The Summer C Project and FlappyPi

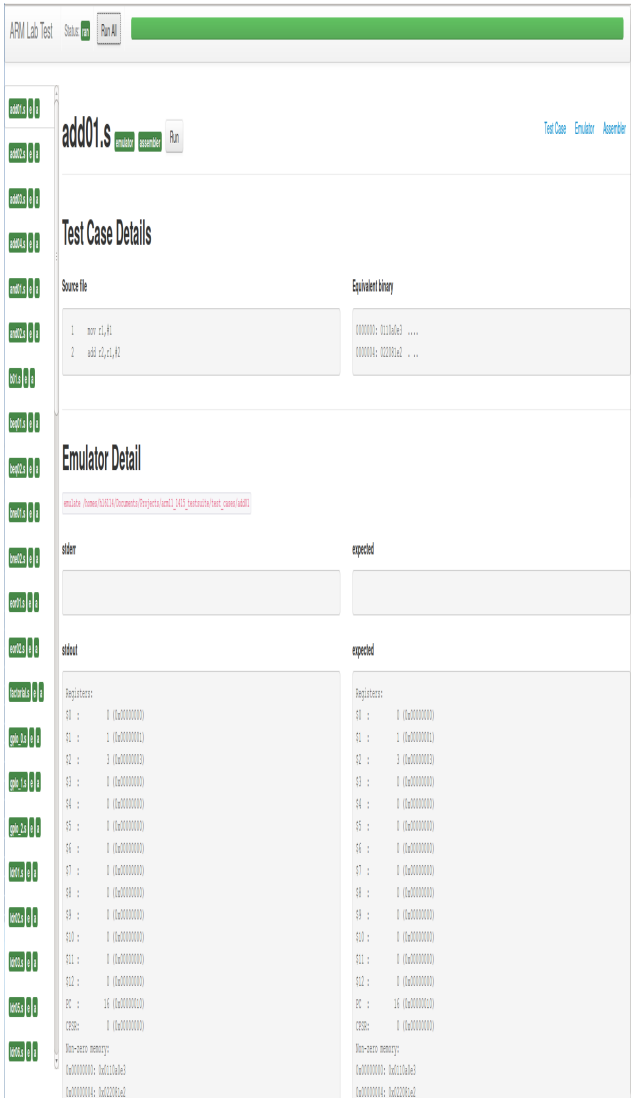
Henryk Haddass Mickey Li Michael Radigan
Oliver Wheeler

