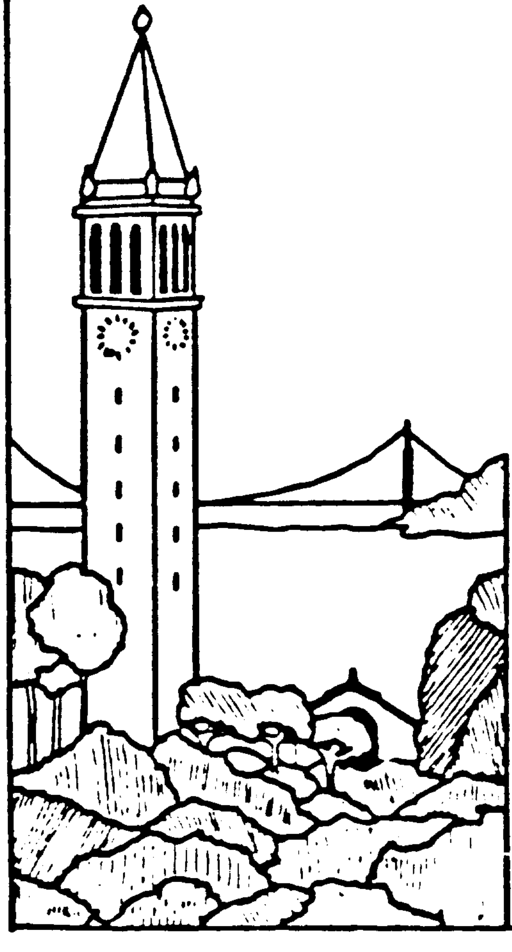
Special or General-Purpose Hardware for Prolog: A Comparison

Gaetano Borriello
Andrew Cherenson
Peter Danzig
Michael Nelson



Report No. UCB/CSD 87/314

October 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Special- or General-Purpose Hardware for Prolog: A Comparison

Gaetano Borriello
Andrew Cherenson
Peter Danzig
Michael Nelson

Report No. UCB/CSD 87/314
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

ABSTRACT

This study compares the performance of executing Prolog code on the Berkeley PLM processor (a special-purpose CISC architecture) and the Berkeley SPUR processor (a general-purpose RISC architecture with tagged data). Fourteen standard benchmark programs were run on both the PLM and SPUR simulators. The two implementations were compared with regard to static and dynamic program size, execution speed, and cache performance. The simulated memory system included a direct-mapped mixed instruction and data cache. We found that, on average, the macro-coded SPUR implementation has a static code size 14 times larger than the PLM, executes 16 times more instructions, yet requires only 2.31 times the number of machine cycles. To have the same miss ratio with a much larger code size the SPUR implementation requires a cache that is 4 to 8 times that of the PLM. We also suggest minor changes to the SPUR instruction set to improve its Prolog execution and outline the design of a special-purpose SPUR coprocessor that would greatly reduce the code size and double SPUR's Prolog performance.

1. Introduction

Logic programming has been the subject of much attention since the 1981 announcement that the Japanese Fifth Generation Project [Moto-oka83] would use Prolog as its principal programming language. Many computer researchers in the artificial intelligence community believe that logic programming provides a more direct and natural mapping of problem specifications into machine language than traditional high-level languages. In the past few years, much work has been done to develop high-performance machines for logic programming.

Prolog is the most popular of the logic programming languages. It was designed at the University of Marseille around 1970 by Alain Colmerauer and his associates. In 1977, David Warren at the University of Edinburgh developed the first compiled implementation of Prolog [Warren77]. The compiler ran on a DECsystem-10 and was dramatically faster than previous, interpreted implementations. Since then, many approaches, ranging from advanced compiler techniques to microcoded hardware enhancements, have been used to improve the performance of compiled Prolog code.

Most compiled implementations of Prolog are based on refinements to Warren's original abstract machine (WAM) specification [Warren83]. Warren's instruction set corresponds very closely to the tokens of the Prolog language. Compiling from Prolog to WAM is therefore a simple and straight-forward process that can reasonably be implemented in Prolog itself.

The Berkeley Prolog Machine (PLM) is a special-purpose microcoded processor that uses a slightly modified version of the WAM instruction set [Dobry84c]. Efficient Prolog execution is achieved through the much higher code density of the PLM as compared to conventional, general-purpose architectures. The PLM is expected to run Prolog ten times faster than the compiled implementation for the DEC-2060. The PLM is part of the larger Berkeley Aquarius project whose aim is to build a 16 processor Prolog multiprocessor with a shared synchronization memory [Dobry85].

The Berkeley SPUR (Symbolic Processing Using RISCs) project aims to produce a multiprocessor personal workstation for high-performance general-purpose processing with some support for Lisp and floating-point computation [Hill86]. The SPUR microprocessor is a reduced instruction set computer (RISC) with extensions for tagged data types and a large mixed instruction and data cache. It includes a tightly-coupled coprocessor interface. The first coprocessor to be implemented will be used for high-performance IEEE standard floating-point operations.

Our objective is to show how well Prolog programs can be executed on SPUR, a processor not designed with logic programming applications in mind. We did not compile Prolog directly into SPUR machine code but instead used the output of the PLM compiler and performed a macro-expansion of the PLM/WAM instructions into SPUR instructions. Improvements in performance could certainly be gained by building a Prolog compiler for the SPUR architecture. We chose to use a macro-expansion technique so as to save time (there was no Prolog implementation for SPUR) and also to better compare the two architectures rather than the difference between two compilers. We feel we have achieved our objective of finding a lower bound for Prolog execution using macro-expansion with a few straight-forward optimizations.

If Prolog runs efficiently on SPUR, then Prolog programs can be easily integrated with an operating system, floating point hardware, and other applications programs to create a test-bed for experiments in mixed-paradigm programming systems. Although both SPUR and PLM are the basic elements of larger multiprocessor machines, we did not consider the issues of parallelism inherent in Prolog on either of these two architectures.

The next two sections provide background information on the PLM and SPUR architectures. We then discuss how PLM instructions were translated into SPUR instructions using macro-expansion. Section 4 considers tradeoffs with respect to register allocation, stack usage, and the prolog unification operations in mapping PLM to SPUR. Section 5 presents our performance comparisons results including static and dynamic program size, execution speed, and memory cache effects. We conclude with suggestions to improve Prolog performance with slight modifications to the SPUR architecture or with the use of a special coprocessor.

The appendices contain the programs that perform the macro-expansions, the macro-expansions, and the details of a possible implementation of a Prolog coprocessor for SPUR.

2. The PLM Architecture

The Berkeley PLM is a TTL implementation based on the Warren Abstract Machine, the target machine of the first Prolog compiler [Warren77]. Warren implemented a compiler that translates Prolog into abstract machine instructions that were then macro-expanded into DEC-10 machine code. Previous Prolog implementations were interpreters that were usually written in Lisp, and therefore suffered from the inefficiency of being translated twice, once from Prolog to Lisp and again from Lisp to machine code.

Warren's abstract machine is the basis of most of the work being done on special-purpose Prolog hardware. This section describes WAM as modified by Tick and Warren [Tick83, Warren83] and by the Berkeley PLM group [Dobry84b, Fagin85]. Dobry gives detailed descriptions of the PLM in [Dobry84a, Dobry84c, Dobry85]. Clocksin and Mellish [Clocksin81] provide a good foundation for the basics of Prolog and logic programming.

2.1. PLM Data Types and Representation

The PLM supports four (tagged) data types: constants, variables, lists, and structures. Constants have a minor type of nil, integer, atom, or floating point. Variables, actually pointers to other data structures, are bound (point to a data cell) or unbound (point to themselves). Lists and structures are cdr-coded to eliminate pointers to successive locations (car and cdr cells are distinguished by using a tag bit). This technique can eliminate up to half the amount of memory necessary to represent lists and structures and many pointer dereferences when a the elements of a list can be kept in a contiguous group of memory locations. Figure 1 summarizes the PLM data types and illustrates their layouts. It is important to note that the PLM tags consist of three orthogonal fields: type, sub-type, and cdr (or bound) bit. All tags also have a bit used by a garbage collection algorithm.

2.2. PLM Registers and Data Structures

The PLM organizes memory into

- a code segment;
- five stacks, consisting of the environment stack, choice point stack, "heap", trail stack, and the "push down list";
- sixteen data registers;
- a few mode bits.

These features are described below.

2.2.1. Environment Stack

The environment stack contains activation records of active Prolog clauses. An activation record consists of a pointer to the previous record, a code space address to jump to should the clause succeed, the clause argument count, room to store the clause's local variables, and a pointer to the last choice point entry

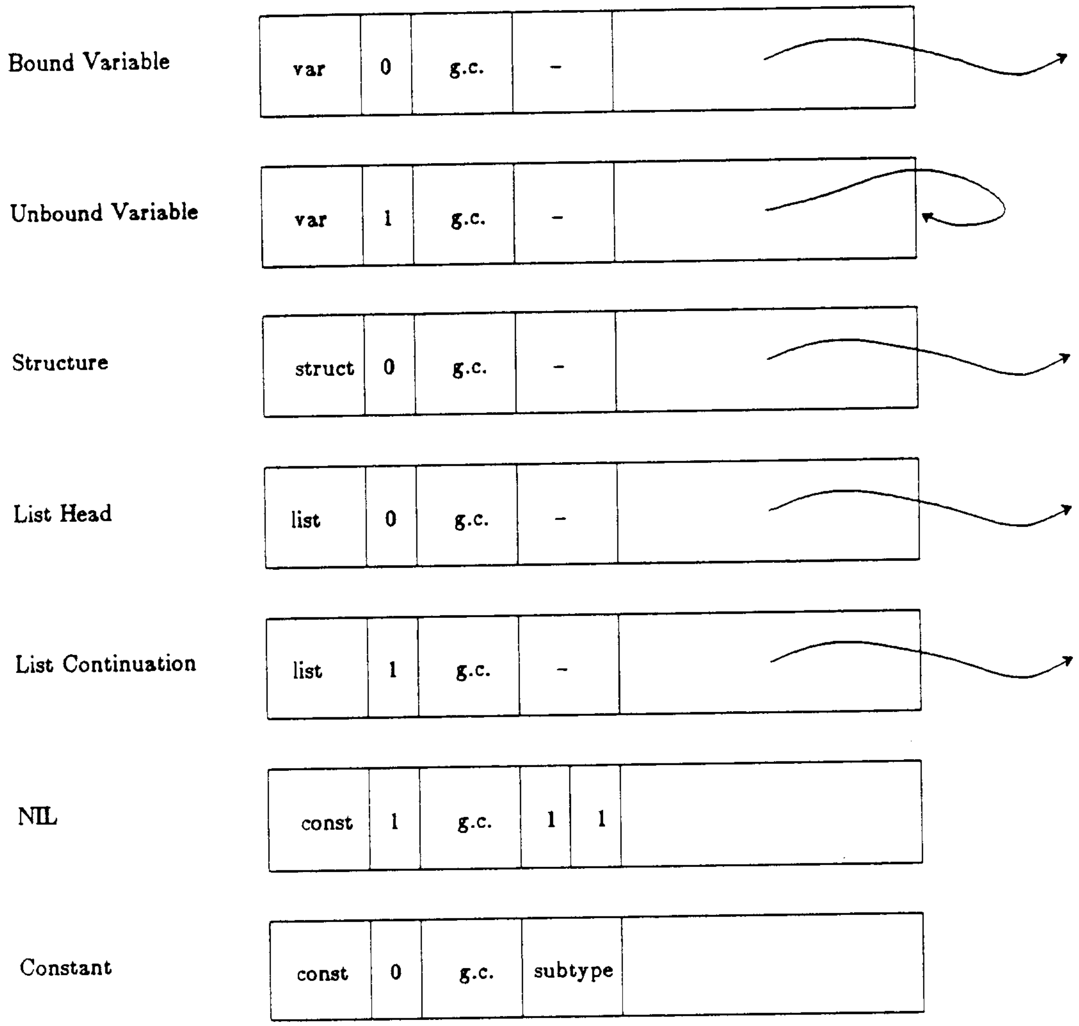


Figure 1. PLM data types. Tags consist of a two-bit field for the type of data: variable, structure, list, or constant. Variables contain a one-bit field indicating whether they are bound (and point to another data cell) or unbound (and point to themselves). This bit is always zero in structure pointers. In list pointers this field distinguishes car from cdr cells in the cdr-coded data structure supported by the PLM. This bit is also set to indicate a constant is nil. Constants also have another two-bit subtype field. All tags have a bit used by a garbage collection algorithm.

should the clause use the cut operator. (The cut operator increases backtracking efficiency by pruning branches from the depth-first search tree.)

2.2.2. Choice Point Stack

This stack contains procedure choice points. A choice point is a set of 15 PLM registers containing the necessary state to backtrack to a previous node in the search tree. It consists of the original procedure arguments, a pointer to the environment of the calling procedure, a pointer to the top of the heap at the time the procedure was invoked, a pointer to the top of the trail stack at procedure invocation time, a code space address should the procedure succeed, a code space address should backtracking be necessary, and a pointer to the previous choice point (this is needed since the PLM interleaves the choice point stack and the environment stack in the same stack segment).

2.2.3. Heap

Space for dynamically constructed lists and structures is sequentially allocated from the heap, which behaves very much like a stack. Heap space is reclaimed when a procedure fails, but must be garbage-collected periodically since data structures created by successful clauses would not otherwise be reclaimed.

2.2.4. Trail Stack

As Prolog programs perform their pattern matching functions, variables in one data structure are "bound" (i.e. made to point) to the corresponding element of the second data structure. In order to restore the computation's state when a clause fails (pattern matching fails) and backtracking occurs, all variable bindings must be reversible. The trail contains pointers to variables on the heap that have become bound during procedure execution. On goal failure, all trail entries above the saved heap pointer stored in the choice point are read and the variables they point to are unbound.

2.2.5. Push Down List

The push down list is a high-speed stack inside the PLM and is used to store pointers into two data structures that are being unified (i.e. pattern matched). Since lists or structures can contain embedded lists or structures as elements, unification is a recursive process. List or structures are unified in a depth-first, post-order tree traversal. The push down list is an optimization to reduce the complexity of managing another stack in memory and to increase unification performance.

2.2.6. Registers

Finally, the PLM has the following special-purpose registers: a program counter register (P); a goal success program counter (CP); pointers to the top of the environment, choice point, trail, and heap stacks (E, B, Tr, and H respectively); a structure pointer register (S); the procedure argument count register (N); eight argument registers (AX1-AX8); and two bits of control state called the cut bit and the read/write mode bit.

2.3. PLM Instructions

PLM instructions fall into eight classes:

- procedure control instructions that manipulate choice points;
- indexing instructions that perform multi-way branches depending on the type and value of an argument register;
- clause control instructions that manipulate activation records on the environment stack;
- get and put instructions that verify and prepare goal arguments respectively; and
- unify instructions that construct and compare structures and lists one element at a time by pattern matching the corresponding elements.

Instruction lengths range from 1 to 6 bytes. Tailoring the instruction set to the language produces high code density for the PLM. Smaller program sizes result in much improved instruction buffer and cache performance. This will be shown to be the primary reason for the PLM's excellent performance. However, instruction fetching and decoding is greatly complicated due to the variable-length unaligned instruction format.

The PLM's instruction buffer performs the bulk of the instruction prefetching and decoding functions. Instructions are broken up into an 8-bit opcode field, a 32-bit first argument field, and for instructions of more than one argument there is an additional 32-bit field containing the two single-byte second and third arguments. These are presented to the central processor for final decoding and execution. There is only a small set of conditional branch instructions in the PLM. The instruction buffer stops prefetching when one of these instructions is encountered and simply waits for the branch to be resolved. The PLM instruction buffer typically contains from five to sixteen instructions.

2.4. Stack Allocation Optimizations for Prolog

Warren included two more memory saving optimizations in his original implementation: environment trimming and tail recursion elimination. Environment trimming frees space from the activation record of a clause as soon as a variable is no longer referenced. This requires an additional field to the call instruction so the size of the activation record can be updated as each subclause of a clause is invoked. Larger memory sizes have probably made this optimization unnecessary.

Tail recursion elimination discards the activation record of a clause before the invocation of the last subclause. This optimization is quite valuable in recursive procedures that would otherwise quickly fill up stack space with unused activation records. The only restriction imposed by this method is that recursive clauses should be purely tail recursive. This condition is usually true in practice and can be enforced in almost every case. However, this important optimization does require a special PLM instruction to move needed variables from the activation record to the heap. The activation record's registers can replace those of the parent clause since failure of the last recursive clause implies a failure of the parent clause, popping both records off the stack.

2.5. System Support Functions

PLM instructions are complex since they execute in an indeterminate amount of time (e.g. recursive unification) and they can generate an indeterminate number of memory references per instruction. The first point makes instructions difficult to restart. A large amount of micro-engine state must be preserved between instructions. The second point implies that page faults may occur during the execution of a long instruction. In the current architecture, the PLM is a coprocessor to an NCR-32 main processor that handles memory management and process scheduling functions for the PLM. It is important to note that the PLM cannot be easily context-switched to another Prolog process while a page request is being serviced.

The NCR host processor not only provides virtual memory support but also performs I/O system calls and floating point operations for PLM escape instructions. However, these operations are expensive since they must be performed by a loosely-coupled coprocessor reached through the system bus rather than a tightly-coupled coprocessor reached through a direct interface.

3. The SPUR Architecture

The SPUR (Symbolic Processing Using RISCs) architecture is a RISC architecture augmented with special support for LISP processing and floating-point computation. The added capabilities include tagged data types and a tightly-coupled coprocessor interface. SPUR has been designed as a multiprocessor workstation. It has a 128KB cache that maintains data coherency by using hardware support for bus snooping. SPUR extends the work of the earlier Berkeley RISC and SOAR architectures [Katevenis83, Ungar84]. A summary of SPUR and PLM features is listed in Table 1.

3.1. SPUR Registers and Tags

The basic instruction and data word is 32 bits, however, registers are 40 bits wide so as to support an 8-bit data tag. Data is always word-aligned. Tagged data is stored in 2 words containing a total of 64 bits: the first word is the data and the second is the tag. Although the SPUR system bus supports only 32-bit transfers to the processor cache, all other busses are 40 bits wide. Therefore, a penalty for tagged data transfers is only incurred when data is brought into the cache or written back out to memory.

Comparison of PLM and SPUR		
Features	PLM	SPUR
architecture	tagged CISC	tagged RISC
target languages	Prolog only	LISP, C & others
instruction size	1 to 6 bytes	4 bytes
cycles per instr. (no misses) †	1 to 26+	1 (2 for stores) 4+ for floating-point ops
avg. cycles/instr.	7	1
cycle time	100–150 ns	100–140 ns
registers	9 special 8 argument	138 GPRs in 8 overlapped windows (10 global, 6 input, 10 local, and 6 output per window)
cache	separate I&D, 16KB each	mixed I&D, 128KB
instr. buffer size	5 to 16 instructions	128 instructions
microcode size	1K x 134 bits	not applicable

Table 1. Summary of features of the PLM and SPUR architectures.
† This assumes a perfect memory (i.e. no cache or instruction buffer misses) for both PLM and SPUR. For SPUR, load instructions are assumed to be followed by a non-dependent instructions (as is the case in our implementation). SPUR store intructions stall the pipeline and hence require two cycles.

There are 138 general-purpose registers in SPUR organized into 8 overlapped windows of 32 registers. Of these 32, there are 10 global registers accessible in all windows, 6 input registers that overlap with the previous window, 10 local registers that are accessible in only one window, and 6 outputs that overlap with the next window. When a function call is executed, the CPU shifts its current window. Input parameters and output results are passed between functions by using the overlapped registers. All registers are general-purpose except one of the global registers that is hard-wired to a zero value.

Special trap conditions are signaled when the windows overflow (i.e. a function call depth of more than 8 is reached) or underflow and a trap handler is used to move registers to and from memory. When a window is available, function calls proceed very quickly, only when windows overflow or underflow are memory accesses to preserve register state necessary. Halbert and Kessler have shown that window overflows occur less 1% of the time for a window size of 8 [Halbert80].

3.2. SPUR Instructions

Instructions fall into three basic types. Most instructions are register-to-register operations involving the entire 40-bit quantity. Special exceptions are signaled depending on the value of the tags (e.g. adding two pointers). There are also instructions that cause operations to be performed on coprocessor registers (e.g. floating-point add).

The load-to-register and store-from-register instructions are the only instructions that access memory. There are instructions for making either 32-bit untagged data and instruction access as well as full 40-bit tagged access. Specialized instructions are provided for moving data between the cache and the coprocessor registers in sizes corresponding to the IEEE floating-point standard and for transfers between processor and coprocessor registers. Lastly, there are comparison and branch instructions that alter control flow depending on the value of a CPU register, coprocessor register, or data tag.

SPUR is equipped with an on-chip instruction buffer of 128 words. Instructions are prefetched but compete with load and store instructions for access to the mixed instruction and data cache. All instructions are aligned and have uniform argument format, consequently, no alignment or predecoding is necessary. As a result the instruction buffer is a straight-forward on-chip instruction cache.

3.3. Pipeline Execution

SPUR has a four-stage pipeline: instruction fetch, register read and an ALU operation either to combine operands or compute and effective address, memory access (for load and store instructions) or nothing (for register-to-register instructions), and register write. The pipeline includes circuitry for forwarding data that may be required by the next instruction and not yet stored back into a register. The effective throughput is therefore one instruction per cycle.

Due to the structure of the pipeline, the effects of load and branch instructions are delayed one cycle beyond their execution. The contents of a register that is loaded from the cache cannot be accessed by the next instruction. It is necessary to include some operations that do not use that register or a NOP in the slot immediately following the load instruction. A similar situation holds for branch instructions. The instruction immediately following a branch will be executed whether or not the branch is taken. SPUR provides special branch instructions that cancel the effects of the subsequent instruction when the branch is taken. However, carefully placement of instructions in the slot following a branch can greatly improve throughput.

3.4. SPUR Address Spaces

SPUR has a 38-bit global virtual address space and 32-bit process virtual address space. The high-order two bits of the address select one of four segments and the remaining 30 bits are an offset into the segment. Typically, the first segment contains the operating system and the other three contain the process's code, data, and stack segments, respectively. During address translation, the 2 bit segment number is used to index a set of four segment registers each of which contains an 8-bit global segment number that selects one of 256 segments. The 8-bit segment number is concatenated with the 30-bit offset to form the 38-bit global virtual address.

3.5. Coprocessor Interface

SPUR supports a tightly-coupled coprocessor model. The CPU initiates all data transfers between the coprocessor registers and the cache using a 64-bit wide data bus. All instruction dispatching is performed by the CPU. When the CPU fetches an instruction that is not for the main processor it forwards it to the coprocessor.

The coprocessor interface consists of a set of lines for communicating the coprocessor instruction opcode and register arguments. One control signal indicates to the coprocessor that a new instruction is being presented by the CPU and another indicates the coprocessor has completed execution. When the coprocessor requires more than one cycle for instruction execution it suspends the CPU pipeline by asserting a coprocessor busy signal.

Coprocessors can also operate in parallel with the CPU. All responsibility for waiting the appropriate amount of time for results to be available rests with the CPU. An extra bit in the interface is used to select one of two coprocessors that could both be used in parallel.

3.6. System Support Functions

SPUR handles all its own traps and page faults. For this reason, unlike the PLM, all instructions are restartable and atomic in their operation. The SPUR processor has all the general-purpose instructions and trap handling capability to directly support an operating system. The PLM would require additional microcode support these functions.

4. Implementing Prolog on SPUR

This section describes the mechanics of macro-expanding Prolog into SPUR assembly language and how the state of the PLM is mapped onto the SPUR registers.

