TECHNICAL REPORT NO. 284

A Pipelined Architecture for Logic Progamming with a Complex but Single-Cycle Instruction Set

by

Jonathan W. Mills

July 1989

COMPUTER SCIENCE DEPARTMENT

INDIANA UNIVERSITY

Bloomington, Indiana 47405–4101

A Pipelined Architecture for Logic Programming with a Complex but Single-Cycle Instruction Set

Jonathan Wayne Mills
Computer Science Department
Indiana University
Bloomington, Indiana 47405

An architecture is described which executes logic programs using fewer instruction cycles than hardware implementations of the Warren Abstract Machine or the Berkeley SPUR augmented with a Prolog coprocessor. This is achieved by balancing the characteristics of CISC and RISC architectures. Specifically, this architecture provides support for the semantics of logic programs using complex instructions and multiple pipelined functional units. Examples of complex instructions include **partial unify**, **push and load reference**, **pop and dereference**, and **switch on type**; all typically execute in a single clock cycle from a full pipeline. Conditional instruction execution reduces branch frequency to 0.09%, which keeps the pipeline full and allows 16-way memory interleaving. Under these conditions, one LIBRA processor using 100ns memory is estimated to execute nine million logical inferences per second.

1. INTRODUCTION

The design of the LIBRA, or **L**ogical **I**nference **B**alanced **R**ISC **A**rchitecture, has been a process of unlearning assumptions about the hardware required for a fifth generation language, Prolog. The belief that logic programs required a non-von Neumann architecture led to the design of complex computer architectures, such as the PLM, the PSI, the HPM, and the PEK. Later, attempts to obtain performance using RISC architectures produced the SPUR and its Prolog coprocessor, the LISC-P, the RPM, and the Pegasus. These RISC architectures suffered from the "add-on" symptom of design: an existing architecture was extended at the lowest level to support Prolog, rather than designed from the top down.

The LIBRA was designed in a "top-down" fashion over a four-year period, with each version integrating one or more functions to support logic programs into a RISC-like architecture. The development is summarized in the following chronology:

1985 WAM was analyzed, component functions used to design a sub-WAM in software for Ken Bowen's Applied Logic System's PC Prolog compiler. The implementation was efficient (>5 KLIPS on an IBM PC) but indicated a need for integrated tag and value processing in hardware.

1986 LOW RISC I. Seven instructions, register windows. Verified need for integrated tag and value processing; had problems with excessive branching and dereference loops; found that register windows were under-utilized.

1987 LOW RISC II. 15 instructions; no register windows. Simulated, nominal speeds 300 - 800 KLIPS. Still had excessive branching problems (particularly in unification).

1988 LIBRA. An approximately balanced architecture with 35 instructions. *Balance* is a concept inspired Flynn et. al. (1987), who introduced the idea of a **balanced optimization** to an instruction set, arguing for register windows only if instruction traffic could be reduced. In this paper a *perfectly balanced architecture* is defined to be one where the instruction traffic equals the data traffic, and where for each instruction fetch there is a simultaneous data fetch. The LIBRA is not perfectly balanced.

Currently a VLSI bit-slice of the LIBRA's value ALU has been designed, simulated in SPICE and will be submitted to MOSIS for fabrication in June 1989.

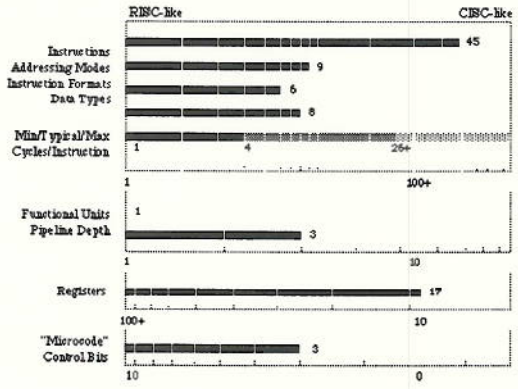
2. PREVIOUS LOGIC PROGRAMMING ARCHITECTURES

Although a number of logic programming architectures have been described, the PLM and the SPUR are selected to represent the classes of CISC and RISC logic programming architectures, respectively. For discussion of other logic programming architectures, see (Mills 1988).

2.1 THE PLM

The PLM is an extension of the Warren abstract Prolog machine (WAM). The profile of the PLM shows an architecture that is CISC-like in the number of registers and instruction complexity, but RISC-like in the number of functional units. The PLM is an overlapped fetch-and-execute machine. Instructions are prefetched, partially decoded and stored in an instruction buffer. Each PLM instruction is microcoded, and because many instructions share microcode for dereferencing, de-cdring and trailing a single-level microcode call is implemented. Some instructions may invoke a recursive unifier, so an eight-level push-down list is implemented. The microcode word is 128 bits wide, necessary because 11 buses may be used during a single cycle.

Eight hardware data types are defined using four basic tag values. All data types may be identified as cdr or non-cdr coded values using a secondary type field, and include a garbage collect bit. Tag bits from three argument registers (A0, Ax[arg1], Ax[arg2]) are always provided to the next



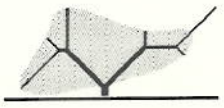


Figure 1. PLM profile

to the next micro-address multiplexer, with the two primary tag bits used to make a four-way branch. This increases the ability of the PLM to perform **switches**, but falls short of the **partial unify** instruction found in the LIBRA, which can perform most unifications in a single cycle; in the PLM unification takes a minimum of four cycles.

The PLM implements a Harvard bus architecture, having separate instruction and data memory spaces each accessed by its own address bus and data bus. The data memory is allocated to heap, environment/choice point stack, trail and symbol tables. Within the PLM seventeen registers are visible to the programmer. Nine are state registers needed to control the operation of the PLM, and eight are argument registers.

Because the micro-architecture of the PLM is conventional and not optimized for Prolog (although the macro-architecture is) and because the PLM is vertically microcoded, the execution time of each instruction varies widely: some instructions require as many as 26 cycles. The PLM's microcoded WAM instructions are complex without being flexible.

2.2 THE SPUR AND ITS PROLOG COPROCESSOR

The SPUR is an extension of the Berkeley RISC II targeted for Common Lisp but not for Prolog (Borrielo et. al. 1987). The Lisp orientation increases the semantic gap between the architecture and Prolog and contributes to the poor performance of an unaugmented SPUR running Prolog. To

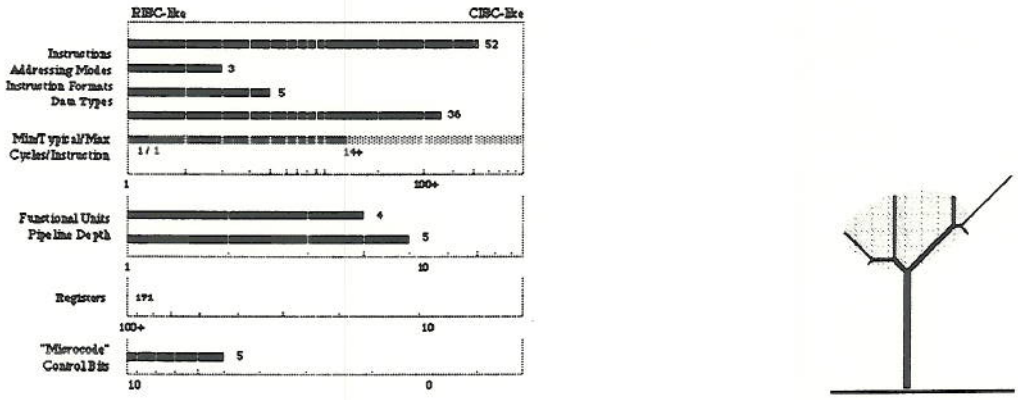


Figure 2. SPUR-Prolog coprocessor profiles

improve performance, a SPUR coprocessor was introduced for unification, tag dispatching and dereferencing. The SPUR / Prolog co-processor combination executes Prolog slightly faster than the PLM. The SPUR contains three processors: an integer CPU, a floating-point CPU (FPU), and a cache controller (CC). The integer CPU is tagged, using six bits of each 40-bit word for the tag. The floating point CPU is a hardwired implementation of the IEEE binary floating-point arithmetic standard, and uses three bits of the 87-bit word for the tag. The SPUR's 4-stage pipeline includes forwarding logic, allowing results to be used before they have been stored in, and become available from, the register file. This gives the SPUR an effective instruction execution speed of one cycle per instruction.