Group 16

June 16, 2015

Overview

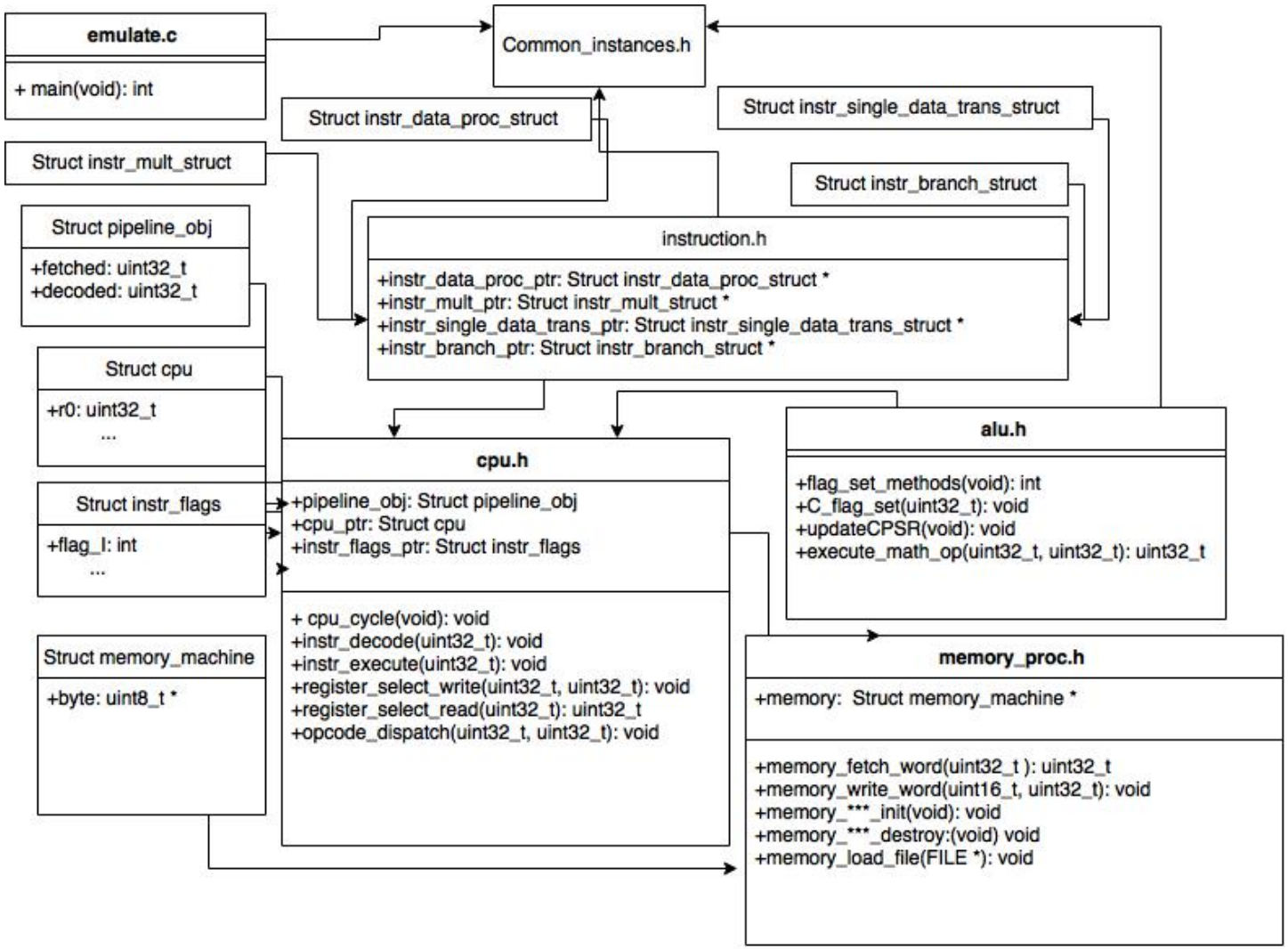
1. Emulator
   - Implementation
   - Outcome
   - Evaluation
2. Assembler
   - Implementation
   - Outcome
   - Evaluation
3. Proof of use
4. Extension
   - Implementation
   - Outcome
   - Evaluation
5. Project Evaluation
   - Evaluation



