Computer Architecture and Assembly Language

Practical Session 5

Addressing Mode - "memory address calculation mode"

An **addressing mode** specifies **how to calculate the effective memory address of an operand**.

x86 addressing mode rule: up to two of the 32-bit registers and a 32-bit signed constant can be added together to compute a memory address. One of the registers can be optionally pre-multiplied by 2, 4, or 8.

Example of **right** usage

```
mov eax, [ebx]          ; Move the 4 bytes in memory at the address contained in EBX into EAX
mov [var], ebx          ; Move the contents of EBX into the 4 bytes at memory address "var"
                        ; (Note, "var" is a 32-bit constant).
mov eax, [esi-4]        ; Move 4 bytes at memory address ESI+(-4) into EAX
mov [esi+eax], cl       ; Move the contents of CL into the byte at address ESI+EAX
mov edx, [esi+4*ebx]    ; Move the 4 bytes of data at address ESI+4*EBX into EDX
```

- **mov dword [myArray + ebx*4 + eax], ecx**

Examples of **wrong** usage

```
mov eax, [ebx-ecx]      ; Can only add register values
mov [eax+esi+edi], ebx  ; At most 2 registers in address computation
```

Addressing Mode - "operand accessing mode"

- **Register Addressing**

operate on a variable or intermediate variable

```
inc eax
```

- **Immediate Addressing**

operate on a CONSTANT

```
add ax, 0x4501
```

- **Absolute (Direct) Addressing**

specify the address as a number or label

```
inc word [myString]
inc word [0x1000]
```

- **Register Indirect Addressing**

```
inc byte [ebx]
```

- **Displacement Addressing**

Effective Address=[register]+displacement

```
inc byte [ebx+0x10]
```

- **Relative Addressing**

Effective Address =[PC]+ displacement

```
        jnz next        ; ≡ jnz $+4
        inc eax
next: neg eax
```

- **Indexed Addressing (with Displacement)**

useful for array indices

```
inc dword [ebx*4]
inc dword [ebx*4+eax]
inc dword [myArray + ebx*4]
inc dword [myArray + ebx*4+eax]
```

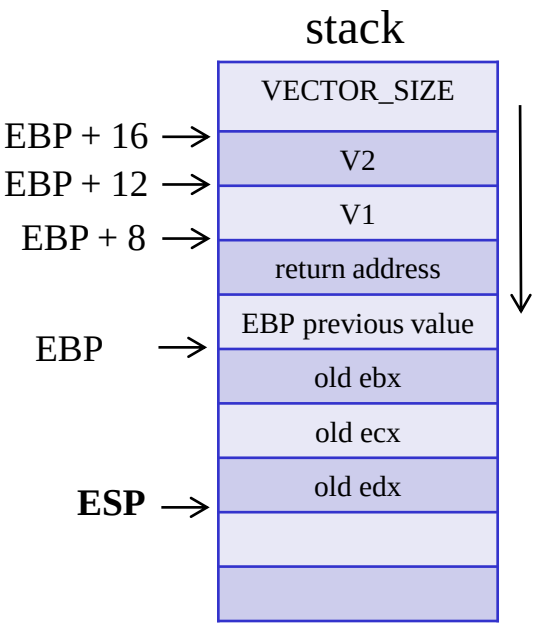
Addressing Modes - Example

The dot product of two vectors **a** = [a_1, a_2, ..., a_n] and **b** = [b_1, b_2, ..., b_n] is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

```
#include <stdio.h>
#define VECTOR_SIZE 5


extern long long* DotProduct
                  (int V1[VECTOR_SIZE],
                   int V2[VECTOR_SIZE],
                   int size);

int main ()
{
  int V1[VECTOR_SIZE] = {1,0,1,0,2};
  int V2[VECTOR_SIZE] = {1,0,1,0,-2};

  long long* result = DotProduct(V1,V2,VECTOR_SIZE);
  printf ("Dot product result: %#llx \n ", result);

  return 0;
  }
```

```
int V1[VECTOR_SIZE] = {1,0,1,0,2};
int V2[VECTOR_SIZE] = {1,0,1,0,-2};

long long* result = DotProduct(V1,V2,VECTOR_SIZE);
```



