

Veryl で作る CPU

— 基本編 —

[著] 阿部奏太

<https://cpu.kanataso.net/>

■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。

目次

第 I 部	RV64IMAC の実装	1
第 1 章	M 拡張の実装	2
1.1	概要	2
1.2	命令のデコード	4
1.3	muldivunit モジュールの実装	5
1.3.1	muldivunit モジュールを作成する	5
1.3.2	EX ステージを変更する	6
1.4	符号無し乗算器の実装	8
1.4.1	mulunit モジュールを実装する	8
1.4.2	mulunit モジュールをインスタンス化する	10
1.5	MULHU 命令の実装	11
1.6	MUL、MULH 命令の実装	11
1.6.1	符号付き乗算を符号なし乗算器で実現する	11
1.6.2	符号付き乗算を実装する	12
1.6.3	MULHSU 命令の実装	13
1.6.4	MULW 命令の実装	14
1.7	符号無し割り算の実装	16
1.7.1	divunit モジュールを実装する	16
1.7.2	divunit モジュールをインスタンス化する	19
1.8	DIVU、REMU 命令の実装	19
1.9	DIV、REM 命令の実装	20
1.9.1	符号付き除算を符号無し除算器で実現する	20
1.9.2	符号付き除算を実装する	20
1.10	DIVW、DIVUW、REMW、REMUW 命令の実装	22
第 2 章	例外の実装	24
2.1	例外とは何か?	24
2.2	例外情報の伝達	25
2.2.1	Environment call from M-mode 例外を IF ステージで処理する	25
2.2.2	mtval レジスタを実装する	27
2.3	Breakpoint 例外の実装	29
2.4	Illegal instruction 例外の実装	30
2.4.1	不正な命令ビット列で例外を起こす	30
2.4.2	読み込み専用の CSR への書き込みで例外を起こす	32

2.5	命令アドレスのミスアライン例外	35
2.6	ロードストア命令のミスアライン例外	36
第 3 章	Memory-mapped I/O の実装	38
3.1	Memory-mapped I/O とは何か?	38
3.2	定数の定義	38
3.3	コントローラ	39
3.3.1	コントローラを実装する	39
3.3.2	コントローラを接続する	39
3.4	ROM の実装	39
3.5	RAM のベースアドレスの変更	39
3.6	RAM の実装	39
3.7	デバッグ用 IO の実装	40
第 4 章	A 拡張の実装	41
4.1	アトミック操作	41
4.1.1	アトミック操作とは何か?	41
4.1.2	Zaamo 拡張	41
4.1.3	Zalrsc 拡張	42
4.2	命令の順序	42
4.3	amounit モジュールの作成	43
4.4	命令のデコード	43
4.4.1	is_amo フラグを実装する	43
4.4.2	アドレスを変更する	44
4.4.3	メモリインターフェースを変更する	44
4.5	Zalrsc 拡張の実装	44
4.5.1	LR.W、LR.D 命令を実装する	45
4.5.2	SC.W、SC.D 命令を実装する	45
4.6	Zaamo 拡張の実装	46
4.6.1	Zaamo 拡張の命令を実装する	46
第 5 章	C 拡張の実装	47
5.1	概要	47
5.2	IALIGN の変更	47
5.3	実装方針	48
5.4	命令フェッチモジュールの実装	48
5.4.1	既存の動作を実現する	48
5.4.2	16 ビット境界に配置された 32 ビット幅の命令をサポートする	48
5.5	RVC 命令のデコード	49
5.5.1	圧縮命令を 32 ビット幅の命令に変換する	49

5.5.2	圧縮命令フラグを実装する	49
第 II 部	特権/割り込みの実装	50
第 6 章	M-mode の実装 (1. CSR の実装)	51
6.1	概要	51
6.1.1	特権レベルとは何か?	51
6.1.2	特権レベルの実装順序	52
6.2	CSR のアドレスの追加	52
6.3	misa レジスタ	52
6.4	mimpid レジスタ	52
6.5	mhartid レジスタ	52
6.6	mstatus レジスタ	52
6.7	mcycle レジスタ	52
6.8	minstret レジスタ	53
6.9	mscratch レジスタ	53
第 7 章	M-mode の実装 (2. 割り込み、Zicntr 拡張の実装)	54
7.1	割り込みとは何か?	54
7.2	ACLINT	54
7.3	ソフトウェア割り込みの実装	54
7.3.1	msip レジスタを実装する	54
7.3.2	mip、mie レジスタを実装する	54
7.3.3	mstatus の MIE、MPIE ビットを実装する	54
7.3.4	割り込み処理の実装	54
7.4	タイマ割り込みの実装	55
7.4.1	タイマ割り込みとは何か?	55
7.4.2	mtime、mtimecmp レジスタを実装する	55
7.4.3	割り込み要因を設定する	55
7.5	WFI 命令の実装	55
7.6	Zicntr 拡張の実装	55
第 8 章	U-mode の実装	56
8.1	misa レジスタの変更	56
8.2	mstatus の UXL、TW ビットの実装	56
8.3	権限レベルの実装	56
8.4	CSR の読み書き権限の確認	56
8.5	mcounteren レジスタの実装	56
8.6	MRET 命令の実行制限	56

8.7	トラップ処理の変更	57
8.7.1	mstatus の MPP ビットを実装する	57
8.7.2	ECALL の要因を変更する	57
8.7.3	割り込み条件を変更する	57
第 9 章	S-mode の実装	58
9.1	CSR のアドレスの追加	58
9.2	misa レジスタの変更	58
9.3	mstatus レジスタの変更	58
9.3.1	SXL	58
9.3.2	MPP	58
9.4	scounteren レジスタの実装	58
9.5	sstatus レジスタの実装	58
9.6	stvec レジスタの実装	59
9.7	sscratch レジスタの実装	59
9.8	sepc レジスタの実装	59
9.9	scause レジスタの実装	59
9.10	stval レジスタの実装	59
9.11	トラップ処理の変更	59
9.11.1	sstatus の SIE、SPIE、SPP レジスタの実装	59
9.11.2	SRET 命令の実装	59
9.11.3	mip レジスタの変更	59
9.12	トラップの委譲の実装	60
9.12.1	mideleg、mideleg レジスタを作成する	60
9.12.2	sie、sip レジスタを実装する	60
9.12.3	トラップの委譲を実装する	60
第 10 章	仮想記憶システムの実装	61
10.1	仮想記憶とは何か	61
10.2	RISC-V のページング	61
10.3	メモリアクセス例外の実装	61
10.3.1	例外情報を作成する	61
10.3.2	例外の発生アドレスを特定する	61
10.4	アドレス変換モジュールの作成	61
10.5	satp レジスタの作成	61
10.6	mstatus の MXR、SUM ビットの作成	62
10.7	Sv39 の実装	62
10.8	mstatus の MPRV ビットの実装	62
10.9	SFENCE.VMA 命令の実装	62

10.10	mstatus の TVM ビットの実装	62
10.11	satp、mstatus レジスタの変更の対応	62
第 11 章	PLIC の実装	63
11.1	概要	63
11.2	デバッグ入力の実装	63
11.3	PLIC モジュールの作成	63
11.4	外部割込みの実装	63
第 12 章	Linux を動かす	64
	あとがき	65

第 I 部

RV64IMAC の実装

第 1 章

M 拡張の実装

1.1 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** と表現します。

M 拡張には、XLEN が 32 のときは表 1.1 の命令が定義されています。XLEN が 64 のときは表 1.2 の命令が定義されています。

▼ 表 1.1: M 拡張の命令 (XLEN=32)

命令	動作
MUL	rs1(符号付き) × 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) × 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) × 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) × 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) % rs2(符号付き) を求める
REMU	rs1(符号無し) % rs2(符号無し) を求める

Veril では積、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 1.2: M 拡張の命令 (XLEN=64)

命令	動作
MUL	rs1(符号付き) × rs2(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](符号付き) × rs2[31:0](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(符号付き) × rs2(符号付き) の結果 (128 ビット) の上位 64 ビットを求める
MULHU	rs1(符号無し) × rs2(符号無し) の結果 (128 ビット) の上位 64 ビットを求める
MULHSU	rs1(符号付き) × rs2(符号無し) の結果 (128 ビット) の上位 64 ビットを求める
DIV	rs1(符号付き) / rs2(符号付き) を求める
DIVW	rs1[31:0](符号付き) / rs2[31:0](符号付き) を求める 結果は符号拡張する
DIVU	rs1(符号無し) / rs2(符号無し) を求める
DIVWU	rs1[31:0](符号無し) / rs2[31:0](符号無し) を求める 結果は符号拡張する
REM	rs1(符号付き) % rs2(符号付き) を求める
REMW	rs1[31:0](符号付き) % rs2[31:0](符号付き) を求める 結果は符号拡張する
REMU	rs1(符号無し) % rs2(符号無し) を求める
REMUW	rs1[31:0](符号無し) % rs2[31:0](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 1.1)。

