A VLIW Architecture for Executing Scalar/Vector Instructions on Instruction Level Parallelism

ARSHIYA KAREEM¹, B. UPENDER²

¹PG Scholar, Dept of ECE, Shadan College of Engineering & Technology, Hyderabad, India. Email:cute.arshiya69@gmail.com.
²Associate Professor, Dept of ECE, Shadan College of Engineering & Technology, Hyderabad, India.

Abstract: This paper proposes new processor architecture for accelerating data-parallel applications based on the combination of VLIW and vector processing paradigms. It uses VLIW architecture for processing multiple independent scalar instructions concurrently on parallel execution units. Data parallelism is expressed by vector ISA and processed on the same parallel execution units of the VLIW architecture. The proposed processor, which is called VecLIW, has unified register file of 64x32-bit registers in the decode stage for storing scalar/vector data. VecLIW can issue up to four scalar/vector operations in each cycle for parallel processing a set of operands and producing up to four results. However, it cannot issue more than one memory operation at a time, which loads/stores 128-bit scalar/vector data from/to data cache. Four 32-bit results can be written back into VecLIW register file. The complete design of our proposed VecLIW processor is implemented using VHDL targeting the Xilinx FPGA Virtex-5, XC5VLX110T-3FF1136 device. The required numbers of slice registers and LUTs are 3,992 and 14,826 (14,570 for logic and 256 for memory), respectively. The number of LUT-FF pairs used is 17,425, where 13,433 for unused flip-flops, 2,599 for unused LUT, and 1,393 for fully used LUT-FF pairs.

Keywords: VLIW Architecture; Vector Processing; Data-Level Parallelism; Unified Data Path; FPGA/VHDL Implementation.

I. INTRODUCTION

One of the most important methods for achieving high performance is taking advantage of parallelism. The simplest way to take the advantage of parallelism among instructions is through pipelining, which overlaps instruction execution to reduce the total time to complete an instruction sequence (see [2] for more detail). All processors since about 1985 use the pipelining technique to improve performance by exploiting instruction-level parallelism (ILP). The instructions can be processed in parallel because not every instruction depends on its immediate predecessor. After eliminating data and control stalls, the use of pipelining technique can achieve an ideal performance of one clock cycle per operation (CPO). To further improve the performance, the CPO would be decreased to less than one. Obviously, the CPO cannot be reduced below one if the issue width is only one operation per clock cycle. Therefore, multiple-issue scalar processors fetch multiple scalar instructions and allow multiple operations to issue in a clock cycle. However, vector processors fetch a single vector instruction (ν operations) and issue multiple operations per clock cycle. Statically/dynamically scheduled superscalar processors issue varying numbers of operations per clock cycle and use in-order/out-of-order execution [3, 4]. Very long instruction word (VLIW) processors, in contrast, issue a fixed number of operations formatted either as one large instruction or as a fixed instruction packet with the parallelism among independent operations explicitly indicated by the instruction [5]. VLIW and superscalar implementations of traditional scalar instruction sets share some characteristics: multiple execution units and the ability to execute multiple operations simultaneously. However, the parallelism is explicit in VLIW instructions and must be discovered by hardware at run time in superscalar processors. Thus, for high performance, VLIW implementations are simpler and cheaper than super scalars because of further hardware simplifications.

However, VLIW architectures require more compiler support. See [6] for more detail VLIW architectures are characterized by instructions that each specify several independent operations. Thus, VLIW is not CISC instruction, which typically specify several dependent operations. However, VLIW instructions are like RISC instructions except that they are longer to allow them to specify multiple independent simple operations. A VLIW instruction can be thought of as several RISC instructions packed together, where RISC instructions typically specify one operation. The explicit encoding of multiple operations into VLIW instruction leads to dramatically reduced hardware complexity compared to superscalar. Thus, the main advantage of VLIW is that the highly parallel implementation is much simpler and cheaper to build the equivalently concurrent RISC or CISC chips. See [7] for architectural comparison between CISC, RISC, and VLIW.
On multiple execution units, this paper proposes new processor architecture for accelerating data-parallel applications by the combination of VLIW and vector processing paradigms. It is based on VLIW architecture for processing multiple scalar instructions concurrently. Moreover, data-level parallelism (DLP) is expressed efficiently using vector instructions and processed on the same parallel execution units of the VLIW architecture. Thus, the proposed processor, which is called VecLIW, exploits ILP using VLIW instructions and DLP using vector instructions.

