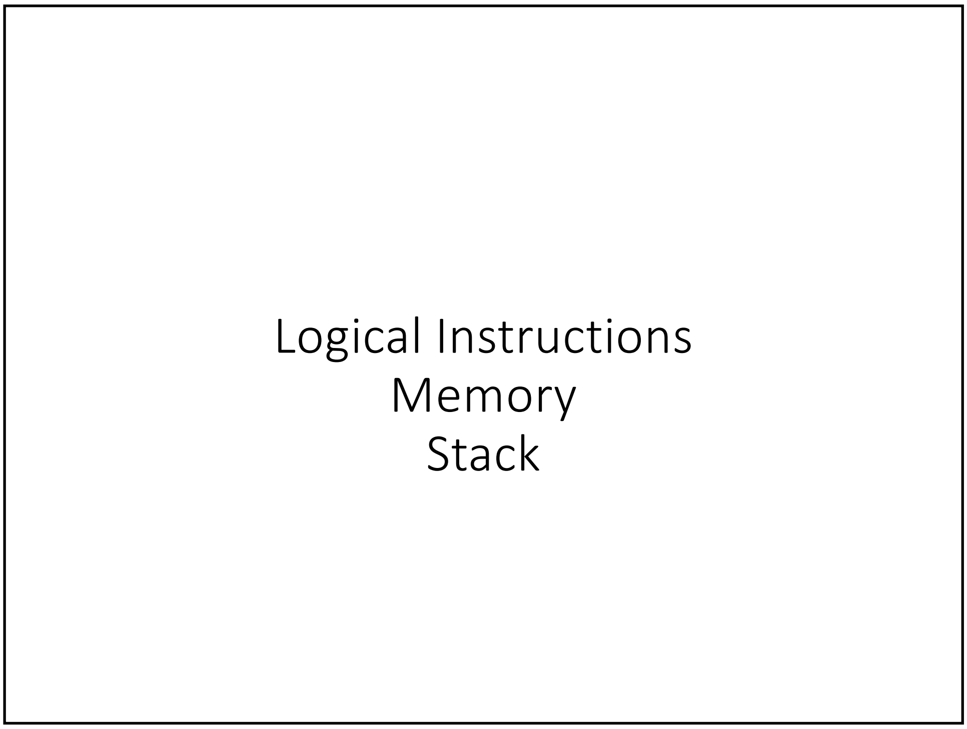
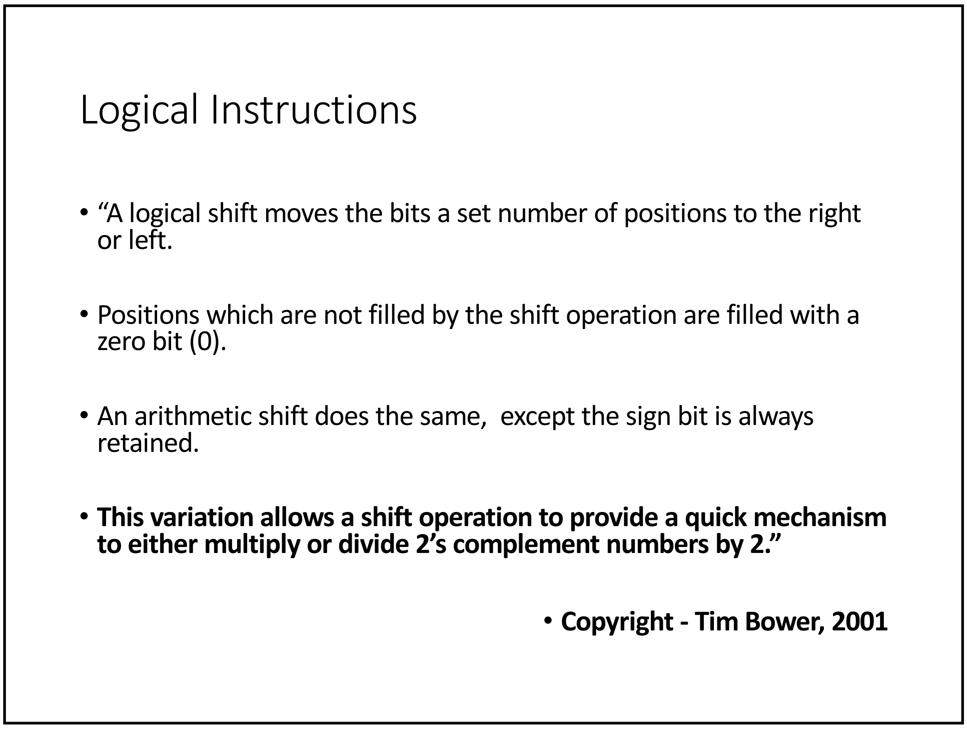


Logical Instructions
Memory
Stack

1



Logical Instructions

- “A logical shift moves the bits a set number of positions to the right or left.
- Positions which are not filled by the shift operation are filled with a zero bit (0).
- An arithmetic shift does the same, except the sign bit is always retained.
- **This variation allows a shift operation to provide a quick mechanism to either multiply or divide 2’s complement numbers by 2.”**

- **Copyright - Tim Bower, 2001**

2

Logical instructions

SHR *operand,value*

shift right the *operand* by *value* bits.

