VLIW-DLX Simulator for Educational Purposes

Miloš Bečvář and Stanislav Kahánek

Department of Computer Science and Engineering,
Faculty of Electrical Engineering,
Czech Technical University in Prague,
Karlovo nám. 13, Prague 2, Czech Republic
E-mail: becvarm@fel.cvut.cz

Abstract

VLIW-DLX is graphical simulator of simple VLIW processor. It is targeted to be used in undergraduate computer architecture courses. VLIW-DLX uses similar GUI to well-known WinDLX simulator and its ISA uses scalar DLX instructions as the building blocks. Simulator is implemented in Java and allows future modifications of the architecture including instruction set expansion. Paper discusses choices made in developing VLIW-DLX and its intended educational use.

Categories and Subject Descriptors

C.1.1 Computer Systems Organization: PROCESSOR ARCHITECTURES Single Data Stream Architectures *VLIW architectures*

General Terms

Design

Keywords

Computer architecture, VLIW, Simulation, Education.

WCAE '07, June 9, 2007 San Diego, CA

1. Introduction

Very Long Instruction Word (VLIW) processors [1] have never reached the same level of attention in computer architecture courses as their superscalar counterparts. An interest in VLIWs increased and fallen in waves similar to a success or failure of their vendors [2], [3]. In the middle of 90's the VLIWs were seen as a rescue from complexity of out-of-order speculative superscalar processors in workstations and servers. However, the physical realization of evolved VLIWs (called EPIC) [4] came late and its performance mainly on integer code was rather disappointing [5]. However, excluding the mainstream workstation and server market, the success of VLIWs is evident in high-performance computing area and in a subset of embedded computers – typically DSPs (e.g. TI TMS320C6xx) or media processors (Philips Trimedia [6]). Recently published excellent textbook [7] authored by the classics of the field covers new embedded system oriented usage of VLIW architectures.

Our intention to build a new VLIW simulator was motivated by the fact that typical EECS graduate in Europe will most likely be a part of a team designing systems for embedded applications. Coverage of VLIWs, DSPs and simple processor cores is more important than detailed description of out-of-order superscalar processor implementation. Although there are several advanced VLIW compilers and simulators available namely Trimaran [8] [9], and VEX [10], we found that none of them is suitable for an elementary introduction to VLIWs. We have developed the VLIW-DLX simulator for use in undergraduate computer architecture courses. It is simpler than any known available simulator and shows the relationship between VLIWs and conventional scalar RISC pipelines. Based on our long term experience with WinDLX [11] we wanted a similar tool for VLIW introduction.

In the rest of this paper we present first results of our work. Section 2 describes first version of the VLIW-DLX processor model. It discusses several design decisions made during the development. Section 3 contains short description of simulator implementation and its possible modifications and extensions. Intended use of the simulator in computer architecture courses is presented in section 4 and our future plans and conclusions in section 5.

2.VLIW-DLX Architecture Overview

Main goal of VLIW-DLX simulator is the introduction of VLIW philosophy to students familiar with conventional ISAs. Following sections describe the further design decision made in the process. Our decisions were influenced by the two „simplicity“ requirements. First requirement was the simplicity for students which allow them to easily understand the architecture. Due to short time available we have stressed the principle that everything which is not essential to illustrate VLIW principles should be similar to DLX processor [12]. Second requirement was the simplicity of (possible) hardware implementation. This was the one of the foundation principles of first VLIWs [1] which was the key difference to superscalar processors with complex scheduling hardware. Yet this simplicity leads to a lot of inefficiencies which motivates adding of VLIW-specific complex logic to overcome them (namely predication, load/store speculation, rotating register files etc). It is therefore not clear whether current high-performance VLIW designs are simpler than their superscalar counterparts. Since our intention is to show the links between VLIWs and conventional scalar pipelines, we decided for not implementing these enhancements.

