load

Format: `load dst(r) src(k)`

Copies constant src to register dst.

new_object

Format: `new_object dst(r)`

Constructs a new empty Object instance using the original constructor, and puts the result in register dst.

new_array

Format: `new_array dst(r) firstArg(r) argCount(n)`

Constructs a new Array instance using the original constructor, and puts the result in register dst. The array will contain argCount elements with values taken from registers starting at register firstArg.

new_regexp

Format: `new_regexp dst(r) regExp(re)`

Constructs a new RegExp instance using the original constructor from regexp regExp, and puts the result in register dst.

mov

Format: `mov dst(r) src(r)`

Copies register src to register dst.

eq

Format: `eq dst(r) src1(r) src2(r)`

Checks whether register src1 and register src2 are equal, as with the ECMAScript '==' operator, and puts the result as a boolean in register dst.

neq

Format: `neq dst(r) src1(r) src2(r)`

Checks whether register src1 and register src2 are not equal, as with the ECMAScript '!=' operator, and puts the result as a boolean in register dst.

stricteq

Format: `stricteq dst(r) src1(r) src2(r)`

Checks whether register src1 and register src2 are strictly equal, as with the ECMAScript '===' operator, and puts the result as a boolean in register dst.

nstricteq

Format: `nstricteq dst(r) src1(r) src2(r)`

Checks whether register src1 and register src2 are not strictly equal, as with the ECMAScript '!==' operator, and puts the result as a boolean in register dst.

less

Format: `less dst(r) src1(r) src2(r)`

Checks whether register src1 is less than register src2, as with the ECMAScript '<' operator, and puts the result as a boolean in register dst.

lesseq

Format: `lesseq dst(r) src1(r) src2(r)`

Checks whether register src1 is less than or equal to register src2, as with the ECMAScript '<=' operator, and puts the result as a boolean in register dst.

pre_inc

Format: `pre_inc srcDst(r)`

Converts register srcDst to number, adds one, and puts the result back in register srcDst.

pre_dec

Format: `pre_dec srcDst(r)`

Converts register srcDst to number, subtracts one, and puts the result back in register srcDst.

post_inc

Format: `post_inc dst(r) srcDst(r)`

Converts register srcDst to number. The number itself is written to register dst, and the number plus one is written back to register srcDst.

post_dec

Format: `post_dec dst(r) srcDst(r)`

Converts register srcDst to number. The number itself is written to register dst, and the number minus one is written back to register srcDst.

to_jsnumber

Format: `to_jsnumber dst(r) src(r)`

Converts register src to number, and puts the result in register dst.

negate

Format: `negate dst(r) src(r)`

Converts register src to number, negates it, and puts the result in register dst.

add

Format: `add dst(r) src1(r) src2(r)`

Adds register src1 and register src2, and puts the result in register dst. (JS add may be string concatenation or numeric add, depending on the types of the operands.)

mul

Format: `mul dst(r) src1(r) src2(r)`

Multiplies register src1 and register src2 (converted to numbers), and puts the product in register dst.

div

Format: `div dst(r) dividend(r) divisor(r)`

Divides register dividend (converted to number) by the register divisor (converted to number), and puts the quotient in register dst.

mod

Format: `mod dst(r) dividend(r) divisor(r)`

Divides register dividend (converted to number) by register divisor (converted to number), and puts the remainder in register dst.

sub

Format: `sub dst(r) src1(r) src2(r)`

Subtracts register src2 (converted to number) from register src1 (converted to number), and puts the difference in register dst.

lshift

Format: `lshift dst(r) val(r) shift(r)`

Performs left shift of register val (converted to int32) by register shift (converted to uint32), and puts the result in register dst.

rshift

Format: `rshift dst(r) val(r) shift(r)`

Performs arithmetic right shift of register val (converted to int32) by register shift (converted to uint32), and puts the result in register dst.

urshift

Format: `rshift dst(r) val(r) shift(r)`

Performs logical right shift of register val (converted to uint32) by register shift (converted to uint32), and puts the result in register dst.

