PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

A WIDE INSTRUCTION WORD ARCHITECTURE FOR PARALLEL EXECUTION OF LOGIC PROGRAMS CODED IN BSL

Kemal Ebcioğlu and Manoj Kumar

IBM Research Division
Thomas J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.

Abstract

This paper begins by describing BSL, a new logic programming language fundamentally different from Prolog. BSL is a nondeterministic Algol-class language whose programs have a natural translation to first order logic; executing a BSL program without free variables amounts to proving the corresponding first order sentence. A new approach is proposed for parallel execution of logic programs coded in BSL, that relies on advanced compilation techniques for extracting fine grain parallelism from sequential code. We describe a new 'Very Long Instruction Word' (VLIW) architecture for parallel execution of BSL programs. The architecture, now being designed at the IBM Thomas J. Watson Research Center, avoids the synchronization and communication delays (normally associated with parallel execution of logic programs on multiprocessors), by determining data dependences between operations at compile time, and by coupling the processing elements very tightly, via a single central shared register file. A simulator for the architecture has been implemented and some simulation results are reported in the paper, which are encouraging.

1. Introduction and motivations

Logic programming is definitely a desirable means to implement large artificial intelligence (A.I.) applications. The logic programming framework provides a well understood relationship between the actual execution of the rules of an expert system, and the declarative meaning of the same rules (this relationship is not present in many popular expert system tools; for example, in the OPS family of production systems [Forgy and McDermott 77], there is no natural formalism that clarifies the declarative meaning of a rule). Also, since the knowledge base of an expert system implemented with a logic programming language consists entirely of logical assertions, there is an increased opportunity for imposing a good formal organization on the knowledge representation. But we feel that logic programming research should not confine itself to the versatile but narrow Prolog-and-variants paradigm to achieve such important benefits of logic programming.

This paper will begin by describing the Backtracking Specification Language (BSL) [Ebcioğlu 87b], which is fundamentally different from Prolog, but which nevertheless allows the benefits of programming with the concepts of first order logic including universal and existential quantifiers, and which is capable of achieving very efficient execution. BSL has been used for implementing a 350-rule expert system for harmonizing chorales in the style of J.S. Bach [Ebcioğlu 87a, 88b, 88c]. BSL is an Algol-class nondeterministic language whose programs have a natural translation to first-order logic: executing a BSL program without free variables amounts to proving the corresponding first order sentence. BSL compiles into efficient backtracking programs in C, and has the capability of taking direct advantage of the native instruction set of RISC and mainframe architectures, and state-of-the-art compiler optimizations such as common expression elimination, code motion, strength reduction, induction variable elimination, and register allocation. As for the parallel execution of BSL, the fact that BSL's backtracking semantics is defined sequentially does not constitute an impediment to parallelism, in the view of recent advances in compilation techniques for extracting parallelism from ordinary sequential code. Also, BSL has a single assignment feature (originally intended to preserve logical soundness) that sets it aside from ordinary sequential languages and increases the parallelism that can be extracted from BSL programs via such compilation techniques.

After the initial exposition about BSL, we discuss how BSL programs can be executed efficiently using a recent approach to fine-grain parallelism, namely the Very Long Instruction Word (VLIW) architecture and compilation techniques for these architectures [Fisher 79, Ellis 86, Nicolau 85, Ebcioğlu 87c]. A new VLIW architecture and machine organization is proposed, which takes advantage of the recent advances in VLSI technology. In this machine the multiple ALUs are coupled by shared register file, which provides very low latency sharing of intermediate results, and is therefore suitable for exploiting fine grained parallelism. Unlike J. Fisher's previous approach, our VLIW architecture does not rely on optimizing a single "most probable" path through the code, it has the features to perform well on code with unpredictable conditional branches, which are common in A.I. software. A novel mechanism that initiates execution of operations in conditional sections before the enclosing condition is known, accepts/discards the results of these operations based on the condition, and allows multiple way branching base on multiple condition codes, provides the architectural support to exploit parallelism beyond basic block boundaries. A compiler is being developed for this machine, and performance measurements obtained from a register-level simulator have yielded encouraging results.

In Section 2 of this paper we describe the BSL language and briefly indicate how BSL programs are translated into C pro-

grams. The VLIW machine architecture and hardware organization being proposed by us is described in Section 3. The process of translating BSL programs into VLIW machine code is outlined in Section 4, and in Section 5 we discuss the performance of the VLIW machine on A.I. type application kernels, based on simulation results. Finally, in Section 6 we discuss our conclusions and the status of the project.

2. An overview of BSL

BSL (Backtracking Specification Language) is an Algol-class nondeterministic language where more than one explicit assignment to a variable is forbidden. BSL has a Lisp-like syntax and is compiled into C via a Lisp program. We have provided BSL with formal semantics, in a style inspired from [de Bakker 79], and [Harel 79]. The semantics of a BSL program F is defined via a ternary relation Ψ, such that $\Psi(F, \sigma, \sigma')$ means program F leads to final state σ' when started in initial state σ, where a state is a mapping from variable names to elements of a "computer" universe, consisting of integers, arrays, records, and other ancillary objects. What makes BSL different from ordinary nondeterministic languages [Floyd 67, Smith and Enea 73, Cohen 79], and relates it to logic, is that there is a simple mapping that translates a BSL program to a formula of a first-order language, such that *if* a BSL program terminates in some state σ, *then* the corresponding first order formula is true in σ (where the truth of a formula in a given state σ is evaluated in a fixed "computer" interpretation after replacing any free variables x in the formula by $\sigma(x)$.) A BSL program is very similar in appearance to the corresponding first order formula, and for this reason, we call BSL programs formulas. A formal description of BSL, and a proof of its soundness can be found in [Ebcioglu 87a]. The description of the language in this paper will be informal.

Here is an example of a simple BSL program to solve a classic puzzle, followed by its first order translation: Place 8 queens on a chess board, so that no queen takes another (i.e. no two queens are on the same row, column or diagonal). Assume that the rows and columns are numbered from 0 to 7, and that the array elements p[0], ... p[7] represent the column number of the queen in row 0,...,7, respectively.

```
(E ((p (array (8) integer)))
     (A n 0 (< n 8) (1+ n)
          (E j 0 (< j 8) (1+ j)
               (and (A k (1- n) (>= k 0) (1- k)
                         (and (!= j (p k))
                              (!= (- j (p k)) (- n k))
                              (!= (- (p k) j) (- n k))))
                    (:= (p n) j)))))
```

First-order translation:

$$(\exists p \mid type(p)=\text{"(array (8) integer)"})\ (\forall n \mid 0\le n<8)\ (\exists j \mid 0\le j<8)\ [(\forall k \mid n-1\ge k\ge 0)\ [j\neq p[k]\ \&\ j-p[k]\neq n-k\ \&\ p[k]-j\neq n-k]\ \&\ p[n]=j]$$

Because of the similarity between a BSL formula and its logical counterpart, a BSL formula is like a specification for its own self: it describes what it computes. As the reader can readily see, the BSL formula shown above specifies what a solution to the eight queens problem should satisfy, assuming we read an assignment symbol as equality, and we translate the quantifiers to a conventional notation. This BSL formula compiles into an efficient backtracking program in C that finds and prints instantiations for the array p, that would make the ($\exists$p)-quantified part of the corresponding first order formula true in the fixed interpretation. The BSL compiler presently runs on Lisp/VM and IBM 3090 computers, and generates code acceptable by the C version of the PL.8 compiler [Warren et al. 86], and also the AT&T C compiler.

