RISC-V Shadow Stacks and Landing Pads

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

Version v, 2023-08-23:

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 Favor, Greg McGary, Henry Hsieh, Johan Klockars, John Hauser, John Ingalls, Kip Walker, Kito Cheng, Lasse Collin, Liu Zhiwei, Mark Hill, Nick Kossifidis, Phillip Reames, Sotiris Ioannidis, Stefan O’Rear, Thurston Dang, Tsukasa OI, Vedvyas Shanbhogue

Chapter 1. Introduction

The Zicfiss extension provides Control-flow Integrity (CFI) capabilities to defend against Return-Oriented Programming (ROP) and the Zicfilp extension provides CFI capabilities to defend against Call/Jump-Oriented Programming (COP/JOP) style control-flow subversion attacks. These attack methodologies use 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 memory used to obtain the target address for a call or jump. Attackers stitch these sequences together by diverting the control flow instructions (e.g., `JALR`, `C.JR`, `C.JALR`), from their original target address to a new target via modification in the return stack or in the memory used to obtain the jump/call target address.

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 a procedure call and `JALR` to 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 `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 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 `rd == x0` and 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*.

The Zicfiss and Zicfilp extensions build on these conventions and hints.

1.1. Backward-edge control-flow integrity

To enforce backward-edge control-flow integrity, the Zicfiss extension introduces a shadow stack.

The shadow stack is designed to provide integrity to control transfers performed using a *return* (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 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 active shadow stack.

The shadow stack is architecturally protected from inadvertent corruptions and modifications, as detailed later (See Section 2.8).

The Zicfiss extension provides instructions to store and load the link register to/from the shadow stack and to check the integrity of the return address. The extension provides instructions to support common stack maintenance operations such as stack unwinding and stack switching.

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

The Zicfiss extensions may be activated for use individually and independently for each privilege mode.

Compilers should flag each object file (for example, using flags in the elf attributes) to indicate if the object file has been compiled with the Zicfiss instructions. The linker should flag (for example, using flags in the elf attributes) the binary/executable generated by linking objects as being compiled with the Zicfiss instructions only if all the object files that are linked have the same Zicfiss attributes.

The dynamic loader should activate the use of Zicfiss extension for an application only if all executables (the application and the dependent dynamically linked libraries) used by that application use the Zicfiss extension.

An application that has the Zicfiss extension active may request the dynamic loader at runtime to load a new dynamic shared object (using dlopen() for example). If the requested object does not have the Zicfiss attribute then the dynamic loader, based on its policy (e.g, established by the operating system or the administrator) configuration, either fail the request or deactivate the Zicfiss extension for the application.

When the Zicfiss extension is not active or not implemented, the Zicfiss instructions revert to their Zimop/Zcmop defined behavior. This allows a compiled with Zicfiss instructions to operate correctly but without backward-edge control-flow integrity.

The Zicfiss extension is specified in Chapter 2 and the CSR state introduced is specified in [CSRs]. The Zicfiss extension depends on the Zicsr, A, Zimop, and Zcmop extensions.

1.2. Forward-edge control-flow integrity

To enforce forward edge control-flow integrity, Zicfilp extension introduces a landing pad (`lpad`) instruction that allows software to indicate valid targets for indirect calls and jumps in a program.

Compilers emit a landing pad instruction as the first instruction of an address-taken functions, as well as at any indirect jump targets. A landing pad instruction is not required in functions that are only reached using a direct call or direct jump.

The landing pad is designed to provide integrity to control transfers performed using indirect call and jumps, and this is referred to as forward-edge protection. When the Zicfilp is active, the hart tracks an expected landing pad (`ELP`) state that is updated by an *indirect_call* or *indirect_jump* to require a landing pad instruction at the target of the branch. If the instruction at the target is not a landing pad, then a software error exception is raised.

A landing pad may be optionally associated with a 20-bit label. 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. Labeling of the landing pads enables software to achieve greater precision in pairing up indirect call/jump sites with valid targets. When labeling of landing pads is used, indirect call or indirect jump site can specify the expected label of the landing pad and thereby constrain the set of landing pads that may be reached from each indirect call or indirect jump site in the program.

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 multiple labels per function, which means that if a function is called from two or more call sites, the functions can be labeled as reachable from each of the call sites. For instance, consider two call sites A and B, where A calls the functions X and Y, and B calls the 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).

The `lpad` instruction uses the code points defined as HINTs for the `AUIPC` opcode. When Zicfilp is not active at a privilege level or when the extension is not implemented, the landing pad instruction executes as a no-op. A program that is built with `lpad` instruction can thus continue to operate correctly, but without forward-edge control-flow integrity, on processors that do not support the Zicfilp extension or if the Zicfilp extension is not active.

As discussed earlier for the Zicfiss extension, compilers, linkers, and dynamic loaders should provided an attribute flag to indicate if the program has been compiled with the Zicfilp extension and use that to determine if the Zicfilp extension should be activated.

