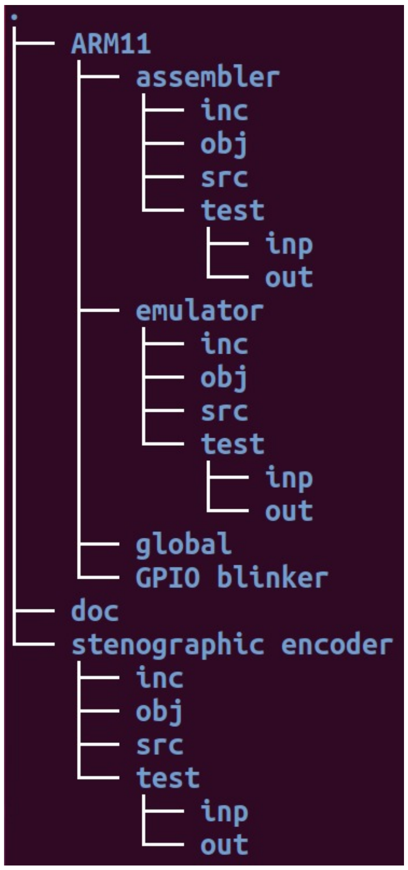
ARM11 Final Report

Raul Patel, Joshua Priestley, Rohan Gupta and Gavin Wu

July 8, 2020

1 Project Repository

- `inc/` directories contain `*.h` header files which hold important structs, enums, aliases and constants used in the parent program.
- `obj/` directories contain `*.o` object files generated from compiling the parent program.
- `src/` directories contain the `*.c` source code files of the parent program.
- `test/` directories has two sub-directories `inp/` and `out/` which respectively contain unit test files and the results from the tests.
- `./ARM11/` is a project directory, containing the parts I, II, III defined here.
- `./stenographic encoder/` is a project directory, containing our extension which function is defined here.
- A `Makefile` is found under each program directory. This is used to compile, clean and run tests on a parent program. To use, run commands `make all`, `make clean`, `make test` and `make test-clean` in terminal.



```
.
├── ARM11
│   ├── assembler
│   │   ├── inc
│   │   ├── obj
│   │   ├── src
│   │   └── test
│   │       ├── inp
│   │       └── out
│   ├── emulator
│   │   ├── inc
│   │   ├── obj
│   │   ├── src
│   │   └── test
│   │       ├── inp
│   │       └── out
│   ├── global
│   └── GPIO blinker
├── doc
└── stenographic encoder
    ├── inc
    ├── obj
    ├── src
    └── test
        ├── inp
        └── out
```

Project Tree

2 ARM11 Programs (`./ARM11/`)

Both Emulator and Assembler share source code and config files which can be located in `./ARM11/global/`

2.1 Assembler (`./ARM11/assembler/`)

2.1.1 Explanation

The assembler program reads ARM assembly code from an `*.s` file (argument 1), parses and translates each instruction into machine code (binary code) one by one. Once the entire source code is translated it is written to a specified destination file (argument 2).

2.1.2 Implementation

See Figure 1 in the Appendix (Page 7) for a detailed project data structure diagram.

- `assemble.c` contains the `main(..)` function, and takes 2 command line arguments: the first being the ARM assembly source code, and the second being the output path where the translated machine code will be written. These files are then safely opened and created in a binary mode, and are passed as parameters to `{tokenize(..)->tokenizer.h}`.

- `tokenizer.c` contains `tokenize(..)` and `convert_instruction(..)`.

  `tokenize(..)` handles the tokenization process based on the two-pass assembler algorithm. The first pass removes irrelevant code (e.g. comments, white space) and parses label instructions, storing their corresponding memory addresses in a symbol table (see SymbolTable in 1). The second pass iterates through each instruction, converting each line to machine code using `{convert_instruction(..)->converters.h}`.

- `converters.c` contains `convert_instruction(..)` and helper functions (`convert_data_processing(...)`, `convert_multiply(...)`, `convert_single_data_transfer(...)`, `convert_branch(...)`, `convert_shift(...)`) to respectively convert each type of instruction (data processing, multiply, single data transfer, branch and shift) into it's 32-bit machine code equivalent.

