Preliminary RISC-V HFI Spec (v1)

Authors: Shravan Narayan (UT Austin), Tal Garfinkel (UC San Diego),Ravi Sahita (Rivos), Deepak Gupta (Rivos), Ziang Tian (UT Austin)

Version v1, 7/2025: Draft

Table of Contents

List of figures

List of tables

Contributors

This HFI RISC-V specification has been contributed to directly or indirectly by:

- Shravan Narayan <required1@email.com>
- Tal Garfinkel <required2@email.com>
- Ravi Sahita <required3@email.com>
- Deepak Gupta <required4@email.com>
- Ziang Tian <ztian@utexas.edu>

Chapter 1. Introduction

Hardware-assisted Fault Isolation (HFI) is a hardware extension designed to support in-process isolation (sandboxing) of untrusted code. It targets both unmodified native binaries and existing Software-Based Fault Isolation (SFI (Wahbe et al., 1993)) systems, such as WebAssembly (Haas et al., 2017) and Google's Ubercage (V8 JIT sandbox (saelo, 2021)), to improve their performance, security, and scalability (Narayan et al., 2023).

HFI aims to provide the capabilities required for secure sandboxing: robust data and control flow isolation, as well as complete and efficient mediation of privileged instructions (e.g., system calls). This specification describes the functional properties of HFI on RISC-V; a more complete description of the background and motivation is presented in the ASPLOS 2023 paper (Narayan et al., 2023).

1.1. HFI Execution Model

To support these capabilities, HFI defines a shared execution model involving a privileged runtime and a restricted sandbox mode. The code responsible for managing the sandbox is referred to as the *sandboxing runtime*. The runtime is responsible for saving and restoring context appropriately and can use HFI to multiplex many sandboxes across harts, scheduling them as it sees fit. The runtime controls the transition into the sandbox using the `hfienter` instruction and regains control when the sandboxed code executes an `hfiexit` instruction. When a hart operates in HFI Mode, execution is restricted to ensure isolation. To prevent the sandbox from bypassing these restrictions, HFI interposes on all paths out of the sandbox, ensuring the runtime can completely mediate all control flow and access to sensitive OS resources.

To address diverse sandboxing requirements—ranging from legacy native binaries to modern SFI runtimes—HFI is divided into two distinct extensions: the Base HFI Extension and the HFI Explicit Extension.

1.2. Base HFI Extension

The Base HFI extension is the foundational layer of HFI, designed primarily to enable the secure sandboxing of unmodified native binaries and libraries. For native code to run inside a sandbox without modification or recompilation, it must be able to use standard RISC-V load, store, and jump instructions. The Base extension achieves this natural isolation by overlaying the standard address space with Implicit Data and Implicit Code regions. These regions transparently apply bounds checks to every standard memory access and instruction fetch performed by the sandboxed code, allowing native binaries to operate normally while determining access permissions on a first-match basis.

To ensure the native binary cannot bypass the sandbox or access sensitive OS resources, the Base extension hardens its sandbox by trapping and redirecting privileged operations, specifically system calls. Moreover, to prevent guest application from explicitly calling sandbox exits, sandbox exit instructions are also interposed on. Both system calls and sandbox exits are redirected to a trusted exit handler that can be configured by the runtime.

This extension provides the core Interposition and Implicit Region logic necessary for coarse-grained isolation. The Base HFI extension is specified in Chapter 2.

1.3. Explicit HFI Extension

The HFI Explicit extension builds upon the Base HFI extension to accelerate existing Software-Based

Fault Isolation (SFI) technologies. This extension requires the Base HFI extension to be implemented and targets runtimes that utilize a linear memory model, such as WebAssembly. While the Base extension targets standard memory models, SFI runtimes often require memory access to be relative to a specific heap base with precise bounds.

The HFI Explicit extension offloads the software overhead of bound-checking these accesses to hardware through two mechanisms. First, it provides Explicit Data Regions which, unlike the coarse-grained power-of-two sizing of implicit regions, support byte-granular sizing critical for exact heap bounds. Second, it introduces dedicated h-prefixed instructions (e.g., `hlw`, `hsw`) that perform region-relative addressing. By using these instructions, an SFI compiler can map software linear memory directly to hardware regions, allowing the hardware to perform bound checks automatically and eliminating the need for costly software guard instructions. The HFI Explicit extension is specified in Chapter 3.

Chapter 2. Base HFI Extension

2.1. Sandbox Setup

The sandboxing runtime can query if the processor supports HFI by checking the supported extension string from the device tree or the ACPI RISC-V Hart Capabilities Table as is the common practice (Jones, 2023). Specifically, the string will list "hfi-*version*" if HFI is supported. The HFI version described in this spec is "1".

If HFI is supported, HFI mode is enabled with the `hfienter` instruction, and disabled with the `hfiexit` instruction. The `hfienter` instruction has two variants---the first variant takes a single register operand which specifies the sandbox configuration options as a bit vector called `hfioptions_t`, that has the following fields:

- *lock_regions*: *reg_u1*---Regions configurations are normally specified prior to `hfienter`, using instructions such `hfisetregionsize` (See Chapter 2.3). If *lock_regions* is set to *false (0)*, region manipulation instructions can also be called while in HFI mode (i.e., the region configuration can be modified in the sandbox). If *true (1)*, these instructions cannot be called while in HFI mode.
- *redirect_system_calls*: *reg_u1*---HFI sandboxes can interpose on system calls made by sandboxed code. If *redirect_system_calls* is set to *false (0)*, system calls are unaffected by HFI. If *true (1)*, HFI will redirect system calls (`ecall`) instructions to the HFI exit handler (which can be set with the `hfisetexithandler` instruction discussed in Chapter 2.2).
- *redirect_exits*: *reg_u1*---HFI sandboxes can interpose on invocations of `hfiexit` so the application can take control once the sandbox code completes. If *redirect_exits* is set to *false (0)*, `hfiexit` disables HFI and execution simply falls through to the next instruction. If *true (1)*, `hfiexit` disables HFI and redirects control flow to the exit_handler (which can be set with the `hfisetexithandler` instruction discussed in Chapter 2.2).
- *serialize_enter_exits*: *reg_u1*---HFI sandboxes can add fences sandbox entries and exit (through the `hfienter` and `hfiexit` instructions), a required step for Spectre protections. If *serialize_enter_exits* is *false (0)*, these instructions do not introduce any serialization or fencing. If *true (1)*, these instructions act as a fence for all memory operations that are run prior to these instructions.

The second variant of `hfienter` operates like `hfienter` variant 1 plus a jump instruction; it takes the `hfioptions_t` as the first operand and takes the target of the jump in a register as the second operand.

