# IA-32 Assembly

(Deeper and deeper...)

Emmanuel Fleury
LaBRI, Office 261
<emmanuel.fleury@labri.fr>

# Outline

- Motivations & Warnings
- IA-32 Processors Family
- Memory Discipline: Stack & Heap
- Computational Model
- IA-32 Registers
- IA-32 Assembly Syntax(es)
- Basic Instruction Sets
- Interruptions & System Calls
- Advanced Instruction Sets

# Motivations
# &
# Warnings

# What's Assembly Useful For ?

- **Understand the machine**
  (find bugs is easier, less program design errors, ...)
- **Code hardware dedicated subroutines**
  (compilers, drivers, operating systems, ...)
- **Optimize high-level language code**
  (manage your compiler...)
- **Reverse engineering**
  (malware analysis, driver analysis, ...)

**Knowing Assembly will enhance your code !**

# What's Assembly Bad At ?

- **Portability is lost**
  (code is working only for only family of processors)
- **Optimization is tedious**
  (compilers are more efficient than humans to optimize code)
- **Obfuscate the code**
  (only few programmers can read assembly)
- **Debugging is difficult**
  (most of the debuggers are lost when hitting assembly code)

**Use it with caution and sparsity !!!**

# IA-32 Processors Familly

# Ancestors (Before IA-32)

- Intel 4004 (1971): First microchip
  4bits memory words, 640b of addressable memory, 740kHz
- Intel 8008 (1972):
  8bits memory words, 16kb of addressable memory, 800kHz
- Intel 8086 (1978):
  16bits memory words, 1Mb of addressable memory, 10MHz
- Intel 80286 (1982):
  16bits memory words, 16Mb of addressable memory, 12.5MHz

# First IA-32 Generation

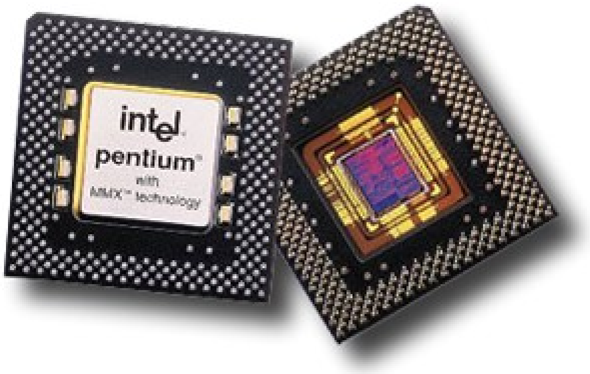
- Intel 80386(DX) (1985):

  Introduce a Memory Management Unit (MMU)
  32bits memory words, 4Gb of addressable memory, 16MHz


- Intel 80486(DX) (1989):

  Mathematics co-processor built on-chip
  32bits memory words, 4Gb of addressable memory, 16MHz

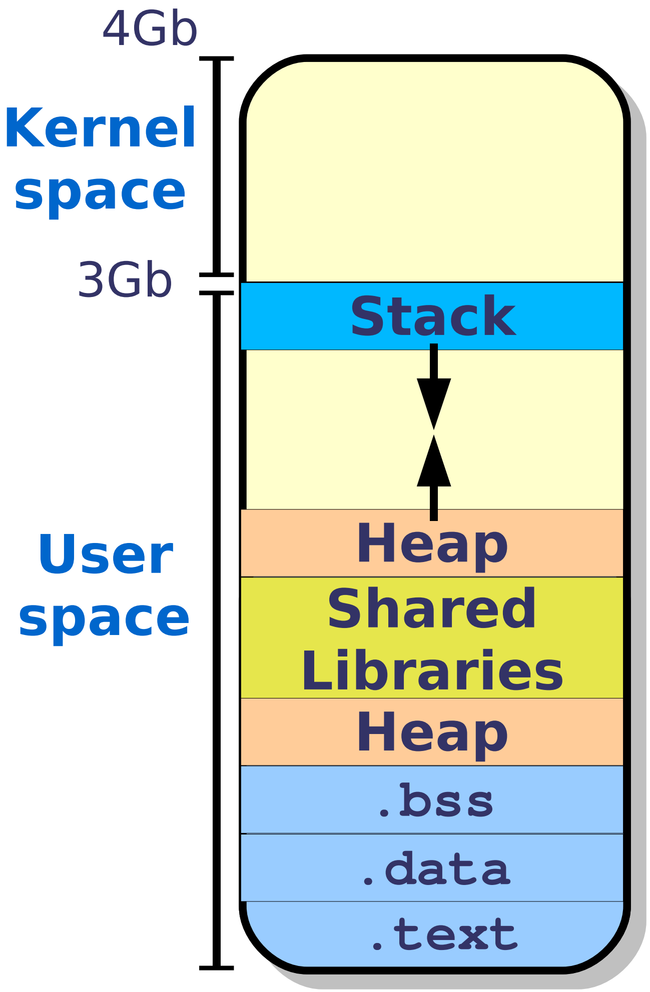
# IA-32 Compatible Processors




- Intel IA-32 Processors:
  Pentium, Pentium II, Pentium III, Pentium 4,
  Pentium M, Celeron, Core, Core 2;
- AMD IA-32 Processors:
  K5, K6, K7 (Athlon, Sempron, Duron),
  K8 (Athlon, Sempron, Duron);
- Transmeta IA-32 Processors:
  Crusoe, Efficeon.
- Others…

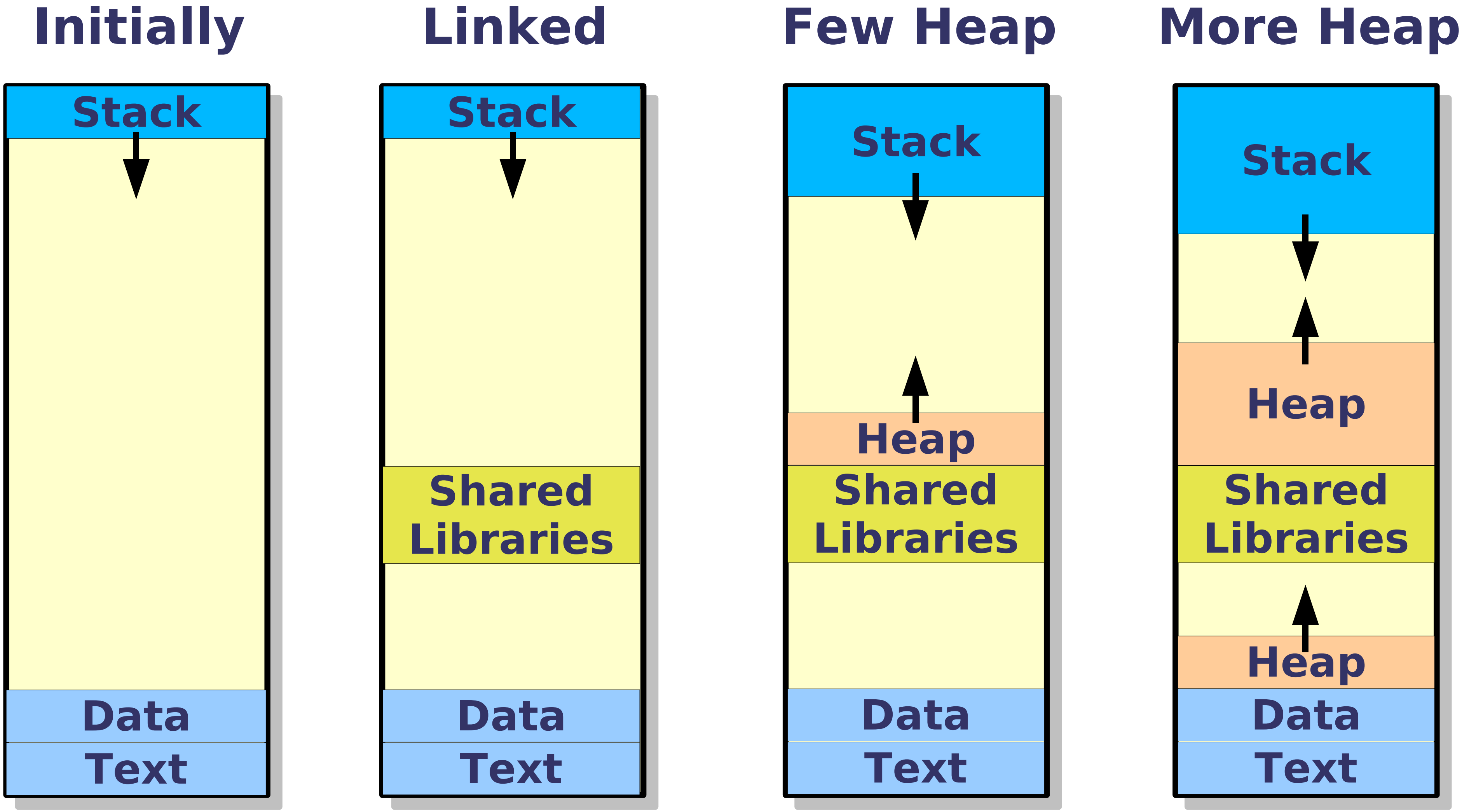
# Memory Discipline: Stack & Heap

# Linux Memory Layout




- **Stack**
  Runtime stack (8Mb limit by default);
- **Heap**
  Dynamically allocated storage (`malloc()`, `calloc()`, `new`, `...`);
- **Shared/Dynamic Libraries**
  Libraries routines (*e.g.* `printf()`, ...);
- **`.bss`, `.data`**
  Statically allocated data (BSS start all zeroed);
- **`.text`, RODATA (Read-Only Data)**
  Executable machines instructions and Read-Only DATA (string literals).

