Interrupt times measurement by software

Software Manual

Real Time Embedded System Course

LAP – IC – EPFL – 2011

Version 0.2 (Preliminary)

Cagri Onal, René Beuchat

1 Introduction

This guide has been prepared to help students following the Real Time Embedded System Course in I&C at EPFL. A development board FPGA4U is used during the laboratories with Quartus 10.1 and NIOS 10.1 from Altera and ModelSim-Altera from Mentor.

The latest copy of the tools can be found at LAP for personal installation:

\\lapsrv1\distribution\Altera\Tools_For_Windows\To_install_QuartusII_10_1

(or you can follow http:\\www.altera.com to download the install files after registration).

This guide assumes that the users know how to use Quartus10.1, NIOS 10.1 and ModelSim-Altera 6.6.For starters, here are some manuals helping to use these tools:

\\lapsrv1\distribution\Laboratoires\LaboFPGA4U\Using_NIOS2_EDS_10_v1x.pdf *\\lapsrv1\distribution\Laboratoires\LaboFPGA4U\UsingQuartusII_ModelSim_v0.6x.pdf*

2 Checking hardware components of the system working correctly

The hardware system to be designed for the interrupt analysis laboratory is as follows:

Conn...	Module	Description	Clock	Base	End	IRQ
	⊟ **cpu_0**	Nios II Processor	[clk]			
	instruction_master	Avalon Memory Mapped Master	**clk_0**			
	data_master	Avalon Memory Mapped Master	[clk]	IRQ 0	IRQ 31	
	jtag_debug_module	Avalon Memory Mapped Slave	[clk]	0x04009000	0x040097ff	
	⊟ **timer_0**	Interval Timer	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x0400a040	0x0400a05f	0
	⊟ **performance_counte...**	Performance Counter Unit	[clk]			
	control_slave	Avalon Memory Mapped Slave	**clk_0**	0x0400a000	0x0400a03f	
	⊟ **sdram_0**	SDRAM Controller	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x02000000	0x03ffffff	
	⊟ **onchip_memory2_0**	On-Chip Memory (RAM or ROM)	[clk1]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x04004000	0x04007fff	
	⊟ **jtag_uart_0**	JTAG UART	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	**clk_0**	0x0400a110	0x0400a117	1
	⊟ **epcs_flash_controlle...**	EPCS Serial Flash Controller	[clk]			
	epcs_control_port	Avalon Memory Mapped Slave	**clk_0**	0x04009800	0x04009fff	2
	⊟ **pio_0**	PIO (Parallel I/O)	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x0400a080	0x0400a08f	3
	⊟ **pio_1**	PIO (Parallel I/O)	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x0400a090	0x0400a09f	
	⊟ **pio_2**	PIO (Parallel I/O)	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x0400a0a0	0x0400a0af	
	⊟ **pio_3**	PIO (Parallel I/O)	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x0400a0b0	0x0400a0bf	
	⊟ **pio_4**	PIO (Parallel I/O)	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x0400a0c0	0x0400a0cf	
	⊟ **pio_5**	PIO (Parallel I/O)	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x0400a0d0	0x0400a0df	
	⊟ **pio_6**	PIO (Parallel I/O)	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x0400a0e0	0x0400a0ef	
	⊟ **pio_7**	PIO (Parallel I/O)	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x0400a0f0	0x0400a0ff	
	⊟ **pio_8**	PIO (Parallel I/O)	[clk]			
	s1	Avalon Memory Mapped Slave	**clk_0**	0x0400a100	0x0400a10f	
	⊟ **specific_counter_0**	specific_counter	[clock_sink]			
	avalon_slave_0	Avalon Memory Mapped Slave	**clk_0**	0x0400a060	0x0400a07f	4
	⊟ **ParPort_0**	ParPort	[clock_sink]			
	avalon_slave_0	Avalon Memory Mapped Slave	**clk_0**	0x0400a118	0x0400a11f	5

Fig. 1. Hardware system

After the hardware system is designed with Quartus 10.1 and tested with ModelSim, the next step will be to write suitable software for each timing measurement. Before that, testing each hardware peripheral with a small software is a good idea and assure that your hardware is working correctly. After that, any problem that will be faced with can be solved by using appropriate software.

