INTEL IA-32 ARCHITECTURE

Methods and the stack

Richard Hayden (with thanks to **Naranker Dulay**)
rh@doc.ic.ac.uk

http://www.doc.ic.ac.uk/~rh/teaching
or
https://www.doc.ic.ac.uk/~wl/teachlocal/arch1
or
CATE

Stacks

- Methods are usually implemented using a **stack**
- A region of memory accessed in a very specific manner
- The two basic stack operations are:

PUSH *data onto the **top** of the stack*
POP *data off of the **top** of the stack*

LIFO **"Last-in, first-out"**

Methods (a.k.a. sub-routines, functions, procedures)

- Provide the ability to **jump to the beginning** of a portion of code, **execute it** and then **return (possibly with a result)** back to where we were

We will also consider:

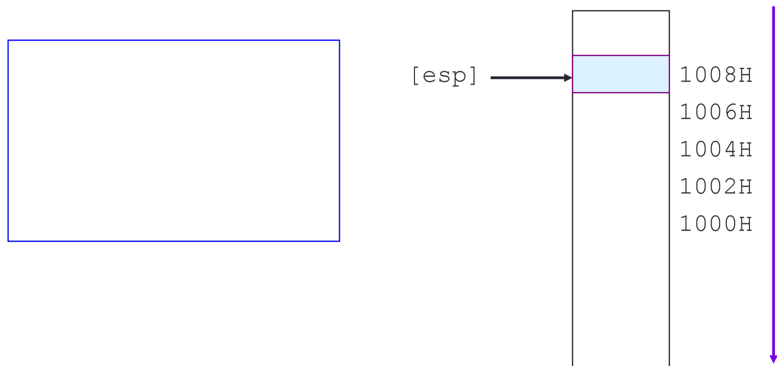
- Passing **parameters** to methods
- Allocating and accessing variables which are **local** to a method
- For object methods, accessing the **fields of the object**
- Making **nested** and **recursive** method calls

IA-32 system stack

- The IA-32 architecture **provides a framework** for managing the stack
- `esp` – the **stack pointer register** always **points to the top of the stack**
- We will also use the register `ebp` (**base pointer**) to help us **keep track of data on the stack**, including **local variables** and **parameters**
- **Note:** On the IA-32, **cannot push bytes** on to the stack, must work with **multiples of words**

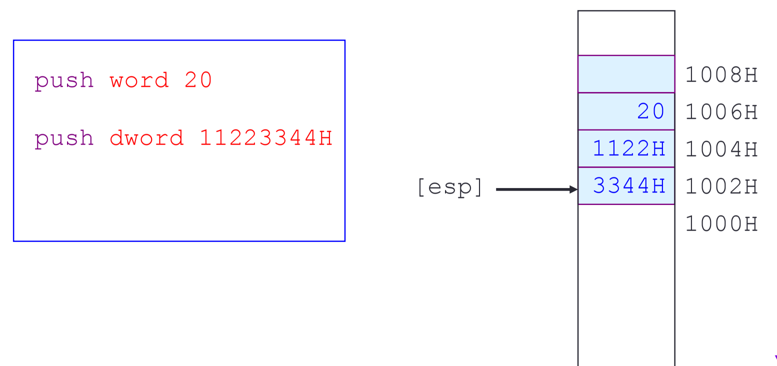
Example

- IA-32 stack **grows downwards**, that is, **higher addresses to lower addresses**



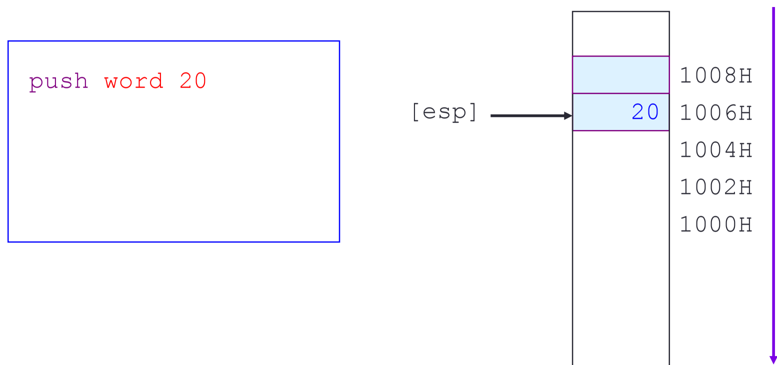
Example

- IA-32 stack **grows downwards**, that is, **higher addresses to lower addresses**