Behavior The `hfienter` instruction interacts with multiple internal registers. It sets the field `hfiusermodeenabled` in an internal register `hfistatusreg` to 1 on execution, and clears the `hfifaultoccurred` field in another internal register `hfifaultstatusreg` indicating that there has not been a violation of HFI's region rules since the last invocation of `hfienter`. The second variant of `hfienter` additionally sets the PC to the target specified in the second operand after its execution. The `hfiexit` instruction interacts with all three fields from `hfistatusreg`. It sets its `hfiusermodeenabled` field to 0 on execution, write the value of the program counter of the `hfiexit` instruction to its `hfiexitpc` field, and set its `hfiexitreason` field to 1, indicating the sandbox exit was due to and `hfiexit` instruction. `hfienter` and `hfiexit` must always act as fence for other HFI instructions, i.e., they only execute after all in-flight HFI instructions have completed.

Faults The `hfienter` instruction will trap if the CPU is already in HFI mode. The `hfiexit` instruction will trap if the CPU is not in HFI mode.

Design Rationale Most of the options in `hfioptions_t` are present to support different use cases. When executing unmodified native code, the sandboxed code is totally untrusted, thus regions are

locked, system calls are redirected, etc. However, when sandboxing code from existing SFI systems such as WebAssembly, it is more efficient/not necessary for HFI to do some checks, since these systems already handle some safety checks in their compiler/runtime. For example, these systems may not want to redirect system calls. Similarly, the choice of whether or not to serialize `hfiexit` and `hfienter` will depend on the threat model that a particular application is trying to enforce (i.e. if they want greater Spectre safety), we leave it to the user to choose whether they wish to opt into this added overhead. Finally, `hfienter` is offered in two variants as some sandboxing runtimes may want the option to jump directly to executing sandboxed code after sandboxing is enabled via `hfienter`, while other runtimes may want the instruction to fall through to avoid any costs due to extra control flow. The former behavior can be difficult to accomplish with the single operand variant of `hfienter` as it requires the instruction following `hfienter` to be part of the sandbox's code---which may not always be possible. The latter behavior's performance is difficult to achieve with an instruction that includes control flow.

2.2. Configuring the Exit handler

HFI supports interposition via. redirection on all paths out of the sandbox including sandbox exits (via. `hfiexit`) and system calls (and by extension signals). As noted, which instructions (system calls and/or `hfiexit`) are redirected is configured through `hfioptions_t`, that is passed as an operand to `hfienter`.

To handle this redirection, an exit handler is setup with the `hfisetexithandler` instruction. This instruction should be invoked prior to `hfiexit`. `hfisetexithandler` takes one operand, a 64-bit register, that holds the address of the exit handler. The current exit handler can be retrieved via the `hfigetexithandler` instruction.

Behavior The `hfisetexithandler` instruction sets the `hfiexithandlerreg` internal register to the address specified in its operand.

- If `hfiexit` is configured to invoke the exit handler, the `hfiexit` instruction, when executed by sandbox code, will jump to the exit handler *after* execution (during which `hfiexitreason` in `hfistatusreg` is set to 1 indicating the exit was due to the `hfiexit` instruction).
- If system calls are configured to invoke the exit handler, the system call instruction, when executed by sandbox code, will jump to the exit handler *before* execution, set the `hfiexitreason` field to 2 indicating the exit was due to a system call, and disable sandboxing by setting the `hfiusermodeenabled` field to 0.

Faults The `hfisetexithandler` instruction will trap if the CPU is already in HFI mode. If the exit handler is set to a location without code permissions, the behavior of the CPU would be identical to a normal jump to an address without code permissions.

Design Rationale Sandboxed code will invoke `hfiexit` to exit the sandbox. We must ensure that the sandboxed code always returns control to the sandboxing runtime after exits, which means, invocation of `hfiexit` should return control to trusted code. Thus, we have added support for redirecting all invocations of `hfiexit`.

Sandboxed code can also invoke system calls; since HFI's restrictions don't apply to kernel code, system calls could be used to bypass isolation enforced by the hardware (Connor et al., 2020). Thus, trusted code needs the ability to interpose on system calls, so that it can restrict the invocation of unsafe system calls by sandboxed code. Such interposition on system calls could be done using assembly rewriting or using kernel features such as EBPF, however, this is slow and cumbersome. To allow efficient interposition of system calls, HFI provides hardware support to redirect system calls.

When enabled, system calls simply act like a jump instruction to the exit handler.

2.3. Regions

Regions offer a limited version of the functionality found in traditional segmented memory systems, which control access to contiguous ranges of memory using `<base, bound, permission>` tuples.

By default, a processor in HFI mode has no access to memory, i.e. it cannot read data or run code. To enable sandboxed code to run, a sandbox runtime must explicitly configure regions prior executing `hfienter`. In the next sections, we provide a brief overview of HFI regions, and then specify how to configure these them.

2.3.1. Regions Types

HFI base extension provides regions that *implicitly* apply bound checks to every memory access when the sandbox is enabled. These regions are thus also called "implicit regions". This is essential for isolating memory accesses and control flow of unmodified native code. HFI discriminates such regions into code and data ones, to keep the control and data pipelines simpler and more efficient. Data regions can grant read and write access and only apply to loads and stores, while code regions apply only to instruction fetches, and can only grant execute permissions.

With every memory access under bound checks, data regions grant access on a first-match basis. For example, if sandboxed code executes an “lw rd, rs(offset)” instruction, HFI will check if the address in `rs + offset` is in range for *any* of the existing regions in parallel. For the first matching region, it will check the permissions to see if reads are allowed — if so, it will proceed. If the permission check fails, or if there is no match, HFI will trap. Code regions apply similar checks to code.

Specifically, regions perform bound checks based on *prefix matching*. Each region specifies a *base_prefix* (the `base` component in the region’s tuple) and an *lsb_mask* (the `bound` component in the region’s tuple). To check if an address is in bounds, HFI uses the *lsb_mask* to remove the least significant bits of the address, and compares the remaining prefix to *base_prefix*. Thus, regions must be power of two sized and aligned — thus, they trade granularity for efficient checking — in particular, checks can be implemented with simple masking operations.

Notes on size and alignment Unaligned memory operations that are split by the micro-architecture into independent operations will be checked independently by HFI. If one split faults, the original fused instruction will fault, as will the split instruction that violates HFI bounds. However, the split operation that is within bounds may be allowed to have visible *micro-architectural* side effects within the sandbox.

Regions also have minimum sizes. The smallest implicit region is 64 bytes. The behavior of regions that do not meet these minimums is undefined.

