PROLOG ENGINE ON THE 3600

David H D Warren
Artificial Intelligence Center
SRI International
7 April 1983

Introduction

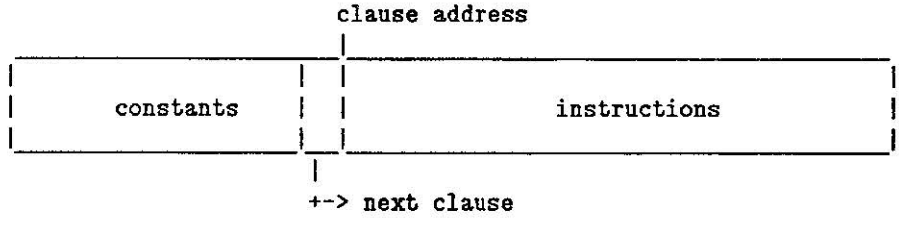
This note sketches some initial ideas for how Prolog Engine could be implemented on the Symbolics 3600 Lisp Machine. The 3600 architecture seems to be almost ideal! The hardware supported tagging and stack buffering are just what we need, and the instruction format will serve very nicely.

Data Formats (cf. Fernando Pereira's LM Prolog)

```
unbound variable        ->    locative pointing to self

reference               ->    locative (or invisible pointer)

constants               ->    same as Lisp
       (integer,                     (fixnum,
        fraction,                     flonum,
        atom)                         symbol)

structure               ->    cdr-coded list
                                 (functor arg1 ... argN)
                              or flavor instance ?

functor                 ->    symbol (of a special kind)
                              or flavor ?

predicate               ->    compiled function (?)
```

Code Format

A clause is encoded, roughly, as:



Thus "big" operands will not be intermingled with instructions, but will be stored in a separate table (one per clause), analogous to the way things are done for Lisp functions.

Instructions are simply 3600 macro-instructions, consisting of a 9-bit opcode and an optional 8-bit operand. The instructions needed are essentially as follows:

```
        OPCODE          OPERAND

{ pop   }   { void    Number                }
{ unify } - { var     Offset-in-stackframe  }
{ push  }   { val     Offset-in-stackframe  }
            { const   Offset-in-clause      }
            { struct  Offset-in-clause      }

  push-pred             Offset-in-clause
  execute               Offset-in-clause

  resume
  proceed
  succeed
  cut-and-succeed
  cut-and-proceed
  cut
    etc.
```

Note that the opcodes for data manipulation are made up of a context (pop, unify, push) plus an operand type ("void" (i.e. single occurrence) variable, unbound variable, bound variable, constant, structure type (i.e. "functor")). Thus we need, as a basic minimum, $3*5 + 2 = 17$ one-operand opcodes, and rather more than 6 no-operand opcodes. If necessary, the number of one-operand opcodes could be reduced by not encoding the context information (pop, unify, push) in the opcode. Such operations would then need extra cycles to dispatch on the context. The number of one-operand opcodes would thereby be reduced to $5 + 2 = 7$.

Some of the operations can probably be arranged to be identical with existing 3600 instructions:

```
pop-void        =       pop-n (?)
pop-var         =       pop-local
push-val        =       push-local
push-const      =       push-constant
(push-pred      =       push-address-local + push-constant ?)
```

Thus only 13 of the original 17 one-operand opcodes are actually new.

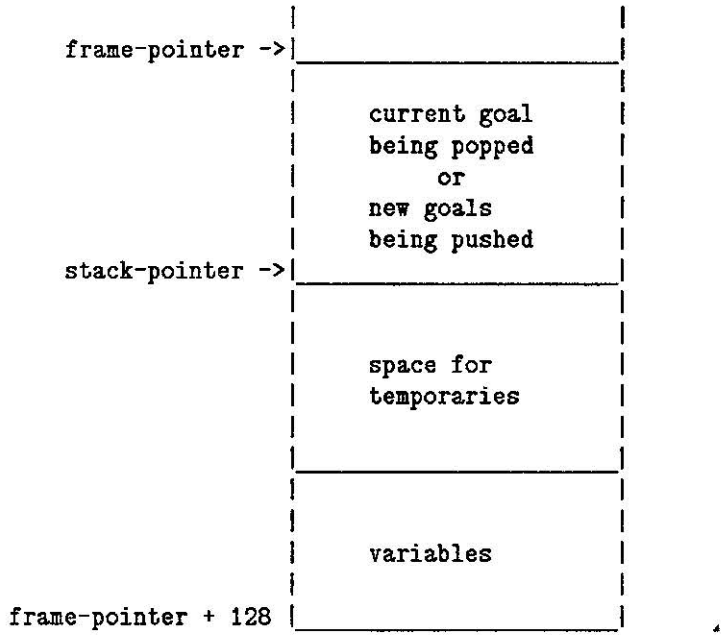
It is envisaged that the more exotic features of Prolog (i.e. evaluable predicates) will be implemented using ordinary 3600 macro-instructions making calls to "quick" functions. Care will be needed to conform to the Lisp conventions for stack usage, etc.

