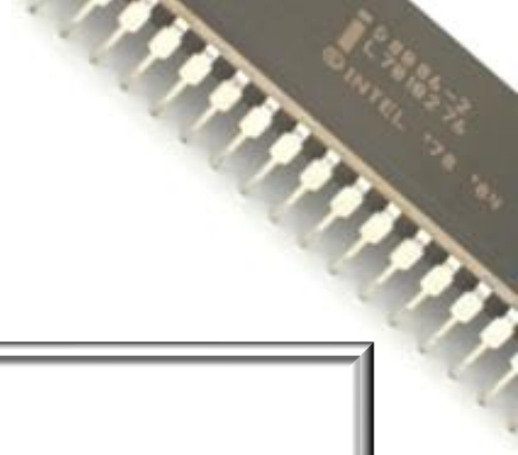


COMPUTER ARCHITECTURES

Imre Varga PhD

University of Debrecen,

Department of IT Systems and Networks

Version 2.0.0
12 February 2023

Warning

Lexical knowledge about the slides
are **necessary**, but **not enough** to pass.

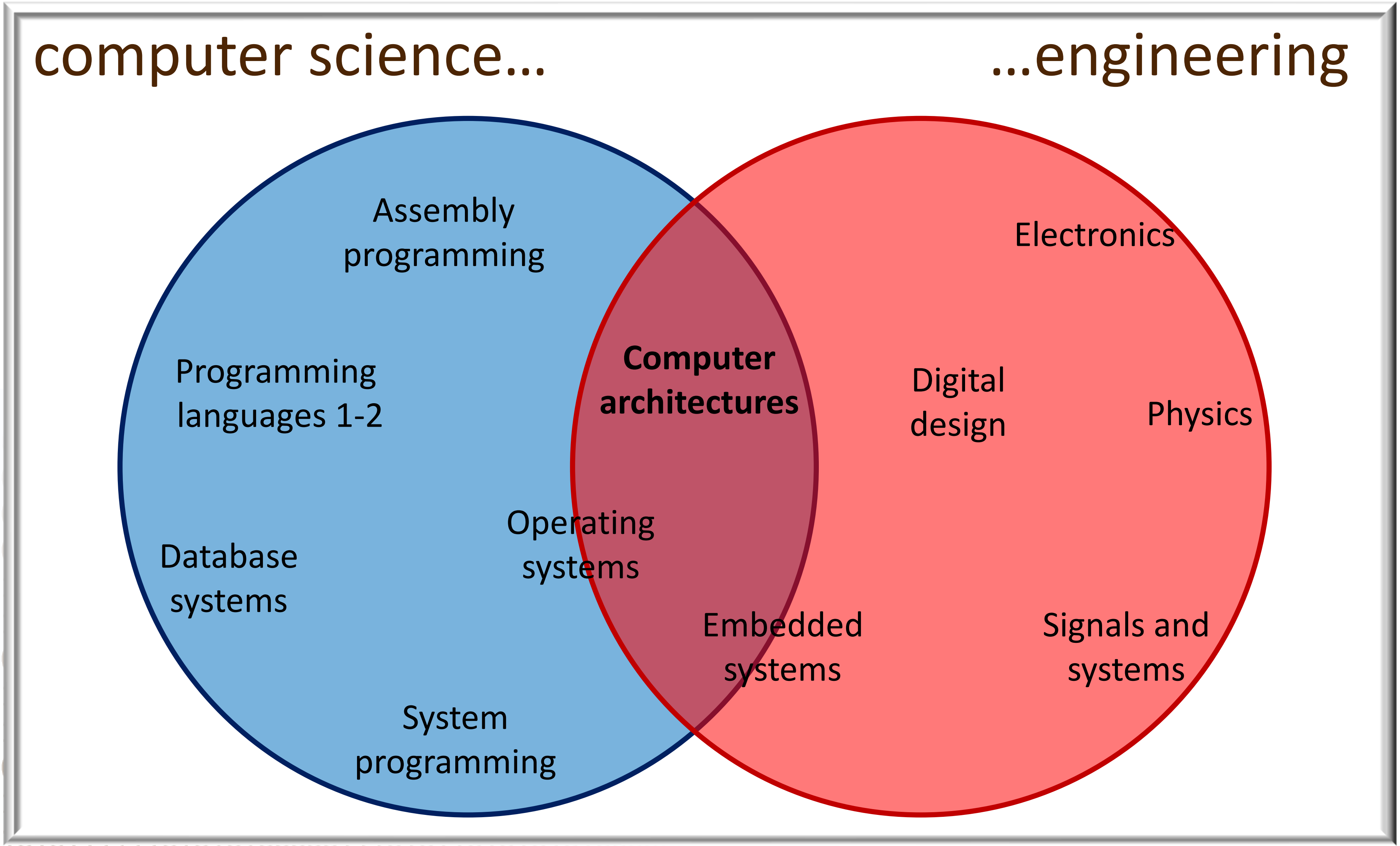
Understanding is necessary, that is why
active participating on lectures are advantages.

Other literature can be also useful.

⚡ Slide for staying in game.

Slide to avoide mediocrity. ☀

Computer Science Engineering



computer science…

…engineering

Assembly programming

Programming languages 1-2

Database systems

System programming

Operating systems

Computer architectures

Embedded systems

Digital design

Electronics

Physics

Signals and systems

Topics

- ✦ Number representation, datatype implementation
- ✦ Essential structure and work of CPUs
- ✦ Modern processors
- ✦ Concrete processor architecture
  - ➥ Instruction-set, programming
- ✦ Aspects of programming closely related to the hardware
- ✦ Basics of digital technology from the point of view of the hardware
- ✦ Assembly programming
  - ➥ Mapping high-level programming to low-level

The goal of the subject

- Giving **knowledge about hardware**
- Creating connection between …
  - ‚programming’ and ‚electronics’
  - ‚abstract’ and ‚fundamental’ knowledge
- Deeper understanding of high-level programming
  - More effective programming
- Introduction to programming
- **Approach shaping**
  - Overall insight into computer systems

Number representation, datatypes

Fixed-point representation

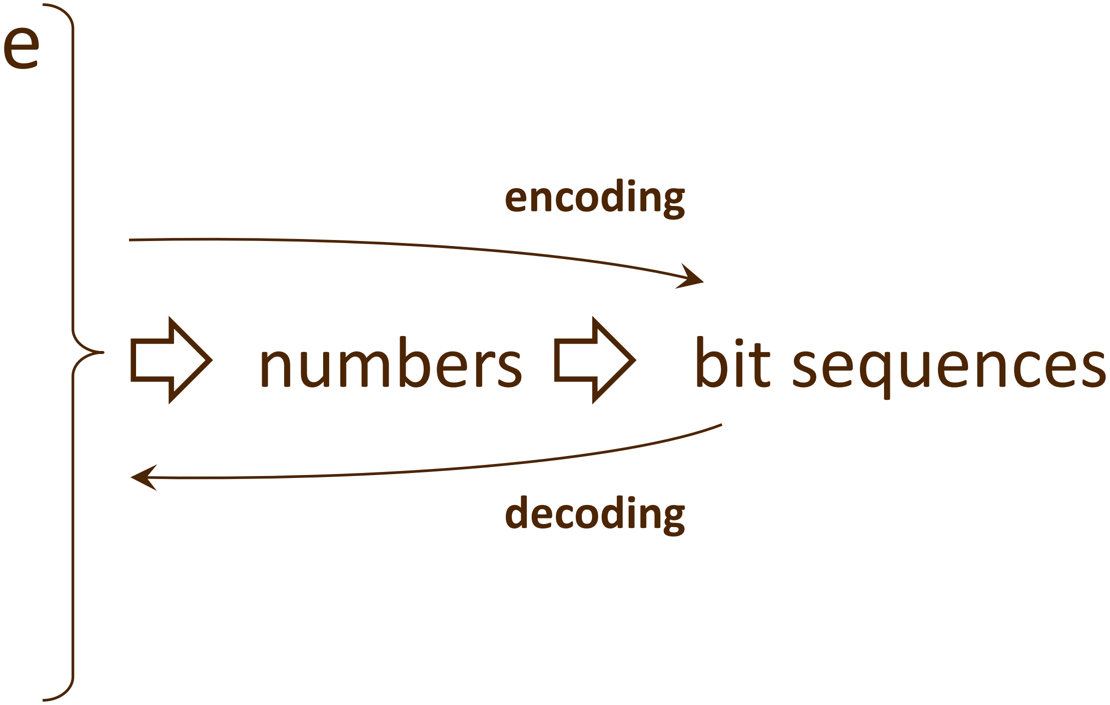
Floating-point representation

Main datatypes

Data

- Information is stored as different type datas
- Computer store everything as bit sequences

numerical value
text
image
voice/sound
video
database
program



Thought-provoking questions

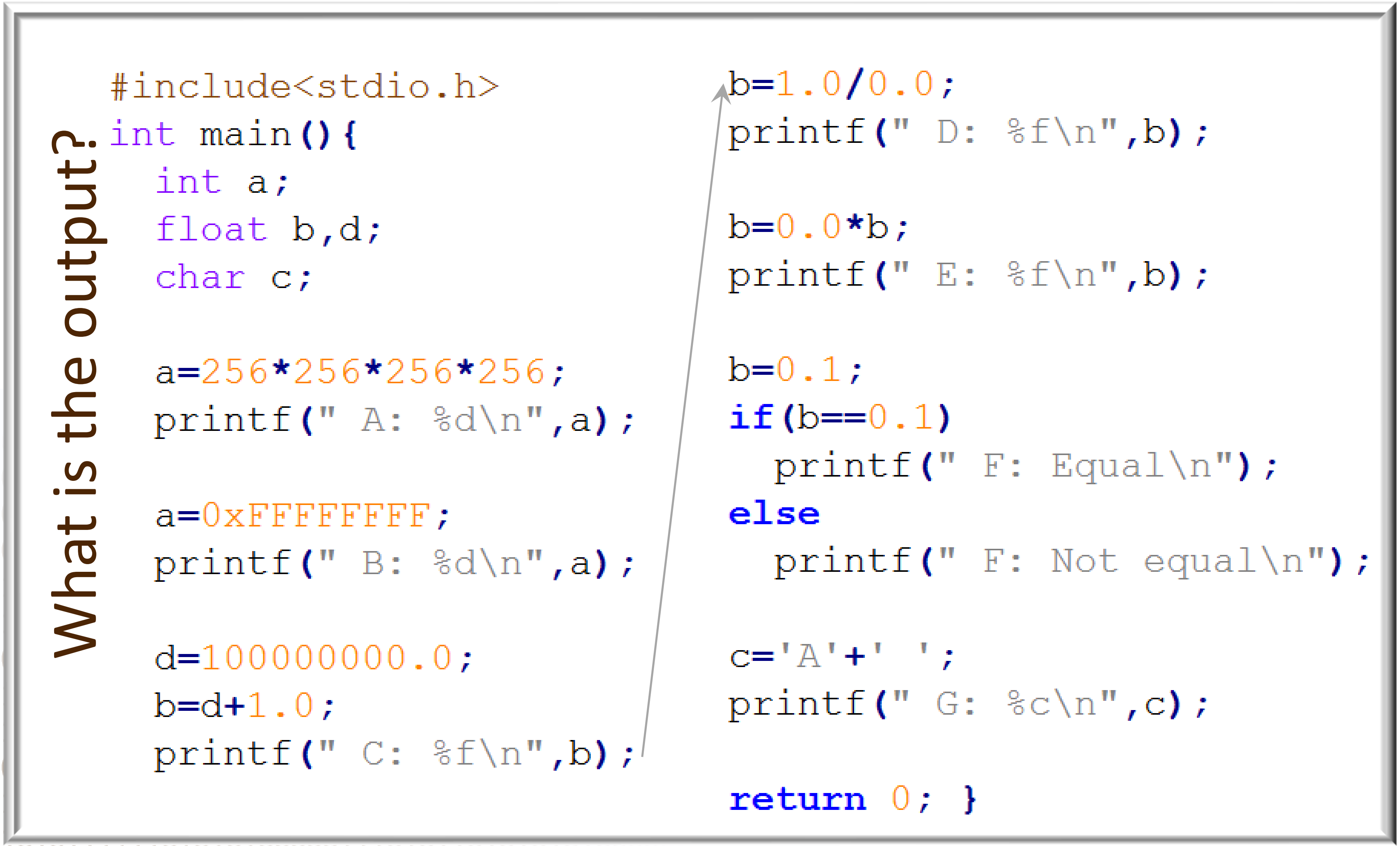
- How to encode the following values, i.e. how can we write them by bits?

