Virtual Machine (VIM)

This section describes the architecture and the instruction set of the **VI**rtual **M**achine (VIM).

Architecture

Before the VIM code is interpreted, it is read into the instruction table and the instruction counter **IC** (initially 0) points to the instruction being executed. The instruction table is executed sequentially, which means that after the execution of an instruction the next value of **IC** (**NextIC**) is assigned the value **IC**+1. An exception to this rule are those instructions involving jumps and subroutine calls.

Each instruction can be preceded by an optional label number (followed by a colon). If a label number *lnr* is read, then the corresponding **IC** will be stored in the label table **LabTab** at index *lnr*. When a conditional jump instruction with argument **label** is executed, the value of **LabTab**[**label**] is assigned to **NextIC** if the jump condition is true. For example, the VIM code of a conditional clause could be:

```
    ...
    <boolean expression>
    jiff 22       /* jump if false */
    <then clause>
    jump 23       /* jump always */
22: <else clause>
23: ...
    ...
```

The interpreter uses a stack to temporarily store results of instructions. The stack is an array of integer-typed elements, which is initially empty. All other typed elements are mapped onto the type integer.
The instructions are implemented using the stack operators **push** to push an element on top of the stack, and **pop** to pop the topmost element off the stack.

Return addresses are saved on the return stack. The *call* instruction pushes the current value of **IC**+1 on the return stack. The moment a *return_* instruction is executed the return address is popped off the return stack and assigned to **NextIC**.

On entering a program block or procedure, a data segment of appropriate length must be created. A data segment contains the following information:

- sn, len — segment number (i.e. scope level) and segment length.
- static — pointing to the data segment of the defining environment.
- dynamic — pointing to the data segment of the calling environment.
- data — the data space for local variables and parameters.

Using this information, the calling environment can be recovered when we exit from a program block or procedure. Only when calling a procedure from the declaring environment, will the static and dynamic information be equal, otherwise they will differ. On entering a program block, the static and dynamic information are always equal. A program block is considered to be a non-parameterised procedure called from the defining environment.
The *crseg sn, index* instruction initiates a new segment of appropriate length, the *static* and *dynamic* information are administered also.

Usually, the segment length is unknown at the time a segment is to be created. Therefore, the argument *index* of the *crseg* instruction points to an entry in the length table **LenTab** containing the desired segment length. The length table is filled at compile-time (using API actions *get_index()* and *enter_length()*) and is appended after the last instruction by API action *finalise_vimcode()*
The space for data segments is contained in the array **Data**, the data-stack pointer **Dsp** points to the first free position of the array **Data**. The locations of variables in a data segment are indicated by the segment number *sn* in the Current Segment Group array **CSG** and the displacement *dpl* within that segment, i.e. **Data**[**CSG**[*sn*]+*dpl*]. The current segment number is stored in the variable **Csn**.
String constants can be written to the standard output with the *wrstring* instruction. String constants are stored in the string table **StringTab** that is filled at compile-time (using API action *int_repr_string()*).
The string table is appended to the length table by the API action *finalise_vimcode()*. New string constants can be added to the string table with the *rdstring* instruction.

A number of limits apply to the interpreter. Exceeding these limits will cause an error and the interpreter will stop. An error message and the current value of the instruction counter will be printed. The limits are:

- stack-element: sizeof(integer),
- size of instruction table: 1000 entries,
- size of label table: 250 entries,
- size of stack: 100 entries,
- size of return stack: 100 entries,
- size of length table: 250 entries,
- size of string table: 200 entries,
- size of type string: 255 characters, and
- nesting of data segments: 15 levels.

Instruction set

This section describes the instruction set of the VIrtual Machine (VIM) code interpreter of SLADE. The instruction set includes stack instructions, arithmetic, boolean, and relational operators, jump and call instructions, segment instructions, load and store instructions, and I/O instructions. The effects of these operations are described using the mechanisms and functions mentioned above. The API module *vimcode* contains actions for generating VIM instructions (see previous chapter).