2.3.2. Manipulating Regions

Region state, which is stored in internal registers, can only be read or modified by HFI instructions. The instructions broadly operate on regions by a region number (`region_number_t`) — a unique number/index assigned to each HFI region on the CPU.

Region number assignments The version of HFI in this spec (hfi1) up to this chapter defines two regions: one implicit data region of `region_number_t` of 1, and one implicit data region of

`region_number_t` of 2. Future versions may define multiple regions of each type (Chapter 3.5), and each region will be assigned a unique `region_number_t`.

We now introduce the instructions that can configure these regions:

Setting Region base and size The following instructions are used to specify the range of memory a region applies to. To configure a specific region's base and size, a sandboxing runtime must first select the target region. This is accomplished by direct instructions that write to a dedicated register indicating current region being configured. Once a region is selected, its base address and mask (bound) can be configured using separate instructions that target dedicated registers. This approach breaks down the operation to fit standard RISC-V instruction formats, avoiding having multiple destination registers.

`hfiselectregion`(*region_number_t*)

`hfisetregionbase`(*reg_u64 base*)

`hfigetregionbase`() → *reg_u64 base*

`hfisetregionbound`(*reg_u64 mask_or_bound*)

`hfigetregionbound`() → *reg_u64 mask_or_bound*

Behaviour The region to configure is set with `hfiselectregion`, and then its sizes and locations are set using `hfisetregionbase` and `hfisetregionbound`. Regions are typically setup prior to entering the sandbox (with `hfienter`). If this instruction runs in a sandbox, i.e., `hfiusermodeenabled` is 1 — which is allowed when `lock_regions` is 0 — it must act as a memory fence. All prior memory instructions must complete before executing this instruction and subsequent memory operations should be issued only after the update.

`hfiselectregion` has `region_number_t` as its operand register, and works closely with the other four instructions. While `hfisetregionbase` has the base of the region as its operand, `hfisetregionbound's operand depends on the type of region. The third operand should contain the mask for the region. The value of the base and bound/mask should additionally conform to the per-region size and alignment requirements in Chapter 2.3.1. However, the instruction does not check whether operands meet this criteria. If the operands don't meet the criteria, the resulting behavior is undefined.

`hfigetregionbound`, `hfigetregionbase`, and `hfigetregionbound` return region size information based on the previously fed `region_number_t`.

Faults These instructions fault if the region number specified does not exist (i.e., it is greater than the total number of regions). For efficiency, these instructions should not check whether region locations or sizes are invalid (e.g., the program has specifies an implicit region base that is not aligned to its size); rather the hardware will continue to operate using the provided base and size, although this behavior is to be considered undefined.

Design Rationale When `hfisetregionsize` is run prior to `hfienter`, it doesn't need to act as a fence as `hfienter` can fulfill this purpose. However, when used inside the sandbox, `hfisetregionsize` must act as a fence, as otherwise inflight memory operations could potentially access memory outside the sandbox when they were issued if a region resized.

Setting Permissions, Enabling/Disabling Regions The following instructions are used to configure permissions on regions, as well as to enable and disable regions.

`hfisetregionpermission`(*permission_set_t*, *permission_t*)

`hfigetregionpermission`(*permission_set_t*) → *permission_t*

Behaviour Region permissions are set using the `hfisetregionpermission` instruction. This instruction sets the permissions of all regions using a single bit vector. The instruction takes `permission_set_t` as the first argument. This is a 32-bit register which must have the value 0; other values are reserved for future use. The second operand to this instruction is a permission bit-vector `permission_t` encoded as follows:

- Bits 0 to 2 are permissions for implicit data region 1. Bit 0 indicates if the region is enabled (i.e., should be enforced by the sandbox). Bit 1 and 2 indicates whether the region has read and write permissions respectively.
- Bits 3 to 4 are permissions for implicit data region 1. Bit 3 indicates if the region is enabled. Bit 4 indicates whether the region has execute permissions.

Faults This instruction will fault if the `permission_set_t` is not set to 0, or if `hfiusermodeenabled` and `lock_regions` is set to 1 (when hfi mode is on with region configurations locked). Any unused bits of `permission_t` are ignored; thus setting an unused bit will not fault.

Design Rationale An alternate design for this instruction would be to split up `hfisetregionpermission` to operate per-region (by changing the instruction to take `region_number_t` to operate on as the first parameter). However, this would lead to additional overheads in practice. This is because this instruction is primarily used when switching between active sandboxes. In this case, the permissions of all regions would likely need to be adjusting. If `hfisetregionpermission` operated per region, then three calls to this instruction would be needed to adjust the permissions of the three regions. In contrast, our design allows this to occur in a single instruction. Additionally, the `permission_set_t` parameter further future-proofs this design by allowing us to modify the format of this instruction for future versions of HFI, without breaking backward compatibility.

Clearing Regions To clear region state rapidly e.g. on context switches, HFI offers `hfiresetregions`.

Behaviour This instruction sets the base and bound/mask of all regions to zero (equivalent to calling `hfisetregionsize` with arguments of 0 on all regions), disables all regions and sets their permissions to zero (equivalent to calling `hfisetregionpermission` with a permission vector of 0).

Faults This instruction faults if `hfiusermodeenabled` and `lock_regions` is set to 1 (when hfi mode is on with region configurations locked).

Design Rationale While this instruction can effectively be achieved using combinations of other instructions, unifying this "reset" operation into a single instruction allows optimizing a number of paths. For example, when an application needs to switch between multiple sandboxes, software has to clear the state of the first sandbox before applying the state of the second sandbox. This allows optimizing the first step sequence. This operation is also useful when the OS kernel is switching between two scheduled processes both of which may use HFI. The OS kernel is responsible for saving and restoring each processes' HFI state, and thus can also use this instruction.

2.3.3. Implementation Considerations

Implementation Semantics and Spectre To ensure Spectre safety, the following guidance is offered for implementers.

For code regions: To ensure security, prefix-checking should be carried out in parallel with the decode stage. If the check finds a matching region with execute permissions, it succeeds, and decode carries on normally. If the check fails, it prevents the decoded micro-ops from entering the pipeline, and instead translates all instructions into a faulting NOP micro-op. This ensures that instructions that are out-of-bounds are not executed during committed execution, and are also not executed speculatively.

For data regions: Bounds checking, DTLB lookup, and cache index lookups should happen in parallel. One concern here is that, cache state could be modified as a result of secret (out-of-bounds) data. To prevent this sort of side-channel attack all bounds must checks occur *before* the processor resolves the physical address of a memory access. This is secure because the processor can update cache metadata like the LRU bits (for hits) or fetch new data blocks (for misses) only after resolving the physical address. HFI can therefore strictly prevent any metadata updates if there has been a fault.

