MSc THESIS

Interrupt Support on the ρ-VEX processor

Quan Kong

Abstract

In this thesis, we present a design of interrupt system upon an extensible and reconfigurable VLIW softcore processor: ρ-VEX. This interrupt system is designed and implemented in four mechanisms to match different application requirements in terms of the hardware consumption and performance issues (interrupt latency). On the other hand, due to the fact that the VEX compiler is not an open-source compiler, extra requirements to the assembler are also considered to make our work feasible. Our interrupt system itself can also be parameterized to fit different applications. These parameters include the number of interrupt vectors, interrupt priority of each vector and Interrupt Service Routines (ISRs) location address in the instruction memory. The testing results show that each version of our interrupt system takes reasonable amount of hardware usage.

We implemented our interrupt system on a virtex-6 FPGA. Besides, the interrupt latency can be reduced to only 2 clock cycles which is even better than some RISC-based softcore processors like Microblaze. This project creates a prototype of interrupt system that could work on VLIW softcore processor which extends the functionality and capability of the processor such as running operating systems and establishing a multi-core system.
Interrupt Support on the $\rho$-VEX processor

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Quan Kong
born in Hangzhou, China

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Interrupt Support on the \( \rho \)-VEX processor

by Quan Kong

Abstract

In this thesis, we present a design of interrupt system upon an extensible and reconfigurable VLIW softcore processor: \( \rho \)-VEX. This interrupt system is designed and implemented in four mechanisms to match different application requirements in terms of the hardware consumption and performance issues (interrupt latency). On the other hand, due to the fact that the VEX compiler is not an open-source compiler, extra requirements to the assembler are also considered to make our work feasible. Our interrupt system itself can also be parameterized to fit different applications. These parameters include the number of interrupt vectors, interrupt priority of each vector and Interrupt Service Routines (ISRs) location address in the instruction memory. The testing results show that each version of our interrupt system takes reasonable amount of hardware usage. We implemented our interrupt system on a virtex-6 FPGA board. Besides, the interrupt latency can be reduced to only 2 clock cycles which is even better than some RISC-based softcore processors like Microblaze. This project creates a prototype of interrupt system that could work on VLIW softcore processor which extends the functionality and capability of the processor such as running operating systems and establishing a multi-core system.

Laboratory : Computer Engineering
Codenummer : CE-MS-2011-01

Committee Members : 

Advisor: dr.ir. J.S.S.M. Wong, CE, TU Delft
Advisor: Msc. F. Anjam, CE, TU Delft
Chairperson: dr.ir. K.L.M. Bertels, CE, TU Delft
Member: dr.ir. Z. Al-Ars, CE, TU Delft
Member: dr.ir. T.G.R.M. van Leuken, CAS, TU Delft
Member:
Dedicated to my parents, for their love and support
Contents

List of Figures ix
List of Tables xi
Acknowledgements xiii

1 Introduction 1
   1.1 Motivation .................................................. 1
   1.2 Goal Definitions ............................................ 2
   1.3 Layout of the Thesis ........................................ 3

2 Background 5
   2.1 MOLEN and ρ-VEX ........................................... 5
      2.1.1 The MOLEN polymorphic processor .................... 5
      2.1.2 The ρ-VEX VLIW processor ........................... 6
   2.2 Background of Interrupt System ........................... 7
      2.2.1 Context Switching ....................................... 8
      2.2.2 Interrupt Task Scheduler .............................. 8
      2.2.3 Performance Issues .................................... 9
   2.3 Design Alternatives ....................................... 9
      2.3.1 Classic 8051/52 microprocessors ..................... 9
      2.3.2 Microblaze RISC softcore ............................ 10
      2.3.3 VLIW softcores ....................................... 11
   2.4 Conclusions ................................................. 11

3 Design 13
   3.1 System Organization ....................................... 13
   3.2 Interrupt Scheduler ....................................... 14
      3.2.1 Dataflow ............................................... 14
      3.2.2Interrupt Vector Table .................................. 15
      3.2.3 Scheduling Issues ...................................... 16
   3.3 Interrupt Controller ....................................... 16
      3.3.1 Design Logic ........................................... 16
      3.3.2 Method for Direct Context Switching (DCS) .......... 17
      3.3.3 Method for Hardware Instructions Context Switching (HICS) .. 19
      3.3.4 Method for Software Instruction Context Switching (SICS) .... 22
      3.3.5 Method for Page-able Register Files (PRF) ........... 22
   3.4 Interrupt Nesting .......................................... 24
   3.5 Integration with the ρ-VEX processor ...................... 24
      3.5.1 Fetch Stage ............................................ 25
# List of Figures

2.1 The MOLEN machine organization ................................. 5  
2.2 ρ-VEX in the MOLEN workflow .................................. 6  
2.3 4-issue ρ-VEX structure .......................................... 7  
2.4 Memory architecture of 8051 microcontroller .................... 10  
3.1 ρ-VEX organization with interrupter inserted .................... 13  
3.2 Concept Scheme of Interrupter ................................... 14  
3.3 Dataflow in interrupt scheduler ................................. 15  
3.4 Finite State Machine of interrupt controller ................... 17  
3.5 Interrupter block scheme for DCS method ....................... 18  
3.6 Instruction sequence that doing context switch .................. 21  
3.7 Interrupt block scheme of DICS method ......................... 21  
3.8 Interrupt block scheme of SICS method ......................... 22  
3.9 Interrupt block scheme of PRF method ........................... 23  
3.10 Example of handling nested interrupt ............................ 25  
4.1 The modular of interrupter ........................................ 28  
4.2 The organization of interrupt scheduler .......................... 28  
4.3 How Register Monitor is attached into ρ-VEX .................... 33  
4.4 Fetch stage that could receive context switch instructions ...... 35  
5.1 Performance comparison between different versions .............. 39  
5.2 Hardware utilization for different versions ....................... 40  
6.1 Signal flow for arithmetic overflow exception .................. 44  
6.2 Signal flow for invalid opcode exception ....................... 45  
6.3 Instruction layout example ....................................... 45  
6.4 Signal flow for unavailable hardware unit exception .......... 45  
6.5 Signal flow for division by zero exception ...................... 46  
6.6 Signal flow for software interrupts ............................. 48  
6.7 Syllable layout for INT_SOFT syllable ........................... 48  
6.8 Control signal that could enable/disable interrupts .......... 48  
6.9 Syllable layout for INT_WEN syllable ............................ 49  
6.10 Syllable layout for INT_CEN syllable ............................ 49  
6.11 Logic for switching tasks between cores ....................... 50  
6.12 Finite state machine for kernel mode ........................... 51  
B.1 Execution sequence of foreground program ...................... 69  
B.2 GR value storing and restoring .................................. 70  
B.3 BR value storing and restoring .................................. 70  
B.4 Context that stored in the data memory ........................... 70  
B.5 Data in data memory after execution ............................. 71
# List of Tables

<table>
<thead>
<tr>
<th>Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Example of interrupt vector table</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>layout of instructions for context switching</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Resource consumption for multiple pages of register files</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Logic for recording Context Switch information</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Logic for updating interrupt vector table</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Logic for scheduling task</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Logic for updating task queue</td>
<td>30</td>
</tr>
<tr>
<td>4.4</td>
<td>Control logic of the FSM in the interrupt controller</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Signal control for DCS method</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>Signal control for HICS method</td>
<td>32</td>
</tr>
<tr>
<td>4.7</td>
<td>Instructions generated when storing context</td>
<td>33</td>
</tr>
<tr>
<td>4.8</td>
<td>Signal control for PRF method</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>Description for different versions of interrupt handler</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Performance comparison between different versions</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Details of resource usage</td>
<td>40</td>
</tr>
<tr>
<td>5.4</td>
<td>Interrupt latency of Microblaze</td>
<td>40</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparison of interrupt latency between Microblaze and our design</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>Examples that enable/disable certain interrupt(s)</td>
<td>49</td>
</tr>
<tr>
<td>7.1</td>
<td>Requirements to the compiler (assembler)</td>
<td>56</td>
</tr>
</tbody>
</table>
Firstly, I would like to thank my advisor dr. Stephan Wong for giving me the opportunity to do the MSc project at the Computer Engineering Group. In addition to this, I also want to thank him for the advices about the project and proofreading my thesis.

I also want to express my gratitude to my daily supervisor PhD student Fakhar Anjam who guided me through out my work and gave many valuable suggestions on the project. He also helped a lot on proofreading my thesis and the paper resulted from my project, for which I am grateful.

I would like to thank to PhD student Roël Seedorf for many valuable discussions on the interrupt system and micro-architecture which is very useful for the design and implementation of my project. He also reviewed my thesis and gave many suggestions on the thesis writing and some technical issues.

Last but not least, I would like to thank for my parents and friends who gave me much support during my studies.

Quan Kong
Delft, The Netherlands
April 1, 2011
Field-Programmable Gate Arrays (FPGAs) are constantly improving in terms of speed and area and thus reconfigurable computing is becoming more popular nowadays. The MOLEN polymorphic processor provides one scheme to generate application-specific hardware unit and co-operate with general purpose processor to gain high performance. Another alternative for Custom Computing Unit (CCU) within the MOLEN architecture could be a reconfigurable and extensible Very Long Instruction Word (VILW) processor ($\rho$-VEX) which is developed at TU Delft. This thesis presents a subsystem which could be integrated into $\rho$-VEX to make the processor interruptible. Consequently, with this interrupt support, a range of applications could run on the $\rho$-VEX processor and reconfiguration at run-time becomes possible.

Section 1.1 presents motivation behind the presented work. The goal of this project is discussed in section 1.2. Finally, in section 1.3, an overview of this thesis’s organization is given.

1.1 Motivation

In the last decade, with the rapid technology progress, Field-Programmable Gate Arrays (FPGAs) are becoming more and more attractive to digital electronic system designers according to their flexibility and significant performance improvement. General purpose (GP) cores are widely used and applied to various applications because of their flexibilities. Application-Specific Intergraded Circuits (ASICs) on the other hand could show perfect performance due to the dedicated hardware units but only can be employed into specific domains and small number of applications. The emergence of reconfigurable processor bridges the gap between GP cores and ASICs so that more higher performance could be achieved without losing flexibility.

Reconfigurable processor is a microprocessor that contains hardware units that erase and re-adjust itself dynamically. It provides an efficient way for microprocessors to adapt into large amount of applications. The MOLEN polymorphic processor [17] shows a scheme that how a general-purpose core can be tied to a reconfigurable hardware Custom Computing Unit (CCU) dealing with application-specific calculations and gain overall performance improvement. Testing results show that more than 3 times speedup have been achieved on different state-of-the-art multimedia applications [17].

Processor $\rho$-VEX [19] is a reconfigurable and extensible open source Very Long Instruction Word (VLIW) softcore processor based on Lx/ST200 ISA developed by HP and STMicroelectronics [18]. It has been proved that it can be adapted to many applications and gain considerable reduction of execution time [14]. Nevertheless, due to the fact that the nature of VLIW structure would lead to low slot utilization for some applications or segments of code (high number of NOPs, in some cases it could go up
to 50% [18]), we would like to reconfigure the entire or partial hardware at run-time to increase performance in terms of area and speed. This functionality requires the ρ-VEX processor being able to be interrupted by outside world and to remember its execution context. Besides, we might also want to put the processor into the domain of real-time applications and give it the capability of dealing with exceptions within the processor at certain point. It is where the interruptibility support comes into the picture.

In this thesis, we present the design and implementation of four mechanisms to realize the interrupt system with respect to interrupt latency, hardware consumptions and the stress on the compiler and/or related toolchain. Our interrupt system is designed upon the VEX (VLIW Example) Instruction Set Architecture (ISA) [6] and the extensible and reconfigurable ρ-VEX softcore processor [18]. The interrupt system itself could also be reconfigured to support different applications. Parameters include the number of interrupt vectors, interrupt priority for each vector and Interrupt Service Routines (ISRs) location address. Since the current version of the VEX compiler which freely provided by Hewlett-Packard(HP) does not support interrupts, we might need to adjust back-end assembler a bit to make it work as a whole in certain mechanism(s). The compiler/assembler requirements for supporting interrupts are also discussed in this thesis.

1.2 Goal Definitions

The main goal of this project is to design and implement such hardware unit that could stop the foreground program running on the ρ-VEX processor and continue the task at a later phase without errors. During the time that the foreground program is stopped, the processor could give response according to different interrupt requests or reconfigure itself dynamically. These interrupt requests could be UART data transferring, periodically task switching request (when operating system is running on the ρ-VEX), any exceptions observed by the hardware or software interrupt request. Following steps are identified in order to achieve our goal:

1. A basic interrupt system is structured, it receives interrupt signals, then stops the foreground program, loads the ISR into pipeline and resumes the foreground program after the execution of ISR. The very first interrupt system does not check the interrupt priority, deal with context switching and support nested interrupts.

2. Design and implement a component that could raise interrupts according to priorities, thus lower priority interrupts can not interrupt the higher priority tasks.

3. Design and implement a context switching mechanism into the interrupt system, then we can manipulate register files in the ISR body without corrupting the context of the foreground program.

4. When context switching scheme is working, we then adjust it to support nested interrupts.
1.3  Layout of the Thesis

The remainder of the thesis is organized as follows. The research background is presented in chapter 2 where information about the ρ-VEX softcore processor, the necessary knowledge of interrupts and some alternative interrupt designs can be found. Chapter 3 discusses the possible designs of our interrupt system on the given processor. Consequently, implementation of our design is given in chapter 4. After that, chapter 5 presents the testing results of the implemented design and chapter 6 introduces system extensions such as exceptions support, software interrupt support, enable/disable interrupts support and a multi-core system that could make use of interrupt system to switch tasks between cores. Chapter 7 evaluates our design such that to give readers an overview of what we have achieved and where we could put more effort on in future. Finally, conclusions are drawn in chapter 8.

Appendix A presents how to configure the interrupt system in the source files. Appendix B shows the testing results that verified the correctness of our implementation of interrupt system. Appendix C explains how to add the source files of interrupt system into a newer version of ρ-VEX.
CHAPTER 1. INTRODUCTION
Our work is mainly focusing on setting up interrupt system on the ρ-VEX processor. This chapter introduces necessary investigations on the current processor and background technology and information on interrupts which would help to better understand of our design.

Section 2.1 presents an overview of related work about the ρ-VEX processor. Consequently, interrupt system and its important features are discussed in section 2.2. Then alternative possible designs are presented and compared in section 2.3. Section 2.4 concludes this chapter.

2.1 MOLEN and ρ-VEX

Since our work is mainly based on the ρ-VEX processor, we should investigate it first to get an overview and then to find an appropriate point to embed our subsystem into the processor.