We can observe some examples of BSL language features in this 8-queens program: The basic building blocks of BSL are *constants*, that consist of integers, such as -2, 0, 3, and record tags, which are identifiers such as ssn, salary; and *variables*, which are identifiers such as x, p, n, or emp (for convenience, we assume that variables are distinct from record tags). Each variable and constant is a BSL *term*, and if t_1 and t_2 are BSL terms, and *binop* is one of the binary operators +,-,*,/,sub, and dot, then (*binop* t_1 t_2) is also a BSL term (sub and dot are intended to be the subscript and field extraction operators, respectively). Examples of BSL terms are 0, (+ x 2), or (* 2 (dot emp salary)). The constructs (1+ x), (1- x) may be used as abbreviations for (+ x 1) and (- x 1), respectively. A BSL *lvalue* is either a variable, or a term of the form (f_1 ... (f_{n-1} (f_n x ...) ...) ...) where each of f_1,... f_n is either sub or dot, and where x is a variable. Lvalues are terms that can appear as the left-hand operand of an assignment, and are exemplified by x, (dot emp salary), or (sub p n). Lvalues can also be abbreviated as long as their normal notation can be inferred from context, for example the latter two lvalues can be written as (salary emp), and (p n), in the proper contexts. A BSL *atomic formula* is either an *assignment* of the form (:= l t_1), or a *test* of the form (*relop* t_1 t_2), where l is an lvalue, t_1, t_2 are terms, and *relop* is one of == (equal), != (not equal), <, >=, <=, or >. A BSL atomic formula is a BSL *formula*. Assuming F_1 and F_2 are BSL formulas, then so are the following: (and F_1 F_2), (or F_1 F_2),[1] (A x *init cond incr* F_1), (E x *init cond incr* F_1), and (E ($(x\ typ)$) F_1), where x is a variable, *init* , *incr* are terms where *init* does not contain x, and *cond* is a BSL formula not containing any occurrences of A, E, or :=, and *typ* is type. The BSL types are similar to the type declarations of an Algol-class language, and allow integer, array and record declarations. Examples of BSL types are integer, (array (3) integer), and (record (ssn integer) (salary integer)). In general, "integer" is a BSL *type*, and if *typ*, typ_1,..., typ_k are BSL

[1] In the eight queens program above the construct (and F_1 F_2 F_3) abbreviates (and F_1 (and F_2 F_3)). In general, "and" and "or" associate to the right, and thus (and ...) and (or ...) can contain more than two subformulas.

types, $k \geq 1$, and $y_1,...,y_k$ are distinct record tags, and n is a positive integer, then (array (n) *typ*), and (record (y_1 typ_1) ... (y_k typ_k)) are BSL types.

We give here an informal description of the nondeterministic program semantics of BSL: The variables of BSL can range over objects, each of which has a corresponding type. Objects of type integer are constants such as -2, 0, 3, and U (called the *unassigned* constant). An object can also be an array, which is a list of objects of the same type, or a record, which is a list of alternating record tags and objects, not necessarily of the same type. Arrays and records are exemplified by (1 2 U), which is an object of type (array (3) integer), and (ssn 999123456 salary 25000), which is an object of type (record (ssn integer) (salary integer)). The value of a BSL term, in a particular state during execution, is computed by using the usual meanings of the binary operators +,-,*,/,sub, and dot. sub is defined to be the subscript operator which takes an array object and an integer i and returns the i'th element of the array object (the array elements are numbered starting from 0); and dot is defined to be an operator that extracts a subobject of a given record object as determined by a given record tag (it performs a function similar to the dot within the expression "employee.salary" in PL/I). BSL atomic formulas, i.e. assignments and tests, are executed in the conventional manner: the tests are executed by performing the indicated comparison operation after computing the current values of the two terms to be compared; and the assignments are executed by computing the current value of the right hand side term, and then destructively changing the value of the left hand side term to reflect the current value of the right hand side term. If the comparison operation indicated in a test comes out to be true, the effect of the test is a no-op. However, if a test does not come out to be true, or if an assignment is attempted when the current value of the left hand side is not U, or when the current value of the right hand side is not an integer, or if an attempt is made to perform an illegal computation (such as using a variable whose value is U in an arithmetic operation or comparison, or dividing by zero), execution does not terminate. (and F_1 F_2) is executed by first executing F_1, then F_2. (or F_1 F_2) is executed by executing one of F_1 or F_2. (A x *init cond incr* F_1) is similar to the C "for" loop, it is executed by saving the old value of x, setting x to *init*, while *cond* is true repetitively executing F_1 and setting x to *incr*, and restoring the old value of x if and when *cond* is finally false. (E x *init cond incr* F_1) is executed by saving the old value of x, setting x to *init*, setting x to *incr* an arbitrary number of times (possibly zero times), and finally deciding not to set x to *incr* any more, executing F_1, and then restoring the old value of x. *cond* must be true after x is set to *init* and after each time x is set to *incr*, or else execution does not terminate. (E ((x *typ*)) F_1) is similar to a "begin-end" block with a local variable, it is executed by saving the old value of x, setting x to an object of type *typ* all of whose scalar (i.e. integer) subobjects have the value U, executing F_1, and then restoring the old value of x.

The translation of a BSL program to the first order assertion that is true at any of its termination states, is for the most part obvious, as exemplified by the 8-queens program above; however, both the assignment symbol (:=) and the equality test (==) of BSL get translated to the equality symbol in the logical counterpart, that is, the program contains *procedural* information not present in its logical counterpart. First, assume that F' denotes the first order translation of a BSL term, formula or operator F. The translation of BSL terms to first order logic is straightforward: for example (+ x 2), (dot (sub emp i) salary), translate into +(x,2), dot(sub(emp,i),salary) (which can also be abbreviated as x+2, emp[i].salary). The first order translation of the comparison operators <, >=, <=, >, ==, != are the predicate symbols $<, \geq, \leq, >, =, \neq$, respectively. Tests such as (*relop* t_1 t_2), and assignments such as (:= l t_1) translate into the first order atomic formulas t'_1 *relop*' t'_2, and $l' = t'_1$, respectively. (and F_1 F_2) translates into $[F'_1 \,\&\, F'_2]$, (or F_1 F_2) translates into $[F'_1 \vee F'_2]$, and (E ((x *typ*)) F_1) translates into $(\exists x \mid \text{type}(x)=\text{“}typ\text{”})[F'_1]$ (which can be abbreviated as $(\exists x{:}typ)[F'_1]$). For a simple subset of BSL, where the only allowable looping constructs are of the form (A x t_1 (< x t_2) (1+ x) F), (E x t_1 (< x t_2) (1+ x) F), and variants thereof, the translation of these to bounded quantifiers of first order logic, namely $(\forall x \mid t'_1 \leq x < t'_2)[F']$, $(\exists x \mid t'_1 \leq x < t'_2)[F']$,... works; where t'_1, t'_2 are the first-order translations of BSL terms t_1 and t_2, respectively, and where x does not occur in either t_1 or t_2. However, for the general case involving arbitrary *cond* and *incr* expressions, which we will not elaborate here, the rigorous translation of BSL formulas involves associating a different function symbol of the first order language with every quantified formula of BSL, and is less natural.[2]

