EECS 388 Lab #6

Timer Interrupt Handling

In this lab, you will write a timer interrupt handler.

Part 0: Setup the project

Sign in to the Canvas account and go to the EEC 388 Embedded Systems section. Inside the "Files" section, go to the "Source Codes" folder. Click on the **lab06.tar.gz** file to download the source code for lab6. Extract the file inside a folder of your PC.

After that, add this folder to the VS Code workspace (like the previous labs).

Part 1: Understanding Timer and Interrupt Handling in RISC-V

Interrupts are a mechanism for CPUs to be able to handle new tasks or information while it might be already doing some other processing task. You can read more about interrupts in general here.

The interrupt feature can be used to create a “timer”. On a high level, the CPU on the HiFive board (which is a RISC-V CPU) has a “counter”, which is basically a register that counts up by 1 on every CPU cycle. Thus, its value represents how many cycles have elapsed. It can be used as a proxy for the time elapsed if you know the frequency of the CPU (the number of cycles per second). The name of this register is **mtime**.

This counter can be used in combination with another special register, called **mtimecmp**, to implement a “timer” of sorts. This timer’s function is similar to that of a regular kitchen timer, in the sense that it generates a signal (an interrupt to be specific) after a certain amount of time has passed. The setup works by setting a specific value into the mtimecmp register and when the mtime register counts up and reaches the same value as the mtimecmp register, the HiFive board generates an interrupt to the CPU. By writing appropriate values into the mtimecmp register, you can control how long it takes for the counter to reach that value and generate an interrupt, effectively creating a timer.

To do this lab, you need to understand a bit of background on how interrupts work in RISC-V. Open the CPU datasheet (docs/FE310-G002.pdf) and read Chapter 8. Figure 4 shows how various interrupts are connected to the CPU core (E31). What you will use in this lab is ‘Machine Timer Interrupt’ as shown in the figure below.

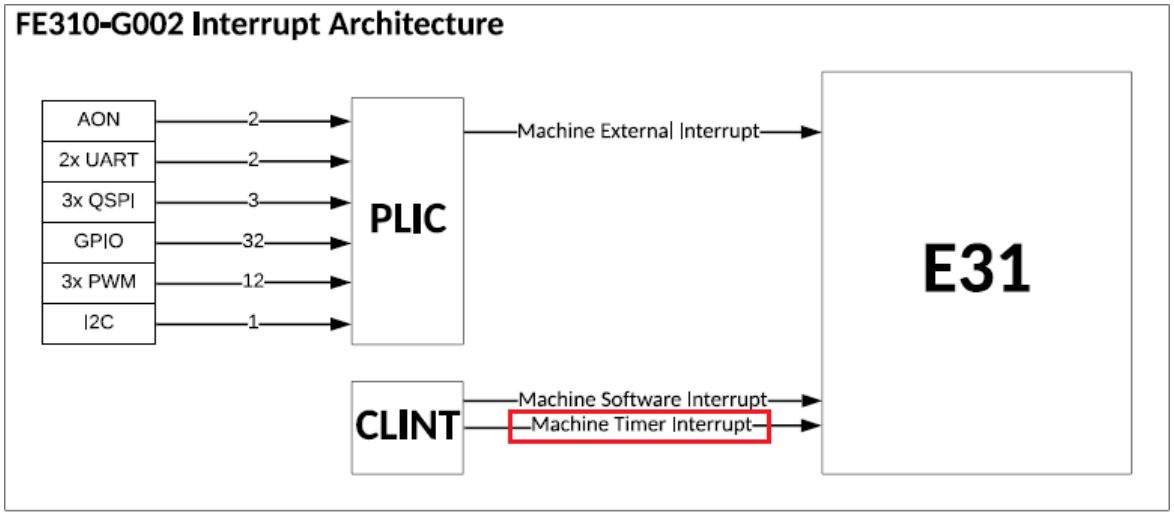

Figure 4: FE310-G002 Interrupt Architecture Block Diagram.

'Machine Timer Interrupt' is a core local timer interrupt, which is generated by using the two architecturally defined timer registers: **mtime** and **mtimecmp**.

In RISC-V, each core is required to provide a 64-bit real-time counter, which is monotonically increasing at a constant speed and is exposed as the memory mapped register, **mtime**. In the E31 core of the HiFive1 board, CLINT (core local interruptor) hardware block (Chapter 9) is responsible to provide the real-time counter and is mapped at the following address:

0x200bff8	8B	RW	mtime	Timer Register

As mentioned before, on the HiFive1 platform, the mtime register contains the number of cycles counted from the system's real-time clock, which is running at 32.678kHz (32768Hz).

The following get_cycles function, which is provided in eecs388_lib.c, returns the value of the mtime register (and thereby, the number of cycles elapsed so far).

```
uint64_t get_cycles(void)
{
 return *(volatile uint64_t *)(CLINT_CTRL_ADDR + CLINT_MTIME);
}
```

Test Yourself: How long (in years) will it take to overflow the 64bit timer? Assume the counter begins from zero at reset, and is incremented at the speed of 32678Hz.

In order to generate a timer interrupt, one should update **mtimecmp** register, which is mapped in the following address on the HiFive 1 platform.

