要素にアクセスする

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

`import` 文を使用すると、要素へのアクセス時にパッケージ名の指定を省略できます（リスト 2.47）。

## ▼リスト 2.47: パッケージを import する

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

## 2.2.8 ジェネリクス

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

例えば、要素に任意の型 `T` や `W` ビットのデータを持つ構造体は、次のようにジェネリックパラメータ (generic parameter) を使うことで定義できます（リスト 2.48）。ジェネリックパラメータに渡される値は、ジェネリクスの定義位置からアクセスできる定数である必要があります。

## ▼リスト 2.48: パラメータ化された構造体

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

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

```
C: logic<0>
}

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

## 2.2.9 その他の機能、文

### initial、final

**initial** ブロックの中の文はシミュレーションの開始時に実行されます。**final** ブロックの中の文はシミュレーションの終了時に実行されます(リスト 2.49)。

#### ▼リスト 2.49: initial、final ブロック

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

### SystemVerilog との連携

SystemVerilog のモジュールやパッケージ、インターフェースを利用できます。SystemVerilog のリソースにアクセスするには `$sv::` を使用します(リスト 2.50)。

#### ▼リスト 2.50: SystemVerilog の要素を利用する

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

    var s: logic;
    var c: logic;

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

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

```
}
```

SystemVerilog のソースコードを直接埋め込み、展開できます (リスト 2.51)。

#### ▼リスト 2.51: SystemVerilog 記述を埋め込む

```
// SystemVerilog記述を直接埋め込む
embed (inline) sv{{{
    module ModuleA(
        output logic a
    );
        assign a = 0;
    endmodule
}}}

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

### システム関数、システムタスク

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

▼表 2.6: 本書で使用するシステム関数、システムタスク

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

それぞれの使用例は次の通りです (リスト 2.52)。システム関数やシステムタスクを利用するとときは、通常の関数呼び出しのように使用します。

#### ▼リスト 2.52: システム関数、システムタスクの使用例

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

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

var uvalue : u32;
```

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

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

## アトリビュート

アトリビュートを使うと、宣言に注釈をつけられます。例えばリスト 2.53 は、リスト 2.54 にトランスペイルされます。

### ▼ リスト 2.53: アトリビュートを使った Veril コード

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

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

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

### ▼ リスト 2.54: 同じ意味の SystemVerilog コード

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

`ifdef IS_DEBUG
logic bbb;
`endif

`ifndef TEST
logic ccc;
`endif
```

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

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

## 標準ライブラリ

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

本書では標準ライブラリを使用していないため、説明は割愛します。

# 第3章

## RV32I の実装

本章では、RISC-V の基本整数命令セットである **RV32I** を実装します。基本整数命令という名前の通り、整数の足し引きやビット演算、ジャンプ、分岐命令などの最小限の命令しか実装されていません。また、32 ビット幅の汎用レジスタが 32 個定義されています。ただし、0 番目のレジスタの値は常に 0 です。

RISC-V の CPU は基本整数命令セットを必ず実装して、他の命令や機能は拡張として実装します。複雑な機能を持つ CPU を実装する前に、まずは最小限の命令を実行できる CPU を実装しましょう。

### 3.1 CPU は何をやっているのか？

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

1. メモリ (memory, 記憶装置) からプログラムを読み込む
2. プログラムを実行する
3. 1、2 の繰り返し

ここで、メモリから読み込まれる「プログラム」とは一体何を指しているのでしょうか？普通のプログラマが書くのは 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 が読み込んで実行できる形式のプログラムです。これはよく機械語 (machine code) と呼ばれ、0 と 1 で表される 2 進数のビット列<sup>\*1</sup>で記述されています。

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

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

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

CPU は機械語を実行しますが、一気にすべての機械語を読み込んだり実行するわけではなく、機械語の最小単位である命令(instruction)を一つずつ読み込んで実行します。命令をメモリに要求、取得することを、命令をフェッチすると呼びます。

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

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

- レジスタ (= CPU 内に存在する計算データ用のレジスタ列) の番号
- 即値 (= 命令のビット列から生成される数値)

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

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

計算やメモリの読み書きが終わると、その結果をレジスタに格納します。例えば、足し算を行う命令なら足し算の結果、メモリから値を読み込む命令なら読み込まれた値を格納します。

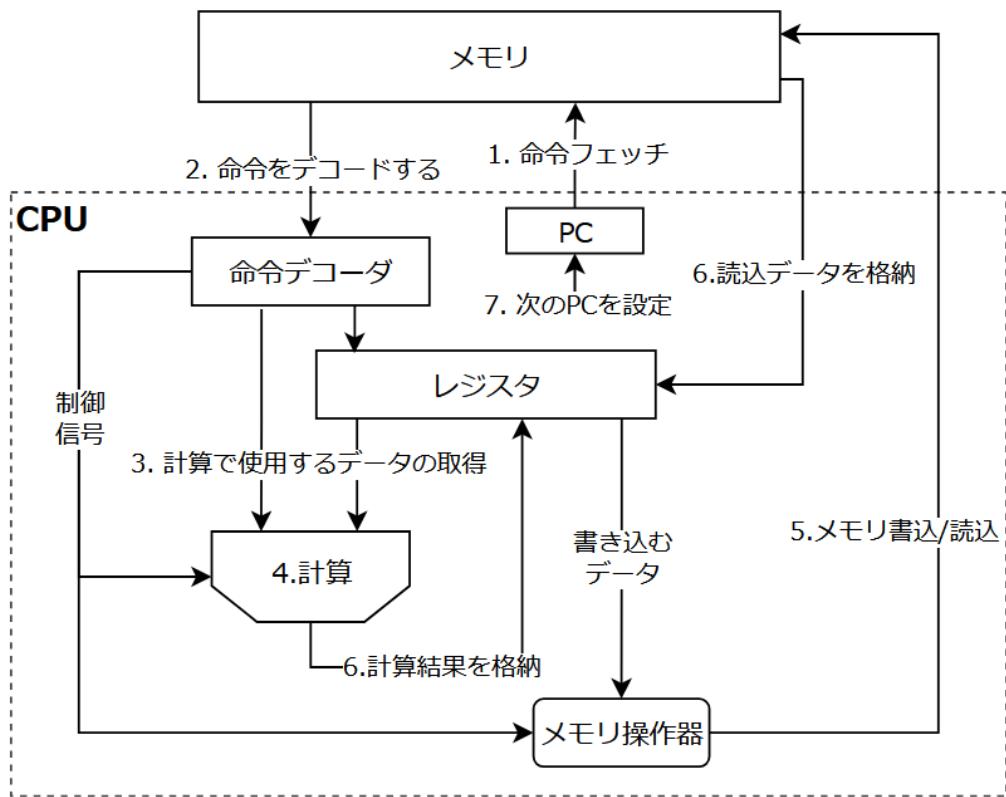
これで命令の実行は終わりですが、CPU は次の命令を実行する必要があります。今現在実行している命令のアドレスを格納しているレジスタのことをプログラムカウンタ(program counter, PC)と呼びます。CPU は PC の値をメモリに渡すことで命令をフェッチしています。

CPU は次の命令を実行するために、PC の値を次の命令のアドレスに設定します。ジャンプ命令の場合は PC の値をジャンプ先のアドレスに設定します。分岐命令の場合は、まず、分岐の成否を判定します。分岐が成立する場合は PC の値を分岐先のアドレスに設定します。分岐が成立しない場合は通常の命令と同じです。

ここまで話をまとめると、CPU の動作は次のようにになります(図 3.1)。

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

CPU が一体どんなものなのかが分かりましたか? 実装を始めましょう。



▲図 3.1: CPU の動作

## 3.2 プロジェクトの作成

まず、Veryl のプロジェクトを作成します (リスト 3.1)。プロジェクトは core という名前にしています。

### ▼リスト 3.1: 新規プロジェクトの作成

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

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

### ▼リスト 3.2: Veryl.toml

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

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

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

▼リスト 3.3: src ディレクトリを作成する

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

## 3.3 定数の定義

いよいよコードを記述します。まず、CPU 内で何度も使用する定数や型を書いておくためのパッケージを作成します。

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

▼リスト 3.4: eei.veryl

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

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

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

eei パッケージには、次の定数を定義します。

### XLEN

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

RV32I のレジスタの長さは 32 ビットであるため、値を 32 にしています。

### ILEN

ILEN は、RISC-V において CPU の実装がサポートする命令の最大の幅を示す値として定義されています。RISC-V の命令の幅は、後の章で説明する圧縮命令を除けばすべて 32 ビットです。そのため、値を 32 にしています。

また、何度も使用することになる型に、type 文によって別名を付けています。

**UIntX、UInt32、UInt64**

幅がそれぞれ XLEN、32、64 の符号なし整数型

**Inst**

命令のビット列を格納するための型

**Addr**

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

## 3.4 メモリ

CPU はメモリに格納された命令を実行します。そのため、CPU の実装のためにはメモリの実装が必要です。RV32Iにおいて命令の幅は 32 ビット (ILEN) です。また、メモリからの読み込み命令、書き込み命令の最大の幅も 32 ビットです。

これを実現するために、次のような要件のメモリを実装します。

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

### 3.4.1 メモリのインターフェースを定義する

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

▼表 3.1: メモリモジュールに必要なポート

ポート名	型	向き	意味
valid	logic	input	メモリアクセスを要求しているかどうか
ready	logic	output	メモリアクセス要求を受容するかどうか
addr	logic<ADDR_WIDTH>	input	アクセス先のアドレス
wen	logic	input	書き込みかどうか (1 なら書き込み)
wdata	logic<DATA_WIDTH>	input	書き込むデータ
rvalid	logic	output	受容した要求の処理が終了したかどうか
rdata	logic<DATA_WIDTH>	output	受容した読み込み命令の結果

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

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

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

```

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

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

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

```

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

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

### 3.4.2 メモリモジュールを実装する

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

#### ▼リスト 3.6: メモリモジュールの定義 (memory.veryl)

```

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

```

```

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

always_comb {
    membus.ready = 1;
}

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

```

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

### **DATA\_WIDTH**

メモリのデータの単位の幅を指定するためのパラメータです。

この単位ビットでデータを読み書きします。

### **ADDR\_WIDTH**

データのアドレスの幅(メモリの容量)を指定するためのパラメータです。

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

ポートには、クロック信号とリセット信号と `membus_if` インターフェースを定義しています。  
読み込み、書き込み時の動作は次の通りです。

### **読み込み**

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

## 書き込み

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

### 3.4.3 メモリの初期化、環境変数の読み込み

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

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

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

#### ▼ リスト 3.7: util.veryl

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

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

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

## 3.5 最上位モジュールの作成

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

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

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

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

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

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

#### ▼リスト 3.9: 最上位モジュールの定義 (top.veryl)

```
import eei::*;

module top #(
    param MEMORY_FILEPATH_IS_ENV: bit      = 1,
    param MEMORY_FILEPATH        : string = "MEMORY_FILE_PATH",
) (
    clk: input clock,
    rst: input reset,
) {
    inst membus: membus_if::<MEM_DATA_WIDTH, MEM_ADDR_WIDTH>;
    inst mem: memory::<MEM_DATA_WIDTH, MEM_ADDR_WIDTH> #(
        FILEPATH_IS_ENV: MEMORY_FILEPATH_IS_ENV,
        FILEPATH        : MEMORY_FILEPATH        ,
    ) (
        clk      ,
        rst      ,
        membus   ,
    );
}
```

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

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

## 3.6 命令フェッチ

メモリを作成したので、命令フェッチ処理を作れるようになりました。  
いよいよ、CPU のメインの部分を作成します。

### 3.6.1 命令フェッチを実装する

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

#### ▼ リスト 3.10: core.veryl

```
import eei::*;

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

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

    let if_pc_next: Addr = if_pc + 4;

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

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

```

        if_pc_requested = if_pc;
    }
}
}

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

```

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

`if_pc` レジスタは PC(プログラムカウンタ) です。ここで `if_` という接頭辞は instruction fetch(命令フェッチ) の略です。`if_is_requested` はフェッチ中かどうかを管理しており、フェッチ中のアドレスを `if_pc_requested` に格納しています。どのレジスタも `0` で初期化しています。

`always_comb` ブロックでは、アドレス `if_pc` にあるデータを常にメモリに要求しています。命令フェッチではメモリの読み込みしか行わないため、`membus.wen` は `0` にしています。

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

メモリにデータを要求するとき、`if_pc` を次の命令のアドレス (`4` を足したアドレス) に変更して、`if_is_requested` を `1` に変更しています。フェッチ中かつ `membus.rvalid` が `1` のとき、命令フェッチが完了し、データが `membus.rdata` に供給されています。メモリが ready 状態なら、すぐに次の命令フェッチを開始します。この状態遷移を繰り返すことによって、アドレス `0 → 4 → 8 → c → 10 ...` の命令を次々にフェッチします。

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

### 3.6.2 memory モジュールと core モジュールを接続する

次に、top モジュールで core モジュールをインスタンス化し、membus\_if インターフェースでメモリと接続します。

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

まず、1 バイト単位のアドレスを、4 バイト単位のアドレスに変換する関数を作成します (リスト 3.11)。これは、1 バイト単位のアドレスの下位 2 ビットを切り詰めることによって実現できます。

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

```
// アドレスをメモリのデータ単位でのアドレスに変換する
function addr_to_memaddr (
    addr: input logic<XLEN> ,
) -> logic<MEM_ADDR_WIDTH> {
    return addr[$clog2(MEM_DATA_WIDTH / 8)+:MEM_ADDR_WIDTH];
}
```

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

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

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

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

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

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

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

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

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

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

### 3.6.3 命令フェッチをテストする

ここまで書いたコードが正しく動くかを検証します。

Veryl で記述されたコードは `veryl build` コマンドで SystemVerilog のコードに変換できます。変換されたソースコードをオープンソースの Verilog シミュレータである Verilator で実行することで、命令フェッチが正しく動いていることを確認します。

まず、Veryl のプロジェクトをビルドします（リスト 3.15）。

▼ リスト 3.15: Veryl のプロジェクトのビルド

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

上記のコマンドを実行すると、veryl ファイルと同名の `sv` ファイルと `core.f` ファイルが生成されます。拡張子が `sv` のファイルは SystemVerilog のファイルで、`core.f` には生成された SystemVerilog のファイルのリストが記載されています。これをシミュレータのビルドに利用します。

シミュレータのビルドには Verilator を利用します。Verilator は、与えられた SystemVerilog のコードを C++ プログラムに変換することでシミュレータを生成します。Verilator を利用するために、次のような C++ プログラムを書きます\*2。

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

▼ リスト 3.16: tb\_verilator.cpp

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

namespace fs = std::filesystem;

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

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

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

    // メモリの初期値を格納しているファイル名
    std::string memory_file_path = argv[1];
    try {
```

\*2 Verilog のソースファイルだけでビルドすることもできます

```

// 絶対パスに変換する
fs::path absolutePath = fs::absolute(memory_file_path);
memory_file_path = absolutePath.string();
} catch (const std::exception& e) {
    std::cerr << "Invalid memory file path : " << e.what() << std::endl;
    return 1;
}

// シミュレーションを実行するクロックサイクル数
unsigned long long cycles = 0;
if (argc >= 3) {
    std::string cycles_string = argv[2];
    try {
        cycles = stoull(cycles_string);
    } catch (const std::exception& e) {
        std::cerr << "Invalid number: " << argv[2] << std::endl;
        return 1;
    }
}

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

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

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

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

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

dut->final();
}

```

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

このプログラムは、コマンドライン引数として次の 2 つの値を受け取ります。

## **MEMORY\_FILE\_PATH**

メモリの初期値のファイルへのパス

実行時に環境変数 MEMORY\_FILE\_PATH として渡されます。

## **CYCLE**

何クロックで実行を終了するかを表す値

0 のときは終了しません。デフォルト値は 0 です。

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

## シミュレータのビルド

verilator コマンドを実行し、シミュレータをビルドします（リスト 3.17）。

### ▼リスト 3.17: シミュレータのビルド

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

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

### -f

SystemVerilog ソースファイルのファイルリストを指定します。今回は core.f を指定しています。

### --exe

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

### --top-module

トップモジュールを指定します。今回は top モジュールを指定しています。

### --Mdir

成果物の生成先を指定します。今回は obj\_dir ディレクトリに指定しています。

リスト 3.17 のコマンドの実行により、シミュレータが obj\_dir/sim に生成されました。

## メモリの初期化用ファイルの作成

シミュレータを実行する前にメモリの初期値となるファイルを作成します。src/sample.hex を作成し、次のように記述します（リスト 3.18）。

## ▼リスト 3.18: sample.hex

```
01234567
89abcdef
deadbeef
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）。

▼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
0x00000000	01234567
0x00000004	89abcdef
0x00000008	deadbeef
0x0000000c	cafebebe
0x00000010～	不定

**シミュレータの実行**

生成されたシミュレータを実行し、アドレスが `0`、`4`、`8`、`c` のデータが正しくフェッチされていることを確認します（リスト 3.19）。

## ▼リスト 3.19: 命令フェッチの動作チェック

```
$ obj_dir/sim src/sample.hex 5
00000000 : 01234567
00000004 : 89abcdef
00000008 : deadbeef
0000000c : cafebebe
```

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

**Makefile の作成**

ビルド、シミュレータのビルドのために一々コマンドを打つのは非常に面倒です。これらの作業を一つのコマンドで済ますために、`Makefile` を作成し、次のように記述します（リスト 3.20）。

## ▼リスト 3.20: Makefile

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

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

```

build:
    veryl fmt
    veryl build

clean:
    veryl clean
    rm -rf $(OBJ_DIR)

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

.PHONY: build clean sim

```

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

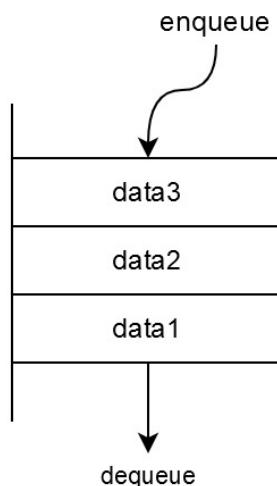
▼ リスト 3.21: Makefile によって追加されたコマンド

```

$ make build ← Verilator のソースコードのビルド
$ make sim  ← シミュレータのビルド
$ make clean ← ビルドした成果物の削除

```

### 3.6.4 フェッチした命令を FIFO に格納する



▲ 図 3.2: FIFO

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

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

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

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

## FIFO の作成

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

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

▼リスト 3.22: FIFO モジュールの実装 (fifo.veryl)

```
module fifo #(
    param DATA_TYPE: type = logic,
    param WIDTH     : u32   = 2      ,
) (
    clk      : input  clock      ,
    rst      : input  reset      ,
    wready: output logic      ,
    wvalid: input  logic      ,
    wdata : input  DATA_TYPE,
    rready: input  logic      ,
    rvalid: output logic      ,
    rdata : output DATA_TYPE,
) {
    // 2つ以上空きがあるかどうか
    var wready_two: logic;

    if WIDTH == 1 :width_one {
        always_comb {
            wready      = !rvalid || rready;
            wready_two = 0;
        }
        always_ff {
            if_reset {
                rvalid = 0;
            } else {
                if wready && wvalid {
                    rdata  = wdata;
                    rvalid = 1;
                } else if rready {

```

<sup>\*3</sup> <https://std.veryl-lang.org/>

```

        rvalid = 0;
    }
}
} else {
    type Ptr = logic<WIDTH>;
    var head      : Ptr;
    var tail      : Ptr;
    let tail_plus1: Ptr = tail + 1 as Ptr;
    let tail_plus2: Ptr = tail + 2 as Ptr;

    var mem: DATA_TYPE [2 ** WIDTH];
    always_comb {
        wready      = tail_plus1 != head;
        wready_two = wready && tail_plus2 != head;
        rvalid     = head != tail;
        rdata      = mem[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` 個格納できる FIFO です。操作は次のように行います。

### データを追加する

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

### データを取り出す

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

データの格納状況は、`head` レジスタと `tail` レジスタで管理します。データを追加するとき、つまり `wready && wvalid` のとき、`tail = tail + 1` しています。データを取り出すとき、つまり `rready && rvalid` のとき、`head = head + 1` しています。

データを追加できる状況とは、`tail` に 1 を足しても `head` を超えないとき、つまり、`tail` が指す場所が一周し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できません。データを取り出せる状況とは、`head` と `tail` の指す場所が違うときです。`WIDTH` が 1 のときは特別で、既にデータが 1 つ入っていても、`rready` が 1 のときはデータを追加できるようにしています。

### 命令フェッチ処理の変更

`fifo` モジュールを使って、命令フェッチ処理を変更します。

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

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

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

次に、FIFO と接続するための変数を定義します（リスト 3.24）。

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

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

FIFO モジュールをインスタンス化します（リスト 3.25）。`DATA_TYPE` パラメータに `if_fifo_type` を渡すことで、アドレスと命令のペアを格納できるようにします。`WIDTH` パラメータには 3 を指定することで、サイズを `2 ** 3 - 1 = 7` にしています。このサイズは適当です。

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

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

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

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

#### ▼ リスト 3.26: フェッチ処理の変更 (core.veryl)

```
// 命令フェッチ処理
always_comb {
    // FIFOに2個以上空きがあるとき、命令をフェッチする
    membus.valid = if_fifo.wready_two;
    membus.addr = if_pc;
    membus.wen = 0;
    membus.wdata = 'x; // wdataは使用しない

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

リスト 3.26 では、メモリに命令フェッチを要求する条件を FIFO に 2 つ以上空きがあるという条件に変更しています<sup>\*4</sup>。これにより、FIFO があふれてしまうことがなくなります。また、FIFO から常にデータを取り出すようにしています。

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

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

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

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

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

## ▼ リスト 3.28: 変数の初期化 (core.veryl)

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

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

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

命令フェッチは FIFO に 2 つ以上空きがあるときに行うため、まだ追加されていないデータが `if_fifo_wdata` に格納されていても、別のデータに上書きされてしまうことはありません。

**FIFO のテスト**

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

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

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

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

シミュレータを実行します（リスト 3.30）。命令がフェッチされて表示されるまでに、FIFO に格納してから取り出すクロック分だけ遅延があることに注意してください。

## ▼ リスト 3.30: FIFO をテストする

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

**3.7****命令のデコードと即値の生成**

命令をフェッチできたら、フェッチした命令がどのような意味を持つかをチェックし、CPU が

何をすればいいかを判断するためのフラグや値を生成します。この作業のことを命令のデコード (decode) と呼びます。

RISC-V の命令のビット列には次のような要素が含まれています。

### オペコード (opcode)

5 ビットの値です。命令を区別するために使用されます。

### funct3、funct7

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

### 即値 (Immediate, imm)

命令のビット列の中に直接含まれる数値です。

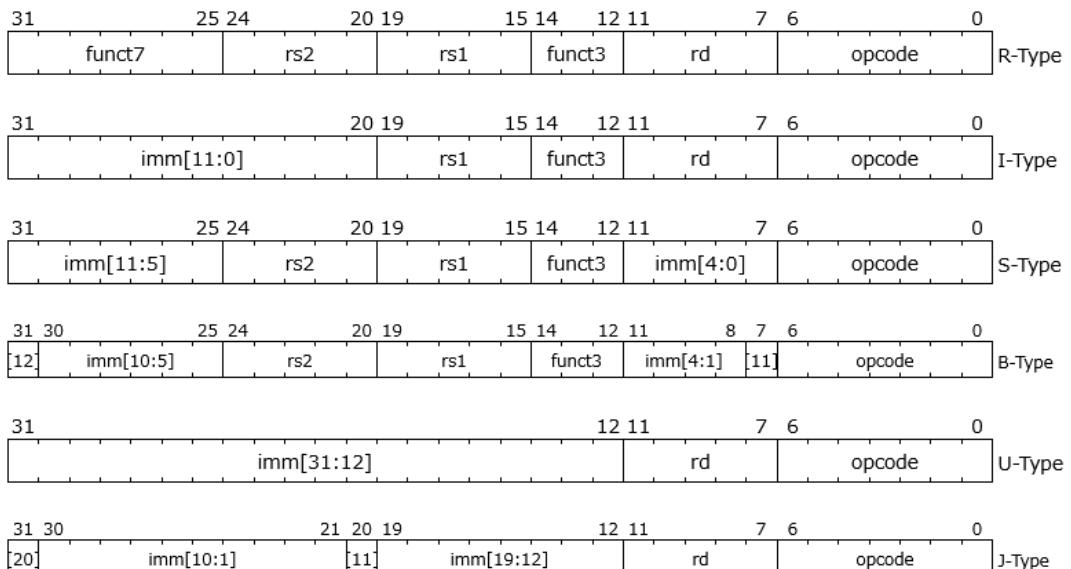
### ソースレジスタ (Source Register) の番号

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

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

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

RISC-V にはいくつかの命令の形式がありますが、RV32I には R、I、S、B、U、J の 6 つの形式の命令が存在しています (図 3.3)。



▲図 3.3: RISC-V の命令形式 [5]

### R 形式

ソースレジスタ (rs1, rs2) が 2 つ、デスティネーションレジスタ (rd) が 1 つの命令形式で

す。2つのソースレジスタの値を使って計算し、その結果をデスティネーションレジスタに格納します。例えば ADD(足し算)、SUB(引き算) 命令に使用されています。

#### I 形式

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

#### S 形式

ソースレジスタ (rs1, rs2) が 2 つの命令形式です。12 ビットの即値 (imm[11:5], imm[4:0]) が命令中に含まれており、即値と rs1 を足し合わせたメモリのアドレスに、rs2 を書き込みます。例えば SW 命令 (メモリに 32 ビット書き込む命令) に使用されています。

#### B 形式

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

#### U 形式

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

#### J 形式

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

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

### 3.7.1 デコード用の定数と型を定義する

デコード処理を書く前に、デコードに利用する定数と型を定義します。 `src/corectrl.veryl` を作成し、次のように記述します (リスト 3.31)。

#### ▼ リスト 3.31: corectrl.veryl

```
import eei::*;

package corectrl {
    // 命令形式を表す列挙型
    enum InstType: logic<6> {
        X = 6'b000000,
        R = 6'b000001,
        I = 6'b000010,
        S = 6'b000100,
```

```

B = 6'b001000,
U = 6'b010000,
J = 6'b100000,
}

// 制御に使うフラグ用の構造体
struct InstCtrl {
    itype : InstType, // 命令の形式
    rwb_en : logic, // レジスタに書き込むかどうか
    is_lui : logic, // LUI命令である
    is_aluop: logic, // ALUを利用する命令である
    is_jump : logic, // ジャンプ命令である
    is_load : logic, // ロード命令である
    funct3 : logic <3>, // 命令のfunct3フィールド
    funct7 : logic <7>, // 命令のfunct7フィールド
}
}
}

```

`InstType` は、命令の形式を表すための列挙型です。`InstType` の幅は 6 ビットで、それぞれのビットに 1 つの命令形式が対応しています。どの命令形式にも対応しない場合、すべてのビットが 0 の `InstType::X` を対応させます。

`InstCtrl` は、制御に使うフラグをひとまとめにした構造体です。`itype` には命令の形式、`funct3` と `funct7` にはそれぞれ命令の `funct3` と `funct7` フィールドを格納します。これ以外の構造体のフィールドは、使用するときに説明します。

命令をデコードするとき、まず `opcode` を使って判別します。このために、デコードに使う定数を `eei` パッケージに記述します（リスト 3.32）。

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

```

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

```

これらの値とそれぞれの命令の対応は、仕様書 [6] を確認してください。

### 3.7.2 制御フラグと即値を生成する

デコード処理を書く準備が整いました。`src/inst_decoder.veryl` を作成し、次のように記述します（リスト 3.33）。

## ▼リスト 3.33: inst\_decoder.veryl

```

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

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

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

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

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

    always_comb {
        imm = case op {
            OP_LUI, OP_AUIPC: imm_u,
            OP_JAL : imm_j,
            OP_JALR, OP_LOAD: imm_i,
            OP_OP_IMM : imm_i,
            OP_BRANCH : imm_b,
            OP_STORE : imm_s,
            default : 'x,
        };
        ctrl = {case op {
            OP_LUI : {InstType::U, T, T, F, F, F},
            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 形式の命令を考えます。まず、命令のビット列から即値部分を取り出して変数 `imm_b_g` を生成します。B 形式の命令内に含まれている即値は 12 ビットで、最上位ビットは符号ビットです。最上位ビットを繰り返す(符号拡張する)ことによって、32 ビットの即値 `imm_b` を生成します。

`always_comb` ブロックでは、`opcode` を `case` 式で分岐することにより `imm` ポートに適切な即値を供給しています。

### 制御フラグの生成

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

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

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

### 3.7.3 デコーダをインスタンス化する

`inst_decoder` モジュールを、core モジュールでインスタンス化します(リスト 3.34)。

#### ▼リスト 3.34: `inst_decoder` モジュールのインスタンス化 (core.veryl)

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

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

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

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

## ▼リスト 3.35: デコード結果のデバッグ表示 (core.veryl)

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

`src/sample.hex` をメモリの初期値として使い、デコード結果を確認します（リスト 3.36）。

## ▼リスト 3.36: デコーダをテストする

```
$ make build
$ make sim
$ obj_dir/sim src/sample.hex 7
00000000 : 01234567
  itype   : 000010
  imm     : 00000012
00000004 : 89abcdef
  itype   : 100000
  imm     : ffffb09a
00000008 : deadbeef
  itype   : 100000
  imm     : fffffdb5ea
0000000c : cafebeb0
  itype   : 000000
  imm     : 00000000
```

例えば `32'h01234567` は、`jalr x10, 18(x6)` という命令のビット列になります。命令の種類は JALR で、命令形式は I 形式、即値は 10 進数で 18 です。デコード結果を確認すると、`itype` が `32'h0000010`、`imm` が `32'h00000012` になっており、正しくデコードできていることを確認できます。

## 3.8 レジスタの定義と読み込み

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

### 3.8.1 レジスタファイルを定義する

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

## ▼リスト 3.37: レジスタの定義 (core.veryl)

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

レジスタをまとめたものなどをレジスタファイル (register file) と呼ぶため、`regfile` という名前をつけています。

### 3.8.2 レジスタの値を読み込む

レジスタを定義したので、命令が使用するレジスタの値を取得します。

図 3.3 を見るとわかるように、RISC-V の命令は形式によってソースレジスタの数が異なります。例えば、R 形式はソースレジスタが 2 つで、2 つのレジスタの値を使って実行されます。それに対して、I 形式のソースレジスタは 1 つです。I 形式の命令の実行にはソースレジスタの値と即値を利用します。

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

## ▼リスト 3.38: 命令が使うレジスタの値を取得する (core.veryl)

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

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

let rs2_data: UIntX = if rs2_addr == 0 {
    0
} else {
    regfile[rs2_addr]
};
```

`if` 式を使うことで、0 番目のレジスタが指定されたときは、値が常に `0` になるようにします。

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

## ▼リスト 3.39: レジスタの値をデバッグ表示する (core.veryl)

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

```

        $display(" rs1[%d] : %h", rs1_addr, rs1_data);
        $display(" rs2[%d] : %h", rs2_addr, rs2_data);
    }
}

```

早速動作のテストをしたいところですが、今のままだとレジスタの値が初期化されておらず、0番目のレジスタの値以外は不定値<sup>\*5</sup>になってしまいます。

これではテストする意味がないため、レジスタの値を適当な値に初期化します。always\_ff ブロックの if\_reset で、`i` 番目 ( $0 < i < 32$ ) のレジスタの値を `i + 100` で初期化します（リスト 3.40）。

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

```

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

```

レジスタの値を読み込んでいることを確認します（リスト 3.41）。

#### ▼ リスト 3.41: レジスタ読み込みのデバッグ

```

$ make build
$ make sim
$ obj_dir/sim sample.hex 7
00000000 : 01234567
    itype   : 000010
    imm     : 00000012
    rs1[ 6] : 0000006a
    rs2[18] : 00000076
00000004 : 89abcdef
    itype   : 100000
    imm     : ffffb09a
    rs1[23] : 0000007b
    rs2[26] : 0000007e
00000008 : deadbeef
    itype   : 100000
    imm     : fffffdb5ea
    rs1[27] : 0000007f
    rs2[10] : 0000006e
0000000c : cafebeb0
    itype   : 000000
    imm     : 00000000
    rs1[29] : 00000081
    rs2[15] : 00000073

```

<sup>\*5</sup> 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

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

シミュレーションと結果が一致していることを確認してください。

## 3.9 ALU による計算の実装

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

### 3.9.1 ALU モジュールを作成する

レジスタと即値の幅は XLEN です。計算には符号付き整数と符号なし整数向けの計算があります。符号付き整数を利用するため、eei モジュールに XLEN ビットの符号付き整数型を定義します(リスト 3.42)。

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

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

次に、`src/alu.veryl` を作成し、次のように記述します(リスト 3.43)。

▼ リスト 3.43: alu.veryl

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

module alu (
    ctrl : input InstCtrl,
    op1 : input UIntX ,
    op2 : input UIntX ,
    result: output UIntX ,
) {
    let add: UIntX = op1 + op2;
    let sub: UIntX = op1 - op2;

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

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

    always_comb {
        if ctrl.is_aluop {
```

```

        case ctrl.funct3 {
            3'b000: result = if ctrl.itype == InstType::I | ctrl.funct7 == 0 {
                add
            } else {
                sub
            };
            3'b001: result = sll;
            3'b10: result = slt;
            3'b011: result = sltu;
            3'b100: result = op1 ^ op2;
            3'b101: result = if ctrl.funct7 == 0 {
                srl
            } else {
                sra
            };
            3'b110 : result = op1 | op2;
            3'b111 : result = op1 & op2;
            default: result = 'x;
        }
    } else {
        result = add;
    }
}

```

alu モジュールには、次のポートを定義します（表 3.3）。

▼表 3.3: alu モジュールのポート定義

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

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

それ以外の命令は、足し算しか行いません。そのため、デコード時に整数演算命令とそれ以外の命令を `InstCtrl.is_aluop` で区別し、整数演算命令以外は常に足し算を行うようにしています。具体的には、opcode が OP か OP-IMM の命令の `InstCtrl.is_aluop` を 1 にしています（リスト 3.33）。

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

### 3.9.2 ALU モジュールをインスタンス化する

次に、ALU に渡すデータを用意します。 UIntX 型の変数 `op1`、`op2`、`alu_result` を定義し、 always comb ブロックで値を割り当てます（リスト 3.44）。

▼表 3.4: ALU の演算の種類

funct3	演算
3'b000	加算、または減算
3'b001	左シフト
3'b010	符号付き <=
3'b011	符号なし <=
3'b100	ビット単位 XOR
3'b101	右論理、右算術シフト
3'b110	ビット単位 OR
3'b111	ビット単位 AND

▼リスト 3.44: ALU に渡すデータの用意 (core.vverly)

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

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

割り当てるデータは、命令形式によって次のように異なります。

### R 形式、B 形式

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

### I 形式、S 形式

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

れており、レジスタの値と即値を足し合わせた値がアクセスするアドレスになります。

### U 形式、J 形式

U 形式と J 形式は、即値と PC を足した値、または即値を使う命令に使われています。 `op1` と `op2` は、それぞれ PC `inst_pc` と即値 `inst_imm` になります。J 形式は JAL 命令に利用されており、PC に即値を足した値がジャンプ先になります。U 形式は AUIPC 命令と LUI 命令に利用されています。AUIPC 命令は、PC に即値を足した値をデスティネーションレジスタに格納します。LUI 命令は、即値をそのままデスティネーションレジスタに格納します。

ALU に渡すデータを用意したので、alu モジュールをインスタンス化します（リスト 3.45）。結果を受け取る用の変数として、`alu_result` を指定します。

#### ▼ リスト 3.45: ALU のインスタンス化 (core.veryl)

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

### 3.9.3 ALU モジュールをテストする

最後に ALU が正しく動くことを確認します。

`always_ff` ブロックで、`op1` と `op2`、`alu_result` をデバッグ表示します（リスト 3.46）。

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

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

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

#### ▼ リスト 3.47: sample.hex を書き換える

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

それぞれの命令の意味は次のとおりです(表3.5)。

▼表3.5: 命令の意味

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

シミュレータを実行し、結果を確かめます(リスト3.48)。

▼リスト3.48: ALUのデバッグ

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

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

**addi x1, x0, 32**

`op1` は 0 番目のレジスタの値です。0 番目のレジスタの値は常に `0` であるため、`32'h00000000` と表示されています。`op2` は即値です。即値は 32 であるため、`32'h00000020` と表示されています。ALU の計算結果として、0 と 32 を足した結果 `32'h00000020` が表示されています。

**auipc x2, 256**

`op1` は PC です。 `op1` には、命令のアドレス `0x00000004` が表示されています。 `op2` は即値です。256 を 12bit 左にシフトした値 `32'h00100000` が表示されています。ALU の計算結果として、これを足した結果 `32'h00100004` が表示されています。

**add x3, x1, x2**

`op1` は 1 番目のレジスタの値です。1 番目のレジスタは 101 として初期化しているので、`32'h00000065` と表示されています。 `op2` は 2 番目のレジスタの値です。2 番目のレジスタは 102 として初期化しているので、`32'h00000066` と表示されています。ALU の計算結果として、これを足した結果 `32'h000000cb` が表示されています。

## 3.10 レジスタに結果を書き込む

CPU はレジスタから値を読み込み、計算して、レジスタに結果の値を書き戻します。レジスタに値を書き戻すことを、値をライトバック (write-back) すると呼びます。

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

### 3.10.1 ライトバック処理を実装する

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

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

#### ▼ リスト 3.49: ライトバック処理の実装 (core.veryl)

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

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

```
    }
```

### 3.10.2 ライトバック処理をテストする

デバッグ表示用の always\_ff ブロックで、ライトバック処理の概要をデバッグ表示します（リスト 3.50）。処理している命令がライトバックする命令のときにのみ、\$display システムタスクを呼び出します。

▼リスト 3.50: ライトバックのデバッグ表示 (core.veryl)

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

シミュレータを実行し、結果を確かめます（リスト 3.51）。

▼リスト 3.51: ライトバックのデバッグ

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

**auipc x2, 256**

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

**add x3, x1, x2**

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

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

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

## ▼リスト 3.52: レジスタの初期化をやめる (core.veryl)

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

## 3.11 ロード命令とストア命令の実装

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

▼表 3.6: RV32I のロード命令、ストア命令

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

ロード命令は I 形式、ストア命令は S 形式です。これらの命令で指定するメモリのアドレスは、rs1 と即値の足し算です。ALU に渡すデータが rs1 と即値になっていることを確認してください（リスト 3.44）。ストア命令は、rs2 の値をメモリに格納します。

### 3.11.1 LW、SW 命令を実装する

8 ビット、16 ビット単位で読み書きを行う命令の実装は少し大変です。まず、32 ビット単位で

読み書きを行う LW 命令と SW 命令を実装します。

### memunit モジュールの作成

メモリ操作を行うモジュールを、`src/memunit.veryl` に記述します（リスト 3.53）。

#### ▼リスト 3.53: memunit.veryl

```

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

module memunit (
    clk   : input  clock      ,
    rst   : input  reset      ,
    valid : input  logic      ,
    is_new: input  logic      , // 命令が新しく供給されたかどうか
) >うか
    ctrl  : input  InstCtrl  , // 命令のInstCtrl
    addr  : input  Addr      , // アクセスするアドレス
    rs2   : input  UIntX    , // ストア命令で書き込むデータ
) >タ
    rdata : output UIntX    , // ロード命令の結果 (stall = 1)
) >0のときに有効)
    stall : output logic    , // メモリアクセス命令が完了したか
) >ていない
    membus: modport membus_if:<MEM_DATA_WIDTH, XLEN>::master, // メモリとのinterface
) {

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

    var state: State;

    var req_wen  : logic      ;
    var req_addr : Addr       ;
    var req_wdata: logic<MEM_DATA_WIDTH>;

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

```

```

        default      : 0,
    };
}

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

```

memunit モジュールでは、命令がメモリにアクセスする命令のとき、ALU から受け取ったアドレスをメモリに渡して操作を実行します。

命令がメモリにアクセスする命令かどうかは `inst_is_memop` 関数で判定します。ストア命令のとき、命令の形式は S 形式です。ロード命令のとき、デコーダは `InstCtrl.is_load` を 1 にしています（リスト 3.33）。

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

## State::Init

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

### **State::WaitReady**

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

### State::WaitValid

メモリの処理が終了した (`rvalid`) とき、状態を `State::Init` に移動します。

メモリにアクセスする命令のとき、memunit モジュールは `Init` → `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止める必要があります。

CPU の実行が止まることを、CPU がストール (Stall) すると呼びます。メモリアクセス中のストールを実現するために、memunit モジュールには処理中かどうかを表す `stall` フラグを実装しています。有効な命令が供給されているとき、`state` やメモリの状態に応じて、次のように `stall` の値を決定します (表 3.7)。

▼ 表 3.7: `stall` の値の決定方法

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



#### アドレスが 4 バイトに整列されていない場合の動作

memory モジュールはアドレスの下位 2 ビットを無視するため、`addr` の下位 2 ビットが `00` ではない、つまり、4 で割り切れないアドレスに対して LW 命令か SW 命令を実行する場合、memunit モジュールは正しい動作をしません。この問題は後の章で対応するため、全てのロードストア命令は、アクセスするビット幅で割り切れるアドレスにしかアクセスしないということにしておきます。

### memunit モジュールのインスタンス化

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

まず、命令が供給されていることを示す信号 `inst_valid` と、命令が現在のクロックで供給されたことを示す信号 `inst_is_new` を作成します (リスト 3.54)。命令が供給されているかどうかは `if_fifo_rvalid` と同値です。これを機に、`if_fifo_rvalid` を使用しているところを `inst_valid` に置き換えましょう。

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

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

次に、`inst_is_new` の値を更新します (リスト 3.55)。命令が現在のクロックで供給されたかどうかは、FIFO の `rvalid` と `rready` を観測することでわかります。`rvalid` が 1 のとき、`rready` が 1 なら、次のクロックで供給される命令は新しく供給される命令です。`rready` が 0 な

ら、次のクロックで供給されている命令は現在のクロックと同じ命令になります。`rvalid` が 0 のとき、次のクロックで供給される命令は常に新しく供給される命令になります(次のクロックで `rvalid` が 1 かどうかは考えません)。

#### ▼リスト 3.55: `inst_is_new` の実装 (core.veryl)

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

`memunit` モジュールをインスタンス化する前に、メモリとの接続方法を考える必要があります。`core` モジュールには、メモリとの接続点として `membus` ポートが存在します。しかし、これは命令フェッチに使用されているため、`memunit` モジュールのために使用できません。また、`memory` モジュールは同時に 2 つの操作を受け付けられません。

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

まず、`core` モジュールに命令フェッチ用のポート `i_membus` と、ロードストア命令用のポート `d_membus` の 2 つのポートを用意します(リスト 3.56)。

#### ▼リスト 3.56: `core` モジュールのポート定義 (core.veryl)

```
module core (
    clk      : input  clock ,
    rst      : input  reset ,
    i_membus: modport membus_if::<ILEN, XLEN>::master ,
    d_membus: modport membus_if::<MEM_DATA_WIDTH, XLEN>::master ,
) {
```

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

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

```
// FIFOに2個以上空きがあるとき、命令をフェッチする
i_membus.valid = if_fifo.wready_two;
i_membus.addr  = if_pc;
i_membus.wen   = 0;
i_membus.wdata = 'x; // wdataは使用しない
```

次に、`top` モジュールでの調停を実装します(リスト 3.58)。新しく `i_membus` と `d_membus` をインスタンス化し、それを `membus` と接続します。

## ▼ リスト 3.58: メモリへのアクセス要求の調停 (top.vhdl)

```

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

var memarb_last_i: logic;

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

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

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

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

```

調停の仕組みは次のとおりです。

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

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

core モジュールとの接続を次のように変更します（リスト 3.59）。

## ▼リスト 3.59: membus を 2 つに分けて接続する (top.veryl)

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

memory モジュールと memunit モジュールを接続する準備が整ったので、memunit モジュールをインスタンス化します（リスト 3.60）。

## ▼リスト 3.60: memunit モジュールのインスタンス化 (core.veryl)

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

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

**memunit モジュールの処理待ちとライトバック**

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

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

## ▼リスト 3.61: memunit モジュールの処理が終わるのを待つ (core.veryl)

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

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

次に、ロード命令の結果をレジスタにライトバックします（リスト 3.62）。ライトバック処理では、命令がロード命令のとき（`inst_ctrl.is_load`）、`memu_rdata` を `wb_data` に設定します。

## ▼リスト 3.62: memunit モジュールの結果をライトバックする (core.veryl)

```
let rd_addr: logic<5> = inst_bits[11:7];
let wb_data: UIntX    = if inst_ctrl.is_lui {
```

```

    inst_imm
} else if inst_ctrl.is_load {
    memu_rdata
} else {
    alu_result
};
```

ところで、現在のコードでは memunit の処理が終了していないときも値をライトバックし続けています。レジスタへのライトバックは命令の実行が終了したときのみで良いため、次のようにコードを変更します（リスト 3.63）。

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

```

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

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

▼ リスト 3.64: ライトバックするときにのみデバッグ表示する (core.veryl)

```

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

## LW、SW 命令のテスト

LW 命令と SW 命令が正しく動作していることを確認するために、デバッグ表示に次のコードを追加します（リスト 3.65）。

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

```

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

ここからのテストは実行するクロック数が多くなります。そこで、ログに何クロック目かを表示することでログを読みやすくします（リスト 3.66）。

▼ リスト 3.66: 何クロック目かを出力する (core.veryl)

```

var clock_count: u64;

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

```
$display("  itype      : %b", inst_ctrl.itype);
```

LW、SW 命令のテストのために、`src/sample.hex` を次のように変更します（リスト 3.67）。

▼ リスト 3.67: テスト用のプログラムを記述する (`sample.hex`)

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

プログラムは次のようになっています（表 3.8）。

▼ 表 3.8: メモリに格納する命令

アドレス	命令	意味
0x00000000	lw x10, 0x20(x0)	x10 に、アドレスが 0x20 のデータを読み込む
0x00000004	addi x11, x0, 0x400	x11 = 0x400
0x00000008	sw x11, 0x20(x0)	アドレス 0x20 に x11 の値を書き込む
0x0000000c	lw x12, 0x20(x0)	x12 に、アドレスが 0x20 のデータを読み込む

アドレス `0x00000020` には、データ `32'hdeadbeef` を格納しています。1 つ目の命令で `32'hdeadbeef` が読み込まれ、3 つ目の命令で `32'h00000400` を書き込み、4 つ目の命令で `32'h00000400` が読み込まれます。

シミュレータを実行し、結果を確かめます（リスト 3.68）。

▼ リスト 3.68: LW、SW 命令のテスト

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

#
#          4
00000000 : 02002503
  itype    : 000010
  imm     : 00000020
  rs1[ 0] : 00000000
  rs2[ 0] : 00000000
  op1     : 00000000
  op2     : 00000020
  alu res : 00000020
mem stall : 1 ← LW命令でストールしている
mem rdata : 02b02023
...
#
#          6
```

```

00000000 : 02002503
  itype      : 000010
  imm       : 00000020
  rs1[ 0]   : 00000000
  rs2[ 0]   : 00000000
  op1       : 00000000
  op2       : 00000020
  alu res   : 00000020
  mem stall : 0 ← LWが終わったので0になった
  mem rdata : deadbeef
  reg[10] <= deadbeef ← 0x20の値が読み込まれた
...
#           13
0000000c : 02002603
  itype      : 000010
  imm       : 00000020
  rs1[ 0]   : 00000000
  rs2[ 0]   : 00000000
  op1       : 00000000
  op2       : 00000020
  alu res   : 00000020
  mem stall : 0
  mem rdata : 00000400
  reg[12] <= 00000400 ← 書き込んだ値が読み込まれた

```

### 3.11.2 LB、LBU、LH、LHU 命令を実装する

LB と LBU と SB 命令は 8 ビット単位、LH と LHU と SH 命令は 16 ビット単位でロードストアを行う命令です。まず、ロード命令を実装します。ロード命令は 32 ビット単位でデータを読み込み、その結果の一部を切り取ることで実装できます。

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

▼表 3.9: ロード命令の funct3

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

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

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

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

funct3 を case 文で分岐し、アドレスの下位ビットを見ることで、命令とアドレスに応じた値を rdata に設定します（リスト 3.70）。

## ▼リスト 3.70: rdata をアドレスと読み込みサイズに応じて変更する (memunit.veryl)

```
// loadの結果
rdata = case ctrl.funct3[1:0] {
    2'b00 : case addr[1:0] {
        0      : {sext & D[7] repeat W - 8, D[7:0]},
        1      : {sext & D[15] repeat W - 8, D[15:8]},
        2      : {sext & D[23] repeat W - 8, D[23:16]},
        3      : {sext & D[31] repeat W - 8, D[31:24]},
        default: 'x,
    },
    2'b01 : case addr[1:0] {
        0      : {sext & D[15] repeat W - 16, D[15:0]},
        2      : {sext & D[31] repeat W - 16, D[31:16]},
        default: 'x,
    },
    2'b10 : D,
    default: 'x,
};
```

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

### 3.11.3 SB、SH 命令を実装する

次に、SB、SH 命令を実装します。

#### memory モジュールで書き込みマスクをサポートする

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

まず、membus\_if インターフェースに、書き込む場所をバイト単位で示す信号 `wmask` を追加します（リスト 3.71、リスト 3.72、リスト 3.73）。

## ▼リスト 3.71: wmask の定義 (membus\_if.veryl)

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

## ▼リスト 3.72: modport master に wmask を追加する (membus\_if.veryl)

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

## ▼リスト 3.73: modport slave に wmask を追加する (membus\_if.veryl)

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

`wmask` には、書き込む部分を `1`、書き込まない部分を `0` で指定します。このような挙動をする値を、書き込みマスクと呼びます。バイト単位で指定するため、`wmask` の幅は `DATA_WIDTH / 8` ビットです。

次に、memory モジュールで書き込みマスクをサポートします（リスト 3.74）。

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

```
module memory::<DATA_WIDTH: const, ADDR_WIDTH: const> #(
    param FILEPATH_IS_ENV: logic = 0, // FILEPATHが環境変数名かどうか
    param FILEPATH        : string = "", // メモリの初期化用ファイルのパス、または環境変数名
) (
    clk      : input   clock           ,
    rst      : input   reset          ,
    membus: modport membus_if::<DATA_WIDTH, ADDR_WIDTH>::slave,
) {
    type DataType = logic<DATA_WIDTH>     ;
    type MaskType = logic<DATA_WIDTH / 8>;
    var mem: DataType [2 ** ADDR_WIDTH];
    // 書き込みマスクをDATA_WIDTHに展開した値
    var wmask_expand: DataType;
    always_comb {
        for i: u32 in 0..DATA_WIDTH {
            wmask_expand[i] = wmask_saved[i / 8];
        }
    }
    initial {
        // memを初期化する
        if FILEPATH != "" {
            if FILEPATH_IS_ENV {
                $readmemh(util::get_env(FILEPATH), mem);
            } else {
                $readmemh(FILEPATH, mem);
            }
        }
    }
}
```

```
}

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

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

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

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

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

```
    }
}
```

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

### State::Ready

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

### State::WriteValid

書き込みマスクつきの書き込みを行う。状態を `State::Ready` に移動する。

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

top モジュールの調停処理で、 `wmask` も調停します（リスト 3.75）。

#### ▼ リスト 3.75: wmask の調停 (top.veryl)

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

### memunit モジュールの実装

memory モジュールが書き込みマスクをサポートしたので、memunit モジュールで `wmask` を設定します。

`req_wmask` レジスタを作成し、 `membus.wmask` と接続します（リスト 3.76、リスト 3.77）。

#### ▼ リスト 3.76: req\_wmask の定義 (memunit.veryl)

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

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

```
// メモリアクセス
membus.valid = state == State::WaitReady;
membus.addr = req_addr;
membus.wen = req_wen;
membus.wdata = req_wdata;
membus.wmask = req_wmask;
```

always\_ff の中で、`req_wmask` の値を設定します。それぞれの命令のとき、`wmask` がどうなるかを確認してください（リスト 3.78、リスト 3.79）。

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

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

▼ リスト 3.79: メモリにアクセスする命令のとき、`wmask` を設定する (memunit.veryl)

```
req_wmask = case ctrl.funct3[1:0] {
    2'b00 : 4'b1 << addr[1:0], ← SB命令のとき、アドレス下位2ビット分だけ1を左シフトする
    2'b01 : case addr[1:0] { ← SH命令のとき
        2      : 4'b1100, ← 上位2バイトに書き込む
        0      : 4'b0011, ← 下位2バイトに書き込む
        default: 'x,
    },
    2'b10 : 4'b1111, ← SW命令のとき、全体に書き込む
    default: 'x,
};
```

### 3.11.4 LB、LBU、LH、LHU、SB、SH 命令をテストする

簡単なテストを作成し、動作をテストします。2つテストを記載するので、正しく動いているか確認してください。

▼ リスト 3.80: src/sample\_lbh.hex

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

▼ リスト 3.81: src/sample\_sbsh.hex

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

## 3.12 ジャンプ命令、分岐命令の実装

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

▼表 3.10: ジャンプ命令、分岐命令

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

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

### 3.12.1 JAL、JALR 命令を実装する

まず、無条件ジャンプを実装します。

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

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

inst\_decoder モジュールは、JAL 命令か JALR 命令のとき、InstCtrl.rwb\_en を 1、InstCtrl.is\_aluop を 0、InstCtrl.is\_jump を 1 としてデコードします。

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

#### 無条件ジャンプの実装

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

## ▼リスト 3.82: pc + 4 を書き込む (core.veryl)

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

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

## ▼リスト 3.83: control\_hazard と control\_hazard\_pc\_next の定義 (core.veryl)

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

## ▼リスト 3.84: control\_hazard と control\_hazard\_pc\_next の割り当て (core.veryl)

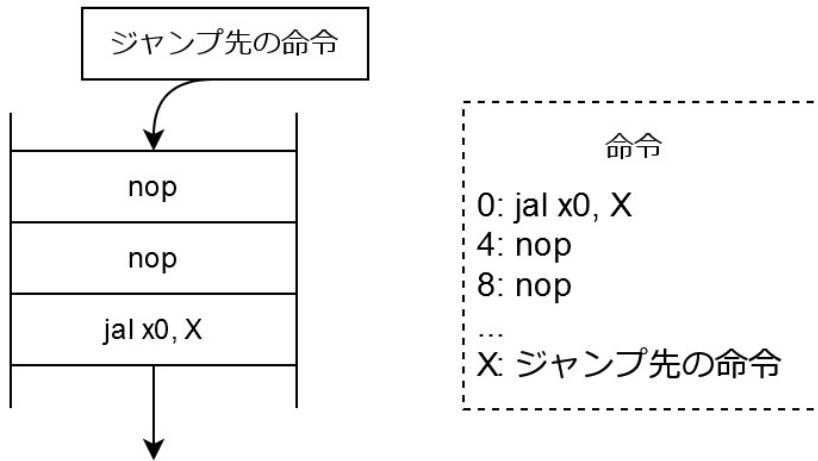
```
assign control_hazard      = inst_valid && inst_ctrl.is_jump;
assign control_hazard_pc_next = alu_result & ~1;
```

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

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

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

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



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

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

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

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

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

`flush` が 1 のとき、`head` と `tail` を 0 に初期化することで FIFO を空にします（リスト 3.87、リスト 3.88）。

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

```
always_ff {
    if_reset {
        rvalid = 0;
    } else {
        if flush {
            rvalid = 0;
        } else {
            if wready && wvalid {
                rdata  = wdata;
                rvalid = 1;
            } else if rready {
```

```
        rvalid = 0;
    }
}
```

▼リスト 3.88: flush が 1 のとき、FIFO を空にする (fifo.vryl、WIDTH!=1)

```

always_ff {
    if_reset {
        head = 0;
        tail = 0;
    } else {
        if flush {
            head = 0;
            tail = 0;
        } else {
            if wready && wvalid {
                mem[tail] = wdata;
                tail      = tail + 1;
            }
            if rready && rvalid {
                head = head + 1;
            }
        }
    }
}

```

core モジュールで、control hazard と flush を接続し、FIFO をリセットします（リスト 3.89）。

▼リスト 3.89: ジャンプ命令のとき、FIFO をリセットする (core.vyrl)

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

## 無条件ジャンプのテスト

簡単なテストを作成し、動作をテストします（リスト 3.90、リスト 3.91）。

▼ U3 to 3.90: 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
00000067 // 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 3.91: テストの実行

```
$ make build
$ make sim
$ obj_dir/sim src/sample_jump.hex 17
#
#          4
00000000 : 0100006f
    reg[ 0] <= 00000004 ← rd = PC + 4
#
#          8
00000010 : 01800093 ← 0x00 → 0x10にジャンプしている
    reg[ 1] <= 00000018
#
#          9
00000014 : 00808067
    reg[ 0] <= 00000018 ← rd = PC + 4
#
#          13
00000020 : fe1ff06f ← 0x14 → 0x20にジャンプしている
    reg[ 0] <= 00000024 ← rd = PC + 4
#
#          17
00000000 : 0100006f ← 0x20 → 0x00にジャンプしている
    reg[ 0] <= 00000004
```

### 3.12.2 条件分岐命令を実装する

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

▼表 3.11: 条件分岐命令と funct3

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

### 条件分岐の実装

分岐の条件が成立するかどうかを判定するモジュールを作成します。`src/brunit.vyrl` を作成し、次のように記述します（リスト 3.92）。

▼リスト 3.92: brunit.vyrl

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


```

```

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

    always_comb {
        case funct3 {
            3'b000 : take = beq;
            3'b001 : take = !beq;
            3'b100 : take = blt;
            3'b101 : take = !blt;
            3'b110 : take = bltu;
            3'b111 : take = !bltu;
            default: take = 0;
        }
    }
}

```

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

brunit モジュールを、core モジュールでインスタンス化します（リスト 3.93）。命令が B 形式のとき、`op1` は `rs1_data`、`op2` は `rs2_data` になっていることを確認してください（リスト 3.44）。

#### ▼ リスト 3.93: brunit モジュールのインスタンス化 (core.veryl)

```

var brunit_take: logic;

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

```

命令が条件分岐命令で `brunit_take` が 1 のとき、次の PC を  $PC + \text{即値}$  にします（リスト 3.94、リスト 3.95）。

#### ▼ リスト 3.94: 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 3.95: 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 & ~1
};

```

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

## 条件分岐命令のテスト

条件分岐命令を実行するとき、分岐の成否をデバッグ表示します。デバッグ表示を行っている always\_ff ブロック内に、次のコードを追加します（リスト 3.96）。

## ▼ リスト 3.96: 分岐判定のデバッグ表示 (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 3.97、リスト 3.98）。

## ▼ リスト 3.97: 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 3.98: テストの実行

```

$ make build
$ make sim
$ obj_dir/sim src/sample_br.hex 15
#
#          4
00000000 : 00100093 ← x1に1を代入
#
#          5
00000004 : 10100063
  op1      : 00000000
  op2      : 00000001
  br take  : 0 ← x0 != x1なので不成立
#
#          6
00000008 : 00101863
  op1      : 00000000
  op2      : 00000001

```

```
br take    : 1 ← x0 != x1なので成立
#                      10
00000018 : 0000d063 ← 0x08 → 0x18にジャンプ
br take    : 1 ← x1 > x0なので成立
#                      14
00000018 : 0000d063 ← 0x18 → 0x18にジャンプ
br take    : 1
```

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



### 実装していない RV32I の命令

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

# 第 4 章

## Zicsr 拡張の実装

### 4.1 CSR とは何か?

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

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

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

▼表 4.1: Zicsr 拡張に定義されている命令

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

\*1 それぞれの機能は実装するときに解説します

## 4.2 CSR命令のデコード

まず、Zicsrに定義されている命令(表4.1)をデコードします。

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

### ▼リスト4.1: opcode用の定数の定義(eei.veryl)

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

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

### ▼リスト4.2: is\_csrを追加する(corectrl.veryl)

```
// 制御に使うフラグ用の構造体
struct InstCtrl {
    itype      : InstType, // 命令の形式
    rwb_en     : logic,   // レジスタに書き込むかどうか
    is_lui     : logic,   // LUI命令である
    is_aluop   : logic,   // ALUを利用する命令である
    is_jump    : logic,   // ジャンプ命令である
    is_load    : logic,   // ロード命令である
    is_csr     : logic,   // CSR命令である
    funct3    : logic <3>, // 命令のfunct3フィールド
    funct7    : logic <7>, // 命令のfunct7フィールド
}
```

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

### ▼リスト4.3: OP\_SYSTEMとis\_csrを追加する(inst\_decoder.veryl)

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

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

るということにしています。

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

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

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

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

## 4.3 csrunit モジュールの実装

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

### 4.3.1 csrunit モジュールを作成する

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

▼リスト 4.4: csrunit.veryl

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

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

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

▼表4.3: csrunitモジュールのポート定義

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

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

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

```
var csru_rdata: UIntX;

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

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

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

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

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

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

```

} else {
    alu_result
};

```

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

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

```

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

```

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

### 4.3.2 mtvec レジスタを実装する

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

#### mtvec レジスタ、トラップ



▲図 4.1: mtvec のエンコーディング [9]

mtvec レジスタは、仕様書 [10] に定義されています。mtvec は、MXLEN ビットの WARL 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問題ありません。WARL は Write Any Values, Reads Legal Values の略です。その名の通り、好きな値を書き込みますが読み出すときには合法な値<sup>\*2</sup>になっているという認識で問題ありません。

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

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

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

<sup>\*2</sup> 合法な値とは実装がサポートしている有効な値のことです

<sup>\*3</sup> トラップや例外、割り込みは Volume I の 1.6Exceptions, Traps, and Interrupts に定義されています

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

### mtvec レジスタの実装

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

```

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

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

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

```

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

```

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

### 4.3.3 csrunitモジュールをテストする

`mtvec` レジスタの書き込み、読み込みができるることを確認します。

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

#### ▼リスト 4.13: sample\_csr.hex

```

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

```

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

シミュレータを実行し、結果を確かめます (リスト 4.14)。

## ▼リスト4.14: mtvecの読み込み/書き込みテストの実行

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

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

## 4.4 ECALL命令の実装

せっかくmtvecレジスタを実装したので、これを使う命令を実装します。

### 4.4.1 ECALL命令とは何か？

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

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

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

▼表4.4: 権限レベルとECALLによる例外

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

### mcause、mepc レジスタ

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

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

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

## 4.4.2 トランプを実装する

それでは、ECALL命令とトランプの仕組みを実装します。

### 定数の定義

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

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

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

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

#### ▼リスト4.16: CsrCause型の定義(csrunit.veryl)

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

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

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

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

### mepcとmcauseレジスタの実装

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

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

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

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

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

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

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

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

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

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

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

```

    }
}
}
```

## 例外の実装

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

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

```

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

それぞれの用途は次の通りです。

#### **pc**

現在処理している命令のアドレスを受け取ります。

例外が発生するとき、mepcにPCを格納するために使います。

#### **rd\_addr**

現在処理している命令のrdの番号を受け取ります。

命令がECALL命令かどうかを判定するために使います。

#### **raise\_trap**

例外が発生するとき、値を1にします。

#### **trap\_vector**

例外が発生するとき、ジャンプ先のアドレスを出力します。

csrunitモジュールの中身を実装する前に、coreモジュールに例外発生時の動作を実装します。

csrunitモジュールと接続するための変数を定義してcsrunitモジュールと接続します(リスト4.23、リスト4.24)。

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

```

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

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

```

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

```

次に、トラップするときにトラップ先にジャンプさせます。

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

## ▼リスト4.25: 例外の発生時にジャンプさせる(core.veryl)

```

assign control_hazard = inst_valid && (
    csrunit_raise_trap ||
    inst_ctrl.is_jump ||
    inst_is_br(inst_ctrl) && brunit_take
);
assign control_hazard_pc_next = if csrunit_raise_trap {
    csrunit_trap_vector ←トラップするとき、trap_vectorに飛ぶ
} else if inst_is_br(inst_ctrl) {
    inst_pc + inst_imm
} else {
    alu_result & ~1
};

```

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

▲図4.2: ECALL命令のフォーマット[6]

それでは、csrunitモジュールにトラップの処理を実装します。

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

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

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

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

## ▼リスト4.27: 例外とトラップの判定(csrunit.veryl)

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

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

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

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

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

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

#### 4.4.3 ECALL命令をテストする

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

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

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

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

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

#### ▼リスト4.30: sample\_ecall.hex

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

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

シミュレータを実行し、結果を確かめます（リスト4.31）。

#### ▼リスト4.31: ECALL命令のテストの実行

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

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

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

## 4.5 MRET命令の実装

MRET命令<sup>\*4</sup>は、トラップ先からトラップ元に戻るための命令です。MRET命令を実行すると、mepcレジスタに格納されたアドレスにジャンプします<sup>\*5</sup>。例えば、権限のあるOSから権限のないユーザー空間に戻るために利用します。

### 4.5.1 MRET命令を実装する

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

▲図4.3: MRET命令のフォーマット [11]

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

▼リスト4.32: MRET命令の判定(csrunit.veryl)

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

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

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

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

.....  
例外が優先

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

\*4 MRET命令はVolume IIの3.3.2. Trap-Return Instructionsに定義されています

\*5 他のCSRや権限レベルが実装されている場合は、他にも行うことがあります

## 4.5.2 MRET命令をテストする

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

### ▼リスト4.34: sample\_mret.hex

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

シミュレータを実行し、結果を確かめます(リスト4.35)。

### ▼リスト4.35: MRET命令のテストの実行

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

MRET命令によってmepcにジャンプすることを確認できます。

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

# 第 5 章

## riscv-tests によるテスト

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

### 5.1 riscv-tests とは何か？

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

GitHub の riscv-software-src/riscv-tests<sup>2</sup> からソースコードをダウンロードできます。

### 5.2 riscv-tests のビルド



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

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

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

#### 5.2.1 riscv-tests をビルドする

まず、riscv-tests を clone します（リスト 5.1）。

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

<sup>2</sup> <https://github.com/riscv-software-src/riscv-tests>

## ▼リスト 5.1: riscv-tests の clone

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

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

## ▼リスト 5.2: riscv-tests/env/p/link.ld

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

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

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

## ▼リスト 5.3: riscv-tests のビルド

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

core/test に share ディレクトリが作成されます。

## 5.2.2 成果物を\$readmemhで読み込める形式に変換する

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

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

## ▼リスト 5.4: test/bin2hex.py

```
import sys

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

```

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

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

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

# #0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<sup>\*3</sup>です。CPU は ELF を直接に実行する機能を持っていないため、`riscv64-unknown-elf-objcopy` を利用して、ELF ファイルを余計な情報を取り除いたバイナリファイルに変換します（リスト 5.5）。

#### ▼リスト 5.5: ELF ファイルをバイナリファイルに変換する

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

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

---

<sup>\*3</sup> ELF(Executable and Linkable Format) とは実行可能ファイルの形式です

## ▼リスト 5.6: バイナリファイルを HEX ファイルに変換する

```
$ find share/ -type f -name "*.bin" -exec sh -c "python3 bin2hex.py 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 `test/share/riscv-tests/isa/rv32ui-p-add.dump` を読んでみます（リスト 5.7）。`rv32ui-p-add.dump` は、`rv32ui-p-add` のダンプファイルです。

▼リスト 5.7: `rv32ui-p-add.dump`

```
Disassembly of section .text.init:

00000000 <_start>:
 0: 0500006f          j      50 <reset_vector>

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

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

```

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

```

命令のテストは次の流れで実行されます。

1. `_start` : `reset_vector` にジャンプする。
2. `reset_vector` : 各種状態を初期化する。
3. `test_*` : テストを実行する。命令の結果がおかしかったら `fail` にジャンプする。最後まで正常に実行できたら `pass` にジャンプする。
4. `fail`、`pass` : テストの成否をレジスタに書き込み、`trap_vector` にジャンプする。
5. `trap_vector` : `write_tohost` にジャンプする。
6. `write_tohost` : テスト結果をメモリに書き込む。ここでループする。