next element of vector V1

next element of vector V2

```
section     .data
result:   dd  0,0
section     .text
          global  DotProduct
DotProduct:
          push    ebp
          mov     ebp, esp
          push    ebx
          push    ecx
          push    edx
          mov     ecx, 0
DotProduct_start:
          mov     edx, 0
          cmp     ecx, dword [ebp+16]
          je       DotProduct_end
          mov     ebx, dword [ebp+8]
          mov     eax, dword [ebx + (4*ecx)]
          mov     ebx, dword [ebp+12]
          imul    dword [ebx + (4*ecx)]
          add     dword [result], eax
          adc     dword [result+4], edx
          inc     ecx
          jmp      DotProduct_start
DotProduct_end:
          mov     eax, result  ; return value
          pop     edx
          pop     ecx
          pop     ebx
          mov     esp, ebp
          pop     ebp
          ret
```

Since the function is called from C, the function is allowed to mess up the values of EAX, ECX and EDX registers. Since EBX register is also used by function, its value should be preserved and restored.

But in fact it is not 100% sure that all C code obeys this (may be compiler implementation dependent), so it is a good idea to save/restore all registers used by function

Linux System Calls

- System calls are low level functions the operating system makes available to applications via a defined API (Application Programming Interface)
- System calls represent the *interface* the kernel presents to user applications.
- In Linux all low-level I/O is done by reading and writing file handles, regardless of what particular peripheral **device** is being accessed - a tape, a socket, even your terminal, they **are all *files***. Files are referenced by an integer file descriptor.
- Low level I/O is performed by making *system calls*

Linux System Call format

A system call is explicit request to the kernel, made via a software interrupt

- Put the system call number in `EAX` register
- Set up the arguments to the system call.
  - The first argument goes in EBX, the second in ECX, then EDX, ESI, EDI, and finally EBP. If more then 6 arguments needed (not likely), the EBX register must contain the memory location where the list of arguments is stored.
- Call the relevant interrupt (for Linux it is 0x80)
- The result is usually returned in `EAX`

sys_read – read from a file

- system call number (in EAX): **3**

- arguments:
  - EBX: file descriptor (to read from it)
  - ECX: pointer to input buffer (to keep a read data into it)
  - EDX: maximal number of bytes to receive (maximal buffer size)

- return value (in EAX):
  - number of bytes received
  - On errors: -1or 0 (no bits read)

```
section     .bss
   buffer:   resb    1

section     .text
  global _start
_start:

   mov eax, 3                ; system call number (sys_read)
   mov ebx, 0                ; file descriptor (stdin)
   mov ecx, buffer           ; buffer to keep the read data
   mov edx, 1                ; bytes to read
   int 0x80                   ;call kernel

   mov eax,1                 ;system call number (sys_exit)
   mov ebx, 0                ;exit status
   int 0x80                  ;call kernel
```

sys_write – write into a file

- system call number (in EAX): **4**

- arguments:
  - EBX: file descriptor (to write to it)
  - ECX: pointer to the first byte to write (beginning of the string)
  - EDX: number of bytes (characters) to write

- return value (in EAX):
  - number of bytes written
  - On errors: -1

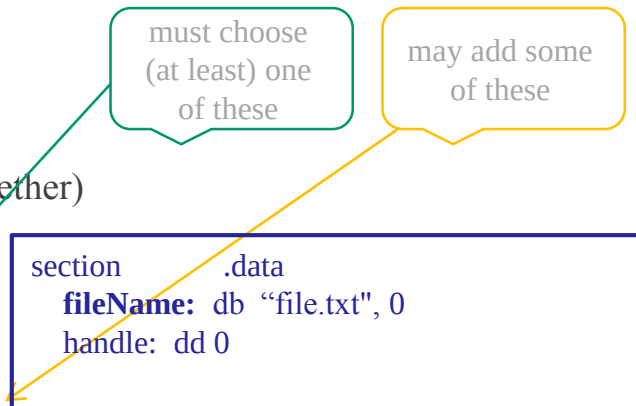
```
section     .data
   msg: db 'Message',0xa        ;our string, 0xA is newline
   len:  equ $ - msg            ;length of our string

section     .text
   global _start
_start:

    mov ebx,1                   ;file descriptor (stdout)
    mov ecx, msg                ;message to write
    mov edx, len                ;message length
    mov eax,4                   ;system call number (sys_write)
    int 0x80                    ;call kernel

    mov eax,1                   ;system call number (sys_exit)
    mov ebx, 0                  ;exit status
    int 0x80                    ;call kernel
```

