Appendix D

Semantics of the Berkeley Abstract Machine

1. Introduction

This appendix gives an English-language description of the semantics of the Berkeley Abstract Machine (BAM) as comments attached to a Prolog specification of its syntax. The BAM is intended to operate on the same data structures as the Warren Abstract Machine (WAM), therefore some familiarity with the WAM is an advantage. The semantics are represented by short descriptions supplemented by pseudo-code and examples where necessary.

The BAM is designed to be simple and easily translated to most general-purpose processors. Many of its optimizations apply to any processor, for example the streamlined choice point management and the use of write-once permanent variables to simplify trailing. Although the first target is the VLSI-BAM processor, we have built translators for other processors including the MIPS and the MC68020. Pragmas give information that is used to obtain the best translation for different processors.

The instruction set is divided in six categories, each in a different section. Each section starts with a box giving the syntax of the instructions presented in that section. This is followed by a description of the instructions' actions. Section 2 gives the unification instructions. Section 3 gives the conditional control flow instructions. Section 4 gives the arithmetic instructions. Section 5 gives the procedural control flow instructions. Section 6 gives the pragmas, which contain information that allows better translation. Section 7 gives the user instructions, additions to the BAM that are never output by the compiler but are intended for the BAM assembly programmer. The last section defines the syntax and semantics of the addressing modes used in the instructions.

In explaining the semantics, a few assumptions are made about the data representation. An infinite number of registers is assumed; the translator should map registers of sufficiently large index to memory. A tagged architecture is assumed; i.e. each word contains a tag and a value field which are treated as separate entities in some instructions and as a unit in other instructions. A load-store architecture is assumed; almost any architecture has a subset of instructions that satisfy this assumption. The actual details of the translation to the target architecture are not given since they depend on the characteristics of the architecture. These characteristics include the number of registers, the addressing modes, hardware support for certain features (tagging, dereferencing, trailing, etc.), the precise format of choice points and environments, and so forth.

2. Unification instructions

Unification syntax

```
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).
```

`deref(V,W)` — Dereference the argument V and store the result in W. The argument V is unchanged. This is the only instruction which dereferences its argument. All other instructions assume that their arguments are dereferenced. Giving the dereference instruction two arguments simplifies the implementation of write-once permanent variables and makes a fast implementation of trailing possible.

`equal(X,Y,L)` — Compare X to Y and branch to L if they are not equal. The comparison is a full word operation, equivalent to ‘‘eq’’ in Lisp. It is assumed that X and Y are dereferenced.

`unify(X,Y,T,U,L)` — Perform a general unification of X and Y, and branch to L if it fails. Always binds oldest variables to the youngest. In the failure case all bindings are undone. It is assumed that X and Y are dereferenced. The two parameters T and U are added as an optimization, and may be safely ignored. They are flags (with values `'?'`, `var`, or `nonvar`) that say whether it is known if X and Y are variables or nonvariables. With this information a better translation to the target processor can be done.

`trail(X)` — Push the address of X on the trail stack if the trail condition X<`r(hb)` is satisfied. It is assumed that X is a dereferenced unbound variable, i.e. it has a `tvar` tag. Only one comparison is necessary for the trail check. The state register `r(hb)` points to the heap location which was the top of the heap when the most recent choice point was created.

`move(X,Y)` — Move X to Y. Depending on the addressing mode, this instruction does a load or store or creates a tagged value.

`push(X,R,N)` — Push X on the stack with stack pointer R, then increment R by N. This instruction is used for write mode unification.

`adda(X,Y,R)` — Add X and Y into R. This is a full word operation which never traps, unlike the arithmetic instructions in section 4. This instruction is used to allocate space for uninitialized variables. The second argument Y is an offset which is scaled properly by the translator (i.e. it is unchanged for the VLSI-BAM since it is word-addressed, and it is multiplied by 4 for the MIPS, since it is byte-addressed).

`pad(N)` Add N words to the heap pointer `r(h)`. This is a full word operation which never traps, unlike the arithmetic instructions in section 4. It is used to ensure the correct alignment of compound terms. The space reserved by pad will never be stored to. If the increment is a multiple of the alignment then the pad disappears. The increment is scaled properly by the translator (see previous description of `adda`).

`unify_atomic(X,Y,L)` Unify the variable X with the atomic term Y, and branch to L if it fails. It is assumed that X is dereferenced. The `unify_atomic` instruction is a special case of general unification that is added to reduce code size in the VLSI-BAM processor. There is a compiler option to enable or disable the generation of this instruction.

`fail` Untrail all variable bindings and jump to the retry address. Do not restore argument registers. Argument registers are restored by the choice point management instructions.