0 18 -6 98374 3,25 0,0021 31/7 π

- How to decode, i.e. what is the meaning of the `10011010` bitsequence?
- What about the length of the bit sequence? Fix of flexible? What size?
- How to perform operations with these bitsequences?

Thought-provoking questions

What is the output?



```
#include<stdio.h>
int main(){
  int a;
  float b,d;
  char c;

  a=256*256*256*256;
  printf(" A: %d\n",a);

  a=0xFFFFFFFF;
  printf(" B: %d\n",a);

  d=100000000.0;
  b=d+1.0;
  printf(" C: %f\n",b);
b=1.0/0.0;
printf(" D: %f\n",b);

b=0.0*b;
printf(" E: %f\n",b);

b=0.1;
if(b==0.1)
  printf(" F: Equal\n");
else
  printf(" F: Not equal\n");

c='A'+' ';
printf(" G: %c\n",c);

return 0; }
```

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**

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

Arithmetic operations in binary

addition

```
  10011001
 +01011010
 ---------
  11110011
```

subtraction

```
  10110011
 -01101010
 ---------
  01001001
```

multiplication

```
101110*101
------
101110
  000000
    101110
----------
11100110
```

division

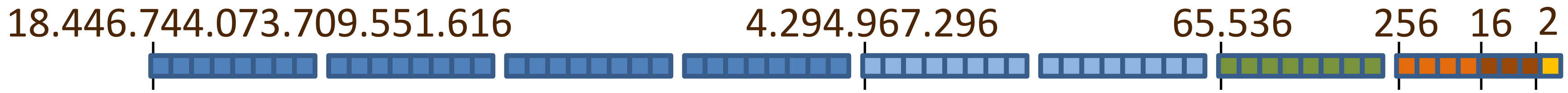
```
101110:100=1011
0011
  0111
    0110
      010
```

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



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

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

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)

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

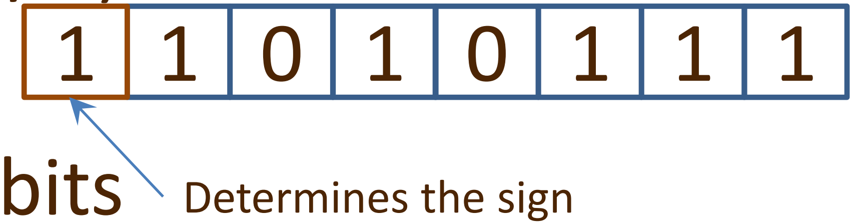
0	0	1	0	1	0	0	1

- Representation on N bits
  - Lowest value: 0
  - Highest value: 2^N-1

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)

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



1	1	0	1	0	1	1	1

Determines the sign

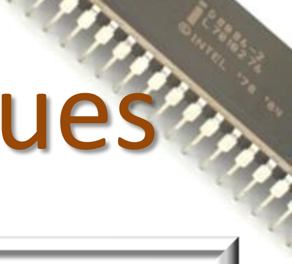
- Representation on N bits
  - Lowest value : $-(2^{N-1})$
  - Highest value: $2^{N-1}-1$

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

Calculation with fixed-point values



Same algorithm can be used…

- for addition of positive and negative values
- for addition and subtraction

Example (in case of 1 byte long data)

67+54=121

```
 01000011
+00110110
 01111001
```

67-54=13

67+(-54)=13

```
 01000011
+11001010
 00001101
```

Absolut value (+54):

00110110

One’s complement:

11001001

Two’s complement (-54):

11001010

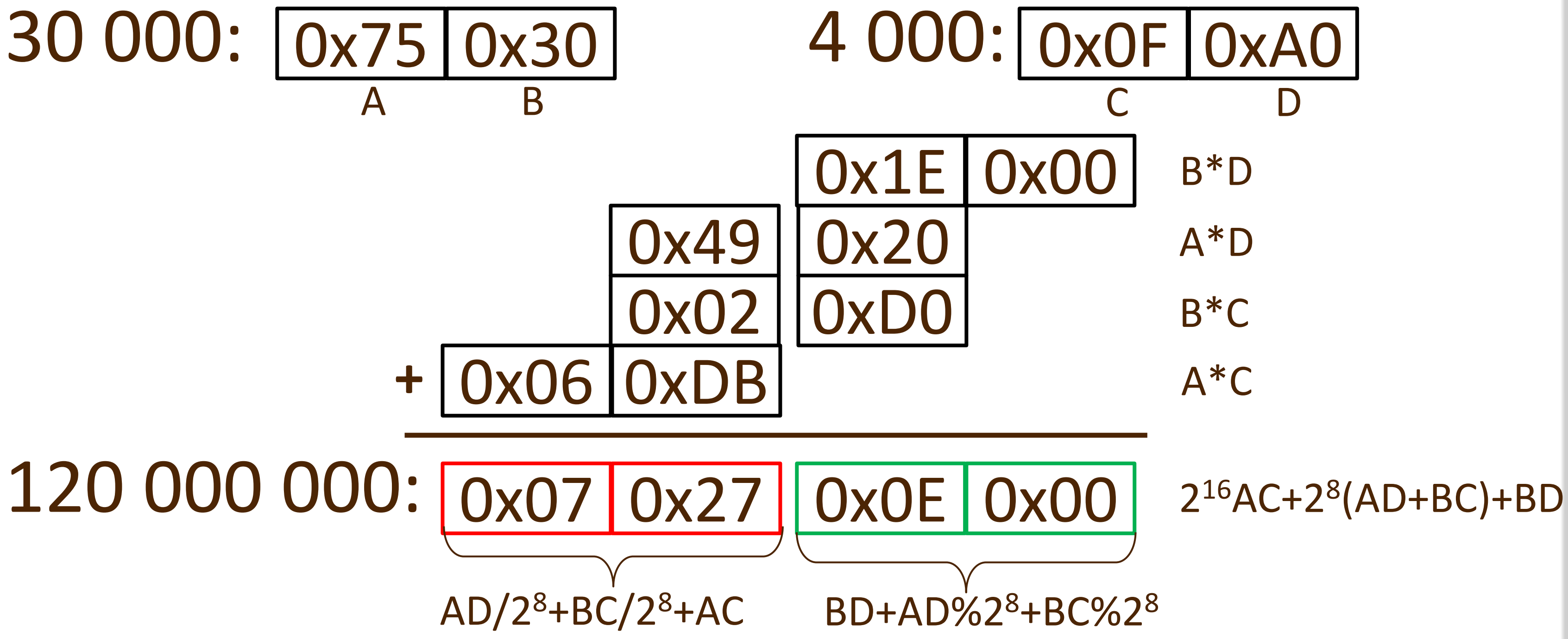
„Sign-and-magnitude” (-54)

~~10110110~~

Calculation with fixed-point values

Multiplication: product of two N-bit values is 2N-bit

Example: 16-bits operands



30 000: 0x75 | 0x30 (A B)

4 000: 0x0F | 0xA0 (C D)

0x1E 0x00 B*D

0x49 0x20 A*D

0x02 0xD0 B*C

+ 0x06 0xDB A*C

120 000 000: 0x07 0x27 0x0E 0x00 $2^{16}AC+2^{8}(AD+BC)+BD$

$AD/2^{8}+BC/2^{8}+AC$

$BD+AD\%2^{8}+BC\%2^{8}$

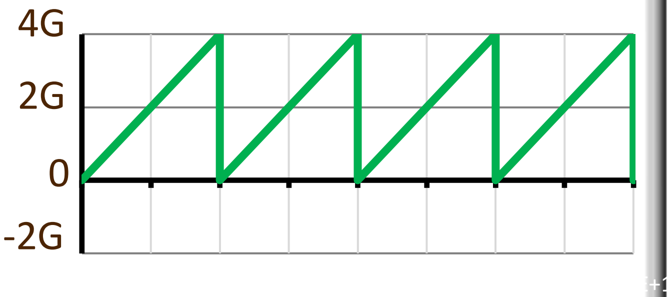
Result in two registers or overflow (error)

Arithmetic problems

Two special situations

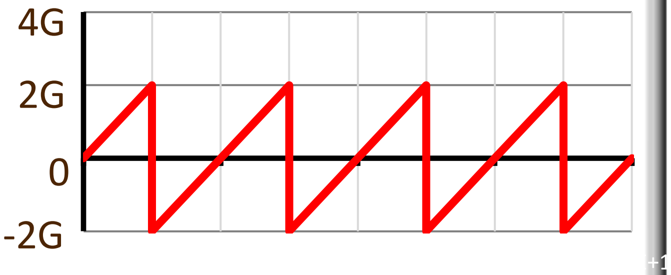
- Unsigned: more bit needed (**carry**)

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



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

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



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;`

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:

25 32	17 24	9 16	1 8
0x4C	**0x3D**	**0x2E**	**0x1F**

Big-endian:

0x1F	**0x2E**	**0x3D**	**0x4C**
1 8	9 16	17 24	25 32

BCD representation

Binary-Coded Decimal

- ✦ All digits represented on 4 bits
- ✦ Unused bit combinations

Unpacked BCD

- ✦ 1 digit - 1 byte: starting with 0000 or (0011*)

  1395 → 00000001 | 00000011 | 00001001 | 00000101

Packed BCD

- ✦ 2 digits - 1 byte: nibble-based

  1395 → 0001|0011 | 1001|0101

*due to ASCII

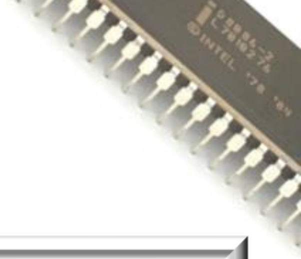
Character type

- Values are character codes
  - Mapping: symbols → integers (codetable)
  - Not the shape of the character is stored, but a number as fixed-point value
    (in C language 1 byte, in Java 2 bytes)
- Character code tables
  - ASCII: 7 bits
  - ISO-8859-1 „Latin 1”: 8 bits
  - Unicode: 16 bits

Character code tables

- ASCII codes
  - 0x00-0x1F: control characters (Tab, ESC, LF, CR, …)
  - 0x30-0x39: decimal digits
  - 0x41-0x5A: English upper-case letters
  - 0x61-0x7A: English lower-case letters
- ISO-8859-1 „Latin 1”
  - 0x00-0x7F: ASCII (compatibility)
  - 0x80-0xFF: western European special characters (Ä, Æ, Ñ, Ë, ß, Ö)

Character code tables



- Unicode
  - ASCII compatibility
  - 16 bits code planes (17 pcs)
  - 1st plane: Basic Multilingual Plane (BMP)
  - More than 1 million code points (17*65536)
  - More than 140 thousand defined symbols (Latin, Cyrillic, Arabic, Hebrew, kanji, hiragana, Egyptian hieroglyphs, Sumero-Akkadian cuneiform, mathematical symbols, other characters)

ß Д ש ش. 漢 あ 𓀩 ∲ ♋ ☑ ⇨

Unicode encoding techniques

- ✦ Rare unicode codevalues can be represented even in 4 bytes
- ✦ Code of most often used characters are short
  - ➥ Lot of leading 0s are needed, wasting space
- ✦ Solution: variable length encoding
  - ➥ The length of character codes can be different
- ✦ Encoding techniques
  - ➥ **UTF-8**
  - ➥ UCS-2
  - ➥ UTF-16
  - ➥ UTF-32

UTF-8

- ASCII: All bytes are ASCII codes

```
unicode 00000000 00000000 00000000 0abcdefg
                                    usful bits
UTF-8   0abcdefg
```

- Non-ASCII: None of bytes are ASCII codes

```
unicode 00000000 00000000 00000abc defghijk
UTF-8   110abcde 10fghijk

unicode 00000000 00000000 abcdefgh ijklmnop
UTF-8   1110abcd 10efghij 10klmnop

unicode 00000000 000abcde fghijklm nopqrstu
UTF-8   11110abc 10defghi 10jklmno 10pqrstu
```

UTF-8 examples

Character: 京 (simplified Chinese, pinyin: jing)

