

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。

まえがき

TODO。Linux を起動できるくらいの CPU にするよ。一部は Web 版だけだよ

注意

本書は「Veryl で作る CPU 基本編」の第 II 部と第 III 部です。第 I 部の内容は含まれていません。

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/techbookfest18-support-page>

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)。

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、それ以後がユーザ入力

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

目次

まえがき

i

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 符号無しの乗算器の実装	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無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 符号無し除算の実装	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	39
3.3	mmio_controller モジュールの作成	41
3.4	RAM の接続	45
3.4.1	mmio_controller モジュールに RAM を追加する	45
3.4.2	RAM と mmio_controller モジュールを接続する	47
3.4.3	PC の初期値の変更	49
3.5	ROM の実装	51
3.5.1	mmio_controller モジュールに ROM を追加する	51
3.5.2	ROM の初期値のパラメータを作成する	52
3.5.3	ROM と mmio_controller モジュールを接続する	54
3.5.4	ROM から RAM にジャンプする	55
3.6	デバッグ用の入出力デバイスの実装	56
3.6.1	デバイスのアドレスを設定する	56
3.6.2	mmio_controller モジュールにデバイスを追加する	57
3.6.3	出力を実装する	59
3.6.4	出力をテストする	60
3.6.5	riscv-tests に対応する	63
3.6.6	入力を実装する	64
3.6.7	入力をテストする	67
第 4 章	A 拡張の実装	68
4.1	アトミック操作	68
4.1.1	アトミック操作とは何か?	68
4.1.2	Zaamo 拡張	68
4.1.3	Zalrsc 拡張	69
4.1.4	命令の順序	69
4.2	命令のデコード	70
4.2.1	is_amo フラグを実装する	70
4.2.2	アドレスを変更する	72
4.2.3	ライトバックする条件を変更する	72
4.3	amounit モジュールの作成	73
4.3.1	インターフェースを作成する	73
4.3.2	amounit モジュールの作成	74

4.4	Zalrsc 拡張の実装	79
4.4.1	LR.W、LR.D 命令を実装する	79
4.4.2	SC.W、SC.D 命令を実装する	80
4.5	Zaamo 拡張の実装	83
第 5 章 C 拡張の実装		86
5.1	概要	86
5.2	IALIGN の変更	87
5.3	実装方針	87
5.4	命令フェッチモジュールの実装	88
5.4.1	インターフェースを作成する	88
5.4.2	core モジュールの IF ステージを削除する	88
5.4.3	inst_fetcher モジュールを作成する	89
5.4.4	inst_fetcher モジュールと core モジュールを接続する	94
5.5	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	95
5.6	RVC 命令の変換	97
5.6.1	RVC 命令フラグの実装	97
5.6.2	32 ビット幅の命令に変換する	99
5.6.3	RVC 命令を発行する	104
第 II 部 特権/割り込みの実装		107
第 6 章 M-mode の実装 (1. CSR の実装)		108
6.1	概要	108
6.1.1	特権レベルとは何か？	108
6.1.2	特権レベルの実装順序	109
6.1.3	XLEN の定義	110
6.2	misa レジスタ (Machine ISA)	110
6.3	mimpid レジスタ (Machine Implementation ID)	111
6.4	mhartid レジスタ (Hart ID)	111
6.5	mstatus レジスタ (Machine Status)	112
6.6	ハードウェアパフォーマンスマニタ	113
6.6.1	mcycle レジスタ	114
6.6.2	minstret レジスタ	114
6.7	mscratch レジスタ (Machine Scratch)	115
第 7 章 M-mode の実装 (2. 割り込みの実装)		118
7.1	概要	118
7.1.1	割り込みとは何か？	118

7.1.2	RISC-V の割り込み	118
7.1.3	割り込みの優先順位	119
7.1.4	割り込みの原因 (cause)	120
7.1.5	ACLINT (Advanced Core Local Interruptor)	120
7.2	aclint_memory モジュールの作成	121
7.2.1	インターフェースを作成する	121
7.2.2	aclint_memory モジュールを作成する	121
7.2.3	mmio_controller モジュールに ACLINT を追加する	122
7.2.4	ACLINT と mmio_controller、csrunit モジュールを接続する	123
7.3	ソフトウェア割り込みの実装 (MSWI)	125
7.3.1	MSIP レジスタを実装する	125
7.3.2	mip、mie レジスタを実装する	126
7.3.3	mstatus の MIE、MPIE ビットを実装する	127
7.3.4	割り込み処理の実装	128
7.3.5	ソフトウェア割り込みをテストする	130
7.4	mtvec の Vectored モードの実装	131
7.5	タイマ割り込みの実装 (MTIMER)	132
7.5.1	タイマ割り込みの仕組み	132
7.5.2	MTIME、MTIMECMP レジスタを実装する	132
7.5.3	mip.MTIP、割り込み原因を設定する	134
7.5.4	タイマ割り込みをテストする	134
7.6	WFI 命令の実装	135
7.7	time、instret、cycle レジスタの実装	136
第 8 章	U-mode の実装	140
8.1	misa.Extensions の変更	140
8.2	mstatus.UXL の実装	140
8.3	mstatus.TW の実装	141
8.4	mstatus.MPP の実装	141
8.5	CSR のアクセス権限の確認	143
8.6	mcounteren レジスタの実装	144
8.7	MRET 命令の実行を制限する	145
8.8	ECALL 命令の cause を変更する	146
8.9	割り込み条件の変更	147
第 9 章	S-mode の実装 (1. CSR の実装)	148
9.1	misa.Extensions、mstatus.SXL、mstatus.MPP の実装	149
9.2	scounteren レジスタの実装	149
9.3	sstatus レジスタの実装	151
9.4	トラップの委譲	152

9.4.1	トラップの委譲	152
9.4.2	トラップに関連するレジスタを作成する	153
9.4.3	stvec レジスタの実装	154
9.4.4	トラップで sepc、scause、stval レジスタを変更する	155
9.4.5	mstatus の SIE、SPIE、SPP ビットを実装する	155
9.4.6	SRET 命令を実装する	156
9.4.7	SEI、SSI、STI を実装する	158
9.4.8	割り込み条件、トラップの動作を変更する	161
9.5	ソフトウェア割り込みの実装 (SSWI)	162
第 10 章 S-mode の実装 (2. 仮想記憶システム)		164
10.1	概要	164
10.1.1	仮想記憶システム	164
10.1.2	ページング方式	164
10.1.3	satp レジスタ、アドレス変換プロセス	165
10.1.4	実装順序	166
10.2	メモリインターフェースの例外の実装	166
10.2.1	例外を伝達する	167
10.2.2	ページフォルトが発生した正確なアドレスを特定する	172
10.3	satp レジスタの作成	173
10.4	mstatus の MXR、SUM、MPRV ビットの作成	174
10.5	アドレス変換モジュール (PTW) の作成	175
10.5.1	CSR のインターフェースを実装する	175
10.5.2	Bare だけの ptw モジュールを作成する	176
10.5.3	ptw モジュールをインスタンス化する	179
10.6	Sv39 の実装	182
10.6.1	定数の定義	182
10.6.2	PTE の定義	183
10.6.3	ptw モジュールの実装	185
10.7	SFENCE.VMA 命令の実装	188
10.7.1	SFENCE.VMA 命令をデコードする	189
10.7.2	特権レベルの確認、mstatus.TVM を実装する	189
10.8	パイプラインをフラッシュする	190
10.8.1	CSR の変更	190
10.8.2	FENCE.I 命令の実装	191
第 11 章 PLIC の実装		192
第 12 章 Linux を動かす		193
あとがき		194

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、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 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) \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) \div rs2(符号付き) を求める
DIVU	rs1(符号無し) \div rs2(符号無し) を求める
REM	rs1(符号付き) $\% \div$ rs2(符号付き) を求める
REMU	rs1(符号無し) $\% \div$ 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1.2: 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。

^{*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。

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、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.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        : {}
        }
    }
}
}

```

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。

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 1.5）。

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

```

let exs_muldiv_valid : logic = exs_valid && exs_ctrl.is_muldiv && !exs_data_hazard && !exs_muldiv_is_requested;
var exs_muldiv_ready : logic;
var exs_muldiv_rvalid: logic;
var exs_muldiv_result: UIntX;

inst mdu: muldivunit (
    clk           ,
    rst           ,
    valid : exs_muldiv_valid ,
    ready : exs_muldiv_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 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_rvalidated: logic;
let exs_muldiv_stall  : logic = exs_ctrl.is_muldiv && !exs_muldiv_rvalid && !exs_muldiv_rva>rlid;
>rlid;

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

```

```

        } else {
            // muldivunitの処理が完了していたら1にする
            exs_muldiv_rvalided |= 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_rvalided` 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_rvalided`) 場合になります。

1.4 符号無しの乗算器の実装

1.4.1 mulunit モジュールを実装する

`WIDTH` ビットの符号無しの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: 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` クロックかけて行って求めていきます(図 1.1)。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 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.veryl)

```

// 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 ? !funct3[2] : !funct3_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。

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 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 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 `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 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 sext::<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 ? sext::<32, XLEN>(op1[31:0]) : op1), // MUL
    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 ? sext::<32, XLEN>(op2[31:0]) : op2), // MUL, MULH
    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_result;
    state = State::Finish;
    result = case funct3_saved[1:0] {
        2'b00 : if is_op32_saved ? sext::<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 符号無し除算の実装

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（リスト 1.29）。

▼リスト 1.29: 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;
        }
    }
  }
}

```

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 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 muldivunit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(muldivunit.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なります(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(muldivunit.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、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。

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モジュール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<XLLEN>,
) -> 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モジュール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_remainder;
    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: 除数、被除数を 32 ビットの値にする (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]) : va>
>value), // DIV, REM
        2'b01, 2'b11: if is_op32 ? {1'b0 repeat DIV_WIDTH - 32, value[31:0]} : value, // DI>
>VU, 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: 32 ビット演算のときの例外的な処理に対応する (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: 32ビット演算のとき、結果を符号拡張する(muldivunit.veril)

```
    } 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_rema
        >inder;
        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違法 (illegal)^{*2}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)^{*3}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。

^{*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} 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

^{*3} 割り込みは第 7 章「M-mode の実装 (2. 割り込みの実装)」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 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 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)

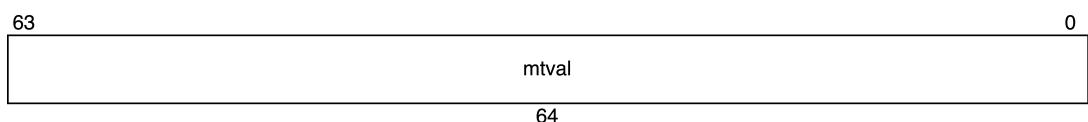
```
// CSRR(W|S|C)[I]命令かどうか
let is_wsc: logic = ctrl.is_csr && ctrl.funct3[1:0] != 0;
// ECALL命令かどうか
let is_ecall: logic = ctrl.is_csr && csr_addr == 0 && rs1[4:0] == 0 && ctrl.funct3 == 0 &&
rd_addr == 0;
```

▼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;
```

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 2.1: 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 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;
```

▼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 命令を判定して、tval に PC を設定します (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、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。

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。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 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'b00000001: f3 == 3'b000 || f3[2] == 1'b1, // RV64M
        7'b00000000: f3 == 3'b000 || f3 == 3'b001 || f3 == 3'b101, // ADDW, SLLW, SRLW
        7'b01000000: 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,
};

```

rv32i-tests でメモリ読み書きの順序を保証する FENCE 命令^{*4}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 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 に書き込みを行うとすると Illegal instruction 例外が発生します。

^{*4} 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 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⁵ (csrunit.veryl)

```
let wsource: UIntX = if ctrl.funct3[2] ? {1'b0 repeat XLEN - 5, rs1_addr} : rs1_data;
wdata  = case ctrl.funct3[1:0] {
  2'b01  : wsource,
  2'b10  : rdata | wsource,
  2'b11  : rdata & ~wsource,
  default: 'x,
} & 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⁶(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.funct3[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.funct3[1:0] == 2 || ctrl.funct3[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'b011; // 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
};
```

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

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。

⁶ ID ステージで判定することもできます。

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: トラン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: トランプが発生したかを WB ステージに伝える (core.veryl)

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

▼リスト 2.36: トラン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⁷。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 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,
}
```

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 は遷移先のアドレスになることに注意してください。

⁷ 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。

▼ 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'b00;
  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⁸。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)

```

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
}

```

⁸ 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。

```

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。

TODO 図

本章では CPU のメモリ部分を RAM (Random Access Memory)^{*1} 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) も追加します。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。TODO 図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。

^{*1} 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。

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。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 3.1）。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 3.9）。

▼ リスト 3.1: メモリマップの定義 (eei.veryl)

```
// メモリバスのデータ幅
const MEMBUS_DATA_WIDTH: u32 = 64;
// メモリのアドレス幅
const MEM_ADDR_WIDTH: u32 = 16;

// RAM
const RAM_ADDR_WIDTH: u32 = 16;
const RAM_DATA_WIDTH: u32 = 64;
const MMAP_RAM_BEGIN: Addr = 'h8000_0000 as Addr;

// ROM
const ROM_ADDR_WIDTH: u32 = 9;
const ROM_DATA_WIDTH: u32 = 64;
const MMAP_ROM_BEGIN: Addr = 'h1000 as Addr;
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 3.2 リスト 3.3）。

▼ リスト 3.2: 定数名を変更する (membus_if.veryl)

```
alias interface Membus = membus_if::<eei::MEMBUS_DATA_WIDTH, eei::XLEN>;
```

▼ リスト 3.3: Membus に置き換える (core.veryl)

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

▼ リスト 3.4: Membus に置き換える (memunit.veryl)

```
membus: modport Membus::master, // メモリとのinterface
```

▼ リスト 3.5: 定数名を変更する (memunit.veryl)

```

var req_wen  : logic          ;
var req_addr : Addr          ;
var req_wdata: logic<MEMBUS_DATA_WIDTH>    ;
var req_wmask: logic<MEMBUS_DATA_WIDTH / 8>;

const W    : u32          = XLEN;
let D    : logic<MEMBUS_DATA_WIDTH> = membus.rdata;
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 3.6）。

▼リスト 3.6: 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)

```

inst membus  : membus_if::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>;
inst i_membus: membus_if::<ILEN, XLEN>; // 命令フェッチ用
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 3.7、リスト 3.8、）。

▼リスト 3.7: addr_to_memaddr 関数をジェネリック関数に変更する (top.veryl)

```

// アドレスをデータ単位でのアドレスに変換する
function addr_to_memaddr::<DATA_WIDTH: u32, ADDR_WIDTH: u32> (
    addr: input logic<XLEN>,
) -> logic<ADDR_WIDTH> {
    return addr[$clog2(DATA_WIDTH / 8)+:ADDR_WIDTH];
}

```

▼リスト 3.8: ジェネリックパラメータを指定する (top.veryl)

```

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

```

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 3.9、リスト 3.10）。

▼リスト 3.9: パラメータ名を変更する (top.veryl)

```

module top #(
    param RAM_FILEPATH_IS_ENV: bit      = 1
)

```

```

param RAM_FILEPATH      : string = "RAM_FILE_PATH",
) (
  clk      : input clock,
  rst      : input reset,
  MMAP_DBG_ADDR: input Addr ,

```

▼リスト 3.10: パラメータ名を変更する (top.veryl)

```

inst ram: memory::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH> #(
  FILEPATH_IS_ENV: RAM_FILEPATH_IS_ENV,
  FILEPATH      : RAM_FILEPATH      ,
) (

```

シミュレータ用の C++ プログラムも変更します (リスト 3.11、リスト 3.12、リスト 3.13)。

▼リスト 3.11: 引数の名称を変える (tb_verilator.cpp)

```

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

```

▼リスト 3.12: 環境変数名を変える (tb_verilator.cpp)

```

// 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("RAM_FILE_PATH");
setenv("RAM_FILE_PATH", memory_file_path.c_str(), 1);

```

▼リスト 3.13: 環境変数名を変える (tb_verilator.cpp)

```

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

```

3.3 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 3.14)。

▼リスト 3.14: mmio_controller.veryl

```

import eei::*;

module mmio_controller (
  clk      : input  clock      ,
  rst      : input  reset      ,
  req_core: modport Membus::slave,
) {

```

```
enum Device {
    UNKNOWN,
}

inst req_saved: Membus;

var last_device : Device;
var is_requested: logic ;

// masterを0でリセットする
function reset_membus_master (
    master: modport Membus::master_output,
) {
    master.valid = 0;
    master.addr  = 0;
    master.wen   = 0;
    master.wdata = 0;
    master.wmask = 0;
}

// すべてのデバイスのmasterをリセットする
function reset_all_device_masters () {}

// アドレスからデバイスを取得する
function get_device (
    addr: input Addr,
) -> Device {
    return Device::UNKNOWN;
}

// デバイスのmasterにreqの情報を割り当てる
function assign_device_master (
    req: modport Membus::all_input,
) {}

// デバイスのrvalid、rdataをreqに割り当てる
function assign_device_slave (
    device: input Device ,
    req   : modport Membus::response,
) {
    req.rvalid = 1;
    req.rdata  = 0;
}

// デバイスのreadyを取得する
function get_device_ready (
    device: input Device,
) -> logic {
    return 1;
}

// デバイスのrvalidを取得する
function get_device_rvalid (
```

```

device: input Device,
) -> logic {
    return 1;
}

// req_coreの割り当て
always_comb {
    req_core.ready  = 0;
    req_core.rvalid = 0;
    req_core.rdata  = 0;

    if req_saved.valid {
        if is_requested {
            // 結果を返す
            assign_device_slave(last_device, req_core);
            req_core.ready      = get_device_rvalid(last_device);
        }
    } else {
        req_core.ready = 1;
    }
}

// デバイスのmasterの割り当て
always_comb {
    reset_all_device_masters();
    if req_saved.valid {
        if is_requested {
            if get_device_rvalid(last_device) {
                // 新しく要求を受け入れる
                if req_core.ready && req_core.valid {
                    assign_device_master(req_core);
                }
            }
        } else {
            // デバイスにreq_savedを割り当てる
            assign_device_master(req_saved);
        }
    } else {
        // 新しく要求を受け入れる
        if req_core.ready && req_core.valid {
            assign_device_master(req_core);
        }
    }
}

// 新しく要求を受け入れる
function accept_request () {
    req_saved.valid = req_core.ready && req_core.valid;
    if req_core.ready && req_core.valid {
        last_device  = get_device(req_core.addr);
        is_requested = get_device_ready(last_device);
        // reqを保存
        req_saved.addr = req_core.addr;
    }
}

```

```

        req_saved.wen    = req_core.wen;
        req_saved.wdata = req_core.wdata;
        req_saved.wmask = req_core.wmask;
    }
}

function on_clock () {
    if req_saved.valid {
        if is_requested {
            if get_device_rvalid(last_device) {
                accept_request();
            }
        } else {
            is_requested = get_device_ready(last_device);
        }
    } else {
        accept_request();
    }
}

function on_reset () {
    last_device        = Device::UNKNOWN;
    is_requested      = 0;
    reset_membus_master(req_saved);
}

always_ff {
    if_reset {
        on_reset();
    } else {
        on_clock();
    }
}
}

```

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 3.15）。

▼ リスト 3.15: modport 宣言を追加する (membus_if.veril)

```

modport all_input {
    ..input
}

modport response {
    rvalid: output,
    rdata : output,
}

modport slave_output {
    ready: output,
    ..same(response)
}

```

```
modport master_output {
    valid: output,
    addr : output,
    wen  : output,
    wdata: output,
    wmask: output,
}
```

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 3.16)。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 3.16: Device 型の定義 (mmio_controller.veryl)

```
enum Device {
    UNKNOWN,
}
```

`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。

3.4 RAM の接続

3.4.1 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 (@
<list>{mmio_controller.veryl.ram.Device}、<list>{mmio_controller.veryl.ram.get_device}

)。

▼ リスト 3.17: Device 型に RAM を追加する (mmio_controller.veryl)

```
enum Device {
    UNKNOWN,
    RAM,
}
```

▼ リスト 3.18: get_device 関数で RAM の範囲を定義する (mmio_controller.veryl)

```
function get_device (
    addr: input Addr,
) -> Device {
    if addr >= MMAP_RAM_BEGIN {
        return Device::RAM;
    }
    return Device::UNKNOWN;
}
```

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 3.19、リスト 3.20）。

▼ リスト 3.19: RAM とのインターフェースを追加する (mmio_controller.veryl)

```
module mmio_controller (
    clk      : input  clock      ,
    rst      : input  reset      ,
    req_core : modport Membus::slave ,
    ram_membus: modport Membus::master,
)
```

▼ リスト 3.20: インターフェースの要求部分をリセットする (mmio_controller.veryl)

```
function reset_all_device_masters () {
    reset_membus_master(ram_membus);
}
```

`ready`、`rvalid` を取得する関数に RAM を登録します（リスト 3.21、リスト 3.22）。

▼ リスト 3.21: インターフェースの `ready` を返す (mmio_controller.veryl)

```
function get_device_ready (
    device: input Device,
) -> logic {
    case device {
        Device::RAM: return ram_membus.ready;
        default   : {}
    }
    return 1;
}
```

▼リスト 3.22: インターフェースの rvalid を返す (mmio_controller.veryl)

```
function get_device_rvalid (
    device: input Device,
) -> logic {
    case device {
        Device::RAM: return ram_membus.rvalid;
        default : {}
    }
    return 1;
}
```

RAM の `rvalid`、`rdata` を `req_core` に割り当てます (リスト 3.23)。

▼リスト 3.23: RAM へのアクセス結果を `req` に割り当てる (mmio_controller.veryl)

```
function assign_device_slave (
    device: input Device ,
    req : modport Membus::response,
) {
    req.rvalid = 1;
    req.rdata = 0;
    case device {
        Device::RAM: req <> ram_membus;
        default : {}
    }
}
```

RAM のインターフェースに要求を割り当てます (リスト 3.24)。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 3.24: RAM に `req` を割り当ててアクセス要求する (mmio_controller.veryl)

```
function assign_device_master (
    req: modport Membus::all_input,
) {
    case get_device(req.addr) {
        Device::RAM: {
            ram_membus <> req;
            ram_membus.addr -= MMAP_RAM_BEGIN;
        }
        default: {}
    }
}
```

3.4.2 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 3.25、リスト 3.26）。

▼リスト 3.25: 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)

```
inst mmio_membus : Membus;
inst mmio_ram_membus: Membus;
inst ram_membus : membus_if::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>;
```

▼リスト 3.26: ポート名を変更する (top.veryl)

```
inst ram: memory::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH> #(
    FILEPATH_IS_ENV: RAM_FILEPATH_IS_ENV,
    FILEPATH : RAM_FILEPATH ,
) (
    clk ,
    rst ,
    membus: ram_membus,
);
```

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 3.27）。

▼リスト 3.27: 調停する対象を mmio_membus に変更する (top.veryl)

```
// mmio_controllerへの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 mmio_membus.ready {
            memarb_last_i      = !d_membus.valid;
            memarb_last_iaddr = i_membus.addr;
        }
    }
}

always_comb {
    i_membus.ready  = mmio_membus.ready && !d_membus.valid;
    i_membus.rvalid = mmio_membus.rvalid && memarb_last_i;
    i_membus.rdata  = if memarb_last_iaddr[2] == 0 ? mmio_membus.rdata[31:0] : mmio_membus.rdata[63:32];

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

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

```

    mmio_membus.addr  = i_membus.addr;
    mmio_membus.wen   = 0; // 命令フェッチは常に読み込み
    mmio_membus.wdata = 'x;
    mmio_membus.wmask = 'x;
}
}

```

mmio_controller をインスタンス化し、RAM と接続します。（リスト 3.28、リスト 3.29）。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 3.28: mmio_controller モジュールをインスタンス化する (top.veryl)

```

inst mmioc: mmio_controller (
    clk           ,
    rst           ,
    req_core : mmio_membus ,
    ram_membus: mmio_ram_membus,
);

```

▼リスト 3.29: mmio_controller モジュールと RAM を接続する (top.veryl)

```

always_comb {
    // mmio <> RAM
    ram_membus.valid      = mmio_ram_membus.valid;
    mmio_ram_membus.ready  = ram_membus.ready;
    ram_membus.addr        = addr_to_memaddr:<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>(mmio_ram_membus>us.addr);
    ram_membus.wen         = mmio_ram_membus.wen;
    ram_membus.wdata       = mmio_ram_membus.wdata;
    ram_membus.wmask       = mmio_ram_membus.wmask;
    mmio_ram_membus.rvalid = ram_membus.rvalid;
    mmio_ram_membus.rdata  = ram_membus.rdata;
}

```

3.4.3 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 3.30、リスト 3.31）。

▼リスト 3.30: PC の初期値を定義する (eei.veryl)

```

// pc on reset
const INITIAL_PC: Addr = MMAP_RAM_BEGIN;

```

▼リスト 3.31: PC の初期値を設定する (core.veryl)

```

always_ff {
    if_reset {
        if_pc      = INITIAL_PC;
        if_is_requested = 0;
        if_pc_requested = 0;
    }
}

