



RISC-V Shadow Stacks and Landing Pads (Zisslpcfi)

RISC-V Shadow-stack and Landing-pads Task Group

Version 0.1, 02/2023: This document is in development. Assume everything can change. See <http://riscv.org/spec-state> for details.

Table of Contents

Preamble.....	1
Copyright and license information.....	2
Contributors.....	3
1. Introduction.....	4
2. Shadow Stack and Landing Pad CSRs	8
2.1. Machine environment configuration registers (menvcfg and menvcfgh)	8
2.2. Hypervisor environment configuration registers (henvcfg and henvcfgh)	9
2.3. Machine status registers (mstatus)	9
2.4. Supervisor status registers (sstatus)	10
2.5. Virtual supervisor status registers (vsstatus)	11
2.6. Landing pad label (lpl)	11
2.7. Shadow stack pointer (ssp)	12
2.8. Machine Security Configuration (mseccfg)	12
3. Backward-edge control-flow integrity	14
3.1. Backward-edge CFI instruction encoding	14
3.2. Push to and Pop from the shadow stack	16
3.3. Read ssp into a register	19
3.4. Verifying return address.....	19
3.5. Atomic Swap from a shadow stack location	20
3.6. Shadow Stack Memory Protection	21
3.6.1. Virtual-Memory system extension for Shadow Stack	21
3.6.2. PMP extension for shadow stack.....	23
4. Forward-edge control-flow integrity.....	24
4.1. Forward-edge CFI Instruction encoding	25
4.2. Landing pad instruction	26
4.3. Label matching instructions	27
4.4. Setting up landing pad label register.....	28
4.5. Preserving expected landing pad state on traps.....	30
Bibliography.....	?

Preamble



This document is in the [Development state](#)

Assume everything can change. This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Copyright and license information

This specification is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). The full license text is available at creativecommons.org/licenses/by/4.0/.

Copyright 2022 by RISC-V International.

Contributors

This RISC-V specification has been contributed to directly or indirectly by (in alphabetical order):

Andrew Waterman, Antoine Linarès, Dean Liberty, Deepak Gupta, George Christou, Greg McGary, Henry Hsieh, Johan Klockars, Liu Zhiwei, Mark Hill, Nick Kossifidis, Tsukasa OI, Vedvyas Shanbhogue

Chapter 1. Introduction

Control-flow Integrity (CFI) provides CPU instruction set architecture (ISA) capabilities to defend against Return-Oriented Programming (ROP) and Call/Jump-Oriented Programming (COP/JOP) style control-flow subversion attacks. This attack methodology uses code sequences in authorized modules with at least one instruction in the sequence being a control transfer instruction that depends on attacker-controlled data either in the return stack or in a register/memory for the target address. Attackers stitch these sequences together by diverting the control flow instruction (e.g., RET, CALL, JMP) from its original target address to a new target via modification in the data stack or in the register or memory used by these instructions.

This specification describes CFI security objectives, threat model, and various architectural design choices to ensure that the design meets the security objectives.

RV32/RV64 provide two types of control transfer instructions - unconditional jumps and conditional branches. Conditional branches encode an offset in the immediate field of the instruction and are thus direct branches that are not susceptible to control flow subversion.

Unconditional direct jumps using JAL transfer control to a target that is in a +/- 1 MiB range from the current pc. Unconditional indirect jumps using the JALR obtain their branch target by adding the sign extended 12-bit immediate encoded in the instruction to the rs1 register.

The RV32I/RV64I does not have a dedicated instruction for calling a procedure or returning from a procedure. A JAL or JALR may be used to perform either a procedure call or a return from a procedure. The RISC-V ABI however defines the convention that a JAL/JALR where rd (i.e. the link register) is x1 or x5 is a procedure call, and a JAL/JALR where rs1 is the conventional link register (i.e. x1 or x5) is a return from procedure. The architecture allows for using these hints and conventions to support return address prediction and the hints are specified in Table 2.1 of the Unprivileged ISA specifications [1].

The RVC standard extension for compressed instructions provides unconditional jump and conditional branch instructions. The C.J and C.JAL instructions encode an offset in the immediate field of the instruction and thus are not susceptible to control flow subversion.

The C.JR and C.JALR RVC instruction performs a unconditional control transfer to the address in register rs1. The C.JALR additionally writes the address of the instruction following the jump (pc+2) to the link register x1 and is a procedure call. The C.JR where rs1 is the conventional link register (i.e. x1 or x5) is a return from procedure. A C.JR where rs1 is not the conventional link register is a indirect jump.

The RISC-V control-flow integrity (CFI) extension (Zisslpcfi) builds on these conventions and hints.

The term call is used to refer to a JAL or JALR instruction (and their compressed forms C.JAL and C.JALR) with x1 or x5 as the rd. A call using JAL and C.JAL is termed a direct call. A call using JALR and C.JALR is termed an indirect call.

The term return is used to refer to a JALR instructions (and its corresponding compressed form C.JR) with x1 or x5 as the rs1.

The term indirect jump is used to refer to an unconditional indirect jump using **JALR/C.JALR** instruction where the **rd** i.e. the link register is not **x1** or **x5** (i.e. not an indirect call) and where the **rs1** is not **x1** or **x5** (i.e. not a return).

The definition of **Indirect Jump** or **Call** is needed to determine if the control flow is doing an indirect forward edge as they need to be protected:

Listing 1. Indirect Forward Edge determination



```
if ( ( JALR && ( JALR.rd == x0 ) ) || C.JR )
    indirect_forward_edge = ( instr.rs1 != x1 ) || ( instr.rs1 != x5 )
    //these are indirect jump
else if ( JALR || C.JALR )
    indirect_forward_edge = ( ( instr.rs1 != x1 ) || ( instr.rs1 != x5
) ) ||
    //these are indirect jump
                                ( ( instr.rd == x1 ) || ( instr.rd = x5 )
)
    //these are indirect call
else
    indirect_forward_edge = 0
```

To enforce backward edge control flow integrity, the extension introduces a shadow stack. The shadow-stack is designed to provide integrity to control transfers performed using return instruction (where the return may be from a procedure invoked using an indirect call or a direct call) and this is referred to as backward-edge protection. A program using backward control flow integrity has two stack - a regular stack and a shadow stack - and the shadow stack is used exclusively to store the shadow copies of return addresses.

The shadow stack is used to spill the link register if required by non-leaf functions. A shadow-stack-pointer (**ssp**) register is introduced in the architecture to hold the address of the top of the current active shadow stack. The shadow stack is protected from inadvertent corruptions and modifications as detailed later. The extension provides instructions to store and load the link register to the shadow stack. A function in a program compiled to use shadow stacks stores the link register to the data stack and a shadow copy of the link register to the shadow stack when the function is entered (the prologue). When the function needs to return (the epilogue), the function loads the link register from the data stack and the shadow copy of the link register from the shadow stack. The link register value from the data stack and the shadow link register value from the shadow stack are compared. A mismatch of the two values is indicative of a subversion of the return address control variable and causes an illegal-instruction trap.



Operating in shadow stack mode, i.e., where the call stack layout is preserved and the shadow stack is used to store a shadow copy of the link register, allowing preserving the ABI.

