Lab 1: Interrupt Analysis

Gao Xufeng, Sun Haoxin

1. Introduction

In this lab, a FPGA-based system is designed to generate interrupt and to measure several interrupt parameters on the NIOS-2 system. In general, in the first part, we first design a custom PIO and a custom counter to generate interrupt and help measure time intervals. Then, we develop various ways to measure the response time, recovery time and latency time of the interrupt in the CPU. Besides, we also consider the effect of memory settings in this lab, so that we measure these parameters with on-chip memory or SDRAM, as well as with/without data cache and instruction cache. Then in the second part, we design a system with MircoC/OS-II to measure the timing parameters of the semaphore, flags, mailbox and queue.

This report is organized as follows: Section 2 demonstrates our hardware design, e.g., the custom IPs. Section 3 displays our schemes and results in the measurements of three interrupt parameters. Section 4 shows how we measure the parameters in a system with RTOS. In Section 5, we conclude this report.

2. Hardware Design

The entire system consists of the following components: a NIOS-2 CPU core, a JTAG UART, an on-chip Memory, a SDRAM Controller (connected to the SDRAM on the board), a PLL, two interval timers (one used as the system clock for MicroC/OS-II, one used for counting clock cycles), and two custom IPs: PIO and counter, which is shown as Fig. 1.

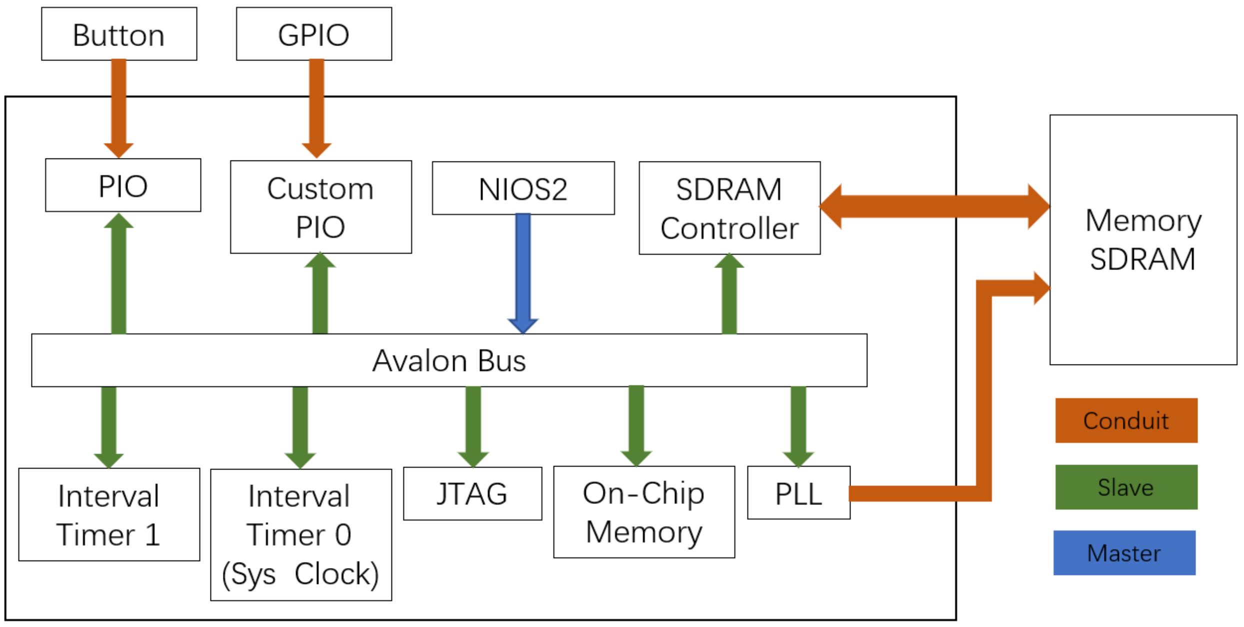


Figure 1. System block for the measurements.

In the rest of this part, we will demonstrate how we design the custom PIO and custom counter in detail, as well as the usage of the SDRAM controller and the PLL.

