A Detailed Description of the VLSI-PLM Instruction Set: A WAM Based Processor for Prolog

Bruce K. Holmer
holmer@ernie.berkeley.edu

Aquarius Group
Computer Science Division
University of California, Berkeley

ABSTRACT

This document describes the VLSI-PLM instruction set and includes small programs to test details of its implementation. The VLSI-PLM is a single chip implementation of the PLM, a WAM based instruction set for the execution of Prolog. The instruction set is described using C-like code based on the actual microcode of the VLSI-PLM. The test programs are a collection of simple Prolog programs which were used to debug the microcode. This report complements the report of Fagin and Dobry, *The Berkeley PLM Instruction Set: An Instruction Set for Prolog.*

This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N000014-88-K-0579. Support was also received from NCR Corporation. Equipment for the project was provided by DEC, NCR, and Apollo.

A Detailed Description of the VLSI-PLM Instruction Set

Bruce K. Holmer
Aquarius Group
Computer Science Division
University of California, Berkeley

1. Introduction

This document describes the VLSI-PLM instruction set and includes small programs to test details of its implementation.

Given below is a set of C code that describes the actions of each VLSI-PLM instruction. It is essentially a level one simulator presented instruction by instruction. I assume that the reader has on hand a copy of *The Berkeley PLM Instruction Set: An Instruction Set for Prolog* [FaDo]. Warren's original definition of the WAM [War] and Dobry's thesis [Dob] are also very helpful. Material in these documents are not duplicated here unless needed for completeness or to point out corrections.

The code is not a direct transliteration of the VLSI-PLM microcode. I have rearranged operations if the result is more clear. In the use of C, I have restricted myself to arithmetic and logic operations, assignments, if and switch statements, jumps, and function calls. These roughly correspond to the operations available on most microarchitectures, and therefore the elemental operations are less likely to be hidden by C constructs.

2. Fundamentals

2.1. Registers and Memory Layout

The registers and state bits of the VLSI-PLM are:

`P`	`program counter (30-bit byte address)`
`CP`	`continuation pointer (return address)`
`H`	`heap pointer`
`B`	`choice point pointer`
`E`	`environment pointer`
`TR`	`trail pointer`
`HB`	`heap pointer on backtracking`
`N`	`number of permanent variables`
`A0 through A7`	`eight argument registers`
`CUT`	`cut bit`
`MODE`	`mode bit (READ or WRITE)`
`PDL`	`unification push down list pointer`
`H2`	`global heap pointer`

In [FaDo] the argument registers where specified with either `Ai` or `Xi`. In this document, I will use only `Ai` and number the registers starting from 0 ([FaDo] and the PLM compiler both start the numbering from 1). Similarly, `Yi` addressing starts from zero (rather than one as in the compiler). I have chosen this convention since it matches the actual value used in the argument byte of the machine instruction.

The program counter (P) is not contained in the VLSI-PLM, but instead is kept in the external instruction prefetch unit. The VLSI-PLM can request a new value for P by either sending the entire new 30-bit value (as in proceed) or by sending an 8-bit offset (as in switch_on_term). The current prefetch unit treats offsets as positive only (as shown in the pseudo code), however, with a change to the prefetch unit (and assembler) this could be changed to signed offsets.

Choice point structure:

```
          <---  B          (points at one position beyond end of choice pt)
TR
P
CP
E
A7
A6
A5
A4
A3
A2                  -
A1                  |        higher addresses
A0                  |
N
H
B
```

Structure of environment:

```
Yn
...
Y1
Y0        <---  E
N
CP
CUT | B
E
```

Structure of trail:

```
...
VALUE                          |
NONVAR | ADDR                  | address/value pair from setarg/3
VAR | X                          normal trail entries
VAR | Y
          <--- TR
```

Unification of nested structures requires a push down list. On the VLSI-PLM this is supplied as an on-chip circular buffer (with automatic spill to memory on overflow). This mechanism is not included in the following pseudo-code since it would overly complicate the unify algorithm. Instead, I assume the existence of two arrays, PDLl and PDLr, and an index pointer, PDL. Since a very small fraction of the total execution time is spent in the general unify routine, these arrays could be placed on top of the

environment or heap for the duration of the unify routine with little loss of performance.

In the pseudo code, the comparison `ADDRESS == Yi` is often used. This does not refer to real registers or state bits, but reflects the decoding of the opcode of the current instruction. If the opcode indicates that the addressing mode refers to a register (`Xi`), then this test would be false. Otherwise, if the addressing mode refers to a permanent variable (`Yi`), then the test succeeds.

2.2. Cdr coding

The primary rule for handling cdr bits is: the cdr bit is associated with the memory location, not the data value. Cdr bits (that have meaning) can appear only on the heap and trail, and can only be created by the `unify_nil` and `unify_cdr` instructions. The cdr bit of a stack variable has no meaning, since lists and structures are not built on the stack. Cdr bits can be removed from a given location only when that location is reclaimed on backtracking. Binding and detrailing do not change the value of the cdr bit. During unification the cdr bit does not affect the match between two constants.

The mixture of cdr coding and variable dereferencing gives rise a subtle point. The value (and tag) of a variable is determined by the data at the end of its dereference chain. However, the cdr bit of the variable itself must be used (rather than the cdr bit of the dereferenced value).

The Xenologic X1 microcode takes the position that the cdr bit of data in an A register has no meaning, and therefore is always cleared. This was not done in the VLSI-PLM, since the non-WAM instructions allow the cdr bit to be manipulated by assembly code.