3. Conditional control flow instructions

Clause selection syntax	
`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).`

`switch(T,R,A,B,C)` A three-way branch: branch to the label A, B, C depending on whether the tag of R is `tvar`, T, or any other value. The label `fail` is not an address, but denotes a branch to the global failure routine. It is assumed that R is dereferenced.

`choice(I/N,RS,L)` The choice point management instruction for choosing clause I out of N clauses. Choice points are of variable size. The semantics of choice depends on I as follows:

- I=1 Create a choice point with retry address L. Save in it the registers listed in RS.
- 1<I<N Restore the registers mentioned in RS from the choice point, ignoring `no` terms. The `no` terms make it possible to know the position of the registers in the choice point without an explicit size field in the choice point. Update the retry address to L.
- I=N Restore the registers mentioned in RS, ignoring `no` terms. Remove the choice point. (L will always be `fail` when I=N.)

The above notation is consistent with three possible implementations (in order of decreasing efficiency): (1) The implementation given above, in which only those registers listed in RS are saved and restored, and the choice point does not have a size field. Restoring registers is done by the choice instructions, not by the fail instruction. The compiler does an effort to minimize the set of registers mentioned in RS. (2) Saving all registers up to the maximum register listed in RS. In this case the choice points are of variable size, and the `no` terms in RS are ignored. The notation is consistent with choice points containing a size field. (3) Always saving and restoring all registers. In this case the choice points are of fixed size, the RS argument is ignored, and the fail instruction restores the registers. In this case the semantics correspond to the try, retry, and trust instructions of the WAM.

`test(E,T,X,L)` Branch to label L if the tag of X is equal/not equal to T. Equality/nonequality is selected by the value of E. The label `fail` is not an address, but denotes a branch to the global failure routine. It is assumed that X is dereferenced.

`jump(C,X,Y,L)` Compare X and Y and jump to L if the comparison is true. The kind of comparison is given by C. This instruction traps if either argument is not an integer. The label `fail` is not an address, but denotes a branch to the global failure routine.

`cut(X)` Implement the cut operation. Move X into the `r(b)` register; also move the value of `r(h)` in this choice point into the `r(hb)` register. The latter move is an optimization that reduces the number of trailed variables, but is not needed for correctness. The compiler ensures that X contains a pointer to the choice point which was most recent when the current predicate was entered.

`hash(T,R,N,L)` Look up register R in a hash table located at label L. The hash table contains atomic terms (when T=`atomic`) or the main functors of structures (when T=`structure`). If R is not in the hash table, then execution falls through to the next instruction. Otherwise execution continues at the label contained in the hash table. When T=`structure` the compiler guarantees that R points to a structure. The following is an example of hash table code:

```
 ...
hash(Type,Reg,N,Lbl). ; Hash Reg into table at Lbl
 ...                  ; Fall through if not present


label(Lbl).           ; The hash table
hash_length(N).       ; Length of the hash table
pair(E1,L1).          ; N entries
pair(E2,L2).
 ...
pair(Ei,Li).          ; Jump to Li if Reg = Ei
 ...
pair(EN,LN).
```

`pair(E,L)` A hash table entry. E is either an atom or the main functor of a structure. The label L is the address where execution continues if the supplied value matches E.

4. Arithmetic instructions

Arithmetic syntax	
`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(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).`

All arithmetic instructions assume that their operands are dereferenced and destructively overwrite the result register. All perform operations on integers with correct tag and return a result with correct tag, trapping if either operand or the result is not a integer. Arithmetic semantics are:

`add(X,Y,Z)`	Z ← X+Y
`sub(X,Y,Z)`	Z ← X-Y
`mul(X,Y,Z)`	Z ← X*Y
`div(X,Y,Z)`	Z ← X/Y
`and(X,Y,Z)`	Z ← X and Y (bitwise and)
`or(X,Y,Z)`	Z ← X or Y (bitwise or)
`xor(X,Y,Z)`	Z ← X xor Y (bitwise exclusive or)
`sll(X,Y,Z)`	Z ← X << Y (logical shift of X left Y places)
`sra(X,Y,Z)`	Z ← X >> Y (arithmetic shift of X right Y places)
`not(X,Z)`	Z ← not X (bitwise invert X into Z)

5. Procedural control flow instructions

Procedural syntax	
`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).`

`procedure(P)`	The entry point of procedure P.
`call(N/A)`	Call the procedure `N/A`, assuming a fixed location for the arguments. The arguments of `N/A` are sequentially loaded into argument registers. By default the registers used are numbered from zero, i.e. `r(0)`, `r(1)`, ... This call is used for all user-defined predicates. It may be nested, but must be surrounded by an allocate-deallocate pair when used in the body of a predicate.
`return`	Return from a call.