  `convert_instruction(..)` first maps the instruction to it's specific type using `instructionMap` and then calls the corresponding helper function. There are some unique cases such as: the `andeq` instruction instantly returning 0, branch instructions requiring the SymbolTable and current line address (for calculating offsets), and single data transfer instructions requiring the `FileBuffer` (see FileBuffer 1) (for writing buffered data to the `eof`).

  All the helper functions follow a similar pattern: the full instruction of ARM assembly is split based on delimiter ",". For example, `add r2,r1,#2` would be split into tokens: $\boxed{\texttt{add}}$ $\boxed{\texttt{r2}}$ $\boxed{\texttt{r1}}$ $\boxed{\texttt{\#2}}$ represented through iterations 0, 1, 2, 3 respectively. We then process each token individually.

- `parse_utils.c` contains `parse_register(..)` and `parse_immediate(..)` for use throughout the program, in order to help parse string instructions. `parse_register(..)` retrieves the integer part of a register token (e.g `parse_register("r11")` returns 11). `parse_immediate(..)` retrieves a decimal version of the immediate part of a `<{#/=}immediate>` token, (e.g. `parse_immediate("#0x12")` returns 18, and `parse_immediate("#12")` returns 12).

- `maps.c` contains `get_map_value(<map>,<key>)` which is self explanatory.

- `sym_tab.c` creates the ADT for the symbol table, and implements functions for initialisation, insertion and freeing of its dynamic fields. The table increases size dynamically for the best memory efficiency.

- `file_buffer.c` contains utility instructions for creating, adding and deleting from a file buffer. The file buffer is utilized by the `ldr Rd <=expression>` to store the value of `<expression>` at the `eof`.

Headers for each `*.c` are stored in a `./inc/` folder (below are the important headers):

- `maps.h` contains constant null terminated map declarations `instructionMap`, `dpiMap`, `branchMap` and `shiftMap` which each map a string operand (key) to a integer code (value).

2.2 GPIO Blinking LED (`./ARM11/GPIO blinker/`)

The ARM source code for the blinking LED program is located in `./ARM11/GPIO blinker/gpio.s`. As we had no Raspberry Pi to test if the implementation works, the solution Raul came up with is completely theoretical. We have however, managed to build a unit test, using our previous programs `./emulate` and `./assemble` to prove that our solution will work with real ARM11 processor. To test this for yourself run the following command in terminal `cd ARM11/GPIO blinker/ && make all`, this will compile the `gpio.s` into `gpio.bin` and then emulate an ARM11 processor executing each instruction. Type `make clean` once done to clean up the files generated.

3 Extension: Digital Steganographic Encoding Using Cryptography

3.1 Description

Steganography is the practise of hiding sensitive data in plain sight. For our extension we set out to developing a tool such that we could conceal any type of file within a .BMP image, whilst ensuring the new altered image is essentially indistinguishable from the original. To challenge ourselves, we took the approach of only producing legacy code (without the use of libraries).

Encoding and decoding data from .BMP images is actually quite simple. Every pixel in a BMP image is represented by 3 bytes of code (red, green and blue), which ultimately determines the colour of that pixel. The algorithm we developed looks at both the size of the file to encode and the size of all the pixels in the image, and then determines the smallest bit encoding level possible to store the entire file within the image. The bit encoding level determines how many LSB bits of each pixel will be removed in order to store the data (the smaller the better). The encoding process works by using the first few bytes of the image to store meta data (necessary for decoding), and subsequently removing the last few bits of each pixel (at the bit encoding level). After this, the data to be encoded is iteratively stored in the LSB of each pixel.

The decode process works inversely; by first extracting the meta data which includes: the original file name, bit encoding level, and original file size. The last few bits of each pixel (at the extracted bit encoding level) are then pulled from the image, and rebuilt into the original file.

Cryptography and steganography are both highly effective ways of protecting secret information - whilst steganography deals with hiding the existence of data, cryptography is able to hide the meaning of data. In the case that our encoding mechanism could be detected by some sort of AI program and somehow decoded, we agreed upon the need for a high level of encryption to prevent the original file from being deciphered. Therefore we decided that each byte of data to encode will be encrypted using a secure yet efficient encryption algorithm before encoding it into the image.

