Lab-1 : CHUẨN BỊ MÔI TRƯỜNG THỰC HÀNH

1. **Chuẩn đầu ra :** Sau bài này, người học có thể :
   - ✓ Cài đặt phần mềm máy ảo VMWARE hoặc VIRTUALBOX
   - ✓ Cài đặt hệ điều hành UBUNTU 16-04 64 bit trên máy ảo.
   - ✓ Cài đặt trình hợp dịch NASM trên ubuntu
   - ✓
2. **Chuẩn bị :** Đọc trước phần lý thuyết về cấu trúc của chương trình hợp ngữ.
3. **Phương tiện :**
   - ✓ Máy vi tính.
   - ✓ Chương trình VMWARE 14 và file ISO ubuntu 16-04 64 bit.
4. **Thời lượng : 4 tiết**
5. **Giới thiệu về hợp ngữ trên Linux**

   a. Sự khác nhau chính giữa hợp ngữ trên DOS và hợp ngữ trên Linux

   In DOS assembly, most things get done with the DOS services interrupt int 21h, and the BIOS service interrupts like int 10h and int 16h. In Linux, all these functions are handled by the kernel. Everything gets done with "kernel system calls", and you call the kernel with int 80h. One of the wonderful things about Linux system calls are that there are fewer of them (about 190) than DOS, but they are far more practical (you don't have obsolete crap like functions that load casette BASIC and things left over from DOS 1.0). Linux system calls create files, handle processes and other such useful stuff - no strings attached

   Linux is a true, 32-bit protected mode operating system, so this enables us to do real, up-to-date 32-bit assembly. This 32-bit code runs in the *flat* memory model, which basically means you don't have to worry about segments at all. This makes life a lot easier, because you never need to use a segment override or modify any segment register, and every address is 32 bits long and contains only an offset part. (If this is just a lot of waffling to you, don't worry, just know that it's good and will simplify things for you.)

   In 32-bit assembly, you use the extended 32-bit registers EAX, EBX, ECX and so on instead of the normal 16-bit registers AX, BX, CX etc.

   DOS is dead. It's 16-bit. It's obsolete. The only people that still write DOS assembly are crazy old hackers that are too attached to their 386s to throw them away. Linux assembly has practical applications (parts of the OS are written in assembler, hardware drivers are often coded in assembler)

   b. NASM package

   - NASM package available as source or as executables
   - Typical : /usr/bin/nasm and /usr/bin/ndisasm
   - Assembly :
     - Linux requires elf format for object files

- Elf = Executable and Linking Format
- Typical header size 330h bytes
-

- To assembly type command
  - nasm –f elf [-o outputfile] filename
- Linking
  - Object file can be linked with gcc
  - gcc [-options] filename.o [otherfile.o]
  - or using ld command
- Disassembly
  - View executable as 32-bit assembly code
  - ndisasm –e 330h –b 32 a.out

c. gcc stages

d.