Note that out-of-bounds address can affect metadata of the DTLB or i-cache---e.g., LRU bits. However, the invariant we guarantee---no secret (data stored outside the boundaries of the region) ever affects architectural state---is still not violated, since we do not allow the *result* of an out-of-bounds memory operation to propagate into any of these structure.

To summarize, HFI's data pipeline is Spectre safe, since the data cache is not updated prior to bounds checks being completed; HFI's control pipeline is safe as bounds checks finish prior to instruction decode which is before the execution of instructions. This approach also helps to guarantee that any code executed as the result of PHT, BTB, and RSB (speculative) predictions are checked prior to execution.

2.4. New Internal Registers

HFI stores state for the current sandbox in internal registers including: (1) the sandbox status (2) the exit handler (3) region configuration (4) the cause and status of HFI induced faults (traps).

2.4.1. Sandbox Status Register

Fields The sandbox status, stored in the `hfistatusreg` internal register, is read-only to userspace software, but writable by kernel code. The contents of the `hfistatusreg` register are as follows:

- *hfi_usermode_enabled* (1-bit): is updated automatically during `hfienter` (set to true), `hfiexit` (set to false), and when the exit handler is called (set to false).
- *hfi_exit_reason* (2-bit): indicates the reason for the last sandbox exit. Exit due to `hfiexit` would leave this set to 1. Exit due to a system call with leave this set to 2.
- *hfi_exit_pc* (60-bit): indicates the PC of the instruction causing the last exit. The first 2-bits and the last 2-bits of the PC are dropped and are assumed to be zero. Thus 60-bits of the PC are stored.

Serialization Caveat The `hfistatusreg` register should support updates via register renaming to support the performance expectations of hardware sandboxing. These values are expected to change frequently (once in few thousand instructions), and serializing this register updates with techniques like score-boarding will hinder practical adoption.

2.4.2. Exit Handler Registers

Fields The address of the sandbox exit handler is stored in the `hfiexithandlerreg` internal register. When a sandboxed hart traps or executes an `hfiexit` instruction, the Program Counter (PC) is set to

the value stored in this register. This register is written to using the `hfisetexithandler` instruction and read using the `hfigetexithandler` instruction. The contents of the `hfiexithandlerreg` register are as follows:

- *Exit Handler Address* (60-bit): indicates the address of the exit handler set for current sandbox. The first 2 bits and last 2 bits of this address are dropped and assumed to be zero.

Serialization Caveat The exit handler of the sandbox is stored in the `hfiexithandlerreg` internal register and can be set or read via the `hfisetexithandler` and `hfigetexithandler` instructions. Since the register is not expected to be frequently updated, it's updates may be scoreboarded if needed.

2.4.3. Region Registers

Fields There are three sets of internal registers dedicated to region configurations required by hfi: bounds registers, permissions registers, and selector registers. Different version of HFI may require different numbers of these registers, as discussed in Chapter 3.5. Bounds registers specify the bounds and sizes of a region. Each implicit region (data or code) has a 64-bit prefix register and a 64-bit mask register. Permissions registers specify whether a region can be read, written, or executed. Each data region has a 1-bit write field and a 1-bit read field, while each code region has a 1-bit execute field. On top of these registers is the selector register, which contains the region number of the region currently being configured. Its only field is a 32-bit `region_number_t`.

Serialization Caveat The region registers cannot be directly named, and must instead be modified or accessed through HFI instructions (e.g. `hfigetregionsize`). Since these registers should updated whenever a sandbox switch occurs, updates to these registers should ideally be supported through register renaming; however, implementations may choose to use the slightly less expensive scheme. In particular, implementations can configure the regions outside the sandbox without serializing, and defer fencing to when an `hfienter` instruction executes. This way, updates to these registers do not fence, and the `hfienter` instruction waits for all pending region register updates to complete. When a sandbox is enabled (`hfiusermodeenabled` set to 1), region register updates are not permitted if `lock_regions` option is set to 0. In the case that the region is not locked, however, all updates to the region register within the sandbox must serialize.

2.4.4. Fault Registers

Fields The result of an HFI fault due to an HFI policy violation is stored in the `hfifaultstatusreg` register. This register stores the precise cause of an HFI induced fault (trap). Since the register is not expected to be frequently updated, it's updates may be scoreboarded if needed. Information in `hfifaultstatusreg` is readable in user space so the runtime can respond appropriately, and read-write in the kernel so that this state can be saved/restored as part of the process context, as multiple processes may be using HFI, and fault delivery is asynchronous. This register has four fields:

- *hfi_fault_occurred* (1-bit): is a bit that indicates that an hfi fault has occurred.
- *hfi_fault_region* (8-bit): indicates which region the fault occured in. If the fault was caused by an implicit region i.e. insufficient permission to access a matched region, it will contain the number of the region. If the fault was cause because no implicit region matched an operation, it will contain 0.
- *hfi_fault_op* (2-bit): indicates the operation that faulted, this will be a `LOAD_FAULT` or `STORE_FAULT` for a failed load or store, `FETCH_FAULT` for a failed instruction fetch, or `AMO_FAULT` for a fault that occurred during an atomic memory operation.

- *hfi_fault_type* (1-bit): indicates the type of fault. If `OUT_OF_BOUNDS`, it indicates that no implicit region matched the operation (in this case `hfi_fault_region` will be equal to zero). If `INSUFFICIENT_PERMISSIONS`, whichever implicit region matched the operation did not have sufficient permissions.

Design Rationale The `hfienter` and `hfiexit` instructions modify the various bits of the `hfistatusreg` register. The updates to this register must be fast to allow rapid entries and exits to HFI mode; thus this must be supported by register renaming. Region registers are updated frequently as well when switching between different sandboxes. While this would ideally support register renaming for efficient register updates, an alternate scheme that ensures a single serialization (on entry into the sandbox) for a batch of region updates prior to entry, would provide adequate performance. Finally, updates to the region configuration within the sandbox should serialize, as an updates during speculative execution may allow Spectre-style attacks to break out of the sandbox.

2.5. Using HFI

Here we explore how HFI's features are used to create sandboxing runtimes in userspace, and the HFI support needed from the operating systems.

