Amelia Vogel
Yusra Suhail
Lakshinee Rungadoo

Universal Machine Design Document

The representation of segments

We will make an array of our 8 registers like this registers[u32;8], and we will reference the register using registers[i]. We will be using an array because the number of registers are fixed and we can easily just refer to those registers whenever we want.

The memory of UM needs to be a large space so we can easily create a vector of vectors of u32 for that. We will use a mutable vector of u32 for our segments because segments contain words in it. We are using mutable vectors because we don't have to define the size of vectors when declaring them, and we can grow the vectors and remove and insert elements as required.

We will also need a 32 bit program counter which we will use to access the value at the segment and get the instruction from it so that we can perform operations according to the opcode. This counter is just a u32 number which can go up to a maximum of 2^32 to access the address within the segment so we can get the word at the required address.

Invariants for segments:

- We have to make sure that the first segment always has to be 0 because it means to run the program while the other segments have to be unmapped. If the first segment is not zero the program will fail to run. Hence when declaring the vector for segments we will initialize it with 0 to run the program.
- Each segment in the memory has to be reachable by the program counter, because if it is not reachable then the program will not execute. Also, this program counter needs to only execute the instructions which are present and should not go out of bounds. If the counter goes out of bounds, the program will fail. Hence we will have our program counter be a 32 bit number so that it is large enough to handle all the possible address spaces.
- The program counter should always point towards a valid instruction, and if it points to something that is not valid, the machine should fail. We can handle it by using the match arm which checks if the instruction is valid or not.
- **Relationships between unmapped and mapped segments and identifiers is key**

The architecture

We will create a module called **structures** which will store only the structs and enums that we create. This includes an enum for all the opcodes - their numbers and instruction name. The second struct will have the register a, b, c, value and opcode which we will call as instructions. The structs in this module will be public, but we will only use them when we are trying to parse the instructions.

We will have a **main** which will read the input from the command line, will boot it and run the machine. This will then send the input to the **instruction** module that will parse the instructions inside the input. So it only needs to know the instruction module.

The **instruction** module will parse the instructions. It will use the program counter to progress through the segments. It will then extract the instructions from the word, figure out the opcodes, and decide if they are valid or not. Finally it will call the other module which we call **segOperations**, which will do operations according to the opcodes. So it needs to know about all the functions which are going to deal with the opcode. This module also needs to be able to access the **structures** module.

The **SegOperations** will deal with mapping and unmapping of the segment, loading a value or program, conditional move, segmented load and segmented store and all other opcodes. We will have functions which will deal with all of these opcodes separately. The functions directly dealing with opcodes need to be public and will be used by the **instructions** module; the rest will be private.

We will also have a module called **emulation** which can do the emulation from 32 bits to 64 bits. We will probably be using this module in almost all the modules, so the functions need to be public.

The test plan

We will test our program using rumdump that we created in the lab to see what the program is doing and test that if we parsed the instruction correctly. This program just reads the instructions and prints out what the opcodes should do and if our UM does the same thing then we pass the test.

We will also have test cases that can check if we are getting the opcode, register A, B, C and value from the right place because all of them are designated at special places.

Another test we can do is to check that our machine always terminates. We can do this by giving an instruction that takes too long or requires more resources than the machine can handle then the program should halt with a checked run time error.

We will also have test cases that give instructions which are not valid opcodes to test that our machine handles such casualties.