Department of Computer Science and Engineering
IIT Guwahati

Implementation of Programming Languages Lab: CS 348

Assignment - 6: Target Code Generator for nanoC *Marks: 100*

In this assignment you will write a target code translator from the *TAC* quad *array* (with the supporting *symbol table*, and other *auxiliary data structures*) to the *assembly language of* x86 / IA-32 / x86-64. The translation is now machine-specific and your generated assembly code would be translated with the gcc assembler to produce the final executable codes for the nanoC program.

1 Scope of Target Translation

- For simplicity restrict nanoC further:
  1. Support only `void` and `int` types. Skip `char` type.
  2. Support only one–dimensional arrays.
  3. Support only `int` and `int*` types for parameter types of functions.
  4. Support only `void`, `int`, and `int*` types for returns types of functions.
  5. No type conversion is to be supported.
  6. Support only the global and function scopes. No nested scope to be supported.
- For I/O, provide a library using in-line assembly language program of x86 / IA-32 / x86-64 along with `syscall` for gcc assembler.:
  - `int printStr(char *s)`: prints a string of characters. The parameter is terminated by `\0`. The return value is the number of characters printed. If `s = "\n"`, a newline is printed.
  - `int printInt(int n)`: prints the integer value of `n` (no newline). It returns the number of characters printed.
  - `int readInt(int *eP)`: reads an integer (signed) and returns it. The parameter is for error (`ERR = 1, OK = 0`).

  The header file `myl.h` of the library will be as follows:

```
#ifndef _MYL_H
#define _MYL_H
#define ERR 1
#define OK 0
int printStr(char *);
int printInt(int);
int readInt(int *eP);  // *eP is for error, if the input is not an integer
#endif
```

2 Design of the Translator

The steps for target code generation were outlined in Target Code Generation lecture presentations. In this assignment, however, you *do not need to deal with any machine-independent or machine-specific optimization*. Hence the translation comprises the following major steps only:

1. **Memory Binding**: This deals with the design of the allocation schema of variables (including parameters and constants) that associates each variable to the respective address expression or register. This needs to handle the following:
   - *Handle local variables, parameters, and return value for a function.* These are automatic and reside in the Activation Record (AR) of the function. Various design schema for AR are possible based on the calling sequence protocol. A sample AR design could be as follows:

Offset	Stack Item	Responsibility
-ve	Saved Registers	Callee Saves & Restores
-ve	Callee Local Data	Callee defines and uses
0	Base Pointer of Caller	Callee Saves & Restores
	Return Address	Saved by call, used by ret
+ve	Return Value	Callee writes, Caller reads
+ve	Parameters	Caller writes, Callee reads

Activation Record Structure with Management Protocol

  - Offset's in the AR are with respect to the Base Pointer of Callee.
  - Return Value can alternatively be returned through a register (like accumulator or `eax`).
  - The AR will be populated from the Symbol Table of the function.
- *Handle global variables* (note that local static variables are not allowed in nanoC) as static and generate allocations in static area. This will be populated from global symbol table (ST.gbl).
- *Register Allocations & Assignment*: Create memory binding for variables in registers:
  - After a load / store the variable on the activation record and the register have identical values
  - Registers can be used to store temporary computed values
  - Register allocations are often used to pass `int` or pointer parameters
  - Register allocations are often used to return `int` or pointer values

Note: *Refer to* Run-Time Environment *lecture presentations for details and examples on memory binding.*

2. **Code Translation**: This deals with the translation of 3–Address quad's to x86 / IA-32 / x86-64 assembly code. This needs to handle:
   - *Generation of Function Prologue*: Few lines of code at the beginning of a function, which prepare the stack and registers for use within the function.
   - *Generate Function Epilogue*: Appears at the end of the function, and restores the stack and registers to the state they were in before the function was called.
   - *Map 3–Address Code to Assembly*: To translate the function body do:
     - Choose optimized assembly instructions for every expression, assignment and control quad.
     - Use algebraic simplification & reduction of strength for choice of assembly instructions from a quad.

   Note: *Refer to* Target Code Generation *lecture presentations for details.*
3. **Target Code**: Integrate all the above code into an Assembly File for gcc assembler.

3 The Assignment

1. Write a target code (x86 / IA-32 / x86-64) translator from the 3-Address quad's generated from the flex and bison specifications of nanoC (with restrictions as mentioned in Section 1). Assume that the input nanoC file is lexically, syntactically, and semantically correct. Hence no error handling and / or recovery is expected.
2. Prepare a Makefile to compile and test the project.
3. Prepare test input files `A6_group.nc` to test the target code translation and generate the translation output in `A6_group.asm`. Multiple input files may be serially numbered.
4. Name your files as follows:

File	Naming
Flex Specification	`A6_group.l`
Bison Specification	`A6_group.y`
Data Structures Definitions & Global Function Prototypes	`A6_group_translator.h`
Data Structures, Function Implementations & Translator `main()`	`A6_group_translator.c`
Test Inputs: Number the tests with `<number> = 1, 2, 3, ...`	`A6_group_test<number>.nc`
Test Outputs: Output of 3-address codes for test `<number>`	`A6_group_quads<number>.out`
Test Outputs: Output of assembly codes for test `<number>`	`A6_group_quads<number>.asm`

5. Prepare a tar-archive with the name `A6_group.tar` containing all the files and upload.

4 Credits

While it is desirable, but it is not mandatory to:

1. Implement the I/O library.
2. Use explicit register allocation and assignment. This is the simplest table lookup scheme. But it generates a lot of redundant load-store instructions.

With the above exclusions, the credit distribution will be as follows:

Design of Memory Binding: **20 + 10 = 30**
Handling of Activation Records
Handling of Static Memory & Binding

Design of Code Translation: **5 + 5 + 20 = 30**
Handling of Prologue
Handling of Epilogue
Handling of Function Body

Design of Target Code Management: **10**
Integration of translated codes into an assembly file

Design of Test files and correctness of outputs: **5 * 3 + 10 = 25**
Test at least 5 i/p files covering all rules
Shortcoming and / or bugs, if any, should be highlighted

Integrated interface of the nanoC Compiler: **5**