```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

ASSEMBLY PROGRAMMING

Imre Varga PhD

University of Debrecen

Faculty of Informatics

Department of IT Systems and Networks

Version 2.0.0

3 September 2023

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Warning

- A thorough knowledge of the C programming language and an ability to write algorithms are essential for the successful completion of the subject.
- Reviving them is not the task of this slides.
- Self-preparation is required.

```
int FastSum(int *A, int N){
  int i, s=0;
  for(i=0;i<N;i++)
    s+=A[i];
  return s;
  }
```

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Number representation, datatypes

Fixed-point representation

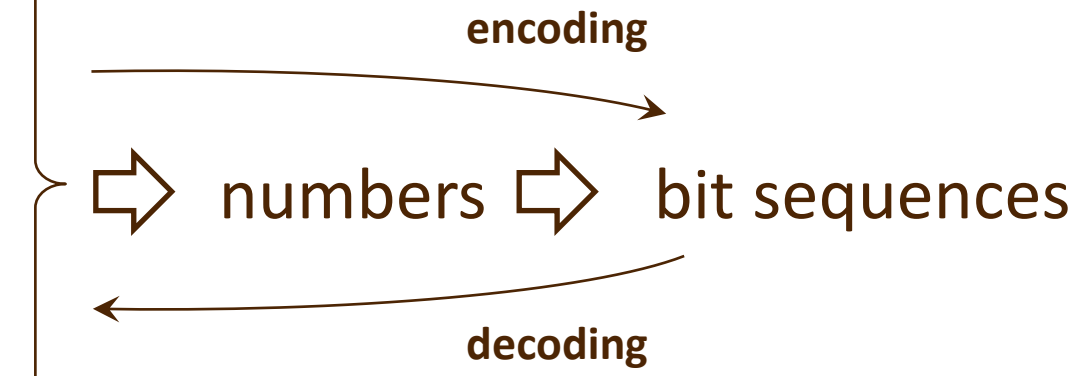
Floating-point representation

Main datatypes

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Data

- Information is stored as different type datas
- Computer store everything as bit sequences
  - numerical value
  - text
  - image
  - voice/sound
  - video
  - database
  - program



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Binary numbers

- Only two digits: 0 and 1
- Significance: 2 states (yes/no, true/false, etc.)
- Conversion $69.375_{10} = 1000101.011_2$

69	2
34	1
17	0
8	1
4	0
2	0
1	0
0	1

2	375
0	750
1	500
1	**000**

64	32	16	8	4	2	1	1/2	**1/4**	**1/8**
2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
1	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Hexadecimal numbers

- 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Close relation to binary, but shorter form
- Notation: $1FA_{16}$, 0x1FA, $1FA
- Conversion

1*256	15*16	10*1	**=506_{10}**
16^2	16^1	16^0	
1	**F**	**A**	

0	0	0	1	1	1	1	1	1	0	1	0	
2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
2048	1024	512	256	128	64	32	16	8	4	2	1	**=506_{10}**

0x1FA = 0b111111010 = 506

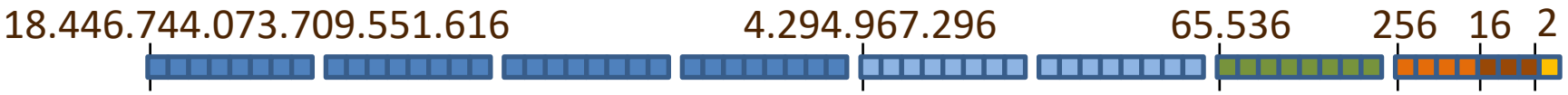
```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Organization of data

Data units

- Bit: **two states** (0, 1)
- Nibble (half byte): 4 bits
- Byte: 8 bits
- Word: (mostly) 16 bits
- Double word : 32 bits
- Quad word: 64 bits

One addition bit means double possible values



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Data types

Different cases have different data expectations so, we introduce data types

Necessary components

- Representation: How to encode?
- Domain: Which values can be used?
- Operations: What is it used for?

Main datatypes

- integer, real, character, pointer, logical, array, record, string

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Integer type

Representation

- „Sign-and-magnitude” (not practical)
- **Fixed-point** (2, 4 or 8 bytes)
  - ➥ Signed or unsigned
- BCD
  - ➥ Packed or unpacked

Different forms

- E.g., C language:
  30=30u=30l=036=0x1E=0b11110*

*ISO C99

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Fixed-point data representation

- For storing integer values
- **Unsigned** (only not negative) case
  - Conversion to binary
  - Add leading zeros to reach the appropriate size
- Example (in case of 1 byte)

  $41_{10} \rightarrow 101001_2 \rightarrow$ | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
- Representation on N bits
  - Lowest value: 0
  - Highest value: 2^N-1

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Fixed-point data representation

- **Signed** case
  - In case of not negative value: as unsigned case
  - In case of negative value: **two's complement**
    - Inversion of bits in absolute value (one's complement)
    - Increasing the result by 1
- Example (in case of 1 byte)

  $-41_{10} \rightarrow -101001_2 \rightarrow$

- Representation on N bits
  - Lowest value : $-(2^{N-1})$
  - Highest value: $2^{N-1}-1$

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Integer types of the C language

- Fixed-point representation

type	size	domain	
[signed] char	1 byte	-128	127
unsigned char		0	255
[signed] short	2 byte	-32.768	32.767
unsigned short		0	65.535
[signed] int	4 byte (2 byte)	-2.147.483.648	2.147.483.647
unsigned int		0	4.294.967.295
[signed] long	8 byte (4 byte)	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
unsigned long		0	18.446.744.073.709.551.615
[signed] long long*	8 byte	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
unsigned long long		0	18.446.744.073.709.551.615

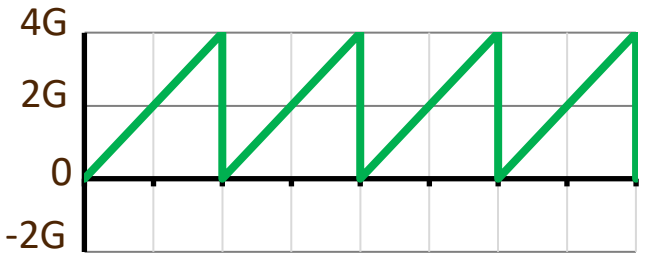
```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Arithmetic problems

Two special situations

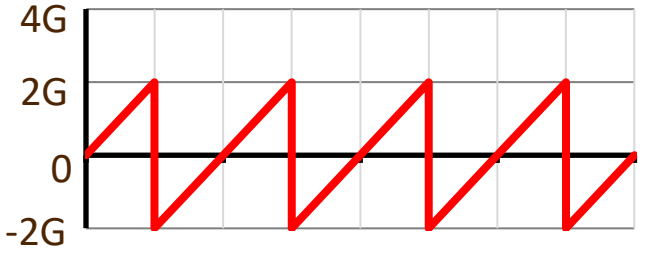
- Unsigned: more bit needed (**carry**)

```
 01011010     90
+11110001   +241
-----------------
 101001011    75 ≠331 >255
```



- Signed: unexpected sign change (**overflow**)