▼ リスト 1.1: 演算子による実装例

```
assign mul = op1 * op2;
assign div = op1 / op2;
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*¹。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*²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*³求める実装は検討せず、できるだけ単純な方法で実装し

*¹ 手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

*² そもそも除算器が搭載されていない場合があります

*³ 「効率」は、計算に要する時間や回路面積などの効率のことです。高速に計算する方法については応用編で検討します。

ます。

1.2 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 1.3)。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 1.3: 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、DIVUW	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 1.2)。

▼ リスト 1.2: `is_muldiv` フラグを追加する (`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_muldiv  : logic       , // M拡張の命令である
    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` 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 1.3)。その他

の opcode の `is_muldiv` は `F` に設定してください。

▼ リスト 1.3: `is_muldiv` を設定する (`inst_decoder.veryl`) (一部)

```
OP_OP: {
    InstType::R, T, F, T, f7 == 7'b0000001, F, F, F, F
},
OP_OP_IMM: {
    InstType::I, T, F, T, F, F, F, F, F
},
OP_OP_32: {
    InstType::R, T, F, T, f7 == 7'b0000001, T, F, F, F
},
```

1.3 muldivunit モジュールの実装

1.3.1 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 1.4)。

▼ リスト 1.4: `muldivunit` モジュール (`muldivunit.veryl`)

```
import eei::*;

module muldivunit (
    clk    : input  clock    ,
    rst    : input  reset    ,
    ready  : output logic    ,
    valid  : input  logic    ,
    funct3 : input  logic<3> ,
    op1    : input  UIntX    ,
    op2    : input  UIntX    ,
    rvalid : output logic    ,
    result : output UIntX    ,
) {

    enum State {
        Idle,
        WaitValid,
        Finish,
    }

    var state: State;

    // saved_data
    var funct3_saved: logic<3>;
```

```

always_comb {
    ready = state == State::Idle;
    rvalid = state == State::Finish;
}

always_ff {
    if_reset {
        state      = State::Idle;
        result     = 0;
        funct3_saved = 0;
    } else {
        case state {
            State::Idle: if ready && valid {
                state      = State::WaitValid;
                funct3_saved = funct3;
            }
            State::WaitValid: state = State::Finish;
            State::Finish   : state = State::Idle;
            default         : {}
        }
    }
}
}
}

```

multivunit 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いないため `result` は常に 0 を返します。次の計算を開始するまで `result` の値は維持しておきます。

1.3.2 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 の結果ではなく multivunit モジュールの結果を利用するように変更します。

まず、multivunit モジュールをインスタンス化します (リスト 1.5)。

▼ リスト 1.5: multivunit モジュールをインスタンス化する (core.veryl)

```

let exs_multiv_valid : logic = exs_valid && exs_ctrl.is_multiv && !exs_data_hazard && !exs_m>
>multiv_is_requested;
var exs_multiv_ready : logic;
var exs_multiv_rvalid: logic;
var exs_multiv_result: UIntX;

inst mdu: multivunit (
    clk           ,
    rst           ,
    valid : exs_multiv_valid ,
    ready  : exs_multiv_ready ,
    funct3: exs_ctrl.funct3 ,

```

```

    op1   : exs_op1      ,
    op2   : exs_op2      ,
    rvalid: exs_muldiv_rvalid,
    result: exs_muldiv_result,
);

```

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`)、既に計算をリクエストしていない (`!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 への計算リクエストの状態によって値が変わるようにします (リスト 1.6)。

▼ リスト 1.6: `exs_muldiv_is_requested` 変数 (core.veryl)

```

var exs_muldiv_is_requested: logic;

always_ff {
    if_reset {
        exs_muldiv_is_requested = 0;
    } else {
        // 次のステージに遷移
        if exq_rvalid && exq_rready {
            exs_muldiv_is_requested = 0;
        } else {
            // muldivunitにリクエストしたか判定する
            if exs_muldiv_valid && exs_muldiv_ready {
                exs_muldiv_is_requested = 1;
            }
        }
    }
}

```

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 1.7、リスト 1.8)。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 1.8)。

▼ リスト 1.7: EX ステージのストール条件の変更 (core.veryl)

```

var exs_muldiv_rvalided: logic;
let exs_muldiv_stall    : logic = exs_ctrl.is_muldiv && !exs_muldiv_rvalid && !exs_muldiv_rva>
>lided;

always_ff {
    if_reset {
        exs_muldiv_rvalided = 0;
    } else {
        // 次のステージに遷移
        if exq_rvalid && exq_rready {
            exs_muldiv_rvalided = 0;
        }
    }
}

```

```

    } else {
        // muldivunitの処理が完了していたら1にする
        exs_muldiv_rvalid |= exs_muldiv_rvalid;
    }
}
}

```

▼ リスト 1.8: EX ステージのストール条件の変更 (core.veryl)

```

let exs_stall: logic = exs_data_hazard || exs_muldiv_stall;

always_comb {
    // EX -> MEM
    exq_rready      = memq_wready && !exs_stall;
    memq_wvalid     = exq_rvalid && !exs_stall;
    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 = if exs_ctrl.is_muldiv ? exs_muldiv_result : exs_alu_result;
    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 : exs_alu_result & ~>
>1;
}

```

muldivunit モジュールは計算が完了したクロックの間だけしか `rvalid` を 1 に設定しないため、既に計算が完了していることを示す `exs_muldiv_rvalid` 変数を作成しています。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_rvalid`) 場合になります。

1.4 符号無し乗算器の実装

1.4.1 mulunit モジュールを実装する

`WIDTH` ビットの符号無し値同士の積を計算する乗算器を実装します。

`src/muldivunit.veryl` の中に `mulunit` モジュールを作成します (リスト 1.9)。

▼ リスト 1.9: 符号なし乗算器の実装 (muldivunit.veryl)

```

module mulunit #(
    param WIDTH: u32 = 0,
) (
    clk : input  clock
    rst : input  reset

```

```

    valid : input  logic          ,
    op1   : input  logic<WIDTH>   ,
    op2   : input  logic<WIDTH>   ,
    rvalid: output logic          ,
    result: output logic<WIDTH * 2>,
) {
    enum State {
        Idle,
        AddLoop,
        Finish,
    }

    var state: State;

    var op1zext: logic<WIDTH * 2>;
    var op2zext: logic<WIDTH * 2>;

    always_comb {
        rvalid = state == State::Finish;
    }

    var add_count: u32;

    always_ff {
        if_reset {
            state      = State::Idle;
            result      = 0;
            add_count   = 0;
            op1zext     = 0;
            op2zext     = 0;
        } else {
            case state {
                State::Idle: if valid {
                    state      = State::AddLoop;
                    result      = 0;
                    add_count   = 0;
                    op1zext     = {1'b0 repeat WIDTH, op1};
                    op2zext     = {1'b0 repeat WIDTH, op2};
                }
                State::AddLoop: if add_count == WIDTH {
                    state = State::Finish;
                } else {
                    if op2zext[add_count] {
                        result += op1zext;
                    }
                    op1zext    <= 1;
                    add_count += 1;
                }
                State::Finish: state = State::Idle;
                default           : {}
            }
        }
    }
}

```



```
}

```

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 $WIDTH$ クロックかけて行うことによって求めています (図 1.1)。計算を開始すると入力を 0 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 回目の操作のとき、

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

$$\begin{array}{rcl}
 & 1010 & op1(4bit) \\
 \times & 0101 & op2(4bit) \\
 \hline
 & 1010 & = op2 \\
 & 0000 & = (op2 \ll 1) * 0 \\
 & 1010 & = op2 \ll 2 \\
 + & 0000 & = (op2 \ll 3) * 0 \\
 \hline
 00110010 & & result(8bit)
 \end{array}$$

▲ 図 1.1: 符号無し 4 ビットの乗算

1.4.2 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 1.10)。まだ結果は利用しません。

▼ リスト 1.10: mulunit モジュールをインスタンス化する (muldivunit.vhdl)

```

// multiply unit
const MUL_OP_WIDTH : u32 = XLEN;
const MUL_RES_WIDTH: u32 = MUL_OP_WIDTH * 2;

let is_mul    : logic                = if state == State::Idle ? !func3[2] : !func3_saved>
>[2];
var mu_rvalid: logic                ;

```