2.3. Instruction Formats

Although Dobry's thesis contains explicit information about instruction formats, a brief summary of instruction opcodes and arguments is given here. All opcodes are a single byte. Instructions have up to three arguments, the first is either one or four bytes and the next two are always one byte.

Instruction	Opcode (hex)
allocate	00
deallocate	01
proceed	80
cut	05
trust_me_else	0b
fail	82
unify_nil	02
nop	04
halt	06
reset	81

Instructions with no arguments

Instruction	Opcode	Arg1	Arg2
switch_on_constant switch_on_structure	d0 d1	table address	hash mask
try retry trust	45 46 47	destination address	
try_me_else retry_me_else	41 42	continuation address	
cutd	49	continuation address	
execute jump	44 4c	destination address	
call	54	destination address	new N
get_constant put_constant	50 52	constant	A register
get_structure put_structure	51 53	functor	A register
unify_constant	40	constant	

Instructions with four-byte Arg1

Escape	Opcode	Arg1
External (arity 0)	4a	00000100
External (arity 1)	4a	00000101
External (arity 2)	4a	00000102
External (arity 3-7)	4a	00000103
Additional regs dumped		
External (arity 0)	4a	00000180
External (arity 1)	4a	00000181
External (arity 2)	4a	00000182
External (arity 3-7)	4a	00000183
X7 to H2	4a	00000060
H2 to X7	4a	00000061
X7 to H	4a	00000062
H to X7	4a	00000063
X7 to S	4a	00000064
S to X7	4a	00000065
X7 to B	4a	00000066
B to X7	4a	00000067
X7 to E	4a	00000068
E to X7	4a	00000069
X7 to TR	4a	0000006a
TR to X7	4a	0000006b

Escape instructions (Arg1 is four bytes)

Instruction	Opcode	Arg1	Arg2	Arg3
switch_on_term	b4	constant offset	list offset	structure offset
get_nil get_list put_nil put_list	10 11 12 13	A register		
get_variable get_value put_variable put_value	20 21 22 23	A register	A register	
get_variable get_value put_variable put_value	28 29 2a 2b	Y index	A register	
put_unsafe_value	2c	Y index	A register	
unify_variable unify_value unify_cdr	15 17 14	A register		
unify_variable unify_value unify_cdr	1d 1f 1c	Y index		
unify_void	18	count		
deref	1e	A register		
add sub and or eor mult	24 25 26 27 2d 2e	A register source	A register destination	
memread memwrite	34 35	A reg (addr base) A reg (addr base)	A register destination A register source	positive offset positive offset
coderead codewrite	16 19	A reg (address) A reg (address)		
jeq jlt jle	b5 b6 b7	A register	A register	positive branch offset
jumpxn	90	A register		
loadn	1a	new N		

Instructions with one-byte Arg1

3. Details of Pseudo Code

Constants and user visible registers are given names consisting of all capital letters. The constants VAR, CON, LST, and STR represent the tag values for variables, constants, lists, and structures, respectively. Temporary variables are usually specified by lower case, e.g. `t0`, `valt0`, etc. These temporaries do not necessarily correspond to scratch registers in the microarchitecture.

In the VLSI-PLM, logic operations and equality tests are 32-bit. Arithmetic operations and inequality tests operate on only the value field (28 bits). The pseudo code below adopts the usual meaning of these operators in C (working on 32-bit signed integers). Explicit use of `value` along with an operation or

comparison signifies the fact that that operation or comparison is 28-bit (thus the comparisons are unsigned).

There are several basic operations that often occur. I define them here in terms of arithmetic and logic operations, but the microarchitecture directly supports them.

```
tag(t0)
{
        return (t0 >> 30) & 3;                  /* return bits <31:30> */
}

cdr(t0)
{
        return (t0 >> 29) & 1;                  /* return bit <29> */
}



value(t0)
{
        return t0 & 0x0fffffff;                  /* return bits <27:0> */
}

construct(tag, cdr, val)
{                                           /* gc bit is cleared */
        t0 = (tag << 30) & 0xc0000000;
        t1 = (cdr << 29) & 0x20000000;
        t2 = val         & 0x0fffffff;
        return (t0 | t1 | t2);
}
```

In addition, the function `memread(t0)` performs a memory read from the address given by `value(t0)`, and returns the entire 32 bits at this address. The function `memwrite(t0, t1)` performs a memory write to the address given by `value(t0)`, storing the entire 32 bits of `t1` at this address. The functions `coderead` and `codewrite` are similar, but read and write to the code address space (which is separate from the data address space).

4. Basic Operations

4.1. Dereference

This dereference routine assumes that `t0` is a variable. The VLSI-PLM has a conditional micro-subroutine branch based on the tag value, and so this is the natural thing to do. An alternative is to test `t0` in dereference and return immediately if the `t0` is not a variable.

One must beware of the case when the last element of a dereference chain is an unbound variable, and it and the previous link (the variable pointing to the unbound variable) have differing cdr bits. The code given here is conservative and will dereference an extra time in this case.

```
dereference(t0)
{
                        /* assumes that on entry:  tag(t0) == VAR */
L1:
        t1 = memread(t0);
        if (t0 == t1 || tag(t1) != VAR) return t1;
        t0 = t1;
        goto L1;
}
```

4.2. Trail

It is important that the entire 32 bits of the variable be written to the trail stack, since the detrail routine (in fail) restores the value by a 32-bit transfer.