When Zicfilp extension is not active or not implemented, that hart does not required landing pad instructions at targets of indirect calls/jumps and the landing instructions revert to being a no-op. This allows a program compiled with landing pad instructions to operate correctly but without forward-edge control-flow integrity.

The Zicfilp extensions may be activated for use individually and independently for each privilege mode.

The Zicfilp extension is specified in Chapter 3 and the CSR state introduced is specified in [CSRs]. The Zicfilp extension depends on the Zicsr extension.

Chapter 2. Shadow Stack (Zicfiss)

To enforce backward-edge control-flow integrity, the Zicfiss extension introduces a shadow stack. A shadow stack is a second stack used to store a shadow copy of the return address in the link register if it needs to be spilled.

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 stored on the shadow stack.

When Zicfiss is enabled, each function that needs to spill the link register (e.g., non-leaf functions) stores the link register value to the regular stack and a shadow copy of the link register value to the shadow stack when the function is entered (the prologue). When such a function need 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 software error exception (cause=18) with `*tval` set to "shadow stack fault (code=3)". The software error exception caused by the shadow stack fault is lower in priority than the load access fault exception.

The Zicfiss extension introduces the following instructions:

- Push to the shadow stack (See Section 2.3)
  - `sspush x1`, `c.sspush x1`, and `sspush x5`
- Pop from the shadow stack (See Section 2.4)
  - `sspopchk x1`, `c.sspopchk x5`, and `sspopchk x5`
- Load from the shadow stack (See Section 2.5)
  - `sslw x1` and `sslw x5` when effective xlen is 32
  - `ssld x1` and `ssld x5` when effective xlen is 64
- Increment the shadow stack pointer (See Section 2.6)
  - `ssincp` and `c.ssincp`
- Read the value of `ssp` into a register (See Section 2.7)
  - `ssrdp`

The 32-bit instructions are encoded using the `SYSTEM` major opcode and using the `mop.r.0`, `mop.r.1`, and `mop.rr.0` encodings defined by the Zimop extension.

The 16-bit instructions are encoded using the `C.LUI` major opcode and using the `c.mop.0`, `c.mop.1` and `c.mop.2` encodings defined by the Zcmop extension.

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

2.1. Zicfiss CSRs

This chapter specifies the CSR state of the Zicfiss extensions.

2.1.1. Machine environment configuration registers (`menvcfg and menvcfgh`)

63	62	61	60–48
STCE	PBMTE	HADE	WPRI

47–32
WPRI

31–16
WPRI

15–8	7	6	5–4	3	2–1	0
WPRI	CBZE	CBCFE	CBIE	SSE	WPRI	FIOM

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

Zicfiss extension introduces the `SSE` field (bit 3) in `menvcfg`. When `SSE` field is 1, the Zicfiss extension is enabled in S-mode. When `SSE` field is 0, the Zicfiss extension is not enabled in S-mode and the following rules apply to privilege modes less than M.

- Attempts to access the `ssp` CSR raise an illegal instruction exception.
- The 32-bit Zicfiss instructions revert to their Zimop defined behavior.
- The 16-bit Zicfiss instructions revert to their Zcmop defined behavior.
- The `pte.xwr=010b` encoding in S-stage page tables is reserved.
- The `henvcfg.SSE` and `senvcfg.SSE` fields are read-only zero.

2.1.2. Supervisor environment configuration registers (`senvcfg`)

63–48
WPRI

47–32
WPRI

31–16
WPRI

15–8	7	6	5–4	3	2–1	0
WPRI	CBZE	CBCFE	CBIE	SSE	WPRI	FIOM

Figure 2. Supervisor environment configuration register (`senvcfg`) when `SXLEN=64`

Zicfiss extension introduces the `SSE` field (bit 3) in `senvcfg`. When `SSE` field is 1, the Zicfiss extension is enabled in VU/U-mode. When `SSE` field is 0, the Zicfiss extension is not enabled in VS/U-mode and the following rules apply:

- Attempts to access the `ssp` CSR raise an illegal instruction exception.
- The 32-bit Zicfiss instructions revert to their Zimop defined behavior.
- The 16-bit Zicfiss instructions revert to their Zcmop defined behavior.

