# The Delft Reconfigurable VLIW Processor

Stephan Wong, Fakhar Anjam
Computer Engineering Laboratory
Delft University of Technology
Mekelweg 4, 2628 CD Delft,The Netherlands
E-mail: J.S.S.M.Wong@tudelft.nl, F.Anjam@tudelft.nl

***Abstract*—In this paper, we present the rationale and design of the Delft reconfigurable and parameterized VLIW processor called $\rho$-VEX. Its architecture is based on the Lx/ST200 ISA developed by HP and STMicroelectronics. We implemented the processor on an FPGA as an open-source softcore and made it freely available. Using the $\rho$-VEX, we intend to bridge the gap between general-purpose and application-specific processing through parameterization of many architectural and organizational features of the processor. The parameters include: instruction set (number and type of supported instructions), the number and type of functional units (FUs), issue-width (number of slots), register file size, memory bandwidth. The parameters can be set in a static or dynamic manner in order to provide the best performance or the best utilization of available resources on the FPGA. A complete toolchain including a C compiler and a simulator is freely available. Any application written in C can be mapped to the $\rho$-VEX processor. This VLIW processor is able to exploit the instruction level parallelism (ILP) inherent in an application and make its execution faster compared to a RISC processor system. This project creates research opportunities in the domain of softcore embedded VLIW processor prototyping, as well as designs that can be used in high-performance applications.**

**Keywords: Reconfigurable computing, FPGA, softcore, ILP, VLIW.**

## I. Introduction

Very Long Instruction Word (VLIW) processors can be used to increase the performance beyond normal Reduced Instruction Set Computer (RISC) architectures [1]. While RISC architectures only take advantage of temporal parallelism (by utilizing pipelining), VLIW architectures can additionally take advantage of the spatial parallelism by using multiple functional units (FUs) to execute several operations simultaneously. VLIW processor improves the performance by exploiting Instruction Level Parallelism (ILP) in a program. Field-programmable gate arrays (FPGAs) have become a widely used tool for rapid prototyping, providing both flexibility (as in software programming) and performance (as in dedicated hardware). Nowadays, FPGAs are moving beyond their simple prototyping beginnings towards mainstream products being utilized in many markets: general-purpose, high-performance, and embedded.

For an application to take advantage of performance improvement from FPGA, it must possess inherent parallelism, or the application source code should be structured in such a way to expose its parallelism. Applications in different domains such as multimedia, bio-informatics, wireless communication, numerical analysis etc. contain a lot of ILP, as they have many independent repetitive calculations. VLIW processors such as the Lx/ST200 [1] from HP and STMicroelectronics and the TriMedia [2] from NXP can exploit the ILP found in an application by means of a compiler. By issuing multiple operations in one instruction, a VLIW processor is able to accelerate an application many times compared to a RISC system [1][3].

This paper presents the design of an open source, extensible and reconfigurable softcore VLIW processor. The processor architecture is based on the VEX (VLIW Example) Instruction Set Architecture (ISA), as introduced in [4], and is implemented on an FPGA. Parameters of the VLIW processor such as the number and type of functional units (FUs), supported instructions, memory-bandwidth, and register file size can be chosen based on the application and the available resources on the FPGA. A software development toolchain including a highly optimizing C compiler and a simulator for VEX is made freely available by Hewlett-Packard (HP) [5]. We additionally present a development framework to optimally utilize the processor. Any application written in C can be executed on the processor implemented on the FPGA. The ISA can be extended with custom operations and the compiler is able to generate code for the custom hardware units, further enhancing the performance.

The remainder of the paper is organized as follows. Section II explains the rationale behind the project. In Section III, some previous work related to softcore processors is discussed. The VEX VLIW processor architecture and the available software toolchain are discussed in Section IV. Section V presents the design and implementation details of our softcore VLIW processor $\rho$-VEX. Finally, conclusions are presented in Section VI.