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

#### ▼リスト 5.8: riscv-tests/env/p/link.ld

```

OUTPUT_ARCH( "riscv" )
ENTRY(_start)

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

```

## 5.4 テストの終了検知

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

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

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

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

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

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

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

```
module top #(
    param MEMORY_FILEPATH_IS_ENV: bit      = 1           ,
    param MEMORY_FILEPATH          : string = "MEMORY_FILE_PATH",
) (
    clk: input clock,
    rst: input reset,
    #[ifdef(TEST_MODE)]
    test_success: output bit,
)
```

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

## 5.5 テストの実行

試しに ADD 命令のテストを実行してみましょう。ADD 命令のテストの HEX ファイルは `test/share/riscv-tests/isa/rv32ui-p-add.bin.hex` です。

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

▼リスト 5.11: ADD 命令の riscv-tests を実行する

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

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

## 5.6 複数のテストの自動実行

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

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

▼リスト 5.12: test.py

```
import argparse
import os
```

<sup>\*4</sup> 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

```
import subprocess

parser = argparse.ArgumentParser()
parser.add_argument("sim_path", help="path to simulator")
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("exec " + cmd, shell=True, stdout=no, stderr=no)
        try:
            p.wait(None if args.time_limit == 0 else args.time_limit)
            success = p.returncode == 0
        except:
            pass
        finally:
            p.terminate()
            p.kill()
    print(("PASS" if success else "FAIL") + " : " + file_name)
    return (file_name, success)

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

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

    res_strs = []
```

```

res_statuses = []

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

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

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

print(statusText)

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

```

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

次のオプションの引数が存在します。

#### -r

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

#### -e 拡張子

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

#### -o ディレクトリ

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

#### -t 時間

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

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

#### ▼リスト5.13: tb\_verilator.cpp

```

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

```

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

#### ▼ リスト 5.14: `rv32ui-p` から始まるテストを実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv32ui-p-
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-lh.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sb.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sltiu.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sh.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-bltu.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-or.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-sra.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-xor.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-addi.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-srai.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-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 個のテストに成功しました。テストの詳細な結果は `results` ディレクトリに格納されています。

`rv32ui-p-ma_data` は、ロードストアするサイズに整列されていないアドレスへのロードストア

命令のテストです。これは後の章で例外として対処するため、今は無視します。

# 第 6 章

## RV64I の実装

これまでに、RISC-V の 32 ビットの基本整数命令セットである RV32I の CPU を実装しました。RISC-V には 64 ビットの基本整数命令セットとして RV64I が定義されています。本章では、RV32I の CPU を RV64I にアップグレードします。

では、具体的に RV32I と RV64I は何が違うのでしょうか？まず、RV64I では XLEN が 32 ビットから 64 ビットに変更され、レジスタの幅や各種演算命令の演算の幅が 64 ビットになります。それに伴い、32 ビット幅での整数演算を行う命令、64 ビット幅でロードストアを行う命令が追加されます（表 6.1）。また、演算の幅が 64 ビットに広がるだけではなく、一部の命令の動作が少し変わります（表 6.2）。

▼ 表 6.1: RV64I で追加される命令

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

▼ 表 6.2: RV64I で変更される命令

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

実装のテストには riscv-tests を利用します。RV64I 向けのテストは `rv64ui-p-` から始まるテストです。命令を実装するたびにテストを実行することで、命令が正しく実行できていることを確認します。

## 6.1 XLEN の変更

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

### ▼ リスト 6.1: XLEN を変更する (eei.veryl)

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

### 6.1.1 SLL[I]、SRL[I]、SRA[I] 命令を変更する

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

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

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

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

### 6.1.2 LUI、AUIPC 命令を変更する

RV32I では、LUI 命令は 32 ビットの即値をそのままレジスタに格納する命令として定義されています。これが RV64I では、32 ビットの即値を 64 ビットに符号拡張した値を格納する命令に変更されます。AUIPC 命令も同様で、即値に PC を足す前に、即値を 64 ビットに符号拡張します。

この対応ですが、XLEN を 64 に変更した時点ですでに完了しています（リスト 6.3）。そのため、コードの変更の必要はありません。

### ▼ リスト 6.3: U 形式の即値は XLEN ビットに拡張されている (inst\_decoder.veryl)

```
let imm_u: UIntX = {bits[31] repeat XLEN - $bits(imm_u_g) - 12, imm_u_g, 12'b0};
```

### 6.1.3 CSR を変更する

`MXLEN`(=XLEN) が 64 ビットに変更されると、CSR の幅も 64 ビットに変更されます。そのため、`mtvec`、`mepc`、`mcause` レジスタの幅を 64 ビットに変更する必要があります。

しかし、mtvec、mepc、mcause レジスタは XLEN ビットのレジスタ (`UIntX`) として定義しているため、変更の必要はありません。また、mtvec、mepc、mcause レジスタは MXLEN を基準に定義されており、RV32I から RV64I に変わってもフィールドに変化はないため、対応は必要ありません。

唯一、書き込みマスクの幅を広げる必要があります（リスト 6.4）。

#### ▼ リスト 6.4: CSR の書き込みマスクの幅を広げる (csrunit.veryl)

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

### 6.1.4 LW 命令を変更する

LW 命令は 32 ビットの値をロードする命令です。RV64I では、LW 命令の結果が 64 ビットに符号拡張されるようになります。これに対応するため、memunit モジュールの `rdata` の割り当てる LW 部分を変更します（リスト 6.5）。

#### ▼ リスト 6.5: LW 命令のメモリの読み込み結果を符号拡張する (memunit.veryl)

```
2'b10 : {D[31] repeat W - 32, D[31:0]},
```

また、XLEN が 64 に変更されたことで、幅を `MEM_DATA_WIDTH` (=32) として定義している `req_wdata` の代入文のビット幅が左右で合わなくなってしまっています。ビット幅を合わせるために、rs2 の下位 `MEM_DATA_WIDTH` ビットだけを切り取ります（リスト 6.6）。

#### ▼ リスト 6.6: 左辺と右辺でビット幅を合わせる (memunit.veryl)

```
case state {
    State::Init: if is_new & inst_is_memop(ctrl) {
        state      = State::WaitReady;
        req_wen   = inst_is_store(ctrl);
        req_addr  = addr;
        req_wdata = rs2[MEM_DATA_WIDTH - 1:0] << {addr[1:0], 3'b0};
```

### 6.1.5 riscv-tests でテストする

#### RV32I 向けのテストの実行

まず、RV32I 向けのテストが正しく動くことを確認します（リスト 6.7）。

#### ▼ リスト 6.7: RV32I 向けのテストを実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv32ui-p-
...
PASS : ~/core/test/share/riscv-tests/isa/rv32ui-p-srl.bin.hex
Test Result : 40 / 40
```

RV32I 向けのテストにすべて成功しました。しかし、`rv32ui-p-ma_data` は失敗するはずです（リスト 5.14）。これは、riscv-tests の RV32I 向けのテストは、XLEN が 64 のときはテストを実行せずに成功とするためです（リスト 6.8）。

▼ リスト 6.8: `rv32ui-p-add` は XLEN が 64 のときにテストせずに成功する (`rv32ui-p-add.dump`)

```
00000050 <reset_vector>:
...
13c: 00100513      li    a0,1 ← a0 = 1
140: 01f51513      slli   a0,a0,0x1f ← a0を31ビット左シフト
144: 00054c63      bltz   a0,15c <reset_vector+0x10c> ← a0が0より小さかったらジンプ
>ジャンプ
148: 0ff0000f      fence
14c: 00100193      li    gp,1 ← gp=1（テスト成功）にする
150: 05d00893      li    a7,93
154: 00000513      li    a0,0
158: 00000073      ecall ← trap_vectorにジャンプして終了
```

riscv-tests は、a0 に 1 を代入した後、a0 を 31 ビット左シフトします。XLEN が 32 のとき、a0 の最上位ビット（符号ビット）が 1 になり、a0 は 0 より小さくなります。XLEN が 64 のとき、a0 の符号は変わらないため、a0 は 0 より大きくなります。これを利用して、XLEN が 32 ではないときは `trap_vector` にジャンプして、テスト成功として終了しています。

## RV64I 向けのテストの実行

それでは、RV64I 向けのテストを実行します（リスト 6.9）。RV64I 向けのテストは名前が `rv64ui-p-` から始まります、

▼ リスト 6.9: RV64I 向けのテストを実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-
...
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-add.bin.hex
...
Test Result : 14 / 52
```

ADD 命令のテストを含む、ほとんどのテストに失敗してしまいました。これは riscv-tests のテストが、まだ未実装の命令を含むためです（リスト 6.10）。

▼ リスト 6.10: ADD 命令のテストは未実装の命令 (ADDIW 命令) を含む (`rv64ui-p-add.dump`)

```
0000000000000000208 <test_7>:
208: 00700193      li    gp,7
20c: 800005b7      lui   a1,0x80000
210: ffff8637      lui   a2,0xfffff8
214: 00c58733      add   a4,a1,a2
218: ffff03b7      lui   t2,0xfffff0
21c: fff3839b      addiw t2,t2,-1 # ffffffffffffffeffff <_end+0xfffffffffffffedff>
>f>
220: 00f39393      slli   t2,t2,0xf
```

224: 46771063	bne a4,t2,684 <fail>
---------------	----------------------

ということで、失敗していることを気にせずに実装を進めます。

## 6.2 ADD[I]W、SUBW 命令の実装

RV64Iでは、ADD命令は64ビット単位で演算する命令になり、32ビットの加算をするADDW命令とADDIW命令が追加されます。同様に、SUB命令は64ビット単位の演算になり、32ビットの減算をするSUBW命令が追加されます。32ビットの演算結果は符号拡張します。

### 6.2.1 ADD[I]W、SUBW 命令をデコードする

imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW

▲図 6.1: ADDW、ADDIW、SUBW 命令のフォーマット [6]

ADDW命令とSUBW命令はR形式で、opcodeはOP-32(7'b0111011)です。ADDIW命令はI形式で、opcodeはOP-IMM-32(7'b0011011)です。

まず、eeiパッケージにopcodeの定数を定義します(リスト6.11)。

#### ▼リスト 6.11: opcodeを定義する(eei.veryl)

```
const OP_OP_32    : logic<7> = 7'b0111011;
const OP_OP_IMM_32: logic<7> = 7'b0011011;
```

次に、InstCtrl構造体に、32ビット単位で演算を行う命令であることを示すis\_op32フラグを追加します(リスト6.12)。

#### ▼リスト 6.12: is\_op32を追加する(corectrl.veryl)

```
struct InstCtrl {
    itype   : InstType , // 命令の形式
    rwb_en : logic    , // レジスタに書き込むかどうか
    is_lui : logic    , // LUI命令である
    is_aluop: logic   , // ALUを利用する命令である
    is_op32 : logic   , // OP-32またはOP-IMM-32である
    is_jump : logic   , // ジャンプ命令である
    is_load : logic   , // ロード命令である
    is_csr  : logic   , // CSR命令である
    funct3 : logic    <3>, // 命令のfunct3フィールド
    funct7 : logic    <7>, // 命令のfunct7フィールド
}
```

inst\_decoder モジュールの `InstCtrl` と即値を生成している部分を変更します（リスト 6.13、リスト 6.14）。これでデコードは完了です。

▼ リスト 6.13: OP-32、OP-IMM-32 の InstCtrl の生成 (inst\_decoder.veryl)

```
is_op32を追加
ctrl = {case op {
    OP_LUI      : {InstType::U, T, T, F, F, F, F, F},
    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},
    OP_OP_IMM_32: {InstType::I, T, F, T, T, F, F, F},
    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 6.14: OP-IMM-32 の即値の生成 (inst\_decoder.veryl)

```
imm = case op {
    OP_LUI, OP_AUIPC      : imm_u,
    OP_JAL                 : imm_j,
    OP_JALR, OP_LOAD       : imm_i,
    OP_OP_IMM, OP_OP_IMM_32: imm_i,
    OP_BRANCH              : imm_b,
    OP_STORE               : imm_s,
    default                : 'x,
};
```

## 6.2.2 ALU に ADDW、SUBW を実装する

制御フラグを生成できたので、それに応じて 32 ビットの ADD と SUB を計算します。

まず、32 ビットの足し算と引き算の結果を生成します（リスト 6.15）。

▼ リスト 6.15: 32 ビットの足し算と引き算をする (alu.veryl)

```
let add32: UInt32 = op1[31:0] + op2[31:0];
let sub32: UInt32 = op1[31:0] - op2[31:0];
```

次に、フラグによって演算結果を選択する関数 `sel_w` を作成します（リスト 6.16）。この関数は、`is_op32` が 1 なら `value32` を 64 ビットに符号拡張した値、0 なら `value64` を返します。

▼ リスト 6.16: 演算結果を選択する関数を作成する (alu.veryl)

```
function sel_w (
    is_op32: input logic ,
    value32: input UInt32,
```

```

    value64: input UInt64,
) -> UInt64 {
    if is_op32 {
        return {value32[msb] repeat 32, value32};
    } else {
        return value64;
    }
}

```

sel\_w 関数を使用し、alu モジュールの演算処理を変更します。case 文の足し算と引き算の部分を次のように変更します（リスト 6.17）。

#### ▼リスト 6.17: 32 ビットの演算結果を選択する (alu.veryl)

```

3'b000: result = if ctrl.itype == InstType::I | ctrl.funct7 == 0 {
    sel_w(ctrl.is_op32, add32, add)
} else {
    sel_w(ctrl.is_op32, sub32, sub)
};

```

### 6.2.3 ADD[I]W、SUBW 命令をテストする

RV64I 向けのテストを実行して、結果ファイルを確認します（リスト 6.18、リスト 6.19）。

#### ▼リスト 6.18: RV64I 向けのテストを実行する

```

$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-

```

#### ▼リスト 6.19: テストの実行結果 (results/result.txt)

```

Test Result : 42 / 52
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ld.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-lwu.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ma_data.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-sd.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-slliw.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-sllw.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-sraiw.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-sraw.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-srliw.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-srlw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-add.bin.hex
...
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-addiw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-addw.bin.hex
...
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-subw.bin.hex
...

```

ADDIW、ADDDW、SUBW だけでなく、未実装の命令以外のテストにも成功しました。

## 6.3 SLL[I]W、SRL[I]W、SRA[I]W 命令の実装

RV64I では、SLL[I]、SRL[I]、SRA[I] 命令は rs1 を 0 ~ 63 ビットシフトする命令になり、rs1 の下位 32 ビットを 0 ~ 31 ビットシフトする SLL[I]W、SRL[I]W、SRA[I]W 命令が追加されます。32 ビットの演算結果は符号拡張します。

000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

▲図 6.2: SLL[I]W、SRL[I]W、SRA[I]W 命令のフォーマット [6]

SLL[I]W、SRL[I]W、SRA[I]W 命令のフォーマットは、RV32I の SLL[I]、SRL[I]、SRA[I] 命令の opcode を変えたものと同じです。SLLW、SRLW、SRAW 命令は R 形式で、opcode は OP-32 です。SLLIW、SRLIW、SRAIW 命令は I 形式で、opcode は OP-IMM-32 です。どちらの opcode の命令も、ADD[I]W 命令と SUBW 命令の実装時にデコードが完了しています。

alu モジュールで、32 ビットのシフト演算の結果を生成します（リスト 6.20）。

### ▼リスト 6.20: 32 ビットのシフト演算をする (alu.veryl)

```
let sll32: UInt32 = op1[31:0] << op2[4:0];
let srl32: UInt32 = op1[31:0] >> op2[4:0];
let sra32: SInt32 = $signed(op1[31:0]) >>> op2[4:0];
```

生成したシフト演算の結果を sel\_w 関数で選択します。case 文のシフト演算の部分を次のように変更します（リスト 6.21）。

### ▼リスト 6.21: 32 ビットの演算結果を選択する (alu.veryl)

```
3'b001: result = sel_w(ctrl.is_op32, sll32, sll);
...
3'b101: result = if ctrl.funct7 == 0 {
    sel_w(ctrl.is_op32, srl32, srl)
} else {
    sel_w(ctrl.is_op32, sra32, sra)
};
```

### 6.3.1 SLL[I]W、SRL[I]W、SRA[I]W 命令をテストする

RV64I 向けのテストを実行し、結果ファイルを確認します（リスト 6.22、リスト 6.23）。

## ▼リスト 6.22: RV64I 向けのテストを実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-
```

## ▼リスト 6.23: テストの実行結果 (results/result.txt)

```
Test Result : 48 / 52
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ld.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-lwu.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ma_data.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-sd.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-add.bin.hex
...
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sll.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-slli.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-slliw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sllw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sra.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-srai.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sraiw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sraw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-srl.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-srli.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-srliw.bin.hex
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-srlw.bin.hex
...
...
```

SLLW、SLLIW、SRLW、SR LIW、SRAW、SRAIW 命令のテストに成功していることを確認できます。

## 6.4 LWU命令の実装

LB、LH 命令は、ロードした値を符号拡張した値をレジスタに格納します。これに対して、LBU、LHU 命令は、ロードした値をゼロで拡張した値をレジスタに格納します。

同様に、LW 命令は、ロードした値を符号拡張した値をレジスタに格納します。これに対して、RV64I では、ロードした 32 ビットの値をゼロで拡張した値をレジスタに格納する LWU 命令が追加されます。

imm[11:0]	rs1	110	rd	0000011	LWU
-----------	-----	-----	----	---------	-----

▲図 6.3: LWU命令のフォーマット [6]

LWU 命令は I 形式で、opcode は LOAD です。ロードストア命令は funct3 によって区別でき、LWU 命令の funct3 は 3'b110 です。デコード処理に変更は必要なく、メモリにアクセスする処理を変更する必要があります。

memunit モジュールの、ロードする部分を変更します。32 ビットを `rdata` に割り当てるとき、`sext` によって符号かゼロで拡張するかを選択します（リスト 6.24）。

▼ リスト 6.24: LWU 命令の実装 (memunit.veryl)

```
2'b10 : {sext & D[31] repeat W - 32, D[31:0]},
```

### 6.4.1 LWU 命令をテストする

LWU 命令のテストを実行します（リスト 6.25）。

▼ リスト 6.25: LWU 命令をテストする

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-lwu
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-lwu.bin.hex
Test Result : 1 / 1
```

## 6.5 LD、SD 命令の実装

RV64I には、64 ビット単位でロードストアを行う LD 命令と SD 命令が定義されています。

imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD

▲ 図 6.4: LD、SD 命令のフォーマット

LD 命令は I 形式で、opcode は `LOAD` です。SD 命令は S 形式で、opcode は `STORE` です。どちらの命令も funct3 は `3'b011` です。デコード処理に変更は必要ありません。

### 6.5.1 メモリの幅を広げる

現在のメモリの 1 つのデータの幅 (`MEM_DATA_WIDTH`) は 32 ビットですが、このままだと 64 ビットでロードやストアを行うときに、最低 2 回のメモリアクセスが必要です。これを 1 回のメモリアクセスで済ませるために、データの幅を 32 ビットから 64 ビットに広げます（リスト 6.26）。

▼ リスト 6.26: MEM\_DATA\_WIDTH を 64 ビットに変更する (eei.veryl)

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

### 6.5.2 命令フェッチ処理を修正する

`XLEN`、`MEM_DATA_WIDTH` が変わっても、命令の長さ (`ILEN`) は 32 ビットのままです。そのた

め、topモジュールの `i_membus.rdata` の幅は32ビットなのに対し、`membus.rdata` は64ビットになり、ビット幅が一致しません。

ビット幅を合わせて正しく命令をフェッチするために、64ビットの読み出しデータの上位32ビット、下位32ビットをアドレスの下位ビットで選択します。アドレスが8の倍数のときは下位32ビット、それ以外のときは上位32ビットを選択します。

まず、命令フェッチの要求アドレスをレジスタに格納します（リスト6.27、リスト6.28）。

▼リスト6.27: アドレスを格納するためのレジスタの定義（top.veryl）

```
var memarb_last_i : logic;
var memarb_last_iaddr: Addr ;
```

▼リスト6.28: レジスタに命令フェッチの要求アドレスを格納する（top.veryl）

```
//メモリアクセスを調停する
always_ff {
    if_reset {
        memarb_last_i      = 0;
        memarb_last_iaddr = 0;
    } else {
        if membustx.ready {
            memarb_last_i      = !d_membus.valid;
            memarb_last_iaddr = i_membus.addr;
        }
    }
}
```

このレジスタの値を利用し、`i_membus.rdata` に割り当てる値を選択します（リスト6.29）。

▼リスト6.29: アドレスによってデータを選択する（top.veryl）

```
i_membus.rdata = if memarb_last_iaddr[2] == 0 {
    membustx.rdata[31:0]
} else {
    membustx.rdata[63:32]
};
```

### 6.5.3 SD命令を実装する

SD命令の実装のためには、書き込むデータ（`wdata`）と書き込みマスク（`wmask`）を変更する必要があります。

書き込むデータはアドレスの下位2ビットではなく下位3ビット分シフトします（リスト6.30）。

▼リスト6.30: 書き込むデータの変更（memunit.veryl）

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

書き込みマスクは4ビットから8ビットに拡張されるため、アドレスの下位2ビットではなく下位3ビットで選択します（リスト6.31）。

## ▼ リスト 6.31: 書き込みマスクの変更 (memunit.veryl)

```

req_wmask = case ctrl.funct3[1:0] {
    2'b00 : 8'b1 << addr[2:0],
    2'b01 : case addr[2:0] {
        6      : 8'b11000000,
        4      : 8'b00011000,
        2      : 8'b00001100,
        0      : 8'b00000011,
        default: 'x,
    },
    2'b10 : case addr[2:0] {
        0      : 8'b00001111,
        4      : 8'b11110000,
        default: 'x,
    },
    2'b11 : 8'b11111111,
    default: 'x,
};

```

## 6.5.4 LD 命令を実装する

メモリのデータ幅が 64 ビットに広がるため、`rdata` に割り当てる値を、アドレスの下位 2 ビットではなく下位 3 ビットで選択します（リスト 6.32）。

## ▼ リスト 6.32: rdata の変更 (memunit.veryl)

```

rdata = case ctrl.funct3[1:0] {
    2'b00 : case addr[2:0] {
        0      : {sext & D[7] repeat W - 8, D[7:0]},
        1      : {sext & D[15] repeat W - 8, D[15:8]},
        2      : {sext & D[23] repeat W - 8, D[23:16]},
        3      : {sext & D[31] repeat W - 8, D[31:24]},
        4      : {sext & D[39] repeat W - 8, D[39:32]},
        5      : {sext & D[47] repeat W - 8, D[47:40]},
        6      : {sext & D[55] repeat W - 8, D[55:48]},
        7      : {sext & D[63] repeat W - 8, D[63:56]},
        default: 'x,
    },
    2'b01 : case addr[2:0] {
        0      : {sext & D[15] repeat W - 16, D[15:0]},
        2      : {sext & D[31] repeat W - 16, D[31:16]},
        4      : {sext & D[47] repeat W - 16, D[47:32]},
        6      : {sext & D[63] repeat W - 16, D[63:48]},
        default: 'x,
    },
    2'b10 : case addr[2:0] {
        0      : {sext & D[31] repeat W - 32, D[31:0]},
        4      : {sext & D[63] repeat W - 32, D[63:32]},
        default: 'x,
    },
    2'b11 : D,
    default: 'x,
};

```

```
};
```

### 6.5.5 LD、SD命令をテストする

LD、SD命令のテストを実行する前に、メモリのデータ単位が4バイトから8バイトになったため、テストのHEXファイルを4バイト単位の改行から8バイト単位の改行に変更します（リスト6.33）。

#### ▼リスト6.33: HEXファイルを8バイト単位に変更する

```
$ cd test  
$ find share/ -type f -name "*.bin" -exec sh -c "python3 bin2hex.py 8 {} > {}.hex" \;
```

riscv-testsを実行します（リスト6.34）。

#### ▼リスト6.34: RV32I、RV64Iをテストする

```
$ make build  
$ make sim VERILATOR_FLAGS="-DTEST_MODE"  
$ python3 test/test.py -r obj_dir/sim test/share/rv32ui-p-  
...  
Test Result : 40 / 40  
$ python3 test/test.py -r obj_dir/sim test/share/rv64ui-p-  
...  
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ma_data.bin.hex  
...  
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-ld.bin.hex  
PASS : ~/core/test/share/riscv-tests/isa/rv64ui-p-sd.bin.hex  
...  
Test Result : 51 / 52
```

RV64IのCPUを実装できました。

---

# 第 7 章

## CPU のパイプライン化

---

これまでの章では、同時に 1 つの命令を実行する CPU を実装しました。高機能な CPU を実装するのは面白いですが、プログラムの実行が遅くてはいけません。機能を増やす前に、一度性能のことを考えてみましょう。

### 7.1 CPU の速度

CPU の性能指標は、例えば消費電力や実行速度が考えられます。本章では、プログラムの実行速度を考えます。

#### 7.1.1 CPU の性能を考える

性能の比較にはクロック周波数やコア数などが用いられますが、プログラムの実行速度を比較する場合、プログラムの実行にかかる時間のみが絶対的な指標になります。プログラムの実行時間は、次のような式で表せます（図 7.1）

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

▲ 図 7.1: CPU 性能方程式 [12]

それぞれの用語の定義は次の通りです。

##### CPU 時間 (CPU time)

プログラムの実行のために CPU が費やした時間

##### 実行命令数

プログラムの実行で実行される命令数

##### CPI (Clock cycles Per Instruction)

プログラム全体またはプログラムの一部分の命令を実行した時の 1 命令当たりの平均クロックサイクル数

ク・サイクル数

### クロック周波数 (clock rate)

クロック・サイクル時間 (clock cycle time) の逆数

クロック・サイクル時間は、クロックが  $0 \rightarrow 1 \rightarrow 1$  になる周期のこと

今のところ、CPU は命令をスキップしたり無駄に実行することはありません。そのため、実行命令数は、プログラムを 1 命令ずつ順に実行していった時の実行命令数になります。

CPI を計測するためには、何の命令にどれだけのクロック・サイクル数がかかるかと、それぞれの命令の割合が必要です。今のところ、メモリにアクセスする命令は 3 ~ 4 クロック、それ以外の命令は 1 クロックで実行されます。命令の割合は考えないでおきます。

クロック周波数は、CPU の回路のクリティカルパスの長さによって決まります。クリティカルパスとは、組み合わせ回路の中で最も大きな遅延を持つ経路のことです。

## 7.1.2 実行速度を上げる方法を考える

CPU 性能方程式の各項に注目すると、CPU 時間を減らすためには、実行命令数を減らすか、CPI を減らすか、クロック周波数を増大させる必要があります。

### 実行命令数に注目する

実行命令数を減らすためには、コンパイラによる最適化でプログラムの命令数を減らすソフトウェア的な方法と、命令セットアーキテクチャ (ISA) を変更することで必要な命令数を減らす方法が存在します。どちらも本書の目的とするところではないので、検討しません<sup>\*1</sup>。

### CPI に注目する

CPI を減らすためには、例えどの命令も 1 クロックで実行してしまうという方法が考えられます。しかし、そのために論理回路を大きくすると、その分クリティカルパスが長くなってしまう場合があります。また、1 クロックに 1 命令しか実行しない場合、どう頑張っても CPI は 1 より小さくなりません。

CPI をより効果的に減らすためには、1 クロックで 1 つ以上の命令を実行開始し、1 つ以上の命令を実行完了すればいいです。これを実現する手法として、スーパースカラやアウトオブオーダー実行が存在します。これらの手法はずっと後の章で解説、実装します。

### クロック周波数に注目する

クロック周波数を増大させるには、クリティカルパスの長さを短くする必要があります。

今のところ、CPU は計算命令を 1 クロック (シングルサイクル) で実行します。例えば ADD 命令を実行するとき、FIFO に保存された ADD 命令をデコードし、命令のビット列をもとにレジスタの値を選択し、ALU で足し算を実行し、その結果をレジスタにライトバックします。これらを 1 クロックで実行するということは、命令が保存されている 32 ビットのレジスタと 32\*64 ビット

---

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

のレジスタファイルを入力に、64 ビットの ADD 演算の結果を出力する組み合わせ回路が存在するということです。この回路は大変に段数の深い組み合わせ回路を必要とし、長いクリティカルパスを生成する原因になります。

クロック周波数を増大させるもっとも単純な方法は、命令の処理をいくつかのステージ(段)に分割し、複数クロックで1つの命令を実行することです。複数のクロック・サイクルで命令を実行することから、この形式のCPUはマルチサイクルCPUと呼びます。

\時間(t)	t = 1	t = 2	t = 3	t = 4	t = 5	t = 6
命令1	ステージ1	ステージ2	ステージ3			
命令2				ステージ1	ステージ2	ステージ3

▲図 7.2: 命令の実行(マルチサイクル)

命令の処理をいくつかのステージに分割すると、それに合わせて回路の深さが軽減され、クロック周波数を増大させられます。

図 7.2 では、1つの命令を3クロック(ステージ)で実行しています。3クロックもかかるのであれば、CPI が3倍になり、CPU 時間が増えてしまいそうです。しかし、処理を均等な3ステージに分割できた場合、クロック周波数は3分の1になる<sup>\*2</sup>ため、それほどCPU 時間は増えません。

しかし、CPI がステージ分だけ増大してしまうのは問題です。この問題は、命令の処理を、まるで車の組立のように流れ作業で行うことで緩和できます(図 7.3)。このような処理のことを、パイプライン処理と呼びます。

\時間(t)	t = 1	t = 2	t = 3	t = 4	t = 5
命令1	ステージ1	ステージ2	ステージ3		
命令2		ステージ1	ステージ2	ステージ3	
命令3			ステージ1	ステージ2	ステージ3

▲図 7.3: 命令の実行(パイプライン処理)

本章では、CPU をパイプライン化することで性能の向上を図ります。

### 7.1.3 パイプライン処理のステージを考える

具体的に処理をどのようなステージに分割してパイプライン処理を実現すればいいでしょうか?これをるために、第3章の最初で検討したCPUの動作を振り返ります。第3章では、CPUの動作を次のように順序付けしました。

<sup>\*2</sup> 実際のところは均等に分割することはできないため、Nステージに分割してもクロック周波数はN分の1になります

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

もう少し大きな処理単位に分割しなおすと、次の 5 つの処理 (ステージ) を構成できます。ステージ名の後ろに、それぞれ対応する上のリストの処理の番号を記載しています。

### IF (Instruction Fetch) ステージ (1)

メモリから命令をフェッチします。

フェッチした命令を ID ステージに受け渡します。

### ID (Instruction Decode) ステージ (2、3)

命令をデコードし、制御フラグと即値を生成します。

生成したデータを EX ステージに渡します。

### EX (Execute) ステージ (3、4)

制御フラグ、即値、レジスタの値を利用し、ALU で計算します。

分岐判定やジャンプ先の計算も行い、生成したデータを MEM ステージに渡します。

### MEM (Memory) ステージ (5、7)

メモリにアクセスする命令と CSR 命令を処理します。

分岐命令かつ分岐が成立する、ジャンプ命令である、またはトランプが発生するとき、IF、ID、EX ステージにある命令を無効化して、ジャンプ先を IF ステージに伝えます。メモリのロード、CSR の読み込み結果を WB ステージに渡します。

### WB (WriteBack) ステージ (6)

ALU の演算結果、メモリや CSR の読み込み結果など、命令の処理結果をレジスタに書き込みます。

MEM ステージではジャンプするときに IF、ID、EX ステージにある命令を無効化します。これは、IF、ID、EX ステージにある命令は、ジャンプによって実行されない命令になるためです。パイプラインのステージにある命令を無効化することを、パイプラインをフラッシュ (flush) すると呼びます。

IF、ID、EX、MEM、WB の 5 段の構成を、**5 段パイプライン** (Five Stage Pipeline) と呼ぶことがあります。

### CSR を MEM ステージで処理する

上記の 5 段のパイプライン処理では、CSR の処理を MEM ステージで行っています。これはいったいなぜでしょうか?

CPU には ECALL 命令による例外しか実装してしないため、EX ステージで CSR の処理を行ってしても問題ありません。しかし、他の例外、例えばメモリアクセスに伴う例外を実装するとき、問題が生じます。

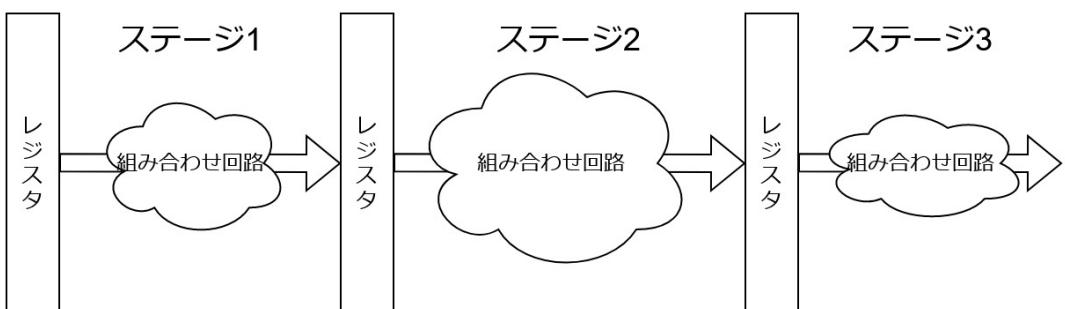
メモリアクセスに起因する例外が発生するのは MEM ステージです。このとき、EX ステージで CSR の処理を行っていて、EX ステージに存在する命令が mtvec レジスタに書き込む CSRRW 命令だった場合、本来は MEM ステージで発生した例外によって実行されないはずである CSRRW 命令によって、既に mtvec レジスタが書き換えられているかもしれません。これを復元する処理を書くことはできますが、MEM ステージ以降で CSR を処理することでもこの事態を回避できるため、MEM ステージで CSR を処理しています。

## 7.2 パイプライン処理の実装

### 7.2.1 ステージに分割する準備をする

それでは、CPUをパイプライン化します。

パイプライン処理では、複数のステージがそれぞれ違う命令を処理します。そのため、それぞれのステージのために、現在処理している命令を保持するためのレジスタ（パイプラインレジスタ）を用意します。



▲図 7.4: パイプライン処理の概略図

まず、処理を複数ステージに分割する前に、既存の変数の名前を変更します。

core モジュールでは、命令をフェッチする処理に使う変数の名前の先頭に `if_` 、 FIFO から取り出した命令の情報を表す変数の名前の先頭に `inst_` をつけています。

命令をフェッチする処理は IF ステージに該当するため、`if_` から始まる変数はこ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、どのステージの変数か分からなくなります。IF ステージの次は ID ステージであるため、変数が ID ステージのものであることを示す名前に変えてしまします。

## ▼リスト 7.1: 変数名を変更する (core.verv1)

```

let ids_valid      : logic      = if_fifo_rvalid;
var ids_is_new    : logic      ; // 命令が現在のクロックで供給されたかどうか
let ids_pc        : Addr       = if_fifo_rdata.addr;
let ids_inst_bits: Inst       = if_fifo_rdata.bits;
var ids_ctrl      : InstCtrl;
var ids_imm       : UIntX     ;

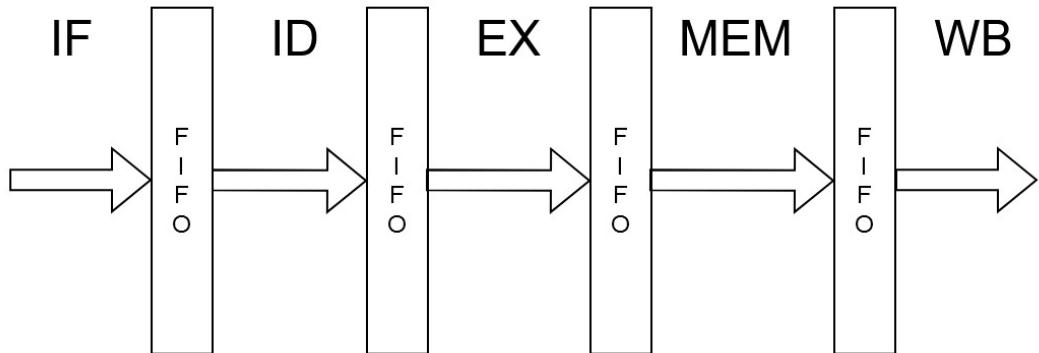
```

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

## 7.2.2 FIFO を作成する

命令フェッチ処理とそれ以降の処理は、それぞれ独立して動作しています。実は既に CPU は、IF と ID ステージ (命令フェッチ以外の処理を行うステージ) の 2 ステージのパイプライン処理を行っています。

IF ステージと ID ステージは FIFO で区切られており、FIFO のレジスタを経由して命令の受け渡しを行います。これと同様に、5 ステージのパイプライン処理の実装では、それぞれのステージを FIFO で接続します (図 7.5)。ただし、FIFO のサイズは 1 とします。この場合、FIFO はただの 1 つのレジスタです。

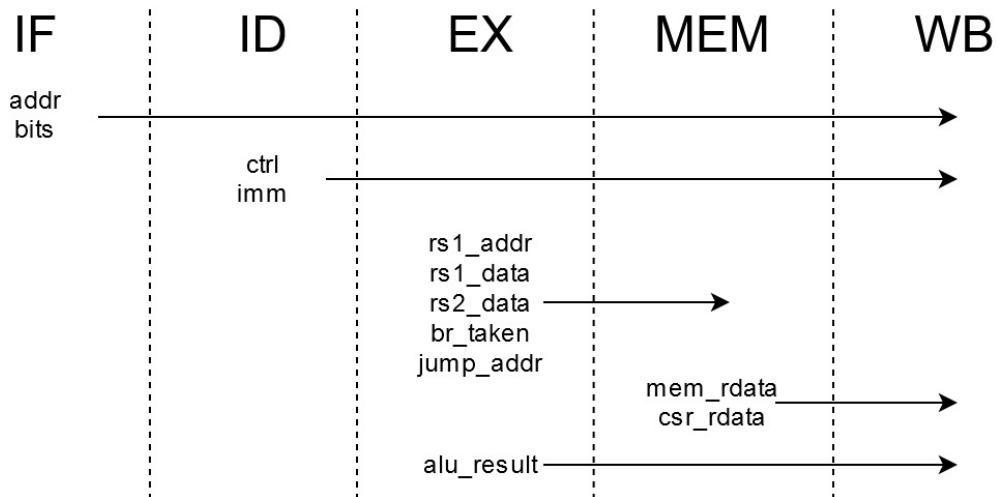


▲図 7.5: FIFO を利用したパイプライン処理

IF から ID への FIFO は存在するため、ID から EX、EX から MEM、MEM から WB への FIFO を作成します。

### 構造体の定義

まず、FIFO に格納するデータの型を定義します。それぞれのフィールドが存在する区間は図 7.6 の通りです。



▲図 7.6: 構造体のフィールドの生存区間

## ▼リスト 7.2: ID → EX の間の FIFO のデータ型 (core.veryl)

```
struct exq_type {
    addr: Addr      ,
    bits: Inst      ,
    ctrl: InstCtrl ,
    imm : UIntX    ,
}
```

ID ステージは、IF ステージから命令のアドレスと命令のビット列を受け取ります。命令のビット列をデコードして、制御フラグと即値を生成し、EX ステージに渡します（リスト 7.2）。

## ▼リスト 7.3: EX → MEM の間の FIFO のデータ型 (core.veryl)

```
struct memq_type {
    addr      : Addr      ,
    bits     : Inst      ,
    ctrl     : InstCtrl ,
    imm      : UIntX    ,
    alu_result: UIntX   ,
    rs1_addr : logic <5>,
    rs1_data : UIntX   ,
    rs2_data : UIntX   ,
    br_taken : logic   ,
    jump_addr: Addr    ,
}
```

EX ステージは、ID ステージで生成された制御フラグと即値を受け取ります。整数演算命令のとき、レジスタの値を使って計算します。分岐命令のとき、分岐判定を行います。CSR やメモリアクセスで rs1 と rs2 を利用するため、演算の結果とともに MEM ステージに渡します（リスト 7.3）。

## ▼リスト 7.4: MEM → WB の間の FIFO のデータ型 (core.veryl)

```
struct wbq_type {
    addr      : Addr      ,
    bits      : Inst      ,
    ctrl      : InstCtrl ,
    imm       : UIntX    ,
    alu_result: UIntX   ,
    mem_rdata : UIntX   ,
    csr_rdata : UIntX   ,
}
```

MEM ステージは、メモリのロード結果と CSR の読み込みデータを生成し、WB ステージに渡します（リスト 7.4）。

WB ステージでは、命令がライトバックする命令のとき、即値、ALU の計算結果、メモリのロード結果、CSR の読み込みデータから 1 つを選択し、レジスタに値を書き込みます。

構造体のフィールドの生存区間が図 7.6 のようになっている理由が分かったでしょうか？

**FIFO のインスタンス化**

FIFO と接続するための変数を定義し、FIFO をインスタンス化します（リスト 7.5、リスト 7.6）。

`DATA_TYPE` パラメータには先ほど作成した構造体を設定します。FIFO のデータ個数は 1 であるため、`WIDTH` パラメータには `1` を設定します<sup>\*3</sup>。`mem_wb_fifo` の `flush` は `0` にしています。

## ▼リスト 7.5: FIFO と接続するための変数を定義する (core.veryl)

```
// ID -> EXのFIFO
var exq_wready: logic      ;
var exq_wvalid: logic      ;
var exq_wdata : exq_type;
var exq_rready: logic      ;
var exq_rvalid: logic      ;
var exq_rdata : exq_type;

// EX -> MEMのFIFO
var memq_wready: logic      ;
var memq_wvalid: logic      ;
var memq_wdata : memq_type;
var memq_rready: logic      ;
var memq_rvalid: logic      ;
var memq_rdata : memq_type;

// MEM -> WBのFIFO
var wbq_wready: logic      ;
var wbq_wvalid: logic      ;
var wbq_wdata : wbq_type;
var wbq_rready: logic      ;
var wbq_rvalid: logic      ;
var wbq_rdata : wbq_type;
```

<sup>\*3</sup> FIFO のデータ個数は `2 ** WIDTH - 1` です

## ▼リスト 7.6: FIFO のインスタンス化 (core.veryl)

```

inst id_ex_fifo: fifo #(
    DATA_TYPE: exq_type,
    WIDTH     : 1      ,
) (
    clk          ,
    rst          ,
    flush : control_hazard,
    wready: exq_wready   ,
    wvalid: exq_wvalid   ,
    wdata : exq_wdata    ,
    rready: exq_rready   ,
    rvalid: exq_rvalid   ,
    rdata : exq_rdata    ,
);

inst ex_mem_fifo: fifo #(
    DATA_TYPE: memq_type,
    WIDTH     : 1      ,
) (
    clk          ,
    rst          ,
    flush : control_hazard,
    wready: memq_wready   ,
    wvalid: memq_wvalid   ,
    wdata : memq_wdata    ,
    rready: memq_rready   ,
    rvalid: memq_rvalid   ,
    rdata : memq_rdata    ,
);

inst mem_wb_fifo: fifo #(
    DATA_TYPE: wbq_type,
    WIDTH     : 1      ,
) (
    clk          ,
    rst          ,
    flush : 0      ,
    wready: wbq_wready,
    wvalid: wbq_wvalid,
    wdata : wbq_wdata ,
    rready: wbq_rready,
    rvalid: wbq_rvalid,
    rdata : wbq_rdata ,
);

```

### 7.2.3 IF ステージを実装する

まず、IF ステージを実装します。… といっても、既に IF ステージ (=命令フェッチ処理) は独立に動くものとして実装されているため、手を加える必要はありません。

リスト 7.7 のようなコメントを挿入すると、ステージの処理を書いている区間が分かりやすくなる

ります。ID、EX、MEM、WB ステージを実装するときにも同様のコメントを挿入し、ステージの処理のコードをまとまった場所に配置しましょう。

▼リスト 7.7: IF ステージが始まることを示すコメントを挿入する (core.veryl)

```
////////////////////////////// IF Stage /////////////////////////////////  
  
var if_pc : Addr ;  
...
```

## 7.2.4 ID ステージを実装する

ID ステージでは、命令をデコードします。既に `ids_ctrl` と `ids_imm` には、デコード結果の制御フラグと即値が割り当てられているため、既存のコードの変更は必要ありません。

デコード結果は EX ステージに渡します。EX ステージにデータを渡すには、`exq_wdata` にデータを割り当てます（リスト 7.8）。

▼リスト 7.8: EX ステージに値を渡す (core.veryl)

```
always_comb {  
    // ID -> EX  
    if_fifo_rready = exq_wready;  
    exq_wvalid = if_fifo_rvalid;  
    exq_wdata.addr = if_fifo_rdata.addr;  
    exq_wdata.bits = if_fifo_rdata.bits;  
    exq_wdata.ctrl = ids_ctrl;  
    exq_wdata.imm = ids_imm;  
}
```

ID ステージにある命令は、EX ステージが命令を受け入れられるとき（`exq_wready`）、ID ステージを完了して EX ステージに処理を進められます。この仕組みは、`if_fifo_rready` に `exq_wready` を割り当てることで実現できます。

最後に、命令が現在のクロックで供給されたかどうかを示す変数 `id_is_new` は必要ないため削除します（リスト 7.9）。

▼リスト 7.9: `ids_is_new` を削除する (core.veryl)

```
var ids_is_new : logic ;
```

## 7.2.5 EX ステージを実装する

EX ステージでは、整数演算命令のときは ALU で計算し、分岐命令のときは分岐判定を行います。

まず、EX ステージに存在する命令の情報を `exq_rdata` から取り出します（リスト 7.10）。

▼リスト 7.10: 変数の定義 (core.veryl)

```
let exs_valid : logic = exq_rvalid;  
let exs_pc : Addr = exq_rdata.addr;
```

```
let exs_inst_bits: Inst      = exq_rdata.bits;
let exs_ctrl      : InstCtrl = exq_rdata.ctrl;
let exs_imm       : UIntX    = exq_rdata.imm;
```

次に、EX ステージで扱う変数の名前を変更します。変数の名前に `exs_` をつけます（リスト 7.11）。

#### ▼リスト 7.11: 変数名を変更する (core.veryl)

```
// レジスタ番号
let exs_rs1_addr: logic<5> = exs_inst_bits[19:15];
let exs_rs2_addr: logic<5> = exs_inst_bits[24:20];

// ソースレジスタのデータ
let exs_rs1_data: UIntX = if exs_rs1_addr == 0 {
    0
} else {
    regfile[exs_rs1_addr]
};

let exs_rs2_data: UIntX = if exs_rs2_addr == 0 {
    0
} else {
    regfile[exs_rs2_addr]
};

// ALU
var exs_op1        : UIntX;
var exs_op2        : UIntX;
var exs_alu_result: UIntX;

always_comb {
    case exs_ctrl.iotype {
        InstType::R, InstType::B: {
            exs_op1 = exs_rs1_data;
            exs_op2 = exs_rs2_data;
        }
        InstType::I, InstType::S: {
            exs_op1 = exs_rs1_data;
            exs_op2 = exs_imm;
        }
        InstType::U, InstType::J: {
            exs_op1 = exs_pc;
            exs_op2 = exs_imm;
        }
        default: {
            exs_op1 = 'x;
            exs_op2 = 'x;
        }
    }
}

inst alum: alu (
    ctrl  : exs_ctrl      ,
    ...
```

```

op1    : exs_op1      ,
op2    : exs_op2      ,
result: exs_alu_result,
);

var exs_brunit_take: logic;

inst bru: brunit (
  funct3: exs_ctrl.funct3,
  op1    : exs_op1      ,
  op2    : exs_op2      ,
  take   : exs_brunit_take,
);

```

最後に、MEM ステージに命令とデータを渡します。MEM ステージにデータを渡すために、

`memq_wdata` にデータを割り当てます（リスト 7.12）。

#### ▼ リスト 7.12: MEM ステージにデータを渡す (core.veryl)

```

always_comb {
  // EX -> MEM
  exq_rready      = memq_wready;
  memq_wvalid     = exq_rvalid;
  memq_wdata.addr = exq_rdata.addr;
  memq_wdata.bits = exq_rdata.bits;
  memq_wdata.ctrl = exq_rdata.ctrl;
  memq_wdata.imm  = exq_rdata.imm;
  memq_wdata.rs1_addr = exs_rs1_addr;
  memq_wdata.rs1_data = exs_rs1_data;
  memq_wdata.rs2_data = exs_rs2_data;
  memq_wdata.alu_result = exs_alu_result;
  ←ジャンプ命令、または、分岐命令かつ分岐が成立するとき、1にする
  memq_wdata.br_taken = exs_ctrl.is_jump || inst_is_br(exs_ctrl) && exs_brunit_take;
  memq_wdata.jump_addr = if inst_is_br(exs_ctrl) {
    exs_pc + exs_imm ←分岐命令の分岐先アドレス
  } else {
    exs_alu_result & ~1 ←ジャンプ命令のジャンプ先アドレス
  };
}

```

`br_taken` には、ジャンプ命令かどうか、または分岐命令かつ分岐が成立するか、という条件を割り当てます。`jump_addr` には、分岐命令、またはジャンプ命令のジャンプ先アドレスを割り当てます。MEM ステージではこれを用いてジャンプと分岐を処理します。

EX ステージにある命令は、MEM ステージが命令を受け入れられるとき（`memq_wready`）、EX ステージを完了して MEM ステージに処理を進められます。この仕組みは、`exq_rready` に `memq_wready` を割り当てることで実現できます。

## 7.2.6 MEM ステージを実装する

MEM ステージでは、メモリにアクセスする命令と CSR 命令を処理します。また、ジャンプ命

令、分岐命令かつ分岐が成立、またはトランプが発生するとき、次に実行する命令のアドレスを変更します。

ロードストア命令でメモリにアクセスしているとき、EX ステージから MEM ステージに別の命令の処理を進めることはできず、パイプライン処理は止まってしまいます。パイプライン処理を進められない状態のことをパイプラインハザード (pipeline hazard) と呼びます。

まず、MEM ステージに存在する命令の情報を `memq_rdata` から取り出します (リスト 7.13)。MEM ステージでは、csrunit モジュールに、命令が現在のクロックで MEM ステージに供給されたかどうかの情報を渡します。そのため、変数 `mem_is_new` を定義しています。

#### ▼ リスト 7.13: 変数の定義 (core.veryl)

```
var mems_is_new : logic      ;
let mems_valid  : logic      = memq_rvalid;
let mems_pc    : Addr        = memq_rdata.addr;
let mems_inst_bits: Inst     = memq_rdata.bits;
let mems_ctrl   : InstCtrl   = memq_rdata.ctrl;
let mems_rd_addr : logic <5> = mems_inst_bits[11:7];
```

`mem_is_new` には、`id_is_new` の更新に利用していたコードを利用します (リスト 7.14)。

#### ▼ リスト 7.14: mem\_is\_new の更新 (core.veryl)

```
always_ff {
    if_reset {
        mems_is_new = 0;
    } else {
        if memq_rvalid {
            mems_is_new = memq_rready;
        } else {
            mems_is_new = 1;
        }
    }
}
```

次に、MEM モジュールで使う変数に合わせて、memunit モジュールと csrunit モジュールのポートに割り当てる変数名を変更します (リスト 7.15)。

#### ▼ リスト 7.15: 変数名を変更する (core.veryl)

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

inst memu: memunit (
    clk           ,
    rst           ,
    valid : mems_valid      ,
    is_new: mems_is_new     ,
    ctrl  : mems_ctrl       ,
    addr  : memq_rdata.alu_result,
    rs2   : memq_rdata.rs2_data ,
```

```

rdata : memu_rdata      ,
stall : memu_stall     ,
membus: d_membus       ,
);

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

inst csru: csrunit (
    clk           ,
    rst           ,
    valid        : mems_valid   ,
    pc            : mems_pc     ,
    ctrl          : mems_ctrl   ,
    rd_addr       : mems_rd_addr,
    csr_addr      : mems_inst_bits[31:20],
    rs1          : if mems_ctrl.funct3[2] == 1 && mems_ctrl.funct3[1:0] != 0 {
        {1'b0 repeat XLEN - $bits(memq_rdata.rs1_addr), memq_rdata.rs1_addr} // rs1を0で拡張する
    } else {
        memq_rdata.rs1_data
    },
    rdata        : csru_rdata,
    raise_trap  : csru_raise_trap,
    trap_vector : csru_trap_vector,
);

```

フェッチ先が変わったことを表す変数 `control_hazard` と、新しいフェッチ先を示す信号 `control_hazard_pc_next` では、EX ステージで計算したデータと CSR ステージのトラップ情報を利用します（リスト 7.16）。

#### ▼リスト 7.16: ジャンプの判定処理 (core.veryl)

```

assign control_hazard      = mems_valid && (csru_raise_trap || mems_ctrl.is_jump || memq_rdata.br_taken);
assign control_hazard_pc_next = if csru_raise_trap {
    csru_trap_vector
} else {
    memq_rdata.jump_addr
};

```

ジャンプ命令の後ろの余計な命令を実行しないために、`control_hazard` が 1 になったとき、ID、EX、MEM ステージに命令を供給する FIFO をフラッシュします。`control_hazard` が 1 になるとき、MEM ステージの処理は完了しています。後述しますが、WB ステージの処理は必ず 1 クロックで終了します。そのため、フラッシュするとき、MEM ステージにある命令は必ず WB ステージに移動します。

最後に、WB ステージに命令とデータを渡します（リスト 7.17）。WB ステージにデータを渡すために、`wbq_wdata` にデータを割り当てます

## ▼リスト 7.17: WB ステージにデータを渡す (core.veryl)

```
always_comb {
    // MEM -> WB
    memq_rready      = wbq_wready && !memu_stall;
    wbq_wvalid        = memq_rvalid && !memu_stall;
    wbq_wdata.addr   = memq_rdata.addr;
    wbq_wdata.bits   = memq_rdata.bits;
    wbq_wdata.ctrl   = memq_rdata.ctrl;
    wbq_wdata.imm    = memq_rdata.imm;
    wbq_wdata.alu_result = memq_rdata.alu_result;
    wbq_wdata.mem_rdata = memu_rdata;
    wbq_wdata.csr_rdata = csru_rdata;
}
```

MEM ステージにある命令は、memunit モジュールが処理中ではなく (`!memu_stall`)、WB ステージが命令を受け入れられるとき (`wbq_wready`)、MEM ステージを完了して WB ステージに処理を進められます。この仕組みは、`memq_rready` と `wbq_wvalid` を確認してください。

### 7.2.7 WB ステージを実装する

WB ステージでは、命令の結果をレジスタにライトバックします。WB ステージが完了したら命令の処理は終わりなので、命令を破棄します。

まず、WB ステージに存在する命令の情報を `wbq_rdata` から取り出します (リスト 7.18)。

## ▼リスト 7.18: 変数の定義 (core.veryl)

```
let wbs_valid : logic = wbq_rvalid;
let wbs_pc : Addr = wbq_rdata.addr;
let wbs_inst_bits: Inst = wbq_rdata.bits;
let wbs_ctrl : InstCtrl = wbq_rdata.ctrl;
let wbs_imm : UIntX = wbq_rdata.imm;
```

次に、WB ステージで扱う変数名を変更します。変数名に `wbs_` をつけます (リスト 7.19)。

## ▼リスト 7.19: 変数名を変更する (core.veryl)

```
let wbs_rd_addr: logic<5> = wbs_inst_bits[11:7];
let wbs_wb_data: UIntX = if wbs_ctrl.is_lui {
    wbs_imm
} else if wbs_ctrl.is_jump {
    wbs_pc + 4
} else if wbs_ctrl.is_load {
    wbq_rdata.mem_rdata
} else if wbs_ctrl.is_csr {
    wbq_rdata.csr_rdata
} else {
    wbq_rdata.alu_result
};

always_ff {
    if wbs_valid && wbs_ctrl.rwb_en {
```

```

        regfile[wbs_rd_addr] = wbs_wb_data;
    }
}

```

最後に、命令を FIFO から取り出します。WB ステージでは命令を複数クロックで処理することはなく、WB ステージの次のステージを待つ必要もありません。`wbq_rready` に 1 を割り当てることで、常に FIFO から命令を取り出します（リスト 7.20）。

#### ▼リスト 7.20: 命令を FIFO から取り出す (core.veryl)

```

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

```

これで、IF、ID、EX、MEM、WB ステージを作成できました。

### 7.2.8 デバッグのために情報を表示する

今までには同時に 1 つの命令しか処理していませんでしたが、これからは全てのステージで別の命令を処理することになります。デバッグ表示を変更しておきましょう。

リスト 7.21 のように、デバッグ表示の `always_ff` ブロックを変更します。

#### ▼リスト 7.21: 各ステージの情報をデバッグ表示する (core.veryl)

```

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

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

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

        $display("IF -----");
        $display("    pc : %h", if_pc);
        $display(" is req : %b", if_is_requested);
        $display(" pc req : %h", if_pc_requested);
        $display("ID -----");
        if ids_valid {
            $display("    %h : %h", ids_pc, if_fifo_rdata.bits);
            $display("    itype : %b", ids_ctrl.itype);
            $display("    imm   : %h", ids_imm);
        }
        $display("EX -----");
        if exs_valid {
            $display("    %h : %h", exq_rdata.addr, exq_rdata.bits);
            $display("    op1     : %h", exs_op1);
        }
    }
}

```

```

$display(" op2      : %h", exs_op2);
$display(" alu      : %h", exs_alu_result);
if inst_is_br(exs_ctrl) {
    $display(" br take : ", exs_brunit_take);
}
$display("MEM ----");
if mems_valid {
    $display(" %h : %h", memq_rdata.addr, memq_rdata.bits);
    $display(" mem stall : %b", memu_stall);
    $display(" mem rdata : %h", memu_rdata);
    if mems_ctrl.is_csr {
        $display(" csr rdata : %h", csru_rdata);
        $display(" csr trap  : %b", csru_raise_trap);
        $display(" csr vec   : %h", csru_trap_vector);
    }
    if memq_rdata.br_taken {
        $display(" JUMP TO   : %h", memq_rdata.jump_addr);
    }
}
$display("WB ----");
if wbs_valid {
    $display(" %h : %h", wbq_rdata.addr, wbq_rdata.bits);
    if wbs_ctrl.rwb_en {
        $display(" reg[%d] <= %h", wbs_rd_addr, wbs_wb_data);
    }
}
}

```

### 7.2.9 パイプライン処理をテストする

それでは、`riscv-tests` を実行してみましょう。試しに、RV64I の ADD のテストを実行します。

▼リスト 7.22: パイプライン処理のテスト

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-add.bin.hex
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-add.bin.hex
Test Result : 0 / 1
```

おや？ テストに失敗してしまいました。一体何が起きているのでしょうか？

### 7.3 データ依存の対処

### 7.3.1 正しく動かないプログラムを確認する

実は、ただ IF、ID、EX、MEM、WB ステージに処理を分割するだけでは、正しく命令を実行できません。例えば、リスト 7.23 のようなプログラムは正しく動きません。

`test/sample_datahazard.hex` を作成し、次のように記述します（リスト 7.23）。

▼ リスト 7.23: sample\_datahazard.hex

```
0010811300100093 // 0: addi x1, x0, 1      4: addi x2, x1, 1
```

このプログラムでは、 $x_1$  に  $x_0 + 1$  を代入した後、 $x_2$  に  $x_1 + 1$  を代入します。シミュレータを実行し、どのように実行されるかを確かめます（リスト 7.24）。

▼ リスト 7.24: sample\_datahazard.hex を実行する

```
$ make build
$ make sim
$ ./obj_dir/sim test/sample_datahazard.hex 7
...
#           5
ID -----
 0000000000000004 : 00108113
  itype : 000010
  imm   : 0000000000000001
EX -----
  0000000000000000 : 00100093
  op1    : 0000000000000000 ← x0
  op2    : 0000000000000001 ← 即値
  alu    : 0000000000000001 ← ゼロレジスタ + 1 = 1

#           6
ID -----
  0000000000000008 : 00000000
  itype : 000000
  imm   : 0000000000000000
EX -----
  0000000000000004 : 00108113
  op1    : 0000000000000000 ← x1
  op2    : 0000000000000001 ← 即値
  alu    : 0000000000000001 ← x1 + 1 = 2のはずだが1になっている

MEM -----
  0000000000000000 : 00100093
  ...
...
```

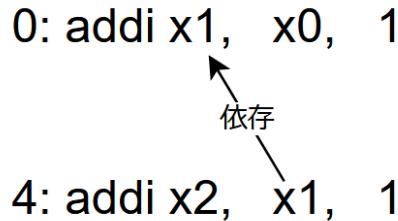
ログを確認すると、アドレス 0 の命令で  $x_1$  が 1 になっているはずですが、アドレス 4 の命令で  $x_1$  を読み込むときに  $x_1$  は 0 になっています。

この問題は、まだアドレス 0 の命令の結果がレジスタファイルに書き込まれていないのに、アドレス 4 の命令でレジスタファイルで結果を読み出しているために発生しています。

### 7.3.2 データ依存とは何か？

ある命令 A の実行結果の値を利用する命令 B が存在するとき、命令 A と命令 B の間にはデータ依存（data dependence）があると呼びます。データ依存に対処するためには、命令 A の結果がレジスタに書き込まれるのを待つ必要があります。データ依存があることにより発生するパイプラ

インハザードのことをデータハザード (data hazard) と呼びます。



▲図 7.7: データ依存関係のあるプログラム

### 7.3.3 データ依存に対処する

レジスタの値を読み出すのは EX ステージです。データ依存に対処するために、データ依存関係があるときに EX ステージをストールさせます。

まず、MEM と EX か、WB と EX ステージにある命令の間にデータ依存があることを検知します (リスト 7.25)。例えば MEM ステージとデータ依存の関係にあるとき、MEM ステージの命令はライトバックする命令で、rd が EX ステージの rs1、または rs2 と一致しています。

#### ▼リスト 7.25: データ依存の検知 (core.veryl)

```

// データハザード
let exs_mem_data_hazard: logic = mems_valid && mems_ctrl.rwb_en && (mems_rd_addr == exs_rs1_
>addr || mems_rd_addr == exs_rs2_addr);
let exs_wb_data_hazard : logic = wbs_valid && wbs_ctrl.rwb_en && (wbs_rd_addr == exs_rs1_add_
>r || wbs_rd_addr == exs_rs2_addr);
let exs_data_hazard : logic = exs_mem_data_hazard || exs_wb_data_hazard;

```

次に、データ依存があるときに、データハザードを発生させます (リスト 7.26)。データハザードを起こすためには、EX ステージの FIFO の `rready` と MEM ステージの `wvalid` に、データハザードが発生していないという条件を加えます。

#### ▼リスト 7.26: データ依存があるときにデータハザードを起こす (core.veryl)

```

always_comb {
    // EX -> MEM
    exq_rready      = memq_wready && !exs_data_hazard;
    memq_wvalid     = exq_rvalid && !exs_data_hazard;
}

```

最後に、データハザードが発生しているかどうかをデバッグ表示します (リスト 7.27)。

#### ▼リスト 7.27: データハザードが発生しているかをデバッグ表示する (core.veryl)

```

$display("EX -----");
if exs_valid {
    $display("%h : %h", exq_rdata.addr, exq_rdata.bits);
    $display(" op1      : %h", exs_op1);
}

```

```
$display(" op2      : %h", exs_op2);
$display(" alu      : %h", exs_alu_result);
$display(" dhazard : %b", exs_data_hazard);
```

### 7.3.4 パイプライン処理をテストする

`test/sample_datahazard.hex` が正しく動くことを確認します。

▼リスト 7.28: `sample_datahazard.hex` が正しく動くことを確認する

```
$ make build
$ make sim
$ ./obj_dir/sim test/sample_datahazard.hex 7
...
#                      5
...
ID -----
 0000000000000004 : 00108113
  type : 000010
  imm  : 0000000000000001
EX -----
 0000000000000000 : 00100093
  op1   : 0000000000000000
  op2   : 0000000000000001
  alu   : 0000000000000001
  dhazard : 0
...
#                      6
...
EX -----
 0000000000000004 : 00108113
  op1   : 0000000000000000
  op2   : 0000000000000001
  alu   : 0000000000000001
  dhazard : 1 ←データハザードが発生している
MEM -----
 0000000000000000 : 00100093
  mem stall : 0
  mem rdata : 0000000000000000
WB -----
#
#                      7
...
EX -----
 0000000000000004 : 00108113
  op1   : 0000000000000000
  op2   : 0000000000000001
  alu   : 0000000000000001
  dhazard : 1
MEM -----
WB -----
 0000000000000000 : 00100093
```

```
reg[ 1] <= 0000000000000001 ← 1が書き込まれる

#
#          8
...
EX ----
0000000000000004 : 00108113
op1      : 0000000000000001 ← x1=1が読み込まれた
op2      : 0000000000000001
alu       : 0000000000000002 ← 正しい計算が行われている
dhsazard : 0 ← データハザードが解消された
MEM ----
WB ----
```

アドレス 4 の命令が、6 クロック目と 7 クロック目に EX ステージでデータハザードが発生し、アドレス 0 の命令が実行終了するのを待っているのを確認できます。

RV64I の riscv-tests も実行します。

▼ リスト 7.29: riscv-tests を実行する

```
$ make build
$ make sim VERILATOR_FLAGS="-DTEST_MODE"
$ python3 test/test.py -r obj_dir/sim test/share rv64ui-p-
...
FAIL : ~/core/test/share/riscv-tests/isa/rv64ui-p-ma_data.bin.hex
...
Test Result : 51 / 52
```

正しくパイプライン処理が動いていることを確認できました。

## **第 II 部**

# **RV64IMAC の実装**

# 第 8 章

## M 拡張の実装

### 8.1 概要

「第 I 部 RV32I / RV64I の実装」では RV64I の CPU を実装しました。「第 II 部 RV64IMAC の実装」では、次のような機能を実装します。

- 乗算、除算、剰余演算命令 (M 拡張)
- 不可分操作命令 (A 拡張)
- 圧縮命令 (C 拡張)
- 例外
- Memory-mapped I/O

本章では積、商、剰余を求める命令を実装します。RISC-V の乗算、除算、剰余演算を行う命令は M 拡張に定義されており、M 拡張を実装した RV64I の ISA のことを **RV64IM** と表記します。

M 拡張には、XLEN が **32** のときは表 8.1 の命令が定義されています。XLEN が **64** のときは表 8.2 の命令が定義されています。

▼表 8.1: M 拡張の命令 (XLEN=32)

命令	動作
MUL	rs1(符号付き) $\times$ rs2(符号付き) の結果 (64 ビット) の下位 32 ビットを求める
MULH	rs1(符号付き) $\times$ rs2(符号付き) の結果 (64 ビット) の上位 32 ビットを求める
MULHU	rs1(符号無し) $\times$ rs2(符号無し) の結果 (64 ビット) の上位 32 ビットを求める
MULHSU	rs1(符号付き) $\times$ rs2(符号無し) の結果 (64 ビット) の上位 32 ビットを求める
DIV	rs1(符号付き) $/$ rs2(符号付き) を求める
DIVU	rs1(符号無し) $/$ rs2(符号無し) を求める
REM	rs1(符号付き) $\% \text{ rs2(符号付き)}$ を求める
REMU	rs1(符号無し) $\% \text{ rs2(符号無し)}$ を求める

Veryl には積、商、剰余を求める演算子 **\***、**/**、**%** が定義されており、これを利用することで

▼表8.2: M拡張の命令 (XLEN=64)

命令	動作
MUL	$rs1(\text{符号付き}) \times rs2(\text{符号付き})$ の結果 (128ビット) の下位 64 ビットを求める
MULW	$rs1[31:0](\text{符号付き}) \times rs2[31:0](\text{符号付き})$ の結果 (64ビット) の下位 32 ビットを求める 結果は符号拡張する
MULH	$rs1(\text{符号付き}) \times rs2(\text{符号付き})$ の結果 (128ビット) の上位 64 ビットを求める
MULHU	$rs1(\text{符号無し}) \times rs2(\text{符号無し})$ の結果 (128ビット) の上位 64 ビットを求める
MULHSU	$rs1(\text{符号付き}) \times rs2(\text{符号無し})$ の結果 (128ビット) の上位 64 ビットを求める
DIV	$rs1(\text{符号付き}) / rs2(\text{符号付き})$ を求める
DIVW	$rs1[31:0](\text{符号付き}) / rs2[31:0](\text{符号付き})$ を求める 結果は符号拡張する
DIVU	$rs1(\text{符号無し}) / rs2(\text{符号無し})$ を求める
DIVWU	$rs1[31:0](\text{符号無し}) / rs2[31:0](\text{符号無し})$ を求める 結果は符号拡張する
REM	$rs1(\text{符号付き}) \% rs2(\text{符号付き})$ を求める
REMW	$rs1[31:0](\text{符号付き}) \% rs2[31:0](\text{符号付き})$ を求める 結果は符号拡張する
REMU	$rs1(\text{符号無し}) \% rs2(\text{符号無し})$ を求める
REMUW	$rs1[31:0](\text{符号無し}) \% rs2[31:0](\text{符号無し})$ を求める 結果は符号拡張する

簡単に計算を実装できます (リスト8.1)。

▼リスト8.1: 演算子による実装例

