Contents

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.

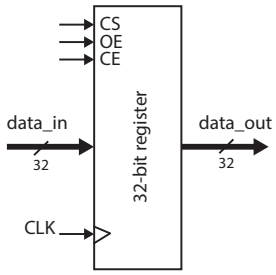

Fig. 1.1: A 32-bit register with the CS, OE and CE signals.

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.

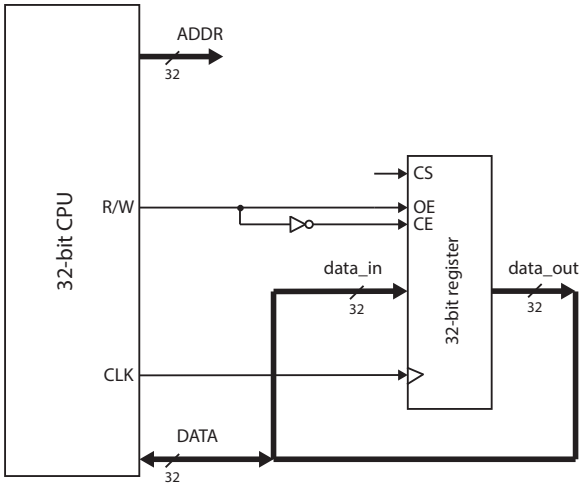

Fig. 1.2: A 32-bit register connected to the data bus and to the CPU's R/W signal.

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

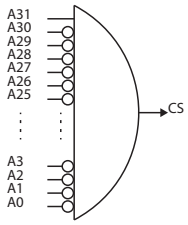

Fig. 1.3: A 32-input AND gate used to decode address 0x80000000.

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

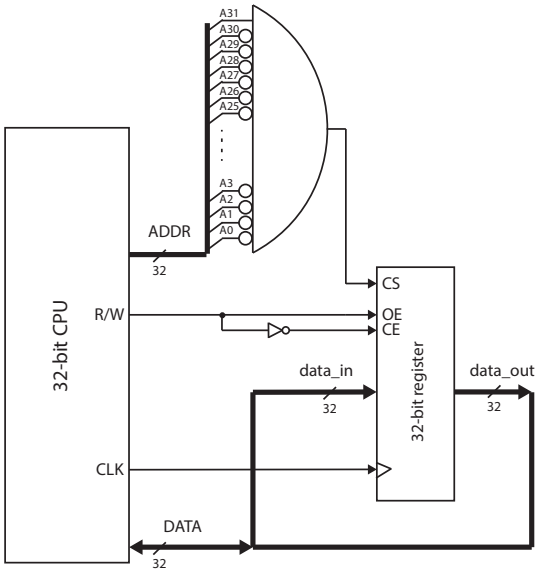

Fig. 1.4: A 32-input AND gate used to decode address 0x80000000.

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

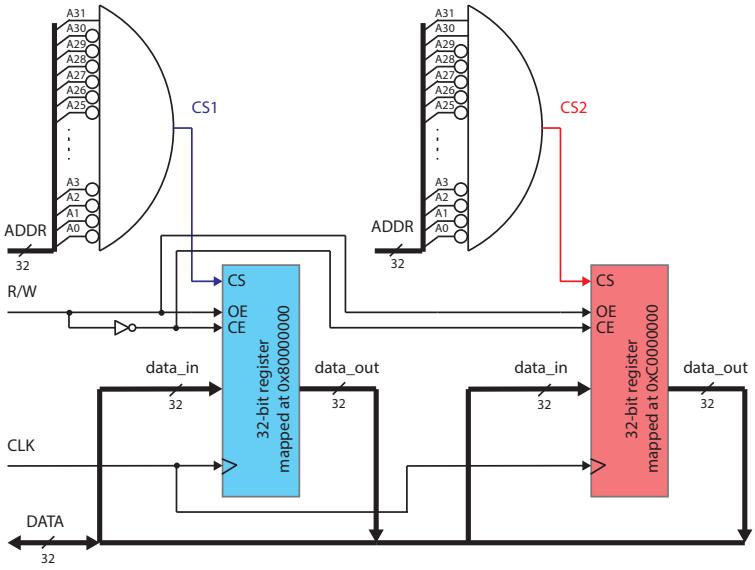

Fig. 1.5: Two memory-mapped registers at addresses 0x80000000 and 0xC0000000, respectively.

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.