4.1. Macro-expanding PLM on SPUR

We chose macro-expanding PLM instructions rather than writing a full Prolog compiler for SPUR because its simplicity enabled us to find a lower bound on SPUR Prolog performance in just a few weeks. Undoubtedly, a compiler would achieve much better performance. In this section, we review the design alternatives we considered to represent the PLM's state and describe the design we chose to implement. We also describe the tools we developed to automatically macro-expand PLM instructions to SPUR code.

4.1.1. Choice Points, Environments, and Registers

Many of the PLM registers point into its multiple data and activation record stacks. We chose a register allocation scheme that follows the tenet that the optimal register layout is the one that reduces the processor-memory bandwidth. Since choice points are much larger than environment activation records, we decided to exploit SPUR's register windows for choice point buffering. Register windows cannot be used to represent the environment and heap structures since it must be possible to bind to these structures and it would be extremely complex to bind to registers (registers don't have a memory address). Using them as a trail buffer is difficult since trail entries are single words and really require a hardware stack rather than SPUR's overlapping register windows. Also, since access to the trail is strictly LIFO, any buffering scheme would only eliminate a single store and a single load per entry. The Berkeley PLM research group estimated that trail buffering in hardware could at best yield a 1% performance gain [Dobry84c].

Our register usage is shown in Table 2. From this table one can see that there is a close match between the size of a choice point and the size of SPUR register window. Each window keep the argument registers local and overlaps state registers with the preceding and following windows (choice points). This is precisely the type of behavior required, access to the previous choice point is also required in the PLM. Choice point buffering with register windows reduces the instruction data fetch collisions on the try, try_me_else, retry, and retry_me_else instructions. Rather than interfering with data fetches, the contents of choice point registers can be obtained from internal processor registers rather than a stack frame in memory. Backtracking is accomplished with register move operations and the shifting of register windows. Choice point buffering appears to be the only natural use for SPUR's register windows.

The equivalent of the PLM B register in the SPUR implementation no longer contains a pointer into memory but now points to a register window. This number is incremented when a choice point is pushed and decremented when a

SPUR Register Allocation for PLM		
Type	Register	Use
Globals	0	hardwired 0
	1–8	PLM AX1–AX8
	9	Pointer to constant table
Inputs		Previous Choice Point
	10	PLM E register
	11	PLM TR register
	12	PLM H register
	13	PLM B register
	14	PLM BP register
	15	PLM CP register
Locals		Linkage and Temporaries
	16	PLM S mode flag
	17	PLM Cut mode flag
	18–25	PLM AX1–AX8 and CP when window becomes choice point
Outputs		Current Choice Point
	26	PLM E register
	27	PLM TR register
	28	PLM H register
	29	PLM B register
	30	PLM BP register
	31	PLM CP register

Table 2. Register allocation for PLM registers in SPUR register windows. Globals are used for the argument registers of the current choice point and for pointers to constant tables. When a choice point needs to pushed onto the stack, the argument register are moved into the local registers and the register window shifted. Otherwise the locals are used for temporaries and the S register, which is not shared across choice points. The overlap of the input and output registers makes the values of the registers of the previous choice point available to the current one.

choice point is popped. The PLM's S register is a local temporary register since its value is not needed across choice points. The ten SPUR local registers are used as temporaries by the macro expansion code. The eight argument registers are kept in global registers. This leaves only a single register (R9) for indexing into a constant table. This originally caused us to suggest that more global SPUR registers would be useful, however, a level of indirection solves this problem at minimal cost.

Since the PLM design includes only a single choice point buffer, we anticipate better performance than the PLM for programs that push and pop choice points often but not too deeply (more than 8). Operations that push choice points account for 10% of the execution time of PLM and, as expected, operations that pop choice points account for only 3% of its operation time. This asymmetry is due to popping multiple choice points during a single cut instruction. Efficient programs perform less backtracking and would be expected to pop more than one choice point in a single operation; a 3:1 push to pop ratio is typical. For programs that do push choice points deeply, our design will suffer the same performance degradation as the PLM. However, this type of behavior is unlikely to be a part of most programs.

An alternative to the register layout that we chose is to eliminate the argument registers (AX) and store the arguments in the procedure activation record as done in other programming languages. However, this requires more load and store instructions that would add to the instruction-data fetch bandwidth bottleneck. The advantage of this method is that it frees the SPUR global registers for other uses. Neither approach requires modification to the PLM compiler and their true merits will best be determined by simulation.

4.1.2. Register Windows and the Recursive Unify Operations

Unfortunately, the large size of SPUR's register windows reduces their effectiveness for recursive unification. Recursive unification must save only three registers per invocation. Since 16 new register become available on a window shift, this leaves 13 registers unused. Unrolling the unify code five times would permit the use of 15 of the 16 registers. This seems promising, however, the increased code size could be detrimental to instruction buffer and cache performance. In addition to unrolling the unify code, there are at least two other possible approaches to implementing recursive unify: implementing the recursion stack in memory and using the register windows directly by replacing the overflow/underflow trap handler to save and restore only three registers. Yet another possibility is to have the SPUR hardware provide two window sizes, one as currently implemented and another small one for procedures that may be highly recursive but only require a small number of arguments. We chose the simplest path and implemented our own recursion stack in memory.

4.1.3. Macro-expanding PLM Instructions

Although there are papers that describe the PLM instruction set, the PLM simulator [Dobry84a] is the only place that accurately describes the semantics of all PLM instructions. Our approach to macro-expanding PLM to SPUR was to use the model of PLM functionality provided by the PLM level 1 simulator. The simulator is written in C and contains a separate procedure for each PLM instruction. Some of the procedures share common subroutines. Essentially, we hand-compiled the procedures in the PLM simulator into SPUR assembly language; the functions that simulated an instruction were made into macros and the common functions were put into a subroutine library loaded with every SPUR program. Since we tried not to deviate from the PLM simulator, we employed minimal optimizations. The only optimizations were to use the SPUR tags to simulate the PLM tags and register windows for choice points.

In addition to the special stack needed for recursive unify, we also had to implement our own procedure call mechanism for calling the common functions. When calling a common function, the arguments are put into temporary registers, the return address is put into a temporary register, and then the SPUR *jump* instruction is used to execute the procedure. No registers have to be saved.

We implemented two of the commonly used large macros as function calls in order to reduce the code size at the expense of four extra instructions, two to call and two to return. Since these macros were complex, the extra overhead is small compared to the number of instructions executed in the function. Table 3 shows the improvements in code size because of this optimization on the Prolog

Static Code Size of Macro-expanded Benchmarks

Benchmark	No functions (C1)	Functions for two largest macros (C2)	C1 / C2	Functions for 3 more large macros (C3)	C2 / C3	C1 / C3
con1	594	414	1.44	385	1.08	1.54
con6	610	430	1.42	401	1.07	1.52
divide10	4922	3988	1.23	1688	2.36	2.92
hanoi	585	385	1.52	385	1.00	1.52
log10	4606	4040	1.14	1676	2.41	2.75
mutest	2945	1703	1.73	1152	1.48	2.56
nrev1	2153	761	2.83	669	1.14	3.22
ops8	4692	3804	1.23	1632	2.33	2.88
palin25	3982	2556	1.56	1632	1.57	2.44
pri2	2061	1933	1.07	1704	1.13	1.21
qs4	3608	1230	2.93	1038	1.19	3.48
queens	3826	3636	1.05	2998	1.21	1.28
query	4136	3942	1.05	3768	1.05	1.10
times10	4398	3988	1.25	1728	2.31	2.55
Geom. mean			1.45		1.44	2.06

Table 3. This table shows the reduction in static code size when large commonly used macros are implemented as subroutine calls to a library routine. The first column shows the code size if none of the macros are turned into functions. The second column shows the code size if *unify_constant* and *unify_value* are turned into functions as was done in our macro-expansion. The third column shows the ratio of column 1 to column 2. Column 4 shows the code size if *get_list*, *get_structure* and *unify_variable* were turned into functions. Columns 5 and 6 show the ratio of column 2 to column 4 and column 1 to column 4 respectively. The bottom row is the geometric mean over all of the benchmarks. A short description of the benchmarks is given in Table 4.

benchmarks. It also shows what would happen if we implemented some of the smaller commonly used macros as function calls. We see that the benchmarks with two macros implemented as function calls have a code size 44 percent greater than that attainable if five macros were function calls. Overall, benchmarks that do not use function calls for the macros are over twice as large as benchmarks that use all five.

4.1.4. Software to Apply Macro-expansions to Benchmark Programs

Once we had generated the macro-expansions of PLM instructions to SPUR instructions we developed a software system to automatically apply the macro-expansions to the compiled PLM benchmark programs. We developed **preproc** and **postproc**, and used **/lib/cpp** (a standard Unix utility) as well as **sas** and **sld** (written by the SPUR Lisp group) to generate the macro-expansions. The sequence that transforms a PLM program into SPUR assembly code is:

preproc

It takes the PLM instructions and puts them in a format that can be used by **/lib/cpp** to perform the macro-expansion. It also extracts all constants from the PLM code and puts them into a constant table.

/lib/cpp

This program is the standard Unix C preprocessor. Its purpose is to macro-expand the properly formatted PLM instructions into SPUR assembly language.

postproc

By the time this program sees the PLM program it has already been macro-expanded to SPUR code. However, most of the macros have labels within them. Since a macro can be used in many different places in the code and labels must be global, there would be many label conflicts if the code was passed to the SPUR assembler. It is the purpose of **postproc** to change all of the labels to global labels.

sas and **sld**

These are the SPUR assembler and loader. They take SPUR assembly code and turn it into object code that runs on the SPUR simulator.

A script to run this sequence of commands to produce a file that runs on the SPUR simulator is shown in Appendix 1.

5. Comparison of Prolog Performance on PLM and SPUR

Our goal of running the benchmark programs on the SPUR and PLM simulators and comparing their performance was accomplished in three steps. First we wrote macro-expansion and software development tools and applied these to the benchmarks listed in Table 4. We automatically generated SPUR instructions from their PLM instructions for all but one (ckt2) of these benchmarks programs. Next we ran the macro-expanded programs on the SPUR simulator to determine if the expansions were correct and to generate memory references traces. To verify the correctness of the macro-expansions, we modified the SPUR simulator to print out the data structures generated by the Prolog program. Lastly, we modified the PLM simulator to generate memory traces.

5.1. Modifications to the PLM and SPUR Simulators

5.1.1. The PLM Simulator

Two PLM simulators were graciously provided by Tep Dobry of the Aquarius-PLM project. The level 1 simulator simulates the macro-architecture of PLM whereas the level 2 simulator simulates the micro-architecture. We chose to

Prolog Benchmarks		
Name	Description	Lines of PLM
con1	Deterministic concatenation of two lists.	29
con6	Non-deterministic concatenation of two lists.	33
ckt2	Design of a 2 by 1 MUX using NAND gates.	601
divide10	Symbolic differentiation using division.	222
hanoi	Solution to Tower of Hanoi problem with 8 disks.	55
log10	Symbolic differentiation using logarithms.	216
mutest	Proof of theorem in Hofstadter's mu math system.	142
nrev1	Naive reversal of a list of 30 numbers.	73
ops8	Symbolic differentiation using a polynomial.	214
palin25	Program to generate a palindrome.	187
pri2	Program to find prime numbers less than 100.	141
qs4	A quicksort program of 50 numbers.	125
queens	Solution to the queens problem on a 4 by 4 chess board.	267
query	A data base query problem.	340
times10	Symbolic differentiation using multiplication.	222

Table 4. The 15 benchmarks used in the PLM performance study [Dobry85]. All have been implemented on SPUR except for ckt2.

instrument the level 1 simulator because it is much easier to understand and modify than the level 2 version. It is important to note that the level 1 simulator was designed to run the benchmarks and is not capable of executing all Prolog programs. Many system support and escape instructions are not implemented. This is the reason that larger benchmark programs were not run.

The simulator, as provided, kept frequency counts of instructions and statistics on the number of reads and writes, dereferences, unifications and bindings. We enhanced the simulator by adding code to output memory reference traces and to compute the number of cycles executed. To generate data for our cache studies, we modified the simulator to log memory reference traces. Very few changes were required to record data references because data references go through the two routines *stick* (data write) and *stuck* (data read). However, more detective work was needed to make sure that all references to the code space ("Cspace") were recorded since constants as well as instructions are stored in Cspace. In addition, we modified the simulator to record the real size of instructions so instruction references also record the size of the instruction being fetched.

The level 1 simulator, unlike the level 2 simulator, does not keep track of the number of cycles executed. We added a table containing the average number of cycles executed for each instruction. The values in the table were derived using the same calculation style as the Berkeley PLM group. Where decisions had to be made, we attempted to calculate the worst case path with the exception of general unify, decdr, and dereference operations since the PLM chose average times for these operations in their calculations. Hence, data structures with multiple dereferences take longer than the table suggests. We compute the total number of cycles executed by a program from the instruction frequency and the cycle tables. The total cycles executed is not a precise value but a lower bound of the real value.

A 'hook' routine was added to **barb** (the SPUR simulator) to handle escape calls. Escapes are functions that cannot be handled by the PLM and must be handled by the host. In our implementation arithmetic comparison escapes are handled in-line, but escapes for I/O and arithmetic are handled in **barb**. Many escapes are analogous to system calls so it is fair not to expect either the SPUR or PLM implementations to handle them in-line.

5.2. Results

We compared the static and dynamic code sizes, number of instructions executed, of the SPUR and PLM versions of the benchmarks in tables 5 and 6. The SPUR versions of the benchmarks are, as expected, uniformly larger than their PLM counterparts. Table 5 shows static sizes of the benchmarks in instructions and bytes. The instruction ratios range from 7.40 (hanoi) to 19.52 (log10). The low ratio for hanoi is because one-half of its PLM instructions map to sequences of 1 or 2 SPUR instructions, fully one-third of the PLM instructions executed map to sequences of just 1 SPUR instruction. The high ratio for log10 is because 40% of

Static Code Size							
	PLM		SPUR		S/P		
Benchmark	#Instr.	#Bytes	#Instr.	#Bytes	Instr. ratio	Bytes ratio	With func bytes
con1	28	87	414	1656	14.79	19.03	51.08
con6	32	106	430	1720	13.44	16.23	42.53
divide10	213	661	3988	15952	18.72	24.13	28.36
hanoi	52	183	385	1540	7.40	8.42	23.65
log10	207	625	4040	16160	19.52	25.86	30.32
mutest	141	468	1703	6812	12.08	14.56	20.51
nrev1	71	260	761	3044	10.72	11.71	22.43
ops8	205	633	3804	15216	18.56	24.04	28.44
palin25	178	565	2556	10224	14.36	18.10	23.03
pri2	132	383	1933	7732	14.64	20.19	27.47
qs4	121	456	1230	4920	10.17	10.79	16.90
queens	242	723	3636	14544	15.03	20.12	23.97
query	273	1138	3942	15768	14.44	13.86	16.31
times10	213	661	3988	15952	18.72	24.13	28.35
Geom. mean					14.00	17.06	26.11

Table 5. Static Code Size for the PLM and SPUR Benchmarks. The average number of bytes per PLM instruction is 3.30. There is an additional 697 instructions (1788 bytes) of functions loaded with each SPUR program.

the PLM instructions are either get_structure (approximately 43 SPUR instructions) or unify_variable (approximately 29 SPUR instructions). The mean SPUR/PLM ratio for instructions and bytes are 14.00 and 17.06 respectively. The byte ratio is larger because the average PLM instruction is 3.30 bytes. Note that these byte ratios do not include the code for a fixed-size subroutine library loaded with each SPUR benchmark. It is interesting compare the size of this subroutine library (1.8KB) with the size of the PLM microcode (about 17KB).

Comparison of the dynamic code size shows that SPUR executes on average about 16 instructions for each PLM instruction (see Table 6). The hanoi benchmark had the lowest ratio (11.81). The query benchmark has the highest ratio (20.77) for which we do not see a ready explanation. These ratios do not reflect the real amount of work done by the PLM since PLM instructions take from one to over 26 cycles to execute while SPUR instructions only take one cycle to execute. Comparing the number of SPUR instructions executed to the number of PLM cycles executed shows that on average, SPUR requires 2.31 cycles for each PLM cycle. The SPUR/PLM cycle ratio ranges from 1.96 for hanoi and pri2 to 4.09 for query. Excluding query, the highest ratio is 2.67 for con6.

Instructions and Cycles Executed									
Benchmark	PLM		SPUR		SPUR/PLM		SPUR NOPs		Barb Hooks
	Instr.	Cycles	Instr.	Cyles	Instr.	Cycles	#	%	#
conl	43	296	627	672	14.58	2.27	78	12.44	2
con6	133	1006	2474	2685	18.60	2.67	294	11.88	30
divide10	447	3512	6724	7374	15.04	2.10	785	11.67	2
hanoi	11996	79097	141644	154911	11.81	1.96	17095	12.07	765
log10	145	1182	2859	3139	19.72	2.57	330	11.54	2
mutest	12983	116967	249960	263243	19.25	2.25	36021	14.41	7
nrev1	4092	31825	62216	65846	15.20	2.07	7993	12.85	2
ops8	260	1935	3583	3918	13.78	2.02	438	12.22	4
palin25	3540	27092	64620	68838	18.25	2.54	9326	14.43	11
pri2	14567	109384	195985	213929	13.45	1.96	22589	11.53	579
qs4	6596	49308	96456	103648	14.62	2.10	11792	12.23	2
queens	5177	36350	76575	83135	14.79	2.29	9509	12.42	236
query	21973	116845	456396	478223	20.77	4.09	55877	12.24	1910
times10	375	2891	5410	5916	14.43	2.05	650	12.01	2
Geom. mean					15.81	2.31		12.39	

Table 6. Dynamic Code Size for PLM and SPUR. For SPUR, the number of cycles executed is calculated by adding the number of instructions executed to the number of data writes from Table 7, effectively double counting store instructions. This is necessary because the store instruction take two cycles to execute. Using the data from this table and from Table 7, the average number of cycles per SPUR instruction is 1.07.

The percentage of no-op instructions in the SPUR code averages 12.39% (Table 6). Most of the no-op slots in the branch, call and return instructions were not used, but many were used after jumps. The barb hook column in Table 6 measures the number of calls to **barb** to handle I/O and arithmetic operations. Table 7 shows the number of data reads and writes for the benchmarks. Generally, SPUR does 3% more reads and 18% more writes than PLM. The ratio of SPUR/PLM reads ranges from 0.72 (log10) to 1.43 (con1) and the ratio for writes ranges from 0.88 (qs4) to 2.37 (con1). The con1 benchmark has anomalous behavior because it performs almost twice as many reads and writes for SPUR than for PLM.

5.3. Analysis of Memory Traces

Up to now, we have compared performance for the two machines assuming that all memory references are completed in one cycle. This is the type of performance measurement used in [Dobry85]. A more realistic model of performance would consider the memory system used by the architecture. The memory reference traces enable us to do detailed simulations of cache performance and compare the effect of SPUR's increased code size on the miss ratio. The memory trace data was analyzed using the **dineroIII** cache simulator [Hill83, Hill85].

Number of Data References						
Benchmark	Reads			Writes		
	PLM	SPUR	S/P	PLM	SPUR	S/P
con1	21	30	1.43	19	45	2.37
con6	225	268	1.19	161	211	1.31
divide10	729	578	0.79	598	650	1.09
hanoi	14792	14798	1.00	17100	13267	0.78
log10	334	240	0.72	122	180	1.48
mutest	16131	16590	1.03	14502	13283	0.92
nrev1	1637	2728	1.67	1697	3630	2.14
ops8	349	335	0.96	314	335	1.07
palin25	3955	5120	1.29	3351	4218	1.26
pri2	19734	19383	0.98	23881	17944	0.75
qs4	8196	6890	0.84	8196	7192	0.88
queens	7475	7507	1.00	7076	6560	0.93
query	41813	48264	1.15	14522	21827	1.50
times10	603	488	0.81	499	506	1.01
Geom. mean			1.03			1.18

Table 7. Number of memory data references for PLM and SPUR. All memory references are assumed to complete within one cycle.

Instruction buffers were not simulated in these studies. We felt that since they perform very different functions for the two architectures it would be difficult to compare the miss ratio results. It is also not clear which architecture would benefit more if its instruction buffer were included. In the case of the PLM the instruction buffer does not reduce memory bandwidth since its function is primarily as a decoder. Four instructions is clearly not enough to capture loops that may exist in the PLM code. In the case of SPUR the instructions buffer is a simple instruction cache that helps to reduce instruction and data fetch contention for the mixed cache. At 128 words it is large enough to hold many loops and recursive procedures in the SPUR macro-expansion.

Simulations were done for two types of caches: a mixed instruction and data cache (as in SPUR) and a separate instruction and data cache (as in PLM). The caches were direct-mapped and varied in size from 2KB to 128KB and infinity. A block size of 32 bytes was used in all the simulations.

Tables 8 and 9 show the result of the simulations done with **dineroIII** for the PLM. Generally, separate I&D caches gave better miss ratios when the cache size was less than 8KB. The mixed and separate miss ratios are the same after 8KB except for nrev1. This is probably a reflection of the small size of the benchmark programs. Data and instruction addresses for the PLM were offset by 2048 to minimize collisions between cache blocks containing data and instructions. In

Separate I&D Cache Miss Ratios for PLM							
Benchmark	2KB	4KB	8KB	16KB	32KB	64KB	128KB
con1	7.23%	7.23%	7.23%	7.23%	7.23%	7.23%	7.23%
con6	11.56	11.56	11.56	2.12	2.12	2.12	2.12
divide10	5.07	3.33	3.33	3.16	3.16	3.16	3.16
hanoi	0.04	0.04	0.04	0.04	0.04	0.04	0.04
log10	3.83	3.83	3.83	3.66	3.66	3.66	3.66
mutest	0.70	0.70	0.70	0.36	0.36	0.36	0.11
nrev1	1.94	1.76	1.63	1.63	1.63	1.63	1.41
ops8	4.66	4.66	4.66	3.90	3.90	3.90	3.90
palin25	3.61	1.53	1.53	0.96	0.96	0.96	0.92
pri2	3.42	0.78	0.78	0.46	0.46	0.46	0.44
qs4	3.08	1.41	1.01	0.64	0.64	0.64	0.64
queens	2.01	2.01	2.01	1.95	1.95	1.95	0.29
query	5.42	3.15	3.15	0.08	0.08	0.08	0.08
times10	3.59	3.25	3.25	3.05	3.05	3.05	3.05

Table 8. Cache miss ratios for PLM using separate instruction and data caches of varying sizes. The sizes listed are the total size for instruction and data caches of equal size. Cache parameters: direct-mapped, separate I+D, 32-byte blocks.

Mixed I&D Cache Miss Ratios for PLM							
Benchmark	2KB	4KB	8KB	16KB	32KB	64KB	128KB
con1	7.23%	7.23%	7.23%	7.23%	7.23%	7.23%	7.23%
con6	18.69	11.56	2.12	2.12	2.12	2.12	2.12
divide10	5.75	3.33	3.16	3.16	3.16	3.16	3.16
hanoi	8.92	0.04	0.04	0.04	0.04	0.04	0.04
log10	8.15	3.83	3.66	3.66	3.66	3.66	3.66
mutest	6.90	0.70	0.36	0.36	0.36	0.11	0.11
nrev1	3.41	2.83	1.63	1.63	1.63	1.41	1.41
ops8	9.21	4.66	3.90	3.90	3.90	3.90	3.90
palin25	2.30	1.53	0.96	0.96	0.96	0.92	0.92
pri2	6.61	1.27	0.56	0.46	0.46	0.44	0.44
qs4	1.98	1.26	0.64	0.64	0.64	0.64	0.64
queens	9.60	2.01	1.95	1.95	1.95	0.29	0.29
query	13.70	3.15	0.08	0.08	0.08	0.08	0.08
times10	5.89	3.25	3.05	3.05	3.05	3.05	3.05

Table 9. Cache Miss Ratios for PLM using a mixed instruction and data cache. Various cache sizes were simulated with dineroIII. Cache parameters: direct-mapped, mixed I+D, 32-byte blocks.

fact, even the numbers for a cache size of 4KB (the minimum to avoid the extra collisions) were close enough that a comparison of PLM and SPUR with a mixed cache is justified. We do not want to include the differences in memory system in the comparison of the two architectures.