2.1.1 The MOLEN polymorphic processor

![Figure 2.1: The MOLEN machine organization](image)

The MOLEN polymorphic processor paradigm presents a scheme that both general purpose and custom computing processing could be integrated together. Along with the
processor, a new programming paradigm, a new instruction set architecture, a microcode-based micro-architecture, and a new compiler methodology are introduced. Thus, it allows the designers to modify and extend the processor functionality at their will [17].

Figure 2.1 depicts the MOLEN machine organization [17]. Instructions are issued to either the core processor, which is a General-Purpose Processor (GPP) or a reconfigurable hardware, which is also known as Custom Computing Unit (CCU) by the Arbiter. Within CCU, selected functions have their very efficient hardware implementation and the \( \mu \rho \) code unit takes care of the control flow within the reconfigurable processor.

The \( \rho \)-VEX processor could be implemented as a CCU for the MOLEN processor, which would act as the reconfigurable processor. The application code segment that run on the \( \rho \)-VEX processor is identified by the Delft Workbench tool-chain [11] and then directly goes to the VEX compiler (including VEX compiler by Hewlett-Packard and \( \rho \)-ASM, developed at TU Delft), where executable code will be generated and then the executable code will be loaded into the instruction memory of the \( \rho \)-VEX. When the \( \rho \)-VEX processor is signaled by ‘start’, it will execute the application kernel. After it finished the execution, a CCU ‘done’ signal will be raised and sent to GPP. Figure 2.2 depicts the work-flow of \( \rho \)-VEX as co-processor in MOLEN processor [19].

![Figure 2.2: \( \rho \)-VEX in the MOLEN workflow](image)

### 2.1.2 The \( \rho \)-VEX VLIW processor

Very Long Instruction Word (VLIW) refers to CPU architecture that could take advantage of Instruction Level Parallelism (ILP). The VEX (VLIW EXample) ISA [6] offers a scalable technology platform for embedded VLIW processors, that allows variation in many aspects, including instruction issue-width, organization of functional units, and instruction set [19].
The $\rho$-VEX processor is designed and implemented based on Lx/ST200 ISA developed by HP and STMicroelectronics. It is Harvard architecture based design and pipelined with five stages. Figure 2.3 shows the 4-issue version of the $\rho$-VEX processor which implements 4 Arithmetic Logic Units (ALUs), 2 Multiplier units (MULs), 1 Control unit (CTRL) and 1 Memory unit (MEM). The General-purpose Register (GR) file is 64 32-bit registers and Branch Register (BR) file has 8 1-bit registers.

![Figure 2.3: 4-issue $\rho$-VEX structure](image)

Fetch stage will get instructions from instruction memory and then pass the instruction to the decode stage. Once branch instruction or stop instruction is decoded, a signal will be given by decode stage to change the instruction order (jump to branch target address) and flush the instruction fetched. In execute stage, decoded instruction will be assigned to different computing units. Finally, Writeback stage will write the results to register files.

### 2.2 Background of Interrupt System

Interrupt system deals with receiving interrupt requests, scheduling new task to CPU, saving and retrieving current execution context. It is vital for the function of any operating systems. Before we could design our interrupt system on $\rho$-VEX, background technology should be researched.


2.2.1 Context Switching

Context switching is a process of storing and restoring state of CPU so that we could resume to execute certain program at the same point at a later time after interrupts. This state (context) switch could refer to a register context switch, a task context switch, a thread context switch, or a process context switch. In multitasking operating system, context switch is an essential feature and always requires a certain amount of time for doing the administration. So much of design of operating system paid lots effort on optimization for the using of context switch. In our project, all temporary data is stored in registers (etc. stack pointer, argument and return values, link register, scratch registers [4, 6]), so contexts would equal to registers in this thesis.

In real world, there are three situations where a context switch needs to occur. They are multitasking, interrupt handling and user and kernel mode switching. In multitasking preemptive operating system, every task will run for certain amount of time slice and then context switch needs to be applied thus each task could be scheduled and executed. Within some operating system, a context switch might take place when a transition between user mode and kernel mode is required. The most common situation that a context switch happens, which is also the main reason for context switching discussed in our project, is interrupt handling. In real-time systems, interrupt handling is very important since we don’t need to busy-wait until the current task is over. It guarantees the safe schedule for critical tasks and ensures we could continue the job at the same point in a later time.

Context switching can be performed primarily by software or hardware. In software context switching, program itself will determine which registers should be saved and later retrieved. The obvious advantage of this method is that it gives much freedom to compiler to save different set of context for different application and thus reduce overhead of context switching. For some mainstream operating system, like Linux [2], is using this method. This is due to the hardware context switching does not save all the registers (only general purpose registers, not floating-point registers) and associated performance issues. For hardware context switching, when a switch occurs, the hardware will automatically somehow remember the current status (including program counter, stack pointer, all temporary registers etc.) where it could be restored and executed later.

2.2.2 Interrupt Task Scheduler

The scheduler basically determines what task is going to be executed at what time. In interrupt system, the scheduler receives interrupt requests, and then decides whether to raise task switch according to their priorities. In operating systems, the scheduler provides ability to schedule the launch of programs at some pre-defined times. For some emergency requests, the scheduler would schedule the task as soon as possible in order to give response to the request before deadline.

The importance of a task is defined by its task priority, or interrupt priority in interrupt system. Whenever an interrupt service request with higher priority (higher than the current running one) comes in, it should always be scheduled and executed as soon as possible. If there is more important task running, the service request should not get response until the current job is done. When two interrupts happen at the same
time, the scheduler should pick the one with higher priority to execute first and let the
other one wait in the "schedule list".

Another important feature for scheduler is that it should know what response should
be given for different interrupt request. When interrupt occurs, processor should execute
certain particular code to response that interrupt. This piece of code is called as Interrupt
Service Routine (ISR). ISRs are located somewhere in the instruction memory, which in
some cases the entry address are predefined as fixed memory address while for others
they could be custom defined. Nevertheless, no matter what kind of mechanism is used,
the hardware would have to load the address of ISR after respective interrupt are raised.

2.2.3 Performance Issues

Performance in interrupt system mainly refers to the length of interrupt latency which
is usually dominated by context switching. In multi-threading systems or real-time sys-
tems, we should always make the interrupt latency as small as possible since it effects
the performance of the whole system. The time spent for context switching is normally
determined by the size of context to be saved and the efficiency of saving the context.
Context switch overhead can also come from hardware interrupts and Dan Tsafrif in-
troduces two implementing methodologies to measure the overhead of context switching
in multi-threading system[16]. Besides, for some specially embedded processor archi-
tecture or operating system, there might be many other things could affect interrupt
performance which is discussed in [13].

In most interrupt systems, the interrupts will be disabled while switching the context.
If the interrupt latency is too long, the possibility of losing interrupts could become high
which is not acceptable in many real-time systems. For that reason, we should minimize
our interrupt latency as much as possible.

2.3 Design Alternatives

In this section, several possible designs of interrupt system are investigated and possibil-
ity of mapping these designs on the $\rho$-VEX processor is discussed in details. This section
also lead to our own design of interrupt support on the $\rho$-VEX processor.

2.3.1 Classic 8051/52 microprocessors

In 8051/8052 memory architecture, register files have 4 copies of them (figure 2.4) and
each of them could be selected to store data during run-time. The 80c51 compiler [1]
generates ISR code either pushing temporary data into stack or switch to use different
bank of register files. This 4-bank register files basically share the same “logic address”
but have different physical address. A pointer (high 2-bit of register address bus) is
employed for switching between banks.

This is very useful when we want to get a fast switch to Interrupt Service Routines
(ISRs) with only writing new value into a specific pointer. In order to manager to
implement such scheme within the VLIW $\rho$-VEX processor, multiple pages of register
files and a “pointer” which could point to different page of register files have to be
implemented. Before any design or implementation of this scheme, evaluation of resource consumption for multi-page register files should be done since the current 64 32-bit register file consumes too much resources on FPGA.

2.3.2 Microblaze RISC softcore

There are many RISC (Reduced Instruction Set Computer) based softcores available nowadays like Microblaze and OpenRISC, as introduced in [8] and [10] respectively. Microblaze is a 32-bit RISC softcore (synthesizable) processor that could be configured by the users to match the requirement of different applications. The basic architecture consists of 32 general-purpose registers, an Arithmetic Logic Unit (ALU), a shift unit, and two levels of interrupts. Since Microblaze has only one interrupt input, an interrupt controller is needed. Therefore, the service routine could “ask” the controller what device(s) caused the interrupt and act accordingly. When an interrupt is detected, (if interrupts are enabled) Microblaze stops executing the current code and jumps to address 0x00000010 to run an ISR program (for details, please consult [8]). Briefly, the interrupt handler has to:

1. first save the context (mainly g/p registers)
2. acknowledge the interrupt
3. service the interrupt
4. restore the context and execute rtdi (return from interrupt)
2.3.3 VLIW softcores

In the domain of VLIW softcore, few designs with a complete toolchain could be found in existing literature. The first VLIW softcore processor is Spyder[3][9]. Spyder marked the beginning of reconfigurable VLIW softcore processors but did not evolve extensively because the processor architecture as well as the compiler were designed from scratch. It implies that both the processor and toolchain have to be put effort on. So no interrupt system support could be found upon this softcore processor.

Several interrupt handling schemes on VLIW processors or DSPs are presented as in [7, 12]. In these papers, register files are divided into several parts to reduce the size of context to be stored and restored [7, 15]. All of these subset of registers will be marked as “live” or “dead”. Only “live” register groups will be included as context for saving. The generic idea of grouping subset registers is to let the compiler pass information of “live” registers to hardware (by an index indicating number of sets[7] or by a bit indicating if the other sets are all “dead”[15]), therefore small set of context switching can be done at run time. They are in principle reducing the size of context to be switched to minimize the interrupt latency. Nevertheless, all the mechanisms need the support of relative compiler and even processor architecture. This technique is not specialized for VLIW processors, we can also find such scheme in [15].

2.4 Conclusions

This chapter presents the background knowledge that is necessary for our thesis work. We are supposed to design an interrupt system and embed it into an extensible and reconfigurable VLIW softcore processor \(\rho\)-VEX with respect to the performance of interrupt latency, hardware consumption on a FPGA and requirements to the relative toolchain. The \(\rho\)-VEX processor can work as a CCU within MOLEN [17] polymorphic processor organization which bridges the gap between general-purpose processors and specific applications, or as a stand-alone processor.

Within an interrupt system, interrupt latency is an important figure to assess the performance of such system. The task of context switching is the major contribution to interrupt latency. The efficiency of storing and restoring context (how many context to store per clock cycle or time unit) and the size of context to be stored determine the performance of context switching. Another issue refers to interrupt priority. Not all the interrupts can interrupt the current task only if the interrupt is more urgent. In other words, only interrupt with higher priority should be treated immediately and interrupt the running task with a lower priority.

In the practical world, there are already many interrupt designs. The classic 8051 architecture treats register files as same virtual address but different physical address [1]. It is useful since 8051 has only 8 registers and 4 banks of them are also easy to implement. The Microblaze RISC softcore has a PIC (Programmable Interrupt Controller) which schedules interrupts one by one. For VLIW softcores, many approaches are also studied and implemented by others. However, these designs need the help of compiler which are not applicable in our case since we can not manipulate compiler at this moment.
Our design aims to embed an interrupt subsystem into the ρ-VEX processor in order to support interrupts on the processor. The interrupt system should be able to decide which instruction goes to decode stage and save the processor’s status to somewhere and restore the status at a later time. This chapter discusses all design details of the interrupt subsystem.

The organization of our subsystem is described in section 3.1. After that, interrupt scheduler is presented in section 3.2 and interrupt controller is detailed in section 3.3. Section 3.4 introduces the design for nested interrupts and section 3.5 discusses the way integrating our design into the ρ-VEX processor. Finally, section 3.6 presents the conclusion of this chapter.

3.1 System Organization

The interrupt subsystem is designed as an extension to the processors’s data path. It should communicate with the ρ-VEX processor but not corrupt the original pipeline. On the other hand, only interrupt pins should be seen as Input/Output (IO) ports from our processor. Any other intermediate or control signals between the interrupt subsystem and the ρ-VEX processor should not appear to users. This would also make our design and implementation simpler. Moreover, it implies that the interrupt subsystem should be located within the ρ-VEX processor, so that it can not only receive interrupts from outside world, but also be able to deal with any exceptions within the processor’s data path.

Figure 3.1 shows the structure of the ρ-VEX processor after we insert the module

![Figure 3.1: ρ-VEX organization with interrupter inserted](image)
Figure 3.2: Concept Scheme of Interrupter

“interrupter” into the processor. This interrupter gets input signals from interrupt pins and then sends control signals to fetch stage to re-schedule instructions for execution of an Interrupt Service Routine (ISR). In the mean time, necessary contexts will be saved by interrupter. When RFI (Return From Interrupt) instruction is decoded, a signal will be passed to interrupter indicating the end of ISR. Subsequently the context can be restored and the CPU is able to continue the original task from the same point where the ISR is invoked.

The interrupter can be divided into two parts: interrupt scheduler and interrupt controller. Interrupt scheduler mainly takes responsibility to receive interrupt signals from different interrupt sources, re-schedule respective tasks into a task queue, enables the interrupt request to interrupt controller, in case that the priority of the requested task is higher than the current one. The interrupt controller’s main job are: to receive interrupt request signal from interrupt scheduler, make context switch, invoke the execution of an ISR and resume the foreground program after the execution of ISR. When an ISR is finished, a ‘clear’ signal will be sent to interrupt scheduler thus other lower priority interrupt requests could be enabled. Figure 3.2 is a concept scheme of the interrupt module. This presents an overview of the structure of “interrupter”.

3.2 Interrupt Scheduler

The interrupt scheduler works as a “monitor” for the status of the running program. It knows what task is executing and what requests is issuing to ρ-VEX processor. Therefore, tasks could be scheduled to execute on the processor by their priorities via the interrupt scheduler.

3.2.1 Dataflow

There are two inputs for the interrupt scheduler: interrupt signals (which sets corresponding interrupt-flag(s)) from outside and clears interrupt-flag signal from interrupt controller. Normally, the interrupt signals adds tasks into a task queue and the clear signals removes them from the task queue. The scheduler decides whether to send the
interrupt to the following connected component - interrupt controller by the priorities of tasks (interrupt vectors). Only when an interrupt with higher priority is received, or a higher priority task is finished and a there is still a task waiting for execution in the task queue, an interrupt signal will be passed to the interrupt controller. Figure 3.3 illustrates the dataflow in the interrupt scheduler. The interrupt flag list records all requests that need to be answered, which is located in the interrupt vector table.

