Assembly Language

Tiago Oliveira
Instituto de Matemática e Estatística
Departamento de Ciência da Computação
Universidade Federal da Bahia

computação
UFBA

Indirect Addressing

Indirect addressing allows registers to act like pointer variables. To indicate that a register is to be used indirectly as a pointer, it is enclosed in square brackets ([ ])

```
mov     ax, [Data]      ; normal direct memory addressing of a word
mov     ebx, Data       ; ebx = & Data
mov     ax, [ebx]       ; ax = *ebx
```

AX holds a word, line 3 reads a word starting at the address stored in EBX. Registers do not have types like variables do in C.

All the 32-bit general purpose (EAX, EBX, ECX, EDX) and index (ESI, EDI) registers can be used for indirect addressing.

Simple Subprogram Example

Like a function in C. Could be done using the indirect form of the JMP instruction;

This form of the instruction uses the value of a register to determine where to jump to (thus, the register acts much like a function pointer in C.)

```
 mov     ecx, ret1          ; store return address into ecx
 jmp     short get_int      ; read integer
get_int:
      call    read_int
      mov     [ebx], eax          ; store input into memory
      jmp     ecx                 ; jump back to caller
ret1:
      mov     eax, prompt2       ; print out prompt
```

There is a much simpler way to invoke subprograms. This method uses the **stack**.

The Stack

Many CPU's have built-in support for a stack. A stack is a Last-In First-Out (LIFO) list. The stack is an **area of memory** that is organized in this fashion.

The **PUSH instruction adds data to the stack** and the **POP instruction removes data**.

The SS **segment register** specifies the segment that contains the stack (usually this is the same segment data is stored into).

The Stack

The **ESP register contains the address** of the data that would be removed from the stack (top of the stack).

Data can **only be added in double word** units. **Can not push a single byte** on the stack.

Actually **words** can be pushed too, but in **32-bit protected mode**, it is better to work with only double words on the stack

The PUSH instruction **inserts a double word 1 on the stack by subtracting 4 from ESP** and then stores the double word at [ESP]. The **POP instruction reads the double word at [ESP]** and then adds 4 to ESP.

The Stack

The code below demostrates how these instructions work and assumes that ESP is initially 1000H.

```
push    dword 1     ; 1 stored at 0FFCh, ESP = 0FFCh
push    dword 2     ; 2 stored at 0FF8h, ESP = 0FF8h
push    dword 3     ; 3 stored at 0FF4h, ESP = 0FF4h
pop     eax         ; EAX = 3, ESP = 0FF8h
pop     ebx         ; EBX = 2, ESP = 0FFCh
pop     ecx         ; ECX = 1, ESP = 1000h
```

The Stack

The stack can be used as a convenient place to store data temporarily.

It is also **used for making subprogram calls**, passing parameters and local variables.

The 80x86 also provides a **PUSHA** instruction that pushes the values of EAX, EBX, ECX, EDX, ESI, EDI and EBP registers (not in this order).

The POPA instruction can be used to pop them all back off.

The CALL and RET Instructions

The **CALL** instruction makes an unconditional jump to a subprogram and pushes the address of the next instruction on the stack.

The **RET** instruction pops off an address and jumps to that address. When using these instructions, it is very important that one **manage the stack correctly so that the right number is popped off by the RET instruction**!

The CALL and RET Instructions

Remember it is very important to pop off all data that is pushed on the stack.

Pops off EAX value, not return address!!

```
mov     ebx, input1
call    get_int

mov     ebx, input2
call    get_int

get_int:
        call    read_int
        mov     [ebx], eax
        ret
```

```
get_int:
        call    read_int
        mov     [ebx], eax
        push    eax
        ret
```

Passing parameters on the stack

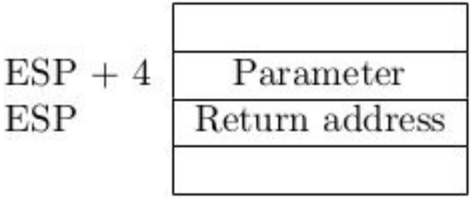
They are pushed onto the stack before the CALL instruction;

If the **parameter is to be changed** by the subprogram, **the address** of the data **must be passed**, not the value;

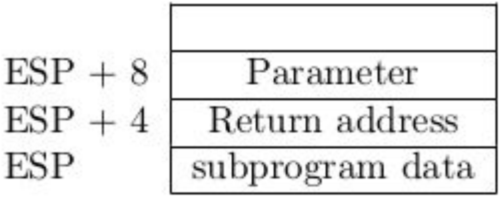
If the parameter's size is **less than a double word**, it **must be converted** to a double word before being pushed;

Passing parameters on the stack

Consider a subprogram that is passed a single parameter ([ESP+4]) on the stack. When the subprogram is invoked, the stack looks like this:



If if a DWORD is pushed the stack (parameter [ESP+8]):



Passing parameters on the stack

If the stack is also used inside the subprogram to store data, the number needed to be added to **ESP will change**.

