

Assembly Language Programming

When to program in assembly language?

Never, unless:

1. This is unavoidable (system, inter-language “glue”, viruses)
2. You think this is fun (e.g. hacking)

NOT a “programming language” per se.

This is just a slightly less painful way to program in machine language.

Every processor type has a different machine language, so a different assembly language!

Must understand the specific computer architecture to program in assembly.

This course: Intel 80X86 (32bit) architecture

Computer Architecture

Assembly Language

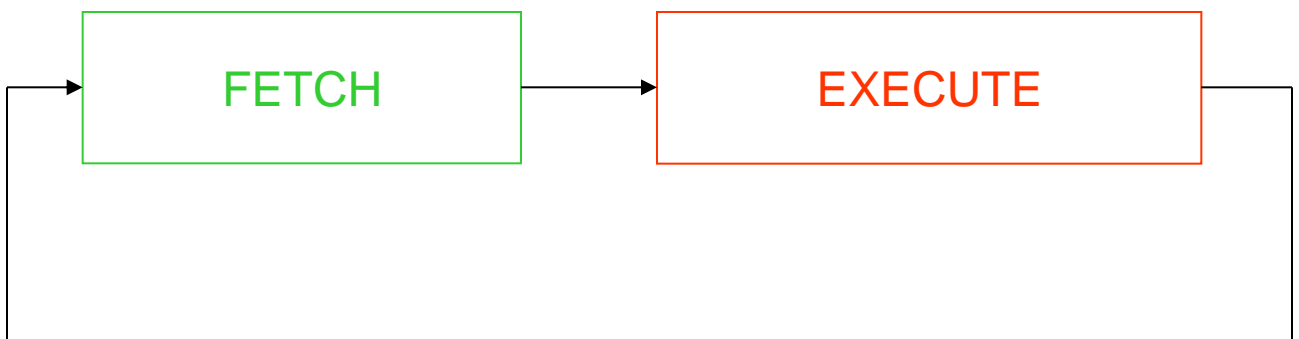
Computer executes a PROGRAM stored in MEMORY.

Basic scheme is - DO FOREVER:

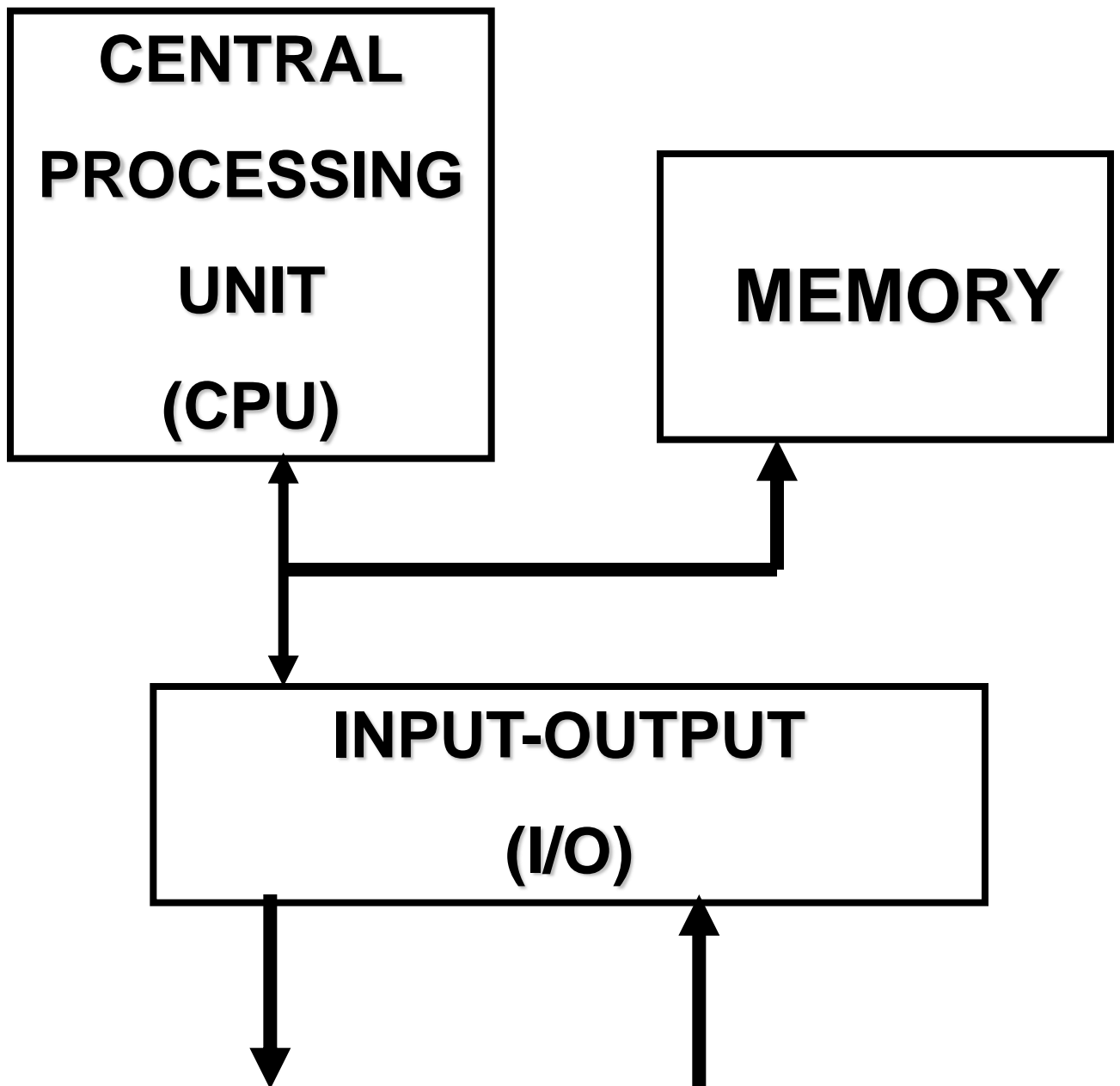
1. FETCH an instruction (from memory).
2. EXECUTE the instruction.

This is the FETCH-EXECUTE cycle.