```

```

        if_fifo_wvalid = 0;
        if_fifo_wdata  = 0;
    } 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 3.32）。

▼リスト 3.32: プログラムの先頭のアドレスを変更する (riscv-tests/env/p/link.ld)

```

OUTPUT_ARCH( "riscv" )
ENTRY(_start)

SECTIONS
{
    . = 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 3.33）。

▼リスト 3.33:

```

$ 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 3.34）。

▼リスト 3.34: .tohost のアドレスを変更する (top.veril)

```

#[ifdef(TEST_MODE)]
always_ff [
    let RISCVTESTS_TOHOST_ADDR: Addr = MMAP_RAM_BEGIN + '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();
    }
]

```

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。

3.5 ROM の実装

3.5.1 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 3.35、リスト 3.36）。

▼リスト 3.35: Device 型に ROM を変更する (mmio_controller.veryl)

```
enum Device {
    UNKNOWN,
    RAM,
    ROM,
}
```

▼リスト 3.36: get_device 関数で ROM の範囲を定義する (mmio_controller.veryl)

```
function get_device (
    addr: input Addr,
) -> Device {
    if MMAP_ROM_BEGIN <= addr && addr <= MMAP_ROM_END {
        return Device::ROM;
    }
    if addr >= MMAP_RAM_BEGIN {
        return Device::RAM;
    }
    return Device::UNKNOWN;
}
```

ROM とのインターフェースを追加します（リスト 3.37、リスト 3.38）。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 3.37: ROM とのインターフェースを追加する (mmio_controller.veryl)

```
module mmio_controller (
    clk      : input  clock      ,
    rst      : input  reset      ,
    req_core : modport Membus::slave ,
    ram_membus: modport Membus::master,
    rom_membus: modport Membus::master,
)
```

▼リスト 3.38: インターフェースの要求部分をリセットする (mmio_controller.veryl)

```
function reset_all_device_masters () {
    reset_membus_master(ram_membus);
    reset_membus_master(rom_membus);
}
```

ready 、 rvalid を取得する関数に ROM を登録します（リスト 3.39、リスト 3.40）。

▼リスト 3.39: インターフェースの ready を返す (mmio_controller.veryl)

```
case device {
    Device::RAM: return ram_membus.ready;
    Device::ROM: return rom_membus.ready;
    default : {}
}
```

▼リスト 3.40: インターフェースの rvalid を返す (mmio_controller.veryl)

```
case device {
    Device::RAM: return ram_membus.rvalid;
    Device::ROM: return rom_membus.rvalid;
    default : {}
}
```

ROM の `rvalid`、`rdata` を `req_core` に割り当てます (リスト 3.41)。

▼リスト 3.41: assign_device_slave 関数で ROM の結果を `req` に割り当てる (mmio_controller.veryl)

```
case device {
    Device::RAM: req <> ram_membus;
    Device::ROM: req <> rom_membus;
    default : {}
}
```

ROM のインターフェースに要求を割り当てます (リスト 3.42)。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 3.42: 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)

```
case get_device(req.addr) {
    Device::RAM: {
        ram_membus <> req;
        ram_membus.addr -= MMAP_RAM_BEGIN;
    }
    Device::ROM: {
        rom_membus <> req;
        rom_membus.addr -= MMAP_ROM_BEGIN;
    }
    default: {}
}
```

3.5.2 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 3.43)。

▼リスト 3.43: パラメータを定義する (top.veryl)

```
module top #(
    param RAM_FILEPATH_IS_ENV: bit      = 1      ,
    param RAM_FILEPATH      : string = "RAM_FILE_PATH",
    param ROM_FILEPATH_IS_ENV: bit      = 1      ,
    param ROM_FILEPATH      : string = "ROM_FILE_PATH"
)
```

```
param ROM_FILEPATH      : string = "ROM_FILE_PATH",
) (
```

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 3.44、リスト 3.45、リスト 3.46、リスト 3.47、リスト 3.48）。

▼ リスト 3.44: 引数の名称を変える (tb_verilator.cpp)

```
if (argc < 3) {
    std::cout << "Usage: " << argv[0] << " ROM_FILE_PATH RAM_FILE_PATH [CYCLE]" << std::endl;
    return 1;
}
```

▼ リスト 3.45: ROM の HEX ファイルのパスを生成する (tb_verilator.cpp)

```
// メモリの初期値を格納しているファイル名
std::string rom_file_path = argv[1];
std::string ram_file_path = argv[2];
try {
    // 絶対パスに変換する
    rom_file_path = fs::absolute(rom_file_path).string();
    ram_file_path = fs::absolute(ram_file_path).string();
} catch (const std::exception& e) {
    std::cerr << "Invalid memory file path : " << e.what() << std::endl;
    return 1;
}
```

▼ リスト 3.46: 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)

```
unsigned long long cycles = 0;
if (argc >= 4) {
    std::string cycles_string = argv[3];
    try {
        cycles = stoull(cycles_string);
    } catch (const std::exception& e) {
        std::cerr << "Invalid number: " << argv[3] << std::endl;
        return 1;
    }
}
```

▼ リスト 3.47: 環境変数を変更する (tb_verilator.cpp)

```
const char* original_env_rom = getenv("ROM_FILE_PATH");
const char* original_env_ram = getenv("RAM_FILE_PATH");
setenv("ROM_FILE_PATH", rom_file_path.c_str(), 1);
setenv("RAM_FILE_PATH", ram_file_path.c_str(), 1);
```

▼リスト 3.48: 環境変数を元に戻す (tb_verilator.cpp)

```
if (original_env_rom != nullptr){
    setenv("ROM_FILE_PATH", original_env_rom, 1);
}
if (original_env_ram != nullptr){
    setenv("RAM_FILE_PATH", original_env_ram, 1);
}
```

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 3.49、リスト 3.50、リスト 3.51)。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 3.49: 引数--rom を追加する (test/test.py)

```
parser.add_argument("--rom", default="bootrom.hex", help="hex file of rom")
```

▼リスト 3.50: シミュレータに ROM の HEX ファイルのパスを渡す (test/test.py)

```
def test(romhex, file_name):
    result_file_path = os.path.join(args.output_dir, file_name.replace(os.sep, "_") + ".txt")
    cmd = f"{args.sim_path} {romhex} {file_name} 0"
    success = False
```

▼リスト 3.51: test 関数に ROM の HEX ファイルのパスを渡す (test/test.py)

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

3.5.3 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 3.52)。

▼リスト 3.52: ROM のインターフェースの定義 (top.veryl)

```
inst mmio_membus : Membus;
inst mmio_ram_membus: Membus;
inst mmio_rom_membus: Membus;
inst ram_membus : membus_if::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>;
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 3.53)。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 3.53: ROM をインスタンス化する (top.veryl)

```
inst rom: memory::<ROM_DATA_WIDTH, ROM_ADDR_WIDTH> #(
    FILEPATH_IS_ENV: ROM_FILEPATH_IS_ENV,
```

```

    FILEPATH      : ROM_FILEPATH      ,
) (
    clk          ,
    rst          ,
    membus: rom_membus,
);

```

mmio_controller モジュールに `rom_membus` を接続します (リスト 3.54)。

▼ リスト 3.54: ROM のインターフェースを接続する (top.veryl)

```

inst mmioc: mmio_controller (
    clk          ,
    rst          ,
    req_core   : mmio_membus ,
    ram_membus: mmio_ram_membus,
    rom_membus: mmio_rom_membus,
);

```

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 3.55)。

▼ リスト 3.55: mmio_controller モジュールと ROM を接続する (top.veryl)

```

always_comb {
    // mmio <> ROM
    rom_membus.valid      = mmio_rom_membus.valid;
    mmio_rom_membus.ready = rom_membus.ready;
    rom_membus.addr       = addr_to_memaddr::<ROM_DATA_WIDTH, ROM_ADDR_WIDTH>(mmio_rom_membu
>us.addr);
    rom_membus.wen         = 0;
    rom_membus.wdata       = 0;
    rom_membus.wmask       = 0;
    mmio_rom_membus.rvalid = rom_membus.rvalid;
    mmio_rom_membus.rdata  = rom_membus.rdata;
}

```

3.5.4 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 3.56)。

▼ リスト 3.56: RAM の開始アドレスにジャンプするプログラム (bootrom.hex)

```
00409093080000b7 // 0: lui x1, 0x08000 4: slli x1, x1, 4
0000000000008067 // 8: jalr x0, 0(x1) c:
0000000000000000 // zero
```

PC の初期値を ROM のベースアドレスに変更します (リスト 3.57)。

▼ リスト 3.57: PC の初期値の変更 (eei.veryl)

```
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。

3.6**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。

3.6.1 デバイスのアドレスを設定する

リスト 3.9 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 3.58)。

▼ リスト 3.58: `DBG_ADDR` ポートに環境変数の値を設定する (`tb_verilator.cpp`)

```
// デバッグ用の入出力デバイスのアドレスを取得する
const char* dbg_addr_c = getenv("DBG_ADDR");
const unsigned long long DBG_ADDR = dbg_addr_c == nullptr ? 0 : std::strtoull(dbg_addr_c, nullptr, 0);

// top
```

```
Vcore_top *dut = new Vcore_top();
dut->MMAP_DBG_ADDR = DBG_ADDR;
```

3.6.2 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 3.59)。

▼ リスト 3.59: `Device` 型にデバッグ用の入出力デバイスを追加する (mmio_controller.veryl)

```
enum Device {
    UNKNOWN,
    RAM,
    ROM,
    DEBUG,
}
```

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 3.60、リスト 3.61)。

▼ リスト 3.60: `DBG_ADDR`、インターフェースを追加する (mmio_controller.veryl)

```
module mmio_controller (
    clk      : input  clock      ,
    rst      : input  reset      ,
    DBG_ADDR : input  Addr      ,
    req_core : modport Membus::slave ,
    ram_membus: modport Membus::master,
    rom_membus: modport Membus::master,
    dbg_membus: modport Membus::master,
) {
```

▼ リスト 3.61: インターフェースの要求部分をリセットする (mmio_controller.veryl)

```
function reset_all_device_masters () {
    reset_membus_master(ram_membus);
    reset_membus_master(rom_membus);
    reset_membus_master(dbg_membus);
}
```

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 3.62)。

▼ リスト 3.62: `get_device` 関数でデバイスの範囲を定義する (mmio_controller.veryl)

```
function get_device (
    addr: input Addr,
) -> Device {
    if DBG_ADDR <= addr && addr <= DBG_ADDR + 7 {
        return Device::DEBUG;
    }
    if MMAP_ROM_BEGIN <= addr && addr <= MMAP_ROM_END {
        return Device::ROM;
    }
}
```

```

    }
    if (addr >= MMAP_RAM_BEGIN) {
        return Device::RAM;
    }
    return Device::UNKNOWN;
}

```

インターフェースを設定します（リスト 3.63、リスト 3.64、リスト 3.65、リスト 3.66）。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 3.63: `assign_device_master` 関数の変更 (mmio_controller.veryl)

```

case get_device(req.addr) {
    Device::RAM: {
        ram_membus <=> req;
        ram_membus.addr -= MMAP_RAM_BEGIN;
    }
    Device::ROM: {
        rom_membus <=> req;
        rom_membus.addr -= MMAP_ROM_BEGIN;
    }
    Device::DEBUG: {
        dbg_membus <=> req;
        dbg_membus.addr -= DBG_ADDR;
    }
    default: {}
}

```

▼ リスト 3.64: `assign_device_slave` 関数の変更 (mmio_controller.veryl)

```

case device {
    Device::RAM : req <=> ram_membus;
    Device::ROM : req <=> rom_membus;
    Device::DEBUG: req <=> dbg_membus;
    default : {}
}

```

▼ リスト 3.65: `get_device_ready` 関数の変更 (mmio_controller.veryl)

```

case device {
    Device::RAM : return ram_membus.ready;
    Device::ROM : return rom_membus.ready;
    Device::DEBUG: return dbg_membus.ready;
    default : {}
}

```

▼ リスト 3.66: `get_device_rvalid` 関数の変更 (mmio_controller.veryl)

```

case device {
    Device::RAM : return ram_membus.rvalid;
    Device::ROM : return rom_membus.rvalid;
    Device::DEBUG: return dbg_membus.rvalid;
    default : {}
}

```

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 3.67、リスト 3.68)。

▼ リスト 3.67: インターフェースのインスタンス化 (top.veryl)

```
inst ram_membus      : membus_if::<RAM_DATA_WIDTH, RAM_ADDR_WIDTH>;
inst rom_membus      : membus_if::<ROM_DATA_WIDTH, ROM_ADDR_WIDTH>;
inst dbg_membus      : Membus;
```

▼ リスト 3.68: インターフェースを接続する (top.veryl)

```
inst mmioc: mmio_controller (
    clk           ,
    rst           ,
    DBG_ADDR    : MMAP_DBG_ADDR  ,
    req_core    : mmio_membus  ,
    ram_membus: mmio_ram_membus,
    rom_membus: mmio_rom_membus,
    dbg_membus      ,
);

```

3.6.3 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 3.69)。

▼ リスト 3.69: `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)

```
// デバッグ用のIO
always_ff {
    dbg_membus.ready  = 1;
    dbg_membus.rvalid = dbg_membus.valid;
    if dbg_membus.valid {
        if dbg_membus.wen {
            if dbg_membus.wdata[MEMBUS_DATA_WIDTH - 1:20] == 20'h01010 {
                $write("%c", dbg_membus.wdata[7:0]);
            } else if dbg_membus.wdata[lsb] == 1'b1 {
                #[ifdef(TEST_MODE)]
                {
                    test_success = dbg_membus.wdata == 1;
                }
                if dbg_membus.wdata == 1 {
                    $display("test success!");
                } else {
                    $display("test failed!");
                    $error ("wdata : %h", dbg_membus.wdata);
                }
                $finish();
            }
        }
    }
}
```

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。

3.6.4 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 3.70）。

▼リスト 3.70: デバッグ出力を `define` で囲う (core.veryl)

```
//////////////////////////// DEBUG //////////////////////////////
#[ifdef(PRINT_DEBUG)]
{
    var clock_count: u64;

    always_ff {
        if_reset {
            clock_count = 1;
        } else {
            clock_count = clock_count + 1;
        }
        $display("");
        $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 3.71）。これは `Hello,world!` と出力するプログラムです。

▼リスト 3.71: `Hello,world!`を出力するプログラム (test/debug_output.c)

```
#define DEBUG_REG ((volatile unsigned long long*)0x40000000)

void main(void) {
    int strlen = 13;
    unsigned char str[13];

    str[0] = 'H';
    str[1] = 'e';
    str[2] = 'l';
    str[3] = 'l';
    str[4] = 'o';
    str[5] = ',';
    str[6] = 'w';
    str[7] = 'o';
    str[8] = 'r';
    str[9] = 'l';
    str[10] = 'd';
    str[11] = '!';
    str[12] = '\n';

    for (int i = 0; i < strlen; i++) {
```

```

        unsigned long long c = str[1];
        *DEBUG_REG = c | (0x01010ULL << 44);
    }
    *DEBUG_REG = 1;
}

```

`DEBUG_OUTPUT` 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 3.72)。

▼ リスト 3.72: test/entry.S

```

.global _start
.section .text.init
_start:
    add x1, x0, x0
    la x2, _stack_bottom
    add x3, x0, x0
    add x4, x0, x0
    add x5, x0, x0
    add x6, x0, x0
    add x7, x0, x0
    add x8, x0, x0
    add x9, x0, x0
    add x10, x0, x0
    add x11, x0, x0
    add x12, x0, x0
    add x13, x0, x0
    add x14, x0, x0
    add x15, x0, x0
    add x16, x0, x0
    add x17, x0, x0
    add x18, x0, x0
    add x19, x0, x0
    add x20, x0, x0
    add x21, x0, x0
    add x22, x0, x0
    add x23, x0, x0
    add x24, x0, x0
    add x25, x0, x0
    add x26, x0, x0
    add x27, x0, x0
    add x28, x0, x0

```

```

add x29, x0, x0
add x30, x0, x0
add x31, x0, x0
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 3.73）。

▼リスト 3.73: test/link.ld

```

OUTPUT_ARCH( "riscv" )
ENTRY(_start)

SECTIONS
{
    . = 0x80000000;
    .text.init : { *(.text.init) }
    .text : { *(.text*) }
    .data : { *(.data*) }
    .bss : {*(.bss*)}
    .stack : {
        . = ALIGN(0x10);
        _stack_top = .;
        . += 4K;
        _stack_bottom = .;
    }
    _end = .;
}

```

`_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 3.72）、`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 3.74）。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 3.74: 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 -mcmode=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 3.75）。

▼リスト 3.75: 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。

3.6.5 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`^{*2}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 3.76、リスト 3.77）。

▼リスト 3.76: `elftools` の `import (test/test.py)`

```
from elftools.elf.elffile import ELFFile
```

▼リスト 3.77: 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`)

```
def is_elf(filepath):
    try:
        with open(filepath, 'rb') as f:
            magic_number = f.read(4)
            return magic_number == b'\x7fELF'
    except:
        return False

def get_section_address(filepath, section_name):
    try:
        with open(filepath, 'rb') as f:
            elffile = ELFFile(f)
            for section in elffile.iter_sections():
                if section.name == section_name:
                    return section.header['sh_addr']
    except:
        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 3.78）。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 3.78: オプションを追加する (`test/test.py`)

^{*2} pip でインストールできます

```
parser.add_argument("-e", "--extension", default=".bin.hex", help="hex file extension")
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 3.79）。

▼ リスト 3.79: dir_walk 関数で ELF ファイルを探す (test/test.py)

```
if entry.is_file():
    if not is_elf(entry.path):
        continue
    if len(args.files) == 0:
        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 3.80、リスト 3.81）。

▼ リスト 3.80: `DBG_ADDR` をシミュレータに渡す (test/test.py)

```
def test(dbg_addr, romhex, file_name):
    result_file_path = os.path.join(args.output_dir, file_name.replace(os.sep, "_") + ".txt")
    env = f"DBG_ADDR={dbg_addr} "
    cmd = f"{args.sim_path} {romhex} {file_name} 0"
    success = False
    with open(result_file_path, "w") as f:
        no = f.fileno()
        p = subprocess.Popen(" ".join([env, "exec", cmd]), shell=True, stdout=no, stderr=no)
```

▼ リスト 3.81: `DBG_ADDR` を `test` 関数に渡す (test/test.py)

```
for elfpath in dir_walk(args.dir):
    hexpath = elfpath + args.extension
    if not os.path.exists(hexpath):
        print("SKIP :", elfpath)
        continue
    dbg_addr = get_section_address(elfpath, args.debug_label)
    f, s = test(dbg_addr, os.path.abspath(args.rom), os.path.abspath(hexpath))
    res_strs.append(("PASS" if s else "FAIL") + " : " + f)
    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。

3.6.6 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 3.82）。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 3.82: 標準入力を 1 文字取得する関数の定義 (src/tb_verilator.cpp)

```
extern "C" const unsigned long long get_input_dpic() {
    unsigned char c = 0;
```

```

ssize_t bytes_read = read(STDIN_FILENO, &c, 1);

if (bytes_read == 1) {
    return static_cast<unsigned long long>(c) | (0x01010ULL << 44);
}
return 0;
}

```

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 3.83、リスト 3.84、リスト 3.85)。

▼ リスト 3.83: include を追加する (src/tb_verilator.cpp)

```

#include <fcntl.h>
#include <termios.h>
#include <signal.h>

```

▼ リスト 3.84: 設定を変更、復元する関数の定義 (src/tb_verilator.cpp)

```

struct termios old_setting;

void restore_termios_setting(void) {
    tcsetattr(STDIN_FILENO, TCSANOW, &old_setting);
}

void sighandler(int signum) {
    restore_termios_setting();
    exit(signum);
}

void set_nonblocking(void) {
    struct termios new_setting;

    if (tcgetattr(STDIN_FILENO, &old_setting) == -1) {
        perror("tcgetattr");
        return;
    }
    new_setting = old_setting;
    new_setting.c_lflag &= ~(ICANON | ECHO);
    if (tcsetattr(STDIN_FILENO, TCSANOW, &new_setting) == -1) {
        perror("tcsetattr");
        return;
    }
    signal(SIGINT, sighandler);
    signal(SIGTERM, sighandler);
    signal(SIGQUIT, sighandler);
    atexit(restore_termios_setting);

    int flags = fcntl(STDIN_FILENO, F_GETFL, 0);
    if (flags == -1) {

```

```

        perror("fcntl(F_GETFL)");
        return;
    }
    if (fcntl(STDIN_FILENO, F_SETFL, flags | O_NONBLOCK) == -1) {
        perror("fcntl(F_SETFL)");
        return;
    }
}

```

▼ リスト 3.85: 設定を変える関数を main 関数から呼び出す (src/tb_verilator.cpp)

```

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

    if (argc < 3) {
        std::cout << "Usage: " << argv[0] << " ROM_FILE_PATH RAM_FILE_PATH [CYCLE]" << std::endl;
        return 1;
    }

    #ifdef ENABLE_DEBUG_INPUT
        set_nonblocking();
    #endif
}

```

src/util.veryl に get_input_dpic 関数を呼び出す関数を実装します (リスト 3.86)。

▼ リスト 3.86: get_input 関数を定義する (src/util.veryl)

```

embed (inline) sv{{{
    package svutil;
    ...
    import "DPI-C" context function longint get_input_dpic();
    function longint get_input();
        return get_input_dpic();
    endfunction
    endpackage
}}}

package util {
    ...
    function get_input () -> u64 {
        return $sv::svutil::get_input();
    }
}

```

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 3.87)。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 3.87: 読み込みで get_input 関数を呼び出す (src/top.veryl)

```

always_ff {
    dbg_membus.ready  = 1;
    dbg_membus.rvalid = dbg_membus.valid;
}

```

```

if dbg_membus.valid {
    if dbg_membus.wen {
        ...
    } else {
        #[ifdef(ENABLE_DEBUG_INPUT)]
        {
            dbg_membus.rdata = util::get_input();
        }
    }
}
}

```

3.6.7 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 3.88）。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 3.88: `test/debug_input.c`

```

#define DEBUG_REG ((volatile unsigned long long*)0x40000000)

void main(void) {
    while (1) {
        unsigned long long c = *DEBUG_REG;
        if (c & (0x01010ULL << 44) == 0) {
            continue;
        }
        c = c & 255;
        *DEBUG_REG = (c + 1) | (0x01010ULL << 44);
    }
}

```

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（`term-run-change-input`）。

▼ リスト 3.89: 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 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单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、ストアを行う单一の命令です。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 Zalrsc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(リスト4.1)。

▼リスト4.1: LR、SC命令によるアトミックな加算

```
atomic_add:
    LR.W x2, (x3) ←アドレスx3の値をx2にロード
    ADDI x2, x2, 1 ←x2に1を足す
    SC.W x4, x2, (x3) ←ストアを試行し、結果をx4に格納
    BNEZ x4, atomic_add ←SC命令が失敗していたらやり直す
```

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。

4.1.4 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=0

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。

4.2 命令のデコード

TODO 命令の図

A拡張の命令はすべてR形式でのopcodeはOP-AMO(`7'b0101111`)です(TODO図)。それぞれの命令はfunct5とfunct3で区別できます(TODOテーブル)。

TODO funct5と命令の対応のテーブル

eeiパッケージにOP-AMOの定数を定義します(リスト4.2)。

▼リスト4.2: OP-AMOの定義(eei.veryl)

```
const OP_AMO      : logic<7> = 7'b0101111;
```

また、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4.3)。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。

▼リスト4.3: AMOOp型の定義(eei.veryl)

```
enum AMOOp: logic<5> {
    LR = 5'b00010,
    SC = 5'b00011,
    SWAP = 5'b00001,
    ADD = 5'b00000,
    XOR = 5'b00100,
    AND = 5'b01100,
    OR = 5'b01000,
    MIN = 5'b10000,
    MAX = 5'b10100,
    MINU = 5'b11000,
    MAXU = 5'b11100,
}
```

4.2.1 `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4.4)。

*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。

▼リスト4.4: InstCtrlにis_amoを定義する(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_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
    is_amo    : logic          , // AMO instruction
    funct3   : logic <3> , // 命令のfunct3フィールド
    funct7   : logic <7> , // 命令のfunct7フィールド
}
```

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4.5)。

▼リスト4.5: 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)

```
function inst_is_memop (
    ctrl: input InstCtrl,
) -> logic {
    return ctrl.itype == InstType::S || ctrl.is_load || ctrl.is_amo;
}
```

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4.6)。その他のopcodeのis_amoはFに設定してください。

▼リスト4.6: is_amoフラグを追加する(inst_decoder.veryl)

```
OP_SYSTEM: {
    InstType::I, T, F, F, F, F, F, F, T, F
},
OP_AMO: {
    InstType::R, T, F, F, F, F, F, F, F, T
},
default: {
    InstType::X, F, F, F, F, F, F, F, F, F
},
```

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4.7)。

▼リスト4.7: A拡張の命令のとき、validフラグを立てる(inst_decoder.veryl)

```
OP_MISC_MEM: T, // FENCE
OP_AMO      : f3 == 3'b010 || f3 == 3'b011, // AMO
default     : F,
```

4.2.2 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4.8)。

▼リスト4.8: メモリアドレスをrs1レジスタの値にする(core.veryl)

```
var memu_rdata: UIntX;
var memu_stall: logic;
let memu_addr : Addr = if mems_ctrl.is_amo ? memq_rdata.rs1_data : memq_rdata.alu_result;

inst memu: memunit (
    clk
    ,
    rst
    ,
    valid : mems_valid && !mems_expt.valid,
    is_new: mems_is_new
    ,
    ctrl : mems_ctrl
    ,
    addr : memu_addr
    ,
    rs2 : memq_rdata.rs2_data
    ,
    rdata : memu_rdata
    ,
    stall : memu_stall
    ,
    membus: d_membus
);

```

A拡張の命令で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4.9)。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4.9: 例外を判定するアドレスを変更する(core.veryl)

```
let memaddr : Addr = if exs_ctrl.is_amo ? exs_rs1_data : exs_a>
>lu_result;
let loadstore_address_misaligned : logic = inst_is_memop(exs_ctrl) && case exs_ctrl.fun>
>ct3[1:0] {
    2'b00 : 0, // B
    2'b01 : memaddr[0] != 1'b0, // H
    2'b10 : memaddr[1:0] != 2'b0, // W
    2'b11 : memaddr[2:0] != 3'b0, // D
    default: 0,
};

```

4.2.3 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4.10)。

▼リスト 4.10: メモリからロードした値をライトバックする (core.veryl)

```

let wbs_wb_data: UIntX = if wbs_ctrl.is_lui ?
    wbs_imm
: if wbs_ctrl.is_jump ?
    wbs_pc + 4
: if wbs_ctrl.is_load || wbs_ctrl.is_amo ?
    wbq_rdata.mem_rdata
: if wbs_ctrl.is_csr ?