Thus, it can be very error prone to use ESP when referencing parameters. To solve this problem, the 80386 supplies another register to use: **EBP**. This register's only purpose is to **reference data on the stack**.

Passing parameters on the stack

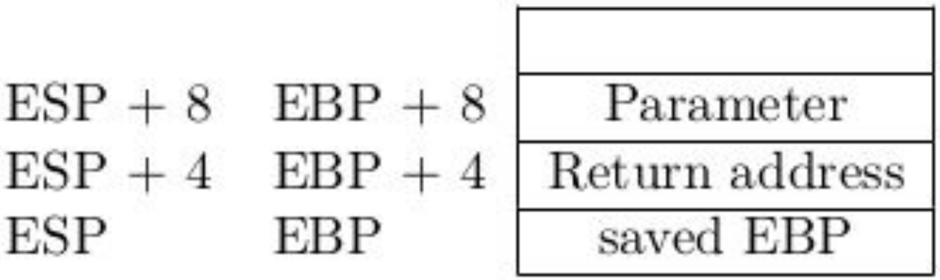
The C calling convention mandates that a subprogram first **save the value of EBP on the stack and then set EBP to be equal to ESP**. This allows ESP to change as data is pushed or popped off the stack without modifying EBP. At the **end of the subprogram**, the original value of **EBP** must be **restored** (this is why it is saved at the start of the subprogram.

```
subprogram_label:
      push   ebp              ; save original EBP value on stack
      mov    ebp, esp         ; new EBP = ESP
; subprogram code
      pop    ebp              ; restore original EBP value
      ret
```

Passing parameters on the stack

Now the **parameter** can be access with **[EBP + 8] at any place** in the subprogram without worrying about what else has been **pushed** onto the stack by the **subprogram**.

After the subprogram is over, the parameters that were pushed on the stack must be removed.



Local variables on the stack

The **stack** can be used as a convenient location **for local variables**.

Data stored on the stack **only use memory** when the subprogram they are defined for is **active**.

Local variables are stored right after the saved EBP value in the stack.

They are **allocated by subtracting the number of bytes required from ESP** in the prologue of the subprogram.

Local variables on the stack

```
subprogram_label:
      push   ebp                  ; save original EBP value on stack
      mov    ebp, esp             ; new EBP = ESP
      sub    esp, LOCAL_BYTES     ; = # bytes needed by locals
; subprogram code
      mov    esp, ebp             ; deallocate locals
      pop    ebp                  ; restore original EBP value
      ret
```

```
void calc_sum( int n, int * sump )
{
  int i, sum = 0;

  for( i=1; i <= n; i++ )
    sum += i;
  *sump = sum;
}
```

ESP + 16	EBP + 12	sump
ESP + 12	EBP + 8	n
ESP + 8	EBP + 4	Return address
ESP + 4	EBP	saved EBP
ESP	EBP - 4	sum

```
cal_sum:
      push   ebp
      mov    ebp, esp
      sub    esp, 4                     ; make room for local sum

      mov    dword [ebp - 4], 0         ; sum = 0
      mov    ebx, 1                     ; ebx (i) = 1
for_loop:
      cmp    ebx, [ebp+8]               ; is i <= n?
      jnle   end_for

      add    [ebp-4], ebx               ; sum += i
      inc    ebx
      jmp    short for_loop

end_for:
      mov    ebx, [ebp+12]              ; ebx = sump
      mov    eax, [ebp-4]               ; eax = sum
      mov    [ebx], eax                 ; *sump = sum;

      mov    esp, ebp
      pop    ebp
      ret
```

Multi-Module Programs

A multi-module program is one composed of more than one object file. All the programs presented here have been multi-module programs. The linker must match up references made to each label in one module.

In order for module A to use a label defined in module B, the **extern** directive must be used. **After the extern directive comes a comma delimited list of labels**.

In assembly, labels can not be accessed externally by default. If a label can be accessed from other modules than the one it is defined in, it must be declared **global in its module**. **Global data labels** work exactly the same way.

Interfacing Assembly with C

This can be done in two ways: calling assembly subroutines from C or inline assembly.

Inline assembly **disadvantages:**

The assembly code must be written in the format the compiler uses.

No compiler at the moment supports NASM's format;

Different compilers require different formats;

Interfacing Assembly with C

The technique of **calling an assembly subroutine** is much more **standardized on the PC**.

Assembly routines are usually used with C for the following reasons:

- Direct access is needed to hardware features of the computer that are difficult or impossible to access from C.
- The routine must be as fast as possible and the programmer can hand optimize the code better than the compiler can

Saving registers

First, C **assumes that a subroutine maintains the values** of the following **registers**: EBX, ESI, EDI, EBP, CS, DS, SS, ES. This does not mean that the **subroutine can not change them internally**. Instead, it means that if it does change their values, it **must restore their original values before the subroutine returns**.

