Extending DLXsim for Parallel Architectures

Celso L. Mendes[*]

Department of Computer Science
University of Illinois
Urbana, Illinois 61801
E-mail: mendes@cs.uiuc.edu

ABSTRACT

This paper presents two extensions of a RISC processor called DLX. We extend DLX's functionality for a vector and for a parallel architecture. The vector extension, DLXV, has pipelined vector functional units and is able to perform chained vector operations. The parallel extension, DLXMP, is a message-passing multicomputer with a basic DLX on each node. We present simulators for both extensions, describe their major features, and show examples of their use.

Key words: RISC processor simulation, vector architectures, multicomputers, high-performance computing.

1 Introduction

This paper describes two extensions of DLXsim [2], the simulator of a typical RISC processor called DLX. Those extensions aim at using DLX as a building block for high performance architectures. This work was done as an intermediary step of our research on performance analysis on multicomputers, where the results of this project will be used to simulate multicomputers having enhancements over currently existing systems.

Given the availability of DLXsim, and considering that most existing multicomputer systems are based on some type of RISC processor, we decided to use DLXsim as a starting point in our implementation. We made two independent extensions to the original DLXsim: the first extension, DLXVsim, simulates DLXV (the vector extension of DLX), as described in the book by Hennessy and Patterson [1]; DLXV has fully pipelined vector functional units, and allows chaining of vector operations. The second extension, DLXMPsim, simulates a message-passing multicomputer having the original DLX as the processing element on each node.

[*]Supported by the Brazilian Institute of Space Research (INPE).

DLXVsim is supposed to run on any Unix based machine; in fact, it should run on any machine that can run DLXsim. On the other hand, for efficiency reasons, DLXMPsim was originally built on a real multicomputer, the Intel Paragon-XP/E, but it can also run on any cluster of workstations where a Paragon simulation environment is available, like that provided by NXlib [5]; this last option might even consist of a configuration with a single Unix machine.

Both DLXVsim and DLXMPsim execute code from assembly programs, possibly written in the C language and compiled by `dlxcc`, the original compiler for DLX. Because `dlxcc` is *not* a vectorizing compiler, vector instructions must be manually inserted by the user directly in the assembly source code.

The remainder of this paper is organized as follows. In §2 we present the features of DLXVsim, and in §3 those of DLXMPsim, both from a user's point of view. §4 shows some examples of use for both simulators. These three sections should provide enough information for those interested only in using DLXVsim and DLXMPsim; for more detailed comments on the internal operation of these simulators, see [4]. §5 gives the current status of the simulators, with information on how to obtain them. Finally, §6 concludes our presentation.

2 DLXVsim Vector Extension

This section presents the main features of DLXV, the vector extension of the DLX processor, and the user commands introduced to handle those features during the simulation with DLXVsim. The desired configuration of the simulated DLXV can be set by the user with corresponding command line switches to DLXVsim, or by changing appropriate constants in the source code, as indicated below, and recompiling the simulator.

2.1 Vector Registers and Functional Units

DLXV has all the original scalar funtional units of DLX, plus a set of vector functional units where vector instructions are executed. Figure 1 shows these functional units. The scalar units are not pipelined, but most of them can be replicated. On the other hand, the vector units are fully pipelined, and allow chaining between vector operations. The latencies for both types of functional units can be selected by the user with the command line switches `-al`, `-ml` and `-dl`, for the adders, multipliers, and dividers, respectively. Notice

