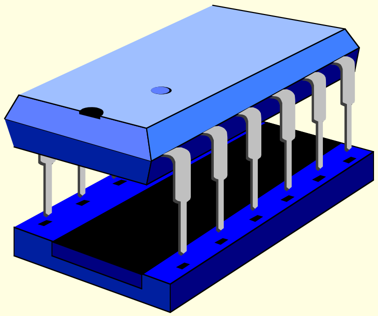
Pentium Architecture:

Methods

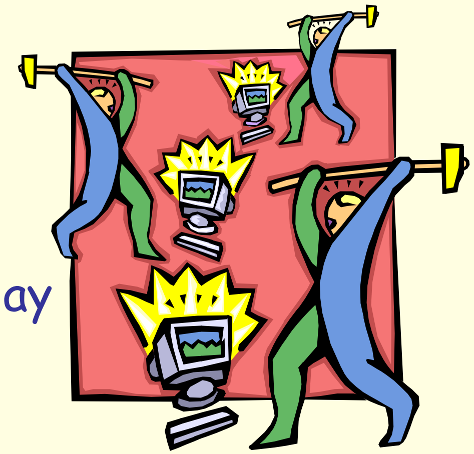
Kin K. Leung

kin.leung@imperial.ac.uk

www.commsp.ee.ic.ac.uk/~kkleung/



Heavily rely on materials from Naranker Dulay



Methods (Procedures and Functions)

- The ability to jump to the beginning of a method **(CALL)** and on completion, the ability to jump back to the instruction following the corresponding method call **(RETURN)**.
- For "function" methods, the ability to pass the **RESULT** value back to the calling method.
- The ability to pass **PARAMETERS** to a method.
- The ability to allocate and access variables that are **LOCAL** to the method.
- For object methods the ability to access the **FIELDS of the OBJECT**
- The ability to make **NESTED** and **RECURSIVE** method calls.

Stacks

- Methods are normally implemented using a *stack*.
- A stack is a region of main memory accessed in a very specific & disciplined way:
- There are 2 Basic Stack Operations:

  PUSH data onto the Top of Stack

  POP data from the Top of Stack

- Stacks follow the *Last-In, First-Out (LIFO) Rule:*

  Last Data Pushed = First Data Popped

Pentium System Stack

- The Pentium provides a "System" Stack, and a group of instructions for managing it.
- The **stack pointer** register (esp) holds the address of the top of stack
- We'll also use the **base pointer** register (ebp) to access data on the stack, typically the parameters and local variables of a method.
- **WARNING**:
  On the Pentium, the value in the stack pointer register (esp) must always be even (word-aligned), e.g. we cannot push/pop a byte directly as this would make esp an odd address.

PUSH and POP Instructions

Push Instruction	Pop Instruction	Notes
`push opr`	`pop opr`	push/pop word or doubleword depending on operand size
`pushfd`	`popfd`	push/pop eflags register

We can only push (pop) operands that are **word** sized or **doubleword** sized with these instructions. Bytes need special handling e.g. we can push a word and then move bytes to it using Register Relative addressing

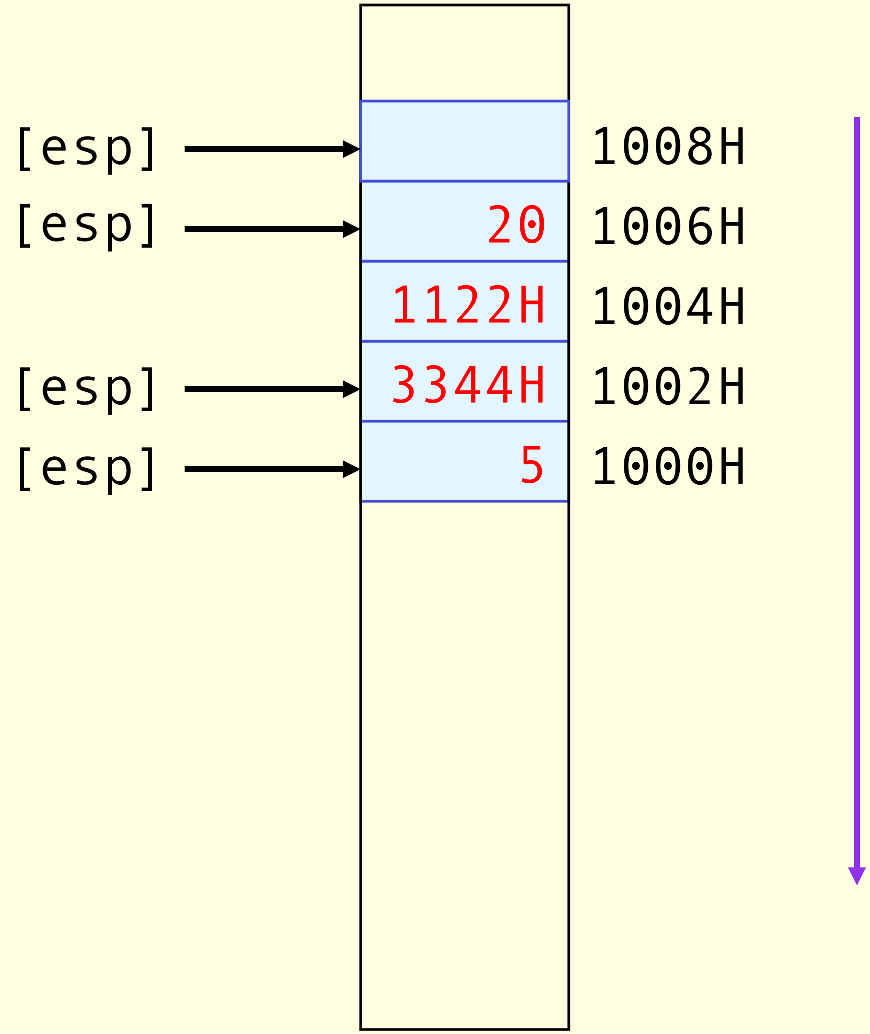
PUSH and POP in Detail (using ESP)

