Question 5 (30 points): When a MIPS assembly program must operate iteratively with the terminal input/output, it cannot be executed in the xspim simulator. Debugging such a program is difficult because the insertion of code to inspect the state of the processor is cumbersome. In this question we will work toward a functionality to make debugging easier. Assume that someone wrote a subroutine called `DumpState` that prints the value of all MIPS registers. The purpose is to help with the debugging of programs that are run in spim (as opposed to xspim). In this question you will write two functions: `FindJal` is a function that scans the binary representation of a MIPS program, in reverse order, and returns the address of the next `jal` instruction found; and `InsertDump` inserts a call to `DumpState` before and after every function call in the program. Inserting instructions in a program requires the fixing of all jump and branch instructions. However, assume that the fixing of these instructions will happen later and you are not concerned about it.

While writing both functions you need to follow the MIPS register-saving calling conventions. You are allowed to use any MIPS instructions, including SPIM pseudo instructions. But you are NOT allowed to use any instruction that takes as a parameter a constant that is larger than 16 bits. Even though SPIM allows you to use larger constants, the MIPS assembly does not. We want you to be thinking about the processor and not about the simulator.

1. (**10 points**) Write the MIPS assembly code for a function called `FindJal` that receives as parameters (1) the address of an instruction where the search should start and (2) the address of the first instruction in the program (which is the last instruction that should be searched if a `jal` is not found). As `FindJal` searches in reverse order, it should return the address of the first `jal` instruction that it finds. If no `jal` instructions are found, it should return the value -1. Here is the interface for `FindJal`

   Parameters:

   - `$a0`: address of the instruction where the search is to start
   - `$a1`: address of the first instruction in the program (last instruction to be inspected)

   Return value:

   - `$v0`: address of the first `jal` instruction encountered
   - `$v0` = -1 if no `jal` instruction is encountered

Space to write code for `FindJal` function

2. (**20 points**) Write the MIPS assembly code for the function `InsertDump`. The parameters for this function are the addresses of the first and last instructions in the program, the number of `jal` instructions in the program, and the address of the subroutine `DumpState` that was written by someone else.

   `InsertDump` must create a binary representation for a `jal DumpState` instruction. The opcode for a `jal` instruction is `000011` and the format is identical to the format of a jump instruction.

   `InsertDump` has to create a new version of the input program that has a call to `DumpState` before and after each `jal` instruction found in the input program. `InsertDump` **must** call `FindJal` to locate the `jal` instructions in the program.

   Hint: Given that `FindJal` searches the binary code in reverse order, it shall be simpler for `InsertDump` to also traverse the code in reverse order. The number of `jal` instructions in the program, provided as a parameter to `InsertDump`, should facilitate the moving of all instructions that must be moved to make room for the new calls to `DumpState`.

		0x4000 0000	add $a0, $0 $0
		0x4000 0004	lui $a1, 0xFFFF
		0x4000 0008	srl $a2, $t1, 8
		0x4000 000C	xor $a3, $t1, $t2
0x4000 0000	add $a0, $0 $0	0x4000 0010	jal DumpState
0x4000 0004	lui $a1, 0xFFFF	0x4000 0014	jal bar
0x4000 0008	srl $a2, $t1, 8	0x4000 0018	jal DumpState
0x4000 000C	xor $a3, $t1, $t2	0x4000 001C	beq $v0, $0, Done
0x4000 0010	jal bar	0x4000 0020	xor $a0, $v0, $0
0x4000 0014	beq $v0, $0, Done	0x4000 0024	nor $a1, $a0, $v0
0x4000 0018	xor $a0, $v0, $0	0x4000 0028	jal DumpState
0x4000 001C	nor $a1, $a0, $v0	0x4000 002C	jal baz
0x4000 0020	jal baz	0x4000 0030	jal DumpState
0x4000 0024	or $v0, $v0, $v1	0x4000 0034	or $v0, $v0, $v1
0x4000 0028	jr $ra	0x4000 0038	jr $ra

(a) Original Code (b) Transformed Code

Figure 1: Code before and after insertion of instructions.

   The example shown in Figure **??** is only intended to illustrate how `InsertDump` is supposed to work — Your code for `InsertDump` must work with any valid MIPS binary input. After the transformation the binary code should start at the same address as the original code.

   Parameters:

   - `$a0`: address of the first instruction in the program
   - `$a1`: address of the last instruction in the program
   - `$a2`: number of `jal` instructions in the program
   - `$a3`: address of the subroutine `DumpState`

Return Values:

- none

Side Effect:

- Calls to `DumpState` are inserted before and after every `jal` instruction.

Invariants:

- The four most-significant bits of the address of all instructions in the program and of the address of the DumpState function are the same. This invariant remains true after the insertion of the calls to DumpState.
- There is enough free space below the binary code to accommodate the code growth caused by the insertion of the calls to DumpState.

Space to write code for `InsertDump` function