### 3.2.2 Interrupt Vector Table

The interrupt vector table records all information that is necessary for scheduling tasks. This information includes interrupt vectors (type of interrupts) and their corresponding priorities, interrupt flag list which shows the status of each interrupt request at run-time, ISR address (could be pre-defined or customer defined while compilation) and interrupt enable bit that can mask certain interrupts.

Table 3.1 shows a set of possible values that can be stored in the vector table. In our design, we assign interrupt vector zero and the lowest priority to the foreground program. This means that interrupt vector zero can not be used by a user. Therefore, when there is no interrupt, the default status (foreground program is running) can be obtained from the vector table which makes our design more generic.

<table>
<thead>
<tr>
<th>Interrupt type</th>
<th>vector</th>
<th>ISR address</th>
<th>priority</th>
<th>flag</th>
<th>enable bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreground program</td>
<td>000</td>
<td>0x00000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UART transfer</td>
<td>001</td>
<td>0x00010000</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>UART receive</td>
<td>010</td>
<td>0x00100000</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Timer</td>
<td>011</td>
<td>0x01000000</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Exceptions</td>
<td>100</td>
<td>0x010000100</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
3.2.3 Scheduling Issues

The scheduler is designed pipelined fashion so that the interrupt requests can be received at any time and multiple interrupts can come in at the same clock cycle without losing any of them. When an interrupt is signaled to the interrupt controller, the scheduler will keep this information in the task queue. Only when a ‘clear’ signal from the interrupt controller is captured, this information will be removed.

Since we do not have instructions that can write the enable bit in the vector table, the default status of all interrupt vectors are enabled. We should be able to control this bit in a later version of ρ-VEX in order to support atomic operations. This bit is designed here only for generic purpose.

3.3 Interrupt Controller

The interrupt controller controls the process of storing and restoring the processor’s status and execution of Interrupt Service Routines (ISRs). The control logic determines the length of interrupt latency and consequently the performance of our interrupt system. In this section, the design scheme of Finite State Machine (FSM) and all of the four mechanisms that deal with context switching are presented.

3.3.1 Design Logic

In contrast to the interrupt scheduler, the interrupt controller can not be designed pipelined because its state of each clock cycle is determined by its previous state. Therefore, we designed the interrupt controller as one Finite State Machine. The input (i.e. interrupt request or Return From Interrupt) signal determines the next state based on the current one. Besides, an interrupt queue is designed as to record information of ISR address, return address and interrupt vector received from interrupt scheduler along with interrupt request signal. Figure 3.4 shows the FSM of interrupt controller:

- **IDLE**: In this state, no ISR is running (i.e. interrupt vector 0 - foreground program is running), the interrupt controller is waiting for interrupt requests. When an interrupt request signal comes, it initializes context switching and then goes to STORE state. We assume that no ‘rfi’ (Return From Interrupt) signal will come in at this state as no ISR is running.

- **STORE**: In this state, the interrupt controller stops the increment of the program counter. It stops getting instructions from the foreground program and stores the current status into memory. Any other interrupt request happens in this state will lead to nothing but update the data in interrupt queue in the sense that the new return address will become the current ISR address. When the context switching finishes, it automatically turns to RUN state. We assume that no ‘rfi’ signal will be captured in this state as no ISR is running.

- **RUN**: This state indicates that there is already at least one ISR is running (or wait-
3.3. INTERRUPT CONTROLLER

1.7

Figure 3.4: Finite State Machine of interrupt controller

ing for execution), any other interrupt request will raise another context switching and jump to STORE state again. Only if ‘rfi’ is detected then it will go to LOAD state.

• LOAD: Basically, this state does the similar job as STORE state only with the difference that it loads data from memory to register files rather than that it stores data into memory. If interrupt request happens in this state, it goes to RUN state immediately since no context switching is needed. It goes back to IDLE mode when context switching is done. We assume that no ‘rfi’ signal will be captured in this state as no ISR is running.

3.3.2 Method for Direct Context Switching (DCS)

This method is an intuitive one. It basically means that we directly connect general-purpose registers, branch registers and data memory to interrupt controller and let the controller do storing and restoring while the context is switched. Within this method, extra multiplexes are introduced which can select the path from either pipeline or interrupt controller to the register files and data memory. Figure 3.5 shows the block scheme of the interrupter of this method:

• “Interrupt in” and “RFI signal in” indicate request of interrupt and finishing of ISR respectively.

---

\(^1\)This interrupt request comes from interrupt scheduler which means the interrupt priority has been checked and verified higher than the current tasks’. So it should surely interrupt the current ISR.

\(^2\)At this point, necessary context of foreground program is saved already and not changed, so we don’t need to store them again.
• “Halt inst.” stops execution of instruction stream and then waits for the “Inst. stopped ack” which meant that instruction has been stopped. After the instruction stream is stopped, we can then do the context switch.

• “PC value in” is the current pc value that is jumped to after execution of ISR program and “ISR address” is the instruction memory address at where the ISR program resides.

• All the other signals deal with context switching.

While responding to an interrupt, the actions that the controller might take are as follows:

1. Interrupt signal received, then “halt inst.” signal is set and sent to decoder stage. This will make the FSM switch from IDLE state to STORE state.

2. This “halt” signal will propagate till the Writeback stage, and the Writeback stage will issue an “Inst. stopped ack” signal to controller.

3. After receiving the acknowledge signal, the controller will start to move data from register files to data memory (context storing).

4. When completing the context storing, the interrupt controller will release the “halt inst.” signal and in the mean while, inject the ISR address into fetch stage. Consequently, it goes to RUN state.

5. Only in the RUN state, a ‘rfi’ signal will be captured by the interrupt controller. When this happens, it will again set “halt inst.” signal to decoder stage and turn to LOAD stage.

6. Whenever the interrupt controller is acknowledged by “Inst. stopped ack”, it will copy data from data memory to register files (context restoring).

7. Finally, when all the data are restored, the previous program counter will be restored and signal “halt inst.” will be released. Then the FSM goes to IDLE state again.
3.3. Method for Hardware Instructions Context Switching (HICS)

Context switching is very important to our interrupt subsystem as it contributes most to the interrupt latency. When the ISR is small, which is usually the case, the overhead dominates the time spending on interrupt response and therefore, it largely degrades the performance of processor. Thus, optimization should be applied to reduce the overhead of context switching.

3.3.3.1 Reduce overhead with “forwarded” pipeline

In order to make sure that correct data are transferred between registers and data memory, we should always wait for all the instructions to be executed and then move the data. When the interrupt handler sends ‘stop’ signal to pipeline, we usually need to wait for 5 clock until the ‘acknowledge’ signal is given which indicates that all instructions are done. During these 5 cycles, no instructions can be pushed into the pipeline and thus resources are wasted.

Instead of directly handle the data transferring, we could push extra instructions into pipeline and let the pipeline perform the context switching. Thus, we make the pipeline always busy and therefore reduce the interrupt latency. However, since these instructions will read data from register files and write them into data memory, it will only work on a forwarded pipeline so RAW hazard can be avoided. Another advantage of this method is that it saves us some hardware consumption (connections between interrupter and register files and data memory). Following are the issues that should be taken into account when considering this optimization:

a. Instruction formulation

Instructions can be inserted into fetch stage by hardware or read from instruction memory by software. For the later method, a set of instructions for the ISR are formulated by compiler and written into some specific address of instruction memory. This works with ISR as a specific function. If we do context switching by software, the only thing we have to do is to write assembly code and let toolchain generate instructions for us. In other words, it only moves the complexity from hardware to software but not simplifies our design. For the hardware instruction generation design, the hardware should be aware of how to somehow “compile” assembly code itself. Table 3.2 are layout of instructions for context switching of current version of r-vex. If different instruction set is used (etc. rearrangement of the opcode space), we might need to modify some of the instructions to make it work.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opcode IS sour reg dest reg Immediate LF</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM_STW</td>
<td>0010101 00 001011 000001 000001000 10</td>
</tr>
<tr>
<td>MEM_LDW</td>
<td>0010101 00 001010 000000 000000100 10</td>
</tr>
<tr>
<td>ALU_MFB</td>
<td>0101110 00 000111 000000 000000001 10</td>
</tr>
<tr>
<td>ALU_CMPNE</td>
<td>1010011 01 000010 000100 000000000 10</td>
</tr>
</tbody>
</table>
Example explanation:

**MEM_STW**  Move from gr to dm (STORE)
\[ gr(001011) = gr(11) \rightarrow dm(gr(1) + 0b1000/4) = dm(0 + 2) \]

**MEM_LDW**  Move from dm to gr (LOAD)
\[ gr(001010) = gr(10) \leftarrow dm(gr(0) + 0b100/4) = dm(0 + 1) \]

**ALU_MFB**  Move br to gr
\[ gr(000111) = gr(7) \leftarrow br(gr(0) + 1) = br(0 + 1) \]

**ALU_CMPNE**  Move gr to br
\[ \text{if } gr(100) = gr(4)! = 0 \text{ then } br(0b10) = br(2) = 1 \text{ else } 0 \]

b. **Instruction sequence**

It is worth to notice that we are not able to move data from branch register to data memory directly by applying this mechanism. Thus, we should introduce at most 8 more instructions to move data from branch register to general register and then move these value to data memory. These extra instructions would destroy original value reside in the general registers, so we should have to save these data in general register in the very beginning. Overall, the logic that being applied to do context switch is as:

1. Move data in 1st general register to data memory.
2. Move data in 2nd general register to data memory and move data in 1st branch register to the 1st general register at the same time.
3. repeat step 2 but save the next data in general register and branch register until all branch register are saved.
4. Move data in the rest general registers to data memory.
5. Move data reside in the 1st to the total number of branch register of general register to data memory.

Figure 3.6 shows the above sequence. When retrieving old status in data memory, similar logic are considered.

3.3.3.2 **Reduce overhead regarding to the size of context**

In order to make it absolutely consistent when we continue to process the current job at a later time after executing ISR, we require that all the context is saved into data memory and retrieve the data back afterwards. This is not always necessary as not all registers are used for storing temporary values at run time. At this point, live range checking of registers might become handy to claim a small set of registers that should be included as the context to save and retrieve back. This live range checking mechanism needs the support of the compiler and sometimes of the embedded processor architecture [15] since

\[ ^{3}\text{In later version, “MFB” might be replaced by others} \]
3.3. INTERRUPT CONTROLLER

Figure 3.6: Instruction sequence that doing context switch

hardware is not able to be aware of whether the register might be read at certain later time.

Nevertheless, the hardware is able to remember which register has been accessed. If we simply assume it is always living afterwards, we can then only manage the living registers. This “remembering” would also save us considerable overhead as not all the registers are used for storing context during run time [6] especially for some small programs.

Normally, the compiler would assign the registers sequentially and registers will be reused after the old values are dead. To simplify our design, only the maximum index of register that being accessed is remembered and determines the number of registers that participate in the context switching. By checking register usage [6] of r-vex compiler an assembler, we can individually deal with the last register ($r0.63) that act as link register.

3.3.3.3 Simplified Interrupter

Since the pipeline can move data between register files and data memory, we might be able to make use of this and therefore reduce the consumption by means of removing the multiplexes we used in the first method. Beside this, a “monitor” which records the maximum index of registers at run-time is introduced to the system as to reduce the size of context switching. In this method, instructions are generated in the interrupt controller and are inserted into pipeline to do context switching. The block scheme of this modified interrupt module is shown in figure 3.7.

Figure 3.7: Interrupt block scheme of DICS method
3.3.4 Method for Software Instruction Context Switching (SICS)

Software instructions that move context means that we give the task of storing and restoring context to software. It will save us hardware consumptions with respect to hardware instructions moving context method. For the software method, a set of instructions are formulated by compiler and written into some specific address of instruction memory. When interrupt happens, any ISR should always call this specific piece of code to first do the context switching and then do its own job. Since this mechanism includes two more extra branch jump (CALL and RETURN), it gives us four more cycles penalty of overhead (2 for instructions, 2 for branch jump penalties) for each context switching. Another scenario is we put these piece of code for storing (restoring) context before (after) the functionality body of ISR thus to save the “branch jump”, while the size of our code might explode if the interrupts are too many.

Figure 3.8 shows the block scheme of interrupter of this method. It becomes much more easier compared to the hardware method. However, we lose the possibility of only switching subset of registers at run-time since all instructions for context switching is fixed after compilation. Moreover, nested interrupts would be difficult to support unless we assign different data memory space that was employed for storing context to interrupt vectors with different interrupt priorities in advance. To sum up, in theory, by this method, we reduced the complexity and consumption of hardware at a price of increasing the complexity of toolchain and consumption of instruction memory.

![Diagram](image)

Figure 3.8: Interrupt block scheme of SICS method

The compiler (assembler) should generate instructions that switches context (these should also be in the order as figure 3.6 illustrates) and put these instructions (or make them as normal functions to be called) before and after functional ISR programs. Since “CALL” instruction will flush data into link register, so data in the link register (&r0.63) must be stored before calling the “storing” function and restored after calling the “restoring” function. In the “storing” and “restoring” functions, there is no need to deal with the link register.

3.3.5 Method for Page-able Register Files (PRF)

Rather than storing and restoring context, we simply use other page of fresh register files for ISRs to use thus to significantly reduce interrupt latency. When the process of switching context is required, the only thing we have to do is to switch to the other page of register files. However, before any design or implementation of this scheme, evaluation of resource consumption for multi-page register files should be done since the current 64
3.3. INTERRUPT CONTROLLER

32-bit register file consumes too much resources on FPGA and we should always make our design feasible in practice.

Table 3.3 shows synthesis results for multiple register files which implemented in a normal way. It is obviously that multiple pages of register files are not applicable base on the current way for implementing the register files. Therefore, we should implement register files in other way that keep hardware consumption within a reasonable figure.

<table>
<thead>
<tr>
<th>Page(s)</th>
<th>Slice Registers</th>
<th>Slice LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2016 (0%)</td>
<td>9799 (6%)</td>
</tr>
<tr>
<td>2</td>
<td>4064 (1%)</td>
<td>23698 (15%)</td>
</tr>
<tr>
<td>4</td>
<td>8160 (2%)</td>
<td>47741 (31%)</td>
</tr>
</tbody>
</table>

3.3.5.1 Block RAM Resources

The block RAM in our target Virtex-6 FPGAs stores up to 36K bits of data and can be configured as either two independent 18 Kb RAMs, or one 36 Kb RAM. Each 18 Kb block RAM can be configured as a 16K x 1, 8K x 2, 4K x 4, 2K x 9, 1K x 18 or 512 x 36 in simple dual-port mode[5].

