Compiling Flintlang to eWASM

Opening notes not specific to any particular option:

- I have not yet been able to successfully deploy any contracts onto a local blockchain (I tried links 10 and 11), so the document talks only about testing and verifying the WASM, not specifically eWASM.
- The Ewasm Contract Interface specifies that the main function must exist, and take no parameters and return nothing. I am not sure what this means for our contracts where we have initialiser parameters.
- It is not enough for us to generate the WASM alone. Because the contract gets hashed when it is put on the blockchain, we will also need to generate an Application Binary Interface (ABI), which is essentially just a JSON description of the external ethereum methods, so it should not be too much of a problem.

Option 1 - Compile to WAT (Web Assembly Text format).

One option is compiling directly (ish) to WASM by generating WAT, the human readable version of WASM. This of course would require a reasonable knowledge of WASM and for this I suggest some of the links at the bottom of the document.

Pros:

- No dependencies required
- Full control over what is generated, meaning it may not be too hard to satisfy the ECI
- We would not rely on any APIs or wrappers
- Accessing ethereum currency and other methods can be done easily

Cons:

- Stack based language - may be difficult to translate to. However, as shown later, there is a WASM syntactic sugar that allows instructions to be written slightly more human readably. Whether this would make enough of a difference, I do not know
- Uses linear memory only. We would need to store all of the contract state as raw bytes, and build a memory manager to ensure we know how to handle data in the contract
- No optimisations (unless we write them ourselves)

General observations:

- WASM has the `unreachable` instruction which as far as I can tell is analogous to throwing an exception unconditionally. Thus assertions should be possible

- Memory can only grow in multiples of the page size (64K) in WASM, and it cannot shrink once grown.
- To be considered eWASM rather than just WASM, we need to meet the ECI specification. This essentially means only importing ethereum namespaces, having a main function that takes no arguments and returns nothing, and exporting our memory.

The simplest example contract of memory use is the counter contract. Here is a handwritten mockup, tested in WebAssembly Studio. Please note that lines starting with // are my comments, but are invalid syntax and need to be removed if you wish to run this yourself.

```
(module
  // Says we want at least 1 page (64K) of memory. Note memory can only
  // grow, it cannot shrink
 (memory 1)
  // This defines all our data. We start at index 0, with offset 0
  // and our data is defined by the string. This string defines four bytes in little
  // endian, so our counter starts at 5
 (data 0 (i32.const 0) "\05\00\00\00")

  // Functions are defined by the func keyword, and can be unnamed (they will be
  // assigned a number) or named as below. We may also take parameters: a signature
  // for setValue may be func $setValue (param $val i32).
 (func $Contract_getValue (result i32)
   // as this is stack based, we push our offset where our counter is stored and
   // then load. This can be written also as (i32.load (i32.const 0))
   i32.const 0
   i32.load
 )

 (func $Contract_Increment
   i32.const 0 // push our offset to the stack
   call $Contract_getValue
   i32.const 1
   i32.add // add the 2 items at the top of the stack
   i32.store // store the value back at offset 0
 )

 // The following is almost identical to increment, but written with syntactic sugar.
 (func $Contract_Decrement
   (i32.store
     (i32.const 0)
     (i32.sub
       (call $Contract_getValue)
       (i32.const 1)))
)
// These essentially make the functions public; they can be called from JS, rust etc.
(export "Contract_getValue" (func $Contract_getValue))
(export "Contract_Increment" (func $Contract_Increment))
(export "Contract_Decrement" (func $Contract_Decrement)))
```

Note that the entirety of our memory is 4 bytes (in little endian) representing one 32 bit number. I emphasise this only to point out that when storing more complex data, it may be quite non trivial to manage the memory properly.

To conclude, I would say that this would be fairly difficult, but not necessarily a bad option given the extra ethereum constraints (the ECI). I believe the main issues would be converting to a stack based language, and organising memory. The main benefits would be that we would be compiling directly to WASM, and would have all the control that comes with that.

Option 2 - Compile to LLVM bytecode, and then onto WASM

We can target wasm32 directly from llvm bytecode. To get a feel for the complexity of doing this in rust, I suggest checking this out, a lexer, parser and compiler in rust using inkwell for the toy language that serves as an LLVM tutorial. If we go with this option, the tutorial is definitely worth reading too.

Pros

- LLVM is well established and used as a back end for many programming languages. It should therefore be relatively easy to use.
- Many built in optimisations
- There are rust crates that support LLVM code generation safely. The goto one appears to be inkwell, so we would not need to worry too much about LLVM syntax.
- The generated WASM will likely have superior memory management with regards to things such as alignment than what we would likely achieve, and at lower effort.
- We can use `unreachable` as an assert(0)

Cons

- Reliance on dependencies (probably inkwell and definitely LLVM), and therefore learning how to use this and giving up some control over the final WASM
- We may have to postprocess the WASM exports to be compatible with the ECI
- Dynamic memory allocation would mean we need to carefully allocate and deallocate memory with LLVM mallocs and frees

In the course of trying to write a simple counter contract with inkwell, I note the following:

- Inkwell documentation is, while useful in places, not especially helpful on its own. Inkwell documentation, plus LLVM documentation takes you a lot further, but I suspect lots of experimentation will also be required.
- Because the underlying API is written in C++, the inkwell code is a little confusing coming from a rust context, especially regarding mutability. This is not a big issue, but I thought I would mention it.