```
 01011010     90
+01110001   +113
-----------------
 11001011    -53 ≠203 > 127
```



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Extension

Storing data on more bits than earlier without change of the value

- **Zero extension**: filling with zeros
  - Long unsigned from shorter unsigned value

    8 bits: 01101001 → 16 bits: 0000000001101001

    8 bits: 11010010 → 16 bits: 0000000011010010

    ```
    unsigned short x=123; unsigned long y=x;
    ```
- **Sign extension**: filling with copy of sign bit
  - Long signed from shorter signed value

    8 bits: 01101001 → 16 bits: 0000000001101001

    8 bits: 11010010 → 16 bits: 1111111111010010

    ```
    short int q=-123; long int z=q;
    ```

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Byte order

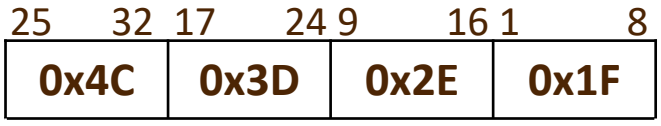
Order of bytes if data is more bytes long

- **Little-endian**: (host byte order)
  - ➥ Started with Least Significant Byte (LSB)
- **Big-endian**: (network byte order)
  - ➥ Started with Most Significant Byte (MSB)

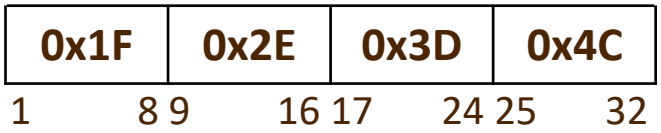
Example:

523124044 = 0b11111 00101110 00111101 01001100 = 0x1F 2E 3D 4C

Little-endian:



Big-endian:



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Real type

- Representation: **floating point**
- Size: 4, 8, 10 byte
- Operations
  - Operator overload
    E.g. the + operator can be integer-integer or real-real
  - Some operators are not allowed
    E.g.: modulo, shifting
- Different forms
  - E.g., in C language:
    0.25, 0.25f, 0.25L, .25, +0.25, 25e-2, 0.025e1

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Floating point representation

- For real (not integer) numbers
- Base is the normalization: $123.45 = 1.2345 \cdot 10^2$
- All (binary) can be written as:

$$(-1)^S \cdot M \cdot B^E$$

  - ➥ Mantissa (M) has „1.F” form, so $1_2 \leq M < 10_2$
  - ➥ Exponent (E) can be positive, negative or zero
- Don’t store: Base (B=2), hidden bit, sign of E
  - ➥ K=E+N is not negative (N: bias, null point)
- Store: S, F, K $(-1)^S \cdot 1.F \cdot A^{K-N}$

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Floating point standard

IEEE 754/1985 standard (ISO/IEC/IEEE 60559:2011)

- Sign
  - Positive: 0; Negative: 1
- Formats

	length	S size	K size	F size	N
Single precision	32 bits	1 bit	8 bits	23 bits	127
Double precision	64 bits	1 bit	11 bits	52 bits	1023
Extended precision	80 bits	1 bit	15 bits	63(+1) bits	16383
Half precision	*16 bits*	*1 bit*	*5 bits*	*10 bits*	*15*

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Floating point representation

Example: -12.8125 single precision

$-12.8125_{10} = -1100.1101_2 = -1^1 \cdot 1.1001101 \cdot 10^{11}$

$(-1)^S \cdot M \cdot B^E$

$(-1)^S \cdot 1.F \cdot 2^{K-127}$

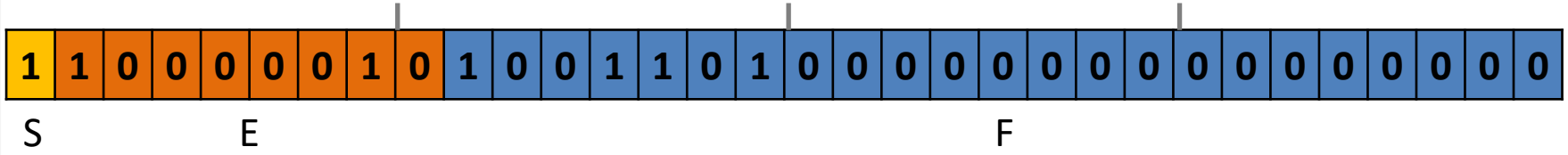
Sign: S = **1**

Exponent: $E = 3_{10} = 11_2$

Mantissa: M = 1.1001101

Fraction field: F = **10011010000000000000000**…

Biased exponent: $K = E+N = 3_{10}+127_{10} = \mathbf{10000010_2}$



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Rounding error

- Reason: finite amount of bits
- Too many useful bits rounding necessary
  - If the last stored bit would be ’0’, then truncate the remaining bits
  - Else round up
- Rounding error: not precise representation
- Example: 0.3 with single precision

  useful bits: 00110011001100110011001100110…

  stored bits: 00110011001100110011010

  0.3 → 0.300000011920929

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Rounding error

- Real numbers:
  'continuous' set with infinite amount of values
- Floating point numbers:
  discrete set with finite amount of values
- Density of representable numbers are not constant. Examples on 4 bytes:

```
1.0000000000; 1.0000001192; 1.0000002384; 1.0000003576
1000000000.0; 1000000064.0; 1000000128.0; 1000000192.0
```

- Just a few digit of stored values are exact (finite precision)

π = 3,141592653… ≠ 3,141592741… (float)

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Real types of the C language

- Floating point representation

type	size	Domain		
		minimum	maximum	accuracy
float	4 byte	$1.18 \cdot 10^{-38}$	$3.40 \cdot 10^{+38}$	≈7 digits
double	8 byte	$2.23 \cdot 10^{-308}$	$1.80 \cdot 10^{+308}$	≈15 digits
long double*	10 byte	$3.36 \cdot 10^{-4932}$	$1.19 \cdot 10^{+4932}$	≈19 digits

*C99

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Special floating point values

- +0 and -0
  sign bit ’0’ (+) or ’1’ (-), all other bits ’0’
  E.g.: 0.0/-1.0, -1.0*0.0
- ±Infinite (inf, 1.#INF00)
  Biased exponent full ’1’, useful significand full ’0’
  E.g.: 1.0/0.0, Inf+1.0, too large value
- „Not a Number” (NaN, 1.#IND00)
  Biased exponent full ’1’, useful significand not full ’0’
  Pl.: 0.0/0.0, Inf-Inf, 0.0*Inf, NaN+1.0
- Denormalized values (subnormal)
  Biased exponent full ’0’, useful significand not full ’0’

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Calculation with floating point values

- Addition: different method from fixed-point
- Example: 7.5 + 0.625

7.5 = 01000000 11110000 00000000 00000000

0.625 = 00111111 00100000 00000000 00000000

10000001 > 01111110 (3)

11110000…
+ 10100…
100000100…

1+10000001 = 10000010

01000001 000000100 00000000 00000000

$(-1)^0 * 1.000001 * 10^{11} =$ **8.125**

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Example: understanding bit sequence

What is the meaning of this little-endian bits?

`11101001 10110000 10100101 01000110`

- 32-bit signed integer: -374 299 322
- 32-bit unsigned integer: 3 920 667 974
- 16-bit signed integers: -5 712; -23 226
- 16-bit unsigned integer: 59 824; 42 310
- Floating point: -266939282492416730000000000.0
- ”Latin 1” text: é°¥F
- ”Latin 2” text: é°ĽF
- Unicode text with UTF8 encoding: 鰥F
- Packed BCD: ?9?0?546 (invalid) or +9-0+546

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Array type

- **Continuously** sored elements of same type
- All element have the same representation
- Name of the array
  - C language: named constant pointer to first element
  - Some languages: reference to all elements
- Address of i^{th} element of a 1D array (A_i)

$$A_i = A_1 + (i - 1)E$$

where E is the size of an element A_1 is the starting address of array

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Assembly programming

Assembly language

Instruction set

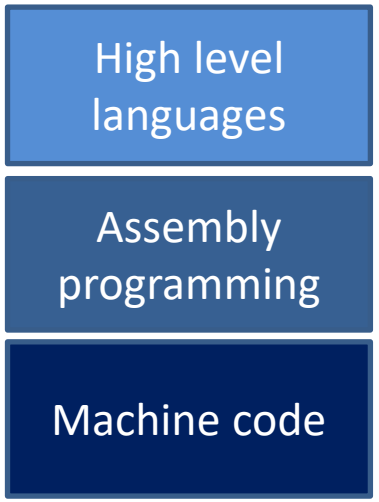
Addressing modes

Machine code

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Assembly programming

- Low-level abstraction, elementary instructions
- Need for hardware knowledge
  - ➥ Platform dependent
- Liberty of programmers
- Ability to optimize code for task
- Higher performance
- Code reading difficulties
- PC, microcontroller



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Structure of instructions

- 4-address instruction

Operation code	1st operand address	2nd operand address	Result address	Next instruction address

- 3-address instruction

Operation code	1st operand address	2nd operand address	Result address

- 2-address instruction

Operation code	1st operand + result address	2nd operand address

- 1-address instruction

Operation code	2nd operand address

- 0-address instruction (e.g., stack-based architectures)

Operation code

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Structure of instructions

Source file

- instruction_1
- instruction_2
- instruction_3 ↔
- instruction_4
- instruction_5
- …

Label	Operation	Operand(s)	Comment
.L1:	mov	eax, 0	# zero into eax

Label	Identifier, closing with colon
Operation	**Mnemonic** of activity
Operand(s)	Data or reference of data
Comment	Until the end of line, compiler skip it

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

ISA

Instruction Set Architecture

Details of computer related to programming

- Register* set
- Word width
- Machine instructions
- Addressing modes
- Memory architecture
- Interrupt handling

*Small capacity, fast access storage in processor.

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Register set

- Registers: Small capacity, fast access storage circuits in processor, to store for instance the operands and results of operations.
- Important:
  size, count, name, role, ect.
- Few examples:
  - ➥Main 32-bit registers of x86 architecture:
    `eax, ebx, ecx, edx, esp, ebp, edi, esi, eip, eflags`
  - ➥Main 32-bit registers of ARM architecture :
    `r0, r1, r2, r3, … r12, SP, LR, PC, CPSR`

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Types of instructions

- Data transfer instructions
- (Integer) arithmetic instructions
- Bitwise logical instructions
- Shift instructions
- Control flow instructions
- String instructions
- BCD and float arithmetic instructions
- Compiler directives
- Other instructions

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Types of instructions

Data transfer instructions

- Register-register (mov, xchg)
- Register-memory (mov)
- Register-I/O port (in, out)
- Register-stack (push, pop)
- Extension (width important) (cbw, cwd, cdqe)
- Setting status bit (sti, cli)

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Types of instructions

Arithmetic instructions

- Addition (with/without carry) (add, adc)
- Subtraction (with/without carry) (sub, sbb)
- Incrementation, decrementation (inc, dec)
- Two's complement (neg)
- Signed/unsigned multiplication (mul, imul)
- Signed/unsigned division (div, idiv)
- Comparison (cmp)

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Types of instructions

Bit-wise logical instructions

- AND operation (and)
- OR operation (or)
- EXCLUSIVE OR operation (xor)
- One's complement (not)
- Logical/unsigned left shift (shl)
- Logical/unsigned right shift (shr)
- Arithmetic/signed right shift (sar)
- Left/right rotation (ror, rol)
- Left/right rotation through carry (rcr, rcl)

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Types of instructions

Control flow instructions

- Unconditional jump (jmp)
- Conditional jump (je, jne, jg, jge, jl, jle, ja, jb, jc, jo…)
- Looping instruction (loop, loopz, loopnz)
- Invoking instruction (call)
- Return from subroutine (ret)
- Software interrupt (int)
- Return from interrupt (iret)

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Types of instructions

String (byte sequence) instructions

- String(component) transfer (movs, movsw, movsd)
- String (component) compare (cmps)
- Search in string(component) (scas)

Other instructions

- Floating point (fld, fst, fadd, fsqrt, …)
- System control (hlt)
- „Empty” instruction (nop)
- Information (cpuid)

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Types of instructions

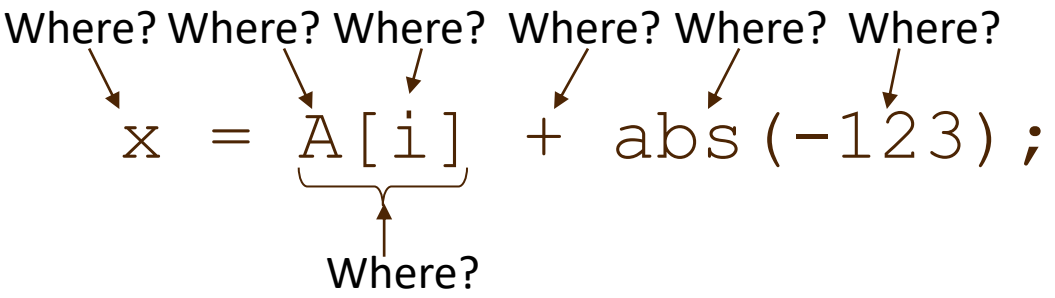
Compiler directives: no machine code, effect on compilation process

- Allocation (.byte, .comm, .zero)
- Scope (.globl, .local)
- Syntax definition (.intel_syntax)
- Definition of memory segments (.text, .data, .bbs)
- Substitute symbol (.set)
- …

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Need for memory access

- Calculations happen in processor
- Instructions and data are in the memory
- Transfer between CPU and RAM needs address
  - Processor has to know what is where in memory
  - Addresses are either stored in registers or provided by calculations



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Addressing modes

How can we access data in memory?

- Implicit, implied
- Immediate
- Direct, absolute
- Register direct
- Indirect
- Register indirect
- Indexed address
- Register relative address
- …

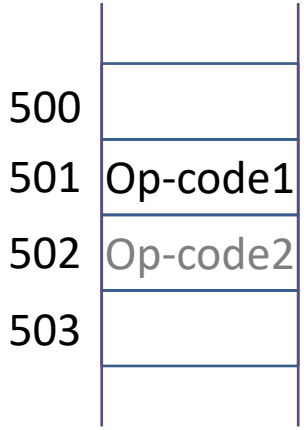
```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Addressing modes

Implicit/implied addressing

- No real address
- E.g., if no operand

RAM:

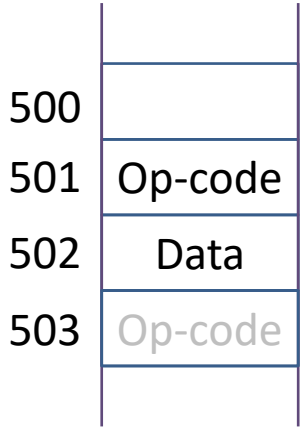


```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Addressing modes

Immediate data

- Data behind operation code
- This is the operand
- Constants in code

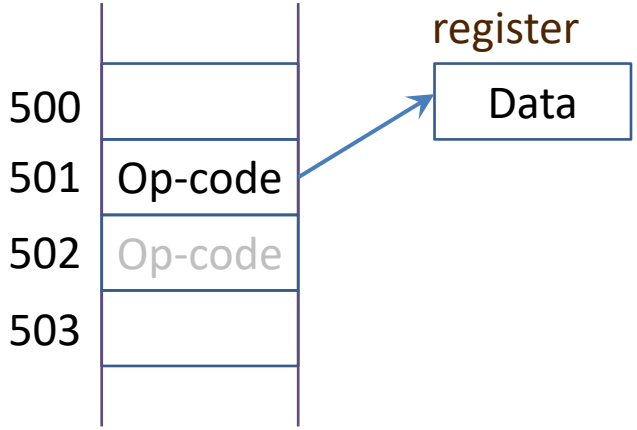


```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Addressing modes

Register direct

- Operation code contains reference to a register
- Operand in this register

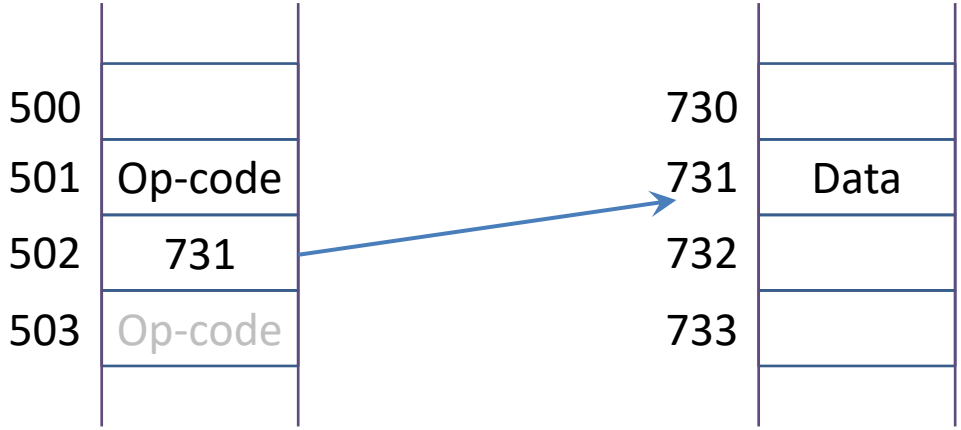


```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Addressing modes

Direct/absolute addressing

- An address behind the operation code
- Operand is located in this address

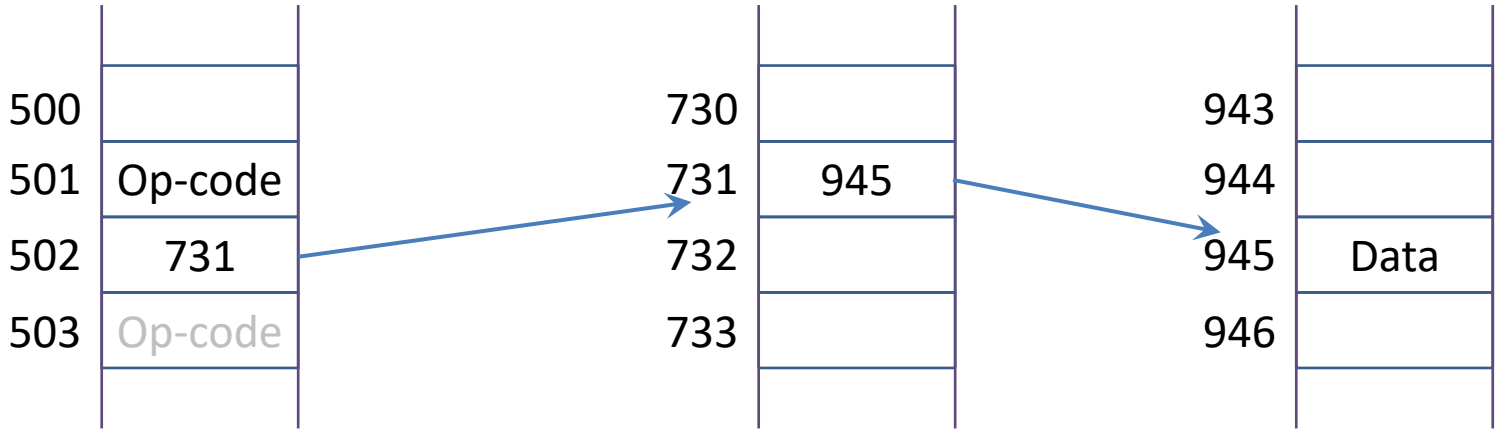


```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Addressing modes

Indirect addressing

- An address is behind the operation code
- The address of operand is stored in this address

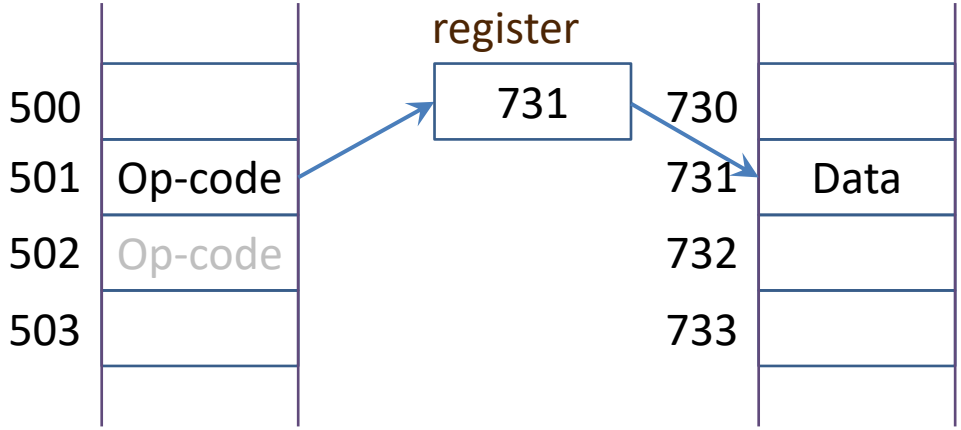


```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Addressing modes

Register indirect addressing

- Operation code refers to a register
- The address of operand is in the register

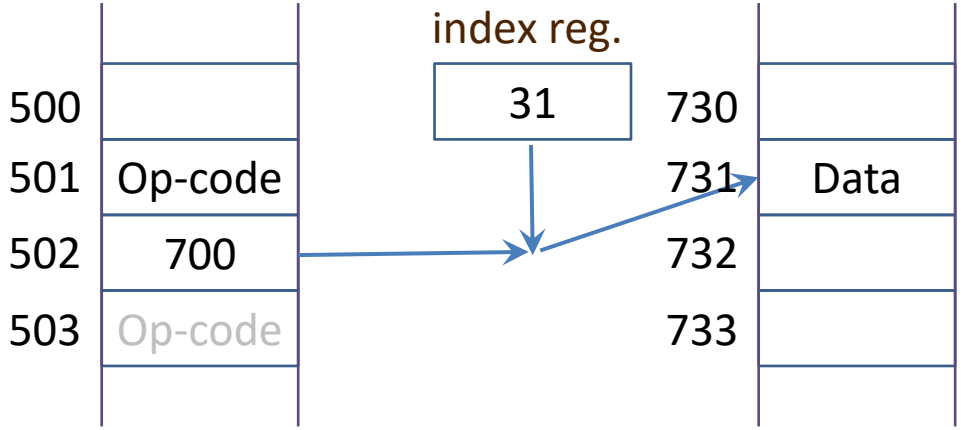


