A VLIW- Architecture for executing Multi-Scalar/vector Instructions on unified datapath

A Durgaprasad
M.Tech Student Scholar
SSJ Engineering College

Prof. K Suresh Kumar
Associate Professor
SSJ Engineering College

Prof.S Jagadeesh
H.O.D ECE department
SSJ Engineering College

Abstract—**this paper proposes new processor architecture for 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 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. Which loads/stores 128- bit scalar/vector data from/to data cache. Four 32-bit results can be written back into VecLIW register file.**

Keywords—VLIW architecture; vector processing; data-level parallelism; 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 [1] 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 (*v* 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 [2, 3]. 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 [4].

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 superscalars because of further hardware simplifications. However, VLIW architectures require more compiler support. See [5] 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 advantages of VLIW is that the highly parallel implementation is much simpler and cheaper to build the equivalently concurrent RISC or CISC chips. See [6] 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 [7, 8]. Thus, vector processors remain the most effective way to exploit data-parallel applications [9, 10]. Therefore, many vector architectures have been proposed in the literature to accelerate data-parallel applications [11-17].

Commercially, the Cell BE architecture [18] 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 datapaths. 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 [19]. 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 [20] 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. Wada et al. [21] 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 datapath, 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. Applying Amdahl'Law on VecLIW is presents in Section II. The VecLIW architecture is depicted in detail 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. Applying Amdahl's Law on VecLIW

Vector ISA reduces the semantic gap between programs and hardware [9]. Programmer can express parallelism to hardware using vector instructions. Otherwise, vector compilers did ultimately get good at synthesizing vector operations even when they were not explicitly expressed. Thus, the generated code says something higher-level, and then the processor manipulates the simple operations on its own (see [5] for more detail).

Programs have several different portions of their runtime that can be accelerated to differing degrees. On VecLIW, programs can be divided into scalar (unvectorizable) and vectorizable parts. Vectorizable parts can be accelerated to differing degrees on parallel execution units using vector ISA. On the other hand, scalar parts may be accelerated on multiple execution units using VLIW. Thus, on VecLIW, not only vectorizable parts are sped up but also scalar parts. It is well known that Amdahl's Law [22] governs the speedup of using parallel processing on a problem versus using sequential processing. It states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used. The main advantage of the VecLIW is the increase of the faster mode fraction by parallel processing VLIW/vector instructions on multiple execution units. Consider an application consists of a scalar code, which cannot be vectorizable, and vectorizable code, which spends V fraction of time on the baseline scalar processor. This vectorizable code can be decomposed further into $V_1, V_2, \ldots, V_m$ fractions that can be sped up by executing vector instructions on multiple execution units by factors of $P_1, P_2, \ldots, P_m$, respectively, where $V = V_1 + V_2 + \ldots + V_m$. Moreover, the unvectorizable, scalar code that can be sped up using VLIW spends S fraction of time on the baseline scalar processor, where scalar code that cannot be sped up using VLIW or vector instructions spends $(1 - S - V)$ fraction of time. S can be decomposed further into $S_1, S_2, \ldots, S_n$ fractions that can be sped up by factors of $q_1, q_2, \ldots, q_n$, respectively, where $S = S_1 + S_2 + \ldots + S_n$. Thus, Amdahl's Law predicts an overall speedup equals

$$\frac{1}{(1-S-V)+S_1/q_1+S_2/q_2+\ldots+S_n/q_n+V_1/P_1+V_2/P_2+\ldots+V_m/P_m}$$

Using Amdahl's Law, the scalar parts $(1 - S - V)$ that are not sped up using VLIW or vector instructions will dominate the running time as increasing the parallel execution units. Thus, VecLIW minimizes the unparallel fraction of time $(1 - S - V)$ by the execute of VLIW instructions in addition to vector instructions on parallel execution units. Obviously, the use of vector ISA alone results in unparallel fraction of time $(1 - V)$, which is greater than the corresponding VecLIW factor $(1 - S - V)$.