In this section, only general functionalities of these programmable specific interfaces will be tested. Interrupt tests with these will be given in section 3.

2.1 Test of Specific Parallel Port

Following function first selects the parallel port as output, writes some value, then selects it as input and catches the input data and prints it on the console.

```
#define IREGDIR   0  //Change the values if your register map is different than
here
#define IREGPIN   1
#define IREGPORT  2
#define PARIRQEN  5
#define PARIRQCLR 6
#define MODE_ALL_OUTPUT 0xFF
#define MODE_ALL_INPUT  0X00
#define ALL_IRQ_EN      0XFF
#define ALL_IRQ_CLR     0xFF

void test_parallel_port()
{
	volatile unsigned int k;
	while(1)
	{
	IOWR_8DIRECT(PARALLELPORT_0_BASE,IREGDIR,MODE_ALL_OUTPUT);
		//Select Parport as output
	alt_printf("iRegDir=%x\n",IORD_8DIRECT(PARALLELPORT_0_BASE,IREGDIR));
	//Read iRegDir to check whether it is written correct
	IOWR_8DIRECT(PARALLELPORT_0_BASE,IREGPORT,0x9b);
		//Write Parport 0x9b as the output value
	alt_printf("iRegPort=%x\n",IORD_8DIRECT(PARALLELPORT_0_BASE,IREGPORT));
	//Read iRegPort to check whether it is written correct
	//Switch LEDS should give 0x9b, observe it
	for(k=0;k<1000000;k++); //software delay

	IOWR_8DIRECT(PARALLELPORT_0_BASE,IREGDIR,MODE_ALL_INPUT);
		//Select Parport as input
	alt_printf("iRegDir=%x\n",IORD_8DIRECT(PARALLELPORT_0_BASE,IREGDIR));
	//Read iRegDir to check whether it is written correct
	//Change the input value to a value different than 0x9b by changing switch
positions
	alt_printf("iRegPin=%x\n",IORD_8DIRECT(PARALLELPORT_0_BASE,IREGPIN));
	//Read iRegPin to take the input Parport value

	for(k=0;k<1000000;k++); //software delay
	}
}
```

Fig. 2. Test program for specific parallel port

Sample console output is as follows: (Note that the input at the 2nd loop is changed from **b4** to **94**)

```
iRegDir=ff        --selected as output
iRegPort=9b       --9b written as the output value
iRegDir=0         --selected as input
iRegPin=b4        --b4 taken as the input value, end of first loop
iRegDir=ff        --selected as output
iRegPort=9b       --9b written as the output value
iRegDir=0         --selected as input
```

2.2 Test of Specific Counter

```
#define IRESETVAL  0 //Change the values if your register map is different than
here
#define ICOUNTER   0
#define IRZ        1
#define ISTART     2
#define ISTOP      3
#define IIRQEN     4
#define ICLREOT    5
#define RESETVAL   0XFF000000 //Counter starts counting from this value
#define IRQENVAL   1
#define IRQDISVAL  0
#define CLREOTVAL  1
#define ARBITVAL   0X0000FFFF //Arbitrary writedata value used for addresses 1,2,3

void test_counter()
{
 IOWR(SPECIFIC_COUNTER_0_BASE, IRESETVAL, RESETVAL);
      //Reset value is loaded
 IOWR(SPECIFIC_COUNTER_0_BASE, IRZ, ARBITVAL);
      //Reset activated to load the counter with the reset value
 alt_printf("iCounter after reset= %x\n",IORD(SPECIFIC_COUNTER_0_BASE, ICOUNTER));
      //Check that counter is loaded with the reset value
 IOWR(SPECIFIC_COUNTER_0_BASE, ISTART, ARBITVAL);
      //Start the counter
 alt_printf("iCounter after start= %x\n",IORD(SPECIFIC_COUNTER_0_BASE, ICOUNTER));
      //Read a value from the running counter
 IOWR(SPECIFIC_COUNTER_0_BASE, ISTOP, ARBITVAL);
 alt_printf("iCounter after stop1= %x\n",IORD(SPECIFIC_COUNTER_0_BASE, ICOUNTER));
 alt_printf("iCounter after stop2= %x\n",IORD(SPECIFIC_COUNTER_0_BASE, ICOUNTER));
      //Two consecutive reads to test that the counter is stopped. They should
      give    the same result
 IOWR(SPECIFIC_COUNTER_0_BASE, ISTART, ARBITVAL);
      //Restart the counter
 alt_printf("iCounter after restart1=
%x\n",IORD(SPECIFIC_COUNTER_0_BASE,ICOUNTER));
 alt_printf("iCounter after restart2=
%x\n",IORD(SPECIFIC_COUNTER_0_BASE,ICOUNTER));
      //Two consecutive reads to test that the counter is stopped. They should
      give different results
 IOWR(SPECIFIC_COUNTER_0_BASE, ISTOP, ARBITVAL);

}
```

