RISC-V Shadow Stacks and Landing Pads (Zisslpcfi)

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

Table of Contents

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

Contributors

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

Adam Zabrocki, Andrew Waterman, Antoine Linarès, Dean Liberty, Deepak Gupta, Eckhard Delfs, George Christou, Greg McGary, Henry Hsieh, Johan Klockars, Kip Walker, Liu Zhiwei, Mark Hill, Nick Kossifidis, Sotiris Ioannidis, Thurston Dang, 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 return stack or in the register or memory used by these instructions.

This specification describes CFI threat model, security objectives and the architectural design choices to ensure that control-flow subversion attacks are effectively thwarted.

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. 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 an 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` is a return from procedure if `rs1` is a conventional link register (i.e. `x1` or `x5`); else it is an 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 with a link register as destination, i.e., `rd != x0`. Conventionally, the link register is `x1` or `x5`. A call using `JAL` or `C.JAL` is termed a direct call. A `C.JALR` expands to `JALR x1, 0(rs1)` and is a call. A call using `JALR` or `C.JALR` is termed an indirect call.

The term "return" is used to refer to a `JALR` instruction with `rs1 == x1` or `rs1 == x5` and `rd == x0`. A `C.JR` instruction expands to `JALR x0, 0(rs1)` and is a return if `rs1 == x1` or `rs1 == x5`.

The term "indirect jump" is used to refer to a `JALR` instruction with `rd == x0` and where the `rs1` is not

`x1` or `x5` (i.e., not a return). A `C.JR` instruction where `rs1` is not `x1` or `x5` (i.e., not a return) is an indirect jump.

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 instructions (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-edge control-flow integrity has two stacks: a regular stack and a shadow stack. The shadow stack is used exclusively to store shadow copies of return addresses.

The shadow stack is used to spill the link register if required by non-leaf functions. An additional register, shadow-stack-pointer (`ssp`), is introduced in the architecture to hold the address of the top of the current active shadow stack. The shadow stack is architecturally protected from inadvertent corruptions and modifications, as detailed later. The extension provides instructions to store and load the link register to/from the shadow stack and to check the integrity of the return address.

Each function in a program that uses a shadow stack stores the link register to the regular 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 regular stack and the shadow copy of the link register from the shadow stack. The link register value from the regular 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 exception.

> Programs that use the shadow stack can operate in two modes: a shadow stack mode or a control stack mode. In shadow stack mode, programs store the return addresses on both the regular stack and the shadow stack in the function prologue, and then pop them them from both stacks and compare the values before returning from the function. In the control stack mode, programs only store the return addresses on the shadow stack and pop it from there to return from the function.
>
> Operating in shadow stack mode preserves the call stack layout and the ABI, while also allowing for the detection of corruption of the return address on the regular stack. Such programs are portable between implementations that support the Zisslpcfi extension as well as those that do not. Most programs are expected to use this mode.
>
> Operating in control stack mode breaks the ABI, but has the benefit of avoiding additional instructions to store the return address to two stacks, and to pop and compare them before returning from a function. This mode also allows the program to have a smaller regular stack as the space to save the return address is not needed. However, such programs are not portable to implementations that do not support the Zisslpcfi extension. Some just-in-time (JIT) compiled programs may dynamically switch between using only the regular stack or only the shadow stack to store return addresses, depending on the capabilities of the implementation.

To enforce forward edge control-flow integrity, Zisslpcfi extension introduces landing pad instructions that allow software to indicate valid targets for indirect calls and jumps in a program.

Compilers emit a landing pad instruction as the first instruction of address-taken functions, as well as at any indirect jump targets.

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

When the landing pad feature is active, the hart tracks an expected landing pad (`ELP`) that is updated by an indirect call or jump to require a landing pad instruction at the target. If the instruction at the target is not a landing pad, 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 jump site can be defined using programming language-based policies. A landing pad label (`lpl`) register is set up prior to initiating an indirect call or jump with the expected landing pad label. If the target of the indirect call or jump is not a landing pad or if the label of the landing pad does not match the label in `lpl` then an illegal-instruction exception is raised. Up to 25-bit wide labels are supported.

In the simplest form, a program can be built with a single label value to implement a coarse-grained version of forward-edge control-flow integrity. By constraining gadgets to be preceded by a landing pad instruction that marks the start of indirect callable functions, the program can significantly reduce the available gadget space.

A second form of label generation may generate a signature, such as a MAC, using the prototype of the function. Programs that use this approach would further constrain the gadgets accessible from a call site to only indirect callable functions that match the prototype of the called functions.

Another approach to label generation involves analyzing the control-flow-graph (CFG) of the program, which can lead to even more stringent constraints on the set of reachable gadgets. Such programs may further use the multi-label capability, which means that if a function is called from two or more call sites, the common functions can be labeled as reachable from each of the call sites.

For instance, consider two call sites A and B, where A calls functions X and Y, and B calls functions Y and Z. In a single label scheme, functions X, Y, and Z would need to be assigned the same label so that both call sites A and B can invoke the common function Y. This scheme would allow call site A to also call function Z and call site B to also call function X.

However, if function Y was assigned two labels - one corresponding to call site A and the other to call site B, then Y can be invoked by both call sites, but X can only be invoked by call site A and Z can only be invoked by call site B.

To support multiple labels, the compiler could generate a call-site-specific entry point for shared functions, with each entry point having its own landing pad instruction followed by a direct branch to the start of the function. This would allow the function to be labeled with multiple labels, each corresponding to a specific call site.

A portion of the label space may be dedicated to labeled landing pads that are only

valid targets of an indirect jump (and not an indirect call).

Forward-edge and backward-edge CFI may be activated independently for software that executes in U-mode, S-mode, or M-mode. The processor keeps track of the CFI activation and CFI state for each mode in the `mstatus` CSR. A subset of the fields in the `mstatus` CSR is accessible using the `sstatus` CSR. VS-mode's version of `sstatus` (`vsstatus`) tracks the CFI state for VS-mode and VU-mode.

The operating system may activate the use of Zisslpcfi by U-mode applications, with or without the extension being used by the operating system itself. The set of U-mode programs installed in an OS may be a mix, where some programs are compiled with Zisslpcfi capabilities and others that are not. The operating system can activate or deactivate the use of the extension in U-mode per application by context switching the Zisslpcfi state.

Hypervisors may activate the use of Zisslpcfi in a virtual machine, with or without the extension being used by the hypervisor itself. Virtual machines that use the extension may coexist with virtual machines that do not, with the hypervisor context switching the Zisslpcfi state of each virtual machine.

To use Zisslpcfi, the operating system and hypervisors have to be modified to use the capabilities, including the context switching of the extension state.

Machine mode firmware may activate the use of the extension in M-mode independent of its use in lower privilege modes.

The Zisslpcfi instructions are encoded using a subset of "May be op" instructions defined by the Zimop and Zcmop extensions. This subset of instructions reverts to their Zimop/Zcmop defined behavior when the Zisslpcfi extension is not implemented or if the extension has not been activated at a privilege mode. A program that is built with Zisslpcfi instructions can thus continue to operate correctly, but without control-flow integrity, on processors that do not support the extension or if the extension is not active.