```

var mu_result: logic<MUL_RES_WIDTH>;

inst mu: mulunit #(
    WIDTH: MUL_OP_WIDTH,
) (
    clk           ,
    rst           ,
    valid : ready && valid && is_mul,
    op1  : op1     ,
    op2  : op2     ,
    rvalid: mu_rvalid ,
    result: mu_result ,
);

```

1.5 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変更します (リスト 1.11)。

▼ リスト 1.11: MULHU モジュールの結果を取得する (muldivunit.veryl)

```

State::WaitValid: if is_mul && mu_rvalid {
    state = State::Finish;
    result = case funct3_saved[1:0] {
        2'b11 : mu_result[XLEN+:XLEN], // MULHU
        default: 0,
    };
}

```

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。

1.6 MUL、MULH 命令の実装

1.6.1 符号付き乗算を符号な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。

1.6.2 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 1.12)。`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 $-2^{(WIDTH - 1)}$ の絶対値も求められることを確認してください。

▼ リスト 1.12: `abs` 関数を実装する (muldivunit.veryl)

```
function abs::<WIDTH: u32> (
  value: input logic<WIDTH>,
) -> logic<WIDTH> {
  return if value[msb] ? ~value + 1 : value;
}
```

`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 1.13、リスト 1.14)。

▼ リスト 1.13: `op1` と `op2` を生成する (muldivunit.veryl)

```
let mu_op1: logic<MUL_OP_WIDTH> = case funct3[1:0] {
  2'b00, 2'b01: abs::<XLEN>(op1), // MUL, MULH
  2'b11      : op1, // MULHU
  default    : 0,
};
let mu_op2: logic<MUL_OP_WIDTH> = case funct3[1:0] {
  2'b00, 2'b01: abs::<XLEN>(op2), // MUL, MULH
  2'b11      : op2, // MULHU
  default    : 0,
};
```

▼ リスト 1.14: `mulunit` に渡す値を変更する (muldivunit.veryl)

```
inst mu: mulunit #(
  WIDTH: MUL_OP_WIDTH,
) (
  clk           ,
  rst           ,
  valid : ready && valid && is_mul,
  op1      : mu_op1           ,
  op2      : mu_op2           ,
  rvalid: mu_rvalid           ,
  result: mu_result           ,
);
```

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 1.15、リスト 1.16、リスト 1.17)。

▼ リスト 1.15: 符号を保存する変数を作成する (muldivunit.veryl)

```
// saved_data
var funct3_saved : logic<3>;
var op1sign_saved: logic  ;
var op2sign_saved: logic  ;
```

▼ リスト 1.16: 変数のリセット (muldivunit.veryl)

```
always_ff {
  if_reset {
    state      = State::Idle;
    result     = 0;
    funct3_saved = 0;
    op1sign_saved = 0;
    op2sign_saved = 0;
  } else {
```

▼ リスト 1.17: 符号を変数に保存する (muldivunit.veryl)

```
case state {
  State::Idle: if ready && valid {
    state      = State::WaitValid;
    funct3_saved = funct3;
    op1sign_saved = op1[msb];
    op2sign_saved = op2[msb];
  }
}
```

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 1.18)。

▼ リスト 1.18: 計算結果の符号を復元する (muldivunit.veryl)

```
State::WaitValid: if is_mul && mu_rvalid {
  let res_signed: logic<MUL_RES_WIDTH> = if op1sign_saved != op2sign_saved ? ~mu_result +>
  > 1 : mu_result;
  state      = State::Finish;
  result     = case funct3_saved[1:0] {
    2'b00 : res_signed[XLEN - 1:0], // MUL
    2'b01 : res_signed[XLEN+:XLEN], // MULH
    2'b11 : mu_result[XLEN+:XLEN], // MULHU
    default: 0,
  };
}
```

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。

1.6.3 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 命令の結果は n ビットの符号無し乗算器の結果の範囲に収まります。そのため、MUL、MULH 命令と同様に符号無しの乗算器で計算を実現できます。

op1 を絶対値に変換し、**op2** はそのままに設定します (リスト 1.19)。

▼ リスト 1.19: MULHSU 命令用に op1、op2 を設定する (muldivunit.veryl)

```
let mu_op1: logic<MUL_OP_WIDTH> = case funct3[1:0] {
  2'b00, 2'b01, 2'b10: abs::<XLEN>(op1), // MUL, MULH, MULHSU
  2'b11             : op1, // MULHU
  default           : 0,
};
let mu_op2: logic<MUL_OP_WIDTH> = case funct3[1:0] {
  2'b00, 2'b01: abs::<XLEN>(op2), // MUL, MULH
  2'b11, 2'b10: op2, // MULHU, MULHSU
  default      : 0,
};
```

計算結果は **op1** の符号にします (リスト 1.20)。

▼ リスト 1.20: 計算結果の符号を復元する (muldivunit.veryl)

```
State::WaitValid: if is_mul && mu_rvalid {
  let res_signed: logic<MUL_RES_WIDTH> = if op1sign_saved != op2sign_saved ? ~mu_result +>
  1 : mu_result;
  let res_mulhsu: logic<MUL_RES_WIDTH> = if op1sign_saved == 1 ? ~mu_result + 1 : mu_resu>
  <lt;
  state      = State::Finish;
  result     = case funct3_saved[1:0] {
    2'b00 : res_signed[XLEN - 1:0], // MUL
    2'b01 : res_signed[XLEN+:XLEN], // MULH
    2'b10 : res_mulhsu[XLEN+:XLEN], // MULHSU
    2'b11 : mu_result[XLEN+:XLEN], // MULHU
    default: 0,
  };
}
```

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。

1.6.4 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知るために、muldivunit モジュールに **is_op32** ポートを作成します (リスト 1.21、リスト 1.22)。

▼ リスト 1.21: is_op32 ポートを追加する (muldivunit.veryl)

```
module muldivunit (
  clk      : input  clock    ,
  rst      : input  reset    ,
  ready    : output logic    ,
  valid    : input  logic    ,
  funct3   : input  logic<3> ,
  is_op32  : input  logic    ,
```

```

    op1    : input UIntX ,
    op2    : input UIntX ,
    rvalid  : output logic ,
    result  : output UIntX ,
  ) {

```

▼ リスト 1.22: is_op32 ポートに値を割り当てる (core.veryl)

```

inst mdu: muldivunit (
  clk          ,
  rst          ,
  valid        : exs_muldiv_valid ,
  ready        : exs_muldiv_ready ,
  funct3       : exs_ctrl.funct3 ,
  is_op32      : exs_ctrl.is_op32 ,
  op1          : exs_op1          ,
  op2          : exs_op2          ,
  rvalid       : exs_muldiv_rvalid,
  result       : exs_muldiv_result,
);

```

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 1.23、リスト 1.24、リスト 1.25)。

▼ リスト 1.23: is_op32 を保存する変数を作成する (muldivunit.veryl)

```

// saved_data
var funct3_saved : logic<3>;
var is_op32_saved: logic ;
var op1sign_saved: logic ;
var op2sign_saved: logic ;

```

▼ リスト 1.24: 変数のリセット (muldivunit.veryl)

```

always_ff {
  if_reset {
    state      = State::Idle;
    result     = 0;
    funct3_saved = 0;
    is_op32_saved = 0;
    op1sign_saved = 0;
    op2sign_saved = 0;
  } else {

```

▼ リスト 1.25: is_op32 を変数に保存する (muldivunit.veryl)

```

State::Idle: if ready && valid {
  state      = State::WaitValid;
  funct3_saved = funct3;
  is_op32_saved = is_op32;
  op1sign_saved = op1[msb];
  op2sign_saved = op2[msb];
}

```

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 `sxt` 関数を作成し、`mu_op1`、`mu_op2` の割り当てに利用します (リスト 1.26、リスト 1.27)。

▼ リスト 1.26: 符号拡張する関数を作成する (muldivunit.veryl)

```
function sxt::<WIDTH_IN: u32, WIDTH_OUT: u32> (
  value: input logic<WIDTH_IN>,
) -> logic<WIDTH_OUT> {
  return {value[msb] repeat WIDTH_OUT - WIDTH_IN, value};
}
```

▼ リスト 1.27: MULW 命令用に `op1`、`op2` を設定する (muldivunit.veryl)

```
let mu_op1: logic<MUL_OP_WIDTH> = case funct3[1:0] {
  2'b00, 2'b01, 2'b10: abs::<XLEN>(if is_op32 ? sxt::<32, XLEN>(op1[31:0]) : op1), // MU
>L, MULH, MULHSU, MULW
  2'b11          : op1, // MULHU
  default        : 0,
};
let mu_op2: logic<MUL_OP_WIDTH> = case funct3[1:0] {
  2'b00, 2'b01: abs::<XLEN>(if is_op32 ? sxt::<32, XLEN>(op2[31:0]) : op2), // MUL, MULH
>, MULW
  2'b11, 2'b10: op2, // MULHU, MULHSU
  default      : 0,
};
```

最後に、計算結果を符号拡張した値に設定します (リスト 1.28)。

▼ リスト 1.28: 計算結果を符号拡張する (muldivunit.veryl)

