ARM Project

Group 39
Tom Price, Sarthak Kumar, Shiraz Butt and Andy Hume

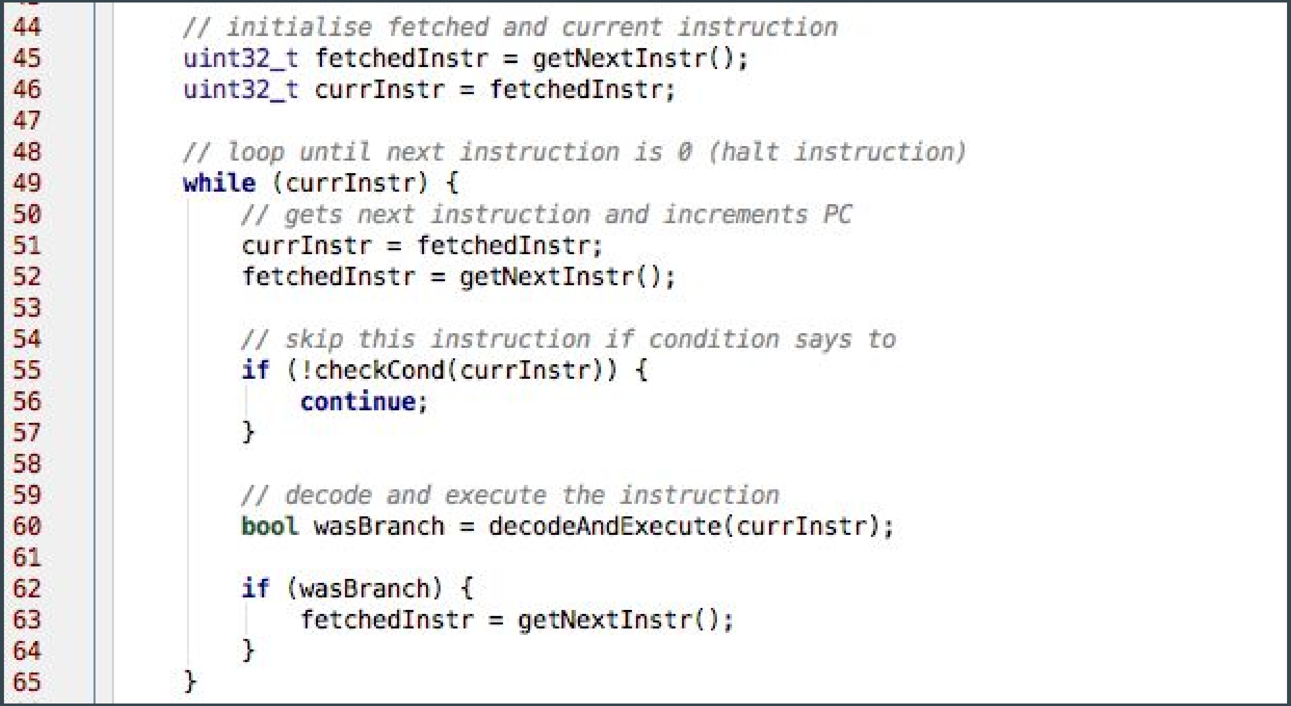
Overview

- Tests
- Emulator structure
- Assembler structure
- Extension
- Reflections

Tests

Emulator

Design



```
    // initialise fetched and current instruction
    uint32_t fetchedInstr = getNextInstr();
    uint32_t currInstr = fetchedInstr;

    // loop until next instruction is 0 (halt instruction)
    while (currInstr) {
        // gets next instruction and increments PC
        currInstr = fetchedInstr;
        fetchedInstr = getNextInstr();

        // skip this instruction if condition says to
        if (!checkCond(currInstr)) {
            continue;
        }

        // decode and execute the instruction
        bool wasBranch = decodeAndExecute(currInstr);

        if (wasBranch) {
            fetchedInstr = getNextInstr();
        }
    }
```

