We store the ASID and the page table base address in the same CSR to allow the pair to be changed atomically on a context switch. Swapping them non-atomically could pollute the old virtual address space with new translations, or vice-versa. This approach also slightly reduces the cost of a context switch.

Table 4.4 shows the encodings of the MODE field when SXLEN=32 and SXLEN=64. When MODE=Bare, supervisor virtual addresses are equal to supervisor physical addresses, and there is no additional memory protection beyond the physical memory protection scheme described in Section 3.7. To select MODE=Bare, software must write zero to the remaining fields of `satp` (bits 30–0 when SXLEN=32, or bits 59–0 when SXLEN=64). Attempting to select MODE=Bare with a nonzero pattern in the remaining fields has an UNSPECIFIED effect on the value that the remaining fields assume and an UNSPECIFIED effect on address translation and protection behavior.

When SXLEN=32, the `satp` encodings corresponding to MODE=Bare and ASID[8:7]=3 are designated for custom use, whereas the encodings corresponding to MODE=Bare and ASID[8:7]≠3 are reserved for future standard use. When SXLEN=64, all `satp` encodings corresponding to MODE=Bare are reserved for future standard use.

Version 1.11 of this standard stated that the remaining fields in `satp` *had no effect when MODE=Bare. Making these fields reserved facilitates future definition of additional translation and protection modes, particularly in RV32, for which all patterns of the existing MODE field have already been allocated.*

When SXLEN=32, the only other valid setting for MODE is Sv32, a paged virtual-memory scheme described in Section 4.3.

When SXLEN=64, three paged virtual-memory schemes are defined: Sv39, Sv48, and Sv57, described in Sections 4.4, 4.5, and 4.6, respectively. One additional scheme, Sv64, will be defined in a later version of this specification. The remaining MODE settings are reserved for future use and may define different interpretations of the other fields in `satp`.

Implementations are not required to support all MODE settings, and if `satp` is written with an unsupported MODE, the entire write has no effect; no fields in `satp` are modified.

The number of ASID bits is UNSPECIFIED and may be zero. The number of implemented ASID bits, termed *ASIDLEN*, may be determined by writing one to every bit position in the ASID field, then reading back the value in `satp` to see which bit positions in the ASID field hold a one. The least-significant bits of ASID are implemented first: that is, if ASIDLEN > 0, ASID[ASIDLEN-1:0] is writable. The maximal value of ASIDLEN, termed ASIDMAX, is 9 for Sv32 or 16 for Sv39, Sv48, and Sv57.

For many applications, the choice of page size has a substantial performance impact. A large page size increases TLB reach and loosens the associativity constraints on virtually indexed, physically tagged caches. At the same time, large pages exacerbate internal fragmentation, wasting physical memory and possibly cache capacity.

After much deliberation, we have settled on a conventional page size of 4 KiB for both RV32 and RV64. We expect this decision to ease the porting of low-level runtime software and device drivers. The TLB reach problem is ameliorated by transparent superpage support in modern

SXLEN=32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing (see Section 4.3).

SXLEN=64		
Value	Name	Description
0	Bare	No translation or protection.
1–7	—	*Reserved for standard use*
8	Sv39	Page-based 39-bit virtual addressing (see Section 4.4).
9	Sv48	Page-based 48-bit virtual addressing (see Section 4.5).
10	Sv57	Page-based 57-bit virtual addressing (see Section 4.6).
11	*Sv64*	*Reserved for page-based 64-bit virtual addressing.*
12–13	—	*Reserved for standard use*
14–15	—	*Designated for custom use*

Table 4.4: Encoding of `satp` MODE field.

operating systems [2]. Additionally, multi-level TLB hierarchies are quite inexpensive relative to the multi-level cache hierarchies whose address space they map.

The `satp` register is considered *active* when the effective privilege mode is S-mode or U-mode. Executions of the address-translation algorithm may only begin using a given value of `satp` when `satp` is active.

Translations that began while `satp` *was active are not required to complete or terminate when* `satp` *is no longer active, unless an SFENCE.VMA instruction matching the address and ASID is executed. The SFENCE.VMA instruction must be used to ensure that updates to the address-translation data structures are observed by subsequent implicit reads to those structures by a hart.*

Note that writing `satp` does not imply any ordering constraints between page-table updates and subsequent address translations, nor does it imply any invalidation of address-translation caches. If the new address space's page tables have been modified, or if an ASID is reused, it may be necessary to execute an SFENCE.VMA instruction (see Section 4.2.1) after, or in some cases before, writing `satp`.

Not imposing upon implementations to flush address-translation caches upon `satp` *writes reduces the cost of context switches, provided a sufficiently large ASID space.*

4.2 Supervisor Instructions

In addition to the SRET instruction defined in Section 3.3.2, one new supervisor-level instruction is provided.