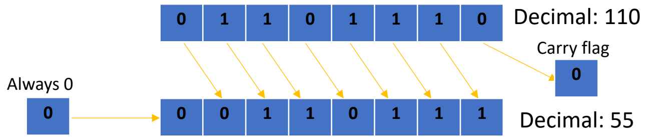
operand can be memory or register (8, 16 or 32 bits)

value can be a numerical value or CL register

put 0 in the left bit

divide the unsigned *operand* by 2^{value} (very fast division)

Example
MOV AL,01101110B
SHR AL,1



Logical instructions

SHL *operand, value*

shift left the *operand* by *value* bits.

operand can be memory or register (8, 16 or 32 bits)

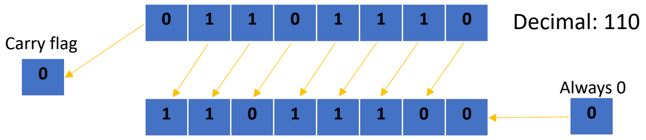
value can be a numerical value or CL register

put 0 in the right bit

multiply the *operand* by 2^{value} (very fast multiplication)

Decimal: 220
2's complemente:
011011100 (9 bits)

Example
MOV AL,01101110B
SHL AL,1



Logical instructions

SAR *operand, value* (shift arithmetic right)

shift right the *operand* by *value* bits.

operand can be memory or register (8, 16 or 32 bits)

value can be a numerical value or CL register

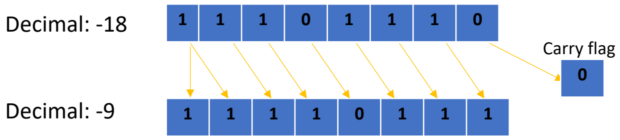
the left bit remains the same

divide the signed *operand* by 2^{value}

Note: the result for negative values may be different from the result using idiv

-9/4=-2 using IDIV and -3 using SAR -> rounding towards negative infinity.

```
MOV AL,11101110B
SAR AL,1
```



Logical instructions

SAL *operand, value* (shift arithmetic left)

shift left the *operand* by *value* bits.

operand can be memory or register (8, 16 or 32 bits)

value can be a numerical value or CL register

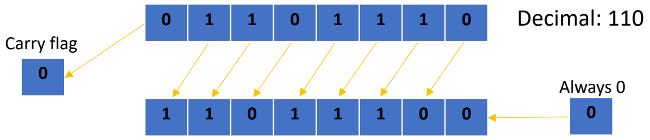
put 0 in the right bit

multiply the *operand* by 2^{value} (very fast multiplication)

Decimal: 220
2's complemente:
011011100 (9 bits)

Example

```
MOV AL 01101110B
SAL AL,1
```



Logical instructions

ROL *operand, value* (rotate left)

rotate left the *operand* by *value* bits.

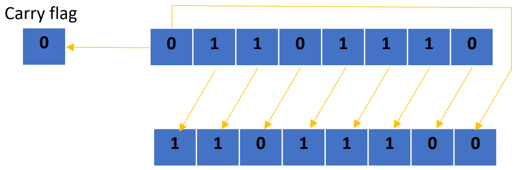
operand can be memory or register (8, 16 or 32 bits)

value can be a numerical value or CL register

Example

MOV AL, 01101110B

ROL AL,1



7

Logical instructions

ROR *operand, value* (rotate right)

rotate right the *operand* by *value* bits.

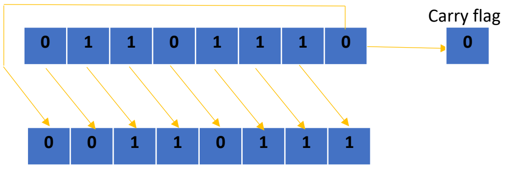
operand can be memory or register (8, 16 or 32 bits)

value can be a numerical value or CL register

Example

MOV AL, 01101110B

ROR AL,1



8

Logical instructions

AND *destination, source*

perform a bit by bit AND operation

destination and *source* have the same rules than mov

Example
MOV AL,01101110B
MOV BL,00110111B
AND AL,BL

	0	1	1	0	1	1	1	0	AL operand
AND	0	0	1	1	0	1	1	1	BL operand
	0	0	1	0	0	1	1	0	AL destination

9

Logical instructions

OR *destination,source*

perform a bit by bit OR operation

destination and *source* have the same rules than mov

Example
MOV AL,01101110B
MOV BL,00110111B
OR AL,BL

	0	1	1	0	1	1	1	0	AL operand
OR	0	0	1	1	0	1	1	1	BL operand
	0	1	1	1	1	1	1	1	AL destination

10

Logical instructions

XOR *destination,source*

perform a bit by bit XOR operation

destination and *source* have the same rules than mov

```
Example
MOV AL,01101110B
MOV BL,00110111B
XOR AL,BL
```

	0	1	1	0	1	1	1	0	AL operand
XOR	0	0	1	1	0	1	1	1	BL operand
	0	1	0	1	1	0	0	1	AL destination

