Compiler Construction - Assignment 4

LIACS, Leiden University

Fall 2008

Introduction

In this assignment we will make a fully functional compiler of the subset Pascal compiler that has been studied in the previous assignments. This means, you have to write a code generator unit, which takes the intermediate code and translates this into assembly instructions. In general, these instructions are fed to an assembler, which produces binary code. However, we provide a simulator that can act directly on plain-text assembly.

Framework

In your CVS repository, a module named "`assignment4`" is available. This module contains a framework for a subset Pascal compiler. Everything up to and including the intermediate code generation is already filled in; you have to add (naive) MIPS assembly code generation. A makefile is provided; after performing a checkout of the CVS module, first execute `make first` once. In order to build the entire compiler, simply execute `make`.

Data structures

The framework of this assignment is an extension to the framework of the previous assignments. You should already be familiar with the data structures that are used for syntax tree construction, symbol table management and intermediate code generation. If not, refer to the documentation of the previous assignment(s). Several new data structures that are needed during the code generation phase are already provided. In the next paragraphs, we briefly describe them. Note that you are not allowed to make any changes to any of the provided data structures, unless stated otherwise. Futhermore, you should avoid triggering warning or error messages from the different object methods.

CodeGenerator

The `CodeGenerator` data structure is meant to handle the assembly code generation. You are encouraged to modify and extend this class. By default, the class offers four (almost) empty functions: `GenerateHeader()`, which should generate a header in the output file; `GenerateGlobalDecls()`, which should generate the declarations for all global variables that appear in the input program; `GenerateCode()`, which should perform the actual code generation; and `GenerateTrailer()`, which should output other information necessary to complete the assembly file.

StackFrameManager

The `StackFrameManager` data structure is meant to handle the stack frame related issues of the assembly code generation. You are encouraged to modify and extend this class. In the `StackFrameManager.cc` file, several function templates are given that might help you in handling stack frames. However, you are also allowed to devise your own mechanism for this.

The methods which you have to implement and/or extend are marked with the keyword `TODO`. In order to get an overview of all of these locations, you can use the command `grep -n TODO *.cc` for example.

Some examples

To give you a better understanding of the desired output, we describe some examples in this section. Consider the following input code (where x & y are global integer variables):

```
if (x = 1) then
  y := 2
else
  y := 3;
```

The intermediate code that is generated by the framework will look like the following:

```
#   operator      | operand1          | operand2          | result
------------------+-------------------+-------------------+-------------------
0:  BNE_I         | sym: x            | int: 1            | sym: _L1
1:  ASSIGN_I      | int: 2            |                   | sym: y
2:  GOTO          | sym: _L0          |                   |
3:  LABEL         | sym: _L1          |                   |
4:  ASSIGN_I      | int: 3            |                   | sym: y
5:  LABEL         | sym: _L0          |                   |
------------------+-------------------+-------------------+-------------------
```

Corresponding MIPS assembly could look like the following code fragment. For each instruction, the related intermediate statement is indicated.

```
    lw    $3, x                 # statement 0: load the value of x into reg $3
    addi  $4, $0, 1             # statement 0: put the value 1 into reg $4
    bne   $3, $4, _L1           # statement 0: if ($3 != $4) goto _L1
    addi  $3, $0, 2             # statement 1: put the value 2 into reg $3
    sw    $3, y                 # statement 1: store the value of $3 in y
    j     _L0                   # statement 2: goto _L0
_L1:                            # statement 3
    addi  $3, $0, 3             # statement 4: put the value 3 into reg $3
    sw    $3, y                 # statement 4: store the value of $3 in y
_L0:                            # statement 5
```

As another example, consider a function `increase` that takes a parameter x, and returns $x + 1$. A dump of the corresponding intermediate code could look like the following:

```
#   operator      | operand1          | operand2          | result
------------------+-------------------+-------------------+-------------------
0:  SUBPROG       | sym: increase     |                   |
1:  ADD_I         | sym: x            | int: 1            | sym: t1
2:  RETURN_I      | sym: t1           |                   |
```

Since this is a subprogram, we also need to take care of the activation record (stack frame). In Figure 1, a visual representation of the stack frame is given. In case the subprogram would take more parameters, they would be added at the upper side of the figure, such that address `$fp + 8` is assigned to the second parameter , `$fp + 12` to the third parameter, etc. This means that the parameters are pushed on the stack

high addresses

`$fp + 4`	parameter: `x`
`$fp + 0`	return address
`$fp - 4`	previous frame pointer
`$fp - 8`	local variable: `t1`

low addresses

Figure 1: The stack frame layout used in our example.

