The iVM assembler and prototype machine implementations (architecture version 2.0, may 2023)

Source: https://github.com/immortalvm/ivm-implementations

Introduction

The iVM assembler was made for two reasons: To make it feasible to program the iVM by hand, and also to act as a target for the iVM compiler. The assembly language provides some useful shorthands and it abstracts away details that does not regard the assembly programmer or the compiler.

The assembly language: semi-formal EBNF, ignoring whitespace and comments

```
program = import* statement*;

import = "IMPORT" (identifier "/")+ identifier

statement = identifier ":"                                        (* label *)
          | "EXPORT" identifier                                   (* export declaration *)
          | identifier "=" expression                             (* abbreviation *)
          | "data1" "[" expression* "]" ("*" positive_numeral)?    (* data segment,  8 bits per value *)
          | "data2" "[" expression* "]" ("*" positive_numeral)?    (* data segment, 16 bits per value *)
          | "data4" "[" expression* "]" ("*" positive_numeral)?    (* data segment, 32 bits per value *)
          | "data8" "[" expression* "]" ("*" positive_numeral)?    (* data segment, 64 bits per value *)
          | "space" expression                                    (* pointer static byte array *)

          | "exit" | "exit!" expression
          | "push" | "push!" expression | "push!!" expression expression | ...
          | "set_sp" | "set_sp!" expression
          | "jump" | "jump!" expression
          | "jump_zero" | "jump_zero!" expression | "jump_zero!!" expression expression
          | "jump_not_zero" | "jump_not_zero!" expression | "jump_not_zero!!" expression expression
          | "call" | "call!" expression
          | "return"                                              (* alias for "jump" *)
          | "check_version"

          | "load1" | "load1!" expression
          | "load2" | "load2!" expression
          | "load4" | "load4!" expression
          | "load8" | "load8!" expression
          | "sigx1" | "sigx1!" expression
          | "sigx2" | "sigx2!" expression
          | "sigx4" | "sigx4!" expression
          | "sigx8" | "sigx8!" expression                         (* no-op *)
          | "store1" | "store1!" expression | "store1!!" expression expression
          | "store2" | "store2!" expression | "store2!!" expression expression
          | "store4" | "store4!" expression | "store4!!" expression expression
          | "store8" | "store8!" expression | "store8!!" expression expression

          | "add" | "add!" expression | "add!!" expression expression
          | "sub" | "sub!" expression | "sub!!" expression expression
          | "mult" | "mult!" expression | "mult!!" expression expression
          | "neg" | "neg!" expression
          | "and" | "and!" expression | "and!!" expression expression
          | "or" | "or!" expression | "or!!" expression expression
          | "xor" | "xor!" expression | "xor!!" expression expression
          | "not" | "not!" expression
          | "pow2" | "pow2!" expression
```

```
          | "shift_l" | "shift_l!" expression | "shift_l!!" expression expression
          | "shift_ru" | "shift_ru!" expression | "shift_ru!!" expression expression
          | "shift_rs" | "shift_rs!" expression | "shift_rs!!" expression expression

          | "div_u" | "div_u!" expression | "div_u!!" expression expression
          | "div_s" | "div_s!" expression | "div_s!!" expression expression
          | "rem_u" | "rem_u!" expression | "rem_u!!" expression expression
          | "rem_s" | "rem_s!" expression | "rem_s!!" expression expression

          | "lt_u" | "lt_u!" expression | "lt_u!!" expression expression
          | "lt_s" | "lt_s!" expression | "lt_s!!" expression expression
          | "lte_u" | "lte_u!" expression | "lte_u!!" expression expression
          | "lte_s" | "lte_s!" expression | "lte_s!!" expression expression
          | "eq" | "eq!" expression | "eq!!" expression expression
          | "gte_u" | "gte_u!" expression | "gte_u!!" expression expression
          | "gte_s" | "gte_s!" expression | "gte_s!!" expression expression
          | "gt_u" | "gt_u!" expression | "gt_u!!" expression expression
          | "gt_s" | "gt_s!" expression | "gt_s!!" expression expression

          | "read_char"                                            (* interactive *)
          | "read_frame" | "read_frame!" expression
          | ... | "read_pixel!!" expression expression

          | ... | "put_char!" expression
          | ... | "put_byte!" expression
          | ... | "new_frame!!!" expression expression expression
          | ... | "set_pixel!!!!!" expression ...
          | ... | "add_sample!!" expression expression;

expression = positive_numeral  (* 0 to 2^64-1 *)
           | identifier        (* label or abbreviation *)

           | "-" expression    (* corresponding statement: neg *)
           | "~" expression    (* not *)
           | "$" expression    (* stack content *)
           | "&" expression    (* stack pointer *)

           | "(" "+" expression* ")"                (* add *)
           | "(" "*" expression* ")"                (* mult *)
           | "(" "&" expression* ")"                (* and *)
           | "(" "|" expression* ")"                (* or  *)
           | "(" "^" expression* ")"                (* xor *)

           | "(" "="   expression expression ")"    (* eq *)
           | "(" "<u"  expression expression ")"    (* lt_u *)
           | "(" "<s"  expression expression ")"    (* lt_s *)
           | "(" "<=u" expression expression ")"    (* lte_u *)
           | "(" "<=s" expression expression ")"    (* lte_s *)
           | "(" ">u"  expression expression ")"    (* gt_u *)
           | "(" ">s"  expression expression ")"    (* gt_s *)
           | "(" ">=u" expression expression ")"    (* gte_u *)
           | "(" ">=s" expression expression ")"    (* gte_s *)

           | "(" "<<"  expression expression ")"    (* shift_l *)
           | "(" ">>u" expression expression ")"    (* shift_r unsigned, unsigned *)
           | "(" ">>s" expression expression ")"    (* shift_r signed, unsigned *)
           | "(" "/u"  expression expression ")"    (* div_u *)
           | "(" "/s"  expression expression ")"    (* div_s *)
```

```
           | "(" "%u"  expression expression ")"    (* rem_u *)
           | "(" "%s"  expression expression ")"    (* rem_s *)

           | "(" "load1" expression ")"
           | "(" "load2" expression ")"
           | "(" "load4" expression ")"
           | "(" "load8" expression ")"
           | "(" "sigx1" expression ")"
           | "(" "sigx2" expression ")"
           | "(" "sigx4" expression ")"
           | "(" "sigx8" expression ")"             (* identity function *)

identifier = (letter | "_" | "." ) (letter | "_" | "." | digit)*;
```

