NIW: A Simple Superscalar Architecture

Crispin Cowan

Computer Science Department
University of Western Ontario
London, Ontario

crispin@csd.uwo.ca

Abstract

Many papers have appeared in the literature recently describing strategies to improve processor performance by issuing more than one instruction per processor clock cycle, a technique collectively known as *superscalar*. Frequently, superscalar systems involve a great deal of complexity in the processor design, the compilers, or both. There have also been many papers addressing issues in the design of cache memories, to reduce performance degradation caused by processors being forced to wait for their relatively slow main memories. In this paper we present a design that implements a subset of the general superscalar concept, providing one load/store operation, and one concurrent ALU operation in each clock cycle. Such a design provides the benefits of the general superscalar concept up to parallelism of degree 2, while alleviate the memory bottleneck by fully exploiting the processor's channel to memory by issuing a load or store instruction on every cycle.

Arguments are presented as to why such a design would not impose great demands on CPU engineers. Simple algorithms are presented that approach full utilization of such a processor, and a working compiler implementation is being developed. Finally, small code sequences are analyzed comparing a conventional pipelined RISC processor to one modified as described.

1 Introduction

The last decade has been a fruitful one for producing improvements in processor performance, both through process technology (making the silicon go faster) and architecture (using the silicon to greater effect). Because CPUs are more complex than memory chips, there is more room for improvement, and they have improved more quickly. As a result, modern pipelined RISC processors are now bottlenecked on their memory bandwidth.

To illustrate this, consider the DLX architecture[9] (very similar to the MIPS R2000/R3000[6]). In [9], Hennessy and Patterson found that this processor spent 26% of its instructions loading or storing data. It is hard to push this fraction down much further, because even an aggressive compiler cannot keep more than about 26 registers in use, and so the compiler must resort to memory references.

An alternative approach taken by the SPARC architecture[2] is to use register windows. A register window is a bank of registers, the bank being indicated by bank selection register. When a subroutine is called, a new bank is selected. The banks are managed in a ring; when more banks are needed than exist, a "spill" occurs, and the top-most bank is written out to memory. This approach works well when the depth of the subroutine call stack does not exceed the number of banks provided, but is of little or no benefit when the depth exceeds the number of banks, as is frequently the case in large, modular programs. The negative impact of a "spill" is that, instead of the compiler writing one or two registers to memory as needed, the entire bank is written out, even though only half of it may contain useful data.

Observing that the above machines (which we regard as typical) spend a large amount of their time reading and writing data to memory instead of computing, we now propose an architecture which will compute one value while reading or writing another. A register-register compute instruction, and an independent load or store instruction, are encoded into a single 32-bit instruction word. These two independent instructions, statically bound together by the compiler, will be executed in parallel by the CPU. The NIW is a superscalar architecture that is similar to machines known as VLIW, or Very Long Instruction Word, which encode several (many) instructions into a single, long instruction word. Because we are encoding multiple instructions (two) into a single instruction word, similar to VLIW architectures, and we are only using 32-bits, we name this machine the Not-so-wide Instruction Word architecture, or NIW.

It has occasionally been observed that RISC architectures fail to capture the implicit parallelism of CISC complex addressing modes, such as auto-increment. The NIW architecture re-captures this parallelism, while retaining the simple instructions and simple addressing modes of a RISC processor. By preserving simple instructions, the NIW architecture retains the ability to use a simple pipeline in the execution engine, and a relatively simple instruction scheduler in the compiler. Because the compute instruction and the load/store instruction are independent, and both general in nature, the compiler is free to schedule any pair of compute and load instructions together, instead of trying to fit what it wants to compute into a preconceived complex addressing mode.

While it is true that the NIW is limited to a maximum degree of parallelism of two, it has been shown that the natural degree of parallelism present in compiled code, barring loop unrolling, is in fact two[5, 10]. Since loop-unrolling is not generally applicable outside of scientific programs, we regard this as a typical degree of instruction-level parallelism present in frequently used user and system programs. Because instruction-level parallelism is inherently limited to degree two, further costly and sophisticated superscalar efforts will be of little benefit.

