Control and Status Registers (CSRs)

CSRs are special-purpose registers used to control the operation of the processor, monitor its state, and manage exceptions and interrupts.

1. CSR Categories

1. **Machine-Level CSRs:**
    - These are mandatory and provide control at the highest privilege level.
    - Examples:
        - `mstatus`: Machine status register.
        - `misa`: Machine ISA and extensions.
        - `mtvec`: Machine trap vector base address.
        - `mcause`: Trap cause.
        - `mepc`: Machine exception program counter.
2. **Supervisor-Level CSRs (Optional):**
    - Used in systems with a supervisor mode (e.g., operating systems).
    - Examples:
        - `sstatus`: Supervisor status register.
        - `stvec`: Supervisor trap vector.
        - `scause`: Supervisor trap cause.
        - `sepc`: Supervisor exception program counter.
3. **User-Level CSRs (Optional):**
    - Typically used for performance monitoring and low-privilege tasks.
    - Examples:
        - `cycle`: Counts the number of clock cycles.
        - `time`: Timer value.
        - `instret`: Number of instructions retired.

2. Accessing CSRs

RISC-V provides specialized instructions to interact with CSRs:

- `csrr`: Read CSR.
- `csrw`: Write CSR.
- `csrrs`: Read and set bits in a CSR.
- `csrrc`: Read and clear bits in a CSR.
- `csrwi`, `csrsi`, `csrci`: Immediate versions of the above instructions.

Example: Reading and Writing a CSR

```
csrr a0, mstatus   # Read mstatus into a0
csrsi mstatus, 1   # Set the least significant bit of mstatus
```

3. Important Machine-Level CSRs

1. `mstatus` **(Machine Status Register):**
   - Controls global machine-level settings (e.g., interrupt enable).
   - Key fields:
     - `MIE`: Machine Interrupt Enable.
     - `MPIE`: Machine Previous Interrupt Enable.
     - `MPRV`: Memory privilege.
2. `mtvec` **(Machine Trap-Vector Base Address):**
   - Determines the base address for trap vectors.
   - Configurable for direct or vectored mode.
3. `mcause` **(Machine Cause Register):**
   - Stores the reason for the last trap.

  - Distinguishes between interrupt and exception types.

4. `mepc` **(Machine Exception Program Counter):**
   - Holds the address of the instruction that caused the exception.

4. Practical Example: Configuring Interrupts

Example: Setting Up a Trap Vector

```
la t0, trap_handler  # Load address of trap handler
csrw mtvec, t0       # Set trap handler address in mtvec
li t1, 0x8           # Example interrupt enable mask
csrs mstatus, t1    # Enable interrupts in mstatus
```

Trap Handler:

```
trap_handler:
    csrr t0, mcause  # Read trap cause
    # Handle interrupt or exception
    mret             # Return from trap
```

5. CSR Usage in Performance Monitoring

CSRs like `cycle` and `instret` are useful for profiling and debugging:

```
csrr a0, cycle      # Read the cycle count
csrr a1, instret    # Read the instruction count
```

Exception Handling in RISC-V

Exceptions are unexpected events that disrupt the normal flow of execution, such as system calls, illegal instructions, or memory access violations.

1. Key Components in Exception Handling

1. **Trap:**
    - The mechanism through which exceptions and interrupts are delivered to the processor.
2. **Trap Vector Base Address (`mtvec`):**
    - Determines where the processor jumps when a trap occurs.
    - Two modes:
        - **Direct Mode:** Single trap handler address.
        - **Vectored Mode:** Base address + cause-specific offset.
3. **Exception Program Counter (`mepc`):**
    - Stores the address of the instruction causing the exception.
4. **Trap Cause (`mcause`):**
    - Identifies the type of exception or interrupt.
    - `mcause[31]`: Indicates whether it's an interrupt.
    - `mcause[30:0]`: Encodes the cause.
5. **Machine Status (`mstatus`):**
    - Controls global interrupt enable and privilege level.

