Inside WebAssembly

The format, the machine and the host, for people who compile to it

Written for WebAssembly 3.0, checked against Node 22, wabt 1.0.39 and Wasmtime 50, November 2026

Contents

Part I: Getting Started 10

1 Introduction **11**

1.1 Who this book is for 11
1.2 What WebAssembly is, and is not 12
1.3 How the book is organised 13
1.4 The tools 14
1.5 How the examples are checked 14
1.6 Notes in the margin 15
1.7 A note on versions 15

2 Hello, WebAssembly **16**

2.1 The smallest module 16
2.2 A function that returns a number 16
2.3 Turning text into a binary 17
2.4 Running a module from JavaScript 19
2.5 Saying hello 19
2.6 The book's host 21
2.7 What you have seen 21

3 A Stack Machine **23**

3.1 Pushing and popping 23
3.2 Watching the stack 24
3.3 The folded form 25
3.4 Locals 25
3.5 drop and select 26
3.6 The stack is checked before anything runs 27
3.7 The stack does not exist 28
3.8 What you have seen 28

4 Numbers **29**

4.1 Integers have no sign 29
4.2 Wrapping and trapping 30
4.3 Small integers 31
4.4 64-bit integers 32
4.5 Floating point 33
4.6 Conversions 35
4.7 What you have seen 36

5 Functions 37
5.1 Signatures and calls 37
5.2 Recursion 37
5.3 Multiple results 38
5.4 Types and the type section 39
5.5 The function index space 39
5.6 The invisible call stack 40
5.7 Stack overflow 41
5.8 What you have seen 42

Part II: Structure 43

6 Structured Control Flow 44
6.1 block and br 44
6.2 loop 45
6.3 if 46
6.4 Blocks with results 46
6.5 br_table 48
6.6 Compiling loops 49
6.7 Short-circuit operators 50
6.8 How validation handles control flow 50
6.9 What you have seen 52

7 Control Flow Graphs 53
7.1 Why there is no goto 53
7.2 You may not need a graph 53
7.3 Reducible graphs 54
7.4 From a graph to blocks 54
7.5 The Relooper 56
7.6 Irreducible control flow 56
7.7 What you have seen 58

8 Modules and Instances 59
8.1 What a module holds 59
8.2 Imports and exports 60
8.3 Modules and instances 61
8.4 Linking modules together 62
8.5 Instantiation, step by step 63
8.6 The start function 64
8.7 What you have seen 64

Part III: Memory and Indirection 66

9 Linear Memory 67
9.1 Declaring a memory 67
9.2 Loads and stores 67
9.3 Offsets and alignment 68
9.4 Out of bounds 69
9.5 Growing memory 70
9.6 Data segments 71
9.7 Filling and copying 71
9.8 Memory from JavaScript 72
9.9 Multiple memories and 64-bit memories 73
9.10 What you have seen 74

10 Laying Out Memory 75
10.1 A memory map 75
10.2 The shadow stack 76
10.3 Struct layout 78
10.4 Passing and returning aggregates 79
10.5 A bump allocator 79
10.6 Null pointers 81
10.7 Strings, arrays and slices 81
10.8 What you have seen 82

11 Globals and Tables 83
11.1 Globals 83
11.2 Globals shared with the host 84
11.3 Tables and indirect calls 84
11.4 Virtual calls 86
11.5 Closures 87
11.6 Tables as runtime state 88
11.7 Host references 89
11.8 What you have seen 90

Part IV: The Binary Format 91

12 Bytes on the Wire 92
12.1 The preamble 92
12.2 LEB128 92
12.3 Vectors and names 94
12.4 Sections 94

12.5 Types 94
12.6 Imports, functions and exports 95
12.7 A module by hand 95
12.8 Letting a tool count for you 97
12.9 What you have seen 99

13 Encoding Instructions 100
13.1 Function bodies 100
13.2 Opcodes and immediates 100
13.3 Block types 101
13.4 Memory immediates 102
13.5 Prefixed opcodes 103
13.6 An encoder 103
13.7 Things a real encoder adds 107
13.8 What you have seen 108

14 Custom Sections and Debugging 109
14.1 The format 109
14.2 The name section 110
14.3 Names in stack traces 111
14.4 Producers and features 112
14.5 Source maps and DWARF 113
14.6 Debugging in the browser 114
14.7 What you have seen 114

Part V: The Host 115

15 The JavaScript API 116
15.1 Compiling and instantiating 116
15.2 Looking inside a module 117
15.3 Exported functions 118
15.4 Imported functions 118
15.5 Converting values 119
15.6 Memory, Table and Global objects 120
15.7 Errors 120
15.8 A minimal loader 120
15.9 What you have seen 121

16 Across the Boundary 122
16.1 Strings out 122
16.2 Strings in 122

16.3 Bulk data without copying 124
16.4 Batching: the command buffer 124
16.5 Calling back into WebAssembly 126
16.6 Designing the interface 127
16.7 What you have seen 127

17 Traps 128
17.1 Everything that traps 128
17.2 After a trap 129
17.3 Traps and the language you are compiling 130
17.4 What you have seen 132

Part VI: Beyond the First Version 133

18 Versions and Feature Detection 134
18.1 The versions 134
18.2 Support in browsers 135
18.3 Why feature detection matters 136
18.4 Detecting features 136
18.5 Telling tools which features you use 137
18.6 What you have seen 137

19 SIMD 138
19.1 One type, many shapes 138
19.2 The instructions 139
19.3 Vector maths 140
19.4 Structure of arrays 141
19.5 SIMD and the boundary 143
19.6 Relaxed SIMD 143
19.7 What you have seen 144

20 Exceptions 145
20.1 Tags and throw 145
20.2 try_table and catch 145
20.3 exnref and rethrowing 146
20.4 Exceptions and JavaScript 147
20.5 Compiling try and catch 149
20.6 The legacy instructions 150
20.7 What you have seen 150

21 Tail Calls 151
21.1 return_call 151

21.2 The rules 152
21.3 What tail calls make possible 152
21.4 What you have seen 154

22 Garbage Collection 155
22.1 Struct types 155
22.2 Nullable and non-null references 156
22.3 Arrays 156
22.4 The type hierarchy 157
22.5 Subtyping and casts 158
22.6 Recursive types 158
22.7 i31ref 161
22.8 GC objects and JavaScript 161
22.9 Should your language use it? 162
22.10 What you have seen 163

23 Threads and Atomics 165
23.1 Shared memory 165
23.2 Atomic instructions 166
23.3 Threads counting together 167
23.4 Waiting and waking 168
23.5 What is per thread 169
23.6 What you have seen 169

Part VII: Compiling to WebAssembly 171

24 Mapping a Language onto WebAssembly 172
24.1 The decisions 172
24.2 Scalars 173
24.3 Aggregates and unions 173
24.4 Calling convention 174
24.5 Generics 175
24.6 Control flow and the call stack 175
24.7 Runtime 176
24.8 A worked example 176
24.9 What you have seen 178

25 Performance 179
25.1 How engines compile 179
25.2 What the engine does for you 180
25.3 What it does not 180

25.4 Binaryen's optimiser 181
25.5 Code size and start-up 182
25.6 The cost of the boundary 183
25.7 Memory64 and other costs 183
25.8 Practical rules 183
25.9 What you have seen 184

Part VIII: Testing 185

26 The Spec's Test Format 186
26.1 A first script 186
26.2 The commands 187
26.3 Running scripts 189
26.4 How the book's runner works 189
26.5 Using WAST in a compiler's tests 190
26.6 What you have seen 191

27 Testing a Compiler 193
27.1 The layers 193
27.2 Layer 1: validate everything 193
27.3 Layer 2: golden output 195
27.4 Layer 3: execution tests 196
27.5 Layer 4: differential testing 196
27.6 Layer 5: fuzzing 198
27.7 Layer 6: the browser 201
27.8 Performance tests 201
27.9 A case study: this book 202
27.10 What you have seen 202

Appendix A: Instruction Reference 204
A.1 Control 205
A.2 Parametric and variables 206
A.3 Tables 206
A.4 Memory 207
A.5 Constants 208
A.6 i32 operations 209
A.7 i64 operations 210
A.8 f32 operations 211
A.9 f64 operations 212
A.10 Conversions 213
A.11 References 215

A.12 Garbage collection 216
A.13 SIMD (a selection) 218
A.14 Threads (a selection) 219

Appendix B: Tools 220
B.1 wabt 220
B.2 Binaryen 221
B.3 wasm-tools 222
B.4 Wasmtime 223
B.5 Node and V8 224
B.6 This book's scripts 224

Appendix C: Further Reading 225
C.1 The specification 225
C.2 Papers 225
C.3 Compilers to read 226
C.4 Books 226

PART I

Getting Started

The first five chapters set up the machine. You will write modules by hand in the text format, run them in the same engine a browser uses, and learn the handful of ideas everything else is built on: a typed operand stack, four number types whose signedness lives in the instructions, and functions whose call stack you cannot see.

Chapter 1

Introduction

WebAssembly is an instruction set for a machine that does not exist. No chip runs it. Browsers and other hosts take it in, check it, and translate it into the native code of whatever processor they are running on. What makes it unusual is how much of that process the format was designed around. A module can be validated in one pass. It can be compiled while it is still downloading. Running it cannot corrupt the host, and nearly everything it does takes exactly one meaning on every engine.

If you are writing a compiler, that is a remarkable target. You get one output format that runs in every browser at close to native speed, inside a sandbox you do not have to build, with no garbage collector in the way unless you ask for one. In exchange you give up some things a native target lets you do. There is no `goto`. You cannot see the call stack. You cannot take the address of a local variable. Memory is a single array of bytes whose index 0 is a perfectly good address. Most of this book is about what those constraints are, why they are there, and how a compiler works with them.

Who this book is for 1.1

This book is for programmers who want to *generate* WebAssembly: people writing a compiler, a JIT, a code generator for a game's scripting language, or a backend for an existing language. It assumes you can read C, Rust or JavaScript, that you know roughly what a compiler does, and that you have at least a passing familiarity with how a native program uses registers, a stack and a heap.

It does not assume you have written a line of WebAssembly. It starts from an empty module and builds up, one idea at a time, to the parts that matter most to a compiler writer: how control flow must be shaped, how a language's data is laid out in linear memory, how the binary encoding works byte by byte, how a module talks to the JavaScript engine around it, and how to test that what your compiler emits is right.

The running motivation throughout is a compiler whose output runs inside a game engine in the browser. That is where WebAssembly's strengths matter most. A game needs predictable

frame times, so it wants no pauses for garbage collection. It wants tight numeric code, so it wants SIMD. It wants to call into the browser's graphics and audio APIs many times a frame, so it cares about what crossing the boundary costs. None of the book is a tutorial for a particular engine, though, and nothing in it depends on one.

What WebAssembly is, and is not 1.2

It is easy to get the wrong picture of WebAssembly from its name, so here is a short list of what it actually is.

It is a virtual instruction set, not a language. You can write it by hand in the text format, and this book does a great deal of that, but it was designed to be produced by compilers. There are no variables beyond numbered locals, no expressions beyond what fits on a stack, and no types beyond a few numbers and references.

It is a stack machine on paper and a register machine in practice. Instructions take their operands from an implicit stack and push results back onto it. That makes the encoding small. But the stack's shape at every instruction is known before the code runs, so engines turn it into registers when they compile. Nothing is pushed or popped at runtime.

It is structured. Control flow is made of nested blocks, loops and `if`s, with branches that can only leave a block or restart a loop. That is what makes single-pass validation and compilation possible, and it is the biggest single difference from targeting a native ISA.

It is sandboxed. A module can touch only its own memories, tables and globals, and whatever its host chose to give it. Every memory access is checked against the memory's size. Every indirect call is checked against the expected signature. Code is never stored in the memory a program can write, so a program cannot read or rewrite its own code.

It is portable and nearly deterministic. Integer arithmetic wraps the same way everywhere. Floating point is IEEE 754 with round-to-nearest. The few places where engines may differ, such as the bit patterns of NaNs and the results of the "relaxed" SIMD instructions, are listed and narrow.

It is not a JavaScript replacement and not a VM with a runtime. There is no standard library, no I/O, no threads beyond what the host gives you, no exceptions beyond what you declare, and (unless you use the garbage collection feature) no objects. Anything a module does to the outside world goes through a function the host imported into it.

How the book is organised 1.3

The book has eight parts. They are meant to be read in order the first time, since each one leans on the ones before it. Later, the chapters work as a reference.

Part I, Getting Started, writes and runs the first modules and covers the core machine: the operand stack, the four number types, and functions.

Part II, Structure, covers structured control flow, how to compile the control flow of a real language into it (including the control flow WebAssembly does not have), and the module as a whole: its imports, exports and index spaces.

Part III, Memory and Indirection, covers linear memory, how a compiler lays out a language's data inside it, the shadow stack, and the tables and globals that make function pointers and virtual calls possible.

Part IV, The Binary Format, takes the format apart byte by byte. By the end you will have written a module in hex, and a small encoder that emits modules that run.

Part V, The Host, is about the JavaScript side: the `WebAssembly` object, how values cross the boundary, how to pass strings and arrays and callbacks, and what happens when a module traps.

Part VI, Beyond the First Version, covers the features added since WebAssembly 1.0 that matter to a compiler: SIMD, exception handling, tail calls, garbage-collected structs and arrays, and threads.

Part VII, Compiling to WebAssembly, pulls it together into the design questions you face when mapping a language onto WebAssembly, and how engines turn your output into machine code, which tells you what is fast.

Part VIII, Testing, covers the spec's own test format, and builds up a strategy for testing a compiler: validation, execution tests,

golden files, differential testing across engines, and fuzzing, all on top of existing tools.

Three appendices summarise the instructions, the tools, and where to read further.

The tools 1.4

Everything in this book runs on freely available tools. You will meet each of them properly when it first becomes useful.

Tool	Where it comes from	Used for
Node.js 22	nodejs.org	running modules in V8, the engine inside Chrome
wabt	`npm install wabt`	`wat2wasm`, `wasm2wat`, `wasm-objdump`, `wasm-interp`, `wasm-validate`
Binaryen	`npm install binaryen`	`wasm-opt`, the optimiser most toolchains use
Wasmtime	`pip install wasmtime`	a second, independent engine, and the `wat` crate's assembler
wasm-tools	`cargo install wasm-tools`	the Bytecode Alliance's Swiss army knife (optional)

The npm packages are wabt and Binaryen compiled to WebAssembly themselves, so they need nothing but Node. If you prefer native builds, every tool here also has release binaries on GitHub, and the commands are the same.

To follow along, install Node 22 or later, then in an empty directory:

```
npm install wabt binaryen
pip install wasmtime
```

and put `node_modules/.bin` on your `PATH`.

How the examples are checked 1.5

Every example in this book is run when the book is built, and the build fails if any of them stops matching what the book says it does. A module shown with output below it was assembled, validated, run in Node, and produced exactly that output. A module shown with an error below it really was rejected with that message. A command shown with output really printed it.

The modules are run by a small host, about eighty lines of JavaScript, that gives each module a few functions to print with. Chapter 2 shows exactly what it does, so that nothing in the examples is magic, and Chapter 27 explains how the checking itself works. That checking is a miniature version of the test harness Part VIII recommends for a compiler.

Notes in the margin 1.6

Some paragraphs are set off in boxes like this one:

> **From native code**
>
> These boxes compare WebAssembly with what you would do targeting x86 or ARM through LLVM, GCC or your own backend. Skip them if that is not your background.

> **For compiler writers**
>
> These boxes point out the consequences of a feature for code generation: what to emit, what to avoid, and what the engine will do with it.

> **Under the hood**
>
> These boxes explain how engines implement something. You do not need them to write correct code, but they explain why some code is faster than other code.

A note on versions 1.7

WebAssembly is versioned by the specification. Version 1.0 (2017–2019) is often called the MVP, the minimum viable product. Version 2.0 (2022) added features that had shipped in browsers since, such as bulk memory operations, multiple results, reference types and SIMD. Version 3.0 (2025) added garbage collection, typed function references, exception handling, tail calls, 64-bit memories, multiple memories and relaxed SIMD.

Everything in Parts I to V is in WebAssembly 1.0 or 2.0 and runs in every current browser. Part VI covers the 3.0 features, and Chapter 18 says how to detect, at runtime, which ones the engine you are running on supports. Every example in the book runs in Node 22, which uses the same V8 engine as Chrome.

Chapter 2

Hello, WebAssembly

This chapter writes a module, turns it into a binary, and runs it, first with the book's host and then with a dozen lines of your own JavaScript. Along the way it introduces the tools you will use for the rest of the book.

The smallest module 2.1

Here is a complete, valid WebAssembly module in the text format:

```
(module)
```

It does nothing. It has no functions, no memory, no imports and no exports. But it is valid, and it can be instantiated. Turned into a binary, it is eight bytes long:

```
(module)
```

```
wat2wasm empty.wat -o empty.wasm
wc -c empty.wasm
od -An -tx1 empty.wasm
```

```
8 empty.wasm
 00 61 73 6d 01 00 00 00
```

The first four bytes are a magic number, a zero byte followed by the letters `asm`. The next four are the version of the binary format, 1, as a little-endian 32-bit integer. Every module starts with these eight bytes, and everything else is optional. Part IV takes the rest of the format apart.

The text format is made of S-*expressions*: parenthesised lists whose first word says what the list is. `(module ...)` holds everything else, and inside it you will write `(func ...)`, `(memory ...)`, `(import ...)` and so on. Comments start with `;;` and run to the end of the line, or are written `(; like this ;)` and can span lines.

A function that returns a number 2.2

A module becomes useful when it exports something. This one exports a function that returns 42:

```
(module
  (func (export "main") (result i32)
    i32.const 42))
```

```
42
```

Line by line:

- `(func ...)` defines a function. Functions have no names in the binary format. They are numbered, starting from 0, in the order they appear.
- `(export "main")` exports this function under the name `main`, so the host can find it. The export name is a string, and it can be anything.
- `(result i32)` says the function returns one 32-bit integer.
- `i32.const 42` is the whole body: a single instruction that pushes the constant 42 onto the operand stack. When a function's body ends, the values left on the stack are its results.

The book's host called `main` because it was exported, and printed what it returned. That is all the host does with a function that returns something: print the results, separated by spaces.

Turning text into a binary 2.3

Browsers run the binary format, not the text format. The text format is for people: for reading what a compiler produced, for writing tests, and for books. The tool that turns one into the other is `wat2wasm`, from the WebAssembly Binary Toolkit, *wabt* (say "wabbit"):

```
(module
  (func (export "main") (result i32)
    i32.const 42))
```

```
wat2wasm answer.wat -o answer.wasm
wc -c answer.wasm
```

```
37 answer.wasm
```

Thirty-seven bytes. `wasm2wat` goes the other way, which is what you will reach for when you want to read what your compiler wrote:

```
wat2wasm answer.wat -o answer.wasm
wasm2wat answer.wasm
```

```
(module
  (type (;0;) (func (result i32)))
  (func (;0;) (type 0) (result i32)
    i32.const 42)
  (export "main" (func 0)))
```

The round trip added two things you did not write. One is the comments `(;0;)`, which give the index of each definition. The other is a whole `(type ...)` definition. In the binary format, a function does not carry its signature. It refers by index to a signature in the module's *type section*, and many functions can share one. The text format lets you write signatures inline and creates the type entries for you.

The third tool you will use constantly is `wasm-objdump`, which prints the structure of a binary:

```
wasm-objdump -x answer.wasm
```

```
answer.wasm:    file format wasm 0x1

Section Details:

Type[1]:
 - type[0] () -> i32
Function[1]:
 - func[0] sig=0 <main>
Export[1]:
 - func[0] <main> -> "main"
Code[1]:
 - func[0] size=4 <main>
```

A binary module is a sequence of *sections*, each holding one kind of thing. This module has four: the types, the list of which type each function has, the exports, and the code. Separating a function's signature (in the Function section) from its body (in the Code section) looks fussy, but it lets an engine know every function's type before it reads any code, so it can start compiling bodies in parallel, or while the rest of the file is still downloading.

`wasm-objdump -d` disassembles the code, showing each instruction's bytes beside it:

```
wasm-objdump -d answer.wasm
```

```
answer.wasm:    file format wasm 0x1

Code Disassembly:

000021 func[0] <main>:
 000022: 41 2a                      | i32.const 42
 000024: 0b                         | end
```

`i32.const` is the byte `0x41`, and 42 is `0x2a`. The function ends with `0x0b`, the `end` instruction, which the text format writes for you.

Two assemblers

> The book's examples are assembled by the `wat` crate, the text-format parser that Wasmtime and `wasm-tools` use, by way of Wasmtime's Python package. The shell examples use wabt's `wat2wasm`. Both follow the specification, and either is fine for your own work. The `wat` crate tends to support new proposals first.

Running a module from JavaScript 2.4

Every browser and Node has a global `WebAssembly` object. The shortest way to run a module is `WebAssembly.instantiate`, which takes the bytes and an object of imports, and resolves to the module and an instance of it:

```
import fs from 'node:fs';

const bytes = fs.readFileSync('answer.wasm');
const { instance } = await WebAssembly.instantiate(bytes, {});
console.log(instance.exports.main());
```

```
node run.mjs
```

```
42
```

An exported function is an ordinary JavaScript function. Call it, and the engine runs the WebAssembly code and converts the result to a JavaScript number.

In a browser, you would fetch the file instead of reading it, and use the streaming form, which compiles while the bytes arrive:

```
const { instance } = await WebAssembly.instantiateStreaming(
  fetch('answer.wasm'), {});
console.log(instance.exports.main());
```

The server must send the file with the MIME type `application/wasm` for the streaming form to accept it. Chapter 15 covers the whole JavaScript API.

Saying hello 2.5

A module cannot print. It has no I/O of any kind. It can only call functions that its host gives it, so printing means importing a function from the host. The book's host offers a few, in a module it calls `env`:

```
(module
  (import "env" "print_i32" (func $print_i32 (param i32)))
  (func (export "main")
    i32.const 7
    call $print_i32
    i32.const -7
    call $print_i32))
```

```
7
-7
```

An import names a *module* and a *field*, here `env` and `print_i32`, and says what kind of thing it expects: a function taking one `i32`. The names are just strings that the host uses to look up what to provide. `env` is a common convention but nothing more.

`$print_i32` is a *symbolic name*, written with a `$`. It exists only in the text format, as a convenience. The assembler replaces each one with an index. Imported functions get the first indices, so `$print_i32` is function 0 and `main` is function 1.

`main` returns nothing this time, so the host prints nothing after it.

To print text, the module needs somewhere to keep the characters: a *linear memory*, which is an array of bytes the module can read and write. A *data segment* puts the bytes of a string into the memory when the module is instantiated:

```
(module
  (import "env" "print" (func $print (param i32 i32)))
  (memory (export "memory") 1)
  (data (i32.const 16) "Hello, WebAssembly!")
  (func (export "main")
    i32.const 16       ;; where the string starts
    i32.const 19       ;; how many bytes it is
    call $print))
```

```
Hello, WebAssembly!
```

`(memory 1)` declares a memory one *page* long. A page is 64 KiB. The data segment copies 19 bytes to address 16. The `print` import takes an address and a length, and the host reads the bytes out of the memory the module exported as `memory`, decodes them as UTF-8, and prints them.

This is how all text crosses between WebAssembly and JavaScript: as bytes in memory, with an address and a length. WebAssembly has no string type. Chapter 16 covers strings in depth.

The book's host 2.6

Every example marked with output in this book is run by the same small program, `scripts/run.mjs`, in Node. Here is everything it does:

1. Compile the module, and look at what it imports.
2. Provide each import it recognises, all from module `env`:

Import	Signature	Does
`print_i32`	`(param i32)`	prints a signed 32-bit integer and a newline
`print_u32`	`(param i32)`	the same bits, printed as unsigned
`print_i64`	`(param i64)`	prints a signed 64-bit integer
`print_f32`	`(param f32)`	prints a 32-bit float
`print_f64`	`(param f64)`	prints a 64-bit float
`print`	`(param i32 i32)`	prints UTF-8 text from memory, given its address and length, and a newline
`write`	`(param i32 i32)`	the same, without the newline
`memory`	`(memory 1)`	a memory, for modules that import one rather than defining their own

3. Instantiate the module, then call the exports the example names, or `main` if it names none, and print whatever each one returns.
4. If anything traps, print the error on stderr and stop.

Some examples call an export with arguments. When they do, the book says which, like this:

```
(module
  (func (export "add") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add))
```

The host called `add(2, 3)` and then `add(-1, 1)`:

called as `add(2, 3)` and `add(-1, 1)`

```
5
0
```

What you have seen 2.7

A module is a list of definitions: functions, memories, data, imports and exports. The text format writes them as S-

expressions; the binary format, which is what engines run, encodes them as sections of bytes. Functions compute by pushing and popping values on an operand stack, and anything that touches the outside world is an import.

The next chapter looks properly at that operand stack.

Chapter 3

A Stack Machine

WebAssembly code computes with an *operand stack*. Instructions pop their operands off it and push their results onto it. This chapter shows how that works, how the text format lets you write it in two different shapes, how local variables fit in, and why the stack, despite appearances, is not something the machine maintains at runtime.

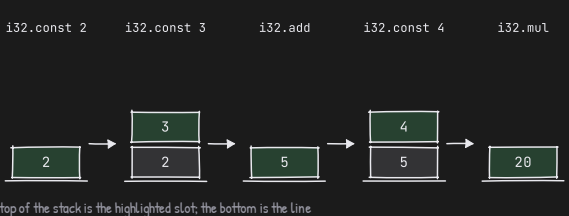
Pushing and popping 3.1

Here is a function that computes (2 + 3) × 4:

```
(module
  (func (export "main") (result i32)
    i32.const 2
    i32.const 3
    i32.add
    i32.const 4
    i32.mul))
```

```
20
```

Each instruction has a fixed effect on the stack. `i32.const 2` pushes 2. `i32.add` pops two `i32`s and pushes their sum. `i32.mul` pops two and pushes their product. When the body ends, the one value left is the function's result.



The operand stack after each instruction of (2 + 3) × 4.

The specification writes each instruction's effect as a type, with what it pops on the left and what it pushes on the right:

Instruction	Type
`i32.const 2`	`[] → [i32]`
`i32.add`	`[i32 i32] → [i32]`
`i32.eqz`	`[i32] → [i32]`
`drop`	`[t] → []`
`select`	`[t t i32] → [t]`
`nop`	`[] → []`

The order matters for instructions that are not commutative. The *first* operand is the one pushed first, so it is deeper in the stack:

```
(module
  (func (export "main") (result i32)
    i32.const 10
    i32.const 3
    i32.sub))     ;; 10 - 3, not 3 - 10
```

```
7
```

Watching the stack 3.2

wabt's `wasm-interp` is a simple interpreter, and with `--trace` it shows the stack as it runs. It is a good way to check your understanding of a piece of code:

```
(module
  (func (export "main") (result i32)
    i32.const 10
    i32.const 3
    i32.sub))
```

```
wasm-interp sub.wasm --run-all-exports --trace
```

```
>>> running export "main":
#0.    0: V:0  | i32.const 10
#0.    8: V:1  | i32.const 3
#0.   16: V:2  | i32.sub 10, 3
#0.   20: V:1  | return
main() => i32:7
```

Each line shows where the instruction is in the interpreter's own translation of the code (not the file), the stack depth before it runs (`V:2` means two values), and for instructions that pop, the values they popped. The `return` at the end is the interpreter's rendering of the function's final `end`.

The folded form 3.3

Writing one instruction per line gets hard to read once expressions nest. The text format has a second shape for the same instructions, the *folded* form, where an instruction is written in parentheses with its operands inside it:

```
(module
  (func (export "main") (result i32)
    (i32.mul
      (i32.add (i32.const 2) (i32.const 3))
      (i32.const 4))))
```

```
20
```

This is pure notation. The assembler unfolds it, operands first, into exactly the same five instructions as before. You can mix the two forms freely, and this book does: folded where it makes the structure clearer, flat where the order of operations is the point.

```
(module
  (func (export "main") (result i32)
    (i32.add (i32.const 2) (i32.const 3))   ;; leaves 5 on the
        stack
    i32.const 4
    i32.mul))
```

```
20
```

`wasm2wat` always prints the flat form, since that is what is actually in the binary. It can print the folded form if you ask, with `--fold-exprs`.

Locals 3.4

A function's parameters and its local variables live in numbered slots called *locals*. `local.get` pushes a local's value. `local.set` pops a value into a local. `local.tee` stores into a local but leaves the value on the stack, like an assignment expression in C.

Parameters are the first locals. Any further locals are declared at the start of the function with `(local ...)`, and they all start at zero:

```
(module
  (func (export "hypot2") (param $x i32) (param $y i32) (result
      i32)
    (local $xx i32)
    (local.set $xx (i32.mul (local.get $x) (local.get $x)))
    (i32.add
      (local.get $xx)
      (i32.mul (local.get $y) (local.get $y)))))
```

called as `hypot2(3, 4)`

```
25
```

`$x` is local 0, `$y` is local 1, and `$xx` is local 2. As with functions, the names are a text-format convenience. Here is the same function with indices, and `local.tee` saving a `local.set` and a `local.get`:

```
(module
  (func (export "hypot2") (param i32 i32) (result i32) (local i32)
    local.get 0
    local.get 0
    i32.mul
    local.tee 2          ;; store x*x in local 2, and keep it on
        the stack
    local.get 1
    local.get 1
    i32.mul
    i32.add))
```

called as `hypot2(3, 4)`

```
25
```

Locals are typed. Each holds one value of its declared type, and every `local.get` and `local.set` is checked against that type before the code ever runs.

> **For compiler writers**
>
> Locals are the closest thing WebAssembly has to registers, and engines treat them that way: a register allocator decides which locals live in machine registers. You can use as many as you like (the limit is 50,000 per function), so the usual approach is to give every temporary and every variable whose address is never taken a local of its own, and let the engine's allocator sort it out. What you cannot do is take a local's address. Chapter 10 covers what to do about variables that need one.

drop and select 3.5

`drop` throws away the top of the stack. You need it whenever an instruction produces a value you do not want, since a function's stack must hold exactly its results when it ends:

```
(module
  (func (export "main") (result i32)
    i32.const 1
    i32.const 2
    drop))
```

```
1
```

`select` is a conditional move. It pops a condition and two values, and pushes the first value if the condition is non-zero and the

second otherwise. Both values are always computed, so it is for cheap operands, like a C ternary whose arms have no side effects:

```
(module
  (func (export "max") (param $a i32) (param $b i32) (result i32)
    (select
      (local.get $a)
      (local.get $b)
      (i32.gt_s (local.get $a) (local.get $b)))))
```

called as `max(3, 9)`, `max(9, 3)` and `max(-5, -6)`

```
9
9
-5
```

`i32.gt_s` compares two `i32`s as signed numbers and pushes 1 if the first is greater, 0 otherwise. There is no boolean type. Comparisons produce an `i32` that is 0 or 1, and conditions accept any `i32`, treating zero as false and everything else as true.

The stack is checked before anything runs 3.6

Every function's code is *validated* when the module is compiled. The validator walks the instructions once, tracking the types on the stack, and rejects the module if any instruction would pop a value that is not there, or pop the wrong type, or if the stack does not hold exactly the function's results at the end.

Leaving an extra value behind is an error:

```
(module
  (func (export "main") (result i32)
    i32.const 1
    i32.const 2))
```

```
CompileError: WebAssembly.Module(): Compiling function #0 failed:
    expected 1 elements on the stack for fallthru, found 2 @+38
```

So is popping a value of the wrong type:

```
(module
  (func (export "main") (result i32)
    i32.const 1
    i64.const 2
    i32.add))
```

```
i32.add[1] expected type i32, found i64.const of type i64
```

These are V8's messages, which is what you would see in Chrome's console. `@+38` is the byte offset in the binary where V8 found the problem; the *fallthru* is the end of the function, where control falls out of the body. Each engine words its messages

differently. Wasmtime, for one, would say `type mismatch: expected i32, found i64`. Chapter 6 shows how validation handles control flow, and Chapter 27 shows how to get every engine's opinion on a module your compiler produced.

Because of validation, there is never any doubt about the stack at a given instruction. Its depth and the type of every slot are fixed, the same every time that instruction runs, no matter what path led there.

The stack does not exist 3.7

That last fact is what lets engines throw the stack away.

Under the hood

When V8 or Wasmtime compiles a function, it does not allocate a stack and push values onto it. Since every stack slot's position and type is known at every instruction, each slot can be treated as a virtual register. The expression `(i32.add (local.get 0) (i32.const 1))` becomes a single machine `add` of a register and an immediate. The stack machine is an *encoding*: it lets an instruction name its operands without spending bytes on register numbers, which makes modules small. It is not an execution model.

This has a practical consequence for compilers. There is no cost to the stack itself, so you do not need to minimise stack traffic, and there is no benefit to clever stack scheduling. Emit whatever shape of code is simplest for your compiler. The engine's register allocator will undo it.

What *can* matter is the number of locals and how long each one is live, since those are what the register allocator works with. That is the same problem a native backend has, and it is the engine's to solve, which is one of the main reasons WebAssembly is a good compilation target: instruction selection and register allocation, the hardest parts of a native backend, are done for you.

What you have seen 3.8

Instructions pop operands from a typed stack and push results. The folded form is a notation for writing nested expressions, and unfolds to the same instructions. Locals are numbered, typed slots, parameters first. Validation checks every stack effect before the code runs, which is what lets engines turn the stack into registers.

The next chapter looks at the values themselves.

Chapter 4

Numbers

WebAssembly has four number types: `i32`, `i64`, `f32` and `f64`. (A fifth, `v128`, holds SIMD vectors and has a chapter of its own.) That is a short list for a language that wants to compile C, Rust, or your language, with its signed and unsigned integers of four sizes, its booleans, characters and enums. This chapter shows how the short list covers the long one, and the exact rules for overflow, division, conversions and floating point, which are the kind of detail a compiler has to get exactly right.

Integers have no sign 4.1

An `i32` is 32 bits. It is not signed and it is not unsigned. The *instructions* decide how to interpret the bits, and where the interpretation matters there are two versions of the instruction, with `_s` or `_u` on the end:

```
(module
  (func (export "div") (param i32 i32) (result i32)
    (i32.div_s (local.get 0) (local.get 1)))
  (func (export "divu") (param i32 i32) (result i32)
    (i32.div_u (local.get 0) (local.get 1))))
```

called as `div(-7, 2)` and `divu(-7, 2)`

```
-3
2147483644
```

The same bits, `0xfffffff9`, are −7 to `div_s` and 4,294,967,289 to `div_u`. The host prints results as signed, which is why the unsigned quotient, 2,147,483,644, happens to look sensible.

Where the interpretation does not matter there is only one instruction. Addition, subtraction and multiplication give the same bits whether you think of the operands as signed or unsigned (that is the beauty of two's complement), so there is one `i32.add`, one `i32.sub` and one `i32.mul`. Equality is the same, so there is one `i32.eq`.

Here is every `i32` instruction, grouped by what it does. `i64` has exactly the same set.

Kind	Instructions
arithmetic	`add` `sub` `mul` `div_s` `div_u` `rem_s` `rem_u`
bitwise	`and` `or` `xor` `shl` `shr_s` `shr_u` `rotl` `rotr`
counting	`clz` `ctz` `popcnt`
tests	`eqz`
comparison	`eq` `ne` `lt_s` `lt_u` `gt_s` `gt_u` `le_s` `le_u` `ge_s` `ge_u`
sign extension	`extend8_s` `extend16_s` (and `i64.extend32_s`)

`shr_s` is the arithmetic right shift, which copies the sign bit in; `shr_u` is the logical one, which shifts in zeros. `clz` and `ctz` count leading and trailing zero bits, and `popcnt` counts the bits that are set. Every one of these exists on modern hardware as a single instruction, and engines use it.

> **For compiler writers**
>
> Your language's `u8`, `i16`, `u32` and so on all become `i32` (or `i64` for the 64-bit ones). The type tells you which instructions to pick: `div_u` for unsigned division, `lt_s` for a signed comparison. For the types narrower than 32 bits, you must also decide when to make sure the upper bits are right. Section 4.3 covers that.

Wrapping and trapping 4.2

Integer arithmetic wraps around. There is no overflow flag and no undefined behaviour:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "main")
    (call $print (i32.add (i32.const 0x7fffffff) (i32.const 1)))
    (call $print (i32.mul (i32.const 65536) (i32.const 65536)))
    (call $print (i32.sub (i32.const 0) (i32.const 1)))))
```

```
-2147483648
0
-1
```

Shift counts are taken modulo the width, so shifting an `i32` by 33 is shifting it by 1. That matches x86, but it does not match C, where it is undefined, or Rust, where it panics in debug builds. If your language has a rule for oversized shifts, you have to implement it yourself:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "main")
    (call $print (i32.shl (i32.const 1) (i32.const 33)))
    (call $print (i32.shr_s (i32.const -64) (i32.const 2)))
    (call $print (i32.shr_u (i32.const -64) (i32.const 28)))))
```

```
2
-16
15
```

Only two integer operations can fail, and both *trap*: dividing by zero, and dividing the most negative number by −1, whose result does not fit:

```
(module
  (func (export "main") (result i32)
    (i32.div_s (i32.const 1) (i32.const 0))))
```

```
RuntimeError: divide by zero
```

```
(module
  (func (export "main") (result i32)
    (i32.div_s (i32.const -2147483648) (i32.const -1))))
```

```
RuntimeError: divide result unrepresentable
```

A trap stops the WebAssembly code immediately and throws a `WebAssembly.RuntimeError` in the JavaScript that called it. There is no way to catch a trap from inside WebAssembly. Chapter 17 covers traps in detail.

The remainder, `rem_s`, does *not* trap on the most negative number divided by −1: the remainder is 0, which fits. It still traps on zero. Its sign follows the dividend, as in C, Rust and Go:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "main")
    (call $print (i32.rem_s (i32.const -7) (i32.const 2)))
    (call $print (i32.rem_s (i32.const 7) (i32.const -2)))
    (call $print (i32.rem_s (i32.const -2147483648) (i32.const
        -1)))))
```

```
-1
1
0
```

If your language's `%` is a modulus whose sign follows the divisor, as in Python, you need a fix-up after `rem_s`.

Small integers 4.3

Since there are only 32- and 64-bit integers, a compiler has to represent an 8- or 16-bit value in an `i32`, and decide what the upper bits hold. There are two common approaches.

The first keeps every small value *normalised*: the upper bits always hold the sign extension (for signed types) or zeros (for unsigned types). After every operation that could disturb them, the compiler normalises again. For unsigned types that is an `and` with a mask. For signed types it is `extend8_s` or `extend16_s`, which take the low 8 or 16 bits and copy their top bit upwards:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "main")
    ;; a u8 holding 200, plus 100: 300 must wrap to 44
    (call $print (i32.and (i32.add (i32.const 200) (i32.const
        100)) (i32.const 0xff)))
    ;; an i8 holding 100, plus 100: 200 must wrap to -56
    (call $print (i32.extend8_s (i32.add (i32.const 100)
        (i32.const 100))))))
```

```
44
-56
```

The second approach lets the upper bits hold garbage, and normalises only where it matters: before a comparison, a division, a right shift, a conversion, or a store that will be read back as a wider type. Addition, subtraction, multiplication, and the bitwise operations never let garbage in the upper bits affect the lower ones, so they need no fix-up.

The second approach emits less code, and is what LLVM does. The first is simpler to get right. Either is fine, as long as you pick one and test the edges.

