Instituto de Matemática e Estatística

USP

MAC422
Sistemas Operacionais

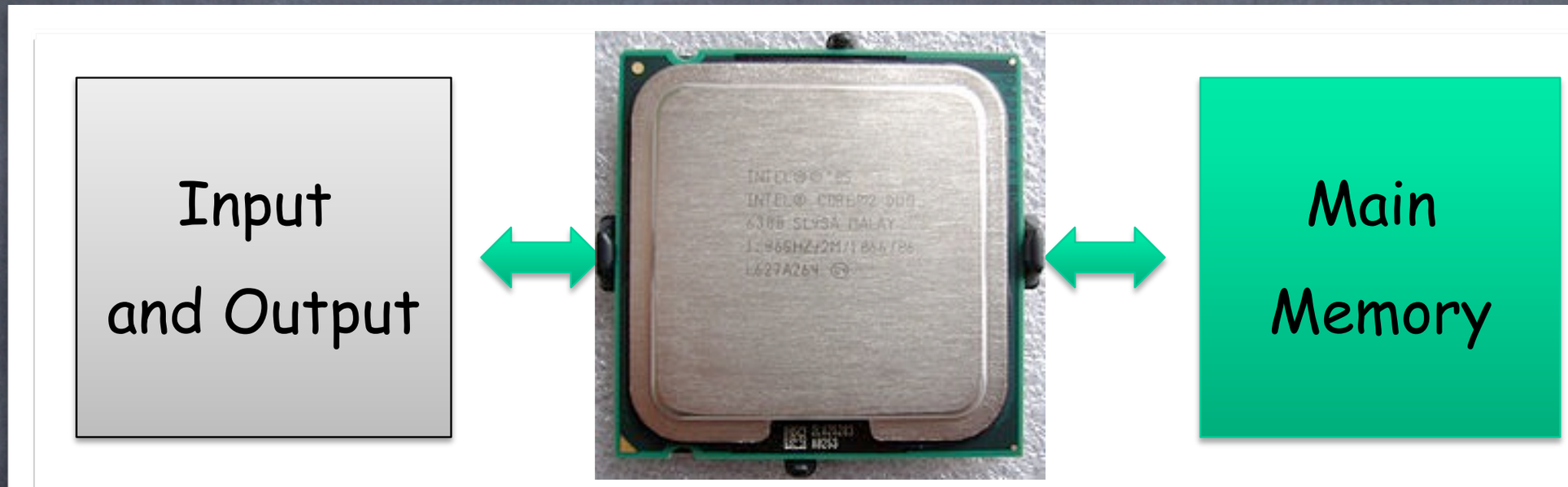
Prof. Marcel P. Jackowski

Aula #2

Hardware e Programação x86

Referências: Sistemas Operacionais Modernos (3a. Ed.), Fundamentos de Sistemas Operacionais; aulas antigas de MAC-422 (IME-USP), W4118 (Columbia University), 6.828 (MIT)

