What does CPU do internally?
( Remember Lecture 01 - Introduction?)
CPU execution loop

- CPU repeatedly reads instructions from memory
- Executes them

Example

```
ADD EDX, EAX, EBX
// EDX = EAX + EBX
```
Stack and procedure calls
What is stack?
Stack

• It's just a region of memory
  • Pointed by a special register ESP
• You can change ESP
  • Get a new stack
Why do we need stack?
Calling functions

// some code...
foo();
// more code..

- Stack contains information for how to return from a subroutine
  - i.e., foo()
Stack

- **Main purpose:**
  - Store the return address for the current procedure
  - Caller pushes return address on the stack
  - Callee pops it and jumps
Stack

- Main purpose:
  - Store the return address for the current procedure
  - Caller pushes return address on the stack
  - Callee pops it and jumps
Stack

- Other uses:
  - Local data storage
  - Parameter passing
  - Evaluation stack
    - Register spill
Call/return

• CALL instruction
  • Makes an unconditional jump to a subprogram and pushes the address of the next instruction on the stack

  ```
  push eip + sizeof(CALL); save return
  jmp _my_function
  ```

• RET instruction
  • Pops off an address and jumps to that address
Manipulating stack

- ESP register
  - Contains the memory address of the topmost element in the stack

- PUSH instruction
  - `push 0xBAR`
  - Insert data on the stack
  - Subtract 4 from ESP
Manipulating stack

- **POP instruction**
  
  \[ \text{pop EAX} \]

  - Removes data from the stack
  - Saves in register or memory
  - Adds 4 to ESP
Calling conventions
Calling conventions

- Goal: reentrant programs
  - How to pass arguments
    - On the stack?
    - In registers?
  - How to return values
    - On the stack?
    - In registers?
- Conventions differ from compiler, optimizations, etc.
Maintain stack as frames

- Each function has a new frame

```c
void DrawSquare(...) {
    ...
    DrawLine(x, y, z);
}
```

- Use dedicated register EBP (frame pointer)
  - Points to the base of the frame
Maintain stack as frames

- Each function has a new frame

```c
void DrawSquare(...) {
    ...
    DrawLine(x, y, z);
}
```

- Use dedicated register **EBP** (frame pointer)
  - Points to the base of the frame
Stack consists of frames

- Each function has a new frame

```c
void DrawSquare(...) {
    ...  
    DrawLine(x, y, z);
}
```

- Use dedicated register EBP (frame pointer)
  - Points to the base of the frame
Prologue/epilogue

• Each function maintains the frame
  • A dedicated register EBP is used to keep the frame pointer
  • Each function uses prologue code (blue), and epilogue (yellow) to maintain the frame

my_function:
  push ebp ; save original EBP value on stack
  mov ebp, esp ; new EBP = ESP
  .... ; function body
  pop ebp ; restore original EBP value
  ret
How to allocate local variables

• Each function has private instances of local variables

```c
foo(int x) {
    int a, b, c;
    ...
    return;
}
```

• Function can be called recursively

```c
foo(int x) {
    int a, b, c;
    a = x + 1;
    if (a < 100)
        foo(a);
    return;
}
```
How to allocate local variables?

void my_function()
{
    int a, b, c;
    ...
}

void my_function()  
{
    int a, b, c;
    ...
}

• On the stack!
Allocating local variables

- Stored right after the saved EBP value in the stack
- Allocated by subtracting the number of bytes required from ESP

```assembly
_my_function:
push ebp           ; save original EBP value on stack
mov ebp, esp       ; new EBP = ESP
sub esp, LOCAL_BYTES ; = # bytes needed by locals
...               ; function body
mov esp, ebp       ; deallocate locals
pop ebp            ; restore original EBP value
ret
```
void my_function() {
    int a, b, c;
    ...