```

4.3 amounit モジュールの作成

TODO 図

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 (TODO 図)。

4.3.1 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 4.11)。

▼リスト 4.11: core_data_if.veryl

```

import eei::*;

interface core_data_if {
    var valid : logic ;
    var ready : logic ;
    var addr : logic<XLEN> ;
    var wen : logic ;
    var wdata : logic<MEMBUS_DATA_WIDTH> ;
    var wmask : logic<MEMBUS_DATA_WIDTH / 8> ;
    var rvalid: logic ;
    var rdata : logic<MEMBUS_DATA_WIDTH> ;

    var is_amo: logic ;
    var aq : logic ;
    var rl : logic ;
    var amoop : AM0Op ;
    var funct3: logic<3>;

    modport master {
        valid : output,
        ready : input ,

```

```

    addr  : output,
    wen   : output,
    wdata : output,
    wmask : output,
    rvalid: input ,
    rdata : input ,
    is_amo: output,
    aq    : output,
    rl    : output,
    amoop : output,
    funct3: output,
}

modport slave {
    ..converse(master)
}

modport all_input {
    ..input
}
}
}

```

4.3.2 amountit 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/amountit.veryl` を作成し、次のように記述します（リスト 4.12）。

▼リスト 4.12: amountit.veryl

```

import eei::*;

module amountit (
    clk   : input  clock           ,
    rst   : input  reset           ,
    slave : modport core_data_if::slave,
    master: modport Membus::master   ,
) {

    enum State {
        Init,
        WaitReady,
        WaitValid,
    }

    var state      : State;
    inst slave_saved: core_data_if;

    // masterをリセットする
    function reset_master () {
        master.valid = 0;
        master.addr  = 0;
        master.wen   = 0;
    }
}

```

```

        master.wdata = 0;
        master.wmask = 0;
    }

    // masterに要求を割り当てる
    function assign_master (
        addr : input Addr
        ,
        wen : input logic
        ,
        wdata: input UIntX
        ,
        wmask: input logic<$size(UIntX) / 8>,
    ) {
        master.valid = 1;
        master.addr = addr;
        master.wen = wen;
        master.wdata = wdata;
        master.wmask = wmask;
    }

    // 新しく要求を受け入れる
    function accept_request_comb () {
        if slave.ready && slave.valid {
            assign_master(slave.addr, slave.wen, slave.wdata, slave.wmask);
        }
    }

    // slaveに結果を割り当てる
    always_comb {
        slave.ready = 0;
        slave.rvalid = 0;
        slave.rdata = 0;

        case state {
            State::Init: {
                slave.ready = 1;
            }
            State::WaitValid: {
                slave.ready = master.rvalid;
                slave.rvalid = master.rvalid;
                slave.rdata = master.rdata;
            }
            default: {}
        }
    }

    // masterに要求を割り当てる
    always_comb {
        reset_master();
        case state {
            State::Init : accept_request_comb();
            State::WaitReady: {
                assign_master(slave_saved.addr, slave_saved.wen, slave_saved.wdata, slave_saved>
                .wmask);
            }
        }
    }
}

```

```

        State::WaitValid: accept_request_comb();
        default          : {}
    }

}

// 新しく要求を受け入れる
function accept_request_ff () {
    slave_saved.valid = slave.ready && slave.valid;
    if slave.ready && slave.valid {
        slave_saved.addr  = slave.addr;
        slave_saved.wen   = slave.wen;
        slave_saved.wdata  = slave.wdata;
        slave_saved.wmask  = slave.wmask;
        slave_saved.is_amo = slave.is_amo;
        slave_saved.amoop = slave.amoop;
        slave_saved.aq    = slave.aq;
        slave_saved.rl    = slave.rl;
        slave_saved.funct3 = slave.funct3;
        state             = if master.ready ? State::WaitValid : State::WaitReady;
    } else {
        state = State::Init;
    }
}

function on_clock () {
    case state {
        State::Init      : accept_request_ff();
        State::WaitReady: if master.ready {
            state = State::WaitValid;
        }
        State::WaitValid: if master.rvalid {
            accept_request_ff();
        }
        default: {}
    }
}

function on_reset () {
    state      = State::Init;
    slave_saved.addr  = 0;
    slave_saved.wen   = 0;
    slave_saved.wdata  = 0;
    slave_saved.wmask  = 0;
    slave_saved.is_amo = 0;
    slave_saved.amoop = 0 as AM0Op;
    slave_saved.aq    = 0;
    slave_saved.rl    = 0;
    slave_saved.funct3 = 0;
}

always_ff {
    if_reset {
        on_reset();
    }
}

```

```

    } else {
        on_clock();
    }
}
}

```

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 4.13、リスト 4.14、リスト 4.15)。

▼リスト 4.13: `d_membus` の型を変更する (core.veryl)

```

i_membus: modport membus_if:<ILEN, XLEN>::master,
d_membus: modport core_data_if::master          ,
led      : output  UIntX                      ,

```

▼リスト 4.14: `core_data_if` インターフェースのインスタンス化 (top.veryl)

```
inst d_membus_core: core_data_if;
```

▼リスト 4.15: ポート名に割り当てるインターフェースを変更する (top.veryl)

```

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

```

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 4.16、リスト 4.17、リスト 4.19、リスト 4.18、リスト 4.20)。

▼リスト 4.16: `membus` の型を変更する (memunit.veryl)

```

stall : output logic          , // メモリアクセス命令が完了していない
membus: modport core_data_if::master, // メモリとのinterface
) {

```

▼リスト 4.17: 一時保存するレジスタの定義 (memunit.veryl)

```

var req_wen   : logic          ;
var req_addr  : Addr          ;
var req_wdata : logic<MEMBUS_DATA_WIDTH> ;
var req_wmask : logic<MEMBUS_DATA_WIDTH / 8>;
var req_is_amo: logic          ;
var req_amoop : AMOOp          ;
var req_aq   : logic          ;
var req_rl   : logic          ;
var req_funct3: logic<3>      ;

```

▼リスト 4.18: レジスタをリセットする (memunit.veryl)

```
always_ff {
    if_reset {
        state      = State::Init;
        req_wen    = 0;
        req_addr   = 0;
        req_wdata  = 0;
        req_wmask  = 0;
        req_is_amo = 0;
        req_amoop  = 0 as AM0Op;
        req_aq     = 0;
        req_rl     = 0;
        req_funct3 = 0;
    } else {
```

▼リスト 4.19: membus にレジスタの値を割り当てる (memunit.veryl)

```
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;
    membus.wmask  = req_wmask;
    membus.is_amo = req_is_amo;
    membus.amoop  = req_amoop;
    membus.aq    = req_aq;
    membus.rl    = req_rl;
    membus.funct3 = req_funct3;
```

▼リスト 4.20: 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)

```
case state {
    State::Init: if is_new & inst_is_memop(ctrl) {
        ...
        req_is_amo = ctrl.is_amo;
        req_amoop  = ctrl.funct7[6:2] as AM0Op;
        req_aq     = ctrl.funct7[1];
        req_rl     = ctrl.funct7[0];
        req_funct3 = ctrl.funct3;
    }
    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 4.21）。

▼リスト 4.21: amounit モジュールをインスタンス化する (top.veryl)

```
inst amou: amounit (
    clk           ,
    rst           ,
    slave : d_membus_core,
    master: d_membus ,
```

```
 );
```

4.4 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ます。本書では1つのアドレスのみ保持できるようにします。

4.4.1 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4.22、リスト4.23）。`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4.22: 予約セットの定義(amounit.veryl)

```
// lr/sc
var is_addr_reserved: logic;
var reserved_addr : Addr ;
```

▼リスト4.23: レジスタをリセットする(amounit.veryl)

```
is_addr_reserved = 0;
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4.24、リスト4.25、リスト4.26）。既に予約セットが使われている場合でもアドレスを上書きします。

▼リスト4.24: accept_request_comb関数の実装(amounit.veryl)

```
function accept_request_comb () {
    if slave.ready && slave.valid {
        if slave.is_amo {
            case slave.amoop {
                AMOp::LR: assign_master(slave.addr, 0, 0, 0);
                default : {}
            }
        } else {
            assign_master(slave.addr, slave.wen, slave.wdata, slave.wmask);
        }
    }
}
```

▼リスト 4.25: LR 命令のときに master にロード要求を割り当てる (amounit.veryl)

```

always_comb {
    reset_master();
    case state {
        State::Init : accept_request_comb();
        State::WaitReady: if slave_saved.is_amo {
            case slave_saved.amoop {
                AM0Op::LR: assign_master(slave_saved.addr, 0, 0, 0);
                default : {}
            }
        } else {
            assign_master(slave_saved.addr, slave_saved.wen, slave_saved.wdata, slave_saved.wmask);
        }
    }
}

```

▼リスト 4.26: LR 命令のときに予約セットを設定する (amounit.veryl)

```

function accept_request_ff () {
    slave_saved.valid = slave.ready && slave.valid;
    if slave.ready && slave.valid {
        slave_saved.addr = slave.addr;
        ...
        slave_saved.funct3 = slave.funct3;
        if slave.is_amo {
            case slave.amoop {
                AM0Op::LR: {
                    // reserve address
                    is_addr_reserved = 1;
                    reserved_addr = slave.addr;
                    state = if master.ready ? State::WaitValid : State::WaitReady;
                }
            }
        } else {
            state = if master.ready ? State::WaitValid : State::WaitReady;
        }
    }
}

```

4.4.2 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、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（リスト 4.27）。

▼リスト 4.27: SC 命令用の状態の定義 (amounit.veryl)

```

enum State {
    Init,
    WaitReady,
    WaitValid,
    SCSuccess,
}

```

```
    SCFail,
}
```

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 4.28）。`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 4.28: slave に SC 命令の結果を割り当てる (amounit.veryl)

```
State::SCSuccess: {
    slave.ready = master.rvalid;
    slave.rvalid = master.rvalid;
    slave.rdata = 0;
}
State::SCFail: {
    slave.ready = 1;
    slave.rvalid = 1;
    slave.rdata = 1;
}
```

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（リスト 4.29）。成功、失敗に関係なく、予約セットを空にします。

▼リスト 4.29: accept_request_ff 関数で予約セットを確認する (amounit.veryl)

```
AM0Op::SC: {
    // reset reserved
    let prev : logic = is_addr_reserved;
    is_addr_reserved = 0;
    // check
    if prev && slave.addr == reserved_addr {
        state = if master.ready ? State::SCSuccess : State::WaitReady;
    } else {
        state = State::SCFail;
    }
}
```

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（リスト 4.30）。

▼リスト 4.30: SC 命令の状態遷移 (amounit.veryl)

```
function on_clock () {
    case state {
        State::Init : accept_request_ff();
        State::WaitReady: if master.ready {
            if slave_saved.is_amo && slave_saved.amoop == AM0Op::SC {
                state = State::SCSuccess;
            } else {
                state = State::WaitValid;
            }
        }
    }
}
```

```

        }
    }
    State::WaitValid: if master.rvalid {
        accept_request_ff();
    }
    State::SCSuccess: if master.rvalid {
        accept_request_ff();
    }
    State::SCFail: accept_request_ff();
    default      : {}
}
}

```

SC 命令によるメモリへの書き込みを実装します（リスト 4.31、リスト 4.32）。

▼リスト 4.31: 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)

```

case slave.amoop {
    AM0Op::LR: assign_master(slave.addr, 0, 0, 0);
    AM0Op::SC: if is_addr_reserved && slave.addr == reserved_addr {
        @<b>      assign_master(slave.addr, 1, slave.wdata, slave.wmask); |</b>
        @<b> } |</b>
    default: {}
}

```

▼リスト 4.32: master に値を割り当てる (amounit.veryl)

```

always_comb {
    reset_master();
    case state {
        State::Init      : accept_request_comb();
        State::WaitReady: if slave_saved.is_amo {
            case slave_saved.amoop {
                AM0Op::LR: assign_master(slave_saved.addr, 0, 0, 0);
                AM0Op::SC: assign_master(slave_saved.addr, 1, slave_saved.wdata, slave_saved>
.d.wmask);
                default  : {}
            }
        } else {
            assign_master(slave_saved.addr, slave_saved.wen, slave_saved.wdata, slave_saved>
.wmask);
        }
        State::WaitValid           : accept_request_comb();
        State::SCFail, State::SCSuccess: accept_request_comb();
        default                  : {}
    }
}

```

4.5 Zaamo 拡張の実装

図 TODO

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 拡張の命令を図 TODO のような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 4.33）。

▼リスト 4.33: Zaamo 拡張の命令用の状態の定義 (amounit.veryl)

```
enum State {
    Init,
    WaitReady,
    WaitValid,
    SCSuccess,
    SCFail,
    AMOLoadReady,
    AMOLoadValid,
    AMOSToreReady,
    AMOSToreValid,
}
```

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 4.34、リスト 4.35）。`modport` に `import` 宣言を追加してください。

▼リスト 4.34: is_Zaamo 関数の定義 (core_data_if.veryl)

```
function is_Zaamo () -> logic {
    return is_amo && (amoop != AMOp::LR && amoop != AMOp::SC);
}
```

▼リスト 4.35: master に is_Zaamo 関数を import する (core_data_if.veryl)

```
amoop : output,
funct3 : output,
is_Zaamo: import,
}
```

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（リスト 4.36）。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 4.36: Zaamo 拡張の命令の計算を行う関数の定義 (amounit.veryl)

```
// AMO ALU
function calc_amo::<W: u32> (
    amoop: input AMOp ,
    wdata: input logic<W>,
    rdata: input logic<W>,
) -> logic<W> {
```

```

let lts: logic = $signed(wdata) <: $signed(rdata);
let ltu: logic = wdata <: rdata;

return case amoop {
    AM0Op::SWAP: wdata,
    AM0Op::ADD : rdata + wdata,
    AM0Op::XOR : rdata ^ wdata,
    AM0Op::AND : rdata & wdata,
    AM0Op::OR  : rdata | wdata,
    AM0Op::MIN : if lts ? wdata : rdata,
    AM0Op::MAX : if !lts ? wdata : rdata,
    AM0Op::MINU: if ltu ? wdata : rdata,
    AM0Op::MAXU: if !ltu ? wdata : rdata,
    default     : 0,
};

}

// Zaamo拡張の命令のwdataを生成する
function gen_amo_wdata (
    req  : modport core_data_if::all_input,
    rdata: input  UIntX
) -> UIntX {
    case req.funct3 {
        3'b010: { // word
            let low   : logic = req.addr[2] == 0;
            let rdata32: UInt32 = if low ? rdata[31:0] : rdata[63:32];
            let wdata32: UInt32 = if low ? req.wdata[31:0] : req.wdata[63:32];
            let result : UInt32 = calc_amo::<32>(req.amoop, wdata32, rdata32);
            return if low ? {rdata[63:32], result} : {result, rdata[31:0]};
        }
        3'b011 : return calc_amo::<64>(req.amoop, req.wdata, rdata); // double
        default: return 0;
    }
}

```

ロードした値を保存しておくレジスタを作成します（リスト 4.37、リスト 4.38）。

▼リスト 4.37: ロードしたデータを格納するレジスタの定義 (amounit.veryl)

```

// amo
var zaamo_fetched_data: UIntX;

```

▼リスト 4.38: レジスタのリセット (amounit.veryl)

```

reserved_addr      = 0;
zaamo_fetched_data = 0;
}

```

メモリアクセスが終了したら、ロードしておいた値を返します（リスト 4.39）。

▼リスト 4.39: 命令の結果を返す (amounit.veryl)

```
State::AMOStoreValid: {
    slave.ready = master.rvalid;
    slave.rvalid = master.rvalid;
    slave.rdata = zaamo_fetched_data;
}
```

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 4.40、リスト 4.41）。

▼ リスト 4.40: accept_request_comb 関数で、まずロード要求を行う (amountunit.veryl)

```
default: if slave.is_Zaamo() {
    assign_master(slave.addr, 0, 0, 0);
}
```

▼ リスト 4.41: 状態に基づいてロード、ストア要求を行う (amountunit.veryl)

```
State::AMOLoadReady : assign_master(slave_saved.addr, 0, 0, 0);
State::AMOLoadValid, State::AMOStoreReady: {
    let rdata : UIntX = if state == State::AMOLoadValid ? master.rdata : zaamo_fetch>
>ed_data;
    let wdata : UIntX = gen_amo_wdata(slave_saved, rdata);
    assign_master(slave_saved.addr, 1, wdata, slave_saved.wmask);
}
State::AMOStoreValid: accept_request_comb();
```

TODO 図に基づいて状態を遷移させます（リスト 4.42）。

▼ リスト 4.42: accept_request_ff 関数で、master の ready によって次の state を決める (amountunit.veryl)

```
default: if slave.is_Zaamo() {
    state = if master.ready ? State::AMOLoadValid : State::AMOLoadReady;
}
```

▼ リスト 4.43: Zaamo 拡張の命令の状態の遷移 (amountunit.veryl)

```
State::AMOLoadReady: if master.ready {
    state = State::AMOLoadValid;
}
State::AMOLoadValid: if master.rvalid {
    zaamo_fetched_data = master.rdata;
    state = if slave.ready ? State::AMOStoreValid : State::AMOStoreReady;
}
State::AMOStoreReady: if master.ready {
    state = State::AMOStoreValid;
}
State::AMOStoreValid: if master.rvalid {
    accept_request_ff();
}
```

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 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。

RVC 命令は表 TODO の 9 つのフォーマットが定義されています。

表 TODO

`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「5.6.2 32 ビット幅の命令に変換する」(p.99) を参照してください。

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 reserved と HINT についても書く表 TODO reserved と HINT についても書く

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。

¹ 1 Zc*拡張の一部の命令は複数の命令になります

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 (TODO 図)。

eei パッケージに定数 `IALIGN` を定義します (リスト 5.1)。

▼ リスト 5.1: IALIGN の定義 (eei.veryl)

```
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 5.2)。

▼ リスト 5.2: MEPC の書き込みマスクを変更する (eei.veryl)

```
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 5.3)。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。

▼ リスト 5.3: IALIGN が 16 のときに例外が発生しないようにする (core.veryl)

```
let instruction_address_misaligned: logic = IALIGN == 32 && memq_wdata.br_taken && memq_wdata.jump_addr[1:0] != 2'b00;
```

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 (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 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 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 5.4）。

▼リスト 5.4: `core_inst_if.veryl`

```
import eei::*;

interface core_inst_if {
    var rvalid  : logic;
    var rready  : logic;
    var raddr   : Addr ;
    var rdata   : Inst ;
    var is_hazard: logic;
    var next_pc : Addr ;

    modport master {
        rvalid  : input ,
        rready  : output,
        raddr   : input ,
        rdata   : input ,
        is_hazard: output, // control hazard
        next_pc : output, // actual next pc
    }

    modport slave {
        ..converse(master)
    }
}
```

`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。

5.4.2 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^{*2}。

core モジュールの `i_membus` の型を `core_inst_if::master` に変更します（リスト 5.5）。

▼リスト 5.5: `i_membus` の型を変更する (`core.veryl`)

```
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 5.6）。

^{*2} ここで削除するコードは次の「5.4.3 `inst_fetcher` モジュールを作成する」(p.89) 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。

▼リスト 5.6: IF ステージの変更 (core.veryl)

```
//////////////////////////// IF Stage //////////////////////////////

var control_hazard      : logic;
var control_hazard_pc_next: Addr ;

always_comb {
    i_membus.is_hazard = control_hazard;
    i_membus.next_pc = control_hazard_pc_next;
}
```

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コードになっています。 `if_fifo_type` 型、 `if_fifo_` から始まる変数は使わなくなるため削除してください。

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 5.7、リスト 5.8)。もともと `if_fifo` の `rvalid`、`rready`、`rdata` だった部分をインターフェースに変更しています。

▼リスト 5.7: ID ステージと `i_membus` を接続する (core.veryl)

```
let ids_valid      : logic      = i_membus.rvalid;
let ids_pc        : Addr        = i_membus.raddr;
let ids_inst_bits : Inst       = i_membus.rdata;
```

▼リスト 5.8: EX ステージに進められるときに `rready` を 1 にする (core.veryl)

```
always_comb {
    // ID -> EX
    i_membus.rready = exq_wready;
    exq_wvalid      = i_membus.rvalid;
    exq_wdata.addr  = i_membus.raddr;
    exq_wdata.bits  = i_membus.rdata;
    exq_wdata.ctrl  = ids_ctrl;
    exq_wdata.imm   = ids_imm;
```

5.4.3 `inst_fetcher` モジュールを作成する

core モジュールの 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。PC 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.veryl` を作成し、次のように

に記述します（リスト 5.9）。

▼リスト 5.9: ポートの定義 (inst_fetcher.veryl)

```
module inst_fetcher (
    clk      : input  clock      ,
    rst      : input  reset      ,
    core_if: modport core_inst_if::slave,
    mem_if : modport Membus::master      ,
) {
```

`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 との間の FIFO を作成します（リスト 5.10、リスト 5.11）。

▼リスト 5.10: fetch と issue を繋ぐ FIFO の作成 (inst_fetcher.veryl)

```
struct fetch_fifo_type {
    Addr: Addr      ,
    bits: logic<MEMBUS_DATA_WIDTH>,
}

var fetch_fifo_flush : logic      ;
var fetch_fifo_wvalid: logic      ;
var fetch_fifo_wready: logic      ;
var fetch_fifo_wdata : fetch_fifo_type;
var fetch_fifo_rdata : fetch_fifo_type;
var fetch_fifo_rready: logic      ;
var fetch_fifo_rvalid: logic      ;

inst fetch_fifo: fifo #(
    DATA_TYPE: fetch_fifo_type,
    WIDTH    : 3      ,
) (
    clk      ,
    rst      ,
    flush    : fetch_fifo_flush ,
    wready   : _      ,
    wready_two: fetch_fifo_wready,
    wvalid   : fetch_fifo_wvalid,
    wdata    : fetch_fifo_wdata ,
    rready   : fetch_fifo_rready,
    rvalid   : fetch_fifo_rvalid,
    rdata    : fetch_fifo_rdata ,
);
```

▼リスト 5.11: issue と core モジュールを繋ぐ FIFO の作成 (inst_fetcher.veryl)

```
struct issue_fifo_type {
    Addr: Addr,
    bits: Inst,
}
```

```

var issue_fifo_flush : logic          ;
var issue_fifo_wvalid: logic          ;
var issue_fifo_wready: logic          ;
var issue_fifo_wdata : issue_fifo_type;
var issue_fifo_rdata : issue_fifo_type;
var issue_fifo_rready: logic          ;
var issue_fifo_rvalid: logic          ;

inst issue_fifo: fifo #(
  DATA_TYPE: issue_fifo_type,
  WIDTH     : 3
) (
  clk          ,
  rst          ,
  flush        ,
  wready       ,
  wvalid       ,
  wdata        ,
  rready       ,
  rvalid       ,
  rdata        ,
);

```

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 5.12、リスト 5.13、リスト 5.14)。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 5.12: PC と状態管理用の変数の定義 (inst_fetcher.veryl)