In our case, the size of general-purpose register files is 64 x 32, so we are able to implement 8 x 64 x 32-bit register files (8 pages of register files) on the FPGA. Normally, the level of interrupts (interrupt priority level) can not be so many, so within this BRAM implementation, page-able register files method becomes applicable for our design.

3.3.5.2 Page-able design itself

Figure 3.9 depicts the interrupter block scheme of page-able register files method. Multi-page register files are implemented as BRAMs so the hardware consumptions are limited in a considerable use percentage. Therefore, this method is also implemented in our design. When implementing this method, the following issues are worth mention:

- Before switching page of register files, we should make sure the last instruction should write correct data into correct address (if the last instruction writes data into
CHAPTER 3. DESIGN

general-purpose register). So we should delay at least 4 cycles for switching pages right after jumping to target ISR program. We assume that first 4 instructions in ISR will not read data from registers.

- After switching back to the previous page of register files, the next instruction might be reading data from register files. So in this case, we do not delay such 4 cycles otherwise the data might be wrong since the reading address is not as expected. We also assume that the last 4 instructions in ISR will not write data into register files.

3.4 Interrupt Nesting

Preemption of low priority interrupt by another high priority interrupt leads to interrupt nesting. When a nested interrupt happens, multiple sets of context rather than only one set should be stored by the controller. In order to handle this circumstance, a series of hardware queues should be introduced into our interrupt system in order to record necessary information for every interrupt vector that being nested.

Figure 3.10 depicts an example how nested interrupt can be handled and what information should be logged during the process of context switching for the method of hardware instruction context switching. We assume that interrupt 2 has the higher priority than interrupt 1 in this picture. The necessary information includes the size of general-purpose registers and branch registers that being stored and the return address of each interrupt.

Table 3.4 shows that how we record relative information into queues that ensure the correctness of context switching. The first column is the state when interrupts happen while the rest columns are the data should be written into queues.

<table>
<thead>
<tr>
<th>state</th>
<th>return address</th>
<th>ISR address</th>
<th>size of GR</th>
<th>size of BR</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>next PC</td>
<td>addr. of issued ISR</td>
<td>max-index GR</td>
<td>max-index BR</td>
</tr>
<tr>
<td>STORE</td>
<td>last ISR address</td>
<td>addr. of issued ISR</td>
<td>NO need to save</td>
<td>NO need to save</td>
</tr>
<tr>
<td>RUN</td>
<td>next PC</td>
<td>addr. of issued ISR</td>
<td>max-index GR</td>
<td>max-index BR</td>
</tr>
<tr>
<td>LOAD</td>
<td>last return address</td>
<td>addr. of issued ISR</td>
<td>NO need to save</td>
<td>NO need to save</td>
</tr>
</tbody>
</table>

3.5 Integration with the $\rho$-VEX processor

Our interrupter (modular of interrupt subsystem) should be embedded into the $\rho$-VEX processor and thus make the processor interruptible. The original processor does not

---

4This is reasonable since we normally should initialize (write) variables before we can use (read) it.

5This is also reasonable since after leaving ISR, all data in ISR will be dead. So we don’t write at the end of ISR because it is useless.

6the other method to handle nested interrupt would be easier than this method since not only the nested level but the number of registers being saved at each level should be concerned in this method.
introduce any interface that could communicate with our interrupt subsystem, so modifications should be made apart from the interrupt system itself.

### 3.5.1 Fetch Stage

Fetch stage is the first stage of pipeline. It determines what instruction enters into pipeline and will be executed by the following stages. The interrupter should be able to control this stage to either push ISR address and return address into pipeline or push context-switching instructions directly. Besides, it should also be able to stop the program counter so that the address the PC stopped at could be captured by interrupter.

### 3.5.2 Decode Stage

In decoder stage, the opcode of each instruction is checked so “RFI” instruction can be detected in this stage. Whenever “RFI” is found, a signal should be set to interrupter in order to do context restoring. In the meanwhile, the next instruction should be flushed as a branch instruction would do.

### 3.6 Conclusions

This chapter presents the design of our interrupt system. The top module (interrupter) of this system consists of two sub-modules: interrupt scheduler and interrupt controller. Interrupt scheduler is designed as a pipeline thus no interrupts will be lost and they will
be scheduled to a task queue and interrupt controller according to their priorities. The interrupt controller is designed as a Finite State Machine (FSM) since the current state is always depending on the previous state and we are not able to finish context switching within single clock cycle.

Four mechanisms to do context switching are presented in this chapter so one could understand how we reduce the interrupt latency and/or hardware consumption through the process of design. The first method is an intuitive one that directly moves context while an interrupt comes. The second method reduces the interrupt latency by reducing the size of registers to be stored and restored. Upon a “forwarded pipeline”, the interrupt latency could be further decreased by this method. After we move the complexity of context switching from hardware to software, we get the third mechanism. This mechanism is very simple to implement from the hardware point of view but would require the toolchain to produce softcode that deal with context switching. The last method results in the fast context switching compared to all other methods at a price of certain special hardware utilization: block memory on a FPGA.
In this chapter, all implementation details the designs that were presented in chapter 3 are discussed. With the help of this chapter, one is able to understand how we map our design into an implementation and might be able to extend or improve it at a later time.

This chapter is organized as follows. Section 4.1 presents the top-down implementation mechanism. Subsequently top module of our interrupt system is discussed in section 4.2. Section 4.3 and section 4.4 present the implementation of interrupt scheduler and interrupt controller respectively. Section 4.5 discusses how we can adjust the original pipeline to support our interrupt system. Finally, section 4.6 concludes this chapter.

4.1 Top-down Implementation

A top-down implementation is utilized for our interrupt system. The first goal is to structure the top-level module of the interrupt system and adjust the interface between this module and the $\rho$-VEX processor. After a simple test of this module, we implement the interrupt scheduler and the interrupt controller separately. Within these sub-components, we also implement in top-down fashion. This made it easy to debug and maintains the system’s compatibility.

The $\rho$-VEX processor and our interrupt system is to be parametric, so the top-down implementation would be very suitable for this design, as we consider the parametric nature from the very beginning. Therefore, we do not worry if we set up the interrupt system but it does not apply to the processor or not the way what we expected. For different way to do contexts switching, we only add different implementations into the interrupt module, which would lead to different versions of interrupt system.

4.2 Modules of the Interrupter System

The module interrupter is the top component of our interrupt subsystem. It directly communicates with the other components in the processor. Figure 4.1 shows all necessary I/O ports for transferring signals between the interrupter and the original processor’s pipeline. Signal “interrupt in” is the interrupt request, “RFI signal” comes from branch unit, “PC value” indicates the current value of program counter which can be recorded as return address. All these signals go to the interrupt scheduler. “ISR/RFI address” will be sent to the fetch stage to change the piece of code to be executed and “Control signals” are set of signals that controls the necessary contexts switching.

From the figure 4.1, we already known what should be implemented in our design. In the following sections, all details are discussed.
4.3 Interrupt Scheduler

The interrupt scheduler is the first component that deals with interrupt requests for the interrupter. It has the knowledge of what program is running on processor and how to schedule tasks according to requests. All implementation details of this component are presented in this section.

4.3.1 Pipeline Organization

The interrupt scheduler is designed pipelined and consists of three stages: absorbing, scheduling and updating. In absorbing stage, interrupt signals from interrupt pins and interrupt done signal from interrupt controller will be collected and written into an “interrupt vector table”. The vector table is visible to scheduling stage, where a new task can be scheduled and a service request can then be generated to interrupt controller. The task queue keeps the history of scheduling results and is updated in the last stage. Figure 4.2 shows the organization of this pipelined interrupt scheduler and signal flow is also presented in this picture.
4.3. INTERRUPT SCHEDULER

4.3.2 Absorbing Stage

The absorbing stage is the first stage of interrupt scheduler. Two inputs go to this stage: interrupt and interrupt done. An interrupt signal will set interrupt flag of respective interrupt vector while interrupt done signal cancels it. Whenever an interrupt done signal comes in, it will be propagated as a ‘clear’ signal to updating stage in order to clear the current active task from task queue. Table 4.1 shows actions that should be done for every combination of the input signals. So it is implemented as a look-up table.

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Interrupt done</th>
<th>Set flag</th>
<th>Clear flag</th>
<th>Propagate ‘clear’</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>√</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3.3 Scheduling Stage

In this stage, the vector table (see table 3.1) will be checked according to the state of the flag bit. Only when the interrupt flag bit is set (logical high), the respective interrupt priority will be selected and compared with others. Thus, we get the highest priority of “active” vector after we checked the flag bit of every interrupt vector. In the end, we compare this highest priority to the priority of current running task (foreground program can be seen as default “ISR” which has priority of zero). If it is higher, we update the priority of current running task and issue an interrupt request to interrupt controller. If it is lower (which only happens when the running task was finished and respective interrupt flag was cleared\(^1\)), and the vector of request interrupt does not appear in task queue\(^2\), we issue interrupt request to the interrupt controller. We always update the priority of current running task. Table 4.2 lists the scheduling logic of what is discussed above. (V.T. = Vector Table)

<table>
<thead>
<tr>
<th>Priority comparison</th>
<th>Update states</th>
<th>Issue interrupt request</th>
</tr>
</thead>
<tbody>
<tr>
<td>V.T. highest vs. running task</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

The scheduling stage will generate interrupt request for the following component (interrupt controller), so only the scheduling stage and absorbing stage will contribute to the interrupt latency. Besides, ‘clear’ signal will be propagated to the next stage.

---

\(^1\)Which means a new coming request with lower priority happens when the higher priority task is running.

\(^2\)This means the task has never be scheduled into the task queue.
4.3.4 Updating Stage

This stage is designed for extensibility at a later time. In this stage, a task queue can be updated to indicate the running task and the tasks that are waiting for execution. Table 4.3 shows the logic of how to update the task queue.

Table 4.3: Logic for updating task queue

<table>
<thead>
<tr>
<th>Interrupt request</th>
<th>‘Clear’ signal</th>
<th>Queue pointer</th>
<th>Queue content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>maintain</td>
<td>maintain</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>increase</td>
<td>update</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>decrease</td>
<td>remove</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>maintain</td>
<td>update</td>
</tr>
</tbody>
</table>

4.4 Interrupt Controller

The interrupt controller is the main component that contributes to interrupt latency and consequently determines the overall performance of our interrupt sub-system. In this section, the controller itself and all implementation methods are detailed. These methods are Direct Contexts Switching (DCS), Hardware Instruction Contexts Switching (HICS), Software Instruction Contexts Switching (SICS) and Page-able Register Files (PRF).

4.4.1 Finite State Machine

The interrupt controller is designed to a Finite State Machine (FSM) so it is implemented first. Subsequently each method to do contexts switching is added into this FSM to realize four different ways of implementation.

The FSM has 4 states (IDLE, STORE, RUN, LOAD) and 4 inputs (int_en, rfi, cur_addr, int_addr), states switching is determined according to these 4 inputs. Table 4.4 is derived from the FSM design scheme that is shown in figure 3.4. This control logic makes sure that no interrupts will be lost and prevents the control from deadlock. How to save the signals that control the instruction flow is listed as follows:

**IDLE** int_en = 1
- Save cur_addr as return address, int_addr as ISR address.

**STORE** int_en = 1
- Save previous ISR address as return address, int_addr as the new ISR address.

**RUN** int_en = 1
- Save cur_addr as return address, int_addr as ISR address.

**RUN** int_en, rfi = 1
- Save int_addr as ISR address, keep the return address.

**LOAD** int_en = 1
- Save int_addr as ISR address, keep the return address.
4.4. INTERRUPT CONTROLLER

Table 4.4: Control logic of the FSM in the interrupt controller

<table>
<thead>
<tr>
<th>State</th>
<th>Trigger</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>int. request (int_en = 1)</td>
<td>goto STORE</td>
</tr>
<tr>
<td>STORE</td>
<td>int. request (int_en = 1)</td>
<td>new ISR address</td>
</tr>
<tr>
<td></td>
<td>CS done</td>
<td>goto RUN</td>
</tr>
<tr>
<td>RUN</td>
<td>int. request (int_en = 1)</td>
<td>goto STORE</td>
</tr>
<tr>
<td>RUN</td>
<td>RFI raised (rfi = 1)</td>
<td>goto LOAD</td>
</tr>
<tr>
<td></td>
<td>int. &amp; RFI (int_en,rfi = 1)</td>
<td>goto RUN</td>
</tr>
<tr>
<td>LOAD</td>
<td>int. request (int_en = 1)</td>
<td>goto RUN</td>
</tr>
<tr>
<td>LOAD</td>
<td>CS done</td>
<td>goto IDLE</td>
</tr>
</tbody>
</table>

4.4.2 Control Logic for DCS method

DCS method refers to Direct Contexts Switching method. Since it will directly read data from register files and write them to data memory (context storing), it has connections to both register files and data memory (etc. enable write signal, write/read address/data signal). A signal called halt inst can be used to halt the pipeline. Thus register files and data memory can then be accessed by the interrupt controller. Table 4.5 shows the control logic for DCS method. Apart from contexts switching signals, other signals are not listed.

Table 4.5: Signal control for DCS method

<table>
<thead>
<tr>
<th>State</th>
<th>Trigger</th>
<th>halt inst</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>int. request (int_en = 1)</td>
<td>set</td>
<td>goto STORE</td>
</tr>
<tr>
<td>STORE</td>
<td>inst. stopped ack = 1</td>
<td>clear</td>
<td>goto RUN</td>
</tr>
<tr>
<td></td>
<td>CS done</td>
<td>set</td>
<td>goto STORE</td>
</tr>
<tr>
<td>RUN</td>
<td>int. request (int_en = 1)</td>
<td>set</td>
<td>goto LOAD</td>
</tr>
<tr>
<td>LOAD</td>
<td>inst. stopped ack = 1</td>
<td>clear</td>
<td>goto IDLE</td>
</tr>
<tr>
<td>LOAD</td>
<td>CS done</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4.3 Control Logic for HICS method

HICS method refers to Hardware Instruction Contexts Switching method. The control signals for contexts switching including ISR address (int_address_o), context-switch instructions (instr_cs), context-switch select (instr_cs_src).

When an interrupt is generated, the first context-switch instruction will be triggered on instr_cs and the select signal instr_cs_src will be set, so at the next clock cycle, the instruction from instr_cs will go to the pipeline rather than the next instruction of original program. This select signal will also stop the current program counter. Therefore, it can be remembered by interrupt controller. After finishing context switching, the ISR address will be pushed into the fetch stage in order to start executing the ISR
that resides in instruction memory. When restoring contexts, similar logic is required. Figure 4.6 shows the logic that controls the output signals.