# Linux Memory Allocation

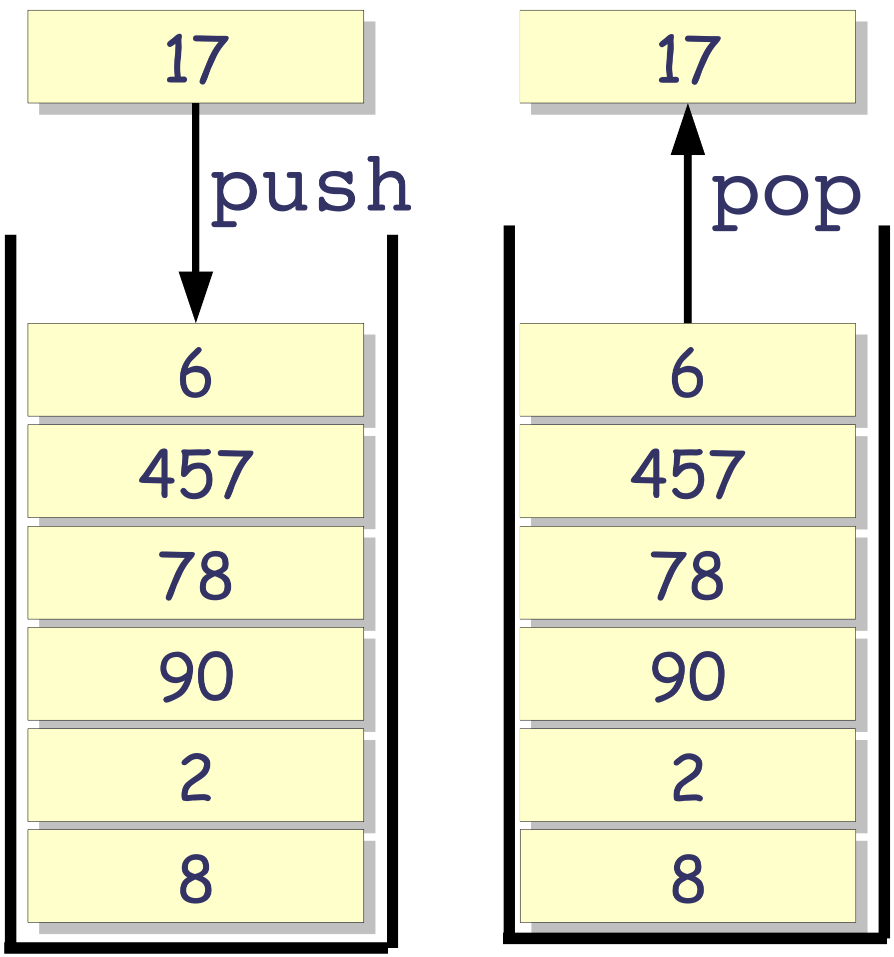


# Stack Discipline

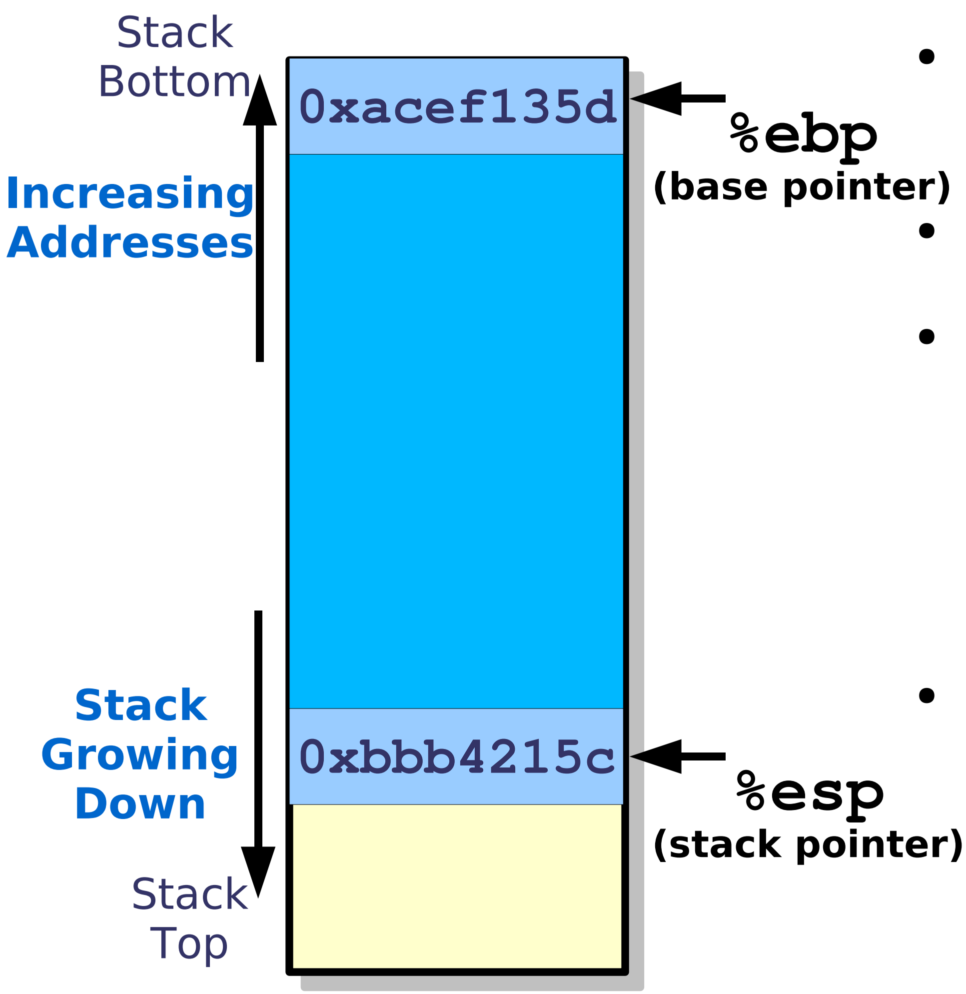
# What is a Stack (FILO) ?

## First In Last Out

- Implemented via:
  Array, Linked-list
- Applications:
  Stack based calculus
  (CPU, parser, ...)
- Interface through:
  Push & Pop