0x2004000	8B	RW	mtimecmp for hart 0	MTIMECMP Registers

A timer interrupt (if enabled) is generated whenever **mtime** is *greater than or equal to* the value in the **mtimecmp** register. Therefore, to generate a timer interrupt after X cycles, one can update mtimecmp register as follows: **mtimecmp = mtime + X.**

You can use the following set_cycle function, which is provided in eecs388_lib.c, to update the mtimecmp register. If you recall from before, when the mtime register counts up to the new value of mtimecmp, it will generate an interrupt.

```
void set_cycles(uint64_t cycle)
{
 *(volatile uint64_t *)(CLINT_CTRL_ADDR + CLINT_MTIMECMP) = cycle;
}
```

Note that the timer interrupt can be enabled or disabled by updating an architecturally defined control register, **mie**, shown below. To enable the timer interrupt, the MTIE field (i.e. bit 7) of the mie register should be set (to 1), and it should be cleared (set to 0) to disable the timer interrupt.

Machine Interrupt Enable Register			
CSR	`mie`		
Bits	**Field Name**	**Attr.**	**Description**
[2:0]	Reserved	WPRI	
3	MSIE	RW	Machine Software Interrupt Enable
[6:4]	Reserved	WPRI	
7	MTIE	RW	Machine Timer Interrupt Enable
[10:8]	Reserved	WPRI	
11	MEIE	RW	Machine External Interrupt Enable
[31:12]	Reserved	WPRI	

Table 20: `mie` Register

Machine Status Register			
CSR	`mstatus`		
Bits	**Field Name**	**Attr.**	**Description**
[2:0]	Reserved	WPRI	
3	MIE	RW	Machine Interrupt Enable
[6:4]	Reserved	WPRI	
7	MPIE	RW	Machine Previous Interrupt Enable
[10:8]	Reserved	WPRI	
[12:11]	MPP	RW	Machine Previous Privilege Mode

Table 17: FE310-G002 `mstatus` Register (partial)

In addition to the mie register, there is another control register called **mstatus**, which can enable or disable *all* interrupts at once (there are other kinds of interrupts besides timer interrupts). To enable the timer interrupt, both mie and mstatus registers should be updated as follows: mie.MTIE = 1 (i.e. the MTIE field or bit 7 of mie register should be 1) and mstatus.MIE = 1 (i.e. the MIE field or bit 3 of mie register should be 1).

Note that unlike the timer registers, mie and mstatus registers are not memory-mapped and can only be accessed by executing special instructions: csrr and csrw for read and write, respectively. You can use the following macros, which are provided in the eecs388_lib.h

```
#define read_csr(reg) ({ unsigned long _tmp; \
 asm volatile ("csrr %0, " #reg : "=r"(_tmp)); \
 _tmp; })
#define write_csr(reg, val) ({ \
 asm volatile ("csrw " #reg ", %0" :: "rK"(val)); })
```

For example, read_csr(mie) will return the value of the mie register, while write_csr(mie, <32bit value>) will update the mie register.

Once an interrupt is generated, the CPU *traps* (i.e. moves or looks) to the address stored in mtvec register, which has the trap handler. The trap handler is responsible for identifying the cause of the interrupt and to jump to the appropriate interrupt service routine (interrupt handler). The cause of the interrupt can be read via the mcause register, shown below, which is also accessible only via csrr or cswr instructions (using the macros above, for

example, read_csr(mcause)). For the timer interrupt, the exception code in the mcause register is 7.

Machine Cause Register			
CSR	`mcause`		
Bits	**Field Name**	**Attr.**	**Description**
[9:0]	Exception Code	WLRL	A code identifying the last exception.
[30:10]	Reserved	WLRL	
31	Interrupt	WARL	1 if the trap was caused by an interrupt; 0 otherwise.

Table 22: `mcause` Register

Interrupt Exception Codes		
Interrupt	**Exception Code**	**Description**
1	0–2	Reserved
1	3	Machine software interrupt
1	4–6	Reserved
1	7	Machine timer interrupt
1	8–10	Reserved
1	11	Machine external interrupt
1	≥ 12	Reserved
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9–10	Reserved
0	11	Environment call from M-mode
0	≥ 12	Reserved

Table 23: `mcause` Exception Codes

Part 2: Write your periodic timer interrupt handler – 100 points

The goal of this lab is to blink an LED at a constant 100ms interval. You need to program the timer handler and interrupt enable/disable functions to complete the lab.

First, review the main program '**eecs388_interrupt.c**' and the two library files '**eecs388_lib.h**' and '**eecs388_lib.c'**. Currently, the functions **timer_handler**, **enable_interrupt**, and **disable_interrupt** are partially filled or empty in the **eecs388_interrupt.c** file. Your task for this lab is to complete the functions.

The timer_handler function essentially has to set the mtimecmpr register to the correct value so that an interrupt is generated after 100 ms. The enable_interrupt and disable_interrupt functions only need to modify the mstatus register so that the correct bit is set/reset to turn interrupts on or off altogether respectively.

Submission

Go to the "Assignments" section of the Canvas and submit your modified eecs388_interrupt.c file to the Lab06 submission link. Also make sure that you have shown your completed work to your respective GTA for demo.

Screenshots/PDF/Text files will NOT be allowed/graded as submission. You need to submit the modified eecs388_interrupt.c file only.