Generally, vector hardware works better than superscalar/VLIW processors on very regular code containing long vectors, whereas superscalar/VLIW processors tend to work better than vector processors when the structure of the code is somewhat more irregular, and when vectors are short. In addition, superscalar/VLIW processors offer the advantage that the same hardware can be used on the parts of code that are not vectorizable, whereas vector hardware is dedicated to vector use only. However, converting a vector instruction into scalar instructions degrades the performance because scalar ISA cannot convoy parallelism to processor. The use of vector ISA leads to multiple homogeneous, independent operations are packaged into a single short vector instruction, resulting compact, expressive, and scalable code. The vector code is compact because a single vector instruction can describe *v* scalar operations and address up to 3*v* element operands. This dramatically reduces instruction bandwidth requirements. Moreover, many of the looping constructs required to iterate a scalar processor over the *v* operations are implicit in the vector instructions, reducing instruction bandwidth requirements even further. The code is expressive because software can pass on much valuable information to hardware about this group of *v* operations. See [7, 11] for more details. Therefore, VecLIW combines the advantages of VLIW that exploits ILP and vector processing that exploits DLP.

III. The Architecture of VecLIW Processor

VecLIW is a load-store architecture with simple hardware, fixed-length instruction encoding, and simple code generation model. It supports few addressing modes to specify operands: *register*, *immediate*, and *displacement* addressing modes. In the displacement addressing mode, a constant offset is signed-extended and added to a scalar register to form the memory address for loading/storing 128-bit data. VecLIW has a simple and easy-to -pipeline ISA, which supports the general categories of operations (data transfer, arithmetic, logical, and control) . According to the upper two bits (*SV* field) of the opcode, there are four types of the VecLIW instructions:

(1) *SV* field = $(00)_2$ for scalar-scalar instructions (*.ss* type),
(2) *SV* field = $(01)_2$ for scalar-vector instructions (*.sv* type),
(3) *SV* field = $(10)_2$ for vector-scalar instructions (*.vs* type),
(4) *SV* field = $(11)_2$ for vector-vector instructions (*.vv* type).

For example sub.ss, sub.sv, sub.vs, and sub.vv perform scalar subtraction, scalar-vector subtraction, vector-scalar subtraction, and vector-vector subtraction, respectively, on *v*-element operands, where $1 \leq v \leq$ MVL (maximum vector length).

VecLIW uses *fixed length* for encoding scalar/vector instructions. All VecLIW instructions are 4×32-bit

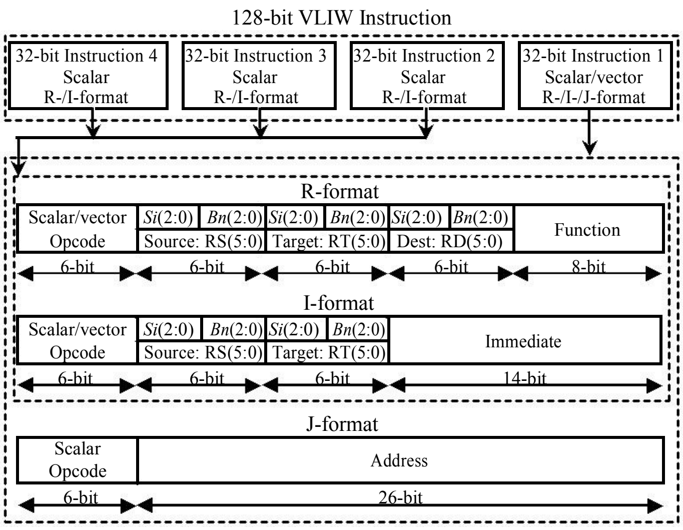

128-bit VLIW Instruction
32-bit Instruction 4 Scalar R-/I-format
32-bit Instruction 3 Scalar R-/I-format
32-bit Instruction 2 Scalar R-/I-format
32-bit Instruction 1 Scalar/vector R-/I-/J-format
R-format
Scalar/vector Opcode
Si(2:0) Bn(2:0) Source: RS(5:0)
Si(2:0) Bn(2:0) Target: RT(5:0)
Si(2:0) Bn(2:0) Dest: RD(5:0)
Function
6-bit 6-bit 6-bit 6-bit 8-bit
I-format
Scalar/vector Opcode
Si(2:0) Bn(2:0) Source: RS(5:0)
Si(2:0) Bn(2:0) Target: RT(5:0)
Immediate
6-bit 6-bit 6-bit 14-bit
J-format
Scalar Opcode
Address
6-bit 26-bit

Fig. 1. VecLIW ISA formats.

