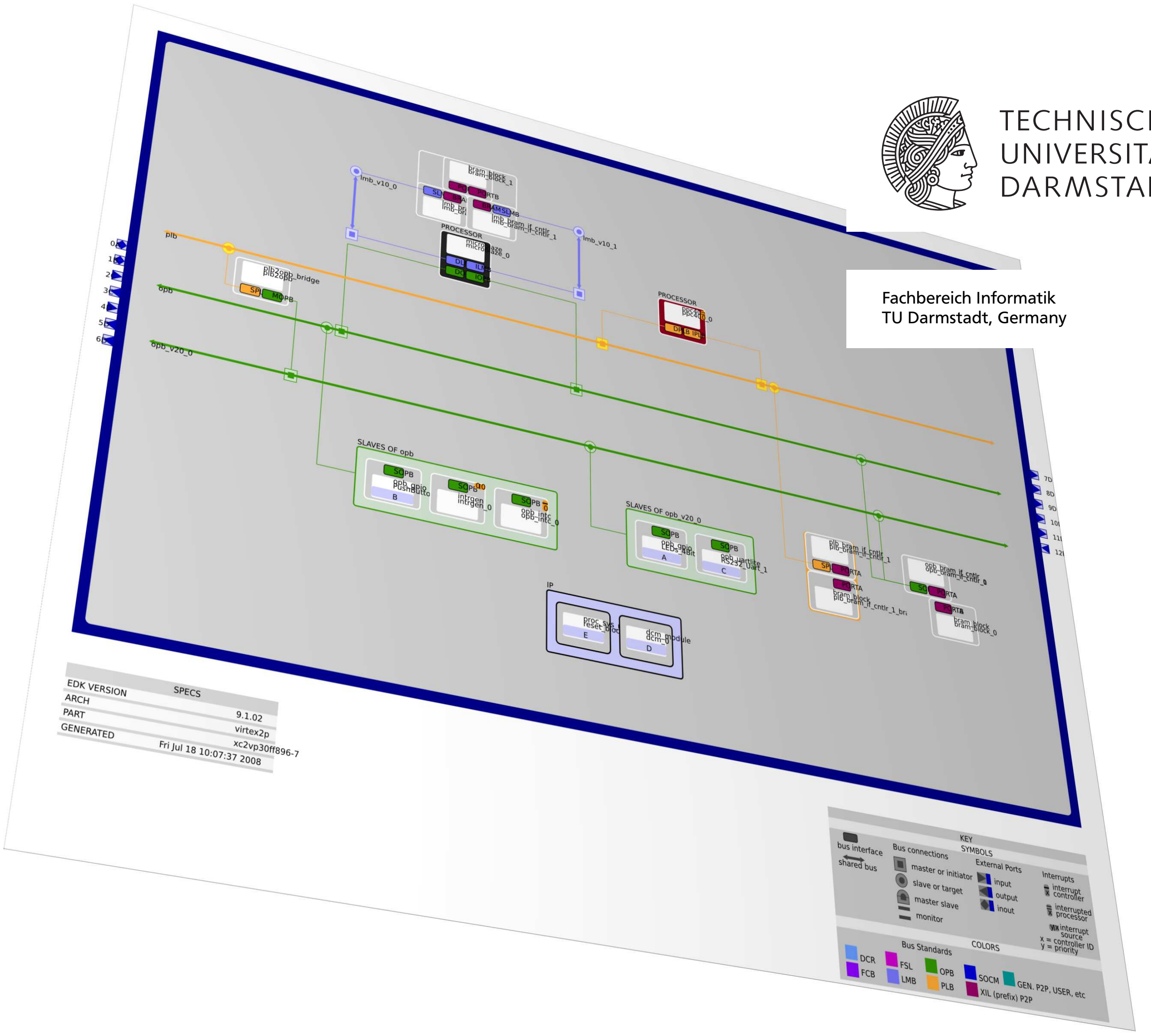
Technical Report
TUD-CS-2008-1103

Designing a Coprocessor for Interrupt Handling on an FPGA

H. G. Molter, H. Shao, H. Sudbrock, S. A. Huss, H. Mantel



TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
TU Darmstadt, Germany

Contents

1 Introduction

In order to enforce system-wide information-flow control policies, one needs to control the communication channels between different system processes. Covert channels [Lam73] are information channels that pose particular problems in this context. Such channels are means for the transmission of information, that are not intended for that purpose and can, e.g., be established by exploiting asynchronous hardware interrupts for the transmission of information [MS07, MS08]. Covert channels exploiting interrupt requests for covert communication are called interrupt-related covert channels.

There are various possibilities to mitigate interrupt-related covert channels without giving up on interrupt-based communication between hardware devices and the CPU. In our current project we explore approaches that avoid interrupt handling performed by the CPU by transferring the task of handling asynchronous hardware interrupts to an additional hardware component. This component contains the necessary functionality to take over the time-critical handling of asynchronous interrupts. The non-time-critical handling is taken over by the CPU after synchronization with the new component. We implement the additional hardware component on an FPGA.[1].

In this report, we present a specialized interrupt controller that is able to take over the handling of asynchronous interrupt requests caused by pressing push-button switches that, in our example scenario, control an array of LEDs. The design, comprising a CPU and the interrupt controller, is implemented on a Xilinx Virtex-II Pro System Development Board (XUP V2-Pro) [Xild]. The interrupt controller is implemented on a Xilinx MicroBlaze softcore processor. Besides the description of the hardware architecture and the hardware and software implementation we provide analysis results suggesting that such interrupt controllers implemented on a MicroBlaze softcore processor can be effectively used to mitigate interrupt-related covert channels.

This report is structured as follows: In Section 2 we introduce the class of interrupt-related covert channels in more detail. In Section 3 we describe the design of the specialized interrupt controller. In Section 4 we evaluate the effectiveness of our design with respect to the elimination of interrupt-related covert channels. We conclude in Section 5. The source code of the developed software is contained in Appendix A.

2 Interrupt-related Covert Channels

Using an interrupt-related covert channel, one process running on a system (the sending process) can transmit information to a second process running on the system (the receiving process). The transmission of information is based on operations that result in asynchronous interrupt requests. The receiving process continuously monitors a clock during its execution. This allows the process to notice the times at which it has been preempted by an interrupt request. For example, the observation that it was preempted at least once during a given time-slot could be interpreted as the value 1 and that it was not preempted as the value 0. To transmit the value 0 over this channel, the sending process only needs to refrain from executing operations that result in interrupt requests and, to send a 1, it performs such operations. Such an interrupt-related channel cannot be mitigated by assigning a constant quota of resource usage to each process [MS07], a technique that can be used to mitigate many other types of covert channels.

One can establish interrupt-related covert channels based on different operations and hardware devices. For example, one could use a network interface card to generate interrupt requests (see also Figure 2.1). Consider a network interface card that requests interrupts on two occasions: after a packet has been transmitted to the network and after a packet has been received from the network. To generate an interrupt request, the sending process could request the transmission of a packet via the NIC, since after the transmission of the packet the network interface card acknowledges the transmission by an interrupt request. The handling of this interrupt will occur during the receiving process' time-slot if the transmission request is issued at the proper time by the sending process.

