1 Purpose

This programming assignment is intended to help you understand (i) how an assembler translates assembly language programs to machine language, and (ii) how a simulator interprets machine language programs.

This assignment has 2 parts. In the first part, you will modify the assembler part of SPIM to support a few additional MIPS-I like instructions. In the second part, you will modify the interpreter part of SPIM to support the additional MIPS-I like instructions.

The following table describes the additional MIPS-I like instructions.

<table>
<thead>
<tr>
<th>Assembly Format</th>
<th>Pseudo-op/Format</th>
<th>Opcode (binary)</th>
<th>Func (binary)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>lsb</em> rd, rs</td>
<td>pseudo-op</td>
<td></td>
<td></td>
<td>Extract the least significant bit of the value in register rs, and place it in register rd</td>
</tr>
<tr>
<td><em>nand</em> rd, rs, rt</td>
<td>R-type</td>
<td>000000</td>
<td>101000</td>
<td>NAND the values in registers rs and rt, and store the result in register rd</td>
</tr>
<tr>
<td><em>push</em> rs</td>
<td>R-type</td>
<td>000110</td>
<td></td>
<td>Decrement <em>sp</em> register $29 by 4, and then copy the 32-bit value in register rs onto new top of stack</td>
</tr>
<tr>
<td><em>pop</em> rd</td>
<td>R-type</td>
<td>000111</td>
<td></td>
<td>Copy the 32-bit value on the top of stack to register rd, and then increment <em>sp</em> register $29 by 4</td>
</tr>
</tbody>
</table>
2 Procedure

2.1 Download SPIM Source Code to a Unix/Linux System

The first part of the project is to copy the SPIM source code from
http://www.cs.wisc.edu/larusk/spim.html to a Unix/Linux system. Then do the following sequence of steps:

1. gunzip spim.tar.gz
2. tar -xvf spim.tar
3. cd spim-6.4
4. Configure
5. setenv CC gcc
6. xmkmf
7. make

After this, you should be able to run the newly made spim by typing ./spim &; if you type xspim & instead, it might run an already existing xspim executable, which is not what you want in this assignment.

2.2 Modify the Assembler

The second part of the project is to modify the assembler part of SPIM to include the 4 new assembly language instructions. The SPIM assembler is written by specifying the grammar in the yacc language. You don’t have to study the yacc language; you should be able to do the project by looking at how existing instructions have been specified. Take a look at the contents of op.h, and see how existing AL instructions have been specified there. This file specifies the list of opcode/assembler tokens that can be read by the SPIM assembler. You will see that there is a row for each AL instruction, and that the rows are ordered alphabetically by the mnemonic. Each row specifies an opcode name, its token name, instruction type, and the instruction bit pattern template. You will have to add a row in op.h for each new AL instruction; remember that the overall list should still be in alphabetical order. To do this you have to assign a token name, and determine the instruction type and template.

Instruction Type: The purpose of this field is to tell the parser what to expect after seeing the opcode field. For assembler directives, the instruction type is ASM_DIR. For pseudo-ops, the instruction type is PSEUDO_OP. Notice that SPIM uses a more elaborate classification of the 3-format classification we used based on instruction encoding (R-type, I-type, and J-type). You have to determine the instruction type for each of the 4 new AL instructions. The instruction type for lsb and and are quite straightforward. The push instruction and pop instruction come under two different types; you can figure this out by going through the list of MIPS-I AL instructions that explicitly specify a single register operand.
Instruction Bit Pattern Template: The purpose of this template is to construct the instruction bit pattern of the machine language instruction. For psuedo-ops and assembler directives, this template is not useful, and so is specified as -1. For the regular instructions (the ones that are also present in the ISA), the template is a 32-bit value obtained by assigning a value of zero to the relevant fields of the instruction. For instance, the template for `lw—0x8c000000—is obtained by setting its rs, rt, and offset fields to zero, and retaining its opcode field value (100011). Similarly, the template for `add—0x00000020—is obtained by setting its rs, rt, and rd fields to zero, and retaining its opcode field value (000000) and func field value (100000).

Now, take a look at `parser.y`. It has two parts: (i) the list of tokens and (ii) rules for interpreting the tokens. The list of tokens are the same as those defined in `op.h`; so you will have to include the 4 new tokens that you assigned in `op.h`. Although this list need not be alphabetical, for this assignment, you have to insert them alphabetically.

The second part of `parser.y` gives the rule for interpreting each token. So you have to write 4 new rules, one for each new token. Notice that in the rules, `$1, $2, etc$ stand for the tokens scanned from the assembly language program. `$1 stands for the opcode, $2 stands for the first operand specified, and so on. Notice also that in the rules, a register is specified by writing only its number. Thus, 0 stands for register $0$. For a psuedo-op token (those ending with POP), the rule involves generating one or a sequence of machine language instructions to implement the pseudo-op. For instance, for the `YROL POP` token (which corresponds to the psuedo-op `rol` which specifies rotate left), the rule specifies creating a sequence of 4 regular instructions (`subu, srlv, sllv, and or`).

2.3 Modify the Simulator

The third part of the project is to modify the simulator part of SPIM to include the 3 new machine language instructions. You can do this by modifying `run.c`. Follow the example of existing instructions.