University of Amsterdam

Assignment 6

Code Generation

March 18, 2021

Students:
René Kok
13671146

Aram Mutlu
13574116

Lecturer:
Dhr. dr. C.U. Grelck

Course:
Compiler Construction

Course code:
5062COMP6Y

1 Introduction

This report provides information regarding the sixth assignment "Code Generation" of the course Compiler Construction by Dhr. dr. C.U. Grelck. The goal of this assignment is to manually generate CiviC-VM assembly code, point out the relationship between assembly code and source code, add the number of bytes required for each line of CiviC-VM assembly code, compute the proper byte code offset for each jump instruction and lastly devise a compilation scheme that replaces each occurrence of a for-loop in the body of a CiviC function by semantically equivalent CiviC code

2 Code Generation

Consider the following CiviC function definition:

```
int factorial ( int x )
{
    int res ;
    if ( x <= 1) res = 1;
    else res = x * factorial ( x - 1);
    return res ;
}
```

Listing 1: CiviC function definition

A) Manually generate CiviC-VM assembly code for the above function definition. Make use of labels to mark destinations of jump instructions.

```
factorial:
    esr 1               // Add factorial function

    iload_0             // Add (local) int x to stack
    iloadc 0            // Add constant 1 to stack
    ile                 // Int less or equal (x <= 1)
    branch_f L1         // Continue at L1 if above value is true
    iloadc 0            // Add constant 1 to stack
    istore 1            // Assign constant 1 to res
    jump L2             // Jump to L2 (skip the else statement (L1:))

L1:                     // L1 label
    iload_0             // Load (local) var x
    isrg factorial      // Load factorial() function
    iload_0             // Load (local) var x
    iloadc 0            // Load constant 1
    isub                // Subtract the numbers x - 1
    jsr factorial 1     // Call factorial() with 1 param
    imul                // Multiply the outcome -> x*factorial(x-1)
    istore 1            // Store the outcome (res = ...outcome)

L2:                     // end label (if statement)
    iload_1             // Add res to stack
    ireturn             // Return int res
```

Listing 2: Assembly code

B) Point out the relationship between assembly code and source code through line by line comments in the assembly code.
Answer added to Listing of answer A above.
C) Add the number of bytes required for each line of CiviC-VM assembly code. Assume here jump instructions would take byte code offsets as arguments and not labels.

```
factorial:
    esr 1               // 2 bytes

    iload_0             // 1 byte
    iloadc 0            // 3 bytes
    ile                 // 1 byte
    branch_f L1         // 2 bytes      // Jump 7 bytes
    iloadc 0            // 3 bytes
    istore 1            // 2 bytes
    jump L2             // 2 bytes      // Jump 14 bytes
```

```
L2:
    iload_0             // 1 byte
    isrg factorial      // 2 bytes
    iload_0             // 1 byte
    iloadc 0            // 3 bytes
    isub                // 1 byte
    jsr factorial 1     // 3 bytes      // Jump -26 bytes
    imul                // 1 bytes
    istore 1            // 2 bytes

L2:
    iload_1             // 1 byte
    ireturn             // 1 byte
```

Listing 3: Assembly code with number of bytes

D) Compute the proper byte code offset for each jump instruction. Consult the CiviC-VM manual for details on individual instructions.
Answer added to Listing of answer C above.

3 Compilation Schemes Revisited

Devise a compilation scheme that replaces each occurrence of a for-loop in the body of a CiviC function by semantically equivalent CiviC code that makes use of while-loops and/or do/whileloops instead. As a simplification consider only for-loops without a step specification, and assume that CiviC would support arbitrary interleaving of variable declarations and statements in function bodies as in C99 proper.

```
C [[ for (int i = lower, upper) { body } Rest ]]
-> CX [[
        while(lower != upper){
            C[[body]] lower = lower + 1
        }
        C[[Rest]]
    ]]
```

Listing 4: Compilation Scheme