2.5.1. Sandboxing in Userspace

Suppose we have a sandbox runtime (e.g. part of an application implementing a Wasm FaaS server, or a library sandboxing framework (*The RLBox Sandboxing Toolkit*, 2024)), that is ready to create new sandbox. We assume that the runtime has reserved some memory for the sandbox (i.e., the sandbox memory), has placed the input for the sandboxed code in a memory buffer, and the sandboxed code itself is separated from other code and mapped in an isolated contiguous portion of the address space. Our runtime can now take the following steps:

Setting up regions To start, our runtime sets up access to the code, heap, and input memory so our application has everything it needs once the sandbox starts, it does this using the `hfisetregionsize`, `hfisetregionpermission` instructions to setup and enable regions that grant access to the allocated heap and inputs, and the application code. If no code regions are mapped, HFI will immediately trap after `hfi_enter` is called, as the processor will not be able to fetch instructions.

What type of regions the runtime will use, as as how the sandbox is configured will depend on if it is sandboxing a native or SFI (e.g. Wasm) based application.

Sandboxing Native code When sandboxing native code, the code being sandboxed is entirely untrusted, and thus, it cannot be allowed to modify any of HFI's state, or exit the sandbox in unexpected ways, or perform any other operation that would allow it to violate the sandbox's policy. Thus, the sandbox options flags in `hfioptions_t` that are passed to `hfienter` to start the sandbox will: set the `lock_regions` flag to 1 (true), since sandboxed code cannot be trusted to modify regions; set the `redirect_system_calls` flag to 1, as again this code can't be trusted to perform system calls directly; set the `redirect_exit` flag to 1, ensuring all control flow out of the sandbox is redirect to the trusted runtime. The runtime will use an implicit data region to permit the sandboxed code to use of the sandbox memory; the runtime will then ensure the sandboxed code's stack, heap and inputs are part of this implicit region. An implicit code regions will be used to mark the code of the sandbox.

Sandboxing using SFI runtimes When sandboxing code through SFI runtimes such as a Wasm runtime, sandboxing is handled as a mix of compiler/software-runtime checks as well as hardware checks from HFI. In the scenario, implementer has confidence in the correctness of the SFI's runtime

and compiler, and thus their use of HFI is different from sandboxing code, giving it greater flexibility and performance.

For example, on `hfienter` it can set the `lock_regions` flag to 0 (false), allowing the compiler/runtime to modify regions with `hfisetregionsize` and `hfisetregionpermission` without having to exit the sandbox. This can allow regions to be used more flexibly, e.g. it can load and spilling registers, and never need to exit the sandbox. The `redirect_system_calls` flag can also be set to 0 (false), as the Wasm compiler disallows direct access to system call instructions, ensuring that any system calls made will come from the trusted runtime, this can eliminate the overhead of unecessary sandbox exits. What the redirect_exit flag will be set to depends on the SFI implementation, it may set this flag to false and opt to let sandbox exit's fall through to minimize overhead, since it can ensure that it knows that it can control whatever instruction follows an `hfiexit`, or it may opt to set it to true, and setup an exit_handler. For granting access to memory, the runtime will use implicit regions for code and data.

Saving context A sandboxing runtime must protect its own execution context such as its stack and contents of CPU registers, before it switches to sandbox code. HFI leaves this mechanism entirely up to software---this flexibility is important for efficiency. For example, if our runtime is running untrusted native code---it will have to use springboards and trampolines (Yee et al., 2009)---lightweight assembly routines that (1) clear registers and switch to a separate stack prior to executing the sandboxed code and (2) restore these registers after the sandboxed is executed. However, if it is running Wasm code, it could opt to use zero-cost transitions (Kolosick et al., 2022) that rely on the compiler to ensure that the sandbox code cannot misuse the stack or scratch registers.

Setting up an exit handler If our runtime needs an exit handler either to handle `hfiexit` or redirected system calls, it will need to setup an exit handler with `hfisetexithandler`. This exit handler is implemented as a normal function call in the runtime that takes no arguments. It will query the `hfiexitreason` CSR to find out why it was invoked.

Entering the sandbox Having taken all these steps, our runtime is ready to start the sandbox. Once it calls `hfi_enter`, HFI mode is enabled, and the next instruction that runs will be inside a sandbox.

The exit handler (`hfiexit` and system calls) When the exit handler is called after the sandbox exits, it will transfer control to the exit handler function in the runtime, which will check a control and status register (CSR) to identify the cause of the exit, and respond appropriately.

For example, for sandbox exits, it will need to save context unless the sandbox execution has completed. Similarly for system calls, it will need to save context, but then also execute whatever additional logic is need to check the parameters of the system call for safety and finally invoking the system call (Connor et al., 2020).

2.5.2. OS (and VMM) integration

HFI is designed to require only minimal changes/support from the OS kernel. HFI requires support from OS kernels in two areas:

Handling HFI faults A fault may occur in HFI mode for to two reasons: (1) normal processor traps such as illegal instructions. (2) HFI policy violations. In both cases, when the processor traps into the kernel, HFI enforcement is disabled so as not to disrupt kernel execution, and the trap is handled through all the normal OS signal mechanisms (This is automatic as HFI hardware checks are only applied to userspace code). When an instruction can trap due to some HFI policy violation, the normal trap mechanism for the current hart is employed. HFI introduces a new trap code, `hfi_fault` to

support this.

When an HFI trap occurs, additional details about the cause are stored in the `hfifaultstatusreg`. A kernel trap handler can immediately read and store this state into the process struct for the current process, so it can be queried by a signal handler. The OS then invokes the standard signal handler registered by the application for memory access violations. The OS must invoke the signal handler with HFI disabled; if the signal handler returns control to the OS, the OS will re-enable HFI prior to resuming the faulting process.

Process scheduling The kernel must save and restore HFI state (stored in internal registers) when switching between processes/VMs etc. This can be performed using the HFI manipulation instructions `hfiresetregions`, `hfisetregionpermission`, `hfisetregionsize`, `hfigetregionsize`, etc. To know whether HFI mode is currently enabled by the user space process, more privileged code can check the state of the `hfiusermodeenabled` status register.

2.6. Instruction Encoding

HFI base extension introduces a set of functional instructions interacting with specific internal registers. Corresponding to the new internal registers, these instructions primarily fulfill three tasks: HFI mode transition, exit handler configuration, and region manipulation. All of the non-h-prefixed instructions are encoded in a R-type format under the same custom opcode `HFUNCTION` to achieve maximum flexibility (See Figure 5). The funct 3 field differentiates the instructions based on the tasks, and the funct 7 field further distinguishes the specific instruction. The specific encoding is shown in Table 5. The behavior of these instructions has mostly been covered in todo, todo and todo, and this section mainly details their encoding.

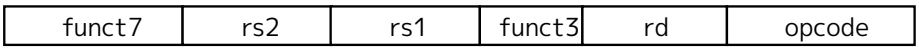

Figure 1. R-type instruction format.