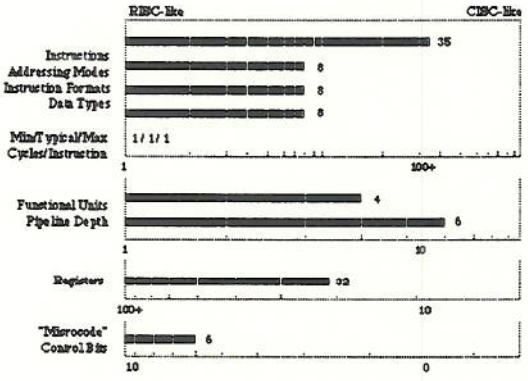
The SPUR can implement 32 tagged data types, although only four types — fixnum, character, cons pointer, and nil — and one condition, tags equal, are implicitly tested. All other tag manipulation is performed explicitly by comparing tags to immediate values. A significant disadvantage of tag processing on the SPUR is the need to read the tag, **and** the tag with a mask, and then jump indirect to implement multi-way tag dispatching. The **and** is necessary because the tag includes two generation scavenging bits which may have an arbitrary value; there is no way to extract the type information from a tag directly. If this sequence were implemented as a single instruction as is done in the LOW RISC and LIBRA processors, the SPUR's Prolog execution speed could be improved by as much as 15% (Borrielo et. al. 1986).

The SPUR has 138 general purpose integer registers in the CPU (eight special purpose registers are not counted in this analysis because they are not used in a sequential WAM implementation), and 16 87-bit floating point registers in the FPU. The 138 integer registers are organized into 10 global registers and eight overlapping windows containing 16 registers each. The size of the register windows and their organization as a monolithic register file limit their use for Prolog. Most Prolog calls use fewer than three parameters, with more than seven parameters used in fewer than 0.3% of all cases (Matsumoto 1985). Using the register file as a choice point stack frequently wastes registers.

No bounds checking instructions are available, nor are stack manipulation instructions. **Push** and **pop** instructions must be synthesized from **load-add** and **subtract-store** instruction pairs, in keeping with the RISC philosophy. Dereferencing for bound variables, called *invisible pointers* by the SPUR designers after Greenblatt (1974), is not supported. This reduces the SPUR's performance for Prolog, and was partly remedied by Borrielo et. al., who put dereferencing back into the architecture in the Prolog coprocessor. However, because the SPUR coprocessor always combines dereferencing with another operation — a branch, unification, or hashing — the coprocessor adds additional complexity but not flexibility.

3. LIBRA: LOGICAL INFERENCE BALANCED RISC ARCHITECTURE

The LOW RISC predecessors to the LIBRA defined the drawbacks of a RISC architecture with a simple instruction set. Although the LOW RISC architectures supported tags and branches, the branch frequency was still high, and the code density low. Furthermore the penalties imposed by calls to the unifier (when most unification can be performed non-recursively) and short loops for dereferencing, short branches for trail checking, and the lack of stack manipulation instructions led to the idea that a RISC architecture for logic programming should have as complex an instruction set as possible, while retaining the constraints of a RISC. As it turned out, the major constraint was that all instructions should execute in a single clock cycle.



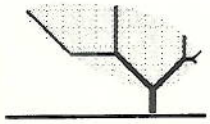


Figure 3. Profile of the LIBRA

The LIBRA is a 40-bit 4-stage pipelined processor with four major functional units, each pipelined and operating synchronously in parallel:

1. Value ALU. Contains separate hardware for arithmetic/logic, dereferencing, bounds checking.
2. Tag ALU. Tag comparison, interface to partial unifier.
3. GC ALU. Support for mark-and-sweep garbage collection algorithms.
4. PC ALU. Next instruction fetching; many branches are simple counter loads, but of a partial field. Fastest branches are within a page, with the page size varying from 512 words to 1 megabyte.

The 40-bit data word is divided into a 3-bit type, 1 bit each for mark and reverse garbage collect flags, and a 35-bit value which can be further typed for use with 32-bit numeric host processors. The machine is microcoded, but uses only one control word per machine instruction. An alternate microcode address composed of the two operands' tags is latched after every instruction that sets condition codes. Minimal arithmetic and operating system support is provided; however, the LIBRA can be extended with a numeric coprocessor such as the Motorola 68882.

Pipeline breaks in the LIBRA are reduced by moving partial unification and trail checking into hardware, and eliminating many short branches by conditional execution of all instructions:

Partial unification ..... uses the alternate microcode address to select one machine instruction to replace the **partial unify** instruction.. Although only one operation can be performed, it is enough to handle most strength-reduced unification in the Prolog.

Trail checking ..... is performed when an unbound variable reference is loaded, with the actual trailing performed by a conditional stack push. When the LIBRA executes a load or dereference instruction it always checks the value loaded. If the check shows that the

value is an unbound variable and must be trailed when it is instantiated, the register into which it is loaded is marked by setting a trail-check flag in the scoreboard. Later, when the unbound variable is bound, the status bit is used to conditionally execute a trailing instruction.

Conditional execution ..... decreases the number of short branches by changing short sequences of "branch around" code into sequential (but possibly not executed) instructions.

Pre- and post- increment and decrement memory addressing modes are also added, all of which operate in a single cycle. Data memory interleaving is enhanced because sequential reads and writes into the Prolog stacks comprise approximately 30% of the data memory references. Because pipeline breaks occur after an average of eight instructions with conditional instruction execution, instruction memory interleaving is also effective.

4. INSTRUCTION SET SUPPORT FOR LOGIC PROGRAMS

The LIBRA instruction set is broken into eight major categories (Figure 4). Data manipulation instructions execute tag and value operations in parallel, with orthogonal behavior in each category. When necessary the tag result can be ignored and the value alone used, giving behavior similar to other RISCs. The evolution of the parallel tag and value operations is described in (Mills 1988). The instructions are orthogonal to allow ease of compilation; as can be seen in Appendix A many of the instruction variants are not used. If the requirement is to emulate the WAM then the instruction set can be reduced to 16 non-orthogonal instructions by eliminating all variants. This forces the compiler to assume specific directions for stack growth, for example, but reduces the control circuitry substantially.