A program may alternatively operate in control stack mode where the link register is only stored on the shadow stack. Such programs break the ABI but benefit from avoiding the additional instructions to store and load the link register to the data

stack and to compare the two before returning from a function. Control stack mode may also allow the program to have a smaller data stack as the space to save the link register is no longer needed.

To enforce forward edge control flow integrity, the extension introduces new landing pad instructions (**lpc11**) that enable software to indicate valid targets for indirect calls and jumps in a program. Compiler is expected to emit a **lpc11** as the first instruction of address-taken functions. Compiler is expected to emit a **lpc11** at an indirect jump target.

The landing-pads are designed to provide integrity to control transfers performed using indirect call and indirect jump and this is referred to as forward-edge protection.

When the landing pad feature is active, the hart tracks an expected landing pad (**ELP**) state that is updated with the expected landing pad instruction on indirect calls and jumps to . An indirect call or jump updates the **ELP** to require a **lpc11** instruction at the target. If the instruction at the target is not **lpc11** then an illegal instruction exception is raised.

The landing pads may be labeled. With labeling enabled, the number of landing pads that can be reached from an indirect call or indirect jump site can be constrained using programming language based policies. A landing pad label (**lp1**) is set up prior to initiating an indirect call or indirect jump with the expected landing pad label using an instruction to set the **lp1**. If the label of the landing pad does not match that in **lp1** then an illegal instruction exception is raised.

Up to 25-bit labels are supported by this extension.

In the simplest form the program may be built with a single label value to implement a coarse-grain version of forward-edge CFI. Such a program would significantly reduce the gadget space by constraining gadgets to be preceded by a landing pad instruction i.e., to the start of indirect callable functions.

A second form of label generation may generate a signature (e.g., a MAC) using the prototype of the functions. Such programs would further constrain the gadgets reachable from a call site to indirect callable functions that have the expected prototype of functions called by that call site.



A third form of label generation may generate labels by analyzing the control-flow-graph (CFG) of the program and lead to even further constraining of the gadgets reachable. Such programs may further use the multi-label capability such that when a function is called from two or more call sites, then such common functions may be labeled as reachable from each of the call sites. For example, consider two call sites A and B. A invokes functions X and Y. B invokes functions Y and Z. With a single label scheme the functions X, Y, and Z would need to be assigned the same label such that both call site A and B can invoke the common function Y. This allows call site A to additionally call function Z and call site B to additionally call function X. However, if function Y was labeled with two labels - one corresponding to call site A and other to call site B then Y can be invoked by both but the X can only be invoked by call site A and Z only by call site B. To support multiple labels, the compiler may create a call site specific entrypoint to such shared functions with each entrypoint containing the call site specific landing pad instruction

followed by a direct branch to the start of the function.

A portion of the label space may be dedicated to label landing pads that are only valid targets of an indirect jump (and not an indirect call).

Forward-edge and backward-edge CFI may be enabled for a program that executes in U-mode, S-mode, or M-mode by itself to enable a mix of CFI enabled applications, operating systems, and machine mode firmware to co-exist. The processor keeps track of the CFI enables and CFI state for each mode in the `mstatus` CSR. A subset of the fields in the `mstatus` CSR are accessible using the `sstatus` CSR. VS-mode's version of `sstatus` (`vsstatus`) tracks the CFI state for VS-mode and VU-mode.



To use Zisslpcfi, the operating system has to be modified to enable Zisslpcfi capabilities, including the context switching of additional CFI extension state. The set of programs installed in such an OS may however be a mix where some programs are compiled with Zisslpcfi capabilities and others are not. Allowing the U-mode CFI be individually enabled from S-mode allows an operating system to keep CFI enabled when operating in S-mode but enable or disable it for U-mode depending on the program being executed in U-mode.

To support backward compatibility of the programs built with Zisslpcfi support, the new instructions to operate on the shadow stack, the landing pad instructions, and the instructions to set the `lpl` are encoded using Zimop encodings. When Zisslpcfi is not enabled for a program or the program is executing on a processor that does not support the Zisslpcfi extension then the instructions introduced by the Zisslpcfi extensions execute as defined by Zimop extension.



An OS distribution compiled with Zisslpcfi extension typically also includes the system libraries (e.g., glibc, etc.) that are also compiled with the Zisslpcfi extension. Such system libraries however may need to link dynamically to programs that are not compiled with the Zisslpcfi extension. When such programs are executing, the OS may disable the Zisslpcfi extension in U-mode. When these system libraries are invoked in U-mode by such programs, the Zisslpcfi instructions in the libraries revert to their Zimop defined behavior. Without such encoding, the OS distribution may need to carry two versions of such libraries, one with Zisslpcfi instructions and one without, and thus need significantly larger cost and complexity for supporting the Zisslpcfi extension.

An OS distribution compiled with Zisslpcfi extension may be installed on a machine that does not support Zisslpcfi extensions. On such machines, as the Zisslpcfi instructions are encoded as Zimop, they revert to their Zimop defined behavior.

A program compiled with the Zisslpcfi extension may be installed on an OS that is not compiled for the Zisslpcfi extension or on a machine that does not support the Zisslpcfi extension. The Zisslpcfi instructions are encoded as Zimop revert back to their Zimop defined behavior.

The Zisslpcfi extension depends on the Zicsr extension.

Chapter 2. Shadow Stack and Landing Pad CSRs

This chapter specifies the CSR state of the Zisslpcfi extension.

2.1. Machine environment configuration registers (menvcfg and menvcfgh)

63	62	61	60	59	58	56
STCE	PBMTE	HADE	CFIE	SFCFIE	WPRI	
55						48
				WPRI		
47						40
				WPRI		
39						32
				WPRI		
31						24
				WPRI		
23						16
				WPRI		
15						8
				WPRI		
7	6	5	4	3	1	0
CBZE	CBCFE	CBIE			WPRI	FIOM

Figure 1. Machine environment configuration register (menvcfg) for MXLEN=64

The **CFIE** (bit 60) field controls if Zisslpcfi extension is available for use in modes less privileged than M. When **CFIE** is 1, the **SFCFIE** (bit 59) field enables forward-edge CFI at S-mode.

When **menvcfg.CFIE** bit is 0, then at privilege modes less privileged than M:

- Attempts to access **ssp** or **lp1** CSR raise an illegal instruction exception.
- Zisslpcfi extension instructions revert to the Zimop defined behavior.
- The **UBCFIE**, **UFCFIE** and **SPELP** fields in **sstatus** are read-only zero.
- The **CFIE** field in **henvcfg** is read-only zero.
- The **pte.xwr=010b** encoding in S-stage page tables is reserved.



When the Zisslpcfi extension is available for use at privilege level less than M, the operating system may use the **UBCFIE** and **UFCFIE** to selectively enable the backward-edge and forward-edge CFI at U mode per application.

When the Zisslpcfi extension is available for use at S-mode, the operating system may use shadow stacks at S-mode. If the operating system uses forward-edge CFI then it may request the SEE to set **SFCFIE** to 1.

With these set of controls the backward-edge and forward-edge CFI may be separately enforced at S-mode and for each application.

