ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor

Accepted at the IEEE International Conference on Field-Programmable Technology 2008 (ICFPT'08)

Stephan Wong, Thijs van As
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Delft, The Netherlands
Email: J.S.S.M.Wong@tudelft.nl, t.vanas@gmail.com

Geoffrey Brown
Department of Computer Science
Indiana University
Bloomington IN, USA
Email: geobrown@cs.indiana.edu

***Abstract*—This paper presents the architectural design of a reconfigurable and extensible Very Long Instruction Word (VLIW) processor. In addition to architectural extensibility, our processor also supports reconfigurable operations. Furthermore, we present an application development framework to optimally exploit the freedom of reconfigurable operations. Because our processor is based on the VEX ISA, we already have a good compiler which is able to deal with ISA extensibility and reconfigurable operations.**

Our results show that different configurations of our processor lead to considerable cycle count reductions for a selected benchmark application.

I. Introduction

The design of (embedded) architectures on reconfigurable hardware is becoming more popular now that classical drawbacks are diminishing. Field-Programmable Gate Arrays (FPGAs) are constantly improving and provide a technology platform that allows fast and complex reconfigurable designs. In many cases, the utilization of FPGAs implies a large reduction in development costs or an enormous speedup of the implemented algorithm. Applications in the multimedia domain happen to contain a lot of Instruction Level Parallelism (ILP), because they typically consist of many independent repetitive calculations. Very Long Instruction Word (VLIW) processors exploit ILP by means of a compiler that is completely aware of the target processor architecture.

In this paper, the design and implementation are presented of an embedded reconfigurable and extensible open source VLIW processor, accompanied by a development framework. Our processor architecture is based on the VEX Instruction Set Architecture (ISA), as introduced in [1]. VEX offers a scalable technology platform that allows variation in many aspects, including instruction issue width, organization of functional units, and instruction set. A software development toolchain for VEX is made freely available by Hewlett-Packard [2]. Our design provides mechanisms that allow parametric extensibility of the new processor, called ρ-VEX. Both reconfigurable operations, as well as the versatility of VEX machine models are supported by ρ-VEX. Our processor and framework are targeted at VLIW prototyping research and embedded processor design.

The remainder of this paper is organized as follows. Section II discusses related work and introduces the background for the VEX VLIW architecture. Section III describes the design of ρ-VEX, focusing on the processor organization, the instruction layout and the extensibility of our architecture. Subsequently, Section IV describes the application development framework. Experimental performance and resource usage results are presented in Section V. Finally, conclusions are presented in Section VI.

More information, as well as all source code can be found at http://r-vex.googlecode.com/.

II. Background

A. Related Work

Different softcore approaches resulted in FPGA-based system designs that achieved high performances. Well-known RISC softcore processors like MicroBlaze (Xilinx) and Nios II (Altera) provide efficient sequential architectures, optimized for the reconfigurable devices of their respective designers. However, these processors only expose a small degree of extensibility. Additionally, they are closed source and in many cases not free to use.

Approaches like MOLEN [3] and Chimaera [4] support issuing of reconfigurable operations. These approaches can be used as an extension or modification to existing architectures.

The first VLIW softcore processor found in literature is Spyder [5]. Later, several customizable VLIW softcore projects like [6], [7] and [8] were presented. Limitations of the former architectures are mainly the exposed extensibility (like adjusting the issue-width and changing the number of functional units), or the absence of a good software toolchain. In [9], a parametric customizable VLIW processor based on a subset of the EPIC ISA [10] is presented. This processor also supports reconfigurable operations. However, the complete support for custom operations throughout the (simulation) software and hardware toolchain and the flexible machine models that enable fast trade-off studies on functional units make our design stand out.

Another hardware implementation of a VEX machine is presented in [11]. In this implementation, VEX assembly is

used as an input to a more conventional hardware compiler. So instead of building a general purpose VEX VLIW processor to execute code, it converts the assembly code into hardware.