An OS modified to support the Zisslpcfi extension typically includes system libraries (such as glibc) that are also compiled with the Zisslpcfi extension. However, these system libraries may need to dynamically link to programs that are not compiled with the Zisslpcfi extension. In such cases, when these programs are executed in user mode, the OS may disable the Zisslpcfi extension. When the Zisslpcfi instructions in the system libraries are invoked by these programs in user mode, they revert to their Zimop/Zcmop defined behavior. The OS only needs to carry one version of the system libraries, which are usable by both applications that use the extension and those that do not.

An OS distribution compiled with Zisslpcfi extension may be installed on a machine that does not support the extension. In such cases, the Zisslpcfi instructions revert to their Zimop/Zcmop defined behavior.

In case a program compiled with the Zisslpcfi extension is installed on an operating system that is not modified to support the extension or on a machine that does not support it, the Zisslpcfi instructions will revert to their Zimop/Zcmop

> defined behavior.

The Zisslpcfi extension depends on the Zicsr, A, Zimop, and Zcmop extensions.

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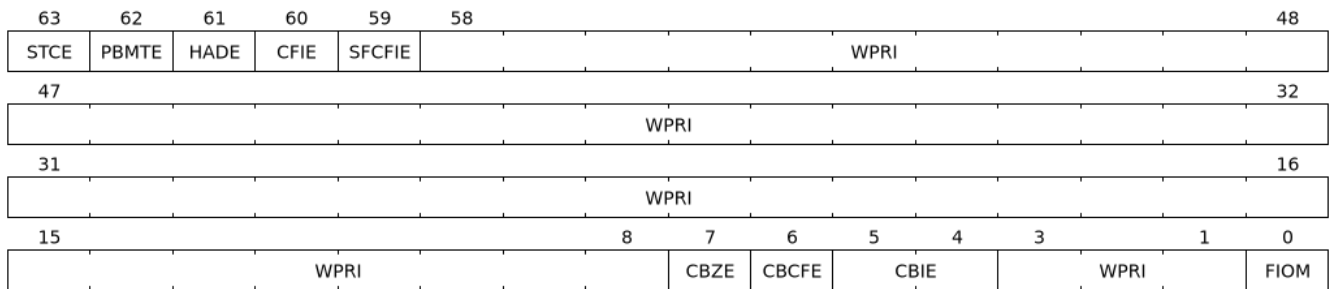
Figure 1. Machine environment configuration register (`menvcfg`) for MXLEN=64

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

When `menvcfg.CFIE` bit is 0, the following rules apply to privilege modes less privileged than M:

- Any attempts to access the `ssp` or `lpl` CSR will result in an illegal-instruction exception.

- Zisslpcfi extension instructions revert to the Zimop/Zcmop defined behavior.

- The `UBCFIE`, `UFCFIE`, and `SPELP` fields in `sstatus` will always read as zero.

- The `CFIE` field in `henvcfg` will always read as zero.

- The `pte.xwr=010b` encoding in S-stage page tables is reserved.

> ℹ When the Zisslpcfi extension is available for use at privilege mode less than M, the operating system may use the `UBCFIE` and `UFCFIE` to selectively enable the backward-edge and forward-edge CFI, respectively, for U-mode applications.
>
> When the Zisslpcfi extension is available for use in S-mode, the operating system may use shadow stacks at S-mode. If the operating system wants to use forward-edge CFI in S-mode, then it should request the SEE to set the `menvcfg.SFCFIE` field to 1 to enable it.
>
> The set of controls described above allows for the separate enforcement of backward-edge and forward-edge CFI at S-mode and at U-mode for each application.

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

63	62	61	60	59	58								48
STCE	PBMTE	HADE	CFIE	SFCFIE				WPRI					

47													32
							WPRI						

31													16
							WPRI						

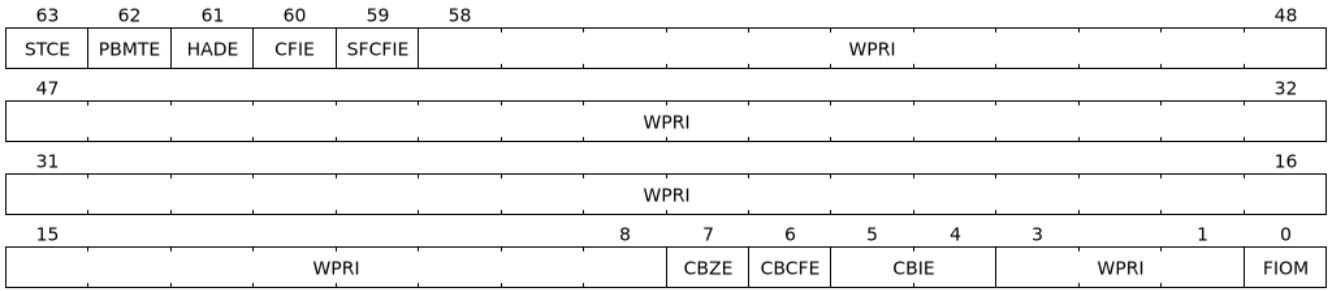| 15					8	7	6	5	4	3		1	0
		WPRI				CBZE	CBCFE	CBIE			WPRI		FIOM

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

The `CFIE` field (bit 60) controls whether the 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 the `ssp` or `lpl` CSR raise an illegal-instruction exception.
- Zisslpcfi extension instructions revert to the Zimop/Zcmop 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` field (bit 59) enables forward-edge CFI at VS-mode.

2.3. Machine status registers (`mstatus`)

63	62												48
SD							WPRI						

47						38	37	36	35	34	33	32
			WPRI				MBE	SBE	SXL[1:0]		UXL[1:0]	

31			27	26	25	24	23	22	21	20	19	18	17	16
	WPRI			MPELP	SPELP	UBCFIE	UFCFIE	TSR	TW	TVM	MXR	SUM	MPRV	XS[1:0]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XS[1:0]		FS[1:0]		MPP[1:0]		VS[1:0]	SPP	MPIE	UBE	SPIE	WPRI	MIE	WPRI	SIE	WPRI
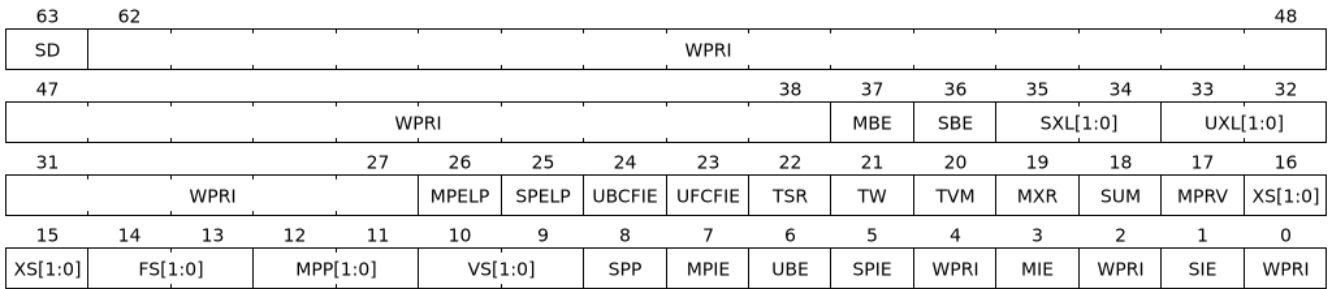
Figure 3. Machine-mode status register (`mstatus`) for RV64

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

The `SPELP` (bit 25) and `MPELP` (bit 26) WARL fields are hold the previous `ELP`, and are updated as specified in Section 4.6. The `xPELP` fields are encoded as follows:

- 0 - `NO_LP_EXPECTED` - no landing pad instruction expected.
- 1 - `LP_EXPECTED` - a landing pad instruction is expected.