We discussed the advantages of some major encryption algorithms and considered whether each one would meet our design specifications. Ultimately we decided upon the Advanced Encryption Standard (AES) algorithm, which has benefits of sporting an excellent trade-off between efficiency, security and most importantly having 1-1 binary conversion (hence it does not increase the file size). There are 3 flavours of the AES algorithm: 128-bit, 192-bit and 256-bit. This describes how many bytes the AES algorithm encrypts in each round. For our program, and most commercial applications, AES-128 was the most suitable choice. One of the advantages of AES is that it uses a unique key each round, which makes deciphering the code extremely difficult. For example, for a file of size 5MB, it generates over 3 million unique keys during the encryption process (since there are 10 rounds for every 128 bytes of encryption).

Furthermore, to enhance the security of the encryption process, we implemented a password encryption process. This password can be optionally specified using the flag `-p <password>`, and this password is converted into the starting key for the encryption process using a hashing algorithm.

Our program is able to encode files of literally any data type. An example is in `./test/out/shakespeare.bmp`. This image has the entire works of Shakespeare (in the form of a `.txt` file) encoded within itself using the password `romeo`. To prove this, run in terminal `./extension -d -i test/out/shakespeare.bmp -p romeo` and you will find a file `full_shakespeare.txt` has appeared.

Extension can be run using the following terminal commands (Note : password is optional):
Encoding: `./extension -e -i <*.bmp> <any file> -o <*.bmp> -p <password>`
Decoding: `./extension -d -i <*.bmp> -p <password>`

3.2 Encryption Process

Before encryption and decryption can begin, the specified password must be converted into a key. To calculate the 16-byte starting key, we used 2 different hashing functions that convert the password string into 2 8-byte values and combined them together to form the unique 16-byte starting key, since `gcc` doesn't support 128-bit integers on all architectures.

Once we have calculated the key from the password, we can begin the AES encryption process. The Rijndael block cipher algorithms won the 2001 National Institute of Standards and Technology's, and thus became dubbed the Advanced Encryption Standard. Figure [1] gives a simple visual representation of the inner workings of the AES encryption algorithm.

As a very brief summary of AES encryption, 16 bytes of data, which can internally be thought of as a 4x4 matrix, are first XORed with the round key in a process called `add_round_key()`. Next, the result then goes through a sequence of `sub_bytes()`, `shift_rows()`, `mix_columns()` and `add_round_key()` processes a total of N times, with

N being 10 for AES-128. The `sub_bytes()` operation represents a non-linear substitution step, whilst `shift_rows()` represents a transposition step whereby each row of the 4x4 matrix is shifted cyclically a different number of times. `mix_columns()` was particularly difficult to implement, and represents matrix multiplication by a predetermined matrix inside the Galois Field 2^8 - this provides diffusion in the columns. In the final round, the `mix_columns()` operation is not performed.

Crucially, the key changes continuously to ensure secure encryption, hence the name ”round key” in AES. For every 16-bytes of data, the data goes through 10 rounds, and during each round the key changes to become a new computed value based on the previous round through a series of complex sub-byte look-ups, rcon operations and many XOR operations.

The decryption processes involves backtracking by undoing each step of the encryption process. Every operation used for encryption has its own inverse, and undoing XOR was especially easy since it is its own inverse. However, reversing the key schedule process proved to be especially challenging; not only was it required to calculate the key after 10 rounds, but inverting the round key process also proved to be highly challenging, since each version of the new key is calculated using the previous key.

As a result of our rigorous password hashing and complex encryption process, if a user enters even a single character of the password process incorrectly then the resulting file output will be completely unreadable. Furthermore, large files of several Megabytes can be encrypted and encoded (or decoded and decrypted) within seconds, due to the highly efficient algorithms we implemented.