HFI Mode Transition Instructions These include the `hfienter` and `hfiexit` instructions. `hfienter's 4-bit option vector comes from `rs1` and its `rd` field is invalid. The `hfiexit` instruction does not have any source or destination register, and thus these fields are all invalid in the instruction.

Exit Handler Instructions These instructions include `hfisetexithandler` and `hfigetexithandler`. They only take one operand from `rs1` holding the address of the exit handler and configure the handler for sandbox exits and syscalls in the dedicated internal register `hfiexithandlerreg`.

Region Manipulation Instructions These include the region-configure instruction `hfiselectregion`, the region size instructions `hfisetregionbase`, `hfigetregionbase`, `hfisetregionbound`, `hfigetregionbound`, and the region permissions `hfisetregionpermission`, `hfigetregionpermission`, `hfiresetregions`. All of these instructions directly interact with the corresponding internal registers, which are otherwise inaccessible. `hfiselectregion`, `hfisetregionbase`, `hfisetregionbound` and `hfisetregionpermission` all take a single operand from `rs1`, respectively the region number, the base address of the region and the mask for the region. The remaining "get" instructions set the `rd` register based on these internal registers.

Table 1. Encoding for R-type Non-h-prefixed Instructions

funct3	funct7	Instruction	funct3	funct7	Instruction
000	0000000	`hfienter`	010	0000000	`hfiselectregion`
000	0000001	`hfiexit`	010	0000001	`hfisetregionbase`

funct3	funct7	Instruction	funct3	funct7	Instruction
000	0000010	`hfientertarget`	010	0000010	`hfigetregionbase`
001	0000000	`hfisetexithandler`	010	0000011	`hfisetregionpermission`
001	0000001	`hfigetexithandler`	010	0000100	`hfigetregionpermission`
010	0000111	`hfiresetregions`			

2.7. Pending Design Considerations

Here, we include topics that we believe merit further discussion but which we have not fully resolved for inclusion in the specification.

2.7.1. Streamlining Control Transfers for Native Binaries

When sandboxing unmodified native binaries, we would ideally like control transfers into and out of the sandbox library to require minimal overhead and complexity. With a few small changes, we could make this simpler than what `hfienter` and `hfiexit` offer today.

At present, control transfers require redirecting control flow through small stubs (trampolines) that need to be mapped by the sandbox runtime into the sandboxed library address space, this adds complexity and overhead.

For example, consider a case where a host application has uses a library sandboxed with HFI; the applications want to invoke a function `foo()` in the library. To call `foo()`, it will need trampoline code---application code that performs a context switch by saving the current registers, switching the stack register to point to memory inside a region, enabling HFI and transferring control to `foo()`. This is mostly straightforward. However, once `foo` finishes executing (`foo` executes a return instruction), execution would attempt to return to the trampoline code---an operation that would fail as the trampoline code is not part of the sandbox code. Thus the host application, must perform an intermediate step---it must call `foo`, while modifying the return address on the stack to point to a stub within the sandbox, which invokes `hfiexit` and then returns to the trampoline. We could eliminate the need for this stub by dedicating a bit in the return address (e.g. it's least significant bit, as this should be unused as instructions are at least 16 bit aligned) on the stack that indicates that if HFI is enabled, this return should simply invoke `hfiexit`. Similar mechanism could also be applied to eliminating the need for trampolines for direct and indirect calls (i.e. callbacks) to host libraries.

2.7.2. HFI in M-mode or S-mode

We plan to add support for HFI in S-mode. Relatively small changes are necessary, however, we have not yet done a full analysis of how privileged instructions are handled. HFI support in m-mode may similarly be possible, but requires additional analysis for hardware implementation details.

Chapter 3. Explicit HFI Extension

The Explicit HFI Extension provides hardware support for fine-grained, region-relative addressing. With the Base HFI Extension, which targets unmodified native binaries using absolute virtual addresses, this extension is designed to accelerate SFI runtimes that utilize a linear memory model, such as WebAssembly.

In these SFI systems, sandboxed code accesses memory as an offset into a contiguous heap (linear memory) rather than via raw pointers. The Explicit HFI Extension maps this software model directly to hardware: the "heap" becomes an "explicit data region", and software bound-checks are replaced by dedicated `h-prefixed` instructions. This allows the hardware to enforce precise heap bounds automatically with every memory access, eliminating the performance overhead of software guard instructions.

Addressing in this mode is always relative to the base of the currently *active region* (by default, explicit data region 1), offering efficient control over access to specific buffers or explicit heaps. HFI provides two different region sizes~(*large*/*small*), with different granularities. Large regions can address up to 256 TiB (2^{48}) and are sized and aligned to multiples of 64K (2^{16}). Small regions, in contrast, can only address up to 4GiB (2^{32}), but are byte granular in size and alignment. Implicit regions checks are not applied to operations on explicit regions.

To access memory through explicit regions, a program must use the h-prefixed variants of normal load and store instructions (`hlw`, `hsw`, `hlh`, etc.). These instruction target the active region, but can be changed to use a different region as described in Chapter 3.5. For example, `hlw x1, A(x0)` will succeed if the address being loaded is falls within explicit region 1, and there is a read permission set on that region, otherwise it will fail.

3.1. Region Characteristics

HFI provides two different region sizes (*large*/*small*) for explicit regions to accommodate different granularity needs:

- **Large regions:** Can address up to 256 TiB (2^{48}) and are sized and aligned to multiples of 64K (2^{16}).
- **Small regions:** Can only address up to 4GiB (2^{32}) but are byte-granular in size and alignment.

Unlike implicit regions which are at lesast 64 bytes, there is no minimum size on small explicit regions, while the smallest large explicit region is 64K. Each explicit region has a 64-bit base register, a 64-bit bound register, and a 1-bit field specifying whether this is a large region or small region.

Design Rationale

The large and small region sizes and alignment constraints on explicit regions allow us to implement explicit regions with a single 32-bit comparator. For small regions, HFI checks the least-significant 32-bits is within bounds, and ensures the top bits are zero. For large regions, HFI will drop the first 16-bits, and compare bits 16-48, while checking the top bits are zero. Like Intel x86, RISCV typically support a 48-bit virtual address space, on which this spec is based. This design rationale also applies to Sv39 or Sv57, though the comparator lengths need to be adjusted accordingly.

While allowing regions that support arbitrary address ranges at with any size and alignment is conceptually simpler than specialized large and small regions, our restrictions allow bounds checking with very simple hardware. For base integer loads/stores, HFI's large and small regions constraints can be checked with a single 32-bit comparator, rather than the more costly multiple 64-bit comparators

needed to check arbitrary region bounds.

