RISC-V Shadow Stacks and Landing Pads

Version v1.0, 2024-07-03: Ratified

Table of Contents

Preamble

This document is in the Ratified state

No changes are allowed. Any desired or needed changes can be the subject of a follow-on new extension. Ratified extensions are never revised.

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, Argyro Palli, 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, Rui Ueyama, Sami Tolvanen, Sotiris Ioannidis, Stefan O'Rear, Thurston Dang, Tsukasa OI, Vedvyas Shanbhogue

Chapter 1. Introduction

Control-flow Integrity (CFI) capabilities help defend against Return-Oriented Programming (ROP) and 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 provides 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 3 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 instructions perform 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 and provide backward-edge and forward-edge control flow integrity respectively. The Zicfilp extension is specified in Chapter 3 and the Zicfiss extension is specified in Chapter 2.

Chapter 2. Shadow Stack (Zicfiss)

The Zicfiss extension introduces a shadow stack to enforce backward-edge control-flow integrity. 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 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, similar to the regular stack, grows downwards, 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, which is the address of the last element stored on the 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.

When Zicfiss is enabled, each function that needs to spill the link register, typically non-leaf functions, store 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 returns (the epilogue), the function loads the link register from the regular stack and the shadow copy of the link register from the shadow stack. Then, 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 a software-check exception.

The Zicfiss instructions are encoded using a subset of May-Be-Operation instructions defined by the Zimop and Zcmop extensions [2]. 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. 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 extension 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, could either deny the request or deactivate the Zicfiss extension for the application. It is strongly recommended that the policy enforces a strict security posture and denies the request.

The Zicfiss extension depends on the Zicsr and Zimop extensions. Furthermore, if the Zcmop extension is implemented, the Zicfiss extension also provides the `C.SSPUSH` and `C.SSPOPCHK` instructions. Moreover, use of Zicfiss in U-mode requires S-mode to be implemented. Use of Zicfiss in M-mode is not supported.

2.1. Zicfiss Instructions Summary

The Zicfiss extension introduces the following instructions:

- Push to the shadow stack (See Section 2.4)
  - `SSPUSH x1` and `SSPUSH x5` - encoded using `MOP.RR.7`
  - `C.SSPUSH x1` - encoded using `C.MOP.1`
- Pop from the shadow stack (See Section 2.5)
  - `SSPOPCHK x1` and `SSPOPCHK x5` - encoded using `MOP.R.28`
  - `C.SSPOPCHK x5` - encoded using `C.MOP.5`
- Read the value of `ssp` into a register (See Section 2.6)
  - `SSRDP` - encoded using `MOP.R.28`
- Perform an atomic swap from a shadow stack location (See Section 2.7)
  - `SSAMOSWAP.W` and `SSAMOSWAP.D`

Zicfiss does not use all encodings of `MOP.RR.7` or `MOP.R.28`. When a `MOP.RR.7` or `MOP.R.28` encoding is not used by the Zicfiss extension, the corresponding instruction adheres to its Zimop-defined behavior, unless redefined by another extension.

If a shadow stack (SS) instruction raises an access-fault, page-fault, or guest-page-fault exception that is supposed to indicate the original instruction type (load or store/AMO), then the reported exception cause is respectively a store/AMO access fault (code 7), a store/AMO page fault (code 15), or a store/AMO guest-page fault (code 23). For shadow stack instructions, the reported instruction type is always as though it were a store or AMO, even for instructions `SSPOPCHK` and `C.SSPOPCHK` that only read from memory and do not write to it.

When Zicfiss is implemented, the existing "store/AMO" exceptions can be thought of as "store/AMO/SS" exceptions, indicating that the trapping instruction is either a store, an AMO, or a shadow stack instruction.

2.2. Zicfiss CSRs

This section specifies the CSR state of the Zicfiss extensions.

2.2.1. Machine Environment Configuration Register (`menvcfg`)

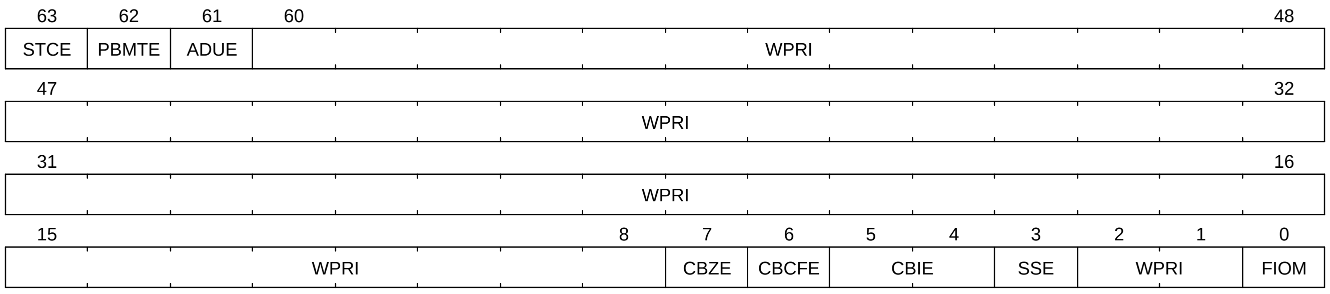


Figure 1. Machine environment configuration register (`menvcfg`)

The Zicfiss extension adds the `SSE` field (bit 3) to `menvcfg`. When the `SSE` field is set to 1 the Zicfiss extension is activated in S-mode.

