On the Implementation of Traps for a Softcore VLIW Processor

Fakhar Anjam, Quan Kong, Roel Seedorf, and Stephan Wong

Computer Engineering Laboratory,
Delft University of Technology,
Mekelweg 4, 2628 CD, Delft, The Netherlands
Email: {F.Anjam, R.A.E.Seedorf, J.S.S.M.Wong}@tudelft.nl,
kongquanquan@hotmail.com

Abstract. This paper presents the design and implementation of traps (interrupts and exceptions handling systems) on an extensible and reconfigurable softcore very long instruction word (VLIW) processor: ρ-VEX. The interrupt subsystem is parameterized and implemented in four different mechanisms to match different application requirements in terms of performance issues (interrupt latency) and hardware consumption. The parameters include the number of interrupt vectors, interrupt priority for each vector, and the interrupt service routine (ISR) location address in the instruction memory. The exception subsystem is implemented on top of the interrupt subsystem. The designs are implemented on a Xilinx Virtex-6 FPGA and the interrupt response time can be reduced to 2 clock cycles. Utilizing the interrupt subsystem, we developed a task switching mechanism for a reconfigurable multi-core processor, where a program running on a core can be shifted to another core (same or different issue-width).

Keywords: Softcore, VLIW processor, Interrupts, Exceptions, Interrupt latency, Task switching

1 Introduction

The ρ-VEX [16] is an open-source reconfigurable and extensible softcore very long instruction word (VLIW) processor. It provides an efficient way to adapt to large number of applications. It can be parameterized at design time and/or reconfigured at run-time when implemented in a field-programmable gate arrays (FPGA). Due to the fact that a VLIW structure could lead to low slot utilization (high number of NOPs, in some cases it could go up to 50% [15]) for some applications or segments of certain code, we can reconfigure the entire or partial datapath of the processor at run-time to reduce the resource and power consumptions. Multiple smaller issue-width cores can be combined to create a larger issue-width core at run-time to exploit the instruction level parallelism (ILP) available in an application and improve the performance [4]. In order to implement these functionalities, more control from within the ρ-VEX processor is required. The processor should be interruptible by the outside world and remember its execution state.

More advanced application schemes can be realized on a processor when it implements interrupt and exception handling systems. Certain critical tasks require that the processor should respond to them within a certain time limit. The exception handling system ensures that the computed result is always correct. The interrupt and exception handling systems are important building blocks on a processor for running an operating system on it. Features like multi-tasking and multi-threading are facilitated by the interrupt system on a processor.

This paper presents the design and implementation of traps on the ρ-VEX processor. The trap is a collective name given to external asynchronous interrupt subsystem and internal synchronous exception handling subsystem. We implemented the interrupt subsystem in four different mechanisms with respect to interrupt latency, hardware consumption and the stress on the compiler and/or related toolchain. The interrupt subsystem is parameterized to support different applications. Parameters include the number of interrupt vectors, the interrupt priority for each vector and the ISR location address in the instruction memory. The exception handling subsystem utilizes the interrupt subsystem and ensures to detect certain conditions which may result in wrong computation. Building on the interrupt subsystem, we developed a mechanism for task switching in a reconfigurable multi-core system [4].

The remainder of the paper is organized as follows. Section 2 presents the related work. The ρ-VEX processor and its toolchain are discussed in Section 3. The design and implementation of the interrupt subsystem and the exception handling subsystem (which make the traps system) for the ρ-VEX processor are presented in Sections 4 and 5, respectively. Experimental results are discussed in Section 6. A use case (task switching) for the traps system on the ρ-VEX processor is presented in Section 7. Finally, Section 8 concludes our paper.

2 Related Work

The Xilinx Microblaze [2], the Altera Nios-II [3], and the OpenRISC [11] are 32-bit single-issue processors that can be configured to different application requirements. Because these processors have only one interrupt input, therefore, an interrupt controller is used. The service routine asks the controller what device(s) caused the interrupt and acts accordingly. When an interrupt is detected and the interrupts are enabled, the interrupt handler: (1) acknowledges the interrupt, (2) stores the context, (3) services the interrupt, (4) restores the context, and (5) returns to the main program. We do not use an interrupt controller.