Fig. 3. Test program for specific counter

Sample console output is as follows:

```
iCounter after reset=    ff000000
iCounter after start=    ff000003
iCounter after stop1=    ff030c95
iCounter after stop2=    ff030c95
iCounter after restart1= ff030ca4
iCounter after restart2= ff06641a
```

3 Interrupt times measurement tests

As mentioned in the class, to successfully generate and execute an ISR firstly depends on the enabling of the interrupts both at the processor and at the interface level. Thus, the programmer has to guarantee

this fact by modifying the registers of the processor (Ienable register and PIE bit) and the interface(IRQen and IRQpend).

The required software for the interface side might be different for different interfaces thus it will be given with respect to the used interrupt source interface in the corresponding sections. However, for the processor side, it is almost the same and this one-line software for enabling the interrupts on the processor side is as follows:

```
alt_ic_isr_register(XXXX_0_IRQ_INTERRUPT_CONTROLLER_ID,XXXX_0_IRQ,my_isr,NULL,NULL
);
```

where XXXX can be either **TIMER** or **SPECIFIC_COUNTER** or **PARALLELPORT** with respect to the used interrupt source interface. 3rd parameter “my_isr” is the name of the ISR function whose prototype is

```
static void my_isr(void* context);
```

In the following sections, the software to measure the mentioned interrupt time is given in two parts; namely, the ISR(**my_isr**) and the required software in the **main** function.

3.1 Response time measurement

Response time can be measured with 3 different methods. The first one uses the timer from library, whereas the specific counter is used instead in the second method. Third method uses both the parallel port and logic analyzer to give a precise result.

3.1.1 With Timer (from library)

```
int main()
{
void *NULL;
alt_ic_isr_register(TIMER_0_IRQ_INTERRUPT_CONTROLLER_ID,TIMER_0_IRQ,my_isr,
NULL,NULL);
flag=0 ; //Flag is a global variable
IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,0);        //Clear control register
IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,2);        //Continuous mode ON
IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_0_BASE, 0xFFFF); //Set initial value
IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_0_BASE, 0x00FF);
IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,3);        //Enable timer interrupt
IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,7);        //Start timer
while(1)
{
// Normal program routine HERE…
   if(flag)
      {
         alt_printf("%x \n",0xffff-snap+1);
         flag=0;
         IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,7); //Enable IRQ and Start
timer
      }
};
return 0;
}
```

Fig. 4. Response time with timer (main part)

```
static void my_isr(void* context)
{
      IOWR_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE, ARBITVAL);
      snap = IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE);
      IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,0);  //Clear interrupt (ITO)
      IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0); //CLEAR TO
```

```
	flag=1; //Flag is a global variable
}
```

Fig. 5. Response time with timer (ISR part)

3.1.2 With Specific Counter

```
 int main()
{
void *NULL;
alt_ic_isr_register(SPECIFIC_COUNTER_0_IRQ_INTERRUPT_CONTROLLER_ID,SPECIFIC_COUNTE
R_0_IRQ,my_isr, NULL,NULL);

flag=0 ; //Flag is a global variable
IOWR(SPECIFIC_COUNTER_0_BASE, IRESETVAL, RESETVAL); //Reset value is loaded
IOWR(SPECIFIC_COUNTER_0_BASE, IRZ, ARBITVAL);       //Reset activated to load the
counter with the reset value
IOWR(SPECIFIC_COUNTER_0_BASE, IIRQEN, IRQENVAL);    //Enable IRQ
IOWR(SPECIFIC_COUNTER_0_BASE, ISTART, ARBITVAL);    //Start the counter

while(1)
{
// Normal program routine HERE…
	if(flag)
		{
		alt_printf("%x \n",snap);
		flag=0;
		IOWR(SPECIFIC_COUNTER_0_BASE, IIRQEN, IRQENVAL); //Enable the
interrupt
		IOWR(SPECIFIC_COUNTER_0_BASE, ISTART, ARBITVAL);    //Start the
counter
		}
};
return 0;
}
```