3.3 Design Implementation

- `main.c` parses command line arguments and flags as described above 3.1.
- `bmp.c` contains helper methods for opening, reading and saving BMP images. It's header `bmp.h` contains important structures `BMPheader` and `BMPImage` which are respectively used to store the BMP's meta data (size, width, height, bits per pixel) and image data (pixel data).
- `encode.c` will `encode(..)` a given image by first calculating the least bit encoding level using (`calculate_bit_encoding_level(..)`), and then iteratively masking these LSB bits of each pixel.
- `decode.c` contains `decode(..)` to exact an inverse of the `encode(..)` function. This works by iteratively pulling the LSB of each pixel to populate an array of bytes, which will recreate the original data.
- `encode_utils.c` contains utility helper functions required for the AES-128 encryption algorithm.
- `encrypt.c` contains functions to `encrypt(..)` and `decrypt(..)` data. It also contains a helper function `hash_password(..)` to convert a string password into 16 byte key.
- `utils.c` contains utility functions for encoding and decoding data structures. The `steno_data` structure is used to store and retrieve meta data that has been encoded/decoded from an image.

3.4 Testing of Extension

Memory Checks We regularly used Valgrind to identify any memory leaks and ensure all allocated memory was being freed before program termination. There are no memory leaks in our extension.

Encoding & Decoding As encoding and decoding are the inverse of each other, we were able to test how well our extension was doing by first encoding a file, and checking if decoding the result gave us back the original file. We encoded and decoded a large range of file types, including `.txt, .pdf, .zip` and even `.wav` files and concluded that in every case decoding the image resulted in a perfect replica of the original file.

Problems in Testing After encoding many different file types, we soon discovered that encoding files with a size greater than around 12% of the maximum .bmp image resulted in a segmentation fault, due to an error in calculating the bit encoding level. This did not impact the overall program, but limited some essential functionality of our extension. We reported this error and gave Raul enough time in advance to revise his algorithm to solve this issue.

4 Group Reflection

With this being our first group project at university level and having only formed a team the day before the group formation deadline, we were understandably anxious about the synergy between our members. Fortunately, team organisation and collaboration went much better than we had anticipated. We kept a realistic attitude towards deadlines, and made effective use of each member's strengths: Raul's proficiency in programming allowed him to be a key player in many programming sections, Joshua's skills complemented producing the report and bug-fixing, whilst Gavin and Rohan were able to power through the design and implementation of the assembler. Additionally, we each contributed in equal proportions to writing the report and producing the presentation.

All this is not to say we did not encounter any difficulties. Many of our members were inexperienced with using Git, and had to slowly get to grips with its basic functionality. Raul was well-versed in Git from his previous projects, and was keen to lend his support in getting us up to speed with branches, and soon enough, we were able to work individually on different functionality within our own branches, merging to master when necessary. Overall, the group project has been invaluable in getting to grips with Git, and this knowledge will undoubtedly prove useful for our careers and beyond.

Conforming to the classic programmer stereotype, it was clear we had more night owls than early birds. Consequently, peak project productivity was seen later in the afternoon, and at times it was difficult to communicate about the project in the morning. Perhaps when working on future projects, it would be better to schedule strict programming times so that work is divided more equally between group members. A pair programming system could also help us to coordinate code better so we would not have so many different versions of the code which becomes confusing. It is clear that an improvement for future projects and beyond, we will need to improve our work schedule both for productivity as well as health, but this has been a learning experience nonetheless.

The extension was also notoriously hard to manage, and in hindsight better planning would have been helpful for improved group coordination. Initially, after we collectively agreed on Raul's idea to implement a steganography tool, he enthusiastically raced ahead with the research and implementation, and this resulted in some members finding it difficult to keep up with the rapid pace of development. Ultimately however, whilst Gavin and Raul worked on the implementation of the steganography program, Rohan and Joshua played a crucial role in the testing phase and work was split up evenly.

This was not helped by the fact that as the impending final deadline approached ever closer, it was difficult for members to communicate effectively between one another as there were many tasks being worked on simultaneously. In future projects, it is important for our teams to be managed using an application such as Trello, which allows teams to record which tasks are being completed, which tasks have been finished and who is working on which task at any one time.

5 Individual Reflections

I believe I integrated into the group sufficiently well. Initially, I found it very difficult to grasp the structure of the code written by other team members, and so felt as though I wasn't contributing as much as others. However, after reading the spec several times and getting to grips with C, I felt as though I was equipped with being able to complete the required tasks. My average DoCPA score at checkpoint was 4.67, showing a high level of collaboration and commitment to the project.