Address generation unit

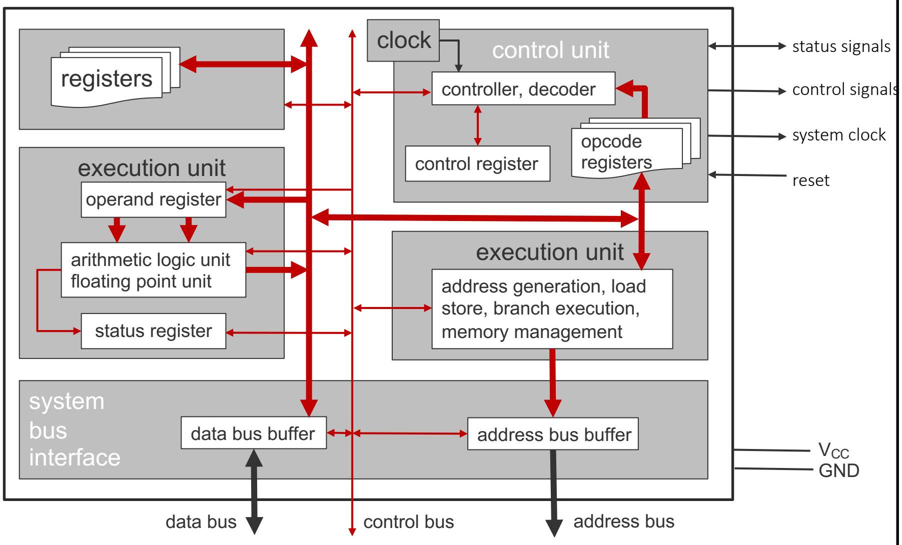
- Specialized part of the execution unit
- Basic operation: calculate an address based on control signals from the control unit and possibly additional content of registers
  - e.g. base pointer + index = address
- Can be very simple, but also very complex
  - MMU (memory management unit)
  - many different modes, memory protection, virtual address space
  - cache optimization, branch prediction, speculative loading etc.
  - covered later!

System bus interface

- The system bus interface (Bus Interface Unit, BIU) is the connection of the microprocessor to its environment (all the other components of a micro computer)

- **Purpose**
  - Buffering of addresses and data (operands and instructions)
  - Adaptation of clock cycles, bus width, voltages
  - Tristate: detaching the processor from the external bus

13



registers

execution unit

operand register

arithmetic logic unit
floating point unit

status register

clock

control unit

controller, decoder

control register

opcode registers

execution unit

address generation, load store, branch execution, memory management

system bus interface

data bus buffer

address bus buffer

status signals

control signals

system clock

reset

V_{CC}

GND

data bus

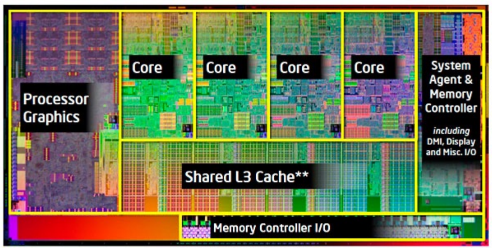
control bus

address bus

14

Additional components of a microprocessor

- Cache memory (fast memory for instructions and operands, covered later)
- Vector processing unit
- Graphics processor
- Signal processing unit
- Neural networks, AI support
- Interrupt controller



Address instructions

LEA *register, variable*

Load Effective Address put the address of *variable* in the *register*

Example

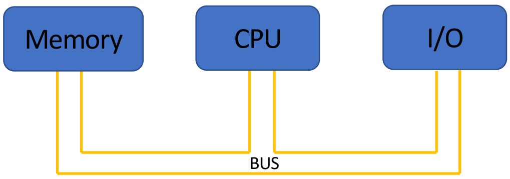
LEA EDI,lista; puts in register EDI the address of variable lista

After this line the following instructions are equivalent

MOV lista[0],AL;
MOV [EDI],AL;

Note: EDI – Extended Destination Index

Basic computer scheme

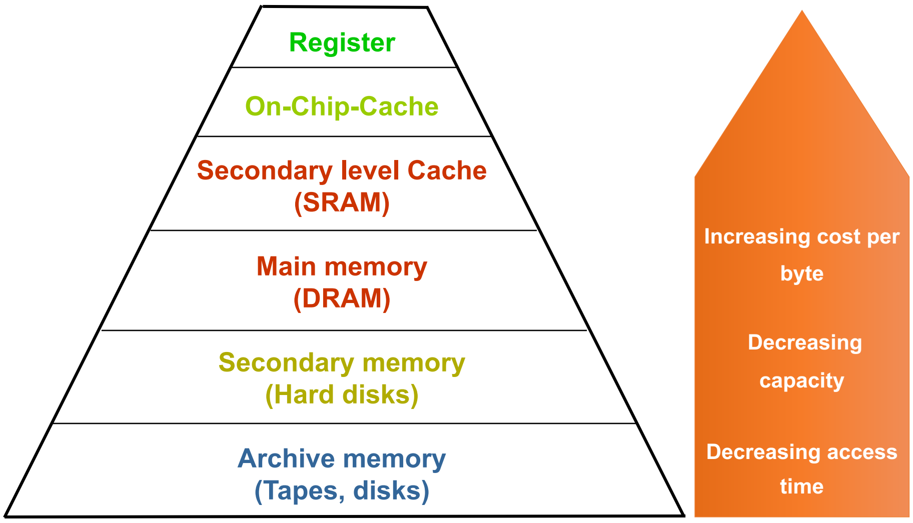

Memory

CPU

I/O

BUS

17


Register

On-Chip-Cache

Secondary level Cache
(SRAM)

Main memory
(DRAM)

Secondary memory
(Hard disks)

Archive memory
(Tapes, disks)

