Multiprocessors

Pinar Ezgi Çöl & Darko Lukić

May 2019

Contents

1 Introduction

In this laboratory, we design and implement a system which has a multiprocessor architecture. Different experiments are conducted with several hardware component which include Parallel Port, Mutex, Mailbox and Counter. The rest of the report is organized as follows: First, we are demonstrating the system design. In the second part, we provide the results of the experiments with the hardware components. In the final section, we conclude this report.

2 System Design

The system is composed of two separate processors which we named CPU0 and CPU1. They are connected to a common Avalon Bus and each of them has the hardware components that are specific to each one. The details of the system design can be observed in Fig. 1.

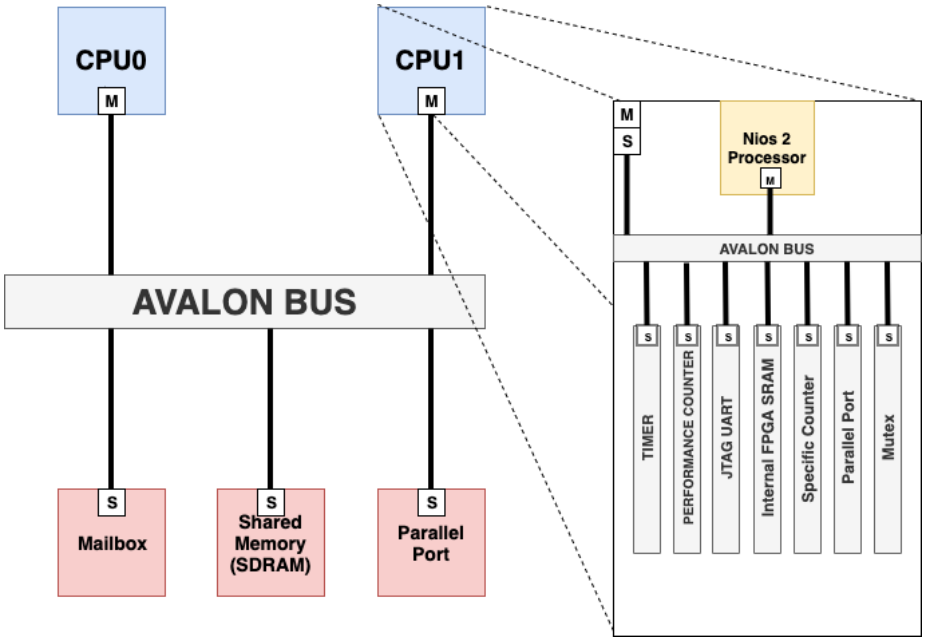

Figure 1: System Design

3 Experiments

We have performed multiple experiments to evaluate different techniques for data sharing and synchronisation between two processors. In the purpose of memory sharing SDRAM is utilized and connected to both processors. By doing it both processors can access to the same memory address, read and write.

General problem for this kind of systems is "data race" when both processors try to edit the same memory locations.

3.1 STDOUT

In order to print from both of processors we decided to use only JTAG UART peripheral. One processor (CPU0) is directly connected to JTAG UART while the other one (CPU1) puts messages on SDRAM. CPU0 is configured to read specific segment of SDRAM when message is written and to print it using JTAG UART. The whole synchronisation logic is done on CPU0.

Another possible approach is to use UART peripheral with CPU1. We could export TX pin in a GPIO bank and connect it to external UART receiver.

3.2 Parallel Port Test

Afterwards, the parallel port is added to SoC design and connected to LEDs. The objective of this experiment was to read value written in parallel port, increment by 1 and write it again to parallel port. The same operations had to be performed on both CPUs.

In order to check if there is a data race we programmed both processors to increment value on the parallel port in 127 iterations. Ideally, we should have value 254 on parallel port. However, we would almost always getting less.

The reason for the observation is a data race. Eg. CPU0 read value 10, CPU1 read value 10, CPU0 increment and write value 11, CPU1 do the same. Instead of incrementing the value for 2, the value is incremented by 1.

3.3 Hardware Mutex

Mutex can be used to avoid issues presented in the previous section. Since we have hardware mutex shared on our SoC we have used it for exclusive access of parallel port between two processors.

Mutex behaves as a gate keeping only one processor to access to parallel port. When the mutex locked the processor cannot lock the mutex until the mutex unlock.

Mutexes can be used to solve synchronisation problems but there are costly. Performing the increment operation with mutexes increases required number of cycles by **1680** (around 300 without mutex and around 1980 with mutex).

