CS253 Architecture II: Introduction to Assembly Language (Lab1)

A CPU fetches an instruction from memory, decodes it and then executes it. The CPU then moves to the next instruction in memory and repeats the cycle. The instruction fetched from memory contains an operator. The operator is just a binary number that tells the CPU what to do. Most instructions require some additional information to act on. The extra information is known as the operand. Any instruction can be represented by an

Operator and Operand

On 8086/Pentium processors the value 180 (decimal) is interpreted as an assignment operation that sets the register *ah* equal to the next byte in memory after the operator, the operand. Thus

180, 100

is equivalent to telling the computer to set *ah=100*.

The computer is quite happy to work in terms of numbers; each number represents a different instruction. Programmers would find it almost impossible to remember all the numbers. In practice each instruction has an equivalent text based mnemonic. The list of numbers that describes the program is called the Machine Code. The list of mnemonics that describes the same program is called Assembly Language.

Assembly language is a programming language that is specific to the processor on which it is to run. Today we will look at the 8086 processor. The Pentium can execute all the instructions of the 8086 (the Pentium is backward compatible with previous PC processors such as the 486 or 8086). Unlike other programming languages there is a one-to-one correspondence between the lines of assembly language (that you write) and the machine code (that the CPU interprets).

When used correctly assembly language provides a means of producing extremely fast code. This is possible because you can optimise your code to be specific for the task, rather than using generic libraries. The fact that the code is so specific to the task can result in very small programs being developed and so the code uses very little memory.

The big negative with assembly language is that it is very difficult to create anything but the simplest of programs with it. You wouldn't try to write a database using assembly language, far better to use a higher level language such as C++ or SQL.

So why learn about Assembly Language?

1. It is representative of the code that the CPU actually executes.
2. It is the output *(*.exe)* for compilers of higher level languages.
3. It is extremely fast (useful when trying to control fast hardware or when doing serious number crunching).
4. It uses only small amounts of memory (useful for memory resident programs such as device drivers).
5. An understanding of Assembly Language allows you to use a higher level language in a way that is likely to optimise speed and memory usage.
6. Sometimes an executable file may need to be reverse engineered to see how it works without sight of the original source code.

The basics of x86 assembly language programming: There are four *general-purpose registers* in the 8086 CPU; they are AX, BX, CX and DX. When you are creating short pieces of code you can treat them as variables. Each register contains a sixteen-bit number that can store a value in the range 0 to 65535 (or -32768 to +32767).

Each general-purpose register is split into two eight-bit registers. Thus AX is made of AH (H for High Byte) and AL (L for Low byte). Each of the smaller registers can store a number in the range 0 to 255. The registers are AH, AL, BH, BL, CH, CL, DH and DL.

The registers that you can use in your program will have to come from the following list, AX, AH, AL, BX, BH, BL, CX, CH, CL, DX, DH and DL. You will probably treat these registers as variables in the early stages of learning to program. Remember though that a variable would be a labelled piece of data, probably (although not definitely) contained in the data segment.

The assignment instruction in 8086 Assembly Language is *mov* (short for move). On other processors the instruction is *ld* (short for load).

Thus

Immediate Addressing: *mov ah,10* is equivalent to *ah=10;* in C.
mov bx, OFFSET msg1 is equivalent to *bx=&[msg1]*

Register Addressing: *mov ax,bx is equivalent to ax=bx in C*

Register Indirect Addressing: *mov dx,[bx]* is similar to *dx=array[bx];* in C

Direct Addressing: *mov bx,Count* is equivalent to *bx=[Count]* in C

There are very many arithmetic instructions available, the basic ones are add, subtract, increment (add 1) and decrement (subtract 1).

inc cx is equivalent to *cx++;*
dec cx is equivalent to *cx--;*
add cx,bx is equivalent to *cx=cx+bx;*
sub al,10 is equivalent to *al=al-10;*

After every instruction is executed the CPU modifies a special flag register to record the effect of the instruction. The flag register can be used to check if a result is positive, negative, equal to zero, even-parity etc.

