Chapter 1
Memory-mapped Input/Output

CHAPTER GOALS

Have you ever wondered how a Central Processing Unit communicate with I/O devices? In modern computer systems, this is done using so-called memory-mapped I/O. In this chapter, we will cover the theory and practice of memory mapping and address decoding. A memory-mapped I/O device is a computer hardware component that uses a portion of the system's memory address space and is accessible by load and store instructions, while address decoding determines which device or peripheral in a computer system should respond to a particular load or store instruction.
Upon completion of this chapter, you will be able to:

- Understand and explain memory mapping.
- Understand and explain address decoding.
- Able to program a simple memory-mapped general-purpose IO device.

1.1 Introduction

Recall that the only way for modern processors (e.g. RISC-V) to access data (read or write) is by using memory load (L) and store (S) instructions. These instructions are a fundamental part of an instruction set architecture (ISA) and allow the processor to interact with various types of memory. An important consequence of this principle is that if we want the CPU to read or write data from input/output (I/O) devices, all I/O devices should be visible to the CPU as a set of memory words. We also say that I/O devices should be memory-mapped. A memory-mapped I/O (MMIO) device is a computer hardware component that uses a portion of the system's memory address space for data transfer and control. This approach simplifies device interaction for software developers and is commonly used in modern computer systems.

In particular, these MMIO devices incorporate a small memory (actually, the memory size depends on the device type and its functionality, but in general, this memory has only a few memory words). Each memory word within this on-device memory is assigned a memory address within the system's memory map and, consequently, is accessible through load and store instructions. Besides, each word within this on-device memory has a dedicated meaning (for example, it can be used by the CPU to monitor the device status, to set some features of the device or to read/write data). Because these on-chip memory words have distinct meanings, each memory location is called **a register**. In other words, these registers control various aspects of the device's operation, such as configuration settings, data transfer, and status checks.

In order to assign a unique memory address to each I/O device and its registers, computer systems rely on **address decoding**. Address decoding is crucial in computer architecture and design, particularly in systems that use memory-mapped I/O, as it determines which device or peripheral in a computer system should respond to a particular load or store instruction provided by the CPU. To determine which device should respond to a specific address provided within a load or store instruction, address decoding logic involves a combination of simple digital logic gates, such as AND gates, OR gates, and NOT gates, or even the usage of decoders. When the CPU wants to read from or write to a specific I/O device register or memory location, it places the desired address on the address bus. For example, RISC-V would place this address on the address bus in the fourth pipeline stage (MEM stage) after the memory address has been calculated from the base address and offset within its third pipeline stage (EXE stage). Besides the address, the CPU would also activate the **read-write (R/W signal)**, which tells the addressed device whether the CPU would like to read from or write to. The address decoding logic continuously monitors the address bus. It compares the incoming address on the bus to predefined address ranges for each device. When it detects a match between the incoming address and one of the assigned address ranges, it generates a so-called **chip-select (CS) signal** for that device. When the chip-select signal for a specific device becomes active (typically pulled low), that device knows it should respond to the CPU's request. It **enables** its data bus interface so that data can be read from or written to the device according to the R/W signal. Once the correct device is selected, data can be transferred between the CPU and the selected device through the data bus. The CPU reads from or writes to the device's registers or memory locations based on the operation it wants to perform. When the CPU puts another address onto the address bus, the address decoding logic deactivates the select signal for the device, allowing other devices to respond to subsequent address requests. In the following subsection, I will explain this important concept in detail using a simple example.

1.2 A memory-mapped register

Suppose we would like to connect a single 32-bit register to a 32-bit CPU (e.g. RISC-V). Also, suppose that the register has chip-enable (CE), output-enable (OE) and chip-select (CS) signals besides the standard data input, data output and clock signals. Such a register is presented in Figure 1.1.

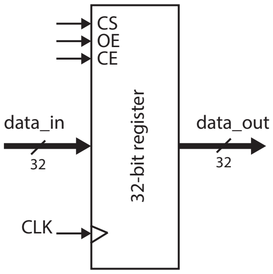

Fig. 1.1: A 32-bit register with the CS, OE and CE signals.

A register with chip-enable, output-enable and chip-select signals is a common component in digital systems, especially those that involve memory-mapped I/O. These signals control the register's behaviour, specifying when it should be enabled or disabled for data read and write operations. Here's how such a register typically works:

1. The output-enable signal (OE) connects or disconnects the output data signal to/from the data bus. When the OE signal is active (high), the register output becomes connected to the data bus. Data stored within the registers appears on the data bus and is accessible for reading. When the OE signal is inactive (low), the data output is disconnected from the data bus, and other components in the computer system can use the data bus.
2. The chip-enable signal (CE) enables the clock signal connected to the register. Hence, when the CE signal is active, data from the data bus will be stored in the register on the rising edge of the clock signal.
3. The chip select signal (CS) is used to activate or deactivate the register. When the CS signal is active, the register is selected and becomes available for read and write operations. When the CS signal is inactive, the register is deselected, and any access to it is disabled.

Using enable and chip select signals provides fine-grained control over register access. It allows you to isolate specific registers or components in a digital system, preventing unintended or erroneous data transfers. These signals are particularly useful in memory-mapped I/O scenarios, where multiple registers or devices share the same address bus.

This control mechanism is essential in digital systems to prevent unintended data transfers and efficiently manage communication with various components. A register with CE, OE and CS signals reads data from or outputs data to the data bus only when it is addressed (selected), and the proper combination of the WE and OE signals is present. But how do we know when to activate these signals? Well, the OE and WE signals depend solely on the R/W signal from the CPU. When the CPU executes a load instruction (reads from memory), the R/W signal is high, and we should activate the OE signal and deactivate the CE signal. On the contrary, when the CPU executes a store instruction, the R/W signal is deactivated, and we should activate the CE signal and deactivate the OE signal. This simple logic is implemented in Figure 1.2.

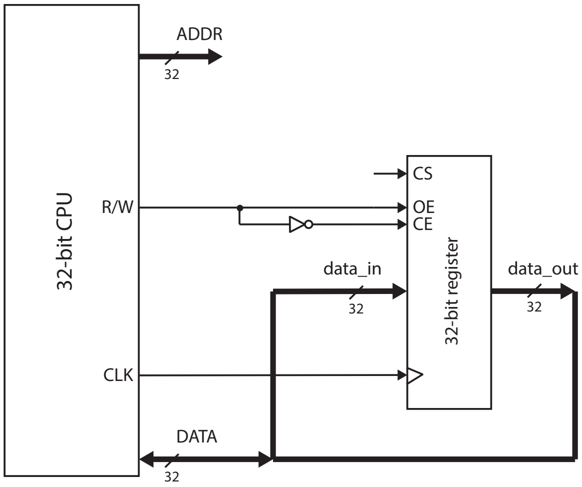

Fig. 1.2: A 32-bit register connected to the data bus and to the CPU's R/W signal.

But how about the CS signal? Well, this signal should be active only when the register is selected. But wait, what does this mean? Who selects the register? The register is selected when the CPU addresses it. Hence, the CS signal depends solely on the content on the address bus. Previously, we said that each memory-mapped register is assigned its own unique address from the CPU address space. The CPU address space is the set of all possible addresses that the CPU can generate. For a 32-bit CPU, the address is 32-bit long, and the CPU can issue any address from 0x00000000 to 0xFFFFFFFF.

Suppose that we would like to connect a register at address 0x80000000. Now, we can use a 32-input AND logical gate to compare the address lines with the desired address. For our example, we should create a logic expression that activates the chip select signal when the address matches 0x800000000. Figure 1.3 shows the solution. This AND gate activates the CS signal when all the specified address lines

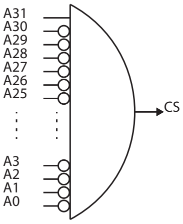

Fig. 1.3: A 32-input AND gate used to decode address 0x80000000.

match their respective logic levels (high or low) as in the assigned address. This process is called **address decoding**. Figure 1.4 presents the final digital circuitry

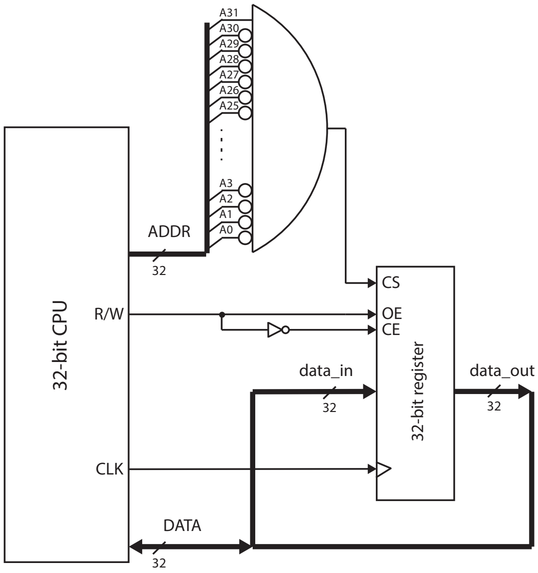

Fig. 1.4: A 32-input AND gate used to decode address 0x80000000.

used to connect the register to the CPU. Now, the register is memory-mapped into the CPU address space, and it is accessible for reading and writing at the address 0x800000000.

We see that address decoding involves constantly comparing the addresses on the address bus and generating the CS signal when the address on the address bus matches the address assigned to an I/O device.

1.3 Two memory mapped registers

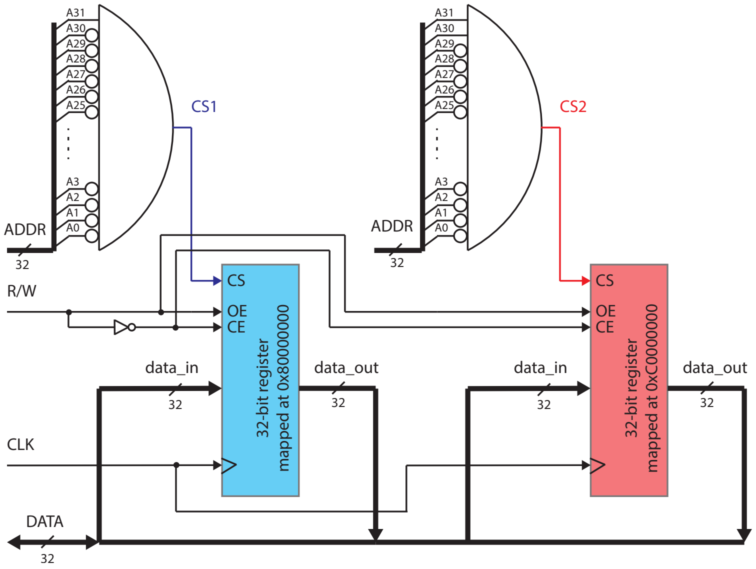

Fig. 1.5: Two memory-mapped registers at addresses 0x80000000 and 0xC0000000, respectively.

