L1VM - course

L1VM - course 01 - (C) 2023 V 1.1 - Stefan Pietzonke aka koder77 – midnight-coding.de

Hello world!

Here I want to start a little course about programming in my language Brackets. Here we go: “Hello world” in Brackets:

```
// hello.l1com
// Brackets - Hello world!
//
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 x 23)
    (set int64 1 y 42)
    (set int64 1 a 0)
    (set string 13 hello "Hello world!")
    // print string
    print_s (hello)
    print_n
    ((x y *) a =)
    print_i (a)
    print_n
    exit (zero)
(funcend)
```

Here is the output:

```
$ l1vm prog/hello -q
Hello world!
966
```

So the string “hello” with the value “Hello world!” gets printed out. And the result of the calculation: “((x y *) a =)” is printed by “print_i (a)”.

The variables are declared by the “set” statement above.

if if+ else endif

Here is a program which uses the if and if+ statements. If you want to use “else” then you have to use “if+”!

```
// if-4.l1com
//
// if+, else, endif demo
(main func)
    (set int64 1 zero 0)
    (set int64 1 one 1)
    // change "x" and look at the output printed
    (set int64 1 x 6)
    //
```

```
    //
    (set int64 1 y 10)
    (set int64 1 z 5)
    (set int64 1 twen 20)
    (set int64 1 f 0)
    (set string s less_str "x < 10")
    (set string s more_str "x => 10")
    (set string s more_less_20_str " x <= 20")
    (set string s five_less_str " x <= 5")
    (set string s five_more_str " x > 5")
    // check if x is less or more ten
    // (optimize-if)
    (((x y <) f =) f if+)
        (6 less_str 0 0 intr0)
        (((x z <=) f =) f if+)
            (6 five_less_str 0 0 intr0)
        (else)
            (6 five_more_str 0 0 intr0)
        (endif)
    (else)
        (6 more_str 0 0 intr0)
        (((x twen <=) f =) f if)
            (6 more_less_20_str 0 0 intr0)
        (endif)
    (endif)
    (7 0 0 0 intr0)
    (255 0 0 0 intr0)
(funcend)
```

Note here: “(6 five_less_str 0 0 intr0)” is the print string interrupt. And the “(7 0 0 0 intr0)” means print a new line. In later programs I use “print_s” and “print_n” interrupt wrapper functions.

And here is the output:

```
$ l1vm prog/if-4 -q
x < 10 x > 5
```

Here we have some nested “if+” and “if” statements. They are comparisons to the “x” variable. Feel free to set “x” to a different value. And watch the output changes! Set the value of “x” to “11”:

```
$ l1vm prog/if-4 -q
x => 10 x <= 20
```

L1VM - course 02

switch

In the first part of the course you did learn how to write a simple “Hello world!” program. And what the “if” and “if+” statement can do.

But what if you have to do multiple checks if a variable has some value and do something? In this case you can use the “switch” statement:

```
(switch)
    (y 23_const ?)
        print_s (23_str)
        print_n
        (break)
    (y 42_const ?)
        print_s (42_str)
        print_n
        (break)
(switchend)
```

The “?” stands for branch if equal. So if “y” is of value “23_const” the string “23_str” is printed. Note: the “(break)” statement does not “break” the “switch” statement. It Is just like an “endif”. So if the first switch statement is true then it continues with the “(y 42_const ?)” comparison. And there is no “default” statement like in C or C++ for example!

Here is a full example:

```
// switch.l1com
// Brackets - Hello world! switch
//
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 x 23)
    (set int64 1 y 42)
    (set string s 23_str "y = 23")
    (set string s 42_str "y = 42")
    (set const-int64 1 23_const 23)
    (set const-int64 1 42_const 42)
    (set string s hello_str "Hello world!")
    (set int64 1 a 0)
    // print string
    print_s (hello_str)
    print_n
    ((x y *) a =)
    print_i (a)
    print_n
    (switch)
        (y 23_const ?)
            print_s (23_str)
            print_n
            (break)
        (y 42_const ?)
            print_s (42_str)
            print_n
            (break)
    (switchend)
    exit (zero)
(funcend)
```

Here is the output:

```
$ l1vm prog/switch -q
Hello world!
966
y = 42
```

In this program “intr.l1h” includes the macros for the interrupt functions. So you can use “print_i” and “print_s” for example instead of the interrupt statement with the number codes.