Unicode: 4EAC (decimal 20140)

Binary: 01001110 10101100

UTF-8: 11100100 10111010 10101100

HTML: &#x4EAC, &#20140

URL: %E4%BA%AC

Latin-1: ä°¬

Latin-2: äşŹ

Other decodings/interpretations

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

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

$$(-1)^S \cdot 1.F \cdot A^{K-N}$$

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*

Floating point representation

Example: -13.8125 single precision

$$-13.8125_{10} = -1101.1101_2 = -1^1 \cdot 1.1011101 \cdot 2^3$$

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

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

Sign: S = **1**

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

Mantissa: M = 1.1011101

Fraction field: F = **10111010000000000000000…**

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

1	1	0	0	0	0	0	1	0	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S					E															F											

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

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)

Real types of the C language

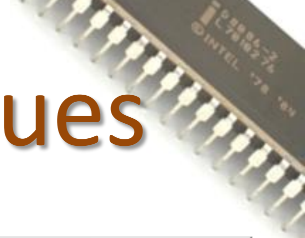
- Floating point representation

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

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’

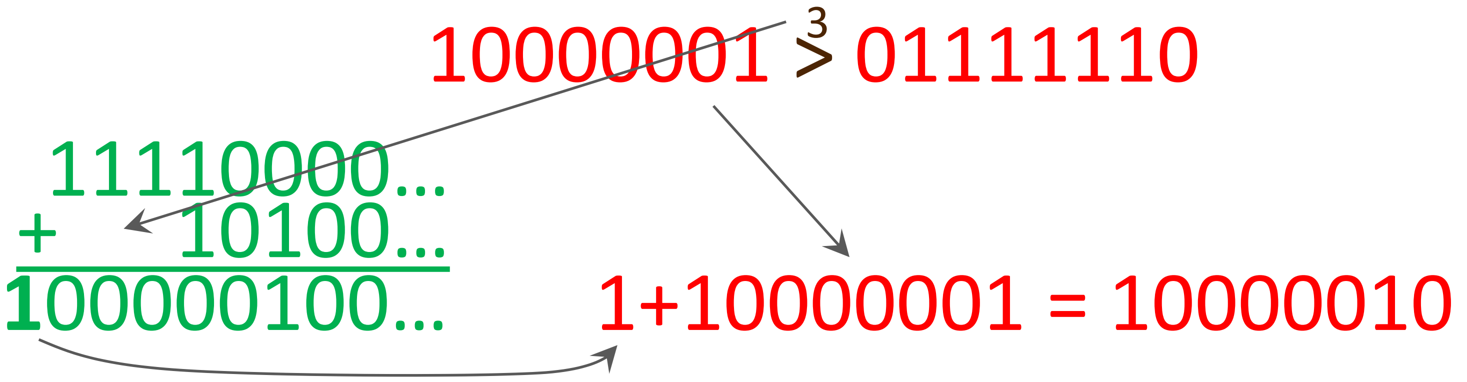


Calculation with floating point values

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

7.5 = 01000000 11110000 00000000 00000000

0.625 = 00111111 00100000 00000000 00000000



$10000001 \overset{3}{>} 01111110$

```
  11110000...
+    10100...
100000100...
```

1+10000001 = 10000010

01000001 000000100 00000000 00000000

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

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: -2669392824924167300000000.0
- ”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

Pointer, Boolean and record types

Pointer

- Indirect reference
- Addresses in the domain
  - Practically unsigned integers
- Pointer variables also have addresses

Logical/Boolean

- 1-byteinteger

Record

- Different type areas together
- All field with their own representation

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 ith 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

Two dimensional array

- Storing in row order

$$M_{3,2} = \begin{pmatrix} e_{1,1} & e_{1,2} \\ e_{2,1} & e_{2,2} \\ e_{3,1} & e_{3,2} \end{pmatrix} \rightarrow M_6 = \left(e_{1,1}\ e_{1,2}\ e_{2,1}\ e_{2,2}\ e_{3,1}\ e_{3,2}\right)$$

- Address of j^{th} element of i^{th} row in NxM matrix

$$A_{i,j} \;=\; A_{1,1} + M(i-1)E + (\mathrm{j}-1)\mathrm{E}$$

where E is the size of an element and $A_{1,1}$ is the starting address of the matrix

String type

- Series of characters
- Elements in inner representation
  - C: 1 byte, ASCII, fixed-point
  - Java: 2 byte, Unicode
- Variable length
  - Length at the beginning of the string (E.g., Pascal)
    Disadvantage: length is limited

6	S	t	r	i	n	g	i	s	a

  - Special closing character at the end (E.g., C)
    Disadvantage: closing character cannot be in string

S	t	r	i	n	g	**\0**	i	s	a

Assembly programming

Assembly language

Instruction set

Addressing modes

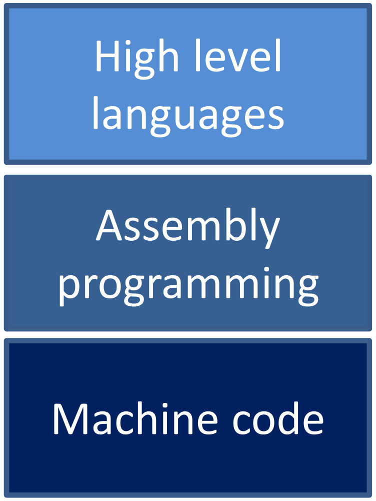
Machine code

Thought-provoking questions

- High-level programming languages (C, Java) are not directly understandable by computer. How to tell for the computer what to do?
- What kind of basic instructions are needed?
- Can we tell the processor what to do, if we don’t know its construction?
- How does the processor know where the desired data is in memory?
- How can we give the instructions as bit sequences?

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



High level languages

Assembly programming

Machine code

Instruction set

- 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

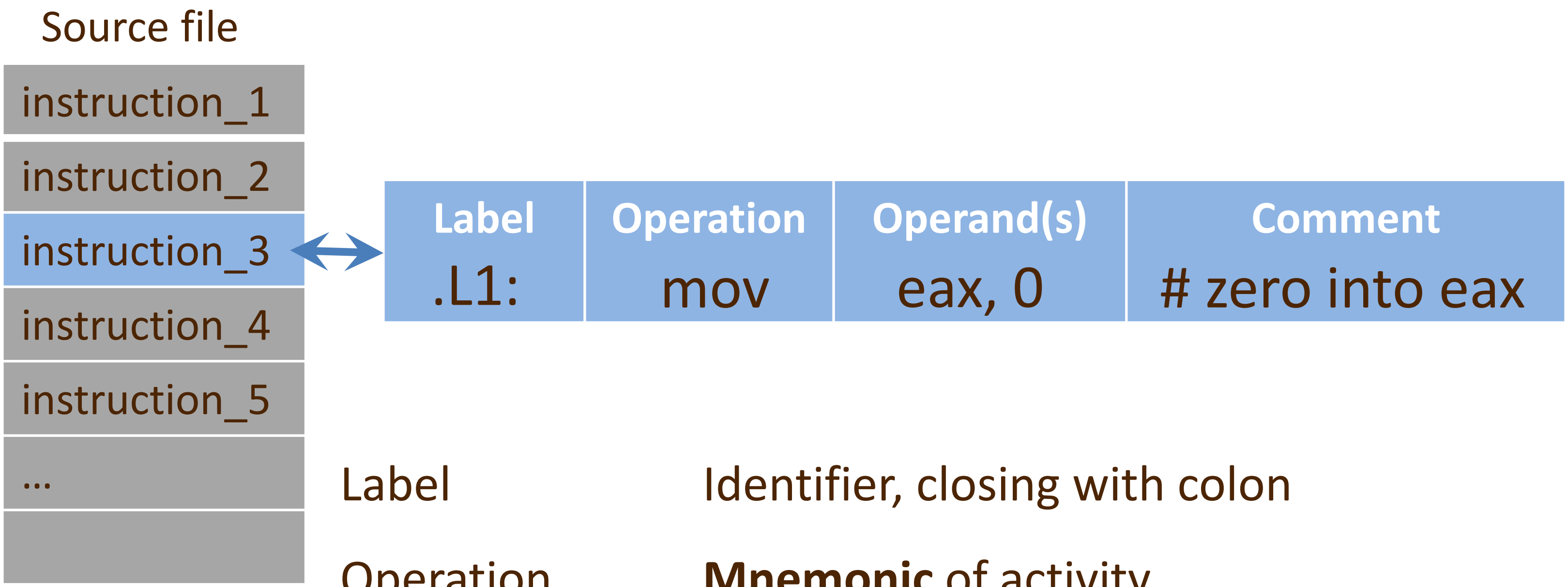
- 1-address instruction

Operation code	2nd operand address

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

Operation code

Structure of instructions



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

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.

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`

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

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)

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)

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)

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)

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)

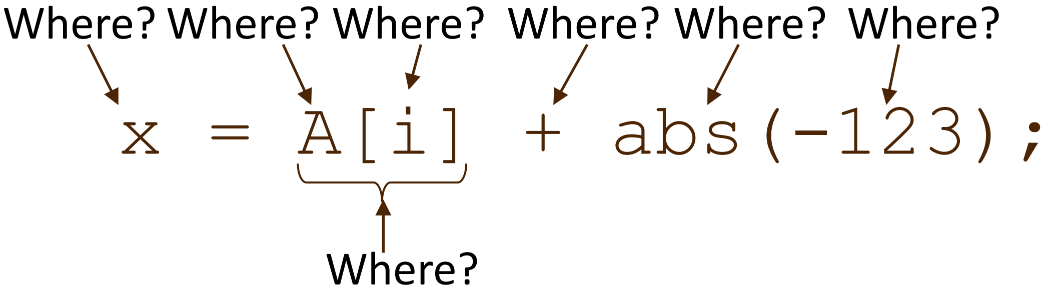
Types of instructions

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)
- …

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



```
x = A[i] + abs(-123);
```

Addressing modes

How can we access data in memory?

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

Addressing modes

Implicit/implied addressing

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

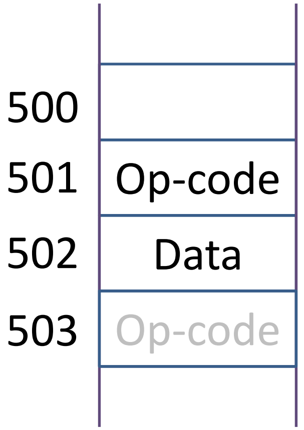
RAM:

500	
501	Op-code1
502	Op-code2
503	

Addressing modes

Immediate data

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

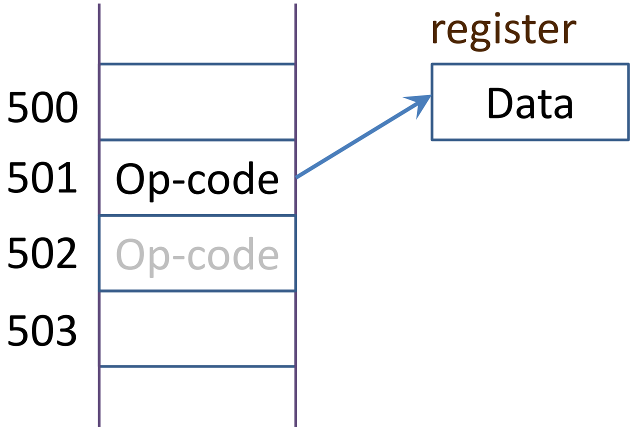


500	
501	Op-code
502	Data
503	Op-code

Addressing modes

Register direct

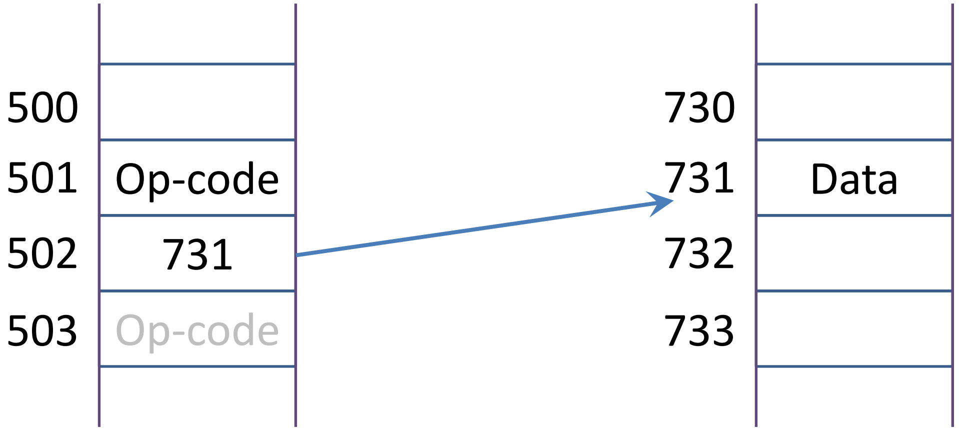
- Operation code contains reference to a register
- Operand in this register



