THE UNIVERSITY OF SYDNEY

COMP2017 / COMP9017 Assignment 2

Due: 11:59PM Sunday 11 April 2021 local Sydney time

This assignment is worth 15% of your final assessment

Task Description

In this assignment you will be implementing and performing operations on a simple virtual machine. You will need to emulate this virtual machine to store and reference variables and stack frame contexts before reading a set of pseudo assembly instructions that dictate the operations that should be performed on the stack. Your program will take a single command line argument being the path to the file containing your x2017 assembly code.

Before attempting this assignment it would be a good idea to familiarise yourself with the stack, registers, stack frames, stack pointers, program counters, assembly and machine code. A strong understanding of these concepts is essential to completing this assignment. Section 3.6 and 3.7 of the course textbook provide specific detail to x86_64 architecture, however you can review these as a reference.

In order to complete this assignment at a technical level you should revise your understanding of bitwise operations, file IO, pointers and arrays.

Some implementation details are purposefully left ambiguous; you have the freedom to decide on the specifics yourself. Additionally this description does not define all possible behaviour that can be exhibited by the system; some error cases are not documented. You are expected to gracefully report and handle these errors yourself.

The Architecture

In this assignment you will be emulating an 8 bit architecture. The memory model of this architecture consists of:

- RAM - Contains 2^8 addresses of 1 byte each
- Register Bank - 8 registers of 1 byte each
- Program Code - Memory required to store the program to be executed.

For full marks the total size on disk of your program's binary should not exceed 10kb.

During execution you should not store any information about the state of the machine outside of the RAM and the register bank.

Note: A register stores a single value using a fixed bit width. It the size of a register corresponding to the processor *word size*, in this case 8 bits. Think of them as a primitive variable. Physical processor hardware is constrained, and the number of registers is always fixed. There are registers which serve specific purposes, and those which are general. Please identify these in the description and consider them for your solution. You need not consider special purpose registers, such as floating point, in this assignment.

Program Code: x2017

Our virtual machine will be operating on a home brewed 'x2017' assembly language. You will be provided with binaries in this language to run on your virtual machine. Each operation within 'x2017' contains an operation specifier code (op code) and takes zero, one or two arguments. Arguments are expressed as one of four different types, while operation codes are selected from a table. The arguments of each function precede the opcode and are expressed as follows:

```
([Second Value][Second Type])([First Value][First Type])[Operation Code]
```

A collection of these operations will form a function.

The type is a two bit field and specifies the type of the preceding value.

00 - value: 1 byte long. The value in the preceding 8 bits should be interpreted as a single byte value.

01 - register address: 3 bits long. This address refers to one of the eight fixed registers

10 - stack symbol: 5 bits long. This refers to a particular symbol within the current stack frame.

11 - pointer valued: 5 bits long. This treats the contents of the address referred to by a particular symbol within the current stack frame as a variable. Pointers may reference variables on different stack frames.

The second address and address type field is optional and will not be required for unary operations. The address type specifies whether it is a stack address, a value, a register address or a pointer to another stack address. Stack addresses are 8 bits long, register addresses are 3 bits long and values are 8 bits long and pointer addresses are also 8 bits long.

A stack symbol is a value that is associated with an address on the stack; it is up to you to find a cogent method of allocating stack space to symbols. Symbols only exist within the scope of the current function; should two functions use the same symbol then they are **not** referencing the same region of memory.

While the exact memory layout within the stack frame is open to interpretation, four registers are reserved for special values.

- **0x07** Stores the program counter.

- **0x06** Will not be referenced by the program; this register exists for your personal use.
- **0x05** Will not be referenced by the program; this register exists for your personal use.
- **0x04** Will not be referenced by the program; this register exists for your personal use.

Registers in the range **0x00-0x03** are general purpose registers and may be explicitly referenced by a program.

The opcodes associated with 'x2017' instructions are detailed below. You will need to read each of the op-codes and implement the operation on the memory specified.

Opcodes:

000 - [MOV A B] - Copies the value at some point B in memory to another point A in memory (register or stack). The destination may not be value typed.

001 - [CAL A] - Calls another function the first argument is a single byte (using the VALUE type) containing the label of the calling function.