In v0.8 we added an alternative notation for “immediate arguments”:

```
push* [ <e1> <e2> ... <en> ]
```

is syntactic sugar for:

```
push!!..! <e1> <e2> ... <en>        # with n exclamation marks
```

Similarly for the other statements, e.g. `set_pixel*`.

The arguments to `space` and `data<N>` must be compile time constants, except that `data8` also accepts labels.

Examples

Simple assembly

```
### iVM assembly language introduction.
###
### Part 1 - Statements
###
### This file explains the iVM assembly  language. It is itself a valid assembly
### file, but the code does not make  much sense. Notice that '#' indicates that
### the rest of the line is a comment.
###
### Other than comments,  an iVM assembly file consist of  a list of statements.
### Whitespace is not significant, but it recommended to put each statement on a
### separate line.
###
### The assembly language  is case-sensitive, with all  the instructions written
### in lower case (even though we have  used upper case in the headings for them
### to stand out).


### 1. SPECIAL STATEMENTS

    ## There are six special  statements: imports, labels, exports, definitions,
    ## data and space statements.

    ## Import statements can only occur at the  top of the file. It means that a
    ## label  in file  can  be referenced  below (provided  that  the label  was
    ## exported in the other file). Circular dependencies are not allowed.
    IMPORT intro2_basics/x

    ## A  label statement  indicates a  place in  the code  (memory address)  at
    ## runtime. By convention, all other statements should be indented.
my_label:

    ## For a label to be visible from other files, it must be "exported".
    EXPORT my_label

    ## Definitions define abbreviations, usually constants.
    prime_number = 982451653

    ## A label can be exported under a different name as follows:
    external_name = my_label
```

```
    EXPORT external_name

    ## Labels and definitions use the  same namespace, which is independent from
    ## the names  of instructions. Thus, you  are free to define  a label called
    ## 'add'  or 'exit'.  Names of  labels and  definitions consist  of letters,
    ## digits and underscore (_), and they cannot start with a digit.

    ## Data statements specify bytes that should be included in the binary as is
    ## (more or less). It includes a whitespace-separated list of constant
    ## expressions, usually numbers. All numbers can be specified in decimal,
    ## octal or hexadecimal notation.

    ## data1 includes a list of bytes in the binary. Only the 8 least
    ## significant bits are used of each number. Thus, the last number can be
    ## replaced by 1.
    data1 [ 0 1 -2 0o200 -0x99ff ]

    ## Similarly, data2 includes a list of (little-endian) 16-bit words, using
    ## the 16 least significant bits.
    data2 [ 0x1000 0x2000 0x3000 ]

    ## data4 and data8 include lists of 32-bit and 64-bit words, respectively.
    data4 [ 0x40000000 ]
    data8 [-0x0123456789abcdef]

    ## It should be noted that starting a program with a data block is generally
    ## a bad  idea, as our VM  executes programs from the  top. (Incidentally, 0
    ## means that  the VM  should terminate  immediately. Thus,  the meaningless
    ## statements below do not cause the machine to crash.)

    ## A space statement inserts a 64-bit pointer to memory allocated by the
    ## program at startup. The argument specifies the number of bytes that will
    ## be allocated.
my_1000_byte_array:
    space 1000

    ## The  remaining  statements  correspond to  actual  machine  instructions.
    ## However, there is not a  one-to-one correspondence. The assembly language
    ## also contain  several "pseudo-instructions" that translate  into multiple
    ## native instructions.  Moreover, the assembler will  handle some technical
