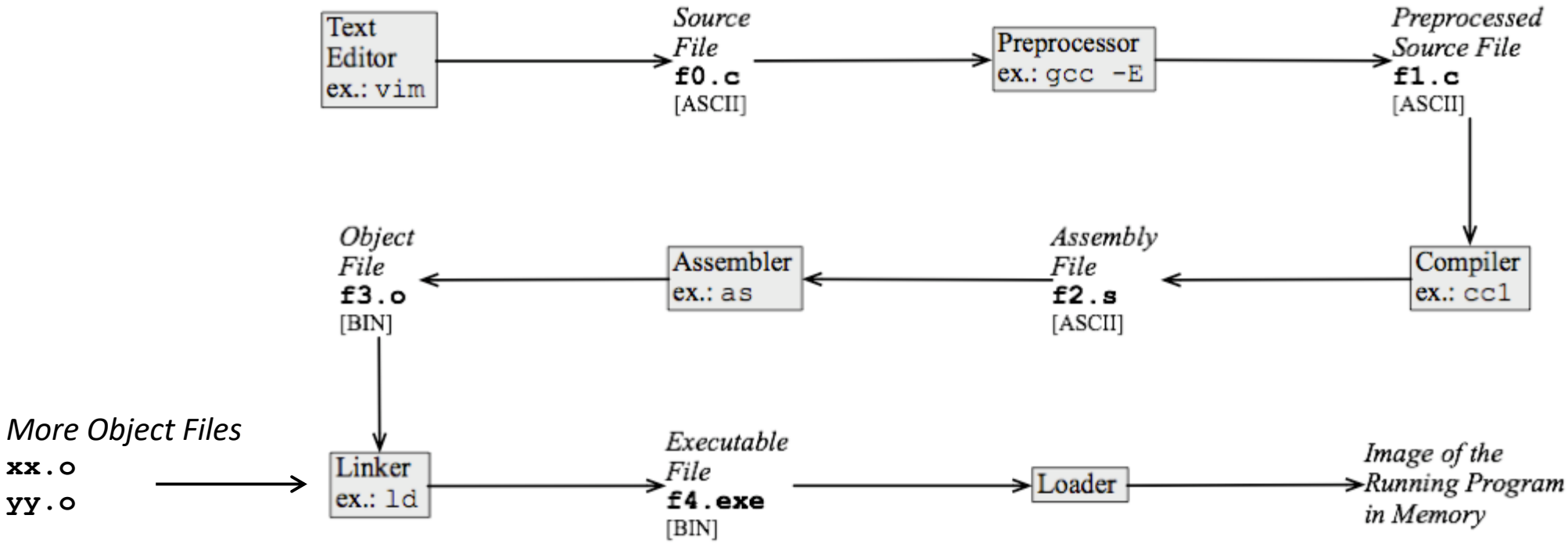
Assembly x86

Quick Guide for the Reverse Engineering