In some cases, you may wish to check that a register (e.g. dl) is equal to some value (the character code for 'q') or another register (e.g. ah). You could subtract the character code for 'q' from dl and check the flag to see if the result is zero. If zero, then dl equalled 'q'. The problem is that you have modified and lost the value in dl.

cmp dl,'q' subtracts character code for 'q' from dl and modifies flags but does not change the value in dl. '*cmp*' is short for compare.

Branching is achieved using jumps. The unconditional jump instruction is *jmp*.

jz is short for jump if zero flag set,
jnz is short for jump if not zero,
jc is short for jump if carry flag set,
jnc is short for jump if carry flag not set.

You can jump to a location identified by a label. A string that is terminated by a colon (:) identifies a label.

Thus the code fragment shown below compares the values in ax and bx and identifies which is the greater.

```
        …
        cmp ax,bx
        jc   skip
        (End up here if AX>=BX)
        jmp  over

skip:(End up here if AX<BX)
over:…
```

The 8086 Processor uses a segmented Architecture. This means that CPU can separate the use of memory into Code, Data and Stack. Memory used to store the executable code is stored in the Code Segment. Memory used to store data (variables, arrays, strings etc) is store in the Data segment. The Stack has its own segment.

Interrupts provide a straightforward way of interfacing with the hardware. To print a character on the screen you set ah = 2, dl = character-code and call interrupt 0x21. An interrupt is similar to a function call. The interrupt uses a vector table to look up the address of the function to be executed. Thus *int 0x21* runs the subroutine pointed to by the 33rd entry in the vector table (21hex=33decimal). The interrupt service routines are common to all PC and form basis of the DOS operating system.

To read a character from the keyboard set ah=8, call interrupt 0x21 and al will hold the value of the key that was pressed.

Note: a typical instruction can execute in four clock-cycles. A PC running at 1GHz can execute about 250 million lines of MASM per second; that's fast!

Before we start:

Rather than say "it doesn't work" or "I can't do it", think about the problem and ask demonstrator specific questions about the task or the programming language or the bug/fault that is troubling you. Assembly Language is very different and takes a while to get use to.

Warning: PDF files can change quote " characters so you may need to retype the code to avoid rogue characters appearing in your listing (cut and paste is risky).

Part1: Building a MASM program using the command line

Enter the program to print 'Hello world' in to notepad or notepad++, this is shown below,

```
       .MODEL    medium
       .STACK
       .DATA
msg1   BYTE    "Hello world$"
       .CODE
       .STARTUP
       mov bx,OFFSET msg1
back:  mov dx,[bx]   ;dl=letters
       cmp dl,'$'
       jz  done
       mov ah,02h
       int 021h
       inc bx
       jmp back
done:  nop
       .EXIT
END
```

Save the file to c:\temp or your X:\ drive make sure the file has the extension *.asm* (*.txt* won't work)

You will now need to compile the program and execute it. To do this on a Windows 10 machine you will need to use DOSBox which is a DOS (Disk Operating System) emulator.

From the Windows start menu click through to the program files and then click on the DOSBox-0.74. (or search for DOS... using the Windows search (or click on the # at the top of the program list))

You will then need to mount two disk drives (connects emulated drives with real drives on the PC),

The "c" drive in the DOS emulator will access the MASM executable code, including the compiler and linker.

The "x" drive in the DOS emulator will access your X: drive containing your code (or c:\temp your choice).

Enter (from the command line)