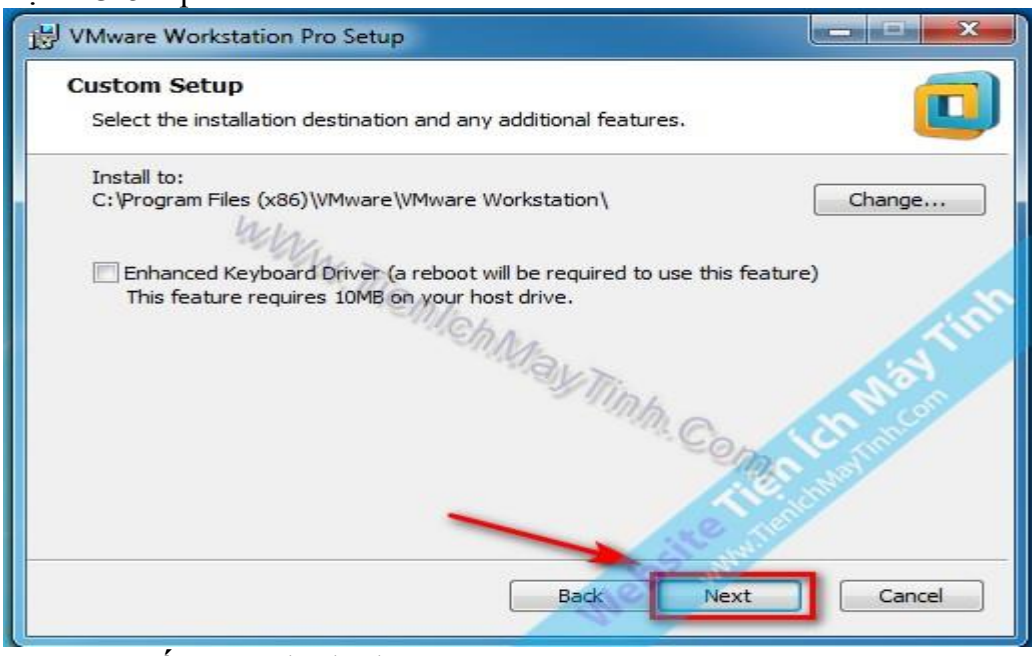
6. Nội dung thực hành

6.1. Cài phần mềm máy ảo VMWARE

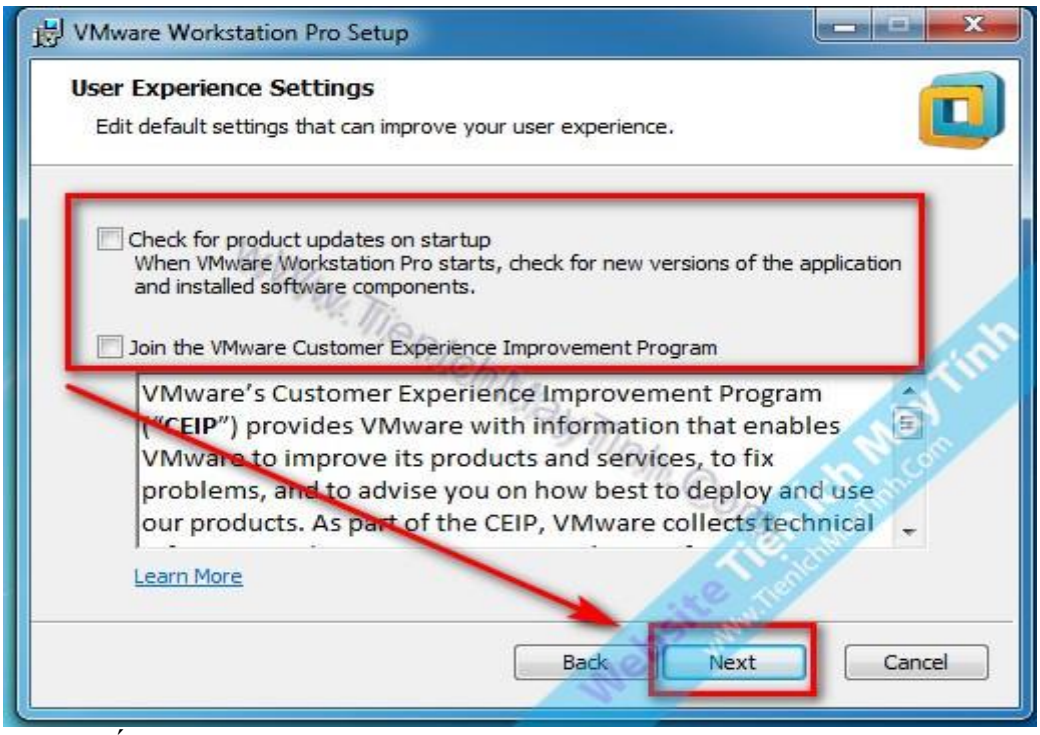
- Giải nén và chạy file .**exe** để bắt đầu cài đặt. Bạn chọn **Next** ở bước này.

- Bạn **tick** vào ô để đồng ý với điều khoản của Vmware và chọn **Next**.