2.1 Custom PIO

In this lab, a custom PIO is designed to investigate how the CPU responds to the interrupt-request inputs asserted by external sources, such as I/O devices. The register map of this IP is shown as:

Address [2:0]	Write Registers	DataWrite [7:0]	Read Registers	DataRead [7:0]
000	RegDir	iRegDir	RegDir	iRegDir
001	---	Don't care	RegPin	iRegPin
010	RegPort	iRegPort	RegPort	iRegPort
011	RegSetPort	iRegPort	---	0x00
100	RegClrPort	iRegPort	---	0x00
101	RegClrInt	iRegClrInt	---	0x00
110	RegEnInt	iRegEnInt	RegEnInt	iRegEnInt
111	---	Don't care	RegInt	iRegInt

Table 1: Register map for custom PIO

And the block diagram of this IP is drawn as:

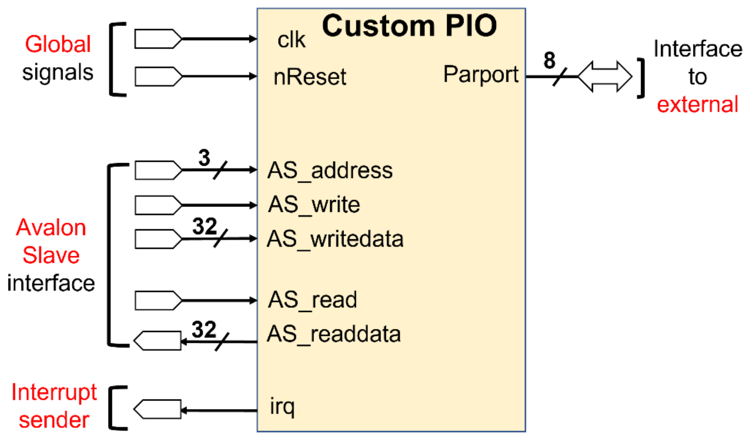


Figure 2: The block diagram of custom PIO

2.2 Custom Counter

A custom counter is designed to assist the measurement of time interval. The main idea is that we hope to use the software program to control its start, stop and reset, and then read its value when needed. Thus, this custom IP is a slave on the Avalon Bus, whose register can be read and written by the CPU. The register map of this IP is shown as:

Address [1:0]	Write Registers	DataWrite [7:0]	Read Registers	DataRead [7:0]
00	RegReset	iRegReset	---	0x00
01	RegSetStart	iRegSetStart	---	0x00
10	RegClrStart	iRegClrStart	---	0x00
11	---	Don't care	RegCounter	iRegCounter

Table 2: Register map for custom counter

And the corresponding block diagram is drawn as:

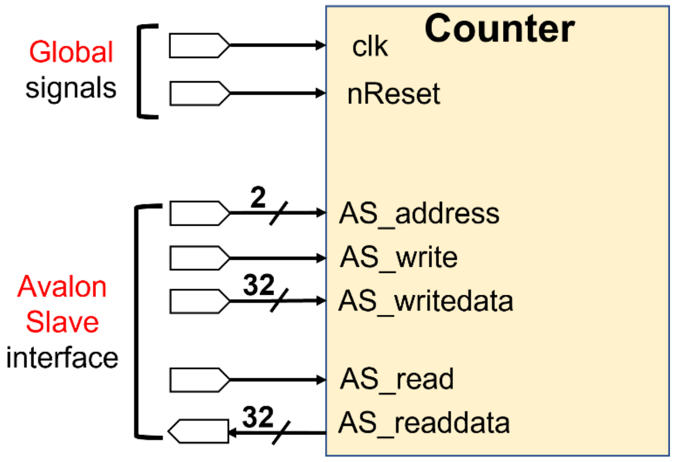


Figure 3: The block diagram of custom counter

2.3 SDRAM Controller and PLL

In order to figure out the effect of memory settings, here we use the SDRAM on the board to replace the on-chip memory to store instructions and data during some measurements. In order to successfully exploit the SDRAM, we implement a SDRAM controller and a PLL in the system design.