<table>
<thead>
<tr>
<th>State</th>
<th>Trigger</th>
<th>instr_cs</th>
<th>instr_cs_src</th>
<th>int_address_o</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>int. request</td>
<td>set</td>
<td>set</td>
<td>clear</td>
<td>goto STORE</td>
</tr>
<tr>
<td>STORE</td>
<td>int. request</td>
<td>set</td>
<td>set</td>
<td>ISR address</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>CS done</td>
<td>clear</td>
<td>clear</td>
<td>clear</td>
<td>goto RUN</td>
</tr>
<tr>
<td>RUN</td>
<td>int. request</td>
<td>set</td>
<td>clear</td>
<td>clear</td>
<td>goto STORE</td>
</tr>
<tr>
<td>RUN</td>
<td>RFI raised</td>
<td>set</td>
<td>set</td>
<td>clear</td>
<td>goto LOAD</td>
</tr>
<tr>
<td>RUN</td>
<td>int. &amp; RFI</td>
<td>set</td>
<td>clear</td>
<td>ISR address</td>
<td>goto RUN</td>
</tr>
<tr>
<td>LOAD</td>
<td>set</td>
<td>set</td>
<td>Return address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>int. request</td>
<td>set</td>
<td>ISR address</td>
<td></td>
<td>goto RUN</td>
</tr>
<tr>
<td>LOAD</td>
<td>CS done</td>
<td>clear</td>
<td>clear</td>
<td>clear</td>
<td>goto IDLE</td>
</tr>
</tbody>
</table>

Instructions for context switching are activated at run time due to the reason that the number of contexts and position storing the contexts are dynamically changeable: The number of contexts is the maximum index of registers \( \max_{g(b)r} \) that are used in the program, so this could change at run-time; When preemptive nested interrupt happens, we should store the context to another address of data memory, rather than the default one. Otherwise, it will corrupt the information that the lower interrupt will retrieve.

When a nested interrupt happens, multiple set of context should be stored into the data memory. In order to determine which address of the data memory a context should be stored, a signal \( \text{cnt} \) is introduced. When storing contexts, we increase \( \text{cnt} \) and vice versa. This number basically shows how many contexts have been stored and where the next context should be stored at run time. Another signal \( \text{ilcnt} \) is introduced as to indicate how many contexts are storing at the current interrupt level (or vector). It is useful when interrupt request happens at “LOAD” state: it will reset \( \text{cnt} \) value.

Table 4.7 illustrates the sequence of generated instructions when storing the contexts. When restoring contexts, the logic are basically the same.

This component is designed as to dynamically determine the size of contexts for switching. It is basically a multi-input comparator that compares address bus of registers in the 4 syllables. Figure 4.3 shows how this component work in the whole system. When the interrupt happens, it would take at least 4 clocks to finish all the instructions in the pipeline and update the “final” maximum index of registers that being employed. Therefore we should delay at least 4 clocks to check the size of contexts to store. In our design, we define the minimum size of registers to be switched is 4 (both general-purpose registers and branch registers).
Table 4.7: Instructions generated when storing context

<table>
<thead>
<tr>
<th>Step</th>
<th>Syllable 3</th>
<th>Syllable 2</th>
<th>Syllable (1,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>gr(0) to dm(cnt)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>gr(1) to dm(cnt+1)</td>
<td>br(0) to gr(0)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>....</td>
<td>....</td>
<td>0</td>
</tr>
<tr>
<td>max_br</td>
<td>gr(step) to dm(cnt+step)</td>
<td>br(step-1) to gr(step-1)</td>
<td>0</td>
</tr>
<tr>
<td>max_br+1</td>
<td>gr(step) to dm(cnt+step)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>....</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>max_gr</td>
<td>gr(0) to dm(cnt+step)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>....</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>max_(g+b)r</td>
<td>gr(max_br) to dm(cnt+step)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>max_(g+b)r+1</td>
<td>gr(63) to dm(cnt+step)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.3: How Register Monitor is attached into $\rho$-VEX

4.4.4 Control Logic for SICS method

SICS (Software Instruction Contexts Switching) method simplify the HICS method in terms of moving the task of contexts switching from hardware to software. Due to the reason we let softcode to store and restore data in the register files, hardware would not concern about switching the contexts, but just jump to ISR program and return back. So for this method, we do not add anything into the FSM.

4.4.5 Control Logic for PRF method

For this PRF (Page-able Register Files) method, we have several pages of register files implemented on FPGA board. When contexts switching is required, the only thing to do is to switch to a different page of register files. For example, if we have 8-bit address size of register files, the left-most 2-bit is working as page number, and the rest is for read and write. For the pipeline, only the right-most 6-bit is visible, so no page-inform is passed to pipeline. The interrupt controller determines which page is using for storing
contexts at run-time. Table 4.8 lists the signal control for this page-able register files method. When implementing this method, the following issues are worth mention:

1. Before switching page of register files, we should make sure the last instruction should write correct data into correct address (if the last instruction writes data into general-purpose register). So we should delay 4 cycles for switching pages right after jumping to target ISR program. We assume that first 4 instructions in ISR will not read data from registers.3

2. After switching back to the previous page of register files, the next instruction might be reading data from register files. So in this case, we do not delay such 4 cycles otherwise the data might be wrong since the reading address is not as it expected. We also assume that the last 4 instructions in ISR will not write data into register files.4

3. The “RFI” instruction will write data into register files(stack) as it is overloaded with “RET”, we do not want it happen (since we do not “CALL” the function), so modification is made in the decode component.5

<table>
<thead>
<tr>
<th>State</th>
<th>Trigger</th>
<th>Page number</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>int. request (int_en = 1)</td>
<td>increase</td>
<td>goto STORE</td>
</tr>
<tr>
<td>STORE</td>
<td>none</td>
<td></td>
<td>goto RUN</td>
</tr>
<tr>
<td>RUN</td>
<td>int. request (int_en = 1)</td>
<td>increase</td>
<td>goto STORE</td>
</tr>
<tr>
<td>RUN</td>
<td>RFI raised (rfi = 1)</td>
<td>decrease</td>
<td>goto LOAD</td>
</tr>
<tr>
<td>LOAD</td>
<td>none</td>
<td></td>
<td>goto IDLE</td>
</tr>
</tbody>
</table>

4.5 Modifications for the original ρ-VEX processor

The original ρ-VEX processor does not have interface that could communicate with interrupter, so we should also adjust some of the components in the pipeline thus make the whole system interruptible.

4.5.1 Fetch Stage

Figure 4.4 is the updated fetch stage. instr_sel could select the foreground program (instr_m) or context switch instructions (instr_cs) coming from interrupter. Whenever context switch instruction stream is selected, PCWrite signal will be updated

---

3This is reasonable since we normally should initialize (write) variables before we can use (read) it.

4This is also reasonable since after leaving ISR, all data in ISR will be dead. So we don’t write because it is useless.

5All other versions of interrupt handler will restore context into register files after “RFI” instruction thus does not make difference.

6The picture is shown for hardware instruction contexts switching method, for other methods, we only need to remove the right most multiplexer.
and stop the current foreground program. Meanwhile, ISR address (also known as int_address_o from interrupter) will be pushed into pipeline and wait for execution.

![Figure 4.4: Fetch stage that could receive context switch instructions](image)

4.5.2 “RFI” Signal

When RFI (Return From Interrupt) instruction is decoded, the instruction stream should be stopped immediately, and it should inform interrupter to do the following context switching. Therefore rfi signal is introduced for function of CTRL_RFI in branch unit. It will stop the instruction stream for one clock and flush the next instruction fetched from instruction memory (working as other branch instructions). Then interrupter will take over to do following actions.

4.5.3 Halt the pipeline

For the method of direct contexts switching, we should somehow halt the pipeline for a while to do contexts switching, so signal of “halt inst.” and “inst. stopped ack” is introduced to the pipeline. The decoder stage receives the “halt” signal from interrupter, and will propagate it to the writeback stage. In writeback stage, it will delay the signal for one clock (waiting for write success), and then give the acknowledge signal to interrupter.

4.6 Conclusions

The implementation methodology and it’s details are discussed in this chapter. We used a top-down approach to implement the design of interrupt system. To begin with, we structured a top module for the interrupt system and then detail it by adding two sub-modules: the interrupt scheduler and the interrupt controller. Within the sub-modules, we also implement from top level and go to details until we finalized the whole design.

The interrupt scheduler is implemented as a 3-stage pipeline. The first stage receives interrupt signals and “rfi”, then updates the interrupt vector table at run-time. The
second stage schedules a task to the interrupt controller according to the interrupt vector table (interrupt priority of each vector is also included in the table). The last stage only updates a task queue based on the schedule result of the second stage.

Interrupt controller is implemented as a basic Finite State Machine (FSM), and then different ways to perform context switching are added to the FSM. The first method (Direct Context Switching) adds certain amount of multiplexer and a signal which halts the pipeline into the processor. Hardware Instruction Context Switching method removes these connections but inserts instructions into the fetch stage. The third method (Software Instruction Context Switching) further removes this generated instruction signal. For the page-able register files version, BRAMs are utilized on FPGA and the page number of register files is controlled by interrupt controller.
5

Experimental Results

To measure the performance of our interrupt system, a simple testbench is utilized for testing. This testing results verified the correctness of our interrupt system and the synthesis results could be compared for a user to chose a suitable implementation mechanism according to available resources and specific application.

Section 5.1 introduces the testbench used for testing. Then synthesis results are presented and compared with each other in section 5.2. Finally, section 5.3 concludes this chapter.

5.1 Testing Methods

Firstly, we tested the correctness and performance of our interrupt system with a simple testbench. This section introduces this testbench. The foreground program realize a simple multiplication. It writes some data into registers, then invokes a function to do the multiplication and finally writes the result into data memory. During this program, usage of several type of registers are included: use of link register, use of general-purpose register and use of caller-save registers. The first level interrupt ISR would write other data into registers that are employed in the foreground program for computation and then write this new value into data memory. The second level ISR would destroy the data of the first level ISR. Besides, ISRs will also modify the branch registers. This basic idea is applied to all 4 versions of interrupt system.

Whenever the context switching is performed in a wrong way, the sequence of foreground program might go wrong or the results of the multiplication might be wrong. If the nested interrupt is not performing as is expected, the data in the data memory will be incorrect.

The following issues were examined to justify the interrupt system. For testing details, please consult appendix B.

1. The sequence of foreground program should remain the same no matter if an interrupt is raised.

2. The data in GRs and BRs should remain the same before and after an ISR.

3. All ISRs should perform correctly no matter when an interrupt is enabled.

After the simple testbench, we test large applications (programs of DFT, ADPCM and CJPEG algorithms) and keep interrupting the processor every certain amount of cycles. In this way, we further verified the correctness of the system. Since a single interrupt has been tested, we do nothing in ISR and focus on correctness of execution results of foreground program with multiple interrupts. While testing, we interrupt the
foreground program every 16/8/4 cycles and compare the results. All results show that interrupts do not effect the correct execution of the foreground program.

5.2 Synthesis Results and Comparison

In order to match requirements of different applications, we might pick different mechanism of interrupt system to synthesis. Table 5.1 lists all designs of interrupt system. The version number will be used representing each version and compared in every domain.

<table>
<thead>
<tr>
<th>Version</th>
<th>Version description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>direct moving data, size of context fixed</td>
</tr>
<tr>
<td>2</td>
<td>hardware context switch, size of context not fixed</td>
</tr>
<tr>
<td>3</td>
<td>software context switch, size of context fixed</td>
</tr>
<tr>
<td>4</td>
<td>page-able register files, no need for context storing/restoring</td>
</tr>
</tbody>
</table>

5.2.1 Performance issues

Table 5.2 and figure 5.1 depicts the performance comparison between all these different versions. All the data is synthesized under Xilinx ISE release version 11.5 and target device is “xc6vlx240t-1-ff1156”\(^1\). Interrupt latency is the time elapsed between the detect of interrupt request and the start of ISR program execution. The figure 5.1 shows the interrupt latency of worst case, notice that for version 2, the best case would be 20 cycles. Besides, hardware consumptions are normalized to fit in this picture.

<table>
<thead>
<tr>
<th>Version</th>
<th>Latency (cycles)</th>
<th>Timing (MHz)</th>
<th>Slice Regs, Slice LUTs</th>
<th>BRAMs(36Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>77</td>
<td>128.932</td>
<td>3769(1%), 24510(16%)</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>20(16)-76(72)</td>
<td>142.039</td>
<td>3711(1%), 24764(16%)</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>78</td>
<td>155.785</td>
<td>3467(1%), 23585(15%)</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>136.074</td>
<td>1272(0%), 15789(10%)</td>
<td>16(4%)</td>
</tr>
</tbody>
</table>

In the first version of interrupter, all registers are stored to data memory. The 77 cycles consist of 4 cycles for stopping the pipeline, 1 cycle for scheduling an interrupt, 64 cycles for moving general purpose registers and 8 cycles for moving branch registers. The second version varies due to the maximum index of registers being used is checked at run-time. If “forwarded pipeline” is employed, the minimum interrupt latency would be 16\(^2\) (20 for non-forwarded) and maximum would be 72\(^3\) (76 for non-forwarded). The

---

\(^1\)For different version of synthesis tool, the exact number of results may different, so we only focus on the gap between each version of interrupt implementations.

\(^2\)1 cycle for scheduling an interrupt, 11(&r0.1-&r0.11,&r0.63) cycles for storing general-purpose registers, 4 cycles for storing branch registers.

\(^3\)1 cycle for scheduling an interrupt, 63(not &r0.0 since it is always zero) cycles for storing general-purpose registers, 8 cycles for storing branch registers.
third version is a software solution, so it is basically determined by software. This 78 latency is given here for a suggestion which “CALL” a function that saves all registers before the execution of ISR body. The last version only consume 2 clock cycles for contexts switching. One of the clock is for scheduling an interrupt and the other is for switching the page of register files.

Basically, interrupt latency of the second version varies on different applications. The less registers are using during the program, the smaller latency could be. Normally, the lowest index of register that being used to store context is 11 in ρ-VEX processor, as the first 10 registers have their special usage [4, 6] in a program. According to the run-time architecture of VEX [4], $r0.57$-$r0.63$ are used as callee-save registers. Once these register are used, improvement from this mechanism could be very small. So for big programs, we may need some other technique to reduce the interrupt latency.

![Figure 5.1: Performance comparison between different versions](image)

From figure 5.1, the worst case of interrupt latency are almost the same for the first 3 versions while the page-able register files method significantly reduces it to only 2 cycles. In the area of hardware consumption, it is worth to notice that the forth version minimize the use of slice and the use of slice LUTs among all the designs, due the utilization of other type of hardware resource (BRAMs) for register files.