When `SSE` field is 0, the following rules apply to privilege modes that are less than M:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- The `pte.xwr=010b` encoding in VS/S-stage page tables becomes reserved.
- The `henvcfg.SSE` and `senvcfg.SSE` fields will read as zero and are read-only.
- `SSAMOSWAP.W/D` raises an illegal-instruction exception.

2.2.2. Supervisor Environment Configuration Register (`senvcfg`)

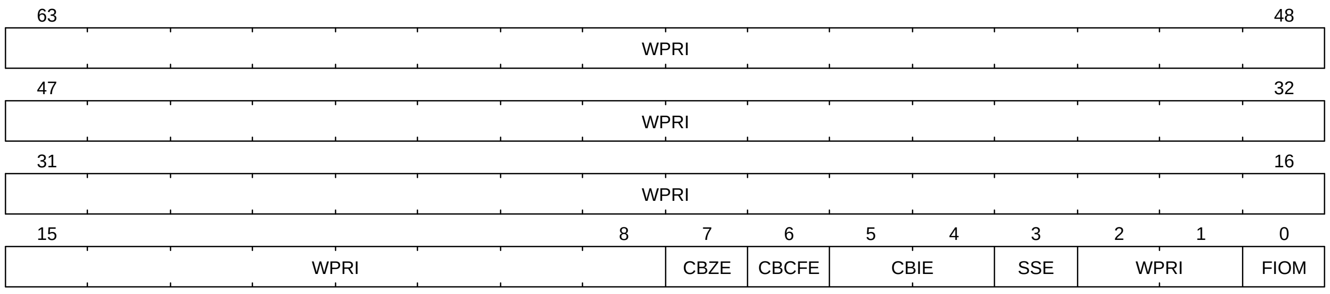


Figure 2. Supervisor environment configuration register (`senvcfg`)

Zicfiss extension introduces the `SSE` field (bit 3) in `senvcfg`. If the `SSE` field is set to 1, the Zicfiss extension is activated in VU/U-mode. When the `SSE` field is 0, the Zicfiss extension remains inactive in VU/U-mode, and the following rules apply:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- When `menvcfg.SSE` is one, `SSAMOSWAP.W/D` raises an illegal-instruction exception in U-mode and a virtual instruction exception in VU-mode.

2.2.3. Hypervisor Environment Configuration Register (`henvcfg`)

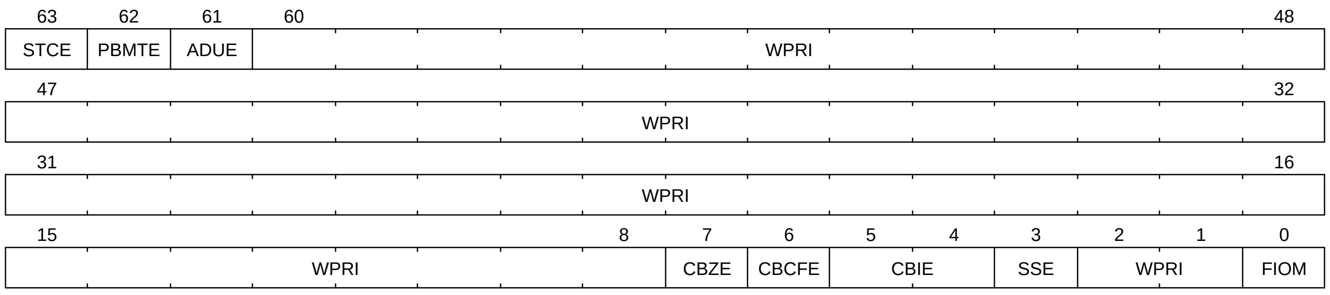


Figure 3. Hypervisor environment configuration register (`henvcfg`)

Zicfiss extension introduces the `SSE` field (bit 3) in `henvcfg`. If the `SSE` field is set to 1, the Zicfiss extension is activated in VS-mode. When the `SSE` field is 0, the Zicfiss extension remains inactive in VS-mode, and the following rules apply when `V=1`:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- The `pte.xwr=010b` encoding in VS-stage page tables becomes reserved.
- The `senvcfg.SSE` field will read as zero and is read-only.
- When `menvcfg.SSE` is one, `SSAMOSWAP.W/D` raises a virtual instruction exception.

2.2.4. 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`). The CSR address is 0x011. 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.

Attempts to access the `ssp` CSR may result in either an illegal-instruction exception or a virtual instruction exception, contingent upon the state of the `xenvcfg.SSE` fields. The conditions are specified as follows:

- If the privilege mode is less than M and `menvcfg.SSE` is 0, an illegal-instruction exception is raised.
- Otherwise, if in U-mode and `senvcfg.SSE` is 0, an illegal-instruction exception is raised.
- Otherwise, if in VS-mode and `henvcfg.SSE` is 0, a virtual instruction exception is raised.
- Otherwise, if in VU-mode and either `henvcfg.SSE` or `senvcfg.SSE` is 0, a virtual instruction exception is raised.
- Otherwise, the access is allowed.

2.3. 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.

When S-mode is implemented, it is determined as follows:

Table 1. `xSSE` determination when S-mode is implemented

Privilege Mode	xSSE
M	`0`
S or HS	`menvcfg.SSE`
VS	`henvcfg.SSE`
U or VU	`senvcfg.SSE`

When S-mode is not implemented, then `xSSE` is 0 at both M and U privilege modes.

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 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.

