**RISCV-IMA NOMMU Linux Specifications**

**Before starting linux:**

1. HART ID must be stored in x10. For nommu we can initialize it with 0x00.
2. Valid DTB pointers in RAM must be stored in x11.
3. Set CSR MISA register to reflect the ISA we are using.
4. Set machine vendor id.
5. Enable machine mode by write to the mstatus register.

**CSR Registers and relevant fields:**

| General Info: | Description |
| --- | --- |
| CSR #: 0x340  Name:  mscratch (Machine Mode Scratch Register) | It stores a pointer to a memory area that holds the machine-mode context for each hart (hardware thread). When a trap occurs in machine mode, mscratch is exchanged with a user register to save the machine-mode state. For example, suppose we want to save all the general-purpose registers (x0-x31) in an interrupt handler. Instead of using labels for each register value, we can do this: store x1 in mscratch, load the address of the memory area for saving registers into x1, and then use x1 as a base register with offsets to store all the other registers. |
| CSR #: 0x305  Name:  Mtvec (Machine Trap Vector Base Address) | It's a register holding trap vector configuration consisting of a vector base address (BASE) and a vector mode (MODE). The register is used as a machine trap-handler (subroutine) base address. |
| CSR #: 0x304  Name:  Mie (machine interrupt-enable register) | It's a register used to enable or disable various interrupt sources. It contains interrupt enable bits, and each bit corresponds to an interrupt cause number in both mip and mie. An interrupt is taken if the corresponding bits in both mip and mie are set, and if interrupts are globally enabled. |
| CSR #: 0x344  Name:  Mip (machine interrupt-pending register) | It's a register storing the pending status of various interrupt sources.  software, timer, and external interrupt signals (msip/mtip/meip signals in the mip register). |
| CSR #: 0x341  Name:  Mepc (machine exception program counter register) | It's a register holding the virtual address of the instruction that caused the most recent trap or exception. When a trap is taken into M-mode, mepc is written with the virtual address of the interrupted instruction. |
| CSR #: 0x300  Name:  Mstatus (machine status register) | Holds the current operating state of the hart. It also hold an mie bit. This bit is essentially the global IRQ enable flag. The Machine Previous Privilege (MPP) is a two-bit field that holds the privilege mode that the hart was in when an exception was taken into M-mode. |
| CSR #: 0x342  Name:  Mcause (machine trap cause register) | It's a register storing the cause of the most recent trap or exception, represented as the trap number. |
| CSR #: 0x343  Name:  Mtval (machine trap value register) | which holds the value (bad address or instruction) that caused the most recent trap or exception. So in the case of load/store traps it stores the address that caused the trap and in case of other traps it stores the pc (instruction) that caused the trap.  When a trap is taken into M-mode, mtval is either set to zero or written with exception-specific information to assist software in handling the trap. Otherwise, mtval is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set mtval informatively and which may unconditionally set it to zero.  When a hardware breakpoint is triggered, or an address-misaligned, access-fault, or page-fault exception occurs on an instruction fetch, load, or store, mtval is written with the faulting virtual address. |
| CSR #: 0xf11  Name:  Mvendorid (machine vendor id) | It's a 32-bit read-only register providing the JEDEC manufacturer ID of the core provider. A value of 0 can be returned if the field is not implemented or if this is a non-commercial implementation. |
| CSR #: 0x301  Name:  Misa (machine ISA reg) | read-write register reporting the ISA supported by the hart.  We have the following configuration for our architecture:   * XLEN=32, Extensions: IMA+X 🡪 0x40401101 |
| CSR #: 0x105  Name:  stvec (Supervisor Trap Vector Base Address) | It's a register used by the system for supervisor traps. It's not necessary to implement it, but interrupts must be enabled using mstatus and flags (WFI 🡪 wait for interrupts) to allow supervisor traps such as sleep. |
| CSR #: 0x302  Name: medeleg (Machine Exception Delegation)  CSR #: 0x303  Name:  mideleg (Machine Exception Delegation) | In systems with only M and U privilege modes, setting a bit in medeleg or mideleg delegates the corresponding trap in U-mode to the U-mode trap handler.  This is used for mret instruction. We may not need to implement this. The emulator simply takes access to these registers and changes mstatus and flags to do mret.  We will need sedeleg and sideleg to allow S-mode to delegate traps to U-mode. |
| MMIO (CLINT)  Name:  Mtime (machine time reg) | A memory-mapped machine-mode read-write register that provides a real-time counter. It must increment at a constant frequency and the platform should have a mechanism to determine its timebase. The register will wrap around if the count overflows. |
| MMIO (CLINT)  Name: Mtimecmp (machine time compare reg) | A memory-mapped machine-mode read-write register that generates a timer interrupt when mtime is greater than or equal to mtimecmp (treated as unsigned integers). The interrupt remains posted until mtimecmp becomes greater than mtime, typically as a result of writing to mtimecmp. The interrupt is only taken if interrupts are enabled and the MTIE bit is set in the mie register.  Note: Writes to mtime and mtimecmp are guaranteed to be reflected in MTIP eventually, but not necessarily immediately. |
| CSR #: 0x136 to 0x140  Name: custom CSRs used for supervisor standard read/write | CSRs 0x136 to 0x139 are used as write only registers. 0x140 is used as read only. These registers are not exactly used by linux. It may be used by the emulator for debugging purposes?  0x136 – decimal integer  0x137 – 8-digit hexadecimal number with leading zeros  0x138 – null terminated string in image  0x139 – character  0x140 – keyboard input waiting to be read |

