Lecture # 29
High-Level Language Interface

- General and Calling Conventions
- Inline assembly code

Introduction

- **Where convenience and development time are more important than speed or code size**

  High-level language is used

- **Assembly language can be used for *fine-tuning* of such programs to speedup critical sections of code**
  - **Control high speed hardware**
  - **Memory-resident code**
  - **Access non-standard hardware devices**
  - **Write platform specific code**
  - **Extend the high-level language capabilities**
- ***Interface* or connection between high-level languages and assembly language**

General Conventions

- ***Naming Convention*** **refers to the rules or characteristics regarding the naming of variables and procedures**
- ***Memory model*** **used by a program determines the segment size and whether calls and references will be near or far**

➢ **Calling Conventions**

Refer to the low-level details about how procedures are called

- **Which registers must be preserved by called procedures**
- **The method used to pass the arguments**
- **Arguments passing order**
- **Whether arguments are passed by value or by reference**
- **How the stack pointer is restored after a procedure call**
- **How functions return values to the calling program**

General Conventions

- **External Identifiers**
  - **Must have compatible naming conventions**
    - **A C-compiler adds an underscore to all names**
      - **Preserves the case**
- **Segment Names**
  - **Segment names must be compatible**
- **Simplified segment directives (.code, .data, .stack) with the segment names produced by most C++ compilers**
- **Memory Models**
  - **A calling program and the called procedure must both use the same memory model**
    - **Model used in real address and protected address mode**