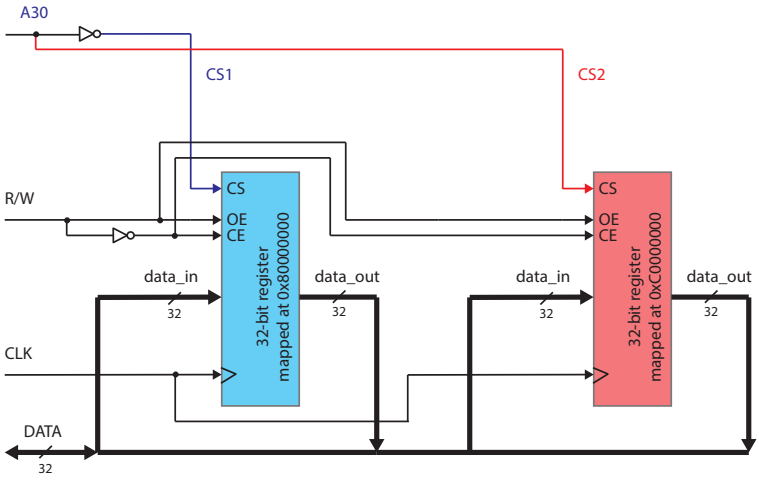

Fig. 1.6: Partial address decoding.

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 1.7 depicts a 3-to-8 address decoder. The decoder has three input lines (A0, A1, and A2), representing

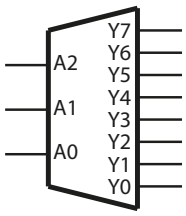

Fig. 1.7: Partial address decoding.

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 1.8 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

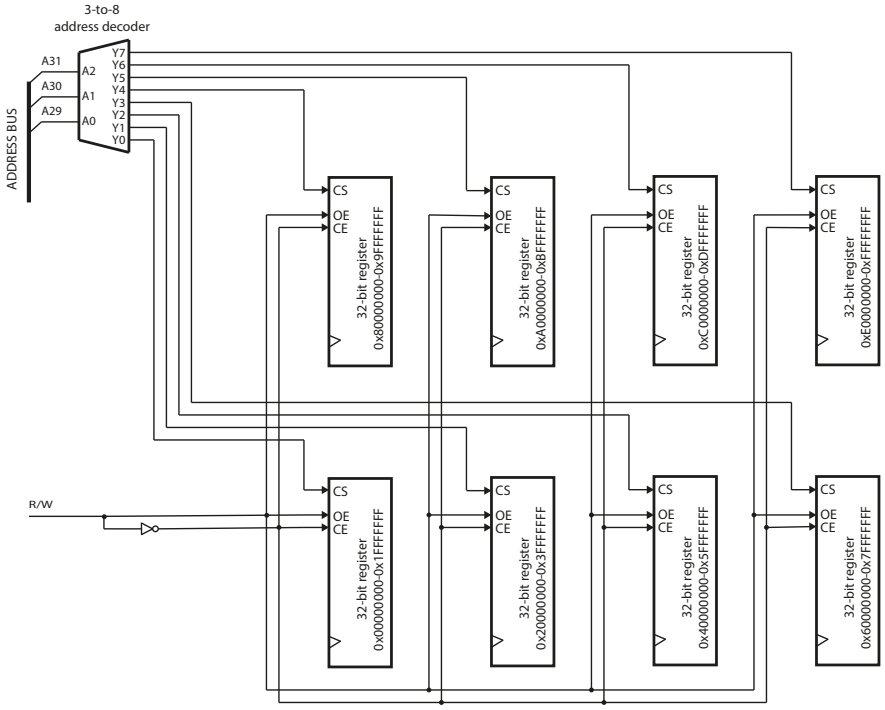

Fig. 1.8: Partial address decoding using a 3-to-8 address decoder to select eight registers.

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.

1.5 Registers mapped at consecutive addresses

In the previous section, we have learned how to memory map a set of registers over the whole memory space using partial address decoding. However, we often aim to map several registers belonging to the same IO device at consecutive memory addresses. Suppose we want to map eight 32-bit registers at the following addresses: 0x80000000, 0x80000004, 0x80000008, 0x8000000C, 0x80000010, 0x80000014, 0x80000018, and 0x8000001C. To decode the registers' addresses, we would use:

1. 2-input AND gates to decode whether the most significant bit A31 is set, and
2. a 3-to-8 address decoder to decode the address bits A4, A3 and A2 and select a particular register.

Figure 1.9 illustrates the solution.

