Project report: MIPS32 Assembler and Simulator

Outline:

1.Purpose of the project

2.Program structure design

3.Implementation details

4.Test and debug

5.Tools and references

1.Purpose of project.

Project 1 is to read a MIPS format assembly source code and generate binary code for further usage. Project 2 will take the code generated in Project 1,

parse them, and execute as they instructed.

To complete project1, we should read the given .asm file from the command line. Then we will request a block of memory in the system like the memory of our simulator. We also have to request a 37 (0-31 for numbered registers,32 for program counter(pc),33 and 34 for hi/lo byte, 35 and 36 for ll/sc status) words space to store our register data.

We also need to divide the .data segment and .text segment, detect functions specified in .text, allocate data and text into the proper address, and store the function address. Then, we can translate instructions into binary code with the address we allocated in the previous steps. The result is directly stored in virtual memory and is invisible to the user.

To complete step 2, we will read binary code from memory, assume the correctness of previous code, and manipulate data and registers exactly as they implied. The result will be printed to screen as well as written into the file name the user-specified if the program calls syscall 1/4/11 to print something. More of file io can be found in the implementation segment.

2.Structure design.

The program is composed of five parts: main, func, dta, oplist and simulator.

a.main

main is the function to control the other programs. It also controls the allocation of virtual memory and file stream. It will open the first parameter of input as a source file and pass the content to func and dta to translate and initialize the memory. It will not open any out file but give the out file name to the simulator.

b.func

func deals with the .text part of a MIPS code. To make it more convenient to deal with functions, we specify a class "func" in the head. The func has four parts: "process\_text" is called by main, take the filestream as input, and calls "write\_func" to write .text part. "write\_func" will detect whether current input is of .text segment, and calls "write" to write each block of codes into a func instance. When the func end, it will add the func into a vector<func> and continue. "write" is called to write each instruction in order and record the size of each func. "modify" is called in three steps above to convert all types of input into the comma-separated format and remove the comment. "addalloc" takes the vector<func> as well as their size as input. Finally, the map contains a function label and the corresponding size is returned to the main.

c.dta

dta deals with the .data part of the input. Also, we specify a class "dta" for it. The dta has two parts. "reformat" is called when input is of type .ascii or .asciiz to combine separated char (like '\', 'n' for '\n' in the input file) to a single char and remove the double quotation marks automatically added when reading the file. the "write\_data" will detect the start and end of .data segment and allocate the data into virtual memory, collect the beginning address of each data as a map and return it to main.

d.oplist

oplist is called by the main, takes the func\_address set and data\_address set generated in the previous stage as input, and translates the instructions. in the head file, we specified different types of instructions. We cluster the functions with the same format of instruction (like add and addu, both are r-type op,$rd,$rs,$rt) into one map, with their name as the key and their opcode/func\_code as value. We also specify a map of numerated registers ($zero-$ra) to their decimal value. oplist has four parts: "select" will take the operate name as input and return its type. "inst2bin" will switch the type of operate name, get the corresponding register name, immediate number, and address of label, and form the zero-one string with "i\_type" and"r\_type". The j\_type instructions are relatively simple so no distinct function for the j\_type parser. Each type of parser will take the opcode,$rs,$rt,$rd,func code in the "inst2bin" and generate a bitset<32> with 31-26 opcode, 21-25 rs, 16-20 rt, 0-15 offset for i\_type/11-15 rd, 6-10 shamt,0-5 func, for r\_type. The zero\_one string will be converted into a hex integer and stored in memory sequentially.

e.simulator

the simulator will do reversely as what we have done in oplist. It will read the integer stored in the memory and switch the parser based on the opcode. There are four kinds of parsers in simulator: "i\_type\_parser","r\_type\_parser","j\_type\_parser" and "rsp\_parser". The last one is for those with a similar structure as r\_type, but different in opcode (including madd, msub, maddu, msubu, mul, clo and clz). For 0xc(syscall), because the options are large, we also have a parser "syscall" for it.

When dealing with load/store or open buffer to write data, we will need to know the real memory address instead of the virtual memory address. "to\_real\_address" takes the virtual address as input and return a void pointer to the corresponding real address. When dealing with signed integers, since sometimes the efficient int length is smaller than the default int size (e.g. 0xffff for -1 instead of 0xffffffff). "sign\_extend" will take a small int as input and extend its most significant bit to a 32bit signed integer.