Although the trail grows toward low memory, the trail pointer, `TR`, points at the next available location (rather than the top entry of the stack).

```
trail(t0)
{
                /* assumes that t0 is the unbound variable to be trailed */
        valt0 = value(t0);
        if ((valt0 < value(B) && valt0 > value(H)) ||    /* unbnd var on stack */
             valt0 < value(HB)) {                        /* unbnd var on heap */
                memwrite(TR, t0);
                TR--;
        }
}
```

4.3. Fail

The VLSI-PLM does not have variable sized choice points, so all of the argument registers are saved, even if it is not necessary. This makes garbage collection harder, since one does not know which values are valid starting points for marking. A possible solution is to store the number of valid argument registers in `X7`'s position (`X7` is used by the compiler only as a temporary).

The detrailing done in fail must be done by popping the trail stack. This is important because several `setarg` entries may modify the same memory location, and the oldest trail entry should be restored. The `setarg` trail entry consists of an address/value pair. The address must have a non-variable tag to differentiate it from a normal (unbound variable) entry.

```
fail()
{
        PDL = 0;
        if (value(B) == value(STKbase)) {
                goalfail();
                return;
        }
        newTR = memread(B - 1);
        P = memread(B - 2);
        CP = memread(B - 3);
        E = memread(B - 4);
        A[7] = memread(B - 5);
        A[6] = memread(B - 6);
        A[5] = memread(B - 7);
        A[4] = memread(B - 8);
        A[3] = memread(B - 9);
        A[2] = memread(B - 10);
        A[1] = memread(B - 11);
        A[0] = memread(B - 12);
        N = memread(B - 13);
        H = memread(B - 14);
L1:
        if (value(newTR) == value(TR)) return;
        TR++;
        t0 = memread(TR);
        if (tag(t0) == VAR) {
                memwrite(t0, t0);          /* normal trail entry */
        } else {
                TR++;                   /* 2 word entry for setarg */
                t1 = memread(TR);
                memwrite(t0, t1);
        }
        goto L1;
}
```

5. Indexing Instructions

5.1. Switch on Term

This instruction takes three 8-bit arguments that give the (positive only) branch displacement for the constant, list, and structure cases. The variable case continues execution with the next statement. The maximum value for the displacement (255) indicates failure.

```
switch_on_term()
{
        t0 = A[0];
        if (tag(t0) == VAR) t0 = dereference(t0);
        switch (tag(t0)) {
            case VAR:
                return;
            case CON:
                t1 = arg1;
                break;
            case LST:
                t1 = arg2;
                break;
            case STR:
                t1 = arg3;
                break;
        }
        if (t1 == 0xff) {
                fail();
                return;
        }
        P = P + t1;
}
```

5.2. Switch on Constant and Structure

Both of these instructions take a 28-bit code space word address (arg1) which specifies the beginning location of the hash table, and an 8-bit mask (arg2) which is anded with the constant or functor value as the hash function.

```
switch_on_constant()
{
        t0 = A[0];
        if (tag(t0) == VAR) t0 = dereference(t0);
        switch(t0);
}
```

```
switch_on_structure()
{
        t0 = A[0];
        if (tag(t0) == VAR) t0 = dereference(t0);
        func = memread(t0);                /* functor of structure */
        switch(func);
}
```

These switch instructions utilize open addressing hashing with linear probing. The functor or constant value is multiplied by two so that consecutive integers will map to consecutive table entries. The hash function is given by the functor or constant value anded with the mask (arg2).

The hash table consists of two word entries, a value and a code address. If the value entry matches the value of the constant or functor, then execution proceeds at the code address. The cdr bit of the value entry is used to denote unused table locations and a valid entry just before an unused location. The value of an empty entry must be unique (different from all constants and functors) since probing can begin at an empty entry. Failure occurs when all the table entries have been unsuccessfully checked or if after looping to the top of the table, the cdr bit of the value is set.

```
switch(t0)
{
        t1 = t0 << 1;
        t1 = t1 & arg2;
        start = arg1 + t1;                  /* pointer into hash table */
        ptr = start;
L1:
        t2 = coderead(ptr);
        if (value(t2) == value(t0)) {
                P = coderead(ptr + 1);
                return;
        }
        ptr = ptr + 2;
        if (cdr(t2) == 0) goto L1;

        ptr = arg1;                         /* start over at beginning of table */
L2:
        if (ptr >= start) {
                fail();
                return;
        }
        t2 = coderead(ptr);
        if (value(t2) == value(t0)) {
                P = coderead(ptr + 1);
                return;
        }
        ptr = ptr + 2;
        if (cdr(t2) == 0) goto L2;

        fail();
        return;
}
```

There are several improvements that can be made with this instruction. The anding of arg2 should be done before the left shift, allowing tables with 256 entries. Also, instead of failing when a hash lookup is unsuccessful, execution should proceed to the next instruction, allowing several switch instructions to be placed one after another (this is what is done in the X1). This is useful for hash tables with more than 128 entries. In addition, eliminating the code which probes from the beginning of the table would make modification of the hash table easier during assert and retract. Finally, the cdr bit should be set only for empty entries, eliminating the need for unique values in empty entries. These changes are summarized below with new pseudo code.

```
switch(t0)                                  /* not implemented */
{
        t1 = t0 & arg2;
        t1 = t1 << 1;
        ptr = arg1 + t1;                    /* pointer into hash table */
L1:
        t2 = coderead(ptr);
        if (cdr(t2) == 1) goto L2;          /* empty if cdr set */
        if (value(t2) == value(t0)) {
                P = coderead(ptr + 1);
                return;
        }
        ptr = ptr + 2;
        goto L1;
L2:
        return;                     /* if no match, continue with instruction */
                                    /* following the switch */
}
```