=my_function:
    push ebp            ; save the value of ebp
    mov ebp, esp        ; ebp = esp, set ebp to be top of the stack (esp)
    sub esp, 12         ; move esp down to allocate space for the
                         ; local variables on the stack

• With frames local variables can be accessed by dereferencing EBP

    mov [ebp - 4], 10     ; location of variable a
    mov [ebp - 8], 5      ; location of b
    mov [ebp - 12], 2     ; location of c
Example

```c
void my_function() {
    int a, b, c;
    ...

    _my_function:
    push ebp          ; save the value of ebp
    mov ebp, esp      ; ebp = esp, set ebp to be top of the stack (esp)
    sub esp, 12       ; move esp down to allocate space for the
                      ; local variables on the stack

    • With frames local variables can be accessed by dereferencing EBP

    mov [ebp -  4], 10 ; location of variable a
    mov [ebp -  8], 5  ; location of b
    mov [ebp - 12], 2  ; location of c
```
Example

void my_function() {
    int a, b, c;
    ...

    _my_function:
    push ebp          ; save the value of ebp
    mov ebp, esp      ; ebp = esp, set ebp to be top of the stack (esp)
    sub esp, 12       ; move esp down to allocate space for the
                      ; local variables on the stack

    ● With frames local variables can be accessed by dereferencing EBP

    mov [ebp -  4], 10  ; location of variable a
    mov [ebp -  8], 5   ; location of b
    mov [ebp - 12], 2   ; location of c
Example

```c
void my_function() {
    int a, b, c;
    ...

    _my_function:
    push ebp          ; save the value of ebp
    mov ebp, esp      ; ebp = esp, set ebp to be top of the stack (esp)
    sub esp, 12       ; move esp down to allocate space for the
                       ; local variables on the stack

    • With frames local variables can be accessed by dereferencing EBP

    mov [ebp -  4], 10  ; location of variable a
    mov [ebp -  8], 5   ; location of b
    mov [ebp - 12], 2   ; location of c
```
How to pass arguments?

- Possible options:
  - In registers
  - On the stack
How to pass arguments?

• x86 32 bit
  • Pass arguments on the stack
  • Return value is in EAX and EDX

• x86 64 bit – more registers!
  • Pass first 6 arguments in registers
    – RDI, RSI, RDX, RCX, R8, and R9
  • The rest on the stack
  • Return value is in RAX and RDX
x86_32: passing arguments on the stack

• Example function

```c
void my_function(int x, int y, int z)
{
    ...
}
```

• Example invocation

```c
my_function(2, 5, 10);
```

• Generated code

```assembly
push 10
push 5
push 2
call _my_function
```
Example stack

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>[ebp + 16] (3rd function argument)</td>
</tr>
<tr>
<td>5</td>
<td>[ebp + 12] (2nd argument)</td>
</tr>
<tr>
<td>2</td>
<td>[ebp + 8] (1st argument)</td>
</tr>
<tr>
<td>RA</td>
<td>[ebp + 4] (return address)</td>
</tr>
<tr>
<td>FP</td>
<td>[ebp] (old ebp value) ← EBP points here</td>
</tr>
<tr>
<td></td>
<td>[ebp - 4] (1st local variable)</td>
</tr>
<tr>
<td></td>
<td>[ebp - X] (esp - the current stack pointer)</td>
</tr>
</tbody>
</table>
Example stack

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>[ebp + 16] (3rd function argument)</td>
</tr>
<tr>
<td>5</td>
<td>[ebp + 12] (2nd argument)</td>
</tr>
<tr>
<td>2</td>
<td>[ebp + 8] (1st argument)</td>
</tr>
<tr>
<td>RA</td>
<td>[ebp + 4] (return address)</td>
</tr>
<tr>
<td>FP</td>
<td>[ebp] (old ebp value) ← EBP points here</td>
</tr>
<tr>
<td></td>
<td>[ebp - 4] (1st local variable)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[ebp - X] (esp – the current stack pointer)</td>
</tr>
</tbody>
</table>
Example stack

| 10 | [ebp + 16] (3rd function argument) |
|  5 | [ebp + 12] (2nd argument) |
|  2 | [ebp + 8] (1st argument) |
| RA | [ebp + 4] (return address) |
| FP | [ebp] (old ebp value) ← EBP points here |
|    | [ebp - 4] (1st local variable) |

| [ebp - X] (esp - the current stack pointer) |
Example stack

:  :  :
| 10 | [ebp + 16] (3rd function argument) |
|  5 | [ebp + 12] (2nd argument)          |
|  2 | [ebp + 8]  (1st argument)          |
| RA | [ebp + 4]  (return address)        |
| FP | [ebp]      (old ebp value) ← EBP points here |
|    | [ebp - 4]  (1st local variable)    |
:  :  :
|    | [ebp - X]  (esp - the current stack pointer) |
Example: caller side code

```c
int callee(int, int, int);

int caller(void)
{
    int ret;
    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

caller:

```
; manage own stack frame
push    ebp
mov     ebp, esp

; push call arguments
push    3
push    2
push    1
; call subroutine 'callee'
call    callee

; remove arguments from frame
add     esp, 12
; use subroutine result
add     eax, 5
; restore old call frame
pop     ebp
; return
ret
```
```c
Example: caller side code

int callee(int, int, int);

int caller(void)
{
    int ret;
    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}

caller:
    ; manage own stack frame
    push ebp
    mov ebp, esp
    ; push call arguments
    push 3
    push 2
    push 1
    ; call subroutine 'callee'
    call callee
    ; remove arguments from frame
    add esp, 12
    ; use subroutine result
    add eax, 5
    ; restore old call frame
    pop ebp
    ; return
    ret
```
int callee(int, int, int);

int caller(void)
{
    int ret;
    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
### Example: caller side code

```c
int callee(int, int, int);

int caller(void)
{
    int ret;
    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

```assembly
caller:
    ; manage own stack frame
    push    ebp
    mov     ebp, esp
    ; push call arguments
    push    3
    push    2
    push    1
    ; call subroutine 'callee'
    call callee
    ; remove arguments from frame
    add     esp, 12
    ; use subroutine result
    add     eax, 5
    ; restore old call frame
    pop     ebp
    ; return
    ret
```
Example: caller side code

```c
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

caller:
```
    ; manage own stack frame
    push    ebp
    mov     ebp, esp
    ; push call arguments
    push    3
    push    2
    push    1
    ; call subroutine 'callee'
call callee
    ; remove arguments from frame
    add     esp, 12
    ; use subroutine result
    add     eax, 5
    ; restore old call frame
    pop     ebp
    ; return
    ret
```
Example: caller side code

```c
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

caller:

; manage own stack frame
push    ebp
mov     ebp, esp
; push call arguments
push    3
push    2
push    1
; call subroutine 'callee'
call    callee
; remove arguments from frame
add     esp, 12
; use subroutine result
add     eax, 5
; restore old call frame
pop     ebp
; return
ret
Wait, where is “return ret;”?

```c
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

caller:

```assembly
; manage own stack frame
push    ebp
mov     ebp, esp
; push call arguments
push    3
push    2
push    1
; call subroutine 'callee'
call    callee
; remove arguments from frame
add     esp, 12
; use subroutine result
add     eax, 5
; restore old call frame
pop     ebp
; return
ret
```
Example: callee
side code

```c
void my_function(int x, int y, int z)
{
    int a, b, c;
    ...
    return;
}

_my_function:
    push ebp
    mov ebp, esp
    sub esp, 12 ; allocate local variables
                ; sizeof(a) + sizeof(b) + sizeof(c)
                ; x = [ebp + 8], y = [ebp + 12], z = [ebp + 16]
    mov [ebp-4], [ebp-8], [ebp-12] = [esp+8],
    ; b=[ebp-8]=[esp+4], c=[ebp-12] = [esp]
    mov esp, ebp ; deallocate local variables
    pop ebp
    ret
```
Example: callee side code

```c
void my_function(int x, int y, int z) {
    int a, b, c;
    ...
    return;
}
```

```assembly
_my_function:
push ebp
mov ebp, esp
sub esp, 12 ; allocate local vars
            ; sizeof(a) + sizeof(b) + sizeof(c)
            ; x = [ebp + 8], y = [ebp + 12], z = [ebp + 16]
            ; a=[ebp-4]=[esp+8],
            ; b=[ebp-8]=[esp+4], c=[ebp-12] = [esp]
mov esp, ebp ; deallocate local vars
pop ebp
ret
```
Example: callee

side code

```c
void my_function(int x, int y, int z)
{
    int a, b, c;
    ...
    return;
}