```
State::WaitValid: if is_mul && mu_rvalid {
  let res_signed: logic<MUL_RES_WIDTH> = if op1sign_saved != op2sign_saved ? ~mu_result +>
> 1 : mu_result;
  let res_mulhsu: logic<MUL_RES_WIDTH> = if op1sign_saved == 1 ? ~mu_result + 1 : mu_resu
>lt;
  state      = State::Finish;
  result     = case funct3_saved[1:0] {
    2'b00 : if is_op32_saved ? sxt::<32, 64>(res_signed[31:0]) : res_signed[XLEN - 1:>
>0], // MUL, MULW
    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。

1.7 符号無し割り算の実装

1.7.1 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
(muldivunit.veryl.divuremu-range.divunit)。

▼ リスト 1.29: 符号無し除算器の実装 (muldivunit.veryl)

```
module divunit #(
    param WIDTH: u32 = 0,
) (
    clk      : input  clock      ,
    rst      : input  reset      ,
    valid    : input  logic      ,
    dividend : input  logic<WIDTH>,
    divisor  : input  logic<WIDTH>,
    rvalid    : output logic      ,
    quotient : output logic<WIDTH>,
    remainder: output logic<WIDTH>,
) {
    enum State {
        Idle,
        ZeroCheck,
        SubLoop,
        Finish,
    }

    var state: State;

    var dividend_saved: logic<WIDTH * 2>;
    var divisor_saved : logic<WIDTH * 2>;

    always_comb {
        rvalid  = state == State::Finish;
        remainder = dividend_saved[WIDTH - 1:0];
    }

    var sub_count: u32;

    always_ff {
        if_reset {
            state      = State::Idle;
            quotient    = 0;
            sub_count   = 0;
            dividend_saved = 0;
            divisor_saved = 0;
        } else {
            case state {
                State::Idle: if valid {
                    state      = State::ZeroCheck;
                    dividend_saved = {1'b0 repeat WIDTH, dividend};
                    divisor_saved = {1'b0, divisor, 1'b0 repeat WIDTH - 1};
                    quotient    = 0;
                    sub_count   = 0;
                }
                State::ZeroCheck: if divisor_saved == 0 {
                    state = State::Finish;
                }
            }
        }
    }
}
```



```

        quotient = '1';
    } else {
        state = State::SubLoop;
    }
    State::SubLoop: if sub_count == WIDTH {
        state = State::Finish;
    } else {
        if dividend_saved >= divisor_saved {
            dividend_saved -= divisor_saved;
            quotient      = (quotient << 1) + 1;
        } else {
            quotient <<= 1;
        }
        divisor_saved >>= 1;
        sub_count    += 1;
    }
    State::Finish: state = State::Idle;
    default          : {}
}

}

}

```

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` クロックかけて行うことによって求めています。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 であ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 1.4 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 1.4: 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

1.7.2 divunit モジュールをインスタンス化する

divunit モジュールを multidivunit モジュールでインスタンス化します (リスト 1.30)。まだ結果は利用しません。

▼ リスト 1.30: divunit モジュールをインスタンス化する (multidivunit.veryl)

```
// divider unit
const DIV_WIDTH: u32 = XLEN;

var du_rvalid    : logic          ;
var du_quotient  : logic<DIV_WIDTH>;
var du_remainder : logic<DIV_WIDTH>;

inst du: divunit #(
    WIDTH: DIV_WIDTH,
) (
    clk          ,
    rst          ,
    valid        : ready && valid && !is_mul,
    dividend     : op1          ,
    divisor      : op2          ,
    rvalid       : du_rvalid    ,
    quotient     : du_quotient  ,
    remainder    : du_remainder ,
);
```

1.8 DIVU、REMU 命令の実装

DIVU、REMU 命令は、符号無し XLEN ビットの rs1(被除数) と符号無し 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。

multidivunit 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記述します (リスト 1.31)。

▼ リスト 1.31: divunit モジュールをインスタンス化する (multidivunit.veryl)

```
State::WaitValid: if is_mul && mu_rvalid {
    ...
} else if !is_mul && du_rvalid {
    result = case funct3_saved[1:0] {
        2'b01 : du_quotient, // DIVU
        2'b11 : du_remainder, // REMU
        default: 0,
    };
    state = State::Finish;
}
```

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。

1.9 DIV、REM 命令の実装

1.9.1 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、剰余演算の結果は被除数と同じ符号になります。符号付き剰余の絶対値は符号無し剰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剰余演算を実現できます。

1.9.2 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 に渡す値を絶対値に設定します (リスト 1.32 リスト 1.33)。

▼ リスト 1.32: op1 と op2 を生成する (muldivunit.veryl)

```
function generate_div_op (
  funct3: input logic<3> ,
  value : input logic<XLEN>,
) -> logic<DIV_WIDTH> {
  return case funct3[1:0] {
    2'b00, 2'b10: abs::<DIV_WIDTH>(value), // DIV, REM
    2'b01, 2'b11: value, // DIVU, REMU
    default      : 0,
  };
}

let du_dividend: logic<DIV_WIDTH> = generate_div_op(funct3, op1);
let du_divisor : logic<DIV_WIDTH> = generate_div_op(funct3, op2);
```

▼ リスト 1.33: divunit に渡す値を変更する (muldivunit.veryl)

```
inst du: divunit #(
  WIDTH: DIV_WIDTH,
) (
  clk           ,
  rst           ,
  valid        : ready && valid && !is_mul && !du_signed_error,
  dividend     : du_dividend ,
  divisor      : du_divisor  ,
  rvalid       : du_rvalid   ,
  quotient     : du_quotient ,
  remainder    : du_remainder,
);
```

表 1.4 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 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 1.34 リスト 1.35)。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 で計算を開始するようにしています。

▼ リスト 1.34: 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)

```
var du_signed_overflow: logic;
var du_signed_divzero : logic;
var du_signed_error   : logic;

always_comb {
    du_signed_overflow = !funct3[0] && op1[msb] == 1 && op1[msb - 1:0] == 0 && &op2;
    du_signed_divzero  = !funct3[0] && op2 == 0;
    du_signed_error    = du_signed_overflow || du_signed_divzero;
}
```

▼ リスト 1.35: 符号付き除算の例外的な結果を処理する (muldivunit.veryl)

```
State::Idle: if ready && valid {
    funct3_saved = funct3;
    is_op32_saved = is_op32;
    op1sign_saved = op1[msb];
    op2sign_saved = op2[msb];
    if is_mul {
        state = State::WaitValid;
    } else {
        if du_signed_overflow {
            state = State::Finish;
            result = if funct3[1] ? 0 : {1'b1, 1'b0 repeat XLEN - 1}; // REM : DIV
        } else if du_signed_divzero {
            state = State::Finish;
            result = if funct3[1] ? op1 : '1; // REM : DIV
        } else {
            state = State::WaitValid;
        }
    }
}
```

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 1.36)。

▼ リスト 1.36: 計算結果の符号を復元する (muldivunit.veryl)

```
} else if !is_mul && du_rvalid {
    let quo_signed: logic<DIV_WIDTH> = if op1sign_saved != op2sign_saved ? ~du_quotient + 1 >
> : du_quotient;
    let rem_signed: logic<DIV_WIDTH> = if op1sign_saved == 1 ? ~du_remainder + 1 : du_remai
>nder;
    result = case funct3_saved[1:0] {
        2'b00 : quo_signed[XLEN - 1:0], // DIV
        2'b01 : du_quotient[XLEN - 1:0], // DIVU
        2'b10 : rem_signed[XLEN - 1:0], // REM
        2'b11 : du_remainder[XLEN - 1:0], // REMU
        default: 0,
    };
    state = State::Finish;
```

```
}

```

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。

1.10 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 1.37)。

▼ リスト 1.37: (muldivunit.veryl)

```
function generate_div_op (
  is_op32: input logic      ,
  funct3 : input logic<3>   ,
  value  : input logic<XLEN>,
) -> logic<DIV_WIDTH> {
  return case funct3[1:0] {
    2'b00, 2'b10: abs::<DIV_WIDTH>(if is_op32 ? sext::<32, DIV_WIDTH>(value[31:0]) : value), // DIV, REM
    2'b01, 2'b11: if is_op32 ? {1'b0 repeat DIV_WIDTH - 32, value[31:0]} : value, // DIVU, REMU
    default      : 0,
  };
}

let du_dividend: logic<DIV_WIDTH> = generate_div_op(is_op32, funct3, op1);
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 1.38)。

▼ リスト 1.38: (muldivunit.veryl)

