Appendix B

Formal specification of the Berkeley Abstract Machine syntax

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Formal specification of the Berkeley Abstract Machine (BAM) syntax
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California
% May be used and modified for non-commercial purposes if this notice is kept.
% Written by Peter Van Roy.

% This file is an executable Prolog program that checks the syntactic
% correctness of BAM instructions.  The predicate instr(I) is true if I is
% a legal BAM instruction.  In addition to instructions output by the Aquarius
% compiler, this predicate also accepts the user instructions of the BAM,
% which allow the run-time system to be written completely in BAM assembly.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Check correctness of a sequence of BAM instructions ***

% Create saved state:
% Note: In C-Prolog this must be started up in a system
% equal to in size or larger than the one which created it.
main :- save(check, 1), prompt(_, ''), read(Instr), pipe(Instr, 0, 0), halt.
main :- halt.

% Pipe working loop:
pipe(end_of_file, M, N) :- !,
   T is M+N,
   write('*** Checked '),write(T),write(' instructions; '),
   write(M),write(' correct and '),write(N),write(' incorrect.'),nl.
pipe(Instr, M, N) :-
   (instr(Instr)
   -> M1 is M+1, N1=N
   ;  M1=M, N1 is N+1,
      write('*** Incorrect: '),write(Instr),nl
   ),
   !, read(NewInstr), pipe(NewInstr, M1, N1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** BAM Instructions ***

% 1. Unification support instructions:
instr(deref(V,W))          :- var_i(V), var_i(W).
instr(equal(EA,A,L))       :- ea_e(EA), arg_i(A), lbl(L).
instr(unify(V,W,F,G,L))    :- var_i(V), var_i(W), nv_flag(F),nv_flag(G),lbl(L).
instr(trail(V))            :- var_i(V).
instr(move(EA,VI))         :- ea_m(EA), var_i(VI).
instr(push(EA,R,N))        :- ea_p(EA), hreg(R), pos(N).
instr(adda(R,S,T))         :- numreg(R), numreg(S), hreg(T).
```

```
instr(pad(N))              :- pos(N).
instr(unify_atomic(V,I,L)) :- var_i(V), an_atomic(I), lbl(L).
instr(fail).

% 2. Conditional control flow instructions:
instr(switch(T,V,A,B,C))   :- a_tag(T), var_i(V), lbl(A), lbl(B), lbl(C).
instr(choice(I/N,Rs,L))    :- pos(I), pos(N), I=<N, lbl(L), regs(Rs).
instr(test(Eq,T,V,L))      :- eq_ne(Eq), var_i(V), a_tag(T), lbl(L).
instr(jump(C,A,B,L))       :- cond(C), numarg_i(A), numarg_i(B), lbl(L).
instr(move(CH,V))          :- a_var(V), choice_ptr(CH).
instr(cut(V))              :- a_var(V).
instr(hash(T,R,N,L))       :- hash_type(T), reg(R), pos(N), lbl(L).
instr(pair(E,L))           :- an_atomic(E), lbl(L).

% 3. Arithmetic instructions:
instr(add(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(sub(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(mul(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(div(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(mod(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(and(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr( or(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(xor(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(not(A,V))            :- numarg_i(A),              a_var(V).
instr(sll(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(sra(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(sll). /* vlsi_plm only */
instr(sra). /* vlsi_plm only */

% 4. Procedural instructions:
instr(procedure(N/A))      :- atom(N), natural(A).
instr(call(N/A))           :- atom(N), natural(A).
instr(return).
instr(simple_call(N/A))    :- atom(N), natural(A).
instr(simple_return).
instr(label(L))            :- lbl(L).
instr(jump(L))             :- lbl(L).
instr(allocate(Perms))     :- natural(Perms).
instr(deallocate(Perms))   :- natural(Perms).
instr(nop).

% 5. Pragma information for translator and reorderer:
instr(pragma(P))           :- pragma(P).

% 6. Additions to BAM for the assembly language programmer:
instr(I)                   :- user_instr(I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Additions to BAM for the assembly language programmer ***

% This section describes the parts of the BAM language that are never output
% by the compiler, but only used by the BAM assembly programmer.  This is used
% to write the run-time system in BAM code, so that it is as portable as
```

```
% possible.  Additional instructions are jump to register address, convert
% tagged atom or integer to untagged integer (ord), its inverse (val), and
% non-trapping full-word unsigned comparison, non-trapping full-word
% arithmetic, and trailing for backtrackable destructive assignment.

user_instr(jump_reg(R))      :- reg(R).
user_instr(jump_nt(C,A,B,L)) :- cond(C), numarg_i(A), numarg_i(B), lbl(L).
user_instr(ord(A,B))         :- arg(A), a_var(B).
user_instr(val(T,A,V))       :- a_tag(T),    numarg_i(A), a_var(V).
user_instr(add_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(sub_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(and_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr( or_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(xor_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(not_nt(A,V))      :- numarg_i(A),              a_var(V).
user_instr(sll_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(sra_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(trail_bda(X))     :- a_var(X).