The parameters of SDRAM controller are set according to the datasheet of DE1-SOC board, and the PLL is used to generate two 100 MHz output clocks with different phase shifts. The one that doesn't have a shift is connected to the SDRAM controller, while the other one with a -3758 ps shift is exported and used as the input clock for the SDRAM.

3. Interrupt Parameters Measurement

3.1 Software Design

In this part, we use several different ways to measure three parameters of the interrupt in the NIOS-2 system. In general, we measure response time, recovery time and latency time of the interrupt as shown in Fig. 4.

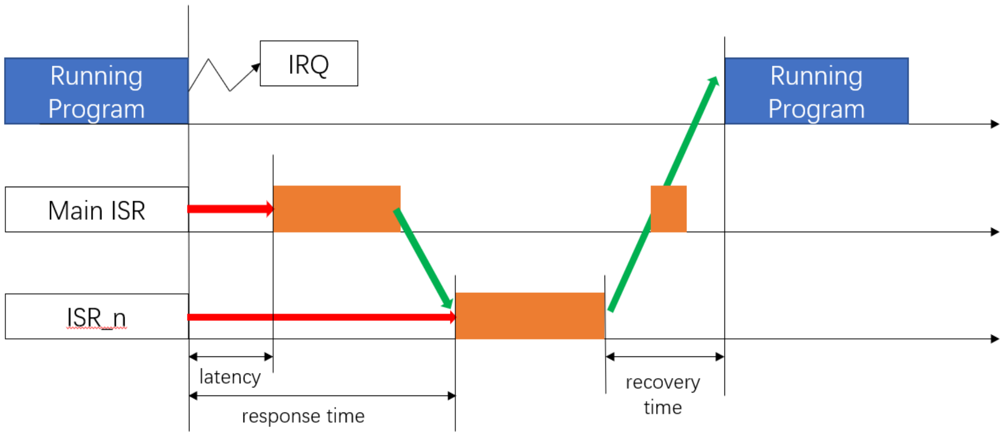
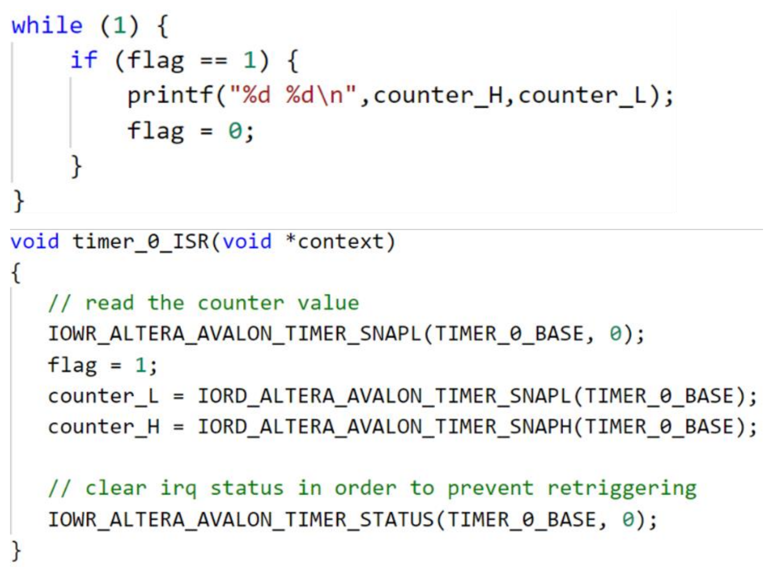


Figure 4. Parameters of interrupt in the NIOS-2 system.

3.1.1 Interrupt Response Time

In this part, we use the normal timer from Quartus IP library, as well as the custom PIO & logic analyzer to measure the interrupt response time. The corresponding results are recorded in Table 4.

In the normal timer method, the timer is set to generate an interrupt when its internal counter decreases to 0. Then at the beginning of the interrupt handler function, we get a snapshot of the counter. Since after the counter decreases to 0, it will be set to the maximum value again according to the timer period, we can use the difference between the maximum value and the snapshot value of the counter to represent the response time. The C code looks like Fig. 5, where we print the value in the main program to prevent deadlock.