Note we may need a flags register (we can just use any of the R/W CSR registers) to hold the following information:

* Bits 0..1 = privilege.
* Bit 2 = WFI (Wait for interrupt)
* Bit 3+ = Load/Store reservation LSBs.

Not sure why flags is storing privilege if it can be accessed using mstatus. We might need it for WFI and Load/Store reservation LSBs. Load/Store reservation is a technique to achieve synchronization in multithreading without using locks. It consists of two instructions: load-reserved (LR) and store-conditional (SC). LR reads the current value of a memory location and marks it as reserved. SC writes a new value to the same memory location only if it is still reserved, otherwise it fails. This way, multiple threads can perform atomic read-modify-write operations on shared data without interfering with each other.

**UART and Keyboard Interrupt Handling:**

* When a key is hit on the keyboard, an electrical signal is generated and transmitted to the UART interface.
* The UART interface converts the electrical signal into a serial data stream and stores it in the receive buffer in MMIO port 1. At the same time, the UART interface updates the status control register in MMIO port 2 to indicate that new data is available in the receive buffer.
* The processor, which is connected to the MMIO interface using AXI protocol, continuously monitors the status control register in MMIO port 2 to check for new data.
* When the processor detects that the status control register has been updated, it reads the receive buffer from MMIO port 1 using AXI read transaction.
* The processor checks whether the receive buffer is empty or not. If the receive buffer is not empty, the processor acknowledges the interrupt to the UART interface by writing to the status control register in MMIO port 2 to indicate that the interrupt has been handled.
* The UART interface generates an interrupt signal to the processor's interrupt controller.
* The interrupt controller prioritizes the UART interrupt and forwards it to the processor.
* The processor saves the current context, which includes the program counter and register values, to the stack, and then jumps to the interrupt service routine (ISR) associated with the UART interrupt.
* The ISR reads the data from the receive buffer and processes it according to the software requirements.
* After the ISR has finished processing the keyboard input, the processor restores the saved context from the stack and resumes execution of the interrupted program.
* The UART in the RISC-V32IMA system is emulated using two memory-mapped registers: the status register and the data buffer register.
  + The status register (x10000000) 🡪 indicates if there is any data waiting in the receive buffer.
  + Data buffer register (x10000005) 🡪 holds the received data or the data to be transmitted.
* When a key is pressed, UART writes a word to the MMIO data buffer register and generates an interrupt. The UART MMIO status buffer is then polled to determine whether data is available for reading. If it is then we read data from the data buffer register and send it to riscv.
* Once riscv realizes that an interrupt has occurred, it must then service it.

**RISCV Traps (Hardware and Software Interrupts):**

CLINT has three purposes:

1. To generate software interrupts to other HARTs – not relevant for us as there is only 1 HART.
2. To read the current time
3. To set the timer compare register (mtimecmp).