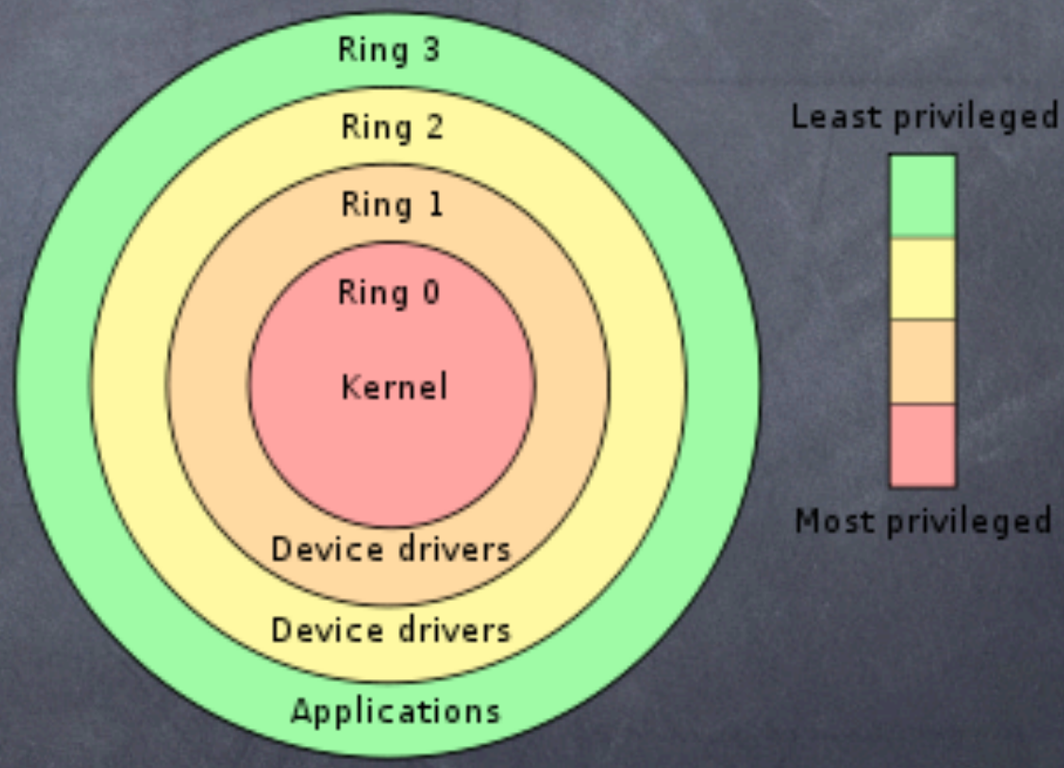
Arquitetura de Von Neumann



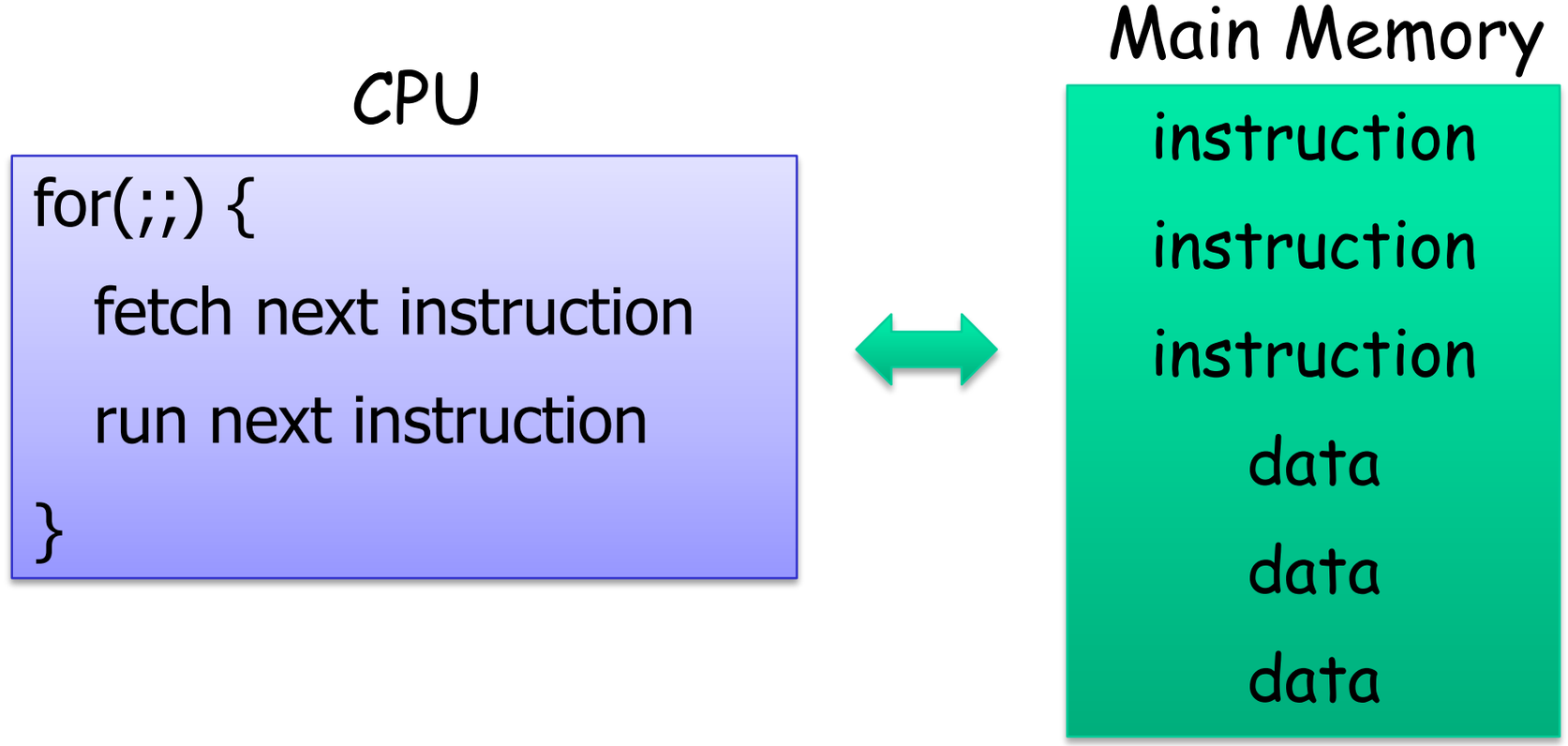
- I/O: communicating data to and from devices
- CPU: digital logic for doing computation
- Memory: N words of B bits

Modos de execução

- **Privilegiado**
  - Monitor, kernel, supervisor, ring 0 ou sistema.
- **Não-privilegiado**
  - Modo usuário ou aplicação.