3.4 Hardware Mailbox

Mailboxes are another tools to achieve synchronisation between processors. Further, we implemented Mailbox peripheral and software for message exchange between processor. CPU1 creates a message on SDRAM and sends a pointer to Mailbox, CPU1 gets notification with a points, reads a message on SDRAM at given memory location and prints it. The detailed implementation is shown in Fig. 2.

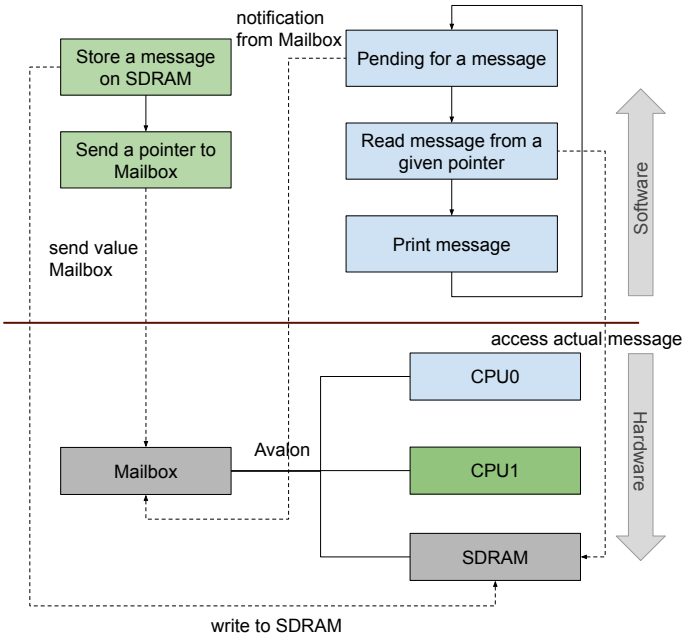

Figure 2: Schematic view of Mailbox hardware and software implementation

In Fig. 2 boxes in green are related to CPU1, boxes in blue to CPU0 and shared resources are in gray. Dashed lines represent a logical connections between components. The more detailed software implementation is available in Appendix.

3.5 Hardware Counter

In this experiment counter is implemented as a peripheral. The counter is designed so the value can be incremented by a given value. Both processor have access to the specific counter. More details are give in Fig. 3 and registry map is described in Table 1.

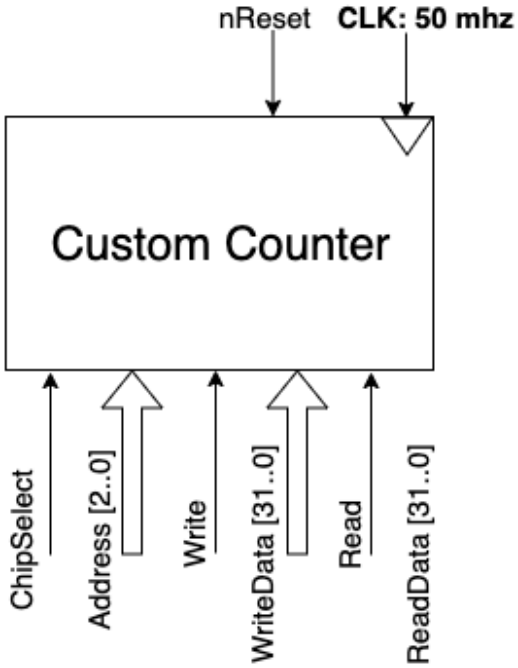

Figure 3: Counter Block Diagram

Reg. Addr.	**Name**	**R/W**	**Size**	**Function**
0	iCounter	R	32	Value of the Counter
1	iRz	W	1	Counter Reset
1	incVal	R	32	Value of the increment amount of the counter
2	iCounter	W	32	Initialize the counter with a value
3	incVal	W	32	Initialize the increment amount of the counter
4	iCounter	W	32	Increment the counter with incVal
5	iCounter	W	32	Decrement the counter with incVal

Table 1: Register Map of the Counter

Since incrementing a value of counter doesn't require multiple operation (read, alternate and write) mutexes are not required. Multiple processors cannot write wrong alternated value but only to send command to the counter increment a value.

