Vectored Interrupt Handling

Integration with Shakti SDK

Author: Prajwal Prakash

Overview:

The aim of this project is to add vectored interrupt handling functionality within the Shakti-SDK. This document explains the project under the following headings:

1. **Vectored Interrupt Handler Requirement**: The need for a vectored interrupt handling mechanism in the SDK (broadly, an advantage in speed of processing an interrupt)
2. **Proposed Improvements to Existing Trap Handler Signature:** Some proposed changes, with justification, to the existing interrupt service routine function signature (independent of the vectored interrupt option)
3. **Important Considerations and Challenges:** The factors that went in while designing and coding the vectored interrupt handler – the most important being easy integration into the SDK, and synchronous exception handling in the vectored mode.
4. **Solution Description:** A brief explanation of the new files added and how to use the added vectored interrupt functionality.

Vectored Interrupt Handler Requirement:

The need for a vectored interrupt handler arose specifically with regards to the PLIC controller. This controller handles all machine mode external interrupts, and the control flow from when the PLIC sends an interrupt signal to when the ISR is called is rather long. The two approaches are contrasted below.

Existing Implementation:

First, at hardware level:

1. The PLIC sends the interrupt signal
2. Hardware sets the PC to the address stored in *mtvec* register – *trap_entry* - and stores the predefined exception number in the *mcause* register (as given in the RISC -V spec sheet)

Then at software level:

3. Context save occurs – all register values are pushed to stack.
4. The software calls a *handle_trap* function
5. This function takes the Exception Program Counter (instruction that caused the exception) and *mcause* value as arguments.
6. It examines the *mcause* value and determines that it is an external machine mode interrupt
7. It then calls the function in the interrupt table
8. This function transfers control to the *mach_plic handler*
9. After the ISR completes execution, it returns control to the *handle_trap* function, which again returns control to the *trap_entry* function
10. The context is restored through software and PC is set back to the address in the *mepc* control register.

Vectored Interrupt Model:

First at hardware level,

1. The PLIC sends the interrupt signal
2. Hardware sets the PC to the address in the relevant ISR, which is the *machine_external_interrupt* handler (at the *mtvec* + *mcause * 4* location)

Then at software level,

3. Context save occurs
4. Control is transferred to the *mach_plic_handler*
5. After the ISR completes execution, it returns control to the machine external interrupt handler
6. The context is restored through software and PC is set back to the address in the *mepc* control register.

This clearly allows the interrupt service routine to begin execution in significantly fewer clock cycles.

Proposed Improvements to Existing Trap Handler Signature:

The existing solution combines both synchronous exception handling and asynchronous exception handling (traps and interrupts, respectively) into one trap handler module. Please refer to the RISC-V spec for the CSR descriptions.

The ISR for interrupts is expected to take the *mcause* and *epc* value as an argument. Even though traps and interrupts are fundamentally different, their handling and ISR structure in the implementation is exactly the same.

1. For traps, having the *mcause* value and *epc* make sense if the developer implements the trap handling with a single trap handler (like the *default_handler* implemented in the SDK) or combines certain traps with the other, since these tend to be errors that are undesirable in practical use cases. The *epc* argument is especially important.

2. For interrupts, the *mcause* register contains no valuable information once the relevant interrupt service routine is called. With the vectored interrupt model, the register is never needed at all by software. Moreover, for interrupts, the *mepc* value has no meaning since they are external in nature and not caused by the instruction being executed.

The change proposed to the existing structure is to have a separate function signature for an ISR for an interrupt versus that of a trap – the interrupt service routine for the asynchronous exception need not take the arguments of *mcause* and *epc*. (this is more obvious if one looks at the *mach_plic_handler* function, where both these arguments are unused).

These changes will have to occur in the traps.h header file, the traps.c file and any existing ISR implementation will have to remove the two arguments (which should not be a problem since these would be unused anyway).

Important Considerations and Challenges:

The following factors were considered while designing the vectored interrupt handler:

1. **Handling synchronous exceptions:** When vectored interrupts are enabled, what happens to traps? RISC-V spec sheet says that all synchronous exceptions have the hardware set PC value to the *mtvec* address. This corresponds to the same address as a user software interrupt. However, in practice, a user software interrupt will always be delegated to be handled in a lower privilege. So, the <u>existing trap handler module can be effectively reused</u> by setting up the vectored interrupt to go to the *trap_entry* already defined on a user software interrupt.

2. **Selecting method of interrupt handling:** The type of trap handling required will vary based on the use case. Trap handling mode is decided when the value of *mtvec* is set. In the SDK, this happens in the *init* function. Here, by default, the *mtvec* is loaded with *trap_entry.* The enabled vectored function call will load *mtvec* with the *vectored_trap_entry*

3. **Reducing the size of the code:** One disadvantage of vectored interrupt handling is that each type of interrupt handler needs its own context save and restore procedure. To prevent repetition of code, the context switch procedures are implemented as assembly functions. The key is to save the value in the *ra* register before calling the save, and restore the value in the *ra* register after calling the restore.

4. **Returning control and handling multiple interrupts:** A challenge of interrupt handling is taking care of an interrupt in the middle of handling another interrupt. This problem is mitigated by the fact that on encountering an interrupt, the *MIE* bit is set to 0. So, until the interrupt is handled completely another interrupt cannot occur.

Solution Description:

New files:

1. **vectored_traps.S** - contains the main vectored trap entry function. Takes care of context save and restore procedures for all interrupts, by mimicking the structure of the *trap_entry* function - context save, transfer control to ISR, return, context restore, *mret.*
2. **vectored.c** - contains wrapper functions to call relevant ISRs. Also contains the *enabled_vectored function.* To use this with the SDK, one has to include vectored.h and interrupts will automatically be handled in vector mode from after enable_vectored is called, and just like in the current implementation user only has to set the mcause interrupt table with the relevant ISR. This means the user can enable vectored interrupts by just calling the *enable_vectored* function.
3. **vectored.h** - function definitions for functions in vectored.c
4. **vectored_test** - application under clint_applns, to test timer interrupts in vectored mode by creating a clock of regular intervals.
5. **asm_vectored -** application under uart_applns, which is an all in one test of vectored mode interrupt handling.