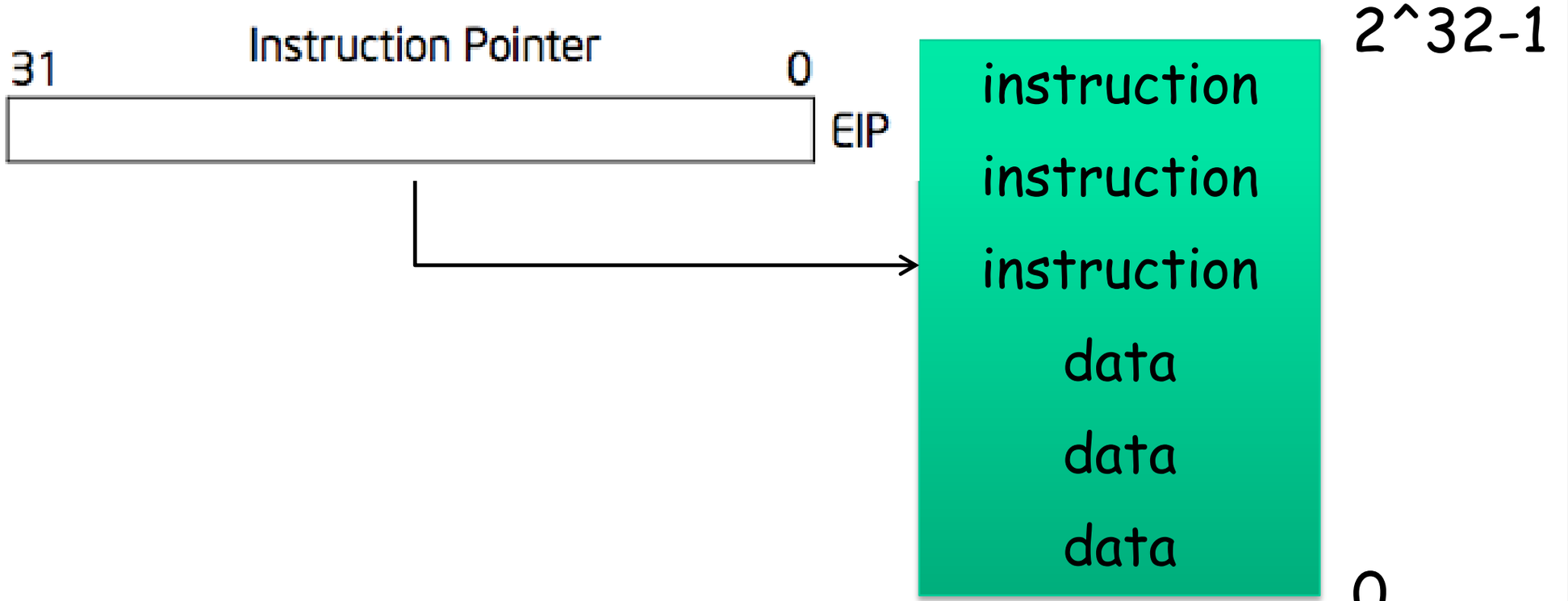


Execução de um programa



- Memory holds both *instructions* and *data*
- CPU interprets instructions
- Instructions read/write data

Contador de programa no x86



- EIP incremented after each instruction
- Variable length instructions
- EIP modified by CALL, RET, JMP, conditional JMP

E em arquiteturas 64 bits ?

Registradores: espaço de trabalho da CPU

General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

- 8, 16, and 32 bit versions
- Example: ADD EAX, 10
  - More: SUB, AND, etc
- By convention some for special purposes