L1VM - course 03

math - old way

In this part I will write about the different ways to do math in Brackets. Here is some code in the older way:

```
((x y +) z =)
(((a b +)(c e *) +) z =)
```

The second line has the two calculations added into the “z” variable! There is two other ways to write math expressions:

math - infix new

```
{z = (x + y)}
{z = (a + b) + (c * e)}
```

Note: you have to use the curly brackets around the expressions: “{}”. And must use at least one normal bracket inside:

WRONG!:

```
{z = a + b + c}
```

RIGHT!:

```
{z = (a + b + c)}
```

math - RPN

This is RPN notation math: reversed polish notation. The operator is after the variables:

```
{z = x y +}
{z = a b + c e * +}
```

This looks like the older way but needs no brackets “()”!

L1VM - course 04

functions

Here I show a simple math function calculating the square of a number. We will use double floating point numbers.

A double number is defined by a “set” call like this:

```
(set double 1 a 2.0)
```

The “1” stands for one number. We can define an array by setting a higher number as “1”. But arrays are a topic for a later course part.

Here is our “square” function:

```
(square func)
    (set double 1 num 0.0)
    (set double 1 square 0.0)
    (num stpopd)
    {square = num num *}
    (square stpushd)
(funcend)
```

It defines two variables: “num” and “square”. With “stpopd” we get the number from the stack. The stack is needed to push and pull variables to and from it. With: ” {square = num num *}” we calculate the square of the number. As the last step the “square” variable is pushed on to the stack. So we can get the result back in the main function.

The call looks like this:

```
 // call square function with numbers a, b and c:
    (a :square !)
    (as stpopd)
    print_d (as)
    print_n
```

The “print_d” and “print_n” are macros defined in the include file: “intr.l1h”. With “print_d” a double number is printed out. The “print_n” just puts a new line out. So you can use them for a formatted text output.

In the variable “as” we get the square of the number “a” back.

Here is the full program:

```
// square-func.l1com
//
// calculate square of number
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set double 1 a 2.0)
    (set double 1 b 13.5)
    (set double 1 c 7.8)
    (set double 1 as 0.0)
    (set double 1 bs 0.0)
    (set double 1 cs 0.0)

    // call square function with numbers a, b and c:
    (a :square !)
    (as stpopd)
    print_d (as)
    print_n
```

```
    (b :square !)
    (bs stpopd)
    print_d (bs)
    print_n

    (c :square !)
    (cs stpopd)
    print_d (cs)
    print_n

    exit (zero)
(funcend)

(square func)
    (set double 1 num 0.0)
    (set double 1 square 0.0)
    (num stpopd)
    {square = num num *}
    (square stpushd)
(funcend)
```

Here is the program output:

```
$ l1vm prog/square-func -q
4.0000000000
182.2500000000
60.8400000000
```

L1VM - course 05

input

Here I will show how to get input on the console.

There are three input commands:

```
input_i (number)
input_d (double_number)
input_s (string)
```

The variable inside the brackets stores the input typed in by the user. You have to take care to use the right “input” command for your input type. The input is taken after you press the “RETURN” key!

Here is a program which calculates the diameter or circumference of a circle. It uses “input_i” and “input_d” commands:

```
// math-circle.l1com
// calculate diameter or circumference of circle
//
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 one 1)
    (set int64 1 two 2)
    (set int64 1 three 3)
```

```
    (set double 1 diam 0.0)
    (set double 1 circ 0.0)
    (set double 1 zerod 0.0)
    (set string s menu_diamstr "1: calculate diameter of circle")
    (set string s menu_circstr "2: calculate circumference of circle")
    (set string s menu_quitstr "3: quit")
    (set string s menu_chstr "? ")
    (set string s diamstr "diameter:      ")
    (set string s circstr "circumference: ")
    (set int64 1 input 0)
    (set int64 1 f 0)
    // set constant ----------------------------------------
    (set const-double 1 m_pi@math 3.14159265358979323846)
    // ----------------------------------------------------
    (:loop)
    print_s (menu_diamstr)
    print_n
    print_s (menu_circstr)
    print_n
    print_s (menu_quitstr)
    print_n
    print_s (menu_chstr)
    // read input
    input_i (input)
    (((input three ==) f =) f if)
        // quit
        exit (zero)
    (endif)
    (((input one ==) f =) f if+)
        // reset used variables and set used registers to zero
        // no context register saving here
        (reset-reg)
        (:calc_diam !)
        (loadreg)
    (else)
        // reset used variables and set used registers to zero
        // no context register saving here
        (reset-reg)
        (:calc_circ !)
        (loadreg)
    (endif)
    print_n
    print_n
    (:loop jmp)
(funcend)
(calc_diam func)
    // calculate diameter
    (zerod circ =)
    (zerod diam =)
    print_s (circstr)
    // input double circ
    input_d (circ)
    ((circ m_pi@math /d) diam =)
    print_s (diamstr)
```

```
    print_d (diam)
(funcend)
(calc_circ func)
    // calculate circumference
    (zerod diam =)
    (zerod circ =)
    print_s (diamstr)
    // input double diam
    input_d (diam)
    ((diam m_pi@math *d) circ =)
    print_s (circstr)
    print_d (circ)
(funcend)
```

Inside the “calc_diam” function: “((circ m_pi@math /d) diam =)” the “/d” means divide a double number. Inside the “calc_circ” function: “*d” means multiply a double number.