B. The VEX VLIW Architecture

The VEX (VLIW Example) ISA [1] is loosely modeled on the ISA of the HP/ST Lx [12] family of VLIW embedded cores. The VEX ISA supports multi-cluster machines, where each cluster provides a separate VEX implementation. Each cluster has support for multi-issue widths. The extensibility of the instruction set enables the definitions of special-purpose instructions in an organized way. VEX does not support floating point operations. By default, a VEX cluster has 4 ALU units, 2 multiplier (MUL) units, 1 branch control (CTRL) unit, 1 memory access (MEM) unit, 64 32-bit general-purpose registers (GR) and 8 1-bit branch registers (BR) per cluster. Also, an instruction- and data-memory cache of 32 kB is present. A VEX instruction consists of one or more syllables, depending on the issue-width. A syllable can be seen as a single 'RISC-style' instruction.

A VEX software toolchain is provided by Hewlett-Packard [2], which offers a C compiler and a simulator. Both tools are parametric by means of machine models. The VEX C compiler is a derivation of the Lx/ST200 C compiler, itself a descendant of the Multiflow C compiler. Profiling of compiled applications is supported via the GNU Profiler *gprof*. The VEX simulator is a *compiled simulator* which translates the target executable binary code to a binary executable that can run on the host system.

The choice for the VEX ISA was made because of the quality of the available toolchain and the highly configurable ISA. Lx performance benchmarks in [12] show that a 1-cluster Lx processor running at 300 MHz achieves higher a performance on application-specific SPECINT'95 benchmarks than a Pentium II at 333 MHz. These aspects provide a solid basis for a reconfigurable and extensible VLIW framework.

III. Design

A. Organization

A four-stage design consisting of *fetch*, *decode*, *execute* and *writeback* stages was applied for ρ-VEX. The standard configuration of a 1-cluster VEX machine was used for a default ρ-VEX processor. We decided not to implement instruction- and data-memory caches for our prototype, because the memory would be on-chip.

Figure 1 depicts the organization of a 4-issue ρ-VEX processor. The *fetch* unit fetches a VLIW instruction from the attached instruction memory, and splits it into syllables that are passed to the *decode* unit. In this stage, register contents used as operands are fetched from the register files. The actual operations take place in either the *execute* unit, or in one of the parallel CTRL or MEM units. ALU and MUL operations (respectively, A and M in Figure 1) are performed in the *execute* stage. This stage is implemented parametric, so that the number of ALU and MUL functional units could be adapted. All jump and branch operations are handled by the

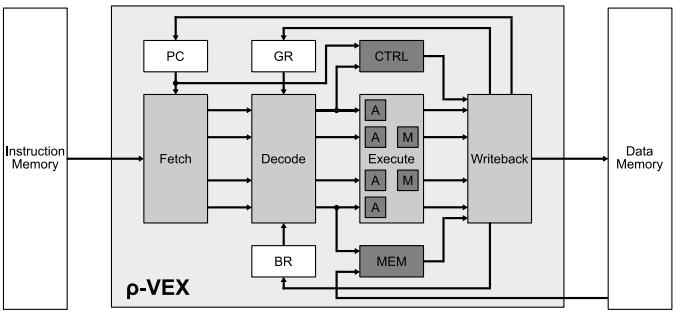

Fig. 1. ρ-VEX organization (4-issue)

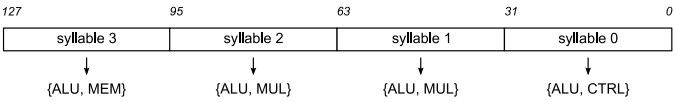

Fig. 2. Instruction layout

CTRL unit, and all data memory load and store operations are handled by the MEM unit. All write activities are performed in the *writeback* unit, to ensure that all targets (GR, BR, Program Counter (PC) or external memory)are written back at the same time. To determine the write targets per syllable, a `target` signal is assigned in the *decode* unit for each syllable. The different write targets could be the GR register file, BR register file, data memory or PC.