in the reverse (right-to-left) order. Additional local variables would appear at the bottom of the figure (with addresses `$fp - 12`, `$fp - 16` etc.). Parameters are pushed on the stack by the caller, and removed from the stack in the epilogue of the callee. All other fields are both pushed and removed by the callee. The following dump shows a possible assembly implementation for the `increase` function:

```
increase:
    # Statement 0: Setup the local stack frame (prologue)
    addi  $sp, $sp, -4              # Reserve a stack word
    sw    $31, ($sp)                # Save return address
    addi  $sp, $sp, -4              # Reserve a stack word
    sw    $fp, ($sp)                # Save current frame pointer
    addi  $fp, $sp, 4               # Make frame pointer point to return address
    addi  $sp, $sp, -4              # Indicate lowerbound of stack frame

    # Statement 1: The actual body of the function
    lw    $3, 4($fp)                # Grab parameter x from the stack
    addi  $3, $3, 1                 # Add 1 to it (statement 1)
    sw    $3, -8($fp)               # Save it to local variable t1

    # Statement 2: Return from the function (epilogue)
    lw    $3, -8($fp)               # Load local variable t1 ...
    add   $2, $0, $3                # ... and put it into the return register

    lw    $31, ($fp)                # Restore the return address
    lw    $fp, -4($fp)              # Restore the frame pointer
    addi  $sp, $sp,16               # Discard stack frame (4 words: x, $31, $fp, t1)
    jr    $31                       # Jump back to the caller
```

Now we call this function, using the assignment `a := increase(41)`. Here we assume `a` is a global variable. The intermediate code could look like the following:

```
 3:  PARAM_I       | int: 41           |                   |
 4:  FUNCCALL      | sym: increase     |                   | sym: a
-----------------+-------------------+-------------------+-------------------
```

This could be translated into the following assembly instructions:

```
    addi  $3, $0, 41                # statement 3: put the value 41 into reg $3
    addi  $sp, $sp, -4              # statement 3: reserve a stack word
    sw    $3, 0($sp)                # statement 3: store parameter value
    jal   increase                  # statement 4: jump to the function
    sw    $2, a                     # statement 4: assign return value to a
```

Note that the code examples given in this section are not optimal in terms of execution speed.

Assignment

What you have to deliver: a compiler which can parse a given code file and translate this into functionally equivalent MIPS assembly that can be simulated on the (X)SPIM simulator. A binary of the XSPIM simulator is available in your repository. The user manual of the (X)SPIM simulator is available on Blackboard, under "Course Documents", "Practicum Resources". The manual describes the architecture and its instruction set, and gives an overview of the available system calls.

You should also provide clear documentation about your implementation decisions. This documentation should be placed in your CVS repository in the form of a file called `DOCUMENTATION.TXT`. Furthermore, you have to take a reasonable test case (i.e., a small test case, with recursion, such as recursive Fibonacci number calculation) and explain in detail why the assembly code generated by your compiler is correct for this case.

In the framework for this assignment, you receive intermediate code that is similar to the intermediate code you constructed for the previous assignment. Using the classes and methods provided by the framework, you have to step through the intermediate code and write the resulting assembly to a file.

The generated assembly code does not have to be efficient. In the next assignment, you will build further upon the code you have written for this assignment and implement some optimizations. It might be useful to keep this in mind while you are working on this exercise.

Some other notes:

- In this and the following assignment, we use call-by-value parameter passing.
- You can assume that the input for the code generation phase is correct. This means that you do not have to write code that verifies if the intermediate code is correct.
- A register usage convention exists for the MIPS architecture (e.g., `R4 - R7` are reserved for the first four actual parameters of a function call). However, you do not have to respect this convention, since it would make this assignment considerably more complicated. For example, for integer values, you can just use any of the registers `R3 - R25`.
- For the `real` data type, use 32-bit single precision floating point arithmetic.
- In the file `main.cc`, some other functions are available, that you may need during this and the next assignment (e.g., the `IsGlobal` function, which determines whether a symbol is a global variable or not).

Submission & Grading

Submit your work using CVS. Do not send anything by email. Make sure the latest version of your work has been committed to the CVS repository. Then, tag that version using the command:

```
cvs tag DEADLINE4
```

This tag command has to be issued *before* November 20th, 2008 at 23:59.

For this assignment, 0-10 points can be obtained, which account for 25% of your final grade. If you do not submit your work in time, it will not be graded. The results, once available, can be found on BlackBoard. Your grade for the assignment will not only depend on the functionality of your compiler, but also on the understandability of your code and the documentation of the design decisions. Documentation should be submitted in English only.

Assistance will be given by Michiel Helvensteijn in room 306/308. A schedule for this can be found on BlackBoard. Additionally, you can go to Sven van Haastregt in room 1.22.