L1VM - course 06

strings

Now I will tell you how to use strings in Brackets. Here is a “set” definition of a string:

```
(set string s hellostr "Hello world!")
```

Note: the “s” stands for set the string size automatically to the text length! You could define it this way too:

```
(set string 13 hellostr "Hello world!")
```

The string is 12 chars long. But we need one char space more to store the binary zero as the string end mark! You should always use the “s” size setting if the string is not changed in the program.

If you need to add chars at the end of the string then you have to increase the size like this:

```
(set string 256 hellostr "Hello world!")
```

So now there is space for 255 chars in the string “hellostr”!

You can copy a string to a new one:

```
(set string s hellostr "Hello world!)
(set string 256 newstr "")

(newstr hellostr :string_copy !)
```

This copies the string “hellostr” to the string “newstr”.

To add a string to a existing one we can use: “string_cat”:

```
(set string 256 hellostr "Hello")
(set string s worldstr " world!")

(hellostr worldstr :string_cat !)
```

Now “hellostr” contains: “Hello world!”.

L1VM - course 07

loops

In this part of the course I show how to use loops in Brackets.

for loop

```
// hello-for - Brackets - Hello world! do loop
//
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 one 1)
    (set int64 1 loop 0)
    (set int64 1 maxloop 10)
    (set int64 1 f 0)
    // set string length automatic, by setting "s"
    (set string s hello "Hello world!")
    // print string
    // for
    (zero loop =)
    (for-loop)
    (((loop maxloop <) f =) f for)
        print_s (hello)
        // print newline
        print_n
        ((loop one +) loop =)
    (next)
    exit (zero)
(funcend)
```

Here the include uses the “intr.l1h” include file. So we can use “print_s” and “print_n” macros. The loop continues running until “loop” is less than “maxloop”. In the line:

```
((loop one +) loop =)
```

the variable “loop” is increased by one. For loops can be used if you know how many times the loop should be run. Here is the output:

```
$ l1vm prog/hello-for -q
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

You can nest the for loops in each other to do more complex stuff!

do while loop

The do while loop is ran at least once. The check is done at the end of the loop:

```
// hello-while - Brackets - Hello world! do loop
//
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 one 1)
    (set int64 1 loop 0)
    (set int64 1 maxloop 10)
    (set int64 1 f 0)
    // set string length automatic, by setting "s"
    (set string s hello "Hello world!")
    // print string
    // while
    (zero loop =)
    (do)
        print_s (hello)
        // print newline
        print_n
        ((loop one +) loop =)
    (((loop maxloop <) f =) f while)
    exit (zero)
(funcend)
```

In the lines:

```
((loop one +) loop =)
    (((loop maxloop <) f =) f while)
```

“loop” is increased by one. And is compared to “maxloop”. If “loop” is less than “maxloop” then the “do while” loop is ran.

Here is the output:

```
$ l1vm prog/hello-while -q
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

The “do while” loops can also be nested into more complex loops.

L1VM - course 08

arrays

Here I will show some array handling code.

This defines an int64 (64 bit int number) array with 10 elements:

```
(set int64 10 array 10 5 8 4 3 2 7 23 45 30)
```

To access the array elements we need an index and an offset:

```
((ind offset *) realind =)
(array [ realind ] num =)
```

The first line calculates the offset and stores the result in “realind”. We are accessing an int64 number in the array: the offset is “8”! On a byte array this offset would be “1” and we would not need it! This offset is needed because internally every variable is stored in a simple byte array! You can see this by looking at an assembly file generated by the compiler (.l1asm).

The VM can find illegal array accesses and breaks the program execution. You will get an error message.

If we set the max number of elements to read to “11” then we get this error message:

```
l1vm-work]$ l1vm prog/array-demo -q
10
5
8
4
3
2
7
23
45
30
memory_bounds: FATAL ERROR: variable not found overflow address: 56, offset:
80!
epos: 231
```

In the assembly file we find address “56” in line 16: “@, 56Q”.

```
Q, 10, array
@, 56Q, 10, 5, 8, 4, 3, 2, 7, 23, 45, 30, ;
```

On the address “56” the array “array” is stored! Here the offset is out of range!