The SPUR trace data was only simulated with a mixed I&D cache since the SPUR hardware will be equipped with a mixed I&D cache. Unlike PLM, SPUR has a single address space per process with separate segments for code and data, hence no offset was used between SPUR instruction and data addresses.

Small benchmarks such as con1, log10, and ops8 had comparatively high miss ratios even with an infinite cache size (see Table 10). The miss ratio of an infinite cache is defined as the number of unique blocks referenced divided by the total number of references. Table 11 compares the data in Tables 9 and 10 of the 8 largest benchmarks and it shows that SPUR requires a cache 4 to 8 times larger than PLM to get approximately equivalent miss ratios. It is interesting to note that this ratio is very close to the actual ratio of cache sizes in the two implementations. The cache sizes were chosen such that miss ratios were under 1% and the SPUR and PLM ratios also were approximately the same. The nrev1 benchmark is interesting because SPUR had a better miss ratio than PLM because under PLM, nrev1 references are 55% code and 45% data while under SPUR references are 91% code and 9% data. The benchmark is small and the code miss ratio is about 0.2% for PLM and SPUR for a cache size of 16KB and greater. The data miss ratio is 3.4% for PLM and 7.8% for SPUR. Since the PLM version is half data, the data miss ratio dominates the overall ratio. Under SPUR, the data miss

Mixed I&D Cache Miss Ratios for SPUR

Benchmark	2KB	4KB	8KB	16KB	32KB	64KB	128KB
con1	14.65%	12.23%	10.67%	10.53%	9.53%	9.53%	9.53%
con6	9.04	5.52	4.43	3.79	2.91	2.91	2.91
divide10	15.76	10.45	6.87	3.87	3.56	3.56	3.56
hanoi	6.46	2.47	0.35	0.12	0.09	0.09	0.09
log10	14.42	10.24	6.37	6.31	5.58	5.58	5.58
mutest	11.1	4.38	1.18	0.43	0.12	0.12	0.11
nrev1	7.38	2.38	1.08	0.96	0.92	0.92	0.76
ops8	16.69	13.96	11.92	7.97	7.03	6.72	6.72
palin25	13.11	7.08	4.09	2.94	1.31	0.88	0.69
pri2	13.40	7.39	2.12	0.99	0.70	0.68	0.35+
qs4	12.70	5.11	2.55	1.77	1.11	1.11	0.45
queens	15.34	9.99	6.62	2.70	1.17	0.54	0.53+
query	12.22	8.57	2.88	0.87	0.47	0.11	0.11
times10	14.30	11.79	7.99	4.17	3.76	3.76	3.76

Table 10. Cache miss ratios for SPUR using a mixed instruction and data caches of various sizes. Except for pri2 and queens, the 128KB cache was equal to an infinite cache. The infinite cache miss ratios for pri2 and queens are 0.34% and 0.48% respectively. Cache parameters: direct-mapped, mixed I+D, 32-byte blocks.

Comparison of Mixed I&D Cache Miss Ratios					
	PLM		SPUR		S/P
Benchmark	Size (KB)	Miss Ratio (%)	Size (KB)	Miss Ratio (%)	
hanoi	4	0.04	32	0.09	8
mutest	4	0.70	16	0.43	4
nrev1	64	1.41	16	0.96	.25
palin25	8	0.96	64	0.88	8
pri2	8	0.56	64	0.68	8
qs4	8	0.64	64	1.11	8
queens	64	0.29	64	0.54	1
query	8	0.08	64	0.11	8

Table 11. Comparison of SPUR and PLM cache miss ratios using a mixed I&D cache. The first PLM miss ratio under 1% was chosen for each benchmark. The corresponding SPUR cache size for an equivalent miss ratio was then used to determine the SPUR/PLM ratio.

ratio contributes only 9% to the overall miss ratio. Examining the trace files shows that the high data miss ratio is probably due to the large number of environment allocations on the stack.

We also simulated fully-associative caches for PLM and SPUR trace data to see the effects of conflicts (see tables 12 and 13). In every benchmark except for queens, the miss ratio for direct-mapped and fully-associative caches were equal after an 8KB cache size for PLM and 32KB for SPUR. Another interesting observation is that the miss ratio for fully-associative caches of sizes 2KB for PLM and 8–16KB for SPUR equaled the miss ratio of infinitely-large caches.

Fully-Associative Mixed I&D Cache Miss Ratios for PLM						
Benchmark	2KB	4KB	8KB	16KB	32KB	64KB
con1	7.23%	7.23%	7.23%	7.23%	7.23%	7.23%
con6	2.12	2.12	2.12	2.12	2.12	2.12
divide10	3.16	3.16	3.16	3.16	3.16	3.16
hanoi	0.04	0.04	0.04	0.04	0.04	0.04
log10	3.66	3.66	3.66	3.66	3.66	3.66
mutest	0.11	0.11	0.11	0.11	0.11	0.11
nrev1	1.52	1.41	1.41	1.41	1.41	1.41
ops8	3.90	3.90	3.90	3.90	3.90	3.90
palin25	1.06	0.92	0.92	0.92	0.92	0.92
pri2	0.47	0.45	0.44	0.44	0.44	0.44
qs4	0.72	0.64	0.64	0.64	0.64	0.64
queens	0.29	0.29	0.29	0.29	0.29	0.29
query	0.08	0.08	0.08	0.08	0.08	0.08
times10	3.05	3.05	3.05	3.05	3.05	3.05

Table 12. Cache miss ratios for PLM using a fully-associative mixed instruction and data cache. Various cache sizes were simulated with dineroIII. Cache block size: 32 bytes.

Fully-Associative Mixed I&D Cache Miss Ratios for SPUR						
Benchmark	2KB	4KB	8KB	16KB	32KB	64KB
con1	9.53%	9.53%	9.53%	9.53%	9.53%	9.53%
con6	11.31	2.91	2.91	2.91	2.91	2.91
divide10	12.62	3.99	3.57	3.56	3.56	3.56
hanoi	4.44	0.09	0.09	0.09	0.09	0.09
log10	5.91	5.61	5.58	5.58	5.58	5.58
mutest	11.55	2.08	0.11	0.11	0.11	0.11
nrev1	1.19	0.81	0.79	0.76	0.76	0.76
ops8	14.34	12.15	6.72	6.72	6.72	6.72
palin25	12.35	3.19	0.84	0.69	0.69	0.69
pri2	13.18	0.84	0.35	0.34	0.34	0.34
qs4	9.94	1.41	0.46	0.45	0.45	0.45
queens	12.21	9.31	3.81	0.48	0.48	0.48
query	11.52	2.66	1.89	0.10	0.10	0.10
times10	12.47	4.12	3.76	3.76	3.76	3.76

Table 13. Cache miss ratios for SPUR using a fully-associative mixed instruction and data caches of various sizes. Cache block size: 32 bytes.

6. Future Work

6.1. Optimizations

Three directions for future work are possible, optimizing our macro-expansions, building a compiler and augmenting SPUR's hardware to support Prolog more directly.

6.1.1. Macro-expansion Optimizations and Compiling

As stated in Section 4.1.3, we made no attempt to optimize the macro-expansions beyond using SPUR's tags and register windows. Many simple optimizations are possible that will greatly improve performance. For example, the effect of using one no-op slot in the variable dereferencing macro decreased the number of cycles executed by the query benchmark by 2.5%. Applying peephole optimizations and macro-expansion optimizations should yield a large improvement.

A much more ambitious project is to compile Prolog into native SPUR code rather than using macro-expansion. In some systems, compilation can improve performance by factors of two or three. It would be interesting to see what a compiler for SPUR could gain in performance and code density over our macro-expansion technique.

6.1.2. Improvements to SPUR

One shortcoming of SPUR is that tags can only be compared with immediates (*tag_cmp_br_delayed*). The PLM, on the other hand, can test a subset of the tag bits for a pattern. In the macro-expansion, tags must be read into a register, anded with a mask, and then a compare-branch instruction used on the result. The sequence of instructions required to perform this operation are *rd_tag*, *and*, and *cmp_br_delayed* (denoted R-A-C). With the SPUR simulator we measured a 15% average improvement in performance if SPUR had a single instruction to replace the R-A-C sequence (Table 14). We calculated this by counting the number of *and* instructions executed since *and* is only used for the masking operation. This is an upper bound because some of the *and* instructions can be done in no-op slots instead of in the R-A-C sequence. The first three columns of Table 14 show that 87% of *and* instructions appear in the R-A-C sequence. Assuming that the static distribution of *and* instructions approximates the dynamic distribution, these results indicate that an improvement of more than 10% would be attainable with the additional instruction. This instruction can be added to the SPUR architecture without affecting the cycle time and at only a modest impact in extra circuitry. A possible format for the instruction is shown in Figure 2.

6.2. A Prolog Coprocessor for SPUR

The other type of performance improvement we considered is a specialized hardware accelerator for SPUR. SPUR supports a tightly-coupled coprocessor

Potential Performance Improvements

Benchmark	R-A-C Sequences	Static Ands	Ratio	Ands Executed	Cycles Saved (%)
con1	57	63	0.90	54	17.23
con6	56	62	0.90	182	14.71
divide10	202	242	0.80	476	14.16
hanoi	48	53	0.91	6378	9.00
log10	206	247	0.83	208	14.55
mutest	112	129	0.87	21320	17.06
nrev1	68	76	0.89	6440	20.70
ops8	194	232	0.84	254	14.18
palin25	154	181	0.85	5426	16.79
pri2	131	150	0.87	11426	11.66
qs4	92	104	0.88	8293	17.20
queens	213	249	0.86	5551	14.50
query	299	319	0.94	43757	19.18
times10	202	242	0.83	386	14.27
Geom. mean			0.87		15.08

Table 14. This table illustrates the possible performance improvement if an instruction that applies a mask to a tag and branches on the result were added to SPUR. The new instruction would replace a sequence of three current instructions. We computed the improvement by dividing twice the number of *and* instructions executed by the total number of cycles currently executed for each benchmark (Table 6), assuming all *and* instructions appear in the R-A-C sequence.

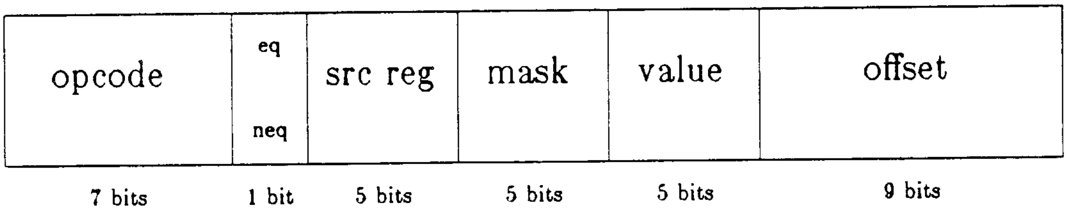


Figure 2. This figure shows a possible format for a combined read-and-compare tag instruction for SPUR. The operands are a flag indicating test for equal/not equal, the source register for the tag, a constant to be masked with the tag, a constant value to be tested with the tag and an offset to add to the program counter if the test is successful.

model where the coprocessor and CPU are on the same side of the system bus and make use of the same caches [Hansen86]. Instruction fetching is under strict control of the CPU. However, execution of some instructions is deferred to the coprocessor. The CPU initiates loads and stores into and out of the coprocessor but the coprocessor latches the data or supplies it directly to the cache. This is ideal for floating point units that otherwise would have too high an overhead if bus accesses were required for each operation. In contrast to SPUR, the PLM employs the loosely-coupled model. The coprocessor is not on the same board as the CPU and the set up of the computation and reading of the results must be done over the system bus.

We considered designing a tightly-coupled coprocessor for SPUR implementing a subset of the PLM instructions. We identified the basic operations performed by the PLM and then determined which could be performed efficiently with standard SPUR instructions and which would benefit from the use of special coprocessor hardware. We suggest extensions to SPUR's coprocessor interface and a coprocessor architecture that allows SPUR to execute Prolog at the same speed as the PLM.

6.2.1. Issues in Coprocessor Design

The current SPUR coprocessor interface is quite limited. All memory operations to or from coprocessor registers are initiated by the CPU; the coprocessor can only manipulate the contents of its registers. The only coprocessor actively planned by the SPUR group is a floating point unit, which has heavily influenced the interface. For our purposes, the current coprocessor interface is unworkable since Prolog does not perform sophisticated bit manipulations of registers as do floating point coprocessors, but instead reads, compares, and updates the contents of memory locations. We will describe the extensions we feel are necessary to add an arbitrary (possibly microcoded) coprocessor to SPUR.

Our Prolog coprocessor design greatly reduces the size of PLM programs on SPUR because it executes much higher level instructions than does SPUR. This reduces SPUR's static and dynamic code sizes, increasing cache and instruction buffer performance†.

For this coprocessing model to be compatible with the RISC nature of SPUR, two simple rules must be enforced. First, all instructions must be restartable. If a coprocessor instruction causes a page fault, it must allow the CPU to perform the necessary operations to bring that page into memory. The CPU then reissues the instruction that caused the trap just as it does for all CPU instructions. Therefore, coprocessor instructions cannot change the internal state of the coprocessor until all memory references have been completed. The second rule is that a system interrupt must not be serviced until all coprocessor instructions in progress have been allowed to complete (unless prevented from doing so by a page fault). This will ensure consistent changes to the internal state of the coprocessor.

---

† In general, if a collection of instructions can be found that contribute greatly to the execution time of certain applications, they can be built into a (microcoded) coprocessor. In fact, a standard, tailorable, micro-engine coprocessor could be designed and tailored to different applications.

Interrupts are currently handled this way for SPUR's floating-point coprocessor.

These two rules at first seem incompatible with the indeterminate and recursive PLM instructions. The most important case of this is the PLM's implementation of the unify instruction. It performs a pattern matching of two arbitrarily large data structures and cannot be suspended and restarted. Hence the PLM must wait for page traps to be resolved by the NCR coprocessor that acts as its host. In contrast, SPUR must service its own page faults. In the next section we show that the unify operation can be unwound so that only one pattern matching step is performed per coprocessor instruction, making the instruction restartable. The unwound instructions have a much lower bound on their execution time and this allows the second rule to be enforced without delaying interrupt servicing appreciably.

6.2.2. Extensions to SPUR's Coprocessor Interface

Besides the addition of the cache address bus and cache operation lines to the coprocessor, the only other additions are a page fault line and the coprocessor memory access line. When a coprocessor instruction needs to generate a data load or store it asserts the memory access line one cycle in advance to prevent SPUR's instruction fetch unit from attempting a cache operation. In effect, this is a cache bus arbitration line that always resolves in favor of the coprocessor. The needed functionality is already present in SPUR's prefetch unit in the circuitry used to resolve instruction and data access collisions. Support for this facility consists of a pin dedicated to the memory access line and an internal OR gate to make this line appear as a CPU instruction generating a data load or store. Traps and exceptions are handled the same way as before. The important rules to follow are (again): instructions must be restartable and interrupts wait for instructions to complete.

6.2.3. Prolog Coprocessor Architecture

The coprocessor design was strongly influenced by the PLM. We looked at the detailed operations each instruction performed and determined which ones would be easily and efficiently handled by SPUR CPU instructions, which ones require complex tag and pointer manipulations and have to be unwound so that they can be restartable, and which ones could be implemented directly.

We placed all Prolog execution state (i.e. the registers of the PLM) in the coprocessor. These are required by most instructions and placing them in the coprocessor enables optimizations for choice point buffering. Currently we do not make use of the register windows although these could easily be implemented directly in the coprocessor. Register file reading and writing is identical to the SPUR CPU including the pipeline forwarding logic. Provisions are made for extra global registers and a special NIL register.

All system support functions are performed as they would be for any other program running on SPUR. Special system or library calls for supporting Prolog can be incorporated in the SPUR operating system. These would be a subset of the escape codes used by the PLM, most would already be available. Code and data segments are managed in the same way as the SPUR-only implementation described above.

6.2.3.1. Coprocessor Instruction Set

Coprocessor instructions can be divided into six groups. A complete list of these and an outline of their microcode is provided in Appendices 3 and 4. The first group includes three types of data transfer instructions. These move data between the coprocessor and memory, between the coprocessor and the CPU, and between registers in the coprocessor. The second group is the state modifying and saving instructions. These are used to push and pop choice points and environments from their respective stacks as well as setting the mode bits. Compare and branch instructions make up the third group. These include a read, mask, and compare tag instruction such as the one suggested for SPUR previously in this section and a condition code test instruction. The next group is the unwound unify instructions that are discussed in more detail below. The fifth group is heap and trail manipulation instructions. These are used to allocate variables on the top of the heap and undo the bindings on the trail stack at goal failure. The last group consists of the special hash instruction used by the PLM to implement a multi-way branch based on the value of an argument. This is currently implemented as a linear search of a table as it is in the PLM.

6.2.3.2. Unwinding the Unify Instruction

To unwind the unify instruction we need to add two special registers to the PLM architecture holding the addresses of the next two items to unify. When a unify instruction cannot complete after having unified the original arguments, rather than continuing as in the PLM, it places the intermediate arguments in these registers (U1 and U2). Unify instructions that have the possibility of becoming recursive (unification with constants and variables cannot possibly be recursive) are followed by a compare and branch instruction that tests whether there is more to unify or not. If there is, the branch back to the unify instruction is taken, if not execution proceeds sequentially. When the unify instruction begins execution it first checks the "more to unify" mode bit and, if it is set, continues execution using the contents of the U1 and U2 registers. If pointers need to be pushed onto the push down list (PDL) this is done at the beginning of the unify instructions; if they need to be popped, this is done at the end. If there is nothing on the PDL then the "more to unify" bit is reset. Therefore the unify arguments are always available at the beginning of instruction execution.

There is a performance penalty for a two instruction loop in SPUR, however. SPUR does not support delayed slot cancellation when a branch is taken. In the case of the unify loop, adding cancellation would eliminate many of the no-ops required in the code. Without it, we are forced to use a three instruction (dynamic) loop where one of the instructions is a no-op. However, when one considers the time spent by the coprocessor in the unify operation in each loop, this extra cost is not large.

6.2.3.3. Interfacing to the SPUR CPU Pipeline

The SPUR pipeline consists of four stages: instruction fetch, register read and address generation, memory access, and register write. The registers allow a single write and two reads in a single cycle. Results of compare instructions must be generated by the end of the second stage so that only one instruction is in the pipeline after the one that causes the branch. This is what is meant by delayed branch. The CPU pipeline can be suspended by the coprocessor by asserting the busy line. This facility allows the coprocessor to arbitrarily extend any cycle for its own purposes. Our coprocessor will not be used in parallel mode, to avoid cache access conflicts. Also, unlike floating point instructions, our instructions are indeterminate in duration and it would be difficult for a compiler to schedule parallel execution.

The Prolog coprocessor must interface to the pipeline of the SPUR CPU. To give the coprocessor the extra time that certain instructions may require, we expand the pipeline between the second and third pipeline stages. The coprocessor is pipelined in the same way as the CPU. Figure 3 explains the coprocessor pipeline graphically.

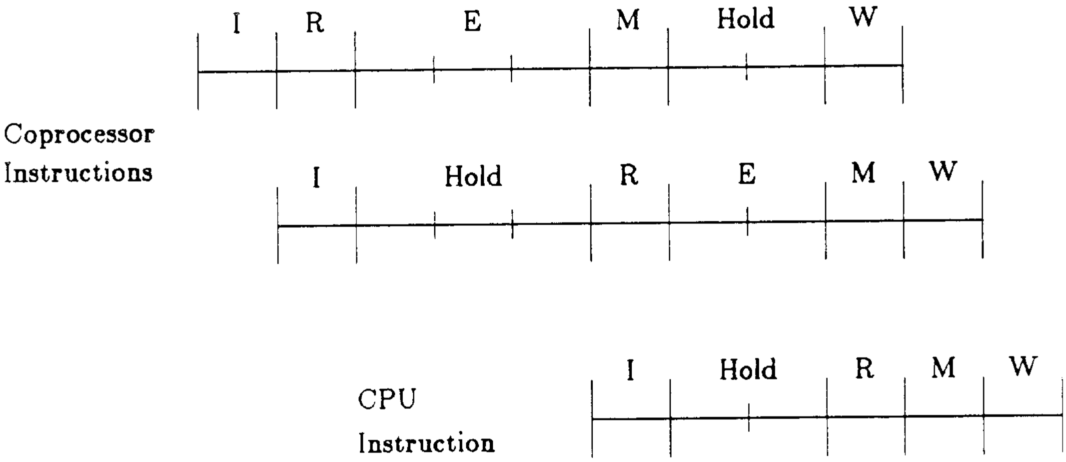


Figure 3. Coprocessor pipeline. The coprocessor finds the extra cycles it may need to execute an instruction between the second and third stages of the SPUR pipeline. By using this extended area and the coprocessor memory access control line (an extension to the current coprocessor interface of SPUR) we ensure that no cache access conflicts occur. The overlap of macro-instructions in the SPUR pipeline is one of the major sources of improved execution time for SPUR over the PLM. (Cycle names: I - instruction fetch, R - register read, E - intsruction execution, M - memory access, W - register write.)

An important consideration mentioned previously is that coprocessor state can only be changed after all memory accesses have been successfully completed. To meet this requirement, a set of memory address and data registers are added to the coprocessor to act as staging areas for all memory transactions. Once the memory accesses are all completed, the contents of the staging registers can be moved to internal registers. This is the primary mechanism for insuring instruction restartability and why it is critical that instructions not be interrupted while updating internal state.

6.2.4. Expected Performance of SPUR with a Prolog Coprocessor

We expect the performance of SPUR with a Prolog coprocessor to be at least as high as the PLM but with smaller and less complex microcode. In Table 15 we see that execution time is approximately 10% better than the PLM. However, the code size required for the coprocessor, although much less than for SPUR alone, is still a factor of 3.4 larger than the PLM (Table 16).

In summary, many instructions have been eliminated and instruction decode is greatly simplified. All instructions are 4 bytes with standard argument formats. The SPUR pipeline does not impart high overhead, as most instructions require a register read or write. All extra micro-cycles required by the coprocessor are available by suspending the CPU pipeline and thus ensuring the absence of cache access conflicts.

Our claim of 10% better performance is not as startling as it may seem. The SPUR macro-instruction cycle time is the same as the PLM micro-instruction cycle time. The PLM micro-engine, although pipelined does not exploit macro-instruction overlap. SPUR's macro-instructions provide many of the primitive operations implemented as more than one cycle in the PLM. SPUR with a coprocessor is faster for the simple instructions and is only slightly slower than the PLM for the recursive unify operations. We feel that this claim is justified since the micro-engine and microcode that must be implemented in the coprocessor are a subset of the PLM's and most operations are performed directly in SPUR instructions. A conservative estimate of the coprocessor's microcode size is 3.3KB, compared to PLM's 17KB. This assumes a 96-bit wide microinstruction and 274 microwords.

