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	funcref	Opaque	Pointers to functions, of any type
	externref		Pointers to other types of objects that can be passed into WebAssembly

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? (docs>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.		

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

Unary operations	ixx.clz	Count leading zeros
	ixx.ctz	Count trailing zeros
	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.

labels vs implicitly referencing by 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

A module is a compilation and loading unit for a WebAssembly program. All Wasm programs are organised into a module.

Modules define the different parts of the program. Modules are split up into sections, which each start with a section ID, the size of the section in bytes, followed by the section contents. Each section is optional, leaving a section out is equivalent to including it and leaving it empty.

To show how modules work, I've used the `wat2wasm` tool to view the generated binary for the following Wasm program:

```
1  (module
2    (import "console" "log" (func $log (param i32)))
3    (func $main (param i32) (param i32)
4      i32.const 10
5      global.get $foo
6      ;; call function at index 0 in table ($add)
7      i32.const 0
8      call_indirect (type $addtype)
9      call $log
10   )
11   (type $addtype (func (param i32) (param i32) (result i32)))
12   (func $add (type $addtype) (param i32) (param i32) (result i32)
13     local.get 0
14     local.get 1
```

```
    i32.add
  )
  (func $startfunc
    i32.const 4
    global.set $foo
  )
  (start $startfunc)
  (global $foo (mut i32) (i32.const 42))
  (memory 1)
  (data (i32.const 0) "Hello world")
  (table $table 16 funcref)
  (elem (i32.const 0) $add)
  (export "main" (func $main))
  (export "memory" (memory 0))
)
```

Preamble

A Wasm binary always starts with an 8-byte preamble. This contains a 4-byte magic number (the string '`\0asm`'), and a WebAssembly version number. The current version number is 1; it could be incremented in future if breaking changes are implemented.

```
0000000: 0061 736d                                 ; WASM_BINARY_MAGIC
0000004: 0100 0000                                 ; WASM_BINARY_VERSION
```

Types section

The section starts with the section code to identify the section, then the size of the section in bytes. The size refers to everything in the section apart from the code and size, so here `0x10` = 16 bytes.

```
; section "Type" (1)
0000008: 01                                        ; section code
0000009: 13                                        ; section size
```

The types section defines the types of all the functions in the module (including imported functions). If multiple functions have the same type, it will only be defined once here; each function specifies which index type it has.

The next byte says how many types are defined here.

```
000000a: 04                                        ; num types
```

The rest of the section is a vector of the function types, that is, just a list of the types after one another. Each function type is identified by the starting byte `0x60`.

The next byte is the number of function parameters, followed by a byte encoding the type of each parameter. Here, `0x7f` is the code for an `i32`.

The result type is encoded in the same way; a count followed by a byte for each result type (functions can have either 0 or 1 return value, currently).

Types are referred to by an index starting from 0.

Type definition for the explicitly defined type (`$addtype`):

```
; func type 0
000000b: 60                                        ; func
```

```
000000c: 02                                        ; num params
000000d: 7f                                        ; i32
000000e: 7f                                        ; i32
000000f: 01                                        ; num results
0000010: 7f                                        ; i32
```

Type definition for the imported function (`$log`):

```
; func type 1
0000011: 60                                        ; func
0000012: 01                                        ; num params
0000013: 7f                                        ; i32
0000014: 00                                        ; num results
```

Type definition for `$main`:

```
; func type 2
0000015: 60                                        ; func
0000016: 02                                        ; num params
0000017: 7f                                        ; i32
0000018: 7f                                        ; i32
0000019: 00                                        ; num results
```

Type definition for the `$startfunc`:

```
; func type 3
000001a: 60                                        ; func
000001b: 00                                        ; num params
000001c: 00                                        ; num results
```

Imports

The imports section defines all the imports to the module.

```
; section "Import" (2)
000001d: 02                                        ; section code
000001e: 0f                                        ; section size
000001f: 01                                        ; num imports
```

In this program, there's only one import, the `console.log` function. Wasm imports are defined by a two-level hierarchy, a module name and a field name. These are stored at strings, preceded by a byte specifying their length.

The *import kind* byte specifies what type of import this is. `0x00` defines this as a function import. Other types are `0x01` for tables, `0x02` for memory, and `0x03` for globals. The following byte, in the case of functions, specifies which type (indexed from zero) this function has.

Import definition on line 2:

```
; import header 0
0000020: 07                                         ; string length
0000021: 636f 6e73 6f6c 65                         console  ; import module name
0000028: 03                                         ; string length
0000029: 6c6f 67                                   log  ; import field name
000002c: 00                                         ; import kind
000002d: 01                                         ; import signature index
```

Functions

The functions section maps between function definitions and their types. For each function index, it specifies the index of the type defined in the types section that matches that function's type.