010 - [RET] - Terminates the current function, this is guaranteed to always exist at the end of each function. There may be more than one RET in a function. If this function is the entry-point, then the program terminates.

011 - [REF A B] - Takes a stack symbol B and stores its corresponding stack address in A.

100 - [ADD A B] - Takes two register addresses and ADDs their values, storing the result in the first listed register.

101 - [PRINT A] - Takes any address type and prints the contents to a new line of standard output as an unsigned integer.

110 - [NOT A] - Takes a register address and performs a bitwise not operation on the value at that address. The result is stored in the same register

111 - [EQU A] - Takes a register address and tests if it equals zero. The value in the register will be set to 1 if it is 0, or 0 if it is not. The result is stored in the same register.

The state of the registers is preserved between CAL and RET operations.

In the event that the execution of this program enters an undefined state; for example if the amount of stack memory required to execute the program exceeds the RAM buffer, then you should print an appropriate error to standard error and return 1 on exiting main.

The value of the program counter register should reference the current opcode. Your program should increment the program counter **before** executing the associated instruction. You may wish to consider what may happen when the program counter is modified during the execution of an instruction.

Binary File Format:

The first few bits of the file are padding bits to ensure that the total number of bits in the file accumulates to a whole number of bytes. The number of padding bits will always be strictly less than one byte.

The file is broken up into a number of functions. Each function is defined with a three bit header dictating the 'label' of the function and the number of arguments, and a five bit tail specifying the number of instructions in the function. The function with the label 0 is the entry point and should be executed first.

```
[Padding bits]
[function label (3 bits)]
    [OPCODE]
    [OPCODE]
    ...
    [RET]
    [Number of instructions (5 bits)]
[function label (3 bits)]
    [OPCODE]
    [OPCODE]
    ...
    [RET]
    [Number of instructions (5 bits)]
```

Example

The following assembly function moves the values 3 and 5 to separate registers before adding them, moving the value to the stack and returning.

Some equivalent C code might look like this:

```
#define BYTE unsigned char // Because all values are 1 byte
void main()
{
    register BYTE reg_0 = 3; // Storing the value 3 at register 1
    register BYTE reg_1 = 5; // Storing the value 5 at register 2

    reg_0 += reg_1; // ADD The two registers, save the value at r1

    BYTE A = reg_0; // Store the value from register 1 at stack symbol A

    return; // Return
}
```

We can consider some equivalent x2017 assembly:

```
FUNC LABEL 0
    MOV REG 0 VAL 3
    MOV REG 1 VAL 5
    ADD REG 0 REG 1
    MOV STK A REG 0
    RET
```

And this assembly can be compiled down to binary:

```
00000 # Padding for the file
000 # Function labeled 0
    00000011|00|000|01|000 # MOV REG 0 VAL 3
    00000101|00|001|01|000 # MOV REG 1 VAL 5
    001|01|000|01|100      # ADD REG 0 REG 1
    000|01|00000|10|000    # MOV STK A REG 0
    010                    # RET
00101 # Number of instructions in the function
```

Or without the comments:

```
0000000000000011000000100000000101000010
1000001010000110000001000001000001000101
```

And finally; hex formatted:

```
\x00\x03\x02\x01B\x82\x86\x04\x10E
```

Your program will take the path to one of these binary files as a command line argument and execute the instructions within.

Milestone

As part of this assignment, you are expected to submit a milestone prior to your final submission. The milestone task will be to create a disassembler for x2017 binaries. That is; your program will take an x2017 binary and print its contents in a human readable form. Given a path to a file containing:

```
\x00\x03\x02\x01\x42\x82\x86\x04\x10\x45
```

You would be expected to print:

```
FUNC LABEL 0
    MOV REG 0 VAL 3
    MOV REG 1 VAL 5
    ADD REG 0 REG 1
    MOV STK A REG 0
    RET
```

You may notice that the symbol information is stripped in the binary; that is that the value ‘A’ never appears. You should print each unique symbol within each stack frame as a single capital letter in order of appearance starting with ‘A’. As there are 2^5 possible symbols and only 26 letters of the alphabet symbols after Z should instead be displayed as lower case letters starting at a.

Indentation in the output of this program should be performed using four spaces.