The NIW architecture has several advantages over other superscalar architectures. The hardware requirements are simpler: Section 2 will illustrate that the only additional hardware required over a standard pipelined RISC processor is one additional read port and one additional write port into the register file. Furthermore, the number of pipeline stages can be reduced from five to four without performance penalty. The software requirements are simpler: section 3 describes the algorithms necessary to exploit the NIW features, and shows that they are linear with respect to the length of a basic block.

In section 2, we will detail the instruction set and pipeline architecture of the NIW. Section 3 describes the compiler optimization and scheduling techniques necessary to exploit this architecture. Section 4 presents sequences of code generated for the DLX RISC processor, and analyzes the performance impact of transforming this code to use NIW features. Finally, section 5 describes on-going research efforts, principally the compiler and simulator being developed so that the performance impact on real programs may be measured.

2 The NIW Architecture

The NIW architecture is a small modification of the DLX pipelined RISC architecture described in *Computer Architecture: A Quantitative Approach* [9]. The DLX has not actually been implemented; it is a simplification of the MIPS R2000/R3000 architecture, designed for the pedagogical purposes of the text book. As such, it appropriate to describe the NIW as a modification of the DLX for the following reasons:

- By modifying an existing and well studied architecture, we can make direct performance comparisons without having to account for other irrelevant differences.
- The DLX has a simple architecture, simplifying the details of this project.
- A working cross compiler and simulator for the DLX are available, facilitating an easy implementation for the NIW, so that real program measurements can be made.
- Since the DLX itself is similar to the MIPS R2000/R3000, the real-world viability of this architecture is assured.

In this section, we will describe the pertinent details of the DLX architecture, and the changes necessary in each case to implement the NIW architecture. We conclude by discussing the relative hardware cost of the NIW.

2.1 The Instruction Set

The DLX is a straightforward load/store RISC architecture. Operands must be explicitly loaded into registers before they can be operated on; no memory addressing modes are supported. Thus the only permissible operands to an ALU instruction are registers, and 16-bit immediate values. For load and store instructions, there is only one addressing mode: register indirect with 16-bit displacement.

Typical of RISC processors, the DLX has simple instruction formats with fixed fields, to facilitate fast instruction decoding and operand fetching. There are three instruction formats, as illustrated in figure 1. The I-type instructions support instructions with an immediate operand, the R-type instructions support all-register ALU operations, and the J-type instructions support branches with a 26-bit offset.

I-type instruction

6	5	5	16
Opcode	RS1	RD	Immediate

R-type instruction

6	5	5	5	11
Opcode	RS1	RS2	RD	Func

J-type instruction

6	26
Opcode	Offset added to PC

Figure 1: DLX Instruction Formats

I-type instruction

6	5	5	16
Opcode	RS1	RD	Immediate

R-type instruction

6	5	5	5	1	5	5
Opcode	RS1	RS2	RD	L/S	TA	TD

J-type instruction

6	26
Opcode	Offset added to PC

Figure 5: NIW Instruction Formats

Figures 2, 3, and 4 illustrate how the DLX populates these formats with instructions (taken from the DLX simulator documentation[4]). Figure 2 illustrates the I-type instructions. The R-type instructions all have an opcode of `$01`; the specific instruction to be executed is encoded in the Func field, described in figure 3. Similarly, the floating point operations all have an opcode of `$02`, and their function encodings are described figure 4.

To allow a load/store instruction to be encoded along with a register-register ALU operation, it is necessary to restrict the number of instructions and addressing modes, to make room in the instruction encoding. To achieve this, the NIW removes three major classes of operations from the DLX instruction set:

1. Single Precision. Double precision floating point arithmetic is retained. This eliminates a large number of redundant instructions, differing only in the type of the operands, and a large number of conversion instructions.