    ## issues such as choosing between long and short conditional jumps.


### 2. PUSH

    ## The push statement pushes 64-bit numbers onto the stack.
    push! 13    # Push the number 13 onto the stack.

    ## As above, these numbers can be in either decimal, octal or hexadecimal
    ## notation; and they can be both positive and negative, with wrapping.
    push! -1    # Push 0xffffffffffffffff onto the stack.

    ## A single push statement can push multiple numbers, indicated by the
    ## number of exclamation marks (!).
    push!! 0 1  # Push 0 onto the stack, then 1.
    push        # Do nothing

    ## The push statement is not only used to push constants.
    push! my_label      # Push the address of my_label.
    push! prime_number  # Push the result of expanding its definition.

    n = 7      # An arbitrary number
    push! &n   # Push the address PC + n * 8 (of the n'th element on the stack).
    push! $n   # Push the (64-bit) value at PC + n * 8.

    ## It is also possible to push the value of complex (Lisp style)
    ## expressions.
    push! (+ my_label -$0) # Push the label address minus the top stack element.

    ## This will be explained in detail later, but observe that
    #
    ##    'push!! 17 $0'    is equivalent to    'push! 17 push! $1'
    #
```

```
    ## since within a statement $x and &x will be relative to value of the stack
    ## pointer at the start of the statement.

    ## The following notation is more convenient when pushing many values onto
    ## the stack:
    push* [1 2 3 4 5 6 7]

    ## This is syntactic sugar for:
    push!!!!!!! 1 2 3 4 5 6 7


### 3. JUMP, JUMP_ZERO, JUMP_NOT_ZERO

    ## The jump statement changes the program counter to an address popped from
    ## the stack.
    jump

    ## 'jump! x' is (syntactic) sugar for 'push! x jump'.
    jump! my_label  # Jump to my_label

    ## The jump_zero statement pops an address first, then a value, and jumps to
    ## the address if the value is zero.
    jump_zero

    ## 'jump_zero! x' is sugar for 'push! x jump_zero'.
    jump_zero! my_label  # Jump to my_label if value popped is zero.

    ## Similarly, 'jump_zero!! x y' is sugar for 'push!! x y jump_zero'.
    ## The following statement jumps to my_label if the fourth element on the
    ## stack (counting from 0) is equal to prime_number.
    jump_zero!! (+ prime_number -$4) my_label

    ## The jump_not_zero statement is similar.
    jump_not_zero


### 4. CALL, RETURN

    ## When calling a subroutine we want to continue execution at the next statement.
    call! my_label           # Sugar for 'push! <fresh> jump! my_label <fresh>:'
    return                   # Alias for 'jump'


### 5. LOAD, SIGX

    ## The four load statements all pop one address from the stack and push a 64
    ## bit value,  the contents of  the memory  at that location  (and onwards).
    ## When less than 8 bytes are  fetched, the value is considered unsigned and
    ## is 0-padded before  pushing. We use little-endian  encoding of multi-byte
    ## values.
    load1  # Pop address and push the unsigned byte at that memory location.
    load2  # ... unsigned 16-bit word ...
    load4  # ... unsigned 32-bit word ...
    load8  # Pop address and push the 64-bit word at that memory location.

    ## 'load1! x' is sugar for 'push! x load1', similarly for load2/4/8.
    load4! my_label  # Load the unsigned 32-bit word at my_label.

    ## If you want the signed value at a memory location instead, follow the load
    ## statement with a corresponding sigx statement (sigx1, sigx2 or sigx4).
    #
    ## sigx1 performs "sign extension" from 8 to 64 bits. More precisely, it
    ## (1) pops a 64-bit value from the stack,
    ## (2) sets bits 8..63 (counting from 0) to the same state as bit 7,
    ## (3) pushes the result back onto the stack.
    #
    ## sigx2 and sigx4 are similar.
    sigx4  # Sign extension from 32 to 64 bits.