```

1 assign mul = op1 * op2;
2 assign div = op1 / op2;
3 assign rem = op1 % op2;
```

例えば乗算回路を FPGA 上に実装する場合、通常は合成系によって FPGA に搭載されている乗算器が自動的に利用されます<sup>\*1</sup>。これにより、低遅延、低リソースコストで効率的な乗算回路を自動的に実現できます。しかし、32ビットや64ビットの乗算を実装する際、FPGA 上の乗算器の数が不足すると、LUT を用いた大規模な乗算回路が構築されることがあります。このような大規模な回路は FPGA のリソースの使用量や遅延に大きな影響を与えるため好ましくありません。除算や剰余演算でも同じ問題<sup>\*2</sup>が生じることがあります。

\*、/、% 演算子がどのような回路に合成されるかは、合成系が全体の実装を考慮して自動的に決定するため、その挙動をコントロールするのは難しいです。そこで本章では、\*、/、% 演算子を使用せず、足し算やシフト演算などの基本的な論理だけを用いて同等の演算を実装します。

基本編では積、商、剰余を効率よく<sup>\*3</sup>求める実装は検討せず、できるだけ単純な方法で実装し

<sup>\*1</sup> 手動で何をどのように利用するかを選択することもできます。既に用意された回路 (IP) を使うこともできますが、本書は自作することを主軸としているため利用しません。

<sup>\*2</sup> そもそも除算器が搭載されていない場合があります。

<sup>\*3</sup> 「効率」は、計算に要する時間やスループット、回路面積のことです。効率的に計算する方法については応用編で検討します。

ます。

## 8.2 命令のデコード

まず、M拡張の命令をデコードします。M拡張の命令はすべてR形式であり、レジスタの値同士の演算を行います。funct7は7'b0000001です。MUL、MULH、MULHSU、MULHU、DIV、DIVU、REM、REMU命令のopcodeは7'b0110011(OP)で、MULW、DIVW、DIVUW、REMW、REMUW命令のopcodeは7'b0111011(OP-32)です。

それぞれの命令はfunct3で区別します(表8.3)。乗算命令のfunct3はMSBが0、除算と剰余演算命令は1になっています。

▼表8.3: M拡張の命令の区別

命令	funct3
MUL、MULW	000
MULH	001
MULHU	010
MULHSU	011
DIV、DIVW	100
DIVU、DIVWU	101
REM、REMW	110
REMU、REMUW	111

InstCtrl構造体に、M拡張の命令であることを示すis\_muldivフラグを追加します(リスト8.2)。

▼リスト8.2: is\_muldivフラグを追加する(corectrl.veryl)

```

1 // 制御に使うフラグ用の構造体
2 struct InstCtrl {
3     itype      : InstType   , // 命令の形式
4     rwb_en    : logic      , // レジスタに書き込むかどうか
5     is_lui    : logic      , // LUI命令である
6     is_aluop  : logic      , // ALUを利用する命令である
7     is_muldiv: logic      , // M拡張の命令である
8     is_op32   : logic      , // OP-32またはOP-IMM-32である
9     is_jump   : logic      , // ジャンプ命令である
10    is_load   : logic      , // ロード命令である
11    is_csr    : logic      , // CSR命令である
12    funct3   : logic <3>, // 命令のfunct3フィールド
13    funct7   : logic <7>, // 命令のfunct7フィールド
14 }
```

inst\_decoderモジュールのInstCtrlを生成している部分を変更します。opcodeがOPかOP-32の場合はfunct7の値によってis\_muldivを設定します(リスト8.3)。その他

の opcode の `is_muldiv` は `F` に設定してください。

▼リスト 8.3: `is_muldiv` を設定する (`inst_decoder.veryl`) (一部)

```

1      OP_OP: {
2          InstType::R, T, F, T, f7 == 7'b0000001, F, F, F, F
3      },
4      OP_OP_IMM: {
5          InstType::I, T, F, T, F, F, F, F, F
6      },
7      OP_OP_32: {
8          InstType::R, T, F, T, f7 == 7'b0000001, T, F, F, F
9      },

```

## 8.3 muldivunit モジュールの実装

### 8.3.1 muldivunit モジュールを作成する

M拡張の計算を処理するモジュールを作成し、M拡張の命令が ALU の結果ではなくモジュールの結果を利用するように変更します。

`src/muldivunit.veryl` を作成し、次のように記述します（リスト 8.4）。

▼リスト 8.4: `muldivunit.veryl`

```

1 import eei::*;

2 module muldivunit (
3     clk : input clock ,
4     rst : input reset ,
5     ready : output logic ,
6     valid : input logic ,
7     funct3: input logic<3>,
8     op1 : input UIntX ,
9     op2 : input UIntX ,
10    rvalid: output logic ,
11    result: output UIntX ,
12 ) {
13
14     enum State {
15         Idle,
16         WaitValid,
17         Finish,
18     }
19
20     var state: State;
21
22     // saved_data
23     var funct3_saved: logic<3>;
24
25 }
```

```

26    always_comb {
27        ready  = state == State::Idle;
28        rvalid = state == State::Finish;
29    }
30
31    always_ff {
32        if_reset {
33            state      = State::Idle;
34            result     = 0;
35            funct3_saved = 0;
36        } else {
37            case state {
38                State::Idle: if ready && valid {
39                    state      = State::WaitValid;
40                    funct3_saved = funct3;
41                }
42                State::WaitValid: state = State::Finish;
43                State::Finish   : state = State::Idle;
44                default       : {}
45            }
46        }
47    }
48 }

```

muldivunit モジュールは `ready` が 1 のときに計算のリクエストを受け付けます。`valid` が 1 なら計算を開始し、計算が終了したら `rvalid` を 1、計算結果を `result` に設定します。

まだ計算処理を実装しておらず、`result` は常に 0 を返します。次の計算を開始するまで `result` の値を維持します。

### 8.3.2 EXステージを変更する

M拡張の命令が EXステージにあるとき、ALUの結果の代わりに muldivunit モジュールの結果を利用するように変更します。

まず、muldivunit モジュールをインスタンス化します（リスト 8.5）。

#### ▼リスト 8.5: muldivunit モジュールをインスタンス化する (core.veryl)

```

1  let exs_muldiv_valid : logic = exs_valid && exs_ctrl.is_muldiv && !exs_data_hazard && !exs_m
2  >uldiv_is_requested;
3  var exs_muldiv_ready : logic;
4  var exs_muldiv_rvalid: logic;
5  var exs_muldiv_result: UIntX;
6
7  inst mdu: muldivunit (
8      clk           ,
9      rst           ,
10     valid : exs_muldiv_valid ,
11     ready : exs_muldiv_ready ,
12     funct3: exs_ctrl.funct3 ,
13

```

```

12     op1    : exs_op1          ,
13     op2    : exs_op2          ,
14     rvalid: exs_muldiv_rvalid,
15     result: exs_muldiv_result,
16 );

```

muldivunit モジュールで計算を開始するのは、EX ステージに命令が存在し (`exs_valid`)、命令が M 拡張の命令であり (`exs_ctrl.is_muldiv`)、データハザードが発生しておらず (`!exs_data_hazard`)、既に計算を要求していない (`!exs_muldiv_is_requested`) 場合です。

`exs_muldiv_is_requested` 変数を定義し、ステージの遷移条件と muldivunit に計算を要求したかの状態によって値を更新します (リスト 8.6)。

#### ▼ リスト 8.6: `exs_muldiv_is_requested` 変数 (core.veryl)

```

1  var exs_muldiv_is_requested: logic;
2
3  always_ff {
4      if_reset {
5          exs_muldiv_is_requested = 0;
6      } else {
7          // 次のステージに遷移
8          if exq_rvalid && exq_rready {
9              exs_muldiv_is_requested = 0;
10         } else {
11             // muldivunitにリクエストしたか判定する
12             if exs_muldiv_valid && exs_muldiv_ready {
13                 exs_muldiv_is_requested = 1;
14             }
15         }
16     }
17 }

```

muldivunit モジュールは ALU のように 1 クロックの間に入力から出力を生成しないため、計算中は EX ステージをストールさせる必要があります。そのために `exs_muldiv_stall` 変数を定義して、ストールの条件に追加します (リスト 8.7、リスト 8.8)。また、M 拡張の命令の場合は MEM ステージに渡す `alu_result` の値を muldivunit モジュールの結果に設定します (リスト 8.8)。

#### ▼ リスト 8.7: EX ステージのストール条件の変更 (core.veryl)

```

1  var exs_muldiv_rvalidated: logic;
2  let exs_muldiv_stall    : logic = exs_ctrl.is_muldiv && !exs_muldiv_rvalid && !exs_muldiv_rva>
3  >lided;
4
5  always_ff {
6      if_reset {
7          exs_muldiv_rvalidated = 0;
8      } else {
9          // 次のステージに遷移
10         if exq_rvalid && exq_rready {
11             exs_muldiv_rvalidated = 0;
12         }
13     }
14 }

```

```

11     } else {
12         // muldivunitの処理が完了していたら1にする
13         exs_muldiv_rvalied |= exs_muldiv_rvalid;
14     }
15 }
16 }
```

▼リスト 8.8: EXステージのストール条件の変更とM拡張の命令の結果の設定 (core.vveril)

```

1 let exs_stall: logic = exs_data_hazard || exs_muldiv_stall;
2
3 always_comb {
4     // EX -> MEM
5     exq_rready      = memq_wready && !exs_stall;
6     memq_wvalid      = exq_rvalid && !exs_stall;
7     memq_wdata.addr  = exq_rdata.addr;
8     memq_wdata.bits   = exq_rdata.bits;
9     memq_wdata.ctrl    = exq_rdata.ctrl;
10    memq_wdata.imm     = exq_rdata.imm;
11    memq_wdata.rs1_addr = exs_rs1_addr;
12    memq_wdata.rs1_data = exs_rs1_data;
13    memq_wdata.rs2_data = exs_rs2_data;
14    memq_wdata.alu_result = if exs_ctrl.is_muldiv ? exs_muldiv_result : exs_alu_result;
15    memq_wdata.br_taken   = exs_ctrl.is_jump || inst_is_br(exs_ctrl) && exs_brunit_take;
16    memq_wdata.jump_addr  = if inst_is_br(exs_ctrl) ? exs_pc + exs_imm : exs_alu_result & ~
>1;
17 }
```

muldivunit モジュールは計算が完了したクロックでしか `rvalid` を 1 にしないため、既に計算が完了したことを見出す `exs_muldiv_rvalied` 変数で完了状態を管理します。これにより、M拡張の命令によってストールする条件は、命令が M拡張の命令であり (`exs_ctrl.is_muldiv`)、現在のクロックで計算が完了しておらず (`!exs_muldiv_rvalid`)、以前のクロックでも計算が完了していない (`!exs_muldiv_rvalied`) 場合になります。

## 8.4 符号無しの乗算器の実装

### 8.4.1 mulunit モジュールを実装する

`WIDTH` ビットの符号無しの値同士の積を計算する乗算器を実装します。

`src/muldivunit.vveril` の中に mulunit モジュールを作成します (リスト 8.9)。

▼リスト 8.9: muldivunit.vveril

```

1 module mulunit #(
2     param WIDTH: u32 = 0,
3 ) (
4     clk    : input  clock          ,
5     rst    : input  reset          ,
```

```
6     valid : input  logic      ,
7     op1   : input  logic<WIDTH>  ,
8     op2   : input  logic<WIDTH>  ,
9     rvalid: output logic      ,
10    result: output logic<WIDTH * 2>,
11  ) {
12     enum State {
13       Idle,
14       AddLoop,
15       Finish,
16     }
17
18     var state: State;
19
20     var op1zext: logic<WIDTH * 2>;
21     var op2zext: logic<WIDTH * 2>;
22
23     always_comb {
24       rvalid = state == State::Finish;
25     }
26
27     var add_count: logic<32>;
28
29     always_ff {
30       if_reset {
31         state      = State::Idle;
32         result     = 0;
33         add_count = 0;
34         op1zext   = 0;
35         op2zext   = 0;
36       } else {
37         case state {
38           State::Idle: if valid {
39             state      = State::AddLoop;
40             result     = 0;
41             add_count = 0;
42             op1zext   = {1'b0 repeat WIDTH, op1};
43             op2zext   = {1'b0 repeat WIDTH, op2};
44           }
45           State::AddLoop: if add_count == WIDTH {
46             state = State::Finish;
47           } else {
48             if op2zext[add_count] {
49               result += op1zext;
50             }
51             op1zext  <<= 1;
52             add_count += 1;
53           }
54           State::Finish: state = State::Idle;
55           default      : {}
56         }
57       }
58     }
```

59 }

mulunit モジュールは `op1 * op2` を計算するモジュールです。`valid` が 1 になったら計算を開始し、計算が完了したら `rvalid` を 1、`result` を `WIDTH * 2` ビットの計算結果に設定します。

積は `WIDTH` 回の足し算を `WIDTH` クロックかけて行って求めていきます(図 8.1)。計算を開始すると入力をゼロで `WIDTH * 2` ビットに拡張し、`result` を 0 でリセットします。

`State::AddLoop` では、次の操作を `WIDTH` 回行います。`i` 回目では次の操作を行います。

1. `op2[i-1]` が 1 なら `result` に `op1` を足す
2. `op1` を 1 ビット左シフトする
3. カウンタをインクリメントする

$$\begin{array}{r}
 1010 \text{ op1 (4bit)} \\
 \times 0101 \text{ op2 (4bit)} \\
 \hline
 1010 = \text{op2} \\
 0000 = (\text{op2} \ll 1) * 0 \\
 1010 = \text{op2} \ll 2 \\
 + 0000 = (\text{op2} \ll 3) * 0 \\
 \hline
 00110010 \text{ result (8bit)}
 \end{array}$$

▲図 8.1: 符号無し 4 ビットの乗算

### 8.4.2 mulunit モジュールをインスタンス化する

mulunit モジュールを muldivunit モジュールでインスタンス化します(リスト 8.10)。まだ結果は利用しません。

▼リスト 8.10: mulunit モジュールをインスタンス化する (muldivunit.veryl)

```

1 // multiply unit
2 const MUL_OP_WIDTH : u32 = XLEN;
3 const MUL_RES_WIDTH: u32 = MUL_OP_WIDTH * 2;
4
5 let is_mul    : logic          = if state == State::Idle ? !funct3[2] : !funct3_saved > [2];
6 var mu_rvalid: logic          ;

```

```

7  var mu_result: logic<MUL_RES_WIDTH>;
8
9  inst mu: mulunit #(
10    WIDTH: MUL_OP_WIDTH,
11  ) (
12    clk           ,
13    rst           ,
14    valid : ready && valid && is_mul,
15    op1   : op1      ,
16    op2   : op2      ,
17    rvalid: mu_rvalid ,
18    result: mu_result ,
19 );

```

## 8.5 MULHU命令の実装

MULHU命令は、2つの符号無しのXLENビットの値の乗算を実行し、デスティネーションレジスタに結果(XLEN \* 2ビット)の上位XLENビットを書き込む命令です。funct3の下位2ビットによってmulunitモジュールの結果を選択するようにします(リスト8.11)。

▼リスト8.11: MULHUモジュールの結果を取得する(muldivunit.veryl)

```

1  State::WaitValid: if is_mul && mu_rvalid {
2    state = State::Finish;
3    result = case funct3_saved[1:0] {
4      2'b11 : mu_result[XLEN+:XLEN], // MULHU
5      default: 0,
6    };
7  }

```

riscv-testsのrv64um-p-mulhuを実行し、成功することを確認してください。

## 8.6 MUL、MULH命令の実装

### 8.6.1 符号付き乗算を符号無し乗算器で実現する

MUL、MULH命令は、2つの符号付きのXLENビットの値の乗算を実行し、デスティネーションレジスタにそれぞれ結果の下位XLENビット、上位XLENビットを書き込む命令です。

本章ではmulunitモジュールを使って、次のように符号付き乗算を実現します。

1. 符号付きのXLENビットの値を符号無しの値(絶対値)に変換する
2. 符号無しで積を計算する
3. 計算結果の符号を修正する

絶対値で計算することで符号ビットを考慮する必要がなくなり、既に実装してある符号無しの乗算器を変更せずに符号付きの乗算を実現できます。

## 8.6.2 符号付き乗算を実装する

**WIDTH** ビットの符号付きの値を **WIDTH** ビットの符号無しの絶対値に変換する abs 関数を作成します（リスト 8.12）。abs 関数は、値の MSB が 1 ならビットを反転して 1 を足すことで符号を反転しています。最小値  $-2^{**(\text{WIDTH} - 1)}$  の絶対値も求められることを確認してください。

### ▼リスト 8.12: abs 関数を実装する (muldivunit.veryl)

```

1  function abs::<WIDTH: u32> (
2      value: input logic<WIDTH>,
3  ) -> logic<WIDTH> {
4      return if value[msb] ? ~value + 1 : value;
5  }
```

abs 関数を利用して、MUL、MULH 命令のときに mulunit に渡す値を絶対値に設定します（リスト 8.13、リスト 8.14）。

### ▼リスト 8.13: op1 と op2 を生成する (muldivunit.veryl)

```

1  let mu_op1: logic<MUL_OP_WIDTH> = case funct3[1:0] {
2      2'b00, 2'b01: abs::<XLEN>(op1), // MUL, MULH
3      2'b11       : op1, // MULHU
4      default     : 0,
5  };
6  let mu_op2: logic<MUL_OP_WIDTH> = case funct3[1:0] {
7      2'b00, 2'b01: abs::<XLEN>(op2), // MUL, MULH
8      2'b11       : op2, // MULHU
9      default     : 0,
10 };
```

### ▼リスト 8.14: mulunit に渡す値を変更する (muldivunit.veryl)

```

1  inst mu: mulunit #(
2      WIDTH: MUL_OP_WIDTH,
3  ) (
4      clk           ,
5      rst           ,
6      valid : ready && valid && is_mul,
7      op1   : mu_op1        ,
8      op2   : mu_op2        ,
9      rvalid: mu_rvalid    ,
10     result: mu_result    ,
11 );
```

計算結果の符号は **op1** と **op2** の符号が異なる場合に負になります。後で符号の情報を利用するために、muldivunit モジュールが要求を受け入れる時に符号を保存します（リスト 8.15、リスト 8.16、リスト 8.17）。

## ▼リスト 8.15: 符号を保存する変数を作成する (muldivunit.veryl)

```

1 // saved_data
2 var funct3_saved : logic<3>;
3 var op1sign_saved: logic   ;
4 var op2sign_saved: logic   ;

```

## ▼リスト 8.16: 変数のリセット (muldivunit.veryl)

```

1 always_ff {
2     if_reset {
3         state      = State::Idle;
4         result     = 0;
5         funct3_saved = 0;
6         op1sign_saved = 0;
7         op2sign_saved = 0;
8     } else {

```

## ▼リスト 8.17: 符号を変数に保存する (muldivunit.veryl)

```

1 case state {
2     State::Idle: if ready && valid {
3         state      = State::WaitValid;
4         funct3_saved = funct3;
5         op1sign_saved = op1[msb];
6         op2sign_saved = op2[msb];
7     }

```

保存した符号を利用して計算結果の符号を復元します（リスト 8.18）。

## ▼リスト 8.18: 計算結果の符号を復元する (muldivunit.veryl)

```

1 State::WaitValid: if is_mul && mu_rvalid {
2     let res_signed: logic<mul_res_width> = if op1sign_saved != op2sign_saved ? ~mu_result +>
3     1 : mu_result;
4     state      = State::Finish;
5     result     = case funct3_saved[1:0] {
6         2'b00 : res_signed[XLEN - 1:0], // MUL
7         2'b01 : res_signed[XLEN:XLEN], // MULH
8         2'b11 : mu_result[XLEN:XLEN], // MULHU
9         default: 0,
10    };
}

```

riscv-tests の `rv64um-p-mul` と `rv64um-p-mulh` を実行し、成功することを確認してください。

### 8.6.3 MULHSU 命令の実装

MULHSU 命令は、符号付きの XLEN ビットの rs1 と符号無しの XLEN ビットの rs2 の乗算を実行し、デスティネーションレジスタに結果の上位 XLEN ビットを書き込む命令です。計算結果は符号付きの値になります。

MULHSU 命令も、MUL、MULH 命令と同様に符号無しの乗算器で実現します。

`op1` を絶対値に変換し、`op2` はそのままに設定します（リスト 8.19）。

▼ リスト 8.19: MULHSU 命令用に `op1`、`op2` を設定する (muldivunit.veryl)

```

1 let mu_op1: logic<MUL_OP_WIDTH> = case funct3[1:0] {
2     2'b00, 2'b01, 2'b10: abs::<XLEN>(op1), // MUL, MULH, MULHSU
3     2'b11             : op1, // MULHU
4     default           : 0,
5 };
6 let mu_op2: logic<MUL_OP_WIDTH> = case funct3[1:0] {
7     2'b00, 2'b01: abs::<XLEN>(op2), // MUL, MULH
8     2'b11, 2'b10: op2, // MULHU, MULHSU
9     default      : 0,
10};

```

計算結果は `op1` の符号にします（リスト 8.20）。

▼ リスト 8.20: 計算結果の符号を復元する (muldivunit.veryl)

```

1 State::WaitValid: if is_mul && mu_rvalid {
2     let res_signed: logic<MUL_RES_WIDTH> = if op1sign_saved != op2sign_saved ? ~mu_result +>
3         1 : mu_result;
4     let res_mulhsu: logic<MUL_RES_WIDTH> = if op1sign_saved == 1 ? ~mu_result + 1 : mu_resu>
5     >lt;
6         state      = State::Finish;
7         result     = case funct3_saved[1:0] {
8             2'b00   : res_signed[XLEN - 1:0], // MUL
9             2'b01   : res_signed[XLEN+:XLEN], // MULH
10            2'b10   : res_mulhsu[XLEN+:XLEN], // MULHSU
11            2'b11   : mu_result[XLEN+:XLEN], // MULHU
12            default: 0,
13        };
14    }
15}

```

riscv-tests の `rv64um-p-mulhsu` を実行し、成功することを確認してください。

## 8.6.4 MULW 命令の実装

MULW 命令は、2 つの符号付きの 32 ビットの値の乗算を実行し、デスティネーションレジスターに結果の下位 32 ビットを符号拡張した値を書き込む命令です。

32 ビット演算の命令であることを判定するために、muldivunit モジュールに `is_op32` ポートを作成します（リスト 8.21、リスト 8.22）。

▼ リスト 8.21: `is_op32` ポートを追加する (muldivunit.veryl)

```

1 module muldivunit (
2     clk      : input  clock   ,
3     rst      : input  reset   ,
4     ready    : output logic   ,
5     valid   : input  logic   ,
6     funct3 : input  logic<3>,
7     is_op32: input  logic   ,

```

```

8   op1    : input  UIntX   ,
9   op2    : input  UIntX   ,
10  rvalid : output logic   ,
11  result : output UIntX   ,
12 ) {
```

## ▼リスト 8.22: is\_op32 ポートに値を割り当てる (core.veryl)

```

1 inst mdu: muldivunit (
2     clk           ,
3     rst           ,
4     valid        : exs_muldiv_valid ,
5     ready        : exs_muldiv_ready ,
6     funct3      : exs_ctrl.funct3 ,
7     is_op32: exs_ctrl.is_op32 ,
8     op1          : exs_op1 ,
9     op2          : exs_op2 ,
10    rvalid       : exs_muldiv_rvalid,
11    result       : exs_muldiv_result,
12 );
```

muldivunit モジュールが要求を受け入れる時に `is_op32` を保存します（リスト 8.23、リスト 8.24、リスト 8.25）。

## ▼リスト 8.23: is\_op32 を保存する変数を作成する (muldivunit.veryl)

```

1 // saved_data
2 var funct3_saved : logic<3>;
3 var is_op32_saved: logic  ;
4 var op1sign_saved: logic  ;
5 var op2sign_saved: logic  ;
```

## ▼リスト 8.24: 変数のリセット (muldivunit.veryl)

```

1 always_ff {
2     if_reset {
3         state      = State::Idle;
4         result     = 0;
5         funct3_saved = 0;
6         is_op32_saved = 0;
7         op1sign_saved = 0;
8         op2sign_saved = 0;
9     } else {
```

## ▼リスト 8.25: is\_op32 を変数に保存する (muldivunit.veryl)

```

1 State::Idle: if ready && valid {
2     state      = State::WaitValid;
3     funct3_saved = funct3;
4     is_op32_saved = is_op32;
5     op1sign_saved = op1[msb];
6     op2sign_saved = op2[msb];
7 }
```

mulunit モジュールの `op1` と `op2` に、64 ビットの値の下位 32 ビットを符号拡張した値を割り当てます。符号拡張を行う `sext` 関数を作成し、`mu_op1`、`mu_op2` の割り当てに利用します（リスト 8.26、リスト 8.27）。

#### ▼ リスト 8.26: 符号拡張する関数を作成する (muldivunit.veryl)

```

1  function sext::<WIDTH_IN: u32, WIDTH_OUT: u32> (
2      value: input logic<WIDTH_IN>,
3      ) -> logic<WIDTH_OUT> {
4          return {value[msb] repeat WIDTH_OUT - WIDTH_IN, value};
5      }

```

#### ▼ リスト 8.27: MULW 命令用に op1、op2 を設定する (muldivunit.veryl)

```

1  let mu_op1: logic<MUL_OP_WIDTH> = case funct3[1:0] {
2      2'b00, 2'b01, 2'b10: abs::<XLEN>(if is_op32 ? sext::<32, XLEN>(op1[31:0]) : op1), // MUL
>L, MULH, MULHSU, MULW
3      2'b11           : op1, // MULHU
4      default        : 0,
5  };
6  let mu_op2: logic<MUL_OP_WIDTH> = case funct3[1:0] {
7      2'b00, 2'b01: abs::<XLEN>(if is_op32 ? sext::<32, XLEN>(op2[31:0]) : op2), // MUL, MULH
>, MULW
8      2'b11, 2'b10: op2, // MULHU, MULHSU
9      default      : 0,
10 };

```

最後に、計算結果を符号拡張した値に設定します（リスト 8.28）。

#### ▼ リスト 8.28: 計算結果を符号拡張する (muldivunit.veryl)

```

1  State::WaitValid: if is_mul && mu_rvalid {
2      let res_signed: logic<MUL_RES_WIDTH> = if op1sign_saved != op2sign_saved ? ~mu_result +>
> 1 : mu_result;
3      let res_mulhsu: logic<MUL_RES_WIDTH> = if op1sign_saved == 1 ? ~mu_result + 1 : mu_resu>lt;
4      state      = State::Finish;
5      result     = case funct3_saved[1:0] {
6          2'b00 : if is_op32_saved ? sext::<32, 64>(res_signed[31:0]) : res_signed[XLEN - 1:>0], // MUL, MULW
7          2'b01 : res_signed[XLEN:XLEN], // MULH

```

riscv-tests の `rv64um-p-mulw` を実行し、成功することを確認してください。

## 8.7 符号無し除算の実装

### 8.7.1 divunit モジュールを実装する

`WIDTH` ビットの除算を計算する除算器を実装します。

`src/muldivunit.veryl` の中に `divunit` モジュールを作成します（リスト 8.29）。

▼ リスト 8.29: multdivunit.veryl

```

1 module divunit #(
2     param WIDTH: u32 = 0,
3 ) (
4     clk      : input  clock      ,
5     rst      : input  reset      ,
6     valid    : input  logic      ,
7     dividend : input  logic<WIDTH>,
8     divisor  : input  logic<WIDTH>,
9     rvalid   : output logic      ,
10    quotient : output logic<WIDTH>,
11    remainder: output logic<WIDTH>,
12 ) {
13     enum State {
14         Idle,
15         ZeroCheck,
16         SubLoop,
17         Finish,
18     }
19
20     var state: State;
21
22     var dividend_saved: logic<WIDTH * 2>;
23     var divisor_saved : logic<WIDTH * 2>;
24
25     always_comb {
26         rvalid    = state == State::Finish;
27         remainder = dividend_saved[WIDTH - 1:0];
28     }
29
30     var sub_count: u32;
31
32     always_ff {
33         if_reset {
34             state        = State::Idle;
35             quotient    = 0;
36             sub_count   = 0;
37             dividend_saved = 0;
38             divisor_saved = 0;
39         } else {
40             case state {
41                 State::Idle: if valid {
42                     state        = State::ZeroCheck;
43                     dividend_saved = {1'b0 repeat WIDTH, dividend};
44                     divisor_saved = {1'b0, divisor, 1'b0 repeat WIDTH - 1};
45                     quotient    = 0;
46                     sub_count   = 0;
47                 }
48                 State::ZeroCheck: if divisor_saved == 0 {
49                     state        = State::Finish;
50                     quotient    = '1;
51                 } else {
52                     state = State::SubLoop;
53                 }
54             }
55         }
56     }
57 }
```

```

53 }
54 State::SubLoop: if sub_count == WIDTH {
55     state = State::Finish;
56 } else {
57     if dividend_saved >= divisor_saved {
58         dividend_saved -= divisor_saved;
59         quotient      = (quotient << 1) + 1;
60     } else {
61         quotient <= 1;
62     }
63     divisor_saved >= 1;
64     sub_count      += 1;
65 }
66 State::Finish: state = State::Idle;
67 default      : {}
68 }
69 }
70 }
71 }

```

divunit モジュールは被除数 (`dividend`) と除数 (`divisor`) の商 (`quotient`) と剰余 (`remainder`) を計算するモジュールです。`valid` が 1 になったら計算を開始し、計算が完了したら `rvalid` を 1 に設定します。

商と剰余は `WIDTH` 回の引き算を `WIDTH` クロックかけて行って求めています。計算を開始すると被除数を 0 で `WIDTH * 2` ビットに拡張し、除数を `WIDTH-1` ビット左シフトします。また、商を 0 でリセットします。

`State::SubLoop` では、次の操作を `WIDTH` 回行います。

1. 被除数が除数よりも大きいなら、被除数から除数を引き、商の LSB を 1 にする
2. 商を 1 ビット左シフトする
3. 除数を 1 ビット右シフトする
4. カウンタをインクリメントする

RISC-V では、除数が 0 だったり結果がオーバーフローするような L ビットの除算の結果は表 8.4 のようになると定められています。このうち divunit モジュールは符号無しの除算 (DIVU、REMU 命令) のゼロ除算だけを対処しています。

▼表 8.4: 除算の例外的な動作と結果

操作	ゼロ除算	オーバーフロー
符号付き除算	-1	$-2^{**L-1}$
符号付き剰余	被除数	0
符号無し除算	$2^{**L-1}$	発生しない
符号無し剰余	被除数	発生しない

## 8.7.2 divunitモジュールをインスタンス化する

divunitモジュールを muldivunitモジュールでインスタンス化します(リスト8.30)。まだ結果は利用しません。

▼リスト8.30: divunitモジュールをインスタンス化する(muldivunit.veryl)

```

1 // divider unit
2 const DIV_WIDTH: u32 = XLEN;
3
4 var du_rvalid    : logic          ;
5 var du_quotient : logic<DIV_WIDTH>;
6 var du_remainder: logic<DIV_WIDTH>;
7
8 inst du: divunit #(
9     WIDTH: DIV_WIDTH,
10 ) (
11     clk           ,
12     rst           ,
13     valid        : ready && valid && !is_mul,
14     dividend    : op1           ,
15     divisor     : op2           ,
16     rvalid       : du_rvalid   ,
17     quotient    : du_quotient ,
18     remainder   : du_remainder,
19 );

```

## 8.8 DIVU、REMU命令の実装

DIVU、REMU命令は、符号無しのXLENビットのrs1(被除数)と符号無しのXLENビットのrs2(除数)の商、剰余を計算し、デスティネーションレジスタにそれぞれ結果を書き込む命令です。

muldivunitモジュールで、divunitモジュールの処理が終わったら結果をresultレジスタに割り当てるようにします(リスト8.31)。

▼リスト8.31: divunitモジュールの結果をresultに割り当てる(muldivunit.veryl)

```

1 State::WaitValid: if is_mul && mu_rvalid {
2     ...
3 } else if !is_mul && du_rvalid {
4     result = case funct3_saved[1:0] {
5         2'b01  : du_quotient, // DIVU
6         2'b11  : du_remainder, // REMU
7         default: 0,
8     };
9     state = State::Finish;
10 }

```

riscv-testsのrv64um-p-divu、rv64um-p-remuを実行し、成功することを確認してください。

## 8.9 DIV、REM命令の実装

### 8.9.1 符号付き除算を符号無し除算器で実現する

DIV、REM命令は、それぞれ DIVU、REMU命令の動作を符号付きに変えた命令です。本章では、符号付き乗算と同じように値を絶対値に変換して計算することで符号付き除算を実現します。

RISC-Vの符号付き除算の結果は0の方向に丸められた整数になり、剩余演算の結果は被除数と同じ符号になります。符号付き剩余の絶対値は符号無し剩余の結果と一致するため、絶対値で計算してから符号を戻すことで、符号無し除算器だけで符号付きの剩余演算を実現できます。

### 8.9.2 符号付き除算を実装する

`abs`関数を利用して、DIV、REM命令のときに `divunit`モジュールに渡す値を絶対値に設定します（リスト8.32 リスト8.33）。

#### ▼リスト8.32: 除数と被除数を生成する (`muldivunit.vyrl`)

```

1  function generate_div_op (
2      funct3: input logic<3> ,
3      value : input logic<XLEN>,
4  ) -> logic<DIV_WIDTH> {
5      return case funct3[1:0] {
6          2'b00, 2'b10: abs::<DIV_WIDTH>(value), // DIV, REM
7          2'b01, 2'b11: value, // DIVU, REMU
8          default      : 0,
9      };
10 }
11
12 let du_dividend: logic<DIV_WIDTH> = generate_div_op(funct3, op1);
13 let du_divisor : logic<DIV_WIDTH> = generate_div_op(funct3, op2);

```

#### ▼リスト8.33: `divunit`に渡す値を変更する (`muldivunit.vyrl`)

```

1  inst du: divunit #(
2      WIDTH: DIV_WIDTH,
3  ) (
4      clk
5      ,
6      rst
7      ,
8      valid   : ready && valid && !is_mul && !du_signed_error,
9      dividend : du_dividend
10     ,
11     divisor : du_divisor
12     ,
13     rvalid  : du_rvalid
14     ,
15     quotient : du_quotient
16     ,
17     remainder: du_remainder
18 );

```

表8.4にあるように、符号付き演算は結果がオーバーフローする場合とゼロで割る場合の結果が定められています。その場合には、`divunit`モジュールで除算を実行せず、`muldivunit`で計算結果を直接生成するようにします（リスト8.34 リスト8.35）。符号付き演算かどうかを `funct3` の

LSBで確認し、例外的な処理ではない場合にのみ divunitモジュールで計算を開始するようにします。

▼リスト8.34: 符号付き除算がオーバーフローするか、ゼロ除算かどうかを判定する(muldivunit.veryl)

```

1  var du_signed_overflow: logic;
2  var du_signed_divzero : logic;
3  var du_signed_error   : logic;
4
5  always_comb {
6      du_signed_overflow = !funct3[0] && op1[msb] == 1 && op1[msb - 1:0] == 0 && &op2;
7      du_signed_divzero = !funct3[0] && op2 == 0;
8      du_signed_error   = du_signed_overflow || du_signed_divzero;
9  }

```

▼リスト8.35: 符号付き除算の例外的な結果を処理する(muldivunit.veryl)

```

1  State::Idle: if ready && valid {
2      funct3_saved  = funct3;
3      is_op32_saved = is_op32;
4      op1sign_saved = op1[msb];
5      op2sign_saved = op2[msb];
6      if is_mul {
7          state = State::WaitValid;
8      } else {
9          if du_signed_overflow {
10              state = State::Finish;
11              result = if funct3[1] ? 0 : {1'b1, 1'b0 repeat XLEN - 1}; // REM : DIV
12          } else if du_signed_divzero {
13              state = State::Finish;
14              result = if funct3[1] ? op1 : '1; // REM : DIV
15          } else {
16              state = State::WaitValid;
17          }
18      }
19  }

```

計算が終了したら、商と剰余の符号を復元します。商の符号は除数と被除数の符号が異なる場合に負になります。剰余の符号は被除数の符号にします(リスト8.36)。

▼リスト8.36: 計算結果の符号を復元する(muldivunit.veryl)

```

1  } else if !is_mul && du_rvalid {
2      let quo_signed: logic<DIV_WIDTH> = if op1sign_saved != op2sign_saved ? ~du_quotient + 1 :
3          du_quotient;
4      let rem_signed: logic<DIV_WIDTH> = if op1sign_saved == 1 ? ~du_remainder + 1 : du_remainder;
5      result     = case funct3_saved[1:0] {
6          2'b00 : quo_signed[XLEN - 1:0], // DIV
7          2'b01 : du_quotient[XLEN - 1:0], // DIVU
8          2'b10 : rem_signed[XLEN - 1:0], // REM
9          2'b11 : du_remainder[XLEN - 1:0], // REMU
10         default: 0,
11     };

```

```

11     state = State::Finish;
12 }
```

riscv-tests の `rv64um-p-div`、`rv64um-p-rem` を実行し、成功することを確認してください。

## 8.10 DIVW、DIVUW、REMW、REMUW 命令の実装

DIVW、DIVUW、REMW、REMUW 命令は、それぞれ DIV、DIVU、REM、REMU 命令の動作を 32 ビット同士の演算に変えた命令です。32 ビットの結果を XLEN ビットに符号拡張した値をデスティネーションレジスタに書き込みます。

`generate_div_op` 関数に `is_op32` フラグを追加して、`is_op32` が 1 なら値を `DIV_WIDTH` ビットに拡張したものに変更します（リスト 8.37）。

▼リスト 8.37: 除数、被除数を 32 ビットの値にする (`muldivunit.veryl`)

```

1  function generate_div_op (
2      is_op32: input logic ,
3      funct3 : input logic<3> ,
4      value   : input logic<XLEN>,
5  ) -> logic<DIV_WIDTH> {
6      return case funct3[1:0] {
7          2'b00, 2'b10: abs::<DIV_WIDTH>(if is_op32 ? sext::<32, DIV_WIDTH>(value[31:0]) : va>
>lue), // DIV, REM
8          2'b01, 2'b11: if is_op32 ? {1'b0 repeat DIV_WIDTH - 32, value[31:0]} : value, // DI>
>VU, REMU
9          default      : 0,
10         };
11     }
12
13     let du_dividend: logic<DIV_WIDTH> = generate_div_op(is_op32, funct3, op1);
14     let du_divisor : logic<DIV_WIDTH> = generate_div_op(is_op32, funct3, op2);
```

符号付き除算のオーバーフローとゼロ除算の判定を `is_op32` で変更します（リスト 8.38）。

▼リスト 8.38: 32 ビット演算のときの例外的な処理に対応する (`muldivunit.veryl`)

```

1  always_comb {
2      if is_op32 {
3          du_signed_overflow = !funct3[0] && op1[31] == 1 && op1[31:0] == 0 && &op2[31:0];
4          du_signed_divzero = !funct3[0] && op2[31:0] == 0;
5      } else {
6          du_signed_overflow = !funct3[0] && op1[msb] == 1 && op1[msb - 1:0] == 0 && &op2;
7          du_signed_divzero = !funct3[0] && op2 == 0;
8      }
9      du_signed_error = du_signed_overflow || du_signed_divzero;
10 }
```

最後に、32 ビットの結果を XLEN ビットに符号拡張します（リスト 8.39）。符号付き、符号無し

演算のどちらも 32 ビットの結果を符号拡張したものが結果になります。

▼リスト 8.39: 32 ビット演算のとき、結果を符号拡張する (muldivunit.veril)

```
1 } else if !is_mul && du_rvalid {
2     let quo_signed: logic<DIV_WIDTH> = if op1sign_saved != op2sign_saved ? ~du_quotient + 1
3     : du_quotient;
4     let rem_signed: logic<DIV_WIDTH> = if op1sign_saved == 1 ? ~du_remainder + 1 : du_rema
5     inder;
6     let resultX : UIntX           = case funct3_saved[1:0] {
7         2'b00 : quo_signed[XLEN - 1:0], // DIV
8         2'b01 : du_quotient[XLEN - 1:0], // DIVU
9         2'b10 : rem_signed[XLEN - 1:0], // REM
10        2'b11 : du_remainder[XLEN - 1:0], // REMU
11        default: 0,
12    };
13    state  = State::Finish;
14    result = if is_op32_saved ? sext::<32, 64>(resultX[31:0]) : resultX;
15 }
```

riscv-tests の `rv64um-p-` から始まるテストを実行し、成功することを確認してください。

これで M 拡張を実装できました。

# 第9章

## 例外の実装

### 9.1 例外とは何か？

CPU がソフトウェアを実行するとき、処理を中断したり終了しなければならないような異常な状態<sup>\*1</sup>が発生することがあります。例えば、実行環境 (EEI) がサポートしていない、または実行を禁止しているような違法 (illegal)<sup>\*2</sup>な命令を実行しようとする場合です。このとき、CPU はどのような動作をすればいいのでしょうか？

RISC-V では、命令によって引き起こされる異常な状態のことを**例外 (Exception)** と呼び、例外が発生した場合には**トラップ (Trap)** を引き起こします。トラップとは例外、または割り込み (Interrupt)<sup>\*3</sup>によって CPU の状態、制御を変更することです。具体的には PC をトラップベクタ (trap vector) に移動したり、CSR を変更します。

本書では既に ECALL 命令の実行によって発生する Environment call from M-mode 例外を実装しており、例外が発生したら次のように動作します。

1. mcause レジスタにトラップの発生原因を示す値 ( 11 ) を書き込む
2. mepc レジスタに PC の値を書き込む
3. PC を mtvec レジスタの値に設定する

本章では、例外発生時に例外に固有の情報を書き込む mtval レジスタと、現在の実装で発生する可能性がある例外を実装します。本書ではこれ以降、トラップの発生原因を示す値のことを cause と呼びます。

<sup>\*1</sup> 異常な状態 (unusual condition)。予期しない (unexpected) 事象と呼ぶ場合もあります。

<sup>\*2</sup> 不正と呼ぶこともあります。逆に実行できる命令のことを合法 (legal) な命令と呼びます

<sup>\*3</sup> 割り込みは第 14 章「M-mode の実装 (2. 割り込みの実装)」で実装します。

## 9.2

## 例外情報の伝達

### 9.2.1 Environment call from M-mode 例外を IF ステージで処理する

今のところ、ECALL 命令による例外は MEM(CSR) ステージの csrunit モジュールで例外判定、処理されています。ECALL 命令によって例外が発生するかは命令が ECALL であるかどうかだけを判定すれば分かるため、命令をデコードする時点、つまり ID ステージで判定できます。

本章で実装する例外には MEM ステージよりも前で発生する例外があるため、ID ステージから順に次のステージに例外の有無、cause を受け渡していく仕組みを実装します。

まず、例外が発生するかどうか (`valid`)、例外の cause (`cause`) をまとめた `ExceptionInfo` 構造体を定義します (リスト 9.1)。

#### ▼リスト 9.1: ExceptionInfo 構造体を定義する (corectrl.veryl)

```

1 // 例外の情報を保存するための型
2 struct ExceptionInfo {
3     valid: logic ,
4     cause: CsrCause,
5 }
```

EX ステージ、MEM ステージの FIFO のデータ型に構造体を追加します (リスト 9.2、リスト 9.3)。

#### ▼リスト 9.2: EX ステージの FIFO に ExceptionInfo を追加する (core.veryl)

```

1 struct exq_type {
2     addr: Addr      ,
3     bits: Inst     ,
4     ctrl: InstCtrl ,
5     imm : UIntX    ,
6     expt: ExceptionInfo,
7 }
```

#### ▼リスト 9.3: MEM ステージの FIFO に ExceptionInfo を追加する (core.veryl)

```

1 struct memq_type {
2     addr      : Addr      ,
3     bits     : Inst     ,
4     ctrl     : InstCtrl ,
5     imm      : UIntX    ,
6     expt    : ExceptionInfo ,
7     alu_result: UIntX    ,
8     rs1_addr : logic      <5>,
```

ID ステージから EX ステージに命令を渡すとき、命令が ECALL 命令なら例外が発生することを伝えます (リスト 9.4)。

## ▼リスト 9.4: ID ステージで ECALL 命令を判定する (core.veryl)

```

1  always_comb {
2      // ID -> EX
3      if_fifo_rready = exq_wready;
4      exq_wvalid     = if_fifo_rvalid;
5      exq_wdata.addr = if_fifo_rdata.addr;
6      exq_wdata.bits = if_fifo_rdata.bits;
7      exq_wdata.ctrl = ids_ctrl;
8      exq_wdata.imm  = ids_imm;
9      // exception
10     exq_wdata.expt.valid = ids_inst_bits == 32'h00000073; // ECALL
11     exq_wdata.expt.cause = CsrCause::ENVIRONMENT_CALL_FROM_M_MODE;
12 }

```

EX ステージで例外は発生しないので、例外情報をそのまま MEM ステージに渡します（リスト 9.5）。

## ▼リスト 9.5: EX ステージから MEM ステージに例外情報を渡す (core.veryl)

```

1  always_comb {
2      // EX -> MEM
3      exq_rready          = memq_wready && !exs_stall;
4      ...
5      memq_wdata.jump_addr = if inst_is_br(exs_ctrl) ? exs_pc + exs_imm : exs_alu_result & ~
>1;
6      memq_wdata.expt      = exq_rdata.expt;
7 }

```

csrunit モジュールを変更します。 `expt_info` ポートを追加して、MEM ステージ以前の例外情報を受け取ります（リスト 9.6、リスト 9.7、リスト 9.8）。

## ▼リスト 9.6: csrunit モジュールに例外情報を受け取るためのポートを追加する (csrunit.veryl)

```

1 module csrunit (
2     clk      : input  clock           ,
3     rst      : input  reset           ,
4     valid    : input  logic           ,
5     pc       : input  Addr            ,
6     ctrl     : input  InstCtrl       ,
7     expt_info : input  ExceptionInfo ,
8     rd_addr  : input  logic <5>      ,

```

## ▼リスト 9.7: MEM ステージの例外情報の変数を作成する (core.veryl)

```

1 //////////////////////////////// MEM Stage ///////////////////////////////
2 var mems_is_new   : logic      ;
3 let mems_valid    : logic      = memq_rvalid;
4 let mems_pc       : Addr       = memq_rdata.addr;
5 let mems_inst_bits: Inst      = memq_rdata.bits;
6 let mems_ctrl     : InstCtrl  = memq_rdata.ctrl;
7 let mems_expt    : ExceptionInfo = memq_rdata.expt;
8 let mems_rd_addr : logic <5> = mems_inst_bits[11:7];

```

## ▼リスト 9.8: csrunit モジュールに例外情報を供給する (core.veryl)

```

1 inst csrunit: csrunit (
2     clk
3     ,
4     rst
5     ,
6     valid : mems_valid
7     ,
8     pc : mems_pc
9     ,
10    ctrl : mems_ctrl
11    ,
12    expt_info: mems_expt
13    ,
14    rd_addr : mems_rd_addr
15    ,

```

ECALL 命令かどうかを判定する `is_ecall` 変数を削除して、例外の発生条件、例外の種類を示す値を変更します（リスト 9.9、リスト 9.10）。

## ▼リスト 9.9: csrunit モジュールでの ECALL 命令の判定を削除する (csrunit.veryl)

```

1 // CSRR(W|S|C)[I]命令かどうか
2 let is_wsc: logic = ctrl.is_csr && ctrl.funct3[1:0] != 0;
3 // ECALL命令かどうか
4 let is_ecall: logic = ctrl.is_csr && csr_addr == 0 && rs1[4:0] == 0 && ctrl.funct3 == 0 &&
5 rd_addr == 0;

```

## ▼リスト 9.10: ExceptionInfo を使って例外を起こす (csrunit.veryl)

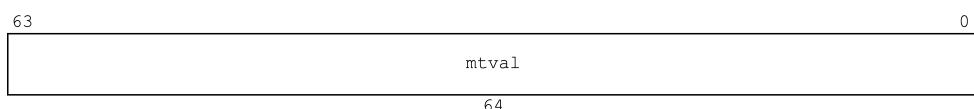
```

1 // Exception
2 let raise_expt: logic = valid && expt_info.valid;
3 let expt_cause: UIntX = expt_info.cause;

```

## 9.2.2 mtval レジスタを実装する

例外が発生すると、CPU はトラップベクタにジャンプして例外を処理します。mcause レジスタを読むことでどの例外が発生したかを判別できますが、その例外の詳しい情報を知りたいことがあります。



▲図 9.1: mtval レジスタ

RISC-V には、例外が発生したときのソフトウェアによるハンドリングを補助するために、MXLEN ビットの mtval レジスタが定義されています（図 9.1）。例外が発生したとき、CPU は mtval レジスタに例外に固有の情報を書き込みます。これ以降、例外に固有の情報のことを tval と呼びます。

`ExceptionInfo` 構造体に例外に固有の情報を示す `value` を追加します（リスト 9.11）。

## ▼リスト 9.11: tval を ExceptionInfo に追加する (corectrl.veryl)

```

1 struct ExceptionInfo {
2     valid: logic ,
3     cause: CsrCause,
4     value: UIntX ,
5 }
```

ECALL 命令は mtval に書き込むような情報がないので 0 に設定します (リスト 9.12)。

## ▼リスト 9.12: ECALL 命令の tval を設定する (corectrl.veryl)

```

1 // exception
2 exq_wdata.expt.valid = ids_inst_bits == 32'h00000073; // ECALL
3 exq_wdata.expt.cause = CsrCause::ENVIRONMENT_CALL_FROM_M_MODE;
4 exq_wdata.expt.value = 0;
```

CsrAddr 型に mtval レジスタのアドレスを追加します (リスト 9.13)。

## ▼リスト 9.13: mtval のアドレスを定義する (eei.veryl)

```

1 enum CsrAddr: logic<12> {
2     MTVEC = 12'h305,
3     MEPC = 12'h341,
4     MCAUSE = 12'h342,
5     MTVAL = 12'h343,
6     LED = 12'h800,
7 }
```

mtval レジスタを実装して、書き込み、読み込みできるようにします (リスト 9.14、リスト 9.15、リスト 9.16、リスト 9.17、リスト 9.18)。

## ▼リスト 9.14: mtval の書き込みマスクを定義する (csrunit.veryl)

```
1 const MTVAL_WMASK : UIntX = 'hffff_ffff_ffff_ffff;
```

## ▼リスト 9.15: mtval レジスタを作成する (csrunit.veryl)

```

1 var mtvec : UIntX;
2 var mepc : UIntX;
3 var mcause: UIntX;
4 var mtval : UIntX;
```

## ▼リスト 9.16: mtval の読み込みデータ、書き込みマスクを設定する (csrunit.veryl)

```

1 always_comb {
2     // read
3     rdata = case csr_addr {
4         ...
5             CsrAddr::MTVAL : mtval,
6             ...
7     };
8     // write
9     wmask = case csr_addr {
```

```

10     ...
11     CsrAddr::MTVAL : MTVAL_WMASK,
12     ...
13 };

```

## ▼リスト 9.17: mtval レジスタをリセットする (csrunit.veryl)

```

1 always_ff {
2     if_reset {
3         mtvec = 0;
4         mepc = 0;
5         mcause = 0;
6         mtval = 0;
7         led = 0;

```

## ▼リスト 9.18: mtval に書き込めるようにする (csrunit.veryl)

```

1 } else {
2     if is_wsc {
3         case csr_addr {
4             ...
5                 CsrAddr::MTVAL : mtval = wdata;
6                 ...
7             }
8         }
9     }

```

例外が発生するとき、mtval レジスタに `expt_info.value` を書き込むようにします（リスト 9.19、リスト 9.20）。

## ▼リスト 9.19: tval を変数に割り当てる (csrunit.veryl)

```

1 let raise_expt : logic = valid && expt_info.valid;
2 let expt_cause : UIntX = expt_info.cause;
3 let expt_value : UIntX = expt_info.value;

```

## ▼リスト 9.20: 例外が発生するとき、mtval に tval を書き込む (csrunit.veryl)

```

1 if valid {
2     if raise_trap {
3         if raise_expt {
4             mepc = pc;
5             mcause = trap_cause;
6             mtval = expt_value;
7         }
}

```

## 9.3 Breakpoint 例外の実装

Breakpoint 例外は、EBREAK 命令によって引き起こされる例外です。EBREAK 命令はデバッガがプログラムを中断させる場合などに利用されます。EBREAK 命令は ECALL 命令と同様に

例外を発生させるだけで、ほかに操作を行いません。causeは3で、tvalは例外が発生した命令のアドレスになります。

`CsrCause`型にBreakpoint例外のcauseを追加します(リスト9.21)。

▼リスト9.21: Breakpoint例外のcauseを定義する(`eei.veryl`)

```
1 enum CsrCause: UIntX {
2     BREAKPOINT = 3,
3     ENVIRONMENT_CALL_FROM_M_MODE = 11,
4 }
```

IDステージでEBREAK命令を判定して、tvalにPCを設定します(リスト9.22)。

▼リスト9.22: IDステージでEBREAK命令を判定する(`core.veryl`)

```
1 exq_wdata.expt = 0;
2 if ids_inst_bits == 32'h00000073 {
3     // ECALL
4     exq_wdata.expt.valid = 1;
5     exq_wdata.expt.cause = CsrCause::ENVIRONMENT_CALL_FROM_M_MODE;
6     exq_wdata.expt.value = 0;
7 } else if ids_inst_bits == 32'h00100073 {
8     // EBREAK
9     exq_wdata.expt.valid = 1;
10    exq_wdata.expt.cause = CsrCause::BREAKPOINT;
11    exq_wdata.expt.value = ids_pc;
12 }
```

## 9.4 Illegal instruction 例外の実装

Illegal instruction例外は、現在の環境で実行できない命令を実行しようとしたときに発生する例外です。causeは2で、tvalは例外が発生した命令のビット列になります。

本章では、EEIが認識できない不正な命令ビット列を実行しようとした場合と、読み込み専用のCSRに書き込もうとした場合の2つの状況で例外を発生させます。

### 9.4.1 不正な命令ビット列で例外を起こす

CPUに実装していない命令、つまりデコードできない命令を実行しようとするとき、Illegal instruction例外が発生します。

今のところ未知の命令は何もしない命令として実行しています。ここで、`inst_decoder`モジュールを、未知の命令であることを報告するように変更します。

`inst_decoder`モジュールに、命令が有効かどうかを示す`valid`ポートを追加します(リスト9.23、リスト9.24)。

## ▼リスト 9.23: valid ポートを追加する (inst\_decoder.veryl)

```

1 module inst_decoder (
2     bits : input Inst      ,
3     valid: output logic   ,
4     ctrl : output InstCtrl,
5     imm  : output UIntX   ,
6 ) {

```

## ▼リスト 9.24: inst\_decoder モジュールの valid ポートと変数を接続する (core.veryl)

```

1 let ids_valid      : logic    = if_fifo_rvalid;
2 let ids_pc        : Addr     = if_fifo_rdata.addr;
3 let ids_inst_bits : Inst     = if_fifo_rdata.bits;
4 var ids_inst_valid: logic   ;
5 var ids_ctrl       : InstCtrl;
6 var ids_imm        : UIntX   ;
7
8 inst decoder: inst_decoder (
9     bits : ids_inst_bits ,
10    valid: ids_inst_valid,
11    ctrl : ids_ctrl       ,
12    imm  : ids_imm        ,
13 );

```

今のところ実装してある命令を有効な命令として判定する処理を always\_comb ブロックに記述します（リスト 9.25）。

## ▼リスト 9.25: 命令の有効判定を行う (inst\_decoder.veryl)

```

1 valid = case op {
2     OP_LUI, OP_AUIPC, OP_JAL, OP_JALR: T,
3     OP_BRANCH                  : f3 != 3'b010 && f3 != 3'b011,
4     OP_LOAD                    : f3 != 3'b111,
5     OP_STORE                   : f3[2] == 1'b0,
6     OP_OP                      : case f7 {
7         7'b0000000              : T, // RV32I
8         7'b0100000              : f3 == 3'b000 || f3 == 3'b101, // SUB, SRA
9         7'b0000001              : T, // RV32M
10        default                 : F,
11    },
12    OP_OP_IMM: case f3 {
13        3'b001 : f7[6:1] == 6'b000000, // SLLI (RV64I)
14        3'b101 : f7[6:1] == 6'b000000 || f7[6:1] == 6'b010000, // SRLI, SRAI (RV64I)
15        default : T,
16    },
17    OP_OP_32 : case f7 {
18        7'b0000001: f3 == 3'b000 || f3[2] == 1'b1, // RV64M
19        7'b0000000: f3 == 3'b000 || f3 == 3'b001 || f3 == 3'b101, // ADDW, SLLW, SRLW
20        7'b0100000: f3 == 3'b000 || f3 == 3'b101, // SUBW, SRAW
21        default   : F,
22    },
23    OP_OP_IMM_32: case f3 {
24        3'b000 : T, // ADDIW

```

```

25      3'b001      : f7 == 7'b0000000, // SLLIW
26      3'b101      : f7 == 7'b0000000 || f7 == 7'b0100000, // SRLIW, SRAIW
27      default     : F,
28 },
29 OP_SYSTEM: f3 != 3'b000 && f3 != 3'b100 || // CSRR(W|S|C)[I]
30     bits == 32'h00000073 || // ECALL
31     bits == 32'h00100073 || // EBREAK
32     bits == 32'h30200073, // MRET
33 OP_MISC_MEM: T, // FENCE
34 default     : F,
35 };

```

riscv-tests でメモリ読み書きの順序を保証する FENCE 命令<sup>\*4</sup>を使用しているため、opcode が OP-MISC である命令を合法な命令として取り扱っています。OP-MISC の opcode( 7'b0001111 ) を eei パッケージに定義してください (リスト 9.26)。

#### ▼リスト 9.26: OP-MISC のビット列を定義する (eei.veryl)

```
1 const OP_MISC_MEM : logic<7> = 7'b0001111;
```

CsrCause 型に Illegal instruction 例外の cause を追加します (リスト 9.27)。

#### ▼リスト 9.27: Illegal instruction 例外の cause を定義する (eei.veryl)

```

1 enum CsrCause: UIntX {
2     ILLEGAL_INSTRUCTION = 2,
3     BREAKPOINT = 3,
4     ENVIRONMENT_CALL_FROM_M_MODE = 11,
5 }

```

valid フラグを利用して、ID ステージで Illegal instruction 例外を発生させます (リスト 9.28)。tval には、命令を右に詰めてゼロで拡張した値を設定します。

#### ▼リスト 9.28: 不正な命令のとき、例外を発生させる (core.veryl)

```

1     exq_wdata.expt = 0;
2     if !ids_inst_valid {
3         // illegal instruction
4         exq_wdata.expt.valid = 1;
5         exq_wdata.expt.cause = CsrCause::ILLEGAL_INSTRUCTION;
6         exq_wdata.expt.value = {1'b0 repeat XLEN - ILEN, ids_inst_bits};
7     } else if ids_inst_bits == 32'h00000073 {

```

## 9.4.2 読み込み専用の CSR への書き込みで例外を起こす

RISC-V の CSR には読み込み専用のレジスタが存在しており、アドレスの上位 2 ビットが 2'b11 の CSR が読み込み専用として定義されています。読み込み専用の CSR に書き込みを行おうとすると Illegal instruction 例外が発生します。

---

<sup>\*4</sup> 基本編で実装する CPU はロードストア命令を直列に実行するため順序を保証する必要がありません。そのため FENCE 命令は何もしない命令として扱います。

CSR に値が書き込まれるのは次のいずれかの場合です。読み書き可能なレジスタ内の読み込み専用のフィールドへの書き込みは例外を引き起しません。

1. CSRRW、CSRRWI 命令である
2. CSRRS 命令で rs1 が 0 番目のレジスタ以外である
3. CSRRSI 命令で即値が 0 以外である
4. CSRRC 命令で rs1 が 0 番目のレジスタ以外である
5. CSRRCI 命令で即値が 0 以外である

ソースレジスタの値が 0 だとしても、0 番目のレジスタではない場合には CSR に書き込むと判断します。CSR に書き込むかどうかを正しく判定するために、csrunit モジュールの `rs1` ポートを `rs1_addr` と `rs1_data` に分解します（リスト 9.30、リスト 9.29、リスト 9.31）\*5。また、cause を設定するために csrunit モジュールに命令のビット列を供給します。

#### ▼リスト 9.29: csrunit モジュールのポート定義を変更する (csrunit.veryl)

```

1 module csrunit (
2     clk      : input  clock          ,
3     rst      : input  reset          ,
4     valid    : input  logic          ,
5     pc       : input  Addr          ,
6     inst_bits : input  Inst          ,
7     ctrl     : input  InstCtrl      ,
8     expt_info : input  ExceptionInfo ,
9     rd_addr  : input  logic <5> ,
10    csr_addr : input  logic <12>,
11    rs1_addr : input  logic <5> ,
12    rs1_data : input  UIntX         ,
13    rdata    : output UIntX         ,
14    raise_trap: output logic       ,
15    trap_vector: output Addr       ,
16    led      : output UIntX         ,
17 ) {

```

#### ▼リスト 9.30: csrunit モジュールのポート定義を変更する (core.veryl)

```

1 inst csru: csrunit (
2     clk          ,
3     rst          ,
4     valid       : mems_valid      ,
5     pc          : mems_pc        ,
6     inst_bits   : mems_inst_bits ,
7     ctrl        : mems_ctrl      ,
8     expt_info   : mems_expt      ,
9     rd_addr    : mems_rd_addr   ,
10    csr_addr   : mems_inst_bits[31:20],
11    rs1_addr   : memq_rdata.rs1_addr ,
12    rs1_data   : memq_rdata.rs1_data ,
13    rdata      : csru_rdata      ,

```

\*5 基本編 第1部の初版の `wdata` の生成ロジックに間違いがあったので訂正しております。

```

14     raise_trap : csru_raise_trap      ,
15     trap_vector: csru_trap_vector   ,
16     led          ,
17 );

```

▼リスト 9.31: rs1 の変更に対応する (csrunit.veryl)

```

1 let wsource: UIntX = if ctrl.funct3[2] ? {1'b0 repeat XLEN - 5, rs1_addr} : rs1_data;
2 wdata  = case ctrl.funct3[1:0] {
3     2'b01  : wsource,
4     2'b10  : rdata | wsource,
5     2'b11  : rdata & ~wsource,
6     default: 'x,
7 } & wmask | (rdata & ~wmask);

```

命令の funct3 と rs1 のアドレスを利用して、書き込み先が読み込み専用レジスタかどうかを判定します<sup>\*6</sup>(リスト 9.32)。また、命令のビット列を利用できるようになったので、MRET 命令の判定を命令のビット列の比較に書き換えています。

▼リスト 9.32: 読み込み専用 CSR への書き込みが発生するか判定する (csrunit.veryl)

```

1 // CSRR(W|S|C)[I]命令かどうか
2 let is_wsc: logic = ctrl.is_csr && ctrl.funct3[1:0] != 0;
3 // MRET命令かどうか
4 let is_mret: logic = inst_bits == 32'h30200073;
5
6 // Check CSR access
7 let will_not_write_csr    : logic = (ctrl.funct3[1:0] == 2 || ctrl.funct3[1:0] == 3) && rs>
8 >1_addr == 0; // set/clear with source = 0
9 let expt_write_READONLY_csr: logic = is_wsc && !will_not_write_csr && csr_addr[11:10] == 2'b>
10 >11; // attempt to write read-only CSR

```

例外が発生するとき、cause と tval を設定します(リスト 9.33)。

▼リスト 9.33: 読み込み専用 CSR の書き込みで例外を発生させる (csrunit.veryl)

```

1 let raise_expt: logic = valid && (expt_info.valid || expt_write_READONLY_csr);
2 let expt_cause: UIntX = switch {
3     expt_info.valid      : expt_info.cause,
4     expt_WRITE_READONLY_CSR: CsrCause::ILLEGAL_INSTRUCTION,
5     default              : 0,
6 };
7 let expt_value: UIntX = switch {
8     expt_info.valid      : expt_info.value,
9     expt_cause == CsrCause::ILLEGAL_INSTRUCTION: {1'b0 repeat XLEN - $bits(Inst), inst_bits}>
10 ,           default              : 0
11 };

```

この変更により、レジスタにライトバックするようにデコードされた命令が csrunit モジュール

<sup>\*6</sup> ID ステージで判定することもできます。

でトラップを起こすようになりました。トラップが発生するときに WB ステージでライトバックしないように変更します（リスト 9.34、リスト 9.35、リスト 9.36）。

▼ リスト 9.34: トラップが発生したかを示す logic を wbq\_type に追加する (core.veryl)

```
1 struct wbq_type {
2     ...
3     csr_rdata : UIntX ,
4     raise_trap: logic ,
5 }
```

▼ リスト 9.35: トラップが発生したかを WB ステージに伝える (core.veryl)

```
1 wbq_wdata.raise_trap = csru_raise_trap;
```

▼ リスト 9.36: トラップが発生しているとき、レジスタにデータを書き込まないようにする (core.veryl)

```
1 always_ff {
2     if wbs_valid && wbs_ctrl.rwb_en && !wbq_rdata.raise_trap {
3         regfile[wbs_rd_addr] = wbs_wb_data;
4     }
5 }
```

## 9.5

## 命令アドレスのミスマッチ例外

RISC-V では、命令アドレスが IALIGN ビット境界に整列されていない場合に Instruction address misaligned 例外が発生します。cause は 0 で、tval は命令のアドレスになります。

第 12 章「C 拡張の実装」で実装する C 拡張が実装されていない場合、IALIGN は 32 と定義されています。C 拡張が定義されている場合は 16 になります。

IALIGN ビット境界に整列されていない命令アドレスになるのはジャンプ命令、分岐命令を実行する場合です<sup>7</sup>。PC の遷移先が整列されていない場合に例外が発生します。分岐命令の場合、分岐が成立する場合にしか例外は発生しません。

CsrCause 型に Instruction address misaligned 例外の cause を追加します（リスト 9.37）。

▼ リスト 9.37: Instruction address misaligned 例外の cause を定義する (eei.veryl)

```
1 enum CsrCause: UIntX {
2     INSTRUCTION_ADDRESS_MISALIGNED = 0,
3     ILLEGAL_INSTRUCTION = 2,
4     BREAKPOINT = 3,
5     ENVIRONMENT_CALL_FROM_M_MODE = 11,
6 }
```

<sup>7</sup> mepc、mtvec は IALIGN ビットに整列されたアドレスしか書き込めないため、遷移先のアドレスは常に整列されています。

EXステージでアドレスを確認して例外を判定します（リスト9.38）。tvalは遷移先のアドレスになることに注意してください。

▼リスト9.38: EXステージでInstruction address misaligned例外の判定を行う(core.veryl)