ESP: stack pointer

EBP: frame base pointer

ESI: source index

EDI: destination index

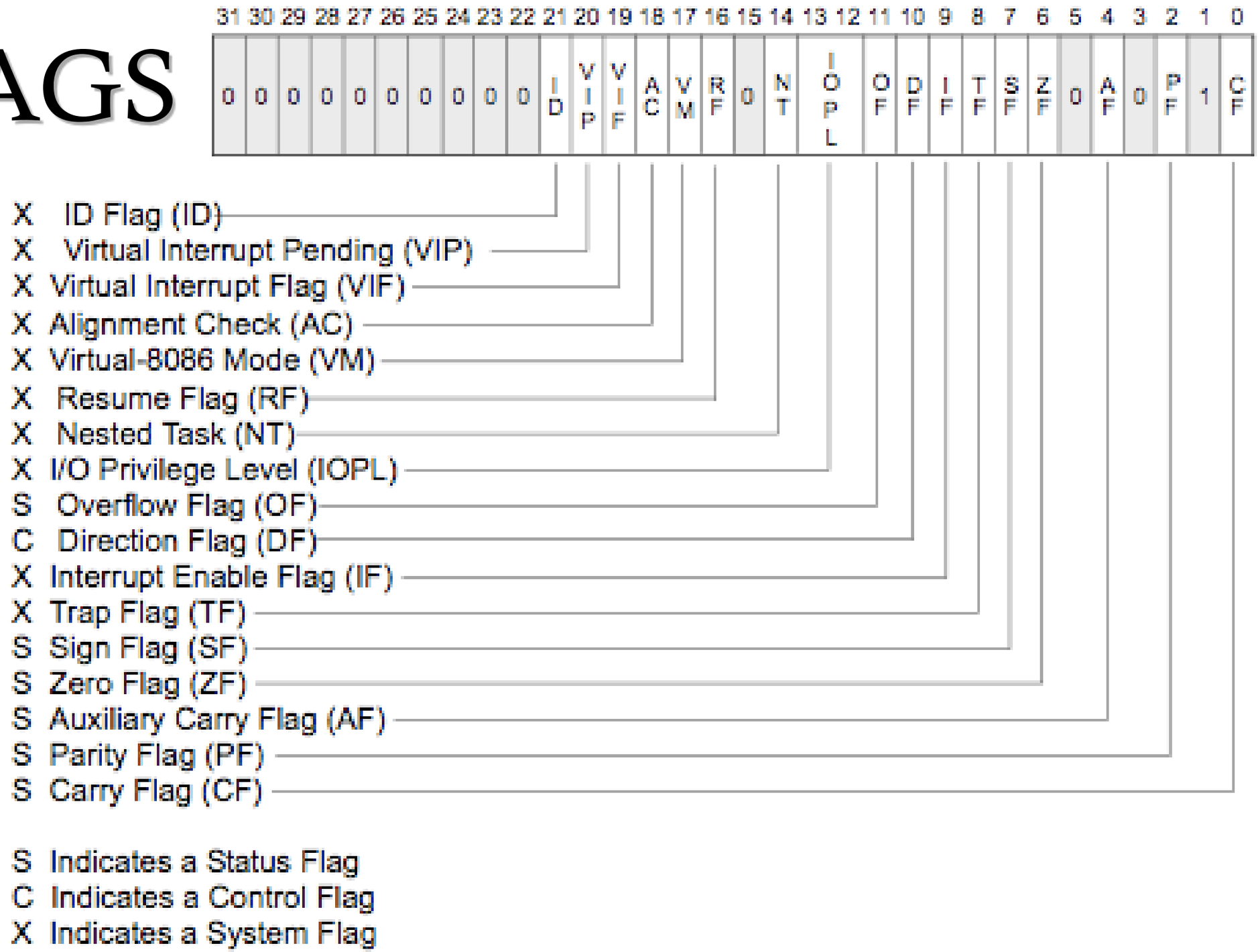
Agora 64 bits

- Exemplo:
  - Acumulador

```
|63..32|31..16|15-8|7-0|
                |AH.|AL.|
                |AX.....|
        |EAX............|
|RAX....................|
```

- Registradores extras:
  - R8-R15: 64 bits

EFLAGS



- Track current CPU status

TEST EAX, 0

JNZ address

Categorias

- Instruction classes
  - Data movement: MOV, PUSH, POP, …
  - Arithmetic: TEST, SHL, ADD, AND, …
  - I/O: IN, OUT, …
  - Control: JMP, JZ, JNZ, CALL, RET
  - String: MOVSB, REP, …
  - System: INT, IRET

- Instruction syntax
  - Intel manual Volumne 2: op dst, src
  - AT&T (gcc/gas): op src, dst
    - op uses suffix b, w, l for 8, 16, 32-bit operands

Sintaxe Intel vs. AT&T

Intel Code	AT&T Code
mov eax,1	movl $1,%eax
mov ebx,0ffh	movl $0xff,%ebx
int 80h	int $0x80
mov ebx, eax	movl %eax, %ebx
mov eax,[ecx]	movl (%ecx),%eax
mov eax,[ebx+3]	movl 3(%ebx),%eax
mov eax,[ebx+20h]	movl 0x20(%ebx),%eax
add eax,[ebx+ecx*2h]	addl (%ebx,%ecx,0x2),%eax
lea eax,[ebx+ecx]	leal (%ebx,%ecx),%eax
sub eax,[ebx+ecx*4h-20h]	subl -0x20(%ebx,%ecx,0x4),%eax