2.2. Hypervisor environment configuration registers (**henvcfg** and **henvcfgh**)

63	62	61	60	59	58	56
STCE	PBMTE	HADE	CFIE	SFCFIE		WPRI
55						48
				WPRI		
47						40
				WPRI		
39						32
				WPRI		
31						24
				WPRI		
23						16
				WPRI		
15						8
				WPRI		
7	6	5	4	3	1	0
CBZE	CBCFE		CBIE		WPRI	FIOM

Figure 2. Hypervisor environment configuration register (**henvcfg**) for *MXLEN=64*

The **CFIE** (bit 60) bit controls if Zisslpcfi extension is available for use in VS and VU modes. When **menvcfg.CFIE** is 0, **henvcfg.CFIE** is read-only zero.

When **henvcfg.CFIE** bit is 0, then at privilege modes VS and VU:

- Attempts to access **ssp** or **lp1** CSR raise an illegal instruction exception.
- Zisslpcfi extension instructions revert to the Zimop defined behavior.
- The **UBCFIE**, **UFCFIE** and **SPELP** fields in **sstatus** (really **vsstatus**) are read-only zero.
- The **pte.xwr=010b** encoding in VS-stage page tables remains reserved.

When **henvcfg.CFIE** is 1, the **henvcfg.SFCFIE** (bit 59) field enables forward-edge CFI at VS-mode.

2.3. Machine status registers (**mstatus**)

63	62						56
SD				WPRI			
55							48
				WPRI			
47							40
				WPRI			
39	38	37	36	35	34	33	32
	WPRI	MBE	SBE	SXL[1:0]		UXL[1:0]	
31				27	26	25	24
		WPRI			MPELP	SPELP	UBCFIE
23	22	21	20	19	18	17	16
UFCFIE	TSR	TW	TVM	MXR	SUM	MPRV	XS[1:0]
15	14	13	12	11	10	9	8
XS[1:0]		FS[1:0]		MPP[1:0]		VS[1:0]	SPP
7	6	5	4	3	2	1	0
MPIE	UBE	SPIE	WPRI	MIE	WPRI	SIE	WPRI

Figure 3. Machine-mode status register (*mstatus*) for RV64

The **UFCFIE** (bit 23) and **UBCFIE** (bit 24) are WARL fields that when set to 1 enable forward-edge and backward-edge CFI respectively at U-mode.

The **SPELP** (bit 25) and **MPELP** (bit 26) WARL fields are updated when a trap is taken into S-mode or M-mode respectively. When a trap is taken into privilege mode **x**, the **xPELP** fields are updated to indicate that a landing pad was expected at the privilege level **xPP** at the time of taking the trap.

The **xPELP** fields are encoded as follows:

- 0 - **NO_LP_EXPECTED** - no landing pad instruction expected
- 1 - **LP_EXPECTED** - landing pad instruction expected

2.4. Supervisor status registers (*sstatus*)

63	62						56
SD				WPRI			
55							48
				WPRI			
47							40
				WPRI			
39					34	33	32
			WPRI			UXL[1:0]	
31					26	25	24
			WPRI			SPELP	UBCFIE
23	22		20	19	18	17	16
UFCFIE		WPRI		MXR	SUM	WPRI	XS[1:0]
15	14	13	12	11	10	9	8
XS[1:0]		FS[1:0]		WPRI		VS[1:0]	SPP
7	6	5	4		2	1	0
WPRI	UBE	SPIE		WPRI		SIE	WPRI

Figure 4. Supervisor-mode status register (*sstatus*) when **SXLEN=64**

When **menvcfg.CFIE** is 1, access to the following fields accesses the homonymous field of *mstatus* register. When **menvcfg.CFIE** is 0, these fields are read-only zero.

- **UFCFIE** (bit 23)
- **UBCFIE** (bit 24)
- **SPELP** (bit 25)

2.5. Virtual supervisor status registers (vsstatus)

63	62						56
SD				WPRI			
55							48
				WPRI			
47							40
				WPRI			
39					34	33	32
			WPRI			UXL[1:0]	
31					26	25	24
			WPRI			SPELP	UBCFIE
23	22		20	19	18	17	16
UFCFIE		WPRI		MXR	SUM	WPRI	XS[1:0]
15	14	13	12	11	10	9	8
XS[1:0]		FS[1:0]		WPRI		VS[1:0]	SPP
7	6	5	4		2	1	0
WPRI	UBE	SPIE		WPRI		SIE	WPRI

Figure 5. Virtual supervisor status register (**vsstatus**) when **VSXLEN=64**

The `vsstatus` register is VS-mode's version of `sstatus` and the Zisslpcfi extension introduces the following fields.

- **UFCFIE** (bit 23)
- **UBCFIE** (bit 24)
- **SPELP** (bit 25)

When `menvcfg.CFIE` is 0, these fields are read-only zero. When `menvcfg.CFIE` is 1 and `henvcfg.CFIE` is 0, these fields are read-only zero in `sstatus` (really `vsstatus`) when `V=1`.



The `vsstatus` and `henvcfg` CSR for a virtual machine may be restored in any order. The state of `henvcfg.CFIE` does not prevent access to the bits introduced in `vsstatus` when the CSR is accessed in HS-mode.

2.6. Landing pad label (lpl)

The **lpl** CSR is a supervisor read-write (SRW) 32-bit register that holds the label expected at the target of an indirect call or an indirect jump. The label is split into a 8-bit upper label (**UL**), 8-bit middle label (**ML**), and a 9-bit lower label (**LL**).

Figure 6. `lpl` for RV32 and RV64

When `menvcfg.CFIE` is 0, an attempt to access `lpl` in a mode other than M-mode raises an illegal instruction exception.



Access to `lpl` at S-mode is not dependent on `sstatus.UFCFIE` or `menvcfg.SFCFIE` to allow an operating system to be able to context switch U-mode `lpl` state even when the operating system itself does not enable the use of forward-edge CFI at S-mode.

When `menvcfg.CFIE` is 1 but `henvcfg.CFIE` is 0, an attempt to access `lpl` when `V=1` raises a virtual instruction exception.

2.7. Shadow stack pointer (`ssp`)

The `ssp` CSR is an unprivileged read-write (URW) CSR that reads and writes `XLEN` low order bits of the shadow stack pointer (`ssp`). There is no high CSR defined as the `ssp` is always as wide as the `XLEN` of the current privilege level.

When `menvcfg.CFIE` is 0, an attempt to access `ssp` in a mode other than M-mode raises an illegal instruction exception. When `sstatus.UBCFIE` is 0, an attempt to access `ssp` in U-mode raises an illegal instruction exception.



Access to `ssp` at S-mode is not dependent on `sstatus.UBCFIE` to allow an operating system to be able to context switch U-mode `ssp` per application.

When `menvcfg.CFIE` is 1 but `henvcfg.CFIE` is 0, an attempt to access `ssp` when `V=1` raises a virtual instruction exception.

When `menvcfg.CFIE` and `henvcfg.CFIE` are both 1 but `vsstatus.UBCFIE` is 0, an attempt to access `ssp` in VU-mode raises an illegal instruction exception.