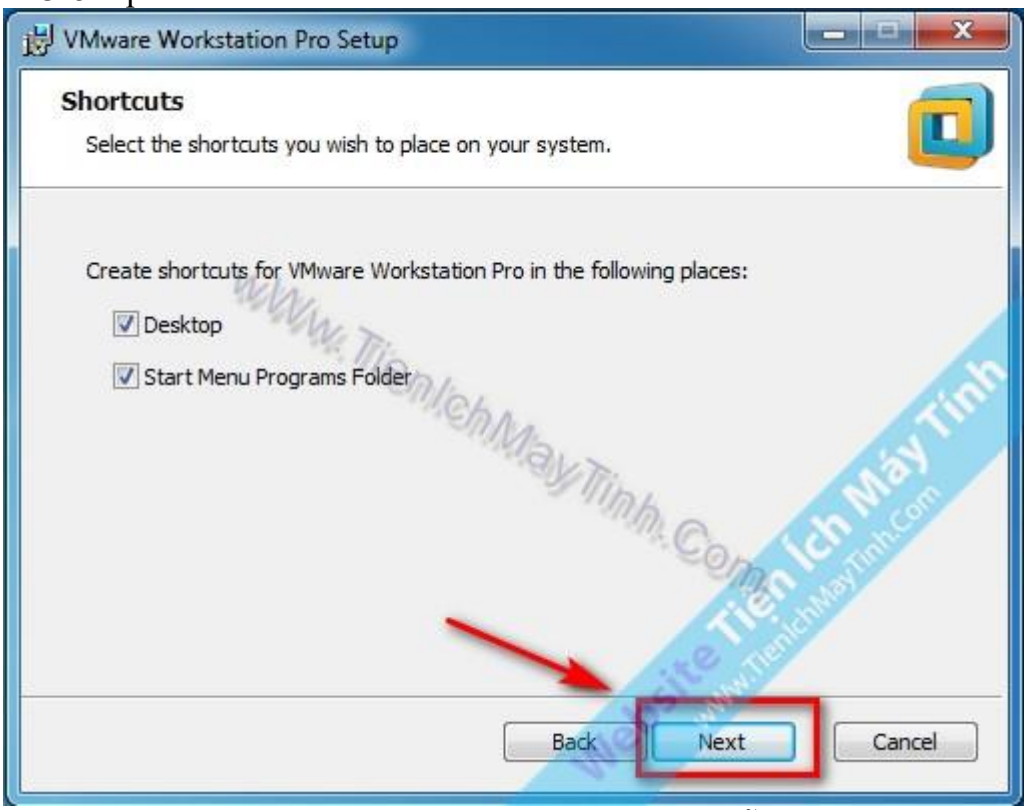
- Bạn **Next** tiếp.



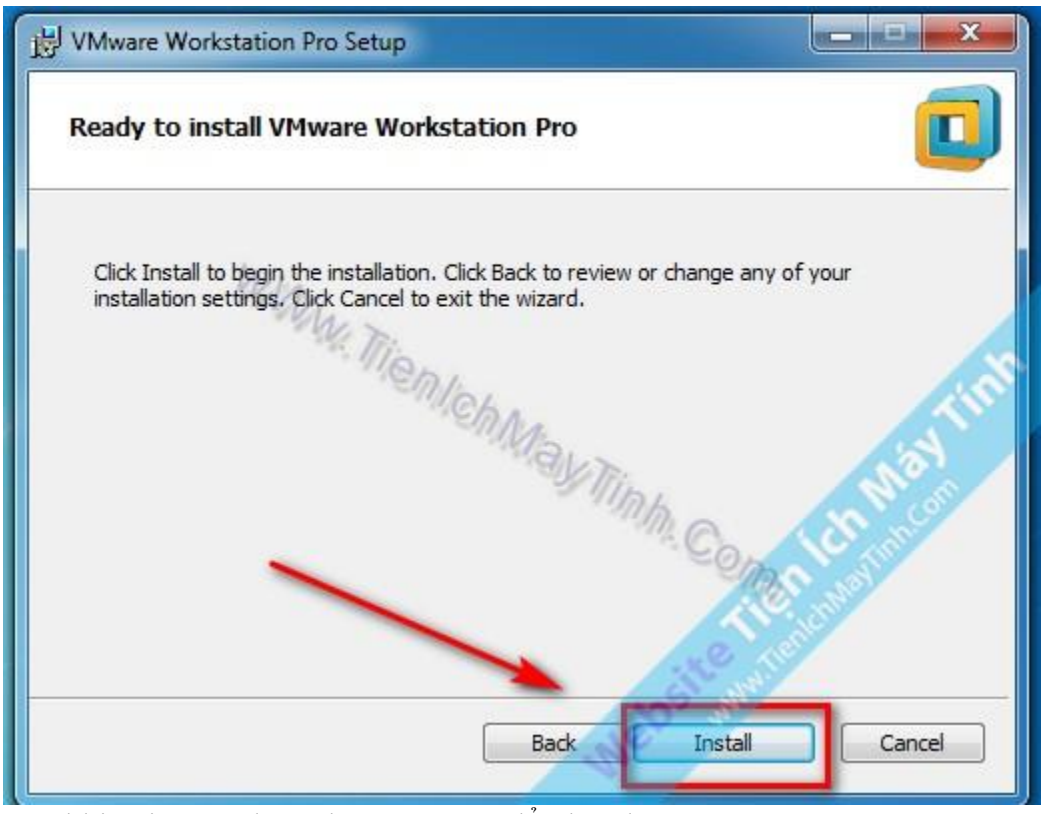
- Bạn **bỏ 2 dấu tick** như hình và **Next**.