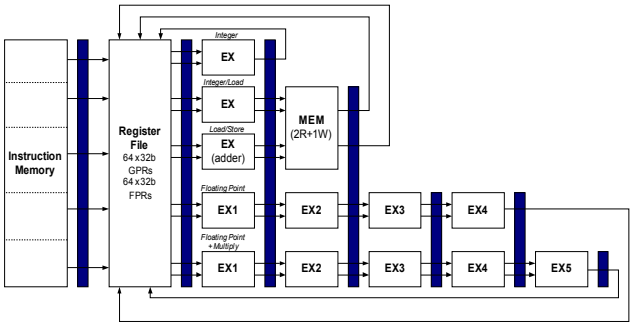

Figure 1. VLIW-DLX pipeline

2.1. Main VLIW-DLX features

First design decision was that VLIW-DLX instruction will consist of a collection of independent scalar DLX instructions which now become „operations“ within a single VLIW instruction. Similarly the VLIW pipeline should be a simple collection of parallel DLX pipelines (see Figure 1). We believe that this illustrates the main VLIW principle of „exposing the ILP on the level of architecture“ [7] and yet it saves the students from learning a whole new ISA. Another advantage of this approach is that students can start with program for conventional DLX and modify it for VLIW-DLX.

2.2. Number and type of parallel pipelines

Selection of good execution unit mixture is essential for efficient use of the parallel pipeline. It is not difficult to build a simulator consisting of a large number of execution units of many types. However, building such processor is not trivial and sometimes even not realistic. In particular - building of many parallel Load/Store units is rather difficult and lack of them will limit usability of other type of functional units.

We have also expected that it would not be easy for our students to use many instances of execution units efficiently when programming the processor in assembly language.

Table 1. VLIW-DLX Operation Groups

Operation group	***DLX instructions = VLIW DLX operations***
Control	beqz, bnez, j, jal, jr, jalr, bfpt, bfpf
Float	addf, subf, cvtf2i, cvti2f, eqf, nef, ltf, gtf, lef, gef
Multiply	multf, mult, multu
ALU	add, addu, addi, addui, sub, subu, subi, subui, and, andi, movf, lhi, or, ori, xor, xori, sll, slli, srl, srli, sra, srai,seq, sne, slt, sgt, sle, sge, seqi, snei, slti, sgti, slei, sgei, sequ, sneu,sltu, sgtu, sleu, sgeu, sequi, sneui, sltui, sgtui, sleui, sgeui
Load	lb, lbu, lh, lhu, lw, lf, movi2fp
Store	sb, sh, sw, sf, movfp2i

From all these reasons we have decided for the architecture consisting of 5 parallel pipelines described in Tables 1 and 2. First pipeline executes a single cycle integer instructions and branches. Second pipeline executes also the integer instructions and load instructions. Third pipeline is the load/store pipeline. Remaining two pipelines execute floating point instructions and complex integer mult instructions.

For the sake of simplicity we have decided not to implement division instruction (which would complicate the rules for latencies and write back buses) and double

precision floating point instructions (which require reading of 32-bit register pairs). We believe that using the single precision operations is sufficient for illustration purposes.

Table 2. VLIW-DLX Pipeline slot definition

VLIW pipeline	*Operation group allowed*	*Latency*
1	Control, ALU	2
2	ALU, Load	3
3	Load / Store	3
4	Float	4
5	Float, Multiply	5

2.3. Organization of register files

Conventional DLX defines 32 integer and 32 floating point registers. This is sufficient for conventional scalar processor. However, for efficient use of parallel pipelines, the number of registers must be increased. In the current implementation we have decided to double the number of available integer and floating point registers which should be sufficient for 5 parallel pipelines and assembly programming.

VLIW architectures sometimes use specialized types of registers - branch registers and predication registers [7],[8] which allow optimized implementation of control flow instructions. For the simplicity reasons we have decided to stick with the DLX concept of using general purpose registers for storing the branch addresses and condition results.

For the same reason we have decided not to implement rotating register files [4],[8], (used to simplify the software pipelining of loops). We believe that students will more appreciate the availability of these features if they try to implement these algorithms without this support.

