
NanoWasm Specification

The SpecTec Team

Feb 05, 2025

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 <i>x</i>	8
3.5	local.set <i>x</i>	8
3.6	global.get <i>x</i>	8
3.7	global.set <i>x</i>	8
4	Binary Format	9

NanoWasm is a small language with simple types and instructions.

ABSTRACT SYNTAX

The *abstract syntax* of types is as follows:

$$\begin{aligned} mut &::= \text{mut} \\ valtype &::= i32 \mid i64 \mid f32 \mid f64 \\ functype &::= valtype^* \rightarrow valtype^* \\ globaltype &::= mut^? valtype \end{aligned}$$

Instructions take the following form:

$$\begin{aligned} const &::= 0 \mid 1 \mid 2 \mid \dots \\ instr &::= \begin{array}{l} \text{nop} \\ | \\ \text{drop} \\ | \\ \text{select} \\ | \\ valtype.\text{const } const \\ | \\ \text{local.get } localidx \\ | \\ \text{local.set } localidx \\ | \\ \text{global.get } globalidx \\ | \\ \text{global.set } globalidx \end{array} \end{aligned}$$

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.

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 equal to t .

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

2.6 local.set

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

- $C.\text{locals}[x]$ exists.
- $C.\text{locals}[x]$ is equal to t .

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

2.7 global.get

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

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

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

2.8 global.set

(global.set x) is valid with $t \rightarrow \epsilon$ if:

- $C.\text{globals}[x]$ exists.
- $C.\text{globals}[x]$ is equal to $(\text{mut } t)$.

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

EXECUTION

NanoWasm execution requires a suitable definition of state and configuration:

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

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

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

With that, execution is defined as follows:

3.1 nop

1. Do nothing.

$$\text{nop} \hookrightarrow \epsilon$$

3.2 drop

1. Assert: Due to validation, a value is on the top of the stack.
2. Pop the value val from the stack.
3. Do nothing.

$$\text{val drop} \hookrightarrow \epsilon$$

3.3 select

1. Assert: Due to validation, a value type is on the top of the stack.
2. Pop the value $(\text{i32.const } c)$ from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop the value val_2 from the stack.
5. Assert: Due to validation, a value is on the top of the stack.

6. Pop the value 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.

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

3.4 local.get x

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

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

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.
3. Do nothing.

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

3.6 global.get x

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

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

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.
3. Do nothing.

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

BINARY FORMAT

The following grammars define the binary representation of NanoWasm programs.

First, constants are represented in LEB format:

$$\begin{aligned}
 \text{byte} &::= b:0x00 \mid \dots \mid b:0xFF &\Rightarrow b \\
 \text{u}(N) &::= n:\text{byte} &\Rightarrow n &\quad \text{if } n < 2^7 \wedge n < 2^N \\
 &\mid n:\text{byte } m:\text{u}(N-7) &\Rightarrow 2^7 \cdot m + (n - 2^7) &\quad \text{if } n \geq 2^7 \wedge N > 7 \\
 \text{u32} &::= n:\text{u}(32) &\Rightarrow n \\
 \text{u64} &::= n:\text{u}(64) &\Rightarrow n \\
 \text{f}(N) &::= b*:\text{byte}^{N/8} &\Rightarrow \text{float}(N, b^*) \\
 \text{f32} &::= p:\text{f}(32) &\Rightarrow p \\
 \text{f64} &::= p:\text{f}(64) &\Rightarrow p
 \end{aligned}$$

Types are encoded as follows:

$$\begin{aligned}
 \text{valtype} &::= 0x7F &\Rightarrow i32 \\
 &\mid 0x7E &\Rightarrow i64 \\
 &\mid 0x7D &\Rightarrow f32 \\
 &\mid 0x7C &\Rightarrow f64 \\
 \text{mut} &::= 0x00 &\Rightarrow \epsilon \\
 &\mid 0x01 &\Rightarrow \text{mut} \\
 \text{globaltype} &::= t:\text{valtype } mut:\text{mut} &\Rightarrow mut \ t \\
 \text{resulttype} &::= n:\text{u32 } (t:\text{valtype})^n &\Rightarrow t^n \\
 \text{functype} &::= 0x60 \ t_1^*:\text{resulttype } t_2^*:\text{resulttype} &\Rightarrow t_1^* \rightarrow t_2^*
 \end{aligned}$$

Finally, instruction opcodes:

$$\begin{aligned}
 \text{globalidx} &::= x:\text{u32} &\Rightarrow x \\
 \text{localidx} &::= x:\text{u32} &\Rightarrow x \\
 \text{instr} &::= 0x01 &\Rightarrow \text{nop} \\
 &\mid 0x1A &\Rightarrow \text{drop} \\
 &\mid 0x1B &\Rightarrow \text{select} \\
 &\mid 0x20 \ x:\text{localidx} &\Rightarrow \text{local.get } x \\
 &\mid 0x21 \ x:\text{localidx} &\Rightarrow \text{local.set } x \\
 &\mid 0x23 \ x:\text{globalidx} &\Rightarrow \text{global.get } x \\
 &\mid 0x24 \ x:\text{globalidx} &\Rightarrow \text{global.set } x \\
 &\mid 0x41 \ n:\text{u32} &\Rightarrow i32.\text{const } n \\
 &\mid 0x42 \ n:\text{u64} &\Rightarrow i64.\text{const } n \\
 &\mid 0x43 \ p:\text{f32} &\Rightarrow f32.\text{const } p \\
 &\mid 0x44 \ p:\text{f64} &\Rightarrow f64.\text{const } p
 \end{aligned}$$