# The ZIP Virtual Machine

W F Clocksin

January 1983, with subsequent revisions

Prolog-X is an implementation of Prolog which makes use of an abstract (virtual) machine called the ZIP Machine. The ZIP Machine is defined by a pointer format, 32 registers, the format of storage areas, an instruction set, and assumptions about the layout of data structures in memory.

## POINTERS

Every data structure is represented by a (T+V)-bit word which is divided into a tag field of T bits in length, and a val field of V bits in length. The tag field should be 8 bits long. The present software simulator (Prolog-X) reserves 8 but uses about 4. The val field is ideally 32 bits long so that floats can be represented directly. The present software simulator uses a 24-bit val and does not implement floats.

The following formats denote data structures:

| TAG | MNEMONIC | VAL | PURPOSE |
|---|---|---|---|
| 0 | INT | integer value | integer |
| 1 | FLOAT | float value | float |
| 2 | BOX | pointer to a block cell | raw byte string |
| 3 | ATOM | pointer to an atom cell | atom |
| 4 | TERM | pointer to a term instance | term |
| 5 | LINK | pointer to variable referent | instantiation |
| 6 | reserved | reserved | |
| 7 | UNDEF | uninstantiated variable | variable |
| 8 | FUNCTOR | pointer to a functor cell | functor |
| 9 | BLOCK | byte count | cell header |
| 10 | EMPTY | don't care | to catch bugs |
| 11 | TERMIN | don't care | terminate chains |
| 12 | CLAUSE | pointer to a clause cell | clause |
| 13 | TABLE | word count + 1 | cell header |
| 14 | TABREF | pointer to a table cell | vector |
| 15 | PROC | pointer to a fixture | procedure |

Some of the above tags are not strictly necessary. It is possible to represent procedures, clauses, tables, and functors as general terms. This would provide a more uniform and simple set of data structures, but would cost more in memory fetches. For example, to test whether something is a CLAUSE, we test the tag, which is very fast. Representing a clause as a term would mean testing for a clause by fetching and testing its term header (a functor pointer). With little extra effort we could make clauses, etc, seem like terms to the user, and this is done most of the time.

## LAYOUT OF DATA CELLS

In common with other systems used in LISP and POP-2, multiword storage cells are accessed by a pointer to the first word in the cell. The most obvious difference is that LINK pointers are allowed to point directly to words within a cell. Other pointers to within cells are stored in the Trail, discussed below.

```
[ATOM |  ---]------> [ATOM |  ---]---> (hash chain)
                     [FUNC |  ---]---> (functor chain)
                     [BOX  |  ---]---> (name string)



[FUNC |  ---]------> [ATOM |  ---]---> (atom backpointer)
                     [INT  | arity ]
                     [FUNC |  ---]---> (next functor)
                     [PROC |  ---]---> (procedure chain)



[PROC |  ---]------> [INT  | flags ]
                     [ATOM |  ---]---> (defining module)
                     [ATOM |  ---]---> (visible module)
                     [FUNC |  ---]---> (functor backpointer)
                     [PROC |  ---]---> (next procedure)
                     [     |       ] (clauses INT or TABREF)



[CLAU |  ---]------> [INT  | flags ]
                     [     |       ] (index key (various))
                     [PROC |  ---]---> (proc backpointer)
                     [BOX  |  ---]---> (code block)
                     [TABR |  ---]---> (XR table)
                     [     |       ] (clause backpointer)
                     [     |       ] (next clause)



[TERM |  ---]------> [FUNC |  ---]---> (functor pointer)
                     [     |       ] --+
                    .       |
                    .       | (N components)
                    .       |
                 [  |     ]--+


[BOX |   ---]------> [BLOC |  N  ]
                 [        ]--+
                    .       |
                    .       | (N bytes, 0 padded)
                    .       |
                 [        ]--+


[TABR |   ---]------> [TABL |  N  ]--+
                 [  |     ] |
                    .       |
                    .       | (N words, incl header)
                    .       |
                 [  |     ]--+
```

## REGISTERS

The major registers are:

| | |
|---|---|
| XC | current clause pointer |
| XR | external references pointer |
| D | current data pointer |
| PC | current program counter |
| L | current (target) local frame |
| CL | current (source) local frame |
| CP | forward continuation program counter |
| CL0 | forward continuation local frame |
| G | global stack allocated top |
| G0 | global stack committed top |
| H | heap freelist |
| BL | backtrack continuation local frame |
| BG | backtrack global stack top |
| BP | backtrack continuation clause |
| TR | trail allocated top |
| TR0 | trail committed top |

Other registers are internal temporaries or status bits, etc.