> **Loads do it for free**
>
> Values that live in memory need no explicit fix-up: the load instructions come in widening versions, such as `i32.load8_s` and `i32.load16_u`, that sign- or zero-extend as they load. Chapter 9 covers them.

Booleans become `i32`s holding 0 or 1, which is exactly what the comparison instructions produce. `i32.eqz` is logical *not*. A `char` is an `i32` holding a code point. Enums are `i32`s, or `i64`s if their values need it.

64-bit integers 4.4

`i64` has the same instructions as `i32`. Mixing the two needs explicit conversions:

Instruction	Does
`i32.wrap_i64`	keeps the low 32 bits
`i64.extend_i32_s`	sign-extends to 64 bits
`i64.extend_i32_u`	zero-extends to 64 bits

```
(module
  (import "env" "print_i64" (func $print (param i64)))
  (func (export "main")
    (call $print (i64.extend_i32_s (i32.const -1)))
    (call $print (i64.extend_i32_u (i32.const -1)))
    (call $print (i64.mul (i64.const 4294967296) (i64.const
        4294967296)))))
```

```
-1
4294967295
0
```

In JavaScript, an `i64` crosses the boundary as a `BigInt`, since a JavaScript number cannot hold every 64-bit integer exactly. That is why calling an `i64` function from JavaScript needs arguments like `5n` rather than `5`:

```
(module
  (func (export "square") (param i64) (result i64)
    (i64.mul (local.get 0) (local.get 0))))
```

called as `square(3000000000n)` and `square(4000000000n)`

```
9000000000000000000
-2446744073709551616
```

The first square fits in 63 bits. The second, 1.6×10^{19}, does not, so it wrapped to $1.6 \times 10^{19} - 2^{64}$. When you write tests for wrapping arithmetic, compute the expected values with a language that has arbitrary-precision integers rather than by hand. In Python:

```
x = 4_000_000_000 ** 2
print((x + 2**63) % 2**64 - 2**63)
```

```
-2446744073709551616
```

Floating point 4.5

`f32` and `f64` are IEEE 754 single and double precision, with round to nearest, ties to even. The instructions are the ones you would expect:

Kind	Instructions
arithmetic	`add` `sub` `mul` `div` `sqrt`
rounding	`ceil` `floor` `trunc` `nearest`
sign	`abs` `neg` `copysign`
comparison	`eq` `ne` `lt` `gt` `le` `ge`
other	`min` `max`

`nearest` rounds to the nearest integer, ties to even, unlike C's `round`, which rounds ties away from zero. Comparisons follow IEEE: every comparison with a NaN is false except `ne`. `min` and `max` return NaN if either operand is NaN, and treat −0 as less than +0. Those are not the rules of C's `fmin` or of x86's `minss`, which is why `f32.min` usually compiles to several machine instructions rather than one.

```
(module
  (import "env" "print_f64" (func $print (param f64)))
  (func (export "main")
    (call $print (f64.div (f64.const 1) (f64.const 3)))
    (call $print (f64.nearest (f64.const 2.5)))
    (call $print (f64.nearest (f64.const 3.5)))
    (call $print (f64.min (f64.const -0) (f64.const 0)))
    (call $print (f64.div (f64.const 1) (f64.const 0)))
    (call $print (f64.sqrt (f64.const -1)))))
```

```
0.3333333333333333
2
4
-0
Infinity
NaN
```

Floating-point division by zero does not trap. It gives an infinity or a NaN, as IEEE says.

`f32` values are rounded to single precision after every operation, so they behave exactly like C's `float` with strict semantics. When an `f32` crosses into JavaScript it becomes a double holding the same value, which is why it prints with so many digits:

```
(module
  (import "env" "print_f32" (func $print (param f32)))
  (func (export "main")
    (call $print (f32.const 0.1))
    (call $print (f32.add (f32.const 16777216) (f32.const 1)))))
```

```
0.10000000149011612
16777216
```

The literal syntax for floats is rich, since a compiler writing tests needs to be able to write any value exactly: decimal (`1.5`, `1e-3`), hexadecimal (`0x1.8p1`, which is 3), `inf`, `-inf`, `nan`, and NaNs with a chosen payload (`nan:0x200000`).

Determinism, and its one exception

Floating-point arithmetic in WebAssembly is deterministic, with one exception: the bit pattern of a NaN result. When an operation produces a NaN, the specification says which NaNs it may be, but not which one, and x86 and ARM produce different ones. If your language lets a program look at the bits of a NaN, those bits may differ between machines. Nothing else does: there is no fused multiply-add unless you ask for it, no extended precision, no flush-to-zero of denormals.

Conversions 4.6

Converting between integers and floats has a large family of instructions, with names that say both types:

Instruction	From → to	Notes
`f64.convert_i32_s`	`i32` → `f64`	also `_u`, and every int/float pair
`i32.trunc_f64_s`	`f64` → `i32`	traps on NaN or out of range
`i32.trunc_sat_f64_s`	`f64` → `i32`	saturates instead: NaN gives 0
`f32.demote_f64`	`f64` → `f32`	rounds to nearest
`f64.promote_f32`	`f32` → `f64`	exact
`i32.reinterpret_f32`	`f32` → `i32`	the same bits, a different type
`f32.reinterpret_i32`	`i32` → `f32`	

The two kinds of float-to-integer conversion need a closer look. The plain `trunc` instructions trap if the value is NaN or does not fit:

```
(module
  (func (export "main") (result i32)
    (i32.trunc_f64_s (f64.const 3e9))))
```

```
RuntimeError: float unrepresentable in integer range
```

The `trunc_sat` instructions, added in WebAssembly 2.0, never trap. Values too large become the largest integer, too small the smallest, and NaN becomes 0:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "main")
    (call $print (i32.trunc_sat_f64_s (f64.const 3e9)))
    (call $print (i32.trunc_sat_f64_s (f64.const -3e9)))
    (call $print (i32.trunc_sat_f64_s (f64.const nan)))
    (call $print (i32.trunc_sat_f64_s (f64.const -2.9)))))
```

```
2147483647
-2147483648
0
-2
```

Pick the one that matches your language. Rust's `as` saturates, so a Rust compiler emits `trunc_sat`. C leaves out-of-range conversions undefined, so either will do, and `trunc_sat` avoids a trap. Java's conversions also saturate. If your language promises an error on overflow, use the trapping form, or test the range yourself first and raise your own error.

The `reinterpret` instructions are free: they change only the type, not the bits. They are how you implement bit tricks on floats, or a union of a float and an integer, without going through memory:

```
(module
  (import "env" "print_u32" (func $print (param i32)))
  (func (export "main")
    (call $print (i32.reinterpret_f32 (f32.const 1.0)))
    (call $print (i32.reinterpret_f32 (f32.const -0.0)))))
```

```
1065353216
2147483648
```

That is `0x3f800000` and `0x80000000`.

What you have seen 4.7

Four number types cover every scalar in a source language. Integers are bits, and the instruction says whether they are signed. Arithmetic wraps, shifts take their count modulo the width, and only division can trap. Floats are IEEE, deterministic except for NaN bits. Conversions are explicit, and float-to-int comes in a trapping and a saturating form.

The next chapter puts instructions into functions and calls them.

Chapter 5

Functions

Functions are where all WebAssembly code lives. There is no code outside them: no top-level statements, no initialisers that run arbitrary code (apart from one optional *start* function), no code in data. This chapter covers defining and calling functions, multiple results, and the call stack, which is the first place WebAssembly differs sharply from a native target.

Signatures and calls 5.1

A function has a *type*: a list of parameter types and a list of result types. `call` takes a function index, pops that function's parameters off the stack, runs it, and pushes its results:

```
(module
  (func $square (param i32) (result i32)
    (i32.mul (local.get 0) (local.get 0)))
  (func (export "cube") (param $x i32) (result i32)
    (i32.mul (call $square (local.get $x)) (local.get $x))))
```

called as `cube(3)` and `cube(-4)`

```
27
-64
```

Arguments are pushed in order, first argument first, exactly like the operands of an instruction. A call is, as far as the stack is concerned, just an instruction whose type is the callee's signature.

The `return` instruction leaves a function early, taking the function's results from the top of the stack:

```
(module
  (func (export "abs") (param $x i32) (result i32)
    (if (i32.ge_s (local.get $x) (i32.const 0))
      (then (return (local.get $x))))
    (i32.sub (i32.const 0) (local.get $x))))
```

called as `abs(-5)` and `abs(5)`

```
5
5
```

(`if` is the subject of the next chapter.)

Recursion 5.2

Functions can call themselves, and each other, freely:

```
(module
  (func $fac (export "fac") (param $n i64) (result i64)
    (if (result i64) (i64.le_u (local.get $n) (i64.const 1))
      (then (i64.const 1))
      (else
        (i64.mul
          (local.get $n)
          (call $fac (i64.sub (local.get $n) (i64.const 1))))))))
```

called as `fac(10n)` and `fac(20n)`

```
3628800
2432902008176640000
```

The function takes an `i64`, so the host passed the arguments as BigInts, `10n` and `20n`. Passing a plain JavaScript number where an `i64` is expected is an error, raised before any WebAssembly code runs:

```
(module
  (func $fac (export "fac") (param $n i64) (result i64)
    (if (result i64) (i64.le_u (local.get $n) (i64.const 1))
      (then (i64.const 1))
      (else
        (i64.mul
          (local.get $n)
          (call $fac (i64.sub (local.get $n) (i64.const 1))))))))
```

called as `fac(10)`

```
TypeError: Cannot convert 10 to a BigInt
```

Multiple results 5.3

A function can return more than one value. This was added in WebAssembly 2.0, and every engine supports it:

```
(module
  (func (export "divmod") (param $a i32) (param $b i32) (result
     i32 i32)
    (i32.div_s (local.get $a) (local.get $b))
    (i32.rem_s (local.get $a) (local.get $b))))
```

called as `divmod(17, 5)` and `divmod(-17, 5)`

```
3 2
-3 -2
```

The results are left on the stack in order, the first result deepest. A caller in WebAssembly gets them all pushed onto its stack. A caller in JavaScript gets an array, which the book's host prints separated by spaces.

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func $divmod (param $a i32) (param $b i32) (result i32 i32)
    (i32.div_s (local.get $a) (local.get $b))
    (i32.rem_s (local.get $a) (local.get $b)))
  (func (export "main")
    (local $q i32)
    (local $r i32)
    (call $divmod (i32.const 100) (i32.const 7))
    local.set $r          ;; the last result is on top
    local.set $q
    (call $print (local.get $q))
    (call $print (local.get $r))))
```

```
14
2
```

For compiler writers

Multiple results are how to return small structs, tuples and `Result`-style values without going through memory. A function returning a pair of floats can return `(result f32 f32)`, and the engine will return them in registers. Large aggregates are still better returned through memory, through a pointer the caller passes in, which is what C compilers targeting WebAssembly do for any struct, by convention. Chapter 24 covers calling conventions.

Types and the type section 5.4

Every function's signature lives in the module's *type section*, and the function refers to it by index. The text format has let you write signatures inline so far, but you can also declare a type by name and refer to it:

```
(module
  (type $binop (func (param i32 i32) (result i32)))
  (func (export "add") (type $binop)
    (i32.add (local.get 0) (local.get 1)))
  (func (export "sub") (type $binop)
    (i32.sub (local.get 0) (local.get 1))))
```

called as `add(1, 2)` and `sub(1, 2)`

```
3
-1
```

When you write a signature inline, the assembler looks for an existing type with that signature and uses it, or adds one if there is none. Named types matter more once function references and indirect calls are involved (Chapter 11), because there you have to name the type of the function you expect to call.

The function index space 5.5

Functions are numbered in one *index space*, and imported functions come first. A module with two imports and three

definitions has functions 0 and 1 imported, and 2, 3 and 4 defined. The same is true of every other kind of thing a module can import: memories, tables, globals and tags each have an index space, with imports first.

```
(module
  (import "env" "print_i32" (func $print_i32 (param i32)))
  (import "env" "print_f64" (func $print_f64 (param f64)))
  (func $a)
  (func $b)
  (func $c (export "main")
    (call $print_i32 (i32.const 1))))
```

```
wasm-objdump -x -j Import indices.wasm | tail -n +6
wasm-objdump -x -j Function indices.wasm | tail -n +6
```

```
Import[2]:
 - func[0] sig=0 <print_i32> <- env.print_i32
 - func[1] sig=1 <print_f64> <- env.print_f64
Function[3]:
 - func[2] sig=2 <a>
 - func[3] sig=2 <b>
 - func[4] sig=2 <c>
```

The names in angle brackets come from the module's *name section*, an optional part of the binary where the assembler recorded the `$names` from the text, for tools and debuggers to show. Chapter 14 covers it.

This matters to a compiler in one practical way: if you decide to add an import late in compilation, every function index you have already emitted shifts. Most compilers collect their imports first, or number functions symbolically and resolve the numbers when they write the binary.

The invisible call stack 5.6

Here is the most important thing in this chapter. In WebAssembly, *you cannot see the call stack.*

There is no stack pointer register. There is no instruction to read a return address, or walk up the frames, or take the address of a local variable. Locals are not in memory at all, as far as WebAssembly code can tell; they are just numbered slots that `local.get` and `local.set` reach. The engine keeps return addresses and spilled registers on a stack of its own, in memory the module cannot read or write.

This is part of WebAssembly's safety. A buffer overflow in linear memory cannot overwrite a return address, because return

addresses are not in linear memory. A module cannot jump to an address it computed, because it has no way to name an address in code.

It also means a lot of things a native compiler takes for granted need another approach:

- **Address-taken locals.** If your language lets you take a pointer to a local variable (`&x` in C or Rust, or passing `x` by reference), that variable has to live in linear memory, not in a WebAssembly local. Most compilers keep a second stack in linear memory for exactly this, called the *shadow stack*. Chapter 10 builds one.
- **Garbage collectors** that scan the stack for roots cannot do so. They have to keep roots in a shadow stack too, or use WebAssembly's own garbage collection (Chapter 22).
- **Exceptions, coroutines and `setjmp`** cannot be built by saving and restoring stack pointers. Exceptions have their own instructions (Chapter 20); coroutines need either a transformation of your code or the stack-switching proposal still being standardised.
- **Stack traces** come from the engine, not from walking frames yourself.

Stack overflow 5.7

The engine's stack is finite, and a call that would overflow it is reported as a JavaScript `RangeError`, the same error as runaway recursion in JavaScript:

```
(module
  (func $forever (export "main")
    (call $forever)))
```

```
RangeError: Maximum call stack size exceeded
```

How deep you can recurse depends on the engine, its settings, and the size of each frame, which depends on how many locals the engine had to spill. In V8 with default settings it is typically some tens of thousands of frames. A compiler for a language with deep recursion, or a language that promises unbounded recursion, cannot rely on it. Chapter 21 shows how tail calls help with one important case.

What you have seen 5.8

Functions have types made of parameter and result lists, which live in a type section shared by the whole module. `call` pops arguments and pushes results, and a function can return several values. Functions are numbered in an index space with imports first. The call stack belongs to the engine: you cannot read it, take addresses in it, or rely on its depth.

That ends Part I. Part II is about control flow inside functions, and about modules as a whole.

PART II

Structure

Native code jumps wherever it likes. WebAssembly code cannot: its control flow is nested blocks and loops, and a branch can only leave a block or restart a loop. The first two chapters of this part show how that works and how a compiler fits a language's control flow into it, including the control flow that does not fit naturally. The third steps back to look at a module as a whole: what it can contain, how its parts are numbered, and how it connects to the outside world.

Chapter 6

Structured Control Flow

WebAssembly has no jump instruction. It has three ways to group instructions, `block`, `loop` and `if`, and branches that can only target one of the groups that encloses them. This chapter covers each piece, then shows how the loops, `break`s, `continue`s and `switch`es of an ordinary language compile into them.

block and br 6.1

A `block` groups instructions, and gives a branch somewhere to go. A branch to a block jumps to its *end*:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "main")
    (block $done
      (call $print (i32.const 1))
      (br $done)
      (call $print (i32.const 2)))   ;; skipped
    (call $print (i32.const 3))))
```

```
1
3
```

`br $done` is an unconditional branch to the end of the block labelled `$done`. `br_if` branches only if the `i32` it pops is non-zero:

```
(module
  (func (export "sign") (param $x i32) (result i32)
    (block $not_positive
      (br_if $not_positive (i32.le_s (local.get $x) (i32.const
          0)))
      (return (i32.const 1)))
    (block $not_negative
      (br_if $not_negative (i32.ge_s (local.get $x) (i32.const
          0)))
      (return (i32.const -1)))
    (i32.const 0)))
```

called as `sign(5)`, `sign(0)` and `sign(-5)`

```
1
0
-1
```

Labels such as `$done` exist only in the text format. In the binary, a branch names its target by *depth*: 0 is the innermost enclosing block, 1 the one around that, and so on. `wasm2wat` shows what the assembler made of the names:

```
(module
  (func (export "main")
    (block $outer
      (block $inner
        (br $outer)
        (br $inner)))))
```

```
wasm2wat depths.wasm
```

```
(module
  (type (;0;) (func))
  (func (;0;) (type 0)
    block  ;; label = @1
      block  ;; label = @2
        br 1 (;@1;)
        br 0 (;@2;)
      end
    end)
  (export "main" (func 0)))
```

A compiler emitting the binary format directly has to compute these depths, which is easy: keep a stack of the blocks you are inside, and a branch's depth is how far down that stack its target is.

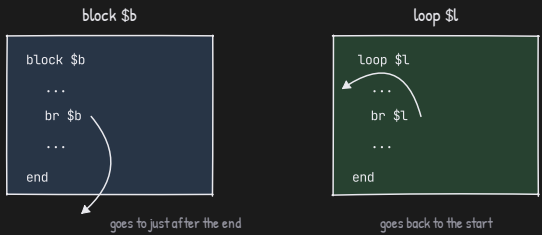
loop 6.2

A `loop` also groups instructions, but a branch to a loop jumps to its *start*. A loop does not repeat by itself: if control reaches the end of a loop, it carries on after it. The repetition comes from branching back:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "main")
    (local $i i32)
    (loop $again
      (call $print (local.get $i))
      (local.set $i (i32.add (local.get $i) (i32.const 1)))
      (br_if $again (i32.lt_s (local.get $i) (i32.const 3))))))
```

```
0
1
2
```

That is the whole difference between `block` and `loop`: where a branch to it goes.



A branch to a block goes to its end; a branch to a loop goes to its start.

if 6.3

`if` pops an `i32` and runs its `then` arm if it is non-zero, and its optional `else` arm otherwise:

```
(module
  (import "env" "print" (func $print (param i32 i32)))
  (memory (export "memory") 1)
  (data (i32.const 0) "evenodd")
  (func (export "parity") (param $x i32)
    (if (i32.and (local.get $x) (i32.const 1))
      (then (call $print (i32.const 4) (i32.const 3)))
      (else (call $print (i32.const 0) (i32.const 4))))))
```

called as `parity(4)` and `parity(7)`

```
even
odd
```

In the flat form, `if` pops its condition from the stack, and the arms are separated by `else` and closed by `end`:

```
local.get $x
i32.const 1
i32.and
if
  ;; then
else
  ;; else
end
```

An `if` is also a label, and a branch to it goes to its end, like a `block`.

Blocks with results 6.4

Blocks, loops and `if`s can produce values. They declare what they leave on the stack with a `result`, just as functions do. This makes `if` an expression, like the ternary operator:

```
(module
  (func (export "max") (param $a i32) (param $b i32) (result i32)
    (if (result i32) (i32.gt_s (local.get $a) (local.get $b))
      (then (local.get $a))
      (else (local.get $b)))))
```

called as `max(3, 8)`

```
8
```

When a branch leaves a block that has results, it takes the values with it, from the top of the stack. So a block with a result is how you write an early exit that produces a value:

```
(module
  ;; the leading decimal digit of a positive number
  (func (export "first_digit") (param $n i32) (result i32)
    (block $found (result i32)
      (loop $shrink
        (br_if $found (local.get $n) (i32.lt_u (local.get $n)
            (i32.const 10)))
        (local.set $n (i32.div_u (local.get $n) (i32.const 10)))
        (br $shrink))
      (unreachable))))
```

called as `first_digit(4567)` and `first_digit(9)`

```
4
9
```

`br_if $found` here takes two operands: the value to carry out, and the condition. If the condition is true, the value leaves the block as its result. If it is false, the value stays on the stack, unused. That is harmless: a branch discards whatever is left on the stack inside the construct it targets, beyond the values it carries, so when `br $shrink` jumps back to the top of the loop, the stale value goes with it.

`unreachable` traps if it is ever executed. Here it marks a point control cannot reach (the loop only leaves by branching to `$found`), and it also satisfies the validator, as section 6.8 explains.

Blocks can also take *parameters*: values from the enclosing stack that the block consumes. A loop with parameters can carry values around each iteration on the stack instead of in locals. To write one, you declare its type, like a function's:

```
(module
  (func (export "main") (result i32)
    (local $c i32)
    ;; 10 + 9 + ... + 1, with the total and the counter carried on
        the stack
    (i32.const 0)                       ;; total
    (i32.const 10)                      ;; counter
    (loop $sum (param i32 i32) (result i32)
      (local.set $c)                    ;; stack: total
      (i32.add (local.get $c))          ;; stack: total+c
      (local.tee $c (i32.sub (local.get $c) (i32.const 1)))
      (br_if $sum (local.get $c))       ;; again with (total+c, c-
          1) if c-1 != 0
      drop)))                           ;; leave the total
```

```
55
```

The folded `(i32.add (local.get $c))` has only one operand written inside it; the other is already on the stack. The folded form allows that, since it only decides the order instructions are written in.

Loops with parameters are awkward to write by hand: there is no instruction to duplicate a value, so anything used twice goes through a local anyway. They are mostly useful to compilers that want to avoid creating locals. The point to take away is the typing rule. A branch to a loop carries the loop's *parameters*, since it goes to the start, where those are expected. A branch to a block or `if` carries its *results*, since it goes to the end.

br_table 6.5

`br_table` is the multi-way branch. It pops an `i32` index and branches to the label at that position in its list, or to the last label, the default, if the index is out of range:

```
(module
  (func (export "classify") (param $x i32) (result i32)
    (block $default
      (block $two
        (block $one
          (block $zero
            (br_table $zero $one $two $default (local.get $x)))
          ;; case 0
          (return (i32.const 100)))
        ;; case 1
        (return (i32.const 101)))
      ;; case 2
      (return (i32.const 102)))
    ;; default
    (i32.const -1)))
```

called as `classify(0)`, `classify(1)`, `classify(2)` and `classify(99)`

```
100
101
102
-1
```

This is how a `switch` compiles: one block per case, nested so that the innermost block's end is where the first case's code starts. A branch to `$zero` lands just after `$zero`'s end, which is case 0's code; a branch to `$one` lands after `$one`'s end, which is case 1's code. Each case ends by leaving the whole construct, here with `return`, in general with a `br` to an outer block. If a case should fall through to the next, as in C, it simply does not branch out.

> **Under the hood**
>
> Engines compile a dense `br_table` into a jump table: a bounds check and one indirect jump, as a C compiler would for a dense `switch`. A sparse `switch` with cases at 1, 1000 and 100000 is better compiled as a tree of comparisons, since `br_table` would need a list of a hundred thousand labels.

Compiling loops 6.6

Here are the usual shapes. A `while` loop tests its condition first, so it needs a block to leave and a loop to repeat:

```
;; while (cond) { body }
(block $break
  (loop $continue
    (br_if $break (i32.eqz cond))
    body
    (br $continue)))
```

`break` is `br $break`, and `continue` is `br $continue`. A `do-while` tests at the bottom, so the loop alone is enough:

```
;; do { body } while (cond)
(loop $top
  body
  (br_if $top cond))
```

But a `continue` inside a `do-while` must go to the *test*, not the top. So does a `continue` inside a `for` loop, which must run the increment. Those need one more block, wrapping the body, so that `continue` can branch to its end:

```
;; for (init; cond; step) { body }
init
(block $break
  (loop $top
    (br_if $break (i32.eqz cond))
    (block $continue
      body)                ;; continue is (br $continue)
    step
    (br $top)))
```

Here is that shape, summing the odd numbers below 10 with a `continue` for the even ones:

```
(module
  (func (export "main") (result i32)
    (local $i i32)
    (local $sum i32)
    ;; for (i = 0; i < 10; i++) { if (i % 2 == 0) continue; sum +=
       i; }
    (block $break
      (loop $top
        (br_if $break (i32.ge_s (local.get $i) (i32.const 10)))
        (block $continue
          (br_if $continue (i32.eqz (i32.rem_s (local.get $i)
              (i32.const 2))))
          (local.set $sum (i32.add (local.get $sum) (local.get
              $i))))
        (local.set $i (i32.add (local.get $i) (i32.const 1)))
        (br $top)))
    (local.get $sum)))
```

```
25
```

For compiler writers

Rotating a `while` loop into an `if` around a `do-while` saves one branch per iteration: `(if cond (then (loop body (br_if 0 cond))))`. It is the same transformation a native compiler makes. Engines with optimising tiers often do it themselves, but the baseline compilers that run your code first do not, so it is worth doing in your compiler.

Labelled `break` and `continue`, which leave or restart an outer loop, are no harder: they branch to that loop's `$break` block or `$continue` label, at a greater depth.

Short-circuit operators 6.7

`&&` and `||` must not evaluate their right operand unless they need to, so they are an `if` with a result:

```
;; a && b
(if (result i32) a
  (then b)
  (else (i32.const 0)))

;; a || b
(if (result i32) a
  (then (i32.const 1))
  (else b))
```

When the right operand is cheap and has no side effects, such as a local or a comparison of locals, `select` or a plain `i32.and` of two normalised booleans is shorter and avoids a branch.

How validation handles control flow 6.8

Each block, loop and `if` is validated as if it were a little function:

- Inside a block, the stack starts with only the block's parameters. The instructions inside cannot pop values that were on the stack before the block began. They are not lost, just out of reach until the block ends.
- At the end of a block, the stack must hold exactly the block's results.
- A branch must have the values its target expects on top of the stack.

The interesting case is code *after* an unconditional branch:

```
(module
  (func (export "main") (result i32)
    (block $b (result i32)
      (br $b (i32.const 7))
      i32.add)))       ;; never runs, and has no operands
```

```
7
```

That validates, even though the `i32.add` has nothing to pop. After `br`, `return`, `br_table` and `unreachable`, control cannot continue, so the rest of the block is dead code, and the validator treats the stack there as *polymorphic*: it can supply values of any type, as many as needed. This saves a compiler from having to make dead code type-check, which matters most at the end of a function whose last statement is a `return`:

```
(module
  (func (export "f") (param i32) (result i32)
    (return (local.get 0))))
```

called as `f(3)`

```
3
```

Without the polymorphic rule, that function would need the `i32` the body promised to be on the stack after the `return` too.

When the validator does object, V8's messages name the problem directly:

```
(module
  (func (export "main") (result i32)
    (block $b (result i32)
      (br $b))))
```

```
expected 1 elements on the stack for branch, found 0
```

What you have seen 6.9

`block`, `loop` and `if` nest. A branch names an enclosing one by depth, and goes to the end of a block or `if`, or the start of a loop, carrying values with it. `br_if` branches conditionally and `br_table` by index. Loops, `break`, `continue`, `switch` and short-circuit operators all compile into these shapes.

What does not compile into them so directly is a `goto`, or any control flow graph that is not already nested. That is the next chapter.

Chapter 7

Control Flow Graphs

The last chapter compiled loops and `switch`es straight from source-level constructs. That works when your compiler goes from a syntax tree to WebAssembly in one step. Many compilers do not. They lower functions to a *control flow graph* first, a set of basic blocks joined by jumps, because that is where optimisations are easiest to write. Turning a control flow graph back into nested blocks is the problem this chapter solves.

Why there is no goto 7.1

WebAssembly could have had a `goto`. It does not because structured control flow makes two things cheap that a `goto` makes expensive.

The first is validation. With nested blocks, the validator knows at every point exactly which branch targets are in scope and what type each one expects, and it can check a function in one pass with a simple stack. With arbitrary jumps, it would have to find every path to each instruction and check they agree on the stack's types, which is a dataflow analysis.

The second is compilation. A baseline compiler like V8's Liftoff reads a function once, front to back, and emits machine code as it goes. Nested blocks tell it, at each branch, where the target is and what is live there. With arbitrary jumps it would need to build a graph first.

So the burden moves to the producer: your compiler. In return, you get fast startup on every engine.

You may not need a graph 7.2

If your compiler emits code from a syntax tree, and your language has only structured control flow (`if`, `while`, `for`, `break`, `continue`, `return`, `switch`, labelled breaks), you never have a graph to turn back into structure. The last chapter's patterns are enough. This is the simplest design, and a good one for a new compiler: engines do the low-level optimisations, so a WebAssembly backend gains less from its own graph-level optimisation than a native backend would.

The rest of this chapter is for compilers that do build a graph, or whose language has `goto`.

Reducible graphs 7.3

A control flow graph is *reducible* if every loop in it has a single entry: one block, the loop *header*, through which every path into the loop passes. Graphs built from structured code are always reducible. So are nearly all graphs that come out of optimising them, since the standard optimisations preserve reducibility. Graphs can become *irreducible* through `goto`, through some kinds of jump threading, or through lowering things like coroutines and generators.

Every reducible graph can be turned into nested `block`s and `loop`s without duplicating any code and without adding any variables. Irreducible graphs cannot, and need one of the techniques in section 7.6.

From a graph to blocks 7.4

The algorithm most compilers use today is the one in Norman Ramsey's paper *Beyond Relooper* (2022), which LLVM's WebAssembly backend arrived at in its own form, called CFGStackify. It goes like this:

1. Put the basic blocks in an order where every block comes after all of its forward predecessors: a *reverse postorder*. In this order, every edge either goes forward, or goes backward to a loop header.
2. For each loop header, open a `loop` just before it. A backward edge to the header becomes a `br` to that `loop`. The loop ends after the last block in the loop.
3. For each forward edge that does not go to the very next block, there must be a `block` that ends just before its target, so the edge can be a `br` to that `block`. Open each such `block` as late as possible while still enclosing every branch to it, which is at the start of the target's immediate dominator.
4. Edges to the very next block are fall-through: no branch at all.

The blocks nest properly because of how dominators work: if block X dominates block Y, everything between X and Y in the order is dominated by X too, so the `block` constructs opened in step 3 can always be nested.

Here is a small graph, a diamond:

```
B0:  if (c) goto B2; else goto B1
B1:  x = 10; goto B3
B2:  x = 20; goto B3
B3:  return x + 1
```

In the order B0, B1, B2, B3: the edge B0→B1 falls through. B0→B2 skips over B1, so it needs a block ending before B2. B1→B3 skips over B2, so it needs a block ending before B3. B2→B3 falls through. Both blocks are opened at B0, which dominates everything, with the one that ends later on the outside:

```
(module
  (func (export "diamond") (param $c i32) (result i32)
    (local $x i32)
    (block $B3
      (block $B2
        ;; B0
        (br_if $B2 (local.get $c))
        ;; B1
        (local.set $x (i32.const 10))
        (br $B3))
      ;; B2
      (local.set $x (i32.const 20)))
    ;; B3
    (i32.add (local.get $x) (i32.const 1))))
```

called as `diamond(0)` and `diamond(1)`

```
11
21
```

That is more nesting than a person would write; a person would use an `if`. But it is mechanical, it is always correct, and engines compile it to the same machine code as the `if`. LLVM additionally recognises diamonds and emits `if`-`else` for them, which makes the output smaller.

Here is a graph with a loop, the `for` loop from the last chapter:

```
B0:  i = 0; sum = 0
B1:  if (i >= 10) goto B5          ;; loop header
B2:  if (i % 2 == 0) goto B4
B3:  sum = sum + i
B4:  i = i + 1; goto B1            ;; back edge
B5:  return sum
```

B1 is a loop header, so a `loop` opens before it, and the loop holds B1 to B4. The edge B1→B5 skips forward out of the loop, so a block ending before B5 is opened at B1's dominator, B0, outside the loop. The edge B2→B4 skips B3, so a block ending before B4 is opened at B2:

```
(module
  (func (export "main") (result i32)
    (local $i i32)
    (local $sum i32)
    ;; B0 (locals start at zero)
    (block $B5
      (loop $B1
        (br_if $B5 (i32.ge_s (local.get $i) (i32.const 10)))
        ;; B2
        (block $B4
          (br_if $B4 (i32.eqz (i32.rem_s (local.get $i) (i32.const
              2))))
          ;; B3
          (local.set $sum (i32.add (local.get $sum) (local.get
              $i))))
        ;; B4
        (local.set $i (i32.add (local.get $i) (i32.const 1)))
        (br $B1)))
    ;; B5
    (local.get $sum)))
```

```
25
```

It is exactly the shape the last chapter built by hand from the `for` loop, which is no coincidence.

> **For compiler writers**
>
> If your compiler uses SSA form, the φ-nodes at block entries become `local.set`s on each incoming edge, just before the branch. Values that flow along an edge can also be carried by the branch itself, as block results or loop parameters, but locals are simpler and engines handle them equally well.

The Relooper 7.5

Before structured-control-flow algorithms based on dominators became standard, Emscripten used an algorithm called the *Relooper* (Alon Zakai, 2011), first to turn C into JavaScript and then into WebAssembly. Binaryen still provides it as a library, so a compiler can hand Binaryen a graph of basic blocks and get structured WebAssembly back. If you already use Binaryen to build your output, its Relooper is the quickest way to get started. It handles irreducible graphs too, using the technique in the next section.

Irreducible control flow 7.6

An irreducible graph has a loop with more than one entry. The smallest example is a `goto` into the middle of a loop:

```
    if (skip) goto middle;
top:
    print(1);
middle:
    print(2);
    if (--k > 0) goto top;
```

The loop `top` → `middle` → `top` can be entered at `top` (by falling into it) or at `middle` (by the `goto`). No placement of a single `loop` works, since a `loop` has one entry, its start.

The general fix is a *dispatch loop*: a `loop` around a `br_table` that switches on a local variable saying which block to run next. Each block sets the variable and branches back to the dispatcher:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "f") (param $skip i32) (param $k i32)
    (local $next i32)                  ;; 0: top, 1: middle, 2:
        exit
    (local.set $next (local.get $skip))
    (loop $dispatch
      (block $exit
        (block $middle
          (block $top
            (br_table $top $middle $exit (local.get $next)))
          ;; top
          (call $print (i32.const 1)))
        ;; middle (top falls into it)
        (call $print (i32.const 2))
        (local.set $k (i32.sub (local.get $k) (i32.const 1)))
        (local.set $next (select (i32.const 0) (i32.const 2)
                                 (i32.gt_s (local.get $k)
                                     (i32.const 0))))
        (br $dispatch)))
    ;; exit
    ))
```

The first call falls into `top` and goes round twice. The second skips straight to `middle` the first time round:

called as `f(0, 2)` and `f(1, 2)`

```
1
2
1
2
2
1
2
```

Only the blocks involved in the irreducible loop need to go through the dispatcher. The rest of the function can be structured normally. The cost is an indirect jump per transfer, and an engine cannot optimise the loop as well as a natural one, so this is for the rare case, not the common one.

The other fix is *node splitting*: duplicate the blocks of the loop that are reachable from the second entry, so each copy has a single entry. It produces faster code at the cost of size, and in the worst case the size grows exponentially, so compilers use it only for small regions.

> **Generators and coroutines**
>
> A generator or `async` function compiled without stack switching becomes exactly this shape: a dispatch loop whose state variable says where to resume, with the state saved in memory between calls. The C `switch` trick known as Duff's device or "protothreads" is the same idea.

What you have seen 7.7

WebAssembly's structured control flow puts the work of structuring on the producer, to make validation and compilation single-pass. A compiler that works from a syntax tree can emit structure directly. One that builds a control flow graph can restore structure from any reducible graph with an algorithm based on dominators, placing a `loop` at each header and a `block` before each forward-branch target. Irreducible graphs need a dispatch loop or node splitting.

The next chapter looks at modules as a whole.

Chapter 8

Modules and Instances

So far modules have been containers for functions. This chapter looks at everything else a module can hold, how its contents are numbered, how it connects to its host and to other modules through imports and exports, and what happens, step by step, when it is instantiated.

What a module holds 8.1

A module is a list of definitions of a few kinds. Each kind has its own index space, and imports of that kind come first in it.

Kind	What it is	Covered in
type	a function signature, or (with GC) a struct or array type	Chapters 5, 22
function	code	Chapter 5
table	an array of references, such as function references	Chapter 11
memory	a linear array of bytes	Chapter 9
global	a single typed value, mutable or not	Chapter 11
tag	an exception's signature	Chapter 20
element segment	references to put into a table	Chapter 11
data segment	bytes to put into a memory	Chapter 9

Then there are the connections: *imports*, which name things the host must supply; *exports*, which name things the module gives to the host; and an optional *start* function, run when the module is instantiated.

In the binary format, each kind lives in its own section, and the sections must come in a fixed order:

Id	Section	Holds
1	Type	the type definitions
2	Import	the imports, of every kind
3	Function	the type index of each defined function
4	Table	the defined tables
5	Memory	the defined memories
13	Tag	the defined tags
6	Global	the defined globals, with their initial values
7	Export	the exports
8	Start	the index of the start function
9	Element	the element segments
12	Data count	how many data segments there are
10	Code	the bodies of the defined functions
11	Data	the data segments
0	Custom	anything else, such as names; allowed anywhere

The ids are in the order sections were invented, not the order they appear, which is why the tag section (added with exceptions) is 13 but sits between memories and globals, and the data count section (added with bulk memory, so that the code can refer to data segments before they are defined) is 12 but sits before the code. The order is designed so that everything an engine needs to compile a function body is known before the first body arrives.

The text format is more relaxed. You can write definitions in any order, and the assembler sorts them into sections.

Imports and exports 8.2

Any kind of definition, except types and segments, can be imported or exported. An import has a two-level name, a module and a field, and a description of what it expects:

```
(module
  (import "env" "log"     (func $log (param i32)))
  (import "env" "memory"  (memory 1))
  (import "env" "table"   (table 4 funcref))
  (import "env" "gravity" (global $g f32))
  (import "env" "frame"   (global $frame (mut i32))))
```

The two-level name is a convention. Hosts usually treat the first part as the name of a JavaScript object and the second as a

property of it, so the imports above would be supplied as `{ env: { log, memory, table, gravity, frame } }`.

An export has a single name and refers to a definition by index:

```
(module
  (func $f)
  (memory $m 1)
  (global $answer i32 (i32.const 42))
  (export "f" (func $f))
  (export "memory" (memory $m))
  (export "answer" (global $answer)))
```

The text format also lets you write the export inside the definition, as the book has been doing with `(func (export "main") ...)`, and the same for imports: `(func $log (import "env" "log") (param i32))`. Both are shorthand for the separate forms.

A definition can be exported under several names, and the export names must all be different.

Modules and instances 8.3

A *module* is code: a validated, compiled, stateless thing. An *instance* is a module brought to life with its own memory, tables and globals, and its imports filled in. One module can be instantiated many times, and each instance has its own state:

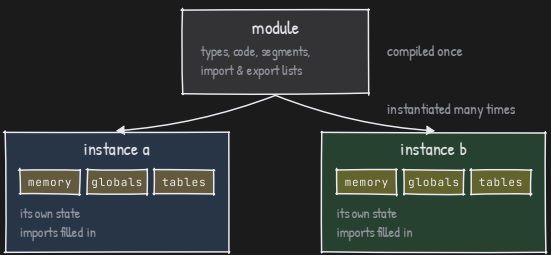
```
(module
  (global $n (mut i32) (i32.const 0))
  (func (export "next") (result i32)
    (global.set $n (i32.add (global.get $n) (i32.const 1)))
    (global.get $n)))
```

```
import fs from 'node:fs';

const module = new
    WebAssembly.Module(fs.readFileSync('main.wasm'));
const a = new WebAssembly.Instance(module);
const b = new WebAssembly.Instance(module);
console.log(a.exports.next(), a.exports.next(), a.exports.next());
console.log(b.exports.next());
```

```
1 2 3
1
```

Compiling is the expensive part. Instantiating an already-compiled module is cheap, so if you need several copies of a module (one per game entity running a script, say, or one per worker thread), compile it once and instantiate it many times.



A module is compiled code; each instance has its own state and imports.

Linking modules together 8.4

Since one instance's exports are ordinary JavaScript values, they can be passed as another instance's imports. This is how two WebAssembly modules are linked: by the host, at instantiation time. Here, one module owns a memory and a function, and another imports both:

```
(module
  (memory (export "memory") 1)
  (data (i32.const 0) "linked!")
  (func (export "double") (param i32) (result i32)
    (i32.mul (local.get 0) (i32.const 2))))
```

```
(module
  (import "lib" "memory" (memory 1))
  (import "lib" "double" (func $double (param i32) (result i32)))
  (func (export "first_byte_doubled") (result i32)
    (call $double (i32.load8_u (i32.const 0)))))
```

```
import fs from 'node:fs';

const load = (f) => new WebAssembly.Module(fs.readFileSync(f));
const lib = new WebAssembly.Instance(load('lib.wasm'));
const app = new WebAssembly.Instance(load('app.wasm'), { lib:
    lib.exports });
console.log(app.exports.first_byte_doubled());   // 'l' is 108
```

```
216
```

The two instances share one memory: `app` reads the bytes `lib`'s data segment wrote. And `app`'s call to `$double` goes directly from one module's code to the other's, with no JavaScript in between. Engines recognise that the import is a WebAssembly function and call it directly.

This is the basis of dynamic linking schemes, such as Emscripten's side modules, and of splitting a large program so that rarely used

parts load later. Each scheme adds conventions on top, such as how to share a stack pointer and how to relocate data, but the mechanism underneath is just this.

Instantiation, step by step 8.5

When a module is instantiated, these things happen in order:

1. **Imports are resolved.** Each import is looked up in the import object, and checked against what the module expects. A function must have exactly the right signature, a memory must be at least as large as the minimum declared, a global must have the right type and mutability. A mismatch is a `WebAssembly.LinkError`, and nothing else happens.
2. **Definitions are created.** The module's own memories, tables and globals are allocated. Globals get their initial values, which are *constant expressions*: a constant, a `global.get` of an imported or earlier global, and (since WebAssembly 3.0) a few arithmetic instructions.
3. **Segments are applied.** Active element segments are copied into their tables, and active data segments into their memories, in order. If one does not fit, instantiation traps, partway through: segments already applied stay applied, which matters only if the memory or table was imported and is visible elsewhere.
4. **The start function runs**, if there is one.

A JavaScript function has no WebAssembly signature, so it satisfies a function import of any signature: the engine converts the arguments on each call, as Chapter 15 describes. This module expects an `i64` parameter, and is given `console.log`, which is fine:

```
(module
  (import "env" "print_i32" (func (param i64))))
```

```
import fs from 'node:fs';

const module = new WebAssembly.Module(fs.readFileSync('m.wasm'));
const instance = new WebAssembly.Instance(module, { env: {
    print_i32: console.log } });
console.log('instantiated', Object.keys(instance.exports));
```

```
instantiated []
```

A link error on a function import happens when the import is a *WebAssembly* function with a different signature, such as another instance's export:

```
(module
  (func (export "print_i32") (param i32)))
```

```
import fs from 'node:fs';

const load = (f) => new WebAssembly.Module(fs.readFileSync(f));
const other = new WebAssembly.Instance(load('p.wasm'));
try {
  new WebAssembly.Instance(load('m.wasm'), { env: { print_i32:
      other.exports.print_i32 } });
} catch (e) {
  console.log(e.constructor.name + ': ' + e.message);
}
```

```
LinkError: WebAssembly.Instance(): Import #0 module="env"
    function="print_i32": imported function does not match the
    expected type
```

And a missing import is a `TypeError` if the whole module object is missing, and a `LinkError` if only the field is.

The start function 8.6

A module can name a function to run when it is instantiated. It must take no parameters and return nothing:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func $init
    (call $print (i32.const 1)))
  (start $init)
  (func (export "main")
    (call $print (i32.const 2))))
```

```
1
2
```

The start function runs before `instantiate` returns, so the host has no access to the instance's exports while it runs. In particular, if the module imports a function that needs to read the module's own exported memory, that function will fail when called from the start function: the host does not have the memory yet. For this reason most toolchains do not use the start function for their initialisation. They export an initialisation function, conventionally `_initialize` or `_start`, and the host calls it explicitly after instantiating. Emscripten, Rust's `wasm-bindgen` and WASI all do this.