I feel as though my strengths were on the organisation side: I regularly communicated with the group about what tasks each person was currently working on. I ensured that tasks were completed in a timely manner so that there was sufficient time to clean up any inefficient code and so that everyone could look through the final project to check that they were satisfied. I was also proficient in LaTeX, so was more than happy to take lead on the reports and final presentation slides.

Using Git turned out to be one of my biggest weaknesses: I often found it difficult to keep up with and merge my changes with the different tasks other team members were working with. However, this was minimised by having Raul do most of the Git organisation (creating branches, merging to master, etc.). In addition, I often felt as though a lot of my group members were ahead with progress compared to myself. Perhaps next time I should take a more "hands-on" and proactive approach, as well as designating tasks more distinctly rather than the looser approach we took here. **- Joshua**

This project proved to be a huge educational experience for me. I had a large part to play in the creation and the optimisation of the assembler. My main strength was in my ability to debug errors and clean up code, which made refactoring easier later on. I also felt I co-operated with the other team members well, by having individual calls explaining changes I had implemented and regularly updating the group.

However, I found I had some periods where I was very productive and got lots of work done and others where I hit a brick wall and was stuck on one thing for a long time. This meant my contributions were periodic and I let myself fall behind on progress in the early stages of the emulator implementation.

If I were to repeat the project, I would suggest we implement a pair programming strategy, because it often got confusing with everybody working on separate versions of the code. I feel I would work better discussing the problem whilst coding, as this is similar to how I approached the lab exercises this year. **- Rohan**

I believe I contributed effectively to the dynamics of the group, playing a key part in the programming of the assembler program, as well as contributing substantially to the report, and this is reflected by my admirable DoCPA score. I played a central role in the design of the structure of the assembler program, and offered help to those who were unsure how to implement particular functions. In hindsight, I realise the vital importance of providing documentation so that other developers on my team can read up on the structure of the program, instead of trying to figure it out themselves.

The highlight of the coursework project for was definitely implementing the AES encryption process in the extension. Despite it being a challenging process, I not only gained extra familiarity with the C language but developed an appreciation of the high degree of clever maths that make up the foundations of the Computer Science world. The Rijndael encryption process is notorious for making heavy use of Galois Field mathematics, which I am glad to have been able to implement in my programming.

One of the most important lessons I learned during this coursework was the importance of testing early on and incrementally. Changing my habits to test more frequently actually sped up my development time since I spent less time debugging! **- Gavin**

I am extremely happy with not only my group's performance, but my own as well throughout this project. I believe that I integrated well within my team and made a significant contribution to each aspect of this project. Taking on the role as development manager, I am proud of my ability to manage a team online and organize a simple workflow to produce high quality code. This is well reflected not only by my DocPA scores but also in the performance of our programs.

Being confident in my knowledge of C and git, I urged my team to spend the first week familiarising themselves with C and the specification , whilst I began to setup our project repository, create rules (for style, merging, testing, etc) and research pre-existing implementations of emulators and assemblers. I then began to plan simple pseudo solutions (see 1) to the emulator. I tried to create a high enough level of abstraction in the solutions, so my team could quickly get to grips with tasks they were given without the need to have a full understanding of the entire program.

I pushed the idea of using agile development as I believed that with an open-line of communication this would have been easily doable. However, I soon discovered how tasking it is to track the progress each team member made. In fact, our biggest setback came during the assembler development, where we started to overwrite each others code, causing a complete loss in a sprint cycle as we had to spend days debugging and solving conflicts. I believe this was largely due to my lack of communication with the team (definitely my greatest weakness), and if I were to redo this project, I would instead make use of issue boards and ticket systems to assign and track tasks.

At the end of each sprint cycle I would analyse and merge code into the production branch. This produced huge benefits, as at one point I realised the code was becoming extremely messy, and I proposed a complete rework of the assembler. This, unsurprisingly, was met with some resentment, however once complete, it resulted in a reduction 500+ lines of code and a 100% performance speed up.