Fig. 6. Response time with specific counter (main part)

```
static void my_isr(void* context)
{
	snap = IORD(SPECIFIC_COUNTER_0_BASE, ICOUNTER);
		IOWR(SPECIFIC_COUNTER_0_BASE, IIRQEN, IRQDISVAL);   //Clear interrupt
		IOWR(SPECIFIC_COUNTER_0_BASE, ICLREOT, CLREOTVAL);  //Clear iEOT
		IOWR(SPECIFIC_COUNTER_0_BASE, ISTOP, ARBITVAL);     //Stop the counter
		IOWR(SPECIFIC_COUNTER_0_BASE, IRZ, ARBITVAL);       //Reset the
counter
	flag=1; //Flag is a global variable
}
```

Fig. 7. Response time with specific counter (ISR part)

3.1.3 With Specific Parallel Port and Logic Analyzer

The idea is to generate an interrupt by setting one bit of the parallel port and clear this bit at the beginning of the ISR. This gives a pulse whose duration at 1-level is the response time. This pulse can be measured by the logic analyzer. Note that the duration of the pulse at 0-level also gives the recovery time.

```
int main()
{
	void *NULL;
	alt_ic_isr_register(PARPORT_0_IRQ_INTERRUPT_CONTROLLER_ID,PARPORT_0_IRQ,my_i
sr, NULL,NULL);
	IOWR_8DIRECT(PARPORT_0_BASE,IREGDIR,MODE_ALL_OUTPUT); //Selected as output
	IOWR_8DIRECT(PARPORT_0_BASE,IREGPORT,0x00);
	IOWR_8DIRECT(PARPORT_0_BASE,PARIRQEN,ALL_IRQ_EN);//Enable IRQ on each bit
while(1)
	IOWR_8DIRECT(PARPORT_0_BASE,IREGPORT,0x02);	//Bit2 is SET st IRQ
generated
return 0;
}
```

Fig. 8. Response time with specific parallel port & logic analyzer (main part)

```
static void my_isr(void* context)
{
	IOWR_8DIRECT(PARPORT_0_BASE,IREGPORT,0);
	IOWR_8DIRECT(PARPORT_0_BASE,PARIRQCLR,ALL_IRQ_CLR); //CLEAR IRQ
}
```

Fig. 9. Response time with specific parallel port & logic analyzer (ISR part)

A sample logic analyzer output is as follows:

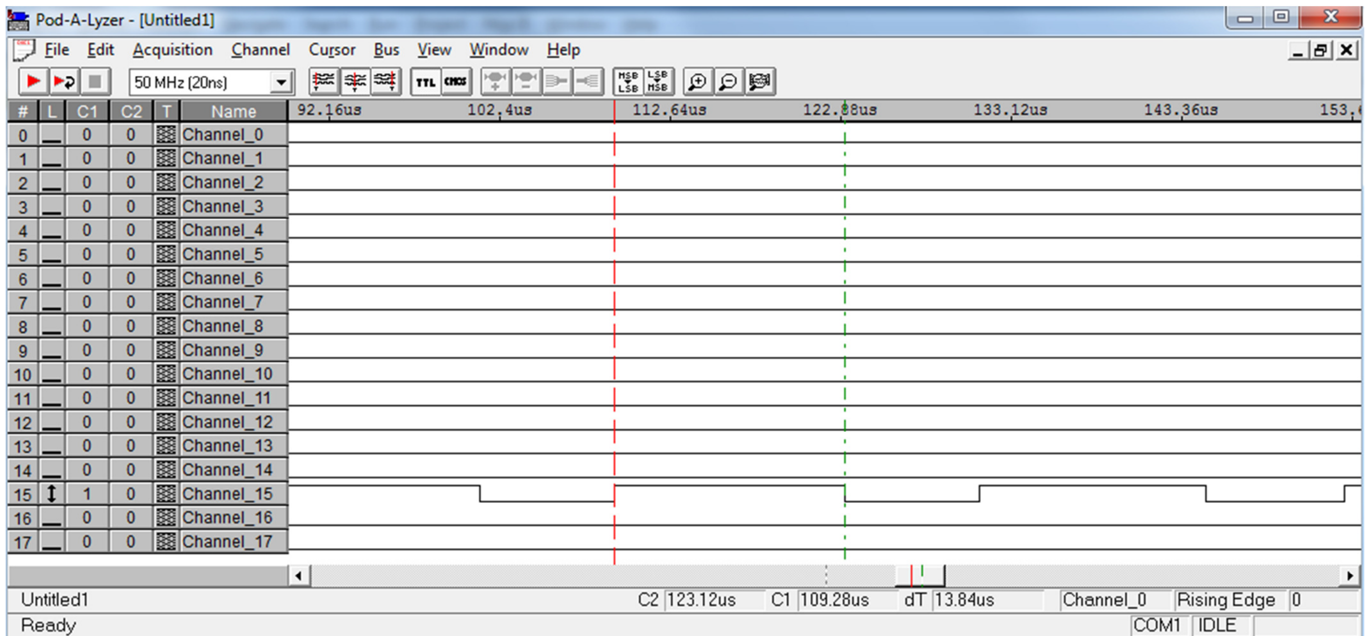

Fig. 10. Logic Analyzer output (recovery time)

3.2 Recovery time measurement

During this measurement, while one of the interfaces is used as the IRQ generation source, the other one will help to determine the result.

3.2.1 With Timer (from library) and Specific Counter

In this method, while timer is used as the IRQ source, specific counter will start counting from 0 at the end of ISR. A polling mechanism in the main function continuously checks the value of specific counter. Whenever we met a nonzero value, we print this value.

```
int main()
{
	void *NULL;
	alt_ic_isr_register(TIMER_0_IRQ_INTERRUPT_CONTROLLER_ID,TIMER_0_IRQ,my_isr,
NULL,NULL);

	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,0);  //Clear control register
	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,2);  //Continuous mode ON
	IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_0_BASE, 0xFFFF);
	IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_0_BASE, 0x00FF);

	IOWR(SPECIFIC_COUNTER_0_BASE, IRESETVAL, 0);       //Reset value(0) is
loaded
	IOWR(SPECIFIC_COUNTER_0_BASE, IRZ, ARBITVAL);      //Reset activated to load
the counter with the reset value
	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,3); //Enable timer interrupt
	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,7); //Start timer
while(1)
{
	while(IORD(SPECIFIC_COUNTER_0_BASE, ICOUNTER)==0);
	alt_printf("%x \n",IORD(SPECIFIC_COUNTER_0_BASE, ICOUNTER));
	IOWR(SPECIFIC_COUNTER_0_BASE, ISTOP, ARBITVAL);     //Stop the counter
	IOWR(SPECIFIC_COUNTER_0_BASE, IRZ, ARBITVAL);       //Reset the counter

	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,7);   //Start timer
};
return 0;
}
```

Fig. 11. Recovery time with timer & specific counter (main part)

```
static void my_isr(void* context)
{
	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,0);   //Clear interrupt (ITO)
	IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0);   //CLEAR TO
	IOWR(SPECIFIC_COUNTER_0_BASE, ISTART, ARBITVAL);    //Start the counter
}
```

Fig. 12. Recovery time with timer & specific counter (ISR part)