On the Pentium we grow the system stack **downwards** in memory with push instructions (i.e. higher addresses to lower addresses) and shrink it upwards with pop instructions (i.e. lower addresses to higher addresses)

```
push wordop      esp = esp - 2
                 memory[esp] = wordopr

pop  wordop      wordop = memory[esp]
                 esp = esp + 2

push dwordop     esp = esp - 4
                 memory[esp] = dwordop

pop  dwordop     dwordop = memory[esp]
                 esp = esp + 4
```

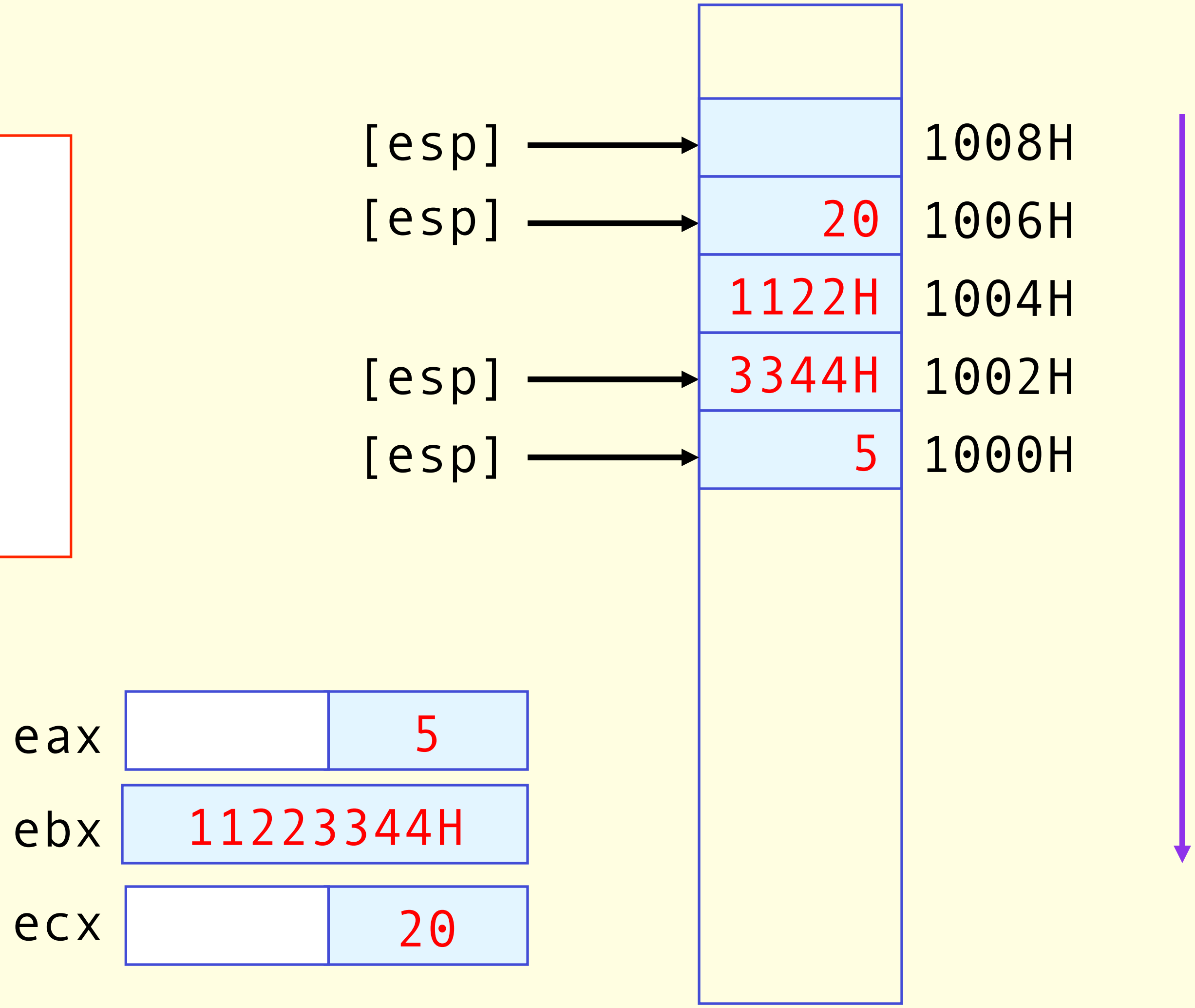
Example

```
push word 20

push dword 11223344H

push word 5
```