2.8. Machine Security Configuration (`mseccfg`)

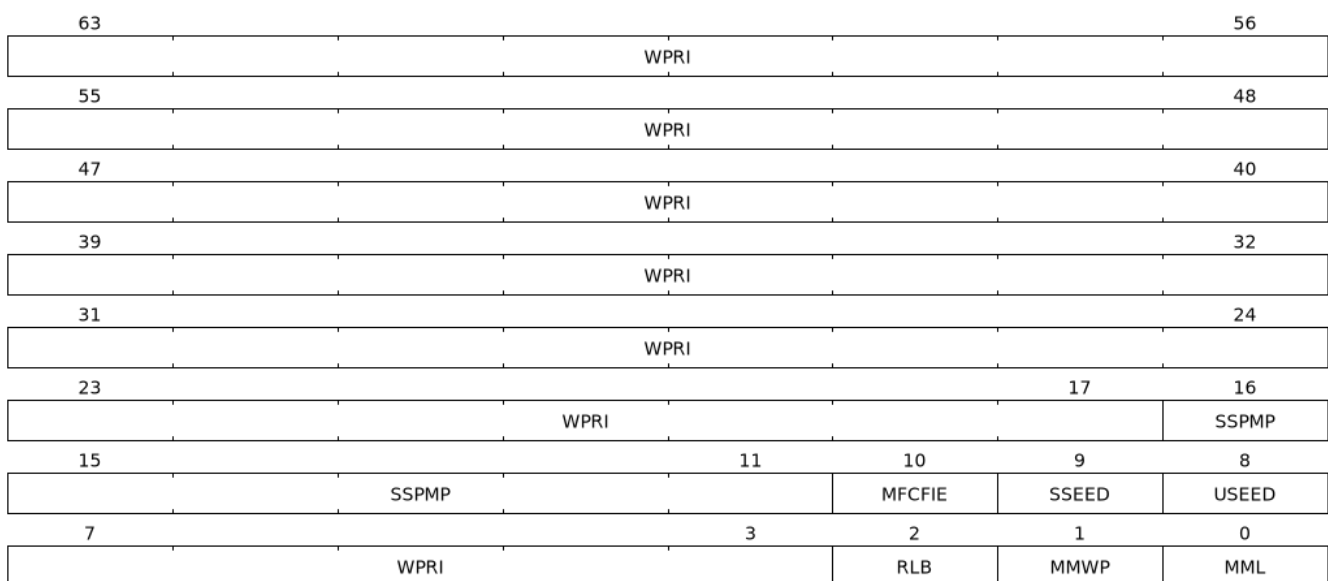


Figure 7. Machine security configuration register (`mseccfg`) when `MXLEN=64`

When the `Smeppmp` extension is implemented, a new WARL field `sspmp` is defined in bits 16:11 of the

`mseccfg` CSR to identify a PMP entry as the shadow stack memory region for M-mode accesses. The rules enforced by PMP for M-mode shadow stack memory accesses are outlined in [Section 3.6.2](#).

The `MFCFIE` (bit 10) is a WARL field that when set to 1 enables forward-edge CFI at M-mode.

Chapter 3. Backward-edge control-flow integrity

A shadow stack is a second stack used to push the link register if it needs to be spilled to make a new procedure call. Leaf functions do not need to spill the link register.

The shadow stack, similar to the regular stack, grows downwards, i.e. from higher addresses to lower addresses. Each entry on the shadow stack is **XLEN** wide and holds the link register value. The **ssp** points to the top of the shadow stack, i.e. address of the last element pushed on the shadow stack.



Compilers when generating code for a CFI enabled program must protect the link register, e.g. **x1** and/or **x5**, from arbitrary modification by not emitting unsafe code sequences.

3.1. Backward-edge CFI instruction encoding

The backward-edge CFI extension introduces the following instructions for shadow stack operations. All instructions are encoded using the SYSTEM major opcode and using the **mop.r** and **mop.rr** instructions defined by the Zimop extension.

(mop.r)	31	20 19	15 14	12 11	7 6	0
mnemonic	1.00..0111..	rs1	func3	rd	opcode	
sspop	100000011100	00000	100	dst	1110011	
sspr	100000011101	00000	100	dst	1110011	
	12	5	3	5	7	

Encoding **rd** as **x0** is not supported for **sspop** and **sspr**.

(mop.rr)	31	25 24	20 19	15 14	12 11	7 6	0
mnemonic	1.00..1	rs2	rs1	func3	rd	opcode	
ssamoswap	1000001	src	addr	100	dst	1110011	
sschkra	1000101	00001	00101	100	0	1110011	
sspush	1000101	src	00000	100	0	1110011	
	7	5	5	3	5	7	

Encoding **rd** as **x0** is not supported for **ssamoswap**.

When **menvcfg.CFIE** is 0, then Zisslpcfi is not enabled for privilege modes less than M and backward-edge CFI is not enabled at privilege levels less than M.

When **V=0** and **menvcfg.CFIE** is 1, then backward-edge CFI is enabled at S-mode. When **V=0** and **menvcfg.CFIE** is 1, then backward-edge CFI is enabled at U-mode if **mstatus.UBCFIE** is 1.

When **henvcfg.CFIE** is 0, then Zisslpcfi is not enabled for use when **V=1**.

When `V=1` and `menvcfg.CFIE` and `henvcfg.CFIE` are both 1, then backward-edge CFI is enabled in VS-mode. When `V=1` and `menvcfg.CFIE` and `henvcfg.CFIE` are both 1, then backward-edge CFI is enabled in VU-mode if `vsstatus.UBCFIE` is 1.

The term `xBCFIE` is used to determine if backward-edge CFI is enabled at a privilege level `x` and is defined as follows:

Listing 2. `xBCFIE` determination

```
if ( privilege == M-mode )
    xBCFIE = 1
else if ( menvcfg.CFIE == 1 && V == 0 && privilege == S-mode )
    xBCFIE = 1
else if ( menvcfg.CFIE == 1 && V == 0 && privilege == U-mode )
    xBCFIE = mstatus.UBCFIE
else if ( menvcfg.CFIE == 1 && henvcfg.CFIE == 1 && V == 1 && privilege == S-mode )
    xBCFIE = 1
else if ( menvcfg.CFIE == 1 && henvcfg.CFIE == 1 && V == 1 && privilege == U-mode )
    xBCFIE = vsstatus.UBCFIE
else
    xBCFIE = 0
```

When backward-edge CFI is not enabled(`xBCFIE = 0`):

- The instructions defined for backward-edge CFI revert to their Zimop defined behavior and write 0 to [rd].



The use of shadow stacks at U-mode must be explicitly enabled per application. Explicit enable for user mode applications allows such an application to invoke shared libraries that may have shadow stack instructions even when the application itself has backward-edge CFI not enable. The shadow stack instructions invoked in the context of this application revert to their Zimop defined behavior.

When `Zisslpcfi` is enabled, the use of backward-edge CFI is always enabled for use at S-mode. However, it is benign to use an operating system that has not been compiled with shadow stack instructions. Such an operating system that does not use backward-edge CFI for S-mode execution may still enable the backward-edge CFI use by U-mode.

When `Zisslpcfi` is implemented, the use of backward-edge CFI is always enabled at M-mode. However, it is benign to use M-mode firmware that has not been compiled with shadow stack instructions.