- Bạn **next** tiếp.

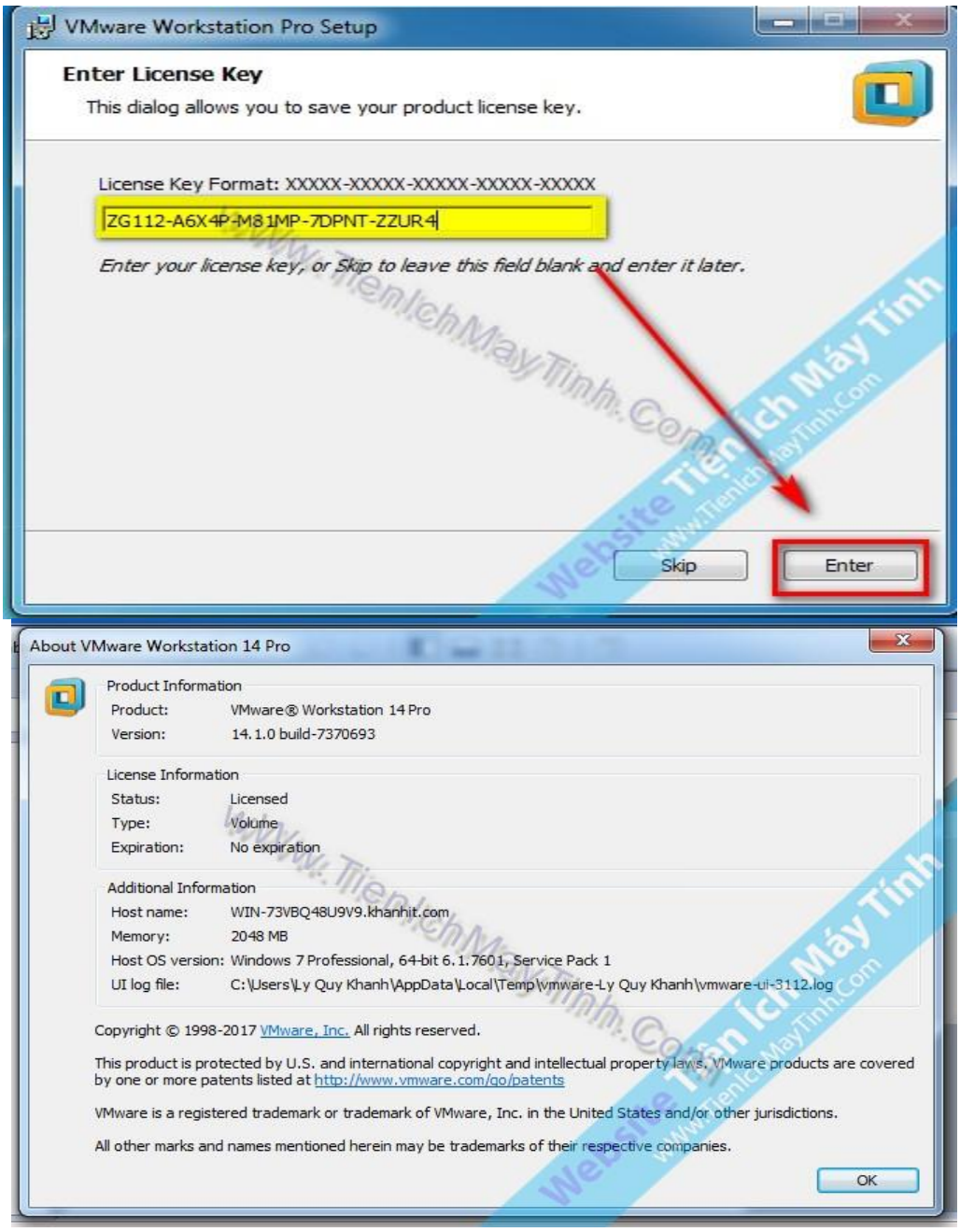


- Bạn chọn **Install**. Và đợi một lúc cho quá trình cài đặt diễn ra.