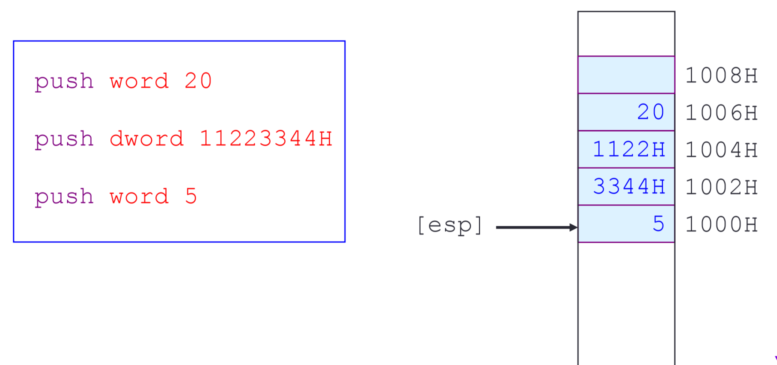
Example

- IA-32 stack **grows downwards**, that is, **higher addresses to lower addresses**



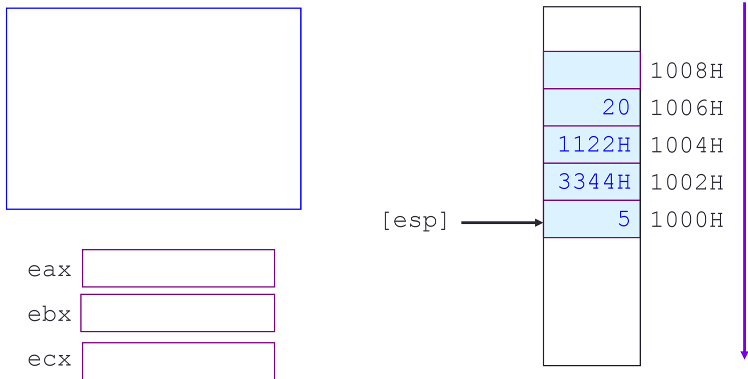
Example

- IA-32 stack **grows downwards**, that is, **higher addresses to lower addresses**



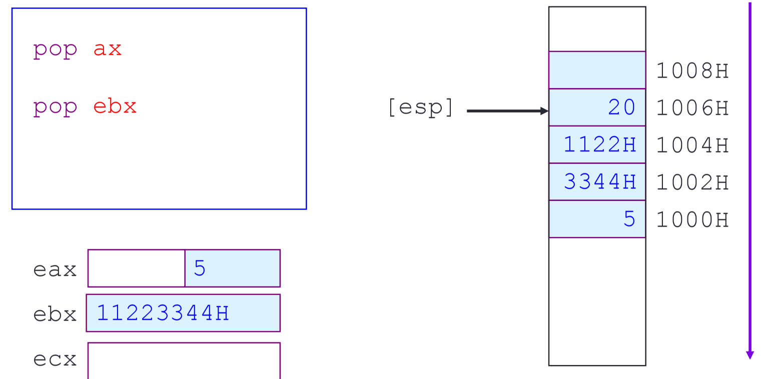
Example (cont.)