Addressing modes

Direct/absolute addressing

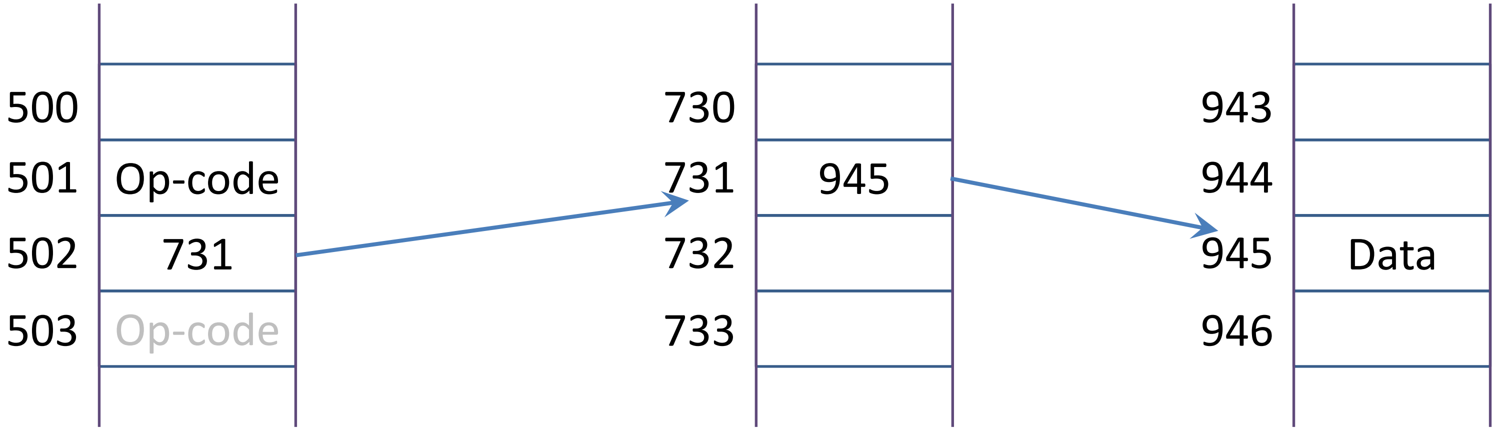
- An address behind the operation code
- Operand is located in this address



Addressing modes

Indirect addressing

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



500	
501	Op-code
502	731
503	Op-code

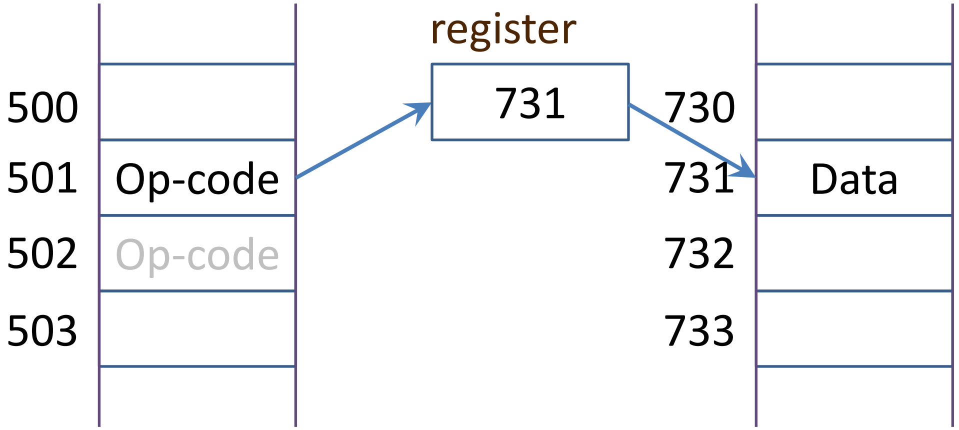
730	
731	945
732	
733	

943	
944	
945	Data
946	

Addressing modes

Register indirect addressing

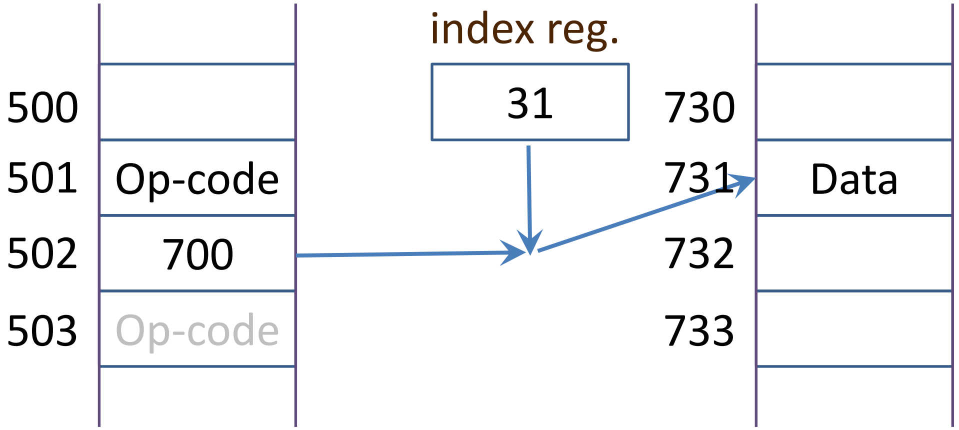
- Operation code refers to a register
- The address of operand is in the register



Addressing modes

Indexed addressing

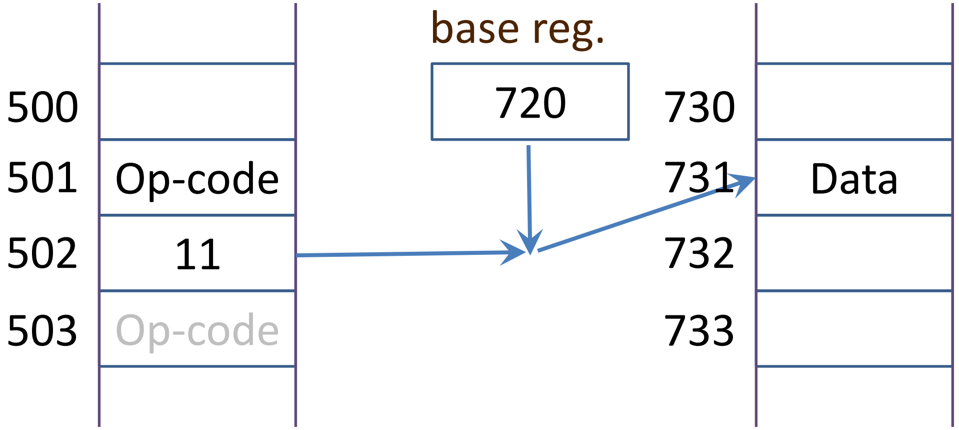
- An address is behind the operation code
- Add the content of index register to the address
- The sum is the address of operand



Addressing modes

Register relative addressing

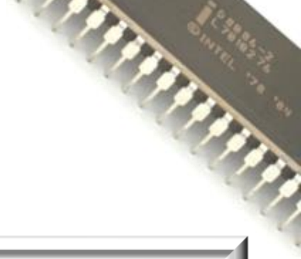
- An offset value is behind the operation code
- Add the content of base register to offset value
- The sum is the address of operand



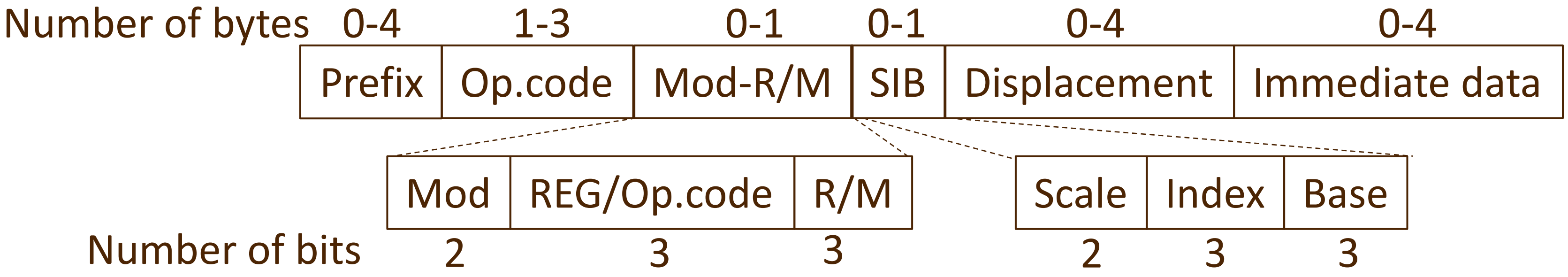
Assembly code example

```
      …
# eax=-1
      mov   eax, 0xFFFFFFFF
# edx=edx+3
      add   edx, 3
# edx=edx*4
      shl   edx, 2
# eax==edx or eax>edx or eax<edx ?
      cmp   eax, edx
# if eax==edx goto L (so skip next instruction)
      jz    .L0
# eax=eax-1
      dec   eax
#  (*ecx)=eax  (ecx: address of a varaible)
.L0:  mov   dword ptr[ecx], eax
      …
```

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:



Number of bytes 0-4 1-3 0-1 0-1 0-4 0-4

Prefix | Op.code | Mod-R/M | SIB | Displacement | Immediate data

Mod | REG/Op.code | R/M

Scale | Index | Base

Number of bits 2 3 3 2 3 3

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

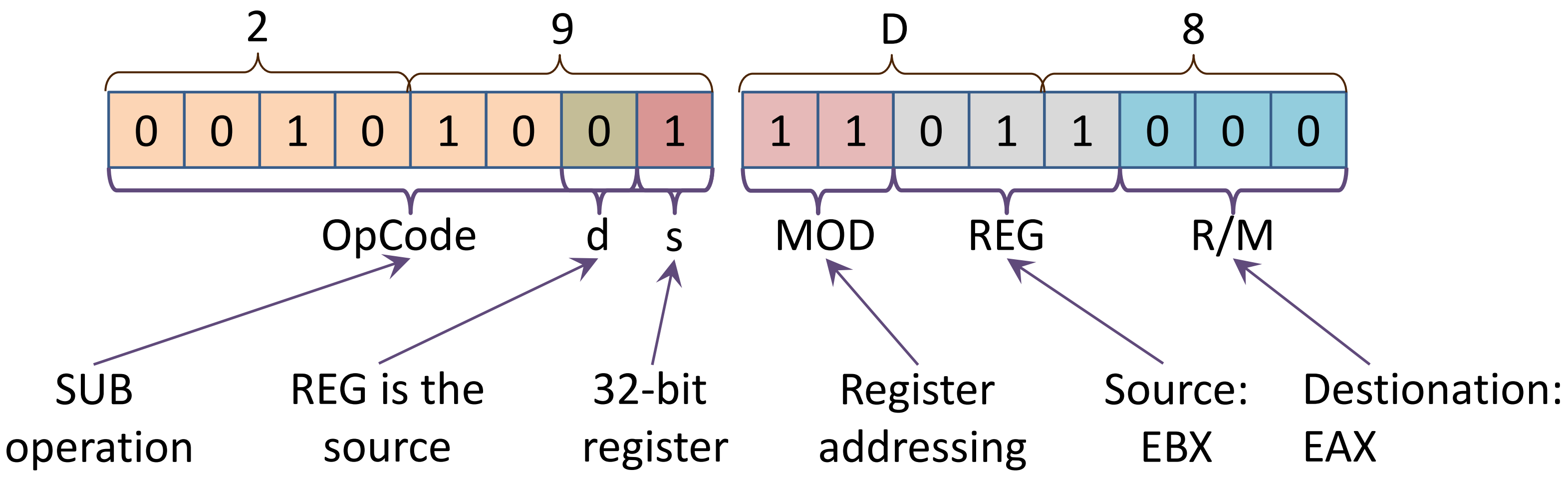
```
c7 45 f4 05 00 00 00 | c7 45 f8 06 00 00 00
8b 5d f4 | 89 d8 | c1 e0 03 | 29 d8 | 03 45 f8
89 45 f0
```