What you have seen 8.7

A module holds types, functions, tables, memories, globals, tags and segments, each in its own index space, plus imports, exports

and an optional start function. The binary format puts each kind in a section, in a fixed order designed for streaming compilation. A module is stateless code; an instance has state. Instances are linked by passing one's exports as another's imports. Instantiation resolves imports, creates definitions, applies segments and runs the start function, in that order.

That ends Part II. Part III is about memory.

PART III

Memory and Indirection

A WebAssembly function can keep values in locals and on the operand stack, but anything with an address lives in *linear memory*: one flat array of bytes per memory, bounds-checked on every access. This part covers the instructions that use it, how a compiler lays out a language's data and its stack inside it, and the two other kinds of state a module has: globals, and the tables that make indirect calls possible.

Chapter 9

Linear Memory

A *linear memory* is an array of bytes, indexed from zero, that a module can read and write with load and store instructions. It is where a compiled program keeps everything that has an address: its static data, its heap, and the parts of its stack that locals cannot hold. This chapter covers the memory itself and every instruction that uses it.

Declaring a memory 9.1

A memory's size is counted in *pages* of 64 KiB (65,536 bytes). A declaration gives the initial size, and optionally a maximum:

```
(module
  (memory $m 1 16))     ;; starts at 64 KiB, may grow to 1 MiB
```

Without a maximum, a memory can grow until the engine refuses. For a 32-bit memory, the hard limit is 65,536 pages, which is 4 GiB, since addresses are `i32`s. Browsers impose their own limits below that, and a tab may fail to grow far sooner if the device is short of memory.

Most modules have one memory, and the instructions in this chapter use memory 0 unless told otherwise. WebAssembly 3.0 allows several (section 9.9).

Loads and stores 9.2

`i32.load` pops an address and pushes the 4 bytes there, as an `i32`. `i32.store` pops an address and a value, and writes the value:

```
(module
  (memory 1)
  (func (export "main") (result i32)
    (i32.store (i32.const 100) (i32.const 0x12345678))
    (i32.load (i32.const 100))))
```

```
305419896
```

Memory is little-endian, always, whatever the hardware:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (memory 1)
  (func (export "main")
    (i32.store (i32.const 0) (i32.const 0x12345678))
    (call $print (i32.load8_u (i32.const 0)))     ;; 0x78
    (call $print (i32.load8_u (i32.const 3)))))   ;; 0x12
```

```
120
18
```

Every number type has a load and a store. The integer types also have *narrow* versions that read or write fewer bytes. A narrow load extends the value to the full width, signed or unsigned. A narrow store keeps the low bytes:

Load	Reads	Store	Writes
`i32.load`	4 bytes	`i32.store`	4 bytes
`i32.load8_s`, `i32.load8_u`	1 byte, extended	`i32.store8`	the low byte
`i32.load16_s`, `i32.load16_u`	2 bytes, extended	`i32.store16`	the low 2 bytes
`i64.load`	8 bytes	`i64.store`	8 bytes
`i64.load8_s` … `i64.load32_u`	1, 2 or 4 bytes, extended	`i64.store8` … `i64.store32`	the low 1, 2 or 4 bytes
`f32.load`	4 bytes	`f32.store`	4 bytes
`f64.load`	8 bytes	`f64.store`	8 bytes

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (memory 1)
  (func (export "main")
    (i32.store8 (i32.const 0) (i32.const 0xfff0))   ;; stores 0xf0
    (call $print (i32.load8_u (i32.const 0)))
    (call $print (i32.load8_s (i32.const 0)))))
```

```
240
-16
```

These are what a compiler uses for the narrow types of its source language. A `u8` field is loaded with `load8_u` and stored with `store8`, and arrives in an `i32` already normalised.

Offsets and alignment 9.3

Every load and store has two immediates, written after the instruction: an `offset` and an `align`:

```
(i32.load offset=8 align=4 (local.get $p))
```

The *effective address* is the address popped from the stack plus the offset. The offset is how struct field access is usually compiled: the struct's address is on the stack, and the field's position within it is the offset, at no cost:

```
(module
  (memory 1)
  ;; struct Point { i32 x; i32 y; i32 z; }
  (func $set_point (param $p i32) (param $x i32) (param $y i32)
      (param $z i32)
    (i32.store offset=0 (local.get $p) (local.get $x))
    (i32.store offset=4 (local.get $p) (local.get $y))
    (i32.store offset=8 (local.get $p) (local.get $z)))
  (func (export "main") (result i32)
    (call $set_point (i32.const 64) (i32.const 1) (i32.const 2)
        (i32.const 3))
    (i32.load offset=8 (i32.const 64))))     ;; p->z
```

```
3
```

The addition does not wrap. The address is treated as an unsigned 32-bit number, the offset is added with full precision, and if the result is past the end of memory, the access traps. So `(i32.load offset=8 (i32.const -4))` is an access at address 4,294,967,300, not at address 4.

`align` is a hint about the address's alignment, written in bytes in the text format and as a power of two in the binary. It is never checked: unaligned accesses are allowed, and work. The hint lets an engine choose faster instructions on hardware where unaligned access is slow, but on x86 and modern ARM it makes no difference, and engines largely ignore it. It may not be larger than the access itself (`align=8` on an `i32.load` is a validation error). When you leave it out, it defaults to the access size, which is right for naturally aligned data.

> **For compiler writers**
>
> Emit the true alignment when you know it, and `align=1` when you do not, such as for a packed struct. Lying in the other direction, claiming alignment you do not have, is allowed and still correct, since the hint is never trusted, but it may be slow on some hardware.

Out of bounds 9.4

An access past the end of memory traps:

```
(module
  (memory 1)
  (func (export "main") (result i32)
    (i32.load (i32.const 65535))))     ;; needs bytes 65535..65538
```

```
RuntimeError: memory access out of bounds
```

The check covers the whole access, so a 4-byte load at the last byte fails even though its first byte is in bounds. Nothing is read or written when an access traps.

Address 0 is valid, and holds whatever was stored there. There is no unmapped page to catch null pointer dereferences:

```
(module
  (memory 1)
  (func (export "main") (result i32)
    (i32.store (i32.const 0) (i32.const 99))
    (i32.load (i32.const 0))))
```

```
99
```

If your language has null pointers, a null dereference will quietly read or write the bytes at address 0 unless you check for it. Many toolchains leave the first few bytes or kilobytes of memory unused, so that at least writes through null do not damage anything, and so that a debugger shows obviously wrong values. Chapter 10 comes back to this.

Under the hood

Engines do not compare every address against the memory size. On 64-bit hardware, a 32-bit memory is placed at the start of a reservation of 8 GiB or more of virtual address space, of which only the memory's current size is accessible. Any `i32` address plus any 32-bit offset lands somewhere in that reservation, and an access beyond the memory hits an inaccessible page, which the engine's signal handler turns into a trap. Bounds checks cost nothing at all. Chapter 25 says what this means for 64-bit memories, where the trick does not work.

Growing memory 9.5

`memory.size` pushes the current size in pages. `memory.grow` pops a number of pages to add, and pushes the *old* size, or −1 if the memory cannot grow:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (memory 1 3)
  (func (export "main")
    (call $print (memory.size))
    (call $print (memory.grow (i32.const 1)))
    (call $print (memory.size))
    (call $print (memory.grow (i32.const 5)))      ;; past the
        maximum
    (call $print (memory.size))))
```

```
1
1
2
-1
2
```

New pages are zero-filled. Memory never shrinks: there is no instruction to give pages back. A program that allocates a large

buffer once keeps that much memory for as long as the instance lives, which matters in a browser tab. Chapter 10 discusses allocators.

Data segments 9.6

A *data segment* is a run of bytes in the module that is copied into memory. The ones you have seen are *active*: copied at instantiation, to an address given by a constant expression:

```
(module
  (import "env" "print" (func $print (param i32 i32)))
  (memory (export "memory") 1)
  (data (i32.const 0) "abc")
  (data (i32.const 3) "\64\65\66")      ;; hex escapes: "def"
  (func (export "main")
    (call $print (i32.const 0) (i32.const 6))))
```

```
abcdef
```

Strings in the text format can contain any bytes, written as `\xx` hex escapes, so data segments can hold binary data, such as a precomputed table, as well as text.

A *passive* segment, declared without an address, is not copied automatically. Instead, `memory.init` copies some or all of it to an address chosen at runtime, and `data.drop` discards it once it is no longer needed, so the engine can free it:

```
(module
  (import "env" "print" (func $print (param i32 i32)))
  (memory (export "memory") 1)
  (data $greeting "Hello, again")       ;; passive: no address
  (func (export "main")
    ;; copy 5 bytes, from offset 0 of the segment, to address 1000
    (memory.init $greeting (i32.const 1000) (i32.const 0)
        (i32.const 5))
    (data.drop $greeting)
    (call $print (i32.const 1000) (i32.const 5))))
```

```
Hello
```

Passive segments are for data whose address is not known until runtime: data copied into a freshly allocated block, or, with threads, data that should be initialised by one thread rather than by every instance that shares the memory.

Filling and copying 9.7

`memory.fill` sets a range of bytes to a value, and `memory.copy` copies a range, correctly even when the source and destination overlap,

like C's `memmove`:

```
(module
  (import "env" "print" (func $print (param i32 i32)))
  (memory (export "memory") 1)
  (data (i32.const 0) "abcdefgh")
  (func (export "main")
    (memory.copy (i32.const 2) (i32.const 0) (i32.const 4))   ;;
        dest, src, length
    (call $print (i32.const 0) (i32.const 8))
    (memory.fill (i32.const 0) (i32.const 0x2d) (i32.const 3)) ;;
        dest, byte, length
    (call $print (i32.const 0) (i32.const 8))))
```

```
ababcdgh
---bcdgh
```

The bulk memory instructions arrived in WebAssembly 2.0. Before them, compilers emitted byte-copying loops, and old code you read may still have them. They are worth using: engines turn them into calls to the platform's optimised `memmove` and `memset`, and they are much smaller than loops.

Memory from JavaScript 9.8

On the JavaScript side, a memory is a `WebAssembly.Memory` object whose `buffer` property is an `ArrayBuffer` holding its bytes. Typed arrays over that buffer read and write the same bytes the module does, which is how large amounts of data get in and out:

```
(module
  (memory (export "memory") 1)
  (func (export "sum") (param $p i32) (param $n i32) (result i32)
    (local $total i32)
    (block $done
      (loop $next
        (br_if $done (i32.eqz (local.get $n)))
        (local.set $total (i32.add (local.get $total) (i32.load8_u
            (local.get $p))))
        (local.set $p (i32.add (local.get $p) (i32.const 1)))
        (local.set $n (i32.sub (local.get $n) (i32.const 1)))
        (br $next)))
    (local.get $total)))
```

```
import fs from 'node:fs';

const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'));
const { memory, sum } = instance.exports;
const bytes = new Uint8Array(memory.buffer);
bytes.set([10, 20, 30, 40], 256);          // write four bytes at
    address 256
console.log(sum(256, 4));
```

```
100
```

There is one trap for the unwary here. When a memory grows, its old `ArrayBuffer` is *detached*: it becomes zero bytes long, and any typed array over it becomes empty. Code that keeps a typed array across a call that might grow memory will find it unexpectedly useless:

```
(module
  (memory (export "memory") 1)
  (func (export "grow") (drop (memory.grow (i32.const 1)))))
```

```
import fs from 'node:fs';

const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'));
const { memory, grow } = instance.exports;
let view = new Uint8Array(memory.buffer);
console.log('before:', view.length);
grow();
console.log('after: ', view.length);
view = new Uint8Array(memory.buffer);       // take a fresh view
console.log('fresh: ', view.length);
```

```
before: 65536
after:  0
fresh:  131072
```

The safe habit is to create typed arrays over `memory.buffer` just before using them, rather than caching them, or to check `view.length === 0` and recreate them. Chapter 16 has more on moving data across the boundary.

Multiple memories and 64-bit memories 9.9

WebAssembly 3.0 added two extensions to memory.

Multiple memories: a module can define or import several memories, and every memory instruction takes an optional memory index:

```
(module
  (memory $main 1)
  (memory $scratch 1)
  (func (export "main") (result i32)
    (i32.store $scratch (i32.const 0) (i32.const 7))
    (i32.store $main (i32.const 0) (i32.const 9))
    (i32.load $scratch (i32.const 0))))
```

```
7
```

This is useful for keeping data the program must never touch through its ordinary pointers apart from the rest, such as a garbage collector's metadata, or a sanitiser's shadow memory, or a buffer shared with the host that should not be mixed into the heap.

64-bit memories, from the *memory64* proposal, take `i64` addresses and can exceed 4 GiB:

```
(module
  (memory i64 1)
  (func (export "main") (result i32)
    (i32.store (i64.const 8) (i32.const 5))
    (i32.load (i64.const 8))))
```

```
5
```

They cost a bounds check on every access, since the guard-page trick in section 9.4 does not extend to 64-bit addresses, and few browser programs need more than 4 GiB. For a game engine, 32-bit memory is the right default. Chapter 18 covers how to detect support for each feature.

What you have seen 9.10

A linear memory is a growable, bounds-checked array of bytes in 64 KiB pages, little-endian, with address 0 as valid as any other. Loads and stores come in every width, with an offset immediate for free address arithmetic and an alignment hint that is never checked. Data segments initialise memory, actively at instantiation or passively on demand, and `memory.copy` and `memory.fill` move bytes in bulk. JavaScript sees memory as an `ArrayBuffer` that is replaced whenever memory grows.

The next chapter is about what a compiler puts in this memory, and where.

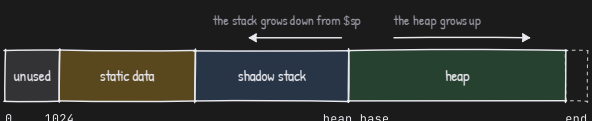
Chapter 10

Laying Out Memory

Linear memory comes with no structure at all. It is a run of zero bytes, and a compiler has to decide what goes where: the program's constants and static variables, a stack for the values that cannot live in locals, and a heap. This chapter goes through those decisions and builds the two pieces nearly every compiler needs: a shadow stack, and an allocator.

A memory map 10.1

Here is the layout most toolchains use, LLVM's among them:



A typical layout of linear memory: static data, then the shadow stack growing down towards it, then the heap growing up.

- **Low addresses are left unused.** LLVM starts static data at 1024 by default. The gap means a write through a null pointer, or through a small offset from one, lands in memory nobody uses.
- **Static data** comes next: string literals, constant tables, and initialised and zero-initialised global variables. Their addresses are fixed at compile time, so code refers to them by constant address.
- **The shadow stack** follows, growing *downwards* from its top towards the static data. Its size is chosen at link time; LLVM's default is 64 KiB, and real programs usually need more.
- **The heap** starts above the stack and grows upwards, using `memory.grow` when it runs out of room.

With this layout, a stack overflow runs down into the static data and corrupts it silently. LLVM's linker has an option, `--stack-first`, that puts the stack at the very bottom of memory instead, below the static data. Then an overflow takes the stack pointer below address 0, where it wraps around to a huge unsigned address, and the next access through it traps as out of bounds. That turns a

silent corruption into a crash, at the cost of giving up the unused low area for catching null pointers. Either way, a compiler can also check the stack pointer against a limit in each prologue, and trap explicitly.

None of this is part of WebAssembly. It is a convention between the compiler, its runtime library and the host. Your compiler can use a different layout, as long as everything that shares a memory agrees.

The shadow stack 10.2

Chapter 5 explained that the engine's call stack is invisible, and that locals have no addresses. A variable whose address is taken must live in linear memory instead. The usual home for such variables is a *shadow stack*: an area of linear memory, managed exactly like a native stack, with the stack pointer kept in a mutable global.

Each function that needs stack memory subtracts its frame size from the stack pointer on entry, uses the memory between the new and old values, and adds the frame size back before it returns. Functions that need no stack memory do not touch the stack pointer at all, and in practice that is most of them.

Here is a C function that needs a frame, because it passes the addresses of two locals to another function:

```
void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }

int test(void) {
    int x = 1, y = 2;
    swap(&x, &y);
    return x * 10 + y;
}
```

And here it is, compiled by hand the way a compiler would:

```
(module
  (memory 1)
  (global $sp (mut i32) (i32.const 65536))      ;; the stack
      starts at the top of memory

  (func $swap (param $a i32) (param $b i32)
    (local $t i32)
    (local.set $t (i32.load (local.get $a)))
    (i32.store (local.get $a) (i32.load (local.get $b)))
    (i32.store (local.get $b) (local.get $t)))

  (func (export "main") (result i32)
    (local $frame i32)
    ;; prologue: an 8-byte frame, x at frame+0 and y at frame+4
    (local.set $frame (i32.sub (global.get $sp) (i32.const 8)))
    (global.set $sp (local.get $frame))

    (i32.store offset=0 (local.get $frame) (i32.const 1))
    (i32.store offset=4 (local.get $frame) (i32.const 2))
    (call $swap
      (local.get $frame)
      (i32.add (local.get $frame) (i32.const 4)))

    ;; epilogue, with the return value computed first
    (i32.add
      (i32.mul (i32.load offset=0 (local.get $frame)) (i32.const
          10))
      (i32.load offset=4 (local.get $frame)))
    (global.set $sp (i32.add (local.get $frame) (i32.const 8)))))
```

```
21
```

The frame pointer is kept in a local, `$frame`, so that every variable in the frame is reached as `$frame` plus a constant offset, which the load and store `offset` immediates make free.

Only `x` and `y` live in memory, because only their addresses are taken. Everything else, including `$t` in `swap`, stays in locals, where the engine can keep it in a register.

> **For compiler writers**
>
> Finding the variables that need a stack slot is an *escape analysis*: a variable needs memory if its address is taken and might be used after the point where you could reload it into a local. The simple rule, "a stack slot for any variable whose address is ever taken", is correct and is what unoptimised compilers do. Note that the same applies to aggregates too large or too awkward to keep in locals, such as arrays and structs passed by reference.

The stack pointer in LLVM's convention is a global named `__stack_pointer`. It is usually not exported, but when several modules share a memory, as in dynamic linking, it must be imported by all of them from a single place, so they agree on where the stack is.

If a function returns early, from several places, it must restore the stack pointer on every path. Wrapping the body in a block and

branching to an epilogue at its end is the easy way. If the function can be left by an exception (Chapter 20), the stack pointer must be restored by whoever catches it, which is one reason compilers save it before each `try`.

Struct layout 10.3

A compiler chooses the layout of each aggregate type: the offset of each field, and the alignment and size of the whole. WebAssembly does not care, since every access is just an address and an offset, and unaligned accesses work. But the C rules are a good default, and are what you need if your structs are shared with C code compiled to the same memory:

- each field is placed at the next offset that is a multiple of its alignment, which for numbers is their size;
- the struct's alignment is the largest alignment of its fields;
- the struct's size is rounded up to a multiple of its alignment, so that arrays of it keep every element aligned.

For example:

Field	Type	Size	Offset
`tag`	`u8`	1	0
(padding)		3	1
`value`	`i32`	4	4
`count`	`u16`	2	8
(padding)		2	10
	total	**12**	alignment 4

Reordering the fields from largest to smallest alignment (`value`, `count`, `tag`) would make it 8 bytes. A language that does not promise field order, such as Rust, can do that reordering automatically.

Natural alignment matters less for speed on WebAssembly than it once did on native hardware, but it is still worth having: engines may emit different code for unaligned accesses on some ARM chips, and the atomic instructions of Chapter 23 *require* natural alignment and trap without it.

Passing and returning aggregates 10.4

A struct passed by value can be split into its fields and passed as several parameters, which keeps it in registers. Returning it can use multiple results (Chapter 5). For anything larger, the common convention, and the one C uses on WebAssembly, is:

- to pass a large struct, the caller copies it to its own stack frame and passes a pointer;
- to return one, the caller allocates space in its frame and passes a pointer to it as a hidden first parameter, which the callee fills in.

A compiler that will never call C, and never be called by it, can choose its own convention. Splitting small structs into scalars, up to four or so fields, and using memory beyond that, is a reasonable rule.

A bump allocator 10.5

The simplest heap is a *bump allocator*: a pointer to the first free byte, which each allocation moves forwards. When the pointer passes the end of memory, it grows memory. Nothing is ever freed individually:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (memory 1)
  (global $heap (mut i32) (i32.const 1024))

  ;; allocate $size bytes, aligned to 8
  (func $alloc (param $size i32) (result i32)
    (local $p i32)
    (local $end i32)
    (local.set $p (i32.and (i32.add (global.get $heap) (i32.const
        7)) (i32.const -8)))
    (local.set $end (i32.add (local.get $p) (local.get $size)))
    ;; grow memory if the block does not fit
    (if (i32.gt_u (local.get $end) (i32.mul (memory.size)
        (i32.const 65536)))
      (then
        (if (i32.eq
              (memory.grow
                (i32.shr_u
                  (i32.sub
                    (i32.add (local.get $end) (i32.const 65535))
                    (i32.mul (memory.size) (i32.const 65536)))
                  (i32.const 16)))
              (i32.const -1))
          (then unreachable))))       ;; out of memory
    (global.set $heap (local.get $end))
    (local.get $p))

  (func (export "main")
    (call $print (call $alloc (i32.const 3)))
    (call $print (call $alloc (i32.const 100)))
    (call $print (call $alloc (i32.const 200000)))   ;; needs 3
        more pages
    (call $print (memory.size))))
```

```
1024
1032
1136
4
```

Rounding up to a multiple of 8 is the usual `(p + 7) & ~7`. The amount to grow is the shortfall rounded up to whole pages: add 65,535 and shift right by 16, since a page is 2^{16} bytes.

A bump allocator on its own is only right for programs that never free memory, or that free everything at once. That second case covers more than it sounds like. A game can give each frame an *arena*: allocate freely during the frame, then reset the pointer to where it started when the frame ends, freeing everything in one store. Level loading can use another arena, reset when the level is unloaded.

For general-purpose `malloc` and `free`, compilers link in a real allocator, compiled to WebAssembly like any other code. Emscripten uses dlmalloc or its own *emmalloc*; Rust's standard library uses dlmalloc too. Since memory never shrinks, an

allocator's peak usage is what the tab keeps, so fragmentation matters more than usual.

Null pointers 10.6

Since address 0 is valid memory, dereferencing a null pointer does not trap. If your language promises that it will, you have three options:

- check before each dereference, and trap or raise your own error if the pointer is zero. The check is a compare and a branch, cheap but not free;
- make the address of every valid object large and every invalid one small, and rely on the reserved low memory catching the damage, without a guarantee;
- use the garbage-collected references of Chapter 22, where `null` is a real value and dereferencing it traps, but which do not live in linear memory at all.

Checks that can be proven redundant, such as a second dereference of the same pointer, can be removed by your compiler. Engines will not remove them for you, because they cannot know what the check means.

Strings, arrays and slices 10.7

A dynamically sized array in linear memory is usually represented as a pointer and a length, passed together. With multiple results and multiple parameters, the pair can stay in two locals rather than being stored as a struct:

```
(module
  (import "env" "print" (func $print (param i32 i32)))
  (memory (export "memory") 1)
  (data (i32.const 100) "  trim me  ")

  ;; return the slice without leading and trailing spaces
  (func $trim (param $p i32) (param $n i32) (result i32 i32)
    (block $done
      (loop $front
        (br_if $done (i32.eqz (local.get $n)))
        (br_if $done (i32.ne (i32.load8_u (local.get $p))
            (i32.const 32)))
        (local.set $p (i32.add (local.get $p) (i32.const 1)))
        (local.set $n (i32.sub (local.get $n) (i32.const 1)))
        (br $front)))
    (block $done
      (loop $back
        (br_if $done (i32.eqz (local.get $n)))
        (br_if $done (i32.ne (i32.load8_u (i32.add (local.get $p)
            (i32.sub (local.get $n) (i32.const 1))))
                             (i32.const 32)))
        (local.set $n (i32.sub (local.get $n) (i32.const 1)))
        (br $back)))
    (local.get $p)
    (local.get $n))

  (func (export "main")
    (call $print (call $trim (i32.const 100) (i32.const 11)))))
```

```
trim me
```

The two results of `$trim` are exactly the two parameters `$print` expects, so the call passes them straight through.

What you have seen 10.8

A compiler decides the layout of linear memory: a reserved low area, static data at fixed addresses, a shadow stack for address-taken locals with its pointer in a global, and a heap grown with `memory.grow`. Struct layout follows whatever rules you choose, and C's are a good default. Allocators are ordinary WebAssembly code, and a bump allocator with per-frame arenas suits games well. Null is a valid address, so null checks are your compiler's job.

The next chapter covers the rest of a module's state: globals and tables.

Chapter 11

Globals and Tables

Linear memory holds bytes. Two other kinds of state hold things that are not bytes. A *global* holds a single typed value, which may be shared with the host. A *table* holds references, most often references to functions, and is what makes function pointers, virtual calls and closures possible in a machine whose code has no addresses.

Globals 11.1

A global is declared with a type, a mutability, and an initial value:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (global $limit i32 (i32.const 100))          ;; immutable
  (global $count (mut i32) (i32.const 0))      ;; mutable
  (func (export "main")
    (global.set $count (i32.add (global.get $count) (global.get
        $limit)))
    (global.set $count (i32.add (global.get $count) (global.get
        $limit)))
    (call $print (global.get $count))))
```

```
200
```

`global.get` pushes a global's value and `global.set` pops a value into a mutable one. Setting an immutable global is a validation error.

The initial value must be a *constant expression*. Originally that meant a single constant, or a `global.get` of an imported immutable global. WebAssembly 3.0 adds `i32.add`, `i32.sub`, `i32.mul` and their `i64` forms, so a compiler can write `base + 16` as an initialiser:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (global $base i32 (i32.const 1024))
  (global $after_header i32 (i32.add (global.get $base) (i32.const
      16)))
  (func (export "main")
    (call $print (global.get $after_header))))
```

```
1040
```

Engines treat immutable globals as constants, which they can fold into the code that uses them. Mutable globals cost a load or store, much like a static variable in native code, though engines often

keep the most used ones, such as a stack pointer, in a register within a function.

Globals shared with the host 11.2

A global can be imported or exported. On the JavaScript side it is a `WebAssembly.Global` object with a `value` property:

```
(module
  (import "env" "step" (global $step i32))
  (global $total (export "total") (mut i32) (i32.const 0))
  (func (export "tick")
    (global.set $total (i32.add (global.get $total) (global.get
        $step)))))
```

```
import fs from 'node:fs';

const step = new WebAssembly.Global({ value: 'i32', mutable: false
    }, 5);
const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'), { env: {
    step } });
const { tick, total } = instance.exports;
tick(); tick(); tick();
console.log(total.value);
total.value = 1000;          // the host can write an exported
    mutable global
tick();
console.log(total.value);
```

```
15
1005
```

Imported immutable globals are how a module receives configuration that is fixed at instantiation: a memory base address in a dynamically linked module, say, or a feature flag. Since they are constants to the code that uses them, an engine can specialise the code for their values.

Tables and indirect calls 11.3

A WebAssembly function has an index but no address. Code cannot compute a function index at runtime and call it; `call` takes its index as an immediate, fixed in the code. To call a function chosen at runtime, you put function references in a *table* and call through it with `call_indirect`:

```
(module
  (type $unop (func (param i32) (result i32)))
  (table 3 funcref)
  (elem (i32.const 0) $double $square $negate)
  (func $double (param i32) (result i32) (i32.add (local.get 0)
      (local.get 0)))
  (func $square (param i32) (result i32) (i32.mul (local.get 0)
      (local.get 0)))
  (func $negate (param i32) (result i32) (i32.sub (i32.const 0)
      (local.get 0)))
  (func (export "apply") (param $which i32) (param $x i32) (result
      i32)
    (call_indirect (type $unop) (local.get $x) (local.get
        $which))))
```

called as `apply(0, 7)`, `apply(1, 7)` and `apply(2, 7)`

```
14
49
-7
```

The pieces:

- `(table 3 funcref)` declares a table of three elements, each a `funcref`, a reference to a function. Elements start out null.
- `(elem (i32.const 0) $double $square $negate)` is an *element segment*: it puts references to those three functions into the table, starting at index 0, when the module is instantiated. Element segments are to tables what data segments are to memories.
- `call_indirect (type $unop)` pops a table index (last, on top of the arguments), looks up the function there, checks that its type is `$unop`, and calls it with the arguments below.

The type check is what keeps indirect calls safe. It is done at runtime, on every call, and a mismatch traps:

```
(module
  (type $unop (func (param i32) (result i32)))
  (table 2 funcref)
  (elem (i32.const 0) $no_params)
  (func $no_params (result i32) (i32.const 1))
  (func (export "call") (param $i i32) (result i32)
    (call_indirect (type $unop) (i32.const 5) (local.get $i))))
```

called as `call(0)`

```
RuntimeError: null function or function signature mismatch
```

So does calling a null element, or an index past the end of the table:

```
(module
  (type $v (func))
  (table 2 funcref)
  (func (export "call") (param $i i32)
    (call_indirect (type $v) (local.get $i))))
```

called as `call(7)`

```
RuntimeError: table index is out of bounds
```

For compiler writers

A function pointer in your language becomes a table index: an `i32` you can store in memory, compare, and pass around like any other integer. The usual approach, and LLVM's, is one table holding every function whose address is ever taken, numbered at link time, with index 0 left null so that a null function pointer traps when called. Functions whose address is never taken need not be in the table at all.

Under the hood

The signature check is cheap: engines canonicalise every function type to a small integer, keep it beside each table entry, and compare it with the expected one before the call. A `call_indirect` costs a bounds check, a load, a compare and an indirect call, which is more than a direct call but far less than a call from WebAssembly to JavaScript.

Virtual calls 11.4

A vtable becomes a list of table indices in linear memory. An object holds a pointer to its class's vtable, and a virtual call loads the vtable pointer, loads the method's table index from it, and calls through the table:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (memory 1)
  (type $method (func (param i32) (result i32)))   ;; takes `this`
  (table 4 funcref)
  (elem (i32.const 1) $square_area $rect_area)

  ;; vtables: one slot each, holding a table index
  (data (i32.const 16) "\01\00\00\00")             ;; Square's
      vtable: area = 1
  (data (i32.const 20) "\02\00\00\00")             ;; Rect's
      vtable:   area = 2

  ;; objects: [vtable pointer, fields...]
  (data (i32.const 64) "\10\00\00\00" "\05\00\00\00")
      ;; Square{side: 5}
  (data (i32.const 80) "\14\00\00\00" "\03\00\00\00"
      "\04\00\00\00")    ;; Rect{w: 3, h: 4}

  (func $square_area (param $this i32) (result i32)
    (i32.mul (i32.load offset=4 (local.get $this)) (i32.load
        offset=4 (local.get $this))))
  (func $rect_area (param $this i32) (result i32)
    (i32.mul (i32.load offset=4 (local.get $this)) (i32.load
        offset=8 (local.get $this))))

  ;; shape->area()
  (func $area (param $shape i32) (result i32)
    (call_indirect (type $method)
      (local.get $shape)                                  ;; this
      (i32.load offset=0                                  ;;
          vtable[0]
        (i32.load (local.get $shape)))))                  ;;
            shape->vtable

  (func (export "main")
    (call $print (call $area (i32.const 64)))
    (call $print (call $area (i32.const 80)))))
```

```
25
12
```

Interfaces and trait objects work the same way, with the vtable pointer carried beside the data pointer as a pair rather than inside the object.

Closures 11.5

A closure is a function and its captured environment. In linear memory, it is a pair: a pointer to the environment, and a table index for the code. Calling it passes the environment as an extra first parameter:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (memory 1)
  (global $heap (mut i32) (i32.const 1024))
  (type $closure_fn (func (param i32 i32) (result i32)))   ;; env,
      x
  (table 2 funcref)
  (elem (i32.const 1) $adder_code)

  ;; the code of `|x| x + n`, with n in the environment
  (func $adder_code (param $env i32) (param $x i32) (result i32)
    (i32.add (local.get $x) (i32.load (local.get $env))))

  ;; make_adder(n) returns a closure: [env pointer, table index]
  (func $make_adder (param $n i32) (result i32)
    (local $env i32)
    (local $clo i32)
    (local.set $env (global.get $heap))
    (i32.store (local.get $env) (local.get $n))
    (local.set $clo (i32.add (local.get $env) (i32.const 4)))
    (i32.store offset=0 (local.get $clo) (local.get $env))
    (i32.store offset=4 (local.get $clo) (i32.const 1))
    (global.set $heap (i32.add (local.get $clo) (i32.const 8)))
    (local.get $clo))

  (func $call_closure (param $clo i32) (param $x i32) (result i32)
    (call_indirect (type $closure_fn)
      (i32.load offset=0 (local.get $clo))     ;; env
      (local.get $x)
      (i32.load offset=4 (local.get $clo))))   ;; code

  (func (export "main")
    (local $add10 i32)
    (local $add100 i32)
    (local.set $add10 (call $make_adder (i32.const 10)))
    (local.set $add100 (call $make_adder (i32.const 100)))
    (call $print (call $call_closure (local.get $add10) (i32.const
        1)))
    (call $print (call $call_closure (local.get $add100)
        (i32.const 1)))))
```

```
11
101
```

If every function value in your language is a closure, give plain functions the same calling convention with an unused environment parameter, so a single kind of call works for both.

Tables as runtime state 11.6

Tables are not fixed. The table instructions read and change them at runtime, much as memory instructions do for bytes:

Instruction	Type	Does
`table.get $t`	`[i32] → [ref]`	reads an element
`table.set $t`	`[i32 ref] → []`	writes an element
`table.size $t`	`[] → [i32]`	the number of elements
`table.grow $t`	`[ref i32] → [i32]`	adds elements, set to the given reference; returns the old size or −1
`table.fill $t`	`[i32 ref i32] → []`	sets a range of elements
`table.copy`, `table.init`, `elem.drop`		the bulk operations, as for memory
`ref.func $f`	`[] → [funcref]`	a reference to a function
`ref.null func`	`[] → [funcref]`	the null reference
`ref.is_null`	`[ref] → [i32]`	tests for null

These let a program add functions to its table as it goes, which is what dynamic linking needs: a newly loaded module's functions are added to the shared table, and their new indices are its function pointers.