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

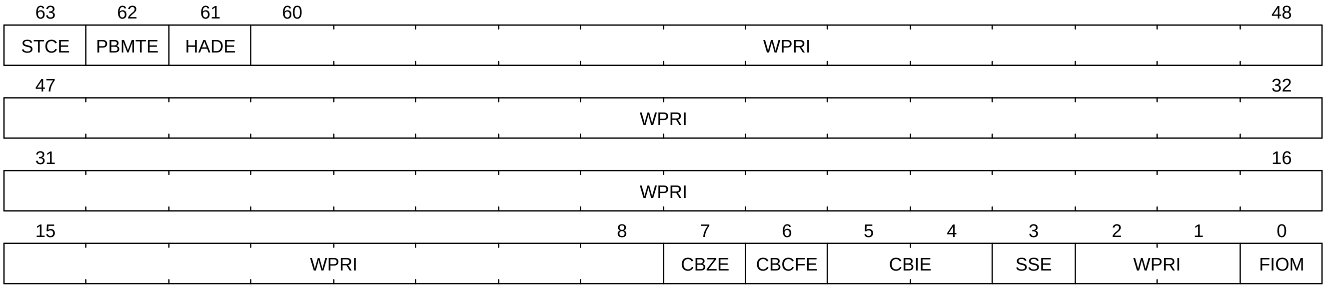

Figure 3. Hypervisor environment configuration register (`henvcfg`) for MXLEN=64

Zicfiss extension introduces the `SSE` field (bit 3) in `henvcfg`. When `SSE` field is 1, the Zicfiss extension is enabled in VS-mode. When `SSE` field is 0, the Zicfiss extension is not enabled in VS-mode and the following rules apply when `V=1`.

- Attempts to access the `ssp` CSR raise an illegal instruction exception.
- The 32-bit Zicfiss instructions revert to their Zimop defined behavior.
- The 16-bit Zicfiss instructions revert to their Zcmop defined behavior.
- The `pte.xwr=010b` encoding in VS-stage page tables is reserved.
- The `senvcfg.SSE` field is read-only zero.

2.1.4. 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. The bits 1:0 of `ssp` are read-only zero. If the UXLEN or SXLEN may never be 32, then the bit 2 is also read-only zero.

2.1.5. Machine Security Configuration (`mseccfg`)

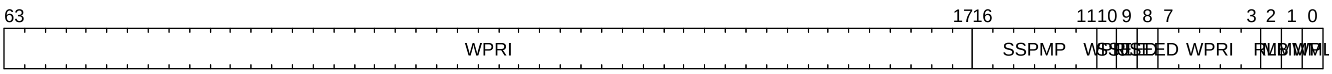

Figure 4. Machine security configuration register (`mseccfg`) when `MXLEN=64`

The Zicfiss extension introduces the `SSPMP` WARL field in `mseccfg`. The `SSPMP` field identifies a PMP entry as the shadow stack memory region for M-mode use. The rules enforced by PMP for M-mode shadow stack memory accesses are specified in Section 2.8.2.

2.2. Shadow-Stack-Enabled (SSE) state

The term `xSSE` is used to determine if backward-edge CFI using shadow stacks provided by the Zicfiss extension is enabled at a privilege mode and it is determined as follows:

Table 1. `xSSE` determination

Privilege Mode	xSSE
M	`1`
S or HS	`menvcfg.SSE`
VS	`henvcfg.SSE`
U or VU	`senvcfg.SSE`

> Activating Zicfiss in U-mode must be done explicitly per process. Not activating Zicfiss at U-mode for a process when that application is not compiled with Zicfiss allows it to invoke shared libraries that may contain Zicfiss instructions. The Zicfiss instructions in the shared library revert to their Zimop/Zcmop-defined behavior in this case.
>
> When Zicfiss is enabled in S-mode it is benign to use an operating system that is not compiled with Zicfiss instructions. Such an operating system that does not use backward-edge CFI for S-mode execution may still activate Zicfiss for U-mode applications.
>
> When Zicfiss is implemented, the extension is always enabled in M-mode. However, it is benign to use M-mode firmware that has not been compiled with Zicfiss instructions. Such M-mode firmware that does not use backward-edge CFI for M-mode execution may still enable the use of Zicfiss by lower privilege modes.
>
> When programs that use Zicfiss instructions are installed on a processor that supports the Zicfiss extension but the extension is not enabled at the privilege mode where the program executes, the program continues to function correctly but without backward-edge CFI protection as the Zicfiss instructions will revert to their Zimop/Zcmop-defined behavior.
>
> When programs that use Zicfiss instructions are installed on a processor that does not support the Zicfiss extension but supports the Zimop and Zcmop extensions, the programs continues to function correctly but without backward-edge CFI protection as the Zicfiss instructions will revert to their Zimop/Zcmop-defined behavior.
>
> On processors that do not support Zimop/Zcmop extensions, all Zimop/Zcmop code points including those used for Zicfiss instructions may cause an illegal instruction exception. Execution of programs that use these instructions on such machines is not supported.

2.3. Push to 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.

31–25	24–20	19–15	14–12	11–7	6–0
1000001	rs2	rs1	funct3	rd	opcode
sspush x1 sspush x5	00001 00101	00000	100	00000	SYSTEM

15–13	12	11–7	6–2	1–0
011	0	rd	00000	op
c.sspush x1		00001		C1

Only `x1` and `x5` encodings are supported as `rs2` for `sspush`. Zicfiss provides 16-bit versions of the `sspush x1` instruction using the Zcmop defined `c.mop.0` encoding. The `c.sspush x1` expands to `sspush x1`.