Increasing cost per byte

Decreasing capacity

Decreasing access time

18

Memory organization

- Memory is organized in bytes
- Binary is always the system!
- Each address as only one byte
- Each byte as only one address

- We can do only two operations in memory
  - Read
  - Write

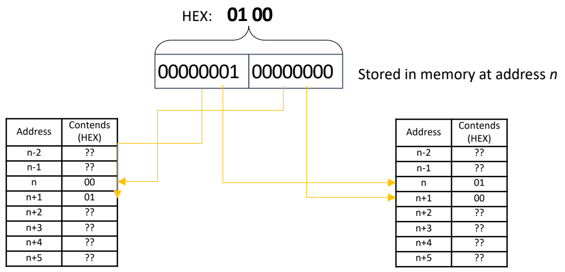
Address (binary)	Contends (HEX)
000	00
001	15
010	34
011	87
100	FA
101	FF
110	03
111	1F

19

Memory organization

How to represent data grater than 1 byte

Example: short integer (2 bytes), with value 256d



Stored in memory at address *n*

Address	Contends (HEX)
n-2	??
n-1	??
n	00
n+1	01
n+2	??
n+3	??
n+4	??
n+5	??

System **little endian**, used in the **Intel** architecture

The least significant byte is stored in the lower address memory

Address	Contends (HEX)
n-2	??
n-1	??
n	01
n+1	00
n+2	??
n+3	??
n+4	??
n+5	??

System **big endian**, used in the **RISC** architecture

The most significant byte is stored in the lower address memory

20

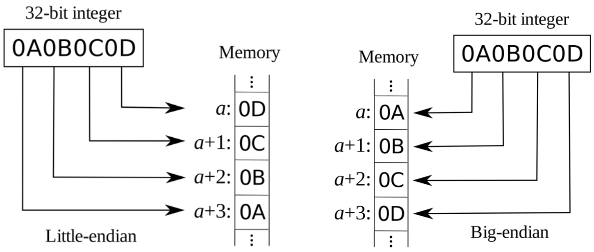
Memory organization

Exercises

Represent the following 4 bytes in little- and big-endian system at address *a*

0000 1010 0000 1011 0000 1100 0000 1101

0A 0B 0C 0D



21

Memory write and read operations

One single instruction in a **64 bits CPU** can write or read 1, 2, 4 or 8 bytes

Special instructions in a 64 bits CPU can also write or read 16, 32 or 64 bytes (128, 256 or 512 bits). **Details later.**

The number of bytes to write or read is defined by de operand size.

size 1 write\read 1byte

size 2 write\read 2bytes

size 4 write\read 4bytes

size 8 write\read 8bytes

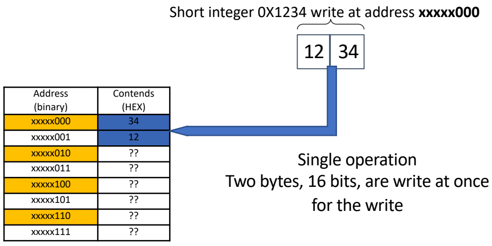
Isn't possible in a single instruction write or read 3, 5, 6 and 7 bytes

22

Memory align

The read and write operations are faster if the memory is aligned with operand size. Some systems must use always memory aligned

If the **operand size is 2** and the **address memory is multiple of 2**



Short integer 0X1234 write at address **xxxxx000**

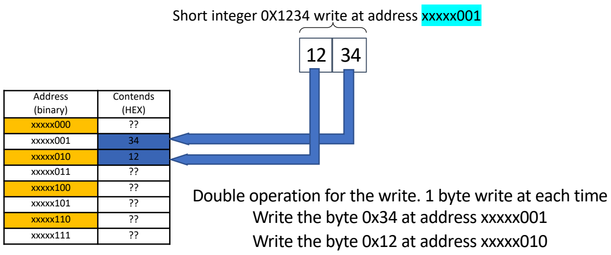
Address (binary)	Contends (HEX)
xxxxx000	34
xxxxx001	12
xxxxx010	??
xxxxx011	??
xxxxx100	??
xxxxx101	??
xxxxx110	??
xxxxx111	??

Single operation
Two bytes, 16 bits, are write at once
for the write

Memory align

The read and write operations are slower if the memory isn't aligned with operand size.

If the operand **size is 2** and **address memory isn't multiple of 2**



Short integer 0X1234 write at address xxxxx001

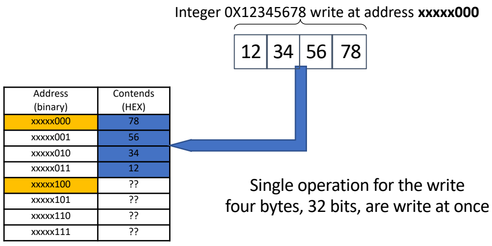
Address (binary)	Contends (HEX)
xxxxx000	??
xxxxx001	34
xxxxx010	12
xxxxx011	??
xxxxx100	??
xxxxx101	??
xxxxx110	??
xxxxx111	??

Double operation for the write. 1 byte write at each time
Write the byte 0x34 at address xxxxx001
Write the byte 0x12 at address xxxxx010

Memory align

The read and write operations are faster if the memory is aligned with operand size.