```

1      memq_wdata.jump_addr = if inst_is_br(exs_ctrl) ? exs_pc + exs_imm : exs_alu_result & ~
2      >1;
3      // exception
4      let instruction_address_misaligned: logic = memq_wdata.br_taken && memq_wdata.jump_addr[>
5      >1:0] != 2'b00;
6      memq_wdata.expt            = exq_rdata.expt;
7      if !memq_rdata.expt.valid {
8          if instruction_address_misaligned {
9              memq_wdata.expt.valid = 1;
10             memq_wdata.expt.cause = CsrCause::INSTRUCTION_ADDRESS_MISALIGNED;
11             memq_wdata.expt.value = memq_wdata.jump_addr;
12         }
13     }

```

## 9.6 ロードストア命令のミスアライン例外

RISC-Vでは、ロード、ストア命令でアクセスするメモリのアドレスが、ロード、ストアするビット幅に整列されていない場合に、それぞれLoad address misaligned例外、Store/AMO address misaligned例外が発生します<sup>\*8</sup>。例えばLW命令は4バイトに整列されたアドレス、LD命令は8バイトに整列されたアドレスにしかアクセスできません。causeはそれぞれ4、6で、tvalはアクセスするメモリのアドレスになります。

CsrCause型に例外のcauseを追加します（リスト9.39）。

▼リスト9.39: 例外のcauseを定義する(eei.veryl)

```

1 enum CsrCause: UIntX {
2     INSTRUCTION_ADDRESS_MISALIGNED = 0,
3     ILLEGAL_INSTRUCTION = 2,
4     BREAKPOINT = 3,
5     LOAD_ADDRESS_MISALIGNED = 4,
6     STORE_AMO_ADDRESS_MISALIGNED = 6,
7     ENVIRONMENT_CALL_FROM_M_MODE = 11,
8 }

```

EXステージでアドレスを確認して例外を判定します（リスト9.40）。

▼リスト9.40: EXステージで例外の判定を行う(core.veryl)

<sup>\*8</sup>例外を発生させず、そのようなメモリアクセスをサポートすることもできます。本書ではCPUを単純に実装するために例外とします。

```

1      let instruction_address_misaligned: logic = memq_wdata.br_taken && memq_wdata.jump_addr[>
>1:0] != 2'b00;
2      let loadstore_address_misaligned : logic = inst_is_memop(exs_ctrl) && case exs_ctrl.fun
>ct3[1:0] {
3          2'b00 : 0, // B
4          2'b01 : exs_alu_result[0] != 1'b0, // H
5          2'b10 : exs_alu_result[1:0] != 2'b0, // W
6          2'b11 : exs_alu_result[2:0] != 3'b0, // D
7          default: 0,
8      };
9      memq_wdata.expt = exq_rdata.expt;
10     if !memq_rdata.expt.valid {
11         if instruction_address_misaligned {
12             memq_wdata.expt.valid = 1;
13             memq_wdata.expt.cause = CsrCause::INSTRUCTION_ADDRESS_MISALIGNED;
14             memq_wdata.expt.value = memq_wdata.jump_addr;
15         } else if loadstore_address_misaligned {
16             memq_wdata.expt.valid = 1;
17             memq_wdata.expt.cause = if exs_ctrl.is_load ? CsrCause::LOAD_ADDRESS_MISALIGNED
> : CsrCause::STORE_AMO_ADDRESS_MISALIGNED;
18             memq_wdata.expt.value = exs_alu_result;
19         }
20     }

```

例外が発生するときに memunit モジュールが動作しないようにします（リスト 9.41）。

▼リスト 9.41: 例外が発生するとき、memunit の valid を 0 にする (core.veryl)

```

1  inst memu: memunit (
2      clk
3      ,
4      rst
5      ,
6      valid : mems_valid && !mems_expt.valid,
7      is_new: mems_is_new
8      ,
9      ctrl : mems_ctrl
10     ,
11     addr : memq_rdata.alu_result
12     ,
13     rs2 : memq_rdata.rs2_data
14     ,
15     rdata : memu_rdata
16     ,
17     stall : memu_stall
18     ,
19     membust: d_membus
20 );

```

## 第 10 章

# Memory-mapped I/O の実装

### 10.1 Memory-mapped I/O とは何か？

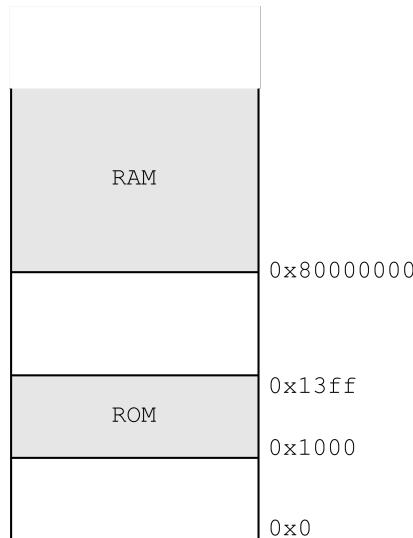
これまでの実装では、CPU に内蔵された 1 つの大きなメモリ空間、1 つのメモリデバイス (memory モジュール) に命令データを格納、実行し、データのロードストア命令も同じメモリに対して実行してきました。

一般に流通するコンピュータは複数のデバイスに接続されています。CPU が起動すると、読み込み専用の小さなメモリ (ROM) に格納されたプログラムから命令の実行を開始します。プログラムは周辺デバイスの初期化などを行ったあと、動かしたいアプリケーションの命令やデータを RAM に展開して、制御をアプリケーションに移します。

CPU がデバイスにアクセスする方法には CSR やメモリ空間を経由する方法があります。一般的な方法はメモリ空間を通じてデバイスにアクセスする方法であり、この方式のことをメモリマップド IO (Memory-mapped I/O, MMIO) と呼びます。メモリ空間の一部を、デバイスにアクセスするための空間として扱うことを、メモリ (またはアドレス) にマップすると呼びます。RAM と ROM もメモリデバイスであり、異なるアドレスにマップされています。

本章では CPU のメモリ部分を RAM(Random Access Memory)<sup>\*1</sup> と ROM(Read Only Memory) に分割し、アクセスするアドレスに応じてアクセスするデバイスを切り替える機能を実装します。また、デバッグ用の入出力デバイス (64 ビットのレジスタ) も実装します。デバイスとメモリ空間の対応は図 10.1 のように設定します。図 10.1 のようにメモリがどのように配置されているかを示す図のことをメモリマップ (Memory map) と呼びます。あるメモリ空間の先頭アドレスのことをベースアドレス (base address) と呼ぶことがあります。

<sup>\*1</sup> 本章では実際の RAM デバイスへのアクセスを実装せず memory モジュールで代用します。FPGA に合成するときに実際のデバイスへのアクセスに置き換えます。



▲図 10.1: メモリマップ

## 10.2 定数の定義

eei パッケージに定義しているメモリの定数を RAM 用の定数に変更します。また、新しく RAM のベースアドレス、メモリバスのデータ幅、ROM のメモリマップを示す定数を定義してください（リスト 10.1）。デバッグ入出力デバイス（レジスタ）の位置は、top モジュールのポートで定義します（リスト 10.9）。

### ▼リスト 10.1: メモリマップの定義 (eei.veryl)

```

1 // メモリバスのデータ幅
2 const MEMBUS_DATA_WIDTH: u32 = 64;
3 // メモリのアドレス幅
4 const MEM_ADDR_WIDTH: u32 = 16;
5
6 // RAM
7 const RAM_ADDR_WIDTH: u32 = 16;
8 const RAM_DATA_WIDTH: u32 = 64;
9 const MMAP_RAM_BEGIN: Addr = 'h8000_0000 as Addr;
10
11 // ROM
12 const ROM_ADDR_WIDTH: u32 = 9;
13 const ROM_DATA_WIDTH: u32 = 64;
14 const MMAP_ROM_BEGIN: Addr = 'h1000 as Addr;
15 const MMAP_ROM_END : Addr = MMAP_ROM_BEGIN + 'h3ff as Addr;
```

`MEM_DATA_WIDTH`、`MEM_ADDR_WIDTH` を使っている部分を `MEMBUS_DATA_WIDTH`、`XLEN` に置き換えます。`MEMBUS_DATA_WIDTH` と `XLEN` を使う membus\_if インターフェースに別名 `Membus` をつけ

て利用します（リスト 10.2、リスト 10.3、リスト 10.4、リスト 10.5）。

▼ リスト 10.2: 別名の定義 (membus\_if.veryl)

```
1 alias interface Membus = membus_if::<eei::MEMBUS_DATA_WIDTH, eei::XLEN>;
```

▼ リスト 10.3: Membus に置き換える (core.veryl)

```
1 module core (
2     clk      : input    clock                  ,
3     rst      : input    reset                  ,
4     i_membus: modport membus_if::<ILEN, XLEN>::master,
5     d_membus: modport Membus::master          ,
6     led      : output   UIntX                ,
7 ) {
```

▼ リスト 10.4: Membus に置き換える (memunit.veryl)

```
1 membus: modport Membus::master, // メモリとのinterface
```

▼ リスト 10.5: 定数名を変更する (memunit.veryl)

```
1 var req_wen  : logic           ;
2 var req_addr : Addr            ;
3 var req_wdata: logic<MEMBUS_DATA_WIDTH>   ;
4 var req_wmask: logic<MEMBUS_DATA_WIDTH / 8>;
5
6 const W    : u32              = XLEN;
7 let D    : logic<MEMBUS_DATA_WIDTH> = membus.rdata;
8 let sext: logic               = ctrl.funct3[2] == 1'b0;
```

top モジュールでインスタンス化している membus\_if インターフェースのジェネリックパラメータを変更します（リスト 10.6）。

▼ リスト 10.6: ジェネリックパラメータを変更する / Membus に置き換える (top.veryl)

```
1 inst membus  : membus_if::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>;
2 inst i_membus: membus_if::<ILEN, XLEN>; // 命令フェッチ用
3 inst d_membus: Membus; // ロードストア命令用
```

addr\_to\_memaddr 関数をジェネリック関数にして、呼び出すときに RAM のパラメータを使用するように変更します（リスト 10.7、リスト 10.8、）。

▼ リスト 10.7: addr\_to\_memaddr 関数をジェネリック関数に変更する (top.veryl)

```
1 // アドレスをデータ単位でのアドレスに変換する
2 function addr_to_memaddr::<DATA_WIDTH: u32, ADDR_WIDTH: u32> (
3     addr: input logic<XLEN>,
4 ) -> logic<ADDR_WIDTH> {
5     return addr[$clog2(DATA_WIDTH / 8)+:ADDR_WIDTH];
6 }
```

## ▼リスト 10.8: ジェネリックパラメータを指定する (top.veryl)

```

1    memb.us.valid = i_membus.valid | d_membus.valid;
2    if d_membus.valid {
3        memb.us.addr = addr_to_memaddr::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>(d_membus.addr);
4        memb.us.wen = d_membus.wen;
5        memb.us.wdata = d_membus.wdata;
6        memb.us.wmask = d_membus.wmask;
7    } else {
8        memb.us.addr = addr_to_memaddr::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>(i_membus.addr);
9        memb.us.wen = 0; // 命令フェッチは常に読み込み
10       memb.us.wdata = 'x;
11       memb.us.wmask = 'x;
12   }

```

メモリに読み込む HEX ファイルを指定するパラメータの名前を変更します（リスト 10.9、リスト 10.10）。

## ▼リスト 10.9: パラメータ名を変更する (top.veryl)

```

1 module top #(
2     param RAM_FILEPATH_IS_ENV: bit      = 1
3     param RAM_FILEPATH      : string = "RAM_FILE_PATH",
4 ) (
5     clk          : input clock,
6     rst          : input reset,
7     MMAP_DBG_ADDR: input Addr ,

```

## ▼リスト 10.10: パラメータ名を変更する (top.veryl)

```

1 inst ram: memory::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH> #(
2     FILEPATH_IS_ENV: RAM_FILEPATH_IS_ENV,
3     FILEPATH      : RAM_FILEPATH      ,
4 ) (

```

シミュレータ用の C++ プログラムも変更します（リスト 10.11、リスト 10.12、リスト 10.13）。

## ▼リスト 10.11: 引数の名称を変える (tb\_verilator.cpp)

```

1 if (argc < 2) {
2     std::cout << "Usage: " << argv[0] << " RAM_FILE_PATH [CYCLE]" << std::endl;
3     return 1;
4 }

```

## ▼リスト 10.12: 環境変数名を変える (tb\_verilator.cpp)

```

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

```

## ▼リスト 10.13: 環境変数名を変える (tb\_verilator.cpp)

```

1 // 環境変数を元に戻す
2 if (original_env != nullptr){
3     setenv("RAM_FILE_PATH", original_env, 1);
4 }
```

## 10.3 mmio\_controller モジュールの作成

アクセスするアドレスに応じてアクセス先のデバイスを切り替えるモジュールを実装します。

`src/mmio_controller.veryl` を作成し、次のように記述します（リスト 10.14）。

## ▼リスト 10.14: mmio\_controller.veryl

```

1 import eei::*;
2
3 module mmio_controller (
4     clk      : input    clock      ,
5     rst      : input    reset      ,
6     req_core: modport Membus::slave,
7 ) {
8
9     enum Device {
10         UNKNOWN,
11     }
12
13     inst req_saved: Membus;
14
15     var last_device : Device;
16     var is_requested: logic ;
17
18     // masterを0でリセットする
19     function reset_membus_master (
20         master: modport Membus::master_output,
21     ) {
22         master.valid = 0;
23         master.addr  = 0;
24         master.wen   = 0;
25         master.wdata  = 0;
26         master.wmask  = 0;
27     }
28
29     // すべてのデバイスのmasterをリセットする
30     function reset_all_device_masters () {}
31
32     // アドレスからデバイスを取得する
33     function get_device (
34         addr: input Addr,
35     ) -> Device {
36         return Device::UNKNOWN;
```

```
37 }
38
39 // デバイスのmasterにreqの情報を割り当てる
40 function assign_device_master (
41     req: modport Membus::all_input,
42 ) {}
43
44 // デバイスのrvalid、rdataをreqに割り当てる
45 function assign_device_slave (
46     device: input Device ,
47     req : modport Membus::response,
48 ) {
49     req.rvalid = 1;
50     req.rdata  = 0;
51 }
52
53 // デバイスのreadyを取得する
54 function get_device_ready (
55     device: input Device,
56 ) -> logic {
57     return 1;
58 }
59
60 // デバイスのrvalidを取得する
61 function get_device_rvalid (
62     device: input Device,
63 ) -> logic {
64     return 1;
65 }
66
67 // req_coreの割り当て
68 always_comb {
69     req_core.ready  = 0;
70     req_core.rvalid = 0;
71     req_core.rdata  = 0;
72
73     if req_saved.valid {
74         if is_requested {
75             // 結果を返す
76             assign_device_slave(last_device, req_core);
77             req_core.ready      = get_device_rvalid(last_device);
78         }
79     } else {
80         req_core.ready = 1;
81     }
82 }
83
84 // デバイスのmasterの割り当て
85 always_comb {
86     reset_all_device_masters();
87     if req_saved.valid {
88         if is_requested {
89             if get_device_rvalid(last_device) {
```

```
90          // 新しく要求を受け入れる
91          if req_core.ready && req_core.valid {
92              assign_device_master(req_core);
93          }
94      } else {
95          // デバイスにreq_savedを割り当てる
96          assign_device_master(req_saved);
97      }
98  } else {
99      // 新しく要求を受け入れる
100     if req_core.ready && req_core.valid {
101         assign_device_master(req_core);
102     }
103 }
104 }
105 }

106 // 新しく要求を受け入れる
107 function accept_request () {
108     req_saved.valid = req_core.ready && req_core.valid;
109     if req_core.ready && req_core.valid {
110         last_device = get_device(req_core.addr);
111         is_requested = get_device_ready(last_device);
112         // reqを保存
113         req_saved.addr = req_core.addr;
114         req_saved.wen = req_core.wen;
115         req_saved.wdata = req_core.wdata;
116         req_saved.wmask = req_core.wmask;
117     }
118 }
119 }
120 }

121 function on_clock () {
122     if req_saved.valid {
123         if is_requested {
124             if get_device_rvalid(last_device) {
125                 accept_request();
126             }
127         } else {
128             is_requested = get_device_ready(last_device);
129         }
130     } else {
131         accept_request();
132     }
133 }
134 }

135 function on_reset () {
136     last_device = Device::UNKNOWN;
137     is_requested = 0;
138     reset_membus_master(req_saved);
139 }
140 }

141 always_ff {
142     if_reset {
```

```

143         on_reset();
144     } else {
145         on_clock();
146     }
147 }
148 }
```

mmio\_controller モジュールの関数の引数に membus\_if インターフェースを使うために、新しい modport を宣言します（リスト 10.15）。

#### ▼リスト 10.15: modport 宣言を追加する (membus\_if.veryl)

```

1 modport all_input {
2     ..input
3 }
4
5 modport response {
6     rvalid: output,
7     rdata : output,
8 }
9
10 modport slave_output {
11     ready: output,
12     ..same(response)
13 }
14
15 modport master_output {
16     valid: output,
17     addr : output,
18     wen  : output,
19     wdata: output,
20     wmask: output,
21 }
```

mmio\_controller モジュールは req\_core からメモリアクセス要求を受け付け、アクセス対象のモジュールからの結果を返すモジュールです。

**Device** 型は実装しているデバイスを表現するための列挙型です（リスト 10.16）。まだデバイスを接続していないので、不明なデバイス（`Device::UNKNOWN`）だけ定義しています。

#### ▼リスト 10.16: Device 型の定義 (mmio\_controller.veryl)

```

1 enum Device {
2     UNKNOWN,
3 }
```

`reset_membus_master`、`reset_all_device_masters` 関数はインターフェースの値の割り当てを `0` でリセットするためのユーティリティ関数です。名前が `get_device_`、`assign_device` から始まる関数は、デバイスの状態を取得したり、インターフェースに値を割り当てる関数です。`get_device` 関数はアドレスに対応する `Device` を取得する関数です。

`always_comb`、`always_ff` ブロックはこれらの関数を利用してメモリアクセスを制御します。

`always_ff` ブロックは、メモリアクセス要求の処理中ではない場合とメモリアクセスが終わった場合にメモリアクセス要求を受け入れます。要求を受け入れるとき、`req_core` の値を `req_saved` に保存します。

`always_comb` ブロックはデバイスにアクセスし `req_core` に結果を返します。`is_requested` は、メモリアクセス要求を処理している場合に既にデバイスが要求を受け入れたかを示すフラグです。新しく要求を受け入れるときと `is_requested` が 0 のときにデバイスに要求を割り当て、`is_requested` が 1 かつ `rvalid` が 1 のときに結果を返します。

まだアクセス先のデバイスを実装していないため、常に 0 を読み込み、`ready` と `rvalid` は常に 1 にして、書き込みは無視します。

## 10.4 RAM の接続

### 10.4.1 mmio\_controller モジュールに RAM を追加する

`mmio_controller` モジュールに RAM とのインターフェースを実装します。

`Device` 型に RAM を追加して、アドレスに RAM をマップします（リスト 10.17、リスト 10.18）。

▼ リスト 10.17: Device 型に RAM を追加する (`mmio_controller.veryl`)

```

1 enum Device {
2     UNKNOWN,
3     RAM,
4 }
```

▼ リスト 10.18: `get_device` 関数で RAM の範囲を定義する (`mmio_controller.veryl`)

```

1 function get_device (
2     addr: input Addr,
3 ) -> Device {
4     if addr >= MMAP_RAM_BEGIN {
5         return Device::RAM;
6     }
7     return Device::UNKNOWN;
8 }
```

RAM とのインターフェースを追加し、`reset_all_device_masters` 関数に要求をリセットするコードを追加します（リスト 10.19、リスト 10.20）。

▼ リスト 10.19: RAM とのインターフェースを追加する (`mmio_controller.veryl`)

```

1 module mmio_controller (
2     clk      : input  clock      ,
3     rst      : input  reset      ,
4     req_core : modport Membus::slave ,
5     ram_membus: modport Membus::master,
```

```
6 ) {
```

▼リスト 10.20: インターフェースの要求部分をリセットする (mmio\_controller.veryl)

```
1 function reset_all_device_masters () {
2     reset_membus_master(ram_membus);
3 }
```

`ready`、`rvalid` を取得する関数に RAM を登録します (リスト 10.21、リスト 10.22)。

▼リスト 10.21: インターフェースの `ready` を返す (mmio\_controller.veryl)

```
1 function get_device_ready (
2     device: input Device,
3 ) -> logic {
4     case device {
5         Device::RAM: return ram_membus.ready;
6         default : {}
7     }
8     return 1;
9 }
```

▼リスト 10.22: インターフェースの `rvalid` を返す (mmio\_controller.veryl)

```
1 function get_device_rvalid (
2     device: input Device,
3 ) -> logic {
4     case device {
5         Device::RAM: return ram_membus.rvalid;
6         default : {}
7     }
8     return 1;
9 }
```

RAM の `rvalid`、`rdata` を `req_core` に割り当てます (リスト 10.23)。

▼リスト 10.23: RAM へのアクセス結果を `req` に割り当てる (mmio\_controller.veryl)

```
1 function assign_device_slave (
2     device: input Device ,
3     req : modport Membus::response,
4 ) {
5     req.rvalid = 1;
6     req.rdata = 0;
7     case device {
8         Device::RAM: req <> ram_membus;
9         default : {}
10    }
11 }
```

RAM のインターフェースに要求を割り当てます (リスト 10.24)。ここで RAM のベースアドレスを引いたアドレスを割り当てることで、`MMAP_RAM_BEGIN` が `0` になるようにしています。

## ▼ リスト 10.24: RAM に req を割り当ててアクセス要求する (mmio\_controller.veryl)

```

1  function assign_device_master (
2      req: modport Membus::all_input,
3  ) {
4      case get_device(req.addr) {
5          Device::RAM: {
6              ram_membus      < req;
7              ram_membus.addr -= MMAP_RAM_BEGIN;
8          }
9          default: {}
10     }
11 }
```

**10.4.2 RAM と mmio\_controller モジュールを接続する**

top モジュールに mmio\_controller モジュールをインスタンス化し、RAM と mmio\_controller モジュール、mmio\_controller モジュールと core モジュールを接続します。

RAM と mmio\_controller モジュールを接続するインターフェース (`mmio_ram_membus`)、core モジュールと mmio\_controller モジュールを接続するインターフェース (`mmio_membus`) を定義し、`membus` を `ram_membus` に改名します (リスト 10.25、リスト 10.26)。

## ▼ リスト 10.25: インターフェースの定義 / インスタンス名を変更する (top.veryl)

```

1  inst mmio_membus : Membus;
2  inst mmio_ram_membus: Membus;
3  inst ram_membus   : membus_if:<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>;
```

## ▼ リスト 10.26: ポート名を変更する (top.veryl)

```

1  inst ram: memory:<RAM_DATA_WIDTH, RAM_ADDR_WIDTH> #(
2      FILEPATH_IS_ENV: RAM_FILEPATH_IS_ENV,
3      FILEPATH       : RAM_FILEPATH           ,
4  ) (
5      clk           ,
6      rst           ,
7      membust: ram_membus,
8 );
```

core モジュールから RAM へのメモリアクセスを調停する処理を、core モジュールから mmio\_controller モジュールへのアクセスを調停する処理に変更します (リスト 10.27)。

## ▼ リスト 10.27: 調停する対象を mmio\_membus に変更する (top.veryl)

```

1 // mmio_controllerへのメモリアクセスを調停する
2 always_ff {
3     if_reset {
4         memarb_last_i      = 0;
5         memarb_last_iaddr = 0;
6     } else {
7         if mmio_membus.ready {
```

```

8          memarb_last_i    = !d_membus.valid;
9      }
10     }
11 }
12 }
13
14 always_comb {
15     i_membus.ready  = mmio_membus.ready && !d_membus.valid;
16     i_membus.rvalid = mmio_membus.rvalid && memarb_last_i;
17     i_membus.rdata  = if memarb_last_iaddr[2] == 0 ? mmio_membus.rdata[31:0] : mmio_membus.>rdata[63:32];
18
19     d_membus.ready  = mmio_membus.ready;
20     d_membus.rvalid = mmio_membus.rvalid && !memarb_last_i;
21     d_membus.rdata  = mmio_membus.rdata;
22
23     mmio_membus.valid = i_membus.valid | d_membus.valid;
24     if d_membus.valid {
25         mmio_membus.addr  = d_membus.addr;
26         mmio_membus.wen   = d_membus.wen;
27         mmio_membus.wdata = d_membus.wdata;
28         mmio_membus.wmask = d_membus.wmask;
29     } else {
30         mmio_membus.addr  = i_membus.addr;
31         mmio_membus.wen   = 0; // 命令フェッチは常に読み込み
32         mmio_membus.wdata = 'x;
33         mmio_membus.wmask = 'x;
34     }
35 }

```

mmio\_controller をインスタンス化し、RAM と接続します。（リスト 10.28、リスト 10.29）。RAM のアドレスへの変換は調停処理から接続部分に移動しています。

#### ▼リスト 10.28: mmio\_controller モジュールをインスタンス化する (top.veryl)

```

1 inst mmioc: mmio_controller (
2     clk           ,
3     rst           ,
4     req_core : mmio_membus ,
5     ram_membus: mmio_ram_membus,
6 );

```

#### ▼リスト 10.29: mmio\_controller モジュールと RAM を接続する (top.veryl)

```

1 always_comb {
2     // mmio <> RAM
3     ram_membus.valid    = mmio_ram_membus.valid;
4     mmio_ram_membus.ready = ram_membus.ready;
5     ram_membus.addr     = addr_to_memaddr::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>(mmio_ram_membus.>us.addr);
6     ram_membus.wen      = mmio_ram_membus.wen;
7     ram_membus.wdata    = mmio_ram_membus.wdata;
8     ram_membus.wmask    = mmio_ram_membus.wmask;

```

```

9     mmio_ram_membus.rvalid = ram_membus.rvalid;
10    mmio_ram_membus.rdata  = ram_membus.rdata;
11 }
```

### 10.4.3 PC の初期値の変更

PC の初期値を `MMAP_RAM_BEGIN` にすることで、RAM のベースアドレスからプログラムの実行を開始するように変更します。eei パッケージに `INITIAL_PC` を定義し、PC のリセット時に利用します（リスト 10.30、リスト 10.31）。

#### ▼リスト 10.30: PC の初期値を定義する (eei.veryl)

```

1 // pc on reset
2 const INITIAL_PC: Addr = MMAP_RAM_BEGIN;
```

#### ▼リスト 10.31: PC の初期値を設定する (core.veryl)

```

1 always_ff {
2     if_reset {
3         if_pc          = INITIAL_PC;
4         if_is_requested = 0;
5         if_pc_requested = 0;
6         if_fifo_wvalid = 0;
7         if_fifo_wdata   = 0;
8     } else {
```

riscv-tests を実行して RAM にアクセスできているか確認します。今のところ riscv-tests はアドレス `0` から配置されるようにリンクしているため、riscv-tests の `env/p/link.ld` を変更します（リスト 10.32）。

#### ▼リスト 10.32: プログラムの先頭のアドレスを変更する (riscv-tests/env/p/link.ld)

```

1 OUTPUT_ARCH( "riscv" )
2 ENTRY(_start)
3
4 SECTIONS
5 {
6     . = 0x00000000; ←先頭を0x80000000に変更する（戻す）
```

riscv-tests をビルドしなおし、成果物を test ディレクトリに配置してください。ビルドしなおしたので、HEX ファイルを再度生成します（リスト 10.33）。

#### ▼リスト 10.33: HEX ファイルの再生成

```

$ cd test
$ find share/ -type f -not -name "*.dump" -exec riscv64-unknown-elf-objcopy -O binary {} {}.bin \;
>;}
$ find share/ -type f -name "*.bin" -exec sh -c "python3 bin2hex.py 8 {} > {}.hex" \;
```

riscv-tests の終了判定用のアドレスを `MMAP_RAM_BEGIN` 基準のアドレスに変更します（リスト 10.34）。

## ▼リスト 10.34: .tohost のアドレスを変更する (top.veryl)

```

1  #[ifdef(TEST_MODE)]
2  always_ff {
3      let RISCVTESTS_TOHOST_ADDR: Addr = MMAP_RAM_BEGIN + 'h1000 as Addr;
4      if d_membus.valid && d_membus.ready && d_membus.wen == 1 && d_membus.addr == RISCVTESTS>
5      _TOHOST_ADDR && d_membus.wdata[lsb] == 1'b1 {
6          test_success = d_membus.wdata == 1;
7          if d_membus.wdata == 1 {
8              $display("riscv-tests success!");
9          } else {
10             $display("riscv-tests failed!");
11             $error ("wdata : %h", d_membus.wdata);
12         }
13     }
14 }
```

riscv-tests を実行し、RAM にアクセスできてテストに成功することを確認してください。

## 10.5 ROM の実装

### 10.5.1 mmio\_controller モジュールに ROM を追加する

mmio\_controller モジュールに ROM とのインターフェースを実装します。

`Device` 型に ROM を追加して、アドレスに ROM をマップします（リスト 10.35、リスト 10.36）。

## ▼リスト 10.35: Device 型に ROM を変更する (mmio\_controller.veryl)

```

1 enum Device {
2     UNKNOWN,
3     RAM,
4     ROM,
5 }
```

## ▼リスト 10.36: get\_device 関数で ROM の範囲を定義する (mmio\_controller.veryl)

```

1 function get_device (
2     addr: input Addr,
3 ) -> Device {
4     if MMAP_ROM_BEGIN <= addr && addr <= MMAP_ROM_END {
5         return Device::ROM;
6     }
7     if addr >= MMAP_RAM_BEGIN {
8         return Device::RAM;
9     }
10    return Device::UNKNOWN;
11 }
```

ROMとのインターフェースを追加します（リスト10.37、リスト10.38）。reset\_all\_device\_masters関数でインターフェースをリセットします。

▼リスト10.37: ROMとのインターフェースを追加する (mmio\_controller.veryl)

```
1 module mmio_controller (
2     clk      : input  clock      ,
3     rst      : input  reset      ,
4     req_core : modport Membus::slave ,
5     ram_membus: modport Membus::master,
6     rom_membus: modport Membus::master,
7 ) {
```

▼リスト10.38: インターフェースの要求部分をリセットする (mmio\_controller.veryl)

```
1 function reset_all_device_masters () {
2     reset_membus_master(ram_membus);
3     reset_membus_master(rom_membus);
4 }
```

ready、rvalidを取得する関数にROMを登録します（リスト10.39、リスト10.40）。

▼リスト10.39: インターフェースのreadyを返す (mmio\_controller.veryl)

```
1 case device {
2     Device::RAM: return ram_membus.ready;
3     Device::ROM: return rom_membus.ready;
4     default   : {}
5 }
```

▼リスト10.40: インターフェースのrvalidを返す (mmio\_controller.veryl)

```
1 case device {
2     Device::RAM: return ram_membus.rvalid;
3     Device::ROM: return rom_membus.rvalid;
4     default   : {}
5 }
```

ROMのrvalid、rdataをreq\_coreに割り当てます（リスト10.41）。

▼リスト10.41: assign\_device\_slave関数でROMの結果をreqに割り当てる (mmio\_controller.veryl)

```
1 case device {
2     Device::RAM: req <> ram_membus;
3     Device::ROM: req <> rom_membus;
4     default   : {}
5 }
```

ROMのインターフェースに要求を割り当てます（リスト10.42）。RAMと同じようにメモリマップのベースアドレスを引いたアドレスを割り当てます。

## ▼リスト 10.42: get\_device 関数で ROM に req を割り当ててアクセス要求する (mmio\_controller.veryl)

```

1   case get_device(req.addr) {
2     Device::RAM: {
3       ram_membus      <=> req;
4       ram_membus.addr -= MMAP_RAM_BEGIN;
5     }
6     Device::ROM: {
7       rom_membus      <=> req;
8       rom_membus.addr -= MMAP_ROM_BEGIN;
9     }
10    default: {}
11  }

```

## 10.5.2 ROM の初期値のパラメータを作成する

top モジュールに ROM の初期値を指定するパラメータを定義します（リスト 10.43）。

## ▼リスト 10.43: パラメータを定義する (top.veryl)

```

1 module top #(
2   param RAM_FILEPATH_IS_ENV: bit      = 1
3   param RAM_FILEPATH      : string = "RAM_FILE_PATH",
4   param ROM_FILEPATH_IS_ENV: bit      = 1
5   param ROM_FILEPATH      : string = "ROM_FILE_PATH",
6 ) (

```

RAM と同じように、シミュレータ用のプログラムで ROM の HEX ファイルのパスを指定するようにします。1 番目の引数を ROM 用の HEX ファイルのパスに変更し、環境変数 `ROM_FILE_PATH` をその値に設定します（リスト 10.44、リスト 10.45、リスト 10.46、リスト 10.47、リスト 10.48）。

## ▼リスト 10.44: 引数の名称を変える (tb\_verilator.cpp)

```

1 if (argc < 3) {
2   std::cout << "Usage: " << argv[0] << " ROM_FILE_PATH RAM_FILE_PATH [CYCLE]" << std::endl>
>l;
3   return 1;
4 }

```

## ▼リスト 10.45: ROM の HEX ファイルのパスを生成する (tb\_verilator.cpp)

```

1 // メモリの初期値を格納しているファイル名
2 std::string rom_file_path = argv[1];
3 std::string ram_file_path = argv[2];
4 try {
5   // 絶対パスに変換する
6   rom_file_path = fs::absolute(rom_file_path).string();
7   ram_file_path = fs::absolute(ram_file_path).string();
8 } catch (const std::exception& e) {
9   std::cerr << "Invalid memory file path : " << e.what() << std::endl;
10  return 1;
11 }

```

## ▼ リスト 10.46: 引数の数が変わったのでインデックスを変更する (tb\_verilator.cpp)

```

1  unsigned long long cycles = 0;
2  if (argc >= 4) {
3      std::string cycles_string = argv[3];
4      try {
5          cycles = stoull(cycles_string);
6      } catch (const std::exception& e) {
7          std::cerr << "Invalid number: " << argv[3] << std::endl;
8          return 1;
9      }
10 }
```

## ▼ リスト 10.47: 環境変数を変更する (tb\_verilator.cpp)

```

1  const char* original_env_rom = getenv("ROM_FILE_PATH");
2  const char* original_env_ram = getenv("RAM_FILE_PATH");
3  setenv("ROM_FILE_PATH", rom_file_path.c_str(), 1);
4  setenv("RAM_FILE_PATH", ram_file_path.c_str(), 1);
```

## ▼ リスト 10.48: 環境変数を元に戻す (tb\_verilator.cpp)

```

1  if (original_env_rom != nullptr){
2      setenv("ROM_FILE_PATH", original_env_rom, 1);
3  }
4  if (original_env_ram != nullptr){
5      setenv("RAM_FILE_PATH", original_env_ram, 1);
6  }
```

テストを実行するための Python プログラムで ROM の HEX ファイルを指定できるようにします（リスト 10.49、リスト 10.50、リスト 10.51）。デフォルト値はカレントディレクトリの `bootrom.hex` にしておきます。

## ▼ リスト 10.49: 引数--rom を追加する (test/test.py)

```
1 parser.add_argument("--rom", default="bootrom.hex", help="hex file of rom")
```

## ▼ リスト 10.50: シミュレータに ROM の HEX ファイルのパスを渡す (test/test.py)

```

1 def test(romhex, file_name):
2     result_file_path = os.path.join(args.output_dir, file_name.replace(os.sep, "_") + ".txt")
3     cmd = f"{args.sim_path} {romhex} {file_name} 0"
4     success = False
```

## ▼ リスト 10.51: test 関数に ROM の HEX ファイルのパスを渡す (test/test.py)

```

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

### 10.5.3 ROM と mmio\_controller モジュールを接続する

ROM をインスタンス化して mmio\_controller モジュールと接続します。

ROM と mmio\_controller モジュールを接続するインターフェース (`mmio_rom_membus`)、ROM のインターフェース (`rom_membus`) を定義します (リスト 10.52)。

▼ リスト 10.52: ROM のインターフェースの定義 (top.vverly)

```

1 inst mmio_membus : Membus;
2 inst mmio_ram_membus: Membus;
3 inst mmio_rom_membus: Membus;
4 inst ram_membus : membus_if:<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>;
5 inst rom_membus : membus_if:<ROM_DATA_WIDTH, ROM_ADDR_WIDTH>;

```

ROM をインスタンス化します (リスト 10.53)。パラメータには top モジュールのパラメータを割り当てます。

▼ リスト 10.53: ROM をインスタンス化する (top.vverly)

```

1 inst rom: memory:<ROM_DATA_WIDTH, ROM_ADDR_WIDTH> #(
2   FILEPATH_IS_ENV: ROM_FILEPATH_IS_ENV,
3   FILEPATH      : ROM_FILEPATH      ,
4 ) (
5   clk          ,
6   rst          ,
7   membus: rom_membus,
8 );

```

mmio\_controller モジュールに `rom_membus` を接続します (リスト 10.54)。

▼ リスト 10.54: ROM のインターフェースを接続する (top.vverly)

```

1 inst mmioc: mmio_controller (
2   clk          ,
3   rst          ,
4   req_core   : mmio_membus   ,
5   ram_membus: mmio_ram_membus,
6   rom_membus: mmio_rom_membus,
7 );

```

mmio\_controller モジュールと ROM を接続します。アドレスの変換のために `addr_to_memaddr` 関数を使用しています (リスト 10.55)。

▼ リスト 10.55: mmio\_controller モジュールと ROM を接続する (top.vverly)

```

1 always_comb {
2   // mmio <-> ROM
3   rom_membus.valid      = mmio_rom_membus.valid;
4   mmio_rom_membus.ready = rom_membus.ready;
5   rom_membus.addr       = addr_to_memaddr:<ROM_DATA_WIDTH, ROM_ADDR_WIDTH>(mmio_rom_membu
6   us.addr);
7   rom_membus.wen        = 0;
8   rom_membus.wdata      = 0;
9   rom_membus.wmask      = 0;
10  mmio_rom_membus.rvalid = rom_membus.rvalid;
11  mmio_rom_membus.rdata  = rom_membus.rdata;

```

11 }

### 10.5.4 ROM から RAM にジャンプする

PC の初期値を ROM のベースアドレスに変更し、ROM から RAM にジャンプする仕組みを実現します。

一般的に CPU の電源をつけると、CPU は ROM のようなメモリデバイスに入ったソフトウェアから実行を開始します。そのソフトウェアは次に実行するソフトウェアを外部記憶装置から読み取り、RAM にソフトウェアを適切にコピー、配置して実行します。

本章では RAM、ROM ともに `$readmemh` システムタスクで初期化するように実装しているので、RAM のベースアドレスにジャンプするだけのプログラムを ROM に設定します。

ROM に設定するための HEX ファイルを作成します（リスト 10.56）。

#### ▼ リスト 10.56: RAM の開始アドレスにジャンプするプログラム (bootrom.hex)

```
1 00409093080000b7 // 0: lui x1, 0x08000 4: slli x1, x1, 4
2 0000000000008067 // 8: jalr x0, 0(x1) c:
3 0000000000000000 // zero
4
```

PC の初期値を ROM のベースアドレスに変更します（リスト 10.57）。

#### ▼ リスト 10.57: PC の初期値の変更 (eei.veryl)

```
1 const INITIAL_PC: Addr = MMAP_ROM_BEGIN;
```

riscv-tests を実行し、ROM( `0x1000` ) から実行を開始して RAM( `0x80000000` ) にジャンプしてテストを開始していることを確かめてください。

## 10.6 デバッグ用の入出力デバイスの実装

CPU が文字を送信したり受信するためのデバッグ用の入出力デバイスを実装します。今のところ riscv-tests の結果を受け取るためのアドレスを RAM のベースアドレス + `0x1000` にしていますが、この処理もデバイスに実装します。

本章では、デバッグ用の入出力デバイスに次のような 64 ビットレジスタを実装します。

#### 上位 20 ビットが `20'h01010` な値を書き込み

下位 8 ビットを文字として解釈し `$write` システムタスクで出力します。

#### 上位 20 ビットが `20'h01010` ではない LSB が 1 な値を書き込み

今までの riscv-tests の終了判定処理を行います。

#### 読み込み

C++ プログラムの関数を利用して 1 文字入力を受け取ります。有効な入力の場合は上位 20

ビットが `20'h01010`、無効な入力の場合は `0` になります。

### 10.6.1 デバイスのアドレスを設定する

リスト 10.9 でデバイスのアドレスをポートで設定できるようにしたので、`tb_verilator.cpp` で環境変数の値をデバイスのアドレスに設定するようにします。

環境変数 `DBG_ADDR` を読み込み、`DBG_ADDR` ポートに設定します（リスト 10.58）。

#### ▼リスト 10.58: `DBG_ADDR` ポートに環境変数の値を設定する (`tb_verilator.cpp`)

```

1 // デバッグ用の入出力デバイスのアドレスを取得する
2 const char* dbg_addr_c = getenv("DBG_ADDR");
3 const unsigned long long DBG_ADDR = dbg_addr_c == nullptr ? 0 : std::strtoull(dbg_addr_c, nu>llptr, 0);
4
5 // top
6 Vcore_top *dut = new Vcore_top();
7 dut->MMAP_DBG_ADDR = DBG_ADDR;

```

### 10.6.2 `mmio_controller` モジュールにデバイスを追加する

`mmio_controller` モジュールにデバイスを追加します。

`Device` 型に `Device::DEBUG` を追加します（リスト 10.59）。

#### ▼リスト 10.59: `Device` 型にデバッグ用の入出力デバイスを追加する (`mmio_controller.veryl`)

```

1 enum Device {
2     UNKNOWN,
3     RAM,
4     ROM,
5     DEBUG,
6 }

```

ポートにインターフェースとデバイスのアドレスを追加します（リスト 10.60、リスト 10.61）。

#### ▼リスト 10.60: `DBG_ADDR`、インターフェースを追加する (`mmio_controller.veryl`)

```

1 module mmio_controller (
2     clk      : input  clock      ,
3     rst      : input  reset      ,
4     DBG_ADDR : input  Addr       ,
5     req_core : modport Membus::slave ,
6     ram_membus: modport Membus::master,
7     rom_membus: modport Membus::master,
8     dbg_membus: modport Membus::master,
9 ) {

```

#### ▼リスト 10.61: インターフェースの要求部分をリセットする (`mmio_controller.veryl`)

```

1     function reset_all_device_masters () {
2         reset_membus_master(ram_membus);
3         reset_membus_master(rom_membus);

```

```

4     reset_membus_master(dbg_membus);
5 }
```

デバイスの位置を設定します。最初にチェックすることで、他のデバイスとアドレスを被らせたとしてもデバッグ用の入出力デバイスを優先します（リスト 10.62）。

▼リスト 10.62: get\_device 関数でデバイスの範囲を定義する (mmio\_controller.veryl)

```

1 function get_device (
2     addr: input Addr,
3 ) -> Device {
4     if DBG_ADDR <= addr && addr <= DBG_ADDR + 7 {
5         return Device::DEBUG;
6     }
7     if MMAP_ROM_BEGIN <= addr && addr <= MMAP_ROM_END {
8         return Device::ROM;
9     }
10    if addr >= MMAP_RAM_BEGIN {
11        return Device::RAM;
12    }
13    return Device::UNKNOWN;
14 }
```

インターフェースを設定します（リスト 10.63、リスト 10.64、リスト 10.65、リスト 10.66）。この変更は ROM を追加したときとほとんど同じです。

▼リスト 10.63: assign\_device\_master 関数の変更 (mmio\_controller.veryl)

```

1 case get_device(req.addr) {
2     Device::RAM: {
3         ram_membus      <=> req;
4         ram_membus.addr -= MMAP_RAM_BEGIN;
5     }
6     Device::ROM: {
7         rom_membus      <=> req;
8         rom_membus.addr -= MMAP_ROM_BEGIN;
9     }
10    Device::DEBUG: {
11        dbg_membus      <=> req;
12        dbg_membus.addr -= DBG_ADDR;
13    }
14    default: {}
15 }
```

▼リスト 10.64: assign\_device\_slave 関数の変更 (mmio\_controller.veryl)

```

1 case device {
2     Device::RAM  : req <=> ram_membus;
3     Device::ROM  : req <=> rom_membus;
4     Device::DEBUG: req <=> dbg_membus;
5     default      : {}
6 }
```

## ▼リスト 10.65: get\_device\_ready 関数の変更 (mmio\_controller.veryl)

```

1  case device {
2      Device::RAM  : return ram_membus.ready;
3      Device::ROM  : return rom_membus.ready;
4      Device::DEBUG: return dbg_membus.ready;
5      default      : {}
6  }

```

## ▼リスト 10.66: get\_device\_rvalid 関数の変更 (mmio\_controller.veryl)

```

1  case device {
2      Device::RAM  : return ram_membus.rvalid;
3      Device::ROM  : return rom_membus.rvalid;
4      Device::DEBUG: return dbg_membus.rvalid;
5      default      : {}
6  }

```

top モジュールにデバッグ用の入出力デバイスのインターフェース (`dbg_membus`) を定義し、`mmio_controller` モジュールと接続します (リスト 10.67、リスト 10.68)。

## ▼リスト 10.67: インターフェースのインスタンス化 (top.veryl)

```

1  inst ram_membus      : membus_if:<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>;
2  inst rom_membus      : membus_if:<ROM_DATA_WIDTH, ROM_ADDR_WIDTH>;
3  inst dbg_membus      : Membus;

```

## ▼リスト 10.68: インターフェースを接続する (top.veryl)

```

1  inst mmioc: mmio_controller (
2     clk           ,
3     rst           ,
4     DBG_ADDR    : MMAP_DBG_ADDR  ,
5     req_core   : mmio_membus  ,
6     ram_membus: mmio_ram_membus,
7     rom_membus: mmio_rom_membus,
8     dbg_membus  ,
9 );

```

### 10.6.3 出力を実装する

`dbg_membus` を使い、デバッグ出力処理を実装します。既存の `riscv-tests` の終了検知処理を次のように書き換えます (リスト 10.69)。

## ▼リスト 10.69: riscv-tests の終了検知処理をデバッグ用の入出力デバイスに変更する (top.veryl)

```

1 // デバッグ用のIO
2 always_ff {
3     dbg_membus.ready  = 1;
4     dbg_membus.rvalid = dbg_membus.valid;
5     if dbg_membus.valid {
6         if dbg_membus.wen {
7             if dbg_membus.wdata[MEMBUS_DATA_WIDTH - 1:20] == 20'h01010 {

```

```

8          $write("%c", dbg_membus.wdata[7:0]);
9      } else if dbg_membus.wdata[lsb] == 1'b1 {
10         #[ifdef(TEST_MODE)]
11         {
12             test_success = dbg_membus.wdata == 1;
13         }
14         if dbg_membus.wdata == 1 {
15             $display("test success!");
16         } else {
17             $display("test failed!");
18             $error ("wdata : %h", dbg_membus.wdata);
19         }
20         $finish();
21     }
22 }
23 }
24 }
```

常に要求を受け付け、書き込みの時は書き込むデータ（`wdata`）を確認します。`wdata` の上位 20 ビットが `20'h01010` なら下位 8 ビットを出力し、LSB が `1` ならテストの成功判定をして `$finish` システムタスクを呼び出します。

#### 10.6.4 出力をテストする

実装した出力デバイスで文字を出力できることを確認します。

デバッグ用に `$display` システムタスクで表示している情報が邪魔になるので、デバッグ情報の表示を環境変数 `PRINT_DEBUG` で制御できるようにします（リスト 10.70）。

##### ▼ リスト 10.70: デバッグ出力を define で囲う (core.veryl)

```

1 ////////////////////////////////////////////////////////////////// DEBUG //////////////////////////////////////////////////////////////////
2 #[ifdef(PRINT_DEBUG)]
3 {
4     var clock_count: u64;
5
6     always_ff {
7         if_reset {
8             clock_count = 1;
9         } else {
10            clock_count = clock_count + 1;
11
12            $display("");
13            $display("# %d", clock_count);
```

`test/debug_output.c` を作成し、次のように記述します（リスト 10.71）。これは `Hello,world!` と出力するプログラムです。

## ▼リスト 10.71: Hello,world!を出力するプログラム (test/debug\_output.c)

```

1 #define DEBUG_REG ((volatile unsigned long long*)0x40000000)
2
3 void main(void) {
4     int strlen = 13;
5     unsigned char str[13];
6
7     str[0] = 'H';
8     str[1] = 'e';
9     str[2] = 'l';
10    str[3] = 'l';
11    str[4] = 'o';
12    str[5] = ',';
13    str[6] = 'w';
14    str[7] = 'o';
15    str[8] = 'r';
16    str[9] = 'l';
17    str[10] = 'd';
18    str[11] = '!';
19    str[12] = '\n';
20
21    for (int i = 0; i < strlen; i++) {
22        unsigned long long c = str[i];
23        *DEBUG_REG = c | (0x01010ULL << 44);
24    }
25    *DEBUG_REG = 1;
26 }
```

`DEBUG_REG` は出力デバイスのアドレスです。ここに `0x01010` を 44 ビット左シフトした値と文字を OR 演算した値を書き込むことで文字を出力します。最後に `1` を書き込み、テストを終了しています。

`main` 関数をそのままコンパイルして RAM に配置すると、スタックポインタ (stack pointer, `sp`) の値が適切に設定されていないのでうまく動きません。スタックポインタとは、プログラムが一時的に利用する値を格納しておくためのメモリ (スタック) のアドレスへのポインタのことです。RISC-V の規約では `sp(x2)` レジスタをスタックポインタとして利用することが定められています。

そのため、レジスタの値を適切な値にリセットして `main` 関数を呼び出す別のプログラムが必要です。`test/entry.S` を作成し、次のように記述します (リスト 10.72)。

## ▼リスト 10.72: test/entry.S

```

1 .global _start
2 .section .text.init
3 _start:
4     add x1, x0, x0
5     la x2, _stack_bottom
6     add x3, x0, x0
7     add x4, x0, x0
8     add x5, x0, x0
9     add x6, x0, x0
10    add x7, x0, x0
```

```

11 add x8, x0, x0
12 add x9, x0, x0
13 add x10, x0, x0
14 add x11, x0, x0
15 add x12, x0, x0
16 add x13, x0, x0
17 add x14, x0, x0
18 add x15, x0, x0
19 add x16, x0, x0
20 add x17, x0, x0
21 add x18, x0, x0
22 add x19, x0, x0
23 add x20, x0, x0
24 add x21, x0, x0
25 add x22, x0, x0
26 add x23, x0, x0
27 add x24, x0, x0
28 add x25, x0, x0
29 add x26, x0, x0
30 add x27, x0, x0
31 add x28, x0, x0
32 add x29, x0, x0
33 add x30, x0, x0
34 add x31, x0, x0
35 call main

```

このアセンブリは `sp(x2)` レジスタを `_stack_bottom` のアドレスに設定し、他のレジスタを `0` でリセットしたあとに `main` にジャンプします。

`_stack_bottom` は、リンクの設定ファイルに記述します。`test/link.ld` を作成し、次のように記述します（リスト 10.73）。

#### ▼リスト 10.73: test/link.ld

```

1 OUTPUT_ARCH( "riscv" )
2 ENTRY(_start)
3
4 SECTIONS
5 {
6     . = 0x80000000;
7     .text.init : { *(.text.init) }
8     .text : { *(.text*) }
9     .data : { *(.data*) }
10    .bss : {*(.bss*)}
11    .stack : {
12        . = ALIGN(0x10);
13        _stack_top = .;
14        . += 4K;
15        _stack_bottom = .;
16    }
17    _end = .;
18 }

```

`_stack_bottom` と `_stack_top` の間は 4KB あるので、スタックのサイズは 4KB になります。`_start` を `.text.init` に配置し（リスト 10.72）、`SECTIONS` の先頭に `.text.init` を配置しているため、アドレス `0x80000000` に `_start` が配置されます。

これらのファイルを利用し、テストプログラムをコンパイルします（リスト 10.74）。gcc の `-march` フラグでは C 拡張を抜いた ISA を指定しています。このフラグを記述しないと、まだ実装していない命令が含まれた ELF ファイルにコンパイルされてしまいます。

#### ▼ リスト 10.74: テストプログラムをコンパイル、HEX ファイルに変換する

```
$ cd test
$ riscv64-unknown-elf-gcc -nostartfiles -nostdlib -mcpu=medany -T link.ld -march=rv64imad debug
$ riscv64-unknown-elf-objcopy a.out -O binary test.bin
$ python3 bin2hex.py 8 test.bin > test.bin.hex ← HEXファイルに変換する
```

シミュレータをビルドし、テストプログラムを実行します（リスト 10.75）。

#### ▼ リスト 10.75: テストプログラムを実行する

```
$ make build sim
$ DBG_ADDR=0x40000000 ./obj_dir/sim bootrom.hex test/test.bin.hex
Hello,world!
- ~/core/src/top.sv:62: Verilog $finish
```

`Hello,world!` と出力されたあと、プログラムが終了しました。

### 10.6.5 riscv-tests に対応する

riscv-tests を実行するとき、終了判定用のレジスタの位置を `DBG_ADDR` に設定するようにします。

`test/test.py` を、ELF ファイルを探して自動で `DBG_ADDR` を設定してテストを実行するプログラムに変更します。

elftools<sup>\*2</sup>を使用し、ELF ファイルの判定、セクションのアドレスを取得する関数を定義します（リスト 10.76、リスト 10.77）。

#### ▼ リスト 10.76: elftools の import (test/test.py)

```
1 from elftools.elf.elffile import ELFFile
```

#### ▼ リスト 10.77: ELF の判定、セクションのアドレスを取得する関数の定義 (test/test.py)

```
1 def is_elf(filepath):
2     try:
3         with open(filepath, 'rb') as f:
4             magic_number = f.read(4)
5             return magic_number == b'\x7fELF'
6     except:
7         return False
8
```

<sup>\*2</sup> pip でインストールできます

```

9 def get_section_address(filepath, section_name):
10    try:
11        with open(filepath, 'rb') as f:
12            elffile = ELFfile(f)
13            for section in elffile.iter_sections():
14                if section.name == section_name:
15                    return section.header['sh_addr']
16    return 0
17 except:
18    return 0

```

デバッグ用の入出力デバイスのセクション名を指定する引数を作成します（リスト 10.78）。また、テストするファイルの拡張子を指定していた引数を、ELF ファイルに付加することで HEX ファイルのパスを得るために引数に変更します。

#### ▼リスト 10.78: オプションを追加する (test/test.py)

```

1 parser.add_argument("-e", "--extension", default=".bin.hex", help="hex file extension")
2 parser.add_argument("-d", "--debug_label", default=".tohost", help="debug device label")

```

`dir_walk` 関数を、ELF ファイルを探す関数に変更します（リスト 10.79）。

#### ▼リスト 10.79: `dir_walk` 関数で ELF ファイルを探す (test/test.py)

```

1 if entry.is_file():
2     if not is_elf(entry.path):
3         continue
4     if len(args.files) == 0:
5         yield entry.path

```

シミュレータの実行で `DBG_ADDR` を指定するようにします（リスト 10.80、リスト 10.81）。

#### ▼リスト 10.80: `DBG_ADDR` をシミュレータに渡す (test/test.py)

```

1 def test(dbg_addr, romhex, file_name):
2     result_file_path = os.path.join(args.output_dir, file_name.replace(os.sep, "_") + ".txt")
3     env = f"DBG_ADDR={dbg_addr} "
4     cmd = f"{args.sim_path} {romhex} {file_name} 0"
5     success = False
6     with open(result_file_path, "w") as f:
7         no = f.fileno()
8         p = subprocess.Popen(" ".join([env, "exec", cmd]), shell=True, stdout=no, stderr=no)

```

#### ▼リスト 10.81: `DBG_ADDR` を `test` 関数に渡す (test/test.py)

```

1 for elfpath in dir_walk(args.dir):
2     hexpath = elfpath + args.extension
3     if not os.path.exists(hexpath):
4         print("SKIP :", elfpath)
5         continue
6     dbg_addr = get_section_address(elfpath, args.debug_label)
7     f, s = test(dbg_addr, os.path.abspath(args.rom), os.path.abspath(hexpath))
8     res_strs.append(("PASS" if s else "FAIL") + " : " + f)

```

9 res\_statuses.append(s)

`VERILATOR_FLAGS="-DTEST_MODE"` をつけてシミュレータをビルドし、`riscv-tests` が正常終了することを確かめてください。

## 10.6.6 入力を実装する

`dbg_membus` を使い、デバッグ入力処理を実装します。

まず、`src/tb_verilator.cpp` に、標準入力から 1 文字取得する関数を定義します（リスト 10.82）。入力がない場合は `0`、ある場合は上位 20 ビットを `0x01010` にした値を返します。

▼ リスト 10.82: 標準入力を 1 文字取得する関数の定義 (`src/tb_verilator.cpp`)

```
1 extern "C" const unsigned long long get_input_dpic() {
2     unsigned char c = 0;
3     ssize_t bytes_read = read(STDIN_FILENO, &c, 1);
4
5     if (bytes_read == 1) {
6         return static_cast<unsigned long long>(c) | (0x01010ULL << 44);
7     }
8     return 0;
9 }
```

ここで、`read` 関数の呼び出しでシミュレータを止めず（`O_NONBLOCK`）、シェルが入力をバッファリングしなくする（`~ICANON`）ために設定を変えるコードを挿入します。また、シェルが文字列をローカルエコー（入力した文字列を表示）しないようにします（`~ECHO`）（リスト 10.83、リスト 10.84、リスト 10.85）。

▼ リスト 10.83: include を追加する (`src/tb_verilator.cpp`)

```
1 #include <fcntl.h>
2 #include <termios.h>
3 #include <signal.h>
```

▼ リスト 10.84: 設定を変更、復元する関数の定義 (`src/tb_verilator.cpp`)

```
1 struct termios old_setting;
2
3 void restore_termios_setting(void) {
4     tcsetattr(STDIN_FILENO, TCSANOW, &old_setting);
5 }
6
7 void sighandler(int signum) {
8     restore_termios_setting();
9     exit(signum);
10}
11
12 void set_nonblocking(void) {
13     struct termios new_setting;
```

```

15 if (tcgetattr(STDIN_FILENO, &old_setting) == -1) {
16     perror("tcgetattr");
17     return;
18 }
19 new_setting = old_setting;
20 new_setting.c_lflag &= ~(ICANON | ECHO);
21 if (tcsetattr(STDIN_FILENO, TCSANOW, &new_setting) == -1) {
22     perror("tcsetattr");
23     return;
24 }
25 signal(SIGINT, sighandler);
26 signal(SIGTERM, sighandler);
27 signal(SIGQUIT, sighandler);
28 atexit(restore_termios_setting);
29
30 int flags = fcntl(STDIN_FILENO, F_GETFL, 0);
31 if (flags == -1) {
32     perror("fcntl(F_GETFL)");
33     return;
34 }
35 if (fcntl(STDIN_FILENO, F_SETFL, flags | O_NONBLOCK) == -1) {
36     perror("fcntl(F_SETFL)");
37     return;
38 }
39 }
```

▼リスト 10.85: 設定を変える関数を main 関数から呼び出す (src/tb\_verilator.cpp)

```

1 int main(int argc, char** argv) {
2     Verilated::commandArgs(argc, argv);
3
4     if (argc < 3) {
5         std::cout << "Usage: " << argv[0] << " ROM_FILE_PATH RAM_FILE_PATH [CYCLE]" << std::endl;
6         return 1;
7     }
8
9 #ifdef ENABLE_DEBUG_INPUT
10     set_nonblocking();
11 #endif
```

src/util.veryl に get\_input\_dplic 関数を呼び出す関数を実装します ( リスト 10.86 )。

▼リスト 10.86: get\_input 関数を定義する (src/util.veryl)

```

1 embed (inline) sv{{{
2     package svutil;
3     ...
4     import "DPI-C" context function longint get_input_dplic();
5     function longint get_input();
6         return get_input_dplic();
7     endfunction
8 }}}
endpackage
```

```

9 }}}
10
11 package util {
12 ...
13     function get_input () -> u64 {
14         return $sv::svutil::get_input();
15     }
16 }
```

デバッグ用の入出力デバイスのロードで `util::get_input` の結果を返すようにします（リスト 10.87）。このコードは合成できないので、有効化オプション `ENABLE_DEBUG_INPUT` をつけます。

#### ▼リスト 10.87: 読み込みで `get_input` 関数を呼び出す (src/top.veryl)

```

1 always_ff {
2     dbg_membus.ready = 1;
3     dbg_membus.rvalid = dbg_membus.valid;
4     if dbg_membus.valid {
5         if dbg_membus.wen {
6             ...
7         } else {
8             #[ifdef(ENABLE_DEBUG_INPUT)]
9             {
10                 dbg_membus.rdata = util::get_input();
11             }
12         }
13     }
14 }
```

## 10.6.7 入力をテストする

実装した入出力デバイスで文字を入出力できることを確認します。

`test/debug_input.c` を作成し、次のように記述します（リスト 10.88）。これは入力された文字に 1 を足した値を出力するプログラムです。

#### ▼リスト 10.88: test/debug\_input.c

```

1 #define DEBUG_REG ((volatile unsigned long long*)0x40000000)
2
3 void main(void) {
4     while (1) {
5         unsigned long long c = *DEBUG_REG;
6         if (c & (0x01010ULL << 44) == 0) {
7             continue;
8         }
9         c = c & 255;
10        *DEBUG_REG = (c + 1) | (0x01010ULL << 44);
11    }
12 }
```

プログラムをコンパイルしてシミュレータを実行し、入力した文字が 1 文字ずれて表示されるこ

とを確認してください（リスト 10.89）。

▼ リスト 10.89: テストプログラムを実行する

```
$ make build sim VERILATOR_FLAGS="-DENABLE_DEBUG_INPUT" ← 入力を有効にしてシミュレータをビルド
$ ./obj_dir/sim bootrom.hex test/test.bin.hex ← (事前にHEXファイルを作成しておく)
bcd← abcと入力して改行
efg← defと入力する
```

# 第 11 章

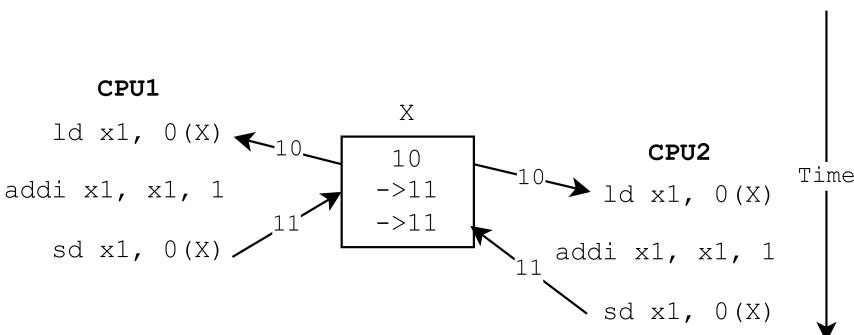
## A 拡張の実装

本章では、メモリの不可分操作を実現する A 拡張を実装します。A 拡張には Load-Reserved、Store-Conditional を実現する Zalrsc 拡張(表 11.2)、ロードした値を加工し、その結果をメモリにストアする操作を单一の命令で実装する Zaamo 拡張(表 11.1)が含まれています。A 拡張の命令を利用すると、同じメモリ空間で複数のソフトウェアを並列、並行して実行するとき、ソフトウェア間で同期をとりながら実行できます。

### 11.1 アトミック操作

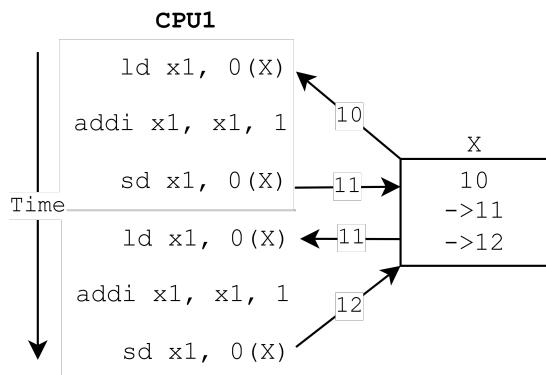
#### 11.1.1 アトミック操作とは何か？

アトミック操作(Atomic operation、不可分操作)とは、他のシステムからその操作を観測するとき、1つの操作として観測される操作のことです。つまり、他のシステムは、アトミック操作を行う前、アトミック操作を行った後の状態しか観測できません。



▲図 11.1: 図 11.2 のプログラムを 2 つに分割して 2 つの CPU で実行する (X は 11 になる)

アトミック操作は実行、観測される順序が重要なアプリケーションで利用します。例えば、アドレス X の値をロードして 1 を足した値を書き戻すプログラムを、2 つのコアで同時に実行するとし



▲図 11.2: 1 つの CPU でメモリ上の値を 2 回インクリメントする (X は 12 になる)

ます(図 11.1)。このとき命令の実行順序によっては、最終的な値が 1 つのコアで 2 回プログラムを実行した場合と異なってしまいます(図 11.2)。この状態を避けるためにはロード、加算、ストアをアトミックに行う必要があります。このアトミック操作の実現方法として、A 拡張は AMOADD 命令、LR 命令と SC 命令を提供します。

### 11.1.2 Zaamo 拡張

Zaamo 拡張は、値をロードして、演算した値をストアする操作を 1 つの命令で行う命令を定義しています。AMOADD 命令はロード、加算、ストアを行う单一の命令です。Zaamo 拡張は他にも簡単な操作を行う命令も提供しています。

### 11.1.3 Zalrsc 拡張

Zalrsc 拡張は、LR 命令と SC 命令を定義しています。LR、SC 命令は、それぞれ Load-Reserved、Store-Conditional 操作を実現する命令です。それぞれ次のように動作します。

#### LR 命令

指定されたアドレスのデータを読み込み、指定されたアドレスを予約セット(Reservation set)に登録します。ロードしたデータをレジスタにライトバックします。

#### SC 命令

指定されたアドレスが予約セットに存在する場合、指定されたアドレスにデータを書き込みます(ストア成功)。予約セットにアドレスが存在しない場合は書き込みません(ストア失敗)。ストアに成功したら 0、失敗したら 0 以外の値をレジスタにライトバックします。命令の実行後に必ず予約セットを空にします。

LR、SC 命令を使うことで、アトミックなロード、加算、ストアを次のように記述できます(リスト 11.1)。

▼表 11.1: Zaamo 拡張の命令

命令	動作 (読み込んだ値をレジスタにライトバックする)
AMOSWAP.W/D	メモリから 32/64 ビット読み込み、rs2 の値を書き込む
AMOADD.W/D	メモリから 32/64 ビット (符号付き) 読み込み rs2(符号付き) の値を足して書き込む
AMOAND.W/D	メモリから 32/64 ビット読み込み rs2 の値を AND 演算して書き込む
AMOOR.W/D	メモリから 32/64 ビット読み込み rs2 の値を OR 演算して書き込む
AMOXOR.W/D	メモリから 32/64 ビット読み込み rs2 の値を XOR 演算して書き込む
AMOMIN.W/D	メモリから 32/64 ビット (符号付き) 読み込み rs2(符号付き) の値と比べて小さい値を書き込む
AMOMAX.W/D	メモリから 32/64 ビット (符号付き) 読み込み rs2(符号付き) の値と比べて大きい値を書き込む
AMOMINU.W/D	メモリから 32/64 ビット (符号無し) 読み込み rs2(符号無し) の値と比べて小さい値を書き込む
AMOMAXU.W/D	メモリから 32/64 ビット (符号無し) 読み込み rs2(符号無し) の値と比べて大きい値を書き込む

▼リスト 11.1: LR、SC 命令によるアトミックな加算

```

1 atomic_add:
2   LR.W x2, (x3) ←アドレスx3の値をx2にロード
3   ADDI x2, x2, 1 ←x2に1を足す
4   SC.W x4, x2, (x3) ←ストアを試行し、結果をx4に格納
5   BNEZ x4, atomic_add ←SC命令が失敗していたらやり直す

```

例えば同時に 2 つのコアがリスト 11.1 を実行するとき、同期をとれていない書き込みは SC 命令で失敗します。失敗したら LR 命令からやり直すことで、1 つのコアで 2 回実行した場合と同一の結果 (1 を 2 回加算) になります。

予約セットのサイズは実装によって異なります。

▼表 11.2: Zalrsc 拡張の命令

命令	動作
LR.W/D	メモリから 32/64 ビット読み込み、予約セットにアドレスを登録する 読み込んだ値をレジスタにライトバックする
SC.W/D	予約セットに rs1 の値が登録されている場合、メモリに rs2 の値を書き込み 0 をレジスタにライトバックする。予約セットにアドレスが登録されていない場合 メモリに書き込みず、0 以外の値をレジスタにライトバックする。 命令の実行後に予約セットを空にする

### 11.1.4 命令の順序

A 拡張の命令のビット列は、それぞれ 1 ビットの aq、rl ビットを含んでいます。このビットは、

他のコアやハードウェアスレッドからメモリ操作を観測したときにメモリ操作がどのような順序で観測されるかを制御するものです。

A 拡張の命令を A とするとき、それぞれのビットの状態に応じて、A によるメモリ操作は次のように観測されます。

### **aq=0、rl=0**

A の前後でメモリ操作の順序は保証されません。

### **aq=1、rl=0**

A の後ろにあるメモリを操作する命令は、A のメモリ操作の後に観測されることが保証されます。

### **aq=0、rl=1**

A のメモリ操作は、A の前にあるメモリを操作する命令が観測できるようになった後に観測されることが保証されます。

### **aq=1、rl=1**

A のメモリ操作は、A の前にあるメモリを操作する命令よりも後、A の後ろにあるメモリを操作する命令よりも前に観測されることが保証されます。

今のところ、CPU はメモリ操作を 1 命令ずつ直列に実行するため、常に aq が 1、rl が 1 であるように動作します。そのため、本章では aq、rl ビットを考慮しないで実装を行います<sup>\*1</sup>。

## 11.2 命令のデコード

A 拡張の命令はすべて R 形式で、opcode は OP-AMO( 7'b0101111 )です。それぞれの命令は funct5( リスト 11.3 )と funct3(W は 2 、 D は 3 )で区別できます。

eei パッケージに OP-AMO の定数を定義します(リスト 11.2)。

### ▼ リスト 11.2: OP-AMO の定義 (eei.vervyl)

```
1 const OP_AMO      : logic<7> = 7'b0101111;
```

A 拡張の命令を区別するための列挙型 AMOOp を定義します(リスト 11.3)。それぞれ命令の funct5 と対応しています。

### ▼ リスト 11.3: AMOOp 型の定義 (eei.vervyl)

```
1 enum AMOOp: logic<5> {
2     LR = 5'b00010,
3     SC = 5'b00011,
4     SWAP = 5'b00001,
5     ADD = 5'b00000,
6     XOR = 5'b00100,
```

\*1 メモリ操作の並び替えによる高速化は応用編で検討します。

```

7     AND = 5'b01100,
8     OR = 5'b01000,
9     MIN = 5'b10000,
10    MAX = 5'b10100,
11    MINU = 5'b11000,
12    MAXU = 5'b11100,
13 }

```

### 11.2.1 is\_amo フラグを実装する

`InstCtrl` 構造体に、A 拡張の命令であることを示す `is_amo` フラグを追加します（リスト 11.4）。

#### ▼ リスト 11.4: InstCtrl に is\_amo を定義する (corectrl.veryl)

```

1 struct InstCtrl {
2     itype      : InstType   , // 命令の形式
3     rwb_en    : logic      , // レジスタに書き込むかどうか
4     is_lui    : logic      , // LUI命令である
5     is_aluop  : logic      , // ALUを利用する命令である
6     is_muldiv: logic      , // M拡張の命令である
7     is_op32   : logic      , // OP-32またはOP-IMM-32である
8     is_jump   : logic      , // ジャンプ命令である
9     is_load   : logic      , // ロード命令である
10    is_csr    : logic      , // CSR命令である
11    is_amo    : logic      , // AMO instruction
12    funct3   : logic <3>, // 命令のfunct3フィールド
13    funct7   : logic <7>, // 命令のfunct7フィールド
14 }

```

命令がメモリにアクセスするかを判定する `inst_is_memop` 関数を、`is_amo` フラグを利用するように変更します（リスト 11.5）。

#### ▼ リスト 11.5: A 拡張の命令がメモリにアクセスする命令と判定する (corectrl.veryl)

```

1 function inst_is_memop (
2     ctrl: input InstCtrl,
3 ) -> logic {
4     return ctrl.itype == InstType::S || ctrl.is_load || ctrl.is_amo;
5 }

```

`inst_decoder` モジュールの `InstCtrl` を生成している部分を変更します。opcode が `OP-AMO` のとき、`is_amo` を `T` に設定します（リスト 11.6）。その他の opcode の `is_amo` は `F` に設定してください。

#### ▼ リスト 11.6: is\_amo フラグを追加する (inst\_decoder.veryl)

```

1         OP_SYSTEM: {
2             InstType::I, T, F, F, F, F, F, F, T, F
3         },
4         OP_AMO: {
5             InstType::R, T, F, F, F, F, F, F, F, T

```

```

6      },
7      default: {
8          InstType::X, F, F, F, F, F, F, F, F, F
9      },

```

また、A 拡張の命令が有効な命令として判断されるようにします（リスト 11.7）。

▼ リスト 11.7: A 拡張の命令のとき、valid フラグを立てる (inst\_decoder.veryl)

```

1     OP_MISC_MEM: T, // FENCE
2     OP_AMO    : f3 == 3'b010 || f3 == 3'b011, // AMO
3     default   : F,

```

## 11.2.2 アドレスを変更する

A 拡張でアクセスするメモリのアドレスは rs1 で指定されたレジスタの値です。これは基本整数命令セットのロードストア命令のアドレス指定方法（rs1 と即値を足し合わせる）とは異なるため、memunit モジュールの `addr` ポートに割り当てる値を `is_amo` フラグによって切り替えます（リスト 11.8）。

▼ リスト 11.8: メモリアドレスを rs1 レジスタの値にする (core.veryl)

```

1 var memu_rdata: UIntX;
2 var memu_stall: logic;
3 let memu_addr : Addr = if mems_ctrl.is_amo ? memq_rdata.rs1_data : memq_rdata.alu_result;
4
5 inst memu: memunit (
6     clk           ,
7     rst           ,
8     valid : mems_valid && !mems_expt.valid,
9     is_new: mems_is_new ,
10    ctrl : mems_ctrl ,
11    addr : memu_addr ,
12    rs2 : memq_rdata.rs2_data ,
13    rdata : memu_rdata ,
14    stall : memu_stall ,
15    membus: d_membus ,
16 );

```

A 拡張の命令のメモリアドレスが、操作するデータの幅に整列されていないとき、Store/AMO address misaligned 例外が発生します。この例外はストア命令の場合の例外と同じです。

EX ステージの例外判定でアドレスを使っている部分を変更します（リスト 11.9）。cause と tval の割り当てがストア命令の場合と同じになっていることを確認してください。

▼ リスト 11.9: 例外を判定するアドレスを変更する (core.veryl)

```

1 let memaddr           : Addr = if exs_ctrl.is_amo ? exs_rs1_data : exs_a>
>lu_result;
2 let loadstore_address_misaligned : logic = inst_is_memop(exs_ctrl) && case exs_ctrl.fun>
>ct3[1:0] {

```

```

3      2'b00 : 0, // B
4      2'b01 : memaddr[0] != 1'b0, // H
5      2'b10 : memaddr[1:0] != 2'b0, // W
6      2'b11 : memaddr[2:0] != 3'b0, // D
7      default: 0,
8  };

```

### 11.2.3 ライトバックする条件を変更する

A 拡張の命令を実行するとき、ロードした値をレジスタにライトバックするように変更します（リスト 11.10）。

#### ▼リスト 11.10: メモリからロードした値をライトバックする (core.veryl)

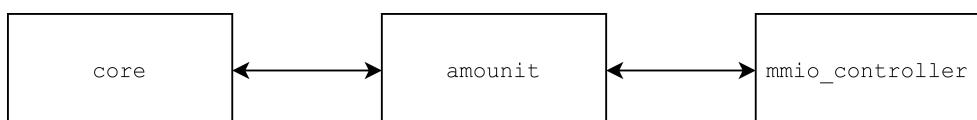
```

1  let wbs_wb_data: UIntX = if wbs_ctrl.is_lui ?
2    wbs_imm
3    : if wbs_ctrl.is_jump ?
4      wbs_pc + 4
5    : if wbs_ctrl.is_load || wbs_ctrl.is_amo ?
6      wbq_rdata.mem_rdata
7    : if wbs_ctrl.is_csr ?