## II. The Rationale

The utilization of reconfigurable hardware (with the most common nowadays: field-programmable gate array (FPGA)) has increased tremendously in the past years due to their inherent parallelism[1] that can be exploited in order to improve the execution of many applications, e.g., multimedia, bio-informatics, and many large-scale scientific computing applications. Many approaches have been adopted to exploit recon-

[1]There are multiple factors that played a role, e.g., lowering cost of ownership, but these are not mentioned as the discussion is focussed on performance.

figurable hardware, but no single all-encompassing solution has emerged as each performs usually only very well for their particular environment or supported application(s). However, many of these solutions are hampered not by their ingenious designs but by the lack of tools to fully exploit the solution for more general-purpose cases. Therefore, we proposed the $\rho$-VEX processor as a reconfigurable and extensible VLIW softcore processor to bridge the gap between application-specific and general-purpose processing. In the following, we first highlight the advantages of our choice for a VLIW processor as a starting point:

- **simple hardware:** One of the main advantages of VLIW processors is that their hardware design is relatively simple compared to other RISC processors as there is no need for complex instruction decoders (e.g., out-of-order execution) in hardware since the compiler has already taken care of the instruction scheduling. *This means* that the hardware we need to implement on the FPGA can be kept simple and, therefore, higher clock frequencies can be achieved to improve performance. Furthermore, additional parallelism can be provided by simply adding more issue slots or functional units.
- **availability of existing tools:** Compilers for VLIW processor are readily available and research and development effort in improving them is still ongoing. Moreover, for the VEX that we have chosen as a basis, a simulator is available to investigate the performance gains for different architectural instances of the VEX processor. *This means* we can exploit existing compilers (and simulators) and future advancements without the need to dedicate much effort in their development.
- **no need for language translations:** Another benefit of using an existing VLIW architecture and its toolchain is that there is no longer need for translators and automatic synthesis tools. Nowadays, e.g., when looking at C-2-VHDL tools, restrictions must be placed on the C constructs before they can be utilized for the purpose of automatic hardware synthesis and sometimes code rewriting is necessary to achieve improved performance. In the latter, the (software) programmer needs to possess hardware knowledge, which is not always the case. *This means* we can take any existing code and compile it to our VLIW processor without rewriting code and without requiring the programmer to have hardware knowledge. We see a clear motivation for a reconfigurable VLIW processor between hardware design using automatic synthesis tools (starting from C) and manual design as adequate performance can be achieved after just the compilation time.

The choice for a VLIW processor clearly has its advantages and in the following, we will discuss reconfigurability-specific benefits we foresee:

- **static resource sharing:** When the size of the reconfigurable hardware structure and the available hardware area are known beforehand, one or several pre-configured VLIW softcore(s) can be instantiated and configured on the FPGA. In this manner, a short trade-off study, e.g., via a simulator or model, can determine the parameters most-suited for the available hardware and targeted application(s) at hand. This scenario is most suited for the embedded design environment as the requirements and platform are usually well-known and fixed. The sharing of resources between multiple VLIW processors is also pre-determined.
- **dynamic resource sharing:** When neither the application nor the precise characteristics of the attached reconfigurable hardware is known at design time, the most appropriate scenario is to allow for dynamic resource sharing. In this scenario, enough resources are instantiated to allow for sharing among the multiple VLIW processors running on the same chip. The method how to do this is under investigation and initial solutions have been proposed already.
- **on-the-fly resource instantiation:** When new resources are needed they can be instantiated on-the-fly. Similarly, when they are no longer needed their space can be freed and be dedicated to other applications.

The most promising solution to implement is most certainly the combination of the second and the third benefit mentioned above. On the other hand, one must not loose sight of certain intrinsic disadvantages of VLIW architectures that prevented it to become mainstream processors. However, we believe that these disadvantages are mainly due to their fixed design and many of these disadvantages can be mitigated when being implemented on reconfigurable hardware. We will highlight several issues[2] in the following and how they could be addressed:

- **varying instruction word widths:** Different applications contain different levels of parallelism (this is true even within the same application). In order to fully exploit this more issue slots should be used leading to longer (and therefore, different) instruction widths. Moreover, when using a different number of instructions can lead to a different encoding scheme of the VLIW instructions and thereby varying their length again. This issue can be easily dealt with by the reconfigurable nature of a reconfigurable and parameterized VLIW processor as different instruction decoders can be instantiated. This can be achieved with or without reconfiguring the issue slots (in the latter, unused issue slots can be shared among other different softcores).
- **high number of NOPs:** Due to the traditionally fixed implementation nature of VLIW processors, their organization may not completely match the parallelism inherent in the application leading to a high number of NOPs being scheduled. This leads to an under-utilization of the available resources (in some cases to over 50%). Instead

[2]The length of this paper does not allow for an extensive discussion of the shortcomings of VLIW processors and how they can be addressed. Therefore, we only mention the most important ones.

of idling issue slots, the reconfigurable VLIW processor can reconfigure the issue slots, or reduce the number of issue slots - i.e., either physically or enable sharing.

- **unbalanced issue slots:** This issue is tightly coupled with the previous issue as it is one of the causes for the scheduling of NOPs as functional units might not be available across all issue slots. This issue can be addressed by adding more functional units per issue slot.

Having stated how a reconfigurable and parameterized VLIW can overcome the traditional shortcomings of a VLIW processor, we will highlight in the following how such a reconfigurable VLIW processor can be used in two likely scenarios:

1) **stand-alone general-purpose processor:** In this scenario, complete applications (or application threads) run on the VLIW processor. The implementation of the processor can be fixed during the execution of multiple applications, but our envisioned reconfigurable VLIW processor should be able to adapt itself to different applications (or even to code portions within a single application).
2) **application-specific co-processor:** In this scenario, only specific kernels that require acceleration are being compiled to the VLIW processor. The benefits are: (1) no need for code rewriting, (2) avoidance of using complex tools such as C-2-VHDL translators, and (3) manual design of accelerators can be skipped. We have to note again that we are not stating that there is no need for the aforementioned actions or tools, but they can be avoided when the VLIW processor is capable of providing good enough performance within the requirements (such as, power, area) set.

Having stated the above, we present an advantage due to the existence of a reconfigurable and parameterized VLIW processor, namely instruction-set architecture (ISA) emulation. This means that we can implement different ISAs on top of the VLIW processor and ensure that each emulation is the most efficient. This will have the obvious advantage that applications compiled for different architectures can be executed without code recompilation (cumbersome) or software code emulation (slow). Moreover, having the mentioned ability allows for the following scenarios:

1) **ISA extension emulation:** When new ISA extensions are being introduced, much research and development effort is needed in order to ensure marked acceptance. However, with a reconfigurable ISA emulator it is possible to implement and ship the (draft) extension to potential end-users for actual use and evaluation. Furthermore, bug reports can lead to further improvements before the extension is fixed in hardware. The latter is still needed since the performance and power utilization of reconfigurable hardware is usually not optional. However, early-on experience of developers can lead to a much earlier market adoption of the intended ISA extension.
2) **instantiation of dedicated processor organizations:** When new processors are released, in many cases code recompilation is needed to take advantage of new organizational improvements. This need can be relaxed as dedicated organizational features can be provided in the reconfigurable hardware for particular already-compiled code.
3) **relaxation of backwards compatibility:** Rarely used instructions can be implemented in reconfigurable hardware and their implementation can be instantiated when needed. This means that complex instruction decoding hardware can be avoided leading to simpler hardware design and potentially lower power consumption.

By no means, our research in the design of the $\rho$-VEX processor is finished and there are still many open questions that need to be solved. However, discussing them is beyond the scope of this paper. In the remainder of this paper, we highlight several other similar approaches and describe more in-depth the design of our $\rho$-VEX processor and its current development status.

## III. Related Work

In literature, few softcore VLIW processors with a complete toolchain can be found. The first VLIW softcore processor found in literature is Spyder [6]. The design and implementation of Spyder marked the beginning of a reconfigurable VLIW softcore processor. Spyder consists of three reconfigurable units. A compiler toolchain was made available. One of the drawbacks of Spyder was that both the processor architecture and the compiler were designed from scratch. Because the designer had to put efforts in both directions, the processor did not evolve extensively.