```

var fetch_pc      : Addr ;
var fetch_requested : logic;
var fetch_pc_requested: Addr ;

```

▼ リスト 5.13: メモリへの要求の割り当て (inst_fetcher.veryl)

```

always_comb {
  mem_if.valid = 0;
  mem_if.addr  = 0;
  mem_if.wen   = 0;
  mem_if.wdata = 0;
  mem_if.wmask = 0;
  if !core_if.is_hazard {
    mem_if.valid = fetch_fifo_wready;
    if fetch_requested {
      mem_if.valid = mem_if.valid && mem_if.rvalid;
    }
    mem_if.addr = fetch_pc;
  }
}

```

▼リスト 5.14: PC、状態の更新 (inst_fetcher.veryl)

```

always_ff {
    if_reset {
        fetch_pc          = INITIAL_PC;
        fetch_requested   = 0;
        fetch_pc_requested = 0;
    } else {
        if core_if.is_hazard {
            fetch_pc          = {core_if.next_pc[XLEN - 1:3], 3'b0};
            fetch_requested   = 0;
            fetch_pc_requested = 0;
        } else {
            if fetch_requested {
                if mem_if.rvalid {
                    fetch_requested = mem_if.ready && mem_if.valid;
                    if mem_if.ready && mem_if.valid {
                        fetch_pc_requested = fetch_pc;
                        fetch_pc          += 8;
                    }
                }
            } else {
                if mem_if.ready && mem_if.valid {
                    fetch_requested   = 1;
                    fetch_pc_requested = fetch_pc;
                    fetch_pc          += 8;
                }
            }
        }
    }
}

```

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 5.15)。

▼リスト 5.15: ロードした 64 ビットの値を FIFO に格納する (inst_fetcher.veryl)

```

// memory -> fetch_fifo
always_comb {
    fetch_fifo_flush      = core_if.is_hazard;
    fetch_fifo_wvalid     = fetch_requested && mem_if.rvalid;
    fetch_fifo_wdata.addr = fetch_pc_requested;
    fetch_fifo_wdata.bits = mem_if.rdata;
}

```

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 5.16、リスト 5.17)。

▼リスト 5.16: オフセットの更新 (inst_fetcher.veryl)

```

var issue_pc_offset: logic<3>;
always_ff {
    if_reset {

```

```

        issue_pc_offset = 0;
    } else {
        if core_if.is_hazard {
            issue_pc_offset = core_if.next_pc[2:0];
        } else {
            if issue_fifo_wready && issue_fifo_wvalid {
                issue_pc_offset += 4;
            }
        }
    }
}

```

▼リスト 5.17: issue_fifo に 32 ビットずつ命令を格納する (inst_fetcher.veryl)

```

// fetch_fifo <-> issue_fifo
always_comb {
    let raddr : Addr          = fetch_fifo_rdata.addr;
    let rdata : logic<MEMBUS_DATA_WIDTH> = fetch_fifo_rdata.bits;
    let offset: logic<3>        = issue_pc_offset;

    fetch_fifo_rready = 0;
    issue_fifo_wvalid = 0;
    issue_fifo_wdata  = 0;

    if !core_if.is_hazard && fetch_fifo_rvalid {
        if issue_fifo_wready {
            fetch_fifo_rready = offset == 4;
            issue_fifo_wvalid = 1;
            issue_fifo_wdata.addr = {raddr[msb:3], offset};
            issue_fifo_wdata.bits = case offset {
                0      : rdata[31:0],
                4      : rdata[63:32],
                default: 0,
            };
        }
    }
}

```

core_if と FIFO を接続します (リスト 5.18)。

▼リスト 5.18: issue_fifo とインターフェースを接続する (inst_fetcher.veryl)

```

// issue_fifo <-> core
always_comb {
    issue_fifo_flush  = core_if.is_hazard;
    issue_fifo_rready = core_if.rready;
    core_if.rvalid    = issue_fifo_rvalid;
    core_if.raddr     = issue_fifo_rdata.addr;
    core_if.rdata     = issue_fifo_rdata.bits;
}

```

5.4.4 `inst_fetcher` モジュールと `core` モジュールを接続する

`core_inst_if` をインスタンス化します。(リスト 5.19)。

▼リスト 5.19: インターフェースの定義 (top.veryl)

```
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 5.20、リスト 5.21)。

▼リスト 5.20: `inst_fetcher` モジュールのインスタンス化 (top.veryl)

```
inst fetcher: inst_fetcher (
    clk
    ,
    rst
    ,
    core_if: i_membus_core,
    mem_if : i_membus
);
```

▼リスト 5.21: インターフェースを変更する (top.veryl)

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

`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るようになったので、`d_membus` との調停のところでアドレスを読んで 32 ビットずつ選択する処理が必要なくなりました。そのため、`rdata` をそのまま割り当てて、`memarb_last_iaddr` 変数とビット選択処理を削除します(リスト 5.22、リスト 5.23、リスト 5.24)。

▼リスト 5.22: 使用しない変数を削除する (top.veryl)

```
var memarb_last_i: logic;
var memarb_last_iaddr: Addr;
```

▼リスト 5.23: 使用しない変数を削除する (top.veryl)

```
always_ff {
    if_reset {
        memarb_last_i = 0;
        memarb_last_i = 0;
    } else {
        if mmio_membus.ready {
            memarb_last_i = !d_membus.valid;
            memarb_last_iaddr = i_membus.addr;
        }
    }
}
```

▼リスト 5.24: ビットの選択処理を削除する (top.veryl)

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

5.5**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 バイトの倍数な 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存するための変数を作成します (リスト 5.25)。

▼リスト 5.25: データを一時保存するための変数の定義 (inst_fetcher.veryl)

```
var issue_is_rdata_saved: logic      ;
var issue_saved_addr   : Addr        ;
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 5.26)。

▼リスト 5.26: 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)

```
always_ff {
    if_reset {
        issue_pc_offset      = 0;
        issue_is_rdata_saved = 0;
        issue_saved_addr    = 0;
        issue_saved_bits   = 0;
    } else {
        if core_if.is_hazard {
            issue_pc_offset      = core_if.next_pc[2:0];
            issue_is_rdata_saved = 0;
        } else {
            // offsetが6な32ビット命令の場合、
            // アドレスと上位16ビットを保存してFIFOを読み進める
            if issue_pc_offset == 6 && !issue_is_rdata_saved {
                if fetch_fifo_rvalid {
                    issue_is_rdata_saved = 1;
```

`issue_pc_offset` が 2、6 の場合の `issue_fifo` の書き込み処理を実装します (リスト 5.27)。6 の場合、保存していた 16 ビットと新しく読みだした 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 5.27: issue fifo に offset が 2, 6 の命令を格納する (inst fetcher.veryl)

```

if !core_if.is_hazard && fetch_fifo_rvalid {
    if issue_fifo_wready {
        if offset == 6 {
            // offsetが6な32ビット命令の場合、
            // 命令は{rdata_next[15:0], rdata[63:48]}になる
            if issue_is_rdata_saved {
                issue_fifo_wvalid      = 1;
                issue_fifo_wdata.addr = {issue_saved_addr[msb:3], offset};
                issue_fifo_wdata.bits = {rdata[15:0], issue_saved_bits};
            } else {
                // Read next 8 bytes
                fetch_fifo_rready = 1;
            }
        } else {
            fetch_fifo_rready      = offset == 4;
            issue_fifo_wvalid      = 1;
            issue_fifo_wdata.addr = {raddr[msb:3], offset};
            issue_fifo_wdata.bits = case offset {
                0      : rdata[31:0],
                2      : rdata[47:16],
                4      : rdata[63:32],
                default: 0,
            };
        };
    }
}

```

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。

5.6 RVC 命令の変換

5.6.1 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 5.28、リスト 5.29、リスト 5.30）。

▼リスト 5.28: is_rvc フラグの定義 (core_inst_if.veryl)

```
var rdata      : Inst ;
var is_rvc    : logic;
var is_hazard: logic;
```

▼リスト 5.29: modport に is_rvc を追加する (core_inst_if.veryl)

```
modport master {
    rvalid   : input ,
    rready   : output,
    raddr    : input ,
    rdata    : input ,
    is_rvc   : input ,
    is_hazard: output, // control hazard
    next_pc  : output, // actual next pc
}
```

▼リスト 5.30: InstCtrl 型に is_rvc フラグを追加する (corectrl.veryl)

```
is_amo   : logic      , // AMO instruction
is_rvc   : logic      , // RVC instruction
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` モジュールに供給するようにします（リスト 5.31、リスト 5.32、リスト 5.33）。

▼リスト 5.31: issue_fifo_type 型に is_rvc フラグを追加する (inst_fetcher.veryl)

```
struct issue_fifo_type {
    addr  : Addr ,
    bits  : Inst ,
    is_rvc: logic,
}
```

▼リスト 5.32: is_rvc フラグを 0 に設定する (inst_fetcher.veryl)

```
if offset == 6 {
    // offsetが6な32ビット命令の場合、
    // 命令は{rdata_next[15:0], rdata[63:48]}になる
    if issue_is_rdata_saved {
        issue_fifo_wvalid      = 1;
```

```

        issue_fifo_wdata.addr  = {issue_saved_addr[msb:3], offset};
        issue_fifo_wdata.bits  = {rdata[15:0], issue_saved_bits};
        issue_fifo_wdata.is_rvc = 0;
    } else {
        // Read next 8 bytes
        fetch_fifo_rready = 1;
    }
} else {
    fetch_fifo_rready      = offset == 4;
    issue_fifo_wvalid      = 1;
    issue_fifo_wdata.addr  = {raddr[msb:3], offset};
    issue_fifo_wdata.bits  = case offset {
        0      : rdata[31:0],
        2      : rdata[47:16],
        4      : rdata[63:32],
        default: 0,
    };
    issue_fifo_wdata.is_rvc = 0;
}

```

▼リスト 5.33: is_rvc フラグを接続する (inst_fetcher.veryl)

```

always_comb {
    issue_fifo_flush  = core_if.is_hazard;
    issue_fifo_rready = core_if.rready;
    core_if.rvalid    = issue_fifo_rvalid;
    core_if.raddr     = issue_fifo_rdata.addr;
    core_if.rdata     = issue_fifo_rdata.bits;
    core_if.is_rvc    = issue_fifo_rdata.is_rvc;
}

```

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 5.34、リスト 5.35、リスト 5.36）。また、C拡張が無効なのにRVC命令が供給されたら@<code> valid フラグを `0` に設定するようにします。

▼リスト 5.34: is_rvc フラグをポートに追加する (inst_decoder.veryl)

```

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

```

▼リスト 5.35: InstCtrl に is_rvc フラグを設定する (inst_decoder.veryl)

```

    default: {
        InstType::X, F, F, F, F, F, F, F, F, F
    },
    }, is_rvc, f3, f7
);

```

▼リスト 5.36: IALIGN が 32 ではないとき、不正な命令にする (inst_decoder.veryl)

```

OP_AMO      : f3 == 3'b010 || f3 == 3'b011, // AMO
default     : F,
} && (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` フラグを設定します (リスト 5.37)。

▼リスト 5.37: is_rvc フラグを inst_decoder に渡す (core.veryl)

```

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

```

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 5.38)。

▼リスト 5.38: 次の命令のアドレスを変える (core.veryl)

```

let wbs_wb_data: UIntX    = if wbs_ctrl.is_lui ?
    wbs_imm
: if wbs_ctrl.is_jump ?
    wbs_pc + (if wbs_ctrl.is_rvc ? 2 : 4)
: if wbs_ctrl.is_load || wbs_ctrl.is_amo ?

```

5.6.2 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 5.39)。

▼リスト 5.39: 命令のビット列を生成する関数を定義する (inst_gen_pkg.veryl)

```

import eei::*;

package inst_gen_pkg {
    function add (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
        return {7'b0000000, rs2, rs1, 3'b000, rd, OP_OP};
    }

    function addw (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
        return {7'b0000000, rs2, rs1, 3'b000, rd, OP_OP_32};
    }

    function addi (rd : input logic<5> , rs1: input logic<5> , imm: input logic<12>) -> Inst {
        return {imm, rs1, 3'b000, rd, OP_OP_IMM};
    }
}

```

```

function addiw (rd: input logic<5>, rs1: input logic<5>, imm: input logic<12>) -> Inst {
    return {imm, rs1, 3'b000, rd, OP_OP_IMM_32};
}

function sub (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
    return {7'b0100000, rs2, rs1, 3'b000, rd, OP_OP};
}

function subw (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
    return {7'b0100000, rs2, rs1, 3'b000, rd, OP_OP_32};
}

function inst_xor (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
    return {7'b0000000, rs2, rs1, 3'b100, rd, OP_OP};
}

function inst_or (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
    return {7'b0000000, rs2, rs1, 3'b110, rd, OP_OP};
}

function inst_and (rd: input logic<5>, rs1: input logic<5>, rs2: input logic<5>) -> Inst {
    return {7'b0000000, rs2, rs1, 3'b111, rd, OP_OP};
}

function andi (rd: input logic<5>, rs1: input logic<5>, imm: input logic<12>) -> Inst {
    return {imm, rs1, 3'b111, rd, OP_OP_IMM};
}

function slli (rd: input logic<5>, rs1: input logic<5>, shamt: input logic<6>) -> Inst {
    return {6'b000000, shamt, rs1, 3'b001, rd, OP_OP_IMM};
}

function srli (rd: input logic<5>, rs1: input logic<5>, shamt: input logic<6>) -> Inst {
    return {6'b000000, shamt, rs1, 3'b101, rd, OP_OP_IMM};
}

function srai (rd: input logic<5>, rs1: input logic<5>, shamt: input logic<6>) -> Inst {
    return {6'b010000, shamt, rs1, 3'b101, rd, OP_OP_IMM};
}

function lui (rd: input logic<5>, imm: input logic<20>) -> Inst {
    return {imm, rd, OP_LUI};
}

function load (rd: input logic<5>, rs1: input logic<5>, imm: input logic<12>, funct3: input logic<3>) -> Inst {
    return {imm, rs1, funct3, rd, OP_LOAD};
}

function store (rs1: input logic<5>, rs2: input logic<5>, imm: input logic<12>, funct3: input logic<3>) -> Inst {
    return {imm[11:5], rs2, rs1, funct3, imm[4:0], OP_STORE};
}

```

```

function jal (rd : input logic<5>, imm: input logic<20>) -> Inst {
    return {imm[19], imm[9:0], imm[10], imm[18:11], rd, OP_JAL};
}

function jalr (rd: input logic<5>, rs1: input logic<5>, imm: input logic<12>) -> Inst {
    return {imm, rs1, 3'b000, rd, OP_JALR};
}

function beq (rs1: input logic<5>, rs2: input logic<5>, imm: input logic<12>) -> Inst {
    return {imm[11], imm[9:4], rs2, rs1, 3'b000, imm[3:0], imm[10], OP_BRANCH};
}

function bne (rs1: input logic<5>, rs2: input logic<5>, imm: input logic<12>) -> Inst {
    return {imm[11], imm[9:4], rs2, rs1, 3'b001, imm[3:0], imm[10], OP_BRANCH};
}

function ebreak () -> Inst {
    return 32'h00100073;
}
}

```

rvc_converter モジュールのポートを定義します。 `src/rvc_converter.veryl` を作成し、次のように記述します（リスト 5.40）。

▼ リスト 5.40: ポートの定義 (rvc_converter.veryl)

```

import eei::*;
import inst_gen_pkg::*;

module rvc_converter (
    inst16: input logic<16>,
    is_rvc: output logic ,
    inst32: output Inst ,
) {

```

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 5.41）。 `rs1d` 、 `rs2d` の番号の範囲は `x8` から `x15` です。

▼ リスト 5.41: レジスタ番号の生成 (rvc_converter.veryl)

```

let rs1 : logic<5> = inst16[11:7];
let rs2 : logic<5> = inst16[6:2];
let rs1d: logic<5> = {2'b01, inst16[9:7]};
let rs2d: logic<5> = {2'b01, inst16[4:2]};

```

`inst16` から即値を生成します（リスト 5.42）。

▼リスト 5.42: 即値の生成 (rvc_converter.vmyl)

```

let imm_i      : logic<12> = {inst16[12] repeat 7, inst16[6:2]};
let imm_shamt: logic<6>  = {inst16[12], inst16[6:2]};
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]};
let imm_br    : logic<12> = {inst16[12] repeat 5, inst16[6:5], inst16[2], inst16[11:10], inst>
>16[4:3]};
let c0_mem_w : logic<12> = {5'b0, inst16[5], inst16[12:10], inst16[6], 2'b0}; // C.LW, C.SW
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 5.43)。`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`、ソースレジスタ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。

これにより、`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 5.43: RVC命令を 32ビット幅の命令に変換する (rvc_converter.vmyl)

```

always_comb {
    is_rvc = inst16[1:0] != 2'b11;
    inst32 = {16'b0, inst16};

    let funct3: logic<3> = inst16[15:13];
    case inst16[1:0] { // opcode
        2'b00: case funct3 { // C0
            3'b000: if inst16 != 0 { // C.ADDI4SPN
                let nzuimm: logic<10> = {inst16[10:7], inst16[12:11], inst16[5], inst16[6],>
> 2'b0};
                inst32 = addi(rs2d, 2, {2'b0, nzuimm});
            }
            3'b010: inst32 = load(rs2d, rs1d, c0_mem_w, 3'b010); // C.LW
            3'b011: if XLEN >= 64 { // C.LD
                inst32 = load(rs2d, rs1d, c0_mem_d, 3'b011);
            }
            3'b110: inst32 = store(rs1d, rs2d, c0_mem_w, 3'b010); // C.SW
            3'b111: if XLEN >= 64 { // C.SD
                inst32 = store(rs1d, rs2d, c0_mem_d, 3'b011);
            }
            default: {}
        }
        2'b01: case funct3 { // C1
            3'b000: inst32 = addi(rs1, rs1, imm_i); // C.ADDI
            3'b001: inst32 = if XLEN == 32 ? jal(1, imm_j) : addiw(rs1, rs1, imm_i); // C.JAL / C.ADDIW
            3'b010: inst32 = addi(rs1, 0, imm_i); // C.LI
            3'b011: if rs1 == 2 { // C.ADDI16SP
                let imm : logic<10> = {inst16[12], inst16[4:3], inst16[5], inst16[2], in>

```

```

>st16[6], 4'b0};

    inst32 = addi(2, 2, {imm[msb] repeat 2, imm});
} else { // C.LUI
    inst32 = lui(rs1, {imm_i[msb] repeat 8, imm_i});
}
3'b100: case inst16[11:10] { // funct2 or funct6[1:0]
    2'b00: if !(XLEN == 32 && imm_shamt[msb] == 1) {
        inst32 = srli(rs1d, rs1d, imm_shamt); // C.SRLI
    }
    2'b01: if !(XLEN == 32 && imm_shamt[msb] == 1) {
        inst32 = srai(rs1d, rs1d, imm_shamt); // C.SRAI
    }
    2'b10: inst32 = andi(rs1d, rs1d, imm_i); // C.ANDI
    2'b11: if inst16[12] == 0 {
        case inst16[6:5] {
            2'b00 : inst32 = sub(rs1d, rs1d, rs2d); // C.SUB
            2'b01 : inst32 = inst_xor(rs1d, rs1d, rs2d); // C.XOR
            2'b10 : inst32 = inst_or(rs1d, rs1d, rs2d); // C.OR
            2'b11 : inst32 = inst_and(rs1d, rs1d, rs2d); // C.AND
            default: {}
        }
    } else {
        if XLEN >= 64 {
            if inst16[6:5] == 2'b00 {
                inst32 = subw(rs1d, rs1d, rs2d); // C.SUBW
            } else if inst16[6:5] == 2'b01 {
                inst32 = addw(rs1d, rs1d, rs2d); // C.ADDW
            }
        }
    }
    default: {}
}
3'b101 : inst32 = jal(0, imm_j); // C.J
3'b110 : inst32 = beq(rs1d, 0, imm_br); // C.BEQZ
3'b111 : inst32 = bne(rs1d, 0, imm_br); // C.BNEZ
default: {}

}
2'b10: case funct3 { // C2
    3'b000: if !(XLEN == 32 && imm_shamt[msb] == 1) {
        inst32 = slli(rs1, rs1, imm_shamt); // C.SLLI
    }
    3'b010: if rs1 != 0 { // C.LWSP
        let offset: logic<8> = {inst16[3:2], inst16[12], inst16[6:4], 2'b0};
        inst32 = load(rs1, 2, {4'b0, offset}, 3'b010);
    }
    3'b011: if XLEN >= 64 && rs1 != 0 { // C.LDSP
        let offset: logic<9> = {inst16[4:2], inst16[12], inst16[6:5], 3'b0};
        inst32 = load(rs1, 2, {3'b0, offset}, 3'b011);
    }
    3'b100: if inst16[12] == 0 {
        inst32 = if rs2 == 0 ? jalr(0, rs1, 0) : addi(rs1, rs2, 0); // C.JR / C.M
    }
}

```

```

>LR
        if rs2 == 0 {
            inst32 = if rs1 == 0 ? ebreak() : jalr(1, rs1, 0); // C.EBREAK : C.JA
        } else {
            inst32 = add(rs1, rs1, rs2); // C.ADD
        }
    }
    3'b110: { // C.SWSP
        let offset: logic<8> = {inst16[8:7], inst16[12:9], 2'b0};
        inst32 = store(2, rs2, {4'b0, offset}, 3'b010);
    }
    3'b111: if XLEN >= 64 { // C.SDSP
        let offset: logic<9> = {inst16[9:7], inst16[12:10], 3'b0};
        inst32 = store(2, rs2, {3'b0, offset}, 3'b011);
    }
    default: {}
}
default: {}
}

```

5.6.3 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 5.44）。

▼リスト 5.44: ryc_converter モジュールのインスタンス化 (inst_fetcher.vryl)

```

// instruction converter
var rvcc_inst16: logic<16>;
var rvcc_is_rvc: logic      ;
var rvcc_inst32: Inst       ;

inst rvcc: rvc_converter (
    inst16: case issue_pc_offset {
        0      : fetch_fifo_rdata.bits[15:0],
        2      : fetch_fifo_rdata.bits[31:16],
        4      : fetch_fifo_rdata.bits[47:32],
        6      : fetch_fifo_rdata.bits[63:48],
        default: 0,
    },
    is_rvc: rvcc_is_rvc,
    inst32: rvcc_inst32,
);

```

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 5.45、リスト 5.46)。

▼リスト 5.45: RVC 命令のときのオフセットの更新 (inst_fetcher.veryl)

```

// offsetが6な32ビット命令の場合、
// アドレスと上位16ビットを保存してFIFOを読み進める
if issue_pc_offset == 6 && !rvcc_is_rvc && !issue_is_rdata_saved {
    if fetch_fifo_rvalid {
        issue_is_rdata_saved = 1;
        issue_saved_addr     = fetch_fifo_rdata.addr;
        issue_saved_bits     = fetch_fifo_rdata.bits[63:48];
    }
} else {
    if issue_fifo_wready && issue_fifo_wvalid {
        issue_pc_offset      += if issue_is_rdata_saved || !rvcc_is_rvc ? 4 : 2;
        issue_is_rdata_saved = 0;
    }
}

```

▼リスト 5.46: RVC 命令のときの issue_fifo への書き込み (inst_fetcher.veryl)

```

if !core_if.is_hazard && fetch_fifo_rvalid {
    if issue_fifo_wready {
        if offset == 6 {
            // offsetが6な32ビット命令の場合、
            // 命令は{rdata_next[15:0], rdata[63:48]}になる
            if issue_is_rdata_saved {
                issue_fifo_wvalid      = 1;
                issue_fifo_wdata.addr  = {issue_saved_addr[msb:3], offset};
                issue_fifo_wdata.bits  = {rdata[15:0], issue_saved_bits};
                issue_fifo_wdata.is_rvc = 0;
            } else {
                fetch_fifo_rready = 1;
                if rvcc_is_rvc {
                    issue_fifo_wvalid      = 1;
                    issue_fifo_wdata.addr  = {raddr[msb:3], offset};
                    issue_fifo_wdata.is_rvc = 1;
                    issue_fifo_wdata.bits  = rvcc_inst32;
                } else {
                    // Read next 8 bytes
                }
            }
        } else {
            fetch_fifo_rready      = !rvcc_is_rvc && offset == 4;
            issue_fifo_wvalid      = 1;
            issue_fifo_wdata.addr  = {raddr[msb:3], offset};
            if rvcc_is_rvc {
                issue_fifo_wdata.bits = rvcc_inst32;
            } else {
                issue_fifo_wdata.bits = case offset {
                    0      : rdata[31:0],
                    2      : rdata[47:16],
                    4      : rdata[63:32],
                    default: 0,
                };
            }
        }
    }
}

```

```
        issue_fifo_wdata.is_rvc = rvcc_is_rvc;  
    }  
}  
}
```

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 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、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、次のような機能を実装します。

- 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 を動かすための十分な機能を持っています。第 III 部の最後では Linux を動作させます。

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、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