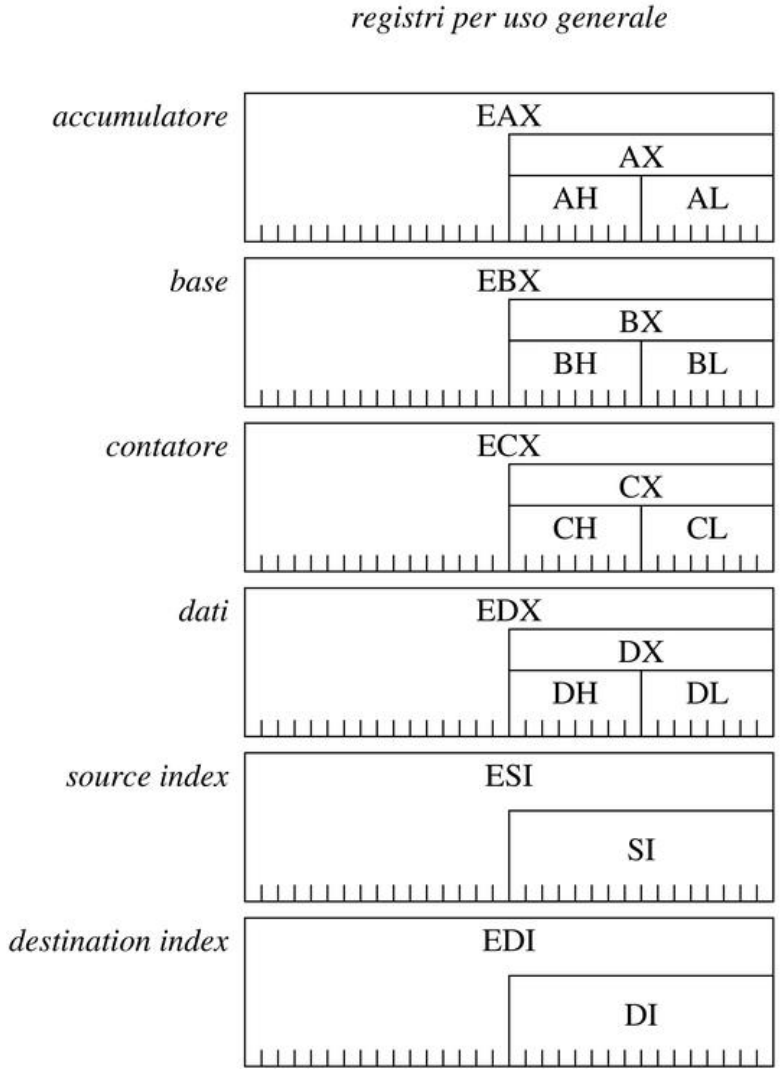
Luigi Coppolino

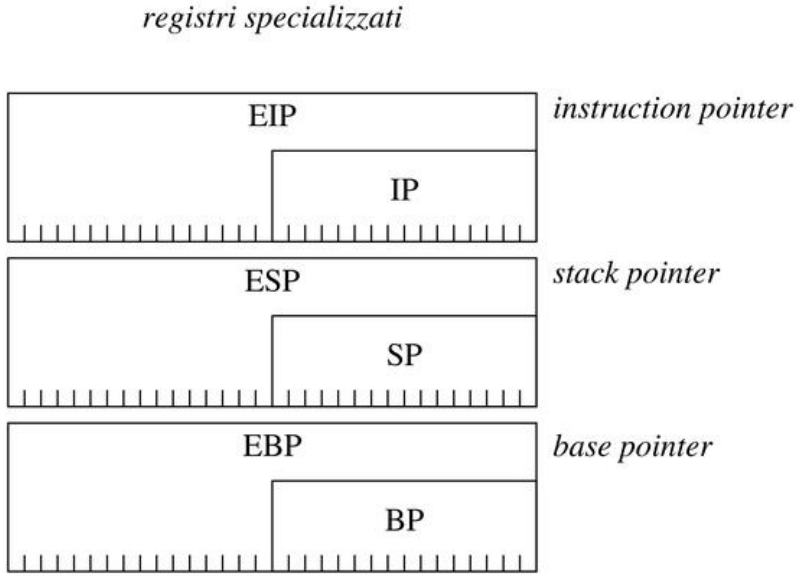
software development process