The available softcore VLIW processors in literature are always restricted in some ways. Spyder [8] is the first reported softcore VLIW processor without a complete toolchain and interrupt system. Instance-specific implementations of VLIW processors are presented in [10] which do not represent more general VLIW processors. An FPGA-based design of a softcore VLIW processor based on the ISA of the Altera Nios-II soft processor is presented in [9]. Due to the licensed Altera Nios-II, this design is not much flexible and not open-source. The design can use the interrupt system of the Nios-II architecture. In [13], the micro-architecture of a customizable softcore VLIW processor is presented with the limitation of a compiler and interrupts system.

Several interrupt handling schemes to reduce the size of contexts to be switched to minimize the interrupt latency for VLIW and DSP processors are presented in [12][7][14]. All these mechanisms need the support of a relative compiler and even processor architecture. The ρ-VEX softcore is a parameterized open-source softcore VLIW processor and can be adapted to different applications. In this paper, we implemented the interrupt and exception handling systems on the ρ-VEX processor to further enhance its capabilities, and use it for task migration among different cores.

3 The ρ-VEX VLIW Processor

The VEX ISA is a 32-bit clustered VLIW ISA that is scalable and customizable to individual application domains. The VEX ISA is loosely modeled on the ISA of the HP/ST Lx (ST200) family of VLIW embedded cores [5]. Based on trace scheduling, the VEX C compiler is a parameterized ISO/C89 compiler. A flexible programmable machine model determines the target architecture, which is provided as input to the compiler. A VEX software toolchain including the VEX C compiler and the VEX simulator is made freely available by the Hewlett Packard Laboratories [1].

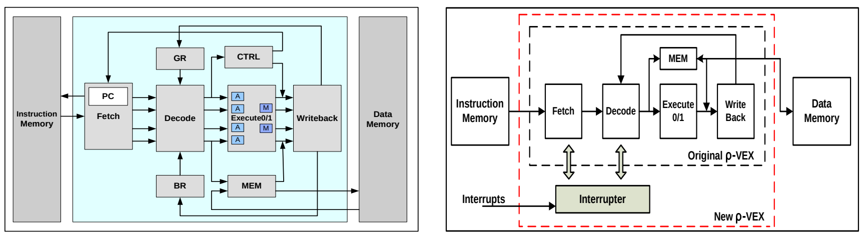
The ρ-VEX is a configurable (design-time) open-source softcore VLIW processor [15]. The ISA is based on the VEX ISA [6]. Different parameters of the ρ-VEX processor, such as the number and type of functional units (FUs), number of multiported registers (size of register file), number and type of accessible FUs per syllable, width of memory buses, and different latencies can be changed at design time. Figure 1(a) depicts the organization of a 32-bit, 4-issue ρ-VEX VLIW processor implemented in an FPGA. The ρ-VEX processor is a 5-stage pipelined processor consisting of fetch, decode, execute 0, execute 1/memory, and writeback stages. There are four arithmetic logic units (ALUs), two multiplication units (MUL), a control/branch unit (CTRL), and a load/store (LS) or memory unit (MEM). There are two multiported register files: a 64×32-bit general-purpose register (GR) file and an 8×1-bit branch register (BR) file. The instruction and data memories for the processor are implemented with block RAMs (BRAMs). The data memory is also utilized for storing the state or context of a program when an interrupt is serviced. Additionally, the ρ-VEX processor supports reconfigurable operations, as the VEX compiler supports the use of custom operations via pragmas inside the application code.

4 Interrupt Subsystem

The interrupt subsystem called the *interrupter* is designed in a modular fashion. It can be easily plugged in or out of the ρ-VEX core. Figure 1(b) depicts the structure of the ρ-VEX processor with the interrupter. The interrupter receives input signals from interrupt pins and then generates and sends control signals to the fetch stage to reschedule instructions to execute an ISR. Meanwhile, the context is also stored by the interrupter. When a *return from interrupt (RFI)* instruction is decoded, a signal is passed to the interrupter indicating the end of an ISR. After that the context is restored back and the core can continue with the original execution. The interrupter has two submodules: *interrupt scheduler* and *interrupt controller*. Figure 2 depicts a general view of our interrupt scheme.

