Dr Mark Doughty,
School of Engineering and Physical Sciences,
University of Lincoln.

C Programming Project 1

A (Simple) Virtual Machine

This describes a really simple approach. You could modify the operators, or the operation of the VM to make it more comprehensive or to handle more sophisticated programs.

We have a set of registers (think of these as just variables) – `R0, R1, R2, R3, R4, R5, ip`, and `sp` (although it is probably easier to have these in an array).

We have a stack data structure implemented as a 256 element array.

The instruction set can be implemented as an enum in C:

```
typedef enum{
    HLT,
    POP,
    LDC,
    CPY,
    ADD,
    SUB,
    PRR
}InstructionSet;
```

We will implement each program as an array, eg:

```
const int program[] = {
    LDC, 5,
    LDC, 3,
    ADD, R0,
    POP, R0,
    PRR, R0,
    HLT
}
```

This test program, just loads two constant values onto the stack, adds the values together and stores the result in R0. It then prints out the contents of R0.

A VM has a program counter – this records where we are in the program. As our program is an array of ints, then the program counter (or 'instruction pointer') is just the index of this array. We store the value of the instruction pointer in the `ip` register.

Dr Mark Doughty,
School of Engineering and Physical Sciences,
University of Lincoln.

Likewise, the VM needs to keep track of the top of the stack. As this is just an array, then this is just an integer as well. It is stored in the stack pointer (`sp`) register.

So, the fetch, decode, execute cycle would look like this for our VM:

FETCH

Get the instruction at the current instruction pointer. The instruction is in two parts: the opcode (what the instruction is) and the operand(s) (what to operate on).

DECODE

To make this easier, we could have each instruction and its arguments as one value. Each part being one byte in width, and using bit masks and bit shifts to 'unpack' the instruction and its arguments.

If not, the decode step is just reading the instruction and arguments from the array.

Opcodes:

LDC # (Push the operand onto the top of the stack. If # is a register, load its contents).

POP # (Remove value from the top of the stack and place in the operand register).

ADD # (Add the values at the top two positions of the stack and place result in the operand register).

SUB # (Subtract the values at the top two positions of the stack and place result in the operand register).

CPY # # (Copy the contents of the first register to the second register).

PRR # (Print the contents of the operand register).

HLT (Stop the execution – like an interrupt).

EXECUTE

We could do this with a switch statement – so the 'execute' function of the VM could just take an instruction as an argument, and then evaluate it with a switch statement:

```
void execute(int instruction){
    switch(instruction){
        case HLT:
            running = false;
            break;
        case LDC:
            //etc
    }
}
```

Have a program loop which runs to fetch each instruction until HLT is reached.

Compiling the VM

Use a simple gcc command:

```
gcc -Wall -o virtualMachine virtualMachine.c
```

Challenges

Implementing the various actions like ADD and SUB - how would we implement more complex operations like multiply and divide?

Using only bitshift operators, implement simple mathematical operations (ADD, SUB, etc).

Our VM doesn't handle branching programs, how could you implement this? Maybe its as easy as setting to `ip` to a particular value?

Can we load the VM program from a file into the array?

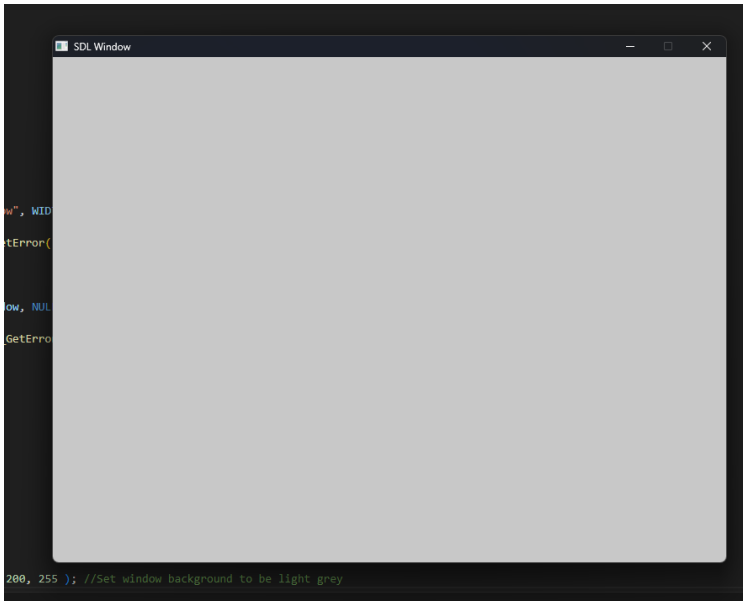
C Programming Project 2

A Game

An SDL3 framework is set up in C. This framework has been set up on my computer – so it relates to 64bit Windows architecture. You may need to reconfigure this framework so that it works with your architecture.

The framework as it stands creates and opens an 800x600 window:

Dr Mark Doughty,
School of Engineering and Physical Sciences,
University of Lincoln.



As well as the SDL initialisation code, there is a rudimentary game loop which accepts SDL event inputs.

To run, I used the following gcc command in the command window while in the cGame directory:

```
gcc -I src/include -L src/lib -o main main.c -lmingw32 -lSDL3
```

This locates the include files and the libraries for SDL, as well as the gcc compiler libraries.