- IA-32 stack **shrinks upwards**, that is, **lower addresses to higher addresses**



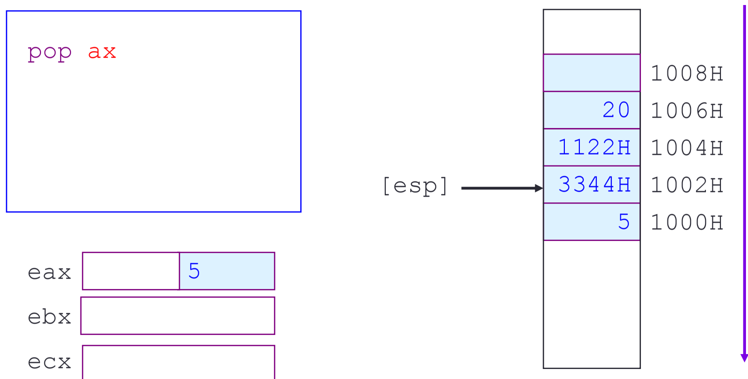
Example (cont.)

- IA-32 stack **shrinks upwards**, that is, **lower addresses to higher addresses**



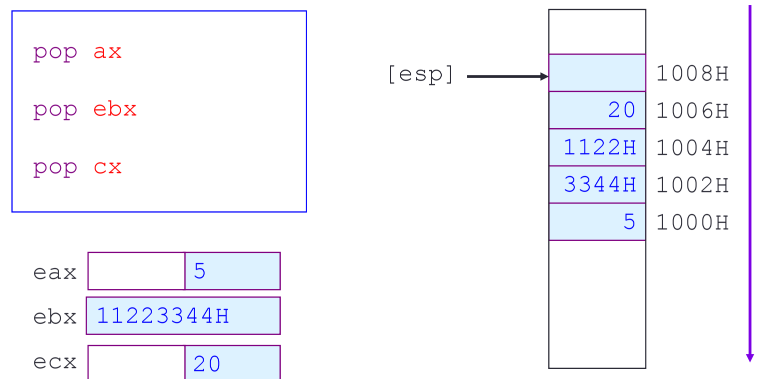
Example (cont.)

- IA-32 stack **shrinks upwards**, that is, **lower addresses to higher addresses**



Example (cont.)

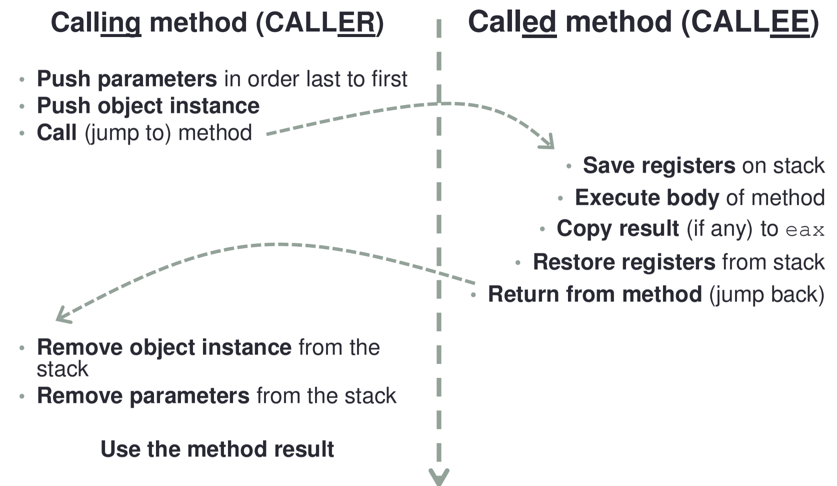
- IA-32 stack **shrinks upwards**, that is, **lower addresses to higher addresses**



IA-32 push and pop instructions

Instruction	Operation	Notes
push word_opr	esp = esp – 2 memory[esp] = word_opr	Push word onto stack
pop word_opr	word_opr = memory[esp] esp = esp + 2	Pop word off of stack
push dword_opr	esp = esp – 4 memory[esp] = dword_opr	Push dword onto stack
pop dword_opr	dword_opr = memory[esp] esp = esp + 4	Pop dword off of stack
pushfd	esp = esp – 4 memory[esp] = EFLAGS	Push EFLAGS onto stack
popfd	EFLAGS = memory[esp] esp = esp + 4	Pop EFLAGS off of stack