The EBX, ESI and EDI values **must be unmodified** because **C uses these registers** for register variables. Usually the **stack is used to save the original values of these registers**.

Labels of functions

Most C compilers prepend a single underscore (_) character at the beginning of the names of functions and global/static variables.

For example, a function named **f** will be assigned the label **_f**.

Thus, if this is to be an assembly routine, it **must be labelled _f**, not f.

Passing parameters

Under the C calling convention, the arguments of a function are pushed on the stack in the reverse order that they appear in the function call.

Calculating addresses of local variables

Calculating the address of a local variable (or parameter) on the stack is not as straightforward.

In case of passing the address of a variable to a function If is located at EBP – 8 on the stack, one cannot just use: **mov eax, ebp - 8**.

It is called LEA (for Load Effective Address). The following would calculate the address of x and store it into EAX: **lea eax, [ebp - 8]**.

Now EAX holds the address of the variable and could be pushed on the stack when calling function

Returning values

Return values are passed via registers; returned in the EAX register.

If they are smaller than 32-bits, they are extended to 32-bits when stored in EAX (depends on if they are signed or unsigned types).

64-bit values are returned in the EDX:EAX register pair.

Pointer values are also stored in EAX.

Floating point values are stored in the *ST0 register* of the math coprocessor.

Reentrant and Recursive Subprograms

A reentrant subprogram must satisfy the following properties:

It must not modify global data;

All variables are stored on the stack;

A reentrant program can be shared by multiple processes.

Reentrant subprograms work much better in multi-threaded 5 programs.

A reentrant subprogram can be called recursively.

Recursive subprograms

These types of subprograms call themselves. The recursion can be either direct or indirect.

Direct recursion occurs when a subprogram, say foo, calls itself inside foo’s body.

Indirect recursion occurs when a subprogram is not called by itself directly, but by another subprogram it calls. For example, subprogram foo could call bar and bar could call foo.

A multi-threaded program has multiple threads of execution. That is, the program itself is multi-tasked.

Recursive subprograms

```
; finds n!
segment .text
      global _fact
_fact:
      enter  0,0

      mov    eax, [ebp+8]     ; eax = n
      cmp    eax, 1
      jbe    term_cond        ; if n <= 1, terminate
      dec    eax
      push   eax
      call   _fact            ; eax = fact(n-1)
      pop    ecx              ; answer in eax
      mul    dword [ebp+8]    ; edx:eax = eax * [ebp+8]
      jmp    short end_fact
term_cond:
      mov    eax, 1
end_fact:
      leave
      ret
```

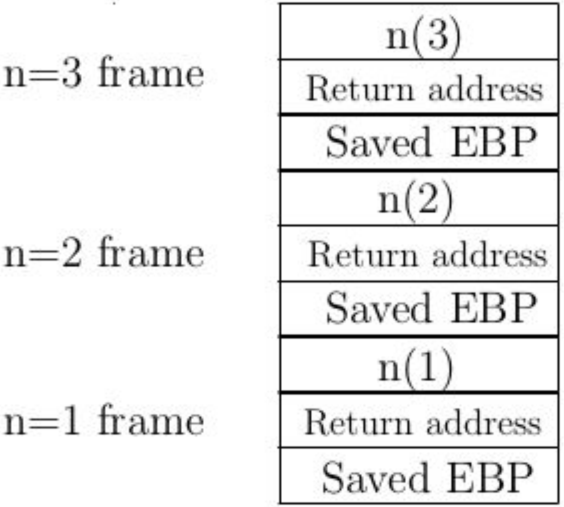
enter and leave

Enter creates a stack frame

enter 4, 0

leave destroys a stack frame

```c
void f( int x )
{
  int i;
  for( i=0; i < x; i++ ) {
    printf ("%d\n", i);
    f(i);
  }
}
```




```nasm
%define i ebp-4
%define x ebp+8          ; useful macros
segment .data
format      db "%d", 10, 0     ; 10 = '\n'
segment .text
      global _f
      extern _printf
_f:
      enter  4,0          ; allocate room on stack for i

      mov    dword [i], 0  ; i = 0
lp:
      mov    eax, [i]      ; is i < x?
      cmp    eax, [x]
      jnl    quit

      push   eax           ; call printf
      push   format
      call   _printf
      add    esp, 8

      push   dword [i]     ; call f
      call   _f
      pop    eax

      inc    dword [i]     ; i++
      jmp    short lp
quit:
      leave
      ret
```

Bibliografia

CARTER, Paul A. **PC Assembly Language.** Github, 2004.

Contatos:
tiagocompuesc@gmail.com