Instance-specific VLIW processors are presented in [7][8]. These architectures are specific implementations for some applications, and do not represent a more general VLIW processor. A VLIW processor with reconfigurable instruction set is presented in [9]. An FPGA based design of a VLIW softcore processor is presented in [10]. Additionally, this processor is able to execute custom hardware. It has an ISA that is binary-code compatible with the Altera NIOS-II soft processor. To support this architecture, a compilation and design automation flow are described for programs written in C. The compilation scheme consists of a Trimaran [11] as the front-end and the extended NIOS-II as the back-end. Due to the licensed Altera NIOS-II, this VLIW design is less flexible and not open source.

In [12], a modular design of a VLIW processor is reported. Certain parameters of the processor architecture could be altered in a modular fashion. The lack of a good software toolchain and the absence of parametric extensibility limited the use of this architecture. In [13], the architecture and micro-architecture of a customizable soft VLIW processor are presented. Additionally, tools are discussed to customize, generate and program this processor. Performance and area trade-offs achieved by customizing the processor's datapath and ISA are evaluated. The limitation is the absence of a compiler. In [14], the design and architecture of a VLIW

microprocessor is presented without any tool chain, which restricts the processor usability.

In [3], we presented the design and implementation of a reconfigurable VLIW softcore processor called $\rho$-VEX. In addition, a development framework to utilize the processor is presented. The processor architecture is based on the VLIW Example (VEX) ISA, as introduced in [4]. VEX represents 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 the VEX architecture [5] is freely available from Hewlett-Packard (HP). The $\rho$-VEX processor is open-source and implemented on an FPGA. Different parameters such as the number and types of functional units, supported instructions, memory bandwidth, and size of register file can be chosen based on the application requirements and available resources on the FPGA. Initially, an instruction ROM file has to be generated for each application to be run on the processor and the design has to be re-synthesized along with the instruction ROM file. Now a boot loader like functionality has been added and the executable files can be downloaded to the instruction memory and executed directly avoiding the necessity of resynthesis.

## IV. The VEX VLIW Processor: Hardware and Related Software

Compared to superscalar and RISC processors, VLIW architecture requires a more powerful compiler due to more complex operation scheduling. [4] presents definition of the VLIW design philosophy as: *"The VLIW design philosophy is to design processors that offer ILP in ways completely visible in the machine-level program and to the compiler"*.

### A. The VEX System

The VEX stands for VLIW Example. VEX is a system developed according to the VLIW philosophy by Hewlett-Packard (HP). VEX includes three basic components [4]:

1) *The VEX ISA*: VEX Instruction Set Architecture (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 HP/ST Lx (ST200) family of VLIW embedded cores [1]. VEX ISA is scalable because different parameters of the processor such as the number of clusters, FUs, registers, and latencies can be changed. VEX ISA is customizable because special-purpose instructions can be defined in a structured way.
2) *The VEX C Compiler*: Based on *trace scheduling*, the VEX C compiler is an ISO/C89 compiler. It is derived from the Lx/ST200 C compiler, which itself is a descendant of the Multiflow C compiler. A very flexible programmable machine model determines the target architecture, which is provided as input to the compiler. This means that without the need to recompile the compiler, architecture exploration of the VEX ISA is possible with this compiler.
3) *The VEX Simulation System*: The VEX simulator is an architectural-level simulator that uses *compiled simulator* technology to achieve faster execution. It additionally provides a set of POSIX-like *libc* and *libm* libraries (based on the GNU *newlib* libraries), a simple built-in cache simulator (level-1 cache only) and an Application Program Interface (API) that enables other plug-ins used for modeling the memory system.

A VEX software toolchain including the VEX C compiler and the VEX simulator is made freely available by the Hewlett-Packard Laboratories [5]. The reason behind choosing the VEX architecture for our project is the scalability and customizability of the VEX ISA and the availability of the free C compiler and simulator, which can be used for architecture exploration.

