6. A Stack Architecture as Target

Emil Sekerinski, McMaster University, updated April 2022

> This notebook depends on three packages that need to be installed if the notebook is run locally:
>
> - WABT, the WebAssembly Binary Toolkit
> - pywasm, a WebAssembly interpreter written in Python
> - Wasmer, a collection of WebAssembly compilers with a Python embedding

This chapter extends the P0 compiler with code generation for WebAssembly. Like CLI (Common Language Infrastructure, the standard used for .NET) and JVM (Java Virtual Machine), WebAssembly is a *virtual* architecture: WebAssembly code needs either to be interpreted or compiled to *native* code. At the time of writing, WebAssembly can be interpreted with pywasm, can be compiled through Wasmer with a number of compilers, and can be executed in all common web browsers by *just-in-time* compilation. However, there is nothing web-specific about WebAssembly: its purpose is to allow safe and efficient execution of code that can originate from untrusted sources.

WebAssembly differs from the currently dominant *RISC* (Reduced Instruction Set Computer) architectures in a number of ways:

- *Stack architecture*: all operands of operations have to be pushed on a stack, there are no registers.
- *Byte code*: Instructions are encoded as single bytes, rather than whole words.
- *Statically typed*: type-checking ensures that the types of operands and operations match.
- *Block-structured*: there are constructs for if-statements and loops; jumps to arbitrary locations are not possible.
- *Non-uniform store*: rather than having memory as a flat sequenced of addressable bytes, there are several stores that are all addressed differently.

These notes introduce WebAssembly to the extent needed for P0.

WebAssembly programs are composed of *modules* that may depend on other modules and may interact with a *host environment*, like a web browser or another programming language. Modules are stored in two equivalent and mutually convertible forms,

- in binary .wasm files, with a flat sequences of instructions that each takes one byte,
- in textual .wat files with instructions in textual form where some locations can be names rather than numbers.

For readability we use the textual form and return to the details of the binary form.

WebAssembly Stores

WebAssembly programs distinguish different kind of stores,

- the *code*, where the program resides,
- the *memory*, where variables with computed locations (e.g. arrays) are stored,
- the *global* store, where initialized global variables and constants are stored,
- the *stack*, where *operands* of instructions, *call frames* with *local variables*, and certain *labels* are stored.

The code is *immutable*: there is no way to manipulate the code once it is loaded by the virtual machine. While we think of the code as being the WebAssembly program itself, an implementation is free to represent the programs in another form or to compile the program to machine code; except for efficiency, this would not be observable.

The code is made up of a collection of procedures, called *functions*. Functions are numbered and each function is a sequence of instructions. If a program has `FN` functions, the code is abstractly represented by:

```
var c: [0 .. FN) → seq(byte)
```

The memory is a contiguously addressed array of bytes; its size is specified in multiples of *pages* of 2^{16} bytes. Its initial size is specified in a WebAssembly file but can explicitly grow. Bytes in the memory can be accessed arbitrarily. Abstractly, the memory is:

```
var m: [0 .. MemSize) → byte
```

In WebAssembly, the value and result parameters of functions are typed as are local variables, global variables, and intermediate results on the stack. The only exception is the memory, which allows bytes to be interpreted as needed. The only types are 32 and 64 bit integer and floating-point numbers: `i32`, `i64`, `f32`, `f64`. There are no booleans, enumerations, arrays, tuples, lists, records, unions, or classes. These have to be either expressed in terms of supported types or mapped to the memory.

The global store is an array of typed values, which are one of `i32`, `i64`, `f32`, `f64`.

```
var g: [0 .. GlobalVars) → Value
```

The stack can only be accessed through specific arithmetic instructions and control flow instructions. Abstractly, the stack is an array of values; the *stack pointer* `sp` points to the next available entry:

```
var s: [0 .. StackSize) → Value
var sp: [0 .. StackSize] = 0
```

WebAssembly Instructions

Operands of instructions have first to be pushed on the stack. The instructions to push a constant on the stack are `i32.const n`, `i64.const n`, `f32.const f`, `f64.const f`. Arithmetic instructions specify the type of the operands, for example for addition `i32.add`, `i64.add`, `f32.add`, `f64.add`. For P0 the sole WebAssembly type in the generated code is `i32`, so the P0 expression `3 + 4` corresponds to:

```
i32.const 3
i32.const 4
i32.add
```

To illustrate typing, the code

```
i32.const 3
i64.const 4
i32.add
```

is not type-correct: the *validation* of a WebAssembly program ensures that entries of the same type are popped as were pushed, which WebAssembly allows to do statically. That is, above code will be rejected when loaded into the virtual machine. Likewise, trying to pop from the empty stack is also detected during validation, e.g. if a program starts with:

```
i32.const 3
i32.add
```

The `if ... else ... end` instruction pops the top element from the stack as the condition. If it is not zero, the instructions following `if` are executed, otherwise those following `else`. For example, if `x`, `y`, `m` are local variable, following P0 statement can be translated as to the right.

```
if x > y then
    m := x
else
    m := y
```

```
local.get $x
local.get $y
i32.gt_s
if
  local.get $x
  local.set $m
else
  local.get $y
  local.set $m
end
```