2.4. Supervisor status registers (`sstatus`)

63	62														48
SD						WPRI									

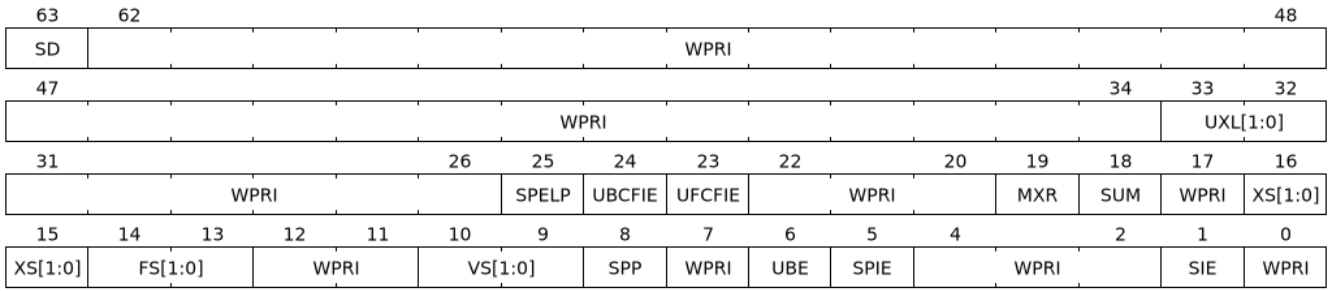| 47											34	33			32
					WPRI							UXL[1:0]			

31				26	25	24	23	22		20	19	18	17	16
	WPRI				SPELP	UBCFIE	UFCFIE		WPRI		MXR	SUM	WPRI	XS[1:0]

15	14	13	12	11	10	9	8	7	6	5	4		2	1	0
XS[1:0]	FS[1:0]		WPRI			VS[1:0]	SPP	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 the `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														48
SD						WPRI									

47											34	33			32
					WPRI							UXL[1:0]			

31				26	25	24	23	22		20	19	18	17	16
	WPRI				SPELP	UBCFIE	UFCFIE		WPRI		MXR	SUM	WPRI	XS[1:0]

15	14	13	12	11	10	9	8	7	6	5	4		2	1	0
XS[1:0]	FS[1:0]		WPRI			VS[1:0]	SPP	WPRI	UBE	SPIE		WPRI		SIE	WPRI

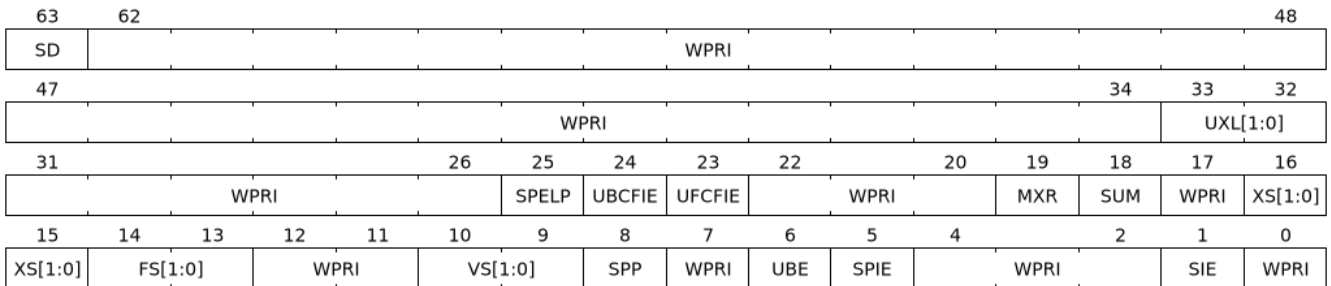*Figure 5. Virtual supervisor status register (*`vsstatus`*) when* `VSXLEN=64`

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.

> ℹ 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.
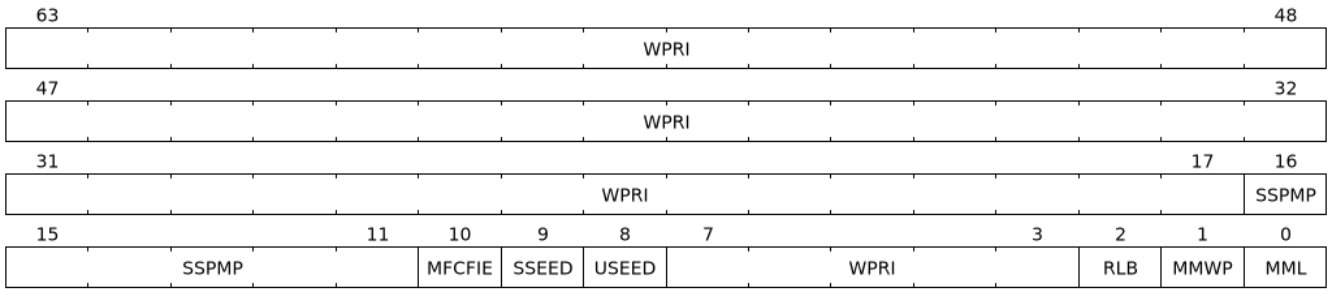
2.6. Machine Security Configuration (`mseccfg`)

Figure 6. Machine security configuration register (`mseccfg`*) when* `MXLEN=64`

A new WARL field `sspmp` is defined 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.

2.7. 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 an 8-bit upper label (`UL`), an 8-bit middle label (`ML`), and a 9-bit lower label (`LL`).
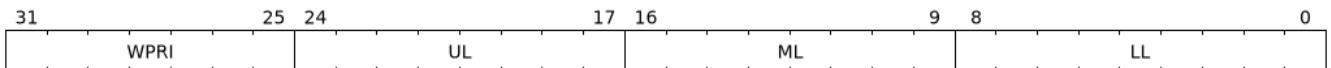

Figure 7. `lpl` *for RV32 and RV64*

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

> **ℹ** 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.

2.8. 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 mode.

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.

> **ℹ** Access to `ssp` at S-mode is not dependent on `sstatus.UBCFIE`, allowing an operating system to context switch U-mode `ssp` for each application as needed.

When `menvcfg.CFIE` is 1 but `henvcfg.CFIE` is 0, accessing `ssp` in VS-mode raises a virtual-instruction exception.

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

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.

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.