4.1 Interrupt Scheduler

The interrupt scheduler is made parameterized in the number of interrupt vectors, interrupt priority for each vector, and the ISR location address in the instruction memory. The interrupt scheduler is responsible for (1) receiving interrupt input signals from different sources, (2) scheduling different tasks into the task queue, and (3) enabling interrupt requests to the interrupt controller when the priority of the requested task is higher than the current task. There are two inputs for the interrupt scheduler: external *interrupt in* signals from outside world and the internal *clear* interrupt flag signal from the interrupt controller. The *interrupt in* signal adds tasks to the task queue and *clear* signal removes it from the task queue. Only if an interrupt with higher priority comes in, or a higher priority task is finished, a waiting task can then become active. The interrupt vector table records information such as the interrupt vectors (type of interrupts) and their priorities, interrupt flags which show the status of each interrupt request, ISR addresses, and the interrupt enable bits to mask the interrupts. Figure 2(b) depicts the dataflow in the interrupt scheduler.



(a) A 4-issue ρ-VEX processor (b) The ρ-VEX organization with the interrupter

Fig. 1. The ρ-VEX processor and the interrupt scheme

4.2 Interrupt Controller

The interrupt controller's main jobs are (1) receiving interrupt request signals from the interrupt scheduler, (2) storing the context, (3) loading the ISR address (4) restoring the context, and (5) restarting the main program again from the point where it was left before the interrupt. The interrupt controller is designed as a finite state machine (FSM). An interrupt queue is implemented to record information of ISR addresses, return addresses and interrupt vectors received from the interrupt scheduler along with the interrupt request signal.

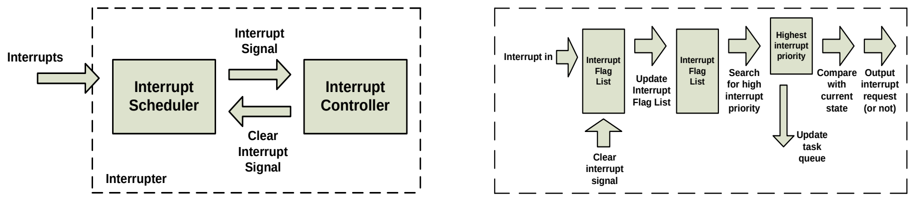
4.3 Implementation Types for the Interrupt Controller

We implemented the interrupt controller in four different methods in order to match different application requirements and resource usage. The first three methods utilize the ρ-VEX processor whose register file is implemented with the FPGA's configurable resources (slice registers and slice LUTs). The fourth method utilizes the ρ-VEX processor whose register file is implemented with the block RAMs (BRAMs). These implementations differ by the way the context (the general-purpose and the branch registers) is stored and restored.

Directly Switching Context Method: Here, the context is stored/restored through dedicated paths, and the processor pipeline is not utilized for context switching. The advantage of this method is that the size of the ISR code becomes smaller as instructions for context switching are not required in the ISR.

Hardware Instructions Switching Context Method: Here, the processor pipeline is utilized, and the context switching instructions are generated in the interrupt controller hardware. This method also reduces the ISR code size. Additionally, a hardware *monitor* is introduced which records the maximum index of registers at run-time in order to reduce the size of the context to be stored.

Software Instructions Switching Context Method: Here, the processor pipeline is utilized, and the instructions for context switching are generated in



(a) The interrupter (b) Dataflow in the interrupt scheduler

Fig. 2. General view of the interrupt scheme

software (compiler/assembler). The advantage is that extra hardware for the context switching is not required; however, this method introduces extra overhead for the size of the ISR code.

Page-able Register File Method: Instead of switching the context, here, the page of the register file is switched while executing an ISR. For this method, we utilized the ρ-VEX processor whose register file is implemented with BRAMs (instead of LUTs as in the first three methods) and the interrupt latency can be reduced to a minimum possible (2 cycles only). A 4-issue ρ-VEX processor requires a 64×32-bit register file with 4-write and 8-read ports. Multiple BRAMs are utilized to implement the register file. The 18 kbits BRAM-based register file can provide up to 512×32-bit registers or up to 8 copies or pages of 64×32-bit register file. We modified the design of the register file and exploited the extra unused available registers as multiple sets of the register file.