Explicit regions' added granularity is critical for supporting Wasm heaps, which grow in 64K increments (Haas et al., 2017)---while byte granularity is critical for efficiently sharing individual memory objects and sandboxing legacy code, as existing buffers can be shared in-place changing code or allocators.

To safely implement explicit bounds checks, the bounds check must be applied to the operand of an operation (i.e., the address being loaded/stored to) plus the size of the operand, to ensure that longer operations do not exceed bounds checks.

3.2. Configuring and Accessing Explicit Regions

Configuration for explicit regions take place through the same interface as implicit regions, detailed in Chapter 2.3.2. The only difference is that the runtime passes the bound of an explicit region instead of the mask of an implicit region. The runtime would still need to specify a region number to configure before setting the base and bound.

In addition to the two implicit regions defined in Chapter 2, this chapter defines a third explicit data region with a `region_number_t` of 3. Notably, `hfigetregionbound` returns the mask if the region is implicit (`region_number_t` is 1 or 2) and the bound if the region is explicit (`region_number_t` is 3). With a new explicit data region, bit 5 to 8 in the permission vector now correspond to explicit region 3: bit 5, 6 and 7 indicate enabled state, read and write permissions respectively, while bit 8 indicates if the explicit data region is a large region (i.e., a region with a bound greater than 4GB).

To access memory through explicit regions, a program must use the `h-prefixed` variants of normal load and store instructions (`hlw`, `hsw`, `hlh`, etc.). These instructions target the active region but can be changed to use a different region as described in the Standard Profile section. For example, `hlw x1, A(x0)` will succeed only if the address `Base(ActiveRegion) + A` falls within the active explicit region and the region has read permissions enabled.

In terms of size and alignment, unaligned memory operations that are split by the micro-architecture into independent operations will be checked independently by HFI. If one split faults, the original fused instruction will fault. However, the split operation that is within bounds is allowed to have visible *micro-architectural* side effects within the sandbox. To safely implement explicit bounds checks, the check must be applied to the effective address plus the size of the operand to ensure that multi-byte operations do not cross the region boundary. The smallest large explicit region is 64K; there is no minimum size on small explicit regions. The behavior of regions that do not meet these minimums is undefined.

When the runtime grants access to memory, it can choose to have an explicit region(s) for the applications heap(s) and inputs, as the SFI compiler can use h-prefixed instructions for accessing these directly. This allows it to exploit the greater flexibility of explicit regions for sizing and alignment that are necessary to support Wasm and similar systems. Notably, a Wasm runtime in the sandbox may opt to place it's own data into an implicit data region, to ensure Spectre attacks cannot be used to trick the sandbox runtime into leaking its own data.

3.3. Fault Registers

The HFI Explicit extension extends the fault status register defined in Chapter 2.4.4 with additional fields to capture explicit region faults. Specifically, when an explicit region fault occurs, the `hfi_fault_region` field indicates which explicit region caused the fault (i.e., region number 3). If the

faulted region is an explicit region, the `hfi_fault_type` will be set to `OUT_OF_BOUNDS` if the access was out of bound for the explicit region, and `INSUFFICIENT_PERMISSIONS` if the explicit region did not have sufficient permissions.

3.4. Explicit Instruction Encoding

On top of instructions introduced in HFI base extension, HFI explicit extension further introduces "explicit" memory instructions that mimic native load and store instructions but are used exclusively for explicit regions. These instructions are encoded using opcodes that fail if not implemented, ensuring that binaries using explicit regions cannot run inadvertently on non-HFI hardware. However, binaries that only use HFI's implicit regions would remain backward compatible on CPUs that don't support HFI (albeit without the isolation enforcement).

To provide full support for modern RV64GV processors, HFI defines explicit-region variants for the Base Integer set, as well as the Atomic (A), Floating Point (F, D, Q), and Vector (V) extensions.

3.4.1. Base Loads and Stores

H-prefixed base loads and stores automatically check against the bounds specified by the explicit region registers. The effective address (EA) calculation matches native instructions, but the base of the region is added to the EA to form the final virtual address. The bound check compares the EA against the region bound. Implementations typically place the bound check logic in parallel with the final address computation or TLB lookup to minimize the critical path.

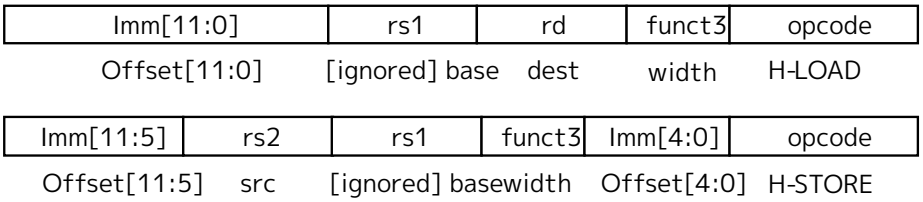

Figure 2. H-prefixed Base Loads and Stores

Table 2. Encodings for h-prefixed base loads and stores

Opcode	Instruction	Funct3	Opcode	Instruction	Funct3
0001011	`hlb`	000	0001011	`hlwu`	110
0001011	`hlh`	001	0101011	`hsb`	000
0001011	`hlw`	010	0101011	`hsh`	001
0001011	`hld`	011	0101011	`hsw`	010
0001011	`hlbu`	100	0101011	`hsd`	011
0001011	`hlhu`	101			

3.4.2. Floating Point Loads and Stores

H-prefixed Floating Point (FP) loads and stores are encoded under the same opcode as standard FP instructions but differentiated with a 3-bit width field. They behave identically to base loads/stores but utilize the floating point register file.

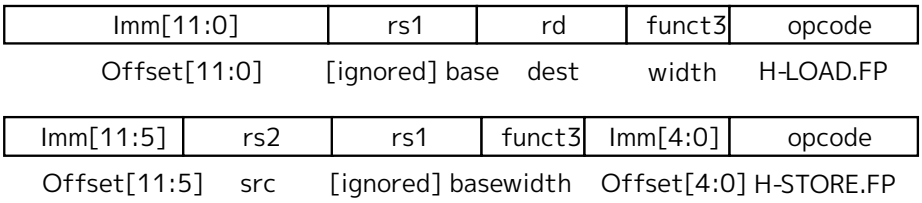

Figure 3. H-prefixed floating point loads and stores

Table 3. Encodings for h-prefixed FP loads and stores

Opcode	Instruction	Funct3	Opcode	Instruction	Funct3
0011111	`hflw`	010	0111111	`hfsw`	010
0011111	`hfld`	011	0111111	`hfsd`	011
0011111	`hflq`	100	0111111	`hfsq`	100

3.4.3. Vector Memory Operations

Vector memory instructions complicate bound checking because a single instruction accesses multiple memory locations. H-prefixed vector instructions reuse the HFI FP opcodes but retain the standard Vector extension encoding for `mop` (addressing mode) and `nf` (segment size) fields.