The `sspush` instruction and its compressed form `c.sspush` can be used, to push a link register on the shadow stack. 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` and `c.sspush` instructions require the virtual address in `ssp` to have a shadow stack attribute (see Section 2.8). Correct execution of `sspush` and `c.sspush` requires that `ssp` refers to idempotent memory. If the memory referenced by `ssp` is not idempotent, then the `sspush`/`c.sspush` instructions cause a store/AMO access fault exception. If the virtual address in `ssp` is not `XLEN` aligned, then the `sspush`/`c.sspush` instructions cause a store/AMO access fault exception.

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

Listing 1. `sspush` and `c.sspush` operation

```
If (xSSE == 1)
    mem[ssp - (XLEN/8)] = X(src)  # Store src value to ssp - XLEN/8
    ssp = ssp - (XLEN/8)          # decrement ssp by XLEN/8
endif
```

2.4. Pop from 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
sspopchk x1 sspopchk x5	00001 00101	100	00000 00000	SYSTEM

15–13	12	11–7	6–2	1–0
011	0	rd	00000	op
c.sspopchk x5		00101		C1

Only `x1` and `x5` encodings are supported as `rs1` for `sspopchk`. Zicfiss provides a 16-bit version of the `sspopchk x5` using Zcmop define `c.mop.2` encoding. The `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 of non-leaf functions. Such programs when returning from the non-leaf function pop the link register from the regular 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 on the regular 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 and if not cause a software integrity fault exception with `*tval` set to "shadow stack fault (code=3)".

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 Unprivileged specification [1].
>
> 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.
>
> Compilers when generating code with backward-edge CFI must protect the link register, e.g. `x1` and/or `x5`, from arbitrary modification by not emitting unsafe code sequences.

> 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 Zicfiss 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 Zicfiss 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.
>
> The prologue and epilogue of a non-leaf function in shadow stack mode is as follows:

```
function_entry:
    addi sp,sp,-8  # push link register x1
    sd x1,(sp)     # on data stack
    sspush x1      # push link register x1 on shadow stack
     :
     :
    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
    ret
```

These examples illustrate the use of `x1` register as the link register. Alternatively, the `x5` register may also be used 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 `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 `sspopchk` and `c.sspopchk` instructions require the virtual address in `ssp` to have a shadow stack attribute (see Section 2.8). Correct execution of `sspopchk` and `c.sspopchk` requires that `ssp` refers to idempotent memory. If the memory reference by `ssp` is not idempotent, then the instructions cause a load access fault exception. If the virtual address in `ssp` is not `XLEN` aligned, then `sspopchk` and `c.sspopchk` instructions cause a load access fault exception

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. Instructions that load from the shadow stack 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.

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

Listing 2. `sspopchk` and `c.sspopchk` operation

```
if (xSSE == 1)
    temp = mem[ssp]            # Load temp from address in ssp and
    if temp != X(src)          # Compare temp to value in src and
                               # cause an SW integrity fault exception
                               # if they are not bitwise equal.
                               # Only x1 and x5 may be used as src
       Raise software error exception
    else
       ssp = ssp + (XLEN/8)    # increment ssp by XLEN/8.
    endif
endif
```

The `ssp` is incremented by `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
> ```

2.5. Load from the shadow stack

The `sslw` instruction can be used, when effective xlen is 32, to load a return address from the shadow stack into a link register. The `ssld` instruction can be used, when effective xlen is 64, to load a return address from the shadow stack into a link register.

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
100000011100	rs1	funct3	rd	opcode
sslw x1	00000	100	00001	SYSTEM
sslw x5	00000		00101	
ssld x1	00000		00001	
ssld x5	00000		00101	

The `sslw` and `ssld` are both encoded identically. They have different mnemonics to illustrate that the instructions operates on a *word* when XLEN is 32 and on a *doubleword* when XLEN is 64.

The `sslw` and `ssld` instructions require the virtual address in `ssp` to have a shadow stack attribute (see Section 2.8). Correct execution of `sslw` and `ssld` requires that `ssp` refers to idempotent memory. If the memory reference by `ssp` is not idempotent, then the instructions cause a load access fault exception. If the virtual address in `ssp` is not `XLEN` aligned, then `sslw` and `ssld` instructions cause a load access fault exception

The operation of the `sslw` and `ssld` instructions is as follows:

Listing 3. `sslw` and `ssld` operation

```
if (xSSE == 1)
    X(dst) = mem[ssp]           # Load dst with XLEN bits from address in ssp
                                # Only x1 and x5 may be used as dst
else
    X(dst) = 0
endif
```

Store-to-load forwarding is a common technique employed by high-performance processor implementations. Zicfiss implementations may prevent forwarding from a non-shadow-stack store to `sslw`/`ssld`/`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 `sslw`/`ssld` /`sspopchk`/`c.sspopchk`.