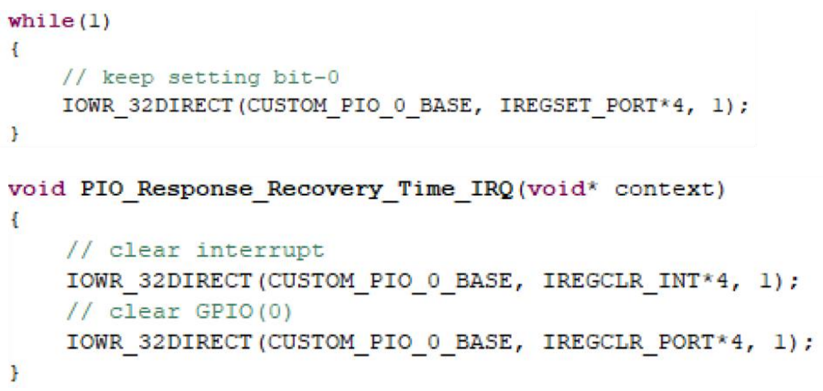


```c
while (1) {
    if (flag == 1) {
        printf("%d %d\n",counter_H,counter_L);
        flag = 0;
    }
}
```

```c
void timer_0_ISR(void *context)
{
    // read the counter value
    IOWR_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE, 0);
    flag = 1;
    counter_L = IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE);
    counter_H = IORD_ALTERA_AVALON_TIMER_SNAPH(TIMER_0_BASE);

    // clear irq status in order to prevent retriggering
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0);
}
```

Figure 5. Main program and IRQ handler function of the timer for response time.

In the custom PIO & logic analyzer method, the designed parallel port is initialized as output and connected to the external pins. In the main code, the interrupt is triggered by setting a bit on the parallel port. After that, in the IRQ handler function of the PIO, the port set before is cleared. Thus, the duration of the signal's high-level on the external port can be seen as the response time and can be measured by logic analyzer. The below figures show related software settings:



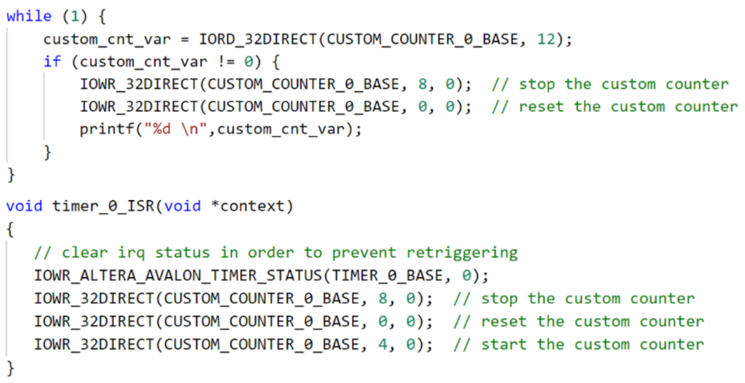
```c
while(1)
{
    // keep setting bit-0
    IOWR_32DIRECT(CUSTOM_PIO_0_BASE, IREGSET_PORT*4, 1);
}

void PIO_Response_Recovery_Time_IRQ(void* context)
{
    // clear interrupt
    IOWR_32DIRECT(CUSTOM_PIO_0_BASE, IREGCLR_INT*4, 1);
    // clear GPIO(0)
    IOWR_32DIRECT(CUSTOM_PIO_0_BASE, IREGCLR_PORT*4, 1);
}
```

Figure 6. Main program and IRQ handler function of custom PIO.

3.1.2 Interrupt Recovery Time

In this part, we use the timer & custom counter, as well as the custom PIO & logic analyzer to measure the interrupt recovery time. The corresponding results are recorded in Table 5.