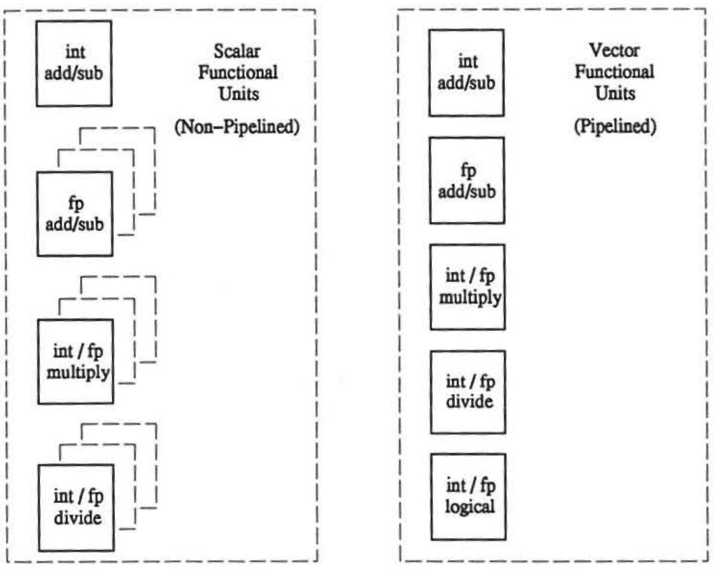

Figure 1: DLXV functional units.

that the same latency is assumed for the scalar and vector units on the same operation. The vector integer adder and the vector logical unit have a latency of one cycle, similarly to the scalar integer adder.

In addition to the same register set of DLX (general-purpose registers R0-R31 and floating-point registers F0-F31), DLXV has a vector register file composed of a group of vector registers; the default number of vector registers is eight, but it can be changed with the use of an appropriate command line switch. Each vector register has sixty-four 64-bit elements. There are also two special registers, VLR (Vector-Length Register) and VMR (Vector-Mask Register). The contents of VLR may vary between 0 and 64, defining the length of any vector operation; VMR is a 64-bit register, which can be used to disable operations on particular elements of a vector (by storing the value 0 in the corresponding bit of VMR).

Figure 2 shows the connection between the vector register file and the other system components. There can be one or more 64-bit pipelined buses between the vector register file and memory, in both directions. Each vector register has one write port, and one or more read ports, as indicated by Figure 3, so that more than one vector functional unit may receive data from the same vector register simultaneously. We assume that the inputs of each vector functional unit can be connected to the output of any vector register, by means

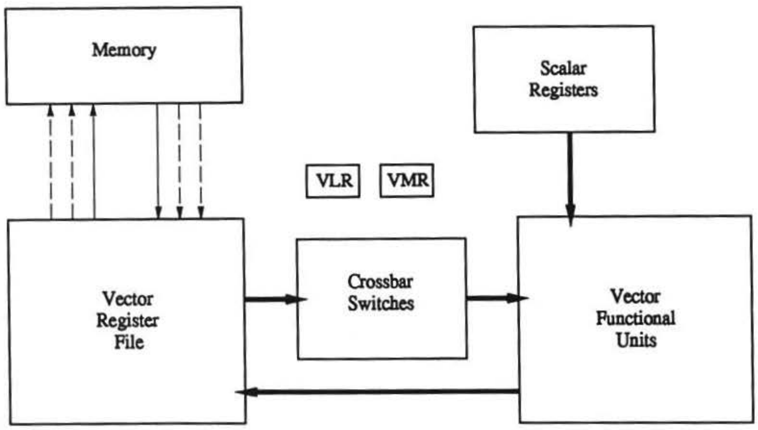

Figure 2: DLXV vector register file.

of crossbar switches, like mentioned in [3].

2.2 Vector Instructions

The instruction set of DLXV contains as a subset all the instructions originally present on DLX. Figure 4 shows the complete list of DLXV instructions, with their corresponding opcodes and binary representation. On that list, instructions introduced for DLXV are represented in *italics*.