A function referenced with `ref.func` must be *declared* as referenceable, by appearing in some element segment, so the engine knows ahead of time which functions might be called indirectly. A *declarative* segment exists only for this purpose:

```
(module
  (type $v (func (result i32)))
  (table 1 funcref)
  (elem declare func $seven)
  (func $seven (result i32) (i32.const 7))
  (func (export "main") (result i32)
    (table.set (i32.const 0) (ref.func $seven))
    (call_indirect (type $v) (i32.const 0))))
```

```
7
```

Host references 11.7

Tables can hold another kind of reference: `externref`, an opaque reference to anything the host likes, which in a browser means

any JavaScript value. WebAssembly code cannot look inside an `externref`; it can only store it, pass it around, and hand it back.

This is the clean way for a module to hold on to host objects, such as WebGL textures or DOM elements: keep them in a table, and use their table indices as handles in linear memory. Before `externref`, toolchains kept a JavaScript array beside every instance and passed integer handles across, which works but leaves the bookkeeping to hand-written glue.

```
(module
  (import "env" "describe" (func $describe (param externref)))
  (table $objects (export "objects") 8 externref)
  (func (export "show") (param $handle i32)
    (call $describe (table.get $objects (local.get $handle)))))
```

```
import fs from 'node:fs';

const env = { describe: (obj) => console.log(`a ${obj.kind} called
    ${obj.name}`) };
const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'), { env
    });
const { objects, show } = instance.exports;
objects.set(3, { kind: 'texture', name: 'grass' });
objects.set(4, { kind: 'sound', name: 'jump' });
show(4);
show(3);
```

```
a sound called jump
a texture called grass
```

The engine's garbage collector sees references in tables, so an object a table holds stays alive until the slot is overwritten or the table itself is collected.

What you have seen 11.8

Globals hold single typed values, initialised by constant expressions, and can be shared with the host. Tables hold references. With `call_indirect`, a table of `funcref`s turns an integer into a function call, checked against the expected signature at runtime, and that is how function pointers, vtables and closures are compiled. Tables can be changed at runtime, and can hold `externref`s, opaque host values, which are the tidy way to give a module handles to host objects.

That ends Part III. Part IV looks at how all of this is encoded in bytes.

PART IV

The Binary Format

Everything so far has been written in the text format and turned into bytes by an assembler. A compiler usually skips the text and writes the bytes itself, which is simpler than it sounds: the binary format was designed to be easy to produce and fast to decode. This part goes through it from the first byte to the last, writes a module in hex by hand, builds a small encoder, and covers the optional sections that carry names and debugging information.

Chapter 12

Bytes on the Wire

This chapter covers the parts of the binary format that are not instructions: how integers and strings are encoded, how a module is cut into sections, and how types, imports and exports are written. By the end of it you will have written a module byte by byte, and it will run.

The preamble 12.1

Every module starts with the same eight bytes:

```
00 61 73 6d   ;; magic: "\0asm"
01 00 00 00   ;; version 1
```

That is a complete module. The version has been 1 since the MVP: new features add new sections, instructions and encodings, but never change what an existing byte sequence means, so there has never been a need for version 2. (The component model, a separate layer on top of core WebAssembly, uses a different version number to tell its binaries apart.)

Anything else at the start is rejected:

```
00 61 73 6d   ;; magic
02 00 00 00   ;; version 2
```

```
CompileError: WebAssembly.Module(): expected version 01 00 00 00,
    found 02 00 00 00 @+4
```

LEB128 12.2

Most integers in the format are written in *LEB128*, a variable-length encoding that uses as few bytes as the value needs. Each byte holds seven bits of the value, least significant first, and its top bit says whether more bytes follow:

- 5 is `05`, one byte;
- 127 is `7f`, one byte;
- 128 is `80 01`: the low seven bits (0) with the continuation bit set, then the next seven bits (1);
- 624,485 is `e5 8e 26`.

The *signed* form, used for constants that can be negative, is the same, except that the last byte's seventh bit is a sign bit, sign-

extended to the full width when decoding. So −1 is `7f`, and 63 is `3f` but 64 is `c0 00`, because a single byte `40` would mean −64.

Here are both, in Python:

```
def uleb(n):
    out = bytearray()
    while True:
        byte = n & 0x7f
        n >>= 7
        if n:
            out.append(byte | 0x80)
        else:
            out.append(byte)
            return bytes(out)

def sleb(n):
    out = bytearray()
    while True:
        byte = n & 0x7f
        n >>= 7                         # an arithmetic shift: n
            keeps its sign
        done = (n == 0 and not byte & 0x40) or (n == -1 and byte &
            0x40)
        out.append(byte if done else byte | 0x80)
        if done:
            return bytes(out)
```

```
for n in (5, 127, 128, 624485):
    print(f"uleb({n}) = {uleb(n).hex(' ')}")
for n in (-1, 63, 64, -64, -65, -123456):
    print(f"sleb({n}) = {sleb(n).hex(' ')}")
```

```
uleb(5) = 05
uleb(127) = 7f
uleb(128) = 80 01
uleb(624485) = e5 8e 26
sleb(-1) = 7f
sleb(63) = 3f
sleb(64) = c0 00
sleb(-64) = 40
sleb(-65) = bf 7f
sleb(-123456) = c0 bb 78
```

A `u32` in LEB128 takes at most five bytes, and an `i64` at most ten. The specification also allows *padded* encodings, with redundant continuation bytes, up to that maximum: 5 can be written `85 80 80 80 00`. Decoders must accept these. Producers use them when they need to write a number before they know it, reserving five bytes and filling them in later without moving anything. LLVM's object files rely on this for relocations.

The format uses LEB128 for counts, sizes, indices and integer constants. Floating-point constants are not LEB128; they are written as raw IEEE 754 bytes, little-endian, four or eight of them.

Vectors and names 12.3

A *vector* is a count followed by that many items. Almost every list in the format, from the functions in a section to the parameters of a type, is a vector.

A *name* is a vector of bytes holding UTF-8. So the export name `main` is `04 6d 61 69 6e`: four bytes, then the letters.

Sections 12.4

After the preamble, a module is a sequence of sections. Each one is:

```
section id    1 byte
size          u32, LEB128: the number of bytes that follow
contents      that many bytes
```

The size lets a decoder skip any section it does not care about. A browser downloading a module can parse the sections it needs to start compiling and skip over the rest, and a tool can ignore sections it does not understand. Chapter 8 listed the sections and the fixed order they must appear in. Every known section may appear at most once, and an empty one can be left out.

Types 12.5

The type section is a vector of function types. A function type is the byte `60`, then a vector of parameter types, then a vector of result types. Each *value type* is a single byte:

Byte	Type
`7f`	`i32`
`7e`	`i64`
`7d`	`f32`
`7c`	`f64`
`7b`	`v128`
`70`	`funcref`
`6f`	`externref`
`69`	`exnref`

The garbage collection feature adds more reference types, some of which take more than one byte; Chapter 22 shows them. The bytes count down from `7f` because they were chosen to be negative numbers in signed LEB128 (`7f` is −1, `7e` is −2), leaving the

non-negative numbers free to be type indices in places where either can appear.

So the type `(func (param i32 i32) (result i32))` is:

```
60          func
02 7f 7f    two parameters, both i32
01 7f       one result, an i32
```

Imports, functions and exports 12.6

An *import* is a module name, a field name, and a description, which starts with a kind byte:

Kind	Import description
`00`	a function: its type index
`01`	a table: its element type and limits
`02`	a memory: its limits
`03`	a global: its value type and a mutability byte, `00` or `01`
`04`	a tag: `00`, then a type index

Limits, which size memories and tables, are a flag byte followed by LEB128 numbers: `00 n` for a minimum of `n` with no maximum, `01 n m` for a minimum and a maximum. Other flag bits mark a memory as shared (Chapter 23) or 64-bit.

The *function section* is a vector of type indices, one per function the module defines. The bodies come later, in the code section, in the same order.

An *export* is a name, a kind byte from the same table, and an index.

A module by hand 12.7

That is enough to write a module that exports a function returning 42. Section by section:

- **Type section** (id 1): one type, `() -> i32`.
- **Function section** (id 3): one function, of type 0.
- **Export section** (id 7): one export, `"main"`, function 0.
- **Code section** (id 10): one body. A body is its size, then a vector of local declarations (here empty), then the instructions, ending with `0b`, the `end` opcode. `i32.const 42` is opcode `41` followed by 42 in signed LEB128, `2a`.

Every section's size has to be counted by hand, which is the tedious part:

```
00 61 73 6d  01 00 00 00     ;; preamble

01                           ;; type section
05                           ;;   5 bytes follow
01                           ;;   one type
60 00 01 7f                  ;;   func, no params, one result: i32

03                           ;; function section
02                           ;;   2 bytes follow
01 00                        ;;   one function, of type 0

07                           ;; export section
08                           ;;   8 bytes follow
01                           ;;   one export
04 6d 61 69 6e               ;;   name: "main"
00 00                        ;;   a function, index 0

0a                           ;; code section
06                           ;;   6 bytes follow
01                           ;;   one body
04                           ;;   the body is 4 bytes
00                           ;;   no locals
41 2a                        ;;   i32.const 42
0b                           ;;   end
```

```
42
```

Thirty-seven bytes, the same as `wat2wasm` produced from the text in Chapter 2.

Getting a size wrong is the classic mistake of a hand-written or newly written encoder. Here is the same module with the code section's size one too small:

```
00 61 73 6d  01 00 00 00
01 05 01 60 00 01 7f
03 02 01 00
07 08 01 04 6d 61 69 6e 00 00
0a 05 01 04 00 41 2a 0b      ;; the size should be 06
```

```
CompileError: WebAssembly.Module(): expected 4 bytes, fell off
    end @+33
```

V8 read the body's own size, 4, and found the section, and so the module, ended after only three of those bytes. Wasmtime says `unexpected end-of-file (at offset 0x20)` about the same module. Neither message names the real mistake, which is typical of size errors: the decoder notices only when the bytes stop making sense, some way after the wrong number.

And here it is with the sections out of order:

```
00 61 73 6d  01 00 00 00
01 05 01 60 00 01 7f
03 02 01 00
0a 06 01 04 00 41 2a 0b
07 08 01 04 6d 61 69 6e 00 00  ;; the export section must come
    before code
```

```
CompileError: WebAssembly.Module(): unexpected section <Export>
    @+29
```

The `@+29` is the offset of the offending section's id byte. Wasmtime reports the same module as `section out of order (at offset 0x1d)`.

Letting a tool count for you 12.8

`wat2wasm -v` prints the binary it writes, annotated byte by byte, on stderr. When an encoder you are writing produces something you do not understand, writing the same module in the text format and comparing your bytes against this dump is the quickest way to find the difference:

```
(module
  (func (export "add") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add))
```

```
wat2wasm -v add.wat -o add.wasm 2>&1
```

```
0000000: 0061 736d                                ;
    WASM_BINARY_MAGIC
0000004: 0100 0000                                ;
    WASM_BINARY_VERSION
; section "Type" (1)
0000008: 01                                       ; section code
0000009: 00                                       ; section size
    (guess)
000000a: 01                                       ; num types
; func type 0
000000b: 60                                       ; func
000000c: 02                                       ; num params
000000d: 7f                                       ; i32
000000e: 7f                                       ; i32
000000f: 01                                       ; num results
0000010: 7f                                       ; i32
0000009: 07                                       ; FIXUP section
    size
; section "Function" (3)
0000011: 03                                       ; section code
0000012: 00                                       ; section size
    (guess)
0000013: 01                                       ; num functions
0000014: 00                                       ; function 0
    signature index
0000012: 02                                       ; FIXUP section
    size
; section "Export" (7)
0000015: 07                                       ; section code
0000016: 00                                       ; section size
    (guess)
0000017: 01                                       ; num exports
0000018: 03                                       ; string length
0000019: 6164 64                                  add  ; export
    name
000001c: 00                                       ; export kind
000001d: 00                                       ; export func
    index
0000016: 07                                       ; FIXUP section
    size
; section "Code" (10)
000001e: 0a                                       ; section code
000001f: 00                                       ; section size
    (guess)
0000020: 01                                       ; num functions
; function body 0
0000021: 00                                       ; func body
    size (guess)
0000022: 00                                       ; local decl
    count
0000023: 20                                       ; local.get
0000024: 00                                       ; local index
0000025: 20                                       ; local.get
0000026: 01                                       ; local index
0000027: 6a                                       ; i32.add
0000028: 0b                                       ; end
0000021: 07                                       ; FIXUP func
    body size
000001f: 09                                       ; FIXUP section
    size
```

The `FIXUP` lines show how `wat2wasm` handles sizes: it writes a placeholder, writes the contents, then goes back and fills in the size. The placeholder is one byte here because every size turned

out to fit in one; when one does not, wabt moves the contents along to make room. The next chapter shows a simpler approach.

`wasm-objdump -s` prints the raw contents of each section, and `-x` the decoded details. Between them and `-v`, you can see a module at every level.

What you have seen 12.9

A module is the preamble followed by sections, each an id, a size and contents. Integers are LEB128, signed where they can be negative, and floats are raw IEEE bytes. Lists are vectors, names are UTF-8 vectors, value types are single bytes counting down from `7f`. Types, imports, functions and exports are each a few bytes, and a complete module that returns 42 is thirty-seven of them.

The next chapter encodes the code section's contents: instructions.

Chapter 13

Encoding Instructions

The last chapter encoded everything but code. This one encodes function bodies: local declarations, opcodes, and the immediates that follow them. Then it puts everything together into a small encoder, about a hundred lines of Python, that writes modules an engine will run. The encoder is not meant to be used as it stands, but it shows every decision a real one makes, and it is a template for the backend of your own compiler.

Function bodies 13.1

The code section is a vector of bodies, one per function in the function section, in the same order. Each body is:

```
size        u32: the number of bytes in the rest of the body
locals      a vector of (count, value type) pairs
code        instructions, ending with 0b (end)
```

The local declarations are run-length encoded. A function with locals `i32 i32 i32 f64 i32` declares them as three groups: `(3, i32) (1, f64) (1, i32)`, which is `03 03 7f 01 7c 01 7f`. Parameters are not declared here, since they come from the function's type; the declared locals are numbered after them.

Opcodes and immediates 13.2

Each instruction is an opcode byte, followed by its *immediates*, the operands that are part of the instruction rather than taken from the stack. Most instructions have none:

```
(module
  (func (export "f") (param i32 i32) (result i32)
    (local i64 i64 f32)
    (i32.mul (i32.add (local.get 0) (local.get 1)) (i32.const
        -2))))
```

```
wasm-objdump -d arith.wasm | tail -n +5
```

```
000020 func[0] <f>:
 000021: 02 7e                      | local[2..3] type=i64
 000023: 01 7d                      | local[4] type=f32
 000025: 20 00                      | local.get 0
 000027: 20 01                      | local.get 1
 000029: 6a                         | i32.add
 00002a: 41 7e                      | i32.const 4294967294
 00002c: 6c                         | i32.mul
 00002d: 0b                         | end
```

`local.get` (`20`) takes a local index. `i32.const` (`41`) takes its value in *signed* LEB128, so −2 is the single byte `7e`; `wasm-objdump` prints it as unsigned. `i64.const` (`42`) is the same with up to ten bytes. `f32.const` (`43`) and `f64.const` (`44`) take four or eight raw bytes.

Here are the immediates of the instructions you have met so far:

Instruction	Opcode	Immediates
`local.get`, `local.set`, `local.tee`	`20`, `21`, `22`	local index
`global.get`, `global.set`	`23`, `24`	global index
`call`	`10`	function index
`call_indirect`	`11`	type index, then table index
`br`, `br_if`	`0c`, `0d`	label depth
`br_table`	`0e`	a vector of depths, then the default depth
`block`, `loop`, `if`	`02`, `03`, `04`	a block type
loads and stores	`28` – `3e`	a *memarg*: alignment, then offset
`memory.size`, `memory.grow`	`3f`, `40`	memory index (`00`)
`i32.const`, `i64.const`	`41`, `42`	signed LEB128 value
`f32.const`, `f64.const`	`43`, `44`	IEEE 754 bytes, little-endian
`select` with a type	`1c`	a vector of value types

Appendix A lists every opcode.

Block types 13.3

`block`, `loop` and `if` are followed by a *block type*, saying what the construct takes and leaves on the stack. It has three forms, chosen by the first byte:

- `40`: no parameters and no results;
- a value type such as `7f`: no parameters and that one result;
- a type index, as a signed LEB128 number (33 bits wide, so all valid indices are non-negative): the parameters and results of that function type. This form is needed for blocks with parameters or with more than one result.

The three can be told apart because value types are bytes in the range of small negative numbers, and `40` is −64, which no value

type uses. The construct ends with `0b`; an `if` with an `else` arm has `05` between the arms.

```
(module
  (type $pair (func (param i32) (result i32 i32)))
  (func (export "f") (param i32) (result i32)
    (block $outer (result i32)
      (loop $l
        (br_if $l (i32.eqz (local.get 0))))
      (local.get 0)
      (block (type $pair)
        (i32.const 1))
      (i32.add)
      (if (param i32) (result i32) (local.get 0)
        (then)
        (else (drop) (i32.const 0)))
      (br 0))))
```

```
wasm-objdump -d blocks.wasm | tail -n +6
```

```
000025 func[0] <f>:
 000026: 02 7f                      | block i32
 000028: 03 40                      |   loop
 00002a: 20 00                      |     local.get 0
 00002c: 45                         |     i32.eqz
 00002d: 0d 00                      |     br_if 0
 00002f: 0b                         |   end
 000030: 20 00                      |   local.get 0
 000032: 02 00                      |   block type[0]
 000034: 41 01                      |     i32.const 1
 000036: 0b                         |   end
 000037: 6a                         |   i32.add
 000038: 20 00                      |   local.get 0
 00003a: 04 01                      |   if type[1]
 00003c: 05                         |   else
 00003d: 1a                         |     drop
 00003e: 41 00                      |     i32.const 0
 000040: 0b                         |   end
 000041: 0c 00                      |   br 0
 000043: 0b                         | end
 000044: 0b                         | end
```

The `if` with a parameter needed a function type `[i32] → [i32]`, which the assembler added to the type section as type 1. An encoder that supports blocks with parameters has to do the same: intern a function type for each such block signature.

Memory immediates 13.4

A load or store is followed by a *memarg*: the alignment, as a power of two (so `align=4` is written `02`), then the offset, both as unsigned LEB128:

```
(module
  (memory 1)
  (func (export "f") (param i32) (result i32)
    (i32.store8 offset=3 (local.get 0) (i32.const 7))
    (i32.load offset=1024 align=1 (local.get 0))))
```

```
wasm-objdump -d mem.wasm | tail -n +6
```

```
000024 func[0] <f>:
 000025: 20 00                      | local.get 0
 000027: 41 07                      | i32.const 7
 000029: 3a 00 03                   | i32.store8 0 3
 00002c: 20 00                      | local.get 0
 00002e: 28 00 80 08                | i32.load 0 1024
 000032: 0b                         | end
```

`3a 00 03` is `i32.store8` with alignment 2^0 and offset 3. `28 00 80 08` is `i32.load` with alignment 2^0 (the `align=1` the text asked for) and offset 1024 in LEB128. With multiple memories, a memarg whose alignment field has bit 6 set (64 added to it) is followed by a memory index.

Prefixed opcodes 13.5

One byte allows only 256 opcodes, and they ran out. Newer instructions use a *prefix* byte followed by a second opcode in unsigned LEB128:

Prefix	Holds
`fb`	garbage collection: `struct.new`, `array.get`, `ref.cast`, …
`fc`	saturating truncation (`fc 00` – `fc 07`), bulk memory and table operations
`fd`	SIMD: every `v128` instruction
`fe`	threads: atomic loads, stores, read-modify-write, wait and notify

```
(module
  (memory 1)
  (func (export "f") (param f64) (result i32)
    (memory.fill (i32.const 0) (i32.const 0) (i32.const 16))
    (i32.trunc_sat_f64_s (local.get 0))))
```

```
wasm-objdump -d prefixed.wasm | tail -n +6
```

```
000024 func[0] <f>:
 000025: 41 00                      | i32.const 0
 000027: 41 00                      | i32.const 0
 000029: 41 10                      | i32.const 16
 00002b: fc 0b 00                   | memory.fill 0
 00002e: 20 00                      | local.get 0
 000030: fc 02                      | i32.trunc_sat_f64_s
 000032: 0b                         | end
```

An encoder 13.6

That is everything needed to write an encoder. A real one has three parts: a *byte buffer* with methods for each encoding (LEB128,

names, vectors), a *module builder* that collects types, imports, functions and so on and assigns their indices, and a *function builder* that turns instructions into bytes. Here they are, small enough to read in one go.

First, the encodings. `leb` and `sleb` are from the last chapter:

```
import struct

def uleb(n):
    out = bytearray()
    while True:
        byte, n = n & 0x7f, n >> 7
        out.append(byte | (0x80 if n else 0))
        if not n:
            return bytes(out)

def sleb(n):
    out = bytearray()
    while True:
        byte, n = n & 0x7f, n >> 7
        done = (n == 0 and not byte & 0x40) or (n == -1 and byte &
            0x40)
        out.append(byte if done else byte | 0x80)
        if done:
            return bytes(out)

def name(s):
    b = s.encode("utf-8")
    return uleb(len(b)) + b

def vec(items):
    return uleb(len(items)) + b"".join(items)

def section(id, contents):
    return bytes([id]) + uleb(len(contents)) + contents

I32, I64, F32, F64 = 0x7f, 0x7e, 0x7d, 0x7c
```

`section` sidesteps the size problem the last chapter mentioned: it builds the contents first, as a separate `bytes` object, and only then writes the size in front. The cost is a copy of each section, which is negligible. Encoders that care about every copy, such as one generating code on the fly in a browser, reserve five bytes for each size and fill them in with a padded LEB128 afterwards instead.

Next, a function builder. It has a method per instruction that the examples need, and every method returns `self`, so calls can be chained. A real encoder would generate these methods from a table:

```
class Func:
    def __init__(self, params, results):
        self.params, self.results = params, results
        self.locals = []
        self.code = bytearray()

    def local(self, ty):
        """Declare a local, and return its index."""
        self.locals.append(ty)
        return len(self.params) + len(self.locals) - 1

    def op(self, opcode, *immediates):
        self.code.append(opcode)
        for imm in immediates:
            self.code += imm
        return self

    # a few instructions
    def local_get(self, i): return self.op(0x20, uleb(i))
    def local_set(self, i): return self.op(0x21, uleb(i))
    def local_tee(self, i): return self.op(0x22, uleb(i))
    def call(self, f):      return self.op(0x10, uleb(f))
    def i64_const(self, n): return self.op(0x42, sleb(n))
    def i32_const(self, n): return self.op(0x41, sleb(n))
    def f64_const(self, x): return self.op(0x44, struct.pack("<d",
        x))
    def block(self):        return self.op(0x02, b"\x40")
    def loop(self):         return self.op(0x03, b"\x40")
    def end(self):          return self.op(0x0b)
    def br(self, depth):    return self.op(0x0c, uleb(depth))
    def br_if(self, depth): return self.op(0x0d, uleb(depth))
    def i64_eqz(self):      return self.op(0x50)
    def i64_mul(self):      return self.op(0x7e)
    def i64_sub(self):      return self.op(0x7d)

    def body(self):
        # run-length encode the locals: consecutive locals of one
            type form a group
        groups = []
        for ty in self.locals:
            if groups and groups[-1][1] == ty:
                groups[-1][0] += 1
            else:
                groups.append([1, ty])
        decls = vec([uleb(n) + bytes([ty]) for n, ty in groups])
        contents = decls + bytes(self.code) + b"\x0b"      # the
            function's own end
        return uleb(len(contents)) + contents
```

Finally, the module builder. It interns function types, so each signature is in the type section once, and numbers functions with imports first:

```
class Module:
    def __init__(self):
        self.types, self.imports, self.funcs, self.exports = [],
            [], [], []

    def type_index(self, params, results):
        t = (tuple(params), tuple(results))
        if t not in self.types:
            self.types.append(t)
        return self.types.index(t)

    def import_func(self, module, field, params, results):
        assert not self.funcs, "imports must be added before
            functions"
        self.imports.append((module, field,
            self.type_index(params, results)))
        return len(self.imports) - 1

    def func(self, params, results, export=None):
        f = Func(params, results)
        self.funcs.append(f)
        index = len(self.imports) + len(self.funcs) - 1
        if export:
            self.exports.append((export, index))
        return f, index

    def encode(self):
        for f in self.funcs:
            self.type_index(f.params, f.results)
        types = vec([b"\x60" + vec([bytes([p]) for p in ps]) +
            vec([bytes([r]) for r in rs])
                     for ps, rs in self.types])
        imports = vec([name(m) + name(f) + b"\x00" + uleb(t) for
            m, f, t in self.imports])
        funcs = vec([uleb(self.type_index(f.params, f.results))
            for f in self.funcs])
        exports = vec([name(n) + b"\x00" + uleb(i) for n, i in
            self.exports])
        code = vec([f.body() for f in self.funcs])
        out = b"\x00asm\x01\x00\x00\x00" + section(1, types)
        if self.imports:
            out += section(2, imports)
        return out + section(3, funcs) + section(7, exports) +
            section(10, code)
```

Now a program that uses it: an iterative factorial on `i64`s, and a `main` that prints two results through the book's host:

```
m = Module()
print_i64 = m.import_func("env", "print_i64", [I64], [])

# fac(n): acc = 1; while (n != 0) { acc *= n; n -= 1 } return acc
fac, fac_index = m.func([I64], [I64], export="fac")
n = 0
acc = fac.local(I64)
(fac.i64_const(1).local_set(acc)
    .block().loop()
        .local_get(n).i64_eqz().br_if(1)
        .local_get(acc).local_get(n).i64_mul().local_set(acc)
        .local_get(n).i64_const(1).i64_sub().local_set(n)
        .br(0)
    .end().end()
    .local_get(acc))

main, _ = m.func([], [], export="main")
main.i64_const(10).call(fac_index).call(print_i64)
main.i64_const(20).call(fac_index).call(print_i64)

wasm = m.encode()
open("out.wasm", "wb").write(wasm)
print(len(wasm), "bytes")
```

```
119 bytes
3628800
2432902008176640000
```

The book's harness validated `out.wasm` with V8 and ran it, which is the same thing the encoder's own tests should do. Part VIII builds on exactly that.

Things a real encoder adds 13.7

The toy above leaves out several things a production encoder needs:

- **Every instruction**, ideally generated from a table such as the one in Appendix A, so that no opcode is mistyped by hand.
- **Label depths from names.** Asking the code generator to count depths invites mistakes. Better to have `block()` and `loop()` return a label object, keep a stack of open constructs, and compute the depth when a branch to a label object is emitted.
- **Validation as it goes.** Tracking the operand stack's types while emitting, as the validator would, catches code generator bugs at the instruction that caused them, rather than as an opaque error from the engine later. Binaryen and the `wasm-encoder` crate leave this to a separate validator; some JIT encoders build it in.
- **The other sections**, and their order: memory, tables, globals, data and element segments, the data count section (needed

whenever the code uses `memory.init` or `data.drop`), and custom sections.

You may not want to write an encoder at all. Libraries exist for most languages a compiler is written in. In Rust, `wasm-encoder` (part of `wasm-tools`) is a thin, complete, fast encoder much like the one above. Binaryen, usable from C, C++, JavaScript and through bindings elsewhere, is a whole compiler infrastructure with its own IR, optimiser and validator. Its IR is an expression tree, so a code generator can hand it nested expressions and let it worry about the stack. Writing your own gives you full control and no dependencies; using Binaryen gives you its optimiser, which Chapter 25 shows is worth having.

What you have seen 13.8

A function body is its size, run-length-encoded local declarations, and instructions ending with `0b`. Each instruction is an opcode, or a prefix and an opcode, followed by immediates: indices and constants in LEB128, float constants as raw bytes, block types in one of three forms, and memargs giving alignment and offset. An encoder is a byte buffer, a module builder that assigns indices and interns types, and a function builder that appends instructions, and the result can be tested by running it.

The next chapter covers the bytes that are not code at all: custom sections, names and debugging information.

Chapter 14

Custom Sections and Debugging

A module can carry sections that engines ignore: *custom sections*, with id 0. They hold anything a toolchain wants to attach to a module, and in practice that means names for functions and locals, information about the tools that produced the module, and debugging information. This chapter covers the format, the name section that every compiler should emit, and how debugging works in a browser.

The format 14.1

A custom section is a section with id 0 whose contents start with a name, the section's own name, followed by whatever bytes its producer likes:

```
00            section id: custom
size          u32
name          a name: the custom section's name
payload       the rest of the section
```

Custom sections can appear anywhere, any number of times, even between known sections. An engine must accept a module whose custom sections it does not understand, so they never affect whether a module validates or how it runs, with one narrow exception: an engine may report a malformed *name* section as a warning.

Here is a module with a custom section added by hand, using the encoder helpers from the last chapter:

```
def uleb(n):
    out = bytearray()
    while True:
        byte, n = n & 0x7f, n >> 7
        out.append(byte | (0x80 if n else 0))
        if not n:
            return bytes(out)

def name(s):
    b = s.encode("utf-8")
    return uleb(len(b)) + b

def section(id, contents):
    return bytes([id]) + uleb(len(contents)) + contents
```

```
import subprocess
wasm = b"\x00asm\x01\x00\x00\x00" + section(0, name("build-info")
    + b"engine-v3 2026-11-01")
open("custom.wasm", "wb").write(wasm)
print(subprocess.run(["wasm-objdump", "-x", "custom.wasm"],
    capture_output=True, text=True).stdout)
```

```
custom.wasm:    file format wasm 0x1

Section Details:

Custom:
 - name: "build-info"
```

JavaScript can read custom sections from a compiled module, which makes them a convenient place to put metadata a game engine wants to read before instantiating a module, such as a version, or a list of assets a script needs:

```
wasm = b"\x00asm\x01\x00\x00\x00" + section(0, name("build-info")
    + b"engine-v3 2026-11-01")
open("custom.wasm", "wb").write(wasm)
```

```
import fs from 'node:fs';

const module = new
    WebAssembly.Module(fs.readFileSync('custom.wasm'));
for (const buf of WebAssembly.Module.customSections(module,
    'build-info'))
  console.log(new TextDecoder().decode(buf));
```

```
$PYTHON make.py
node read.mjs
```

```
engine-v3 2026-11-01
```

The name section 14.2

The custom section named `name` maps indices to names, for tools to show. Without it, a stack trace or a profile shows functions as numbers; with it, they have the names your compiler chose. It costs nothing at runtime, and it is by far the most useful custom section, so a compiler should emit it at least in development builds.

The payload is a sequence of subsections, each a one-byte id, a size, and contents. The standard ones are:

Id	Subsection	Contents
0	module	the module's name
1	function	a *name map*: a vector of (function index, name) pairs
2	local	for each function, a name map of its locals

Later additions (from the *extended name section* proposal) give names to labels (3), types (4), tables (5), memories (6), globals (7), element segments (8), data segments (9), struct fields (10) and tags (11). Tools support these unevenly, but they cost nothing to emit.

A name map must be sorted by index, and each subsection may appear at most once, in order of id.

The `wat` crate's assembler writes a name section with every `$name` in the text, which is why `wasm-objdump` has been able to show names throughout this book:

```
(module $physics
  (func $integrate (param $pos f32) (param $vel f32) (param $dt
      f32) (result f32)
    (f32.add (local.get $pos) (f32.mul (local.get $vel) (local.get
      $dt))))
  (func $step (export "step") (result f32)
    (call $integrate (f32.const 0) (f32.const 2) (f32.const
      0.5))))
```

```
wasm-objdump -x -j name named.wasm | tail -n +5
```

```
Section Details:

Custom:
 - name: "name"
 - module <physics>
 - func[0] <integrate>
 - func[1] <step>
 - func[0] local[0] <pos>
 - func[0] local[1] <vel>
 - func[0] local[2] <dt>
```

Names in stack traces 14.3

When a trap or an exception passes through WebAssembly code, the JavaScript `Error` it produces has a stack trace with a line for each WebAssembly frame. Here is a trap three calls deep, in a module with names:

```
(module
  (func $inner (unreachable))
  (func $middle (call $inner))
  (func $outer (export "main") (call $middle)))
```

```
import fs from 'node:fs';

const file = process.argv[2];
const { instance } = await
    WebAssembly.instantiate(fs.readFileSync(file));
try {
  instance.exports.main();
} catch (e) {
  const frames = e.stack.split('\n').filter((l) =>
      l.includes('wasm://'));
  // the URL's hash depends on the module's bytes, so it is left
      out here
  for (const f of frames)
      console.log(f.trim().replace(/wasm:\/\/wasm\/[0-9a-f]+/,
      'wasm://wasm/…'));
}
```

```
node trace.mjs deep.wasm
```

```
at inner (wasm://wasm/…:wasm-function[0]:0x23)
at middle (wasm://wasm/…:wasm-function[1]:0x27)
at outer (wasm://wasm/…:wasm-function[2]:0x2c)
```

Each frame shows the function's name, its index, and the byte offset of the instruction that was executing, from the start of the module. And here is the same module without its name section, removed by `wasm-strip`, which strips every custom section:

```
wasm-strip deep.wasm
node trace.mjs deep.wasm
```

```
at wasm://wasm/…:wasm-function[0]:0x23
at wasm://wasm/…:wasm-function[1]:0x27
at wasm://wasm/…:wasm-function[2]:0x2c
```

Without names, each frame says only which function index it was in. You can look those up yourself in a build that kept the names, provided the function indices are the same in both, which they are if stripping is the only difference. That is the usual approach for release builds: ship a stripped module, keep the unstripped one, and translate stack traces from the field after the fact.

The offsets are the same in both, because an offset counts from the start of the module, not the function, and the name section comes after the code. Toolchains put it last for exactly this reason: adding or removing names does not move any code.

Producers and features 14.4

Two other custom sections are common in the wild:

- `producers` records the languages, tools and SDKs that produced the module, as name and version pairs. Browsers use it for telemetry about which toolchains are in use; it has no effect on execution.
- `target_features` is written by LLVM into object files, to record which features (SIMD, bulk memory, and so on) the code uses, so the linker can check that everything linked together agrees.

Source maps and DWARF 14.5

A name section tells a debugger which function it is in. To show the *source line*, it needs a mapping from byte offsets in the code to positions in the source. There are two standard ways to provide one.

Source maps are the format JavaScript minifiers use: a JSON file mapping generated positions to source positions. For WebAssembly, the "generated position" is a byte offset in the code section. A module points at its map with a custom section named `sourceMappingURL`, whose payload is the map's URL as a name. Chrome, Firefox and Safari all support this, and it is the simplest option for a new compiler: you emit one JSON file alongside the module, and every browser's debugger can then step through your source, line by line. Source maps cannot describe variables, types or scopes, only positions.

DWARF is the debugging format native compilers use, and LLVM writes it into WebAssembly as custom sections named `.debug_info`, `.debug_line` and so on, exactly as it would in an ELF file, with code addresses replaced by code-section offsets. It describes everything: lines, variables, types, scopes and inlining. Chrome supports it through a DevTools extension for C and C++, and can show local variables and evaluate expressions in source terms. DWARF is far more work to produce than a source map, and makes modules much bigger, so it usually lives in a separate file named by a custom section called `external_debug_info`.

> **For compiler writers**
>
> A reasonable progression is: always emit a name section in development builds; emit a source map once your compiler tracks source positions through to code generation; consider DWARF only if your users need to inspect variables in the browser debugger and the source map is not enough. Keep the offset of every instruction you emit associated with the source position that produced it, from the start, since retrofitting that is painful.

Debugging in the browser 14.6

In Chrome or Firefox, a module appears in the debugger's source list under `wasm://`. Without a source map, the debugger shows the disassembled text format, and lets you set breakpoints on instructions, step, and inspect locals, globals, the operand stack and linear memory. With names, everything has the names your compiler gave it. That alone is enough to debug a compiler's output: put a breakpoint where something goes wrong, and compare what the machine is doing with what your compiler meant it to do.

Two more facilities are worth knowing about:

- **Memory inspection.** Chrome's *Memory Inspector* panel shows a linear memory as bytes and lets you jump to an address, which is invaluable for checking struct layouts and the shadow stack.
- **Logging imports.** The cheapest debugging tool remains an imported `print` function. A compiler flag that inserts calls to a trace import at each function entry, with the function index as an argument, gives you a call log for free.

What you have seen 14.7

Custom sections, id 0, carry a name and arbitrary bytes, and engines ignore them. The name section maps function, local and other indices to names, and makes stack traces, profiles and the debugger readable. Source maps, pointed to by a `sourceMappingURL` section, give browsers line-level debugging cheaply; DWARF gives full source-level debugging at greater cost. Stripping custom sections for release, and keeping an unstripped copy, is the usual practice.

That ends Part IV. Part V turns to the JavaScript side of the boundary.

PART V

The Host

A WebAssembly module on its own can compute, but it cannot see or do anything: every effect goes through its host. In a browser, the host is the JavaScript engine. This part covers the JavaScript API for compiling and instantiating modules, the rules for converting values at the boundary, the patterns for moving strings, arrays and callbacks across it efficiently, and what happens when a module traps.

Chapter 15

The JavaScript API

Everything a browser or Node offers for WebAssembly hangs off one global object, `WebAssembly`. This chapter goes through it: compiling and instantiating, looking inside a compiled module, the objects that represent memories, tables and globals, and the exact rules for converting values between JavaScript and WebAssembly.

Compiling and instantiating 15.1

There are two steps, which you can take separately or together:

Function	Takes	Gives
`WebAssembly.validate(bytes)`	bytes	`true` or `false`
`WebAssembly.compile(bytes)`	bytes	a promise of a `Module`
`WebAssembly.compileStreaming(response)`	a `fetch` response	a promise of a `Module`
`WebAssembly.instantiate(bytes, imports)`	bytes	a promise of `{ module, instance }`
`WebAssembly.instantiate(module, imports)`	a `Module`	a promise of an `Instance`
`WebAssembly.instantiateStreaming(response, imports)`	a `fetch` response	a promise of `{ module, instance }`
`new WebAssembly.Module(bytes)`	bytes	a `Module`, synchronously
`new WebAssembly.Instance(module, imports)`	a `Module`	an `Instance`, synchronously

Note the two shapes of `instantiate`: given bytes, it resolves to an object holding both the module and the instance; given a module, it resolves to just the instance. This trips everybody up once.

In a browser, use the streaming functions. They compile the module while it is still downloading, so for a large module, compilation is mostly finished by the time the last byte arrives. They need the server to send the file as `application/wasm`. The synchronous constructors block the thread while compiling, and Chrome refuses to compile large modules synchronously on the

main thread, so keep them for small modules, for workers, and for Node scripts like the ones in this book.

Under the hood

V8 compiles a module twice. As the bytes arrive, its baseline compiler, Liftoff, translates each function quickly into unoptimised machine code, so the module can start running at once. Then, as functions turn out to be hot, its optimising compiler, TurboFan, recompiles them in the background. Firefox's SpiderMonkey does the same with its baseline and Ion compilers. Chapter 25 covers what this means for performance. Browsers also cache the compiled code of modules loaded by `instantiateStreaming` from the same URL, so a second visit to a page skips most compilation.

A compiled `Module` can be sent to a worker with `postMessage`, and the worker can instantiate it without compiling it again. That is how a game engine running scripts on worker threads avoids paying for compilation on every thread.