```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Addressing modes

Indexed addressing

- An address is behind the operation code
- Add the content of index register to the address
- The sum is the address of operand

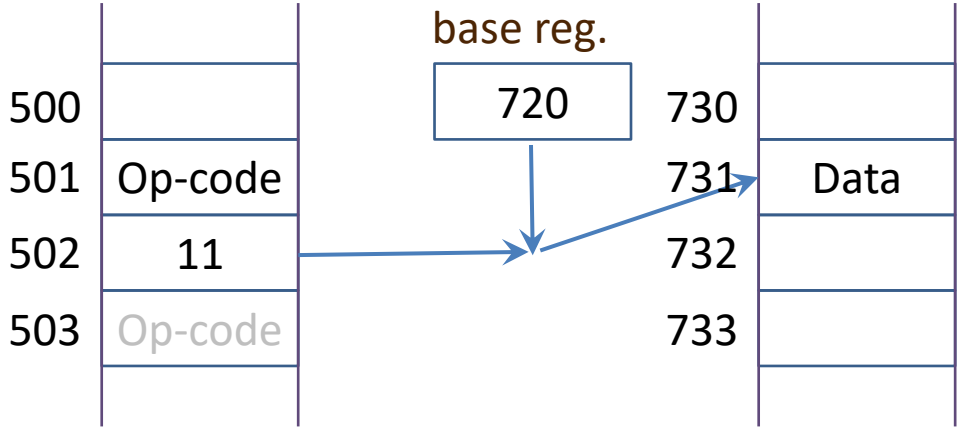


```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Addressing modes

Register relative addressing

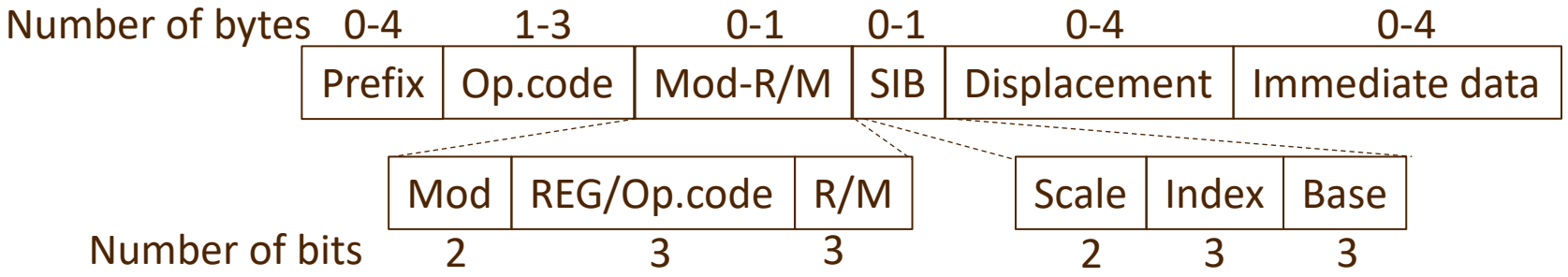
- An offset value is behind the operation code
- Add the content of base register to offset value
- The sum is the address of operand



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Machine code

- Only language understandable for processor
- Binary format
- Operation, addressing mode, operand size, operand type (reg./mem./const.), etc.
- Can vary from processor to processor
- Must be easily decodable by the hardware
- Format in x86 architecture:



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Machine code example

C language code

```
y=5;
z=6;
x=z+y*7;
```

Assembly code (x86)