Calling convention



IA-32 call and ret instructions

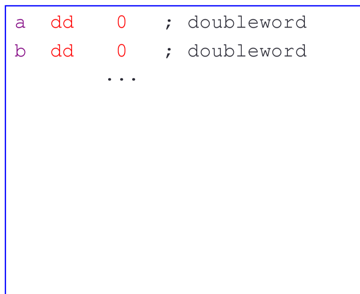
Instruction	Operation	Notes
call method	push eip jmp method	Push return address and then jump to method code
ret	pop eip	Pop return address into eip (thus jumping back)

- **IMPORTANT:** Since `eip` is **incremented at the beginning of the fetch-execute cycle**, the address pushed by `call` will be that of the **next instruction** to resume execution after the called method has finished

Example (caller)

```
int a, b;
// We'll use 32-bit integers
....
a = max(a, b);
```

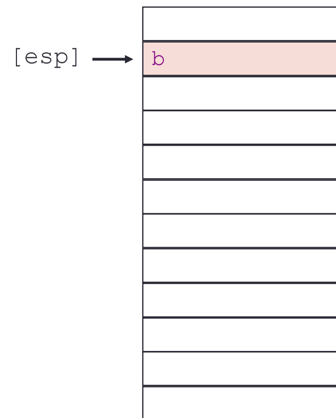
```
a   dd    0    ; doubleword
b   dd    0    ; doubleword
      ...
```



Example (caller)

```
int a, b;
// We'll use 32-bit integers
....
a = max(a, b);
```

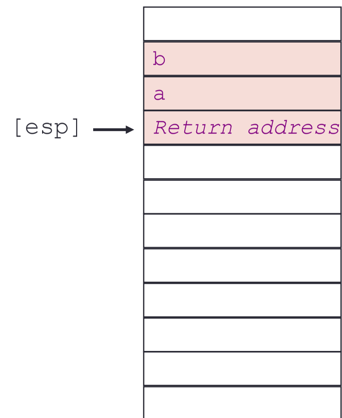
```
a  dd   0   ; doubleword
b  dd   0   ; doubleword
       ...
   push dword [b]
```



Example (caller)

```
int a, b;
// We'll use 32-bit integers
....
a = max(a, b);
```

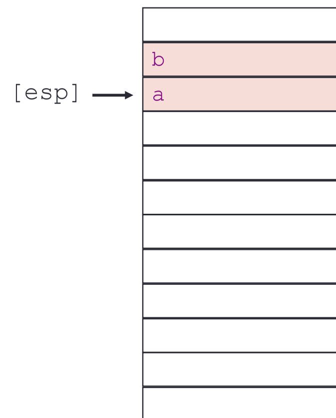
```
a  dd   0   ; doubleword
b  dd   0   ; doubleword
       ...
   push dword [b]
   push dword [a]
   call max
```



Example (caller)

```
int a, b;
// We'll use 32-bit integers
....
a = max(a, b);
```

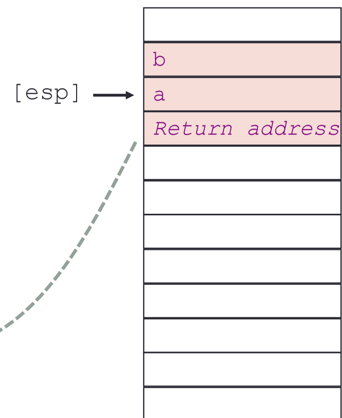
```
a  dd   0   ; doubleword
b  dd   0   ; doubleword
       ...
   push dword [b]
   push dword [a]
```



Example (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
```



Example (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
```

[esp] → (top of stack)

Stack
b
a
Return address

Example (callee)

```
int max (int x, int y) {
    if (x < y) {
        return y;
    } else {
        return x;
    }
}
```

```
max:
```

	Stack
[esp+8] →	y = b
[esp+4] →	x = a
[esp] →	*Return address*

Example (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
```

[esp] → (top of stack)

Stack
b
a
Return address

Example (callee)

