

## Microprocessor Design Trends

- Joy's Law [Bill Joy of BSD4.x and Sun fame]

$$\text{MIPS} = 2^{\text{year}-1984}$$

- Millions of instructions per second [MIPS] executed by a single chip microprocessor
- More realistic rate is a doubling of MIPS every 18 months [or a quadrupling every 3 years]
- What ideas and techniques in new microprocessor designs have contributed to this continued rate of improvement?

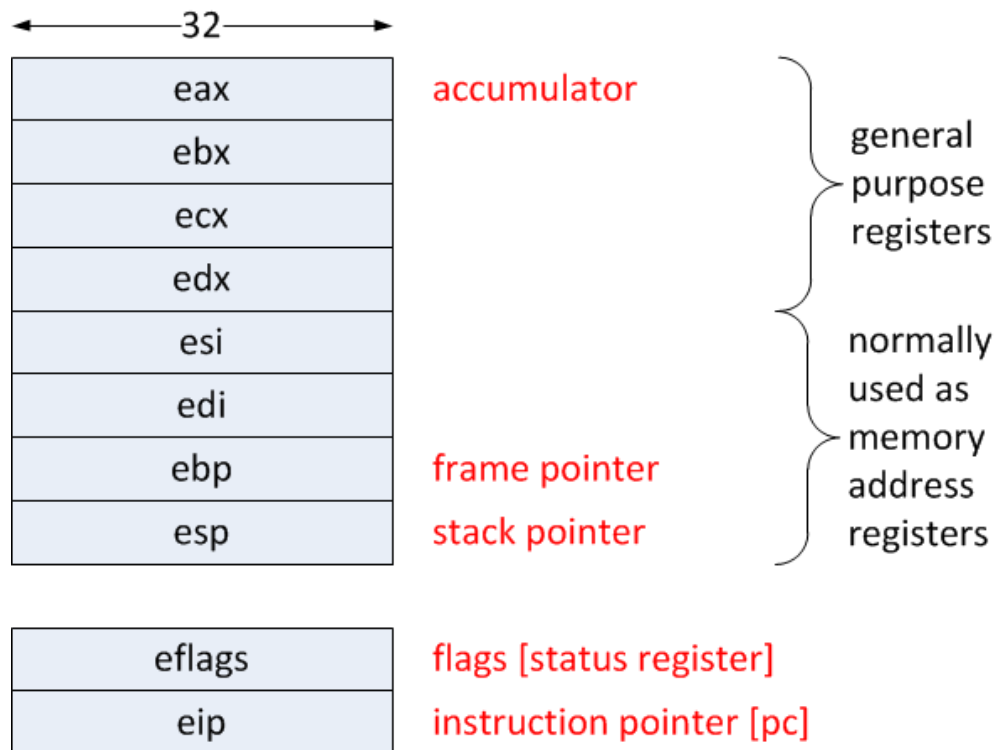
## Some of improvements made over the last 40 years ...

- smaller VLSI feature sizes [**1 micron ( $\mu$ ) ... 7nm**]
- increased clock rate [**1MHz ... 4GHz**]
- reduced vs complex instruction sets [**RISC vs CISC**]
- faster memory access modes (eg burst accesses)
- integrated on-chip MMUs, FPU, ...
- pipelining
- superscalar [**multiple instructions/clock cycle**]
- multi-level on-chip instruction and data caches
- streaming SIMD [**single instruction multiple data**] instruction extensions [**MMX, SSE**]
- hyper threading, multi-core and multiprocessor support
- direct programming of graphics co-processor
- high speed point to point interconnect [**Intel QuickPath, AMD HyperTransport**]
- solid state disks
- ...

## IA32 [Intel Architecture 32 bit]

- IA32 first released in 1985 with the 80386 microprocessor
- IA32 still used today by current Intel CPUs
- modern Intel CPUs have many additions to the original IA32 including MMX, SSE1, SSE2, SSE3, SSE4, SSE5, AVX, AVX2 and AVX512 [Streaming SIMD Extensions] and an extended 64 bit instruction set when operating in 64 bit mode [named IA-32e or IA-32e or x64]
- 32 bit CPU [performs 8, 16 and 32 bit integer + 32 and 64 bit floating point arithmetic]
- 32 bit virtual and physical address space  $2^{32}$  bytes [4GB]
- each instruction a multiple of bytes in length [from 1 to 17+]