Now that we understand how a register can be memory-mapped into the CPU address space and which signals are used in this process, we can try to memory-map and connect two registers to a CPU. Suppose we connect one register at address 0x80000000 and the other to address 0xC0000000. Actually, this task is straightforward and is depicted in Figure 1.5. We should use two AND gates to decode two addresses. One AND gate decodes address 0x80000000 and selects the first register, while the second AND gate decodes address 0xC0000000 and selects the second register. Both registers can share the address, CE and OE signals because address decoding logic ensures that registers cannot be selected (active) simultaneously. Hence, we can see that address decoding isolates two or more registers or components in a computer system and is a crucial concept in memory-mapped I/O systems, where multiple registers or devices share the same address bus.

Although the presented address decoding with AND gates seems very simple, it has a serious drawback. In CMOS technology used to implement basic logic gates, we can usually implement only 2- or 3-input logic gates. In our example, we used 32-input AND gates that do not exist in the real world. Hence, in the real world, we would use tens of 2-input AND gates to implement the address decoding for only one address. In real-world computer systems with tens of I/O devices and hundreds of memory-mapped registers, this solution would be very inefficient in terms of the number of logic gates used. Hence, we should use a different solution to decode the addresses and select the I/O devices and their registers.

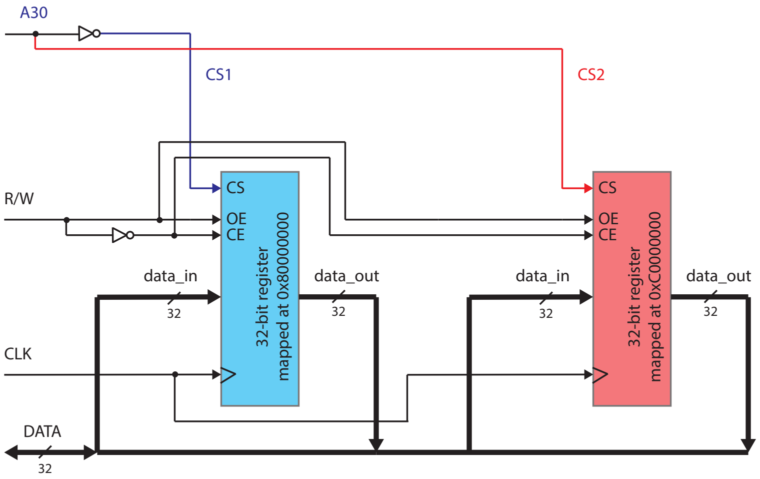
A30
CS1
CS2
R/W
CS
OE
CE
32-bit register mapped at 0x80000000
data_in
32
data_out
32
CLK
DATA
32
CS
OE
CE
32-bit register mapped at 0xC0000000
data_in
32
data_out
32

Fig. 1.6: Partial address decoding.

Let us return to our simple example with two memory-mapped registers. Recall that one register is accessible at address 0x80000000 and the other at address 0xC0000000. The two addresses differ only in address bit A30. Hence, we could select the registers based only on this bit and ignore all other address bits. We could select the first register when the address bit A30 is low and the second register when the address bit A30 is high. This solution is depicted in Figure 1.6.

But wait! The CS signal for the first register will now be active when CPU issues address 0x00000000, 0x80000000 or 0xA03F0147. Actually, it is selected whenever the CPU issues an address with bit A30 set low. The second register will be selected when the CPU issues any address with the address bit A30 set high. In other words, each register is assigned exactly half of the CPU address space and not only one particular address! But this is not a problem at all if we have only these two registers in the system. Even now, they can be selected with their previously assigned addresses 0x80000000 and 0xC0000000. This method of address decoding is called **partial address decoding**. This is contrary to the previously presented method, called **full address decoding**, where each register is assigned only one address from the address space. Here, using partial address decoding, both 0x80000000 or 0xA03F0147 addresses point to the same memory location (the first register). In general, a set of memory addresses that point to the same memory location or an I/O device is called **aliases**. Modern computer systems use partial address decoding whenever possible to reduce the number of logic gates required to implement address decoding logic.

1.4 Several memory mapped registers

This time, we want to connect eight registers to a CPU and map them into the CPU address space. Again, we will use partial address decoding to simplify the logic required to decode the addresses and select the registers. For this purpose, we will use an address decoder. Address decoders are fundamental logic components in digital systems, often used for selecting input/output devices. Recall that a 3-to-8 address decoder is a combinational logic circuit that takes a 3-bit binary input and activates one of its eight output lines based on the input value. Figure **??** depicts a 3-to-8 address decoder. The decoder has three input lines (A0, A1, and A2), representing a 3-bit binary number. These input lines can be either high (1) or low (0), creating eight possible binary combinations: 000 to 111. The decoder has eight output lines (Y0 to Y7), and each output corresponds to one of the possible input combinations. The operation of a 3-to-8 decoder can be described using the following truth table:

$A2$	$A1$	$A0$	$Y7$	$Y6$	$Y5$	$Y4$	$Y3$	$Y2$	$Y1$	$Y0$
0	0	0	0	0	0	0	0	0	0	**1**
0	0	1	0	0	0	0	0	0	**1**	0
0	1	0	0	0	0	0	0	**1**	0	0
0	1	1	0	0	0	0	**1**	0	0	0
1	0	0	0	0	0	**1**	0	0	0	0
1	0	1	0	0	**1**	0	0	0	0	0
1	1	0	0	**1**	0	0	0	0	0	0
1	1	1	**1**	0	0	0	0	0	0	0

When we provide a 3-bit binary number as input to the decoder, it decodes that binary value and activates the corresponding output line while setting all other output lines to 0. This operation allows us to select one of the eight output lines based on the input value. A 3-to-8 address decoder simplifies selecting devices based on a 3-bit binary input value and is used in address decoding to select one of eight memory-mapped I/O devices in a digital system.

Figure 1.7 shows the application of a 3-to-8 address decoder to select one of eight registers mapped into the CPU memory space. Each of the eight registers is assigned 1/8 of the CPU memory space in this case. For example, the first register will be accessible at addresses 0x00000000 to 0x1FFFFFFF. The second register is accessible at addresses 0x20000000 to 0x3FFFFFFF and so on, until the last one, which is accessible at addresses 0xE0000000 to 0xFFFFFFFF. Because of partial address decoding, the registers do not have only one address; instead, each register is assigned 512 MB (one-eighth of a 4GB) of address space.

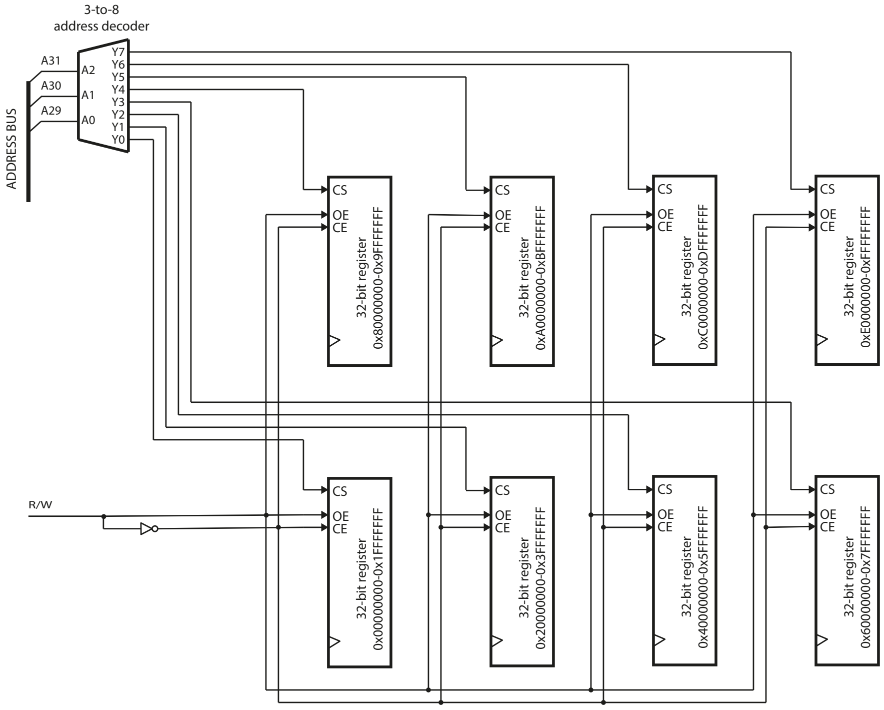

Fig. 1.7: Partial address decoding using a 3-to-8 address decoder to select eight registers.

1.5 Partial vs. Full Address Decoding

Let us summarize what we have learned so far. Partial address decoding and full address decoding are two different methods used in computer memory and memory-mapped I/O systems to determine which memory locations or I/O devices are accessed at a particular address.

Full address decoding involves all address lines generated by the CPU or processing unit to select a specific memory location or I/O device. It is usually used when we need to uniquely identify and select individual memory locations or I/O devices, each with a distinct address. Full address decoding provides precise control over memory or I/O access but requires more complex hardware, especially when dealing with a large number of unique addresses.

On the contrary, partial address decoding involves examining only a portion of the address lines generated by the CPU to decode an address and select a memory location or an I/O device. For example, suppose you have a memory system with 16 memory locations or I/O devices. In that case, we may use partial address decoding and compare only four higher-order address lines (e.g., A31-28) to determine which device is being accessed. The lower-order address lines (e.g., A27-A0) are ignored. This method is more efficient regarding hardware complexity than full address decoding because it reduces the number of logic gates required to decode an address.

In partial address decoding, **aliases** occur. Aliases are multiple addresses that map to the same memory location or I/O device. Aliases occur because only a portion of the address lines is used to select a specific memory location or device, allowing multiple addresses to access the same location due to address overlap. In general, aliases are not a problem. If address decoding is carefully designed and with appropriate software handling, aliases do not lead to conflicts in accessing memory or I/O resources. Despite aliases, partial address decoding offers several advantages over full address decoding. It reduces hardware complexity, lowers power consumption, simplifies PCB (printed circuit board) design and enables faster decoding.

The choice between partial address decoding and full address decoding depends on the specific requirements of the system design. Partial address decoding is often used when memory banks or I/O devices are organized in a structured way, with common prefixes, while full address decoding is necessary when each memory location or I/O device must have a unique address.

1.6 Case study: Using the GPIO Interface in FE310-G002 RISC-V based System-On-chip

PIO stands for General Purpose Input/Output, and it refers to a type of interface on a microcontroller that is used for simple digital input or output operations. GPIO interface controls GPIO pins that can be configured to serve various purposes, such as reading digital signals (input) or sending digital signals (output). GPIO pins are "general purpose" because they are not dedicated to a specific function. Instead, we can program them to perform various tasks based on the needs of your project.