6. Procedure Control Instructions

6.1. Try

```
try()
{
        CUT = 1;
        if (value(E) < value(B)) {
                t0 = B;
        } else {
                t0 = E + N;
        }
        memwrite(t0,      B);
        memwrite(t0 +  1, H);
        memwrite(t0 +  2, N);
        memwrite(t0 +  3, A[0]);
        memwrite(t0 +  4, A[1]);
        memwrite(t0 +  5, A[2]);
        memwrite(t0 +  6, A[3]);
        memwrite(t0 +  7, A[4]);
        memwrite(t0 +  8, A[5]);
        memwrite(t0 +  9, A[6]);
        memwrite(t0 + 10, A[7]);
        memwrite(t0 + 11, E);
        memwrite(t0 + 12, CP);
        memwrite(t0 + 13, P);
        memwrite(t0 + 14, TR);
        B = t0 + 15;
        HB = H;
        P = arg1;
}
```

6.2. Retry

```
retry()
{
        CUT = 1;
        memwrite(B - 2, P);
        P = arg1;
}
```

6.3. Trust

```
trust()
{
        CUT = 0;
        t0 = B;
        B  = memread(t0 - 15);
        HB = memread(t0 - 14);              /* not the optimal value for HB */
        P = arg1;
}
```

The HB register is not updated optimally in the PLM. Whenever choice points are discarded, the HB register is restored first, and then the choice point is thrown away. What should happen is that the choice point should be discarded first, and then the HB register loaded from the newly activated choice point on top of the stack.

The current HB register updating scheme will give correct operation, but will trail more often than necessary. Fixing the level1 simulator reduced the number of trails on the quicksort benchmark by more than half.

If the last choice point on the stack is discarded, and then the PLM attempts an HB register update, then the HB register gets loaded with garbage. This is because the HB register is always updated from the H field of the current choice point; if their are no choice points on the stack then the value obtained in this way is obviously not meaningful.

However, this doesn't affect the operation of the PLM, provided the address generated when the PLM attempts to update the HB from the nonexistent choice point is accepted by the memory system. Consider for a moment why this must be so. Only two incorrect actions are possible as a result of an incorrect HB value: either a value is trailed that should not be, or a value is not trailed that should be. In either case, the undesirable effects will not be felt until failure occurs, and the trail is unwound. However, the trail section built with the garbage HB value was placed on the trail stack when there was no choice point, so when failure occurs the PLM reports top level failure and quits.

This problem didn't arise before because the HB register was updated from the last choice point discarded from the stack, thus avoiding a special check for the top of the stack every time a choice point was discarded to prevent HB from getting a garbage value. (The above comments are originally from Barry Fagin).

The problem with loading HB with garbage can be eliminated by initializing the choice point stack with a "sentinel" choice point. This choice point would contain the initial values of all the stack pointers and the code address of the routine for goal failure. If this scheme were used, the check `B == STKbase` in `fail` could be eliminated.

6.4. Try me Else

```
try_me_else()
{
        CUT = 1;
        if (value(E) < value(B)) {
                t0 = B;
        } else {
                t0 = E + N;
        }
        memwrite(t0,      B);
        memwrite(t0 +  1, H);
        memwrite(t0 +  2, N);
        memwrite(t0 +  3, A[0]);
        memwrite(t0 +  4, A[1]);
        memwrite(t0 +  5, A[2]);
        memwrite(t0 +  6, A[3]);
        memwrite(t0 +  7, A[4]);
        memwrite(t0 +  8, A[5]);
        memwrite(t0 +  9, A[6]);
        memwrite(t0 + 10, A[7]);
        memwrite(t0 + 11, E);
        memwrite(t0 + 12, CP);
        memwrite(t0 + 13, arg1);
        memwrite(t0 + 14, TR);
        B = t0 + 15;
        HB = H;
}
```

6.5. Retry me Else

```
retry_me_else()
{
        CUT = 1;
        memwrite(B - 2, arg1);
}
```

6.6. Trust me Else

```
trust_me_else()
{
        CUT = 0;
        t0 = B;
        B  = memread(t0 - 15);
        HB = memread(t0 - 14);              /* not the optimal value for HB */
}
```

6.7. Cut

```
cut()
{
        t0 = memread(E - 3);
        B = value(t0);                          /* must get rid of cdr (cut) bit */
        if (cdr(t0) == 1) {
                B  = memread(t0 - 15);
                HB = memread(t0 - 14);          /* not the optimal value for HB */
        }
}
```

6.8. Cutd

```
cutd()
{
L1:
        t0 = B;
        B  = memread(t0 - 15);
        t1 = memread(t0 -  2);
        if (t1 != arg1) goto L1;
        HB = memread(t0 - 14);
}
```

7. Clause Control Instructions

7.1. Proceed

```
proceed()
{
        P = CP;
        CUT = 0;
        if (value(CP) == 0) goalsuccess();
}
```

The test for CP being a null value could be eliminated if CP is initialized to point to the code for goal success.