The use of vector instruction set architecture (ISA) lead to expressing programs in a more concise and efficient way (high semantic), encoding parallelism explicitly in each vector instruction, and using simple design techniques (heavy pipelining and functional unit replication) that achieve high performance at low cost [8, 9]. Thus, vector processors remain the most effective way to exploit data-parallel applications [10, 11]. Therefore, many vector architectures have been proposed in the literature to accelerate data-parallel applications [12]. Commericially, the Cell BE architecture is based on heterogeneous, shared-memory chip multiprocessing with nine processors: Power processor element is optimized for control tasks and the eight synergistic processor elements (SPEs) provide an execution environment optimized for data processing. SPE performs both scalar and data-parallel SIMD execution on a wide data path. NEC Corporation introduced SX-9 processors that run at 3.2 GHz, with eight-way replicated vector pipes, each having two multiply units and two addition units. The peak vector performance of SX-9 processor is 102.4 GFLOPS. For non-vectorized code, there is a scalar processor that runs at half the speed of the vector unit, i.e. 1.6 GHz.

To exploit VLIW and vector techniques, Salami and Valero proposed and evaluated adding vector capabilities to a μSIMD-VLIW core to speed-up the execution of the DLP regions, while reducing the fetch bandwidth requirements are introduced a VLIW vector media coprocessor, “vector coprocessor (VCP),” that included three asymmetric execution pipelines with cascaded SIMD ALUs. To improve performance efficiency, they reduced the area ratio of the control circuit while increasing the ratio of the arithmetic circuit. This paper combines VLIW and vector processing paradigms to accelerate data-parallel applications. On unified parallel data path, our proposed VecLIW processes multiple scalar instructions packed in VLIW and vector instructions by issuing up to four scalar/vector operations in each cycle. However, it cannot issue more than one memory operation at a time, which loads/stores 128-bit scalar/vector data from/to data cache. Four 32-bit results can be written back into VecLIW register file. The complete design of our proposed VecLIW processor is implemented using VHDL targeting the Xilinx FPGA Virtex5, XC5VLX110T-3FF1136 device. The rest of the paper is organized as follows. Block Diagram of Generic VLIW Processor in Section II. The VT Architectural Paradigm in Section III. Section IV describes the FPGA/VHDL implementation of VecLIW. Finally, Section V concludes this paper and gives directions for future work.

II. BLOCK DIAGRAM OF GENERIC VLIW PROCESSOR

VLIW architectures offer high performance at a much lower cost than dynamic out-of-order superscalar processors. By allowing the compiler to directly schedule machine resource usage, the need for expensive instruction issue logic is obviated. Furthermore, while the enormous complexity of superscalar issue logic limits the number of instructions that can be issued simultaneously, VLIW machines can be built with a large number of functional units allowing a much higher degree of instruction-level parallelism (ILP). VLIW instructions indicate several independent operations. Instead of using hardware for parallelism, VLIW processors use compiler that generates the VLIW code to clearly specify parallelism.

![Fig.1. Block diagram of generic VLIW implementation](image-url)

In VLIW complexity of hardware is moved to software. This trade-off has a benefit: only once the complexity is paid when the compiler is written instead of every time a chip is fabricated. Smaller chip, which leads to increased profits for the microprocessor vendor and/or cheaper prices for the customers. It’s easier to deal Complexity with in a software design than in a hardware design. Thus, the chip may cost less to design, be quicker to design, and may require less debugging, all of which are factors that can make the design cheaper. Improvements to the compiler can be made after chips have been fabricated; improvements to superscalar dispatch hardware require changes to the microprocessor, which naturally incurs all the expenses of turning a chip design. VLIW instruction format encodes an operation for every execution unit. This shows that every instruction will always have something useful for every execution unit. Unfortunately it’s not possible to pack every instruction with work for all execution units. Also, in a VLIW machine that has both integer and floating-point execution units, the best compiler would not be able to keep the floating point units busy during the execution of an
A VLIW Architecture for Executing Scalar/Vector Instructions on Instruction Level Parallelism

integer-only application. The problem with some VLIW instructions is that they do not make full use of all execution units which results in waste of precious processor resources like waste of instruction memory space, instruction cache space, and bus bandwidth.