In order to get deep insight of resource utilization of hardware, we would list details of hardware usage as illustrated in table 5.3 and figure 5.2. Since the hardware instruction context switching method uses the hardware to generate instructions at run-time, so more Look-up tables are employed. In the page-able register files method, BRAM is implemented on the FPGA, extra 4% (16 out of 36 KB BRAMs) of block of RAM are used (It is the figure when we implement 4 pages of register files).
CHAPTER 5. EXPERIMENTAL RESULTS

Table 5.3: Details of resource usage

<table>
<thead>
<tr>
<th>Version</th>
<th>Registers</th>
<th>Logic LUTs</th>
<th>Memory LUTs</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3769 (1%)</td>
<td>24510 (16%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>2</td>
<td>3711 (1%)</td>
<td>24764 (16%)</td>
<td>6 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>3</td>
<td>3467 (0%)</td>
<td>23585 (15%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>4</td>
<td>1272 (0%)</td>
<td>15789 (10%)</td>
<td>48 (0%)</td>
<td>16 (4%)</td>
</tr>
</tbody>
</table>

Figure 5.2: Hardware utilization for different versions

5.2.2 Comparison with RISC processors

In order to further evaluate our work, we compare our interrupt latency with other soft-core processor’s. The interrupt controller of Microblaze is very similar to our interrupt system and the relative information is easily accessible online. Therefore, Microblaze is picked as for comparison in our project. Figure 5.4 is taken from [8] which shows the interrupt latency for different scenarios in Microblaze. The cycle count includes completing current instruction, branch penalty and access to the first instruction in the ISR (NO context switching).

Table 5.4: Interrupt latency of Microblaze

<table>
<thead>
<tr>
<th>Scenario</th>
<th>ISR in LMB</th>
<th>LSR in OPB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normally</td>
<td>4 cycles</td>
<td>6 cycles</td>
</tr>
<tr>
<td>Worst case without hardware divider</td>
<td>6 cycles</td>
<td>8 cycles</td>
</tr>
<tr>
<td>Worst case with hardware divider</td>
<td>38 cycles</td>
<td>40 cycles</td>
</tr>
</tbody>
</table>

Microblaze has a 3-stage of pipeline and the data about interrupt latency does not include context switching. Thus we adjust our testing result to make it easy to compare as listed in table 5.5.

Table 5.5 shows that a VLIW processor has more pipeline stages but could perform even better than some RISC-based softcores in terms of interrupt latency by using our
5.3. CONCLUSIONS

Table 5.5: Comparison of interrupt latency between Microblaze and our design

<table>
<thead>
<tr>
<th>Processor</th>
<th>pipeline stage</th>
<th>interrupt latency (worst case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microblaze</td>
<td>3 stages</td>
<td>8 cycles (no divider)</td>
</tr>
<tr>
<td>ρ-VEX Version 1</td>
<td>5 stages</td>
<td>5 cycles (no context switching)</td>
</tr>
<tr>
<td>ρ-VEX Version 2</td>
<td>5 stages</td>
<td>5 cycles (no context switching)</td>
</tr>
<tr>
<td>ρ-VEX Version 3</td>
<td>5 stages</td>
<td>5 cycles (no context switching)</td>
</tr>
<tr>
<td>ρ-VEX Version 4</td>
<td>5 stages</td>
<td>2 cycles (including context switching)</td>
</tr>
</tbody>
</table>

design of interrupt system.

5.3 Conclusions

This chapter presents the testing results of our work. We verified the functionality of our interrupt system in the following aspects: (1) the sequence of foreground program remains the same no matter an interrupt is raised or not; (2) the data in GRs and BRs (processor’s context) remains the same before and after an ISR; (3) all ISRs should perform correctly no matter when to enable the interrupt.

We synthesized our design on the xilinx “xc6vlx240t-1-ff1156” FPGA under Xilinx ISE release version 11.5. The results show that our interrupt system itself does not worsen the critical path (maximum clock frequency that processor can run). The page-able register files method put the register files into BRAM hardware resources (this approach makes multi-page register files applicable) and thus we get only 2 cycles of interrupt latency. For other methods, the worst case of interrupt latency are almost the same. However we could be able reduce the latency at run-time at a price of some hardware consumptions. Besides, comparison with other softcores show that the performance of our system is good.
In this chapter, we present the design and implementation of possible system extensions. These extensions include exceptions support, interrupt enhancement that extend the functionality of interrupts and multi-core system that make use of the interrupt system.

Exceptions are introduced and discussed in section 6.1. Then interrupt enhancement is presented in section 6.2. After that, Section 6.3 introduced a multi-core system that switches task among cores. Section 6.4 concludes this chapter.

6.1 Exceptions

Both interrupts and exceptions will alter the program flow. The difference between the two is that interrupts are used to handle external events (serial ports, buttons etc.) while exceptions are used to handle internal instruction faults (division by zero, illegal opcode etc.). Exceptions can be divided into three categories in general: fault, trap and abort. Faults should be detected and fixed before the faulting instruction. Traps are serviced after the instruction causing the trap. Aborts are used only when severe system error happens and operation is no longer possible.

Due to the implementation of micro-architecture of the \( \rho \)-VEX processor, several types of exceptions can be defined. In this section, possible exceptions and method to handle these exceptions are discussed in details.

6.1.1 Arithmetic Overflow

When the result of addition or multiplication is too large for the hardware or register to represent, arithmetic overflow comes into the context. Arithmetic underflow (or “floating point underflow”, or just “underflow”) on the other hand only happens when the true result of a floating point operation is smaller in magnitude (that is, closer to zero) than the smallest value representable as a normal floating point number in the target datatype. Since there is no floating-point support in the \( \rho \)-VEX processor, only arithmetic overflow should be supported.

Arithmetic overflow can be detected only after the arithmetic operation has been done in ALU unit (or MUL unit). So this exception is detected in the execute stage. For the addition operation, the maximum result of two 32-bit operands could be 33-bit data, so we could simply pick the leftmost bit as the overflow flag and use it as exception signal to raise an internal interrupt. For multiplications, the overflow could be more than one bit. In the defined opcode list of the \( \rho \)-VEX processor, following operations might lead to arithmetic overflow: (all of them would result in 48-bit value but only 32-bit value could be represented in the \( \rho \)-VEX processor.)

- MUL_MPYL : Multiply signed low 16 (s2) x 32 (s1) bits
CHAPTER 6. SYSTEM EXTENSIONS

- **MUL_MPYLU**: Multiply unsigned low 16 (s2) x 32 (s1) bits
- **MUL_MPYH**: Multiply signed high 16 (s2) x 32 (s1) bits
- **MUL_MPYHU**: Multiply unsigned high 16 (s2) x 32 (s1) bits
- **MUL_MPYHS**: Multiply signed high 16 (s2) x 32 (s1) bits, shift left 16

For the unsigned multiplication, arithmetic overflow occurs when the leftmost 16-bit of result is not x“0000”. While in signed multiplications, overflow occurs when the leftmost 16-bit is not x“0000” or x“FFFF”. We set the carry out bit to 1 if arithmetic overflow of multiplication occurs and vice versa. Figure 6.1 illustrates that how this signal is sent to interrupt system.

![Figure 6.1: Signal flow for arithmetic overflow exception](image)

6.1.2 Invalid Opcode

When an opcode that is not supported by the processor is fetched into the pipeline, it will cause the exception of invalid opcode (or illegal opcode). This could be detected in the decoder stage since the opcode is checked and then operand is selected and passed to execute stage based on the opcode in the decoder stage.

The 7-bit opcode field can represent 128 possible instructions in total and most of them are used in the instruction set. So rather than check the available opcode, we define unused opcode as special instructions and generate exception signals when these opcode are matched. In this case, better organization of instruction set will get us easier implementation of the invalid opcode exception. Figure 6.2 depicts that how the exception of invalid opcode is sent to interrupt system.

6.1.3 Unavailable Hardware Unit

Four ALU units, two MUL units, one MEM unit and one CTRL unit is distributed over 4 syllables in the ρ-VEX processor. So not all the operations can be executed in every syllable. If we assign instructions in a wrong syllable, the CPU can do nothing with that, this is where the exception of unavailable hardware unit comes into the picture. Due to the reason that opcode is checked in decoder stage, this exception can also be detected in the decoder stage, as the exception of invalid opcode.
6.1. EXCEPTIONS

Take instruction layout depicted in figure 6.3 for example: there are no MEM unit and MUL unit in syllable 0, no MEM unit and CTRL unit in syllable 1 and syllable 2, no MUL unit and CTRL unit in syllable 3. If any of operations are sent to the wrong syllable, a signal indicates the exception of unavailable hardware unit should be sent to interrupt system.

![Signal flow for invalid opcode exception](image)

Figure 6.2: Signal flow for invalid opcode exception

6.1.4 Division by Zero

In mathematics, divisor should never be zero as it makes no sense. In computer computing, an attempt to divide by zero should also generate an exception to notify users that an error may occur during the computing. In the $\rho$-VEX processor, no hardware
divider is available so division is implemented as emulation in ALU unit. Thus it can be detected in execute stage, as the exception of arithmetic overflow.

When the operation is division, we check the divisor (second operand of instruction). If it is zero, we set the exception and clear it if not. Figure 6.5 shows the exception signal flow for division by zero.

![Figure 6.5: Signal flow for division by zero exception](image)

### 6.1.5 Mode of Exceptions

In general, exceptions can be divided into two modes: precise exception and imprecise exception. Precise exception requires all exceptions except aborts report the exception on a precise instruction boundary. It has features as follows:

- Exceptions are reported on the faulting instruction.
- All previous instructions are completed before the interruption point.
- All subsequent operations are nullified.
- After handling the exception, the execution resumes at the faulting instruction (fault) or at the next instruction (trap).

Precise exception is necessary when the error may effect the following instructions or even the whole system (etc. memory access error, page fault). The problem should have to be fixed before we can execute more program. In order to decide which mode we have to support, the effects of each exception should be examined first, as follows:

- Arithmetic overflow indicates that the result of addition or multiplication is not representable under the architecture of the processor. This could either because the calculation is not possible/correct on the processor or the original data is not correct. Since we have no idea wether it is the error of current operation or operation in previous (when the operand is the result of previous operations), precise exception can not make sure to fix the error. in the Interrupt Service Routine (ISR) of this exception, we may reset the original data and start over the computation.
• Division by zero is another type of arithmetic exceptions. If there is a hardware divider being used, divisor to be zero may not be acceptable by the hardware divider. In this case, this exception has to be serviced and fixed before any further operations. In the $\rho$-VEX processor, division is implemented as emulation (multiple additions and/or subtractions which is generated by the assembler), so divisor to be zero will not result in severe problems. Therefore, imprecise exception is acceptable. Besides, this error may come from the previously operations (we assume instruction memory access is reliable and no one would write a program that divides some number by zero), precise exception does not help to fix the problem.

• Invalid opcode and unavailable hardware unit can be grouped into one as opcode error. This error will produce nothing to the system, which act as a “NOP” instruction. This type of error might happen when the assembler generates wrong instructions on a different version of the processor. Therefore, imprecise exception is also acceptable for these two exceptions.

In sum, precise exception is not necessary to support at this phase. Hence, imprecise exception is implemented as the mechanism for handling exception. It is the same as interrupts, only the resource of exception/interrupt is different (exception is internal and interrupt is external).

6.2 Interrupt Enhancements

The interrupt system is implemented under the available micro-architecture of the $\rho$-VEX processor and instruction set\(^1\). In order to support more functionalities for the interrupt system, the original micro-architecture and instruction set of the $\rho$-VEX processor are extended. This section discuss these interrupt enhancements.

6.2.1 Software Interrupt Support

In practical, one might want to raise interrupt from software in order to do a critical computing or task. This requires the possibility that instructions can also “generate” interrupt signals to the interrupt system. Figure 6.6 shows the signal flow of software interrupt. We simply assign an interrupt pin to this software interrupt, the rest is the same for interrupt system.

Since we don’t have instructions that could “generate” interrupts, we should also adjust the instruction set to support this functionality. Figure 6.7 depicts the layout of such instruction that designed for software interrupts. Only the opcode field is used in this instruction so we may treat it like some other special instructions like “STOP” or “NOP”. Therefore, syllable 0 is chosen for decoding this special instruction. It is very easy to extend to other syllables if one is working.

Whenever “INT_SOFT” is decoded, we set an interrupt signal to interrupt system, otherwise we clear that signal. Since instructions are given to decode stage per cycle, so this “software interrupt” signal will last at least one clock cycle, which will make sure that we can always interrupt from software.

\(^1\)The original instruction set does not support the manipulation of the interrupt system
6.2.2 Atomic Operation Support

Atomic operation refers to multiple operations that should be done at one time and cannot be interrupted by others in the middle of the execution. This is important when we are doing critical operations. During this critical operations, we do not want others to modify the intermediate results of our computing or we have to complete the computing before a deadline. In short, atomic operation requires that we are able to enable or disable an interrupt or all of the interrupts.

In the interrupt scheduler, we have reserved a bit that can mask interrupts for every vector. Only when the corresponding enable bit is ‘on’, we update the interrupt flag by interrupt input, otherwise the interrupt request will be ignored. Figure 6.8 is the scheme of “enable/disable” interrupts. In this picture, enable/disable signals are sent to interrupt system.

Figure 6.9 and figure 6.10 show the layout of instruction “INT_WEN” and “INT_CEN” respectively. The filed of “enable bit” and “disable bit” indicate the interrupt(s) you want to enable or disable. Logic high is valid for “enable bit” instruction and logic low is valid for “disable bit” instruction. Examples are given in table 6.1.
Syllable 0 is assigned to support these operations, as we did for software interrupt.

![Figure 6.9: Syllable layout for INT_WEN syllable](image)

![Figure 6.10: Syllable layout for INT_CEN syllable](image)

Table 6.1: Examples that enable/disable certain interrupt(s)

<table>
<thead>
<tr>
<th>explanation</th>
<th>opcode</th>
<th>enable(bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable interrupt</td>
<td>INT_WEN</td>
<td>0x0e</td>
</tr>
<tr>
<td>disable interrupt</td>
<td>INT_CEN</td>
<td>0x06</td>
</tr>
</tbody>
</table>

Table 6.1 lists examples that enable and disable interrupts:

- For “INT_WEN”, ‘1’ is valid so 0x0e = 0b 1110 means enable interrupt vector(3 downto 1). So when implementing, we simply ‘OR’ this field with interrupt enable bit.

- For “INT_CEN”, ‘0’ is valid so 0x06 = 0b 0110 means disable interrupt vector 3 (interrupt vector 0 is assigned to foreground program so it makes no different). So when implementing, we simply ‘AND’ this field with interrupt enable bit.