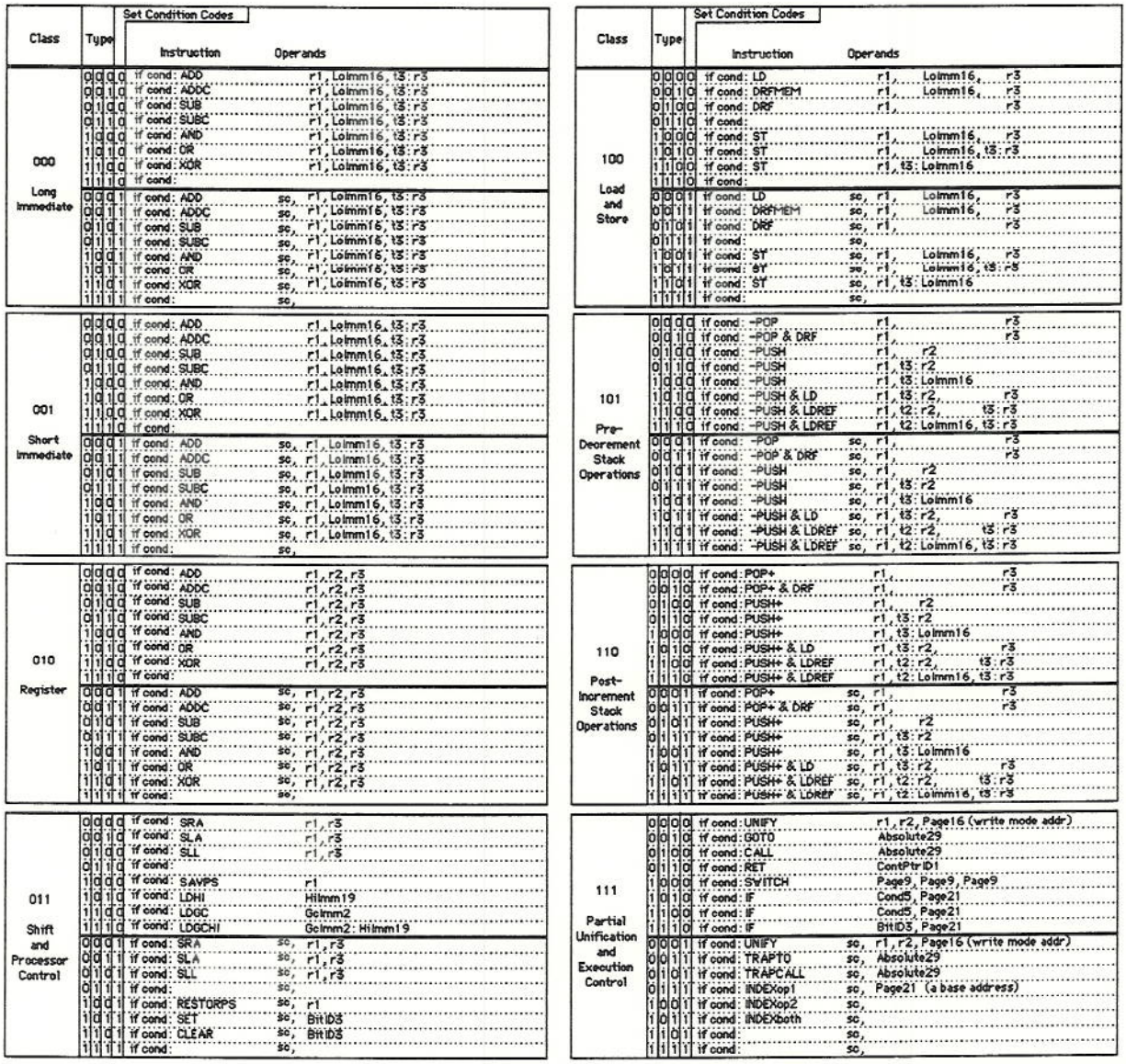


Figure 4. LIBRA instruction set

4.1 HOW IT SUPPORTS LOGIC PROGRAMS

One way the LIBRA instruction set can be used to implement logic programs is to simply macro-expand WAM instructions into LIBRA instructions (Appendix A). Because the instruction set is so efficient this is an efficient solution, in fact, the LIBRA uses 2.3 times fewer cycles than does the PLM to execute WAM-encoded Prolog programs (Appendix B).

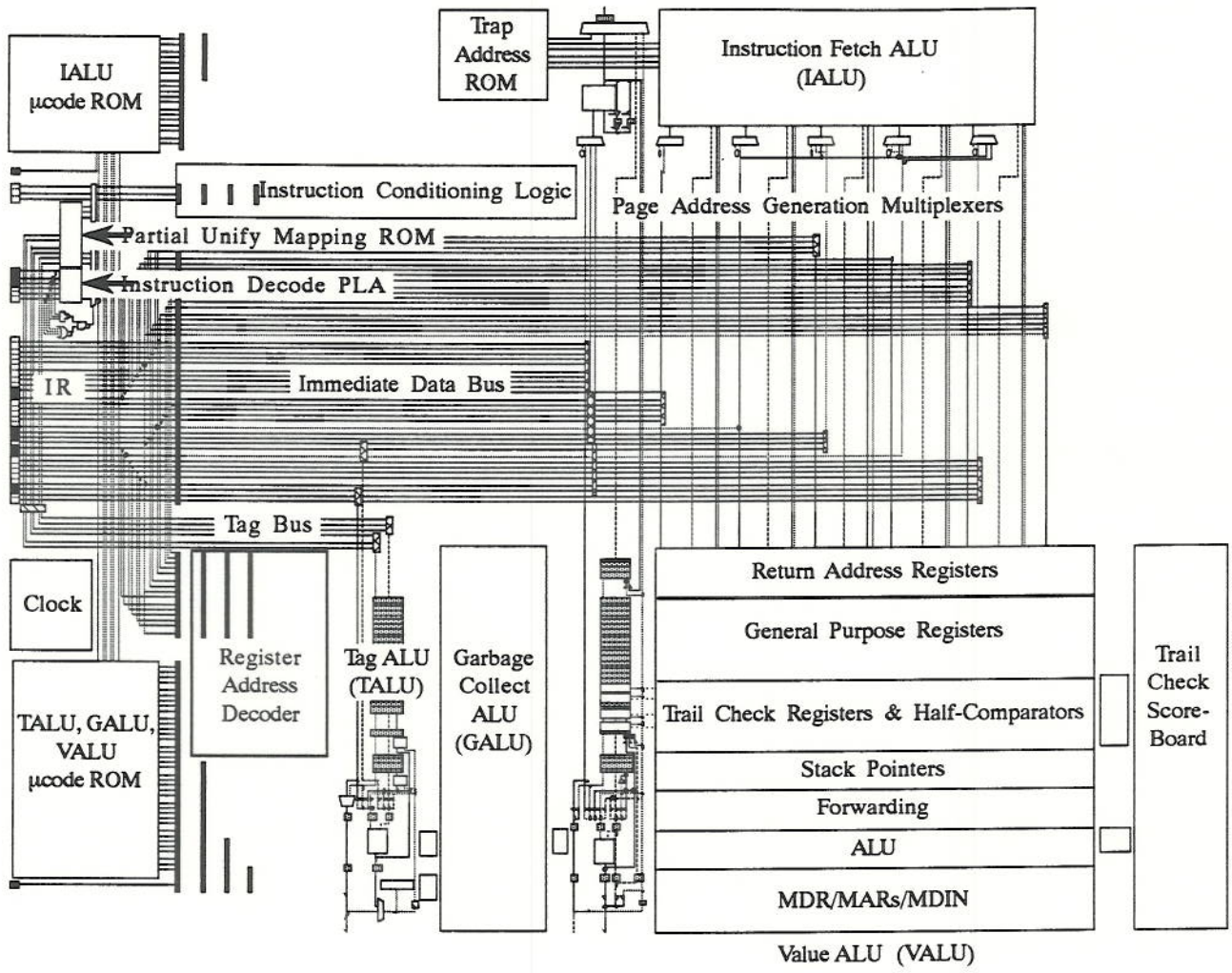


Figure 5. LIBRA schematic block diagram

5. HARDWARE SUPPORT FOR INSTRUCTION SET

Figure 5 shows the organization of the LIBRA.