2. Steps in Exception Handling

1. **Trap Entry:**
    - Processor saves the program counter (`mepc`) and current status (`mstatus`).

   - Jumps to the address specified in `mtvec`.

2. **Trap Handling:**

   - The trap handler examines `mcause` to identify the exception type.
   - Performs necessary actions to handle the exception.

3. **Trap Exit:**

   - Restores the saved program counter from `mepc`.
   - Uses the `mret` instruction to return to the interrupted code.

3. Exception Handling Example

Setup Trap Vector:

```
.section .text
.global _start

_start:
    # initialize stack
    la sp, _stack_top    # Load stack pointer with the top of the stack

    la t0, trap_handler   # Load address of trap handler
    csrw mtvec, t0        # Write to mtvec
    li t1, 0x8            # Enable global interrupts
    csrs mstatus, t1

    # Simulate an exception (illegal instruction)
    .word 0xFFFFFFFF       # Illegal instruction to trigger exception
exit:
    j exit                 # Halt

trap_handler:
    csrr t0, mcause        # Read cause of the trap
    li t1, 2               # Cause code for illegal instruction
    beq t0, t1, handle_illegal_instruction
    j other_exception

handle_illegal_instruction:
    # Handle illegal instruction
    mret

other_exception:
    # Handle other exceptions
    mret
```

4. Common Exception Types

Cause	Description
0	Instruction address misaligned
2	Illegal instruction
5	Load address misaligned
7	Store address misaligned

5. Practical Considerations

1. **Ensure Proper `mtvec` Configuration:**
   - Use direct mode for simple handlers.
   - Use vectored mode for performance-critical systems.
2. **Save and Restore Context:**
   - Save registers that may be clobbered in the trap handler.

3. **Debugging Exceptions:**
   - Use `mepc` and `mcause` to identify the offending instruction and exception cause.

System Calls in RISC-V

System calls (via the `ecall` instruction) enable user applications to request services from the operating system or underlying system firmware. These are essential for operations like I/O, file handling, memory management, or privileged instructions that user-mode programs cannot execute directly.

1. What Happens During an `ecall`?

1. **Trigger a Trap:**
   - When a user-mode application executes an `ecall`, it generates a synchronous trap.
   - The processor switches from user mode (U) to supervisor mode (S) or machine mode (M), depending on the system's setup.
2. **Trap Handling:**
   - The trap handler uses the `mcause` (or `scause` in supervisor mode) CSR to determine that the trap was caused by an `ecall`.
3. **System Call Service:**
   - The handler reads the requested system call parameters, typically passed in registers like `a0`-`a7`.
4. **Return to User Mode:**
   - After servicing the request, the processor returns control to the user application at the instruction following the `ecall`.

2. System Call Convention

- **Register Usage:**
  - `a7`: Holds the system call number.
  - `a0`–`a3`: Hold system call arguments.
  - `a0`: Used to return the result.

3. Example: Using `ecall` for Printing a String

Let's implement a system call to print a string.

Assembly Code:

```
.section .text
.global _start

_start:

    # Exit the program (Linux _exit() syscall)
    li a7, 93               # Syscall number for exit
    li a0, 0                # Exit status
    ecall
```

4. Trap Handler Workflow

1. **Detect the Trap:**
   - Check the value in the `mcause` CSR.
   - For an `ecall` from user mode, `mcause` will have the value **8**.
2. **Handle the Request:**
   - Read the system call number from `a7`.
   - Execute the corresponding function (e.g., print string or exit).
3. **Return Control:**
   - Restore registers and use the `mret` instruction to return to user mode.