4.2.1 Supervisor Memory-Management Fence Instruction

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
SFENCE.VMA	asid	vaddr	PRIV	0	SYSTEM

The supervisor memory-management fence instruction SFENCE.VMA is used to synchronize updates to in-memory memory-management data structures with current execution. Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to explicit loads and stores. Executing an SFENCE.VMA instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before certain implicit references by subsequent instructions in that hart to the memory-management data structures. The specific set of operations ordered by SFENCE.VMA is determined by *rs1* and *rs2*, as described below. SFENCE.VMA is also used to invalidate entries in the address-translation cache associated with a hart (see Section 4.3.2). Further details on the behavior of this instruction are described in Section 3.1.6.5 and Section 3.7.2.

The SFENCE.VMA is used to flush any local hardware caches related to address translation. It is specified as a fence rather than a TLB flush to provide cleaner semantics with respect to which instructions are affected by the flush operation and to support a wider variety of dynamic caching structures and memory-management schemes. SFENCE.VMA is also used by higher privilege levels to synchronize page table writes and the address translation hardware.

SFENCE.VMA orders only the local hart's implicit references to the memory-management data structures.

Consequently, other harts must be notified separately when the memory-management data structures have been modified. One approach is to use 1) a local data fence to ensure local writes are visible globally, then 2) an interprocessor interrupt to the other thread, then 3) a local SFENCE.VMA in the interrupt handler of the remote thread, and finally 4) signal back to originating thread that operation is complete. This is, of course, the RISC-V analog to a TLB shootdown.

For the common case that the translation data structures have only been modified for a single address mapping (i.e., one page or superpage), *rs1* can specify a virtual address within that mapping to effect a translation fence for that mapping only. Furthermore, for the common case that the translation data structures have only been modified for a single address-space identifier, *rs2* can specify the address space. The behavior of SFENCE.VMA depends on *rs1* and *rs2* as follows:

- If *rs1*=`x0` and *rs2*=`x0`, the fence orders all reads and writes made to any level of the page tables, for all address spaces. The fence also invalidates all address-translation cache entries, for all address spaces.
- If *rs1*=`x0` and *rs2*≠`x0`, the fence orders all reads and writes made to any level of the page tables, but only for the address space identified by integer register *rs2*. Accesses to *global* mappings (see Section 4.3.1) are not ordered. The fence also invalidates all address-translation cache entries matching the address space identified by integer register *rs2*, except for entries containing global mappings.

- If *rs1*≠`x0` and *rs2*=`x0`, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1*, for all address spaces. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in *rs1*, for all address spaces.
- If *rs1*≠`x0` and *rs2*≠`x0`, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1*, for the address space identified by integer register *rs2*. Accesses to global mappings are not ordered. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in *rs1* and that match the address space identified by integer register *rs2*, except for entries containing global mappings.

If the value held in *rs1* is not a valid virtual address, then the SFENCE.VMA instruction has no effect. No exception is raised in this case.

When *rs2*≠`x0`, bits SXLEN-1:ASIDMAX of the value held in *rs2* are reserved for future standard use. Until their use is defined by a standard extension, they should be zeroed by software and ignored by current implementations. Furthermore, if ASIDLEN < ASIDMAX, the implementation shall ignore bits ASIDMAX-1:ASIDLEN of the value held in *rs2*.

It is always legal to over-fence, e.g., by fencing only based on a subset of the bits in rs1 *and/or* rs2*, and/or by simply treating all SFENCE.VMA instructions as having* rs1=`x0` *and/or* rs2=`x0`*. For example, simpler implementations can ignore the virtual address in* rs1 *and the ASID value in* rs2 *and always perform a global fence. The choice not to raise an exception when an invalid virtual address is held in* rs1 *facilitates this type of simplification.*

An implicit read of the memory-management data structures may return any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. The ordering implied by SFENCE.VMA does not place implicit reads and writes to the memory-management data structures into the global memory order in a way that interacts cleanly with the standard RVWMO ordering rules. In particular, even though an SFENCE.VMA orders prior explicit accesses before subsequent implicit accesses, and those implicit accesses are ordered before their associated explicit accesses, SFENCE.VMA does not necessarily place prior explicit accesses before subsequent explicit accesses in the global memory order. These implicit loads also need not otherwise obey normal program order semantics with respect to prior loads or stores to the same address.

A consequence of this specification is that an implementation may use any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. In particular, if a leaf PTE is modified but a subsuming SFENCE.VMA is not executed, either the old translation or the new translation will be used, but the choice is unpredictable. The behavior is otherwise well-defined.

In a conventional TLB design, it is possible for multiple entries to match a single address if, for example, a page is upgraded to a superpage without first clearing the original non-leaf PTE's valid bit and executing an SFENCE.VMA with rs1=`x0`*. In this case, a similar remark applies: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.*

Another consequence of this specification is that it is generally unsafe to update a PTE using a set of stores of a width less than the width of the PTE, as it is legal for the implementation to read the PTE at any time, including when only some of the partial stores have taken effect.