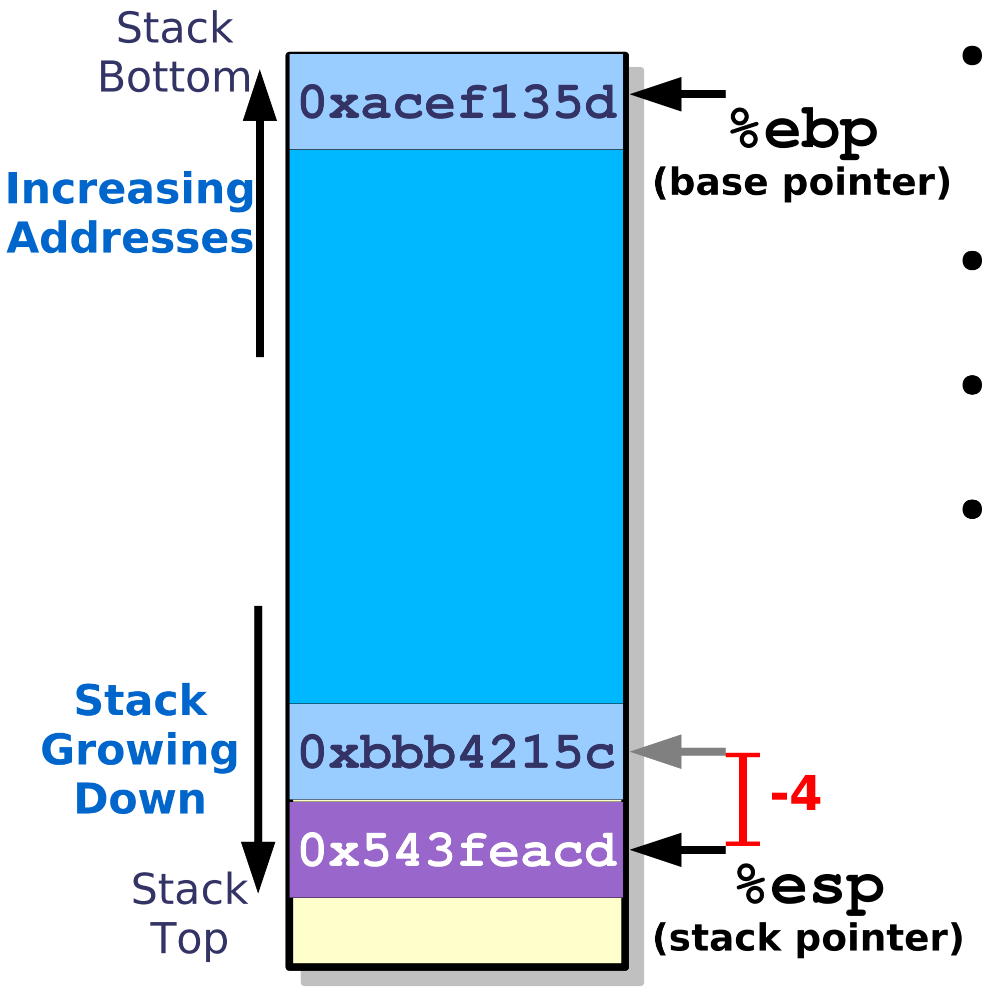


# IA-32 Stack



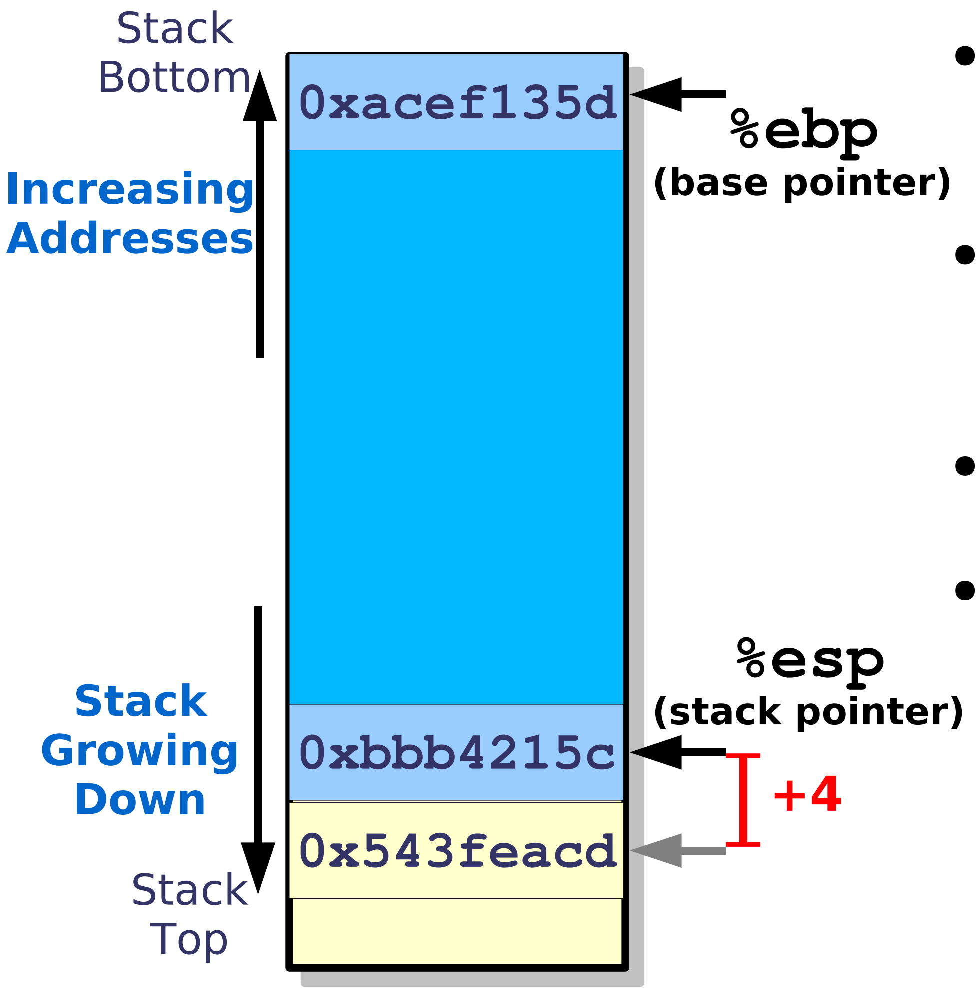

- Memory zone managed with "*stack discipline*"
- Grows toward *lower addresses*
- Register **`%esp`** indicates lowest stack address
  - Mark the "top" element
  - "*stack pointer*" (SP)
- Register **`%ebp`** indicates highest stack address
  - Mark the "bottom" element
  - "*base pointer*" (BP)

# IA-32 Stack Pushing



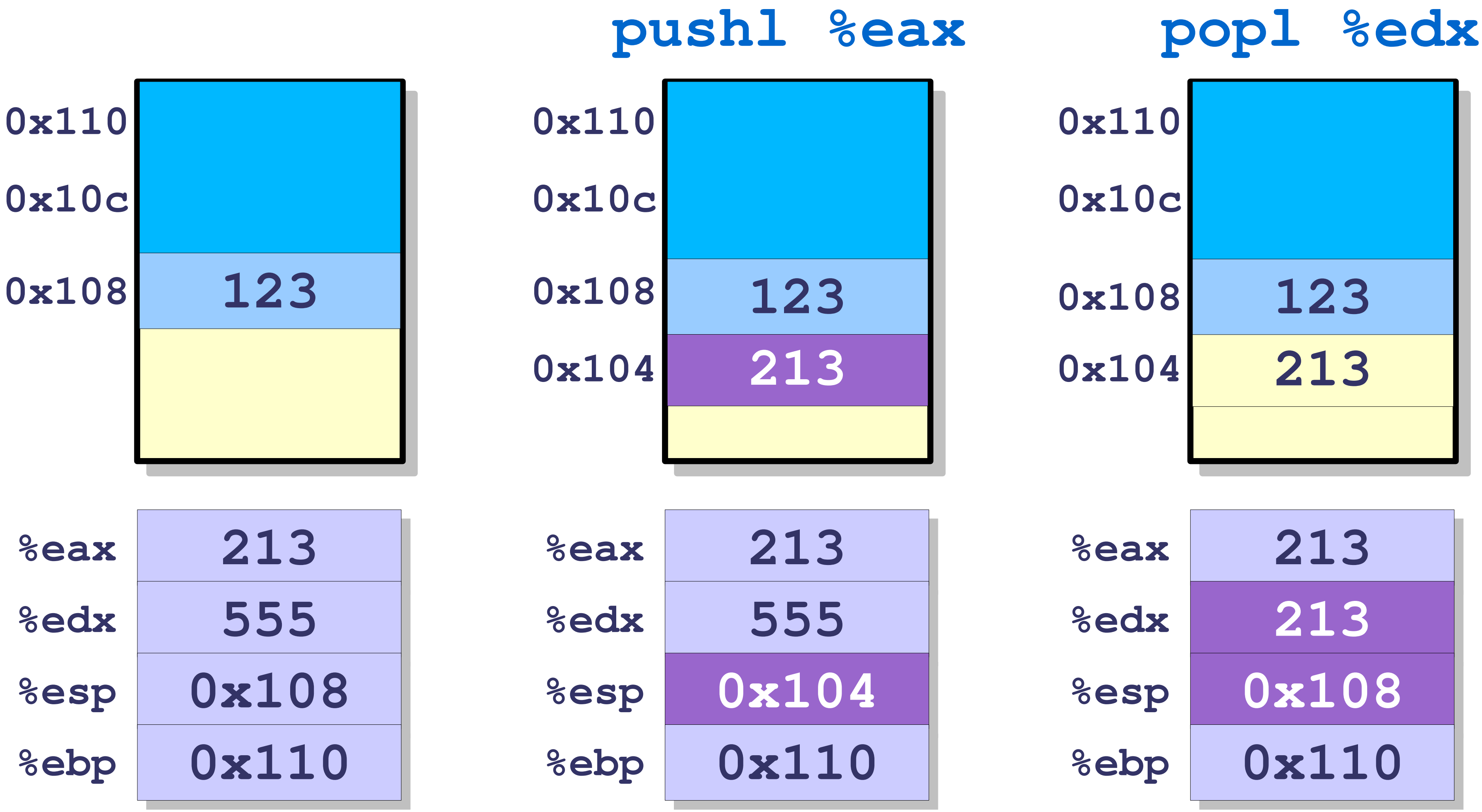

- **`pushl Src`** (`Src`=memory address)
- Fetch operand at **`Src`**
- Decrement **`%esp`** by 4
- Write operand at **`(%esp)`**

# IA-32 Stack Popping



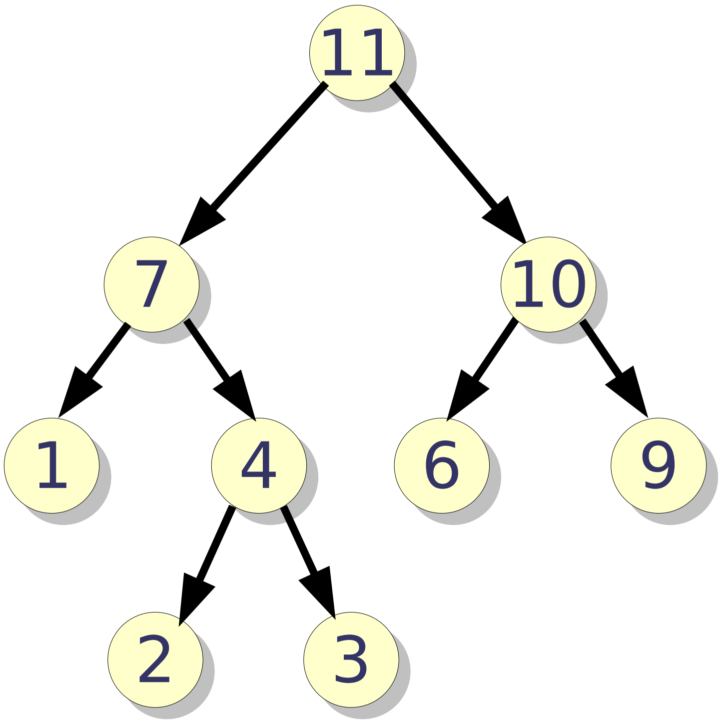

- `pop Dest` (`Dest`=memory address)
- Read operand at (`%esp`)
- Increment `%esp` by 4
- Write operand to `(Dest)`