Consider following P0 procedure and its WebAssembly translation to the right:

```
procedure QuotRem(x, y: integer)
    var q, r: integer
      q := 0; r := x
      while r ≥ y do
        r := r - y; q := q + 1
      write(q); write(r)
```

WebAssembly functions can take value and result parameters. Names have to be prefixed with `$`. The code declares function `$QuotRem` with value parameters `$x` and `$y` and local variables `$q` and `$r`, all of type `i32`. Parameters and local variable are accessed identically with `local.set` and `local.get` instructions: `local.set x` removes the top element from the stack and stores it at the location named `x` on the stack, which must have been declared to be of the same type; `local.get x` reads the local variable from the location named `x` on the stack and pushed it on top of the stack.

```
(func $QuotRem (param $x i32) (param $y i32)
  (local $q i32)
  (local $r i32)
  i32.const 0
  local.set $q
  local.get $x
  local.set $r
  loop $label0
    local.get $r
    local.get $y
    i32.ge_s
    if
      local.get $r
      local.get $y
      i32.sub
      local.set $r
      local.get $q
      i32.const 1
      i32.add
      local.set $q
      br $label0
    end
  end
  local.get $q
  call $write
  local.get $r
  call $write
)
```

WebAssembly allows branches to structured control instructions. If `l` is a name,

- within `block l ... end` a branch instruction `br l` will transfer control to the end of the construct, i.e. the instruction following `end`;
- within `loop l ... end` a branch instruction `br l` will transfer control to the beginning of the construct, i.e. the instruction following `loop l`.

WebAssembly allows `i32` values to be interpreted as either signed or unsigned 32-bit integers: the instruction `i32.lt_s` performs a signed comparison and pushes `1` or the stack if the result is true and `0` otherwise. The instruction `br_if l` transfers control to the `block` or `loop` instruction labelled `l` if the top of the stack is not zero. Note that it is impossible to branch inside a `block` or `loop`, branches can only go outwards.

Functions are called by pushing first all arguments on the stack, calling the function, and the popping the results from the stack. Thus the call `write(q)` first pushes `q` and then calls `$write`, which is assumed to be defined elsewhere.

The grammar for WebAssembly function definitions, to the extent needed for P0, is:

```
num ::= digit {digit}
int ::= [ '-' ] num
name ::= '$' (letter | digit) {letter | digit}
num_type ::= "i32" | "i64"
func ::= "(" "func" name func_type { local } { instr } ")"
func_type ::= { "(" "param" name num_type ")" } { "(" "result" num_type ")" }
local ::= "(" "local" name num_type ")"
instr ::= "i32.const" int | "i64.const" int |
          "i32.add" | "i64.add" | "i32.sub" | "i64.sub" | "i32.mul" | "i64.mul" |
          "i32.div_s" | "i64.div_s" | "i32.rem_s" | "i64.rem_s" |
          "i32.eqz" | "i64.eqz" | "i32.eq" |"i64.eq" | "i32.ne" | "i64.ne" |
          "i32.lt_s" | "i64.lt_s" | "i32.gt_s" | "i64.gt_s" |
          "i32.le_s" | "i64.le_s" | "i32.ge_s" | "i64.ge_s" |
          "i32.load" "offset" "=" num | "i32.load" "offset" "=" num |
          "i32.store" "offset" "=" num | "i32.store" "offset" "=" num |
          "global.get" name |
          "global.set" name |
          "local.get" name |
          "local.set" name |
          "block" name { instr } "end" |
          "loop" name { instr } "end" |
          "if" { instr } [ "else" { instr } ] "end" |
          "br" name |
          "br_if" name |
          "return" |
          "call" name
```

The effect of arithmetic instructions and load/store instructions on `m`, `g`, `s` can be described by assignment statements. Let `i` be an integer, `x` is the name of a local or global variable, and `loc(x)` the index of variable `x` on the stack, to be made precise later:

instruction	effect	trap condition
`i32/64.const i`	`s[sp], sp := i, sp + 1`	`sp < StackSize`
`i32/64.add`	`s[sp - 2], sp := s[sp - 2] + s[sp - 1], sp - 1`	
`i32/64.sub`	`s[sp - 2], sp := s[sp - 2] - s[sp - 1], sp - 1`	
`i32/64.mul`	`s[sp - 2], sp := s[sp - 2] × s[sp - 1], sp - 1`	
`i32/64.div_s`	`s[sp - 2], sp := s[sp - 2] div s[sp - 1], sp - 1`	`s[sp - 1] = 0`
`i32/64.rem_s`	`s[sp - 2], sp := s[sp - 2] mod s[sp - 1], sp - 1`	`s[sp - 1] = 0`
`i32/64.eqz`	`s[sp - 1] := s[sp - 1] = 0`	
`i32/64.eq`	`s[sp - 2], sp := s[sp - 2] = s[sp - 1], sp - 1`	
`i32/64.ne`	`s[sp - 2], sp := s[sp - 2] ≠ s[sp - 1], sp - 1`	
`i32/64.lt_s`	`s[sp - 2], sp := s[sp - 2] < s[sp - 1], sp - 1`	
`i32/64.gt_s`	`s[sp - 2], sp := s[sp - 2] > s[sp - 1], sp - 1`	
`i32/64.le_s`	`s[sp - 2], sp := s[sp - 2] ≤ s[sp - 1], sp - 1`	
`i32/64.ge_s`	`s[sp - 2], sp := s[sp - 2] ≥ s[sp - 1], sp - 1`	
`i32/64.load offset = n`	`s[sp - 1] := m[s[sp - 1] + n]`	`0 ≤ s[sp - 1] + n < MemSize`
`i32/64.store offset = n`	`m[s[sp - 2] + n], sp := s[sp - 1], sp - 2`	`0 ≤ s[sp - 1] + n < MemSize`
`local.get x`	`s[sp], sp := s[loc(x)], sp + 1`	
`local.set x`	`s[loc(x)], sp := s[sp - 1], sp - 1`	
`global.get x`	`s[sp], sp := g[loc(x)], sp + 1`	
`global.set x`	`g[loc(x)], sp := s[sp - 1], sp - 1`	

