- Exceptions (pg. 35)
    - Instruction Address Misaligned (0x0)
        - An instruction address misaligned exception is generated on a taken branch of unconditional jump if the target address is not 4-byte aligned.
        - JAL and JALR will generate misaligned instruction fetch exception if the target address is not 4-byte aligned
    - Misaligned Address Load (0x4)/Store (0x6)
        - A load or store to a misaligned address will generate a misaligned address exception
        - A LR/SC to a misaligned address will generate a misaligned address exception
    - Illegal Instruction (0x2)
        - When writing an illegal value to a CSR, the hardware will generate an Illegal Instruction exception
    - Instruction Access Fault (0x1)
    - Breakpoint (0x3)
    - Load Address Fault (0x5)
    - Store Address Fault (0x7)
    - ECALL (System Call) (0x8)
    - Instruction Page Fault (0xC)
    - Load Page Fault (0xD)
    - Store Page Fault (0xF)
    - Machine Timer Interrupt (0x8000_0007)
        - Timer Tick
    - Machine External Interrupt (0x8000_0009)
        - Keyboard Events
        - Other IO/Button Press
    -
- Immediates
    - Except for 5-bit immediates used in CSR instructions, immediates are always sign-extended (B type and J type)
    - The immediate field used to encode branch offsets is in multiples of 2 (implied LSB is 0)
- JALR
    - JALR sets the LSB to 0 when computing the target
- Memory
    - Loads with a destination of x0 must still raise any exceptions and action any other side effects even though the load value is discarded
    - **RISC-V ISA permits misaligned loads and stores; however, our architecture will generate an exception**
- CSRs
    - Instructions
        - CSRRS (Atomic Read and Set Bits in CSR)

        - RD = CSR[*csr*] (0-extend); CSR[*csr*] = CSR[*csr*] | RS1
    - CSRRW (Atomic Read/Write CSR)
        - RD = CSR[*csr*] (0-extend); CSR[*csr*] = RS1
    - CSRRC (Atomic Read and Clear Bits in CSR)
        - RD = CSR[*csr*] (0-extend); CSR[*csr*] = CSR[*csr*] & ~RS1
- https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md
- CSRs
    - **mcycle** RDCYCLE[H]
        - Cycles since boot
        - Encoded (0xC00, 0xC80)
    - ~~**mtime** RDTIME[H]~~
        - ~~Wall Clock Time since epoch~~
        - ~~Encoded (0xC01, 0xC81)~~
    - **minstret** RDINSTRET[H]
        - Instructions retired
        - Encoded (0xC02, 0xC82)
    - Performance Counters
        - Encoding (0xC03 - 0xC1F) High Bits (0xC83, 0xC9F)
        - Instructions Fetched (0xC03)
        - Forward Branch Instructions Retired/Executed (0xC04)
        - Backward Branch Instructions Retired/Executed (0xC04)
        - Forward Branch Instructions Prediction Correct (0xC05)
        - Backward Branch Instructions Prediction Correct (0xC05)
        - Forward Branch Instructions Prediction Incorrect (0xC06)
        - Backward Branch Instructions Prediction Incorrect (0xC06)
        - Cache Instruction Hits (0xC07)
        - Cache Instruction Misses (0xC08)
        - Cache Data Hits (0xC09)
        - Cache Data Misses (0xC0A)
        - LR/SC Success (0xC0B)
        - LR/SC Rejected (0xC0C)
        - Keyboard Interrupts Delivered in U-Mode (0xC0D)
        - Keyboard Interrupts Delivered in M-Mode (0xC0E)
        - Exceptions in U-Mode (0xC0F)
        - Exceptions in M-Mode (0xC10)
        - Timer Interrupts Delivered in U-Mode (0xC11)
        - Timer Interrupts Delivered in M-Mode (0xC12)
        - Forward Jump Instructions Retired
        - Backward Jump Instructions Retired
        - Forward Jump Instructions Predicted Correct
        - Backward Jump Instructions Predicted Correct
        - Forward Jump Instructions Predicted Incorrect
        - Backward Jump Instructions Predicted Incorrect