# Stack Operation Examples



# Heap Discipline

# What is a (binary) Heap ?

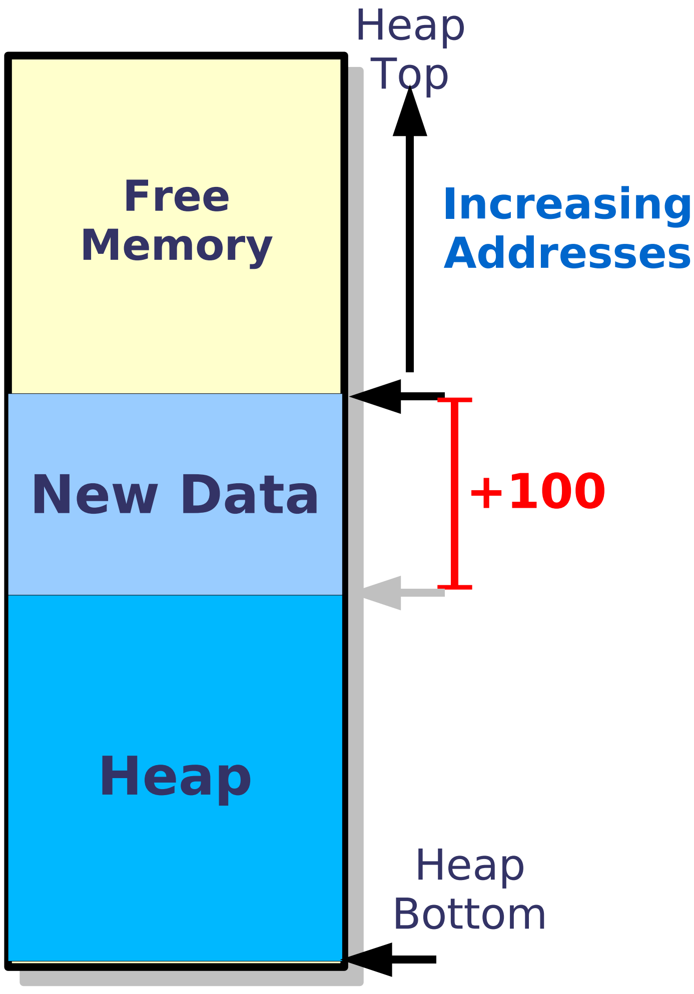



A heap is a tree structure such that,
if A and B are nodes of a heap and
B is a child of A, then:

**key(A)≥key(B)**

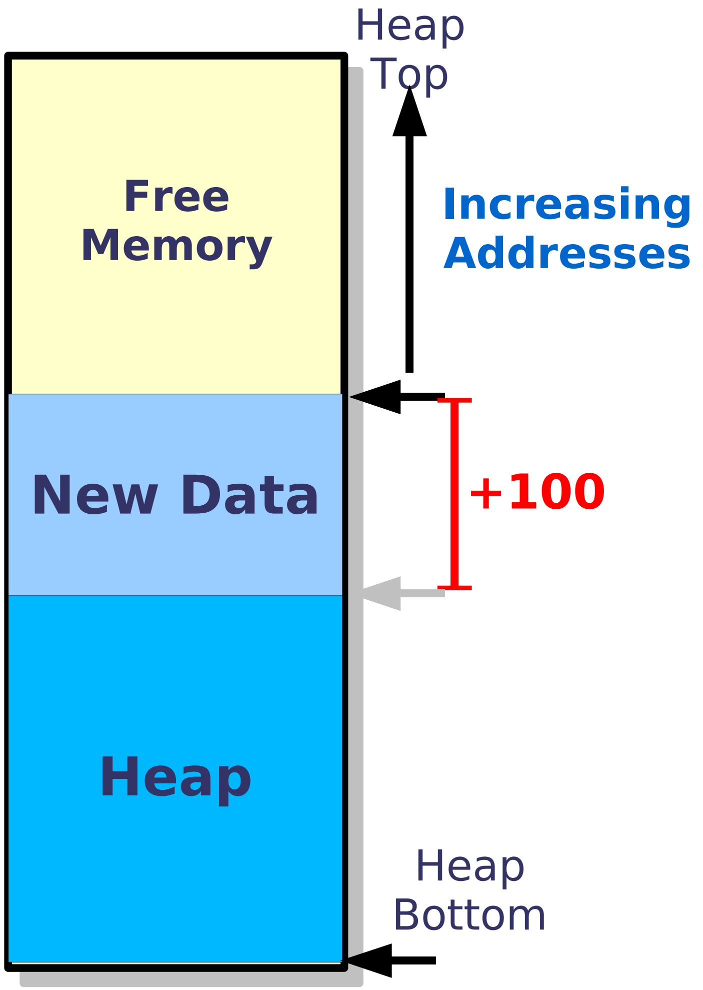
- Implemented via:
  - Arrays
    (a[i] has two children
    a[2i+1],a[2i+2])
  - Trees
- Applications:
  Quick access to data (databases)
- Groups of Data:
  In Doug Lea malloc (dlmalloc)
  memory chunks are classified by
  size (bytes).

# IA-32 Heap




- Memory zone managed with "heap *discipline*"
- Grows toward higher addresses
- From programmer point of view:
  Managed through a language dependent interface (C, C++,...).
- From the system point of view:
  Managed through specific system calls (`mmap()`, `brk()`).

# IA-32 Heap (System Calls)




**`brk(void *end_data_segment):`**

Sets the end of the data segment to the value specified by **`end_data_segment`**, when that value is reasonable, the system does have enough memory and the process does not exceed its max data size.

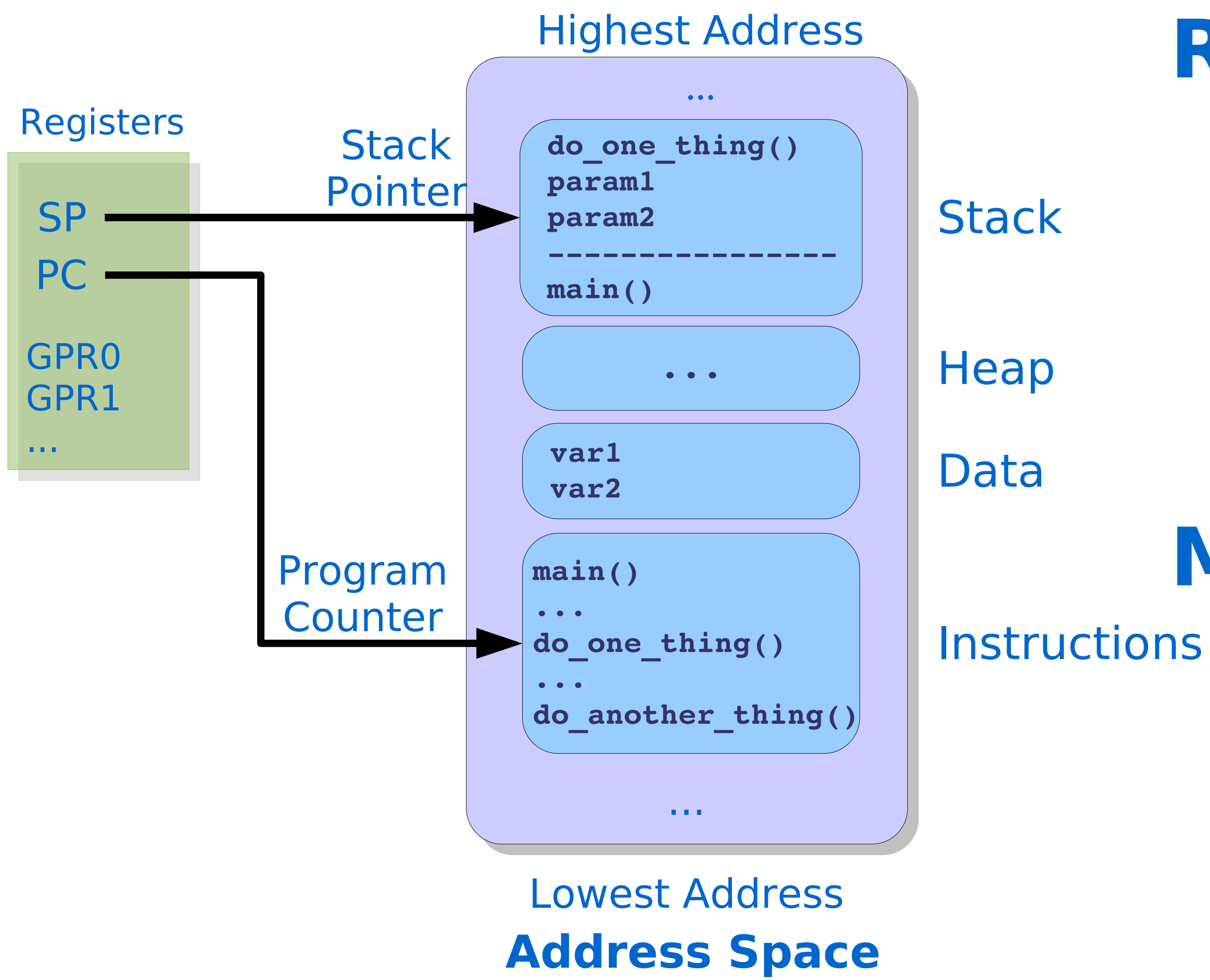
**`mmap():`**

Asks to map length bytes starting at offset offset from the file (or other object) specified by the file descriptor **`fd`** into memory, preferably at address start. This latter address is a hint only, and is usually specified as 0. The actual place where the object is mapped is returned by **`mmap()`**, and is never 0.

> **`brk()`** is used for small chunks, where **`mmap()`** is used for big ones.