2.4. Pipeline latencies and bypassing network

One of the differences of VLIW architectural style is the ability to expose pipeline latencies to software optimization [7]. Non-unit latency model is more common to VLIW architectures than unit-latency (atomic instruction) model in conventional ISAs. Related issue is selection between LEQ and EQ latency models [7]. In EQ model the operation executes exactly the latency which is specified, in LEQ latency model, the operation can take shorter time than latency specified by the architecture. Finally the question is whether to support any precise exception model or not and associated question of the availability of result bypassing network.

VLIW-DLX implements non-unit EQ latency model without any support of precise exceptions. VLIW-DLX does not implement bypassing network for simplicity and contrast to scalar DLX (see Figure 1). Every operation in a given pipeline slot take the latency corresponding to number of stages after register operands are fetched (ID stage) till the result is written into register file (WB stage). Each pipeline slot has associated the single register file write port and there are no structural conflicts on WB ports possible. If the register is read by subsequent VLIW instruction in ID stage while it is written by the instruction in WB stage, the register file provides the new value to subsequent instruction. This simple model means that all register values are communicated over the register file. The programmer must schedule instruction in a way that dependent instructions will read the correct data from registers. This is very different from the WinDLX model with hazard detection and forwarding. One particular feature of our pipeline is the fact that latency of integer operation in a slot 1 is 2 cycles, while it is 3 cycles in Load/Integer slot 2 of the pipeline.

2.5. Execution unit clustering

Implementation of large multiported register file and result bypassing network is the major hurdle in scaling the number of functional units in ILP processors [7]. As a compromise the functional units and registers are divided into clusters which are easily implementable. Software is typically responsible for inter-cluster communication and efficient mapping of parallel operations into execution clusters. We think that for the first introduction to VLIWs the clustering will impose additional constraint for students. Therefore we have decided not to implement clustering in VLIW-DLX model. We think that 10 x read-port / 5 x write-port register file of 128x32b registers is implementable even in mature silicon technologies without being the performance limiting factor of the architecture.

We have also considered optimized implementation of separate 6 x read-port/ 3 x write port integer register file and 4 x read-port / 2 x write-port FP register file. However, it introduces additional structural hazards and scheduling constrains and we decided not to include this feature in first version of VLIW-DLX.

2.6. Implementation of control instructions

Control instructions in VLIW-DLX are fully based on conventional DLX processor model. Two PC-relative conditional braches BEQZ, BNEZ, BFPT, and BFPF together with unconditional J, JAL, JALR and JR are all processed in the pipeline slot 1. The actual PC address is modified in the ID stage together with condition evaluation. This leads to 1 cycle branch penalty which now

corresponds to 5-operation slots. For allowing more optimizations for students, we have implemented delayed branch semantics in our VLIW pipeline. As a result, the VLIW instruction in the branch delay slot is always executed.

2.7. VLIW operation slots vs. exec. pipelines

It is already clear that there is a 1:1 relationship between the execution pipelines and operation slots in VLIW-DLX instructions. This is a very straightforward approach used in first VLIW processors. It has the well known disadvantages of wasted space in the instruction memory (occupied by NOPs in unused slots). We feel that it is better to stay with this simple interface for an introductory simulator. It also shows to students why the commercial VLIW architectures uses concept of instruction compression or operation bundles and templates.

3. VLIW-DLX Simulator

3.1. Overview of simulator implementation

The whole simulator is written in Java and runs on every computer with Java J2SE runtime environment 5.0 or later. It uses a popular Swing library for GUI implementation and requires a minimal resolution of 1024x768. Implementation of the VLIW pipeline structure is fixed in the simulator executable and corresponds to model described in section 2. However, the provisions were made that the number of execution pipelines and mapping of operation groups to VLIW slots can be easily modified.

Table 3. Main classes used in VLIW-DLX sim.

