Sathiarith Chau

# Rust Universal Machine (RUM) Design Documentation

## Problem statement

Given a .um file or standard input, the RUM program can read the input, identify what of the 14 instructions are given and execute the instruction. The universal machine will behave accordingly to the instructions. Given a broken code or buggy code, the program will either terminate or produce undefined behavior respective to the logic error. In this way, the program is an emulator.

## Invariants

1. The universal machine shall have 8 registers available to store values (u32s).
2. The universal machine shall provide memory (in the form of mem_segs, i.e., vec<vec<u32>>) to store program information.
    a. The memory shall be organized (by map_seg instruction)
    b. Unused memory will be “free”/unmapped (by convention, a vector will keep track of which of the memory/mem_segs vector can be rewritten).
3. The universal machine will keep count of the program’s list of instructions.
    a. Memory segment 0 will be loaded the user’s program.
    b. An infinite loop will repeatedly execute the call to read in an instruction.
    c. The program will only stop (without input/logic errors in the user’s program) when it reaches the halt instruction.
    d. The program counter will increment before call to the opcode instruction.
4. Instructions are u32s with 2 variations:
    a. For both, the first 4 bits (lsb being 28) identify the instruction type.
    b. For opcode 13, register A is 3 bits wide with lsb of 25, and a value field 25 bits wide with 0 lsb.
    c. The other opcodes have three designated registers A, B, and C at lsb of 6, 3, and 0 respectively.

## Architecture

Modules: main.rs, um.rs, parser.rs, rumload.rs, instructions.rs, test.rs, and lib.rs

The module main.rs will the driver code which:

1. receives the user input.
2. Creates a UM (universal machine) struct
3. Call rumload::load(input.as_deref()) and store in UM’s field, mem_segments vector (this would be $m[0])
4. Call parser function to iteratively process instructions

The module um.rs defines the public struct, UniversalMachine with the following fields:

1. pub registers: Vec<u32> //max capacity being 8, index 0 to 7

2. pub mem_segments: Vec<Vec<u32>>
3. pub free_segments: Vec<u32>
4. pub program_counter: usize
5. pub fn new() -> Self { fields initialized }

The module parser.rs will match the u32s to the instruction set using the first 4 bits in order to call the specific instruction function defined in instruction.rs. This is essentially a repurposed disassembler from rumdump.

1. type Umi = u32;
2. pub struct Field {
    a. pub width: u32,
    b. pub lsb: u32, }
3. pub static RA: Field = Field { width: 3, lsb: 6 };
4. pub static RB: Field = Field { width: 3, lsb: 3 };
5. pub static RC: Field = Field { width: 3, lsb: 0 };
6. pub static RL: Field = Field { width: 3, lsb: 25 };
7. pub static VL: Field = Field { width: 25, lsb: 0 };
8. pub static OP: Field = Field { width: 4, lsb: 28 };
9. enum Opcode {
    a. CMov,
    b. SegLoad,
    c. SegStore,
    d. Add,
    e. Mul,
    f. Div,
    g. Nand,
    h. Halt,
    i. MapSeg
    j. UnmapSeg,
    k. Output,
    l. Input,
    m. LoadProg,
    n. LoadValue, }
10. fn mask(bits: u32) -> u32 {}
11. pub fn get(field: &Field, instruction: &Umi) -> u32 {}
12. pub fn op(instruction: Umi) -> u32 {}
13. pub fn parse(um: &mut UniversalMachine) {
    a. let instruction = seg0 be the instruction indexed at program_counter
    b. program_counter++
    c. match get(&OP, instruction) {
        i. call cmov function from instruction.rs
        ii. … etc }

The module instruction.rs will define the implementation of each instruction (not all defined due to needing to plan it out but an example provided):

1. pub fn cmov(um: &mut UniversalMachine, A: &u32, B: &u32, C: &u32) {
    a. if um.register[C] != 0 { um.register[A] = um.register[B]; }
   } // if $r[C]= 0 then $r[A] := $r[B]
2. pub fn seg_load
3. …
4. pub load_value

The rumload.rs is the exact module used in rumdump lab to populate the words for processing:

1. pub fn load(input: Option<&str>) -> Vec<u32> { … }

The module test.rs will have the unit tests to assert the expected outcome for key functions. A common objective of the tests was to verify that the register(s) contain the correct value. With coordination of register assignments, values can be stored in memory and loaded. Tests for arithmetic operations, had their sum/product/quotient value verified at the assigned register.

1. Load_val_test
2. Cmove_test
3. Seg_store_test
4. Seg_load_test
5. Add_test
6. Mult_test
7. Div_test
8. Output_test
9. Map_seg_test

The module lib.rs is for the purpose of loading our imports:

1. pub mod um;
2. pub mod instructions;
3. pub mod rumload;
4. pub mod tests;
5. pub mod parser;