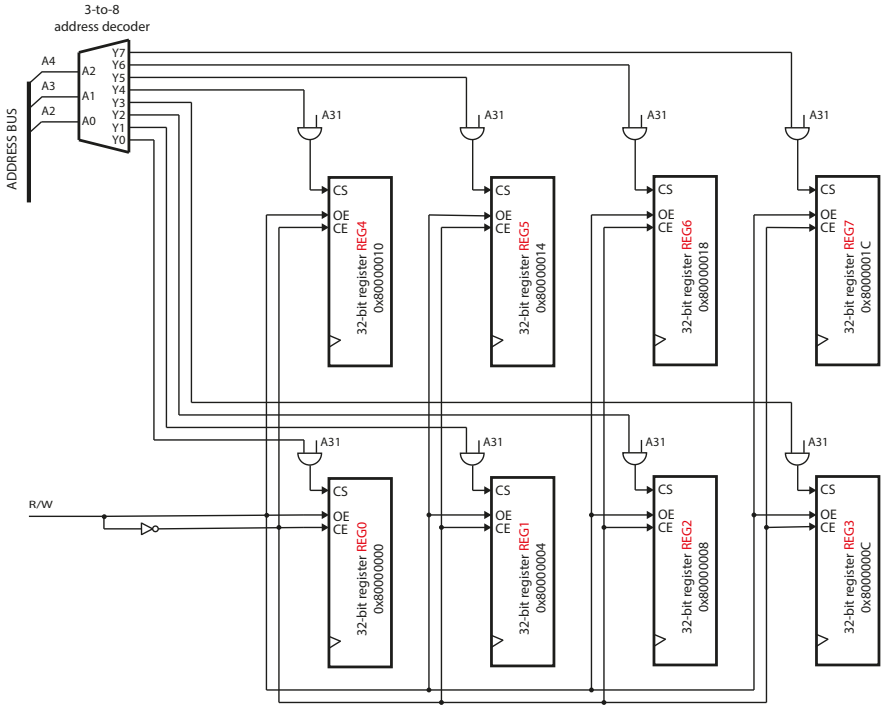

Fig. 1.9: Eight 32-bit registers mapped at consecutive addresses.

There is a positive side-effect of consecutively memory-mapped registers that belong to the same IO device. Using a C structure and pointers, we can conveniently work with the consecutively memory-mapped registers as if they were C structure members, making the IO device's driver code well-organized and readable. This approach is commonly used when working with memory-mapped peripherals and hardware registers in embedded systems and microcontroller programming. To represent consecutively mapped registers using a C structure, we define a C structure where each member corresponds to a specific register at consecutive addresses. Here's an example for the registers from Figure 1.9:

```
#define BASE_ADDRESS 0x80000000

// Define a structure to represent the memory-mapped registers
typedef struct {
```

```
    volatile uint32_t REG0;
    volatile uint32_t REG1;
    volatile uint32_t REG2;
    volatile uint32_t REG3;
    volatile uint32_t REG4;
    volatile uint32_t REG5;
    volatile uint32_t REG6;
    volatile uint32_t REG7;
} Registers_t;

// Define a pointer to the base address of the memory-mapped registers
Registers_t *pMMIORegs = ((Registers_t *)BASE_ADDRESS);

int main() {
    // Access and manipulate the registers:
    pMMIORegs->REG0 = 0x12345678;       // Write to REG0
    pMMIORegs->REG2 |= 0x01 << 13;      // Set bit 13 in REG2
    pMMIORegs->REG7 &= ~(0x01 << 27);   // Clear bit 27 in REG2
    puint32_t value = pMMIORegs->REG6;  // Read from REG6
    ...
    return 0;
}
```

Listing 1.1: Representing and manipulating consecutively memory-mapped registers in C.

In the above code, we define a structure type named textttRegisters_t, where each member represents a specific register at consecutive addresses. Then, we create the pointer `pMMIORegs` to the textttRegisters_t type structure. We assume that the registers are memory-mapped to the base address 0x80000000, and we set this address to the pointer `pMMIORegs`. Finally, as shown in the above example, we can access and manipulate the registers using the `pMMIORegs` pointer and the structure members.

1.6 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 organised 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.

