CS 2210 Programming Project (Part IV)

Code Generation

This project is intended to give you experience in writing a code generator as well as bring together the various issues of code generation discussed in the text and in class.

Due date

The assignment is due **December 9th, 2023**, **11:59pm**. This is the hard deadline and no extensions will be given for this project.

Project Summary

Your task is to write a code generator, the final phase of your compiler. It produces (target) assembly code for the MIPS R2000 architecture. It takes as input the augmented AST and symbol table produced by the previous phases of your compiler. The generated code will be executed using SPIM S20, a simulator for the MIPS R2000.

Code generation will consist of assigning memory addresses for each variable used in the MINI-JAVA program and translating subtrees of the AST (intermediate language representation) into sequences of assembly instructions that perform the same task.

Code Generation

This is the only phase of your compiler which is machine dependent. Examples of assembly code programs will be provided on the class webpage.

The important/interesting issues in generating code for MINI-JAVA are discussed in the following paragraphs. Please refer to chapter 7, 8 and class notes for further details on these issues,

You can make the following assumptions to simplify the project.

- Code is generated for the intermediate instructions on a statement-by-statement basis without taking into account the context of an intermediate instruction

- code is generated for the intermediate instructions in the order that they occur within the intermediate instruction sequence.

- You may take advantage of any special instruction of the machine when choosing target instructions for a given intermediate instruction.

- You do not have to any fancy register allocation or optimization on your target code.

Computing Memory Addresses

Since address information has not been computed and entered into the symbol table by earlier phases, the first task of the code generator is to compute the offsets of each variable name (both global and local); that is, the address of each local data object and formal parameter within the activation record for ____ ____ they are declared. This can be done by initializing a variable **offset** at the ____ ____ ____ation section, and as each declaration is processed, the current value of **offset** ____ ____ ____tribute of that symbol, and **offset** is then incremented by the total width of ____ ____ ____ending on its type).

The program exec____ ____ ____method called `main()` being called. Since the language has no way to initiate ____ ____ ____re instantiated when program execution begins. Thus, all storage for all cla____ ____ ____ally. The offsets of variables within classes should be computed and stored as an attribute of the variable name, typically relative to the start of the class or can just be relative to the start of the global storage.

For simplicity, declarations within a method donot contain any objects whose types are classes. That is, local variables can only be of integer type or integer array type.

Call-by-value parameters will have a width dependent on the type of the parameter (remember we are using only integer parameters), whereas call-by-reference parameters will have a width equal to ONE word to store an address. The total activation record size of each method should be computed at this time and entered in the symbol table as an attribute of the method name.

The machine architecture must be take into account when computing these widths, that is, an integer in the MIPS processor is 4 bytes. Offsets of locals can be implemented as a negative offset from the frame pointer while offsets of parameters can be positive from the frame pointer. Thus, the computation of offsets of arguments and local variables can be done independently. This information will be used upon every reference to the data object in addition to being used in the allocation of storage for activation records.

Handling Structure Data Types

Storage for an array is allocated as a consecutive block of memory. Access to individual elements of an array is handled by generating an address calculation using the base address of the array, the index of the desired element, and the size of the elements. You are free to choose the layout of elements of an array in your implementation (e.g., row major or column major order).

Simple Control Flow

Code for simple control statements, namely conditional and loops in MINI-JAVA, can be generated according to the semantics of conventional programming languages using the compare and branch instructions of the assembly language. Unique target labels will also have to be generated.

Method Invocation, Prologues, and Epilogues

Recursion in MINI-JAVA prevents the use of a static storage allocation strategy. However, the language has no features that prevent the deallocation of activation records in a last-in-first-out manner. That is, activation records containing data local to an execution of a method, actual parameters, saved machine status, and other information needed to manage the activation of the

method can be stored on a run-time stack. The MIPS assembly language provides the subroutine call mechanisms to manipulate the run-time user stack.

An activation record for a method is pushed onto the stack upon a call to that method, while the activation record for the method is popped from the stack when execution of the method is finished and control is returned to the caller. As MINI-JAVA does not allow dynamic classes and arrays, the sizes of the activation records are known at compile time.

Method calls result in the execution of a calling sequence. Upon a call, an activation record for the callee must be set up and control must be transferred to the callee after saving the appropriate information in the activation record. For each method, the generated code sequence will consist of a prologue, the code for the execution of the method, and the epilogue. Typically, the prologue saves the registers upon entry to the method, allocates space on the stack for local variables, whereas the epilogue consists of the code to restore the machine status and returning control to the point in the caller immediately after the point of call. The handout on the MIPS assembly language explains the instructions used to implement these actions.

Parameter passing

In order to correctly handle the formal parameters within the body of the callee, the symbol table entry for each formal parameter must include an attribute that indicates the parameter passing mode, that is, by-value or by-reference. Remember that we do not pass arrays as parameters. On a method invocation, call-by-value parameters are handled by allocating the local store for the size of the object in the activation record of the callee and then evaluating the actual parameter and initializing the local store within the callee with the value of the actual parameter. All accesses to that formal parameter will change the value in the local space, with no effect on the caller. On a return, no values are copied back to the caller.

Call-by-reference parameters are handled by allocating local space in the callee's activation record for the address of the actual parameter and then copying the address of the actual parameter into that local space. All accesses to that formal parameter during execution of the callee are indirect accesses through this address, having a direct effect on the caller. On return, no action is taken other than reclaiming the space.

Note that MIPS has a convention that the first 4 arguments of a method call are passed in register $a0-$a3. You have the option to follow this convention.

Register Usage

In the MIPS processor, certain registers are typically reserved for special purposes. You should abide by these conventions.

Possible Functions

You may want to write the following functions to help in the code generation.

1. emit_call(func_name, num_arg)/*emit a call instruction; func_name: function id lexeme pointer; num_arg: number of arguments */

2. emit_label(l_num)/*emit a definition of a label; l_num: label number; example: L=102, code generated = "L_102" */

3. emit_goto(operator, l_num) /*emit unconditional and conditional jump instructions; operator: an operator in the branch/jump group; l_num: label number*/

4. emit_data(name, type, size) /* emit one data line, which is used for STATIC allocation; name: data object id lexeme pointer; type: type width; size: number of elements of above type */

5. emit_str(name, ...) constant definition line; name: pointer to the name lexeme; str: poi....

6. emit_most(oper........ op1, op2, op3) /* emit most of the instructions; operator: one of the instr........ group; type: data type; num_op: number of operators 1, 2 and/or 3; op1........ and op3 can be omitted depending on num_op */

Run-time Error Detection

You do not need to generate any run-time checks. Thus, you can assume that array bounds are within range and scalars are within range.

Testing your code

The code generated is MIPS assembly code and should follow the descriptions specified in the handout SPIM S20. Samples will be put on the class webpage.

You can run the generated assembly code on the simulator and check the results. However, the correct output does not guarantee that your code is completely correct. You should examine your generated code carefully.

> proj4> codeGen < sample1.java
> proj4> spim -asm -file code.s
> ... ...
> **or**
> proj4> spim
> (spim) load "code.s"
> (spim) step
> [0x00400000] 0x8fa40000 lw $4,0($29)
> (spim) run
> ... ...

Assignment submission

Please submit your project in Canvas before the due time. The submission should be a compressed file that contains your project source code and readme file (if any).