> ℹ️ 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 instructions

The backward-edge CFI extension introduces instructions for following shadow stack operations:

- Push to and pop from the shadow stack (See Section 3.3)
    - `sspush x1`, `c.sspush x1`, and `sspush x5`
    - `sspopchk x1`, `sspopchk x5`, and `c.sspopchk x5`
    - `sspop x1` and `sspop x5`
- Read the value of ssp into a register (See Section 3.4)
    - `ssprr`
- Perform atomic swap from a shadow stack location (See Section 3.5)
    - `ssamoswap`

The 32-bit instructions are encoded using the SYSTEM major opcode and using the `mop.r` and `mop.rr` encodings defined by the Zimop extension.

The 16-bit instructions are encoded using the `C.LUI` major opcode and using the `c.mop.0` and `c.mop.2` encodings defined by the Zcmop extension.

When a Zimop encoding is not used by the Zisslpcfi extension then the instruction follows its Zimop defined behavior.

3.2. Backward-edge CFI enables

When `menvcfg.CFIE` is 0, then Zisslpcfi is not enabled for privilege modes less than M.

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

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

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

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

Listing 1. `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 32-bit instructions defined for backward-edge CFI revert to their Zimop defined behavior and write 0 to [rd].

- The 16-bit instructions defined for backward-edge CFI revert to their Zcmop defined behavior of not performing any operation.

> Enabling the use of shadow stacks at U-mode must be done explicitly on a per-application basis. Explicitly enabling the use of shadow stacks for a user-mode application allows it to invoke shared libraries that may contain shadow stack instructions, even if the application itself does not have backward-edge CFI enabled. When shadow stack instructions are invoked in the context of this application, they revert to their Zimop/Zcmop-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 applications.
>
> 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 to execute with backward-edge CFI enabled.

When programs that use shadow stack instructions are installed on a processor that supports the Zisslpcfi extension but the extension has not been enabled at the privilege mode where the program executes, the program continues to function correctly. However, the shadow stack instructions will revert to their Zimop/Zcmop-defined behavior.
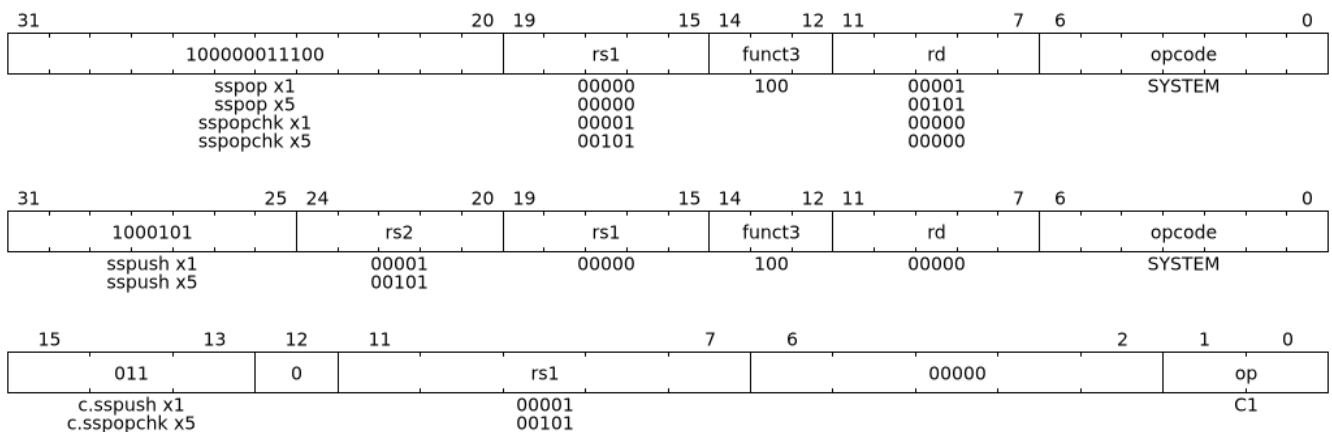
When programs that use shadow stack instructions are installed on a processor that does not support the Zisslpcfi extension but supports the Zimop/Zcmop extensions, the programs continues to function correctly. However, the shadow stack instructions will revert to their Zimop/Zcmop-defined behavior.

On processors that do not support Zimop/Zcmop extensions, the corresponding shadow stack instructions cause an illegal-instruction exception. Installing programs that use these instructions on such machines is not supported.

3.3. Push to and Pop from the shadow stack

A shadow stack 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 shadow stack pop operation is defined as a `XLEN` wide read from the current top of the shadow stack followed by an increment of the `ssp` by `XLEN`.

31	20	19	15	14	12	11	7	6	0
100000011100		rs1		funct3		rd		opcode	

sspop x1		00000		100		00001		SYSTEM	
sspop x5		00000				00101			
sspopchk x1		00001				00000			
sspopchk x5		00101				00000			

31	25	24	20	19	15	14	12	11	7	6	0
1000101		rs2		rs1		funct3		rd		opcode	

| sspush x1 | | 00001 | | 00000 | | 100 | | 00000 | | SYSTEM | |
| sspush x5 | | 00101 | | | | | | | | | |

15	13	12	11	7	6	2	1	0
011		0	rs1		00000		op	

| c.sspush x1 | | | 00001 | | | | C1 | |
| c.sspopchk x5 | | | 00101 | | | | | |

Only `x1` and `x5` encodings are supported as `rd` for `sspop`. Only `x1` and `x5` encodings are supported as `rs1` for `sspopchk`. Only `x1` and `x5` encodings are supported as `rs2` for `sspush`.

The extension includes 16-bit versions of the `sspush x1` and `sspopchk x5` instructions using the Zcmop encodings. The `c.sspush x1` and the `c.sspopchk x5` instructions are encoded using the `C.LUI` major opcode and using the `c.mop.0` and `c.mop.2` encodings defined by the Zcmop extension.

The `c.sspush x1` expands to `sspush x1` and `c.sspopchk x5` expands to `sspopchk x5`.

Usually programs with a shadow stack push the return address onto the regular 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.

The `sspush` instruction and its compressed form `c.sspush` can be used, to push a link register on the shadow stack.

The `sspopchk` instruction and its compressed form `c.sspopchk` can be used to pop the shadow return address value from the shadow stack and check that the value matches the contents of the link register.

The `sspop` instruction can be used to pop a return address from the shadow stack into a link register.

While any register may be used as link register, conventionally the `x1` or `x5` registers are used. The shadow stack instructions are designed to be most efficient when the `x1` and `x5` registers are used as the link register.

> Return-address prediction stacks are a common feature of high-performance instruction-fetch units, but they require accurate detection of instructions used for procedure calls and returns to be effective. For RISC-V, hints as to the instructions usage are encoded implicitly via the register numbers used. The return-address stack (RAS) actions to pop and/or push onto the RAS are specified in Table 2.1 of the Privileged specification.
>
> Using `x1` or `x5` as the link register allows a program to benefit from the return-address prediction stacks. Additionally, since the shadow stack instructions are designed around the use of `x1` or `x5` as the link register, using any other register as a link register would incur the cost of additional register movements.