4.4 Software Interrupt Support and Interrupt Enable/Disable:

Since in the original VEX ISA there are no instructions for interrupts, therefore, we extended the instruction set with a custom instruction for the ρ-VEX processor. With this instruction, a software code can interrupt the core. The same instruction is also used for enabling or disabling (masking) the interrupts.

5 Exceptions Handling Subsystem

The difference between interrupts and exceptions 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.) The exception handling mainly relies on the interrupt subsystem for its implementation. Unlike the interrupts which can occur asynchronously, exceptions occur synchronously when an instruction is decoded or executed. Different conditions are tested at decode and execute stages and internal interrupt is raised whenever there is an exception. We enhanced the ρ-VEX processor and implemented the following important exceptions, and the system can further be easily extended.

1. Arithmetic Overflow
2. Division by Zero
3. Invalid Opcode
4. Unavailable Hardware Unit

6 Experimental Results

The following key metrics determine performance of an interrupt system: **Interrupt latency** – The time when the processor is ready to start saving the context after receiving an interrupt. **Interrupt response time** – The time when the processor runs the first instruction in an ISR after receiving an interrupt (interrupt latency plus the time for context storing and calling an ISR). Table 1 lists the implementation types and the interrupt response time for the four types of our interrupt subsystem. We utilized the Xilinx ISE release version 12.4 for synthesis and implementation with *XC6VLX240T-1-FF1156* as the target device available on the *ML605* development board.

In the ρ-VEX architecture, the general-purpose (GR) register number 0 ($r0.0) is hardwired to value zero, therefore, it is not stored during the context store. For version 1 of the interrupter, the interrupt response time of 76 cycles includes 4 cycles for completing the currently fetched instruction and stopping the pipeline, 1 cycle for scheduling the interrupt, 63 cycles for moving the GR registers and 8 cycles for moving the branch (BR) registers. For version 2, a hardware monitor records the maximum index of the registers that are used in the program before the processor is interrupted. Therefore, the interrupt response time depends on when the currently executing program is interrupted. The worst case could be 76 cycles. The best case could be 17 cycles (4 cycles for completing the currently fetched instruction and stopping the pipeline, 1 cycle for scheduling the interrupt, 12 cycles for moving the GR registers ($r0.1 to $r0.11, and $r0.63)). These registers have special purposes in the ISA and are mostly used in a program. For version 3, the interrupt response time is pre-determined at compile time. Still, there could be two scenarios. First, when the context storing routine is placed within the body of the ISR, the interrupt response time is 76 cycles. Second, when the context storing routine is placed at a separate location and is called from within the ISR, the interrupt response time is 78 cycles, as there would be an extra 2 cycles branch latency. In later case, the size of the ISR code is reduced.

The interrupt latency for version 4 is 2 clock cycles. One clock for scheduling the interrupt and other for switching the register file page. When implementing this method, the first and the last 4 instructions in the ISR should not read and write data from/to registers, respectively, in order to allow the currently fetched instruction to be passed through the pipeline. This is reasonable because at the beginning of a program (ISR), variables are normally initialized first before they can be read, and at the end of a program, the already computed data is consumed or spilled to memory instead of generating new data (writing to registers).

In order to evaluate our work, we compare the interrupt latency of the ρ-VEX processor to that of the *Microblaze* [2] and *Nios-II* [3] processors in Table 2, which also presents the hardware resource usage for our interrupt systems. The

Table 1. Implementation types and interrupt response time for the interrupt subsystem

Version	Description	Interrupt response time
1	directly switching context method	76 cycles
2	hardware instructions switching context method	17 – 76 cycles
3	software instructions switching context method	76 – 78 cycles
4	page-able register file method	2 cycles