    ## For completeness, we also include the no-op sigx8, and let 'sigxN! x' be
    ## sugar for 'push! x sigxN'.
    sigx8
    sigx1! 0xff  # Push -1.
```

```
### 6. STORE

    ## The four store operations pop an address A, then a value V from the stack
    ## and writes the least significant bytes of V to A.
    store1                      # Write the least significant byte of V to A.
    store2                      # Write 16 bits of V to A.
    store4                      # Write 32 bits of V to A.
    store8                      # Write all ov V to A.

    ## 'storeN! x' is sugar for 'push! x storeN'
    store4! my_label  # Pop value and write lower 32 bits to memory at my_label.

    ## 'storeN!! x y' is sugar for 'push!! x y storeN'
    store8!! prime_number my_label  # Write prime_number to memory at my_label.


### 7. ARITHMETIC OPERATIONS

    xx = 99
    yy = -13

    add           # Pop two values and push their sum (with wrapping).

    ## All arithmetic statements have corresponding "sugared" variants.
    add!  xx      # Sugar for 'push! xx add'.
    add!! xx yy   # Sugar for 'push!! xx yy add'.

    sub           # Pop x, then y, and push y - x.  Notice the order!
    sub! xx       # Subtract xx from the value on top of the stack.
    sub!! xx yy   # Push xx - yy.

    mult          # Pop two values and push their product.
    neg           # Pop value and push its additive inverse (two's complement).

    div_u         # Pop x, then y, and push y / x using unsigned division.
    div_s         # Similar, but using signed division.

    rem_u         # Pop two values and push the remainder of unsigned division.
    rem_s         # Similar, but using signed division.


### 8. BITWISE OPERATIONS

    and           # Pop two (64-bit) values and push their binary "and".

    ## All bitwise statements have corresponding "sugared" variants.
    and! 0x7f                 # Sugar for 'push! 0x7f and'.
    and!! 0xfff prime_number  # Sugar for 'push!! 0xfff prime_number'.

    or          # Pop two values and push their binary "or".
    xor         # Pop two values and push their binary "xor".
    not         # Pop one value and push its binary negation (one's complement).

    ## Notice the distinction between 'not' (flipping all bits) and 'neg' (which
    ## leaves 0 unchanged). These names are standard in assembly language.

    ## The pow2 statement pops x (unsigned) and pushes 2 to the power of x.
    pow2
    shift_l       # Sugar for 'pow2 mult'
    shift_ru      # Sugar for 'pow2 div_u'
    shift_rs      # Almost sugar for 'pow2 div_s', except handle the special
                  # case of shifting the least negative value 63 bits to the
                  # right. (It should be -1.)


### 9. COMPARISON

    ## For convenience, the comparison predicates below all return 0 for false
    ## and -1 (all bits set) for true. They also come in sugared variants.

    eq           # Pop two values. Push -1 if they are equal, otherwise push 0.
    eq! 7        # Pop value and compare it to 7.
```

```
    eq!! xx yy   # Sugar for 'push!! xx yy eq'.

    ## The remaining comparison operators come in signed and unsigned variants.
    lt_u         # Pop x, then y. Push -1 if y < x (unsigned), otherwise push 0.
    lt_s         # Pop x, then y. Push -1 if y < x (signed), otherwise push 0.
    lte_u        # less than or equal (unsigned)
    lte_s        # less than or equal (signed)

    gt_u         # y > x (unsigned)
    gt_s         # y > x (signed)
    gte_u        # greater than or equal (unsigned)
    gte_s        # greater than or equal (signed)


### 10. IO STATEMENTS

    ## To be determined later.


### 11. SET_SP, EXIT

    set_sp      # Pop A from the stack, and set the stack pointer to A.
    set_sp! 9   # Sugar for 'push! 9 set_sp'.

    exit        # Terminate the machine.

### EXPECTED STACK:
###
```