Example Contd.

```
pop ax

pop ebx

pop cx
```



Our Calling Convention

CALLING Method (CALLER)

Pass Parameters if any

Pass Object Instance

Call Method

Remove Parameters & Object Instance

Copy or Apply Method Result

CALLED Method (CALLEE)

Setup Frame Pointer (`ebp`) & Allocate Local Variables (Method Entry)

Save registers (on the Stack)

Execute Body of Method

Copy Method Result (if any) to `eax`

Restore Registers (from the Stack)

De-allocate Local Variables and Restore Frame Pointer (Method Exit)

Return from Method

Our Parameter Passing Convention

Caller Actions:

Push Last (rightmost) parameter onto the stack
Push Next-to-Last Parameter onto the stack
...
Push 2nd Parameter onto the stack
Push 1st Parameter onto the stack
Push Object Instance
Call Method
Remove Parameters & Object Instance from stack
Expect method result in register `eax` (or `ax` if 16-bit or `al` if 8-bit)

Other Parameter Passing Conventions:

Pass parameters left-to-right (Push 1st parameter first)
Pass parameters via registers
Return method result via the stack

CALL and RETURN Instructions

```
call method    = push eip       ; push return address and
jmp method                 ; jump to start of method

ret            = pop  eip       ; pop return address into eip
```

Since `eip` is incremented during the Fetch-Execute cycle, the return address is the address of the next instruction, i.e. the instruction to resume execution after completion of the called method.

Example: Max (Caller)

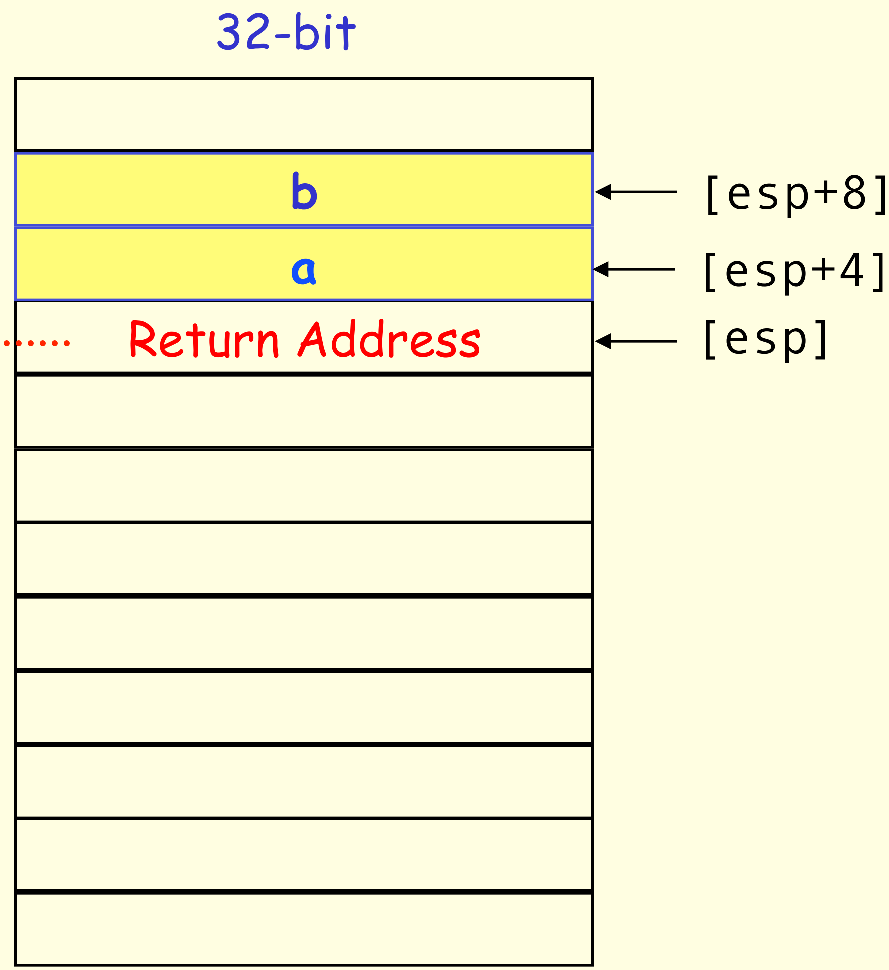
int a, b

// We'll use 32-bit integers

....

a = *max* (a, b)