In the first scheme, the timer is only used to generate an interrupt periodically, while the custom counter can indicate the time interval between the end of ISR_n and the beginning of the main program. Thus, we can reset and start the counter at the end of the IRQ handler function of the timer, then in main program, we keep reading the counter and when its value is non-zero, we can record this value, which indicates the recovery time, and then clear it. The snippet of our software program is shown as Fig. 7.



```
while (1) {
    custom_cnt_var = IORD_32DIRECT(CUSTOM_COUNTER_0_BASE, 12);
    if (custom_cnt_var != 0) {
        IOWR_32DIRECT(CUSTOM_COUNTER_0_BASE, 8, 0);  // stop the custom counter
        IOWR_32DIRECT(CUSTOM_COUNTER_0_BASE, 0, 0);  // reset the custom counter
        printf("%d \n",custom_cnt_var);
    }
}
void timer_0_ISR(void *context)
{
   // clear irq status in order to prevent retriggering
   IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0);
   IOWR_32DIRECT(CUSTOM_COUNTER_0_BASE, 8, 0);  // stop the custom counter
   IOWR_32DIRECT(CUSTOM_COUNTER_0_BASE, 0, 0);  // reset the custom counter
   IOWR_32DIRECT(CUSTOM_COUNTER_0_BASE, 4, 0);  // start the custom counter
}
```

Figure 7. Main program and IRQ handler function of the timer for recovery time.

In the second method, the same codes shown in Fig. 6 are used, which provide capabilities to measure the recovery time. At the end of the IRQ handler function, the external port is cleared. After that, the program goes back to main code and sets the port again. Thus, the duration of the signal's low-level in a period of square wave can be seen as the interrupt recovery time and can be measured by the logic analyzer.

3.1.3 Interrupt Latency Time

Since the latency is relatively short and difficult to capture in software, we use the Signal Tap Logic Analyzer tool to measure this parameter. The related measurements are record in Table 6.

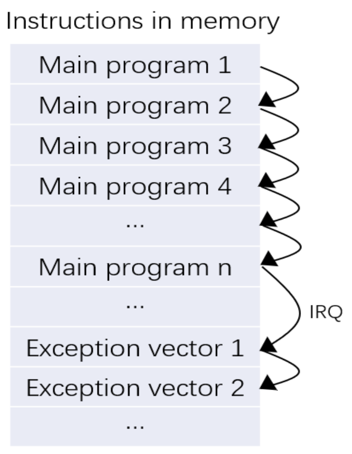


Figure 8. Memory layout and instruction address behaviors.

When the CPU goes from the main program to the main_ISR, the program counter (PC) will jump to the start address of exception vector specified in NIOS-2 core's settings, as shown in the Fig. 8. Thus, by observing the behavior of the PC, i.e., when it jumps from an ordinary address to the start address of main_ISR, we can measure how long the interrupt latency time is. To generate an interrupt, we implement the custom PIO, which can be added to Signal Tap too, and pull it up periodically. Then the time interval between the port's rising edge and a sudden jump of PC to the start address of main_ISR can be regarded as the latency time.

3.1.4 Program response under different memory settings

In this part, we firstly investigate how data cache and instruction cache affect the response of program by using our own designed PIO and an external logic analyzer. The following tasks are performed:

- The system clock frequency is set to 50 MHz and the operation of outputting continuous pulse waves on a physical pin GPIO-0 (i.e., Set bit 0, Clr bit 0, Set bit 0, Clr bit 0) are implemented in an infinite loop, then the logic analyzer with a sampling rate of 100 MHz is used to observe the pulse waves and make timing measurements. The results of pulse frequency in MHz and the system clock cycles that correspond to a period of the pulse wave are recorded in Table 3 below. To make our measurements more reliable and more accurate, 10 samples are taken for each configuration, and the corresponding mean value and standard deviation (shown in bracket) are computed.

Memory type	Cache memory configuration		Measurements of 4-instructions Loop	
	Instruction cache	Data cache	Pulse frequency (MHz)	Cycles in a PIO period
On-Chip memory	Disabled	Disabled	1.1166 (0.0783)	45 (3)
	Enabled	Disabled	4.5603 (0.5243)	11 (1)
	Enabled	Enabled	4.5835 (0.4165)	11 (1)
SDRAM memory	Disabled	Disabled	0.4472 (0.0267)	112 (7)
	Enabled	Disabled	4.5435 (0.4165)	10 (1)
	Enabled	Enabled	4.5835 (0.4165)	11 (1)

Table 3. Results of measurements of pulse waves with different methods and configurations