(VLIW [127:0]), which simplifies instruction decoding. Figure 1 shows the VecLIW instruction formats (R-format, I-format, and J-format), which are very close to MIPS [23, 24]. The first 32-bit instruction (VLIW [31:0]) can be scalar/vector/control instruction. However, the remaining 32-bit instructions (VLIW [63:32], VLIW [95:64], and VLIW [127:96]) must be scalars. This simplifies the implementation of VecLIW and does not effect on the performance drastically since a vector instruction can code up to eight vector operations instead of four scalar operations stored in VLIW. However, control instructions encode only one operation. In this paper, a subset of the VecLIW ISA is used to build a simple and easy to explain version of 32-bit VecLIW architecture.

Figure 2 shows the block diagram of our proposed VecLIW processor, which has common datapath for executing VLIW/vector instructions. Instruction cache stores 128-bit VLIW instructions of an application. Data cache loads/stores scalar/vector data needed for processing scalar/vector

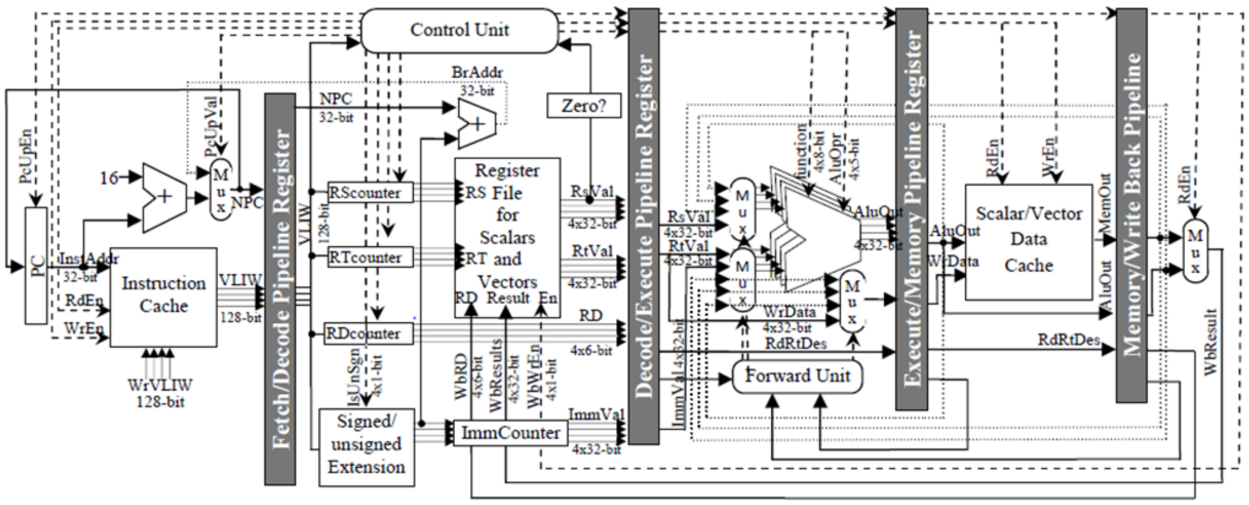


Fig. 2. VecLIW data path for executing multi-scalar/vector instructions.

instructions. A single register file is used for both multi-scalar/vector elements. The control unit feeds the parallel execution units by the required operands (scalar/vector elements) and can produce up to four results each clock cycle. Scalar/vector loads/stores take place from/to the data cache of VecLIW in a rate of 128-bit (four elements: 4×32 -bit) per clock cycle. Finally, the writeback stage writes into the VecLIW register file up to 4×32-bit results per clock cycle coming from the memory system or from the execution units. The use of unified hardware for processing multi-scalar/vector data makes efficient exploitation of resources even though the percentage of DLP is low. Comparing with the baseline scalar processor (five stage pipeline), the complexity of decode, execute, and writeback stages of the VecLIW are about four times. However, fetch and memory access stages are approximately the same as in the well-known five-stage scalar pipeline. The design of the five-stage MIPS pipeline datapath used in the baseline scalar processor is explained in detail in [25].

