P6

Reid Chen, Gaoyi Hu

December 14, 2021

Contents

So, firstly we will have to implement the feature to assert if there is a main function by lookup the global function name. Then, we set the offset when set the function statement. Next, we add code to MIPS when the code generates the function, global variables and some other cases to executre the exact codes in assembly code scope. Typically, for function, we need to substract the offset space to the sp, and for each line, we exectute it in a single stack. And for each global variable, we simply add them to the base of the stack.

1 Code Generation for Global Variable Declarations

Let `N` denote the size of the global variable in bytes. For each global variable `_v`, we generate the following code snippet:

```
    .data     # in static data area
    .align 2  # align on a word boundary
_v: .space N  # label N bytes area with name _v
```

2 Code Generation for Functions

Need to generate the following 4 parts in order:

1. preamble
2. entry
3. body
4. exit

2.1 Generating Function Preamble

For the `main` function,

```
    .text
    .globl main
main:
```

For all other functions,

```
.text
_<functionName>:
```

where `<functionName>` is a placeholder for the function called `functinoName`. The `.text` indicates the assembler that the instructions below should be stored in the text area.

2.2 Generating Function Entry

We do not need to worry about the actual parameters because the caller of this function has already pushed them on the stack. We need to do the following 4 things in order:

1. push the return address

   ```
   sw   $ra, 0($sp)
   subu $sp, $sp, 4
   ```

2. push the control link

   ```
   sw   $fp, 0($sp)
   subu $sp, $sp, 4
   ```

3. set the `$fp`

   ```
   addu $fp, $sp, 8
   ```

4. push space for local variables

   ```
   subu $sp, $sp, <size of locals in bytes>
   ```

   where `size of locals in bytes` can be calculated during semantic analysis.

2.3 Generating Function Body

No need to generate code for declarations, but need to generate code for each statement.

2.3.1 Write Statement

1. Call the `codeGen` of the expressions in the write statement so that the value to be written will be placed on the top of the stack.

   ```
   myExp.codeGen();
   ```

2. Generate code that pop the value one the top of the stack into `$a0`

   ```
   genPop(a0, 4);
   ```

3. Set `$v0` to 1

   ```
   generate("li", v0, 1);
   ```

4. Make a system call

   ```
   generate("syscall");
   ```

2.3.2 If-Then Statement

1. Evaluate the condition
2. Pop the top-of-stack value into register `$t0`
3. Jump to `FalseLabel` if `$t0` is `FALSE`
4. Code for the statement list
5. `FalseLabel:`

2.3.3 If-Then-Else Statement

1. Evaluate the condition
2. Pop the top-of-stack value into register `$t0`
3. Jump to `FalseLabel` if `$t0` is `FALSE`
4. Code for the then-statement list
5. Jump to `Exit`
6. `FalseLabel:`
7. `Exit:`

2.3.4 While Statement

1. `Start:`
2. Evaluate the condition
3. Pop the top-of-stack value into register `$t0`
4. Jump to `FalseLabel` if `$0` is `FALSE`
5. Code for the statement list
6. `Start`
7. `FalseLabel:`

2.3.5 Identifier

2.4 Generating Function Exit

Want to pop off this function's AR. Then jump to the address that stored in the return address field of this function's AR. Popping off this function's AR means to restore the `$sp` and `$fp` to its caller' values. However, instead of simply setting `$sp` to `$fp`, we want to store `$fp` to a temporary register. Then restore `$fp` using the value stored in the control link field. Lastly, we restore `$sp` using the value stored in that temporary register. We restore `$sp` because a system interrupt may happen and use the stack. If we restore `$sp` at the beginning, the system interrupt may overwrite data we need.

```
lw   $ra, 0($fp)
move $t0, $fp
lw   $fp, -4($fp)
move $sp, $t0
jr   $ra
```