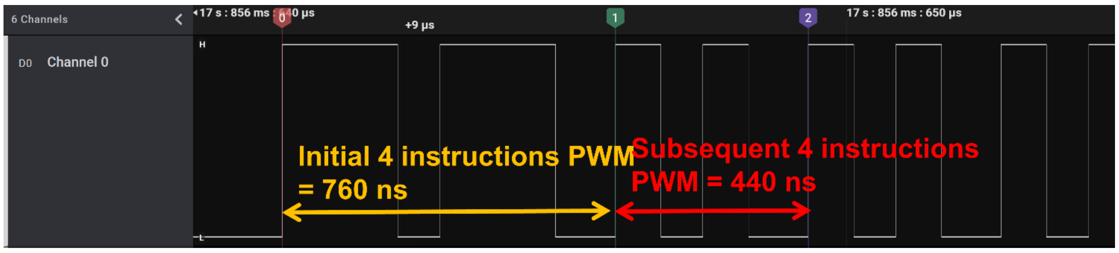

6 Channels
17 s : 856 ms : 640 µs
+9 µs
17 s : 856 ms : 650 µs
D0 Channel 0
H
L
Initial 4 instructions PWM = 760 ns
Subsequent 4 instructions PWM = 440 ns

Figure 9. Timing measurements of pulse waves when both caches are enabled.

First, when both cache memories are enabled, especially the instruction cache in this case, we can observe that the timings measured with on-chip memory are almost the same as the timings measured with SDRAM. This is to be expected because the operation of producing pulse waves only consists of only 4 short instructions that can be found in the instruction cache memory by processor (e.g., cache hit), which is faster than recomputing a result or reading from a slower data store, i.e., 4 times faster than on-chip memory and 10 times faster than SDRAM). The effect of cache can be explained intuitively by Fig. 9 which gives a snapshot of the output pulse waves measured by the logic analyzer showing the timing difference between the first pulse wave and the subsequent stable pulse waves:

From the figure above, we can see that the required clock cycles to output first pulse wave are 1.7 times longer than that required to output subsequent pulse wave. We think that this difference is due to the use of both cache memories, i.e., At the start of run, cache miss occurs because data and instruction are not available in the cache memory. When the processor detects a miss, it processes the miss by fetching requested data and instructions from main memory which causes longer program response times, thus the initial pulse wave required more clock cycles. After that, the required instructions are present in cache so that the cache hit occurs, making the program more responsive, thus the subsequent pulse waves required less clock cycles. Furthermore, it is also clear that the pulse waves measured in on-chip memory case are faster than the ones measured in SDRAM if both cache memories are not included. We think this is due to the physical distance between memory and the processor, that is the on-chip memory is closer to the processor than SDRAM. Thus, an access to the off-chip memory (e.g., here the SDRAM) requires longer access times and the performance of responding program is reduced.

3.2 Measurement Results

In this part, the results of interrupt measurements under different processors & memories configurations are presented and analyzed. To make our measurements more reliable and more accurate, 10 samples are taken for each configuration, and the corresponding mean value and standard deviation (shown in bracket) are computed and recorded in following tables.

Results for interrupt response time:

Memory type	Cache memory configuration		Response time (cycles)	
	Instruction cache	Data cache	Normal timer	PIO + logic analyzer
On-Chip memory	Disabled	Disabled	503 (2)	486 (1)
	Enabled	Disabled	224 (0)	201 (1)
	Enabled	Enabled	134 (4)	103 (0)
SDRAM memory	Disabled	Disabled	1602 (10)	1083 (8)
	Enabled	Disabled	473 (3)	282 (2)
	Enabled	Enabled	148 (0)	122 (1)

Table 4. Results of measurements of interrupt response with different configurations and methods

Results for interrupt recovery time:

Memory type	Cache memory configuration		Recovery time (cycles)	
	Instruction cache	Data cache	Timer + custom counter	PIO + logic analyzer
On-Chip memory	Disabled	Disabled	312 (1)	318 (0)
	Enabled	Disabled	152 (3)	137 (1)
	Enabled	Enabled	94 (4)	74 (1)
SDRAM memory	Disabled	Disabled	1025 (12)	945 (3)
	Enabled	Disabled	448 (5)	389 (3)
	Enabled	Enabled	122 (6)	97 (1)