7.2. Execute

```
execute()
{
        P = arg1;
        CUT = 0;
}
```

7.3. Call

```
call()
{
        CP = P;
        P = arg1;
        N = arg2;
        CUT = 0;
}
```

7.4. Allocate

```
allocate()
{
        if (value(E) < value(B)) {
                t0 = B;
        } else {
                t0 = E + N;
        }
        memwrite(t0,     E);
        memwrite(t0 + 1, construct(0, CUT, B));          /* record the CUT bit */
                                        /* as the cdr bit of B in the environment */
        memwrite(t0 + 2, CP);
        memwrite(t0 + 3, N);
        E = t0 + 4;
}
```

7.5. Deallocate

```
deallocate()
{
        N  = memread(E - 1);
        CP = memread(E - 2);
        E  = memread(E - 4);
}
```

8. Get Instructions

8.1. Get Variable

```
get_variable()
{
        if (ADDRESS == Yi) {
                memwrite(E + arg1, A[arg2]);
        } else {
                A[arg1] = A[arg2];
        }
}
```

8.2. Get Value

```
get_value()
{
        if (ADDRESS == Yi) {
                t0 = memread(E + arg1);
        } else {
                t0 = A[arg1];
        }
        t1 = A[arg2];
        if (tag(t0) == VAR) t0 = dereference(t0);
        if (tag(t1) == VAR) t1 = dereference(t1);
        if (ADDRESS != Yi)  A[arg1] = t1;
        unify(t0, t1);
}
```

8.3. Get Constant

```
get_constant()
{
        t0 = A[arg2];
        if (tag(t0) == VAR) t0 = dereference(t0);
        unify(t0, arg1);
}
```

8.4. Get Nil

```
get_nil()
{
        t0 = A[arg1];
        if (tag(t0) == VAR) t0 = dereference(t0);
        unify(t0, NIL);
}
```

8.5. Get Structure

```
get_structure()
{
        t0 = A[arg2];
        if (tag(t0) == VAR) t0 = dereference(t0);
        switch (tag(t0)) {
            case CON:
            case LST:
                fail();
                return;
            case VAR:
                MODE = WRITE;
                memwrite(t0, construct(STR, cdr(t0), H));
                trail(t0);
                memwrite(H, arg1);
                H++
                break;
            case STR:
                MODE = READ;
                t1 = memread(t0);
                if (t1 != arg1) {    /* 32-bit compare works because cdr bit */
                                     /* is always clear on functor */
                        fail();
                        return;
                }
                S = value(t0) + 1;
                break;
        }
}
```

8.6. Get List

```
get_list()
{
        t0 = A[arg1];
        if (tag(t0) == VAR) t0 = dereference(t0);
        switch (tag(t0)) {
            case CON:
            case STR:
                fail();
                return;
            case VAR:
                MODE = WRITE;
                memwrite(t0, construct(LST, cdr(t0), H));
                trail(t0);
                break;
            case LST:
                MODE = READ;
                S = value(t0);
                break;
        }
}
```

9. Put Instructions

9.1. Put Variable

```
put_variable()
{
        if (ADDRESS == Yi) {
                t0 = construct(VAR, 0, E + arg1);
                memwrite(E + arg1, t0);
        } else {
                t0 = construct(VAR, 0, H));
                memwrite(H, t0);
                H++;
                A[arg1] = t0;
        }
        A[arg2] = t0;
}
```

9.2. Put Value

```
put_value()
{
        if (ADDRESS == Yi) {
                A[arg2] = memread(E + arg1);
        } else {
                A[arg2] = A[arg1];
        }
}
```

9.3. Put Unsafe Value

```
put_unsafe_value()
{
        t0 = memread(E + arg1);
        if (tag(t0) == VAR) t0 = dereference(t0);
        if (tag(t0) != VAR || value(t0) < value(E)) {
                A[arg2] = t0;
                return;
        }
        t1 = construct(VAR, 0, H);
        memwrite(H, t1);
        H++;
        A[arg2] = t1;
        memwrite(t0, t1);
        trail(t0);
}
```

9.4. Put Constant

```
put_constant()
{
        A[arg2] = arg1;
}
```

9.5. Put Nil

```
put_nil()
{
        A[arg1] = NIL;
}
```

9.6. Put Structure

```
put_structure()
{
        MODE = WRITE;
        A[arg2] = construct(STR, 0, H);
        memwrite(H, arg1);
        H++;
}
```

9.7. Put List

```
put_list()
{
        MODE = WRITE;
        A[arg1] = construct(LST, 0, H);
}
```

10. Unify Instructions

It is very important that before calling the general unify routine given below, that both arguments are fully dereferenced.

```
unify(t0, t1)
{
                /* t0 and t1 must be dereferenced on entry */
L1:
        if (tag(t0) != VAR && tag(t1) != VAR) {          /* neither is a VAR */
                if (tag(t0) != tag(t1)) {
                        fail();
                        return;
                }
                if (tag(t0) == CON) {
                        if (value(t0) != value(t1)) {
                                fail();
                                return;
                        }
                        if (value(PDL) == 0) return;
                        goto L2;
                } else {
                        t2 = memread(t0);
                        t3 = memread(t1);
                        t0++;
                        t1++;
                        PDL++;
                        PDL1[PDL] = t0;
                        PDLr[PDL] = t1;
                        t0 = t2;
                        t1 = t3;
                        if (tag(t0) == VAR) t0 = dereference(t0);
                        if (tag(t1) == VAR) t1 = dereference(t1);
                        goto L1;
                }
        } else if (tag(t0) == VAR && tag(t1) == VAR) {       /* both are VAR */
                if (value(t0) < value(t1)) {
                        memwrite(t1, construct(tag(t0), cdr(t1), t0));
                        trail(t1);
                } else {
                        memwrite(t0, construct(tag(t1), cdr(t0), t1));
                        trail(t0);
                }
        } else {                                             /* one is a VAR */
                if (tag(t0) == VAR) {
                        memwrite(t0, construct(tag(t1), cdr(t0), t1));
                        trail(t0);
                } else {
                        memwrite(t1, construct(tag(t0), cdr(t1), t0));
                        trail(t1);
                }
        }
        if (value(PDL) == 0) return;
L2:                                                  /* PDL is not empty */
        t0 = PDL1[PDL];
        t1 = PDLr[PDL];
        PDL--;
        t2 = memread(t0);
```

```
        t3 = memread(t1);
        if (cdr(t2) == 1) {
                if (tag(t2) == VAR) t2 = dereference(t2);
                t0 = t2;
        }
        if (cdr(t3) == 1) {
                if (tag(t3) == VAR) t3 = dereference(t3);
                t1 = t3;
        }
        goto L1;
}
```