GPIO pins can be configured as either input or output. Through input pins, the GPIO interface can detect whether the logical level on the pin is high (usually 3.3V or 5V) or low (0V). Through output pins, the GPIO interface can set the logic level on the pin to high or low, which we often use for tasks such as reading sensors (temperature, humidity, motion), controlling external devices like LEDs, controlling actuators (motors, relays), and interfacing with other digital devices.

A GPIO interface comprises a set of memory-mapped registers. These registers allow us to set the pin direction (input or output), read or write values to the pins, and handle events triggered by changes in the pin's state.

The SiFive Freedom FE310 is a microcontroller-based system-on-a-chip (SoC) developed by SiFive. The FE310 is built around the RISC-V E31 CPU core. The E31 CPU 32-bit core is based on the RISC-V RV32IMAC instruction-set architecture (ISA), which is an open-source and royalty-free ISA. RISC-V is gaining popularity in the embedded and processor design communities due to its flexibility, simplicity, and extensibility. The E31 RISC-V CPU comprises a single-issue, in-order pipeline.

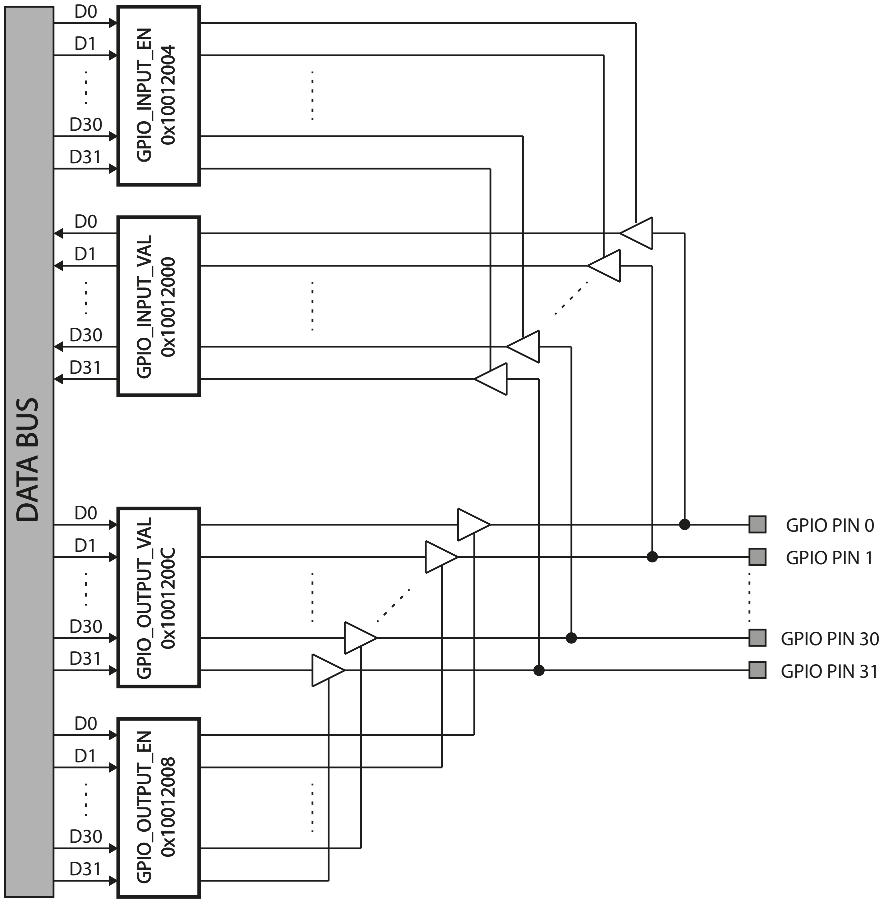

Fig. 1.8: A simplified steructure of the GPIO interface in SiFive Freedom FE310.

The pipeline comprises five stages: instruction fetch, instruction decode and register fetch, execute, data memory access, and register writeback. The pipeline has a peak execution rate of one instruction per clock cycle and is fully bypassed so that most instructions have a one-cycle result latency.

The FE310 includes on-chip memory components such as SRAM for program and data storage. Besides, it offers various peripherals and I/O options, including GPIO pins, UART (serial communication), SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), and timers. These peripherals enable the FE310 to interface with other hardware components and sensors.

The SiFive FE310 microcontroller had 32 GPIO pins. The GPIO interface in the SiFive FE310 comprises a set of special registers. Each bit in these registers manages the state and behaviour of a corresponding individual GPIO pin. These registers are part of the microcontroller's memory-mapped I/O (MMIO) address space. Figure 1.8 presents a simplified structure of the GPIO interface in the SiFive FE310. Several registers that are present in the GPIO interface are omitted for the sake of simplicity and clarity.

There are several key registers involved in configuring and controlling GPIO pins on the SiFive FE310:

1. **GPIO_INPUT_VAL**: This register stores the current input values of all GPIO pins. Each bit in this register corresponds to a specific pin, with '1' indicating a high voltage (logic level 1) and '0' indicating a low voltage (logic level 0).
2. **GPIO_OUTPUT_VAL**: This register stores the values to be output on the GPIO pins when they are configured as outputs.
3. **GPIO_OUTPUT_EN**: This register controls whether a GPIO pin is enabled as output by driving its tri-state buffer. When a bit in this register is '1', the corresponding bit in the **GPIO_OUTPUT_VAL** register is connected to the GPIO pin through the corresponding tri-state buffer. When the specific GPIO pin is output enabled, the content of the corresponding bit in the **GPIO_OUTPUT_VAL** register appears at the GPIO pin.
4. **GPIO_INPUT_EN**: This register controls whether a GPIO pin is enabled as input. When a bit in this register is '1', the GPIO pin is connected to the corresponding bit in the **GPIO_INPUT_VAL** register through the tri-state buffer. When the specific GPIO pin is input enabled, the content of the GPIO pin is stored in the corresponding bit in the **GPIO_INPPUT_VAL** register.

1.6.1 Program GPIO in Assembly

Using the GPIO interface to control the pins on the SiFive FE310 microcontroller in assembly language involves configuring the GPIO registers to control the behaviour of individual pins. To enable a GPIO pin as an output on the SiFive FE310 microcontroller using assembly language, we need to configure the **GPIO_OUTPUT_EN** register appropriately. Below is an example of enabling a GPIO pin as an output in assembly for the SiFive FE310. The pin number is given as the function parameter in the register `a0`:

```
; /*  GPIO output enable
;     Input: a0 - pin number
;     Output: None */
.align 2
.global gpio_output_en
.type gpio_output_en, @function
gpio_output_en:
    # prologue:
    addi sp, sp, -16     # Allocate the routine
                         #   stack frame
    sw ra, 12(sp)        # Save the return address
    sw fp, 8(sp)         # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16     # Set the framepointer

    # function body :
    li t0, 0x10012000   # load GPIO base address
    lw t1, 0x08(t0)     # read GPIO_OUTPUT_EN
    li t2, 0x01
    sll t2, t2, a0      # shift 1 to pin position
    or t1, t1, t2       # set the bit @ pin position
    sw t1, 0x08(t0)     # Store back

    # epilogue:
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)        # restore the frame pointer
    lw ra, 12(sp)       # restore the return address
    addi sp, sp, 16     # de-allocate the routine
                        #   stack frame
    ret
```

Listing 1.1: Assembly code used to implement the function that enables output on a GPIO pin.

Similarly, to enable a GPIO pin as an input on the SiFive FE310 microcontroller using assembly language, we need to configure the **GPIO_INPUT_EN** register appropriately. Below is an example of enabling a GPIO pin as an input in assembly for the SiFive FE310. The pin number is given as the function parameter in the register `a0`:

```
; /*  GPIO input enable
;    Input: a0 - pin number
;    Output: None */
.align 2
.global gpio_input_en
.type gpio_input_en, @function
gpio_input_en:
    # prologue:
    addi sp, sp, -16    # Allocate the routine
                        #   stack frame
    sw ra, 12(sp)       # Save the return address
    sw fp, 8(sp)        # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16     # Set the framepointer

    # function body :
    li t0, 0x10012000   # load GPIO base address
    lw t1, 0x04(t0)     # read GPIO_INPUT_EN
    li t2, 0x01
    sll t2, t2, a0      # shift 1 to the pin position
    or t1, t1, t2       # set the bit @ pin position
    sw t1, 0x04(t0)     # Store back

    # epilogue:
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)        # restore the frame pointer
    lw ra, 12(sp)       # restore the return address
    addi sp, sp, 16     # de-allocate the routine
                        #   stack frame
    ret
```

Listing 1.2: Assembly code used to implement the function that enables input on a GPIO pin.

To set a GPIO pin, we need to set the corresponding bit in the **GPIO_OUTPUT_VAL** register:

```
; /*  GPIO set pin
;     Input: a0 - pin number
;     Output: None */
.align 2
.global gpio_set_pin
.type gpio_set_pin, @function
gpio_set_pin:
    # prologue:
    addi sp, sp, -16    # Allocate the routine
                        #   stack frame
    sw ra, 12(sp)       # Save the return address
    sw fp, 8(sp)        # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16     # Set the framepointer

    # function body :
    li t0, 0x10012000   # load GPIO base address
    lw t1, 0x0C(t0)     # read GPIO_OUTPUT_VAL
    li t2, 0x01
    sll t2, t2, a0      # shift 1 to pin position
    or t1, t1, t2       # set the bit @ pin position
    sw t1, 0x0C(t0)     # Store back

    # epilogue:
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)        # restore the frame pointer
    lw ra, 12(sp)       # restore the return address
    addi sp, sp, 16     # de-allocate the routine
                        #   stack frame
    ret
```

Listing 1.3: Assembly code used to implement the function for setting a GPIO pin.

To reset a GPIO pin, we need to reset the corresponding bit in the **GPIO_OUTPUT_VAL** register:

```
; /*  GPIO clear pin
;     Input: a0 - pin number
;     Output: None */
.align 2
.global gpio_clear_pin
.type gpio_clear_pin, @function
gpio_clear_pin:
    # prologue:
    addi sp, sp, -16    # Allocate the routine
                        #   stack frame
    sw ra, 12(sp)       # Save the return address
    sw fp, 8(sp)        # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16     # Set the framepointer

    # function body :
    li t0, 0x10012000    # load GPIO base address
    lw t1, 0x0C(t0)      # read GPIO_OUTPUT_VAL
    li t2, 0x01
    sll t2, t2, a0       # shift 1 to pin position
    not t2, t2           # 1' complement
    and  t1, t1, t2      # clear pin
    sw t1, 0x0C(t0)      # Store back

    # epilogue:
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)         # restore the frame pointer
    lw ra, 12(sp)        # restore the return address
    addi sp, sp, 16      # de-allocate the routine
                         #   stack frame
    ret
```

Listing 1.4: Assembly code used to implement the function for resetting a GPIO pin.