> Programs may operate in shadow stack mode or in control stack mode.
>
> When operating in shadow stack mode, the program uses the shadow stack to store a shadow copy of the link register. Such programs push the link register on the regular stack as well as the shadow stack in the prologue of the function. In the epilogue, the link register value from the regular stack is compared to the shadow copy on the shadow stack. Programs operating in shadow stack mode are portable to implementations that do not support the Zisslpcfi extension. On implementations where the extension is not supported, the shadow stack instructions revert to their Zimop defined behavior but the program continues to function as the link register is also pushed and popped from the regular stack. Pushing and popping the link register to regular stack allows such programs to comply with the ABI. The prologue and epilogue of a function in shadow stack mode is as follows:

```
function_entry:
    addi sp,sp,-8  # push link register x1
    sd x1,(sp)     # on data stack
    #
    # Let the contents of ssp register be 0x0000000121679F8 and
    # XLEN be 64 ssp register holds the address of the top of
    # shadow stack. Let the contents of the link register x1
    # be 0x0000000010252000
    #
    # 0x00000000121679E8:[                    ]
    # 0x00000000121679F0:[                    ]
```

```
        # 0x00000000121679F8:[0xrrrrrrrrrrrrrrrr] <- ssp
        #
        sspush x1     # push link register x1 on shadow stack
        #
        # sspush store the source register value to address
        # (ssp - XLEN/8) and updates ssp to (ssp - XLEN/8) - does
        # a push. Following completion of # sspush the ssp value is
        # the new top of stack i.e. 0x0000000121679F0 and the value
        # in x1 is stored at this location
        #
        # 0x00000000121679E8:[                    ]
        # 0x00000000121679F0:[0x0000000010252000] <- ssp
        # 0x00000000121679F8:[0xrrrrrrrrrrrrrrrr]
        #
         :
         :
        ld x1,(sp)    # pop link register x1 from data stack
        addi sp,sp,8
        sspopchk x1   # compare link register x1 to shadow
                      # return address; faults if not same
        #
        # sspopchk loads the value from location addressed by ssp and
        # compares the loaded value to the value held in the register
        # source and if the two are identical updates ssp to
        # (ssp + XLEN/8) - does a pop and a check. Following
        # completion of sspopchk the ssp value is the # new top of
        # stack i.e. 0x00000000121679F8
        #
        # 0x00000000121679E8:[                    ]
        # 0x00000000121679F0:[0x0000000010252000]
        # 0x00000000121679F8:[0xrrrrrrrrrrrrrrrr] <- ssp
        #
        ret
```

Programs operating in the control stack mode store the return address only on the shadow stack. Such programs are not portable to implementations that do not support the Zisslpcfi extension. As these programs do not push a return address on the regular stack they may not be compliant with the ABI. The prologue and epilogue of a function when operating in control stack mode is as follows:

```
function_entry:
    #
    # Let the contents of ssp register be 0x19740428 and XLEN be 32
    # ssp register holds the address of the top of shadow stack
    # Let the contents of the link register x1 be 0x19791216
    #
    # 0x19740418:[          ]
    # 0x19740420:[          ]
    # 0x19740428:[0xrrrrrrrr] <- ssp
    #
```

```
        sspush x1      # push link register x1 on shadow stack
        #
        # Following sspush the shadow stack and ssp are as follows:
        #
        # 0x19740418:[           ]
        # 0x19740420:[0x19791216] <- ssp
        # 0x19740428:[0xrrrrrrrr]
        #
          :
          :
        sspop  x1      # pop return address from shadow stack
        #
        # sspop loads the value from location addressed by ssp into
        # destination register and updates ssp to (ssp + XLEN/8)
        # - does a pop. Following completion of sspop the ssp value
        # is the new top of stack i.e. 0x19740428
        #
        # 0x19740418:[           ]
        # 0x19740420:[0x19791216]
        # 0x19740428:[0xrrrrrrrr] <- ssp
        #
        ret
```

These examples illustrate the use of x1 register by the ABI as the link register. Alternatively, the ABI may use x5 as the 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 in either shadow stack mode or in control stack mode. The return value may be held in the link register itself for the duration of the leaf function execution.

The sspop, c.sspopchk, and sspopchk instructions perform a load identically to the existing LOAD instruction, with the difference that the base is implicitly ssp and the width is implicitly XLEN.

The sspush and c.sspush instructions performs a store identically to the existing STORE instruction, with the difference that the base is implicitly ssp and the width is implicitly XLEN.

The sspush, c.sspush, sspopchk, c.sspopchk, and sspop require the virtual address in ssp to have a shadow stack attribute (see Section 3.6).

Correct execution of sspush, c.sspush, sspopchk, c.sspopchk, and sspop require that ssp refers to idempotent memory. If the memory reference by the ssp is not idempotent, then the sspush/c.sspush instructions causes a store/AMO access-fault, and the sspop/sspopchk/c.sspopchk instructions cause a load access-fault.

If the virtual address in ssp is not XLEN aligned, then the sspop/ sspopchk/c.sspopchk instructions cause a load access-fault, and the sspush/ c.sspush instructions cause a store/AMO access-fault.

Misaligned accesses to shadow stack are not required and enforcing alignment is more secure to detect errors in the program. An access-fault exception is raised

instead of address-misaligned exception in such cases to indicate fatality and that the instruction must not be emulated by a trap handler.

The `sspopchk` instruction performs a load followed by a check of the loaded data value with the link register source. If the check against the link register faults, and the instruction is restarted by the trap handler, then the instruction will perform a load again. If the memory from which the load is performed is non-idempotent, then the second load may cause unexpected side effects. Shadow stack instructions require the memory referenced by `ssp` to be idempotent to avoid such concerns. Locating shadow stacks in non-idempotent memory, such as non-idempotent device memory, is not an expected usage, and requiring memory referenced by `ssp` to be idempotent does not pose a significant restriction.

When backward-edge CFI is enabled (i.e., `xBCFIE = 1`), the `c.sspush x1` instruction behaves identically to the `sspush x1` instruction, and the `c.sspopchk x5` instruction behaves identically to the `sspopchk x5` instruction.

The operation of the `sspush` and `c.sspush` instructions is as follows:

Listing 2. `sspush` and `c.sspush` operation

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

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

Listing 3. `sspop` operation

```
if (xBCFIE = 1)
    dst   = *[ssp]              # Load dst from address in ssp
                               # Only x1 and x5 may be used as dst
    [ssp] = [ssp] + (XLEN/8)   # Increment ssp by XLEN/8.
else
    [dst] = 0;
endif
```

The operation of the `sspopchk` and `c.sspopchk` instructions is as follows:

Listing 4. `sspopchk` and `c.sspopchk` operation

```
if (xBCFIE = 1)
    temp  = *[ssp]             # Load temp from address in ssp and
    if temp != [src]          # Compare temp to value in src and
                              # cause an illegal-instruction exception
                              # if they are not bitwise equal.
```

```
                            # Only x1 and x5 may be used as src
        Raise illegal-instruction exception
    else
        [ssp] = [ssp] + (XLEN/8)# increment ssp by XLEN/8.
    endif
else
    [dst] = 0;
endif
```

The `ssp` is incremented by `sspop`, `sspopchk`, and `c.sspopchk` only if the load from the shadow stack completes successfully. The `ssp` is decremented by `sspush` and `c.sspush` only if the store to the shadow stack completes successfully.