10.1. Unify Void

```
unify_void()
{
        count = arg1;
        if (MODE == READ) {
L1:
                if (count == 0) return;
                t0 = memread(S);
                if (cdr(t0) == 1) {
                        if (tag(t0) == VAR) t0 = dereference(t0);
                        switch (tag(t0)) {
                            case CON:
                            case STR:
                                fail();
                                return;
                            case VAR:
                                MODE = WRITE;
                                memwrite(t0, construct(LST, cdr(t0), H));
                                trail(t0);
                                goto L2;
                            case LST:
                                S = value(t0);
                        }
                }
                count--;
                S++;
                goto L1;
        } else {
L2:
                if (count == 0) return;
                memwrite(H, construct(VAR, 0, H));
                count--;
                H++;
                goto L2;
        }
}
```

10.2. Unify Value

```
unify_value()
{
        if (ADDRESS == Yi) {
                t0 = memread(E + arg1);
        } else {
                t0 = A[arg1];
        }
        if (tag(t0) == VAR) t0 = dereference(t0);
        if (MODE == READ) {
                t1 = memread(S);
                if (cdr(t1) == 1) {
                        if (tag(t1) == VAR) t1 = dereference(t1);
                        switch (tag(t1)) {
                            case CON:
                            case STR:
                                fail();
                                return;
                            case VAR:
                                MODE = WRITE;
                                memwrite(t1, construct(LST, cdr(t1), H));
                                trail(t1);
                                goto L1;
                            case LST:
                                S = t1;
                                t1 = memread(S);
                        }
                }
                S++;
                if (tag(t1) == VAR) t1 = dereference(t1);
                unify(t0, t1);
                return;
        } else {
L1:
                if (tag(t0) == VAR && value(t0) > value(H)) {
                        t1 = construct(VAR, 0, H);
                        memwrite(H, t1);
                        memwrite(t0, t1);      /* clearing cdr is OK--on stack */
                        trail(t0);
                } else {
                        memwrite(H, construct(tag(t0), 0, t0));
                                               /* make sure cdr bit is clear! */
                }
                H++;
        }
}
```

The PLM compiler produces the instruction, `unify_unsafe_value`. For execution by the VLSI-PLM, this instruction should be replaced with `unify_value` (the assembler could do this).

In write mode, if the argument of `unify_value` dereferences to an unbound variable on the stack, then it would be wrong for this to be simply copied to the top of the heap (since a variable link would then point upwards). Instead, a new unbound variable is created on the heap and is bound to the variable

on the stack, the binding is trailed if necessary. Thus, our `unify_value` is Warren's `unify_local_value`.

There is no version of `unify_value` without the check for an unbound variable on the stack because one can never tell at compile time whether the check can be safely eliminated. The following code illustrates this point.

```
main :- a(X), d(X), e, write(X), nl.
a(Y) :- c(X,Y), b(X).
b(_).
c(X, [X]).          % called from a/1 with X set to unbnd var on stack
d([x]).
e :- b(X), b(X).
```

10.3. Unify Variable

```
unify_variable()
{
        if (MODE == READ) {
                t0 = memread(S);
                if (cdr(t0) == 1) {                    /* S points to cdred data */
                        if (tag(t0) == VAR) t0 = dereference(t0);
                        switch (tag(t0)) {
                            case CON:
                            case STR:
                                fail();
                                return;
                            case VAR:
                                MODE = WRITE;
                                memwrite(t0, construct(LST, cdr(t0), H));
                                trail(t0);
                                goto L1;
                            case LST:
                                S = value(t0);
                                t0 = memread(S);
                        }
                }
                S++;
                if (tag(t0) == VAR) t0 = dereference(t0);
                if (ADDRESS == Yi)
                        memwrite(E + arg1, t0);
                else
                        A[arg1] = t0;

        } else {                        /* MODE == WRITE */
L1:
                t0 = construct(VAR, 0, H);
                memwrite(H, t0);                    /* push unbound var on heap */
                H++;
                if (ADDRESS == Yi)
                        memwrite(E + arg1, t0);
                else
                        A[arg1] = t0;
        }
}
```

10.4. Unify Constant

```
unify_constant()
{
        if (MODE == READ) {
                t0 = memread(S);
                if (cdr(t0) == 1) {
                        if (tag(t0) == VAR) t0 = dereference(t0);
                        switch (tag(t0)) {
                            case CON:
                            case STR:
                                fail();
                                return;
                            case VAR:
                                MODE = WRITE;
                                memwrite(t0, construct(LST, cdr(t0), H));
                                trail(t0);
                                goto L1;
                            case LST:
                                S = value(t0);
                                t0 = memread(S);
                        }
                }
                S++;
                if (tag(t0) == VAR) t0 = dereference(t0);
                unify(t0, arg1);
                return;
        } else {                        /* MODE == WRITE */
L1:
                memwrite(H, arg1);
                H++;
        }
}
```