Good point

1. Passed all the test cases

Bad points

- Global variable for the memory and registers
  - No encapsulation
  - Difficult to debug
  - Restricted design options (couldn’t put the Armstate on the stack)
- Really slow: loop01 timed out

Bad points

- Not extendable
  - When handling branch instructions we had to change main
  - Can’t emulate multiple programs
- Duplicated code
- More macros than lines of code
- More bugs than lines of code

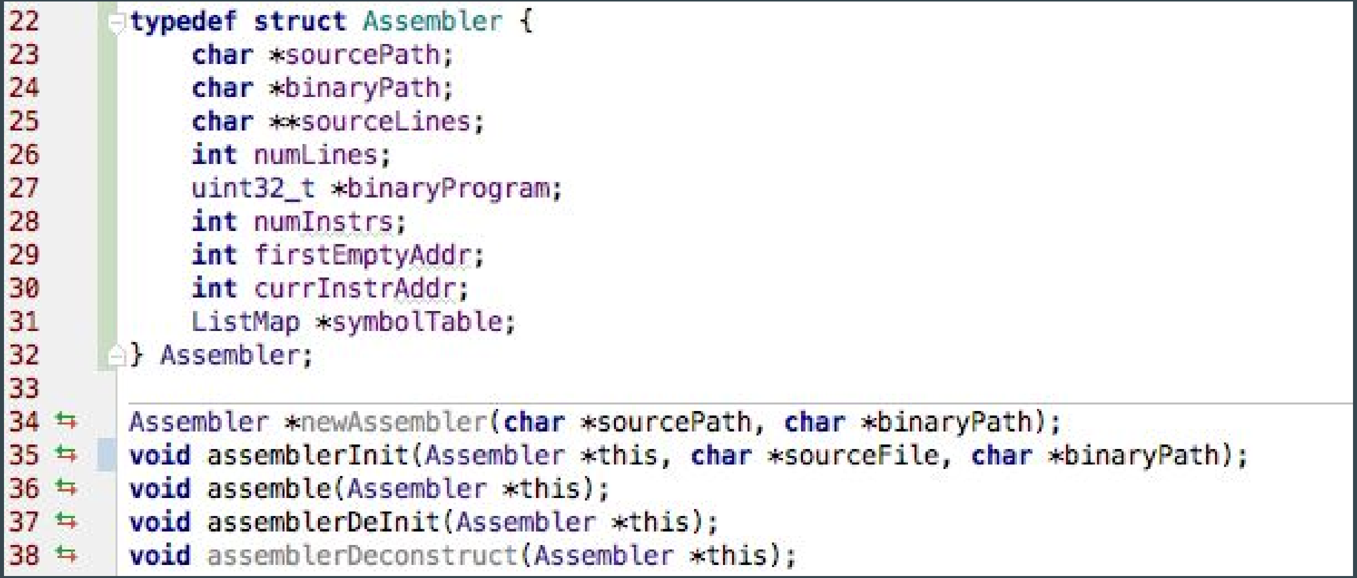
Assembler

Design

Java > C

Design

Assembler ‘Class’