Table 5. Results of measurements of interrupt recovery with different configurations and methods

Results for interrupt latency time:

Memory type	Cache memory configuration		Latency time (cycles)
	Instruction cache	Data cache	PIO + logic analyzer
On-Chip memory	Disabled	Disabled	15 (0)
	Enabled	Disabled	6 (0)
	Enabled	Enabled	6 (0)
SDRAM memory	Disabled	Disabled	38 (1)
	Enabled	Disabled	8 (0)
	Enabled	Enabled	8 (0)

Table 6. Results of measurements of interrupt latency with different configurations and methods

First, we observe that all interrupt timings are faster with the on-chip memory than with the SDRAM. This is to be expected, because the on-chip memory is closer to the processor than SDRAM. Thus, an access to the off-chip memory (here the SDRAM) requires longer access times. Furthermore, it is obvious that the delays required by interrupt are effectively reduced by enabling cache configurations. One thing should be noted here is that the data cache seems to have no effect on interrupt latency. From Table. 6, we can see that the performance of processor to access the interrupt handler is improved (e.g., reduced interrupt latency) when instruction cache is enabled, which is 2.5x and 5x faster for On-chip memory and SDRAM memory cases, respectively. We think these results are due to the usage of instruction cache, that is the processor can always 'hit' the instructions that are mostly commonly used in this lab to access the interrupt vector table that associates a list of interrupt handlers. It is also clear that of all three timing parameters, the interrupt response time is the longest, which accounts for most of the overhead involved in handing the interrupt (i.e., typically for saving CPU's context). For a given configuration, the measurements obtained using the timer method and the PIO method are generally comparable, although some differences can be noted. For example, under the SDRAM memory configuration, the recovery time measurements remain consistent between the two methods, while the corresponding response time measurements show quite large discrepancies.

4. Measurement with RTOS

In this section, we use MicroC/OS-II operating system to measure the overhead of different synchronization primitives provided by the OS kernel, e.g., semaphore, flags, mailbox, and queue. Because the configuration for this OS is done by the C program, the hardware part of the system is not much different from the previous ones, and we use the button (PIO from the library) on the board to trigger interrupt. Besides, SDRAM with enabled caches configuration is also deployed because there is not enough on-chip memory to hold MicroC/OS-II.

4.1 Software Design

The main work in this section is done in software program. We first construct a template project with Hello MicroC/OS-II, then develop the C program based on different parameters which need to be measured.

4.1.1 Semaphore

Semaphores are used to manage multi-tasks which may access shared resources. Thus, in order to measure the time for semaphores to convert from one task to the other, we assign only one available resource to the system. Then in the main loop, we call the PEND function, so that the task will stop and wait for the only one resource. In the IRQ handler function of PIO, we will call POST function to release the resource and allow other tasks to execute. Thus, by doing this, the time interval between the execution of the POST function and the PEND function can be regarded as the time needed to transfer from one task to the other in semaphore.

Therefore, in the IRQ handler function of the button, after executing POST function, we snapshot the value of the interval timer's counter. Then in the main loop, we snapshot again of the timer's counter value. Thus, the difference between these two snapshots is the time needed for transferring. Then, the program snippet can be found from the hello_ucosii.c file.

4.1.2 Flags

The idea to measure the parameters of flags is almost the same as the one in semaphore's measurement. After creating an event flag group, we also call the PEND function in the main loop, which waits for all flag bits to be set, and call the POST function in IRQ handler function, which will set the corresponding flag bit. Thus, when all buttons are pushed, all flag bits will be set, and the main task can enter PEND function. We use the interval timer to record and calculate the time interval between the previous task's completion and the next task's execution. This condition is known as flag AND. Flag also provides another case called OR. To test the speed of OR condition, we use the similar measurement procedures described in AND case but consider a new condition for calling the interrupt that is wait on one of the buttons' falling edge. The corresponding program snippet can be found from the hello_ucosii.c file.

