Group members: Jarne Thys, Martijn Snoeks

Code repository: https://github.com/JarneT-2159795/seis-jarnethys-martijnsnoeks

- Code VM and compiler: root/includes
- Code Cloudflare: root/test-cf/vm-worker
- Cloudflare VM only: https://vm-worker.jarne-thys.workers.dev/
- Cloudflare VM and compiler combined: https://vm-worker.jarne-thys.workers.dev/compile

Main blog page: https://jarnet-2159795.github.io/seis-jarnethys-martijnsnoeks/

Blogs:

- WASM-VM: https://jarnet-2159795.github.io/seis-jarnethys-martijnsnoeks/wasm-vm
- WAT-compiler: https://jarnet-2159795.github.io/seis-jarnethys-martijnsnoeks/wasm-compiler
- Cloudflare: https://jarnet-2159795.github.io/seis-jarnethys-martijnsnoeks/cloudflare
- Paper review: https://jarnet-2159795.github.io/seis-jarnethys-martijnsnoeks/paper

Paper used: Paper: https://www.usenix.org/conference/atc19/presentation/jangda

Implemented features

The next pages consist of the assignments we got. We crossed of every feature we implemented. For task 1 that is all must have and nice to have features and some optional features like a start section, global get and set, and i64 and f64 support. For the second task we have implemented all but one must have feature: the proper handling of incorrect syntax. Unfortunately we didn’t have enough time left to correctly implement this. We do have the nice to haves like using local variables with $var, constant folding, section size indicators and func exports. Task 3 can perform all the features from task 1 and 2. We provided a very basic interface to use the worker. For task 5 we provided a high level overview and discussed their findings. We also give our own opinion on their used methods.

Task 1: WASM-VM

MUST have:
- ~~Support for LEB128 encoded integer values and floats~~
- ~~Support for reading ASCII strings (e.g., export names, data segment contents)~~
- ~~Way to call WASM exported functions from the VM by string-name and at least print their results~~
- ~~"func" import (should be emulated/faked/hardlinked. No actual dynamic coupling with C++ functions necessary)~~
- ~~"data" section~~
- ~~"memory" import~~
- ~~"type" section~~
- Support for the following instructions
    - ~~block, loop, if, else, br, br_if, return, call~~
    - ~~local.get, local.set, local.tee~~
    - ~~i32.load, i32.store (with support for offset= parameter!), memory.size, memory.grow~~
    - ~~i32.const, i32.add, i32.sub, i32.div, i32.mul, i32.and, i32.or, i32.xor, i32.shl, i32.shr~~
    - ~~f32.const, f32.add~~
    - At least 1 float-to-int conversion
        - Pick yourself from ~~trunc~~_, convert_, promote_, demote_, wrap_, extend_, ~~reinterpret_~~ functions
    - ~~All i32 comparison operators (eqz, eq, ne, lt variants, gt variants, le variants, ge variants)~~
- ~~Properly handle errors/unsupported instructions~~
    - ~~Proper error messages + graceful exit~~
    - ~~Problems in 1 function/section shouldn't necessarily mean other functions can't be properly called!~~

Nice to have/expected for good score:
- ~~Infinite loop detection~~
- ~~Memory out of bounds~~ detection
- The following instructions:
    - ~~i32.rem, i32.rotr, i32.rotl~~
    - ~~drop~~

Optional:
- ~~"start" section~~
- Proper validation that the function implementations/types actually adhere to the WASM "types" section
    - Other semantic validation (e.g., how many values can be left on the stack when returning from any code path)
- The following instructions:
    - ~~global.get, global.set, clz, ctz, popcnt~~
- ~~i64 and f64 instruction support~~
- ~~f32 support beyond what's listed above~~
- ~~float-to-int conversion beyond what's listed above~~

- "memory" export
- ~~memory.fill, memory.copy,~~ memory.init
- data.drop
- any instructions not mentioned explicitly in this assignment

Task 2: WAT-parser

For this task, you will write a lexer, parser and compiler that transforms WebAssembly Text Format (WAT) into WebAssembly bytecode (WASM).
The generated WASM of course has to be compliant with the WASM spec and be executable by any WASM engine, including your own VM from task 1!

Since there are several variants of WAT, we will focus on just one: the output of the wat-desugar tool in the wabt toolkit (https://github.com/WebAssembly/wabt).