2. Immediate Operands. Most of the ALU instructions that the DLX allows with immediate operands have been removed. This eliminates a large number of redundant instructions, differentiated only by the presence of an immediate operand. These instructions were not frequently used in the code anyway, except for `addi` (which the DLX uses for "load immediate value") and `slli` (which is used for scaling an index into two- or four-byte member tables).

3. Displacement Addressing. Load and store operations are restricted to simple register indirect. While displacement addressing is frequently used, it can be simulated with almost zero cost by explicitly calculating addresses, due to NIW's available parallelism.

The above pruning of the instruction set produces a severely reduced instruction set computer, but also one in which two instructions can be encoded in a single 32-bit word. Figure 5 illustrates the modified NIW instruction format. The I-type and J-type instructions are unchanged from the DLX format, but the R-type has changed considerably. The Func field has been removed, and the functions merged into the general Opcode field (made possible by the reduction in the number of instructions). There is now a L/S bit, which indicates whether a load or a store is to be executed in parallel with the ALU operation. The TA and TD fields specify the Transfer Address and Transfer Data registers, respectively, for the load/store instruction. To inhibit the load/store operation, specify R0 (which is hard-wired to the value 0) as both the TA and TD register. Figure 6 lists the instructions supported by the NIW.

2.2 Pipeline

The five stages of the DLX pipeline are illustrated in figure 7. Each pipe stage takes one clock cycle to execute, and thus under optimal conditions (no stalls), the DLX issues one instruction per cycle. In practice, including cache stalls, the DLX executes at approximately 1.5 CPI (Clocks Per Instruction).

As is evident from figure 8 (reproduced from figure 6.4 of [9]), ALU instructions do not use the MEM

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

Figure 2: DLX Instruction Encoding: Main Opcodes

	$00	$01	$02	$03	$04	$05	$06	$07
$00	SLLI		SRLI	SRAI	SLL		SRL	SRA
$08					TRAP			
$10	SEQU	SNEU	SLTU	SGTU	SLEU	SGEU		
$18	MULT	MULTU	DIV	DIVU				
$20	ADD	ADDU	SUB	SUBU	AND	OR	XOR	
$28	SEQ	SNE	SLT	SGT	SLE	SGE		
$30	MOVI2S	MOVS2I	MOVF	MOVD	MOVFP2I	MOVI2FP		

Figure 3: DLX Instruction Encoding: Special Opcodes (Main opcode = $00)

Stage	Description
IF	Instruction Fetch
ID	Instruction Decode and operand fetch (RS1 and RS2)
EX	Execute; perform ALU ops and compute displacement addresses
MEM	Memory access
WB	Write Back; write RD back to register file

Figure 7: DLX Pipeline

stage of the pipeline, and branch instructions do not use the WB stage. Thus, if the memory instructions could be shortened, so that they only used 4 pipe stages, then the entire pipeline could be shortened. As it is, one fifth of the processor is idle more than 70% of the time. Also, the additional pipe stage adds a clock cycle to the cost of draining or re-filling the pipeline.

In the NIW, there is not room in an ALU/memory superscalar instruction to express a displacement value for the load/store address, and so displacement addressing has been removed. As such, there is no longer any need for memory instructions to use the EX stage of the pipeline. The NIW instead uses a four stage pipeline, as illustrated in figure 9. The IF and ID stages are similar to the corresponding DLX pipe stages, with the ID stage additionally decoding and fetching the TA and TD registers. The WB stage is similar, as well, with the TD register being written to the register file along with the RD register, as necessary. The major modification is to replace the EX and MEM stage with a combined stage called MEX, which processes the ALU component and the load/store component of a superscalar instruction in parallel. Figure 10 illustrates the usage of the NIW pipeline by the three classes of instructions: immediate (I-type), superscalar ALU-load/store (R-type), and branch (J-type).

