Introduction

In this project, functions which used to solve “.data” part are defined in “datasolver.cpp" and declared in “datasolver.h”, functions which used to simulate the memory and instructions are defined in “simulator.cpp” and declared in “simulator.h”. Users can use “make” to compile and “./simulator test.asm test.txt test\_checkpoints.txt test.in test.out” to run this project.

Big Picture Thoughts and Ideas

The whole project is divided into three stages. First, initialize the memory of 6MB and 35 registers. Second, store the machine codes into the memory beginning with virtual address of 0x400000, and store the static data into the memory beginning with virtual address of 0x500000. The real address is not 0x400000 and 0x500000, so this project use “virtual address = real address – beginning real address + 0x400000” to get the virtual address. Third, according to the machine codes in memory, simulating the execution process of MIPS instructions and changing the contents in registers and memory.

High Level Implementation Ideas

For the first stage, this project uses an array of 128 bytes to simulate the 32 general registers, each register takes up 4 bytes. Then, it uses two pointers hi, lo to the HI and LO registers, use a pointer pcregister to the PC registers. Since virtual address and real address are different, it uses “pcregister(virtual address) = real address(pc) – beginning real address(textStart) + 0x400000” to calculate and store into registers. For the memory of 6MB, it uses “malloc” to allocate 6MB and a pointer to the real address of the start position. Last, it uses “memset” function to set them all to 0, and set genReg[28]($gp), genReg[29]($sp), genReg[30]($fp) to 0x508000, 0xa00000, 0xa00000.

For the second stage, this project scans the “test.txt” file and change each line from string to int, and store into the memory. Then, it scans the “test.asm” file to handle the “.data” part and store static data begin with the virtual address of 0x500000, it uses a pointer dynamicData to record the position where dynamic data beginning.

For the third stage, this project divides all instructions into R, I, J three kinds and uses functions to implement every instruction. Then it uses function pointers to call function according to the function names. Also, it uses maps to store function names and their corresponding opcodes so that it is easy to call functions.

Implementation Details

First, this project uses “malloc” and “memset” functions to initialize the memory of 6MB. The pointer (char\* p) points to the starting location of real address. Then, it uses “malloc” and “memset” functions to initialize all the 35 registers. Let pointer (int\* dp) points to the starting location of static data of real address. Let pointer (int\* pc) points to the instruction which is executing with real address. Let pointer (int\* textStart) points to the beginning of the real memory. Let pointer (int\* pcregister) points to the instruction which is executing with virtual address, so that we can store the virtual address. The relationship between virtual and real address is “virtual address = real address – testStart + 0x400000”. So, it can use this formula to convert between virtual and real address.

Second, when storing the machine codes, it uses “binToint” and “storeText” functions. In “binToint”, it can convert a binary code of string type into an int. In “storeText”, it reads the string line by line from “test.txt” and converts it to an int using “binToint”, then stores this int into the memory from 0x400000(virtual). When storing the static data, it first defines a pointer (int\* dp) points to 0x500000(virtual) so that the static data can store begin from 0x500000(virtual). To handle different types of data, it defines different functions. “ASCII” is used to handle “.ascii” and “.asciiz”, it reads the string character by character and stores it in memory. If the type is “.asciiz”, it adds a “\0” into memory and the string length plus 1. According to the total length, it can find the memory address after storing the data, then return the pointer. “WORD” is used to handle “.word”, it reads the string character by character, when it read a “,”, it stores the cached word into memory and clear the cache. Last, it returns the pointer which points to the address after storing every word. “BYTE” and “HALF” are similar, they consider the char like ‘a’ and digits. According to the total times of storing, it can find the memory address after storing the data, then return the pointer. “solver” function is used to scan “test.asm” file so that it can store “.data” part. It reads the string line by line and it divides the line into three strings by stringstream after deleting comments. According to the second string, it can determine the type of data, and use “ASCII”, “WORD”, “BYTE” and “HALF” to handle the third string. There is one thing to watch out for, getline may read “\n” and “\0” as two chars, so we should replace the “\n” and “\0”(two chars) in the third string with ‘\n’ and ‘\0’(one char). After handling all static data, it returns a pointer which points to the real memory address to begin store dynamic data.

Third, to simulate the executing process, it should implement all instructions’ function. Since instructions have R, I, J three types, we have three types of function pointers: (\*R\_instruction) (int\*, int\*, int\*, int), (\*I\_instruction) (int\*, int\*, int), (\*J\_instruction) (int). For an instruction, it first judges whether it is “syscall” or not, if not, use mask to find opcode. According to opcode, use map to find function’s name so that it can call the function. For R instructions, use mask to find funct, rs, rt, rd and shamt, then use function pointer to call. For I instructions, since there are some I instructions which have the same opcode but different rt, so divide it into two subtypes, then use mask to find opcode, rs, rt, imm and subtype. For J instructions, use mask to find opcode and target, change target into the address that it want to jump, then use function pointer to call. For “syscall”, $v0 can get by genReg[2], according to different $v0, do different operations. For the specific implementations of all instructions and syscall, users can view them in the “simulator.cpp” file.