sys_open - open a file

- system call number (in EAX): **5**

- arguments:
  - EBX: pathname of the file to open/create
  - ECX: set file access bits (can be bitwise OR'ed together)
    - O_RDONLY (0x0000) open for reading only
    - O_WRONLY (0x0001) open for writing only
    - O_RDRW (0x0002) open for both reading and writing
    - O_APPEND (0x0008) open for appending to the end of file
    - O_TRUNC (0x0200) truncate to 0 length if file exists
    - **O_CREAT (0x0100) create the file if it doesn't exist**
  - EDX: set file permissions (in a case **O_CREAT** is set; can be bitwise OR'ed together)
    - S_I**RWX**U 0000700 ; **RWX** mask for owner
    - S_I**R**USR 0000400 ; **R**(read) for owner USR(user)
    - S_I**W**USR 0000200 ; **W**(write) for owner
    - S_I**X**USR 0000100 ; **X**(execute) for owner

- return value (in EAX):
  - file descriptor
  - On errors: -1




```
section         .data
   fileName:  db  "file.txt", 0
   handle:  dd 0

section         .text
  global _start
_start:

file_open:
  mov eax, 5                ; system call number (sys_open)
  mov ebx, fileName         ; set file name
  mov ecx, 0x0101           ; set file access bits
                             (O_WRONLY | O_CREAT)
  mov edx, S_IRWXU          ; set file permissions
  int 0x80                  ;call kernel
  mov [handle], eax         ; move file handle to memory

   mov eax, 1                   ;system call number (sys_exit)
   mov ebx, 0                   ;exit status
   int 0x80                     ;call kernel
```

sys_lseek – change a file pointer

- system call number (in EAX): **19**
- arguments:
  - EBX: file descriptor
  - ECX: offset (number of bytes to move)
  - EDX: where to move from
    - SEEK_SET (0) ; beginning of the file
    - SEEK_CUR (1) ; current position of the file pointer
    - SEEK_END (2) ; end of file
- return value (in EAX):
  - Current position of the file pointer
  - On errors: SEEK_SET

```
section         .data
   fileName:  db  “file.txt", 0
   handle:  dd 0

section         .text
  global _start
_start:

file_open:
   mov eax, 5                ; system call number (sys_open)
   mov ecx, 0                ; set file access bits (O_RDONLY )
   mov ebx, fileName         ; set file name
   int 0x80                  ;call kernel
   mov [handle], eax         ; move file handle to memory

   mov ebx, [handle]         ; file descriptor
   mov ecx,15                ; number of byte to move
   mov edx,0                 ; move from beginning of the file
   mov eax,19                ; system call number (lseek)
   int 0x80                  ;call kernel

    mov eax, 1                   ;system call number (sys_exit)
    mov ebx, 0                   ;exit status
    int 0x80                     ;call kernel
```

sys_close – close a file

- system call number (in EAX): **6**
- arguments:
  - EBX: file descriptor
- return value (in EAX):
  - nothing meaningful
  - On errors: -1

```
section          .data
   fileName:  db  “file.txt", 0
   handle:  dd 0

section          .text
  global _start
_start:

file_open:
  mov eax, 5                ; system call number (sys_open)
  mov ecx, 0                ; set file access bits (O_RDONLY)
  mov ebx, fileName         ; set file name
  int 0x80                  ;call kernel
  mov [handle], eax         ; move file handle to memory

  mov ebx, [handle]
  mov eax, 6
  int 0x80

   mov eax, 1                   ;system call number (sys_exit)
   mov ebx, 0                   ;exit status
   int 0x80                     ;call kernel
```

Linux System Calls - Example

```
section         .data
   fileName:  db  “file.txt", 0
   handle:  dd 0

section .bss
   buffer:   resb    1

section         .text
  global _start

_exit:
   mov ebx, [handle]
   mov eax, 6                ; system call (sys_close)
   int 0x80                  ; call kernel

   mov eax,    1             ; system call (sys_exit)
   mov ebx,    0             ; exit status
   int 0x80                  ; call kernel
```

```
_start:

   mov eax, 5                ; system call (sys_open)
   mov ebx, fileName         ; set file name
   mov ecx, O_RDONLY         ; set file access bits (O_RDONLY)
   int 0x80                  ; call kernel
   mov [handle], eax         ; move file handle to memory

_read:

   mov eax, 3                ; system call (sys_read)
   mov ebx, [handle]         ; file handle
   mov ecx, buffer           ; buffer
   mov edx, 1                ; read byte count
   int 0x80                  ; call kernel
   cmp eax, 0
   je _exit

   mov eax, 4                ; system call (sys_write)
   mov ebx, 1                ; stdout
   mov ecx, buffer           ; buffer
   mov edx, 1                ; write byte count
   int 0x80                  ; call kernel

   jmp _read
```