Types of TRAPs:

provides machine-level software interrupts (IPI) and machine-level timer interrupts for each HART on a RISC-V platform

* Hardware interrupts: Usually these interrupts are generated from an external device. It is asynchronous in nature, and it doesn’t increment the PC.
  + Timer – handled by CLINT (Note we don’t need ACLINT because we only have 1 core, if we had multiple then we would have to use ACLINT).
    - Bits mip.MTIP and mie.MTIE are the interrupt-pending and interrupt-enable bits for machine timer interrupts. MTIP is read-only in mip and is cleared by writing to the memory-mapped machine-mode timer compare register.
  + External devices – priority of interrupts from multiple sources handled by PLIC. Note that we will not need to implement a PLIC if UART is the only external source that is sending interrupts. If any other external source also sends an interrupt, then we must also implement PLIC. If we do implement a PLIC however, we need to take care of 2 types of hardware interrupts:
    - Maskable Interrupts – These interrupts can be delayed when the CPU receives higher priority interrupts.
    - Non-Maskable Interrupt – It is not possible to delay these interrupts. The CPU should consider them immediately.
    - \*Bits mip.MEIP and mie.MEIE are the interrupt-pending and interrupt-enable bits for machine-level external interrupts.
* Software Interrupts: Usually these interrupts are caused by an instruction in the program. It is synchronous in nature, and it increments the PC. There can only be 1 software interrupt at a time so no priority between different software interrupts needed.
  + Ecall – Used to do a syscall. This is a request of a user program for operating system services and it crosses privilege boundaries in a well-controlled manner. This can also be used for signals (linux does this).
  + Exceptions (can also be called TRAPs) – This is a request of a user program for operating system services, and it crosses privilege boundaries in a well-controlled manner. The following TRAPs need to be implemented:
    - 0: Instruction address misaligned
    - 1: Instruction access fault
    - 2: Illegal instruction
    - 3: Breakpoint
    - 4: Load address misaligned
    - 5: Load access fault
    - 6: Store/AMO address misaligned
    - 7: Store/AMO access fault
    - 8: Environment call from U-mode
    - 9: Environment call from S-mode
    - 10: Environment call from M-mode
    - 11: Instruction page fault
    - 12: Load page fault
    - 13: Store/AMO page fault
    - \* Note: divide by zero is not implemented in riscv, software is expected to deal with it. We might need to implement some hardware (check stack overflow sources)
  + \* Bits mip.MSIP and mie.MSIE are the interrupt-pending and interrupt-enable bits for machine-level software interrupts.

On any Trap, RISCV must do the following:

* Hardware Interrupts (also includes timer interrupts):
  + Entry:
    - mcause.interrupt = 1; mcause.exception\_code = I
    - mstatus.mpie = 1, save previous interrupt enable. (See ISR stack.)
    - mstatus.mpp = Previous privilege mode (m, s or u).
    - mstatus.mie = 0, interrupts are disabled unless the ISR re-writes this.
    - mepc = Interrupted PC, save the return address.
    - IF mtvec.Mode == Direct(0) THEN; PC = (mtvec & ~0xF)
    - IF mtvec.Mode == Vectored(1) THEN; PC = (mtvec & ~0xF) + (I \* 4)
  + Exit (mret):
    - mstatus.mie = mstatus.mpie, restore interrupt enable
    - Privilege mode = mstatus.mpp, restore privilege
    - mstatus.mpp = mstatus.mpie = 0
    - PC = mepc
* Software Interrupts:
  + Entry:
    - mcause.interrupt = 0; mcause.exception\_code = E
    - mstatus.mpie = mstatus.mie, save previous interrupt enable
    - mstatus.mpp = Previous privilege mode (m, s or u)
    - mstatus.mie = 0
    - Privilege mode = m
    - mtval = Exception specific information related to the cause.
    - mepc = PC, Instruction that caused trap.
    - PC = (mtvec & ~0xF) (Exceptions are not vectored)
  + Exit (mret):
    - Same as hardware interrupts

mtvec stores the subroutine for software / hardware interrupts in the following way:

* mtvec.Mode, the lower 2 bits of the vector.
  + When this is 0 (direct), it points to a single interrupt service routine (ISR).
  + When this is 1 (vectored), it points to the base of table of ISR entry points (4 bytes per entry) 🡪 This is basically our IDT.

Priority:

1. Machine external interrupts
2. Machine software interrupts
3. Machine timer interrupts

In Linux, the timer interrupt has the highest priority followed by software interrupts and then external interrupts such as the UART keyboard interrupt.

In a RISC-V system, the priority between timer and software interrupts is typically determined by the CLIC (Core Local Interrupt Controller), which is responsible for handling local interrupts on a per-core basis.

