Computer Organization
Final Project Report

ARM Assembly to C Decompiler

Submitted 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 be done in 4 broad phases.

Phase 1 - Preprocessing

The source file is read and transformed into a format that adds/removes new lines, removes comments, etc. This phase can be also referred to as pre-processing stage.

Phase 2 - Tokenizing Commands

ARM instructions are split by space and stored in a vector<vector<string> > Data-Structure.

Phase 3 - Modelling Control Flow Graph

The entire program hasl be divided into nodes that contains a block of code that can be executed sequentially. Each node would consist of simple arithmetic/memory instructions without any jump or branch instructions. Nodes can be obtained by splitting the source code at points wherever a jump statement/jump target is detected. These nodes will be used to form a control-flow graph that models how control is transferred from one node into 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

Algorithms used

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.

2) **Implementing Function Pointers -** We don't need to write huge if/else conditions about which parser to use like using add/adds/mul parsers.

3) **Detecting conditional jumps (if clause) -** 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. Hence, this kind of a situation can be handled using if clause.

4) **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.

5) **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

6) **Detecting conditional jumps (if-else clause)** The system, by default tries to detect any jump from top to bottom as an if clause. However, at times, two different if clauses might intersect. In that case, it will probably be an if-else clause.

7) **Detecting function calls :** Instructions of the format blx 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.

8) **Processing individual nodes:** Sequential Translation - As per the design obtained at the end of Phase 3, a node contains a set of sequential statements. Hence, those statements can be translated directly, line by line. For example, ADD r1,r1,r2 can be translated to register1 = register1+register2; in C.

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
- Supports int and char data types
- Support for function calls using blx has been added

- Support for commands like add, sub, mul, adds, cmp, and b* (* represents comparison condition) 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 a while/if loop.