> The use of the compressed instruction `c.sspush x1` to push on the shadow stack is most efficient when the ABI uses `x1` as the link register, as the link register may then be pushed without needing a register-to-register move in the function prologue. To use the compressed instruction `c.sspopchk x5`, the function should pop the return address from regular stack into the alternate link register `x5` and use the `c.sspopchk x5` to compare the return address to the shadow copy stored on the shadow stack. The function then uses `c.jr x5` to jump to the return address.

```
    function_entry:
        c.addi sp,sp,-8  # push link register x1
        c.sd x1,(sp)     # on data stack
        c.sspush x1      # push link register x1 on shadow stack
         :
         :
        c.ld x5,(sp)     # pop link register x5 from data stack
        c.addi sp,sp,8
        c.sspopchk x5    # compare link register x5 to shadow
                         # return address; faults if not same
        c.jr x5
```

> Store-to-load forwarding is a common technique employed by high-performance processor implementations. CFI implementations may prevent forwarding from a non-shadow-stack store to `sspop`/`sspopchk`/`c.sspopchk` instructions. 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`/`sspopchk` /`c.sspopchk`.
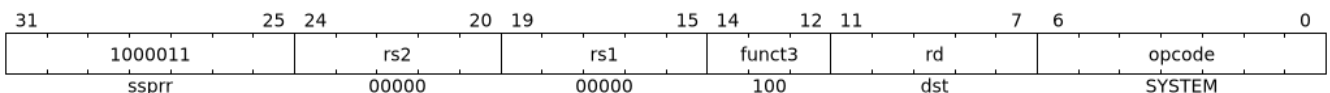
> A common operation performed on stacks is to unwind them to support constructs like `setjmp`/`longjmp`, C++ exception handling, etc. A program that uses shadow stacks must unwind the shadow stack in addition to the stack used to store data. The unwind function must verify that it does not accidentally 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 shadow stack instructions to unwind a shadow stack. This example assumes that the `setjmp` function itself does not push on to the shadow stack (being a leaf function, it is not required to).

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

3.4. Read `ssp` into a register

The `ssprr` instruction is provided to move the contents of `ssp` to the destination register.

31		25	24		20	19		15	14		12	11		7	6		0
	1000011			rs2			rs1			funct3			rd			opcode	
	ssprr			00000			00000			100			dst			SYSTEM	

Encoding `rd` as `x0` is not supported for `ssprr`.

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

Listing 5. `ssprr` operation

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

> ℹ️ 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:
>
> ```
>         ssprr 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:
> ```

3.5. Atomic Swap from a shadow stack location

The `ssamoswap` instruction performs an atomic swap operation between the `XLEN` bits of the `src` register and the `XLEN` bits located on the shadow stack at the address specified in the `addr` register. The resulting value from the swap operation is then stored into the register specified in the `dst` operand.

31 25	24 20	19 15	14 12	11 7	6 0
1000001	rs2	rs1	funct3	rd	opcode
ssamoswap	src	addr	100	dst	SYSTEM
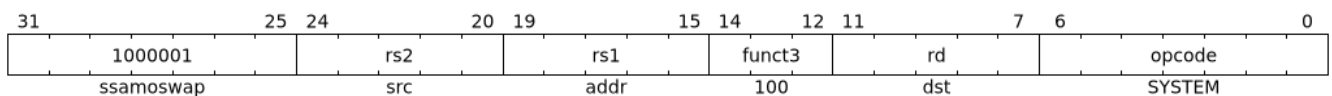
Encoding `rd` as `x0` is not supported for `ssamoswap`.

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 Section 3.6).

If the virtual address is not `XLEN` aligned, then `ssamoswap` causes a store/AMO access-fault exception.

If the memory reference by the `ssp` is not idempotent, then `ssamoswap` causes a store/AMO access-fault exception.

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

Listing 6. `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. If the pointer to the top of the deactivated shadow stack is held in a context data structure, then it may be susceptible to memory corruption vulnerabilities. To protect the pointer value, the program may store it at the top of the deactivated 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:
    ssprr  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 checkpoint data is zeroed as part of the restore sequence and if multiple hart attempt to restore the checkpoint 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 `Smepmp` 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`, `c.sspush`, and `ssamoswap`. The `sspop`, `sspopchk`, and `c.sspopchk` instructions 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 represent a shadow stack page. When `menvcfg.CFIE=0`, this encoding remains reserved. When `V=1` and `henvcfg.CFIE=0`, this encoding remains 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 store/AMO access-fault.

   b. Instruction fetches cause an instruction page-fault.

2. If the accessed page is not a shadow stack page or if the page is in non-idempotent memory:

   a. `ssamoswap`, `c.sspush`, and `sspush` cause a store/AMO access-fault.

   b. `sspop`, `c.sspopchk`, and `sspopchk` cause a load access-fault.

   > Stores to shadow stack by instructions other than `sspush`, `c.sspush`, and `ssamoswap` cause an access-fault, rather than a page-fault, to indicate fatality.
   >
   > If a page-fault was triggered, it would suggest that the operating system should service that fault and correct the condition. Correcting the condition is not possible in this case. The page-fault handler would have to resort to decoding the opcode of the instruction that caused the page-fault to determine if it was caused by non-shadow-stack-stores to shadow stack pages (which is a fatal condition) vs. a page fault caused by an `sspush`, `c.sspush`, or `ssamoswap` to a non-resident page (which is a recoverable condition). Since the operating system page-fault handler is typically performance-critical, causing an access-fault instead of a page-fault enables the operating system to easily distinguish between the fatal/non-recoverable conditions and recoverable page-faults.
   >
   > On implementations where address-misaligned exception is prioritized higher than access-fault exception, a trap handler handler that emulates misaligned

stores must cause an access-fault exception if the store is not `sspush`, `c.sspush`, or, `ssamoswap`, and the store is being made to a shadow stack page.

Shadow stack instructions cause an access-fault if the accessed page is not a shadow stack page or if the page is in non-idempotent memory to similarly indicate fatality.

Instruction fetch from a shadow stack page causes a page-fault because this condition is clearly distinguished by a unique cause code and is non-recoverable.

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 PAGESIZE` 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 PMA checks are extended to require memory referenced by `sspush`, `sspop`, `ssamoswap`, `c.sspush`, `c.sspopchk`, and `sspopchk` to be idempotent.

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 Svnapot extensions are supported for shadow stack pages.

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

> Shadow stacks are expected to be bounded on each end using guard pages, so that no two shadow stacks are adjacent to each other. This guards against accidentally underflowing or overflowing from one shadow stack to another. Traditionally, a guard page for a stack is a page that is inaccessible to the process owning the stack. For shadow stacks, the guard page may also be a non-shadow-stack page that is otherwise accessible to the process owning the shadow stack because shadow stack loads and stores to non-shadow-stack pages will result in an exception.

The G-stage address translation and protections remain affected by the shadow stack extension.

When G-stage page tables are active, the `ssamoswap`, `sspop`, `c.sspopchk`, and `sspopchk` instructions require the G-stage page table to have read permission for the accessed memory, whereas the `ssamoswap`, `c.sspush`, and `sspush` instructions require write permission. The `xwr == 010b` encoding in the G-stage PTE remains reserved.