Assembly inline (gcc)

- Can embed assembly code in C code
  - Many examples in xv6

- Basic syntax: asm ("assembly code")

  e.g., `asm ("movl %eax %ebx")`

- Advanced syntax:

```
asm ( assembler template
      : output operands /* optional */
      : input operands  /* optional */
      : list of clobbered registers /* optional */ );
```

e.g.,
```
int val;
asm ("movl %%ebp,%0" : "=r" (val));
```

Mais exemplos

```
int a=10, b;
asm ("movl %1, %%eax;
      movl %%eax, %0;"
      :"=r"(b)          /* output */
      :"r"(a)           /* input */
      :"%eax"           /* clobbered register */
);


asm ("movl %0,%%eax;
      movl %1,%%ecx;
      call _foo"
      : /* no outputs */
      : "g" (from), "g" (to)
      : "eax", "ecx"
);
```

Exemplo de uso dos EFLAGS

```
int main()
{
    int x = 0;
    if (x < 0)
        x++;
    else
        x--;
    return 0;
}
```

gcc -c -S -m32 if.c

```
        .text
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $0, -12(%ebp)
        cmpl    $0, -12(%ebp)
        jns     L2
        incl    -12(%ebp)
        jmp     L4
L2:
        decl    -12(%ebp)
L4:
        movl    $0, %eax
        leave
        ret
        .subsections_via_symbols
```

RAM: maior espaço de trabalho

```
movl %eax, %edx      edx = eax;                      register mode
movl $0x123, %edx    edx = 0x123;                    immediate
movl 0x123, %edx     edx = *(int32_t*)0x123;         direct
movl (%ebx), %edx    edx = *(int32_t*)ebx;           indirect
movl 4(%ebx), %edx   edx = *(int32_t*)(ebx+4);       displaced
```

- Memory instructions: MOV, PUSH, POP, etc
- Most instructions can take a memory address

Pilha: estrutura de dados em RAM

Example instruction	What it does
pushl %eax	subl $4, %esp movl %eax, (%esp)
popl %eax	movl (%esp), %eax addl $4, %esp
call 0x12345	pushl %eip (*) movl $0x12345, %eip (*)
ret	popl %eip (*)

- For implementing function calls
- Stack grows "down" on x86

Alguns detalhes...

- 8086 16-bit register and 20-bit bus addresses
- These extra 4 bits come from *segment register*
  - CS: code segment, for EIP
    - Instruction address: CS * 16 + EIP
  - SS: stack segment, for ESP and EBP
  - DS: data segment for load/store via other registers
  - ES: another data segment, destination for string ops
- Make life more complicated
  - Cannot directly use 16-bit stack address as pointer
  - For a far pointer programmer must specify segment reg
  - Pointer arithmetic and array indexing across seg bound

Mais alguns detalhes...

- 80386: 32 bit register and addresses (1985)
- AMD k8: 64 bit (2003)
  - RAX instead of EAX
  - x86-64, x64, amd64, intel64: all same thing

- Backward compatibility
  - Boots in 16-bit mode; bootasm.S switches to 32
  - Prefix 0x66 gets 32-bit mode instructions
    - MOVW in 32-bit mode = 0x66 + MOVW in 16-bit mode
  - .code32 in bootasm.S tells assembler to insert 0x66