There are two solutions to reduce the waste of resources. 1. Instructions can be compressed with a more highly-encoded representation. Different techniques, such as Huffman encoding can be employed to allocate the fewest bits to the most frequently used operations. 2. To define an instruction word that encodes fewer operations than the number of available execution units.

A. Generic Architecture of VLIW Processor

Each 128-bit VLIW instruction word consists of two operations. The architecture is built such that two operations can be executed in parallel to maximize the performance ability. Each operation uses a register file. Register file consist a set of sixteen internal registers each are of 64-bit.

Fig.2. Block diagram of VLIW architecture

On multiple execution units, this paper proposes new processor architecture for accelerating data-parallel applications by the combination of VLIW and vector processing paradigms. It is based on VLIW architecture for processing multiple scalar instructions concurrently. Moreover, data-level parallelism (DLP) is expressed efficiently using vector instructions and processed on the same parallel execution units of the VLIW architecture. Thus, the proposed processor, which is called, exploits ILP using VLIW instructions and DLP using vector instructions. The use of vector instruction set architecture (ISA) lead to expressing programs in a more concise and efficient way (high semantic), encoding parallelism explicitly in each vector instruction, and using simple design techniques (heavy pipelining and functional unit replication) that achieve high performance at low cost. Thus, vector processors remain the most effective way to exploit data-parallel applications each operation is divided into four stages:

1. Fetch stage
2. Decode stage
3. Execute stage and
4. Write back stage.

Fetch stage: The next instruction is fetched from the memory address that is currently stored in the program counter (PC), and stored in the instruction register (IR). At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.

Decode stage: interprets the instruction. During this cycle the instruction inside the IR (instruction register) gets decoded.

Execute stage: The control unit of the CPU passes the decoded information as a sequence of control signals to the relevant function units of the CPU to perform the actions required by the instruction such as reading values from registers, passing them to the ALU to perform mathematical or logic functions on them, and writing the result back to a register. If the ALU is involved, it sends a condition signal back to the CU.

Write back stage: The result generated by the operation is stored in the main memory, or sent to an output device. Based on the condition of any feedback from the ALU, Program Counter may be updated to a different address from which the next instruction will be fetched during the decode stage, data are read from the register file and during write back stage, data are written into the register file. Based on these requirements, the VLIW microprocessor is implemented. The incoming instructions and data from external systems to the VLIW microprocessor are fetched by the fetch unit.

After the instruction and data have been fetched, it is given to the decode stage. The 128-bit instruction consists of two operations. Each operation is given to the corresponding decode stage. Each operation is also passed from the fetch stage to the register file to allow the data to be read from the register file for each corresponding operation. In the decode stage, the operations are decoded and passed onto the execute stage. The execute stage, as its name implies, will execute the corresponding decoded operation. The execute stage has access to the shared register file for reading of data during execution. Upon completion of execution of an operation, the final stage (write back stage) will write the results of the operation into the register file, or read data to the output of the VLIW microprocessor for read operation.

B. Top Level Architecture

Instructions and data are fetched using an external instruction memory that has its own instruction cache. The defined VLIW microprocessor loads instructions and data directly from the external instruction memory through the 6-bit bus interface word and the 128-bit bus interface data. The output interface signal jump from the VLIW microprocessor is feedback as and input to the external
in an indicator that a branch has been taken and the instruction memory needs to pass another portion of instructions and data to the VLIW microprocessor. The top level architecture of VLIW processor is shown in the below Fig.3.