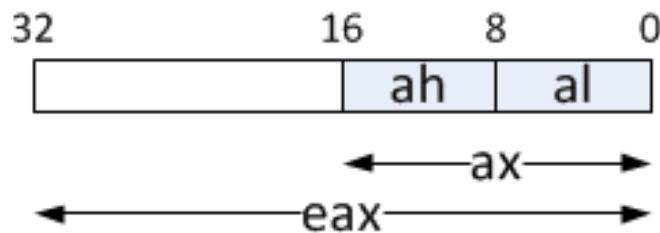
## Registers [far fewer than a typical RISC]



NB: floating point and SSE registers, ... not shown

## Registers...

- "e" in eax = extended = 32bits



- possible to access 8 and 16 bit parts of eax, ebx, ecx and edx using alternate register names ah, al and ax

## Instruction Format

- two address [**will use Microsoft assembly language syntax used by VC++, MASM**]

add     ←     eax, ebx                    ; eax = eax + ebx [**right to left**]

- alternative gnu syntax*

addl     →     %ebx, %eax                ; eax = eax + ebx [**left to right**]

- two operands normally

←  
register/register  
register/immediate  
register/memory  
memory/register

- memory/memory and memory/immediate are NOT allowed

## Supported Addressing Modes

$[a]$  = contents of  
memory address  $a$

| <i>addressing mode</i> | <i>example</i>  |  |
|------------------------|---|--|
| immediate              | <code>mov eax, <math>n</math></code>                                | $\text{eax} = n$                                 |
| register               | <code>mov eax, ebx</code>   | $\text{eax} = \text{ebx}$                        |
| direct/absolute        | <code>mov eax, <math>[a]</math></code>                              | $\text{eax} = [a]$                               |
| indexed                | <code>mov eax, [ebx]</code>   | $\text{eax} = [\text{ebx}]$                      |
| indexed                | <code>mov eax, [ebx + <math>n</math>]</code>                        | $\text{eax} = [\text{ebx} + n]$                  |
| scaled indexed         | <code>mov eax, [ebx * <math>s</math> + <math>n</math>]</code>       | $\text{eax} = [\text{ebx} * s + n]$              |
| scaled indexed         | <code>mov eax, [ebx + ecx]</code>                                   | $\text{eax} = [\text{ebx} + \text{ecx}]$         |
| scaled indexed         | <code>mov eax, [ebx + ecx * <math>s</math> + <math>n</math>]</code> | $\text{eax} = [\text{ebx} + \text{ecx} * s + n]$ |

- address computed as the sum of a register, a scaled register and a 1, 2 or 4 byte signed constant  $n$ ; can use most registers
- scaled indexed addressing used to index into arrays
- scaling constant  $s$  can be 1, 2, 4 or 8

# IA32 AND X64

## IA32 basic instruction set

|      |                            |
|------|----------------------------|
| mov  | move                       |
| xchg | exchange                   |
| add  | add                        |
| sub  | subtract                   |
| cdq  | convert double to quadword |
| idiv | unsigned divide            |
| imul | signed multiply            |
| inc  | increment by 1             |
| dec  | decrement by 1             |
| neg  | negate                     |
| cmp  | compare                    |
| lea  | load effective address     |
| test | AND operands and set flags |

|     |              |
|-----|--------------|
| and | and          |
| or  | or           |
| xor | exclusive or |
| not | not          |

|                         |                        |
|-------------------------|------------------------|
| push                    | push onto stack        |
| pop                     | pop from stack         |
| sar                     | shift arithmetic right |
| shl                     | shift logical left     |
| shr                     | shift logical right    |
| jmp                     | unconditional jump     |
| j {e, ne, l, le, g, ge} | signed jump            |
| j {b, be, a, ae}        | unsigned jump          |
| call                    | call subroutine        |
| ret                     | return from subroutine |

- SHOULD BE ENOUGH INSTRUCTIONS TO COMPLETE TUTORIALS
- Google [Intel® 64 and IA-32 Architectures Software Developer's Manual 2A, 2B, 2C](#) for details



## IA32 Assembly Language examples

- size of operation can often be determined implicitly by MASM, but when unable to do so, size needs to be specified explicitly

```
mov    eax, [ebp+8]           ; implicitly 32 bit [as eax is 32 bits]
```

```
mov    ah, [ebp+8]           ; implicitly 8 bit [as ah is 8 bits]
```

```
dec    [ebp+8]               ; decrement memory location [ebp+8] by 1  
                                           ; assembler unable to determine operand size  
                                           ; is it an 8, 16 or 32 bit value??
```

```
dec    DWORD PTR [ebp+8]     ; make explicitly 32 bit
```

```
dec    WORD PTR [ebp+8]      ; make explicitly 16 bit
```

```
dec    BYTE PTR [ebp+8]      ; make explicitly 8 bit
```