Machine code example

x86 architektúra:

assembly: sub eax, ebx # R1 = R1-R2

machine code: 0x29 0xD8



2 9 D 8

0 0 1 0 1 0 0 1 1 1 0 1 1 0 0 0

OpCode d s MOD REG R/M

SUB operation

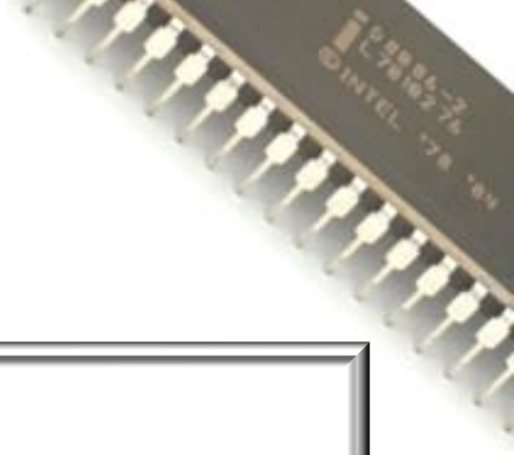
REG is the source

32-bit register

Register addressing

Source: EBX

Destionation: EAX



Basics of digital design

Semiconductor

Diode, transistor

Logical gate

Multiplexer, decoder, adder

Latch, flip-flop, register

Thought-provoking questions

- The electric insulators and conductors common, they are familiar, but what is a semiconductor? Why it is important?
- What is the relationship between tansistor and computer?
- Why must a computer science engineer study digital design?
- How can a circuit remember, perform calculation, make decision or control another circuit?

Physics

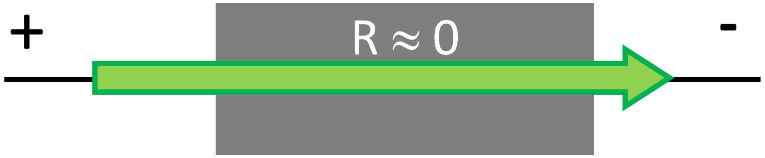
- matter → atom → electron^{-} & proton^{+} & neutron^{0}
- Coulomb force (fundamental interaction)

$$\vec{F} = \frac{1}{4\pi\epsilon_0}\frac{Q_1 Q_2}{r^2}\frac{\vec{r}}{|\vec{r}|}$$

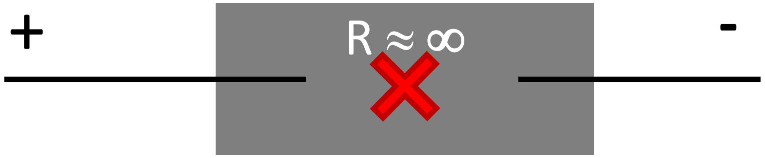
- Voltage: derives from force acting to charges
- Electric current: flow of charges
- Resistance: flow of moving charges is impeded
  - ↳ Insulators, conductors, semiconductors

Cinductors and insulators

- Conductors: no resistance, free charges can move according to the electric field, electric current flows



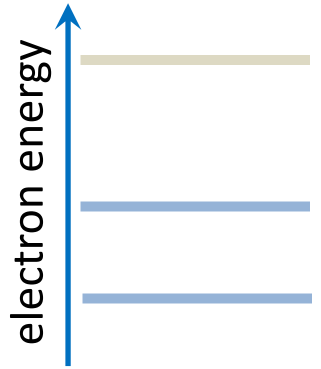
- Insulators: huge resistance, motion of charges is restricted, no electric current



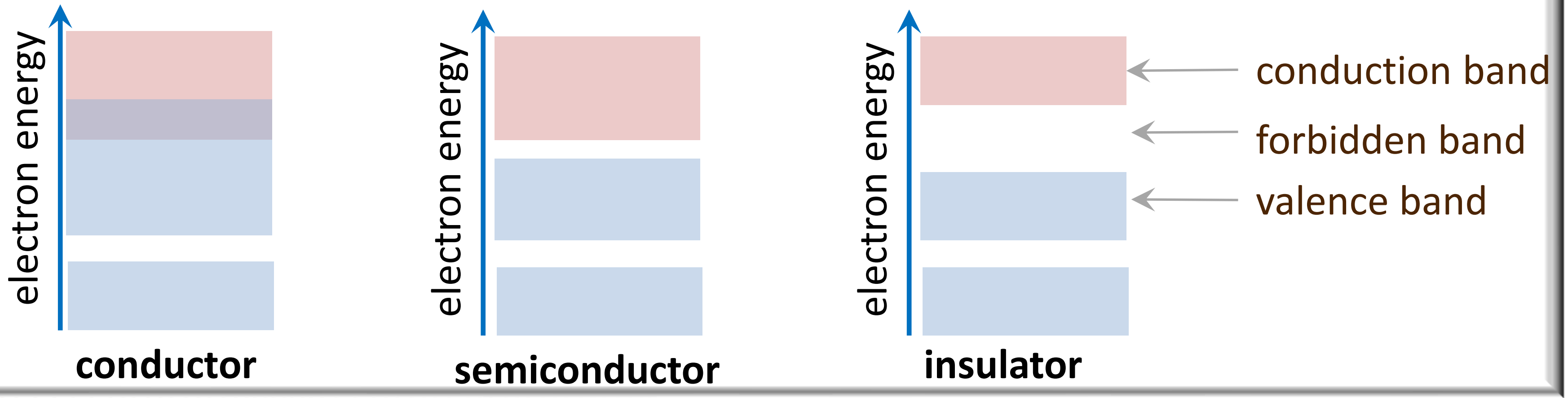
Semiconductor

- „Bad conductor, bad insulator"
- Reason: electronic band structure
  - 4 valence electron (Si, Ge)
  - Quantum physics
  - Bohr atom model

in an atom

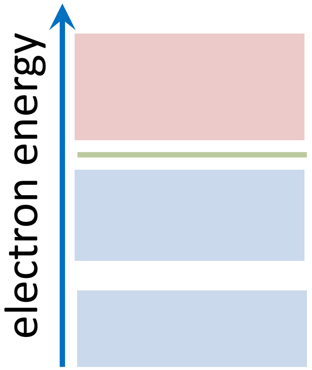


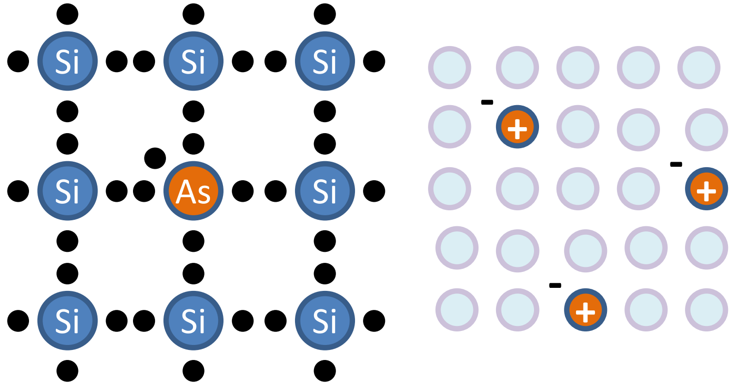
in solid states



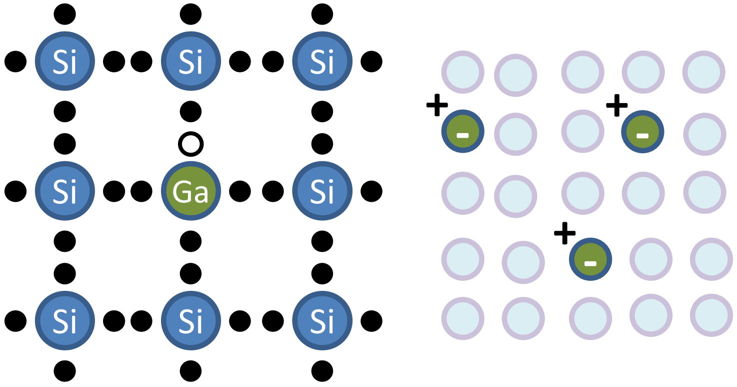
Extrinsic semiconductors

- n-type (electron)
  - Doping 5 valence electrons (As, Sb)
- p-type ('hole')
  - Doping 3 valence electrons (Ga, In)
- New energy level in forbidden band





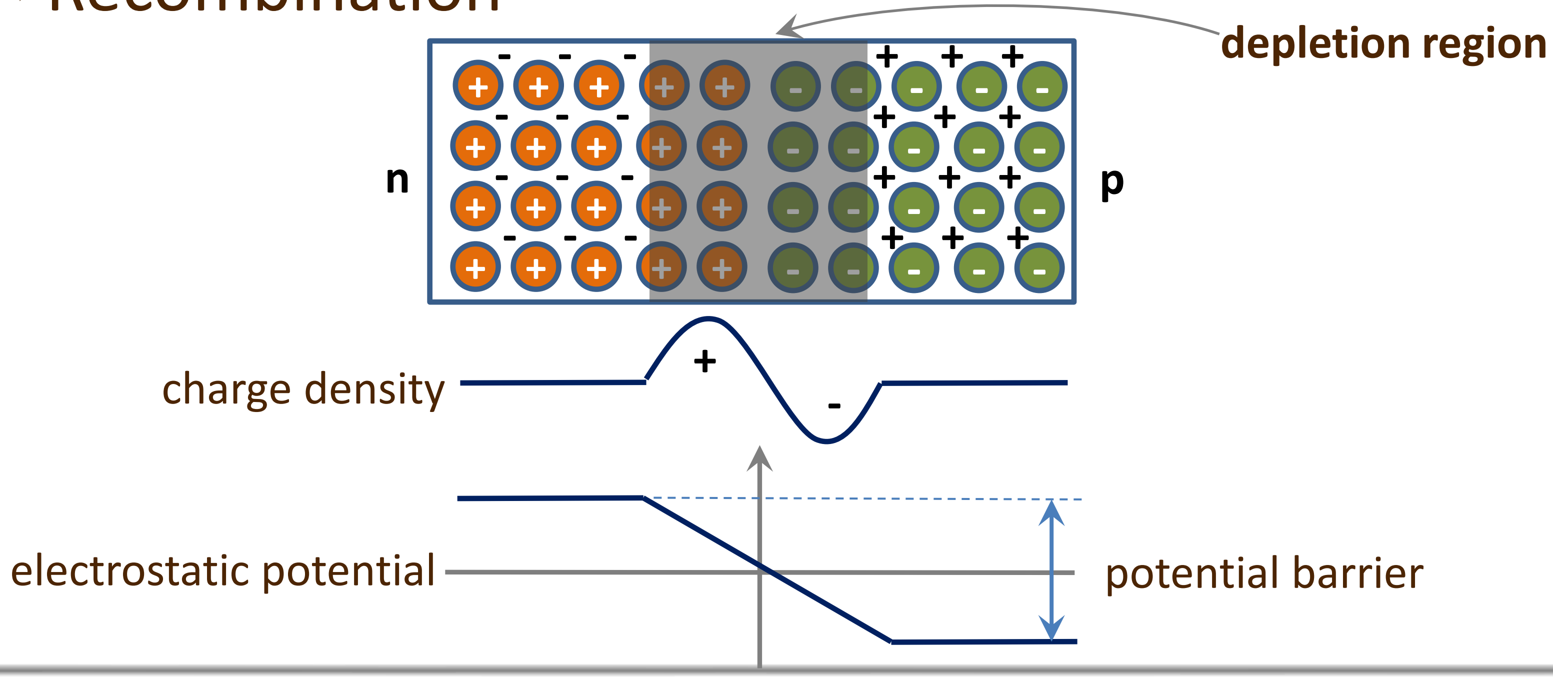
n-type



p-type

p-n junction

- Between differently doped semiconductor crystal region
- Free charge carrier move to opposite layers
  - ➥ Recombination