5.1 NON-ORTHOGONAL REGISTER FILE WITHOUT REGISTER WINDOWS

There are 32 registers, all visible. 4 are stack pointers, 20 are general purpose, 4 are bounds checking registers, and 4 are return address registers for subroutine calling (or continuation pointers for the WAM). The register bank is non-orthogonal to allow single cycle instructions such as **push and load reference** which can:

push register A using register B as a pointer to memory,

increment/decrement register B,

store register B into register C, overwriting the tag in C with an immediate.

These instructions make use of parts of the datapath that would be idle during forwarding, and routes their contents back to the register file, which is useful in a structure-copying implementation of Prolog.

Tick (1988) and evaluation of the Pegasus RISC (Seo and Yokota n.d.) suggest that a single specialized window ("shadow registers") provides the optimum performance improvement that a Prolog processor can obtain from multiple register sets. However, Flynn et. al. (1987) argue for register windows only if instruction traffic could be reduced. This is supported by the earlier experiences with the LOW RISC I and II, and the SPUR: a Berkeley RISC II-style set of register windows is not useful for a Prolog processor because the number of parameters passed is frequently small; also, register windows assume that only one stack need be buffered in the CPU, and that the buffer depth is deep.

5.2 SMART CACHE

The data cache supports memory references into the heap, the trail and the local stack. Cache data management typically deals with what is removed from a cache. But some data, particularly pointer chains, can cause a cache to be flushed unnecessarily. If the tag from a data word is used to prevent transient data, such as intermediate elements in a pointer chain, from entering a cache, then the cache hit rate will be improved. This implies that dereferencing should take place outside the cache, rather than at the chip (which was suggested in 1986 by Mats Carlson). If a "dereferenced choice point" (Mills 1986) is built once for a procedure, then the bottleneck that this would otherwise form can be reduced. Also, the only hardware interlock built into the LIBRA is

used to stall the pipeline during dereferencing. Moving dereferencing into the cache would clean up the design.

It is also possible to do a CDC-6600 like trick, and move trailing and failing out of the CPU to a peripheral processor. This allows the failure of a previous goal to overlap the execution of the subsequent goal, thus improving backtracking performance. As failure(s) occur, the trail-fail processor is passed the new trail stack bounds, and begins to untrail variables while the next clause is executed by the CPU. Implementing this overlap requires the trail-fail processor to monitor CPU requests for data. If an as-yet-untrailed value is requested by the CPU, the trail-fail processor must supply the reset value.

The instruction cache takes advantage of the locality of reference exhibited by Prolog procedures during head unification and tail recursion. The LOW RISC clause compiler produces in-line code for the head of a clause consisting of multiple blocks of three to ten instructions, all linked by forward references. An instruction cache that prefetches a 4 word block allows the LOW RISC to execute the head of a Prolog clause with few misses. Cache misses would occur when a goal was called, and at the termination of a clause.

5.3 CONDITIONAL INSTRUCTION EXECUTION AND MEMORY SUBSYSTEM DESIGN

Conditional instruction execution decreases the number of short branches by changing short sequences of "branch around" code into sequential (but possibly not executed) instructions. The LIBRA uses conditional instruction execution to implement preferred branches and to control the execution of every instruction in a manner similar to the Acorn RISC Machine (Acorn Computers Limited 1986, 1987). The LIBRA architecture extends this concept by adding symbolic conditions: certain variable types (bound1, bound2), trail and environment checks (trail1, trail2, env1, env2), and collision checks ("sticky overflow"). For examples of the use of conditional execution, see Appendix D, Emulating the WAM, in (Mills 88).

Conditioning instruction execution on these flags allows operations which formerly needed several test-and-branch instructions to be coded instead as a sequence of instructions, all of which are conditionally **nops**. This reduces the branch frequency of the LIBRA, and improves its ability to use interleaved memory (Figures 6, 7, and 8). Branch frequencies were evaluated from native-coded WAM instructions weighted by the dynamic frequency of occurrence of the WAM instructions (Dobry et. al. 1987).

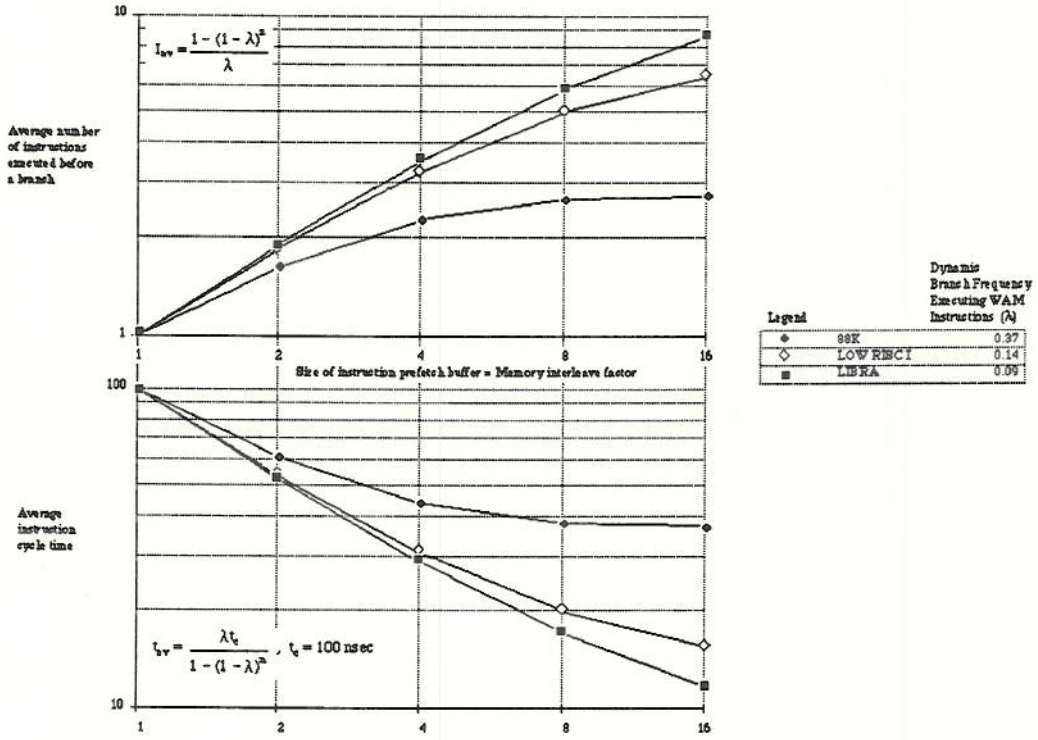


Figure 6. Instruction interleaving and cycle time

	1	2	4	8	16
88K	1.00	1.63	2.28	2.64	2.70
LOW RISC I	1.00	1.86	3.24	5.01	6.50
LIBRA	1.00	1.91	3.49	5.89	8.65

Figure 7. Average number of instructions executed before a branch

	1	2	4	8	16
88K	100	61	44	38	37
LOW RISC I	100	54	31	20	15
LIBRA	100	52	29	17	12

Figure 8. Average instruction cycle time

Overall, the use of conditional instruction execution, memory interleaving, tag-controlled caching, instruction prefetching and loop-buffering (lookahead, look-behind) allow the LIBRA to execute short inner loops and shallow backtracking at speeds limited primarily by the cache memory cycle time rather than the hit rate.

5.4 BOUNDS CHECKING AND TRAIL-CHECK SCOREBOARDING

Automatic bounds check are provided for trail and current environment checking. Some bounds checks are "sticky", to allow detection of an "almost stack collision" condition. Then garbage collection can be initiated at programmer defined points. This saves stack collision checking at every call, and further reduces pipeline breaks.

