Writing the ASCII Table to the VDU

The following code will eventually overwite the contents of the program, as BL overflows past FF:

```
mov al, 00
mov bl, 00
start:
inc al
mov [bl], al
inc bl
jmp start
end
```

Note that we can't alter RAM values directly, e.g. `inc c0`, so we alter values in registers and move them into memory.

Fixing the Code with the Compare Instruction

```
cmp bl, value
```

The `cmp` instruction will subtract 'value' from the value in bl. If the contents of bl are equal to value, the zero flag will be set, which we can check on using `jz`.

Conditional Jump

We use `jz` in this case. With this command, a jump is made if the `0` flag is set when we reach the `jz` instruction.

So our code now becomes:

```
mov al, 00
begin:
mov bl, c0
start:
inc al
mov [bl], al
inc bl
cmp bl, 00
jz begin
jmp start
end
```

We have now added another label. Lots of labels can make a program difficult to read and understand.

Note also that we could omit the `cmp bl, 00` line if we wanted, as the `inc bl` instruction will set the `0` flag once bl gets to `00`.

Taking Input from the Keyboard

IN Instruction

The following code will take input from port `00` (the keyboard port), and place it automatically into the al register:

```
in 00
```

Input coming from the keyboard will be ASCII data. So, if the user presses the '0' key in reponse to the `in 00` instruction, the value 30 will be placed into the AL register.

This makes it easy to reflect keyboard input back onto the VDU:

```
    mov bl, c0
loop:
    in 00
    mov [bl], al
    inc bl
    jmp loop
end
```

The user's input is stored in al and displayed also on the VDU, but as the ASCII hex value. E.g. if the user enters '1', the stored value is '31'.

The Out Instruction

We can use the following code to output the value in al to the traffic lights on port 01:

```
mov al, fe
out 01
```

Note that the `out` instruction automatically outputs from a1.

There are other outputs available on different ports, e.g. a thermometer, a digit display, etc.

The Stack Pointer

The stack pointer is an area in the memory where you can temporarily put a value or address that you want to have easy access to.

The command to put something on the stack is `push`:

```
push al
```

You can only push the contents of registers onto the stack. To take something from the stack, use `pop`:

```
pop bl
```

`Pop` can also only be used with a register.

Values in the stack will accumulate back towards 0. When you use the `pop` instruction, you get the value closest to 0, which is the value most recently put on the stack that has not already been popped.

This is abbreviated as 'LIFO' – 'Last In, First Out'.

Converting ASCII Values to Integers

Subtract 30 from an ASCII number digit to give the integer value.

Here's an example code to perform an addition based on user input:

```
in 00           # Input first operand
push al         # Move the first operand to the bl register
pop bl          # using the stack (remember you can't do 'mov al, bl')
sub bl, 30      # Convert bl from ASCII
in 00           # Input second operand
sub al, 30      # Convert al from ASCII
add al, bl      # Add operands, leaving result in al
add al, 30      # Convert back to ASCII
mov [c0], al    # Display to VDU
```