% Additional registers:
% See Implementation Manual for list of existing registers.
user_reg(r(A)) :- atom(A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Pragmas ***

% A variable is a multiple of N.
% Inserted just before loads in readmode unification.
pragma(align(V,N))         :- a_var(V), pos(N).

% Inserted just before a sequence of pushes in writemode unification.
% (The pushes may be interleaved with non-memory moves.)
pragma(push(term(Size)))   :- pos(Size).
pragma(push(cons)).
pragma(push(structure(A))) :- pos(A).
pragma(push(variable)).

% Specify the tag of a variable.
% (This is useful for processors without explicit tag support.)
pragma(tag(V,T))           :- a_var(V), a_tag(T).

% Length of a hash table.
pragma(hash_length(Len))   :- pos(Len).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Tags ***

a_tag(tatm). /* atom */
a_tag(tint). /* integer */
a_tag(tneg). /* negative integer */
a_tag(tpos). /* nonnegative integer */
a_tag(tstr). /* structure */
```

```
a_tag(tlst). /* cons cell */
a_tag(tvar). /* variable */

atom_tag(tatm).

pointer_tag(tstr).
pointer_tag(tlst).
pointer_tag(tvar).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Addressing modes ***

heap_ptr(r(h)).
choice_ptr(r(b)).

reg(r(I))          :- int(I).
reg(U)             :- user_reg(U).

hreg(R)            :- reg(R).
hreg(R)            :- heap_ptr(R).

perm(p(I))         :- natural(I).

an_atomic(I)       :- int(I).
an_atomic(T^A)     :- atom(A), atom_tag(T).
an_atomic(T^(F/N)) :- atom(F), pos(N), atom_tag(T).

a_var(Reg)         :- reg(Reg).
a_var(Perm)        :- perm(Perm).

arg(Arg)           :- a_var(Arg).
arg(Arg)           :- an_atomic(Arg).

var_i(Var)         :- a_var(Var).
var_i([Var])       :- a_var(Var).

arg_i(Arg)         :- var_i(Arg).
arg_i(Arg)         :- an_atomic(Arg).

numreg(Arg)        :- reg(Arg).
numreg(Arg)        :- int(Arg).

numarg_i(Arg)      :- var_i(Arg).
numarg_i(Arg)      :- int(Arg).

var_off([Var])     :- a_var(Var).
var_off([Var+I])   :- a_var(Var), pos(I).

% Effective address for equal:
ea_e(Var)          :- a_var(Var).
ea_e(VarOff)       :- var_off(VarOff).

% Effective address for move:
```

```
ea_m(Arg)          :- arg(Arg).
ea_m(VarOff)       :- var_off(VarOff).
ea_m(Tagˆ H)       :- pointer_tag(Tag), heap_ptr(H).

% Effective address for push:
ea_p(Arg)          :- arg_i(Arg).
ea_p(Tagˆ H)       :- pointer_tag(Tag), heap_ptr(H).
ea_p(Tagˆ (H+D))   :- pointer_tag(Tag), pos(D), heap_ptr(H).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Miscellaneous ***

eq_ne(eq). /* Equal */
eq_ne(ne). /* Not equal */

cond(lts). /* Signed less than */
cond(les). /* Signed less than or equal */
cond(gts). /* Signed greater than */
cond(ges). /* Signed greater than or equal */
cond(eq).  /* Equal */
cond(ne).  /* Not equal */

hash_type(atomic).
hash_type(structure).

lbl(fail).
lbl(N/A)      :- atom(N), natural(A).
lbl(l(N/A,I)) :- atom(N), natural(A), natural(I).

nv_flag(nonvar).
nv_flag(var).
nv_flag('?').

% A list of register numbers:
% (May contain the value 'no' as well)
regs([]).
regs([R|Set]) :- (int(R); R=no), regs(Set).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Utilities ***

ground(X) :- nonvar(X), functor(X, _, N), ground(N, X).

ground(N, _) :- N=:=0.
ground(N, X) :- N=\=0, arg(N, X, A), ground(A), N1 is N-1, ground(N1, X).

int(N)     :- integer(N).
natural(N) :- integer(N), N>=0.
pos(N)     :- integer(N), N>0.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```