## DATA STRUCTURES

Storage is allocated in four main areas, although small scratchpad stacks are used for general unification, reading terms, and other housekeeping that is not a part of the ZIP Machine. The four areas are summarised here, and more detailed discussion is given below.

- Activation records are allocated on the Local stack, which is implemented as a true stack with contiguous storage but allowing indexing into it. Local stack frames do not require garbage collection, as this is done automatically as a result of certain ZIP instructions. Variable slots in the local frames are the source of all roots to data structures.
- The Global stack, in which most temporary data structures are allocated, should also be a true stack, but it is also necessary to index from arbitrary pointers into the stack. Space is automatically recovered on backtracking, although garbage collection is nowadays considered necessary to recover space in situations where backtracking can never occur. Garbage collection of the global stack is discussed below. Persistent data structures are allocated in the Heap, which should be implemented as a heap. Allocations and deallocations are programmed explicitly (for example asserting and retracting clauses), so a simple reference bit garbage collection scheme discussed below is used. The ZIP machine sees the heap only in that registers PC, CP, XR, XC, and BP point into it.
- The Trail is an historical record of variable instantiations. When a variable is instantiated, a pointer to the variable is entered on the Trail so that the variable can be reset when backtracking. The trail also holds other information of a chronological nature required for garbage collection. This is also a true stack, with pushes and pops being done by the ZIP machine.

In addition, the ZIP machine uses a small scratchpad stack when executing code between the FUNCTOR and POP instructions.

Local stack frames, called activation records, are offset from CL. The order of the first eight entries is unimportant but must be consistent. A complete stack frame stores the following entries, although in some cases (determinacy), not all register save entries are used. An ordering could be imposed to increase speed, as registers CP, CL0, XC, and G0 are saved or restored at the same time by several of the machine instructions.

| | |
|---|---|
| 0 | (reserved) |
| 1 | CP |
| 2 | CL0 |
| 3 | BP |
| 4 | G0 |
| 5 | BL |
| 6 | TR0 |
| 7 | XC |

(arguments and local variables)
(temporary local variables)

The argument slots hold the actual parameters of the procedure call. If a variable appears at the top-level in the head of a clause, then its value it simply that of the corresponding actual parameter, and there is no need to allocate a variable slot for it. Locals classified by the compiler as temporaries are allocated nearest the top of the stack, so that the stack space occupied by temporaries can be recovered automatically when the neck of a clause is executed. is

## INSTRUCTIONS

The ZIP Machine is always in one of three states called Processor Modes: ARG, COPY, or MATCH. Many of the below instructions have alternative interpretations depending on the current Processor Mode. Some of the instructions switch the Processor Mode.

Each instruction is encoded by a byte containing a number 0..63, which is combined with the contents of the PM register (2-bit Processor Mode). Arguments, if any, follow in the succeeding zero, one, or two bytes. The argument bytes encode either an XR-pointer offset, a CL-pointer offset, or a literal value 0..255.

In the description of each instruction, we will use the verb 'to mode' to mean 'to take appropriate action depending on the current Processor Mode'. Depending on the current Processor Mode, moding involves: constructing arguments on the local stack (ARG), or unifying individual data structures with others (MATCH), or constructing new individual data structures (COPY). The D Register is normally maintained at the destination of the moding operation. How much work the processor does depends on what is moded. For example, moding anonymous variables involves little or no work.

All of these instructions are generated by an optimising compiler. The compiler is capable of identifying cases where full unification is not required, where data structures should migrate to other areas, where tail recursion is used, where unit and doublet clauses require less housekeeping, and where certain built-in predicates are translated directly as ZIP instructions (instead of generating calls). Special-purpose instructions are generated in these cases. It has been found that over 80 percent of code generated is special-purpose. Thus when the description below refers to a 'general case', it does not mean 'fast and most popular', but usually means 'slower and rarer'.

| | |
|---|---|
| immed n | Modes the integer n. |
| constant n | Modes a constant at XR+n. |
| functor n a | Modes a functor of arity a at XR+n. This instruction is followed by instructions that mode each component of the |