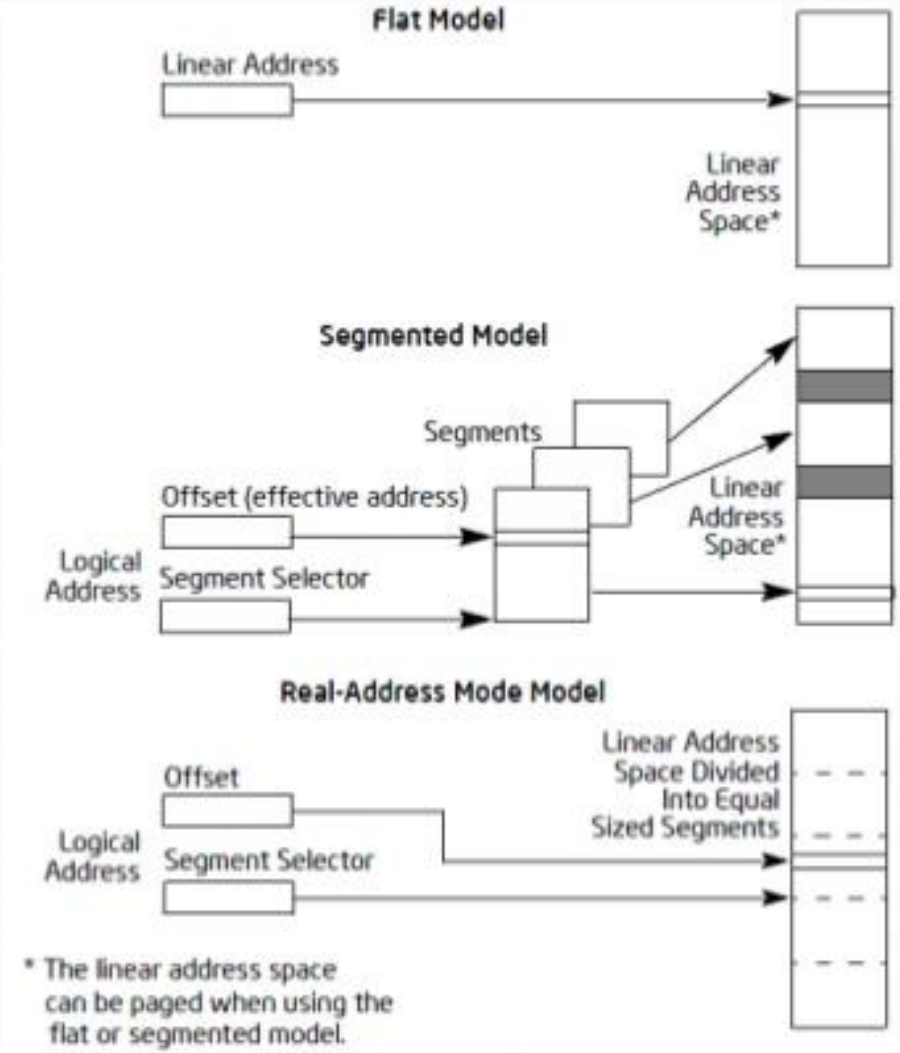
.Model Directive

- **This directives determines several characteristics of the program**
  - **Memory model type**
  - **Procedure naming scheme**
  - **Parameter passing convention**
- **Syntax is *.model memorymodel [, modeloptions]***
  - **All models except *flat* is used in real address mode programming**
    - **Only flat model is used in protected mode programming**
  - ***modeloptions* include language specifier**
    - **Determines calling and naming conventions for procedures and public symbols**

Table 13-1 Memory Models.

Model	Description
Tiny	A single segment, containing both code and data. This model is used by programs having a .com extension in their filenames.
Small	One code segment and one data segment. All code and data are near, by default.
Medium	Multiple code segments and a single data segment.
Compact	One code segment and multiple data segments.
Large	Multiple code and data segments.
Huge	Same as the large model, except that individual data items may be larger than a single segment.
Flat	Protected mode. Uses 32-bit offsets for code and data. All data and code (including system resources) are in a single 32-bit segment.

Memory Models



- No segmentation
- Code, Data, stacks are all contained in this address space.
- 32 bit addressing

- Code, Data, stacks are typically contained in separate segments for better isolation.
- 32 bit addressing (32 bit offset, 16 bit seg. selector)

- Compatibility mode for 8086 processor.
- 20 bit addressing (16 bit offset, 16 seg. selector)

Flat Memory Model

- Modern operating systems turn segmentation off
- Each program uses one 32-bit linear address space
  - Up to 2^{32} = 4 GB of memory can be addressed
  - Segment registers are defined by the operating system
  - All segments are mapped to the same linear address space
- In assembly language, we use .MODEL flat directive
  - To indicate the Flat memory model
- A linear address is also called a virtual address
  - Operating system maps virtual address onto physical addresses
  - Using a technique called paging

NEAR CALL

- 3 bytes long.
  - the first byte contains the opcode; the second and third bytes contain the displacement
- When the near CALL executes, it first pushes the offset address of the next instruction onto the stack.
  - offset address of the next instruction appears in the instruction pointer (IP or EIP)
- It then adds displacement from bytes 2 & 3 to the IP to transfer control to the procedure.
- Why save the IP or EIP on the stack?
  - the instruction pointer always points to the next instruction in the program
- For the CALL instruction, the contents of IP/EIP are pushed onto the stack.
  - program control passes to the instruction following the CALL after a procedure ends.

Prepared by [illegible]

FAR CALL

- 5-byte instruction contains an opcode followed by the next value for the IP and CS registers.
  - bytes 2 and 3 contain new contents of the IP
  - bytes 4 and 5 contain the new contents for CS
- Far CALL places the contents of both IP and CS on the stack before jumping to the address indicated by bytes 2 through 5.
- This allows far CALL to call a procedure located anywhere in the memory and return from that procedure.
- The program branches to the procedure.
  - A variant of far call exists as CALLF, but should be avoided in favor of defining the type of call instruction with the PROC statement
- In 64-bit mode a far call is to any memory location and information placed onto the stack is an 8-byte number.
  - the far return instruction retrieves an 8-byte return address from the stack and places it into RIP

Language Specifiers

- **STDCALL Specifier**

  Dictates that procedure arguments be pushed on the stack in reverse order – An example

  - **Also determines how procedure arguments are removed from the stack after a procedure call**

    A constant operand must be supplied to the RET instruction

    The constant is added to ESP after the return address is popped from the stack

- **STDCALL also modifies exported (public) procedure names by storing them in the format _*name@nn***

  ***nn* indicates the number of bytes used by the procedure parameters**

  Rounded upwards to a multiple of 4

Language Specifier

- ***C Specifier***

  Dictates that the procedure arguments are pushed on the stack in reverse order

- **Removing arguments**

  A constant is added in the calling program

  External procedure names are handled in the same way as STDCALL

- ***PASCAL Specifier***

  Dictates that procedure arguments be pushed in forward order

- **Removing arguments: Same as STDCALL**
- **Procedure Name**

  PASCAL: Procedure name is converted to uppercase