In more details, VecLIW has a modified five-stage pipeline for executing multi-scalar/vector instructions by: (1) fetching 128-bit VLIW instruction, (2) decoding/reading operands of four individual instructions, (3) executing four scalar/vector operations, (4) accessing memory to load/store 128-bit data, and (5) writing back four results. The VLIW instruction pointed by PC is read from the instruction cache of the fetch stage and stored in the instruction fetch/decode (IF/ID) pipeline register. The control unit in the decode stage reads the fetched VLIW instruction from IF/ID pipeline register to generate the proper control signals needed for processing multiple scalar or vector data. The register file of the VecLIW has eight banks (B0 to B7), eight-element each (B0.0 to B0.7, B1.0 to B1.7, …, and B7.0 to B7.7). Scalar/vector data are accessed from VecLIW register file using 3-bit bank number (*Bn*) concatenated with 3-bit start index (*Si*). 2×4×32-bit operands can be read and 4×32-bit can be written to the VecLIW register file each clock cycle. Thus, the control unit reads the *Si.Bn* fields of RS (register source), RT (register target), and RD (register destination) of each individual instruction in the fetched VLIW as well as VLR (vector length register) to generate the sequence of control signals needed for reading/writing multi-scalar/vector data from/to VecLIW register file. The VecLIW register file can be seen as 64×32-bit scalar registers or 8×8×32-bit vector registers (eight 8 -element vector registers). Moreover, the start index (*Si*) could be non-zero ($0 \le Si \le 7$) and the vector data are stored in VecLIW banks in round-robin fashion.

Four individual instructions packed in VLIW instruction are decoded and their operands are read from the unified register file ($RsVal_1/RtVal_1$, $RsVal_2/RtVal_2$, $RsVal_3/RtVal_3$, and $RsVal_4/RtVal_4$) according to four pairs of RS/RT fields (RS_1/RT_1, RS_2/RT_2, RS_3/RT_3, and RS_4/RT_4), respectively. Moreover, the 14-bit immediate values (VLIW[13:0], VLIW[45:32], VLIW[77:64], and VLIW[109:96]) of the I-format are signed-/unsigned-extended into 4×32-bit immediate values ($ImmVal_1$, $ImmVal_2$, $ImmVal_3$, and $ImmVal_4$) . These RsVal, RtVal, and ImmVal values are stored in the ID/EX pipeline register for processing in the execute stage. In addition to decoding the individual instructions of VLIW and accessing VecLIW register file, RS.Si, RT.Si, RD.Si, and ImmVal values Are loaded into counters called RScounter, RTcounter, RDcounter, and ImmCounter, respectively. For decoding vector instructions, the control unit stalls the fetch stage and iterates the process of reading vector elements, incrementing RScounter, RTcounter, and RDcounter by four and the immediate value (ImmCounter) by 16, and calculating the destination registers. Depending on the vector length (v), the control unit issues operands to the execution units through ID/EX pipeline register number of times equals $v/4$, where $1 \le v \le MVL$, where MVL equals eight in the first implementation of VecLIW. After issuing each operation in the vector instruction, it is removed from IF/ID pipeline register and new VLIW instruction is fetched from instruction cache.

The execute units of VecLIW operate on the operands prepared in the decode stage and perform operations specified by the control unit, which depends on $opcode_1/function_1$, $opcode_2/function_2$, $opcode_3/function_3$, and $opcode_4/function_4$ fields of the individual instructions in VLIW. For load/store instructions, the first execute unit adds $RsVal_1$ and $ImmVal_1$ to form the effective address. For register-register instructions, the execute units perform the operations specified by the control unit on the operands fed from the register file ($RsVal_1/RtVal_1$, $RsVal_2/RtVal_2$, $RsVal_3/RtVal_3$, and $RsVal_4/RtVal_4$) through ID/EX pipeline register. For register-immediate instructions, the execute units perform the operations on the source values ($RsVal_1$, $RsVal_2$, $RsVal_3$, and $RsVal_4$) and the extended immediate values ($ImmVal_1$, $ImmVal_2$, $ImmVal_3$, and $ImmVal_4$). In all cases, the results of the execute units is placed in the EX/MEM pipeline register.