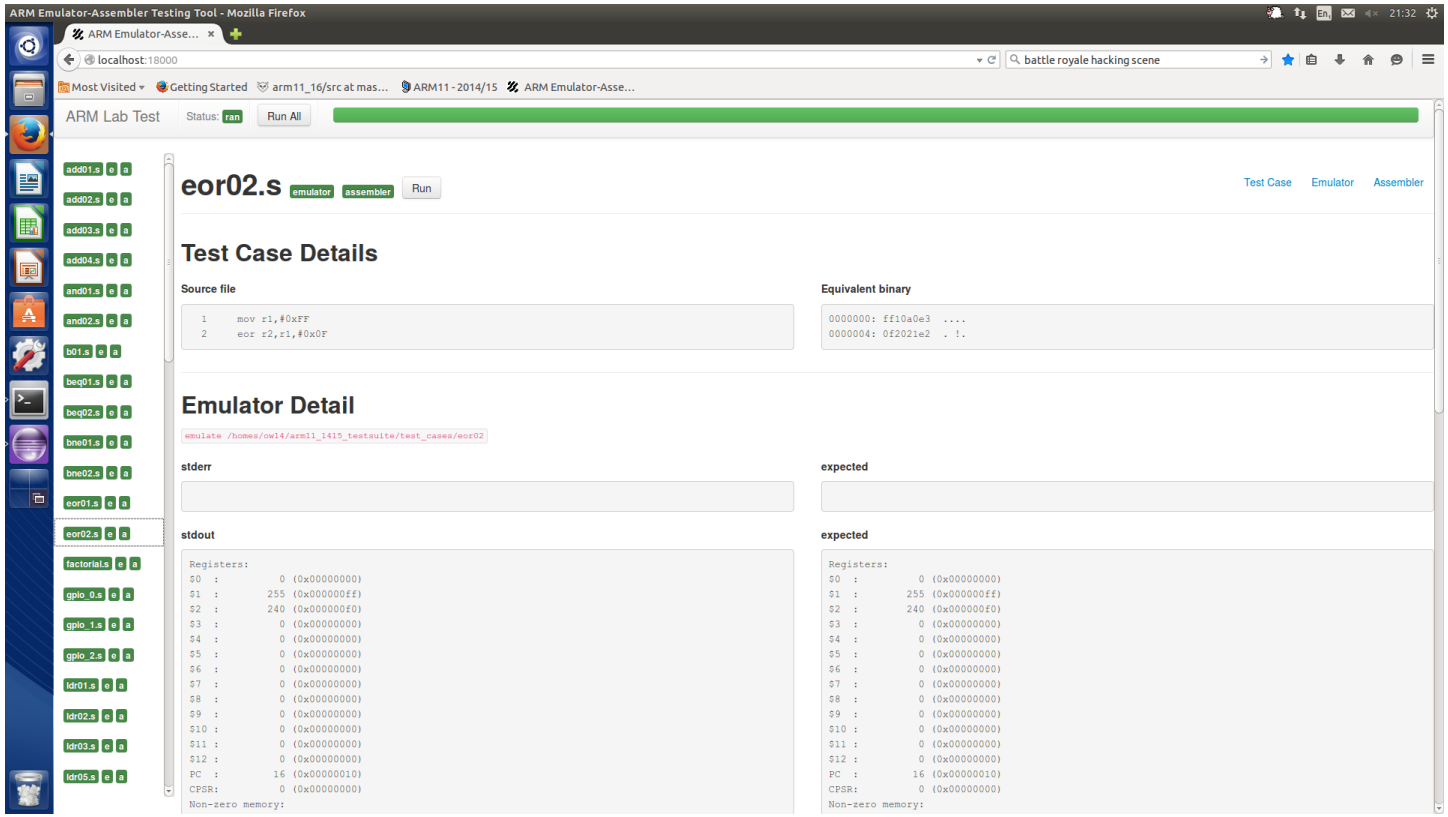
Our Approach

- We based the design on the structure of the CPU
- This allowed us to split the design into functional sections
- We emulated specific functionalities of the CPU with corresponding method names
- The code was written so as not to be repetitive

Structure


emulate.c
+ main(void): int
Common_instances.h
Struct instr_data_proc_struct
Struct instr_single_data_trans_struct
Struct instr_mult_struct
Struct instr_branch_struct
Struct pipeline_obj
+fetched: uint32_t
+decoded: uint32_t
instruction.h
+instr_data_proc_ptr: Struct instr_data_proc_struct *
+instr_mult_ptr: Struct instr_mult_struct *
+instr_single_data_trans_ptr: Struct instr_single_data_trans_struct *
+instr_branch_ptr: Struct instr_branch_struct *
Struct cpu
+r0: uint32_t
...
alu.h
+flag_set_methods(void): int
+C_flag_set(uint32_t): void
+updateCPSR(void): void
+execute_math_op(uint32_t, uint32_t): uint32_t
cpu.h
+pipeline_obj: Struct pipeline_obj
+cpu_ptr: Struct cpu
+instr_flags_ptr: Struct instr_flags
Struct instr_flags
+flag_I: int
...
+ cpu_cycle(void): void
+instr_decode(uint32_t): void
+instr_execute(uint32_t): void
+register_select_write(uint32_t, uint32_t): void
+register_select_read(uint32_t): uint32_t
+opcode_dispatch(uint32_t, uint32_t): void
Struct memory_machine
+byte: uint8_t *
memory_proc.h
+memory: Struct memory_machine *
+memory_fetch_word(uint32_t): uint32_t
+memory_write_word(uint16_t, uint32_t): void
+memory_***_init(void): void
+memory_***_destroy:(void) void
+memory_load_file(FILE *): void