^{*1} 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。

トの数値で表すことができます。

TODO table 高い低い

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 6.1)。

▼ リスト 6.1: (eei.veryl)

```
enum PrivMode: logic<2> {
    M = 2'b11,
    S = 2'b01,
    U = 2'b00,
}
```

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 だけが実装されているように扱います。

TODO M, MU, MSU テーブルと章

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 6.2)。

▼ リスト 6.2: (csrunit.veryl)

```
var mode: PrivMode;
```

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

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 6.3)。

▼ リスト 6.3: (eei.veryl)

```
enum CsrAddr: logic<12> {
    // Machine Information Registers
    MIMPID = 12'hf13,
    MHARTID = 12'hf14,
    // Machine Trap Setup
    MSTATUS = 12'h300,
    MISA = 12'h301,
    MEDELEG = 12'h302,
    MIDELEG = 12'h303,
    MIE = 12'h304,
    MTVEC = 12'h305,
    MCOUNTEREN = 12'h306,
    // Machine Trap Handling
    MSCRATCH = 12'h340,
    MEPC = 12'h341,
    MCAUSE = 12'h342,
```

```

    MVAL = 12'h343,
    MIP = 12'h344,
    // Machine Counter/Timers
    MCYCLE = 12'hB00,
    MINSTRET = 12'hB02,
    // Custom
    LED = 12'h800,
}

```

6.1.3 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、 $MXLEN \geq SXLEN \geq UXLEN$ を満たす必要があります。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。

6.2 misa レジスタ (Machine ISA)

63	62 61	26 25	0
MXL	0	Extensions	

▲図 6.1: 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 (table TODO) 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 6.4、リスト 6.5)。CPU 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 にしています。

▼リスト 6.4: (csrununit.veryl)

```

let misa : UIntX = {2'd2, 1'b0 repeat XLEN - 28, 26'b000000000000100010000101}; // M, I,>
> C, A

```

▼リスト 6.5: (csrunit.veryl)

```
rdata = case csr_addr {
    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。

6.3 mimpid レジスタ (Machine Implementation ID)

63

0

Implementation

64

▲図 6.2: 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^{*2}、marchid^{*3}) 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 6.6、リスト 6.7)。

▼リスト 6.6: (eei.veryl)

```
// Machine Implementation ID
const MACHINE_IMPLEMENTATION_ID: UIntX = 1;
```

▼リスト 6.7: (csrunit.veryl)

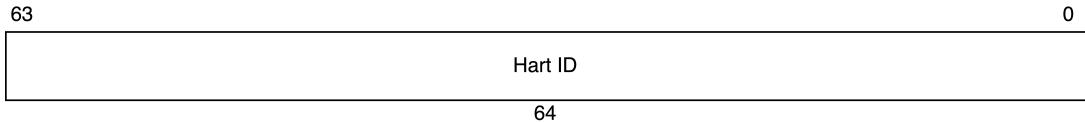
```
rdata = case csr_addr {
    CsrAddr::MISA : misa,
    CsrAddr::MIMPID: MACHINE_IMPLEMENTATION_ID,
```

6.4 mhartid レジスタ (Hart ID)

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できます。ID はどんな値でも良いですが、ID が `0` のハードウェア

^{*2} 製造業者の ID(JEDEC ID) を格納します

^{*3} マイクロアーキテクチャの種類を示す ID を格納します



▲図 6.3: mhartid レジスタ

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 変数を作成し、読み込めるようにします（リスト 6.8、リスト 6.9）。

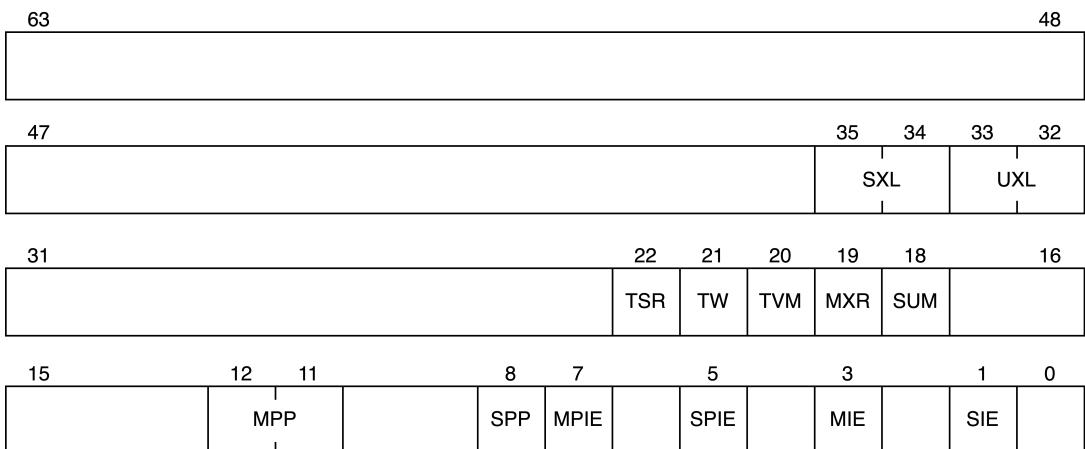
▼リスト 6.8: (csrunit.veryl)

```
let mhartid: UIntX = 0;
```

▼リスト 6.9: (csrunit.veryl)

```
rdata = case csr_addr {
    CsrAddr::MISA    : misa,
    CsrAddr::MIMPID : MACHINE_IMPLEMENTATION_ID,
    CsrAddr::MHARTID: mhartid,
```

6.5 mstatus レジスタ (Machine Status)



▲図 6.4: mstatus レジスタ

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 6.4 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ておきます（リスト 6.10、リスト 6.11、リスト 6.12、リスト 6.13、リスト 6.14、リスト 6.15）。

▼リスト 6.10: (csrunit.veryl)

```
const MSTATUS_WMASK: UIntX = 'h0000_0000_0000_0000 as UIntX;
```

▼リスト 6.11: (csrunit.veryl)

```
wmask = case csr_addr {
    CsrAddr::MSTATUS: MSTATUS_WMASK,
```

▼リスト 6.12: (csrunit.veryl)

```
var mstatus: UIntX;
```

▼リスト 6.13: (csrunit.veryl)

```
rdata = case csr_addr {
    CsrAddr::MISA    : misa,
    CsrAddr::MIMPID : MACHINE_IMPLEMENTATION_ID,
    CsrAddr::MHARTID: mhartid,
    CsrAddr::MSTATUS: mstatus,
```

▼リスト 6.14: (csrunit.veryl)

```
always_ff {
    if_reset {
        mode    = PrivMode::M;
        mstatus = 0;
```

▼リスト 6.15: (csrunit.veryl)

```
if is_wsc {
    case csr_addr {
        CsrAddr::MSTATUS: mstatus = wdata;
        CsrAddr::MTVEC   : mtvec   = wdata;
```

6.6

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。

6.6.1 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 6.16、リスト 6.17)。

▼ リスト 6.16: (csrunit.veryl)

```
var mcycle : UInt64;
```

▼ リスト 6.17: (csrunit.veryl)

```
var mcycle : UInt64;
```

always_ff ブロックで、クロックごとに値を更新します(リスト 6.18)。

▼ リスト 6.18: (csrunit.veryl)

```
always_ff {
    if_reset {
        mode      = PrivMode::M;
        mstatus   = 0;
        mtvec    = 0;
        mcycle  = 0;
        mepc     = 0;
        mcause   = 0;
        mtval    = 0;
        led      = 0;
    } else {
        mcycle += 1;
    }
}
```

6.6.2 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 6.19、リスト 6.20)。

▼ リスト 6.19: (core.veryl)

```
var minstret      : UInt64;
```

▼ リスト 6.20: (core.veryl)

```
always_ff {
    if_reset {
        minstret = 0;
    } else {
        if wbq_rvalid && wbq_rready && !wbq_rdata.raise_trap {
            minstret += 1;
        }
    }
}
```

```
}
```

`minstret` の値を `csrunit` モジュールに渡し、読み込めるようにします（リスト 6.21、リスト 6.22、リスト 6.23）。

▼リスト 6.21: (core.veryl)

```
minstret ,
```

▼リスト 6.22: (csrunit.veryl)

```
minstret : input UInt64 ,
```

▼リスト 6.23: (csrunit.veryl)

```
CsrAddr::MCYCLE : mcycle,  
CsrAddr::MINSTRET: minstret,  
CsrAddr::MEPC : mepc,
```

`csrunit` モジュールは MRET 命令でも `raise_trap` フラグを立てているため、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 6.24、リスト 6.25、リスト 6.26、リスト 6.27）。

▼リスト 6.24: (csrunit.veryl)

```
trap_return: output logic ,
```

▼リスト 6.25: (csrunit.veryl)

```
// Trap Return  
assign trap_return = valid && is_mret && !raise_expt;  
  
// Trap  
assign raise_trap = raise_expt || trap_return;
```

▼リスト 6.26: (core.veryl)

```
trap_return: csru_trap_return ,
```

▼リスト 6.27: (core.veryl)

```
wbq_wdata.raise_trap = csru_raise_trap && !csru_trap_return;
```

6.7

mscratch レジスタ (Machine Scratch)

`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^{*4}()。

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の情報) 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 6.28、リスト 6.29、リスト 6.30、リスト 6.31、リスト 6.32、リスト 6.33)。

▼ リスト 6.28: (csrunit.veryl)

```
var mcycle : UInt64;
var mscratch: UIntX ;
var mepc : UIntX ;

//list[csrunit.veryl.mscratch.rdata][ (csrunit.veryl)]{
  CsrAddr::MINSTRET: minstret,
  CsrAddr::MSCRATCH: mscratch,
  CsrAddr::MEPC : mepc,
```

▼ リスト 6.30: (csrunit.veryl)

```
const MTVEC_WMASK : UIntX = 'hffff_ffff_ffff_fffc;
const MSCRATCH_WMASK: UIntX = 'hffff_ffff_ffff_ffff;
const MEPC_WMASK : UIntX = 'hffff_ffff_ffff_fffe;
```

▼ リスト 6.31: (csrunit.veryl)

```
CsrAddr::MTVEC : MTVEC_WMASK,
CsrAddr::MSCRATCH: MSCRATCH_WMASK,
CsrAddr::MEPC : MEPC_WMASK,
```

▼ リスト 6.32: (csrunit.veryl)

```
mtvec = 0;
mscratch = 0;
mcycle = 0;
```

^{*4} 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 6.33: (csrunit.veryl)

```
CsrAddr::MTVEC    : mtvec    = wdata;  
CsrAddr::MSCRATCH: mscratch = wdata;  
CsrAddr::MEPC     : mepc     = wdata;
```

第 7 章

M-mode の実装 (2. 割り込みの実装)

7.1 概要

7.1.1 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。

TODO 図

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秒くらいかかる処理 A を繰り返すとして、繰り返しごとに入力の有無を確認する場合、最大 1 秒の遅延が発生します (TODO 図)。待ち時間減らすために処理 A を分割すると遅延は減少しますが、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。

TODO 図

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て通知内容を処理することです。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 (TODO 図)。

7.1.2 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11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「8.9 割り込み条件の変更」(p.147)、「9.4 トランプの委譲」(p.152)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の発生を待っている(待機)状態を示す 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管理する制御する 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 レジスタ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、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。

7.1.3 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テーブル TODO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。

TODO テーブル

7.1.4 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 7.1)。

▼ リスト 7.1: (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,
    SUPERVISOR_SOFTWARE_INTERRUPT = 'h8000_0000_0000_0001,
    MACHINE_SOFTWARE_INTERRUPT = 'h8000_0000_0000_0003,
    SUPERVISOR_TIMER_INTERRUPT = 'h8000_0000_0000_0005,
    MACHINE_TIMER_INTERRUPT = 'h8000_0000_0000_0007,
    SUPERVISOR_EXTERNAL_INTERRUPT = 'h8000_0000_0000_0009,
    MACHINE_EXTERNAL_INTERRUPT = 'h8000_0000_0000_000b,
}
```

7.1.5 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。それぞれタイマ割り込み、ソフトウェア割り込み、ソフトウェア割り込み向けのデバイスで、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 7.1 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、「9.5 ソフトウェア割り込みの実装 (SSWI)」(p.162) 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 7.2)。

▼ リスト 7.2: (eei.veryl)

```
// ACLINT
const MMAP_ACLINT_BEGIN : Addr = 'h200_0000 as Addr;
const MMAP_ACLINT_MSIP : Addr = 0;
const MMAP_ACLINT_MTIMECMP: Addr = 'h4000 as Addr;
const MMAP_ACLINT_MTIME : Addr = 'h7ff8 as Addr;
```

```
const MMAP_ACLINT_SETSSIP : Addr = 'h8000 as Addr;
const MMAP_ACLINT_END      : Addr = MMAP_ACLINT_BEGIN + 'hbfff as Addr;
```

7.2 aclint_memory モジュールの作成

本章では、ACLINT のデバイスを 1 つの 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。

7.2.1 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 7.3)。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 7.3: (aclint_if.veryl)

```
interface aclint_if {
    modport master {
        // TODO
    }
    modport slave {
        ..converse(master)
    }
}
```

7.2.2 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 7.4)。まだどのレジスタも実装していません。

▼リスト 7.4: (aclint_memory.veryl)

```
import eei::*;

module aclint_memory (
    clk    : input  clock      ,
    rst    : input  reset      ,
    membus: modport Membus::slave  ,
    aclint: modport aclint_if::master,
) {
    assign membus.ready = 1;
    always_ff {
        if_reset {
            membus.rvalid = 0;
            membus.rdata  = 0;
        } else {
            membus.rvalid = membus.valid;
        }
    }
}
```

```

    }
}
}
```

7.2.3 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、アドレスに ACLINT をマップします (リスト 7.5、リスト 7.6)。

▼ リスト 7.5: (mmio_controller.veryl)

```

enum Device {
    UNKNOWN,
    RAM,
    ROM,
    DEBUG,
    ACLINT,
}
```

▼ リスト 7.6: (mmio_controller.veryl)

```

if MMAP_ACLINT_BEGIN <= addr && addr <= MMAP_ACLINT_END {
    return Device::ACLINT;
}
```

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 7.7、リスト 7.8)。

▼ リスト 7.7: (mmio_controller.veryl)

```

module mmio_controller (
    clk      : input  clock      ,
    rst      : input  reset      ,
    DBG_ADDR : input  Addr       ,
    req_core : modport Membus::slave ,
    ram_membus : modport Membus::master,
    rom_membus : modport Membus::master,
    dbg_membus : modport Membus::master,
    aclint_membus: modport Membus::master,
) {
```

▼ リスト 7.8: (mmio_controller.veryl)

```

function reset_all_device_masters () {
    reset_membus_master(ram_membus);
    reset_membus_master(rom_membus);
    reset_membus_master(dbg_membus);
    reset_membus_master(aclint_membus);
}
```

`ready`、`rvalid` を取得する関数に ACLINT を登録します (リスト 7.9、リスト 7.10)。

▼ リスト 7.9: (mmio_controller.veryl)

```
Device::ACLINT: return aclint_membus.ready;
```

▼ リスト 7.10: (mmio_controller.veryl)

```
Device::ACLINT: return aclint_membus.rvalid;
```

ACLINT の `rvalid`、`rdata` を `req_core` に割り当てます (リスト 7.11)。

▼ リスト 7.11: (mmio_controller.veryl)

```
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 7.12)。

▼ リスト 7.12: (mmio_controller.veryl)

```
Device::ACLINT: {
    aclint_membus <> req;
    aclint_membus.addr -= MMAP_ACLINT_BEGIN;
}
```

7.2.4 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_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をインスタンス化します (リスト 7.13、リスト 7.14)。

▼ リスト 7.13: (top.veryl)

```
inst aclint_membus : Membus;
```

▼ リスト 7.14: (top.veryl)

```
inst aclint_core_bus: aclint_if;
```

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 7.15、リスト 7.16)。

▼ リスト 7.15: (top.veryl)

```
inst aclintm: aclint_memory (
    clk           ,
    rst           ,
    membus: aclint_membus ,
    aclint: aclint_core_bus,
);
```

▼ リスト 7.16: (top.veryl)

```
inst mmioc: mmio_controller (
    clk           ,
    rst           ,
    DBG_ADDR     : MMAP_DBG_ADDR  ,
    req_core     : mmio_membus   ,
    ram_membus   : mmio_ram_membus,
    rom_membus   : mmio_rom_membus,
    dbg_membus   ,
    aclint_membus   ,
);

```

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 7.17、リスト 7.18、リスト 7.19、リスト 7.20）。

▼リスト 7.17: (core.veryl)

```
module core (
    clk      : input  clock           ,
    rst      : input  reset           ,
    i_membus: modport core_inst_if::master,
    d_membus: modport core_data_if::master,
    led      : output UIntX          ,
    aclint  : modport aclint_if::slave  ,
) {
```

▼リスト 7.18: (top.veryl)

```
inst c: core (
    clk           ,
    rst           ,
    i_membus: i_membus_core  ,
    d_membus: d_membus_core  ,
    led           ,
    aclint  : aclint_core_bus,
);

```

▼リスト 7.19: (csrunit.veryl)

```
minstret  : input  UInt64          ,
led       : output UIntX          ,
aclint   : modport aclint_if::slave  ,
) {
```

▼リスト 7.20: (core.veryl)

```
minstret          ,
led              ,
aclint          ,
);

```

7.3 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 (TODO テーブル)。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。

TODO テーブル (最大 4095 個)

仕様上は `mhartid` と `ACLINT` のレジスタの `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。

7.3.1 MSIP レジスタを実装する

`ACLINT` モジュールに MSIP レジスタを実装します (図 7.2)。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 7.21)。

▼ リスト 7.21: (aclint_if.veryl)

```
interface aclint_if {
    var msip: logic;
    modport master {
        msip: output,
    }
    modport slave {
        ..converse(master)
    }
}
```

`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 7.22、リスト 7.23、リスト 7.24)。

▼ リスト 7.22: (aclint_memory.veryl)

```
var msip0: logic;
```

▼ リスト 7.23: (aclint_memory.veryl)

```
always_ff {
    if_reset {
        membus.rvalid = 0;
        membus.rdata  = 0;
        msip0        = 0;
    }
}
```

▼ リスト 7.24: (aclint_memory.veryl)

```
if membus.valid {
    let addr: Addr = {membus.addr[XLEN - 1:3], 3'b0};
```

```

if membus.wen {
    let M: logic<MEMBUS_DATA_WIDTH> = membus.wmask_expand();
    let D: logic<MEMBUS_DATA_WIDTH> = membus.wdata & M;
    case addr {
        MMAP_ACLINT_MSIP: msip0 = D[0] | msip0 & ~M[0];
        default           : {}
    }
} else {
    membus.rdata = case addr {
        MMAP_ACLINT_MSIP: {63'b0, msip0},
        default           : 0,
    };
}
}

```

`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 7.25)。

▼ リスト 7.25: (aclint_memory.veryl)

```

always_comb {
    aclint.msip = msip0;
}

```

7.3.2 mip、mie レジスタを実装する

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。

csrunit モジュールに mie レジスタを作成します (リスト 7.26、リスト 7.27)。

▼ リスト 7.26: (csrunit.veryl)

```

var mie      : UIntX ;

```

▼ リスト 7.27: (csrunit.veryl)

```

if_reset {
    mode      = PrivMode::M;
    mstatus  = 0;
    mtvec    = 0;
    mie      = 0;
    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 7.28)。

▼ リスト 7.28: (csrunit.veryl)

```

let mip: UIntX = {
    1'b0 repeat XLEN - 12, // 0
    1'b0, // MEIP
}

```

```

1'b0, // 0
1'b0, // SEIP
1'b0, // 0
1'b0, // MTIP
1'b0, // 0
1'b0, // STIP
1'b0, // 0
aclint.msip, // MSIP
1'b0, // 0
1'b0, // SSIP
1'b0, // 0
};

```

mie、mip レジスタの値を読み込むようにします (リスト 7.29)。

▼リスト 7.29: (csrunit.veryl)

```

CsrAddr::MTVEC : mtvec,
CsrAddr::MIP   : mip,
CsrAddr::MIE   : mie,
CsrAddr::MCYCLE : mcycle,

```

mie レジスタの書き込みマスクを設定して、MSIE ビットを書き込むようにします (リスト 7.30、リスト 7.31、リスト 7.32)。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 7.30: (csrunit.veryl)

```
const MIE_WMASK : UIntX = 'h0000_0000_0000_0088 as UIntX;
```

▼リスト 7.31: (csrunit.veryl)

```

CsrAddr::MTVEC : MTVEC_WMASK,
CsrAddr::MIE   : MIE_WMASK,
CsrAddr::MSCRATCH: MSCRATCH_WMASK,

```

▼リスト 7.32: (csrunit.veryl)

```

if is_wsc {
    case csr_addr {
        CsrAddr::MSTATUS : mstatus = wdata;
        CsrAddr::MTVEC   : mtvec   = wdata;
        CsrAddr::MIE     : mie     = wdata;
        CsrAddr::MSCRATCH: mscratch = wdata;
    }
}

```

7.3.3 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 7.33、リスト 7.34)。

▼リスト 7.33: (csrunit.veryl)

```
const MSTATUS_WMASK : UIntX = 'h0000_0000_0000_0088 as UIntX;
```

▼リスト 7.34: (csrunit.veryl)

```
// mstatus bits
let mstatus_mpie: logic = mstatus[7];
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 7.35)。また、MRET 命令で mmstatus.MIE に mstatus.MPIE、mstatus.MPIE に 0 を設定します。

▼リスト 7.35: (csrunit.veryl)

```
if raise_trap {
    if raise_expt {
        mepc    = pc;
        mcause = trap_cause;
        mtval   = expt_value;
        // save mstatus.mie to mstatus.mpie
        // and set mstatus.mie = 0
        mstatus[7] = mstatus[3];
        mstatus[3] = 0;
    } else if trap_return {
        // set mstatus.mie = mstatus.mpie
        //      mstatus.mpie = 0
        mstatus[3] = mstatus[7];
        mstatus[7] = 0;
    }
}
```

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。

7.3.4 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

割り込みは csrunit モジュールに有効な命令が供給されているときにのみ発生させることができます、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 ステージと同時に実行しているため、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

もしれませんが、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ときに MEM ステージを無効化します。

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 7.36、リスト 7.37)。

▼ リスト 7.36: (csrunit.veryl)

```
rs1_data  : input  UIntX      ,
can_intr  : input  logic      ,
rdata      : output UIntX      ,
```

▼ リスト 7.37: (core.veryl)

```
rs1_data  : memq_rdata.rs1_data  ,
can_intr  : mems_is_new          ,
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 7.38)。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 7.38: (core.veryl)

```
inst memu: memunit (
    clk          ,
    rst          ,
    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 7.39)。

▼ リスト 7.39: (core.veryl)

```
} else {
    if !valid {
        state = State::Init;
    } else {
        case state {
            State::Init: if is_new & inst_is_memop(ctrl) {
```

割り込みの判定

割り込みを起こせるかどうか、`cause`、ジャンプ先を示す変数を作成します (リスト 7.40)。

▼ リスト 7.40: (csrunit.veryl)

```
// Interrupt
let raise_interrupt : logic = valid && can_intr && mstatus_mie && (mip & mie) != 0;
let interrupt_cause : UIntX = CsrCause::MACHINE_SOFTWARE_INTERRUPT;
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 7.41)。本書では例外を優先します。

▼ リスト 7.41: (csrunit.veryl)

```
assign raise_trap = raise_expt || raise_interrupt || trap_return;
let trap_cause: UIntX = switch {
    raise_expt      : expt_cause,
    raise_interrupt: interrupt_cause,
    default         : 0,
};
assign trap_vector = switch {
    raise_expt      : mtvec,
    raise_interrupt: interrupt_vector,
    trap_return     : mepc,
    default         : 0,
};
```

割り込みの時に MRET 命令の判定が 0 になるようにしておきます (リスト 7.42)。

▼ リスト 7.42: (csrunit.veryl)

```
// Trap Return
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 7.43)。

▼ リスト 7.43: (csrunit.veryl)

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

7.3.5 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 7.44)。

▼ リスト 7.44: (test/mswi.c)

```
#define MSIP0 ((volatile unsigned int *)0x2000000)
#define DEBUG_REG ((volatile unsigned long long*)0x40000000)
#define MIE_MSIE (1 << 3)
#define MSTATUS_MIE (1 << 3)

void interrupt_handler(void);
```

```

void w_mtvec(unsigned long long x) {
    asm volatile("csrw mtvec, %0" : : "r" (x));
}

void w_mie(unsigned long long x) {
    asm volatile("csrw mie, %0" : : "r" (x));
}

void w_mstatus(unsigned long long x) {
    asm volatile("csrw mstatus, %0" : : "r" (x));
}

void main(void) {
    w_mtvec((unsigned long long)interrupt_handler);
    w_mie(MIE_MSIE);
    w_mstatus(MSTATUS_MIE);
    *MSIP0 = 1;
    while (1) *DEBUG_REG = 3; // fail
}

void interrupt_handler(void) {
    *DEBUG_REG = 1; // success
}

```

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^{*1}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。

7.4 mtvec の Vectored モードの実装

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 7.5)。

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。

例外のジャンプ先のアドレスは、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 7.45)。

^{*1} コンパイル、実行方法は「3.6.4 出力をテストする」(p.60) を参考にしてください。

▼リスト 7.45: (csrunit.veryl)