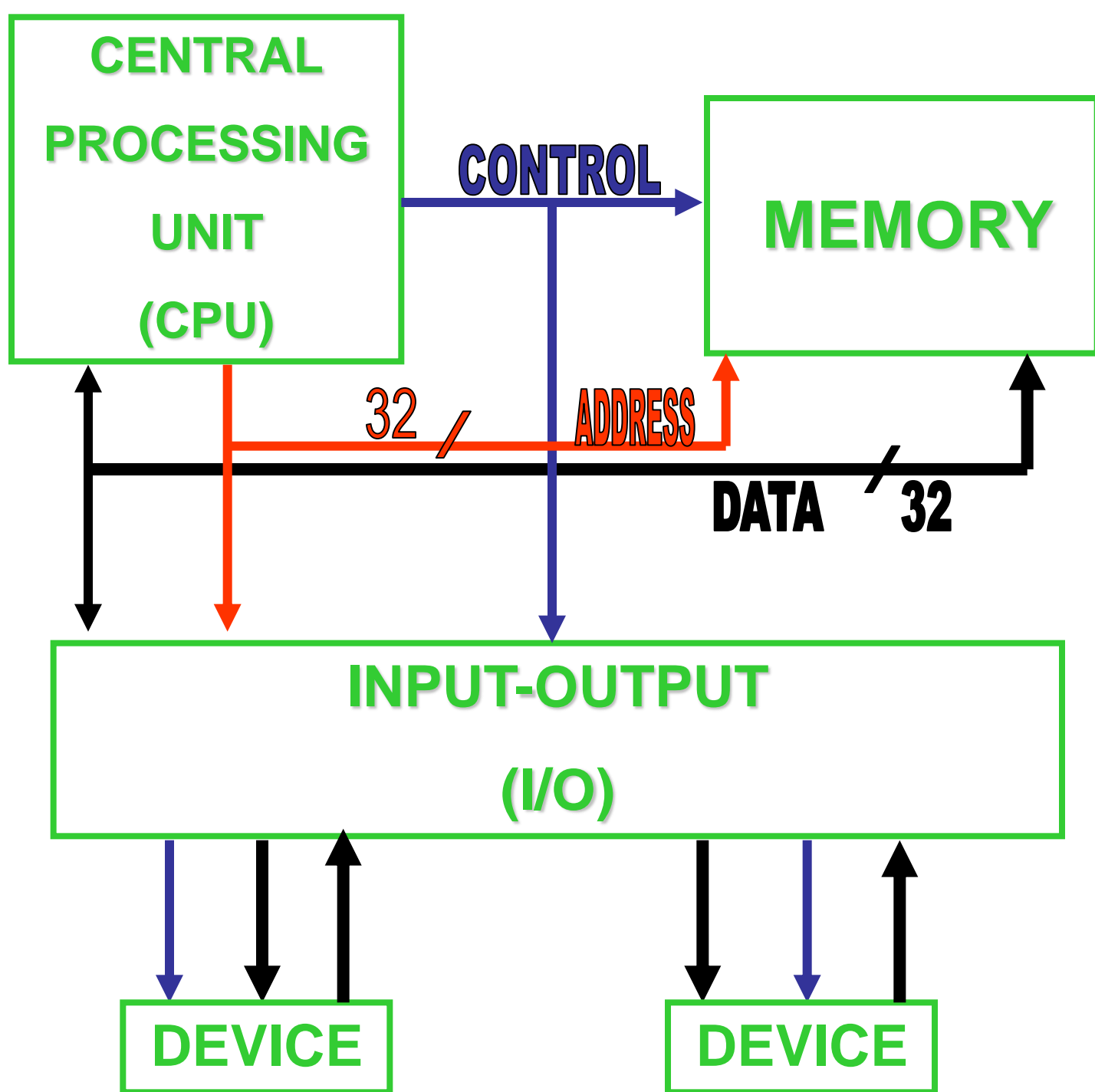
More complicated in REAL machines (e.g. interrupts).