Compilation and Execution

Your program will be compiled by running the default rule of a make file. Upon compiling your program should produce a single ‘vm_x2017’ binary. Your binary should accept a single argument in the form of the path to a ‘x2017’ binary file to execute.

```
make
./vm_x2017 <x2017_binary>
```

Your milestone disassembler will be compiled and run using the make rule `make objdump_x2017`

```
make objdump_x2017
./objdump_x2017 <x2017_binary>
```

Tests will be compiled and run using two make rules; `make tests` and `make run_tests`.

```
make tests
make run_tests
```

These rules should build any tests you need, then execute each test and report back on your correctness.

Failing to adhere to these conventions will prevent your markers from running your code and tests. In this circumstance you will be awarded a mark of 0 for this assignment.

Marking Criteria

The following is the marking breakdown, each point contributes a portion to the total 15% of the assignment. You will receive a result of zero if your program fails to compile.

Marks are allocated on the basis of:

- Milestone Correctness - 3 - Passing automatic test cases, a number of tests will *not* be released or run until after your final submission.
- Final Correctness - 3 - Passing automatic test cases, a number of tests will *not* be released or run until after your final submission.
- Binary size - 2 - If all test cases are passed, then an additional 2 marks are awarded if the binary does not exceed 10kB. If the test cases are not passed, then these marks are not awarded.
- Solution Discussion - 7 - You will need to answer questions from a COMP2017 teaching staff member regarding your implementation. You will be required to attend a zoom session with COMP2017 teaching staff member after the code submission deadline. A reasonable attempt will need to be made, otherwise you will receive zero for the assessment.

  In this session, you will be asked to explain:

  - How your program counter is affected by the opcodes executed.
  - How your code organises and manages the stack memory for function calls.
  - What are the edge cases you considered for when your program returns 1.
  - Answer further questions.
  - Your code will also be assessed on C coding style conventions (see Ed resources). Clean code will attract the best grade.

Correctness is based on:

- `vm_x2017` - return value of the program
- `vm_x2017` - printed standard output of the `x2017` binary program execution (all PRINT ops)
- `objdump_x2017` - printed standard output of the disassembler

Additionally marks will be deducted on the basis of:

- Compilation - If your submission does not compile you will receive an automatic mark of zero for this assessment.
- Style - 2 - Poor code readability will result in the deduction of up to 2 marks. Your code and test cases should be neatly divided between header and source files in appropriate directories, should be commented, contain meaningful variable names, useful indentation, white space and functions should be used appropriately. Please refer to this course's style guide for more details.
- Tests - 2 - A lack of tests, or a lack of thorough testing will result in the deduction of up to 2 marks. Please provide your test cases along with appropriate scripts to build, run and report the results of your tests. As a number of tests will not be released until after your final submission you are strongly encouraged to test all aspects of your program.
- Graceful Error Handling - 1 - The description above contains a number of undefined behaviours. Your program should gracefully catch each of these error, report, and exit. Should your program crash marks will be deducted. Your error messages should be meaningful but you will not be provided with a standard format.
- Memory Leaks - Code that leaks memory will receive a mark of 0. Savvy readers may note that this task does not necessarily require dynamically allocated memory.

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not follow the assignment description or if your code is unnecessarily or deliberately obfuscated.

Helpful Hints / Where to Start

- Start by reading in the operation code from the above example, you will need to use bitwise operations here and should refer to the relevant lectures and tutorial sheets.

- You do not know the number of padding bits at the start of the file in advance! Consider how you might calculate this.
- Start with the milestone; you can't execute the program until you have parsed it; and parsing the program is a requirement for completing the milestone.
- Spend some time working out how the CAL and RET operations work before you start writing your implementation.
- You will not be tested on the internal state of your stack and as a result you have a degree of flexibility in your implementation. You will need to decide how to manage memory on your stack yourself.
- Invalid inputs may be provided; when you detect such a case you should return 1 and exit with an appropriate error message.
- When writing test cases you may find it useful to write your own ascii to x2017 compiler and bundle it with your test builder.
- You have three registers and an arbitrary amount of stack space set aside for your own use; the success of your approach will depend on how you utilise this memory.

Academic declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.