Figure 2 illustrates the transmission of the bit sequence $\langle 1, 1, 0\rangle$ from the sending process (denoted with A) to the receiving process (denoted with B) in a simple example scenario where processes are scheduled alternately, and the only other active process is an operating system process (denoted with OS). In the diagram, time progresses from left to right, and the labeled boxes represent the time quanta where the label indicates the process.

The labeled circles indicate the points in time when transmission requests occur. The sending process performs a transmission request at α during its first quantum, and the corresponding interrupt is handled during the receiving process' first quantum (indicated by the shaded area in the box representing the receiving process' first time quantum). The sending process requests another transmission at time β. During its third time quantum, the sending

[1] Field programmable gate array, an electronic components whose inner logic can be freely programmed within certain restrictions

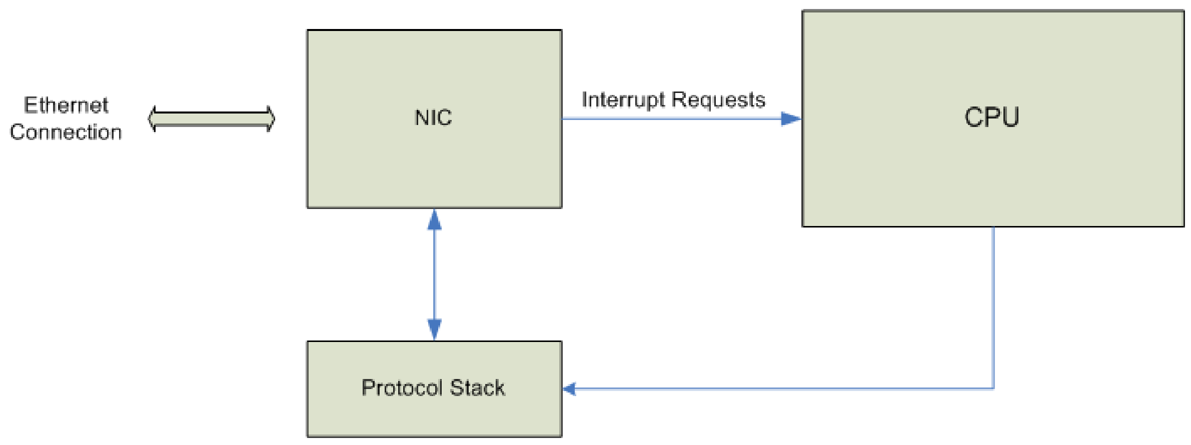


Figure 2.1: Typical architecture for a system containing a Network Interface Card (NIC) communicating with the CPU via asynchronous interrupts

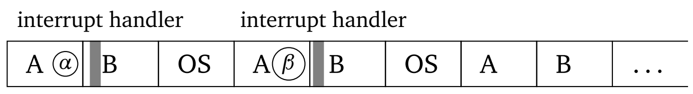


Figure 2.2: Exemplary transmission via an interrupt-related covert channel

process does not request any transmissions. The receiving process can then reconstruct the sequence by measuring the delay caused by interrupts in its three time-slots.

Exploitation scenarios like the one illustrated above are realistic threats. This has been demonstrated by an implementation of an interrupt-related covert channel exploiting interrupts from a network interface card [Gay08].

3 Coprocessor for Interrupt Handling

In this section, we present a proof-of-concept implementation for a coprocessor that handles interrupts of a simple hardware device.

The hardware device consists of five push-buttons ($A, \dots, E$) that are connected to a five-bit port P. Each bit of the port corresponds to one of the five push-buttons. Whenever a button is pressed down the corresponding bit of port P is set to one, otherwise it is set to zero. Furthermore, a button press results in the generation of an interrupt request. The corresponding interrupt handler reads port P and lights four LEDs according to the bits of P that are asserted.

One solution to free the CPU from handling these button interrupts would be to use polling instead of interrupt-based communication. However, depending on the polling frequency, button presses can be lost. Consider, e.g., a polling frequency of 1 Hz. If the buttons are pressed 3 times per second, button presses are lost.

This illustrates that giving up interrupt-based communication is an obvious disadvantage. We therefore implement a coprocessor that takes away the interrupt handling from the CPU itself. The interrupts are handled by the coprocessor which communicates them to the CPU via a shared memory.

3.1 System Architecture

In this project a custom processing system is developed and configured on an XUP[1] Virtex-II Pro Board [Xild] by combining a programmable Platform FPGA with a tightly integrated embedded design environment. For the implementation, the Embedded Development Kit (EDK, [Xila]) from Xilinx is used.

[1] Xilinx University Program

3.1.1 XUP Virtex-II Pro Development System

The main hard and soft components of the designed system are as follows:

Processor Cores:

- PowerPC (PPC) hard processor core (two embedded instances running at 300MHz)
- 32-bit MicroBlaze soft processor core (virtual core on the FPGA, defined by a VHDL description)

Peripheral IP[2] Cores:

- Parameterizable standard set of peripherals
- User defined peripherals

Interconnect Buses:

- Processor Local Bus (PLB) for fast communication with the processor
- On-Chip Peripheral Bus (OPB) for slow communication between peripheral devices
- Local Memory Bus (LMB) for communication with the local memory
- Fast Simplex Link (FSL) for uni-directional point-to-point communication between two peripheral devices

Input and Output Devices:

- Universal Asynchronous Receiver-Transmitter (RS232_UART serial port)
- Four LEDs connected to IO pins
- Five Push-button Switches connected to IO pins

Memories:

- On-chip Block RAM (BRAM)

Software Platform:

- Board Support Packages: Xilinx MicroKernel (XMK)
- Drivers for peripheral devices
- Custom proof-of-concept software

3.1.2 Interrupts on the XUP Virtex-II Pro Development System

The interrupt and exception handling on the Xilinx board is processed by a dual-level interrupt control structure composed of interrupt registers and the interrupt controller interface [Xilb].

Interrupt Controller: The interrupt controller interface is an external interrupt controller that combines asynchronous interrupt inputs from on-chip and off-chip sources and sends them to the PowerPC processor using two interrupt signals (one for the class of critical and one for the class of non-critical interrupts). When asserted, these signals indicate that interrupts are being requested. When deasserted, no interrupts are currently requested. In other words, the interrupt controller is responsible for collecting interrupt requests from peripherals and presenting them as a single critical or non-critical request to the PowerPC processor block. In addition, the interrupt controller is responsible for specifying the interrupt priorities, masking, and interrupt handlers as well.

Interrupt Ports: While the PowerPC has two interrupt ports, one port for critical and one port for non-critical interrupts, the MicroBlaze softcore processor has only one interrupt input port, to which a single interrupt signal can be connected directly. In both cases, if it is possible that multiple interrupts are generated at the same time, an interrupt controller must be present to handle the simultaneous interrupts.

[2] Intellectual Property

Interrupt Handling Mechanism: An interrupt handler is also known as an **I***nterrupt* **S***ervice* **R***outine (ISR)*. When an interrupt occurs, the PowerPC respectively the MicroBlaze processor will stop executing the current code and call the associated interrupt handler to manage the interrupts. More precisely, on the occurrence of interrupt requests, the PowerPC processor or, respectively, the MicroBlaze softcore processor jumps firstly to the location/address of the main ISR. Then from this main ISR the processor jumps to the ISR of the actual interrupt source.

If the source is an interrupt controller, the controller's ISR is responsible for managing the ISRs of the interrupts connected to it. It contains the routine that determines which individual ISR should be executed. From there, the processor jumps to the address of the individual ISR and executes the routine programmed at that location.