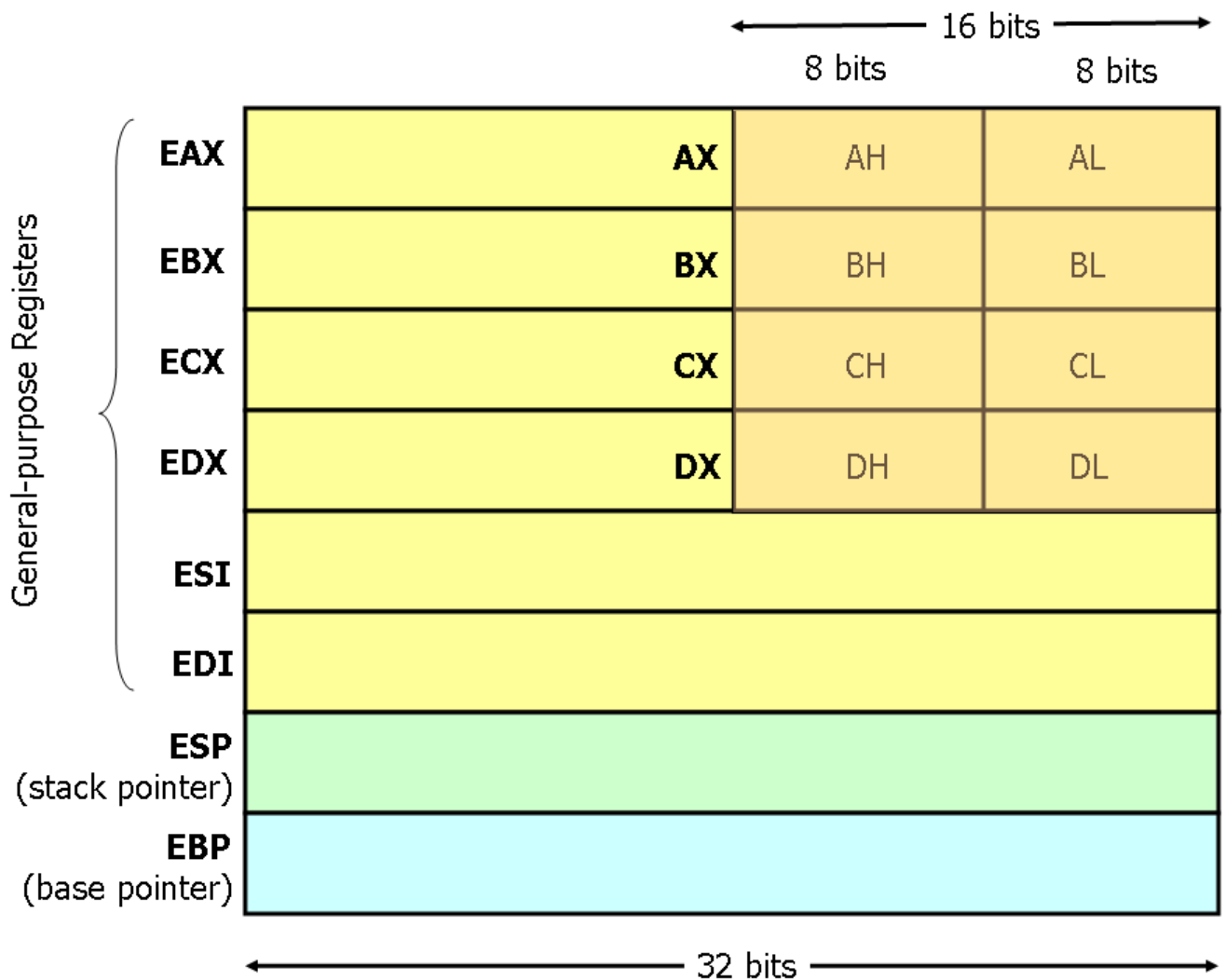
Block Diagram of a Computer



Refined Block Diagram



Registers (80X86)



Not shown above:

32-bit instruction pointer (IP) cannot access directly

16-bit segment regs (CS,SS,DS,ES,...)



not used in this course

Flags (80X86)

Each FLAG represents a BIT of important information:

MACHINE STATUS (error, interrupt, mode)

COMPUTATION STATUS:

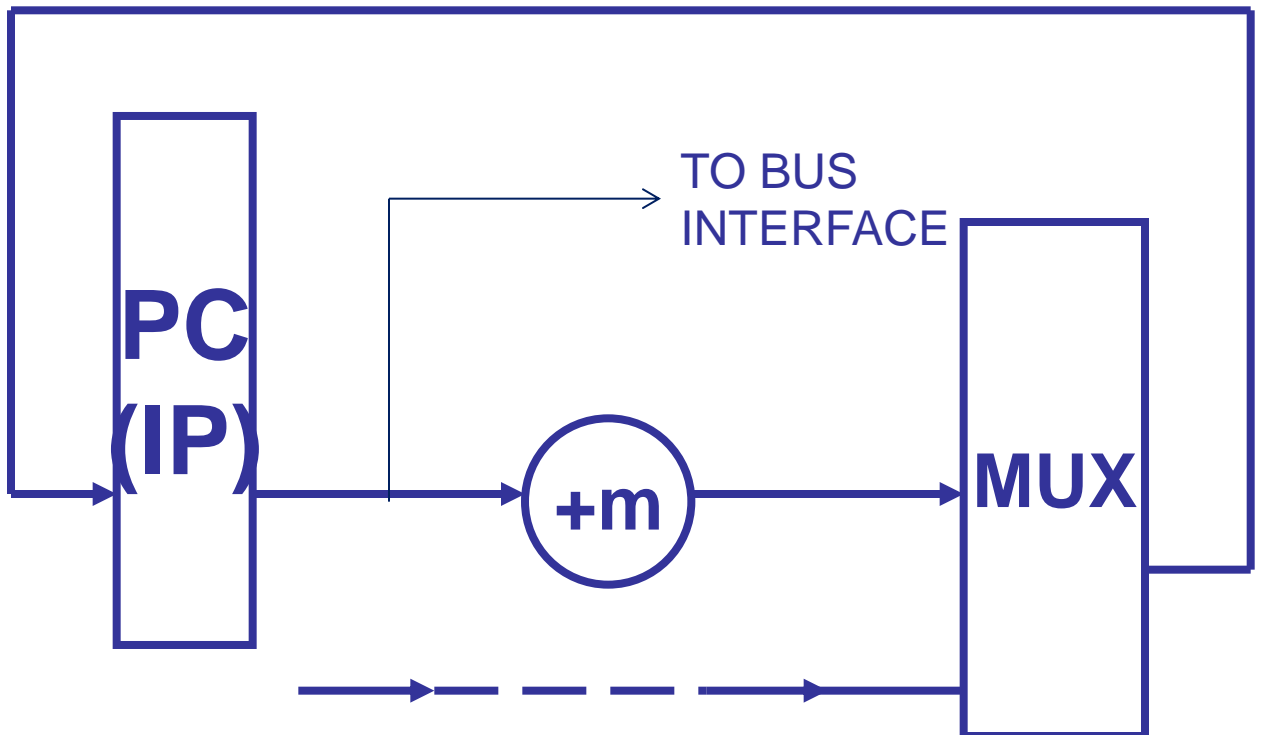
carry (CF), overflow (OF), zero (ZF), sign (SF), parity (PF)  