5. Minimal Trap Handler Example

Here’s how a trap handler might look in machine mode:

Trap Handler in Assembly:

```
trap_handler:
    csrr t0, mcause         # Read mcause
    li t1, 8                # Check if it’s an ecall
    beq t0, t1, handle_ecall
    j other_trap

handle_ecall:
    # Read syscall number from a7
    csrr a0, a7
    li t2, 64                # Syscall number for print
    beq a0, t2, syscall_print
    li t2, 93               # Syscall number for exit
    beq a0, t2, syscall_exit
    j trap_exit

syscall_print:
    # Custom logic for printing (platform-dependent)
    j trap_exit

syscall_exit:
    # Custom logic for exiting
    j trap_exit

other_trap:
    # Handle other traps
    j trap_exit

trap_exit:
    mret
```

6. Practical Notes

1. **Platform Dependency:**
    - System call numbers and functionality vary across operating systems (e.g., Linux RISC-V ABI).
2. **Error Handling:**
    - System calls return error codes in `a0`. Applications should check this value.
3. **Debugging:**
    - Use `mcause` and `mepc` to trace issues during trap handling.

Deep Dive into CSRs

Control and Status Registers are critical for managing exceptions, interrupts, and processor state. They allow privileged software (like an operating system) to control and monitor the processor.

1. CSR Addressing

CSRs are accessed using specific addresses, grouped as follows:

- **Machine-Level (Mandatory):** Address range `0x300–0x3FF`.
- **Supervisor-Level (Optional):** Address range `0x100–0x1FF`.
- **User-Level (Optional):** Address range `0xC00–0xCFF`.

2. Key Machine-Level CSRs for Exception Handling

1. **`mstatus` (Machine Status Register):**
    - Manages global control of the processor, including privilege levels and interrupt enables.
    - Important Fields:
        - `MIE` (Bit 3): Global interrupt enable.
        - `MPIE` (Bit 7): Previous value of `MIE` (saved during traps).

        - `MPP` (Bits 11-12): Previous privilege level.

2. `mtvec` **(Machine Trap-Vector Base Address):**

    - Specifies the address to jump to on a trap.
    - Fields:
        - `BASE` (Bits 2–31): Base address of the trap vector.
        - `MODE` (Bits 0–1): Trap handling mode (Direct or Vectored).

3. `mcause` **(Machine Cause Register):**

    - Encodes the cause of the trap.
    - Fields:
        - `Interrupt` (Bit 31): 1 for interrupts, 0 for exceptions.
        - `Exception Code` (Bits 0–30): Cause of the trap.

4. `mepc` **(Machine Exception Program Counter):**

    - Holds the address of the instruction that caused the exception.

5. `mscratch` **(Machine Scratch Register):**

    - Temporary storage for trap handlers.

3. Accessing and Modifying CSRs

RISC-V provides instructions for interacting with CSRs:

- `csrrw` (CSR Read/Write)
- `csrrs` (CSR Read and Set Bits)
- `csrrc` (CSR Read and Clear Bits)
- Immediate versions: `csrwi`, `csrrsi`, `csrrci`.

Example: Enable Machine Interrupts

```
li t0, 0x8            # Load MIE bitmask
csrs mstatus, t0      # Set MIE in mstatus
```

Example: Configure Trap Vector

```
la t0, trap_handler   # Load trap handler address
csrw mtvec, t0        # Write to mtvec
```

4. Advanced Example: Exception Handling Using CSRs

```
.section .text
.global _start

_start:
    # Initialize stack
    la sp, _stack_top    # Load stack pointer with the top of the stack

main:
    # Set up the trap handler
    la t0, trap_handler   # Load address of trap handler
    csrw mtvec, t0        # Write trap handler base address to mtvec
    li t1, 0x8            # Enable global interrupts
    csrs mstatus, t1      # Set global interrupts enable in mstatus

    # Simulate a memory access exception while accessing 0xFFFFFFFF
    li t0, -1               # Load an invalid memory address
    lw t1, 0(t0)           # Attempt to read from invalid memory (triggers exception)

exit:
    j exit
```

```
trap_handler:
    # Read trap cause from mcause
    csrr t0, mcause
    li t1, 5                # Cause code for load access fault (5)
    beq t0, t1, handle_load_access_fault

    # Handle other exceptions
    j other_exception

handle_load_access_fault:
    # Handle load access fault (e.g., invalid memory access)
    # You could log or recover here
    j trap_exit

other_exception:
    # Handle any other exception
    j trap_exit

trap_exit:
    # Update mepc to skip the faulting instruction
    csrr t0, mepc          # Read mepc
    addi t0, t0, 4         # Increment to skip the faulting instruction
    csrw mepc, t0          # Update mepc
    mret                   # Return from trap to next instruction
```