The CLIC receives interrupt requests from various sources, such as the timer and software interrupt signals, and it prioritizes these interrupts based on their importance or urgency. The priority scheme used by the CLIC may vary depending on the specific implementation and configuration of the RISC-V system, but typically the timer interrupts have higher priority than software interrupts.

The CLINT (Core Local Interruptor) provides the timer interrupt signals to the CLIC, and it is responsible for generating timer interrupts on a per-core basis. The CLINT does not typically determine the priority between the timer and software interrupts, but instead simply provides the timer interrupt signals to the CLIC for further processing.

The CLIC is the interrupt controller integrated into the RISC-V processor, and it can be used to handle and prioritize interrupt requests from various sources, including CLINT, UART, keyboard, and software interrupts in a single-core RISC-V system.

The only cross-HART communication is through a software interrupt, which can be triggered through the core-local interrupter (CLINT). Each HART has a section in the CLINT to trigger a software interrupt. Since it is memory-mapped, any hart can write to any other hart’s software interrupt pin.

Additional Functionality Required:

* Wait for Interrupt (WFI) Instruction – The WFI instruction is a hint to the system that the current hart can pause until an interrupt needs attention. It can also suggest that this hart should get priority for interrupts. WFI can be used in any mode, except when TW=1 in mstatus. If an interrupt happens while the hart is paused, it will be handled by the next instruction and resume after the WFI. The WFI instruction may do nothing if the system does not support it. The WFI instruction works even when interrupts are globally disabled, but only for locally enabled interrupts at any level. Used by linux to make riscv go to sleep.
* Trap Return (MRET) Instruction – MRET returns from traps in machine mode. They restore the previous interrupt enable and privilege mode settings and jump to the address saved in the xepc register. They may or may not clear the LR reservation for atomic operations, so trap handlers should do it manually if needed.
* Reset – When a hart is reset, it starts in machine mode with interrupts and memory privilege disabled. It also uses little-endian memory accesses if supported. It enables all the supported ISA extensions and sets the pc to a predefined address (0x80000000). It records the reason for the reset in mcause, which can be different for different types of resets. The reset values of mcause may overlap with other exception values, but they can be distinguished by the pc value. Used for restart/shutdown.
  + syscon (system configuration) are some MMIO regs used by LINUX to:
    - shutdown -> sw rs2, 0(rs1) -> where rs2 will be 3 and rs1 will 0x11100000
    - restart -> sw rs2, 0(rs1) -> where rs2 will be x7777 and rs1 will 0x11100000

\*\*Note that the entire interrupt section is very different to what we learned in x86.

**RAM**

* 64 MB
* Initialize RAM with 0s the put linux image on it
* Normal code -> MMIO Regs? -> DTB -> core
* PC starts at 0x80000000
* location of DTB is at the end of memory (right before the core)

**DTB (Device Tree Blob):**

* Around 1536 bytes
* A device tree blob is a way of describing the hardware of a system in a binary format. To create one for RISC-V, you need to write a source file that follows the RISC-V rules and compile it with a tool such as device tree compiler. You can then load it into your system and pass its address to the software that needs it.