- Status CSR (**ustatus, mstatus**) (pg. 20)
    - Interrupts Enabled
        - [0] = UIE
            - Hardwire to 0 (no user traps)
        - [3] = MIE
            - 1 => Interrupts Enabled
            - 0 => Interrupts Disabled
        - [4] = UPIE
            - Hardware to 0 (no user traps)
        - Interrupts for lower privilege modes are always disabled, interrupts or higher privilege modes are always enabled
            - No need to worry about interrupts being enabled when running U-Mode code; interrupts should always be enabled for U-Mode
    - Memory Privilege
        - MPRV[17]
            - Set to 0 (translation and protection behave as normal)
        - MXR[19]
            - Hardwire to 0
        - SUM[18]
            - Hardwire to 0
        - TVM[20]
            - Hardwire to 0
        - TW[21]
            - Hardwire to 0
        - TSR[22]
            - Hardwire to 0
    - Context Switch State
        - FS[14:13] hardwired to 0 (no floating point unit)
        - XS[16:15] hardwired to 0 (no extensions)
        - SD[31] hardwired to 0 (no FS or XS)
- **mtvec**
    - Machine Trap Base Address Register (used to call M-Mode traps)
    - All traps are sent to the same handler and resolved dynamically in software
    - MODE[1:0] = 0
        - All exceptions set pc to BASE
    - BASE[31:2] = HANDLER_ADDRESS
        - Must be 4-byte aligned
- Trap/Interrupt Delivery
    - Transition from U-Mode to M-Mode

        - MPIE[7] is set to MIE[3] (should be 1 since no nested traps)
        - MIE[3] is set to 0 (all gates are interrupt gates)
        - MPP[12:11] = 0 (U-Mode)
    - When a trap is delivered
        - **mcause** is set with the trap cause
        - **mepc** is set with the virtual address of the instruction that triggered the trap
        - **mtval** is set appropriately with the error code
        - MPP[12:11] in **mstatus** is set to the privileged mode at the time of the trap (0 if user mode, 3 if kernel mode)
        - MPIE[7] in **mstatus** is set to MIE[3] (should be 1)
        - MIE[3] in **mstatus** is cleared (to disable interrupts)
            - Every interrupt/exception is an interrupt gate
    - MRET returns form kernel interrupt handler
        - MIE[3] is set to MPIE[7] (should be 1 to enable interrupts)
        - To enter user mode:
            - MPP[12:11] should be 0
            - Current Privilege is set to 0
        - To remain in kernel mode
            - MPP[12:11] should be 3
            - Current privilege is set to 3
        - MPP[12:11] is set to 0 (U-Mode)
        - MRET sets the PC value to the value stored in **mepc**
    - An interrupt is delivered if it is enabled in **mip** and **mie** and global interrupts are enabled
        - Interrupts are globally enabled if the privilege level is U-Mode or if the privilege level is M-Mode and MIE[3] in **mstatus** is enabled

- ~~**mip**~~
    - ~~MTIP[7] (Timer Interrupt Pending)~~
        - ~~Read-Only and is cleared when writing to memory-mapped M-Mode timer compare register~~
    - ~~MEIP[11] (External Interrupt Pending)~~
        - ~~Read-Only bit that indicates a M-Mode external interrupt is pending (Keyboard Interrupt)~~
        - ~~MEIP is sset and cleared by platform-specific interrupt controller~~
- ~~**mie**~~
    - ~~MTIE[7] (Timer Interrupt Enabled)~~
        - ~~Should be set to 1 when leaving kernel mode for the first time on boot (this will prevent junk timer ticks during the boot sequence even if interrupts are enabled)~~

            - ~~MEIE[11] (Machine External Interrupt Enabled)~~
                - ~~Enables machine external interrupts when set~~
                - ~~Used to enable keyboard interrupts at boot~~
        - **mscratch**
            - Scratch register for holding context while U-Mode code is running
                - Used like %esp3
        - **mepc**
            - mepc[1:0] are always 0
            - mepc can never hold a PC value that would cause an instruction-address misaligned exception
            - When a trap is taken into M-Mode **mepc** is written with the virtual address of the instruction that encountered the exception
        - **mcause**
            - The code indicating what event caused the trap
            - Interrupt[31]
                - 1 => Cause is an interrupt
                - 0 => Cause is exception/trap
        - **mtval**
            - Breakpoints, Misaligned Address, and Page Fault exceptions load **mtval** with the faulting virtual address (like %cr2)
            - On illegal instruction, **mtval** is loaded with the instruction word
        - **stap**
            - Supervisor Address Translation and Protection
            - MODE[31]
                - 0 => No Virtual Address Translation
                - **1 => Page Based 32-bit address translation**
            - ASID[30:22]
                - The address space identifier
            - PPN[21:0]
                - The physical page number of the root page table (%CR3)
            - Writing to **stap** does not invalidate address translations. We also need to execute SFENCE.VMA after writing **stap**
                - The implementation technically says we should apply the fence before the operation to set the new address space; however, if we implement the fence as a TLB invalidate it will make more sense to call the fence after the new translation is set up and it doesn't seem like there would be any issues
    - Encoding (CSR[11:0])
        - The upper 4 bits are used to specify the permissions for the CSR
        - CSR[11:10] read/write (00, 01, 10) or read-only (11)
        - CSR[9:8] encode lowest privilege level that can access the CSR