For any customized peripheral that generates an interrupt, an associated user ISR might be programmed and specified in the appropriate software application. This ISR should be registered at the interrupt vector table of the interrupt controller with it's function `Intc_RegisterHandler` afterwards. If an interrupt routine is not specified for a peripheral device, a default dummy interrupt handler is used. The main interrupt service routine is initiated with the function `XExc_RegisterHandler` for the PowerPC processor and with `microblaze_register_handler` for the MicroBlaze softcore processor, respectively. Once the process is started, the interrupt handling mechanism is very similar for both the PowerPC and the MicroBlaze core.

Interrupt Processing Time: For the MicroBlaze softcore processor the interrupt processing time is particularly critical, as it is not as fast as, e.g., the PowerPC processor. Usually, peripheral devices cannot wait for too long for an interrupt to be served. Therefore, interrupt handlers have to work very fast.

The time it takes the MicroBlaze softcore processor to enter an Interrupt Service Routine from the time an interrupt occurs depends on the configuration of the MicroBlaze processor and the latency of the memory controller storing the interrupt vectors. In addition, the user handlers take a variable amount of time depending on the associated service routines.

3.2 Design Space Exploration

In order to eliminate interrupt-related covert channels, one possibility is to free the PowerPC from the time-critical handling of asynchronous interrupt requests. To achieve this, we insert an additional coprocessor into the embedded system, which is designed and implemented on the Xilinx board (XUP Virtex-II Pro).

For time-critical asynchronous interrupts, the additional coprocessor should take the responsibility for handling these interrupts by servicing the associated interrupt handlers appropriately. For asynchronous interrupts that are not time-critical, the additional coprocessor should take the responsibility for collecting these interrupt requests, and presenting them in an aggregated form to the PowerPC as soon as the PowerPC is ready for synchronous communication. By this means, interrupt handling that is not time-critical is handed over periodically to the PowerPC and is serviced synchronously.

Figure 3.1 illustrates a variety of designs, which could, in principle, realize the desired supplemental functionality.

- In design variants 1a and 1b, a custom-made interrupt controller is used as a coprocessor. The custom-made interrupt controller shall take the responsibility of collecting the asynchronous interrupts from the hardware devices within the system, passing interrupts which are not time-critical to the PowerPC core, and handling the time-critical interrupts independently as well. In variant 1a, the collected interrupt data are transmitted to the PowerPC synchronously, which is implemented as one of the functionalities referred to the custom-made interrupt controller. In variant 1b, the collected interrupt data are transmitted to the PowerPC periodically due to periodic polling performed by the PowerPC. By this means, the interrupt data will be processed by the PowerPC processor core synchronously and periodically, which is sufficient for eliminating the potential occurrence of timing channels caused by asynchronous interrupt-driven communication.

- In design variants 2a and 2b, the functionalities realized by the custom-made interrupt controller in variants 1a and 1b are implemented without the basic functionality of a standard interrupt controller. Basic interrupt controller functionality is provided by a configured interrupt controller on the FPGA.

- In design variants 3 and 4, the self-defined interrupt controller is substituted by a configured OPB interrupt controller and a processor core (MicroBlaze softcore in variant 3, PowerPC in variant 4). The additional