The “epos” is the execution position of the current opcode. We can find it in the “out.l1dbg” file:

In line “19”:

```
epos: 231, 41 line num
```

This shows the line in the assembly source code: lines 38 - 41:

```
loada num, 0, 9
loada realind, 0, 10
load array, 0, 11
pushqw 11, 10, 9
```

So here we see the opcode in line 41:

```
pushqw 11, 10, 9
```

In line 40 the array address is load into the register “11” and the “realind” variable into register “10”. So we found the line with the error in it! This is not easy to find but it is possible. You can also take a look at the assembly file output of the Brackets compiler to learn more about the assembly code!

Here is the full array demo program:

```
// array-demo.l1com
// array with values demo
// shows how to set up an array
// and how to access the elements
//
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 one 1)
    (set const-int64 1 offset 8)
    (set int64 1 ind 0)
    (set int64 1 f 0)
    (set int64 1 num 0)
    (set int64 1 realind 0)
    // set array data using spaces as elements separator
    (set int64 10 array 10 5 8 4 3 2 7 23 45 30)
    (set int64 1 maxarray 10)
    // print array
    (for-loop)
    (((ind maxarray <) f =) f for)
        ((ind offset *) realind =)
        (array [ realind ] num =)
        print_i (num)
        print_n
        ((ind one +) ind =)
    (next)
    exit (zero)
(funcend)
```

Now we get this output:

```
$ l1vm prog/array-demo -q
10
5
8
4
3
2
7
23
45
30
```

And everything runs just fine!

To store a number in the array, we can do:

```
(num array [ realind ] =)
```

L1VM - course 09

logical operators, bit shifting

logical operators

Here I will show how to use the logical operators in Brackets. They work like in C or other languages.

The “||” or operator:

```
 ((((x a ==)(x b ==) ||) f =) f if)
```

The if will be executed if “x” equals “a” or “x” equals “b”. If you want to do an if when at least two cases are true then you use the “&&” and operator:

```
((((x a ==)(y b ==) &&) f =) f if)
```

Here the if will be executed if “x” equals “a” and “y” equals “b”. If only one of the cases is true it will not be executed!

You can also write:

```
{i = (x == a) && (y == b)}
(i if)
    // code inside of if
```

bit shifting

```
{b = (a <| shift)}
```

Here the “a” variable is “shifted” “shift” times to the left. If “a” is “2” and “shift” is “1” then we will get the value “4”. The bit pattern is shifted “1” bit to the left so the number is increasing. If the “shift” is “5” then we will get “64”.

The “>|” right shift operator works the other way round. So the result is the bit pattern shifted to the right by “shift” bits. The value will decrease.

Here is a full example:

```
// math-test.l1com - Brackets - new math expression test
//
//
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 xd 23)
    (set int64 1 yd 42)
    (set int64 1 zd 0)
    (set int64 1 zerod 0)
    (set int64 1 i 0)
```

```
    (set int64 1 f 0)
    (set int64 1 a 2)
    (set int64 1 b 0)
    (set int64 1 shift 5)
    (set string s messagestr "xd < yd and xd > 0")
    // new math expression:
    {i = (xd < yd) && (xd > zerod)}
    print_i (i)
    print_n
    (i if)
        print_s (messagestr)
        print_n
    (endif)
    ((((xd yd <)(xd zerod >) &&) f =) f if)
        print_s (messagestr)
        print_n
    (endif)
    // shift
    {b = (a <| shift)}
    print_i (b)
    print_n
    {b = (b >| shift)}
    print_i (b)
    print_n
    (255 zero 0 0 intr0)
(funcend)
```

L1VM - cli arguments

Here I will show how to get the cli arguments in Brackets.

```
shell_args (args)
```

Here the number of shell arguments is stored in the variable “args”. Zero means there is no argument.

```
get_shell_arg (i, shell_arg)
```

Here the argument “i” will be stored in the string variable “shell_arg”. If there are two arguments then you would use “0” and “1” as values for “i”!

Here is an example cli call:

```
$ l1vm prog/shell-args -args foo bar foobar
```

You have to add the arguments behind the “-args” flag. Without this flag it won’t work!

Here is a full example:

```
// show shell arguments
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 one 1)
    (set int64 1 args 0)
    (set string 255 shell_arg "")
    (set int64 1 i 0)
    (set int64 1 f 0)
```

```
    (set string s argstr " shell arguments: ")
    // get number of shell arguments:
    shell_args (args)
    print_i (args)
    print_s (argstr)
    print_n
    print_n
    (((args zero ==) f =) f if)
    // no arguments, exit!
        (255 0 0 0 intr0)
    (endif)
    (zero i =)
    (:loop)
    // get shell argument in variable "shell_arg"
    get_shell_arg (i, shell_arg)
    print_s (shell_arg)
    print_n
    ((i one +) i =)
    (((i args <) f =) f if)
        (:loop jmp)
    (endif)
    exit (zero)
(funcend)
$ l1vm prog/shell-args -q -args foo bar foobar
3 shell arguments:

foo
bar
foobar
```

L1VM - preprocessor

Here I will show how to use the L1VM preprocessor to include headers and defining macros.