> ℹ️ A future extension may define a shadow stack encoding in the G-stage page table to support use cases such as a hypervisor enforcing shadow stack protections for its guests.

> ℹ️ 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 debugging, 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 `sspush`, `c.sspush`, and `ssamoswap` must provide write permission and the PMP region accessed by `sspop`, `c.sspopchk`, and `sspopchk` must provide read permission.

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

The M-mode memory accesses by `sspop`, `c.sspopchk`, and `sspopchk` instructions test for read permission in the matching PMP entry when permission checking is required.

A new WARL field `sspmp` is defined in the `mseccfg` CSR to identify a PMP entry as the shadow stack memory region for M-mode accesses.

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

When the `sspmp` field is implemented, the following rules are additionally enforced for M-mode memory accesses:

- `sspush`, `c.sspush`, `sspop`, `sspopchk`, `c.sspopchk`, and `ssamoswap` instructions must match PMP entry `sspmp`.
- Write by instructions other than `sspush`, `c.sspush`, and `ssamoswap` that match PMP entry `sspmp` cause an access-violation exception.

> ℹ️ 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 instruction (`lpcll`) 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 an indirect call or jump. The `ELP` state can be one of:

- 0 - `NO_LP_EXPECTED`
- 1 - `LP_EXPECTED`

The Zisslpcfi extension determines if an indirect call or an indirect jump must land on landing pad, as specified in Listing 7. If `is_lp_expected` is 1, an indirect call or jump updates the `ELP` to `LP_EXPECTED`.

Listing 7. Landing pad expected determination

```
is_indirect_call_jump = ( (JALR || C.JR || C.JALR) &&
                          (rs1 != x1) && (rs1 != x5) ) ? 1 : 0;
is_sw_guarded_jump = ( JALR && rd == x7 && rs1 == x7 ) ? 1 : 0;
is_lp_expected = is_indirect_call_jump & ~is_sw_guarded_jump;
```

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

> ℹ️ The tracking of `ELP` and the requirement for 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, which use speculative execution to leak data through side channels.

When the indirect branch using `JALR` encodes both `rd` and `rs1` as `x7`, the branch is termed a software guarded branch. Such branches do not need to land on a landing pad and thus do not update `ELP`. Such branches must be used by a program only when the compiler or the program has emitted code to explicitly verify that the target held in `x7` is a valid target for that branch.

> ℹ️ Software guarded branches are expected to be used by compilers for generating

code for constructs like switch-cases. When using the software guarded branches, the compiler is required to ensure it has full control on the possible jump targets (e.g., by obtaining the targets from a read-only table in memory and performing bounds checking on the index into the table, etc.).

While software guarded branches may be secured using such compiler generated checks, in some cases they may be susceptible. For example, where software can be interrupted, the `x7` register may be spilled to mutable memory by the interrupt or signal handler. The memory location where the register is spilled may be susceptible to modifications. Therefore, software should opt to use the software guarded branches only where such threats are not applicable or are mitigated.

By default a landing pad allows an indirect call/jump to land on any `lpcll` in the program, which significantly reduces the number of valid targets for an indirect call/jump. Labeling of the landing pads enables software to achieve greater precision in pairing up indirect call/jump sites 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 (`lpl`) register. If the target of the indirect call/jump is a valid landing pad instruction, the expected label established in the `lpl` 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 can be up to 25-bits wide. The `lpl` has three subfields - a 9-bit lower label (`LL`), an 8-bit middle label (`ML`), and an 8-bit upper label (`UL`).

The `lpsll` instruction is used to set the expected `LL` in the `lpl` and to set the `ML` and `UL` fields of `lpl` to 0. The `lpsml` instruction is used to set the expected `ML` in `lpl` while keeping the `UL` and `LL` fields unchanged. The `lpsul` instruction is used to set the expected `UL` in `lpl` while keeping the `LL` and `ML` fields unchanged. The values to be set are embedded as immediate fields in these instructions.

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

For label widths up to 17-bits, 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-bits, 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 Instructions

The forward-edge CFI introduces the following instructions for landing pad operations:

- Landing pad (See Section 4.3)
  - `lpcll`
- Label matching (See Section 4.4)
  - `lpcml` and `lpcul`

- Setup landing pad label register (See Section 4.5)

  ◦ `lpsll`, `lpsml`, and `lpsul`

These instructions are encoded using the SYSTEM major opcode and the `mop.rr` encodings defined by the Zimop extension.

When a Zimop encoding is not used by the Zisslpcfi extension then the instruction follows its Zimop defined behavior.

4.2. Forward-edge CFI enables

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

When `menvcfg.CFIE` is 0, 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 in S-mode if `menvcfg.SFCFIE` is 1 and is active in U-mode if `mstatus.UFCFIE` is 1.

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

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

The term `xFCFIE` is used to determine if forward-edge CFI is active at privilege mode `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
```
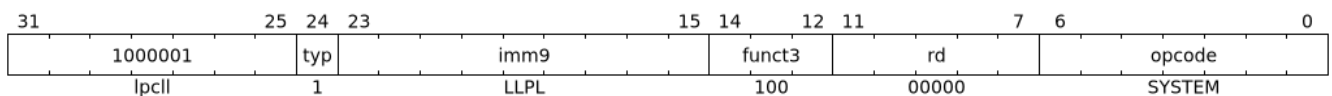
When forward-edge CFI is not active (`xFCFIE = 0`):

- The hart does not update the expected landing pad (`ELP`) state on an indirect call or jump, and does not require the instruction at the target of an indirect call or jump to be a landing pad instruction.

- The hart does not update the expected landing pad (`ELP`) when `lpcll` is executed.

- The instructions defined for forward-edge CFI revert to their Zimop-defined behavior and do

not set or check landing pad labels.

4.3. Landing pad instruction

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

31		25	24 23			15 14	12 11		7 6		0
	1000001		typ		imm9		funct3	rd		opcode	
	lpcll		1		LLPL		100	00000		SYSTEM	

When the instruction causes 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 4.6.

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

Listing 9. `lpcll` 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
```

Whereas `lpcll` is the only instruction that can execute when `ELP` is `LP_EXPECTED`, `lpcll` can also execute when `ELP` is `NO_LP_EXPECTED`.
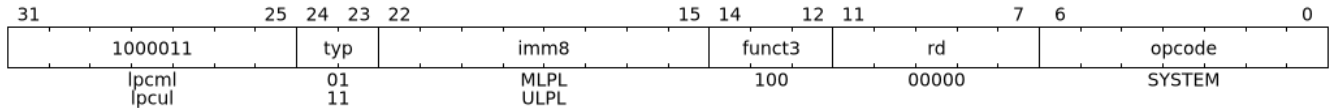
> ℹ️ Concatenation of two instructions `A` and `B` can accidentally form a valid landing pad in the program. For example, consider a 32-bit instruction where the bytes 3 and 2 have a pattern of `4073h` or `c073h` (for example, the immediate fields of a `lui`, `auipc`, or a `jal` instruction), followed 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 `lpcll` requires a 4-byte alignment. When patterns that can accidentaly form a valid landing pad are detected, the assembler/linker can force instruction `A` to be aligned to a 4-byte boundary to force the unintended `lpcll` pattern to become misaligned and thus not a valid landing pad.

