INSTRUCTION-LEVEL MIPS SIMULATOR

Introduction

A C program that is an instruction-level simulator for a limited subset of the MIPS instruction set. This instruction-level simulator will model each instruction's behaviour, allowing the user to run MIPS programs and see their outputs. In later labs, you will use this simulator as a reference to verify that your later labs execute code correctly.

The simulator will process an input file that contains a MIPS program. Each input file line corresponds to a single MIPS instruction written as a hexadecimal string. For example, 2402000a is the hexadecimal representation of addiu $v0, $zero, 10. We will provide several input files. But you should also create additional input files to test your simulator more comprehensively.

The simulator will execute the input program one instruction at a time. After each instruction, the simulator will modify the MIPS architectural state: values stored in registers and memory. The simulator is partitioned into two main sections: the (1) shell and the (2) simulation routine. Your job is to implement the simulation routine.

In the src/ directory, we provide two files (shell.c and shell.h) that already implement the shell. There is a third file (sim.c) where you will implement the simulator routine – this is the only file that you are allowed to change.

The Shell

The purpose of the shell is to provide the user with commands to control the execution of the simulator. The shell accepts one or more program files as command line arguments and loads them into the memory image. In order to extract information from the simulator, a file named dumpsim will be created to hold information requested from the simulator. The shell supports the following commands:

1. go: simulate the program until it indicates that the simulator should halt. (As we define below, this is when a SYSCALL instruction is executed when the value in $v0 (register 2) is equal to 0x0A.)

2. run : simulate the execution of the machine for n instructions.

3. mdump : dump the contents of memory, from location low to location high to the screen and the dump file (dumpsim).

4. rdump: dump the current instruction count, the contents of R0 – R31, and the PC to the screen and the dump file (dumpsim).

5. input reg num reg val: set general purpose register reg num to value reg val.

6. high value: set the HI register to value.

7. low value: set the LO register to value.

8.? : print out a list of all shell commands.

9. quit: quit the shell.

The Simulation Routine

The simulation routine carries out the instruction-level simulation of the input MIPS program. The architectural state includes the PC, the general-purpose registers, and the memory image. The state is contained in the following global variables:

```
#define MIPS_REGS 32

typedef struct CPU_State {

        uint32_t PC; /* program counter */

        uint32_t REGS[MIPS_REGS]; /* register file. */

        uint32_t HI, LO; } CPU_State; /* multiplier HI and LO regs. */

CPU_State STATE_CURRENT, STATE_NEXT;

int RUN_BIT;
```

Furthermore, the simulator models the simulated system's memory. You need to use the following two functions, which we provide, to access the simulated memory: uint32_t mem_read_32(uint32_t address); void mem_write_32(uint32_t address, uint32_t value); Note that in MIPS, memory is byte-addressable. Furthermore, we will implement a little-endian architecture. This means that machine words (32 bits) are stored with the least-significant byte at the lowest address, and the most-significant byte at the highest address. To implement loads and stores of 16-bit and 8-bit values, you will need to use these 32-bit memory access primitives (hint: be sure to modify only the appropriate part of a 32-bit word!). In particular, you should call mem read 32 and mem write 32 with only 32-bit-aligned addresses (i.e., the bottom two bits of the address should be zero). The simulator skeleton that we provide includes an empty function named process instruction () in the file sim.c. This function is called by the shell to simulate one machine instruction. You have to write the code for the process instruction () to simulate the execution of instructions. You can also write additional functions to make the simulation modular.

Implementation

J	**JAL**	**BEQ**	**BNE**	**BLEZ**	**BGTZ**	**ADDI**
ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
LB	LH	LW	LBU	LHU	SB	SH
SW	BLTZ	BGEZ	BLTZAL	BGEZAL	SLL	SRL
SRA	SLLV	SRLV	SRAV	JR	JALR	ADD
ADDU	SUB	SUBU	AND	OR	XOR	NOR
SLT	SLTU	MULT	MFHI	MFLO	MTHI	MTLO
MULTU	DIV	DIVU	SYSCALL			

Note that for the SYSCALL instruction, you only need to implement the following behaviour: if the register $v0 (register 2) has value 0x0A (decimal 10) when SYSCALL is executed, then the go command should stop its simulation loop and return to the simulator shell's prompt. If $v0 has any other value, the instruction should have no effect. No registers are modified in either case, except that PC is incremented to the next instruction as usual. The process instruction () function that you write should cause the main simulation loop to terminate by setting the global variable RUN BIT to 0.

The accuracy of your simulator is your main priority. Specifically, make sure the architectural state is correctly updated after the execution of each instruction. We will test your simulator with many input programs (some provided with the handout, some not) to ensure that each instruction is simulated correctly.

In order to test that your simulator is working correctly, you should run the input programs we provide you with and also write one or more programs using all of the required MIPS instructions that are listed in the table above, and execute them one instruction at a time (run 1). You can use the rdump command to verify that the state of the machine is updated correctly after the execution of each instruction.

While the table appears to have many instructions, there are only a few unique instruction behaviours with some minor variations. You should tackle the instructions in groups: R-type ALU, I-type ALU, LW, SW, Jump, Branch, and so on.

However, unlike the manual, we will implement our architecture without “branch delay slots.” We will talk more about branch delay slots in class. For Lab 1, this means that branch instructions can update NEXT STATE.PC directly to the branch target when the branch is taken. Furthermore, “jump-and- link” instructions (JAL, JALR, BLTZAL, BGEZAL) store PC + 4 in R31, rather than PC + 8 as specified in the manual in these instructions’ descriptions. In this lab, you do not need to handle overflow exceptions that can be raised by certain arithmetic instructions (e.g., ADDI).

Finally, note that your simulator does not have to handle instructions that we do not include in the table above, or any other invalid instructions. We will only test your simulator with valid code that uses the instructions listed above.