Lab 3: Multiprocessors Design

Gao Xufeng, Sun Haoxin

June 3, 2022

1 Introduction

In this lab, we develop a multiprocessor system in FPGA, which contains two parallel NIOS-II cores. To be more specific, we evaluate how the system behaves when there are multiple masters working simultaneously. We first implement separate subsystems, where each CPU works alone and doesn't interact with the other one, then the interaction between them is considered by introducing shared memory and peripherals. Hardware Mutex and Mailbox IPs are included so that these two CPUs can efficiently work together. Finally, we design a hardware counter which can operate in a concurrent environment without the assistance of the Mutex or Mailbox.

This report is organized as follows. Section 2 demonstrates our top-level scheme for this laboratory. Then Section 3 introduces how we design the multiprocessor system with different hardware modules and software programs, and the corresponding measurement results and analysis are provided. Finally in Section 4, we conclude this laboratory.

2 General System Design

In this section, we illustrate our design scheme for the multiprocessor system. The system block diagram of our design is shown as Fig. 1.

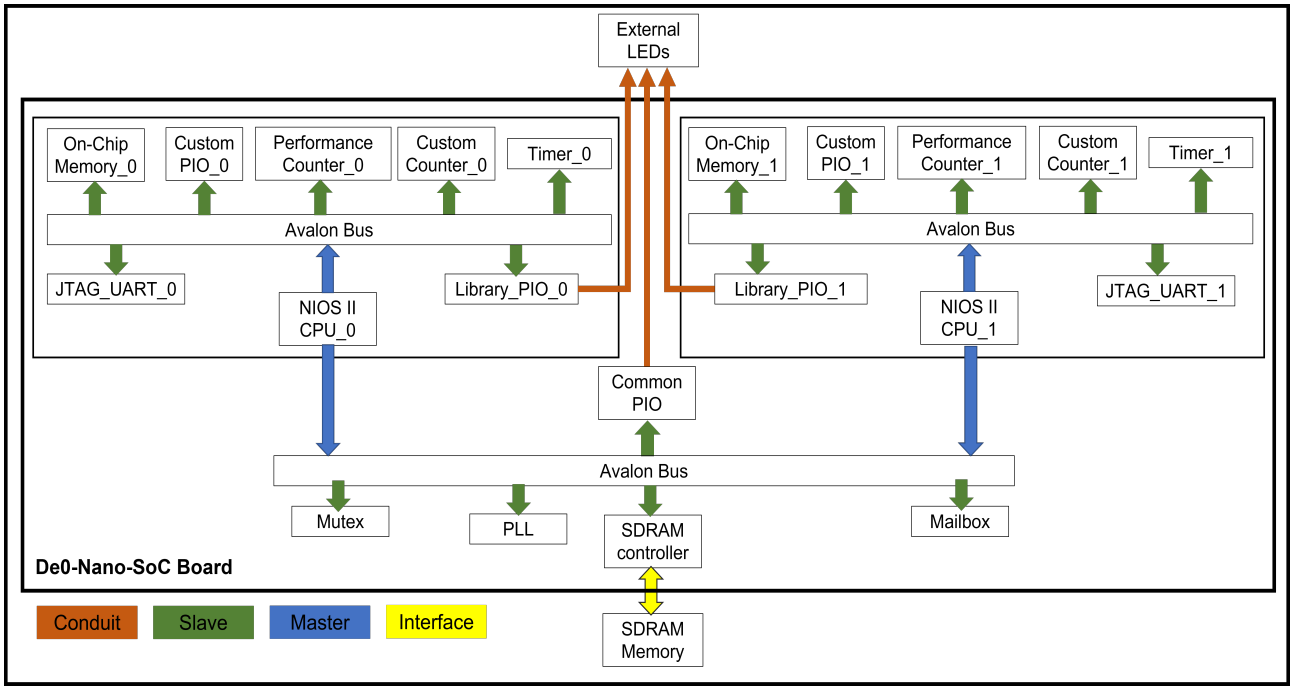

Figure 1: Block diagram of our designed system.