```
mov   DWORD PTR [ebp-12], 5
mov   DWORD PTR [ebp-8], 6
mov   ebx, DWORD PTR [ebp-12]
mov   eax, ebx
sal   eax, 3
sub   eax, ebx
add   eax, DWORD PTR [ebp-8]
mov   DWORD PTR [ebp-16], eax
```

Machine code

c7	45	f4	05	00	00	00	c7	45	f8	06	00	00	00	
8b	5d	f4	89	d8	c1	e0	03	**29**	**d8**	03	45	f8		
89	45	f0												

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Schematic structure of computers

Architecture levels

Processor

Bus system

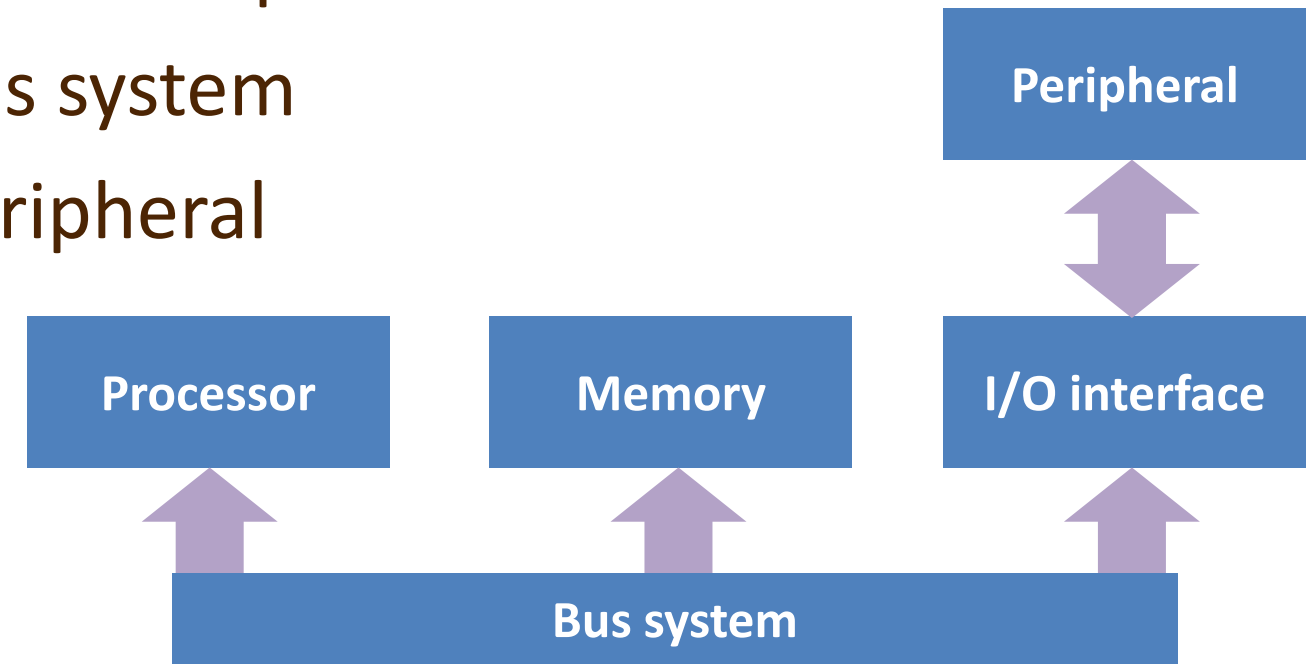
Memory

Peripheral

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Schematic structure of computers

- Processor
- Memory
- Input-Output interface
- Bus system
- Peripheral



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Processor

- **Central Processing Unit** (CPU)
- The „brain” of the computer
  - Controlling
  - Instruction execution

AMD Sempron processor

- (Micro)processor contains integrated circuits
- For synchronization of operations and units it uses a clock signal


Intel Core i3 processor

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Memory

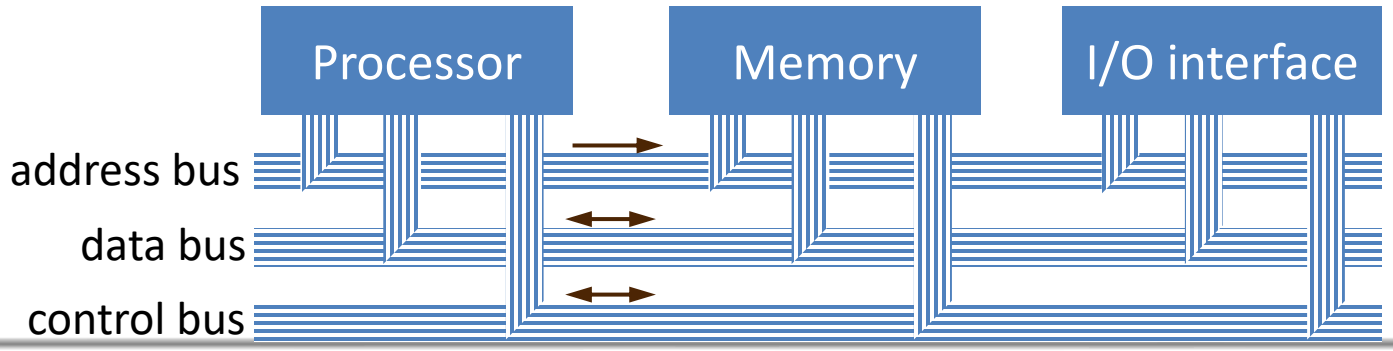
- Operative storage
  - „short-term” memory for the actual operations
  - Primary storage
- Data „cells” are addressable
- Storing data and instructions as well
- Electronic operation
  - Integrated circuits

DDR RAM

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Bus system

- Bus: a group of wires to transmit digital signals
- Logical component of bus systems
  - Address bus: to transmit addresses
  - Data bus: carries data bits
  - Control bus: transmit signals to synchronize the operation of components



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

The processor: structure and operation

CPU

Registers

Fetch-execute cycle

RISC / CISC processors

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Processor

- Central Processing Unit (CPU)
- Parts:
  - Control Unit (CU)
  - Arithmetic Logic Unit (ALU)

  } Execution Unit (EU)
  - Addressing Unit (AU) and Bus Interface Unit (BIU)
  - Registers
  - Inner bus system
  - Inner cache
  - Other (E.g., clock generator)

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

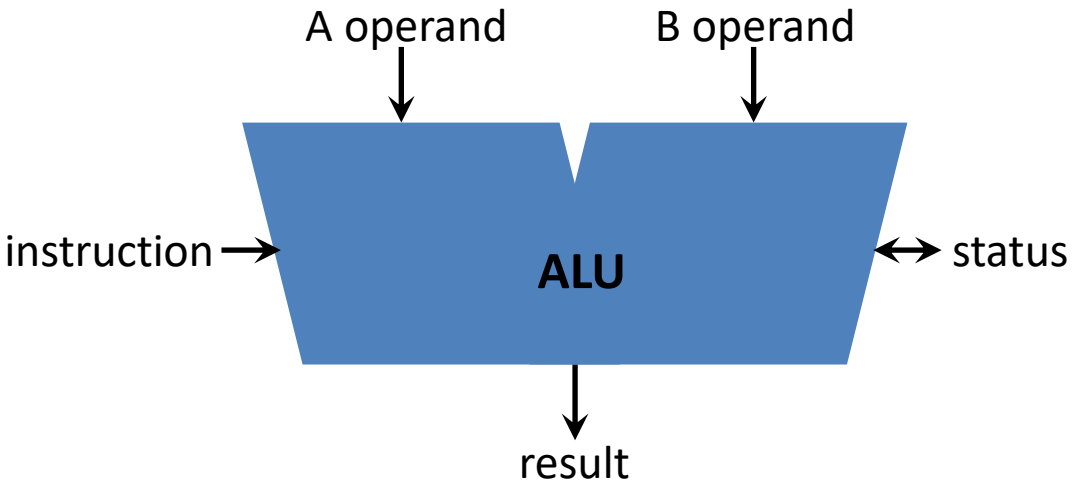
Registers

- Small (flip-flop-like) storage circuit
  - Size often equal to width of data bus
  - Generally, it can store 8-512 bits
- Fast access (access time < 1ns)
- Their number depends on the processor (10-100)
- They build register files/blocks
- Sometimes renamable
- 3 categories:
  - System- , general purpose- and special purpose reg.

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Arithmetic logic unit

- Performs calculations
- Contains: fixed-point adder, complement composer, shift registers, bitwise and logic operation circuit, etc.



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Control unit

- Based on the content of IR controls/governs operation of other units (E.g., ALU)
- Important registers: IR, PC, SR
- The control can be …
  - Hard-wired (direct) way
    Execution of every instruction is based complex digital electronical circuits
  - Microprogramed (indirect) way
    All operation code launch a tinny microprogram (stored in ROM)

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Addressing unit

Addressing Unit (AU), Address Generation Unit (AGU)

- Instructions have several addressing modes to find out the address of operands
- The AU places the address of operands into the MAR
- References in instructions are mapped to „physical” memory address
- Handling memory protection errors
- Related to BIU

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Operation of CPU

- Iteration of same atomic operations
  - Fetch-execute cycle
- Synchronized by the clock
- CU controls
- Infinite, monotonic, mechanic iteration of …
  - Data movement
  - Execution of operations
- Important the content of registers (PC, IR, SR, ACC, MAR, MDR, etc.) and their change

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Fetch-Execute cycle

1. Instruction fetch (IF)
   PC register refers to the memory address of the next instruction. Reading from here to IR.
   PC is updated to the address of next instruction
2. Instruction decoding (ID)
   Interpretation of the opcode. What kind of operation? What is the input data? Where to save the result? (Which registers are used?)
   Instruction Set Architecture (ISA) defines it.
   It can be hard-wired or microprogrammed.

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Fetch-Execute cycle

3. Execution or address calculation (EX)
   ALU works, result into internal temporal register, in case of Load/Store instruction calculation of the proper memory address.
4. Memory access (MEM)
   In case of Load/Store instruction reading/writing of given data memory address.
5. Writing back (WB)
   Result of operation or read data stored into destination register.

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

The x86 architecture

Structure of the processor

Register set

Memory handling

Assembly language

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Beginnings of x86 architecture

- Intel developed a „new” CISC processor between 1976-78 called **Intel 8086**
- Later it was improved
  - Intel 80186 (1982)
  - Intel 80286 (1982)
  - Intel 80386 (1986)
  - Intel 80486 (1989)
  - …, still go on
  - New processors are backward compatible
- The processor family is referred as **x86**
- Main manufacturers: Intel, AMD, VIA

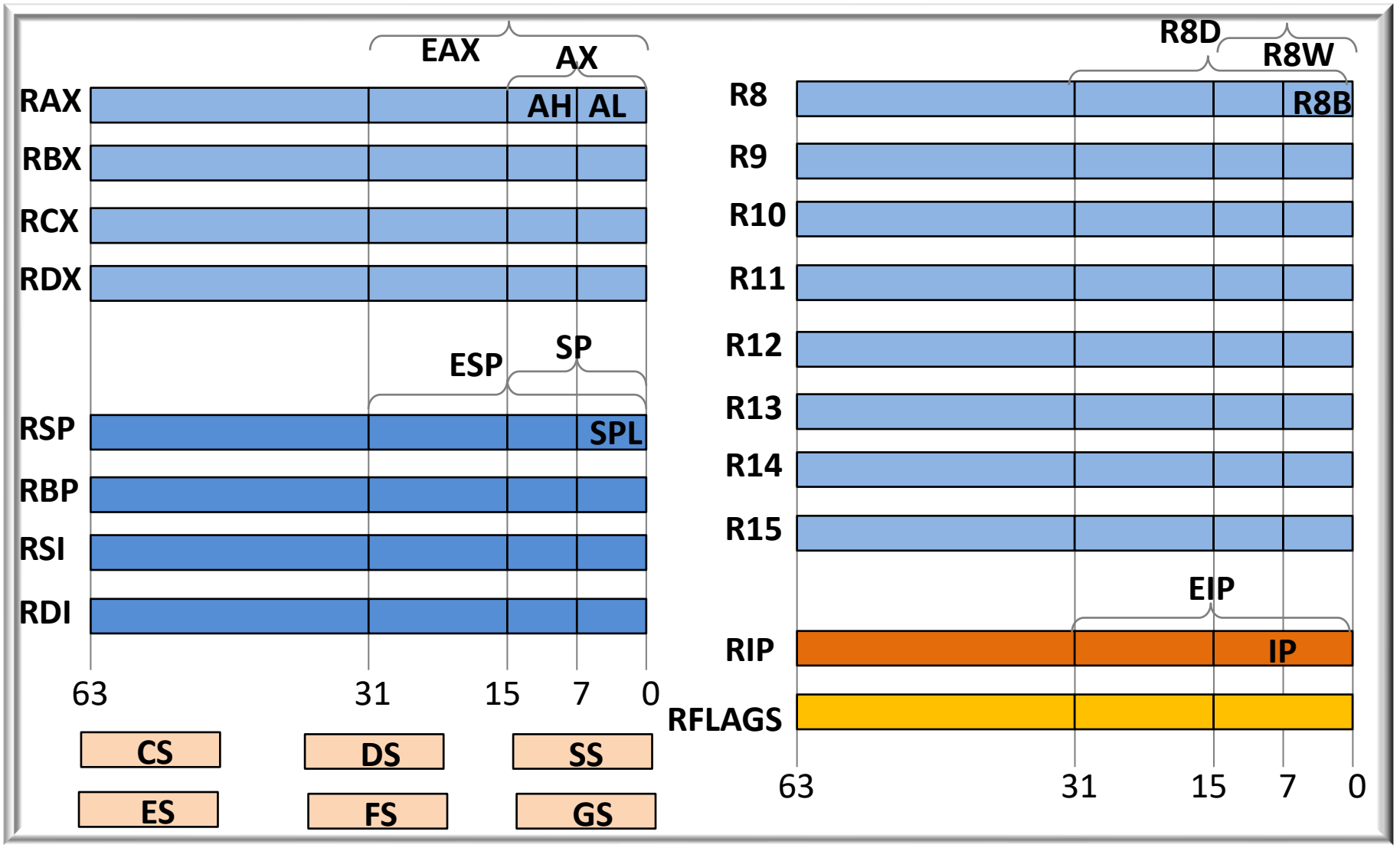
```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Memory segmentation