1.7 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. The pipeline comprises five stages: instruction fetch, instruction decode and register

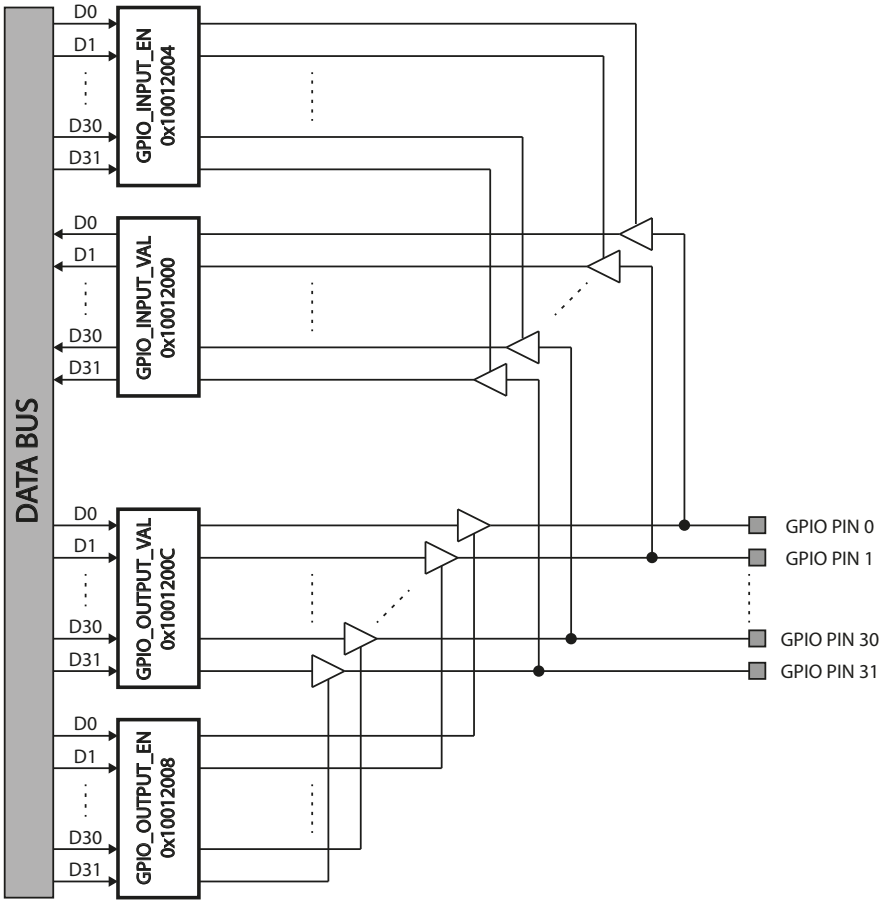

Fig. 1.10: A simplified steructure of the GPIO interface in SiFive Freedom FE310.

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 has 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. The GPIO interface is mapped at address `0x10012000` and comprises 19 data and control registers. To keep the description simple, we will focus only on four data and control registers. The memory map for the selected four GPIO control and data registers is shown in Table 1.1. Each register is 32 bits wide.

Table 1.1: GPIO peripheral register offsets. All registers are reset to 0.

Offset	Name	Description
0x00	**GPIO_INPUT_VAL**	Pin input value
0x04	**GPIO_INPUT_EN**	Pin input enable
0x08	**GPIO_OUTPUT_EN**	Pin output enable
0x0C	**GPIO_OUTPUT_VAL**	Pin output value

Figure 1.10 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.

1.7.1 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 1.2: 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 1.3: 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 1.4: 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 1.5: 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
```

```
25
26      # epilogue:
27      lw s2, 0(sp)
28      lw s1, 4(sp)
29      lw fp, 8(sp)          # restore the frame pointer
30      lw ra, 12(sp)         # restore the return address
31      addi sp, sp, 16       # de-allocate the routine
32                            #   stack frame
33      ret
```

Listing 1.6: Assembly code used to implement the function for toggling a GPIO pin.

1.7.2 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:

```
1 typedef struct
  {
3   volatile int GPIO_INPUT_VAL;
    volatile int GPIO_INPUT_EN;
5   volatile int GPIO_OUTPUT_EN;
    volatile int GPIO_OUTPUT_VAL;
7 } GPIO_Registers_t;
```

Listing 1.7: 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 1.8: 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 1.9: Enabling and setting a GPIO in C.

Instead of using the above `GPIO` pointer to access the GPIO registers directly, we usually define an initialization structure and implement several access functions. This is especially true when we implement the hardware abstraction layer (HAL) of an MMIO device that other users will use. In the hardware abstraction layer, we try to provide more user-friendly way to configure the peripheral without forcing the programmers to know how to configure its registers in detail. For example, to configure and use GPIO, we define several other constants and the `GPIO_InitTypeDef` structure in C:

```
#define GPIO_MODE_INPUT 0x00U
#define GPIO_MODE_OUTPUT 0x01U

/* GPIO pins define
  *
  */
#define GPIO_PIN_0                 ((uint32_t)0x00000001)
#define GPIO_PIN_1                 ((uint32_t)0x00000002)
#define GPIO_PIN_2                 ((uint32_t)0x00000004)
#define GPIO_PIN_3                 ((uint32_t)0x00000008)

...