Looking inside a module 15.2

A compiled module can be asked what it imports and exports:

```
(module
  (import "env" "log" (func (param i32)))
  (import "env" "memory" (memory 1))
  (global (export "version") i32 (i32.const 3))
  (table (export "callbacks") 4 funcref)
  (func (export "update") (param f32)))
```

```
import fs from 'node:fs';

const module = new
    WebAssembly.Module(fs.readFileSync('main.wasm'));
console.log(WebAssembly.Module.imports(module));
console.log(WebAssembly.Module.exports(module));
```

```
[
  { module: 'env', name: 'log', kind: 'function' },
  { module: 'env', name: 'memory', kind: 'memory' }
]
[
  { name: 'version', kind: 'global' },
  { name: 'callbacks', kind: 'table' },
  { name: 'update', kind: 'function' }
]
```

A host can use this to build the imports object a module needs, or to check before instantiating that a module only asks for imports the engine is willing to give it. That is useful when the modules come from somewhere you do not fully trust, such as user-written mods for a game. The *type reflection* proposal adds the

full types of imports and exports to this listing, but it is not yet available everywhere.

Exported functions 15.3

Each exported function is a JavaScript function. Its `length` is its number of parameters, and its `name` is its function index, as a string. Calling it converts the arguments to WebAssembly values, runs the code, and converts the results back, by the rules in section 15.5.

The same WebAssembly function, exported twice, or exported by one instance and imported and re-exported by another, is the *same* JavaScript function object. So functions can be compared with `===`, and used as keys in a `Map`.

Imported functions 15.4

A JavaScript function imported into a module is called with arguments converted from WebAssembly values, and its return value is converted to the import's declared result type. The conversions are forgiving in the way JavaScript usually is:

```
(module
  (import "env" "get" (func $get (param i32) (result i32)))
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "main")
    (call $print (call $get (i32.const 0)))
    (call $print (call $get (i32.const 1)))
    (call $print (call $get (i32.const 2)))
    (call $print (call $get (i32.const 3)))
    (call $print (call $get (i32.const 4)))))
```

```
import fs from 'node:fs';

const values = [3.7, '12', 2 ** 32 + 5, undefined, -1.5];
const env = { get: (i) => values[i], print_i32: (x) =>
    console.log(x) };
const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'), { env
    });
instance.exports.main();
```

```
3
12
5
0
-1
```

The string `'12'` became 12, `undefined` became 0, $2^{32} + 5$ wrapped to 5, and the fractions were truncated towards zero. This is

JavaScript's `ToInt32`, the same conversion as `x | 0`. It never fails, which means a host function that accidentally returns the wrong thing will not be caught at the boundary.

If a JavaScript import throws, the exception passes straight through the WebAssembly frames between it and the nearest JavaScript caller, which can catch it. The WebAssembly code in between simply stops. Chapter 20 shows how WebAssembly code can catch it instead.

Converting values 15.5

Here are the rules for each type, in both directions:

Type	WebAssembly → JavaScript	JavaScript → WebAssembly
`i32`	a Number, signed	`ToInt32`: truncate, then wrap modulo 2^{32}
`i64`	a BigInt, signed	`ToBigInt64`: must be a BigInt (or a string or boolean); a Number is a `TypeError`
`f32`	a Number, exactly	`ToNumber`, then round to single precision
`f64`	a Number, exactly	`ToNumber`
`v128`	`TypeError`	`TypeError`
`externref`	the original value, or `null`	any value
`funcref`	a function, or `null`	an exported WebAssembly function, or `null`
GC references	an opaque object, or `null`	(see Chapter 22)

Two rows deserve emphasis. An `i64` crosses as a BigInt, which is exact but slower than a Number, so an API used many times a frame should prefer `i32` and `f64` parameters. And a `v128` cannot cross at all: a function with a `v128` parameter or result can be called only from WebAssembly.

NaNs may lose their payload bits when they cross into JavaScript, which represents every NaN the same way. A module that cares about NaN payloads, such as a language that boxes values in NaNs, must not let them cross as floats; it can reinterpret them as integers first.

Memory, Table and Global objects 15.6

The other kinds of import and export are objects with constructors of their own, so a host can create them before any module exists:

```
const memory = new WebAssembly.Memory({ initial: 16, maximum: 1024
    });
const table = new WebAssembly.Table({ initial: 8, element:
    'anyfunc' });
const frame = new WebAssembly.Global({ value: 'i32', mutable: true
    }, 0);
```

A `Memory` has a `buffer`, which is replaced when the memory grows (Chapter 9), and a `grow(pages)` method that grows it from JavaScript. A `Table` has `get`, `set`, `grow` and `length`. A `Global` has `value`.

Creating these in JavaScript and importing them into a module, rather than having the module define and export them, means the host controls their sizes, and can share them between several modules or instances. Memory shared with workers must be created this way, with `shared: true` (Chapter 23).

Errors 15.7

The API reports three kinds of failure, each its own class:

Error	Thrown when
`WebAssembly.CompileError`	the bytes are malformed or do not validate
`WebAssembly.LinkError`	an import is missing or has the wrong type
`WebAssembly.RuntimeError`	the code traps (Chapter 17)

Stack overflow is a `RangeError`, and out-of-memory during instantiation is a `RangeError` as well. Exceptions thrown by imported functions are whatever was thrown. The `WebAssembly.Exception` class, for exceptions thrown by WebAssembly code, is the subject of Chapter 20.

A minimal loader 15.8

Putting it together, here is the shape of a loader for a game engine running in a browser, the counterpart of the book's own host:

```
const memory = new WebAssembly.Memory({ initial: 256 });   // 16
    MiB to start

const imports = {
  env: {
    memory,
    print: (ptr, len) => console.log(readString(ptr, len)),
    now: () => performance.now(),
  },
};

const { instance } = await
    WebAssembly.instantiateStreaming(fetch('game.wasm'), imports);
instance.exports._initialize?.();

function frame(time) {
  instance.exports.frame(time);
  requestAnimationFrame(frame);
}
requestAnimationFrame(frame);

function readString(ptr, len) {
  return new TextDecoder().decode(new Uint8Array(memory.buffer,
      ptr, len));
}
```

The memory is created by the host and imported, so the host can read strings out of it from the very first call. Initialisation is an explicit export, called after instantiation, for the reasons Chapter 8 gave. And the frame loop is driven from JavaScript, since only JavaScript can ask the browser for animation frames.

What you have seen 15.9

`WebAssembly.compile` and `instantiate`, preferably in their streaming forms, turn bytes into modules and instances. Modules can be inspected for their imports and exports, and sent to workers. Exported functions are JavaScript functions; imported JavaScript functions accept anything, converted by JavaScript's usual rules, with `i64` as BigInt and `v128` not allowed. Memories, tables and globals have JavaScript objects of their own, and failures are reported as `CompileError`, `LinkError` or `RuntimeError`.

The next chapter is about passing more than single numbers across the boundary.

Chapter 16

Across the Boundary

Numbers cross between JavaScript and WebAssembly as function arguments. Everything else, strings, arrays, structs, crosses through linear memory, and the design of that crossing decides much of how pleasant and how fast the integration between a compiled language and the browser is. This chapter covers the standard patterns: strings in both directions, bulk data handed straight to browser APIs, batching many calls into one, and calling back into WebAssembly from JavaScript.

Strings out 16.1

A string leaves WebAssembly as an address and a length, and JavaScript decodes the bytes. This is the book host's `print`, and it is as simple as it looks:

```
const decoder = new TextDecoder();     // UTF-8 by default
const print = (ptr, len) =>
  console.log(decoder.decode(new Uint8Array(memory.buffer, ptr,
      len)));
```

Create the decoder once. Create the `Uint8Array` on every call, since a cached one would be detached if memory had grown (Chapter 9). Both are cheap: the view is a few fields, and `decode` copies the bytes once into a JavaScript string.

If your language uses UTF-16 strings, as Java, C# and Dart do, decode with `new TextDecoder('utf-16le')` over the same bytes, or use `String.fromCharCode` on a `Uint16Array`, and the lengths are in code units rather than bytes.

Strings in 16.2

Going the other way, JavaScript has to put the bytes somewhere in linear memory. JavaScript cannot allocate in WebAssembly's heap by itself, since the allocator is WebAssembly code, so the module exports its allocator and JavaScript calls it:

```
(module
  (memory (export "memory") 1)
  (global $heap (mut i32) (i32.const 1024))
  (func (export "alloc") (param $n i32) (result i32)
    (global.get $heap)
    (global.set $heap (i32.add (global.get $heap) (local.get
        $n))))
  ;; count the bytes that start a UTF-8 code point, i.e. are not
      10xxxxxx
  (func (export "count_chars") (param $p i32) (param $n i32)
      (result i32)
    (local $count i32)
    (block $done
      (loop $next
        (br_if $done (i32.eqz (local.get $n)))
        (if (i32.ne (i32.and (i32.load8_u (local.get $p))
            (i32.const 0xc0)) (i32.const 0x80))
          (then (local.set $count (i32.add (local.get $count)
              (i32.const 1)))))
        (local.set $p (i32.add (local.get $p) (i32.const 1)))
        (local.set $n (i32.sub (local.get $n) (i32.const 1)))
        (br $next)))
    (local.get $count)))
```

```
import fs from 'node:fs';

const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'));
const { memory, alloc, count_chars } = instance.exports;
const encoder = new TextEncoder();

function passString(s) {
  // UTF-8 needs at most 3 bytes per UTF-16 code unit
  const max = s.length * 3;
  const ptr = alloc(max);
  const { written } = encoder.encodeInto(s, new
      Uint8Array(memory.buffer, ptr, max));
  return [ptr, written];
}

for (const s of ['hello', 'naïve', '日本語', '🦀🦀']) {
  const [ptr, len] = passString(s);
  console.log(s, len, 'bytes,', count_chars(ptr, len), 'code
      points');
}
```

```
hello 5 bytes, 5 code points
naïve 6 bytes, 5 code points
日本語 9 bytes, 3 code points
🦀🦀 8 bytes, 2 code points
```

`encodeInto` writes straight into memory without an intermediate copy. The allocation is sized for the worst case, which wastes a little space but avoids encoding twice. A real host would also free the string when the module is done with it, which needs a `free` export and an agreement about who owns the memory, the same agreement any C API needs.

When strings stay in JavaScript

> For a language whose strings are immutable and mostly passed around rather than inspected byte by byte, copying every string into linear memory may be the wrong design. With reference types, a module can hold JavaScript strings as `externref`s and call imported JavaScript functions to work with them. The *JS string builtins*, part of WebAssembly 3.0's JavaScript API, make this fast: a module imports functions such as `length` and `charCodeAt` from the reserved module `"wasm:js-string"`, and the engine implements them directly, without a call into JavaScript. This suits languages compiled with WasmGC (Chapter 22) best.

Bulk data without copying 16.3

The best way across the boundary is not to cross it at all. A typed array over linear memory is a window onto the module's own bytes, and browser APIs that take typed arrays read them directly. A game can build its vertex data in linear memory and hand it to WebGL with no copying:

```
// WebAssembly fills `count` floats at `ptr`; JavaScript uploads
    them as they are
const vertices = new Float32Array(memory.buffer, ptr, count);
gl.bufferSubData(gl.ARRAY_BUFFER, 0, vertices);
```

The same works for audio (`AudioBuffer.copyToChannel`), image data (`new ImageData(new Uint8ClampedArray(memory.buffer, ptr, w * h * 4), w, h)`), WebGPU's `queue.writeBuffer`, and anything else that accepts an `ArrayBufferView`. The browser reads the bytes once, when it needs them.

The constraint is alignment: a `Float32Array` must start at a multiple of 4 bytes, so the data's address must be aligned, which your allocator should guarantee.

Batching: the command buffer 16.4

Each call from WebAssembly to JavaScript costs something. In current engines the call itself is cheap, a few nanoseconds for an import with simple numeric parameters, but the JavaScript on the other side, and the browser API under that, often is not. A game that makes thousands of draw calls a frame, each with several arguments, spends much of its time at the boundary.

The usual fix is a *command buffer*: WebAssembly writes a compact description of what it wants done into memory, and JavaScript reads and executes the whole batch with one call per frame. Here is a tiny version, in which each command is an opcode followed by its arguments, all `i32`s:

```
(module
  (memory (export "memory") 1)
  (global $cmd (mut i32) (i32.const 0))
  (global $buffer i32 (i32.const 4096))

  (func $emit (param $x i32)
    (i32.store (i32.add (global.get $buffer) (global.get $cmd))
        (local.get $x))
    (global.set $cmd (i32.add (global.get $cmd) (i32.const 4))))

  ;; opcode 1: clear(colour); opcode 2: rect(x, y, w, h)
  (func $clear (param $colour i32)
    (call $emit (i32.const 1)) (call $emit (local.get $colour)))
  (func $rect (param $x i32) (param $y i32) (param $w i32) (param
      $h i32)
    (call $emit (i32.const 2))
    (call $emit (local.get $x)) (call $emit (local.get $y))
    (call $emit (local.get $w)) (call $emit (local.get $h)))

  ;; build one frame's commands; return where they are and how
      many bytes
  (func (export "frame") (param $t i32) (result i32 i32)
    (local $i i32)
    (global.set $cmd (i32.const 0))
    (call $clear (i32.const 0x202040))
    (loop $each
      (call $rect
        (i32.add (i32.mul (local.get $i) (i32.const 40))
            (local.get $t))
        (i32.const 10) (i32.const 32) (i32.const 32))
      (local.set $i (i32.add (local.get $i) (i32.const 1)))
      (br_if $each (i32.lt_u (local.get $i) (i32.const 3))))
    (global.get $buffer)
    (global.get $cmd)))
```

```
import fs from 'node:fs';

const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'));
const { memory, frame } = instance.exports;

// stands in for a canvas: a real host would call ctx.fillRect and
    so on
function execute(ptr, bytes) {
  const words = new Int32Array(memory.buffer, ptr, bytes / 4);
  for (let i = 0; i < words.length;) {
    switch (words[i++]) {
      case 1: console.log(`clear #${words[i++].toString(16)}`);
          break;
      case 2: console.log(`rect ${words[i++]},${words[i++]}
          ${words[i++]}x${words[i++]}`); break;
      default: throw new Error('bad command');
    }
  }
}

const [ptr, bytes] = frame(5);
execute(ptr, bytes);
```

```
clear #202040
rect 5,10 32x32
rect 45,10 32x32
rect 85,10 32x32
```

One call into WebAssembly produced four commands, executed by one JavaScript loop. The format can be anything both sides agree on. Real engines use the same idea at larger scale: Emscripten's WebGL bindings can proxy calls through a command buffer to a worker, and WebGPU's own design is built around recording commands into buffers.

A command buffer also makes the module easier to test: the commands are data, so a test can run a frame and compare the buffer with what it expects, without a browser.

Calling back into WebAssembly 16.5

Some browser APIs call you: animation frames, input events, audio processing. The JavaScript side receives the call and passes it on to an export. When the module needs to choose which of its functions gets called, it can pass a table index, or with reference types, export a table and let JavaScript call `table.get(i)`:

```
(module
  (import "env" "on_key" (func $on_key (param i32 i32)))   ;; key,
      callback index
  (import "env" "print_i32" (func $print (param i32)))
  (table (export "callbacks") 2 funcref)
  (elem (i32.const 0) $jump $fire)
  (func $jump (param $key i32) (call $print (i32.add (i32.const
      1000) (local.get $key))))
  (func $fire (param $key i32) (call $print (i32.add (i32.const
      2000) (local.get $key))))
  (func (export "_initialize")
    (call $on_key (i32.const 32) (i32.const 0))      ;; space:
        jump
    (call $on_key (i32.const 70) (i32.const 1))))    ;; F: fire
```

```
import fs from 'node:fs';

const handlers = new Map();
const env = {
  on_key: (key, index) => handlers.set(key, index),
  print_i32: (x) => console.log(x),
};
const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'), { env
    });
const { callbacks, _initialize } = instance.exports;
_initialize();

// a pretend keyboard: in a browser this would be
    addEventListener('keydown', ...)
for (const key of [70, 32, 70]) callbacks.get(handlers.get(key))
    (key);
```

```
2070
1032
2070
```

Designing the interface 16.6

The boundary between a compiled language and its host is an API like any other, and the same advice applies. A few points are specific to WebAssembly:

- **Prefer numbers.** `i32` and `f64` parameters cross most cheaply. `i64`s become BigInts, which allocate. Pass several scalars rather than a pointer to a struct when there are only a few.
- **Prefer few calls with much data to many calls with little.** Command buffers, and typed arrays over memory, beat per-item calls.
- **Make ownership explicit.** Every pointer that crosses needs a rule saying who frees it and when. The simplest rule is that memory passed in a call belongs to the caller and is valid only during the call.
- **Generate the glue.** Hand-written glue code drifts out of sync with the module. Tools like Rust's `wasm-bindgen` and Emscripten's `embind` generate both sides from one description. For your own language, generating the JavaScript glue from the same declarations that produce the imports is worth the effort as soon as the interface has more than a handful of functions.

What you have seen 16.7

Strings cross as bytes in memory: decoded with `TextDecoder` on the way out, encoded with `encodeInto` into memory the module allocated on the way in. Bulk data is best not copied at all, but handed to browser APIs as typed arrays over memory. Many small calls can be batched into a command buffer, executed once per frame. Callbacks reach the right function through a table.

The next chapter covers what happens when things go wrong.

Chapter 17

Traps

A *trap* is WebAssembly's only built-in kind of failure. When an instruction cannot do what it says, such as reading past the end of memory or dividing by zero, execution stops at once, the WebAssembly frames on the stack are abandoned, and the host gets an error. This chapter lists what traps, shows what the host sees, and covers what a compiler should do about traps, including using them deliberately.

Everything that traps 17.1

Here is one module with a function for each of the common traps, and a host that calls each one and prints what it catches:

```
(module
  (memory 1)
  (type $v (func))
  (table 2 funcref)
  (elem (i32.const 0) $f)
  (func $f (param i32))
  (func (export "unreachable") unreachable)
  (func (export "divide by zero") (drop (i32.div_u (i32.const 1)
      (i32.const 0))))
  (func (export "division overflow") (drop (i32.div_s (i32.const
      0x80000000) (i32.const -1))))
  (func (export "bad conversion") (drop (i32.trunc_f32_s
      (f32.const nan))))
  (func (export "memory out of bounds") (drop (i32.load (i32.const
      70000))))
  (func (export "table out of bounds") (call_indirect (type $v)
      (i32.const 5)))
  (func (export "null table entry") (call_indirect (type $v)
      (i32.const 1)))
  (func (export "signature mismatch") (call_indirect (type $v)
      (i32.const 0)))
  (func $deep (export "stack overflow") (call $deep)))
```

```
import fs from 'node:fs';

const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'));
for (const [name, f] of Object.entries(instance.exports)) {
  try {
    f();
  } catch (e) {
    console.log(`${name.padEnd(22)} ${e.constructor.name}:
        ${e.message}`);
  }
}
```

```
unreachable             RuntimeError: unreachable
divide by zero          RuntimeError: divide by zero
division overflow       RuntimeError: divide result unrepresentable
bad conversion          RuntimeError: float unrepresentable in
    integer range
memory out of bounds    RuntimeError: memory access out of bounds
table out of bounds     RuntimeError: table index is out of bounds
null table entry        RuntimeError: null function or function
    signature mismatch
signature mismatch      RuntimeError: null function or function
    signature mismatch
stack overflow          RangeError: Maximum call stack size
    exceeded
```

The messages are V8's; other engines word them differently, and the specification does not fix them. What is fixed is that each of these traps, on every engine, at exactly the same instruction. That determinism is what lets a test suite check traps at all, and Part VIII relies on it.

Stack overflow is the odd one out. It is a `RangeError` rather than a `RuntimeError`, because V8 reports it the same way as runaway JavaScript recursion, and the point at which it happens depends on the engine and its settings, so it is not deterministic in the same way.

Later features add a few more traps: dereferencing a null GC reference, a failed `ref.cast`, an out-of-bounds GC array access (Chapter 22), and an unaligned atomic access (Chapter 23). Instantiation can also trap, if an active data or element segment does not fit in its memory or table.

After a trap 17.2

A trap does not destroy the instance. Its memory, tables and globals keep whatever values they had at the moment of the trap, and its exports can be called again. Whether that is safe depends entirely on what the code was doing when it stopped.

Here is the problem in its most common form. A function with a shadow stack frame (Chapter 10) traps halfway through, so its epilogue never runs, and the stack pointer is never restored:

```
(module
  (memory 1)
  (global $sp (export "sp") (mut i32) (i32.const 65536))
  (func (export "work") (param $fail i32)
    (local $frame i32)
    (local.set $frame (i32.sub (global.get $sp) (i32.const 256)))
    (global.set $sp (local.get $frame))
    (if (local.get $fail) (then unreachable))
    (global.set $sp (i32.add (local.get $frame) (i32.const
        256)))))
```

```
import fs from 'node:fs';

const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'));
const { work, sp } = instance.exports;
work(0);
console.log('after a normal call:', sp.value);
for (let i = 0; i < 3; i++) {
  try { work(1); } catch (e) { console.log('trapped, sp is now',
      sp.value); }
}
```

```
after a normal call: 65536
trapped, sp is now 65280
trapped, sp is now 65024
trapped, sp is now 64768
```

Each trap leaked 256 bytes of shadow stack. Locks held, half-updated data structures, an allocator interrupted mid-operation: anything the code was in the middle of stays that way.

So a host has three reasonable responses to a trap:

1. **Treat it as fatal.** Report the error and discard the instance. This is the only response that is always safe, and the right default.
2. **Reset what you know about.** If the module's runtime keeps all its cross-call state in a few known places, such as the stack pointer, the host can restore them. Saving `sp.value` before each call from the host, and restoring it after a trap, fixes the example above. The module must be designed for this.
3. **Throw it away and start again.** Since instantiating an already compiled module is cheap, a game that runs untrusted or buggy scripts can create a fresh instance for each script after a trap, keeping the compiled module.

Traps and the language you are compiling 17.3

Your language has its own notion of failure: panics, assertions, exceptions, undefined behaviour. Each maps onto WebAssembly differently.

Undefined behaviour in the source can become whatever the WebAssembly instructions do. C's signed overflow simply wraps; a C null dereference reads address 0. Nothing traps unless an instruction traps by itself.

Checks your language promises, such as bounds checks on arrays or checks for integer overflow, are your compiler's job. Emit the comparison, and on failure, `unreachable`:

```
(module
  (memory 1)
  ;; an array of 4 i32s at address 16; get(i) checks i against the
     length
  (func (export "get") (param $i i32) (result i32)
    (if (i32.ge_u (local.get $i) (i32.const 4))
      (then unreachable))
    (i32.load offset=16 (i32.shl (local.get $i) (i32.const 2)))))
```

called as `get(7)`

```
RuntimeError: unreachable
```

The single `i32.ge_u` handles negative indices too, since as unsigned numbers they are enormous. Engines lay out the trapping path out of line, so the cost on the normal path is one compare and one predictable branch.

Panics with messages need the message to reach the developer. The usual approach is an imported function that records the message, followed by `unreachable`:

```
(module
  (import "env" "print" (func $print (param i32 i32)))
  (memory (export "memory") 1)
  (data (i32.const 0) "panic: index out of range")
  (func $panic (param $msg i32) (param $len i32)
    (call $print (local.get $msg) (local.get $len))
    unreachable)
  (func (export "main")
    (call $panic (i32.const 0) (i32.const 25))))
```

```
panic: index out of range
```

```
RuntimeError: unreachable
```

A host can make the import throw a JavaScript error carrying the message instead of printing it, which makes the message part of the error the caller catches. Either way, the trap's stack trace shows where the panic happened, provided the module has a name section.

Recoverable errors that your language expects programs to catch, such as exceptions, should not be traps at all, since WebAssembly code cannot catch a trap. They need the exception handling of Chapter 20, or a convention of returning error codes.

A deliberate design choice

> That WebAssembly code cannot catch traps is deliberate. A trap means the code has done something the machine could not do, and allowing it to continue could hide bugs or open security holes. The exception handling feature keeps this separation: `try_table` catches exceptions thrown with `throw`, never traps.

What you have seen 17.4

Traps stop execution at a precise instruction: unreachable code, integer division errors, failed float-to-int conversions, out-of-bounds memory and table accesses, and bad indirect calls. The host sees a `WebAssembly.RuntimeError`, or a `RangeError` for stack overflow. The instance survives, with whatever partial state the trap left, so the safest response is to discard it. Compilers use `unreachable` to implement the checks their languages promise, with an import to carry a message.

That ends Part V. Part VI covers the features added to WebAssembly since its first version.

PART VI

Beyond the First Version

WebAssembly 1.0 was deliberately minimal. Everything in this part came later: SIMD for numeric code, exception handling, tail calls, garbage collected structs and arrays, and shared-memory threads. Each one changes what a compiler can emit, and some change how a whole language should be mapped onto WebAssembly. The part starts with how to find out which of them the engine you are running on supports.

Chapter 18

Versions and Feature Detection

WebAssembly grows by *proposals*. Each one goes through a public process in the WebAssembly Community Group, is implemented in engines, and once it is finished, is folded into the next version of the specification. This chapter lists what each version contains, explains how a module can depend on newer features safely, and shows how to detect them at runtime.

The versions 18.1

WebAssembly 1.0, the MVP, standardised in 2019 but shipping in browsers since 2017: the four number types, functions, structured control flow, one linear memory, one table of functions, globals, imports and exports. Everything in Parts I to IV that is not marked otherwise.

WebAssembly 2.0, finished in 2022, collected the proposals that had shipped since:

Feature	What it adds	Chapter
sign extension	`i32.extend8_s` and friends	4
non-trapping float-to-int	the `trunc_sat` instructions	4
multiple values	multiple results, and blocks with parameters	5, 6
bulk memory	`memory.copy`, `memory.fill`, passive segments	9
reference types	`externref`, multiple tables, `table.get` and friends	11
mutable globals	importing and exporting mutable globals	11
fixed-width SIMD	the `v128` type and its instructions	19

WebAssembly 3.0, finished in 2025:

Feature	What it adds	Chapter
garbage collection	struct and array types managed by the engine	22
typed function references	non-null references, `call_ref`	22
exception handling	tags, `throw`, `try_table`, `exnref`	20
tail calls	`return_call` and friends	21
memory64	64-bit addresses	9
multiple memories	more than one memory per module	9
extended constant expressions	arithmetic in initialisers	11
relaxed SIMD	faster SIMD with some non-determinism	19
JS string builtins	fast JavaScript strings, in the JS API	16

Threads, despite shipping in every browser since around 2020, are still a separate proposal on the way to standardisation: the atomic instructions and shared memories of Chapter 23.

Proposals still in progress, as this book is written, include stack switching (for coroutines and effect handlers), JSPI (JavaScript promise integration, which lets synchronous WebAssembly code wait for a JavaScript promise), branch hinting, wide arithmetic (128-bit multiplication), custom page sizes, and memory control (unmapping and protecting pages).

Support in browsers 18.2

Every current version of Chrome, Edge, Firefox and Safari supports all of WebAssembly 2.0, and threads when the page is cross-origin isolated (Chapter 23). Support for the 3.0 features arrived at different times in different browsers: garbage collection and tail calls have been available everywhere since early 2025, while memory64 and some others reached Safari later than the others. The WebAssembly project keeps a current table at `webassembly.org/features`, and that, rather than any book, is the place to check.

For a game shipping today, the practical baseline is WebAssembly 2.0 plus whichever 3.0 features your language needs and your audience's browsers support. Everything in Parts I to V of this book is in that baseline.

Why feature detection matters 18.3

A module that uses an instruction an engine does not know fails to compile. There is no partial loading and no fallback: one unknown opcode anywhere, even in a function that is never called, and the whole module is rejected with a `CompileError`. So a program that wants to use a newer feature where it is available, without breaking where it is not, must ship two builds and choose between them before compiling either.

The way to choose is to ask the engine. Not by looking at the browser's version, which is unreliable, but by asking it to validate a tiny module that uses the feature. If it validates, the feature is there.

Detecting features 18.4

Here are four probes. Each is the smallest module that uses its feature:

```
(module
  (func (result v128) (v128.const i32x4 0 0 0 0)))
```

```
(module
  (func $f (return_call $f)))
```

```
(module
  (type $t (struct (field i32)))
  (func (result anyref) (struct.new $t (i32.const 0))))
```

```
(module
  (func (result exnref) (ref.null exn)))
```

```
import fs from 'node:fs';

for (const feature of ['simd', 'tail', 'gc', 'exnref']) {
  const bytes = fs.readFileSync(`${feature}.wasm`);
  console.log(feature.padEnd(8), WebAssembly.validate(bytes) ?
      'supported' : 'not supported');
}
```

```
simd     supported
tail     supported
gc       supported
exnref   supported
```

Node 22 supports all four. In a page you would embed the probe modules as byte arrays in the JavaScript, since they are tiny:

```
node -e
    "console.log(JSON.stringify([...require('fs').readFileSync('si
    md.wasm')]))"
```

```
[0,97,115,109,1,0,0,0,1,5,1,96,0,1,123,3,2,1,0,10,22,1,20,0,253,12
    ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,11]
```

and choose a build:

```
const simd = new
    Uint8Array([0,97,115,109,1,0,0,0,1,5,1,96,0,1,123,3,2,1,0,
  10,22,1,20,0,253,12,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,11]);
const url = WebAssembly.validate(simd) ? 'game.simd.wasm' :
    'game.wasm';
const { instance } = await
    WebAssembly.instantiateStreaming(fetch(url), imports);
```

The `wasm-feature-detect` package on npm is a ready-made collection of probes like these, one for every proposal.

Validation alone cannot detect everything. Threads need a shared memory, which a browser allows only on a cross-origin isolated page, so the probe for threads must check `crossOriginIsolated` too. JavaScript API features, such as the JS string builtins, are detected by trying the API.

> **For compiler writers**
>
> Make every post-2.0 feature a compiler flag, as LLVM does with `-msimd128`, `-mtail-call` and so on, and have your build produce the variants you want to ship. Keep the variants' code otherwise identical, so a bug that appears in one build and not the other can be blamed on the feature. Your test suite should run every variant (Chapter 27).

Telling tools which features you use 18.5

The tools in this book accept every standard feature by default. Older tools, and some current ones, need to be told: wabt's tools take flags such as `--enable-tail-call` or `--enable-all`; Binaryen's take `--enable-gc`, `--enable-simd` or `--all-features`. When a tool rejects a module that an engine accepts, an unenabled feature is the usual reason.

What you have seen 18.6

WebAssembly grows through proposals, gathered into versions: 2.0 added multiple values, bulk memory, reference types and SIMD; 3.0 added garbage collection, exceptions, tail calls, memory64 and more. A module using an unsupported feature fails to compile entirely, so programs that want newer features ship several builds and pick one by validating tiny probe modules at runtime.

The following chapters take the most important newer features one at a time, starting with SIMD.

Chapter 19

SIMD

The fifth value type, `v128`, holds 128 bits that instructions treat as a vector of *lanes*: sixteen 8-bit integers, eight 16-bit, four 32-bit or two 64-bit integers, or four `f32`s or two `f64`s. One instruction operates on every lane at once. For the numeric loops at the heart of a game engine, such as vector maths, particle updates, skinning, audio mixing and image processing, that can mean a speedup of two to four times. This chapter covers the instruction set and how to use it, then the *relaxed* SIMD instructions added in WebAssembly 3.0.

One type, many shapes 19.1

A `v128` has no lane shape of its own. As with signedness for integers, the *instruction* says how to interpret the bits, through a prefix naming the shape:

Shape	Lanes
`i8x16`	sixteen 8-bit integers
`i16x8`	eight 16-bit integers
`i32x4`	four 32-bit integers
`i64x2`	two 64-bit integers
`f32x4`	four `f32`s
`f64x2`	two `f64`s

`v128.const` writes a constant in any shape, and `extract_lane` reads one lane out as a scalar:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "main")
    (local $v v128)
    (local.set $v (i32x4.add
      (v128.const i32x4 1 2 3 4)
      (v128.const i32x4 10 20 30 40)))
    (call $print (i32x4.extract_lane 0 (local.get $v)))
    (call $print (i32x4.extract_lane 3 (local.get $v)))
    ;; the same 128 bits, read as sixteen bytes: lane 4 is the low
       byte of the second i32
    (call $print (i8x16.extract_lane_u 4 (local.get $v)))))
```

```
11
44
22
```

Lane indices are immediates: fixed in the code, not computed at runtime. Lane 0 is at the lowest address when a vector is stored to memory, so a `v128.load` of an array of four `f32`s puts element 0 in lane 0.

The instructions 19.2

The SIMD instruction set has about two hundred instructions, all behind the `fd` prefix. They fall into a few groups:

Group	Examples
construct	`v128.const`, `i32x4.splat` (copy a scalar into every lane)
lanes	`extract_lane`, `replace_lane`
memory	`v128.load`, `v128.store`, `v128.load32_splat`, `v128.load8x8_s` (widen as it loads), `v128.load32_lane`
arithmetic	`add`, `sub`, `mul`, `neg`, `abs`, `min`, `max`, `avgr_u`, saturating `add_sat_s` and `sub_sat_u`
floating point	`f32x4.div`, `sqrt`, `ceil`, `floor`, `nearest`, `pmin`, `pmax`
comparison	`eq`, `lt_s`, `gt`, ..., giving all-ones or all-zeros per lane
bitwise	`v128.and`, `or`, `xor`, `not`, `andnot`, `bitselect`
tests	`v128.any_true`, `i32x4.all_true`, `i32x4.bitmask`
rearrange	`i8x16.shuffle` (16 immediate lane indices), `i8x16.swizzle` (indices from a vector)
convert	`f32x4.convert_i32x4_s`, `i32x4.trunc_sat_f32x4_s`, `extend_low`, `narrow`
specialised	`i32x4.dot_i16x8_s`, `i16x8.q15mulr_sat_s`, `extmul`

Comparisons produce masks, not booleans: each lane becomes all ones if the comparison holds and all zeros if not. Masks combine with `bitselect` to choose between two vectors lane by lane, which is how branches are removed from vectorised code:

```
(module
  (import "env" "print_f32" (func $print (param f32)))
  (func (export "main")
    (local $x v128)
    (local $mask v128)
    (local $clamped v128)
    ;; clamp negative lanes to zero, without a branch
    (local.set $x (v128.const f32x4 -1.5 2.5 -3.5 4.5))
    (local.set $mask (f32x4.lt (local.get $x) (v128.const f32x4 0
        0 0 0)))
    (local.set $clamped (v128.bitselect
      (v128.const f32x4 0 0 0 0)     ;; where the mask is set
      (local.get $x)                 ;; where it is clear
      (local.get $mask)))
    (call $print (f32x4.extract_lane 0 (local.get $clamped)))
    (call $print (f32x4.extract_lane 1 (local.get $clamped)))
    (call $print (f32x4.extract_lane 2 (local.get $clamped)))))
```

```
0
2.5
0
```

(This particular clamp is also `f32x4.max` with zero, but the masking pattern works for any condition.)

Vector maths 19.3

Games are full of 4-component vectors and 4×4 matrices, and an `f32x4` holds a whole vector. Here is a matrix-vector product, with the matrix stored in column-major order as graphics APIs expect: the result is each column scaled by one component of the vector, summed.