Computation status flags changed by some instructions (inc, cmp..), used to control conditional jumps.

Instruction Sequencing

Instructions usually fetched from consecutive memory locations, determined by INSTRUCTION POINTER (IP)

Except for JUMP, CALL, or INTERRUPT.



Programming in Assembly Language

Assembly language programming:
Don't unless you have to!

ASSEMBLY LANGUAGE is (almost) 1
to 1 with MACHINE INSTRUCTIONS

Assembly language constructs are:

- Symbolic version of **machine instructions**
- **Labels** (standing for constants and memory addresses)
- **Pseudo-operations** (directives)

Also contains:

- Comments, which are optional
- Macros, skipped in this course

Example NASM Assembly Language Program

```
global my_func      ; This is a comment
```

```
extern printf
```

```
section .rodata
```



```
Str1: db "I am alive", 10, 0
```

```
Str2: db "My lucky number: %d", 10, 0
```

```
num1: dd 15      ; int num1=15;
```

```
section .text
```

```
my_func:
```

```
    pushad
```

```
    push     dword Str1
```

```
    call     printf ; printf(Str1);
```

```
    add     esp, 4
```

```
    push     dword [num1]
```

```
    push     dword Str2
```

```
    call     printf ; printf(Str2, num1);
```

```
    add     esp, 8
```

```
    popad
```

```
    ret
```

The Assembler

ASSEMBLER converts “blah.s” program, creating “blah.o” object file in 2 passes:

- Pass I: translate symbolic instructions into binary code, create SYMBOL TABLE of labels.
- Pass II: translate labels into (relocatable) addresses, fix binary code, and create object file with relocation information.

Object file is a “binary” file in some format (ELF in this course).

Optionally use “-l filename” to generate “listing” file, shown next.

Listing File Example

```
1          global my_func      ; This is a comment
2          extern printf
3          section .rodata
4 00000000 4920616D20616C6976- Str1: db "I am alive", 10, 0
5 00000009 650A00
6 0000000C 4D79206C75636B7920- Str2: db "My lucky number: %d", 10, 0
7 00000015 6E756D6265723A2025-
8 0000001E 640A00
9 00000021 0F000000          num1: dd 15      ; int num1=15;
10         section .text
11         my_func:
12 00000000 60          pushad
13 00000001 68[00000000]      push dword Str1
14 00000006 E8(00000000)      call printf ; printf(Str1);
15 0000000B 83C404          add esp, 4
16 0000000E FF35[21000000]      push dword [num1]
17 00000014 68[0C000000]      push dword Str2
18 00000019 E8(00000000)      call printf ; printf(Str2, num1);
19 0000001E 83C408          add esp, 8
20 00000021 61          popad
21 00000022 C3          ret
```

Assembler Directives

section <section name>

following code/data go to <section name>

common names: `.text` `.data` `.bss` `.rodata`

db <number>,... reserve bytes and initialize

can use a “string” as list of numbers

dw <number>,... reserve “words” and init.

dd <number>,... reserve “dwords” and init.

note: “word” is 2 bytes, “dword” is 4 bytes !?!