Outcome


ARM Emulator-Assembler Testing Tool - Mozilla Firefox
ARM Emulator-Asse...
localhost:18000
battle royale hacking scene
Most Visited
Getting Started
arm11_16/src at mas...
ARM11 - 2014/15
ARM Emulator-Asse...
ARM Lab Test
Status: ran
Run All
add01.s e a
add02.s e a
add03.s e a
add04.s e a
and01.s e a
and02.s e a
b01.s e a
beq01.s e a
beq02.s e a
bne01.s e a
bne02.s e a
eor01.s e a
eor02.s e a
factorial.s e a
gpio_0.s e a
gpio_1.s e a
gpio_2.s e a
ldr01.s e a
ldr02.s e a
ldr03.s e a
ldr05.s e a
eor02.s
emulator
assembler
Run
Test Case
Emulator
Assembler
Test Case Details
Source file
1 mov r1,#0xFF
2 eor r2,r1,#0x0F
Equivalent binary
0000000: ff10a0e3 ....
0000004: 0f2021e2 . !.
Emulator Detail
emulate /homes/ow14/arm11_1415_testsuite/test_cases/eor02
stderr
expected
stdout
Registers:
$0 : 0 (0x00000000)
$1 : 255 (0x000000ff)
$2 : 240 (0x000000f0)
$3 : 0 (0x00000000)
$4 : 0 (0x00000000)
$5 : 0 (0x00000000)
$6 : 0 (0x00000000)
$7 : 0 (0x00000000)
$8 : 0 (0x00000000)
$9 : 0 (0x00000000)
$10 : 0 (0x00000000)
$11 : 0 (0x00000000)
$12 : 0 (0x00000000)
PC : 16 (0x00000010)
CPSR: 0 (0x00000000)
Non-zero memory:
expected
Registers:
$0 : 0 (0x00000000)
$1 : 255 (0x000000ff)
$2 : 240 (0x000000f0)
$3 : 0 (0x00000000)
$4 : 0 (0x00000000)
$5 : 0 (0x00000000)
$6 : 0 (0x00000000)
$7 : 0 (0x00000000)
$8 : 0 (0x00000000)
$9 : 0 (0x00000000)
$10 : 0 (0x00000000)
$11 : 0 (0x00000000)
$12 : 0 (0x00000000)
PC : 16 (0x00000010)
CPSR: 0 (0x00000000)
Non-zero memory:

Problems and Challenges

Design

We were initially unsure of the most efficient (or readable) way to design our implementation.

I want to break free

We had difficulty freeing pointers in the correct place, leading to segmentation faults

Debugging

It took us the same amount of time to debug our code as to write it. We debugged our code by using Eclipse, which was useful in showing us which variable values we had at any given time. Implement a pipeline at the start not at the end, which caused us offset issues

Improvements

- Make our code more compact and not be as explicit
- Write our code to be reusable
- Plan THEN code
- A good night's sleep - for general health