```
(module
  (import "env" "print_f32" (func $print (param f32)))
  (memory 1)
  ;; a matrix that scales by 2 and translates by (10, 20, 30),
      column-major,
  ;; then the vector (1, 2, 3, 1)
  (data (i32.const 0)
    "\00\00\00\40" "\00\00\00\00" "\00\00\00\00" "\00\00\00\00"
        ;; column 0: 2 0 0 0
    "\00\00\00\00" "\00\00\00\40" "\00\00\00\00" "\00\00\00\00"
        ;; column 1: 0 2 0 0
    "\00\00\00\00" "\00\00\00\00" "\00\00\00\40" "\00\00\00\00"
        ;; column 2: 0 0 2 0
    "\00\00\20\41" "\00\00\a0\41" "\00\00\f0\41" "\00\00\80\3f")
        ;; column 3: 10 20 30 1
  (data (i32.const 64) "\00\00\80\3f" "\00\00\00\40"
      "\00\00\40\40" "\00\00\80\3f")

  ;; out = m * v, for a matrix at $m and vectors at $v and $out
  (func $mul_mat4_vec4 (param $m i32) (param $v i32) (param $out
      i32)
    (v128.store (local.get $out)
      (f32x4.add
        (f32x4.add
          (f32x4.mul (v128.load offset=0 (local.get $m))
              (v128.load32_splat offset=0 (local.get $v)))
          (f32x4.mul (v128.load offset=16 (local.get $m))
              (v128.load32_splat offset=4 (local.get $v))))
        (f32x4.add
          (f32x4.mul (v128.load offset=32 (local.get $m))
              (v128.load32_splat offset=8 (local.get $v)))
          (f32x4.mul (v128.load offset=48 (local.get $m))
              (v128.load32_splat offset=12 (local.get $v)))))))

  (func (export "main")
    (call $mul_mat4_vec4 (i32.const 0) (i32.const 64) (i32.const
        128))
    (call $print (f32.load offset=0 (i32.const 128)))
    (call $print (f32.load offset=4 (i32.const 128)))
    (call $print (f32.load offset=8 (i32.const 128)))
    (call $print (f32.load offset=12 (i32.const 128)))))
```

```
12
24
36
1
```

`v128.load32_splat` loads one `f32` and copies it into all four lanes, which is exactly what scaling a column by one component needs. The whole product is four loads of columns, four splat-loads, four multiplies and three adds.

Structure of arrays 19.4

Vectorising one 4-component vector at a time, as above, uses the lanes well only when the data happens to have four components. The bigger wins come from processing four *items* at a time: four particles, four vertices, four samples. That needs the data laid out

as a *structure of arrays*, with all the x coordinates together, then all the y coordinates, rather than an array of structures:

```
array of structures:   x0 y0 z0 x1 y1 z1 x2 y2 z2 x3 y3 z3 ...
structure of arrays:   x0 x1 x2 x3 ...   y0 y1 y2 y3 ...   z0 z1
    z2 z3 ...
```

Then a loop advances through each array four elements at a time. Here is a particle update, `x += vx * dt`, for one coordinate:

```
(module
  (import "env" "print_f32" (func $print (param f32)))
  (memory 1)
  ;; 8 positions at 0, 8 velocities at 1024
  (func $init
    (local $i i32)
    (loop $l
      (f32.store (i32.shl (local.get $i) (i32.const 2))
          (f32.convert_i32_u (local.get $i)))
      (f32.store offset=1024 (i32.shl (local.get $i) (i32.const
          2)) (f32.const 10))
      (local.set $i (i32.add (local.get $i) (i32.const 1)))
      (br_if $l (i32.lt_u (local.get $i) (i32.const 8)))))

  ;; x[i] += vx[i] * dt for n particles, n a multiple of 4
  (func $integrate (param $n i32) (param $dt f32)
    (local $p i32)
    (local $end i32)
    (local $dt4 v128)
    (local.set $dt4 (f32x4.splat (local.get $dt)))
    (local.set $end (i32.shl (local.get $n) (i32.const 2)))
    (loop $l
      (v128.store (local.get $p)
        (f32x4.add
          (v128.load (local.get $p))
          (f32x4.mul (v128.load offset=1024 (local.get $p))
              (local.get $dt4))))
      (local.set $p (i32.add (local.get $p) (i32.const 16)))
      (br_if $l (i32.lt_u (local.get $p) (local.get $end)))))

  (func (export "main")
    (call $init)
    (call $integrate (i32.const 8) (f32.const 0.5))
    (call $print (f32.load (i32.const 0)))
    (call $print (f32.load (i32.const 28)))))
```

```
5
12
```

A real version handles a count that is not a multiple of four, either by padding the arrays or with a scalar loop for the remainder.

> Auto-vectorisation, turning scalar loops into SIMD automatically, is hard, and engines do not do it: what you emit is what runs. LLVM vectorises C and Rust loops for WebAssembly when SIMD is enabled. For your own language, the most dependable approach is to expose vector types and operations directly, as `float4` or `Vec4` types with operator overloading, and compile them straight to `v128` instructions. Game programmers are used to writing SIMD-friendly code with such types, and it gives them predictable results.

SIMD and the boundary 19.5

A `v128` cannot cross into JavaScript: a function with a `v128` parameter or result is callable only from WebAssembly, and calling it from JavaScript is a `TypeError`. SIMD code reads its input from memory and writes its output to memory, and the boundary sees only addresses and counts.

Relaxed SIMD 19.6

The original SIMD instructions are fully deterministic, which sometimes costs speed. A fused multiply-add, for instance, must not be used for `f32x4.mul` followed by `f32x4.add`, because rounding once instead of twice gives different results. Some operations, such as a float-to-int conversion of out-of-range lanes, have no single fast implementation on both x86 and ARM, so the deterministic version is slow on one of them.

Relaxed SIMD, in WebAssembly 3.0, adds instructions that allow a small, documented set of results, so each engine can use its hardware's fastest instruction:

Instruction	Relaxation
`f32x4.relaxed_madd`, `relaxed_nmadd`	fused or unfused multiply-add
`i32x4.relaxed_trunc_f32x4_s`	out-of-range lanes give any of a few values
`i8x16.relaxed_swizzle`	out-of-range indices give zero or another lane
`f32x4.relaxed_min`, `relaxed_max`	NaN and signed zero handling may differ
`i32x4.relaxed_laneselect`	uses the whole lane mask or just its top bit
`i16x8.relaxed_dot_i8x16_i7x16_s`	dot products of small integers

```
(module
  (import "env" "print_f32" (func $print (param f32)))
  (func (export "main")
    ;; a*b + c, fused or not: the engine chooses
    (call $print (f32x4.extract_lane 0
      (f32x4.relaxed_madd
        (v128.const f32x4 1.5 0 0 0)
        (v128.const f32x4 2 0 0 0)
        (v128.const f32x4 1 0 0 0))))))
```

```
4
```

For this input both answers are the same, which is the common case: the results differ only in rounding of the last bit, or for lanes that are out of range. For graphics and audio, relaxed SIMD is almost always fine. For anything that must be reproducible across machines, such as lockstep networked physics, use the deterministic instructions.

Under the hood

Engines translate SIMD instructions one-for-one, or nearly, to SSE and AVX on x86 and to NEON on ARM. WebAssembly's vectors are fixed at 128 bits, the width every platform supports, so the wider AVX registers on desktop CPUs go unused. Instructions that do not exist on one platform, such as `i8x16.shuffle` with an arbitrary pattern on SSE, cost a few instructions instead of one. On x86, a shuffle pattern that matches a native instruction compiles to that instruction, so common patterns are cheap.

What you have seen 19.7

`v128` holds a 128-bit vector, and each instruction names the lane shape it works with. SIMD instructions cover arithmetic, comparisons that produce masks, lane selection, shuffles, conversions and loads that splat or widen. Vector maths maps naturally onto `f32x4`, and the big wins come from processing four items at once over data stored as a structure of arrays. Vectors stay inside WebAssembly. Relaxed SIMD trades bit-exact results for speed.

The next chapter covers exception handling.

Chapter 20

Exceptions

A language with exceptions needs a way to unwind the stack to a handler, and WebAssembly's hidden call stack (Chapter 5) means a compiler cannot do it by itself. Before exception handling was added, compilers resorted to calling out to JavaScript to throw and catch, or to checking a flag after every call, both of which are slow. The exception handling feature, in WebAssembly 3.0, adds the instructions to do it properly. This chapter covers them, how JavaScript sees them, and how to compile `try`, `catch` and `finally` with them.

Tags and throw 20.1

An exception is identified by a *tag*, declared at the module level with the types of the values the exception carries:

```
(tag $not_found (param i32))          ;; carries one i32
(tag $parse_error (param i32 i32))    ;; carries two
```

`throw` pops the tag's values and throws an exception with them. Control leaves the current function, and every function between it and the nearest handler, at once.

try_table and catch 20.2

A handler is a `try_table` block with a list of *catch clauses*. Each clause names a tag and a label to branch to when an exception with that tag is thrown inside the block. The exception's values are passed to the label as branch operands, so the label's block must have results of the right types:

```
(module
  (tag $not_found (param i32))
  (func $find (param $key i32) (result i32)
    (if (i32.gt_u (local.get $key) (i32.const 5))
      (then (throw $not_found (local.get $key))))
    (i32.mul (local.get $key) (i32.const 100)))
  (func (export "lookup") (param $key i32) (result i32)
    (block $handler (result i32)              ;; receives the
        exception's i32
      (try_table (result i32) (catch $not_found $handler)
        (call $find (local.get $key)))
      (return))                               ;; normal path:
          return the value
    ;; exceptional path: the key is on the stack
    (i32.mul (i32.const -1))))
```

called as `lookup(3)` and `lookup(9)`

```
300
-9
```

If `$find` returns normally, the `try_table` ends with its result on the stack and the function returns it. If `$find` throws `$not_found`, the `catch` clause branches to `$handler`, carrying the thrown key, and the code after the `$handler` block negates it (multiplying by −1, since the key is already on the stack and `i32.sub` would need it on top).

The four kinds of catch clause are:

Clause	Catches	Passes to the label
`catch $tag $l`	exceptions with `$tag`	the exception's values
`catch_ref $tag $l`	exceptions with `$tag`	the values, then an `exnref`
`catch_all $l`	every exception	nothing
`catch_all_ref $l`	every exception	an `exnref`

Clauses are tried in order. An exception that matches none of them continues outward to the next handler.

exnref and rethrowing 20.3

An `exnref` is a reference to a caught exception. Its main use is `throw_ref`, which throws it again, unchanged. That is how a handler that does some work and then lets the exception continue, such as a `finally` block or a destructor, is compiled:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (tag $error (param i32))
  (func $work (param $fail i32)
    (call $print (i32.const 1))
    (if (local.get $fail) (then (throw $error (i32.const 42))))
    (call $print (i32.const 2)))

  ;; try { work(fail) } finally { print(3) }
  (func $with_finally (param $fail i32)
    (block $on_exception (result exnref)
      (try_table (catch_all_ref $on_exception)
        (call $work (local.get $fail)))
      (call $print (i32.const 3))              ;; finally, on the
          normal path
      (return))
    (call $print (i32.const 3))                ;; finally, on the
        exceptional path
    (throw_ref))                               ;; and carry on
        unwinding

  (func (export "run") (param $fail i32)
    (block $caught (result i32)
      (try_table (catch $error $caught)
        (call $with_finally (local.get $fail)))
      (return))
    (call $print)))                            ;; print the
        error's value
```

called as `run(0)` and `run(1)`

```
1
2
3
1
3
42
```

The `finally` code appears twice, once on each path. That duplication is normal: a compiler emits the cleanup code once per way out of the protected region, or moves it into a function it calls from each.

Exceptions and JavaScript 20.4

Exceptions cross the boundary in both directions.

A WebAssembly exception that reaches JavaScript is a `WebAssembly.Exception` object. If the module exports the tag, JavaScript can check an exception's tag and read its values:

```
(module
  (tag $overflow (export "overflow") (param i32 i32))
  (func (export "add_checked") (param $a i32) (param $b i32)
      (result i32)
    (local $sum i64)
    (local.set $sum (i64.add (i64.extend_i32_s (local.get $a))
        (i64.extend_i32_s (local.get $b))))
    (if (i64.ne (local.get $sum) (i64.extend_i32_s (i32.wrap_i64
        (local.get $sum))))
      (then (throw $overflow (local.get $a) (local.get $b))))
    (i32.wrap_i64 (local.get $sum))))
```

```
import fs from 'node:fs';

const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'));
const { add_checked, overflow } = instance.exports;
console.log(add_checked(1, 2));
try {
  add_checked(2 ** 31 - 1, 1);
} catch (e) {
  console.log(e instanceof WebAssembly.Exception, e.is(overflow));
  console.log('overflow adding', e.getArg(overflow, 0), 'and',
      e.getArg(overflow, 1));
}
```

```
3
true true
overflow adding 2147483647 and 1
```

JavaScript can also create tags with `new WebAssembly.Tag({ parameters: ['i32'] })`, pass them to modules as imports, and throw WebAssembly exceptions itself with `new WebAssembly.Exception(tag, [values])`.

A JavaScript exception thrown into WebAssembly, from an imported function, can be caught by `catch_all` or `catch_all_ref`, since it has no tag the module knows. The module cannot look inside it, but it can run cleanup code and rethrow it with `throw_ref`, or swallow it:

```
(module
  (import "env" "risky" (func $risky (param i32) (result i32)))
  (func (export "safe") (param $x i32) (result i32)
    (block $failed
      (try_table (result i32) (catch_all $failed)
        (call $risky (local.get $x)))
      (return))
    (i32.const -1)))
```

```
import fs from 'node:fs';

const env = { risky: (x) => { if (x < 0) throw new
    Error('negative!'); return x * 2; } };
const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'), { env
    });
console.log(instance.exports.safe(21), instance.exports.safe(-5));
```

```
42 -1
```

The JavaScript API also provides `WebAssembly.JSTag`, a tag whose single `externref` value is the thrown JavaScript value, so a module that imports it can catch JavaScript exceptions with `catch` and inspect what was thrown.

Compiling try and catch 20.5

A source-level `try`/`catch` maps onto `try_table` directly: one tag per exception type the language distinguishes, or one tag for all exceptions carrying a pointer to an exception object, which is simpler and what C++ compilers do. Matching by the source type, for languages where a `catch` clause names a class and must also catch subclasses, then happens in the handler: catch the one tag, test the object's type, and `throw_ref` if it does not match.

Three details need care:

- **The shadow stack pointer.** If the code inside the `try` adjusted the stack pointer in linear memory (Chapter 10), unwinding skipped the epilogues that would have restored it. Save the stack pointer in a local at the start of the `try`, and restore it at the start of each handler.
- **Destructors and `defer`.** Code that must run when a scope exits by any route is a `catch_all_ref` that runs the code and rethrows, exactly like the `finally` example.
- **Traps are not exceptions.** No catch clause catches a trap. A language whose division by zero raises a catchable exception must test for zero and `throw`, rather than relying on the trap.

Under the hood

Engines implement exceptions as *zero-cost* on the normal path: entering a `try_table` emits no instructions, and the engine keeps a side table mapping code ranges to handlers, consulted only when something is thrown. Throwing is correspondingly expensive, since it walks the stack and allocates an exception object. Use exceptions for exceptional cases, not for control flow in hot loops.

The legacy instructions 20.6

Before the current design, browsers shipped an earlier version of the proposal, with `try`, `catch`, `catch_all`, `rethrow` and `delegate` instructions, and no `exnref`. Toolchains emitted it for several years, so you will find it in existing modules, and engines still accept it. It proved hard to optimise and to fit with other features, and was replaced by `try_table` and `exnref` in 2024. New compilers should emit only the new instructions.

What you have seen 20.7

Tags declare exception types. `throw` raises an exception, and `try_table` catches it with clauses that branch to labels, carrying the exception's values and optionally an `exnref`. `throw_ref` rethrows, which is how `finally` and destructors are compiled. Exceptions cross to and from JavaScript, as `WebAssembly.Exception` objects one way and as catch-all exceptions the other. Entering a handler is free; throwing is not.

The next chapter covers tail calls.

Chapter 21

Tail Calls

A call is a *tail call* when it is the last thing a function does: the caller will return whatever the callee returns, and has nothing else to do. A tail call does not need its own stack frame, since the caller's frame is finished with. Native compilers turn tail calls into jumps when they can. A WebAssembly compiler cannot, because there are no jumps between functions, so WebAssembly 3.0 added instructions that make a tail call explicitly. This chapter covers them and what they make possible.

return_call 21.1

`return_call $f` is `call $f` followed by `return`, except that the caller's frame is removed *before* the call rather than after. A recursive function written with it runs in constant stack space, however deep the recursion:

```
(module
  (func $count (export "count") (param $n i32) (result i32)
    (if (result i32) (i32.eqz (local.get $n))
      (then (i32.const 0))
      (else (return_call $count (i32.sub (local.get $n) (i32.const
          1)))))))
```

called as `count(1000000)`

```
0
```

The same function with an ordinary `call` runs out of stack long before a million levels:

```
(module
  (func $count (export "count") (param $n i32) (result i32)
    (if (result i32) (i32.eqz (local.get $n))
      (then (i32.const 0))
      (else (call $count (i32.sub (local.get $n) (i32.const
          1)))))))
```

called as `count(1000000)`

```
RangeError: Maximum call stack size exceeded
```

Tail calls are not limited to recursion. Any call in tail position can use `return_call`, including calls between different functions:

```
(module
  (func $is_even (export "is_even") (param $n i32) (result i32)
    (if (result i32) (i32.eqz (local.get $n))
      (then (i32.const 1))
      (else (return_call $is_odd (i32.sub (local.get $n)
          (i32.const 1))))))
  (func $is_odd (param $n i32) (result i32)
    (if (result i32) (i32.eqz (local.get $n))
      (then (i32.const 0))
      (else (return_call $is_even (i32.sub (local.get $n)
          (i32.const 1)))))))
```

called as `is_even(1000001)` and `is_even(1000000)`

```
0
1
```

Mutual recursion like this cannot be turned into a loop by a compiler working one function at a time, which is why the instruction is needed.

The rules 21.2

A tail call replaces the caller's return, so the callee's results must be what the caller returns: its result types must match the caller's, or be subtypes of them. The parameters can be anything:

```
(module
  (func $f (result i64) (i64.const 1))
  (func (result i32)
    (return_call $f)))
```

```
CompileError: WebAssembly.Module(): Compiling function #1 failed:
    return_call: tail call type error @+34
```

There are three instructions, one for each kind of call:

Instruction	Like
`return_call $f`	`call $f`
`return_call_indirect $table (type $t)`	`call_indirect`
`return_call_ref $t`	`call_ref` (Chapter 22)

Like `return`, a tail call ends the block it is in, and the code after it is unreachable.

What tail calls make possible 21.3

Functional languages. Scheme requires that tail calls run in constant space, and ML, Haskell, Elixir and similar languages assume it. Without tail calls, compilers for these languages used *trampolines*: functions return a description of the next call to a

loop that makes it. That works, but every call goes through the loop, which is slow and hard for engines to optimise.

Interpreters. A fast way to write an interpreter is to give each opcode its own handler function, and have each handler tail-call the next opcode's handler through a table. There is no central dispatch loop, so each handler's indirect jump is predicted separately by the hardware. This is the technique behind several fast interpreters, and it needs tail calls:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (memory 1)
  ;; an accumulator machine. 1 n: acc += n; 2 n: acc *= n; 3:
     print acc; 0: halt
  (data (i32.const 0) "\01\05" "\02\07" "\01\03" "\03" "\00")
  (type $handler (func (param i32 i32)))         ;; pc, acc
  (table 4 funcref)
  (elem (i32.const 0) $halt $add $mul $out)

  ;; fetch the opcode at pc and tail-call its handler
  (func $next (param $pc i32) (param $acc i32)
    (return_call_indirect (type $handler)
      (local.get $pc) (local.get $acc)
      (i32.load8_u (local.get $pc))))

  (func $halt (param $pc i32) (param $acc i32))
  (func $add (param $pc i32) (param $acc i32)
    (return_call $next
      (i32.add (local.get $pc) (i32.const 2))
      (i32.add (local.get $acc) (i32.load8_u offset=1 (local.get
          $pc)))))
  (func $mul (param $pc i32) (param $acc i32)
    (return_call $next
      (i32.add (local.get $pc) (i32.const 2))
      (i32.mul (local.get $acc) (i32.load8_u offset=1 (local.get
          $pc)))))
  (func $out (param $pc i32) (param $acc i32)
    (call $print (local.get $acc))
    (return_call $next (i32.add (local.get $pc) (i32.const 1))
        (local.get $acc)))

  (func (export "main")
    (call $next (i32.const 0) (i32.const 0))))
```

The program computes (0 + 5) × 7 + 3:

```
38
```

Every handler ends by tail-calling `$next`, which tail-calls the next handler, so however long the program runs, the stack never grows. A real interpreter would inline `$next` into each handler, so that every handler ends with its own `return_call_indirect`, giving the hardware's branch predictor a separate history for each.

State machines and continuations. A state machine can make each state a function, with transitions as tail calls, which is often

clearer than a dispatch loop. A compiler using continuation-passing style, where every function takes the function to call next as a parameter, needs tail calls for every call.

Under the hood

A tail call is not always free. When the callee needs more stack space for arguments than the caller's frame had, the engine has to move things around before jumping, and V8's implementation of `return_call_indirect` is somewhat slower than a plain `call_indirect`. In return, the stack cannot overflow, and for interpreters the dispatch is faster overall. Measure before assuming a tail call is faster than an ordinary one; use it when you need the constant stack space.

What you have seen 21.4

`return_call`, `return_call_indirect` and `return_call_ref` call a function in place of the caller, so a chain of tail calls runs in constant stack space. The callee must return what the caller returns. Tail calls let functional languages compile calls directly instead of through trampolines, and enable fast interpreter designs and state machines written as functions.

The next chapter covers the biggest addition to WebAssembly since its first version: garbage collection.

Chapter 22

Garbage Collection

Everything so far has kept data in linear memory, where the program manages it. That suits C, C++, Rust, Zig, Odin, and any language where the programmer or the compiler decides when memory is freed. A language with a garbage collector has a harder time: it must bring its own collector, compiled to WebAssembly, and that collector cannot see the call stack to find roots. WebAssembly 3.0's *garbage collection* feature, usually called *WasmGC*, takes a different approach: the module declares struct and array types, allocates objects of those types, and the engine's own collector, the one that already manages JavaScript objects, manages them. This chapter covers the type system, the instructions, and how to decide whether your language should use it.

Struct types 22.1

A struct type declares a list of fields, each with a type and an optional `mut`:

```
(module
  (import "env" "print_f64" (func $print (param f64)))
  (type $point (struct (field $x (mut f64)) (field $y (mut f64))))
  (func (export "main")
    (local $p (ref $point))
    (local.set $p (struct.new $point (f64.const 1.5) (f64.const
        2.5)))
    (struct.set $point $x (local.get $p) (f64.const 10))
    (call $print (struct.get $point $x (local.get $p)))
    (call $print (struct.get $point $y (local.get $p)))))
```

```
10
2.5
```

`struct.new $point` pops a value for each field and pushes a reference to a new object. `struct.get` and `struct.set` take the type and the field as immediates, so the field's position and type are known statically and the access compiles to a single load or store.

The local's type is `(ref $point)`: a reference to a `$point`. There is no pointer arithmetic, no way to get the address of a field, and no way to free the object. It lives until nothing refers to it.

Fields can also be *packed*, `i8` or `i16`, to save space. Packed fields are read with `struct.get_s` or `struct.get_u`, which sign- or zero-extend them to an `i32`.

Nullable and non-null references 22.2

A reference type is either non-null, `(ref $t)`, or nullable, `(ref null $t)`. A non-null local must be assigned before it is read, which the validator checks. A nullable one starts as null, and every access through it is checked at runtime:

```
(module
  (type $point (struct (field $x f64)))
  (func (export "main") (result f64)
    (local $p (ref null $point))
    (struct.get $point $x (local.get $p))))
```

```
RuntimeError: dereferencing a null pointer
```

`ref.as_non_null` converts a nullable reference to a non-null one, trapping if it is null. `br_on_null` and `br_on_non_null` branch on whether a reference is null, and refine its type on the path where it is not. A compiler for a language with non-nullable types can use non-null references throughout, and the engine can then omit the null checks.

Arrays 22.3

An array type has a single element type, and each array has a length fixed when it is created:

```
(module
  (import "env" "print_f64" (func $print (param f64)))
  (type $floats (array (mut f64)))
  (func $sum (param $a (ref $floats)) (result f64)
    (local $i i32)
    (local $total f64)
    (block $done
      (loop $next
        (br_if $done (i32.ge_u (local.get $i) (array.len
            (local.get $a))))
        (local.set $total (f64.add (local.get $total)
          (array.get $floats (local.get $a) (local.get $i))))
        (local.set $i (i32.add (local.get $i) (i32.const 1)))
        (br $next)))
    (local.get $total))
  (func (export "main")
    (local $a (ref $floats))
    (local.set $a (array.new $floats (f64.const 0.5) (i32.const
        10)))   ;; ten 0.5s
    (array.set $floats (local.get $a) (i32.const 3) (f64.const
        100))
    (call $print (call $sum (local.get $a)))
    (call $print (call $sum (array.new_fixed $floats 3
      (f64.const 1) (f64.const 2) (f64.const 3))))))
```

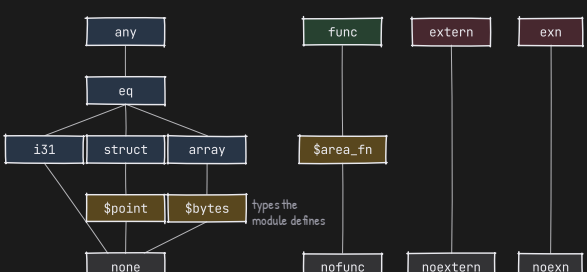
```
104.5
6
```

The array instructions include `array.new` (every element the same), `array.new_default` (zeros or nulls), `array.new_fixed` (the elements given), `array.new_data` and `array.new_elem` (from a segment), `array.get` and `array.set`, `array.len`, `array.fill`, `array.copy`, and `array.init_data`. Every access is bounds-checked, and an out-of-bounds index traps.

An array of `i8` is the natural representation of a byte string, and `array.new_data` creates one from a data segment, so string literals can live in data segments and be turned into GC arrays when needed.

The type hierarchy 22.4

GC references fit into a hierarchy of *heap types*:



The heap type hierarchy. Each type is a subtype of the one above it; the bottom types have only the null value.

- `any` is the top of the internal hierarchy. Every struct, array and `i31` reference is an `anyref`.
- `eq` is the types whose references can be compared with `ref.eq`.
- `struct` and `array` are the supertypes of every struct type and every array type.
- `i31` is the type of unboxed 31-bit integers (section 22.7).
- `func` is a separate hierarchy for function references, and `extern` one for host references; `exn` holds exception references.
- `none`, `nofunc`, `noextern` and `noexn` are the *bottom* types, the types of `null` in each hierarchy.

The familiar `funcref` and `externref` from Chapter 11 are abbreviations for `(ref null func)` and `(ref null extern)`. Likewise `anyref`, `eqref`, `structref`, `arrayref` and `i31ref` are the nullable references to the abstract types.

Subtyping and casts 22.5

A struct type can be declared a *subtype* of another, if it has the same fields at the start, possibly followed by more. A reference to the subtype can then be used wherever the supertype is expected. This is how class hierarchies are compiled:

```
(type $animal (sub (struct (field $legs i32))))
(type $bird (sub $animal (struct (field $legs i32) (field
    $wingspan f32))))
```

A type written with `sub` can have subtypes; a type without it is *final*, and cannot. Final types are the default because the engine can then test the exact type with a single comparison.

Going the other way, from a supertype to a subtype, needs a runtime check. `ref.test` tests whether a reference is of a type, `ref.cast` converts it and traps if it is not, and `br_on_cast` and `br_on_cast_fail` branch on the result:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (type $a (struct (field i32)))
  (type $b (array i8))
  ;; 1 for a struct $a, 2 for an array $b, 3 for an i31, 0
      otherwise
  (func $kind (param $x anyref) (result i32)
    (if (ref.test (ref $a) (local.get $x)) (then (return
        (i32.const 1))))
    (if (ref.test (ref $b) (local.get $x)) (then (return
        (i32.const 2))))
    (if (ref.test (ref i31) (local.get $x)) (then (return
        (i32.const 3))))
    (i32.const 0))
  (func (export "main")
    (call $print (call $kind (struct.new $a (i32.const 1))))
    (call $print (call $kind (array.new_default $b (i32.const
        4))))
    (call $print (call $kind (ref.i31 (i32.const 7))))
    (call $print (call $kind (ref.null any)))))
```

```
1
2
3
0
```

Recursive types 22.6

Types often refer to each other. A linked list node refers to its own type, which needs nothing special:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (type $node (struct (field $value i32) (field $next (ref null
      $node))))
  (func (export "main")
    (local $list (ref null $node))
    (local $i i32)
    (local $sum i32)
    ;; build 1 -> 2 -> 3 -> 4 -> 5 by pushing 5, 4, 3, 2, 1 onto
        the front
    (local.set $i (i32.const 5))
    (loop $build
      (local.set $list (struct.new $node (local.get $i) (local.get
          $list)))
      (local.set $i (i32.sub (local.get $i) (i32.const 1)))
      (br_if $build (local.get $i)))
    ;; walk it
    (block $end
      (loop $walk
        (br_on_null $end (local.get $list))      ;; leaves a non-
            null reference
        (struct.get $node $value)                ;; ... which this
            consumes
        (local.set $sum (i32.add (local.get $sum)))
        (local.set $list (struct.get $node $next (ref.as_non_null
            (local.get $list))))
        (br $walk)))
    (call $print (local.get $sum))))
```

```
15
```

When several types refer to one another, they must be declared together in a *recursion group*, `(rec ...)`. A class with a vtable is the classic case: the object refers to the vtable, the vtable holds functions, and the functions take the object. Here is Chapter 11's shapes example again, with GC objects in place of linear memory:

```
(module
  (import "env" "print_f64" (func $print (param f64)))
  (rec
    (type $shape (sub (struct (field $vt (ref $vtable)))))
    (type $vtable (struct (field $area (ref $area_fn))))
    (type $area_fn (func (param (ref $shape)) (result f64))))
  (type $circle (sub final $shape (struct (field $vt (ref
      $vtable)) (field $r f64))))
  (type $rect (sub final $shape (struct (field $vt (ref $vtable))
      (field $w f64) (field $h f64))))

  (func $circle_area (type $area_fn) (param $s (ref $shape))
      (result f64)
    (local $c (ref $circle))
    (local.set $c (ref.cast (ref $circle) (local.get $s)))
    (f64.mul (f64.const 3.14159)
      (f64.mul (struct.get $circle $r (local.get $c)) (struct.get
          $circle $r (local.get $c)))))
  (func $rect_area (type $area_fn) (param $s (ref $shape)) (result
      f64)
    (local $r (ref $rect))
    (local.set $r (ref.cast (ref $rect) (local.get $s)))
    (f64.mul (struct.get $rect $w (local.get $r)) (struct.get
        $rect $h (local.get $r))))

  (global $circle_vt (ref $vtable) (struct.new $vtable (ref.func
      $circle_area)))
  (global $rect_vt (ref $vtable) (struct.new $vtable (ref.func
      $rect_area)))

  ;; shape.area(): load the vtable, load the method, call it
  (func $area (param $s (ref $shape)) (result f64)
    (call_ref $area_fn
      (local.get $s)
      (struct.get $vtable $area (struct.get $shape $vt (local.get
          $s)))))

  (func (export "main")
    (call $print (call $area (struct.new $circle (global.get
        $circle_vt) (f64.const 2))))
    (call $print (call $area (struct.new $rect (global.get
        $rect_vt) (f64.const 3) (f64.const 4))))))
```

```
12.56636
12
```

Two new things appear here. `ref.func` creates a *typed* function reference, of type `(ref $area_fn)`, and `call_ref` calls through one. Unlike `call_indirect`, `call_ref` needs no table, and no runtime signature check, since the reference's type already guarantees the signature. Typed function references are also how closures are compiled with GC: a struct holding the captured variables and a `(ref $fn)`.

The cast in each area function is the price of a method receiving its object as the base type. Engines make a cast to a final type cheap: one load and one compare.

Type equivalence

> Two modules that declare structurally identical types get the same types, because WebAssembly types are compared *structurally*, by recursion group: two recursion groups with the same shape define the same types. This is what lets separately compiled modules exchange GC objects without agreeing on names. It also means two classes with identical fields would be the same type, so compilers that need nominal types to stay distinct add a distinguishing field, or put them in different recursion groups.

i31ref 22.7

An `i31ref` is a 31-bit integer stored directly in a reference, with no allocation. `ref.i31` makes one from the low 31 bits of an `i32`, and `i31.get_s` and `i31.get_u` read it back:

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (func (export "main")
    (call $print (i31.get_s (ref.i31 (i32.const -5))))
    (call $print (i31.get_u (ref.i31 (i32.const -5))))))
```

```
-5
2147483643
```

Dynamically typed languages represent every value as a reference, and `i31ref` lets small integers share that representation without allocating a box for each. An `anyref` can hold either an object or an `i31`, and `ref.test` or `br_on_cast` tells them apart. OCaml, Scheme and Python compilers targeting WasmGC use it this way.

GC objects and JavaScript 22.8

GC references can be passed to JavaScript and back. JavaScript sees them as opaque objects: it can store them, pass them back, and compare them, but it cannot read their fields or call methods on them:

```
(module
  (type $point (struct (field $x f64) (field $y f64)))
  (func (export "make") (param f64 f64) (result (ref $point))
    (struct.new $point (local.get 0) (local.get 1)))
  (func (export "x") (param (ref $point)) (result f64)
    (struct.get $point $x (local.get 0))))
```

```
import fs from 'node:fs';

const { instance } = await
    WebAssembly.instantiate(fs.readFileSync('main.wasm'));
const p = instance.exports.make(3, 4);
console.log(typeof p, Object.keys(p).length, p.x);
console.log(instance.exports.x(p));
try {
  instance.exports.x({ x: 3 });
} catch (e) {
  console.log(e.constructor.name);
}
```

```
object 0 undefined
3
TypeError
```

Passing an ordinary JavaScript object where a `(ref $point)` is expected is a `TypeError`, checked at the boundary. Inside WebAssembly, `extern .convert_any` and `any.convert_extern` convert between the `any` and `extern` hierarchies, so a value of either kind can be stored as the other.

Most importantly, the engine has *one* garbage collector for JavaScript and WasmGC objects. A WasmGC object referring to a JavaScript object, which refers back to it, is collected when both are unreachable. With a collector in linear memory, such cycles between the two heaps leak.

Should your language use it? 22.9

WasmGC is not a better linear memory. It is a different model, with different strengths:

	Linear memory	WasmGC
Memory management	yours: manual, reference counting, or your own collector	the engine's collector
Layout control	complete: unions, interior pointers, custom alignment	structs and arrays only; no interior pointers, no unions
Code size	includes your allocator and collector	no runtime needed
Finding roots	needs a shadow stack (Chapter 10)	automatic
Cycles with JavaScript	leak, unless managed by hand	collected
Pauses	yours to control	the engine's, short and mostly concurrent
Interop with C libraries compiled to Wasm	direct: same memory	needs copying
Bulk data for browser APIs	typed arrays over memory, no copy	must be copied out

For a language whose values are naturally objects, such as Java, Kotlin, Dart, OCaml, Scheme or a scripting language, WasmGC is usually the better choice: smaller modules, no collector to write, and easy interop with JavaScript. Kotlin, Dart and Java (through J2Wasm) target it in production.

For a game engine's systems language, linear memory usually remains the better choice. A game wants control over layout and allocation, wants to pass vertex and audio data to browser APIs without copying, wants to share memory with C libraries, and wants to decide itself when memory is reclaimed, for instance freeing a whole frame's arena at once. Nothing stops a language from using both: linear memory for bulk data, GC references for a scripting layer's objects, or for handles to host objects.

What you have seen 22.10

WasmGC adds struct and array types whose objects the engine allocates and collects. References are typed, nullable or not, and fit into a hierarchy with `any`, `eq`, `struct`, `array` and `i31` in one tree and functions and host references in others. Subtyping, casts and recursion groups let classes and vtables be expressed directly, typed function references replace tables for calls, and `i31ref`

unboxes small integers. JavaScript sees GC objects as opaque, and one collector manages both heaps. It is the natural target for object-oriented and functional languages, and less so for systems languages that want control over memory.

The next chapter covers the last major feature: threads.

Chapter 23

Threads and Atomics

A game engine wants every core it can get: physics on one, animation on another, asset decoding on the rest. WebAssembly has no instruction to start a thread. Instead, the host starts threads, Web Workers in a browser, and gives each one an instance of the same module, all sharing one memory. What WebAssembly provides is the rest: memories that can be shared, *atomic* instructions to access them safely, and instructions to wait and wake. This chapter covers them, and how a compiler and its runtime fit threads together.

Shared memory 23.1

A memory declared `shared` can be imported by several instances at once, each running on its own thread, all seeing the same bytes. A shared memory must have a maximum size, because engines reserve its whole address range up front, so it never moves when it grows:

```
(module
  (memory (export "memory") 1 100 shared))
```

On the JavaScript side, a shared memory's `buffer` is a `SharedArrayBuffer` instead of an `ArrayBuffer`, and it can be sent to a worker with `postMessage`, which shares it rather than copying it. The usual arrangement is for the host to create the memory, and pass it to every instance as an import:

```
const memory = new WebAssembly.Memory({ initial: 256, maximum:
    16384, shared: true });
const module = await
    WebAssembly.compileStreaming(fetch('engine.wasm'));
for (let i = 0; i < navigator.hardwareConcurrency - 1; i++) {
  const worker = new Worker('worker.js');
  worker.postMessage({ module, memory, id: i });     // each
      worker instantiates the module
}
```

Each worker instantiates the same compiled module with the same memory, and calls an export to start its work.

Cross-origin isolation

> Browsers allow `SharedArrayBuffer`, and so shared memories, only on pages that are *cross-origin isolated*: served with the headers `Cross-Origin-Opener-Policy: same-origin` and `Cross-Origin-Embedder-Policy: require-corp`. These stop the page sharing a process with content from other sites, which closes off timing attacks that shared memory would otherwise make possible. A game that wants threads must be served with these headers, and every resource it loads from another origin must allow it. JavaScript checks with `crossOriginIsolated`.

Atomic instructions 23.2

Ordinary loads and stores to shared memory are allowed, but when two threads touch the same location and at least one writes, the result is a *data race*: each thread may see the other's writes late, partially, or in a surprising order. *Atomic* instructions access memory indivisibly and in a single order that every thread agrees on. They all live behind the `fe` prefix:

Instructions	Do
`i32.atomic.load`, `i64.atomic.load`, and 8-, 16- and 32-bit `_u` forms	load atomically
`i32.atomic.store` and the narrow forms	store atomically
`i32.atomic.rmw.add`, `.sub`, `.and`, `.or`, `.xor`, `.xchg`	read, modify and write in one step; push the old value
`i32.atomic.rmw.cmpxchg`	compare-and-swap: write the new value only if the old one matches; push the old value
`memory.atomic.wait32`, `wait64`	sleep until woken, if a location holds an expected value
`memory.atomic.notify`	wake threads waiting on a location
`atomic.fence`	order surrounding memory accesses

Every `i32` form has an `i64` counterpart. Atomic accesses must be naturally aligned, and an unaligned one traps, unlike every other memory access:

```
(module
  (memory 1 1 shared)
  (func (export "main") (result i32)
    (i32.atomic.load (i32.const 2))))
```

```
RuntimeError: operation does not support unaligned accesses
```

The atomic instructions also work on unshared memories, where they behave like ordinary accesses (with the alignment rule). That lets the same code run single-threaded without a separate build.

All atomic instructions are *sequentially consistent*: there is one global order of all atomic operations, consistent with each thread's program order. That is the strongest and simplest ordering, matching C++'s `memory_order_seq_cst`. A proposal for weaker, cheaper orderings is in progress.

Threads counting together 23.3

Here are four threads each adding 1 to a shared counter 100,000 times. The module is the same for every thread:

```
(module
  (import "env" "memory" (memory 1 1 shared))
  (func (export "count") (param $n i32)
    (loop $again
      (drop (i32.atomic.rmw.add (i32.const 0) (i32.const 1)))
      (local.set $n (i32.sub (local.get $n) (i32.const 1)))
      (br_if $again (local.get $n))))
  (func (export "total") (result i32)
    (i32.atomic.load (i32.const 0))))
```

The worker instantiates it with the memory it is given, counts, and reports back:

```
import { parentPort, workerData } from 'node:worker_threads';

const { module, memory } = workerData;
const instance = new WebAssembly.Instance(module, { env: { memory
    } });
instance.exports.count(100000);
parentPort.postMessage('done');
```

And the main thread creates the memory, compiles the module once, and starts the workers:

```
import fs from 'node:fs';
import { Worker } from 'node:worker_threads';

const memory = new WebAssembly.Memory({ initial: 1, maximum: 1,
    shared: true });
const module = new
    WebAssembly.Module(fs.readFileSync('counter.wasm'));

const workers = Array.from({ length: 4 },
  () => new Worker('./worker.mjs', { workerData: { module, memory
      } }));
await Promise.all(workers.map((w) => new Promise((done) =>
    w.once('message', done))));
workers.forEach((w) => w.terminate());

const main = new WebAssembly.Instance(module, { env: { memory }
    });
console.log(main.exports.total());
```

```
400000
```

With an ordinary `i32.load`, `i32.add` and `i32.store` in place of the atomic add, the threads would overwrite each other's updates, and the total would come out lower, by a different amount on each run. (That version is not shown, since its output is not reproducible.)

Node's `worker_threads` stand in for Web Workers here. In a browser, the worker would receive the module and memory through `postMessage` rather than `workerData`, but everything else is the same.

Waiting and waking 23.4

Spinning on an atomic load wastes a core. `memory.atomic.wait32` puts a thread to sleep instead. It takes an address, an expected value, and a timeout in nanoseconds (−1 for none). If the location does not hold the expected value, it returns 1 at once; otherwise it sleeps until another thread calls `memory.atomic.notify` on that address (returning 0) or the timeout passes (returning 2):

```
(module
  (import "env" "print_i32" (func $print (param i32)))
  (memory 1 1 shared)
  (func (export "main")
    ;; the location holds 0, not 5: returns 1, "not equal",
       without sleeping
    (call $print (memory.atomic.wait32 (i32.const 0) (i32.const 5)
       (i64.const -1)))
    ;; it does hold 0: sleeps for the 1 ms timeout, then returns
       2, "timed out"
    (call $print (memory.atomic.wait32 (i32.const 0) (i32.const 0)
       (i64.const 1000000)))
    ;; wakes nobody, since nobody is waiting: returns the number
       woken
    (call $print (memory.atomic.notify (i32.const 0) (i32.const
       1)))))
```

```
1
2
0
```

The check and the sleep happen atomically, so a thread cannot miss a notification that arrives between checking the value and going to sleep. These two instructions are exactly what is needed to build mutexes, condition variables, semaphores and job queues, the same way *futexes* are used on Linux.

A browser does not allow the main thread to block, so `wait` traps there. The main thread must use JavaScript's `Atomics.waitAsync`, or

avoid waiting altogether, which suits a game anyway: the main thread runs the frame loop, and hands work to workers.