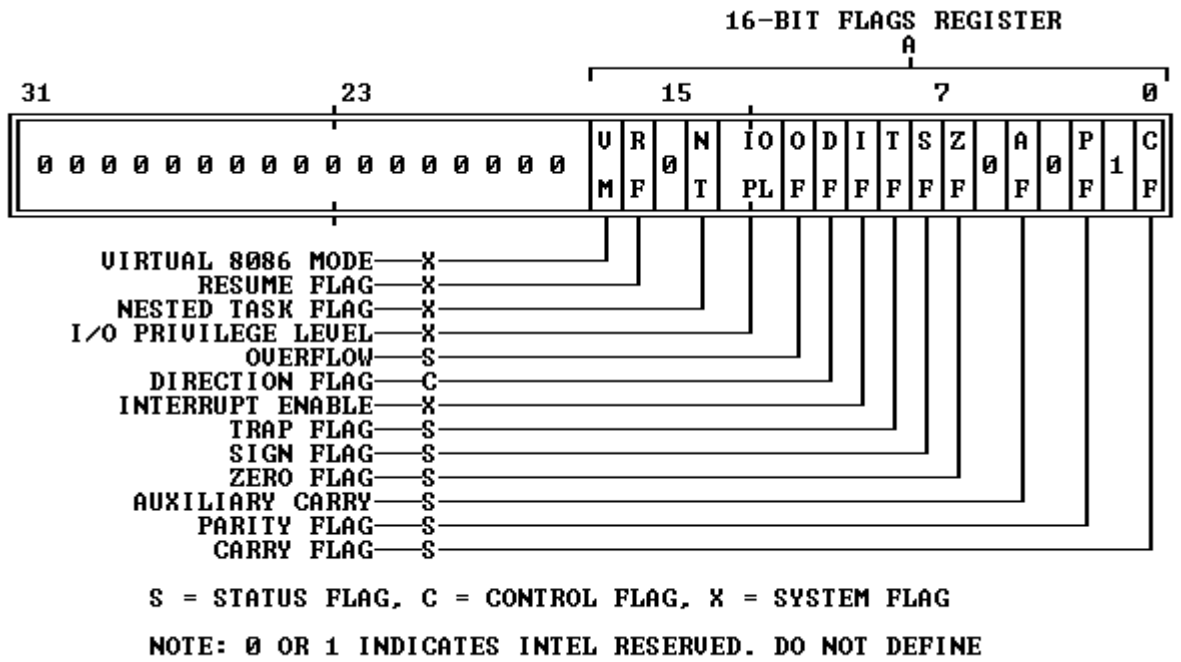
- *Object*

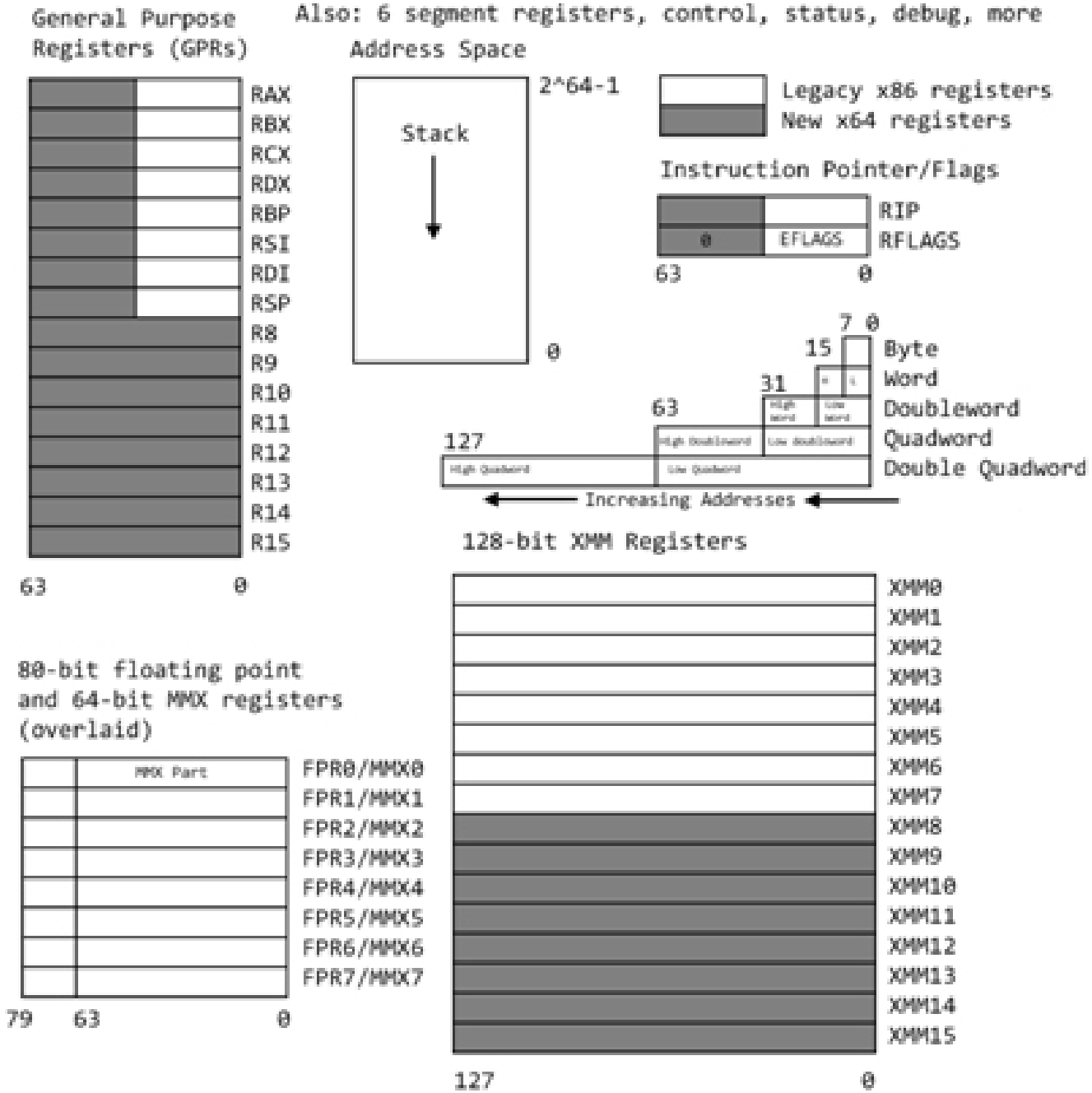


Registers x86 (IA-32)