By default, <number> in decimal.

Can use e.g. 0x1F (hexadecimal), ‘z’ (ASCII)

resb <number> reserve <number> bytes

global <name> make label visible outside

extern <name> label defined elsewhere

<name> EQU <number> name a constant

Labels (names) in Assembly Language

`<name>:` defines the label `<name>`
its “value” is (usually) the “current”
memory address.

Label `<name>` can be used as a number
anywhere else.

Examples:

`Num: ;` defines the label (in data or code)

`dd Num ;` initializes data to Num

`push dword Num ;` push Num on stack

`call Num ;` call function code at Num

(Assembler/linker will use actual memory
address in generated code/data)

Machine Instructions

Come in different categories:

- Data transfer
- Arithmetic/logic/shift
- Flow control (control transfer)
- System control
- Floating point

Most instructions have **operands**: the data operated on by the instruction.

In this course we only cover a small subset of some of the categories.

See e.g. NASM manual or manufacturer's datasheet for full set.

Data Transfer Instructions

MOV [size] <destination>, <source>

copy <source> to <destination>

size is one of: byte, word, dword

<source> is: immediate, register, memory

<destination> is: register, memory

Memory: not both source and destination

(Rules are for most 2-operand instructions,
size of source and destination: the same)

MOV  eax, 55

MOV byte  [label], 55

MOV ebx, eax

Interlude: Addressing Modes

IMMEDIATE (operation on constants)

decimal, hex, char, label

```
MOV al, 49
```

```
MOV al, 0x31
```

```
MOV al, '1'
```

(all above are actually the same code!)

```
MOV eax, printf
```

place in eax value of symbol printf
(which is address of function printf)

REGISTER

(operation on registers: temp “variables”)

Addressing Modes (cont)

ABSOLUTE (global variables): **[label]**

operation on memory at constant address

section .data

x: db 55

y: dw 0x1111

z: db 0x55

“Same” as C code (in global declaration)

char x=55;

short y=0x1111;

char z=0x55;

section .text

MOV byte [x], 0

“Same” as C code: x=0;

CAVEAT: no types or checking !!!

MOV dword [x], 0

Legal! assigns 0 to x, y, and z !!!

Addressing Modes (cont)

REGISTER INDIRECT (access by pointer)

`MOV byte [ebx], 0`

Sets value of byte at memory address specified by ebx register to 0

Fake C code: `*ebx = 0;`

CAVEAT: no types or checking !!!

Note that to access using a pointer, must place pointer in register first.

Obviously, recall warning on access through uninitialized pointer...

Addressing Modes (cont)

REGISTER INDIRECT + DISPLACEMENT
(structure element access)

```
struct my_struct {int field1; int field2};  
struct my_struct *p; /* global declarations */  
p=malloc(8); /* in some function */
```

To access a structure element, say in C:

```
p->field2 = 666;
```

Do in assembly language:

```
MOV eax, [p] ; place pointer in eax
```

```
MOV dword [eax+4], 666
```

4 is the displacement after structure start

Type checking: you must be kidding!

This is 100% programmer's responsibility.

Addressing Modes (cont)

INDEXED (array element access)

syntax: [label +<register>*<size>]

```
int index, my_array[10]; /* global decl */
```

To access an array element, say in C:

```
my_array[i] = 666;
```

Do in assembly language:

```
MOV dword eax, [index]
```

```
MOV dword [my_array+eax*4], 666
```

Machine only allows multipliers of 1, 2, 4,
(and 8, but the latter is a secret)

Type checking: what, you are still asking?