**MMU (memory management unit):**

* Translates 32 Virtual Address into a 34 bit Physical Address
* CSR needed: satp (similar to cr3 in x86)
  + Mode (1 bit) - only 2 relevant modes for a 32 bit system
    - 0x0 - No translation
    - 0x1 - Page based 32-bit virtual addressing
  + ASID (9 bits) - essentially just PID
  + PPN (22 bits) - Physical address of the top of the level 1 page table (level 1 is essentially page directory table). We take the 32 bit physical address and right shift it by 12 bits. This means that the level 2 page table must be 2^12 (4KB) aligned.
* CSR needed: mstatus // sstatus
  + The SUM bit (permit Supervisor User Memory access) allows S-mode to access U-mode memory regions when set to 1.
  + SUM does not allow S-mode to execute instructions from user pages.
  + SUM helps prevent supervisor software from accidentally accessing user memory.
  + The MXR bit (Make eXecutable Readable) allows read-only pages to be also executable. If 'mxr' is set to 1, then the MMU allows execution of code from pages that are marked read-only. If 'mxr' is 0, then the MMU treats pages marked read-only as non-executable.
  + The MPRV bit allows M-mode to temporarily change the privilege level for load and store operations.
  + The sstatus register is the exact same register as the mstatus register, but when we make writes to it, the CPU will not change any machine-only bits. Any change that you make to the mstatus bits such as SUM/MXR will be reflected in status and vice versa.
* There are a total of 2^10 (1024) entries in each table, and since each entry is 32 bits, the size of the page table is 4KB (the same as a page).
* The MMU is not used in machine mode because machine mode uses physical memory addresses. The MMU is only used in supervisor or user mode when the MMU is switched on by setting the MODE field to 1. This allows the MMU to use the page table base address in the SATP register and perform the address translation according to the Sv32 mode.
* Virtual Address to Physical Address Translation:
  + Virtual Address (32 bits):
    - VPN[1] (10 bits) - Index into the level 1 page table. Dereferencing this will give us a page table entry. If the PTE has XWR bits as all 0s as it is a pointer to the next level of the page table (level 0 is essentially page table). It doesn’t have to point to the next table, it can just point to a 4MB page (if it is a leaf)
    - VPN[0] (10 bits) - Index into the level 0 page table. Gives the remaining bits of the physical address PPN[1] (12 bits) and PPN[0] (10 bits). This PTE must have XWR bits as 1 (as it points to a page) otherwise throw a page fault.
    - Page Offset (12 bits) - offset for a byte in the page.
  + Physical Address (34 bits):
    - PPN[1] (12 bits) - Comes from PTE level 0 PTE
    - PPN[0] (10 bits) - Comes from PTE level 0 PTE
    - Page Offset (12 bits) - Comes from the virtual address to index into the page.
  + Page Table Entry (32 bits):
    - PPN[1] (12 bits) -
    - PPN[0] (10 bits) -
    - RSW (2 bits) -
    - D (1 bit) - Everytime we write something to memory, the processor must set this as 1.
    - A (1 bit) - Access. Everytime a page is accessed (read, stored, or fetched from). The CPU will set it to 1.
    - G (1 bit) - Global space means that it belongs to all address space identifiers (PIDs). This is usually only done for certain I/O and virtual, dynamic shared objects, such as gettimeofday().
      * Not setting this will simply result in slower performance. But setting it may lead to a bug.
    - U (1 bit) - The U bit stands for user page.
      * If U = 1, then the page can be accessed by user mode, subject to the protection flags.
      * If U = 0, then the page can only be accessed by supervisor mode or higher, subject to the protection flags. If the U bit is set to 0 and a user mode process tries to translate the address, it is met with a page fault.
      * The OS (supervisor level) will only be able to access a user page if mstatus.SUM is 1.
      * mmu\_translate function can be used by the operating system to access user pages without causing a page fault.
    - X (1 bit) - Execute (for instructions). Fetching an instruction from a page that does not have execute permissions raises a fetch page page-fault exception.
    - W (1 bit) - Write (stores) Executing a store, store-conditional (regardless of success) or AMO whose effective address lies within a page without write permissions raises a store page fault exception.
    - R (1 bit) - Load (load). Executing a load or load reserved instruction whose effective address lies within a page without read permissions raises a load page fault exception.
    - V (1 bit) - Valid bit. Throw a page fault if this PTE is accessed.
  + You can have a leaf at level 1, meaning that level 1 can point to a larger 4 MB page. To declare a leaf node level 1 PTE must have XWR set to 1. For a 4MB page, only PPN[1] requires translation. Everything else is taken from the virtual address (VPN[0] = PPN[0], along with the page offset). This is because the page is now 4MB aligned. Generally if XWR are 0s then it is known as a leaf entry otherwise it is known as a branch entry.

Important note from the spec:

*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. The PTE update 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.*

Virtual Address Translation Process:

1. Find the top of level 2’s page table by multiplying the page number (satp.ppn) with the page size (PAGESIZE). This determines the starting address of the current page table level. Also initialize i = Levels - 1
   1. Equation: a = satp.ppn \* PAGESIZE (this is ‘a’ right shift by 12 for a 4KB page)
2. Access the page table entry (PTE) at the virtual address's page table index by adding the virtual page number for the current level (va.vpn[i]) multiplied by the PTE size (PTESIZE) to the starting address calculated in step 1. If this access violates any protection checks, raise an access exception corresponding to the original access type.
   1. Equation: pte = \*(a + va.vpn[i] \* PTESIZE)
3. If the PTE is invalid or writable but not readable, raise a page-fault exception corresponding to the original access type.
4. If the PTE is a pointer to the next level of the page table (WRX of pte is 1), update the index level (i = i - 1) and calculate the physical address of the next level of the page table by multiplying the page number in the PTE (pte.ppn) by the page size (PAGESIZE). Raise a page fault if i < 0. Now go back to step 2.
   1. Equation: a = pte.ppn \* PAGESIZE
5. If the PTE is a leaf entry, check if the requested memory access is allowed by the PTE's read (pte.r), write (pte.w), execute (pte.x), and user (pte.u) bits, given the current privilege mode and the values of the SUM and MXR fields of the mstatus register. If not, raise a page-fault exception corresponding to the original access type.
6. If this is a misaligned superpage, raise a page-fault exception corresponding to the original access type. It is a misaligned superpage when:
   1. Equation: i > 0 and pte.ppn[i-1:0]
7. If the PTE is not accessed (pte.a is 0) or the memory access is a store but the PTE is not dirty (pte.d is 0), either raise a page-fault exception corresponding to the original access type or update the PTE's accessed and dirty bits. If updating the PTE's accessed and dirty bits violates any protection checks (PMA or PMP), raise an access exception corresponding to the original access type. Note that this update and the loading of pte in tep 2 must be atomic (no intervening store to PTE has occurred in between).
8. If the translation is successful, calculate the physical address from the virtual address and the PTE's page number fields, taking into account whether this is a superpage translation.
   1. pa.pgoff = va.pgoff.
   2. If i > 0, then this is a superpage translation and pa.ppn[i − 1 : 0] = va.vpn[i − 1 : 0].
   3. pa.ppn[LEVELS−1 : i] = pte.ppn[LEVELS−1 : i].

\*\*Note we don’t need to use sptbr csr as that was used in older versions of riscv and is no longer relevant now.

The SATP register is immediate. As soon as we set MODE=1, any memory transaction (read, write, or execute) will now go through the MMU as long as the current privilege level is not machine mode (no mmu there). This includes the instruction fetch cycle! Things will come to a halt if they are not mapped correctly at this point.

**SBI (Supervisor Binary Interface) in a MMU system**

Firmware vs Bootloaders:

* Firmware is responsible for initializing the hardware and providing low-level services to the operating system, stored in non-volatile memory.
* Bootloader is responsible for initializing the system, configuring hardware, and loading the operating system kernel into memory.
* Bootloader runs as a standalone program loaded into memory, while firmware is executed as part of the system initialization process.
* Firmware is specific to the hardware platform and provided by the system vendor, while bootloader is usually provided by the operating system vendor.

Bootloader in NoMMU vs MMU systems:

* In a NoMMU system, the bootloader is responsible for initializing the hardware and loading the kernel into memory, but it doesn't need to set up any memory mappings. A bootloader for NoMMU can be a very simple program (for example a start or a jump) since it doesn't need to initialize the MMU (because it doesn't exist).
* In a MMU system, the bootloader needs to set up initial memory mappings before loading the kernel into memory and may also need to interact with firmware through SBI to perform additional system initialization tasks.

SBI:

* SBI is a standardized interface for the operating system kernel to interact with firmware in an MMU system.
* SBI provides function calls for the kernel to perform various tasks, such as querying system information, setting up memory mappings, and configuring device drivers.
* SBI is needed in an MMU system because the firmware and the kernel operate at different privilege levels, and the kernel needs to make requests to the firmware through the SBI to access system resources and hardware that are not available to it directly.

Boot Linux MMU without a bootloader:

* It is not possible to boot Linux MMU without a bootloader.
* BBL is one possible bootloader used for Linux and sets up some attributes of the MMU, such as the satp register and pmpcfg and pmpaddr registers. BBL also calls the SBI function sbi\_hart\_start to start the Linux kernel on each hart. The Linux kernel initializes the MMU by setting up page tables or mappings.
* While SBI provides some functions to help the kernel manage the MMU, the kernel still does most of the work to initialize the MMU itself.
* SBI is not strictly needed for MMU RISC-V, but it is recommended and widely used.
* Also, there may be other ways to transfer control to Linux without using SBI, such as using a custom bootloader like FVM.