Vector instructions can operate on data consisting of integer or floating-point numbers. Integer numbers are treated as *signed* integers, and floating-point numbers as being in *double-precision* format. This is not as flexible as in DLXsim, in which integers can be signed or unsigned, and floating-point numbers can be in

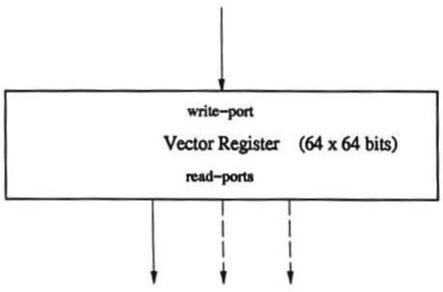

Figure 3: DLXV vector register

Main opcodes

	$00	$01	$02	$03	$04	$05	$06	$07
$00	SPECIAL	FPARITH	J	JAL	BEQZ	BNEZ	BFPF	BFPT
$08	ADDI	ADDUI	SUBI	SUBUI	ANDI	ORI	XORI	LHI
$10	RFE	TRAP	JR	JALR	SLLI		SRLI	SRAI
$18	SEQI	SNEI	SLTI	SGTI	SLEI	SGEI		*LVD*
$20	LB	LH	*LV*	LW	LBU	LHU	LF	LD
$28	SB	SH	*SV*	SW			SF	SD
$30	SEQUI	SNEUI	SLTUI	SGTUI	SLEUI	SGEUI		*SVD*
$38								

Special opcodes (Main opcode = $00)

	$00	$01	$02	$03	$04	$05	$06	$07
$00	NOP			*POP*	SLL	*LVWS*	SRL	SRA
$08	*ADDV*	*ADDSV*	*MULTV*	*MULTSV*		*SVWS*	*SUBV*	*DIVV*
$10	SEQU	SNEU	SLTU	SGTU	SLEU	SGEU	*SUBSV*	*DIVSV*
$18	*SEQV*	*SNEV*	*SLTV*	*SGTV*	*SLEV*	*SGEV*	*SUBVS*	*DIVVS*
$20	ADD	ADDU	SUB	SUBU	AND	OR	XOR	*CVM*
$28	SEQ	SNE	SLT	SGT	SLE	SGE	*MOVI2L*	*MOVL2I*
$30	MOVI2S	MOVS2I	MOVF	MOVD	MOVFP2I	MOVI2FP	*MOVFP2M*	*MOVM2FP*
$38	*SEQSV*	*SNESV*	*SLTSV*	*SGTSV*	*SLESV*	*SGESV*	*LVDWS*	*SVDWS*

Floating Point opcodes (Main opcode = $01)

	$00	$01	$02	$03	$04	$05	$06	$07
$00	ADDF	SUBF	MULTF	DIVF	ADDD	SUBD	MULTD	DIVD
$08	CVTF2D	CVTF2I	CVTD2F	CVTD2I	CVTI2F	CVTI2D	MULT	DIV
$10	EQF	NEF	LTF	GTF	LEF	GEF	MULTU	DIVU
$18	EQD	NED	LTD	GTD	LED	GED	*SUBVSD*	*DIVVSD*
$20	*ADDVD*	*SUBVD*	*MULTVD*	*DIVVD*	*ADDSVD*	*SUBSVD*	*MULTSVD*	*DIVSVD*
$28	*SEQVD*	*SNEVD*	*SLTVD*	*SGTVD*	*SLEVD*	*SGEVD*		
$30	*SEQSVD*	*SNESVD*	*SLTSVD*	*SGTSVD*	*SLESVD*	*SGESVD*		
$38								

Figure 4: DLXV opcodes

single or double-precision, but should be enough to handle most scientific programs, where signed integers are the common case, and single-precision floating-point values can always be converted to double-precision without loss of accuracy. Thus, vector registers are supposed to contain either signed integer values (in the lower 32 bits) or 64-bit double precision values.

There are instructions for operations between two vectors (e.g. `addv`), producing a third vector as a result, and for operations between a scalar and a vector (e.g. `addsv`), producing the result in another vector. Vector instructions involving floating-point operands are appended by the letter "d" (e.g. `addvd`).

2.3 New Switches and Commands

In addition to all the command line switches available in DLXsim, the following switches were introduced for DLXVsim:

- `-vr`: specify the number of vector registers;
- `-wb`: specify the number of buses from the vector register file to memory;
- `-rb`: specify the number of buses from memory to the vector register file;
- `-rp`: specify the number of read ports in each vector register.