Comparison of Execution Times

		PLM		SPUR-CoP	
Instruction	Freq	Cycles	Weight	Cycles	Weight
put_value	10.70	3	32.1	1.5	16.1
unify_variable	8.80	7	61.6	8	70.4
get_list	7.27	10	72.7	7	50.9
unify_cdr	6.88	5	34.4	4.5	31.0
unify_value	4.96	14.5	71.9	14.5	71.9
escapes	4.90				
switch_on_term	4.87	11	53.6	5	24.4
unify_nil	4.86	4	19.5	4.5	21.9
get_structure	4.11	12	49.3	13.5	55.5
execute	4.01	1	4.0	2	8.0
allocate	3.47	11	38.2	5	17.4
get_variable	3.44	2.5	8.6	1.5	5.2
unify_constant	3.33	8	26.6	10	33.3
deallocate	2.87	6	17.2	4	11.5
put_constant	2.71	2	5.4	2	5.4
proceed	2.65	1	2.7	4	10.6
try_me_else	2.45	20	49.0	20	49.0
call	2.00	1	2.0	4	8.0
cut	1.85	10	18.5	5.5	10.2
get_constant	1.83	11	20.1	10	18.3
put_variable	1.79	3.5	6.3	2.5	4.5
get_value	1.44	13	18.7	20	28.8
trust_me_else	1.32	5	6.6	3	4.0
get_nil	1.29	11	14.2	7	9.0
put_unsafe_value	1.24	10	12.4	6	7.4
retry_me_else	0.88	2	1.8	6	5.3
switch_on_structure	0.87	13	11.3	13	11.3
put_list	0.77	3	2.3	2	1.5
try	0.71	20	14.2	22	15.6
fail	0.56	23	12.9	21	11.8
trust	0.35	5	1.8	5	1.8
unify_void	0.33	6	2.0	16	5.3
switch_on_constant	0.20	10	2.0	13	2.6
retry	0.06	2	0.1	4	0.2
put_structure	0.05	4	0.2	6	0.3
put_nil	0.01	2	0.0	1	0.0
Total Weights			694.1		628.6
Relative Performance			1.00		0.91

Table 15. This table compares performance of Prolog on the the PLM and SPUR using a specialized Prolog coprocessor. The figures for instructions frequency and PLM cycle time are summarized from [Dobry85] using more up-to-date values. The SPUR implementation is simple macro-expansions and does not consider what could be achievable by shuffling in-

struction to take better advantage of delayed load and store slots or other optimizations.

Comparison of Code Size					
		PLM		SPUR-CoP	
Instruction	Freq.	Bytes	Weight	Bytes	Weight
put_value	10.70	3	32.1	4	42.8
unify_variable	8.80	2	17.6	8	70.4
get_list	7.27	2	14.5	8	58.2
unify_cdr	6.88	2	9.9	12	55.0
unify_value	4.96	2	9.9	12	59.5
escapes	4.90				
switch_on_term	4.87	4	19.5	16	77.9
unify_nil	4.86	1	4.9	8	38.9
get_structure	4.11	6	24.7	24	98.6
execute	4.01	5	20.1	8	32.1
allocate	3.47	1	3.5	4	13.9
get_variable	3.44	3	10.3	4	13.8
unify_constant	3.33	5	16.7	20	66.7
deallocate	2.87	1	2.9	4	11.5
put_constant	2.71	6	16.3	8	21.7
proceed	2.65	1	2.7	16	42.4
try_me_else	2.45	5	12.3	16	39.2
call	2.00	6	12.0	16	32.0
cut	1.85	3	5.6	10	18.5
get_constant	1.83	6	11.0	20	36.6
put_variable	1.79	3	5.4	6	10.7
get_value	1.44	3	4.3	14	20.2
trust_me_else	1.32	1	1.3	4	5.3
get_nil	1.29	2	2.6	8	10.3
put_unsafe_value	1.24	3	3.7	4	5.0
retry_me_else	0.88	5	4.4	24	21.1
switch_on_structure	0.87	6	5.2	36	31.3
put_list	0.77	2	1.5	8	6.2
try	0.71	5	3.6	24	17.0
fail	0.56	1	0.6	24	13.4
trust	0.35	5	1.8	12	4.2
unify_void	0.33	2	0.7	16	5.3
switch_on_constant	0.20	6	1.2	36	7.2
retry	0.06	5	0.3	24	1.4
put_structure	0.05	6	0.3	20	1.0
put_nil	0.01	2	0.0	4	0.0
Total Weights			287.3		989.3
Relative Code Size			1.00		3.44

Table 16. This table compares the code size of Prolog programs on the Berkeley PLM and on SPUR with a Prolog co-processor. The SPUR code size is calculated from simple macro-expansion of the instructions. Constants form part of some PLM instructions. For SPUR, we counted the fetching of these constants as part of the code size cost.

7. Conclusions

In summary, we expect that a macro-expansion of the PLM instruction set will run Prolog programs on SPUR at 43% of the speed of the PLM when memory accesses are assumed to be one cycle. SPUR with a Prolog coprocessor can be made to be 10% faster than the PLM due to simplification of the micro-engine and the advantages of pipelining across instructions.

When cache and instruction fetch behavior is taken into account we suspect that the pure SPUR implementation will suffer greatly. Due to the expanded code size of the SPUR implementation, the SPUR cache will have to be 4 to 8 times larger than one used for the PLM in order to obtain the same miss ratio. Also SPUR will require a somewhat larger processor-to-memory bandwidth because of the expanded code size and tag storage. The coprocessor implementation will remain very close to PLM performance due to the much more similar code density. There is much more work to be done on the coprocessor design before a definitive statement can be made.

However, we feel that the system support advantages of SPUR make it competitive with the PLM for large applications, with or without the coprocessor. This is especially true when one considers large real applications that involve a large amount of interactions with the operating system for I/O or are floating-point arithmetic intensive. The utility of a SPUR Prolog implementation is especially important in mixed paradigm programming systems where only a part of the computations would be in the logic programming paradigm. SPUR is reasonably high-performance for Prolog and very competitive in running other languages. The PLM's special hardware and loosely-coupled coprocessor model makes running mixed-language applications less efficient.

A Prolog coprocessor for SPUR can be added when applications demand an improved logic programming performance. The coprocessor interface changes required to support microcoded accelerators are minimal. The architecture of the coprocessor is a hybrid of the PLM and SPUR architectures. We feel that a tightly-coupled VLSI Prolog coprocessor for SPUR is a viable alternative to a specialized loosely-coupled Prolog accelerator such as the PLM.

The bottom line of this study is that SPUR can support a language other than Lisp or C with excellent performance. In fact, SPUR would place third among the Prolog implementations listed in Table 17, and with a coprocessor, it would be the fastest.

Performance Estimates for Logic Programming Systems

Deterministic Concatenate Benchmark (conl)

Machine	System	Performance (in LIPS)	Reference
Berkeley SPUR	coprocessor	465,000	estimate
Berkeley PLM	(TTL)/Compiled	425,000	simulation (no wait states)
Tick & Warren	VLSI	415,000	Estimate, Tick & Warren
Aquarius I	(TTL)/Compiler	305,000	simulation (NCR bus)
Berkeley SPUR	Macro-expansion	184,000	simulation
Symbolics 3600	Microcoded	110,000	estimate, Tick & Warren
DEC 2060	Warren Compiled	43,000	Warren
Japan 5th Gen PSI	Microcoded	30,000	estimate, PSI paper
IBM 3033	Waterloo	27,000	Warren
DEC VAX-11/780	Macrocoded	15,000	estimate, Tick & Warren
Sun-2	Quintus Compiler	14,000	Warren
LMI/Lambda	Uppsala	8,000	Warren
DEC VAX-11/780	POPLOG	2,000	Warren
DEC VAX-11/780	M-PROLOG	2,000	Warren
DEC VAX-11/780	C-PROLOG	1,500	Warren
Symbolics 3600	Interpreter	1,500	Warren
DEC PDP-11/70	Interpreter	1,000	Warren
Z-80	MicroProlog	120	Warren
Apple-II	Interpreter	8	Warren

Performance on General Benchmark Programs

Machine	System	Performance (in LIPS)	Reference
Berkeley SPUR	Coprocessor	225,000	estimate
Berkeley PLM	(TTL)/Compiled	205,000	simulation
Berkeley SPUR	Macro-expansion	89,000	simulation
LMI/Lambda	Micro/Compiled	12,400	LMI Corp.
Japan 5th Gen PPC	Microcoded	10,000	estimate, NTIS (#N83-31379)
LM-2	Microcoded	9,500	Prolog Digest v2.20
LMI/Lambda	Macro/Compiled	6,200	LMI Corp.
Symbolics 3600	Microcoded	5,000	Prolog Digest v2.20
LMI/Lambda	Micro/Interpreter	3,400	LMI Corp.
LMI/Lambda	Macro/Interpreter	1,700	LMI Corp.
Apple-II	Pascal Interpreter	10	Colmerauer

Performance on the Warren Benchmarks
(times10 divide10 log10 ops8 palin25 query list30 list50)

Machine	System	Performance (in LIPS)	Reference
Berkeley SPUR	Coprocessor	163,000	estimate
Berkeley PLM	(TTL)/Compiled	149,216	simulation
Berkeley SPUR	Macro-expansion	60,000	simulation (excl. list30,50)
LMI/Lambda	Micro/Compiled	12,400	LMI Corp.
DEC 2060	Warren Compiled	12,175	Warren thesis

Table 17. Performance of various Prolog implementations. This table was adapted from [Dobry85].

8. Acknowledgements

We relied on the help of many people while working on this project. David Patterson provided the guidance and the topic for us to attack. We had to become familiar with two complex computers and a new programming language and we relied heavily upon our fellow graduate students. The Berkeley PLM and SPUR groups were extremely helpful and encouraging. Wayne Citrin, Tep Dobry, and Barry Fagin answered our unending stream of questions about the PLM and helped us understand logic programming. Tep and Wayne graciously provided us with the PLM simulator and compiler. Mark Hill, David Wood, Paul Hansen, and George Taylor provided invaluable discussions on the SPUR architecture, coprocessor interface, and the tradeoffs involved. Mark's **dineroIII** cache simulator made it possible for us to do our studies. Also, Ben Zorn and George Taylor were very helpful with modifying **barb** to suit our needs. The interesting combination of computer research issues with which this project was concerned made it an invaluable learning experience for us. Alvin Despain, Mark Hill, David Patterson, Herve Touati and Peter Van Roy provided useful comments on a draft of this report.

This report was published by the Berkeley SPUR project. Principal funding for the SPUR project is provided by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269. Additional support for SPUR was provided by the State of California MICRO program, by a Digital Equipment Corporation CAD/CAM grant, by the National Science Foundation under grant DCR-8202591, by equipment donations from Texas Instruments, Inc., and by computer resources provided under DARPA contract N00039-84-C-0089.

9. References

[Clocksin81] Clocksin, W.F. and Mellish, C.S. *Programming in Prolog*, Springer-Verlag, New York, 1981.

[Dobry84a] Dobry, T.P. "PLM Simulator Reference Manual, working paper 3.3", University of California, Berkeley, September 1984.

[Dobry84b] Dobry, T.P. "A Prolog Machine Architecture Working Paper 4.0", University of California, Berkeley, December 1984.

[Dobry84c] Dobry, T.P. "Design Decisions Influencing the Microarchitecture for a Prolog Machine." IEEE MICRO **17**, Proceedings, October 1984.

[Dobry85] Dobry, T.P., Despain, A.M., Patt, Y.N. "Performance Studies of a Prolog Machine Architecture." Proceedings of the 12th Annual International Symposium on Computer Architecture, Boston, MA, June 1985.

[Fagin85] Fagin, B. and Dobry, T.P. "The Berkeley PLM Instruction Set: An Instruction Set for Prolog." UCB/CSD 86/257, University of California, Berkeley, September 1985.

[Halbert80] Halbert, D. and Kessler, P. "Windows of Overlapping Registers", Final Report for U.C. Berkeley CS-292r, June 1980.

[Hansen86] Hansen, P. and Kong, S. "SPUR Coprocessor Interface Description", UCB/CSD 87/308, University of California, Berkeley, October 1986.

[Hill83] Hill, M.D. "Evaluation of On-Chip Cache Memories." Master's Report, U.C. Berkeley Computer Science Division, December 1983.

[Hill85] Hill, M.D. "DineroIII Documentation." Unpublished Unix-style Man Page, October 1985.

[Hill86] Hill, M.D., et al. "Design Decisions in SPUR: a VLSI Multiprocessor." IEEE Computer **19**, November 1986.

[Katevenis83] Katevenis, M.G.H. "Reduced Instruction Set Computer Architectures for VLSI." Ph.D. Dissertation, Computer Sciences (EECS), U.C. Berkeley, 1983. Also as Technical Report UCB/CSD 83/141.

[Moto-oka83] Moto-oka, T. "Overview of the Fifth Generation Project." Proceedings of the 10th Annual International Symposium on Computer Architecture, Stockholm, Sweden, June 1983.

[Tick83] Tick, E. and Warren, D.H.D. "Towards a Pipelined Prolog Processor." Artificial Intelligence Center, SRI International, August 1983.

[Ungar84] Ungar, D., Blau, R., Foley, P., Samples, D., Patterson, D. "Architecture of SOAR: Smalltalk on a RISC." Proceedings of the 11th Annual International Symposium on Computer Architecture, Ann Arbor, MI, June 1984.

[Warren77] Warren, D.H.D. "Applied Logic: its use and implementation as programming tool." Ph.D. Thesis, U. of Edinburgh, Scotland, 1977. Available as Technical Note 290, Artificial Intelligence Center, SRI International.

[Warren83] Warren, D.H.D. "An Abstract Prolog Instruction Set." Artificial Intelligence Center, SRI International, August 1983.

Appendices

There are four appendices. Appendix 1 contains listings of the software tools developed to macro-expand PLM instructions to SPUR instructions and Appendix 2 lists the actual code used used to implement each PLM instruction. Appendix 3 describes the code for macro-expansion onto SPUR with a Prolog coprocessor. The numbers in Tables 15 and 16 were derived from these macro-expansions. The final appendix is an outline of the microcode that will be required for SPUR's Prolog coprocessor.

Appendix 1: Listings of Software Tools

This appendix contains listings of the two software tools that we wrote to allow us to automatically macro-expand Prolog programs from PLM code to SPUR. The first program is **preproc.c**. Its purpose is to put the PLM instructions into a format that can be fed to the C preprocessor and to set up the constant table. The second program is **postproc.c**. Its purpose is to turn local labels generated by the macros into global labels. It produces output suitable as input to the SPUR assembler. The final listing shows a csh command file to convert PLM code into a binary file that can be run on the SPUR simulator.

The macro-expansion process uses /lib/cpp. The input to cpp is:

headers.h
defs.h
instructions.h (macro definitions)
PLM code preprocessed with **preproc**
funcs.a
trailers.h
constants table (created by **preproc**)

(The files headers.h, defs.h, instructions.h, trailers.h and funcs.a are listed in Appendix 2.) The output from cpp is post-processed with **postproc** before it is assembled and linked with **sas** and **sld**.

```
/*
 * preproc.c --
 *
 *        A filter that takes PLM assembly code and converts it
 *        into a form that /lib/cpp can handle.
 */

typedef int Boolean;
#define TRUE 1
#define FALSE 0
#include "list.h"
#undef NULL
#include <stdio.h>
#include <ctype.h>
#include <strings.h>

#define CONST_TABLE_START 44
int       constTableOffset = CONST_TABLE_START;
#define MAX_CONST_OFFSET  0x1000

#define NIL_STR (char *) NULL

#define LINE_SIZE 80

typedef enum {
     LABEL,
     STRING,
     NUMBER,
} ConstType;

typedef   struct {
     List_Links     links;
     char           name[LINE_SIZE+1];
     ConstType      type;
     int            offset;
} ConstRec;

FILE      *constFilePtr = (FILE *) NULL;
FILE      *oldFilePtr = stdin;
FILE      *curFilePtr = stdin;

typedef   struct {
     List_Links     links;
     char           name[LINE_SIZE+1];
     char           command[LINE_SIZE+1];
     int            offset;
} EscapeRec;

typedef   struct {
     List_Links     links;
     char           name[LINE_SIZE+1];
     int            (*func)();
} InstrRec;

#define HASH_SIZE   137
List_Links     instrHashTable[HASH_SIZE];
List_Links     constHashTable[HASH_SIZE];
List_Links     escapeHashTable[HASH_SIZE];

#define   HASH_FUNC(name, len) ((len+(name[0] * name[len - 1])) % HASH_SIZE)

char *SkipWhiteSpace();
char *FindWhiteSpace();

InstrRec *InstrHashFind();
void InstrHashInsert();
```

```
ConstRec *ConstHashFind();
ConstRec *ConstHashInsert();

EscapeRec *EscapeHashFind();
void EscapeHashInsert();

/*
 * Main --
 *
 *         Opens the input file and calls Preprocess to interpret
 *         the file.
 *
 */
```

main

```
main(argc, argv)
     int              argc;
     char             **argv;
{

     InitHash();

     if (argc > 1) {
          constFilePtr = fopen(argv[1], "w");
          if (constFilePtr == (FILE *) NULL) {
               fprintf(stderr, "Can't open %s\n", argv[1]);
               exit(1);
          }
     } else {
          fprintf(stderr, "Missing argument for constants file.\n");
          exit(1);
     }
     Preprocess(stdin);
}

/*
 * Preprocess
 *
 *         Reads each line in the file and parses it into
 *         the label, instructio and argument components.
 *         If the instruction is valid, a instruction-dependent
 *         routine is called to process it.
 */
```

Preprocess

```
Preprocess(filePtr)
     FILE *filePtr;
{
     char             line[LINE_SIZE+1];
     char             *colon;
     char             *end;
     char             *linePtr;
     char             *args;
     int              status;
     int              retStat;
     int              len;
     InstrRec         *instrRecPtr;

     status = GetLine(filePtr, line);
     while (status) {
          if (strlen(line) == 0) {
               status = GetLine(filePtr, line);
               continue;
          }

          linePtr = line;
```

```
        /*
         * Look for an optional label followed by : before white space
         *
         *      label:_ _ _ _instr_ _arg1,arg2
         *               ^-start of white space
         *             |-colon
         *    This scheme fails if a : is an argument with white space before .
         */

        colon = index(linePtr, ':');
        end = FindWhiteSpace(linePtr);
        if (colon != NIL_STR && (colon < end || end == NIL_STR)) {
            /*
             * Found label  -- print it on a separate line.
             */
            printf("%.*s\n", colon - line + 1, line);

            linePtr = SkipWhiteSpace(colon+1);
            if (linePtr == NIL_STR) {
                /* nothing else on line */
                status = GetLine(filePtr, line);
                continue;
            }
        }

        /*
         * Look for end of instruction name
         */
        end = FindWhiteSpace(linePtr);
        if (end == NIL_STR)  {
            /* no white space */
            len = strlen(linePtr);
        } else {
            len = end - linePtr;
        }

        instrRecPtr = InstrHashFind(linePtr, len);

        if (instrRecPtr == (InstrRec *) NULL) {
            fprintf(stderr, "Unknown instruction: '%s'\n", linePtr);
        } else {
            if (end == NIL_STR) {
                args = "";
            } else {
                args = SkipWhiteSpace(end+1);
            }
            retStat = (*instrRecPtr->func)(linePtr, len, args);
            if (!retStat) {
                fprintf(stderr, "Bad input: %s\n", line);
            }
        }

        status = GetLine(filePtr, line);
    }
}
/*
 * Instruction Processing Routines
 *
 *      Converts each instruction into a CPP macro.
 *
 *      NoArgsFunc      - instructions w/o arguments.
 *      DefaultFunc     - instructions with arguments but no special
 *                        processing is needed.
 *      CallFunc        - changes the name of the call instructions
 *      VariableFunc    - if the instruction argument is a register
```

```
 *                              then convert name to use the _reg form
 *                              else convert the name to use the _var form.
 *      ConstantFunc         -  the instruction uses a constant, which must
 *                              be recorded in the constants table.
 *      JumpFunc             -  records the label in a jump-type instruction.
 *      SwitchOnConstantFunc -  handles switch_on_constant.
 *      SwitchOnStructFunc   -  handles switch_on_structure.
 *      EscapeFunc           -  processes escapes.
 *
 */
```

NoArgsFunc

```
NoArgsFunc(line, len, args)
	char		*line;
	int		len;
	char		*args;
{
	printf("	%.*s()\n", len, line);
	return(TRUE);
}
```

DefaultFunc

```
DefaultFunc(line, len, args)
	char		*line;
	int		len;
	char		*args;
{
	/*
	 * Add () to args.
	 */
	printf("	%.*s(%s)\n", len, line, args);
	return(TRUE);
}
```

CallFunc

```
CallFunc(line, len, args)
	char		*line;
	int		len;
	char		*args;
{
	char *slash = index(args, '/');
	if (slash != NIL_STR) {
		*slash = '_';
	}

	/*
	 * Convert call to call_proc, fail to call_fail, leave procedure
	 * and execute alone.
	 */
	if (line[0] == 'c') {
		printf("	call_proc(%s)\n", args);
	} else if (line[0] == 'f') {
		printf("	call_fail(%s)\n", args);
	} else if (line[0] == 'p') {
		printf("	procedure(%s)\n", args);
	} else if (line[0] == 'e') {
		printf("	execute(%s)\n", args);
	} else {
		fprintf(stderr,"Unknown instruction for CallFunc: %s\n", line);
	}
	return(TRUE);
}
```

VariableFunc

```
VariableFunc(line, len, args)
	char		*line;
	int		len;
	char		*args;
```

```
{
    if (args == NIL_STR) {
        return(FALSE);
    }

    if (index(args, 'Y') == NIL_STR) {
        printf("                %.*s_reg(%s)\n", len, line, args);
    } else {
        printf("                %.*s_var(%s)\n", len, line, args);
    }
    return(TRUE);
}
```

ConstantFunc

```
ConstantFunc(line, len, args)
    char        *line;
    int         len;
    char        *args;
{
    char *ptr;
    int  offset;
    ConstType type;

    if (args == NIL_STR) {
        return(FALSE);
    }

    /*
     * See if instruction is unify_constant
     */
    if (line[0] == 'u') {
        if (args[0] == '&') {
            offset = AddConst(NUMBER, args+1, strlen(args)-1);
            printf("        unify_constant_number(%d)\n", offset);
        } else {
            offset = AddConst(STRING, args, strlen(args));
            printf("        unify_constant_string(%d)\n", offset);
        }
    } else {
        /*
         * See if instruction is {get,put}_constant
         */
        ptr = 1+ index(line,'_');
        if (*ptr == 'c') {
            ptr = index(args, ',');
            if (ptr == NIL_STR) {
                return(FALSE);      /* missing comma */
            }
            /*
             * See if comma is itself an argument.
             * If not the first argument is missing
             */
            if (*ptr == args[0]) {
                ptr = index(args, ',');
                if (ptr == NIL_STR) {
                    return(FALSE);
                }
                offset = AddConst(STRING, "," , 1);
                type = STRING;
            } else {
                if (args[0] == '&') {
                    offset = AddConst(NUMBER, args+1, ptr - args -1);
                    type = NUMBER;
                } else {
                    offset = AddConst(STRING, args, ptr - args);
                    type = STRING;
                }
```

```
                }
                if (type == STRING) {
                    printf("            %.*s_string(%d%s)\n", len, line, offset, ptr);
                } else {
                    printf("            %.*s_number(%d%s)\n", len, line, offset, ptr);
                }
            } else {
                /*
                 * Instruction is {get,put}_structure
                 */
                ptr = index(args, ',');
                if (ptr == NIL_STR) {
                    return(FALSE);
                }
                offset = AddConst(STRING, args, ptr - args);
                printf("        %.*s(%d%s)\n", len, line, offset, ptr);
            }
        }
        return(TRUE);
}
```