        - Implementation will likely involve some sort of magical address translation and routing to a variety of locations (or a highly ported register file)
- ECALL
    - ECALL causes the **epc** register to be set with the address of the ECALL instruction, not the next instruction to run.
        - Handlers need to advance PC appropriately to return
- “M” extension
    - MUL/MULH[S,U]
        - Perform full 32*32 bit multiplication (MUL rd = [31:0] vs. MULH rd = [63:32])
    - DIV[U]/REM[U]
        - Do not generate exceptions on divide by zero
        - x/0 => DIV = -1 && REM = x
        - INT_MIN/-1 => DIV = INT_MIN && REM = 0
- “A” extension
    - **RISC-V ISA specifies SWAP, ADD, XOR, AND, OR, MIN[U], MAX[U] operations which we will not implement in hardware due to microarchitectural challenges.**
        - Instead we will implement the two atomic operations we need (SWAP and ADD) using LR/SC in assembly routines with a while loop around the operations in order to ensure they succeed. Because we are a single core architecture, it is extremely likely that these operations will succeed on the first try since the only way to trigger a failure will be to receive an interrupt between LR and SC
    - LR/SC
        - LR loads a word from address in RS1 and places the value in RD and registers a reservation for the address in RS1
        - SC writes a word in RS2 to the address in RS1 if the reservation on the address in RS1 still is valid
            - SC writes a 0 to RD on success and a 1 to RD on failure
        - Our architectural will require there be a at most a single address in a reservation
            - In order to help ensure safety, any interrupt, branch, load/store, or pipeline flush will automatically invalidate the reservation
- M-Mode (3)
    - Machine mode is entered at hardware reset
    - Assembly test binaries will likely run in M-Mode
    - Kernel will boot and run in M-Mode
        - (We may not actually need to transition to supervisor mode, but it’s probably good practice)
- U-Mode (0)
- S-Mode (1)
    - Not Implemented (Kernel Runs in M-Mode)

- Reset
    - Upon reset the privilege mode is M-Mode
    - MIE and MPRV are set to 0 (disable interrupts)
    - PC is set to an arbitrary reset vector (This should be synced between the boot loader and the compiler to map to _start)
- VRAM
    - A page of physical memory will be allocated that is register backed for VRAM
    - This address should live in the kernel address space and always be direct mapped
    - The memory controller will manage directing accesses to these addresses to the register array instead of going through the cache and DRAM
- SFENCE.VMA (pg. 58)
    - Flushes the TLB for ASID and an virtual address translation
    - If rs1 = x0 && rs2 = x0 :: All translations are flushed
    - If rs1 =x0 && rs2 != x0 :: All translations are flushed for the ASID in rs2
    - If rs1 != x0 && rs2 != x0 :: The translation for virtual address in rs1 is flushed for all ASIDs
    - If rs1 != x0 && rs2 != x0 :: The translation for virtual address in rs1 is flushed for the address space ASID in rs2
- Virtual Memory Translation (pg. 62)
    - VA[31:0] => VPN[1] = VA[31:22], VPN[0] = VA[21:12], PPO = VA[11:0]
    - PA[33:0] => PPN[1] = PA[33:22], PPN[0] = PA[21:12], PPO = VA[11:0] = PA[11:0]
    - The physical page number of the root page table is stored in **satp** CSR
    - Page Table Entry
        - Page Tables consist of 1024 page table entries each is 4 bytes (same as x86)
        - Page Tables must always be page aligned
        - Fields
            - PPN[1:0] = PTE[31:10]
            - RSW[1:0] = PTE[9:8]
                - Reserved for Software
            - D = PTE[7] (Dirty)
                - Not Used (Set to 1)
            - A = PTE[6] (Accessed)
                - Not Used (Set to 1)
            - G = PTE[5] (Global)
            - U = PTE[4] (isUserAssessible)
            - X = PTE[3] (Executable)
                - We'll likely want all pages to be executable for compatibility with 410 kernels
            - W = PTE[2] (Writable)

        - If a page table entry is writable, it must also be marked readable
    - R = PTE[1] (Readable)
    - V = PTE[0] (Valid)
- If RWX are all zero then this page table entry represents a page directory entry and represents a pointer to the next level in the page table hierarchy