**Sources:**

**Mini-rv32IMA:**

* <https://github.com/cnlohr/mini-rv32ima>

**CSR:**

* <https://five-embeddev.com/quickref/csrs.html>
* <https://book.rvemu.app/hardware-components/03-csrs.html#:~:text=The%20trap%20delegation%20registers%2C%20medeleg%20for%20machine-level%20exception,at%200x302%20and%20mideleg%20is%20allocated%20at%200x303>.
* <https://domipheus.com/blog/designing-a-risc-v-cpu-in-vhdl-part-18-control-and-status-register-unit/>
* <https://raw.githubusercontent.com/riscv/virtual-memory/main/specs/663-Svpbmt.pdf>

**UART:**

* <https://ece353.engr.wisc.edu/serial-interfaces/uart-advanced-features/#:~:text=UART%20interrupts%20allow%20the%20main%20application%20to%20carry,a%20receive%20interrupt%20when%20new%20data%20has%20arrived>.

**I/M/A extensions:**

* <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=21>
* <https://github.com/johnwinans/rvalp/releases>
* <https://msyksphinz-self.github.io/riscv-isadoc/html/rva.html>

**Linux:**

* <https://github.com/torvalds/linux>

**TRAPs (Timers(CLINT), Hardware Interrupts, Software Interrupts):**

* <https://chromitem-soc.readthedocs.io/en/latest/clint.html>
* <https://mullerlee.cyou/2020/07/09/riscv-exception-interrupt/#:~:text=We%20can%20enable%20interrupts%20in%20every%20mode%20by,and%20waiting%20the%20interrupt%20handler%20to%20reset%20it>
* <https://github.com/pulp-platform/clint/tree/master>
* <https://github.com/riscv/riscv-aclint>
* <https://five-embeddev.com/quickref/interrupts.html>
* <https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc>
* <https://www.youtube.com/watch?v=l7JIry6PEX4>
* <https://stackoverflow.com/questions/64863737/risc-v-software-interrupts>
* <https://mi-v-ecosystem.github.io/SoftConsole-Documentation/SoftConsole-v2021.3/troubleshooting/riscv_trap.html>
* <https://marz.utk.edu/my-courses/cosc562/riscv/>
* <https://stackoverflow.com/questions/70876942/why-has-risc-v-output-1-or-xffffffff-from-dividing-by-0>
* <https://five-embeddev.com/riscv-isa-manual/latest/machine.html>
* <https://chromitem-soc.readthedocs.io/en/latest/interrupts.html?highlight=mip#machine-interrupt-pending-mip>
* <https://docs.openhwgroup.org/projects/cv32e40s-user-manual/en/latest/exceptions_interrupts.html>

**MMU:**

* <https://riscv.org/wp-content/uploads/2019/08/riscv-privileged-20190608-1.pdf>
* <https://marz.utk.edu/my-courses/cosc562/mmu/>
* <https://chromite.readthedocs.io/en/latest/mmu.html>
* <https://stackoverflow.com/questions/10000693/enabling-mmu-in-linux>
* <https://github.com/PiMaker/rvc>
* <https://blog.stephenmarz.com/2020/11/23/back-that-s-up/#:~:text=Luckily%2C%20RISC-V%20gives%20us%20%E2%80%9Cviews%E2%80%9D%20of%20the%20same,and%20we%20can%20only%20set%20the%20supervisor%20bits>.
* <https://luplab.gitlab.io/rvcodecjs/#q=0x10500073&abi=false&isa=AUTO>

**SBI:**

* <https://github.com/riscv-software-src/opensbi>
* <https://marz.utk.edu/my-courses/cosc562/sbi/> (also really good to understand software interrupts in riscv)
* <https://github.com/slavaim/riscv-notes/blob/master/linux/memory-initialization.md>

**QEMU, Buildroot, Bootloader:**

* <https://github.com/damien-lemoal/buildroot/blob/master/configs/qemu_riscv32_virt_defconfig>
* <https://github.com/regymm/buildroot/tree/master/board/qemu/riscv32-virt>
* <https://superuser.com/questions/415429/how-to-boot-linux-kernel-without-bootloader#:~:text=It%27s%20not%20possible.%20At%20least%2C%20not%20with%20the,more-or-less%20like%20floppy%20booting%20in%20the%20old%20days>.
* <https://www.sifive.com/blog/all-aboard-part-6-booting-a-risc-v-linux-kernel>
* <https://embeddedinn.xyz/articles/tutorial/RISCV-Uncovering-the-Mysteries-of-Linux-Boot-on-RISC-V-QEMU-Machines/>