mount c c:\programs\masm\bin

and

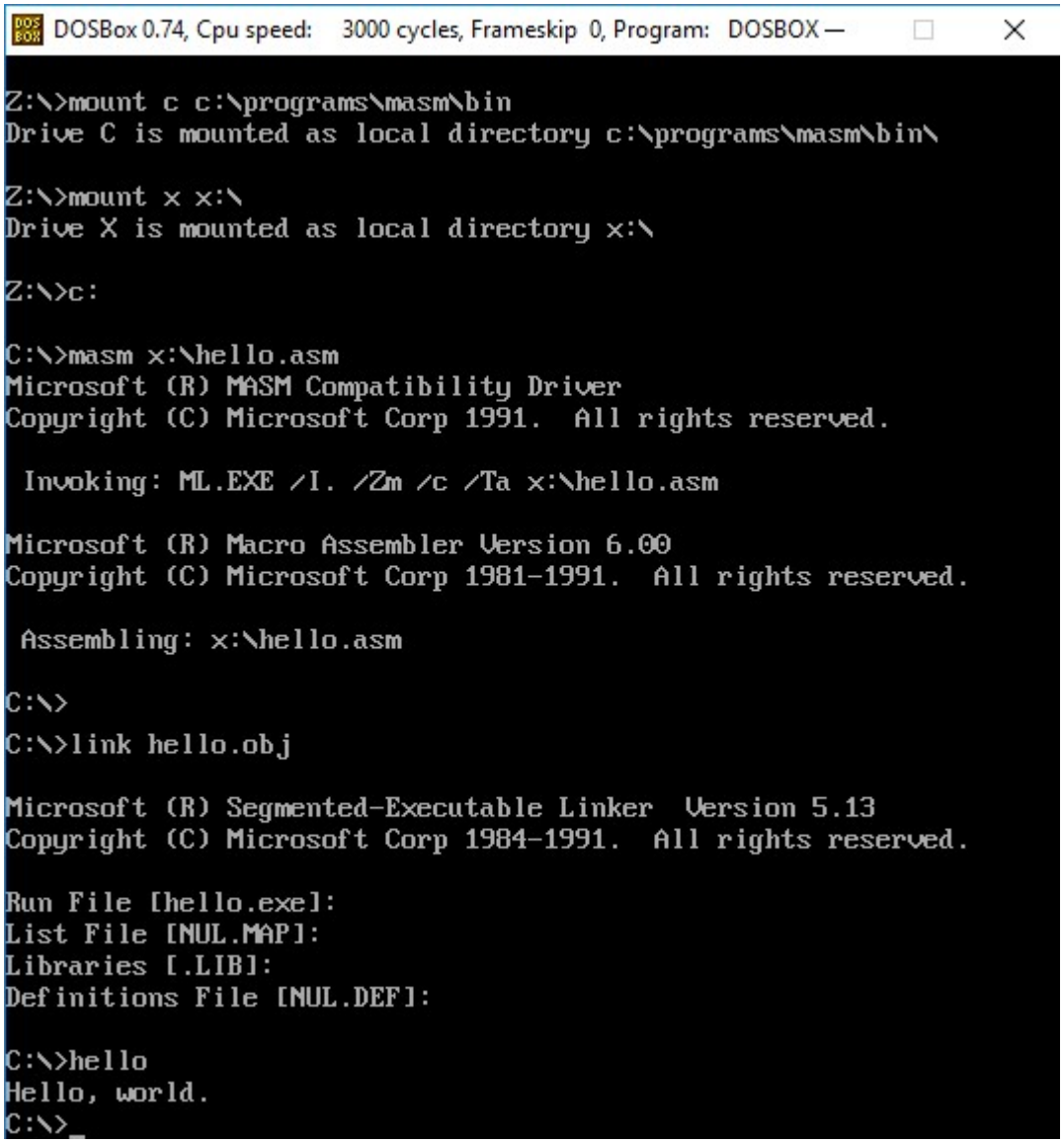
mount x x:\ or mount x c:\temp or mount x x:\cs253\lab1

from the command line

Create your code on the ***X:*** drive, change folder to the **c** drive (enter **c: <*return*>**) and then compile, using

masm x:\hello.asm <return>.

To move up a directory use ***cd..*** <return> and to change drive use ***c:*** <return> or ***x:*** <return>.



Once you have successfully compiled the code you will need to link it using

link hello.obj <return>

Just press <ret> in response to all the questions.