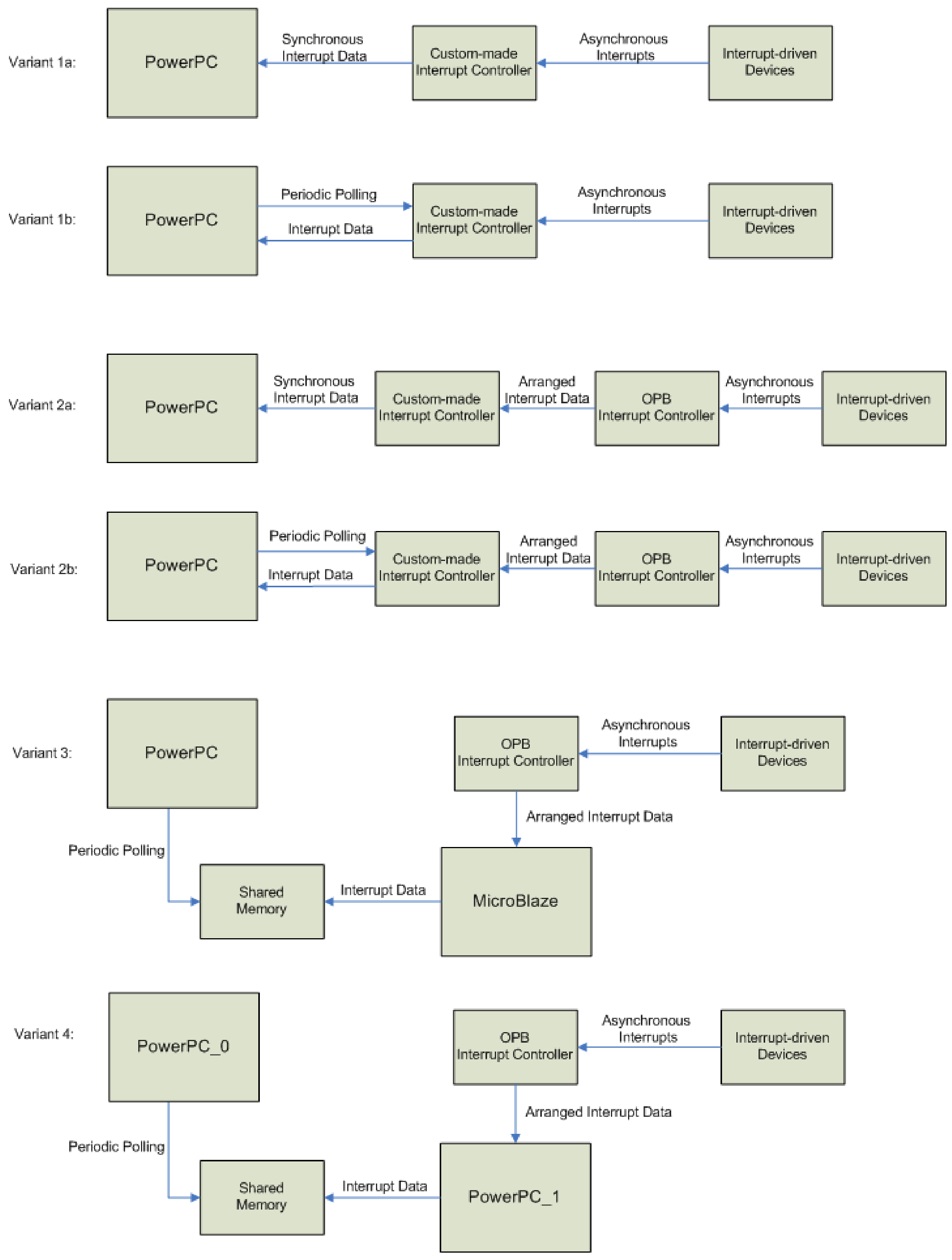


Figure 3.1: Design with additional coprocessors

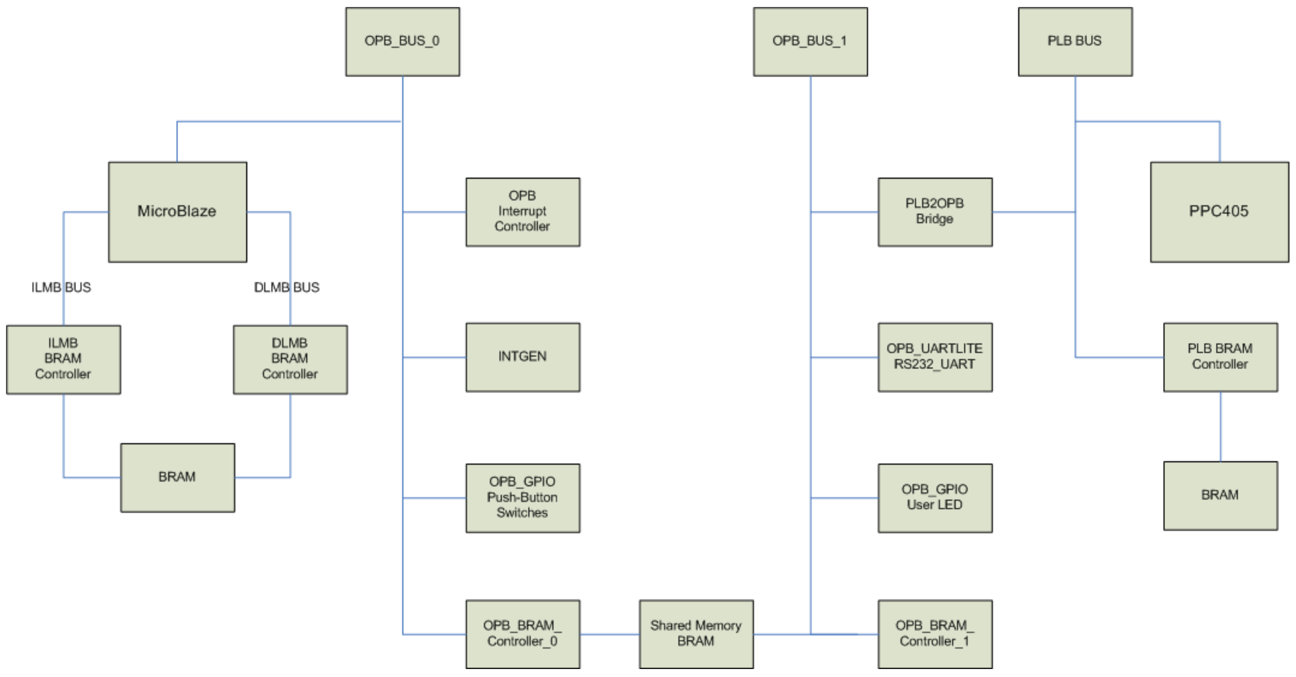


Figure 3.2: System with two processor cores (PowerPC and MicroBlaze)

processor core can fetch instructions and data from memory, execute program flow control instructions, perform input/output operations, manage memory, etc. In addition, the communication between the two processor cores is accomplished by a shared memory.

In order to ease the system development process, we chose design variant 3 for the implementation of the interrupt coprocessor. Using the rich functionality of a processor core makes it much easier to develop the functionality required by the design goal, as compared to the development of a custom-made interrupt controller from scratch. As the MicroBlaze processor is a softcore processor, once the design is finished, unnecessary parts of the processor can be easily removed, allowing for a smaller and cheaper solution than a full-fledged PowerPC core.

3.3 Implementation

The concrete implementation of the customized embedded system on the Xilinx board is illustrated in Figure 3.2. It consists of a PowerPC processor, a MicroBlaze softcore processor and a variety of peripheral components. The PowerPC processor is used as the CPU of the implemented system. On the PowerPC, the Xilinx Micro-Kernel (XMK, [Xilc]) is used as the operating system. The MicroBlaze softcore processor is applied in this system as a customized peripheral component, its main task is to manage and service the external and internal asynchronous interrupts. On the MicroBlaze softcore processor no special operating system is used, as it does not need any operating system functionalities like, e.g., threading.

3.3.1 Hardware Components

The configured settings of the hardware components within this system are summarized in Table 3.1. The peripheral *INTGEN* is used as an interrupt source in this system. The push-button switches on the Xilinx board are used as its interrupt trigger.

If one of the push-buttons is pressed, an interrupt request will be generated by *INTGEN*. This request is transmitted via the connected OPB bus to the interrupt controller *OPB_INTC*, in which interrupt requests are managed according to the configured specifications. Thereafter it is submitted to the *MicroBlaze* softcore processor, which handles and services the interrupt appropriately.

The processed output data from the *MicroBlaze* softcore processor will be delivered to the shared memory and redrawn by the *PowerPC* processor later. The *PowerPC* is responsible for forwarding the interrupt-associated infor-

System Property	Setting
FPGA Board	*Xilinx Virtex-II Pro* Development System
Processor Core	*PowerPC 405* with clock frequency 300 MHz *MicroBlaze* with clock frequency 100 MHz
Internal Interconnection	*PLB*, *OPB*, and *LMB* Bus with bus frequency 100 MHz peripheral *PLB2OPB_Bridge*
I/O Devices	peripheral *RS232_Uart* *Push-Button Switches* and *LEDs* on board
Interrupt Controller	peripheral *OPB_INTC*
Interrupt Source	custom peripheral *INTGEN*
Memory	*BRAMs* using associated memory controller: *plb_bram_if_cntlr*, *opb_bram_if_cntlr* and *lmb_bram_if_cntlr*

Table 3.1: Hardware components within system *sys-with-coprocessor*

mation via the *RS232_UART* receiver/transmitter to the outside world and to control the LEDs on the Xilinx board based on the interrupt-associated information.

Note that the interrupt sources are no longer directly connected to the PowerPC core. Information about interrupt requests can only reach the PowerPC if it is written to the shared Block RAM by the interrupt controller running on the MicroBlaze softcore processor.

3.3.2 Software Applications

Two software application projects (*microblaze-sw* and *powerpc-sw*) are developed in this system, one of them contains the software running on the PowerPC core, the other contains the software running on the MicroBlaze softcore processor.

In project *microblaze-sw* the customized interrupt handler *intr_generated* is implemented and registered to the interrupt vector table of the *MicroBlaze* softcore processor for handling the interrupt request submitted by the peripheral *INTGEN*. The interrupt handler reads the information about which button is pressed from the button controller. It then writes this information to the shared block memory.

In project *powerpc-sw*, the software running on the PowerPC is specified. This software polls the interrupt-associated information, which specify the exact location of the button being pressed and the occurrence times of interrupt events, from the shared block memory. It then transmits this information via the RS232 interface to the output terminal, and, based on this information, controls the status of an array of LEDS on the Xilinx board. The mapping table of the LEDs-light-on and the push-buttons-pressed is listed in the table below:

pressed Push-Button	light-on LED
Button UP	LED 0
Button DOWN	LED 1
Button RIGHT	LED 2
Button LEFT	LED 3
Button ENTER	LEDs 0 & 1 & 2 & 3

The source codes for these projects are provided in Appendix A.2.

4 Evaluation by Timing Measurements

To illustrate the effect of the interrupt coprocessor, we measured interruption times of a process running under the Xilinx Micro-Kernel on the PowerPC for two different scenarios: Without the dedicated interrupt coprocessor as presented in the previous section, and with this dedicated interrupt coprocessor. Without the dedicated interrupt coprocessor, the interruptions of the process by the interrupt handler running on the PowerPC should be notable by the measuring process, while in the presence of the coprocessor the interruptions should no longer be observable by the measuring process. Our measurement results support that this is in fact the case.