Prolog Machine State

The (local) stack will coincide with the (hardware-supported) Lisp stack, while the heap and trail will be separate regions of main memory. (The A-memory stack buffer contains 4 pages; if these pages do not need to be consecutive in VM, it might be possible to buffer the top of the heap and/or trail as well).

If the currently executing goal is not at the top of the stack (because there is a choice point after it), then it will probably be necessary to copy the goal to the top of the stack so that it can be accessed via the stack-pointer. (NB. Quick function calls and other Lisp macro-instructions we would like to use will corrupt locations above the top-of-stack). This scheme has the side benefit that the compiler can be sure that the current goal is being overwritten, allowing for some neat optimisations!

The layout of memory in the stack buffer will therefore probably be:



This scheme allows us to use Format 2 instructions to address variables. It doesn't allow too much room for body goals and variables, but there should be enough for most clauses in practice. An alternative would be to have variables addressed via XBAS, but the variables would still have to share the stack buffer space in A-memory, and access would be less swift (I think) since we would not be able to take advantage of the hardware-supported Format 2 instruction. If we can get away with it, a better solution would be to have the frame-pointer point directly at the base of the variables. This assumes that there is no hardware constraint requiring the frame-pointer to point before the stack-pointer, and that we are not going to be tripped up by the conventions of the standard microcode which we obviously want to depend on to a large extent.

Sample of Microcoding of a Prolog Instruction

```
(definst unify-var (address-operand needs-stack)
  (parallel                                                    ;[1]
    (assign vma structure-pointer)
    (if (test-some-tag-bit structure-pointer)
      ;Read Mode
      (sequential
        (start-memory read)                                    ;[2]
        (assign structure-pointer (1+ structure-pointer))      ;[3]
        (parallel                                              ;[4]
          (assign address-operand memory-data)
          (next-instruction)))
      ;Write Mode
      (sequential
        (parallel                                              ;[2]
          (start-memory write)
          (assign memory-data (set-type structure-pointer dtp-locative)))
        (assign address-operand
                (set-type structure-pointer dtp-locative))     ;[3]
        (parallel                                              ;[4]
          (assign structure-pointer (1+ structure-pointer))
          (next-instruction))))))
```

Speed Estimate for 'concatenate' Cycle

The instructions executed for one cycle of the (tail-recursive) 'concatenate' clause:

```
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

(corresponding to one call of a user-defined **append** in Lisp) would be as follows, with estimates of the number of cycles needed for each instruction:

```
                          Cycles          Memory Accesses

pop_list                   2
unify_var X                4               1
unify_var L1               4               1
pop_var L2                 2
pop_list                  10 = 2+4+4       2
unify_val X                4               1
unify_var L3               4               1
(succeed)
push_val L3                1
push_val L2                1
push_val L1                1
execute "concatenate"
   read predicate          4               1
   read clause addr        4               1
   other gunge             4
                          ---             --
TOTAL                      45              8

45 * 200 ns (?) = 9.0 microseconds  i.e. 110,000 lips !!!
```

110,000 lips is 2.5 times the performance of compiled Prolog on the DEC 2060, and is about 4 times the performance the Japanese Fifth Generation project is predicting for its Prolog machine, Psi. This estimate is probably somewhat optimistic, though, since we've doubtless overlooked a few steps.

Requirements and Potential Problems

To implement Prolog on the 3600 in the way discussed will require:

- about 30 spare opcodes;
- a few A-memory locations;
- some space in control store for a few hundred microinstructions;
- access to the microcode tools;
- information about the conventions etc. of the standard microcode.

The main problem is going to be dovetailing the Prolog microcode into the standard Lisp firmware, in such a way that we can capitalise on all the standard system functions without screwing things up or sacrificing Prolog performance. Handling of interrupts (sequence breaks) is just one area that needs to be thought about.