```
always_comb {
  if is_op32 {
    du_signed_overflow = !funct3[0] && op1[31] == 1 && op1[31:0] == 0 && op2[31:0];
    du_signed_divzero  = !funct3[0] && op2[31:0] == 0;
  } else {
    du_signed_overflow = !funct3[0] && op1[msb] == 1 && op1[msb - 1:0] == 0 && op2;
    du_signed_divzero  = !funct3[0] && op2 == 0;
  }
  du_signed_error = du_signed_overflow || du_signed_divzero;
}

```

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 1.39)。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 1.39: (muldivunit.veryl)

```

    } else if !is_mul && du_rvalid {
        let quo_signed: logic<DIV_WIDTH> = if op1sign_saved != op2sign_saved ? ~du_quotient + 1 >
    > : du_quotient;
        let rem_signed: logic<DIV_WIDTH> = if op1sign_saved == 1 ? ~du_remainder + 1 : du_remai>
    >nder;

    let resultX : UIntX = case funct3_saved[1:0] {
        2'b00 : quo_signed[XLEN - 1:0], // DIV
        2'b01 : du_quotient[XLEN - 1:0], // DIVU
        2'b10 : rem_signed[XLEN - 1:0], // REM
        2'b11 : du_remainder[XLEN - 1:0], // REMU
        default: 0,
    };
    state = State::Finish;
    result = if is_op32_saved ? sext::<32, 64>(resultX[31:0]) : resultX;
}

```

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 2 章

例外の実装

2.1 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^{*1}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不正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)^{*2}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 レジスタにプログラムカウンタの値を書き込む
3. プログラムカウンタを 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。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。

^{*1} 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。

^{*2} 割り込みは第 7 章「M-mode の実装 (2. 割り込み、Zicntr 拡張の実装)」で実装します。

2.2 例外情報の伝達

2.2.1 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どうかは命令が ECALL であるかどうか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作っておきます。

まず、例外が発生するかどうか、例外の種類を示す値をまとめた `ExceptionInfo` 構造体を定義します (リスト 2.1)。

▼ リスト 2.1: ExceptionInfo 構造体を定義する (corectrl.veryl)

```
// 例外の情報を保存するための型
struct ExceptionInfo {
    valid: logic    ,
    cause: CsrCause,
}
```

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 2.2、リスト 2.3)。

▼ リスト 2.2: EX ステージの FIFO に ExceptionInfo を追加する (core.veryl)

```
struct exq_type {
    addr: Addr      ,
    bits: Inst      ,
    ctrl: InstCtrl  ,
    imm : UIntX     ,
    expt: ExceptionInfo,
}
```

▼ リスト 2.3: MEM ステージの FIFO に ExceptionInfo を追加する (core.veryl)

```
struct memq_type {
    addr      : Addr      ,
    bits      : Inst      ,
    ctrl      : InstCtrl  ,
    imm       : UIntX     ,
    expt      : ExceptionInfo ,
    alu_result: UIntX     ,
    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 2.4)。

▼リスト 2.4: ID ステージで ECALL 命令を判定する (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;
    // exception
    exq_wdata.expt.valid = ids_inst_bits == 32'h00000073; // ECALL
    exq_wdata.expt.cause = CsrCause::ENVIRONMENT_CALL_FROM_M_MODE;
}

```

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 2.5)。

▼リスト 2.5: EX ステージから MEM ステージに例外情報を渡す (core.veryl)

```

always_comb {
    // EX -> MEM
    exq_rready          = memq_wready && !exs_stall;
    ...
    memq_wdata.jump_addr = if inst_is_br(exs_ctrl) ? exs_pc + exs_imm : exs_alu_result & ~>
>1;
    memq_wdata.expt      = exq_rdata.expt;
}

```

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 2.6、リスト 2.7、リスト 2.8)。

▼リスト 2.6: 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)

```

module csrunit (
    clk      : input  clock          ,
    rst      : input  reset          ,
    valid    : input  logic          ,
    pc       : input  Addr           ,
    ctrl     : input  InstCtrl       ,
    expt_info : input  ExceptionInfo ,
    rd_addr  : input  logic          <5> ,

```

▼リスト 2.7: MEM ステージの例外情報の変数を作成する (core.veryl)

```

//////////////////// MEM Stage //////////////////////
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_expt    : ExceptionInfo = memq_rdata.expt;
let mems_rd_addr : logic          <5> = mems_inst_bits[11:7];

```

▼ リスト 2.8: csrunit モジュールに例外情報を供給する (core.veryl)

```

inst csru: csrunit (
    clk          ,
    rst          ,
    valid       : mems_valid      ,
    pc          : mems_pc        ,
    ctrl        : mems_ctrl      ,
    expt_info: mems_expt        ,
    rd_addr     : mems_rd_addr   ,

```

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 2.9、リスト 2.10)。

▼ リスト 2.9: csrunit モジュールでの ECALL 命令の判定を削除する (csrunit.veryl)

```

// CSRR(W|S|C)[I]命令かどうか
let is_wsc: logic = ctrl.is_csr && ctrl.func3[1:0] != 0;
// ECALL命令かどうか
let is_ecall: logic = ctrl.is_csr && csr_addr == 0 && rs1[4:0] == 0 && ctrl.func3 == 0 &&
rd_addr == 0;

```

▼ リスト 2.10: ExceptionInfo を使って例外を起こす (csrunit.veryl)

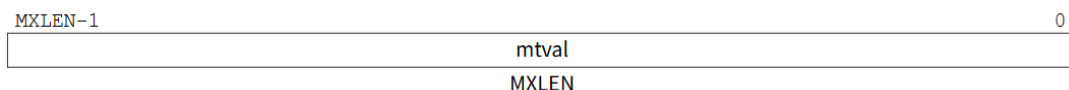
```

// Exception
let raise_expt : logic = valid && expt_info.valid;
let expt_cause : UIntX = expt_info.cause;
let expt_vector: Addr  = mtvec;

```

2.2.2 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処理を実行します。mcause レジスタを読むことでどの例外が発生したかを判別することができますが、その例外の詳しい情報を知りたいことがあります。



▲ 図 2.1: mtval レジスタ

RISC-V には、例外が発生した時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 2.1)。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 2.11)。

▼ リスト 2.11: tval を ExceptionInfo に追加する (corectrl.veryl)

```

struct ExceptionInfo {
    valid: logic ,

```

```

    cause: CsrCause,
    value: UIntX,
}

```

ECALL 命令は mtval に書き込むような情報がないので 0 に設定しておきます (リスト 2.12)。

▼ リスト 2.12: ECALL 命令の tval を設定する (corectrl.veryl)

```

// exception
exq_wdata.expt.valid = ids_inst_bits == 32'h00000073; // ECALL
exq_wdata.expt.cause = CsrCause::ENVIRONMENT_CALL_FROM_M_MODE;
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 2.13)。

▼ リスト 2.13: mtval のアドレスを定義する (eei.veryl)

```

enum CsrAddr: logic<12> {
    MTVEC = 12'h305,
    MEPC = 12'h341,
    MCAUSE = 12'h342,
    MTVAL = 12'h343,
    LED = 12'h800,
}

```

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 2.14、リスト 2.15、リスト 2.16、リスト 2.17、リスト 2.18)。

▼ リスト 2.14: mtval の書き込みマスクを定義する (csrunit.veryl)

```

const MTVAL_WMASK : UIntX = 'hffff_ffff_ffff_ffff;

```

▼ リスト 2.15: mtval 変数を作成する (csrunit.veryl)

```

var mtvec : UIntX;
var mepc : UIntX;
var mcause: UIntX;
var mtval : UIntX;

```

▼ リスト 2.16: mtval の読み込みデータ、書き込みマスクを設定する (csrunit.veryl)

```

always_comb {
    // read
    rdata = case csr_addr {
        ...
        CsrAddr::MTVAL : mtval,
        ...
    };
    // write
    wmask = case csr_addr {
        ...
        CsrAddr::MTVAL : MTVAL_WMASK,
        ...
    };
}

```

```
};
```

▼ リスト 2.17: mtval 変数をリセットする (csrunit.veryl)

```
always_ff {
  if_reset {
    mtvec = 0;
    mepc = 0;
    mcause = 0;
    mtval = 0;
    led = 0;
  }
}
```

▼ リスト 2.18: mtval に書き込めるようにする (csrunit.veryl)

```
} else {
  if is_wsc {
    case csr_addr {
      ...
      CsrAddr::MTVAL : mtval = wdata;
      ...
    }
  }
}
```