In our design, there are two parallel NIOS-II cores. For each of them, an on-chip memory module is included to store all the software programs; an interval timer is used as the system clock; a PIO connected to the LED on the board is implemented to indicate the working states; a JTAG UART module is also needed to debug and print information on the console; a custom PIO connected to the GPIO can be regarded as a software counter, and a performance counter and a custom counter can help with the timing measurement. Besides these modules which are dedicated to each CPU, we have implemented some shared resources in the system. We include a SDRAM controller and a PLL so that both CPUs can store their data in the off-chip memory. The shared PIO and hardware Mailbox are two peripherals that can be accessed by these two processors. Finally, to coordinate their access to the shared PIO, a hardware Mutex module is designed.

3 Design for Multiprocessor Systems

To speed up a system, it is intuitive to add more processors when the computation can be executed in parallel, as several cores can work concurrently and save a lot of time. Therefore, in this section, we display three different ways to design multiprocessor systems, in which more than one CPU is deployed.

3.1 Isolated working subsystems

In this lab, we first design a system where two processors are operating separately without any interactions. In other words, the CPU only interacts with its exclusive resources like the JTAG, LED, and the custom counter. Since no shared resources are accessed, no concurrency needs to be considered in this situation. Therefore, the system is just a combination of two one-core systems.

To check if each subsystem can work properly, we let processors test the ***printf*** through their own JTAG UARTs. Through the NIOS-II console, we can see that both processors can print the results independently, which confirms the parallelism of the multiprocessor system.

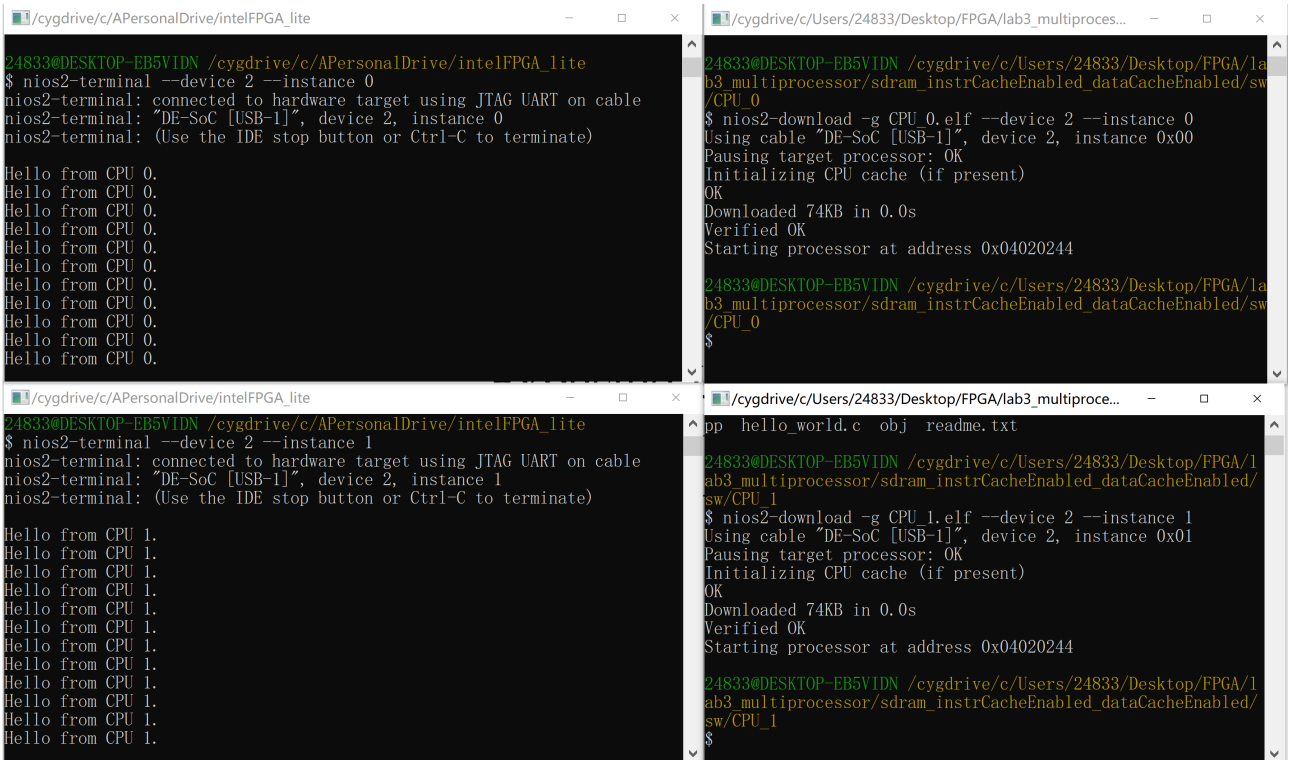

Figure 2: Console results of testing ***printf***.