```
int max (int x, int y) {
    if (x < y) {
        return y;
    } else {
        return x;
    }
}
```

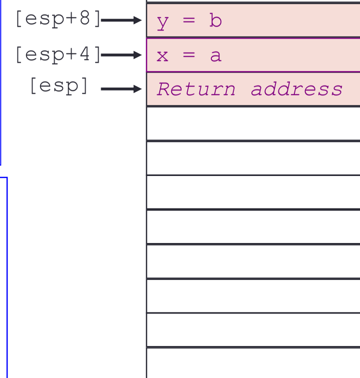
```
max:
    mov eax, [esp+4]   ; eax = x
```

	Stack
[esp+8] →	y = b
[esp+4] →	x = a
[esp] →	*Return address*

Example (callee)

```
int max (int x, int y) {
    if (x < y) {
        return y;
    } else {
        return x;
    }
}
```

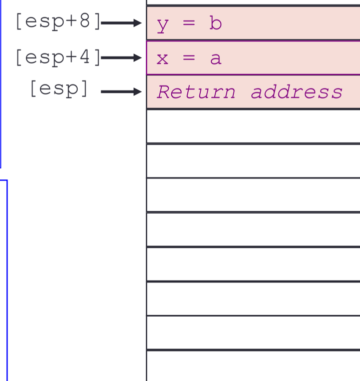
```
max:
    mov eax, [esp+4]   ; eax = x
    cmp eax, [esp+8]   ; is x >= y?
```



Example (callee)

```
int max (int x, int y) {
    if (x < y) {
        return y;
    } else {
        return x;
    }
}
```

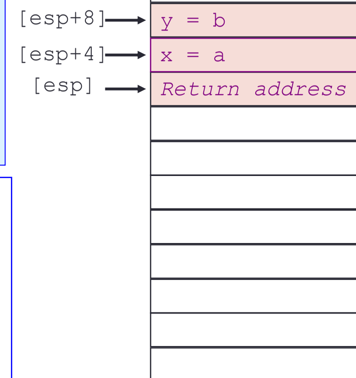
```
max:
    mov eax, [esp+4]   ; eax = x
    cmp eax, [esp+8]   ; is x >= y?
    jge endmax

endmax:
```



Example (callee)

```
int max (int x, int y) {
    if (x < y) {
        return y;
    } else {
        return x;
    }
}
```

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



Local variables

- "Lifetime" of local variables is **limited to the execution of the method** they are declared in
- Can use **registers** or **allocate space on the stack** for them – or a combination of both
- For convenience, we use the **base pointer register** `ebp` to point in between parameters and stack local variables
- In this context, `ebp` is known as the **frame pointer**

Handling stack local variables

- The **callee** handles making space on the stack for local variables
- **Entry:** setup frame pointer and allocate space for local variables

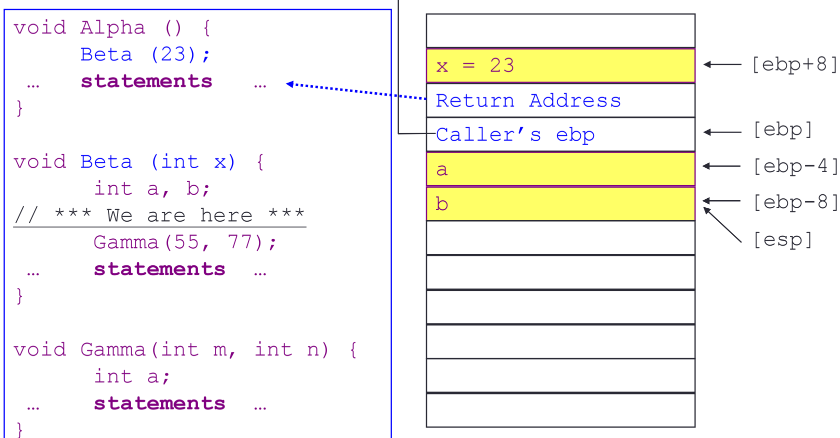
```
push ebp                 ; save caller's frame pointer on the stack
mov  ebp, esp            ; set frame pointer for called method