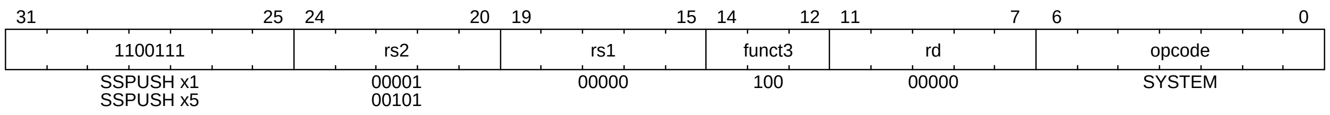
When programs that use Zicfiss instructions are installed on a processor that does not support the Zicfiss extension but supports the Zimop and Zcmop extensions, the programs continues to function correctly but without backward-edge CFI protection as the Zicfiss instructions will revert to their Zimop/Zcmop-defined behavior.

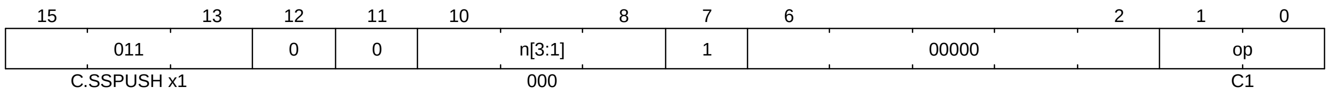
On processors that do not support Zimop/Zcmop extensions, all Zimop/Zcmop code points including those used for Zicfiss instructions may cause an illegal-instruction exception. Execution of programs that use these instructions on such machines is not supported.

Activating Zicfiss in M-mode is currently not supported. Additionally, when S-mode is not implemented, activation in U-mode is also not supported. These functionalities may be introduced in a future standard extension.

2.4. Push to the Shadow Stack

A shadow stack push operation is defined as decrement of the `ssp` by `XLEN/8` followed by a store of the value in the link register to memory at the new top of the shadow stack.





Only `x1` and `x5` registers are supported as `rs2` for `SSPUSH`. Zicfiss provides a 16-bit version of the `SSPUSH x1` instruction using the Zcmop defined `C.MOP.1` 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 perform a store identically to the existing store instructions, 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:

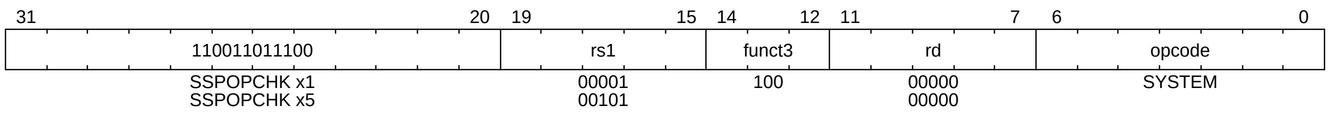
Listing 1. `SSPUSH` and `C.SSPUSH` operation

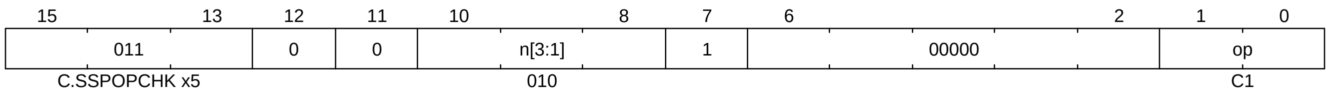
```
if (xSSE == 1)
    mem[ssp - (XLEN/8)] = X(src)  # Store src value to ssp - XLEN/8
    ssp = ssp - (XLEN/8)          # decrement ssp by XLEN/8
endif
```

The `ssp` is decremented by `SSPUSH` and `C.SSPUSH` only if the store to the shadow stack completes successfully.

2.5. Pop from the Shadow Stack

A shadow stack pop operation is defined as an `XLEN` wide read from the current top of the shadow stack followed by an increment of the `ssp` by `XLEN/8`.





Only `x1` and `x5` registers are supported as `rs1` for `SSPOPCHK`. Zicfiss provides a 16-bit version of the `SSPOPCHK x5` using the Zcmop defined `C.MOP.5` encoding. The `C.SSPOPCHK x5` expands to `SSPOPCHK x5`.

Programs with a shadow stack push the return address onto the regular stack as well as the shadow stack in the prologue of non-leaf functions. When returning from these non-leaf functions, such programs 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-check exception with `xtval` 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 3 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.

Storing the return address on both stacks preserves the call stack layout and the ABI, while also allowing for the detection of corruption of the return address on the regular stack. The prologue and epilogue of a non-leaf function that uses shadow stacks is as follows:

```
    function_entry:
        addi sp,sp,-8  # push link register x1
        sd x1,(sp)     # on regular stack
        sspush x1      # push link register x1 on shadow stack
          :
        ld x1,(sp)     # pop link register x1 from regular stack
        addi sp,sp,8
        sspopchk x1    # fault if x1 not equal to shadow return address
        ret
```

This example illustrates the use of `x1` *register as the link register. Alternatively, the* `x5` *register may also be used as the link register.*

A leaf function, a function that does not itself make function calls, does not need to spill the link register. Consequently, the return value may be held in the link register itself for the duration of the leaf function's execution.