depletion region

n

p

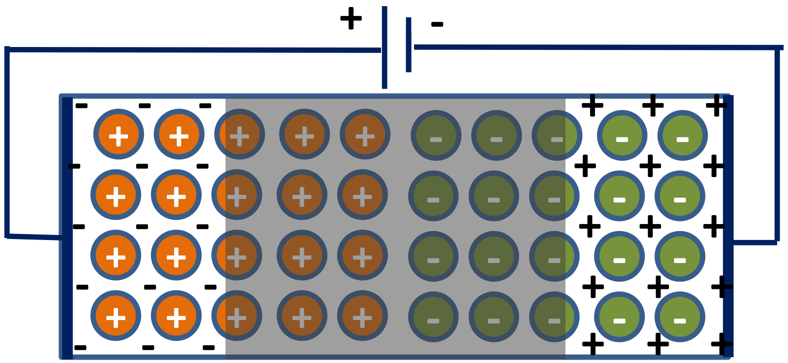
charge density

electrostatic potential

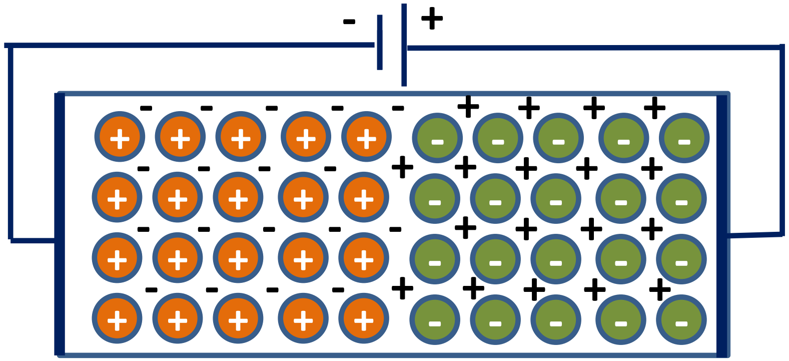
potential barrier

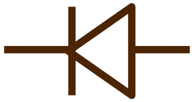
Diode

Reverse bias



Forward bias

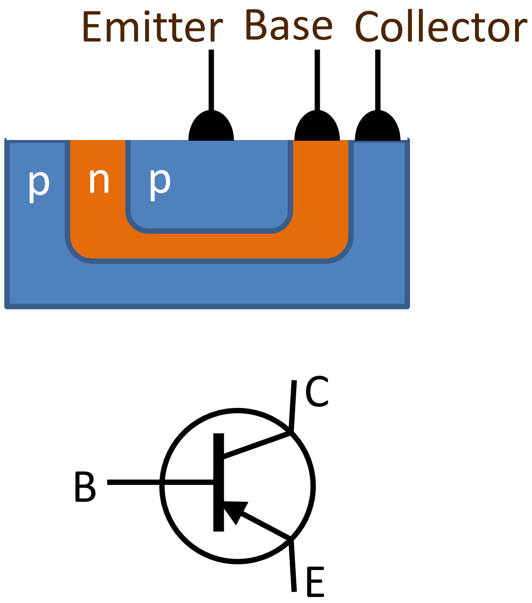


- ✦Diode: can be used as rectifier
- ✦Sign: 
- ✦Special case: LED (Light-Emitting Diode)

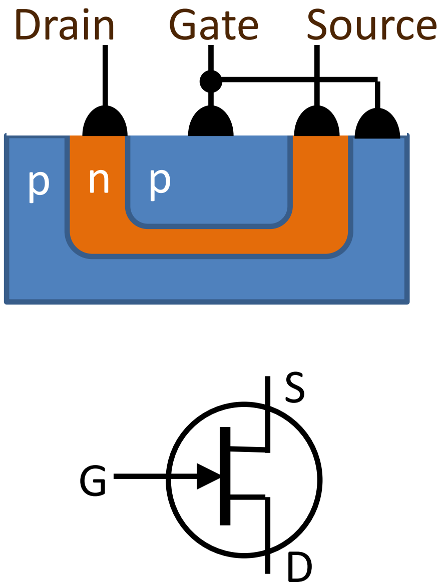
Transistor

- 2 p-n junctions
- Switch, amplifier

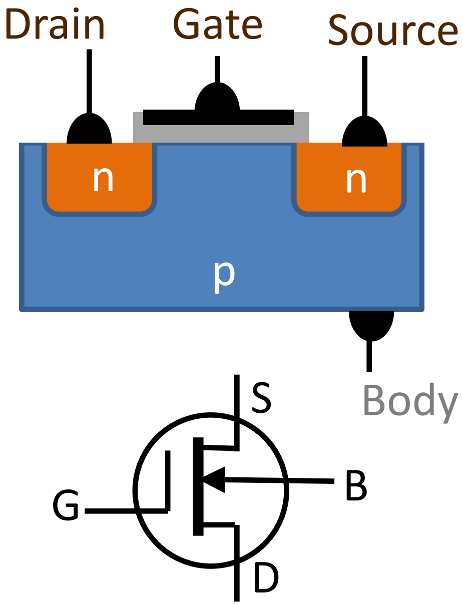
Bipolar



Unipolar



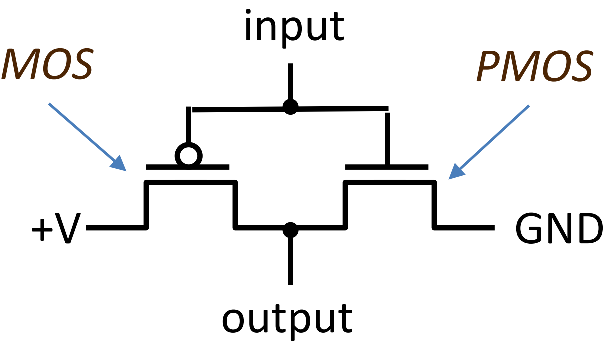
n-channel JFET

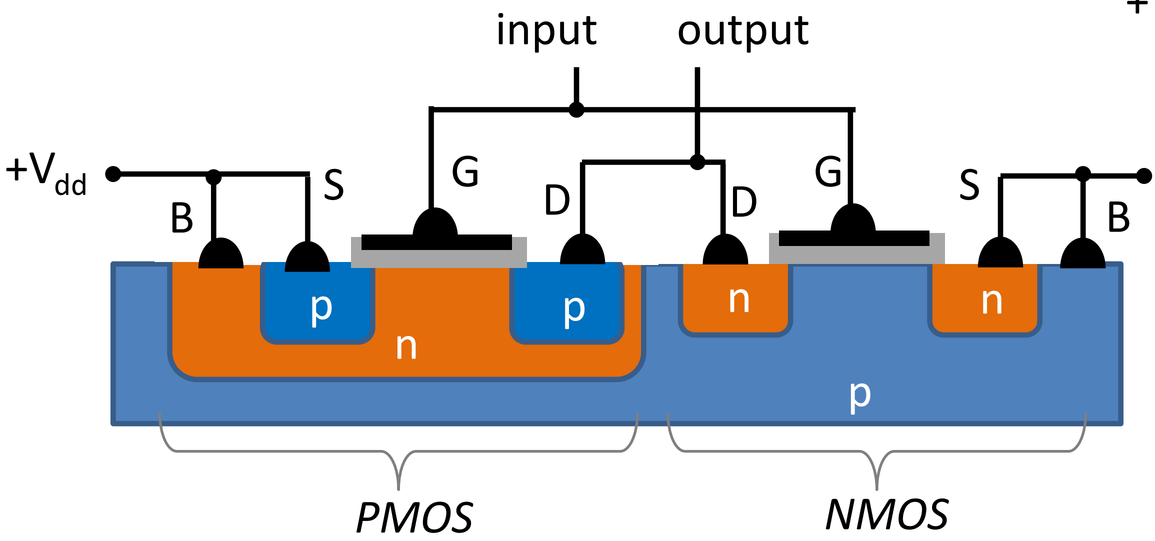


n-channel **MOSFET**

CMOS

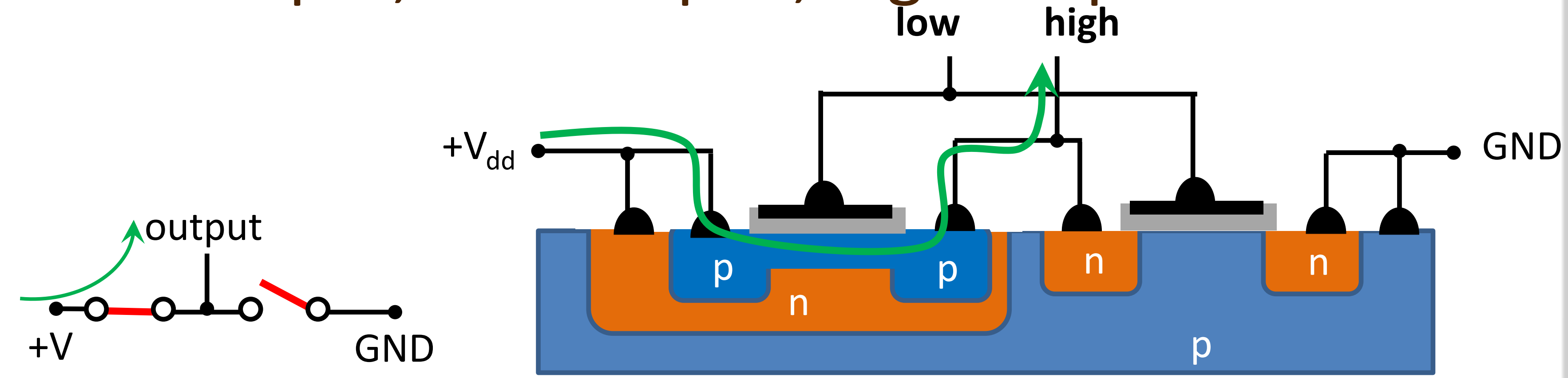
- Complimentary Metal Oxid Semiconductor
- CMOS=PMOS+NMOS
- Inverter funcionality



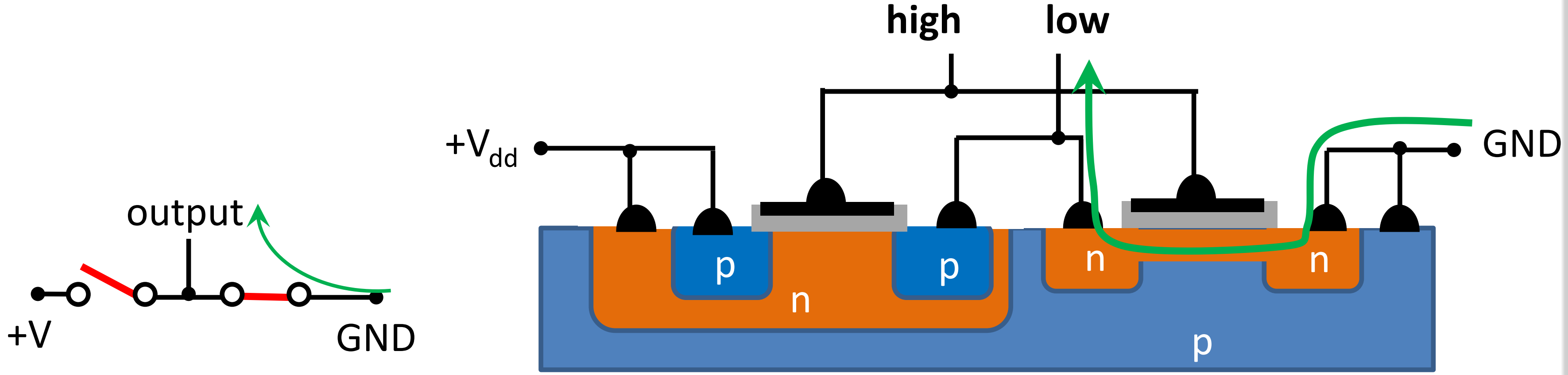


CMOS: inverter

- Low input, PMOS open, high output

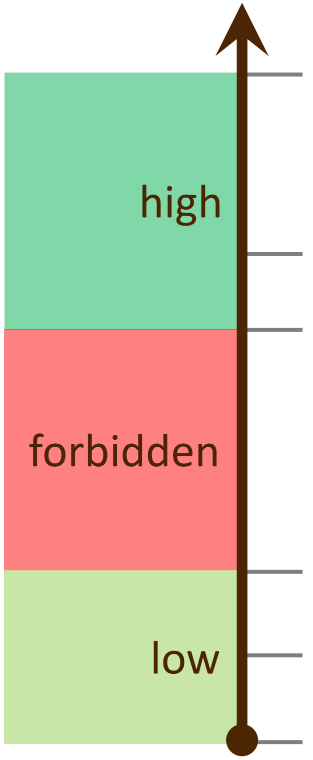


- High input, NMOS open, low output



Logic levels

- Mapping voltage levels and bits
  - Positive logic: high = 1 ; low = 0
  - Negative logic: high = 0 ; low = 1