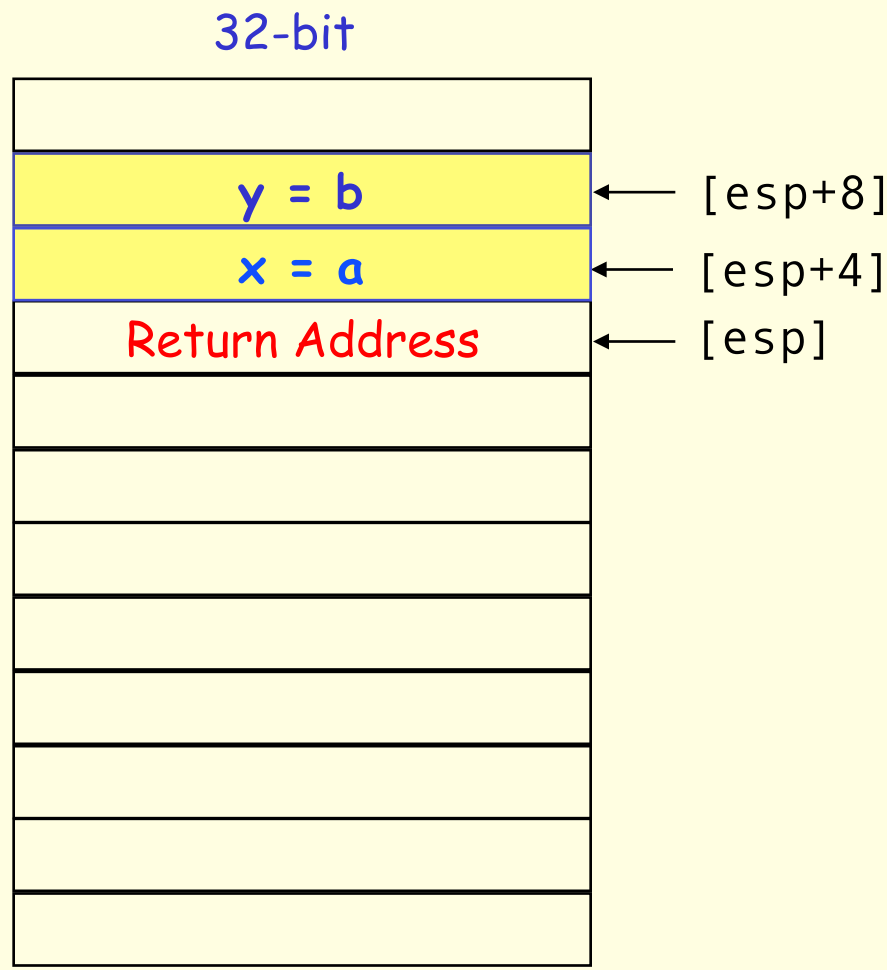
```
a   dd   0    ; doubleword
b   dd   0    ; doubleword
        ...
    push dword [b]
    push dword [a]
    call max
    add   esp, 8
    mov   [a], eax
```



Example: Max

```
int max ( int x, y ) {
    eax = x
    if (eax < y) eax = y
}
```

```
max:
    mov eax,[esp+4]   ; eax=x
    cmp eax,[esp+8]   ; is eax>=y
    jge endmax
    mov eax, [esp+8]  ; eax=y
endmax:
    ret
```



Local Variables

- The "lifetime" of local variables is limited to the execution of the method they are declared in.

- We can allocate/deallocate local variables on the system stack. But as an optimisation and for convenience we'll use registers for local variables instead of the stack.

- Local variables & parameters allocated on the stack will be accessed indirectly via the **base pointer** register (ebp). When used in this way ebp is known as the **frame pointer** (or link pointer or local Base).

  Unlike the stack pointer which can change during a method's execution, the frame pointer will be "anchored" (i.e. will not change) for the execution of the method.

Method Entry & Exit

- Setup Frame Pointer & Allocate space for Local Variables (Entry)