I was able to create an LLVM representation of the counter contract with fairly little (but not especially good) code, which I have put here. It includes a built in ethereum function `getAddress` just to test importing. The LLVM code it outputs is here. Using LLVM and WABT, I wrote a small shell script which creates the WAT file for us to look at. A few notes on this:

- When I pasted the file into WebAssembly Studio, it told me the token `funcref` after a table is an unexpected token. Currently I do not know why this is, because it is defined in the WASM specification here, and WABT does not complain, so it may just be a WebAssembly Studio specific thing.
- After fixing the above, the file worked exactly as expected when calling individual functions from javascript.
- The ECI states that there should be exactly two exports: a main function with 0 inputs and outputs, and a memory called memory. Currently this contract does not satisfy this, but since I cannot deploy it on a testnet I do not think this matters for now.
- This contract has a useless function foo. It exists only for me to demonstrate that we can call ethereum methods (without foo the external function would be optimised away)

How to link to ethereum resources:

- Define a function of the same name and type in the LLVM. Define it to be externally linked, and add attributes to say where to import it from.
- Add the --allow-undefined flag to the wasm linker in the compile process.
- This creates an import in the WASM

To conclude, compiling to LLVM seems like a very solid option. Though it may be a little opaque and C++ oriented, the LLVM documentation is certainly comprehensive. The main issues are reliance on the inkwell crate and LLVM as a dependency. I also am not yet 100% sure how we will make exports work with the ECI, but I do not imagine this to be a major problem as we can always do it ‘manually’ with a post processing stage if we must.

Option 3 - Compile to intermediate language e.g. typescript or C or Go, and convert to WASM

Subsection: C:

Here is the counter contract written in C, which I can confirm works.

Pros

- We already know C so it is familiar
- C has support for compiling to WASM
- C is very simple, and the kind of C we would generate is unlikely to be especially difficult, since most of our data will be known at compile time and can therefore be on the stack

Cons

- Reliance on a framework like emscripten to convert C to WASM especially when using C specific things such as assert.h
- All the support for C to WASM is normally through emscripten. I ran the very simple contract above and it generated over 9000 lines of WAT. This leads me to believe that we would have to compile to WASM from C ourselves, using the LLVM tools. This is ok with very plain C but this is not really possible if we want to use any C library (e.g. assert.h) so I worry it would be limiting.
- It is not at all clear how we would interact with ethereum methods. In LLVM bytecode we can say where to link from, but it would probably be quite complicated in C
- Dynamic memory would be essentially impossible without writing our own malloc/ free functions, because it would require linking to C stdlib, which would violate the ECI.

I started thinking it would be very viable due to the WASM support for C. However, I now think that the WASM support is too geared towards web applications, and so I am not convinced it would as easily translate into eWASM. This would leave us having to translate it ourselves with the LLVM backend, but this leaves no room for C assertions etc. This, and the difficulties interfacing with ethereum, make me think that this option would be a bad idea.

Subsection: AssemblyScript

AssemblyScript is a subset of TypeScript, which is a typed subset of JavaScript. It is very well supported to translate to WASM. Here is a quick counter contract, which generated this WASM.

Pros

- Excellent support for WASM
- We would theoretically get good dynamic memory support from AssemblyScript
- We can declare functions and link them to the ethereum namespaces quite easily

Cons

- We still would not be able to use built in things such as assertions, since this compiles to an `import “env” ...` in WASM, which does not satisfy the ECI. Having said that, the EEI defines functions that may serve as assertions, in finish and revert, so this may not be a big issue.
- Setting up a simple hello world module locally ends up spawning a giant WASM file which includes an `import “env” “abort” .. ` which is against the ECI. It can be removed with a command line flag `--use abort=` but it is hard to say whether we will have other imports brought in, and the file is still very large for such a simple program.
- Lack of fine grained control over WASM output

I believe that this would be a much better option than C, since we can import and use ethereum methods and have control over the exports, while still generating quite simple readable code. However, I still believe it may just be too geared towards webapps and general purpose WASM to make it easy to generate eWASM.

More relevant links:

1. https://webassembly.github.io/spec/core/syntax/instructions.html
2. https://webassembly.github.io/spec/core/intro/overview.html#trap
3. https://webassembly.github.io/spec/core/appendix/index-instructions.html
4. https://webassembly.studio/
5. https://github.com/WebAssembly/wabt
6. https://blog.scottlogic.com/2018/04/26/webassembly-by-hand.html
7. https://webassembly.github.io/wabt/demo/wat2wasm/
8. https://www.llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html
9. https://rsms.me/wasm-intro
10. https://github.com/ethereum/go-ethereum
11. https://medium.com/zeonlab-blockchain-semantic-blog/how-to-install-testnet-for-the-ethereum-blockchain-client-27d8be4beecb
12. https://github.com/ewasm/design/blob/master/eth_interface.md
13. https://emscripten.org/docs/compiling/WebAssembly.html
14. https://github.com/TheDan64/inkwell
15. https://github.com/second-state/SOLL/ (Greatly helped me understand how to link ethereum methods into LLVM, and may well be useful as a reference, given that it’s a similar project to ours)

16. https://surma.dev/things/c-to-webassembly/