A handy function is to toggle a GPIO pin. To toggle a GPIO pin, we need to EXOR the corresponding bit in the **GPIO_OUTPUT_VAL** register with '1':

```
; /*  GPIO clear pin
;     Input: a0 - pin number
;     Output: None */
.align 2
.global gpio_toggle_pin
.type gpio_toggle_pin, @function
gpio_toggle_pin:
    # prologue:
    addi sp, sp, -16     # Allocate the routine
                         #   stack frame
    sw ra, 12(sp)        # Save the return address
    sw fp, 8(sp)         # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16      # Set the framepointer

    # function body :
    # function body :
    li t0, 0x10012000    # load GPIO base address
    lw t1, 0x0C(t0)      # read GPIO_OUTPUT_VAL
    li t2, 0x01
    sll t2, t2, a0       # shift 1 to pin position
    xor t1, t1, t2       # toggle the bit @ pin position
    sw t1, 0x0C(t0)      # Store back

    # epilogue:
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)         # restore the frame pointer
    lw ra, 12(sp)        # restore the return address
    addi sp, sp, 16      # de-allocate the routine
                         #   stack frame
    ret
```

Listing 1.5: Assembly code used to implement the function for toggling a GPIO pin..

1.6.2 Program GPIO in C

We can also program a memory-mapped I/O device in C. We abstract an MMIO device with a C structure that represents and mirrors the layout of the registers in the MMIO device. We will present this concept using the GPIO Interface in FE310-G002 RISC-V based System-On-chip. To abstract GPIO registers with a C structure, we create a structure that mirrors the layout of the GPIO registers:

```
typedef struct
{
  volatile int GPIO_INPUT_VAL;
  volatile int GPIO_INPUT_EN;
  volatile int GPIO_OUTPUT_EN;
  volatile int GPIO_OUTPUT_VAL;
}GPIO_Registers_t;
```

Listing 1.6: A C structure that mirrors the GPIO registers layout.

This abstraction makes it easier to access and manipulate GPIO registers and pins and control their behaviour. Each member of the structure corresponds to a specific register in the GPIO interface, such as the input value register, output value register, etc. The layout of the members of the structure exactly mirrors the layout of the registers in memory, i.e. the members are in the same order as the registers in memory space. Recall that in C, the `volatile` keyword is used to indicate to the compiler that a variable can change its value at any time, even if it doesn't appear to be modified by the program. It informs the compiler that the variable should always be fetched from memory when needed rather than relying on cached values or optimizations that could result in unexpected behaviour. When working with hardware peripherals, we often access memory-mapped registers that control or represent hardware components. These registers can be modified by the hardware (e.g., GPIO pins) at any time outside our program, and the compiler might not be aware of these changes. By declaring such registers as volatile, you ensure the compiler generates code that correctly reflects the behaviour of hardware registers, making it suitable for hardware interaction.

Next, we define a pointer (in our example, the pointer is named GPIO, but you are free to use any name you wish) that holds the base address of the GPIO interface:

```
#define GPIO_BASEADDR      0x10012000

GPIO_Registers_t *GPIO = (GPIO_Registers_t*) GPIO_BASEADDR;
```

Listing 1.7: a pointer that holds the base address of the GPIO interface.

This pointer is used to access the GPIO registers as if they were part of a C structure. For example, we set pin 19 as output in the output enable register and toggle the state of pin 19 in the output value register:

```
GPIO->GPIO_OUTPUT_EN |= (0x01 << 19);
```

```
GPIO->GPIO_OUTPUT_VAL ^= (0x01 << 19);
```

Listing 1.8: Enabling and setting a GPIO in C.

Chapter 2
Interrupts and interrupt handling

CHAPTER GOALS

Have you ever wondered how computer components demand and get attention from the CPU? How do they tell the CPU or operating system that something important has just happened in the computer system, which requires an immediate response from the CPU, e.g., new data has just arrived at an I/O interface and should be processed immediately? This is done using so-called interrupts. This chapter will cover the theory and practice of interrupts and their handling. An interrupt is a hardware-initiated procedure that interrupts whatever program (CPU) is currently executing and requests that the CPU immediately start running another program that is written to service the particular interrupt request.
Upon completion of this chapter, you will be able to:

- Distinguish between interrupts and exceptions.
- Explain the operation of the interrupt signals.
- Explain the interrupt and exception handling.
- Explain the function of interrupt vectors and vector tabels.
- Explain the function of an interrupt controller.
- Explain the interrupts and interrupt handling in the Intel and ARM family of processors.

2.1 Introduction

During my childhood, there were two powerful military blocs in Europe and the world: the Eastern (Soviet) Block and the Western (USA) Block. That was a period of geopolitical tension between the Soviet Union and the United States and their respective allies, the Eastern Bloc and the Western Bloc. The country where I grew up, former Yugoslavia, was not part of any of these military blocks, though politically, it was closer to the eastern block. In the 1970s, former Yugoslav air force purchased a number of Soviet MIG-21 fighter aircraft from the USSR. The MIG-21 aircraft sold to Yugoslav air force had virtually no modern electronic devices, and the military of Yugoslavia wanted to install missile sensors in the planes. However, the USA and its allies have imposed an embargo on the purchase of electronic and computer components against Yugoslavia. Among all the universities in Yugoslavia, only the University of Ljubljana was allowed to purchase a few pieces (up to 20) of each chip that would be used only in the educational process. That's why the Yugoslav Army approached the University of Ljubljana to buy all the necessary electronic and computer components and develop a system that would be installed on the aircraft and would detect missiles. The system at the time had to be based on the modern Motorola 6800 microprocessors from the US. At its core, the system had a microcomputer built on the Motorola 6800 processor and a missile sensor. In addition to detecting missiles, the microcomputer had to do other things, also. If the missile sensor detected a rocket, the computer system had to immediately stop whatever it was currently doing and alert the pilot to the approaching missile. But how would a missile sensor be able to communicate to the CPU if the CPU could do nothing but fetch and execute instructions from memory? Remember that the CPU fetches and executes instructions every clock cycle. That's all it is able to do. So there must be some mechanism by which the CPU can be immediately interrupted and required to start another program. In our case, the CPU would run another program (e.g., display the current altitude and speed of the aircraft). In the event that the sensor detects a missile, it must, in some way, immediately suspend the currently running program and require the CPU to execute a program to flash the warning lights and alert the pilot. So, the CPU must have some mechanism in place to immediately stop the execution of one program and start another program. This mechanism is called **interrupts**, and the program that the CPU starts running in the response to an is called **interrupt service program (ISP)** or **interrupt handler**.

Interrupts and interrupt handling must be **transparent**. This means that the stopped (interrupted) program must not know that it has been stopped and must continue after the termination of the interrupt service program as if it had not been interrupted at all.

In the following chapters, we will learn about the interrupt mechanism and interrupt handling.

2.2 Interrupts

As we said in the Introduction, we want to have to ability to service external interrupts. This is useful if a device external to the processor needs attention. Figure 2.1 illustrates a simplified system with a CPU and a peripheral device. To be able to respond to interrupt requests from a peripheral device, a CPU usually has at least one interrupt request (IRQ) pin and one interrupt acknowledge (INTA) pin. The IRQ pin is the input used by a peripheral device to interrupt the processor (i.e., to interrupt the normal program flow in the CPU.). Since the CPU should finish executing the current instruction(s) before servicing any external interrupts, the peripheral device may have to wait for several clock cycles before the CPU responds to the interrupt request. The INTA pin is the output used to signal the peripheral device, which has requested an interrupt via the IRQ signal, that the CPU has started servicing the interrupt request and that the IRQ signal can be deactivated. Both pins in Figure 2.1, IRQ and INTA, are active low. Two resistors are used to establish a logic one on both signals IRQ and INTA (i.e., both signals are deactivated) when no one drives them.

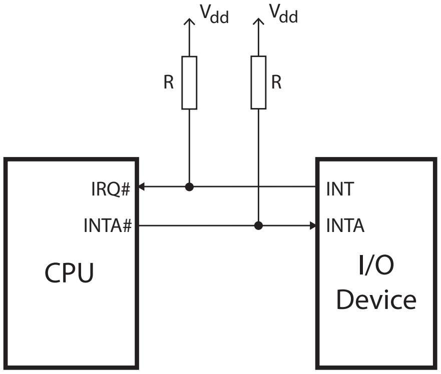

Fig. 2.1: A simplified block diagram of a computer system with interrupt controlling signals.

In general, CPUs can respond to interrupts in two different ways: in either an **edge-sensitive** or **level-sensitive** manner. In an edge-sensitive manner, the interrupt signal input is designed to be triggered by a particular signal edge (level transition): either a falling edge (high to low) or a rising edge (low to high). In a level-sensitive manner, the interrupt signal input is designed to be triggered by a logic signal level. A peripheral device invokes a level-triggered interrupt by driving the signal to and holding it at the active level. We refer to this operation as **asserting the signal**. It de-asserts the signal when the processor signals it to do so. One advantage of level-triggered interrupt inputs is that they allow multiple devices to share a common interrupt signal. Most often, the active level of an interrupt input signal is LOW. In such a case, the interrupt signal is tied to the HIGH voltage level using a pull-up resistor. When multiple peripheral devices share one level-triggered interrupt input signal, the device that wants to assert the interrupt request simply connects the signal to the ground (pulls thew signal LOW). The system in Figure 2.1 uses level-sensitive interrupt signals.

Summary: Assering and de-asserting a signal

Some signals are active high, and some signals are active low. To avoid the problem of high vs. low and the fact that for some signals, active means high and for some signals active means low, we just say asserted (activated) vs. de-asserted (deactivated).

When the device needs the attention from the CPU, it activates (asserts) the IRQ pin on the CPU. During the normal flow of execution through a program, the program counter increases sequentially through the address space, with branches to nearby labels or branches and links to subroutines. The CPU checks the status of the IRQ pin every time before a new instruction pointed to by the program counter is fetched from memory. When a peripheral device requests the interrupt, it is necessary to preserve the previous processor status while handling the interrupt, so that execution of the program that was running when the interrupt request occurred can resume when the appropriate interrupt handler has completed. We say that the interrupts must be 100% transparent. So, when an interrupt request occurs, the CPU

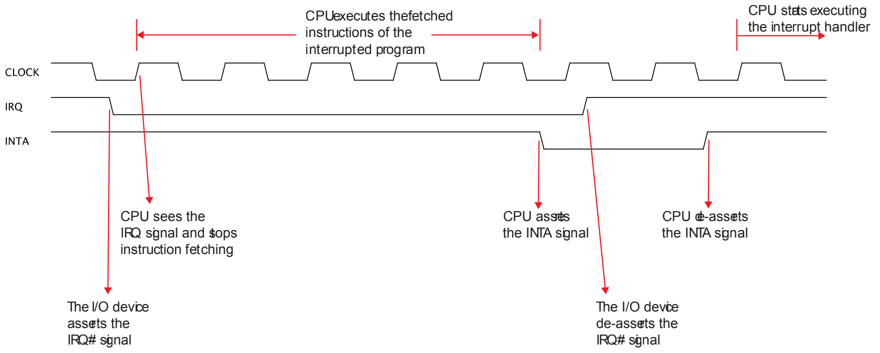
CPUexecutes thefetched instructions of the interrupted program
CPU starts executing the interrupt handler
CLOCK
IRQ
INTA
CPU sees the IRQ signal and stops instruction fetching
CPU asserts the INTA signal
CPU de-asserts the INTA signal
The I/O device asserts the IRQ# signal
The I/O device de-asserts the IRQ# signal

Fig. 2.2: A timing diagram for an external interrupt request.

completes the current instruction and asserts the INTA signal. When a peripheral device sees the INTA signal, it de-asserts the IRQ signal. Figure 2.2 shows the timing diagram for an external interrupt request for the simple system from Figure 2.1.