Class name	*Purpose*
`Instruction`	Every DLX instruction (VLIW-DLX operation) is implemented as a class which is descendant of Instruction class. Method `execute()` defines the behavior of a given operation.
`InstructionSet`	Descendants of this class define groups of operations which could be mapped onto execution pipelines. Subset of `InstructionDLXSet`.
`Pipeline`	Defines properties of execution pipelines i.e. number of stages and groups of operations executable in this pipeline.
`Pipelines`	Defines overall structure of the VLIW pipeline.

Main classes in the program are described in the Table 3. User can define a new instruction (VLIW operation) by defining a new class (descendant of `Instruction`) and register it in the `InstructionDLXSet` class. Operations can be grouped into groups represented by class `InstructionSet.` A snippet of the code in the example 1 shows the definition of first execution pipeline and their incorporation into VLIW-DLX pipeline.

Example 1. Definition of VLIW-DLX pipeline

```
// Defining operation groups (instruction sets)
InstructionSet ISetB = new InstructionSetBranch();
InstructionSet ISetI = new InstructionSetInteger();
InstructionSet IntegerSet = new InstructionSet("1. Pipeline");

// Registering instructions
IntegerSet.addInstructionSet(ISetB);
IntegerSet.addInstructionSet(ISetI);

// Definition of execution pipeline IntegerPipel
Pipeline IntegerPipe1 = new Pipeline(IntegerSet, new String[]
{"IF", "ID", "EX", "WB"});
// number of stages can be defined here
// allowed stage names are any strings. Behavior of
// instructions is defined for ID, MEM and WB stages

   // definition of VLIW-DLX pipeline
new Pipelines(new Pipeline[] {IntegerPipe1, IntLoadPipe,
LoadStorePipe, FPPipe, MultPipe});
```

3.2. VLIW-DLX Simulator Features

To simplify orientation of students VLIW-DLX uses the same colors for the pipeline stages as the WinDLX (see Figure 2). It also uses the same assembly declaration conventions which allow rewriting the DLX programs for VLIW-DLX easily.

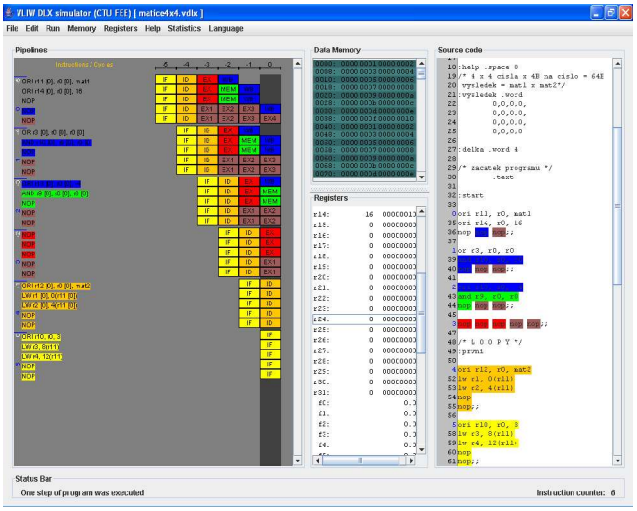

Figure 2. VLIW-DLX simulator snapshot

A single VLIW-DLX instruction is specified as a group of 5 DLX instructions which corresponds to VLIW operations. VLIW instructions are separated from each other by two semicolons. Positions of operations within an instruction must match the pipeline structure. If a given execution slot is not used, explicit `nop` operation must be included in a given position. The extreme case is the instruction with
5 `nop`s which is sometimes inevitable.

Example 2. VLIW-DLX assembly syntax

```
beqz r14, finish
addi r11, r11, 4
sw result(r13), r22
nop
mult r18, r4, r8;;

/* Delay slot */
bnez r10, lab
subi r10, r10, 1
nop nop nop ;;

nop nop nop nop nop;;
```

Simulator includes the editor of the program in the assembly language with common editing features. After the edition is completed, the program must be compiled into internal representation of the simulator. During the compilation, the compatibility of operations with VLIW execution slots is checked and in case that there is the miss-match, the error is reported. Currently the compiler is not checking that the RAW dependences over registers are correctly handled. This effect is intentionally left to students to debug in the runtime.