bitand

Format: `bitand dst(r) src1(r) src2(r)`

Computes bitwise AND of register src1 (converted to int32) and register src2 (converted to int32), and puts the result in register dst.

bitxor

Format: `bitxor dst(r) src1(r) src2(r)`

Computes bitwise XOR of register src1 (converted to int32) and register src2 (converted to int32), and puts the result in register dst.

bitor

Format: `bitor dst(r) src1(r) src2(r)`

Computes bitwise OR of register src1 (converted to int32) and register src2 (converted to int32), and puts the result in register dst.

bitnot

Format: `bitnot dst(r) src(r)`

Computes bitwise NOT of register src1 (converted to int32), and puts the result in register dst.

not

Format: `not dst(r) src1(r) src2(r)`

Computes logical NOT of register src1 (converted to boolean), and puts the result in register dst.

instanceof

Format: `instanceof dst(r) value(r) constructor(r)`

Tests whether register value is an instance of register constructor, and puts the boolean result in register dst. Raises an exception if register constructor is not an object.

typeof

Format: `typeof dst(r) src(r)`

Determines the type string for src according to ECMAScript rules, and puts the result in register dst.

in

Format: `in dst(r) property(r) base(r)`

Tests whether register base has a property named register property, and puts the boolean result in register dst. Raises an exception if register constructor is not an object.

resolve

Format: `resolve dst(r) property(id)`

Looks up the property named by identifier property in the scope chain, and writes the resulting value to register dst. If the property is not found, raises an exception.

resolve_skip

Format: `resolve_skip dst(r) property(id) skip(n)`

Looks up the property named by identifier property in the scope chain skipping the top 'skip' levels, and writes the resulting value to register dst. If the property is not found, raises an exception.

get_scoped_var

Format: `get_scoped_var dst(r) index(n) skip(n)`

Loads the contents of the index-th local from the scope skip nodes from the top of the scope chain, and places it in register dst

put_scoped_var

Format: `put_scoped_var index(n) skip(n) value(r)`

resolve_base

Format: `resolve_base dst(r) property(id)`

Searches the scope chain for an object containing identifier property, and if one is found, writes it to register dst. If none is found, the outermost scope (which will be the global object) is stored in register dst.

resolve_with_base

Format: `resolve_with_base baseDst(r) propDst(r) property(id)`

Searches the scope chain for an object containing identifier property, and if one is found, writes it to register srcDst, and the retrieved property value to register propDst. If the property is not found, raises an exception. This is more efficient than doing resolve_base followed by resolve, or resolve_base followed by get_by_id, as it avoids duplicate hash lookups.

resolve_func

Format: `resolve_func baseDst(r) funcDst(r) property(id)`

Searches the scope chain for an object containing identifier property, and if one is found, writes the appropriate object to use as "this" when calling its properties to register baseDst; and the retrieved property value to register propDst. If the property is not found, raises an exception. This differs from resolve_with_base, because the global this value will be substituted for activations or the global object, which is the right behavior for function calls but not for other property lookup.

get_by_id

Format: `get_by_id dst(r) base(r) property(id)`

Converts register base to Object, gets the property named by identifier property from the object, and puts the result in register dst.

put_by_id

Format: `put_by_id base(r) property(id) value(r)`

Sets register value on register base as the property named by identifier property. Base is converted to object first. Unlike many opcodes, this one does not write any output to the register file.

del_by_id

Format: `del_by_id dst(r) base(r) property(id)`

Converts register base to Object, deletes the property named by identifier property from the object, and writes a boolean indicating success (if true) or failure (if false) to register dst.

get_by_val

Format: `get_by_val dst(r) base(r) property(r)`

Converts register base to Object, gets the property named by register property from the object, and puts the result in register dst. property is nominally converted to string but numbers are treated more efficiently.

put_by_val

Format: `put_by_val base(r) property(r) value(r)`

Sets register value on register base as the property named by register property. Base is converted to object first. register property is nominally converted to string but numbers are treated more efficiently. Unlike many opcodes, this one does not write any output to the register file.