10.5. Unify Cdr

```
unify_cdr()
{
        if (MODE == READ) {
                t0 = memread(S);
                if (cdr(t0) == 0) t0 = construct(LST, 0, S);
        } else {
                t0 = construct(VAR, 1, H);
                memwrite(H, t0);
                H++;
        }
        if (ADDRESS == Yi) {
                memwrite(E + arg1, t0);
        } else {
                A[arg1] = t0;
        }
}
```

10.6. Unify Nil

```
unify_nil()
{
        if (MODE == READ) {
                t0 = memread(S);
                if (cdr(t0) == 0) {
                        fail()
                        return;
                }
                if (tag(t0) == VAR) t0 = dereference(t0);
                unify(t0, NIL);
                return;
        } else {
                memwrite(H, construct(CON, 1, NIL));
                H++;
        }
}
```

11. VLSI-PLM Specific Instructions

11.1. Deref

```
deref()
{
        t0 = A[arg1];
        if (tag(t0) == VAR) t0 = dereference(t0);
        A[arg1] = t0;
}
```

11.2. Add

```
add()
{
        t0 = A[arg1];
        t1 = A[arg2];
        A[arg2] = construct(CON, 0, t0 + t1);
}
```

11.3. Sub

```
sub()
{
        t0 = A[arg1];
        t1 = A[arg2];
        A[arg2] = construct(CON, 0, t1 - t0);
}
```

11.4. Mult

Multiply does an unsigned multiply of two 27-bit integers to produce a 27-bit result. If the result requires more than 27-bits, then an overflow is indicated by setting the answer to the constant NIL. To count the number of iterations, one is shifted left 27 times.

```
mult()
{
        multiplier = A[arg1];
        multiplicand = A[arg2];
        accum = 0;
        cnt = 1;
L1:
        accum = accum << 1;
        if (accum & 0x08000000 == 1) goto ovrflw;
        multiplier = multiplier << 1;
        if (multiplier & 0x08000000 == 1) {
                accum = accum + multiplicand;
                if (accum & 0x08000000 == 1) goto ovrflw;
        }
        cnt = cnt << 1;
        if (cnt & 0x08000000 == 0) goto L1;

        A[arg2] = construct(CON, 0, accum);
        return;
ovrflw:
        A[arg2] = NIL;
        return;
}
```

11.5. And

```
and()
{
        t0 = A[arg1];
        t1 = A[arg2];
        A[arg2] = t0 & t1;                /* 32-bit operation !! */
}
```

11.6. Or

```
or()
{
        t0 = A[arg1];
        t1 = A[arg2];
        A[arg2] = t0 | t1;                /* 32-bit operation !! */
}
```

11.7. Eor

```
eor()
{
        t0 = A[arg1];
        t1 = A[arg2];
        A[arg2] = t0 ^ t1;                /* 32-bit operation !! */
}
```

11.8. Memread

```
memread()
{
        A[arg2] = memread(A[arg1] + arg3);
}
```

11.9. Memwrite

```
memwrite()
{
        memwrite(A[arg1] + arg3, A[arg2]);
}
```

11.10. Coderead

```
coderead()
{
        A[7] = coderead(A[arg1]);
}
```

11.11. Codewrite

```
codewrite()
{
        codewrite(A[arg1], A[7]);
}
```

11.12. Jump

Same as execute, except that the CUT bit is not affected.

```
jump()
{
        P = arg1;
}
```

11.13. Jlt

```
jlt()
{
        if (value(A[arg1]) < value(A[arg2])) P = P + arg3;
}
```

11.14. Jeq

```
jeq()
{
        if (A[arg1] == A[arg2]) P = P + arg3;          /* 32-bit equality!! */
}
```

11.15. Jle

```
jle()
{
        if (value(A[arg1]) <= value(A[arg2])) P = P + arg3;
}
```

11.16. JumpXn

```
jumpxn()
{
        P = A[arg1];
}
```

11.17. LoadN

```
loadn()
{
        N = arg1;
}
```

12. Test Programs

The following test programs at one time were not correctly executed due to microcode bugs. They all now work correctly on the VLSI-PLM (however, some versions of the PLM level1, PLM level2, and PPP level1 simulators have not been corrected). The programs are given here as test cases for new simulators and PLM implementations.

12.1. Dereferencing

Check dereferencing of arguments in general unify:

```
main :- a(X), a(Y), b(X,Y), c(X), d(X,Y), write(Y), nl.

a([S,a]).               % create a list on the heap with a variable in it

b([A|_],[A|_]).         % bind the two variables together

c([a|_]).               % bind a constant to the variables
                % at this point one list has a constant as its first element
                % and the other has a variable bound to a constant

d(A,A).                 % this unification should succeed
                        % fails here if variables not deref'd in unification
```

12.2. Cdr Bits

Check that NIL, when used as a constant, does not have its cdr bit set:

```
main :- a([[],[]],[A,B]), write(A), nl, write(B), nl.
a(X,X).
```

Another check on the constant NIL:

```
main :- write([x,[]]), nl.
```

Test that cdr bit stays the same during variable binding:

```
main :- A = [x|X], B = [x,Y], X = Y, Y = a, write(x(A,B)), nl.
```