5. Practical Considerations

1. **Trap Handling Optimization:**
    - Use `mtvec` in vectored mode for better performance in interrupt-heavy systems.
2. **Customizing Privilege Levels:**
    - Use the `MPP` field in `mstatus` to manage privilege transitions.

3. **Debugging Traps:**
    - Inspect `mepc` and `mcause` to trace and resolve exceptions.

Interrupts in RISC-V

Interrupts are hardware-triggered events that temporarily pause the execution of the current program to execute a specific routine, called an interrupt service routine (ISR) or trap handler.

1. Types of Interrupts

RISC-V interrupts are categorized into:

1. **Machine-Level Interrupts:**
    - Always available and handled at the machine privilege level.
    - Examples:
        - Timer interrupt.
        - External interrupt.
        - Software interrupt.

2. **Supervisor-Level Interrupts (Optional):**
    - Handled at the supervisor privilege level, typically by an OS.

3. **User-Level Interrupts (Optional):**
    - Handled in user mode for specific applications.

2. CSR Registers for Interrupts

Key registers involved in interrupt management include:

1. **`mstatus`:**
    - `MIE` (Bit 3): Global machine interrupt enable.

- `MPIE` (Bit 7): Stores previous interrupt enable status.

2. **`mie`:**
    - Machine interrupt-enable register.
    - Enables specific interrupts (e.g., timer, external, software).
3. **`mip`:**
    - Machine interrupt-pending register.
    - Indicates pending interrupts.
4. **`mtvec`:**
    - Specifies the base address of the trap vector.
5. **`mcause`:**
    - Identifies the interrupt type and cause.

3. Steps for Handling Interrupts

1. **Interrupt Triggered:**
    - The processor halts the current execution and saves the program counter (`mepc`).
2. **Trap Handler Execution:**
    - Jumps to the address specified by `mtvec`.
    - The handler reads `mcause` to identify the interrupt type.
3. **Interrupt Servicing:**
    - The handler performs the required action for the interrupt.
4. **Return from Interrupt:**

- Restores the saved state and resumes execution using the `mret` instruction.

4. Configuring Interrupts

Example: Enabling Timer Interrupt

```
.section .text
.global _start

_start:
    # Initialize stack
    la sp, _stack_top    # Load stack pointer with the top of the stack

main:
    la t0, trap_handler    # Load trap handler address
    csrw mtvec, t0         # Set trap vector base address
    li t1, 0x80            # Enable timer interrupt
    csrw mie, t1           # Write to mie to enable specific interrupts
    csrsi mstatus, 0x8     # Enable global interrupts (MIE)

    # need to set timcmp register. when time > timcmp, interrupt will raise
    # if time is not supported on processor, interrupt will not be generated.
    # Simulated workload
    loop:
        nop                # Do nothing
        j loop             # Infinite loop

trap_handler:
    csrr t0, mcause        # Read mcause to identify the interrupt
    li t1, 7              # Timer interrupt cause
    beq t0, t1, handle_timer_interrupt
    j default_handler

handle_timer_interrupt:
    # Handle timer interrupt
    mret                       # Return from interrupt

default_handler:
    # Handle other interrupts
    mret                       # Return from interrupt
```

5. Vectored vs. Direct Mode

- **Direct Mode:** All traps go to a single handler address.
- **Vectored Mode:** Uses an offset to call specific handlers for different interrupts.