The corresponding program is shown below:

```c
// Code for test printf function

// Manipulation 1: Prints "Hello World from CPU's ID"
void manip1(){
  while (1){
    printf("Hello from CPU %d.\n", ALT_CPU_CPU_ID_VALUE);
  }
}
```

To measure the timing in this parallel system, in each subsystem, we increase the counter by the software every 50ms, i.e., read the register's value, add 1 on this value, and then write it back to the register. By recording the custom counter's value before and after this operation, we can calculate the duration of this software increment. Then, the snippet of the software C program can be shown as below:

```c
// Code for Isolated working subsystems: CPU_0 as example

// Set up custom IPs: own-PIO (e.g., software counter, LED), counter
void setUp(){
  // reset + start counter
  IOWR_32DIRECT(CPU_0_0_CUSTOM_COUNTER_0_BASE, IRERESET, 0xffffffff);
  IOWR_32DIRECT(CPU_0_0_CUSTOM_COUNTER_0_BASE, IRESTART, 0xffffffff);

  // set LED_0 and shared PIO as output
  IOWR_8DIRECT(CPU_0_0_LED_0_BASE, IREGDIR, 0xff);
  IOWR_8DIRECT(SHARED_PIO_BASE, IREGDIR, 0xff);
}

// read_modify_write PIO counter
void Read_Modify_Write(uint32_t base_addr, uint8_t increment){
  // read PIO
  uint8_t current = IORD_8DIRECT(base_addr, IREGPIN);
  // modify + write
  IOWR_8DIRECT(base_addr, IREGPORT, current+increment);
}

// Increment own software counter every 50ms
void manip2_1(){
  setUp();

  // each processor read_modify_write the LED every 50ms
  while(1){
    uint32_t start = IORD_32DIRECT(CPU_0_0_CUSTOM_COUNTER_0_BASE, IRECOUNTER);
    Read_Modify_Write(CPU_0_0_CUSTOMPIO_0_BASE, 1);
    uint32_t end = IORD_32DIRECT(CPU_0_0_CUSTOM_COUNTER_0_BASE, IRECOUNTER);
    printf("Access time to LED: %ld cycles\n", end-start);
    usleep(50000);  // 50ms
  }
}
```

The relevant access timing records of each processor are as follows:

Table 1: Time overheads of accessing to PIO counters owned by CPU itself

CPU ID	Total Overheads (in clock cycles)
CPU0	89
CPU1	89

As Table 1 shows, in the context of 50MHz clock, the difference in time overhead required by CPU_0 and CPU_1 is 0 cycles. Thus, we can conclude that both processors can access their own PIO counters independently without conflicts.

3.2 Cooperative System with the Mutex and Mailbox

In a real multiprocessor application, processors need to communicate and cooperate with each other. This can be achieved by sharing resources in the system. However, accessing the same resources simultaneously by several CPUs may cause problems. Thus, it is important to have some mechanisms to coordinate these cores. In this subsection, we develop a system in which the two CPUs can access the shared memory and PIO, with the assistance of the Mutex and Mailbox tools.

3.2.1 Hardware Mutex

Indeed, without these hardware locks, access to the shared counter will leads to wrong values, like Fig. 3. Here the shared counter has an initial value 0. In the time slot, CPU0 increases it to 1. Then in the second slot, both CPUs read the current value 1, and are going to increase it to 2. However, CPU0 sleeps for a long time after reading, which is common in many operating systems. During this period, CPU1 keeps working and increases the counter to 3. When CPU0 wakes up in the fourth slot, it writes 2 back to the counter. As a result, the shared counter is increased 4 times, but its value is 2 and wrong. Therefore, in order to avoid such situations, locks like the Mutex are indispensable in the system.