When programs using shadow stack instructions are installed on a machine that supports the CFI extensions but the operating system installed does not enable the CFI extensions, the program continues to function due to Zimop defined behavior of writing 0 to [rd] and not causing an illegal-instruction exception.

When programs using shadow stack instructions are installed on a machine that

does not support the CFI extensions but support the Zimop extension, the program continues to function due to Zimop defined behavior of writing 0 to [rd] and not causing an illegal-instruction exception.


On machines that do not support Zimop extension, the instructions cause an illegal-instruction exception. Installing programs that use the shadow stack instructions on such machines is not supported.

3.2. Push to and Pop from the shadow stack

A push operation is defined as decrement of the `ssp` by `XLEN` followed by a write of the link register at the new top of the shadow stack. A pop operation is defined as a `XLEN` wide read from the current top of the shadow stack followed by an increment of the `ssp`.

To push a link register on the shadow stack, the CFI extension provides `sspush` instruction. To pop a link register from the shadow stack, the CFI extension provides `sspop` instruction.

Programs operating in shadow stack mode store the return address to the data stack as well as the shadow stack in the function prologue. Such programs when returning from the function load the link register from the data stack and load a shadow copy of the link register from the shadow stack. The two values are then compared (using the `sschkra` instruction that is discussed later). If the values do not match it is indicative of a shadow stack violation and causes an illegal-instruction exception. The function prologue and epilog of a function using shadow stacks is as follows:



```
function_entry:
    sspush x1      # push link register x1 on shadow stack
    :
    :
    sspop x5       # pop from shadow stack into x5
    sschkra x5, x1 # compare link register x1 to shadow
                  # link register x5; faults if not same
    ret
```

When `x1` is used by the ABI as the link register, the `x5` may be used to load the shadow link register value from the shadow stack. Alternatively, if the ABI uses `x5` as the link register, then the `x1` register may be used as the shadow link register.

A leaf function i.e. a function that does not itself make function calls does not need to push the link register to the shadow stack or pop it from the shadow stack. The return value may be held in the link register itself for the duration of the function execution.

In the RVI psABI, the `x1` register is designated as the link register and `x5` register is designated as a temporary register. Since the shadow link register is loaded at the tail of the function, prior to return, the `x5` register being used as the shadow link register does not impose a burden on the compiler as the `x5` register, being a

temporary register that is not preserved across a call, is usually free for use at the tail of the function.

Programs operating in the control stack mode may store the return address only to the shadow stack in the function prologue. Such functions when returning from the function load the link register value from the shadow stack. Such programs may either use the x1 or x5 register, depending on their ABI, as the link register. The hardware return-address prediction stacks detect the use of x1/x5 as the rd and x1/x5 as the source for a JALR instruction to infer if the JALR is used for a call or a return semantic for the purposes of prediction. To support both options the CFI extension provides the x1 and x5 variants of the shadow stack load and store.

`sspop` performs a load identically to the existing `LOAD` instruction with the difference that the base is implicitly `ssp`, the width is implicitly `XLEN`. `sspush` performs a store identically to the existing `STORE` instruction with the difference that the base is implicitly `ssp`, the width is implicitly `XLEN`.

The `sspush` and `sspop` require the virtual address in `ssp` to have a shadow stack attribute (see Shadow Stack Memory Protection).

If the virtual address in `ssp` is not `XLEN` aligned then the instructions cause a load or store/AMO address-misaligned exception.



Misaligned accesses to shadow stack are not required and enforcing alignment is more secure to detect errors in the program.

The operation of the `sspush` instructions is as follows:

Listing 3. `sspush` operation

```
If (xBCFIE = 1)
    [ssp] = [ssp] - (XLEN/8)    # decrement ssp by XLEN/8
    *[ssp] = [src]             # Store src value to address in ssp
else
    [dst] = 0
endif
```

The operation of the `sspop` instructions is as follows:

Listing 4. `sspop` operation

```
If (xBCFIE = 1)
    dst = *[ssp]               # Load dst from address in ssp
    [ssp] = [ssp] + (XLEN/8)    # increment ssp by XLEN/8
else
    [dst] = 0;
endif
```



Store to load forwarding is a common technique employed by high performance processor implementations. CFI implementations may restrict forwarding from a

non-shadow-stack store to a `sspop` instruction. A non-shadow-stack store causes a fault if done to a page mapped as a shadow stack. However such determination may be delayed till the PTE has been examined and thus may be used to transiently forward the data from such stores to a `sspop`.

A common operation performed on stacks is to unwind them to support constructs like `setjmp/longjmp`, C++ exception handling, etc. A program that uses shadow stacks must unwind the shadow stack in addition to the stack used to store data. The unwind function must verify that it does not accidentally unwind past the bounds of the shadow stack. Shadow stacks are expected to be bounded on each end using guard pages i.e. pages that do not have a shadow stack attribute. To detect if the unwind occurs past the bounds of the shadow stack the unwind may be done in maximal increments of 4 KiB and testing for the `ssp` to be still pointing to a shadow stack page or has unwound into the guard page. The following examples illustrate the use of backward-edge CFI instructions to unwind a shadow stack.



```
setjmp() {
    :
    :
    // read and save top of stack pointer to jmp_buf
    asm("sspr %0" : "=r"(cur_esp));
    jmp_buf->sav_esp = cur_esp;
    :
    :
}
longjmp() {
    :
    // Read current shadow stack pointer and
    // compute number of call frames to unwind
    asm("sspr %0" : "=r"(cur_esp));
    // Skip the unwind if backward-edge CFI not enabled
    asm("beqz %0, back_cfi_not_enabled" : "=r"(cur_esp));
    num_unwind = jmp_buf->sav_esp - cur_esp;
    // Unwind the frames in a loop
    while ( num_unwind > 0 ) {
        step = ( num_unwind >= 4096 ) ? 4096 : num_unwind;
        cur_esp += step;
        num_unwind -= step;
        // write the esp register with unwound value
        asm("csrw %0, $esp_csr_num" : "=r"(cur_esp));
        // Test if unwound past the shadow stack bounds
        asm("sspush x5");
        asm("sspop x5");
    }
    back_cfi_not_enabled:
    :
}
```

3.3. Read **ssp** into a register

The **sspr** instruction is provided to move the contents of **ssp** to the destination register.

The operation of the **sspr** instructions is as follows:

*Listing 5. **sspr** operation*

```
If (xBCFIE = 1)
    [dst] = [ssp]
else
    [dst] = 0;
endif
```

The property of Zimop writing 0 to the rd when the extension using Zimop is not present or not enabled may be used by such functions to skip over unwind actions by dynamically detecting if the backward-edge CFI extension is enabled.

An example sequence such as the following may be used:



```
sspr t0                # mv ssp to t0
beqz bcfi_not_enabled  # zero is not a valid shadow stack
                        # pointer by convention

# Shadow stacks enabled
:
:
bcfi_not_enabled:
```

3.4. Verifying return address

Programs operating with a shadow stack push the return address onto the data stack as well as the shadow stack in the function prologue. Such programs when returning from the function pop the link register from the data stack and pop a shadow copy of the link register from the shadow stack. The two values are then compared. If the values do not match it is indicative of a corruption of the return address variable and the program causes an illegal instruction exception.