![Fig.3. Top level architecture of VLIW.](image)

III. THE VT ARCHITECTURAL PARADIGM

Parallelism and locality are the key application characteristics exploited by computer architects to make productive use of increasing transistor counts while coping with wire delay and power dissipation. Conventional sequential ISAs provide minimal support for encoding parallelism or locality, so high-performance implementations are forced to devote considerable area and power to on-chip structures that extract parallelism or that support arbitrary global communication. The large area and power overheads are justified by the demand for even small improvements in performance on legacy codes for popular ISAs. Many important applications have abundant parallelism, however, with dependencies and communication patterns that can be statically determined. ISAs that expose more parallelism reduce the need for area and power intensive structures to extract dependencies dynamically. Similarly, ISAs that allow locality to be expressed reduce the need for long range communication and complex interconnects. The challenge is to develop an efficient encoding of an application’s parallel dependency graph and to reduce the area and power consumption of the micro-architecture that will execute this dependency graph.

In this paper, we unify the vector and multithreaded execution models with the vector-thread (VT) architectural paradigm. VT allows large amounts of structured parallelism to be compactly encoded in a form that allows a simple micro-architecture to attain high performance at low power by avoiding complex control and data path structures and by reducing activity on long wires. The VT programmer’s model extends a conventional scalar control processor with an array of slave virtual processors (VPs). VPs execute strings of RISC-like instructions packaged into atomic instruction blocks (AIBs). To execute data-parallel code, the control processor broadcasts AIBs to the entire slave VPs. To execute thread parallel code, each VP directs its own control flow by fetching its own AIBs. Implementations of the VT architecture can also exploit instruction-level parallelism within AIBs. In this way, the VT architecture supports a model of parallelism. This flexibility provides new ways to parallelize codes that are difficult to vectorize or that incur excessive synchronization costs when threaded. Instruction locality is improved by allowing common code to be factored out and executed only once on the control processor, and by executing the same AIB multiple times on each VP in turn. Data locality is improved as most operand communication is isolated to within an individual VP.

We are developing a prototype processor, SCALE, which is an instantiation of the vector-thread architecture designed for low-power and high-performance embedded systems. As transistors have become cheaper and faster, embedded applications have evolved from simple control functions to cell phones that run multitasking networked operating systems with real-time video, three-dimensional graphics, and dynamic compilation of garbage collected languages. Many other embedded applications require sophisticated high-performance information processing, including streaming media devices, network routers, and wireless base stations. In this paper, we show how benchmarks taken from these embedded domains can be mapped efficiently to the SCALE vector thread architecture. In many cases, the codes exploit multiple types of parallelism simultaneously for greater efficiency. An architectural paradigm consists of the programmer’s model for a class of machines plus the expected structure of implementations of these machines. This section first describes the abstraction VT architecture provides to a programmer, and then gives an overview of the physical model for a VT machine.

A. VT Abstract Model

The vector-thread architecture is a hybrid of the vector and multithreaded models. A conventional control processor interacts with a virtual processor vector (VPV), as shown in Fig.4. The programming model consists of two interacting instruction sets, one for the control processor and one for the VPs. Applications can be mapped to the VT architecture in a variety of ways but it is especially well suited to executing loops; each VP executes a single iteration of the loop and the control processor is responsible for managing the execution. A virtual processor contains a set of registers and has the ability to execute RISC-like instructions with virtual register specifies. VP instructions are grouped into atomic instruction blocks (AIBs), the unit of work issued to a VP at one time. There is no automatic program counter or implicit instruction fetch mechanism for VPs; all instruction blocks must be explicitly requested by either the control processor or the VP itself.
A VLIW Architecture for Executing Scalar/Vector Instructions on Instruction Level Parallelism

