CSC3050 Computer Architecture

Project 1 report

Name: Xiao Nan

Student ID: 119010344

Date: 2021/3/20

The Chinese University of Hong Kong, Shen Zhen

1. Introduction:

In this project I made an assembler which can assemble MIPS program into machine codes and a simulator which can simulate the execution of the machine codes using C/C++ under the environment of Linux 5.10.5-zen1-1-zen, gcc (GCC) 10.2.0, GNU ld (GNU Binutils) 2.35.1 in a x86-64 computer’s memory. Also, I wrote a Makefile to help compile them.

1. Big picture:

In general, the running of a MIPS program contains two parts: Assembling and executing. In the assembling stage, the computer interprets the .text part of the MIPS program as machine codes and places them in the “text segment” area of the main memory, then the data encoded in .data segment in the program would be put into the static data area of main memory, and the end of the static data area would be identified as the start of the dynamic data area. After that, the next stage, execution stage, can be divided into five parts: Instruction fetch, instruction decode, execution of instruction, memory reference and writing back. In instruction fetch and decode stage, the computer would maintain a program counter and during each clock cycle (regardless of pipelining), the program counter will fetch one instruction (instruction = memory[PC]) and move to the next instruction (PC += 4 bytes), and then the fetched instruction would be decoded (separated into different parts) into different parts of machine codes according to the instruction type, then different decoded parts would be sent to different units in computer to execute the program: For example, register-related parts (rs, rt, rd) would access the register file to read the value stored in certain registers, while control-related parts (op and funct field) would send control signals to the two control units (main control and ALU control) to manage the operation of other hardware units in computer. In the instruction execution stage, ALU would perform arithmetic operations (like add for addition/sub for subtraction) according to the control signal sent from control unit. In the memory-referencing stage, the computer may access memory using the results of ALU and load/store data depending on the instruction. For example, a lw/sw instruction would require calculating the effective address based on offset and the base address in a register, then the result of ALU, the effective address, should be used to find the corresponding memory location. However, if the instructions (like add/sub) do not require referencing memory, the ALU result would be transmitted through this stage without passing through memory. In the writing back stage, either the result of referencing memory or the result of ALU calculation would be written back to the register specified in the decoded instruction. Then this clock cycle ends and when the next clock cycle is triggered, the next instruction is fetched and decoded and the above process would be repeated. Finally, the program terminates either when an exception (like segmentation error) takes place with abort() called because of badly organized MIPS codes or the exit() system call is invoked indicating normal termination.

1. Implementation ideas:

I break down the problem into three parts mainly: Assembling, initializing and executing module. In the assembling module I scanned the instructions twice. first recording the location of all labels and second extract key information in order to encode the instructions into machine codes. After that, an intermediate file containing the machine codes of the instructions would be outputted. In the initializing module I allocated 6MB space for the simulated memory. I placed the .data part of the MIPS program to the simulated static data area (located 1MB above the starting location of simulated memory) and the assembled code read from the intermediate file to “text segment” part (starting from the beginning of the simulated memory). I allocated a program counter which captures 4 bytes pointing to the first instruction. I allocated 4-bytes space for the each of the 32 32-bit registers as simulated register file. In execution module I judge the instruction type based on the instruction op code and funct code and then implement corresponding operations using C/C++ or UNIX system functions. Besides, I wrote a Makefile for the convenience of compilation in different computers.

1. Implementation details:

In assembler, I defined 3 bit fields with the size of 32 bits each but containing different bit patterns to store the different types of instructions (R, I and J format) and to print them out. I used a map with structure {register name: register ID} for the convenience of looking up register ID and encode them into the machine code. I used another map with structure {label: difference to the first instruction in the number of instructions (when using, just add that difference to the base address)} to store the labels and its corresponding address. In simulator, I used another map with structure {ID: register pointer} for the convenience of referencing registers using their ID. I defined two unions for the convenience of zero-sign extension (by putting the target value in LSB of the union containing less bits and extract the result with more bits). I used integer pointers (int\*) to indicate pointer to instructions and registers for the convenience of value assignment and type casting.

1. Conclusion:

In this project I wrote an assembler and a simulator which can simulate executing a MIPS program file on a 64-bit computer. I divided the project into three parts: Assembling MIPS program file, initializing memory and registers as well as executing machine codes. I used multiple data structures like bit fields and unions to make the implementation more efficient and robust.