### B. The VEX Instruction Set Architecture

VEX offers a 32-bit clustered VLIW ISA. VEX models a scalable technology platform for embedded VLIW processors that allows variations in the parameters of the processor. Following the VLIW design philosophy, the parameters of the processor, such as issue width, FUs, register files and processor instruction set can be varied. The compiler is responsible for scheduling the instructions. Along with basic data and operation semantics, VEX includes many features for compiler flexibility in scheduling multiple concurrent operations. Some of these features are [4]:

- Parallel execution units, such as integer ALUs and multipliers.
- Parallel memory pipelines, including access to multiple data memory ports.
- Data prefetching and other locality hints supported by the architecture.
- A large multiported shared register file made visible by the architecture.
- Partial predication through *select* operations.
- Multiple condition registers to make efficient branch architecture.
- Long immediate operands can be encoded in the same instruction.

Table I presents the parameters that could be changed for VEX VLIW processor.

The most basic unit of execution in VEX is an *operation*, which is equivalent to a typical RISC-style instruction. An

Table I
The VEX Design Parameters

| Processor Resource | Design Parameters |
|---|---|
| Functional Units | Number of FUs, type, supported instructions, degree of pipelining |
| Register File | Register size, register file size, number of read ports, number of write ports |
| Load/Store Unit | Number of memory ports, memory latency, cache size, line unit |
| Interconnection Network | Number and width of buses, forwarding connections between units |

encoded operation in VEX system is called a *syllable*. Multiple syllables are combined to form an *instruction*, which is an atomic unit of execution in a VLIW processor. The instruction issue-width is the number of syllables in an instruction that could be issued, and it depends on the number of FUs in the processor. An instruction having multiple syllables or operations is issued every cycle by the compiler to the multiple execution units of the VLIW processor, which is the main reason for performance compared to a RISC processor, which has an issue-width of one.

*1) Multicluster Organization:* The number of read ports of the shared multiported register file in a VLIW processor is twice the issue-width, and the number of write ports is equal to the issue-width (assuming that each FU requires two input operands and writes one output as a result). Therefore, the issue-width is proportional to the product of the number of read and write ports of the shared register file. The resource/area requirement for a multiported register file is directly proportional to the product of number of read and write ports, therefore these parameters are not scalable to a large extent or we can say that the issue-width is not scalable to a large extent. To reduce this pressure on the number of read and write ports of the shared register file, VEX defines a *clustered* architecture [4]. Using modular execution clusters, the VEX provides scalability of issue-width and functionality. A cluster is a collection of register files and a tightly coupled set of FUs.

VEX clusters are numbered from zero. Cluster 0 is a special cluster that must always be present in any VEX implementation, because the control operations execute on this cluster. Different clusters have different unit/register mixes, but a single Program Counter (PC) and a unified I-cache control them all, so that they run in lockstep or proper sequence [1]. The structure of a VEX multicluster architecture is depicted in Figure 1.

FUs within a cluster can only access registers in the same cluster. VEX provides a simple pair of *send-receive* instruction primitives that move data among registers on different clusters. These intercluster copy operations may consume resources

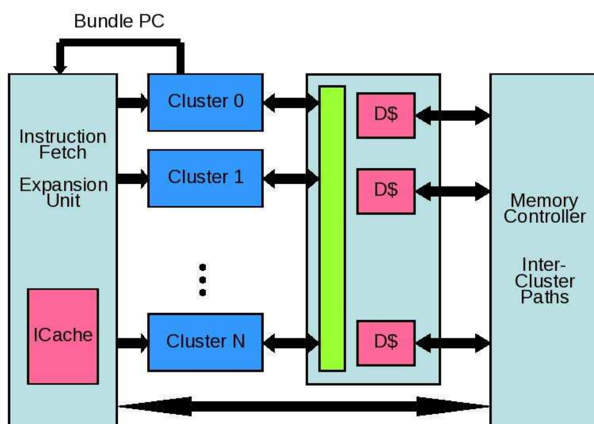


Figure 1. The VEX Multicluster Organization

in both the source and destination cluster and may require more than one cycle (pipelined or not). There is only a single instruction cache (I-cache), but different data cache (D-cache) ports and/or private memories can be associated with each cluster. This means that VEX allows multiple memory accesses to execute simultaneously. Figure 1 depicts multiple D-cache blocks, attached by a crossbar to different clusters, which allows a variety of memory configurations. VEX clusters obey the following set of rules [4]:

- Each cluster has the ability to issue multiple operations in the same instruction.
- Different clusters can have different issue-widths and different types of operations.
- Different clusters can have different VEX ISA, and not all clusters have to support the entire VEX ISA.
- All units within a cluster are indistinguishable or equally likely for selection. This means that the operations to be executed by a cluster do not have to be assigned to particular units within this cluster. To assign operations to the units within a cluster is the job of the hardware decoding logic.

*2) Structure of the Default VEX Cluster:* The default single VEX cluster is a 4-issue VLIW core, as depicted in Figure 2, and consists of the following units [4]:

- Four 32-bit integer ALUs
- Two 16x32 multipliers (MULs)
- One Load/Store Unit
- One Branch Unit
- 64 32-bit general-purpose registers (GRs)
- 8 1-bit branch registers (BRs)

This cluster can issue up to four operations per instruction. These operations could be either integer ALU, or MUL or Load/Store operations. All FUs are directly connected to registers, and no FU is directly connected to another FU. Two types of register banks are GR and BR. Both are multiported and shared register files. Memory units support only load and store operations, i.e., operations that act on memory and save results directly in memory are not supported by the VEX system. The branch unit (control unit) in the default cluster is used for program sequencing and is present only in cluster 0 in case of a multicluster machine.

### C. The VEX C Compiler

The VEX C compiler is derived from Lx/ST200 C compiler, which is itself derived from the Multiflow C compiler [15]. The Multiflow compiler includes high-level optimization algorithms based on Trace Scheduling [16]. The VEX C compiler is provided as a part of the freely available VEX toolchain by HP. The compiler supports the old C language, as well as the ISO/ANSI C. The toolchain has a command line interface and is provided in the form of binaries. Different command line options are provided for the compiler and the toolchain. Applications can be compiled with *profiling flags*, and *GNU gprof* can be used to visualize the profile data. Because the VEX processor is scalable and customizable,

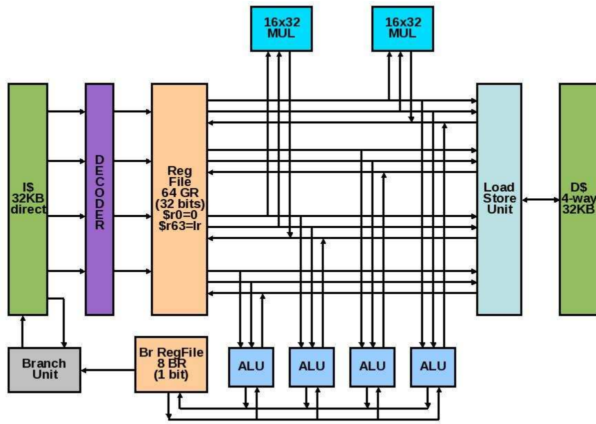


Figure 2. The Default VEX Cluster

the compiler supports this scalability and customizability. To compile for a different configuration, the compiler is provided with configuration information in the form of Machine Model Configuration (fmm) file. To include a custom instruction, the application code is annotated with pragmas. Different compiler pragmas are available to improve the performance. Refer to [4] for details on how to use the VEX compiler.

### *D. The VEX Simulation System*

The VEX toolchain provides tools that allow C programs compiled for a VEX VLIW configuration to be simulated on a host workstation. The VEX simulator is a fast *compiled simulator (CS)* that translates VEX binary to a host computer binary. It first converts VEX binary to C and then using the C compiler of a host generates a host executable. The compiled simulation workflow is depicted in the Figure 3.