Pushing on a full stack, dividing by zero, or accessing the memory outside its bounds is an error. In that case, the program *traps*, meaning that it terminates and passes control to its environment.

A WebAssembly module can import functions, declare initialized global variables, define functions, declare the initial size of the memory, and designate one function as the start function that is executed when the module is loaded. The grammar is:

```
string ::= \" {char} \"
module ::= "(" "module" {import} {global} {func} [memory] [start] ")"
import ::= "(" "import" string string "(" "func" name func_type ")" ")"
global ::= "(" "global" name "(" "mut" num_type ")" instr ")"
memory ::= "(" "memory" num ")"
start ::= "(" "start" name ")"
```

In global variable declarations, the keyword `mut` specifies that the variable is mutable, otherwise it is a constant. The subsequent instructions have to initialize the variable with a value of matching type.

Executing WebAssembly

For P0 programs, we specify the following in WebAssembly modules:

- The standard library consists of the procedures `write`, `writeln`, and `read`; these have to be imported from the host environment.
- The size of the memory is specified as 1 page with 2^{16} bytes.
- The main program in P0 translates to a function `$program` in WebAssembly and is designated as a the start function.

```
procedure QuotRem(x, y: integer)
    var q, r: integer
        q := 0; r := x
        while r ≥ y do // q × y + r = x ∧ r ≥ y
            r := r - y; q := q + 1
        write(q); write(r)

program arithmetic
  var x, y: integer
    x ← read(); y ← read()
    QuotRem(x, y)
```

```
(module
  (import "P0lib" "write" (func $write (param i32)))
  (import "P0lib" "writeln" (func $writeln))
  (import "P0lib" "read" (func $read (result i32)))
  (func $QuotRem (param $x i32) (param $y i32)
  ...
  )
  (func $program
    (local $x i32)
    (local $y i32)
    call $read
    local.set $x
    call $read
    local.set $y
    local.get $x
    local.get $y
    call $QuotRem
    )
  (memory 1)
  (start $program)
)
```

WebAssembly programs can be run in a web browser through a JavaScript extension that is currently supported by all main web browsers. For this, the P0 standard library has to be implemented in JavaScript so it can be imported in WebAssembly. The library is a JavaScript structure `P0lib` with fields `write`, `writeln`, and `read`, which are all JavaScript functions. That structure is then collected with potentially other parameters (e.g. other libraries) in one JavaScript structure `params`:

```
const params = {
        P0lib: {
            write: i => this.append_stream({text: '' + i, name: 'stdout'}),
            writeln: () => this.append_stream({text: '\\n', name: 'stdout'}),
            read: () => window.prompt()
        }
    }
```

The WebAssembly code is assumed to be in a binary WebAssembly file with url in JavaScript variable `wasmfile`. The JavaScript function `fetch(wasmfile)` reads that file, but does so asynchronously in the background, i.e. does not return the content of the file but rather a `Promise<Response>` that eventually resolves to the `Response` of the `http` request. To read the whole file as a binary sequence, the response has to be converted to an `ArrayBuffer`, as needed for execution with WebAssembly. The `Promise` method `.then` takes a function as a parameter; that function is called with the resolved value of the promise, `response` below, when the promise is resolved successfully (for treatment of errors, a `.catch` method is provided, which won't be used here). The function `WebAssembly.compile` takes an `ArrayBuffer` object, `code` below, and returns the executable module. That module is without state, i.e. can in principle be shared among multiple executions. The function `WebAssembly.instantiate` allocates the memory, sets up the stack, binds imported functions, and calls the start function of the module:

```
fetch(wasmfile)
      .then(response => response.arrayBuffer())
      .then(code => WebAssembly.compile(code))
      .then(module => WebAssembly.instantiate(module, params))
```

JavaScript can be executed in Jupyter notebooks by displaying HTML code with JavaScript in the web browser. The library `IPython.core.display` provides a Python function to that end. The JavaScript code for fetching a WebAssembly file and executing it with the standard P0 library are placed in the Python function `runwasm(wasmfile)`:

In [ ]:

```
def runwasm(wasmfile):
    from IPython.display import display, Javascript
    display(Javascript("""
```

```
    const params = {
        P0lib: {
            write: i => this.append_stream({text: '' + i, name: 'stdout'}),
            writeln: () => this.append_stream({text: '\\n', name: 'stdout'}),
            read: () => window.prompt()
        }
    }
    fetch('""" + wasmfile + """') // asynchronously fetch file, return Response object
      .then(response => response.arrayBuffer()) // read the response to completion and stores it in an ArrayBuf
      .then(code => WebAssembly.compile(code)) // compile (sharable) code.wasm
      .then(module => WebAssembly.instantiate(module, params)) // create an instance with memory
    // .then(instance => instance.exports.program()); // run the main program; not needed if start function spe
    """))
```

For example, the complete textual WebAssembly file corresponding to the P0 program `arithmetic` is:

In [ ]:
```
%%writefile arithmetic.wat
(module
  (import "P0lib" "write" (func $write (param i32)))
  (import "P0lib" "writeln" (func $writeln))
  (import "P0lib" "read" (func $read (result i32)))
  (func $QuotRem (param $x i32) (param $y i32)
    (local $q i32)
    (local $r i32)
    i32.const 0
    local.set $q
    local.get $x
    local.set $r
    loop $label0
      local.get $r
      local.get $y
      i32.ge_s
      if
        local.get $r
        local.get $y
        i32.sub
        local.set $r
        local.get $q
        i32.const 1
        i32.add
        local.set $q
        br $label0
      end
    end
    local.get $q
    call $write
    local.get $r
    call $write
  )
  (func $program
    (local $x i32)
    (local $y i32)
    call $read
    local.set $x
    call $read
    local.set $y
    local.get $x
    local.get $y
    call $QuotRem
  )
  (memory 1)
  (start $program)
)
```

That has to be converted to a binary form for execution:

In [ ]:
```
!wat2wasm arithmetic.wat
```

Now the generate code can be executed. Note that the WebAssembly code runs natively on the computer on which the web browser runs, not on the Jupyter server, where the Python kernel runs:

In [ ]:
```
runwasm("arithmetic.wasm")
```

Alternatively to running WebAssembly programs in the browser, programs can be interpreted by `pywasm`. In this case, Python is the host enviroment and provides an implementation of the standard library:

In [ ]:
```
def runpywasm(wasmfile):
    def write(s, i): print(i)
    def writeln(s): print()
    def read(s): return int(input())
```

```
    import pywasm
    vm = pywasm.load(wasmfile, {'P0lib': {'write': write, 'writeln': writeln, 'read': read}})
```

In [ ]:
```
runpywasm("arithmetic.wasm")
```

The third option is to use `Wasmer`, which supports several compilers for compiling wasm to native code, including LLVM. The cell below uses the `cranelift` compiler, which reportedly run faster than LLVM, but does not generate as efficient code. With the Python binding of `Wasmer`, Python can the be host environment:

In [ ]:
```
from wasmer import engine, Store, Module, Instance, ImportObject, Function
from wasmer_compiler_cranelift import Compiler

def runwasmer(wasmfile):
    def write(i: int): print(i)
    def writeln(): print()
    def read() -> int: return int(input())
    store = Store(engine.JIT(Compiler))
    module = Module(store, open(wasmfile, 'rb').read())
    import_object = ImportObject()
    import_object.register("P0lib", {"write": Function(store, write),
                                    "writeln": Function(store, writeln),"read": Function(store, read)})
    instance = Instance(module, import_object)
```

In [ ]:
```
runwasmer("arithmetic.wasm")
```

Translation Scheme for Expressions

The translation scheme for arithmetic expressions is:

E	code(E)	condition
x	local.get \$x	if x local variable
x	global.get \$x	if x global variable
n	i32.const n	if n integer constant
$E_1 \times E_2$	code(E_1) code(E_2) i32.mul	
E_1 div E_2	code(E_1) code(E_2) i32.div_s	
E_1 mod E_2	code(E_1) code(E_2) i32.rem_s	
+ E	code(E)	
- E	i32.const 0 code(E) i32.sub	
$E_1 + E_2$	code(E_1) code(E_2) i32.add	
$E_1 - E_2$	code(E_1) code(E_2) i32.sub	

For boolean expression, the translation scheme is analogous, except that no code is generated for the negation of a relational operation, only the relation is negated:

E	code(E)	E	cond(E)
x	code(x)	not x	i32.const 1 code(x) i32.sub
$E_1 = E_2$	code(E_1) code(E_2) i32.eq	not($E_1 = E_2$)	code(E_1) code(E_2) i32.ne
$E_1 \neq E_2$	code(E_1) code(E_2) i32.ne	not($E_1 \neq E_2$)	code(E_1) code(E_2) i32.eq
$E_1 < E_2$	code(E_1) code(E_2) i32.lt_s	not($E_1 < E_2$)	code(E_1) code(E_2) i32.ge_s
$E_1 \leq E_2$	code(E_1) code(E_2) i32.le_s	not($E_1 \leq E_2$)	code(E_1) code(E_2) i32.gt_s
	code(E_1)		code(E_1)