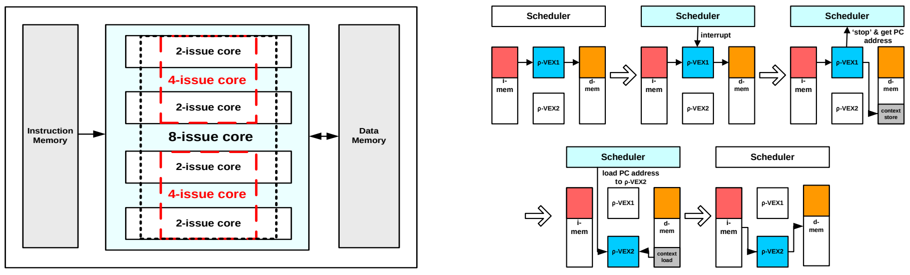
Table 2. Hardware resource utilization for the ρ-VEX and the worst-case interrupt latencies for the ρ-VEX, Microblaze, and Nios-II (without hardware divider unit)

Processor	Version	Registers	LUTs	BRAMs (36 kbits)	Interrupt latency
Original ρ-VEX	1, 2, 3	3055	23253	0	N/A
ρ-VEX with interrupt	1	3796	24510	0	5 cycles
ρ-VEX with interrupt	2	3711	24764	0	5 cycles
ρ-VEX with interrupt	3	3467	23585	0	5 cycles
Original ρ-VEX	4	1046	12899	16	N/A
ρ-VEX with interrupt	4	1727	15789	16	2 cycles
Microblaze	N/A	N/A	N/A	N/A	8 cycles
Nios-II/f/s/e	N/A	N/A	N/A	N/A	10/10/15 cycles

ρ-VEX processor can run up to 110 MHz in the Xilinx Virtex-6 *XC6VLX240T-1-FF1156* and the Altera Stratix IV *EP4SGX530KH40C2* FPGAs. The Microblaze and the Nios-II can run at about 200 MHz in the Virtex-6 and the Stratix IV FPGAs, respectively.

7 Use case

Task migration among cores is used to balance workload and/or network traffic and to reduce hotspots and power consumption. Additionally, a code running on a core can be migrated to a larger or smaller issue-width to increase the performance or reduce power consumption, respectively. We implemented the interrupt system (version 3) in each core of the adjustable/reconfigurable issue-slots VLIW multi-core processor [4]. In this processor, there are four 2-issue cores. Each core can be run independently. Multiple 2-issue cores can be combined or split at run-time, with the following possible configurations of VLIW cores: (1) four 2-issue cores, (2) one 4-issue and two 2-issue cores, (3) two 4-issue cores, and (4) one 8-issue core. Figure 3(a) depicts the general view of this adjustable issue-slots processor. In the original version of the multi-core processor, cores could only be combined or split when they were idle. With the development of



(a) Adjustable issue-slots processor (b) Mechanism for task switching

Fig. 3. The Adjustable issue-slots processor and the task switching mechanism

the interrupt system, each core is now able to pass on its environment (execution state) to another core of the same or different type in order to manage the cores utilization at run-time. We can now combine/split the cores that are even not idle. We implemented a task switching or migration mechanism for the cores.

A scheduler (currently implemented in hardware, but in future may be a software running on a certain core) controls the process of migration. For task migration from ρ-VEX1 (say, a 2-issue core) to ρ-VEX2 (say, a 4-issue core), the scheduler performs the following steps as depicted in Fig. 3(b): (1) generate an interrupt on ρ-VEX1 core, (2) a special ISR is called on ρ-VEX1 which stores the context into data memory (shared memory accessible to all cores) and records the program counter (PC) address with respect to a defined switching point, (3) generate an interrupt on ρ-VEX2 core, (4) a special ISR is called on ρ-VEX2 which restores the context from data memory, (5) load the recorded PC address to ρ-VEX2, and (6) start ρ-VEX2 to execute the remaining code. Here, we assume that different code versions of the same application are accessible and there are defined switching points available in the codes.

We only store/restore the GR registers and the branch registers. We implement the stack in the data memory accessible to both cores and hence, we do not switch the stack (both cores know the address of the stack). This reduces the migration time. A task migration from ρ-VEX1 to ρ-VEX2 requires a total of 154 cycles (76 cycles each for context store and restore and 1 cycle each for PC store and restore). Hence, switching an application running on a smaller to a larger issue-width core consumes 154 extra cycles, but then the execution time for the remaining part of the application can be reduced much. Table 3 presents the execution cycles and maximum possible performance improvement for some benchmark applications/kernels, when a code running on a 2-issue core is shifted to a 4-issue in the very beginning of the execution. Similarly, when a code running on a larger issue-width core is shifted to a smaller issue-width core and the unused issue-slots are gated off, power consumption can be reduced.

