
NanoWasm Specification

The SpecTec Team

Feb 13, 2026

CONTENTS

1 Abstract Syntax	3
2 Validation	5
2.1 nop	5
2.2 drop	5
2.3 select	5
2.4 const	5
2.5 local.get	5
2.6 local.set	6
2.7 global.get	6
2.8 global.set	6
3 Execution	7
3.1 nop	7
3.2 drop	7
3.3 select	7
3.4 local.get x	8
3.5 local.set x	8
3.6 global.get x	8
3.7 global.set x	8
4 Binary Format	9

NanoWasm is a small language with simple types and instructions.

CHAPTER
ONE

ABSTRACT SYNTAX

The *abstract syntax* of types is as follows:

```
mut    ::=  mut
valtype ::=  i32 | i64 | f32 | f64
functype ::=  valtype* → valtype*
globaltype ::=  mut? valtype
```

Instructions take the following form:

```
const  ::=  0 | 1 | 2 | ...
instr  ::=  nop
        |
        | drop
        | select
        | valtype.const const
        | local.get localidx
        | local.set localidx
        | global.get globalidx
        | global.set globalidx
```

The instruction `nop` does nothing, `drop` removes an operand from the stack, `select` picks one of two operands depending on a condition value. The instruction `t.const c` pushed the constant `c` to the stack. The remaining instructions access local and global variables.

CHAPTER
TWO

VALIDATION

NanoWasm instructions are *type-checked* under a context that assigns types to indices:

$$\text{context} ::= \{\text{globals } \text{globaltype}^*, \text{locals } \text{valtype}^*\}$$

2.1 nop

nop is valid with $\epsilon \rightarrow \epsilon$.

$$\overline{C \vdash \text{nop} : \epsilon \rightarrow \epsilon}$$

2.2 drop

drop is valid with $t \rightarrow \epsilon$.

$$\overline{C \vdash \text{drop} : t \rightarrow \epsilon}$$

2.3 select

select is valid with $t \ t \ \text{i32} \rightarrow t$.

$$\overline{C \vdash \text{select} : t \ t \ \text{i32} \rightarrow t}$$

2.4 const

$(t.\text{const } c)$ is valid with $\epsilon \rightarrow t$.

$$\overline{C \vdash t.\text{const } c : \epsilon \rightarrow t}$$

2.5 local.get

$(\text{local.get } x)$ is valid with $\epsilon \rightarrow t$ if:

- $C.\text{locals}[x]$ exists.
- $C.\text{locals}[x]$ is of the form t .

$$\frac{C.\text{locals}[x] = t}{C \vdash \text{local.get } x : \epsilon \rightarrow t}$$

2.6 local.set

$(\text{local.set } x)$ is valid with $t \rightarrow \epsilon$ if:

- $C.\text{locals}[x]$ exists.
- $C.\text{locals}[x]$ is of the form t .

$$\frac{C.\text{locals}[x] = t}{C \vdash \text{local.set } x : t \rightarrow \epsilon}$$

2.7 global.get

$(\text{global.get } x)$ is valid with $\epsilon \rightarrow t$ if:

- $C.\text{globals}[x]$ exists.
- $C.\text{globals}[x]$ is of the form $(\text{mut}^? t)$.

$$\frac{C.\text{globals}[x] = \text{mut}^? t}{C \vdash \text{global.get } x : \epsilon \rightarrow t}$$

2.8 global.set

$(\text{global.get } x)$ is valid with $t \rightarrow \epsilon$ if:

- $C.\text{globals}[x]$ exists.
- $C.\text{globals}[x]$ is of the form $(\text{mut } t)$.

$$\frac{C.\text{globals}[x] = \text{mut } t}{C \vdash \text{global.get } x : t \rightarrow \epsilon}$$

EXECUTION

NanoWasm execution requires a suitable definition of state and configuration:

$$\begin{aligned}
 \textit{addr} &::= 0 \mid 1 \mid 2 \mid \dots \\
 \textit{moduleinst} &::= \{\text{globals } \textit{addr}^*\} \\
 \textit{val} &::= \text{const } \textit{valtype} \text{ const} \\
 \textit{store} &::= \{\text{globals } \textit{val}^*\} \\
 \textit{frame} &::= \{\text{locals } \textit{val}^*, \text{module } \textit{moduleinst}\} \\
 \textit{state} &::= \textit{store}; \textit{frame} \\
 \textit{config} &::= \textit{state}; \textit{instr}^*
 \end{aligned}$$