Scoreboarding allows the efficient use of a machine resource after all necessary conditions are met. Generalized scoreboarding manages the previous use of a resource as well, avoiding the later use of a resource at an inconvenient time. If an operation can be divided so that a resource can be used in advance, then the scoreboard can mark pre-processed data as well as data waiting to be processed. The LIBRA architecture uses generalized scoreboarding by performing bounds checks during a load on unbound variables. The trail check is moved to occur during a load or the final stage of a dereference. If the check showed that an unbound variable is being loaded, and it needs to be trailed if it is instantiated later, the register loaded is marked by setting a trail-check flag in the scoreboard. When the LIBRA executes a load instruction, it checks unbound variables against internal bounds registers, and stores the result of the check in a status bit associated with each register. Later, if the unbound variable is bound, the status bit is used to conditionally execute a trailing instruction (Mills 88).

5.5 PARTIAL UNIFICATION AND DEREFERENCING

Unification requires an execution sequence of five instructions if **compare** and **branch** instructions are used to build a "tree-structured" unifier. Because the **compare** instruction checks

only for equality or inequality of the tags it does not allow an easy way to identify the relationship between an unbound and a list, for example.

The solution is a new instruction which operates in a single cycle and is "complex" in that it encodes unification except in the case where a recursion is necessary. The instruction is based on the notion that, except for recursive unification, the other operations in a table-driven unifier are single instruction operations (Figure 9). The complexity of a table-driven unifier results from the need to determine the types of the two operands, and index into the rows and columns of the tables based on the types. If the tags of the two operands could be used to form an index into the microprogram, then the tag checking and indexing instructions could be eliminated.

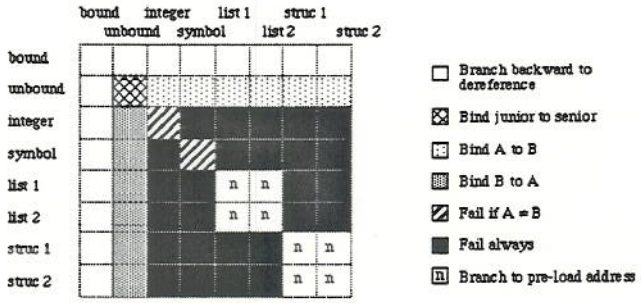


Figure 9. LIBRA partial unification

To accomplish this in the LIBRA the **partial unify** instruction is introduced which operates as follows:

0. During each **compare** instruction, the operand tags are concatenated and latched,
1. During each instruction decode the latched tags *and* the opcode are translated into a microprogram address,
2. If it is a **partial unify**, the tag microprogram address is used instead of the opcode microprogram address,
3. The control word for the pre-selected instruction is executed.

This allows the **partial unify** instruction to *replace* itself with any instruction's microcode word.

Typically the single-cycle partial unify performs either a **nop**, a **store**, a **call** or a **branch**, thus condensing three to five tag checking instructions into one. Although only one operation can be

performed by **partial unify**, it is enough to handle most strength-reduced unification in the Warren machine model. Partial unification can perform very efficient detection of special cases (such as two structure pointers being identical), which can reduce the overhead placed on the calling mechanism. This means that whenever it is possible to avoid a pipeline break for a **switch** or a **call** instruction, the **partial unify** instruction can do so. Thus, the partial unify instruction can eliminate as many as 30% of the subroutine calls performed by a general purpose RISC running Prolog.

5.6 TEMPLATE AND DIFFERENCE PROGRAM COUNTERS

The major problem affecting open-coded WAM logic programs is the expansion in code size, which can range from three to seven times larger. The LIBRA addresses this problem by providing hardware support for *templates*, or instruction sequences that have "holes" in them (Mills and Buettner 1988). During shallow backtracking, the LIBRA allows a template for a clause to be fetched, which is then executed repeatedly with the "holes" filled in by executing instructions from another instruction stream composed only of those instructions that differ from one clause to the next (Figure 10 and 11).

```
ir( min(X,Z,Z), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-

        sc( max(Y,Z,Y1), H1),
        sc( min(X,Y,X1), H2),
        sc( min(X,Y1,Z1), H3).

ir( min(X,Y,X1), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-

        sc( max(Y,Z,Y1), H1),
        sc( min(X,Z,Z), H2),
        sc( min(X,Y1,Z1), H3).

ir( min(X,Y1,Z1), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-

        sc( max(Y,Z,Y1), H1),
        sc( min(X,Y,X1), H2),
        sc( min(X,Z,Z), H3).
```

Figure 10. Original clauses marked with differences

```
ir( min(X,·,·), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-

        sc( max(Y,Z,Y1), H1),
        sc( min(X,·,·), H2),
        sc( min(X,·,·), H3).

Z,  Z,  Y, X1,  Y1,  Z1.
Y,  X1, Z, Z,   Y1,  Z1.
Y1, Z1, Y, X1,  Z,   Z.
```

Figure 11. Template clause and difference instructions

In the example shown here the native LIBRA code for the original clauses would require 120 instructions (Figure 10). Using the template and difference program counters of the LIBRA reduces this code to 34 instructions for the template and 3 x 6, or 18, difference instructions, a total of 52 instructions (Figure 11). Reduction factors range from 1.15 for unit clauses that differ greatly to more than 3 for clauses that are similar in all but one or two positions. The example has a reduction factor of 2.3.

6. PERFORMANCE EVALUATION

The performance resulting from this choice of instruction set, and the hardware support provided for it, is shown in the following diagram (Figure 12). The average instruction cycles for executing WAM instructions are plotted against the execution speed of each architecture in logical inferences per second x 1000 (KLIPS). The profile trees are shown for clarity and may be compared using the legend. From this we can conclude that the LIBRA runs logic programs 2.3 times faster than the PLM, with code optimizations such as template/difference compiling and conditionally omitting dereferencing using partial unification improving this even more. When the effects of interleaving are considered the LIBRA is faster by a factor ranging from 2.5 to 21.

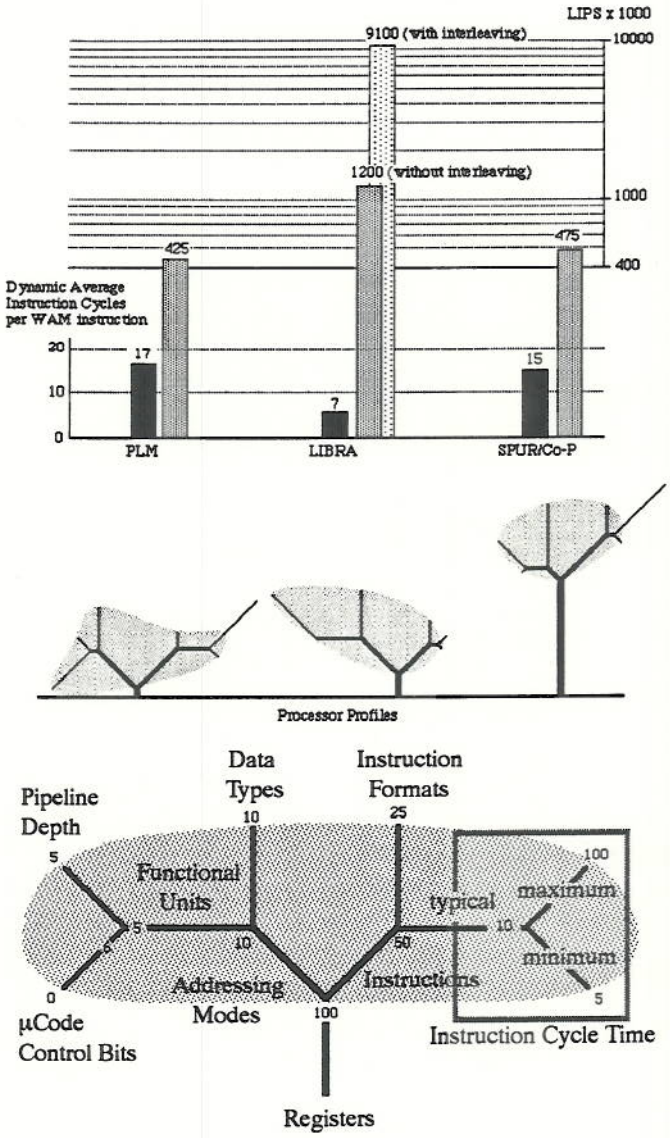


Figure 12. Logical Inferences Per Second x 1000 normalized to a 100 ns instruction cycle (LIBRA with 13ns cycle time with interleaving)

