Computer Organization
Final Project Report

ARM Assembly to C Decompiler

Submitted on 30 Nov, 2017 by
Nihesh Anderson | 2016059
Rahul Garg | 2016072
Harsh Pathak | 2016041

Abstract

The ARM Assembly to C Decompiler is written in C++ and designed for converting ARM Assembly program to C.

Phases of decompilation

Decompilation is done in 4 broad phases.

Phase 1 - Preprocessing
The ARM source file is read line by line and comments are removed. Also, unnecessary tabs, spaces and new lines are removed and the code is converted into a standard format.

Phase 2 - Tokenizing Commands
The formatted input file from Phase 1 is used and the ARM instructions are split by space and stored in a vector<vector<string>> Data-Structure. The components of a specific ARM instruction are also split by space.
Example - *add r0,r1,r2* gets stored in a vector<string> with elements as *“add”, ”r0”, ”r1”* and *”r2”*.

Phase 3 - Modelling Control Flow Graph
The entire program has be divided into *nodes/blocks* which contain a codesuite(group of instructions belonging to that node) that can be executed sequentially. Each node would consist of simple arithmetic/copy instructions without any jump or branch instructions. Nodes can be obtained by splitting the source code at points wherever a *jump statement/jump target* {bl/blx not included because of their use in function calls} is detected. They will be also split on the basis of a label name present. These nodes will be used to form a *control-flow graph* that models how control is transferred from one node to the other**.** Further, all the jumps will be denoted using directed edges from one node to another node in the call flow graph.

Phase 4 - Sequential Translation to C

The control-flow graph will be traversed from the top node to the bottom and translated to an equivalent C code depending upon the flow of control based on certain logic, which has been described below. Drawing inferences from the graph obtained from phase 3 is the major component of this phase of the decompilation algorithm. Once loops and jumps are inferred from the graph, they are inserted in between the sequential lines, appropriately.

Control Flow Analysis Algorithm

1. **Detecting Do-While Loops** Suppose if there is an edge from a node X to node Y, such that X>Y, certainly, the program tries to execute a set of statements that it previously executed. Hence, all the nodes in between and including X and Y will be a part of a do-while loop. Now, before executing sequential translation of block X to block Y, a do loop must be opened, which is closed at block Y

2. **Detecting continue and break statements in Loop - (1)** describes the loop implementation in our decompiler. Now if there is a Node Z between X and Y in the order that leads back to Node X, then for sure it is a *continue* statement. Similarly, a break condition would be detected if a Node Z between X and Y led to node Y+1. We have made sure to store all the break and continue conditions of such a loop in the loop class.

3. **Detecting conditional if jumps** Consider two nodes X and Y such that there is a directed edge from X to Y. If X<Y, it means we are skipping some nodes between X and Y when certain conditions are met. Intuitively, we can handle using an if clause. Let's consider a branch condition c. There is a jump from block X to block Y if condition c is satisfied. So, The blocks X to Y are executed only if ~c (negation of c) is satisfied. This is the condition of the if branch. An if class has been made for storing the necessary information of an if block.

4. **Detecting conditional if-else jumps** Consider we detect a jump from node X to node Y (X<Y). Now, there might be a case where there is a jump from Y-1 to Z such that Y<Z. This is an intersection of two if conditions, which is not possible in C. Hence, it is most probably a if-else clause, assuming only codes that are compiled to ARM are being decompiled.

5. **Detecting function calls :** Instructions of the format blx and bl label gives a clear indication that "label" is a function, where label is a block or group of blocks. So, the block of code from "label" to the instruction "MOV pc,lr" will be considered to be a part of the function body. A separate call flow graph will be created for this function, and it will be linked to the node that calls the function.

Sequential Translation Algorithm

1. **Character printing support** In arm, character input is fed as an intermediate using "#" prefix. For example, #' ' denotes a space. Mov parser detects the presence of # and parses appropriately and the value is converted into ASCII (int) implicitly and stores the value in the target register.

2. **Sequential commands** add, adds, sub, subs, mul, mov, cmp, swi have a general syntax. The logic has been understood and parsed accordingly. For example, *add r1,r2,r3* means there is a '+' operator between r2 and r3 and stored into r1

3. **Processing individual nodes** A node in the control flow graph is a set of sequential commands. In the previous step, we solved the issue of how to decompile a single sequential instruction. The logic has been extended to decompile a set of sequential instructions mutually exclusively as the decompilation of one statement doesn't depend on the other

4. **Implementing Function Pointers** Different parsers were written to parse sequential instructions. These are called depending upon the first command of the instruction. For example, the first command of *MUL r0, r1, r2* is MUL. Hence, the MUL parser is called to parse this instruction. To make function calling simpler, function pointers were created for different functions and called appropriately.

Global Variables and Function Prototypes

Three header files are included in the decompiled C program by default; stdio.h, math.h and stdlib.h. A global integer variable is created for every register from r0 to r15. These are used by the sequential instruction decompiler to decompile instructions depending upon the registers used. For every function that is detected, a function prototype is declared by default before decompiling the function. A special variable named compareRegister is used for keeping track of the flags that will be set by the ARM code, to replicate its behaviour. The cmp, adds and subs commands modify the compare register, and while/if conditions are decided based on the value of this compareRegister variable.

Features

- ARM code is decompiled into a compilable C code.
- Do - While and If conditions have been detected to increase interpretability of the code.
- The output code is formatted to improve readability, which includes inserting tabs when a loop is detected by keeping track of nesting depth.
- Supports int and char data types.
- Support for function calls using blx and bl (branch with link) have been added.
- Support for commands like add, sub, mul, adds, cmp, and b* (* represents general comparison conditions) have been added.
- Support for various SWI instructions have been added.

Implementation Issues

- While checking if-else loops, there was a necessity to detect two mutually exclusive if conditions
- Function parsing methods were not straightforward. There were many corner cases. For example, mov r0, #' ' is parsed into mov | r0 | #' | ' when split by space. This is not desired
- Detecting functions was a major challenge. There was a need to find the right "mov pc, lr" command that marks the end of the function body. This was resolved by finding the first "mov pc,lr" command that is not present inside any while/if loop.

Project Repository

https://github.com/RahulGarg2/Decompiler

The above URL is a link to the project repository. It contains the code of the working decompiler compatible with GNU G++11 or above, along with a few examples that the algorithm was tested on.