# Computational Model

# Computational Model




## Registers:

- SP (Stack Pointer)
- PC (Program Counter)
- GPR (General Purpose Register)

## Memory:

- Stack
- Heap
- Data
- Instructions

# Computational Model (Example)

## Computing “(5*3+2)-5”:

```
movl    $5,%eax # eax = 5;
imull   $3,%eax # eax = 3*eax;
addl    $2,%eax # eax = eax+2;
subl    $5,%eax # eax = eax-5;
```

Instructions

“Immediate” Values

Register Name

Comments

# Computational Model (Example)

## Computing “(5*3+2)-5”:

```
movl    $5,%eax # eax = 5;
imull   $3,%eax # eax = 3*eax;
addl    $2,%eax # eax = eax+2;
subl    $5,%eax # eax = eax−5;
```

eax 5

# Computational Model (Example)

## Computing “(5*3+2)-5”:

```
movl    $5,%eax # eax = 5;
imull   $3,%eax # eax = 3*eax;
addl    $2,%eax # eax = eax+2;
subl    $5,%eax # eax = eax-5;
```

eax $\boxed{5}$ *3=15

# Computational Model (Example)

## Computing “(5*3+2)-5”:

```
movl    $5,%eax # eax = 5;
imull   $3,%eax # eax = 3*eax;
addl    $2,%eax # eax = eax+2;
subl    $5,%eax # eax = eax-5;
```

eax 15 +2=17

# Computational Model (Example)

## Computing “(5*3+2)-5”:

```
   movl    $5,%eax # eax = 5;
   imull   $3,%eax # eax = 3*eax;
   addl    $2,%eax # eax = eax+2;
→  subl    $5,%eax # eax = eax-5;
```

eax $\boxed{17}$ –5=12

# Computational Model (Example)

## Computing "(5*3+2)-5":

```
movl    $5,%eax # eax = 5;
imull   $3,%eax # eax = 3*eax;
addl    $2,%eax # eax = eax+2;
subl    $5,%eax # eax = eax-5;
```

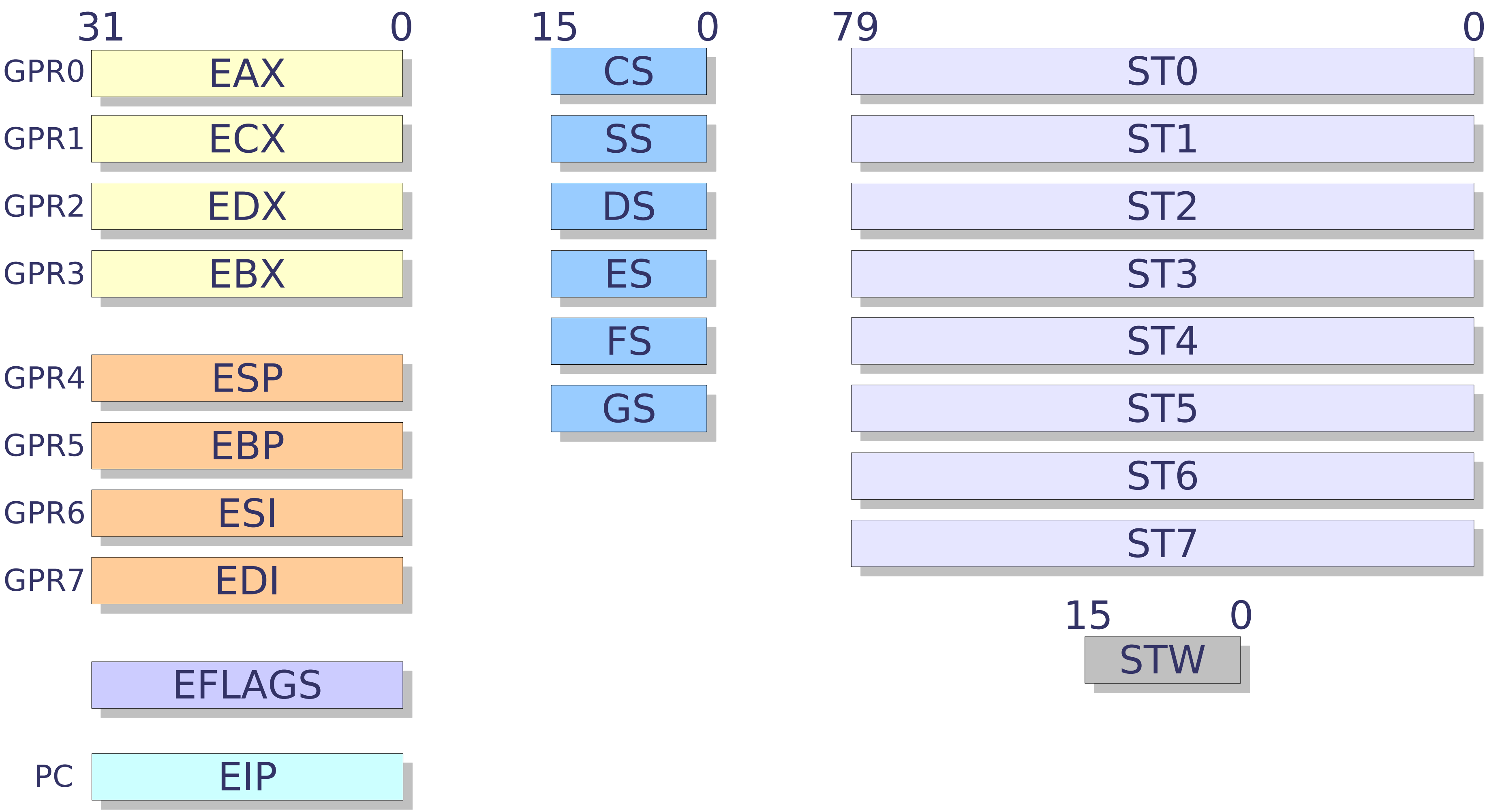
eax 12

# IA-32 Registers

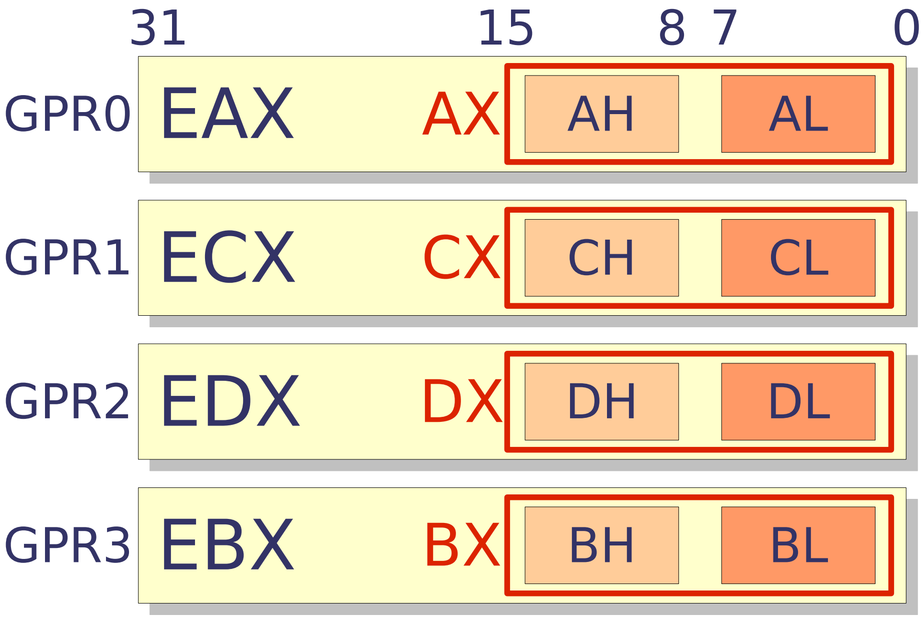
# IA-32 Registers

- **Data Registers (Read/Write)**
  (EAX, EBX, ECX, EDX)
- **Index & Pointers Registers (Read/Write)**
  (EBP, EIP, ESP, ESI, EDI)
- **Segment Registers (Protected)**
  (CS, DS, ES, FS, GS, SS)
- **Flags Registers (Read)**
  (EFLAGS)
- **Floating-point Registers (Read/Write)**
  (ST0, ..., ST7)

# IA-32 Registers



# Data Registers




For **backward compatibility** old **8 bits** and **16 bits** registers have been **preserved**. You still can address them.

**EAX** (Accumulator)

Accumulator for operands and results data (addition, subtraction, ...);

**EBX** (Base Register)

Usually used to store the base address of a data-structure in memory;

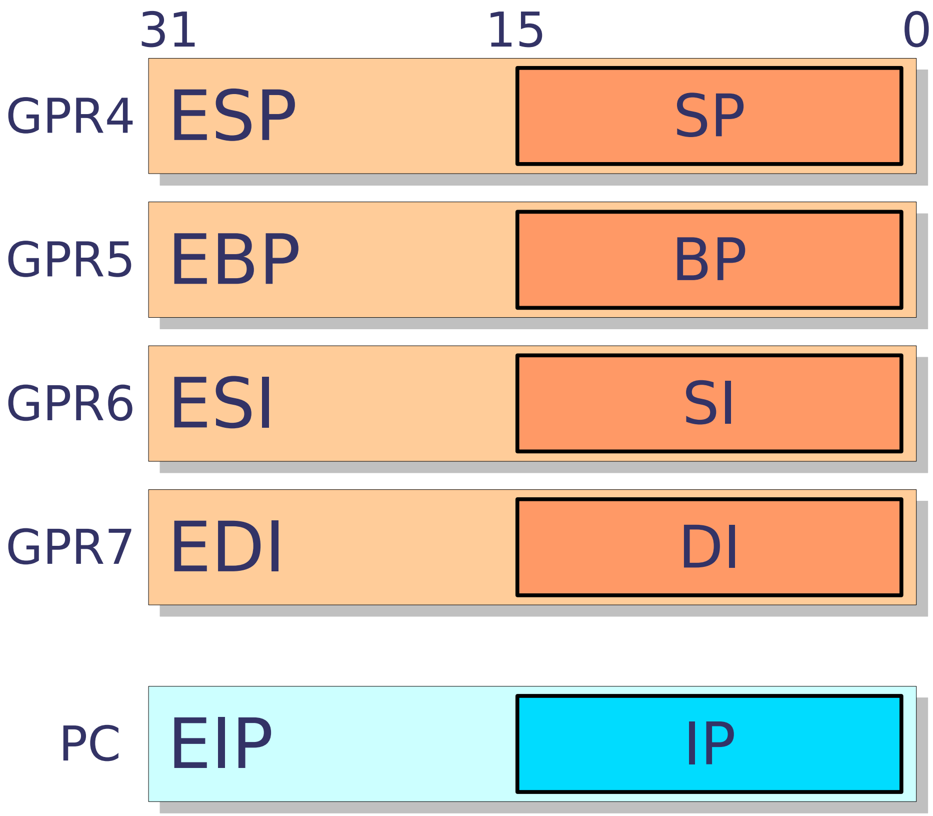
**ECX** (Count Register)

Usually used as a loop counter;

**EDX** (Data Register)

Used to store operand and result for multiplications and divisions.