To include the “intr.l1h” interrupt macro header we use:

```
#include <intr.l1h>
```

This must be included at the top of a program before the main function. Note: the modules includes are included at the bottom of a program.

There are three types of preprocessor macros:

```
#define
#func
#var
```

With “#define” we can set a replacement text for a macro. Here is an example out of “intr.l1h” (this shows the stack pointer):

```
#define show_st_p (14 0 0 0 intr0)
```

The preprocessor replaces every text with “show_st_p” by the text “(14 0 0 0 intr0)”. So you can write the macro name in your program without need to know the interrupt “0” number “14”!

The “#func” macro defines a function with variables that can be used to call the function:

```
#func get_time (THOUR, TMIN, TSEC) :(17 THOUR TMIN TSEC intr0)
```

This is the “get_time” macro it has three arguments: “THOUR”, “TMIN”, and “TSEC”. The replacement part begins after the “:” char. Here the macro variables are inserted into: “(17 THOUR TMIN TSEC intr0)”.

The macro can be called with something like this:

```
get_time (hour, min, sec)
```

The three variables can be of type int for example.

Here is a more advanced function:

```
#func inside_coord (MX, MY, X, Y, X2, Y2) :{f~ = ((MX > X) && (MX < X2)) &&
((MY > Y) && (MY < Y2))}
```

To use this you just do:

```
inside_coord (mx, my, x, y, x2, y2)
```

Here “mx/my” is the coordinate to check. If the two coordinate variables “mx” and “my” are in the range of “x”, “x2” and “y”, “y2” then the variable “f~” is set to one. So “f~” can be checked with an “if”.

Here is a full example:

```
// coord-xy.l1com
// preprocessor function demo
//
#include <intr.l1h>

#func inside_coord (MX, MY, X, Y, X2, Y2) :{f~ = ((MX > X) && (MX < X2)) &&
((MY > Y) && (MY < Y2))}

(main func)
    (set int64 1 zero 0)
    (set int64 1 one 1)
    (set int64 1 x 100)
    (set int64 1 x2 200)
    (set int64 1 y 50)
    (set int64 1 y2 100)
    (set int64 1 mx 110)
    (set int64 1 my 60)
    (set int64 1 f~ 0)
    (set string s insidestr "m position inside!")
    (set string s outsidestr "m position outside!")
    inside_coord (mx, my, x, y, x2, y2)
    (f~ if+)
        print_s (insidestr)
        print_n
    (else)
        print_s (outsidestr)
        print_n
    (endif)
    exit (zero)
(funcend)
```

The “#var” macro defines the function name ending on variables. Here is a function taken from my “pov-edit” POV editor:

```
(init_paint_rect func)
    #var ~ draw_paint_rect
    (set const-int64 1 zero~ 0)
    (set const-int64 1 one~ 1)
    (set int64 1 x_start~ 10)
    (set int64 1 y_start~ 10)
    (set const-int64 1 x_width~ 50)
    (set const-int64 1 y_height~ 8)
    (set const-int64 1 tile_width~ 18)
    (set int64 1 x_tiles~ 0)
    (set int64 1 y_tiles~ 0)
    (set byte 1 color~ 0)
    (set byte 1 alpha~ 255)
    (zero~ zero~ x_start~ y_start~ x_width~ y_height~ tile_width~ tile_width~
color~ color~ color~
    alpha~ one~ :set_gadget_box_grid call)
(funcend)
```

So here “#var ~ draw_paint_rect” sets the variable name ending to “draw_paint_rect”. So the variable “zero~” will be expanded to “zerodraw_paint_rect”.

L1VM - assembly VOL I

Here I will show how to read the assembly output of my compiler Brackets. A program can be build in the l1vm directory:

```
./build.sh prog/hello
```

This creates the byte code in “prog/hello.l1obj” and in the root directory: “out.l1asm” and “out.l1dbg”. Here is “prog/hello.l1com” in Brackets:

```
// hello.l1com
// Brackets - Hello world!
//
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 x 23)
    (set int64 1 y 42)
    (set int64 1 a 0)
    (set string 13 hello "Hello world!")
    // print string
    print_s (hello)
    print_n
    ((x y *) a =)
    print_i (a)
    print_n
    exit (zero)
(funcend)
```