3.2.2 With Specific Parallel Port and Specific Counter

Here, IRQ source is the specific parallel port and we measure recovery time by the specific counter value. First, parallel port is defined as input so that whenever the user changes the switch positions, an IRQ is generated. At the end of ISR, specific counter starts counting from 0. A polling mechanism in the main function detects that counter value is nonzero and prints it on the console.

```
int main()
{
	void *NULL;
	alt_ic_isr_register(PARPORT_0_IRQ_INTERRUPT_CONTROLLER_ID,PARPORT_0_IRQ,my_i
sr, NULL,NULL);

	IOWR(SPECIFIC_COUNTER_0_BASE, IRESETVAL, 0);		//Reset value(0) is
loaded
	IOWR(SPECIFIC_COUNTER_0_BASE, IRZ, ARBITVAL);		//Reset activated to load
the counter with the reset value
	IOWR_8DIRECT(PARPORT_0_BASE,IREGDIR,MODE_ALL_INPUT); //Selected as output
	IOWR_8DIRECT(PARPORT_0_BASE,IREGPORT,0x00);
	IOWR_8DIRECT(PARPORT_0_BASE,PARIRQEN,ALL_IRQ_EN);	//Enable IRQ on each
bit
while(1)
{
	while(IORD(SPECIFIC_COUNTER_0_BASE, ICOUNTER)==0);
	alt_printf("%x \n",IORD(SPECIFIC_COUNTER_0_BASE, ICOUNTER));
	IOWR(SPECIFIC_COUNTER_0_BASE, ISTOP, ARBITVAL);		//Stop the counter
	IOWR(SPECIFIC_COUNTER_0_BASE, IRZ, ARBITVAL);		//Reset the counter

};
return 0;
}
```

Fig. 13. Recovery time with specific parallel port & specific counter (main part)

```
static void my_isr(void* context)
{
	IOWR_8DIRECT(PARPORT_0_BASE,PARIRQCLR,ALL_IRQ_CLR); //CLEAR IRQ
	IOWR(SPECIFIC_COUNTER_0_BASE, ISTART, ARBITVAL);	//Start the counter
}
```

Fig. 14. Recovery time with specific parallel port & specific counter (ISR part)

3.2.3 With Specific Parallel Port and Logic Analyzer

The idea is similar to the measurement of response time. This time, we set one bit of the parallel port of at the end of ISR and clear the same bit in the main program. Every clear operation results in a new interrupt since the parallel port interrupt is sensitive to both edges. Check 3.1.3 for implementation.

3.3 Latency measurement

The specific parallel port and the logic analyzer are used for these measurements. The measurements are done in different levels with respect to their software overhead. More precise results are expected from the measurements done in Assembly level since its overhead is smaller. The software in the main program is same for both measurements.

```
int main()
{
	void *NULL;
```

```
	alt_ic_isr_register(PARPORT_0_IRQ_INTERRUPT_CONTROLLER_ID,PARPORT_0_IRQ,my_isr, NULL,NULL);
	IOWR_8DIRECT(PARPORT_0_BASE,IREGDIR,MODE_ALL_OUTPUT); //Selected as output
	IOWR_8DIRECT(PARPORT_0_BASE,IREGPORT,0x00);
	IOWR_8DIRECT(PARPORT_0_BASE,PARIRQEN,ALL_IRQ_EN);//Enable IRQ on each bit
while(1)
	IOWR_8DIRECT(PARPORT_0_BASE,IREGPORT,0x02);	//Bit2 is SET st IRQ generated
return 0;
}
```

Fig. 15. Latency measurement (main part)

3.3.1 Measurement in C level

The least overhead C level for latency measurement is the source file **"alt_irq_handler.c".** In the main function, we can create an interrupt by setting one bit of the parallel port which was cleared before. Then by clearing the same bit in this source code **"alt_irq_handler.c"** gives us a pulse whose duration is the latency and this pulse can be observed by the logic analyzer. The corresponding C code is as follows:

```
	IOWR_8DIRECT(PARPORT_0_BASE,IREGPORT,0x00);	//Bit2 is CLEARED
```

Fig. 16. Latency measurement (IRQ Handler in C)

With this method, we measure a latency of 70 cycles with a processor without cache.

3.3.2 Measurement in Assembly level

The least overhead Assembly level for latency measurement is the source file **"alt_exception_entry.S".** In the main function, we can create an interrupt by setting one bit of the parallel port which was cleared before. Then by clearing the same bit in this source code **"alt_exception_entry.S"** gives us a pulse whose duration is the latency and this pulse can be observed by the logic analyzer. The corresponding Assembly code is as follows:

4 MicroC/OS-II

EXPLANATION HERE

```
#include <stdio.h>
#include "includes.h"
#include "system.h"
#include "io.h"
```

```
#include "sys/alt_irq.h"
#include "altera_avalon_timer_regs.h"

/* Definition of Task Stacks */
#define   TASK_STACKSIZE       2048
OS_STK    task1_stk[TASK_STACKSIZE];
OS_STK    task2_stk[TASK_STACKSIZE];
OS_STK    task3_stk[TASK_STACKSIZE];
OS_STK    task4_stk[TASK_STACKSIZE];

/* Definition of Task Priorities */

#define TASK1_PRIORITY      1
#define TASK2_PRIORITY      2
#define TASK3_PRIORITY      3
#define TASK4_PRIORITY      4

#define QUEUE_SIZE 10
OS_EVENT *sem_res;
OS_FLAG_GRP *flag_res;
OS_FLAGS flags;
OS_EVENT *mail_res;
OS_EVENT *queue_res;
//unsigned char c = 0x0f;

typedef struct {
    unsigned char button_number;
    unsigned char edge;
} msg;

msg* msg_queue[10];
msg some_msg;

unsigned int start,stop;
```

Fig. 17. Common definitions

4.1 Semaphore

EXPLANATION HERE

```
void task_sem(void* pdata)
{
  INT8U err;
  while (1)
  {
    OSSemPend(sem_res,0,&err);
    IOWR_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE, 9);
    stop = IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE);
    if(err == OS_NO_ERR)
```

```
        printf("Getting semaphore in : %u cycle\n", start - stop);
    else
        printf("Not getting semaphore\n");
  }
}
```

EXPLANATION HERE

```
void my_isr(void* context)
{
    INT8U err;
    unsigned char v = IORD(PIO_0_BASE,3);

    IOWR(PIO_1_BASE,0,v);
    switch (v){
        case 0x1:
            if ((IORD(PIO_0_BASE,0)&0x1) == 0)
             {
                OSSemPost(sem_res);
                IOWR_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE, 9);
                start = IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE);
             }
            break;

        default:
            break;
    }
}
```

4.2 Flags

EXPLANATION HERE

```
void task_flag(void* pdata)
{
    INT8U err;
  while (1)
  {
    OSFlagPend(flag_res, 0xf, OS_FLAG_WAIT_SET_ALL,0,&err);
    IOWR_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE, 9);
    stop = IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE);
    if(err == OS_NO_ERR)
        printf("Getting flag in : %u cycle\n", start - stop);
    else
        printf("Not getting flag\n");
  }
}
```

EXPLANATION HERE

```
void my_isr(void* context)
{

    OSFlagPost(flag_res, v,OS_FLAG_SET,&err);
    IOWR_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE, 9);
    start = IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE);
    IOWR(PIO_0_BASE,3,0x0f);
}
```

4.3 Mailbox

EXPLANATION HERE

```
void task_mail(void* pdata)
{
    INT8U err;
    msg* bla;

  while (1)
  {
    bla = (struct msg*)OSMboxPend(mail_res, 0, &err);
    IOWR_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE, 9);
    stop = IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE);
  if(err == OS_NO_ERR)
    printf("Getting message in : %u cycles with button number %u and edge %02X\n",
start - stop, bla->button_number, bla->edge);
    else
        printf("Not getting message\n");
  }
}
```

EXPLANATION HERE

```
void my_isr(void* context)
{

    some_msg.button_number = v;
    some_msg.edge = ((IORD(PIO_0_BASE,0)&0x1) == 0) ? 0x80 : 0;
    OSMboxPost(mail_res, &some_msg);
    IOWR_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE, 9);
    start = IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE);
    IOWR(PIO_0_BASE,3,0x0f);
}
```

4.4 Queue

EXPLANATION HERE

```
void task_queue(void* pdata)
{
    INT8U err;
    msg* bla;

  while (1)
  {
    bla = (struct msg*)OSQPend(queue_res, 0, &err);
    IOWR_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE, 9);
    stop = IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE);
  if(err == OS_NO_ERR)
    printf("Getting message from queue in : %u cycles with button number %u and
edge %02X\n", start - stop, bla->button_number, bla->edge);
    else
        printf("Not getting message\n");
  }
}
```

EXPLANATION HERE

```
  void my_isr(void* context)
{
    some_msg.button_number = v;
    some_msg.edge = ((IORD(PIO_0_BASE,0)&0x1) == 0) ? 0x80 : 0;
    OSQPost(queue_res, &some_msg);
    IOWR_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE, 9);
    start = IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER_0_BASE);
    IOWR(PIO_0_BASE,3,0x0f);
}
```