# Index & Pointers Registers




For **backward compatibility** old **16 bits** registers have been **preserved**. You still can address them.

**ESP** (Stack Pointer)

Pointer to the last cell of the stack;

**EBP** (Base Pointer)

Pointer to the base cell of the current stack frame;

**ESI** (Source Index)

Used in string operations as source;

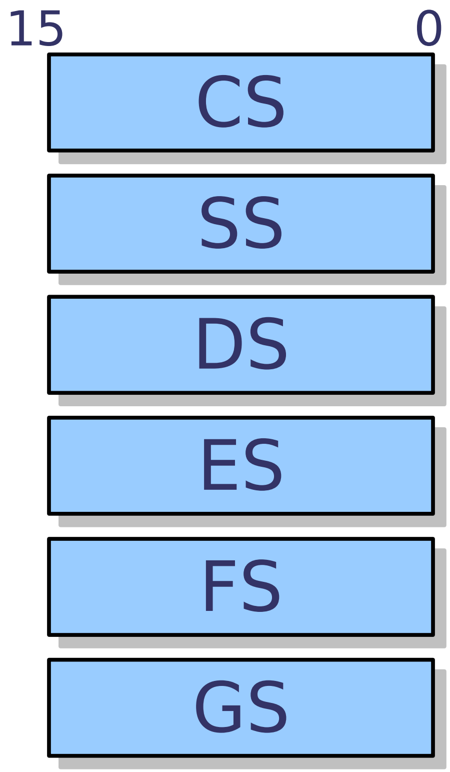
**EDI** (Destination Index)

Used in string operations as destination;

**EIP** (Instruction Pointer)

Point to the next instruction.

# Segment Registers




**CS** (Code Segment)

Point to the current code-segment;

**SS** (Stack Segment)

Point to the current stack-segment;

**DS** (Data Segment)

Point to the current data-segment;

**ES**, **FS**, **GS** (Extra Data Segments)

Extra segments registers available for far pointer addressing (video memory and others).

# Flag Registers (Types)

EFLAGS | FLAGS

## Status Flags (stat)

Give the result of arithmetic instructions (add, sub, mul or div)

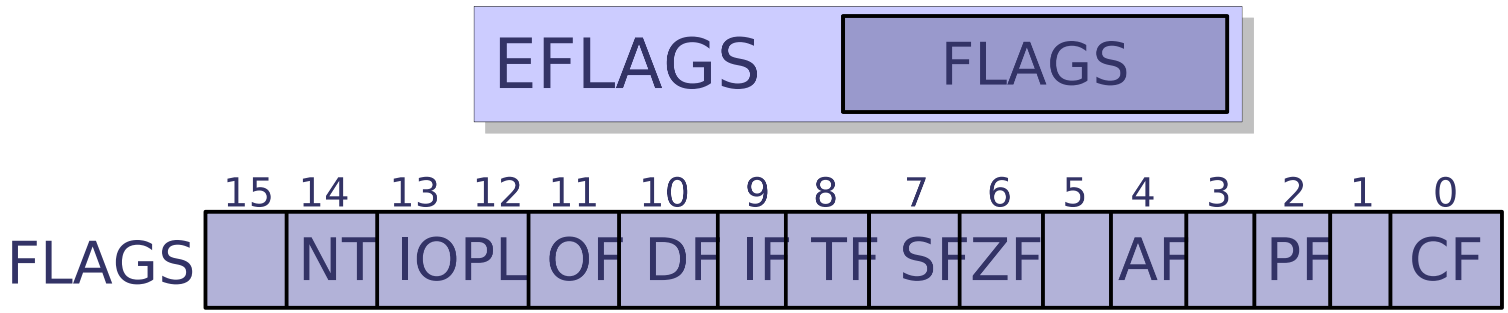
## Control Flags (ctrl)

Change the behavior of the processor on some instructions (std, cld)

## System Flags (sys)

Accessible by kernel only

# Flag Registers (FLAGS)




**CF** (Carry Flag, stat): Left most bit of result;

**PF** (Parity Flag, stat): '1' if result is even;

**AF** (Auxiliary Carry Flag, stat): Extra carry flag;

**ZF** (Zero Flag, stat): '1' if result is '==0';

**SF** (Sign Flag, stat): '1' if result is '<0';

**TF** (Trap Flag, sys): Set CPU in single-step mode;

**IF** (Interrupt Flag, sys): '1' if interruptions are on;

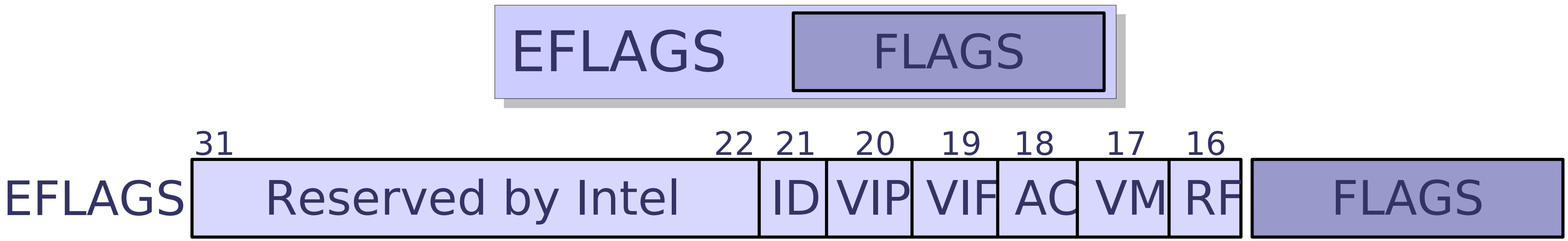
**DF** (Direction Flag, ctrl): Strings reading order. Modified by `std` ('1', go from higher to lower addresses) and `cld` ('0', reverse order);

**OF** (Overflow Flag, stat): '1' if an overflow occurs;

**IOPL** (I/O Privilege Level, sys): Current task privilege;

**NT** (Nested Task Flag, sys): '1' if current task is linked to the previous one.

# Flag Registers (EFLAGS)




**RF** (Resume Flag, sys):
Set CPU in debug mode;

**VM** (Virtual Mode, sys):
Set the 8086 virtual mode (unset the protected mode);

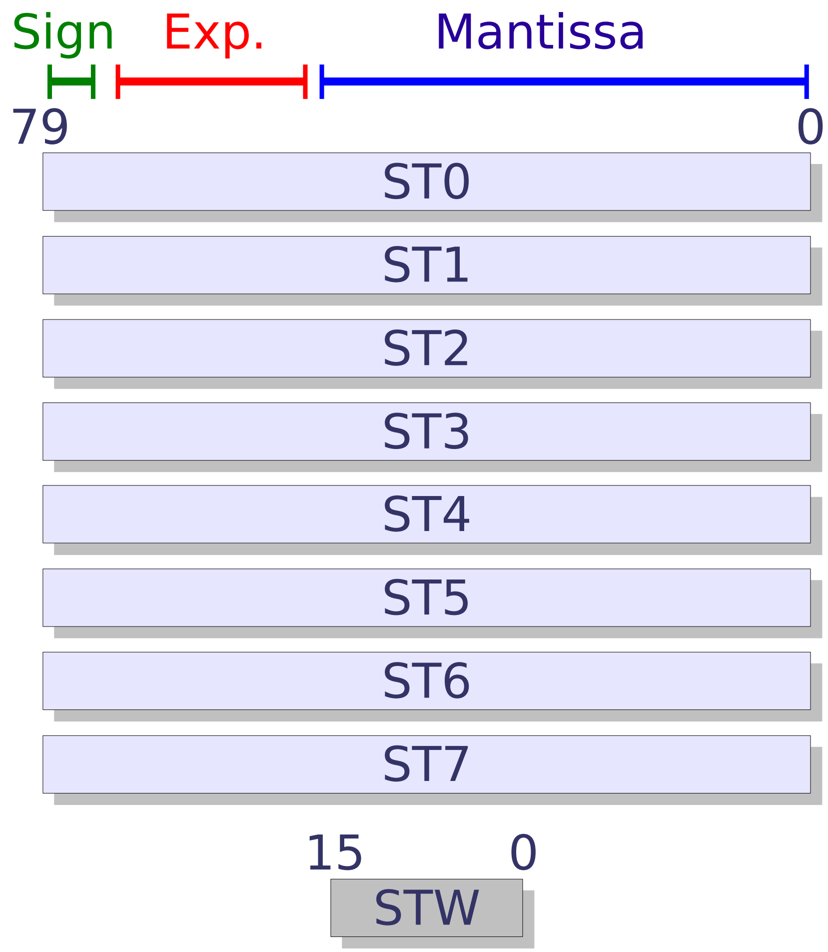
**AC** (Alignment Check Flag, sys):
Set alignment checking mode for memory references;

**VIF** (Virtual Interrupt Flag, sys):
Virtual image of the IF flag (used in conjunction with VIP);

**VIP** (Virtual Interrupt Pending Flag, sys):
Indicates pending interruptions ('1' if one is pending);

**ID** (ID Flag, sys):
Triggers the CPUID instruction support.

# Floating-point Registers




- 80 bits wide registers
- Accessed as a stack
  (can't be addressed directly)
- Decomposed into:
  - Sign: 1 bit;
  - Exponent: 15 bits;
  - Mantissa: 64 bits;

# Status Word Register

| | 15 | 14 | 13 12 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| STW | B | C3 | ST | C2 | C1 | C0 | IR | | EU | PE | OE | ZE | DE | IE |

## Exception Flags (bits 0-6)

**IE** Invalid Operation Exception
(Uncertain result)

**DE** Denormalized Operation Exception
(Operation on denormalized number)

**ZE** Zero Divide Exception (Division by zero)

**OE** Overflow Exception (Result is too large)

**UE** Underflow Exception (Result is too small)

**PE** Precision Exception
(Loss of precision occured)

**ST** (Stack pointer): Counting free registers;

**Condition Code** (C0,C1,C2,C3): Store results of boolean tests on float in these bits;

**Others** (IR, B):

**IR** Interruption Request

**B** Busy (on-going operation)

# IA-32 Assembly Syntax(es)

# Intel *vs.* AT&T Syntaxes

Tools are using two types of syntaxes !