- 80386 also added virtual memory addresses

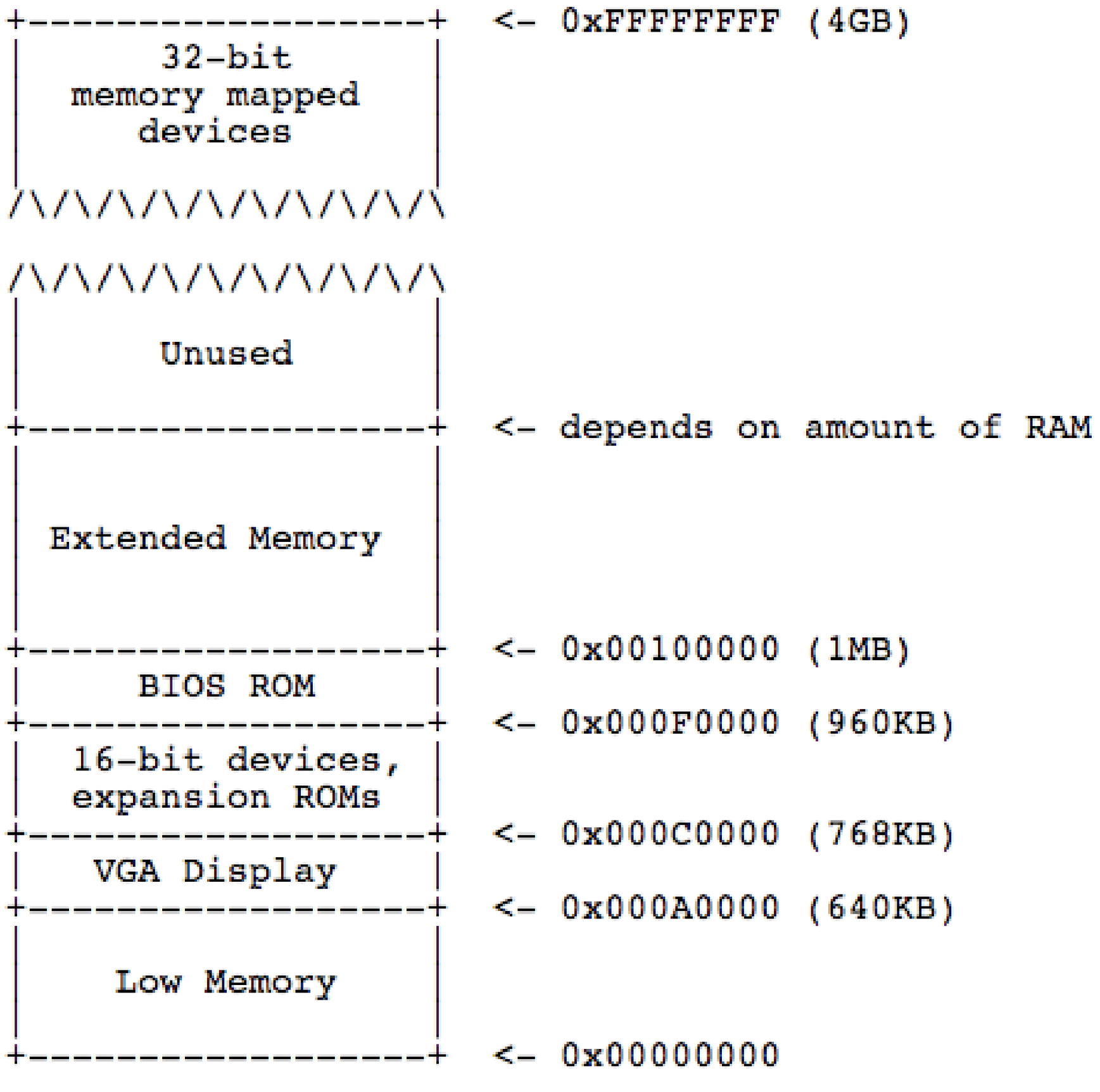
Entrada/Saída

```
#define DATA_PORT    0x378
#define STATUS_PORT  0x379
#define   BUSY 0x80
#define CONTROL_PORT 0x37A
#define   STROBE 0x01
void
lpt_putc(int c)
{
  /* wait for printer to consume previous byte */
  while((inb(STATUS_PORT) & BUSY) == 0)
    ;

  /* put the byte on the parallel lines */
  outb(DATA_PORT, c);

  /* tell the printer to look at the data */
  outb(CONTROL_PORT, STROBE);
  outb(CONTROL_PORT, 0);
}
```

- 8086: only 1024 addresses

E/S mapeada em memória

- Use normal addresses for I/O
  - No special instructions
  - No 1024 limit
  - Hardware routes to device
- Works like "magic" memory
  - I/O device addressed and accessed like memory
  - However, reads and writes have "side effects"
  - Read result can change due to external events


32-bit
memory mapped
devices
Unused
Extended Memory
BIOS ROM
16-bit devices,
expansion ROMs
VGA Display
Low Memory
<- 0xFFFFFFFF (4GB)
<- depends on amount of RAM
<- 0x00100000 (1MB)
<- 0x000F0000 (960KB)
<- 0x000C0000 (768KB)
<- 0x000A0000 (640KB)
<- 0x00000000

Convenções do gcc (i): chamdas de funções

```
1  subprogram_label:
2        push   ebp             ; save original EBP value on stack
3        mov    ebp, esp        ; new EBP = ESP
4  ; subprogram code
5        pop    ebp             ; restore original EBP value
6        ret
```