JumpFunc

```
JumpFunc(line, len, args)
        char            *line;
        int             len;
        char            *args;
{
        int     offset;
        if (args == NIL_STR) {
            return(FALSE);
        }
        offset = AddConst(LABEL, args, strlen(args));
        printf("        %.*s(%d)\n", len, line, offset);
        return(TRUE);
}
```

SwitchOnConstantFunc

```
SwitchOnConstantFunc(line, len, args)
        char            *line;
        int             len;
        char            *args;
{
        char            temp[LINE_SIZE+1];
        int             mask;
        int             maskOffset;
        int             i;
        int             tableLen;
        int             status;

#define MAX_ENTRIES 128
        struct {
            int         const;
            char        label[LINE_SIZE+1];
            ConstType           type;
        } entry[MAX_ENTRIES];

        if (sscanf(args,"%d,", &mask) != 1) {
            fprintf(stderr,"Switch: missing mask: %s\n", args);
            return(FALSE);
        }
        if (mask > MAX_ENTRIES) {
            fprintf(stderr,"Switch: table too big: %s\n", args);
            return(FALSE);
        }
        sprintf(temp, "%d", mask + 1);
        maskOffset = AddConst(NUMBER, temp, strlen(temp));
```

```
    /*
     * Table label is on next line following instruction.
     * Read it and forget it.
     */
    status = GetLine(curFilePtr, temp);

    /*
     * Mask is (table size -1) *2. An entry is on 2 lines.
     */

    tableLen = (mask + 1) /2;
    for (i = 0; i < tableLen; i++) {
        status = GetLine(curFilePtr, temp);
        if (temp[0] == '&') {
            entry[i].const = atoi(&(temp[1]));
            entry[i].type = NUMBER;
        } else {
            entry[i].const = AddConst(STRING, temp, strlen(temp));
            entry[i].type = STRING;
        }
        status = GetLine(curFilePtr, entry[i].label);
    }

    printf("        switch_on_constant(%d, %d)\n", maskOffset, constTableOffset);
    fprintf(constFilePtr,
            " #   switch on constant(%d, %d)\n", maskOffset, constTableOffset);
    for (i = 0; i < tableLen; i++) {
        fprintf(constFilePtr, ".long           %d\n", entry[i].const);
        if (entry[i].type == NUMBER) {
            fprintf(constFilePtr, ".long               const_num_type\n");
        } else {
            fprintf(constFilePtr, ".long               const_type\n");
        }
        fprintf(constFilePtr, ".long           %s\n", entry[i].label);
    }
    constTableOffset += 12 * tableLen;
    fprintf(constFilePtr, "\n");

    if (constTableOffset > MAX_CONST_OFFSET) {
        fprintf(stderr, "Warning: constant table overflow\n");
        fprintf(constFilePtr, " # Warning: constant table overflow\n");
    }

    return(TRUE);
}
```

SwitchOnStructFunc

```
SwitchOnStructFunc(line, len, args)
    char        *line;
    int         len;
    char        *args;
{
    char        temp[LINE_SIZE+1];
    int         mask;
    int         maskOffset;
    int         i;
    int         tableLen;
    int         status;

#define MAX_ENTRIES 128
    struct {
        int         const;
        char        label[LINE_SIZE+1];
    } entry[MAX_ENTRIES];

    if (sscanf(args,"%d,", &mask) != 1) {
        fprintf(stderr,"Switch: missing mask: %s\n", args);
```

```
            return(FALSE);
        }
        if (mask > MAX_ENTRIES) {
            fprintf(stderr,"Switch: table too big: %s\n", args);
            return(FALSE);
        }
        sprintf(temp, "%d", mask + 1);
        maskOffset = AddConst(NUMBER, temp, strlen(temp));

        /*
         * Table label is on next line following instruction.
         * Read it and forget it.
         */
        status = GetLine(curFilePtr, temp);

        /*
         * Mask is (table size -1) *2. An entry is on 2 lines.
         */

        tableLen = (mask + 1) /2;
        for (i = 0; i < tableLen; i++) {
            status = GetLine(curFilePtr, temp);
            entry[i].const = AddConst(STRING, temp, strlen(temp));
            status = GetLine(curFilePtr, entry[i].label);
        }

        printf("        switch_on_structure(%d, %d)\n", maskOffset, constTableOffset);
        fprintf(constFilePtr,
                " #   switch on structure(%d, %d)\n", maskOffset, constTableOffset);
        for (i = 0; i < tableLen; i++) {
            fprintf(constFilePtr, ".long               %d\n", entry[i].const);
            fprintf(constFilePtr, ".long               %s\n", entry[i].label);
        }
        constTableOffset += 8 * tableLen;
        fprintf(constFilePtr, "\n");

        if (constTableOffset > MAX_CONST_OFFSET) {
            fprintf(stderr, "Warning: constant table overflow\n");
            fprintf(constFilePtr, " # Warning: constant table overflow\n");
        }

        return(TRUE);
}
```

EscapeFunc

```
EscapeFunc(line, len, args)
        char            *line;
        int             len;
        char            *args;
{
        EscapeRec       *ptr;
        char            newArg[LINE_SIZE+1];
        char            *comma;

        comma = index(args, ',');
        if (comma == NIL_STR) {
            strcpy(newArg, args);
        } else {
            strncpy(newArg, args, comma-args);
            newArg[comma-args] = '\0';
        }

        ptr = EscapeHashFind(newArg);
        if (ptr == (EscapeRec *) NULL) {
            printf(" #          ESCAPE(%s,T1)\n", args);
            fprintf(stderr,"Unknown escape: %s\n", args);
        } else {
```

```
            printf("                %s\n", ptr->command);
        }
        return(TRUE);
}

/*
 * AddConst --
 *
 *          Adds a constant to the constant table if it was not
 *          already in it.
 */
```

AddConst

```
AddConst(type, name, len)
        ConstType type;
        char *name;
        int  len;
{
        ConstRec        *ptr;

        while (name[len - 1] == ' ' || name[len - 1] == '\t') {
            len--;
        }
        ptr = ConstHashFind(type, name, len);
        if (ptr == (ConstRec *) NULL) {
            ptr = ConstHashInsert(type, name, len);
        }
        return(ptr->offset);
}
```

InitHash

```
InitHash()
{
        int i;
        for (i = 0; i < HASH_SIZE; i++) {
            List_Init(&(instrHashTable[i]));
            List_Init(&(constHashTable[i]));
            List_Init(&(escapeHashTable[i]));
        }
        InstrHashInsert("end",                  NoArgsFunc);
        InstrHashInsert("allocate",             NoArgsFunc);
        InstrHashInsert("cut",                  NoArgsFunc);
        InstrHashInsert("deallocate",           NoArgsFunc);
        InstrHashInsert("proceed",              NoArgsFunc);
        InstrHashInsert("quit",                 NoArgsFunc);
        InstrHashInsert("unify_nil",            NoArgsFunc);

        InstrHashInsert("get_list",             DefaultFunc);
        InstrHashInsert("get_nil",              DefaultFunc);
        InstrHashInsert("mark",                 DefaultFunc);
        InstrHashInsert("pause",                DefaultFunc);
        InstrHashInsert("put_list",             DefaultFunc);
        InstrHashInsert("put_nil",              DefaultFunc);
        InstrHashInsert("switch_on_term",       DefaultFunc);
        InstrHashInsert("trust_me_else",        DefaultFunc);
        InstrHashInsert("unify_void",           DefaultFunc);
        InstrHashInsert("put_unsafe_value", DefaultFunc);

        InstrHashInsert("switch_on_constant",SwitchOnConstantFunc);
        InstrHashInsert("switch_on_structure",SwitchOnStructFunc);
        InstrHashInsert("procedure",            CallFunc);
        InstrHashInsert("call",                 CallFunc);
        InstrHashInsert("fail",                 CallFunc);
        InstrHashInsert("execute",              CallFunc);
        InstrHashInsert("escape",               EscapeFunc);

        InstrHashInsert("get_variable",         VariableFunc);
```

```
        InstrHashInsert("get_value",                    VariableFunc);
        InstrHashInsert("put_variable",                 VariableFunc);
        InstrHashInsert("put_value",                    VariableFunc);
        InstrHashInsert("unify_cdr",                    VariableFunc);
        InstrHashInsert("unify_value",                  VariableFunc);
        InstrHashInsert("unify_unsafe_value",     VariableFunc);
        InstrHashInsert("unify_variable",               VariableFunc);

        InstrHashInsert("get_constant",                 ConstantFunc);
        InstrHashInsert("get_structure",                ConstantFunc);
        InstrHashInsert("put_constant",                 ConstantFunc);
        InstrHashInsert("put_structure",                ConstantFunc);
        InstrHashInsert("unify_constant",               ConstantFunc);

        InstrHashInsert("cutd",                         JumpFunc);
        InstrHashInsert("retry_me_else",                JumpFunc);
        InstrHashInsert("retry",                        JumpFunc);
        InstrHashInsert("try_me_else",                  JumpFunc);
        InstrHashInsert("try",                          JumpFunc);
        InstrHashInsert("trust",                        JumpFunc);

        EscapeHashInsert("<",                           "less_than()");
        EscapeHashInsert("< /2",                        "less_than()");
        EscapeHashInsert("<=",                          "less_than_or_equal()");
        EscapeHashInsert("=",                           "equal()");
        EscapeHashInsert("=:= /2",                      "equal()");
        EscapeHashInsert("=< /2",                       "less_than_or_equal()");
        EscapeHashInsert(">",                           "greater_than()");
        EscapeHashInsert("> /2",                        "greater_than()");
        EscapeHashInsert(">=",                          "greater_than_or_equal()");
        EscapeHashInsert(">= /2",                       "greater_than_or_equal()");
        EscapeHashInsert("access",                      "access()");
        EscapeHashInsert("access /2",                   "access()");
        EscapeHashInsert("integer",                     "integer()");
        EscapeHashInsert("integer /1",                  "integer()");
        EscapeHashInsert("is",                          "is_escape()");
        EscapeHashInsert("is /4",                       "is_escape()");
        EscapeHashInsert("is /2",                       "is_2_escape()");
        EscapeHashInsert("nl",                          "escape(NL,T1)");
        EscapeHashInsert("nl /0",                       "escape(NL,T1)");
        EscapeHashInsert("set",                         "setter()");
        EscapeHashInsert("set /2",                      "setter()");
        EscapeHashInsert("var",                         "var_escape()");
        EscapeHashInsert("var /1",                      "var_escape()");
        EscapeHashInsert("write",                       "escape(WRITE,T1)");
        EscapeHashInsert("write /1",                    "escape(WRITE,T1)");

        /*
         * These escapes aren't handled yet.
         *
        EscapeHashInsert("=\= /2",              "----");
        EscapeHashInsert("asserta",             "----");
        EscapeHashInsert("assertz",             "----");
        EscapeHashInsert("call",                "Do by hand for now.");
        EscapeHashInsert("retracta",                    "----");
        EscapeHashInsert("retractp",                    "----");
        */
}

/*
 * Hash Routines::
 *
 *        Insert and Find routines for instructions, constants and escapes.
 */
```

InstrHashFind

```
InstrRec *
InstrHashFind(name, len)
	char		*name;
	int		len;
{
	int				hashId;
	register	InstrRec		*instrRecPtr;

	hashId = HASH_FUNC(name, len);

	LIST_FORALL(&instrHashTable[hashId], (List_Links *) instrRecPtr) {
		if (strncmp(instrRecPtr->name, name, len) == 0) {
			return(instrRecPtr);
			break;
		}
	}

	return((InstrRec *) NULL);
}
```

InstrHashInsert

```
void
InstrHashInsert(name, func)
	char *name;
	int (*func)();
{
	int		hashId;
	InstrRec	*instrRecPtr;

	hashId = HASH_FUNC(name, strlen(name));
	instrRecPtr = (InstrRec *) calloc(1, sizeof(InstrRec));
	if (instrRecPtr == (InstrRec *) NULL) {
		fprintf(stderr, "Calloc failed in InstrHashInsert\n");
		exit(1);
	}
	strcpy(instrRecPtr->name , name);
	instrRecPtr->func = func;
	List_Insert((List_Links *) instrRecPtr,
			LIST_ATFRONT(&instrHashTable[hashId]));
}
```

ConstHashFind

```
ConstRec *
ConstHashFind(type, name, len)
	ConstType	type;
	char		*name;
	int		len;
{
	register	ConstRec		*constRecPtr;
	int				hashId;

	hashId = HASH_FUNC(name, len);

	LIST_FORALL(&constHashTable[hashId], (List_Links *) constRecPtr) {
		if ((strncmp(constRecPtr->name, name, len) == 0) &&
			(constRecPtr->type == type)) {
			return(constRecPtr);
			break;
		}
	}
	return((ConstRec *) NULL);
}
```

ConstHashInsert

```
ConstRec *
ConstHashInsert(type, name, len)
	ConstType	type;
	char		*name;
```

```
    int             len;
{
    int             hashId;
    ConstRec        *constRecPtr;
    int             i;

    hashId = HASH_FUNC(name, len);
    constRecPtr = (ConstRec *) calloc(1, sizeof(ConstRec));
    if (constRecPtr == (ConstRec *) NULL) {
        fprintf(stderr, "Calloc failed in ConstHashInsert\n");
        exit(1);
    }
    strncpy(constRecPtr->name, name, len);
    constRecPtr->name[len] = '\0';
    constRecPtr->type = type;
    constRecPtr->offset = constTableOffset;
    List_Insert((List_Links *) constRecPtr,
                LIST_ATFRONT(&constHashTable[hashId]));

    if (type == LABEL) {
        fprintf(constFilePtr, ".long   %.*s    # label, offset = %d\n\n",
                        len, name, constTableOffset);
        constTableOffset += 4;
    } else if (type == STRING) {
        fprintf(constFilePtr, " # string '%.*s', offset = %d\n",
                        len, name, constTableOffset);
        for (i = 0; i < len; i++) {
            fprintf(constFilePtr, ".long   %3d    #  '%c'\n", name[i], name[i]);
        }
        fprintf(constFilePtr, ".long   0\n\n");
        constTableOffset += 4 * (len + 1);
    } else {
        fprintf(constFilePtr, ".long   %.*s    # number, offset = %d\n\n",
                        len, name, constTableOffset);
        constTableOffset += 4;
    }
    if (constTableOffset > MAX_CONST_OFFSET) {
        fprintf(stderr, "Warning: constant table overflow\n");
        fprintf(constFilePtr, " # Warning: constant table overflow\n");
    }
    return(constRecPtr);
}

EscapeRec *
EscapeHashFind(name)                                        EscapeHashFind
    char            *name;
{
    int                             hashId;
    register        EscapeRec       *ptr;

    hashId = HASH_FUNC(name, strlen(name));

    LIST_FORALL(&escapeHashTable[hashId], (List_Links *) ptr) {
        if (strcmp(ptr->name, name) == 0) {
            return(ptr);
            break;
        }
    }

    return((EscapeRec *) NULL);
}

void
EscapeHashInsert(name, command)                             EscapeHashInsert
    char *name;
    char *command;
```

```
{
	int		hashId:
	EscapeRec	*ptr;

	hashId = HASH_FUNC(name, strlen(name));
	ptr = (EscapeRec *) calloc(1, sizeof(EscapeRec));
	if (ptr == (EscapeRec *) NULL) {
		fprintf(stderr, "Calloc failed in EscapeHashInsert\n");
		exit(1);
	}
	strcpy(ptr->name , name);
	strcpy(ptr->command , command);
	List_Insert((List_Links *) ptr, LIST_ATFRONT(&escapeHashTable[hashId]));
}
```

```
/*
 *----------------------------------------------------------------------
 *
 * GetLine --
 *
 *      Gets a line from the file that does not start with a comment
 *      character (#). The line is null-terminated and the first
 *      newline '\n' is set to '\0'.
 *
 * Result:
 *        0       There was an error or EOF condition reading the line.
 *        1       The line was read succesfully.
 *
 *----------------------------------------------------------------------
 */

int
GetLine(filePtr, buffer)
     FILE *filePtr;
     char *buffer;
{
          char *status;
          char line[LINE_SIZE+1];
          int i;
          int j;

          /*
           * Skip the line if it begins with a comment character.
           */

          do {
               status = fgets(line, LINE_SIZE, filePtr);
               if (status == NIL_STR) {
                    return(0);          /* error or EOF */
               }
          } while (line[0] == '#') ;

          /*
           *  Trim leading white space while copying to the outpur arg.
           *  Convert the \n (if there is one) to a null character.
           */
          i = 0;
          while(i < LINE_SIZE && (line[i] == ' ' || line[i] == '\t')) {
               i++;
          }

          j = 0;
          buffer[j] = '\0';
          while((line[i] != '\n') && (line[i] != '\0')) {
               buffer[j] = line[i];
               i++;
               j++;
          }
          buffer[j] = '\0';

          if (buffer[0] == '!') {
               FILE *filePtr;
               filePtr = fopen(&(buffer[1]), "r");
               if (filePtr == (FILE *) NULL) {
                    fprintf(stderr, "Can't open %s for reading\n", &(buffer[1]));
               } else {
                    oldFilePtr = curFilePtr;
                    curFilePtr = filePtr;
                    Preprocess(filePtr);
                    curFilePtr = oldFilePtr;
```

GetLine

```
			}
			buffer[0] = '\0';
		}
		return(1);
}

/*
 * SkipWhiteSpace
 * FindWhiteSpace
 *
 *	Routines to skip over white space or
 *	skip over non-white space in a line.
 */
```

SkipWhiteSpace

```
char *
SkipWhiteSpace(string)
	char *string;
{
	register char *s = string;

	while (1) {
		if (*s == ' ' || *s == '\t') {
			s++;
		} else if (*s == '\0') {
			return(NIL_STR);
		}
		return (s);
	}
}
```

FindWhiteSpace

```
char *
FindWhiteSpace(string)
	char *string;
{
	register char *s = string;

	while (1) {
		if (*s == ' ' || *s == '\t') {
			return (s);
		} else if (*s == '\0') {
			return(NIL_STR);
		}
		s++;
	}
}
```

```
/*
 * Postproc.c --
 *
 *         A filter to convert the output of the CPP into a form
 *         that the SPUR assembler can handle. Local labels are
 *         resolved into unique names.
 *
 */

#include <stdio.h>
#include <ctype.h>

#define TRUE 1
#define FALSE 0
typedef int Boolean;
#include "list.h"

int        lineNum = 1;
int        labelNum = 1;

typedef    struct {
    List_Links      links;
    char            name[80];
    int             number;
    Boolean         forwardRef;
} LabelRec;

#define HASH_SIZE   101
List_Links      hashTable[HASH_SIZE];

#define    HASH_FUNC(name) \
           ((name[0] * name[strlen(name) - 1]) % HASH_SIZE)

/*
 * HashFind --
 *
 *         Find and retrieve the record for a given label in the hash table.
 *
 */
```

HashFind

```
LabelRec *
HashFind(name)
    char             *name;
{
    int                                  hashId;
    register        LabelRec             *labelRecPtr;
    Boolean                              found = FALSE;

    hashId = HASH_FUNC(name);

    LIST_FORALL(&hashTable[hashId], (List_Links *) labelRecPtr) {
        if (strcmp(labelRecPtr->name, name) == 0) {
            found = TRUE;
            break;
        }
    }

    if (!found) {
        return((LabelRec *) NULL);
    } else {
        return(labelRecPtr);
    }
}
```

HashInsert

```
/*
 * HashInsert --
 *
 *	Insert a label record in the hash table.
 *
 */
void
HashInsert(labelRecPtr)
	LabelRec	*labelRecPtr;
{
	int		hashId;

	hashId = HASH_FUNC(labelRecPtr->name);

	List_Insert((List_Links *) labelRecPtr,
			LIST_ATFRONT(&hashTable[hashId]));
}
```

HashDelete

```
/*
 * HashDelete --
 *
 *	Delete a label record from the hash table.
 *
 */
void
HashDelete(labelRecPtr)
	LabelRec	*labelRecPtr;
{
	List_Remove((List_Links *) labelRecPtr);
}
```

LabelProcess

```
/*
 * LabelProcess --
 *
 *	Resolves forward and backward label references.
 *	Labels are stored in a hash table.
 *
 */

LabelProcess()
{
	char		name[80];
	char		c;
	int		i = 0;
	LabelRec	*labelRecPtr;

	c = getchar();
	while (c != 'f' && c != 'b' && c != ':') {
		if (!isalpha(c) && !isdigit(c) && c != '_') {
			fprintf(stderr, "Malformed label at line %d\n", lineNum);
			exit(1);
		}
		name[i] = c;
		i++;
		c = getchar();
	}

	name[i] = '\0';

/*
	fprintf(stderr, "Label %s\n", name);
*/
	labelRecPtr = HashFind(name);
	if (labelRecPtr != NULL) {
		switch (c) {
			case ':':
```

```
                if (labelRecPtr->forwardRef) {
                    labelRecPtr->forwardRef = FALSE;
                } else {
                    labelRecPtr->number = labelNum;
                    labelNum++;
                }
                printf("Z%d:", labelRecPtr->number);
                break;
            case 'f':
                if (!labelRecPtr->forwardRef) {
                    labelRecPtr->forwardRef = TRUE;
                    labelRecPtr->number = labelNum;
                    labelNum++;
                }
                printf("Z%d", labelRecPtr->number);
                break;
            case 'b':
                printf("Z%d", labelRecPtr->number);
                break;
        }
    } else {
        if (c == 'b') {
            fprintf(stderr, "Undefined label at line %d\n", lineNum);
            exit(1);
        }
        labelRecPtr = (LabelRec *) malloc(sizeof(LabelRec));
        labelRecPtr->forwardRef = FALSE;
        strcpy(labelRecPtr->name,name);
        HashInsert(labelRecPtr);
        switch (c) {
            case ':':
                labelRecPtr->forwardRef = FALSE;
                labelRecPtr->number = labelNum;
                labelNum++;
                printf("Z%d:", labelRecPtr->number);
                break;
            case 'f':
                labelRecPtr->forwardRef = TRUE;
                labelRecPtr->number = labelNum;
                labelNum++;
                printf("Z%d", labelRecPtr->number);
                break;
        }
    }
}

/*
 * Main --
 *
 *      Scans through the file, and applies the following mappings:
 *      ';' -> '\n'
 *      '#' -> '# '
 *      '!' -> '#'
 *      '@label' -> Calls LabelProc with label
 *
 */

main()                                                                  main
{
    char        c;
    int     i;
    Boolean     justHadNL = TRUE;

    for (i = 0; i < HASH_SIZE; i++) {
        List_Init(&(hashTable[i]));
    }
```

```
    c = getchar();

    while (c != EOF) {
        if (c == '\n') {
            lineNum++;
            if (!justHadNL) {
                putchar(c);
            }
            justHadNL = TRUE;
            c = getchar();
            continue;
        }

        justHadNL = FALSE;

        if (c == ';') {
            putchar('\n');
        } else if (c == '#') {
            putchar(' ');
            putchar(c);
        } else if (c == '!') {
            putchar('#');
        } else if (c == '@') {
            LabelProcess();
        } else {
            putchar(c);
        }
        c = getchar();
    }
}
```

```
/* list.c --
 *
 * This file contains procedures for manipulating lists.
 * Structures may be inserted into or deleted from lists, and
 * they may be moved from one place in a list to another.
 *
 * The header file contains macros to help in determining the destination
 * locations for List_Insert and List_Move.  See list.h for details.
 *
 * Copyright (C) 1985 Regents of the University of California
 * All rights reserved.
 */

#ifndef lint
static char rcsid[] = "$Header: list.c,v 1.3 86/02/22 14:26:31 nelson Exp $ SPRITE (Berkeley)";
#endif not lint

#include "list.h"
```

