Assembler Arrays

Declaring an array

An array is declared as shown below, with the values listed, separated by commas.

```
arr  dword   5, 7, 19
```

You could also declare the array elements as individual items:

```
arr  dword   5
     dword   7
     dword   19
```

Notice that only the first item in the array has a name attached. How can we refer to the remaining items? We can use the offset, which is the distance of each element, in bytes, from the beginning of the array.

arr

+0	+4	+8
5	7	19

We address an array element using square brackets around an address calculation: [arr+4].

We could sum the array using these values:

```
sub  eax,eax
add  eax,[arr+0]
add  eax,[arr+4]
add  eax,[arr+8]
```

The problem with this is that we can't use a loop. How can we change the code to use a loop?

Using an index register to process an array

To process an array in a loop, we can put the offset into a register. The register we use to hold an offset is called an index register. Traditionally the registers used for this purpose are esi or edi. The 'I' stands for "index."

To initialize an index register, zero it out:

```
sub esi,esi           ; index register
```

Then use the register in place of the constant: add eax,[arr+esi].
This line of code can be reused in a loop by changing the value of esi:

To process an array using an index register, start with initializing a loop counter: to do that, put the number of values in the array into ecx. Also initialize the index register to 0. In the loop, increment the index register by the size of each array element, which is 4 for a doubleword.

In this example, we'll sum the 4 values in the arr array; that means we must also set sum (eax) to 0 and add each array value into eax:

```
        mov ecx, 3              ;  loop counter
        sub   esi,esi           ;  index register
        sub   eax,eax           ;  sum =  0
top:    add   eax,[arr+esi]
        add   esi,4             ; increment index
        loop  top
```

Using a base register to process an array

Another way to process an array is to use a base register. While the index register holds the offset of the array from the beginning of the array, the base register holds the address of the array from the beginning of the data segment:

Suppose the data segment looks like the following (the left column is the offset of each item from the beginning of the data segment, measured in bytes).

```
            .data
0000   x    dword   10
0004   arr  dword   5, 7, 19
0010   n    dword   3
```

If we were to list all the elements individually, the address of each element would be clearer:

```
            .data
0000   x    dword   10
0004   arr  dword   5
0008        dword   7
000C        dword   19
0010   n    dword   3
```

```
arr
+0        +4         +8             <-- index register contents
|        5 |        7 |        19 |
0004      0008       000C           <-- base register contents
```

To process an array using a base register, you must put the address of the first element of the array into a base register. Traditionally the base register is ebx.

The instruction to put the address of an array into a register is lea, or load effective address:

```
        lea   ebx,arr
```

Using the example above, this puts 0004 into ebx.

Base register notation is the following: [ebx]

To move the first value from the array into eax, write the following:

```
        mov   eax,[ebx]
```

To change to the next value in the array, increment ebx by the size of each array element; in an array of dwords, this is 4:

```
          add ebx,4
```

To process an array using a base register, start with initializing a loop counter Also initialize the index register to 0. In the loop, increment the base register by the size of each array element, which is 4 for a doubleword:

To process an array using a base register, start by initializing a loop counter. Also initialize the base register to the initial address of the array. In the loop, increment the base register by the size of each array element, which is 4 for a doubleword.

In this example, we'll sum the 4 values in the arr array; that means we must also set sum (eax) to 0 and add each array value into eax:

```
      mov ecx,3             ;  loop counter
      lea   ebx, arr        ;  base register
      sub   eax,eax         ;  sum =  0
top:  add   eax,[ebx]
      add   ebx,4            ; increment base register
      loop  top
```

Passing an array as a parameter

Between index register and base register notation, index register notation is clearer and easier to read. However, it is necessary to use base register notation when sending an array as a parameter to a procedure. A procedure can't refer to any variables declared in main, and the array is declared in main.

```
             .data
0000   x     dword   10
0004   arr  dword   5, 7, 19
0010   n    dword   4
```

To send an array as a parameter to a procedure, you must pass the base address of the array. It is most efficient to pass the address in ebx. In addition, you must pass the number of filled positions in the array; this is not passed by address or reference, but by value: you put the value of n into a register. It is most efficient to put that value into ecx, which Is where it will be used.

To pass an array arr and n as parameters to the function addup, do the following. The function will return the sum in eax:

```
              lea   ebx,arr
              mov ecx,n
              call   addup
              mov sum,eax         ; store return value
```

Inside the function, you can begin to process the array directly using base address notation, since the address is already in ebx, and the loop counter is already in ecx.

```
 addup      proc
            sub  eax, eax         ; sum = 0
  addtop:   add  eax,[ebx]
            add  ebx,4
            loop addtop
            ret
addup       endp
```

What's wrong with this? We haven't pushed and popped registers that are changed but that are not used to return a value. Those would be ebx and ecx, so we modify the code as follows:

```
 addup      proc
            push ebx
            push ecx
            sub   eax, eax         ; sum = 0
  addtop:   add   eax,[ebx]
            add   ebx,4
            loop  addtop
            pop   ecx
            pop   ebx
            ret
addup       endp
```

Reference parameters

As you learned in C++, an array is passed by reference. That's what is happening when we pass the address of an array to a procedure. Any change to the array inside the procedure will be reflected upon return to main.

To pass a scalar (individual) value by reference, put its address in the register.
Use the lea instruction:

```
        .data
num dword  9
        .code

        lea ebx,num
        mov  eax,6
        call change_num


change_num proc
         sub  [ebx], eax
        ret
change_num endp
```

This example (silly, of course) shows how to change num inside the procedure. Upon entry, num has the value 9, from which the function subtracts 6. Upon return to main, num has the value 3.

I passed the 6 as a parameter because of a the problem with using a base address and a literal.

```
       sub [ebx],6
```

Machine can't tell from either operand the size of the operands. Solution:

```
      sub dword ptr[ebx],6
```

dword ptr is necessary to provide the size of the operands. More on this later.

Another example using a local variable

```
          lea  ebx,arr
          mov ecx,n
          call   lessthan10

;count number of values in array
;that are less than 10
lessthan10 proc
                 .data
```

```
val             dword  10
                .code
                push ebx
                push ecx
                push edx
                sub eax,eax  ; counter
                mov edx,val
   cmptop:      cmp [ebx],edx
                jge outt
                inc eax       ; special case
                add ebx,4     ; all cases
                loop cmptop
    outt:        pop edx
                 pop ecx
                 pop ebx
                 ret
     lessthan10 endp
```