Basic assembly

```
### iVM assembly language introduction.
###
### Part 2 - Basics
###
### This is part 2 of the iVM assembly language introduction. Unlike in part 1,
### the statements below should actually make some sense.


### 1. THE STACK

    ## When the machine  starts there are two segments of  allocated memory. The
    ## first segment  contains the program.  The program counter  (PC) initially
    ## points to the  start of this segment. The second  segment simply contains
    ## 24 bytes (3  * 64 bits). The  stack pointer (SP) initially  points to the
    ## the first byte  _after_ this segment. In other words,  we have an initial
    ## stack size of 3, which is just  enough for the program itself to create a
    ## fresh stack. Since  this is something every non-trivial  program must do,
    ## the necessary code  is prepended to every executable  binary. The default
    ## stack size is 64 KiB, but it  can be changed in the project file (.proj).

    ## It is often necessary push a copy of an element in the stack onto the
    ## stack. This can be done as follows:
    push!!!! 13 12 11 10    # Push 4 numbers onto the stack (from left to right).
    push!! $0 $3   # Push copies of the stack elements 0 and 3 (counting from 0).

    ## Be careful to use the right number of exclamation marks.
    ## Otherwise, the parser gets very confused.
    ## Now the stack is (13, 10, 10, 11, 12, 13) from the top
    ## since the second statement is sugar for:
    push! $0
    push! $4                    # Notice the offset.

    ## Stack: (11, 13, 13, 10, 10, 11, 12, 13)

    ## The previous statement is sugar for:
    push! (load8 &4)

    ## which is sugar for:
    push! &4
    load8
```

```
    ## In other words, &n is the address of the element n on the stack.
    ## This is e.g. useful when we want to pop elements from the stack:
    set_sp! &10

    ## Now the stack is empty again.


### 2. LABELS

    ## A data segment can be used for shared global variables.
    jump! after_x               # Equivalent to 'jump! (+ x 1)'
x:
    data1 [0]
after_x:
    store1!! (+ 1 (load1 x)) x  # Increase x with 1

    ## If we export x, it can even be accessed from other files.
    EXPORT x

    ## The store statement above is simply (syntactic) sugar for:
    push! (+ 1 (load1 x))
    push! x
    store1

    ## Moreover, the first push statement is sugar for:
    push! 1
    push! (load1 x)
    add

    ## Finally, the second of these statements is sugar for:
    push! x
    load1

    ## Now the byte at x is 2, and the stack is (2, 3) from the top.

    ## Observe  that x  is  not translated  into a  constant  by the  assembler.
    ## Whereas we plan to add an  optimization for the initial program (which is
    ## guaranteed to  be located  at  address 0) the  default mode is  to produce
    ## code which is independent of where it  is located in memory. In fact, the
    ## current virtual  machine currently  randomizes where the  code is  put in
    ## order to detect code relying on absolute addresses.


### 3.

    ## TO BE CONTINUED ...


    exit

### EXPECTED STACK:
### 2
### 3
```