And here is “out.l1asm”:

```
.data
Q, 1, zero
@, 0Q, 0
Q, 1, x
@, 8Q, 23
Q, 1, y
@, 16Q, 42
Q, 1, a
@, 24Q, 0
B, 13, hello
@, 32Q, "Hello world!"
Q, 1, helloaddr
@, 45Q, 32Q
.dend
.code
:main
loada zero, 0, 0
loada helloaddr, 0, 1
intr0 6, 1, 0, 0
intr0 7, 0, 0, 0
loada x, 0, 2
loada y, 0, 3
muli 2, 3, 4
load a, 0, 5
pullqw 4, 5, 0
loada a, 0, 6
intr0 4, 6, 0, 0
intr0 7, 0, 0, 0
intr0 255, 0, 0, 0
rts
.cend
```

The first line begins the “.data” section with all variable declarations. The second and third line declare the variable “zero”:

```
Q, 1, zero
```

This sets “Q” for “int64” type, “1” for one variable and “zero” as the variable name.

```
@, 0Q, 0
```

Now here we have the memory address “0Q” and the value “0”.

And on line four and five we have:

```
Q, 1, x
```

This sets “Q” for “int64” type, “1” for one variable and “x” as the variable name.

```
@, 8Q, 23
```

Here we get the memory address “8Q” and the value “23”. So the memory address is increased by 8 byte to store the variable “x”. The compiler sets the right memory addresses for us, so have no worries about this!

The string declaration is different:

```
B, 13, hello
```

Here we have type “B” for byte or string. And “13” chars. The name is “hello”.

```
@, 32Q, "Hello world!"
```

The address is “32Q” and the value “Hello world!”. However here are two more lines:

```
Q, 1, helloaddr
```

This defines a “int64” type variable “helloaddr” storing the memory address of the string “hello”:

```
@, 45Q, 32Q
```

The memory address “32Q” is stored at location “45Q”. Internally a string variable is referenced by its address. This is done automatically.

Lets have a look at the multiplication code:

Brackets:

```
((x y *) a =)
```

Assembly:

```
loada x, 0, 2
loada y, 0, 3
muli 2, 3, 4
load a, 0, 5
pullqw 4, 5, 0
```

Here the “loada” opcode loads the “int64” variable “x” into register “2”. Then the next “loada” opcode loads the “int64” variable “y” into register “3”.

Now the “muli” opcode just multiplies the registers “2” and “3” and stores the result in the target register “4”. Now the compiler needs to store the register back into the variable “a”. This is done by load the address of “a” into register “5”:

```
load a, 0, 5
```

Now the register “4” is pulled into address “5” (a) by the “pullqw” opcode:

```
pullqw 4, 5, 0
```

I hope this posting could tell you more about what is going on in my compiler.

L1VM - assembly VOL II

Here I will show how to read more advanced assembly expressions.

This is a part of my “prog/hello-math.l1com” program:

```
// hello-math.l1com - Brackets - Hello world
//
// RPN math and infix math expressions
//
(main func)
    (set int64 1 zero 0)
    (set int64 1 x 23)
    (set int64 1 y 42)
```

```
    (set int64 1 z 13)
    (set int64 1 foo 23)
    (set int64 1 ret 0)
    (set string 13 hello "Hello world!")
    // print string
    (6 hello 0 0 intr0)
    (7 0 0 0 intr0)
    // reversed polish notation (RPN):
    {ret = x y + z x * *}
    (4 ret 0 0 intr0)
    (7 0 0 0 intr0)
```

Lets take a look at:

```
{ret = x y + z x * *}
```

This is a math expression in RPN (reversed polish notation) with the operands after the numbers.

The assembly output is this:

```
.data
Q, 1, zero
@, 0Q, 0
Q, 1, x
@, 8Q, 23
Q, 1, y
@, 16Q, 42
Q, 1, z
@, 24Q, 13
Q, 1, foo
@, 32Q, 23
Q, 1, ret
@, 40Q, 0
B, 13, hello
@, 48Q, "Hello world!"
Q, 1, helloaddr
@, 61Q, 48Q
.dend
.code
:main
loada zero, 0, 0
loada helloaddr, 0, 1
intr0 6, 1, 0, 0
intr0 7, 0, 0, 0
loada x, 0, 2
loada y, 0, 3
addi 2, 3, 4
loada z, 0, 5
muli 5, 2, 6
muli 4, 6, 7
load ret, 0, 8
pullqw 7, 8, 0
loada ret, 0, 7
intr0 4, 7, 0, 0
intr0 7, 0, 0, 0
```

The “x” variable is load into register “2”. The “y” variable is load into register “3”:

```
loada x, 0, 2
loada y, 0, 3
```

Now “x” and “y” are added and the result is stored into register “4”:

```
addi 2, 3, 4
```

Then the variable “z” is load into register “5”:

```
loada z, 0, 5
```

The next code does the multiplications. The variable “z” is multiplied by “x” and the result is stored in register “6”:

```
muli 5, 2, 6
```

The next multiplication multiplies the register “4” by register “6” and stores the result in register “7”:

```
muli 4, 6, 7
```

The final step is to store the result into variable “ret”:

```
load ret, 0, 8
pullqw 7, 8, 0
```

L1VM - assembly VOL III

Here I will show how function calls are done in assembly.