```

## 11.3 amounit モジュールの作成

A 拡張は他のコア、ハードウェアスレッドと同期してメモリ操作を行うためのものであるため、A 拡張の操作は core モジュールの外、メモリよりも前で行います。本書では、core モジュールと mmio\_controller モジュールの間に、A 拡張の命令を処理する amounit モジュールを実装します（図 11.3）。



▲図 11.3: amounit モジュールと他のモジュールの接続

### 11.3.1 インターフェースを作成する

amounit モジュールに A 拡張の操作を指示するために、`is_amo` フラグ、`aq` ビット、`rl` ビット、`AM0Op` 型を `membus_if` インターフェースに追加で定義したインターフェースを作成します。

`src/core_data_if.veryl` を作成し、次のように記述します（リスト 11.11）。

## ▼リスト 11.11: core\_data\_if.veryl

```

1 import eei::*;

2
3 interface core_data_if {
4     var valid : logic           ;
5     var ready : logic           ;
6     var addr  : logic<XLEN>    ;
7     var wen   : logic           ;
8     var wdata  : logic<MEMBUS_DATA_WIDTH> ;
9     var wmask : logic<MEMBUS_DATA_WIDTH / 8>;
10    var rvalid: logic           ;
11    var rdata : logic<MEMBUS_DATA_WIDTH>    ;
12
13    var is_amo: logic   ;
14    var aq     : logic   ;
15    var rl     : logic   ;
16    var amoop : AMOOp   ;
17    var funct3: logic<3>;
18
19    modport master {
20        valid : output,
21        ready : input ,
22        addr  : output,
23        wen   : output,
24        wdata  : output,
25        wmask : output,
26        rvalid: input ,
27        rdata : input ,
28        is_amo: output,
29        aq   : output,
30        rl   : output,
31        amoop : output,
32        funct3: output,
33    }
34
35    modport slave {
36        ..converse(master)
37    }
38
39    modport all_input {
40        ..input
41    }
42}

```

### 11.3.2 amounit モジュールの作成

メモリ操作を core モジュールからそのまま mmio\_controller モジュールに受け渡しするだけのモジュールを作成します。 `src/amounit.veryl` を作成し、次のように記述します（リスト 11.12）。

## ▼リスト 11.12: amounit.veryl

```

1 import eei::*;

2 module amounit (
3     clk : input    clock      ,
4     rst : input    reset      ,
5     slave : modport core_data_if::slave,
6     master: modport Membus::master      ,
7 ) {
8 }

9     enum State {
10         Init,
11         WaitReady,
12         WaitValid,
13     }
14 }

15     var state      : State;
16     inst slave_saved: core_data_if;
17

18     // masterをリセットする
19     function reset_master () {
20         master.valid = 0;
21         master.addr  = 0;
22         master.wen   = 0;
23         master.wdata  = 0;
24         master.wmask  = 0;
25     }
26

27     // masterに要求を割り当てる
28     function assign_master (
29         addr : input Addr      ,
30         wen  : input logic     ,
31         wdata: input UIntX      ,
32         wmask: input logic<$size(UIntX) / 8>,
33     ) {
34         master.valid = 1;
35         master.addr  = addr;
36         master.wen   = wen;
37         master.wdata  = wdata;
38         master.wmask  = wmask;
39     }
40

41     // 新しく要求を受け入れる
42     function accept_request_comb () {
43         if slave.ready && slave.valid {
44             assign_master(slave.addr, slave.wen, slave.wdata, slave.wmask);
45         }
46     }
47 }

48     // slaveに結果を割り当てる
49     always_comb {
50         slave.ready  = 0;
51         slave.rvalid = 0;
52     }

```

```
53     slave.rdata = 0;
54
55     case state {
56         State::Init: {
57             slave.ready = 1;
58         }
59         State::WaitValid: {
60             slave.ready = master.rvalid;
61             slave.rvalid = master.rvalid;
62             slave.rdata = master.rdata;
63         }
64         default: {}
65     }
66 }
67
68 // masterに要求を割り当てる
69 always_comb {
70     reset_master();
71     case state {
72         State::Init : accept_request_comb();
73         State::WaitReady: {
74             assign_master(slave_saved.addr, slave_saved.wen, slave_saved.wdata, slave_saved>.wmask);
75         }
76         State::WaitValid: accept_request_comb();
77         default : {}
78     }
79 }
80
81 // 新しく要求を受け入れる
82 function accept_request_ff () {
83     slave_saved.valid = slave.ready && slave.valid;
84     if slave.ready && slave.valid {
85         slave_saved.addr = slave.addr;
86         slave_saved.wen = slave.wen;
87         slave_saved.wdata = slave.wdata;
88         slave_saved.wmask = slave.wmask;
89         slave_saved.is_amo = slave.is_amo;
90         slave_saved.amoop = slave.amoop;
91         slave_saved.aq = slave.aq;
92         slave_saved.rl = slave.rl;
93         slave_saved.funct3 = slave.funct3;
94         state = if master.ready ? State::WaitValid : State::WaitReady;
95     } else {
96         state = State::Init;
97     }
98 }
99
100 function on_clock () {
101     case state {
102         State::Init : accept_request_ff();
103         State::WaitReady: if master.ready {
104             state = State::WaitValid;
```

```

105      }
106      State::WaitValid: if master.rvalid {
107          accept_request_ff();
108      }
109      default: {}
110  }
111 }
112
113 function on_reset () {
114     state          = State::Init;
115     slave_saved.addr = 0;
116     slave_saved.wen  = 0;
117     slave_saved.wdata = 0;
118     slave_saved.wmask = 0;
119     slave_saved.is_amo = 0;
120     slave_saved.amoop = 0 as AMOOp;
121     slave_saved.aq   = 0;
122     slave_saved.rl   = 0;
123     slave_saved.funct3 = 0;
124 }
125
126 always_ff {
127     if_reset {
128         on_reset();
129     } else {
130         on_clock();
131     }
132 }
133 }
```

amounit モジュールは `State::Init`、( `State::WaitReady`、) `State::WaitValid` の順に状態を移動し、通常のロードストア命令を処理します。

core モジュールのロードストア用のインターフェースを `membus_if` から `core_data_if` に変更します (リスト 11.13、リスト 11.14、リスト 11.15)。

#### ▼ リスト 11.13: d\_membus の型を変更する (core.veryl)

```

1 i_membus: modport membus_if:<ILEN, XLEN>::master,
2 d_membus: modport core_data_if::master           ,
3 led      : output UIntX                         ,
```

#### ▼ リスト 11.14: core\_data\_if インターフェースのインスタンス化 (top.veryl)

```

1 inst d_membus_core: core_data_if;
```

#### ▼ リスト 11.15: ポートに割り当てるインターフェースを変更する (top.veryl)

```

1 inst c: core (
2     clk           ,
3     rst           ,
4     i_membus      ,
5     d_membus: d_membus_core,
```

```

6     led          ,
7 );

```

memunit モジュールのインターフェースも変更し、`is_amo`、`aq`、`rl`、`amoop` に値を割り当てます（リスト 11.16、リスト 11.17、リスト 11.19、リスト 11.18、リスト 11.20）。

#### ▼リスト 11.16: membus の型を変更する (memunit.veryl)

```

1 stall : output logic           , // メモリアクセス命令が完了していない
2 membus: modport core_data_if::master, // メモリとのinterface
3 )

```

#### ▼リスト 11.17: 一時保存するレジスタの定義 (memunit.veryl)

```

1 var req_wen    : logic          ;
2 var req_addr   : Addr          ;
3 var req_wdata  : logic<MEMBUS_DATA_WIDTH>      ;
4 var req_wmask  : logic<MEMBUS_DATA_WIDTH / 8> ;
5 var req_is_amo: logic          ;
6 var req_amoop  : AMOOp         ;
7 var req_aq     : logic          ;
8 var req_rl     : logic          ;
9 var req_funct3: logic<3>       ;

```

#### ▼リスト 11.18: レジスタをリセットする (memunit.veryl)

```

1 always_ff {
2     if_reset {
3         state      = State::Init;
4         req_wen   = 0;
5         req_addr  = 0;
6         req_wdata = 0;
7         req_wmask = 0;
8         req_is_amo = 0;
9         req_amoop = 0 as AMOOp;
10        req_aq   = 0;
11        req_rl   = 0;
12        req_funct3 = 0;
13    } else {

```

#### ▼リスト 11.19: membus にレジスタの値を割り当てる (memunit.veryl)

```

1 always_comb {
2     // メモリアクセス
3     membus.valid = state == State::WaitReady;
4     membus.addr  = req_addr;
5     membus.wen   = req_wen;
6     membus.wdata  = req_wdata;
7     membus.wmask  = req_wmask;
8     membus.is_amo = req_is_amo;
9     membus.amoop  = req_amoop;
10    membus.aq    = req_aq;
11    membus.rl    = req_rl;

```

```
12 membus.funct3 = req_funct3;
```

▼リスト 11.20: メモリにアクセスする命令のとき、レジスタに情報を設定する (memunit.veryl)

```
1 case state {
2     State::Init: if is_new & inst_is_memop(ctrl) {
3         ...
4             req_is_amo = ctrl.is_amo;
5             req_amoop = ctrl.funct7[6:2] as AMOp;
6             req_aq = ctrl.funct7[1];
7             req_rl = ctrl.funct7[0];
8             req_funct3 = ctrl.funct3;
9     }
10    State::WaitReady: if membus.ready {
```

amounit モジュールを top モジュールでインスタンス化し、core モジュールと mmio\_controller モジュールのインターフェースを接続します（リスト 11.21）。

▼リスト 11.21: amounit モジュールをインスタンス化する (top.veryl)

```
1 inst amou: amounit (
2     clk           ,
3     rst           ,
4     slave : d_membus_core,
5     master: d_membus   ,
6 );
```

## 11.4 Zalrsc 拡張の実装

Zalrsc 拡張の命令を実装します。予約セットのサイズは実装が自由に決めるため、本書では 1 つのアドレスのみ保持できるようにします。

### 11.4.1 LR.W、LR.D 命令を実装する

32 ビット幅、64 ビット幅の LR 命令を実装します。LR.W 命令は memunit モジュールで 64 ビットに符号拡張されるため、amounit モジュールで LR.W 命令と LR.D 命令を区別する必要はありません。

amounit モジュールに予約セットを作成します（リスト 11.22、リスト 11.23）。  
is\_addr\_reserved で、予約セットに有効なアドレスが格納されているかを管理します。

▼リスト 11.22: 予約セットの定義 (amounit.veryl)

```
1 // lr/sc
2 var is_addr_reserved: logic;
3 var reserved_addr : Addr ;
```

## ▼ リスト 11.23: レジスタをリセットする (amounit.veryl)

```

1      is_addr_reserved = 0;
2      reserved_addr    = 0;

```

LR 命令を実行するとき、予約セットにアドレスを登録してロード結果を返すようにします（リスト 11.24、リスト 11.25、リスト 11.26）。既に予約セットが使われている場合はアドレスを上書きします。

## ▼ リスト 11.24: accept\_request\_comb 関数の実装 (amounit.veryl)

```

1  function accept_request_comb () {
2      if slave.ready && slave.valid {
3          if slave.is_amo {
4              case slave.amoop {
5                  AM0Op::LR: assign_master(slave.addr, 0, 0, 0);
6                  default : {}
7              }
8          } else {
9              assign_master(slave.addr, slave.wen, slave.wdata, slave.wmask);
10         }
11     }
12 }

```

## ▼ リスト 11.25: LR 命令のときに master にロード要求を割り当てる (amounit.veryl)

```

1  always_comb {
2      reset_master();
3      case state {
4          State::Init      : accept_request_comb();
5          State::WaitReady: if slave_saved.is_amo {
6              case slave_saved.amoop {
7                  AM0Op::LR: assign_master(slave_saved.addr, 0, 0, 0);
8                  default : {}
9              }
10         } else {
11             assign_master(slave_saved.addr, slave_saved.wen, slave_saved.wdata, slave_saved>.wmask);
12         }
13     }

```

## ▼ リスト 11.26: LR 命令のときに予約セットを設定する (amounit.veryl)

```

1  function accept_request_ff () {
2      slave_saved.valid = slave.ready && slave.valid;
3      if slave.ready && slave.valid {
4          slave_saved.addr    = slave.addr;
5          ...
6          slave_saved.funct3 = slave.funct3;
7          if slave.is_amo {
8              case slave.amoop {
9                  AM0Op::LR: {
10                      // reserve address
11                      is_addr_reserved = 1;
12                      reserved_addr   = slave.addr;
13                  }
14              }
15          }
16      }
17  }

```

```

13         state      = if master.ready ? State::WaitValid : State::WaitRe>
14     >ady;
15         }
16     default: {}
17     }
18 } else {
19     state = if master.ready ? State::WaitValid : State::WaitReady;
20 }
```

## 11.4.2 SC.W、SC.D 命令を実装する

32 ビット幅、64 ビット幅の SC 命令を実装します。SC.W 命令は memunit モジュールで書き込みマスクを設定しているため、amountunit モジュールで SC.W 命令と SC.D 命令を区別する必要はありません。

SC 命令が成功、失敗したときに結果を返すための状態を `State` 型に追加します（リスト 11.27）。

### ▼ リスト 11.27: SC 命令用の状態の定義 (amountunit.veryl)

```

1 enum State {
2     Init,
3     WaitReady,
4     WaitValid,
5     SCSuccess,
6     SCFail,
7 }
```

それぞれの状態で結果を返し、新しく要求を受け入れるようにします（リスト 11.28）。`State::SCSuccess` は SC 命令に成功してストアが終わったときに結果を返します。成功したら `0`、失敗したら `1` を返します。

### ▼ リスト 11.28: slave に SC 命令の結果を割り当てる (amountunit.veryl)

```

1 State::SCSuccess: {
2     slave.ready  = master.rvalid;
3     slave.rvalid = master.rvalid;
4     slave.rdata  = 0;
5 }
6 State::SCFail: {
7     slave.ready  = 1;
8     slave.rvalid = 1;
9     slave.rdata  = 1;
10 }
```

SC 命令を受け入れるときに予約セットを確認し、アドレスが予約セットのアドレスと異なる場合は状態を `State::SCFail` に移動します（リスト 11.29）。成功、失敗に関係なく、予約セットを空にします。

## ▼ リスト 11.29: accept\_request\_ff 関数で予約セットを確認する (amounit.veryl)

```

1  AM0Op::SC: {
2      // reset reserved
3      let prev           : logic = is_addr_reserved;
4      is_addr_reserved = 0;
5      // check
6      if prev && slave.addr == reserved_addr {
7          state = if master.ready ? State::SCSuccess : State::WaitReady;
8      } else {
9          state = State::SCFail;
10     }
11 }
```

SC 命令でメモリの `ready` が 1 になるのを待っているとき、`ready` が 1 になったら状態を `State::SCSuccess` に移動します（リスト 11.30）。また、命令の実行が終了したときに新しく要求を受け入れるようにします。

## ▼ リスト 11.30: SC 命令の状態遷移 (amounit.veryl)

```

1  function on_clock () {
2      case state {
3          State::Init      : accept_request_ff();
4          State::WaitReady: if master.ready {
5              if slave_saved.is_amo && slave_saved.amoop == AM0Op::SC {
6                  state = State::SCSuccess;
7              } else {
8                  state = State::WaitValid;
9              }
10         }
11         State::WaitValid: if master.rvalid {
12             accept_request_ff();
13         }
14         State::SCSuccess: if master.rvalid {
15             accept_request_ff();
16         }
17         State::SCFail: accept_request_ff();
18         default       : {}
19     }
20 }
```

SC 命令によるメモリへの書き込みを実装します（リスト 11.31、リスト 11.32）。

## ▼ リスト 11.31: accept\_request\_comb 関数で、予約セットをチェックしてからストアを要求する (amounit.veryl)

```

1  case slave.amoop {
2      AM0Op::LR: assign_master(slave.addr, 0, 0, 0);
3      AM0Op::SC: if is_addr_reserved && slave.addr == reserved_addr {
4          @<b> assign_master(slave.addr, 1, slave.wdata, slave.wmask); |</b>
5          @<b> } |</b>
6          default: {}
7      }
```

## ▼ リスト 11.32: master に値を割り当てる (amountunit.veryl)

```

1  always_comb {
2      reset_master();
3      case state {
4          State::Init      : accept_request_comb();
5          State::WaitReady: if slave_saved.is_amo {
6              case slave_saved.amoop {
7                  AMOp::LR: assign_master(slave_saved.addr, 0, 0, 0);
8                  AMOp::SC: assign_master(slave_saved.addr, 1, slave_saved.wdata, slave_saved>
9                      .wmask);
10             default : {}
11         }
12     } else {
13         assign_master(slave_saved.addr, slave_saved.wen, slave_saved.wdata, slave_saved>
14             .wmask);
15     }
16     State::WaitValid           : accept_request_comb();
17     State::SCFail, State::SCSuccess: accept_request_comb();
18     default                   : {}
19 }
20 }
```

## 11.5 Zaamo 拡張の実装

Zaamo 拡張の命令はロード、演算、ストアを行います。本章では、Zaamo 拡張の命令を `State::Init` (、`State::AMOLoadReady`)、`State::AMOLoadValid` (、`State::AMOSToreReady`)、`State::AMOSToreValid` という状態遷移で処理するように実装します。

`State` 型に新しい状態を定義してください (リスト 11.33)。

## ▼ リスト 11.33: Zaamo 拡張の命令用の状態の定義 (amountunit.veryl)

```

1 enum State {
2     Init,
3     WaitReady,
4     WaitValid,
5     SCSuccess,
6     SCFail,
7     AMOLoadReady,
8     AMOLoadValid,
9     AMOSToreReady,
10    AMOSToreValid,
11 }
```

簡単に Zalrsc 拡張と区別するために、Zaamo 拡張による要求かどうかを判定する関数 (`is_Zaamo`) を `core_data_if` インターフェースに作成します (リスト 11.34、リスト 11.35)。modport に import 宣言を追加してください。

## ▼リスト 11.34: is\_Zaamo 関数の定義 (core\_data\_if.veryl)

```

1  function is_Zaamo () -> logic {
2      return is_amo && (amoop != AMOp::LR && amoop != AMOp::SC);
3  }

```

## ▼リスト 11.35: master に is\_Zaamo 関数を import する (core\_data\_if.veryl)

```

1  amoop   : output,
2  funct3 : output,
3  is_Zaamo: import,
4 }

```

ロードした値と `wdata`、フラグを利用して、ストアする値を生成する関数を作成します（リスト 11.36）。32 ビット演算のとき、下位 32 ビットと上位 32 ビットのどちらを使うかをアドレスによって判別しています。

## ▼リスト 11.36: Zaamo 拡張の命令の計算を行う関数の定義 (amounit.veryl)

```

1 // AMO ALU
2 function calc_amo::<W: u32> (
3     amoop: input AMOp ,
4     wdata: input logic<W>,
5     rdata: input logic<W>,
6 ) -> logic<W> {
7     let lts: logic = $signed(wdata) <: $signed(rdata);
8     let ltu: logic = wdata <: rdata;
9
10    return case amoop {
11        AMOp::SWAP: wdata,
12        AMOp::ADD : rdata + wdata,
13        AMOp::XOR : rdata ^ wdata,
14        AMOp::AND : rdata & wdata,
15        AMOp::OR  : rdata | wdata,
16        AMOp::MIN : if lts ? wdata : rdata,
17        AMOp::MAX : if !lts ? wdata : rdata,
18        AMOp::MINU: if ltu ? wdata : rdata,
19        AMOp::MAXU: if !ltu ? wdata : rdata,
20        default   : 0,
21    };
22 }
23
24 // Zaamo拡張の命令のwdataを生成する
25 function gen_amo_wdata (
26     req : modport core_data_if::all_input,
27     rdata: input UIntX           ,
28 ) -> UIntX {
29     case req.funct3 {
30         3'b010: { // word
31             let low   : logic = req.addr[2] == 0;
32             let rdata32: UInt32 = if low ? rdata[31:0] : rdata[63:32];
33             let wdata32: UInt32 = if low ? req.wdata[31:0] : req.wdata[63:32];
34             let result : UInt32 = calc_amo::<32>(req.amoop, wdata32, rdata32);
35             return if low ? {rdata[63:32], result} : {result, rdata[31:0]};
36     }
37 }

```

```

36         }
37         3'b011 : return calc_amo::<64>(req.amoop, req.wdata, rdata); // double
38         default: return 0;
39     }
40 }
```

ロードした値が命令の結果になるため、値を保持するためのレジスタを作成します（リスト 11.37、リスト 11.38）。

#### ▼ リスト 11.37: ロードしたデータを格納するレジスタの定義 (amounit.veryl)

```

1 // amo
2 var zaamo_fetched_data: UIntX;
```

#### ▼ リスト 11.38: レジスタのリセット (amounit.veryl)

```

1 reserved_addr      = 0;
2 zaamo_fetched_data = 0;
3 }
```

メモリアクセスが終了したら、ロードした値を返します（リスト 11.39）。

#### ▼ リスト 11.39: 命令の結果を返す (amounit.veryl)

```

1 State::AMOStoreValid: {
2     slave.ready  = master.rvalid;
3     slave.rvalid = master.rvalid;
4     slave.rdata  = zaamo_fetched_data;
5 }
```

状態に基づいて、メモリへのロード、ストア要求を割り当てます（リスト 11.40、リスト 11.41）。

#### ▼ リスト 11.40: accept\_request\_comb 関数で、まずロード要求を行う (amounit.veryl)

```

1 default: if slave.is_Zaamo() {
2     assign_master(slave.addr, 0, 0, 0);
3 }
```

#### ▼ リスト 11.41: 状態に基づいてロード、ストア要求を行う (amounit.veryl)

```

1 State::AMOLoadReady           : assign_master    (slave_saved.addr, 0, 0, 0);
2 State::AMOLoadValid, State::AMOStoreReady: {
3     let rdata      : UIntX = if state == State::AMOLoadValid ? master.rdata : zaamo_fetch-
4     ed_data;
5     let wdata      : UIntX = gen_amo_wdata(slave_saved, rdata);
6     assign_master(slave_saved.addr, 1, wdata, slave_saved.wmask);
7 }
```

`master`、`slave` の状態によって `state` を遷移します（リスト 11.42）。

▼ リスト 11.42: accept\_request\_ff 関数で、master の ready によって次の state を決める (amounit.veryl)

```
1 default: if slave.is_Zaamo() {  
2     state = if master.ready ? State::AMOLoadValid : State::AMOLoadReady;  
3 }
```

▼ リスト 11.43: Zaamo 拡張の命令の状態の遷移 (amounit.veryl)

```
1 State::AMOLoadReady: if master.ready {  
2     state = State::AMOLoadValid;  
3 }  
4 State::AMOLoadValid: if master.rvalid {  
5     zaamo_fetched_data = master.rdata;  
6     state           = if slave.ready ? State::AMOStoreValid : State::AMOStoreReady;  
7 }  
8 State::AMOStoreReady: if master.ready {  
9     state = State::AMOStoreValid;  
10 }  
11 State::AMOStoreValid: if master.rvalid {  
12     accept_request_ff();  
13 }
```

riscv-tests の rv64ua-p- から始まるテストを実行し、成功することを確認してください。

# 第 12 章

## C 拡張の実装

### 12.1 概要

これまでに実装した命令はすべて 32 ビット幅のものでした。RISC-V には 32 ビット幅以外の命令が定義されており、命令の下位ビットで何ビット幅の命令か判断できます（表 12.1）。

▼表 12.1: RISC-V の命令長とエンコーディング

命令幅	命令の下位 5 ビット
16-bit (aa ≠ 11)	xxxxa
32-bit (bbb ≠ 111)	bbb11

C 拡張は 16 ビット幅の命令を定義する拡張です。よく使われる命令の幅を 16 ビットに圧縮できるようにすることでコードサイズを削減できます。これ以降、C 拡張によって導入される 16 ビット幅の命令のことを RVC 命令と呼びます。

全ての RVC 命令には同じ操作をする 32 ビット幅の命令が存在します<sup>\*1</sup>。

RVC 命令は図 12.1 の 9 つのフォーマットが定義されています。

`rs1'`、`rs2'`、`rd'` は 3 ビットのフィールドで、よく使われる 8 番 (x8) から 15 番 (x15) のレジスタを指定します。即値の並び方やそれぞれの命令の具体的なフォーマットについては、仕様書か「12.6.2 32 ビット幅の命令に変換する」(p.253) のコードを参照してください。

RV64I の CPU に実装される C 拡張には表 12.2 の RVC 命令が定義されています。

C.ADDIW 命令は RV32I の C 拡張に定義されている C.JAL 命令とエンコーディングが同じです。本書で実装するモジュールは RV32I の C 拡張にも対応したものになっています。RV32I の C 拡張については、仕様書か「12.6.2 32 ビット幅の命令に変換する」(p.253) のコードを参照してください。

C 拡張は浮動小数点命令をサポートする F、D 拡張が実装されている場合に他の命令を定義しま

<sup>\*1</sup> Zc\*拡張の一部の命令は複数の命令になります

Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
CR	funct4				rd/rs1				rs2				op					
CI	funct3		imm		rd/rs1				imm				op					
CSS	funct3		imm				rs2				op							
CIW	funct3		imm				rd'				op							
CL	funct3		imm		rs1'		imm		rd'		op							
CS	funct3		imm		rs1'		imm		rs2'		op							
CA	funct6				rd'/rs1'		funct2		rs2'		op							
CB	funct3		offset		rd'/rs1'		offset				op							
CJ	funct3		jump target				op											

▲図 12.1: RVC 命令のフォーマット

ですが、基本編では F、D 拡張を実装しないため実装、解説しません。

## 12.2 IALIGN の変更

「9.5 命令アドレスのミスアライン例外」(p.185) で解説したように、命令は IALIGN ビットに整列したアドレスに配置されます。C 拡張は IALIGN による制限を 16 ビットに緩め、全ての命令が 16 ビットに整列されたアドレスに配置されるように変更します。これにより、RVC 命令と 32 ビット幅の命令の組み合わせがあったとしても効果的にコードサイズを削減できます。

eei パッケージに定数 `IALIGN` を定義します (リスト 12.1)。

### ▼リスト 12.1: IALIGN の定義 (eei.veryl)

```
1 const IALIGN: u32 = 16;
```

mepc レジスタの書き込みマスクを変更して、トラップ時のジャンプ先アドレスに 16 ビットに整列されたアドレスを指定できるようにします (リスト 12.2)。

### ▼リスト 12.2: MEPC の書き込みマスクを変更する (eei.veryl)

```
1 const MEPC_WMASK : UIntX = 'hffff_ffff_ffff_ffffe;
```

命令アドレスのミスアライン例外の判定を変更します。IALIGN が 16 の場合は例外が発生しないようにします (リスト 12.3)。ジャンプ、分岐命令は 2 バイト単位のアドレスしか指定できないため、C 拡張が実装されている場合には例外が発生しません。

▼表12.2: C拡張の命令

命令	同じ意味の32ビット幅の命令	形式
C.LWSP	lw rd, offset(x2)	CI
C.LDSP	ld rd, offset(x2)	CI
C.SWSP	sw rs2, offset(x2)	CSS
C.SDSP	sd rs2, offset(x2)	CSS
C.LW	lw rd, offset(rs)	CL
C.LD	ld rd, offset(rs)	CL
C.SW	sw rs2, offset(rs1)	CS
C.SD	sd rs2, offset(rs1)	CS
C.J	jal x0, offset	CJ
C.JR	jalr x0, 0(rs1)	CR
C.JALR	jalr x1, 0(rs1)	CR
C.BEQZ	beq rs1, x0, offset	CB
C.BNEZ	bne rs1, x0, offset	CB
C.LI	addi rd, x0, imm	CI
C.LUI	lui rd, imm	CI
C.ADDI	addi rd, rd, imm	CI
C.ADDIW	addiw rd, rd, imm	CI
C.ADDI16SP	addi x2, x2, imm	CI
C.ADDI4SPN	addi rd, x2, imm	CIW
C.SLLI	slli rd, rd, shamt	CI
C.SRLI	srli rd, rd, shamt	CB
C.SRAI	srai rd, rd, shamt	CB
C.ANDI	andi rd, rd, imm	CB
C.MV	add rd, x0, rs2	CR
C.ADD	add rd, rd, rs2	CR
C.AND	and rd, rd, rs2	CA
C.OR	or rd, rd, rs2	CA
C.XOR	xor rd, rd, rs2	CA
C.SUB	sub rd, rd, rs2	CA
C.EBREAK	ebreak	CR

▼リスト12.3: IALIGNが16のときに例外が発生しないようにする(core.veril)

```
1 let instruction_address_misaligned: logic = IALIGN == 32 && memq_wdata.br_taken && memq_>
>wdata.jump_addr[1:0] != 2'b00;
```

## 12.3 実装方針

本章では次の順序でC拡張を実装します。

1. 命令フェッチ処理 (IF ステージ) を core モジュールから分離する
2. 16 ビットに整列されたアドレスに配置された 32 ビット幅の命令を処理できるようにする
3. RVC 命令を 32 ビット幅の命令に変換するモジュールを作成する
4. RVC 命令を 32 ビット幅の命令に変換して core モジュールに供給する

最終的な命令フェッチ処理の構成は図図 12.2 のようになります。



▲ 図 12.2: 命令フェッチ処理の構成

## 12.4 命令フェッチモジュールの実装

### 12.4.1 インターフェースを作成する

まず、命令フェッチを行うモジュールと core モジュールのインターフェースを定義します。

`src/core_inst_if.veryl` を作成し、次のように記述します (リスト 12.4)。

▼ リスト 12.4: `core_inst_if.veryl`

```

1 import eei::*;

2
3 interface core_inst_if {
4     var rvalid   : logic;
5     var rready   : logic;
6     var raddr    : Addr ;
7     var rdata    : Inst ;
8     var is_hazard: logic;
9     var next_pc  : Addr ;
10
11    modport master {
12        rvalid   : input ,
13        rready   : output,
14        raddr    : input ,
15        rdata    : input ,
16        is_hazard: output, // control hazard
17        next_pc  : output, // actual next pc
18    }
19
20    modport slave {
21        ..converse(master)
22    }
23}

```

`rvalid`、`rready`、`raddr`、`rdata` は、core モジュールの FIFO(`if_fifo`) の `wvalid`、

`wready`、`wdata.addr`、`wdata.bits` と同じ役割を果たします。`is_hazard`、`next_pc` は制御ハザードの情報を伝えるための変数です。

### 12.4.2 core モジュールの IF ステージを削除する

core モジュールの IF ステージを削除し、`core_inst_if` インターフェースで代替します<sup>\*2</sup>。

core モジュールの `i_membus` の型を `core_inst_if` に変更します（リスト 12.5）。

#### ▼ リスト 12.5: `i_membus` の型を変更する (core.veryl)

```
1 i_membus: modport core_inst_if::master,
```

IF ステージ部分のコードを次のように変更します（リスト 12.6）。

#### ▼ リスト 12.6: IF ステージの変更 (core.veryl)

```
1 ////////////////////////////////////////////////////////////////// IF Stage //////////////////////////////////////////////////////////////////
2
3 var control_hazard      : logic;
4 var control_hazard_pc_next: Addr ;
5
6 always_comb {
7     i_membus.is_hazard = control_hazard;
8     i_membus.next_pc = control_hazard_pc_next;
9 }
```

core モジュールの新しい IF ステージ部分は、制御ハザードの情報をインターフェースに割り当てるだけの簡単なものになっています。`if_fifo_type` 型、`if_fifo_` から始まる変数は使わなくなったので削除してください。

ID ステージと `core_inst_if` インターフェースを接続します（リスト 12.7、リスト 12.8）。もともと `if_fifo` の `rvalid`、`rready`、`rdata` だった部分を `i_membus` に変更しています。

#### ▼ リスト 12.7: ID ステージと `i_membus` を接続する (core.veryl)

```
1 let ids_valid      : logic    = i_membus.rvalid;
2 let ids_pc        : Addr     = i_membus.raddr;
3 let ids_inst_bits : Inst     = i_membus.rdata;
```

#### ▼ リスト 12.8: EX ステージに進められるときに `rready` を 1 にする (core.veryl)

```
1 always_comb {
2     // ID -> EX
3     i_membus.rready = exq_wready;
4     exq_wvalid     = i_membus.rvalid;
5     exq_wdata.addr = i_membus.raddr;
6     exq_wdata.bits = i_membus.rdata;
7     exq_wdata.ctrl = ids_ctrl;
8     exq_wdata.imm  = ids_imm;
```

<sup>\*2</sup> ここで削除するコードは次の「12.4.3 `inst_fetcher` モジュールを作成する」(p.244) で実装するコードと似通っているため、削除せずにコメントアウトしておくと少し楽に実装できます。

### 12.4.3 inst\_fetcher モジュールを作成する

IF ステージの代わりに命令フェッチをする inst\_fetcher モジュールを作成します。inst\_fetcher モジュールでは命令フェッチ処理を fetch、issue の 2 段階で行います。

#### fetch

メモリから 64 ビットの値を読み込み、issue との間の FIFO に格納する。アドレスを 8 進めて、次の 64 ビットを読み込む。

#### issue

fetch との間の FIFO から 64 ビットを読み込み、32 ビットずつ core モジュールとの間の FIFO に格納する。

fetch と issue は並列に独立して動かします。

inst\_fetcher モジュールのポートを定義します。 `src/inst_fetcher.vyrl` を作成し、次のように記述します（リスト 12.9）。

#### ▼ リスト 12.9: ポートの定義 (inst\_fetcher.vyrl)

```

1 module inst_fetcher (
2     clk      : input    clock          ,
3     rst      : input    reset          ,
4     core_if: modport core_inst_if::slave,
5     mem_if : modport Membus::master   ,
6 )

```

`core_if` は core モジュールとのインターフェース、`mem_if` はメモリとのインターフェースです。

fetch と issue、issue と core\_if の間の FIFO を作成します（リスト 12.10、リスト 12.11）。

#### ▼ リスト 12.10: fetch と issue を繋ぐ FIFO の作成 (inst\_fetcher.vyrl)

```

1 struct fetch_fifo_type {
2     addr: Addr          ,
3     bits: logic<MEMBUS_DATA_WIDTH>,
4 }
5
6 var fetch_fifo_flush : logic      ;
7 var fetch_fifo_wvalid: logic      ;
8 var fetch_fifo_wready: logic      ;
9 var fetch_fifo_wdata : fetch_fifo_type;
10 var fetch_fifo_rdata : fetch_fifo_type;
11 var fetch_fifo_rready: logic      ;
12 var fetch_fifo_rvalid: logic      ;
13
14 inst fetch_fifo: fifo #(
15     DATA_TYPE: fetch_fifo_type,
16     WIDTH    : 3           ,
17 ) (
18     clk      ,             ,
19     rst      ,

```

```

20     flush      : fetch_fifo_flush ,
21     wready     : _ ,
22     wready_two: fetch_fifo_wready,
23     wvalid     : fetch_fifo_wvalid,
24     wdata      : fetch_fifo_wdata ,
25     rready     : fetch_fifo_rready,
26     rvalid     : fetch_fifo_rvalid,
27     rdata      : fetch_fifo_rdata ,
28 );

```

## ▼ リスト 12.11: issue と core モジュールを繋ぐ FIFO の作成 (inst\_fetcher.vrtl)

```

1  struct issue_fifo_type {
2      addr: Addr,
3      bits: Inst,
4  }
5
6  var issue_fifo_flush : logic          ;
7  var issue_fifo_wvalid: logic          ;
8  var issue_fifo_wready: logic          ;
9  var issue_fifo_wdata : issue_fifo_type;
10 var issue_fifo_rdata : issue_fifo_type;
11 var issue_fifo_rready: logic          ;
12 var issue_fifo_rvalid: logic          ;
13
14 inst issue_fifo: fifo #(
15     DATA_TYPE: issue_fifo_type,
16     WIDTH    : 3
17 ) (
18     clk           ,
19     rst           ,
20     flush : issue_fifo_flush ,
21     wready: issue_fifo_wready,
22     wvalid: issue_fifo_wvalid,
23     wdata : issue_fifo_wdata ,
24     rready: issue_fifo_rready,
25     rvalid: issue_fifo_rvalid,
26     rdata : issue_fifo_rdata ,
27 );

```

メモリへのアクセス処理 (fetch) を実装します。FIFO に空きがあるとき、64 ビットの値を読み込んで PC を 8 進めます (リスト 12.12、リスト 12.13、リスト 12.14)。この処理は core モジュールの元の IF ステージとほとんど同じです。

## ▼ リスト 12.12: PC と状態管理用の変数の定義 (inst\_fetcher.vrtl)

```

1  var fetch_pc        : Addr ;
2  var fetch_requested : logic;
3  var fetch_pc_requested: Addr ;

```

## ▼リスト12.13: メモリへの要求の割り当て(inst\_fetcher.veryl)

```

1  always_comb {
2      mem_if.valid = 0;
3      mem_if.addr  = 0;
4      mem_if.wen   = 0;
5      mem_if.wdata = 0;
6      mem_if.wmask = 0;
7      if !core_if.is_hazard {
8          mem_if.valid = fetch_fifo_wready;
9          if fetch_requested {
10             mem_if.valid = mem_if.valid && mem_if.rvalid;
11         }
12         mem_if.addr = fetch_pc;
13     }
14 }
```

## ▼リスト12.14: PC、状態の更新(inst\_fetcher.veryl)

```

1  always_ff {
2      if_reset {
3          fetch_pc        = INITIAL_PC;
4          fetch_requested = 0;
5          fetch_pc_requested = 0;
6      } else {
7          if core_if.is_hazard {
8              fetch_pc        = {core_if.next_pc[XLEN - 1:3], 3'b0};
9              fetch_requested = 0;
10             fetch_pc_requested = 0;
11         } else {
12             if fetch_requested {
13                 if mem_if.rvalid {
14                     fetch_requested = mem_if.ready && mem_if.valid;
15                     if mem_if.ready && mem_if.valid {
16                         fetch_pc_requested = fetch_pc;
17                         fetch_pc        += 8;
18                     }
19                 }
20             } else {
21                 if mem_if.ready && mem_if.valid {
22                     fetch_requested = 1;
23                     fetch_pc_requested = fetch_pc;
24                     fetch_pc        += 8;
25                 }
26             }
27         }
28     }
29 }
```

メモリから読み込んだ値をissueとの間のFIFOに格納します(リスト12.15)。

## ▼リスト12.15: ロードした64ビットの値をFIFOに格納する(inst\_fetcher.veryl)

```

1 // memory -> fetch_fifo
2 always_comb {
```

```

3     fetch_fifo_flush      = core_if.is_hazard;
4     fetch_fifo_wvalid    = fetch_requested && mem_if.rvalid;
5     fetch_fifo_wdata.addr = fetch_pc_requested;
6     fetch_fifo_wdata.bits = mem_if.rdata;
7 }
```

core モジュールに命令を供給する処理(issue)を実装します。FIFO にデータが入っているとき、32 ビットずつ core モジュールとの間の FIFO に格納します。2 つの 32 ビットの命令を FIFO に格納出来たら、fetch との間の FIFO を読み進めます(リスト 12.16、リスト 12.17)。

#### ▼リスト 12.16: オフセットの更新(inst\_fetcher.veryl)

```

1 var issue_pc_offset: logic<3>;
2
3 always_ff {
4     if_reset {
5         issue_pc_offset = 0;
6     } else {
7         if core_if.is_hazard {
8             issue_pc_offset = core_if.next_pc[2:0];
9         } else {
10            if issue_fifo_wready && issue_fifo_wvalid {
11                issue_pc_offset += 4;
12            }
13        }
14    }
15 }
```

#### ▼リスト 12.17: issue\_fifo に 32 ビットずつ命令を格納する(inst\_fetcher.veryl)

```

1 // fetch_fifo <-> issue_fifo
2 always_comb {
3     let raddr : Addr                  = fetch_fifo_rdata.addr;
4     let rdata : logic<MEMBUS_DATA_WIDTH> = fetch_fifo_rdata.bits;
5     let offset: logic<3>              = issue_pc_offset;
6
7     fetch_fifo_rready = 0;
8     issue_fifo_wvalid = 0;
9     issue_fifo_wdata  = 0;
10
11    if !core_if.is_hazard && fetch_fifo_rvalid {
12        if issue_fifo_wready {
13            fetch_fifo_rready = offset == 4;
14            issue_fifo_wvalid = 1;
15            issue_fifo_wdata.addr = {raddr[msb:3], offset};
16            issue_fifo_wdata.bits = case offset {
17                0      : rdata[31:0],
18                4      : rdata[63:32],
19                default: 0,
20            };
21        }
22    }
23 }
```

`core_if` と FIFO を接続します (リスト 12.18)。

▼ リスト 12.18: `issue_fifo` とインターフェースを接続する (`inst_fetcher.veryl`)

```

1 // issue_fifo <-> core
2 always_comb {
3     issue_fifo_flush = core_if.is_hazard;
4     issue_fifo_rready = core_if.rready;
5     core_if.rvalid = issue_fifo_rvalid;
6     core_if.raddr = issue_fifo_rdata.addr;
7     core_if.rdata = issue_fifo_rdata.bits;
8 }
```

#### 12.4.4 `inst_fetcher` モジュールと `core` モジュールを接続する

top モジュールで、`core_inst_if` をインスタンス化します。 (リスト 12.19)。

▼ リスト 12.19: インターフェースの定義 (`top.veryl`)

```
1 inst i_membus_core: core_inst_if;
```

`inst_fetcher` モジュールをインスタンス化し、`core` モジュールと接続します (リスト 12.20、リスト 12.21)。

▼ リスト 12.20: `inst_fetcher` モジュールのインスタンス化 (`top.veryl`)

```

1 inst fetcher: inst_fetcher (
2     clk           ,
3     rst           ,
4     core_if: i_membus_core,
5     mem_if : i_membus      ,
6 );
```

▼ リスト 12.21: インターフェースを変更する (`top.veryl`)

```

1 inst c: core (
2     clk           ,
3     rst           ,
4     i_membus: i_membus_core,
5     d_membus: d_membus_core,
6     led          ,
7 );
```

`inst_fetcher` モジュールが 64 ビットのデータを 32 ビットの命令の列に変換してくれたようになったので、`d_membus` との調停のところで 32 ビットずつ選択する必要がなくなりました。そのため、`rdata` をそのまま割り当てて、`memarb_last_iaddr` 変数とビットの選択処理を削除します (リスト 12.22、リスト 12.23、リスト 12.24)。

▼ リスト 12.22: 使用しない変数を削除する (`top.veryl`)

```

1 var memarb_last_i: logic;
2 var memarb_last_iaddr: Addr;
```

▼リスト12.23: 使用しない変数を削除する(*top.veryl*)

```

1 always_ff {
2     if_reset {
3         memarb_last_i = 0;
4         memarb_last_i = 0;
5     } else {
6         if mmio_membus.ready {
7             memarb_last_i = !d_membus.valid;
8             memarb_last_i_addr = i_membus.addr;
9         }
10    }
11 }
```

▼リスト12.24: ビットの選択処理を削除する(*top.veryl*)

```

1 always_comb {
2     i_membus.ready = mmio_membus.ready && !d_membus.valid;
3     i_membus.rvalid = mmio_membus.rvalid && memarb_last_i;
4     i_membus.rdata = mmio_membus.rdata;
```

## 12.5 16ビット境界に配置された32ビット幅の命令のサポート

*inst\_fetcher*モジュールで、アドレスが2バイトの倍数の32ビット幅の命令をcoreモジュールに供給できるようにします。

アドレスの下位3ビット(*issue\_pc\_offset*)が6の場合、*issue*とcoreの間に供給する命令のビット列は*fetch\_fifo\_rdata*の上位16ビットと*fetch\_fifo*に格納されている次のデータの下位16ビットを結合したものになります。このとき、*fetch\_fifo\_rdata*のデータの下位16ビットとアドレスを保存して、次のデータを読み出します。*fetch\_fifo*から次のデータを読み出せたら、保存していたデータと結合し、アドレスとともに*issue\_fifo*に書き込みます。*issue\_pc\_offset*が0、2、4の場合、既存の処理との変更点はありません。

*fetch\_fifo\_rdata*のデータの下位16ビットとアドレスを保持する変数を作成します(リスト12.25)。

▼リスト12.25: データを一時保存するための変数の定義(*inst\_fetcher.veryl*)

```

1 var issue_is_rdata_saved: logic      ;
2 var issue_saved_addr    : Addr      ;
3 var issue_saved_bits   : logic<16>; // rdata[63:48]
```

*issue\_pc\_offset*が6のとき、変数にデータを保存します(リスト12.26)。

▼リスト12.26: offsetが6のとき、変数に命令の下位16ビットとアドレスを保存する(*inst\_fetcher.veryl*)

```

1 always_ff {
2     if_reset {
```

```

3      issue_pc_offset      = 0;
4      issue_is_rdata_saved = 0;
5      issue_saved_addr    = 0;
6      issue_saved_bits    = 0;
7 } else {
8     if core_if.is_hazard {
9         issue_pc_offset      = core_if.next_pc[2:0];
10        issue_is_rdata_saved = 0;
11    } else {
12        // offsetが6な32ビット命令の場合、
13        // アドレスと上位16ビットを保存してFIFOを読み進める
14        if issue_pc_offset == 6 && !issue_is_rdata_saved {
15            if fetch_fifo_rvalid {
16                issue_is_rdata_saved = 1;
17                issue_saved_addr    = fetch_fifo_rdata.addr;
18                issue_saved_bits    = fetch_fifo_rdata.bits[63:48];
19            }
20        } else {
21            if issue_fifo_wready && issue_fifo_wvalid {
22                issue_pc_offset      += 4;
23                issue_is_rdata_saved = 0;
24            }
25        }
26    }
27 }
28 }
```

`issue_pc_offset` が 2、6 の場合の `issue_fifo` への書き込みを実装します（リスト 12.27）。  
 6 の場合、保存していた 16 ビットと新しく読み出した 16 ビットを結合した値、保存していたアドレスを書き込みます。

▼リスト 12.27: `issue_fifo` に offset が 2、6 の命令を格納する (`inst_fetcher.vrtl`)

```

1  if !core_if.is_hazard && fetch_fifo_rvalid {
2      if issue_fifo_wready {
3          if offset == 6 {
4              // offsetが6な32ビット命令の場合、
5              // 命令は{rdata[15:0], rdata[63:48]}になる
6              if issue_is_rdata_saved {
7                  issue_fifo_wvalid      = 1;
8                  issue_fifo_wdata.addr = {issue_saved_addr[msb:3], offset};
9                  issue_fifo_wdata.bits = {rdata[15:0], issue_saved_bits};
10             } else {
11                 // Read next 8 bytes
12                 fetch_fifo_rready = 1;
13             }
14         } else {
15             fetch_fifo_rready      = offset == 4;
16             issue_fifo_wvalid      = 1;
17             issue_fifo_wdata.addr = {raddr[msb:3], offset};
18             issue_fifo_wdata.bits = case offset {
19                 0      : rdata[31:0],
20                 2      : rdata[47:16],
```

```

21          4      : rdata[63:32],
22          default: 0,
23      };
24  }
25 }
26

```

32 ビット幅の命令の下位 16 ビットが既に保存されている（`issue_is_rdata_saved` が 1）とき、`fetch_fifo` から供給されるデータには、32 ビット幅の命令の上位 16 ビットを除いた残りの 48 ビットが含まれているので `fetch_fifo_rready` を 1 に設定しないことに注意してください。

## 12.6 RVC 命令の変換

### 12.6.1 RVC 命令フラグの実装

RVC 命令を 32 ビット幅の命令に変換するモジュールを作る前に、RVC 命令かどうかを示すフラグを作成します。

まず、`core_inst_if` インターフェースと `InstCtrl` 構造体に `is_rvc` フラグを追加します（リスト 12.28、リスト 12.29、リスト 12.30）。

#### ▼リスト 12.28: is\_rvc フラグの定義 (core\_inst\_if.veryl)

```

1  var rdata    : Inst ;
2  var is_rvc   : logic;
3  var is_hazard: logic;

```

#### ▼リスト 12.29: modport に is\_rvc を追加する (core\_inst\_if.veryl)

```

1  modport master {
2      rvalid   : input ,
3      rready   : output,
4      raddr    : input ,
5      rdata    : input ,
6      is_rvc   : input ,
7      is_hazard: output, // control hazard
8      next_pc  : output, // actual next pc
9  }

```

#### ▼リスト 12.30: InstCtrl 型に is\_rvc フラグを追加する (corectrl.veryl)

```

1  is_amo   : logic      , // AMO instruction
2  is_rvc   : logic      , // RVC instruction
3  funct3  : logic <3>, // 命令のfunct3フィールド

```

`inst_fetcher` モジュールで、`is_rvc` を 0 に設定して core モジュールに供給します（リスト 12.31、リスト 12.32、リスト 12.33）。

## ▼リスト12.31: issue\_fifo\_type型にis\_rvcフラグを追加する(inst\_fetcher.veryl)

```

1 struct issue_fifo_type {
2     addr : Addr ,
3     bits : Inst ,
4     is_rvc: logic,
5 }
```

## ▼リスト12.32: is\_rvcフラグを0に設定する(inst\_fetcher.veryl)

```

1 if offset == 6 {
2     // offsetが6な32ビット命令の場合、
3     // 命令は{rdata_next[15:0], rdata[63:48]}になる
4     if issue_is_rdata_saved {
5         issue_fifo_wvalid      = 1;
6         issue_fifo_wdata.addr  = {issue_saved_addr[msb:3], offset};
7         issue_fifo_wdata.bits   = {rdata[15:0], issue_saved_bits};
8         issue_fifo_wdata.is_rvc = 0;
9     } else {
10        // Read next 8 bytes
11        fetch_fifo_rready = 1;
12    }
13 } else {
14     fetch_fifo_rready      = offset == 4;
15     issue_fifo_wvalid      = 1;
16     issue_fifo_wdata.addr  = {raddr[msb:3], offset};
17     issue_fifo_wdata.bits   = case offset {
18         0      : rdata[31:0],
19         2      : rdata[47:16],
20         4      : rdata[63:32],
21         default: 0,
22     };
23     issue_fifo_wdata.is_rvc = 0;
24 }
```

## ▼リスト12.33: is\_rvcフラグを接続する(inst\_fetcher.veryl)

```

1 always_comb {
2     issue_fifo_flush  = core_if.is_hazard;
3     issue_fifo_rready = core_if.rready;
4     core_if.rvalid    = issue_fifo_rvalid;
5     core_if.raddr     = issue_fifo_rdata.addr;
6     core_if.rdata     = issue_fifo_rdata.bits;
7     core_if.is_rvc   = issue_fifo_rdata.is_rvc;
8 }
```

inst\_decoderモジュールで、InstCtrl構造体のis\_rvcフラグを設定します(リスト12.34、リスト12.35、リスト12.36)。また、C拡張が無効なのにRVC命令が供給されたらvalidフラグを0に設定します。

## ▼リスト12.34: is\_rvcフラグをポートに追加する(inst\_decoder.veryl)

```

1 module inst_decoder (
2     bits : input Inst ,
```

```

3   is_rvc: input logic ,
4   valid : output logic ,
5   ctrl  : output InstCtrl,
6   imm   : output UIntX ,
7 ) {

```

## ▼リスト 12.35: InstCtrl に is\_rvc フラグを設定する (inst\_decoder.veryl)

```

1     default: {
2         InstType::X, F, F, F, F, F, F, F, F, F
3     },
4     }, is_rvc, f3, f7
5 };

```

## ▼リスト 12.36: IALIGN が 32 ではないとき、不正な命令にする (inst\_decoder.veryl)

```

1     OP_AMO      : f3 == 3'b010 || f3 == 3'b011, // AMO
2     default     : F,
3     } && (IALIGN == 16 || !is_rvc); // IALIGN == 32 のとき、C拡張は無効

```

core モジュールで、inst\_decoder モジュールに `is_rvc` フラグを渡します（リスト 12.37）。

## ▼リスト 12.37: is\_rvc フラグを inst\_decoder に渡す (core.veryl)

```

1   inst decoder: inst_decoder (
2     bits  : ids_inst_bits ,
3     is_rvc: i_membus.is_rvc,
4     valid : ids_inst_valid ,
5     ctrl  : ids_ctrl       ,
6     imm   : ids_imm        ,
7 );

```

ジャンプ命令でライトバックする値は次の命令のアドレスであるため、RVC 命令の場合は PC に `2` を足した値を設定します（リスト 12.38）。

## ▼リスト 12.38: 次の命令のアドレスを変える (core.veryl)

```

1   let wbs_wb_data: UIntX    = if wbs_ctrl.is_lui ?
2     wbs_imm
3     : if wbs_ctrl.is_jump ?
4       wbs_pc + (if wbs_ctrl.is_rvc ? 2 : 4)
5     : if wbs_ctrl.is_load || wbs_ctrl.is_amo ?