The following translation examples should demonstrate the intuition behind the relationship of a BSL program to its first-order translation: When either (:= x 0) is successfully executed (i.e. x is initially U), or (== x 0) is successfully executed (i.e. x is initially 0), the assertion x=0 is true at the termination state. When (or (== x 0) (== x 1)) is successfully executed, (i.e. x is initially 0 or 1, and the proper subformula of the "or" is chosen for execution), the assertion [x=0 ∨ x=1] is true at the termination state. When

```
(A i 0 (< i 10) (1+ i) (E ((j integer))
                  (and (or (:= j 0) (:= j 1)) (:= (sub a i) j))))
```

is successfully executed (i.e. "a" is initially an array object whose first ten elements are U),

$$(\forall i \mid 0 \leq i < 10)(\exists j \mid \text{type}(j)=\text{“integer”})[[j=0 \vee j=1] \,\&\, a[i]=j]$$

is true in the termination state. This assertion says that the first 10 elements of "a" are an arbitrary sequence of 0's and 1's. To see why this assertion is true at the termination state of the program, observe that during the execution of the program, for each i=0,...,9, the assertion $(\exists j \mid \text{type}(j)=\text{“integer”})[[j=0 \vee j=1] \,\&\, a[i]=j]$ was made true, by creating (for each i) an integer j equal to 0 or 1, and then making a[i]=j true by assigning j to a[i]. The first order translation of (and (:= x 0) (:= x (1+ x))) is [x=0 & x=x+1], but such a BSL formula can never reach a termi-

[2] See [Ebcioğlu 87a] for details. In practice, the general case is rarely needed, because BSL programs are often first conceived as first order assertions, rather than, say, while loops.

nation state, no matter what the initial value of x is, because it violates the single assignment rule enforced by the program semantics of BSL (the single assignment rule is the one that verifies that the left hand side is U, and the right hand side is an integer, before each explicit assignment). The intuitive purpose of the single assignment rule is to ensure that the continuation of execution does not destroy the truth of the assertions that were previously made true. Top-level formulas (i.e. complete programs) of the BSL subset we are describing, such as the 8-queens program given above, do not contain free variables, so the truth of the assertions corresponding to such formulas is not affected by the value of any variable in the termination state. Successfully executing such a top-level BSL formula is equivalent to constructively proving that the corresponding first-order sentence is true in a fixed interpretation that involves integers, arrays, records, and operations on such objects (or in all models of a suitably axiomatized "theory of integers, arrays, and records").

A BSL program of the form (E ((*x typ*)) *F*) is implemented on a real, deterministic computer via a modified backtracking method, which *in principle* attempts to simulate all possible executions of the BSL program, and prints out the value of *x* just before the end of every execution that turns out to be successful. Whenever a choice has to be made between executing F_1 and executing F_2 in the context (or F_1 F_2), the current state is pushed down to enable restarting by executing F_2 , and F_1 is executed. Whenever a choice has to be made between executing *F* and setting *n* to *incr* in the context (E *n init cond incr F*), the current state is pushed down to enable restarting by setting *n* to *incr*, and *F* is executed (pushing down all variables in the current state would be an inefficient implementation of this mechanism; in practice, only a few variables need to be pushed down, by virtue of a compiler optimization to be described below.) Whenever a test (*relop* t_1 t_2) is found to be false, or if *cond* is found to be false in the context (E *n init cond incr F*), and each time after the top level (E ((*x typ*)) ...) is successfully executed and *x* is printed, the state that existed at the most recent choice point is popped from the stack, and execution restarts at that choice point. Attempting to make more than one explicit assignment to a scalar variable or to a scalar subpart of an aggregate variable, and illegal computations (such as attempting to add a number to a variable whose value is U) are considered errors and should never occur during the backtracking execution of a correct BSL program (however, the run time checks for detecting such errors can be omitted for efficiency reasons). Execution begins with an empty choice-point stack and ends when an attempt is made to pop something from an empty stack.

A modification is made to this basic backtracking technique for the case of assignment-free formulas F_1 in the context (or F_1 F_2), or (E *n* ... F_1). After a formula F_1 in such a context is successfully executed, the most recent choice point on the stack is discarded (which would be the choice point for restarting at F_2, or F_1 with a different value of *n*, assuming the modification is uniformly applied). This convention, similar to the "cut" operation of Prolog, serves to prevent duplicate solutions for *x* from being printed out (or redundant failures from occurring) when F_1 and F_2 do not express mutually exclusive conditions, or when F_1 is true for more than one *n* in its quantifier range. Here is an example that demonstrates the motivation behind this modification to backtracking: suppose that many elements of an array "a" are equal to 0 in a particular state during backtracking execution; if in this state, an assignment free subformula (E i 0 (< i N) (1+ i) (== (a i) 0)) is executed and succeeds after finding that for a particular i, a[i]=0, and immediately thereafter a failure occurs (or some solution is printed), there is no point in backtracking to the point in the subformula where (== (a i) 0) is re-executed with the next higher value of i, and then succeeding again after finding another element of "a" that is equal to 0, because the program will then fail in exactly the same way as before (or will print the same solution that it printed before). So the choice point for backtracking to (== (a i) 0) with the next value of i, is discarded when (E i ...) succeeds.

The BSL compiler attempts to produce extremely efficient C code, rather than to implement the above semantics literally. For efficiency, the run-time checks for single assignment are omitted in the present implementation (i.e. variables are not initialized to U upon creation, and the left hand side is not checked for U before assignments. Some simple coding conventions may be used to help to ensure that the program is in fact correct in the sense that a variable is not assigned more than once.) The single assignment nature of BSL allows a substantial optimization in backtracking. Most variables are not pushed down at choice points or later restored, for the following reason: if the variable is already assigned, then it will not have been assigned again and its storage space will have remained intact when a backtracking return is made to this choice point; otherwise, if the variable is not yet assigned, then its old value (conceptually U) will not be used after backtracing is made to this choice point (i.e., it is OK if after backtracking the variable contains garbage resulting from assignments in the paths that failed). In either case, there is no need to save and restore the variable.[3] This approach tends to have less overhead than the technique of pushing and restoring the variables on a special "trail" stack as in Prolog implementations [Turk 86, Fagin and Dobry 85]. Similarly, for the case of subformulas F_1 in the context (or F_1 F_2), or (E *n* ... F_1), the pushdown of the variables (to backtrack to F_2 or to F_1 with the next value of *n*) is not done *before* executing F_1, as the naive semantics requires; the pushdown of the choice

[3] In the present implementation, where variables are allocated in registers or static storage for the BSL subset we are describing, the variables that need to be pushed down at a point of nondeterministic choice are precisely those that are both declared in quantifiers enclosing the choice point, and that are also enclosed in a universal quantifier. Such variables typically consist of quantifier indices. For example, when the nondeterministic choice is being made between executing (and (A k ...) ...) and incrementing j in (E j ... (and (A k ...) ...)) in the eight queens program above, only n and j (but not the elements of p), need to be pushed down, in order to later backtrack to the point where j is incremented and (and (A k ...) ...) is executed with the next value of j. n needs to be pushed down since it will be incremented during the continuation of execution, and j needs to be pushed down since the storage (register) allocated to it will be re-used for a new j during the next iteration of the enclosing universal quantifier (A n ...). But for any i, p[i] does not need to be pushed down, since if p[i] is already assigned now, then it will not have changed when backtracking occurs and its storage space will have remained intact; and if p[i] is not yet assigned now, we will not care about what it contains after backtracking occurs. The required pushdown and restore operations are always compiled inline.

point is delayed as long as possible, by emitting compare and branches, and having the code for the initial part of F_1 branch directly to the next alternative when it is found that F_1 fails.[4] If F_1 is assignment free, no pushdowns are generated at all. The motivation of this optimization is that F_1 may fail and branch directly to the next alternative before a choice point needs to be pushed down.