Test cdr links:

```
main :- a([A]), X = [a|Y], Y = [b|A], b(X), write(X), nl, fail.
main :- a([A]), X = [a|Y], Y = [b|A], A = [c], b(X), write(X), nl, fail.
a(_).
b([A,B,C]) :- a([A,B,C]).
```

Another test of cdr links:

```
main :- a([X]), A = [a|Y], Y = [b|X], X = [c], b(A), write(A), nl, fail.
main :- a([X]), A = [a|Y], Y = [b|X], b(A), write(A).
a(_).
b([X,Y,Z]) :- a(X), a(Y), a(Z).
```

Check that the cdr bit of an unbound variable is preserved by `get_structure`:

```
main :- write([a|b(x)]), nl.
```

Check decdring in general unify:

```
% from Chien Chen
main :- a(A), b(B), c(A, B), d(A).
a([a|X]).
b([a| foo(b,c)]).
c(X, X).
d([_|X]) :- c(X, foo(b,c)).
```

Check that the cdr bit is cleared before writing to the heap in `unify_value` (write mode):

```
% from Barry Fagin
main :- a([H|T]), X = Y-T, foo(X), write(X).

a(_).
foo(X-X).
```

Test `unify_void`:

```
main :- y(A), write(a),
        z(A,Z), write(b),
        a([X,Y|Z]), write(c),
        z(M,L), write(d),
        z(M,[a]), write(e),
        a([a|L]).
a([_,_,_]).
y(A).
z(A,A).
```

12.3. Unsafe Variables

Test comparison for current environment in `put_unsafe_value`:

```
main :- a(X), a(Y), b(X,Y), c(X,Y).     % this just causes to put_unsafe's
                                        % but one does not transfer pointer to heap
a(X).
b(X,X).
c(X,Y) :- d(X,Y), e.                    % 'e' is just to force an allocate
d(X,Y) :- a(X), a(Y), f(X,Y), e.        % another allocate that destroys
                                        % pointer for X and Y
e.
f(a,a).                                 % this should succeed
```

Test `put_unsafe_value` when two variable are bound together in the current environment (the end of the variable chain must be changed to point to a newly created unbound variable on the heap):

```
% from Jeff Gee:
n :- a(X,Y),b(Y),write(X),nl.

a(V,V).
b(joe).
```

Check that `unify_value` does unsafe variable globalization in write mode:

```
main :- a(X), d(X), e, write(X), nl.
a(Y) :- c(X,Y), b(X).
b(_).
c(X, [X]).          % called from a/1 with X set to unbnd var on stack
d([x]).
e :- b(X), b(X).
```

Test that the overwritten variable on the stack in `unify_value` (write mode) is trailed.

```
main :- a(X,Y), b(Z), c(X,Y), d(Z).

a(X,X).

b([]).
b([a,b,c]).

c(X,Y) :- var(X), !.
c(X,Y) :- write('*** BUG ***'), nl.

d([_|_]).
```

12.4. Detrailing

Test that multiple setargs (to the same location) are untrailed properly:

```
main :- X = a(a), b(X), fail.

b(X) :- write(X), nl,
     setarg(1, X, b), write(X), nl,
     setarg(1, X, c), write(X), nl.
b(X) :- write(X), nl.
```

13. Suggestions for Future Instruction Sets

13.1. Eliminate Cdr coding

Cdr coding has been shown not to yield any performance advantage [ToDe], and it complicates the microcode. Many of the last bugs to be removed from the VLSI-PLM microcode where related to cdr coding.

13.2. Eliminate Unsafe Variables

Unsafe variables have also been the source of several bugs. By changing `put_variable Yi` to create an unbound variable on the heap and a pointer to it in the environment, unsafe variables are eliminated. The benefits include elimination of the `put_unsafe_value` instruction, simplification of the trail routine (one comparison rather than three), and simplification of `unify_value` in write mode. The drawbacks include an extra dereference link and the creation of more garbage on the heap. The highly recursive Takeuchi function (see Gabriel's lisp benchmarks) is an example of the second drawback:

```
main :- tak(18,12,6,A), write(A), nl.

tak(X,Y,Z,A) :-
     X =< Y, !,
     Z = A.
tak(X,Y,Z,A) :-
     X1 is X-1,
     tak(X1,Y,Z,A1),
     Y1 is Y-1,
     tak(Y1,Z,X,A2),
     Z1 is Z-1,
     tak(Z1,X,Y,A3),
     tak(A1,A2,A3,A).
```

In the VLSI-PLM the variables A1, A2, and A3 are allocated on the stack, and the heap is never used. However, if the these variables are allocated on the heap, 47,706 words are required.

14. Acknowledgements

Thanks to all who contributed to debugging the VLSI-PLM microcode. Special thanks to Tep Dobry, Barry Fagin, Jeff Gee, Chien Chen, Peter Van Roy, Mike Carlton, Jerric Tam, and Vason Srini.

Many of the descriptions given above are derived from electronic mail discussions and I would like to thank those that contributed.

15. References

[Dob] Dobry, T., *A High Performance Architecture for Prolog*, Ph.D. Thesis, Report No. UCB/CSD 87/352, University of California, Berkeley, May 1987.

[FaDo] Fagin, B. and T. Dobry, *The Berkeley PLM Instruction Set: An Instruction Set for Prolog*, Report No. UCB/CSD 86/257, University of California, Berkeley, September 1985.

[ToDe] Touati, H., A. Despain, *An Empirical Study of the Warren Abstract Machine*, 4th Symp. on Logic Programming, 1987.

[War] Warren, D. H. D., *An Abstract Prolog Instruction Set*, Technical Note 309, SRI, October 1983.