#define GPIO_PIN_30                ((uint32_t)0x40000000)
#define GPIO_PIN_31                ((uint32_t)0x80000000)

typedef struct
{
  uint32_t Pin;       /* GPIO pins to be configured. */
  uint32_t Mode;      /* Operating mode for the selected pins */
} GPIO_InitTypeDef;
```

Listing 1.10: Pins definition and a C structure used to configure the GPIO.

The meaning of each field of the struct is:

1. `Pin`: it is the position of the pin in a 32-bit word, starting from 0, of the pins we will configure. For example, for pin 22 it assumes the value texttt`GPIO_PIN_22`. Take note that the texttt`GPIO_PIN_x` is a bit mask, where the i-th pin corresponds to the i-th bit of a `uint32_t` datatype. For example, the `GPIO_PIN_9` has a value of `0x00000200` . We can use the same

`GPIO_InitTypeDef` instance to configure several pins at once, doing a bitwise OR (e.g., `GPIO_PIN_1|GPIO_PIN_21|GPIO_PIN_22`).
2. `Mode`: it is the operating mode of the pin, and it can be `GPIO_MODE_INPUT` or `GPIO_MODE_OUTPUT`.

We can now write HAL functions in C that will provide manipulation routines to initialize and change the state of GPIO pins. For example, to initialize and toggle GPIO pins, we implement the following C functions:

```
void HAL_GPIO_Init(GPIO_Registers_t *GPIO, GPIO_InitTypeDef *GPIO_Init)↩
    {

  if (GPIO_Init->Mode == GPIO_MODE_INPUT) {
    GPIO->GPIO_INPUT_EN |= GPIO_Init->Pin;
    GPIO->GPIO_OUTPUT_EN &= ~(GPIO_Init->Pin);
  }

  else if (GPIO_Init->Mode == GPIO_MODE_OUTPUT) {
    GPIO->GPIO_OUTPUT_EN |= GPIO_Init->Pin;
    GPIO->GPIO_INPUT_EN &= ~(GPIO_Init->Pin);
  }
}

void HAL_GPIO_TogglePin(GPIO_Registers_t *GPIO, uint32_t GPIO_Pin){
  GPIO->GPIO_OUTPUT_VAL ^= GPIO_Pin;
}
```

Listing 1.11: Hardware abstraction layer functions for the GPIO.

For example, the `HAL_GPIO_Init` function accepts the GPIO register and the initialization structures. For example, to initialize pins 19, 21 and 22 as outputs, we use the following C code:

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

GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT;
GPIO_InitStruct.Pin = GPIO_PIN_19 | GPIO_PIN_21 | GPIO_PIN_22;
HAL_GPIO_Init(GPIO, &GPIO_InitStruct);
```

Listing 1.12: GPIO pins initialization.

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

Here, we will show how to program another handy memory-mapped IO device, Universal Asynchronous Receiver Transmitter (UART), but we will use only C this time. This is indeed possible for all memory-mapped IO devices, and there is no need to use an assembler. UART is a commonly used serial communication interface that allows asynchronous data transfer between a microcontroller, such as the SiFive FE310, and external devices like sensors, displays, other microcontrollers or

even desktop computers. The SiFive FE310 microcontroller features two memory-mapped UART interfaces that provide serial communication capabilities.

1.8.1 Universal Asynchronous Receiver Transmitter

Before we start explaining the UART provided in SiFive FE310, let us briefly describe the UART interface and its communication protocol. When we want to exchange data between two devices, we generally have two alternatives. Firstly, we can simultaneously transmit all bits **in parallel** using a number of GPIO lines. The number of GPIO lines would be equal to the size of the data word (e.g., eight GPIO lines for a word made of eight bits). Secondly, we can transmit each bit, constituting a data word, one by one **serially**, i.e., in a continuous stream of bits flowing on a single wire. A UART is a device that translates parallel bits in a data word (usually grouped in a byte) into a continuous stream of bits and puts them one by one on a single wire. When the data flows between two devices serially (here, we refer to them as the **sender** and the **receiver**), they have to agree on the timing. Timing defines how long it takes to transmit each individual bit of the data. In **synchronous serial transmission**, the sender and the receiver share a common clock generated by the sender. The clock's frequency determines how fast we can transmit a single bit. But if both devices involved in data transmission agree on how long it takes to transmit a single bit and how to distinguish the start and finish of transmission, then we can avoid using a dedicated clock line. In this case, we have **asynchronous serial transmission**.

