1) Segments take the memory and split it up into groups that have common attributes. The segments help you to control different areas of the code. We can protect the memory by splitting it up into sections where some sections have readable/writable/executable rights in them to protect certain areas from actions we don't want to happen in that memory. Segments allow us to have relative addressing. If you have a pointer to each segment, then you can have relative addressing to that segment.

2) The three basic modes of operand addressing are register, immediate, and memory.

In register operand addressing, the operand lives inside a register. An advantage to this approach is that there is no need for references to memory because the value to be operated on is present in the register. Since it does not involve memory, it provides the fastest processing of data. A disadvantage is that registers have a limited amount of available address space, thus there is a limit to the size of a value that can be stored. There are different sized registers for different machines and situations.

In immediate operand addressing, the operand value is found in the program. This can have the advantage of needing no reference to memory outside of a fetch to get the operand from the instruction. In other words, all the data is retrieved with the completion of an operand fetch. A disadvantage is that the size is restricted to the size of the address field the operand resides in. The range of constants is restricted by the address field in this mode such as 8, 16, or 32.

In memory operand addressing, the operand is in the memory. An advantage is having a single reference needed to access data. From what I can see there are multiple kinds of memory operand addressing like direct and indirect, but in general the size of what you are storing seems to be a bit more flexible in this mode as opposed to being restricted by register space and the address field in the other two methods. A disadvantage of this method is that it tends to be one of the slower modes because the correct spot in memory must be located. The data you want to use must be found in memory which is slower than register mode which holds that value in a register.

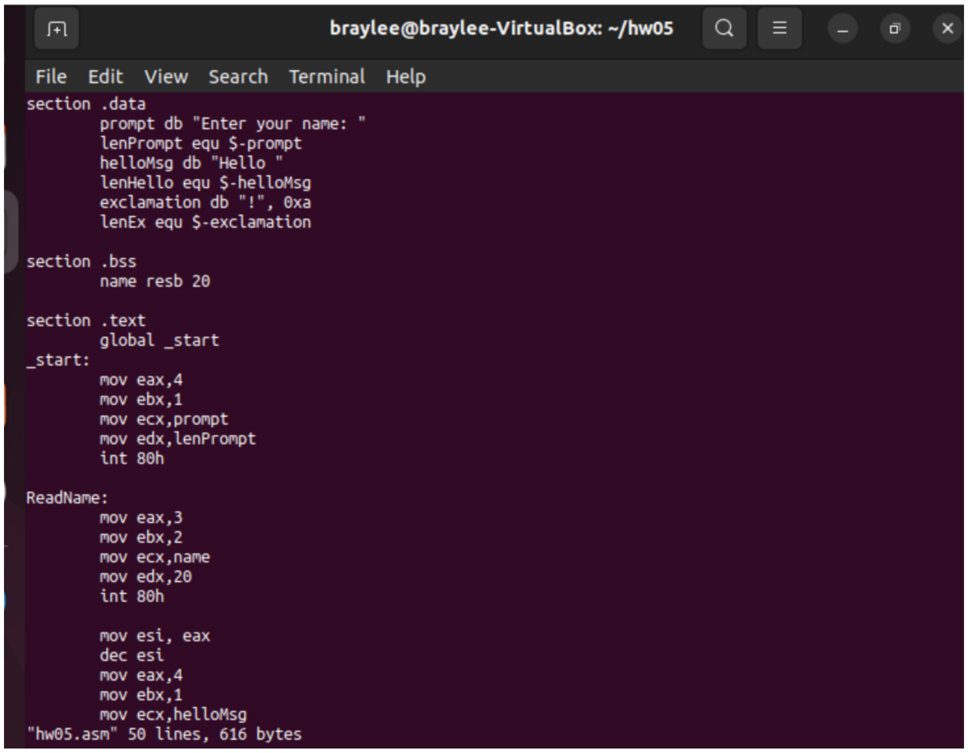
3) The components that can be used to generate an effective address are the BaseReg, IndexReg, scaling and displacement. The BaseReg is the portion that is always used in addressing while the other three are options for more detailed addressing. Both the BaseReg and the IndexReg are general purpose registers. The displacement value is a constant offset that is encoded in an instruction. Scaling is how we deal with things like moving forward an index if you set the scaling to a size of an array element.

4)

These are the screenshots of the assembly code. It didn't quite fit on one screen. The red mark on the second image is the first line that gets cut off from the first screenshot.

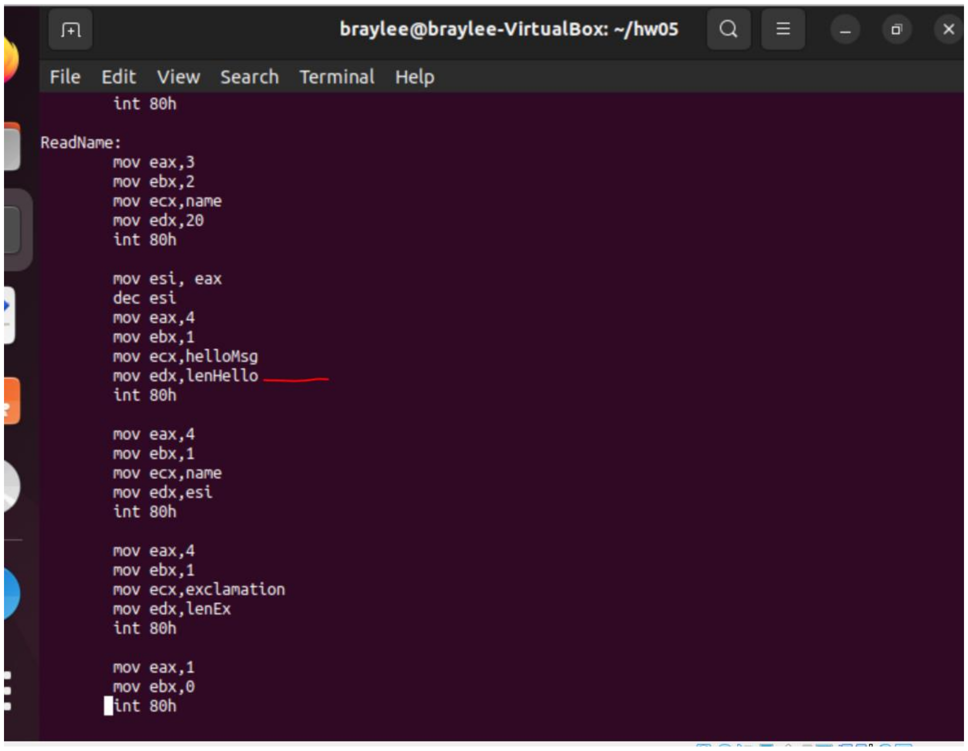
A few notes:

I limited the name to 20 bytes in this code. I used the register esi to hold the length of the name input by the user. The esi register is decremented after it is assigned the length of the input to remove the newline character from the user input.



File Edit View Search Terminal Help

```
section .data
        prompt db "Enter your name: "
        lenPrompt equ $-prompt
        helloMsg db "Hello "
        lenHello equ $-helloMsg
        exclamation db "!", 0xa
        lenEx equ $-exclamation

section .bss
        name resb 20

section .text
        global _start
_start:
        mov eax,4
        mov ebx,1
        mov ecx,prompt
        mov edx,lenPrompt
        int 80h

ReadName:
        mov eax,3
        mov ebx,2
        mov ecx,name
        mov edx,20
        int 80h

        mov esi, eax
        dec esi
        mov eax,4
        mov ebx,1
        mov ecx,helloMsg
"hw05.asm" 50 lines, 616 bytes
```



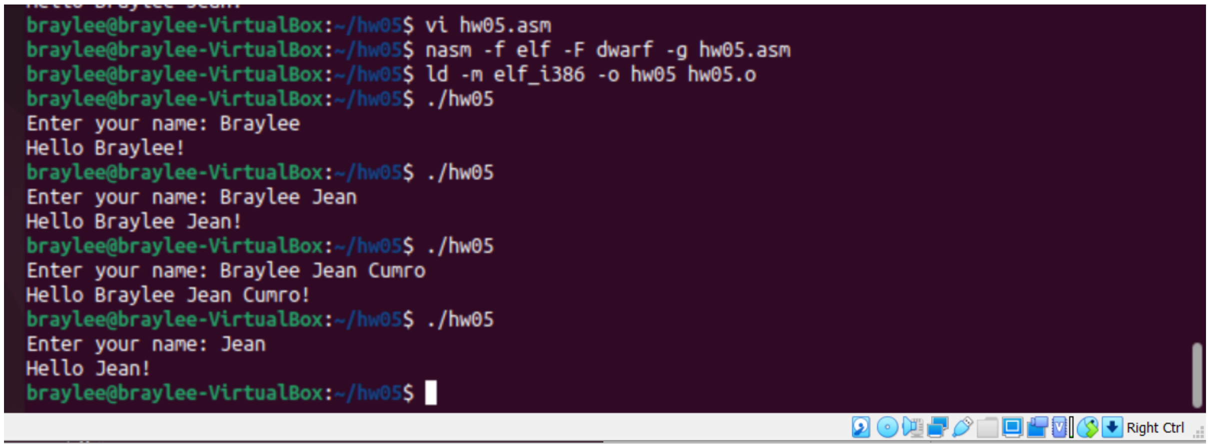
File Edit View Search Terminal Help

```
        int 80h

ReadName:
        mov eax,3
        mov ebx,2
        mov ecx,name
        mov edx,20
        int 80h

        mov esi, eax
        dec esi
        mov eax,4
        mov ebx,1
        mov ecx,helloMsg
        mov edx,lenHello
        int 80h

        mov eax,4
        mov ebx,1
        mov ecx,name
        mov edx,esi
        int 80h

        mov eax,4
        mov ebx,1
        mov ecx,exclamation
        mov edx,lenEx
        int 80h

        mov eax,1
        mov ebx,0
        int 80h
```

The following screenshot is my code running at the command line.

The first line here was me opening it in vi to get the screenshots of the code. Then I made object code with nasm. The next line made the executable hw05. Then I called the executable with ./hw05.

I showed four inputs of variable length here just to show it working with different inputs.



```
braylee@braylee-VirtualBox:~/hw05$ vi hw05.asm
braylee@braylee-VirtualBox:~/hw05$ nasm -f elf -F dwarf -g hw05.asm
braylee@braylee-VirtualBox:~/hw05$ ld -m elf_i386 -o hw05 hw05.o
braylee@braylee-VirtualBox:~/hw05$ ./hw05
Enter your name: Braylee
Hello Braylee!
braylee@braylee-VirtualBox:~/hw05$ ./hw05
Enter your name: Braylee Jean
Hello Braylee Jean!
braylee@braylee-VirtualBox:~/hw05$ ./hw05
Enter your name: Braylee Jean Cumro
Hello Braylee Jean Cumro!
braylee@braylee-VirtualBox:~/hw05$ ./hw05
Enter your name: Jean
Hello Jean!
braylee@braylee-VirtualBox:~/hw05$ █
```