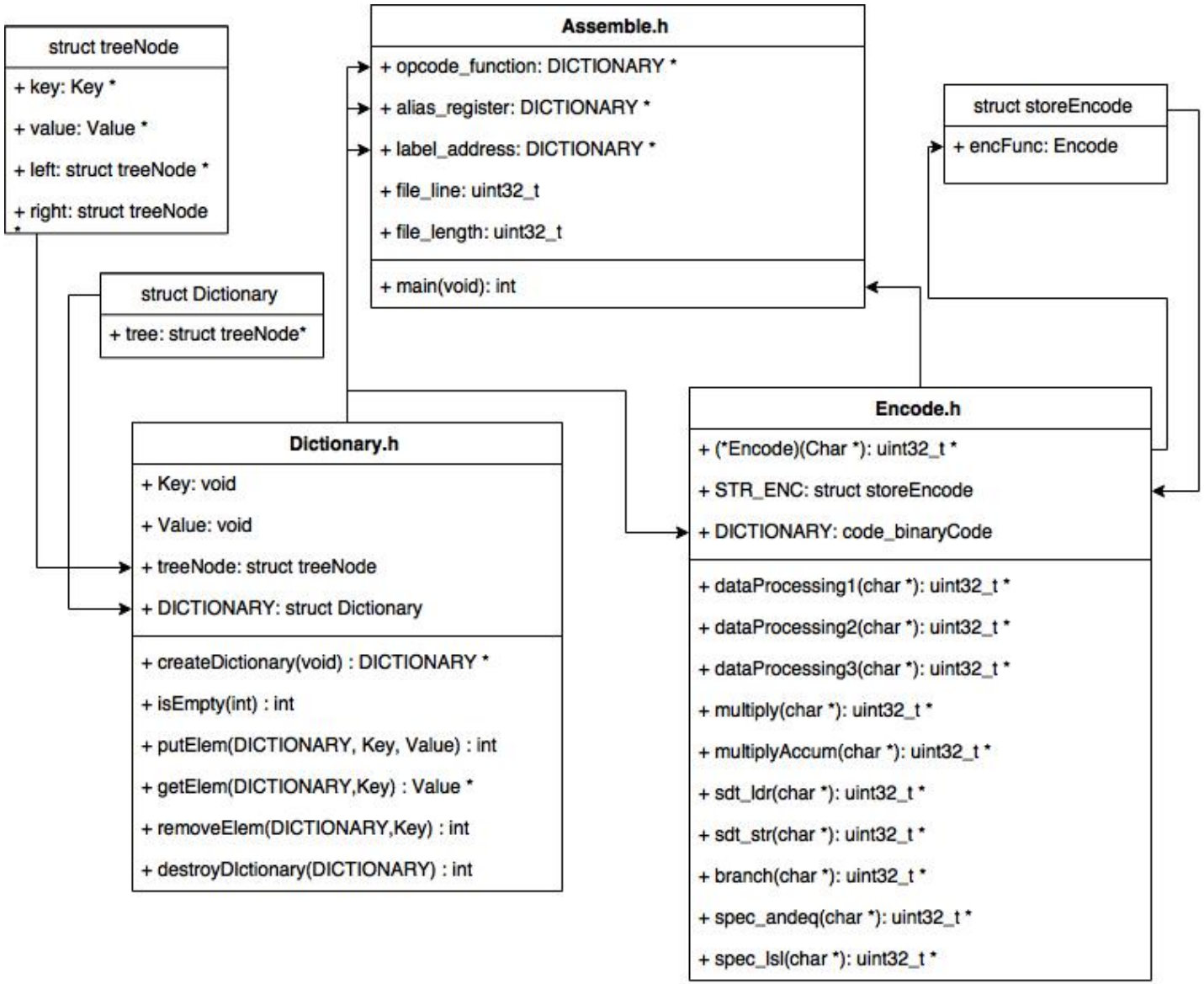
Our Approach

We used a simple 3 modules approach - *assemble.c*, *dictionary.c* and *encode.c*.