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

- **Exit:** de-allocate stack local variables and restore frame pointer

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

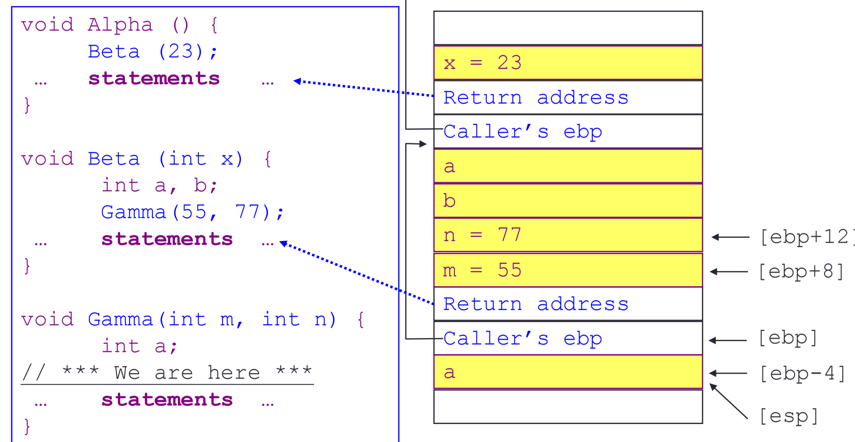
Stack frame example

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

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

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



Stack frame example (cont.)

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

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

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



Array and object parameters

- For array and object parameters, we push the **start address** rather than its value. We then access the array/object by reference using this address rather than a copy of it.
- Can compute this address using the **load effective address** instruction:

  `lea     dstreg, [BaseReg + Scale * IndexReg + Disp]`

- `lea` only **computes** the address – **does not access** the memory location

```
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
```

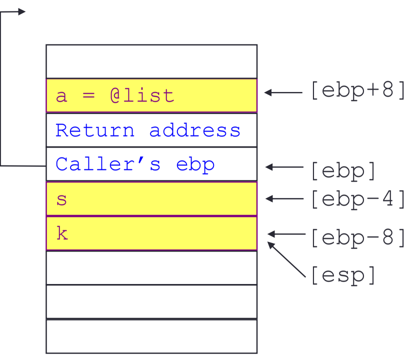
Example: vector sum (callee) (2)

```
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 frame ptr.
      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
      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
```

Example: vector sum (callee) (1)

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

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

  return s;
}
```



Saving and restoring registers

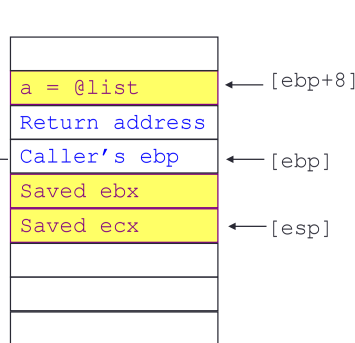
- We must ensure that any registers used by a method are **saved and restored** across a method call. This responsibility is usually left to the **callee** (called) method
- **Example:** suppose we use the two registers `edi` and `ecx` in a method:

```
; Save registers              ; Restore registers
push    edi                   pop    ecx
push    ecx                   pop    edi
```

- **Note:** in the calling convention we're using, `eax` will always be available for returning method results – **is caller's responsibility to ensure it does not hold any valuable data**

Example: vector sum with registers (callee) (1)

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

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

  return s;
}
```



Classes and objects

- So far methods are **class-less** and do not operate on an object
- In OO languages, **methods typically belong to classes** and **act on the fields of objects** of that class
- We will **group the fields of objects together** and **allocate as one memory block.** In practise, they are **allocated on the heap**
- Class method names will be translated to a concatenation of the **class name** and the **method name**
- For class method calls, we'll **pass the address of the object** as a **hidden parameter (parameter 0)** and access the fields indirectly via this parameter

Example: vector sum with registers (callee) (2)

```
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 frame ptr.
                                  ; 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                 ; compare k
      jg   endforK                ; end for if k>3
      add  eax, [ebx+4*ecx]       ; s = s + a[k]

      inc  ecx                    ; k++
      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
```

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:

```
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
```

y = 5
x = 3
this = @point
Return address

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
```

Stack	Address
y = 5	[ebp+16]
x = 3	[ebp+12]
this = @point	[ebp+8]
Return address	
Caller's ebp	[ebp]
Saved esi	[esp]