Department of Computer Science
Ashoka University

Programming Language Design and Implementation (PLDI): CS-1319-1

Assignment - 5: `Target Code Generator for` $\mathcal{F}_{-15}$ *Marks: 100*
Assign Date: *November 17, 2024* Submit Date: *23:55, November 30, 2024*

---

1. *You must submit your assignment using the prefix* `name = FirstName_LastName` *for all files.*
2. *To receive full credit, your program must be correct and your* `.pdf` *file must explain your program adequately.*

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 $\mathcal{F}_{-15}$ program.

1 IO Library

For I/O (`read` and `print`), 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
```

1.1 IO Library Codes

```
#include <stdio.h>

// IO Library header
int printStr(char *s);
int printInt(int n);
int readInt(int *eP);

char format[] = "%s %d%c\n\n";
char string[] = "Inlined output is";
int integer = 10;
int charac = 'h';
```

```
int main() {
    // inline assembly for printf("%s %d%c\n", "Inlined output is", 10, 'h');
    __asm
    {
        // push the parameters in right-to-left order
        mov  eax, charac
        push eax
        mov  eax, integer
        push eax
        mov  eax, offset string // Offset used to access global array
        push eax
        mov  eax, offset format
        push eax
        call printf
        // clean up the stack so that main can exit cleanly
        // use the unused register ebx to do the cleanup
        pop  ebx
        pop  ebx
        pop  ebx
        pop  ebx
    }

    int len;
    // len = printStr("Inline string test\n\n");
    char *s = "Inline string test\n\n";
    __asm {
        mov  eax, s
        push eax
        call printStr
        pop  ebx
        mov len, eax
    }
    printStr("Length of output = ");
    printInt(len);
    printStr("\n\n");

    // len = printInt(13);
    int n = 13;
    __asm {
        mov  eax, n
        push eax
        call printInt
        pop  ebx
        mov len, eax
    }
    printStr("\n\nLength of output = ");
    printInt(len);
    printStr("\n\n");

    printStr("Input integer\n\n");

    int m;
    int *eP;
    //m = readInt(eP);
    __asm {
        mov  eax, eP
        push eax
```

```
        call readInt
        pop  ebx
        mov m, eax
    }
    printStr("Input received = ");
    printInt(m);

    return 0;
}

// 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 printStr(char *s) {
    char *pSformat = "%s";
    __asm
    {
        mov  eax, s
        push eax
        mov  eax, pSformat
        push eax
        call printf
        pop  ebx
        pop  ebx
    }
} // Return value is in eax


// Prints the integer value of n (no newline).
// It returns the number of characters printed.
int printInt(int n) {
    char *pIformat = "%d";
    __asm
    {
        mov  eax, n
        push eax
        mov  eax, pIformat
        push eax
        call printf
        pop  ebx
        pop  ebx
    }
} // Return value is in eax

// Reads an integer (signed) and returns it.
// The parameter is for error (ERR = 1, OK = 0)
int readInt(int *eP) {
    char *rIformat = "%d";
    int n;
    int *p = &n;
    __asm
    {
        mov  eax, p
        push eax
        mov  eax, rIformat
        push eax
```

```
        call scanf
        pop  ebx
        pop  ebx
    }
    return n;
} // Error not handled
```

Sources

- gcc
  1. C program to create assembly for reading integer, Stackoverflow
- MSVC
  1. Calling C Functions in Inline Assembly, Microsoft
  2. __asm, Microsoft
  3. Inline Assembly function calling, GitHub

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 that associates each variable to the respective address expression or register. This needs to handle the following:
   - *Handle global variables* 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:
   - *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 $\mathcal{F}_{-15}$. Assume that the input $\mathcal{F}_{-15}$ file is lexically, syntactically, and semantically correct. Hence no error handling and / or recovery is expected.
2. You are given 6 problems. Write $\mathcal{F}_{-15}$ program for each problem. test files for the following problems to test your translator. Run the target code translation on them and generate the translation output in `name_A5_quads<number>.asm` where `<number>` is respective test-file number.
   (a) `Test1.f15`: Add two numbers given in `Test1` and print the result.

(b) Test2.f15: Evaluate `x = 2 + 3 - (7 - 4) + 8` and print the result.

(c) Test3.f15: Read two numbers, convert both to respective absolute values, add these values, and print the result.

(d) Test4.f15: Read four numbers, compute the maximum and minimum of numbers, and print the results.

(e) Test5.f15: Read `n`, add first `n` natural numbers, and print the result.

(f) Test6.f15: Read `n` and print the first `n + 1` Fibonacci numbers. For example, for `n = 4`, `Test6` should print `0 1 1 2 3`.

3. Prepare a Makefile to compile and test the project. Ensure your code compiles with the command `make build` and produces an executable named `compiler`.
4. Add a command `make test` which tests your `compiler` against all given test-cases and stores output in respective `.asm` files as per above format.
5. Name your files as follows:

File	**Naming**
Flex Specification	name_A5.l
Bison Specification	name_A5.y
Data Structures Definitions & Global Function Prototypes	name_A5_translator.h
Data Structures, Function Implementations & Translator `main()`	name_A5_translator.(c|cxx)
Test Outputs: Output of 3-address codes for test `<number>`	name_A5_quads<number>.out
Test Outputs: Output of assembly codes for test `<number>`	name_A5_quads<number>.asm
Makefile	name_A5.mak
Explanations of the translator desing	name_A5.pdf

6. Prepare a tar-archive with the name name_A5.tar containing all the files and upload.

4 Credits

The credit distribution will be as follows:

1. Writing the test cases **[2 + 2 + 3 + 4 + 4 + 5 = 20]**
2. Working of the translator on the test cases **[6 + 6 + 9 + 12 + 12 + 15 = 60]**
3. Explanation of Program **[20]**

   In your pdf, clearly specify which cases are passing correctly and which are failing. Explain the reason why they are failing (not implemented, unresolved errors, etc.).