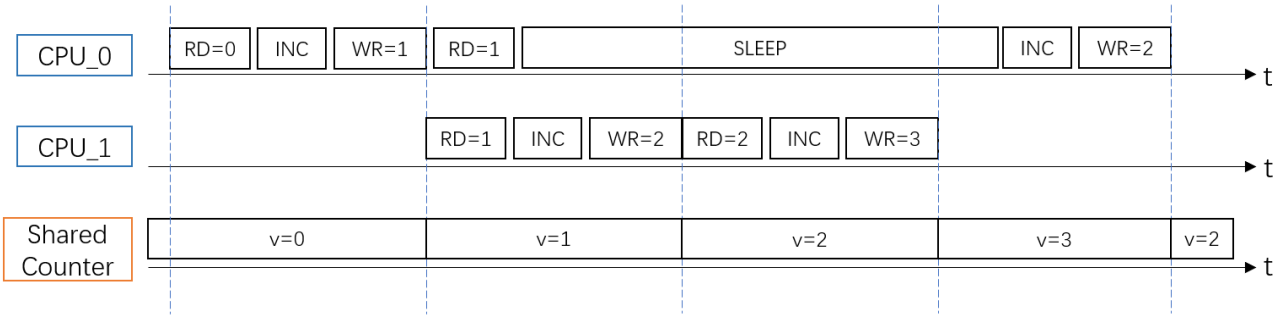

Figure 3: An operation leading to wrong values in the counter.

In our design, we use the hardware Mutex IP to protect the shared PIO. The program and results are as follows.

```
// Manipulation 2.2: Access common PIO with Hardware Mutex, CPU_0 as example

// exclusive read_modify_write PIO counter
void Ex_Read_Modify_Write(uint32_t base_addr, uint8_t increment, alt_mutex_dev* mutex){
  // acquire the mutex, setting the value to 1
  altera_avalon_mutex_lock(mutex, 1);
  Read_Modify_Write(base_addr, increment);
  altera_avalon_mutex_unlock(mutex);
}
```

```
void manip2_2(){
  setUp();
  // get the mutex device handle
  alt_mutex_dev* cpu_0_mutex = altera_avalon_mutex_open(SHARED_MUTEX_PIO_NAME);

  // processor 0 exclusive read_modify_write the LED every 20ms
  while(1){
    uint32_t start = IORD_32DIRECT(CPU_0_0_CUSTOM_COUNTER_0_BASE, IRECOUNTER);
    Ex_Read_Modify_Write(CPU_0_0_CUSTOMPIO_0_BASE, 1, cpu_0_mutex);
    uint32_t end = IORD_32DIRECT(CPU_0_0_CUSTOM_COUNTER_0_BASE, IRECOUNTER);
    printf("Exclusive access time from CPU0 to shared PIO: %ld cycles\n", end-start);
    usleep(20000); // 20ms
  }
}
```

Table 2: Time overheads of accessing to the common PIO counter for processors

CPU ID	Total Overheads (in clock cycles)
CPU0	372
CPU1	471

From Table 2, we can see that access to the PIO counter requires more cycles than before due to the mutual exclusion. This is to be expected, because mutex uses hardware-based atomic test-and-set operations to determine which CPU can get the ownership of the common PIO counter. Thus, more resources are consumed in this case. By checking the difference with the previous results shown in Table 1, the additional overheads required by hardware mutex can be computed as:

- Around 283 cycles on CPU0
- Around 382 cycles on CPU1

To further evaluate the timing performance of this hardware Mutex tool, we measure the switching time, i.e., the interval between one CPU releases the lock (calls the ***unlock*** function) and the other which waits for the lock gets it (enters the ***lock*** function). In the optimal situation, this overhead should be close to 0.

We use the shared PIO and the logic analyzer to measure this parameter. The idea is that each CPU pulls up an output pin when it gets the lock, and pulls down before it releases the lock. Therefore, by comparing the gap between one PIO is 0 and the other is 1, we can get this transition timing parameter. The measured time overheads of switching Mutex ownership are recorded in Table. 3.

Table 3: Time overheads of switching Mutex ownership between CPUs

Mutex Switch	Total Overheads (in μs)
From CPU0 to CPU1	4.616
From CPU1 to CPU0	5.432

It can be seen that the switching time is smaller for CPU0 to CPU1 than for CPU1 to CPU0, which is a same relationship displayed in Table 2.

3.2.2 Hardware Mailbox

In our design, two processors named CPU0 and CPU1 are employed. To enable the communication between processors, a hardware mailbox core with an Avalon interface is introduced. Fig. 4 gives the implementation of our mailbox system.