The VEX simulator produces instrumentation code to count execution cycles and other statistics and generates a log file at the end of simulation. This log file has all the statistical information that can be analyzed for performance analysis and architecture exploration. The simulator provides a simple cache simulation library, which models an L1 instruction and data cache. The default cache simulation can be replaced by a user-defined library. In addition, the VEX simulator includes support for *gprof*, and different statistical files generated at the end of simulation can be used with *gprof* tool for analysis and profiling of the simulated application. Refer to [4] for details on how to use the VEX simulator.

## V. THE $\rho$-VEX SOFTCORE VLIW PROCESSOR

In [3], we presented the design and implementation of a reconfigurable and extensible softcore VLIW processor. We implemented a single-cluster standard configuration of VEX

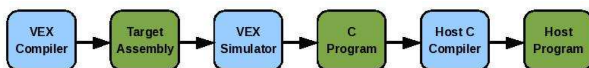


Figure 3. The VEX Simulation Flow

machine for our processor called $\rho$-VEX. Figure 4 depicts the organization of our 32-bit, 4-issue VLIW processor implemented on an FPGA. The $\rho$-VEX processor consists of *fetch*, *decode*, *execute* and *writeback* stages. The fetch unit fetches a VLIW instruction from the attached instruction memory, and splits it into syllables that are passed on to the decode unit. In the decode stage, the instructions are decoded and register contents used as operands are fetched from the register file. The actual operations take place in either the execute unit, or in one of the parallel CTRL or MEM units. ALU and MUL operations 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 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 are written back at the same time. Different write targets could be the GR register file, BR register file, data memory or PC.

The $\rho$-VEX implements all of the 73 operations of the VEX operations set. It additionally supports reconfigurable operations as the VEX compiler supports the use of custom instructions via pragmas inside the application code. In the current $\rho$-VEX prototype, it takes only a few lines of VHDL code to add a custom operation to the architecture. One of the 24 available reserved opcodes could be chosen, and a provided template VHDL function could be extended with the custom functionality. Currently, the following properties of $\rho$-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

To optimally exploit the processor utilization, a development framework is provided, which consists of compiling a piece of C code with VEX compiler and then generating a VHDL instruction ROM file by assembling the assembly file with our assembler [3]. The ROM file is then synthesized with the rest of the processor VHDL design files.

As the target reconfigurable technology, 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 $\rho$-VEX system with 32 general purpose registers (GR). A data memory of 1 kB implemented using BlockRAM was connected to $\rho$-VEX to store results. The issue-width of $\rho$-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 $\rho$-VEX configurations had 2 MUL units. 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

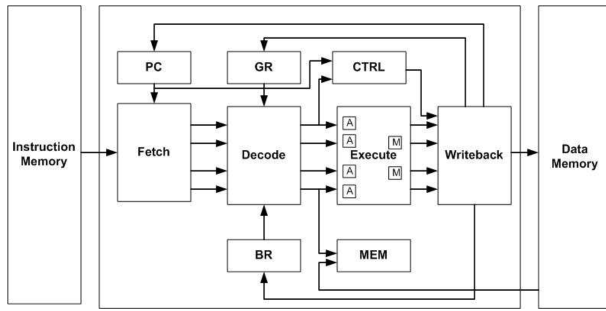


Figure 4. The $\rho$-VEX VLIW Processor

of the internal $\rho$-VEX cycle counter register. Synthesis results for $\rho$-VEX processor are presented in Table II.

### A. Recent Developments

The following design improvements are added to the original $\rho$-VEX processor:

- The assembler has been extended and it now generates a binary executable file for the processor. The hardware design is modified and BlockRAM is used as the instruction memory. The executable file can be downloaded into the instruction memory of the already placed processor on FPGA using a serial port on the PC and the FPGA development board, and therefore, re-synthesis and re-implementation of the processor with changing the application is not required.
- We implemented dynamically reconfigurable register file for the $\rho$-VEX processor to reduce the resources required by the multiported register file [17]. The VEX architecture supports up to 64 multiported shared registers in a register file for a single cluster VLIW processor. This register file accounts for a considerable amount of area when the VLIW processor is implemented on an FPGA. Our processor design supports dynamic partial reconfiguration allowing the creation of dedicated register file sizes for different applications. The processor can dynamically create its own register file composed of the actual number of registers the application needs. This means that valuable area can be freed and utilized for other implementations running on the same FPGA when not the full register size is needed. Our design needs 924 slices on a Virtex-2 Pro device for dynamically placing a chunk of 8 registers, and places registers in multiples of 8 registers to simplify the design. The processor does not need permanently 64 registers requiring 8594 slices thereby considerably reducing the slice utilization at run-time without increasing cycle count.