4.1.3 Mailbox

Mailbox is also a structure to do arrangement of multi-tasks. Thus, the basic program is the same as the previous two situations. After creating a message mailbox in the beginning, in the main task, we use PEND to wait for the incoming message. In the IRQ handler function, we call the POST function to send a new message, and record the counter value of the interval timer. Then when the program goes back to the main task and enter PEND again, we record the end value of timer. Thus, the interval between these two counter values of the interval timer is the timing parameter for the mailbox. The corresponding code can be found from the hello_ucosii.c file.

4.1.4 Queue

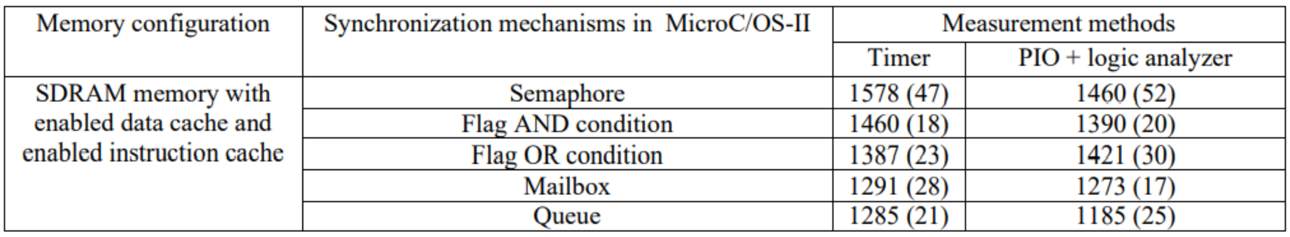
The scheme for measuring the timing parameter in queue is the same. We first create a message queue, then in the main loop, the PEND function will wait for a message in the queue, while the IRQ handler function of the button will send a message to the queue. So that we can measure the difference clock cycles between POST ends and PEND enters. The corresponding code can be found from the hello_ucosii.c file.

4.2 Measurement Results

In this part, the results of measurements for the semaphore, flags, mailbox and queue are displayed and compared. To make our measurements more reliable and more accurate, 10 samples are collected for each configuration, and the corresponding mean and standard deviation (shown in bracket) are computed. Moreover, to validate the elapsed time measured with our specific timer, all measurements are performed again with another method PIO & logic analyzer:

- Each time a signal operation is done, the bit-0 of PIO port corresponding to external GPIO-0 pin will be activated, i.e., signal’s LOW state to signal’s HIGH state. Each time a wait operation is done, the bit-0 is deactivated, i.e., signal is reset to LOW state. The duration of the signal’s HIGH state is captured and considered as the overhead time of the various synchronization primitive produced by operating system.

Table 7 shows the measurement results. As we can see from the table, the overhead timings are quite similar between different synchronization mechanisms, ranging from 1200 to 1500 cycles. To be specific, the semaphore takes the longest time to transfer from one task to the other, while the mailbox and the queue that transfer the most data seem to be the fastest synchronization mechanisms. Furthermore, the results are highly consistent between the timer method and the PIO & logic analyzer method.



Memory configuration	Synchronization mechanisms in MicroC/OS-II	Measurement methods	
		Timer	PIO + logic analyzer
SDRAM memory with enabled data cache and enabled instruction cache	Semaphore	1578 (47)	1460 (52)
	Flag AND condition	1460 (18)	1390 (20)
	Flag OR condition	1387 (23)	1421 (30)
	Mailbox	1291 (28)	1273 (17)
	Queue	1285 (21)	1185 (25)

Table 7. Measurement results of four synchronization mechanisms in MicroC/OS-II.

5. Conclusion

In this report, we illustrate how to measure the timing parameters for interrupt without RTOS and task transfer in MicroC/OS-II. In the first part, we measure the response time, recovery time and latency for interrupt in NIOS-2 CPU. We use the interval timer, pulse generated by custom PIO to do these measurements. These results are then displayed and analyzed. In the second part, we focus on the timing parameters for synchronization mechanisms for multi-tasks in MicroC/OS-II. We use the interval timer to measure this parameter for the semaphore, flags, mailbox and queue. Then the result shows that all of these mechanisms need some clock cycles to transfer from one task to the other, which should be taken into consideration when developing delay-sensitive applications.