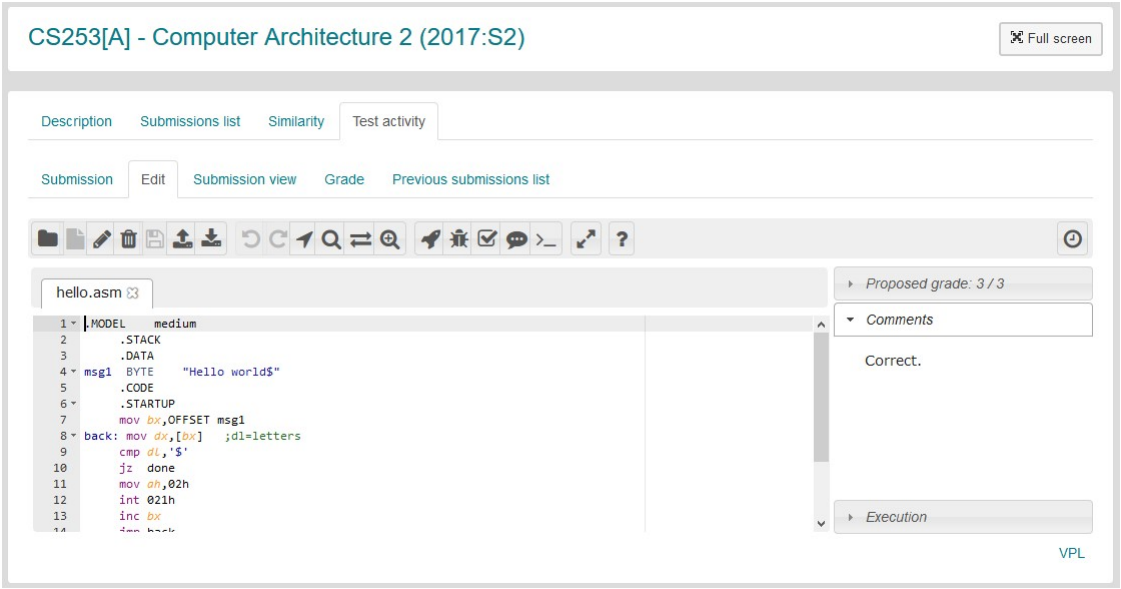
Finally: to run the program just type the name of the exe and press return, so on the ***c:*** drive enter

hello and press ***<return>***

Once you see ***"Hello world"*** on the screen you can proceed to the next stage, well done!

Part 2: Using VPL (Virtual Programming Laboratory) to do the same within the Moodle environment. To assist with recording you progress we have installed MASM on a server that Moodle can connect to using an interface called VPL (Virtual Programming Laboratory). This allows you to edit, compile and run your code through a web interface. Importantly you can submit the code for automated testing and get instantaneous feedback on its functionality (and the grade). This is its second trial with a class so it will be interesting to see how it works.

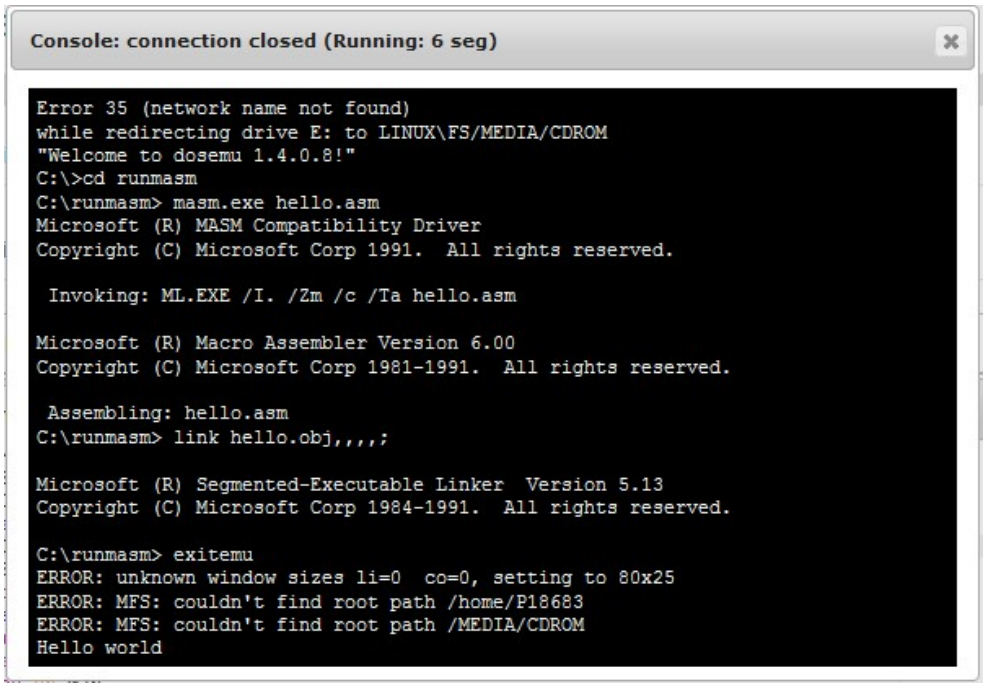
Click on "Part 2: VPL (Virtual Programming Lab) Hello world (3 marks)" on the CS253 Moodle page, select Test Activity and then Edit…



Enter your "Hello world" program, it must be called *hello.asm* and click on "Save"

Press "run" (the rocket icon) when you are happy with the code, don't worry if there is an error the compiler will report it and you can re-edit and try again.