Here is the Brackets program:

```
// square-func.l1com
//
// calculate square of number
//
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set double 1 a 2.0)
    (set double 1 b 13.5)
    (set double 1 c 7.8)
    (set double 1 as 0.0)
    (set double 1 bs 0.0)
    (set double 1 cs 0.0)

    // call square functions with numbers a, b and c:
    (a :square !)
    (as stpopd)
    print_d (as)
    print_n

    (b :square !)
    (bs stpopd)
    print_d (bs)
    print_n

    (c :square !)
```

```
    (cs stpopd)
    print_d (cs)
    print_n

    exit (zero)
(funcend)

(square func)
    (set double 1 num 0.0)
    (set double 1 square 0.0)
    (num stpopd)
    {square = num num *}
    (square stpushd)
(funcend)
```

Lets have a look at:

```
 // call square functions with numbers a, b and c:
    (a :square !)
    (as stpopd)
    print_d (as)
    print_n
```

Here is the assemby output:

```
:main
loada zero, 0, 0
stpushi 0
loadd a, 0, 1
stpushd 1
jsr :square
stpopd 2
load as, 0, 3
pulld 2, 3, 0
stpopi 0
loadd as, 0, 1
intr0 5, 1, 0, 0
intr0 7, 0, 0, 0
```

Here the “loadd a, 0, 1” opcode loads the double variable “a” into register “1”. Then with “stpushd 1” the register “1” is put on the stack. We need this for the function call: “jsr: square”. After the function call we need to get the result from the stack: “stpopd 2”. Then the result is stored into variable “as”:

```
load as, 0, 3
pulld 2, 3, 0
```

Now I will show the function “square”:

```
(square func)
    (set double 1 num 0.0)
    (set double 1 square 0.0)
    (num stpopd)
    {square = num num *}
    (square stpushd)
(funcend)
```

The assembly looks like this:

```
:square
loada zero, 0, 0
stpopd 1
load num, 0, 2
pulld 1, 2, 0
muld 1, 1, 2
load square, 0, 3
pulld 2, 3, 0
loadd square, 0, 2
stpushd 2
rts
```

The “stpopd 1” gets the double number from the stack. The “muld 1, 1, 2” multiplies the number with it self and stores the result in register “2”.

Then the result register is stored into variable “square”:

```
load square, 0, 3
pulld 2, 3, 0
```

The last step pushes the variable “square” to the stack:

```
loadd square, 0, 2
stpushd 2
```

The function jumps back to the caller by:

```
rts
```

L1VM - assembly VOL IV

Here we will create the “square” calc function of VOL III in inline assembly.

Here is the Brackets code:

```
(square func)
    (set double 1 num 0.0)
    (set double 1 square 0.0)
    (num stpopd)
    {square = num num *}
    (square stpushd)
(funcend)
```

To write inline assembly we need the begin and end command:

```
(ASM)

assembly code

(ASM_END)
```

Thirst wee need to pull the variable from stack. Then we do the multiplication on the variable. And as a last step we need to push the result on the stack.

Here we go:

```
(square func)
    (ASM)
    stpopd 1
    muld 1, 1, 2
    stpushd 2
    (ASM_END)
(funcend)
```

L1VM - assembly VOL V

Here we will debug some code:

```
// debug.l1com
#include <intr.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 one 1)
    (set int64 1 offset 8)
    (set int64 10 array 12 10 23 42 5 6 7 8 9 0)
    (set int64 1 maxarray 11)
    (set int64 1 arrayind 0)
    (set int64 1 val 0)
    (set int64 1 i 0)
    (set int64 1 f 0)
    (set string s hellostr "hello")
    (set string s worldstr " world !")
    (set string s commastr "@@c ")

    (zero :string_init call)
    (hellostr worldstr :string_cat !)

    // assign to array
    (for-loop)
    (((i maxarray <) f =) f for)
        (array [ arrayind ] val =)
        print_i (val)
        print_s (commastr)
        ((i one +) i =)
        ((arrayind offset +) arrayind =)
    (next)

    print_n
    exit (zero)
(funcend)
#include <string.l1h>
```

The program is available on my L1VM GitHub repo! To build it just do:

```
./build.sh prog/debug
```

