The byte values for each instruction in the binary format are listed at `https://webassembly.github.io/spec/core/binary/instructions.html`.

Values

WebAssembly has the following types of values:

Type	Constructor	Bit width	Notes
Integer	`i32`	32-bit	Also used to store booleans and memory addresses.
	`i64`	64-bit	
Float[1]	`f32`	32-bit	
	`f64`	64-bit	
Vector	`v128`	128-bit	Can store floats (4 32-bit, or 2 64-bit) or integers (2 64-bit, 4 32-bit, 8 16-bit, or 16 8-bit).
References		Opaque	Pointers to various types of entities

Integers are interpreted as signed or unsigned numbers, depending on the operations applied to them.

But the spec also says there are (1) unsigned, (2) signed, and (3) uninterpreted integers? (structure>values)

Integer encoding

Integers n are encoded in the binary format using a variable-length integer encoding.

Numeric Instructions

ixx and fxx represent instructions that exist for both 32-bit and 64-bit values. Instructions ending in _u are unsigned operations that have an equivalent signed operation ending in _s.

For binary instructions on the stack, the first operand is always the one that was pushed to the stack first, and th second operand is the one pushed to the stack last. For example,

```
i64.const 10  # first operand
i64.const 2   # second operand
i64.sub
```

will do the operation 10 - 2.

Const	ixx.const n fxx.const z	Creates a constant value of specified type.
Comparison	ixx.eqz	Equal to zero *(no floating-point type equivalent)*
	ixx.eq	Equality
	ixx.ne	Not equal
	ixx.lt_u	Less than
	ixx.gt_u	Greater than
	ixx.le_u	Less than or equal
	ixx.ge_u	Greater than or equal
Equivalent float comparison operators exist, for f32 and f64 respectively, with the difference of not having signed/unsigned variants. For example, the less than operator f64.lt.		
Unary operations	ixx.clz	Count leading zeros
	ixx.ctz	Count trailing zeros

[1] Specified by IEEE 754-2019 (`https://ieeexplore.ieee.org/document/8766229`)

	ixx.popcnt	Count the number of bits set to 1 (population count)
Arithmetic operations	ixx.add	Addition
	ixx.sub	Subtraction
	ixx.mul	Multiplication
	ixx.div_u	Division
	ixx.rem_u	Remainder
Bitwise operations	ixx.and	Bitwise AND
	ixx.or	Bitwise OR
	ixx.xor	Bitwise XOR
	ixx.shl	Bitwise left-shift
	ixx.shr_u	Bitwise right-shift. This operator is signed/unsigned because of sign extension.
	ixx.rotl	Bitwise left-rotate
	ixx.rotr	Bitwise right-rotate
Floating-point specific	fxx.min	Minimum of two numbers
	fxx.max	Maximum of two numbers
	fxx.copysign	Copy the sign bit from the second operand to the first operand
	fxx.abs	Absolute value
	fxx.neg	Negate
	fxx.sqrt	Square root
	fxx.ceil	Ceiling function
	fxx.floor	Floor function
	fxx.trunc	Truncate (discard everything after the decimal point). For negative numbers, floor will round down whereas trunc will round up.
	fxx.nearest	Round to the nearest integer
Conversion	i32.wrap_i64	Reduce an i64 to an i32, taking just the lower 32 bits (i.e. taking it modulo 2^{32})
	i64.extend_i32_u	Sign-extend from an i32 to an i64
	ixx.trunc_fxx_u	Truncate a float to an integer. Available for every combination of 32/64-bit and signed/unsigned.
	f32.demote_f64	Convert a f64 to a f32. Unlike the integer wrap instruction, this will lose precision but not change the value of the number to an entirely different number.
	f64.promote_f32	Convert a f32 to a f64
	fxx.convert_ixx_u	Convert an integer to a floating-point. Available for every combination of 32/64-bit and signed/unsigned.
	ixx.reinterpret_fxx	Reinterpret the bits of a float as an integer.
	fxx.reinterpret_ixx	Reinterpret the bits of an integer as a float.

Variable instructions

These instructions are for getting/setting local and global variables.

Local variables are declared in function definitions, and global variables are declared in module definitions.

local.get x	Get the value of the variable x and put it on the stack
local.set x	Set the value of the variable x to the value on top of the stack (and remove it from the stack)
local.tee x	The same as local.set, but also leaves the value on the stack
global.get x	Same as local.get for a global variable
global.set x	Same as local.set for a global variable

Control flow instructions

Here, '. . .' represents any sequence of instructions. Labels can also be omitted, in which case blocks/loops are implicitly labelled by their nesting depth.

block *$label* . . . end	Creates a block that can be branched out of using a br instruction. The label is used to identify which block to branch out of. This treats br like a `break` statement in C.
loop *$label* . . . end	Effectively the opposite of block. loop creates a 'block' that can be branched to the beginning of. It doesn't loop by itself, it needs a br instruction inside the loop to go back to the start of the loop each iteration. This treats br like a `continue` statement in C.
if . . . else . . . end	Executes the first statement if the top value on the stack is true (positive), and the second statement if the top of the stack is false (0).
br *$label*	Unconditionally branches. If *$label* refers to a block, it jumps to the end of the block. If *$label* refers to a loop, it jumps to the start of the loop.
br_if *$label*	Conditional branch
return	Returns from a function. If the stack is empty, nothing is returned. If the stack contains at least as many values as the function's return type signature specifies, those values are returned from the top of the stack, and any other values below them on the stack are discarded.
call *$funcidx*	Calls a function.
call_indirect *$tableidx $typeidx*	Calls a function from a table. *$typeidx* must be funcref.
nop	Does nothing.
unreachable	Marks a point in code that should be unreachable. If this instruction is executed, it unconditionally traps. (Similar to a failed assertion in C.)
select	Chooses between its first two operands, depending on if the third operand is zero (selects the second operand) or not (chooses the first operand).
drop	Pops the top value from the stack and immediately discards it.

Memory instructions

ixx.load	Load an integer from memory, at the address given by the top of the stack.
fxx.load	Load a float from memory.
ixx.load8_u ixx.load16_u i64.load32_u	Integer loads can specify a smaller bit-width to load. Signed and unsigned instructions exist, to specify how to sign-extend the number.
ixx.store fxx.store	Store the second operand at the memory offset given by the first operand.
ixx.store8 ixx.store16 i64.store32	Integer stores can specify a smaller bit-width to store in that location.
memory.grow	Grow the memory by the number of pages given by the operand.
memory.size	Get the number of pages the memory currently has.
memory.fill	Set all the bytes in the specified region to a given byte.
memory.copy	Copy data from one memory region to another (can be overlapping).
memory.init	Copy data from a passive data segment into memory.
data.drop	Prevent any further use of a passive data segment (allows the memory used by it to be freed).

Data segments

Initially, the program's memory is filled with zero bytes. Data segments exist to allow memory to be initialised from static bytes.

Data segments can either be passive or active. Passive data segments are loaded into memory explicitly using the `memory.init` instruction. Active data segments are automatically copied into memory when the program loads. Active data segments specify the offset where they'll be loaded.

Tables

A table is an array of function pointers, that can be used to indirectly call functions. Tables live outside of WebAssembly's memory, so they can't be seen from the program itself. This keeps the memory addresses of functions hidden. To call a function referenced in a table, the `call_indirect` instruction is used.

Modules