What is per thread 23.5

Each thread has its own instance, so anything that belongs to an instance rather than to the shared memory is automatically per thread:

- **Globals** are per instance, so a mutable global is thread-local storage. The shadow stack pointer (Chapter 10) is a global, so each thread automatically has its own. Each thread does need its own *area* of memory for its stack, which the runtime allocates when the thread starts and stores in that thread's stack pointer global.
- **Tables** are per instance, unless imported and shared. A function pointer that one thread stores in memory must mean the same function on every thread, so either every instance must put functions at the same table indices, which they do if the table is filled only by the module's own element segments, or tables must be shared too.
- **Data segments** are applied again by every instance, which would reset the shared memory each time a thread starts. Toolchains make data segments passive (Chapter 9), and have one thread copy them in with `memory.init`, guarded by an atomic flag.

> **For compiler writers**
>
> LLVM's `-pthread` build mode shows the whole design: a shared imported memory, passive data segments initialised once, a thread-local area for each thread with its own stack pointer and copies of thread-local variables, and a small runtime that starts workers. Emscripten builds `pthread_create` and the rest of the POSIX threads API on top of this. A compiler for a new language needs the same pieces, though not necessarily a POSIX-shaped API: a fixed pool of worker threads pulling jobs from a queue suits a game engine well, and keeps the runtime small.

What you have seen 23.6

Threads are host workers, each with an instance of the same module, sharing a `shared` memory that has a fixed maximum size. Atomic loads, stores and read-modify-write operations access it safely, in a single sequentially consistent order, and must be aligned. `wait` and `notify` let threads sleep instead of spinning, but not on a browser's main thread. Globals are per thread, so the stack pointer and thread-local storage come naturally, while data

segments and tables need care. Pages must be cross-origin isolated to use any of it.

That ends Part VI. Part VII brings everything together: how to map a language onto WebAssembly, and how engines turn the result into fast machine code.

PART VII

Compiling to WebAssembly

The earlier parts described the machine one feature at a time. This part looks at it from the compiler's side. The first chapter goes through the decisions involved in mapping a language onto WebAssembly, from scalar types to calling conventions to coroutines, with a worked example. The second explains how engines turn a module into machine code, and what that means for the code your compiler should emit.

Chapter 24

Mapping a Language onto WebAssembly

A compiler targeting WebAssembly makes the same decisions as any backend (how values are represented, how functions are called, where memory comes from) but against a machine with unusual constraints. This chapter goes through those decisions in the order a backend meets them, collecting the advice of the earlier chapters into one place, and finishes by compiling a small program by hand.

The decisions 24.1

Here they are as a checklist. Each row refers to where the book covers it.

Decision	Usual choice	Alternatives	Chapter
where objects live	linear memory	WasmGC	10, 22
pointer size	32-bit	memory64	9
control flow	from the syntax tree	CFG and stackifier	6, 7
address-taken locals	shadow stack	(none)	10
function pointers	table indices	typed function references	11, 22
small aggregates	flattened into scalars	memory	5
large aggregates	pointer to caller's frame	GC references	10
errors	return values	exceptions, traps	17, 20
strings	UTF-8 slices in memory	JS strings as `externref`	16
generics	monomorphisation	boxing and dictionaries	this chapter
coroutines	state machines	stack switching, JSPI	7, this chapter
threads	worker pool, shared memory	none	23

Scalars 24.2

Every scalar in your language becomes one of the four number types, as Chapter 4 described:

Source type	WebAssembly	Notes
`bool`	`i32`	0 or 1
`i8`, `u8`, `i16`, `u16`	`i32`	normalise after arithmetic, or before use
`i32`, `u32`	`i32`	signedness chooses the instructions
`i64`, `u64`	`i64`	BigInt at the JavaScript boundary
`i128`, `u128`	two `i64`s	multiplication and division need library code
`f16`	`f32`	round after each operation
`f32`, `f64`	`f32`, `f64`	
`char`	`i32`	a code point
pointers, `usize`	`i32`	`i64` with memory64
enums	`i32`	or `i64` if values need it

Choosing `i32` for pointers means your language's `usize` is 32 bits on this target, and code that assumes 64-bit pointers will break, just as it would on any 32-bit platform. For a game in a browser, the 4 GiB limit is rarely a problem, and 32-bit pointers make every pointer-heavy data structure smaller and faster.

Aggregates and unions 24.3

Structs, tuples and fixed-size arrays live in linear memory with whatever layout your language specifies, as Chapter 10 described, or are *scalarised* into separate locals when the compiler can see every use of them. A local struct whose address is never taken does not need memory at all: each field can be its own local.

Tagged unions (enums with data, `Option`, `Result`) are a tag followed by storage for the largest variant. In memory, that is the usual layout. As values, a small union can be represented as several scalars, with the tag in one and the payload in others, which suits multiple results. Here is an `Option<i32>`, returned without touching memory:

```
(module
  (memory 1)
  (data (i32.const 0) "\0a\00\00\00" "\14\00\00\00" "\1e\00\00\00"
      "\28\00\00\00")
  ;; fn index_of(x: i32) -> Option<i32>, searching a 4-element
     array at 0
  ;; returns (tag, value): tag 0 is None, 1 is Some
  (func (export "index_of") (param $x i32) (result i32 i32)
    (local $i i32)
    (block $not_found
      (loop $next
        (br_if $not_found (i32.ge_u (local.get $i) (i32.const 4)))
        (if (i32.eq (i32.load (i32.shl (local.get $i) (i32.const
            2))) (local.get $x))
          (then (return (i32.const 1) (local.get $i))))
        (local.set $i (i32.add (local.get $i) (i32.const 1)))
        (br $next)))
    (i32.const 0) (i32.const 0)))
```

called as `index_of(30)` and `index_of(99)`

```
1 2
0 0
```

A union whose variants hold different types, such as an `i64` in one and an `f64` in another, can share one `i64` slot, with `f64.reinterpret_i64` to read the float. For nullable pointers, use 0 as `None` and no tag at all, the same trick native compilers use, since address 0 is never a valid object in a sensible memory layout.

Calling convention 24.4

A calling convention decides how arguments and results are passed. With WebAssembly, most of it comes for free: parameters are parameters, and the engine assigns them to registers. What remains is deciding how to pass the things that are not single scalars:

- **Small aggregates** (up to four or so scalar fields) are passed as separate parameters, and returned as multiple results.
- **Large aggregates** are passed by pointer. To pass one by value, the caller copies it into its own shadow stack frame and passes the address. To return one, the caller passes the address where the callee should write it.
- **Slices and strings** are a pointer and a length, as two parameters.
- **Closures** take their environment as an extra first parameter (Chapter 11).
- **Methods** take the receiver as the first parameter.

- **Variadic functions** receive a pointer to a buffer in the caller's frame holding the extra arguments, which is how C's `printf` works on WebAssembly.

If your compiler ever needs to call C code compiled by Clang, or be called by it, follow the C ABI documented in the WebAssembly tool conventions repository, which pins down these choices for C types. Otherwise, the convention is entirely yours.

Generics 24.5

A generic function can be compiled once per instantiation (*monomorphisation*), or once for all instantiations, with values boxed and operations passed in as function pointers or dictionaries.

Monomorphisation produces the fastest code, and is what Rust, C++ and Zig do, but it multiplies code size, and on the web, code size is download size and compile time. A middle path compiles one copy per *representation* rather than per type: one for every type that is an `i32`, one for every `f64`, one for every pointer. Since WebAssembly only distinguishes a handful of value types, that bounds the number of copies of each generic function by a small constant.

Control flow and the call stack 24.6

Compile structured control flow straight from the syntax tree when you can, and use a stackifier when you have a control flow graph (Chapter 7). Keep every local whose address is never taken in a WebAssembly local, and everything else in a shadow stack frame (Chapter 10).

Three features need more than this, because they need to suspend a computation partway through and resume it later, and WebAssembly offers no way to capture its call stack:

- **Generators, iterators and `async` functions** can be compiled into state machines: each suspension point becomes a state, the function's live variables are saved in a heap object between calls, and resuming dispatches on the state (Chapter 7's dispatch loop). C#, Kotlin and JavaScript itself compile `async` this way, and it works well for functions that suspend only at points visible in their own body.
- **Suspending through ordinary calls**, as with green threads, fibers, or a synchronous API built on an asynchronous browser

call, needs the whole call stack captured. Binaryen's *Asyncify* transformation does it by rewriting every function on the path to save its locals to memory and unwind, then rewind on resumption. It works, at a cost in code size and speed of roughly half again or more.
- **JSPI** (JavaScript Promise Integration), standardised in the JS API and shipping in Chrome, lets an import return a promise and suspends the entire WebAssembly computation until it resolves, with no transformation. The broader *stack switching* proposal, adding first-class continuations, is the eventual general solution for fibers and effect handlers.

For a game engine, the common case is a script that wants to wait for the next frame or for an asset to load. State machines handle that well, and are the easiest to debug.

Runtime 24.7

Every language needs some runtime code, compiled into the module or provided by the host: an allocator, panic handling, string formatting, perhaps a garbage collector. On the web, everything in the module is downloaded and compiled, so the runtime's size matters. Write it in your own language, or in C compiled to WebAssembly and linked in, and keep it small enough that a "hello world" is a few kilobytes, not hundreds.

The host side of the runtime is JavaScript: the glue that provides imports and calls exports. Generate it from your compiler, so that it always matches the module.

A worked example 24.8

Here is a small program in an imaginary game scripting language:

```
struct Vec2 { x: f32, y: f32 }

fn length_sq(v: Vec2) -> f32 {
    return v.x * v.x + v.y * v.y
}

fn total_length_sq(points: []Vec2) -> f32 {
    var total: f32 = 0
    for p in points {
        total += length_sq(p)
    }
    return total
}
```

Applying the decisions above:

- `Vec2` has two `f32` fields, so it is passed flattened, as two `f32` parameters. In memory, it is 8 bytes: `x` at offset 0 and `y` at 4.
- `[]Vec2` is a slice, so a pointer and a count.
- `total` and `p` never have their addresses taken, so they are locals, and `p` is scalarised into its two fields.
- The `for` loop is a `block` and a `loop`, with the test at the top.

```
(module
  (import "env" "print_f32" (func $print (param f32)))
  (memory 1)
  ;; three points: (3, 4), (1, 1), (0, 2)
  (data (i32.const 64)
    "\00\00\40\40" "\00\00\80\40"
    "\00\00\80\3f" "\00\00\80\3f"
    "\00\00\00\00" "\00\00\00\40")

  (func $length_sq (param $v.x f32) (param $v.y f32) (result f32)
    (f32.add
      (f32.mul (local.get $v.x) (local.get $v.x))
      (f32.mul (local.get $v.y) (local.get $v.y))))

  (func $total_length_sq (param $points.ptr i32) (param
      $points.len i32) (result f32)
    (local $total f32)
    (local $end i32)
    (local.set $end (i32.add (local.get $points.ptr)
                             (i32.shl (local.get $points.len)
                                 (i32.const 3))))
    (block $break
      (loop $continue
        (br_if $break (i32.ge_u (local.get $points.ptr) (local.get
            $end)))
        (local.set $total (f32.add (local.get $total)
          (call $length_sq
            (f32.load offset=0 (local.get $points.ptr))
            (f32.load offset=4 (local.get $points.ptr)))))
        (local.set $points.ptr (i32.add (local.get $points.ptr)
            (i32.const 8)))
        (br $continue)))
    (local.get $total))

  (func (export "main")
    (call $print (call $total_length_sq (i32.const 64) (i32.const
        3)))))
```

```
31
```

Three things are worth noticing. The loop iterates with a pointer rather than an index, a classic strength reduction that saves a multiplication per iteration; engines will not do this for you in their baseline tier. The slice's `ptr` parameter is reused as the loop cursor, since parameters are just locals. And `$length_sq` is small enough that a compiler would probably inline it, which engines' optimising tiers also do, but only for hot code, and only after it has run unoptimised for a while.

Naming locals after the source variables, `$points.ptr` and `$v.x`, costs nothing, since names go in the name section, and makes the output far easier to read when debugging the compiler.

What you have seen 24.9

Mapping a language onto WebAssembly comes down to a series of decisions: where objects live, how scalars and aggregates are represented, how functions are called, how generics are instantiated, how control flow and suspension are compiled, and what runtime the module carries. For most of them, there is a well-trodden default. The next chapter looks at what the engine does with the result.

Chapter 25

Performance

A module's speed depends on two compilers: yours, and the engine's. This chapter explains what the engine's does, in V8 in particular, so that you know which optimisations to leave to it and which to do yourself. It then shows Binaryen's optimiser at work, covers code size and start-up time, and finishes with a list of practical rules.

How engines compile 25.1

Every major engine compiles WebAssembly in *tiers*:

Engine	First tier	Optimising tier
V8 (Chrome, Edge, Node)	Liftoff	TurboFan
SpiderMonkey (Firefox)	Baseline	Ion
JavaScriptCore (Safari)	an in-place interpreter, then BBQ	OMG
Wasmtime	Winch (optional)	Cranelift

The first tier exists to start running quickly. Liftoff, for instance, reads each function once, front to back, and emits machine code as it goes, tracking the operand stack in registers where it can and spilling where it cannot. It does no optimisation across instructions: no common subexpression elimination, no loop-invariant code motion, no inlining, and only simple register allocation. It compiles many times faster than the optimising tier, fast enough to keep up with a download, which is why large modules can start almost immediately.

As functions run, V8 counts how much time each one spends, and recompiles hot ones with TurboFan on a background thread, replacing the Liftoff code when it is ready. TurboFan is a real optimising compiler: it builds an SSA graph, inlines small functions (including, speculatively, the usual targets of `call_indirect` and `call_ref`), hoists loop invariants, eliminates redundant bounds checks for GC arrays, and allocates registers properly. Code that runs for more than a fraction of a second spends most of its life in TurboFan code.

Under the hood

> Tiering means a benchmark that runs a function a few times measures Liftoff, not TurboFan, and can be several times slower than steady state. When measuring your compiler's output, run the workload long enough to warm up, or force optimisation (Node and Chrome accept `--no-liftoff` to compile everything with TurboFan up front, and `--liftoff-only` for the opposite).

What the engine does for you 25.2

Because the optimising tier is a real compiler, a lot of classic backend work is done for you:

- **Instruction selection.** `(i32.add (local.get 0) (i32.const 1))` becomes one machine instruction. Addressing modes fold the offset immediate and even shifts into loads.
- **Register allocation.** Locals and operand stack slots become machine registers, or stack slots in the engine's own frame.
- **Bounds checks.** On 64-bit platforms, 32-bit memory accesses are checked by guard pages (Chapter 9), so they cost nothing.
- **Inlining**, of small and hot functions, though less aggressively than an ahead-of-time compiler, since time spent compiling is time the user waits.
- **Common subexpressions, constant folding, dead code**, within a function after inlining.

What it does not 25.3

And here is what you should do yourself:

- **Anything the first tier needs.** Code runs in Liftoff before it is hot, and short-lived code may never leave it. Start-up paths, such as level loading, benefit from code that is already reasonably good.
- **Inlining across a module, and specialisation.** Engines inline conservatively; your compiler knows far more about which calls matter.
- **High-level optimisations.** Escape analysis to keep values out of the shadow stack, scalar replacement of aggregates, removal of redundant bounds or null checks that your language inserted: the engine sees only the comparison and the branch, not what they mean.
- **Loop optimisations** such as strength reduction and vectorisation. Engines do not auto-vectorise.

- **Data layout.** Structure of arrays versus array of structures, field ordering, and alignment are decided by your compiler and nobody else.

Binaryen's optimiser 25.4

Many compilers do no WebAssembly-level optimisation of their own, and run Binaryen's `wasm-opt` over their output instead. It has a few hundred passes, written specifically for WebAssembly: local coalescing, redundant-set elimination, inlining, dead code and dead global removal, constant propagation, code folding, and a set of passes for WasmGC.

Here is code a naive code generator might produce, with a temporary local for every intermediate value and a helper that is called once:

```
(module
  (func $square (param $x i32) (result i32)
    (local $t i32)
    (local.set $t (i32.mul (local.get $x) (local.get $x)))
    (local.get $t))
  (func (export "f") (param $a i32) (param $b i32) (result i32)
    (local $t1 i32) (local $t2 i32) (local $t3 i32)
    (local.set $t1 (call $square (local.get $a)))
    (local.set $t2 (call $square (local.get $b)))
    (local.set $t3 (i32.add (local.get $t1) (local.get $t2)))
    (if (i32.const 0)
      (then (local.set $t3 (i32.const 99))))
    (local.get $t3)))
```

```
wasm-opt -O3 naive.wasm -o opt.wasm
wasm2wat opt.wasm
wc -c naive.wasm opt.wasm
```

```
(module
  (type (;0;) (func (param i32 i32) (result i32)))
  (func (;0;) (type 0) (param i32 i32) (result i32)
    local.get 0
    local.get 0
    i32.mul
    local.get 1
    local.get 1
    i32.mul
    i32.add)
  (export "f" (func 0)))
136 naive.wasm
 45 opt.wasm
181 total
```

The helper was inlined, the temporaries disappeared, the dead `if` was removed, and the name section was dropped (pass `-g` to keep it). Engines would have arrived at much the same machine code

from either version in their optimising tier, but the first tier, the download and the compile all benefit from the smaller one.

`wasm-opt -O3` optimises for speed, `-Oz` for size, and `-O2` and `-Os` are quicker versions of each. Running it costs build time, so a common setup is to skip it in development builds and run it for release.

> **For compiler writers**
>
> A code generator that emits simple, regular code and leaves cleanup to `wasm-opt` is a reasonable design, and it is what several production compilers do. The things `wasm-opt` cannot do are the ones that need your language's semantics, listed in section 25.3, so those are where your own optimisation effort pays off.

Code size and start-up 25.5

On the web, a module must be downloaded, then compiled, before it runs. Both scale with its size:

- **Download.** WebAssembly compresses well, typically to a third or less with Brotli or gzip, so serve it compressed. Size still matters on mobile connections.
- **Compilation.** With streaming compilation, Liftoff keeps up with download speeds, so for most modules the code is ready when the last byte arrives. Very large modules, tens of megabytes, can still take a noticeable time, and TurboFan's background work competes with the game for cores.
- **Caching.** Browsers cache the compiled code of modules loaded with `instantiateStreaming`, keyed by URL, so repeat visits skip most compilation. Changing the module's URL, or its bytes, invalidates the cache.

To keep modules small: avoid unbounded monomorphisation, strip the name section from release builds, prefer table-driven code to large unrolled code, make sure the runtime library is not dragging in unused features, and run `wasm-opt -Oz` when size matters more than the last few per cent of speed. `wasm-objdump -h` and Binaryen's `--func-metrics` show which sections and functions take the space.

A large program can also be split: a small module to start the game quickly, and further modules, linked as Chapter 8 described, loaded later for levels or features. Binaryen's `wasm-split` automates this for programs compiled as one module.

The cost of the boundary 25.6

Calls between WebAssembly and JavaScript used to be expensive, and the folklore persists. In current engines, a call in either direction with numeric arguments costs a few nanoseconds, and V8 inlines small JavaScript-to-WebAssembly calls in its JavaScript optimiser. What still costs:

- BigInt conversion for `i64` parameters, which allocates.
- Anything the JavaScript side does with the arguments, such as creating strings or typed arrays per call.
- The browser API on the other side, which is usually the real cost.

So the advice of Chapter 16 stands: prefer numeric parameters, batch small calls, and pass bulk data as views over memory.

Memory64 and other costs 25.7

A few features have costs worth knowing about:

- **memory64** needs an explicit bounds check on every access, since the guard page trick does not cover 64-bit addresses. Engines are working on reducing it, but expect a slowdown of a few per cent to tens of per cent on memory-heavy code.
- **Multiple memories** cost a little more per access to any memory but the first, since its base address may not be kept in a register.
- `call_indirect` costs a bounds check, a signature check and an indirect jump; `call_ref` skips the signature check. Hot indirect calls to the same target are often inlined speculatively.
- **GC allocation** is fast, a bump allocation in the engine's young generation, but every object has a header and costs collector work.

Practical rules 25.8

1. Keep values in locals. Spill to the shadow stack only what must have an address.
2. Use 32-bit memory and pointers unless you truly need more than 4 GiB.
3. Use multiple results to return small aggregates.
4. Use `memory.copy` and `memory.fill` rather than loops.

5. Give your language SIMD types, and lay hot data out as structures of arrays.
6. Batch boundary crossings, and pass bulk data as typed array views.
7. Emit the name section in development, strip it in release.
8. Run `wasm-opt` for release builds.
9. Measure in steady state, not in the first second.
0. Test the start-up path too, since it runs in the first tier.

What you have seen 25.9

Engines compile in tiers: a fast, simple first tier to start quickly, and an optimising tier for hot code. The optimising tier handles instruction selection, register allocation, bounds checks and some inlining, so your compiler should concentrate on what needs your language's semantics: escape analysis, check elimination, data layout and vectorisation. `wasm-opt` cleans up simple code generators' output. Code size affects download and compilation, and the boundary is cheaper than its reputation.

That ends Part VII. The last part is about making sure your compiler's output is correct.

PART VIII

Testing

A compiler that emits WebAssembly has more ways to be wrong than most programs: it can write malformed bytes, invalid code, valid code that computes the wrong thing, or correct code that one engine runs differently from another. The good news is that the WebAssembly ecosystem has excellent tools for catching each kind of mistake. This part covers the specification's own test format, then builds a layered testing strategy for a compiler on top of existing tools.

Chapter 26

The Spec's Test Format

The WebAssembly specification is tested with a few hundred files written in an extension of the text format, called WAST (WebAssembly Script), with the extension `.wast`. A script defines modules and makes assertions about them: that a call returns a certain value, that it traps, that a module fails to validate. Every engine runs these scripts, so the format is supported by many tools, and it is a convenient way to write tests of your own. This chapter covers the format, how to run it, and how a compiler can use it.

A first script 26.1

A script is a sequence of *commands*. `(module ...)` defines a module and makes it the current one. `assert_return` invokes an export of the current module with arguments, and checks the results:

```
(module
  (func (export "add") (param i32 i32) (result i32)
    (i32.add (local.get 0) (local.get 1)))
  (func (export "div_s") (param i32 i32) (result i32)
    (i32.div_s (local.get 0) (local.get 1))))

(assert_return (invoke "add" (i32.const 1) (i32.const 2))
    (i32.const 3))
(assert_return (invoke "add" (i32.const 0x7fffffff) (i32.const 1))
    (i32.const 0x80000000))
(assert_return (invoke "div_s" (i32.const -7) (i32.const 2))
    (i32.const -3))
(assert_trap (invoke "div_s" (i32.const 1) (i32.const 0)) "integer
    divide by zero")
(assert_trap (invoke "div_s" (i32.const 0x80000000) (i32.const
    -1)) "integer overflow")
```

```
main.wast: 6 passed, 0 failed
```

That ran with this book's runner, `scripts/wast.py`, described in section 26.4. The module counts as one passing command, so there are six.

Arguments and results are written as constant instructions, with the full literal syntax of the text format, so any value can be written exactly: hexadecimal integers, hexadecimal floats, `inf`, and NaNs with specific payloads.

The commands 26.2

Command	Checks
`(module ...)`	defines a module, which must validate and instantiate
`(module $name ...)`	the same, and names it for later commands
`(module binary "...")`	a module given as bytes, in string literals
`(module quote "...")`	a module given as text, in string literals, parsed only when the command runs
`(register "name" $m)`	makes a module's exports importable under `"name"`
`(invoke "f" args...)`	calls an export, ignoring results
`(assert_return (invoke ...) results...)`	the call returns these results
`(assert_trap (invoke ...) "msg")`	the call traps
`(assert_trap (module ...) "msg")`	instantiating the module traps
`(assert_exhaustion (invoke ...) "msg")`	the call overflows the stack
`(assert_invalid (module ...) "msg")`	the module fails validation
`(assert_malformed (module ...) "msg")`	the module fails to parse or decode
`(assert_unlinkable (module ...) "msg")`	the module's imports cannot be satisfied

The message strings are what the reference interpreter says. Other runners compare them loosely or not at all, since every engine words its errors differently; what is checked is the *kind* of failure.

Results can also be patterns. `nan:canonical` matches a NaN with only the quiet bit set in its payload, which is what arithmetic on non-NaN inputs must produce; `nan:arithmetic` matches any NaN with the quiet bit set, which is what arithmetic on NaN inputs may produce:

```
(module
  (func (export "div") (param f32 f32) (result f32)
    (f32.div (local.get 0) (local.get 1)))
  (func (export "add") (param f64 f64) (result f64)
    (f64.add (local.get 0) (local.get 1))))

(assert_return (invoke "div" (f32.const 1) (f32.const 0))
    (f32.const inf))
(assert_return (invoke "div" (f32.const 0) (f32.const 0))
    (f32.const nan:canonical))
(assert_return (invoke "add" (f64.const nan:0x4) (f64.const 1))
    (f64.const nan:arithmetic))
(assert_return (invoke "add" (f64.const 0x1p-1074) (f64.const
    0x1p-1074)) (f64.const 0x1p-1073))
```

```
main.wast: 5 passed, 0 failed
```

The last line adds the smallest denormal to itself, which a runner that flushed denormals to zero would get wrong.

Modules can be linked through `register`, which is how the spec tests imports:

```
(module $math
  (func (export "square") (param i32) (result i32)
    (i32.mul (local.get 0) (local.get 0))))
(register "math" $math)

(module
  (import "math" "square" (func $square (param i32) (result i32)))
  (func (export "sum_of_squares") (param i32 i32) (result i32)
    (i32.add (call $square (local.get 0)) (call $square (local.get
        1)))))

(assert_return (invoke "sum_of_squares" (i32.const 3) (i32.const
    4)) (i32.const 25))
(assert_return (invoke $math "square" (i32.const 9)) (i32.const
    81))
(assert_unlinkable
  (module (import "math" "cube" (func (param i32) (result i32))))
  "unknown import")
```

```
main.wast: 6 passed, 0 failed
```

And validation is tested with modules that must be rejected:

```
(assert_invalid
  (module (func (result i32) (i64.const 1)))
  "type mismatch")
(assert_invalid
  (module (func (local.get 0)))
  "unknown local")
(assert_malformed
  (module quote "(func (i32.add_with_typo))")
  "unknown operator")
(assert_malformed
  (module binary "\00asm" "\02\00\00\00")
  "unknown binary version")
```

```
main.wast: 4 passed, 0 failed
```

When an assertion fails, the runner says where and why:

```
(module
  (func (export "abs") (param i32) (result i32)
    ;; a buggy abs: forgets to negate
    (local.get 0)))
(assert_return (invoke "abs" (i32.const 5)) (i32.const 5))
(assert_return (invoke "abs" (i32.const -5)) (i32.const 5))
```

```
main.wast:6: "abs" returned -5, expected (i32.const 5)
main.wast: 2 passed, 1 failed
```

Running scripts 26.3

Several tools run `.wast` files:

Tool	Runs scripts with
the spec's reference interpreter	`wasm script.wast`, the definitive implementation
Wasmtime	`wasmtime wast script.wast`
wasm-tools	`wasm-tools wast script.wast`, which only parses and validates
wabt	`wast2json` to split a script into modules and a JSON command list, then `spectest-interp` to run it
this book	`python3 scripts/wast.py script.wast`, in Node's V8

Browsers' engines run the spec tests too, through harnesses that convert each script into JavaScript. That is the approach this book's runner takes, for the same reason: to test in the engine the code will really run in.

How the book's runner works 26.4

The runner is about four hundred lines in two files, and is worth understanding because it shows how to build test infrastructure on existing tools rather than from scratch.

Splitting. `scripts/wast.py` parses the script into S-expressions, keeping each one's position in the source. It does not understand the text format at all, only parentheses, strings and comments.

Assembling. Each `(module ...)` form is cut out of the source as text, and given to an assembler, here the `wat` crate's by way of Wasmtime's Python package. So every literal, every instruction

and every abbreviation is parsed by a real, complete assembler, not by the runner.

Checker modules. The difficult part of a WAST runner is comparing results exactly. The arguments and expected results are text-format constants, which a runner would have to parse; and values like `i64`s, NaN payloads and `v128`s cannot cross into JavaScript intact. The runner avoids both problems by turning each `assert_return` into a small module of its own, in the text format:

```
(module
  (import "target" "add" (func $f (param i32 i32) (result i32)))
  (global $r0 (export "r0") (mut i32) (i32.const 0))
  (func (export "run") (result i32)
    (call $f (i32.const 1) (i32.const 2))
    (global.set $r0)
    (i32.const 1) (i32.eq (global.get $r0) (i32.const 3))
        (i32.and)))
```

The checker imports the function under test, calls it with the assertion's arguments, and compares each result with the expected value, inside WebAssembly, returning 1 if they all match. The assembler parses the constants, and the comparison is done by WebAssembly instructions: `i64.eq` for integers, `i32.reinterpret_f32` and an integer comparison for exact float bits, bit masks for the NaN patterns, `i8x16.eq` and `i8x16.all_true` for vectors. The results are also stored in exported globals, so that when a check fails, the runner can read them and report what the function actually returned.

Executing. The assembled modules, checkers included, are written into a JSON plan, which `scripts/wastrun.mjs` executes in Node: it instantiates each module, links the checkers to their targets, calls them, and reports.

The lesson generalises. To check something precisely, express the check in the system under test, and let a mature tool parse the inputs.

Using WAST in a compiler's tests 26.5

The spec test suite tests engines, not compilers, but the format is useful to a compiler in three ways.

Testing the runtime library. Parts of a language runtime are often written directly in the text format, or are small enough to test in isolation: memory allocation, string routines, number formatting.

A WAST file can test them thoroughly, including traps and edge cases.

Testing generated modules. A compiler can write a test as a WAST file with its own output embedded, using `(module binary ...)` with the module's bytes, followed by assertions about its exports. Assertions written by hand about a compiled module check the compiler end to end, at the level of WebAssembly values rather than printed output:

```
import subprocess, wasmtime

# stand-in for your compiler: the "output" here comes from the
    text format
wasm = wasmtime.wat2wasm('''(module (func (export "fib") (param
    i32) (result i32)
  (if (result i32) (i32.lt_u (local.get 0) (i32.const 2))
    (then (local.get 0))
    (else (i32.add
      (call 0 (i32.sub (local.get 0) (i32.const 1)))
      (call 0 (i32.sub (local.get 0) (i32.const 2)))))))''')

escaped = "".join(f"\\{b:02x}" for b in wasm)
script = f'''(module binary "{escaped}")
(assert_return (invoke "fib" (i32.const 0)) (i32.const 0))
(assert_return (invoke "fib" (i32.const 1)) (i32.const 1))
(assert_return (invoke "fib" (i32.const 20)) (i32.const 6765))
'''
open("fib.wast", "w").write(script)
print(subprocess.run(["wast.py", "fib.wast"], capture_output=True,
    text=True).stdout, end="")
```

```
fib.wast: 4 passed, 0 failed
```

Testing a decoder or validator. If your toolchain reads WebAssembly as well as writing it, as a linker, an optimiser or a debugger does, the spec test suite is the most thorough test you will find: its `assert_malformed` and `assert_invalid` cases cover every corner of the binary format and the validation rules. The suite lives in the `WebAssembly/testsuite` repository, with a directory for each proposal.

What you have seen 26.6

WAST scripts define modules and assert facts about them: return values, traps, stack exhaustion, invalid and malformed modules, link errors. Results can be exact, including NaN payloads, or patterns. Many tools run them, and a runner can be built on an existing assembler by turning each assertion into a checker module. A compiler can use WAST to test its runtime library and its generated modules, and the spec test suite to test anything that reads WebAssembly.

The next chapter builds a complete testing strategy for a compiler.

Chapter 27

Testing a Compiler

A compiler is tested at several levels, each catching a different kind of mistake. For a compiler that emits WebAssembly, the levels are unusually well supported by existing tools: a validator to check every module it writes, a disassembler to compare output with expectations, several independent engines to compare results, and generators of random modules for fuzzing. This chapter builds a testing strategy layer by layer, with working examples of each, and finishes with this book's own test harness as a case study.

The layers 27.1

Layer	Catches	Tools
validation	malformed bytes, invalid code	V8, Wasmtime, `wasm-validate`
golden output	unintended changes in generated code	`wasm2wat`, `wasm-objdump`, `diff`
execution	wrong results	Node, Wasmtime, your own expected outputs
differential	engine-dependent behaviour, and wrong results nobody wrote a test for	several engines, a reference interpreter
fuzzing	everything the other layers missed	random programs, `wasm-opt --fuzz-exec`, `wasm-tools smith`
browser	integration with the real host	Playwright

Each layer is cheap to add once the previous one exists, and together they catch nearly everything.

Layer 1: validate everything 27.2

The first rule is that *every* module your compiler writes, in every test, is validated. Validation is fast, and an invalid module is always a compiler bug. The error message also points much closer to the cause than a wrong answer would.

Validate with more than one validator. They should agree on everything standard, and when they disagree, you have found either a bug in an engine (rare, but it happens), or a feature that

one of them has not enabled. Here is a module checked by three validators, and an invalid one:

```
(module
  (func (export "f") (param i32) (result i32)
    (i32.mul (local.get 0) (i32.const 3))))
```

```
(module
  (func (export "f") (param i32) (result i32)
    (i32.mul (local.get 0) (f32.const 3))))
```

```
import re, subprocess, wasmtime

def v8(path):
    js = ("try { new
        WebAssembly.Module(require('fs').readFileSync(process.argv
        [1])) }"
          "catch (e) { console.log(e.message); process.exit(1) }")
    p = subprocess.run(["node", "-e", js, path],
        capture_output=True, text=True)
    return p.returncode == 0, p.stdout.strip()

def wasmtime_(path):
    try:
        wasmtime.Module.validate(wasmtime.Engine(), open(path,
            "rb").read())
        return True, ""
    except wasmtime.WasmtimeError as e:
        return False, str(e).splitlines()[-1].strip()

def wabt(path):
    p = subprocess.run(["wasm-validate", path],
        capture_output=True, text=True)
    return p.returncode == 0, re.sub(r"\x1b\[[0-9;]*m", "",
        p.stderr.strip())   # drop colours

def validate(path):
    results = {name: f(path) for name, f in [("v8", v8),
        ("wasmtime", wasmtime_), ("wabt", wabt)]}
    verdicts = {ok for ok, _ in results.values()}
    if len(verdicts) > 1:
        print(f"{path}: VALIDATORS DISAGREE")
    for name, (ok, msg) in results.items():
        print(f"{path}: {name:8} {'valid' if ok else 'invalid: ' +
            msg}")
```

```
validate("good.wasm")
validate("bad.wasm")
```

```
good.wasm: v8       valid
good.wasm: wasmtime valid
good.wasm: wabt     valid
bad.wasm: v8       invalid: WebAssembly.Module(): Compiling
    function #0 failed: i32.mul[1] expected type i32, found
    f32.const of type f32 @+34
bad.wasm: wasmtime invalid: type mismatch: expected i32, found f32
    (at offset 0x27)
bad.wasm: wabt     invalid: bad.wasm:0000028: error: type mismatch
    in i32.mul, expected [i32, i32] but got [i32, f32]
```

Each validator explains the problem differently, and between them the cause is clear. Each also reports a different byte offset for the same mistake (0x22, 0x27 and 0x28), so tests that check error messages should not expect offsets to agree across tools.

For a compiler, the best place to validate is inside the compiler itself, in debug builds, right after encoding. Binaryen has a built-in validator for this; with your own encoder, the operand-stack tracking described in Chapter 13 catches problems at the instruction that caused them.

Layer 2: golden output 27.3

A *golden test* compares the compiler's output with a saved copy of what it produced before. It catches changes nobody intended, such as an optimisation that stopped firing, or a change in calling convention that was meant to affect one function and affected all of them.

Compare the text format, not the bytes. A diff of the text is readable; a diff of bytes is not. Here is a golden test, with a stand-in for a compiler's output:

```
(module
  (func (export "double") (param i32) (result i32)
    (i32.shl (local.get 0) (i32.const 1))))
```

```
cat > double.golden.wat <<'EOF'
(module
  (type (;0;) (func (param i32) (result i32)))
  (func (;0;) (type 0) (param i32) (result i32)
    local.get 0
    i32.const 1
    i32.shl)
  (export "double" (func 0)))
EOF
wasm2wat out.wasm > double.actual.wat
diff -u double.golden.wat double.actual.wat && echo "golden test
    passed"
```

```
golden test passed
```

Two practical points. First, give the test runner a flag that overwrites the golden files with the current output, so that intended changes are a single command followed by reviewing the diff in version control. (This book's harness has one, `--accept`.) Second, keep golden tests few and focused: one per code

generation pattern, not one per program, or every change will touch hundreds of files and reviewers will stop reading them.

Layer 3: execution tests 27.4

The core of a compiler's test suite is programs in the source language together with their expected behaviour: the output they print, the value they return, or the error they report. These test what the user cares about, independent of how the code is generated, so they survive refactoring of the backend.

A good format keeps the expectation next to the program, in the same file, in comments:

```
// expect: 55
// expect: done
fn main() {
    var total = 0
    for i in 1..=10 { total += i }
    print(total)
    print("done")
}
```

The runner compiles each file, runs it in Node with a small host like this book's, and compares the output with the `expect` lines. Programs that should fail to compile carry `expect-error` lines, and programs that should trap carry `expect-trap` lines. Hundreds of such tests run in seconds, since compiling and instantiating small modules is fast.

Run each test under every configuration you ship: with and without optimisation, with and without SIMD or other optional features. A bug that appears only in one configuration is common, and easy to diagnose once you know which.

Layer 4: differential testing 27.5

Execution tests check what someone thought to check. *Differential testing* runs the same program in two different ways and checks that the results agree, so it finds bugs nobody wrote a test for.

The first kind compares engines. The same module, run in V8, in Wasmtime and in wabt's interpreter, should give the same results, except where WebAssembly allows differences (NaN bit patterns, relaxed SIMD, stack depth). A disagreement means your module depends on something it should not, or, occasionally, that an engine has a bug:

```
(module
  (func (export "f") (param i32) (result i32)
    (i32.add
      (i32.mul (local.get 0) (local.get 0))
      (i32.rotl (local.get 0) (i32.const 7)))))
```

```
import subprocess, wasmtime

def in_wasmtime(x):
    store = wasmtime.Store()
    module = wasmtime.Module.from_file(store.engine, "m.wasm")
    return wasmtime.Instance(store, module, []).exports(store)
        ["f"](store, x)

def in_v8(x):
    out = subprocess.run(["node", "-e",
        "const m = new
            WebAssembly.Module(require('fs').readFileSync('m.wasm'
            ));"
        f"console.log(new
            WebAssembly.Instance(m).exports.f({x}))"],
            capture_output=True, text=True)
    return int(out.stdout)

def in_wabt(x):
    # wasm-interp takes and prints unsigned values:
        f(i32:4294967295) => i32:0
    out = subprocess.run(["wasm-interp", "m.wasm", "-r", "f", "-
        a", f"i32:{x % 2**32}"],
                         capture_output=True, text=True)
    return int(out.stdout.split("=> i32:")[1])

for x in (0, 1, -1, 123456789, -2147483648):
    results = {"wasmtime": in_wasmtime(x), "v8": in_v8(x), "wabt":
        in_wabt(x)}
    # bring everything to signed i32 before comparing
    results = {k: (v + 2**31) % 2**32 - 2**31 for k, v in
        results.items()}
    agree = len(set(results.values())) == 1
    print(f"f({x}) = {results['v8']}", "(all agree)" if agree else
        f"DISAGREE: {results}")
```

```
f(0) = 0 (all agree)
f(1) = 129 (all agree)
f(-1) = 0 (all agree)
f(123456789) = 1159671356 (all agree)
f(-2147483648) = 64 (all agree)
```

The handling of wabt's unsigned numbers is typical of real differential testing: most of the work is normalising each engine's way of taking arguments and reporting results, and the first "disagreements" you find are usually in that normalisation.