The language subset described up to here is called L^*, and constitutes the "pure" subset of BSL. The full language has some more, but not many more features; we tried to keep BSL small. These features are mainly user-defined (possibly recursively defined) predicates that can syntactically replace <, >, etc., and that allow Prolog-style backward chaining; user-defined functions that can syntactically replace +,-,...; enumeration types; and macro and constant definitions that allow access to the full procedural capabilities of Lisp. BSL predicates are similar to Prolog procedures, but whether a parameter is an input to the predicate (i.e. is *used* by the predicate body), or is an output (i.e. is *assigned to* by the predicate body) is determined at program writing time. A limited but conceptually very useful form of the "not" connective is defined as a macro, which is expanded by moving the "not"s in front of the tests via de Morgan-like transformations, and then eliminating the "not"s by changing == to !=, etc.. The language is also extended with *heuristics*, which are BSL formulas themselves, which can guide the backtracking search in order to enumerate the better solutions first. A compiler optimization is used to implement intelligent backtracking with low overhead. The language and its compilation techniques are fully described in [Ebcioğlu 87a].

3. A wide instruction word architecture for executing BSL

This section describes the Very Long Instruction Word (VLIW) architecture being proposed by us, to execute logic programs written in BSL and compiled into VLIW machine language. First we will informally describe the logical structure of the VLIW instructions and define the machine state transition produced by these instructions (i.e. the operational semantics). Next we will define the encoding of these instructions into fixed length bit strings, and we will give a high level view of the VLIW machine organization.

Structure of VLIW instructions

Each VLIW instruction is a binary tree like structure (rooted and oriented). A unique instruction label, which is also the address of the instruction in the instruction memory, is associated with each instruction. Each leaf node in the instruction tree specifies the address of an instruction which can succeed this instruction. The internal nodes of the instruction tree specify either an ALU operation or test on some condition code. Internal nodes that specify ALU/memory operation have precisely one descendent, while internal nodes that specify a test on some condition code have precisely two descendents.

The proposed architecture is not limited to executing just BSL programs, it can also execute any algorithm written in a high level language such as C. To describe the operational semantics of the architecture we will use such a traditional algorithm as an example. The inner loop of the merge algorithm shown below can be coded in 3 VLIW instructions as shown in Figure 1.

```
merge ( a, b, c, n)
int a[], b[], c[], n ;
{
    int i, j, k ;
    i = 0 ;  j = 0 ;
    for ( k=0 ; k<2*n ; k++ ) {
        if ( i>=n | | j<n && a[i]>b[j] )
            { c[k] = b[j++] ; }
        else
            { c[k] = a[i++] ; }
    }
}
```

Here, the label of each instruction is shown right above the root node of the instruction tree. The leaf nodes, which specify the successor instructions, are shown as squares. The ALU/memory operations are specified as circles, and the tests on condition codes are specified as ellipses. The instruction trees can also be encoded in a straightforward way as Lisp lists (this is how the compiler views them), and under Figure 1 we give the list representation of the example tree instructions.

To execute a VLIW instruction, a path from the root to some leaf node is selected, based on the values of condition codes available from the previously executed instructions and the tests specified on these condition codes in the current instruction. Since the instruction tree is oriented, the leaf nodes can be numbered uniquely starting from the left, and the different paths in the tree starting at the root and ending at a leaf node, can be identified by the number of the leaf node on the path. For example, if as a result of executing the instruction labeled L__2 in Figure 1, the values of CC1, CC2, and CC3 are True, True, and False respectively, then the path from the root to the third leaf node (from the left) is selected as shown in Figure 1.

All ALU/memory operations on the specified path are performed and the operations that are not on the selected path are not performed. The register/memory values used as input operands by the ALU/memory operations in the instruction are those that are available from the previously executed instructions (a value generated by some operation in an instruction can not be used by any other operation in the same instruction as an input operand, even when the first operation is modifying a register/memory-location that is being used as

[4] For example, in the eight queens program, no choice points are pushed down before and within (and (A k ...) (:= (p n) j)) in (E j ... (and (A k ...) (:= (p n) j))), until just before the assignment to p[n]. (A k ...) is compiled into code that, in the case of failure, just goes by direct branching to the code that increments j and tries (and (A k ...) ...) with this next value of j.

an input by the second operation, irrespective of the relative positions of these operands in the instruction).

The register/memory-locations specified as destination for the operations on the selected path are updated. If two or more operations on the selected path attempt to modify a common register/memory-location, then only the effect of the operation closest to the leaf node is registered. Once the current instruction is executed, the next instruction selected for execution is the instruction specified by the leaf node of the selected path.

Encoding of the VLIW instructions

The parallel hardware of the VLIW machine can be parameterized by the following two numbers:

1. N, the number of ALUs in the machine.
2. M, the number of branch targets that can be specified in the instruction (including the sequential successor which is implicit).

The instruction for an N ALU VLIW machine capable of doing M way branching comprises of four sets of fields as shown in Figure 2. The structure of the N ALU fields, N/2 memory operation fields, X (> M) immediate fields, and M mask fields is described next.

ALU fields: The interruptable bit, when reset, prevents the operation and operand exceptions (arising from the execution of this operation) from interrupting the VLIW machine. The different values for the 2 bit setcc field indicate that the condition code register associated with the ALU (discussed later) should not be loaded, or should be loaded with a truth value indicating that the result of the ALU operation is greater than, equal to, or less than zero.

The immediate fields for both input operands indicate whether the input is obtained from the register file (described later) or from the immediate fields (the source field for the input operands gives the register number or the immediate field number). When the register transfer enable bit (rte) is set, the destination field specifies the register which will receive the result of the ALU operation.

The transfer enable mask (temask) has M bits. Its purpose is to indicate where this ALU operation is located in the the instruction-tree. A logical true value for the i^{th} bit indicates that the ALU operation occurs on the i^{th} path of the instruction (path from the root to the i^{th} leaf node), and that no other operation closer to the i^{th} leaf node overwrites the same destination.

Memory-Operation fields: The memory operation fields indicate whether a memory operation request is present, whether the request is for a read or for a write operation, and the size/format of the data item being read/written. For the memory operation specified by the i^{th} memory operation field, the left source operand in the $2i^{th}$ ALU field specifies the memory address, the output of $2i + 1^{th}$ ALU provides the data to be stored into the specified location (in case of store operations), and the destination register in the $2i + 1^{th}$ ALU field specifies the register into which the data being read from the memory should be placed (for load operations). When the i^{th} memory operation is a load operation, the $2i + 1^{th}$ ALU cannot update any register. Furthermore, the temask of the $2i + 1^{th}$ ALU operation also serves as the temask for the i^{th} memory operation.

Immediate fields: The immediate fields in the instruction are used to specify immediate operands for ALU operations and the addresses of branch targets. The address of the branch target of the i^{th} path in the instruction (if the i^{th} path exists) is stored in the i^{th} immediate field.

The Mask fields: The i^{th} mask field specifies the conditions under which the i^{th} path in the instruction is selected. Each mask field comprises of 2N bits, two bits corresponding to each ALU. The i^{th} mask field encodes the condition codes encountered on the i^{th} path, and the values they must have for the i^{th} path to be selected.

The VLIW Hardware

The VLIW machine comprises of the following major hardware components:

1. Register File
2. Arithmetic and Logic units (ALUs)
3. Next address selection logic.
4. Data Memory Subsystem.
5. Instruction Memory Subsystem.

The high level organization of the VLIW machine is shown in Figure 3. The register file and has 2N read ports and N write ports (two read ports and one write port for each ALU). Thus, the register file can support 2N reads and N writes simultaneously. RISC-like pipelining techniques with bypass paths are used to reduce cycle time. The number of registers in the register file is also an architectural parameter, and is 64 in the current design. The ALUs are combinatorial devices except that each ALU has a 2 bit condition code register. The values of condition code register can be True, False, or Error.[5] The data memory is a multi-ported memory capable of supporting N/2 read/write operations in each machine cycle. The instruction memory is a standard wide word RAM, with the word width being several hundred bits. The next address selection unit is about six levels of low fan-in combinatorial logic.