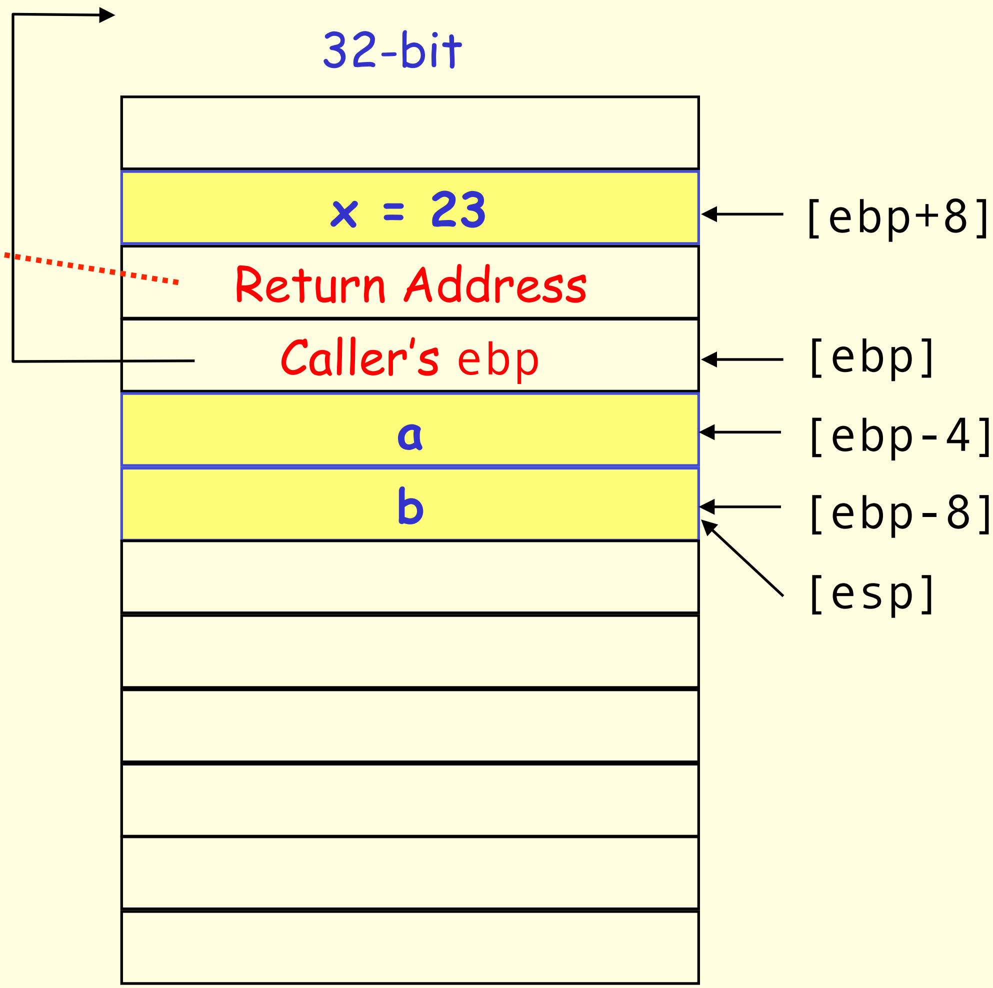
```
push ebp            ; save caller's frame pointer on the stack
mov  ebp, esp       ; set frame pointer for called method

sub  esp, nbytes    ; allocate nbytes for local variables
                    ; nbytes is normally a constant value
```

- De-allocate Local Variables and Restore Frame Pointer (Exit)

```
mov esp, ebp              ; restore stack pointer to that on
entry
pop ebp             ; restore caller's frame pointer
```

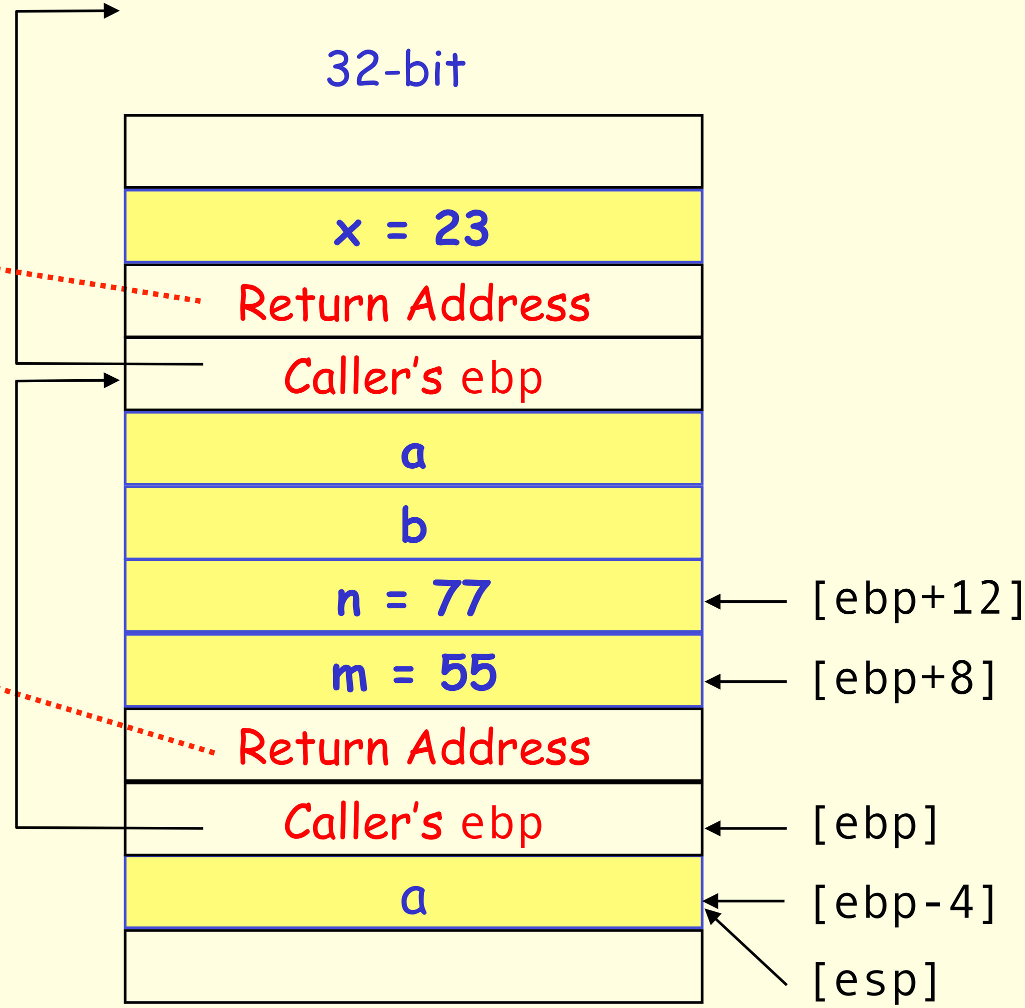
Stack Frame (Activation Record)

```
void Alpha ( ) {
        Beta (23)
        statements
}

void Beta (int x) {
        int a, b
        // *** We are here ***
        Gamma (55, 77)
        statements
}

void Gamma (int m,n) {
        int a
        statements
}
```