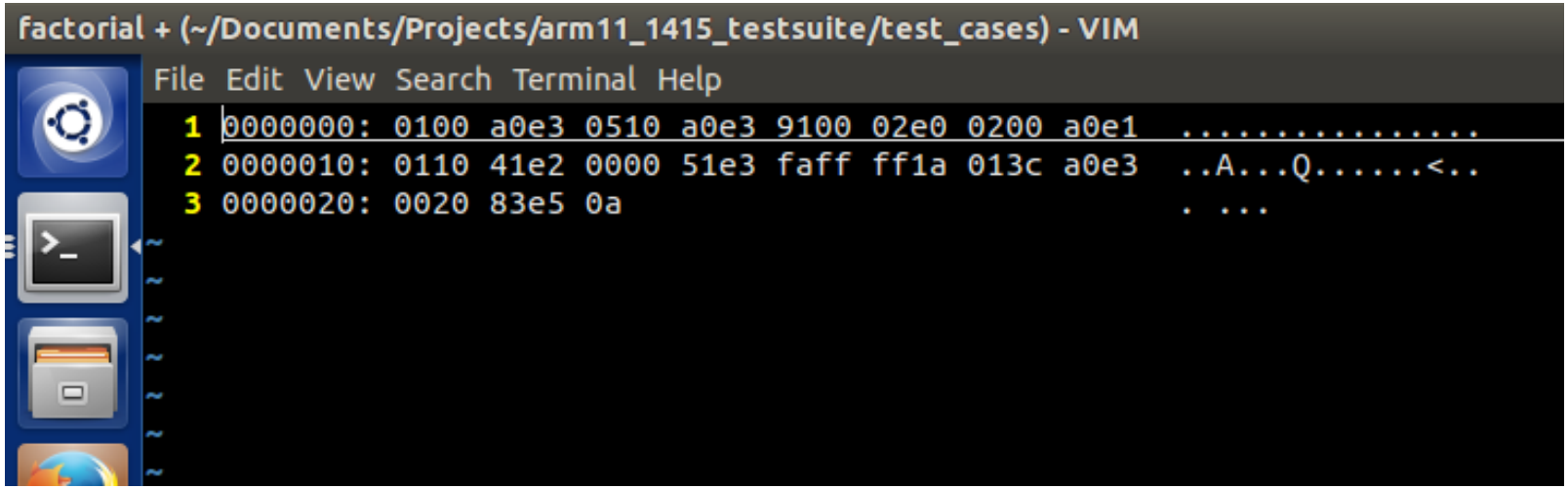
Characteristics of our assembler include:

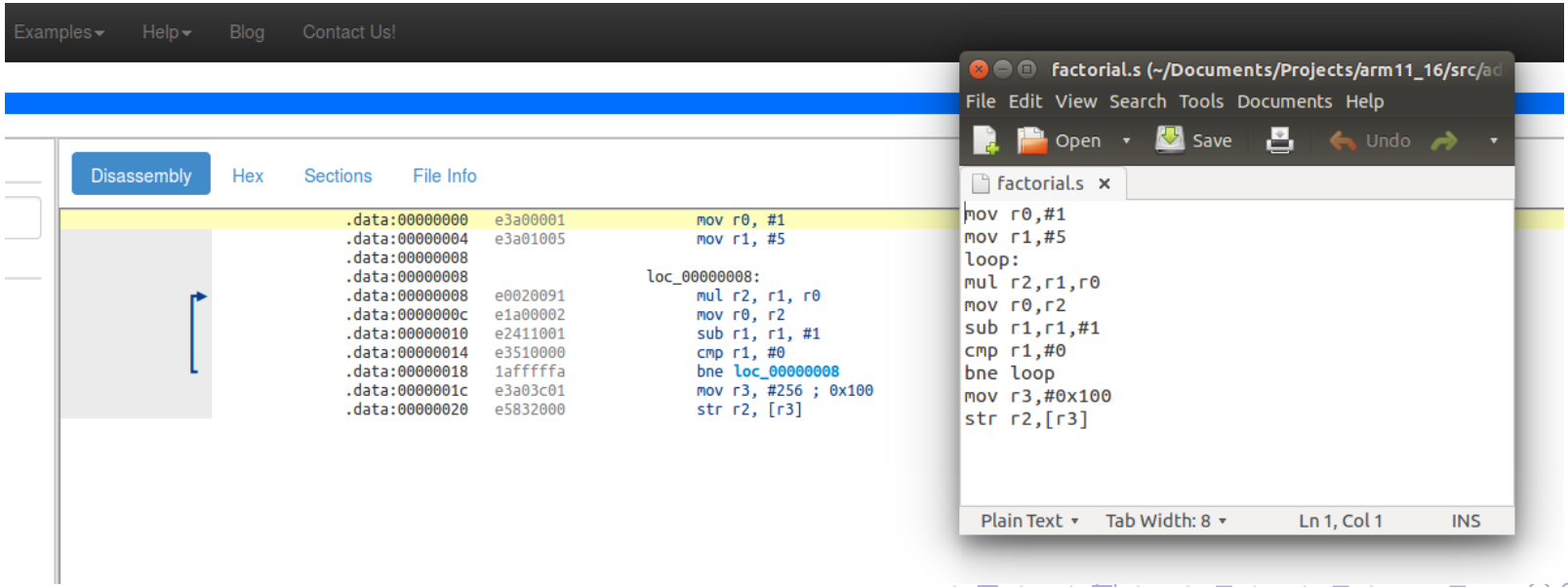
- 2 pass assembler method
- Polymorphic AVL Tree ADT dictionary
- Using the Dictionary to map functions to Opcodes
- Each Function decodes the instruction string themselves