(Imported functions aren't included here, their type is specified in the imports section.)

```
; section "Function" (3)
000002e: 03                                        ; section code
000002f: 04                                        ; section size
```

The body of this section first gives the number of functions, then the index of each function's type, starting at function index 0 and increasing consecutively. (These indexes point to the definitions in the types section.)

```
0000030: 03                                        ; num functions
0000031: 02                                        ; function 0 signature index
0000032: 00                                        ; function 1 signature index
0000033: 03                                        ; function 2 signature index
```

Tables

The tables section contains a vector of all the tables defined in the module.

```
; section "Table" (4)
0000034: 04                                        ; section code
0000035: 04                                        ; section size
0000036: 01                                        ; num tables
```

Each table is encoded with the type it stores as the first byte (funcref or externref), followed by the size limits. If the limits flag is `0x01`, then the minimum and maximum are specified, if it's `0x00` then no maximum is given. In this example, only a minimum is specified, and the `0x10` corresponds to the table size of 16 that we set on line 25.

```
; table 0
0000037: 70                                        ; funcref
0000038: 00                                        ; limits: flags
0000039: 10                                        ; limits: initial
```

Memory

The memory section is very similar to the table section. It contains a vector of the memories defined in the module. (In the current WebAssembly version, only a single memory can be defined.)

```
; section "Memory" (5)
000003a: 05                                        ; section code
000003b: 03                                        ; section size
000003c: 01                                        ; num memories
```

Each memory is specified by its size limits (in units of page size), in the same way as tables above.

```
; memory 0
000003d: 00                                        ; limits: flags
000003e: 01                                        ; limits: initial
```

Globals

The globals section contains a vector of all the global variables in the module and how they're initialised. Globals are initialised by a constant expression.

```
; section "Global" (6)
000003f: 06                                        ; section code
0000040: 06                                        ; section size
0000041: 01                                        ; num globals
```

Each global starts by specifying its type. The first byte gives the value type (here i32), and then its mutability. `0x00` is an immutable global, and `0x01` is a mutable global.

```
0000042: 7f                                        ; i32
0000043: 01                                        ; global mutability
```

After the type is the initialiser expression for the global, which must be a constant expression. Here, the constant expression (i32.const 42) is represented.

```
0000044: 41                                        ; i32.const
0000045: 2a                                        ; i32 literal
0000046: 0b                                        ; end
```

Exports

The exports section defines all the exports from the module.

```
; section "Export" (7)
0000047: 07                                        ; section code
0000048: 11                                        ; section size
0000049: 02                                        ; num exports
```

Each export starts with a string, specifying its name.

After the name is a flag to specify the type of export. `0x00` is a function export, `0x01` is a table export, `0x02` is a memory export, and `0x03` is a global export. After the flag, the index of the respective entity to export is given.

`(export "main" (func $main))`

```
000004a: 04                                         ; string length
000004b: 6d61 696e                                 main  ; export name
000004f: 00                                         ; export kind
0000050: 01                                         ; export func index
```

`(export "memory" (memory 0))`

```
0000051: 06                                         ; string length
0000052: 6d65 6d6f 7279                            memory  ; export name
0000058: 02                                         ; export kind
0000059: 00                                         ; export memory index
```

Start

The start section specifies an optional function that should be run automatically when the module is initialised. This could be used, for example, to initialised a global variable to a non-constant expression.

The section contains a single function index to specify the function to run.

```
; section "Start" (8)
000005a: 08                                        ; section code
000005b: 01                                        ; section size
000005c: 03                                        ; start func index
```

Element

Element segments are used to statically initialise the contents of tables.

```
; section "Elem" (9)
000005d: 09                                        ; section code
000005e: 07                                        ; section size
000005f: 01                                        ; num elem segments
```

Each element segment starts with a flag (`0x00`–`0x07`) that specifies the mode of the element and the binary structure of this segment. In this case of flag `0x00`, the element segment is active (it gets automatically initialised to its table during instantiation), and the element automatically belongs to table 0.

Following the flag is a constant expression that specifies the offset into the table, which is where the elements will be stored.