List_Insert

```
/*
 *----------------------------------------------------------------------
 *
 * List_Insert --
 *
 *	Insert the list element pointed to by itemPtr into a List after
 *	destPtr.
 *
 * Results:
 *	No value is returned.
 *
 * Side effects:
 *	The list containing destPtr is modified to contain itemPtr.
 *
 *----------------------------------------------------------------------
 */
void
List_Insert(itemPtr, destPtr)
    register	List_Links *itemPtr;	/* structure to insert */
    register	List_Links *destPtr;	/* structure after which to insert it */
{
    itemPtr->nextPtr = destPtr->nextPtr;
    itemPtr->prevPtr = destPtr;
    destPtr->nextPtr->prevPtr = itemPtr;
    destPtr->nextPtr = itemPtr;
}
```

List_Remove

```
/*
 * ----------------------------------------------------------------------
 *
 * List_Remove --
 *
 *         Remove a list element from the list in which it is contained.
 *
 * Results:
 *         No value is returned.
 *
 * Side effects:
 *         The given structure is removed from its containing list.
 *
 * ----------------------------------------------------------------------
 */
void
List_Remove(itemPtr)
    register        List_Links *itemPtr;        /* list element to remove */
{
    if (itemPtr == itemPtr->nextPtr) {
        return;
    }
    itemPtr->prevPtr->nextPtr = itemPtr->nextPtr;
    itemPtr->nextPtr->prevPtr = itemPtr->prevPtr;
}
```

```
/*
 * ----------------------------------------------------------------------
 *
 * List_Move --
 *
 *      Move the list element referenced by itemPtr to follow destPtr.
 *
 * Results:
 *      No value is returned.
 *
 * Side effects:
 *      List ordering is modified.
 *
 * ----------------------------------------------------------------------
 */
void
List_Move(itemPtr, destPtr)
    register List_Links *itemPtr; /* list element to be moved */
    register List_Links *destPtr; /* element after which it is to be placed */
{
    /*
     * It is conceivable that someone will try to move a list element to
     * be after itself.
     */
    if (itemPtr == destPtr) {
        return;
    }
    List_Remove(itemPtr);
    List_Insert(itemPtr, destPtr);
}
```

```
/*
 * ----------------------------------------------------------------------
 *
 * List_Init --
 *
 *	Initialize a header pointer to point to an empty list.  The List_Links
 *	structure must already be allocated.
 *
 * Results:
 *	No value is returned.
 *
 * Side effects:
 *	The header's pointers are modified to point to itself.
 *
 * ----------------------------------------------------------------------
 */
void
List_Init(headerPtr)
    register List_Links *headerPtr;	/* Pointer to a List_Links structure
					   to be header */
{
    headerPtr->nextPtr = headerPtr;
    headerPtr->prevPtr = headerPtr;
}
```

List_Init

```
#!/bin/csh -f
#
# A script to "compile" PLM code into a binary that runs on
# the SPUR simulator (barb).
#
onintr end
set nonomatch
set DIR = ~andrew/spur
set POST = $DIR/Proc/postproc
set PRE = $DIR/Proc/preproc
set HEADERS = $DIR/Headers
set INCLUDE = "-I$HEADERS -P"
set BARB = ~zorn/sim/barb
set CODE2 = ~zorn/sim/barb/th/code2.s
set SAS = $BARB/sas/sas
set SLD = $BARB/sas/sld

@ numargs = $#argv
set Preproc = 0
set Postproc = 0
set Assemble = 0
set Load = 0

top:
switch ($1:q)
    case "-help":
        echo "Options:   -pre   == preproc"
        echo "            -post  == postproc"
        echo "            -load  == load"
        echo "            -asm   == asssemble"
        goto end
        breaksw
    case "-pre":
        shift
        set Preproc = 1
        goto top
        breaksw
    case "-post":
        shift
        set Postproc = 1
        goto top
        breaksw
    case "-asm":
        shift
        set Assemble = 1
        goto top
        breaksw
    case "-load":
        shift
        set Load = 1
        goto top
        breaksw
    default:
        set longname=$1
        set name="`basename $1:t .w`"
        if ($numargs == 1) then
            set Preproc = 1
            set Postproc = 1
            set Assemble = 1
            set Load = 1
        endif
        breaksw
endsw

if ($Preproc == 1) then
    echo "Pre-process:"
```

```
     rm -f $name.a $name.spur
     cp $HEADERS/header.h $name.spur
     $PRE $name.const < $longname >> $name.spur
     cat $HEADERS/trailer.h $name.const >> $name.spur
     rm -f $name.const
endif

if ($Postproc == 1) then
     echo "Post-process:"
     cat $name.spur | /lib/cpp $INCLUDE | $POST > $name.a
endif

if ($Assemble == 1) then
     echo "Assemble:"
     rm -f temp $name.s $name.th
     $SAS < $name.a > $name.s
     cat $CODE2 $name.s > temp
     as -o temp2 temp
endif

if ($Load == 1) then
     echo "Load:"
     $SLD -o $name.th temp2
endif

end:
```