Stack Frame Contd.

```
void Alpha ( ) {
        Beta (23)
        statements
}

void Beta (int x) {
        int a, b
        Gamma (55, 77)
        statements
}

void Gamma (int m,n) {
        int  a
        //*** We are here ***
        statements
}
```



Array & Object Parameters

- For array and object parameters we push the start address of the array or object onto the stack rather than its value. Within the method we access the passed array/object indirectly via the pushed address.
- The address of an array/object can be computed with the Load Effective Address (`lea`) instruction which takes the general form:

  lea *Register*, [*BaseReg* + *Scale*IndexReg* + *Displacement*]
- this performs the following assignment:

  Register = BaseReg + Scale*Index + Displacement
- Note: `lea` only computes the address and assigns it to the register it does not access the memory location pointed to by the computed address !

```
lea esi, [ebp+4]          ; esi = ebp + 4
lea edx, [ebx+8*ecx+16]   ; edx = ebx+8*ecx+16
lea eax, [vec]            ; eax = address of global array vec
lea ecx, [vec+4*edx]      ; ecx = address of element of vec
```

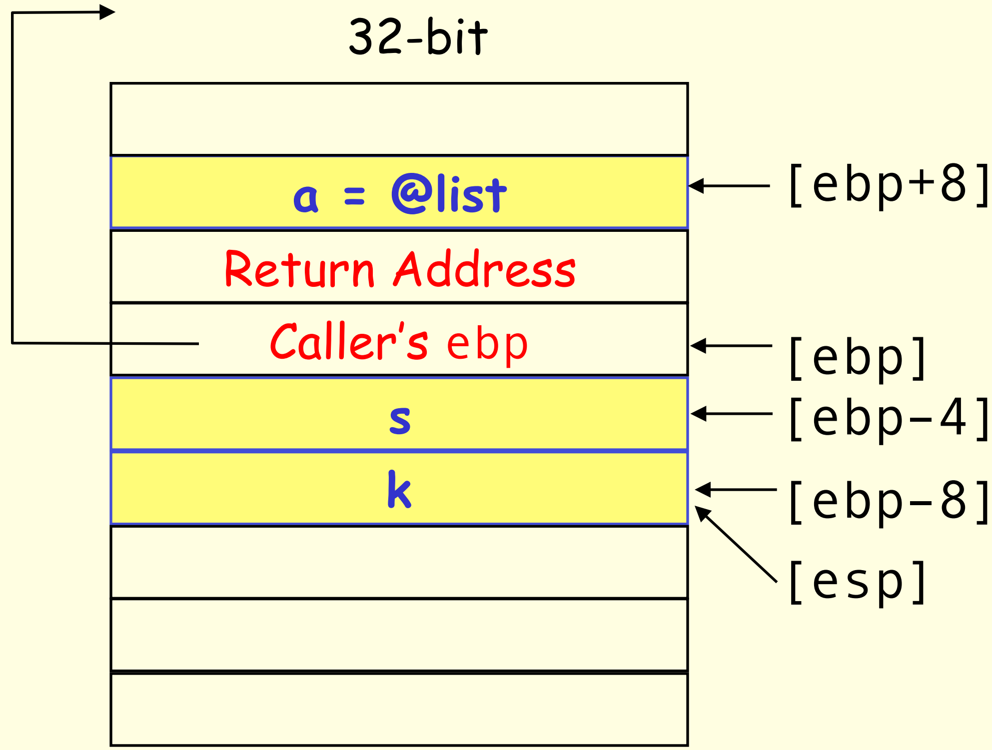
Example: Vector Sum (Caller)

```
int [4] list
int total
....
total = sum (list)
```

```
list      resd 4
total     resd 1
    ...
    push dword list   ; push @list
    call sum          ; call method
    add  esp, 4       ; remove param
    mov  [total],eax  ; assign result
```

Example: Vector Sum I

```
int sum (int [ ] a) {
    int s, k

    s = 0
    for (k=0; k<=3; k++) {
        s = s + a[k]
    }

    return s

}
```



Example: Vector Sum II

```
int sum (int [ ] a) {
   int s, k

   s = 0
   for (k=0; k<=3; k++) {
      s = s + a[k]
   }

   return s

}
```

```
sum:
    push ebp              ; method entry
    mov  ebp, esp         ; setup frameptr
    sub  esp, 8           ; space for s, k

    mov  dword[ebp-4], 0  ; s=0
forK:
    mov  dword[ebp-8], 0  ; k=0
nextK:
    cmp  dword[ebp-8], 3  ; compare k
    jg    endforK         ; end for if k>3
```