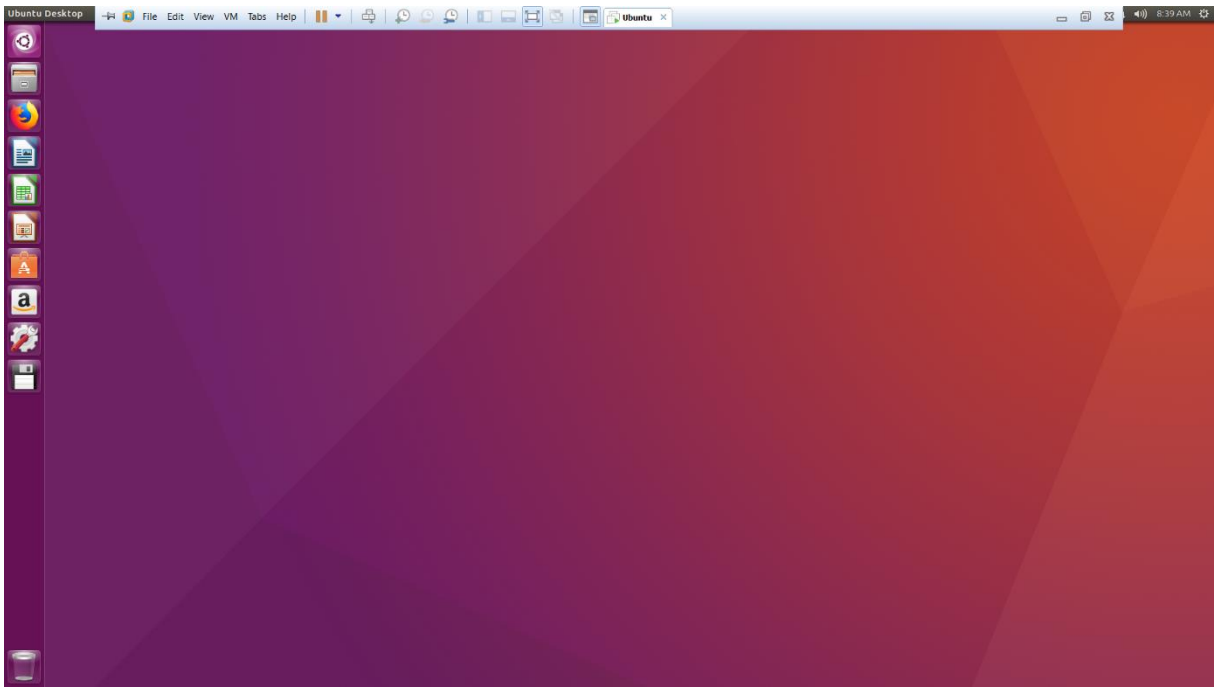
- Sau khi cài xong, bạn chọn **License** để nhập key.

- Copy 1 key ở trên **paste** vào và chọn **Enter**.



6.2. Tạo máy ảo và cài hệ điều hành ubuntu 16-04

Sau khi cài xong, khởi động, màn hình xem như bên dưới



6.3. Cài trình hợp dịch NASM

As we have seen we will be using NASM (Assembler), Linker (ld), GDB (Debugger), gcc (compiler), objdump (Object dump) and additionally Gvim/Vim (code editor). Out of this, we need to install just NASM and/or Gvim, as other tools will be available by default.