If I were in a different group I would again push for agile development as well as continuous testing, as I believe these were the primary factors that led to the success of our project. **- Raul**

6 Appendix

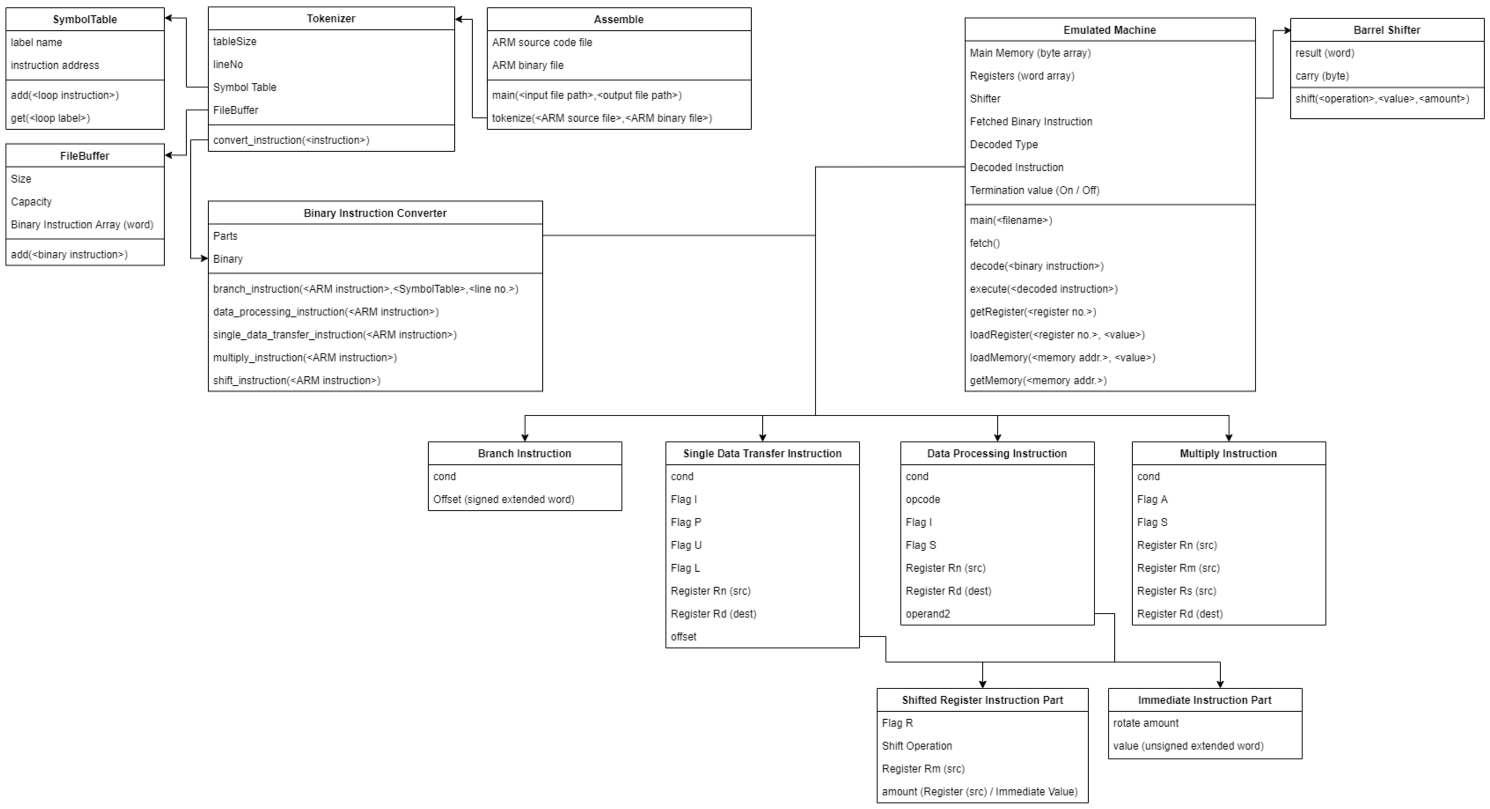

SymbolTable
label name
instruction address
add(<loop instruction>)
get(<loop label>)
FileBuffer
Size
Capacity
Binary Instruction Array (word)
add(<binary instruction>)
Tokenizer
tableSize
lineNo
Symbol Table
FileBuffer
convert_instruction(<instruction>)
Assemble
ARM source code file
ARM binary file
main(<input file path>,<output file path>)
tokenize(<ARM source file>,<ARM binary file>)
Binary Instruction Converter
Parts
Binary
branch_instruction(<ARM instruction>,<SymbolTable>,<line no.>)
data_processing_instruction(<ARM instruction>)
single_data_transfer_instruction(<ARM instruction>)
multiply_instruction(<ARM instruction>)
shift_instruction(<ARM instruction>)
Emulated Machine
Main Memory (byte array)
Registers (word array)
Shifter
Fetched Binary Instruction
Decoded Type
Decoded Instruction