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 2.19、リスト 2.20)。

▼ リスト 2.19: tval を変数に割り当てる (csrunit.veryl)

```
let raise_expt : logic = valid && expt_info.valid;
let expt_cause : UIntX = expt_info.cause;
let expt_value : UIntX = expt_info.value;
let expt_vector: Addr = mtvec;
```

▼ リスト 2.20: 例外が発生するとき、mtval に tval を書き込む (csrunit.veryl)

```
if valid {
  if raise_trap {
    if raise_expt {
      mepc = pc;
      mcause = trap_cause;
      mtval = expt_value;
    }
  }
}
```

2.3 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 2.21)。

▼ リスト 2.21: Breakpoint 例外の cause を定義する (eei.veryl)

```
enum CsrCause: UIntX {
    BREAKPOINT = 3,
    ENVIRONMENT_CALL_FROM_M_MODE = 11,
}
```

ID ステージで EBREAK 命令の判定と例外情報の設定を行います (リスト 2.22)。

▼ リスト 2.22: ID ステージで EBREAK 命令を判定する (core.veryl)

```
exq_wdata.expt = 0;
if ids_inst_bits == 32'h00000073 {
    // ECALL
    exq_wdata.expt.valid = 1;
    exq_wdata.expt.cause = CsrCause::ENVIRONMENT_CALL_FROM_M_MODE;
    exq_wdata.expt.value = 0;
} else if ids_inst_bits == 32'h00100073 {
    // EBREAK
    exq_wdata.expt.valid = 1;
    exq_wdata.expt.cause = CsrCause::BREAKPOINT;
    exq_wdata.expt.value = ids_pc;
}
```

2.4 Illegal instruction 例外の実装

Illegal instruction 例外は、実行することが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 が認識することができない不正な命令ビット列を実行しようとした場合、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。

2.4.1 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、つまりデコードすることが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 opcode が未知の命令は何もしない命令として実行し、それ以外の命令については何も対処していません。inst_decoder モジュールを変更し、実装していない命令で例外が発生するようにします。

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 2.23、リスト 2.24)。

▼ リスト 2.23: valid ポートを追加する (inst_decoder.veryl)

```
module inst_decoder (
    bits : input Inst    ,
    valid: output logic  ,
    ctrl : output InstCtrl,
    imm  : output UIntX  ,
) {
```

▼ リスト 2.24: inst_decoder モジュールの valid ポートと変数を接続する (core.veryl)

```
let ids_valid      : logic    = if_fifo_rvalid;
let ids_pc        : Addr     = if_fifo_rdata.addr;
let ids_inst_bits  : Inst     = if_fifo_rdata.bits;
var ids_inst_valid: logic    ;
var ids_ctrl      : InstCtrl;
var ids_imm       : UIntX    ;

inst decoder: inst_decoder (
    bits : ids_inst_bits ,
    valid: ids_inst_valid,
    ctrl : ids_ctrl      ,
    imm  : ids_imm       ,
);
```

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 2.25)。

▼ リスト 2.25: 命令の有効判定を行う (inst_decoder.veryl)

```
valid = case op {
    OP_LUI, OP_AUIPC, OP_JAL, OP_JALR: T,
    OP_BRANCH                      : f3 != 3'b010 && f3 != 3'b011,
    OP_LOAD                        : f3 != 3'b111,
    OP_STORE                       : f3[2] == 1'b0,
    OP_OP                          : case f7 {
        7'b0000000 : T, // RV32I
        7'b0100000 : f3 == 3'b000 || f3 == 3'b101, // SUB, SRA
        7'b0000001 : T, // RV32M
        default    : F,
    },
    OP_OP_IMM: case f3 {
        3'b001 : f7[6:1] == 6'b000000, // SLLI (RV64I)
        3'b101 : f7[6:1] == 6'b000000 || f7[6:1] == 6'b010000, // SRLI, SRAI (RV64I)
        default : T,
    },
    OP_OP_32 : case f7 {
        7'b0000001: f3 == 3'b000 || f3[2] == 1'b1, // RV64M
        7'b0000000: f3 == 3'b000 || f3 == 3'b001 || f3 == 3'b101, // ADDW, SLLW, SRLW
        7'b0100000: f3 == 3'b000 || f3 == 3'b101, // SUBW, SRAW
        default   : F,
    },
    OP_OP_IMM_32: case f3 {
        3'b000 : T, // ADDIW
```

```

3'b001      : f7 == 7'b0000000, // SLLIW
3'b101      : f7 == 7'b0000000 || f7 == 7'b0100000, // SRLIW, SRAIW
default     : F,
},
OP_SYSTEM: f3 != 3'b000 && f3 != 3'b100 || // CSRR(W|S|C)[I]
bits == 32'h00000073 || // ECALL
bits == 32'h00100073 || // EBREAK
bits == 32'h30200073, //MRET
OP_MISC_MEM: T, // FENCE
default     : F,
};

```

riscv-tests でメモリ読み書きの順序を保証する FENCE 命令^{*3}を使用しているため、opcode が OP-MISC である命令を合法的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 2.26)。

▼ リスト 2.26: OP-MISC のビット列を定義する (eei.veryl)

```
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 2.27)。

▼ リスト 2.27: Illegal instruction 例外の cause を定義する (eei.veryl)

```
enum CsrCause: UIntX {
    ILLEGAL_INSTRUCTION = 2,
    BREAKPOINT = 3,
    ENVIRONMENT_CALL_FROM_M_MODE = 11,
}
```

`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 2.28)。`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 2.28: 不正な命令のとき、例外を発生させる (core.veryl)

```

exq_wdata.expt = 0;
if !ids_inst_valid {
    // illegal instruction
    exq_wdata.expt.valid = 1;
    exq_wdata.expt.cause = CsrCause::ILLEGAL_INSTRUCTION;
    exq_wdata.expt.value = {1'b0 repeat XLEN - ILEN, ids_inst_bits};
} else if ids_inst_bits == 32'h00000073 {

```

2.4.2 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。

^{*3} 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こしません。

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 2.30、リスト 2.29、リスト 2.31)。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 2.29: csrunit モジュールのポート定義を変更する (csrunit.veryl)

```
module csrunit (
    clk      : input  clock      ,
    rst      : input  reset      ,
    valid    : input  logic      ,
    pc       : input  Addr       ,
    inst_bits : input  Inst       ,
    ctrl     : input  InstCtrl    ,
    expt_info : input  ExceptionInfo ,
    rd_addr  : input  logic       <5> ,
    csr_addr  : input  logic       <12> ,
    rs1_addr  : input  logic       <5> ,
    rs1_data  : input  UIntX      ,
    rdata     : output UIntX      ,
    raise_trap : output logic     ,
    trap_vector : output Addr     ,
    led       : output UIntX      ,
) {
```

▼ リスト 2.30: csrunit モジュールのポート定義を変更する (core.veryl)

```
inst csru: csrunit (
    clk      :      ,
    rst      :      ,
    valid    : mems_valid ,
    pc       : mems_pc    ,
    inst_bits : mems_inst_bits ,
    ctrl     : mems_ctrl  ,
    expt_info : mems_expt ,
    rd_addr  : mems_rd_addr ,
    csr_addr  : mems_inst_bits[31:20],
    rs1_addr  : memq_rdata.rs1_addr ,
    rs1_data  : memq_rdata.rs1_data ,
    rdata     : csru_rdata ,
    raise_trap : csru_raise_trap ,
    trap_vector : csru_trap_vector ,
)
```



```
        led
    );
```

▼ リスト 2.31: rs1 の変更に対応する*⁴ (csrunit.veryl)

```
let wsource: UIntX = if ctrl.func3[2] ? {1'b0 repeat XLEN - 5, rs1_addr} : rs1_data;
wdata = case ctrl.func3[1:0] {
    2'b01 : wsource,
    2'b10 : rdata | wsource,
    2'b11 : rdata & ~wsource,
    default: 'x,
} & wmask | (rdata & ~wmask);
```

命令の func3 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*⁵(リスト 2.32)。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 2.32: 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)

```
// CSRR(W|S|C)[I]命令かどうか
let is_wsc: logic = ctrl.is_csr && ctrl.func3[1:0] != 0;
// MRET命令かどうか
let is_mret: logic = inst_bits == 32'h30200073;

// Check CSR access
let will_not_write_csr : logic = (ctrl.func3[1:0] == 2 || ctrl.func3[1:0] == 3) && rs1_addr == 0; // set/clear with source = 0
let expt_write_readonly_csr: logic = is_wsc && !will_not_write_csr && csr_addr[11:10] == 2'b>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 2.33)。