```

## 12.6.2 32 ビット幅の命令に変換する

RVC 命令の opcode、funct などのフィールドを読んで、32 ビット幅の命令を生成する rvc\_converter モジュールを実装します。

その前に、命令のフィールドを引数に 32 ビット幅の命令を生成する関数を実装します。`src/inst_gen_pkg.veryl` を作成し、次のように記述します（リスト 12.39）。関数の名前は基本的に命令名と同じにしていますが、Veryl のキーワードと被るものは `inst_` を prefix にしています。

## ▼リスト12.39: 命令のビット列を生成する関数を定義する(inst\_gen\_pkg.veryl)

```

1 import eei::*;
2
3 package inst_gen_pkg {
4     function add (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
5         return {7'b0000000, rs2, rs1, 3'b000, rd, OP_OP};
6     }
7
8     function addw (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
9         return {7'b0000000, rs2, rs1, 3'b000, rd, OP_OP_32};
10    }
11
12    function addi (rd : input logic<5> , rs1: input logic<5> , imm: input logic<12>) -> Inst {
13        return {imm, rs1, 3'b000, rd, OP_OP_IMM};
14    }
15
16    function addiw (rd: input logic<5> ,rs1: input logic<5>, imm: input logic<12>) -> Inst {
17        return {imm, rs1, 3'b000, rd, OP_OP_IMM_32};
18    }
19
20    function sub (rd: input logic<5>,rs1: input logic<5>, rs2: input logic<5>) -> Inst {
21        return {7'b0100000, rs2, rs1, 3'b000, rd, OP_OP};
22    }
23
24    function subw (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
25        return {7'b0100000, rs2, rs1, 3'b000, rd, OP_OP_32};
26    }
27
28    function inst_xor (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
29        return {7'b0000000, rs2, rs1, 3'b100, rd, OP_OP};
30    }
31
32    function inst_or (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
33        return {7'b0000000, rs2, rs1, 3'b110, rd, OP_OP};
34    }
35
36    function inst_and (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
37        return {7'b0000000, rs2, rs1, 3'b111, rd, OP_OP};
38    }
39
40    function andi (rd: input logic<5> , rs1: input logic<5>, imm: input logic<12>) -> Inst {
41        return {imm, rs1, 3'b111, rd, OP_OP_IMM};
42    }
43
44    function slli (rd: input logic<5>, rs1: input logic<5>, shamt: input logic<6>) -> Inst {
45        return {6'b000000, shamt, rs1, 3'b001, rd, OP_OP_IMM};
46    }
47
48    function srli (rd: input logic<5>, rs1: input logic<5>, shamt: input logic<6>) -> Inst {
49        return {6'b000000, shamt, rs1, 3'b101, rd, OP_OP_IMM};
50    }
51
52    function srai (rd: input logic<5>, rs1: input logic<5>, shamt: input logic<6>) -> Inst {

```

```

53     return {6'b010000, shamt, rs1, 3'b101, rd, OP_OP_IMM};
54 }
55
56     function lui (rd: input logic<5>, imm: input logic<20>) -> Inst {
57         return {imm, rd, OP_LUI};
58     }
59
60     function load (rd: input logic<5>, rs1: input logic<5>, imm: input logic<12>, funct3: input > logic<3>) -> Inst {
61         return {imm, rs1, funct3, rd, OP_LOAD};
62     }
63
64     function store (rs1: input logic<5>, rs2: input logic<5>, imm: input logic<12>, funct3: input > t logic<3>) -> Inst {
65         return {imm[11:5], rs2, rs1, funct3, imm[4:0], OP_STORE};
66     }
67
68     function jal (rd : input logic<5>, imm: input logic<20>) -> Inst {
69         return {imm[19], imm[9:0], imm[10], imm[18:11], rd, OP_JAL};
70     }
71
72     function jalr (rd: input logic<5>, rs1: input logic<5>, imm: input logic<12>) -> Inst {
73         return {imm, rs1, 3'b000, rd, OP_JALR};
74     }
75
76     function beq (rs1: input logic<5>, rs2: input logic<5>, imm: input logic<12>) -> Inst {
77         return {imm[11], imm[9:4], rs2, rs1, 3'b000, imm[3:0], imm[10], OP_BRANCH};
78     }
79
80     function bne (rs1: input logic<5>, rs2: input logic<5>, imm: input logic<12>) -> Inst {
81         return {imm[11], imm[9:4], rs2, rs1, 3'b001, imm[3:0], imm[10], OP_BRANCH};
82     }
83
84     function ebreak () -> Inst {
85         return 32'h00100073;
86     }
87 }
```

rvc\_converter モジュールのポートを定義します。 `src/rvc_converter.vyrl` を作成し、次のように記述します（リスト 12.40）。

#### ▼リスト 12.40: ポートの定義 (rvc\_converter.vyrl)

```

1 import eei::*;
2 import inst_gen_pkg::*;
3
4 module rvc_converter (
5     inst16: input logic<16>,
6     is_rvc: output logic ,
7     inst32: output Inst ,
8 ) {
```

rvc\_converter モジュールは、`inst16` で 16 ビットの値を受け取り、それが RVC 命令な

ら `is_rvc` を 1 にして、`inst32` に同じ意味の 32 ビット幅の命令を出力する組み合わせ回路です。

`inst16` からソースレジスタ番号を生成します（リスト 12.41）。`rs1d`、`rs2d` の範囲は `x8` から `x15` です。

#### ▼リスト 12.41: レジスタ番号の生成 (rvc\_converter.vetyl)

```

1  let rs1 : logic<5> = inst16[11:7];
2  let rs2 : logic<5> = inst16[6:2];
3  let rs1d: logic<5> = {2'b01, inst16[9:7]};
4  let rs2d: logic<5> = {2'b01, inst16[4:2]};
```

`inst16` から即値を生成します（リスト 12.42）。

#### ▼リスト 12.42: 即値の生成 (rvc\_converter.vetyl)

```

1  let imm_i    : logic<12> = {inst16[12] repeat 7, inst16[6:2]};
2  let imm_shamt: logic<6>  = {inst16[12], inst16[6:2]};
3  let imm_j    : logic<20> = {inst16[12] repeat 10, inst16[8], inst16[10:9], inst16[6], inst16
>[7], inst16[2], inst16[11], inst16[5:3]};
4  let imm_br   : logic<12> = {inst16[12] repeat 5, inst16[6:5], inst16[2], inst16[11:10], inst
>16[4:3]};
5  let c0_mem_w : logic<12> = {5'b0, inst16[5], inst16[12:10], inst16[6], 2'b0}; // C.LW, C.SW
6  let c0_mem_d : logic<12> = {4'b0, inst16[6:5], inst16[12:10], 3'b0}; // C.LD, C.SD
```

`inst16` から 32 ビット幅の命令を生成します（リスト 12.43）。`opcode`(`inst16[1:0]`) が `2'b11` 以外なら 16 ビット幅の命令なので、`is_rvc` に 1 を割り当てます。`inst32` には、初期値として右に `inst16` を詰めてゼロで拡張した値を割り当てます。

32 ビット幅の命令への変換は `opcode`、`funct`、レジスタ番号などで分岐して地道に実装します。32 ビット幅の命令に変換できないとき `inst32` の値を更新しません。

`inst16` が不正な RVC 命令のとき、`inst_decoder` モジュールでデコードできない命令を core モジュールに供給して `Illegal instruction` 例外を発生させ、`tval` に 16 ビット幅の不正な命令が設定されます。

#### ▼リスト 12.43: RVC命令を 32ビット幅の命令に変換する (rvc\_converter.vetyl)

```

1  always_comb {
2      is_rvc = inst16[1:0] != 2'b11;
3      inst32 = {16'b0, inst16};
4
5      let funct3: logic<3> = inst16[15:13];
6      case inst16[1:0] { // opcode
7          2'b00: case funct3 { // C0
8              3'b000: if inst16 != 0 { // C.ADDI4SPN
9                  let nzuimm: logic<10> = {inst16[10:7], inst16[12:11], inst16[5], inst16[6], >
10                 2'b0};
11                  inst32 = addi(rs2d, 2, {2'b0, nzuimm});
12              }
13              3'b010: inst32 = load(rs2d, rs1d, c0_mem_w, 3'b010); // C.LW
```

```

13      3'b011: if XLEN >= 64 { // C.LD
14          inst32 = load(rs2d, rs1d, c0_mem_d, 3'b011);
15      }
16      3'b110: inst32 = store(rs1d, rs2d, c0_mem_w, 3'b010); // C.SW
17      3'b111: if XLEN >= 64 { // C.SD
18          inst32 = store(rs1d, rs2d, c0_mem_d, 3'b011);
19      }
20      default: {}
21  }
22  2'b01: case funct3 { // C1
23      3'b000: inst32 = addi(rs1, rs1, imm_i); // C.ADDI
24      3'b001: inst32 = if XLEN == 32 ? jal(1, imm_j) : addiw(rs1, rs1, imm_i); // C.JAL
25      3'b010: inst32 = addi(rs1, 0, imm_i); // C.LI
26      3'b011: if rs1 == 2 { // C.ADDI16SP
27          let imm : logic<10> = {inst16[12], inst16[4:3], inst16[5], inst16[2], imm16[15:16]};
28          inst32 = addi(2, 2, {imm[msb] repeat 2, imm});
29      } else { // C.LUI
30          inst32 = lui(rs1, {imm_i[msb] repeat 8, imm_i});
31      }
32      3'b100: case inst16[11:10] { // funct2 or funct6[1:0]
33          2'b00: if !(XLEN == 32 && imm_shamt[msb] == 1) {
34              inst32 = srli(rs1d, rs1d, imm_shamt); // C.SRLI
35          }
36          2'b01: if !(XLEN == 32 && imm_shamt[msb] == 1) {
37              inst32 = srai(rs1d, rs1d, imm_shamt); // C.SRAI
38          }
39          2'b10: inst32 = andi(rs1d, rs1d, imm_i); // C.ANDI
40          2'b11: if inst16[12] == 0 {
41              case inst16[6:5] {
42                  2'b00 : inst32 = sub(rs1d, rs1d, rs2d); // C.SUB
43                  2'b01 : inst32 = inst_xor(rs1d, rs1d, rs2d); // C.XOR
44                  2'b10 : inst32 = inst_or(rs1d, rs1d, rs2d); // C.OR
45                  2'b11 : inst32 = inst_and(rs1d, rs1d, rs2d); // C.AND
46                  default: {}
47              }
48          } else {
49              if XLEN >= 64 {
50                  if inst16[6:5] == 2'b00 {
51                      inst32 = subw(rs1d, rs1d, rs2d); // C.SUBW
52                  } else if inst16[6:5] == 2'b01 {
53                      inst32 = addw(rs1d, rs1d, rs2d); // C.ADDW
54                  }
55              }
56          }
57          default: {}
58      }
59      3'b101 : inst32 = jal(0, imm_j); // C.J
60      3'b110 : inst32 = beq(rs1d, 0, imm_br); // C.BEQZ
61      3'b111 : inst32 = bne(rs1d, 0, imm_br); // C.BNEZ
62      default: {}
63  }

```

```

64      2'b10: case funct3 { // C2
65          3'b000: if !(XLEN == 32 && imm_shamt[msb] == 1) {
66              inst32 = slli(rs1, rs1, imm_shamt); // C.SLLI
67          }
68          3'b010: if rs1 != 0 { // C.LWSP
69              let offset: logic<8> = {inst16[3:2], inst16[12], inst16[6:4], 2'b0};
70              inst32 = load(rs1, 2, {4'b0, offset}, 3'b010);
71          }
72          3'b011: if XLEN >= 64 && rs1 != 0 { // C.LDSP
73              let offset: logic<9> = {inst16[4:2], inst16[12], inst16[6:5], 3'b0};
74              inst32 = load(rs1, 2, {3'b0, offset}, 3'b011);
75          }
76          3'b100: if inst16[12] == 0 {
77              inst32 = if rs2 == 0 ? jalr(0, rs1, 0) : addi(rs1, rs2, 0); // C.JR / C.MA
    >V
78          } else {
79              if rs2 == 0 {
80                  inst32 = if rs1 == 0 ? ebreak() : jalr(1, rs1, 0); // C.EBREAK : C.JA
    >LR
81              } else {
82                  inst32 = add(rs1, rs1, rs2); // C.ADD
83              }
84          }
85          3'b110: { // C.SWSP
86              let offset: logic<8> = {inst16[8:7], inst16[12:9], 2'b0};
87              inst32 = store(2, rs2, {4'b0, offset}, 3'b010);
88          }
89          3'b111: if XLEN >= 64 { // C.SDSP
90              let offset: logic<9> = {inst16[9:7], inst16[12:10], 3'b0};
91              inst32 = store(2, rs2, {3'b0, offset}, 3'b011);
92          }
93          default: {}
94      }
95      default: {}
96  }
97 }

```

### 12.6.3 RVC命令を発行する

inst\_fetcherモジュールで rvc\_converter モジュールをインスタンス化し、RVC命令を core モジュールに供給します。

まず、rvc\_converter モジュールをインスタンス化します（リスト 12.44）。

#### ▼リスト 12.44: rvc\_converter モジュールのインスタンス化 (inst\_fetcher.veryl)

```

1 // instruction converter
2 var rvcc_inst16: logic<16>;
3 var rvcc_is_rvc: logic      ;
4 var rvcc_inst32: Inst      ;
5
6 inst rvcc: rvc_converter (
7     inst16: case issue_pc_offset {

```

```

8      0      : fetch_fifo_rdata.bits[15:0],
9      2      : fetch_fifo_rdata.bits[31:16],
10     4      : fetch_fifo_rdata.bits[47:32],
11     6      : fetch_fifo_rdata.bits[63:48],
12     default: 0,
13   },
14   is_rvc: rvcc_is_rvc,
15   inst32: rvcc_inst32,
16 );

```

RVC命令のとき、変換された32ビット幅の命令をissue\_fifoに書き込み、issue\_pc\_offsetを4ではなく2増やすようにします(リスト12.45、リスト12.46)。

#### ▼リスト12.45: RVC命令のときのオフセットの更新(inst\_fetcher.veryl)

```

1 // offsetが6な32ビット命令の場合、
2 // アドレスと上位16ビットを保存してFIFOを読み進める
3 if issue_pc_offset == 6 && !rvcc_is_rvc && !issue_is_rdata_saved {
4     if fetch_fifo_rvalid {
5         issue_is_rdata_saved = 1;
6         issue_saved_addr    = fetch_fifo_rdata.addr;
7         issue_saved_bits    = fetch_fifo_rdata.bits[63:48];
8     }
9 } else {
10     if issue_fifo_wready && issue_fifo_wvalid {
11         issue_pc_offset      += if issue_is_rdata_saved || !rvcc_is_rvc ? 4 : 2;
12         issue_is_rdata_saved = 0;
13     }
14 }

```

#### ▼リスト12.46: RVC命令のときのissue\_fifoへの書き込み(inst\_fetcher.veryl)

```

1 if !core_if.is_hazard && fetch_fifo_rvalid {
2     if issue_fifo_wready {
3         if offset == 6 {
4             // offsetが6な32ビット命令の場合、
5             // 命令は{rdata_next[15:0], rdata[63:48]}になる
6             if issue_is_rdata_saved {
7                 issue_fifo_wvalid      = 1;
8                 issue_fifo_wdata.addr  = {issue_saved_addr[msb:3], offset};
9                 issue_fifo_wdata.bits   = {rdata[15:0], issue_saved_bits};
10                issue_fifo_wdata.is_rvc = 0;
11            } else {
12                fetch_fifo_rready = 1;
13                if rvcc_is_rvc {
14                    issue_fifo_wvalid      = 1;
15                    issue_fifo_wdata.addr  = {raddr[msb:3], offset};
16                    issue_fifo_wdata.is_rvc = 1;
17                    issue_fifo_wdata.bits   = rvcc_inst32;
18                } else {
19                    // Read next 8 bytes
20                }
21            }

```

```
22 } else {
23     fetch_fifo_rready      = !rvcc_is_rvc && offset == 4;
24     issue_fifo_wvalid      = 1;
25     issue_fifo_wdata.addr  = {raddr[msb:3], offset};
26     if rvcc_is_rvc {
27         issue_fifo_wdata.bits = rvcc_inst32;
28     } else {
29         issue_fifo_wdata.bits = case offset {
30             0      : rdata[31:0],
31             2      : rdata[47:16],
32             4      : rdata[63:32],
33             default: 0,
34         };
35     }
36     issue_fifo_wdata.is_rvc = rvcc_is_rvc;
37 }
38 }
39 }
```

riscv-tests の `rv64uc-p-` から始まるテストを実行し、成功することを確認してください。

# **第 III 部**

# **特権/割り込みの実装**

# 第 13 章

## M-mode の実装 (1. CSR の実装)

### 13.1 概要

「第 II 部 RV64IMAC の実装」では、RV64IMAC と例外、メモリマップド I/O を実装しました。  
「第 III 部 特権/割り込みの実装」では、次の機能を実装します。

- 特権レベル (M-mode、S-mode、U-mode)
- 仮想記憶システム (ページング)
- 割り込み (CLINT、PLIC)

これらの機能を実装した CPU は OS を動かせる十分な機能を持っています。第 III 部の最後では Linux を動かします。

#### 13.1.1 特権レベルとは何か？

CPU で動くアプリケーションは様々ですが、多くのアプリケーションは OS(Operating System、オペレーティングシステム) の上で動かすことを前提に作成されています。「OS の上で動かす」とは、アプリケーションは OS の機能を使い、OS に管理されながら実行されるということです。

多くの OS はデバイスやメモリなどのリソースの管理を行い、簡単にそれを扱うためのインターフェースをアプリケーションに提供します。また、アプリケーションのデータを別のアプリケーションから保護したり、OS が提供する方法でしかデバイスにアクセスできなくなるなどのセキュリティ機能も備えています。

セキュリティ機能を実現するためには、OS がアプリケーションを実行するときに CPU が提供する一部の機能を制限する機能が必要です。RISC-V では、この機能を特権レベル (privilege level) という機能、枠組みによって提供しています。ほとんどの特権レベルの機能は CSR を通じて提供されます。

特権レベルは M-mode、S-mode、U-mode の 3 種類<sup>\*1</sup>が用意されています。それぞれの特権レ

<sup>\*1</sup> V 拡張が実装されている場合、さらに仮想化のための特権レベルが定義されます。

ベルは 2 ビットの数値で表すことができます (リスト 13.1)。数値が大きい方が高い特権レベルです。

高い特権レベルには低い特権レベルの機能を制限する機能があつたり、高い特権レベルでしか利用できない機能が定義されています。

特権レベルを表す `PrivMode` 型を eei パッケージに定義してください (リスト 13.1)。

#### ▼ リスト 13.1: PrivMode 型の定義 (eei.veryl)

```

1 enum PrivMode: logic<2> {
2     M = 2'b11,
3     S = 2'b01,
4     U = 2'b00,
5 }
```

### 13.1.2 特権レベルの実装順序

RISC-V の CPU に特権レベルを実装するとき、表 13.1 のいずれかの構成にする必要があります。特権レベルを実装していないときは M-mode だけが実装されているように扱います。

▼ 表 13.1: RISC-V の CPU がとれる構成

存在する特権レベル	実装する章
M-mode	第 13 章「M-mode の実装 (1. CSR の実装)」
M-mode、U-mode	第 15 章「U-mode の実装」
M-mode、S-mode、U-mode	第 16 章「S-mode の実装 (1. CSR の実装)」

CPU がリセット (起動) したときの特権レベルは M-mode です。現在の特権レベルを保持するレジスタを csrunit モジュールに作成します (リスト 13.2、リスト 13.3)。

#### ▼ リスト 13.2: 現在の特権レベルを示すレジスタの定義 (csrunit.veryl)

```

1 var mode: PrivMode;
```

#### ▼ リスト 13.3: レジスタを M-mode でリセットする (csrunit.veryl)

```

1 always_ff {
2     if_reset {
3         mode    = PrivMode::M;
```

本書で実装する M-mode の CSR のアドレスをすべて定義します (リスト 13.4)。本章ではこの中の一部の CSR を実装し、新しく実装する機能で使うタイミングで他の CSR を解説、実装します

#### ▼ リスト 13.4: CSR のアドレスを定義する (eei.veryl)

```

1 enum CsrAddr: logic<12> {
2     // Machine Information Registers
3     MIMPID = 12'hf13,
4     MHARTID = 12'hf14,
5     // Machine Trap Setup
```

```

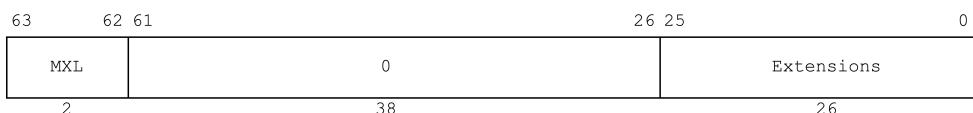
6   MSTATUS = 12'h300,
7   MISA = 12'h301,
8   MEDELEG = 12'h302,
9   MIDELEG = 12'h303,
10  MIE = 12'h304,
11  MTVEC = 12'h305,
12  MCOUNTEREN = 12'h306,
13  // Machine Trap Handling
14  MSCRATCH = 12'h340,
15  MEPC = 12'h341,
16  MCAUSE = 12'h342,
17  MTVAL = 12'h343,
18  MIP = 12'h344,
19  // Machine Counter/Timers
20  MCYCLE = 12'hB00,
21  MINSTRET = 12'hB02,
22  // Custom
23  LED = 12'h800,
24 }

```

### 13.1.3 XLEN の定義

M-mode の CSR の多くは、特権レベルが M-mode のときの XLEN である MXLEN をビット幅として定義されています。S-mode、U-mode のときの XLEN はそれぞれ SXLEN、UXLEN と定義されており、`MXLEN >= SXLEN >= UXLEN` を満たします。仕様上は mstatus レジスタを使用して SXLEN、UXLEN を変更できるように実装できますが、本書では MXLEN、SXLEN、UXLEN が常に 64 (eei パッケージに定義している XLEN) になるように実装します。

## 13.2 misa レジスタ (Machine ISA)



▲図 13.1: misa レジスタ

misa レジスタは、ハードウェアスレッドがサポートする ISA を表す MXLEN ビットのレジスタです。MXL フィールドには MXLEN を表す数値 (表 13.2) が格納されています。Extensions フィールドは下位ビットからそれぞれアルファベットの A、B、C と対応していて、それぞれのビットはそのアルファベットが表す拡張 (例えば A 拡張なら A ビット、C 拡張なら C) が実装されているなら 1 に設定されています。仕様上は Extensions フィールドを書き換えられるように実装できますが、本書では書き換えられないようにします。

misa レジスタを作成し、読み込めるようにします (リスト 13.5、リスト 13.6)。CPU

▼表 13.2: XLEN と数値の対応

XLEN	数値
32	1
64	2
128	3

は RV64IMAC なので MXL フィールドに 64 を表す 2 を設定し、Extensions フィールドの M 拡張 (M)、基本整数命令セット (I)、C 拡張 (C)、A 拡張 (A) のビットを 1 しています。

▼リスト 13.5: misa レジスタの定義 (csrunit.veryl)

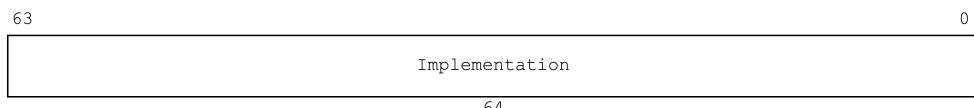
```
1 let misa : UIntX = {2'd2, 1'b0 repeat XLEN - 28, 26'b0000000000000001000100000101}; // M, I,>
> C, A
```

▼リスト 13.6: misa レジスタを読めるようにする (csrunit.veryl)

```
1 rdata = case csr_addr {
2     CsrAddr::MISA : misa,
```

これ以降、A という CSR の B フィールド、ビットのことを A.B と表記することができます。

### 13.3 mimpid レジスタ (Machine Implementation ID)



▲図 13.2: mimpid レジスタ

mimpid レジスタは、プロセッサ実装のバージョンを表す値を格納している MXLEN ビットのレジスタです。値が 0 のときは、mimpid レジスタが実装されていないことを示します。

他にもプロセッサの実装の情報を表すレジスタ (mvendorid<sup>\*2</sup>、marchid<sup>\*3</sup>) がありますが、本書では実装しません。

せっかくなので、適当な値を設定しましょう。eei パッケージに ID を定義して、読み込めるようにします (リスト 13.7、リスト 13.8)。

▼リスト 13.7: ID を適当な値で定義する (eei.veryl)

```
1 // Machine Implementation ID
2 const MACHINE_IMPLEMENTATION_ID: UIntX = 1;
```

<sup>\*2</sup> 製造業者の ID(JEDEC ID) を格納します

<sup>\*3</sup> マイクロアーキテクチャの種類を示す ID を格納します

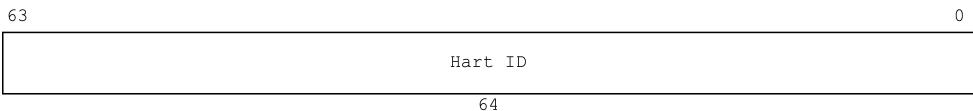
## ▼リスト 13.8: mipmid レジスタを読めるようにする (csrunit.veryl)

```

1     rdata = case csr_addr {
2         CsrAddr::MISA : misa,
3         CsrAddr::MIMPID: MACHINE_IMPLEMENTATION_ID,

```

## 13.4 mhartid レジスタ (Hart ID)



▲図 13.3: mhartid レジスタ

mhartid レジスタは、今実行しているハードウェアスレッド (hart) の ID を格納している MXLEN ビットのレジスタです。複数のプロセッサ、ハードウェアスレッドが存在するときに、それぞれを区別するために使用します。ID はどんな値でも良いですが、環境内に ID が 0 のハードウェアスレッドが 1 つ存在する必要があります。基本編で作る CPU は 1 コア 1 ハードウェアスレッドであるため mhartid レジスタに 0 を設定します。

mhart レジスタを作成し、読み込めるようにします ( リスト 13.9、リスト 13.10 )。

## ▼リスト 13.9: mhartid レジスタの定義 (csrunit.veryl)

```

1 let mhartid: UIntX = 0;

```

## ▼リスト 13.10: mhartid レジスタを読めるようにする (csrunit.veryl)

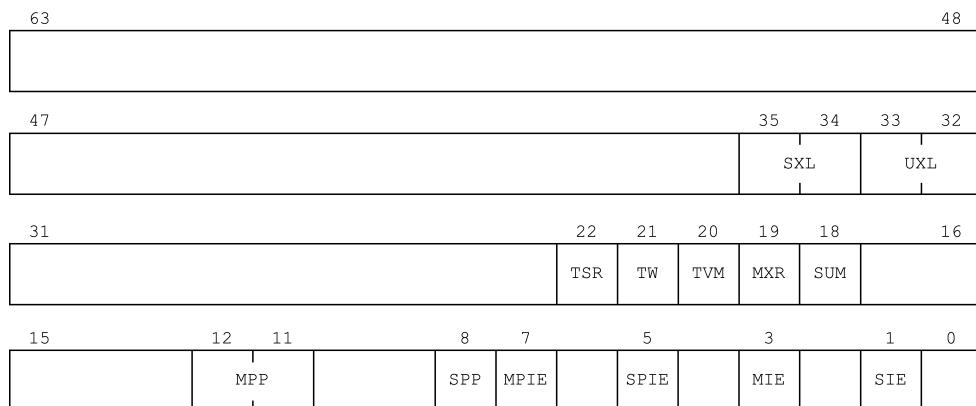
```

1     rdata = case csr_addr {
2         CsrAddr::MISA : misa,
3         CsrAddr::MIMPID : MACHINE_IMPLEMENTATION_ID,
4         CsrAddr::MHARTID: mhartid,

```

## 13.5 mstatus レジスタ (Machine Status)

mstatus レジスタは、拡張の設定やトラップ、状態などを管理する MXLEN ビットのレジスタです。基本編では図 13.4 に示しているフィールドを、そのフィールドが必要になったときに実装します。とりあえず今のところは読み込みだけできるようにします ( リスト 13.11、リスト 13.12、リスト 13.13、リスト 13.14、リスト 13.15、リスト 13.16 )。



▲図 13.4: mstatus レジスタ

## ▼リスト 13.11: 書き込みマスクの定義 (csrunit.veryl)

```
1 const MSTATUS_WMASK: UIntX = 'h0000_0000_0000_0000 as UIntX;
```

## ▼リスト 13.12: 書き込みマスクを設定する (csrunit.veryl)

```
1 wmask = case csr_addr {
2     CsrAddr::MSTATUS: MSTATUS_WMASK,
```

## ▼リスト 13.13: mstatus レジスタの定義 (csrunit.veryl)

```
1 var mstatus: UIntX;
```

## ▼リスト 13.14: mstatus レジスタを読めるようにする (csrunit.veryl)

```
1 rdata = case csr_addr {
2     CsrAddr::MISA    : misa,
3     CsrAddr::MIMPID : MACHINE_IMPLEMENTATION_ID,
4     CsrAddr::MHARTID: mhartid,
5     CsrAddr::MSTATUS: mstatus,
```

## ▼リスト 13.15: mstatus レジスタのリセット (csrunit.veryl)

```
1 always_ff {
2     if_reset {
3         mode    = PrivMode::M;
4         mstatus = 0;
```

## ▼リスト 13.16: mstatus レジスタの書き込み (csrunit.veryl)

```
1 if is_wsc {
2     case csr_addr {
3         CsrAddr::MSTATUS: mstatus = wdata;
4         CsrAddr::MTVEC   : mtvec   = wdata;
```

## 13.6 ハードウェアパフォーマンスマニタ

RISC-V には、ハードウェアの性能評価指標を得るために mcycle と minstret、それぞれ 29 個の mhpmccounter、mhpmevent レジスタが定義されています。それぞれ次の値を得るために利用できます。

### mcycle レジスタ (64 ビット)

ハードウェアスレッドが起動 (リセット) されてから経過したサイクル数

### minstret レジスタ (64 ビット)

ハードウェアスレッドがリタイア (実行完了) した命令数

### mhpmccounter、mhpmevent レジスタ (64 ビット)

mhpmevent レジスタで選択された指標が mhpmccounter レジスタに反映されます。

基本編では mcycle、minstret レジスタを実装します。mhpmccounter、mhpmevent レジスタは表示するような指標がないため実装しません。また、mcountinhibit レジスタを使うとカウントを停止するかを制御できますが、これも実装しません。

### 13.6.1 mcycle レジスタ

mcycle レジスタを定義して読み込めるようにします。(リスト 13.17、リスト 13.18)。

#### ▼ リスト 13.17: mcycle レジスタの定義 (csrunit.veryl)

```
1 var mcycle : UInt64;
```

#### ▼ リスト 13.18: rdata の割り当てで、mcycle レジスタを読めるようにする (csrunit.veryl)

```
1 CsrAddr::MCYCLE : mcycle,
```

always\_ff ブロックで、クロックごとに値を更新します(リスト 13.19)。

#### ▼ リスト 13.19: mcycle レジスタのリセットとインクリメント (csrunit.veryl)

```
1 always_ff {
2     if_reset {
3         mode      = PrivMode::M;
4         mstatus   = 0;
5         mtvec    = 0;
6         mcycle  = 0;
7         mepc    = 0;
8         mcause   = 0;
9         mtval    = 0;
10        led     = 0;
11    } else {
12        mcycle += 1;
13    }
14 }
```

### 13.6.2 minstret レジスタ

core モジュールで instret レジスタを作成し、トラップが発生していない命令が WB ステージに到達した場合にインクリメントします（リスト 13.20、リスト 13.21）。

#### ▼リスト 13.20: minstret レジスタの定義 (core.veryl)

```
1 var minstret      : UInt64;
```

#### ▼リスト 13.21: minstret レジスタのインクリメント (core.veryl)

```
1 always_ff {
2     if_reset {
3         minstret = 0;
4     } else {
5         if wbq_rvalid && wbq_rready && !wbq_rdata.raise_trap {
6             minstret += 1;
7         }
8     }
9 }
```

`minstret` の値を csruni モジュールに渡し、読み込めるようにします（リスト 13.22、リスト 13.23、リスト 13.24）。

#### ▼リスト 13.22: csruni モジュールのポートに minstret を追加する (csruni.veryl)

```
1 minstret : input UInt64 ,
```

#### ▼リスト 13.23: csruni モジュールのインスタンスに minstret レジスタを渡す (core.veryl)

```
1 minstret ,
```

#### ▼リスト 13.24: minstret レジスタを読めるようにする (csruni.veryl)

```
1 CsrAddr::MCYCLE  : mcycle,
2 CsrAddr::MINSTRET: minstret,
3 CsrAddr::MEPC    : mepc,
```

csruni モジュールは MRET 命令でも `raise_trap` フラグを立てるため、このままでは MRET 命令で `minstret` がインクリメントされません。そのため、トラップから戻る命令であることを示すフラグを csruni モジュールに作成し、正しくインクリメントされるようにします（リスト 13.25、リスト 13.26、リスト 13.27、リスト 13.28）。

#### ▼リスト 13.25: csruni モジュールのポートに trap\_return を追加する (csruni.veryl)

```
1 trap_return: output logic ,
```

#### ▼リスト 13.26: MRET 命令の時に trap\_return を 1 にする (csruni.veryl)

```
1 // Trap Return
2 assign trap_return = valid && is_mret && !raise_expt;
3
```

```
4 // Trap
5 assign raise_trap = raise_expt || trap_return;
```

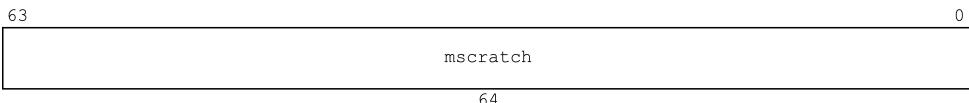
▼リスト 13.27: csrunit モジュールのインスタンスから trap\_return を受け取る (core.veryl)

```
1 trap_return: csru_trap_return ,
```

▼リスト 13.28: MRET 命令なら raise\_trap フラグを立てないようにする (core.veryl)

```
1 wbq_wdata.raise_trap = csru_raise_trap && !csru_trap_return;
```

## 13.7 mscratch レジスタ (Machine Scratch)



▲図 13.5: mscratch レジスタ

mscratch レジスタは、M-mode のときに自由に読み書きできる MXLEN ビットのレジスタです。

mscratch レジスタの典型的な用途はコンテキストスイッチです。コンテキストスイッチとは、実行しているアプリケーション A を別のアプリケーション B に切り替えることを指します。多くの場合、コンテキストスイッチはトラップによって開始しますが、A の実行途中の状態 (レジスタの値) を保存しないと A を実行再開できなくなります。そのため、コンテキストスイッチが始まると、つまりトラップが発生したときにレジスタの値をメモリに保存する必要があります。しかし、ストア命令はアドレスの指定にレジスタの値を使うため、アドレスの指定のために少なくとも 1 つのレジスタの値を犠牲にしなければならず、すべてのレジスタの値を完全に保存できません<sup>\*4</sup>。

この問題を回避するために、一時的な値の保存場所として mscratch レジスタが使用されます。事前に mscratch レジスタにメモリアドレス (やメモリアドレスを得るために情報) を格納しておき、CSRRW 命令で mscratch レジスタの値とレジスタの値を交換することで任意の場所にレジスタの値を保存できます。

mscratch レジスタを定義し、自由に読み書きできるようにします (リスト 13.29、リスト 13.30、リスト 13.31、リスト 13.32、リスト 13.33、リスト 13.34)。

▼リスト 13.29: mscratch レジスタの定義 (csrunit.veryl)

```
1 var mcycle : UInt64;
2 var mscratch: UIntX ;
3 var mepc : UIntX ;
```

<sup>\*4</sup> x0 と即値を使うとアドレス 0 付近にすべてのレジスタの値を保存できますが、一般的な方法ではありません

## ▼リスト 13.30: mscratch レジスタを 0 でリセットする (csrunit.veryl)

```
1    mtvec    = 0;  
2    mscratch = 0;  
3    mcycle   = 0;
```

## ▼リスト 13.31: mscratch レジスタを読めるようにする (csrunit.veryl)

```
1    CsrAddr::MINSTRET: minstret,  
2    CsrAddr::MSCRATCH: mscratch,  
3    CsrAddr::MEPC     : mepc,
```

## ▼リスト 13.32: 書き込みマスクの定義 (csrunit.veryl)

```
1    const MTVEC_WMASK    : UIntX = 'hffff_ffff_ffff_ffffc;  
2    const MSCRATCH_WMASK: UIntX = 'hffff_ffff_ffff_fffff;  
3    const MEPC_WMASK     : UIntX = 'hffff_ffff_ffff_ffffe;
```

## ▼リスト 13.33: 書き込みマスクを wmask に割り当てる (csrunit.veryl)

```
1    CsrAddr::MTVEC     : MTVEC_WMASK,  
2    CsrAddr::MSCRATCH: MSCRATCH_WMASK,  
3    CsrAddr::MEPC     : MEPC_WMASK,
```

## ▼リスト 13.34: mscratch レジスタの書き込み (csrunit.veryl)

```
1    CsrAddr::MTVEC     : mtvec    = wdata;  
2    CsrAddr::MSCRATCH: mscratch = wdata;  
3    CsrAddr::MEPC     : mepc     = wdata;
```

## 第 14 章

# M-mode の実装 (2. 割り込みの実装)

### 14.1 概要

#### 14.1.1 割り込みとは何か？

アプリケーションを記述するとき、キーボードやマウスの入力、時間の経過のようなイベントに起因して何らかのプログラムを実行したいことがあります。例えばキーボードから入力を得たいとき、ポーリング (Polling)、または割り込み (Interrupt) という手法が利用されます。

ポーリングとは、定期的に問い合わせを行う方式のことです。例えばキーボード入力の場合、定期的にキーボードデバイスにアクセスして入力があるかどうかを確かめます。1秒ごとに入力の有無を確認する場合、キーボードの入力から検知までに最大1秒の遅延が発生します。確認頻度をあげると遅延を減らせますが、長時間キーボード入力が無い場合、入力の有無の確認頻度が上がる分だけ何も入力が無いデバイスに対する確認処理が実行されることになります。この問題は、CPU からデバイスに問い合わせをする方式では解決できません。

入力の理想的な確認タイミングは入力が確認できるようになってすぐであるため、入力があったタイミングでデバイス側から CPU にイベントを通知すればいいです。これを実現するのが割り込みです。

割り込みとは、何らかのイベントの通知によって実行中のプログラムを中断し、通知内容を処理する方式のことです。割り込みを使うと、ポーリングのように無駄にデバイスにアクセスをすることなく、入力の処理が必要な時にだけ実行できます。

#### 14.1.2 RISC-V の割り込み

RISC-V では割り込み機能が CSR によって提供されます。割り込みが発生するとトラップが発生します。割り込みを発生させるようなイベントは外部割り込み、ソフトウェア割り込み、タイマ割り込みの 3 つに大別されます。

### 外部割り込み (External Interrupt)

コア外部のデバイスによって発生する割り込み。複数の外部デバイスの割り込みは割り込みコントローラ (第 18 章「PLIC の実装」) などによって調停 (制御) されます。

### ソフトウェア割り込み (Software Interrupt)

CPU で動くソフトウェアが発生させる割り込み。CSR、もしくはメモリにマップされたレジスタ値の変更によって発生します。

### タイマ割り込み (Timer Interrupt)

タイマ回路 (デバイス) によって引き起こされる割り込み。タイマの設定と時間経過によって発生します。

M-mode だけが実装された RISC-V の CPU では、次のような順序で割り込みが提供されます。他に実装されている特権レベルがある場合については「15.9 割り込み条件の変更」(p.301)、「16.4 トランプの委譲」(p.306) で解説します。

1. 割り込みを発生させるようなイベントがデバイスで発生する
2. 割り込み原因に対応した mip レジスタのビットが `0` から `1` になる
3. 割り込み原因に対応した mie レジスタのビットが `1` であることを確認する (`0` なら割り込みは発生しない)
4. mstatus.MIE が `1` であることを確認する (`0` なら割り込みは発生しない)
5. (割り込み (トランプ) 開始)
6. mstatus.MPIE に mstatus.MIE を格納する
7. mstatus.MIE に `0` を格納する
8. mtvec レジスタの値にジャンプする

mip(Machine Interrupt Pending) レジスタは、割り込みを発生させるようなイベントが発生したことを探知する MXLEN ビットの CSR です。mie(Machine Interrupt Enable) レジスタは割り込みを許可するかを原因ごとに制御する MXLEN ビットの CSR です。mstatus.MIE はすべての割り込みを許可するかどうかを制御する 1 ビットのフィールドです。mie と mstatus.MIE のことを割り込みイネーブル (許可) レジスタと呼び、特に mstatus.MIE のようなすべての割り込みを制御するビットのことをグローバル割り込みイネーブルビットと呼びます。

割り込みの発生時に mstatus.MIE を `0` にすることで、割り込みの処理中に割り込みが発生することを防いでいます。また、トランプから戻る (MRET 命令を実行する) ときは、mstatus.MPIE の値を mstatus.MIE に書き戻すことで割り込みの許可状態を戻します。

#### 14.1.3 割り込みの優先順位

RISC-V には外部割り込み、ソフトウェア割り込み、タイマ割り込みがそれぞれ M-mode、S-mode 向けに用意されています。それぞれの割り込みには表 14.1 のような優先順位が定義されていて、複数の割り込みを発生させられるときは優先順位が高い割り込みを発生させます。

▼表 14.1: RISC-V の割り込みの優先順位

cause	説明	優先順位
11	Machine external interrupt	高い
3	Machine software Interrupt	
7	Machine timer interrupt	
9	Supervisor external interrupt	
1	Supervisor software interrupt	
5	Supervisor timer interrupt	低い

#### 14.1.4 割り込みの原因 (cause)

それぞれの割り込みには原因を区別するための値 (cause) が割り当てられています。割り込みの cause の MSB は 1 です。

CsrCause 型に割り込みの cause を追加します (リスト 14.1)。

▼リスト 14.1: 割り込みの原因の定義 (eei.veryl)

```

1 enum CsrCause: UIntX {
2     INSTRUCTION_ADDRESS_MISALIGNED = 0,
3     ILLEGAL_INSTRUCTION = 2,
4     BREAKPOINT = 3,
5     LOAD_ADDRESS_MISALIGNED = 4,
6     STORE_AMO_ADDRESS_MISALIGNED = 6,
7     ENVIRONMENT_CALL_FROM_M_MODE = 11,
8     SUPERVISOR_SOFTWARE_INTERRUPT = 'h8000_0000_0000_0001,
9     MACHINE_SOFTWARE_INTERRUPT = 'h8000_0000_0000_0003,
10    SUPERVISOR_TIMER_INTERRUPT = 'h8000_0000_0000_0005,
11    MACHINE_TIMER_INTERRUPT = 'h8000_0000_0000_0007,
12    SUPERVISOR_EXTERNAL_INTERRUPT = 'h8000_0000_0000_0009,
13    MACHINE_EXTERNAL_INTERRUPT = 'h8000_0000_0000_000b,
14 }
```

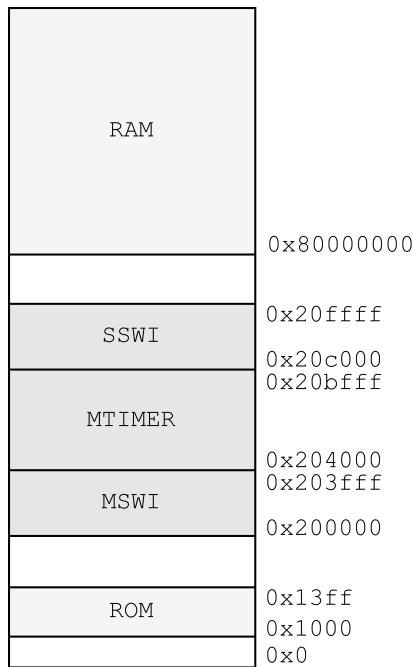
#### 14.1.5 ACLINT (Advanced Core Local Interruptor)

RISC-V にはソフトウェア割り込みとタイマ割り込みを実現するデバイスの仕様である ACLINT が用意されています。ACLINT は、SiFive 社が開発した CLINT(Core-Local Interruptor) デバイスが基になった仕様です。

ACLINT には MTIMER、MSWI、SSWI の 3 つのデバイスが定義されています。MTIMER デバイスはタイマ割り込み、MSWI と SSWI デバイスはソフトウェア割り込み向けのデバイスで、それぞれ mip レジスタの MTIP、MSIP、SSIP ビットに状態を通知します。

本書では ACLINT を図図 14.1 のようなメモリマップで実装します。本章では MTIMER、MSWI デバイスを実装し、「16.5 ソフトウェア割り込みの実装 (SSWI)」(p.316) で SSWI デバイスを実装します。デバイスの具体的な仕様については後で解説します。

メモリマップ用の定数を eei パッケージに記述してください (リスト 14.2)。



▲図 14.1: ACLINT のメモリマップ

## ▼リスト 14.2: メモリマップ用の定数の定義 (eei.veryl)

```

1 // ACLINT
2 const MMAP_ACLINT_BEGIN : Addr = 'h200_0000 as Addr;
3 const MMAP_ACLINT_MSIP : Addr = 0;
4 const MMAP_ACLINT_MTIMECMP: Addr = 'h4000 as Addr;
5 const MMAP_ACLINT_MTIME : Addr = 'h7ff8 as Addr;
6 const MMAP_ACLINT_SETSSIP : Addr = 'h8000 as Addr;
7 const MMAP_ACLINT_END : Addr = MMAP_ACLINT_BEGIN + 'hbfff as Addr;
```

## 14.2 ACLINT モジュールの作成

本章では、ACLINT のデバイスを aclint\_memory モジュールに実装します。aclint\_memory モジュールは割り込みを起こすために csrunit モジュールと接続します。

### 14.2.1 インターフェースを作成する

まず、ACLINT のデバイスと csrunit モジュールを接続するためのインターフェースを作成します。src/aclint\_if.veryl を作成し、次のように記述します（リスト 14.3）。インターフェースの中身は各デバイスの実装時に実装します。

## ▼リスト 14.3: aclint\_if.veryl

```

1 interface aclint_if {
2     modport master {
3         // TODO
4     }
5     modport slave {
6         ..converse(master)
7     }
8 }
```

**14.2.2 aclint\_memory モジュールを作成する**

ACLINT のデバイスを実装するモジュールを作成します。 `src/aclint_memory.veryl` を作成し、次のように記述します（リスト 14.4）。まだどのレジスタも実装していません。

## ▼リスト 14.4: aclint\_memory.veryl

```

1 import eei::*;

2
3 module aclint_memory (
4     clk      : input   clock      ,
5     rst      : input   reset      ,
6     membus: modport Membus::slave ,
7     aclint: modport aclint_if::master,
8 ) {
9     assign membus.ready = 1;
10    always_ff {
11        if_reset {
12            membus.rvalid = 0;
13            membus.rdata  = 0;
14        } else {
15            membus.rvalid = membus.valid;
16        }
17    }
18 }
```

**14.2.3 mmio\_controller モジュールに ACLINT を追加する**

mmio\_controller モジュールに ACLINT デバイスを追加して、aclint\_memory モジュールにアクセスできるようにします。

`Device` 型に `ACLINT` を追加して、ACLINT のデバイスをアドレスにマップします（リスト 14.5、リスト 14.6）。

## ▼リスト 14.5: Device 型に ACLINT を追加する (mmio\_controller.veryl)

```

1 enum Device {
2     UNKNOWN,
3     RAM,
4     ROM,
5     DEBUG,
```

```
6     ACLINT,
7 }
```

## ▼ リスト 14.6: get\_device 関数で ACLINT の範囲を定義する (mmio\_controller.veryl)

```
1 if MMAP_ACLINT_BEGIN <= addr && addr <= MMAP_ACLINT_END {
2     return Device:::ACLINT;
3 }
```

ACLINT とのインターフェースを追加し、reset\_all\_device\_masters 関数にインターフェースをリセットするコードを追加します（リスト 14.7、リスト 14.8）。

## ▼ リスト 14.7: ポートに ACLINT のインターフェースを追加する (mmio\_controller.veryl)

```
1 module mmio_controller (
2     clk          : input    clock      ,
3     rst          : input    reset      ,
4     DBG_ADDR    : input    Addr       ,
5     req_core    : modport Membus::slave ,
6     ram_membus  : modport Membus::master,
7     rom_membus  : modport Membus::master,
8     dbg_membus  : modport Membus::master,
9     aclint_membus: modport Membus::master,
10 ) {
```

## ▼ リスト 14.8: インターフェースの要求部分をリセットする (mmio\_controller.veryl)

```
1 function reset_all_device_masters () {
2     reset_membus_master(ram_membus);
3     reset_membus_master(rom_membus);
4     reset_membus_master(dbg_membus);
5     reset_membus_master(aclint_membus);
6 }
```

`ready`、`rvalid` を取得する関数に ACLINT を登録します（リスト 14.9、リスト 14.10）。

## ▼ リスト 14.9: get\_device\_ready 関数に ACLINT の ready を追加 (mmio\_controller.veryl)

```
1 Device:::ACLINT: return aclint_membus.ready;
```

## ▼ リスト 14.10: get\_device\_rvalid 関数に ACLINT の rvalid を追加 (mmio\_controller.veryl)

```
1 Device:::ACLINT: return aclint_membus.rvalid;
```

ACLINT の `rvalid`、`rdata` を `req_core` に割り当てます（リスト 14.11）。

▼ リスト 14.11: ACLINT へのアクセス結果を `req` に割り当てる (mmio\_controller.veryl)

```
1 Device:::ACLINT: req <=> aclint_membus;
```

ACLINT のインターフェースに要求を割り当てます（リスト 14.12）。

## ▼リスト 14.12: ACLINT に req を割り当ててアクセス要求する (mmio\_controller.veryl)

```

1 Device::ACLINT: {
2     aclint_membus      <=> req;
3     aclint_membus.addr == MMAP_ACLINT_BEGIN;
4 }
```

**14.2.4 ACLINT と mmio\_controller、csrunit モジュールを接続する**

aclint\_if インターフェース ( `aclint_core_bus` )、aclint\_memory モジュールと mmio\_controller モジュールを接続するインターフェース ( `aclint_membus` ) をインスタンス化します ( リスト 14.13、リスト 14.14 )。

## ▼リスト 14.13: aclint\_if インターフェースのインスタンス化 (top.veryl)

```

1 inst aclint_core_bus: aclint_if;
```

## ▼リスト 14.14: mmio\_controller モジュールと接続するインターフェースのインスタンス化 (top.veryl)

```

1 inst aclint_membus : Membus;
```

aclint\_memory モジュールをインスタンス化し、mmio\_controller モジュールと接続します ( リスト 14.15、リスト 14.16 )。

## ▼リスト 14.15: aclint\_memory モジュールをインスタンス化する (top.veryl)

```

1 inst aclintm: aclint_memory (
2     clk           ,
3     rst           ,
4     membust: aclint_membus ,
5     aclint: aclint_core_bus,
6 );
```

## ▼リスト 14.16: mmio\_controller モジュールと接続する (top.veryl)

```

1 inst mmioc: mmio_controller (
2     clk           ,
3     rst           ,
4     DBG_ADDR     : MMAP_DBG_ADDR  ,
5     req_core     : mmio_membus   ,
6     ram_membus   : mmio_ram_membus,
7     rom_membus   : mmio_rom_membus,
8     dbg_membus   ,
9     aclint_membus          ,
10 );
```

core、csrunit モジュールに aclint\_if ポートを追加し、csrunit モジュールと aclint\_memory モジュールを接続します ( リスト 14.17、リスト 14.18、リスト 14.19、リスト 14.20 )。

## ▼リスト 14.17: core モジュールに ACLINT のデバイスとのインターフェースを追加する (core.veryl)

```

1 module core (
2     clk      : input  clock          ,
3     rst      : input  reset          ,
4     i_membus: modport core_inst_if::master,
5     d_membus: modport core_data_if::master,
6     led      : output UIntX        ,
7     aclint   : modport aclint_if::slave  ,
8 ) {

```

▼リスト 14.18: core モジュールに aclint\_if インターフェースを接続する (top.veryl)

```

1 inst c: core (
2     clk          ,
3     rst          ,
4     i_membus: i_membus_core  ,
5     d_membus: d_membus_core  ,
6     led          ,
7     aclint   : aclint_core_bus,
8 );

```

▼リスト 14.19: csrunit モジュール ACLINT デバイスとのインターフェースを追加する (csrunit.veryl)

```

1 minstret  : input  UInt64          ,
2 led       : output UIntX          ,
3 aclint    : modport aclint_if::slave  ,
4 ) {

```

▼リスト 14.20: csrunit モジュールのインスタンスにインターフェースを接続する (core.veryl)

```

1     minstret          ,
2     led              ,
3     aclint           ,
4 );

```

## 14.3 ソフトウェア割り込みの実装 (MSWI)

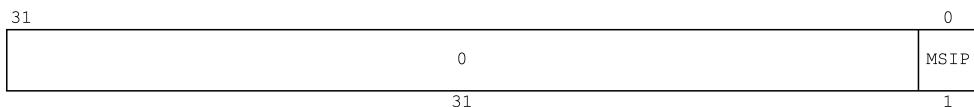
MSWI デバイスはソフトウェア割り込み (machine software interrupt) を提供するためのデバイスです。MSWI デバイスにはハードウェアスレッド毎に 4 バイトの MSIP レジスタが用意されています (表 14.2)。MSIP レジスタの上位 31 ビットは読み込み専用の **0** であり、最下位ビットのみ変更できます。各 MSIP レジスタは、それに対応するハードウェアスレッドの mip.MSIP と接続されています。

仕様上は mhartid と MSIP の後の数字 (hartID) が一致する必要はありませんが、本書では mhartid と hartID が同じになるように実装します。他の ACLINT のデバイスも同様に実装します。

▼表 14.2: MSWI デバイスのメモリマップ

オフセット	レジスタ
0000	MSIP0
0004	MSIP1
0008	MSIP2
..	..
3ff8	MSIP4094
3ffc	予約済み

### 14.3.1 MSIP レジスタを実装する



▲図 14.2: MSIP レジスタ

ACLINT モジュールに MSIP レジスタを実装します(図 14.2)。今のところ CPU には mhartid が **0** のハードウェアスレッドしか存在しないため、MSIP0 のみ実装します。

aclint\_if インターフェースに **msip** を追加します(リスト 14.21)。

▼リスト 14.21: msip ビットをインターフェースに追加する (aclint\_if.vrtl)

```

1 interface aclint_if {
2     var msip: logic;
3     modport master {
4         msip: output,
5     }
6     modport slave {
7         ..converse(master)
8     }
9 }
```

aclint\_memory モジュールに **msip0** レジスタを作成し、読み書きできるようにします(リスト 14.22、リスト 14.23、リスト 14.24)。

▼リスト 14.22: msip0 レジスタの定義 (aclint\_memory.vrtl)

```
1 var msip0: logic;
```

▼リスト 14.23: msip0 レジスタを 0 でリセットする (aclint\_memory.vrtl)

```

1 always_ff {
2     if_reset {
3         membus.rvalid = 0;
4         membus.rdata  = 0;
5         msip0        = 0;
```

## ▼リスト 14.24: msip0 レジスタの書き込み、読み込み (aclint\_memory.veryl)

```

1 if membus.valid {
2     let addr: Addr = {membus.addr[XLEN - 1:3], 3'b0};
3     if membus.wen {
4         let M: logic<MEMBUS_DATA_WIDTH> = membus.wmask_expand();
5         let D: logic<MEMBUS_DATA_WIDTH> = membus.wdata & M;
6         case addr {
7             MMAP_ACLINT_MSIP: msip0 = D[0] | msip0 & ~M[0];
8             default           : {}
9         }
10    } else {
11        membus.rdata = case addr {
12            MMAP_ACLINT_MSIP: {63'b0, msip0},
13            default          : 0,
14        };
15    }
16 }

```

`msip0` レジスタとインターフェースの `msip` を接続します (リスト 14.25)。

▼リスト 14.25: インターフェースの `msip` と `msip0` レジスタを接続する (aclint\_memory.veryl)

```

1 always_comb {
2     aclint.msip = msip0;
3 }

```

### 14.3.2 mip、mie レジスタを実装する

63	12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIP	0	SEIP	0	MTIP	0	STIP	0	MSIP	0	SSIP	0	
52	1	1	1	1	1	1	1	1	1	1	1	1	1

▲図 14.3: mip レジスタ

63	12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIE	0	SEIE	0	MTIE	0	STIE	0	MSIE	0	SSIE	0	
52	1	1	1	1	1	1	1	1	1	1	1	1	1

▲図 14.4: mie レジスタ

mip レジスタの MSIP ビット、mie レジスタの MSIE ビットを実装します。mie.MSIE は MSIP ビットによる割り込み待機を許可するかを制御するビットです。mip.MSIP と mie.MSIE は同じ位置のビットに配置されています。mip.MSIP に書き込むことはできません。

csrunut モジュールに mie レジスタを作成します (リスト 14.26、リスト 14.27)。

## ▼リスト 14.26: mie レジスタの定義 (csrunit.veryl)

```
1 var mie      : UIntX ;
```

## ▼リスト 14.27: mie レジスタを 0 でリセットする (csrunit.veryl)

```
1 if_reset {
2     mode      = PrivMode::M;
3     mstatus   = 0;
4     mtvec    = 0;
5     mie       = 0;
6     mscratch = 0;
```

mip レジスタを作成します。MSIP ビットを MSWI デバイスの MSIP0 レジスタと接続し、それ以外のビットは 0 に設定します（リスト 14.28）。

## ▼リスト 14.28: mip レジスタの定義 (csrunit.veryl)

```
1 let mip: UIntX = {
2     1'b0 repeat XLEN - 12, // 0
3     1'b0, // MEIP
4     1'b0, // 0
5     1'b0, // SEIP
6     1'b0, // 0
7     1'b0, // MTIP
8     1'b0, // 0
9     1'b0, // STIP
10    1'b0, // 0
11    aclint.msip, // MSIP
12    1'b0, // 0
13    1'b0, // SSIP
14    1'b0, // 0
15};
```

mie、mip レジスタの値を読み込めるようにします（リスト 14.29）。

## ▼リスト 14.29: rdata に mip、mie レジスタの値を割り当てる (csrunit.veryl)

```
1 CsrAddr::MTVEC  : mtvec,
2 CsrAddr::MIP    : mip,
3 CsrAddr::MIE    : mie,
4 CsrAddr::MCYCLE : mcycle,
```

mie レジスタの書き込みマスクを設定して、MSIE ビットを書き込めるようにします（リスト 14.30、リスト 14.31、リスト 14.32）。あとで MTIME デバイスを実装するときに MTIE ビットを使うため、ここで MTIE ビットも書き込めるようにしておきます。

## ▼リスト 14.30: mie レジスタの書き込みマスクの定義 (csrunit.veryl)

```
1 const MIE_WMASK : UIntX = 'h0000_0000_0000_0088 as UIntX;
```

## ▼リスト 14.31: wmask に書き込みマスクを設定する (csrunit.veryl)

```

1 CsrAddr::MTVEC    : MTVEC_WMASK,
2 CsrAddr::MIE      : MIE_WMASK,
3 CsrAddr::MSCRATCH: MSCRATCH_WMASK,
```

## ▼リスト 14.32: mie レジスタの書き込み (csrunit.veryl)

```

1 if is_wsc {
2     case csr_addr {
3         CsrAddr::MSTATUS : mstatus = wdata;
4         CsrAddr::MTVEC   : mtvec   = wdata;
5         CsrAddr::MIE     : mie     = wdata;
6         CsrAddr::MSCRATCH: mscratch = wdata;
```

### 14.3.3 mstatus の MIE、MPIE ビットを実装する

mstatus.MIE、MPIE を変更できるようにします ( リスト 14.33、リスト 14.34 )。

## ▼リスト 14.33: 書き込みマスクを変更する (csrunit.veryl)

```
1 const MSTATUS_WMASK : UIntX = 'h0000_0000_0000_0088 as UIntX;
```

## ▼リスト 14.34: レジスタの場所を変数に割り当てる (csrunit.veryl)

```

1 // mstatus bits
2 let mstatus_mpie: logic = mstatus[7];
3 let mstatus_mie : logic = mstatus[3];
```

トラップが発生するとき、mstatus.MPIE に mstatus.MIE、mstatus.MIE に 0 を設定します ( リスト 14.35 )。また、MRET 命令で mstatus.MIE に mstatus.MPIE、mstatus.MPIE に 0 を設定します。

## ▼リスト 14.35: トラップ、MRET 命令の動作の実装 (csrunit.veryl)

```

1 if raise_trap {
2     if raise_expt {
3         mepc  = pc;
4         mcause = trap_cause;
5         mtval = expt_value;
6         // save mstatus.mie to mstatus.mpie
7         // and set mstatus.mie = 0
8         mstatus[7] = mstatus[3];
9         mstatus[3] = 0;
10    } else if trap_return {
11        // set mstatus.mie = mstatus.mpie
12        // mstatus.mpie = 0
13        mstatus[3] = mstatus[7];
14        mstatus[7] = 0;
15    }
}
```

これによりトラップで割り込みを無効化して、トラップから戻るときに mstatus.MIE を元に戻す、という動作が実現されます。

### 14.3.4 割り込み処理の実装

必要なレジスタを実装できたので、割り込みを起こす処理を実装します。割り込みは mip、mie の両方のビット、mstatus.MIE ビットが立っているときに発生します。

#### 割り込みのタイミング

割り込みでトラップを発生させると、トラップが発生した時点の命令のアドレスが必要なため、csrunit モジュールに有効な命令が供給されている必要があります。

割り込みが発生したときに csrunit モジュールに供給されていた命令は実行されません。ここで、割り込みを起こすタイミングに注意が必要です。今のところ、CSR の処理は MEM ステージと同時にしているため、例えばストア命令を memunit モジュールで実行している途中に割り込みを発生させてしまうと、ストア命令の結果がメモリに反映されるにもかかわらず、mepc レジスタにストア命令のアドレスを書き込んでしまいます。

それならば、単純に次の命令のアドレスを mepc レジスタに格納するようにすればいいと思うかもしれません、そもそも実行中のストア命令が本来は最終的に例外を発生させるものかもしれません。

本章ではこの問題に対処するために、割り込みは MEM(CSR) ステージに新しく命令が供給されたクロックでしか起こせなくして、トラップが発生するならば memunit モジュールを無効化します。

割り込みを発生させられるかを示すフラグ (`can_intr`) を csrunit モジュールに定義し、`mems_is_new` フラグを割り当てます (リスト 14.36、リスト 14.37)。

#### ▼ リスト 14.36: csrunit モジュールに can\_intr を追加する (csrunit.veryl)

```

1 rs1_data : input UIntX      ,
2 can_intr : input logic      ,
3 rdata     : output UIntX      ,

```

#### ▼ リスト 14.37: mems\_is\_new を can\_intr に割り当てる (core.veryl)

```

1 rs1_data : memq_rdata.rs1_data ,
2 can_intr : mems_is_new        ,
3 rdata     : csru_rdata       ,

```

トラップが発生するときに memunit モジュールを無効にします (リスト 14.38)。今まで EX ステージまでに例外が発生することが分かっていたら無効にしていましたが、csrunit モジュールからトラップが発生するかどうかの情報を直接得るようにします。

#### ▼ リスト 14.38: valid の条件を変更する (core.veryl)

```

1 inst memu: memunit (
2   clk           ,
3   rst           ,
4   valid : mems_valid && !csru_raise_trap,

```

memunit モジュールが無効 (`!valid`) なとき、`state` を `State::Init` にリセットします (リス

ト 14.39)。

▼ リスト 14.39: valid ではないとき、state を Init にリセットする (core.veryl)

```

1 } else {
2     if !valid {
3         state = State::Init;
4     } else {
5         case state {
6             State::Init: if is_new & inst_is_memop(ctrl) {

```

### 割り込みの判定

割り込みを起こすかどうか (`raise_interrupt`)、割り込みの cause (`interrupt_cause`)、トラップベクタ (`interrupt_vector`) を示す変数を作成します (リスト 14.40)。

▼ リスト 14.40: 割り込みを判定する (csrunit.veryl)

```

1 // Interrupt
2 let raise_interrupt : logic = valid && can_intr && mstatus_mie && (mip & mie) != 0;
3 let interrupt_cause : UIntX = CsrCause::MACHINE_SOFTWARE_INTERRUPT;
4 let interrupt_vector: Addr = mtvec;

```

トラップ情報の変数に、割り込みの情報を割り当てます (リスト 14.41)。本書では例外を優先します。

▼ リスト 14.41: トラップを制御する変数に割り込みの値を割り当てる (csrunit.veryl)

```

1 assign raise_trap = raise_expt || raise_interrupt || trap_return;
2 let trap_cause: UIntX = switch {
3     raise_expt      : expt_cause,
4     raise_interrupt: interrupt_cause,
5     default         : 0,
6 };
7 assign trap_vector = switch {
8     raise_expt      : mtvec,
9     raise_interrupt: interrupt_vector,
10    trap_return     : mepc,
11    default         : 0,
12 };

```

割り込みの時に MRET 命令の判定が `0` になるようにします (リスト 14.42)。

▼ リスト 14.42: 割り込みが発生するとき、trap\_return を 0 にする (csrunit.veryl)

```

1 // Trap Return
2 assign trap_return = valid && is_mret && !raise_expt && !raise_interrupt;

```

トラップが発生するとき、例外の場合にのみ mtval レジスタに例外に固有の情報が書き込まれます。本書では例外を優先するので、`raise_expt` が `1` なら mtval レジスタに書き込むようにします (リスト 14.43)。

## ▼ リスト 14.43: 例外が発生したときにのみ mtval レジスタに書き込む (csrunit.veryl)

```

1 if raise_trap {
2     if raise_expt || raise_interrupt {
3         mepc = pc;
4         mcause = trap_cause;
5         if raise_expt {
6             mtval = expt_value;
7     }

```

### 14.3.5 ソフトウェア割り込みをテストする

ソフトウェア割り込みが正しく動くことを確認します。

`test/mswi.c` を作成し、次のように記述します（リスト 14.44）。

## ▼ リスト 14.44: test/mswi.c

```

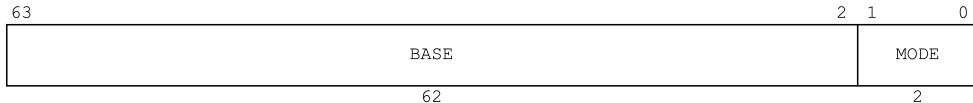
1 #define MSIP0 ((volatile unsigned int *)0x2000000)
2 #define DEBUG_REG ((volatile unsigned long long*)0x40000000)
3 #define MIE_MSIE (1 << 3)
4 #define MSTATUS_MIE (1 << 3)
5
6 void interrupt_handler(void);
7
8 void w_mtvec(unsigned long long x) {
9     asm volatile("csrw mtvec, %0" : : "r" (x));
10 }
11
12 void w_mie(unsigned long long x) {
13     asm volatile("csrw mie, %0" : : "r" (x));
14 }
15
16 void w_mstatus(unsigned long long x) {
17     asm volatile("csrw mstatus, %0" : : "r" (x));
18 }
19
20 void main(void) {
21     w_mtvec((unsigned long long)interrupt_handler);
22     w_mie(MIE_MSIE);
23     w_mstatus(MSTATUS_MIE);
24     *MSIP0 = 1;
25     while (1) *DEBUG_REG = 3; // fail
26 }
27
28 void interrupt_handler(void) {
29     *DEBUG_REG = 1; // success
30 }

```

プログラムでは、mtvec に interrupt\_handler 関数のアドレスを書き込み、mstatus.MIE、mie.MSIE を 1 に設定して割り込みを許可してから MSIP0 レジスタに 1 を書き込んでいます。

プログラムをコンパイルして実行<sup>\*1</sup>すると、ソフトウェア割り込みが発生することで interrupt\_handler にジャンプし、デバッグ用のデバイスに 1 を書き込んで終了することを確認できます。

## 14.4 mtvec の Vectored モードの実装



▲図 14.5: mtvec レジスタ

mtvec レジスタには MODE フィールドがあり、割り込みが発生するときのジャンプ先の決定方法を制御できます(図 14.5)。

MODE が Direct( 2'b00 ) のとき、 `mtvec.BASE << 2` のアドレスにトラップします。Vectored( 2'b01 ) のとき、 `(mtvec.BASE << 2) + 4 * cause` のアドレスにトラップします。ここで cause は割り込みの cause の MSB を除いた値です。例えば machine software interrupt の場合、 `(mtvec.BASE << 2) + 4 * 3` がジャンプ先になります。

例外のトラップベクタは、常に MODE が Direct として計算します。

下位 1 ビットに書き込めるようにすることで、mtvec.MODE に Vectored を書き込めるようにします(リスト 14.45)。

### ▼リスト 14.45: 書き込みマスクを変更する (csrunit.veryl)

```
1 const MTVEC_WMASK : UIntX = 'hffff_ffff_ffff_fffd;
```

割り込みのトラップベクタを MODE と cause に応じて変更します(リスト 14.46)。

### ▼リスト 14.46: 割り込みのトラップベクタを求める (csrunit.veryl)

```
1 let interrupt_vector: Addr = if mtvec[0] == 0 ? {mtvec[msb:2], 2'b0} : // Direct
2   {mtvec[msb:2] + interrupt_cause[msb - 2:0], 2'b0}; // Vectored
```

例外のトラップベクタを、mtvec レジスタの下位 2 ビットを 0 にしたアドレス(Direct)に変更します(リスト 14.47、リスト 14.48)。新しく `expt_vector` を定義し、`trap_vector` に割り当てます。

### ▼リスト 14.47: 例外のトラップベクタ (csrunit.veryl)

```
1 let expt_vector: Addr = {mtvec[msb:2], 2'b0};
```

<sup>\*1</sup> コンパイル、実行方法は「10.6.4 出力をテストする」(p.210) を参考にしてください。

## ▼ リスト 14.48: expt\_vector を trap\_vector に割り当てる (csrunit.veryl)

```

1 assign trap_vector = switch {
2     raise_expt    : expt_vector,
3     raise_interrupt: interrupt_vector,
4     trap_return   : mepc,
5     default        : 0,
6 };

```

## 14.5 タイマ割り込みの実装 (MTIMER)

### 14.5.1 タイマ割り込み

MTIMER デバイスは、タイマ割り込み (machine timer interrupt) を提供するためのデバイスです。MTIMER デバイスには 1 つの 8 バイトの MTIME レジスタ、ハードウェアスレッド毎に 8 バイトの MTIMECMP レジスタが用意されています。本書では MTIMECMP の後ろに MTIME を配置します (表 14.3)。

▼ 表 14.3: 本書の MTIMER デバイスのメモリマップ

オフセット	レジスタ
0000	MTIMECMP0
0008	MTIMECMP1
..	..
7ff0	MTIMECMP4094
7ff8	MTIME

MTIME レジスタは、固定された周波数でのクロックサイクル毎にインクリメントするレジスタです。リセット時に 0 になります。

MTIMER デバイスは、それに対応するハードウェアスレッドの mip.MTIP と接続されており、MTIME が MTIMECMP を上回ったとき mip.MTIP を 1 にします。これにより、指定した時間に割り込みを発生させることができます。

### 14.5.2 MTIME、MTIMECMP レジスタを実装する

ACLINT モジュールに MTIME、MTIMECMP レジスタを実装します。今のところ mhrtid が 0 のハードウェアスレッドしか存在しないため、MTIMECMP0 のみ実装します。

`mtime`、`mtimecmp0` レジスタを作成し、読み書きできるようにします (リスト 14.49、リスト 14.50、リスト 14.51)。`mtime` レジスタはクロック毎にインクリメントします。

## ▼リスト 14.49: mtime、mtimecmp レジスタの定義 (aclint\_memory.veryl)

```

1  var msip0      : logic ;
2  var mtime     : UInt64;
3  var mtimecmp0: UInt64;
```

## ▼リスト 14.50: レジスタを 0 でリセットする (aclint\_memory.veryl)

```

1  always_ff {
2      if_reset {
3          membus.rvalid = 0;
4          membus.rdata  = 0;
5          msip0        = 0;
6          mtime        = 0;
7          mtimecmp0    = 0;
```

## ▼リスト 14.51: mtime、mtimecmp の書き込み、読み込み (aclint\_memory.veryl)

```

1  if membus.wen {
2      let M: logic<MEMBUS_DATA_WIDTH> = membus.wmask_expand();
3      let D: logic<MEMBUS_DATA_WIDTH> = membus.wdata & M;
4      case addr {
5          MMAP_ACLINT_MSIP    : msip0      = D[0] | msip0 & ~M[0];
6          MMAP_ACLINT_MTIME   : mtime      = D | mtime & ~M;
7          MMAP_ACLINT_MTIMECMP: mtimecmp0 = D | mtimecmp0 & ~M;
8          default            : {}
9      }
10 } else {
11     membus.rdata = case addr {
12         MMAP_ACLINT_MSIP    : {63'b0, msip0},
13         MMAP_ACLINT_MTIME   : mtime,
14         MMAP_ACLINT_MTIMECMP: mtimecmp0,
15         default            : 0,
16     };
17 }
```

aclint\_if インターフェースに `mtip` を作成し、タイマ割り込みが発生する条件を設定します（リスト 14.52、リスト 14.53）。

## ▼リスト 14.52: mtip をインターフェースに追加する (aclint\_if.veryl)

```

1  var msip: logic;
2  var mtip: logic;
3  modport master {
4      msip: output,
5      mtip: output,
6  }
```

## ▼リスト 14.53: mtip にタイマ割り込みが発生する条件を設定する (aclint\_memory.veryl)

```

1  always_comb {
2      aclint.msip = msip0;
3      aclint.mtip = mtime >= mtimecmp0;
4  }
```

### 14.5.3 mip.MTIP、割り込み原因を設定する

mip レジスタの MTIP ビットに aclint\_if インターフェースの `mtip` を接続します (リスト 14.54)。

▼ リスト 14.54: mip.MTIP にインターフェースの `mtip` を割り当てる (csruni.vetyl)

```

1 let mip: UIntX = {
2     1'b0 repeat XLEN - 12, // 0, LCOFIP
3     1'b0, // MEIP
4     1'b0, // 0
5     1'b0, // SEIP
6     1'b0, // 0
7     aclint.mtip, // MTIP
8     1'b0, // 0
9     1'b0, // STIP
10    1'b0, // 0
11    aclint.msip, // MSIP
12    1'b0, // 0
13    1'b0, // SSIP
14    1'b0, // 0
15 };

```

割り込み原因を優先順位に応じて設定します。タイマ割り込みはソフトウェア割り込みよりも優先順位が低いため、ソフトウェア割り込みの下で原因を設定します (リスト 14.55)。

▼ リスト 14.55: タイマ割り込みの cause を設定する (csruni.vetyl)

```

1 let interrupt_pending: UIntX = mip & mie;
2 let raise_interrupt : logic = valid && can_intr && mstatus_mie && interrupt_pending != 0;
3 let interrupt_cause : UIntX = switch {
4     interrupt_pending[3]: CsrCause::MACHINE_SOFTWARE_INTERRUPT,
5     interrupt_pending[7]: CsrCause::MACHINE_TIMER_INTERRUPT,
6     default : 0,
7 };
8 let interrupt_vector: Addr = if mtvec[0] == 0 ? {mtvec[msb:2], 2'b0} : // Direct
9     {mtvec[msb:2] + interrupt_cause[msb - 2:0], 2'b0}; // Vectored