$E_1 > E_2$	code(E_2) i32.gt_s	not($E_1 > E_2$)	code(E_2) i32.le_s
$E_1 \geq E_2$	code(E_1) code(E_2) i32.ge_s	not($E_1 \geq E_2$)	code(E_1) code(E_2) i32.lt_s

Translation Scheme for Statements and Declarations

The translation scheme for P0 statements is:

S	code(S)	
x_1, …, x_n := E_1, …, E_n	code(E_1) … code(E_n) set \$$x_n$ … set \$$x_1$	`set` is `local.set` for local variable and `global.set` for global variable
x_1, …, x_m ← p(E_1, …, E_n)	code(E_1) … code(E_n) call \$p set \$$x_m$ … set \$$x_1$	`set` is `local.set` for local variable and `global.set` for global variable
S_1; …; S_n	code(S_1) … code(S_n)	
if E then S	code(E) if code(S) end	

if E then S_1 else S_2	code(E)
	if
	  code(S_1)
	else
	  code(S_2)
	end	

while E do S	loop \$L
	  code(E)
	  if
	    code(S)
	    br \$L
	  end
	end	

The translation scheme for declarations is:

D	code(D)	
`var x: integer`	`(local $x i32)`	for local declaration
`var x: boolean`	`(local $x i32)`	for local declaration
`var x: integer`	`(global $x (mut i32) i32.const 0)`	for global declaration
`var x: boolean`	`(global $x (mut i32) i32.const 0)`	for global declaration

procedure p(v_1: T_1, … , v_n: T_n) → (r_1: U_1, … , r_m: U_m)
  D
  S
	(func \$p (param \$$v_1$ i32) … (param \$$v_n$ i32)
	  (result i32) … (result i32)
	  (local \$$r_1$ i32)
	  …
	  (local \$$r_m$ i32)
	  code(D)
	  code(S)
	  local.get \$$r_1$
	  …
	  local.get \$$r_m$
	)
		if all T_i, U_j are `integer` or `boolean`

…
program n
  D
  S
	(module
	  stdlibimport
	  …
	  (func \$program
	    code(D)
	    code(S)
	  )
	)	

Above, `stdlibimports` stands for the import of the P0 standard library, which is:

```
(import "P0lib" "write" (func $write (param i32)))
(import "P0lib" "writeln" (func $writeln))
(import "P0lib" "read" (func $read (result i32)))
```

Binary WebAssembly Files

It is instructive to "reverse engineer" the binary WebAssembly file by converting it back to the textual form. In WebAssembly, comments are written as `(;comment;)` and by `;;comment` for comments that extend until the end of the line:

In [ ]:
```
!wasm2wat arithmetic.wasm
```

A copy of the output is to the right; it reveals what is stored in the binary format:

- Function parameters and local variables are referred to by numbers starting with 0, rather than by names. That is, in

```
procedure QuotRem(x, y: integer)
    var q, r: integer
      q := 0; r := x
      while r ≥ y do
        r := r - y; q := q + 1
      write(q); write(r)
```

variables `x`, `y`, `q`, `r` are referred to by `0` to `3`, respectively.

- Functions are referred to by numbers rather than names: functions `write`, `writeln`, `read`, `QuotRem`, `program` are referred to by `0` to `4`, respectively.
- The function types are also numbered and referred to by their position number.
- The targets of `br` and `br_if` refer to the enclosing `block ... end`, `loop ... end`, `if ... end` by number: the closest one is `0`, the next closest one is `1`, etc. Recall that the only branches allowed to outer `block` and `end` instructions.

```
(module
  (type (;0;) (func (param i32)))
  (type (;1;) (func))
  (type (;2;) (func (result i32)))
  (type (;3;) (func (param i32 i32)))
  (import "P0lib" "write" (func (;0;) (type 0)))
  (import "P0lib" "writeln" (func (;1;) (type 1)))
  (import "P0lib" "read" (func (;2;) (type 2)))
  (global (;0;) (mut i32) (i32.const 0))
  (global (;1;) (mut i32) (i32.const 0))
  (func (;3;) (type 3) (param i32 i32)
    (local i32 i32)
    i32.const 0
    local.set 2
    local.get 0
    local.set 3
    loop  ;; label = @1
      local.get 3
      local.get 1
      i32.ge_s
      if  ;; label = @2
        local.get 3
        local.get 1
        i32.sub
        local.set 3
        local.get 2
        i32.const 1
        i32.add
        local.set 2
        br 1 (;@1;)
      end
    end
    local.get 2
    call 0
    local.get 3
    call 0)
  (func (;4;) (type 1)
    call 2
    global.set 0
    call 2
    global.set 1
    global.get 0
    global.get 1
    call 3)
  (memory (;0;) 1)
  (start 4))
```

Translation Scheme for Boolean Operators

In P0 boolean operators `and` and `or` evaluate conditionally:

```
p and q = if p then q else false
p or q = if p then true else q
```

As soon as the result is determined, the remaining operands are not evaluated. This way, expressions like

```
(i < N) and (a[i] ≠ x)
(y = 0) or (x div y = m)
```

will not evaluate the second half if the first half determines the result.