Example: Vectored Mode Setup

```
li t0, 0x1                 # Set mtvec mode to vectored
csrw mtvec, t0             # Write to mtvec
```

6. Key Registers Involved

CSR	Role
`mstatus`	Stores interrupt enable state and privilege level.
`mepc`	Saves the address of the interrupted instruction.
`mcause`	Indicates the reason for the interrupt or exception.
`mtvec`	Specifies the base address for the trap vector.
`mie`	Enables specific interrupts at the machine level.
`mip`	Indicates pending interrupts.

6. Practical Considerations

1. **Interrupt Prioritization:**
   - RISC-V does not enforce interrupt prioritization but allows custom implementations using software or additional hardware.
2. **Latency Minimization:**
   - Use vectored mode for high-performance interrupt handling.
3. **Debugging:**
   - Use `mip` and `mcause` to identify and resolve interrupt-related issues.
4. **Efficient Context Switching:**
   - Minimize ISR overhead by saving/restoring only necessary registers.
5. **Nested Interrupts:**
   - For systems requiring nested interrupts, additional software logic is needed to manage interrupt priorities.

C-DAC VEGA Processors

The **VEGA Processors**, developed by the Centre for Development of Advanced Computing (C-DAC) in India, represent a significant milestone in RISC-V adoption. These processors are designed for a range of applications, from IoT and embedded systems to high-performance computing.

1. Overview of C-DAC VEGA Processors

The VEGA processors are built on the RISC-V instruction set architecture and offer a modular, scalable design. Key features include:

- **Compliance with RISC-V Standards:**
  - Fully compliant with the RISC-V ISA, supporting both base and extended instruction sets.
- **Application-Specific Designs:**
  - Designed for domains like IoT, AI, robotics, automotive, and industrial automation.
- **Open-Source Philosophy:**
  - Promotes openness and adaptability, leveraging the advantages of the RISC-V ecosystem.

2. VEGA Processor Family

The VEGA processor family includes several variants tailored to different use cases:

1. **VEGA ET1031:**
   - Ultra-low-power processor core.
   - Targeted at IoT and embedded systems.
   - Implements the RISC-V RV32IMAC ISA.
2. **VEGA AS1061:**
   - High-performance, low-power processor.
   - Suitable for AI and machine learning workloads.
   - Supports the RV64IMAFDC ISA with extensions for floating-point and atomic operations.
3. **VEGA AS1161:**
   - Multi-core architecture for higher performance.
   - Ideal for compute-intensive applications like robotics and automotive systems.
4. **VEGA CL1011:**
   - Customizable processor core.
   - Supports integration into specific application domains with a focus on configurability.

3. Key Features

1. **Scalability:**
   - Ranges from single-core to multi-core designs.
   - Scalable performance for various applications.
2. **Energy Efficiency:**
   - Optimized for low power consumption.

  - Suitable for battery-powered devices.

3. **Customizability:**
   - Offers configurability to include/exclude specific features based on application requirements.
4. **Integration with Peripherals:**
   - Includes built-in interfaces for UART, SPI, I2C, and other common peripherals.
5. **Security Features:**
   - Hardware-level security for critical applications.

4. Example Use Cases

1. **IoT Devices:**
   - VEGA ET1031 is ideal for IoT edge devices, enabling low-power data processing and transmission.
2. **AI and ML Applications:**
   - VEGA AS1061 excels in running AI inference tasks efficiently.
3. **Automotive Systems:**
   - Multi-core VEGA processors can handle the complex computations required in advanced driver-assistance systems (ADAS).
4. **Robotics:**
   - High-performance VEGA cores can manage real-time control systems in robotics.

5. Development Ecosystem

1. **Toolchain Support:**
   - Compatible with GCC and LLVM-based toolchains.

- Debugging and profiling tools tailored for VEGA processors.

2. **SDKs and Libraries:**
   - Offers software development kits for rapid application development.

3. **Simulation Platforms:**
   - Provides simulators for early-stage software development and validation.