This specification permits the caching of PTEs whose V (Valid) bit is clear. Operating systems must be written to cope with this possibility, but implementers are reminded that eagerly caching invalid PTEs will reduce performance by causing additional page faults.

Implementations must only perform implicit reads of the translation data structures pointed to by the current contents of the `satp` register or a subsequent valid (V=1) translation data structure entry, and must only raise exceptions for implicit accesses that are generated as a result of instruction execution, not those that are performed speculatively.

Changes to the `sstatus` fields SUM and MXR take effect immediately, without the need to execute an SFENCE.VMA instruction. Changing `satp`.MODE from Bare to other modes and vice versa also takes effect immediately, without the need to execute an SFENCE.VMA instruction. Likewise, changes to `satp`.ASID take effect immediately.

The following common situations typically require executing an SFENCE.VMA instruction:

- *When software recycles an ASID (i.e., reassociates it with a different page table), it should* first *change* `satp` *to point to the new page table using the recycled ASID,* then *execute SFENCE.VMA with* rs1=`x0` *and* rs2 *set to the recycled ASID. Alternatively, software can execute the same SFENCE.VMA instruction while a different ASID is loaded into* `satp`*, provided the next time* `satp` *is loaded with the recycled ASID, it is simultaneously loaded with the new page table.*
- *If the implementation does not provide ASIDs, or software chooses to always use ASID 0, then after every* `satp` *write, software should execute SFENCE.VMA with* rs1=`x0`*. In the common case that no global translations have been modified,* rs2 *should be set to a register other than* `x0` *but which contains the value zero, so that global translations are not flushed.*
- *If software modifies a non-leaf PTE, it should execute SFENCE.VMA with* rs1=`x0`*. If any PTE along the traversal path had its G bit set,* rs2 *must be* `x0`*; otherwise,* rs2 *should be set to the ASID for which the translation is being modified.*
- *If software modifies a leaf PTE, it should execute SFENCE.VMA with* rs1 *set to a virtual address within the page. If any PTE along the traversal path had its G bit set,* rs2 *must be* `x0`*; otherwise,* rs2 *should be set to the ASID for which the translation is being modified.*
- *For the special cases of increasing the permissions on a leaf PTE and changing an invalid PTE to a valid leaf, software may choose to execute the SFENCE.VMA lazily. After modifying the PTE but before executing SFENCE.VMA, either the new or old permissions will be used. In the latter case, a page-fault exception might occur, at which point software should execute SFENCE.VMA in accordance with the previous bullet point.*

If a hart employs an address-translation cache, that cache must appear to be private to that hart. In particular, the meaning of an ASID is local to a hart; software may choose to use the same ASID to refer to different address spaces on different harts.

A future extension could redefine ASIDs to be global across the SEE, enabling such options as shared translation caches and hardware support for broadcast TLB shootdown. However, as OSes have evolved to significantly reduce the scope of TLB shootdowns using novel ASID-management techniques, we expect the local-ASID scheme to remain attractive for its simplicity and possibly better scalability.

For implementations that make `satp`.MODE read-only zero (always Bare), attempts to execute an SFENCE.VMA instruction might raise an illegal instruction exception.

4.3 Sv32: Page-Based 32-bit Virtual-Memory Systems

When Sv32 is written to the MODE field in the `satp` register (see Section 4.1.11), the supervisor operates in a 32-bit paged virtual-memory system. In this mode, supervisor and user virtual addresses are translated into supervisor physical addresses by traversing a radix-tree page table. Sv32 is supported when SXLEN=32 and is designed to include mechanisms sufficient for supporting modern Unix-based operating systems.

The initial RISC-V paged virtual-memory architectures have been designed as straightforward implementations to support existing operating systems. We have architected page table layouts to support a hardware page-table walker. Software TLB refills are a performance bottleneck on high-performance systems, and are especially troublesome with decoupled specialized coprocessors. An implementation can choose to implement software TLB refills using a machine-mode trap handler as an extension to M-mode.

Some ISAs architecturally expose virtually indexed, physically tagged *caches, in that accesses to the same physical address via different virtual addresses might not be coherent unless the virtual addresses lie within the same cache set. Implicitly, this specification does not permit such behavior to be architecturally exposed.*

4.3.1 Addressing and Memory Protection

Sv32 implementations support a 32-bit virtual address space, divided into 4 KiB pages. An Sv32 virtual address is partitioned into a virtual page number (VPN) and page offset, as shown in Figure 4.16. When Sv32 virtual memory mode is selected in the MODE field of the `satp` register, supervisor virtual addresses are translated into supervisor physical addresses via a two-level page table. The 20-bit VPN is translated into a 22-bit physical page number (PPN), while the 12-bit page offset is untranslated. The resulting supervisor-level physical addresses are then checked using any physical memory protection structures (Sections 3.7), before being directly converted to machine-level physical addresses. If necessary, supervisor-level physical addresses are zero-extended to the number of physical address bits found in the implementation.