For simplification of debugging and visualization, the operations in the listing are colored by the corresponding pipeline stage in which they are in a given cycle. This is very similar to the view in the WinDLX simulator. To enhance the debugging properties, the actual values of input registers are shown in the pipeline viewer. This allows easy checking whether RAW dependences are correctly handled in the program code.

Simulator provides statistics in terms of number of executed VLIW instructions (which is equal to number of execution cycles) and also the number of *NOPs* in every pipeline slot as a measure of efficiency.

4. Teaching with VLIW-DLX

VLIW-DLX is intended to be used in undergraduate computer architecture course (X36APS) shortly after introduction of static superscalar processors and VLIWs on the lectures. It is planned to use a single lab seminar to introduce the VLIW-DLX simulator. Similarity to DLX processor allows saving time when introducing the simulator. As a demonstration program we can use the well known kernel "*adding of constant to an array"* used in [5] to demonstrate basic functionality of loop optimizations and ILP processors (see example 3). It is a very simple loop with wast amount of parallelism. Latency of single iteration is 3 cycles for load, 4 cycles for floating point addition and 1 cycle for storing value back. The trivial implementation on VLIW-DLX takes around 800 cycles and VLIW parallelism is used only for hiding the loop overhead. We can see from the Figure 3 that number of *nops* in various pipelines and number of execution cycles decreases while employing loop unrolling and software pipelining.

Example 3. „Adding of constant to an array“ kernel

```
float x[100];
float k=3.1415;

for (i=0; i<100; i++)
     x[i]+=k;
```

Number of operations per a clock cycle (equivalent to IPC) is the measure of pipeline utilization. We can see on Figure 4 that best utilization is reached by the software pipelining. It achieves OPC=3.57 which is not bad if we consider that the 5th execution pipeline is not used in this particular kernel.

Undergraduate students will be required to optimize a simple loop-intensive kernel and achieve minimal execution time. Our intention is to use the same kernel which students implement on the standard DLX processor. Currently the students are required to optimize the 10x10 matrix multiplication kernel and commonly achieve the execution time of below 5500 cycles by using the loop unrolling on WinDLX. We hope that students will achieve better results by employing loop unrolling and software pipelining on VLIW-DLX.

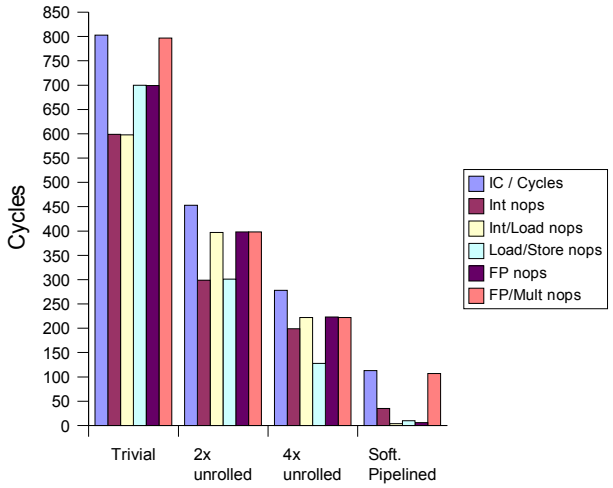

Figure 3. Performance of simple kernel on VLIW-DLX

Tuning of the VLIW-DLX pipeline and debugging of the simulator was done by students of graduate computer architecture course in this semester. Students were required to implement the SAXPY loop (single precision version of the famous DAXPY loop) on the VLIW-DLX. Preliminary results show that efficiency of software pipelining in this particular loop is limited by the number of integer and load/store execution units and their latency. Currently the 4-times unrolled version of SAXPY achieves better results than a software-pipelined version.