Example: Vector Sum III

```
int sum (int [ ] a) {
   int s, k

   s = 0
   for (k=0; k<=3; k++) {
        s = s + a[k]
   }

   return s

}
```

```
    mov ecx,[ebp-8]    ; ecx = k
    mov ebx,[ebp+8]   ; ebx = a = @list
    mov eax,[ebx+4*ecx]   ; eax=a[k]
    add [ebp-4],eax   ; s = s + a[k]

    inc dword[ebp-8]  ; k++
    jmp nextK          ; next iteration

endforK:
    mov eax,[ebp-4]    ; return value=s
    mov esp,ebp        ; restore esp
    pop ebp            ; restore ebp
    ret                ; return
```

Saving & Restoring Registers

- We must ensure that registers with current values are saved and restored across a method call since the called method may wish to use the same register(s). This **responsibility** is commonly left to the CALLED method (CALLEE)

- Example: If we use `edi` and `ecx` in a method we should push these registers on method-entry and pop them on method-exit:

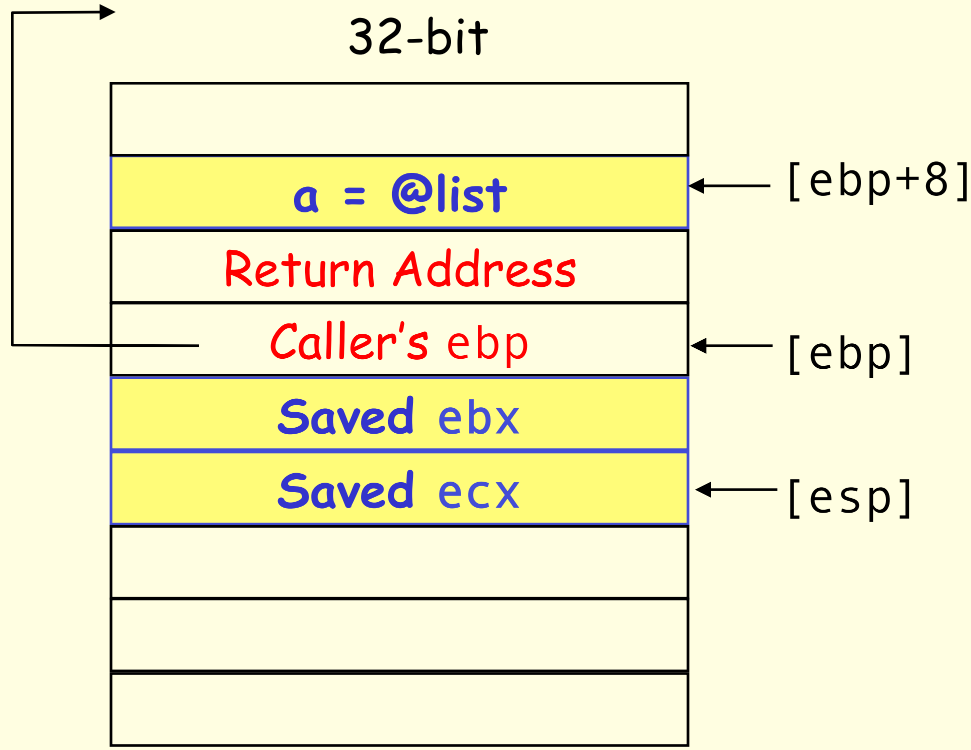
```
; Save Registers
push edi
push ecx
```

```
; Restore Registers
pop ecx
pop edi
```

- **Recall**: In **OUR** calling convention, `eax` will always be available for returning method results, hence it will be the caller's responsibility to ensure that `eax` does not hold any needed data on method-entry.

Vector Sum with Registers I

```
int sum (int [ ] a) {
    int s, k

    s = 0
    for (k=0; k<=3; k++) {
        s = s + a[k]
    }

    return s

}
```



Vector Sum with Registers II

```
int sum (int [ ] a) {
   int s, k

   s = 0
   for (k=0; k<=3; k++) {
      s = s + a[k]
   }

   return s

}
```

```
sum:
    push ebp            ; method entry
    mov  ebp,esp        ; setup frameptr
                        ; eax will hold s
    push ebx            ; ebx will hold a
    push ecx            ; ecx will hold k
    mov  ebx,[ebp+8]   ; a=@list
    mov  eax,0          ; s=0
forK:
    mov  ecx,0          ; k=0
nextK:
    cmp   ecx,3         ; test k
    jg    endforK       ; end for if k>3
```

Vector Sum with Registers III

```
int sum (int [ ] a) {
   int s, k

   s = 0
   for (k=0; k<=3; k++) {
       s = s + a[k]
   }

   return s

}
```

```
                 ; s = s + a[k]
    add eax,[ebx+4*ecx]

    inc ecx      ; k = k+1
    jmp nextK    ; next iteration

endforK:
    pop   ecx     ; restore ecx
    pop   ebx     ; restore ebx
    ; esp already points to old ebp

    pop   ebp     ; restore ebp
    ret           ; return
```