Appendix 2: Listings of Macro-Expansions

This appendix contains listings of the macro-expansions used to convert each of the PLM instructions into SPUR code and the library functions used by the programs. The file instructions.h contains the definitions of the macros for each PLM instruction. The file funcs.a contains the library functions.

```
#include "instructions.h"
#include "defs.h"

/*
 *  Initialize code:
 *
 */

.org    0x3000

/*
 * Turn off tag traps.
 */

wr_special      upsw, r0, $0x880

/*
 * Initialize all of the registers.
 */

add     CONST_PTR, r0, $0x780
sll     CONST_PTR, CONST_PTR, $3
sll     CONST_PTR, CONST_PTR, $3
wr_tag  CONST_PTR, $cut_0

/*
 * Put 64K into T1.
 */

add     T1, r0, $0x800
sll     T1, T1, $3
sll     T1, T1, $2

/*
 * Put 0x500000 in T2.
 */

add     T2, r0, $0x500
sll     T2, T2, $3
sll     T2, T2, $3
sll     T2, T2, $3
sll     T2, T2, $3

/*
 * Put 8K in T3
 */

add     T3, r0, $1024
sll     T3, T3, $3
```

```
/*
 * All data starts at 0x500000.  Each stack and such is put at locations
 * similar to those used in the PLM except that each is multiplied by 8
 * since the PLM is word addressed and SPUR is byte addressed with 8 bytes
 * per word.
 */

add     H, T2, $1024
add     E, T2, T1
add     B, E, 0
st_32   B, CONST_PTR, $stack_bottom
add     TR, E, T3
add     CP, r0, 0
add     S, H, 0
add     T4, TR, T3
st_32   T4, CONST_PTR, $PDL_offset
st_32   T4, CONST_PTR, $stack_offset
add     T4, T2, $128
st_32   T4, CONST_PTR, $H2_offset

/*---------------------------------------------------------------------*/
```

```
#define get_stack_base(reg) \
        ld_32   reg, CONST_PTR, $stack_bottom

/**************** basics *********************/

#define deref(reg, temp) \
        rd_tag          temp, reg; \
        and             temp, temp, $type_mask; \
        cmp_br_delayed  neq, temp, $var_type, @103f; \
        rd_tag          temp, reg; \
        ld_40           reg, reg, 0; \
@101:   rd_tag          temp, reg; \
        and             temp, temp, $type_mask; \
        cmp_br_delayed  neq, temp, $var_type, @103f; \
        rd_tag          temp, reg; \
        ld_40           temp, reg, 0; \
        cmp_br_delayed  eq, temp, reg, @102f; \
        Nop; \
        jump            @101b$w; \
        add             reg, temp, 0; \
@102:   rd_tag          temp, reg; \
@103:

#define trail(reg) \
        st_40           reg, TR, 0; \
        add             TR, TR, $8

#define decdr(reg) \
        tag_cmp_br_delayed      ne_tag, reg, $list_cdr_type, @101f; \
        add_nt          S, S, $8; \
        add_nt          S, reg, 0; \
        ld_40           reg, S, 0; \
        jump            @102f$w; \
        add_nt          S, S, $8; \
@101:   tag_cmp_br_delayed      ne_tag, reg, $nil_const_type, @102f; \
        Nop; \
        ld_40           reg, CONST_PTR, $nil_offset; \
@102:

#define Pop_ChoicePoint(temp) \
        add             OLD_H, H, 0; \
        add             OLD_E, E, 0; \
        add             OLD_TR, TR, 0; \
        add             OLD_CP, CP, 0; \
        add             OLD_BP, BP, 0; \
        rd_special      temp, cpu_pc; \
        return          temp, $12; \
        Nop
```

```
#define write_Nreg(reg) \
        st_32           reg, CONST_PTR, $N_reg_offset
#define read_Nreg(reg) \
        ld_32           reg, CONST_PTR, $N_reg_offset

#define make_const(reg, const) \
        add             reg, CONST_PTR, $const; \
        wr_tag          reg, $const_type

#define make_nil(reg) \
        ld_40           reg, CONST_PTR, $nil_offset

#define bind(binding, bound, temp) \
        rd_tag          temp, bound; \
        and             temp, temp, $cdr_type; \
        cmp_br_delayed  neq, temp, $cdr_type, @101f; \
        rd_tag          temp, binding; \
        or              temp, temp, $cdr_type; \
        wr_tag          binding, temp; \
@101:   st_40           binding, bound, 0; \
        add             temp, bound, 0; \
        wr_tag          temp, 0; \
! Trail in Bind; \
        trail(temp)

#define call_unify() \
        jump            unify$w; \
        rd_special      T3, cpu_pc


/******************* allocate **********************/

#define allocate() \
! Allocate; \
        add     T2, E, 0; \
        cmp_br_delayed  ge, T2, B, @1f; \
        Nop; \
        jump    @2f$w; \
        add     E, B, 0; \
@1:     read_Nreg(T1); \
        add     E, E, $env_size; \
        add     E, E, T1; \
@2:     st_40   CP, E, $saved_CP_offset; \
        st_40   T2, E, $saved_E_offset; \
        st_40   T1, E, $saved_N_offset; \
        rd_tag  T1, CONST_PTR; \
        wr_tag  B, T1; \
        st_40   B, E, $saved_B_offset
```

```
/*********************** deallocate ************************/

#define deallocate() \
! Deallocate; \
        ld_40   CP, E, $saved_CP_offset; \
        ld_40   T1, E, $saved_N_offset; \
        write_Nreg(T1); \
        ld_40   E, E, $saved_E_offset

/******************* call *********************/

#define call_proc(label, N_value) \
! Call_proc; \
        wr_tag          CONST_PTR, $cut_0; \
        add             T1, r0, $N_value; \
        sll             T1, T1, $3; \
        write_Nreg(T1); \
        jump            label/**/$w ;\
        rd_special      CP, cpu_pc

/******************* cut *********************/

#define cut() \
! Cut; \
        get_stack_base(T2); \
@1:     cmp_br_delayed  eq, B, T2, @3f; \
        ld_40           T1, E, $saved_B_offset; \
        cmp_br_delayed  eq, B, T1, @2f; \
        Nop; \
        Pop_ChoicePoint(T1); \
        jump            @1b$w; \
        Nop; \
@2:     tag_cmp_br_delayed      ne_tag, T1, $cut_1, @3f; \
        Nop; \
        Pop_ChoicePoint(T1); \
@3:     wr_tag          CONST_PTR, $cut_0; \
        st_40           B, E, $saved_B_offset

/******************* cutd *********************/

#define cutd(label) \
! Cutd; \
@1:     ld_32           T1, CONST_PTR, $label; \
        cmp_br_delayed  eq, T1, OLD_BP, @2f; \
        Nop; \
        Pop_ChoicePoint(T1); \
        jump            @1b$w; \
        Nop; \
@2:     Pop_ChoicePoint(T1)
```

```
/******************** call_fail ***********************/

#define call_fail() \
! Call_fail; \
        jump            fail$w; \
        Nop; \

/*********************** ESCAPES *************************/

/*********************** escape ********************/

#define escape(fcn, reg) \
! Escape; \
        st_32           r28, CONST_PTR, $save_r28_offset; \
        st_32           r26, CONST_PTR, $save_r10_offset; \
        st_32           r9, CONST_PTR, $save_r9_offset; \
        add             r28, r0, $fcn; \
        call            (-71 & 0xffffff); \
        Nop; \
        add             reg, r28, 0; \
        ld_32           r28, CONST_PTR, $save_r28_offset; \
        ld_32           r26, CONST_PTR, $save_r10_offset; \
        ld_32           r9, CONST_PTR, $save_r9_offset

/*********************** comparison ********************/

#define equal() \
!Equal; \
        add             T1, A1, 0; \
        add             T2, A2, 0; \
        deref(T1, T3); \
        deref(T2, T4); \
        call_unify(); \
        cmp_br_delayed  eq, T4, $1, @1f; \
        Nop; \
        call_fail(); \
@1:

#define compare(op) \
! Compare; \
        deref(A1, T1); \
        deref(A2, T2); \
        cmp_br_delayed  neq, T1, $const_num_type, @1f; \
        Nop; \
        cmp_br_delayed  neq, T2, $const_num_type, @1f; \
        Nop; \
        cmp_br_delayed  op, A1, A2, @2f; \
        Nop; \
```

```
@1:     call_fail(); \
@2:

#define less_than() \
! Less_than; \
        compare(lt)

#define less_than_or_equal() \
! Less_than_or_equal; \
        compare(le)

#define greater_than() \
! Greater_than; \
        compare(gt)

#define greater_than_or_equal() \
! Greater_than_or_equal; \
        compare(ge)

/********************* is *****************************/

#define is_escape() \
! Is; \
        deref(A1, T1); \
        deref(A2, T2); \
        deref(A4, T3); \
        and             T1, T1, $type_mask; \
        cmp_br_delayed  eq, T1, $var_type, @1f; \
        Nop; \
        cmp_br_delayed  neq, T1, $const_type, @2f; \
        Nop; \
@1:     tag_cmp_br_delayed      ne_tag, A2, $const_num_type, @2f; \
        rd_tag          T4, A3; \
        tag_cmp_br_delayed      ne_tag, A4, $const_num_type, @2f; \
        and             T4, T4, $const_type; \
        cmp_br_delayed  neq, T4, $const_type, @2f; \
        Nop; \
        escape(ARITH, T4); \
        tag_cmp_br_delayed      ne_tag, A1, $const_num_type, @3f; \
        Nop; \
        cmp_br_delayed  eq, A1, T4, @3f; \
        Nop; \
@2:     call_fail(); \
@3:     cmp_br_delayed  neq, T1, $var_type, @4f; \
        Nop; \
        wr_tag          T4, $const_num_type; \
        bind(T4, A1, T1); \
@4:
```

```
#define is_2_escape() \
! Is_2; \
        jump            is_2$w; \
        rd_special      T9, cpu_pc

/************************* esc_call **************************/

/*
 * Instead of providing the escape call routine I provide two primitives
 * instead.  The first shifts up all of the argument registers.
 * The second does the jump.  All escape calls have to be translated by
 * hand to use these primitives.
 */

#define esc_shift_regs() \
!Esc_shift_regs; \
        add             A1, A2, 0; \
        add             A2, A3, 0; \
        add             A3, A4, 0; \
        add             A4, A5, 0; \
        add             A5, A6, 0; \
        add             A6, A7, 0; \
        add             A7, A8, 0

#define esc_jump(label) \
!Esc_esc_jump; \
        jump            label/**/$w; \
        rd_special      CP, cpu_pc

/************************* var **************************/

#define var_escape() \
! Var_escape; \
        add             T2, A1, 0; \
        deref(T2, T1); \
        and             T1, T1, $type_mask; \
        cmp_br_delayed  eq, T1, $var_type, @1f; \
        Nop; \
        call_fail(); \
@1:

/************************* setter ***********************/

#define setter() \
! Setter; \
        add             T2, A1, 0; \
        add             T1, A2, 0; \
        deref(T2, T3); \
        deref(T1, T3); \
```

```
        tag_cmp_br_delayed      eq_tag, T1, $const_num_type, @1f; \
        Nop; \
        call_fail(); \
@1:     cmp_br_delayed          le, T1, $15, @2f; \
        Nop; \
        add             T1, r0, 0; \
        wr_tag          T1, 0; \
@2:     wr_tag          T1, $var_type; \
        st_40           T1, T1, 0; \
        add             T8, H, 0; \
        ld_32           H, r9, $H2_offset; \
        jump            esc_unify$w; \
        rd_special      T3, cpu_pc; \
        st_32           H, r9, $H2_offset; \
        add             H, T8, 0; \
@3:

/************************* access **********************/

#define access() \
! Access; \
        add             T2, A1, 0; \
        add             T1, A2, 0; \
        deref(T2, T3); \
        deref(T1, T3); \
        tag_cmp_br_delayed      eq_tag, T1, $const_num_type, @1f; \
        Nop; \
        call_fail(); \
@1:     cmp_br_delayed          le, T1, $15, @2f; \
        Nop; \
        add             T1, r0, 0; \
        wr_tag          T1, 0; \
@2:     ld_40           T1, T1, 0; \
        call_unify(); \
        cmp_br_delayed  neq, T4, 0, @3f; \
        Nop; \
        call_fail(); \
@3:

/****************** integer **********************/

#define integer() \
! Integer; \
        add             T1, A1, 0; \
        deref(T1, T2); \
        tag_cmp_br_delayed      eq_tag, T1, $const_num_type, @1f; \
        Nop; \
        call_fail(); \
@1:
```

```
/******************* execute *************************/

#define execute(label) \
! Execute; \
        jump            label/**/$w;\
        Nop

/****************** get_variable_var ********************/

#define get_variable_reg(An, Ai) \
        add             An, Ai, 0

#define get_variable_var(env_offset, Ai) \
        st_40           Ai, E, $env_offset

/****************** get_constant **********************/

#define get_constant_string(const, Ai) \
! Get_constant_string; \
        add_nt          T2, Ai, 0; \
        deref(T2, T3); \
        make_const(T1, const); \
        call_unify(); \
        cmp_br_delayed  eq, T4, $1, @1f; \
        Nop; \
        jump            fail$w; \
        Nop; \
@1:

#define get_constant_number(const, Ai) \
! Get_constant_number; \
        add_nt          T2, Ai, 0; \
        deref(T2, T3); \
        ld_32           T1, CONST_PTR, $const; \
        wr_tag          T1, $const_num_type; \
        call_unify(); \
        cmp_br_delayed  eq, T4, $1, @1f; \
        Nop; \
        jump            fail$w; \
        Nop; \
@1:

/******************* get_list **********************/

#define get_list(Ai) \
! Get_list; \
        add             T2, Ai, 0;\
        deref(T2, T3);\
```

```
        st_32           r0, CONST_PTR, $smode_offset; \
        and             T3, T3, $type_mask;\
        cmp_br_delayed  neq, T3, $list_type, @1f;\
        Nop;\
        add             S, T2, 0;\
        jump            @4f$w;\
        wr_tag          S, $read_mode;\
@1:     tag_cmp_br_delayed      ne_tag, T2, $unbound_var_type, @2f;\
        sub             T4, H, $8;\
        cmp_br_delayed  neq, T2, T4, @2f;\
        wr_tag          T4, $cdr_type;\
        sub             H, H, $8;\
! Trail in Get_list; \
        trail(T4);\
        jump            @4f$w;\
        wr_tag          S, $write_mode;\
@2:     cmp_br_delayed  neq, T3, $var_type, @3f;\
        add             T1, H, 0;\
        wr_tag          T1, $list_type;\
        call_unify();\
        jump            @4f$w;\
        wr_tag          S, $write_mode;\
@3:     jump            fail$w;\
        Nop;\
@4:

/****************** get_nil **********************/

#define get_nil(Ai) \
! Get_nil; \
        add             T1, Ai, 0; \
        deref(T1, T2); \
        ld_40           T2, CONST_PTR, $nil_offset; \
        wr_tag          T2, $const_type; \
        call_unify(); \
        cmp_br_delayed  eq, T4, $1, @1f; \
        Nop; \
        jump            fail$w; \
        Nop; \
@1:

/****************** get_structure **********************/

#define get_structure(struct, Ai) \
! Get_structure; \
        add             T1, Ai, 0; \
        deref(T1, T3); \
        make_const(T2, struct); \
```

```
        add              T4, r0, $1; \
        st_32            T4, CONST_PTR, $smode_offset; \
        and              T3, T3, $type_mask; \
        cmp_br_delayed   neq, T3, $var_type, @1f; \
        Nop; \
        st_40            T2, H, 0; \
        add              T2, H, 0; \
        wr_tag           T2, $struct_type; \
        add              H, H, $8; \
        st_40            T2, T1, 0; \
        wr_tag           T1, 0; \
! Trail in Get_structure; \
        trail(T1); \
        jump             @3f$w; \
        wr_tag           S, $write_mode; \
@1:     cmp_br_delayed   neq, T3, $struct_type, @2f; \
        ld_40            T3, T1, 0; \
        cmp_br_delayed   neq, T3, T2, @2f; \
        Nop; \
        add              S, T1, $8; \
        jump             @3f$w; \
        wr_tag           S, $read_mode; \
@2:     call_fail(); \
@3:

/******************* get_value *********************/

#define get_value_reg(An, Ai) \
! Get_value_reg; \
        add_nt           T2, An, 0; \
        deref(T2, T3); \
        add_nt           T1, Ai, 0; \
        deref(T1, T3); \
        add              T9, T1, 0; \
        call_unify(); \
        cmp_br_delayed   eq, T4, $1, @1f; \
        Nop; \
        jump             fail$w; \
        Nop; \
@1:     add_nt           An, T9, 0; \

#define get_value_var(env_offset, Ai) \
! Get_value_var; \
        ld_40            T2, E, $env_offset; \
        deref(T2, T3); \
        add_nt           T1, Ai, 0; \
        deref(T1, T3); \
        call_unify(); \
        cmp_br_delayed   eq, T4, $1, @1f; \
```

```
        Nop; \
        jump            fail$w; \
        Nop; \
@1:

/******************* mark **********************/
/*
 * An instruction that isn't used anymore?
 */

#define mark() \
! Mark; \
        Nop

/******************* pause *********************/
/*
 * An instruction that isn't used anymore?
 */

#define pause() \
! Pause; \
        Nop

/*************** procedure ********************/

#define procedure(name) \
! Procedure; \
name:

/*************** proceed ********************/

#define proceed() \
! Proceed; \
        cmp_br_delayed  eq, CP, 0, @1f; \
        jump_reg        CP, $4; \
        wr_tag          CONST_PTR, $cut_0; \
@1:     jump            success$w; \
        Nop

/****************** put_constant *********************/

#define put_constant_string(const, Ai) \
! Put_constant_string; \
        make_const(Ai, const)

#define put_constant_number(const, Ai) \
! Put_constant_number; \
        ld_32           Ai, CONST_PTR, $const; \
        wr_tag          Ai, $const_num_type
```

```
/****************** put_list ********************/

#define put_list(Ai) \
! Put_list; \
        wr_tag          S, $write_mode; \
        st_32           r0, CONST_PTR, $smode_offset; \
        add_nt          Ai, H, 0; \
        wr_tag          Ai, $list_type

/******************* put_nil *********************************/

#define put_nil(Ai) \
! Put_nil; \
        make_nil(Ai); \
        wr_tag          Ai, $const_type

/******************** put_structure ************************/

#define put_structure(struct, Ai) \
! Put_structure; \
        add             T4, r0, $1; \
        st_32           T4, CONST_PTR, $smode_offset; \
        make_const(T1, struct); \
        add             Ai, H, 0; \
        wr_tag          Ai, $struct_type; \
        st_40           T1, H, 0; \
        add             H, H, $8; \
        wr_tag          S, $write_mode

/******************** put_unsafe_value ************************/

#define put_unsafe_value(env_offset, Ai) \
! Put_unsafe_value; \
        ld_40           T1, E, $env_offset; \
        deref(T1, T2); \
        and             T2, T2, $var_type; \
        cmp_br_delayed  neq, T2, $var_type, @1f; \
        Nop; \
        cmp_br_delayed  le, T1, E, @1f; \
        Nop; \
        add             T1, H, 0; \
        wr_tag          T1, $var_type; \
        st_40           T1, H, 0; \
        add             H, H, $8; \
@1:     add_nt          Ai, T1, 0

/**************** put_value ********************/
```

```
#define put_value_var(env_offset, Ai) \
! Put_value_var; \
        ld_40           Ai, E, $env_offset

#define put_value_reg(An, Ai) \
! Put_value_reg; \
        add_nt          Ai, An, 0

/**************** put_variable ***********************/

#define put_variable_var(env_offset, Ai) \
! Put_variable_var; \
        add_nt          Ai, E, $env_offset; \
        wr_tag          Ai, $var_type; \
        st_40           Ai, E, $env_offset

#define put_variable_reg(An, Ai) \
! Put_variable_reg; \
        add_nt          Ai, H, 0; \
        wr_tag          Ai, $var_type; \
        add_nt          An, Ai, 0; \
        st_40           Ai, H, 0; \
        add_nt          H, H, $8

/********************** quit ********************/

#define quit() \
! Quit; \
        add             r28, r0, 0; \
        jump            start$w

#define end() quit()

/****************** retry *********************/

#define retry(label) \
! Retry; \
        wr_tag          CONST_PTR, $cut_1; \
        ld_32           T1, CONST_PTR, $label; \
        rd_special      T2, cpu_pc; \
        jump_reg        T1, 0; \
        add             OLD_BP, T2, $12

/****************** retry_me_else *********************/

#define retry_me_else(label) \
! Retry_me_else; \
        ld_32           OLD_BP, CONST_PTR, $label; \
        wr_tag          CONST_PTR, $cut_1
```

```
/******************** switch_on_constant *********************/

#define switch_on_constant(mask, label) \
! Switch_on_constant; \
        deref(A1, T3); \
        and             T1, T3, $type_mask; \
        cmp_br_delayed  neq, T1, $const_type, @4f; \
        ld_32           T1, CONST_PTR, $mask; \
        add             T2, CONST_PTR, $label; \
        and             T3, T3, $const_num_type; \
@1:     cmp_br_delayed  le, T1, 0, @4f; \
        ld_32           T4, T2, 0; \
        add             T4, T4, CONST_PTR; \
        cmp_br_delayed  eq, A1, T4, @3f; \
        Nop; \
@2:     sub             T1, T1, $2; \
        jump            @1b$w; \
        add             T2, T2, $12; \
@3:     ld_32           T4, T2, $4; \
        cmp_br_delayed  neq, T3, T4, @2b; \
        ld_32           T4, T2, $8; \
        jump_reg        T4, 0; \
        Nop; \
@4:     call_fail()

/******************** switch_on_structure *********************/

#define switch_on_structure(mask, label) \
! Switch_on_structure; \
        deref(A1, T1); \
        and             T1, T1, $type_mask; \
        cmp_br_delayed  neq, T1, $struct_type, @3f; \
        ld_32           T1, CONST_PTR, $mask; \
        add             T2, CONST_PTR, $label; \
        ld_40           T4, A1, 0; \
@1:     cmp_br_delayed  le, T1, 0, @3f; \
        ld_32           T3, T2, 0; \
        add             T3, T3, CONST_PTR; \
        cmp_br_delayed  eq, T4, T3, @2f; \
        sub             T1, T1, $2; \
        jump            @1b$w; \
        add             T2, T2, $8; \
@2:     ld_32           T4, T2, $4; \
        jump_reg        T4, 0; \
        Nop; \
@3:     call_fail()
```

```
/******************** switch_on_term *********************/

#define switch_on_term(const_label,list_label,struct_label) \
! Switch_on_term; \
        add_nt          T1, A1, 0; \
        deref(T1, T2); \
        and             T2, T2, $type_mask; \
        cmp_br_delayed  neq, T2, $const_type, @1f; \
        Nop; \
        jump            const_label/**/$w; \
        Nop; \
@1:     cmp_br_delayed  neq, T2, $list_type, @2f; \
        Nop; \
        jump            list_label/**/$w; \
        Nop; \
@2:     cmp_br_delayed  neq, T2, $struct_type, @3f; \
        Nop; \
        jump            struct_label/**/$w; \
        Nop; \
@3:

/******************** trust **********************/

#define trust(label) \
! Trust; \
        Pop_ChoicePoint(T1); \
        wr_tag          CONST_PTR, $cut_0; \
        ld_32           T1, CONST_PTR, $label; \
        jump_reg        T1, $0; \
        Nop


/********************* trust_me_else **********************/

#define trust_me_else(label) \
! Trust_me_else; \
        Pop_ChoicePoint(T1); \
        wr_tag          CONST_PTR, $cut_0

/******************** try **********************/

#define try(label) \
! Try; \
        add_nt          SAVE_AX1, A1, 0; \
        add_nt          SAVE_AX2, A2, 0; \
        add_nt          SAVE_AX3, A3, 0; \
        add_nt          SAVE_AX4, A4, 0; \
        add_nt          SAVE_AX5, A5, 0; \
        add_nt          SAVE_AX6, A6, 0; \
```

```
        add_nt          SAVE_AX7, A7, 0; \
        add_nt          SAVE_AX8, A8, 0; \
        read_Nreg(SAVE_N); \
        st_32           r26, CONST_PTR, $save_r10_offset; \
        st_32           r9, CONST_PTR, $save_r9_offset; \
        call            @1f$w; \
        Nop; \
@1:     ld_32           r10, CONST_PTR, $save_r10_offset; \
        ld_32           r9, CONST_PTR, $save_r9_offset; \
        add_nt          E, OLD_E, 0; \
        add_nt          TR, OLD_TR, 0; \
        add_nt          H, OLD_H, 0; \
        add_nt          CP, OLD_CP, 0; \
        add_nt          BP, OLD_BP, 0; \
        add             B, OLD_B, 0; \
        cmp_br_delayed  lt, E, B, @1f; \
        Nop; \
        read_Nreg(T1); \
        add             B, T1, E; \
        add             B, B, $env_size; \
@1:     add             B, B, $4; \
        wr_tag          CONST_PTR, $cut_1; \
        ld_32           T1, CONST_PTR, $label; \
        rd_special      T2, cpu_pc; \
        jump_reg        T1, 0; \
        add             OLD_BP, T2, $12

/*********************** try_me_else *************************/

#define try_me_else(label) \
! Try_me_else; \
        add_nt          SAVE_AX1, A1, 0; \
        add_nt          SAVE_AX2, A2, 0; \
        add_nt          SAVE_AX3, A3, 0; \
        add_nt          SAVE_AX4, A4, 0; \
        add_nt          SAVE_AX5, A5, 0; \
        add_nt          SAVE_AX6, A6, 0; \
        add_nt          SAVE_AX7, A7, 0; \
        add_nt          SAVE_AX8, A8, 0; \
        read_Nreg(SAVE_N); \
        st_32           r26, CONST_PTR, $save_r10_offset; \
        st_32           r9, CONST_PTR, $save_r9_offset; \
        call            @1f$w; \
        Nop; \
@1:     ld_32           r10, CONST_PTR, $save_r10_offset; \
        ld_32           r9, CONST_PTR, $save_r9_offset; \
        add_nt          E, OLD_E, 0; \
        add_nt          TR, OLD_TR, 0; \
        add_nt          H, OLD_H, 0; \
```

```
        add_nt          CP, OLD_CP, 0; \
        add_nt          BP, OLD_BP, 0; \
        add             B, OLD_B, 0; \
        cmp_br_delayed  lt, E, B, @1f; \
        Nop; \
        read_Nreg(T1); \
        add             B, T1, E; \
        add             B, B, $env_size; \
@1:     add             B, B, $4; \
        ld_32           OLD_BP, CONST_PTR, $label; \
        wr_tag          CONST_PTR, $cut_1

/********************** unify_cdr **************************/

#define unify_cdr_reg(An) \
! Unify_cdr_reg; \
        tag_cmp_br_delayed      eq_tag, S, $read_mode, @1f; \
        add             T1, H, 0; \
        wr_tag          T1, $unbound_var_type; \
        st_40           T1, H, 0; \
        add_nt          An, H, 0; \
        wr_tag          An, $var_type; \
        jump            @3f$w; \
        add_nt          H, H, $8; \
@1:     ld_40           An, S, 0; \
        rd_tag          T2, An; \
        and             T2, T2, $cdr_type; \
        cmp_br_delayed  eq, T2, $cdr_type, @2f; \
        Nop; \
        add_nt          An, S, 0; \
        jump            @3f$w; \
        wr_tag          An, $list_type; \
@2:     rd_tag          T2, An; \
        and             T2, T2, $not_cdr_type; \
        wr_tag          An, T2; \
@3:

#define unify_cdr_var(env_offset) \
! Unify_cdr_var; \
        tag_cmp_br_delayed      eq_tag, S, $read_mode, @1f; \
        add             T1, H, 0; \
        wr_tag          T1, $unbound_var_type; \
        st_40           T1, H, 0; \
        wr_tag          T1, $var_type; \
        st_40           T1, E, $env_offset; \
        jump            @3f$w; \
        add_nt          H, H, $8; \
@1:     ld_40           T1, S, 0; \
        rd_tag          T2, T1; \
```

```
        and             T2, T2, $cdr_type; \
        cmp_br_delayed  eq, T2, $cdr_type, @2f; \
        Nop; \
        add_nt          T1, S, 0; \
        wr_tag          T1, $list_type; \
        jump            @3f$w; \
        st_40           T1, E, $env_offset; \
@2:     rd_tag          T2, T1; \
        and             T2, T2, $not_cdr_type; \
        wr_tag          T1, T2; \
        st_40           T1, E, $env_offset; \
@3:

/********************** unify_constant ********************/

#define unify_constant_string(const) \
! Unify_constant_string; \
        make_const(T8, const); \
        jump            unify_const_func$w; \
        rd_special      T9, cpu_pc

#define unify_constant_number(const) \
! Unify_constant_number; \
        ld_32           T8, CONST_PTR, $const; \
        wr_tag          T8, $const_num_type; \
        jump            unify_const_func$w; \
        rd_special      T9, cpu_pc

/************************* unify_nil ****************************/

#define unify_nil() \
! Unify_nil; \
        tag_cmp_br_delayed      eq_tag, S, $write_mode, @2f; \
        Nop; \
        ld_40           T1, S, 0; \
        rd_tag          T2, T1; \
        and             T2, T2, $cdr_type; \
        cmp_br_delayed  neq, T2, $cdr_type, @1f; \
        make_nil(T2); \
        call_unify(); \
        cmp_br_delayed  neq, T4, 0, @3f; \
        Nop; \
@1:     jump            fail$w; \
        Nop; \
@2:     make_nil(T1); \
        st_40           T1, H, 0; \
        add             H, H, $8; \
@3:
```

```
/**************** unify_value *************************/

#define unify_unsafe_value_reg(An)          unify_value_reg(An)
#define unify_unsafe_value_var(env_offset) unify_value_var(env_offset)

#define unify_value_reg(An) \
! Unify_value_reg; \
        add             T8, An, 0; \
        jump            unify_value$w; \
        rd_special      T9, cpu_pc

#define unify_value_var(env_offset) \
! Unify_value_var; \
        ld_40           T8, E, $env_offset; \
        jump            unify_value$w; \
        rd_special      T9, cpu_pc

/******************** unify_variable ********************/

#define unify_variable() \
        tag_cmp_br_delayed      eq_tag, S, $write_mode, @2f; \
        Nop; \
        ld_40           T1, S, 0 ; \
        decdr(T1); \
        rd_tag          T3, T1; \
        and             T4, T3, $cdr_type; \
        cmp_br_delayed  neq, T4, $cdr_type, @3f; \
        Nop; \
        tag_cmp_br_delayed      eq_tag, T1, $unbound_var_type, @1f; \
        Nop; \
        jump            fail$w; \
        Nop; \
@1:     add             T2, H, 0; \
        wr_tag          T2, $list_cdr_type; \
        call_unify(); \
        wr_tag          S, $write_mode; \
@2:     add             T1, H, 0; \
        wr_tag          T1, $var_type; \
        st_40           T1, H, 0; \
        add             H, H, $8

#define unify_variable_reg(An) \
! Unify_variable_reg; \
        unify_variable(); \
@3:     add             An, T1, 0

#define unify_variable_var(env_offset) \
! Unify_variable_var; \
        unify_variable(); \
```

```
@3:     st_40               T1, E, $env_offset

/******************** unify_void ***********************/

#define unify_void(num) \
! Unify_void; \
        add                 T5, r0, $num; \
        tag_cmp_br_delayed       eq_tag, S, $write_mode, @5f; \
        ld_32               T1, E, $smode_offset; \
        cmp_br_delayed      eq, T1, $0, @1f; \
        add                 T1, r0, $num; \
        sll                 T1, T1, $3; \
        jump                @7f$w; \
        add                 S, S, T1; \
@1:     add                 T1, r0, 0; \
@2:     cmp_br_delayed      ge, T1, T5, @7f; \
        Nop; \
        ld_40               T2, S, 0; \
        decdr(T2); \
        rd_tag              T3, T2; \
        and                 T3, T3, $cdr_type; \
        cmp_br_delayed      neq, T3, $cdr_type, @4f; \
        rd_tag              T4, T2; \
        and                 T4, T4, $var_type; \
        cmp_br_delayed      eq, T4, $var_type, @3f; \
        Nop; \
        call_fail(); \
@3:     wr_tag              S, $write_mode; \
        sub                 T5, T5, T1; \
        add                 T1, H, 0; \
        wr_tag              T1, $list_cdr_type; \
        jump                @5f$w; \
        st_40               T1, T2, 0; \
@4:     jump                @2b$w; \
        add                 T1, T1, $1; \
@5:     add                 T1, r0, 0; \
@6:     cmp_br_delayed      ge, T1, T5, @7f; \
        add                 T2, H, 0; \
        wr_tag              T2, $var_type; \
        st_40               T2, H, 0; \
        add                 H, H, $8; \
        jump                @6b$w; \
        add                 T1, T1, $1; \
@7:
```

```
#define CONST_PTR       r2
#define A1              r3
#define X1              r3
#define A2              r4
#define X2              r4
#define A3              r5
#define X3              r5
#define A4              r6
#define X4              r6
#define A5              r7
#define X5              r7
#define A6              r8
#define X6              r8
#define A7              r9
#define X7              r9
#define A8              r1
#define X8              r1

#define OLD_BP          r10
#define OLD_E           r11
#define OLD_TR          r12
#define OLD_H           r13
#define OLD_B           r14
#define OLD_CP          r15

#define S               r16
#define SAVE_AX1        r16
#define T1              r17
#define SAVE_AX2        r17
#define T2              r18
#define SAVE_AX3        r18
#define T3              r19
#define SAVE_AX4        r19
#define T4              r20
#define SAVE_AX5        r20
#define T5              r21
#define SAVE_AX6        r21
#define T6              r22
#define SAVE_AX7        r22
#define T7              r23
#define SAVE_AX8        r23
#define T8              r24
#define SAVE_N          r24
#define T9              r25

#define BP              r26
#define E               r27
#define TR              r28
#define H               r29
```

```
#define B                 r30
#define CP                r31

#define type_mask          0x03
#define list_type          0x00
#define struct_type        0x01
#define var_type           0x02
#define const_type         0x03
#define unbound            0x10
#define bound_var_type     0x00
#define unbound_var_type   0x12
#define cdr_type           0x10
#define not_cdr_type       0x0f
#define list_cdr_type      0x10
#define nil_type           0x10
#define nil_const_type     0x13
#define num_type           0x08
#define const_num_type     0x0b
#define cut_0              0x00
#define cut_1              0x01
#define read_mode          0x00
#define write_mode         0x01

#define saved_E_offset              0
#define saved_CP_offset             8
#define saved_B_offset              16
#define saved_N_offset              24
#define env_size                    32
#define Y1                          32
#define Y2                          40
#define Y3                          48
#define Y4                          56
#define Y5                          64
#define Y6                          72
#define Y7                          80
#define Y8                          88
#define Y9                          96
#define Y10                         104

#define nil_offset                  0
#define stack_offset                8
#define save_r28_offset             12
#define save_r10_offset             16
#define save_r9_offset              20
#define N_reg_offset                24
#define smode_offset                28
#define H2_offset                   32
#define PDL_offset                  36
#define stack_bottom                40
```

```
#define first_const_offset      44

#define WRITE           0
#define NL              1
#define ARITH           2
```

```
/*--------------------------------------------------------------------*/

#include "funcs.a"

/*
 * Initialize the constant table:
 *
 * 1) Nil pointer
 * 2) Pointer to stack for recursive unify (32 bits long)
 */

.org    0x1e000
.long   0xffffffff
.long   nil_const_type
.long   0
.long   0
.long   0
.long   0
.long   0
.long   0
.long   0
.long   0
.long   0
```

```
#include "defs.h"

/********************** fail *************************/

fail:
        get_stack_base(T1)
        cmp_br_delayed  le, B, T1, doabort

unbind_loop:
        cmp_br_delayed  le, TR, OLD_TR, trail_empty
        Nop
        sub             TR, TR, $8
        ld_40           T1, TR, 0
        rd_tag          T2, T1
        and             T2, T2, $cdr_type
        cmp_br_delayed  neq, T2, $cdr_type, unbind1
        add             T3, r0, $var_type
        jump            unbind2$w
        add             T3, r0, $unbound_var_type

unbind1:
        ld_40           T2, T1, 0
        rd_tag          T2, T2
        and             T2, T2, $cdr_type
        cmp_br_delayed  neq, T2, $cdr_type, unbind2
        Nop
        add             T3, r0, $unbound_var_type

unbind2:
        add             T4, T1, 0
        wr_tag          T4, T3
        st_40           T4, T1, 0
        jump            unbind_loop$w
        Nop

trail_empty:
        add             E, OLD_E, 0
        add             CP, OLD_CP, 0
        add             H, OLD_H, 0
        rd_special      T1, cpu_pc
        return          T1, $12
        Nop
        add             A1, SAVE_AX1, 0
        add             A2, SAVE_AX2, 0
        add             A3, SAVE_AX3, 0
        add             A4, SAVE_AX4, 0
        add             A5, SAVE_AX5, 0
        add             A6, SAVE_AX6, 0
        add             A7, SAVE_AX7, 0
```

```
        add             A8, SAVE_AX8, 0
        write_Nreg(SAVE_N)
        st_32           r26, CONST_PTR, $save_r10_offset
        st_32           r9, CONST_PTR, $save_r9_offset
        call            @1f$w
        Nop
@1:     ld_32           r10, CONST_PTR, $save_r10_offset
        ld_32           r9, CONST_PTR, $save_r9_offset
        add             T1, OLD_BP, 0
        jump_reg        T1, $0
        Nop

doabort:
        jump            abort$w
        Nop

#define return_val(value) \
        jump_reg        T3, $4; \
        add             T4, r0, $value

#define push(reg) \
        st_40           reg, T5, 0; \
        add             T5, T5, $8

#define pop(reg) \
        sub             T5, T5, $8; \
        ld_40           reg, T5, 0

/****************************** unify *********************/

/*
 * T1           First argument
 * T2           Second argument
 * T3           Return address
 * T4           Return value of unify and temporary until return
 * T5           Stack pointer for recursive unifys
 * T6, T7       Temporaries
 * T8, T9       Cannot use here (needed by callers for temporaries that
 *              exist across calls.)
 */

unify:
        ld_32           T5, CONST_PTR, $stack_offset

unify_rest:
        rd_tag          T6, T1
        and             T6, T6, $type_mask
        rd_tag          T7, T2
        and             T7, T7, $type_mask
```

```
        cmp_br_delayed  eq, T6, $var_type, dobind
        Nop
        cmp_br_delayed  eq, T7, $var_type, dobind
        Nop
        cmp_br_delayed  neq, T6, $const_type, not_const
        Nop
        rd_tag          T6, T1
        or              T6, T6, $cdr_type
        rd_tag          T7, T2
        or              T7, T7, $cdr_type
        cmp_br_delayed  neq, T6, T7, failed
        Nop
        cmp_br_delayed  neq, T1, T2, failed
        Nop
        return_val(1)

failed: return_val(0)

not_const:
        cmp_br_delayed  neq, T6, T7, failed
        Nop
        push(T1)
        push(T2)
        push(T3)
        ld_40           T1, T1, 0
        ld_40           T2, T2, 0
        jump            unify_rest$w
        rd_special      T3, cpu_pc
        pop(T3)
        pop(T2)
        pop(T1)
        cmp_br_delayed  eq, T4, $1, cont1
        Nop
        return_val(0)

cont1:  add             T4, T1, $8
        ld_40           T1, T4, 0
        rd_tag          T6, T1
        and             T6, T6, $cdr_type
        cmp_br_delayed  eq, T6, $cdr_type, cont2
        Nop
        add             T1, T4, 0
        wr_tag          T1, $list_type

cont2:  add             T4, T2, $8
        ld_40           T2, T4, 0
        rd_tag          T6, T2
        and             T6, T6, $cdr_type
        cmp_br_delayed  eq, T6, $cdr_type, cont3
```

```
        Nop
        add                T2, T4, 0
        wr_tag             T2, $list_type
cont3:  push(T3)
        jump               unify_rest$w
        rd_special         T3, cpu_pc
        pop(T3)
        cmp_br_delayed     eq, T4, $1, cont4
        return_val(0)
cont4:  return_val(1)

dobind: cmp_br_delayed     neq, T6, $var_type, one_var
        Nop
        cmp_br_delayed     neq, T7, $var_type, one_var
        Nop
        cmp_br_delayed     ge, T1, T2, bind1
        bind(T1, T2, T4)
        return_val(1)
bind1:  bind(T2, T1, T4)
        return_val(1)
one_var:cmp_br_delayed     neq, T6, $var_type, bind2
        bind(T2, T1, T4)
        return_val(1)
bind2:  bind(T1, T2, T4)
        return_val(1)


/****************************** esc_unify *********************/

/*
 * T1              First argument
 * T2              Second argument
 * T3              Return address
 * T4              Return value of unify and temporary until return
 * T5              Stack pointer for recursive unifys (PDL)
 * T6, T7          Temporaries
 * T8, T9          Cannot use here (needed by callers for temporaries that
 *                 exist across calls.)
 * CP              Used as the PDL here.  It is saved first however.
 */

#define PDL     CP
#define first_call      0x0
#define not_first_call  0x1

#define esc_return_val(value) \
        tag_cmp_br_delayed      eq_tag, PDL, $not_first_call, @100f; \
        Nop; \
        pop(CP); \
```

```
@100:   jump_reg          T3, $4; \
        add               T4, r0, $value

esc_unify:
        ld_32             T5, CONST_PTR, $stack_offset
        push(CP)
        ld_32             PDL, CONST_PTR, $PDL_offset
        wr_tag            PDL, $first_call

esc_unify_rest:
        rd_tag            T6, T1
        and               T6, T6, $type_mask
        rd_tag            T7, T2
        and               T7, T7, $type_mask
        cmp_br_delayed    eq, T6, $var_type, esc_dobind
        Nop
        cmp_br_delayed    eq, T7, $var_type, esc_dobind
        Nop
        cmp_br_delayed    neq, T6, $const_type, esc_not_const
        Nop
        cmp_br_delayed    ne_40, T1, T2, esc_failed
        Nop
        esc_return_val(1)

esc_failed:
        esc_return_val(0)

esc_not_const:
        cmp_br_delayed    neq, T6, T7, esc_failed
        Nop
        push(T1)
        push(T2)
        push(T3)
        push(PDL)
        wr_tag            PDL, $not_first_call
        ld_40             T1, T1, 0
        ld_40             T2, T2, 0
        jump              esc_unify_rest$w
        rd_special        T3, cpu_pc
        pop(PDL)
        pop(T3)
        pop(T2)
        pop(T1)
        cmp_br_delayed    eq, T4, $1, esc_cont1
        Nop
        esc_return_val(0)

esc_cont1:
        add               T4, T1, $1
```

```
        ld_40               T1, T4, 0
        rd_tag              T6, T1
        and                 T6, T6, $cdr_type
        cmp_br_delayed      eq, T6, $cdr_type, esc_cont2
        Nop
        add                 T1, T4, 0
        wr_tag              T1, $list_type

esc_cont2:
        add                 T4, T2, $1
        ld_40               T2, T4, 0
        rd_tag              T6, T2
        and                 T6, T6, $cdr_type
        cmp_br_delayed      eq, T6, $cdr_type, esc_cont3
        Nop
        add                 T2, T4, 0
        wr_tag              T2, $list_type
esc_cont3:
        push(T3)
        push(PDL)
        wr_tag              PDL, $not_first_call
        jump                esc_unify_rest$w
        rd_special          T3, cpu_pc
        pop(PDL)
        pop(T3)
        cmp_br_delayed      eq, T4, $1, esc_cont4
        esc_return_val(0)
esc_cont4:
        esc_return_val(1)

esc_dobind:
        cmp_br_delayed      neq, T6, $var_type, esc_one_var
        Nop
        cmp_br_delayed      neq, T7, $var_type, esc_one_var
        Nop
        cmp_br_delayed      ge, T1, T2, esc_bind1
        Nop
        jump                do_esc_bind$w
        Nop
esc_bind1:
        add                 T4, T1, 0
        add                 T1, T2, 0
        jump                do_esc_bind$w
        add                 T2, T4, 0
esc_one_var:
        cmp_br_delayed      neq, T6, $var_type, do_esc_bind
        add                 T4, T1, 0
        add                 T1, T2, 0
        add                 T2, T4, 0
```

```
#define binding T1
#define bound   T2

do_esc_bind:
          rd_tag           T4, binding
          and              T4, T4, $type_mask
          cmp_br_delayed   eq, T4, $var_type, @1f
          Nop
          cmp_br_delayed   neq, T4, $const_type, else2
          Nop
@1:
          rd_tag           T4, bound
          and              T4, T4, $cdr_type
          cmp_br_delayed   neq, T4, $cdr_type, else1
          rd_tag           T4, binding
          or               T4, T4, $cdr_type
          wr_tag           binding, T4
else1:
          jump             endif1$w
          st_40            binding, bound, 0
else2:
if3:
          rd_tag           T4, bound
          and              T4, T4, $cdr_type
          cmp_br_delayed   neq, T4, $cdr_type, else3
          add              T4, H, 0
          wr_tag           T4, $cdr_type
          jump             endif3$w
          st_40            T4, bound, 0
else3:
          rd_tag           T6, binding
          and              T6, T6, $type_mask
          wr_tag           T4, T6
          st_40            T4, bound, 0
endif3:
for1:
          add              S, binding, 0
          wr_tag           S, 0
for2:
          ld_40            T4, CONST_PTR, $nil_offset
          cmp_br_delayed   eq_40, S, T4, endfor2
if4:
          ld_40            T4, S, 0
          rd_tag           T6, T4
          and              T6, T6, $type_mask
          cmp_br_delayed   neq, T6, $var_type, else4
          Nop
          add              T6, H, 0
```

```
        wr_tag           T6, $var_type
        st_40            T6, H, 0
        st_40            T6, T4, 0
        jump             endif4$w
        add              H, H, $8
else4:
if5:
        cmp_br_delayed   neq, T6, $const_type, else5
        Nop
        st_40            T4, H, 0
        jump             endif5$w
        add              H, H, $8
else5:
        st_40            S, PDL, $-8
        st_40            H, PDL, $-16
        sub              PDL, PDL, $16
        add              T7, r0, $-1
        wr_tag           T7, T6
        st_40            T7, H, 0
        add              H, H, $8
endif5:
endif4:
        add              S, S, $8
        ld_40            T4, S, 0
if6:
        rd_tag           T6, T4
        and              T4, T4, $cdr_type
        cmp_br_delayed   neq, T4, $cdr_type, endif6
if7:
        ld_40            T6, CONST_PTR, $nil_offset
        cmp_br_delayed   eq, T4, T6, @1f
        Nop
        cmp_br_delayed   neq, T4, S, else7
        Nop
@1:
        add              S, T6, 0
if8:
        cmp_br_delayed   neq, T4, T6, else8
        Nop
        st_40            T4, H, 0
        jump             endif8$w
        add              H, H, $8
else8:
        add              T7, H, 0
        wr_tag           T7, $cdr_type
        st_40            T7, H, 0
        add              H, H, $8
endif8:
        jump             endif7$w
```

```
        Nop
else7:
        add                 S, T4, 0
        wr_tag              S, 0
endif7:
endif6:
        jump                for2$w
        Nop
endfor2:
        ld_32               T6, CONST_PTR, $PDL_offset
        cmp_br_delayed      ge, T6, PDL, endfor1
        ld_40               T4, PDL, 0
        ld_40               binding, PDL, $8
        add                 PDL, PDL, $16
        ld_40               binding, binding, 0
        ld_40               T6, T4, 0
        rd_tag              T6, T6
        and                 T6, T6, $type_mask
        add                 T7, H, 0
        wr_tag              T7, T6
        jump                for1$w
        st_40               T7, T4, 0
endfor1:
endif1:
        esc_return_val(1)

#undef  bound
#undef  binding

/******************** abort ************************/

abort:  add      r28, r0, 0
        jump     0
        Nop

/******************** success *************************/

success:add      r28, r0, $1
        jump     0
        Nop

/********************** unify_const_func **********************/

/*
 * T8   constant to unify
 * T9   return address
 */

unify_const_func:
```

```
        tag_cmp_br_delayed      eq_tag, S, $write_mode, @4f
        Nop
        ld_40           T1, S, 0
        decdr(T1)
        rd_tag          T3, T1
        and             T4, T3, $cdr_type
        cmp_br_delayed  neq, T4, $cdr_type, @2f
        Nop
        tag_cmp_br_delayed      ne_tag, T1, $unbound_var_type, @3f
@1:     add             T2, H, 0
        wr_tag          T2, $list_cdr_type
        call_unify()
        jump            @4f$w
        wr_tag          S, $write_mode
@2:     deref(T1, T3)
        add             T2, T8, 0
        call_unify()
        cmp_br_delayed  eq, T4, $1, @5f
        Nop
@3:     jump            fail$w
        Nop
@4:     add             T1, T8, 0
        st_40           T1, H, 0
        add             H, H, $8
@5:     jump_reg        T9, $4
        Nop

/******************************* unify_value **************************/

/*
 * T8   value to unify
 * T9   return address
 */

unify_value:
        deref(T8, T1)
        tag_cmp_br_delayed      eq_tag, S, $write_mode, @4f
        Nop
        ld_40           T2, S, 0
        decdr(T2)
        rd_tag          T3, T2
        and             T4, T3, $cdr_type
        cmp_br_delayed  neq, T4, $cdr_type, @2f
        Nop
        tag_cmp_br_delayed      ne_tag, T1, $unbound_var_type, @3f
@1:     add             T1, T2, 0
        add             T2, H, 0
        wr_tag          T2, $list_cdr_type
        call_unify()
```

```
        jump            @4f$w
        wr_tag          S, $write_mode
@2:     deref(T2, T3)
        add             T1, T8, 0
        call_unify()
        cmp_br_delayed  eq, T4, $1, @5f
        Nop
@3:     jump            fail$w
        Nop
@4:     add             T1, T8, 0
        add             T2, H, 0
        wr_tag          T2, $var_type
        st_40           T2, H, 0
        add             H, H, $8
        call_unify()
@5:     jump_reg        T9, $4
        Nop

/****************************** is_2 *********************/

/*
 * T9           Return address
 */

#define Temp    T3
#define var     T2
#define var_tag Temp
#define val     T1
#define val_tag T4
#define op      T6
#define n1      T7
#define n2      T8
#define ch      T6

is_2:
        ld_32           T5, CONST_PTR, $stack_offset

is_2_rest:
        push(A1)
        push(A2)
        add             var, A1, 0
        deref(var, var_tag)
        add             val, A2, 0
        deref(val, val_tag)
        and             var_tag, var_tag, $type_mask
        cmp_br_delayed  eq, var_tag, $var_type, @1f
        Nop
        cmp_br_delayed  eq, var_tag, $const_type, @1f
        Nop
```

```
        call_fail()
@1:     and               val_tag, val_tag, $type_mask
        cmp_br_delayed    eq, val_tag, $const_type, @9f
        Nop
        cmp_br_delayed    eq, val_tag, $struct_type, @2f
        Nop
        call_fail()
@2:
        add               S, val, 0
        ld_40             ch, S, 0
        add               S, S, $8
        ld_40             n1, S, 0
        ld_40             n2, S, $8
        add               S, S, $16

        tag_cmp_br_delayed        ne_tag, n1, $struct_type, @4f
        Nop
        add               A1, H, 0
        wr_tag            A1, $var_type
        st_40             A1, H, 0
        add               A2, n1, 0
        push(ch)
        push(n2)
        push(var)
        push(T9)
        jump              is_2_rest$w
        rd_special        T9, cpu_pc
        pop(T9)
        pop(var)
        pop(n2)
        pop(ch)
        add               n1, A1, 0
        deref(n1, Temp)
@4:
        tag_cmp_br_delayed        ne_tag, n2, $struct_type, @4f
        Nop
        add               A1, H, 0
        wr_tag            A1, $var_type
        st_40             A1, H, 0
        add               A2, n2, 0
        push(ch)
        push(n1)
        push(var)
        push(T9)
        jump              is_2_rest$w
        rd_special        T9, cpu_pc
        pop(T9)
        pop(var)
        pop(n1)
```

```
        pop(ch)
        add             n2, A1, 0
        deref(n2, Temp)
@4:
        and             Temp, Temp, $const_type
        cmp_br_delayed  eq, Temp, $const_type, @5f

        rd_tag          Temp, n1
        and             Temp, Temp, $const_type
        cmp_br_delayed  eq, Temp, $const_type, @5f
        Nop
        call_fail()
@5:
        push(A2)                     /* arg 1 */
        push(A3)                     /* operator */
        push(A4)                     /* arg 2 */
        add             A2, n1 , 0
        add             A3, ch , 0
        add             A4, n2 , 0
        escape(ARITH, T1)
        pop(A4)
        pop(A3)
        pop(A2)
        wr_tag          T1, $const_num_type
        jump            unify_rest$w
        rd_special      T3, cpu_pc
        pop(A2)
        pop(A1)
@9:
        jump_reg        T9, $4
        Nop
```

