Indirect Addressing

==

Indirect Addressing

==

Indirect addressing is a memory addressing mode in which the **operand of an instruction is the address of the data to be operated on,** but the address of the data is stored in a memory location or register.

This means that the **CPU must first dereference the pointer to the data** before it can access it.

Indirect addressing is more complex than direct addressing, but it has several advantages.

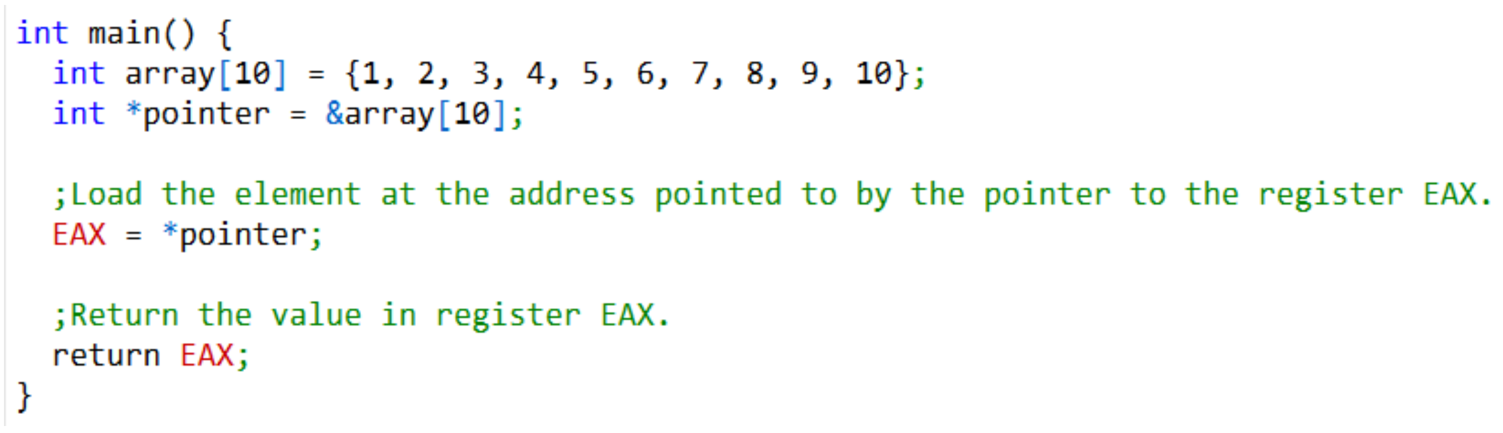
One advantage is that it allows the address of the data to be unknown at compile time.

This means that indirect addressing can be used to access data that is stored dynamically, such as data that is read from a file.

Another advantage of indirect addressing is that it allows the same code to be used to access different data items. This is useful for implementing functions such as loops and arrays.

Here are some examples of indirect addressing in assembly language:

```
; Load the content of memory address stored at memory address 100 to the register R1.
LOAD R1, @100

; Load the content of the memory address stored at register R2 to register R1.
LOAD R1, @R2

; Store the content of register R1 to the memory address stored at register R3.
STORE R1, @R3
```



```
int main() {
  int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int *pointer = &array[10];

  ;Load the element at the address pointed to by the pointer to the register EAX.
  EAX = *pointer;

  ;Return the value in register EAX.
  return EAX;
}
```

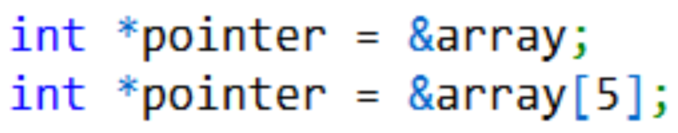
Indirect addressing is a powerful and versatile addressing mode, but it is important to be aware of its limitations.

One limitation is that indirect addressing can cause the code to be slower than direct addressing, because the CPU has to perform an extra step to dereference the pointer.

Another limitation of indirect addressing is that it can lead to errors if the pointer is not initialized properly or if it points to an invalid memory location.

int *pointer = &array[5]; is it necessary to write that 5 in bracket? why not &array ??

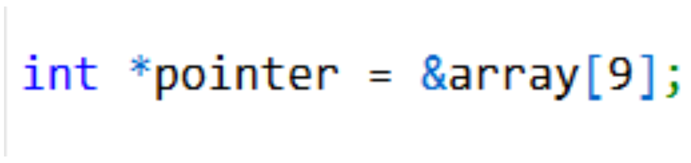
It is not necessary to write the 5 in brackets when declaring the pointer pointer. The following two declarations are equivalent:



```
int *pointer = &array;
int *pointer = &array[5];
```

The reason why you can omit the 5 in brackets is because arrays are implicitly converted to pointers to their first element. This means that the expression &array is equivalent to the expression &array[0].

However, there are some cases where it is good practice to explicitly specify the array index. For example, if you are declaring a pointer to the last element of an array, it is more clear to write:



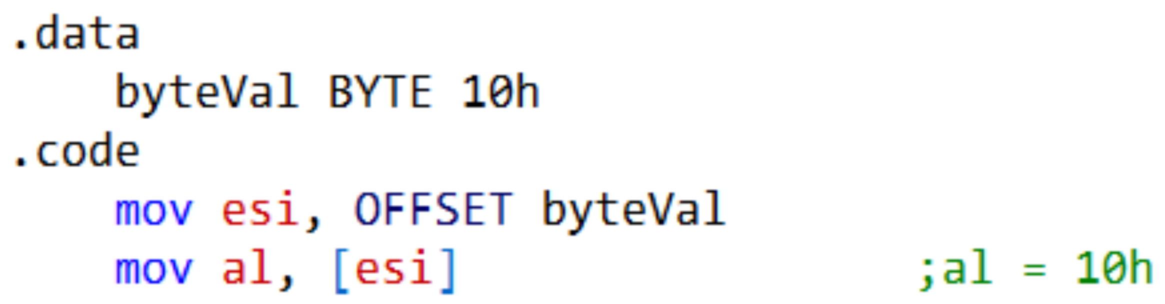
```c
int *pointer = &array[9];
```

This makes it clear that the pointer is pointing to the last element of the array, and not to the first element.