8 Conclusions

In this paper, we presented the design and implementation of traps on the ρ-VEX softcore VLIW processor. The trap includes external asynchronous interrupt and internal synchronous exception handling subsystems. The exception subsystem is built on the interrupt subsystem, which is parameterized and implemented in

Table 3. Execution cycles for the benchmark applications/kernels

Benchmark	2-issue	4-issue	Improvement
IDCT	8986	6864	23.61%
Sobel	5345	3720	30.40%
Matrix multiply	7908	6278	20.61%
Hamming distance	12475	8747	29.88%
FIR filter	86136	61660	28.42%

four different mechanisms to match different application requirements in terms of hardware consumption and performance issues. The parameters include the number of interrupt vectors, interrupt priority for each vector and the ISR location address in the instruction memory. We implemented our designs in a Xilinx Virtex-6 FPGA. All designs consume reasonable amount of hardware resources and the interrupt response time could be reduced to 2 clock cycles. Utilizing the interrupt subsystem, we devised a mechanism for task switching among different cores in a reconfigurable multi-core processor.

References

1. H. P. Laboratories: VEX Toolchain. http://www.hpl.hp.com/downloads/vex/
2. Xilinx Inc.: Microblaze Processor Reference Guide, EDK (v6.1). (2003)
3. Altera Corp.: Nios-II Software Developers Handbook. (2007)
4. Anjam, F., Nadeem, M., Wong, S.: Targeting Code Diversity with Run-time Adjustable Issue-slots in a Chip Multiprocessor. In: Design, Automation, and Test in Europe Conference. pp. 1358–1363 (2011)
5. Faraboschi, P., Brown, G., Fisher, J.A., Desoli, G., Homewood, F.: Lx: A Technology Platform for Customizable VLIW Embedded Processing. In: International Symposim on Computer Architecture. pp. 203–213 (2000)
6. Fisher, J.A., Faraboschi, P., Young, C.: Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools. Morgan Kaufmann (2005)
7. Hsieh, K.Y., Lin, Y.C., Huang, C.C., Lee, J.K.: Enhancing Microkernel Performance on VLIW DSP Processors via Multiset Context Switch. Journal of Signal Processing Systems 51, pp. 257–268 (2008)
8. Iseli, C., Sanchez, E.: Spyder: A Reconfigurable VLIW Processor using FPGAs. In: FPGAs for Custom Computing Machines. pp. 17–24 (1993)
9. Jones, A.K., Hoare, R., Kusic, D., Fazekas, J., Foster, J.: An FPGA-based VLIW Processor with Custom Hardware Execution. In: International Symposium on Field Programmable Gate Arrays. pp. 107–117 (2005)
10. Koester, M., Luk, W., Brown, G.: A Hardware Compilation Flow for Instance-Specific VLIW Cores. In: International Conference on Field Programmable Logic and Applications. pp. 619–622 (2008)
11. Lampret, D.: OpenRISC 1200 IP Core Specification. (2001)
12. Ozer, E., Sathaye, S.W., Menezes, K.N., Banerjia, S., Jennings, M.D., Conte, T.M.: A Fast Interrupt Handling Scheme for VLIW Processors. In: International Conference on Parallel Architectures and Compilation Techniques. pp. 136–141 (1998)
13. Saghir, M.A.R., El-Majzoub, M., Akl, P.: Customizing the Datapath and ISA of Soft VLIW Processors. In: International Conference on High Performance Embedded Architectures and Compilers. pp. 276–290 (2007)
14. Snyder, J.S., Whalley, D.B., Baker, T.P.: Fast Context Switches: Compiler and Architectural Support for Preemptive Scheduling. Microprocessors and Microsystems 19, pp. 35–42 (2000)
15. Wong, S., Anjam, F.: The Delft Reconfigurable VLIW Processor. In: International Conference on Advanced Computing and Communications. pp. 242–251 (2009)
16. Wong, S., van As, T., Brown, G.: ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor. In: International Conference on Field-Programmable Technologies. pp. 369–372 (2008)