del_by_val

Format: `del_by_val dst(r) base(r) property(r)`

Converts register base to Object, deletes the property named by register property from the object, and writes a boolean indicating success (if true) or failure (if false) to register dst.

put_by_index

Format: `put_by_index base(r) property(n) value(r)`

Sets register value on register base as the property named by the immediate number property. Base is converted to object first. Unlike many opcodes, this one does not write any output to the register file. This opcode is mainly used to initialize array literals.

loop

Format: `loop target(offset)`

Jumps unconditionally to offset target from the current instruction. Additionally this loop instruction may terminate JS execution is the JS timeout is reached.

jmp

Format: `jmp target(offset)`

Jumps unconditionally to offset target from the current instruction.

loop_if_true

Format: `loop_if_true cond(r) target(offset)`

Jumps to offset target from the current instruction, if and only if register cond converts to boolean as true. Additionally this loop instruction may terminate JS execution is the JS timeout is reached.

jtrue

Format: `jtrue cond(r) target(offset)`

Jumps to offset target from the current instruction, if and only if register cond converts to boolean as true.

jfalse

Format: `jfalse cond(r) target(offset)`

Jumps to offset target from the current instruction, if and only if register cond converts to boolean as false.

loop_if_less

Format: `loop_if_less src1(r) src2(r) target(offset)`

Checks whether register src1 is less than register src2, as with the ECMAScript '<' operator, and then jumps to offset target from the current instruction, if and only if the result of the comparison is true. Additionally this loop instruction may terminate JS execution is the JS timeout is reached.

jless

Format: `jless src1(r) src2(r) target(offset)`

Checks whether register src1 is less than register src2, as with the ECMAScript '<' operator, and then jumps to offset target from the current instruction, if and only if the result of the comparison is true.

jnless

Format: `jnless src1(r) src2(r) target(offset)`

Checks whether register src1 is less than register src2, as with the ECMAScript '<' operator, and then jumps to offset target from the current instruction, if and only if the result of the comparison is false.

switch_imm

Format: `switch_imm tableIndex(n) defaultOffset(offset) scrutinee(r)`

Performs a range checked switch on the scrutinee value, using the tableIndex-th immediate switch jump table. If the scrutinee value is an immediate number in the range covered by the referenced jump table, and the value at jumpTable[scrutinee value] is non-zero, then that value is used as the jump offset, otherwise defaultOffset is used.

switch_char

Format: `switch_char tableIndex(n) defaultOffset(offset) scrutinee(r)`

Performs a range checked switch on the scrutinee value, using the tableIndex-th character switch jump table. If the scrutinee value is a single character string in the range covered by the referenced jump table, and the value at jumpTable[scrutinee value] is non-zero, then that value is used as the jump offset, otherwise defaultOffset is used.

switch_string

Format: `switch_string tableIndex(n) defaultOffset(offset) scrutinee(r)`

Performs a sparse hashmap based switch on the value in the scrutinee register, using the tableIndex-th string switch jump table. If the scrutinee value is a string that exists as a key in the referenced jump table, then the value associated with the string is used as the jump offset, otherwise defaultOffset is used.

new_func

Format: `new_func dst(r) func(f)`

Constructs a new Function instance from function func and the current scope chain using the original Function constructor, using the rules for function declarations, and puts the result in register dst.

new_func_exp

Format: `new_func_exp dst(r) func(f)`

Constructs a new Function instance from function func and the current scope chain using the original Function constructor, using the rules for function expressions, and puts the result in register dst.

call_eval

Format: `call_eval dst(r) func(r) thisVal(r) firstArg(r) argCount(n)`

Call a function named "eval" with no explicit "this" value (which may therefore be the eval operator). If register thisVal is the global object, and register func contains that global object's original global eval function, then perform the eval operator in local scope (interpreting the argument registers as for the "call" opcode). Otherwise, act exactly as the "call" opcode would.

call

Format: call dst(r) func(r) thisVal(r) firstArg(r) argCount(n)