For strided and indexed instructions: Since these can address disparate parts of memory, HFI checks the entire segment against region bounds for every effective address. To avoid the cost of a full 64-bit adder for every element, implementations can utilize the limited segment size (dictated by `vlen` and `nf`). A cheaper 16-bit adder and 48-bit incrementor (for $\texttt{vlen} \Leftarrow \texttt{4096}$) is sufficient to calculate the segment end address.

For unit-stride instructions: These access contiguous memory. Implementations may optimize by only checking the end address of the last segment, though this requires a multiplier.

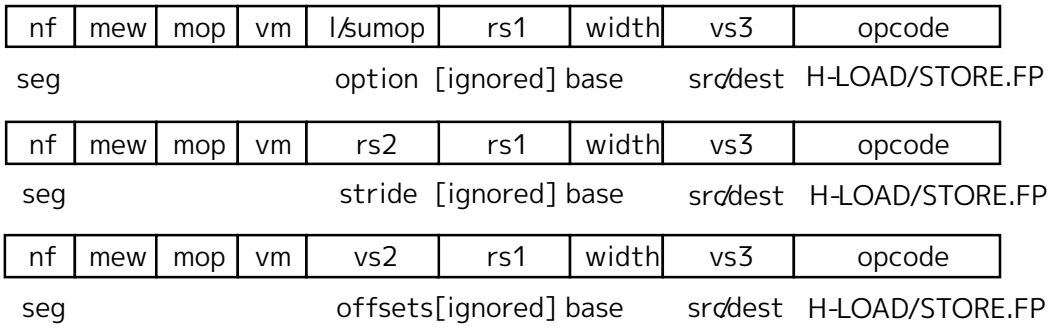

Figure 4. H-prefixed vector loads and stores

Table 4. Field encodings for vector loads and stores

`mop`	Addressing Mode	`funct3`	Width	`opcode`	Operation
00	unit stride	000	byte	0011111	h-load
01	index (unordered)	101	half-word	0111111	h-store
10	constant stride	110	word		
11	index (ordered)	111	double-word		

3.4.4. Atomic Memory Operations

For H-prefixed Atomic Memory Operations (AMOs), the hardware first computes the effective address

(EA) and adds the explicit region base to determine the final address. The EA is checked against the region bound before the atomic operation proceeds.

Fault Handling Because AMOs represent complex Read-Modify-Write sequences, a specific `AMO_FAULT` encoding (`11`) is introduced in the `hfi_fault_op` field of the fault status register. This differentiates atomic faults from simple load/store faults, mirroring standard RISC-V exception handling. Upon an out-of-bounds AMO, the hardware inhibits the write-back stage to prevent data corruption and traps to the kernel.

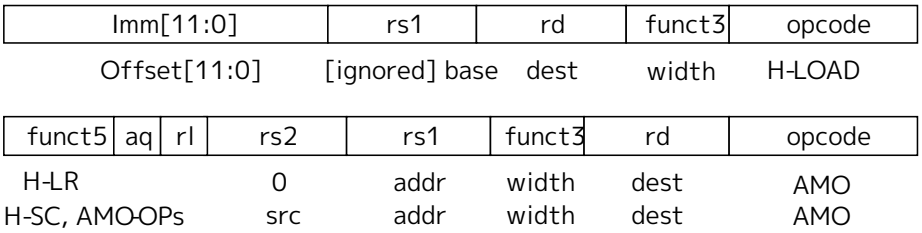

Figure 5. H-prefixed atomic loads, stores and other atomic operations

Table 5. Encodings for h-prefixed atomic memory instructions

Opcode	Instruction	Funct5	Opcode	Instruction	Funct5
0101111	`hlr`	00101	0101111	`hamoor`	01101
0101111	`hsc`	00110	0101011	`hamomin`	01110
0101111	`hamoswap`	00111	0101011	`hamomax`	01111
0101111	`hamoadd`	01001	0101011	`hamominu`	10001
0101111	`hamoxor`	01010	0101011	`hamomaxu`	10010
0101111	`hamoand`	01011			

3.5. HFI Versions (Profiles)

In the future standard profile we aim to develop up to 4 explicit data regions. The minimal profile described in this document supports: 1 implicit code region, 1 implicit data region and 1 explicit data region. The additional standard profile supports: 2 implicit code regions, 4 implicit data regions, and 4 explicit data regions. These regions are numbered as follows: explicit data region 1, implicit data region 1 and implicit code region 1 are regions 1, 2, and 3 in both profiles. In the standard profile: explicit data region 2, 3, and 4 are regions 4, 5, and 6 respectively; implicit data region 2, 3, and 4 are regions 7, 8, and 9 respectively; implicit code region 2 is region 10. Additionally the permissions bit vector operand specified in the `hfisetregionpermission` and `hfigetregionpermission` instructions is also expanded to accommodate region permissions in the same order.

Therefore, an instruction (`hfisetcurrexplicitdataregion` and `hfigetcurrexplicitdataregion`) that sets which explicit region the h-prefixed instructions will use is needed. `hfisetcurrexplicitdataregion` always precedes the h-prefixed instructions and takes the region number from its `rs1`. The instruction pair also double-checks the region number to ensure that it references an explicit region.

- `hfisetcurrexplicitdataregion(region_number_t)`
- `hfigetcurrexplicitdataregion() → region_number_t`

Table 6. Encoding for R-type Non-h-prefixed Instructions (Explicit Only)

funct3	funct7	Instruction
011	0000000	`hfigetcurrexplicitdataregion`
011	0000001	`hfisetcurrexplicitdataregion`

Design Rationale Regions exact some cost in terms of circuit area, and differing trade-offs may make sense for different use cases. Obviously more regions facilitate efficient access to more data and code concurrently, and can simplify runtime implementation.

While the minimal profile is limited in the number of regions, this can still offer meaningful benefits for certain use cases without significant concurrency or memory sharing. For example, the Google Chrome browser's Ubercage JIT isolation scheme (saelo, 2021) would be able to leverage this minimal profile for its isolation requirements (Groß saelo, 2024).

The standard profile offers an expanded number of regions. The particular number of regions was inspired by uses cases such as leveraging WebAssembly for efficient isolation of libraries from applications, and efficient isolation of different clients' code in serverless settings (Narayan et al., 2023). The main observation here is that there is greater concurrency and sharing is present than simple use cases, but this can nevertheless be handled efficiently with a handful of regions. For uses cases that need additional regions, this can be achieved by spilling and restoring regions similar to how this is done with general-purpose registers.