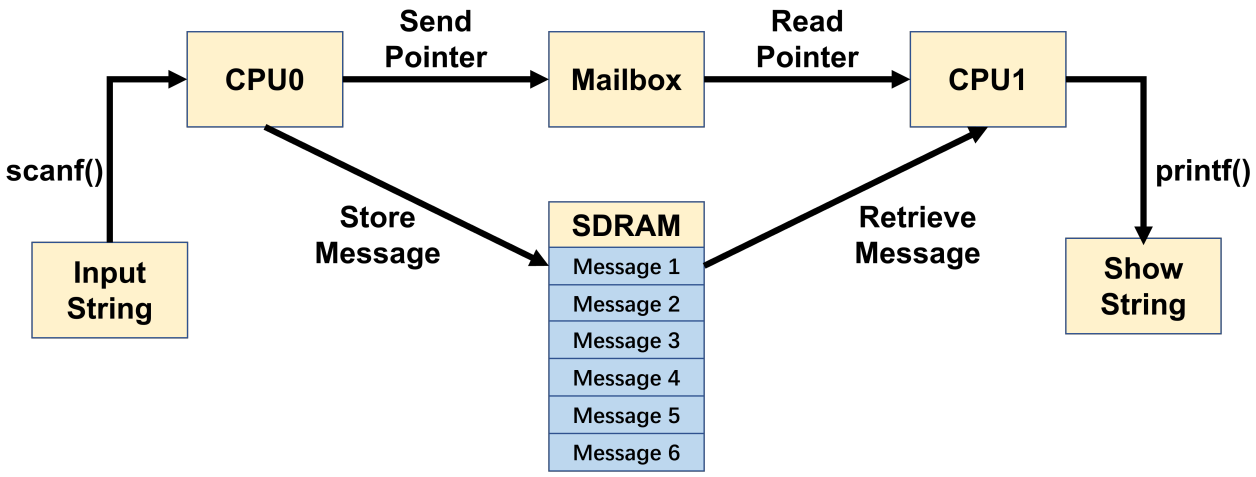

Figure 4: Implementation of mailbox system.

As the figure depicts, CPU0 acts as a message sender, which can read a string from standard inputs (e.g., ***scanf*** in C) and store it in the shared SDRAM memory. To avoid memory inconsistency, the data cache is then flushed. Instead of passing raw data, the address of the message is sent to the Mailbox. For receiver processor CPU1. it keeps polling the Mailbox to determine if any message has arrived. When there is a message pending in the Mailbox, CPU1 reads it, then flushes its cache and retrieves the string through the message pointer. Finally, the string is printed through standard outputs (e.g., ***printf*** in C). It should be noted that the messages should be stored consecutively in the SDRAM by CPU0 to avoid being overwritten. The corresponding C program is shown below:

```c
// Manipulation 3: Hardware Mailbox
// CPU_0 as sender
void manip3_CPU_0()
{
  alt_u32 tx_msg[2] = {0x00001111, 0x0};
  char *msg_pt = SDRAM_CONTROLLER_BASE;
  altera_avalon_mailbox_dev* mailbox_sender;
  mailbox_sender = altera_avalon_mailbox_open(SHARED_MAILBOX_NAME, tx_cb, NULL);
  if(!mailbox_sender){
          printf ("FAIL: Unable to open mailbox_simple");
  }
  printf("Input Message: ");
  scanf("%s", msg_pt);
  tx_msg[1] = (alt_u32) msg_pt;
  alt_dcache_flush_all(); // flush the cache
  // allow CPU_0 to transmit messages
  alt_u32 status = altera_avalon_mailbox_send(mailbox_sender, tx_msg, 0, POLL);
  if(status){
    printf("error in transfer: \n");
  }else{
    printf("\nTransfer Start\n");
    printf("Message Start Address: 0x%x\n", tx_msg[1]);
    printf("Transmit Message Content: %s", msg_pt);
    printf("\nTransfer done!\n");
  }
```

```
  altera_avalon_mailbox_close(mailbox_sender);
}

// CPU_1 as receiver
void manip3_CPU_1()
{
  alt_u32 rx_msg[2];
  altera_avalon_mailbox_dev* mailbox_receiver;
  mailbox_receiver = altera_avalon_mailbox_open(SHARED_MAILBOX_NAME, NULL, rx_cb);
  if(!mailbox_receiver){
          printf ("FAIL: Unable to open mailbox_simple");
  }
        // allow CPU_1 to receive messages
  altera_avalon_mailbox_retrieve_poll(mailbox_receiver, rx_msg, 0);
  if(rx_msg == NULL){
    printf("Receive Error or No message");
  }else{
    alt_dcache_flush_all(); // flush the cache
    char *msg_pt = (void*) rx_msg[1];
    printf("Receiver Start:\n");
    printf("Message Start Address: 0x%x\n", rx_msg[1]);
    printf("Received Message Content: %s\n", msg_pt);
    printf("\nReceiver done!\n");
  }
  altera_avalon_mailbox_close(mailbox_receiver);
}
```