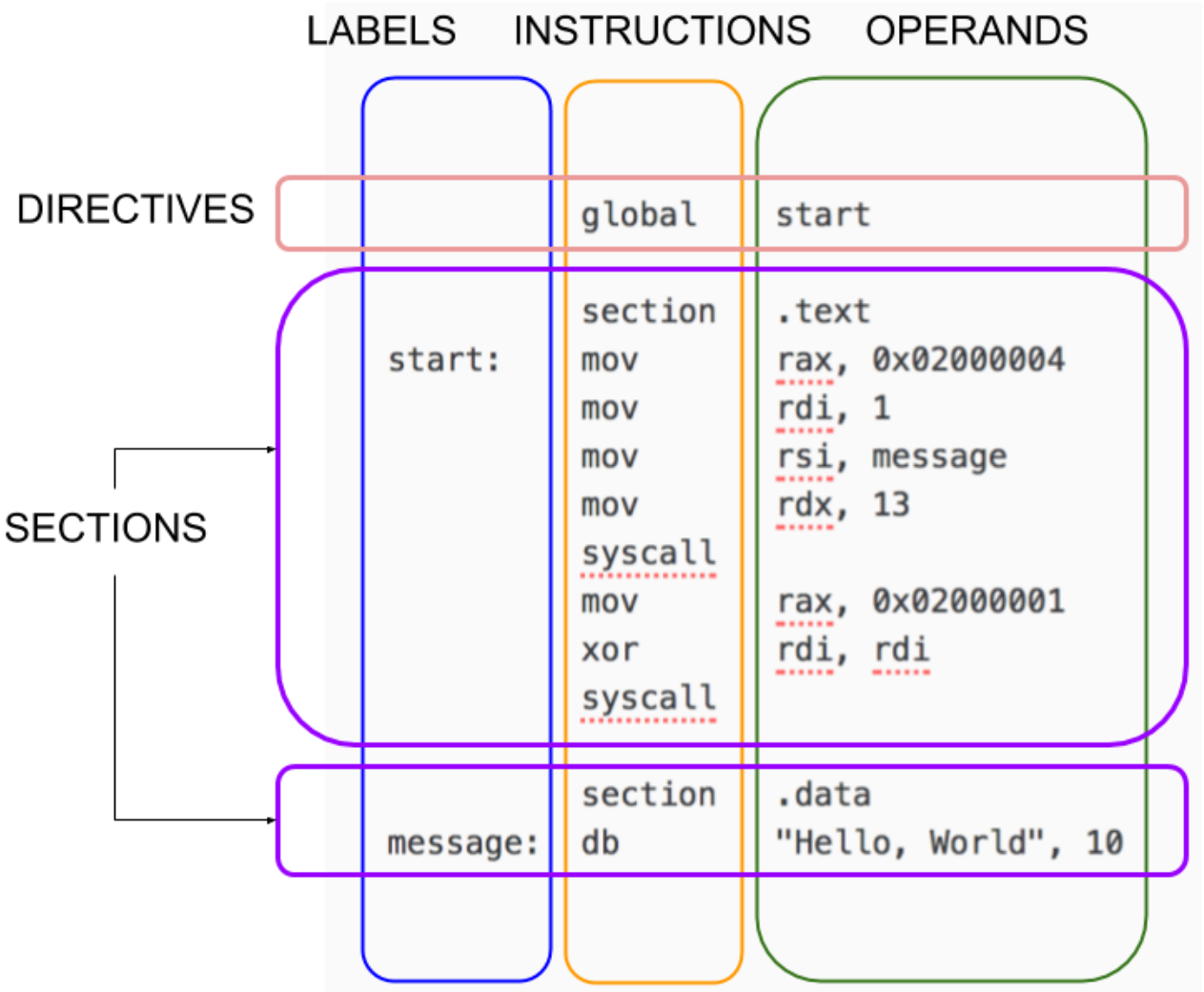
- Cài đặt NASM :
  - Vào cửa sổ dòng lệnh
  - Ra lệnh :
  - $ sudo apt-get install nasm #To install NASM
- Cài đặt Vim-gnome
  - Vào cửa sổ dòng lệnh
  - Ra lệnh
  - $ sudo apt-get install vim-gnome #To install VIM with Gnome GUI
- Cài đặt gcc
  - Ra lệnh $sudo apt-get install gcc

6.4. Cách thực hiện chương trình hợp ngữ trên máy tính

a. **B1:** Dùng một trình soạn thảo để soạn chương trình nguồn
b. **B2:** Lưu chương trình nguồn trên đĩa với phần đuôi là .nasm
c. **B3:** Biên dịch chương trình thành mã đối tượng .o
d. **B4:** Liên kết mã đối tượng thành mã thực thi
e. **B5:** Chạy thử

6.5. Cấu trúc chung của chương trình hợp ngữ trên NASM

An assembly program can be divided into three sections:



- **The .data section**

  This section is for "declaring initialized data", in other words defining "variables" that already contain stuff. However this data does not change at runtime so they're not really variables. The .data section is used for things like filenames and buffer sizes, and you can also define constants using the EQU instruction. Here you can use the DB, DW, DD, DQ and DTinstructions. For example:

```
section .data
message:    db 'Hello world!'
msglength:  equ 12
buffersize: dw 1024
```