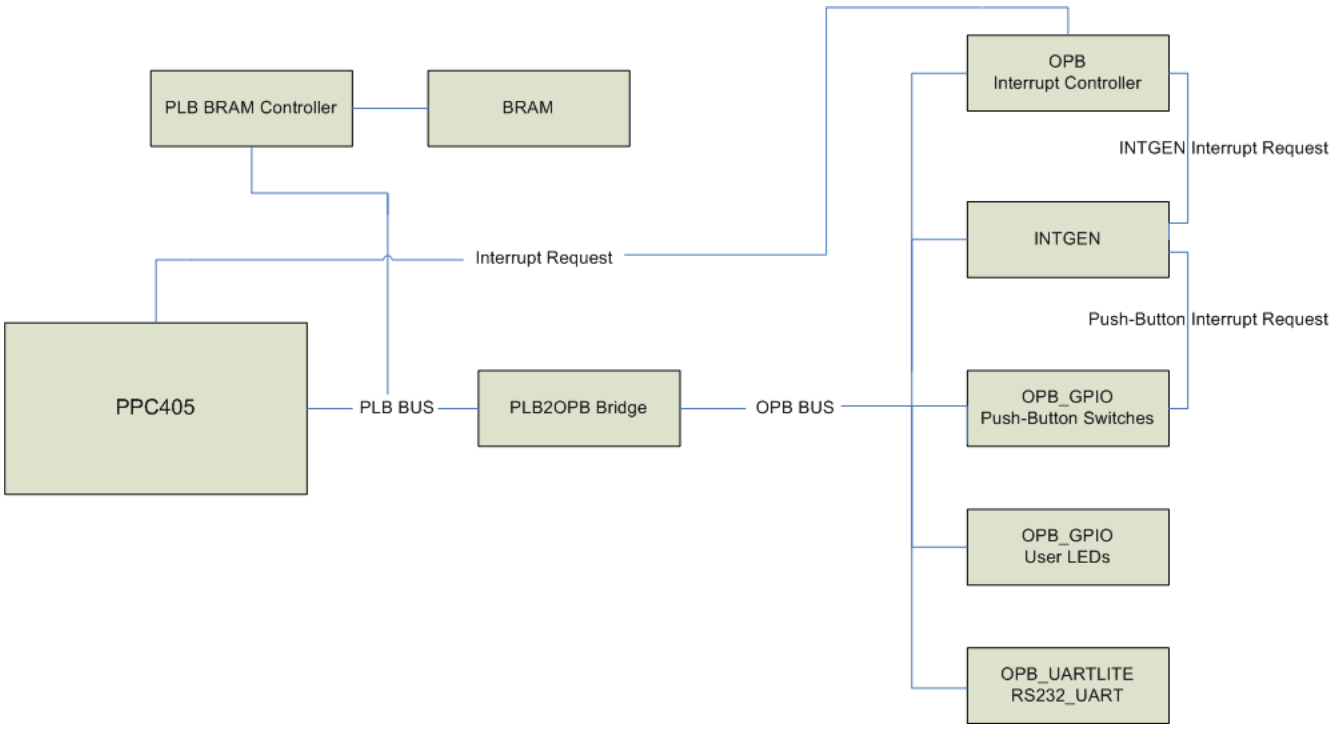


Figure 4.1: PowerPC system using XMK

System Property	Setting
FPGA Board	*Xilinx Virtex-II Pro* Development System
Processor Core	*PowerPC 405* with clock frequency 300 MHz
Internal Interconnection	*PLB* and *OPB* Bus with clock frequency 100 MHz peripheral *PLB2OPB_Bridge*
I/O Devices	peripheral *RS232_Uart* *Push-Button Switches* and *LEDs* on board
Interrupt Controller	peripheral *OPB_INTC*
Interrupt Source	custom peripheral *INTGEN*
Memory	*BRAM* using memory controller *plb_bram_if_cntlr*

Table 4.1: Hardware Components within system *sys-without-coprocessor*

To measure interruptions on a system without dedicated interrupt coprocessor, we developed and configured a second customized embedded system, called *sys-without-coprocessor*, on the XUP Virtex-II Pro Board using the processor PowerPC running the Xilinx Micro-Kernel (XMK). This system is similar to the system presented in the previous section in that it allows to switch LEDs on and off with the help of five push-button switches. However, the interrupt requests generated by the push-button switches are directly handled by the PowerPC, and not by a MicroBlaze softcore processor.

4.1 Hardware Design and Implementation of System without Coprocessor

The hardware infrastructure of the system consists of a PowerPC 405 processor core and a variety of peripherals, which are interconnected through the Processor Local Bus (PLB) and the On-Chip Peripheral Bus (OPB). The hardware architecture is illustrated in Figure 4.1. The configuration of the hardware components is summarized in Table 4.1.

The peripheral *INTGEN* is used as an interrupt source. The push-button switches on the Xilinx board are used as an interrupt trigger for *INTGEN*. If a push-button switch is pressed, the output signal *PushButtons_5Bit_IP2INTC_Irpt*

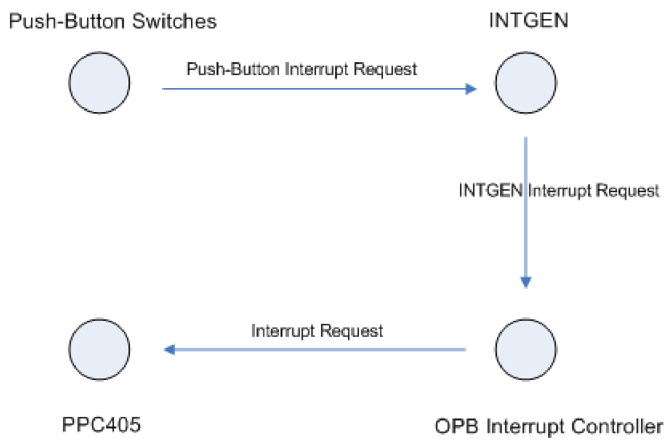


Figure 4.2: Interrupt signal flow within the system

Component	Description
Standard C Libraries (*libc, libm*)	Software libraries available for the embedded processors
Embedded Kernel (*Xilkernel*)	Kernel for the Xilinx embedded processors
Board Support Package *Standalone*	The lowest layer of software modules used to access processor-specific functions
Drivers	Device drivers for supported peripherals

Table 4.2: Components within software system XMK

from the peripheral *PushButtons_5Bit* will be set to '1'. It is captured by the peripheral *INTGEN* and the output signal *intrgen_0_IP2INTC_Irpt* will be asserted to '1'. By this means, the interrupt request is submitted to the interrupt controller *OPB_INTC*. This interrupt request is then forwarded from the interrupt controller *OPB_INTC* to the processor core *PPC405* via the signal *EICC405EXTINPUTIRQ*. Subsequently the associated interrupt handler is invoked by the processor core *PPC405*, and the interrupt is acknowledged and serviced appropriately. The transmission of the interrupt signal caused by pressing a push-button through the hardware components is illustrated in Figure 4.2.