The second, more powerful kind compares your compiler with an *independent implementation of your language*: an interpreter, written as simply as possible, directly from the language's semantics. Any program on which the compiled code and the interpreter disagree has found a bug in one of them. A simple interpreter is worth writing for this alone.

Layer 5: fuzzing 27.6

Differential testing becomes fuzzing when the programs are generated at random. A generator produces programs in your language, the compiler and the reference interpreter run each one, and any disagreement is saved as a failing test. Run long enough, this finds the bugs in the corners that neither you nor your users would have tested.

Here is a complete small fuzzer. It generates random 32-bit integer expressions over one variable, "compiles" them to WebAssembly by printing the text format, runs them in V8, and compares the results with a Python evaluator that implements WebAssembly's integer semantics directly. The compiler takes a flag that introduces a bug, to show the fuzzer finding it:

```
import random, subprocess, json, wasmtime

OPS = ["add", "sub", "mul", "and", "or", "xor", "shl", "shr_s",
    "shr_u", "div_s", "rem_u"]

def gen(rng, depth):
    """A random expression: ("x",), ("const", n) or (op, a, b)."""
    if depth == 0 or rng.random() < 0.3:
        return ("x",) if rng.random() < 0.5 else ("const",
            rng.choice([0, 1, -1, 7, 31, 2**31 - 1, -2**31,
            rng.randrange(-2**31, 2**31)]))
    return (rng.choice(OPS), gen(rng, depth - 1), gen(rng, depth -
        1))

def s32(n):
    return (n + 2**31) % 2**32 - 2**31

class Trap(Exception):
    pass

def evaluate(e, x):
    """The reference: WebAssembly's i32 semantics, written out in
        Python."""
    if e[0] == "x": return x
    if e[0] == "const": return e[1]
    op, a, b = e[0], evaluate(e[1], x), evaluate(e[2], x)
    u = lambda n: n % 2**32
    if op == "add": return s32(a + b)
    if op == "sub": return s32(a - b)
    if op == "mul": return s32(a * b)
    if op == "and": return s32(a & b)
    if op == "or": return s32(a | b)
    if op == "xor": return s32(a ^ b)
    if op == "shl": return s32(a << (b % 32))
    if op == "shr_s": return a >> (b % 32)
    if op == "shr_u": return s32(u(a) >> (b % 32))
    if op == "div_s":
        if b == 0 or (a == -2**31 and b == -1): raise Trap()
        q = abs(a) // abs(b)
        return s32(q if (a < 0) == (b < 0) else -q)
    if op == "rem_u":
        if b == 0: raise Trap()
        return s32(u(a) % u(b))

def compile_expr(e, buggy=False):
    """The 'compiler': an expression to the text format."""
    if e[0] == "x": return "(local.get 0)"
    if e[0] == "const": return f"(i32.const {e[1]})"
    op = e[0]
    if buggy and op == "shr_s":
        op = "shr_u"                                # the
            deliberate bug
    return f"(i32.{op} {compile_expr(e[1], buggy)}
        {compile_expr(e[2], buggy)})"

def run_in_v8(exprs, inputs, buggy):
    funcs = "".join(f'(func (export "f{i}") (param i32) (result
        i32) {compile_expr(e, buggy)})\n'
                    for i, e in enumerate(exprs))
    open("fuzz.wasm", "wb").write(wasmtime.wat2wasm(f"(module
        {funcs})"))
    js = f"""
      const {{ exports }} = new WebAssembly.Instance(new
          WebAssembly.Module(require('fs').readFileSync('fuzz.wasm
          ')));
      const out = [];
      for (let i = 0; i < {len(exprs)}; i++)
```

```
        for (const x of {json.dumps(inputs)}) {{
          try {{ out.push(exports['f' + i](x)); }} catch (e) {{
              out.push('trap'); }}
        }}
      console.log(JSON.stringify(out));"""
    return json.loads(subprocess.run(["node", "-e", js],
        capture_output=True, text=True).stdout)

def fuzz(seed, count, buggy=False):
    rng = random.Random(seed)
    exprs = [gen(rng, 4) for _ in range(count)]
    inputs = [0, 1, -1, 5, -123456, 2**31 - 1, -2**31]
    got = iter(run_in_v8(exprs, inputs, buggy))
    failures = []
    for e in exprs:
        for x in inputs:
            try:
                want = evaluate(e, x)
            except Trap:
                want = "trap"
            if next(got) != want:
                failures.append((e, x, want))
    return exprs, inputs, failures
```

```
exprs, inputs, failures = fuzz(seed=1, count=300)
print(f"{len(exprs)} expressions, {len(exprs) * len(inputs)} runs,
    {len(failures)} disagreements")

exprs, inputs, failures = fuzz(seed=1, count=300, buggy=True)
print(f"with the bug: {len(failures)} disagreements")
smallest = min(failures, key=lambda f: len(compile_expr(f[0])))
print("smallest:", compile_expr(smallest[0]), "with x =",
    smallest[1])
```

```
300 expressions, 2100 runs, 0 disagreements
with the bug: 55 disagreements
smallest: (i32.shr_s (local.get 0) (local.get 0)) with x = -1
```

The correct compiler agrees with the reference on all 2,100 runs. The buggy one is caught at once, and choosing the smallest failing expression points straight at the bug. Real fuzzers *reduce* failures automatically: repeatedly simplify the failing program, keeping each simplification that still fails, until nothing more can be removed. Binaryen's `wasm-reduce` does this for WebAssembly modules, and C-Reduce and its successors do it for source programs.

Several existing tools generate WebAssembly to fuzz with:

- `wasm-tools smith` generates random valid modules, with control over which features they use. It is the standard way to fuzz anything that *reads* WebAssembly, and Wasmtime is fuzzed with it continuously.
- **Binaryen's fuzzer**, `wasm-opt -ttf` (translate to fuzz), turns random bytes into a valid module, and `wasm-opt --fuzz-exec` runs a module's exports before and after optimisation and

compares the results. Running it over your compiler's output tests Binaryen's optimiser on your code patterns, which matters if you rely on it.

- **Generators for your language** are the most valuable and the only ones you have to write yourself. Start with expressions, as above, then add statements, functions and data structures. Generating only well-typed programs is much easier than it sounds if the generator builds a program of a chosen type from the top down.

Layer 6: the browser 27.7

Everything above runs in Node, which uses the same V8 as Chrome. But users also run Firefox and Safari, and the host side of your runtime uses browser APIs that Node lacks. Playwright runs tests in all three browsers' engines, headless, from a script:

```
import { chromium, firefox, webkit } from 'playwright';

for (const browserType of [chromium, firefox, webkit]) {
  const browser = await browserType.launch();
  const page = await browser.newPage();
  await page.goto('http://localhost:8000/tests/runner.html');
  const result = await page.evaluate(() => window.testResults);
      // set by the page
  console.log(browserType.name(), result.passed, 'passed',
      result.failed, 'failed');
  await browser.close();
}
```

Keep the browser layer thin: a page that loads a batch of compiled test modules and reports their output, run against the same expected outputs as layer 3. The point is to catch the differences between engines, not to run the whole suite three more times.

Performance tests 27.8

Speed regressions are bugs too, but timings are noisy, especially on shared CI machines. Track deterministic proxies instead, and time things only occasionally:

- **Code size**, per test program and for a few large ones, is exactly reproducible and correlates with both start-up time and, roughly, speed.
- **Instruction counts** from an instrumented build, or wabt's `wasm-interp` with tracing, are reproducible measures of work done.

- **Specific code patterns**, checked by golden tests, catch the loss of an important optimisation directly.

For real timings, run benchmarks in steady state (Chapter 25), many times, on a quiet machine, and compare distributions rather than single numbers.

A case study: this book 27.9

Every example in this book was checked by a harness built on these ideas, in about five hundred lines of Python and JavaScript, in the `scripts` directory beside the chapters:

- `book.py` reads the chapters and collects every fenced example, with the fence's info string saying how to check it: `wat run`, `wat error`, `wat trap`, `sh run`, `python emit`, `wast test`, and so on. Each example's expected output or error is the fence that follows it. That is layer 3, with the book as the test file.
- Every module is assembled with the `wat` crate and validated by V8 before it runs, and `wat error` examples name the validator whose message they expect. That is layer 1.
- `run.mjs` is the host, a dozen imports for printing, so that every example's output is plain text that can be compared exactly.
- `wast.py` and `wastrun.mjs` run WAST scripts in V8, through checker modules, as Chapter 26 described.
- `book.py test --accept` rewrites the expected output of every example whose output changed, for review in version control, the golden-file workflow of layer 2.
- The tools are pinned versions from npm and PyPI, so the book's checks give the same answers on any machine, today and later.

The book's harness found real mistakes while it was being written: wrong byte counts, wrong offsets in disassembly, misremembered error messages, and more than one example whose output was simply not what the text said. Each was caught because the claim was executable. The same is true of a compiler: every expectation that can be checked by a machine should be.

What you have seen 27.10

A compiler's tests come in layers. Validate every module with several validators. Compare generated code with golden files in the text format. Run programs and compare their output with

expectations stored beside them, in every configuration. Compare engines with each other, and your compiler with a simple interpreter. Fuzz with random programs, and reduce the failures. Run a thin layer of tests in real browsers. Each layer is built from existing tools, and together they make a compiler you can change with confidence.

That is the end of the book. The appendices follow: an instruction reference, a guide to the tools, and suggestions for further reading.

Appendix A: Instruction Reference

This appendix lists the instructions with their opcodes and their types. It covers every instruction in the core of WebAssembly 3.0 except the SIMD and atomic instructions, of which it shows a representative selection; the full lists are in the specification and the threads proposal.

The opcodes were not typed in by hand. `scripts/opcodes.py` assembles each instruction with the `wat` crate and reads its opcode back out of the binary, so the table agrees with the assembler. Prefixed opcodes show the prefix byte and the second opcode as it is encoded, in LEB128, so `fd 85 02` is prefix `fd` followed by opcode 261.

Types are written as in the specification: what the instruction pops on the left, what it pushes on the right, with `t` standing for any value type and `t*` for any sequence of them. Instructions that end control flow, such as `br` and `unreachable`, have types with arbitrary `t1*` and `t2*`, since the stack after them is polymorphic (Chapter 6). The immediates column lists what follows the opcode in the binary, in order; a *memarg* is an alignment exponent and an offset (Chapter 13), and the memory instructions without a memarg take a memory index, which is `00` for memory 0.

Control A.1

Instruction	Opcode	Type	Immediates and notes
`unreachable`	`00`	`[t*] → [t*]`	trap
`nop`	`01`	`[] → []`	
`block bt`	`02`	`[t1*] → [t2*]`	block type; ends with `end`
`loop bt`	`03`	`[t1*] → [t2*]`	block type
`if bt`	`04`	`[t1* i32] → [t2*]`	block type
`else`	`05`		between the arms of an `if`
`end`	`0b`		ends a block, loop, if, try_table or function
`br l`	`0c`	`[t* t1*] → [t2*]`	label depth
`br_if l`	`0d`	`[t* i32] → [t*]`	label depth
`br_table l* l`	`0e`	`[t1* t* i32] → [t2*]`	vector of depths, then default
`return`	`0f`	`[t1* t*] → [t2*]`	
`call f`	`10`	`[t1*] → [t2*]`	function index
`call_indirect t`	`11`	`[t1* i32] → [t2*]`	type index, table index
`return_call f`	`12`	`[t1*] → [t2*]`	tail call
`return_call_indirect t`	`13`	`[t1* i32] → [t2*]`	tail call
`call_ref t`	`14`	`[t1* (ref null $t)] → [t2*]`	type index
`return_call_ref t`	`15`	`[t1* (ref null $t)] → [t2*]`	type index
`throw x`	`08`	`[t1* t*] → [t2*]`	tag index
`throw_ref`	`0a`	`[t1* exnref] → [t2*]`	

Instruction	Opcode	Type	Immediates and notes
`try_table bt c*`	`1f`	`[t1*] → [t2*]`	block type, vector of catch clauses

Parametric and variables A.2

Instruction	Opcode	Type	Immediates and notes
`drop`	`1a`	`[t] → []`	
`select`	`1b`	`[t t i32] → [t]`	numeric or vector operands
`select t`	`1c`	`[t t i32] → [t]`	typed form: vector of value types
`local.get x`	`20`	`[] → [t]`	local index
`local.set x`	`21`	`[t] → []`	local index
`local.tee x`	`22`	`[t] → [t]`	local index
`global.get x`	`23`	`[] → [t]`	global index
`global.set x`	`24`	`[t] → []`	global index

Tables A.3

Instruction	Opcode	Type	Immediates and notes
`table.get x`	`25`	`[i32] → [ref]`	table index
`table.set x`	`26`	`[i32 ref] → []`	table index
`table.init`	`fc 0c`	`[i32 i32 i32] → []`	element segment, table
`elem.drop`	`fc 0d`	`[] → []`	element segment
`table.copy`	`fc 0e`	`[i32 i32 i32] → []`	destination and source tables
`table.grow x`	`fc 0f`	`[ref i32] → [i32]`	table index
`table.size x`	`fc 10`	`[] → [i32]`	table index
`table.fill x`	`fc 11`	`[i32 ref i32] → []`	table index

Instruction	Opcode	Type	Immediates and notes
`i32.load`	`28`	`[i32] → [i32]`	memarg
`i64.load`	`29`	`[i32] → [i64]`	memarg
`f32.load`	`2a`	`[i32] → [f32]`	memarg
`f64.load`	`2b`	`[i32] → [f64]`	memarg
`i32.load8_s`	`2c`	`[i32] → [i32]`	memarg
`i32.load8_u`	`2d`	`[i32] → [i32]`	memarg
`i32.load16_s`	`2e`	`[i32] → [i32]`	memarg
`i32.load16_u`	`2f`	`[i32] → [i32]`	memarg
`i64.load8_s`	`30`	`[i32] → [i64]`	memarg
`i64.load8_u`	`31`	`[i32] → [i64]`	memarg
`i64.load16_s`	`32`	`[i32] → [i64]`	memarg
`i64.load16_u`	`33`	`[i32] → [i64]`	memarg
`i64.load32_s`	`34`	`[i32] → [i64]`	memarg
`i64.load32_u`	`35`	`[i32] → [i64]`	memarg
`i32.store`	`36`	`[i32 i32] → []`	memarg
`i64.store`	`37`	`[i32 i64] → []`	memarg
`f32.store`	`38`	`[i32 f32] → []`	memarg
`f64.store`	`39`	`[i32 f64] → []`	memarg
`i32.store8`	`3a`	`[i32 i32] → []`	memarg
`i32.store16`	`3b`	`[i32 i32] → []`	memarg
`i64.store8`	`3c`	`[i32 i64] → []`	memarg
`i64.store16`	`3d`	`[i32 i64] → []`	memarg
`i64.store32`	`3e`	`[i32 i64] → []`	memarg
`memory.size`	`3f`	`[] → [i32]`	memory index
`memory.grow`	`40`	`[i32] → [i32]`	memory index
`memory.init`	`fc 08`	`[i32 i32 i32] → []`	data segment, memory
`data.drop`	`fc 09`	`[] → []`	data segment
`memory.copy`	`fc 0a`	`[i32 i32 i32] → []`	destination and source memories
`memory.fill`	`fc 0b`	`[i32 i32 i32] → []`	memory index

Constants A.5

Instruction	Opcode	Type	Immediates and notes
`i32.const`	`41`	`[] → [i32]`	signed LEB128
`i64.const`	`42`	`[] → [i64]`	signed LEB128
`f32.const`	`43`	`[] → [f32]`	4 bytes
`f64.const`	`44`	`[] → [f64]`	8 bytes

i32 operations A.6

Instruction	Opcode	Type	Immediates and notes
i32.eqz	45	[i32] → [i32]	
i32.eq	46	[i32 i32] → [i32]	
i32.ne	47	[i32 i32] → [i32]	
i32.lt_s	48	[i32 i32] → [i32]	
i32.lt_u	49	[i32 i32] → [i32]	
i32.gt_s	4a	[i32 i32] → [i32]	
i32.gt_u	4b	[i32 i32] → [i32]	
i32.le_s	4c	[i32 i32] → [i32]	
i32.le_u	4d	[i32 i32] → [i32]	
i32.ge_s	4e	[i32 i32] → [i32]	
i32.ge_u	4f	[i32 i32] → [i32]	
i32.clz	67	[i32] → [i32]	
i32.ctz	68	[i32] → [i32]	
i32.popcnt	69	[i32] → [i32]	
i32.add	6a	[i32 i32] → [i32]	
i32.sub	6b	[i32 i32] → [i32]	
i32.mul	6c	[i32 i32] → [i32]	
i32.div_s	6d	[i32 i32] → [i32]	traps on zero
i32.div_u	6e	[i32 i32] → [i32]	traps on zero
i32.rem_s	6f	[i32 i32] → [i32]	traps on zero
i32.rem_u	70	[i32 i32] → [i32]	traps on zero
i32.and	71	[i32 i32] → [i32]	
i32.or	72	[i32 i32] → [i32]	
i32.xor	73	[i32 i32] → [i32]	
i32.shl	74	[i32 i32] → [i32]	
i32.shr_s	75	[i32 i32] → [i32]	
i32.shr_u	76	[i32 i32] → [i32]	
i32.rotl	77	[i32 i32] → [i32]	
i32.rotr	78	[i32 i32] → [i32]	

i64 operations

A.7

Instruction	Opcode	Type	Immediates and notes
`i64.eqz`	`50`	`[i64] → [i32]`	
`i64.eq`	`51`	`[i64 i64] → [i32]`	
`i64.ne`	`52`	`[i64 i64] → [i32]`	
`i64.lt_s`	`53`	`[i64 i64] → [i32]`	
`i64.lt_u`	`54`	`[i64 i64] → [i32]`	
`i64.gt_s`	`55`	`[i64 i64] → [i32]`	
`i64.gt_u`	`56`	`[i64 i64] → [i32]`	
`i64.le_s`	`57`	`[i64 i64] → [i32]`	
`i64.le_u`	`58`	`[i64 i64] → [i32]`	
`i64.ge_s`	`59`	`[i64 i64] → [i32]`	
`i64.ge_u`	`5a`	`[i64 i64] → [i32]`	
`i64.clz`	`79`	`[i64] → [i64]`	
`i64.ctz`	`7a`	`[i64] → [i64]`	
`i64.popcnt`	`7b`	`[i64] → [i64]`	
`i64.add`	`7c`	`[i64 i64] → [i64]`	
`i64.sub`	`7d`	`[i64 i64] → [i64]`	
`i64.mul`	`7e`	`[i64 i64] → [i64]`	
`i64.div_s`	`7f`	`[i64 i64] → [i64]`	traps on zero
`i64.div_u`	`80`	`[i64 i64] → [i64]`	traps on zero
`i64.rem_s`	`81`	`[i64 i64] → [i64]`	traps on zero
`i64.rem_u`	`82`	`[i64 i64] → [i64]`	traps on zero
`i64.and`	`83`	`[i64 i64] → [i64]`	
`i64.or`	`84`	`[i64 i64] → [i64]`	
`i64.xor`	`85`	`[i64 i64] → [i64]`	
`i64.shl`	`86`	`[i64 i64] → [i64]`	
`i64.shr_s`	`87`	`[i64 i64] → [i64]`	
`i64.shr_u`	`88`	`[i64 i64] → [i64]`	
`i64.rotl`	`89`	`[i64 i64] → [i64]`	
`i64.rotr`	`8a`	`[i64 i64] → [i64]`	

f32 operations A.8

Instruction	Opcode	Type	Immediates and notes
f32.eq	5b	[f32 f32] → [i32]	
f32.ne	5c	[f32 f32] → [i32]	
f32.lt	5d	[f32 f32] → [i32]	
f32.gt	5e	[f32 f32] → [i32]	
f32.le	5f	[f32 f32] → [i32]	
f32.ge	60	[f32 f32] → [i32]	
f32.abs	8b	[f32] → [f32]	
f32.neg	8c	[f32] → [f32]	
f32.ceil	8d	[f32] → [f32]	
f32.floor	8e	[f32] → [f32]	
f32.trunc	8f	[f32] → [f32]	
f32.nearest	90	[f32] → [f32]	
f32.sqrt	91	[f32] → [f32]	
f32.add	92	[f32 f32] → [f32]	
f32.sub	93	[f32 f32] → [f32]	
f32.mul	94	[f32 f32] → [f32]	
f32.div	95	[f32 f32] → [f32]	
f32.min	96	[f32 f32] → [f32]	
f32.max	97	[f32 f32] → [f32]	
f32.copysign	98	[f32 f32] → [f32]	

f64 operations

A.9

Instruction	Opcode	Type	Immediates and notes
f64.eq	61	[f64 f64] → [i32]	
f64.ne	62	[f64 f64] → [i32]	
f64.lt	63	[f64 f64] → [i32]	
f64.gt	64	[f64 f64] → [i32]	
f64.le	65	[f64 f64] → [i32]	
f64.ge	66	[f64 f64] → [i32]	
f64.abs	99	[f64] → [f64]	
f64.neg	9a	[f64] → [f64]	
f64.ceil	9b	[f64] → [f64]	
f64.floor	9c	[f64] → [f64]	
f64.trunc	9d	[f64] → [f64]	
f64.nearest	9e	[f64] → [f64]	
f64.sqrt	9f	[f64] → [f64]	
f64.add	a0	[f64 f64] → [f64]	
f64.sub	a1	[f64 f64] → [f64]	
f64.mul	a2	[f64 f64] → [f64]	
f64.div	a3	[f64 f64] → [f64]	
f64.min	a4	[f64 f64] → [f64]	
f64.max	a5	[f64 f64] → [f64]	
f64.copysign	a6	[f64 f64] → [f64]	

Instruction	Opcode	Type	Immediates and notes
`i32.wrap_i64`	`a7`	`[i64] → [i32]`	
`i32.trunc_f32_s`	`a8`	`[f32] → [i32]`	traps if out of range or NaN
`i32.trunc_f32_u`	`a9`	`[f32] → [i32]`	traps if out of range or NaN
`i32.trunc_f64_s`	`aa`	`[f64] → [i32]`	traps if out of range or NaN
`i32.trunc_f64_u`	`ab`	`[f64] → [i32]`	traps if out of range or NaN
`i64.extend_i32_s`	`ac`	`[i32] → [i64]`	
`i64.extend_i32_u`	`ad`	`[i32] → [i64]`	
`i64.trunc_f32_s`	`ae`	`[f32] → [i64]`	traps if out of range or NaN
`i64.trunc_f32_u`	`af`	`[f32] → [i64]`	traps if out of range or NaN
`i64.trunc_f64_s`	`b0`	`[f64] → [i64]`	traps if out of range or NaN
`i64.trunc_f64_u`	`b1`	`[f64] → [i64]`	traps if out of range or NaN
`f32.convert_i32_s`	`b2`	`[i32] → [f32]`	
`f32.convert_i32_u`	`b3`	`[i32] → [f32]`	
`f32.convert_i64_s`	`b4`	`[i64] → [f32]`	
`f32.convert_i64_u`	`b5`	`[i64] → [f32]`	
`f32.demote_f64`	`b6`	`[f64] → [f32]`	
`f64.convert_i32_s`	`b7`	`[i32] → [f64]`	
`f64.convert_i32_u`	`b8`	`[i32] → [f64]`	

Instruction	Opcode	Type	Immediates and notes
f64.convert_i64_s	b9	[i64] → [f64]	
f64.convert_i64_u	ba	[i64] → [f64]	
f64.promote_f32	bb	[f32] → [f64]	
i32.reinterpret_f32	bc	[f32] → [i32]	
i64.reinterpret_f64	bd	[f64] → [i64]	
f32.reinterpret_i32	be	[i32] → [f32]	
f64.reinterpret_i64	bf	[i64] → [f64]	
i32.extend8_s	c0	[i32] → [i32]	
i32.extend16_s	c1	[i32] → [i32]	
i64.extend8_s	c2	[i64] → [i64]	
i64.extend16_s	c3	[i64] → [i64]	
i64.extend32_s	c4	[i64] → [i64]	
i32.trunc_sat_f32_s	fc 00	[f32] → [i32]	saturates
i32.trunc_sat_f32_u	fc 01	[f32] → [i32]	saturates
i32.trunc_sat_f64_s	fc 02	[f64] → [i32]	saturates
i32.trunc_sat_f64_u	fc 03	[f64] → [i32]	saturates
i64.trunc_sat_f32_s	fc 04	[f32] → [i64]	saturates
i64.trunc_sat_f32_u	fc 05	[f32] → [i64]	saturates
i64.trunc_sat_f64_s	fc 06	[f64] → [i64]	saturates

Instruction	Opcode	Type	Immediates and notes
`i64.trunc_sat_f64_u`	`fc 07`	`[f64] → [i64]`	saturates

References A.11

Instruction	Opcode	Type	Immediates and notes
`ref.null ht`	`d0`	`[] → [(ref null ht)]`	heap type
`ref.is_null`	`d1`	`[ref] → [i32]`	
`ref.func f`	`d2`	`[] → [(ref $t)]`	function index
`ref.eq`	`d3`	`[eqref eqref] → [i32]`	
`ref.as_non_null`	`d4`	`[(ref null ht)] → [(ref ht)]`	traps on null
`br_on_null l`	`d5`	`[t* (ref null ht)] → [t* (ref ht)]`	label depth
`br_on_non_null l`	`d6`	`[t* (ref null ht)] → [t*]`	label depth

Instruction	Opcode	Type	Immediates and notes
`struct.new s`	`fb 00`	`[t*] → [(ref $s)]`	type
`struct.new_default s`	`fb 01`	`[] → [(ref $s)]`	type
`struct.get s i`	`fb 02`	`[(ref null $s)] → [t]`	type, field
`struct.get_s s i`	`fb 03`	`[(ref null $s)] → [i32]`	packed field
`struct.get_u s i`	`fb 04`	`[(ref null $s)] → [i32]`	packed field
`struct.set s i`	`fb 05`	`[(ref null $s) t] → []`	type, field
`array.new a`	`fb 06`	`[t i32] → [(ref $a)]`	type
`array.new_default a`	`fb 07`	`[i32] → [(ref $a)]`	type
`array.new_fixed a n`	`fb 08`	`[t^n] → [(ref $a)]`	type, count
`array.new_data a d`	`fb 09`	`[i32 i32] → [(ref $a)]`	type, data segment
`array.new_elem a e`	`fb 0a`	`[i32 i32] → [(ref $a)]`	type, element segment
`array.get a`	`fb 0b`	`[(ref null $a) i32] → [t]`	type
`array.get_s a`	`fb 0c`	`[(ref null $a) i32] → [i32]`	packed
`array.get_u a`	`fb 0d`	`[(ref null $a) i32] → [i32]`	packed
`array.set a`	`fb 0e`	`[(ref null $a) i32 t] → []`	type
`array.len`	`fb 0f`	`[(ref null array)] → [i32]`	
`array.fill a`	`fb 10`	`[(ref null $a) i32 t i32] → []`	type
`array.copy a b`	`fb 11`	`[(ref null $a) i32 (ref null $b) i32 i32] → []`	two types
`array.init_data a d`	`fb 12`	`[(ref null $a) i32 i32 i32] → []`	type, data segment

Instruction	Opcode	Type	Immediates and notes
`ref.test rt`	`fb 14`	`[ref] → [i32]`	reference type
`ref.cast rt`	`fb 16`	`[ref] → [rt]`	traps if the cast fails
`br_on_cast l rt1 rt2`	`fb 18`	`[t* rt1] → [t* rt1']`	flags, label, two types
`br_on_cast_fail l rt1 rt2`	`fb 19`	`[t* rt1] → [t* rt2]`	flags, label, two types
`any.convert_extern`	`fb 1a`	`[externref] → [anyref]`	
`extern.convert_any`	`fb 1b`	`[anyref] → [externref]`	
`ref.i31`	`fb 1c`	`[i32] → [(ref i31)]`	
`i31.get_s`	`fb 1d`	`[i31ref] → [i32]`	
`i31.get_u`	`fb 1e`	`[i31ref] → [i32]`	

SIMD (a selection) A.13

Instruction	Opcode	Type	Immediates and notes
`v128.load`	`fd 00`	`[i32] → [v128]`	memarg
`v128.store`	`fd 0b`	`[i32 v128] → []`	memarg
`v128.const`	`fd 0c`	`[] → [v128]`	16 bytes
`i8x16.shuffle`	`fd 0d`	`[v128 v128] → [v128]`	16 lane indices
`i8x16.swizzle`	`fd 0e`	`[v128 v128] → [v128]`	
`i32x4.splat`	`fd 11`	`[i32] → [v128]`	
`f32x4.splat`	`fd 13`	`[f32] → [v128]`	
`i32x4.extract_lane`	`fd 1b`	`[v128] → [i32]`	lane index
`f32x4.extract_lane`	`fd 1f`	`[v128] → [f32]`	lane index
`i32x4.replace_lane`	`fd 1c`	`[v128 i32] → [v128]`	lane index
`v128.and`	`fd 4e`	`[v128 v128] → [v128]`	
`v128.bitselect`	`fd 52`	`[v128 v128 v128] → [v128]`	
`v128.any_true`	`fd 53`	`[v128] → [i32]`	
`i32x4.all_true`	`fd a3 01`	`[v128] → [i32]`	
`i32x4.add`	`fd ae 01`	`[v128 v128] → [v128]`	
`i32x4.mul`	`fd b5 01`	`[v128 v128] → [v128]`	
`f32x4.add`	`fd e4 01`	`[v128 v128] → [v128]`	
`f32x4.mul`	`fd e6 01`	`[v128 v128] → [v128]`	
`f32x4.div`	`fd e7 01`	`[v128 v128] → [v128]`	
`f32x4.sqrt`	`fd e3 01`	`[v128] → [v128]`	
`f32x4.lt`	`fd 43`	`[v128 v128] → [v128]`	lane masks
`v128.load32_splat`	`fd 09`	`[i32] → [v128]`	memarg

Instruction	Opcode	Type	Immediates and notes
`f32x4.relaxed_madd`	`fd 85 02`	`[v128 v128 v128] → [v128]`	relaxed

Threads (a selection) A.14

Instruction	Opcode	Type	Immediates and notes
`memory.atomic.notify`	`fe 00`	`[i32 i32] → [i32]`	memarg
`memory.atomic.wait32`	`fe 01`	`[i32 i32 i64] → [i32]`	memarg
`atomic.fence`	`fe 03`	`[] → []`	
`i32.atomic.load`	`fe 10`	`[i32] → [i32]`	memarg
`i32.atomic.store`	`fe 17`	`[i32 i32] → []`	memarg
`i32.atomic.rmw.add`	`fe 1e`	`[i32 i32] → [i32]`	memarg
`i32.atomic.rmw.cmpxchg`	`fe 48`	`[i32 i32 i32] → [i32]`	memarg

Appendix B: Tools

This appendix collects the commands used in the book, and a few more, grouped by tool. The versions the book was checked against are:

```
wat2wasm --version
wasm-opt --version
node --version | cut -d. -f1
python3 -c "import wasmtime, importlib.metadata as m;
    print('wasmtime', m.version('wasmtime'))"
```

```
1.0.39
wasm-opt version 132 (version_132)
v22
wasmtime 50.0.1
```

wabt B.1

The WebAssembly Binary Toolkit: small, focused tools that follow the specification closely. `npm install wabt` gives builds compiled to WebAssembly; native builds are on GitHub.

Command	Does
`wat2wasm in.wat -o out.wasm`	assemble the text format
`wat2wasm -v in.wat -o out.wasm`	the same, printing an annotated byte dump on stderr
`wat2wasm --debug-names in.wat`	keep `$names` in a name section
`wasm2wat in.wasm`	disassemble to the text format
`wasm2wat --fold-exprs in.wasm`	disassemble to the folded form
`wasm-objdump -h in.wasm`	list the sections and their sizes
`wasm-objdump -x in.wasm`	print every section's details
`wasm-objdump -d in.wasm`	disassemble the code, with bytes
`wasm-objdump -s in.wasm`	print each section's raw contents
`wasm-objdump -j Code -x in.wasm`	only one section
`wasm-validate in.wasm`	validate
`wasm-interp in.wasm --run-all-exports`	run every export that takes no arguments
`wasm-interp in.wasm -r f -a i32:5`	call `f(5)`
`wasm-interp in.wasm --run-all-exports --trace`	the same, printing each instruction and the stack
`wasm-strip in.wasm`	remove every custom section
`wasm-decompile in.wasm`	a C-like rendering, easier to skim than the text format
`wasm2c in.wasm -o out.c`	translate to C, for running a module natively

Most tools accept `--enable-<feature>` and `--enable-all` for features that are not on by default.

Binaryen B.2

A compiler infrastructure and optimiser for WebAssembly, used by Emscripten, `wasm-pack`, Kotlin, Dart and many others. `npm install binaryen` gives the tools and a JavaScript API.

Command	Does
`wasm-opt -O3 in.wasm -o out.wasm`	optimise for speed
`wasm-opt -Oz in.wasm -o out.wasm`	optimise for size
`wasm-opt -g ...`	keep the name section
`wasm-opt --print in.wasm`	print Binaryen's view of the module
`wasm-opt --func-metrics in.wasm`	sizes and instruction counts per function
`wasm-opt --fuzz-exec in.wasm`	run the exports before and after optimising, and compare
`wasm-opt random.bin -ttf -o fuzz.wasm`	turn random bytes into a valid module, for fuzzing
`wasm-reduce`	shrink a module while it still shows a bug
`wasm-merge a.wasm a b.wasm b -o out.wasm`	link modules into one
`wasm-split`	split a module into a primary and a lazily loaded secondary
`wasm-metadce`	remove code unreachable from a given set of roots
`wasm2js in.wasm`	translate to JavaScript, for very old browsers
`wasm-as`, `wasm-dis`	Binaryen's own assembler and disassembler

Binaryen's tools take `--enable-<feature>` and `--all-features`.

wasm-tools B.3

The Bytecode Alliance's toolkit, built on the same crates as Wasmtime. Install with `cargo install wasm-tools`, or from release binaries.

Command	Does
`wasm-tools parse in.wat -o out.wasm`	assemble
`wasm-tools print in.wasm`	disassemble
`wasm-tools validate in.wasm`	validate, with every standard feature
`wasm-tools dump in.wasm`	an annotated dump of every byte
`wasm-tools objdump in.wasm`	list the sections
`wasm-tools strip in.wasm`	remove custom sections
`wasm-tools smith < random.bin > out.wasm`	generate a random valid module
`wasm-tools shrink`	reduce a module while a predicate holds
`wasm-tools wast script.wast`	parse and validate a WAST script
`wasm-tools addr2line in.wasm 0x2d`	map a code offset to a source line, through DWARF

Wasmtime B.4

A standalone engine, compiled ahead of time with Cranelift. Useful as a second engine for differential testing.

Command	Does
`wasmtime run in.wasm`	run a WASI command module
`wasmtime run --invoke f in.wasm 1 2`	call an export with arguments
`wasmtime wast script.wast`	run a WAST script
`wasmtime compile in.wasm`	compile ahead of time to a `.cwasm`
`pip install wasmtime`	the Python bindings: `wasmtime.wat2wasm`, and a full embedding API

Node and V8 B.5

Command	Does
`node --liftoff-only script.mjs`	run WebAssembly only in the baseline tier
`node --no-liftoff script.mjs`	compile everything with TurboFan up front
`node --experimental-wasm-<feature>`	enable a proposal not yet on by default
`node --stack-size=N`	change the stack limit, in KiB
`node --prof script.mjs`	a CPU profile, including WebAssembly frames
`node --inspect-brk script.mjs`	debug in Chrome's DevTools

This book's scripts B.6

Command	Does
`python3 scripts/book.py test`	run every example in the book
`python3 scripts/book.py test 06*`	run the examples in matching chapters
`python3 scripts/book.py test --accept`	update expected outputs that changed
`python3 scripts/book.py build`	build the HTML and PDF editions
`node scripts/run.mjs m.wasm --invoke "f 1 2"`	the book's host
`python3 scripts/wast.py script.wast`	run a WAST script in V8
`python3 scripts/opcodes.py`	print Appendix A's tables

Appendix C: Further Reading

The specification C.1

The WebAssembly specification, at `webassembly.github.io/spec`, is unusually readable for a standard. Its structure section defines the abstract syntax; validation gives the typing rules as inference rules (Chapter 6's stack typing, precisely); execution defines what each instruction does; and the binary and text format sections are the definitive references for Part IV. The JavaScript API and Web API specifications, beside it, define everything in Part V.

Proposals each have a repository under `github.com/WebAssembly`, with an *overview* document explaining the design and its motivation. The overviews for exception handling, GC, tail calls, threads and relaxed SIMD are the best introductions to those features after this book. `webassembly.org/features` tracks which engines support which proposals.

The *tool conventions* repository, `WebAssembly/tool-conventions`, documents what is not part of the standard but is shared by toolchains: the C ABI, the linking and relocation format of object files, the producers section, DWARF usage, and the dynamic linking convention.

Papers C.2

- Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai and JF Bastien, *Bringing the Web up to Speed with WebAssembly*, PLDI 2017. The design paper: why the format looks as it does, including structured control flow and validation.
- Conrad Watt, *Mechanising and Verifying the WebAssembly Specification*, CPP 2018. The formal proof of the type system's soundness, which caught real bugs in the specification.
- Alon Zakai, *Emscripten: an LLVM-to-JavaScript Compiler*, SPLASH 2011. Introduces the Relooper.
- Norman Ramsey, *Beyond Relooper: Recursive Translation of Unstructured Control Flow to Structured Control Flow*, ICFP 2022. The algorithm of Chapter 7, explained clearly.
- Abhinav Jangda, Bobby Powers, Emery D. Berger and Arjun Guha, *Not So Fast: Analyzing the Performance of WebAssembly*

vs. Native Code, USENIX ATC 2019. Where the overhead comes from.

Compilers to read C.3

The best way to learn how to target WebAssembly is to read the output of existing compilers, with `wasm2wat` or `wasm-objdump -d`, for small programs in their source languages:

- **Clang and LLVM** (C, C++, and the basis of Rust, Zig, Odin and others): the `WebAssembly` backend, especially `WebAssemblyCFGStackify` for control flow and the frame lowering for the shadow stack.
- **Emscripten**: the complete toolchain around LLVM, including the JavaScript runtime, and the place to see the boundary patterns of Chapter 16 at full scale.
- **Binaryen**: a model of a WebAssembly-specific optimiser, with an IR close to the format.
- **Kotlin/Wasm, Dart and J2Wasm**: production compilers targeting WasmGC.
- **Wasmtime and Cranelift**: an engine whose source is easier to read than a browser's, for seeing how modules are compiled to machine code.
- **V8's blog** (`v8.dev/blog`): articles on Liftoff, TurboFan's WebAssembly pipeline, tiering, WasmGC, and the JS-to-Wasm call path.

Books C.4

- Robert Nystrom, *Crafting Interpreters*. Not about WebAssembly, but the clearest account there is of building a language implementation step by step, and a model for structuring a compiler project.
- Andrew Appel, *Modern Compiler Implementation*. Classic coverage of the middle of a compiler (IR, SSA, dominators, register allocation) that Chapter 7 and Chapter 24 lean on.
- Keith Cooper and Linda Torczon, *Engineering a Compiler*. The same ground with more on optimisation.