IV. FPGA/VHDL IMPLEMENTATION OF VECLIW PROCESSOR

The complete design of the VecLIW is implemented using VHDL targeting the Xilinx FPGA Virtex-5, XC5VLX110T-3FF1136 device. A single Virtex-5 configurable logic blocks (CLB) comprises two slices, with each containing four 6-input LUTs and four flip-flops, for a total of eight 6-LUTs and eight flip-flops per CLB. Virtex-5 logic cell ratings reflect the increased logic capacity offered by the new 6-input LUT architecture. The Virtex-5 family is the first FPGA platform to offer a real 6-input LUT with fully independent (not shared) inputs. By properly loading LUT, any 6-input combinational function can be implemented. Moreover, the LUT can also be configured as a 64×1 or 32×2 distributed RAM. Besides, a single LUT supports a 32-bit shift register for more details. Fig. 5 shows the top-level RTL schematic diagram of our proposed VecLIW. It is generated from synthesizing the VHDL code of the VecLIW processor on Xilinx ISE 13.4. Our implementation of the VecLIW pipeline consists of five components: Fetch Stage, Decode Stage, Execute Stage, Memory_Stage, and Writeback_Stage. The first component (Fetch Stage) has three modules: Program counter, Instruction_adder, and Instruction memory.

The 32-bit output address of the Program counter module is sent to the input of the Instruction memory module to fetch 128-bit VLIW from instruction cache when read enable (RdEn) control signal is asserted. Note that the write enable (WrEn) control signal is used to fill in the instruction cache with the program instructions through 128-bit WrVLIW input. Depending on the PcUpdVal control signal (‘1’ for branch and ‘0’ for other instructions), the input address of Program counter module is connected to the next sequential address (PC + 16: since each VLIW is 4×32-bit or 16 bytes) or the branch address (BrAddr) coming from the decode stage. Program counter is updated synchronously when PcUpdVal is asserted. The outputs of the fetch stage, next PC (128-bit NPC) and 128-bit VLIW instruction, are sent to the decode stage through IF/ID pipeline register. To accelerate the fetch stage, a carry-lookahead adder is implemented in the Instruction_adder module to increment PC by 16. In addition to 128×128-bit single-port RAM, 327 LUT-FF pairs are needed for implementing the fetch stage modules (see Table 1). Note that the fetch stage of VecLIW exceeds the corresponding baseline scalar processor with the same size of the instruction memory (128×128-bit) by only 104 LUT-FF pairs.

The second component (Decode Stage) has three modules: Control unit, Register file, and Hazard detection. Depending on the control signals 4×1-bit IsUnSgn, the 4×1-bit immediate values are signed/unsigned-extended to 4×32-bit. VecLIW register file has eight read ports and four write ports. In the decode stage, the fetched VLIW (4×32-bit instructions) is decoded and the register file is accessed to read multi-scalar/vector elements (4×32-bit RsVal and 4×32-bit RtVal). The 2×4×32-bit outputs of the VecLIW registers are fed to the execute stage through ID/EX pipeline register. Note that decoding is done in parallel with reading registers, which is possible because these fields are at a fixed location in the VecLIW instruction formats. Since the immediate portions of VLIW are located in an identical place in the I-format of VecLIW instructions, the extension of the immediate values are also calculated during this cycle just in case it is needed in the next cycle.

The control unit of VecLIW is based on microprogramming, where control variables are stored in ROM (control memory). Each word in the control memory specifies the following control signals: IsUnSgn (0/1 for signed/unsigned instruction), RdRtDes (0/1 if destination is RT/RD), IsImmIns (0/1 for immediate/un-immediate instruction), RdEn (1 for load instruction), WrEn (1 for store instructions), Wr2Reg (1 when instruction writes result in register), and AluOpr (5-bit ALU operations). Hazard detection module in the decode stage can stall the fetch stage by disallowing PC to be updated (PcUpdEn = ‘0’). Detecting interlocks early in the pipeline reduces the hardware complexity because the hardware never has to suspend an instruction that has updated the state of the processor, unless the entire processor is stalled [2]. In VecLIW, Hazard detection module interlocks the pipeline for a RAW (read after write) hazard with the source coming from a load instruction. If there is a RAW hazard with the source instruction being a load, the load instruction will be in the execute stage (RdEn = ‘1’) when an instruction that needs the loaded data will be in the decode stage. Beside load interlock, the fetch stage should be stalled during the execution of vector instructions. A vector instruction processing v-element stalls the fetch stage \(\frac{v}{4} + 1\) clock cycles in the 8-element vector instruction, processing the second four elements results in activating PcUpdEn.