The VecLIW registers can be loaded/stored individually using load/store instructions. Displacement addressing mode is used for calculating the effective address by adding the singed-extended immediate value ($ImmVal_1$) to RS register ($RsVal_1$) of the first individual instruction in VLIW. In addition, the $ImmVal_1$ register is incremented by 16 to prepare the address of the next 4×32-bit element of the vector data. In the first implementation of our proposed VecLIW processor, four elements (128-bit) can be loaded/stored per clock cycle.

Finally, the writeback stage of VecLIW stores the 4×32-bit results come from the memory system or from the execution units in the VecLIW register file. Depending on the effective *opcode* of each individual instruction in VLIW, the register destination field is specified by either RT or RD. The control signals 4×Wr2Reg are used for enabling the writing 4×32-bit results into the VecLIW register file.

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. See [26-28] for more details.

Figure 3 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 PcUpdEn 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 4×14-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, see Figure 1. 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/unimmediate 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 [1]. 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 *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 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

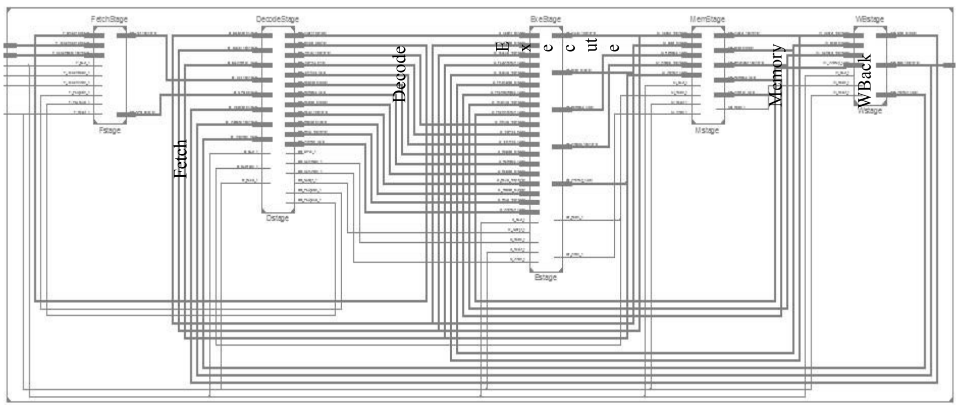


Fig. 3. Top-level RTL schematic diagram of the proposed VecLIW processor.

TABLE I. FPGA STATISTICS FOR VECLIW STAGES

Statistics		Fetch Stage	Decode Stage	Execute Stage	Memory Stage	Writeback Stage
Slice Logic Utilization	Number of Slice Registers:	192	2421	736	128	0
	Number of Slice LUTs:	299	10718	3635	386	156
	Number used as Logic:	43	10718	3635	130	156
	Number used as Memory:	256	0	0	256	0
Slice Logic Distribution	Number of LUT Flip Flop pairs used:	327	10720	3850	386	156
	Number with an unused Flip Flop:	135	8299	3114	258	156
	Number with an unused LUT:	28	2	215	0	0
	Number of fully used LUT-FF pairs:	164	2419	521	128	0
Timing	Maximum Frequency(MHz):	333.989	74.866	386.757	No path found	No path found

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 has two modules: ALUs and Forward_unit. The ALUs operate on four pairs of operands prepared in the decode stage. ALUs 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 ($RsVal_1$ and $ImmVal_1$) to form the effective address, however, the remaining ALUs are ideal. For register-register operations, the ALU_i performs the operation specified by the 8-bit $function_i$ control signals, on the values 32-bit $RsVal_i$ and 32-bit $RtVal_i$, where $1 \leq i \leq 4$. For register-immediate operations, the ALU_i performs the operation specified by the 5-bit $AluOpr_i$ control signals on the values in 32-bit $RsVal_i$ and 32-bit $ImmVal_i$. 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 (WrData). 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 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