Transfer: PUSH and POP

PUSH [size] <source>

push <source> onto stack:

equivalent to: $ESP := ESP - (\text{size in bytes})$

MOV [size] [ESP], <source>

size is one of: word, dword

PUSH dword 55

PUSH eax

POP [size] <destination>

pop from stack to <destination>

equivalent to: MOV [size] <source>, [ESP]

$ESP := ESP + (\text{size in bytes})$

POP eax

POP dword [eax+8]

POP dword [label+4*eax]

PUSHAD ; Push all 32 bit registers

POPAD ; Pop all 32 bit registers

Arithmetic/Logic/Shift

ADD [size] <destination>, <source>

Adds <source> into <destination>

ADD eax, ebx

ADD eax, [ebp+12]

ADD dword [ebp-4], 666

Most arithmetic instructions affect flags:

CF: true if result has a carry

SF: true if MSB is 1 (“negative”)

ZF: true if result is zero

OF: true if carry into MSB

PF: parity of bits (only of LSbyte)

ADC [size] <destination>, <source>

Adds <source>+CF into <destination>

Can be used to “extend” an addition:

More Arithmetic Instructions

SUB [size] <dest>, <source>

<dest> := <dest> - <source>

SUB eax, ebx

SUB eax, [ebp+12]

SUB dword [ebp-4], 666

Affects all flags: CF, OF, ZF, SF, PF

CMP [size] <dest>, <source>

“compare”: same as SUB, but affects ONLY flags

SBB [size] <dest>, <source>

Subtracts with borrow (CF)

Can be used to “extend” a subtraction

More Arithmetic Instructions

INC [size] <dest>

Value of <dest> increases by 1

INC eax

INC dword [ebp+12]

INC dword [x]

Affects all flags except CF

DEC [size] <dest>

Value of <dest> decreases by 1

DEC eax

DEC dword [ebp+12]

DEC dword [x]

Affects all flags except CF

"Logic" Instructions

AND [size] <dest>, <source>

Bitwise AND <source> into <dest>

Affects all flags (CF and OF always 0)

OR [size] <dest>, <source>

Bitwise OR <source> into <dest>

Affects all flags (CF and OF always 0)

XOR [size] <dest>, <source>

Bitwise XOR <source> into <dest>

Affects all flags (CF and OF always 0)

NOT [size] <dest>

Inverts all bits of <dest>

Flags not affected

Shift Instructions

SHR [size] <dest>, <number>

Shift <dest> bits by

<number> positions to the right

“lost” bit goes to CF for <number>=1

SHR dword [eax], 1

SHL [size] <dest>, <number>

Shift <dest> bits by

<number> positions to the left

“lost” bit goes to CF for <number>=1

ROL [size] <dest>, <number>

Same as SHL but with “wraparound”

Other shift instructions:

ROR, RCR, RCL, ASR

See NASM manual

Control Transfer Instructions

JMP <label> ; Jump to <label>

JMP Next

Jcc <label> ; If cc true: jump to <label>

cc can be any “condition” of flags:

C, NC, Z, NZ, PE, PO, S, NS, or

“arithmetic”: E, B, A, BE, AE, GT, LT, GE

Must set the flags before conditional jump
using some other instruction, e.g. **CMP**

JNZ Next ; If ZF false, jump to Next

CALL <label> ; Call function <label>

; (push return address (=IP), then jump)

CALL printf

RET ; Return from function

; (pop IP from stack)

System Control

Many such. Only one needed in course:

INT <vector number>

Saves state (IP, PSW) on interrupt stack,
changes machine mode to “kernel”

jumps to (OS) code as defined by interrupt
vector number.

We will (intentionally) use only:

INT 0x80 ; Linux system service call 

Transfers control to Linux, service request
number determined by eax. Example:

MOV eax, 1

MOV ebx, 0

INT 0x80

Calls “exit” system service, exit code in
ebx (0 is “normal termination”)

Interrupt can be caused by error or
hardware event, e.g. “segmentation fault”.

Calling Conventions

Languages have agreed schemes to pass arguments to functions, and return values, these are **calling conventions**.

In **assembly language**, can do **anything**.

However, e.g. linux system call

INT 0x80

arguments in **registers**, (eax, ebx, ...)

return value (if any) in **eax**.