If the **operand size is 4** and the **address memory is multiple of 4**


Integer 0X12345678 write at address **xxxxx000**

Address (binary)	Contends (HEX)
xxxxx000	78
xxxxx001	56
xxxxx010	34
xxxxx011	12
xxxxx100	??
xxxxx101	??
xxxxx110	??
xxxxx111	??

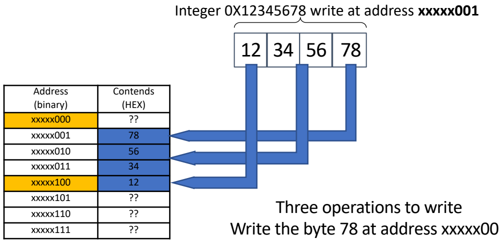
Single operation for the write
four bytes, 32 bits, are write at once

25

Memory align

The read and write operations are slower if the memory isn't aligned with operand size.

If the operand size is **4** and **address memory isn't multiple of 4**


Integer 0X12345678 write at address **xxxxx001**

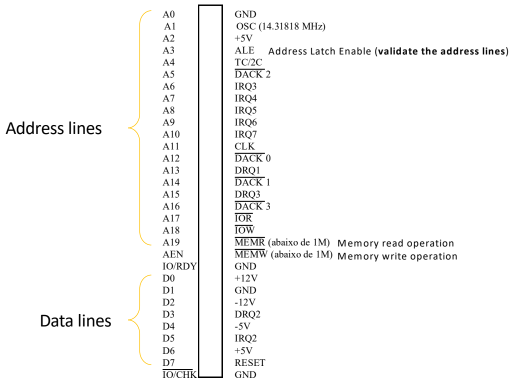
Address (binary)	Contends (HEX)
xxxxx000	??
xxxxx001	78
xxxxx010	56
xxxxx011	34
xxxxx100	12
xxxxx101	??
xxxxx110	??
xxxxx111	??

Three operations to write
Write the byte 78 at address xxxxx001
Write the bytes 3456 at address xxxxx010
Write the byte 12 at address xxxxx100

26

ISA BUS (8 bits)

very simple bus used since the 8086 CPU



27

Write cycle

(simplify)

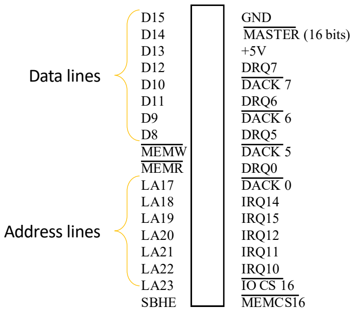
1. The CPU puts the address in the address lines
2. The CPU validate the address trough ALE signal
3. The memory decode the address lines
4. The CPU put the data in the data lines
5. The CPU activate the signal MEMW
6. The memory save the contends of data lines in the address position

28

Read cycle
(simplify)

1. The CPU puts the address in the address lines
2. The CPU validate the address trough ALE signal
3. The memory decode the address lines
4. The CPU activate the signal MEMR
5. The memory put in the data lines the data of the address position
6. The CPU reads the data from data lines

Extended ISA BUS (16 bits)


Data lines
Address lines
D15
D14
D13
D12
D10
D11
D9
D8
$\overline{\text{MEMW}}$
$\overline{\text{MEMR}}$
LA17
LA18
LA19
LA20
LA21
LA22
LA23
SBHE
GND
$\overline{\text{MASTER}}$ (16 bits)
+5V
DRQ7
$\overline{\text{DACK}}$ 7
DRQ6
$\overline{\text{DACK}}$ 6
DRQ5
$\overline{\text{DACK}}$ 5
DRQ0
$\overline{\text{DACK}}$ 0
IRQ14
IRQ15
IRQ12
IRQ11
IRQ10
$\overline{\text{IO CS}}$ 16
$\overline{\text{MEMCS16}}$

Stack

Stack is a special memory zone used to store data

Local variables and parameters are store in the stack

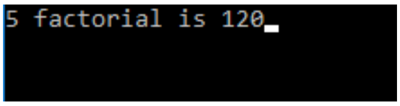
Each function call create a new space for your own stack

This makes possible recursive functions since local variables and parameters for each function call are unique

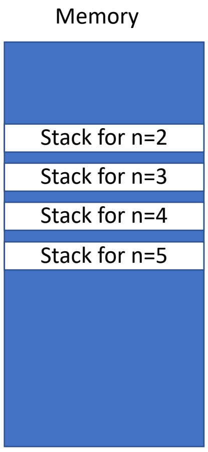
31

Stack

```
unsigned int factorial(unsigned int n)
{
   if (n <= 2)
      return(n);
   return(factorial(n - 1)*n);
}
int main(int argc, char* argv[])
{
   printf(“5 factorial is %d", factorial(5));
}
```



```
5 factorial is 120_
```



32

Stack

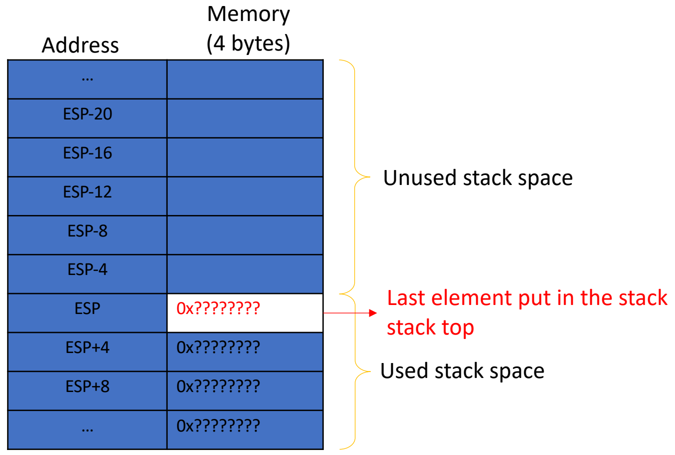
Stack is organized as a LIFO (Last In, First Out)

