Project Compiler - Code Generation

Out: 23 Nov 2011
Due: 16 Dec 2011, 11:59PM

0 Motivation

You have now created a parser and a semantic visitor, catching any errors that may be in the Blaise code. All that's left now is to generate assembly code for your AST.

1 Requirements

- Modify the provided Java code such that it realizes the described specification.
- Hand in your completed compiler by the final deadline.
- Hand in test programs that FULLY test your compiler.
- Your handin must include a `README` file that
  - describes the bugs in your program (or asserts their absence)
  - describes any extra credit you attempted
  - lists any changes made to the support code (i.e. classes added or edited)
  - explains your test program(s)
  - explains anything else the grader needs to know
- Any extra credit should be implemented in a separate version of your compiler; we expect your handin to contain a fully functioning compiler with no implemented extra credit in addition to any extra-credit compiler you wish to submit.

2 Details

Your generated code should produce a program for a stack-based machine. This specification will actually make your implementation much more straightforward. Refer to class notes regarding stack machines and make sure to read the section on Code Generation and stack machines later in this handout.

2.1 Code Generation

Once you have a semantically-correct syntax tree, the next step is to generate corresponding MIPS assembly code.

To do this, you will implement a *Code Generating Visitor.*

2.2 Support

2.2.1 Class overview

The following is an overview of the classes used for this final portion of the assignment.

- **Register** This enumeration contains values for all the usable registers on a MIPS machine, referenced by name. These are passed to the `MIPSCodeGenerator`.
- **Procedure** You will use this class again in code generation to determine where variables have been placed on the stack.
- **MIPSCodeGenerator** Class that supports code generation. `MIPSCodeGenerator` contains a lot of simple methods, each of which writes a MIPS instruction to an instance variable of type String. At any point, this class can write the currently generated code to the screen or a file. This class can also give you unique labels, which you will need for branching and procedure calls.
- **CodeGenVisitor** As in semantic analysis, you will use the visitor pattern to traverse your AST and generate assembly code.

2.3 Memory Allocation and access

During semantic analysis, you determined how much space was needed for each global variable and used the `Procedure` class to generate the location of each local variable and argument on a procedure's frame.

Now we must make use of this data to allocate memory for our variables.

The global variables are simpler to take care of, as they should be declared in the data section of the generated code.

To handle local variables, one must first understand how our frames will work.

2.3.1 Frames and Procedures

As explained in section 2.3.4 of the semantic analysis handout, we will be making use of frames for this assignment. We described the frame layout in the previous handout, so be sure to reread that section.

The following describes the process of setting up and destroying frames.

To set up a frame:

- The caller must:
  1. Allocate space for and push any arguments onto the stack in the order they occur in the procedure call.
- The callee must:
  1. Allocate enough space on the stack for all of the local variables (excluding arguments).

2. Push registers \$s0 through \$s7, the frame pointer, and the return address to the stack. Note that \$s8 and \$fp are different names for the same register, so Mipscope's Register View will show \$s8 instead of \$fp. Nevertheless, you should refer to the register as Register.FP in your compiler to remind yourself of this register's importance. This means that we will push 10 registers total in this step. The support code assumes all 10 registers are pushed, even if they don't need to be saved.
3. Set both the frame pointer and the stack pointer to refer to the first free word on the stack after these saved registers.

To destroy a frame and return from a procedure:

- The callee must:
  1. Store the return value in a register. Remember that `return 0;` will be implicitly called if the end of a procedure is reached before a value is returned.
  2. Restore the saved registers. This includes the frame pointer and return address.
  3. Depending on how you handled the initial space allocation, you may need to reduce the stack pointer to effectively free the space allocated to the local variables and arguments.
  4. Push the return value to the stack.
- The caller must:
  1. Pop the return value from the stack.
  2. Ensure the stack pointer is now where it had been at the start of the procedure call.

2.3.2 Accessing local variables

Assuming you have set up your frames correctly, the `Procedure` class will again come in handy for accessing the variables you have stored on your frame. Each local variable and argument has an offset from the frame pointer, which you can get by calling the `lookup()` method on that variable. Note that you must keep track of which procedure you're in so you know which `Procedure` object to use to look up the variable with.

Once you have the offset of the variable in question, you can access the variable by adding your frame pointer to this offset.

2.3.3 Example

To see how code generation works in practice, consider generating the code for output like

```
output(4);
```

Remember that this is not a procedure call, since `output` is its own instruction. The code generation for this assignment is done in the `handleNodeInstrOutput()` method of the `CodeGenVisitor` class. We have provided this method for you.