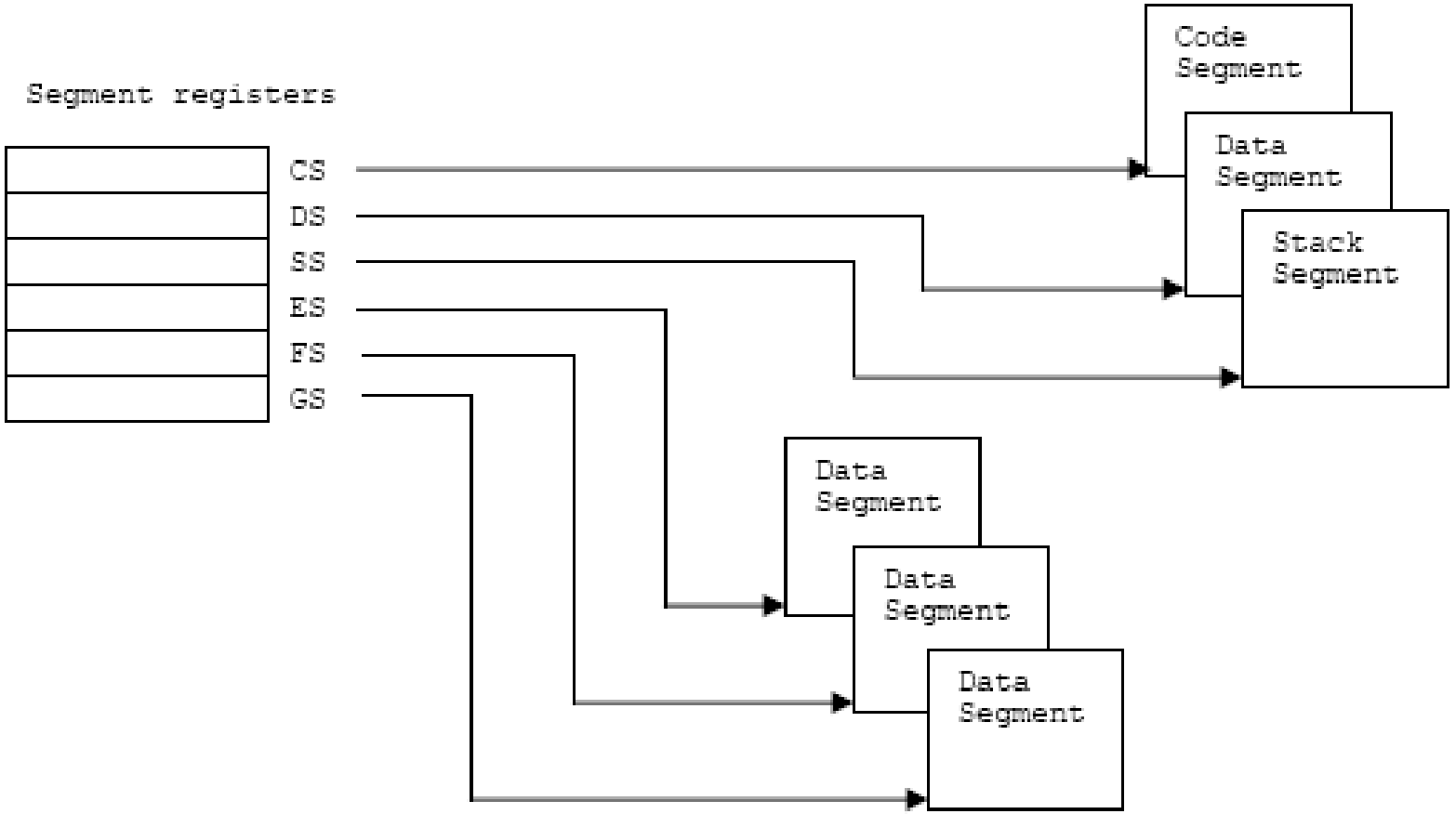







General Purpose Registers (GPRs)
Also: 6 segment registers, control, status, debug, more
Address Space
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
0
Stack
2^64-1
0
Legacy x86 registers
New x64 registers
Instruction Pointer/Flags
RIP
RFLAGS
EFLAGS
63
0
7 0
15
31
63
127
Byte
Word
Doubleword
Quadword
Double Quadword
Increasing Addresses
80-bit floating point and 64-bit MMX registers (overlaid)
MMX Part
FPR0/MMX0
FPR1/MMX1
FPR2/MMX2
FPR3/MMX3
FPR4/MMX4
FPR5/MMX5
FPR6/MMX6
FPR7/MMX7
79
63
0
128-bit XMM Registers
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15
127
0

