Technical Documentation
ARM Assembly Disassembler

Naveen Prabakar

August 3, 2025

1 Overview

This project implements a simple ARM-like assembly language disassembler that reads binary machine code from a file, decodes the instructions, and outputs their string assembly representation.

The program processes a binary input file, interprets 32-bit instructions according to ARM instruction formats, and translates them into human-readable assembly mnemonic instructions with their operands.

2 Key Components

2.1 Classes

`PA2` Main entry class:

- Reads machine code bytes from input file.
- Processes each 4-byte instruction.
- Uses `Register` and `Output` classes to decode instructions.
- Prints disassembled assembly instructions sequentially.
- Detects labels for branches and inserts appropriate labels in output.

`Register`
- Encapsulates bit manipulation logic for a 32-bit binary instruction.
- Provides methods to extract specific parts of the instruction such as registers (Rd, Rn, Rm), immediates, addresses, and shifts.
- Contains utility to convert register numbers to their assembly register names (e.g., SP, XZR, FP, LR, or X0–X31).
- Handles sign extension and conversion of immediate values for negative numbers based on instruction type.

`Output`
- Maintains a lookup table to map opcode binary values to instruction mnemonics.
- Provides a method to identify which opcode form an instruction matches.
- Decodes the binary instruction into the appropriate assembly language mnemonic with correct operands using the `Register` class.
- Handles formatting of branch labels and condition codes.
- Returns the string representation of the decoded instruction.

3 Program Flow

1. **Reading the Input:**
   - The `PA2` class opens a binary input file containing the machine code.
   - The file is read 4 bytes at a time (32 bits per instruction).
2. **Instruction Decoding:**
   - For each 4-byte instruction read, an `Register` instance is created to analyze the fields.
   - An `Output` object uses the extracted opcode bits and the `Register` methods to decode to the corresponding assembly instruction.
   - The opcode-to-mnemonic mapping is done using a hashtable initialized in `Output`.
3. **Label Handling:**
   - Branch instructions compute target addresses with sign-extended offsets.
   - Labels are generated for jump targets to improve human readability.
   - Labels are inserted before instructions at indicated locations.
4. **Output Generation:**
   - The resulting assembly instruction strings are printed to standard output sequentially.
   - Label identifiers (e.g., `label1:`) are included as appropriate.

4 Important Implementation Details

4.1 Opcode Decoding Logic

- `Output.checkopcode(int)` extracts and matches opcode bits of various lengths (6, 8, 10, 11 bits) against a hashtable of supported opcodes.
- Once opcode is identified, it uses that mnemonic to format operands.
- For I-type (immediate) instructions like `ADDI, ANDI`, immediates are decoded with sign extension.
- Branch instructions handle conditional flags and construct labels with offsets.

4.2 Register Encoding

- Registers are decoded from specific bit fields:
  - Rd, Rn, Rm extracted at fixed bit positions.
  - Special registers like SP, FP, LR, and XZR are named appropriately.

4.3 Sign Extension

- Immediate values for instructions that use them (like `ADDI`) are sign-extended as per ARM encoding rules.
- This ensures that negative immediates are correctly parsed for relative addressing and arithmetic.

5 Usage

Compile and run the program with the path to a binary machine code file as the first argument:

```
javac PA2.java Register.java Output.java
java PA2 input.bin
```

The program outputs the assembly language equivalent of all instructions in the file with labels denoting branch targets.

6 Potential Extensions and Improvements

- Support more ARM instruction types and extended mnemonics.
- Enhance label resolution and output formatting.
- Add error handling for invalid or unsupported instructions.
- Generate output to file rather than console.
- Add unit tests for opcode extraction and assembly conversion.

7 Summary

This project provides a straightforward ARM assembly disassembler written in Java, demonstrating bitwise instruction decoding, label management, and output formatting. It is well-structured into classes separating concerns for register field extraction and instruction decoding, suitable for educational or prototyping purposes.