When enable/disable an interrupt:

1. The 1st clock: instruction is fetched in fetch stage;

2. The 2nd clock: instruction is decoded in decode stage, and enable/disable signals are sent to interrupt system;

3. The 3rd clock: enable/disable signals are updated into interrupt vector table;

4. The 4th clock: corresponding interrupt is enabled/disabled;

When an instruction is decoded, we at least wait for 2 cycles until the corresponding bit is really set/clear. In other words, we disable interrupts 2 instructions before we enter into critical operations. On the other hand, we can also enable interrupts 2 instructions before we leave critical operations.
6.3 Multi-core System

In this system, a core scheduler would pick one core to execute a piece of code and then let the second core to execute the rest of the program. These cores have different configuration so we can increase performance of overall system in terms of speed and hardware consumptions by scheduling tasks to different configured cores at run-time. Followings are steps that the scheduler might do while switching cores and figure 6.11 illustrates the steps of switching tasks between cores.

1. Stop the pipeline of first core.
2. When the first core is stopped, get PC address and move context from register files into data memory (or a shared memory that all other cores can access).
3. Load the context into register files of the second core.
4. Load the pc address into the second core.
5. Start execution of second core.

![Figure 6.11: Logic for switching tasks between cores](image)

6.3.1 Halt pipeline and store context

Since the interrupt system could halt the pipeline and get PC value of foreground program (actually it is next PC value as the address that should be executed after return from interrupt), we could make use of this functionality in the multi-core system. We simply assign one interrupt pin to the core scheduler so the scheduler could stop the
pipeline by raising an interrupt on that pin to the first core. In the ISR program of this special interrupt vector, we could let the software to store context and then stop the instruction stream by inserting “STOP” instruction at the end of ISR instead of “RFI”. In the meanwhile, because the return address (next PC address value) is also computed by interrupt system and sent to fetch stage, we could also send this signal to the core scheduler.

### 6.3.2 Load PC address and context into second core

When we are executing program on the first core, we keep ‘reset’ signal valid on the second core so the second core would not run any programs. After the first core is stopped and PC address is sent to core scheduler, the scheduler could then send the address to the second core. The second core would fake an ISR program which the return from interrupt (‘RFI’) address is the address that received from scheduler and the ISR itself is the piece of code that restoring the context. One clock cycle after sending address to the second core, the scheduler do release the ‘reset’ signal and the second core will start executing at the ISR program. Therefore, the context is switched to the second core and the rest of program can subsequently be executed by the second core.

### 6.3.3 Kernel FSM in interrupt controller

According to the new functionality, interrupt controller is adjusted and a concept of “kernel” mode is then introduced. This “kernel” mode is designed for the multi-core system. During switching cores, no interrupt can be received by cores and thus new Finite State Machine is added to the old one. Figure 6.12 shows this new FSM particularly designed for “kernel” mode and it could co-operate with the previous interrupt FSM.

![Finite state machine for kernel mode](image)

### 6.3.4 Testing of Multi-core System

This system is tested only on the software context switching method of interrupt system because:
1. This new “kernel” mode can be simply applied on every method of interrupt system so we don’t need to test on each method.

2. The register files can not be shared by other cores so page-able register files method is not applicable\(^2\).

3. The hardware context switching method should pass extra information about the size of context that stored to the second core which makes the system more complex.

4. The directly moving context method takes more hardware consumptions and does not give the fastest interrupt latency so one might not be interested in it.

The testbench that is used for interrupt system (DFT algorithm) is employed into the multi-core system. We randomly picked certain clock cycle to start the switching core task for testing the correctness of our implementation. The results show that it works\(^3\).

6.4 Conclusions

This chapter introduces the work that extend the functionality of our interrupt system for the \(\rho\)-VEX processor. These extensions are: 4 types of exceptions (arithmetic overflow, invalid opcode, unavailable hardware unit and division by zero), 2 kinds of interrupt enhancement (software interrupts support and enable/disable interrupt support) and a multi-core system that could switch tasks between cores which make use of the interrupt system.

\(^2\)Page-able register files method does not store and restore context but switches to another page of register file

\(^3\)Actually, the difference between multi-core system and the single core interrupt system is that: the core scheduler of multi-core system pass signals between interrupt controller of different cores while the single interrupt system use the signal in the interrupt controller within a single core.
Interrupt System Evaluation

Before warping up our project, we should evaluate the success of our design in such a way that the reader get an overview of what we have achieved in our project and where we still need to put more effort on. The evaluation is done from three objectives: performance (how fast can we respond to an interrupt request), extensibility (how easy can we extend our design to support more functionalities) and feasibility (how easy can we implement our design based on the available hardware platform and toolchain).

Section 7.1 discusses and evaluates the performance we’ve achieved with our design. Then the extensibility of our design is presented in section 7.2. After that, feasibility which means how easy we can realize the interruptibility support on the ρ-VEX is discussed in section 7.3. Section 7.4 concludes this chapter.

7.1 Performance

The time spent to context switching dominates the interrupt latency and thus is the main contribution to the overall performance of the processor. According to context switching, two problems should be solved by our interrupt system: (1) When to switch? (immediately after the detecting of interrupt request or check priority and wait for completion of instructions in the pipeline before doing so); (2) What to switch? (size of contexts and what contexts).

For the method of page-able register files, BRAMs are utilized for multiple pages of register files so we can switch page of register files at run-time. Hence, we can largely reduce the interrupt latency. However, it is realizable only if we have such hardware resources. In this section, we are considering the situation that we can not implement page-able register files based on the available hardware resources.

7.1.1 Minimize overhead of context switching

Normally, we should wait for the last instruction to finish, until we can switch contexts. That is because we could never know if the last instruction would write data to the context we are going to store at run time. In order to make sure that we are transferring the correct data, instruction steam should be completed before any further actions.

Upon the ρ-VEX processor architecture, it takes at least 5 cycles to empty the pipeline before we could start to switch the contexts. In our design, rather than stopping the pipeline, we insert instructions into pipeline to do context switching so the pipeline never has to be stopped during this process. This mechanism takes advantage of the pipeline and existent data bus and address bus of registers and data memory and would work for any applications. The testing result for a “forwarded pipeline” results a 4-cycle reduction of interrupt latency.
CHAPTER 7. INTERRUPT SYSTEM EVALUATION

7.1.2 Minimize size of context for storing

Such contexts should be stored only if it has to be retrieved for computation in the foreground program and the data will be corrupted during the execution of the Interrupt Service Routine (ISR) program. In other words, if we are able to make sure such register will not be read for computation later or it will not be written with a new value in the ISR program, we do not need to save it somewhere else during context switching.

For the $\rho$-VEX processor architecture and available compiler, there is no way the hardware can be aware of which register is going to be read later or used for computing in the foreground program and/or ISR program. So in principle, all registers should be stored and restored to ensure the correctness of our program.

Although we are not able to know the upcoming usage of registers at run time, the registers that have been accessed can be seen on address bus of register files. That gives us a picture of which registers participate into computing and therefore we are able to sweep away registers that are never accessed. Our design will check the address bus of register files and remember the maximum index of them as the size of context for storing. In our test case for the second method, the interrupt latency has been reduced to 16 cycles which is a significant improvement, but it is basically application dependent. If it comes to large program, interrupt latency will also be large. Moreover, if register group $r0.57 - r0.62$ which are used as callee-save registers [4] have been accessed, there is no improvement by applying this mechanism. To sum up, improvement of this mechanism depends on different applications and run time. The improvement could be zero in the worst case.

7.1.3 Can we do better

In our design, we use the maximum index of registers being accessed as the size of context that should be stored while context switching. If we see register zero to register of the maximum index as a set of context, we assumes every register in this set has been accessed. This is not always true since it is basically compiler defined (etc. the special usage of register $r0.57 - r0.62$). So if we check each register individually, interrupt latency could be further improved.

Nevertheless, in order to realize it, we should introduce extra considerable hardware for “remembering” this information and logic for dynamically generating instructions and consequently the context switching could become very complex. In addition, we still get long interrupt latency when it comes to large programs. In short, we gain little by large resource consumption which makes this mechanism non-sense.

7.2 Extensibility

Our interrupt subsystem is a generic design so it can be extended to support advanced functionalities with small modification of the current project. In this section, several possible extensions are discussed.
7.3. FEASIBILITY

7.2.1 Number of Interrupt vectors

By our design, the interrupt vector can be easily extended to any number we want. The only thing we should keep in mind is the depth of interrupt have to be reasonable. Since the nested interrupt will be going to use more space for storing contexts, multi-nested interrupt might easily run into exhaustion of data memory.

7.2.2 Task switching by time-slice

In operating systems, tasks can be switched to execute after a pre-defined period of time. If we assign a set of tasks to interrupt vector zero and interrupt priority zero (as we do for the current foreground program), we gain a set of foreground programs. No matter what task is running, it could be seen as foreground program to the interrupt system. After small modification on the scheduling logic (within component of interrupt scheduler) and implementation of a timer, tasks with same priority could switch to each other by time-slice.

This functionality is useful when we want to run operating system on our platform. But apart from the hardware support, the compiler should also be extended to support the “creation” of such task and pass the information to hardware. At this point, the compiler can only do very little about it so we would prefer to support an operating system at a later phase.

7.2.3 Atomic operations support

In some applications (or under some situation), we would like to make some operations atomically. That is to say, we should be able to disable interrupt for a short period of time to execute critical computations. Interrupt enable bit could be easily added into interrupt vector table in our design to extend this functionality. But this action needs the support of the compiler as we should write data into special hardware unit and the compiler has be aware of this.

7.3 Feasibility

Since the VEX compiler is developed and freely provided from HP, the requirements on the back-end assembler would be an important issue as to evaluate the feasibility of our work. Basically, the assembler should be able to generate ISR code and locate it into specific address of instruction memory. Apart from this requirement, Table 7.1 lists the other requirements on the relative assembler among different versions. For version explanations, please consult Table 5.1.

For the first version, there is no other requirements for the assembler so it would be very easy to realize. The second version would dynamically adjust the size of contexts for storing and restoring. Since the size could only increase during run-time based on our design, the assembler should assign registers as less as possible thus to reduce the

---

1For version 3, the requirement will extend the size of program resides in instruction memory. For version 2, it only require the compiler or assembler to re-mapping register name space thus hardware resource will be saved. In version 4, immediate read(write) means within 4 cycles.
### Table 7.1: Requirements to the compiler (assembler)

<table>
<thead>
<tr>
<th>Version</th>
<th>Requirements to the compiler (assembler)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no specific requirements</td>
</tr>
<tr>
<td>2</td>
<td>using registers(index) as less(small) as possible</td>
</tr>
<tr>
<td>3</td>
<td>specific code should be compiled for context switching</td>
</tr>
<tr>
<td>4</td>
<td>no immediate read(write) GRs before(after) execution of ISR</td>
</tr>
</tbody>
</table>

interrupt latency. Otherwise, we can not gain improvement from this method. The third version requires the assembler to generate instructions of contexts switching for a user. There are two scenarios in this method: embed instructions before and after the body of ISR; “CALL” specific function that deal with contexts switching before and after the body of ISR. The first scenario would save 2 cycles of interrupt latency but the size of softcode would explode if there is too many ISRs because we should add almost the same piece of code into every ISR program. The second one would minimize the instruction memory at a cost of 2 cycles of interrupt latency (“CALL” and “RET”). For the last version, we should make sure the first 4 instructions in the ISR would not read data from registers and the last 4 instructions in the ISR would not write data into registers.

To summarize, the requirements to the assembler are easy to implement for each version of contexts switching mechanism.

### 7.4 Conclusions

This chapter presents an evaluation of our work so the success of it can be established. The evaluation is performed from three points of view: performance, extensibility and feasibility.

The evaluation about performance shows that based on the current toolchain and processor architecture, we are not able to reduce the interrupt latency anymore. The improvement we can gain differs according to applications and run-time register usage. For applications use large amount of registers (etc. scientific computing, signal processing) or programs use the callee-save registers (once it is used, these registers are always “live” afterwards), we can only get very little improvement. This evaluation does not consider the use of BRAM as what we did in the page-able register files method.

The evaluation of extensibility of our work lists some possible extensions we can do at some later phase. This could extend our capability of interrupts (more interrupts support), extend the processor to be able to support operating system, or extend to support atomic operations on the processor. In short, our system is designed and implemented in a generic way that one could extend it to support many functionalities easily.

The evaluation of feasibility focus on how easy we can realize the design upon the current toolchain and available hardware resources. The discussion about this issue leads to a conclusion that all solutions are applicable and easy to implement from the assembler point of view.
Conclusions

In this chapter, we present the conclusions, contributions and some future work. The conclusions are organized by chapters and presented in section 8.1. Then section 8.2 lists several contributions from our work. Finally, some future work to further improve the interrupt system is presented in section 8.3.

8.1 Summary

In this thesis, we present an interrupt system for the ρ-VEX processor [19]: an open source reconfigurable and extensible VLIW softcore processor based on the VEX ISA [6]. This interrupt system is designed and implemented with 4 solutions, with respect to the interrupt latency, hardware resource usage and requirements for the relative toolchain. With this design, we are able to match the requirements of different applications. Each interrupt system implementation itself can also be configured for different applications. Such configuration parameters include the number of interrupt vectors, interrupt priority for each vector and Interrupt Service Routines (ISRs) address in the instruction memory. The synthesis results show that hardware consumption for all versions of our interrupt system are below a reasonable number. The interrupt latency comparison between our work and some other softcores like Microblaze shows that our interrupt system can also response to an interrupt very quickly so it could be used under a real-time situation. Besides, our interrupt system only requires the compiler (assembler) to be slightly adjusted to support the interrupt functionality which makes our work more feasible.

Chapter 2 presents background knowledge. The ρ-VEX processor and a general interrupt system are briefly discussed. The ρ-VEX processor can be utilized as a CCU within MOLEN [17] polymorphic processor organization which bridge the gap between general-purpose processors and specific applications. An interrupt system concerns about: (1) interrupt latency, which is an important figure to assess the performance of such system. The task of context switching is the major contribution to interrupt latency. The efficiency of storing and restoring context (how many context to store per clock cycle or time unit) and the size of context should be stored determine the performance of context switching; (2) interrupt priority: not all the interrupts can interrupt the current task, only if the interrupt is more urgent. In other words, only interrupt with higher priority should be treated immediately and interrupt the running task with a lower priority. In the practical world, there are already many interrupt designs. The classic 8051 architecture treat register files as same virtual address but different physical address [1]. It is useful since 8051 has only 8 registers and 4 banks of them are also easy to implement. The Microblaze RISC softcore has a PIC (Programmable Interrupt Controller) which schedules interrupts. For VLIW softcores, many approaches are also studied and implemented by others [7, 12]. However, these designs need the help of compiler which are
not applicable in our case because the \( \rho \)-VEX compiler is a closed-source VEX compiler.