The `C.SSPOPCHK`, and `SSPOPCHK` instructions perform a load identically to the existing load instructions, 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 store/AMO access-fault exception. If the virtual address in `ssp` is not `XLEN` aligned, then `SSPOPCHK` and `C.SSPOPCHK` instructions cause a store/AMO 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 as 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 software-check exception
                              # if they are not bitwise equal.
                              # Only x1 and x5 may be used as src
       raise software-check exception
    else
       ssp = ssp + (XLEN/8)   # increment ssp by XLEN/8.
    endif
endif
```

If the value loaded from the address in `ssp` does not match the value in `rs1`, a software-check exception (cause=18) is raised with `xtval` set to "shadow stack fault (code=3)". The software-check exception caused by `SSPOPCHK`/ `C.SSPOPCHK` is lower in priority than a load/store/AMO access-fault exception.

The `ssp` is incremented by `SSPOPCHK` and `C.SSPOPCHK` only if the load from the shadow stack completes successfully and no software-check exception is raised.

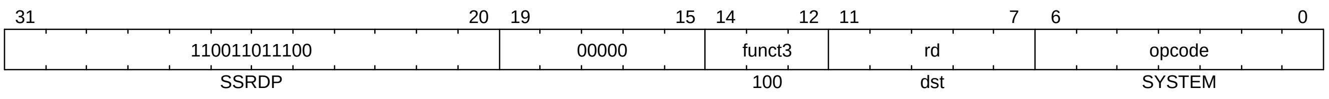
The use of the compressed instruction `C.SSPUSH x1` to push on the shadow stack is most efficient when the ABI uses `x1` as the link register, as the link register may then be pushed without needing a register-to-register move in the function prologue. To use the compressed instruction `C.SSPOPCHK x5`, 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 regular stack
        c.sspush x1      # push link register x1 on shadow stack
         :
        c.ld x5,(sp)     # pop link register x5 from regular stack
        c.addi sp,sp,8
        c.sspopchk x5    # fault if x5 not equal to shadow return address
        c.jr x5
```

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 the `SSPOPCHK` or the `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 `SSPOPCHK` or to `C.SSPOPCHK`.

2.6. Read ssp into a Register

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



Encoding rd as x0 is not supported for SSRDP.

The operation of the SSRDP instructions is as follows:

Listing 3. 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 or not active may be used by to determine if Zicfiss extension is active. For example, functions that unwind shadow stacks may skip over the unwind actions by dynamically detecting if the Zicfiss extension is active.

An example sequence such as the following may be used:

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

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

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. A guard page for a stack is a page that is not accessible by the process that owns the stack. To detect if the unwind occurs past the bounds of the shadow stack, the unwind may be done in maximal increments of 4 KiB, testing whether the* `ssp` *is 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 active
    asm("beqz %0, back_cfi_not_active" : "=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("sspush x5");
        asm("sspopchk x5");
    }
back_cfi_not_active:
    :
}
```

2.7. Atomic Swap from a Shadow Stack Location

31 27	26	25	24 20	19 15	14 12	11 7	6 0
01001	aq	rl	rs2	rs1	funct3	rd	opcode
SSAMOSWAP.W SSAMOSWAP.D			src	addr	010 011	dest	AMO

For RV32, `SSAMOSWAP.W` atomically loads a 32-bit data value from address of a shadow stack location in `rs1`, puts the loaded value into register `rd`, and stores the 32-bit value held in `rs2` to the original address in `rs1`. `SSAMOSWAP.D` (RV64 only) is similar to `SSAMOSWAP.W` but operates on 64-bit data values.

Listing 4. `SSAMOSWAP.W` for RV32 and `SSAMOSWAP.D` (RV64 only) operation

```
if privilege_mode != M && menvcfg.SSE == 0
    raise illegal-instruction exception
if S-mode not implemented
    raise illegal-instruction exception
else if privilege_mode == U && senvcfg.SSE == 0
    raise illegal-instruction exception
else if privilege_mode == VS && henvcfg.SSE == 0
    raise virtual instruction exception
else if privilege_mode == VU && senvcfg.SSE == 0
    raise virtual instruction exception
else
    X(rd) = mem[X(rs1)]
    mem[X(rs1)] = X(rs2)
endif
```

For RV64, `SSAMOSWAP.W` atomically loads a 32-bit data value from address of a shadow stack location in `rs1`, sign-extends the loaded value and puts it in `rd`, and stores the lower 32 bits of the value held in `rs2` to the original address in `rs1`.

Listing 5. `SSAMOSWAP.W` for RV64

```
if privilege_mode != M && menvcfg.SSE == 0
    raise illegal-instruction exception
if S-mode not implemented
    raise illegal-instruction exception
else if privilege_mode == U && senvcfg.SSE == 0
    raise illegal-instruction exception
else if privilege_mode == VS && henvcfg.SSE == 0
    raise virtual instruction exception
else if privilege_mode == VU && senvcfg.SSE == 0
    raise virtual instruction exception
else
    temp[31:0] = mem[X(rs1)]
    X(rd) = SignExtend(temp[31:0])
    mem[X(rs1)] = X(rs2)[31:0]
endif
```

If the memory referenced by `rs1` is not idempotent, then `SSAMOSWAP.W/D` causes a store/AMO access-fault exception.

Just as for AMOs in the A extension, `SSAMOSWAP.W/D` requires that the address held in `rs1` be naturally aligned to the size of the operand (i.e., eight-byte aligned for *doublewords*, and four-byte aligned for *words*). The same exception options apply if the address is not naturally aligned.

Just as for AMOs in the A extension, `SSAMOSWAP.W/D` optionally provides release consistency semantics, using the `aq` and `rl` bits, to help implement multiprocessor synchronization. An `SSAMOSWAP.W/D` operation has acquire semantics if `aq=1` and release semantics if `rl=1`.

The `SSAMOSWAP.W/D` instructions require the PMA of the accessed memory range to provide AMOSwap level support.

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. A legal checkpoint is defined as one that holds a value of `X`*, where* `X` *is the address at which the checkpoint is positioned on the shadow stack.*

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:

```
# a0 hold pointer to top of new shadow stack to switch to
stack_switch:
   ssrdp ra
   beqz ra, 2f                   # skip if Zicfiss not active
   ssamoswap.d ra, x0,  (a0)     # ra=*[a0] and *[a0]=0
   beq         ra, a0,  1f       # [a0] must be == [ra]
   unimp                         # else crash
1: addi        ra, ra,  XLEN/8   # pop the checkpoint
   csrrw       ra, ssp, ra       # swap ssp: ra=ssp, ssp=ra
   addi        ra, ra,  -(XLEN/8) # checkpoint = "old ssp - XLEN/8"
   ssamoswap.d x0, ra,  (ra)     # Save checkpoint at "old ssp - XLEN/8"
2:
```

This sequence uses the `ra` *register. If the privilege mode at which this sequence is executed can be interrupted, then the trap handler should save the* `ra` *on the shadow stack itself. There it is guarded against tampering and can be restored prior to returning from the trap.*

When a new shadow stack is created by the supervisor, it needs to store a checkpoint at the highest address on that stack. This enables the shadow stack pointer to be switched using the process outlined in this note. The `SSAMOSWAP.W/D` *instruction can be used to store this checkpoint. When the old value at the memory location operated on by* `SSAMOSWAP.W/D` *is not required,* `rd` *can be set to* `x0`*.*

2.8. Shadow Stack Memory Protection

To protect shadow stack memory, the memory is associated with a new page type – the Shadow Stack (SS) page – in the single-stage and VS-stage page tables. The encoding `R=0`, `W=1`, and `X=0`, is defined to represent an SS page. When `menvcfg.SSE=0`, this encoding remains reserved. Similarly, when `V=1` and `henvcfg.SSE=0`, this encoding remains reserved at `VS` and `VU` levels.

If `satp.MODE` (or `vsatp.MODE` when `V=1`) is set to `Bare` and the effective privilege mode is below M, shadow stack memory accesses are prohibited, and shadow stack instructions will raise a store/AMO access-fault exception. When the effective privilege mode is M, any memory access by an `SSAMOSWAP.W/D` instruction will result in a store/AMO access-fault exception.

Memory mapped as an SS page cannot be written to by instructions other than `SSAMOSWAP.W/D`, `SSPUSH`, and `C.SSPUSH`. Attempts will raise a store/AMO access-fault exception. Implicit accesses, including instruction fetches to an SS page, are not permitted. Such accesses will raise an access-fault exception appropriate to the access type. However, the shadow stack is readable by all instructions that only load from memory.

Stores to shadow stack pages by instructions other than `SSAMOSWAP`*,* `SSPUSH`*, and* `C.SSPUSH` *will trigger a store/AMO access-fault exception, not a store/AMO page-fault exception, signaling a fatal error. A store/AMO page-fault suggests that the operating system could address and rectify the fault, which is not feasible in this scenario. Hence, the page fault handler must decode the opcode of the faulting instruction to discern whether the fault was caused by a non-shadow-stack instruction writing to an SS page (a fatal condition) or by a shadow stack instruction to a non-resident page (a recoverable condition). The performance-critical nature of operating system page fault handlers necessitates triggering an access-fault instead of a page fault, allowing for a straightforward distinction between fatal conditions and recoverable faults.*

Operating systems must ensure that no writable, non-shadow-stack alias virtual address mappings exist for the physical memory backing the shadow stack. Furthermore, in systems where an address-misaligned exception supersedes the access-fault exception, handlers emulating misaligned stores must be designed to cause an access-fault exception when the store is directed to a shadow stack page.

All instructions that perform load operations are allowed to read from the shadow stack. This feature facilitates debugging and performance profiling by allowing examination of the link register values backed up in the shadow stack.

As of the drafting of this specification, instruction fetches are the sole type of implicit access subjected to single- or VS-stage address translation.

The access type is classified as a store/AMO in the event of an access-fault, page-fault, or guest-page fault exception triggered by shadow stack instructions.

Shadow stack instructions are restricted to accessing shadow stack (`pte.xwr=010b`) pages. Should a shadow stack instruction access a page that is not designated as a shadow stack page and is not marked as read-only (`pte.xwr=001`), a store/AMO access-fault exception will be invoked. Conversely, if the page being accessed by a shadow stack instruction is a read-only page, a store/AMO page-fault exception will be triggered.

