The Chinese University of Hong Kong, Shenzhen

CSC 3050

Computer Architecture

Report for Project 2: MIPS Simulator

Author:
Qiu Weilun

Student Number:
120090771

March 20, 2022

Contents

1 Big Picture of MIPS Simulator **2**

2 High-Level Implementation Idea **2**

3 Detail Implementation **3**

3.1 Implementation of Recognization Process 3

3.2 Implementation of System Call 4

4 Usage of Files **4**

1 Big Picture of MIPS Simulator

In this project, the major task is to simulate the execution of a program. Therefore, the general design of the implementation should follow the principle of machine cycle. The machine cycle will follow the following process:

1. The computer loads the instruction which the PC register is now pointing at.
2. The computer increase the variable PC by 4, which indicates the next instruction.
3. The computer execute the instruction which is loaded from the PC register.

Therefore, in the execution phase of this project, the above process would be strictly followed in order to ensure the correctness of the execution.

Before the execution phase, the memory of the computer should also be prepared. This is refered to as the preparation phase. The starting point of the memory in this project is set to be 0x400000. The first segment of the memory allocation is the **text segment** which stores the information of the executable binary code. This statement starts from 0x400000 and ends at 0x500000, with a memory size of 1MB. After that, the **static data segment** will start at the end of the previous text segment, with the address of 0x500000. This segment stores all the data shown in the ".data" segment in the original MIPS code file. The storage of static data should always follow the alignment principle. The **dynamic data segment** will start at the end of the static data segment. This area of the memory will be used for dynamic memory management(syscall 9). Apart from the above 3 parts, the memory also has the stack segment which is used for function calls. The classification of the address of each function call will be recorded by two registers, the stack pointer($sp) and the frame pointer($fp). The stack segment will start from the highest address which is 0xA00000 and expands to the lower address. The total size of the simulated memory would be 6MB.

Beside the memory allocation, the allocation of register would also be simulated. Each register has the size of 4 bytes. The number of registers is 35 including 32 general purpose registers, the PC register, the HI register and the LO register. Therefore, 140 bytes of memory is needed.

2 High-Level Implementation Idea

For memory and register allocation at the beginning of this project, the malloc() method in C++ would be used to create these two areas of memory. After that, the values of the registers would be initilized. The original values of the 4 register

$pc, $gp, $sp and $fp would be set as 0x400000, 0x508000, 0xA00000 and 0xA00000. The other register would all be initialized as 0.

After all the values of registers and memory allocation are set, the file which stores the executable binary code is scaned through. Each line of the instruction would be stored in a 4-bytes (32-bits) space in text segment. After this task is finished, another file which consists of the original MIPS code is scaned in order to find out the ".data" segment and store all the static data of the program in the static data segment.

In the execution phase, the procedure can be seperated into 3 steps. First of all, the instruction's address store in the pc register would be loaded. The second step is to recognize the type of the instruction, the operation that the instruction intended to conduct, and the register used in the operation. The last step is to conduct the correponding operation. For the first step, a pointer would be used to access the machine code. Then, the recognition part would be conducted in the following way. The first 6 bits (operation code) of the instruction will be taken out first and check whether they are all zero. If yes, then the instruction belongs to R-type instruction. Otherwise, the instruction belongs to I-type or J-type. For R-type instruction, the last 6 bits are taken out because these 6 bits are function code in R-type which determines the exact operation. For I-type and J-type instruction, the operation can be determined direcly by the operation code. The detail implementation of conducting each of the operaiton will be implemented in the R_type.cpp, I_type.cpp and J_type.cpp three files.

Another important component in the execution phase is the system call. The service provided by a system call is determined by the value store in the register $v0. Different services may perform different operations. For instance, the total proram will be terminated when service 10 is called.

3 Detail Implementation

3.1 Implementation of Recognization Process

Since the operation of each instruction and their operation code or function code has a one-to-one mapping relationship, three maps are created in order to make used of this kind of relationship. The map for R-type instructions uses function codes as keys and operation name as values. The maps for I-type and J-type use operation codes as keys and operation names as values. Therefore, the whole procedure would be first look at the operation code to check which type the instruction belongs to. Then figure out what is corresponding operation by looking up the three maps.

3.2 Implementation of System Call

Since the service provided by a system call is depends on the value in register $v0, a switch structure in C++ is used in order to due with the cases effectively. Moreover, some of the services such as service 13 (open), service 14 (read), service 15 (write), and service 16 (close) are follwing the Linux system call style. Therefore, these system calls are implemented using Linux API.

4 Usage of Files

The main procedure is conducted in the Machinecycle.cpp. The I_type.cpp, J_type.cpp and R_type.cpp are the detailed implementation of all the instructions. The Dump.cpp file is for memory dumping. The register dumping has already been implemented in the Machinecycle.cpp. Finally, the simulator.cpp file is for testing.