CMOS

V_{dd} = 3.3V Supply

V_{OH} = 2.4V Minimum high output

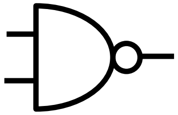
V_{IH} = 2.0V Minimum high input

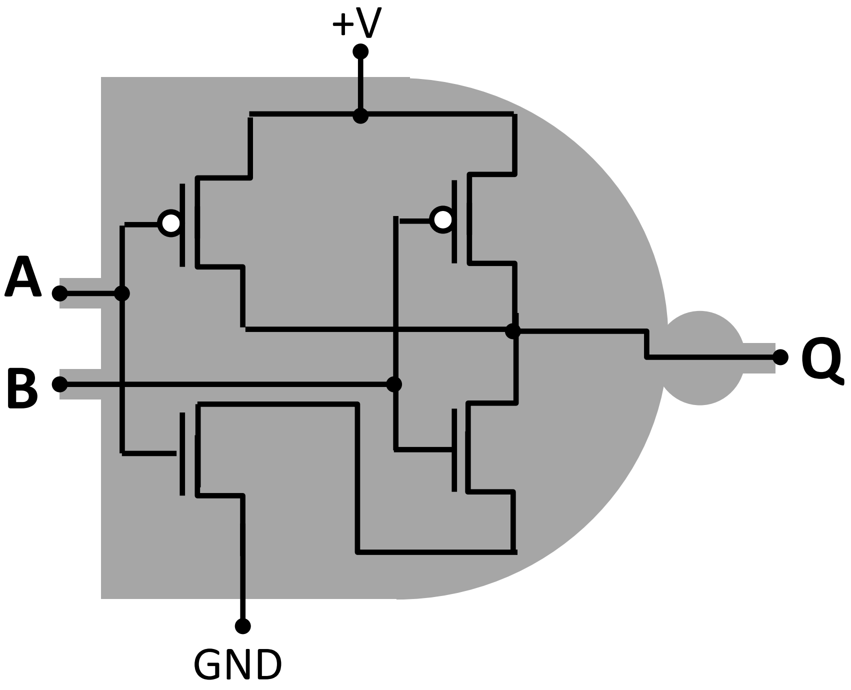
V_{IL} = 0.8V Maximum low input

V_{OL} = 0.4V Maximum low output

GND = 0.0V Ground

NAND gate

- 2-input logical gate („not and”)
- Sign: 
- Structure, function (using CMOS):

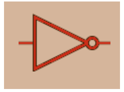


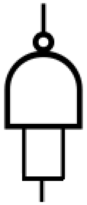
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

Further logical gates

- Can be build from NAND gates

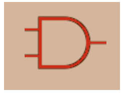
NOT

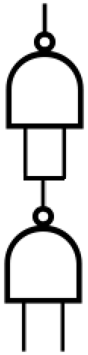




A	Q
0	1
1	0

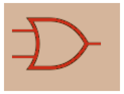
AND

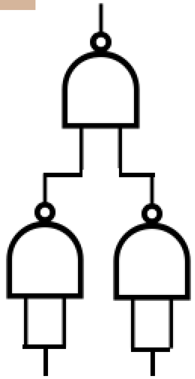




A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

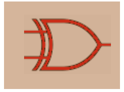
OR

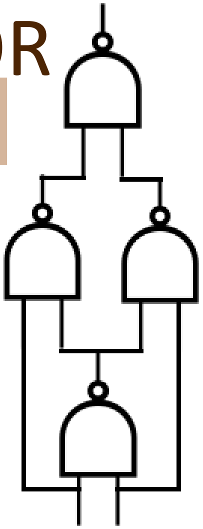




A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

XOR





A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Digital circuits

Combinational logic

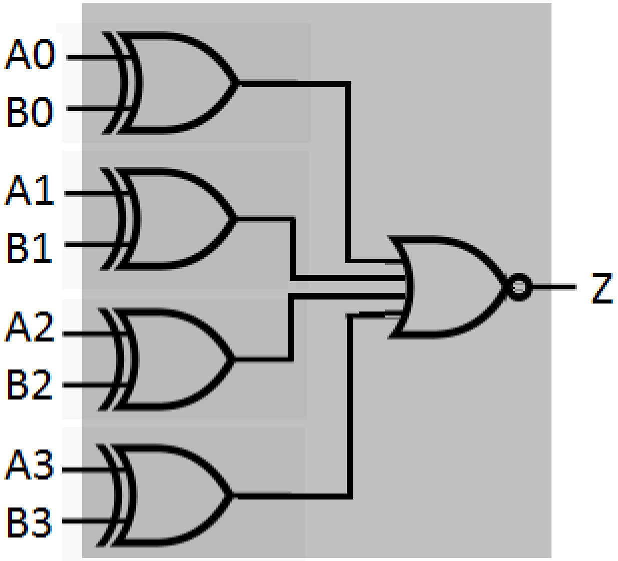
- Output depends on only input
  - Karnaugh map
- Stateless
- E.g.: comparator, multiplexer, decoder, adder

Sequential logic

- Output depends on input and actual state
- Finite State Machine
- E.g.: latch, flip-flop, register, memory

Digital comparator

- Combinational logic circuit (stateless)
- Compares the two digital input
  - If the two input bit sequences are the same, the output is **1**
  - If the two input bit sequences are the different, the output is **0**

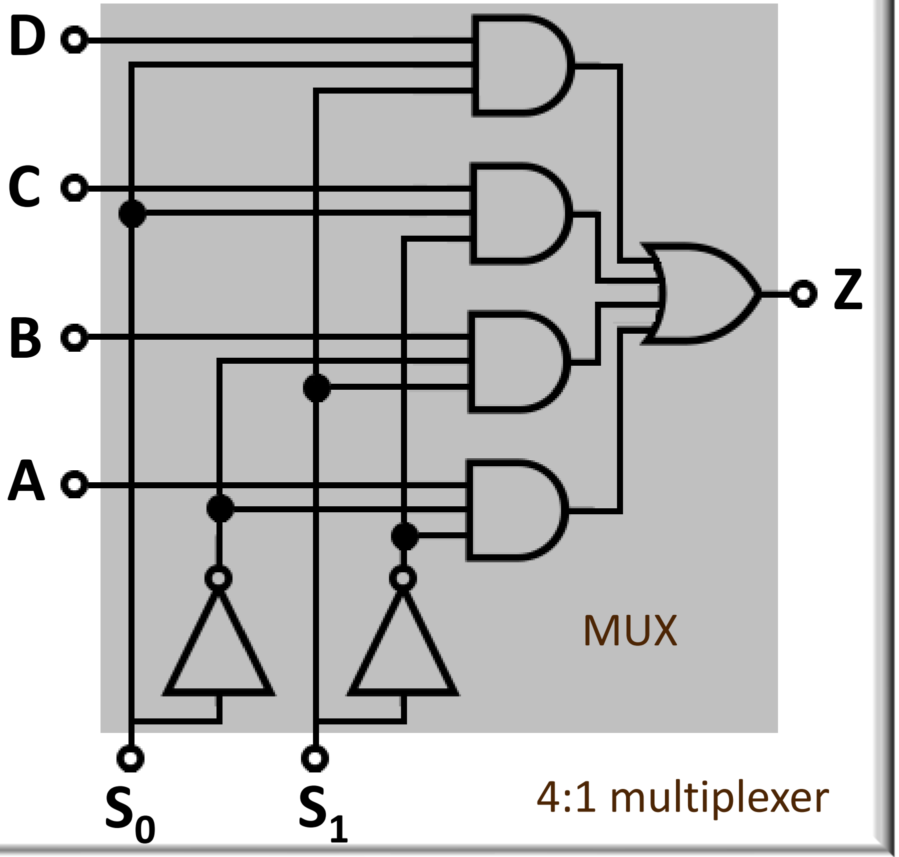


4-bit comparator

Multiplexer

- Combinational logic circuit
- It forwards one signal from the N input, based on the control signals
- K control input identifies 2^K inputs

S_0	S_1	Z
0	0	A
0	1	B
1	0	C
1	1	D

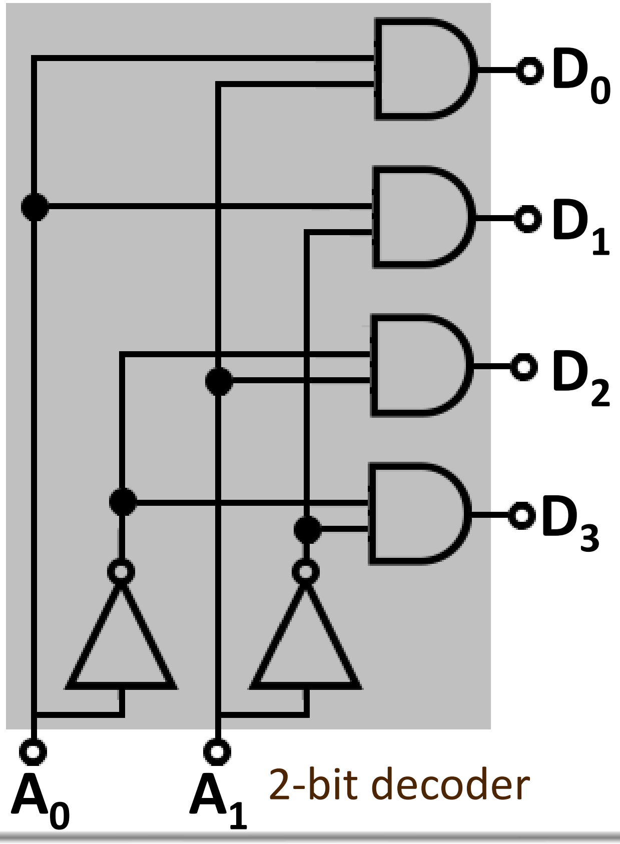


4:1 multiplexer

Decoder

- Combinational logic circuit
- Based on the address input activate only one output line
- An N bit address determines one of the 2^N output line

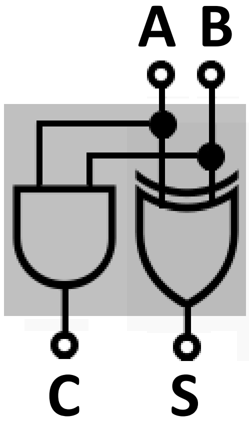
A_0	A_1	D_0	D_1	D_2	D_3
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



2-bit decoder

Adder circuits

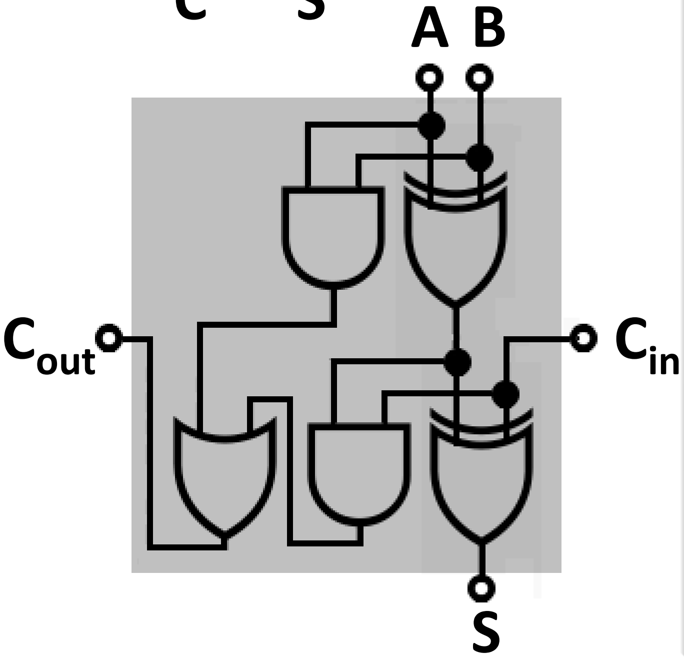
- Half adder: sum of 2 bits



A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- Full adder: sum of 3 bits (2 half adders)