| | |
|---|---|
| | functor, and then a matching 'pop' instruction. |
| lastfunctor n a | Modes a leaf functor of arity a at XR+n, but no matching 'pop' required. |
| void | Modes an anonymous variable. Under certain conditions no operation is performed. |
| voidn n | The next n modings are with anonymous variables. |
| skipvar | The variable at D is known to be moded later (if at all!), so we can ignore it now. Under certain conditions no operation is performed. |
| skipvn n | The next n variables starting at D are known to be moded later (if at all), so they are ignored now. |
| firstvar v | Mode the first occurrence of a temporary or local variable at CL+v which is known to need moding now. |
| glovar v | Mode a variable at CL+v, and migrate it to the global area. |
| glofirvar v | Mode the first occurrence of a variable CL+v and migrate it to the global area. |
| var v | The general case: mode a variable at CL+v. The most common use of this instruction, during ARG mode, entails very little work. A var in ARG mode is also the most popular instruction, being used almost ten times more than the next most popular instruction. |
| pop, popmatch, poparg | These three instructions pop the mode context pushed by the FUNCTOR instruction. Some of them change the Processor Mode. |
| return a | Enter the neck of a unit clause, stack frame size a. |
| argmode a | Enter the neck of a doublet clause, stack frame size a. |
| enter a | The general case: enter the neck of a clause having more than one goal. Stack frame size is a. |
| proceed n a | Call the only goal in a doublet clause. The procedure of arity a is at XR+n. If deterministic, then single solution is implied. |
| depart n a | Call the last goal in a clause. The procedure of arity a is at XR+n. If deterministic, then single solution is implied. |
| call n | General case: call a goal XR+n. |
| callx v | The term at CL+v is to be considered as a goal, and called. The really really general case (interpreted higher-order functions; a 'call(X)' compiles into this). |
| exit | The last goal is an open coded construct, so this forces an exit from the current procedure. |

| | |
|---|---|
| cut a | The activation in the current frame of size a is the only solution (a 'cut' compiles into this). |
| fail | The current activation is not a solution (a 'fail' compiles into this). |
| provar v, prononvar v, proatom v, proint v | In-line tests of CL+v for var, nonvar, atom, and integer (the built-in predicates var, nonvar, atom, integer compile into these). The compiler also attempts where legal to transplant these to the left of an ENTER instruction for better performance. |

There are no instructions defined for disjunction or arithmetic. These facilities are currently written in the base language, and called using procedure calls. This actually gives an incorrect definition of disjunction, for which it is necessary to do tricks with the activation record if a It is intended to define instructions to speed up arithmetic and to properly implement disjunction. At minimum, instructions will be required for add, subtract, multiply, divide, remainder, bit and, bit or, bit not, bit xor, shift left, and shift right. These instructions would use either CL or XR offsets to obtain arguments, and we must therefore also provide a hack to permit passing expressions though variables to be interpreted at run-time.

## NOTES ON USE OF STORAGE AREAS AND GARBAGE COLLECTION

Local Stack. When a goal succeeds determinately, its local frame is discarded. If the procedure is determinate at the point where the last goal in the body of the clause is about to be called, then the frame for that goal replaces the frame for the procedure. This is how tail recursion optimisation is implemented. One problem is as follows. Suppose a goal replaces a frame that has variables that refer to the goal. In this special case, which is detected during compilation, space for the affected variables is migrated to the global stack.

Global Stack. As mentioned above, garbage collection is required for the global stack only when inaccessible structures are created in the absence of backtracking. If it is necessary to garbage collect the global stack, then a normal mark-sweep-reallocate algorithm can be used. References to data in the global stack are rooted in the local stack variables. A refinement of the usual algorithm recognises that it it not strictly necessary to mark accessible structures if it is known that the local variable will not be used subsequently in the current goal. This has the effect of reclaiming much space that normally would not become inaccessible until a determinism has been committed.

Heap. Clauses, which are stored in the heap, are garbage collected after they have been retracted. Two bits for each clause are required for this purpose: a REFERENCED bit and a DOOMED bit. Any built-in predicate that returns a clause pointer checks the REFERENCED bit. If the bit is set then no action is taken. If the bit is clear, then we must set the bit and record a trail entry.

When a clause is retracted, the REFERENCED bit is checked. If the REFERENCED bit is clear (most often the case), then the space occupied by the clause is recovered immediately. If the REFERENCED bit is set, then it is not possible to immediately recover the space for it. Instead, the DOOMED bit is set. When backtracking past a clause pointer in the trail ~~entry~~, the referenced bit is cleared. If at this time the DOOMED bit is set, then the space occupied by the clause is recovered.

It is possible that references to cells on the heap may continue to be referenced after the clause in which they appear has been removed. Instead of adopting a reference count for such cells, it is possible to scan the trail for references to variables that refer to the cells. This is possibly expensive for long trails, but the simplicity of the scheme may outweigh the alternative of full reference counts and some new instructions, as has been proposed elsewhere.

Ideas on garbage collection and storage allocation have changed since this was written. The later notes are in a separate paper.