4.4. Label matching instructions

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

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

31	25	24 23	22	15 14	12 11	7 6	0
1000011		typ	imm8	funct3	rd	opcode	
lpcml		01	MLPL	100	00000	SYSTEM	
lpcul		11	ULPL				

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

Listing 10. `lpcml` operation

```
If xFCFIE != 0
    if (lpl.ML != inst.MLPL)
        cause illegal-instruction exception
else
    [dst] = 0;
endif
```
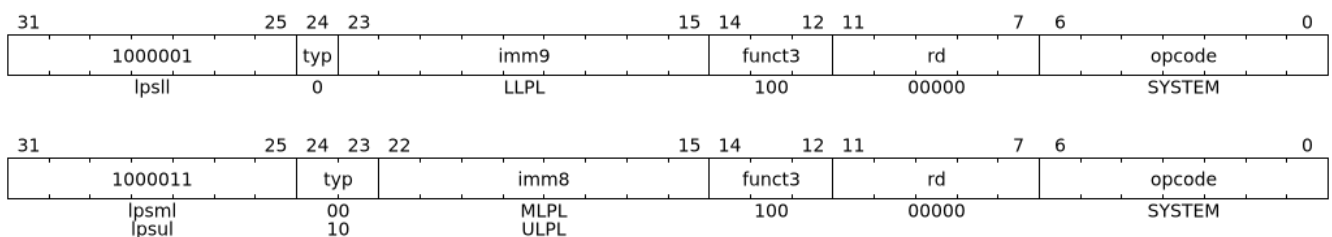
The operation of the `lpcul` instruction is as follows:

Listing 11. `lpcul` operation

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

4.5. 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. The label is a constant encoded into the instructions used to setup the `lpl`.

31	25	24 23	15 14	12 11	7 6	0
1000001		typ	imm9	funct3	rd	opcode
lpsll		0	LLPL	100	00000	SYSTEM

31	25	24 23	22	15 14	12 11	7 6	0
1000011		typ	imm8	funct3	rd	opcode	
lpsml		00	MLPL	100	00000	SYSTEM	
lpsul		10	ULPL				

The `lpsll` instruction is used to set the value of the lower label (`LL`) field of the `lpl`. In addition to setting `LL`, the instruction sets the `ML` and `UL` fields to 0.

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 compiler may emit the following instruction sequence at indirect call/jump sites to set up the landing pad label register when using labels up to 9 bits wide:
>
> ```
> foo:
>     :
>     # x10 is expected to have address of function bar()
>     lpsll $0x1de    # setup lpl.LL with value 0x1de
>     jalr %ra, %x10
>     :
> ```
>
> The compiler may emit the following instruction sequence at the indirect call/jump targets, such as function entry points, to create a landing pad:
>
> ```
> bar:
>     lpcll $0x1de    # Verifies that lpl.LL matches 0x1de
>     :               # continue if landing pad checks succeed
> ```

The `lpsml` instruction is used to set the value of the middle label (`ML`) field of the `lpl`. The `UL` and `LL` fields of the `lpl` remain unchanged. This instruction is typically used when labels wider than 9-bit are required.

The operation of this instruction is as follows:

Listing 13. `lpsml` operation

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

> The compiler may emit the following instruction sequence at indirect call/jump sites to set up the landing pad label register when using labels up to 17 bits wide:

```
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 compiler may emit the following instruction sequence at the indirect call/jump targets, such as function entry points, to create a landing pad:

```
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 used to set the value of upper label (`UL`) field the `lpl`. The `LL` and `ML` fields remain unchanged. This instruction is typically used when labels wider than 17-bit are required.

The operation of this instruction is as follows:

Listing 14. `lpsul` operation

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

The compiler may emit the following instruction sequence at indirect call/jump sites to set up the landing pad label register when using labels up to 25 bits wide:

```
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 compiler may emit the following instruction sequence at the indirect call/jump targets, such as function entry points, to create a landing pad:

```
bar:
```

```
        lpcll $0x1de    # Verifies that lpl.LL matches 0x1de
        lpcml $0x17     # Verifies that lpl.ML matches 0x17
        lpcul $0x13     # Verifies that lpl.ML matches 0x13
        :               # continue if landing pad checks succeed
```

The `lpcml` and `lpcul` need not occur together or in that order. Use of a `lpcul` does not require a preceding or a following `lpcml`. The following sequences are also a valid label check sequence:

```
bar:
    lpcll $lwr_label
    lpcul $upr_label
    :
```

```
bar:
    lpcll $lwr_label
    lpcul $upr_label
    lpcml $mdl_label
    :
```

A `lpsll` sets the `LL` and also initializes the `ML` and `UL` fields to zero. If the label to be assigned has zero for `ML` and `UL`, then there is no need to explicitly set them to zero using a `lpsml` or `lpsul`. `lpsml` and `lpsul` can be used independently and in any order. The use of a `lpsul` does not require a preceding or following `lpsml`.

4.6. Preserving expected landing pad state on traps

A trap may need to be delivered to the same or to a higher privilege mode upon completion of `JALR`/`C.JALR`/`C.JR`, but before the instruction at the target of indirect call/jump was decoded, due to:

- Asynchronous interrupts.

- Synchronous exceptions with priority lower than that of an illegal-instruction exception (See Table 3.7 of Privileged Specification [2]).

- By the illegal-instruction exception due to the instruction at the target not being an `lpcll` instruction, or the `lpcll` instruction not being 4-byte aligned, or due to the `LLPL` encoded in the `lpcll` not matching the `LL` field of `lpl`.

In such cases, the `ELP` prior to the trap, the previous `ELP`, may be `LP_EXPECTED`.

To store the previous `ELP` state on trap delivery to M-mode, a `MPELP` bit is provided in the `mstatus` CSR to hold the previous `ELP`.

To store the previous `ELP` state on trap delivery to S/HS-mode, a `SPELP` bit is provided in the `mstatus` CSR to hold the previous `ELP`. The `SPELP` bit in `mstatus` can be accessed through the `sstatus` CSR.

To store the previous `ELP` state on traps to VS-mode, a `SPELP` bit is defined in the `vsstatus` (VS-modes version of `sstatus`) to hold the previous `ELP`.

When a trap is taken into privilege mode `x`, the `xPELP` is set to `ELP` and `ELP` is set to `NO_LP_EXPECTED`.

An `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 `xPELP`, and the `xPELP` is set to `NO_LP_EXPECTED`.

> The trap handler in privilege mode `x` must save the `xPELP` bit and the `lpl` register before performing an indirect call/jump. If the privilege mode `x` can respond to interrupts, then the trap handler should also save these values before enabling interrupts.
>
> The trap handler in privilege mode `x` must restore the saved `xPELP` bit and the `lpl` register before executing the `xRET` instruction to return from a trap.

Bibliography

[1] "RISC-V Instruction Set Manual, Volume I: Unprivileged ISA ." [Online]. Available: github.com/riscv/riscv-isa-manual.

[2] "RISC-V Instruction Set Manual, Volume II: Privileged Architecture ." [Online]. Available: github.com/riscv/riscv-isa-manual.