In order to perform an output, one will need to create the code to calculate the value of what is being output. In this case, it is simply an integer, 4, but in general, one will need to have the

`CodeGenVisitor visit()` the `NodeInstrOutput`'s `NodeExpr` before you create the code for the actual assignment. The results of a `NodeExpr` should be pushed onto the stack (see below), so you will be able to pop that result from the stack to a register with `genPop()` and output it using a syscall by using the `genOutput()` method of the `MIPSCodeGenerator` class.

2.3.4 Register allocation and the stack

Other compilers use solely the registers to store intermediate values when evaluting large expressions. This approach would produce speedy code, but it would be very feasible to think of complicated-enough expressions that would cause the compiler to run out of registers, and the algorithms to do efficient register allocation can be very complicated[1]. It is also the intention of the project to show how a stack-based system, such as Java, functions. With that in mind, we will be using the stack to keep all intermediate values. Registers will be used solely for moving between memory and the stack, and the stack and the ALU.

This approach relieves much of the strain on the registers. At the end of processing each node, there should be no important data held in the registers; all information should be left on the stack. This also means that while creating the code for any node, all registers (aside from $sp, $ra, the frame pointer and some others) are free for use. You should be able to do all operations with only a few registers, so we suggest using the $t registers (or the $s registers).

There is a `Register` enumeration that contains values for each of the registers on the MIPS machine — all the `MIPSCodeGenerator` methods take a `Register` value when referring to a register (e.g. `codegen.genPush(Register.S2)`).

In order to facilitate the use of the stack, the `MIPSCodeGenerator` has `genPop()` and `genPush()` methods.

2.3.5 Using the *MIPSCodeGenerator*

To make code generation less tedious, there is the `MIPSCodeGenerator` class that generates assembly code for you (e.g., compiler directives, MIPS assembly instructions, and syscalls).

For example, the `genMove()` method will write a single MIPS `move` instruction to the generated program, using the registers you specify. `genMove(Register.S0, Register.S1)` will add the string `move $s0, $s1` with a trailing newline to the generated code.

At any point, you may tell the `MIPSCodeGenerator` to either `writeToConsole()` or `writeToFile(filename)` the currently generated code. This could be very useful for debugging.

A `MIPSCodeGenerator` object simply writes declarations in the order in which they are given to it. This means that you must explicitly tell it when to write directives such as ".data", ".text", and "main:". Also, all your .data (global variable) declarations must be done before your program code is put in.

Although you should be sure to look through the `MIPSCodeGenerator` code on your own so that you have a sense of what methods you will have at your disposal, a couple points are worth noting here.

[1]It is worth noting that the code produced by your compiler and the demo will be a lot longer than a normal stack-based system would use. We are using a register-register machine to simulate a stack machine. On a stack machine, most of the pops and pushes would be implicit.

There are two `get...Label()` methods in the `MIPSCodeGenerator` class. `getProcLabel()`, which is used to generate a unique label for each procedure, takes as an argument the procedure name and returns a label of the form `proc_name`. If called multiple times with the same input, it will return the same label. `getNextLabel()`, which will be used for all other labels, takes no arguments and returns a distinct label each time. Examples are `label2` and `label31`.

Any time a `MIPSCodeGenerator` method expects a label, you should pass in one of these labels.

Both the `genSaveRegisters()` and the `genRestoreRegisters()` methods handle all of the registers we expect you to push and pop - all $s registers, the frame pointer, and the return address. You should use these methods.

2.3.6 Other Notes

Short-circuiting: Boolean-valued expressions should be short-circuited. That is, if the left-hand expression in an OR is true, the rest should never be evaluated (since the whole expression is guaranteed to be true). Likewise, if the left-hand expression of an AND is false, then the whole expression *must* evaluate to false, so the rest of the expression should not be evaluated. Please note that this means boolean expressions will be evaluated **left to right**.

Testing: The time needed to write some excellent tests, like a nontrivial recursive algorithm, will definitely pay you back in debugging time and knowing when your compiler is done.

3 Extra credit

Please see the Compiler Extra Credit handout for the numerous extra credit possibilities for this assignment. **Make sure your basic functionality works before attempting extra credit.**

4 Grading

As stated in the parser handout, your program will be graded primarily based on its adherence to and fulfillment of the assignment specifications. However, your work will also be evaluated in terms of the quality of your commenting of your Java code, your adherence to some reasonable coding conventions, your avoidance of blatantly inefficient techniques, and the quality of your README file. Please note that your commenting and your README are most important when a specific part of your program does not function properly; these two tools will allow the TAs to assign partial credit as appropriate.