NB: unusual assembly language syntax

## IA32 Assembly Language examples ...

- memory/immediate operations NOT allowed

~~mov [ebp+8], 123~~ ; NOT allowed and operation size ALSO unknown

mov eax, 123 ; use 2 instructions instead...  
mov [ebp+8], eax ; implicitly 32 bits

- lea [**load effective address**] is a useful instruction for performing simple arithmetic

lea eax, [ebx+ecx\*4+16] ;  $eax = ebx + ecx * 4 + 16$

- does the effective address calculation, but doesn't access memory

## IA32 Assembly Language examples ...

- quickest way to clear a register?

```
xor    eax, eax
```

; exclusive OR with itself

```
mov    eax, 0
```

; instruction occupies more bytes and...  
; probably takes longer to execute

- quickest way to test if a register is zero?
- NB: mov instruction doesn't update the condition code flags

```
test   eax, eax
```

```
je     ...
```

; AND eax with itself, set flags and...  
; jump if zero

## Function/Procedure Calling

reminder of the steps normally carried out during a function/procedure call and return

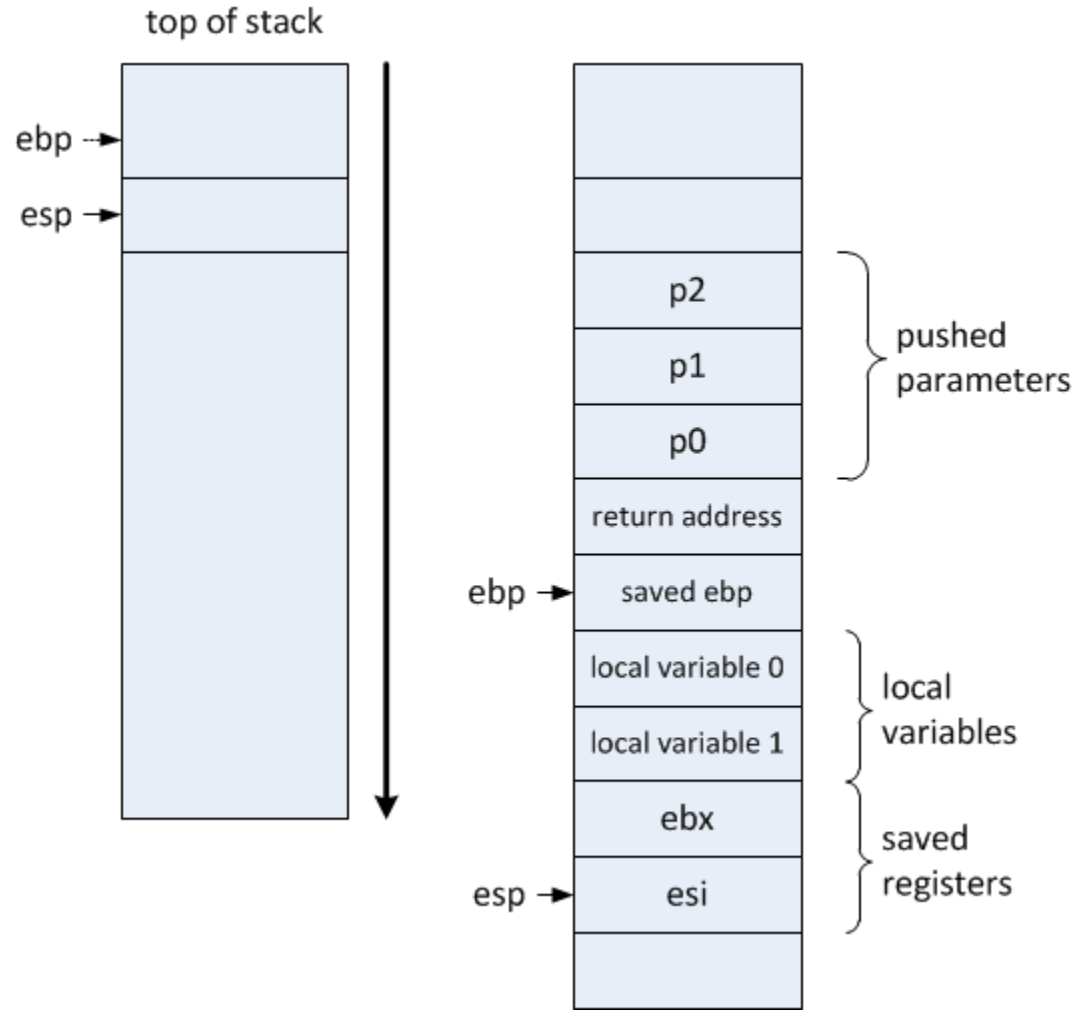
- pass parameters [*IA32: evaluate and push on stack*]
- enter new function [*IA32: push return address and jump to first instruction of function*]
- allocate space for local variables [*IA32: on stack by decrementing esp*]
- save registers [*IA32: on stack*]