B. Instruction Layout

The standard set of VEX operations consists of 73 operations. Opcodes for the inter-cluster operations described by the VEX ISA are reserved, but not used as ρ-VEX (currently) supports only 1-cluster configurations. We complemented this default set of operations with two extra operations: **STOP** and **LONG_IMM**. The former operation tells ρ-VEX when to stop fetching instructions from the instruction memory. The latter is used when *long immediate* operands are handled. To be able to fit opcode bits, register addresses bits and syllable meta-data bits in one syllable, 32 bit syllables are used.

An instruction, by default consisting of four syllables, is a concatenation of all syllables with syllable 0 starting at bit 0. This is not very efficient, and there are many compression techniques possible to reduce the instruction size. Because this was not our primary concern for the current design, we left this unoptimized. Figure 2 depicts the instruction layout, as well as the allowed issue-slots per Functional Unit (FU). In order to utilize the FUs optimally, we spread them across the issue-slots evenly.

The VEX standard defines the use of three types of immediate operands: 9 bit *short immediate* operands, 24 bit *branch offset immediate* operands and 32 bit *long immediate* operands. The first two types are embedded in a single syllable, but the last one is spread over more syllables. We decided to change the size of a *branch offset immediate* operand to 12 bit, to use our syllable layout templates more efficient. Every syllable has an *immediate switch* field consisting of 2 bits that describe the type of immediate operand within the syllable. All ALU

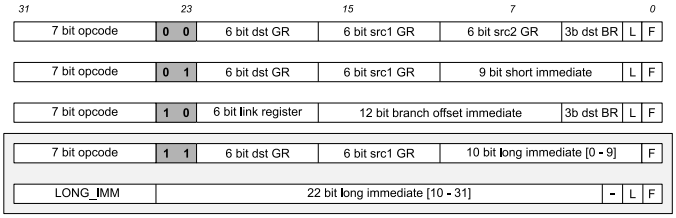

Fig. 3. Syllable layout

and MUL operations are overloaded to support both register operands as well as immediate operands.

Figure 3 depicts the syllable layout templates. The shaded bit-field shows the content of the *immediate switch*. Not all fields are evaluated in all cases. For example, when an ALU operation has no BR destination operand, the *3 bits dst BR* field holds *don't care* values. Special attention should be paid to the *long immediate* syllables. As these operands are always spread over two syllables, a **LONG_IMM** opcode could not be issued without a preceding syllable with its *immediate switch* set to `11`.

ρ-VEX syllables include two bits with syllable meta-data, the *L* and *F* bits. The *L* bit denotes whether a syllable is the last syllable in an instruction and the *F* bit denotes whether it is the first syllable in an instruction. As the first syllable of a *long immediate* operand could not be the last syllable in an instruction, this syllable does not provide an *L* field. In the current ρ-VEX prototype these fields are not evaluated, but these bit-fields allow the implementation of a more sophisticated syllable packing mechanism (as instructions with variable length can be evaluated this way).

All logical and select ALU operations can have a GR register or a BR register as their destination operand. When the GR destination address equals `$r0` (which is hardwired to zero/ground), the BR destination address is used to store the result of the operation. Four ALU operations operate on three source operands: two GR register operands, and one BR register operand. Because some of these operations are also able to operate on either a GR register or BR register as destination, a new location for the BR source register address should be assigned. We assigned special opcodes for these operations, so that the 4 most significant bits are unique. The least significant bits of the opcode field are used in this case to pack the BR source address.

C. Extensibility

ρ-VEX extensibility is provided by two mechanisms, which will be discussed separately.

