
What is it?

1. A simple stack-based virtual machine that runs C in the browser.
2. An interactive tutorial that covers C, how the VM works, and how the language is compiled, everything from the ground up.

Why?

This project demystifies compilers without getting lost in unnecessary details. MiniC visually explores how a compiler can break down the C language into simple instructions and how those are executed by a virtual CPU.

10000	(FSTART) 1000	main	30000	(RET IP) 0
10001	0		30001	(OLD BP) 0
10002	(PUSH) 1002		30002	0
10003	10012		30003	0
10004	(PUSH) 1002		30004	0
10005	5		30005	0
10006	(CALL) 1007		30006	0
10007	1		30007	0
10008	(RET) 1001		30008	0
10009	0		30009	0
10010	(RET) 1001		30010	0
10011	0		30011	0
10012	(FSTART) 1000		30012	0
10013	0		30013	0
10014	(LEA) 1004		30014	0
10015	-2		30015	0
10016	(DRF) 1011		30016	0
10017	(NOT) 1009		30017	0
10018	(JZ) 1005		30018	0
10019	10026		30019	0
10020	(PUSH) 1002		30020	0
10021	0		30021	0
10022	(RET) 1001		30022	0
10023	1		30023	0
10024	(JMP) 1006		30024	0

Can I see it?

1. Sandbox
2. Tutorial (for people with 0 programming experience or willing to learn C) :

-
- Part 1 - Introduction
 - Part 2 - Expressions (part 1)
 - Part 3 - Expressions (part 2)
 - Part 4 - Variables and program structure

Subscribe

Get notified when new tutorials are released.

Feedback

Join the discussion on our subreddit.

Support

Consider supporting the project.

Documentation

Virtual Instruction Set

Note: BEFORE each instruction, IP increases by 1

PUSH * increases SP by 1 and sets the value at SP to the argument ("Pushes the argument onto the stack")

LEA: * similar to PUSH * instead of pushing just the argument, it pushes BP + argument ("Loads the Effective Address") * used to push the address of a local variable on the stack ("LEA 2" is used to push the address of the second local variable on the stack)

DRF: * replaces the address on the top of the stack with the value at that address ("Dereferences the address") * e.g. to push the value of a local variable, we use LEA followed by DRF

POP * decreases SP by 1 ("pops a value off the stack") * used if the value on the top of the stack is not needed, to prevent the stack from growing indefinitely * e.g. after calling a function without intending to use its returned value

JMP * sets IP to the argument ("jump to the address given as argument") * used to compile control structures such as "if" and "while"

JZ: * pops a value off the stack, and if that value is 0, it jumps to the argument (“jump if zero”) * used to compile control structures such as “if” and “while”

FSTART: * a sequence of simpler instructions performed before each function (“Function START”); * pushes the value of BP effectively saving it before setting BP * sets BP to SP * increase SP by the value of the argument, effectively making room for local variables * effectively sets up the stack so that BP so that BP points to the old BP, BP + 1 to the first local variable, BP + 2 to the second, etc. * before the function returns, BP will be restored to its old value

RET: * a sequence of simpler instructions performed before each function RETURNS * remembers the value on the top of the stack. this is the function return value. * sets SP to BP, effectively popping all the local variables and the returned value * sets BP to the value on the top of the stack, effectively restoring BP to its value before FSTART * performs a POP * sets IP to the value on the top of the stack, effectively setting IP to point to the next instruction after the CALL that created this function * performs a POP * decreases SP by the argument, effectively popping all the arguments pushed before the function was called * replaces the value on the top of the stack (which is the address of the current function) with the function return value remembered earlier all the previous POPs ensure that SP is now set to its original value before calling the function + 1. The top of the stack is the returned value of the function

CALL: a sequence of simpler instructions performed to CALL a function. it assumes that the address of the function followed by all the arguments have already been pushed * Pushes IP (which was already increased by 1), effectively pushing the address of the next instruction. This value will be used by the RET instruction to set IP to the address of next instruction after CALL * sets ip to SP - 1 - argument, with is the address of the FSTART instruction of the function being called

ALLOC: * generated by the new operator * a sequence of simpler instructions performed to ALLOCate memory dynamically * remembers the value on the top of the stack, which is the requested amount of memory * saves the value of HP * increases HP by the requested amount of memory * pushes the previously saved value of HP onto the stack

ASSIGN: * used to change the value at some address to the value on the top of the stack (ASSIGNment), e.g. “a = 2;” * remembers VAL, the value on the top of the stack * sets the value at address SP - 1 to VAL * performs a POP * sets the value on the top of the stack to VAL. This leaves the door open to multiple variable assignment (such as “a = b = 3;”)

PLUSEQ: * just like ASSIGN, but performs addition instead of replacing the value (e.g. a += 2;)

MINUSEQ: * just like ASSIGN, but performs subtraction instead of replacing the value (e.g. a -= 2;)

TIMESEQ: * just like ASSIGN, but performs multiplication instead of replacing the value (e.g. a *= 2;)

DIVEQ: * just like ASSIGN, but performs division instead of replacing the value (e.g. a /= 2;)

MODEQ: * just like ASSIGN, but performs the moduls operation instead of replacing the value (e.g. `a %= 2;`)

TIMES: * POPs the two values on the top of the stack, multiplies them, and pushes the result (e.g. `6 * 3`)

DIV: * POPs the two values on the top of the stack, divides them, and pushes the result (e.g. `6 / 3`)

MOD: * POPs the two values on the top of the stack, performs the modulus operation on them, and pushes the result (e.g. `6 % 3`)

PLUS: * POPs the two values on the top of the stack, adds them, and pushes the result (e.g. `6 + 3`)

MINUS: * POPs the two values on the top of the stack, subtracts them, and pushes the result (e.g. `6 - 3`)

NOT: * ... * is used to compile "*!expression*"

OPP: * ... * is used to compile "-" in "*-expression*" ("OPPOSITE")

EQ, NE, LT, GT, LTE, GTE, AND, OR: * ... * are used to compile `==`, `!=`, `<`, `>`, `<=`, `>=`, `&&`, `||` (e.g. "`a < 3 || b == 5`")

Missing language features

- only bool, int, char and pointers as data types and they all have the same size in memory
- no malloc, but operator new is working (like in C++),
- no static arrays and structs (dynamic arrays and pointer to structs work fine).
- no arrays of structs (arrays of pointers to structs works fine).
- no for and switch statements
- no preprocessor directives
- no bitwise operators
- no ++, --, ternary operators
- no union and no enum
- no global variables
- no function pointers
- no free / delete operator
- no function overloading