A console window will appear, check that the code has run and compiled correctly, look at what the MASM compiler reports, the last line printed is the output of your program when run.



Finally, to submit the code for test click on "Evaluate" (tick box icon) and see the grade it returns. If you don't get full marks you can retry.

Part 3: Return to building code within the DOSBox emulator and modify the "Hello world" program so that it prints your name, ten times on successive lines.

Hint: the following instructions and code snippet may help

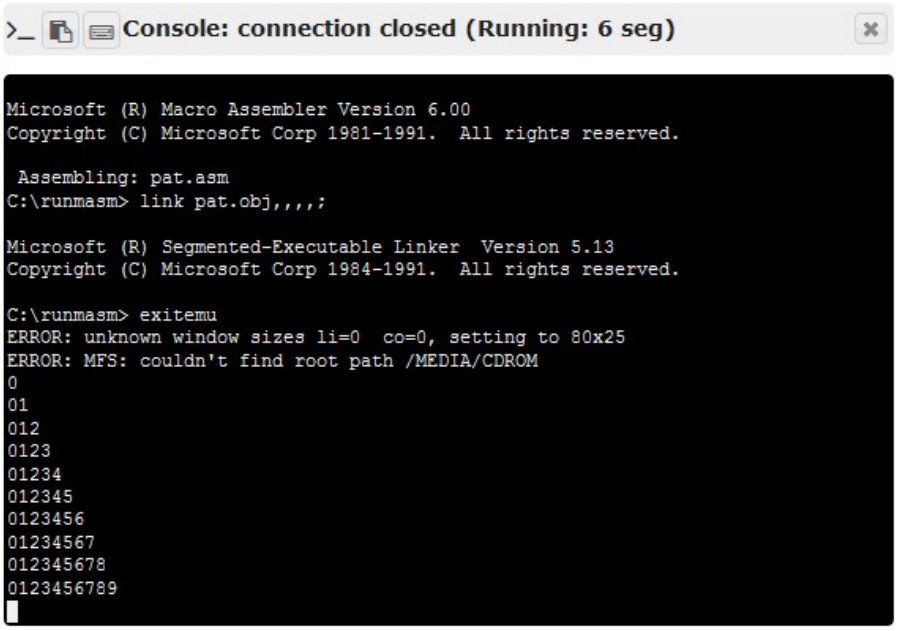
```
msg1        BYTE    "Hello world",13,10,"$"

            mov cx,10

label:      ; existing code to print hello world

            loop  label  ; decrease cx and jump if not zero
```

Note: Character codes 13 and 10 represent carriage return and line feed, used to move cursor to the start of the next line. On a DOS machine you should always do 13 (CR) then 10 (LF).

Submit the assembly language listing file for you program to print your name ten times, e.g. ***hello.lst***

Part 4: Take the following program and extend its behaviour to create the following run time display. This program is described in the first few slides of lecture3. The program must be called *pat.asm*. Submit and evaluate the working version using VPL.

```
              .MODEL MEDIUM
              .STACK
              .DATA
              .CODE
              .STARTUP
              mov bl,4

nextline:     mov dl,'0' ; '0' is ASCII 48

nextchar:     mov ah,02h ; print ASCII char in dl
              int 021h
              inc dl;

              cmp dl,'4' ; '4' is ASCII 53
              jnz nextchar

              push dx    ; save value of dl and dh on stack
              mov ah,02h ; print ASCII char in dl
              mov dl,13  ; carriage return (move to start of line)
              int 021h
              mov dl,10  ; line feed (next line)
              int 021h
              pop dx     ; restore value in dl (and dh)

              dec bl
              jnz nextline
              .EXIT
END
```



Part 5: Complete the Quiz.

C. Markham Feb 2018