5 Handing in

To hand in, type `cs031_handin comp`. It will prompt you for confirmation and then hand in your entire current working directory. If you've done any extra credit, please hand in both submissions together, clearly marking which is extra credit.

As per the course missive, you may not hand in compiler late.

5.1 Sample output

This is the code that the demo produces for the primes example we provide in the stencil. Don't worry if your output doesn't look exactly like this. There are a number of factors that determine the precise formatting, register usage, etc. As long as the code generated by the compiler works as expected, you have nothing to worry about.

The code is given in four columns to save on paper.

```
.data

primes_:        .word   0:5

.text

proc_isPrime:
        li      $t1, 4
        sub     $sp, $sp, $t1
        sub     $sp, $sp, 40
        sw      $ra, 40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        sw      $fp, 4($sp)

        move    $fp, $sp
        li      $t0, 0
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        sw      $t0, ($t1)
        move    $t1, $fp
        li      $t2, 48
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        li      $t0, 2
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t2, 1
        blt     $t0, $t1, label2
        li      $t2, 0
label2:
        sub     $sp, $sp, 4
        sw      $t2, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        bnez    $t0, label1
        j       label0
label1:
        li      $t0, 0
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t6, 4($sp)
        add     $sp, $sp, 4
        lw      $fp, 4($sp)
        lw      $s7, 8($sp)
        lw      $s6, 12($sp)
        lw      $s5, 16($sp)
        lw      $s4, 20($sp)
        lw      $s3, 24($sp)
        lw      $s2, 28($sp)
        lw      $s1, 32($sp)
        lw      $s0, 36($sp)
        lw      $ra, 40($sp)
        add     $sp, $sp, 40
        li      $t1, 8
        add     $sp, $sp, $t1
        sub     $sp, $sp, 4
        sw      $t6, 4($sp)
        jr $ra
```

```

label0:
label5:
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        li      $t0, 5
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t2, 1
        blt     $t0, $t1, label6
        li      $t2, 0
label6:
        sub     $sp, $sp, 4
        sw      $t2, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        bnez    $t0, label4
        j       label3
label4:
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t1, 4
        mul     $t0, $t0, $t1
        lw      $t1, primes_($t0)
        sub     $sp, $sp, 4
        sw      $t1, 4($sp)
        li      $t0, 0
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t2, 1
        beq     $t0, $t1, label13
        li      $t2, 0
label13:
        sub     $sp, $sp, 4
        sw      $t2, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t1, 0
        bnez    $t0, label12
        li      $t1, 1
label12:
        sub     $sp, $sp, 4
        sw      $t1, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t1, 0
        bnez    $t0, label10
        j       label9
label10:
        move    $t1, $fp
        li      $t2, 48
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
```

```
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t1, 4
        mul     $t0, $t0, $t1
        lw      $t1, primes_($t0)
        sub     $sp, $sp, 4
        sw      $t1, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        rem     $t0, $t0, $t1
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        li      $t0, 0
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t2, 1
        beq     $t0, $t1, label14
        li      $t2, 0
label14:
        sub     $sp, $sp, 4
        sw      $t2, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t1, 0
        bnez    $t0, label11
        j       label9
label11:
        li      $t1, 1
label9:
        sub     $sp, $sp, 4
        sw      $t1, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        bnez    $t0, label8
        j       label7
label8:
        li      $t0, 0
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t6, 4($sp)
        add     $sp, $sp, 4
        lw      $fp, 4($sp)
        lw      $s7, 8($sp)
        lw      $s6, 12($sp)
        lw      $s5, 16($sp)
        lw      $s4, 20($sp)
        lw      $s3, 24($sp)
        lw      $s2, 28($sp)
        lw      $s1, 32($sp)
        lw      $s0, 36($sp)
        lw      $ra, 40($sp)
        add     $sp, $sp, 40
        li      $t1, 8
        add     $sp, $sp, $t1
        sub     $sp, $sp, 4
        sw      $t6, 4($sp)
        jr $ra

label7:
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        li      $t0, 1
        sub     $sp, $sp, 4
```

```
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        add     $t0, $t0, $t1
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        sw      $t0, ($t1)
        j       label5
label3:
        li      $t0, 1
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t6, 4($sp)
