1 Preparation

Word budget: ~2500-3000 words

Describe the work undertaken before code was written.
-> Wasm research – include the stuff from the research doc I wrote.
-> include Relooper research here too
"Requirements Analysis" section
-> refer to appropriate software engineering techniques used in the diss
Cite new programming language learnt
Declare starting point
Explain background material required beyond IB
Researching LALRPOP - show good professional use of tools
Talk about revision control strategy, licensing of any libraries I used

1.1 WebAssembly

Before starting to write the compiler, I extensively researched the WebAssembly specification [1], to gain a deep understanding of the binary code the project aims to generate. The following sections provide a high-level overview of the instruction set architecture.

1.1.1 Primitive values

The primitive value types supported by WebAssembly are outlined in Table 1.1, and described in more detail below.

Integers can be interpreted as either signed or unsigned, depending on the operations applied to them. They will also be used to store data such as booleans and memory addresses[1] Integer literals are encoded in the program using the LEB128 variable-length encoding scheme [2]. Listing 1.1 shows how to encode an unsigned integer. The algorithm is almost the same for signed integers (encoded in two's complement); the only difference is that we sign-extend to a multiple of 7 bits, rather than zero-extend. When decoding, the decoder needs to know if the number is signed or unsigned, to know whether to decode it as a two's complement number or not. This is why WebAssembly has signed and unsigned variants of some instructions.

[1] I.e. addresses within WebAssembly's sandboxed linear memory space, rather than function addresses etc..

Type	Constructor	Bit width
Integer	`i32`	32-bit
	`i64`	64-bit
Float	`f32`	32-bit
	`f64`	64-bit
Vector	`v128`	128-bit
Reference	`funcref`	Opaque
	`externref`	

Table 1.1: WebAssembly primitive types.

```
fn leb128_encode_unsigned(n) {
    Zero-extend n to a multiple of 7 bits
    Split n into groups of 7 bits
    Add a 0 bit to the front of the most significant group
    Add a 1 bit to the front of every other group
    Return bytes in little-endian order
}
```

Listing 1.1: Pseudocode for the LEB128 encoding scheme (for unsigned integers).

Floating-point numbers are encoded using the IEEE 754-2019 standard [3]. This is the same standard used by many programming languages, including Rust, so the byte representation will not need to be converted between the compiler and the WebAssembly binary.

WebAssembly also provides vector types and reference types. Vectors can store either integers or floats: in either case, the vector is split into a number of evenly-sized numbers. Vector instructions exist to operate on these values. `funcref` values are pointers to WebAssembly functions, and `externref` values are pointers to other types of object that can be passed into WebAssembly. These would be used for indirect function calls, for example. This project does not have a need for using either of these types; C does not have any concept of vector types, and I am not supporting function references. I will only use the four main integer and float types.

1.1.2 Instructions

WebAssembly is a stack-based architecture; all instructions operate on the stack. For binary instructions that take their operands from the stack, the first operand is the one that was pushed to the stack first, and the second operand is the one pushed to the stack most recently (see Listing 1.2).

```
i32.const 10 ;; first operand
i32.const 2  ;; second operand
i32.sub
```

Listing 1.2: WebAssembly instructions to calculate `10 - 2`.

All arithmetic instructions specify the type of value that they expect. In Listing 1.2, we put two `i32` values on the stack, and use the `i32` variant of the `sub` instruction. The module would fail to instantiate if the types did not match. Some arithmetic instructions have signed and unsigned variants, such as the less-than instructions `i32.lt_u` and `i32.lt_s`. This is the case for all instructions where the signedness of a number would make a difference to the result.

WebAssembly only supports structured control flow, in contrast to the unstructured control flow found in most instruction sets that feature arbitrary jump instructions. There are three types of block: `block`, `loop`, and `if`. The only difference between `block` and `loop` is the semantics of branch instructions. When referring to a `block`, `br` will jump to the end of it, and when referring to a `loop`, `br` will jump back to the start. This is analogous to `break` and `continue` in C, respectively. It is worth noting that `loop` doesn't loop back to the start implicitly; an explicit `br` instruction is required. `if` blocks, which may optionally have an `else` block, conditionally execute depending on the value on top of the stack. With regard to `br` instructions, they behave like `block`.

1.1.3 Modules

A module is a unit of compilation and loading for a WebAssembly program. There is no distinction between a 'library' and a 'program', such as there are in other languages; there are only modules which export functions to the instantiator. Modules are split up into different sections. Each section starts with a section ID and the size of the section in bytes, followed by the body of the section. Each of the sections is described below.

The types section defines any types of functions used in the module, including imported functions. Each type has a type index, allowing the same type to be used by multiple functions.

The imports section defines everything that is imported to the module from the runtime environment. This includes imported functions as well as memories, tables, and globals.

The functions section is a map from function indexes to type indexes. This is separate from the actual body code of the functions, because this allows modules to be decoded in a single pass. The type of all functions will be known before any of the code is read.

TODO rest of module sections

1.2 Project Strategy

1.2.1 Requirements Analysis

1.2.2 Software Engineering Methodology

1.2.3 Testing

1.3 Starting Point

1.3.1 Knowledge and experience

- IB Compilers Course
- Experience with JavaScript + Python (cos I used those for runtime/testing)
- Experience writing C, my source language

1.3.2 Tools Used

- Say here that I learned Rust for this project – talk about the borrow checker and memory safety
- Also used JavaScript for runtime + Python for testing