The register ESP (Stack Pointer) points, at any time, for the last element put in the stack (stack top)

Stack grows from the bottom (high address) to the top (low address)

Stack is organized in 4 bytes by element (exception later…)

Stack



Address	Memory (4 bytes)
…	
ESP-20	
ESP-16	
ESP-12	
ESP-8	
ESP-4	
ESP	0x????????
ESP+4	0x????????
ESP+8	0x????????
…	0x????????

Unused stack space

Last element put in the stack
stack top

Used stack space

ESP points always for the stack top

Stack instructions

PUSH *source*

Put in the stack the source

Source can be a register, memory or value

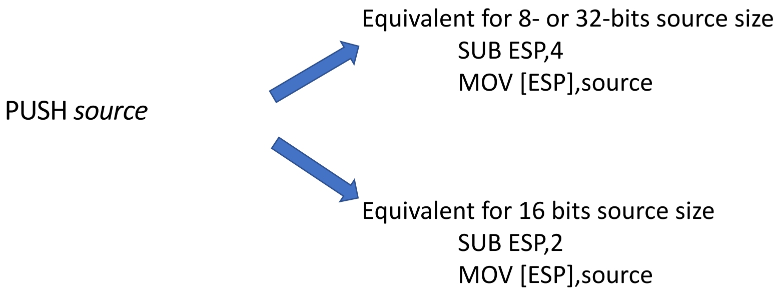
If the source size is 16bits (register or memory) the stack use only 2 bytes

In other situations, 8bits size or 32bits size, the stack use 4 bytes

Using values as *source* the stack use always 4 bytes

35

Stack instructions



PUSH *source*

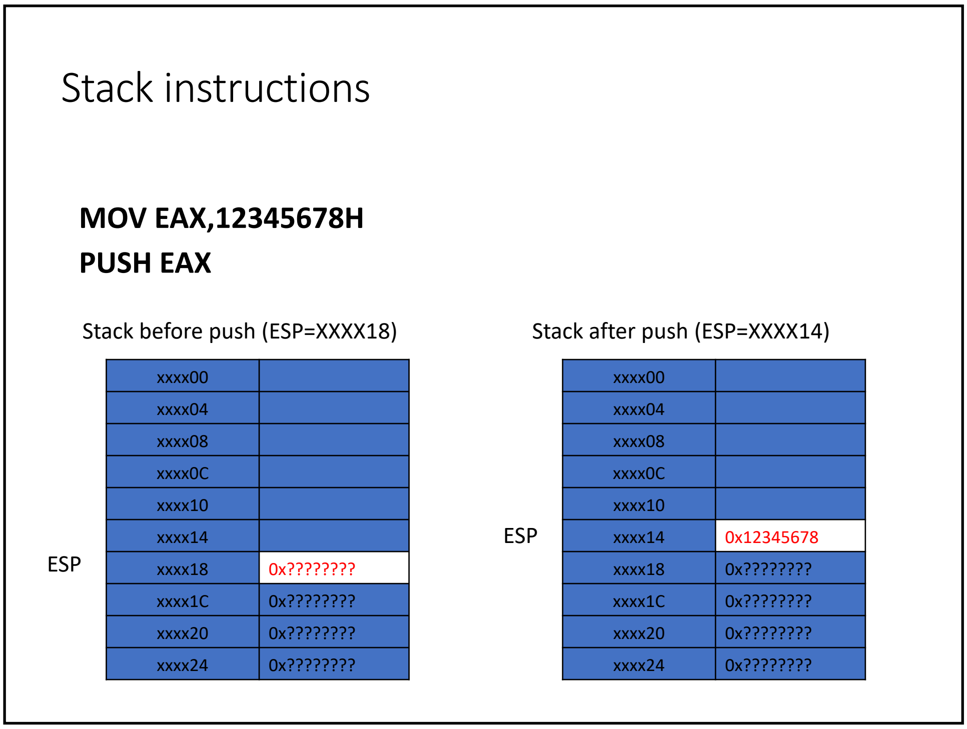
Equivalent for 8- or 32-bits source size

```
SUB ESP,4
MOV [ESP],source
```