Moreover, the PcUpdVal is given value ‘1’ to increments the current PC by 16. As shown on Table 1, 10,720 LUT-FF pairs are required to be implemented the Decode Stage component on FPGA. The complexity of the VecLIW implementation is generated from synthesizing the VHDL code of the VecLIW processor.
decode stage is 413% (10720/2597) higher than the corresponding scalar decode stage because the size of VecLIW register file (8-read and 4-write ports) is about four times the size of the scalar register file (2-read and 1-write ports). Moreover, the decode unit of VecLIW decodes four individual instructions instead of one in case of the baseline scalar processor. Besides, the hazard detection unit of VecLIW checks RAW hazard with the source instruction being a load against the four individual instructions in VLIW instead of one scalar instruction. The execute stage of VecLIW have two modules: ALUs and Forward unit. The ALUs operate on four pairs of operands prepared in the decode stage. ALU\textsubscript{i} perform operations depending on the VLIW opcodes (4×5-bit AluOpr) and functions (4×8-bit Function). For load/store operations, the first ALU adds the operands (RsVal\textsubscript{i} and ImmVal\textsubscript{i}) to form the effective address; however, the remaining ALUs are ideal. For register-register operations, the ALU\textsubscript{i} performs the operation specified by the 8-bit function, control signals, on the values 32-bit RsVal\textsubscript{i} and 32-bit RtVal\textsubscript{i}, where 1 ≤ i ≤ 4. For register immediate operations, the ALU\textsubscript{i} performs the operation specified by the 5-bit AluOpr\textsubscript{i} control signals on the values in 32-bit RsVal\textsubscript{i} and 32-bit ImmVal\textsubscript{i}. In all cases, the result of the ALUs (4×32-bit AluOut) is placed in the EX/MEM pipeline register.

Instead of complicate the decode stage, the forward unit of VecLIW is in the execute stage. The key observation needed to implement the forwarding logic is that the pipeline registers contain both the data to be forwarded as well as the source and destination register fields. In VecLIW processor, all forwarding logically happens from the ALUs or data memory outputs to the ALUs inputs and data to be written in memory (Wr Data). Thus, the forwarding can be implemented by comparing the destination registers of the instructions contained in the EX/MEM or MEM/WB stages against the source registers of the instructions contained in the ID/EX registers. In addition to the comparators and combinational logic required to determine when a forwarding path needs to be enabled, the multiplexers at the ALUs inputs should be enlarged. The execute stage has the about four times higher complexity as in scalar processor. The number of LUT-FF pairs needed for the VecLIW execute stage is 3,850, where 3,114, 215, and 521 for unused flip-flops, unused LUT, and fully used LUT-FF pairs (see Table 1).

The fourth stage is Memory_Stage, which has one component called Data_memory. Not all instructions need to access memory stage. If instruction is a load, 128-bit data returns from memory and is placed in the MEM/WB pipeline register; if it is a store, then the data from the EX/MEM pipeline register (128-bit WrData) is written into memory. In either case the address used (AluOut) is the one computed during the prior cycle and stored in the EX/MEM pipeline register. In addition to 128×128-bit single-port RAM, Table 1 shows that 386 LUT-FF pairs are needed for
A VLIW Architecture for Executing Scalar/Vector Instructions on Instruction Level Parallelism

the memory access stage of the VecLIW, which has 289 LUT-FF pairs over the baseline scalar stage with the same size of data memory. The last stage is the Writeback_Stage, which writes the scalar/vector results into the VecLIW register file. It has multiplexers to select the results come from the memory.