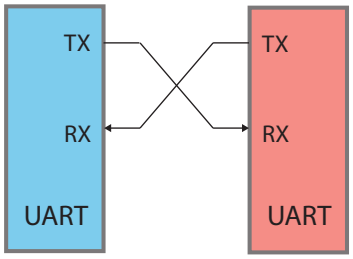

Fig. 1.11: Two UARTs directly communicate with each other.

A Universal Asynchronous Receiver/Transmitter interface is a device able to transmit data word serially using two I/O lines, one acting as a transmitter (TX) and one as a receiver (RX) (Figure 1.11). One of the big advantages of UART is that it is asynchronous – the transmitter and receiver do not share a common clock signal. Although this greatly simplifies the protocol, it does place certain requirements on the transmitter and receiver. Since they do not share a clock, both ends must transmit at the same agreed speed for the same bit timing. Communication in

UART can be simplex (data is sent in one direction only), or full-duplex (both sides can transmit simultaneously).

Data in UART is transmitted in the form of frames. Figure 1.12 shows a UARTS's typical frame and the timing diagram. The high signal on the transmission line represents the idle state (that is, no transmission occurring). Because UART is asynchronous, the transmitter must signal that data bits are coming. This is accomplished by using the start bit. The start bit is a transition from the idle high state to a low state and is immediately followed by eight data bits. The data bits are the user data that come immediately after the start bit. There can be 5 to 9 user data bits, although 8 bits is most common. The least significant bit (LSB) is typically transmitted first. An optional parity bit is then transmitted (for error checking of the data bits). Often, this bit is omitted.

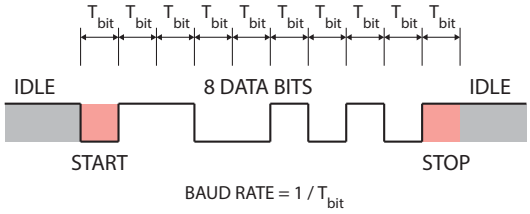

Fig. 1.12: UART frame format.

After the data bits are transmitted, the stop bit indicates the end of user data. The stop bit is either a transition back to the high or idle state or remaining at the high state for an additional bit-time. A second (optional) stop bit can be configured, usually to give the receiver time to get ready for the next frame, but this is uncommon in practice.

The time it takes to transmit a single bit determines the **baud rate**. The baud rate specifies how fast data is sent over a serial line. It's usually expressed in units of bits-per-second (bps). If we invert the baud rate, we can find out just how long it takes to transmit a single bit. This value determines how long the transmitter holds a serial line high/low or at what period the receiver samples its line. Baud rates can be just about any value within reason. The only requirement is that both devices agree upon the same rate. The standard baud rates are 1200, 2400, 4800, 19200, 38400, 57600, and 115200 bits per second.

1.8.2 The UART interface in the SiFive FE310

The UART interface in the SiFive FE310 supports the following features:

1. frames formats: 8 data bits, no parity bit, 1 start bit, 1 or 2 stop bits,

2. 8-entry transmit and receive FIFO buffers with programmable watermark interrupts.

FE310 SoC contains two memory-mapped UART interfaces. The interface UART0 is mapped at address `0x10013000`, while the interface UART1 is mapped at address `0x10023000`. We will focus on UART0 only.

Table 1.2: Register offsets within UART memory map.

Offset	Name	Description
0x00	**txdata**	Transmit data register
0x04	**rxdata**	Receive data register
0x08	**txctrl**	Transmit control register
0x0C	**rxctrl**	Receive control register
0x10	**ie**	UART interrupt enable
0x14	**ip**	UART interrupt pending
0x18	**div**	Baud rate divisor

The UART0 interface in the SiFive FE310 comprises several memory-mapped data and control registers. Table 1.2 presents the memory map for the UART data and control and registers. The UART registers are 32-bit wide, requiring only naturally aligned 32-bit memory accesses.

Here, we will describe only a few registers required to transmit and receive data without using interrupts:

1. **Transmit Data Register (txdata)** (Figure 1.13). Writing to the **txdata** register enqueues the character contained in the data field to the transmit FIFO if the FIFO is able to accept new entries. Reading from **txdata** returns the current value of the FULL flag and zero in the data field. The FULL flag indicates whether the transmit FIFO is able to accept new entries; when set, writes to data are ignored.