The number of vector registers cannot be greater than 32 due to a limitation in the number of bits in the opcode field where the vector register is specified. The maximum value for the other parameters can be changed by modifying the appropriate constants in file `dlx.h`.

In order to support interactive manipulation of the vector registers, we added the following user commands in DLXVsim:

- `vget`: return the contents of a vector register, treating them as integer values;
- `vput`: store an integer value in an element of a vector register;
- `fvget`: return the contents of a vector register, treating them as double-precision floating-point values;
- `fvput`: store a floating-point value in an element of a vector register.

The commands `vget/vput` and `fvget/fvput` work on vector registers in a similar fashion as the commands `get/put` and `fget/fput` work on the general-purpose registers.

2.4 Memory Bank Conflicts

DLXVsim allows the user to assess the effects of memory bank conflicts on a given vector operation. This is done on the basis of individual vector instructions; scalar memory accesses are not affected.

There are three parameters in file `dlx.h` that define the characteristics of DLXV's interleaved memory system: `MEMORY_READ_LATENCY` gives the latency of each memory read operation; `NUM_MEMORY_BANKS` defines the number of memory banks, and `MEMORY_CYCLE_TIME` indicates the minimum valid time interval between successive accesses to the same memory bank.

When a vector load or store instruction is issued, the memory start address and stride define the access pattern, and possible conflicts can be detected and handled, according to the parameters above. Like in DLXsim, memory writes are assumed to have zero latency when initiated, but are still subject to the bank conflicts previously described.

Notice that vector load operations can be chained to other instructions accessing the same vector register. On the other hand, vector store operations *cannot* be chained to subsequent vector load operations involving the same memory region. All the memory accesses from vector instructions are still recorded in the trace file when the "`trace`" mode is on.

2.5 New Simulation Statistics

In DLXVsim, the summary of executed instructions contains also information regarding the vector operations. With the "`stats opcount`" option, in addition to the counts of Integer and Floating-Point operations, the execution counts for all new "Vector-Related Instructions" are also shown.

When stall information is requested by the user, the number of stall cycles due to vector stalls is presented, after the numbers for load and floating-point stalls.

Finally, much more detailed information about the status of DLXV can be displayed on each simulation step, by recompiling the simulator with the compiler switch `-DDEBUG` enabled.

3 DLXMPsim Parallel Extension

DLXMPsim simulates a multicomputer composed of one ore more computing nodes that communicate by exchanging messages. Each computing node is a regular DLX processor, and interprocessor communication is achieved by the execution of appropriate system calls. This section presents the major features of the simulation scenario, with a description of the available system calls comprising the user program message-passing interface.

3.1 Simulation Environment

For efficiency of the simulation, we decided to build DLXMPsim on a real multicomputer, the Intel Paragon, so that the simulation of each DLX computing node is done in one processor of the Paragon. DLXMPsim also works on a workstation cluster where a package like NXlib is installed, and in that case the user has complete control to decide which machine will execute the simulation of each DLX computing node.

A question that may arise at this point is: "Why use DLXMPsim, instead of using the parallel environment (like provided by NXlib) directly?" To answer that question, we present all the arguments that are valid for a simulation approach: with DLXMPsim, the user can change parameters in the processor, like the number and latencies of functional units, or in the interconnection network, like the cost of message transmission, and assess their corresponding effects on total execution time for a given program.

The user program is supposed to be in the SPMD style, with a common object code being executed by every DLX computing node. Data sharing is accomplished by exchanging messages between processors. In addition to the system calls already available in DLXsim, we added a new set of calls which allow message-passing with a variety of blocking and nonblocking functions. DLXMPsim simulates these message-passing calls by using native communication functions of the Paragon, and handling the virtual time of the simulated machine to ensure that causality relationships are mantained across the simulation.

3.2 Message-Passing System Calls

DLXMPsim extends the system calls for a regular DLX processor with a set of functions related to message-passing. These new functions can be called by the user program to send or receive messages, to get information on system configuration, or to obtain the status of a transmission.