- **Intel Syntax:**
  All the Intel manuals are written in this syntax. A lot of tools are using this syntax, mostly tools from the "*Microsoft community*".

- **AT&T Syntax:**
  All the GNU binutils tools use this syntax.
  Most of the "*Unix community*" is also using it.

# Intel vs. AT&T
## (Direction of operand)

### Intel Syntax:

The first operand is the destination and the second operand is the source.

### AT&T Syntax:

The first operand is the source and the second operand is the destination.

**Intel Syntax**

```
Instr.      dest, src
mov         eax,[ecx]
```

**AT&T Syntax**

```
Instr.      src, dest
mov         (%ecx),%eax
```

# Intel vs. AT&T
## (Data types)

### Intel Syntax:

- **10d**: Decimal value (**10** is ok);
- **10h**: Hexadecimal value;
- **1**: Immediate value;
- **eax**: Register;
- **byte ptr**: Address of a byte (8bits)
- **word ptr**: Address of a word (16bits)
- **dword ptr**: Address of a long (32bits)

### AT&T Syntax:

- **0d10**: Decimal value (**10** is ok);
- **0x10**: Hexadecimal value;
- **$1**: Immediate value;
- **%eax**: Register;
- **movb**: Operand on bytes (8bits)
- **movw**: Operand on words (16bits)
- **movl**: Operand on longs (32bits)

# Intel vs. AT&T
## (Data types)

### Intel Syntax

```
mov   eax, 1
mov   ebx,0ffh
int   80h
mov   al,bl
mov   ax,bx
mov   eax,ebx
mov   eax,dword ptr [ebx]
```

### AT&T Syntax

```
movl      $1,%eax
movl      $0xff,%ebx
int       $0x80
movb      %bl,%al
movw      %bx,%ax
movl      %ebx,%eax
movl      (%ebx),%eax
```

# Intel vs. AT&T
## (Addressing memory)

### Intel Syntax:

The address is enclosed in brackets (“**[**“, “**]**”) and specified by:

```
[base+index*scale+offset]
```

### AT&T Syntax:

The address is enclosed in parenthesis (“**(**“, “**)**”) and specified by:

```
offset(base,index,scale)
```

**Intel Syntax**

```
mov eax,[ebx]
mov eax,[ebx+3]
mov eax,[ebx+20h]
add eax,[ebx+ecx*2h]
lea eax,[ebx+ecx]
sub eax,[ebx+ecx*4h-20h]
```

**AT&T Syntax**

```
movl (%ebx),%eax
movl 3(%ebx),%eax
movl 0x20(%ebx),%eax
addl (%ebx,%ecx,0x2),%eax
leal (%ebx,%ecx),%eax
subl -0x20(%ebx,%ecx,0x4),%eax
```

# Basic Instruction Sets

# Types of Instructions

## Memory Operators

Moving data, stack management;

## Arithmetic & Logic

Bitwise operators, shift & rotate, Integer arithmetic, floating-point arithmetic;

## Flow Control

Tests, jump, loops;

# Moving Data

| Mnemonic | Operand | Operand | Operation |
|---|---|---|---|
| **mov** | src | dst | src → dst |
| **xchg** | dst | dst | dst ↔ dst |
| **lea** | mem | reg | mem → reg |
| **movz** | src8 | reg16 | zero-extended src → reg |
| | src8 | reg32 | |
| | src16 | reg32 | |
| **movsz** | src8 | reg16 | sign-extended src → reg |
| | src8 | reg32 | |
| | src16 | reg32 | |
| **cbw** | - | | sign-extended %al → %ax |
| **cwd** | - | | sign-extended %ax → %dx.%ax |
| **cdq** | - | | sign-extended %eax → %edx.%eax |
| **cwde** | - | | sign-extended %ax → %eax |

lea = Load Effective Address

# Moving Data (Example)

- **The code:**

```
# Example 1
.globl main

main:
    movl    $20, %eax
    ret
```

- **Building the binary:**

```
gcc -o example1 asm.s
```

- **Running the binary:**

```
./example1
```

# Stack Management

| Mnemonic | Operand | Operation |
|---|---|---|
| **push** | src8 | %esp-1 → %esp ; src → (%esp) |
| | src16 | %esp-2 → %esp ; src → (%esp) |
| | src32 | %esp-4 → %esp ; src → (%esp) |
| **pop** | dst8 | (%esp) → dst ; %esp+1 → %esp |
| | dst16 | (%esp) → dst ; %esp+2 → %esp |
| | dst32 | (%esp) → dst ; %esp+4 → %esp |
| **pushf** | - | %esp-4 → %esp ; %eflags → (%esp) |
| **popf** | - | (%esp) → %eflags ; %esp+4 → %esp |
| **pusha** | - | Push %eax,%ecx,%edx,%ebx,%esp,%ebp,%esi,%edi |
| **popa** | - | Pop %eax,%ecx,%edx,%ebx,%esp,%ebp,%esi,%edi |
| **enter** | - | Create a stack frame |
| **leave** | - | Restore the previous stack frame |

# Push'n Pop

```
.glob main

main:
   movl    $20, %eax
   pushl   %eax                # Push in the stack
   popl    %ebx                # Pop from the stack

   movl    $15, -4(%ebp)       # Push in the stack
   movl    4(%ebp), %ebx       # Pop from the stack

   ret
```

# Memory Allocation

```
.glob main

main:
    # Allocate memory space
    pushl %ebp              # Save base pointer
    movl  %esp, %ebp        # Set stack pointer at base pointer
    subl  $8, %esp          # Allocate memory space for two words

    # Data manipulations
    pushl $10               # Push 10 in the stack
    pushl $15               # Push 15 in the stack
    popl  %eax              # Pop 15 from the stack
    popl  %ebx              # Pop 10 from the stack

    # Leaving stack-frame
    movl  %ebp, %esp        # Restore previous stack-pointer
    popl  %ebp              # Restore the old base pointer
    ret                     # Restore previous execution flow
```

# Always Restore the Stack

```
.glob main

main:
    # Data manipulations
    pushl $10               # Push 10 in the stack
    pushl $15               # Push 15 in the stack
    popl  %eax              # Pop 15 from the stack
    popl  %ebx              # Pop 10 from the stack

    # Stack-pointer is not restored
    ret                     # Restore previous execution flow
```

## Running the program:

```
#bash> ./test
Segmentation fault
```

# Bitwise Operators

| Mnemonic | Operand | Operand | Operation | Touched Flags |
|---|---|---|---|---|
| **and** | src | dst | src & dst → dst | SF,ZF,PF |
| **or** | src | dst | src \| dst → dst | |
| **xor** | src | dst | src ^ dst → dst | |
| **test** | src | dst | src & dst (result discarded) | |
| **not** | dst | | ~dst → dst | - |
| **cmp** | src | dst | sub src,dst (result discarded) | OF,SF,ZF,AF,CF,PF |

**Example:**

```
.globl main
main:
    movl   $8, %eax
    andl   $9, %eax
    notl   %eax
L0:cmp    $8, %eax      # %eax == 8
    jnz       L1        # Jump to L1 if ZF<>0
    jmp       L0        # Jump to L0
L1:ret
```

# Shift & Rotate

| Mnemonic | Operand | Operand | Operation | Touched Flags |
|---|---|---|---|---|
| **shl/sal** | src | dst | left shift dst of src bits | CF,OF |
| **shr/sar** | src | dst | right shift dst of src bits | |
| **rol** | src | dst | left rotate dst of src bits | |
| **ror** | src | dst | right rotate dst of src bits | |

Note: **shl/shr** are unsigned version of **sal/sar**.

## $2^7$ Multiplication:

```
.globl main
main:
    shll    $7, %eax    # %eax*2^7
    ret
```

# Addition/Subtraction (Int)

| Mnemonic | Operand | Operand | Operation | Touched Flags |
|---|---|---|---|---|
| add | src | dst | src + dst → dst | OF,SF,ZF, AF,CF,PF |
| adc | src | dst | src + dst + CF → dst | |
| sub | src | dst | dst - src → dst | |
| sbb | src | dst | dst -src -CF → dst | |
| inc | dst | | dst + 1 → dst | OF,SF,ZF,AF,PF |
| dec | dst | | dst - 1 → dst | |
| neg | dst | | -dst → dst | OF,SF,ZF,AF,PF<br>CF=0 if dst==0 |

**Example:**

```
.globl main
main:
    movl   $15,%eax
    subl   $7, %eax # %eax = %eax-7
    addl   $30,%eax # %eax = %eax+30
    decl   %eax     # %eax = %eax-1
    ret
```

# Multiplication/Division (Int)

| Mnemonic | Operand | Operation | Flags |
|---|---|---|---|
| **mul** (unsigned) | reg8 | %al*reg8 → %ax | CF,OF |
| | reg16 | %ax*reg16 → %dx.%ax | |
| | reg32 | %al*reg32 → %eax | |
| **imul** (signed) | reg8 | %al*reg8 → %ax | |
| | reg16 | %ax*reg16 → %dx.%ax | |
| | reg32 | %al*reg32 → %eax | |
| **div** (unsigned) | reg8 | %ax/reg8 → %al ; %ax mod reg8 → %ah | OF,SF, ZF,AF, CF,PF |
| | reg16 | %ax/reg16→%al ; %ax mod %reg16→%ah | |
| | reg32 | %eax/reg32→%eax ; %dx.%ax mod reg32→%dx | |
| **idiv** (signed) | reg8 | %ax/reg8→%al ; %ax mod reg8→%ah | |
| | reg16 | %ax/reg16→%al ; %ax mod reg16→%ah | |
| | reg32 | %ax/reg32→%al | |

