ijecs open access www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 4 Issue 10 Oct 2015, Page No. 14589-14593

Advanced Vecliw Architecture For Executing Multi-Scalar/Vector Instructions On Unified Datapath

B.Prasanna, Sharad Kulkarni M.S(E&C),FIETE,MISTE
Department of Electronics and communication Engineering
Audisankara Institute of Technology,Gudur

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 Verilog HDL.**

Keywords: VecLIW architecture; vector processing; data-level parallelism; unified datapath; FPGA/Verilog HDL 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 super scalars because of further hardware simplifications. However, VLIW architectures require more compiler support.

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 are 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.

DOI: 10.18535/ijecs/v4i10.9

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 wide data paths. 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/Verilog HDL 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.

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 $V1$, $V2$, …, Vm fractions that can be sped up by executing vector instructions on multiple execution units by factors of $P1$, $P2$, …, Pm, respectively, where $V = V1 + V2 + \ldots + Vm$. 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 $S1$, $S2$, …, Sn fractions that can be sped up by factors of $q1$, $q2$, …, qn, respectively,

where $S = S1 + S2 + \ldots + Sn$. Thus, Amdahl's Law predicts an overall speedup equals

$$\frac{\mathbf{1}}{(\mathbf{1}-\boldsymbol{S}-\boldsymbol{V})+\frac{\boldsymbol{S1}}{\boldsymbol{q1}}+\frac{\boldsymbol{s2}}{\boldsymbol{q2}}+\cdots+\frac{\boldsymbol{sn}}{\boldsymbol{qn}}+\frac{\boldsymbol{V1}}{\boldsymbol{P1}}+\frac{\boldsymbol{V2}}{\boldsymbol{P2}}+..+\frac{\boldsymbol{Vm}}{\boldsymbol{Pm}}}$$

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 $3v$ 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

DOI: 10.18535/ijecs/v4i10.9

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. 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 (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].

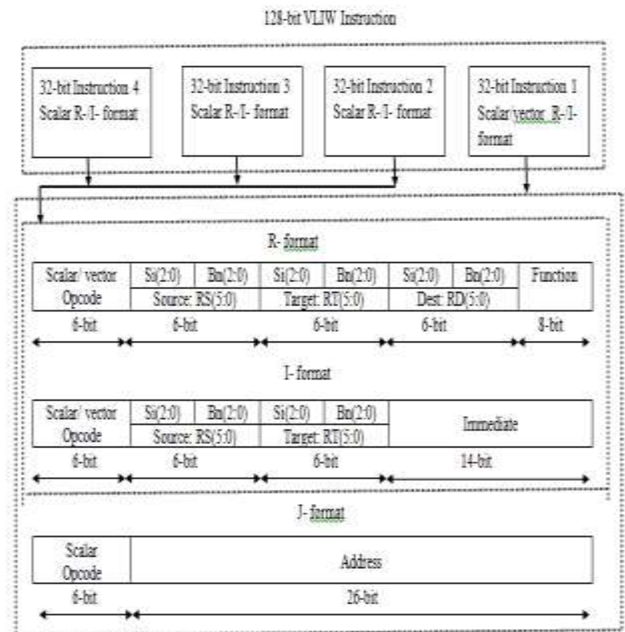

Fig. 1. VecLIW ISA formats

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.

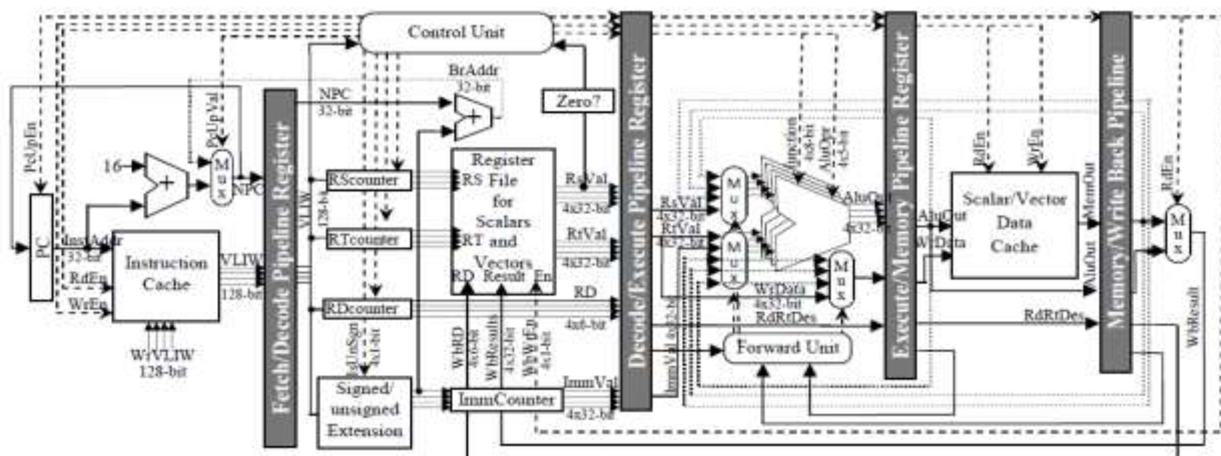

Fig.2. VecLIW datapath for executing multi-scalar/vector instructions.