4.2 Design and Implementation of the Evaluation Software

The software platform used for the measurement of process interruptions is developed and configured based on the *Xilinx Micro-Kernel* (**XMK**). The XMK includes the components listed in Table 4.2.

Operating System

As operating system we use the Xilkernel. In our testing environment, it is accommodated with the following functionalities:

- Thread creation, destruction and manipulation
- Round-robin scheduling with time-slices
- Timing measurement based on kernel clock ticks
- Interrupt controller and interrupt handling

Software Application

The software application project used to perform the measurements, called *measurement-sw*, is developed for this embedded system. It is linked with the Xilkernel described above. This application project consists of two programs written in the C programming language: *procA* and *procB* (the source code of both programs is contained in Appendix A.1, see files `procA.c` and `procB.c`). The *main()* routine of *procA* is declared as the entry point of the

```
unsigned long loop_length = 1000;
XTime start, end, diff;

XTime_GetTime(&start); // Measure time before execution of waster()
waster(loop_length);   // Execute function waster()
XTime_GetTime(&end);   // Measure time after execution of waster()

diff = end-start;      // Compute runtime between time measurements
```

Listing 4.1: Code fragment for runtime measurement

```
void waster (unsigned long loop_length) {
  unsigned long i = loop_length;
  volatile double v = 1.0;
  for (; i>0; --i) v *= 1.5; // Dummy calculation by simple multiplications
}
```

Listing 4.2: The function waster()

kernel. The start routine of thread *procA* (*procA_main*) is the starting point of the software application's execution, from which *procB* is started.

The program *procB* firstly registers the interrupt handler for the push-button interrupts with the operating system. Each time the interrupt handler is executed, a counter is increased, and the new interrupt count is printed on the screen. In addition, the interrupt status is queried and displayed. Finally, the interrupt signal is deasserted.

After registering the interrupt handler, the actual measuring phase takes place. In an infinite loop the program *procB* continuously executes a piece of program code (the function *waster()*) which performs a dummy calculation. By retrieving the system time before and after the execution of this piece of code the runtime of the code and of any interruptions during its execution is determined. The source code fragment for the runtime measuring is displayed in Listing 4.1, the source code fragment for the function *waster()* in Listing 4.2.

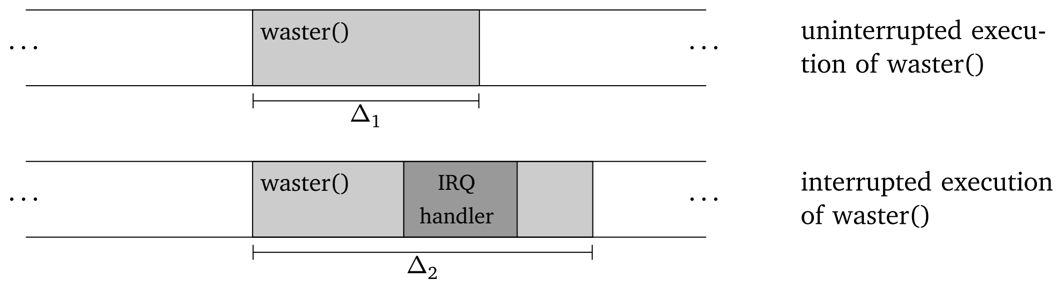
This software allows to detect interruptions of the program *procB*: If the process running *procB* is not interrupted, the measured runtime of the *waster()*-loop will approximately equal the actual runtime of the loop. If, however, the process is interrupted during the execution of the *waster()*-loop, the measured runtime will be as much longer as the time that the process was interrupted. This is illustrated in Figure 4.2: In both diagrams the execution of *procB* is illustrated, where the light-gray area depicts the execution of the function *waster()*, while the dark-gray area depicts the handling of an asynchronous interrupt request. In the upper time-line, the execution of *waster()* is not interrupted, while in the lower time-line the execution is is interrupted by the interrupt handler of an asynchronous hardware interrupt. In the illustration, the measured runtime Δ_2 in the lower diagram is obviously larger than the measured runtime Δ_1 in the upper diagram.

4.3 Evaluation Results

The times measured directly before and directly after the execution of the *waster()*-loop were displayed on the terminal during runtime. Based on the measurements obtained by executing this function for 100 times, we obtained the following results:

- Results for the system **without** dedicated coprocessor:
  - The processing time of the *waster()*-loop in the case that it **was** interrupted by an interrupt handler caused by pressing a push-button switch lies between 7410162 and 7814373 kernel ticks,[1] which is equivalent to the range from $24,70$ to $26,05$ milliseconds.
  - The processing time of the *waster()*-loop in the case that it **was not** interrupted lies between 5115312 and 5125386 kernel ticks, which is equivalent to the range from 17,05 to 17,08 milliseconds.

[1] Kernel ticks are the PowerPC clock ticks, occurring with a frequency of 300 MHz.



...
waster()
...
Δ_1
uninterrupted execution of waster()
waster()
IRQ handler
Δ_2
interrupted execution of waster()

Figure 4.3: Uninterrupted and interrupted runtime measurement, with runtime measurements Δ_1 respectively Δ_2

- Results for the system **with** dedicated coprocessor:
  - The processing time of the *waster()*-loop lies between 5199309 and 5200653 kernel ticks, which is equivalent to the range from $17,33$ to $17,34$ milliseconds, no matter whether interrupts were generated by pressing push-buttons or not.

We conclude from the above results for the system without a dedicated coprocessor that the interruption time of a process due to interrupt handling caused by a push-button press lies between 2284776 and 2699061 kernel ticks, which is equivalent to the range between approximately $7,6$ and $9,0$ milliseconds (the difference between the interrupted execution time of the *waster()*-loop and the uninterrupted execution time of the *waster()*-loop).

One can see from the results that in the presence of a dedicated coprocessor handling the push-button interrupts, the interruptions are no longer detected by our measurements.

The evaluation results suggest that it is indeed possible to mitigate interrupt-related covert channels by the usage of a dedicated coprocessor that shields the CPU from the asynchronous handling of interrupt requests. Since processes are no longer interrupted by asynchronous hardware interrupts, information can no longer be transmitted by generating and measuring such interrupts.

5 Conclusion

In the preceding sections we presented two different systems: one emulates a standard computer system in which the interrupt signal generated by pressing push-button switches is directly handled by the CPU; the other emulates a system with a dedicated interrupt coprocessor, the interrupt signal is handled by the interrupt coprocessor and information about the interrupt requests is saved in a desired form in the shared memory, which is polled synchronously and periodically by the CPU. We showed via experimental measurements that system processes running on the CPU can detect the presence of interrupt requests on the first system by measuring the runtime of a small piece of code. Experimental measurements on the second system suggest that the presence of interrupt requests can no longer be detected in this fashion when our interrupt coprocessor is in place. This means that our interrupt coprocessor is a sensible approach to counter the threat of interrupt-related covert channels. This result demonstrates that, besides software-based countermeasures as, e.g., proposed in [MS07], special-purpose hardware can be used to mitigate such covert channels.

We are currently working on extending this approach to more complex hardware components like network interface cards. Such components are in need of more complex interrupt handling. Some devices request an interrupt for various events, and the interrupt handler needs to handle each event appropriately. E.g., in the case of network interface cards, both incoming and outgoing network packets need to be handled. Furthermore, the handling is time-critical, since network packets need to be transmitted and received at very high speeds. For such hardware components, we aim at developing coprocessors that handle all time-critical communication with the network interface card autonomously, while communicating with the CPU in a synchronous fashion.