`simple_call(N/A)`	Simple call of the procedure `N/A`, assuming the same argument passing as `call(N/A)`. This is a one-level call; it may not be nested. It does not require a surrounding allocate-deallocate pair. It can be implemented by saving the return address in a fixed register. This instruction is useful for interfacing with assembly routines.
`simple_return`	Return from a simple call.
`label(L)`	Denotes a branch destination. The label `fail` is not an address, but denotes a branch to the global failure routine.
`jump(L)`	Jump unconditionally to label L. The label may be to the first instruction of another procedure `N/A` or it may be internal to the current procedure. The label `fail` is not an address, but denotes a branch to the global failure routine.
`allocate(N)`	Create an environment of size N on the local stack, i.e. a new set of N permanent variables which are denoted by `p(I)`. Typically, the only state registers stored in the environment are `r(e)` and `r(cp)`. The environment must NOT contain the `r(b)` register.
`deallocate(N)`	Remove the top-most environment (which is of size N) from the local stack.

6. Pragmas

Pragma syntax

```
instr(pragma(Pragma)) :- pragma(Pragma).

pragma(align(V,N))         :- a_var(V), pos(N).
pragma(push(term(Size)))   :- pos(Size).
pragma(push(cons)).
pragma(push(structure(A))) :- pos(A).
pragma(push(variable)).
pragma(tag(V,T))           :- a_var(V), a_tag(T).
pragma(hash_length(Len))   :- pos(Len).
```

`align(V,N)`	At this point the contents of register or permanent V are a multiple of N. This information helps the reordering stage to generate double-word load instructions for the VLSI-BAM processor.
`hash_length(N)`	N is the length of the hash table starting at this point.
`push(term(S))`	At this point a block of push instructions is about to create a term of size S on the heap.
`push(cons)`	At this point a cons cell (of size two words) is about to be created on the heap. This information helps the reordering stage to generate double-word push instructions for the VLSI-BAM processor.
`push(structure(A))`	At this point a structure of arity A is about to be created on the heap. This information helps the reordering stage to generate double-word push instructions for the BAM processor.
`push(variable)`	At this point an unbound, initialized variable is about to be created on the heap.
`hash_length(N)`	This is the start of a hash table of length N.

`tag(V,T)` The contents of variable V have tag T. This pragma precedes a load or a store with address V. It is used to make loads and stores efficient for processors which do not have explicit tag support.

7. User instructions

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, creating and decomposing tagged words, non-trapping full-word arithmetic, non-trapping full-word unsigned comparison, and trailing for backtrackable destructive assignment. Additional registers are used in implementing the run-time system, and can be mapped to memory locations.

Additional instructions

```
instr(I) :- user_instr(I).

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).

user_reg(r(A))              :- atom(A).
```

`jump_reg(R)` Jump unconditionally to the address stored in register R.

`jump_nt(C,A,B,L)` Compare A and B and jump to L if the comparison is true. The kind of comparison is given by C. This instruction does a full word comparison and never traps. The label `fail` is not an address, but denotes a branch to the global failure routine.

`ord(A,B)` Store in B the machine integer that corresponds to the atom or integer in A. This function strips the tag from A, and therefore depends on the target machine and the program that is compiled. It is used to convert atoms and integers into table indices.

`val(T,A,V)` Create a tagged word in B by combining the tag T and the machine integer in A. This function is the inverse of `ord(A,B)`: In the sequence `ord(A1,B), val(T,B,A2)` the argument `A2` will receive an identical value to `A1` if `T` is the tag of `A1`.

`add_nt(A,B,V)`
`sub_nt(A,B,V)`
`and_nt(A,B,V)`
`or_nt(A,B,V)`
`xor_nt(A,B,V)`
`not_nt(A,B,V)`
`sll_nt(A,B,V)`
`sra_nt(A,B,V)`

These arithmetic instructions destructively overwrite the result register All perform operations on full words, return a full word, and never trap. See the previous section on arithmetic for a description of the operations performed.

`trail_bda(X)`

Push the address and value of X on the trail stack if the trail condition X<`r(hb)` is satisfied. It is assumed that X is dereferenced. When detrailing, the old value of X is restored. This is used to implement backtrackable destructive assignment. Only one comparison is necessary for the trail check. The state register `r(hb)` points to the heap location which was the top of the heap when the most recent choice point was created.

8. Instruction arguments

This section defines the syntax of the instructions' arguments.

Addressing modes for equal, move and push

```
% 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(Tag^H)      :- pointer_tag(Tag), heap_ptr(H).

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

Other addressing modes

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

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

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).
```

Tag syntax

```
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).
```

Conditionals syntax

```
eq_ne(eq).
eq_ne(ne).

cond(eq).  /* Equal */
cond(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 */
```

Miscellaneous syntax

```
hash_type(atomic).
hash_type(structure).

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

Utility predicates

```
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.
```