- Memory divided into logical parts
  - Code segment
  - Data segment
  - Stack segment
- Addressing is helped by registers (CS, DS, SS, ES)
- Addressing: segment starting address + offset
- Memory handling
  - Real-, Protected-, Virtual-, Long mode

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86-64 register set



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86 register set

Main registers (general purpose registers)

- EAX
  - Primary work register, multiplication, division, return value
- EBX
  - Work register, base pointer in DS
- ECX
  - Work register, (loop)counter, 4th parameter
- EDX
  - Work register, input/output, multiplication, division, 3rd parameter

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86 register set

Index (addressing) registers

- ESI (source index register)
  - Source index of string operations, working with DS, 2nd parameter
- EDI (destination index register)
  - Destination index of string operations, working with ES, 1st parameter
- ESP (stack pointer register)
  - Address of data on the top of stack, working with SS
- EBP (base/frame register)
  - Related to subroutines, working with SS

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86 register set

EFLAGS register

- Status bits
- Control bits
- System bits

31	30	29	28	27	26	25	24
0	0	0	0	0	0	0	0

23	22	21	20	19	18	17	16
0	0	ID	VIP	VIF	AC	VM	RF

15	14	13	12	11	10	9	8
0	NT	IOPL		OF	DF	IF	TF

7	6	5	4	3	2	1	0
SF	ZF	0	AF	0	PF	1	CF

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86 register set

Some examples of EFLAGS bits

- CF=1, if arithmetic/logic operation results in carry from the most significant bit position.
- OF=1, if operation leads to overflow.
- SF=1, if the most significant bit of result is 1 (can be interpreted as a negative value)
- ZF=1, if the current result is zero.
- PF=1, if the number of 1 bits in result is even.
- IF=1, if maskable interrupt is enabled.

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86 register set

Program counter

- EIP (Instruction pointer)
  - Refers to next instruction together with CS
  - During all „fetch-execute” cycle it is incremented by the length of instruction (except control passing)

Other registers

- Further registers helping CPU operation
- Hidden from programmer

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86-64 instructions, operands

Several hundred instructions

Instructions has 0, 1 or 2 operand(s)

- Register (8, 16, 32, 64 bits)
- Constant (8, 16, 32, 64 bits)
- Memory content
  - Memory address and size forcing

```
mov  al, BYTE  PTR [v]
mov  ax, WORD  PTR [v]
mov  eax,DWORD PTR [v]
mov  rax,QWORD PTR [v]
```

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86 addressing modes

Summary of effective address (EA) forms

$$\left[\left\{\begin{matrix}CS:\\SS:\\DS:\\ES:\\FS:\\GS:\\none\end{matrix}\right\}\left\{\begin{matrix}EAX\\EBX\\ECX\\EDX\\ESI\\EDI\\EBP\\ESP\\none\end{matrix}\right\}\left\{\begin{matrix}EAX\\EBX\\ECX\\EDX\\ESI\\EDI\\EBP\\none\end{matrix}\right\}\left\{\begin{matrix}1\\2\\4\\8\\none\end{matrix}\right\}\left\{\begin{matrix}+offset\\nothing\end{matrix}\right\}\right]$$

Segment selector *Base* *Index* *Scale factor* *Offset*

Example:

```
mov EAX, [DS:EBP+EDI*4+16]
```

The same instruction in different form:

```
mov EAX, DS:10h[EBP][EDI*4]
```


```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86 assembly syntax

Intel syntax