A more advanced example

```
### iVM assembly language introduction.
###
### Part 3 - Advanced
###
### This is part 3 of the iVM assembly language introduction.


### 1. MEMORY MAP WITH NO ENTRY POINT

    ## In most cases you should specify an entry point when calling the
    ## assembler, e.g. "-e main". This situation is explained in section 2
    ## below.

    ## When an  entry point is  _not_ specified,  execution starts at  the first
    ## statement of the first source file.  Moreover, there is code prepended to
    ## the executable which stores  a pointer to the start of the  heap in the 8
```

```
    ## bytes preceding the rest of the code:
program:
    heap_start = (load8 (+ program -8))

    ## Similarly, we also have a pointer to the argument file:
    arg_location = (load8 (+ program -16))

    ## More precisely, arg_location is the address of a memory location holding
    ## the length of the argument file. The file contents follows.
    arg_length = (load8 arg_location)
    arg_start = (+ arg_location 8)
    arg_stop = (+ arg_start arg_length)


### 2. MEMORY MAP WITH ENTRY POINT

    ## If an entry point _was_ specified ("-e main"), then this is where
    ## execution starts, and the stack looks like this:
    ##
    ## 0: return address
    ## 1: start of argument array
    ## 2: length of argument array
    ## 3: start of free heap space

    ## The entry point is essentially called like this:
    push!!! heap_start arg_length arg_start
    call! main
    set_sp! &2
    exit

    ## The top of the end stack is (truncated and) used as the exit status of
    ## the virtual machine (unless it is run with "ivm check"). Hence, the
    ## pointer to the heap start should be overwritten with 0 to signal success.

    ## Observe that the stack and heap share the same memory. In other words,
    ## the stack pointer marks the end of the heap.

main:


### 3. COMPLEX DATA STATEMENTS

    ## Whereas  the  expressions  used  to  initialize  data  segments  must  be
    ## "assembly  time"  constants,  they  may involve  both  abbreviations  and
    ## labels.
    some_constant = 17
    jump! after_data
my_data:
    data4 [ (* some_constant (+ after_data -main)) ]
after_data:
    # Set exit status 119 (17*7).
    store8!! (load4 my_data) &3


### 4. Check binary version compatibility

    ## Ideally, the iVM machine should never change, but it cannot be ruled out.
    ## If this  happens, it is  important to know  if the machine  is compatible
    ## with  the executable  binaries  at  hand. For  this  reason,  it is  best
    ## practice to start assembly files with the following statement:
    check_version

    ## The effect of this statement is to push to the stack the version of the
    ## machine and instruction set for which the binary was assembled and then
    ## call the CHECK instruction to see if the machine can run it. If not, the
    ## machine should signal an error.


### 5.

    ## To be continued...
    return
```

```
### EXPECTED STACK:
###
### 119
```