2.6. Increment the shadow stack pointer

The `ssincp` instruction adds `XLEN/8` to the `ssp`. This instruction may be used to remove a shadow stack frame from the shadow stack. Zicfiss provides a 16-bit version of the `ssincp` using Zcmop define `c.mop.1` encoding. The `c.ssincp` expands to `ssincp`.

31 20	19 15	14 12	11 7	6 0
100000011100	rs1	funct3	rd	opcode
ssincp	00000	100	00000	SYSTEM

15 13	12	11 7	6 2	1 0
011	0	rd	00000	op
c.ssincp		00011		C1

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

Listing 4. `ssincp` and `c.ssincp` operation

```
if (xSSE == 1)
    ssp = ssp + XLEN/8
endif
```

The prologue and epilogue of a non-leaf function when operating in control stack mode is as follows:

```
    function_entry:
        sspush x1      # push link register x1 on shadow stack
         :
         :
        ssld x1        # load return address from shadow stack
        ssincp         # increment ssp by (XLEN/8)
        ret
```

2.7. Read `ssp` into a register

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

31–20	19–15	14–12	11–7	6–0
100000011101	00000	funct3	rd	opcode
ssrdp		100	dst	SYSTEM

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

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

Listing 5. `ssrdp` operation

```
If (xSSE == 1)
    X(dst) = ssp
else
    X(dst) = 0
endif
```

The property of Zimop writing 0 to the `rd` when the extension using Zimop is not implemented, enabled for use, or not enabled may be used by to determine if Zicfiss extension is enabled. For example, functions that unwind shadow stacks may skip over the unwind actions by dynamically detecting if the Zicfiss extension is enabled.

An example sequence such as the following may be used:

```
    ssrdp t0                        # mv ssp to t0
    beqz t0, zicfiss_not_enabled    # zero is not a valid shadow stack
                                    # pointer by convention
    # Zicfiss is enabled
    :
    :
zicfiss_not_active:
```

Operating systems and runtimes must not locate shadow stacks at address 0 to assist with the use of such code sequences.

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("ssrdp %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("ssrdp %0" : "=r"(cur_ssp):);
    // Skip the unwind if backward-edge CFI not enabled
    asm("beqz %0, back_cfi_not_enabled" : "=r"(cur_ssp):);
    // Unwind the frames in a loop
    while ( jmp_buf->saved_ssp > cur_ssp ) {
        // advance by a maximum of 4K at a time to avoid
        // unwinding past bounds of the shadow stack
        cur_ssp = ( (jmp_buf->saved_ssp - cur_ssp) >= 4096 ) ?
                  (cur_ssp + 4096) : jmp_buf->saved_ssp;
        asm("csrw ssp, %0" : :  "r" (cur_ssp));
        // Test if unwound past the shadow stack bounds
        asm("ssld x5");
    }
back_cfi_not_enabled:
    :
}
```

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 active 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 thereby create a checkpoint.

An example sequence to restore the shadow stack pointer from the new shadow stack and save the old shadow stack pointer on the old shadow stack is as follows:

```
# The a0 register holds the pointer to checkpoint at the top of the
# new shadow stack. x5 at end of the sequence has the address of the
# checkpoint created on current shadow stack. This should be saved
# away in a context structure to restore later.
switch_shadow_stack:
    ssrdp t1                 # read current ssp
    beqz t1, ss_not_enabled  # skip if shadow stacks not enabled
    csrw ssp, a0             # ssp = checkpoint address
    mv t0, a0                # checkpoint value == checkpoint address
    sspopchk t0              # pop and check the checkpoint value
    li t0, 0                 # clear the checkpoint
    sspush t0                # by pushing zero in its place
    addi a0, a0, (XLEN/8)    # a0 now has the top of new shadow stack
    mv t0, t1                # t1 holds the top of old shadow stack
    csrw ssp, t0             # point ssp to old top of old shadow stack
    addi x5, x5, -(XLEN/8)   # checkpoint val = (ssp - XLEN/8)
    sspush x5                # store checkpoint to (ssp - XLEN/8)
    csrw ssp, a0             # setup new ssp
ss_not_enabled:
```

2.8. 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.

2.8.1. 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 `c.sspush`. The `sslw`, `ssld`, `sspopchk`, and `c.sspopchk` instructions can only load from shadow stack memory.

The `sspush` and `c.sspush` instructions perform a store. The `sslw`, `ssld`, `sspopchk`, and `c.sspopchk` instructions perfom a load.

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

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

The encoding `R=0`, `W=1`, and `X=0`, is defined to represent a shadow stack page. When `menvcfg.SSE=0`, this encoding remains reserved. When `V=1` and `henvcfg.SSE=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 `c.sspush` 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. `c.sspush`, and `sspush` cause a store/AMO access fault.
   b. `sslw`, `ssld`, `c.sspopchk`, and `sspopchk` cause a load access fault.

> Stores to shadow stack by instructions other than `sspush`, and `c.sspush` cause a store/AMO access fault exception, rather than a store/AMO page fault exception, to indicate fatality.
>
> If a store/AMO 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` or `c.sspush` 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` or `c.sspush`, 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 `V=0` and `menvcfg.SSE` is 1 or if `V=1` and `henvcfg.SSE` 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`, `sslw`, `ssld`, `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 and Svnapot extensions are supported for shadow stack pages.

> 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.
>
> 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 cause an access fault exception.

The G-stage address translation and protections remain unaffected by Zicfiss extension. When G-stage page tables are active, the `sslw`, `ssld`, `c.sspopchk`, and `sspopchk` instructions require the G-stage page table to have read permission for the accessed memory, whereas the `c.sspush`, and `sspush` instructions require write permission. The `xwr == 010b` encoding in the G-stage PTE remains reserved.

> 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.

2.8.2. PMP extension for shadow stack

When privilege mode is less than M, the PMP region accessed by `sspush` and `c.sspush` must provide write permission and the PMP region accessed by `sslw`, `ssld`, `c.sspopchk`, and `sspopchk` must provide read permission.

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

The M-mode memory accesses by `sslw`, `ssld`, `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 not zero, the following rules are additionally enforced for M-mode memory accesses:

- `sspush`, `c.sspush`, `sslw`, `ssld`, `sspopchk`, and `c.sspopchk` instructions must match the PMP entry identified by `SSPMP` else an access fault exception corresponding to the access type occurs.
- Write by instructions other than `sspush` and `c.sspush` that match the PMP entry identified by `SSPMP` cause an store/AMO access fault 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 3. Landing pad (Zicfilp)

To enforce forward-edge control-flow integrity, the Zicfilp extension introduces a landing pad (`lpad`) instruction. The `lpad` instruction that must be placed at the program locations that are valid targets of indirect jumps or calls. The `lpad` instruction (See Section 3.3) is encoded using the `AUIPC` major opcode with `rd=x0`.

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 an indirect jump. The `ELP` state can be one of:

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

The `ELP` state is initialized to `NO_LP_EXPECTED` by the hardware upon reset.

The Zicfilp extension, when enabled, determines if an indirect call or an indirect jump must land on a landing pad, as specified in Listing 6. If `is_lp_expected` is 1, then the hart updates the `ELP` to `LP_EXPECTED`.

Listing 6. Landing pad expected determination

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

An indirect branch using `JALR`, `C.JALR`, or `C.JR` with `rs1` as `x7` is termed a software guarded branch. Such branches do not need to land on a `lpad` instruction and thus do not set `ELP` to `LP_EXPECTED`.

When the register source is a link register and the register destination is `x0` then its a return from a procedure and does not require a landing pad at the target.

When the register source and register destination are both link registers then its a semantically-direct-call. For example, the `call offset` pseudoinstruction may expand to a two instruction sequence composed of a `lui ra, imm20` or a `auipc ra, imm20` instruction followed by a `jalr ra, imm12(ra)` instruction where `ra` is the link register (either `x1` or `x5`). Since the address of the procedure was not explicitly taken and the computed address is not obtained from mutable memory, such semantically-direct calls do not require a landing pad to be placed at the target. Compilers and JITers must only use the semantically-direct calls only if when the `rs1` was computed as a PC-relative or an absolute offset to the symbol.

The `tail offset` pseudoinstruction used to tail call a far-away procedure may also be expanded to a two instruction sequence composed of a `lui x7, imm20` or `auipc x7, imm20` followed by a `jalr x0, x7`. Since the address of the procedure was not explicitly taken and the computed address is not obtained from mutable memory, such semantically-direct tail-calls do not require a landing pad to be placed at the target.

> Software guarded branches may also be used by compilers to generate 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.).

The landing pad may be labeled. Zicfilp extension designates the register `x7` for use as the landing pad label register. To support labeled landing pads, the indirect call/jump sites establish an expected landing pad label (e.g., using the `lui` instruction) in the bits 31:12 of the `x7` register. The `lpad` instruction is encoded with a 20-bit immediate value called the landing-pad-label (`LPL`) that is matched to the expected landing pad label. When `LPL` is encoded as zero, the `lpad` instruction does not perform the label check and in programs built with this single label mode of operation the indirect call/jump sites do not need to establish an expected landing pad label value in `x7`.