Termination value (On / Off)
main(<filename>)
fetch()
decode(<binary instruction>)
execute(<decoded instruction>)
getRegister(<register no.>)
loadRegister(<register no.>, <value>)
loadMemory(<memory addr.>, <value>)
getMemory(<memory addr.>)
Barrel Shifter
result (word)
carry (byte)
shift(<operation>,<value>,<amount>)
Branch Instruction
cond
Offset (signed extended word)
Single Data Transfer Instruction
cond
Flag I
Flag P
Flag U
Flag L
Register Rn (src)
Register Rd (dest)
offset
Data Processing Instruction
cond
opcode
Flag I
Flag S
Register Rn (src)
Register Rd (dest)
operand2
Multiply Instruction
cond
Flag A
Flag S
Register Rn (src)
Register Rm (src)
Register Rs (src)
Register Rd (dest)
Shifted Register Instruction Part
Flag R
Shift Operation
Register Rm (src)
amount (Register (src) / Immediate Value)
Immediate Instruction Part
rotate amount
value (unsigned extended word)

Figure 1: Project Data Structure/UML diagram

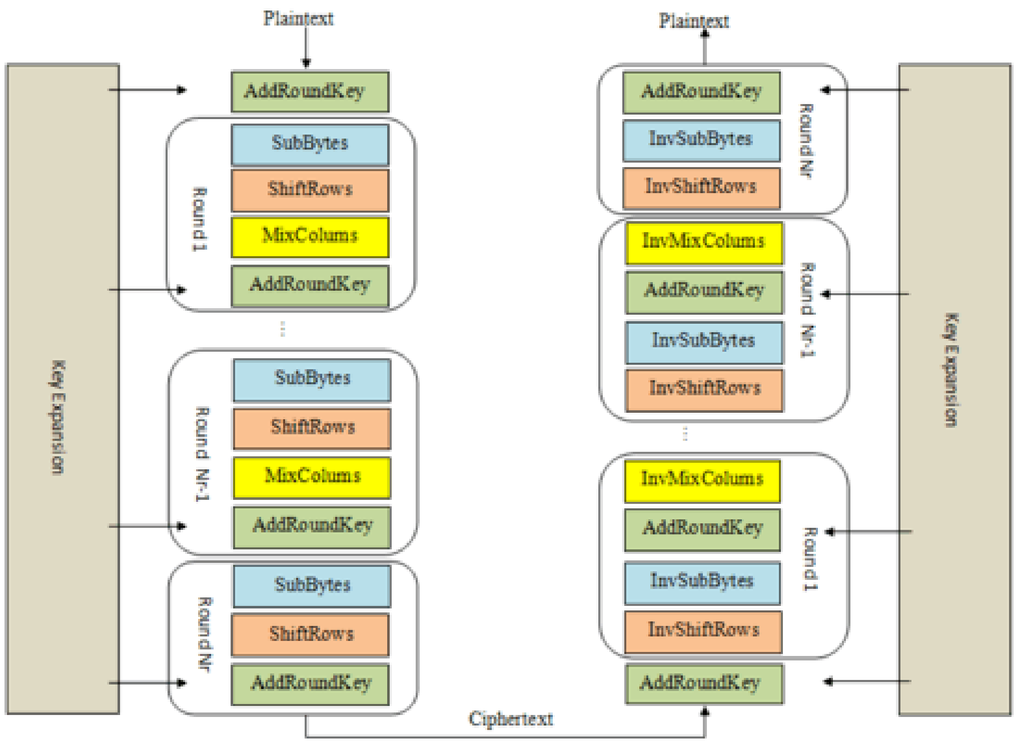


Figure 2: AES encryption algorithm [1] (Left side represents encryption, right side represents decryption)