For example, consider an RV32 system supporting 34 bits of physical address. When the value of `satp`*.MODE is Sv32, a 34-bit physical address is produced directly, and therefore no zero-extension is needed. When the value of* `satp`*.MODE is Bare, the 32-bit virtual address is translated (unmodified) into a 32-bit physical address, and then that physical address is zero-extended into a 34-bit machine-level physical address.*

31 22	21 12	11 0
VPN[1]	VPN[0]	page offset
10	10	12

Figure 4.16: Sv32 virtual address.

Sv32 page tables consist of 2^{10} page-table entries (PTEs), each of four bytes. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the `satp` register.

33 22	21 12	11 0
PPN[1]	PPN[0]	page offset
12	10	12

Figure 4.17: Sv32 physical address.

31 20	19 10	9 8	7	6	5	4	3	2	1	0
PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
12	10	2	1	1	1	1	1	1	1	1

Figure 4.18: Sv32 page table entry.

The PTE format for Sv32 is shown in Figures 4.18. The V bit indicates whether the PTE is valid; if it is 0, all other bits in the PTE are don't-cares and may be used freely by software. The permission bits, R, W, and X, indicate whether the page is readable, writable, and executable, respectively. When all three are zero, the PTE is a pointer to the next level of the page table; otherwise, it is a leaf PTE. Writable pages must also be marked readable; the contrary combinations are reserved for future use. Table 4.5 summarizes the encoding of the permission bits.

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

Table 4.5: Encoding of PTE R/W/X fields.

Attempting to fetch an instruction from a page that does not have execute permissions raises a fetch page-fault exception. Attempting to execute a load or load-reserved instruction whose effective address lies within a page without read permissions raises a load page-fault exception. Attempting to execute a store, store-conditional, or AMO instruction whose effective address lies within a page without write permissions raises a store page-fault exception.

AMOs never raise load page-fault exceptions. Since any unreadable page is also unwritable, attempting to perform an AMO on an unreadable page always raises a store page-fault exception.

The U bit indicates whether the page is accessible to user mode. U-mode software may only access the page when U=1. If the SUM bit in the `sstatus` register is set, supervisor mode software may also access pages with U=1. However, supervisor code normally operates with the SUM bit clear, in which case, supervisor code will fault on accesses to user-mode pages. Irrespective of SUM, the supervisor may not execute code on pages with U=1.

An alternative PTE format would support different permissions for supervisor and user. We omitted this feature because it would be largely redundant with the SUM mechanism (see Section 4.1.1.2) and would require more encoding space in the PTE.

The G bit designates a *global* mapping. Global mappings are those that exist in all address spaces. For non-leaf PTEs, the global setting implies that all mappings in the subsequent levels of the page table are global. Note that failing to mark a global mapping as global merely reduces performance, whereas marking a non-global mapping as global is a software bug that, after switching to an address space with a different non-global mapping for that address range, can unpredictably result in either mapping being used.

Global mappings need not be stored redundantly in address-translation caches for multiple ASIDs. Additionally, they need not be flushed from local address-translation caches when an SFENCE.VMA instruction is executed with rs2≠`x0`.

The RSW field is reserved for use by supervisor software; the implementation shall ignore this field.

Each leaf PTE contains an accessed (A) and dirty (D) bit. The A bit indicates the virtual page has been read, written, or fetched from since the last time the A bit was cleared. The D bit indicates the virtual page has been written since the last time the D bit was cleared.

Two schemes to manage the A and D bits are permitted:

- When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, a page-fault exception is raised.
- When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, the implementation sets the corresponding bit(s) in the PTE. The PTE update must be atomic with respect to other accesses to the PTE, and must atomically check that the PTE is valid and grants sufficient permissions. Updates of the A bit may be performed as a result of speculation, but updates to the D bit must be exact (i.e., not speculative), and observed in program order by the local hart. Furthermore, the PTE update must appear in the global memory order no later than the explicit memory access, or any subsequent explicit memory access to that virtual page by the local hart. The ordering on loads and stores provided by FENCE instructions and the acquire/release bits on atomic instructions also orders the PTE updates associated with those loads and stores as observed by remote harts.

  The PTE update is not required to be atomic with respect to the explicit memory access that caused the update, and the sequence is interruptible. However, the hart must not perform the explicit memory access before the PTE update is globally visible.

All harts in a system must employ the same PTE-update scheme as each other.

Prior versions of this specification required PTE A bit updates to be exact, but allowing the A bit to be updated as a result of speculation simplifies the implementation of address translation prefetchers. System software typically uses the A bit as a page replacement policy hint, but does not require exactness for functional correctness. On the other hand, D bit updates are still required to be exact and performed in program order, as the D bit affects the functional correctness of page eviction.

Implementations are of course still permitted to perform both A and D bit updates only in an exact manner.