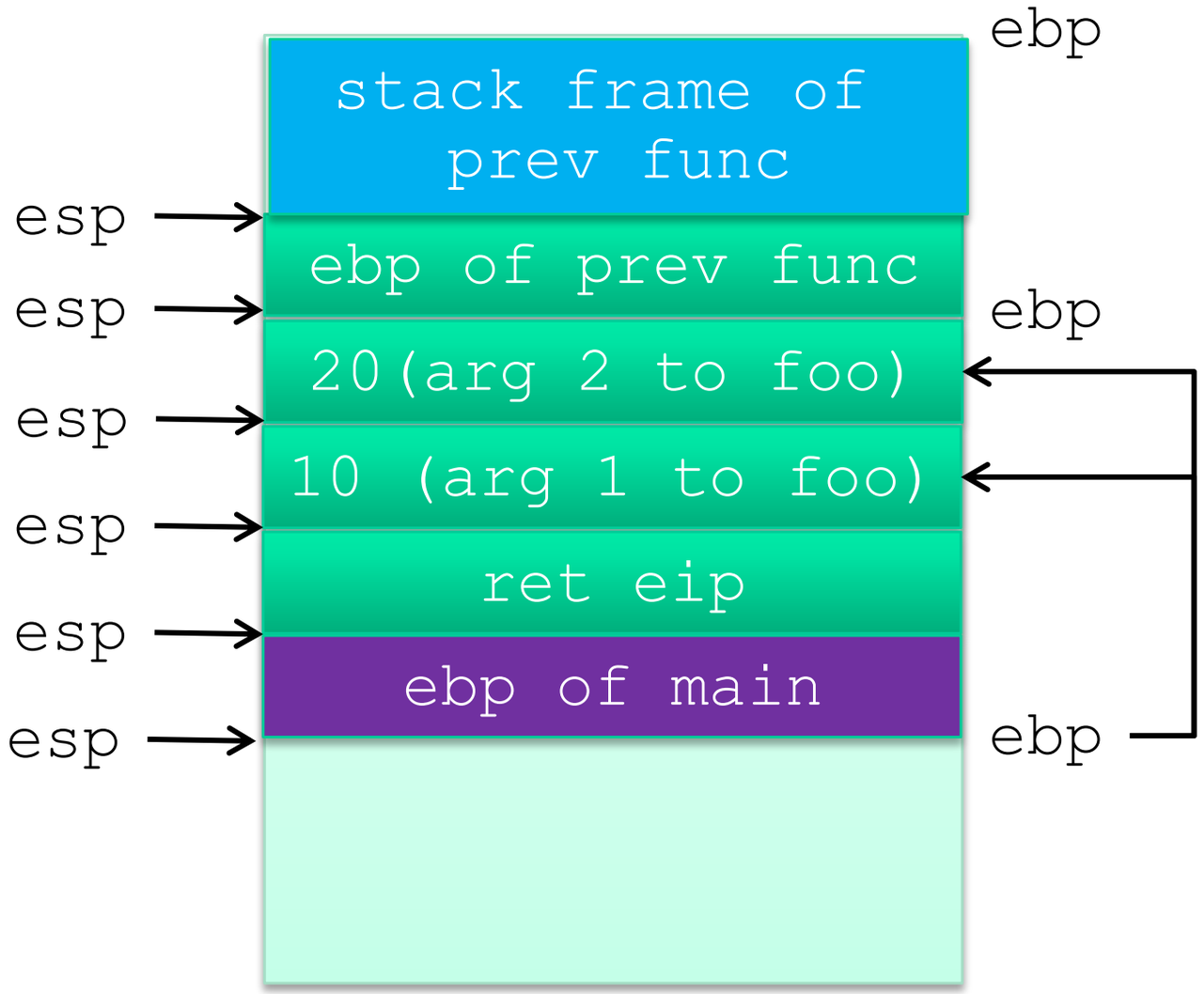
Figure 4.3: General subprogram form

```
1        push   dword 1         ; pass 1 as parameter
2        call   fun
3        add    esp, 4          ; remove parameter from stack
```