Bibliography

[Gay08] Richard Gay. Interrupt-related Covert Channels from an Attacker's Perspective. Diplomarbeit, RWTH Aachen, December 2008.

[Lam73] B. W. Lampson. A Note on the Confinement Problem. *Comm. ACM*, 16(10):613–615, 1973.

[MS07] Heiko Mantel and Henning Sudbrock. Comparing Countermeasures against Interrupt-Related Covert Channels in an Information-Theoretic Framework. In *20th IEEE Computer Security Foundations Symposium, CSF 2007*, pages 326–340, 2007.

[MS08] Heiko Mantel and Henning Sudbrock. Information-theoretic Modeling and Analysis of Interrupt-related Covert Channels. In *Preproceedings of the Workshop on Formal Aspects in Security and Trust, FAST 2008*, 2008.

[Xila] Xilinx,Inc. `http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm` (Abruf: 25.8.2008).

[Xilb] Xilinx,Inc. PowerPC 405 Konzepte. `http://direct.xilinx.com/bvdocs/userguides/ug011.pdf` (Abruf: 09.10.2007).

[Xilc] Xilinx,Inc. XilKernel 3.0a. `http://www.xilinx.com/ise/embedded/edk91i_docs/xilkernel_v3_00_a.pdf` (Abruf: 09.10.2007).

[Xild] Xilinx,Inc. XUP V2-Pro Produktwebseite. `http://www.xilinx.com/univ/xupv2p.html` (Abruf: 09.10.2007).

A Source Code

A.1 Project measurement-sw

A.1.1 File procA.c

```c
#include "xmk.h"
#include <os_config.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <xtime_l.h>
#include <sys/process.h>

extern void* procB_main (void*);

void* procA_main(void* param) {
  /* Declare time counter, thread and its attributes */
  XTime ticks;
  int retval;
  pthread_t B_tid;
  pthread_attr_t attr;
  pthread_attr_init (&attr);

  /* Indicate start of process execution */
  print("-- procA: Started up. --\r\n");

  /* Acquire current running time in unit ticks */
  XTime_GetTime(&ticks);

  /* Convert time from integer to hexadecimal string and print it as output */
  print("procA: clock ticks currently: ");
  putnum(ticks);
  print("\r\n");

  /* Generate thread with start address procB_main and define its attributes */
  retval = pthread_create(&B_tid, &attr, procB_main, NULL);

  if (retval != 0)
    xil_printf("procA: Error during pthread_create for B_tid.\r\n");

  while (1) {
    print(" procA: procA active.\r\n");
    /* Yield execution to the next process context in the queue */
    yield();
  }

  /* Indicate end of process execution */
  print("-- procA: procA finished --\r\n");

  /* Terminate the calling thread */
  pthread_exit(NULL);
  return 0;
}
```

```

int main() {
  print("\r\n--␣Entering␣xilkernel_main()␣--␣\r\n");
  /* Routine for the xilkernel entry point */
  xilkernel_main();
}
```

A.1.2 File procB.c

```
#include "xmk.h"
#include <stdio.h>
#include <pthread.h>
#include <xtime_l.h>

#include <os_config.h>
#include <xparameters.h>
#include <xstatus.h>
#include <sys/intr.h>
#include "xbasic_types.h"
#include "xgpio.h"

/* Declare time counter,interrupt counter and general purpose input device */
static XTime normal = 0;
static XGpio Buttons;
static volatile unsigned int intr_count = 0;

#define intr_true 0x00000001
#define XGPIO_IPIF_OFFSET 0x100
#define XIIF_V123B_IISR_OFFSET 32UL
#define Mask 1UL

/* The main task of waster is killing time */
void* waster(unsigned long loop_length) {
  print("procB:␣waster␣started␣up.");

  unsigned long i = loop_length;
  volatile double v = 1.0;

  /* Perform dummy calculation */
  for (; i>0; --i) v *= 1.5;
}

/* Self-defined interrupt handler */
void extra_int_handler(void *callback) {
  /* Indicate interrupt occurrence */
  printf("\r\n␣-␣Interrupt␣occurs␣\r\n");

  /* Increment the number of occurred interrupts and print it as output */
  unsigned int *count = (unsigned int *)callback;
  if (count != NULL) {
    (*count)++;
    printf("=>␣Interrupts␣generated␣%u␣times.\r\n", (*count));
  }

  /* If an interrupt is invoked successfully by pressing buttons on the FPGA board,
```

```c
   * the interrupt status should be 0x0001 */
  Xuint32 Reg32Value;
  Reg32Value = XGpio_InterruptGetStatus (&Buttons);
  xil_printf("␣-␣Buttons␣interrupt␣status␣:␣0x%08x␣\r\n", Reg32Value);

  /* Clear occurred interrupts */
  XGpio_InterruptClear(&Buttons, XGPIO_IR_CH1_MASK);
}

void* procB_main (void* param) {
  /* Indicate start of process execution */
  print("--␣procB:␣Started␣up.␣--\r\n");

  /* Register interrupt handler with interrupt controller */
  int_id_t id = XPAR_OPB_INTC_0_INTRGEN_0_IP2INTC_IRPT_INTR;
  XStatus status;
  status = register_int_handler(id, extra_int_handler, (void*) &intr_count);

  if (status == XST_SUCCESS)
    print("processB:␣Successfully␣registered␣a␣handler␣for␣extra␣interrupts.\r\n");
  else
    print("processB:␣Unable␣to␣register␣a␣handler␣for␣extra␣interrupts.\r\n");

  /* Enable interrupt within interrupt controller */
  enable_interrupt(id);

  /* Enable interrupt within device INTRGEN */
  INTRGEN_EnableInterrupt(XPAR_INTRGEN_0_BASEADDR);

  /* Initialize and configure the general purpose input device */
  if (XGpio_Initialize(&Buttons, XPAR_PUSHBUTTONS_5BIT_DEVICE_ID) != XST_SUCCESS) {
    printf("Failed␣to␣initialize␣the␣buttons.\r\n");
  }
  else {
    XGpio_SetDataDirection(&Buttons, 1, 0xFFFFFFFF);
    XGpio_InterruptClear(&Buttons, XGPIO_IR_CH1_MASK);
    XGpio_InterruptEnable(&Buttons, XGPIO_IR_CH1_MASK);
    XGpio_InterruptGlobalEnable(&Buttons);
  }

  /* Declare parameters for time counter */
  unsigned long loop_length = 1000;
  XTime start, end, diff;
  XTime diff_min = 0;
  XTime diff_max = 0;
  XTime diff_min_intr = 0;
  XTime diff_max_intr = 0;

  while (1) {
    /* Acquire the starting up and ending time for task waster */
    XTime_GetTime(&start);
    waster(loop_length);
    XTime_GetTime(&end);

    /* Acquire the duration for task execution and print it as output */
    diff = end-start;
    print("␣␣DURATION␣OF␣WASTER:␣");
    putnum(diff);

    if (normal == 0) {
      normal = diff;
      diff_min = diff;
      diff_max = diff;
      diff_min_intr = 0xffffffff;
      diff_max_intr = 0x00000000;
    }

    /* Update the range of execution duration for task waster with and
     * without interrupt disturbance correspondingly */
    if (diff < (0x00580000)) {
      if (diff > diff_max) diff_max = diff;
      if (diff < diff_min) diff_min = diff;
      print("␣=>␣Time␣range␣for␣execution␣WITHOUT␣interrupt␣:␣(");
      putnum(diff_min);
      print("␣,␣");
      putnum(diff_max);
      print(").\r");
    }
    else {
      if (diff > diff_max_intr) diff_max_intr = diff;
      if (diff < diff_min_intr) diff_min_intr = diff;
      print("␣=>␣Time␣range␣for␣execution␣WITH␣interrupt␣:␣(");
      putnum(diff_min_intr);
      print("␣,␣");
      putnum(diff_max_intr);
      print(").\r\n");

      print("␣=>␣Execution␣time␣of␣interrupt␣:␣(");
      putnum(diff_min_intr-diff_max);
      print("␣,␣");
      putnum(diff_max_intr-diff_min);
      print(").\r\n");
    }

    /* Yield execution to the next process context in the queue */
    yield();
  }

  /* Indicate end of process execution */
  print("--␣procB␣finished␣--\r\n");

  /* Terminate the calling thread */
  pthread_exit(NULL);
  return 0;
}
```