The instruction set is based upon the instruction set listed on page 267 of [AN96]. To do the laboratory exercises some extra instructions are defined, these instructions are marked with a *. The following layout is used:

`Name`	names the instructions,
`Definition`	defines the named instructions, indicating how to use them,
`Description`	describes the functionality of the named instructions. The names of the instructions are printed in *italics*, and
`See also`	refers to related VIM instructions or actions from the Application Programming Interface (API).

Noop, pop, swap

Name noop, pop*, swap*.

Definition

```
noop        ;
pop         pop   value;
swap        pop   value1;
            pop   value2;
            push  value1;
            push  value2;
```

Description *noop* is a dummy instruction, it is often used to address the following (non-*noop*) instruction with more than one label.
pop pops one value from the top of the stack (and discards it). This instruction is added to clean up the stack.
swap pops two values from the stack and pushes them back in reverse order (i.e. the two topmost values on the stack are swapped). This instruction is added to reverse the order of an address/value pair on top of the stack.

See also `emit(), emit_swap_pop()` `(Module vimcode)`

Relational instructions

Name eq, ge, gt, le, lt, ne.

Definition

```
eq          pop   value1;
            pop   value2;
            push  (value2 == value1);
ge          pop   value1;
            pop   value2;
            push  (value2 >= value1);
gt          pop   value1;
            pop   value2;
            push  (value2 > value1);
```

```
le              pop   value1;
                pop   value2;
                push  (value2 <= value1);
lt              pop   value1;
                pop   value2;
                push  (value2 < value1);
ne              pop   value1;
                pop   value2;
                push  (value2 != value1);
```

Description The instructions *eq*, *ge*, *gt*, *le*, *lt* and *ne* are the relational ANSI C operators ==, >=, >, <=, < and !=. The right operand is on top off the stack. The left operand is just below the top. The operands are popped off the stack. The result of the relational expression is pushed on the stack.

See also

```
jiff, jift                        (VIM instruction set)
dy_op_int()                            (Module vimcode)
```

Arithmetic instructions

Name abs_, add, dvi, mdl, mul, neg, sub.

Definition

```
abs_            pop   value;
                if (value < 0)
                   push -value;
                else
                   push value;
add             pop   value1;
                pop   value2;
                push  value2 + value1;
dvi             pop   value1;
                pop   value2;
                push  value2 / value1;
mdl             pop   value1;
                pop   value2;
                push  value2 % value1;
mul             pop   value1;
                pop   value2;
                push  value2 * value1;
neg             pop   value;
                push  -value;
sub             pop   value1;
                pop   value2;
                push  value2 - value1;
```

Description The instructions *abs_*, *add*, *dvi*, *mdl*, *mul*, *neg* and *sub* are the arithmetic integer operators absolute, add, divide, modulo, multiply, negate and subtract. Only the instructions *abs_* and *neg* take one operand. The other instructions take two operands. The right operand is on top of the stack. The left operand is just below the top. The operands are popped off the stack. The result of the arithmetic expression is pushed on the stack.

See also `dy_op_int(), mon_op_int()` (Module vimcode)

Boolean instructions

Name and*, or*, not*.

Definition

```
and             pop   value1;
                pop   value2;
                push  (value2 && value1);
or              pop   value1;
                pop   value2;
                push  (value2 || value1);
not             pop   value;
                push  (value == 0);
```

Description The instructions *and*, *or* and *not* are the boolean operators. Only the instruction *not* takes one operand. The other instructions take two operands. The right operand is on top of the stack. The left operand is just below the top. The operands are popped off the stack. The result of the boolean expression is pushed on the stack. These instructions are added to support boolean operators in your input grammar.

See also `jiff, jift` (VIM instruction set)
`dy_op_bool(), mon_op_bool()` (Module vimcode)

Read instructions

Name rdbool*, rdchar*, rdint, rdstring*.

Definition

```
rdbool          read  bool_value;
                push  int_repr(bool_value);
rdchar          read  char_value;
                push  int_repr(char_value);
rdint           read  value;
                push  value;
rdstring        read  string_value;
                push  int_repr(string_value);
```

Description The instructions *rdbool*, *rdchar*, *rdint* and *rdstring* get a value (of the corresponding type) from standard input, convert this value into the internal (integer) representation, and push the converted value. The instruction *rdstring* creates a new entry in the string table and pushes the corresponding index. The instructions marked with a * are added to support reading of boolean, character or string values in your grammar.