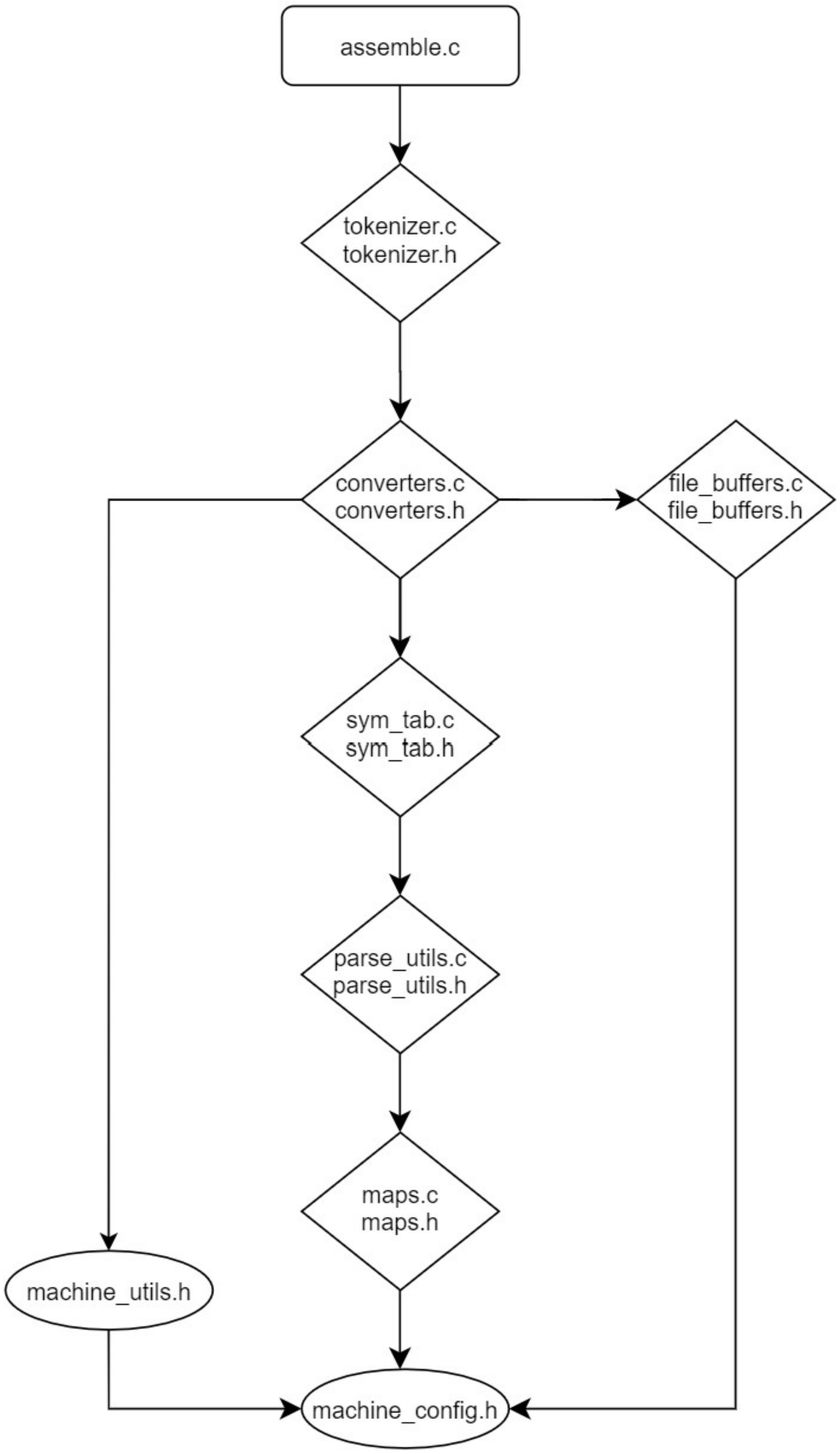


Figure 3: Dependency Tree for assembler

References

[1] K. Dhandhania H. Poston. The advanced encryption standard (aes) algorithm. *CommonLounge*, 2019. https://www.commonlounge.com/discussion/e32fdd267aaa4240a4464723bc74d0a5.