We define the following auxiliary functions for accessing and updating the state:

$$\begin{aligned}
 \text{local}((s; f), x) &= f.\text{locals}[x] \\
 \text{global}((s; f), x) &= s.\text{globals}[f.\text{module}. \text{globals}[x]] \\
 \text{update}_{\text{local}}((s; f), x, v) &= s; f[\text{locals}[x] = v] \\
 \text{update}_{\text{global}}((s; f), x, v) &= s[\text{globals}[f.\text{module}. \text{globals}[x]] = v]; f
 \end{aligned}$$

With that, execution is defined as follows:

3.1 nop

1. Do nothing.

$$\text{nop} \rightarrow \epsilon$$

3.2 drop

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value \textit{val} from the stack.

$$\textit{val} \text{ drop} \rightarrow \epsilon$$

3.3 select

1. Assert: Due to validation, a value of valtype i32 is on the top of the stack.
2. Pop the value (i32.const c) from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value \textit{val}_2 from the stack.
5. Assert: Due to validation, a value is on the top of the stack.
6. Pop the value \textit{val}_1 from the stack.

7. If $c \neq 0$, then:
 - a. Push the value val_1 to the stack.
8. Else:
 - a. Push the value val_2 to the stack.

$$\begin{aligned} val_1 \ val_2 \ (\text{i32.const } c) \ \text{select} &\hookrightarrow val_1 \quad \text{if } c \neq 0 \\ val_1 \ val_2 \ (\text{i32.const } c) \ \text{select} &\hookrightarrow val_2 \quad \text{otherwise} \end{aligned}$$

3.4 local.get x

1. Let z be the current state.
2. Let val be $\text{local}(z, x)$.
3. Push the value val to the stack.

$$z; (\text{local.get } x) \hookrightarrow z; val \quad \text{if } val = \text{local}(z, x)$$

3.5 local.set x

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value val from the stack.

$$z; val \ (\text{local.set } x) \hookrightarrow z'; \epsilon \quad \text{if } z' = \text{update}_{\text{local}}(z, x, val)$$

3.6 global.get x

1. Let z be the current state.
2. Let val be $\text{global}(z, x)$.
3. Push the value val to the stack.

$$z; (\text{global.get } x) \hookrightarrow z; val \quad \text{if } val = \text{global}(z, x)$$

3.7 global.set x

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value val from the stack.

$$z; val \ (\text{global.set } x) \hookrightarrow z'; \epsilon \quad \text{if } z' = \text{update}_{\text{global}}(z, x, val)$$

**CHAPTER
FOUR**

BINARY FORMAT

The following grammars define the binary representation of NanoWasm programs.

First, constants are represented in LEB format:

```
byte ::= 0x00 | ... | 0xFF
u(N) ::= n:byte          ⇒ n           if n < 27 ∧ n < 2N
       | n:byte m:u(N - 7) ⇒ 27 · m + (n - 27) if n ≥ 27 ∧ N > 7
u32 ::= u(32)
u64 ::= u(64)
f(N) ::= b*:byteN/8      ⇒ float(N, b*)
f32 ::= f(32)
f64 ::= f(64)
```

Types are encoded as follows:

```
valtype ::= 0x7F          ⇒ i32
          | 0x7E          ⇒ i64
          | 0x7D          ⇒ f32
          | 0x7C          ⇒ f64
mut ::= 0x00            ⇒ ε
       | 0x01          ⇒ mut
globaltype ::= t:valtype mut:mut    ⇒ mut t
resulttype ::= n:u32 (t:valtype)n   ⇒ tn
functype ::= 0x60 t1*:resulttype t2*:resulttype ⇒ t1* → t2*
```

Finally, instruction opcodes:

```
globalidx ::= x:u32        ⇒ x
localidx ::= x:u32        ⇒ x
instr ::= 0x01            ⇒ nop
         | 0x1A            ⇒ drop
         | 0x1B            ⇒ select
         | 0x20 x:localidx ⇒ local.get x
         | 0x21 x:localidx ⇒ local.set x
         | 0x23 x:globalidx ⇒ global.get x
         | 0x24 x:globalidx ⇒ global.set x
         | 0x41 n:u32        ⇒ i32.const n
         | 0x42 n:u64        ⇒ i64.const n
         | 0x43 p:f32        ⇒ f32.const p
         | 0x44 p:f64        ⇒ f64.const p
```