When x1 is used by the ABI as the link register, the x5 may be used to hold the shadow link register value from the shadow stack. Alternatively, if the ABI uses x5 as the link register, then the x1 register may be used as the shadow link register.

A **sschkra** instruction is provided to perform the comparison.

The operation of the **sschkra** instruction is as follows:

*Listing 6. **sschkra** operation*

```
If (xBCFIE = 1)
    if [x1] != [x5]
```

```

        Raise illegal-instruction exception
    endif
else
    [dst] = 0;
endif

```

3.5. Atomic Swap from a shadow stack location

The CFI extension defines an `ssamoswap` instruction to atomically swap the `XLEN` bits of `src` register with `XLEN` bits on the shadow stack at address in `addr` and store the value from address in `src` into register `dst`.

The `ssamoswap` is always sequentially consistent and cannot be reordered with earlier or later memory operations from the same hart.

The `ssamoswap` requires the virtual address in `addr` to have a shadow stack attribute (see Shadow Stack Memory Protection).

If the virtual address is not `XLEN` aligned then the instructions cause a store/AMO address-misaligned exception.

The operation of the `ssamoswap` instructions is as follows:

Listing 7. ssamoswap operation

```

If (xBCFIE = 1)
    Perform the following atomically with sequential consistency
        [dst] = *[addr]
        *[addr] = [src]
else
    [dst] = 0;
endif

```



Stack switching is a common operation in user programs as well as supervisor programs. When a stack switch is performed the stack pointer of the currently active stack is saved into a context data structure and the new stack is made active by loading a new stack pointer from a context data structure.

When shadow stacks are enabled for a program, the program needs to additionally switch the shadow stack pointer. The pointer to the top of the deactivated shadow stack if held in a context data structure may be susceptible to memory corruption vulnerabilities. To protect the pointer value the program may then store it at the top of the shadow stack itself and thus create a checkpoint.

An example sequence to store and restore the shadow stack pointer is as follows:

```

# The a0 register holds the pointer to top of new shadow
# to switch to. The current ssp is first pushed on the current

```

```

# shadow stack and the ssp is restored from new shadow stack
save_shadow_stack_pointer:
    sspr    x5                # read ssp and push value onto
    sspush  x5                # shadow stack. The [ssp] now
    addi    x5, x5, -(XLEN/8) # holds ptr+XLEN/8. The [x5] now
                                # holds ptr. Save away x5
                                # into a context structure to
                                # restore later.

restore_shadow_stack_pointer:
    ssamoswap t0, x0, (a0)    # t0=[a0] and *[a0]=0
                                # The [a0] should hold ptr'
                                # The [t0] should hold ptr'+XLEN/8

    addi    a0, a0, (XLEN/8)  # a0+XLEN/8 must match to t0
    bne     t0, a0, crash     # if not crash program
    csrw    ssp, t0           # setup new ssp

```

Further the program may enforce an invariant that a shadow stack can be active only on one hart by using the `ssamoswap` when performing the restore from the checkpoint such that the checkpointed data is zeroed as part of the restore sequence and multiple hart attempt to restore the checkpointed data only one of them succeeds.

3.6. Shadow Stack Memory Protection

To protect shadow stack memory the memory is associated with a new page type - Shadow Stack (SS) page - in the page tables.

When the `Smepp` extension is supported the PMP configuration registers are enhanced to support a shadow stack memory region for use by M-mode.

3.6.1. Virtual-Memory system extension for Shadow Stack

The shadow stack memory is protected using page table attributes such that it cannot be stored to by instructions other than `sspush` and `ssamoswap`. The `sspop` instruction can only load from shadow stack memory.

The shadow stack can be read using all instructions that load from memory.

Attempting to fetch an instruction from a shadow stack page raises a fetch page-fault exception.

The encoding `R=0`, `W=1`, and `X=0`, is defined to mean a shadow stack page. When `menvcfg.CFIE=0`, this encoding continues to be reserved. When `V=1` and `henvcfg.CFIE=0`, this encoding continues to be reserved at `VS` and `VU`.

The following faults may occur:

1. If the accessed page is a shadow stack page
 - a. Stores other than `sspush` and `ssamoswap` cause write/AMO access faults.

- b. Instructions fetch causes a page fault
- 2. if the accessed page is not a shadow stack page
 - a. `ssamoswap` and `sspush` cause a store/AMO access fault
 - b. `sspop` causes a load access fault

To support these rules, the virtual address translation process specified in section 4.3.2 of the Privileged Specification [2] is modified as follows:

3. If `pte.v = 0` or if any bits of encodings that are reserved for future standard use are set within `pte`, stop and raise a page-fault exception corresponding to the original access type. The encoding `pte.xwr = 010b` is not reserved if `menvcfg.CFIE` is 1 or if `V=1` and `henvcfg.CFIE` is 1.
4. Otherwise, the PTE is valid. If `pte.r = 1` or `pte.w = 1` or `pte.x = 1`, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let `i = i - 1`. If `i < 0`, store and raise a page-fault exception corresponding to the original access type. Otherwise, let `a = pte.ppn x PAGE_SIZE` and go to step 2.
5. A leaf PTE has been found. If the memory access is by a shadow stack instruction and `pte.xwr != 010b` then cause an access-violation exception corresponding to the access type. If the memory access is a store/AMO and `pte.xwr == 010b` then cause a store/AMO access-violation. If the requested memory access is not allowed by the `pte.r`, `pte.w`, `pte.x`, and `pte.u` bits, given the current privilege mode and the value of the `SUM` and `MXR` fields of the `mstatus` register, stop and raise a page-fault exception corresponding to the original access type.

The `U` and `SUM` bit enforcement is performed normally for shadow stack instruction initiated memory accesses. The state of the `MXR` bit does not affect read access to a shadow stack page as the shadow stack page is always readable by all instructions that load from memory.

Svpbmt extension and Svnop extensions are supported for shadow stack pages.



Operating systems should protect against writeable non-shadow-stack alias virtual-addresses mappings being created to the shadow stack physical memory.

The G-stage address translation and protections are not affected by the shadow stack extension. When G-stage page tables are active, the `ssamoswap`, and `sspop` instructions require the G-stage page table mapping the accessed memory to have read permission and the `ssamoswap` and `sspush` instructions require write permission. The `xwr == 010b` encoding in the G-stage PTE remains reserved.



A future extension may define shadow stack encoding the G-stage page table to support use cases such as a hypervisor enforcing shadow stack protections for virtual-supervisor.



All instructions that load from memory are allowed to read the shadow stack. The shadow stack only holds a copy of the link register as saved on the regular stack. The ability to read the shadow stack is useful for debug, performance profiling, and other use cases.

3.6.2. PMP extension for shadow stack

When privilege mode is less than M, the PMP region accessed by `ssp` and `ssw` must provide write permission and the PMP region accessed by `ssr` must provide read permission.

The M-mode memory accesses by `ssp` and `ssw` instructions test for write permission in the matching PMP entry when permission checking is required.

The M-mode memory accesses by `ssr` instruction tests for read permission in the matching PMP entry when permission checking is required.

When the `Smpmp` extension is implemented, a new WARL field `ssmp` is defined in the `msecfg` CSR to configure a PMP entry as the shadow stack memory region for M-mode accesses.