In chapter 3, we presented the design of our interrupt system. The top module (interrupter) of this system consists of two sub-modules: interrupt scheduler and interrupt controller. Interrupt scheduler is designed pipelined thus no interrupts will be lost and they will be scheduled to a task queue and interrupt controller according to their priorities. The interrupt controller is designed as a Finite State Machine (FSM) since the current state is always depending on the previous state and we are not able to finish context switching within a single clock cycle. Four different mechanisms to do context switching are presented in this chapter so one could understand how we reduce the interrupt latency and/or hardware consumption. The first method is an intuitive one that directly storing context while interrupt comes. The second method reduces the interrupt latency by reducing the size of registers to be stored and restored. Upon a “forwarded pipeline”, the interrupt latency could be further decreased by this method. After we move the complexity of context switching from hardware to software, we get the third mechanism. This mechanism is very simple to implement from the hardware point of view but would require the toolchain to produce program that deal with context switching. The last method results in the fast context switching compared to all other methods at a price of certain special hardware utilization: block memory on an FPGA.

In chapter 4, the implementation methodology and its details are discussed in this chapter. We used a top-down approach to implement the design of interrupt system. To begin with, we structured a top module for the interrupt system and then detail it by adding two sub-modules: the interrupt scheduler and the interrupt controller. Within the sub-modules, we also implement from top level and go to details until we finalized the whole design. The interrupt scheduler is implemented as a 3-stage pipeline. The first stage receives interrupt signals and “rfi”, then updates the interrupt vector table at run-time. The second stage schedules a task to the interrupt controller according to the interrupt vector table (interrupt priority of each vector is also included in the table). The last stage only updates a task queue based on the schedule result of the second stage. Interrupt controller is implemented as a basic Finite State Machine (FSM), and then different ways to perform context switching are added to the FSM. The first method (Direct Context Switching) adds certain amount of multiplexer and a signal which halts the pipeline into the processor. Hardware Instruction Context Switching method removes these connections but inserts instructions into the fetch stage. The third method (Software Instruction Context Switching) further removes this generated instruction signal. For the page-able register files version, BRAMs are utilized on FPGA and the page number of register files is controlled by interrupt controller.

Chapter 5 presents the experimental results of our work. The functionality of our interrupt system is verified. Then our design is synthesized on the target device “xc6vlx240t-1-ff1156” under Xilinx ISE release version 11.5. The results show that our interrupt system itself does not worsen the critical path (maximum clock frequency that processor can run). The page-able register files method implements the register files utilizing BRAM hardware resources and we get only 2 cycles of interrupt latency. For other methods, the worst case for interrupt latency are almost the same. However we could be able reduce the latency at run-time at a price of some hardware consumptions. Besides, comparison with other softcores show that the performance of our system is
good. In chapter 7, our work is further evaluated from three points of view: performance, extensibility and feasibility. The evaluation about performance shows that based on the current toolchain and processor architecture, we are not able to reduce the interrupt latency anymore. The improvement we can gain differs according to applications and run-time register usage. For applications that use a large amount of registers (etc. scientific computing, signal processing) or programs that use the callee-save registers (once it is used, these registers are always “live” afterwards), we can only get very little improvement. This evaluation does not consider the use of BRAMs as what we did in the page-able register files method. The evaluation of extensibility of our work lists some possible extensions we can do at some later phase. This could extend our capability of interrupts (more interrupts support), extend the processor to be able to support operating systems, or extend to support atomic operations on the processor. In short, our system is designed and implemented in a generic way that one could extend it to support many functionalities easily. The evaluation of feasibility focuses on how easily we can realize the design upon the current toolchain and available hardware resources. The discussion about this issue leads to a conclusion that all solutions are applicable and easy to implement from the assembler point of view.

Chapter 6 presents some work to extend the functionality of the interrupt system for the ρ-VEX processor. These extensions include: 4 types of exceptions (arithmetic overflow, invalid opcode, unavailable hardware unit and division by zero), 2 types of interrupt enhancements (software interrupts support and enable/disable interrupt support) and a multi-core system that can switch tasks between cores utilizing the interrupt system.

8.2 Main Contributions

The following contributions can be assigned to our work:

- Interrupt System
  - We prototyped an interrupt system that could schedule interrupts according to their priorities, storing and restoring context, invoke ISRs and resume foreground program. The interrupt system itself could also be parameterized to match requirements for different applications.
- Investigation different context switching mechanisms
  - In our work, 4 methods for context switching are derived and implemented. This gives us flexibility to be able to use any one of them according to the requirements of different applications and available hardware resources on the FPGA.
- Exceptions support and system extensions
  - Some possible exceptions are implemented and imprecise exception is used.
  - Software interrupts and enable/disable interrupt operations supported. This extends the instruction set and directly access the interrupt system.
Some possible extensions are also prototyped in our work. Multi-core solutions could schedule tasks to cores at run-time thus to improve the performance of the overall system.

8.3 Future Work

This section discusses possible ways to further reduce the interrupt latency. It does not relate to the extensibility since extensibility is to support more functionalities on the processor which is already discussed in chapter 7. Besides, suggestions are presented for the case when implementation of the page-able register files could not be realized due to the lack of hardware resources.

According to the current compiler and architecture of ρ-VEX, the hardware can only be aware of very little information of program that is running, so precisely context switching is hard. If we are able to write our own compiler and modify the architecture of ρ-VEX appropriately, we can play more tricks on context switching and therefore make even more faster switching.

8.3.1 4-MEM-unit version of ρ-VEX

In the current version of ρ-VEX, only one port is connected to data memory, so we have to move data between data memory and registers one by one. When 4 MEM unit are assigned to 4 lanes, we could transfer 4 register values at one time thus further reduce the cycles consumed during context switching.

8.3.2 Live range detection

As we have discussed in chapter 3, only live registers are necessary to be saved as context, so if compiler can pass these information to hardware, the interrupt latency could be further reduced. Moreover, the number of living registers is usually small (the number of registers have been accessed is always larger than that still alive) during run time, so we could gain more improvement for more applications. For example, for some scientific applications, a large amount of registers might be employed when scientific calculation is performed. If we do not use live range detecting, we would run at maximum interrupt latency level.

8.3.3 Subset of register files

This could be an ultimate solution for reducing interrupt latency. The basic idea of this method is rather than saving and retrieving context, we turn to other subset of context (register files) instead. The obviously disadvantage of this method is that the resource consumption could be very large. However, if we put some energy on compiler and rearrange the current register files, we might be able to find a balance point. For example, if we assign different subset of register files to some critical tasks, the interrupt latency of these tasks would be very small\(^1\). The less important tasks could apply live

\(^1\)It could be as small as potentially 1 clock cycle for context switching
range detecting mechanism and save resource consumption.


Interrupt Configurations

A.1 Parameters

Following are parameters that can be changed in the ‘int_pkg.vhd’ file:

- **constant** INT_VECTOR : integer := 4;
  - It claims the number of interrupt vectors you could have in your system;
  - 4 → 4 interrupt resource inputs;

- **constant** INT_PRI_LEV : integer := 3;
  - It claims the depth of interrupt priority level you could use in your system;
  - 3 → $2^3 = 8$ interrupt priority level;

- **constant** DM_OFFSET : std_logic_vector(6 downto 0) := “1000000”;
  - It claims the start point of offset in data memory you want to store the context;
  - “1000000” → 64: you store context into data memory starting from dm(63);

- **constant** FWD_DELAY : integer := 4;
  - It is used for Hardware Instruction Context Switching method;
  - It claims the delay for starting context switching (for forwarded pipeline, it should be 0 otherwise 4);

- **constant** addr_gr_min : std_logic_vector(5 downto 0) := “000100”;
  - It is used for Hardware Instruction Context Switching method;
  - It claims the minimum general-purpose registers to be stored/restored during context switching;
  - “000100” → 4: at least 4 general-purpose registers should be stored;

- **constant** addr_br_min : std_logic_vector(2 downto 0) := “100”;
  - It is used for Hardware Instruction Context Switching method;
  - It claims the minimum branch registers to be stored/restored during context switching;
  - “100” → 4: at least 4 branch registers should be stored;
Following are parameters added into ‘r_vex_pkg.vhd’ file that could configure the number of pages of register files:

- **constant INTERRUPT_LEVELS : integer := 4;**
  - It is used for Page-able Register Files method;
  - It claims the levels of interrupts, including foreground program. It is equal to the number of pages of register files;
  - $4 \rightarrow 4$ pages of register files;

- **constant INTERRUPT_LEVEL_WIDTH : integer := 2;**
  - It is used for Page-able Register Files method;
  - It claims the width of signal that pick any page of register files;
  - $2 \rightarrow 2$ bits which corresponds to $2^2 = 4$ levels of interrupts;
  - $8/4/2$ levels $\rightarrow 3/2/1$ width respectively;

### A.2 Function

Following are function definition in ‘int_pkg.vhd’ that are adjustable:

```vhdl
function gen_instruction (inst_syl3 : in std_logic_vector(31 downto 0);
  inst_syl2 : in std_logic_vector(31 downto 0);
  inst_syl1 : in std_logic_vector(31 downto 0);
  inst_syl0 : in std_logic_vector(31 downto 0))
return std_logic_vector is
begin
return inst_syl3 & inst_syl2 & inst_syl1 & inst_syl0;
end function gen_instruction;
```

- It is used for Hardware Instruction Context Switching method;

- For 2-issue $\rho$-VEX processor, you may only return the combination of syllable 3 (MEM-unit) and syllable 2 (ALU-unit);

### A.3 Interrupt Vector Table

ISR address and interrupt priorities should be pre-defined (fixed) at this point. So one should be able to configure it before synthesizing it. These definitions are claimed in ‘int_sche.vhd’ file:

- **signal inst_addr : bank_inst_addr :=**
  - (“00000000000”, “00001000000”, “00100000000”, “00001001000”);
  - It is used for all methods;
  - It claims the ISR addresses for all interrupt vectors;
- It is organized starting from vector 0 to vector INT_VECTOR-1. So ISR address of interrupt vector 0 is “00000000000” and ISR address of interrupt vector 3 is “00001001000” in this case (INT_VECTOR = 4);
- Notice that the ISR address of interrupt vector 0 is always 0 as it stands for the foreground program, one should not adjust it.

- It is organized starting from vector 0 to vector INT_VECTOR-1. So interrupt priority of interrupt vector 0 is “000” and interrupt priority of interrupt vector 3 is “111” in this case (INT_VECTOR = 4). The bigger the number, the higher the priority;
- Notice that the priority of interrupt vector 0 is always 0 as it stands for the foreground program, one should not adjust it.

- signal int_pri : bank_priority :=
  (“000”, “101”, “010”, “111”);
- It is used for all methods;
- It claims the priorities for all interrupt vectors;
Testbench Verification

Functionality testing of our work is demonstrated below, where several issues are verified: (1) Sequence of foreground program will not be changed when interrupts come; (2) Data in register files is the same before and after the execution of ISR; (3) No matter when to raise the interrupt, the program should always produce the same results.

B.1 Sequence of Program

Figure B.1 shows the execution sequence of foreground program without any interrupts. Signal “mpc_r” indicates the pc address of every cycle. Whenever branch instruction is reached, the current instruction will be flushed and new branch address will be inserted into “mpc_r”.

![Figure B.1: Execution sequence of foreground program](image)

According to the Figure B.1, we can get the sequence as “0x00 to 0x09, 0x0c to 0x16, 0x0a” then “stop”. If we check the same signal after we insert interrupt into the system, we can get the same sequence for foreground program. (waveform is not shown here for simplicity reasons)

B.2 Data in Register Files

As we have discussed, all value in GRs and BRs should remain the same before and after the execution of ISRs. Figure B.2 shows that before and after ISR, the contents in GRs are the same. Figure B.3 depicts that BR values would be set to previous one after restoring. It is worth to note that data in GRs and BRs are changed during “running cs” state, but restored to original ones after “loading cs” state.

B.3 Execution Results

Figure B.4 illustrates the data that being stored into the data memory (version of hardware context switching - version 2). The last data “0x0A” is the value in link register.

Figure B.5 shows the final result that being written into data memory. “0x0F” and “0x0B” is coming from 2 different ISRs which are the same as we excepted.
In order to verify the results, following circumstances are tested. We raise the interrupt:

1. when the foreground program “CALL” a function;

**Figure B.2:** GR value storing and restoring

**Figure B.3:** BR value storing and restoring

**Figure B.4:** Context that stored in the data memory
B.3. **EXECUTION RESULTS**

Figure B.5: Data in data memory after execution

2. when the foreground program “writes” data into register files;

3. when the program “RET” from a function;

4. when the program is executing some normal instructions;

The testing results showed that the system produced the correct results under all of these situations.
In order to embed interrupt system into newer version of ρ-VEX, multiple things should be concerned like what method of interrupter being employed and how to integrate it in the source code level. You can find these information below.

Modifications on the original ρ-VEX are listed as follows. Besides these modifications, you also need to add the corresponding interrupt source code (int_handler.vhd; int_sche.vhd; int_ctrl.vhd; int_pkg.vhd; remoni.vhd for hardware context switching method). For detail modifications, please consult the source code for each method when you are embedding interrupt system into a newer version of ρ-VEX.

- **fetch.vhd**
  - Embed a multiplex that could select program counter from interrupter;
  - Move “stop pipeline” logic to decode stage where “rfi” signal comes from.

- **decode.vhd**
  - Remove operations for opcode “CTRL_RFI”.

- **branch_unit.vhd**
  - Add signal “rfi”.

- **decoder.vhd**
  - Connect signals between branch unit and interrupter;
  - Add logic for handle “stop pipeline”.

- **r-vex.vhd**
  - Add component of interrupter (“int_handler”);
  - Add component of register index monitor (“reg_monitor”) for hardware context switching method.
Curriculum Vitae

Quan Kong was born on June 20th, 1983 in Hangzhou, China. From 2001 to 2005, he obtained his bachelor diploma in Zhejiang University, Hangzhou, China. After 4 years professional work in the area of electronic engineering, he went to Delft University of Technology, the Netherlands for master education in September 2009. He followed the program of Embedded Systems in Delft University of Technology and joined the university’s Computer Engineering Group in June 2010.

His particular interests include but not limited to: computer arithmetic, (embedded) computer architecture, reconfigurable computing, (embedded) software engineering and real-time systems. His research on interrupt support on ρ-VEX processor resulted in a paper, which hopefully is going to be published in some conference.