Instruction cache stores 128-bit VLIW instructions of an application. Data cache loads/stores scalar/vector data needed for processing scalar/vector 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 write back 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

Write back 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 nonzero ($0 \leq Si \leq 7$) 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 (RsVal1/RtVal1, RsVal2/RtVal2, RsVal3/RtVal3, and RsVal4/RtVal4) according to four pairs of RS/RT fields (RS1/RT1, RS2/RT2, RS3/RT3, and RS4/RT4), respectively.

Moreover, the 14-bit immediate values (VLIW[13:0], VLIW[45:32], VLIW[77:64], and VLIW[109:96]) of the Iformat are signed-/unsigned-extended into 4×32-bit immediate values (ImmVal1, ImmVal2, ImmVal3, and ImmVal4). 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 (ImmCounters) 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 $1 \leq v \leq MVL$, 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 RsVal1 and ImmVal1 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 (RsVal1/RtVal1, RsVal2/RtVal2, RsVal3/RtVal3, and RsVal4/RtVal4) through ID/EX pipeline register. For register-immediate instructions, the execute units perform the operations on the source values (RsVal1, RsVal2, RsVal3, and RsVal4) and the extended immediate values (ImmVal1, ImmVal2, ImmVal3, and ImmVal4).

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 (ImmVal1) to RS register (RsVal1) of the first individual instruction in VLIW. In addition, the ImmVal1 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. SIMULATION RESULTS

Figure 3 shows the top-level RTL schematic diagram of our proposed VecLIW. It is generated from synthesizing the Verilog HDL code of the VecLIW processor on Xilinx ISE tool.

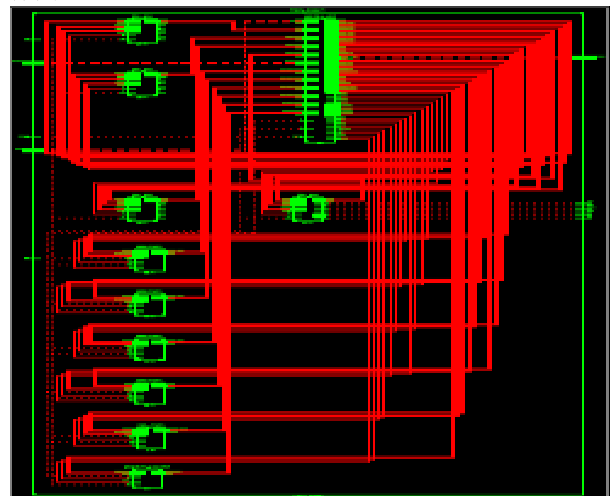

Fig. 3.Top-level RTL schematic diagram of the proposed VecLIW processor

DOI: 10.18535/ijecs/v4i10.9

The simulation results of the proposed VecLIW processor is as shown in fig4.

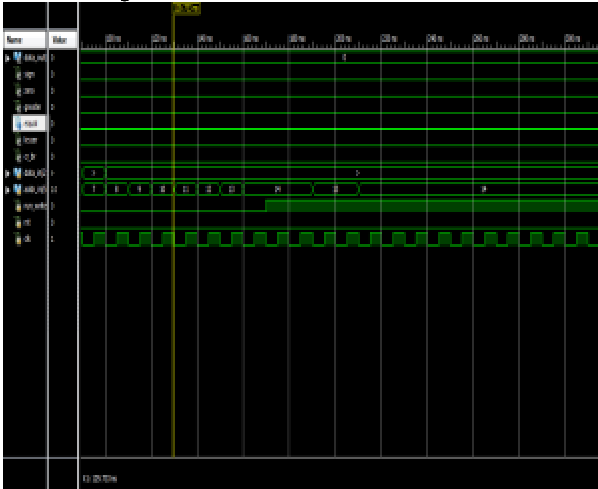

Fig.4. Simulation results of proposed design

V.CONCLUSION

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. And this proposed design is simulated by using the Verilog HDL code.

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 micro architecture 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- Verilog 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.

Author's Profile:

Prasanna.Bandila received his B.TECH degree in Electronics and Communication Engineering from Jagan's College of Engineering & Technology,Choutpalem, SPSR Nellore(Dist), affiliated to JNTU Anantapur. He is currently pursuing M.Tech VLSI in Audisankara Institute of Technology, Gudur,SPSR Nellore (Dist), affiliated to JNTU Anantapur.

Sharad Kulkarni He received his M.S(E&C) degree in Birla Institute of Technology & Science ,Pilani,Jaipur. He is presently working as Professor and HOD in the department of ECE, Audisankara Institute of Technology, Gudur, SPSR Nellore (Dist), affiliated to JNTU Anantapur.