Then the CPU saves the part of the context of the interrupted program in the stack. A context is a state of the program counter, status register, stack pointer, and all other program-visible CPU registers. Some CPUs save the whole context in the stack, while others save only a part of the context in the stack. Since interrupts can happen at any time, there is no way for the active programs to prepare for the interrupt (e.g., by saving registers that the interrupt handler might write to). It is important to note that calling conventions do not apply when handling interrupts: the interrupt is not being "called" by the active program; it is interrupting the active program. Thus, the interrupt handler code must preserve the content ensure that it does not overwrite any registers that the program may be using before their content is saved. After the CPU has saved the context, the CPU automatically loads the address of the interrupt handler into the program counter. The interrupt handler is a program written by the user and depends on the peripheral device's functionality. Depending on how much of the context is automatically saved by the CPU, the interrupt handler must first save every register it intends to use in the stack or somewhere in memory.

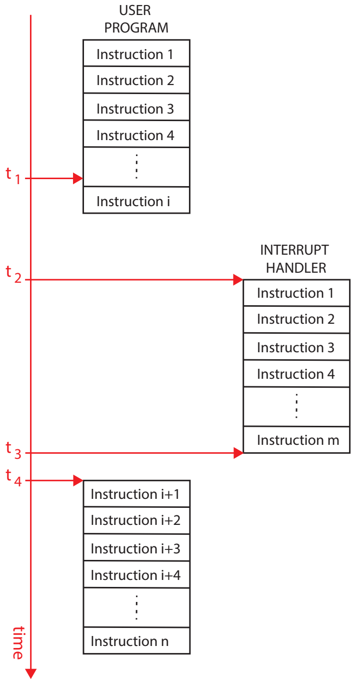

Fig. 2.3: The procedure involved in interrupts.

Figure 2.3 shows the procedure involved in interrupts: the CPU executes the sequence of instructions from a user program until an interrupt request occurs at the time t_1. When the IRQ signal is asserted, the CPU stops executing the user code and starts executing the interrupt handler. But before executing the interrupt handler at time t_2, the CPU must finish the execution of already fetched instructions, save the (part of) context, and obtain the address of the interrupt handler. The time $t_2 - t_1$ required for this procedure is called **interrupt latency**. In general, interrupt latency is the time that elapses from when the IRQ signal is asserted to when the CPU starts to execute the interrupt handler. Interrupt latency duration is usually not predetermined and depends on how many instructions are already in the CPU's pipeline, on how CPU saves the context and on whether any new interrupt requests are temporarily disabled. Once the CPU completes the execution of the interrupt handler at time t_3, it returns back to the execution of the user code at time t_4. Before returning to user code, the CPU must automatically restore the previously saved context.

2.2.1 Types of interrupts

There are typically three types of interrupts regarding the source of the interrupt: external interrupts (or simply interrupts), traps or exceptions, and software interrupts. External interrupts are triggered by an external device by activating the interrupt request pin on the CPU. Traps or exceptions are activated internally in the CPU, usually as a result of some exceptional condition caused by instruction. For example, traps are caused when illegal or undefined instruction is fetched, or when the CPU attempts to execute an instruction that was not fetched because the address was illegal. A special instruction triggers software interrupts. Such instructions function similarly to subroutine calls, but the subroutine, in this case, the interrupt handler, is not being "called", but an interrupt-like sequence occurs. These software-interrupt instructions are useful when the user program does not know or is not allowed to know the address of the routine which it would like to "call", e.g., they are usually used for requesting operating system services and routines.

External interrupts are divided into two types: maskable and non-maskable interrupts. Maskable interrupts can be enabled or disabled by setting a bit in the CPU's control register or by executing a special instruction. For example, Intel has the CLI instruction to mask the interrupts, and ARM has CPSID instruction for this purpose. Non-maskable interrupts have a higher priority than maskable interrupts. That means that if both maskable and non-maskable interrupts are activated at the same time, the CPU will service the non-maskable interrupt first.

2.2.2 Handling interrupts

In a situation where multiple types of interrupts and exceptions can occur, there must be a mechanism in place where different handler code can be executed for different types of events. In general, there are two methods for handling this problem: polled interrupts and vectored interrupts.

In polled interrupts, the processor branches to a specific address that begins a sequence of instructions that check the cause of the interrupt or exception and branch to handler code for the type of interrupt/exception encountered. This is also called polled interrupt/exception handling.

In vectored interrupts, the processor branches to a different address for each type of interrupt or exception. Each exception address is separated by only one word, and these addresses form a table called **interrupt vector table**. Each entry of the interrupt vector table is called **interrupt vector**, and it is the address of an interrupt handler. Hence, the vector table contains the start addresses, called interrupt vectors, for all exception handlers. This method is called **vectored interrupt handling**. This concept is common across many processor architectures, although interrupt vector tables may be implemented in other architecture-specific fashions. For example, another common concept is to place a jump instruction (instead of vectors) at each entry in the table. Each of these jump instructions forces the processor to jump to the handler code for each type of interrupt/exception. In this case, the address of each table entry is considered as an interrupt vector.

2.2.3 ARM 9 interrupts

The ARM9 supports the following six types of interrupts and exceptions:

- Fast interrupt Request,
- Interrupt Request,
- Data and Prefetched abort exceptions,
- Undefined instruction exception, and
- Software interrupt, and
- Reset.

The interrupt instruction SWI raises the software interrupts. The software interrupts allow a program running in the user mode to request privileged operations such as OS functions. The Prefetch abort exception occurs when the CPU fetches an instruction from an illegal address. The Data abort exception occurs when a data transfer instruction attempts to load or store data at an illegal address. The Undefined instruction exception occurs when the processor cannot recognize the currently fetched instruction. The Interrupt request occurs when the processor's external interrupt request pin (IRQ) is asserted (LOW), and the interrupt mask bit (I) in the current program status register (CPSR) is cleared (interrupts enabled). The Fast interrupt request occurs when the processor's external fast interrupt request pin (FIQ) is asserted (LOW), and the interrupt mask bit (F) in the current program status register (CPSR) is cleared (fast interrupts enabled). The Reset interrupt occurs when the processor's reset pin is asserted.

2.2.3.1 Vector table and interrupt priorities

ARM9 processors use the vectored interrupt handling method. Each interrupt/exception has its own entry in the vector table. Each entry in the vector table has only 32 bits, which is not enough to contain the full code for a handler; hence, each entry commonly contains a branch instruction or load pc instruction to the actual handler. Table 2.1 shows the interrupt/exception, its address in the vector table, and its priority. As interrupts/exceptions can coincide, the CPU has to use a priority mechanism to handle the most important interrupt/exception. For example, the Reset interrupt

Table 2.1: ARM9 vector table.

Interrupt/Exception	**Vector Table Address**	**Priority (1-High, 6-Low)**
Reset	0x00000000	1
Undefined Instruction	0x00000004	6
Software Interrupt	0x00000008	6
Prefetch Abort	0x0000000C	5
Data Abort	0x00000010	2
Interrupt Request	0x00000018	4
Fast Interrupt Request	0x0000001C	3

has the highest priority, and it takes precedence over all other interrupts/exceptions. All interrupts/exceptions disable further interrupts/exceptions by setting the I bit in the CPSR register. The Reset and Fast Interrupt Request also set the F bit in the CPSR register and thus mask the Fast interrupt request. Listing 2.1 shows a typical method of implementing a vector table for ARM9 processors.

```
        .org 0x00000000
Vector_Table:
        b Reset_Handler
        b Undefined_Handler
        b SWI_Handler
        b Prefetch_Handler
        b Abort_Handler
        nop                     // never used
        b IRQ_Handler
        b FIQ_HAndler


Reset_Handler:
        <handler instructions>
Undefined_Handler:
        <handler instructions>
SWI_Handler:
        <handler instructions>
Prefetch_Handler:
        <handler instructions>
Abort_Handler:
        <handler instructions>
IRQ_Handler:
        <handler instructions>
FIQ_Handler:
        <handler instructions>
```

Listing 2.1: ARM vector table and interrupt handlers.

Listing 2.1 shows a typical method of implementing a vector table for ARM9 processors. The vector table starts at the address 0x00000000. Each entry in the vector table is 32 bits long and contains a branch instruction (B) to the interrupt handler. When, for example, a Data Abort exception occurs, the CPU stops the execution of the current running program, saves the program context, and moves the vector

0x00000010 into the program counter. This way, the b Abort_Handler instruction is fetched, and the CPU jumps to Abort_Handler.

As we already said, the Reset interrupt is the highest priority interrupt and is always taken whenever the Reset pin is asserted. The reset handler is responsible for initializing the system and other interrupt sources, and to set the stack pointer. So the Reset interrupt masks automatically all other interrupts before their sources are initialized. Only then the reset handler enables other interrupts. Hence, during the first few instructions of the reset handler, we should avoid SWI, undefined instructions, and memory accesses that can cause the Data and Prefetch aborts.

The Fast Interrupt Request (FIQ) occurs when a peripheral asserts the processor's FIQ pin. The peripheral device mus hold the FIQ input low until the processor acknowledges the interrupt request. As a response to FIQ, the CPU disables both Interrupt and Fast Interrupt requests. Hence, no external device can interrupt the CPU unless the IRQ and FIQ interrupts are re-enabled by software. The Fast Interrupt Request reduces the execution time of the exception handler relative to a normal interrupt by removing the requirement for register saving (minimizing the overhead of context switching).

The Interrupt Request (IRQ) is a normal interrupt that occurs when a peripheral device asserts the IRQ pin. The peripheral device mus hold the IRQ input pin low until the processor acknowledges the interrupt request. An IRQ has a lower priority than the FIQ and Data Abort and is masked on entry to an FIQ or Data Abort sequence. On entry to the IRQ handler, the further IRQ interrupts are disabled and should remain disabled until the current interrupt source has been acknowledged, and the IRQ pin has been de-asserted.

We can notice from Table 2.1 that both Software Interrupt and Undefined Instruction have the same level of a priority since they cannot occur at the same time.