See also

```
wrbool, wrchar, wrint, wrstring     (VIM instruction set)
emit(), emit_read()                      (Module vimcode)
```

Write instructions

Name wrbool*, wrchar*, wrint, wrstring*.

Definition

```
wrbool              pop   value;
                    push  value;
                    write bool_repr(value);
wrchar              pop   value;
                    push  value;
                    write char_repr(value);
wrint               pop   value;
                    push  value;
                    write value;
wrstring            pop   value;
                    push  value;
                    write StringTab[value];
```

Description The instructions *wrbool*, *wrchar*, *wrint* and *wrstring* pop and push(!) a value, convert this value from the internal (integer) representation into the corres-ponding type, and write the converted value to standard output. The instructions marked with a * are added to support writing of boolean, character or string values in your grammar. Note that all write instructions leave the top of the stack unchanged, this is to support print statements that should return a value.

See also

```
rdbool, rdchar, rdint, rdstring     (VIM instruction set)
emit(), emit_write()                     (Module vimcode)
```

Jump instructions

Name jiff, jift, jump.

Definition

```
jiff label          pop   value;
                    if (value == 0)
                       NextIC = LabTab[label];
jift label          pop   value;
                    if (value != 0)
                       NextIC = LabTab[label];
jump label          NextIC = LabTab[label];
```

Description *jiff* (jump if false) pops *value* from the stack and jumps only to the specified *label* when *value* equals 0.
jift (jump if true) pops *value* from the stack and jumps only to the specified *label* when *value* is not equal to 0.
jump performs an unconditional jump to the instruction labelled with *label*. The next value of the instruction counter is derived from the label table entry `LabTab[label]`.

See also
```
call, return_                         (VIM instruction set)
emit_jump()                               (Module vimcode)
```

Call, halt, return_

Name call, halt, return_.

Definition
```
call label          push_return IC+1;
                    NextIC = LabTab[label];
halt                halt;
return_             pop_return retaddr;
                    NextIC = retaddr;
```

Description The *call* and *return_* instructions provide for the procedure call and return actions.
call pushes the current value of **IC**+1 on the return stack and jumps unconditionally to the instruction pointed to by **LabTab**[*label*].
halt stops the execution of the VIM code.
*return_* pops a return address *retaddr* off the return stack and jumps unconditionally to the instruction pointed to by *retaddr*.

See also
```
crseg, dlseg                          (VIM instruction set)
emit_call(), finalise_vimcode()           (Module vimcode)
```

Segment instructions

Name crseg, dlseg.

Definition
```
crseg sn idx        len = LenTab[idx];
                    Data = realloc(Data, Dsp+len+4);
                    Data[Dsp++] = sn;
                    Data[Dsp++] = len;
                    Data[Dsp++] = CSG[sn-1];  /* static */
                    Data[Dsp++] = CSG[Csn];  /* dynamic */
                    Csn= sn;
                    CSG[Csn]= Dsp;
dlseg               curseg = CSG[Csn];
                    len = Data[curseg - 3];
                    static_link = Data[curseg - 2];
                    dynamic_link = Data[curseg - 1];
                    if (static_link != dynamic_link) {
```

```
    callseg = dynamic_link;
    callsn = Data[dynamic_link-4];
    for (i = callsn; i >= Csn; i--) {
       CSG[i] = callseg;
       callseg = Data[callseg-2];
    }
    Csn = callsn;
 } else if (Csn > 0) {
    CSG[Csn] = 0;
    Csn--;
 }
 Dsp -= len + 4;
 Data = realloc(Data, Dsp);
```

Description *crseg* creates a data segment with segment number *sn* (= scope level). The parameter *idx* points to an entry in the length table, containing the corres-ponding segment length. The static, dynamic and data information of the data segment are given the proper values.
dlseg deletes the last created data segment, and makes the data segment of the calling environment the current one.

See also

call, return_	(VIM instruction set)
emit_crseg()	(Module vimcode)

Load instructions

Name ldcon, ldind, ldvar, varaddr.