Perform a function call. Specifically, call register func with a "this" value of register thisVal, and put the result in register dst. The arguments start at register firstArg and go up to argCount, but the "this" value is considered an implicit first argument, so the argCount should be one greater than the number of explicit arguments passed, and the register after firstArg should contain the actual first argument. This opcode will copy from the thisVal register to the firstArg register, unless the register index of thisVal is the special missing this object marker, which is 2^31-1; in that case, the global object will be used as the "this" value. If func is a native code function, then this opcode calls it and returns the value immediately. But if it is a JS function, then the current scope chain and code block is set to the function's, and we slide the register window so that the arguments would form the first few local registers of the called function's register window. In addition, a call frame header is written immediately before the arguments; see the call frame documentation for an explanation of how many registers a call frame takes and what they contain. That many registers before the firstArg register will be overwritten by the call. In addition, any registers higher than firstArg + argCount may be overwritten. Once this setup is complete, execution continues from the called function's first argument, and does not return until a "ret" opcode is encountered.

ret

Format: `ret result(r)`

Return register result as the return value of the current function call, writing it into the caller's expected return value register. In addition, unwind one call frame and restore the scope chain, code block instruction pointer and register base to those of the calling function.

construct

Format: `construct dst(r) constr(r) firstArg(r) argCount(n)`

Invoke register "constr" as a constructor. For JS functions, the calling convention is exactly as for the "call" opcode, except that the "this" value is a newly created Object. For native constructors, a null "this" value is passed. In either case, the firstArg and argCount registers are interpreted as for the "call" opcode.

push_scope

Format: `push_scope scope(r)`

Converts register scope to object, and pushes it onto the top of the current scope chain.

pop_scope

Format: `pop_scope`

Removes the top item from the current scope chain.

get_pnames

Format: `get_pnames dst(r) base(r)`

Creates a property name list for register base and puts it in register dst. This is not a true JavaScript value, just a synthetic value used to keep the iteration state in a register.

next_pname

Format: `next_pname dst(r) iter(r) target(offset)`

Tries to copies the next name from property name list in register iter. If there are names left, then copies one to register dst, and jumps to offset target. If there are none left, invalidates the iterator and continues to the next instruction.

jmp_scopes

Format: `jmp_scopes count(n) target(offset)`

Removes the a number of items from the current scope chain specified by immediate number count, then jumps to offset target.

catch

Format: `catch ex(r)`

Retrieves the VMs current exception and puts it in register ex. This is only valid after an exception has been raised, and usually forms the beginning of an exception handler.

throw

Format: `throw ex(r)`

Throws register ex as an exception. This involves three steps: first, it is set as the current exception in the VM's internal state, then the stack is unwound until an exception handler or a native code boundary is found, and then control resumes at the exception handler if any or else the script returns control to the nearest native caller.

new_error

Format: `new_error dst(r) type(n) message(k)`

Constructs a new Error instance using the original constructor, using immediate number n as the type and constant message as the message string. The result is written to register dst.

end

Format: `end result(r)`

Return register result as the value of a global or eval program. Return control to the calling native code.

put_getter

Format: `put_getter base(r) property(id) function(r)`

Sets register function on register base as the getter named by identifier property. Base and function are assumed to be objects as this op should only be used for getters defined in object literal form. Unlike many opcodes, this one does not write any output to the register file.

put_setter

Format: `put_setter base(r) property(id) function(r)`

Sets register function on register base as the setter named by identifier property. Base and function are assumed to be objects as this op should only be used for setters defined in object literal form. Unlike many opcodes, this one does not write any output to the register file.

jsr

Format: `jsr retAddrDst(r) target(offset)`

Places the address of the next instruction into the retAddrDst register and jumps to offset target from the current instruction.

sret

Format: `sret retAddrSrc(r)`

Jumps to the address stored in the retAddrSrc register. This differs from op_jmp because the target address is stored in a register, not as an immediate.

debug

Format: `debug debugHookID(n) firstLine(n) lastLine(n)`

Notifies the debugger of the current state of execution. This opcode is only generated while the debugger is attached.