Segment Registers

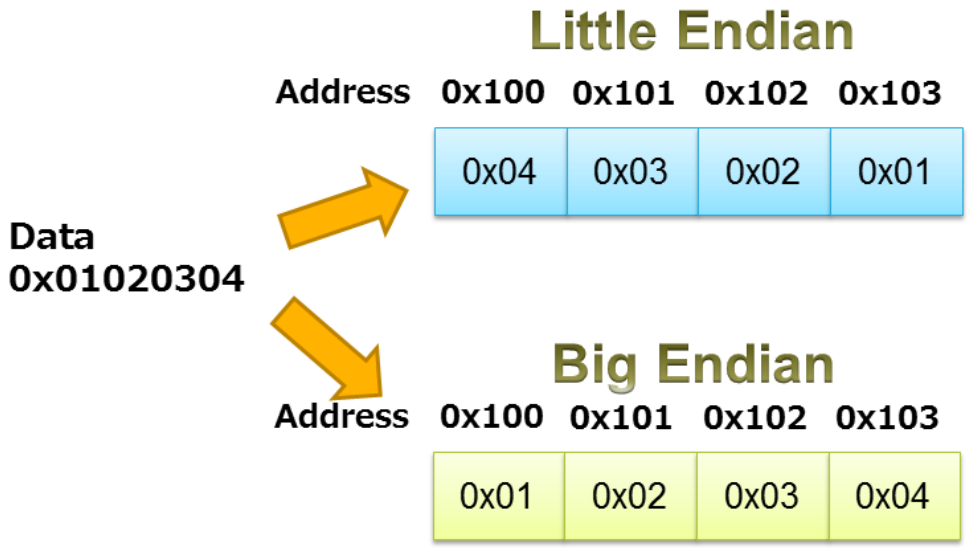


Stack, heap

- HEAP: malloc, calloc, new …
- STACK: automatic variables
  - push
  - pop

Addressing

Data Type	Size
Byte	8 bits
Word	16 bits
Doubleword	32 bits
Quadword	64 bits



Assembly line

[label] mnemonic [operands] [; comment]

```
; This is a comment
    jmp label1 ; This is also a comment
    add eax, ebx
label1:
    sub edx, 32
```

Addressing

Register Addressing Mode

The operand is in a register.

```
        mov EAX, EBX ; move EBX to EAX
```

Immediate Addressing Mode

The operand is part of the instruction.

```
        mov EAX, 132 ; move 132 to EAX
```

Direct addressing mode

The operand is in memory, and the address is specified as an offset.

```
a_letter DB 'c'          ; Allocate one byte of memory, initialize it to 'c'.
        mov AL, a_letter ; Move data at memory location "a_letter" into AL.
                         ; I.e. move 'c' to AL.
```

Addressing

Register Indirect Addressing

The operand is found at the memory location specified by the register.
The register is enclosed in square bracket.

```
        mov EAX, ESP ; Move stack pointer to EAX
        mov EBX, [ESP] ; Move value at top-of-stack to EBX
```

Indirect Addressing Mode

The offset of the data is in one of the eight general-purpose registers.

```
.DATA
array DD 20 DUP (0) ; Array of 20 integers initialized to zero
.CODE
        mov ECX, OFFSET array ; Move starting address of 'array' to ECX
```

Addressing

Based Addressing

One of the eight general-purpose registers acts like a base register in computing the effective address of an operand. The address is computed by adding a signed (8-bit or 32-bit) number to the base address.

```
mov ECX, 20[EBP] ; ECX = memory[EBP + 20]
```

Indexed Addressing

The effective address is computed by:

signed displacement + (Index * scale factor)

```
add AX, [DI + 20]     ; AX = AX + memory[DI + 20]
mov AX, table[ESI*4]; AX = memory[ OFFSET table + ESI * 4 ]
add AX, table[SI]     ; AX = AX + memory[ OFFSET table + SI * 1]
```

scale factor: 1, 2, 4 or 8

index register: EAX, EBX, ECX, EDX, ESI, EDI, EBP

Addressing

Based-Indexed Addressing

In this addressing mode, the effective address is computed as:

Base + (Index * Scale factor) + signed displacement.

```
mov EAX, [EBX+ESI] ; AX = memory[EBX + (ESI * 1) + 0]
mov EAX, [EBX+EPI*4+2] ; AX = memory[EBX + (EPP * 4) + 2]
```

base register: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP

index register: EAX, EBX, ECX, EDX, ESI, EDI, EBP

scale factor: 1, 2, 4 or 8

signed displacement: 8, 16 or 32-bit value

PTR Directive

- Resolves ambiguous operator size

```
mov [ESI], al     ; Store a byte-size value in memory
                  ; location pointed by ESI

mov [ESI], 5      ; Error: operand must have the size specified

mov [ESI], BYTE PTR 5  ; Store 8-bit value


mov      ax, WORD PTR [num] ; Load a word-size value from a DWORD
```

Arithmetic Instructions

- ADD, SUB, MUL, IMUL, DIV, IDIV…
  - ADD/SUB: add dest, src ; left one is source and destination

```
add eax, ebx      ; eax <- eax+ebx
add [esp], eax    ; eax on top of the stack
add eax, [esp]    ; top of the stack in eax
add eax, 4        ; immediate 4 in eax
```

  - DIV/IDIV: div divisor ; dividend always in eax: result in eax and rest in edx

```
mov eax, 65       ; immediate 65 in eax
mov ecx, 4        ; 4 in ecx
div ecx           ; eax<- eax/ecx  edx<- eax%ecx
```

  - MUL/IMUL:

```
mul value         ; eax<-eax*value
mul dest, val1, val2
mul dest, val
```

Bitwise opeartions

- AND, OR, XOR, NOT
  - AND/OR/XOR dst, src
  - NOT eax

Branching

- JMP, JE, JLE, JNZ, JZ, JBE, JGE… : jmp address ; EIP <- OFFSET address
  - Jump could depend on current status of condition codes => result of previous operations (E if ZF is high; LE if ZF==1 ||SF==1; NZ if ZF==0; Z if ZF==1; …)
    - Condition Codes: ZF – zero flag, SF – signed flag, OF – overflow flag, CF – carry flag
  - ADD/SUB: set Z, S, O, C
  - AND: O = C = 0, set Z, S
  - CMP: “cmp dest, src" only purpose is to set flags for following JMP instruction
  - JLE vs JBE : Less or Egual vs Below or Egual : signed vs unsigned

Data Moving

- MOV, MOVS, MOVSB, MOVSW, MOVZX, MOVSX, LEA… :
  - MOV dst, src
    - Move source to destination: reg to reg / mem to reg / reg to mem (NO mem to mem)
  - MOVS.. Move string from memory location (ESI) to memory location (EDI)

Instruction	Description
MOVSB	Move byte at address DS:(E)SI to address ES:(E)DI
MOVSW	Move word at address DS:(E)SI to address ES:(E)DI
MOVSD	Move doubleword at address DS:(E)SI to address ES:(E)DI

  - if DF (Decrement Flag) is 0/1 EDI and ESI are ***properly*** increased/decreased
  - Explicit operand can be
  - Can be prefixed by REP to move ECX bytes/words/double words

```
REP MOVSB ;
```
 moves ECX bytes from ESI to EDI

Loop-ing

- LOOP dst : jump to destination until ECX is zero

```
    mov ecx, 5 ; ecx stands for extended counter
_proc:
    dec ecx ; decrements ecx
    loop _proc ; loops back to _procs
```

- REP/REPE/REPZ/REPNE/REPNZ : like loop but for string management

```
    mov esi, str1
    mov edi, str2
    mov ecx, 10h
    rep cmps ; stops after 16 bytes or strings differ
```

Scan string (SCAS) example

- find the length of a NUL-terminated string:

```
MOV DI, DX      ;Starting address in DX (assume ES = DS)
MOV AL, 0       ;Byte to search for (NUL)
MOV CX, -1      ;Start count at FFFFh
CLD             ;DI=0 =>Increment DI after each character
REPNE SCASB     ;Scan string for AL,inc CX for each char
MOV AX, -2      ;CX=-2 for len 0, -3 for len 1, ...
SUB AX, CX      ;Length in AX
```

Data Allocation

[variable-name] define-directive initial-value [,initial-value],...