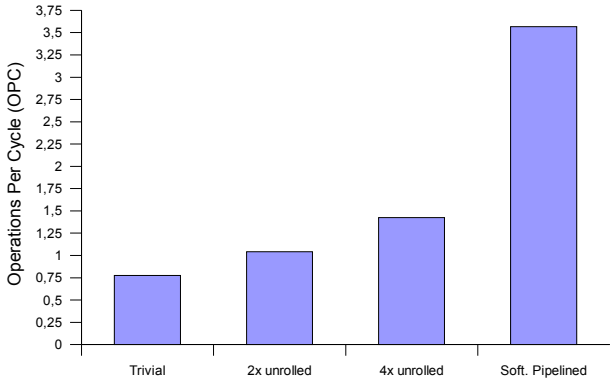

Figure 4. Pipeline utilization on simple kernel

After evaluation of more examples we will modify the VLIW-DLX pipeline to achieve better results by employing the software pipelining optimization.

6. Conclusions and Future Work

Current version of VLIW-DLX presented in this paper is far from optimized VLIW design. However we believe that it will serve its purpose to introduce the concept in undergraduate computer architecture course. Simulator implementation is not fixed and allows architecture tuning and future expansion and modifications. Based on our first experience in use of the simulator we can decide to add more realistic resource constrains (e.g. limiting number of register file read and write ports) or extend the architecture by additional pipeline slot and change the mapping of operations to VLIW slots. Students will use the experience with this architecture in later electory courses when more advanced VLIW processors are introduced.

VLIW-DLX presented in this paper is the next result in our planned computer architecture tool set based on original DLX ISA. VHDL model of integer DLX pipeline HDLDLX [13] is currently used to illustrate implementation aspects of instruction pipelining. New DLX simulator currently under development will replace the aging WinDLX and allow simulating more variants of RISC pipeline including vector instructions.

Current version of VLIW-DLX including source files is available for download at http://service.felk.cvut.cz/vlsi/prj/vliwdlx/

7. References

[1] J.A.Fisher, "Very Long Instruction Word Architectures and the ELI-512", *Proceedings of the 10th Annual International Symposium on Computer Architecture,* June 1983 pp. 140-150

[2] R.P.Colwell, R.P.Nix, J.J.O'Donell, D.B.Papworth and P.K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Transactions on Computers* vol.37, no.8, August 1988, pp.967-979

[3] B.R. Rau, D.W.L. Yen, W.Yen and R.A. Towie, "The Cydra-5 Departmental Supercomputer: Design Philosophies, Decissions, and Trade-Offs," *IEEE Computer,* vol.22, no.1, Jan 1989, pp.12-26, 28-30,32-35

[4] L.Gwennap,"Intel, HP make EPIC disclosure". *Microprocessor Report*, 11(14):1–9, October 1997.

[5] J.L.Hennessy and D.A. Patterson: *"Computer Architecture: A Quatitative Approach",* 4th eddition, Morgan Kaufmann Publishers (an imprint of Elsevier), 2006, pp. 179-181

[6] N. Mitchell, "Philips TriMedia: a digital media convergence platform", *Proceedings of WESCON/97*, Santa Clara, November 1997, pp. 56-60

[7] J.A.Fisher, P.Faraboschi, C.Young:"*Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools",* Morgan Kaufmann Publishers (an imprint of Elsevier), 2005

[8] On web http://www.trimaran.org/

[9] S. Aditya, B.R. Rau and V. Kathail:"Automatic architectural synthesis of VLIW and EPIC processors". In *Proceedings of 12th international symposium on System Synthesis*. November, 1999, pp.2-7.

[10] http://www.vliw.org/book/downloads.htm

[11] H. Gruenbacher, M. Khosravipour:"WinDLX and MIPSim Pipeline Simulators for Teaching Computer Architecture", *Proceedings of IEEE Symposium and Workshop on Engineering of Computer Based Systems* (ECBS'96) Friedrichshafen, 1996, GERMANY

[12] J.L.Hennessy and D.A. Patterson: *"Computer Architecture: A Quatitative Approach",* 1st eddition, Morgan Kaufmann Publishers, 1990

[13] M. Bečvář: "Teaching Basics of Instruction Pipelining with HDLDLX" , *Proceedings of Workshop on Computer Architecture Education,* Munchen 2004, pp.74-78