	$00	$01	$02	$03	$04	$05	$06	$07
$00	ADDF	SUBF	MULTF	DIVF	ADDD	SUBD	MULTD	DIVD
$08	CVTF2D	CVTF2I	CVTD2F	CVTD2I	CVTI2F	CVTI2D		
$10	EQF	NEF	LTF	GTF	LEF	GEF		
$18	EQD	NED	LTD	GTD	LED	GED		

Figure 4: DLX Instruction Encoding: Floating Point Opcodes (Main opcode = $01)

	$00	$01	$02	$03	$04	$05	$06	$07
$00	ADD	ADDD	ADDI	ADDU	ADDUI	AND	BEQZ	BFPF
$08	BFPT	BNEZ	CVTD2I	CVTI2D	DIV	DIVD	DIVU	EQD
$10	GED	GTD	J	JAL	JALR	JR	LB	LED
$18	LH	LHI	LTD	MOVD	MOVFP2I	MOVI2FP	MOVI2S	MOVS2I
$20	MULT	MULTD	MULTU	NED	OR	RFE	SB	SEQ
$28	SEQ	SEQU	SGE	SGEU	SGT	SGTU	SH	SLE
$30	SLEU	SLL	SLLI	SLT	SLTU	SNE	SNEU	SRA
$38	SRL	SUB	SUBD	SUBU	TRAP	XOR	OP_RES	OP_RES

Figure 6: NIW Opcode Encoding

Stage	Description
IF	Instruction Fetch
ID	Instruction Decode and operand fetch (RS1, RS2, TA and TD)
EX	Execute; perform ALU ops and load/stores
WB	Write Back; write RD and TD back to register file

Figure 9: NIW Pipeline

2.3 Hardware Requirements

Two prominent, slightly superscalar microprocessor currently available include the Intel i860 and the IBM RS/6000. The i860 uses almost 1 million transistors, and the RS/6000 uses more than 5 million transistors over a multi-chip set. In this context, the hardware requirements of the above design are modest. The additional hardware required to add NIW features to a DLX or R2000 like CPU is as follows:

Register File Ports The ID stage requires two additional read ports to fetch the values of TA and TD. The WB stage requires one additional write port to write back the value of TD.

Decoding Logic The opcode field is now densely encoded; almost all values are used, and so a full decoder will be required for this field.

Mitigating this cost is the fact that there are also significant reductions in hardware requirements implied by the NIW architecture:

Pipe Stage The NIW uses a four stage pipeline, instead of five, eliminating the associated inter-stage latches and buffers, and control logic.

Fewer Instructions Since there are fewer actual instructions, there is

Stage	ALU Instruction	Load/Store Instruction	Branch Instruction
IF	$IMAR \leftarrow PC;$ $IR \leftarrow Mem[IMAR];$ $PC \leftarrow PC + 4;$	$IMAR \leftarrow PC;$ $IR \leftarrow Mem[IMAR];$ $PC \leftarrow PC + 4;$	$IMAR \leftarrow PC;$ $IR \leftarrow Mem[IMAR];$ $PC \leftarrow PC + 4;$
ID	$A \leftarrow RS1;$ $B \leftarrow RS2;$	$A \leftarrow RS1;$ $B \leftarrow RS2;$	$A \leftarrow RS1;$ $B \leftarrow RS2;$
EX	$ALUoutput \leftarrow AemopB;$ $bfor$ $ALUoutput \leftarrow Aop$ $((IR_{16})^{16}\#\#IR_{16...31});$	$DMAR \leftarrow A+$ $((IR_{16})^{16}\#\#IR_{16...31});$ $MDR \leftarrow B;$	$ALUoutput \leftarrow PC+$ $((IR_{16})^{16}\#\#IR_{16...31});$ $cond \leftarrow (RS1\,op\,0);$
MEM		$MDR \leftarrow Mem[DMAR];$ **or** $Mem[DMAR] \leftarrow MDR;$	$if(cond)$ $PC \leftarrow ALUoutput;$
WB	$RD \leftarrow ALUoutput;$	$RD \leftarrow ALUoutput;$	

Figure 8: DLX Pipeline Usage

less need for corresponding logic to implement these instructions. The largest savings here will be the reduction in the number of conversion instructions between single precision floating point and the other numeric formats (there are two instead of six).