After this come the actual elements themselves, each being a function index.

```
; elem segment header 0
0000060: 00                                        ; segment flags
0000061: 41                                        ; i32.const
0000062: 00                                        ; i32 literal
0000063: 0b                                        ; end
0000064: 01                                        ; num elems
0000065: 02                                        ; elem function index
```

Data count

The data count section is an optional section used by single-pass validators. Because the data section comes after the code section, a validator could only check if instructions that reference data indexes (memory.init, data.drop) are valid until it's read the data section, at which point it would have to go back to the code section again. The data count section helps by storing the number of data segments before the code section, so a validator doesn't have to defer validation.

```
; section "DataCount" (12)
0000066: 0c                                        ; section code
0000067: 01                                        ; section size
0000068: 01                                        ; data count
```

Code

The code section contains the bodies of all the functions of the program.

```
; section "Code" (10)
0000069: 0a                                        ; section code
000006a: 1e                                        ; section size
000006b: 03                                        ; num functions
```

Each entry in the code section is the code for one function. It starts by specifying the size (in bytes) of the function code.

Next is the declaration of any local variables, starting with the number of declarations. Each declaration is a pair of a count and a type, which specifies that many local variables of that type.

The function body code follows this as an expression; that is, a sequence of instructions terminated by an end instruction.

func $main

```
; function body 0
000006c: 0d                                        ; func body size
000006d: 00                                        ; local decl count
000006e: 41                                        ; i32.const
000006f: 0a                                        ; i32 literal
0000070: 23                                        ; global.get
0000071: 00                                        ; global index
0000072: 41                                        ; i32.const
0000073: 00                                        ; i32 literal
0000074: 11                                        ; call_indirect
0000075: 00                                        ; signature index
0000076: 00                                        ; table index
0000077: 10                                        ; call
0000078: 00                                        ; function index
0000079: 0b                                        ; end
```

func $add

```
; function body 1
000007a: 07                                        ; func body size
000007b: 00                                        ; local decl count
000007c: 20                                        ; local.get
000007d: 00                                        ; local index
000007e: 20                                        ; local.get
000007f: 01                                        ; local index
0000080: 6a                                        ; i32.add
0000081: 0b                                        ; end
```

func $startfunc

```
; function body 2
0000082: 06                                        ; func body size
0000083: 00                                        ; local decl count
0000084: 41                                        ; i32.const
0000085: 04                                        ; i32 literal
0000086: 24                                        ; global.set
0000087: 00                                        ; global index
0000088: 0b                                        ; end
```

Data

Data segments are used to statically initialise the contents of memory.

```
; section "Data" (11)
0000086: 0b                                        ; section code
0000087: 11                                        ; section size
0000088: 01                                        ; num data segments
```

Each segment starts with a flag to specify the mode, and structure of this segment. Flag `0x00` is an active data segment, which automatically references memory 0. Flat `0x01` is a passive data segment, which needs to be explicitly initialised into memory with the memory.init instruction. Flag `0x02` is an active data segment, where the memory index is also explicitly defined. (In the current Wasm version, at most one memory is allowed, so there's no real use for this yet.) The flag can be interpreted as a bit field, where bit 0 (least significant) being present indicates the segment being passive, and bit 1 indicates the memory index being explicitly defined.

This is followed by a constant initialiser expression that specifies the offset into the memory to store the data at.

```
; data segment header 0
0000089: 00                                        ; segment flags
000008a: 41                                        ; i32.const
000008b: 00                                        ; i32 literal
000008c: 0b                                        ; end
```

The remainder of the segment is the actual data itself.

```
000008d: 0b                                        ; data segment size
; data segment data 0
000008e: 4865 6c6c 6f20 776f 726c 64               ; data segment data
```

Strings and `printf`

Wasm doesn't have any concept of a 'string'; it can only handle integers and floats. However, since Wasm has linear memory of bytes, we can store strings as an array of their character values, terminated with a `0` (the null character), like in C.

This will work fine for string manipulation in Wasm programs. However, printing a string to console, for example, is more complicated, because this has to be done by calling a function imported from the calling environment (i.e. from JavaScript).

How do I handle this when I write the compiler - do I need to provide a specification for the functions the JS is required to import to Wasm?

Wasm functions take a fixed number of arguments, of the defined value types. When we import a function from JavaScript, we have to also define it's function signature in Wasm. Therefore if we simply import the `console.log()` function directly, we have to give it a fixed number of numeric arguments. This would allow us to print single numbers to console, but doesn't help us print strings.

To print strings, we have to share the Wasm memory with the JavaScript (by exporting it), and then write a function that takes an index into the memory, reads a null-terminated string from there one byte at a time, and then prints the resulting string. This function can then be imported into Wasm, and used as a `printf` function, by passing a pointer to the string to print.

Handling user input from stdin would require a similar function import, that uses JavaScript to take user input and write the string into Wasm memory, returning the index back to Wasm.