*<function body>*

- restore saved registers [*IA32: from stack*]
- de-allocate space for local variables [*IA32: increment esp*]
- return to calling function [*IA32: pop return address from stack*]
- remove parameters [*IA32: increment esp*]

## IA32 Function Stack Frame

- stack frame after call to `f(p0, p1, p2)`
- stack grows down in memory [from highest address to lowest]
- parameters pushed right to left
- NB: stack always aligned on a 4 byte boundary [it's not possible to push a single byte]
- ebp used as a frame pointer  
parameters and locals accessed relative to ebp [`p0 @ ebp+8`]



## IA32 Calling Conventions

- several IA32 procedure/function calling conventions
- will use Microsoft `_cdecl` calling convention [as per previous diagram] so C/C++ and IA32 assembly language code can mixed
  - function result returned in `eax`
  - `eax`, `ecx` and `edx` considered volatile and are NOT preserved across function calls, others registers need to be saved and restored if used
  - caller removes parameters
- why are parameters pushed right-to-left??

C/C++ pushes parameters right-to-left so functions like `printf(char *formats, ...)` [which can accept an arbitrary numbers of parameters] can be handled more easily since the first parameter is always stored at `[ebp+8]` irrespective of how many parameters are pushed

## Accessing Parameters and Local Variables

- ebp used as a frame pointer
- parameters and local variables accessed at offsets from ebp
- can avoid using a frame pointer [**normally for speed**] by accessing parameters and locals variables relative to the stack pointer, but more difficult because the stack pointer can change during execution [**BUT easy for a compiler to track**]
- parameters accessed with +ve offsets from ebp [**see stack frame diagram**]

p0 @ [ebp+8]

p1 @ [ebp+12]

...

- local variables accessed with -ve offsets from ebp [**see stack frame diagram**]

local variable 0 @ [ebp-4]

local variable 1 @ [ebp-8]

...

## Consider the IA32 Code for a Simple Function

```
int f (int p0, int p1, int p2) {    // parameters
    int x, y;                      // local variables
    x = p0 + p1;
    ...
    return x + y;                  // result
}
```

- a call `f(p0, p1, p2)` matches stack frame diagram on previous slide
- 3 parameters *p0*, *p1* and *p2* and 2 local variables *x* and *y*
- need to generate code for
  - *calling function f*
  - *function f entry*
  - *function f body*
  - *function f exit*



## IA32 Code to Call Function f

- parameters  $p0$ ,  $p1$  and  $p2$  pushed onto stack by caller right to left

f(1, 2, 3)

```
push    3
push    2
push    1
call    f
add     esp, 12
```

; push immediate values...


; right to left

;

; call f

; add 12 to esp to remove parameters from stack

push return address  
and jump to f



## Function Entry

- need instructions to save ebp [**old frame pointer**] and ...
- initialize ebp [**new frame pointer**] and ...
- allocate space for local variables on stack and ...
- push non volatile registers used by function onto stack

```
f:    push    ebp                ; save ebp
      mov     ebp, esp          ; ebp -> new stack frame
      sub     esp, 8            ; allocate space for locals x and y
      push    ebx              ; save non volatile registers used by function

      <function body>          ; function body

      <function exit>          ; function exit
```

NB: `_cdecl` convention means there is NO need to save eax, ecx and edx

## Function Body

- parameters pushed on stack and ...
- space already allocated for local variables

*parameters p0 @ [ebp+8] and p1 @ [ebp+12]*

*locals x @ [ebp-4] and y @ [ebp-8]*

- $x = p0 + p1$

```
mov    eax, [ebp+8]    ; eax = p0
add    eax, [ebp+12]   ; eax = p0 + p1
mov    [ebp-4], eax    ; x = p0 + p1
```

- return  $x + y$ ;

```
mov    eax, [ebp-4]    ; eax = x
add    eax, [ebp-8]    ; eax = x + y
```

NB: result returned in eax

## Function Exit

- need instructions to unwind stack frame at function exit

```
...  
pop      ebx           ; restore saved registers  
mov       esp, ebp      ; restore esp  
pop       ebp           ; restore previous ebp  
ret       0             ; return from function
```

- ret pops return address from stack and...
- adds integer parameter to esp [*used to remove parameters from stack*]
- if integer parameter not specified, defaults to 0
- since using *\_cdecl* convention caller will remove parameters from stack
- make sure you know why a stack frame needs to be created for each function call