3.Implementation details

a.memory allocation.

All memory allocations are conducted using arrays. We create an array long vm[] which length is 0x180000 (the space is exactly 0x600000 bytes). The static data segment starts from vm[0x40000], whose address is &vm[0]+0x100000. The dynamic starts 0x100000 over static data. Then we name the address of &vm[0]

as 0x400000. Another data allocation happens when data and texts are given their address. For .data part, .ascii and .asciiz will take celling(length/4) word(s) memory, a .word will take 4-byte space, a .half will take 2-byte space. For funcs, we treat each as an 32-bit binary integer, so each instruction will occupy 4 bytes of space.

However, despite the real or virtual address, the program counter and the jump-related address take the position of int in the array as input. when pc self increased, this means the real address increased by an integer size.

When assigning the function address, the plain text starts at 0, and for the functions coming, each starts at the previous starting plus the previous size. When calculating the jumping address, take the difference of the first line of function a and b, plus the current position in the current function and minus 1. After any instruction has been parsed and executed, the pc increased by 1. For sbrk (syscall 9), just take it as allocate a dynamic data, so the pointer to the new memory address is set to 0x600000(0x200000+0x400000 base). The initial idea is to use malloc to allocate an address, but during the translation from real to virtual address, the segment fault could happen, so we finally use static relative address instead.

b.io

Initially, the io is conducted using fstream and iostream only. But in syscall 13,14,15 it is required to operate a file using file descriptor. Also, when print int, print string, and print char are replaced into 'write to a specified file', using fstream and file descriptor could result in the wrong format. When using the above method to output a file, the result is always in binary format and cannot be read. Therefore when conducting print-related syscall, the ostream is using C format FILE\* pointer and fprintf. To ensure every write-out is in string format, a stringstream ss is used to convert any type of data into a string. To read input from the command line, we use the command line interface for input.

When reading source code input, because the file needs to be processed by func and dta separately, it will be closed and reopen before completed. To read the second file as input, it will be open and closed the instance program exits. To write out the file, it will be opened the instance it is used and closed after a written operation.

c.format file.

Since the input sequence and format could be different among files, as specified in project instruction, the dta and func part have a special mechanism to deal with different input formats. When func gets a filestream, it will not start any process until it finds '.text' in line. Then it decides if the first line of instruction and label in the same line. When it starts process. text, it will exam if '.data' in line to decide whether to exit. If '.data' not in line, it will further looking for the next function's label. If it finds ':' in line, the “write" will break and gives the last line read to write\_func as the next line to conduct the next operation. For .data, the process is similar but without detection of the next label.

As for the input string, the modify will remove all comments and a blank line. Also, it will check if '()' in line, and remove it if true.

4. Test and debug

The test consists of five major parts: io, segmentation, and format, memory allocation, translation, execution.IO test are testing:1. if read/writes success;2. if permission to file has been correctly given;3. if the format is correct.;4. if write to/read from the correct direction.

Test for format include:1.if the string are in homogeneous format(i.e,op,$rs,$rt,$rd);2.if the data/func have been entirely read and stored;3. if the content(i.e., label,content,size) is correct.

The test for memalloc is:1.if the address of each part has been correctly calculated;2.if the size of each data has been correctly assigned;3.if the v\_address has been successfully mapped to real address;4. if the real address access success;5. if the jumping and branching address has been calculated successfully.

The test for translation include:1.if the operating name has been translated correctly;2.if the registered name has been translated correctly;3. if the order of every part of binary code is correct(i.e., $rs,$rd,$rt or$rs,$rt,$rd)?4. if the shamt, offset, or jumping address correct.

The test for execution includes 1.if the type of instruction parsed correctly;2. if the corresponding component correct;3. if the program counter movement correct;4.for detailed instruction, if they are executed correctly.

5.Tools and references

The instruments used in this projects are:

CLion 2020.3.2 for writing code and run code, io, segment, and format, translation debug.

xSPIM and Mars4.5 for executing mips source code and see memory and register status for reference.

edb 1.3.0 for execution debug.

MIPS32® Architecture For Programme Volume II: The MIPS32® Instruction Set by MIPS Technologies ,Inc, to check format and code of instruction, meaning, implementation.

online mips converter to check translation result http://mipsconverter.com/instruction.html?optradio=on