The message send function is nonblocking, meaning that the sender process can proceed independently of the status of the receiver for the underlying message. Upon returning from the send function, the user program can reuse the data buffer.

There are two types of message receive functions: a blocking version, where the receiver process cannot proceed until the message arrives, and a nonblocking version, in which control returns immediately to the receiver; in the latter case, the user program must make a subsequent call to a probe function to obtain the status of message arrival.

The functions introduced in DLXMPsim are summarized below:

- Message-passing functions:
  - `send`: send a message with a given tag to another processor;
  - `recv`: receive a message with a given tag; control will return to the caller only after the message arrives;
  - `nblk_recv`: similar to `recv`, but control returns immediately to the caller; the return value is a unique message-identifier, which can be used in subsequent probe function calls to check if the message has arrived.
- Probe functions:
  - `msgwait`: block the caller until the message with the given identifier arrives;
  - `msgdone`: check if the message with the given identifier has arrived.
- Configuration inquiring functions:
  - `numnodes`: return the number of nodes being simulated;
  - `mynode`: return the node number of the current process.

3.3 DLXMPsim Configuration

Some command line options allow the user to configure the simulation environment of DLXMPsim. In addition to all the options existing in DLXsim, the following switches are also available:

- `-sz`: specify the number of computing nodes being simulated.
- `-io`: this option redirects the standard input and output of the simulator. When executing the simulation in parallel, it becomes difficult (and confusing) to support interactive commands, because it is not clear which processes should be affected by each command. Thus, we decided to provide support for non-interactive user commands. The user builds a command-file where he places all the commands to be executed during the simulation; all the processes read comands from this file. Across the simulation, each process sends its output to a specific output file, and the user can verify the results of the simulation by inspecting these output files.

DLXMPsim represents the time for message transmission by a cost model where the total time to send a given message is composed of a constant value added to a linear function of the message length. Thus, if the message is sent at time T_S, it cannot be received by another node before time $T_S + f(n)$, where n is the message length and f is the transmission cost. In the current version, DLXMPsim does not model any type of contention in the interconnection network between nodes; the network is simply supposed to be always available for message transmission.

During a simulation session in which stall information is requested by the user, DLXMPsim also prints the number of cycles spent on functions that block waiting for the arrival of a message. This is termed as "Msg-Passing Idle Cycles".

Finally, we should notice that it is relatively easy to insert new system calls to the simulator. The functions mentioned above can be used as a model.

4 Examples of Use

We now illustrate the use of our simulators, DLXVsim and DLXMPsim, with two examples. These examples are not intented to cover all the features of the simulators; they should just provide an overview of the corresponding simulation scenarios. The first example refers to DLXVsim, and the last one to DLXMPsim.

4.1 Example of DLXVsim with Chaining

In this example, we present chained vector operations, where two vectors loaded from memory are multiplied, and the resulting vector is added to a given scalar value. We assume that DLXV is configured with the default parameters, and no memory conflicts arise. Figure 5 presents the assembly program for this example.

Figure 6 shows steps of an interactive simulation session with DLXVsim; notice that, at the end of the simulation, the final vector has, indeed, the expected contents. Figure 7 reproduces the execution on a cycle-by-cycle basis; in this figure, terms of the form Xi mark the production of a new value into vector register V_i by instruction X. As one can see, the multiply instruction (MULTVD) is stalled by one cycle, due to a data dependence on vector register V_2; the vector addition is also stalled, and cannot issue before the multiplication results start being produced.

For both the multiplication and the addition, after an initial latency (five cycle for the multiplication and

```
        .data 0
;       scalar constant:
d0:     .double 2.0
;       two original vectors:
d1:     .double 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0
d2:     .double 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0
        .text
;       set vector length register:
Begin:  addi    r8,r0,8
        movi2l  r8
;       load scalar constant:
        ld      f0,d0
;       set mem start addresses:
        addi    r1,r0,d1
        addi    r2,r0,d2
;       load vectors:
        lvd     v1,r1
        lvd     v2,r2
;       multiply the two vectors:
        multvd  v0,v1,v2
;       add scalar value:
Vadd:   addsvd  v4,f0,v0
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
End:
```

Figure 5: Example of a DLXV program with chained vector operations.

two cycles for the addition), one new result per cycle is produced. Although there are no more meaningful instructions in the program after the addition, we must simulate a few more steps (with NOP instructions) so that all the addition results are produced.

