CSC 411 Assignment: A Universal Virtual Machine

Modules:

RUM
- src
  - main.rs
  - rumload.rs
  - machine
    - machine.rs
    - lib.rs
    - dinst.rs
    - error.rs
    - memory
      - memory.rs
      - lib.rs
    - registers
      - registers.rs
      - lib.rs

memory.rs

```
use std::{collections::VecDeque, usize};

pub struct Memory {
   // segmented memory
   pub seg_mem: Vec<Box<Vec<u32>>>,

   // unmapped segments ready to be mapped again
   pub unmapped_segs: VecDeque<usize>
}
impl Memory {
   // Memory constructor function
   pub fn new(instructions: Vec<u32>) -> Self {
       Memory {
           seg_mem: vec![Box::new(instructions)],
           unmapped_segs: VecDeque::new()
       }
   }

   // allocated a segment with a given length and return the Segment ID of the
allocated segment
   pub fn allocate(&mut self, length: usize) -> Option<usize> {
       todo!()
   }
   // deallocate a segment with and ID and return the same ID if it was deallocated
```

```
    pub fn deallocate(&mut self, id: usize) -> Option<usize> {
        todo!()
    }
}
```

registers.rs

```
pub struct CPU {
    pub registers: Vec<u32>
}
impl CPU {
    // CPU constructor with eight registers of u32's
    pub fn new() -> Self {
        CPU {
            registers: vec![0_u32; 8]
        }
    }
    // write a value on a register
    fn write(&mut self, val: u32, register: usize) {
        todo!()
    }
    // return a value from a register
    fn read(&self, register: usize) -> u32 {
        todo!()
    }
}
```

dinst.rs

```
pub struct Dinst {
    op: Option<u32>,
    a: Option<u32>,
    b: Option<u32>,
    c: Option<u32>,
    val: Option<u32>
}

impl Dinst {
    // get the op, a, b, c, val from an instruction and returns it inside of a struct
    pub fn disassemble(inst: Umi) -> Result<Self, std::io::Error> {
        match get(&OP, inst) {
            o if o == Some(Opcode::CMov as u32) => {
                Ok(Dinst {
```

```
                    op: o, a: get(&RA, inst), b: get(&RB, inst), c: get(&RC, inst),
val: None
                })
            },
            o if o ==  Some(Opcode::Load as u32) => {
                Ok(Dinst {
                    op: o, a: get(&RA, inst), b: get(&RB, inst), c: get(&RC, inst),
val: None
                })
            },
            o if o == Some(Opcode::Stor as u32) => {
                Ok(Dinst {
                    op: o, a: get(&RA, inst), b: get(&RB, inst), c: get(&RC, inst),
val: None
                })
            },
            o if o ==  Some(Opcode::ADD as u32) => {
                Ok(Dinst {
                    op: o, a: get(&RA, inst), b: get(&RB, inst), c: get(&RC, inst),
val: None
                })
            },
            o if o == Some(Opcode::MULT as u32) => {
                Ok(Dinst {
                    op: o, a: get(&RA, inst), b: get(&RB, inst), c: get(&RC, inst),
val: None
                })
            },
            o if o ==  Some(Opcode::DIV as u32) => {
                Ok(Dinst {
                    op: o, a: get(&RA, inst), b: get(&RB, inst), c: get(&RC, inst),
val: None
                })
            },
            o if o == Some(Opcode::NAND as u32) => {
                Ok(Dinst {
                    op: o, a: get(&RA, inst), b: get(&RB, inst), c: get(&RC, inst),
val: None
                })
            },
            o if o ==  Some(Opcode::HALT as u32) => {
                Ok(Dinst {
```

```
                    op: o, a: None, b: None, c: None, val: None
                })
            },
            o if o == Some(Opcode::Map as u32) => {
                Ok(Dinst {
                    op: o, a: None, b: get(&RB, inst), c: get(&RC, inst), val: None
                })
            },
            o if o == Some(Opcode::UnMap as u32) => {
                Ok(Dinst {
                    op: o, a: None, b: None, c: get(&RC, inst), val: None
                })
            },
            o if o ==  Some(Opcode::Output as u32) => {
                Ok(Dinst {
                    op: o, a: None, b: None, c: get(&RC, inst), val: None
                })
            },
            o if o == Some(Opcode::Input as u32) => {
                Ok(Dinst {
                    op: o, a: None, b: None, c: get(&RC, inst), val: None
                })
            },
            o if o ==  Some(Opcode::LPro as u32) => {
                Ok(Dinst {
                    op: o, a: None, b: None, c: get(&RC, inst), val: None
                })
            },
            o if o == Some(Opcode::LVal as u32) => {
                Ok(Dinst {
                    op: o, a: get(&RL, inst), b: None, c: get(&RC, inst), val: get(&VL,
inst)
                })
            },
            _ => { Dinst {
                op: None, a: None, b: None, c: None, val: None
                };
                panic!()
            }
        }
    }
}
```