The combined effect of the above hardware changes should be quite minimal; very little additional silicon, if any, will be required to transform a pipelined RISC processor into an NIW processor.

3 Instruction Scheduling Algorithms

In this section, we describe the basic-block algorithms required to schedule instructions for efficient exploitation of the NIW features. Delayed branch scheduling is assumed, and not addressed, since the NIW does not attempt to add superscalar features to branch instructions, and does not change the number of branch delay slots.

3.1 Barriers to Instruction Movement

Instruction scheduling involves the re-ordering of instructions; an instruction is moved from the place that it was found to some other location in the program. In doing so, we must guarantee that the results of running the program will always be identical to that generated without applying instruction scheduling. To do this, we identify instructions that cannot be moved past one another without risking changing the behaviour of the program.

The first consideration is data hazards[9, page 257]. An instruction cannot be moved *backward* past any instruction that reads its destination register. Similarly, an instruction cannot be moved *forward* past any instruction that writes to one of its source registers.

The second consideration is memory data hazards. Complex compile and run-time analysis *could* be performed to decide in general when it is safe to move memory-referencing instructions past one another, but such scheduling is rarely of any great benefit, and frequently results in an answer of "No, you can't do that." Therefore, we take a simpler approach and simply state the following:

- Loads can be moved past one another.
- Loads and stores may not be moved past one another.
- Stores may not be moved past one another.

The final consideration is branches. Since we cannot decide what path a program will follow without performing extensive trace flow analysis[1], we must

Stage	ALU Immediate Instruction	ALU and Load/Store Instruction	Branch Instruction
IF	$IMAR \leftarrow PC;$ $IR \leftarrow Mem[IMAR];$ $PC \leftarrow PC + 4;$	$IMAR \leftarrow PC;$ $IR \leftarrow Mem[IMAR];$ $PC \leftarrow PC + 4;$	$IMAR \leftarrow PC;$ $IR \leftarrow Mem[IMAR];$ $PC \leftarrow PC + 4;$
ID	$A \leftarrow RS1;$ $B \leftarrow RS2;$ $DMAR \leftarrow TA;$ $MDR \leftarrow TD;$	$A \leftarrow RS1;$ $B \leftarrow RS2;$ $DMAR \leftarrow TA;$ $MDR \leftarrow TD;$	$A \leftarrow RS1;$ $B \leftarrow RS2;$ $DMAR \leftarrow TA;$ $MDR \leftarrow TD;$
MEX	$ALUoutput \leftarrow A op$ $((IR_{16})^{16}\#\#IR_{16...31});$	$ALUoutput \leftarrow A op B;$ **and** $(MDR \leftarrow Mem[DMAR];$ **or** $Mem[DMAR] \leftarrow MDR;)$	$ALUoutput \leftarrow PC+$ $((IR_{16})^{16}\#\#IR_{16...31});$ $cond \leftarrow (RS1 op 0);$
WB	$RD \leftarrow ALUoutput;$	$RD \leftarrow ALUoutput;$ $TD \leftarrow MDR$	$if(cond)$ $PC \leftarrow ALUoutput;$

Figure 10: NIW Pipeline Usage

regard branches and branch targets (labels) as barriers to instruction movement. We can simplify this by regarding labels as *writing* to all registers, and branches as *reading* all registers (since, in effect, the code on the other side of these points can actually do that). Thus the range of possible motion of an instruction is ultimately limited to the length of its basic block. In section 3.4 we discuss more aggressive algorithms that could move an instruction outside of its basic block, but that have a higher cost in compile time and compiler complexity.

3.2 Where to Move Instructions

Now that we have decided on the limits to instruction movement, we must decide *how* to order our instructions. Clearly, trying all possible sequences of instructions is unsatisfactory, since this is exponential in the length of the basic block, and most such sequences will be rejected as meaningless anyway.