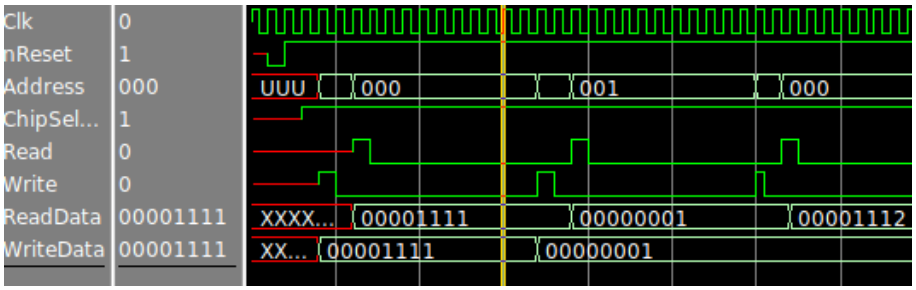

Figure 4: Testbench for the specific counter

4 Conclusion

Our goal was to investigate multiple methods for resource sharing and synchronisation. We have encountered the problem of accessing to the same resource (parallel port) from multiple processors. Therefore, we used hardware mutex, mailbox and specific peripheral to resolve the issue. Every solution comes with certain trade-offs and now we have a better idea what to use in different use-cases.

5 Appendix

5.1 Mutex

```
int main(void) {
    alt_mutex_dev* mutex = altera_avalon_mutex_open("/dev/mutex_0");
  if (! mutex) {
    printf("Mutex error!\n");
  }

  while (1) {
    altera_avalon_mutex_lock(mutex, 1);
    int counter = IORD_ALTERA_AVALON_PIO_DATA(PIO_0_BASE);

    counter--;

    IOWR_ALTERA_AVALON_PIO_SET_BITS(PIO_0_BASE, counter);
    IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(PIO_0_BASE, ~counter);
    altera_avalon_mutex_unlock(mutex);

    for (int i = 0; i < 100000; i++);
  }
}
```

5.2 Mailbox

5.2.1 CPU0

```
/*
Actual output:
  Message from mailbox with message 1
  Message from mailbox with message 2
  Message from mailbox with message 3
  Message from mailbox with message 4
  Message from mailbox with message 5
*/

static void mailbox_rx(void *message, int status) {
  printf("Message from mailbox with message %d \n", status, *((alt_u32 *)message));
}

int main(void) {
  altera_avalon_mailbox_dev* recv_dev;
  recv_dev = altera_avalon_mailbox_open("/dev/mailbox_simple_0", NULL, mailbox_rx);
  if (! recv_dev) {
    printf("Mailbox error!\n");
  }

  while(1);
}
```

5.2.2 CPU1

```
int main(void) {
    int counter = 0;
  altera_avalon_mailbox_dev* send_dev;
  send_dev = altera_avalon_mailbox_open("/dev/mailbox_simple_0", mailbox_tx, NULL);

  IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(PIO_0_BASE, 7);

  while (1) {
    alt_u32 message = counter++;
    altera_avalon_mailbox_send(send_dev, &message, 0, POLL);
    for (int i = 0; i < 1000000; i++);
  }
}
```

5.3 Counter

```
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.numeric_std.all;

entity Counter is
  port (
    Clk : in std_logic;
    nReset : in std_logic;
    Address : in std_logic_vector(2 downto 0);
    ChipSelect : in std_logic;
    Read : in std_logic;
    Write : in std_logic;
    ReadData : out std_logic_vector(31 downto 0);
    WriteData : in std_logic_vector(31 downto 0)
  );
end Counter;

architecture comp of Counter is
  signal iCounter : unsigned(31 downto 0);
  signal incVal : unsigned(31 downto 0);

begin
  pRegWr : process(Clk, nReset) begin
    if nReset = '0' then
      incVal <= (others => '0');
      iCounter <= (others => '0');
    elsif rising_edge(Clk) then
      if ChipSelect = '1' and Write = '1' then -- Write cycle
        case Address(2 downto 0) is
          when "010" => iCounter <= unsigned(WriteData);
          when "011" => incVal <= unsigned(WriteData);
          when "100" => iCounter <= iCounter + incVal;
          when "101" => iCounter <= iCounter - incVal;
          when others => null;
        end case;
      end if;
    end if;
  end process pRegWr;


  pRegRd : process(Clk) begin
    if rising_edge(Clk) then
      ReadData <= (others => '0');
      if ChipSelect = '1' and Read = '1' then -- Read cycle
        case Address(2 downto 0) is
          when "000" => ReadData <= std_logic_vector(iCounter);
          when "001" => ReadData <= std_logic_vector(incVal);
          when others => null;
```

```
        end case;
      end if;
    end if;
  end process pRegRd;

end comp;
```

5.4 Full source code

https://gitlab.com/lukicdarkoo/multiprocessor