In both cases, requiring atomicity ensures that the PTE update will not be interrupted by other intervening writes to the page table, as such interruptions could lead to A/D bits being set on PTEs that have been reused for other purposes, on memory that has been reclaimed for other purposes, and so on. Simple implementations may instead generate page-fault exceptions.

The A and D bits are never cleared by the implementation. If the supervisor software does not rely on accessed and/or dirty bits, e.g. if it does not swap memory pages to secondary storage or if the pages are being used to map I/O space, it should always set them to 1 in the PTE to improve performance.

Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv32 supports 4 MiB *megapages*. A megapage must be virtually and physically aligned to a 4 MiB boundary; a page-fault exception is raised if the physical address is insufficiently aligned.

For non-leaf PTEs, the D, A, and U bits are reserved for future standard use. Until their use is defined by a standard extension, they must be cleared by software for forward compatibility.

For implementations with both page-based virtual memory and the "A" standard extension, the LR/SC reservation set must lie completely within a single base page (i.e., a naturally aligned 4 KiB region).

4.3.2 Virtual Address Translation Process

A virtual address *va* is translated into a physical address *pa* as follows:

1. Let a be `satp`.*ppn* $\times$ PAGESIZE, and let i = LEVELS $-$ 1. (For Sv32, PAGESIZE=2^{12} and LEVELS=2.) The `satp` register must be *active*, i.e., the effective privilege mode must be S-mode or U-mode.
2. Let *pte* be the value of the PTE at address $a + va.vpn[i] \times$PTESIZE. (For Sv32, PTESIZE=4.) If accessing *pte* violates a PMA or PMP check, raise an access-fault exception corresponding to the original access type.
3. If $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, or if any bits or encodings that are reserved for future standard use are set within *pte*, stop and raise a page-fault exception corresponding to the original access type.
4. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i < 0$, stop and raise a page-fault exception corresponding to the original access type. Otherwise, let $a = pte.ppn \times$ PAGESIZE and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the *pte.r*, *pte.w*, *pte.x*, and *pte.u* bits, given the current privilege mode and the value of the SUM and MXR fields of the `mstatus` register. If not, stop and raise a page-fault exception corresponding to the original access type.
6. If $i > 0$ and $pte.ppn[i - 1 : 0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.
7. If $pte.a = 0$, or if the original memory access is a store and $pte.d = 0$, either raise a page-fault exception corresponding to the original access type, or:
    - If a store to *pte* would violate a PMA or PMP check, raise an access-fault exception corresponding to the original access type.

- Perform the following steps atomically:
  - Compare *pte* to the value of the PTE at address $a + va.vpn[i] \times \text{PTESIZE}$.
  - If the values match, set *pte.a* to 1 and, if the original memory access is a store, also set *pte.d* to 1.
  - If the comparison fails, return to step 2

8. The translation is successful. The translated physical address is given as follows:
   - $pa.pgoff = va.pgoff$.
   - If $i > 0$, then this is a superpage translation and $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
   - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$.

All implicit accesses to the address-translation data structures in this algorithm are performed using width PTESIZE.

This implies, for example, that an Sv48 implementation may not use two separate 4B reads to non-atomically access a single 8B PTE, and that A/D bit updates performed by the implementation are treated as atomically updating the entire PTE, rather than just the A and/or D bit alone (even though the PTE value does not otherwise change).

The results of implicit address-translation reads in step 2 may be held in a read-only, incoherent *address-translation cache* but not shared with other harts. The address-translation cache may hold an arbitrary number of entries, including an arbitrary number of entries for the same address and ASID. Entries in the address-translation cache may then satisfy subsequent step 2 reads if the ASID associated with the entry matches the ASID loaded in step 0 or if the entry is associated with a *global* mapping. To ensure that implicit reads observe writes to the same memory locations, an SFENCE.VMA instruction must be executed after the writes to flush the relevant cached translations.

The address-translation cache cannot be used in step 7; accessed and dirty bits may only be updated in memory directly.

It is permitted for multiple address-translation cache entries to co-exist for the same address. This represents the fact that in a conventional TLB hierarchy, it is possible for multiple entries to match a single address if, for example, a page is upgraded to a superpage without first clearing the original non-leaf PTE's valid bit and executing an SFENCE.VMA with rs1=`x0`*, or if multiple TLBs exist in parallel at a given level of the hierarchy. In this case, just as if an SFENCE.VMA is not executed between a write to the memory-management tables and subsequent implicit read of the same address: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.*

Implementations may also execute the address-translation algorithm speculatively at any time, for any virtual address, as long as `satp` is active (as defined in Section 4.1.11). Such speculative executions have the effect of pre-populating the address-translation cache.

Speculative executions of the address-translation algorithm behave as non-speculative executions of the algorithm do, except that they must not set the dirty bit for a PTE, they must not trigger an exception, and they must not create address-translation cache entries if those entries would have been invalidated by any SFENCE.VMA instruction executed by the hart since the speculative execution of the algorithm began.