Definition

```
ldcon value        push  value;
ldind              pop   address;
                   push  address;
                   push  Data[address];
ldvar sn dpl       push  Data[CSG[sn]+dpl];
varaddr sn dpl     push  CSG[sn]+dpl;
```

Description *ldcon* (load constant) pushes the value of its (integer) argument onto the stack.
The instructions *ldind* and *ldvar* are the segment-load instructions, which transfer data from the data segment onto the stack.
ldind (load indirect) pops the absolute address *address* off the stack, and pushes *address* and the contents of **Data**[*address*] on the stack.
Note: *ldind* leaves the address on the stack, this is to support assignment statements, which should return a value.
ldvar (load variable) pushes the contents at **Data**[**CSG**[*sn*]+*dpl*] on the stack, i.e. a combined *varaddr*, *ldind*, *swap*, *pop* sequence.

varaddr (variable address) pushes the absolute address **CSG**[*sn*]+*dpl* on the stack.

See also ldnvar, stind, stnvar, stvar, swap, pop (VIM instruction set)
emit_ldcon(), emit_load() (Module vimcode)

Store instructions

Name stind, stvar.

Definition

```
stind             pop   value;
                  pop   address;
                  push  address;
                  Data[address] = value;
stvar sn dpl      pop   value;
                  Data[CSG[sn]+dpl] = value;
```

Description The instructions *stind* and *stvar* are the segment-store instructions, which transfer data off the stack into the data segment.
stind (store indirect) pops *value* and the absolute address *address* off the stack, pushes *address* back on the stack, and stores *value* to **Data**[*address*].
Note: *stind* leaves the address on the stack, this is to support multiple assignment statements in your input grammar.
stvar (store variable) pops *value* off the stack, and assigns it to **Data**[**CSG**[*sn*]+*dpl*], i.e. a combined *varaddr, swap, stind, pop* sequence.

See also ldind, ldnvar, ldvar, stnvar, varaddr, swap, pop (VIM instruction set)
emit_store() (Module vimcode)

Array instructions

Name descr, eqn*, ldnvar*, ldxvar, nen*, popn*, stnvar*, stxvar, xvaraddr.

Definition

```
descr sn dpl dim  size = 1;
                  for (i = dim; i > 0; i--) {
                     pop   up;
                     pop   lo;
                     size *= up - lo + 1;
                     Data[CSG[sn]+dpl+2*i+1] = lo;
                     Data[CSG[sn]+dpl+2*i] = size;
                  }
                  offset = Data[CSG[sn]-3];
                  Data[CSG[sn]+dpl+1] = offset;
                  Data[CSG[sn]+dpl] = dim;
                  Data[CSG[sn]-3] += size;
```

```
                   Dsp += size; realloc(Data, Dsp);
eqn sn dpl         pop   n;
                   if (n == 0)        /* dynamic array */
                      n = Data[CSG[sn]+dpl+2];
                   push  is_equal(n);

ldnvar sn dpl      pop   n;
                   if (n == 0) {      /* dynamic array */
                      dpl = Data[CSG[sn]+dpl+1];
                      n = Data[CSG[sn]+dpl+2];
                   }
                   for (i = 0; i < n; i++)
                      push  Data[CSG[sn]+dpl+i];
ldxvar sn dpl      xdpl = get_xdpl(sn, dpl);
                   push  Data[CSG[sn]+xdpl];
nen sn dpl         pop   n;
                   if (n == 0)        /* dynamic array */
                      n = Data[CSG[sn]+dpl+2];
                   push  !is_equal(n);
popn sn dpl        pop   n
                   if (n == 0)        /* dynamic array */
                      n = Data[CSG[sn]+dpl+2];
                   for (i = 0; i < n; i++)
                      pop   value;
stnvar sn dpl      pop   n;
                   if (n == 0) {      /* dynamic array */
                      dpl = Data[CSG[sn]+dpl+1];
                      n = Data[CSG[sn]+dpl+2];
                   }
                   for (i = n-1; i >= 0; i--) {
                      pop   value;
                      Data[CSG[sn]+dpl+i] = value;
                   }
stxvar sn dpl      xdpl = get_xdpl(sn, dpl);
                   pop   value;
                   Data[CSG[sn]+xdpl] = value;
xvaraddr sn dpl    /* begin of: get_xdpl(sn, dpl) */
                   pop   dim;
                   xdpl = 0; size = 1;
                   if (dim < 0) {     /* dynamic array */
                      dim = Data[CSG[sn]+dpl];
                      offset = Data[CSG[sn]+dpl+1];
                      for (i = dim; i >= 0; i--) {
                         pop idx;
                         lo = Data[CSG[sn]+dpl+2*i+1];
```

```
      xdpl += (idx-lo)*size;
      size = Data[CSG[sn]+dpl+2*i];
   }
   xdpl += offset;
} else {            /* static array */
   for (i = 0; i < dim; i++) {
      pop idx;
      pop lo;
      xdpl += (idx-lo)*size;
      pop size;
   }
   xdpl += dpl;
}
/* end of: get_xdpl(sn, dpl) */
push  CSG[sn]+xdpl;
```