1) ρ-OPS: The VEX software toolchain supports the use of custom instructions via pragmas inside the application code, as described earlier. With ρ-OPS we provide a mechanism to execute these operations in an ρ-VEX processor. In the current ρ-VEX prototype, it takes only a few lines of VHDL code to add a custom operation to the architecture. One of the 24 available ρ-OPS opcodes should be chosen, and a template VHDL function should be extended with the custom

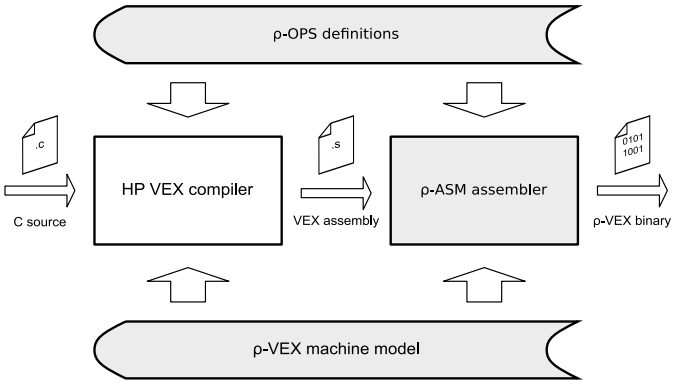

Fig. 4. Application development framework

functionality. ρ-OPS are not restricted combinatorial operators; sequential ρ-OPS consisting of multiple atomic operations are also allowed, as long as the design gets properly synthesized and routed.

2) VEX Machine Models: Currently, the following properties of ρ-VEX are parametric:

- Syllable issue-width
- Number of ALU units
- Number of MUL units
- Number of GR registers (up to 64)
- Number of BR registers (up to 8)
- Types of accessible FUs per syllable
- Width of memory busses

IV. Application Development Framework

To be able to efficiently perform experiments and develop applications for the ρ-VEX platform, a framework (depicted in Figure 4 is being worked on. It consists of two steps:

1) Compile a piece of C code with the VEX compiler. A VEX machine model should be present as a compiler directive when a custom configuration is targeted. When ρ-OPS are used, the code should be augmented with pragmas that define them.
2) The assembly file generated by the compiler should be assembled by ρ-ASM, which generates an instruction ROM for ρ-VEX. ρ-OPS definitions, as well as the machine model definitions should also be passed to ρ-ASM.

V. Experimental Results

The ρ-VEX organization has been described in VHDL and simulated with Mentor Graphics ModelSim SE 6.3d. Synthesis was performed with Xilinx Synthesis Technology (XST) from the Xilinx ISE 8.1.03i suite. As the target reconfigurable technology, the Xilinx Virtex-II PRO (XC2VP30) FPGA was chosen, embedded on the XUP V2P development board by Digilent.

All experiments were performed on a non-pipelined ρ-VEX system with 32 general purpose registers (GR). A data memory of 1 kB implemented using Block RAM was connected to ρ-VEX to store results. The issue width of ρ-VEX was varied

between 1, 2 and 4. All configurations had the same number of ALU units as their issue width. The 2- and 4-issue ρ-VEX configurations had 2 MUL units. ρ-ASM was used to assemble a hand coded VEX assembly benchmark. The application code was loaded in the instruction memory before synthesis.

We developed a debugging UART interface to transmit data via the serial RS-232 protocol. This interface invoked a transmission of the hexadecimal representation of the data memory contents, as well as the contents of the internal ρ-VEX cycle counter register.

A. Fibonacci Sequence Benchmark

We hand coded an assembly program that calculates the 45th Fibonacci number from the Fibonacci sequence. We created the code for 1-, 2- and 4-issue ρ-VEX configurations. We also created two ρ-OPS, **FIB3** and **FIB4**. These operations calculate, respectively, 3 and 4 Fibonacci iterations in one cycle. We adapted our 4-issue code to use the ρ-OPS as a final benchmark. The resulting numbers of executed clock cycles are presented in Table I (the ρ-OPS results are between brackets). Because the code was efficient to parallelize, we can see that the number of clock cycles almost halves when the issue width doubles. After using ρ-OPS, we see the expected speedup of almost 4. Because the core of the application consists of only 2 VLIW instructions, we are able to achieve such a high speedup by only adding 2 ρ-OPS.