For instance, it is illegal for both non-speculative and speculative executions of the translation algorithm to begin, read the level 2 page table, pause while the hart executes an SFENCE.VMA with rs1=rs2=`x0`*, then resume using the now-stale level 2 PTE, as subsequent implicit reads could populate the address-translation cache with stale PTEs.*

In many implementations, an SFENCE.VMA instruction with rs1=`x0` *will therefore either terminate all previously-launched speculative executions of the address-translation algorithm (for the specified ASID, if applicable), or simply wait for them to complete (in which case any address-translation cache entries created will be invalidated by the SFENCE.VMA as appropriate). Likewise, an SFENCE.VMA instruction with* rs1$\neq$`x0` *generally must either ensure that previously-launched speculative executions of the address-translation algorithm (for the specified ASID, if applicable) are prevented from creating new address-translation cache entries mapping leaf PTEs, or wait for them to complete.*

A consequence of implementations being permitted to read the translation data structures arbitrarily early and speculatively is that at any time, all page table entries reachable by executing the algorithm may be loaded into the address-translation cache.

Although it would be uncommon to place page tables in non-idempotent memory, there is no explicit prohibition against doing so. Since the algorithm may only touch page tables reachable from the root page table indicated in `satp`*, the range of addresses that an implementation's page table walker will touch is fully under supervisor control.*

The algorithm does not admit the possibility of ignoring high-order PPN bits for implementations with narrower physical addresses.

4.4 Sv39: Page-Based 39-bit Virtual-Memory System

This section describes a simple paged virtual-memory system for SXLEN=64, which supports 39-bit virtual address spaces. The design of Sv39 follows the overall scheme of Sv32, and this section details only the differences between the schemes.

We specified multiple virtual memory systems for RV64 to relieve the tension between providing a large address space and minimizing address-translation cost. For many systems, 512 GiB of virtual-address space is ample, and so Sv39 suffices. Sv48 increases the virtual address space to 256 TiB, but increases the physical memory capacity dedicated to page tables, the latency of page-table traversals, and the size of hardware structures that store virtual addresses. Sv57 increases the virtual address space, page table capacity requirement, and translation latency even further.

4.4.1 Addressing and Memory Protection

Sv39 implementations support a 39-bit virtual address space, divided into 4 KiB pages. An Sv39 address is partitioned as shown in Figure 4.19. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–39 all equal to bit 38, or else a page-fault exception will occur. The 27-bit VPN is translated into a 44-bit PPN via a three-level page table, while the 12-bit page offset is untranslated.

When mapping between narrower and wider addresses, RISC-V zero-extends a narrower physical address to a wider size. The mapping between 64-bit virtual addresses and the 39-bit usable address space of Sv39 is not based on zero-extension but instead follows an entrenched convention that allows an OS to use one or a few of the most-significant bits of a full-size (64-bit) virtual address to quickly distinguish user and supervisor address regions.

38 30	29 21	20 12	11 0
VPN[2]	VPN[1]	VPN[0]	page offset
9	9	9	12

Figure 4.19: Sv39 virtual address.

55 30	29 21	20 12	11 0
PPN[2]	PPN[1]	PPN[0]	page offset
26	9	9	12

Figure 4.20: Sv39 physical address.

63	62 61	60 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
N	PBMT	*Reserved*	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
1	2	7	26	9	9	2	1	1	1	1	1	1	1	1

Figure 4.21: Sv39 page table entry.

Sv39 page tables contain 2^9 page table entries (PTEs), eight bytes each. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the `satp` register's PPN field.

The PTE format for Sv39 is shown in Figure 4.21. Bits 9–0 have the same meaning as for Sv32. Bit 63 is reserved for use by the Svnapot extension in Chapter 5. If Svnapot is not implemented, bit 63 remains reserved and must be zeroed by software for forward compatibility, or else a page-fault exception is raised. Bits 62–61 are reserved for use by the Svpbmt extension in Chapter 6. If Svpbmt is not implemented, bits 62–61 remain reserved and must be zeroed by software for forward compatibility, or else a page-fault exception is raised. Bits 60–54 are reserved for future standard use and, until their use is defined by some standard extension, must be zeroed by software for forward compatibility. If any of these bits are set, a page-fault exception is raised.

We reserved several PTE bits for a possible extension that improves support for sparse address spaces by allowing page-table levels to be skipped, reducing memory usage and TLB refill latency. These reserved bits may also be used to facilitate research experimentation. The cost is reducing the physical address space, but 64 PiB is presently ample. When it no longer suffices, the reserved bits that remain unallocated could be used to expand the physical address space.

Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv39 supports 2 MiB *megapages* and 1 GiB *gigapages*, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

The algorithm for virtual-to-physical address translation is the same as in Section 4.3.2, except LEVELS equals 3 and PTESIZE equals 8.