The VLIW machine executes one instruction normally in a single cycle. Extra cycles may be spent for the instruction if there are long operations such as multiply or floating add, or memory bank conflicts. To execute an instruction all ALU/memory operations specified in the instruction are initiated concurrently at the begining of the instruction cycle. The input operand values are obtained for all operations before any updates are done by the current instruction. While

[5] Error results, e.g., when a comparison involves the result of a division by zero as one operand. The machine has a mechanism to trap on arithmetic errors and yet be able to execute operations ahead of time past conditional jumps without fear of incurring an exception which would not have occurred in the original sequential program. This mechanism will not be discussed here.

the operations are performed by the ALUs, the next address selection unit concurrently determines the path taken in the VLIW instruction and transmits the M selected path signals (exactly one of which is logical True to indicate the taken path) to all ALUs. These signals are compared with the temask bits in each ALU to determine if the operation being performed by the ALU is on the selected path, in which case the results of the ALU operation are allowed to update the designated registers, condition codes and memory locations. If the operation being performed by the ALU is not on the selected path, the operation is aborted and no updates are made. The selected path signals are also used to select the address of the next instruction from the immediate fields.

4. Compiling BSL programs into VLIW machine code

To transform BSL programs into VLIW machine code, we go through several steps. First we transform BSL programs into C language programs using a BSL source to C source translator discussed in Section 2. C programs are compiled into assembly code for a RISC like machine, using a standard optimizing compiler (currently we hand-compile the C programs, but we hope to interface to the output of the PL.8 compiler later). These assembly language instructions, each comprising of a single operation, are packed into multi-operation VLIW instruction using the percolation scheduling techniques described by Nicolau [Nicolau 85], and enhanced by us. Percolation scheduling is applied to loop-free sequential code, which may otherwise contain arbitrary conditional jumps. Percolation scheduling consists of a small set of semantics-preserving *core transformations*, that can move operations or tests from one VLIW instruction to a preceding one. Operations and tests migrate toward the begining of the program producing packed tree-instructions, and the instructions toward the end of the program eventually become empty and are deleted, thus reducing path lengths. The single assignment feature of BSL enhances the parallelism obtainable from it via percolation scheduling, since it removes anti-dependences and output dependences. The PIPER compiler developed by us performs the percolation scheduling and the pipeline scheduling described next.

To achieve further parallelism on loops whose bodies have already been compacted using percolation, we use *pipeline scheduling*, which is an extension of the "doacross" and "dopipe" techniques proposed for multiprocessors by D. Kuck's group [Davies 81, Cytron 84]. In pipeline scheduled code, a new iteration of an inner loop (possibly containing arbitrary if-then-else statements and conditional exits) can be initiated on every clock period whenever dependences and resources permit. In the code generated by our enhanced pipeline scheduler, iterations can complete out-of-sequence, and there can be arbitrary pauses between the instructions of a given iteration; these features increase the throughput rate. A weaker version of the same pipelining technique is applied to non-inner loops, which may contain other loops and/or subroutine calls. An abstract computational model of a streamlined version of our machine has been formalized using an approach inspired by denotational semantics, and the correctness and termination properties of the pipeline scheduling compilation technique have been proved [Ebcioğlu 87c].

A full description of the compilation techniques is beyond the scope of this paper, and can be found in [Ebcioğlu 87c, Ebcioğlu 88a, Nicolau 85]. We will just give an example in the Appendix A to demonstrate what the compilation techniques can do: suppose we are given the code fragment (A k (1- n) (>= k 0) (1- k) (and ...)) in the 8-queens program. According to the execution semantics of the universal quantifier and the and connective of BSL, this fragment can be compiled into a simple loop of C instructions and subsequently into a simple loop of RISC instructions, on which standard code motion and strength reduction optimizations can be applied, resulting in a code fragment such as the one given in Appendix A. (Compilation of arbitrary BSL code, and the delaying of the emission of the pushdown operations, however, is subtle and the algorithms are described in [Ebcioglu 87a]). The result of applying percolation scheduling to the RISC code, is shown after the RISC code (where the list notation is used for the instruction trees). The result of percolation scheduling on the loop body executes all operation on all paths as soon as they can be executed: In instruction 1, k is compared against 0, p[k] is fetched, n-k is computed, and k and the pointer pk to p[k] are incremented. In instruction 2, p[k]-j, and j-p[k] are computed, j is compared against p[k], and the loop is exited if k (with success) was less than zero as a result of the comparison in the previous instruction. In instruction 3, p[k]-j and j-p[k] are compared against n-k, and the loop is exited (with failure) if j was equal to p[k] during the comparison of instruction 2. In instruction 4, control branches back to the beginning of the loop if p[k]-j and j-p[k] were both unequal to n-k, depending on the comparisons of the previous instruction 3. The result of applying the pipeline scheduling technique to the compacted loop body is shown next. The resulting code initiates a new iteration of the loop on every cycle. So, in the first cycle, iteration 1 executes instruction 1, in the second cycle iteration 1 executes instruction 2 and iteration 2 executes instruction 1, etc.. The throughput of the final code is one iteration per cycle, with 3 cycles to fill the pipeline. Note that, because of the conditional jumps and conditional exits in the loop body, this loop would be difficult to vectorize on a supercomputer, also, it would not be worthwhile to allocate the different iterations of the loop to different processors in a multiprocessor configuration, because of the small number of times the loop is iterated. So the VLIW architecture and compilation techniques seem to be a good match for parallelizing this kind of nonnumerical code.

5. Performance results

To increase our confidence in the performance of the proposed VLIW machine, we developed a register transfer level simulator for the machine. The simulator was exercised by four artificial intelligence type search problem kernels, which are listed in Appendix B, together with their description as first order logic formulas.

The BSL programs were converted to C programs automatically by the translator described in Section 2. The C pro-

grams were hand-coded into sequential assembly language, and these assembly language programs were again automatically translated to compacted VLIW machine code by the PIPER compiler mentioned in Section 4. This compacted code was used to exercise the simulator. To estimate the advantages of the compaction process, we also generated VLIW code from the assembly code and exercised the simulator with this uncompacted code. When executing the uncompacted code, the VLIW machine behaves like a RISC (Reduced Instruction Set Computer) machine. In Table 1 we show the number of VLIW machine cycles required to execute the compacted/uncompacted versions of the VLIW code for the problems listed in Appendix B. The speedup obtained by the compaction process, which is the ratio of these two columns is also shown in column 3. The speedup obtained is in the range of 2.5 to 5.5. (The compiler is unable to pipeline the inner loop of "triangle" due to the lack of a renaming optimization. But we hope to fix this problem in a future version of the compiler).

The C programs obtained from the BSL programs mentioned in Appendix B were compiled and executed on an IBM 3090 (model 200) mainframe under VM/CMS. The command execution times, as reported by VM, are listed in Table 2. These figures are representative of the infinite cache performance of the 3090 since the BSL programs repeatedly access the same small data structures. Next the same algorithms were coded in PROLOG (with the static clause optimization for each program) and were executed using the VM/PROLOG interpreter on the same mainframe, and the execution times are reported in column 2 of Table 2. Finally, assuming that we can prototype a VLIW machine with a 50 ns. worst case cycle time in a conservative CMOS technology (as planned), the time required to execute these problems on the VLIW machine (estimate) is given in column 3 (based on cycle count obtained from simulation).