- Variable-name: identify the storage space allocated.
- Define-directive:

Data directive

Directive	Description of Initializers
BYTE, DB (byte)	Allocates unsigned numbers from 0 to 255.
SBYTE (signed byte)	Allocates signed numbers from –128 to +127.
WORD, DW (word = 2 bytes)	Allocates unsigned numbers from 0 to 65,535 (64K).
SWORD (signed word)	Allocates signed numbers from –32,768 to +32,767.
DWORD, DD (doubleword = 4 bytes),	Allocates unsigned numbers from 0 to 4,294,967,295 (4 megabytes).
SDWORD (signed doubleword)	Allocates signed numbers from –2,147,483,648 to +2,147,483,647.
FWORD, DF (farword = 6 bytes)	Allocates 6-byte (48-bit) integers. These values are normally used only as pointer variables on the 80386/486 processors.
QWORD, DQ (quadword = 8 bytes)	Allocates 8-byte integers used with 8087-family coprocessor instructions.
TBYTE, DT (10 bytes),	Allocates 10-byte (80-bit) integers if the initializer has a radix specifying the base of the number.
REAL4	Short (32-bit) real numbers
REAL8	Long (64-bit) real numbers
REAL10	10-byte (80-bit) real numbers and BCD numbers

- Examples
- letter_c DB 'c' ; Allocate a single byte of memory, and initialize it to the letter 'c'.
- an_integer DD 12425 ; Allocate memory for an integer (4-bytes), and initialize it to 12425.
- a_float REAL4 2.32 ; Allocate memory for a float, and initialize it to 2.32
- message DB 'Hello',13,0 ; Allocate memory for a null terminated string "Hello\n"
- marks DW 0, 0, 0, 0 ; Both allocates memory for an array of 4 * 2 bytes, and ; initialize all elements to zero.
- marks DW 4 DUP (0) ; DUP allows multiple initializations to the same value
- name DB 30 DUP(?) ; Allocate memory for 30 bytes, uninitialized.
- matrix QW 12*10 ; Allocate memory for a 12*10 quad-bytes matrix

Stack management – POP, PUSH

POP, syntax: pop dest

PUSH, syntax: push var/reg

Move to/from the stack and updates ESP

Functions – CALL, RET

(http://pages.cs.wisc.edu/~remzi/Classes/354/Fall2012/Handouts/Handout-CallReturn.pdf)

CALL, syntax: CALL _function

- similar to a JUMP but stores EIP on the stack (for the return)
  1. EIP -> stack ; This is done by the CALL instruction
  2. EBP -> stack ;
  3. EBP <- ESP ; actually a “calling convention” abstraction
  4. ESP is decremented to, among several things, contain the local variables of _function
  5. EIP=OFFSET _function

RET, syntax: RET/RET num

1. EBP <- ESP ; restore the saved EBP
2. EIP <- ESP ; restore RETURN address

CALLing conventions

- On 32-bit x86 on Linux, the calling convention is named cdecl
- caller (parent) pushes the arguments from right to left onto the stack, calls the target function (callee/child), receives the return value in eax, and pops the arguments

```
;nasm -f elf64 -o hw.o helloWorld.asm
;ld -o hw hw.o
BITS 64
SECTION .data
Hello:          db "Hello world!"
len_Hello:      equ $-Hello
SECTION .text
global _start
_start:
        mov rax,1                       ; write syscall (x86_64)
        mov rdi,1                       ; fd = stdout

        mov rsi,Hello                   ; *buf = Hello
        mov rdx,len_Hello   ; count = len_Hello
        syscall

        mov rax,60                              ; exit syscall (x86_64)
        mov rdi,0                       ; status = 0 (exit normally)
        syscall
```

GCC disassembly (Intel Style & AT&T style)

1. **Register Naming**: prefixed with % => registers are %eax, %cl etc (not eax, cl, …)
2. **Ordering of operands**: source(s) first, and destination last.
   Intel syntax "mov eax, edx"
   AT&T assembly "mov %edx, %eax"
3. **Operand Size**: In AT&T syntax, the size of memory= suffix l
   b for (8-bit) byte, w for (16-bit) word, and l for (32-bit) long
   "movl %edx, %eax".
4. **Immediate Operand**: marked with a $ prefix
   "addl $5, %eax"
5. **Memory Operands**: Missing operand => memory-address;
   "movl $bar, %ebx" puts the address of variable bar into register %ebx,
   "movl bar, %ebx" puts the contents of variable bar into register %ebx.
6. **Indexing**: Indexing or indirection is done by enclosing the index register or indirection memory cell address in parentheses.
   "movl 8(%ebp), %eax" (moves the contents at offset 8 from the cell pointed to by %ebp into register %eax).