Appendix 3: Macro-Expansion of PLM Instructions to SPUR/Coprocessor

The Prolog coprocessor instructions are broken up into six groups:

- Data Transfer: LD, ST, TO, FROM, MOVE
- State Saving and Modifying:
  PUSH_CHOICEPT, POP_CHOICEPT, PUSH_ENV, POP_ENV, SET_MODE
- Compare and Branch:
  TAG_CMP_BR_DELAYED, CMP_BR_DELAYED
- Unify: UNIFY_X_BR_DELAYED, UNIFY_Y_BR_DELAYED
- Heap and Trail: MAKE_VAR, PUSH_ONTO_HEAP, UNDO_TRAIL
- Special: HASH

The macro-expansion of PLM instructions into a combination of SPUR and Prolog coprocessor instructions is given below. Note that not all PLM instructions use the coprocessor, many instructions can be implemented directly in SPUR code. The PLM instructions are in boldface and their corresponding SPUR code is immediately below. Although this code is by no means debugged or complete, we feel that it provides enough data to give a reasonable estimate of expected performance and code size. These instruction sequences were used to generate the data in Tables 15 and 16.

```
switch_on_term          Lc,Ll,Ls
        TAG_CMP_BR_DELAYED  const,Xi,-,Lc
        TAG_CMP_BR_DELAYED  list,Xi,-,Ll
        TAG_CMP_BR_DELAYED  struct,Xi,-,Ls
        NOP
```

```
switch_on_constant      N,T
        LD                      GRj,address(T)
        NOP
        HASH                    Xi,GRj,GRk,N
        CMP_BR_DELAYED  failedHash,-,-,fail
        FROM            Ri,GRk
        NOP
        JUMP_REG                Ri
        NOP
```

```
switch_on_structure             N,T
        LD                      GRj,address(T)
        NOP
        HASH                    Xi,GRj,GRk,N
        CMP_BR_DELAYED  failedHash,-,-,fail
        FROM            Ri,GRk
        NOP
        JUMP_REG                Ri
        NOP
```

```
try                             L
        RD_SPECIAL              Ri,PC
        TO                                  P,Ri
        NOP
        PUSH_CHOICEPT
        JUMP                                L
        NOP

retry                           L
        RD_SPECIAL              Ri,PC,0
        FROM                    Rj,B
        NOP
        ST                                  Ri,Rj-Poffset
        JUMP                                L
        SET_MODE                            cut, 1

trust                           L
        POP_CHOICEPT                        trust
        JUMP                                L
        NOP

try_me_else                     L
        LD                                  P,address(L)
        NOP
        PUSH_CHOICEPT

retry_me_else                               L
        LD                                  Ri,address(L)
        FROM                    Rj,B
        NOP
        ST                                  Ri,Rj-Poffset
        SET_MODE                            cut,1

trust_me_else                               fail
        POP_CHOICEPT                        trust

fail
        UNDO_TRAIL
        POP_CHOICEPT                        fail
        FROM                    Ri,P
        NOP
        JUMP_REG                            Ri
        NOP

cut
        POP_CHOICEPT                        cut

cutd                            L
```

```
        LD                          GRi,address(L)
        NOP
        POP_CHOICEPT                cutd,GRi

proceed
        FROM                CP, Ri
        MOVE                CP, P
        JUMP_REG                    Ri
        SET_MODE                    cut,0

execute                     Proc
        JUMP                        Proc
        SET_MODE                    cut,0

call                        n, Proc
        RD_SPECIAL          Ri,PC,0
        TO                          CP,Ri
        JUMP                        Proc
        SET_MODE                    cut,0

allocate
        PUSH_ENV                    N

deallocate
        POP_ENV                     N

get_nil                             Ai
        UNIFY_X_BR_DELAYED          const | get,Xi,NIL,fail
        SET_MODE                    unify,read

get_constant                c,Ai
        LD                          GRi,address(c)
        NOP
        UNIFY_X_BR_DELAYED          const | get,Xi,GRi,fail
        SET_MODE                    unify,read

get_variable                [AX | Y]n,Ai
        MOVE                XX,Xi,Xn
        or
        MOVE                XY,Xi,Yn

get_list                    Ai
        UNIFY_X_BR_DELAYED          list | get,Xi,S,fail
        NOP

get_structure                       F,Ai
        UNIFY_X_BR_DELAYED          struct | get,Xi,S,fail
        LD                          GRi,address(F)
```

```
        NOP
        UNIFY_X_BR_DELAYED      const | unify,GRi,S,fail
        NOP

get_value               [AX | Y]n,Ai
        UNIFY_X_BR_DELAYED      val | get,Xn,Xi,fail
        CMP_BR_DELAYED  moreToUnify,-,-,-1
        SET_MODE                unify,read
        MOVE            XX,Xi,Xn
        or
        UNIFY_Y_BR_DELAYED      val | get,Xi,Yn,fail
        CMP_BR_DELAYED  moreToUnify,-,-,-1
        SET_MODE                unify,read

put_nil                         Ai
        MOVE            XX,NIL,Xi

put_constant            c,Ai
        LD                      Xi,address(c)

put_variable            [AX | Y]n,Ai
        MAKE_VAR                var | heap,-,Xi
        MOVE            XX,Xi,Xn
        or
        MAKE_VAR                var | env,Yn,Xi

put_list                Ai
        MAKE_VAR                list | heap,-,Xi
        SET_MODE                unify,write

put_structure                   F,Ai
        LD                      GRi,address(F)
        MAKE_VAR                struct | heap,-,Xi
        SET_MODE                unify,write
        PUSH_ONTO_HEAP  GRi

put_value               [AX | Y]n,Ai
        MOVE            XX,Xn,Xi
        or
        MOVE            YX,Yn,Xi

put_unsafe_value        Yn,Ai
        MAKE_VAR                safe,Yn,Xi

unify_void              n
        use unify_variable n times

unify_value             [AX | Y]n
```

```
    UNIFY_X_BR_DELAYED    val|unify|incrS,Xn,S,fail
    CMP_BR_DELAYED moreToUnify,-,-,-1
    NOP
    or
    UNIFY_Y_BR_DELAYED    val|unify|incrS,Yn,S,fail
    CMP_BR_DELAYED moreToUnify,-,-,-1
    NOP
```

unify_variable [AX|Y]n

```
    UNIFY_X_BR_DELAYED    var|unify|incrS,NIL,S,fail
    MOVE          XX,U1,Xn
    or
    UNIFY_X_BR_DELAYED    var|unify|incrS,NIL,S,fail
    MOVE          XY,U1,Yn
```

unify_constant c

```
    LD                    GRi,address(c)
    NOP
    UNIFY_X_BR_DELAYED    const|unify|incrS,GRi,S,fail
    NOP
```

unify_cdr [AX|Y]n

```
    UNIFY_X_BR_DELAYED    cdr|unify,NIL,S,fail
    MOVE          XX,U1,Xn
    or
    UNIFY_X_BR_DELAYED    cdr|unify,NIL,S,fail
    MOVE          XY,U1,Yn
```

unify_nil

```
    UNIFY_X_BR_DELAYED    const|unify,NIL,S,fail
    NOP
```

Appendix 4: Microcode for a SPUR Prolog Coprocessor

This appendix provides an outline for the microcode requirements of each coprocessor instruction. The instruction name is in boldface along with a description of the fields in the instruction and their size. Immediately below each instruction is a description of what operations must be performed in each of the SPUR pipeline stages. The instruction fetch cycle is not represented as it is identical for all the instructions. There is a fifth pipeline stage added, the extended processing stage, for providing the coprocessor with the extra execution time it may require. The number next to the heading for the extended processing stage signifies the number of extra cycles required. This special stage occurs between the second and third stages of the SPUR pipeline.

TAG_CMP_BR_DELAYED **mask(5),reg(5),tag(5),offset(9)**
R: read reg, mask tag and cmp if no need to deref
E:2 deref (mem access; test for more deref and update reg), mask tag and cmp
M: -
W: write dereferenced value back into reg

CMP_BR_DELAYED **mask(5),reg(5),tag(5),offset(9)**
R: read reg or mask tag and cmp
M: -
W: -

LD **reg(5),reg(5),reg(5),offset(9)**
R: read regs and calculate source address
M: mem access
W: write reg

HASH **reg(5),reg(5),reg(5),immediate(9)**
R: read regs
E:3+2i deref; read starting addr; linear search through immediate number of entries
M: read address to jump to
W: write address into reg

TO **reg(5),reg(5)**
R: read reg
M: -
W: write reg

FROM **reg(5),reg(5)**
R: read reg
M: -
W: write reg

ST **reg(5),reg(5),immediate(14)**

R: read regs and calculate destination address
M: mem access
W: -

PUSH_CHOICEPT
R: read B reg
E:16 store choice point (15 regs); update HB
M: -
W: write new B

POP_CHOICEPT **type(2)**
trust
R: read B reg, calculate address of H
E:2 read new HB reg, calculate address of B; write HB, read new B reg
M: -
W: write B
cut/cutd
R: read B reg, read E ref, calculate address of H
E:3 read new HB, calculate address of B; write HB, read new B; calculate address of H if loop
M: -
W: write last B
fail
R: read B
E:12 read new regs (12 regs) and update 11
M: -
W: write new B

SET_MODE **bit(1),immediate(1)**
R: execute set or reset for mode or cut bit
M: -
W: -

PUSH_ENV **immediate(9)**
R: read E
E:3 store environment (3 regs)
M: store 4th reg of environment
W: write new E

POP_ENV **immediate(9)**
R: read E reg
E:3 read environment regs (3 regs)
M: -
W: write new E

MOVE **X/Y(1),X/Y(1),reg(5),reg(5)**
XX

R: read reg
M: -
W: write reg
XY
R: read reg, read E
M: write to Y
W: -
YX
R: read E
M: read Y
W: write reg

PUSH_ONTO_HEAP **reg(5)**
R: read reg and H
M: write to heap
W: update H

MAKEVAR **type(3),reg(5),reg(5)**
var | heap
R: read H
M: write var
W: update H
var | env
R: read E
M: write var
W: write into reg
lst | heap,str | heap
R: read H
M: -
W: write into reg
safe
R: read E
E:5 read Y, read H; deref; write; update H
M: -
W: write reg

UNDO_TRAIL
R: read B and TR
E:31 read last TR; read first trail entry, decrement TR;
write to unbind, loop to read next trail entry if more
M: -
W: write new TR

UNIFY_X_BR_DELAYED **type(3),get/unify(1),incrS(1),reg(5),reg(5),offset(9)**
Write:
const/get
R: read regs
E:4 deref; unify (cmp; write binding; write to trail; update TR)

M: last write of unify
W: update TR
lst/get,str/get
R: read regs
E:4 deref, read H; unify
M: last write of unify
W: update TR
val/get/~moreToUnify
R: read regs
E:16 deref; deref; push onto PDL if lst or str, follow pointer; follow other pointer; unify; update U1 and U2 (incr pointer and decdr or from PDL if end of lst or str)
M: -
W: -
val/get/moretounify
R: read U1 and U2
E:16 push onto PDL if lst or str, follow pointer; follow other pointer; deref; deref; unify; update U1 and U2 (incr pointer and decdr or from PDL if end of lst or str)
M: -
W: -

Read Mode:

const/unify
R: read S
E:3 get item pointed to by S; unify
M: last write of unify
W: update TR
const/unify/incrS
R: read S
E:9 next pointed to by S; decdr, read H; unify {; write to heap; update H}
M: -
W: update S
cdr/unify
R: read S
M: get item pointed to by S
W: write to dest reg
var/unify/incrS
R: read S
E:9 next pointed to by S; decdr, read H; unify {; write to heap; update H}
M: -
W: update S
val/unify/incrS/~moreToUnify
R: read S and reg
E:17 next pointed to by S; deref; deref; push onto PDL if lst or str,

follow pointer; follow other pointer, read H; unify; {write to heap; update H;} update U1 and U2 (incr pointer and decdr or from PDL if end of lst or str)

M: -

W: update S

val/unify/incrS/moreToUnify

R: read U1 and U2

E:16 push onto PDL if lst or str, follow pointer; follow other pointer; deref; deref; unify;
update U1 and U2 (incr pointer and decdr or from PDL if end of lst or str)

M: -

W: -

Write Mode:

const/unify,const/unify/incrS

R: read reg

M: write to heap

W: update H

cdr/unify,var/unify/incrS

R: read H

E:1 write to reg

M: write to heap

W: update H

val/unify/incrS

R: read H and reg

E:5 write to heap, unify

M: -

W: update H

UNIFY_Y_BR_DELAYED **type(3),get/unify(1),incrS(1),reg(5),reg(5),offset(9)**

same as UNIFY_X_BR_DELAYED with one extra cycle for access to memory and read E reg (an extra cycle if it can't be done in parallel).