Because the NIW can execute an ALU instruction and a data transfer instruction (load or store) in parallel, it can be thought of as a software-visible pipeline. To get the most efficient use of this pipeline, we want to separate the instruction that *produces* a value as much as possible from the first instruction that *consumes* that value. We also want to bind complimentary (ALU and load/store) instructions together. The following rudimentary algorithm performs this task for basic blocks:

1. Take one pass over the block marking places where values are produced into registers, and consumed from registers. Identify a value by its register number and an increasing sequence number (i.e. the third value to be written into R7 would be 7.3).

2. In a second pass, examine each instruction in turn: if it is already in a complimentary slot, skip it. If it is not, search backward and forward within the allowable range for a suitable complementary slot.

The above algorithm is rudimentary, and far from optimal. A more thorough investigation of scheduling algorithms will be developed in [7].

3.3 Simulated Displacement Addressing: Data Structures

The DLX supports a register indirect with displacement addressing mode, with 16-bit displacements. This addressing mode is extensively used by compilers to access members of a data structure, or local variables, by pointing a register at the first location of a data structure, and using displacement to refer to the members. Since displacement addressing has been removed from the NIW, it is important to simulate this capability efficiently.

To access the members of a data structure naively, we would do a series of immediate additions to the base register to explicitly compute the individual addresses. Figure 11 illustrates the DLX code to save a

```
;; Adjust Stack Pointer
add r14,r14,#-40
;; Save Registers
sw 0(r14),r3
sw 4(r14),r4
sw 8(r14),r5
sw 12(r14),r6
sw 16(r14),r7
sw 20(r14),r8
sw 24(r14),r9
sw 28(r14),r10
```

Figure 11: DLX Register Save

```
;; Adjust Stack Pointer
add r14,r14,#-40
;; Save Registers
sw (r14),r3
add r17,r14,#4
sw (r17),r4
add r17,r14,#8
sw (r17),r5
add r17,r14,#12
sw (r17),r6
add r17,r14,#16
sw (r17),r7
add r17,r14,#20
sw (r17),r8
add r17,r14,#24
sw (r17),r9
add r17,r14,#28
sw (r17),r10
```

Figure 12: NIW Naive Register Save

number of active registers at the beginning of a function. Figure 12 illustrates a naive translation for the NIW.

The naive code is now 77% larger and slower. To fix this, we pipeline the access. Noticing that the addresses to be referenced are all 4 bytes apart, we change the add instructions to an induction series, with a step size of 4. We can now compute the address of the following `sw` instruction in parallel with the current one. Figure 13 illustrates the resultant code. A `%` symbol is used to delimit two instructions

```
;; Adjust Stack Pointer
add r14,r14,#-40
;; Save Registers
add r25,r14,r27 % sw (r14),r3
add r24,r25,r27 % sw (r25),r4
add r25,r24,r27 % sw (r24),r5
add r24,r25,r27 % sw (r25),r6
add r25,r24,r27 % sw (r24),r7
add r24,r25,r27 % sw (r25),r8
add r25,r24,r27 % sw (r24),r9
sw (r25),r10
```

Figure 13: NIW Optimized Register Save

bound into the same instruction word.

Notice that the sequence of `add` and `sw` instructions are alternating between registers `r25` and `r24`, so as not to collide with one another. Only two registers are needed here, since the effective pipeline depth is only two. *Register Forwarding*, as explained in [9, page 261] allows registers to be modified in one instruction and used in the next, despite the 4-stage pipeline.

3.4 Non-basic-block Optimizations: Speculative Computation

The NIW architecture increases the amount of computation that can be performed in a given time through parallelism. However, this parallelism cannot be exploited as frequently as we would like, largely because of data hazards as described in section 3.1. With considerable cost in compile time and compiler complexity, we can ease the restrictions on instruction movement relating to branches and branch targets.