If we run it then we get this error:

```
$ l1vm prog/debug -q
memory_bounds: FATAL ERROR: variable not found overflow address: 144, offset:
13!
string_cat: ERROR: dest string overflow!
```

So there must be something wrong with a string size. We got the address of the string which is “144”. Lets have a look at the “out.l1asm” assembly file and search for “144” in it. We find it at line 21:

```
B, 6, hellostr
@, 144Q, "hello"
Q, 1, hellostraddr
@, 150Q, 144Q
```

Here the string “hellostr” is defined by a size of “6” with place for “5” chars. In the buggy program we tried to cat the string “worldstr” to it. This gives the string overflow error. We have to increase the string size of the string “hellostr”:

```
(set string 14 hellostr "hello")
```

Lets build the program:

```
./build.sh prog/debug
```

And try to run it:

```
$ l1vm prog/debug -q
12, 10, 23, 42, 5, 6, 7, 8, 9, 0, memory_bounds: FATAL ERROR: variable not
found overflow address: 24, offset: 80!
epos: 269
```

Now we get a new error message: here is a memory bounds error of our array variable. Lets search the address “24” in the “out.l1asm” file:

```
Q, 10, array
@, 24Q, 12, 10, 23, 42, 5, 6, 7, 8, 9, 0, ;
```

Here the “24” is in line “9”!

So this is our array “array”. There we did set the 10 array entries. What happened? If we look at the program we can find the error: “maxarray” is “11” but we have only “10” array entries. So we get the variable overflow error.

Lets fix this:

```
(set int64 1 maxarray 10)
```

Rebuild and run it again:

```
$ l1vm prog/debug -q
12, 10, 23, 42, 5, 6, 7, 8, 9, 0,
```

Now all bugs are fixed and it finally runs!

L1VM - course 17

multi threading

Here I want to show how to do multi threading in Brackets. To do this I need to explain something about my language Brackets. You can’t call a function multiple times each one as a new thread. The second thread would overwrite the firsts thread variables! This is because in Brackets there are only global variables. You can set a variable name ending by `#var`:

```
(bar func)
    #var ~ bar

    (set int64 1 foo~ 42)
    print_i (foo~)
    print_n
(funcend)
```

So here we get the full variable name: `foobar`! To use this function multiple times we need to copy the code and replace the variables to another name.

What happens if you launch a new thread? It will load a fresh CPU core with the registers set to zeroes. And the stack will be copied into it. So you can get call variables from the stack.

Here is a full example: It is also available in my `prog` directory on GitHub!

```
// hello-thread-3.l1com
// Brackets - Hello world! threads
//
// This is an exammple how to launch threads using the new compiler opcode
"loadl".
// In this example I used the "intr.l1h" include to use the new interrupt
macros.
//
#include <intr.l1h>
#include <misc-macros.l1h>
(main func)
    (set int64 1 zero 0)
    (set int64 1 one 1)
    (set int64 1 two 2)
    (set int64 1 three 3)
    (set int64 1 four 4)
    (set int64 1 run 0)
    (set int64 1 f 0)
    (set int64 1 delay 4000)
    (set int64 1 cpu_number 0)
    (set string s messagestr "starting...")
    // run the two threads
    get_cpu (cpu_number)
    (((cpu_number zero ==) f =) f if)
        print_s (messagestr)
        print_n
        (:start_thr !)
        detime (delay)
        (one run =)
    (endif)
    exit (zero)
(funcend)
(hello_a func)
    #var ~ @hello_a
    (set string s hellostr~ "Hello world! Thread 1")
    (set int64 1 delay~ 2000)
    print_s (hellostr~)
    print_n
    detime (delay~)
```

```
    threadexit (zero)
(funcend)
(hello_b func)
    #var ~ @hello_b
    (set string s hellostr~ "Hello world! Thread 2")
    (set int64 1 delay~ 2000)
    print_s (hellostr~)
    print_n
    detime (delay~)
    threadexit (zero)
(funcend)
(start_thr func)
    (set int64 1 run_th 0)
    (set int64 1 lab_hello_one 0)
    (set int64 1 lab_hello_two 0)
    (set int64 1 zero_th 0)
    (set int64 1 one_th 1)
    (set int64 1 f_th 0)
    (set int64 1 delay_th 2500)
    (set int64 1 loop 0)
    (set int64 1 maxloop 25)
    (reset-reg)
    (:hello_a lab_hello_one loadl)
    (:hello_b lab_hello_two loadl)
    pull_int64_var (lab_hello_one)
    pull_int64_var (lab_hello_two)
    (((run_th zero_th ==) f_th =) f_th if)
        // run threads
        thread (lab_hello_one)
        thread (lab_hello_two)
        (one_th run_th =)
    (endif)
    detime (delay_th)
    join
    exit (zero)
(funcend)
```