Another case where it is good practice to explicitly specify the array index is if you are using the pointer in a function or library that expects the pointer to point to a specific element of the array.

In general, it is up to you to decide whether or not to explicitly specify the array index when declaring a pointer. However, it is important to be aware of the implications of omitting the array index.

Another example:



```
.data
    byteVal BYTE 10h
.code
    mov esi, OFFSET byteVal
    mov al, [esi]                   ;al = 10h
```

The first line declares a byte variable named **byteVal** and initializes it with the value **0x10.**

The second line moves the offset of the variable byteVal into the register esi.

The offset of a variable is the address of the variable relative to the beginning of the data segment.

In other words, the offset of a variable is the number of bytes from the beginning of the data segment to the first byte of the variable.

In the example you provided, the variable byteVal is declared in the data segment.

This means that the offset of the variable byteVal is the address of the variable byteVal relative to the beginning of the data segment.

The following assembly language instruction moves the offset of the variable byteVal into the register esi:

```
mov ESI, OFFSET byteVal
```

After this instruction is executed, the register esi will contain the offset of the variable byteVal.

The offset of a variable can be used to load or store the value of the variable. For example, the following assembly language instruction loads the value of the variable byteVal into the register al:

```
mov al, [ESI]
```

The offset of a variable is also used by the compiler to generate machine code for function calls. When a function is called, the compiler generates machine code to push the arguments to the function onto the stack. The compiler also generates machine code to call the function.

When the function returns, the compiler generates machine code to pop the arguments to the function from the stack. The compiler also generates machine code to return from the function.

The compiler uses the offset of the function arguments to generate machine code to push the arguments onto the stack. The compiler also uses the offset of the function return value to generate machine code to pop the return value from the stack.

In summary, the offset of a variable is the address of the variable relative to the beginning of the data segment. The offset of a variable can be used to load or store the value of the variable. The offset of a variable is also used by the compiler to generate machine code for function calls.

In this code, we have a variable byteVal declared in the .data section with the value 10h (which is 16 in decimal).

The OFFSET keyword is used to calculate the memory address of byteVal and store that address in the esi register. Here's how the math works:

1. byteVal is declared as a byte variable, which means it occupies one byte in memory.

2. When you use OFFSET byteVal, the assembler calculates the memory address where byteVal is located. **Since byteVal is the first data item declared in the .data section, it's at the beginning of the data segment.**

3. In x86 assembly, memory addresses start at 0x0 for the beginning of the data segment. Therefore, the memory address of byteVal is 0x0.

4. The OFFSET keyword essentially says, **"Give me the memory address of byteVal relative to the start of the data segment."** Since byteVal is at the start of the data segment, the relative address is 0x0.

5. The **mov esi, OFFSET byteVal** instruction places the value 0x0 (the memory address of byteVal) into the esi register.

Now, with esi containing 0x0, the next instruction mov al, [esi] reads the byte value at memory address 0x0, which is the value of byteVal, and stores it in the al register. So, al is set to **10h (or 16 in decimal).**

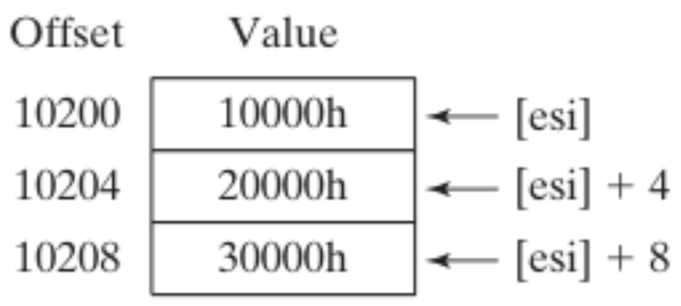
In summary, the OFFSET keyword calculates the relative memory address of byteVal within the data segment, which is 0x0, and that address is loaded into the esi register. This allows subsequent instructions to access the value of byteVal in memory.

```
.data
arrayD DWORD 10000h, 20000h, 30000h  ; Define an array of doublewords

.code
mov esi, OFFSET arrayD    ; Load the memory address of the array into ESI

; Access the first number
mov eax, [esi]              ; Load the first doubleword into EAX
; ESI points to the first doubleword

; Access the second number
add esi, 4                  ; Move ESI to the next doubleword (increment by 4 bytes)
mov eax, [esi]              ; Load the second doubleword into EAX
; ESI points to the second doubleword

; Access the third number
add esi, 4                  ; Move ESI to the next doubleword (increment by 4 bytes)
mov eax, [esi]              ; Load the third doubleword into EAX
; ESI points to the third doubleword
```

This code defines an array of three doublewords (arrayD) and then uses the OFFSET keyword to obtain the memory address of the array.

It then accesses each doubleword within the array sequentially by updating the esi register to point to the next doubleword using the add esi, 4 instruction before loading its value into the eax register.

Suppose arrayD is located at offset 10200h.


Offset
Value
10200
10000h
[esi]
10204
20000h
[esi] + 4
10208
30000h
[esi] + 8