4.5 Sv48: Page-Based 48-bit Virtual-Memory System

This section describes a simple paged virtual-memory system for SXLEN=64, which supports 48-bit virtual address spaces. Sv48 is intended for systems for which a 39-bit virtual address space is insufficient. It closely follows the design of Sv39, simply adding an additional level of page table, and so this chapter only details the differences between the two schemes.

Implementations that support Sv48 must also support Sv39.

Systems that support Sv48 can also support Sv39 at essentially no cost, and so should do so to maintain compatibility with supervisor software that assumes Sv39.

4.5.1 Addressing and Memory Protection

Sv48 implementations support a 48-bit virtual address space, divided into 4 KiB pages. An Sv48 address is partitioned as shown in Figure 4.22. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–48 all equal to bit 47, or else a page-fault exception will occur. The 36-bit VPN is translated into a 44-bit PPN via a four-level page table, while the 12-bit page offset is untranslated.

47 39	38 30	29 21	20 12	11 0
VPN[3]	VPN[2]	VPN[1]	VPN[0]	page offset
9	9	9	9	12

Figure 4.22: Sv48 virtual address.

55 39	38 30	29 21	20 12	11 0
PPN[3]	PPN[2]	PPN[1]	PPN[0]	page offset
17	9	9	9	12

Figure 4.23: Sv48 physical address.

63	62 61	60 54	53 37	36 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
N	PBMT	*Reserved*	PPN[3]	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
1	2	7	17	9	9	9	2	1	1	1	1	1	1	1	1

Figure 4.24: Sv48 page table entry.

The PTE format for Sv48 is shown in Figure 4.24. Bits 63–54 and 9–0 have the same meaning as for Sv39. Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv48 supports 2 MiB *megapages*, 1 GiB *gigapages*, and 512 GiB *terapages*, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

The algorithm for virtual-to-physical address translation is the same as in Section 4.3.2, except LEVELS equals 4 and PTESIZE equals 8.

4.6 Sv57: Page-Based 57-bit Virtual-Memory System

This section describes a simple paged virtual-memory system designed for RV64 systems, which supports 57-bit virtual address spaces. Sv57 is intended for systems for which a 48-bit virtual address space is insufficient. It closely follows the design of Sv48, simply adding an additional level of page table, and so this chapter only details the differences between the two schemes.

Implementations that support Sv57 must also support Sv48.

Systems that support Sv57 can also support Sv48 at essentially no cost, and so should do so to maintain compatibility with supervisor software that assumes Sv48.

4.6.1 Addressing and Memory Protection

Sv57 implementations support a 57-bit virtual address space, divided into 4 KiB pages. An Sv57 address is partitioned as shown in Figure 4.25. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–57 all equal to bit 56, or else a page-fault exception will occur. The 45-bit VPN is translated into a 44-bit PPN via a five-level page table, while the 12-bit page offset is untranslated.

56 48	47 39	38 30	29 21	20 12	11 0
VPN[4]	VPN[3]	VPN[2]	VPN[1]	VPN[0]	page offset
9	9	9	9	9	12

Figure 4.25: Sv57 virtual address.

55 48	47 39	38 30	29 21	20 12	11 0
PPN[4]	PPN[3]	PPN[2]	PPN[1]	PPN[0]	page offset
8	9	9	9	9	12

Figure 4.26: Sv57 physical address.

63	62 61	60 54	53 10	9 8	7	6	5	4	3	2	1	0
N	PBMT	*Reserved*	PPN	RSW	D	A	G	U	X	W	R	V
1	2	7	44	2	1	1	1	1	1	1	1	1

53 46	45 37	36 28	27 19	18 10
PPN[4]	PPN[3]	PPN[2]	PPN[1]	PPN[0]
8	9	9	9	9

Figure 4.27: Sv57 page table entry.

The PTE format for Sv57 is shown in Figure 4.27. Bits 63–54 and 9–0 have the same meaning as for Sv39. Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv57 supports 2 MiB *megapages*, 1 GiB *gigapages*, 512 GiB *terapages*, and 256 TiB *petapages*, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

The algorithm for virtual-to-physical address translation is the same as in Section 4.3.2, except LEVELS equals 5 and PTESIZE equals 8.

Chapter 5

"Svnapot" Standard Extension for NAPOT Translation Contiguity, Version 1.0

In Sv39, Sv48, and Sv57, when a PTE has N=1, the PTE represents a translation that is part of a range of contiguous virtual-to-physical translations with the same values for PTE bits 5–0. Such ranges must be of a naturally aligned power-of-2 (NAPOT) granularity larger than the base page size.