While the results of Table 2 are preliminary and are not sufficient to draw definitive conclusions, the following indications are obvious. Generate-and-test type search algorithms, when coded in BSL, tend to execute much more efficiently than when coded in PROLOG. The speedup is usually around 20 for these particular implementations of BSL and PROLOG.[6] Furthermore, the proposed VLIW machine appears to be faster than the IBM 3090 mainframe by a factor of roughly 3 on the programs listed in Appendix B.

6. Discussion

In this paper, we have described a logic programming language called BSL, and a practical method of extracting parallelism from BSL programs via a new wide instruction word architecture and related compilation techniques.

We wish to remark briefly on how our approach differs from the present parallel execution paradigms for Prolog-like languages. The present parallel execution paradigms for ordinary Prolog and the variants of Prolog designed explicitly for parallelism (e.g. Concurrent Prolog, GHC), typically assume an MIMD multiprocessor system, or an architecture inspired from data flow approaches [Conery 83, Ueda 85, Onai et al. 85, Kalé 87]. In such paradigms, there are overhead issues such as scheduling the processes on the available processors at run time, load balancing, synchronization between processors when one needs a value produced by the other, memory contention when processors share data, and communication delays between processors. As a result of such overhead issues, exploiting parallelism at the finest grain has been considered inadvisable [Maruyama et al. 85]. Our VLIW approach is able to achieve fine grain parallelism and to tolerate very low degrees of problem parallelism, since run-time scheduling and synchronization delays are eliminated by resolving all data dependences and performing scheduling at compile time, and communication delays between processing elements are greatly reduced by virtue of very tight coupling via a shared register file. The parallelism in BSL is mainly fine-grain and-parallelism, with some fine-grain or-parallelism. The or-parallelism is implemented as follows: when the current alternative is being executed, operations from the remaining alternatives start executing before it is known that the current alternative is going to fail or succeed, and if the current alternative fails before a choice point is pushed down (the compiler optimizations greatly increase this possibility) some headway will have been made for the next alternatives. We should remark that the fact that we are presently concentrating on fine grain parallelism does not imply that our approach excludes coarse grain parallelism or denies its utility; for example, higher degrees of or-parallelism in BSL could certainly be exploited by connecting a number of VLIW processors with shared memory.

The language BSL also has some desirable features. Formulas in BSL are not limited to Horn-clause or clausal form, and allow direct coding of universal and existential quantifiers. Such quantifiers are often compiled into loops, thus making it possible to take advantage of sequential and parallel compiler optimizations for loops. BSL avoids unification: the choice between making equality (BSL assignment - analogous to unifying a bound and an unbound variable in Prolog) and checking for equality (BSL equality test - analogous to unifying two already bound variables in Prolog) is made at program writing time. While the removal of unification and other simplifications result in loss of versatility (e.g., relational programming is not possible in BSL), an efficient implementation becomes possible, which enables the programmer to use the concepts of first order logic offered by BSL in computation-intensive A.I. applications.

The status of the architecture/compiler effort for the VLIW machine described in this paper is as follows: We presently have a preliminary working version of a VLIW compiler that takes RISC-like intermediate code as input and produces VLIW tree-instructions, a microassembler that compiles the

[6] Note that VM/PROLOG is an interpreted language, while BSL is a compiled language. Using a Prolog compiler (none were available to us) rather than an interpreter would of course reduce the performance differences between BSL and Prolog, but would probably not eliminate them, since VM/PROLOG already has a very efficient implementation involving clause indexing and static clauses (partial compilation). A high performance, optimizing Prolog compiler for the IBM S/370 (1.42 megalips on *append* on a 3090) that also compiles into PL.8, was reported in [Kurokawa et al. 86], which is (on the basis of code examples and timings given in that paper) about 4.6-6.2 times faster than VM/PROLOG with static clauses.

tree-instructions into binary machine code, and a register transfer level simulator that accepts binary VLIW code, simulates it, and provides detailed timing information about program execution. Gate level logic schematics have been completed for an 8-ALU design for the machine. The planned technology for the prototype is 1.5 micron CMOS VLSI. An effort is presently underway at the IBM T.J. Watson Research Center to build a prototype of this machine, and we will report on our progress in future papers.

Acknowledgements

We would like to thank Fran Allen, Mauricio Breternitz, Micheal Burke, John Cocke, Ron Cytron, Dave George, Jean-Louis Lassez, George Radin, for their comments on the architecture, compilation techniques and parallelism issues discussed in the paper.

References

Cohen, J. (79) "Non-deterministic algorithms" Computing Surveys Vol. 11, No. 2, June 1979.

Conery, J.S. (83) "The AND/OR Process Model for Parallel Interpretation of Logic Programs" Phd thesis and technical report 204, The University of California at Irvine, 1983.

Cytron, R.G. (84) "Compile-time Scheduling and Optimization for Asynchronous Machines" Report no. UIUCDCS-R-84-1177, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1984.

Date, C.J. "Introduction to Database Systems" Addison-Wesley, 1977.

Davies, J.R.B.(81) "Parallel Loop Constructs For Multiprocessors" Report no. UIUDCS-R-81-1070, Dept. of Computer Science, University of Illinois at Urbana-Champaign, May 1981.

de Bakker, J. (79) "Mathematical Theory of Program Correctness" North Holland, 1979.

Ebcioğlu, K. (87a) "Report on the CHORAL project: An Expert System for Harmonizing Four-part Chorales" research report RC12628, IBM Thomas J. Watson Research Center, Yorktown Heights, March 1987. (This is a revised version of the author's Ph.D. dissertation, "An Expert System for Harmonization of Chorales in the Style of J.S. Bach," technical report TR 86-09, Dept. of Computer Science, S.U.N.Y. at Buffalo, March 1986.)

Ebcioğlu, K. (87b) "An Efficient Logic Programming Language and its Application to Music" Proc. 4th ICLP, May 1987.

Ebcioğlu, K. (87c) "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps" Proc. 20th Annual Workshop on Microprogramming (MICRO-20), December 1987.

Ebcioğlu, K. (88a) "Some Design Ideas for a VLIW Architecture for Sequential-Natured Software" Proc. IFIP Working Conference on Parallel Processing, Pisa, Italy, April 1988.

Ebcioğlu, K. (88b) "An Expert System for Harmonizing Chorales in the Style of J.S. Bach" Special Issue of the Journal of Logic Programming on Applications of Logic Programming, to appear, 1988.

Ebcioğlu, K. (88c) "An Expert System for Harmonizing Four-part Chorales" Computer Music Journal, Vol. 12, no. 3, Fall 1988.

Ellis, J.R. (86) "Bulldog: A Compiler for VLIW Architectures" MIT Press, 1986.

Fagin, B. and Dobry, T. (85) "The Berkeley PLM Instruction Set: An Instruction Set for Prolog" Report no. UCB/CSD 86/257, Computer Science Division (EECS), University of California at Berkeley, September 1985.

Floyd, R. (87) "Nondeterministic Algorithms" Journal of the Association for Computing Machinery, Vol. 14, no. 4, October 1967.

Forgy, C. and McDermott, J. (77) "OPS: A Domain Independent Production System Language" Proceedings of the fifth International Joint Conference in Artificial Intelligence, 1977.

Harel, D. (79) "First Order Dynamic Logic" Lecture Notes in Computer Science, Goos and Hartmanis (eds.), Springer-Verlag 1979.

Kalé, L.V. (87) "The REDUCE-OR Process Model for Parallel Evaluation of Logic Programs" Proc. 4th International Conference on Logic Programming, 1987.

Kurokawa, T., Tamura, N., Asakawa, Y., and Komatsu, H. (86) "A Very Fast Prolog Compiler on Multiple Architectures" Proc. FJCC 1986.