The WebAssembly `if` instruction is used for conditional evaluation:

```
if (a < b)       code(a)            if (a < b)      code(a)
and (c = d)      code(b)            or (c = d)      code(b)
and (e ≥ f)      i32.lt_s           or (e ≥ f)      i32.lt_s
then S           if (result i32)    then S          if (result i32)
                  code(c)                            i32.const 1
                  code(d)                           else
                  i32.eq                             code(c)
```

```
else
 i32.const 0
end
if (result i32)
 code(e)
 code(f)
 i32.ge_s
else
 u32.const 0
end
if
 code(S)
end
```

```
 code(d)
 i32.eq
end
if (result i32)
 i32.const 1
else
 code(e)
 code(f)
 i32.ge_s
end
if
 code(S)
end
```

E	code(E)
`not E`	`code(E)` `i32.eqz`

`E₁ and E₂`	`code(E₁)`
	`if (result i32)`
	`  code(E₂)`
	`else`
	`  i32.const 0`
	`end`

`E₁ or E₂`	`code(E₁)`
	`if (result i32)`
	`  i32.const 1`
	`else`
	`  code(E₂)`
	`end`

This translation scheme generates for `if a = b and c > d then S` equivalent code as for `if a = b then if c > d then S`:

`if a = b`
`and c > d`
`then S`
	`get a`
	`get b`
	`i32.eq`
	`if`
	`  get c`
	`  get d`
	`  i32.gt_s`
	`else`
	`  i32.const 0`
	`end`
	`if`
	`  code(S)`
	`end`

`if a = b`
`then`
`  if c > d`
`  then S`
	`get a`
	`get b`
	`i32.eq`
	`if`
	`  get c`
	`  get d`
	`  i32.gt_s`
	`  if`
	`    code(S)`
	`  end`
	`end`

This translation scheme leads `0` (for `false`) and `1` (for `true`) to be explicitly pushed on the stack. Alternatively, using the WebAssembly `block` instruction, conditional evaluation can be expressed with forward branches. Note how for conjunctions each condition is negated, for disjunctions only the last condition is negated. For while-statements, the `loop` instruction is used; branches in conditions go to the outer `block` instruction:

```
if (a < b)   block
and (c = d)   code(a)
and (e ≥ f)   code(b)
then S        i32.ge_s
              br_if 0
              code(c)
              code(d)
              i32.ne
              br_if 0
              code(e)
              code(f)
              i32.lt_s
              br_if 0
              code(S)
             end
```

```
if (a < b)  block
or (c = d)   block
or (e ≥ f)    code(a)
then S        code(b)
              i32.lt_s
              br_if 0
              code(c)
              code(d)
              i32.eq
              br_if 0
              code(e)
              code(f)
              i32.lt_s
              br_if 1
            end
```

```
while a < b    block
do S             loop
                   code(a)
                   code(b)
                   i32.ge_s
                   br_if 1
                   code(S)
                   br 0
                 end
               end
```

```
    code(S)
end
```

These observations motivate following translation scheme: for boolean expression `B` that does not contain conditional boolean operators, `code(B)` specifies the WebAssembly instructions; for an expression `B` with a conditional boolean operator, `condcode(B, L)` specifies the WebAssembly instructions that branch to label `L` if the condition is false and "fall through" otherwise:

<table>
<tr><th>S</th><th>code(S)</th></tr>
<tr><td><pre>if B
then S</pre></td><td><pre>block
  condcode(B, 0)
  code(S)
end</pre></td></tr>
<tr><td><pre>if B
then S1
else S2</pre></td><td><pre>block
  block
    condcode(B, 0)
    code(S1)
    br 1
  end
  code(S2)
end</pre></td></tr>
<tr><td><pre>while B
do S</pre></td><td><pre>block
  loop
    condcode(B, 1)
    code(S)
    br 0
  end
end</pre></td></tr>
</table>

<table>
<tr><th>B</th><th>condcode(B, L)</th></tr>
<tr><td><pre>B1
and ...
and Bn</pre></td><td><pre>condcode(not B1, L)
br if L
...
condcode(not Bn, L)
br_if L</pre></td></tr>
<tr><td><pre>B1
or ...
or Bn</pre></td><td><pre>block
  condcode(B1, 0)
  br if 0
  ...
  condcode(Bn-1, 0)
  br if 0
  condcode(not Bn, L + 1)
  br_if L + 1
end</pre></td></tr>
</table>

Translation Scheme for Arrays

Global arrays are statically allocated consecutively in memory. Below, `adr(x) = 0` and `adr(y) = adr(x) + size(A) = 0 + 7 × 4 = 28`:

```
type A = [1.. 7] → integer
var x: A
var y: A
var i: integer
var h: integer
program p
    i := 3
    h := y[i]
    x[i] := 5
```

```
(global $i (mut i32) i32.const 0)
(global $h (mut i32) i32.const 0)
(func $program
    i32.const 3   ;; 3
    global.set $i
    global.get $i
    i32.const 1    ;; x.lower
    i32.sub
    i32.const 4    ;; size(integer)
    i32.mul
    i32.const 28    ;; adr(y)
    i32.add
    i32.load
    global.set $h
    global.get $i  ;; x[i] := 5
    i32.const 1
    i32.sub
    i32.const 4
    i32.mul
    i32.const 0
    i32.add
    i32.const 5
    i32.store
)
```