4.5 Main Function

EXPLANATION HERE

```
int main(void)
{
    INT8U err;
  //sem_res = OSSemCreate(1);
  //flag_res = OSFlagCreate(0,&err);
  //mail_res = OSMboxCreate(NULL);
  queue_res = OSQCreate(msg_queue, QUEUE_SIZE);
  IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_0_BASE, 0xffff);
  IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_0_BASE, 0x0000);
  IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,2);
  IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE,6);

alt_ic_isr_register(PARALLELPORT_0_IRQ_INTERRUPT_CONTROLLER_ID,PARALLELPORT_0_IRQ,
my_isr,NULL,NULL);

  IOWR(PIO_0_BASE,3,0x0f);
  IOWR(PIO_0_BASE,2,0x0f);

 /*
  OSTaskCreateExt(task_sem,
                  NULL,
                  (void *)&task1_stk[TASK_STACKSIZE-1],
                  TASK1_PRIORITY,
                  TASK1_PRIORITY,
                  task1_stk,
                  TASK_STACKSIZE,
                  NULL,
                  0);
  OSTaskCreateExt(task_flag,
                  NULL,
                  (void *)&task2_stk[TASK_STACKSIZE-1],
                  TASK2_PRIORITY,
                  TASK2_PRIORITY,
                  task2_stk,
                  TASK_STACKSIZE,
                  NULL,
                  0);
  OSTaskCreateExt(task_mail,
                  NULL,
                  (void *)&task3_stk[TASK_STACKSIZE-1],
                  TASK3_PRIORITY,
                  TASK3_PRIORITY,
                  task3_stk,
                  TASK_STACKSIZE,
```

```
                    NULL,
                    0);
*/
  OSTaskCreateExt(task_queue,
                    NULL,
                    (void *)&task4_stk[TASK_STACKSIZE-1],
                    TASK4_PRIORITY,
                    TASK4_PRIORITY,
                    task4_stk,
                    TASK_STACKSIZE,
                    NULL,
                    0);

  OSStart();
  return 0;
}
```

5 How to choose SRAM/SDRAM, With/Without Cache and their comparison

EXPLANATION HERE

Summary

1 Introduction 1
2 Checking hardware components of the system working correctly 2
2.1 Test of Specific Parallel Port 3
2.2 Test of Specific Counter 3
3 Interrupt times measurement tests 4
3.1 Response time measurement 5
3.1.1 With Timer (from library) 5
3.1.2 With Specific Counter 6
3.1.3 With Specific Parallel Port and Logic Analyzer 6
3.2 Recovery time measurement 7
3.2.1 With Timer (from library) and Specific Counter 8
3.2.2 With Specific Parallel Port and Specific Counter 8
3.2.3 With Specific Parallel Port and Logic Analyzer 9
3.3 Latency measurement 9
3.3.1 Measurement in C level 10
3.3.2 Measurement in Assembly level 10
4 MicroC/OS-II 10
4.1 Semaphore 11
4.2 Flags 12
4.3 Mailbox 13
4.4 Queue 14
4.5 Main Function 15
5 How to choose SRAM/SDRAM, With/Without Cache and their comparison 16
List of Figures 18

List of Figures

Fig. 1. Hardware system 2
Fig. 2. Test program for specific parallel port 3
Fig. 3. Test program for specific counter 4
Fig. 4. Response time with timer (main part) 5
Fig. 5. Response time with timer (ISR part) 6
Fig. 6. Response time with specific counter (main part) 6
Fig. 7. Response time with specific counter (ISR part) 6
Fig. 8. Response time with specific parallel port & logic analyzer (main part) 7
Fig. 9. Response time with specific parallel port & logic analyzer (ISR part) 7
Fig. 10. Logic Analyzer output (recovery time) 7
Fig. 11. Recovery time with timer & specific counter (main part) 8
Fig. 12. Recovery time with timer & specific counter (ISR part) 8
Fig. 13. Recovery time with specific parallel port & specific counter (main part) 9
Fig. 14. Recovery time with specific parallel port & specific counter (ISR part) 9
Fig. 15. Latency measurement (main part) 10
Fig. 16. Latency measurement (IRQ Handler in C) 10
Fig. 17. Common definitions 11