When `msecfg.MML` is 1, the `ssmp` field is read-only else it may be written.

When `ssmp` field is implemented and `msecfg.MML` is 1 the following rules are additionally enforced for M-mode memory accesses:

- `ssp`, `ssr`, and `ssw` instructions must match PMP entry `ssmp`.
- Write by instructions other than `ssp` and `ssw` that match PMP entry `ssmp` cause an access violation exception.



The PMP region used for the M-mode shadow stack is expected to be made inaccessible for U-mode and S-mode read and write accesses. Allowing write access violates the integrity of the shadow stack and allowing read access may lead to disclosure of M-mode return addresses.

Chapter 4. Forward-edge control-flow integrity

The forward-edge CFI introduces landing pad instructions that enable software to indicate valid targets for indirect calls and indirect jumps in a program.

A landing pad (**lpc11**) instruction is defined as the instruction that must be placed at the program locations that can be valid targets of indirect jumps or calls.

To enforce that the target of an indirect call or indirect jump must be a valid landing pad instruction, the hart maintains an expected landing pad (**ELP**) state to determine if a landing pad instruction is required at the target of a **JALR** instruction. The **ELP** state can be one of:

- 0 - **NO_LP_EXPECTED**
- 1 - **LP_EXPECTED**

A **JALR** with **rd = x1** or **rd = x5** is an indirect call; it updates **ELP** to **LP_EXPECTED**. A **JALR** with **rd != x1** and **rd != x5** and where **rs1 != x1** and **rs1 != x5** is an indirect jump; it updates the **ELP** to **LP_EXPECTED**. A **C.JALR** is an indirect call; it updates **ELP** to **LP_EXPECTED**. A **C.JR** where **rs1 != x1** and **rs1 != x5** is an indirect jump; it updates **ELP** to **LP_EXPECTED**.

When **ELP** is set to **LP_EXPECTED** and the next instruction in the instruction stream is not 4-byte aligned, or is not a **lpc11**, or if the label encoded in the **lpc11** does not match the lower label in **lp1** register then an illegal instruction exception is raised. If the next instruction in the instruction stream is 4-byte aligned and is a **lpc11** with its label matching the lower label in **lp1** register then the **ELP** updates to **NO_LP_EXPECTED**.



Tracking of **ELP** and requiring valid landing pad instructions at the target of indirect call and jump enables a processor implementation to significantly reduce or to prevent speculation to non-landing-pad instructions. Constraining speculation using this technique greatly reduces the gadget space and increases the difficulty of using techniques such as branch-target-injection, also known as Spectre variant 2, that use speculative execution to leak data through side channels.

The forward-edge CFI also supports labeling of landing pads. By itself a landing pad allows, for example, an indirect call to land on any **lpc11** in the program. This significantly reduces the number of valid targets for an indirect call. Labeling of the landing pads enables software to achieve greater precision in pairing up indirect call sites or indirect jump sites with valid targets. To support labeled landing pads, the indirect call/jump sites establish an expected landing pad label in the landing pad label (**lp1**). If the target of the indirect call/jump is a valid landing pad instruction, the expected label established in the **lp1** is matched with the target's label. If a mismatch is detected then the label check instruction causes an illegal instruction exception.

Each landing pad may be labeled with a label that may be up to 25-bits wide. The **lp1** has three subfields - a 9-bit lower label (**LL**), a 8-bit middle label (**ML**), and an 8-bit upper label (**UL**).

The forward-edge CFI provides a `lpsll` instruction to establish the expected `LL` in the `lpl`, a `lpsml` instruction to establish the `ML`, and a `lpsul` instruction to establish the `UL` in the `lpl`.

The `lpcll` instructions embed a 9-bit immediate field. The instruction compares this value to the `LL` and on a mismatch causes an illegal-instruction exception.

For label widths up to 17-bit a companion instruction `lpcml` is provided. The `lpcml` embeds a 8-bit immediate value that is compared to the `ML` and on a mismatch causes an illegal-instruction exception.

For label widths greater than 17-bit a second companion instruction `lpcul` is provided. The `lpcul` embeds a 8-bit immediate value that is compared to the `UL` and on a mismatch causes an illegal-instruction exception.

4.1. Forward-edge CFI Instruction encoding

The forward-edge CFI introduces the following instructions for landing pad operations. All instructions are encoded using the SYSTEM major opcode and using the `mop.rr` instructions defined by the Zimop extension.

(mop.rr)	31	25	24	23	22	15	14	12	11	7	6	0
mnemonic	1.00..1	t	t	t	t	s	s	s	s	s	s	s
<code>lpsll</code>	1000001	0				LLPL		100		0		1110011
<code>lpcll</code>	1000001	1				LLPL		100		0		1110011
<code>lpsml</code>	1000011	0	0			MLPL		100		0		1110011
<code>lpcml</code>	1000011	0	1			MLPL		100		0		1110011
<code>lpsul</code>	1000011	1	0			ULPL		100		0		1110011
<code>lpcul</code>	1000011	1	1			ULPL		100		0		1110011
	7	1	1			8		3		5		7

When privilege level is M, the forward-edge CFI is active when `MFCFIE` is 1 in `mseccfg` register.

When `menvcfg.CFIE` is 0, then `Zisslpcfi` is not enabled for privilege modes less than M and forward-edge CFI is not active at privilege levels less than M.

When `V=0` and `menvcfg.CFIE` is 1, then forward-edge CFI is active at S-mode if `menvcfg.SFCFIE` is 1 and is active at U-mode if `mstatus.UFCFIE` is 1.

When `henvcfg.CFIE` is 0, then `Zisslpcfi` is not enabled for use when `V=1`.

When `V=1` and `menvcfg.CFIE` and `henvcfg.CFIE` are both 1, then forward-edge CFI is active at S-mode if `henvcfg.SFCFIE` is 1 and is active at U-mode if `vsstatus.UFCFIE` is 1.

The term `xFCFIE` is used to determine if forward-edge CFI is active at privilege level `x` and is defined as follows:

Listing 8. xFCFIE determination

```
if ( privilege == M-mode )
    xFCFIE = mseccfg.MFCFIE
else if ( menvcfg.CFIE == 1 && V == 0 && privilege == S-mode )
    xFCFIE = menvcfg.SFCFIE
else if ( menvcfg.CFIE == 1 && V == 0 && privilege == U-mode )
    xFCFIE = mstatus.UFCFIE
else if ( menvcfg.CFIE == 1 && henvcfg.CFIE == 1 && V == 1 && privilege == S-mode )
    xFCFIE = henvcfg.SFCFIE
else if ( menvcfg.CFIE == 1 && henvcfg.CFIE == 1 && V == 1 && privilege == U-mode )
    xFCFIE = vsstatus.UFCFIE
else
    xFCFIE = 0
```

When forward-edge CFI is not active(xFCFIE = 0):

- The implementation does not update expected landing pad (ELP) state on indirect call or jump and does not require the instruction at the target of a indirect call or jump to be a landing pad instruction.
- The implementation does not update expected landing pad (ELP) when `lpc11` is executed.
- The instructions defined for forward-edge CFI revert to their Zimop defined behavior and write 0 to [rd].