- **The .bss section**

  This section is where you declare your variables. You use the RESB, RESW, RESD, RESQ and REST instructions to reserve uninitialized space in memory for your variables, like this:

```
section .bss
filename:   resb    255         ; Reserve 255 bytes
number:     resb    1           ; Reserve 1 byte
```

```
bignum:    resw   1          ; Reserve 1 word (1 word = 2 bytes)
realarray: resq   10         ; Reserve an array of 10 reals
```

- **The .text section**

  This is where the actual assembly code is written. The .text section must begin with the declaration global _start, which just tells the kernel where the program execution begins. (It's like the main function in C or Java, only it's not a function, just a starting point.) Eg :

```
section .text
  global _start

_start:
  pop    ebx       ; Here is the where the program actually begins
  .
  .
```

-

6.6. Thực hiện chương trình đầu tiên

a. Tạo thư mục TH-KTMT trên máy tính ubuntu

b. Mở trình soạn thảo Gvim và soạn chương trình nguồn như sau

```
; This is simple hello world code
; Author: SLAER (Shashank Gosavi)
 global _start
 section .text
_start:
 xor ecx, ecx          ; Clearing ECX
xor ebx, ebx           ; Clearing EBX
mul ecx                ; Clearing EAX, EDX
                       ; Write subroutine
mov eax, 0x4 ; Moving Write syscall number into EAX
mov ebx, 0x1 ; Moving file descriptor into EBX
mov ecx, $msg ; Moving actual buffer into ECX
mov edx, $len ; Moving the count into EDX
int 0x80 ; Interrupt 80

; Graceful Exit
mov eax, 0x1 ; Moving Exit sysscall number into EAX
mov ebx, 0x0 ; Moving status number = 0 in EBX
int 0x80 ; Interrupt 80

section .data
msg: db "Hello World!",0x0A
len: equ $-msg
```

c. Lưu chương trình nguồn với tên vidu1.nasm

d. Mở cửa sổ giao diện dòng lệnh (terminal)
e. Vào thư mục TH-KTMT (ra lệnh cd TH-KTMT)
f. Biên dịch chương trình nguồn

- Nasm –o vidu1.o -felf64 vidu1.nasm

g. Liên kết chương trình nguồn

- Ld –o vidu1 vidu1.o

h. Chạy thử

- ./vidu1

6.7. Ví dụ : Chương trình in ra hình sao tam giác

```
; ------------------------------------------------------------
; This is an OSX console program that writes a little triangle
; of asterisks to standard
; output. Runs on macOS only.
; nasm -fmacho64 triangle.asm && gcc hola.o && ./a.out
; ------------------------------------------------------------

          global    start
          section   .text
start:
          mov rdx, output    ; rdx holds address of next byte
to write
          mov       r8, 1    ; initial line length
          mov       r9, 0    ; number of stars written on line
so far
line:
          mov       byte [rdx], '*'         ; write single star
          inc       rdx                     ; advance pointer
to next cell to write
          inc       r9                      ; "count" number so
far on line
          cmp       r9, r8                  ; did we reach the
number of stars for this line?
          jne       line                    ; not yet, keep
writing on this line
lineDone:
          mov       byte [rdx], 10          ; write a new line
char
          inc       rdx                     ; and move pointer
to where next char goes
          inc       r8                      ; next line will be
one char longer
          mov       r9, 0                   ; reset count of
```

```
stars written on this line
          cmp       r8, maxlines            ; wait, did we
already finish the last line?
          jng       line                    ; if not, begin
writing this line
done:
          mov       rax, 0x02000004         ; system call for
write
          mov       rdi, 1                  ; file handle 1 is
stdout
          mov       rsi, output             ; address of string
to output
          mov       rdx, dataSize           ; number of bytes
          syscall                           ; invoke operating
system to do the write
          mov       rax, 0x02000001         ; system call for
exit
          xor       rdi, rdi                ; exit code 0
          syscall                           ; invoke operating
system to exit

          section   .bss
maxlines  equ       8
dataSize  equ       44
output:   resb      dataSize
```

- Lưu file với tên triangle.asm
- Biên dịch và liên kết
- Chạy thử

```
$ nasm -fmacho64 triangle.asm && ld triangle.o && ./a.out
*
**
***
****
*****
******
*******
********
```

7. **Bài tập đề nghị**
8.