Two messages are provided, and following results have demonstrated that our mailbox system is working well.

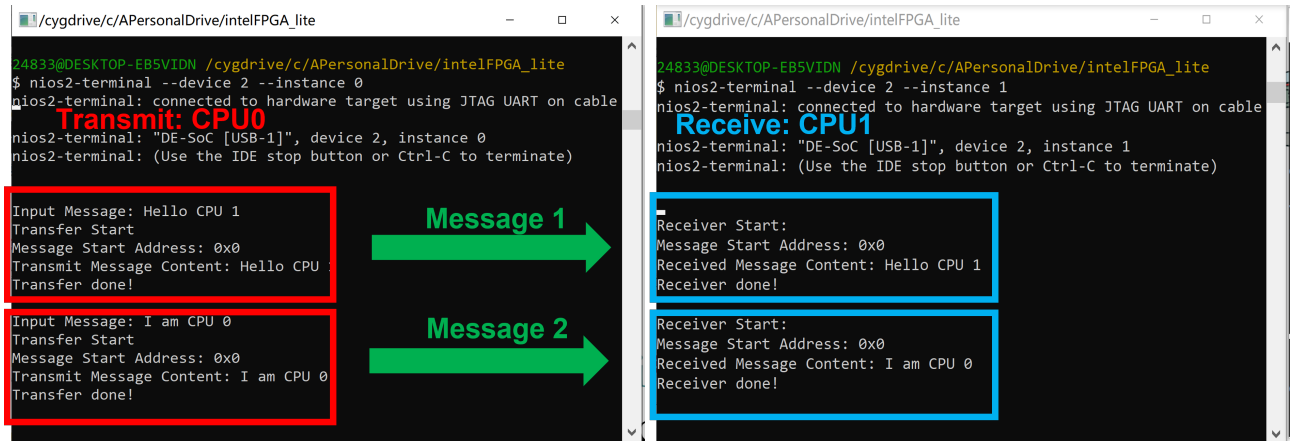

Figure 5: Console results of Mailbox.

3.3 Hardware Counter Without Locks

As discussed in previous parts, locks are necessary for the usage of a shared software counter. Therefore, in order to realize a counter without any locks, we design a hardware counter in VHDL, which can be accessed by multiprocessors too.

The idea is that now CPUs should not have to apply reading, increment, and writing successively to increase the counter, which is not safe for shared peripherals. In the contrast, their operations on this shared counter should be "atomic", i.e., cannot be interrupted by other operations. The Avalon bus's arbitration mechanism can help us with this problem. When there are multiple masters on the bus to access one slave at the same time, the Avalon bus itself will decide the sequence of access, as shown in Fig. 6. In this figure, CPU0 executes 3 increments, CPU1 executes 3 increments and 1 decrement. Although some of these operations are concurrent, the Avalon bus will automatically decide the sequence of operations to access the shared counter. Thus, the correctness of this counter can be protected.