Table II
SYNTHESIS RESULTS FOR $\rho$-VEX PROCESSOR

| $\rho$-VEX | Slices | Max. Frequency |
|---|---|---|
| 1-issue | 1895 (13%) | 89.44 MHz |
| 2-issue | 5105 (37%) | 89.44 MHz |
| 4-issue | 10433 (76%) | 89.44 MHz |

## VI. CONCLUSIONS

In this paper, we presented the design and implementation of a reconfigurable softcore VLIW processor based on the Lx/ST200 ISA, developed by HP and STMicroelectronics. Our processor design called $\rho$-VEX is parametric and different parameters such as the number and type of functional units, supported instructions, memory bandwidth, register file size can be chosen depending upon the application and available resources on the FPGA. A toolchain including a C compiler and a simulator is freely available. We provide a development framework to optimally utilize the reconfigurable VLIW processor. Any application written in C can be mapped to the VLIW processor on the FPGA. This VLIW processor is able to exploit the instruction level parallelism (ILP) inherent in an application and make its execution faster compared to a RISC processor system. We described our rationale for the $\rho$-VEX processor and presented the possible advantages it can provide for its use as a general-purpose processor or application-specific co-processor.

## REFERENCES

[1] 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 (ISCA 00)*, June 2000, pp. 203 - 213.
[2] TriMedia Processor Series. http://www.nxp.com/.
[3] S. Wong, T.V. As, and G. Brown, "$\rho$-VEX: A Reconfigurable and Extensible Softcore VLIW Processor", in *IEEE International Conference on Field-Programmable Technologies (ICFPT 08)*, Taiwan, December 2008.
[4] J. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2004.
[5] Hewlett-Packard Laboratories. VEX Toolchain. [Online]. Available: http://www.hpl.hp.com/downloads/vex/.
[6] C. Iseli and E. Sanchez, "Spyder: A Reconfigurable VLIW Processor using FPGAs", in *FPGAs for Custom Computing Machines*, January 1993, pp. 17-24.
[7] C. Grabbe, M. Bednara, J.V.Z. Gathen, J. Shokrollahi, and J. Teich, " A High Performance VLIW Processor for Finite Field Arithmetic", in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS 03)*, April 2003.
[8] 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 (FPL 08)*, Sep 2008.
[9] 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.
[10] A.K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW Processor with Custom Hardware Execution", in *Proceedings of the 2005 ACM/SIGDA 13th Internal Symposium on Field Programmable Gate Arrays (FPGA 05)*, New York, NY, USA: ACM, 2005, pp. 107 - 117.
[11] http://www.trimaran.org/.
[12] 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.
[13] M.A.R. Saghir, M. El-Majzoub, and P. Akl, "Customizing the Datapath and ISA of Soft VLIW Processors", in *High Performance Embedded Architectures and Compilers (HiPEAC 07)*, LNCS 4367 pp. 276-290, Springer-Verlag Berlin Heidelberg 2007.
[14] W.F. Lee, *VLIW Microprocessor Hardware Design For ASICs and FPGA*. McGraw-Hill, 2008.

[15] P.G. Lowney et al, "The Multiflow Trace Scheduling Compiler", *The Journal of Supercomputing*, 7(1/2), 51-142, 1993.

[16] J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Trans. on Computers*, C-30(7), 478-490, 1981.

[17] S. Wong, F. Anjam, and M.F. Nadeem, "Dynamically Reconfigurable Register File for a Softcore VLIW Processor", *Accepted for publications in DATE 2010*.