Figure 4.5: Sample subprogram call

Exemplo

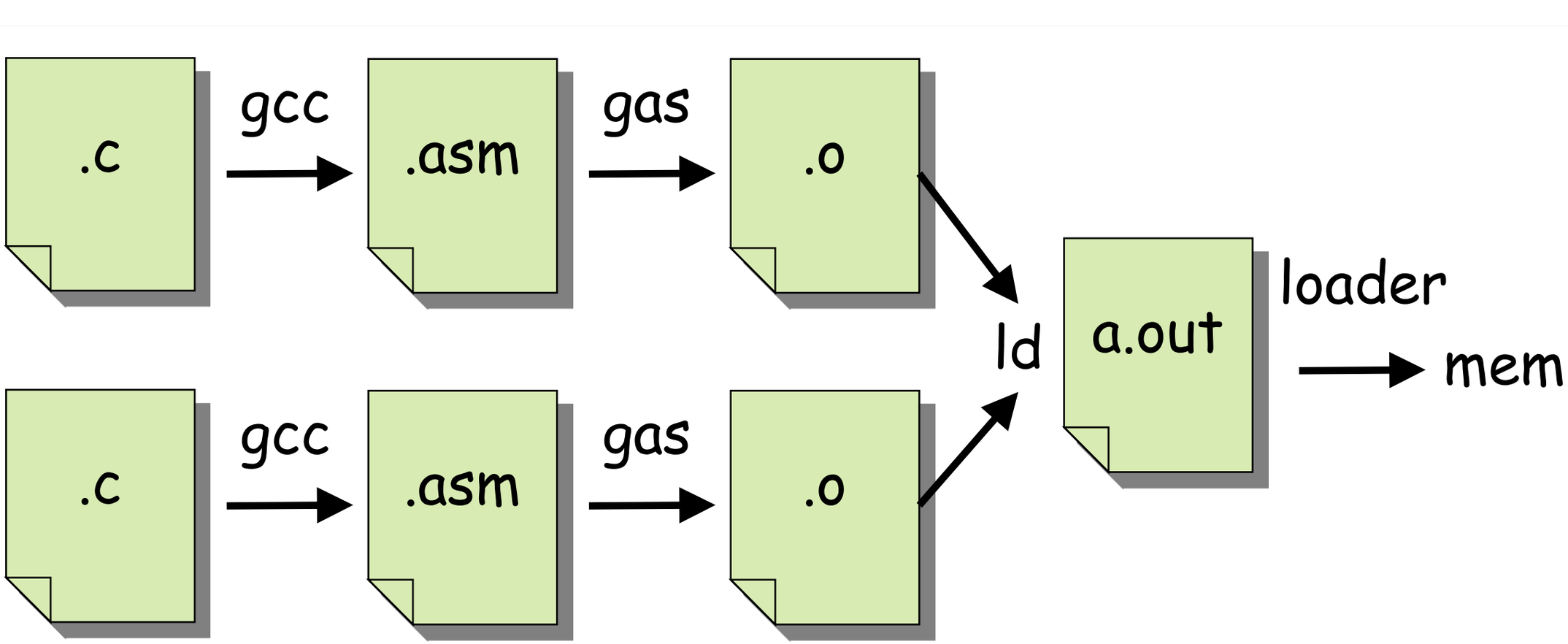
```
main() {
   return foo(10, 20);
}
int foo(int x, inty) {
   return x+y;
}

_main:
   pushl %ebp
   movl %esp, %ebp
   pushl $20
   pushl $10
   call foo
   movl %ebp, %esp
   popl %ebp
   ret

 _foo:
   pushl %ebp
   movl %esp, %ebp
   movl 0xc(%ebp),%eax
   add 0x8(%ebp),%eax
   movl %ebp, %esp
   popl %ebp
   ret
```



Convenções do gcc (ii): registradores

- %eax contains return value, %ecx, %edx may be trashed
  - 64 bit return value: %eax + %edx
- %ebp, %ebx, %esi, %edi must be as before call
- Caller saved: %eax, %ecx, %edx
- Callee saved: %ebp, %ebx, %esi, %edi

Do código-fonte até o executável...



- Compiler, assembler, linker, and loader

Emulador

- QEMU pc emulator
  - Does what a real PC does
  - Except implemented in s/w!
- Run like a normal program on "host" OS

Emulador de Registradores

```
int32_t regs[8];
#define REG_EAX 1;
#define REG_EBX 2;
#define REG_ECX 3;
...
int32_t eip;
int16_t segregs[4];
...
```

Emulador de lógica da CPU

```
for (;;) {
        read_instruction();
        switch (decode_instruction_opcode()) {
        case OPCODE_ADD:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] + regs[src];
                break;
        case OPCODE_SUB:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] - regs[src];
                break;
        ...
        }
        eip += instruction_length;
}
```

Emulador de memória

```
uint8_t read_byte(uint32_t phys_addr) {
        if (phys_addr < LOW_MEMORY)
                return low_mem[phys_addr];
        else if (phys_addr >= 960*KB && phys_addr < 1*MB)
                return rom_bios[phys_addr - 960*KB];
        else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
                return ext_mem[phys_addr-1*MB];
        else ...
}

void write_byte(uint32_t phys_addr, uint8_t val) {
        if (phys_addr < LOW_MEMORY)
                low_mem[phys_addr] = val;
        else if (phys_addr >= 960*KB && phys_addr < 1*MB)
                ; /* ignore attempted write to ROM! */
        else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
                ext_mem[phys_addr-1*MB] = val;
        else ...
}
```