Shadow stack loads and stores will trigger a store/AMO page-fault if the accessed page is read-only, to support copy-on-write (COW) of a shadow stack page. If the page has been marked read-only for COW tracking, the page fault handler responds by creating a copy of the page and updates the `pte.xwr` *to* `010b`*, thereby designating each copy as a shadow stack page. Conversely, if the access targets a genuinely read-only page, the fault being reported as a store/AMO page-fault signals to the operating system that the fault is fatal and non-recoverable. Reporting the fault as a store/AMO page-fault, even for* `SSPOPCHK` *initiated memory access, aids in the determination of fatality; if these were reported as load page-faults, access to a truly read-only page might be mistakenly treated as a recoverable fault, leading to the faulting instruction being retried indefinitely. The PTE does not provide a read-only shadow stack encoding.*

Attempts by shadow stack instructions to access pages marked as read-write, read-write-execute, read-execute, or execute-only result in a store/AMO access-fault exception, similarly indicating a fatal condition.

Shadow stacks should be bounded at each end by guard pages to prevent accidental underflows or overflows from one shadow stack into another. Conventionally, a guard page for a stack is a page that is not accessible by the process that owns the stack.

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.

The G-stage address translation and protections remain unaffected by the Zicfiss extension. The `xwr == 010b` encoding in the G-stage PTE remains reserved. When G-stage page tables are active, the shadow stack instructions that access memory require the G-stage page table to have read-write permission for the accessed memory; else a store/AMO guest-page fault exception is raised.

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.

Svpbmt and Svnapot extensions are supported for shadow stack pages.

The PMA checks are extended to require memory referenced by shadow stack instructions to be idempotent. The PMP checks are extended to require read-write permission for memory accessed by shadow stack instructions. If the PMP does not provide read-write permissions or if the accessed memory is not idempotent then a store/AMO access-fault exception is raised.

Chapter 3. Landing Pad (Zicfilp)

To enforce forward-edge control-flow integrity, the Zicfilp extension introduces a landing pad (`LPAD`) instruction. The `LPAD` instruction must be placed at the program locations that are valid targets of indirect jumps or calls. The `LPAD` instruction (See Section 3.4) is encoded using the `AUIPC` major opcode with `rd=x0`.

Compilers emit a landing pad instruction as the first instruction of an address-taken function, 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 calls 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-check 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 sites 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 indirectly 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 being 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` instructions 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.

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

When Zicfilp extension is not active or not implemented, the hart does not require landing pad instructions at the targets of indirect calls/jumps, and the landing instructions revert to being no-ops. 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 depends on the Zicsr extension.

3.1. Landing Pad Enforcement

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 hart 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 it's 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 it is 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 use the semantically-direct calls only if 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-check exception (cause=18) with `xtval` 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.*

3.2. Zicfilp CSRs

This section specifies the CSR state of the Zicfilp extension.

3.2.1. Machine Environment Configuration Register (`menvcfg`)

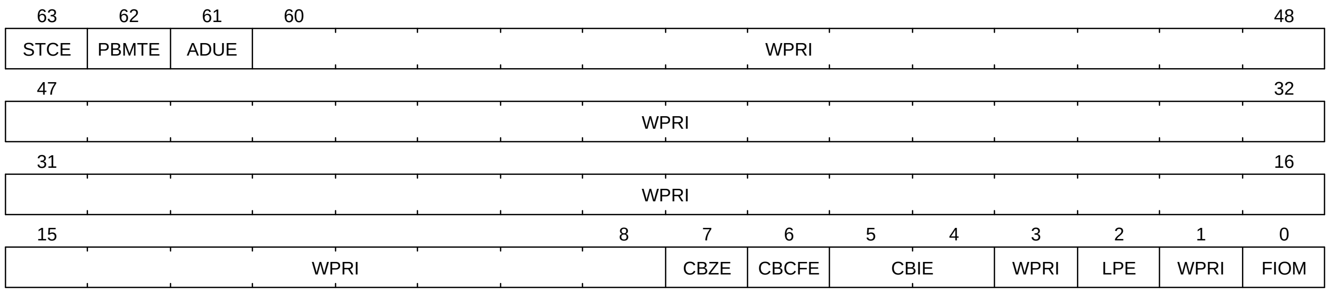


Figure 4. Machine environment configuration register (`menvcfg`)

Zicfilp extension introduces the `LPE` field (bit 2) in `menvcfg`. When the `LPE` field is set to 1 and S-mode is implemented, the Zicfilp extension is enabled in S-mode. If `LPE` field is set to 1 and S-mode is not implemented, the Zicfilp extension is enabled in U-mode.