```

### 14.5.4 タイマ割り込みをテストする

タイマ割り込みが正しく動くことを確認します。

`test/mtime.c` を作成し、次のように記述します (リスト 14.56)。

▼ リスト 14.56: test/mtime.c

```

1 #define MTIMECMP0 ((volatile unsigned int *)0x2004000)
2 #define MTIME   ((volatile unsigned int *)0x2007ff8)
3 #define DEBUG_REG ((volatile unsigned long long*)0x40000000)
4 #define MIE_MTIE (1 << 7)
5 #define MSTATUS_MIE (1 << 3)
6
7 void interrupt_handler(void);

```

```

8
9 void w_mtvec(unsigned long long x) {
10    asm volatile("csrw mtvec, %0" : : "r" (x));
11 }
12
13 void w_mie(unsigned long long x) {
14    asm volatile("csrw mie, %0" : : "r" (x));
15 }
16
17 void w_mstatus(unsigned long long x) {
18    asm volatile("csrw mstatus, %0" : : "r" (x));
19 }
20
21 void main(void) {
22    w_mtvec((unsigned long long)interrupt_handler);
23    *MTIMECMP0 = *MTIME + 1000000; // この数値は適当に調整する
24    w_mie(MIE_MTIE);
25    w_mstatus(MSTATUS_MIE);
26    while (1);
27    *DEBUG_REG = 3; // fail
28 }
29
30 void interrupt_handler(void) {
31    *DEBUG_REG = 1; // success
32 }
```

プログラムでは、mtvec に interrupt\_handler 関数のアドレスを設定し、mtime に 10000000 を足した値を mtimecmp0 に設定した後、mstatus.MIE、mie.MTIE を 1 に設定して割り込みを許可しています。タイマ割り込みが発生するまで while 文で無限ループします。

プログラムをコンパイルして実行すると、時間経過によって main 関数から interrupt\_handler 関数にトラップしてテストが終了します。mtimecmp0 に設定する値を変えることで、タイマ割り込みが発生するまでの時間が変わることを確認してください。

## 14.6 WFI 命令の実装

WFI 命令は、割り込みが発生するまで CPU をストールさせる命令です。ただし、グローバル割り込みイネーブルビットは考慮せず、ある割り込みの待機 (pending) ビットと許可 (enable) ビットの両方が立っているときに実行を再開します。また、それ以外の自由な理由で実行を再開させてもいいです。WFI 命令で割り込みが発生するとき、WFI 命令の次のアドレスの命令で割り込みが起こったことになります。

本書では WFI 命令を何もしない命令として実装します。

inst\_decoder モジュールで WFI 命令をデコードできるようにします (リスト 14.57)。

## ▼リスト 14.57: WFI 命令のデコード (inst\_decoder.veryl)

```

1 OP_SYSTEM: f3 != 3'b000 && f3 != 3'b100 || // CSRR(W|S|C)[I]
2   bits == 32'h00000073 || // ECALL
3   bits == 32'h00100073 || // EBREAK
4   bits == 32'h30200073 || // MRET
5   bits == 32'h10500073, // WFI
6 OP_MISC_MEM: T, // FENCE

```

WFI 命令で割り込みが発生するとき、mepc レジスタに `pc + 4` を書き込むようにします（リスト 14.58、リスト 14.59）。

## ▼リスト 14.58: WFI 命令の判定 (csrunit.veryl)

```

1 let is_wfi: logic = inst_bits == 32'h10500073;

```

▼リスト 14.59: WFI 命令のとき、mepc を `pc+4` にする (csrunit.veryl)

```

1 if raise_expt || raise_interrupt {
2   mepc = if raise_expt ? pc : // exception
3     if raise_interrupt && is_wfi ? pc + 4 : pc; // interrupt when wfi / interrupt
4   mcause = trap_cause;

```

## 14.7 time、instret、cycle レジスタの実装

RISC-V には time、instret、cycle という読み込み専用の CSR が定義されており、それぞれmtime、minstret、mcycle レジスタと同じ値をとります<sup>\*2</sup>。

`CsrAddr` 型にレジスタのアドレスを追加します（リスト 14.60）。

## ▼リスト 14.60: アドレスの定義 (eei.veryl)

```

1 // Unprivileged Counter/Timers
2 CYCLE = 12'hC00,
3 TIME = 12'hC01,
4 INSTRET = 12'hC02,

```

mtime レジスタの値を ACLINT モジュールから csrunit に渡します（リスト 14.61、リスト 14.62）。

## ▼リスト 14.61: mtime をインターフェースに追加する (aclint\_if.veryl)

```

1 import eei::*;
2
3 interface aclint_if {
4   var msip : logic ;

```

<sup>\*2</sup> mhpcounter レジスタと同じ値をとる hpmcounter レジスタもありますが、mhpcounter レジスタを実装していないので実装しません。

```
5  var mtip : logic ;
6  var mtime: UInt64;
7  modport master {
8      msip : output,
9      mtip : output,
10     mtime: output,
11 }
```

## ▼リスト 14.62: mtime をインターフェースに割り当てる (aclint\_memory.veryl)

```
1  always_comb {
2      aclint.msip  = msip0;
3      aclint.mtip  = mtime >= mtimecmp0;
4      aclint.mtime = mtime;
5 }
```

time、instret、cycle レジスタを読み込めるようにします (リスト 14.63)。

## ▼リスト 14.63: rdata にインターフェースの mtime を割り当てる (csrunit.veryl)

```
1  CsrAddr::CYCLE    : mcycle,
2  CsrAddr::TIME     : aclint.mtime,
3  CsrAddr::INSTRET : minstret,
```

# 第 15 章

## U-mode の実装

本章では RISC-V で最も低い特権レベルである User モード (U-mode) を実装します。U-mode は M-mode に管理されてアプリケーションを動かすための特権レベルであり、M-mode で利用できていたほとんどの CSR、機能が制限されます。

本章で実装、変更する主な機能は次の通りです。それぞれ解説しながら実装していきます。

1. mstatus レジスタの一部のフィールド
2. CSR のアクセス権限、MRET 命令の実行権限の確認
3. mcounteren レジスタ
4. 割り込み条件、トラップの動作

### 15.1 misa.Extensions の変更

U-mode を実装しているかどうかは misa.Extensions の U ビットで確認できます。

misa.Extensions の U ビットを 1 にします (リスト 15.1)。

▼リスト 15.1: U ビットを 1 にする (csrunit.veryl)

```
1 let misa      : UIntX = {2'd2, 1'b0 repeat XLEN - 28, 26'b000001000000100010000101}; // U,>
> M, I, C, A
```

### 15.2 mstatus.UXL の実装

U-mode のときの XLEN は UXLEN と定義されており mstatus.UXL で確認できます。仕様上は mstatus.UXL の書き換えで UXLEN を変更できるように実装できますが、本書では UXLEN が常に 64 になるように実装します。

mstatus.UXL を 64 を示す値である 2 に設定します (リスト 15.2、リスト 15.3)。

## ▼ リスト 15.2: mstatus.UXL の定義 (eei.veryl)

```

1 // mstatus
2 const MSTATUS_UXL: UInt64 = 2 << 32;

```

## ▼ リスト 15.3: UXL の初期値を設定する (csrunit.veryl)

```

1 always_ff {
2     if_reset {
3         mode      = PrivMode::M;
4         mstatus   = MSTATUS_UXL;
5         mtvec    = 0;

```

**15.3 mstatus.TW の実装**

mstatus.TW は、M-mode よりも低い特権レベルで WFI 命令を実行するときに時間制限 (Timeout Wait) を設けるためのビットです。mstatus.TW が 0 のとき時間制限はありません。1 に設定されているとき、CPU の実装固有の時間だけ実行の再開を待ち、時間制限を過ぎると Illegal instruction 例外を発生させます。

本書では mstatus.TW が 1 のときに無限時間待つことにして、例外の実装を省略します。mstatus.TW を書き換えられるようにします（リスト 15.4）。

## ▼ リスト 15.4: 書き込みマスクを変更する (csrunit.veryl)

```

1 const MSTATUS_WMASK : UIntX = 'h0000_0000_0020_0088 as UIntX;

```

**15.4 mstatus.MPP の実装**

M-mode、U-mode だけが存在する環境でトラップが発生するとき、CPU は mstatus レジスタの MPP フィールドに現在の特権レベル（を示す値）を保存し、特権レベルを M-mode に変更します。また、MRET 命令を実行すると mstatus.MPP の特権レベルに移動するようになります。

これにより、トラップによる U(M)-mode から M-mode への遷移、MRET 命令による M-mode から U-mode への遷移を実現できます。

MRET 命令を実行すると mstatus.MPP は実装がサポートする最低の特権レベルに設定されます。

M-mode から U-mode に遷移したいときは、mstatus.MPP を U-mode の値に変更し、U-mode で実行を開始したいアドレスを mepc レジスタに設定して MRET 命令を実行します。

mstatus.MPP に値を書き込めるようにします（リスト 15.5）。

## ▼リスト 15.5: 書き込みマスクを変更する (csrunit.veryl)

```
1 const MSTATUS_WMASK : UIntX = 'h0000_0000_0020_1888 as UIntX;
```

MPP には `2'b00` (U-mode) と `2'b11` (M-mode) のみ設定できるようにします。サポートしていない値を書き込もうとする場合は現在の値を維持します ( リスト 15.6、リスト 15.7 )。

## ▼リスト 15.6: mstatus の書き込み (csrunit.veryl)

```
1 CsrAddr::MSTATUS : mstatus = validate_mstatus(mstatus, wdata);
```

## ▼リスト 15.7: mstatus レジスタの値を確認する関数 (csrunit.veryl)

```
1 function validate_mstatus (
2     mstatus: input UIntX,
3     wdata : input UIntX,
4 ) -> UIntX {
5     var result: UIntX;
6     result = wdata;
7     // MPP
8     if wdata[12:11] != PrivMode::M && wdata[12:11] != PrivMode::U {
9         result[12:11] = mstatus[12:11];
10    }
11    return result;
12 }
```

トラップが発生する、トラップから戻るときの遷移先の特権レベルを求めます ( リスト 15.8、リスト 15.9、リスト 15.10、リスト 15.11、リスト 15.12 )。

## ▼リスト 15.8: ビットを変数として定義する (csrunit.veryl)

```
1 let mstatus_mpp : PrivMode = mstatus[12:11] as PrivMode;
2 let mstatus_mpie: logic      = mstatus[7];
3 let mstatus_mie : logic      = mstatus[3];
```

## ▼リスト 15.9: 割り込みの遷移先の特権レベルを示す変数 (csrunit.veryl)

```
1 let interrupt_mode: PrivMode = PrivMode::M;
```

## ▼リスト 15.10: 例外の遷移先の特権レベルを示す変数 (csrunit.veryl)

```
1 let expt_mode : PrivMode = PrivMode::M;
```

## ▼リスト 15.11: MRET 命令の遷移先の特権レベルを示す変数 (csrunit.veryl)

```
1 let trap_return_mode: PrivMode = mstatus_mpp;
```

## ▼リスト 15.12: 遷移先の特権レベルを求める (csrunit.veryl)

```
1 let trap_mode_next: PrivMode = switch {
2     raise_expt      : expt_mode,
3     raise_interrupt: interrupt_mode,
4     trap_return    : trap_return_mode,
5     default        : PrivMode::U,
```

```
6 };
```

トラップが発生するとき、mstatus.MPP に現在の特権レベルを保存します（リスト 15.13）。また、トラップから戻るとき、特権レベルを mstatus.MPP に設定し、mstatus.MPP に実装がサポートする最小の特権レベルである `PrivMode::U` を書き込みます。

#### ▼ リスト 15.13: 特権レベル、mstatus.MPP を更新する (csrunit.veryl)

```
1 if raise_trap {
2     if raise_expt || raise_interrupt {
3         ...
4         // save current privilege level to mstatus.mpp
5         @<b<|mstatus[12:11] = mode; |
6     } else if trap_return {
7         ...
8         // set mstatus.mpp = U (least privilege level)
9         mstatus[12:11] = PrivMode::U;
10    }
11    mode = trap_mode_next;
```

## 15.5 CSR のアクセス権限の確認

CSR のアドレスを `csr_addr` とするとき、`csr_addr[9:8]` の 2 ビットはその CSR にアクセスできる最低の特権レベルを表しています。これを下回る特権レベルで CSR にアクセスしようとすると Illegal instruction 例外が発生します。

CSR のアドレスと特権レベルを確認して、例外を起こすようにします（リスト 15.14、リスト 15.15、リスト 15.16）。

#### ▼ リスト 15.14: 現在の特権レベルで CSR にアクセスできるか判定する (csrunit.veryl)

```
1 let expt_csr_privViolation: logic = is_wsc && csr_addr[9:8] >: mode; // attempt to access C>SR without privilege level
```

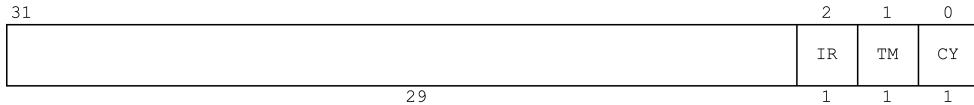
#### ▼ リスト 15.15: 例外の発生条件に追加する (csrunit.veryl)

```
1 let raise_expt: logic = valid && (expt_info.valid || expt_write_READONLY_CSR || expt_csr_priv_violation);
```

#### ▼ リスト 15.16: cause を設定する (csrunit.veryl)

```
1 expt_write_READONLY_CSR: CsrCause::ILLEGAL_INSTRUCTION,
2 expt_csr_privViolation: CsrCause::ILLEGAL_INSTRUCTION,
3 default : 0,
```

## 15.6 mcounteren レジスタの実装



▲図 15.1: mcounteren レジスタ

mcounteren レジスタは、M-mode の次に低い特権レベルでハードウェアパフォーマンスモニタにアクセスできるようにするかを制御する 32 ビットのレジスタです(図 15.1)。CY、TM、IR ビットはそれぞれ cycle、time、instret にアクセスできるかどうかを制御します<sup>\*1</sup>。

本章で M-mode の次に低い特権レベルとして U-mode を実装するため、mcounteren レジスタは U-mode でのアクセスを制御します。mcounteren レジスタで許可されていないまま U-mode で cycle、time、instret レジスタにアクセスしようとすると、Illegal Instruction 例外が発生します。

mcounteren レジスタを作成し、CY、TM、IR ビットに書き込みできるようにします(リスト 15.17、リスト 15.19、リスト 15.20、リスト 15.21、リスト 15.18、リスト 15.22)。

▼リスト 15.17: mcounteren レジスタの定義 (csrunit.veryl)

```
1 var mcounteren: UInt32;
```

▼リスト 15.18: mcounteren レジスタを 0 でリセットする (csrunit.veryl)

```
1 mie      = 0;
2 mcounteren = 0;
3 mscratch = 0;
```

▼リスト 15.19: rdata に mcounteren レジスタを設定する (csrunit.veryl)

```
1 CsrAddr::MIE      : mie,
2 CsrAddr::MCOUNTEREN: {1'b0 repeat XLEN - 32, mcounteren},
3 CsrAddr::MCYCLE   : mcycle,
```

▼リスト 15.20: 書き込みマスクの定義 (csrunit.veryl)

```
1 const MCOUNTEREN_WMASK: UIntX = 'h0000_0000_0000_0007 as UIntX;
```

▼リスト 15.21: wmask に書き込みマスクを設定する (csrunit.veryl)

```
1 CsrAddr::MIE      : MIE_WMASK,
2 CsrAddr::MCOUNTEREN: MCOUNTEREN_WMASK,
3 CsrAddr::MSCRATCH : MSCRATCH_WMASK,
```

<sup>\*1</sup> hpmcounter レジスタを制御する HPM ビットもありますが、hpmcounter レジスタを実装していないので実装しません

## ▼ リスト 15.22: mcounteren レジスタの書き込み (csrunit.veryl)

```

1  CsrAddr::MIE      : mie        = wdata;
2  CsrAddr::MCOUNTEREN: mcounteren = wdata[31:0];
3  CsrAddr::MSCRATCH : mscratch  = wdata;

```

U-mode でハードウェアパフォーマンスマニタにアクセスするとき、mcounteren レジスタのビットが 0 なら Illegal instruction 例外を発生させます（リスト 15.23、リスト 15.24）。

## ▼ リスト 15.23: U-mode のとき、mcounteren レジスタを確認する (csrunit.veryl)

```

1  let expt_zicntr_priv      : logic = is_wsc && mode == PrivMode::U && case csr_addr {
2    CsrAddr::CYCLE : !mcounteren[0],
3    CsrAddr::TIME  : !mcounteren[1],
4    CsrAddr::INSTRET: !mcounteren[2],
5    default        : 0,
6  }; // attempt to access Zicntr CSR without permission

```

## ▼ リスト 15.24: cause を設定する (csrunit.veryl)

```

1  expt_csr_privViolation: CsrCause::ILLEGAL_INSTRUCTION,
2  expt_zicntr_priv       : CsrCause::ILLEGAL_INSTRUCTION,
3  default                 : 0,

```

## 15.7 MRET 命令の実行を制限する

MRET 命令は M-mode 以上の特権レベルのときにしか実行できません。M-mode 未満の特権レベルで MRET 命令を実行しようとすると Illegal instruction 例外が発生します。

命令が MRET 命令のとき、特権レベルを確認して例外を発生させます（リスト 15.25、リスト 15.26、リスト 15.27）。

## ▼ リスト 15.25: MRET 命令を実行するとき、現在の特権レベルを確認する (csrunit.veryl)

```

1  let expt_trap_return_priv: logic = is_mret && mode <: PrivMode::M; // attempt to execute tra>
2  >p return instruction in low privilege level

```

## ▼ リスト 15.26: 例外の発生条件に追加する (csrunit.veryl)

```

1  let raise_expt: logic = valid && (expt_info.valid || expt_write_READONLY_CSR || expt_csr_pri>
2  >vViolation || expt_zicntr_priv || expt_trap_return_priv);

```

## ▼ リスト 15.27: cause を設定する (csrunit.veryl)

```

1  expt_zicntr_priv      : CsrCause::ILLEGAL_INSTRUCTION,
2  expt_trap_return_priv : CsrCause::ILLEGAL_INSTRUCTION,
3  default                 : 0,
4  };

```

## 15.8 ECALL 命令の cause を変更する

M-mode で ECALL 命令を実行すると Environment call from M-mode 例外が発生します。これに対して U-mode で ECALL 命令を実行すると Environment call from U-mode 例外が発生します。特権レベルと例外の対応は表 15.1 のようになっています。

▼ 表 15.1: ECALL 命令を実行したときに発生する例外

特権レベル	例外	cause
M-mode	Environment call from M-mode	11
S-mode	Environment call from S-mode	9
U-mode	Environment call from U-mode	8

ここで各例外の cause が U-mode の cause に特権レベルの数値を足したものになっていることを利用します。 `CsrCause` 型に Environment call from U-mode 例外の cause を追加します（リスト 15.28）。

▼ リスト 15.28: CsrCause 型に例外の cause を追加する (eei.veryl)

```
1 STORE_AMO_ADDRESS_MISALIGNED = 6,
2 ENVIRONMENT_CALL_FROM_U_MODE = 8,
3 ENVIRONMENT_CALL_FROM_M_MODE = 11,
```

csrunit モジュールの `mode` レジスタをポート宣言に移動し、ID ステージで ECALL 命令をコードするときに cause に `mode` を足します（リスト 15.29、リスト 15.30、リスト 15.31、リスト 15.32）。

▼ リスト 15.29: mode レジスタをポートに移動する (csrunit.veryl)

```
1 rdata      : output UIntX      ,
2 mode       : output PrivMode   ,
3 raise_trap : output logic     ,
```

▼ リスト 15.30: csrunit から現在の特権レベルを受け取る変数 (core.veryl)

```
1 var csru_priv_mode : PrivMode;
```

▼ リスト 15.31: csrunit モジュールのインスタンスから現在の特権レベルを受け取る (core.veryl)

```
1 rdata      : csru_rdata      ,
2 mode       : csru_priv_mode   ,
3 raise_trap : csru_raise_trap ,
```

▼ リスト 15.32: Environment call from U-mode 例外の cause に特権レベルの数値を足す (core.veryl)

```
1 } else if ids_inst_bits == 32'h00000073 {
2     // ECALL
3     exq_wdata.expt.valid    = 1;
4     exq_wdata.expt.cause     = CsrCause::ENVIRONMENT_CALL_FROM_U_MODE;
```

```
5     exq_wdata.expt.cause[1:0] = csr_u_priv_mode;
6     exq_wdata.expt.value      = 0;
```

## 15.9 割り込み条件の変更

M-modeだけが実装されたCPUで割り込みが発生する条件は「14.1.2 RISC-Vの割り込み」(p.272)で解説しましたが、M-modeとU-modeだけが実装されたCPUで割り込みが発生する条件は少し異なります。M-modeとU-modeだけが実装されたCPUで割り込みが発生する条件は次の通りです。

1. 割り込み原因に対応した mip レジスタのビットが 1 である
2. 割り込み原因に対応した mie レジスタのビットが 1 である
3. 現在の特権レベルが M-mode 未満である。または mstatus.MIE が 1 である

M-modeだけの場合と違い、現在の特権レベルが U-mode のときはグローバル割り込みイネーブルビット(mstatus.MIE)の値は考慮されずに割り込みが発生します。

現在の特権レベルによって割り込みが発生する条件を切り替えます。U-modeのときは mstatus.MIE を考慮しないようにします(リスト 15.33)。

### ▼リスト 15.33: U-modeのとき、割り込みの発生条件を変更する(csrunit.verv1)

```
1 let raise_interrupt : logic = valid && can_intr && (mode != PrivMode::M || mstatus_mie) &&
  > interrupt_pending != 0;
```

# 第 16 章

## S-mode の実装 (1. CSR の実装)

本章では Supervisor モード (S-mode) を実装します。S-mode は主に OS のようなシステムアプリケーションを動かすために使用される特権レベルです。S-mode がある環境には必ず U-mode が実装されています。

S-mode を導入することで変わる主要な機能はトラップです。M-mode、U-mode だけの環境ではトラップで特権レベルを M-mode に変更していましたが、M-mode ではなく S-mode に遷移できるようになります。これに伴い、トラップ関連の CSR(stvec、sepc、scause、stval など) が追加されます。

S-mode で新しく導入される大きな機能として仮想記憶システムがあります。仮想記憶システムはページングを使って仮想的なアドレスを使用できるようにする仕組みです。これについては第 17 章「S-mode の実装 (2. 仮想記憶システム)」で解説します。

他には scounteren レジスタ、トラップから戻るための SRET 命令などが追加されます。また、Supervisor software interrupt を提供する SSWI デバイスも実装します。それぞれ解説しながら実装します。

eei パッケージに、本書で実装する S-mode の CSR をすべて定義します。

### ▼リスト 16.1: CSR のアドレスを定義する (eei.vetyl)

```
1 enum CsrAddr: logic<12> {
2     // Supervisor Trap Setup
3     SSTATUS = 12'h100,
4     SIE = 12'h104,
5     STVEC = 12'h105,
6     SCOUNTEREN = 12'h106,
7     // Supervisor Trap Handling
8     SSCRATCH = 12'h140,
9     SEPC = 12'h141,
10    SCAUSE = 12'h142,
11    STVAL = 12'h143,
12    SIP = 12'h144,
13    // Supervisor Protection and Translation
14    SATP = 12'h180,
```

## 16.1

# misa.Extensions、mstatus.SXL、mstatus.MPP の実装

S-mode を実装しているかどうかは misa.Extensions の S ビットで確認できます。

misa.Extensions の S ビットを 1 に設定します (リスト 16.2)。

▼ リスト 16.2: S ビットを 1 にする (csrunit.veryl)

```
1 let misa      : UIntX = {2'd2, 1'b0 repeat XLEN - 28, 26'b000001010000100010000101}; // >
> U, S, M, I, C, A
```

S-mode のときの XLEN は SXLEN と定義されており、mstatus.SXL で確認できます。本書では SXLEN が常に 64 になるように実装します。

mstatus.SXL を 64 を示す値である 2 に設定します (リスト 16.3、リスト 16.4)。

▼ リスト 16.3: mstatus.SXL の定義 (eei.veryl)

```
1 const MSTATUS_UXL: UInt64 = 2 << 32;
2 const MSTATUS_SXL: UInt64 = 2 << 34;
```

▼ リスト 16.4: mstatus.SXL の初期値を設定する (csrunit.veryl)

```
1 always_ff {
2     if_reset {
3         mode      = PrivMode:::M;
4         mstatus   = MSTATUS_SXL | MSTATUS_UXL;
```

今のところ mstatus.MPP には M-mode と U-mode を示す値しか書き込めないようにしているので、S-mode の値 ( 2'b10 ) も書き込めるように変更します (リスト 16.5)。これにより、MRET 命令で S-mode に移動できるようになります。

▼ リスト 16.5: MPP に S-mode を書き込めるようにする (csrunit.veryl)

```
1 function validate_mstatus (
2     mstatus: input UIntX,
3     wdata  : input UIntX,
4 ) -> UIntX {
5     var result: UIntX;
6     result = wdata;
7     // MPP
8     if wdata[12:11] == 2'b10 {
9         result[12:11] = mstatus[12:11];
10    }
11    return result;
12 }
```

## 16.2

# scounteren レジスタの実装

「15.6 mcountreren レジスタの実装」(p.298) では、ハードウェアパフォーマンスマニタに U-

mode でアクセスできるかを mcounteren レジスタで制御できるようにしました。S-mode を導入すると mcounteren レジスタは S-mode がハードウェアパフォーマンスマニタにアクセスできるかを制御するレジスタに変わります。また、mcounteren レジスタの代わりに U-mode でハードウェアパフォーマンスマニタにアクセスできるかを制御する 32 ビットの scounteren レジスタが追加されます。

scounteren レジスタのフィールドのビット配置は mcounteren レジスタと同じです。また、U-mode でハードウェアパフォーマンスマニタにアクセスできる条件は、mcounteren レジスタと scounteren レジスタの両方によって許可されている場合になります。

scounteren レジスタを作成し、読み書きできるようにします（リスト 16.6、リスト 16.7、リスト 16.8、リスト 16.9、リスト 16.10、リスト 16.11）。

#### ▼ リスト 16.6: scounteren レジスタの定義 (csrunit.veryl)

```
1 var scounteren: UInt32;
```

#### ▼ リスト 16.7: scounteren レジスタを 0 でリセットする (csrunit.veryl)

```
1 mtval      = 0;
2 scounteren = 0;
3 led       = 0;
```

#### ▼ リスト 16.8: rdata に scounteren レジスタの値を設定する (csrunit.veryl)

```
1 CsrAddr::MTVAL      : mtval,
2 CsrAddr::SCOUNTEREN: {1'b0 repeat XLEN - 32, scounteren},
3 CsrAddr::LED        : led,
```

#### ▼ リスト 16.9: 書き込みマスクの定義 (csrunit.veryl)

```
1 const SCOUNTEREN_WMASK: UIntX = 'h0000_0000_0000_0007 as UIntX;
```

#### ▼ リスト 16.10: wmask に書き込みマスクを設定する (csrunit.veryl)

```
1 CsrAddr::MTVAL      : MTVAL_WMASK,
2 CsrAddr::SCOUNTEREN: SCOUNTEREN_WMASK,
3 CsrAddr::LED        : LED_WMASK,
```

#### ▼ リスト 16.11: scounteren レジスタに書き込む (csrunit.veryl)

```
1 CsrAddr::MTVAL      : mtval      = wdata;
2 CsrAddr::SCOUNTEREN: scounteren = wdata[31:0];
3 CsrAddr::LED        : led       = wdata;
```

ハードウェアパフォーマンスマニタにアクセスするときに許可を確認する仕組みを実装します（リスト 16.12）。S-mode でアクセスするときは mcounteren レジスタだけ確認し、U-mode でアクセスするときは mcounteren レジスタと scounteren レジスタを確認します。

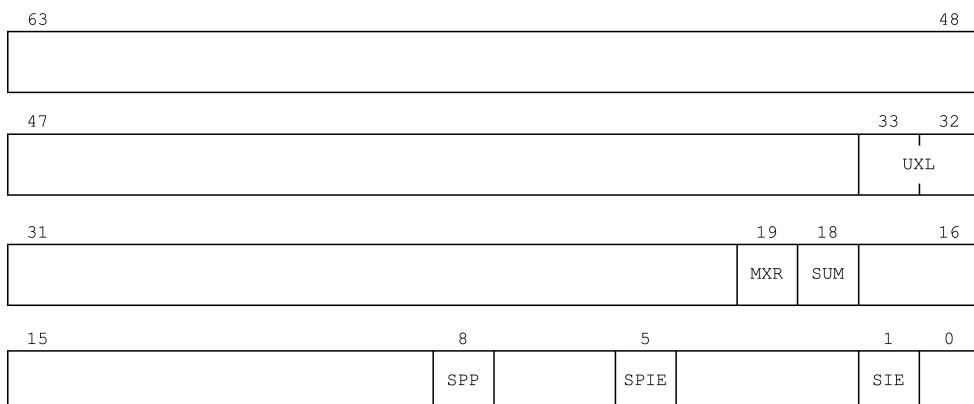
#### ▼ リスト 16.12: 許可の確認ロジックを変更する (csrunit.veryl)

```

1  let expt_zicntr_priv      : logic = is_wsc && (mode <= PrivMode::S && case csr_addr {
2    CsrAddr::CYCLE   : !mcounteren[0],
3    CsrAddr::TIME    : !mcounteren[1],
4    CsrAddr::INSTRET: !mcounteren[2],
5    default         : 0,
6  } || mode <= PrivMode::U && case csr_addr {
7    CsrAddr::CYCLE   : !scounteren[0],
8    CsrAddr::TIME    : !scounteren[1],
9    CsrAddr::INSTRET: !scounteren[2],
10   default        : 0,
11 ); // attempt to access Zicntr CSR without permission

```

## 16.3 sstatus レジスタの実装



▲ 図 16.1: sstatus レジスタ

sstatus レジスタは mstatus レジスタの一部を S-mode で読み込み、書き込みできるようにした SXLEN ビットのレジスタです。本章では mstatus レジスタに読み込み、書き込みマスクを適用することで sstatus レジスタを実装します。

sstatus レジスタの書き込みマスクを定義します（リスト 16.13、リスト 16.14）。

### ▼ リスト 16.13: 書き込みマスクの定義 (csrunit.veryl)

```

1 const SSTATUS_WMASK : UIntX = 'h0000_0000_0000_0000 as UIntX;

```

### ▼ リスト 16.14: wmask に書き込みマスクを設定する (csrunit.veryl)

```

1 CsrAddr::MTVAL      : MTVAL_WMASK,
2 CsrAddr::SSTATUS    : SSTATUS_WMASK,
3 CsrAddr::SCOUNTEREN: SCOUNTEREN_WMASK,

```

読み込みマスクを定義し、mstatus レジスタにマスクを適用した値を sstatus レジスタの値にし

ます ( リスト 16.15、リスト 16.16、リスト 16.17 )。

▼ リスト 16.15: 読み込みマスクの定義 (csrunit.veryl)

```
1 const SSTATUS_RMASK: UIntX = 'h8000_0003_018f_e762;
```

▼ リスト 16.16: sstatus の値を mstatus にマスクを適用したものにする (csrunit.veryl)

```
1 let sstatus : UIntX = mstatus & SSTATUS_RMASK;
```

▼ リスト 16.17: rdata に sstatus レジスタの値を設定する (csrunit.veryl)

```
1 CsrAddr::MTVAL      : mtval,
2 CsrAddr::SSTATUS    : sstatus,
3 CsrAddr::SCOUNTEREN: {1'b0 repeat XLEN - 32, scounteren},
```

マスクを適用した書き込みを実装します ( リスト 16.18 )。書き込みマスクが適用された wdata と、書き込みマスクをビット反転した値でマスクされた mstatus レジスタの値の OR を書き込みます。

▼ リスト 16.18: sstatus レジスタへの書き込みで mstatus レジスタに書き込む (csrunit.veryl)

```
1 CsrAddr::SSTATUS : mstatus = validate_mstatus(mstatus, wdata | mstatus & ~SSTATUS_WMASK>);
```

## 16.4 トランプの委譲

### 16.4.1 トランプの委譲

S-mode が実装されているとき、S-mode と U-mode で発生するトランプの遷移先の特権レベルを M-mode から S-mode に変更 ( 委譲 ) することができます。特権レベルが M-mode のときに発生したトランプの特権レベルの遷移先を S-mode に変更することはできません。

M-mode から S-mode に委譲されたトランプのトランプベクタは、mtvec ではなく stvec になります。また、mepc ではなく sepc にトランプが発生した命令アドレスを格納し、scause にトランプの原因を示す値、stval に例外に固有の情報、sstatus.SPP にトランプ前の特権レベル、sstatus.SPIE に sstatus.SIE、sstatus.SIE に 0 を格納します。これ以降、トランプで x-mode に遷移するときに変更、参照する CSR を例えば xtvec、xepc、xcause、xtval、mstatus.xPP のように頭文字を x にして呼ぶことがあります。

#### 例外の委譲

medeleg レジスタは、どの例外を委譲するかを制御する 64 ビットのレジスタです。medeleg レジスタの下から i 番目のビットが立っているとき、S-mode、U-mode で発生した cause が i の例外を S-mode に委譲します。M-mode で発生した例外は S-mode に委譲されません。

Environment call from M-mode 例外のように委譲することができない命令の medeleg レジスタのビットは 1 に変更できません。

## 割り込みの委譲

mideleg レジスタは、どの割り込みを委譲するかを制御する MXLEN ビットのレジスタです。各割り込みは mie、mip レジスタと同じ場所の mideleg レジスタのビットによって委譲されるかどうかが制御されます。

M-mode、S-mode、U-mode が実装された CPU で、割り込みで M-mode に遷移する条件は次の通りです。

1. 割り込み原因に対応した mip レジスタのビットが **1** である
2. 割り込み原因に対応した mie レジスタのビットが **1** である
3. 現在の特権レベルが M-mode 未満である。または mstatus.MIE が **1** である
4. 割り込み原因に対応した mideleg レジスタのビットが **0** である

割り込みで S-mode に遷移する条件は次の通りです。

1. 割り込み原因に対応した sip レジスタのビットが **1** である
2. 割り込み原因に対応した sie レジスタのビットが **1** である
3. 現在の特権レベルが S-mode 未満である。または S-mode のとき、sstatus.SIE が **1** である

sip、sie レジスタは、それぞれ mip、mie レジスタの委譲された割り込みのビットだけ読み込み、書き込みできるようにしたレジスタです。委譲されていない割り込みに対応したビットは読み込み専用の **0** になります。S-mode に委譲された割り込みは、特権レベルが M-mode のときは発生しません。

S-mode に委譲された割り込みは外部割り込み、ソフトウェア割り込み、タイマ割り込みの順に優先されます。委譲されていない割り込みを同じタイミングで発生させられるとき、委譲されていない割り込みが優先されます。

本書では M-mode の外部割り込み (Machine external interrupt)、ソフトウェア割り込み (Machine software interrupt)、タイマ割り込み (Machine timer interrupt) は S-mode に委譲できないように実装します<sup>\*1</sup>。

### 16.4.2 トランプに関連するレジスタを作成する

S-mode に委譲されたトランプで使用する stvec、sscratch、sepc、scause、stval レジスタを作成します (リスト 16.19、リスト 16.20、リスト 16.21、リスト 16.22、リスト 16.23、リスト 16.24)。

#### ▼ リスト 16.19: レジスタの定義 (csrunit.ver.vy)

```

1  var stvec      : UIntX ;
2  var sscratch   : UIntX ;
3  var sepc       : UIntX ;
4  var scause     : UIntX ;
5  var stval      : UIntX ;

```

<sup>\*1</sup> 多くの実装ではこれらの割り込みを委譲できないように実装するようです。そのため、本書で実装するコアでも委譲できないように実装します。

## ▼リスト 16.20: レジスタを 0 でリセットする (csrunit.veryl)

```

1  stvec      = 0;
2  sscratch   = 0;
3  sepc       = 0;
4  scause     = 0;
5  stval      = 0;

```

## ▼リスト 16.21: rdata にレジスタの値を割り当てる (csrunit.veryl)

```

1  CsrAddr::STVEC    : stvec,
2  CsrAddr::SSCRATCH : sscratch,
3  CsrAddr::SEPC     : sepc,
4  CsrAddr::SCAUSE   : scause,
5  CsrAddr::STVAL    : stval,

```

それぞれ、mtvec、mscratch、mepc、mcause、mtval レジスタと同じ書き込みマスクを設定します。

## ▼リスト 16.22: 書き込みマスクの定義 (csrunit.veryl)

```

1  const STVEC_WMASK   : UIntX = 'hffff_ffff_ffff_fffd;
2  const SSCRATCH_WMASK : UIntX = 'hffff_ffff_ffff_ffff;
3  const SEPC_WMASK    : UIntX = 'hffff_ffff_ffff_fffe;
4  const SCAUSE_WMASK  : UIntX = 'hffff_ffff_ffff_ffff;
5  const STVAL_WMASK   : UIntX = 'hffff_ffff_ffff_ffff;

```

## ▼リスト 16.23: wmask に書き込みマスクを設定する (csrunit.veryl)

```

1  CsrAddr::STVEC    : STVEC_WMASK,
2  CsrAddr::SSCRATCH : SSCRATCH_WMASK,
3  CsrAddr::SEPC     : SEPC_WMASK,
4  CsrAddr::SCAUSE   : SCAUSE_WMASK,
5  CsrAddr::STVAL    : STVAL_WMASK,

```

## ▼リスト 16.24: レジスタの書き込み (csrunit.veryl)

```

1  CsrAddr::STVEC    : stvec      = wdata;
2  CsrAddr::SSCRATCH : sscratch   = wdata;
3  CsrAddr::SEPC     : sepc       = wdata;
4  CsrAddr::SCAUSE   : scause     = wdata;
5  CsrAddr::STVAL    : stval      = wdata;

```

### 16.4.3 stvec レジスタの実装

トランプが発生するとき、遷移先の特権レベルが S-mode なら stvec レジスタの値にジャンプするようにします（リスト 16.25、リスト 16.26）。割り込み、例外それぞれにレジスタを選択する変数を定義し、mtvec を使っていたところを新しい変数に置き換えます。

## ▼リスト 16.25: トランプベクタを遷移先の特権レベルによって変更する (csrunit.veryl)

```

1  let interrupt_xtvec : Addr = if interrupt_mode == PrivMode::M ? mtvec : stvec;
2  let interrupt_vector: Addr = if interrupt_xtvec[0] == 0 ?

```

```

3   {interrupt_xtvec[msb:2], 2'b0}
4   : // Direct
5   {interrupt_xtvec[msb:2] + interrupt_cause[msb - 2:0], 2'b0}
6   ; // Vectored

```

▼ リスト 16.26: トランプベクタを遷移先の特権レベルによって変更する (csrunit.vryl)

```

1 let expt_xtvec : Addr      = if expt_mode == PrivMode::M ? mtvec : stvec;
2 let expt_vector: Addr      = {expt_xtvec[msb:2], 2'b0};

```

#### 16.4.4 トランプで sepc、scause、stval レジスタを変更する

トランプが発生するとき、遷移先の特権レベルが S-mode なら sepc、scause、stval レジスタを変更するようにします。

トランプ時に `trap_mode_next` で処理を分岐します (リスト 16.27)。

▼ リスト 16.27: 遷移先の特権レベルによってトランプ処理を分岐する (csrunit.vryl)

```

1 if raise_expt || raise_interrupt {
2     let xepc: Addr = if raise_expt ? pc : // exception
3         if raise_interrupt && is_wfi ? pc + 4 : pc; // interrupt when wfi / interrupt
4         if trap_mode_next == PrivMode::M {
5             mepc    = xepc;
6             mcause = trap_cause;
7             if raise_expt {
8                 mtval = expt_value;
9             }
10            // save mstatus.mie to mstatus.mpie
11            // and set mstatus.mie = 0
12            mstatus[7] = mstatus[3];
13            mstatus[3] = 0;
14            // save current privilege level to mstatus.mpp
15            mstatus[12:11] = mode;
16        } else {
17            sepc    = xepc;
18            scause = trap_cause;
19            if raise_expt {
20                stval = expt_value;
21            }
22        }

```

#### 16.4.5 mstatus の SIE、SPIE、SPP ビットを実装する

`mstatus` レジスタの SIE、SPIE、SPP ビットを実装します。`mstatus.SIE` は S-mode に委譲された割り込みのグローバル割り込みイネーブルビットです。`mstatus.SPIE` は S-mode に委譲されたトランプが発生するときに `mstatus.SIE` を退避するビットです。`mstatus.SPP` は S-mode に委譲されたトランプが発生するときに、トランプ前の特権レベルを書き込むビットです。S-mode に委譲されたトランプは S-mode か U-mode でしか発生しないため、`mstatus.SPP` は特権レベルを区別するために十分な 1 ビット幅のフィールドになっています。

mstatus, sstatus レジスタの SIE、SPIE、SPP ビットに書き込めるようにします（リスト 16.28、リスト 16.29）。

▼リスト 16.28: 書き込みマスクを変更する (csrunit.veryl)

```
1 const MSTATUS_WMASK : UIntX = 'h0000_0000_0020_19aa as UIntX;
```

▼リスト 16.29: 書き込みマスクを変更する (csrunit.veryl)

```
1 const SSTATUS_WMASK : UIntX = 'h0000_0000_0000_0122 as UIntX;
```

トランプで S-mode に遷移するとき、sstatus.SPIE に sstatus.SIE、sstatus.SIE に 0、sstatus.SPP にトランプ前の特権レベルを格納します（リスト 16.30）。

▼リスト 16.30: sstatus.SPIE、SIE、SPP をトランプで変更する (csrunit.veryl)

```
1 } else {
2     sepc = xepc;
3     scause = trap_cause;
4     if raise_expt {
5         stval = expt_value;
6     }
7     // save sstatus.sie to sstatus.spi
8     // and set sstatus.sie = 0
9     mstatus[5] = mstatus[1];
10    mstatus[1] = 0;
11    // save current privilege mode (S or U) to sstatus.spp
12    mstatus[8] = mode[0];
13 }
```

## 16.4.6 SRET 命令を実装する

### SRET 命令の実装

SRET 命令は、S-mode の CSR(sepc、sstatus など) を利用してトランプ処理から戻るための命令です。SRET 命令は S-mode 以上の特権レベルのときにしか実行できません。

inst\_decoder モジュールで SRET 命令をデコードできるようにします（リスト 16.31）。

▼リスト 16.31: SRET 命令のとき valid を 1 にする (inst\_decoder.veryl)

```
1 OP_SYSTEM: f3 != 3'b000 && f3 != 3'b100 || // CSRR(W|S|C)[I]
2 bits == 32'h00000073 || // ECALL
3 bits == 32'h00100073 || // EBREAK
4 bits == 32'h30200073 || // MRET
5 bits == 32'h10200073 || // SRET
6 bits == 32'h10500073, // WFI
```

SRET 命令を判定し、ジャンプ先と遷移先の特権レベルを命令によって切り替えます（リスト 16.32、リスト 16.33、リスト 16.34）。

## ▼リスト 16.32: SRT 命令の判定 (csrunit.vyrl)

```
1 let is_sret: logic = inst_bits == 32'h10200073;
```

## ▼リスト 16.33: SRET 命令のとき遷移先の特権レベル、アドレスを変更する (csrunit.vyrl)

```
1 assign trap_return      = valid && (is_mret || is_sret) && !raise_expt && !raise_interrup>t;
2   let trap_return_mode : PrivMode = if is_mret ? mstatus_mpp : mstatus_spp;
3   let trap_return_vector: Addr    = if is_mret ? mepc : sepc;
```

## ▼リスト 16.34: trap\_return\_vector を trap\_vector に割り当てる (csrunit.vyrl)

```
1 assign trap_vector = switch {
2   raise_expt      : expt_vector,
3   raise_interrupt: interrupt_vector,
4   trap_return     : trap_return_vector,
5   default         : 0,
6};
```

SRET 命令を実行するとき、sstatus.SIE に sstatus.SPIE、sstatus.SPIE に 0、sstatus.SPP に実装がサポートする最小の特権レベル (U-mode) を示す値を格納します (リスト 16.35)。

## ▼リスト 16.35: SRET 命令による sstatus の変更 (csrunit.vyrl)

```
1 } else if trap_return {
2   if is_mret {
3     // set mstatus.mie = mstatus.mpie
4     //   mstatus.mpie = 0
5     mstatus[3] = mstatus[7];
6     mstatus[7] = 0;
7     // set mstatus.mpp = U (least privilege level)
8     mstatus[12:11] = PrivMode::U;
9   } else if is_sret {
10     // set sstatus.sie = sstatus.spie
11     //   sstatus.spie = 0
12     mstatus[1] = mstatus[5];
13     mstatus[5] = 0;
14     // set sstatus.spp = U (Least privilege level)
15     mstatus[8] = 0;
16   }
17 }
```

SRET 命令を S-mode 未満の特権レベルで実行しようとしたら例外が発生するようにします (リスト 16.36)。

## ▼リスト 16.36: SRET 命令を実行するときに特権レベルを確認する (csrunit.vyrl)

```
1 let expt_trap_return_priv: logic = (is_mret && mode <: PrivMode::M) || (is_sret && mode <: P>rivMode::S);
```

**mstatus.TSR の実装**

mstatus レジスタの TSR(Trap SRET) ビットは、SRET 命令を S-mode で実行したときに例

外を発生させるかを制御するビットです。1 のとき、Illegal instruction 例外が発生するようになります。

mstatus.TSR を変更できるようにします（リスト 16.37）。

▼ リスト 16.37: 書き込みマスクを変更する (csrunit.veryl)

```
1 const MSTATUS_WMASK : UIntX = 'h0000_0000_0060_19aa as UIntX;
```

例外を判定します（リスト 16.38、リスト 16.39）。

▼ リスト 16.38: TSR ビットを表す変数 (csrunit.veryl)

```
1 let mstatus_tsr : logic = mstatus[22];
```

▼ リスト 16.39: mstatus.TSR が 1 のときに S-mode で SRET 命令を実行したら例外にする (csrunit.veryl)

```
1 let expt_trap_return_priv: logic = (is_mret && mode <: PrivMode::M) || (is_sret && (mode <: >PrivMode::S || (mode == PrivMode::S && mstatus_tsr)));
```

## 16.4.7 SEI、SSI、STI を実装する

S-mode を導入すると、S-mode の外部割り込み（Supervisor external interrupt）、ソフトウェア割り込み（Supervisor software interrupt）、タイマ割り込み（Supervisor timer interrupt）に対応する mip、mie レジスタのビットを変更できるようになります。

例外、割り込みはそれぞれ medeleg、mideleg レジスタで S-mode に処理を委譲することができます。委譲された割り込みの mip レジスタの値は sip レジスタで観測できるようになり、割り込みを有効にするかを sie レジスタで制御できるようになります。

### mip、mie レジスタの変更

mip レジスタの SEIP、SSIP、STIP ビット、mie レジスタの SEIE、SSIE、STIE ビットを変更できるようにします。

書き込みマスクを変更、実装します（リスト 16.40、リスト 16.41）。

▼ リスト 16.40: 書き込みマスクの定義 / 変更 (csrunit.veryl)

```
1 const MIP_WMASK : UIntX = 'h0000_0000_0000_0222 as UIntX;
2 const MIE_WMASK : UIntX = 'h0000_0000_0000_02aa as UIntX;
```

▼ リスト 16.41: wmask に書き込みマスクを設定する (csrunit.veryl)

```
1 CsrAddr::MIP : MIP_WMASK,
```

`mip_reg` レジスタを作成します。`mip` の値を、`mip_reg` と ACLINT の状態を OR 演算したものに変更します（リスト 16.42）。

▼ リスト 16.42: レジスタを作成して変数に適用する (csrunit.veryl)

```

1 var mip_reg: UIntX;
2 let mip    : UIntX = mip_reg | {

```

**mip\_reg** レジスタのリセット、書き込みを実装します（リスト 16.43、リスト 16.44）。**wdata** には ACLINT の状態が含まれているので、書き込みマスクをもう一度適用します。

#### ▼ リスト 16.43: レジスタの値を 0 でリセットする (csrunit.veryl)

```

1 mie      = 0;
2 mip_reg = 0;
3 mcounteren = 0;

```

#### ▼ リスト 16.44: mip レジスタの書き込み (csrunit.veryl)

```

1 CsrAddr::MTVEC   : mtvec      = wdata;
2 CsrAddr::MIP     : mip_reg    = wdata & MIP_WMASK;
3 CsrAddr::MIE     : mie        = wdata;

```

### cause の設定

S-mode の割り込みの cause を設定します（リスト 16.45）。

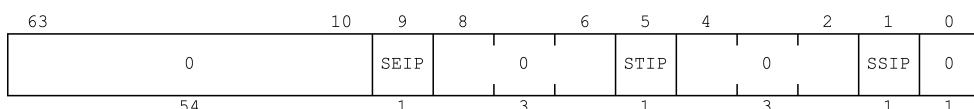
#### ▼ リスト 16.45: 割り込み原因の追加 (csrunit.veryl)

```

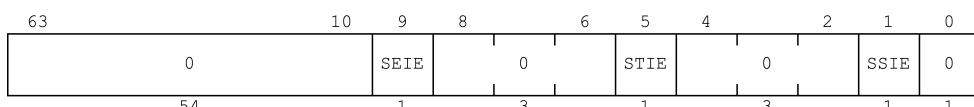
1 let interrupt_cause : UIntX = switch {
2     interrupt_pending[3]: CsrCause::MACHINE_SOFTWARE_INTERRUPT,
3     interrupt_pending[7]: CsrCause::MACHINE_TIMER_INTERRUPT,
4     interrupt_pending[9]: CsrCause::SUPERVISOR_EXTERNAL_INTERRUPT,
5     interrupt_pending[1]: CsrCause::SUPERVISOR_SOFTWARE_INTERRUPT,
6     interrupt_pending[5]: CsrCause::SUPERVISOR_TIMER_INTERRUPT,
7     default             : 0,
8 };

```

### medeleg、mideleg、sip、sie レジスタの実装



▲ 図 16.2: sip レジスタ



▲ 図 16.3: sie レジスタ

medeleg、mideleg、sip、sie レジスタを実装します。

medeleg、mideleg レジスタはそれぞれ委譲できる例外、割り込みに対応するビットだけ書き換えられるようにします。sip レジスタは mideleg レジスタで委譲された割り込みに対応するビットだけ値を参照できるように、sie レジスタは mideleg レジスタで委譲された割り込みに対応するビットだけ書き換えられるようにします。

レジスタを作成し、読み込めるようにします（リスト 16.46、リスト 16.47、リスト 16.48、リスト 16.49、リスト 16.50、リスト 16.51）。

#### ▼ リスト 16.46: medeleg、mideleg レジスタの定義 (csrunit.veryl)

```
1 var medeleg : UInt64;
2 var mideleg : UIntX ;
```

#### ▼ リスト 16.47: sip、sie レジスタの定義 (csrunit.veryl)

```
1 let sip : UIntX = mip & mideleg;
2 var sie : UIntX ;
```

#### ▼ リスト 16.48: medeleg、mideleg レジスタを 0 でリセットする (csrunit.veryl)

```
1 medeleg = 0;
2 mideleg = 0;
```

#### ▼ リスト 16.49: sie レジスタを 0 でリセットする (csrunit.veryl)

```
1 sie = 0;
```

#### ▼ リスト 16.50: rdata に medeleg、mideleg レジスタの値を割り当てる (csrunit.veryl)

```
1 CsrAddr::MEDELEG : medeleg,
2 CsrAddr::MIDELEG : mideleg,
```

#### ▼ リスト 16.51: rdata に sip、sie レジスタの値を割り当てる (csrunit.veryl)

```
1 CsrAddr::SIP : sip,
2 CsrAddr::SIE : sie & mideleg,
```

書き込みマスクを設定し、書き込めるようにします（リスト 16.52、リスト 16.53、リスト 16.54、リスト 16.55、リスト 16.56、リスト 16.57）。

#### ▼ リスト 16.52: 書き込みマスクの定義 (csrunit.veryl)

```
1 const MEDELEG_WMASK : UIntX = 'hffff_ffff_fffe_f7ff;
2 const MIDELEG_WMASK : UIntX = 'h0000_0000_0000_0222 as UIntX;
```

#### ▼ リスト 16.53: 書き込みマスクの定義 (csrunit.veryl)

```
1 const SIE_WMASK : UIntX = 'h0000_0000_0000_0222 as UIntX;
```

#### ▼ リスト 16.54: wmask に書き込みマスクを設定する (csrunit.veryl)

```
1 CsrAddr::MEDELEG : MEDELEG_WMASK,
2 CsrAddr::MIDELEG : MIDELEG_WMASK,
```

## ▼リスト 16.55: wmask に書き込みマスクを設定する (csrunit.veryl)

```
1   CsrAddr::SIE      : SIE_WMASK & mideleg,
```

## ▼リスト 16.56: medeleg、mideleg レジスタの書き込み (csrunit.veryl)

```
1   CsrAddr::MEDELEG : medeleg    = wdata;
2   CsrAddr::MIDELEG : mideleg    = wdata;
```

## ▼リスト 16.57: sie レジスタの書き込み (csrunit.veryl)

```
1   CsrAddr::SIE      : sie       = wdata;
```

## 16.4.8 割り込み条件、トランプの動作を変更する

作成した CSR を利用して、割り込みが発生する条件、トランプが発生したときの CSR の操作を変更します。

例外が発生するとき、遷移先の特権レベルを medeleg レジスタによって変更します（リスト 16.58）。

## ▼リスト 16.58: 例外の遷移先の特権レベルを求める (csrunit.veryl)

```
1   let expt_mode : PrivMode = if mode == PrivMode::M || !medeleg[expt_cause[5:0]] ? PrivMode::>
>M : PrivMode::S;
```

割り込みの発生条件と参照する CSR を、遷移先の特権レベルごとに用意します（リスト 16.59、リスト 16.60）。

## ▼リスト 16.59: M-mode に遷移する割り込みを示す変数 (csrunit.veryl)

```
1   // Interrupt to M-mode
2   let interrupt_pending_mmode: UIntX = mip & mie & ~mideleg;
3   let raise_interrupt_mmode : logic = (mode != PrivMode::M || mstatus_mie) && interrupt_pendi>
>ng_mmode != 0;
4   let interrupt_cause_mmode : UIntX = switch {
5       interrupt_pending_mmode[3]: CsrCause::MACHINE_SOFTWARE_INTERRUPT,
6       interrupt_pending_mmode[7]: CsrCause::MACHINE_TIMER_INTERRUPT,
7       interrupt_pending_mmode[9]: CsrCause::SUPERVISOR_EXTERNAL_INTERRUPT,
8       interrupt_pending_mmode[1]: CsrCause::SUPERVISOR_SOFTWARE_INTERRUPT,
9       interrupt_pending_mmode[5]: CsrCause::SUPERVISOR_TIMER_INTERRUPT,
10      default                  : 0,
11  };
```

## ▼リスト 16.60: S-mode に遷移する割り込みを示す変数 (csrunit.veryl)

```
1   // Interrupt to S-mode
2   let interrupt_pending_smode: UIntX = sip & sie;
3   let raise_interrupt_smode : logic = (mode <: PrivMode::S || (mode == PrivMode::S && mstatus>_sie)) && interrupt_pending_smode != 0;
4   let interrupt_cause_smode : UIntX = switch {
5       interrupt_pending_smode[9]: CsrCause::SUPERVISOR_EXTERNAL_INTERRUPT,
6       interrupt_pending_smode[1]: CsrCause::SUPERVISOR_SOFTWARE_INTERRUPT,
7       interrupt_pending_smode[5]: CsrCause::SUPERVISOR_TIMER_INTERRUPT,
```

```

8     default          : 0,
9 };

```

M-mode 向けの割り込みを優先して利用します (リスト 16.61)。

▼ リスト 16.61: M-mode、S-mode に遷移する割り込みを調停する (csrunit.veril)

```

1 // Interrupt
2 let raise_interrupt : logic = valid && can_intr && (raise_interrupt_mmode || raise_interrupt>
>_smode);
3 let interrupt_cause : UIntX = if raise_interrupt_mmode ? interrupt_cause_mmode : interrupt_c>
>ause_smode;
4 let interrupt_xtvec : Addr = if interrupt_mode == PrivMode::M ? mtvec : stvec;
5 let interrupt_vector: Addr = if interrupt_xtvec[0] == 0 ?
6     {interrupt_xtvec[msb:2], 2'b0}
7 : // Direct
8     {interrupt_xtvec[msb:2] + interrupt_cause[msb - 2:0], 2'b0}
9 ; // Vectored
10 let interrupt_mode: PrivMode = if raise_interrupt_mmode ? PrivMode::M : PrivMode::S;

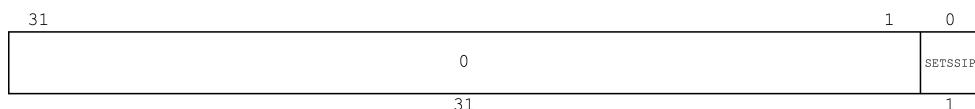
```

## 16.5 ソフトウェア割り込みの実装 (SSWI)

SSWI デバイスはソフトウェア割り込み (Supervisor software interrupt) を提供するためのデバイスです。SSWI デバイスにはハードウェアスレッド毎に 4 バイトの SETSSIP レジスタが用意されています (表 16.1) SETSSIP レジスタを読み込むと常に 0 を返しますが、最下位ビットに 1 を書き込むとそれに対応するハードウェアスレッドの mip.SSIP ビットが 1 になります。

▼ 表 16.1: SSWI デバイスのメモリマップ

オフセット	レジスタ
0000	SETSSIP0
0004	SETSSIP1
..	..
3ff8	SETSSIP4094
3ffc	MTIME



▲ 図 16.4: setssip レジスタ

今のところ mhartid が 0 のハードウェアスレッドしか存在しないため、SETSSIP0 のみ実装します。aclint\_if インターフェースに、mip レジスタの SSIP ビットを 1 にする要求のため

の `setssip` を作成します ( リスト 16.62 )。

▼ リスト 16.62: `setssip` をインターフェースに追加する (`aclint_if.veryl`)

```

1 interface aclint_if {
2     var msip    : logic ;
3     var mtip    : logic ;
4     var mtime   : UInt64;
5     var setssip: logic ;
6     modport master {
7         msip    : output,
8         mtip    : output,
9         mtime   : output,
10        setssip: output,
11    }

```

`aclint` モジュールで `SETSSIP0` への書き込みを検知し、最下位ビットを `setssip` に接続します ( リスト 16.63 )。

▼ リスト 16.63: `SETSSIP0` に書き込むとき `setssip` に LSB を割り当てる (`aclint_memory.veryl`)

```

1 always_comb {
2     aclint.setssip = 0;
3     if membus.valid && membus.wen && membus.addr == MMAP_ACLINT_SETSSIP {
4         aclint.setssip = membus.wdata[0];
5     }
6 }

```

`csrunit` モジュールで `setssip` を確認し、`mip.SSIP` を立てるようになります ( リスト 16.64、リスト 16.65、リスト 16.66 )。

▼ リスト 16.64: `setssip` を `XLEN` ビットに拡張する (`csrunit.veryl`)

```

1 let setssip: UIntX = {1'b0 repeat XLEN - 2, aclint.setssip, 1'b0};

```

▼ リスト 16.65: `setssip` で `mip` を更新する (`csrunit.veryl`)

```

1 } else {
2     mcycle  += 1;
3     mip_reg |= setssip;

```

▼ リスト 16.66: `setssip` で `mip` を更新する (`csrunit.veryl`)

```

1 CsrAddr::MIP      : mip_reg      = (wdata & MIP_WMASK) | setssip;

```

# 第 17 章

## S-mode の実装 (2. 仮想記憶システム)

### 17.1 概要

#### 17.1.1 仮想記憶システム

仮想記憶 (Virtual Memory) とは、メモリを管理する手法の一種です。仮想的なアドレス (virtual address、仮想アドレス) を実際のアドレス (real address、実アドレス) に変換することにより、実際のアドレス空間とは異なるアドレス空間を提供することができます。実アドレスのことを物理アドレス (physical address) と呼ぶことがあります。

仮想記憶を利用すると、次のような動作を実現できます。

1. 連続していない物理アドレス空間を仮想的に連続したアドレス空間として扱う。
2. 特定のアドレスにしか配置できない (特定のアドレスで動くことを前提としている) プログラムを、そのアドレスとは異なる物理アドレスに配置して実行する。
3. アプリケーションごとにアドレス空間を分離する。

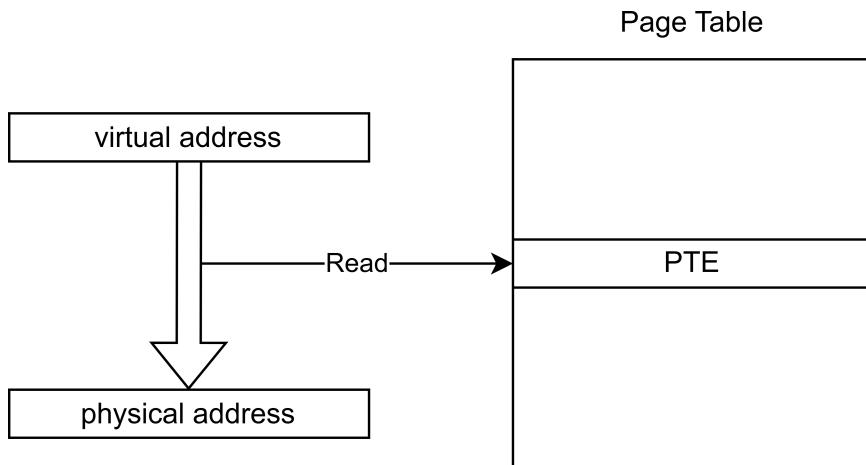
一般的に仮想記憶システムはハードウェアによって提供されます。メモリアクセスを処理するハードウェア部品のことをメモリ管理ユニット (Memory Management Unit, MMU) と呼びます。

#### 17.1.2 ページング方式

仮想記憶システムを実現する方式の 1 つにページング方式 (Paging) があります。ページング方式は、物理アドレス空間の一部をページ (Page) という単位に割り当て、ページを参照するための情報をページテーブル (Page Table) に格納します。ページテーブルに格納する情報の単位のことをページテーブルエントリ (Page Table Entry、PTE) と呼びます。仮想アドレスから物理アドレスへの変換はページテーブルにある PTE を参照して行います (図 17.1)。

#### 17.1.3 RISC-V の仮想記憶システム

RISC-V の仮想記憶システムはページング方式を採用しており、RV32I 向けには Sv32、RV64I



▲図 17.1: 仮想アドレスの変換に PTE を使う

向けには Sv39、Sv48、Sv57 が定義されています。

RISC-V の仮想アドレスの変換を簡単に説明します。仮想アドレスの変換は次のプロセスで行います。

(a) satp レジスタの PPN フィールドと仮想アドレスのフィールドから PTE の物理アドレスを作る。(b) PTE を読み込む。PTE が有効なものか確認する。(c) PTE がページを指しているとき、PTE に書かれている権限を確認してから物理アドレスを作り、アドレス変換終了。(d) PTE が次の PTE を指しているとき、PTE のフィールドと仮想アドレスのフィールドから次の PTE の物理アドレスを作り、(b) に戻る。

satp レジスタは仮想記憶システムを制御するための CSR です。一番最初に参照する PTE のことを root PTE と呼びます。また、PTE がページを指しているとき、その PTE のことを leaf PTE と呼びます。

RISC-V のページングでは、satp レジスタと仮想アドレス、PTE を使って多段階の PTE の参照を行い、仮想アドレスを物理アドレスに変換します。Sv39 の場合、何段階で物理アドレスに変換できるかによってページサイズは 4KiB、2MiB、1GiB と異なります。これ以降、MMU 内のページング方式を実現する部品のことを PTW(Page Table Walker) と呼びます<sup>\*1</sup>。

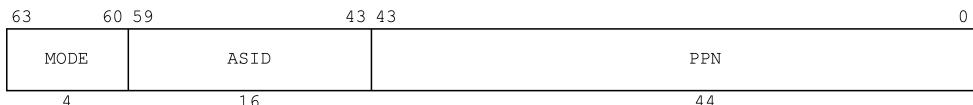
## 17.2 satp レジスタ

RISC-V の仮想記憶システムは satp レジスタによって制御します。

MODE は仮想アドレスの変換方式を指定するフィールドです。方式と値は表 17.1 のように対

---

<sup>\*1</sup> ページテーブルをたどってアドレスを変換するので Page Table Walker と呼びます。アドレスを変換することを Page Table Walk と呼ぶこともあります。



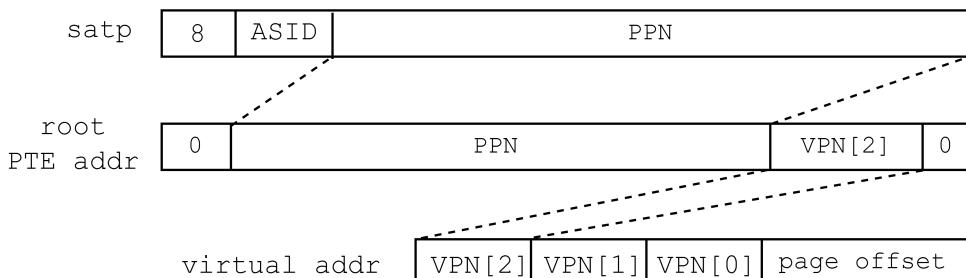
▲図 17.2: satp レジスタ

応しています。方式が Bare(0) のときはアドレス変換を行いません(仮想アドレス=物理アドレス)。

▼表 17.1: 方式と MODE の値の対応

方式	MODE
Bare	0
Sv39	8
Sv48	9
Sv57	10

ASID(Address Space IDentifier) は仮想アドレスが属するアドレス空間の ID です。動かすアプリケーションによって ID を変えることで MMU にアドレス変換の高速化のヒントを与えることができます。本章ではキャッシュ機構を持たない単純なモジュールを実装するため、ASID を無視したアドレス変換を実装します\*2。



▲図 17.3: root PTE のアドレスは satp レジスタと仮想アドレスから構成される

PPN(Physical Page Number) は root PTE の物理アドレスの一部を格納するフィールドです。root PTE のアドレスは仮想アドレスの VPN ビットと組み合わせて作られます(図 17.3)。

## 17.3 Sv39 のアドレス変換

Sv39 では 39 ビットの仮想アドレスを 56 ビットの物理アドレスに変換します。

\*2 PTW はページエントリをキャッシュすることで高速化できます。ASID が異なるときのキャッシュは利用することができません。キャッシュ機構 (TLB) は応用編で実装します。

39	30 29	21 20	12 11	0
VPN [2]	VPN [1]	VPN [0]	page offset	
9	9	9	12	

▲図 17.4: 仮想アドレス

55	30 29	21 20	12 11	0
PPN [2]	PPN [1]	PPN [0]	page offset	
26	9	9	12	

▲図 17.5: 物理アドレス

ページの最小サイズは  $4096(2^{**}12)$  バイト、PTE のサイズは  $8(2^{**}3)$  バイトです。それぞれ 12 と 8 を PAGESIZE、PTESIZE という定数として定義します。

ページテーブルのサイズ (1 つのページテーブルに含まれる PTE の数) は  $512(=2^{**}9)$  個です。1 回のアドレス変換で、最大 3 回 PTE をフェッチし、leaf PTE を見つけます。

アドレスの変換途中で PTE が不正な値だったり、ページが求める権限を持たずにページにアクセスしようとした場合、アクセスする目的に応じたページフォルト (Page fault) 例外が発生します<sup>\*3</sup>。命令フェッチは Instruction page fault 例外、ロード命令は Load page fault 例外、ストアと AMO 命令は Store/AMO page fault 例外が発生します。

### 17.3.1 ページングが有効になる条件

satp レジスタの MODE フィールドが Sv39 のとき、S-mode、U-mode でアドレス変換が有効になります。ただし、ロードストアのときは、mstatus.MPRV が 1 なら特権レベルを mstatus.MPP として判定します。