4.2. Landing pad instruction

`lpc11` is the valid landing pad instructions at target of indirect jumps and indirect calls. When a forward-edge CFI is active, the instructions cause an illegal instruction exception if they are not placed at a 4-byte aligned `pc`. The `lpc11` has the lower landing pad label embedded in the `LLPL` field. `lpc11` causes an illegal instruction exception if the `LLPL` field in the instruction does not match the `lpl.LL` field.

When the instructions cause an illegal-instruction exception, the `ELP` does not change. The behavior of the trap caused by this illegal-instruction exception is specified in section [Section 4.5](#).

The operation of the `lpc11` instruction is as follows:

Listing 9. `lpc11` operation

```
If xFCFIE != 0
    // If PC not 4-byte aligned then illegal-instruction
    if pc[1:0] != 0
        Cause illegal-instruction exception
    // If lower landing pad label not matched -> illegal-instruction
    else if (inst.LLPL != lpl.LL)
        Cause illegal-instruction exception
    else
        ELP = NO_LP_EXPECTED
else
```

```
[rd] = 0;
endif
```



Concatenation of two instructions **A** and **B** may be consumed as a valid landing pad in the program. For example, consider a 32-bit instruction where the bytes 3 and 2 have a pattern of **4073h** or **c073h** (for example, the immediate fields of a **lui**, **auipc**, or a **jal** instruction), followed by a 16-bit or a 32-bit instruction with a second byte with pattern of **83** (for example, an **addi x6, x0, 1**).

The **lpc1l** requires a 4-byte alignment such that when such patterns are detected the assembler/linker the instruction **A** may be forced to be aligned to a 4-byte boundary to cause the unintended **lpc1l** pattern to become misaligned and cause an illegal instruction exception.

4.3. Label matching instructions

The **lpcm1** instruction matches the 8-bit wide middle label in its **MLPL** field with the **lp1.ML** field and causes an illegal instruction exception on a mismatch. The **lpcm1** is not a valid target for an indirect call or jump.

The **lpcu1** instruction matches the 8-bit wide upper label in its **ULPL** field with the **lp1.UL** field and causes an illegal instruction exception on a mismatch. The **lpcu1** is not a valid target for an indirect call or jump.

The operation of the **lpcm1** instruction is as follows:

Listing 10. lpcm1 operation

```
If xFCFIE != 0
    if (lp1.ML != inst.MLPL)
        cause illegal-instruction exception
else
    [dst] = 0;
endif
```

The operation of the **lpcu1** instruction is as follows:

Listing 11. lpcu1 operation

```
If xFCFIE != 0
    if (lp1.UL != inst.ULPL)
        cause illegal-instruction exception
else
    [dst] = 0;
endif
```

4.4. Setting up landing pad label register

Before performing an indirect call or indirect jump to a labeled landing pad, the `lpl` is loaded with the expected landing pad label - a constant determined at compilation time.


A `lpsll` instruction is provided to set the value of the lower label (`LL`) field of the `lpl`.

The operation of this instruction is as follows:

Listing 12. `lpsll` operation

```
If xFCFIE == 1
    lpl.LL = inst.LLPL
    lpl.ML = lpl.UL = 0
else
    [rd] = 0;
endif
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pad label register when labels that are up to 9-bit wide are used:



```
foo:
:
# x10 is expected to have address of function bar()
lpsll $0x1de    # setup lpl.LL with value 0x1de
jalr %ra, %x10
:
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pads at entrypoint of function `bar()`:

```
bar:
    lpcll $0x1de    # Verifies that lpl.LL matches 0x1de
:                  # continue if landing pad checks succeed
```

A `lpsml` instruction is provided to set the value of the middle label (`ML`) field of the `lpl`. This instruction is used when labels wider than 9-bit are used.

The operation of this instruction is as follows:

Listing 13. `lpsml` operation

```
If xFCFIE == 1
    lpl.ML = inst.MLPL
else
    [rd] = 0;
```

endif

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pad label register when labels that are up to 17-bit wide are used:



```
foo:
:
# x10 is expected to have address of function bar()
lpsll $0x1de    # setup lpl.LL with value 0x1de
lpsml $0x17     # setup lpl.ML with value 0x17
jalr %ra, %x10
:
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pads at entrypoint of function bar():

```
bar:
lpcll $0x1de    # Verifies that lpl.LL matches 0x1de
lpcml $0x17     # Verifies that lpl.ML matches 0x17
:               # continue if landing pad checks succeed
```

A **lpsul** instruction is provided to set the value of upper label (UL) field **lpl**. This instruction is used when labels wider than 17-bit are used.

The operation of this instruction is as follows:

Listing 14. lpsul operation

```
If xFCFIE == 1
    lpl.UL = inst.ULPL
else
    [rd] = 0;
endif
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pad label register when labels that are up to 25-bit wide are used:



```
foo:
:
# x10 is expected to have address of function bar()
lpsll $0x1de    # setup lpl.LL with value 0x1de
lpsml $0x17     # setup lpl.ML with value 0x17
lpsul $0x13     # setup lpl.UL with value 0x13
jalr %ra, %x10
```

:

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pads at entrypoint of function `bar()`:

```
bar:
    lpc11 $0x1de    # Verifies that lp1.LL matches 0x1de
    lpcm1 $0x17     # Verifies that lp1.ML matches 0x17
    lpcu1 $0x13     # Verifies that lp1.ML matches 0x13
    :               # continue if landing pad checks succeed
```

4.5. Preserving expected landing pad state on traps

A trap may need to be delivered to the same or higher privilege level on completion of JALR but before the instruction at the target of JALR was decoded due to asynchronous interrupts.

A trap may be caused by synchronous exceptions with priority lower than that of an illegal-instruction exception (See Table 3.7 of Privileged Specification [2]).

A trap may be caused by the illegal-instruction exception due to the instruction at the target of a JALR not being a `lpc11` instruction, or the `lpc11` instruction not being 4-byte aligned, or due to the `LLPL` encoded in the `lpc11` not matching the `LL` field of `lp1`.

To avoid losing previous `ELP` state, `MPERP` and `SPERP` bit is provided in the `mstatus` CSR for M-mode and HS/S-mode respectively. The `SPERP` bit can be accessed through the `sstatus` CSR. To avoid losing `ELP` state on traps to VS-mode, `SPERP` bit is provided in `vsstatus` (VS-modes version of `sstatus`) to hold the `ELP`. When a trap is taken into VS-mode, the `SPERP` bit of `vsstatus` CSR is updated with `ELP`. When `V=1`, `sstatus` aliases to `vsstatus` CSR. The `xPERP` fields in `mstatus` and `vsstatus` are WARL fields. The trap handler should preserve the `lp1` CSR.

When a trap is taken into privilege mode `x`, the `xPERP` bit is updated with current `ELP` and `ELP` is set to `NO_LP_EXPECTED`.

`MRET` or `SRET` instruction is used to return from a trap in M-mode or S-mode respectively. When executing an `xRET` instruction, the `ELP` is set to `xPERP` and `xPERP` is set to `NO_LP_EXPECTED`. The trap handler should restore the preserved `lp1` value before invoking `SRET` or `MRET`.