_my_function:
    push ebp
    mov ebp, esp ; ebp = esp
    sub esp, 12 ; allocate local variables
        ; sizeof(a) + sizeof(b) + sizeof(c)
    ; x = [ebp + 8], y = [ebp + 12], z = [ebp + 16]
    ; a=[ebp-4]=[esp+8],
    ; b=[ebp-8]=[esp+4], c=[ebp-12] = [esp]
    mov esp, ebp ; deallocate local variables (esp = ebp)
    pop ebp
    ret
```
int a, b, c;
...
return;

void my_function(int x, int y, int z) {
  
_my_function:
  push ebp
  mov ebp, esp ; ebp = esp
  sub esp, 12 ; allocate local variables
              ; sizeof(a) + sizeof(b) + sizeof(c)
              ; x = [ebp + 8], y = [ebp + 12], z = [ebp + 16]
              ; a=[ebp-4]=[esp+8],
              ; b=[ebp-8]=[esp+4], c=[ebp-12] = [esp]
  mov esp, ebp ;deallocate local variables (esp = ebp)
  pop ebp
  ret
Example: callee side code

```c
void my_function(int x, int y, int z)
{
    int a, b, c;
    ...
    return;
}
```

```assembly
_my_function:
push ebp
mov ebp, esp ; ebp = esp
sub esp, 12 ; allocate local variables
            ; sizeof(a) + sizeof(b) + sizeof(c)
            ; x = [ebp + 8], y = [ebp + 12], z = [ebp + 16]
            ; a=[ebp-4]=[esp+8],
            ; b=[ebp-8]=[esp+4], c=[ebp-12] = [esp]
mov esp, ebp ;deallocate local variables (esp = ebp)
pop ebp
ret
```
void my_function(int x, int y, int z)
{
    int a, b, c;
    ...
    return;
}

_my_function:
    push ebp
    mov ebp, esp ; ebp = esp
    sub esp, 12 ; allocate local varaibles
                ; sizeof(a) + sizeof(b) + sizeof(c)
                ; x = [ebp + 8], y = [ebp + 12], z = [ebp + 16]
                ; a=[ebp-4]=[esp+8],
                ; b=[ebp-8]=[esp+4], c=[ebp-12] = [esp]
    mov esp, ebp
    pop ebp
    ret

- x86 has a special instruction for this
  - leave
Back to stack frames, so why do we need them?

- ... They are not strictly required
- GCC compiler option `-fomit-frame-pointer` can disable them

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. It also makes debugging impossible on some machines.
Referencing args without frames

Initially parameter is
- \([\text{ESP} + 4]\)

Later as the function pushes things on the stack it changes, e.g.
- \([\text{ESP} + 8]\)
• Debugging becomes hard
  • As ESP changes one has to manually keep track where local variables are relative to ESP (ESP + 4 or +8)
    – Compiler can easily do this and generate correct code!
    – But it's hard for a human
  • It's hard to unwind the stack in case of a crash
    – To print out a backtrace
And you only save...

- A couple instructions required to maintain the stack frame
- And 1 register (EBP)
  - x32 has 8 registers (and one is ESP)
    - So taking another one is 12.5% of register space
    - Sometimes its worse it!
  - x64 has 16 registers, so it doesn't really matter
- That said, GCC sets `-fomit-frame-pointer` to “on”
  - At -O, -O1, -O2 ...
  - Don't get surprised
With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

- **-O** turns on the following optimization flags:

  - -fauto-inc-dec
  - -fbranch-count-reg
  - ...
  - -fomit-frame-pointer
  - -freorder-blocks
Saving and restoring registers
Saving register state across invocations

- Processor doesn't save registers
  - General purpose, segment, flags
- Again, a calling convention is needed
  - Agreement on what gets saved by the callee and the caller
Saving register state across invocations

- Registers EAX, ECX, and EDX are caller-saved
  - The function is free to use them
- ... the rest are callee-saved
  - If the function uses them it has to restore them to the original values
• In general there are multiple calling conventions
  • We described cdecl
  • Make sure you know what you're doing
  • https://en.wikipedia.org/wiki/X86_calling_conventions#List_of_x86_calling_conventions
  • It's easy as long as you know how to read the table
Questions?
References

- https://en.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames
- https://en.wikipedia.org/wiki/Calling_convention