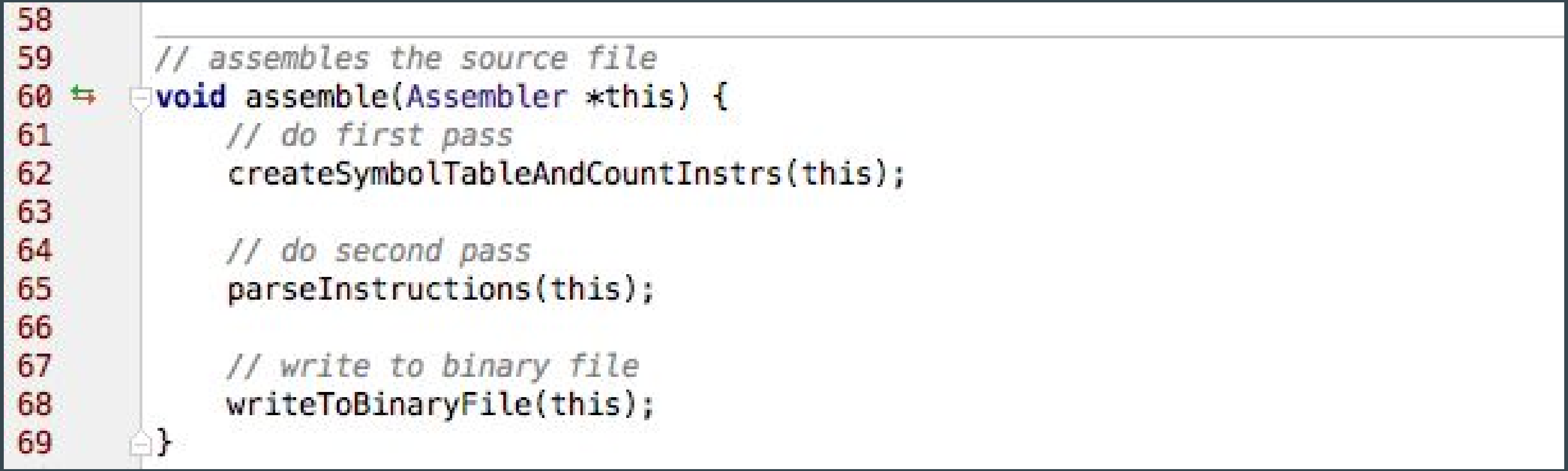


```
typedef struct Assembler {
    char *sourcePath;
    char *binaryPath;
    char **sourceLines;
    int numLines;
    uint32_t *binaryProgram;
    int numInstrs;
    int firstEmptyAddr;
    int currInstrAddr;
    ListMap *symbolTable;
} Assembler;

Assembler *newAssembler(char *sourcePath, char *binaryPath);
void assemblerInit(Assembler *this, char *sourceFile, char *binaryPath);
void assemble(Assembler *this);
void assemblerDeInit(Assembler *this);
void assemblerDeconstruct(Assembler *this);
```

Design

Assemble Function



```
// assembles the source file
void assemble(Assembler *this) {
    // do first pass
    createSymbolTableAndCountInstrs(this);

    // do second pass
    parseInstructions(this);

    // write to binary file
    writeToBinaryFile(this);
}
```

First Pass

1. Count number of lines in the source file for source lines array, read them into it
2. Iterate through lines
    a. Strip leading space and comments, skip if line is empty
    b. Add labels to symbol table
    c. Count number of instructions while going

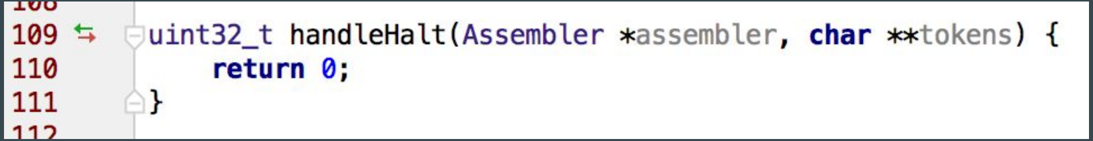
Second Pass

Iterate through the source lines array:

1. Break line into tokens
2. Pass to appropriate handling function based on mnemonic using function table
    a. Use common components (operand2 for DP and address for SDT very similar) to work out fields
    b. If we need to add a constant to the end of the file, use and update nextFreeAddr variable, calculated based on the number of instructions present in the source file
    c. Pass the fields to instruction generators that make the 32 bit instruction
3. Write all instructions out to the file when done

Design

- Token handlers - 👍
  - Gives opportunity to greatly simplify some instructions (like halt)



```
uint32_t handleHalt(Assembler *assembler, char **tokens) {
    return 0;
}
```

- Map for the symbol table and function lookup
  - Use void pointers to make map generic - some complications
  - ISO C forbids casting from void pointer to function pointer
  - Hacky workaround - void ** to function pointer pointer, then dereference

Extension

3 bit counter

Compiler!

Reflections