When the `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 `ELP` state; it remains as `NO_LP_EXPECTED`.
- The `LPAD` instruction operates as a no-op.

If the `LPE` field is 0 and S-mode is not implemented, these rules apply to U-mode.

3.2.2. Supervisor Environment Configuration Register (`senvcfg`)

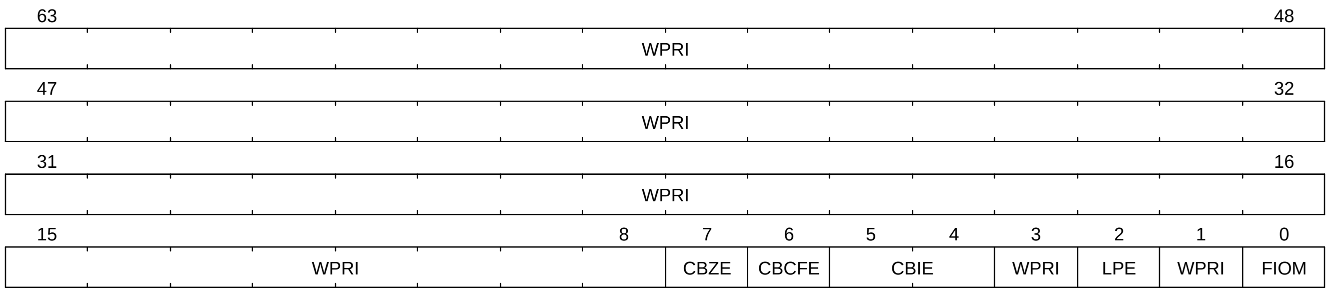


Figure 5. Supervisor environment configuration register (`senvcfg`)

Zicfilp extension introduces the `LPE` field (bit 2) in `senvcfg`. When the `LPE` field is set to 1, the Zicfilp extension is enabled in VU/U-mode. When the `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 `ELP` state; it remains as `NO_LP_EXPECTED`.
- The `LPAD` instruction operates as a no-op.

3.2.3. Hypervisor Environment Configuration Register (`henvcfg`)

63	62	61	60												48
STCE	PBMTE	ADUE	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. Hypervisor environment configuration register (`henvcfg`)

Zicfilp extension introduces the `LPE` field (bit 2) in `henvcfg`. When the `LPE` field is set to 1, the Zicfilp extension is enabled in VS-mode. When the `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 `ELP` state; it remains as `NO_LP_EXPECTED`.
- The `LPAD` instruction operates as a no-op.

3.2.4. Machine Status Register (`mstatus`)

63	62														48
SD	WPRI														
47					42	41	40	39	38	37	36	35	34	33	32
WPRI						MPELP	WPRI	MPV	GVA	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 7. 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 3.5. 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.

3.2.5. Supervisor Status Register (`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 8. Supervisor-mode status register (`sstatus`) when `SXLEN=64`

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

3.2.6. Virtual Supervisor Status Register (`vsstatus`)

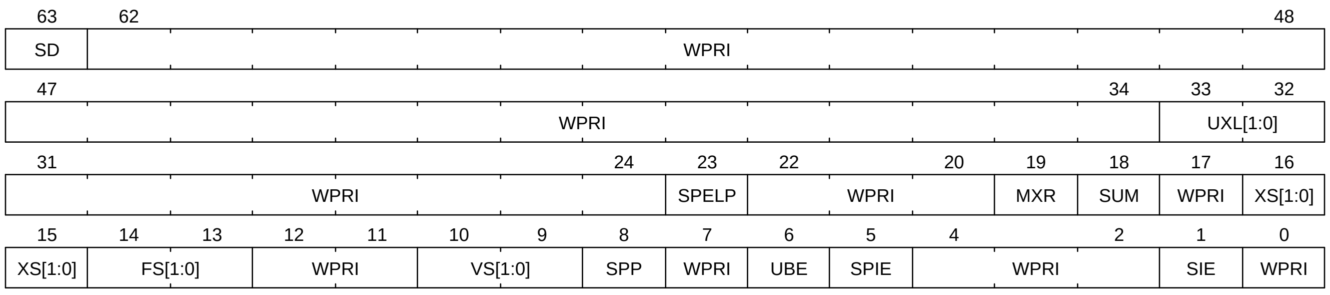


Figure 9. Virtual supervisor status register (`vsstatus`) when `VSXLEN=64`

The Zicfilp extension introduces the `SPELP` (bit 23) field that holds the previous `ELP`, and is updated as specified in Section 3.5. 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.

3.2.7. Machine Security Configuration Register (`mseccfg`)

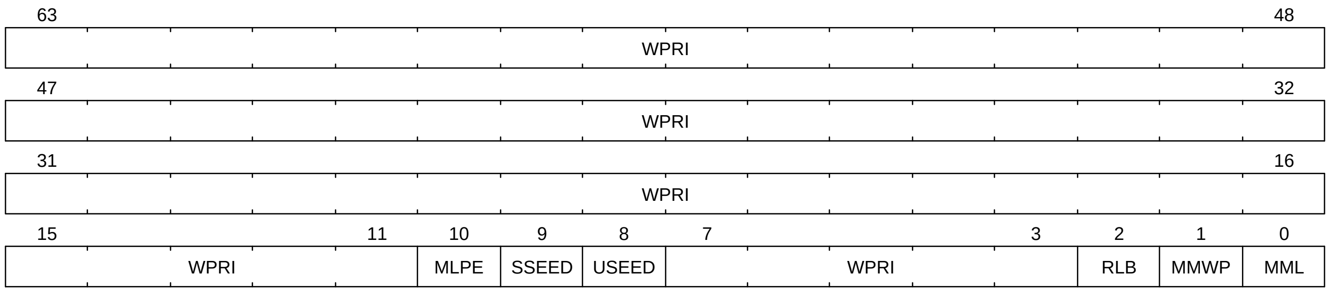


Figure 10. Machine security configuration register (`mseccfg`) when `MXLEN=64`

The Zicfilp extension introduces the `MLPE` (bit 10) field in `mseccfg`. When `MLPE` field is 1, Zicfilp extension is enabled in M-mode. When the `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 `ELP` state; it remains as `NO_LP_EXPECTED`.
- The `LPAD` instruction operates as a no-op.

3.2.8. Debug Control and Status Register (`dcsr`)

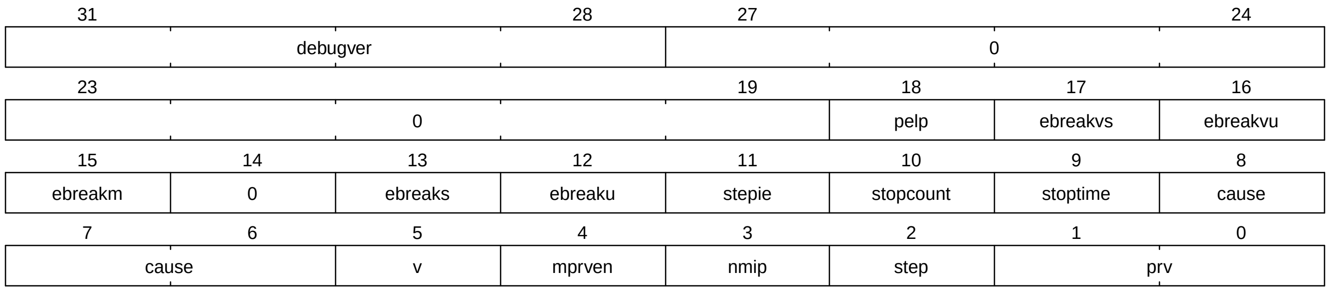


Figure 11. Debug Control and Status (`dcsr`)

The Zicfilp extension introduces the `pelp` (bit 18) in `dcsr`. The `pelp` field holds the previous `ELP`, and is updated as specified in Section 3.5. 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.

3.2.9. Resumable NMI Status Register (`mnstatus`)

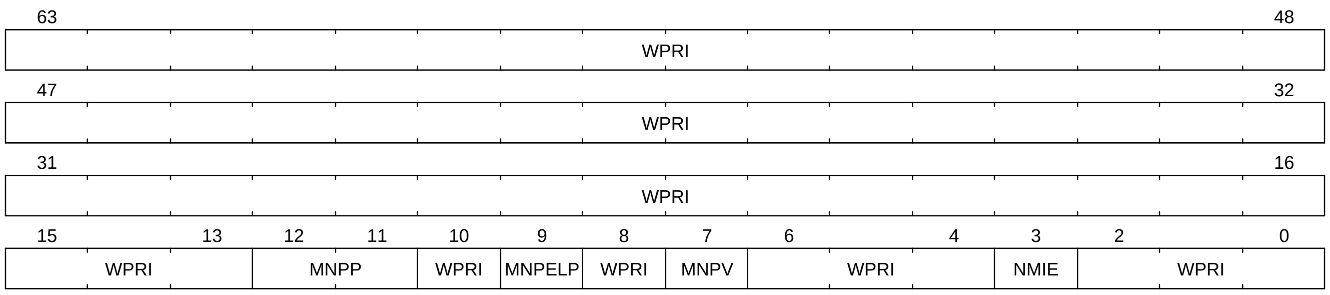


Figure 12. Resumable NMI Status (`mnstatus`) when `MXLEN=64`

The Zicfilp extension introduces the `MNPELP` (bit 9) in `mnstatus`. The `MNPELP` field holds the previous `ELP`, and is updated as specified in Section 3.5. The `MNPELP` field is encoded as follows:

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

3.3. 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.

When S-mode is implemented, it is determined as follows:

Table 2. `xLPE` *determination when S-mode is implemented*

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

When S-mode is not implemented, it is determined as follows:

Table 3. `xLPE` *determination when S-mode is not implemented*

Privilege Mode	xLPE
M	`mseccfg.MLPE`
U	`menvcfg.LPE`

The Zicfilp must be explicitly enabled for use at each privilege mode.

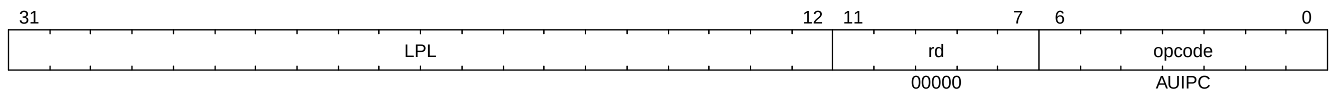
Programs compiled with the `LPAD` *instruction continue to function correctly, but without forward-edge CFI protection, when the Zicfilp extension is not implemented or is not enabled.*

3.4. 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-check exception with `xtval` set to "landing pad fault (code=2)" if any of the following conditions are true:

- The `pc` is not 4-byte aligned and `ELP` is `LP_EXPECTED`.
- 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.

The behavior of the trap if a software-check exception is raised by this instruction is specified in section Section 3.5. If a software-check exception is not caused then the `ELP` is updated to `NO_LP_EXPECTED`.



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

Listing 7. `LPAD` operation