        add     $sp, $sp, 4
        lw      $fp, 4($sp)
        lw      $s7, 8($sp)
        lw      $s6, 12($sp)
        lw      $s5, 16($sp)
        lw      $s4, 20($sp)
        lw      $s3, 24($sp)
        lw      $s2, 28($sp)
        lw      $s1, 32($sp)
        lw      $s0, 36($sp)
        lw      $ra, 40($sp)
        add     $sp, $sp, 40
        li      $t1, 8
        add     $sp, $sp, $t1
        sub     $sp, $sp, 4
        sw      $t6, 4($sp)
        jr $ra

        lw      $fp, 4($sp)
        lw      $s7, 8($sp)
        lw      $s6, 12($sp)
        lw      $s5, 16($sp)
        lw      $s4, 20($sp)
        lw      $s3, 24($sp)
        lw      $s2, 28($sp)
        lw      $s1, 32($sp)
        lw      $s0, 36($sp)
        lw      $ra, 40($sp)
        add     $sp, $sp, 40
        li      $t1, 8
        add     $sp, $sp, $t1
        li      $t1, 0
        sub     $sp, $sp, 4
        sw      $t1, 4($sp)
        jr $ra

main:
proc_main:
        li      $t1, 8
        sub     $sp, $sp, $t1
        sub     $sp, $sp, 40
        sw      $ra, 40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        sw      $fp, 4($sp)

        move    $fp, $sp
        li      $t0, 0
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        sw      $t0, ($t1)
label17:
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        li      $t0, 5
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t2, 1
        blt     $t0, $t1, label18
        li      $t2, 0
label18:
        sub     $sp, $sp, 4
        sw      $t2, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        bnez    $t0, label16
        j       label15
label16:
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        li      $t0, 0
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t2, 4
        mul     $t0, $t0, $t2
        sw      $t1, primes_($t0)
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        li      $t0, 1
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        add     $t0, $t0, $t1
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        sw      $t0, ($t1)
        j       label17
label15:
        li      $t0, 0
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        move    $t1, $fp
        li      $t2, 48
        add     $t1, $t1, $t2
        sw      $t0, ($t1)
        li      $t0, 2
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        sw      $t0, ($t1)
label21:
        move    $t1, $fp
        li      $t2, 48
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        li      $t0, 5
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t2, 1
        blt     $t0, $t1, label22
        li      $t2, 0
label22:
        sub     $sp, $sp, 4
        sw      $t2, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        bnez    $t0, label20
        j       label19
label20:
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        jal     proc_isPrime
        li      $t0, 1
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t2, 1
        beq     $t0, $t1, label25
        li      $t2, 0
label25:
        sub     $sp, $sp, 4
        sw      $t2, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        bnez    $t0, label24
        j       label23
label24:
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $a0, 4($sp)
        add     $sp, $sp, 4
        li      $v0, 1
        syscall
        li      $v0, 11
        li      $a0, 10
        syscall
        move    $t1, $fp
        li      $t2, 48
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        li      $t2, 4
        mul     $t0, $t0, $t2
        sw      $t1, primes_($t0)
        move    $t1, $fp
        li      $t2, 48
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        li      $t0, 1
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        add     $t0, $t0, $t1
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        move    $t1, $fp
        li      $t2, 48
        add     $t1, $t1, $t2
        sw      $t0, ($t1)
label23:
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        lw      $t0, ($t1)
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        li      $t0, 1
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        add     $sp, $sp, 4
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        add     $t0, $t0, $t1
        sub     $sp, $sp, 4
        sw      $t0, 4($sp)
        lw      $t0, 4($sp)
        add     $sp, $sp, 4
        move    $t1, $fp
        li      $t2, 44
        add     $t1, $t1, $t2
        sw      $t0, ($t1)
        j       label21
label19:
        lw      $fp, 4($sp)
        lw      $s7, 8($sp)
        lw      $s6, 12($sp)
        lw      $s5, 16($sp)
        lw      $s4, 20($sp)
        lw      $s3, 24($sp)
        lw      $s2, 28($sp)
        lw      $s1, 32($sp)
        lw      $s0, 36($sp)
        lw      $ra, 40($sp)
        add     $sp, $sp, 40
        li      $t1, 8
        add     $sp, $sp, $t1
        li      $t1, 0
        sub     $sp, $sp, 4
        sw      $t1, 4($sp)
        jr $ra
```