4.2 Example of DLXMPsim

This example shows the simulation of a very simple program to compute the dot product between two arrays, under DLXMPsim. The array values are initialized by node 0 and then distributed to all the computing nodes. Each node computes a partial dot product, and sends its result to node 0, which computes the final answer. Note that this is not the best algorithm for this problem; it merely serves as an example of our simulator.

Figure 8 shows the high-level language program for this example. We simulated the execution of this

```
(dlxvsim)[0] load example_1.s
(dlxvsim)[0] stop at Vadd
(dlxvsim)[0] go Begin
stop 1, pc = Vadd: addsvd v4,f0,v0
(dlxvsim)[9] fvget v1
v1: (0-->1-->2-->...)
 1.000000 2.000000 3.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
(dlxvsim)[9] step
stopped after single step, pc = Vadd+0x4: nop
(dlxvsim)[14] fvget v0
v0: (0-->1-->2-->...)
 11.000000 24.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
(dlxvsim)[14] stop at End
(dlxvsim)[14] go
stop 2, pc = End: nop
(dlxvsim)[22] fvget v4
v4: (0-->1-->2-->...)
 13.000000 26.000000 41.000000 58.000000 77.000000 98.000000 121.000000 146.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
(dlxvsim)[22] quit
```

Figure 6: Simulation of program with chained vector operations under DLXVsim.

```
cycle--> 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22

   i     A  M  L  A  A  L    L1 L1 L1 L1 L1 L1 L1 L1
   n     D  0  D  D  D  V
   s     D  V     D  D  D  L     L2 L2 L2 L2 L2 L2 L2 L2
   t     I  I     I  I     V
   r        2              D  M===             M0 M0 M0 M0 M0 M0 M0 M0
   u        L                 U===
   c                          L===  A============    A4 A4 A4 A4 A4 A4 A4 A4
   t                          T===  D============
   i                          V===  D============  N  N  N  N  N  N  N  N  N
   o                          D===  S============  0  0  0  0  0  0  0  0  0
   n                                V============  P  P  P  P  P  P  P  P  P
                                    D============
```

Figure 7: Cycle-by-cycle execution of program with chained vector operations.

program with four computing nodes, using an input command file with the following contents:

```
load example_dp.s newtraps.s
go _main
stats opcount stalls
quit
```

The simulation results obtained from node 0 are in Figure 9, and those from node 1 in Figure 10 (the results for nodes 2 and 3 are similar to those of node 1). As one can see, the idle time in node 0 is relatively low, corresponding to the wait for the arrival of the first partial result from other nodes. On the other hand, all nodes keep waiting for their first message while node 0 performs the initialization of the whole array.

5 Status and Availability

DLXVsim and DLXMPsim have been tested with several example programs. We are nearly finished with the tests of their preliminary versions. DLXMPsim was tested only on a cluster of Sparcstations, under NXlib-1.0. Our next step is to test it under NXlib-1.1, so that other plataforms can be used.

In the future, we intend to add features to extract more information during execution, regarding the dynamic behavior of the program. This would take the form of execution traces, like DLXsim does with memory accesses.