Assignment #2

- Writing a simple calculator for unlimited-precision integers.
- Operators:
  - **Addition unsigned (+)** (BCD decimal)
  - **Pop-and-print (p)**
  - **Duplicate (d)**
  - **Bitwise AND (&)** (Hexadecimal)
  - **Quit (q)**

- Operands in the input and output will be in **decimal.**

- Your program should allow **“-d”** command line argument, which means a **debug option**. When "-d" option is set, you should print out to stderr various debugging messages (as a minimum, print out every number read from the user, and every result pushed onto the operand stack).

Assignment #2

- The calculator uses Reverse Polish Notation (RPN)
  - i.e. every operator follows all of its operands

Example:

1 2 +	→	1+2
10 6 &	→	bitwise AND of 10, 6

- Operands in the calculator are implicit – taken from a stack that is implemented by you

Assignment #2

>>calc: 9

>>calc: 1

>>calc: d

>>calc: p

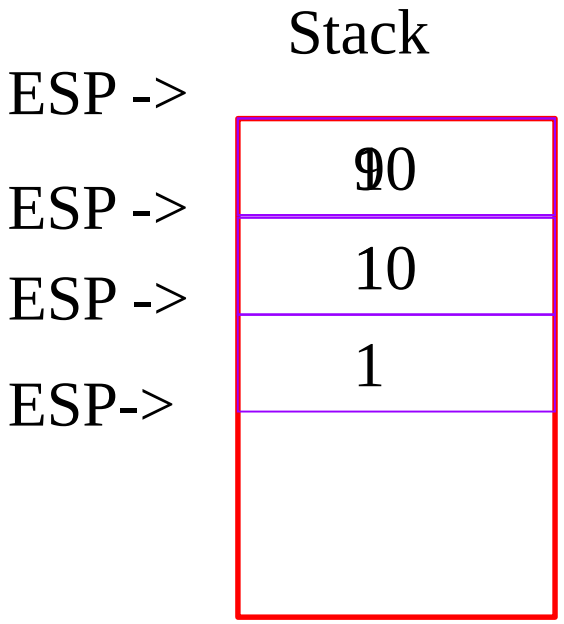
>>1

>>calc: +

>>calc: d

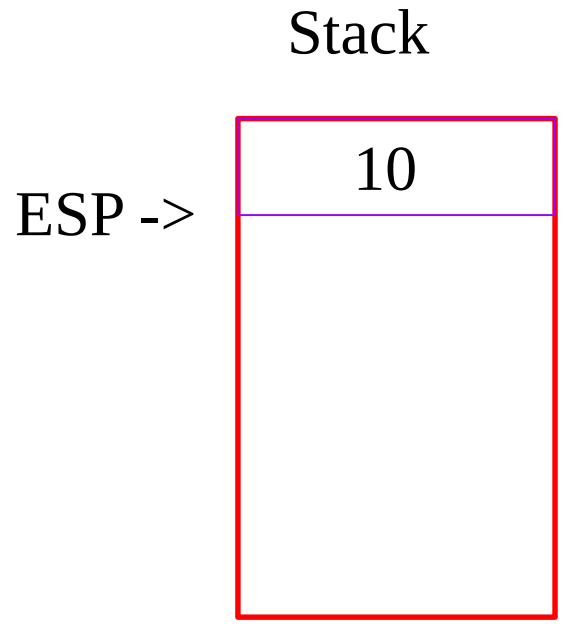
>>calc: p

>>10



Assignment #2

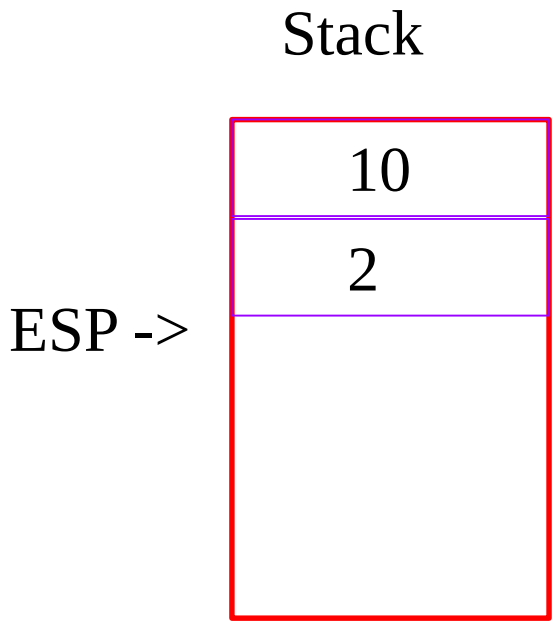
>>calc: 2



Assignment #2

>>calc: 2

>>calc: &

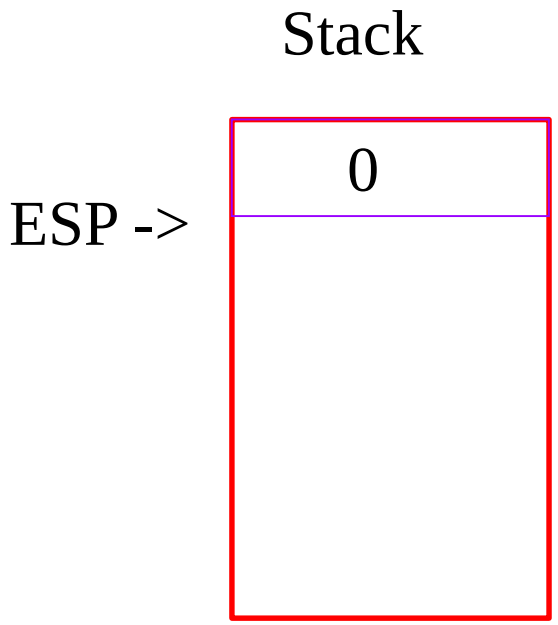


Assignment #2

>>calc: 2

>>calc: &

>>calc: p



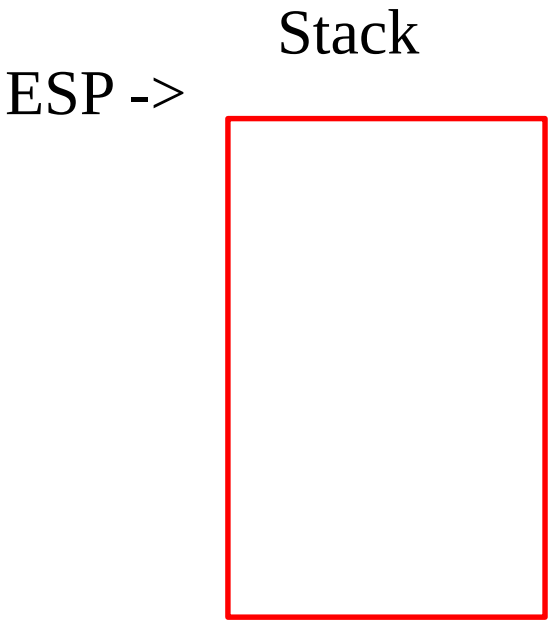
Assignment #2

>>calc: 2

>>calc: &

>>calc: p

>>0



Assignment #2

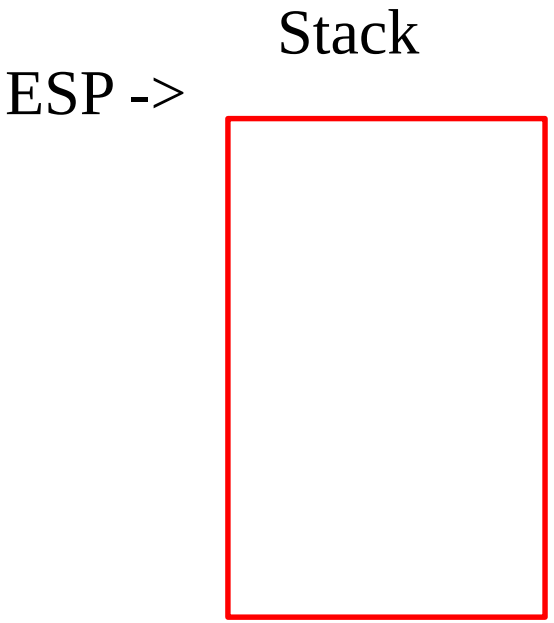
>>calc: 2

>>calc: &

>>calc: p

>>0

>>calc: +



Assignment #2

>>calc: 2

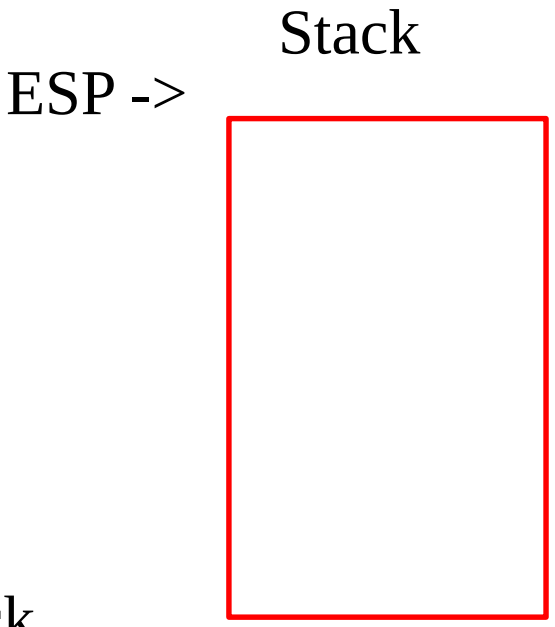
>>calc: &

>>calc: p

>>0

>>calc: +

Error: Not Enough Arguments in Stack



Assignment #2

- Your program should be able to handle an **unbounded operand size**