When `ELP` is set to `LP_EXPECTED`, if the next instruction in the instruction stream is not 4-byte aligned, or is not `lpad`, or if the landing pad label encoded in `lpad` is not zero and does not match the expected landing pad label in bits 31:12 of the `x7` register, then a software error exception (cause=18) with `*tval` set to "landing pad fault (code=2)" is raised else the `ELP` is updated to `NO_LP_EXPECTED`.

> The tracking of `ELP` and the requirement for a landing pad instruction 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.
>
> The `lpad` requires a 4-byte alignment to address the concatenation of two instructions `A` and `B` accidentally forming an unintended landing pad in the program. For example, consider a 32-bit instruction where the bytes 3 and 2 have a pattern of `?017h` (for example, the immediate fields of a `lui`, `auipc`, or a `jal` instruction), followed by a 16-bit or a 32-bit instruction. When patterns that can accidentally form a valid landing pad are detected, the assembler or linker can force instruction `A` to be aligned to a 4-byte boundary to force the unintended `lpad` pattern to become misaligned and thus not a valid landing pad or may use an alternate register allocation to prevent the accidental landing pad.

3.1. Zicfilp CSRs

This chapter specifies the CSR state of the Zicfilp extension.

3.1.1. Machine environment configuration registers (`menvcfg and menvcfgh`)

63	62	61	60–48
STCE	PBMTE	HADE	WPRI

47–32
WPRI

31–16
WPRI

15–8	7	6	5–4	3	2	1	0
WPRI	CBZE	CBCFE	CBIE	WPRI	LPE	WPRI	FIOM

Figure 5. Machine environment configuration register (`menvcfg`) for MXLEN=64

Zicfilp extension introduces the `LPE` field (bit 2) in `menvcfg`. When `LPE` field is 1, the Zicfilp extension is enabled in S-mode. When `LPE` field is 0, the Zicfilp extension is not enabled in S-mode and the following rules apply to S-mode:

- The hart does not update the expected landing pad (`ELP`) state and the `ELP` state is always `NO_LP_EXPECTED`.
- The `lpad` instruction executes as a no-op.

3.1.2. Supervisor environment configuration registers (`senvcfg`)

63–48
WPRI

47–32
WPRI

31–16
WPRI

15–8	7	6	5–4	3	2	1	0
WPRI	CBZE	CBCFE	CBIE	WPRI	LPE	WPRI	FIOM

Figure 6. Supervisor environment configuration register (`senvcfg`) when `SXLEN=64`

Zicfilp extension introduces the `LPE` field (bit 2) in `senvcfg`. When `LPE` field is 1, the Zicfilp extension is enabled in VU/U-mode. When `LPE` field is 0, the Zicfilp extension is not enabled in VU/U-mode and the following rules apply to VU/U-mode:

- The hart does not update the expected landing pad (`ELP`) state and the `ELP` state is always `NO_LP_EXPECTED`.
- The `lpad` instruction executes as a no-op.

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

63	62	61	60–48
STCE	PBMTE	HADE	WPRI

47–32
WPRI

31–16
WPRI

15–8	7	6	5–4	3	2	1	0
WPRI	CBZE	CBCFE	CBIE	WPRI	LPE	WPRI	FIOM

Figure 7. Hypervisor environment configuration register (`henvcfg`) for MXLEN=64

Zicfilp extension introduces the `LPE` field (bit 2) in `henvcfg`. When `LPE` field is 1, the Zicfilp extension is enabled in VS-mode. When `LPE` field is 0, the Zicfilp extension is not enabled in VS-mode and the following rules apply to VS-mode:

- The hart does not update the expected landing pad (`ELP`) state and the `ELP` state is always `NO_LP_EXPECTED`.
- The `lpad` instruction executes as a no-op.

3.1.4. Machine status registers (`mstatus`)

63	62–48
SD	WPRI

47–42	41	40–38	37	36	35–34	33–32
WPRI	MPELP	WPRI	MBE	SBE	SXL[1:0]	UXL[1:0]

31–24	23	22	21	20	19	18	17	16
WPRI	SPELP	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 8. Machine-mode status register (`mstatus`) for RV64

The Zicfilp extension introduces the `SPELP` (bit 23) and `MPELP` (bit 41) fields that hold the previous `ELP`, and are updated as specified in Section 3.4. 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.

3.1.5. Supervisor status registers (`sstatus`)

63	62–48
SD	WPRI

47–34	33–32
WPRI	UXL[1:0]

31–24	23	22–20	19	18	17	16
WPRI	SPELP	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 9. Supervisor-mode status register (`sstatus`) when `SXLEN=64`

Access to the `SPELP` field introducecd by Zicfilp accesses the homonymous fields of `mstatus` when `V=0` and the homonymous fields of `vsstatus` when `V=1`.

3.1.6. Virtual supervisor status registers (`vsstatus`)

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

Figure 10. Virtual supervisor status register (`vsstatus`) when `VSXLEN=64`

The Zicfilp extension introduces the `SPELP` (bit 23) field that hold the previous `ELP`, and is updated as specified in Section 3.4. The `SPELP` field is encoded as follows:

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

3.1.7. Machine Security Configuration (`mseccfg`)

63–48	WPRI
47–32	WPRI
31–16	WPRI
15–11	WPRI
10	MLPE
9	SSEED
8	USEED
7–3	WPRI
2	RLB
1	MMWP
0	MML

Figure 11. Machine security configuration register (`mseccfg`) when `MXLEN=64`

The Zicfilp extension introduces the `MLPE` (bit 10) field in `mseccfg`. When `MLPE` field is 1, Zicfilp extension is enabled in M-mode. When `MLPE` field is 0, the Zicfilp extension is not enabled in M-mode and the following rules apply to M-mode.

- The hart does not update the expected landing pad (`ELP`) state and the `ELP` state is always `NO_LP_EXPECTED`.
- The `lpad` instruction executes as a no-op.

3.1.8. Debug Control and Status (`dcsr`)

31–28	debugver
27–24	0
23–19	0
18	pelp
17	ebreakvs
16	ebreakvu
15	ebreakm
14	0
13	ebreaks
12	ebreaku
11	stepie
10	stopcount
9	stoptime
8–6	cause
5	v
4	mprven
3	nmip
2	step
1–0	prv

Figure 12. Debug Control and Status (`dcsr`)

The Zicfilp extension introduces the `pelp` (bit 18) in `dcsr`. The `pelp` field holds the previous `ELP`, and is updated as specified in Section 3.4. The `pelp` field is encoded as follows:

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

3.2. Landing-Pad-Enabled (LPE) state

The term `xLPE` is used to determine if forward-edge CFI using landing pads provided by the Zicfilp extension is enabled at a privilege mode and it is determined as follows:

Table 2. `xLPE` determination

Privilege Mode	xLPE
M	`mseccfg.MLPE`
S or HS	`menvcfg.LPE`
VS	`henvcfg.LPE`
U or VU	`senvcfg.LPE`

> The Zicfilp must be explicitly enabled for use at each privilege mode.
>
> Programs compiled with the `lpad` instruction continue to function correctly, but without forward-edge CFI protection, when the Zicfilp extension is not implemented or is not enabled.

3.3. Landing pad instruction

When Zicfilp is enabled, `lpad` is the only instruction allowed to execute when the `ELP` state is `LP_EXPECTED`. If Zicfilp is not enabled then the instruction is a no-op. If Zicfilp is enabled, the `lpad` instruction causes a software error exception with `*tval` set to "landing pad fault (code=2)" if any of the following conditions are true:

- The `pc` is not 4-byte aligned.
- The `ELP` is `LP_EXPECTED` and the `LPL` is not zero and the `LPL` does not match the expected landing pad label in bits 31:12 of the `x7` register.

If the instruction causes an software error exception, the `ELP` does not change. The behavior of the trap caused by this software error exception is specified in section Section 3.4. If a software error exception is not caused then the `ELP` is updated to `NO_LP_EXPECTED`.

31 — 12	11 — 7	6 — 0
LPL	rd	opcode
	00000	AUIPC

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

Listing 7. `lpad` operation