machine.rs

```rust
pub struct UM {
    registers: CPU,
    memory: Memory,
    prog_counter: Option<Box<u32>>
}
impl  UM {
    pub fn new(instructions: Vec<u32>) -> Self {
        UM {
            registers: CPU::new(),
            memory: Memory::new(instructions),
            prog_counter: None
        }
    }
    // counter points to an memory location based on id and offset
    pub fn pcount(&mut self, id: usize, offset: usize) -> Result<(), MachError>  {

        todo!()
    }
    // if $r[C] 6= 0 then $r[A] := $r[B]
    pub fn cdmov(&mut self, inst: Dinst) {
        todo!()
    }
    // $r[A] := $m[$r[B]][$r[C]]
    pub fn sload(&mut self, inst: Dinst) -> Result<(), MachError> {
        todo!()
    }
    // $m[$r[A]][$r[B]] := $r[C]
    pub fn store(&mut self, inst: Dinst) -> Result<(), MachError> {
        todo!()
    }
    // $r[A] := ($r[B] + $r[C]) mod 2 ^ 32
    pub fn add(&mut self, inst: Dinst) {
        todo!()
    }
    // $r[A] := ($r[B] × $r[C]) mod 2 ^ 32
    pub fn mult(&mut self, inst: Dinst) {
        todo!()
    }
    // $r[A] := ($r[B] ÷ $r[C]) (integer division)
    pub fn div(&mut self, inst: Dinst) -> Result<(), MachError> {
```

```
        todo!()
    }
    // $r[A] :=¬($r[B]⋀$r[C])
    pub fn nand(&mut self, inst: Dinst) {
        todo!()
    }
    // Computation stops
    pub fn halt(&mut self, inst: Dinst) {
        todo!()
    }
    // new segment is created with a number of words equal to the value in $r[C],
    // words  initialized to zero
    // the new segment is mapped as $m[$r[B]].
    // A bit pattern that is not all zeroes and does not identify any currently mapped
segment is placed in $r[B]
    pub fn map(&mut self, inst: Dinst) -> Result<(), MachError> {
        todo!()
    }
    //  The segment $m[$r[C]] is unmapped
    // Future Map Segment instructions may reuse the identifier $r[C].
    pub fn unmap(&mut self, inst: Dinst) -> Result<(), MachError> {
        todo!()
    }
    // The value in $r[C] is displayed on the I/O
    //  Only values from 0 to 255 are allowed.
    pub fn output(&mut self, inst: Dinst) -> Result<(), MachError> {
        todo!()
    }
    //  UM waits for input on the I/O device
    // $r[c] is loaded with the input
    // must be a value from 0 to 255
    // end of input has been signaled, $r[C] is loaded with a full 32-bit word in which
every bit is 1
    pub fn input(&mut self, inst: Dinst) {
        todo!()
    }
    //  Segment $m[$r[B]] is duplicated
    pub fn pload(&mut self, inst: Dinst) -> Result<(), MachError> {
        todo!()
    }
    // duplicate replaces $m[0],
    // program counter is set to point to $m[0][$r[C]]
```

```
    pub fn vload(&mut self, inst: Dinst) {
        todo!()
    }
}
```