```
.intel_syntax noprefix
.globl main
main: push ebp
      mov  ebp, esp
      sub  esp, 16
      mov  DWORD PTR [ebp-16], 2
      mov  DWORD PTR [ebp-12], 3
      cmp  DWORD PTR [ebp-16], 4
      jne  .L2
      mov  eax, DWORD PTR [ebp-12]
      mov  DWORD PTR [ebp-8], eax
      jmp  .L3
.L2:  mov  DWORD PTR [ebp-8], 4
.L3:  mov  eax, DWORD PTR [ebp-8]
      add  esp, 16
      pop  ebp
      ret
```

AT&T syntax

```
.att_syntax noprefix
.globl main
main: pushl %ebp
      movl  %esp, %ebp
      subl  $16, %esp
      movl  $2, -16(%ebp)
      movl  $3, -12(%ebp)
      cmpl  $4, -16(%ebp)
      jne   .L2
      movl  -12(%ebp), %eax
      movl  %eax, -8(%ebp)
      jmp   .L3
.L2:  movl  $4, -8(%ebp)
.L3:  movl  -8(%ebp), %eax
      addl  $16, %esp
      popl  %ebp
      ret
```

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86 subroutine calling convention

Rules of caller

- Parameters in given order into registers: `edi`, `esi`, `edx`, `ecx`, …
  - Or pushing parameters in reverse order into stack
  - Floating point parameters in other registers (number of them in `eax` register)
- Invocation (return address into stack, update program counter to subroutine address)
- After return removing parameters from stack
- Return value in `eax` register

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

x86 subroutine calling convention

Rules of callee (subroutine)

- Saving base pointer (`EBP`) into stack
- Copying stack pointer (`ESP`) into `EBP`
- Allocation space in stack for local variables
- Necessary registers saved into stack

- Putting return value into `eax` register
- Recovering saved registers and stack
- Return (address is in stack)

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Intel Core i9

- Released in 2017
- 64-bit registers (x86-64)
- Microarchitecture: Skylake, Coffee Lake, Rocket Lake
- 14nm technology
- FCLGA socket (2066 contacts)
- 6-18 cores, 13-24MB L3 smart cache, AVX-512, HT, DMI 3.0, Virtualization Technology (VT-x, VT-d), DDR4, Turbo Boost Max 3.0 (5,2GHz), DL Boost, AES-NI

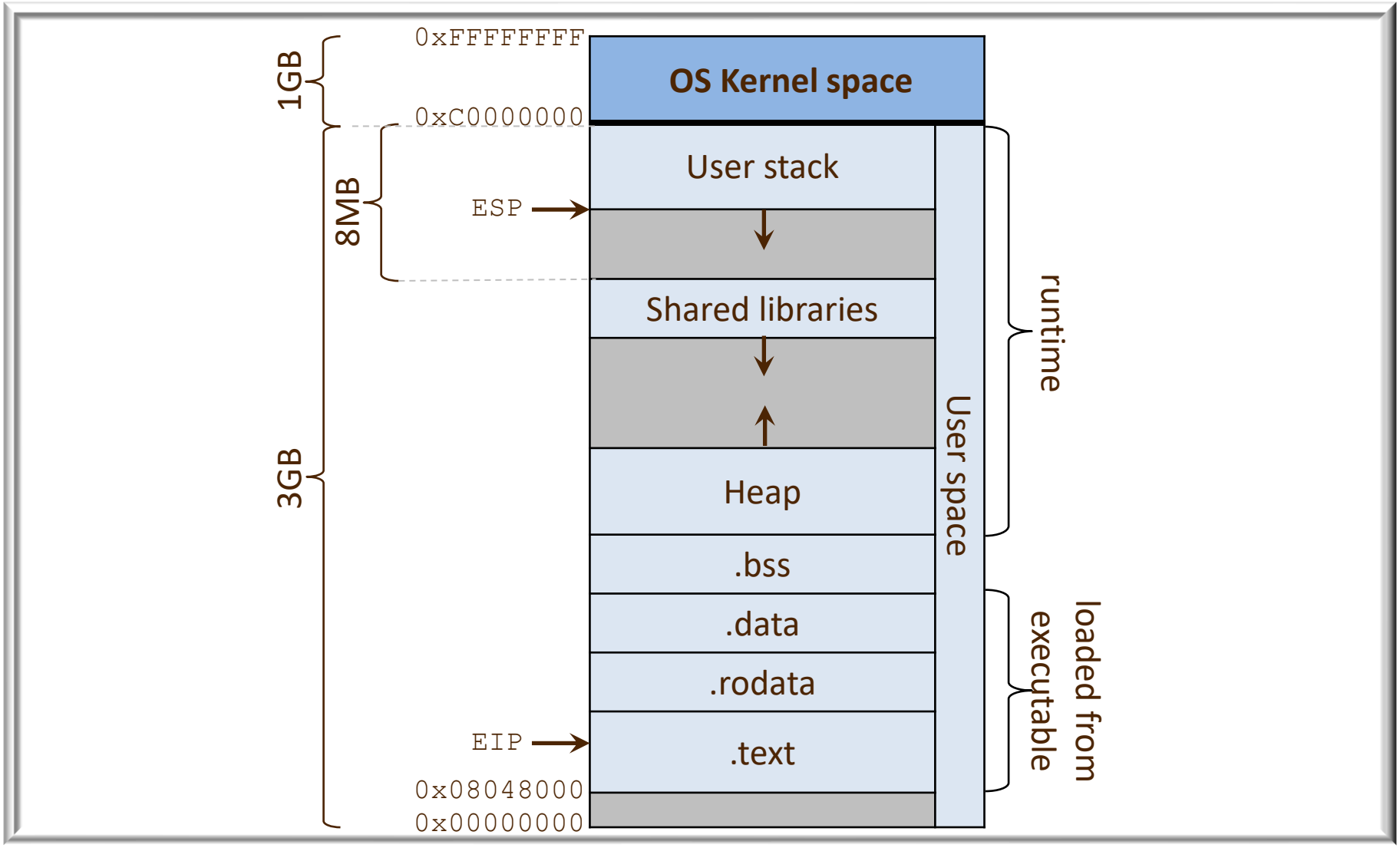
```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

AMD Ryzen 9

- Released in 2019
- 64-bit registers (x86-64)
- Microarchitecture: Zen2, Zen3
- 7-14nm technology
- Ca. 10 billion transistors
- AM4, FP6 socket (1331, 1140 contacts)
- 8-16 cores, 32-64MB L3 cache, AVX2, HT, PCIe 4.0, APU, DDR4, AMD-V virtualization, unlocked, Turbo Core (4,9GHz)

```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

Address space of a program in Linux



```
mov eax, DWORD PTR [rbp+16]
and eax, 0FFFFFD00h
mov rsp, rbp
pop rbp
ret
```

References

- Imre Varga:
  Computer architecture (lecture slides), 2019
- Nicholas Charter:
  Computer architecture, McGraw-Hill, 2001
- R. E. Bryant, D. R. O'Hallaron:
  Computer Systems – A programmer's perspective, Pearson, 2016
- Andrew s. Tanenbaum, Todd Austin:
  Structured Computer Organization, Pearson, 2013
- Joseph Cavanagh:
  X86 Assembly Language and C Fundamentals, CRC Press, 2013
- Richard Blum:
  Professional Assembly Language, Wiley, 2005