6.1 IMPROVING DEREFERENCING USING PARTIAL UNIFICATION

Prolog implementations quite often spend more time checking to see if a basic operation such as unification or trailing must be done than they take to do it. A Prolog program may spend as much as 20% of its time performing dereferencing, or checking to see if it is necessary [Ginosar 87]. The potential to increase the execution speed of a program by improving dereferencing exists because more than 99.3% of the Prolog objects that must be dereferenced may be reached in fewer

than two indirections, while 67% require none [Tick 88]. Although an argument may already be dereferenced the check to verify it appears to be unavoidable. In the original implementation partial unification must be "protected" against bound variables, because its operation in that case is undefined. Thus operands must always be dereferenced before use (Figure 13).

```
Label   Condition   Instr   sc  Operands                    Comment

enter:              sub     sc  Xn  r27     r27
loop1:  if bound1   if          always      loop1
        if br       ld      sc  Xn  0       Xn

                    sub     sc  Ai  r27     r27
loop1:  if bound1   if          always      loop1
        if br       ld      sc  Ai  0       Ai

                    unify   sc  Xn  Ai  exit    exit        ;no mode split
        if trail1   push+       TR  Xn
        if trail2   push+       TR  Ai
exit:
```

Figure 13. Always dereference operands before unification (■ on graphs in Figure 15)

However, the single-cycle partial unification instruction can be modified to improve the dereferencing behavior of Prolog programs. If the pair of checks for a bound variable, one for each operand being unified, are included in the operation of the partial unification instruction by enlarging the unifier table in ROM, up to eight instructions (including two branches) are removed from the direct execution sequence. The modified partial unify instruction thus reduces the number of instruction cycles spent dereferencing objects reached in eight or fewer indirections (Figure 14).

```
Label   Condition   Instr   sc  Operands                    Comment

loop:   if bound1   ld          Xn  0   Xn
        if bound2   ld          Ai  0   Ai
enter:              sub     sc  Xn  Ai  r27
                    unify   sc  Xn  Ai  loop    exit        ;no mode split
        if trail1   push+       TR  Xn
        if trail2   push+       TR  Ai
exit:
```

Figure 14. Never dereference operands before unification (□ on graphs in Figure 15)

When this optimization is evaluated using the dynamic frequency of dereference chains it improves the performance of unification from 60% to 100% (Figure 15).

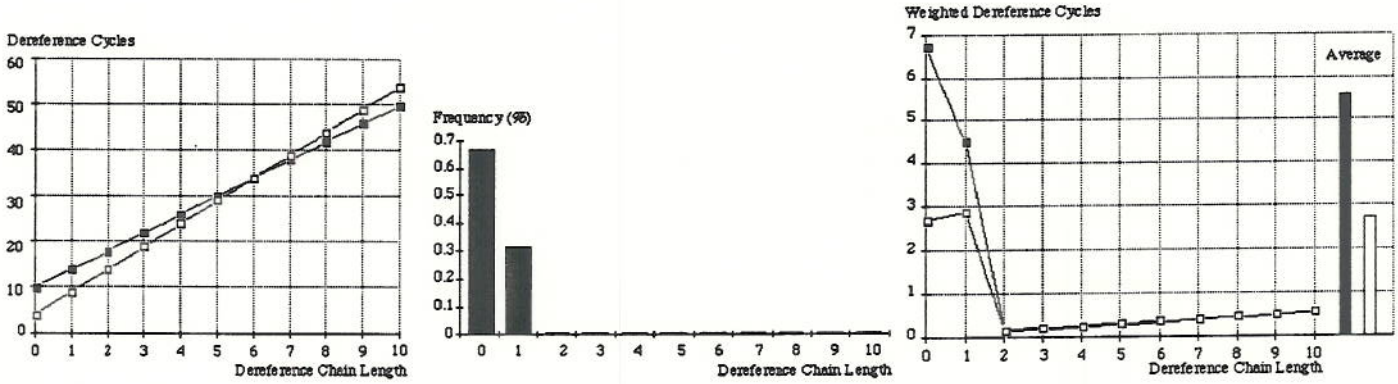


Figure 15. Performance improvement if operands are never dereferenced before unification

REFERENCES

Abe, T., Bandoh, T., Yamaguchi, S., Kurosawa, K, and Kiriyama, K. 1987. High performance integrated Prolog processor IPP. *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, pp. 100-107. Washington, D.C.: IEEE Computer Society Press.

Acorn Computers Limited 1987. *ARM datasheet*, Part No. 1 85250 03600 0, 23 January. Acorn Computers Limited, Cambridge, United Kingdom.

Applied Logic Systems, Inc. 1987. *IBM PC Prolog User's Manual.* Applied Logic Systems, Inc., Syracuse, New York.

Barbacci, M., and Siewiorek, D. 1982. *The design and analysis of instruction set processors*. New York: McGraw-Hill.

Borriello, G., Cherenson, A., Danzig, P., and Nelson, M. 1986. Special or general-purpose hardware for Prolog: A comparison. Report No. UCB/CSD 87/314, Computer Science Division (EECS), University of California, Berkeley, California.

Borriello, G., Cherenson, A., Danzig, P., and Nelson, M. 1987. RISCs vs. CISCs for Prolog: A case study. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, California. In ACM SIGPLAN Notices. 22:136-145.

Bowen, K.A., Buettner, K., Cicekli, I., and Turk, A. 1986. The design and implementation of a high-speed incremental portable Prolog compiler. *Proceedings Third International Conference on Logic Programming*, July 14–18 1986. In *Lecture Notes in Computer Science 225*, ed. E. Shapiro, Springer-Verlag, 1986.

Cheng, C.Y., Chen, C., and Fu, H. C. 1986. Design of LISCP: A fast RISC-style Prolog machine (Part I: Instruction set design and Part II: Basic machine design and performance evaluation). *Proceedings of the International Computer Symposium*, Tainan, Taiwan, R.O.C., pp. 472-485. n.p.

Cheng, C.Y., Chen, C., and Fu, H.C. 1987. RPM: A fast RISC style Prolog machine. *Proceedings VLSI and Computers, First International Conference on Computer Technology, Systems and Applications*, Hamburg, West Germany, pp. 95-98. Washington, D.C.: IEEE Computer Society Press.

Civera, P., Del Corso, D., Maddaleno, F., Piccinini, G., and Zamboni, M. 1988. A 32-bit processor for compiled Prolog. *Proceedings of the International Workshop on VLSI for Artificial Intelligence*, Oxford, England.

Davidson, J., and Vaughan, R.. 1987. The effect of instruction set complexity on program size and memory performance. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, California. In ACM SIGPLAN Notices. 22:60-64.