Because we cannot know, in general, which path through the code a program will follow, when we move an instruction outside of a basic block, we must ensure that the instruction will always be executed whenever the current basic block is reached, and that it has no effect if the current block is not reached. Since the instruction may execute even if the block it is from is not executed, we call this *speculative computation*.

The speculative computation extension to the scheduling algorithm is essentially to consider pushing instructions back through branch targets (labels) or forward through (possibly conditional) branches. In doing so, we must assure the following:

- When an instruction is pushed back through a label, it must be placed in *all* possible paths to that label.
- When an instruction is pushed forward through a conditional branch, it must be placed in *both* the taken and un-taken paths.
- All of the multiple locations that an instruction is pushed into must succeed. If it is not possible to place an instruction one one of the paths because of a data hazard or lack of slots, then the instruction must not be pushed through the branch or label.

As is evident from the above, such scheduling can become very expensive. The cost of scheduling a single instruction in this way is multiplied by the number of paths that must be considered. Development time allowing, experiments may be conducted on the benefits of attempting such a scheduling approach.

4 Analysis

The only truly valid test of the value of an architectural enhancement is an empirical measurement of a large number of real programs. However, this is difficult to do with the compiler still under development, and so instead we present some microscopic analysis of compiler-generated DLX programs, and hand-translated NIW equivalents.

4.1 A Random Number Generator

The first example is a random number generator taken from [3], and used as a compute-intensive load for comparing processor performance in [8]. Figure 14 illustrates the C source for this program. The compute-intensive portion of this program is the function `g()`, and so figure 15 presents the DLX assembler output from the compiler for `g()`, without the function prologue and epilogue. The comments of the form `;; COUNT 11` indicate the number of instructions in the following basic block.

Figure 16 illustrates a translation of the `g()` function for the NIW, with assorted instruction scheduling techniques applied. No instructions have been scheduled outside of a basic block. The first block has been reduced from 11 instructions to 10 by simply removing a redundant computation with a reference to `r29`, which by NIW convention is always set to the value 1, and `r28`, which is set to the value 2. Two instructions are then trivially folded together. The second block is unaffected by NIW features.

```
#define N       5000
int     x[N];
main()
{
    int     i;
    for(i=5; i--; )
        g();
    printf("%d\n%d\n", x[N-2], x[N-1]);
}

g()
{
    int     i;
    x[0] = x[1] = 1;
    for(i=2; i<N; i++)
        x[i] = x[i-x[i-1]] + x[i-x[i-2]];
}
```

Figure 14: A Compute Intensive Random Number Generator

The third block has been reduced from 21 to 17 instructions, yielding an improvement of 23%. Immediate operands of value `#2` have been replaced with references to register `r28` to permit them to be encoded with load and store instructions. Four load and store instructions have been folded together with ALU instructions, sometimes accompanied by slight re-ordering of neighbouring ALU instructions to avoid data hazards.

A second program (quicksort) was translated and analyzed. While it was extensively changed, the net impact on performance was neutral. This effect is primarily due to the very small size of basic blocks in the quicksort program. While the quicksort program is compute intensive, its basic blocks are artificially short, and not representative of typical C programs.

This project is considered to be under development, with most of the working software required to demonstrate performance improvements not yet complete. It is our opinion that when the compiler and instruction scheduler are complete, they will demonstrate a performance improvement of 10% to 30%, depending on the program.

```
        ;; COUNT 11
        lhi r3,(_x>>16)&0xffff
        addui r3,r3,(_x&0xffff)
        addi r10,r0,#1
        sw 4(r3),r10
        addi r10,r0,#1
        sw 0(r3),r10
        addi r6,r0,#2
        addi r3,r0,#4999
        sgt     r1,r6,r3
        bnez    r1,L12
        nop
        ;; COUNT 4
        lhi r7,(_x>>16)&0xffff
        addui r7,r7,(_x&0xffff)
        add r9,r7,#-4
        add r8,r7,#-8
L11:    ;; COUNT 21
        sll r4,r6,#2
        add r5,r7,r4
        add r3,r9,r4
        lw r3,0(r3)
        sub r3,r6,r3
        sll r3,r3,#2
        add r3,r7,r3
        add r4,r8,r4
        lw r4,0(r4)
        sub r4,r6,r4
        sll r4,r4,#2
        add r4,r7,r4
        lw r3,0(r3)
        lw r4,0(r4)
        add r3,r3,r4
        sw 0(r5),r3
        add r6,r6,#1
        addi r3,r0,#4999
        sle     r1,r6,r3
        bnez    r1,L11
        nop
```