有効な仮想アドレスは、MSB で XLEN ビットに拡張された値である必要があります。有効ではない仮想アドレスの場合、ページフォルト例外が発生します。

### 17.3.2 PTE のフェッチ

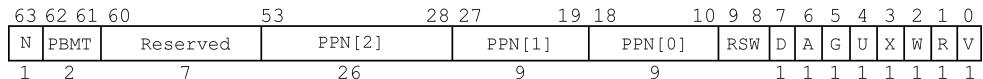
55	12 11	3 2	0
PPN	VPN [i]		
44	9	3	

▲図 17.6: PTE のアドレス

ページングが有効なとき、まず root PTE をフェッチします。ここで level という変数の値を 2 とします。

<sup>\*3</sup> RISC-V の MMU は PMP、PMA という仕組みで物理アドレス空間へのアクセスを制限することができ、それに違反した場合にアクセスフォルト例外を発生させます。本章では PMP、PMA を実装していないのでアクセスフォルト例外に関する機能について説明せず、実装もしません。これらの機能は応用編で実装します。

root PTE の物理アドレスは、satp レジスタの PPN フィールドと仮想アドレスの `VPN[level]` フィールドを結合し、`log2(PTESIZE)` だけ左シフトしたアドレスになります。このアドレスは、PPN フィールドを 12 ビット左シフトしたアドレスに存在するページテーブルの、`VPN[level]` 番目の PTE のアドレスです。



▲図 17.7: PTE のフィールド

PTE のフィールドは図 17.7 のようになっています。このうち N、PBMT、Reserved は使用せず、0 でなければページフォルト例外を発生させます。RSW ビットは無視します。

下位 8 ビットは PTE の状態と権限を表すビットです。

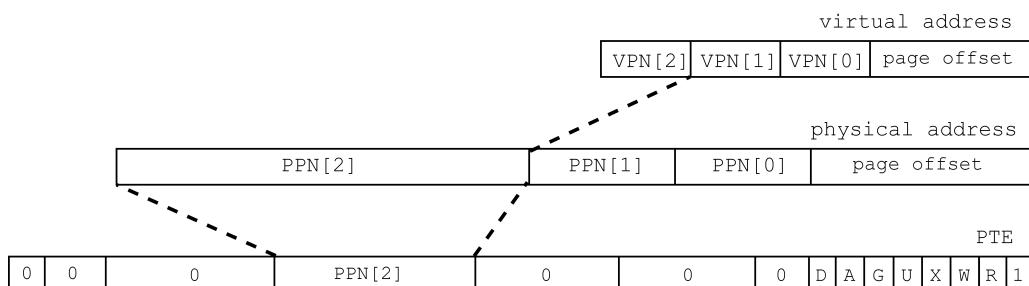
V が 1 のとき、有効な PTE であることを示します。0 ならページフォルト例外を発生させます。

R、W、X、U はページの権限を指定するビットです。R は読み込み許可、W は書き込み許可、X は実行許可、U は U-mode でアクセスできるかを示します。書き込みできる PTE は読み込みできる必要があります、W が 1 なのに R が 0 ならページフォルト例外を発生させます。

R と X が 0 のとき、PTE は次の PTE を指しています。このとき、level が 0 ならこれ以上 PTE を指すことはできない (`VPN[-1]` は無い) ので、ページフォルト例外を発生させます。level が 1 以上なら、level から 1 を引いて PTE をフェッチします。次の PTE のアドレスは、PTE の PPN[2]、PPN[1]、PPN[0] と仮想アドレスの `VPN[level]` を結合し、`log2(PTESIZE)` だけ左シフトしたアドレスになります。

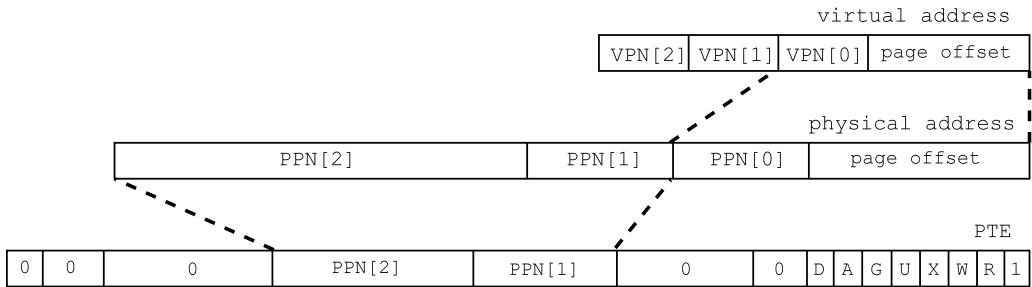
PTE の R か X が 1 のとき、PTE は leaf PTE で、ページを指し示しています。

物理アドレスを計算する前に、R、W、X、U ビットで権限を確認します。命令フェッチのときは X、ロードのときは R、ストアのときは W、U-mode のときは U が立っている必要があります。S-mode のときは、U が立っているページに mstatus.SUM が 0 の状態でアクセスできません。S-mode のときは、U が立っているページの実行はできません。これらに違反した場合、ページフォルト例外が発生します。



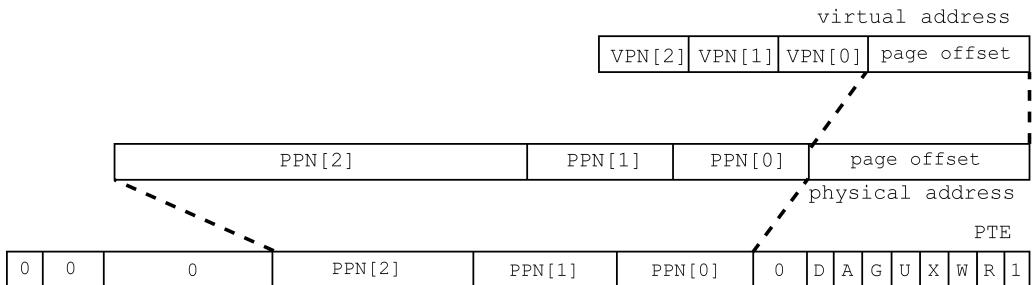
▲図 17.8: level が 2 のときの物理アドレス

level が 2 なら、物理アドレスは PTE の PPN[2]、仮想アドレスの VPN[1]、VPN[0]、page offset を結合した値になります（図 17.8）。



▲図 17.9: level が 1 のときの物理アドレス

level が 1 なら、物理アドレスは PTE の PPN[2]、PPN[1]、仮想アドレスの VPN[0]、page offset を結合した値になります（図 17.9）。



▲図 17.10: level が 0 のときの物理アドレス

level が 0 なら、物理アドレスは PTE の PPN[2]、PPN[1]、PPN[0]、仮想アドレスの page offset を結合した値になります（図 17.10）。

leaf PTE の使わない PPN フィールドは 0 である必要があり、0 ではないならページ fault 例外を発生させます。

求めた物理アドレスにアクセスする前に、leaf PTE の A、D ビットを確認します。A はページがこれまでにアクセスされたか、D はページがこれまでに書き換えられたかを示すビットです。A が 0 のとき、A を 1 に設定します。D が 0 でストアするとき、D を 1 に設定します。A は投機的に 1 に変更できますが、D は命令が実行された場合にしか 1 に変更できません。

## 17.4 実装順序

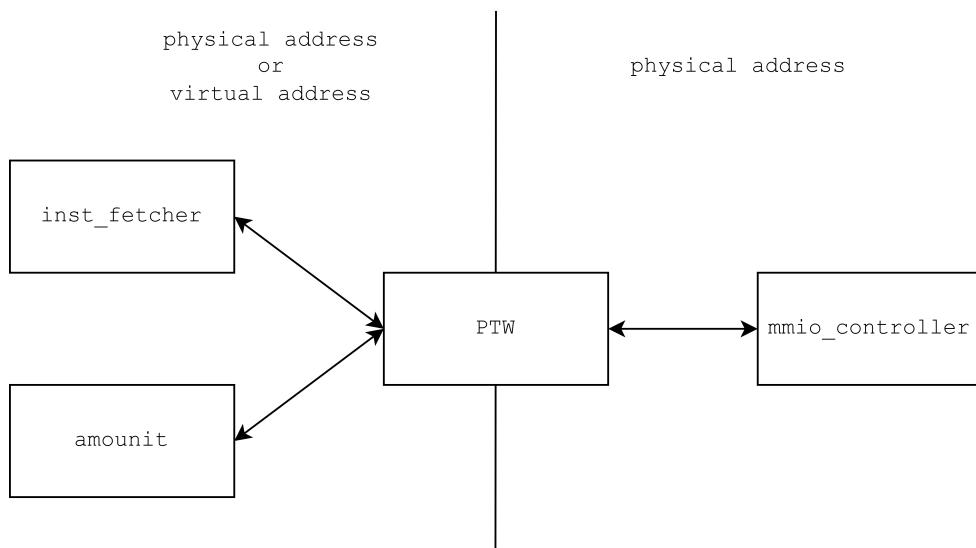
RISC-V では命令フェッチ、データのロードストアの両方でページングを利用できます。命令フェッチ、データのロードストアのそれぞれのために 2 つの PTW を用意してもいいですが、シン

フルなアーキテクチャにするために本章では1つのPTWを共有することにします。

`inst_fetcher` モジュール、`amounit` モジュールは仮想アドレスを扱うことがあります、`mmio_controller` モジュールは常に物理アドレス空間を扱います。そのため、`inst_fetcher` モジュール、`amounit` モジュールと `mmio_controller` モジュールの間にPTWを配置します(図17.11)。

本章では、仮想記憶システムを次の順序で実装します。

1. PTWで発生する例外を `csrunit` モジュールに伝達する
2. Bareにだけ対応したアドレス変換モジュール(ptw)を実装する
3. `satp` レジスタ、`mstatus` の MXR、SUM、MPRV ビットを実装する
4. `Sv39` を実装する
5. `SFENCE.VMA` 命令、`FENCEI` 命令を実装する



▲図 17.11: PTW と他のモジュールの接続

## 17.5 メモリで発生する例外の実装

PTWで発生した例外は、最終的に `csrunit` モジュールで処理します。そのため、例外の情報をメモリのインターフェースを使って伝達します。

ページングによって発生する例外の cause を `CsrCause` 型に追加します(リスト 17.1)。

### ▼リスト 17.1: CsrCause 型にページフォルト例外を追加する(eei.vetyl)

```

1 INSTRUCTION_PAGE_FAULT = 12,
2 LOAD_PAGE_FAULT = 13,
  
```

```
3 STORE_AMO_PAGE_FAULT = 15,
```

## 17.5.1 例外を伝達する

### 構造体の定義

`MemException` 構造体を定義します (リスト 17.2)。メモリアクセス中に発生する例外の情報はこの構造体で管理します。

#### ▼ リスト 17.2: MemException 型の定義 (eei.veryl)

```
1 struct MemException {
2     valid      : logic,
3     page_fault: logic,
4 }
```

`membus_if`、`core_data_if`、`core_inst_if` インターフェースに `MemException` 構造体を追加します (リスト 17.3、リスト 17.4、リスト 17.5、リスト 17.6)。インターフェースの `rvalid` が 1 で、構造体の `valid` と `is_page_fault` が 1 ならページフォルト例外が発生したことを示します。

#### ▼ リスト 17.3: MemException 型を追加する (membus\_if.veryl, core\_data\_if.veryl, core\_inst\_if.veryl)

```
1 var expt : eei::MemException ;
```

#### ▼ リスト 17.4: master に expt を追加する (membus\_if.veryl, core\_data\_if.veryl, core\_inst\_if.veryl)

```
1 modport master {
2     ...
3     expt      : input ,
4     ...
5 }
```

#### ▼ リスト 17.5: slave に expt を追加する (membus\_if.veryl)

```
1 modport slave {
2     ...
3     expt      : output,
4     ...
5 }
```

#### ▼ リスト 17.6: response に expt を追加する (membus\_if.veryl)

```
1 modport response {
2     rvalid: output,
3     rdata : output,
4     expt  : output,
5 }
```

### mmio\_controller モジュールの対応

`mmio_controller` モジュールで構造体の値をすべて 0 に設定します (リスト 17.7、リスト 17.8)。いまのところ、デバイスは例外を発生させません。

## ▼リスト 17.7: expt を 0 に設定する (membus\_if.veryl)

```

1  always_comb {
2      req_core.ready  = 0;
3      req_core.rvalid = 0;
4      req_core.rdata  = 0;
5      req_core.expt  = 0;

```

mmio\_controller モジュールからの例外情報を `core_data_if`、`core_inst_if` インターフェースに伝達します。

## ▼リスト 17.8: expt を伝達する (top.veryl)

```

1  always_comb {
2      i_membus.ready  = mmio_membus.ready && !d_membus.valid;
3      i_membus.rvalid = mmio_membus.rvalid && memarb_last_i;
4      i_membus.rdata  = mmio_membus.rdata;
5      i_membus.expt = mmio_membus.expt;
6
7      d_membus.ready  = mmio_membus.ready;
8      d_membus.rvalid = mmio_membus.rvalid && !memarb_last_i;
9      d_membus.rdata  = mmio_membus.rdata;
10     d_membus.expt = mmio_membus.expt;

```

**inst\_fetcher** モジュールの対応

`inst_fetcher` モジュールから core モジュールに例外情報を伝達します。まず、FIFO の型に例外情報を追加します（リスト 17.9、リスト 17.10）。

## ▼リスト 17.9: fetch\_fifo\_type に MemException 型を追加する (inst\_fetcher.veryl)

```

1  struct fetch_fifo_type {
2      addr: Addr           ,
3      bits: logic          <MEMBUS_DATA_WIDTH>,
4      expt: MemException   ,
5  }

```

## ▼リスト 17.10: issue\_fifo\_type に MemException 型を追加する (inst\_fetcher.veryl)

```

1  struct issue_fifo_type {
2      addr : Addr        ,
3      bits : Inst         ,
4      is_rvc: logic       ,
5      expt : MemException,
6  }

```

メモリからの例外情報を `fetch_fifo` に保存します（リスト 17.11）。

## ▼リスト 17.11: メモリの例外情報を fetch\_fifo に保存する (inst\_fetcher.veryl)

```

1  always_comb {
2      fetch_fifo_flush    = core_if.is_hazard;
3      fetch_fifo_wvalid   = fetch_requested && mem_if.rvalid;
4      fetch_fifo_wdata.addr = fetch_pc_requested;

```

```

5     fetch_fifo_wdata.bits = mem_if.rdata;
6     fetch_fifo_wdata.expt = mem_if.expt;
7 }
```

`fetch_fifo` から `issue_fifo` に例外情報を伝達します (リスト 17.12)、リスト 17.13、リスト 17.14)。offset が 6 で例外が発生しているとき、32 ビット幅の命令の上位 16 ビットを取得せずにすぐに `issue_fifo` に例外を書き込みます。

#### ▼ リスト 17.12: `fetch_fifo` から `issue_fifo` に例外情報を伝達する (`inst_fetcher.veryl`)

```

1 always_comb {
2     let raddr : Addr                      = fetch_fifo_rdata.addr;
3     let rdata : logic <MEMBUS_DATA_WIDTH> = fetch_fifo_rdata.bits;
4     let expt : MemException               = fetch_fifo_rdata.expt;
5     let offset: logic <3>                 = issue_pc_offset;
6
7     fetch_fifo_rready      = 0;
8     issue_fifo_wvalid     = 0;
9     issue_fifo_wdata      = 0;
10    issue_fifo_wdata.expt = expt;
```

▼ リスト 17.13: offset が 6 のときに例外が発生している場合、すぐに `issue_fifo` に例外を書き込む (`inst_fetcher.veryl`)

```

1 fetch_fifo_rready = 1;
2 if rvcc_is_rvc || expt.valid {
3     issue_fifo_wvalid      = 1;
4     issue_fifo_wdata.addr  = {raddr[msb:3], offset};
5     issue_fifo_wdata.is_rvc = 1;
6     issue_fifo_wdata.bits  = rvcc_inst32;
```

▼ リスト 17.14: 例外が発生しているときは 32 ビット幅の命令の上位 16 ビットを取得しない (`inst_fetcher.veryl`)

```

1 if issue_pc_offset == 6 && !rvcc_is_rvc && !issue_is_rdata_saved && !fetch_fifo_rdata.expt.v
2 alid {
3     if fetch_fifo_rvalid {
4         issue_is_rdata_saved = 1;
```

`issue_fifo` から core モジュールに例外情報を伝達します (リスト 17.15)。

#### ▼ リスト 17.15: `issue_fifo` から core モジュールに例外情報を伝達する (`inst_fetcher.veryl`)

```

1 always_comb {
2     issue_fifo_flush  = core_if.is_hazard;
3     issue_fifo_rready = core_if.rready;
4     core_if.rvalid    = issue_fifo_rvalid;
5     core_if.raddr     = issue_fifo_rdata.addr;
6     core_if.rdata     = issue_fifo_rdata.bits;
7     core_if.is_rvc   = issue_fifo_rdata.is_rvc;
8     core_if.expt     = issue_fifo_rdata.expt;
9 }
```

## amounit モジュールの対応

`state` が `State::Init` 以外の時に例外が発生した場合、すぐに結果を返すようにします（リスト 17.16、リスト 17.17、リスト 17.18、）。例外が発生したクロックでは要求を受け付けないようにします。

### ▼ リスト 17.16: slave に expt を割り当てる (amounit.veryl)

```

1  always_comb {
2      slave.ready  = 0;
3      slave.rvalid = 0;
4      slave.rdata  = 0;
5      slave.expt   = master.expt;

```

### ▼ リスト 17.17: 例外が発生したらすぐに結果を返し、ready を 0 にする (amounit.veryl)

```

1      default: {}
2  }
3
4  if state != State::Init && master.expt.valid {
5      slave.ready  = 0;
6      slave.rvalid = 1;
7  }
8

```

### ▼ リスト 17.18: 例外が発生していたら master に要求するのをやめる (amounit.veryl)

```

1      State::AMOStoreValid: accept_request_comb();
2      default           : {}
3  }
4
5  if state != State::Init && master.expt.valid {
6      reset_master();
7  }
8

```

例外が発生したら、`state` を `State::Init` にリセットします（リスト 17.19）。

### ▼ リスト 17.19: 例外が発生していたら state を Init にリセットする (amounit.veryl)

```

1  function on_clock () {
2      if state != State::Init && master.expt.valid {
3          state = State::Init;
4      } else {
5          case state {
6              State::Init    : accept_request_ff();

```

## Instruction page fault 例外の実装

命令フェッチ処理中にページフォルト例外が発生していたとき、Instruction page fault 例外を発生させます。xtval には例外が発生したアドレスを設定します（リスト 17.20）。

## ▼リスト 17.20: i\_membus の例外を ExceptionInfo 型に設定する (core.veryl)

```

1  if i_membus.expt.valid {
2      // fault
3      exq_wdata.expt.valid = 1;
4      exq_wdata.expt.cause = CsrCause::INSTRUCTION_PAGE_FAULT;
5      exq_wdata.expt.value = ids_pc;
6  } else if !ids_inst_valid {

```

## ロード、ストア命令の page fault 例外の実装

ロード命令、ストア命令、A 拡張の命令のメモリアクセス中にページフォルト例外が発生していたとき、Load page fault 例外、Store/AMO page fault 例外を発生させます。

csrunit モジュールに、メモリにアクセスする命令の例外情報を監視するためのポートを作成します（リスト 17.21、リスト 17.22）。

## ▼リスト 17.21: メモリアドレス、例外の監視用のポートを追加する (csrunit.veryl)

```

1 module csrunit (
2     ...
3     can_intr    : input    logic          ,
4     mem_addr    : input    Addr          ,
5     rdata       : output   UIntX        ,
6     ...
7     membuss     : modport core_data_if::master    ,
8 );

```

## ▼リスト 17.22: csrunit モジュールにメモリアドレスとインターフェースを割り当てる (core.veryl)

```

1 inst csru: csrunit (
2     ...
3     mem_addr    : memu_addr          ,
4     ...
5     membuss     : d_membus          ,
6 );

```

例外を発生させます（リスト 17.23、リスト 17.24）。

## ▼リスト 17.23: メモリアクセス中に例外が発生しているかをチェックする (csrunit.veryl)

```

1 let expt_memory_fault    : logic = membuss.rvalid && membuss.expt.valid;

```

## ▼リスト 17.24: 例外を発生させる (csrunit.veryl)

```

1 let raise_expt: logic = valid && (expt_info.valid || expt_write_READONLY_CSR || expt_CSR_PRI_VIOLATION || expt_ZICNTR_PRIV || expt_trap_RETURN_PRIV || expt_MEMORY_FAULT);
2 let expt_cause: UIntX = switch {
3     ...
4     expt_MEMORY_FAULT    : if ctrl.is_load ? CsrCause::LOAD_PAGE_FAULT : CsrCause::STORE_AMO_PAGE_FAULT,
5     default               : 0,
6 };

```

xtval に例外が発生したアドレスを設定します（リスト 17.25）。

## ▼ リスト 17.25: 例外の原因を設定する (csrunit.veryl)

```

1 let expt_value: UIntX = switch {
2     expt_info.valid           : expt_info.value,
3     expt_cause == CsrCause::ILLEGAL_INSTRUCTION : {1'b0 repeat XLEN - $bits(Inst), inst_bits}
4 },
5     expt_cause == CsrCause::LOAD_PAGE_FAULT      : mem_addr,
6     expt_cause == CsrCause::STORE_AMO_PAGE_FAULT: mem_addr,
7     default                                : 0
7 };

```

## 17.5.2 ページフォルトが発生した正確なアドレスを特定する

ページフォルト例外が発生したとき、`xtval` にはページフォルトが発生した仮想アドレスを格納します。

実は現状の実装では、メモリにアクセスする操作がページの境界をまたぐとき、ページフォルトが発生した正確な仮想アドレスを `xtval` に格納できません。

例えば、`inst_fetcher` モジュールで 32 ビット幅の命令を 2 回のメモリ読み込みでフェッチするとき、1 回目 (下位 16 ビット) のロードは成功して、2 回目 (上位 16 ビット) のロードでページフォルトが発生したとします。このとき、ページフォルトが発生したアドレスは 2 回目のロードでアクセスしたアドレスなのに、`xtval` には 1 回目のロードでアクセスしたアドレスが書き込まれます。

これに対処するために、例外が発生したアドレスのオフセットを例外情報に追加します (リスト 17.26)。

▼ リスト 17.26: MemException 型に `addr_offset` を追加する (eei.veryl)

```

1 struct MemException {
2     valid      : logic  ,
3     page_fault : logic  ,
4     addr_offset: logic<3>,
5 }

```

`inst_fetcher` モジュールで、32 ビット幅の命令の上位 16 ビットを読み込んで `issue_fifo` に書き込むときに、オフセットを 2 に設定します (リスト 17.27)。

## ▼ リスト 17.27: オフセットを 2 に設定する (inst\_fetcher.veryl)

```

1 if issue_is_rdata_saved {
2     issue_fifo_wvalid        = 1;
3     issue_fifo_wdata.addr    = {issue_saved_addr[msb:3], offset};
4     issue_fifo_wdata.bits    = {rdata[15:0], issue_saved_bits};
5     issue_fifo_wdata.is_rvc = 0;
6     issue_fifo_wdata.expt.addr_offset = 2;

```

`xtval` を生成するとき、オフセットを足します (リスト 17.28、リスト 17.29)。

## ▼ リスト 17.28: 命令アドレスにオフセットを足す (core.veryl)

```

1 exq_wdata.expt.valid = 1;
2 exq_wdata.expt.cause = CsrCause::INSTRUCTION_PAGE_FAULT;
3 exq_wdata.expt.value = ids_pc + {1'b0 repeat XLEN - 3, i_membus.expt.addr_offset};

```

## ▼ リスト 17.29: ロードストア命令のメモリアドレスにオフセットを足す (csrunit.veryl)

```

1 let expt_value: UIntX = switch {
2     expt_info.valid : expt_info.value,
3     expt_cause == CsrCause::ILLEGAL_INSTRUCTION : {1'b0 repeat XLEN - $bits(Inst), inst_bits} >,
4     expt_cause == CsrCause::LOAD_PAGE_FAULT : mem_addr + {1'b0 repeat XLEN - 3, membust.xpt.addr_offset},
5     expt_cause == CsrCause::STORE_AMO_PAGE_FAULT: mem_addr + {1'b0 repeat XLEN - 3, membust.xpt.addr_offset},
6     default : 0
7 };

```

## 17.6 satp レジスタの作成

satp レジスタを実装します（リスト 17.30、リスト 17.31、リスト 17.32、リスト 17.33、リスト 17.34）。すべてのフィールドを読み書きできるように設定して、値を `0` でリセットします。

## ▼ リスト 17.30: satp レジスタを作成する (csrunit.veryl)

```

1 var satp : UIntX ;

```

## ▼ リスト 17.31: satp レジスタを 0 でリセットする (csrunit.veryl)

```

1 satp = 0;

```

## ▼ リスト 17.32: rdata に satp レジスタの値を設定する (csrunit.veryl)

```

1 CsrAddr::SATP : satp,

```

## ▼ リスト 17.33: 書き込みマスクの定義 (csrunit.veryl)

```

1 const SATP_WMASK : UIntX = 'hffff_ffff_ffff_ffff;

```

## ▼ リスト 17.34: wmask に書き込みマスクを設定する (csrunit.veryl)

```

1 CsrAddr::SATP : SATP_WMASK,

```

satp レジスタは、MODE フィールドに書き込もうとしている値がサポートしない MODE なら、satp レジスタの変更を全ビットについて無視すると定められています。

本章では Bare と Sv39 だけをサポートするため、MODE には `0` と `8` のみ書き込めるようにして、それ以外の値を書き込もうとしたら satp レジスタへの書き込みを無視します（リスト 17.35、リスト 17.36）。

## ▼リスト 17.35: sat に書き込む値を生成する関数 (csrunit.veryl)

```

1  function validate_satp (
2      satp : input UIntX,
3      wdata: input UIntX,
4  ) -> UIntX {
5      // mode
6      if wdata[msb-4] != 0 && wdata[msb-4] != 8 {
7          return satp;
8      }
9      return wdata;
10 }
```

## ▼リスト 17.36: satp レジスタに書き込む (csrunit.veryl)

```
1  CsrAddr::SATP      : satp      = validate_satp(satp, wdata);
```

## 17.7 mstatus の MXR、SUM、MPRV ビットの実装

mstatus レジスタの MXR、SUM、MPRV ビットを変更できるようにします（リスト 17.37、リスト 17.38）。

## ▼リスト 17.37: 書き込みマスクの変更 (csrunit.veryl)

```
1  const MSTATUS_WMASK : UIntX = 'h0000_0000_006e_19aa as UIntX;
```

## ▼リスト 17.38: 書き込みマスクの変更 (csrunit.veryl)

```
1  const SSTATUS_WMASK : UIntX = 'h0000_0000_000c_0122 as UIntX;
```

それぞれのビットを示す変数を作成します（リスト 17.39、リスト 17.40）。

## ▼リスト 17.39: mstatus の MXR、SUM、MPRV ビットを示す変数を作成する (csrunit.veryl)

```

1  let mstatus_mxr : logic    = mstatus[19];
2  let mstatus_sum : logic    = mstatus[18];
3  let mstatus_mprv: logic    = mstatus[17];
```

mstatus.MPRV は、M-mode 以外のモードに戻るときに 0 に設定されると定められています。そのため、trap\_mode\_next を確認して 0 を設定します。

## ▼リスト 17.40: mstatus.MPRV を MRET、SRET 命令で 0 に設定する (csrunit.veryl)

```

1  } else if trap_return {
2      // set mstatus.mprv = 0 when new mode != M-mode
3      if trap_mode_next <: PrivMode::M {
4          mstatus[17] = 0;
5      }
6      if is_mret {
```

## 17.8 アドレス変換モジュール (PTW) の実装

ページテーブルエントリをフェッチしてアドレス変換を行う ptw モジュールを作成します。まず、MODE が Bare のとき (仮想アドレス = 物理アドレス) の動作を実装し、Sv39 を「17.9 Sv39 の実装」(p.340) で実装します。

### 17.8.1 CSR のインターフェースを実装する

ページングで使用する CSR を、csrunit モジュールから ptw モジュールに渡すためのインターフェースを定義します。

`src/ptw_ctrl_if.veryl` を作成し、次のように記述します (リスト 17.41)。

#### ▼リスト 17.41: ptw\_ctrl\_if.veryl

```

1 import eei::*;
2
3 interface ptw_ctrl_if {
4     var priv: PrivMode;
5     var satp: UIntX ;
6     var mxr : logic ;
7     var sum : logic ;
8     var mprv: logic ;
9     var mpp : PrivMode;
10
11    modport master {
12        priv: output,
13        satp: output,
14        mxr : output,
15        sum : output,
16        mprv: output,
17        mpp : output,
18    }
19
20    modport slave {
21        is_enabled: import,
22        ..converse(master)
23    }
24
25    function is_enabled (
26        is_inst: input logic,
27    ) -> logic {
28        if satp[msb-4] == 0 {
29            return 0;
30        }
31        if is_inst {
32            return priv <= PrivMode::S;
33        } else {
34            return (if mprv ? mpp : priv) <= PrivMode::S;
35        }
36    }

```

37 }

`is_enabled` は、CSR とアクセス目的からページングがページングが有効かどうかを判定する関数です。Bare かどうかを判定した後に、命令フェッチかどうか (`is_inst`) によって分岐しています。命令フェッチのときは S-mode 以下の特権レベルのときにページングが有効になります。ロードストアのとき、`mstatus.MPRV` が `1` なら `mstatus.MPP`、`0` なら現在の特権レベルが S-mode 以下ならページングが有効になります。

### 17.8.2 Bare だけに対応するアドレス変換モジュールを実装する

`src/ptw.veryl` を作成し、次のようなポートを記述します (リスト 17.42)。

▼ リスト 17.42: ポートの定義 (ptw.veryl)

```

1 import eei::*;

2

3 module ptw (
4     clk      : input   clock          ,
5     rst      : input   reset          ,
6     is_inst: input   logic          ,
7     slave   : modport Membus::slave ,
8     master  : modport Membus::master,
9     ctrl    : modport ptw_ctrl_if::slave,
10 ) {

```

`slave` は core モジュール側からの仮想アドレスによる要求を受け付けるためのインターフェースです。`master` は mmio\_controller モジュール側に物理アドレスによるアクセスを行うためのインターフェースです。

`is_inst` を使い、ページングが有効かどうか判定します (リスト 17.43)。

▼ リスト 17.43: ページングが有効かどうかを判定する (ptw.veryl)

```

1 let paging_enabled: logic = ctrl.is_enabled(is_inst);

```

状態の管理のために `State` 型を定義します (リスト 17.44)。

▼ リスト 17.44: 状態の定義 (ptw.veryl)

```

1 enum State {
2     IDLE,
3     EXECUTE_READY,
4     EXECUTE_VALID,
5 }
6
7 var state: State;

```

`State::IDLE`

`slave` から要求を受け付け、`master` に物理アドレスでアクセスします。

`master` の `ready` が 1 なら `State::EXECUTE_VALID`、0 なら `EXECUTE_READY` に状態を移動します。

#### `State::EXECUTE_READY`

`master` に物理アドレスでメモリアクセスを要求し続けます。`master` の `ready` が 1 なら状態を `State::EXECUTE_VALID` に移動します。

#### `State::EXECUTE_VALID`

`master` からの結果を待ちます。`master` の `rvalid` が 1 のとき、`State::IDLE` と同じように `slave` からの要求を受け付けます。`slave` が何も要求していないなら、状態を `State::IDLE` に移動します。

`slave` からの要求を保存しておくためのインターフェースをインスタンス化します(リスト 17.45)。

#### ▼ リスト 17.45: slave を保存するためのインターフェースをインスタンス化する (ptw.veryl)

```
1 inst slave_saved: Membus;
```

状態に基づいて、`master` に要求を割り当てます(リスト 17.46、リスト 17.47)。`State::EXECUTE_READY` で `master` に要求を割り当てるとき、`physical_addr` レジスタの値をアドレスに割り当てるようになります。

#### ▼ リスト 17.46: 物理アドレスを保存するためのレジスタを作成する (ptw.veryl)

```
1 var physical_addr: Addr;
```

#### ▼ リスト 17.47: master に要求を割り当てる (ptw.veryl)

```
1 function assign_master (
2     addr : input Addr ,
3     wen  : input logic ,
4     wdata: input logic<MEMBUS_DATA_WIDTH> ,
5     wmask: input logic<MEMBUS_DATA_WIDTH / 8>,
6 ) {
7     master.valid = 1;
8     master.addr  = addr;
9     master.wen   = wen;
10    master.wdata = wdata;
11    master.wmask = wmask;
12 }
13
14 function accept_request_comb () {
15     if slave.ready && slave.valid && !paging_enabled {
16         assign_master(slave.addr, slave.wen, slave.wdata, slave.wmask);
17     }
18 }
19
20 always_comb {
21     master.valid = 0;
22     master.addr  = 0;
```

```

23     master.wen    = 0;
24     master.wdata  = 0;
25     master.wmask  = 0;
26
27     case state {
28         State::IDLE      : accept_request_comb();
29         State::EXECUTE_READY: assign_master      (physical_addr, slave_saved.wen, slave_sav>
>ed.wdata, slave_saved.wmask);
30         State::EXECUTE_VALID: if master.rvalid {
31             accept_request_comb();
32         }
33         default: {}
34     }
35 }
```

状態に基づいて、`ready` と結果を `slave` に割り当てます (リスト 17.48)。

#### ▼ リスト 17.48: slave に結果を割り当てる (ptw.veryl)

```

1 always_comb {
2     slave.ready  = 0;
3     slave.rvalid = 0;
4     slave.rdata  = 0;
5     slave.expt   = 0;
6
7     case state {
8         State::IDLE      : slave.ready = 1;
9         State::EXECUTE_VALID: {
10             slave.ready  = master.rvalid;
11             slave.rvalid = master.rvalid;
12             slave.rdata  = master.rdata;
13             slave.expt   = master.expt;
14         }
15         default: {}
16     }
17 }
```

状態を遷移する処理を記述します (リスト 17.49)。要求を受け入れるとき、`slave_saved` に要求を保存します。

#### ▼ リスト 17.49: 状態を遷移する (ptw.veryl)

```

1 function accept_request_ff () {
2     slave_saved.valid = slave.ready && slave.valid;
3     if slave.ready && slave.valid {
4         slave_saved.addr  = slave.addr;
5         slave_saved.wen   = slave.wen;
6         slave_saved.wdata = slave.wdata;
7         slave_saved.wmask = slave.wmask;
8         if paging_enabled {
9             // TODO
10        } else {
11            state      = if master.ready ? State::EXECUTE_VALID : State::EXECUTE_READY;
```

```

12         physical_addr = slave.addr;
13     }
14 } else {
15     state = State::IDLE;
16 }
17 }
18
19 function on_clock () {
20     case state {
21         State::IDLE      : accept_request_ff();
22         State::EXECUTE_READY: if master.ready {
23             state = State::EXECUTE_VALID;
24         }
25         State::EXECUTE_VALID: if master.rvalid {
26             accept_request_ff();
27         }
28         default: {}
29     }
30 }
31
32 function on_reset () {
33     state          = State::IDLE;
34     physical_addr = 0;
35     slave_saved.valid = 0;
36     slave_saved.addr = 0;
37     slave_saved.wen = 0;
38     slave_saved.wdata = 0;
39     slave_saved.wmask = 0;
40 }
41
42 always_ff {
43     if_reset {
44         on_reset();
45     } else {
46         on_clock();
47     }
48 }
```

### 17.8.3 ptw モジュールをインスタンス化する

top モジュールで、ptw モジュールをインスタンス化します。

ptw モジュールは mmio\_controller モジュールの前で仮想アドレスを物理アドレスに変換するモジュールです。ptw モジュールと mmio\_controller モジュールの間のインターフェースを作成します（リスト 17.50）。

▼ リスト 17.50: ptw モジュールと mmio\_controller モジュールの間のインターフェースを作成する  
(*top.v*)

```
1     inst ptw_membus      : Membus;
```

調停処理を ptw モジュール向けのものに変更します（リスト 17.51）。

## ▼ リスト 17.51: 調停処理を ptw モジュール向けのものに変更する (top.veryl)

```

1  always_ff {
2      if_reset {
3          memarb_last_i = 0;
4      } else {
5          if ptw_membus.ready {
6              memarb_last_i = !d_membus.valid;
7          }
8      }
9  }
10
11 always_comb {
12     i_membus.ready  = ptw_membus.ready && !d_membus.valid;
13     i_membus.rvalid = ptw_membus.rvalid && memarb_last_i;
14     i_membus.rdata  = ptw_membus.rdata;
15     i_membus.expt   = ptw_membus.expt;
16
17     d_membus.ready  = ptw_membus.ready;
18     d_membus.rvalid = ptw_membus.rvalid && !memarb_last_i;
19     d_membus.rdata  = ptw_membus.rdata;
20     d_membus.expt   = ptw_membus.expt;
21
22     ptw_membus.valid = i_membus.valid | d_membus.valid;
23     if d_membus.valid {
24         ptw_membus.addr  = d_membus.addr;
25         ptw_membus.wen    = d_membus.wen;
26         ptw_membus.wdata  = d_membus.wdata;
27         ptw_membus.wmask  = d_membus.wmask;
28     } else {
29         ptw_membus.addr  = i_membus.addr;
30         ptw_membus.wen    = 0; // 命令フェッチは常に読み込み
31         ptw_membus.wdata  = 'x;
32         ptw_membus.wmask  = 'x;
33     }
34 }
```

今処理している要求、または今のクロックから処理し始める要求が命令フェッチによるものか判定する変数を作成します（リスト 17.52）。

## ▼ リスト 17.52: ptw モジュールが処理する要求が命令フェッチによるものかを判定する (top.veryl)

```

1  let ptw_is_inst : logic = (i_membus.ready && i_membus.valid) || // inst ack or
2      !(d_membus.ready && d_membus.valid) && memarb_last_i; // data not ack & last ack is inst
```

ptw モジュールをインスタンス化します（リスト 17.53）。

## ▼ リスト 17.53: ptw モジュールをインスタンス化する (top.veryl)

```

1  inst ptw_ctrl: ptw_ctrl_if;
2  inst paging_unit: ptw (
3      clk           ,
4      rst           ,
5      is_inst: ptw_is_inst,
```

```

6     slave  : ptw_membus ,
7     master : mmio_membus,
8     ctrl   : ptw_ctrl   ,
9 );

```

csrunut モジュールと ptw モジュールを `ptw_ctrl_if` インターフェースで接続するために、core モジュールにポートを追加します（リスト 17.54、リスト 17.55）。

#### ▼ リスト 17.54: core モジュールに `ptw_ctrl_if` インターフェースを追加する (core.veryl)

```

1 module core (
2   clk      : input  clock          ,
3   rst      : input  reset          ,
4   i_membus: modport core_inst_if::master,
5   d_membus: modport core_data_if::master,
6   led      : output UIntX        ,
7   aclint  : modport aclint_if::slave  ,
8   ptw_ctrl: modport ptw_ctrl_if::master ,
9 ) {

```

#### ▼ リスト 17.55: `ptw_ctrl_if` インターフェースを割り当てる (top.veryl)

```

1 inst c: core (
2   clk           ,
3   rst           ,
4   i_membus: i_membus_core  ,
5   d_membus: d_membus_core  ,
6   led           ,
7   aclint : aclint_core_bus,
8   ptw_ctrl      ,
9 );

```

csrunut モジュールにポートを追加し、CSR を割り当てます（リスト 17.56、リスト 17.57、リスト 17.58）。

#### ▼ リスト 17.56: csunit モジュールに `ptw_ctrl_if` インターフェースを追加する (csrunut.veryl)

```

1   membusr    : modport core_data_if::master   ,
2   ptw_ctrlr  : modport ptw_ctrl_if::master   ,
3 ) {

```

#### ▼ リスト 17.57: csrunut モジュールのインスタンスに `ptw_ctrl_if` インターフェースを割り当てる (core.veryl)

```

1     membusr  : d_membus           ,
2     ptw_ctrlr : ptw_ctrl_if      ,
3 );

```

#### ▼ リスト 17.58: インターフェースに CSR の値を割り当てる (csrunut.veryl)

```

1 always_comb {
2   ptw_ctrl.priv = mode;
3   ptw_ctrl.satp = satp;
4   ptw_ctrl.mxr  = mstatus_mxr;

```

```

5     ptw_ctrl.sum  = mstatus_sum;
6     ptw_ctrl.mprv = mstatus_mprv;
7     ptw_ctrl.mpp  = mstatus_mpp;
8 }
```

## 17.9 Sv39 の実装

ptw モジュールに、Sv39 を実装します。ここで定義する関数は、コメントと「17.3 Sv39 のアドレス変換」(p.320) を参考に動作を確認してください。

### 17.9.1 定数の定義

ptw モジュールで使用する定数と関数を実装します。

`src/sv39util.veryl` を作成し、次のように記述します (リスト 17.59)。定数は「17.3 Sv39 のアドレス変換」(p.320) で使用しているものと同じです。

#### ▼ リスト 17.59: sv39util.veryl

```

1 import eei::*;
2 package sv39util {
3     const PAGESIZE: u32      = 12;
4     const PTESIZE : u32      = 8;
5     const LEVELS  : logic<2> = 3;
6
7     type Level = logic<2>;
8
9     // 有効な仮想アドレスか判定する
10    function is_valid_vaddr (
11        va: input Addr,
12    ) -> logic {
13        let hiaddr: logic<26> = va[msb:38];
14        return &hiaddr || &~hiaddr;
15    }
16
17    // 仮想アドレスのVPN[level]フィールドを取得する
18    function vpn (
19        va   : input Addr ,
20        level: input Level,
21    ) -> logic<9> {
22        return case level {
23            0      : va[20:12],
24            1      : va[29:21],
25            2      : va[38:30],
26            default: 0,
27        };
28    }
29
30    // 最初にフェッチするPTEのアドレスを取得する
```

```

31     function get_first_pte_address (
32         satp: input UIntX,
33         va : input Addr ,
34     ) -> Addr {
35         return {
36             1'b0 repeat XLEN - 44 - PAGESIZE,
37             satp[43:0],
38             vpn(va, 2),
39             1'b0 repeat $clog2(PTESIZE)
40         };
41     }
42 }
```

## 17.9.2 PTE の定義

Sv39 の PTE のビットを分かりやすく取得するために、次のインターフェースを定義します。

`src/pte.veryl` を作成し、次のように記述します（リスト 17.60）。

### ▼ リスト 17.60: pte.veryl

```

1 import eei::*;
2 import sv39util::*;
3
4 interface PTE39 {
5     var value: UIntX;
6
7     function v () -> logic { return value[0]; }
8     function r () -> logic { return value[1]; }
9     function w () -> logic { return value[2]; }
10    function x () -> logic { return value[3]; }
11    function u () -> logic { return value[4]; }
12    function a () -> logic { return value[6]; }
13    function d () -> logic { return value[7]; }
14
15    function reserved -> logic<10> { return value[63:54]; }
16
17    function ppm2 () -> logic<26> { return value[53:28]; }
18    function ppm1 () -> logic<9> { return value[27:19]; }
19    function ppm0 () -> logic<9> { return value[18:10]; }
20    function ppm () -> logic<44> { return value[53:10]; }
21 }
```

PTE の値を使った関数を定義します（リスト 17.61）。

### ▼ リスト 17.61: PTE の値を使った関数を定義する (pte.veryl)

```

1 // leaf PTEか判定する
2 function is_leaf () -> logic { return r() || x(); }
3
4 // leaf PTEのとき、PPNがページサイズに整列されているかどうかを判定する
5 function is_ppn_aligned (
6     level: input Level,
```

```

7   ) -> logic {
8     return case level {
9       0      : 1,
10      1      : ppn0() == 0,
11      2      : ppn1() == 0 && ppn0() == 0,
12      default: 1,
13    };
14  }
15
16 // 有効なPTEか判定する
17 function is_valid (
18   level: input Level,
19 ) -> logic {
20   if !v() || reserved() != 0 || !r() && w() {
21     return 0;
22   }
23   if is_leaf() && !is_ppn_aligned(level) {
24     return 0;
25   }
26   if !is_leaf() && level == 0 {
27     return 0;
28   }
29   return 1;
30 }
31
32 // 次のlevelのPTEのアドレスを得る
33 function get_next_pte_addr (
34   level: input Level,
35   va   : input Addr ,
36 ) -> Addr {
37   return {
38     1'b0 repeat XLEN - 44 - PAGESIZE,
39     ppn(),
40     vpn(va, level - 1),
41     1'b0 repeat $clog2(PTESIZE)
42   };
43 }
44
45 // PTEと仮想アドレスから物理アドレスを生成する
46 function get_physical_address (
47   level: input Level,
48   va   : input Addr ,
49 ) -> Addr {
50   return {
51     8'b0, ppn2(), case level {
52       0: {
53         ppn1(), ppn0()
54       },
55       1: {
56         ppn1(), vpn(va, 0)
57       },
58       2: {
59         vpn(va, 1), vpn(va, 0)

```

```

60         },
61         default: 18'b0,
62     }, va[11:0]
63   );
64 }
65
66 // A、Dビットを更新する必要があるかを判定する
67 function need_update_ad (
68   wen: input logic,
69 ) -> logic {
70   return !a() || wen && !d();
71 }
72
73 // A、Dビットを更新したPTEの下位8ビットを生成する
74 function get_updated_ad (
75   wen: input logic,
76 ) -> logic<8> {
77   let a: logic<8> = 1 << 6;
78   let d: logic<8> = wen as u8 << 7;
79   return value[7:0] | a | d;
80 }
```

### 17.9.3 ptw モジュールの実装

sv39util パッケージを import します (リスト 17.62)。

▼ リスト 17.62: sv39util パッケージを import する (ptw.veryl)

```
1 import sv39util::*;


```

PTE39 インターフェースをインスタンス化します (リスト 17.63)。 `value` には `master` のロード結果を割り当てます。

▼ リスト 17.63: PTE39 インターフェースをインスタンス化する (ptw.veryl)

```
1 inst pte      : PTE39;
2 assign pte.value = master.rdata;

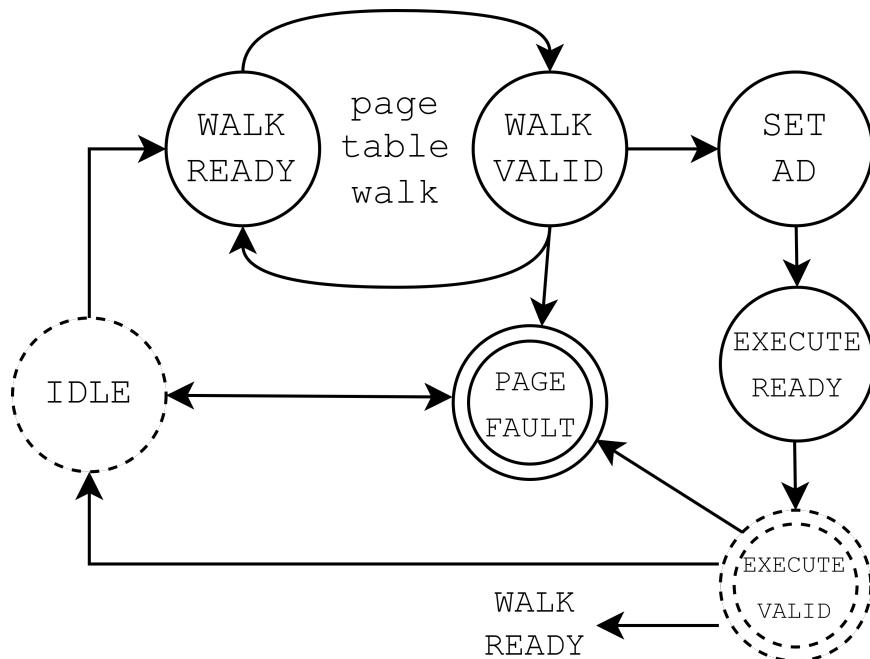

```

仮想アドレスを変換するための状態を追加します (リスト 17.64)。本章ではページングが有効な時に、`state` が図 17.12 のように遷移するようにします。

▼ リスト 17.64: 状態の定義 (ptw.veryl)

```

1 enum State {
2   IDLE,
3   WALK_READY,
4   WALK_VALID,
5   SET_AD,
6   EXECUTE_READY,
7   EXECUTE_VALID,
8   PAGE_FAULT,
9 }
```



▲図 17.12: 状態の遷移図 (点線の状態で新しく要求を受け付け、二重丸の状態で結果を返す)

現在の PTE の level( `level` )、PTE のアドレス ( `taddr` )、要求によって更新される PTE の下位 8 ビット ( `wdata_ad` ) を格納するためのレジスタを定義します ( リスト 17.65、リスト 17.66 )。

▼リスト 17.65: レジスタの定義 (ptw.veryl)

```

1 var physical_addr: Addr      ;
2 var taddr        : Addr      ;
3 var level        : Level     ;
4 var wdata_ad     : logic<8>;

```

▼リスト 17.66: レジスタをリセットする (ptw.veryl)

```

1 function on_reset () {
2     state          = State::IDLE;
3     physical_addr = 0;
4     taddr         = 0;
5     level         = 0;
6     wdata_ad      = 0;

```

PTE のフェッチと A、D ビットの更新のために `master` に要求を割り当てます ( リスト 17.67 )。

PTE は `taddr` を使ってアクセスし、A、D ビットの更新では下位 8 ビットのみの書き込みマスクを設定します。

▼リスト 17.67: master に要求を割り当てる (ptw.veryl)

```

1 case state {
2     State::IDLE      : accept_request_comb();

```

```

3 State::WALK_READY: assign_master      (taddr, 0, 0, 0);
4 State::SET_AD    : assign_master      (taddr, 1, // wen = 1
5   {1'b0 repeat MEMBUS_DATA_WIDTH - 8, wdata_ad}, // wdata
6   {1'b0 repeat XLEN / 8 - 1, 1'b1} // wmask
7 );
8 State::EXECUTE_READY: assign_master(physical_addr, slave_saved.wen, slave_saved.wdata, slave_
> saved.wmask);
9 State::EXECUTE_VALID: if master.rvalid {
10   accept_request_comb();
11 }
12 default: {}
13 }
```

`slave` への結果の割り当てで、ページフォルト例外が発生していた場合の結果を割り当てます (リスト 17.68)。

#### ▼ リスト 17.68: ページフォルト例外のときの結果を割り当てる (ptw.veryl)

```

1 State::PAGE_FAULT: {
2   slave.rvalid      = 1;
3   slave.expt.valid = 1;
4   slave.expt.page_fault = 1;
5 }
```

ページングが有効なときに要求を受け入れる動作を実装します (リスト 17.69)。仮想アドレスが有効かどうかでページフォルト例外を判定し、`taddr` レジスタに最初の PTE のアドレスを割り当てます。`level` の初期値は `LEVELS - 1` とします。

#### ▼ リスト 17.69: ページングが有効なときの要求の受け入れ (ptw.veryl)

```

1 if paging_enabled {
2   state = if is_valid_vaddr(slave.addr) ? State::WALK_READY : State::PAGE_FAULT;
3   taddr = get_first_pte_address(ctrl.satp, slave.addr);
4   level = LEVELS - 1;
5 } else {
6   state      = if master.ready ? State::EXECUTE_VALID : State::EXECUTE_READY;
7   physical_addr = slave.addr;
8 }
```

ページフォルト例外が発生したとき、状態を `State::IDLE` に戻します (リスト 17.70)。

#### ▼ リスト 17.70: ページフォルト例外が発生したときの状態遷移 (ptw.veryl)

```
1 State::PAGE_FAULT: state = State::IDLE;
```

A、D ビットを更新するとき、メモリが書き込み要求を受け入れたら、状態を `State::EXECUTE_READY` に移動します (リスト 17.71)。

#### ▼ リスト 17.71: A、D ビットを更新したときの状態遷移 (ptw.veryl)

```

1 State::SET_AD: if master.ready {
2     state = State::EXECUTE_READY;
3 }

```

ページにアクセスする権限があるかを PTE と要求から判定する関数を定義します（リスト 17.72）。条件の詳細は「17.3 Sv39 のアドレス変換」（p.320）を確認してください。

#### ▼リスト 17.72: ページにアクセスする権限があるかを判定する関数 (ptw.veryl)

```

1 function check_permission (
2     req: modport Membus::all_input,
3 ) -> logic {
4     let priv: PrivMode = if is_inst || !ctrl.mprv ? ctrl.priv : ctrl.mpp;
5
6     // U-mode access with PTE.U=0
7     let u_u0: logic = priv == PrivMode::U && !pte.u();
8     // S-mode load/store with PTE.U=1 & sum=0
9     let sd_u1: logic = !is_inst && priv == PrivMode::S && pte.u() && !ctrl.sum;
10    // S-mode execute with PTE.U=1
11    let si_u1: logic = is_inst && priv == PrivMode::S && pte.u();
12
13    // execute without PTE.X
14    let x: logic = is_inst && !pte.x();
15    // write without PTE.W
16    let w: logic = !is_inst && req.wen && !pte.w();
17    // read without PTE.R (MXR)
18    let r: logic = !is_inst && !req.wen && !pte.r() && !(pte.x() && ctrl.mxr);
19
20    return !(u_u0 | sd_u1 | si_u1 | x | w | r);
21 }

```

PTE をフェッチしてページフォルト例外を判定し、次の PTE のフェッチ、A、D ビットを更新する状態への遷移を実装します（リスト 17.73）。

#### ▼リスト 17.73: PTE のフェッチと PTE の確認 (ptw.veryl)

```

1 State::WALK_READY: if master.ready {
2     state = State::WALK_VALID;
3 }
4 State::WALK_VALID: if master.rvalid {
5     if !pte.is_valid(level) {
6         state = State::PAGE_FAULT;
7     } else {
8         if pte.is_leaf() {
9             if check_permission(slave_saved) {
10                 physical_addr = pte.get_physical_address(level, slave_saved.addr);
11                 if pte.need_update_ad(slave_saved.wen) {
12                     state = State::SET_AD;
13                     wdata_ad = pte.get_updated_ad(slave_saved.wen);
14                 } else {
15                     state = State::EXECUTE_READY;
16                 }
17             } else {

```

```

18         state = State::PAGE_FAULT;
19     }
20 } else {
21     // read next pte
22     state = State::WALK_READY;
23     taddr = pte.get_next_pte_addr(level, slave_saved.addr);
24     level = level - 1;
25 }
26 }
27 }
```

これで Sv39 を ptw モジュールに実装できました。

## 17.10 SFENCE.VMA 命令の実装

SFENCE.VMA 命令は、SFENCE.VMA 命令を実行する以前のストア命令が MMU に反映されたことを保証する命令です。S-mode 以上の特権レベルのときに実行できます。

基本編ではすべてのメモリアクセスを直列に行い、仮想アドレスを変換するために毎回 PTE をフェッチしなおすため、何もしない命令として定義します。

### 17.10.1 SFENCE.VMA 命令をデコードする

SFENCE.VMA 命令を有効な命令としてデコードします（リスト 17.74）。

▼ リスト 17.74: SFENCE.VMA 命令を有効な命令としてデコードする (inst\_decoder.veryl)

```

1 bits == 32'h10200073 || //SRET
2 bits == 32'h10500073 || // WFI
3 f7 == 7'b0001001 && bits[11:7] == 0, // SFENCE.VMA
```

### 17.10.2 特権レベルの確認、mstatus.TVM を実装する

S-mode 未満の特権レベルで SFENCE.VMA 命令を実行しようとしたとき、Illegal instruction 例外が発生します。

mstatus.TVM は S-mode のときに satp レジスタにアクセスできるか、SFENCE.VMA 命令を実行できるかを制御するビットです。mstatus.TVM が 1 にされているとき、Illegal instruction 例外が発生します。

mstatus.TVM を書き込めるようにします（リスト 17.75）。

▼ リスト 17.75: mstatus レジスタの書き込みマスクを変更する (csrunit.veryl)

```
1 const MSTATUS_WMASK : UIntX = 'h0000_0000_007e_19aa as UIntX;
```

## ▼リスト 17.76: mstatus.TVM を示す変数を作成する (csrunit.veryl)

```
1 let mstatus_tvm : logic = mstatus[20];
```

特権レベルを確認して、例外を発生させます（リスト 17.77、リスト 17.78、リスト 17.79）。

## ▼リスト 17.77: SFENCE.VMA 命令かどうかを判定する (csrunit.veryl)

```
1 let is_sfence_vma: logic = ctrl.is_csr && ctrl.funct7 == 7'b0001001 && ctrl.funct3 == 0 && r>
>d_addr == 0;
```

## ▼リスト 17.78: SFENCE.VMA 命令の例外を判定する (csrunit.veryl)

```
1 let expt_tvm: logic = (is_sfence_vma && mode <: PrivMode::S) || (mstatus_tvm && mode == Priv>
>Mode::S && (is_wsc && csr_addr == CsrAddr::SATP || is_sfence_vma));
```

## ▼リスト 17.79: 例外を発生させる (csrunit.veryl)

```
1 let raise_expt: logic = valid && (expt_info.valid || expt_write_READONLY_CSR || expt_CSR_PRI>
>V_VIOLATION || expt_ZICNTR_PRIV || expt_trap_RETURN_PRIV || expt_MEMORY_FAULT || expt_tvm);
2 let expt_cause: UIntX = switch {
3     ...
4     expt_tvm           : CsrCause::ILLEGAL_INSTRUCTION,
5     default            : 0,
6 };

```

## 17.11 パイプラインをフラッシュする

本書はパイプライン化した CPU を実装しているため、命令フェッチは前の命令を待たずに次々に行われます。

### 17.11.1 CSR の変更

mstatus レジスタの MXR、SUM、TVM ビット、satp レジスタを書き換えたとき、CSR を書き換える命令の後ろの命令は、CSR の変更が反映されていない状態でアドレス変換してフェッチした命令になっている可能性があります。

CSR の書き換えをページングに反映するために、特定の CSR を書き換えたらパイプラインをフラッシュするようにします。

csrunit モジュールに、フラッシュするためのフラグを追加します（リスト 17.80、リスト 17.81、リスト 17.82）。

## ▼リスト 17.80: csrunit モジュールのポートにフラッシュするためのフラグを追加する (csrunit.veryl)

```
1 flush      : output logic          ,
2 minstret   : input   UInt64        ,
```

## ▼リスト 17.81: csru\_flush 変数の定義 (core.veryl)

```

1 var csru_trap_return: logic      ;
2 var csru_flush      : logic      ;
3 var minstret       : UInt64     ;

```

## ▼リスト 17.82: csrunit モジュールの flush フラグを csru\_flush に割り当てる (core.veryl)

```

1 flush      : csru_flush      ,
2 minstret   :                  ,

```

satp、mstatus、sstatus レジスタが変更されるときに `flush` を 1 にします (リスト 17.83)。

## ▼リスト 17.83: satp、mstatus、sstatus レジスタが変更されるときに flush を 1 にする (csrunit.veryl)

```

1 let wsc_flush: logic = is_wsc && (csr_addr == CsrAddr::SATP || csr_addr == CsrAddr::MSTATUS >
>|| csr_addr == CsrAddr::SSTATUS);
2 assign flush      = valid && wsc_flush;

```

`flush` が 1 のとき、制御ハザードが発生したことにしてパイプラインをフラッシュします (リスト 17.84)。

## ▼リスト 17.84: csru\_flush が 1 のときにパイプラインをフラッシュする (core.veryl)

```

1 assign control_hazard      = mems_valid && (csru_raise_trap || mems_ctrl.is_jump || memq>
>_rdata.br_taken || csru_flush);
2 assign control_hazard_pc_next = if csru_raise_trap ? csru_trap_vector : // trap
3     if csru_flush ? mems_pc + 4 : memq_rdata.jump_addr; // flush or jump

```

## 17.11.2 FENCE.I 命令の実装

あるアドレスにデータを書き込むとき、データを書き込んだ後の命令が、書き換えられたアドレスにある命令だった場合、命令のビット列がデータが書き換えられる前のものになっている可能性があります。

FENCE.I 命令は、FENCE.I 命令の後の命令のフェッチ処理がストア命令の完了後に行われる事を保証する命令です。例えばユーザーのアプリケーションのプログラムをページに書き込んで実行するとき、ページへの書き込みを反映させるために使用します。

FENCE.I 命令を判定し、パイプラインをフラッシュする条件に設定します (リスト 17.85、リスト 17.86)。

## ▼リスト 17.85: FENCE.I 命令かどうかを判定する (csrunit.veryl)

```

1 let is_fence_i: logic = inst_bits[6:0] == OP_MISC_MEM && ctrl.funct3 == 3'b001;

```

## ▼リスト 17.86: FENCE.I 命令のときに flush を 1 にする (csrunit.veryl)

```

1 assign flush      = valid && (wsc_flush || is_fence_i);

```

riscv-tests の `-v-` を含むテストを実行し、実装している命令のテストに成功することを確認してください。

---

# **第 18 章**

## **PLIC の実装**

---

本章では外部割り込みと複数の入出力デバイスの割り込みを調停するための仕組みを実装します。

本章は Web 版で提供します。

サポートページを確認してください。

---

# 第 19 章

## Linux を動かす

---

本章では Linux を CPU で動かします。

本章は Web 版で提供します。

サポートページを確認してください。

# あとがき (第Ⅰ部)

いかがだったでしょうか。本書が自作CPUの助けになれば幸いです。

## 著者について



**阿部奏太** (kanataso) (kanapipopipo@X/Twitter, nananapo@GitHub)

いつの間にか自作CPUの沼に沈んでいました。

カラオケまねきねこ (のまねっこー) とコメダ珈琲 (のエッグサンド) が好き。

計算機と法律に興味があります。

## 謝辞

本書は次の方々にレビューしていただきました。

- 石谷太一 (@taichi-ishitani<sup>\*1</sup>)
- 井田健太 (@ciniml<sup>\*2</sup>)
- 内田公太 (@uchan\_nos<sup>\*3</sup>)
- 初田直也 (@dalance<sup>\*4</sup>)

筆者がCPUを作り始めたのは、井田さんの「RISC-VとChiselで学ぶはじめてのCPU自作<sup>\*5</sup>」を読んだのがきっかけでした。この本が無ければ、筆者はCPUを作ろうとは思わなかったかと思います。

CPU自作を始めて半年後から約一年間、サイボウズ・ラボ株式会社のサイボウズ・ラボユースの支援を受けることで、自作CPUに集中して取り組むことができました(本書の一部はラボユースの期間に執筆されました)。メンターの内田さんにはとても感謝しています。

Verylの作者の初田さんには、筆者がVerylでCPUを作るにあたって見つけた不具合を迅速に修正していただきました。初田さんと石谷さんにはレビューでとても多くの指摘をいただき、本書の品質を向上できました。

執筆にあたって関わったすべての方に、この場をお借りしてお礼申し上げます。

<sup>\*1</sup> <https://github.com/taichi-ishitani>

<sup>\*2</sup> <https://github.com/ciniml>

<sup>\*3</sup> [https://x.com/uchan\\_nos](https://x.com/uchan_nos)

<sup>\*4</sup> <https://github.com/dalance>

<sup>\*5</sup> <https://gihyo.jp/book/2021/978-4-297-12305-5>

# あとがき（第II部、第III部）

いかがだったでしょうか。

本書（基本編）はこれで終わりになります。だいぶ駆け足になってしましましたが、RISC-V の CPU の具体的な書き方が分かったかと思います。

基本編では RISC-V の CPU をゼロから書き始め、Linux を起動できるくらいの基本的な機能を実装する方法を解説しました。しかし、Linux を起動できるといつても速度や機能は現代的な CPU に遠く及びません。次巻の「Veryl で作る CPU 応用編」ではキャッシュ、アウトオブオーダー実行などを実装し、CPU の高速化と他の機能について解説する予定です。

教科書を読んでなんとなく CPU を理解した気がするけど作り方がわからない、既存の CPU 実装を参考に自分で CPU を書いてみたいけど何から作れば良いかわからない、という方に本書が役立つことを願っています。

2025年5月28日

## 著者について



阿部奏太 (kanataso) (kanapipopipo@X/Twitter, nananapo@GitHub)

カラオケまねきねこダイヤモンド会員 (3期目)

最近はキョーちゃん (有斐閣のキャラクター) が気になっている。

今後数年間は CPU から逃げられなくなりました。

## 謝辞

本書を執筆するにあたって、石谷太一氏 (@taichi-ishitani<sup>\*6</sup>)、初田直也氏 (@dalance<sup>\*7</sup>) にレビューして頂きました。お二方には執筆中に見つけたバグをすぐに直して頂き、スムーズに執筆を進めることができました。また、栗本龍一氏 (@ryone9re<sup>\*8</sup>) には図や表を作るのを手伝っていただき、どうにか入稿に間に合わせることができました。

本書は大変に短い期間で執筆されたため、各所にご迷惑をおかけしました。執筆にあたって関わったすべての方に、この場をお借りしてお礼申し上げます。

<sup>\*6</sup> <https://github.com/taichi-ishitani>

<sup>\*7</sup> <https://github.com/dalance>

<sup>\*8</sup> <https://github.com/ryone9re>

# このプロジェクトに貢献する

本書は常に提案やリクエストを受け付けています。質問などがある場合も、お気軽に issue を作成してください。

※今のところ誤字脱字などの軽微な修正の PR は受け入れていますが、コード<sup>\*9</sup>の修正についてはおそらく過酷な作業になりそうなので、どうしてもやりたいという方は issue を立てるか twitter<sup>\*10</sup>でメンションしてください。

[nananapo/veryl-riscv-book](https://github.com/nananapo/veryl-riscv-book)<sup>\*11</sup>

---

<sup>\*9</sup> <https://github.com/nananapo/bluecore>

<sup>\*10</sup> <https://x.com/kanapipopipo>

<sup>\*11</sup> <https://github.com/nananapo/veryl-riscv-book/tree/main>

# 参考文献

- [1] 天野 英晴, FPGA の原理と構成, オーム社
- [2] 坂井 修一, 論理回路入門, 培風館
- [3] 演算子の優先順位 - The Veryl Hardware Description Language,[https://doc.veryl-lang.org/book/ja/05\\_language\\_reference/04\\_expression/01\\_operator\\_precedence.html](https://doc.veryl-lang.org/book/ja/05_language_reference/04_expression/01_operator_precedence.html)
- [4] 演算子 - The Veryl Hardware Description Language,[https://doc.veryl-lang.org/book/ja/05\\_language\\_reference/02\\_lexical\\_structure/01\\_operator.html](https://doc.veryl-lang.org/book/ja/05_language_reference/02_lexical_structure/01_operator.html)
- [5] The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411  
2.3. Immediate Encoding Variants
- [6] The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411  
37. RV32/64G Instruction Set Listings
- [7] The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411  
2.4. Integer Computational Instructions
- [8] The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture version 20240411  
2.5. Control Transfer Instructions
- [9] The RISC-V Instruction Set Manual Volume II: Privileged Architecture version 20240411  
Figure 10. Encoding of mtvec MODE field.
- [10] The RISC-V Instruction Set Manual Volume II: Privileged Architecture version 20240411  
3.1.7. Machine Trap-Vector Base-Address Register
- [11] The RISC-V Instruction Set Manual Volume II: Privileged Architecture version 20240411  
15. RISC-V Privileged Instruction Set Listings
- [12] David Patterson, John Hennessy(著), 成田 光彰 訳, コンピュータの構成と設計 MIPS Edition 第6版 [上] ~ハードウェアとソフトウェアのインタフェース~, 日経BP

[13]

PYNQ-Z1 Reference Manual - Digilent Reference, <https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual>

# Veryl で作る CPU

## 基本編

---

2024 年 11 月 3 日 基本編 第 I 部 ver 1.0 (技術書典 17)

<https://cpu.kanataso.net/>

2025 年 6 月 1 日 基本編 第 II 部、第 III 部 ver 1.0 (技術書典 18)

著 者 阿部奏太

発行者 阿部奏太

連絡先 [kanastudio@oekaki.chat](mailto:kanastudio@oekaki.chat)

印刷所 株式会社栄光

---

© 2024- ミ一ミミ研究室