Despain, A., Patt, Y., Srini, V., Bitar, P., Bush, W., Chien, C., Citrin, W., Fagin, B., Hwu, W., Melvin, S., McGeer, R., Singhal, A., Shebanow, A., and Van Roy, P. 1987. Aquarius. *ACM SIGARCH Computer Architecture News*. 15: 22-34.

Dobry, T. 1987. *A high performance architecture for Prolog.* Report No. UCB/CSD 87/352. Computer Science Division (EECS), University of California, Berkeley, California.

Dobry, T., Patt, Y., and Despain, A. 1984. Design decisions influencing the micro-architecture for a Prolog machine. *Proceedings of the 17th Annual International Workshop on Microprogramming*, New Orleans, Louisiana, pp. 217-231. Washington, D.C.: IEEE Computer Society Press.

Dobry, T., Patt, Y., and Despain, A. 1985. Performance studies of a Prolog machine architecture. *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, Massachusetts, pp. 180-190. Washington, D.C.: IEEE Computer Society Press.

DuBose, D., Fotakis, D., and Tabak, D. 1987. A microcoded RISC. *ACM SIGARCH Computer Architecture News*. 14:5-16.

Eickmeyer, R., and Patel, J. 1987. Performance evaluation of multiple register sets. *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, pp. 264-271. Washington, D.C.: IEEE Computer Society Press.

Fagin, B., Patt, Y., Srini, V., and Despain, A. 1985. Compiling Prolog into microcode: A case study using the NCR/32-000. *Proceedings of the 18th Annual Workshop on Microprogramming*, Pacific Grove, California, pp. 79-88. Washington, D.C.: IEEE Computer Society Press.

Fisher, J. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*. C-30:478-490.

Flynn, M., Mitchell, C., and Mulder, J. 1987. And now a case for more complex instruction sets. *IEEE Computer*. 20(September):71-83.

Ginosar, R., and A. Harsat 1987a. "Profiling LOGIX: A step towards a flat concurrent Prolog processor." Department of Electrical Engineering, Technion - Israel Institute of Technology, Haifa, Israel (unpublished).

Ginosar, R., and Harsat, A. 1987b. "CARMEL: A VLSI architecture for flat concurrent Prolog." Department of Electrical Engineering, Technion - Israel Institute of Technology, Haifa, Israel (unpublished).

Hitchcock, C. and Sprunt, H. 1985. Analyzing multiple register sets. *Proceedings of the 12th Annual International Symposium on Computer Architecture*, Boston, Massachusetts, pp. 55-63. Washington, D.C.: IEEE Computer Society Press.

Ito, N., Sato, M., Kuno, E., and Rokusawa, K. 1986. The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D. *Proceedings of the 13th Annual International Symposium on Computer Architecture*, Tokyo, Japan, pp. 149-156. Washington, D.C.: IEEE Computer Society Press.

Katevenis, M. 1985. *Reduced instruction set computer architecture for VLSI.* Cambridge, Massachusetts: MIT Press.

Keller, R. 1976. Look-ahead processors. *ACM Computing Surveys*. 7:177-195.

Kogge, P. 1981. *The architecture of pipelined computers*. New York: McGraw-Hill.

Kogge, P. 1987. *The architecture of logic-based computing systems*. Reading, Massachusetts: Addison-Wesley (manuscript).

Lang, T., and Huguet, M. 1986. Reduced register saving / restoring in single-window register files. *ACM SIGARCH Computer Architecture News*. 14:17-26.

Matsumoto, H. 1985. A static analysis of Prolog programs. *ACM SIGPLAN Notices*. 20: 48-59.

McFarling, S., and Hennessy, J. 1986. Reducing the cost of branches. *Proceedings of the 13th Annual International Symposium on Computer Architecture*, Tokyo, Japan, pp. 396-403. Washington, D.C.: IEEE Computer Society Press.

McNeley, K., and Milutinovic, V. 1987. Emulating a complex instruction set computer with a reduced instruction set computer. *IEEE Micro*. 7: 60-71.

Mills, J. 1985. "A description of the operation of the Warren abstract Prolog machine using a RISC-like instruction set." Private communication to K.A. Bowen.

Mills, J. 1986. An implementation of the Warren abstract Prolog machine for segmented memory architectures. Technical Memo TM-44, Argonne National Laboratory.

Mills, J. 1987. Coming to grips with a RISC: A report of the progress of the LOW RISC design group. *ACM SIGARCH Computer Architecture News*. 15: 53-62.

Mills, J. 1988. "LIBRA: A high performance RISC for Prolog."

Mills, J. n.d. "A high performance LOW RISC machine for logic programming." *Journal of Logic Programming* (accepted 1987).

Moto-oka, T., et. al. 1982. Challenge for knowledge information processing systems (Preliminary report on fifth generation computing systems). In *Fifth Generation Computing Systems*, pp. 1-32. Edited by T. Moto-oka. Amsterdam: North-Holland.

Myers, G. 1982. *Advances in computer architecture*, 2nd. ed. New York: John Wiley & Sons.

Nakazaki, R., Konagaya, A., Habata, S., Shimazu, H., Umemura, M., Yamamoto, M., Yokota, M., and Chikayama, T. 1985. Design of a high-speed Prolog machine (HPM). *Proceedings of the 12th Annual International Symposium on Computer Architecture*, Boston, Massachusetts, pp. 191-197. Washington, D.C.: IEEE Computer Society Press.

Onai, R., Shimizu, H., Masuda, K., and Aso, M. 1986. Analysis of sequential Prolog programs. *Journal of Logic Programming*. 2:119-141.

Patterson, D. 1985. Reduced instruction set computers. *Communications of the ACM*. 28: pp. 8-21.

Patterson, D. 1987. A progress report on SPUR: February 1, 1987. *ACM SIGARCH Computer Architecture News*. 15:15-21.

Patterson, D. and Sequin, C. 1982. A VLSI RISC. *IEEE Computer*. 15(September):8-21.

Ross, M.L. and Ramamohanarao, K. 1986. Paging strategy for Prolog based dynamic virtual memory. *Proceedings of the 1986 Symposium on Logic Programming*, Salt Lake City, Utah, pp. 46-57. Washington, D.C.: IEEE Computer Society Press.

Seo, K., and Yokota, T. n.d. "Pegasus: A RISC processor for high-performance execution of Prolog programs." (Unpublished).

Short, B. 1987. "A Preliminary Evaluation of the LOW RISC." Computer Science Department, Arizona State University, Tempe, Arizona (unpublished).

Short, B. 1988. "Extending a Reduced Instruction Set Computer to Support Prolog." Master's report, Department of Computer Science, Arizona State University, Tempe, Arizona.

Srini, V. 1985. "VLSI-PLM chip." Laboratory note, University of California, Berkeley, California (unpublished).

Taki, K., Nakajima, K., Nakashima, H., and Ikeda, M. 1987. Performance and architectural evaluation of the PSI machine. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, California. In *ACM SIGPLAN Notices*. 22:128-135.

Taki, K., Yokota, M., Yamamoto, A., Nishikawa, H., and Uchida, S. 1984. Hardware design and implementation of the personal sequential inference machine (PSI). *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, ICOT, pp. 398-409. n.p.

Tamura, N., Wada, K., Matsuda, H., Kaneda, Y., and Maekawa, S. 1984. Sequential Prolog machine PEK. *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, ICOT, pp. 542-550. n.p.

Taylor, G., Hilfinger, P., Larus, J., Patterson, D., and Zorn, B. 1986. Evaluation of the SPUR Lisp architecture. *Proceedings of the 13th Annual International Symposium on Computer Architecture*, Tokyo, Japan, pp. 444-452. Washington, D.C.: IEEE Computer Society Press.