Maruyama, T., Hirata, K., Tanaka, H., and Moto-Oka, T. (85) "A Note on the Elementary Execution Unit in a Parallel Inference Machine" Proc. 4th Conference on Logic Programming, Tokyo, 1985.

Nicolau, A. (85) "Percolation Scheduling: A Parallel Compilation Technique" TR 85-678, Dept. of Computer Science, Cornell University, May 1985.

Onai, R., Shimizu, H., Masuda, K., Matsumoto, A., Aso, M., (85) "Architecture and Evaluation of a Reduction-based Parallel Inference Machine: PIM-R" Proc. 4th Conference on Logic Programming, Tokyo, 1985.

Robinson, J.A. (65) "A Machine Oriented Logic Based on the Resolution Principle" Journal of the Association for Computing Machinery 12, 1965.

Smith, D.C. and Enea, H.J. (73) "Backtracking in Mlisp2" Proceedings of the third International Joint Conference in Artificial Intelligence, 1973.

Turk, A.W. (86) "Compiler Optimizations for the WAM" Proc. 3rd ICLP, 1986.

Ueda, K. (85) "Guarded Horn Clauses" Technical report TR-103, ICOT, 1985, Tokyo.

Warren, S.H., Auslander, M.A., Chaitin, G.J., Chibib, A.C., Hopkins, M.E., and MacKay, A.L. (86) "Final Code Generation in the PL.8 Compiler" report no. RC 11974, IBM T.J. Watson Research Center, 1986.

APPENDIX A

RISC-like intermediate code for inner loop of an 8-queens fragment. (The destination register occurs last in the following intermediate code instructions, so (LT K 0 CC1) means CC1:=(K<0)).

```
(A k (1- n) (>= k 0) (1- k)
     (and (!= j (p k))
          (!= (- j (p k)) (- n k))
          (!= (- (p k) j) (- n k))))
```

```
L_12
    (LT K 0 CC1)
    (IF CC1 (GOTO L_11))
    (LOAD P PK PSUBK)
    (EQ J PSUBK CC2)
    (IF CC2 (GOTO L_10))
    (SUB J PSUBK T1)
    (SUB N K T2)
    (EQ T1 T2 CC3)
    (IF CC3 (GOTO L_10))
    (SUB PSUBK J T3)
    (EQ T3 T2 CC4)
    (IF CC4 (GOTO L_10))
    (SUB K 1 K)
    (SUB PK 4 PK)
    (GOTO L_12)
```

Percolation scheduling result:

```
(L_12
  T (LT K 0 CC1) (LOAD P PK PSUBK) (SUB N K T2)
    (SUB K 1 K) (SUB PK 4 PK) (GOTO _75) ))

(_75
  ( (EQ J PSUBK CC2) (SUB J PSUBK T1) (SUB PSUBK J T3)
    (IF CC1 ((GOTO L_11)) ELSE ((GOTO _74))) ))

(_74 ((EQ T1 T2 CC3) (EQ T3 T2 CC4) (IF CC2 ((GOTO L_10))
                                     ELSE ((GOTO _73))))

(_73
  ((IF CC3 ((GOTO L_10)) ELSE ((IF CC4 ((GOTO L_10))
                                ELSE ((GOTO L_12))))))
```

Pipeline scheduling result:

```
(L_12
  T (LT K 0 CC1) (LOAD P PK PSUBK) (SUB N K T2)
    (SUB K 1 K) (SUB PK 4 PK)   (GOTO (_75 L_12)) ))

((_75 L_12)
  T (EQ J PSUBK CC2) (SUB J PSUBK T1) (SUB PSUBK J T3)
    (IF CC1 ((GOTO L_11)))
     ELSE ( (COPY T2 T2_P) (LT K 0 CC1) (LOAD P PK PSUBK)
            (SUB N K T2) (SUB K 1 K) (SUB PK 4 PK)
            (GOTO (_74 (_75 L_12))) )) ))

((_74 (_75 L_12))
  T (EQ T1 T2_P CC3) (EQ T3 T2_P CC4)
    (IF CC2 ((GOTO L_10))
     ELSE ( (EQ J PSUBK CC2) (SUB J PSUBK T1) (SUB PSUBK J T3)
            (IF CC1 ((GOTO (_73 L_11)))
             ELSE ( (COPY T2 T2_P) (LT K 0 CC1) (LOAD P PK PSUBK)
                    (SUB N K T2) (SUB K 1 K) (SUB PK 4 PK)
                    (GOTO (_73 (_74 (_75 L_12)))) )) )) ))

((_73 L_11)
  T((IF CC3 ((GOTO L_10)) ELSE ((IF CC4 ((GOTO L_10))
                                 ELSE ((GOTO L_11)))))))

((_73 (_74 (_75 L_12)))
  T ((IF CC3 T((GOTO L_10))
     ELSE ( (IF CC4 T((GOTO L_10))
             ELSE ( (EQ T1 T2_P CC3) (EQ T3 T2_P CC4)
                    (IF CC2 ((GOTO L_10))
                     ELSE ( (EQ J PSUBK CC2) (SUB J PSUBK T1)
                            (SUB PSUBK J T3)
                            (IF CC1 ((GOTO (_73 L_11)))
                             ELSE ( (COPY T2 T2_P) (LT K 0 CC1)
                                    (LOAD P PK PSUBK) (SUB N K T2)
                                    (SUB K 1 K) (SUB PK 4 PK)
                                    (GOTO (_73 (_74 (_75 L_12)))) ))
                                       ))) )) ))
```

APPENDIX B

Examples used to estimate the performance of the VLIW machine. The assignment statements of the BSL programs have been kept intact in their logical translations given here, so that the original BSL programs can be inferred.

triangle: enumerate all triples of integers x,y,z, 0<x<y<z<60, such that x**2+y**2=z**2 (Pythagorean numbers).

```
(∃x,y,z:integer)(∃i | 1≤i<58)(∃j | i+1≤j<59)(∃k | j+1≤k<60)
        [i*i+j*j=k*k & x:=i & y:=j & z:=k].
```

permute: enumerate all permutations of the digits 0,1,...,6

```
(∃p:(array (7) integer))
    (∀n | 0≤n<7)
        (∃j | 0≤j<7)
            [(∀k | n-1≥k≥0)[j≠p[k]]
             & p[n]:=j].
```

queens: find all solutions to the 8-queens problem. The rows and columns are numbered as 0,1,...,7, and the array elements p[0],...,p[7] represent the column no. of the queen on row 0,...,7, respectively.

```
(∃p:(array (8) integer))
    (∀n | 0≤n<8)
        (∃j | 0≤j<8)
            [(∀k | n-1≥k≥0)
                [j≠p[k] &j-p[k]≠n-k & p[k]-j≠n-k]
             & p[n]:=j].
```

dslalpha: enumerate the names of the suppliers who supply all parts. Taken from a DSL ALPHA query for the suppliers-parts database in [Date 77].

```
(∃s,p,sp)
    [s="((s__sno S1 s__sname SMITH s__status 20
                  s__city LONDON) ...)" &
     p="((p__pno P1 p__pname NUT p__color RED
                  p__weight 12 p__city LONDON) ...)" &
     sp="((sp__sno S1 sp__pno P1 sp__qty 300) ...)" &
     (∃ans:nametype)
         (∃n | 0≤n<S__SIZE)
             [(∀i | 0≤i<P__SIZE)
                 (∃j | 0≤j<SP__SIZE)
                     [sp[j].sp__sno=s[n].s__sno
                      & sp[j].sp__pno=p[i].p__pno]
              & ans:=s[n].s__sname]].
```