![Fig.6. VecLIW pipeline](image)

**TABLE II: FPGA Statistics for VECLIW Processor**

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x128x128-bit single-port RAM</td>
<td>2x128-bit latch</td>
</tr>
<tr>
<td>4x32x32-bit multiplier</td>
<td>4x64-bit latch</td>
</tr>
<tr>
<td>6x32-bit adder</td>
<td>4x32-bit adder</td>
</tr>
<tr>
<td>4x32-bit subtractor</td>
<td>128-bit comparator equal</td>
</tr>
<tr>
<td>4x32-bit subtractor</td>
<td>32-bit comparator</td>
</tr>
<tr>
<td>4x32-bit subtractor</td>
<td>greatequal</td>
</tr>
<tr>
<td>3x64-bit adder</td>
<td>128-bit comparator greater</td>
</tr>
<tr>
<td>6x32-bit adder</td>
<td>32-bit comparator less</td>
</tr>
<tr>
<td>3x64-bit multiplexer</td>
<td>32-bit comparator</td>
</tr>
<tr>
<td>6x32-bit selector</td>
<td>32-bit comparator</td>
</tr>
<tr>
<td>7x1-bit register</td>
<td>lesseq</td>
</tr>
<tr>
<td>128x1-bit register</td>
<td>2x64-bit comparator</td>
</tr>
<tr>
<td>2-bit register</td>
<td>lesseq</td>
</tr>
<tr>
<td>20-bit register</td>
<td>8x32-bit 64-to-1</td>
</tr>
<tr>
<td>5x24-bit register</td>
<td>128x1-bit 32-bit</td>
</tr>
<tr>
<td>7x42-bit register</td>
<td>11-bit xor2</td>
</tr>
<tr>
<td>7x2-bit register</td>
<td>11-bit xor2</td>
</tr>
</tbody>
</table>

**Slice Logic Utilization**

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Registers: 3992</td>
</tr>
<tr>
<td>Number of Slice LUTs: 14826</td>
</tr>
<tr>
<td>Number used as Logic: 14570</td>
</tr>
<tr>
<td>Number used as Memory: 256</td>
</tr>
</tbody>
</table>

**Slice Logic Distribution**

| Number of LUT Flip Flop pairs used: 17425 | Number with an unused Flip Flop: 13433 |
| Number with an unused LUT: 2599 | Number of fully used LUT-FF pairs: 1393 |

**Timing**

| Maximum Frequency: 75.3235MHz |

system (128-bit Mem Out) or the results come from the ALUs (4x32-bit Alu Out). Depending on the opcodes, the register destination field is specified by RD or RT (RdRtDes). Only 156 LUT-FF pairs are needed for VecLIW write back stage (see Table 1), which has 124 LUT-FF pairs over the corresponding Write back stage of the baseline scalar processor. The complexity of the VecLIW write back stage is 488% (156/32) higher than the corresponding scalar write back stage because VecLIW writes back four elements per clock cycle instead of one. Table 2 summarizes the VecLIW complexity of VecLIW pipeline shown in Fig.6. The total number of LUT-FF pairs used is 17,425, where the number with an unused flip-flop is 13,433, the number with an unused LUT is 2,599, and the number of fully used LUT-FF pairs is 1,393. The complexity of four-issue VecLIW is 443% higher the single issue scalar processor.

**V. CONCLUSIONS AND FUTURE WORK**

This paper proposes new processor architecture called VecLIW for accelerating data-parallel applications. VecLIW executes multi-scalar and vector instructions on the same parallel execution data path. VecLIW has a modified five-stage pipeline for (1) fetching 128-bit VLIW instruction (four individual instructions), (2) decoding/reading operands of the four instructions packed in VLIW, (3) executing four operations on parallel execution units, (4) loading/storing 128-bit (4x32-bit scalar/vector) data from/to data memory, and (5) writing back 4x32-bit scalar/vector results. Moreover, this paper presents the FPGA implementation of our proposed VecLIW our implementation of VecLIW using VHDL targeting the Xilinx FPGA Virtex-5, XC5VLX110T-3FF1136 device. It requires 17,425 LUT-FF pairs, where the number with an unused flip-flop is 13,433, the number with an unused LUT is 2,599, and the number of fully used LUT-FF pairs is 1,393. The complexity of four-issue VecLIW is 443% higher the single issue scalar processor. In the future, the performance of our proposed VecLIW will be evaluated on scientific and multimedia kernels/applications.

**VI. REFERENCES**