```
const MTVEC_WMASK : UIntX = 'hffff_ffff_ffff_fffd;
```

割り込みのジャンプ先を MODE と cause に応じて変更します (リスト 7.46)。

▼リスト 7.46: (csrunit.veryl)

```
let interrupt_vector: Addr = if mtvec[0] == 0 ? {mtvec[msb:2], 2'b0} : // Direct
    {mtvec[msb:2] + interrupt_cause[msb - 2:0], 2'b0}; // Vectored
```

例外のジャンプ先を、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 7.47、リスト 7.48)。新しく `expt_vector` を定義し、`trap_vector` に割り当てます。

▼リスト 7.47: (csrunit.veryl)

```
let expt_vector: Addr = {mtvec[msb:2], 2'b0};
```

▼リスト 7.48: (csrunit.veryl)

```
assign trap_vector = switch {
    raise_expt      : expt_vector,
    raise_interrupt: interrupt_vector,
    trap_return     : mepc,
    default         : 0,
};
```

7.5 タイマ割り込みの実装 (MTIMER)

7.5.1 タイマ割り込みの仕組み

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 (TODO テーブル)。本書では MTIMECMP の後ろに MTIME を配置します。

TODO テーブル (MTIME) TODO テーブル (MTIMECMP 最大 4095 個)

MTIME レジスタは、固定された周波数でのサイクル数をカウ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。

7.5.2 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 mhartid 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 7.49、リスト 7.50、リスト 7.51)。`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 7.49: (aclint_memory.veryl)

```
var msip0      : logic ;
var mtime      : UInt64;
var mtimecmp0: UInt64;
```

▼ リスト 7.50: (aclint_memory.veryl)

```
always_ff {
    if_reset {
        membus.rvalid = 0;
        membus.rdata  = 0;
        msip0        = 0;
        mtime        = 0;
        mtimecmp0    = 0;
    }
}
```

▼ リスト 7.51: (aclint_memory.veryl)

```
if membus.wen {
    let M: logic<MEMBUS_DATA_WIDTH> = membus.wmask_expand();
    let D: logic<MEMBUS_DATA_WIDTH> = membus.wdata & M;
    case addr {
        MMAP_ACLINT_MSIP    : msip0      = D[0] | msip0 & ~M[0];
        MMAP_ACLINT_MTIME   : mtime      = D | mtime & ~M;
        MMAP_ACLINT_MTIMECMP: mtimecmp0 = D | mtimecmp0 & ~M;
        default             : {}
    }
} else {
    membus.rdata = case addr {
        MMAP_ACLINT_MSIP    : {63'b0, msip0},
        MMAP_ACLINT_MTIME   : mtime,
        MMAP_ACLINT_MTIMECMP: mtimecmp0,
        default             : 0,
    };
}
```

`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 7.52、リスト 7.53)。

▼ リスト 7.52: (aclint_if.veryl)

```
var msip: logic;
var mtip: logic;
modport master {
    msip: output,
    mtip: output,
}
```

▼リスト 7.53: (aclint_memory.veryl)

```
always_comb {
    aclint.msip = msip0;
    aclint.mtip = mtime >= mtimemcmp0;
}
```

7.5.3 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 7.54)。

▼リスト 7.54: (csrunit.veryl)

```
let mip: UIntX = {
    1'b0 repeat XLEN - 12, // 0, LCOFIP
    1'b0, // MEIP
    1'b0, // 0
    1'b0, // SEIP
    1'b0, // 0
    aclint.mtip, // MTIP
    1'b0, // 0
    1'b0, // STIP
    1'b0, // 0
    aclint.msip, // MSIP
    1'b0, // 0
    1'b0, // SSIP
    1'b0, // 0
};
```

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 7.55)。

▼リスト 7.55: (csrunit.veryl)

```
let interrupt_pending: UIntX = mip & mie;
let raise_interrupt : logic = valid && can_intr && mstatus_mie && interrupt_pending != 0;
let interrupt_cause : UIntX = switch {
    interrupt_pending[3]: CsrCause::MACHINE_SOFTWARE_INTERRUPT,
    interrupt_pending[7]: CsrCause::MACHINE_TIMER_INTERRUPT,
    default : 0,
};
let interrupt_vector: Addr = if mtvec[0] == 0 ? {mtvec[msb:2], 2'b0} : // Direct
    {mtvec[msb:2] + interrupt_cause[msb - 2:0], 2'b0}; // Vectored
```

7.5.4 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 7.56)。

▼リスト 7.56: (mtime.c)

```

#define MTIMECMP0 ((volatile unsigned int *)0x2004000)
#define MTIME     ((volatile unsigned int *)0x2007ff8)
#define DEBUG_REG ((volatile unsigned long long*)0x40000000)
#define MIE_MTIE (1 << 7)
#define MSTATUS_MIE (1 << 3)

void interrupt_handler(void);

void w_mtvec(unsigned long long x) {
    asm volatile("csrw mtvec, %0" : : "r" (x));
}

void w_mie(unsigned long long x) {
    asm volatile("csrw mie, %0" : : "r" (x));
}

void w_mstatus(unsigned long long x) {
    asm volatile("csrw mstatus, %0" : : "r" (x));
}

void main(void) {
    w_mtvec((unsigned long long)interrupt_handler);
    *MTIMECMP0 = *MTIME + 1000000; // この数値は適当に調整する
    w_mie(MIE_MTIE);
    w_mstatus(MSTATUS_MIE);
    while (1);
    *DEBUG_REG = 3; // fail
}

void interrupt_handler(void) {
    *DEBUG_REG = 1; // success
}

```

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 mtimetcmp0 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。mtimetcmp0 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。

7.6 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します。

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 7.57)。

▼ リスト 7.57: (inst_decoder.veryl)

```
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
bits == 32'h10500073, // WFI
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 7.58、リスト 7.59)。

▼ リスト 7.58: (csrunit.veryl)

```
let is_wfi: logic = inst_bits == 32'h10500073;
```

▼ リスト 7.59: (csrunit.veryl)

```
if raise_expt || raise_interrupt {
    mepc = if raise_expt ? pc : // exception
        if raise_interrupt && is_wfi ? pc + 4 : pc; // interrupt when wfi / interrupt
    mcause = trap_cause;
```

7.7 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*2。

`CsrAddr` 型にレジスタのアドレスを追加します (リスト 7.60)。

▼ リスト 7.60: (eei.veryl)

```
// Unprivileged Counter/Timers
CYCLE = 12'hC00,
TIME = 12'hC01,
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 7.61、リスト 7.62)。

*2 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。

▼リスト 7.61: (aclint_if.veryl)

```
import eei::*;

interface aclint_if {
    var msip : logic ;
    var mtip : logic ;
    var mtime: UInt64;
    modport master {
        msip : output,
        mtip : output,
        mtime: output,
    }
}
```

▼リスト 7.62: (aclint_memory.veryl)

```
always_comb {
    aclint.msip  = msip0;
    aclint.mtip  = mtime >= mtimecmp0;
    aclint.mtime = mtime;
}
```

time、instret、cycle レジスタを読み込むようにします (リスト 7.63)。

▼リスト 7.63: (csrunit.veryl)

```
CsrAddr::CYCLE  : mcycle,
CsrAddr::TIME   : aclint.mtime,
CsrAddr::INSTRET : minstret,
```

RAM	0x80000000
Debug I/O	0x40000fff 0x40000000
SSWI	0x20ffff 0x20c000 0x20bfff
MTIMER	0x204000 0x203fff
MSWI	0x200000

31	0																											0	
																												1	
31																													1

▲図 7.2: MSIP レジスタ

63	12	11	10	9	8	7	6	5	4	3	2	1	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 7.3: mip レジスタ

63	12	11	10	9	8	7	6	5	4	3	2	1	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 7.4: mie レジスタ

63	BASE												2	1	0
62													2		

▲図 7.5: mtvec レジスタ

第 8 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

8.1 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 の値を変更します (リスト 8.1)。

▼ リスト 8.1: (csrunit.veryl)

```
let misa      : UIntX  = {2'd2, 1'b0 repeat XLEN - 28, 26'b000001000000100010000101}; // U,>
> M, I, C, A
```

8.2 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 を書き換えて 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 8.2、リスト 8.3)。

▼リスト 8.2: (eei.veryl)

```
// mstatus
const MSTATUS_UXL: UInt64 = 2 << 32;
```

▼リスト 8.3: (csrunit.veryl)

```
always_ff {
    if_reset {
        mode      = PrivMode::M;
        mstatus   = MSTATUS_UXL;
        mtvec    = 0;
```

8.3 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、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 8.4)。

▼リスト 8.4: (csrunit.veryl)

```
const MSTATUS_WMASK : UIntX = 'h0000_0000_0020_0088 as UIntX;
```

8.4 mstatus.MPP の実装

図 TODO

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 (図 TODO)。

これにより、トラップによる U(M)-mode から M-mode への遷移 (図 TODO)、MRET 命令による M-mode から U-mode への遷移 (図 TODO) 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 ()

▼リスト 8.5:

```
TODO M->Uのサンプルコード
```

mstatus.MPP に値を書き込めるようにします (リスト 8.6)。

▼リスト 8.6: (csrunit.veryl)

```
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 8.7、リスト 8.8)。

▼リスト 8.7: (csrunit.veryl)

```
CsrAddr::MSTATUS : mstatus = validate_mstatus(mstatus, wdata);
```

▼リスト 8.8: (csrunit.veryl)

```
function validate_mstatus (
    mstatus: input UIntX,
    wdata : input UIntX,
) -> UIntX {
    var result: UIntX;
    result = wdata;
    // MPP
    if wdata[12:11] != PrivMode::M && wdata[12:11] != PrivMode::U {
        result[12:11] = mstatus[12:11];
    }
    return result;
}
```

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る (リスト 8.9、リスト 8.10、リスト 8.11、リスト 8.12、リスト 8.13)。

▼リスト 8.9: (csrunit.veryl)

```
let mstatus_mpp : PrivMode = mstatus[12:11] as PrivMode;
let mstatus_mpie: logic      = mstatus[7];
let mstatus_mie : logic      = mstatus[3];
```

▼リスト 8.10: (csrunit.veryl)

```
let interrupt_mode: PrivMode = PrivMode::M;
```

▼リスト 8.11: (csrunit.veryl)

```
let expt_mode : PrivMode = PrivMode::M;
```

▼リスト 8.12: (csrunit.veryl)

```
let trap_return_mode: PrivMode = mstatus_mpp;
```

▼リスト 8.13: (csrunit.veryl)

```
let trap_mode_next: PrivMode = switch {
    raise_expt      : expt_mode,
    raise_interrupt: interrupt_mode,
    trap_return     : trap_return_mode,
    default         : PrivMode::U,
};
```

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 8.14）。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 8.14: (csrunit.veryl)

```
if raise_trap {
    if raise_expt || raise_interrupt {
        ...
        // save current privilege level to mstatus.mpp
        @<b<|mstatus[12:11] = mode; |
    } else if trap_return {
        ...
        // set mstatus.mpp = U (least privilege level)
        mstatus[12:11] = PrivMode::U;
    }
    mode = trap_mode_next;
```

8.5 CSR のアクセス権限の確認

TODO 図

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権限レベルを表しています（TOOD 図）。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 8.15、リスト 8.16、リスト 8.17）。

▼リスト 8.15: (csrunit.veryl)

```
let expt_csr_privViolation: logic = is_wsc && csr_addr[9:8] >: mode; // attempt to access C>SR without privilege level
```

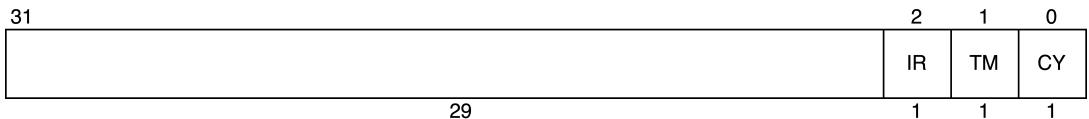
▼リスト 8.16: (csrunit.veryl)

```
let raise_expt: logic = valid && (expt_info.valid || expt_write_READONLY_CSR || expt_csr_priv>vViolation);
```

▼ リスト 8.17: (csrunit.veryl)

```
expt_write_READONLY_CSR: CsrCause::ILLEGAL_INSTRUCTION,
expt_CSR_PRIV_VIOLATION: CsrCause::ILLEGAL_INSTRUCTION,
default : 0,
```

8.6 mcounteren レジスタの実装



▲ 図 8.1: 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マニア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 8.1)。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^{*1}。

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 8.18、リスト 8.19、リスト 8.20、リスト 8.21、リスト 8.22、リスト 8.23)。

▼ リスト 8.18: (csrunit.veryl)

```
var mcounteren: UInt32;
```

▼ リスト 8.19: (csrunit.veryl)

```
CsrAddr::MIE : mie,
CsrAddr::MCOUNTEREN: {1'b0 repeat XLEN - 32, mcounteren},
CsrAddr::MCYCLE : mcycle,
```

▼ リスト 8.20: (csrunit.veryl)

```
const MCOUNTEREN_WMASK: UIntX = 'h0000_0000_0000_0007 as UIntX;
```

▼ リスト 8.21: (csrunit.veryl)

*1 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

```
CsrAddr::MIE      : MIE_WMASK,
CsrAddr::MCOUNTEREN: MCOUNTEREN_WMASK,
CsrAddr::MSCRATCH : MSCRATCH_WMASK,
```

▼リスト 8.22: (csrunit.veryl)

```
mie      = 0;
mcOUNTEREN = 0;
mscratch = 0;
```

▼リスト 8.23: (csrunit.veryl)

```
CsrAddr::MIE      : mie      = wdata;
CsrAddr::MCOUNTEREN: mcOUNTEREN = wdata[31:0];
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 8.24、リスト 8.25）。

▼リスト 8.24: (csrunit.veryl)

```
let expt_zicntr_priv : logic = is_wsc && mode == PrivMode::U && case csr_addr {
    CsrAddr::CYCLE : !mcOUNTEREN[0],
    CsrAddr::TIME  : !mcOUNTEREN[1],
    CsrAddr::INSTRET: !mcOUNTEREN[2],
    default        : 0,
}; // attempt to access Zicntr CSR without permission
```

▼リスト 8.25: (csrunit.veryl)

```
expt_csr_privViolation: CsrCause::ILLEGAL_INSTRUCTION,
expt_zicntr_priv      : CsrCause::ILLEGAL_INSTRUCTION,
default                : 0,
```

8.7

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 8.26、リスト 8.27、リスト 8.28）。

▼リスト 8.26: (csrunit.veryl)

```
let expt_trap_return_priv: logic = is_mret && mode <: PrivMode::M; // attempt to execute trap return instruction in low privilege level
```

▼リスト 8.27: (csrunit.veryl)

```
let raise_expt: logic = valid && (expt_info.valid || expt_write_READONLY_CSR || expt_CSR_PRI>v_violation || expt_ZICNTR_PRIV || expt_trap_return_PRIV);
```

▼リスト 8.28: (csrunit.veryl)

```
expt_ZICNTR_PRIV      : CsrCause::ILLEGAL_INSTRUCTION,  
expt_trap_return_PRIV : CsrCause::ILLEGAL_INSTRUCTION,  
default                : 0,  
};
```

8.8 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図 TODO のようになっています。

図 TODO cause

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 8.29）。

▼リスト 8.29: (eei.veryl)

```
STORE_AMO_ADDRESS_MISALIGNED = 6,  
ENVIRONMENT_CALL_FROM_U_MODE = 8,  
ENVIRONMENT_CALL_FROM_M_MODE = 11,
```

csrunit モジュールの `mode` をポートとして宣言し、ID ステージで ECALL 命令をデコードするときに cause に `mode` を足します（リスト 8.30、リスト 8.31、リスト 8.32、リスト 8.33）。

▼リスト 8.30: (csrunit.veryl)

```
rdata      : output UIntX      ,  
mode       : output PrivMode   ,  
raise_trap : output logic     ,
```

▼リスト 8.31: (core.veryl)

```
var csru_priv_mode : PrivMode;
```

▼リスト 8.32: (core.veryl)

```
rdata      : csru_rdata      ,  
mode       : csru_priv_mode  ,  
raise_trap : csru_raise_trap ,
```

▼リスト 8.33: (core.veryl)

```

} else if ids_inst_bits == 32'h00000073 {
    // ECALL
    exq_wdata.expt.valid      = 1;
    exq_wdata.expt.cause      = CsrCause::ENVIRONMENT_CALL_FROM_U_MODE;
    exq_wdata.expt.cause[1:0]  = csr_u_priv_mode;
    exq_wdata.expt.value      = 0;

```

8.9**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「7.1.2 RISC-V の割り込み」(p.118) 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 8.34)。

▼リスト 8.34: (csrunit.veryl)

```

let raise_interrupt : logic = valid && can_intr && (mode != PrivMode::M || mstatus_mie) &&
> interrupt_pending != 0;

```

第 9 章

S-mode の実装 (1. CSR の実装)

本章では Supervisort 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べき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 に遷移するように変更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 10 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ていきます。

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 9.1: (eei.veryl)

```
enum CsrAddr: logic<12> {
    // Supervisor Trap Setup
    SSTATUS = 12'h100,
    SIE = 12'h104,
    STVEC = 12'h105,
    SCOUNTEREN = 12'h106,
    // Supervisor Trap Handling
    SSCRATCH = 12'h140,
    SEPC = 12'h141,
    SCAUSE = 12'h142,
    STVAL = 12'h143,
    SIP = 12'h144,
    // Supervisor Protection and Translation
    SATP = 12'h180,
```

9.1

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 9.2)。

▼ リスト 9.2: (csrunit.veryl)

```
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、UXLEN と同じように 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 9.3、リスト 9.4)。

▼ リスト 9.3: (eei.veryl)

```
const MSTATUS_UXL: UInt64 = 2 << 32;
const MSTATUS_SXL: UInt64 = 2 << 34;
```

▼ リスト 9.4: (csrunit.veryl)

```
always_ff {
    if_reset {
        mode      = PrivMode:::M;
        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ため、これを S-mode の値 (2'b10) も書き込めるように変更します (リスト 9.5)。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 9.5: (csrunit.veryl)

```
function validate_mstatus (
    mstatus: input UIntX,
    wdata  : input UIntX,
) -> UIntX {
    var result: UIntX;
    result = wdata;
    // MPP
    if wdata[12:11] == 2'b10 {
        result[12:11] = mstatus[12:11];
    }
    return result;
}
```

9.2

scounteren レジスタの実装

「8.6 mcounteren レジスタの実装」(p.144) では mcounteren レジスタによってハードウェ

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 がハードウェアアパフォーマンスマニタにアクセスできるようにす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 でハードウェアアパフォーマンスマ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 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 レジスタと等しいです。また、U-mode でハードウェアアパフォーマンス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 9.6、リスト 9.7、リスト 9.8、リスト 9.9、リスト 9.10、リスト 9.11）。

▼ リスト 9.6: (csrunit.veryl)

```
var scounteren: UInt32;
```

▼ リスト 9.7: (csrunit.veryl)

```
mtval      = 0;
scounteren = 0;
led       = 0;
```

▼ リスト 9.8: (csrunit.veryl)

```
CsrAddr::MTVAL      : mtval,
CsrAddr::SCOUNTEREN: {1'b0 repeat XLEN - 32, scounteren},
CsrAddr::LED        : led,
```

▼ リスト 9.9: (csrunit.veryl)

```
const SCOUNTEREN_WMASK: UIntX = 'h0000_0000_0000_0007 as UIntX;
```

▼ リスト 9.10: (csrunit.veryl)

```
CsrAddr::MTVAL      : MTVAL_WMASK,
CsrAddr::SCOUNTEREN: SCOUNTEREN_WMASK,
CsrAddr::LED        : LED_WMASK,
```

▼ リスト 9.11: (csrunit.veryl)

```
CsrAddr::MTVAL      : mtval      = wdata;
CsrAddr::SCOUNTEREN: scounteren = wdata[31:0];
CsrAddr::LED        : led       = wdata;
```

ハードウェア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 9.12）。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。

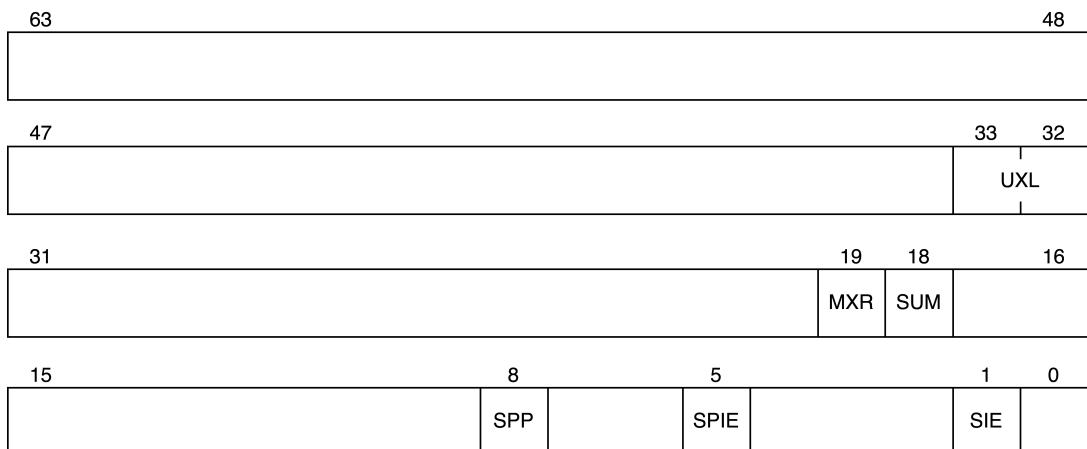
▼ リスト 9.12: (csrunit.veryl)

```

let expt_zicntr_priv      : logic = is_wsc && (mode <= PrivMode::S && case csr_addr {
    CsrAddr::CYCLE  : !mcounteren[0],
    CsrAddr::TIME   : !mcounteren[1],
    CsrAddr::INSTRET: !mcounteren[2],
    default         : 0,
} || mode <= PrivMode::U && case csr_addr {
    CsrAddr::CYCLE  : !scounteren[0],
    CsrAddr::TIME   : !scounteren[1],
    CsrAddr::INSTRET: !scounteren[2],
    default         : 0,
}); // attempt to access Zicntr CSR without permission

```

9.3 sstatus レジスタの実装



▲ 図 9.1: 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 9.13、リスト 9.14)。

▼ リスト 9.13: (csrunit.veryl)

```
const SSTATUS_WMASK : UIntX = 'h0000_0000_0000_0000 as UIntX;
```

▼ リスト 9.14: (csrunit.veryl)

```

CsrAddr::MTVAL      : MTVAL_WMASK,
CsrAddr::SSTATUS    : SSTATUS_WMASK,
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 9.15、リスト 9.16、リスト 9.17)。

▼ リスト 9.15: (csrunit.veryl)

```
const SSTATUS_RMASK: UIntX = 'h8000_0003_018f_e762;
```

▼ リスト 9.16: (csrunit.veryl)

```
let sstatus : UIntX = mstatus & SSTATUS_RMASK;
```

▼ リスト 9.17: (csrunit.veryl)

```
CsrAddr::MTVAL      : mtval,
CsrAddr::SSTATUS    : sstatus,
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 9.18)。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 9.18: (csrunit.veryl)

```
CsrAddr::SSTATUS    : mstatus = validate_mstatus(mstatus, wdata | mstatus & ~SSTATUS_WMASK);
);
```

9.4 トランプの委譲

9.4.1 トランプの委譲

S-mode が実装されているとき、S-mode と U-mode で発生する割り込みと例外のトランプ先の特権レベルを M-mode から S-mode に委譲することができます。現在の特権レベルが 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は、mtvec ではなく stvec にジャンプします。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

図 TODO

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 (図 TODO)。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

図 TODO

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 (図 TODO)。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

割り込みがで 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。委譲された割り込みは現在の特権レベルが 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。同時に委譲されていない割り込みを発生させられると、委譲されていない割り込みを優先します。

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*1。

9.4.2 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レジスタを作成します。stvec、sscratch、sepc、scause、stval レジスタを作成します (リスト 9.19、リスト 9.20、リスト 9.21、リスト 9.22、リスト 9.23、リスト 9.24)。それぞれ、mtvec、sscratch、sepc、scause、stval レジスタと同じ書き込みマスクを設定しています。

*1 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 9.19: (csrunit.veryl)

```
var stvec      : UIntX ;
var sscratch   : UIntX ;
var sepc       : UIntX ;
var scause     : UIntX ;
var stval      : UIntX ;
```

▼リスト 9.20: (csrunit.veryl)

```
stvec      = 0;
sscratch   = 0;
sepc       = 0;
scause     = 0;
stval      = 0;
```

▼リスト 9.21: (csrunit.veryl)