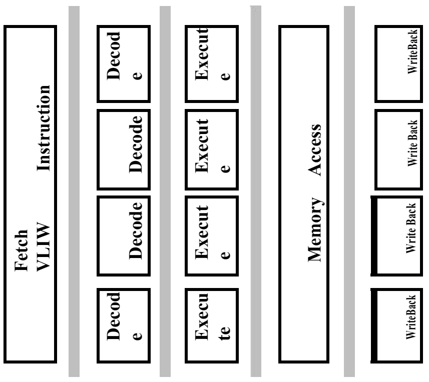


Fig. 4. VecLIW pipeline.

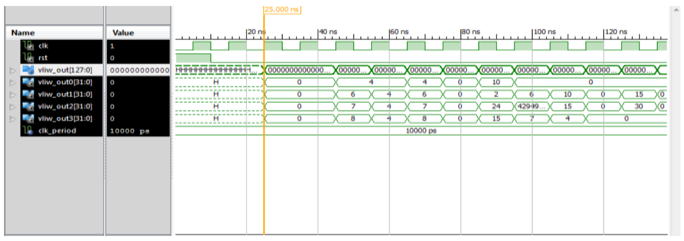
TABLE II. FPGA STATISTICS FOR VECLIW PROCESSOR

Statistics		
HDL Synthesis Report	2x128x128-bit single-port RAM	
	4x32x32-bit multiplier	2x128-bit latch
	6x32-bit adder	4x64-bit latch
	4x32-bit subtractor	128-bit comparator equal
	4-bit adder	32-bit comparator
	4-bit subtractor	greatequal
	33x6-bit adder	32-bit comparator greater
	3x64-bit adder	4x32-bit comparator less
	3x64-bit up accumulator	32-bit comparator
	7x1-bit register	lessequal
	7x128-bit register	2x4-bit comparator
	2-bit register	lessequal
	20-bit register	76x6-bit comparator equal
	5x24-bit register	8x32-bit 64-to-1
	74x32-bit register	multiplexer
	9x4-bit register	128x1-bit xor2
	72x1-bit latch	11-bit xor2
Slice Logic Utilization	Number of Slice Registers: 3992	
	Number of Slice LUTs: 14826	
	Number used as Logic: 14570	
	Number used as Memory: 256	
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: 75.825MHz	

system (128-bit MemOut) or the results come from the ALUs (4×32-bit AluOut). Depending on the *opcodes*, the register destination field is specified by RD or RT (RdRtDes). Only 156 LUT -FF pairs are needed for VecLIW writeback stage (see Table 1), which has 124 LUT-FF pairs over the corresponding writeback stage of the baseline scalar processor. The complexity of the VecLIW writeback stage is 488% (156/32) higher than the corresponding scalar writeback stage because VecLIW writes back four elements per clock cycle instead of one.

Table 2 summarizes the VecLIW complexity of VecLIW pipeline shown in Figure 4. The total number of LUT-FF pairs used is 17,425, where the number with an unused flip-flops 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.

SIMULATION RESULTS:



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 datapath. 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 (4×32-bit scalar/vector) data from/to data memory, and (5) writing back 4×32 -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 -flops 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.