Classes & Objects

- The methods we've been writing to date do not operate on an object. Rather they assume the method is class-less. In object oriented languages, methods belong to classes and are typically invoked on objects of the class. Within the method the fields of the invoking object can also be accessed. To handle classes and object method calls, we'll extend our approach as follows:

- The fields of the object will be grouped together and allocated as a memory block, and allocated globally with data declaration directives. Note: In practice objects are allocated in a memory area reserved for dynamically created objects known as the **HEAP**. The Heap and memory management techniques for objects will be covered next year, in the Compilers course

- Class method names will be translated to a concatenation of the CLASS name and the METHOD name in assembly language.

- For object method calls we'll pass the address of the object as a hidden innermost parameter (parameter 0) and access the fields of the object indirectly via this hidden parameter.

Example: Object method call (1)

- The method *setpos* in:

```
class coord {
    int row; int col;
    void setpos (int x, int y) { row = x;  col = y; }
}
```

  is translated as if it was written without a class, e.g.:

```
void coord_setpos (coord this, int x, int y) {
    this.row = x;  this.col = y
}
```

- Then the call

```
coord point
point.setpos (3, 5)
```

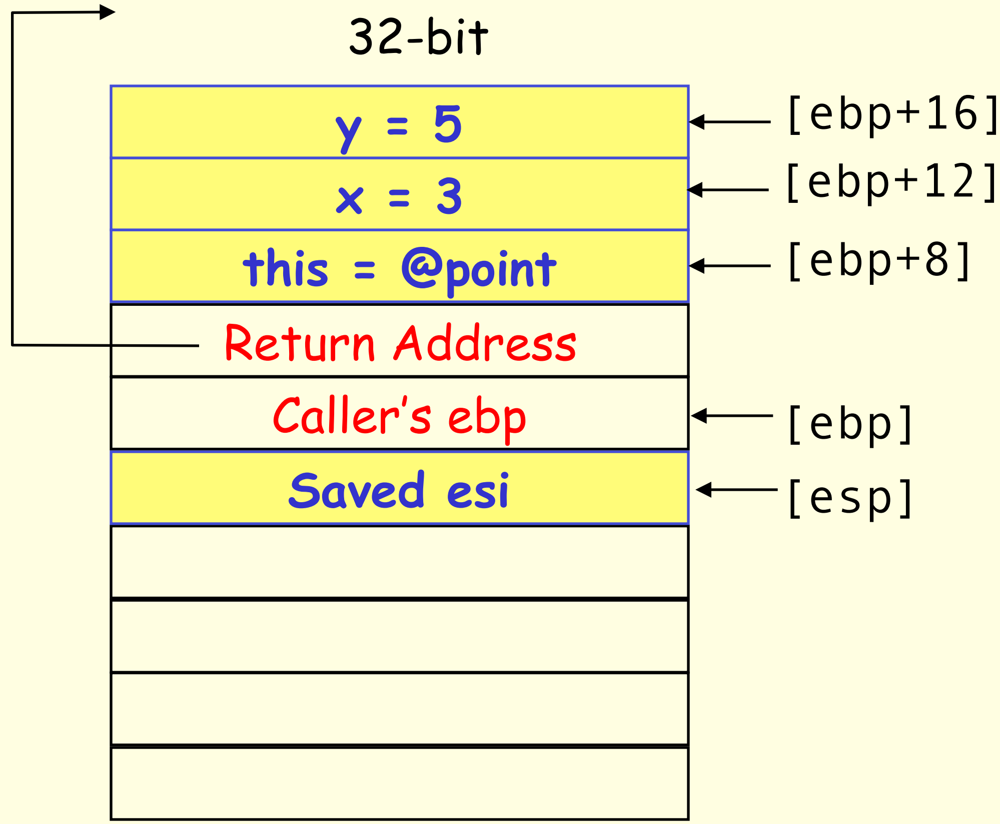
  is translated to:

```
coord_setpos (point,3, 5)
```

Example: Object method call (2)

coord point
 ...
point.*setpos* (3, 5)

```
; allocate point.row & col
point   resd 2
   ...
; call point.setpos
push dword 5   ; push 5
push dword 3   ; push 3
push dword point ; push
                 ; @point
call coord_setpos
add  esp, 12
```



Example: Object method call (3)

```
class coord {
    int row;
    int col;
    void setpos (int x, int y) {
        row = x;
        col = y;
    }
}
```

```
coord_setpos:
    push ebp            ; setup new frameptr
    mov  ebp,esp
    push esi            ; save esi
    mov  esi,[ebp+8]    ; esi = this

    mov  eax,[ebp+12]   ; eax = x
    mov  [esi], eax     ; this.row = x
    mov  eax,[ebp+16]   ; eax = y
    mov  [esi+4],eax    ; this.col = y

    pop  esi            ; restore esi
    pop  ebp            ; restore ebp
    ret                 ; return
```

That's all for now folks !