▼ リスト 2.33: 読み込み専用 CSR の書き込みで例外を発生させる (csrunit.veryl)

```
let raise_expt: logic = valid && (expt_info.valid || expt_write_readonly_csr);
let expt_cause: UIntX = switch {
    expt_info.valid : expt_info.cause,
    expt_write_readonly_csr: CsrCause::ILLEGAL_INSTRUCTION,
    default : 0,
};
let expt_value: UIntX = switch {
    expt_info.valid : expt_info.value,
    expt_cause == CsrCause::ILLEGAL_INSTRUCTION: {1'b0 repeat XLEN - $bits(Inst), inst_bits}>,
    default : 0,
};
let expt_vector: Addr = mtvec;
```

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

*⁴ 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あります。

*⁵ 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 2.34、リスト 2.35、リスト 2.36)。

▼ リスト 2.34: トラップが発生したかを示す logic を wbq_type に追加する (core.veryl)

```
struct wbq_type {
    ...
    csr_rdata : UIntX    ,
    raise_trap: logic    ,
}
```

▼ リスト 2.35: トラップが発生したかを WB ステージに伝える (core.veryl)

```
wbq_wdata.raise_trap = csru_raise_trap;
```

▼ リスト 2.36: 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)

```
always_ff {
    if wbs_valid && wbs_ctrl.rwb_en && !wbq_rdata.raise_trap {
        regfile[wbs_rd_addr] = wbs_wb_data;
    }
}
```

2.5 命令アドレスのミスアライン例外

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 5 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*6。プログラムカウンタの遷移先が整列されていない場合、ジャンプ命令、または分岐命令で例外が発生します。分岐命令の場合、分岐が成立する場合にしか例外が発生しません。

CsrCause 型に Instruction address misaligned 例外の cause を追加します (リスト 2.37)。

▼ リスト 2.37: Instruction address misaligned 例外の cause を定義する (eei.veryl)

```
enum CsrCause: UIntX {
    INSTRUCTION_ADDRESS_MISALIGNED = 0,
    ILLEGAL_INSTRUCTION = 2,
    BREAKPOINT = 3,
    ENVIRONMENT_CALL_FROM_M_MODE = 11,
}
```

*6 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、トラップ後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 2.38)。tval は遷移しようとしたアドレスになることに注意してください。

▼ リスト 2.38: EX ステージで Instruction address misaligned 例外の判定を行う (core.veryl)

```
memq_wdata.jump_addr = if inst_is_br(exs_ctrl) ? exs_pc + exs_imm : exs_alu_result & ~>
>1;
// exception
let instruction_address_misaligned: logic = memq_wdata.br_taken && memq_wdata.jump_addr[>
>1:0] != 2'b000;
memq_wdata.expt          = exq_rdata.expt;
if !memq_rdata.expt.valid {
    if instruction_address_misaligned {
        memq_wdata.expt.valid = 1;
        memq_wdata.expt.cause = CsrCause::INSTRUCTION_ADDRESS_MISALIGNED;
        memq_wdata.expt.value = memq_wdata.jump_addr;
    }
}
```

2.6 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*7。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 2.39)。

▼ リスト 2.39: 例外の cause を定義する (eei.veryl)

```
enum CsrCause: UIntX {
    INSTRUCTION_ADDRESS_MISALIGNED = 0,
    ILLEGAL_INSTRUCTION = 2,
    BREAKPOINT = 3,
    LOAD_ADDRESS_MISALIGNED = 4,
    STORE_AMO_ADDRESS_MISALIGNED = 6,
    ENVIRONMENT_CALL_FROM_M_MODE = 11,
}
```

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 2.40)。

▼ リスト 2.40: EX ステージで例外の判定を行う (core.veryl)

*7 例外を発生させず、そのようなロードストア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。

```

    let instruction_address_misaligned: logic = memq_wdata.br_taken && memq_wdata.jump_addr[>
>1:0] != 2'b00;
    let loadstore_address_misaligned : logic = inst_is_memop(exs_ctrl) && case exs_ctrl.fun>
>ct3[1:0] {
        2'b00 : 0, // B
        2'b01 : exs_alu_result[0] != 1'b0, // H
        2'b10 : exs_alu_result[1:0] != 2'b0, // W
        2'b11 : exs_alu_result[2:0] != 3'b0, // D
        default: 0,
    };
    memq_wdata.expt = exq_rdata.expt;
    if !memq_rdata.expt.valid {
        if instruction_address_misaligned {
            memq_wdata.expt.valid = 1;
            memq_wdata.expt.cause = CsrCause::INSTRUCTION_ADDRESS_MISALIGNED;
            memq_wdata.expt.value = memq_wdata.jump_addr;
        } else if loadstore_address_misaligned {
            memq_wdata.expt.valid = 1;
            memq_wdata.expt.cause = if exs_ctrl.is_load ? CsrCause::LOAD_ADDRESS_MISALIGNED>
> : CsrCause::STORE_AMO_ADDRESS_MISALIGNED;
            memq_wdata.expt.value = exs_alu_result;
        }
    }
}

```

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 2.41)。

▼ リスト 2.41: 例外が発生するとき、memunit の valid を 0 にする (core.veryl)

```

inst memu: memunit (
    clk
    rst
    valid : mems_valid && !mems_expt.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
);

```

第 3 章

Memory-mapped I/O の実装

3.1 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 TODO 図のように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) に格納されたブートローダから命令の実行を開始します。ブートローダ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 と ROM 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。デバイスとメモリ空間の対応は TODO 図のように設定します。

3.2 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。新しく RAM の開始アドレス、メモリバスのデータ幅、ROM の範囲を示す定数を定義してください ()。

MEM_DATA_WIDTH、MEM_ADDR_WIDTH を使っている部分を MEM-BUS_DATA_WIDTH に置き換えます。

3.3 コントローラ

3.3.1 コントローラを実装する

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 ()。

`mmio_controller` モジュールは、`membus` 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レスポンスを返すモジュールです。 `State` に応じて次のように動作します。

TODO 各状態の説明

まだアクセス対象のデバイスを実装していないため、常に `0` を読み取り、書き込みは無視します。

3.3.2 コントローラを接続する

`core` モジュールと `mmio_controller` モジュールを接続します。既存の `memory` モジュールはコメントアウトしてください。

`top` モジュールでコントローラをインスタンス化します ()。

3.4 ROM の実装

3.5 RAM のベースアドレスの変更

3.6 RAM の実装

3.7 デバッグ用 IO の実装

ROM から RAM にジャンプする

終わり=> Web 版で UART を実装します

第 4 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 (TODO table)、ロードした値を加工した値をメモリにストアする操作を単一の命令で実装する Zaamo 拡張 (TODO table) 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が並列、並行して実行されるとき、ソフトウェア間で同期をとりながら実行することができます。

4.1 アトミック操作

4.1.1 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、他のシステムからは、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することができません。

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 1 から N までの和を求めるプログラムを考えます (図 TODO)。2 つのコアで同時にアドレス X、または Y の値を変更しようとするとき、命令の実行順序によって最終的な値が 1 つのコアで実行した場合と異なってしまいます。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。

4.1.2 Zaamo 拡張

AMOADD 命令はロード、加算、ストアを行う単一の命令です。Zaamo 拡張は他の簡単な操作を行う命令も提供しています。

TODO table

4.1.3 Zlrrsc 拡張

LR 命令と SC 命令はそれぞれ Load-Reserved、Store Conditional 操作を実現する命令です。LR、SC 命令は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、予約セット (Reservation set) に指定されたアドレスを登録します。

SC 命令

予約セットに指定されたアドレスが存在する場合、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 ()。

同時に他のコアが同じプログラムを実行するとき、間違った値の書き込みは 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つのコアで実行した場合と同一の結果になります。予約セットのサイズは実装によって異なります。

4.2 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。

TODO それぞれの図

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 $aq=1$ 、 $rl=1$ 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^{*1}。

4.3 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うのが好ましいです。具体的には、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モジュールを追加、実装します。

TODO RAM を外に出す→top から消すのをやめる

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 ()。

TODO amounit は connect ではなくステートを持たせる

amounit モジュールを top モジュールでインスタンス化して、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 ()。

4.4 命令のデコード

TODO 命令の図

A 拡張の命令の opcode はすべて OP-AMO(7'b0101111) です (TODO 図)。それぞれの命令は funct5 と funct3 で区別することができます (TODO テーブル)。

TODO funct5 と命令の対応のテーブル

4.4.1 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 ()。

▼ is_amo フラグを追加する (corectrl.veryl)

また、A 拡張の命令を区別するための列挙型を定義します ()。それぞれ、命令の funct5 と対応していることを確認してください。

^{*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。

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 ()。その他の opcode の is_amo は F に設定してください。

▼ is_amo フラグを追加する (corectrl.veryl)