2.2.3.2 ARM9 interrupt handling

ARM9 processors are 5-stage pipelined machines with Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM) and Write-Back (WB) stages. In a pipelined machine, an instruction is executed step by step and is not completed for several clock cycles. An external interrupt can occur at any time during the execution of an instruction. Also, other instructions in the pipeline can raise exceptions that may force the machine to abort the instructions in the pipeline before they have been completed. One of the problems with interrupts in the pipelined CPUs is when to halt instruction in the pipeline. In the case of external interrupts, one possible solution would be to execute all fetched instructions before handling the interrupt request. But the problem with this approach would be a long interrupt latency. The other solution would be to halt the execution of all fetched instructions and fetch them again upon returning from the interrupt handler. This way, we would have minimal interrupt latency. Obviously, this is not a good idea because some instructions, such as STORE instructions, can modify the content in memory and should not be stoped and executed again. Also, arithmetic instructions might have

2.5 Case Study: Exceptions in FE310-G002 RISC-V based System-On-Chip

RISC-V architecture defines different privilege modes that determine the level of access and control a program or process has over the system's resources. A privileged mode in a CPU refers to a specific operating mode in which the CPU has access to various system resources. Privileged modes are often used in modern computer architectures to ensure the proper operation, security, and control of the system. Privileged modes are crucial in separating user-level programs from system-level operations and for managing system security, isolation, and resource allocation. For example, a modern CPU restricts a user program from accessing system critical resources (e.g. special CPU registers, memory regions, special instructions, etc.), while the system programs may access all system resources. Privileged modes are the mechanism to achieve this differentiation between user-level and system-level programs. Modern CPUs usually have a separate set of control and status registers (CSRs) for each privileged mode and a special control register that tells which privileged mode the CPU is currently running. Depending on the status of this special control register (i.e. current privileged mode), the CPU can access the corresponding set of CSRs and execute only the instructions allowed in the current privileged level. For example, if the CPU is currently running in a user-privileged mode, it can execute only the standard instruction set. At the same time, executing some special instructions that can alter critical system resources is prohibited. Besides, programs running in user-privileged mode can never alter the content of this special control register and thus switch between privileged modes. But wait, how can we change a privileged mode once the CPU runs in user-privileged mode? Well, it depends on the current privileged mode:

1. If the CPU runs in user-level privileged mode, the only way to switch to a system-level privileged mode is through exceptions (traps or interrupts). Exceptions can trigger mode transitions. When an exception (a trap or an interrupt) occurs, the CPU automatically switches to system-level privileged mode, and the exception handling routine executes in the system-level privileged mode. Upon exiting the exception handler, the CPU automatically switches to the previous (e.g., user-level) privileged mode.
2. If the CPU runs in system-level privileged mode, the CPU can switch to a user-level privileged mode simply by executing a special instruction that alters the content of the special control register and, hence, changes the current system-level privileged mode to user-level privileged mode. CPUs have specific instructions that are used to initiate mode transitions. These instructions are often called privileged and can only be executed when the CPU is in a system-level privileged mode.

2.5.1 RISC-V Privileged Modes

In order to be able to understand interrupts and interrupts handling in RISC-V, we'll briefly describe and explain the privileged modes in RISC-V. Privileged modes are a fundamental part of RISC-V's flexibility, as they enable various operating systems, hypervisors, and security models to be implemented on the same instruction set architecture. Here is a brief description and explanation of three basic privileged modes in RISC-V:

1. **User Mode (U)**: User mode is the lowest privilege mode in RISC-V. In this mode, a user-level application or program runs with restricted access to system resources. User mode provides the least privilege and is suitable for application-level code. In user mode, applications can execute most instructions but have limited access to privileged instructions and control registers. User mode can execute basic instructions, access memory, and perform arithmetic operations. However, it cannot directly manipulate control and status registers (CSRs) related to exception handling or interrupt control.
2. **Supervisor Mode (S):** Supervisor mode is a privilege level above user mode. It is designed for operating system kernel code, which needs greater control over system resources and privilege to perform tasks like context switching and managing hardware devices. Supervisor mode has more access to control registers and instructions compared to user mode. It can perform operations related to exception handling, interrupt control, and system management. S-mode can execute privileged instructions that deal with system control and exception handling. It can access and modify most control and status registers (CSRs), including those related to interrupts and exceptions.
3. **Machine Mode (M):** Machine mode is the highest privilege mode in RISC-V. It provides complete control over the system, including access to all resources and system-wide configuration. M-mode has full access to all instructions, control registers, and hardware resources, making it suitable for tasks such as system initialization, low-level device control, and platform management. M-mode can execute all RISC-V instructions, including those reserved for privileged and system-level operations. It can access and modify all control and status registers (CSRs), and it has control over exceptions and interrupts across all privilege levels. Upon reset, RISC-V enters machine mode.

The E31 RISC-V core in FE-310 SoC supports only Machine and User privilege modes. The transition between privilege modes in E31 RISC-V is typically controlled by changing specific bits in control and status registers (CSRs). The machine mode handles these transitions, ensuring that the processor switches between user and machine modes appropriately. Additionally, exceptions and interrupts may trigger mode transitions, allowing the processor to respond to exceptional conditions or external events. As all exceptions (traps and interrupts) execute in Machine mode, we will restrict the description of exceptions only to this privilege mode.

2.5.2 RISC-V Machine Modes Exceptions

According to the RISC-V Privileged Architecture [**?**], the E31 RISC-V CPU comprises five control and status registers for Machine privilege mode:

1. **mstatus:** In RISC-V, the **mstatus** (Machine Status) register is a critical control and status register (CSR) used to manage and store various information related to the Machine privilege mode. The **mstatus** register plays a central role in controlling exception handling, interrupt handling, and the overall operation of the processor in machine mode. The **mstatus** register keeps track of and controls the CPU's current operating state, including whether or not interrupts are enabled. A summary of the **mstatus** bits related to interrupts in the E31 RISC-V CPU is provided in Figure 2.26. Note that this is not a complete description of **mstatus** as it contains fields unrelated to interrupts. For the full description of **mstatus**, please consult the RISC-V Instruction Set Manual, Volume II: Privileged Architecture. The **mstatus** register contains the following exception-related bits:

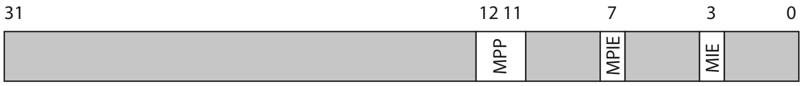

Fig. 2.26: The **mstatus** register.

   a. **MIE** (Machine Interrupt Enable): This bit controls whether machine-level interrupts are globally enabled or disabled. When MIE is set, the CPU can process machine-level interrupts; when it is cleared, machine-level interrupts are disabled.
   b. **MPIE** (Machine Previous Interrupt Enable): This bit stores the previous state of MIE before it was modified due to an interrupt. It helps manage interrupt nesting by preserving the previous interrupt-enable state.
   c. **MPP** (Machine Previous Privilege Mode): This two-bit field stores the previous privilege mode before the CPU entered machine mode due to an interrupt. It is used during return from interrupt to return to the appropriate privilege mode after processing an interrupt.

2. **mie:** The **mie** (Machine Interrupt Enable) register is responsible for enabling or disabling various types of interrupts that can interrupt the execution of the CPU in machine mode. Individual interrupts are enabled by setting the appropriate bit in the **mie** register. The **mie** register is depicted in Figure 2.27. The **mie** register contains the following bits:

   a. **MSIE** (Machine Software Interrupt Enable): This bit controls whether machine-level software interrupts are enabled or disabled. When MSIE

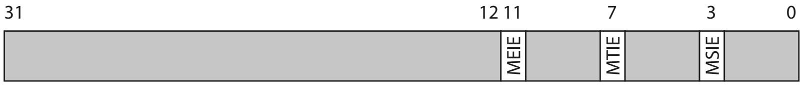

Fig. 2.27: The **mie** register.

is set, the CPU can process machine-level software interrupts; otherwise, machine-level software interrupts are disabled.

b. **MTIE** (Machine Timer Interrupt Enable): This bit controls whether machine-level timer interrupts are enabled or disabled. When MTIE is set, the CPU can process machine-level timer interrupts.
c. **MEIE** (Machine External Interrupt Enable): This bit controls whether machine-level external interrupts are enabled or disabled. When MEIE is set, the CPU can process machine-level external interrupts.

3. **mip:** The **mip** (Machine Interrupt Pending) register indicates which interrupts are currently pending. The **mip** register is depicted in Figure 2.28. When an interrupt occurs, the corresponding bit in **mip** is set to 1. When the CPU takes an interrupt, the corresponding bit in **mip** is cleared. The **mip** register contains the following bits:

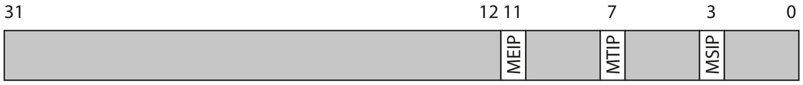

Fig. 2.28: The **mip** register.

a. **MSIP** (Machine Software Interrupt Pending): When MSIP is set, the Machine Software Interrupt is pending.
b. **MTIP** (Machine Timer Interrupt Pending): When MTIP is set, the Machine Timer Interrupt is pending.
c. **MEIP** (Machine External Interrupt Pending): When MEIP is set, the MachineExternal Interrupt is pending.

If more than one interrupt is pending, the RISC-V CPU prioritizes the interrupts as follows, in decreasing order of priority: Machine External Interrupts (highest priority), Machine Software Interrupts, and Machine Timer Interrupts (lowest priority).

4. **mcause:** In RISC-V architecture, the **mcause** register is a control and status register (CSR) that is used to provide information about the cause of an exception or interrupt that occurred in machine mode. A summary of the **mcause** bits related to interrupts in the E31 RISC-V CPU is provided in Figure 2.29. When a trap is taken in machine mode, the most significant bit in **mcause** (bit INT) is 0,

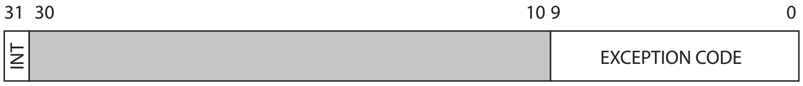

Fig. 2.29: The **mcause** register.

and the ten least-significant bits (EXCEPTION CAUSE field) are written with a code indicating the event that caused the trap. When an interrupt is taken, the most significant bit of **mcause** (bit INT) is set to 1, and the ten least-significant bits (EXCEPTION CAUSE field) contain the interrupt number, using the same encoding as the bit positions in the **mip** register. Table 2.4 lists exception codes and their description. For example, a Machine Timer Interrupt causes **mcause** to be set to `0x80000007`.

Table 2.4: **mcause** Exception Codes and their description.