The compiler uses variable `memsize` to keep track of statically allocated memory; it is initially `0`. No code is generated for a variable declaration, rather the address is stored in field `adr` of the symbol table entry for the variable. With `A = [l .. u] → T`:

D	code(D)	effect

`var x: A`	`x.adr := memsize; memsize := memsize + size(A)`	global declaration

Local arrays are dynamically allocated in memory in a stack that mimics the calling stack. For local array `x`, local variable `$x` of type `i32` points to the address of `x` in memory. Global WebAssembly variable `$memsize` points to the top of the stack. Each procedure has a local variable `$mp` with the pointer the top of the stack before the allocation of local variables. In the procedure *prologue*, `$memsize` is saved in `$mp` and restored in the *epilogue*. Variable `$memsize` is initially the size of all statically allocated arrays:

```
type A = [1.. 7] → integer
var x: A
procedure q()
  var y: A
    y[3] := 5
program p
  var z: A
    q()
```

```
(func $q
    (local $y i32)
    (local $mp i32)
    global.get $memsize
    local.set $mp
    global.get $memsize
    local.tee $y
    i32.const 28
    i32.add
    global.set $memsize
    i32.const 3
    i32.const 1
    i32.sub
    i32.const 4
    i32.mul
    local.get $y
    i32.add
    i32.const 5
    i32.store
    local.get $mp
    global.set $memsize
)
(global $memsize (mut i32) i32.const 28)
(func $program
    (local $z i32)
    (local $mp i32)
    global.get $memsize
    local.set $mp
    global.get $memsize
    i32.const 28
    i32.add
    local.tee $z
    global.set $memsize
    call $q
    local.get $mp
    global.set $memsize
)
```

With `A = [l .. u] → T`, the translation scheme for programs with local array declarations is:

D	code(D)	

`var x: A`	`(local $x i32)`
	`…`
	`global.get $memsize`
	`local.tee $x`
	`i32.const size(A)`
	`i32.add`
	`global.set $memsize`
		local declaration

`procedure p(v₁: T₁, … , vₙ: Tₙ) → (r₁: U₁, … , rₘ: Uₘ)`
`  D`
`  S`
	`(func $p (param $v₁ i32) … (param $vₙ i32)`
	`  (result i32) … (result i32)`
	`  (local $r₁ i32)`
	`  …`
	`  (local $rₘ i32)`
	`  code(D)`
	`  (local $mp i32)`
	`  global.get $memsize`
	`  local.set $mp`
	`  code(S)`
	`  local.get $r₁`
	`  …`
	`  local.get $rₘ`
	`  local.get $mp`
	`  global.set $memsize`
	`)`	

`D₁`
`program n`
`  D₂`
`  S`
	`(module`
	`  stdlibimport`
	`  code(D₁)`
	`  (global $memsize (mut i32) i32.const memsize)`
	`  (func $program`
	`    code(D₂)`
	`    (local $mp i32)`
	`    global.get $memsize`
	`    local.set $mp`	

```
        local.set $mp
        code(S)
        local.get $mp
        global.set $memsize
    )
)
```

With `A = [l .. u] → T`, the translation scheme for array indexing and array assignment is:

E	code(E)	
`x`	`i32.const x.adr`	if `x` global variable
`x`	`local.get $x`	if `x` local variable

`x[E]`	`code(E)`
	`i32.const x.lower`
	`i32.sub`
	`i32.const size(A)`
	`i32.mul`
	`code(x)`
	`i32.add`
	`i32.load`	

S	code(S)

`x[E] := F`	`code(E)`
	`i32.const x.lower`
	`i32.sub`
	`i32.const size(T)`
	`i32.mul`
	`code(x)`
	`i32.add`
	`code(F)`
	`i32.store`

`x1 := x2`	`code(x1)`
	`code(x2)`
	`i32.const size(T)`
	`memory.copy`

If `s`, `d`, `n` of type `i32` are the top elements of the stack, the instruction `memory.copy` copies `n` bytes in memory starting at index `s` to index `d`.

When arrays are passed as value and result parameters, only a pointer to the array in memory is passed:

```
type A = [1.. 7] → integer
type B = [0 .. 1] → A
var b: B
procedure q(x: A) → (y: A)
    y := x
program p
  b[0] ← q(b[1])
```

```
(func $q (param $x i32) (result i32)
    (local $y i32)
    (local $mp i32)
    global.get $memsize
    local.set $mp
    global.get $memsize
    local.tee $y
    i32.const 28
    i32.add
    global.set $memsize
    local.get $y
    local.get $x
    i32.const 28
    memory.copy
    local.get $mp
    global.set $memsize
    local.get $y
)
(global $memsize (mut i32) i32.const 56)
(func $program
    i32.const 0
    i32.const 28
    call $q
    i32.const 28
    memory.copy
)
```

Passing pointers leads to *aliasing* when two pointers point to the same address and modifications through one pointer are visible through the other pointer. To avoid aliasing,

- an array parameter passed by value is a local constant, rather than local variable; it has to be copied to a local variable before it can be updated,
- a global variable that is passed as a parameter must not be accessed as a global variable; this can be simply enforced by not allowing global variables altogether.

```
type A = [1.. 7] → integer
```

```
var x: A
procedure q(y, z: A)
    y[2] := 3; write(x[2])    // writes 3
    write(z[2])    // writes 3
program p
    q(x, x)
```

Currently, P0 does not check for aliasing.