Inline ASM code

```
#include <stdio.h>

int main() {
   /* Add 10 and 20 and store result into register
            %eax */
      __asm__ ( "movl $10, %eax;"
                  "movl $20, %ebx;"
                  "addl %ebx, %eax;"
      );

   /* Subtract 20 from 10 and store result into
            register %eax */
   __asm__ ( "movl $10, %eax;"
                                 "movl $20, %ebx;"
                                 "subl %ebx, %eax;"
   );

   /* Multiply 10 and 20 and store result into
            register %eax */
      __asm__ ( "movl $10, %eax;"
                                 "movl $20, %ebx;"
                                 "imull %ebx, %eax;"
      );

   return 0 ;
}
```

Operands from/to variables

```
int no = 100, val ;
    asm ("movl %1, %%ebx;"
         "movl %%ebx, %0;"
         : "=r" ( val )    /* output «=r» param %0 */
         : "r" ( no )      /* input param %1 */
         : "%ebx"          /* clobbered register (GCC do
                              not use*/
     );
```

Complete example

```
#include <stdio.h>

int main() {

    int arg1, arg2, add, sub, mul, quo, rem ;

    printf( "Enter two integer numbers : " );
    scanf( "%d%d", &arg1, &arg2 );

    /* Perform Addition, Subtraction, Multiplication & Division */
    __asm__ ( "addl %%ebx, %%eax;" : "=a" (add) : "a" (arg1) , "b" (arg2) );
    __asm__ ( "subl %%ebx, %%eax;" : "=a" (sub) : "a" (arg1) , "b" (arg2) );
    __asm__ ( "imull %%ebx, %%eax;" : "=a" (mul) : "a" (arg1) , "b" (arg2) );

    __asm__ ( "movl $0x0, %%edx;"
              "movl %2, %%eax;"
              "movl %3, %%ebx;"
               "idivl %%ebx;" : "=a" (quo), "=d" (rem) : "g" (arg1), "g" (arg2) );

    printf( "%d + %d = %d\n", arg1, arg2, add );
    printf( "%d - %d = %d\n", arg1, arg2, sub );
    printf( "%d * %d = %d\n", arg1, arg2, mul );
    printf( "%d / %d = %d\n", arg1, arg2, quo );
    printf( "%d %% %d = %d\n", arg1, arg2, rem );

    return 0 ;
}
```

Invoking an assembler function: main.c

```
// Compile as gcc -o main.c asm_mod_array.sbrk
#include <stdio.h>
/* prototype for asm function */
int asm_mod_array(int *ptr,int size);

int main() {
        int fren[5]={ 1, 2, 3, 4, 5 };
        int i;
        /* call the asm function */
        int n=asm_mod_array(fren, 5);
        printf ("n = %d", n);
        for (i =0 ;i < 5;i++){
                printf("\n %d, ", fren[i]);
        }
        return 0;
}
```

Invoking an assembler function: asm_mod_array.asm

```
# Parameters are passed in rdi and rsi
# return value is received in rax
#
.section .text
.globl asm_mod_array
.type asm_mod_array, @function
asm_mod_array:
        push %rbp
        mov %rsp, %rbp
        mov %rdi,%rax        # get pointer to start of array passed from C
        mov %rsi,%rcx        # get size of array
        xorl %edi, %edi      # zero out our array index
start_loop:          # start loop
        cmpl %edi, %ecx      #check to see if we've hit the end
        je loop_exit
        movl (%rax,%rdi,4), %edx   # store the element in %edx for calculations
        lea 5(,%edx,2), %edx   # multiply array element by 2 and add 5
        movl %edx, (%rax,%rdi,4)   # overwrite old element with new value
        incl %edi       # increment the index, moving through the array.
        jmp start_loop          # jump to loop beginning


loop_exit:              # function epilogue
        movl $6, %eax           # only to show a return value
        mov %rbp, %rsp
        pop %rbp
        ret                 # pop the return address and jmp to it
```

References

- http://www.cs.umd.edu/~meesh/cmsc311/links/handouts/ia32.pdf
- https://sensepost.com/blogstatic/2014/01/SensePost_crash_course_in_x86_assembly-.pdf
- http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C
- https://www.youtube.com/watch?v=75gBFiFtAb8
- https://software.intel.com/en-us/articles/introduction-to-x64-assembly
- https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc/Using-Assembly-Language-with-C.html#Using-Assembly-Language-with-C