# -15*(8/2+3)

```
.globl main

main:
    movl    $8, %eax
    movl    $0, %edx
    movl    $2, %ebx
    divl    %ebx            # %eax = %edx.%eax / %ebx
    addl    $3, %eax        # %eax = %eax+3
    movl    $-15, %ebx
    imull   %ebx            # %eax = -15*%eax

    ret
```

# Jump Operators

| Mnemonic | Operand | Operation | Notes |
|---|---|---|---|
| **jmp** | lbl | jump to lbl | - |
| **ja/jne** | lbl | Jump if above / not below or equal | unsigned operands |
| **jae/jnb** | lbl | Jump if above or equal / not below | |
| **jbe/jna** | lbl | Jump if below or equal / not above | |
| **jb/jnae** | lbl | Jump if below / not above or equal | |
| **jg/jnle** | lbl | Jump if greater / not less or equal | signed operands |
| **jge/jnl** | lbl | Jump if greater or equal / not less | |
| **jle/jng** | lbl | Jump if less or equal / not greater | |
| **jl/jnge** | lbl | Jump if less / not greater or equal | |
| **je/jz** | lbl | Jump if equal / zero (ZF=1) | equality testing |
| **jne/jnz** | lbl | Jump if not equal / not zero (ZF-0) | |
| **jc** | lbl | Jump if (CF=1) | - |
| **jnc** | lbl | Jump if (CF=0) | |
| **js** | lbl | Jump if (SF=1) | |
| **jns** | lbl | Jump if (SF=0) | |

# IF … THEN … ELSE …

```
.globl main

main:
        movl    $8, %ebx
        cmpl    %eax, %ebx      # Compare %eax, %ebx
        jle     L0              # If %eax≤%ebx go to L0
        incl    %ebx            # Increment %ebx (then)
        ret

L0:     decl    %ebx            # Decrement %ebx (else)
        ret
```

# Loops

| Mnemonic | Operand | Operation |
|---|---|---|
| `loop` | lbl | %cx-1→%cx ; if (%cx!=0) jump to lbl |
| `loope` | lbl | %cx-1→%cx ; if (%cx<>0) and (ZF=1) jump to lbl |
| `loopne` | lbl | %cx-1→%cx ; if (%cx<>0) and (ZF=0) jump to lbl |
| `loopz` | lbl | %cx-1→%cx ; if (%cx<>0) and (ZF=1) jump to lbl |
| `loopnz` | lbl | %cx-1→%cx ; if (%cx<>0) and (ZF=0) jump to lbl |

**Example:**

```
.globl main
main:
    movl  $3, %ecx
L0: addl  $5, %eax
    loop  L0

    ret
```

# Floating-point Unit Instruction Set

# Moving Data in FPU

| Mnemonic | Operand | Operation |
|---|---|---|
| `finit` | - | FPU Initialization |
| `fincstp` | - | Increment the FPU stack pointer |
| `fdecstp` | - | Decrement the FPU stack pointer |
| `ffree` | st(i) | Free the content of st(i) |
| `fldz` | - | Load zero in st(0) |
| `fld1` | - | Load one in st(0) |
| `fldpi` | - | Load π in st(0) |
| `fld` | ? | Load a float in st(0) |
| `fild` | ? | Load an int in st(0) |
| `fst` | ? | Write a float in main memory |
| `fstp` | ? | Write a float in main memory and pop |
| `fxch` | ? | Exchange two registers content |

# FPU Arithmetic

| Mnemonic | Operand | Operation |
|---|---|---|
| **`fadd`** | - | st(0)+st(1)→st(0) |
| | mem/st(i) | st(0)+mem/st(i)→st(0) |
| | mem/st(i) mem/st(j) | st(0)+mem/st(i)→st(0) |
| **`fsub`** | - | Similar to **`fadd`** but for substraction |
| **`fmul/fdiv`** | - | Similar to **`fadd`** but for multiplication/division |
| **`fchs`** | - | Change sign |
| **`fabs`** | - | Absolute value |
| **`fsqr`** | - | Square |
| **`fsqrt`** | - | Square root |
| **`fcos`** | - | Cosine |
| **`fyl2x`** | - | $y*\log_2(x)$ |
| **`frndint`** | - | Round to integer value |

# FPU Conditionals

| Mnemonic | Operand | Operand | Operation |
|---|---|---|---|
| **`fcom`** | mem/st(i) | mem/st(j) | Compare two operands, store result in STW |
| **`fcomp`** | mem/st(i) | mem/st(j) | Compare two operands, store result in STW and pop |
| **`fcomi`** | mem/st(i) | mem/st(j) | Compare two int operands, store result in STW |
| **`fcomip`** | mem/st(i) | mem/st(j) | Compare two int operands, store result in STW and pop |

# Interruptions
# &
# System Calls

# Interruptions

## What does an interruption do:

1. Stop current activity of CPU and save the status;
2. Call a specific subroutine (interrupt handler);
3. Depending on the interruption call (0-255), the interrupt handler load an interrupt vector which jumps to the corresponding subroutine;
4. If several interruptions are occurring at the same time, CPU has a priority order to apply;
5. When the subroutine is finished the CPU restore the CPU status and restart previous execution.

# Types of Interruptions

## Internal Hardware Interrupts:

Event occurring during the execution of a program
(e.g. division by zero, overflow, general protection error, ...);

## External Hardware Interrupts:

Event produced by controllers of external devices
(e.g. PCI/AGP bus, hard-drive, graphic cards, keyboard, ...);

## Software Interrupts:

Event produced by programs (mainly by the OS). These
interrupts can be produced by using the instruction **`int`**.

# List of Interruptions (Linux)

| ID | Message |
|---|---|
| 0x00 | Division Error |
| 0x01 | Single Step Mode (Debug) |
| 0x02 | NMI Interrupt |
| 0x03 | Breakpoint |
| 0x04 | Overflow |
| 0x05 | Bound Range Exceeded |
| 0x06 | Invalid Opcode |
| 0x07 | Coprocessor Not Available |
| 0x08 | Double Exception |
| 0x09 | Coprocessor Segment Overrun |
| 0x0a | Invalid Task State Segment |
| 0x0b | Segment Not Present |
| 0x0c | Stack Fault |
| 0x0d | General Protection |
| 0x0e | Page Fault |
| 0x0f | Reserved |
| 0x10 | Coprocessor Error |
| 0x11–0x1f | Reserved |
| 0x12–0xffffff | Coprocessor Error |

# System Calls (Linux)

A system call is a software interrupt tight to a specific subroutine of the OS. (e.g. get a char from keyboard, print on stdout, ...).

- **Arguments:**
  - `%eax`: Syscall ID
  - `%ebx`,`%ecx`,`%edx`,`%esi`,`%edi`: Syscall arguments
- **Calling a syscall:**
  - `int $0x80`
- **Example:**

```
.globl main
main:
   movl  $1,%eax   # Interruption ID
   int       $0x80 # Calling the OS
   ret
```

# Few System Calls (Linux)

| %eax | Name | %ebx | %ecx | %edx | %esi | %edi |
|---|---|---|---|---|---|---|
| 1 | sys_exit | int | - | - | - | - |
| 2 | sys_fork | struct pt_regs | - | - | - | - |
| 3 | sys_read | unsigned int | char* | size_t | - | - |
| 4 | sys_write | unsigned int | const char* | size_t | - | - |
| 5 | sys_open | const char* | int | int | - | - |
| 6 | sys_close | unsigned int | - | - | - | - |
| 7 | sys_waitpid | pid_t | unsigned int | int | - | - |
| 8 | sys_create | const char* | int | - | - | - |
| 9 | sys_link | const char* | const char* | - | - | - |
| 10 | sys_unlink | const char* | - | - | - | - |
| 11 | sys_execve | struct pt_regs | - | - | - | - |
| 12 | sys_chdir | const char* | - | - | - | - |
| 13 | sys_time | int* | - | - | - | - |
| 14 | sys_mknod | const char* | mode_t | dev_t | - | - |
| 15 | sys_chmod | const char* | mode_t | - | - | - |
| ... | ... | ... | ... | ... | ... | ... |

See in linux-2.6.x.y/arch/i386/kernel/syscall_table.S

# Hello World !

```
.data                          # Data section
msg:
    .ascii "Hello World!\n"    # String
    len = . -msg               # String length

.text                          # Text section
    .globl main                # Export entry point to ELF linker

main:
# Write the string to stdout
    movl    $len, %edx         # 3rd argument: string length
    movl    $mgg, %ecx         # 2nd argument: pointer to string
    movl    $1, %ebx           # 1st argument: file handler (stdout)
    movl    $4, %eax           # System call number (sys_write)
    int     $0x80              # Kernel call
# and exit
    movl    $0, %ebx           # 1st argument: exit code
    movl    $1, %eax           # System call number (sys_exit)
    int     $0x80              # Kernel call
```

# Other Instructions

- **`hlt:`**
  Stop the CPU until the next interruption occurs
- **`nop:`**
  No Operation. Used to avoid to stall the pipeline (waiting for data to evaluate a test)
- **`wait:`**
  Deprecated instruction used to wait for the results coming from arithmetic co-processor

# Advanced Instruction Sets

# MMX Instructions

# SSE Instructions

# SSE2 Instructions

# SSE3 Instructions

# 3DNow! Instructions

# Next Time

## From C to Assembly (and back...)