Program 2 Instruction

Overview

In program 1, you have written an assembler for MIPS Assembly Language, which you take an MIPS code input and convert it to its corresponding machine language (binary) code. Program 2 is called a MIPS Simulator, in which you will take a MIPS code input and simulate executing the code. For example, if an adder MIPS code is given to you, your first program should translate it into a binary code file, and your second program should prompt the user to enter two integers and output the correct answer.

Logic

Unlike the first program in which you are allowed to achieve the goal however you wanted to, you will have some restrictions this time. That is, you will execute the MIPS code just like how a machine does it. The logic behind this program is fairly easy if you understand how machine works. Recall the basic machine cycle, where you load an instruction, increment PC, and execute the loaded instruction. You will do the exact same thing.

With that being said, you will need to:

1. Simulate memory. In your C/C++ program, allocate a block of memory of 6MB, which should be sufficient. This is used to simulate the main memory. For more details of simulating main memory in case you forgot, refer to page A20-A22 of the book (Appendix A). In our case, reserve 1MB for text section. You are going to map the memory address of the block you allocated to 0x00400000, which is a 32 bit address. When you do operations that involve memory, you simply reversely map the 32 bit address back to 64 bit, for access in your computer's memory. For example, you called malloc for allocating a 6MB of memory, it returned 0x100000000022000, which is a 64 bit address. Then, you give it a name of 0x00400000, which is a 32 bit address. After you do this, everything else will just act normal. In your MIPS when you want to access some memory address, you will simply calculate the offset of that address from 0x00400000, and add that offset to 0x100000000022000. In short, you will give the 64 bit address a 32 bit "name".
2. Allocate and maintain memory spaces to simulate the regular 32 registers. Correctly use them, including $fp, $sp, etc. Since we are using a mapping from 64 bit address to 32 bit address, the address can be put in a 32 bit register. You can allocate 4 bytes for each register.
3. Maintain a PC, which indicates the next instruction to load.
4. Maintain a code section in your simulated memory, where the assembled MIPS machine code is put.
5. Maintain a data section in your simulated memory, where the pre-defined data in MIPS code (.data section) is put. Located on top of text section. For dynamic data, it grows towards higher memory address. For the data you read in from .data section, you will line them up in order in data segment of your simulated memory. The first piece of data in .data section of MIPS code will be put at the lowest memory address of your data segment. The rule to put data is each piece of data is going to occupy the full block (4 bytes) even if it used only part

of a block. For example, a string of length 11 is using 11 bytes in memory, and put at address 0x0. Then it used up two full blocks (8 bytes), and 3/4 of the third block, which starts at 0x8. The next piece of data should not be put at address 0xB, instead, it should be put at address 0xC.

6. Maintain a stack section in your simulated memory, which grows towards lower memory address.
7. Simulate what each instruction does using C/C++. This could be done by writing a function for each instruction.
8. Simulate the execution. You are expected to follow the machine cycle. Judge which instruction it is, call the function.
9. You need to support syscall instruction (in your assembler and simulator), which is down below. You also need to support SPIM syscalls below.

Grading

Instruction/syscall execution - 40%

each wrongly implemented instruction/syscall will result in 2% deduction.

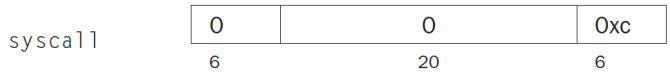
Memory and register simulation - 30%

Machine cycle - 10%

Coding style/comments - 10%

Report - 10%

System call



Register `$v0` contains the number of the system call (see Figure A.9.1) provided by SPIM.

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

FIGURE A.9.1 System services.