Structure


struct treeNode
+ key: Key *
+ value: Value *
+ left: struct treeNode *
+ right: struct treeNode *
Assemble.h
+ opcode_function: DICTIONARY *
+ alias_register: DICTIONARY *
+ label_address: DICTIONARY *
+ file_line: uint32_t
+ file_length: uint32_t
+ main(void): int
struct storeEncode
+ encFunc: Encode
struct Dictionary
+ tree: struct treeNode*
Dictionary.h
+ Key: void
+ Value: void
+ treeNode: struct treeNode
+ DICTIONARY: struct Dictionary
+ createDictionary(void) : DICTIONARY *
+ isEmpty(int) : int
+ putElem(DICTIONARY, Key, Value) : int
+ getElem(DICTIONARY,Key) : Value *
+ removeElem(DICTIONARY,Key) : int
+ destroyDIctionary(DICTIONARY) : int
Encode.h
+ (*Encode)(Char *): uint32_t *
+ STR_ENC: struct storeEncode
+ DICTIONARY: code_binaryCode
+ dataProcessing1(char *): uint32_t *
+ dataProcessing2(char *): uint32_t *
+ dataProcessing3(char *): uint32_t *
+ multiply(char *): uint32_t *
+ multiplyAccum(char *): uint32_t *
+ sdt_ldr(char *): uint32_t *
+ sdt_str(char *): uint32_t *
+ branch(char *): uint32_t *
+ spec_andeq(char *): uint32_t *
+ spec_lsl(char *): uint32_t *

Outcome - Disassembler





Problems and Challenges

String Tokenising

Using **strtok()** and **sscanf()** proved to be very limiting

Dictionary ADT

Development of the AVL tree was time consuming and took a long time before it was bug free

Memory Allocation

Many mistakes and errors ended up being A problem with the use of **malloc()** and **free()**.

Code Reuse

We definitely could have made a better attempt to reuse code from emulator

Improvements

- Creating our own string tokeniser which is more flexible then the implementation of **strtok()**.
- Spending a bit more time optimising and improving the quality of the code.
- Reducing redundant code
- Rethinking the use of pointers

Flashing GPIO

Figure : Our Flashing Gpio Led

FlappyPi

Figure : The Bird Is The Word.

Extending the Assembler

To make it possible for us to use our own Assembler to process the assemble source code, we have had to do the following:

Stack (block data transfer)
: Enabling the use of **stm** and **ldm** in the assembler, so that the stack can be utilised.