Equivalent for 16 bits source size

```
SUB ESP,2
MOV [ESP],source
```

36



Stack instructions

MOV EAX,12345678H
PUSH EAX

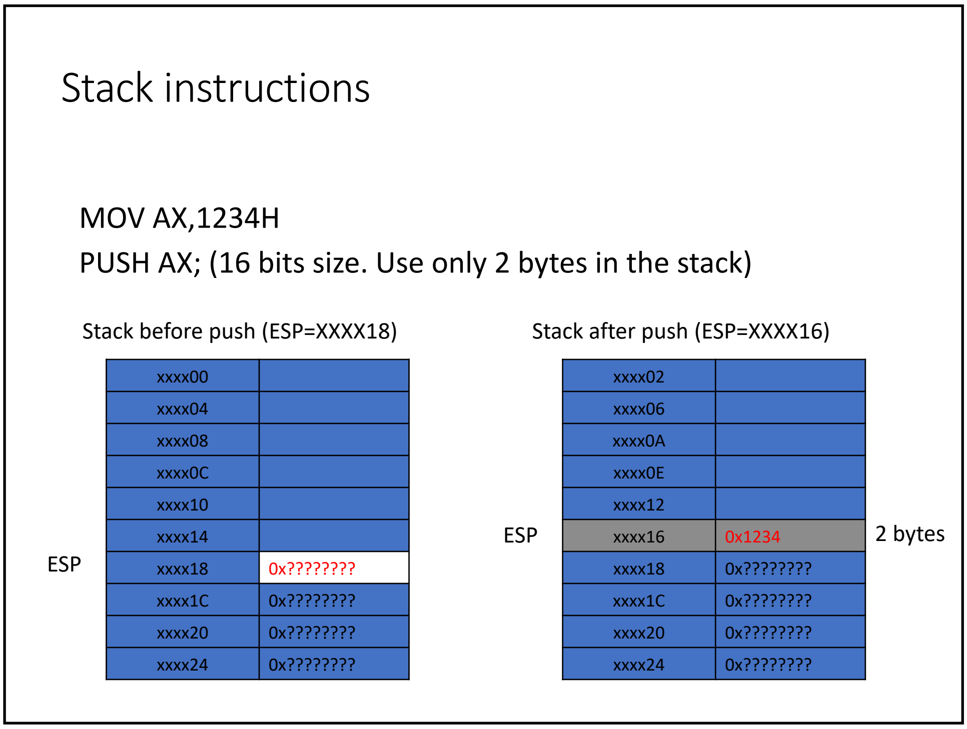
Stack before push (ESP=XXXX18)

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
	xxxx14	
ESP	xxxx18	0x????????
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

Stack after push (ESP=XXXX14)

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
ESP	xxxx14	0x12345678
	xxxx18	0x????????
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

37



Stack instructions

MOV AX,1234H
PUSH AX; (16 bits size. Use only 2 bytes in the stack)

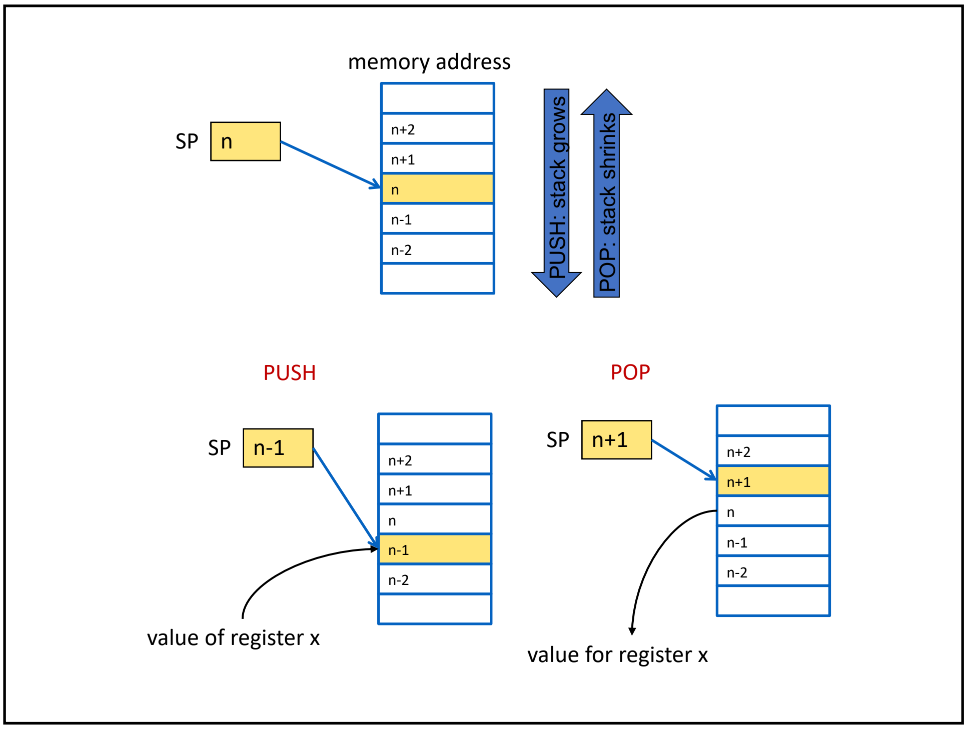
Stack before push (ESP=XXXX18)

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
	xxxx14	
ESP	xxxx18	0x????????
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

Stack after push (ESP=XXXX16)

	xxxx02		
	xxxx06		
	xxxx0A		
	xxxx0E		
	xxxx12		
ESP	xxxx16	0x1234	2 bytes
	xxxx18	0x????????	
	xxxx1C	0x????????	
	xxxx20	0x????????	
	xxxx24	0x????????	