  This **may** be implemented as a linked list:
  - each element represents 2 decimal digits in the number, in **BCD** (binary-coded decimal) representation
  - an element block is composed of a 4 byte pointer to the next element, and a byte data

BCD (binary-coded decimal) is a class of binary encodings of decimal numbers where each decimal digit is represented by 4 bits. For example, **56** decimal is represented by

0	1	0	1	0	1	1	0

Indeed, we get the value of 0x56, so in BCD a decimal number is represented by hexadecimal number with the same digits.

Example:

123456 decimal number could be represented by the following linked list:

0x56 → 0x34 → 0x12 → Null

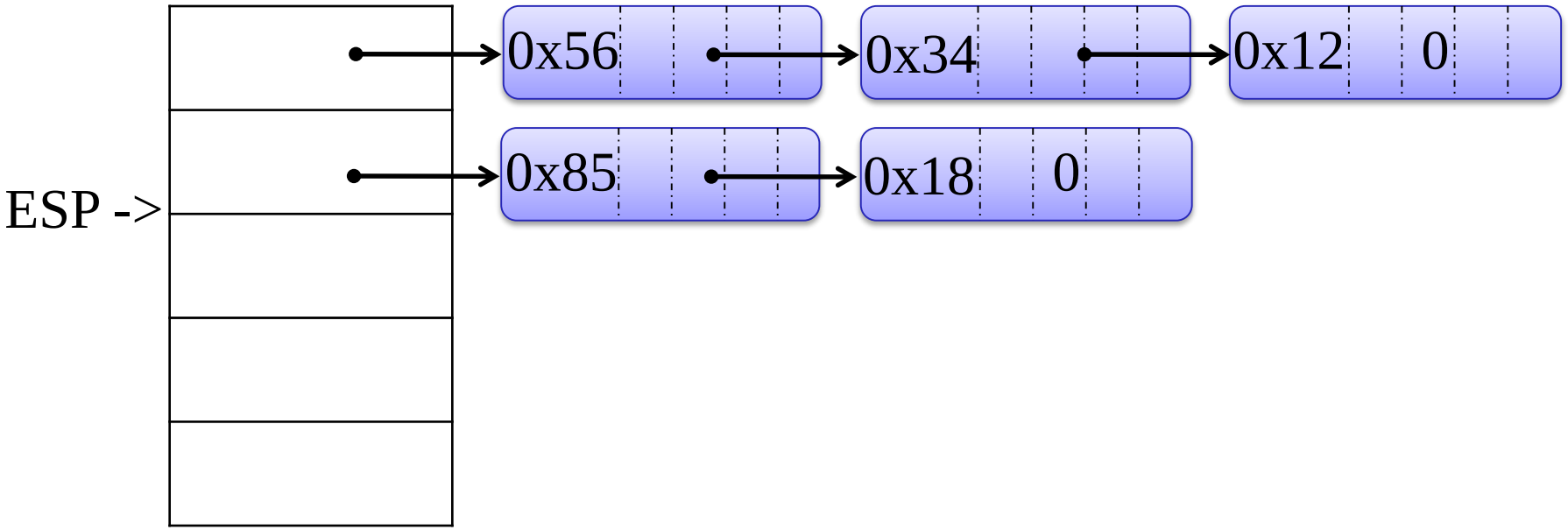
Using this representation, each stack element is simply a list head pointer.