Fig. 1.13: The **txdata** register.

2. **Receive Data Register (rxdata)** (Figure 1.14). Reading the **rxdata** register dequeues a character from the receive FIFO and returns the value in the data field. The EMPTY flag indicates if the receive FIFO was empty; when set, the data field does not contain a valid character. Writes to **rxdata** are ignored.
3. **Transmit Control Register (txctrl)** (Figure 1.15). The read-write **txctrl** register controls the operation of the transmitter. The TXEN bit controls whether the transmitter is enabled. When cleared, the transmission is suppressed, and the

Fig. 1.14: The **rxdata** register.

TX pin is driven high. The NSTOP field specifies the number of stop bits: 0 for one stop bit and 1 for two stop bits.

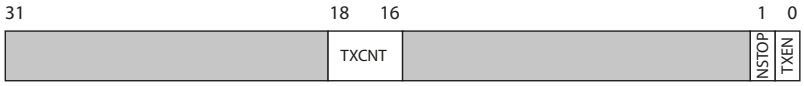

Fig. 1.15: The **txctrl** register.

4. **Receive Control Register (rxctrl)** (Figure 1.16). The read-write **rxctrl** register controls the receiver's operation. The RXEN bit controls whether the receive is enabled. When cleared, the state of the RX pin is ignored.

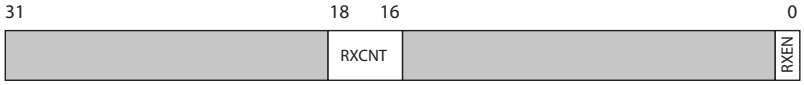

Fig. 1.16: The **rxctrl** register.

5. **Baud Rate Divisor Register (div)** (Figure 1.17). The read-write, **div** register specifies the divisor used by the baud rate generator to divide the CPU's clock frequency to generate a desired baud rate. For example, to set the baud rate of 115200 bits per second, the **div** register should be set to 139. We should refer to the SiFive FE310 documentation and reference manual for precise details on configuring the **div** register.

Fig. 1.17: The **div** register.

1.8.3 Program UART in C

To abstract UART registers with a C structure, we create a structure that mirrors the layout of the UART registers:

```
typedef struct
{
  volatile int UART_TXDATA;
  volatile int UART_RXDATA;
  volatile int UART_TXCTRL;
  volatile int UART_RXCTRL;
  volatile int UART_IE;
  volatile int UART_IP;
  volatile int UART_DIV;
} UART_Registers_t;
```

Listing 1.13: A C structure that mirrors the UART registers layout.

Next, we define a pointer (in our example, the pointer is named UART0, but you are free to use any name you wish) that holds the base address of the UART0 interface:

```
#define UART0_BASEADDR     0x10013000

UART_Registers_t *UART0 = (UART_Registers_t*) UART0_BASEADDR;
```

Listing 1.14: A pointer that holds the base address of the UART interface.

This pointer is used to access the GPIO registers as if they were part of a C structure. Here, we present a few useful UART functions:

```
/*
 * Set Baud Rate to 115200
 * With tlclk at 16Mhz, to achieve 115200 baud,
 *  divisor should be 139. SiFive FE310-G002 Manual:, page 85
 * @arguments:
 *  uart: UART0 or UART1
 */
void uart_set_baud(UART_Registers_t *uart){
  uart->UART_DIV = 139;
}

/*
 * Enable TX
 * @arguments:
 *  uart: UART0 or UART1
 */
void uart_enable_tx(UART_Registers_t *uart){
  uart->UART_TXCTRL |= 0x00000001;
}

/*
 * Set No. stop bits
 * @arguments:
 *  uart: UART0 or UART1
 *  nstop: UART_1_STOP_BIT or UART_2_STOP_BIT
 */
void uart_set_nstop(UART_Registers_t *uart, unsigned int nstop){
```

```
29   if (nstop == UART_1_STOP_BIT) {
       uart->UART_RXCTRL &= 0xfffffffd;
31   }
     else if (nstop == UART_2_STOP_BIT) {
33     uart->UART_RXCTRL |= 0x00000002;
     }
35 }
```

Listing 1.15: Enabling and setting a GPIO in C.