```
CsrAddr::STVEC      : stvec,
CsrAddr::SSCRATCH   : sscratch,
CsrAddr::SEPC       : sepc,
CsrAddr::SCAUSE     : scause,
CsrAddr::STVAL      : stval,
```

▼リスト 9.22: (csrunit.veryl)

```
const STVEC_WMASK      : UIntX = 'hffff_ffff_ffff_ffffd;
const SSCRATCH_WMASK   : UIntX = 'hffff_ffff_ffff_ffff;
const SEPC_WMASK       : UIntX = 'hffff_ffff_ffff_ffffe;
const SCAUSE_WMASK     : UIntX = 'hffff_ffff_ffff_fffff;
const STVAL_WMASK      : UIntX = 'hffff_ffff_ffff_ffff;
```

▼リスト 9.23: (csrunit.veryl)

```
CsrAddr::STVEC      : STVEC_WMASK,
CsrAddr::SSCRATCH   : SSCRATCH_WMASK,
CsrAddr::SEPC       : SEPC_WMASK,
CsrAddr::SCAUSE     : SCAUSE_WMASK,
CsrAddr::STVAL      : STVAL_WMASK,
```

▼リスト 9.24: (csrunit.veryl)

```
CsrAddr::STVEC      : stvec      = wdata;
CsrAddr::SSCRATCH   : sscratch   = wdata;
CsrAddr::SEPC       : sepc       = wdata;
CsrAddr::SCAUSE     : scause     = wdata;
CsrAddr::STVAL      : stval      = wdata;
```

9.4.3 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 9.25、リスト 9.26）。割り込み、例外それぞれの xtvec 変数を定義し、mtvec を使っていたところを xtvec に置き換えていました。

▼リスト 9.25: (csrunit.veryl)

```

let interrupt_xtvec : Addr = if interrupt_mode == PrivMode::M ? mtvec : stvec;
let interrupt_vector: Addr = if interrupt_xtvec[0] == 0 ?
    {interrupt_xtvec[msb:2], 2'b0}
: // Direct
    {interrupt_xtvec[msb:2] + interrupt_cause[msb - 2:0], 2'b0}
; // Vectored

```

▼リスト 9.26: (csrunit.veryl)

```

let expt_xtvec : Addr      = if expt_mode == PrivMode::M ? mtvec : stvec;
let expt_vector: Addr      = {expt_xtvec[msb:2], 2'b0};

```

9.4.4 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 9.27)。

▼リスト 9.27: (csrunit.veryl)

```

if raise_expt || raise_interrupt {
    let xepc: Addr = if raise_expt ? pc : // exception
        if raise_interrupt && is_wfi ? pc + 4 : pc; // interrupt when wfi / interrupt
    if trap_mode_next == PrivMode::M {
        mepc    = xepc;
        mcause = trap_cause;
        if raise_expt {
            mtval = expt_value;
        }
        // save mstatus.mie to mstatus.mpie
        // and set mstatus.mie = 0
        mstatus[7] = mstatus[3];
        mstatus[3] = 0;
        // save current privilege level to mstatus.mpp
        mstatus[12:11] = mode;
    } else {
        sepc    = xepc;
        scause = trap_cause;
        if raise_expt {
            stval = expt_value;
        }
    }
}

```

9.4.5 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 はそれを区別するために必要な 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 9.28、リスト 9.29)。

▼ リスト 9.28: (csrunit.veryl)

```
const MSTATUS_WMASK : UIntX = 'h0000_0000_0020_19aa as UIntX;
```

▼ リスト 9.29: (csrunit.veryl)

```
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 9.30)。

▼ リスト 9.30: (csrunit.veryl)

```
} else {
    sepc = xepc;
    scause = trap_cause;
    if raise_expt {
        stval = expt_value;
    }
    // save sstatus.sie to sstatus.spie
    // and set sstatus.sie = 0
    mstatus[5] = mstatus[1];
    mstatus[1] = 0;
    // save current privilege mode (S or U) to sstatus.spp
    mstatus[8] = mode[0];
}
```

9.4.6 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 9.31)。

▼ リスト 9.31: (inst_decoder.veryl)

```
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
bits == 32'h10200073 || // SRET
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 9.32、リスト 9.33、リスト 9.34)。

▼リスト 9.32: (csrunit.veryl)

```
let is_sret: logic = inst_bits == 32'h10200073;
```

▼リスト 9.33: (csrunit.veryl)

```
assign trap_return      = valid && (is_mret || is_sret) && !raise_expt && !raise_interrup>
>t;
let trap_return_mode : PrivMode = if is_mret ? mstatus_mpp : mstatus_spp;
let trap_return_vector: Addr   = if is_mret ? mepc : sepc;
```

▼リスト 9.34: (csrunit.veryl)

```
assign trap_vector = switch {
    raise_expt      : expt_vector,
    raise_interrupt: interrupt_vector,
    trap_return     : trap_return_vector,
    default         : 0,
};
```

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 9.35)。

▼リスト 9.35: (csrunit.veryl)

```
} else if trap_return {
    if is_mret {
        // set mstatus.mie = mstatus.mpie
        //     mstatus.mpie = 0
        mstatus[3] = mstatus[7];
        mstatus[7] = 0;
        // set mstatus.mpp = U (least privilege level)
        mstatus[12:11] = PrivMode::U;
    } else if is_sret {
        // set sstatus.sie = sstatus.spie
        //     sstatus.spie = 0
        mstatus[1] = mstatus[5];
        mstatus[5] = 0;
        // set sstatus.spp = U (Least privilege level)
        mstatus[8] = 0;
    }
}
```

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 9.36)。

▼リスト 9.36: (csrunit.veryl)

```
let expt_trap_return_priv: logic = (is_mret && mode <: PrivMode::M) || (is_sret && mode <: PrivMode::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9.37)。

▼リスト9.37: (csrunit.veryl)

```
const MSTATUS_WMASK : UIntX = 'h0000_0000_0060_19aa as UIntX;
```

例外を判定します(リスト9.38、リスト9.39)。

▼リスト9.38: (csrunit.veryl)

```
let mstatus_ts : logic = mstatus[22];
```

▼リスト9.39: (csrunit.veryl)

```
let expt_trap_return_priv: logic = (is_mret && mode <: PrivMode::M) || (is_sret && (mode <: PrivMode::S || (mode == PrivMode::S && mstatus_ts)));
```

9.4.7 SEI、SSI、STIを実装する

S-modeを導入すると、mip、mieレジスタの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)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9.40、リスト9.41、リスト9.42、リスト9.43、リスト9.44)。

▼リスト9.40: (csrunit.veryl)

```
const MIP_WMASK : UIntX = 'h0000_0000_0000_0222 as UIntX;
const MIE_WMASK : UIntX = 'h0000_0000_0000_02aa as UIntX;
```

▼リスト9.41: (csrunit.veryl)

```
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演算したもに変更します()。

▼リスト 9.42: (csrunit.veryl)

```
var mip_reg: UIntX;
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`()`。**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 9.43: (csrunit.veryl)

```
mie      = 0;
mip_reg = 0;
mcounteren = 0;
```

▼リスト 9.44: (csrunit.veryl)

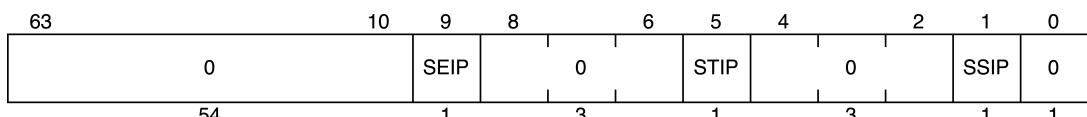
```
CsrAddr::MTVEC    : mtvec      = wdata;
CsrAddr::MIP      : mip_reg   = wdata & MIP_WMASK;
CsrAddr::MIE      : mie        = wdata;
```

cause の設定

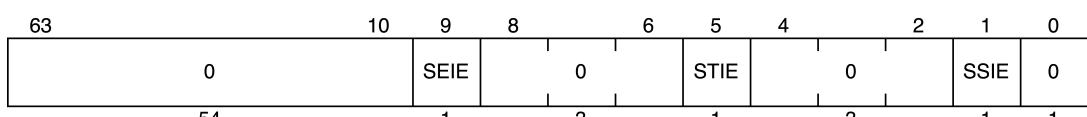
S-mode の割り込みの cause を設定します (リスト 9.45)。

▼リスト 9.45: (csrunit.veryl)

```
let interrupt_cause : UIntX = switch {
    interrupt_pending[3]: CsrCause::MACHINE_SOFTWARE_INTERRUPT,
    interrupt_pending[7]: CsrCause::MACHINE_TIMER_INTERRUPT,
    interrupt_pending[9]: CsrCause::SUPERVISOR_EXTERNAL_INTERRUPT,
    interrupt_pending[1]: CsrCause::SUPERVISOR_SOFTWARE_INTERRUPT,
    interrupt_pending[5]: CsrCause::SUPERVISOR_TIMER_INTERRUPT,
    default              : 0,
};
```

medeleg、**mideleg**、**sip**、**sie** レジスタの実装

▲図 9.2: sip レジスタ



▲図 9.3: 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てい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 9.46、リスト 9.47、リスト 9.48、リスト 9.49、リスト 9.50、リスト 9.51）。

▼ リスト 9.46: (csrunit.veryl)

```
var medeleg    : UInt64;
var mideleg    : UIntX ;
```

▼ リスト 9.47: (csrunit.veryl)

```
let sip        : UIntX  = mip & mideleg;
var sie        : UIntX ;
```

▼ リスト 9.48: (csrunit.veryl)

```
medeleg      = 0;
mideleg      = 0;
```

▼ リスト 9.49: (csrunit.veryl)

```
sie          = 0;
```

▼ リスト 9.50: (csrunit.veryl)

```
CsrAddr::MEDELEG  : medeleg,
CsrAddr::MIDELEG  : mideleg,
```

▼ リスト 9.51: (csrunit.veryl)

```
CsrAddr::SIP      : sip,
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 9.52、リスト 9.53、リスト 9.54、リスト 9.55、リスト 9.56、リスト 9.57）。

▼ リスト 9.52: (csrunit.veryl)

```
const MEDELEG_WMASK  : UIntX = 'hffff_ffff_ffffe_f7ff;
const MIDELEG_WMASK  : UIntX = 'h0000_0000_0000_0222 as UIntX;
```

▼ リスト 9.53: (csrunit.veryl)

```
const SIE_WMASK      : UIntX = 'h0000_0000_0000_0222 as UIntX;
```

▼ リスト 9.54: (csrunit.veryl)

```
CsrAddr::MEDELEG    : MEDELEG_WMASK,
CsrAddr::MIDELEG    : MIDELEG_WMASK,
```

▼リスト 9.55: (csrunit.veryl)

```
CsrAddr::SIE      : SIE_WMASK & mideleg,
```

▼リスト 9.56: (csrunit.veryl)

```
CsrAddr::MEDELEG : medeleg    = wdata;
CsrAddr::MIDELEG : mideleg    = wdata;
```

▼リスト 9.57: (csrunit.veryl)

```
CsrAddr::SIE      : sie        = wdata;
```

9.4.8 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 9.58）。

▼リスト 9.58: (csrunit.veryl)

```
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 9.59、リスト 9.60）。

▼リスト 9.59: (csrunit.veryl)

```
// Interrupt to M-mode
let interrupt_pending_mmode: UIntX = mip & mie & ~mideleg;
let raise_interrupt_mmode : logic = (mode != PrivMode::M || mstatus_mie) && interrupt_pending_mmode != 0;
let interrupt_cause_mmode : UIntX = switch {
    interrupt_pending_mmode[3]: CsrCause::MACHINE_SOFTWARE_INTERRUPT,
    interrupt_pending_mmode[7]: CsrCause::MACHINE_TIMER_INTERRUPT,
    interrupt_pending_mmode[9]: CsrCause::SUPERVISOR_EXTERNAL_INTERRUPT,
    interrupt_pending_mmode[1]: CsrCause::SUPERVISOR_SOFTWARE_INTERRUPT,
    interrupt_pending_mmode[5]: CsrCause::SUPERVISOR_TIMER_INTERRUPT,
    default                  : 0,
};
```

▼リスト 9.60: (csrunit.veryl)

```
// Interrupt to S-mode
let interrupt_pending_smode: UIntX = sip & sie;
let raise_interrupt_smode : logic = (mode <: PrivMode::S || (mode == PrivMode::S && mstatus_sie)) && interrupt_pending_smode != 0;
let interrupt_cause_smode : UIntX = switch {
    interrupt_pending_smode[9]: CsrCause::SUPERVISOR_EXTERNAL_INTERRUPT,
    interrupt_pending_smode[1]: CsrCause::SUPERVISOR_SOFTWARE_INTERRUPT,
    interrupt_pending_smode[5]: CsrCause::SUPERVISOR_TIMER_INTERRUPT,
```

```
    default          : 0,
};
```

M-mode 向けの割り込みを優先して利用します (リスト 9.61)。

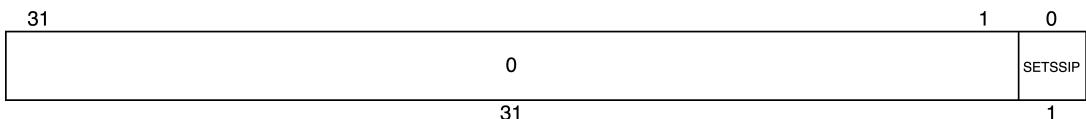
▼ リスト 9.61: (csrunit.veryl)

```
// Interrupt
let raise_interrupt : logic = valid && can_intr && (raise_interrupt_mmode || raise_interrupt_smode);
let interrupt_cause : UIntX = if raise_interrupt_mmode ? interrupt_cause_mmode : interrupt_cause_smode;
let interrupt_xtvec : Addr = if interrupt_mode == PrivMode::M ? mtvec : stvec;
let interrupt_vector: Addr = if interrupt_xtvec[0] == 0 ?
    {interrupt_xtvec[msb:2], 2'b0}
: // Direct
    {interrupt_xtvec[msb:2] + interrupt_cause[msb - 2:0], 2'b0}
; // Vectored
let interrupt_mode: PrivMode = if raise_interrupt_mmode ? PrivMode::M : PrivMode::S;
```

9.5 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 (TODO テーブル) 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。

TODO テーブル 4095 個



▲ 図 9.4: 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 9.62、リスト 9.63)。

▼ リスト 9.62: (aclint_if.veryl)

```
interface aclint_if {
    var msip    : logic ;
    var mtip    : logic ;
```

```

var mtime : UInt64;
var setssip: logic ;
modport master {
    msip    : output,
    mtip    : output,
    mtime   : output,
    setssip: output,
}

```

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 9.63: (aclint_memory.veryl)

```

always_comb {
    aclint.setssip = 0;
    if membus.valid && membus.wen && membus.addr == MMAP_ACLINT_SETSSIP {
        aclint.setssip = membus.wdata[0];
    }
}

```

csrunit モジュールで `setssip` を確認し、mip.SSIP を立てるようにします（リスト 9.64、リスト 9.65、リスト 9.66）。

▼ リスト 9.64: (csrunit.veryl)

```
let setssip: UIntX = {1'b0 repeat XLEN - 2, aclint.setssip, 1'b0};
```

▼ リスト 9.65: (csrunit.veryl)

```

} else {
    mcycle  += 1;
    mip_reg |= setssip;
}

```

▼ リスト 9.66: (csrunit.veryl)

```
CsrAddr::MIP      : mip_reg    = (wdata & MIP_WMASK) | setssip;
```

第 10 章

S-mode の実装 (2. 仮想記憶システム)

10.1 概要

10.1.1 仮想記憶システム

TODO 図

仮想記憶 (Virtual Memory) とは、メモリを管理する手法の一種です。論理的なアドレス (logical address、論理アドレス) を物理的なアドレス (physical address、物理アドレス) に変換することにより、実際のアドレス (real address、実アドレス) 空間とは異なる仮想的なアドレス (virtual address、仮想アドレス) 空間を提供することができます。

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。

10.1.2 ページング方式

図

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、ページテーブル (Page Table) にページを参照するための情報を格納します。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) と呼びます。論理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ページテーブルエントリを参照して行います。これ以降、ページテーブルエントリのことを PTE と呼びます。

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 向けには Sv39、Sv48、Sv57 が定義されています。

TODO 図

本章で実装する Sv39 のアドレス変換を簡単に説明します。

(a) satp レジスタの PPN フィールドと論理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最終的な物理アドレスを作る。(d) PTE が次の PTE を指しているとき、PTE のフィールドと論理アドレスのフィールドから次の PTE の物理アドレスを作る。(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。

TODO 図

このように satp レジスタと論理アドレス、PTE を使って多段階のメモリアクセスを行って論理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^{*1}。

10.1.3 satp レジスタ、アドレス変換プロセス

satp レジスタ

63	60 59	43 43	0
MODE	ASID	PPN	
4	16	44	

▲図 10.1: 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 TODO テーブルのように対応しています。MODE 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)。

TODO テーブル

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 ASID を無視したアドレス変換を実装します^{*2}。

^{*1} 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。

^{*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。

TODO 図 (Sv39)

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 (TODO 図)。

アドレス変換プロセス (Sv39)

Sv39 の仮想アドレスは次の方法によって物理アドレスに変換されます^{*3}。

TODO プロセス

基本的にアドレス変換は S-mode、U-mode で有効になります。TODO ここで説明 mstatus レジスタの MXR、SUM、MPRV ビットを利用すると、プロセス TODO の特権レベル、PTE の権限について挙動を少し変更できます。これらのビットについては実装するときに解説します。

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。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。

10.1.4 実装順序

図

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、シンプ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、amountit モジュールは仮想アドレスを扱うことがありますが、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、amountit モジュールと mmio_controller モジュールの間に PTW を配置します (図 TODO)。

TODO 図

本章では、仮想記憶システムを次の順序で実装します。

- 例外を伝達するインターフェースを実装する
- Bare にだけ対応したアドレス変換モジュールを実装する
- satp レジスタ、mstatus の MXR、SUM、MPRV ビットを作成する
- Sv39 を実装する
- SFENCE.VMA 命令、FENCEI 命令を実装する

10.2 メモリインターフェースの例外の実装

PTW で発生した例外は、最終的に csruniit モジュールで処理します。そのために、例外の情報

^{*3} 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。

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` 型に追加してください (リスト 10.1)。

▼ リスト 10.1: (eei.veryl)

```
INSTRUCTION_PAGE_FAULT = 12,
LOAD_PAGE_FAULT = 13,
STORE_AMO_PAGE_FAULT = 15,
```

10.2.1 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 10.2)。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 10.2: (eei.veryl)

```
struct MemException {
    valid      : logic,
    page_fault: logic,
}
```

`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 10.3、リスト 10.4、リスト 10.5、リスト 10.6)。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 10.3: (membus_if.veryl, core_data_if.veryl, core_inst_if.veryl)

```
var expt  : eei::MemException          ;
```

▼ リスト 10.4: (membus_if.veryl, core_data_if.veryl, core_inst_if.veryl)

```
modport master {
    ...
    expt      : input ,
    ...
}
```

▼ リスト 10.5: (membus_if.veryl)

```
modport slave {
    ...
    expt      : output,
    ...
}
```

▼ リスト 10.6: (membus_if.veryl)

```
modport response {
    rvalid: output,
    rdata : output,
```

```
    expt : output,
}
```

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 10.7、リスト 10.8)。いまのところ、デバイスは例外を発生させません。

▼ リスト 10.7: (membus_if.veryl)

```
always_comb {
    req_core.ready  = 0;
    req_core.rvalid = 0;
    req_core.rdata  = 0;
    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 10.8: (top.veryl)

```
always_comb {
    i_membus.ready  = mmio_membus.ready && !d_membus.valid;
    i_membus.rvalid = mmio_membus.rvalid && memarb_last_i;
    i_membus.rdata  = mmio_membus.rdata;
    i_membus.expt   = mmio_membus.expt;

    d_membus.ready  = mmio_membus.ready;
    d_membus.rvalid = mmio_membus.rvalid && !memarb_last_i;
    d_membus.rdata  = mmio_membus.rdata;
    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 10.9、リスト 10.10)。

▼ リスト 10.9: (inst_fetcher.veryl)

```
struct fetch_fifo_type {
    Addr: Addr           ,
    bits: logic <MEMBUS_DATA_WIDTH>, ,
    expt: MemException   ,
}
```

▼ リスト 10.10: (inst_fetcher.veryl)

```
struct issue_fifo_type {
    Addr : Addr           ,
    bits : Inst           ,
    is_rvc: logic          ,
    expt : MemException,
}
```

メモリからの例外情報を `fetch_fifo` に保存します (リスト 10.11)。

▼ リスト 10.11: (inst_fetcher.veryl)

```
always_comb {
    fetch_fifo_flush      = core_if.is_hazard;
    fetch_fifo_wvalid     = fetch_requested && mem_if.rvalid;
    fetch_fifo_wdata.addr = fetch_pc_requested;
    fetch_fifo_wdata.bits = mem_if.rdata;
    fetch_fifo_wdata.expt = mem_if.expt;
}
```

`fetch_fifo` から `issue_fifo` に例外情報を伝達します (リスト 10.12)、リスト 10.13、リスト 10.14)。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 10.12: (inst_fetcher.veryl)

```
always_comb {
    let raddr : Addr                  = fetch_fifo_rdata.addr;
    let rdata : logic <MEMBUS_DATA_WIDTH> = fetch_fifo_rdata.bits;
    let expt : MemException          = fetch_fifo_rdata.expt;
    let offset: logic <3>            = issue_pc_offset;

    fetch_fifo_rready     = 0;
    issue_fifo_wvalid    = 0;
    issue_fifo_wdata     = 0;
    issue_fifo_wdata.expt = expt;
}
```

▼ リスト 10.13: (inst_fetcher.veryl)

```
fetch_fifo_rready = 1;
if rvcc_is_rvc || expt.valid {
    issue_fifo_wvalid    = 1;
    issue_fifo_wdata.addr = {raddr[msb:3], offset};
    issue_fifo_wdata.is_rvc = 1;
    issue_fifo_wdata.bits  = rvcc_inst32;
```

▼ リスト 10.14: (inst_fetcher.veryl)

```
if issue_pc_offset == 6 && !rvcc_is_rvc && !issue_is_rdata_saved && !fetch_fifo_rdata.expt.v>
>alid {
    if fetch_fifo_rvalid {
        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 10.15)。

▼ リスト 10.15: (inst_fetcher.veryl)

```
always_comb {
    issue_fifo_flush  = core_if.is_hazard;
    issue_fifo_rready = core_if.rready;
    core_if.rvalid    = issue_fifo_rvalid;
```

```

core_if.raddr      = issue_fifo_rdata.addr;
core_if.rdata      = issue_fifo_rdata.bits;
core_if.is_rvc    = issue_fifo_rdata.is_rvc;
core_if.expt      = issue_fifo_rdata.expt;
}

```

amountunit 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 10.16、リスト 10.17、リスト 10.18、）。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 10.16: (amountunit.veryl)

```

always_comb {
    slave.ready  = 0;
    slave.rvalid = 0;
    slave.rdata  = 0;
    slave.expt    = master.expt;
}

```

▼リスト 10.17: (amountunit.veryl)

```

    default: {}
}

if state != State::Init && master.expt.valid {
    slave.ready  = 0;
    slave.rvalid = 1;
}
}

```

▼リスト 10.18: (amountunit.veryl)

```

    State::AMOStoreValid: accept_request_comb();
    default             : {}
}

if state != State::Init && master.expt.valid {
    reset_master();
}
}

```

例外が発生したら、`state` を `State::Init` にリセットするようにします（リスト 10.19）。

▼リスト 10.19: (amountunit.veryl)

```

function on_clock () {
    if state != State::Init && master.expt.valid {
        state = State::Init;
    } else {
        case state {
            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 10.20)。

▼リスト 10.20: (core.veryl)

```
if i_membus.expt.valid {
    // fault
    exq_wdata.expt.valid = 1;
    exq_wdata.expt.cause = CsrCause::INSTRUCTION_PAGE_FAULT;
    exq_wdata.expt.value = ids_pc;
} 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 10.21、リスト 10.22、リスト 10.23、リスト 10.24、リスト 10.25)。

▼リスト 10.21: (csrunit.veryl)

```
module csrunit (
    ...
    can_intr  : input  logic ,
    mem_addr  : input  Addr ,
    rdata      : output UIntX ,
    ...
    membus     : modport core_data_if::master ,
) {
```

▼リスト 10.22: (core.veryl)

```
inst csru: csrunit (
    ...
    mem_addr  : memu_addr ,
    ...
    membus     : d_membus ,
);
```

例外を発生させます。

▼リスト 10.23: (csrunit.veryl)

```
let expt_memory_fault : logic = membus.rvalid && membus.expt.valid;
```

▼リスト 10.24: (csrunit.veryl)

```
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);
let expt_cause: UIntX = switch {
    ...
}
```