Assignment #2

- The stack is implemented by you
- The stack is of size 5, i.e. you can insert only 5 inputs in it
- We insert into stack only pointers to elements

>>calc:123456
>>calc:1885

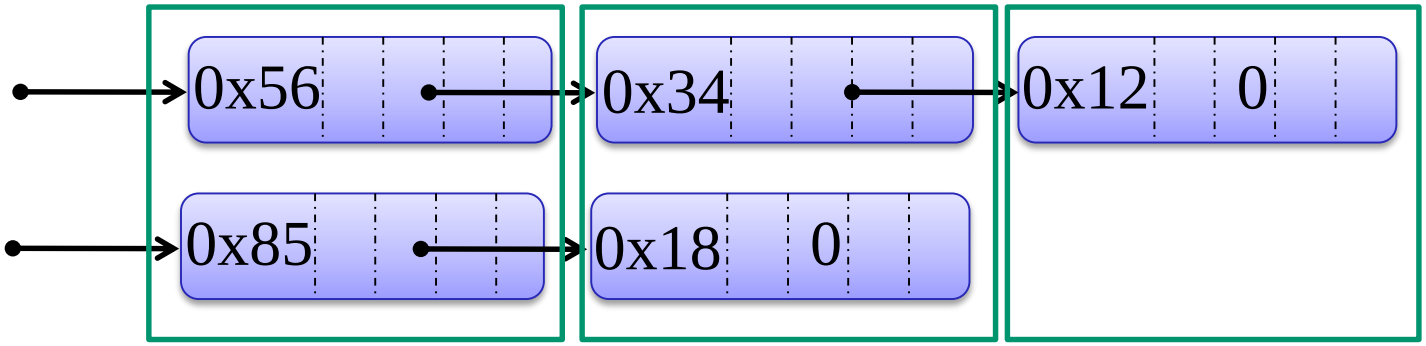
Our Stack



Assignment #2

- Add two elements byte by byte using ‘add’ instruction
- After addition, use ‘daa’ instruction to decimal adjust the result after addition

Example: 123456 + 1885= 125341



AL ← 0x56 + 0x85, then adjust to decimal

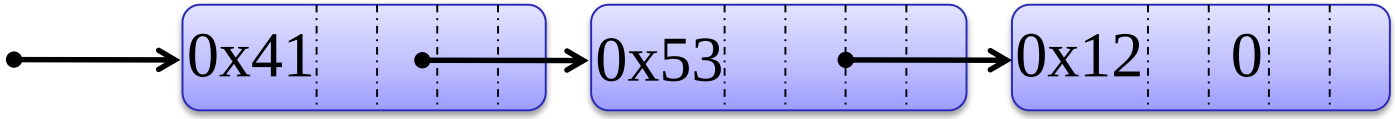
AL ← 0x34 + 0x18 + carry of the addition from the previous byte, then adjust to decimal

AL ← 0x12 + carry of the addition from the previous byte , then adjust to decimal

AL ← 0x56 + 0x85 = 0xDB
daa ; AL ← 0x41, CF = 1 since the right result is 0x**1**41