INT	**EXCEPTION CODE**	**Description**
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 address misaligned
0	7	Store access fault
1	3	Machine software interrupt
1	7	Machine timer interrupt
1	11	Machine external interrupt

5. **mtvec:** The **mtvec** register has two main functions. Firstly, it specifies the base address for the vector table, which contains the addresses of exception handlers. Secondly, it sets the mode by which the E31 CPU will process exceptions. The RISC-V CPU can process exceptions in two modes: direct and vectored. In **direct mode**, the **mtvec** register holds the address of a single global exception handler. The processor directly jumps to this global handler's address when a trap or interrupt occurs. In direct mode, we might use a single handler for all exceptions, simplifying the exception-handling process. However, it may not be suitable for systems requiring fine-grained control over exception handling. In **vectored mode**, the **mtvec** register holds the base address of the vector table. In this mode, the processor uses a 10-bit field in the In **mcause register** to index the vector table and find the appropriate handler for the specific trap or interrupt that occurred. The vectored mode allows more flexibility in handling various exceptions and interrupts with different routines. In vectored mode, we can have multiple handlers for different exceptions and interrupts, allowing us to handle

each type of exception or interrupt differently. This is often the preferred way of interrupt handling. The **mtvec** register is depicted in Figure 2.30. The **mtvec** register contains the following bit fields:

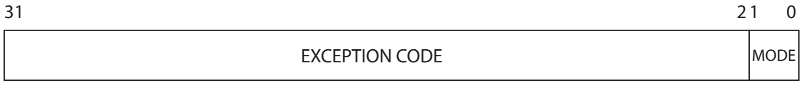

Fig. 2.30: The **mstvec** register..

a. **MODE:** This 2-bit field sets the interrupt processing mode (00-Direct, 01-Vectored).
b. **BASE:** This 30-bit field contains the vector table base address. This field requires 64-byte alignment.

Table 2.5 describes how an address of the exception handler is computed in two different interrupt processing modes. For example, suppose the global and machine timer interrupts are enabled, and the BASE is set to 0x20011500. If the vectored mode is selected and a machine timer interrupt occurs, the EXCEPTION CODE in the **mcause** register will be 0x07. Then, the address of the interrupt handler that processes the machine timer interrupts will be 0x20011500 + 4 x 0x07 = 0x20011500 + 0x1C = 0x2001151C. Hence, when the interrupt is taken, the PC is set to 0x2001151C, and the first instruction of the interrupt handler should be at this address.

Table 2.5: **mtvec** Modes and Address of Exception Handler Encoding.

MODE	**Interrupt Processing Mode**	**Address of Exception Handler**
0	Direct	`PC = BASE`
1	Vectored	`PC = BASE + 4 x mcause[EXCEPTION CODE]` *NOTE: BASE must be 64-byte aligned. This is to avoid an adder in the above computation*

Configuring these five Control and Status Registers registers correctly is crucial for proper exception handling in RISC-V systems, as they dictate where the processor should jump when an exception occurs and how exceptions are managed. These CSRs are not memory-mapped and can only be accessed through special privileged instructions: **csrr** and **csrw** for read and write, respectively. Hence, To work with these CSRs, developers must use assembly language instructions to read and modify these registers as needed.

2.5.3 FE-310 Interrupts

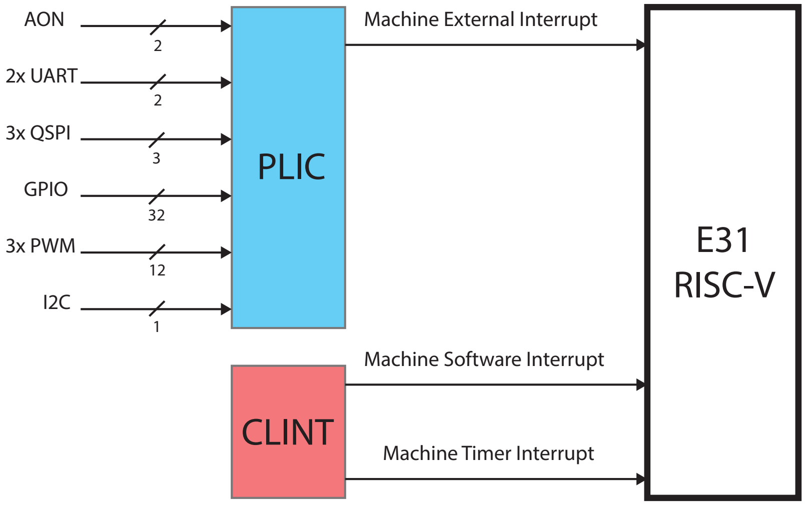

Fig. 2.31: FE310 Interrupt Architecture Block Diagram.

The FE310 SoC supports two types of RISC-V interrupts: local and global. Local interrupts are signalled directly to the RISC-V E31 CPU with a dedicated interrupt line for each local interrupt. The RISC-V E31 CPU has three interrupt lines for external, software and timer interrupts (Figure 2.31). Software and timer interrupts are local interrupts generated by the Core-Local Interruptor (CLINT). Besides software and timer interrupts, various I/O devices (e.g., UART, GPIO, etc.) can use global interrupts to activate the external interrupt line and to interrupt the CPU. Global interrupts from I/O devices are routed through a Platform-Level Interrupt Controller (PLIC), which will be described later.

The CLINT is a mandatory component in RISC-V processor systems. It's responsible for managing timer-related and software-generated interrupts at the core level. The CLINT generates two interrupts:

1. Machine Timer Interrupts: The CLINT contains a timer called the Machine Timer, which can generate timer interrupts for various purposes, including timekeeping, scheduling, and triggering tasks at specific intervals.
2. Machine Software Interrupts: In RISC-V, the software can generate software interrupts to communicate with the operating system. In general, the program running in user mode is not allowed to call operating system procedures. Hence, the only way a user program makes a system call is by generating a software interrupt. The software interrupt handler running in machine mode then calls an

operating system procedure. The CLINT can be used to handle these software-generated interrupts.

The CLINT comprises memory-mapped control and status registers related to software and timer interrupts. Table 2.6 shows the memory map for CLINT on SiFive FE310.

Table 2.6: Memory map for CLINT registers on SiFive FE310 SoC.

Address	Width	Register
0x02000000	4B	**msip**
0x02004000	8B	**mtimecmp**
0x0200BFF8	8B	**mtime**

2.5.3.1 Machine Software Interrupts

A machine software interrupt is an interrupt generated by software running in machine mode to request attention from the processor for specific tasks or events. Machine software interrupts are generated by writing '1' to the **msip** register within CLINT. The **msip** register is a 32-bit memory-mapped register where the upper 31 bits are hardwired to zero. The least significant bit of the **msip** register is reflected in the MSIP bit of the **mip** register. On reset, the **msip** register is cleared to zero.

2.5.3.2 Machine Timer Interrupts

CLINT, which is a mandatory part of RISC-V architecture, provides a 64-bit real-time counter, which monotonically increases at a clock speed, and its content is visible as a memory-mapped register **mtime**. In the FE310 SoC, CLINT is responsible for providing the real-time counter. Machine timer interrupt is a local interrupt, which can be generated by using two architecturally defined timer registers: **mtime** and **mtimecmp**:

1. **mtime** register: The 64-bit **mtime** register stores the current value of the 64-bit timer counter. The software can read this register to determine the current time.
2. **mtimecmp** register: The **mtimecmp** register holds a value that is compared with the **mtime** register. When **mtime** reaches the value stored in **mtimecmp**, it triggers a timer interrupt. This register is used to set up timer interrupts for specific time intervals.

In summary, the machine timer generates timer interrupts when the **mtime** matches or exceeds the value stored in the **mtimecmp** register. This feature is crucial for implementing preemptive multitasking, where the processor can switch between tasks at predefined time intervals.

2.5.4 Interrupt Entry and Exit

Interrupt entry and exit refer to the processes by which a RISC-V processor handles interrupts. These processes involve transitioning from regular program execution to an interrupt handler and returning to regular program execution after the interrupt is serviced. In the following subsections, we describe and explain interrupt entry and exit in RISC-V.

2.5.4.1 Interrupt Entry

When a machine interrupt occurs:

1. The value of the MIE bit in **mstatus** is copied into the MPIE bit in **mstatus**, and then MIE is cleared, effectively disabling interrupts.
2. The privilege mode prior to the interrupt is saved in the MPP field in **mstatus**.
3. The cause of the interrupt is encoded into EXCEPTION CODE in **mcause**.
4. The current PC is copied into the **mepc** register, and then the PC is set to the value specified by textbfmtvec as described in Table 2.5.

At this point, control is handed over to software in the interrupt handler with interrupts disabled. Interrupts can be re-enabled by explicitly setting the MIE bit in **mstatus** or by executing an `mret` instruction to exit the handler.

2.5.4.2 Interrupt Exit

To exit from a machine interrupt, the `mret` instruction must be executed at the end of the interrupt handler. When a `mret` instruction is executed, the following occurs:

1. The privilege mode is set to the value encoded in the MPP field in **mstatus**.
2. In the **mstatus** register, the MIE bit is set to the value of MPIE.
3. The PC is set to the value of **mepc**, hence pointing to the instruction, which was interrupted.

At this point, control is handed over to the previously interrupted program.

2.5.5 Implementing Vector Table and Handlers

Implementing a vector table and handlers in assembly language for RISC-V involves setting up a program structure to store the addresses of exception handlers and configuring the system to use this table when exceptions occur to jump to the interrupt-specific handler. Below are the steps to implement an exception table and handlers in RISC-V assembly:

1. **Define the Vector Table**: Create a program structure that serves as the vector table. As we have learned, the address of the first instruction of an interrupt handler is calculated using the BASE address of the vector table and the exception cause (Table 2.5). Each entry in the vector table occupies exactly 4 bytes, and there is only room for one instruction per handler in the vector table. Therefore, the only instructions in the exception table should be the jump instructions that transfer control to an interrupt-specific handler. An example of the vector table is as follows:
   The vector table is populated with jump instructions to transfer control to interrupt-specific handlers. For example, the jump instruction (`j _mtim_interrupt_handler`) that causes the jump to the timer interrupt handler is placed at the offset 7 x 4 = 0x1C from the beginning of the vector table. So when a machine timer interrupt occurs, the PC is set to BASE + 0x1C and the CPU will execute the `j _mtim_interrupt_handler` instruction.

```
# ---------------------------------
#
#   V E C T O R    T A B L E
#
#     must be 64-byte aligned.
# ---------------------------------

.balign 64
.global _vector_table
_vector_table:                    # BASE
    j _default_handler
    j _default_handler
    j _default_handler
    # -----------------------------
    j _msw_interrupt_handler      # 3
    # -----------------------------
    j _default_handler
    j _default_handler
    j _default_handler
    # -----------------------------
    j _mtim_interrupt_handler     # 7
    # -----------------------------
    j _default_handler
    j _default_handler
    j _default_handler
    # -----------------------------
    j _mext_interrupt_handler     # 11
    # -----------------------------
```

Listing 2.10: A vector table for E31 RISC-V.