6. VEGA and India's Strategic Vision

The VEGA processors are part of India's push toward self-reliance in semiconductor design. They aim to reduce dependency on foreign technologies while fostering local innovation in high-tech domains.

Mode Transitions in RISC-V (M, S, and U)

RISC-V processors support multiple privilege modes to ensure secure and efficient operation. These modes define the level of access a program has to system resources, protecting critical operations from untrusted code.

1. Privilege Modes in RISC-V

1. **Machine Mode (M):**
   - Highest privilege level.
   - Used for firmware, bootloader, or low-level resource management.
   - Has unrestricted access to the entire system, including memory, peripherals, and CSRs.

2. **Supervisor Mode (S):**
   - Mid-level privilege.
   - Typically used by operating systems.
   - Access to user applications and system resources but restricted from certain hardware functions.

3. **User Mode (U):**

  - Lowest privilege level.
  - For running user applications.
  - Cannot directly access hardware resources or privileged instructions.

2. Why Transition Between Modes?

- To ensure system security by isolating untrusted code (e.g., user applications).
- To handle exceptions and system calls (e.g., `ecall` transitions from User to Supervisor or Machine mode).
- To implement privilege separation between the operating system and firmware.

3. Mode Transition Mechanisms

1. **From Lower to Higher Privilege (e.g., U → S):**
   - Triggered by **traps** such as:
     - System calls (`ecall`).
     - Exceptions (e.g., illegal instructions, memory access violations).
   - The current program counter (PC) and mode are saved in a CSR like `mepc` or `sepc`.
2. **From Higher to Lower Privilege (e.g., S → U):**
   - Controlled by the operating system or firmware.
   - Use instructions like `mret` or `sret` to return to the previous mode.

4. CSRs Related to Mode Transitions

1. **Status Registers (`mstatus`, `sstatus`):**
   - Indicate the current privilege level and enable/disable interrupts.
2. **Exception PC Registers (`mepc`, `sepc`):**
   - Store the return address when transitioning due to a trap.
3. **Cause Registers (`mcause`, `scause`):**

   - Record the reason for the trap.

4. **Trap Vector Registers (`mtvec`, `stvec`):**

   - Specify the address of the trap handler.

5. Example: Transition Workflow

User Mode to Supervisor Mode (`ecall`):

1. **User Code:**

```
li a7, 1            # Syscall number
ecall               # Trigger trap
```

2. **Trap Handling in Supervisor Mode:**

```
trap_handler:
    csrr t0, scause # Determine trap cause
    li t1, 9        # Check for ecall
    beq t0, t1, handle_syscall

    # Handle other traps
    j other_trap

handle_syscall:
    # Read syscall number from a7 and perform action
    j return_to_user

return_to_user:
    sret             # Return to user mode
```

Supervisor Mode to User Mode (`sret`):

1. **OS Code:**
    - Prepares the user program's state (e.g., PC and registers).
    - Loads the address of the user program into `sepc`.
    - Executes `sret` to transition to User mode.
2. **Instruction:**

```
csrw sepc, user_pc    # Set user program counter
sret                  # Switch to User mode
```

6. Example: Switching Between Modes in Practice

Full Workflow (Machine to User):

1. **Bootloader in Machine Mode:**
    - Initializes the system.
    - Loads the OS into memory.
    - Sets the trap vector and switches to Supervisor mode.
2. **OS in Supervisor Mode:**
    - Sets up user program execution.
    - Handles system calls and exceptions.
3. **User Program in User Mode:**
    - Executes application logic.
    - Requests system resources via system calls.

7. Debugging Mode Transitions

1. **Check CSRs:**
    - Use `mcause`, `mepc`, and `mstatus` to trace transitions.
2. **Trap Handlers:**
    - Verify trap vector (`mtvec` or `stvec`) and handler implementation.
3. **Common Issues:**
    - Incorrect privilege level settings in `mstatus` or `sstatus`.
    - Improper initialization of trap vector or return PC.

Build reference

- https://mth.st/blog/riscv-qemu/