As a note, variations of above scheme are possible:

- If the size of all local arrays can be statically determined, `$memsize` can be incremented by that amount in the procedure prologue and decremented in the epilogue without a need for local variable `$mp`. While not strictly needed in P0, having `$mp` allows more easily extensions with arrays of dynamic size and other dynamic data structures.
- If `x` is an array, in the call `x ← p()` only the address of the result is returned and the caller copies the result to `x`. Alternatively, the callee can perform that copy, leading to less duplication of copying code. For this, the address of `x` has to be passed as an additional parameter, as in the call `p(x)`.

Translation Scheme for Records

Like arrays, records are allocated consecutively in memory. Below, `adr(x) = 0` and `adr(y) = adr(x) + size([1 .. 7] → R) = 7 × size(R) = 7 × 8 = 56`:

```
type R = (f: integer, g: integer)   (func $program
var x: [1 .. 7] → R                     (local $i i32)
var y: R                                i32.const 3  ;; 3
program p                               local.set $i
    var i: integer                      local.get $i
        i := 3                          i32.const 1   ;; x.lower
        x[i].g := 5                     i32.sub
        y.f := 7                        i32.const 8   ;; size(R)
                                        i32.mul
                                        i32.const 0   ;; adr(x)
                                        i32.add
                                        i32.const 4   ;; offset(g)
                                        i32.add
                                        i32.const 5   ;; 5
                                        i32.store
                                        i32.const 56   ;; adr(y) + offset(f)
                                        i32.const 7    ;; 7
                                        i32.store
                                    )
```

No code is generated for a variable declaration, but the address is stored as a field of the symbol table entry for the variable. With `R = (f₁: T₁, f₂: T₂, …)`:

D	code(D)	effect
`var x: R`		`x.adr := memsize; memsize := memsize + size(R)`

Assume that the address of `x` is on the stack. The the code for `x.f` updates that address:

E	code(E)
`x.f`	`i32.const offset(f)` `i32.add`

Translation Scheme for Exceptions

```
try                 try $l0
    write(3)          i32.const 3
    throw             call $write
    write(5)          throw $l0
catch                 i32.const 5
    write(7)          call $write
write(9)            catch
                      drop
                      i32.const 7
                      call $write
                    end
                    i32.const 9
                    call $write
```

In [ ]:
```
%%writefile exception.wat
```

```
(module
  (import "P0lib" "write" (func $write (param i32)))
  (import "P0lib" "writeln" (func $writeln))
  (import "P0lib" "read" (func $read (result i32)))
  (tag $e)
  (func $q (param $x i32) (result i32)
    throw $e
    local.get $x
  )
  (func $program
    try
      i32.const 3
      call $q
      call $write
    catch $e
      i32.const 7
      call $write
    end
  )
  (memory 1)
  (start $program)
)
```

In [ ]:
```
!wat2wasm --enable-exceptions exception.wat
```

To run this in Chrome, Experimental WebAssembly has to be enabled under `chrome://flags`:

In [ ]:
```
runwasm("exception.wasm")
```

```
try                        try $l0
    try                      i32.const 3
        write(3)             call $write
        throw                throw $l0
        write(5)             i32.const 5
    catch                    call $write
        write(7)           catch
    write(9)                 drop
    throw                    i32.const 7
    write(11)                call $write
catch                      end
    write(13)              i32.const 9
write(15)                  call $write
```

In [ ]:
```
%%writefile exception.wat
(module
  (import "P0lib" "write" (func $write (param i32)))
  (import "P0lib" "writeln" (func $writeln))
  (import "P0lib" "read" (func $read (result i32)))
  (tag $e)
 (func $q (param $x i32) (result i32)
    (local $y i32)
    i32.const 1
    call $write
    try
      try
        i32.const 3
        call $write
        throw $e
        i32.const 5
        call $write
      catch $e
        i32.const 7
        call $write
      end
      i32.const 9
      call $write
      throw $e
      i32.const 11
      call $write
    catch $e
      i32.const 13
      call $write
    end
    i32.const 15
  )
  (func $program
    i32.const 1
    call $q
    call $write
  )
```

```
  (memory 1)
  (start $program)
)
```

In [ ]:
```
!wat2wasm --enable-exceptions exception.wat
```

To run this in Chrome, Experimental WebAssembly has to be enabled under `chrome://flags`:

In [ ]:
```
runwasm("exception.wasm")
```

In [ ]:
```
runwasmer("exception.wasm")
```

SIMD Operations

Bulk Memory Operations