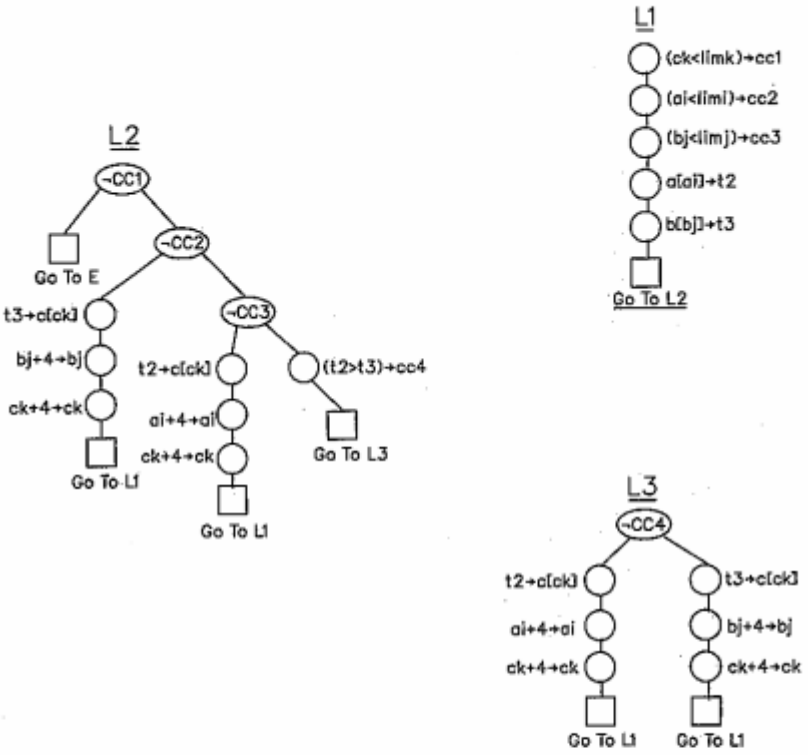
TABLE 1

Programs	Compact.	Un−comp.	Ratio
Queens	145568	677950	4.66
Permute	671868	2575107	3.83
Triangle	181557	450289	2.48
Dslalpha	187	1034	5.53

Table 2*

Programs	PROLOG	BSL	VLIW
Queens	600	26	7.3
Permute	937	97	33.6
Triangle	617	24	9
Dslalpha	.19	.021	.0093

* all times are in miliseconds




```
(L_1
  ((LT CK LIMK CC1) (LT AI LIMI CC2) (LT BJ LIMJ CC3) (LOAD A AI T2)
    (LOAD B BJ T3) (GOTO L_2) ))

(L_2
  ((IF (NOT CC1) ((GOTO E))
    ELSE ((IF (NOT CC2)
              ((STORE C T3 CK C) (ADD BJ 4 BJ) (ADD CK 4 CK) (GOTO L_1))
            ELSE ( (IF (NOT CC3)
              ((STORE C T2 CK C) (ADD AI 4 AI) (ADD CK 4 CK) (GOTO L_1))
                  ELSE  ((GT T2 T3 CC4) (GOTO L_3))) )) )) ))

(L_3
  ((IF (NOT CC4) ((STORE C T2 CK C) (ADD AI 4 AI) (ADD CK 4 CK)
                                                  (GOTO L_1))
    ELSE ((STORE C T3 CK C) (ADD BJ 4 BJ) (ADD CK 4 CK) (GOTO L_1)))T ))
```

Figure 1: VLIW Instructions

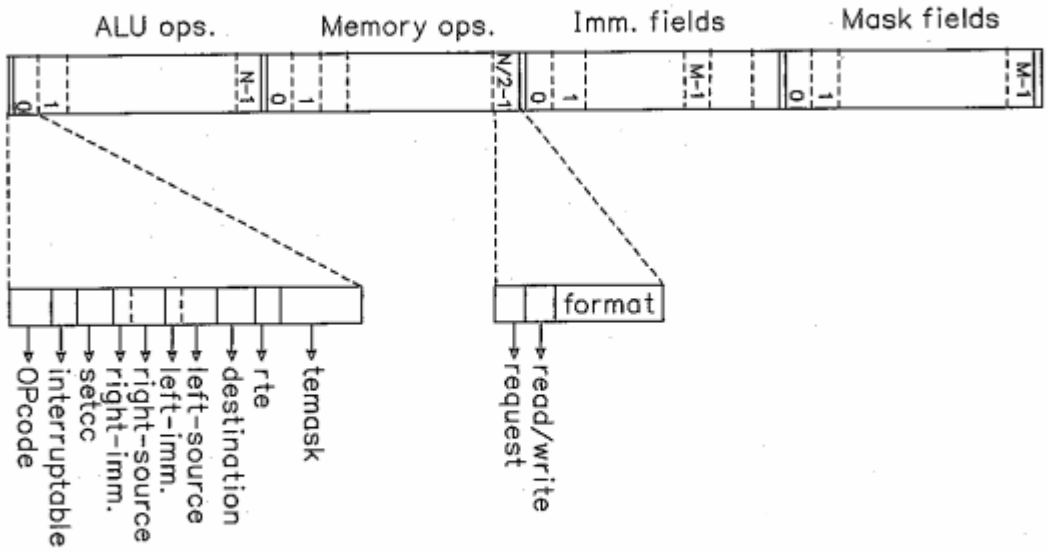


Figure 2: Encoding of a VLIW Instruction.

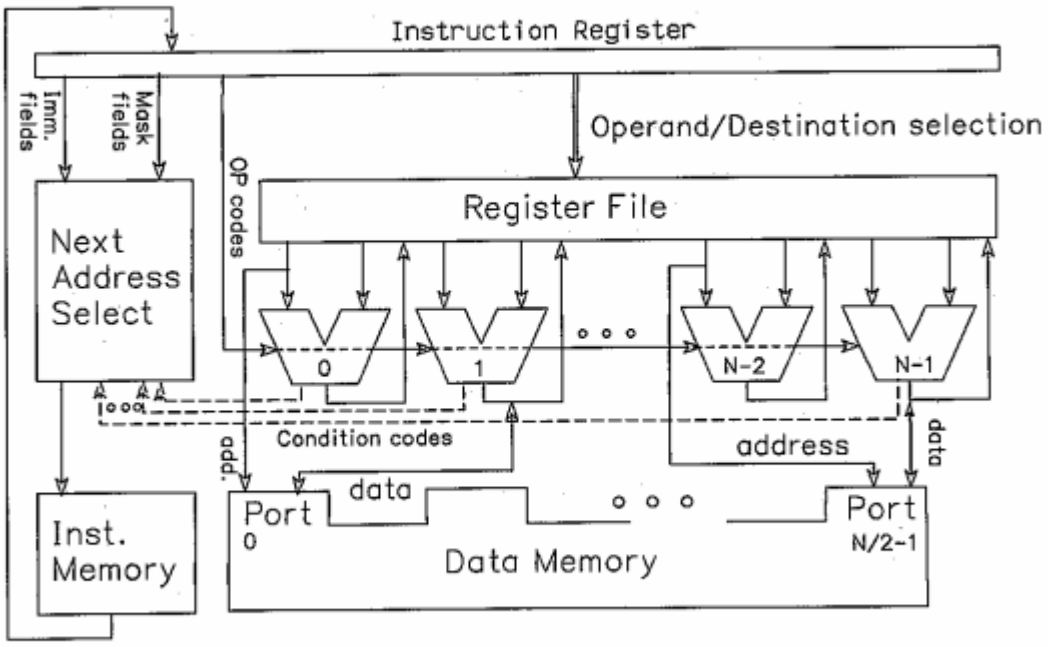


Figure 3: Overview of the VLIW Machine Hardware.