In C (32-bit Linux):

x=foo(a,*b,&c);

arguments on **stack**,

(values pushed starting from rightmost)

return value (if any) in **eax**.

C: Calling a Function

`x=foo(a,*b,&c); /* assume all global int */`

Equivalent code in assembly language

`PUSH dword c`

`MOV eax, [b] /* retrieve b into reg */`

`PUSH dword [eax]`

`PUSH dword [a]`

`CALL foo`

`ADD esp, 12 /* clean up stack */`

`MOV dword [x], eax /* Assign to x */`

C: Called Function Code

```
foo(int a, int b, int *c) {  
    int x=0;  
    *c=a+b;  
    return (c);  
}
```

Equivalent code in assembly language

```
PUSH    ebp        /* function entry code */  
MOV     ebp, esp  
SUB     esp, 4      /* allocate for local var */  
MOV     dword [ebp-4], 0 /* local x=0; */  
MOV     ecx, [ebp+8] /* get 1st arg, a */  
ADD     ecx, [ebp+12] /* add 2nd arg, b */  
MOV     eax, [ebp+16] /* get 3rd arg. c */  
MOV     [eax], ecx   /* store in *c */  
MOV     eax, ecx     /* set up return value */  
MOV     esp, ebp     /* deallocate locals */  
POP     ebp          /* function exit code */  
RET
```

Programming in Assembly Language

OK, so we have to, what do we do?

1. Write algorithm in pseudocode (or C)
2. Determine **exact** steps: **what** storage locations need to be changed, and **how**
3. Find instructions to do each step
4. In some cases, can use prescriptions, e.g. calling C functions (use calling convention).

In remainder of lecture, some examples, until time runs out.

Example NASM Program (Revisited)

```
int num1=15;
void my_func( ) {
    printf("I am alive\n");
    printf("My lucky number: %d\n", num1); }
```

```
global my_func    ; This is a comment
extern printf
section .rodata
    Str1: db "I am alive", 10, 0
    Str2: db "My lucky number: %d", 10, 0
section .data
    num1: dd 15    ; int num1=15;
section .text
my_func:
    pushad
    push dword Str1
    call  printf    ; printf(Str1);
    add  esp, 4
    push dword [num1]
    push dword Str2
    call  printf    ; printf(Str2, num1);
    add  esp, 8
    popad
    ret
```

Example: Multi-Precision Addition

```
multi_len EQU 2
```

```
global multi_add
```

```
global x_struct
```

```
global y_struct
```

```
section .data
```

```
    x_struct: x_len: dd multi_len
```

```
        x_val: resd multi_len
```

```
    y_struct: y_len: dd multi_len    ; Assume equal len
```

```
        y_val: resd multi_len
```



```
section .text
```

```
multi_add: pushad    ; pusha, make sure is in dwords
```

```
        mov     ecx, 0        ; set index to 0
```

```
        mov     edx, [x_len]
```

```
        and     al, al        ; clear CF
```

```
do_rep:  mov     eax, [y_val+ecx*4]
```

```
        adc     [x_val+ecx*4], eax
```

```
        inc     ecx          ; prepare for next
```

```
        dec     edx          ; are we done?
```

```
        jnz     do_rep
```

```
        popad
```

```
        ret
```

Output String to stdout, NO STDLIB

WRITE EQU 4

STDOUT EQU 1

global my_puts ; void my_puts(char *p);

section .text

```
my_puts: push ebp
         mov  ebp, esp
         pushad
         mov  ecx, [ebp+8] ; Get first argument p
         call my_strlen
         mov  ecx, [ebp+8] ; Get first argument
         mov  edx, eax      ; Count of bytes
         mov  ebx, STDOUT
         mov  eax, WRITE
         int  0x80          ; Linux system call
         popad
         mov  esp, ebp
         pop  ebp
         ret
```

```
my_strlen: mov  eax, 1
```

```
cont:     cmp  byte [ecx], 0
```

```
         jz   done
```

```
         inc  ecx
```

```
         inc  eax
```

```
         jmp  cont
```

```
done:     ret
```