A.2 Projects powerpc-sw and microblaze-sw

A.2.1 File proc_XMK.c

```
#include "xparameters.h"
#include "xstatus.h"
#include "stdio.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "xio.h"
#include "prj.h"
#include "xtime_l.h"

/* Declare the general purpose output device */
 static XGpio Lights;

/* The main task of waster is killing time */
void* waster(unsigned long loop_length) {
  print("\r␣–␣Waster␣started␣up.");

  unsigned long i = loop_length;
  volatile double v = 1.0;

  /* Perform dummy calculation */
  for (; i>0; --i) v *= 1.5;
}

int main (void) {
  /* Indicate start of execution */
  print("\r\n—␣Entering␣main()␣––\r\n");

  unsigned int intr_count;
  Xuint32 light =0;

  /* Initialize and configure the general purpose output device */
  if (XGpio_Initialize(&Lights, XPAR_LEDS_4BIT_DEVICE_ID) != XST_SUCCESS) {
    printf("Failed␣to␣initialize␣the␣LEDs.\r\n");
  }
  else {
    XGpio_SetDataDirection(&Lights, 1, 0x00000000);
    /* Switch all LEDs off */
    XGpio_DiscreteWrite(&Lights, 1, 0x0F);
  }

  printf("Started␣up.\r\n");

  /* Declare parameters for time counter */
  unsigned int tmp_count = 0;
  unsigned long loop_length = 1000;
  XTime start, end, diff;
  XTime diff_min, diff_max;
  diff_min = 0xffffffff;
  diff_max = 0x00000000;

  /* Never exit the main function */
  while(1) {
    /* Acquire the starting up and ending time for task waster */
    XTime_GetTime(&start);
    waster(loop_length);
    XTime_GetTime(&end);

    /* Acquire the duration for task execution and print it as output */
    diff = end-start;
    print("  DURATION OF WASTER: ");
    putnum(diff);

    /* Update the range of execution duration */
    if (diff > diff_max) diff_max = diff;
    if (diff < diff_min) diff_min = diff;

    /* Read the output information out of the shared memory and
     * send it to the general purpose output device */
    light = LIGHT_mReadReg(XPAR_OPB_BRAM_IF_CNTLR_1_BASEADDR);
    XGpio_DiscreteWrite(&Lights, 1, ~light);

    /* Update the number of occurred interrupts in the shared memory*/
    intr_count = INTR_COUNT_mReadReg(XPAR_OPB_BRAM_IF_CNTLR_1_BASEADDR);

    /* Output interrupt information if an interrupt occurs */
    if (intr_count > tmp_count) {
      printf("\r\n - Interrupt occurs \r\n");
      printf("=> Interrupts generated %u times.\r\n", intr_count);
      tmp_count =  intr_count;

      print("=> Time range for execution of waster : (");
      putnum(diff_min);
      print(" , ");
      putnum(diff_max);
      print(").\r\n");
    }

  }

  /* Indicate end of execution */
  print("-- Exiting main() --\r\n");
  return 0;
}
```

A.2.2 File proc_MB.c

```
#include "xparameters.h"
#include "xstatus.h"
#include "xbasic_types.h"
#include "xio.h"
#include "xgpio.h"
#include "xintc_l.h"
#include "intrgen.h"
#include "prj.h"

#define BTN_RIGHT   0x0001
#define BTN_LEFT    0x0002
#define BTN_DOWN    0x0004
#define BTN_UP      0x0008
#define BTN_CENTER  0x0010
```

```c

/* Declare the general purpose output device */
static  XGpio Buttons;

/* Self-defined interrupt handler */
void intr_generated(void *callback) {
  /* Increment the interrupt counter in the shared memory
           in case of interrupt occurrence */
  unsigned int count = INTR_COUNT_mReadReg(XPAR_OPB_BRAM_IF_CNTLR_0_BASEADDR);
  count++;
  INTR_COUNT_mWriteReg(XPAR_OPB_BRAM_IF_CNTLR_0_BASEADDR, count);

  Xuint32 pressed, light =0;

  /* Update the pressed status of the general input device */
  pressed = XGpio_DiscreteRead(&Buttons, 1);

  /* Update the lighting status of the general output device and save
   * it into the shared memory */
  if (~pressed & BTN_LEFT)
    light |= 1;
  if (~pressed & BTN_RIGHT)
    light |= 2;
  if (~pressed & BTN_DOWN)
    light |= 4;
  if (~pressed & BTN_UP)
    light |= 8;
  if (~pressed & BTN_CENTER)
    light |= 0x0F;

  LIGHT_mWriteReg(XPAR_OPB_BRAM_IF_CNTLR_0_BASEADDR, light);

  /* clear interrupts */
  XGpio_InterruptClear(&Buttons, XGPIO_IR_CH1_MASK);

}

int main (void) {
  /* Enable interrupts within MicroBlaze */
  microblaze_enable_interrupts();

  /* Register the INTGEN interrupt handler in the vector table */
  XIntc_RegisterHandler(XPAR_OPB_INTC_0_BASEADDR,
    XPAR_OPB_INTC_0_INTRGEN_0_IP2INTC_IRPT_INTR, (XInterruptHandler) intr_generated,
    (void*)0);

  /* Start interrupt controller */
  XIntc_mMasterEnable(XPAR_OPB_INTC_0_BASEADDR);

  /* Enable interrupt requests within interrupt controller */
  XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR,XPAR_INTRGEN_0_IP2INTC_IRPT_MASK);

  /* Enable interrupts within INTRGEN */
  INTRGEN_EnableInterrupt(XPAR_INTRGEN_0_BASEADDR);

  /* Initialize and configure the general purpose input device */
  if (XGpio_Initialize(&Buttons, XPAR_PUSHBUTTONS_5BIT_DEVICE_ID) == XST_SUCCESS) {
    XGpio_SetDataDirection(&Buttons, 1, 0xFFFFFFFF);
    XGpio_InterruptClear(&Buttons, XGPIO_IR_CH1_MASK);
    XGpio_InterruptEnable(&Buttons, XGPIO_IR_CH1_MASK);
    XGpio_InterruptGlobalEnable(&Buttons);
  }
  return 0;
}
```