Tick, E. 1985a. Lisp and Prolog memory performance. Research Paper 86-291, Computer Systems Laboratory, Stanford University, Stanford, California.

Tick, E. 1985b. Prolog memory-referencing behavior. Research Paper 85-281, Computer Systems Laboratory, Stanford University, Stanford, California.

Tick, E. 1987. Studies in Prolog architectures. Technical Report No. CSL-TR-87-329, Computer Systems Laboratory, Stanford University, Stanford, California.

Tick, E. 1988. Data buffer performance for sequential Prolog architectures. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, pp. 434-442. Washington, D.C.: IEEE Computer Society Press.

Tick, E. and Warren, D.H.D. 1984. Towards a pipelined Prolog processor. *1984 International Symposium on Logic Programming*, Atlantic City, New Jersey, pp. 29-40. Silver Spring, Maryland: IEEE Computer Society Press.

Warren, D. H. D. 1983a. An abstract Prolog instruction set. Technical Note 309, SRI International, Stanford, California.

Warren, D. H. D. 1983b. Applied logic - Its use and implementation as a programming tool. Technical Note 290, SRI International, Stanford, California.

APPENDIX A. EXAMPLE WAM INSTRUCTIONS CODED USING THE LIBRA

call

```
          call                  calladdress29
          add                   r0  envsize     N
```

execute

```
          goto                  address29
```

proceed

```
          ret                   CP0
```

put_variable Xn

```
          push+ & ldref             H   unb:H   bnd: Ai
          add                   Ai  0   bnd: Xn
```

put_value Xn

```
          add                   Xn  r0  Ai
```

put_unsafe_value Yn

```
          drfmem            sc  E   Yoffset     Ai
          if                    not currenv     next_macro
   if nobranch      st           sc  Ai  0   bnd: H
   if trail         push+            TR  Ai
          push+ & ldref         H   unb: H  bnd: Ai
```

put_constant C, Ai

```
          add                   r0  const16     con: Ai
```

put_structure F, Ai

```
          push+ & ldref             H   con: fn16   struc: Ai
```

put_list Ai

```
          add                   H   0   list: Ai
```

get_variable Yn, Ai

```
          st                    E   Yoffset     Ai
```

get_variable Xn, Ai

```
          add                   Ai  r0  Xn
```

get_value Xn, Ai

```
          drf                   Xn  T1
          drf                   Ai  T2
          sub               sc  T1  T2  r0
          unify             sc  T1  T2  $+2     (no mode splitting)
   if trail1        push+            TR  T1
   if trail2        push+            TR  T2
```

get_constant C, Ai

```
          drf                Ai  T1
          add                r0  C16 con: T2
          sub            sc  T1  T2  r0
          unify          sc  T1  T2  $+2     (no mode splitting)
   if trail1       push+              TR  T1
```

get_structure F, Ai

```
          drf            sc  Ai  T1
          switch             fail    fail    readmode (mode split)
   if var st             sc  T1  0   struc: H
   if trail1       push+              TR  T1
          push+              H   con: F16

   readmode:
          pop+               T1  F
          sub            sc  F   con: F16    r0
          if t≠ or v≠        fail
```

get_list Ai

```
          drf            sc  Ai  T1
          switch             fail    readmode    fail       (mode split)
   if var st             sc  T1  0   struc: H
   if trail1       push+              TR  T1

   readmode:
          < T1 contains S >
```

unify_variable Xn

```
   writemode:
          push+ & ldref          H   unb: H  bnd: Xn

   readmode:
          pop+               S   Xn
```

unify_value Xn

```
   writemode:
          push+              H   Xn

   readmode:
          pop+ & drf         S   T1
          drf                Xn  T2
          sub            sc  T1  T2  r0
          unify          sc  T1  T2  $+2     (no mode splitting)
   if trail1       push+              TR  T1
   if trail2       push+              TR  T2
```

unify_constant C

```
   writemode:
          push+              H   con: C16

   readmode:
          pop+ & drf         S   T1
          add                r0  con: C16    T2
          sub            sc  T1  T2  r0
          unify          sc  T1  T2  $+2     (no mode splitting)
   if trail1       push+              TR  T1
```

try_me_else L

```
        push+           B   A1
        push+           B   A2
        push+           B   A3
        push+           B   A4
        push+           B   A5
        push+           B   A6
        push+           B   A7
        push+           B   A8
        push+           B   CP
        push+           B   TR
        push+           B   E
        push+           B   H
        ldhi                LaddrHi
        push+           B   LaddrLo
        st              HB  0   H
        st              EB  0   E
```

retry_me_else L

```
        ldhi                LaddrHi
        add             r0  LaddrLo     T1
        st              B   -1  T1
```

trust_me_else fail

```
        ld              B   -14 T1
        ld              B   -15 T2
        sub             B   12  B
        st              HB  0   T1
        st              EB  0   T2
```

APPENDIX B. PERFORMANCE COMPARISON

		PLM		LIBRA	
WAM Instruction	Frequency (%)	cycles	weight	cycles	weight
unify_variable_X (read)	8.8	5	37.40	1	13.20
unify_variable_X (write)		3		1	
unify_variable_Y (read)		6		2	
unify_variable_Y (write)		3		2	
get_list (set read)	7.27	8	72.70	3	25.45
get_list (set write)		12		4	
unify_cdr (read)	6.88	6	34.40		0.00
unify_cdr (write)		4			
unify_value_X (read)	4.96	23	71.92	6	18.60
unify_value_X (write)		3		1	
unify_value_Y (read)		26		6	
unify_value_Y (write)		6		2	
escape - various	4.9				
switch_on_term	4.87	6	29.22	2	9.74
unify_nil (read)	4.86	6	19.44	4	12.15
unify_nil (write)		2		1	
get_structure (set read)	4.11	11	49.32	6	22.61
get_structure (set write)		13		5	
execute	4.01	1	4.01	1	4.01
allocate	3.47	11	38.17	6	20.82
get_variable_X	3.44	2	8.60	1	3.44
get_variable_Y		3		1	
unify_constant (read)	3.33	14	26.64	5	9.99
unify_constant (write)		2		1	
deallocate	2.87	6	17.22	2	5.74
put_constant	2.71	2	5.42	1	2.71
proceed	2.65	1	2.65	1	2.65
try_me_else	2.45	20	49.00	16	39.20
call	2	1	2.00	2	4.00
cut	1.85	10	18.50	2	3.70
get_constant	1.83	11	20.13	5	9.15
put_variable_X	1.79	4	6.27	2	2.69
put_variable_Y		3		1	
get_value_X	1.44	12	18.72	6	8.64
get_value_Y		14		6	
trust_me_else	1.32	5	6.60	5	6.60
get_nil	1.29	11	14.19	5	6.45
put_unsafe_value	1.24	10	12.40	4	4.96
retry_me_else	0.88	2	1.76	3	2.64
switch_on_structure	0.866	13	11.26		0.00
put_list	0.769	3	2.31	1	0.77
try	0.711	20	14.22	20	14.22
fail	0.564	23	12.97	19	10.72
trust	0.35	5	1.75	8	2.80
unify_void	0.324	6	1.94	2	0.65
switch_on_constant	0.201	10	2.01		0.00
retry	0.0593	2	0.12	6	0.36
put_structure	0.052	4	0.21	1	0.05
unify_local_value	0.0127	10	0.13	4	0.05
put_nil	0.00267	2	0.01	1	0.00

		PLM		LIBRA	
	Frequency (%)	cycles	weight	cycles	weight
Overall	99.83	392	645.69	186	279.44