38



Stack instructions

POP *destination*

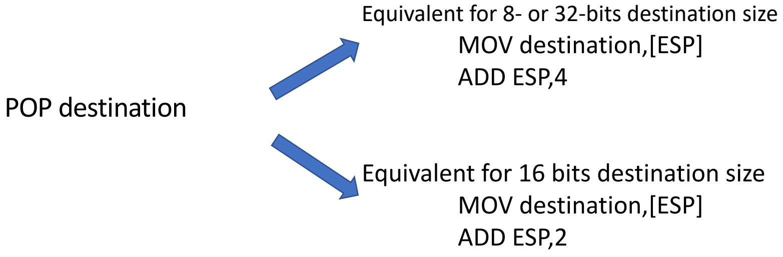
Get from stack the last element and put it in destination

Destination can be a register or memory

If the destination size is 16bits only 2 bytes are taken from stack

In other situations, 8bits size or 32bits size, 4 bytes are taken from stack

Stack instructions



POP destination

Equivalent for 8- or 32-bits destination size

MOV destination,[ESP]

ADD ESP,4

Equivalent for 16 bits destination size

MOV destination,[ESP]

ADD ESP,2

41

Stack instructions

POP EBX

Stack before pop (ESP=XXXX14)

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
ESP	xxxx14	0x12345678
	xxxx18	0x????????
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

Stack after pop (ESP=XXXX18)

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
	xxxx14	0x12345678
ESP	xxxx18	0x????????
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

42

Stack instructions

POP BX; (16 bits size. Use only 2 bytes in the stack)

Stack before pop (ESP=XXXX16)

	xxxx02		
	xxxx06		
	xxxx0A		
	xxxx0E		
	xxxx12		
ESP	xxxx16	0x1234	2 bytes
	xxxx18	0x????????	
	xxxx1C	0x????????	
	xxxx20	0x????????	
	xxxx24	0x????????	

Stack after pop (ESP=XXXX18)

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
	xxxx14	0x1234????
ESP	xxxx18	0x????????
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

43

Stack instructions

PUSH EAX
PUSH EBX
POP EAX
POP EBX

} Find what this code do

Stack after push EAX

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
	xxxx14	
ESP	xxxx18	EAX
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

Stack after push EBX

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
ESP	xxxx14	EBX
	xxxx18	EAX
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

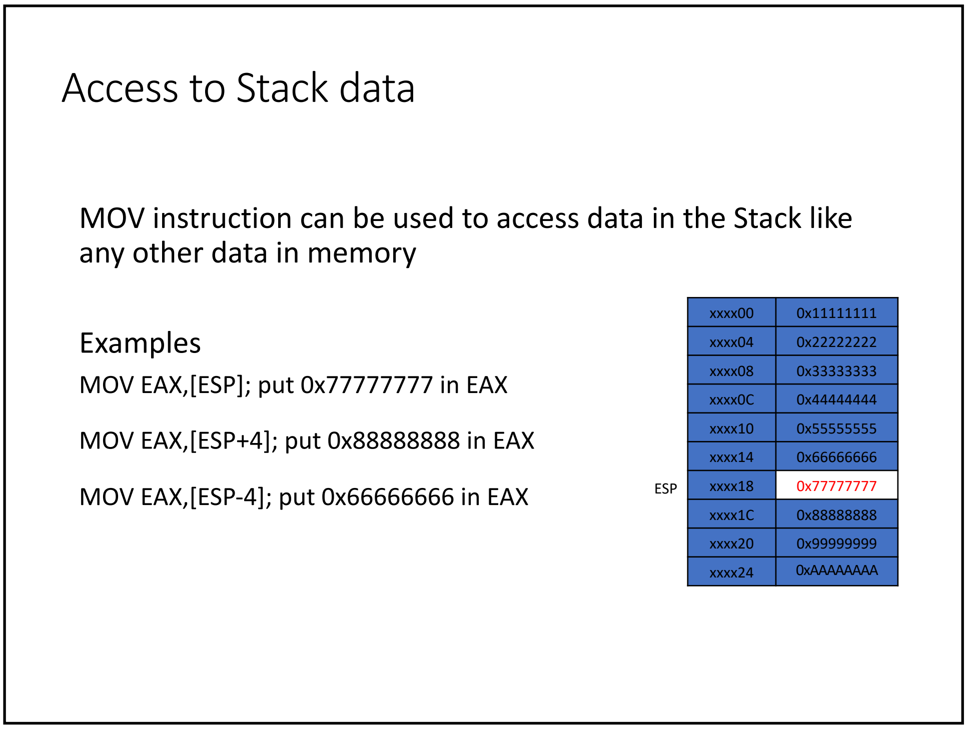
Stack after pop EAX

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
	xxxx14	EBX
ESP	xxxx18	EAX
	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

Stack after pop EBX

	xxxx00	
	xxxx04	
	xxxx08	
	xxxx0C	
	xxxx10	
	xxxx14	EBX
	xxxx18	EAX
ESP	xxxx1C	0x????????
	xxxx20	0x????????
	xxxx24	0x????????

44



Access to Stack data

MOV instruction can be used to access data in the Stack like any other data in memory

Examples

MOV EAX,[ESP]; put 0x77777777 in EAX

MOV EAX,[ESP+4]; put 0x88888888 in EAX

MOV EAX,[ESP-4]; put 0x66666666 in EAX

	xxxx00	0x11111111
	xxxx04	0x22222222
	xxxx08	0x33333333
	xxxx0C	0x44444444
	xxxx10	0x55555555
	xxxx14	0x66666666
ESP	xxxx18	0x77777777
	xxxx1C	0x88888888
	xxxx20	0x99999999
	xxxx24	0xAAAAAAAA