Opcodes Suffixes
: Supporting the addition of condition codes appended to existing opcodes.

Aliasing
: Supporting the aliasing of registers and commands for clearer use code.

Commenting
: Supporting commenting for clearer programs.

Multi-File Programs
: Enabling the use of larger programs.

Problems and Challenges - Assembler extension

Aliasing

Memory corruption due to lack of compatibility with our dictionary

push/pop

Difficult to debug as the error was very hard to find

Problems and Challenges - FlappyPi

Lack of resources

There is a small number of helpful articles online covering assembly, As a result it was difficult to implement. However, the Cambridge bakingPi tutorial was very helpful for the screen output

Tedious Flappy bird sprite was 1200+ lines of assembly which was very repetitive

Improvements

- Stop the bird form flashing
- Randomly generate terrain
- Include a button and the ability to move the bird
- Collision detection to allow the game to be played

Problems and Solutions

Organisation

- Lack of a Complete Plan from the start
- Redundant code
- Regularly missed personal deadlines
- Distribution of tasks

Communication

- Instant messenger issues
- Lack of Clarity
- Sleeping Pattens

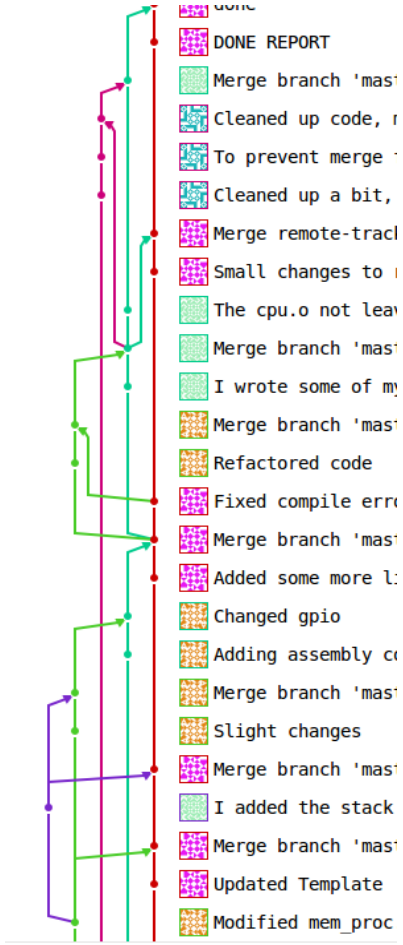
Effectiveness

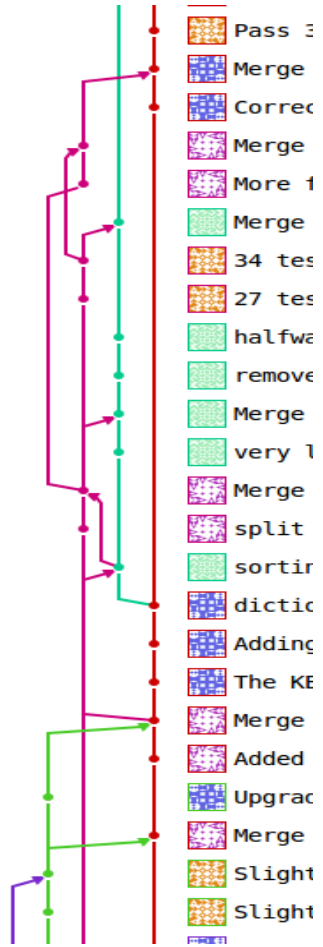
- C knowledge
- Testing and Debugging
- Git

Use of GIT

We did not use Git effectively nor efficiently

- Branching
- Merging
- Excessive use of Master Branch
- Non-compiling/ broken code regularly pushed





Conclusion

- Enjoyable and informative
- Learned new languages of Assembly and C
- Deeper understanding of the CPU
- Low level Graphical Interfaces
- Learnt key aspects of working in a team.