   We can see from Listing 2.10 that besides the jump instructions to exception handlers for software, timer and external interrupts, there is also a jump instruction to `_default_handler` in all other entries in the vector table. We have already learned that there are only three interrupt sources in FE310 SOC (software, timer and external), so why do we need the fourth interrupt handler `_default_handler`? This is to ensure that in case of a trap (INT=0 in **mcause**), the CPU executes `_default_handler`.
2. **Register the Base Vector Table Address:** We should configure the **mtvec** register to point to the exception table. Also, we should set the preferred interrupt

processing mode in mtvec. Listing 2.11 presents the RISC-V assembly code to register the base address and to select the vectored mode:

```
#---------------------------------------
#     Register the base address for vector table
#       in mtvec
#
#@arguments:
#     # a0 - interrupt vector table base address
#     # a1 - interrupt processing mode
#             (0x0 - direct, 0x1 - vectored)
#---------------------------------------
.balign 4
.global register_handler
.type register_handler, @function
register_handler:
    # prologue:
    addi sp, sp, -16      # Allocate the routine
                          #   stack frame
    sw ra, 12(sp)         # Save the return address
    sw fp, 8(sp)          # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16       # Set the framepointer

    or a0, a0, a1         # OR base address with mode
    csrw mtvec, a0        # and save into mtvec

    # epilogue:
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)          # restore the frame pointer
    lw ra, 12(sp)         # restore the return address
    addi sp, sp, 16       # de-allocate the routine
                          #   stack frame
    ret
```

Listing 2.11: Assembly function for registering the vector table base addreess.

3.
4. **Define Exception Handler:** Write the exception handler routines in assembly language. Each handler should be a separate section of code that corresponds to a specific exception type and ends with the `mret` instruction. The prologue of an interrupt handler usually begins with saving the registers onto the stack to avoid overwriting the contents of the saved registers (s0-s11). After the body of the exception handler executes, the epilogue of an interrupt handler restores the saved registers from the stack. Finally, the handler returns with `mret`, an instruction unique to machine mode. The `mret` instruction restores the PC from **mepc**, the previous interrupt-enable setting, and the privilege mode as described in Subsection 2.5.4.2. For example, the following code (Listing **??**) presents the RISC-V assembly code for a machine timer interrupt handler:

```
%%
%% This is file `h.tex',
%% generated with the docstrip utility.
%%
%% The original source files were:
%%
```

```
%% fileerr.dtx  (with options: `help')
%%
%% This is a generated file.
%%
%% The source is maintained by the LaTeX Project team and bug
%% reports for it can be opened at https://latex-project.org/bugs/
%% (but please observe conditions on bug reports sent to that address!)
%%
%%
%% Copyright (C) 1993-2022
%% The LaTeX Project and any individual authors listed elsewhere
%% in this file.
%%
%% This file was generated from file(s) of the Standard LaTeX `Tools Bundle'.
%% ---------------------------------------------------------------------------
%%
%% It may be distributed and/or modified under the
%% conditions of the LaTeX Project Public License, either version 1.3c
%% of this license or (at your option) any later version.
%% The latest version of this license is in
%%    https://www.latex-project.org/lppl.txt
%% and version 1.3c or later is part of all distributions of LaTeX
%% version 2005/12/01 or later.
%%
%% This file may only be distributed together with a copy of the LaTeX
%% `Tools Bundle'. You may however distribute the LaTeX `Tools Bundle'
%% without such generated files.
%%
%% The list of all files belonging to the LaTeX `Tools Bundle' is
%% given in the file `manifest.txt'.
%%
\newlinechar=`\^^J
\message{! The file name provided could not be found.^^J%
Use `<enter>' to continue processing,^^J%
`S' to scroll  future errors^^J%
`R' to run without stopping,^^J%
`Q' to run quietly,^^J%
or `X' to terminate TeX}
\errmessage{}
\endinput
%%
%% End of file `h.tex'.
```

Listing 2.12: Assembly cod for the machine timer interrupt handler.s

e code in Listing **??** assumes that interrupts are globally enabled in **mstatus** (MIE=1), that timer interrupts have been enabled in **mie**, and that **mtvec** has been set to the base address of the vector table with the interrupt processing mode set to vectored. The prologue preserves 16 registers according to RISC-V ABI (Application Binary Interface). You may find this a little odd — why waste 16 instructions and 64 bytes in memory to save these registers? Well, it turns out there is a very good reason we do this. When writing an interrupt handler in RISC-V assembly language, it's essential to save and restore the necessary registers to ensure the proper operation of the interrupted program. The specific registers that should be saved onto the stack can vary depending on the RISC-V privilege mode, the interrupt source, and the calling conventions of the platform. However, here's a general guideline for which registers we should consider saving:

a. **ra** register stores the return address for function calls. Saving and restoring this register ensures that control can return correctly to the interrupted program.
b. **Caller-Saved Registers t0-t6** can be freely modified by the caller (interrupted program) without the caller being responsible for saving their original values. If the interrupt handler modifies any of these registers, we should save and restore them to maintain the integrity of the interrupted program.
c. **Stack Pointer** when the interrupt handler needs additional stack space. In such a case, we need to save and restore the stack pointer to ensure that stack operations do not interfere with the interrupted program's stack.
d. **Other Registers Used by the Interrupt Handler**. Depending on the specific needs of the interrupt handler, we may use additional registers for temporary storage or calculations or for passing arguments. If these registers are modified, we should save and restore them.

After the prologue, the handler decodes the exception cause by examining **mcause**: interrupt if **mcause** < 0, trap otherwise. Then, it simply increments the time comparator so that the next timer interrupt occurs about 1000 timer cycles in the future. The handler is not preemptible, as it keeps interrupts disabled throughout the handler. Finally, the epilogue restores saved registers and returns with `mret`.

We can also write interrupt handlers in C. To write an interrupt handler in C for a RISC-V-based system, we typically need to use a combination of assembly language and C code. For example, reading and writing CSRs (e.g., **mcause**) is only possible with the special `csrr`, `csrw` instructions; hence, we are forced to use assembly language for such operations. The exact details of how to implement interrupt handlers in C can vary depending on your platform and compiler, but we will give a general outline of how to write an interrupt handler in C for a RISC-V system:

a. **Mark the Function as an Interrupt Handler**: Usually, we use compiler-specific attributes or pragmas to mark the function as an interrupt handler. This attribute is crucial for the compiler to generate prologue and epilogue sequences for an interrupt handler and to put the `mret` instruction at the end of the generated code. The following C code presents how to mark a function as an interrupt handler:

```
/*
 * Use "interrupt" attribute to indicate that the specified
 * function is an interrupt handler.
 * The compiler generates function entry and exit
 * sequences suitable for use in an interrupt handler
 * when this attribute is present.
*/

__attribute__((interrupt)) void interrupt_handler(void) {
    // Interrupt handling code
}
```

Listing 2.13: Interrupt handler function in C.

b. **Use inline assembly for accessing CSRs:** To read/write the CSRs registers in RISC-V, we should use inline assembly. The exact details of how to use inline assembly depend on the compiler, so we should always consult the compiler manual. Here is an example of how to write inline assembly to read the **mcause** register in C:

```
unsigned int mcause_value;

// Inline assembly to read mcause
asm volatile(
    "csrr %0, mcause"  // Read mcause into %0
    : "=r" (mcause_value)  // Output : mcause_value
);
```

Listing 2.14: Inline assembly to read **mcause**.

The `volatile` qualifier is necessary as GCC optimizers sometimes discard `asm` statements if they determine there is no need for the output variables. Using the `volatile` qualifier disables these optimizations.

Listing 2.15) presents the machine timer interrupt handler.

```
unsigned int *pMTime  = (unsigned int *)0x0200bff8;
unsigned int *pMTimeCmp = (unsigned int *)0x02004000;

__attribute__ ((interrupt)) void mtime__handler (void) {

  unsigneg int mcause_value;
  // Decode interrupt cause:
  // Non memory-mapped CSR registers can only be accessed
  // using special CSR instructions. Hence, we should use
  // inline assembly:
  __asm__ volatile ("csrr %0, mcause"
                     : "=r" (mcause_value) /* output */
                     : /* input : none */
                     : /* clobbers: none */
    );

  if (mcause_value & 0x8000007) { // mtime interrupt!
    // Increment timer compare by 500 ms:
    *pMTimeCmp = *pMTime + 16384;
  }
}
```

Listing 2.15: Machine timer interrupt handler in C.

5. **Enable Global Interrupts:** To enable machine-level interrupts, we should set the MIE bit in the **mstatus** register. The following code (Listing 2.16) presents the RISC-V assembly code to enable global machine-level interrupts :

```

.equ MSTATUS_MIE_BIT_MASK,  0x00000008  # bit 3

#---------------------------------------
#    Enable global interrupts in mstatus
#---------------------------------------
.balign 4
.global enable_global_interrupts
.type enable_global_interrupts, @function
enable_global_interrupts:
    # prologue:
    addi sp, sp, -16     # Allocate the routine
                         #      stack frame
    sw ra, 12(sp)        # Save the return address
    sw fp, 8(sp)         # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16      # Set the framepointer

    li t0, MSTATUS_MIE_BIT_MASK
    csrs mstatus, t0     #  set the MIE bit in mstatus

    # epilogue:
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)         # restore the frame pointer
    lw ra, 12(sp)        # restore the return address
    addi sp, sp, 16      # de-allocate the routine
                         #      stack frame
    ret
```

Listing 2.16: Assembly function for enabling global interrupts in the **mstatus** register.

6. **Enable Particular Interrupt:** Depending on what particular interrupt (software, timer or external) we would like to enable, we should set an appropriate bit in the textbfmie register. Listing 2.17 presents the RISC-V assembly code to enable the machine timer interrupt:

```

.equ MIE_MTIE_BIT_MASK,      0x00000080  # bit 7

#---------------------------------------
#    Enable machine timer interrupt in mie
#---------------------------------------

.balign 4
.global enable_mtimer_interrupt
.type enable_mtimer_interrupt, @function
enable_mtimer_interrupt:
    # prologue:
    addi sp, sp, -16     # Allocate the routine
                         #    stack frame
    sw ra, 12(sp)        # Save the return address
    sw fp, 8(sp)         # Save the frame pointer
    sw s1, 4(sp)
    sw s2, 0(sp)
    addi fp, sp, 16     # Set the framepointer

    li t0, MIE_MTIE_BIT_MASK
    csrs mie, t0        # set MTIE in mie

    # epilogue :
    lw s2, 0(sp)
    lw s1, 4(sp)
    lw fp, 8(sp)        # restore the frame pointer
    lw ra, 12(sp)       # restore the return address
    addi sp, sp, 16     # de-allocate the routine
                        #   stack frame
    ret
```

Listing 2.17: Assembly function for enabling the machine timer interrup in the **mie** register.