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 ()。

▼ A 拡張の命令のとき、valid フラグを立てる

4.4.2 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 RV64I のロードストア命令で使用する rs1 と即値を足し合わせたものとは異なるため、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 ()。

TODO ここで memu_addr を作る

▼ core

4.4.3 メモリインターフェースを変更する

amounit モジュールに A 拡張の操作を指示するために、データ用のメモリインターフェースに変更が必要です。memunit モジュールと amounit モジュールの間のインターフェースを新しく定義します。

membus_if インターフェースに is_amo フラグ、aq ビット、rl ビット、AM00p 型を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 ()。wmask を DATA_WIDTH ビットに展開する関数も定義しています。

memunit モジュールから amounit モジュールへの接続で core_data_if を使用するように変更します ()。

memunit モジュールで、is_amo、aq、rl、am00p に値を割り当てます ()。

4.5 Zalrsc 拡張の実装

予約セットのサイズは実装が自由に決めることができるため、本書では1つのアドレスのみ保持できるようにします。

4.5.1 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 ()。

LR 命令を実行するとき、予約セットにアドレスを登録し、ロード結果を返すようにします。

4.5.2 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、amounit 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 ()。

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 ()。 `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。

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` に移動させます ()。

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` に移動させます ()。

TODO 遷移がおかしいね (WaitReady)

4.6 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。

32ビット演算と64ビット演算を区別するために、`core_data_if` インターフェースに `is_op32` フラグを追加します ()。

▼ `core_data_if`

▼ `memunit`

Zaamo 拡張の命令かどうかを判定する関数を作成します ()。

ロードしたデータと `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 ()。

4.6.1 Zaamo 拡張の命令を実装する

TODO 図

Zaamo 拡張の命令を TODO 図のような状態遷移で処理するようにします。`is_zaamo_fetched` はロード済かを示すフラグで、`zaamo_fetched_data` はロードしたデータを保存するレジスタです。

`is_zaamo_fetched` と `zaamo_fetched_data` レジスタを作成します。

▼ `reset`

TODO 図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 ()。

TODO 図に基づいて状態を遷移させます ()。

ストアが完了したら結果を返し、新しく要求を受け入れます ()。

第 5 章

C 拡張の実装

5.1 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、それぞれ命令の下位ビットで何ビット幅の命令か判断することができます (TODO 図)。

TODO 図

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*¹。

RV32I の CPU に実装される C 拡張には表 TODO の RVC 命令が定義されています。RV64I の CPU に実装される C 拡張には表 TODO に加えて表 TODO の RVC 命令が定義されています。一部の RV32I の RVC 命令は RV64I で別の命令に置き換わっていることに注意してください。

表 TODO 表 TODO

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すが、基本編では F、D 拡張を実装しないため解説しません。

5.2 IALIGN の変更

TODO 図

「2.5 命令アドレスのミスアライン例外」(p.35) 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

*¹ Zc*拡張の一部の命令は複数の命令になります

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することができます (TODO 図)。

eei パッケージの IALIGN を変更します ()。

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 ()。mtvec レジスタの下位 2 ビットはモードを指定するために使用されているため、変更の必要はありません。

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 ()。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。

5.3 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 IF ステージで扱えるようにする

最終的な命令フェッチ処理の構成は図 TODO のようになります。

TODO core <-> inst_fetcher <-> mem の図

5.4 命令フェッチモジュールの実装

5.4.1 既存の動作を実現する

5.4.2 16 ビット境界に配置された 32 ビット幅の命令をサポートする

5.5 RVC 命令のデコード

5.5.1 圧縮命令を 32 ビット幅の命令に変換する

5.5.2 圧縮命令フラグを実装する

interface やら core やらインスタンス化してつなぐ

第Ⅱ部

特権/割り込みの実装

第 6 章

M-mode の実装 (1. CSR の実装)

6.1 概要

「第 II 部 RV64IMAC の実装」では、RV64IMAC と例外、MMIO を実装しました。「第 III 部 特権/割り込みの実装」次のような機能を実装します。

- 特権 (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)

6.1.1 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) の上で動くように作成されています。「OS の上で動く」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、アプリケーションに対して簡単にそれを扱うためのインターフェースを提供します。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する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) 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。

特権レベルは次の 3 種類^{*1}が用意されています (TODO table)。それぞれの特権レベルは 2 ビットの数値で表すことができます。

TODO table

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ったり、高い特権レベルでしか利

^{*1} V 拡張が実装されている場合、さらに仮想化のためのレベルが追加されます。

用できない機能が定義されています。

6.1.2 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、TODO テーブル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 のみになります。

TODO M, MU, MSU テーブル

本章では M-mode 向けの CSR の一部を実装します。実装する機能、レジスタと章の対応は表 TODO の通りです

table 実装する機能とレジスタと章の対応

6.2 CSR のアドレスの追加

6.3 misa レジスタ

6.4 mimpid レジスタ

6.5 mhartid レジスタ

6.6 mstatus レジスタ

6.7 mcycle レジスタ

6.8 minstret レジスタ

6.9 mscratch レジスタ

第 7 章

M-mode の実装 (2. 割り込み、Zicntr 拡張の実装)

7.1 割り込みとは何か？

7.2 ACLINT

7.3 ソフトウェア割り込みの実装

7.3.1 msip レジスタを実装する

7.3.2 mip、mie レジスタを実装する

7.3.3 mstatus の MIE、MPIE ビットを実装する

7.3.4 割り込み処理の実装

7.4 タイマ割り込みの実装

7.4.1 タイマ割り込みとは何か？

7.4.2 mtime、mtimecmp レジスタを実装する

7.4.3 割り込み要因を設定する

7.5 WFI 命令の実装

7.6 Zicntr 拡張の実装

TODO memunit を止められていない

第 8 章

U-mode の実装

8.1 misa レジスタの変更

U ビット

8.2 mstatus の UXL、TW ビットの実装

8.3 権限レベルの実装

8.4 CSR の読み書き権限の確認

8.5 mcounteren レジスタの実装

8.6 MRET 命令の実行制限

8.7 **トラップ処理の変更**

8.7.1 mstatus の MPP ビットを実装する

8.7.2 ECALL の要因を変更する

8.7.3 割り込み条件を変更する

第 9 章

S-mode の実装

権限レベルを追加する

9.1 CSR のアドレスの追加

9.2 misa レジスタの変更

9.3 mstatus レジスタの変更

9.3.1 SXL

9.3.2 MPP

9.4 scounteren レジスタの実装

9.5 sstatus レジスタの実装

9.6 stvec レジスタの実装

9.7 sscratch レジスタの実装

9.8 sepc レジスタの実装

9.9 scause レジスタの実装

9.10 stval レジスタの実装

9.11 トラップ処理の変更

9.11.1 sstatus の SIE、SPIE、SPP レジスタの実装

9.11.2 SRET 命令の実装

mstatus.TSR

9.11.3 mip レジスタの変更

9.12 トラップの委譲の実装

9.12.1 medeleg、mideleg レジスタを作成する

9.12.2 sie、sip レジスタを実装する

9.12.3 トラップの委譲を実装する

第 10 章

仮想記憶システムの実装

10.1 仮想記憶とは何か

10.2 RISC-V のページング

10.3 メモリアクセス例外の実装

10.3.1 例外情報を作成する

10.3.2 例外の発生アドレスを特定する

10.4 アドレス変換モジュールの作成

10.5 satp レジスタの作成

10.6 mstatus の MXR、SUM ビットの作成

10.7 Sv39 の実装

10.8 mstatus の MPRV ビットの実装

10.9 SFENCE.VMA 命令の実装

10.10 mstatus の TVM ビットの実装

10.11 satp、mstatus レジスタの変更の対応

第 11 章

PLIC の実装

11.1 概要

11.2 デバッグ入力の実装

11.3 PLIC モジュールの作成

11.4 外部割込みの実装

第 12 章

Linux を動かす

本章では著名な OS である Linux を動かします。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

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 (のまねっきー) 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^{*1})
- 井田健太 (@ciniml^{*2})
- 内田公太 (@uchan_nos^{*3})
- 初田直也 (@dalance^{*4})

筆者が CPU を作り始めたのは、井田さんの「RISC-V と Chisel で学ぶ はじめての CPU 自作^{*5}」を読んだのがきっかけでした。この本が無ければ、筆者は CPU を作ろうとは思わなかったかと思います。

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。

^{*1} <https://github.com/taichi-ishitani>

^{*2} <https://github.com/ciniml>

^{*3} https://x.com/uchan_nos

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

^{*5} <https://gihyo.jp/book/2021/978-4-297-12305-5>

VeryI 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/>

著 者 阿部奏太

発行者 阿部奏太

連絡先 kanastudio@oekaki.chat

印刷所 株式会社栄光

© 2025 ミーミミ研究室