MUST have features:
- ~~Support for LEB128 encoded integer values and floats~~
- ~~Support for reading ASCII strings (e.g., export names, data segment contents)~~
- ~~Way to call WASM exported functions from the VM by string-name and at least print their results~~
- ~~"func" import (should be emulated/faked/hardlinked. No actual dynamic coupling with C++ functions necessary)~~
- ~~"data" section~~
- ~~"memory" import~~
- ~~"type" section~~
- Support for the following instructions
    - ~~block, loop,~~ ~~if, else~~, ~~br, br_if,~~ ~~return~~, ~~call~~
    - ~~local.get, local.set, local.tee~~
    - ~~i32.load, i32.store (with support for offset= parameter!)~~, ~~memory.size, memory.grow~~
    - ~~i32.const, i32.add, i32.sub,~~ ~~i32.div~~, ~~i32.mul,~~ ~~i32.and, i32.or, i32.xor, i32.shl, i32.shr~~
    - ~~f32.const, f32.add~~
    - At least 1 float-to-int conversion
        - Pick yourself from ~~trunc_~~, convert_, promote_, demote_, ~~wrap_~~, extend_, ~~reinterpret_~~ functions
    - ~~All i32 comparison operators (eqz, eq, ne, lt variants, gt variants, le variants, ge variants)~~
- ~~Support comments in both forms (single-line ;; and inline (;...;))~~
- Properly handle errors/unexpected syntax
    - Proper error messages + graceful exit
    - Problems in 1 function/section shouldn't mean other functions can't be properly compiled

Nice to have/expected for good score:
- ~~Support for $variable syntax~~ works only with local variables e.g. (local $var i32)
- ~~The "constant folding" optimization (applied recursively)~~
- ~~Proper size indicators at the start of sections (so VM doesn't necessarily need to rely on FIXUPS)~~
- ~~Named func exports~~

Task 3: WASM at the Edge

For this task, you will cross-compile your WAT compiler and WASM VM (tasks 2 and 1) from C++ to WASM
so you can compile and run WebAssembly, in WebAssembly, both in a browser and on the edge (in a Cloudflare edge worker).

The goal is to have a user be able to provide a WAT string (e.g., via a <textarea> on a web page),
and then have that string compiled and executed, the result shown to the user in the same web page.

For this task, you get a lot of example code/detailed explanations for the basics,
but you'll still have to figure out some things along the way to make it work properly.

MUST have features:

- ~~Browser-based execution of task 1 and 2 via emscripten cross compilation~~
- ~~Cloudflare worker-based execution of task 1 and 2 via emscripten cross compilation~~
- ~~Basic user interface (minimal: pass WAT string via URL request parameter)~~
- ~~Basic output from WASM execution (minimal: single integer output, console logs in wrangler/browser logs)~~

Nice to have/expected for good score:

- ~~Proper user interface (e.g., <textarea> where user can provide larger scripts)~~
  - similar to https://webassembly.github.io/wabt/demo/wat2wasm/
- ~~Proper output (e.g., full WASM bytestream output as well as execution output values)~~
  - similar to https://webassembly.github.io/wabt/demo/wat2wasm/
- ~~Ideally, this task 3 version of your compiler+VM supports all the features the "local" compilations of task 1 and 2 also support~~
  - There can be some exceptions, see also below

Optional:

- Find an open source C++/RUST/... project, cross-compile it to WASM as well, and run it at the CloudFlare edge
  - This can even be one that already has instructions on how to compile to wasm!

- Stress test the Free cloudflare workers plan
  - CF workers is limited to 10ms execution time and a small amount of memory
  - You can try to stress tests their systems by executing WASM code that takes a long time, has an infinite loop, that attempts to allocate lots of memory, etc.
  - The goal is to see how far you can go before they detect problems + how they handle errors gracefully

Task 5: Academic WASM

For this task, you will search for and then read an academic paper on WebAssembly.
This paper can be on any subtopic/aspect of WASM, but it needs to be a clearly published/peer-reviewed academic conference/journal paper.
- -> Just because it might look like a "paper" doesn't mean it's not just a student report ;)
- -> A published paper is typically at least 6 pages long (short paper) up to 12 or more (long paper)

We recommend using https://scholar.google.com/ to search for papers
- -> for most papers, you can download the .PDF straight from there

For your read paper, you will write a blog post summarizing and discussing its contents.

MUST have features:
- blog post that:
    - ~~Provides a (high-level) summary of the paper contents~~
    - ~~Provides a (basic) critical discussion of the paper's findings/conclusions, their usefulness in practice, their correctness, relevance today, etc.~~

Nice to have/expected for good score:
- ~~Personal opinion on the paper's approach (e.g., list things you would have done differently, missing items, things you doubt are correct, etc.)~~