rumload.rs

```
use std::convert::TryInto;

pub fn load(input: Option<&str>) -> Vec<u32> {
    let mut raw_reader: Box<dyn std::io::BufRead> = match input {
        None => Box::new(std::io::BufReader::new(std::io::stdin())),
        Some(filename) =>
Box::new(std::io::BufReader::new(std::fs::File::open(filename).unwrap()))
    };
    let mut buf = Vec::<u8>::new();
    raw_reader.read_to_end(&mut buf).unwrap();

    let instructions: Vec<u32> = buf.chunks_exact(4)
        .map(|x| u32::from_be_bytes(x.try_into().unwrap()))
        .collect();
    instructions
}
```

main.rs

```
mod machine;
use crate::machine::machine::*;
use crate::machine::dinst::*;
use rum::rumload;
use std::convert::TryInto;

fn main() {
    let input = std::env::args().nth(1);
    let instructions = rumload::load(input.as_deref());
    // println!("{} instructions", instructions.len());
    let mut machine = UM::new(instructions.clone());
    machine.prog_counter = Some(Box::new((*(machine.memory.seg_mem[0]))[0]));

    loop {
        let dinst =
Dinst::disassemble(**((machine.prog_counter).as_ref().unwrap())).unwrap();

        // match on each instruction to respective function and execute it
```

```
        // match &dinst.op {
        //     todo!()
        // }
    }
}
```

Debug testing - error.rs

error.rs

```
use derive_more::{Display, Error};

#[derive(Display, Debug, Error)]
pub enum MachError {
    #[display(fmt = "beginning of a machine cycle the program counter points outside the bounds of $m[0]")]
    OutOfBound,
    #[display(fmt = "word pointed to by the program counter does not code for a valid instruction")]
    Unvalid_Instruction,
    #[display(fmt = " segmented load refers to an unmapped segment")]
    NotFoundLoadSegment,
    #[display(fmt = "segmented store refers to an unmapped segment")]
    NotFoundStoreSegment,
    #[display(fmt = "segmented load refers to a location outside the bounds of a mapped segment")]
    LoadOutOfBound,
    #[display(fmt = "segmented store refers to a location outside the bounds of a mapped segment")]
    StoreOutOfBound,
    #[display(fmt = "unmaps a segment that is not mapped")]
    NotFoundUnmapSegment,
    #[display(fmt = " instruction divides by zero")]
    DivisionByZero,
    #[display(fmt = "instruction loads a program from a segment that is not mapped")]
    NotFoundLoadProgramSegment,
    #[display(fmt = "instruction outputs a value larger than 255")]
    UnvalidOutput,
}
```

Testing - src/lib.rs

To test my program I would use a function similar to the Rumdump lab

Arithmetic

- For every instruction print the value contained in the registers based on the opcode
- Assert that the values expected after the execution are equivalent to the value contained in the registers based on the opcode

Map / unmap

- We would assert that a segment vector of u32s to be unmapped by making its pointer point to NULL is currently pointed to by a pointer.

- We would assert that a segment vector of u32s to be mapped by having a pointer point to it, isn't pointed by anything yet

Load / store

For loads and sores we would assert that the segment to load or store is from a segment with an id outside of the range of our vector of boxes AND different from the segment ids present in our VecDeque containing the unmapped segment ids

Then, we would also assert that the offset is within the length of that segment

Our Error.rs allows us to debug with more ease with meaningful and specific errors

<u>Seg representation, mapped seg ids, invariants:</u>

Or segmented memory is a vector of boxes, each pointing either to null (unmapped segments) or to a vector of a sequence of u32 words on the heap (mapped segments)
Vec<Box<Vec>>

We use push_back to store unmapped seg ids in a VecDeque<usize>, then pop_front to be reuse them

- (First stored id, first reused id)

If our VecDeque of unmapped reusable ids is empty then we create a new vector and push a new Box<Vec> pointing to that vector into our segmented memory (Vec<Box<Vec>>)

Our program counter is an option box pointing to a u32 from a segment offset
If equal to none the machine may fail
prog_counter: Option<Box<u32>>