1.8.4 UART pins

Many GPIO pins on the FE310 can serve dual purposes. In addition to their basic input and output capabilities that we presented in Section 1.7, these pins can be controlled by other IO devices in the FE310 SoC. Each GPIO pin can implement up to two so-called IO functions (IOF) enabled with the GPIO_IOF_EN register. Which IOF is used is selected with the GPIO_IOF_SEL register. These alternative functions are often related to various peripherals or communication interfaces available on the microcontroller. IOF allows us to assign alternative functions to GPIO pins, such as enabling them as inputs or outputs for specific peripherals or features like UART. We should refer to the SiFive FE310 datasheet and reference manual for precise information on configuring IOF and alternative functions for GPIO pins on your particular hardware setup. For example, GPIO pin 17 can be used by UART0 transmitter (UART0_TX). Figure 1.18 shows all registers that control the behaviour of the GPIO pin 17. In Section 1.7, we have already explained the purpose of GPIO input, output and enable registers. These registres are depicted in light grey in Figure 1.18. Besides these registers, there are two more registers, GPIO_IOF_SEL and GPIO_IOF_EN. These two registers enable and select an IO function for a particular pin. For example, for the GPIO pin 17, bit 17 in GPIO_IOF_SEL selects an IO function. If this bit is 0, the UART0 transmitter can drive GPIO pin 17. Bit 17 in GPIO_IOF_SEL enables the IO function on pin 17. If bit 17 is set, IOF is enabled for pin 17.

In order to set the UART IO function for GPIO pin 17, we should implement a complete C data structure that mirrors all GPIO registers (refer to SiFive FE310 Manual):

```
1  typedef struct
   {
3    volatile int GPIO_INPUT_VAL;
     volatile int GPIO_INPUT_EN;
5    volatile int GPIO_OUTPUT_EN;
     volatile int GPIO_OUTPUT_VAL;
7    volatile int GPIO_PUE;
     volatile int GPIO_DS;
9    volatile int GPIO_RISE_IE;
     volatile int GPIO_RISE_IP;
11   volatile int GPIO_FALL_IE;
     volatile int GPIO_FALL_IP;
```

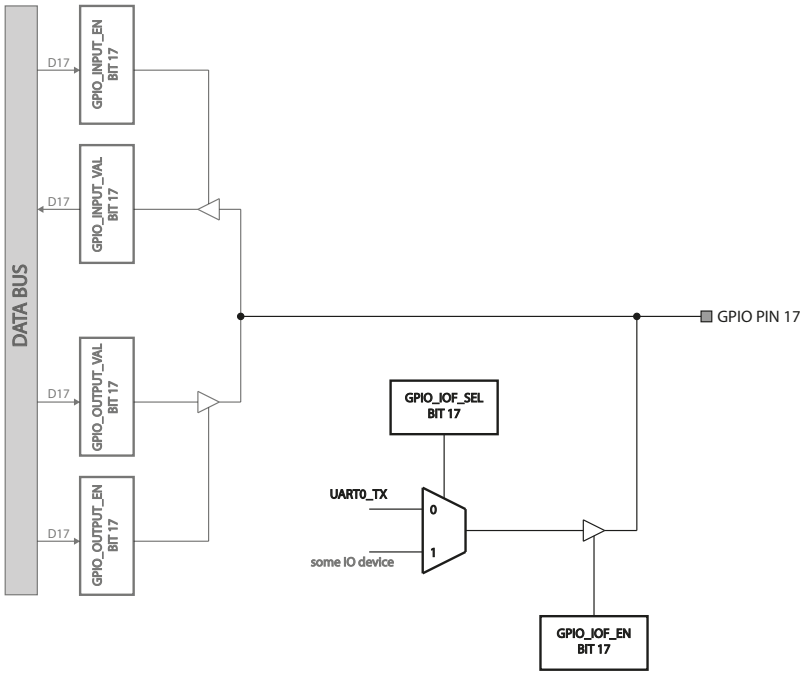

Fig. 1.18: The IO function for GPIO pin 17.

```
  volatile int GPIO_HIGH_IE;
  volatile int GPIO_HIGH_IP;
  volatile int GPIO_LOW_IE;
  volatile int GPIO_LOW_IP;
  volatile int GPIO_IOF_EN;
  volatile int GPIO_IOF_SEL;
  volatile int GPIO_OUT_XOR;
} GPIO_Registers_t;
```

Listing 1.16: A complete C structure for GPIO.

To set up UART0 IOF, we need to configure the GPIO_IOF_EN and GPIO_IOF_SEL registers appropriately. These registers control which alternative functions are enabled for specific GPIO pins. Below is an example of how to configure UART0 IOF for UART TX on GPIO pin 17:

```
GPIO->GPIO_IOF_SEL &= (1 << 17);
GPIO->GPIO_IOF_EN |= (1 << 17);
```

Listing 1.17: A code for setting up UART0 IO function.