```

    expt_memory_fault : if ctrl.is_load ? CsrCause::LOAD_PAGE_FAULT : CsrCause::STORE_
>AMO_PAGE_FAULT,
    default : 0,
};

```

xtval に例外が発生したアドレスを設定します。

▼リスト 10.25: (csrunit.veryl)

```

let expt_value: UIntX = switch {
    expt_info.valid : expt_info.value,
    expt_cause == CsrCause::ILLEGAL_INSTRUCTION : {1'b0 repeat XLEN - $bits(Inst), inst_bits}
>},
    expt_cause == CsrCause::LOAD_PAGE_FAULT : mem_addr,
    expt_cause == CsrCause::STORE_AMO_PAGE_FAULT: mem_addr,
    default : 0
};

```

10.2.2 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 に格納できていません。

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 10.26、リスト 10.27、リスト 10.28、リスト 10.29)。

▼リスト 10.26: (eei.veryl)

```

struct MemException {
    valid : logic ,
    page_fault : logic ,
    addr_offset: logic<3>,
}

```

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 10.27: (inst_fetcher.veryl)

```

if issue_is_rdata_saved {
    issue_fifo_wvalid = 1;
    issue_fifo_wdata.addr = {issue_saved_addr[msb:3], offset};
    issue_fifo_wdata.bits = {rdata[15:0], issue_saved_bits};
}

```

```
issue_fifo_wdata.is_rvc      = 0;
issue_fifo_wdata.expt.addr_offset = 2;
```

tval を生成するとき、オフセット足します。

▼ リスト 10.28: (core.veryl)

```
exq_wdata.expt.valid = 1;
exq_wdata.expt.cause = CsrCause::INSTRUCTION_PAGE_FAULT;
exq_wdata.expt.value = ids_pc + {1'b0 repeat XLEN - 3, i_membus.expt.addr_offset};
```

▼ リスト 10.29: (csrunit.veryl)

```
let expt_value: UIntX = switch {
    expt_info.valid                      : expt_info.value,
    expt_cause == CsrCause::ILLEGAL_INSTRUCTION : {1'b0 repeat XLEN - $bits(Inst), inst_bits},
    expt_cause == CsrCause::LOAD_PAGE_FAULT      : mem_addr + {1'b0 repeat XLEN - 3, membus.>
    >expt.addr_offset},
    expt_cause == CsrCause::STORE_AMO_PAGE_FAULT: mem_addr + {1'b0 repeat XLEN - 3, membus.e>
    >xpt.addr_offset},
    default                                : 0
};
```

10.3 satp レジスタの作成

satp レジスタを実装します (リスト 10.30、リスト 10.31、リスト 10.32、リスト 10.33、リスト 10.34、リスト 10.35、リスト 10.36)。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 10.30: (csrunit.veryl)

```
var satp      : UIntX ;
```

▼ リスト 10.31: (csrunit.veryl)

```
satp      = 0;
```

▼ リスト 10.32: (csrunit.veryl)

```
CsrAddr::SATP      : satp,
```

▼ リスト 10.33: (csrunit.veryl)

```
CsrAddr::SATP      : SATP_WMASK,
```

▼ リスト 10.34: (csrunit.veryl)

```
const SATP_WMASK : UIntX = 'hffff_ffff_ffff_ffff;
```

satp レジスタは、MODE フィールド変更するとき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 10.35: (csrunit.veryl)

```
function validate_satp (
    satp : input UIntX,
    wdata: input UIntX,
) -> UIntX {
    // mode
    if wdata[msb-4] != 0 && wdata[msb-4] != 8 {
        return satp;
    }
    return wdata;
}
```

▼ リスト 10.36: (csrunit.veryl)

```
CsrAddr::SATP      : satp      = validate_satp(satp, wdata);
```

10.4 mstatus の MXR、SUM、MPRV ビットの作成

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 10.37、リスト 10.38）。

▼ リスト 10.37: (csrunit.veryl)

```
const MSTATUS_WMASK : UIntX = 'h0000_0000_006e_19aa as UIntX;
```

▼ リスト 10.38: (csrunit.veryl)

```
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 10.39、リスト 10.40）。

▼ リスト 10.39: (csrunit.veryl)

```
let mstatus_mxr : logic  = mstatus[19];
let mstatus_sum : logic = mstatus[18];
let mstatus_mprv: logic = mstatus[17];
```

mstatus.MPRV は、M-mode 以外のモードにトラップするときに `0` に設定す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 10.40: (csrunit.veryl)

```

} else if trap_return {
    // set mstatus.mprv = 0 when new mode != M-mode
    if trap_mode_next <: PrivMode::M {
        mstatus[17] = 0;
    }
    if is_mret {

```

10.5 アドレス変換モジュール (PTW) の作成

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モジュール (ptw) 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「10.6 Sv39 の実装」(p.182) で実装します。

10.5.1 CSR のインターフェースを実装する

ptw で使用する CSR を csrunit モジュールから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 10.41)。

▼リスト 10.41: (ptw_ctrl_if.veryl)

```

import eei::*;

interface ptw_ctrl_if {
    var priv: PrivMode;
    var satp: UIntX ;
    var mxr : logic ;
    var sum : logic ;
    var mprv: logic ;
    var mpp : PrivMode;

    modport master {
        priv: output,
        satp: output,
        mxr : output,
        sum : output,
        mprv: output,
        mpp : output,
    }

    modport slave {
        is_enabled: import,
        ..converse(master)
    }

    function is_enabled (
        is_inst: input logic,
    ) -> logic {

```

```

    if satp[msb-4] == 0 {
        return 0;
    }
    if is_inst {
        return priv <= PrivMode::S;
    } else {
        return (if mprv ? mpp : priv) <= PrivMode::S;
    }
}
}

```

`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 以下なら有効になります。

10.5.2 Bare だけの ptw モジュールを作成する

`src/ptw.veryl` を作成し、次のようなポートを記述します (リスト 10.42)。

▼ リスト 10.42: (ptw.veryl)

```

import eei::*;

module ptw (
    clk      : input  clock      ,
    rst      : input  reset      ,
    is_inst: input  logic      ,
    slave   : modport Membus::slave ,
    master  : modport Membus::master ,
    ctrl    : modport ptw_ctrl_if::slave,
) {

```

`slave` は core モジュール側から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、`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 10.43)。

▼ リスト 10.43: (ptw.veryl)

```
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 10.44)。

▼ リスト 10.44: (ptw.veryl)

```
enum State {
    IDLE,
```

```

    EXECUTE_READY,
    EXECUTE_VALID,
}

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` からのレスポンスを待ちます。`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ておきます(リスト 10.45)。

▼ リスト 10.45: (ptw.veryl)

```
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 10.46、リスト 10.47)。`master` に要求を割り当てるとき、アドレスだけ `physical_addr` レジスタの値を割り当てるようにしておきます。

▼ リスト 10.46: (ptw.veryl)

```
var physical_addr: Addr;
```

▼ リスト 10.47: (ptw.veryl)

```

function assign_master (
    addr : input Addr           ,
    wen  : input logic          ,
    wdata: input logic<MEMBUS_DATA_WIDTH>  ,
    wmask: input logic<MEMBUS_DATA_WIDTH / 8>,
) {
    master.valid = 1;
    master.addr  = addr;
    master.wen   = wen;
    master.wdata = wdata;
    master.wmask = wmask;
}

```

```

function accept_request_comb () {
    if slave.ready && slave.valid && !paging_enabled {
        assign_master(slave.addr, slave.wen, slave.wdata, slave.wmask);
    }
}

always_comb {
    master.valid = 0;
    master.addr = 0;
    master.wen = 0;
    master.wdata = 0;
    master.wmask = 0;

    case state {
        State::IDLE : accept_request_comb();
        State::EXECUTE_READY: assign_master(physical_addr, slave_saved.wen, slave_saved.wdata, slave_saved.wmask);
        State::EXECUTE_VALID: if master.rvalid {
            accept_request_comb();
        }
        default: {}
    }
}

```

状態に基づいて、 `slave` に `ready` と結果を割り当てます (リスト 10.48)。

▼ リスト 10.48: (ptw.veryl)

```

always_comb {
    slave.ready = 0;
    slave.rvalid = 0;
    slave.rdata = 0;
    slave.expt = 0;

    case state {
        State::IDLE : slave.ready = 1;
        State::EXECUTE_VALID: {
            slave.ready = master.rvalid;
            slave.rvalid = master.rvalid;
            slave.rdata = master.rdata;
            slave.expt = master.expt;
        }
        default: {}
    }
}

```

状態を遷移する処理を記述します (リスト 10.49)。要求を受け入れるとき、 `slave_saved` に要求を保存します。

▼ リスト 10.49: (ptw.veryl)

```

function accept_request_ff () {
    slave_saved.valid = slave.ready && slave.valid;
    if slave.ready && slave.valid {
        slave_saved.addr = slave.addr;
        slave_saved.wen = slave.wen;
        slave_saved.wdata = slave.wdata;
        slave_saved.wmask = slave.wmask;
        if paging_enabled {
            // TODO
        } else {
            state = if master.ready ? State::EXECUTE_VALID : State::EXECUTE_READY;
            physical_addr = slave.addr;
        }
    } else {
        state = State::IDLE;
    }
}

function on_clock () {
    case state {
        State::IDLE : accept_request_ff();
        State::EXECUTE_READY: if master.ready {
            state = State::EXECUTE_VALID;
        }
        State::EXECUTE_VALID: if master.rvalid {
            accept_request_ff();
        }
        default: {}
    }
}

function on_reset () {
    state = State::IDLE;
    physical_addr = 0;
    slave_saved.valid = 0;
    slave_saved.addr = 0;
    slave_saved.wen = 0;
    slave_saved.wdata = 0;
    slave_saved.wmask = 0;
}

always_ff {
    if_reset {
        on_reset();
    } else {
        on_clock();
    }
}
}

```

10.5.3 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 10.50)。

▼ リスト 10.50: (top.veryl)

```
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 10.51)。

▼ リスト 10.51: (top.veryl)

```
always_ff {
    if_reset {
        memarb_last_i = 0;
    } else {
        if ptw_membus.ready {
            memarb_last_i = !d_membus.valid;
        }
    }
}

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

    d_membus.ready  = ptw_membus.ready;
    d_membus.rvalid = ptw_membus.rvalid && !memarb_last_i;
    d_membus.rdata  = ptw_membus.rdata;
    d_membus.expt   = ptw_membus.expt;

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

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 10.52)。

▼リスト 10.52: (top.veryl)

```
let ptw_is_inst : logic = (i_membus.ready && i_membus.valid) || // inst ack or
  !(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 10.53)。

▼リスト 10.53: (top.veryl)

```
inst ptw_ctrl: ptw_ctrl_if;
inst paging_unit: ptw (
  clk           ,
  rst           ,
  is_inst: ptw_is_inst,
  slave : ptw_membus ,
  master : mmio_membus,
  ctrl  : ptw_ctrl  ,
);
```

csrunit 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 10.54、リスト 10.55)。

▼リスト 10.54: (core.veryl)

```
module core (
  clk      : input  clock           ,
  rst      : input  reset           ,
  i_membus: modport core_inst_if::master,
  d_membus: modport core_data_if::master,
  led      : output UIntX          ,
  aclint   : modport aclint_if::slave  ,
  ptw_ctrl: modport ptw_ctrl_if::master ,
) {
```

▼リスト 10.55: (top.veryl)

```
inst c: core (
  clk           ,
  rst           ,
  i_membus: i_membus_core  ,
  d_membus: d_membus_core  ,
  led           ,
  aclint : aclint_core_bus,
  ptw_ctrl     ,
);
```

csrunit モジュールにポートを追加し、CSR を割り当てます (リスト 10.56、リスト 10.57、リスト 10.58)。

▼リスト 10.56: (csrunit.veryl)

```
membus      : modport core_data_if::master   ,
ptw_ctrl   : modport ptw_ctrl_if::master   ,
) {
```

▼リスト 10.57: (core.veryl)

```
membus      : d_membus      ,
ptw_ctrl    ,  
);
```

▼リスト 10.58: (csrunit.veryl)

```
always_comb {  
    ptw_ctrl.priv = mode;  
    ptw_ctrl.satp = satp;  
    ptw_ctrl.mxr  = mstatus_mxr;  
    ptw_ctrl.sum  = mstatus_sum;  
    ptw_ctrl.mprv = mstatus_mprv;  
    ptw_ctrl.mpp  = mstatus_mpp;  
}
```

10.6 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「アドレス変換プロセス (Sv39)」(p.166) を参考に動作を確認してください。

10.6.1 定数の定義

ptw モジュールで使用する定数とユーティリティ関数を実装します。

`src/sv39util.veryl` を作成し、次のように記述します (リスト 10.59)。定数は「アドレス変換プロセス (Sv39)」(p.166) で使用しているものと同じです。

▼リスト 10.59: (sv39util.veryl)

```
import eei::*;
package sv39util {
    const PAGESIZE: u32      = 12;
    const PTESIZE : u32      = 8;
    const LEVELS  : logic<2> = 3;

    type Level = logic<2>;

    // 有効な仮想アドレスか判定する
    function is_valid_vaddr (
        va: input Addr,
    ) -> logic {
        let hiaddr: logic<26> = va[msb:38];
        return &hiaddr || ~&hiaddr;
    }

    // 仮想アドレスのVPN[level]フィールドを取得する
    function vpn (
        va   : input Addr ,
```

```

    level: input Level,
) -> logic<9> {
    return case level {
        0      : va[20:12],
        1      : va[29:21],
        2      : va[38:30],
        default: 0,
    };
}

// 最初にフェッチするPTEのアドレスを取得する
function get_first_pte_address (
    satp: input UIntX,
    va : input Addr ,
) -> Addr {
    return {1'b0 repeat XLEN - 44 - PAGESIZE, satp[43:0], 1'b0 repeat PAGESIZE} // a
        + {1'b0 repeat XLEN - 9 - $clog2(PTESIZE), va[38:30], 1'b0 repeat $clog2(PTESIZE)}; // >
    > vpn[2]
}
}

```

10.6.2 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 10.60）。

▼ リスト 10.60: (pte.veryl)

```

import eei::*;
import sv39util::*;

interface PTE39 {
    var value: UIntX;

    function v () -> logic { return value[0]; }
    function r () -> logic { return value[1]; }
    function w () -> logic { return value[2]; }
    function x () -> logic { return value[3]; }
    function u () -> logic { return value[4]; }
    function a () -> logic { return value[6]; }
    function d () -> logic { return value[7]; }

    function reserved -> logic<10> { return value[63:54]; }

    function ppm2 () -> logic<26> { return value[53:28]; }
    function ppm1 () -> logic<9> { return value[27:19]; }
    function ppm0 () -> logic<9> { return value[18:10]; }
    function ppm () -> logic<44> { return value[53:10]; }
}

```

PTE を使ったユーティリティ関数を追加します（リスト 10.61）。

▼リスト 10.61: (pte.veryl)

```

// leaf PTEか判定する
function is_leaf () -> logic { return r() || x(); }

// 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
function is_ppn_aligned (
    level: input Level,
) -> logic {
    return case level {
        0      : 1,
        1      : ppn0() == 0,
        2      : ppn1() == 0 && ppn0() == 0,
        default: 1,
    };
}

// 有効なPTEか判定する
function is_valid (
    level: input Level,
) -> logic {
    if !v() || reserved() != 0 || !r() && w() {
        return 0;
    }
    if is_leaf() && !is_ppn_aligned(level) {
        return 0;
    }
    if !is_leaf() && level == 0 {
        return 0;
    }
    return 1;
}

// 次のlevelのPTEのアドレスを得る
function get_next_pte_addr (
    level: input Level,
    va   : input Addr ,
) -> Addr {
    return {1'b0 repeat XLEN - 44 - PAGESIZE, ppn(), 1'b0 repeat PAGESIZE} + // a
           {1'b0 repeat XLEN - 9 - $clog2(PTESIZE), vpn(va, level - 1), 1'b0 repeat $clog2(PTESIZ>E)};
}

// PTEと仮想アドレスから物理アドレスを生成する
function get_physical_address (
    level: input Level,
    va   : input Addr ,
) -> Addr {
    return {
        8'b0, ppn2(), case level {
            0: {
                ppn1(), ppn0()
            },
            1: {

```

```

        ppn1(), vpn(va, 0)
    },
    2: {
        vpn(va, 1), vpn(va, 0)
    },
    default: 18'b0,
}, va[11:0]
};

}

// A、Dビットを更新する必要があるかを判定する
function need_update_ad (
    wen: input logic,
) -> logic {
    return !a() || wen && !d();
}

// A、Dビットを更新したPTEの下位8ビットを生成する
function get_updated_ad (
    wen: input logic,
) -> logic<8> {
    let a: logic<8> = 1 << 6;
    let d: logic<8> = wen as u8 << 7;
    return value[7:0] | a | d;
}
}

```

10.6.3 ptw モジュールの実装

sv39util パッケージを import します (リスト 10.62)。

▼ リスト 10.62: (ptw.veryl)

```
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 10.63)。 `value` には `master` のロード結果を割り当てます。

▼ リスト 10.63: (ptw.veryl)

```
inst pte      : PTE39;
assign pte.value = master.rdata;
```

TODO 図

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 10.64)。本章ではページングが有効な時に、状態が TODO 図のように遷移するようにします。

▼ リスト 10.64: (ptw.veryl)

```
enum State {
    IDLE,
    WALK_READY,
    WALK_VALID,
```

```

SET_AD,
EXECUTE_READY,
EXECUTE_VALID,
PAGE_FAULT,
}

```

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 ビットを格納するためのレジスタを定義します (リスト 10.65、リスト 10.66)。

▼ リスト 10.65: (ptw.veryl)

```

var physical_addr: Addr      ;
var taddr          : Addr      ;
var level          : Level     ;
var wdata_ad       : logic<8>;

```

▼ リスト 10.66: (ptw.veryl)

```

function on_reset () {
    state          = State::IDLE;
    physical_addr = 0;
    taddr          = 0;
    level          = 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 10.67)。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ています。

▼ リスト 10.67: (ptw.veryl)

```

case state {
    State::IDLE      : accept_request_comb();
    State::WALK_READY: assign_master      (taddr, 0, 0, 0);
    State::SET_AD     : assign_master      (taddr, 1, // wen = 1
    {1'b0 repeat MEMBUS_DATA_WIDTH - 8, wdata_ad}, // wdata
    {1'b0 repeat XLEN / 8 - 1, 1'b1} // wmask
    );
    State::EXECUTE_READY: assign_master(physical_addr, slave_saved.wen, slave_saved.wdata, slave_>
>saved.wmask);
    State::EXECUTE_VALID: if master.rvalid {
        accept_request_comb();
    }
    default: {}
}

```

`slave` への結果の割り当てで、ページフォルトが発生していた場合の結果を割り当てます (リスト 10.68)。

▼ リスト 10.68: (ptw.veryl)

```

State::PAGE_FAULT: {
    slave.rvalid      = 1;
}

```

```

slave.expt.valid      = 1;
slave.expt.page_fault = 1;
}

```

ページングが有効なときの要求を受け入れる動作を実装します (リスト 10.69)。仮想アドレスが有効かどうかでページフォルトを判定し、`taddr` レジスタには最初の 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 10.69: (ptw.veryl)

```

if paging_enabled {
    state = if is_valid_vaddr(slave.addr) ? State::WALK_READY : State::PAGE_FAULT;
    taddr = get_first_pte_address(ctrl.satp, slave.addr);
    level = LEVELS - 1;
} else {
    state      = if master.ready ? State::EXECUTE_VALID : State::EXECUTE_READY;
    physical_addr = slave.addr;
}

```

ページフォルトが発生したとき、状態を `State::IDLE` に戻します (リスト 10.70)。

▼ リスト 10.70: (ptw.veryl)

```
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 10.71)。

▼ リスト 10.71: (ptw.veryl)

```

State::SET_AD: if master.ready {
    state = State::EXECUTE_READY;
}

```

PTE と要求から、ページにアクセスする権限があるかどうかを確認する関数を定義します (リスト 10.72)。条件の詳細は「アドレス変換プロセス (Sv39)」(p.166) を確認してください。

▼ リスト 10.72: (ptw.veryl)

```

function check_permission (
    req: modport Membus::all_input,
) -> logic {
    let priv: PrivMode = if is_inst || !ctrl.mprv ? ctrl.priv : ctrl.mpp;

    // U-mode access with PTE.U=0
    let u_u0: logic = priv == PrivMode::U && !pte.u();
    // S-mode load/store with PTE.U=1 & sum=0
    let sd_u1: logic = !is_inst && priv == PrivMode::S && pte.u() && !ctrl.sum;
    // S-mode execute with PTE.U=1
    let si_u1: logic = is_inst && priv == PrivMode::S && pte.u();
}

```

```

// execute without PTE.X
let x: logic = is_inst && !pte.x();
// write without PTE.W
let w: logic = !is_inst && req.wen && !pte.w();
// read without PTE.R (MXR)
let r: logic = !is_inst && !req.wen && !pte.r() && !(pte.x() && ctrl.mxr);

return !(u_u0 | sd_u1 | si_u1 | x | w | r);
}

```

PTE をフェッチし、ページフォルトの判定、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 10.73）。

▼リスト 10.73: (ptw.veryl)

```

State::WALK_READY: if master.ready {
    state = State::WALK_VALID;
}
State::WALK_VALID: if master.rvalid {
    if !pte.is_valid(level) {
        state = State::PAGE_FAULT;
    } else {
        if pte.is_leaf() {
            if check_permission(slave_saved) {
                physical_addr = pte.get_physical_address(level, slave_saved.addr);
                if pte.need_update_ad(slave_saved.wen) {
                    state = State::SET_AD;
                    wdata_ad = pte.get_updated_ad(slave_saved.wen);
                } else {
                    state = State::EXECUTE_READY;
                }
            } else {
                state = State::PAGE_FAULT;
            }
        } else {
            // read next pte
            state = State::WALK_READY;
            taddr = pte.get_next_pte_addr(level, slave_saved.addr);
            level = level - 1;
        }
    }
}

```

これで Sv39 を ptw モジュールに実装できました。

10.7 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うため、何もしない命令として定義します。

10.7.1 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 10.74)。

▼ リスト 10.74: (inst_decoder.veryl)

```
bits == 32'h10200073 || //SRET
bits == 32'h10500073 || // WFI
f7 == 7'b0001001 && bits[11:7] == 0, // SFENCE.VMA
```

10.7.2 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 10.75)。

▼ リスト 10.75: (csrunit.veryl)

```
const MSTATUS_WMASK : UIntX = 'h0000_0000_007e_19aa as UIntX;
```

▼ リスト 10.76: (csrunit.veryl)

```
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 10.77、リスト 10.78、リスト 10.79)。

▼ リスト 10.77: (csrunit.veryl)

```
let is_sfence_vma: logic = ctrl.is_csr && ctrl.funct7 == 7'b0001001 && ctrl.funct3 == 0 && r>d_addr == 0;
```

▼ リスト 10.78: (csrunit.veryl)

```
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 10.79: (csrunit.veryl)

```
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);
let expt_cause: UIntX = switch {
  ...
  expt_tvm : CsrCause::ILLEGAL_INSTRUCTION,
  default : 0,
};
```

10.8 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。

10.8.1 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、変更が反映されていない状態で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10.80、リスト10.81、リスト10.82）。

▼リスト10.80: (csrunit.veryl)

```
flush      : output  logic          ,
minstret   : input   UInt64        ,
```

▼リスト10.81: (core.veryl)

```
flush      : csru_flush          ,
minstret   :                   ,
```

▼リスト10.82: (core.veryl)

```
var csru_trap_return: logic  ;
var csru_flush      : logic  ;
var minstret        : UInt64 ;
```

`flush`はsatpレジスタ、mstatusレジスタが変更されるときに1になります（リスト10.83）。

▼リスト10.83: (csrunit.veryl)

```
let wsc_flush: logic = is_wsc && (csr_addr == CsrAddr::SATP || csr_addr == CsrAddr::MSTATUS)>
>;
assign flush      = valid && wsc_flush;
```

coreモジュールで、制御ハザードが発生したことにします（リスト10.84）。

▼リスト10.84: (core.veryl)

```
assign control_hazard      = mems_valid && (csru_raise_trap || mems_ctrl.is_jump || memq>
>_rdata.br_taken || csru_flush);
assign control_hazard_pc_next = if csru_raise_trap ? csru_trap_vector : // trap
                                if csru_flush ? mems_pc + 4 : memq_rdata.jump_addr; // flush or jump
```

10.8.2 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の命令だった場合、命令のビット列はデータが書き換えられる前のものにな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ことを保証する命令です。

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 10.85、リスト 10.86）。

▼ リスト 10.85: (csrunit.veryl)

```
let is_fence_i: logic = inst_bits[6:0] == OP_MISC_MEM && ctrl.funct3 == 3'b001;
```

▼ リスト 10.86: (csrunit.veryl)

```
assign flush      = valid && (wsc_flush || is_fence_i);
```

第 11 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 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(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20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mond会員 (3期目)

最近はキヨーちゃん(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

本書は次の方々にレビューしていただきました。

TODO

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。

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/>

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 ミ一ミミ研究室