Description

descr creates a dynamic array descriptor in the data segment at position **CSG**[*sn*]+*dpl*. First the bounds are popped off the stack and stored into the descriptor. Then the offset is computed and stored into the descriptor. Next the dimension *dim* is stored into the descriptor. Finally the size of the data segment is increased to hold the contents of the dynamic array.

eqn compares two arrays of length *n* on the stack. First *n* is popped off the stack, if *n* equals 0, the length *n* is obtained from the dynamic array descriptor at position **CSG**[*sn*]+*dpl*. Then **is_equal**(*n*) is called to compare the first *n* elements with the next *n* elements on the stack. Finally the return value of **is_equal**(*n*) is pushed onto the stack (1 if equal, else 0).

ldnvar loads an array of length *n* from the data segment onto the stack. First *n* is popped off the stack, if *n* equals 0 the displacement *dpl* and length *n* of the array are obtained from the dynamic array descriptor at position **CSG**[*sn*] +*dpl*. Finally the array of length *n*, starting at position **CSG**[*sn*]+*dpl* is pushed onto the stack.

ldxvar (load indexed variable) calculates the value of *xdpl* (see *xvaraddr*) and pushes the contents at **Data**[**CSG**[*sn*]+*xdpl*] on the stack, i.e. a combined *xvaraddr*, *ldind*, *swap*, *pop* sequence.
nen compares two arrays of length *n* on the stack. First *n* is popped off the stack, if *n* equals 0, the length *n* is obtained from the dynamic array descriptor at position **CSG**[*sn*]+*dpl*. Then **!is_equal**(*n*) is called to compare the first *n* elements with the next *n* elements on the stack. Finally the return value of **!is_equal**(*n*) is pushed onto the stack (1 if not equal, else 0).

popn pops an array of length *n* off the stack. First *n* is popped off the stack, if *n* equals 0 the length *n* is obtained from the dynamic array descriptor at position **CSG**[*sn*]+*dpl*. Finally the array of length *n* is popped off the stack.

stnvar stores an array of length *n* from the stack into the data segment. First *n* is popped off the stack, if *n* equals 0 the displacement *dpl* and length *n* of the array are obtained from the dynamic array descriptor at position **CSG**[*sn*]+*dpl*. Finally the array of length *n* is popped off the stack and stored at position **CSG**[*sn*]+*dpl*.

stxvar (store indexed variable) calculates the value of *xdpl* (see *xvaraddr*), pops *value* off the stack and assigns *value* to **Data**[**CSG**[*sn*]+*xdpl*], i.e. a combined *xvaraddr, swap, stind, pop* sequence.

xvaraddr loads the address of an indexed array element onto the stack. First the dimension *dim* is popped off the stack, if *dim* is less than 0 the dimension *dim* is obtained from the dynamic array descriptor at position **CSG**[*sn*]+*dpl*. Then the indices and corresponding bounds are popped off the stack and the displacement is calculated. Finally the address of the indexed array element is pushed onto the stack.

The instructions marked with a * are added to do the laboratory exercises with arrays and records on page 51 and further.

See also

```
eq, ldind, ldvar, ne, pop,
stind, stvar                        (VIM instruction set)
emit_descr(), emit_bound(), emit_opn()   (Module vimcode)
```