References

[1] J. Hennessay and D. Patterson, Computer Architecture A Quantitative Approach, 5th ed, Morgan-Kaufmann, September 2011.
[2] J. Mike, Superscalar Microprocessor Design, Prentice Hall (Prentice Hall Series in Innovative Technology), 1991.
[3] J. Smith and G. Sohi, “The microarchitecture of superscalar processors,” Proceedings of the IEEE, vol. 83, no. 12, pp. 1609-1624, December 1995.
[4] J. Fisher, “VLIW architectures and the ELI-512,” Proc. 10th International Symposium on Computer Architecture, Stockholm, Sweden, pp. 140-150, June 1983.
[5] J. Fisher, P. Faraboschi, and C. Young, Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools, Morgan Kaufmann, 2004.
[6] Philips, Inc., An Introduction to Very-Long Instruction Word (VLIW) Computer Architecture, Philips Semiconductors, 1997.
[7] R. Espasa, M. Valero, and J. Smith, “Vector architectures: past, present and future,” Proc. 2nd International Conference on Supercomputing, Melbourne, Australia, pp. 425-432, July 1998.
[8] F. Quintana, R. Espasa, and M. Valero, “An ISA comparison between superscalar and vector processors,” in VECPAR, vol. 1573, Springer-Verlag London, pp. 548-560, 1998.
[9] J. Smith, “The best way to achieve vector-like performance?,” Keynote Speech, in 21st International Symposium on Computer Architecture, Chicago, IL, April 1994.
[10] C. Kozyrakis and D. Patterson, “Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks,” Proc. 35th International Symposium on Microarchitecture, Istanbul, Turkey, pp. 283-293, November 2002.
[11] K. Asanovic, Vector Microprocessors, Ph.D. Thesis, Computer Science Division, University of California at Berkeley, 1998.
[12] S. Kaxiras, “Distributed vector architectures,” Journal of Systems Architecture, Elsevier Science B.V., vol. 46, no. 11, pp. 973-990, 2000.
[13] C. Kozyrakis, Scalable Vector Media-processors for Embedded Systems, Ph.D. Thesis, Computer Science Division, University of California at Berkeley, 2002.
[14] R. Krashinsky, Vector-Thread Architecture and Implementation, Ph.D. Thesis, Massachusetts Institute of Technology, 2007.
[15] J. Gebis, Low-complexity Vector Microprocessor Extensions, Ph.D. thesis, University of California at Berkeley, 2008.
[16] C. Batten, Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators, Ph.D. Thesis, Massachusetts Institute of Technology, 2010.
[17] P. Yiannacouras, J. Steffan, and J. Rose, “Portable, flexible, and scalable soft vector processors,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 20, no. 8, pp. 1429-1442, August 2012.
[18] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, “Synergistic processing in Cell's multicore architecture,” Journal IEEE Micro, vol. 26, no. 2, pp. 10-24, March 2006.
[19] T. Zeiser, G. Hager, and G. Wellein, “Performance results from the NEC SX-9,” Proc. IEEE International Symposium on Parallel & Distributed Processing, IPDPS-2009, pp. 1-8, 2009.
[20] E. Salami and M. Valero “A vector-µSIMD-VLIW architecture for multimedia applications,” Proc. IEEE International Conference on Parallel Processing, ICPP-2005, pp. 69-77, 2005.
[21] T. Wada, S. Ishiwata, K. Kimura, K. Nakanishi, M. Sumiyoshi, T. Miyamori, and M. Nakagawa, “A VLIW vector media coprocessor with cascaded SIMD ALUs,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 17, no. 9, pp. 1285-1296, 2009.
[22] G. Amdahl, “Validity of the single-processor approach to achieving large scale computing capabilities,” Proc. AFIPS 1967 Spring Joint Computer Conference, Atlantic City, New Jersey, AFIPS Press, vol. 30, pp. 483-485, April 1967.
[23] G. Kane, MIPS RISC Architecture (R2000/R3000), Prentice Hall, 1989.
[24] MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set, MIPS, Revision 0.95, 2011. Available at: http://www.cs.cornell.edu/courses/cs3410/2011sp/MIPS Vol2.pdf.
[25] D. Patterson and J. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 4th ed, Morgan Kaufman, December 2011.
[26] A. Cosoroaba and F. Rivoallon, “Achieving higher system performance with the Virtex-5 family of FPGAs,” White Paper: Virtex-5 Family of FPGAs, Xilinx WP245 (v1.1.1), July 2006.
[27] A. Percey, “Advantages of the Virtex-5 FPGA 6-Input LUT architecture,” White Paper: Virtex-5 FPGAs, Xilinx WP284 (v1.0), December 2007.
[28] Virtex-5 FPGA User Guide, UG190 (v5.4), March 2012. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.