AL ← 0x34 + 0x18 + CF = 0x4D
daa ; AL ← 0x53, CF = 0

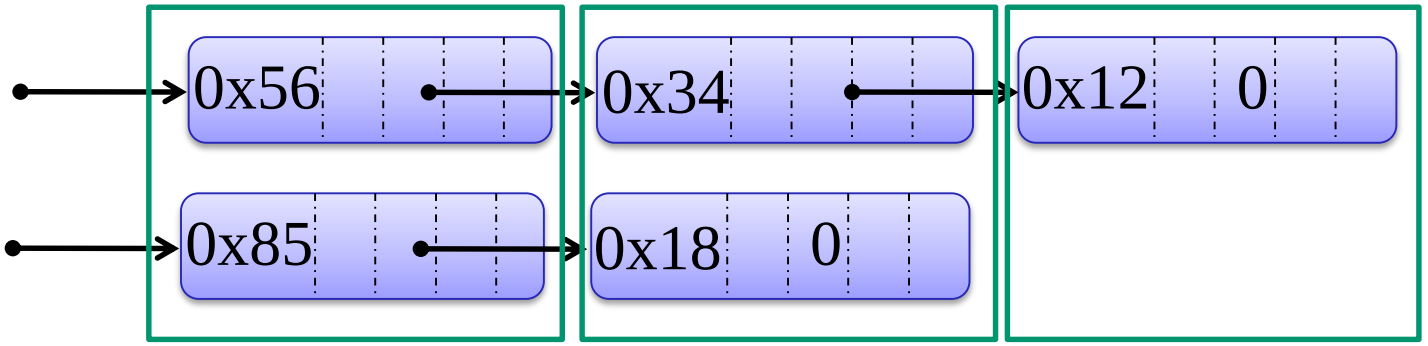
AL ← 0x12 + CF = 0x12
daa ; AL ← 0x12



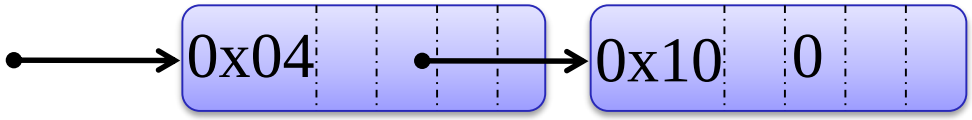
Assignment #2

- Bitwise AND two elements byte by byte using ‘AND’ instruction

Example: 123456 & 1885= 1004



and 0x56, 0x85 ; = 0x04 and 0x34, 0x18 ; = 0x10 and 0x12, 0x00 ; = 0x00



Assignment #2

(Packed) **BCD representation** - saves space by packing two digits into a byte (one decimal digit (0..9) per nibble).

Example:
1234 is stored as 12 34H

Two instructions to process packed BCD numbers

daa (decimal adjust after addition)

- decimal adjust after addition

das (decimal adjust after subtraction)

- decimal adjust after subtraction

Algorithm of daa:

if low nibble of AL > 9 or AF = 1 then:
    AL = AL + 6
    AF = 1
if AL > 9Fh or CF = 1 then:
    AL = AL + 60h
    CF = 1

Algorithm of das:

if low nibble of AL > 9 or AF = 1 then:
    AL = AL - 6
    AF = 1
if AL > 9Fh or CF = 1 then:
    AL = AL - 60h
    CF = 1

Assignment #2

Packed BCD addition examples:

```
mov AL, 05h
add  AL, 09h     ; AL = 0Fh (15)
daa              ; AL = 15h
```

```
mov AL,71h
add  AL,43h    ; AL := B4h
daa            ; AL := 14h and CF := 1
```

The result including the carry (i.e., 114h) is the correct answer

Packed BCD subtraction example:

```
mov AL,71h
sub  AL,43h    ; AL := 2Eh
das            ; AL := 28h
```

Assignment #2

C functions you may use in your assembly code:

- char *fgets(char *str, int n, FILE *stream) // use fgets(buffer, BUFFERSIZE , stdin) to read from standard input
- int fprintf(FILE *stream, const char *format, arg list …) // use fprintf(stderr, …) to print to standard error (usually same as stdout)
- int printf(char *format, arg list …)
- void* malloc(size_t size) // size_t is unsigned int for our purpose
- void free(void *ptr)

If you use those functions the beginning of your text section will be as follows (**no _start label**):

```
section .text
    align 16
    global main
    extern printf
    extern fprintf
    extern malloc
    extern free
    extern fgets
main:
    … ; your code
```

Compile and link your assembly file **calc.s** as follows:

```
nasm -f elf calc.s -o calc.o
gcc -m32 -Wall -g calc.o -o calc.bin   ; -Wall enables all warnings
```

Note: there is no need in c file. gcc will “connect” external c functions to your assembly program.