The Svnapot extension depends on Sv39.

i	*pte.ppn*[*i*]	Description	*pte.napot_bits*
0	`x xxxx xxx1`	*Reserved*	–
0	`x xxxx xx1x`	*Reserved*	–
0	`x xxxx x1xx`	*Reserved*	–
0	`x xxxx 1000`	64 KiB contiguous region	4
0	`x xxxx 0xxx`	*Reserved*	–
≥ 1	`x xxxx xxxx`	*Reserved*	–

Table 5.1: Page table entry encodings when *pte*.N=1

NAPOT PTEs behave identically to non-NAPOT PTEs within the address-translation algorithm in Section 4.3.2, except that:

- If the encoding in *pte* is valid according to Table 5.1, then instead of returning the original value of *pte*, implicit reads of a NAPOT PTE return a copy of *pte* in which $pte.ppn[pte.napot\_bits - 1 : 0]$ is replaced by $vpn[i][pte.napot\_bits - 1 : 0]$. If the encoding in *pte* is reserved according to Table 5.1, then a page-fault exception must be raised.
- Implicit reads of NAPOT page table entries may create address-translation cache entries mapping $a + va.vpn[j] \times \text{PTESIZE}$ to a copy of *pte* in which $pte.ppn[pte.napot\_bits - 1 : 0]$ is replaced by $vpn[0][pte.napot\_bits - 1 : 0]$, for any or all j such that $j[8 : napot\_bits] = i[8 : napot\_bits]$, all for the address space identified in *satp* as loaded by step 0.

The motivation for a NAPOT PTE is that it can be cached in a TLB as one or more entries representing the contiguous region as if it were a single (large) page covered by a single translation. This compaction can help relieve TLB pressure in some scenarios. The encoding is designed to fit within the pre-existing Sv39, Sv48, and Sv57 PTE formats so as not to disrupt existing implementations or designs that choose not to implement the scheme. It is also designed so as not to complicate the definition of the address-translation algorithm.

The address translation cache abstraction captures the behavior that would result from the creation of a single TLB entry covering the entire NAPOT region. It is also designed to be consistent with implementations that support NAPOT PTEs by splitting the NAPOT region into TLB entries covering any smaller power-of-two region sizes. For example, a 64 KiB NAPOT PTE might trigger the creation of 16 standard 4 KiB TLB entries, all with contents generated from the NAPOT PTE (even if the PTEs for the other 4 KiB regions have different contents).

In typical usage scenarios, NAPOT PTEs in the same region will have the same attributes, same PPNs, and same values for bits 5–0. RSW remains reserved for supervisor software control. It is the responsibility of the OS and/or hypervisor to configure the page tables in such a way that there are no inconsistencies between NAPOT PTEs and other NAPOT or non-NAPOT PTEs that overlap the same address range. If an update needs to be made, the OS generally should first mark all of the PTEs invalid, then issue SFENCE.VMA instruction(s) covering all 4 KiB regions within the range (either via a single SFENCE.VMA with rs1=`x0`*, or with multiple SFENCE.VMA instructions with* rs1≠`x0`*), then update the PTE(s), as described in Section 4.2.1, unless any inconsistencies are known to be benign. If any inconsistencies do exist, then the effect is the same as when SFENCE.VMA is used incorrectly: one of the translations will be chosen, but the choice is unpredictable.*

If an implementation chooses to use a NAPOT PTE (or cached version thereof), it might not consult the PTE directly specified by the algorithm in Section 4.3.2 at all. Therefore, the D and A bits may not be identical across all mappings of the same address range even in typical use cases The operating system must query all NAPOT aliases of a page to determine whether that page has been accessed and/or is dirty. If the OS manually sets the A and/or D bits for a page, it is recommended that the OS also set the A and/or D bits for other NAPOT aliases as appropriate in order to avoid unnecessary traps.

Just as with normal PTEs, TLBs are permitted to cache NAPOT PTEs whose V (Valid) bit is clear.

Depending on need, the NAPOT scheme may be extended to other intermediate page sizes and/or to other levels of the page table in the future. The encoding is designed to accommodate other NAPOT sizes should that need arise. For example:

i	*pte.ppn*[*i*]	Description	*pte.napot_bits*
0	`x xxxx xxx1`	8 KiB contiguous region	1
0	`x xxxx xx10`	16 KiB contiguous region	2
0	`x xxxx x100`	32 KiB contiguous region	3
0	`x xxxx 1000`	64 KiB contiguous region	4
0	`x xxx1 0000`	128 KiB contiguous region	5
...	...	...	...
1	`x xxxx xxx1`	4 MiB contiguous region	1
1	`x xxxx xx10`	8 MiB contiguous region	2
...	...	...	...

In such a case, an implementation may or may not support all options. The discoverability mechanism for this extension would be extended to allow system software to determine which sizes are supported.

Other sizes may remain deliberately excluded, so that PPN bits not being used to indicate a valid NAPOT region size (e.g., the least-significant bit of pte.ppn[*i*]*) may be repurposed for other uses in the future.*

However, in case finer-grained intermediate page size support proves not to be useful, we have chosen to standardize only 64 KiB support as a first step.