Figure 15: DLX assembler for `g()`

```
        ;; COUNT 10
        lhi r3,(_x>>16)&0xffff
        addui r3,r3,(_x&0xffff)
        addi r10,r0,r29
        add r25,r3,r27
        sw (r25),r10
        add r6,r0,r28   % sw (r3),r10
        addi r3,r0,#4999
        sgt     r1,r6,r3
        bnez    r1,L12
        nop
        ;; COUNT 4
        lhi r7,(_x>>16)&0xffff
        addui r7,r7,(_x&0xffff)
        add r9,r7,r26
        add r8,r7,#-8
L11:    ;; COUNT 17
        sll r4,r6,r28
        add r3,r9,r4
        add r5,r7,r4    % lw r3,0(r3)
        sub r3,r6,r3
        sll r3,r3,r28
        add r4,r8,r4
        add r3,r7,r3    % lw r4,0(r4)
        sub r4,r6,r4
        sll r4,r4,r28
        add r4,r7,r4    % lw r3,0(r3)
        lw r4,0(r4)
        add r3,r3,r4
        sw 0(r5),r3     % add r6,r6,r29
        addi r3,r0,#4999
        sle     r1,r6,r3
        bnez    r1,L11
        nop
```

Figure 16: NIW scheduled assembler for `g()`

5 Ongoing Research

While the DLX architecture has not actually been implemented, there does exist a cross-compiler (a port of the GNU gcc compiler[11]) and simulator (called dlxsim[4]). We have installed (and debugged) this system, and added debugging enhancements and delayed branch scheduling.

To test the concepts of the NIW architecture, we have created a modified version of the DLX simulator to accept only those instructions supported by the NIW. Future development includes:

- Restrict the NIW simulator to register indirect addressing.
- Modify the gcc DLX cross-compiler to generate

only NIW instructions and addressing.

- Develop an instruction scheduler that will efficiently exploit the NIW features.

The last item above is to be the subject of a Master's thesis[7]. Once all of the above tools have been developed, real programs will be compiled and run, and the performance results compared to those obtained for the DLX architecture.

References

[1] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, 1985.

[2] R. Garner, A. Agarwal, F. Briggs, E. Brown, D. Hough, B. Joy, S. Kleiman, S. Munchnik, M. Namjoo, D. Patterson, J. Pendleton, and R. Tuck. Scaleable Processor Architecture (SPARC). In *COMPCON, IEEE*, San Francisco, California, 1988.

[3] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid.* Basic Books, 1979.

[4] Larry B. Hostetler and Brian Mirtich. DLXsim–A Simulator for DLX. DLX Documentation, August 1990.

[5] Norman P. Jouppi and David W. Wall. Avaliable Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, Boston, MA, April 1989.

[6] G. Kane. *MIPS R2000 RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[7] David Kinchlea. [approx.] Instruction Scheduling for the NIW Architecture. Master's thesis, University of Western Ontario, 1991. Under development.

[8] Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the Symposium on Operating Systems Principles*, 1989.

[9] David A. Patterson and John L. Hennessy. *Computer Arcitecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, first edition, 1990.

[10] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on Multiple Instruction Issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, Boston, MA, April 1989.

[11] Richard M. Stallman. *Using and Porting GNU C.* Free Software Foundation, Inc., Cambridge, MA.