```
if (xLPE == 1 && ELP == LP_EXPECTED)
    // If PC not 4-byte aligned then software-check exception
    if pc[1:0] != 0
        raise software-check exception
    // If landing pad label not matched -> software-check exception
    else if (inst.LPL != x7[31:12] && inst.LPL != 0)
        raise software-check exception
    else
        ELP = NO_LP_EXPECTED
else
    no-op
endif
```

3.5. 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 a software-check exception with `xtval` set to "landing pad fault (code=2)" (See Table 3.7 of Privileged Specification [3]).

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

The software-check exception due to the instruction not being an `LPAD` instruction when `ELP` is `LP_EXPECTED` or an software-check exception caused by the `LPAD` instruction itself (See Section 3.4) 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, an `MPELP` bit is provided in the `mstatus` CSR. To store the previous `ELP` state on trap delivery to S/HS-mode, an `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. When executing an `xRET` instruction, if `xPP` holds the value `y`, then `ELP` is set to the value of `xPELP` if `yLPE` is 1; otherwise, it is set to `NO_LP_EXPECTED`; `xPELP` is set 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, if `dcsr.prv` holds the value `y`, then `ELP` is set to the value of `pelp` if `yLPE` is 1; otherwise, it is set to `NO_LP_EXPECTED`.

When the Smrnmi [4] extension is implemented, a `MNPELP` field (bit 9) is provided in the `mnstatus` CSR to hold the previous `ELP` state on a trap to the RNMI handler. When a RNMI trap is delivered, the `MNPELP` is set to `ELP` and `ELP` set to `NO_LP_EXPECTED`. Upon a `MNRET`, if the `mnstatus.MNPP` holds the value `y`, then `ELP` is set to the value of `MNPELP` if `yLPE` is 1; otherwise, it is set to `NO_LP_EXPECTED`.

The trap handler in privilege mode `x` must save the `xPELP` bit and the `x7` register before performing an indirect call/jump if `xLPE=1`. If the privilege mode `x` can respond to interrupts and `xLPE=1`, 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] “Zimop May-Be-Operations Extension.” [Online]. Available: github.com/riscv/riscv-isa-manual/blob/main/src/zimop.adoc.

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

[4] “Smrnmi Standard Extension for Resumable Non-Maskable Interrupts.” [Online]. Available: github.com/riscv/riscv-isa-manual/blob/main/src/rnmi.adoc.