```
if (xLPE != 0)
    // If PC not 4-byte aligned then software integrity fault
    if pc[1:0] != 0
```

```
        Cause software error exception
    // If landing pad label not matched -> software integrity fault
    else if (inst.LPL != x7[31:12] && inst.LPL != 0 && ELP == LP_EXPECTED)
        Cause software error exception
    else
        ELP = NO_LP_EXPECTED
else
    no-op
endif
```

3.4. 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 higher than that of an software error exception with `*tval` set to "landing pad fault (code=2)" (See Table 3.7 of Privileged Specification [2]).

The software error exception caused by Zicfilp has higher priority than an illegal instruction exception but lower priority than instruction access fault.

The software error exception due to the instruction not being an `lpad` instruction when `ELP` is `LP_EXPECTED` or an software error exception caused by the `lpad` instruction itself (See Section 3.3) leads to a trap being delivered to the same or to a higher privilege mode.

In such cases, the `ELP` prior to the trap, the previous `ELP`, must be preserved by the trap delivery such that it can be restored on a return from the trap. To store the previous `ELP` state on trap delivery to M-mode, a `MPELP` bit is provided in the `mstatus` CSR. To store the previous `ELP` state on trap delivery to S/HS-mode, a `SPELP` bit is provided in the `mstatus` CSR. 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 store the previous `ELP` state on transition to Debug Mode, a `pelp` bit is defined in the `dcsr` register.

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. An `xRET` instruction sets the `ELP` to `xPELP`, and sets `xPELP` to `NO_LP_EXPECTED`.

Upon entry into Debug Mode, the `pelp` bit in `dcsr` is updated with the `ELP` at the privilege level the hart was previously in and the `ELP` is set to `NO_LP_EXPECTED`. When a hart resumes from Debug Mode, the `ELP` is changed to that specified by `pelp` in `dcsr`.

> The trap handler in privilege mode `x` must save the `xPELP` bit and the `x7` 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 `x7`

> 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.