B. Resource Usage

The aforementioned ρ-VEX configurations were synthesized without any memories connected, to check the resource usage. Table I presents the results of the measurements. Next to the number of slices, the percentage of slices used from the total available slices on the FPGA is presented. Increasing the issue-width has a large impact on the resource usage. This can be mainly ascribed to the growing GR register file. Because the GR register file in the 1-issue ρ-VEX can be totally implemented in dedicated Xilinx 2-port Block RAM, the implementation uses no slices. The 2- and 4-issue ρ-VEX configurations, however, require 4- and 8-port register files, respectively. These memory configurations can not be instantiated as Xilinx primitive elements, therefore they need to be formed by slices. An interesting trade-off might be a multi-cluster machine configuration consisting of single-issue ρ-VEX cores instead of one single-cluster, multi-issue ρ-VEX machine. As the VEX compiler has the ability to schedule data moves across VEX clusters, we already have architectural support for this.

All configurations were synthesized to run at the same clock speed of 89.44 MHz.

TABLE I
RESOURCE USAGE FOR DIFFERENT ρ-VEX CONFIGURATIONS

ρ-VEX	Cycles	Max. freq.	Slices	Slices GR
1-issue	1906	89.44 MHz	1895 (13%)	1 (0%)
2-issue	1080	89.44 MHz	5105 (37%)	3370 (24%)
4-issue	537 (141)	89.44 MHz	10433 (76%)	3927 (28%)

VI. CONCLUSIONS

In this paper, we presented ρ-VEX, an open source reconfigurable and extensible VLIW processor based on the VEX ISA [1]. Various architectural aspects like operation issue-width, the number of functional units and the sizes of register files are parametric. Reconfigurable operations are also supported by means of ρ-OPS. Because of the existing extensible VEX compiler by HP [2], we already possess a good compiler for our processor.

The processor is accompanied by an application development framework, to optimally exploit the various degrees of freedom for development. Our processor and framework are targeted at VLIW prototyping research and development of embedded processors. Experimental results showed cycle count reductions when exploiting the extensibility of the ISA and operations for a selected benchmark.

For future work we are interested whether pipelining and variations of pipeline and memory latencies would help the overall performance. Because the VEX machine model is already parametric in these areas, we already possess a very flexible framework to exploit such architectural trade-offs.

REFERENCES

[1] J. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2004.

[2] Hewlett-Packard Laboratories. VEX Toolchain. [Online]. Available: http://www.hpl.hp.com/downloads/vex/

[3] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN Polymorphic Processor," in *IEEE Transactions on Computers*, vol. 53, no. 11, Sep 2004, pp. 1363–1375.

[4] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera Reconfigurable Functional Unit," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 2, Feb 2004.

[5] C. Iseli and E. Sanchez, "Spyder: A Reconfigurable VLIW Processor using FPGAs," in *IEEE Workshop on FPGAs for Custom Computing Machines, 1993.*, Apr 1993, pp. 17–24.

[6] V. Brost, F. Yang, and M. Paindavoine, "A modular VLIW Processor," in *IEEE International Symposium on Circuits and Systems, ISCAS 2007.*, Apr 2007, pp. 3968 – 3971.

[7] A. Lodi, M. Toma, F. Campi, A. Cappelli, and R. Canegallo, "A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications," in *IEEE Journal on Solid-State Circuits*, vol. 38, no. 11, Jan 2003, pp. 1876 – 1886.

[8] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW Processor with Custom Hardware Execution," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2005, pp. 107–117.

[9] W. Chu, R. Dimond, S. Perrott, S. Seng, and W. Luk, "Customisable EPIC Processor: Architecture and Tools," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, vol. 3, Feb 2004.

[10] M. Schlansker and B. Rau, "EPIC: Explicitly Parallel Instruction Computing," in *Computer*, vol. 33, no. 2, Feb 2000, pp. 37–45.

[11] M. Koester, W. Luk, and G. Brown, "A Hardware Compilation Flow For Instance-Specific VLIW Cores," in *Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL08)*, Sep 2008.

[12] P. Faraboschi, G. Brown, J.A.Fisher, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," in *Proceedings of the 27th annual International Symposium of Computer Architecture*, June 2000, pp. 203–213.