The simulators are available to anybody who wants to use them and is willing to report problems, enhancements or experiences. They may be obtained by contacting the author by e-mail.

```
#define DATAVECTOR       0                /* data vector message type       */
#define RESULT           1                /* result message type            */
#define VECLEN           256              /* problem size */

main()
{
        float   X[VECLEN], Y[VECLEN], PartialProduct, InnerProduct;
        int     i,N,Offset,UpperBound,VectorLength,myNode,nodeCount;

        myNode = mynode();
        nodeCount = numnodes();
        N = VECLEN;

        if (myNode) {
                recv(DATAVECTOR, X, sizeof(float) * N);
                recv(DATAVECTOR, Y, sizeof(float) * N);
        }
        else {  /* Node 0 initializes and distributes array values */
                printf("Vector length: %d\n", N);
                for (i = 0; i < N; i++) {
                        X[i] = i; Y[i] = 1.0;
                }
                send(DATAVECTOR, X, sizeof(float) * N, -1);
                send(DATAVECTOR, Y, sizeof(float) * N, -1);
        }

        VectorLength = N / nodeCount;
        Offset = VectorLength * myNode;
        PartialProduct = 0.0;

        if (myNode == nodeCount - 1)  UpperBound = N;
        else UpperBound = (myNode + 1) * VectorLength;

        /* Local computation of partial results */
        for (i=Offset; i<UpperBound; i++) PartialProduct += X[i] * Y[i];

        if (myNode != 0) send(RESULT, &PartialProduct, sizeof(float), 0);
        else {  /* Node 0 computes final result */
           InnerProduct = PartialProduct;
           for (i = 0; i < nodeCount - 1; i++) {
              recv(RESULT, &PartialProduct, sizeof(float));
              InnerProduct += PartialProduct;
           }
           printf("Inner product is: %8.4f\n", InnerProduct);
        }
}
```

Figure 8: Example of a message-passing program to compute dot product.

```
(dlxmpsim) load example_dp.s newtraps.s
(dlxmpsim) go _main
Vector length: 256
Inner product is: 32640.0000
TRAP #0 received
(dlxmpsim) stats opcount stalls
Load Stalls = 1306
Floating Point Stalls = 221
Msg-Passing Idle Cycles = 1111
...
Total integer operations = 8758
                        FLOATING POINT OPERATIONS
    ADDD        0       ADDF        67     CVTD2F       0    CVTD2I       0
...
   MULTF       64       MULTU        0        NED       0       NEF       0
...
Total floating point operations = 391
Total operations = 9149
Total cycles = 11787
(dlxmpsim) quit
```

Figure 9: Output from simulation of dot product program produced by node 0.

```
(dlxmpsim) load example_dp.s newtraps.s
(dlxmpsim) go _main
TRAP #0 received
(dlxmpsim) stats opcount stalls
Load Stalls = 264
Floating Point Stalls = 218
Msg-Passing Idle Cycles = 9290
...
Total integer operations = 1737
                        FLOATING POINT OPERATIONS
    ADDD        0       ADDF        64     CVTD2F       0    CVTD2I       0
...
   MULTF       64       MULTU        0        NED       0       NEF       0
...
Total floating point operations = 131
Total operations = 1868
Total cycles = 11640
(dlxmpsim) quit
```

Figure 10: Output from simulation of dot product program produced by node 1.

6 Conclusion

We presented two extensions of the RISC DLX processor, targeting high performance architectures. DLXVsim allows the user to simulate a vector machine similar to traditional supercomputers. DLXMPsim can simulate massively parallel systems, where communication between processors is performed by message-passing.

Besides their more obvious use as educational resources, one can use such simulators as a tool in the simulation of more sophisticated systems. We are currently investigating their integration to simulate a multicomputer where each node has a vector architecture.

With these simulators, the user can not only obtain quantitative performance information from the execution of a given program, but also change the machine configuration, and assess the effects of such changes on overall performance.

References

[1] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[2] HOSTETLER, L. B., AND MIRTICH, B. *DLXsim — A Simulator for DLX.* University of California, 1990.

[3] LEE, C. G., AND SMITH, J. E. A study of partitioned vector register files. In *Supercomputing'92* (Minneapolis, November 1992), pp. 94–103.

[4] MENDES, C. L. *Guidelines for Using DLXVsim and DLXMPsim.* University of Illinois at Urbana-Champaign, 1994.

[5] STELLNER, G., ET AL. *NXLIB User's Guide.* Technische Universität München, 1993.