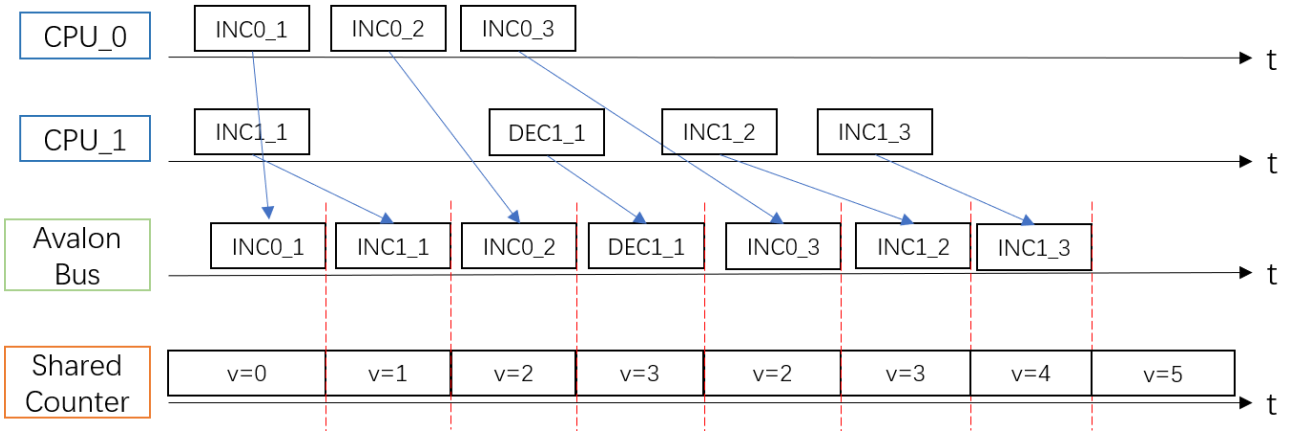

Figure 6: Operations of this hardware counter with Avalon bus's arbitration.

Based on this analysis, a new register called R_{INC} with address offset 7 is introduced in our custom PIO IP. As the below VHDL code snippet shows, when writing to this register, the value stored in the parallel port is incremented by the written value (i.e., 1 in our case).

```
-- VHDL: Register map in custom PIO IP
if Write = '1' then
    case Address is
        when "000" => iRegDir    <= WriteData(7 downto 0);
        when "010" => iRegPort   <= WriteData(7 downto 0);
        when "011" => iRegPort   <= iRegPort or WriteData(7 downto 0); -- iRegSet
        when "100" => iRegPort   <= iRegPort and not WriteData(7 downto 0); -- iRegClr
        when "101" => iRegClrInt  <= WriteData(7 downto 0);
        when "110" => iRegEnInt   <= WriteData(7 downto 0);

        -- Register R_INC to implement Hardware Counter without Locks
        when "111" => iRegPort <= std_logic_vector(unsigned(iRegPort) +
        ↪  unsigned(WriteData(7 downto 0)));

        when others => null;
    end case;
end if;
```

The corresponding C program is as follows:

```
// Code for simple write of increment/initialize value: CPU_0 as example
void simple_modify(uint32_t base_addr, uint8_t increment){
  // simple modify + write
```

```
  IOWR_8DIRECT(base_addr, IREGPORT_INC, increment);
}

// Manipulation 4: Hardware Counter
void manip4()
{
  setUp();
  // processor 0 simple-increment the shared PIO counter
  while(1){
    uint32_t start = IORD_32DIRECT(CPU_0_0_CUSTOM_COUNTER_0_BASE, IRECOUNTER);
    simple_modify(SHARED_PIO_BASE, 1);
    uint32_t end = IORD_32DIRECT(CPU_0_0_CUSTOM_COUNTER_0_BASE, IRECOUNTER);
    printf("Access time to shared PIO: %ld cycles\n", end-start);
    usleep(20000); // 20ms
  }

}
```

Compared with the original custom counter, once receiving a signal from CPUs, this shared counter can increase or decrease its value by itself. Therefore, CPUs can do increment or decrement in only one *IOWR* instruction, protecting the atomicity. Table 4 gives the measured results.

Table 4: Time overheads of accessing to the hardware counter for processors

CPU ID	Total Overheads (in clock cycles)
CPU0	47
CPU1	65

Compared to Table 2, we can observe that the time overheads are reduced to around 47 and 65 cycles, respectively. With such a hardware counter, the CPUs can remove the mutex, but the data consistency can still be guaranteed and the processing speeds are faster than the ones we measured in the case of using mutex.

4 Conclusions

In this lab, we implement a multiprocessor system including two parallel NIOS-II cores. We first test this system with separate counters and peripherals. Then a shared counter and PIO are included with the assistance of Mutex and Mailbox tools. Finally, we design a hardware counter which can be accessed by multiple CPUs and do increments and decrements without the Mutex or Mailbox. To analyze their performance, we measure the time consumption in each scheme. By making comparisons, some insights about this multiprocessor system are derived.