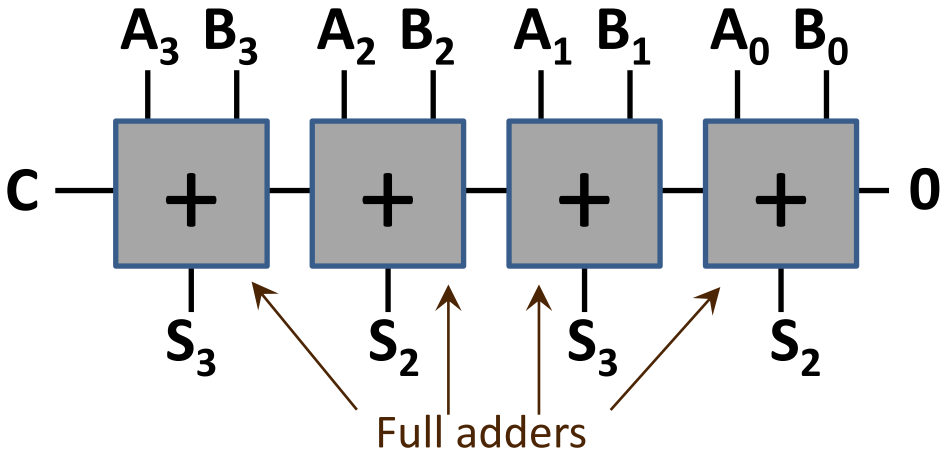
A	B	C_i	C_o	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1



Multi-bit adders

Ripple-carry adder

- Linked full adders

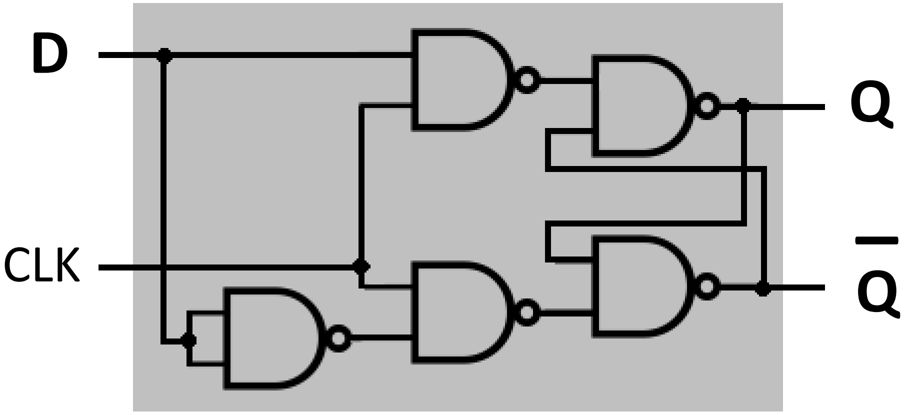


Carry look-ahead adder

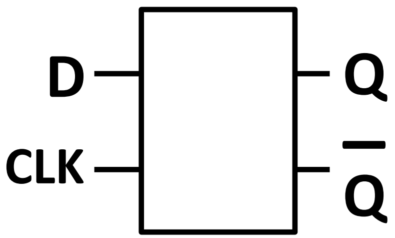
- Parallel carry
- Faster operation than ripple-carry adder

Latch

- Sequential logic circuit
- Able to store 1 bit
- **Level triggered**: If CLK is high (1), then stores the value of input D, else (CLK=0) keeps it previous state.

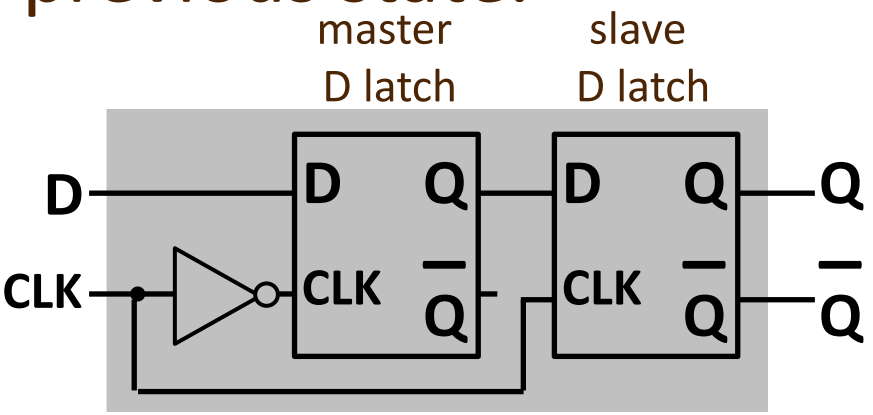


D latch symbol:

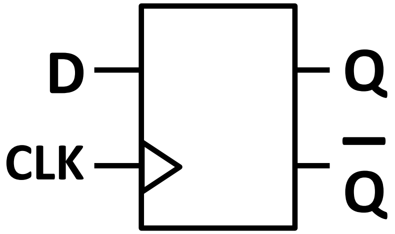


Flip-flop

- Sequential logic circuit
- Able to store 1 bit
- **Edge triggered**: If clock has rising edge (in the moment when CLK jumps from 0 to 1), then stores the value of input D, else keeps it previous state.



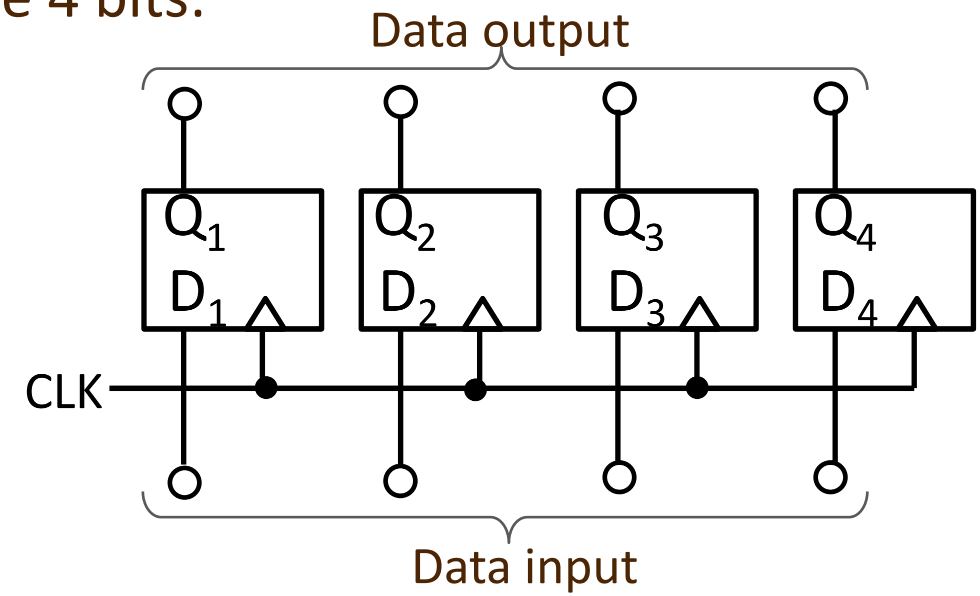
D flip-flop symbol:



Register

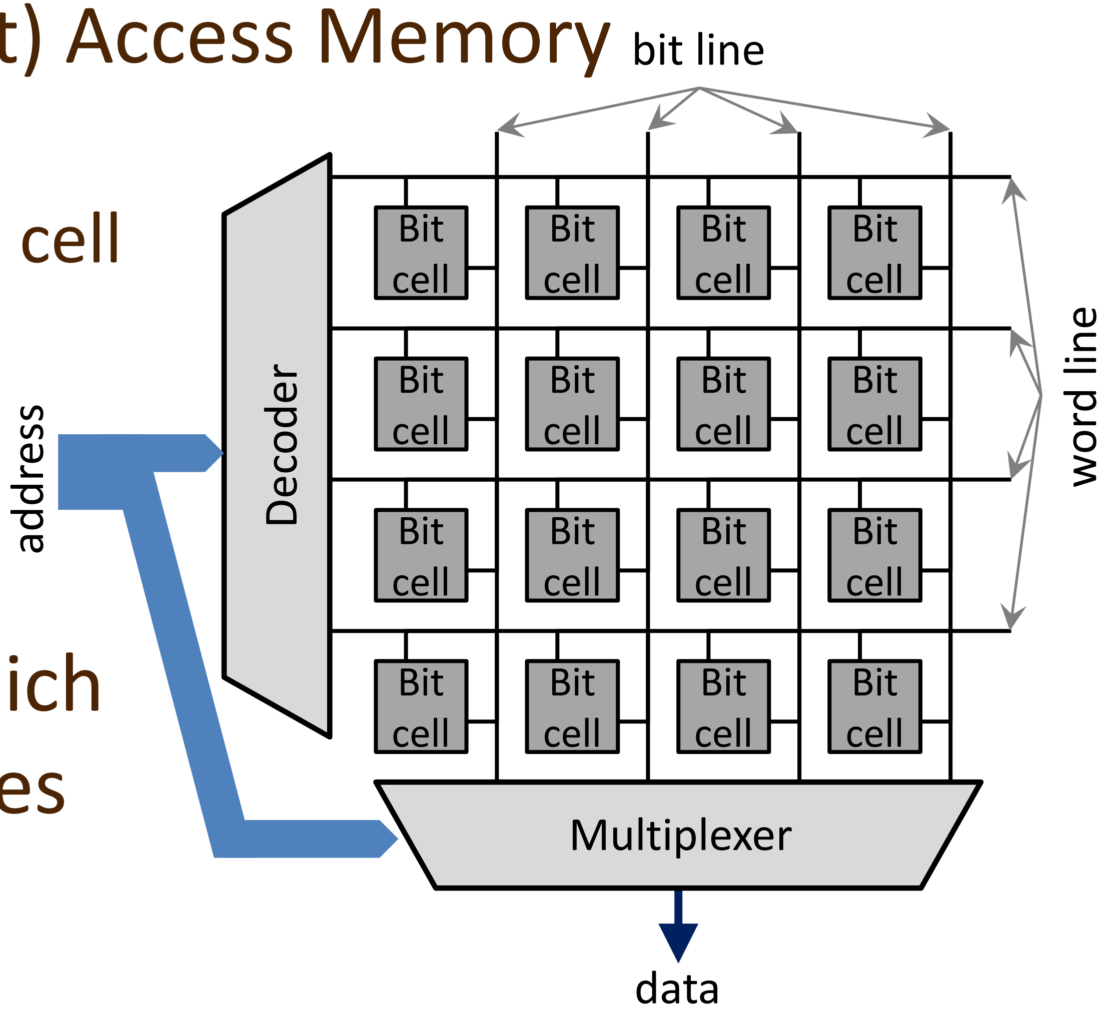
- ✦ Sequential logic circuit
- ✦ Able to store N bits
  (N flip-flop with common clock)
- ✦ Application: storage or shift register

register to store 4 bits:

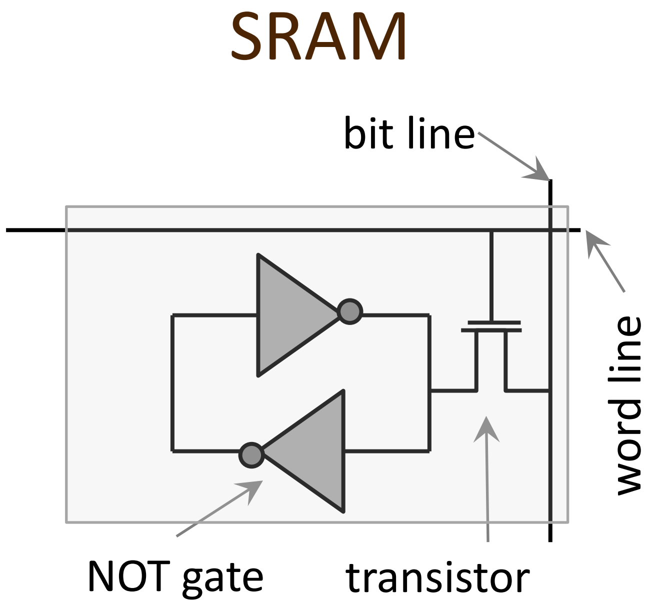


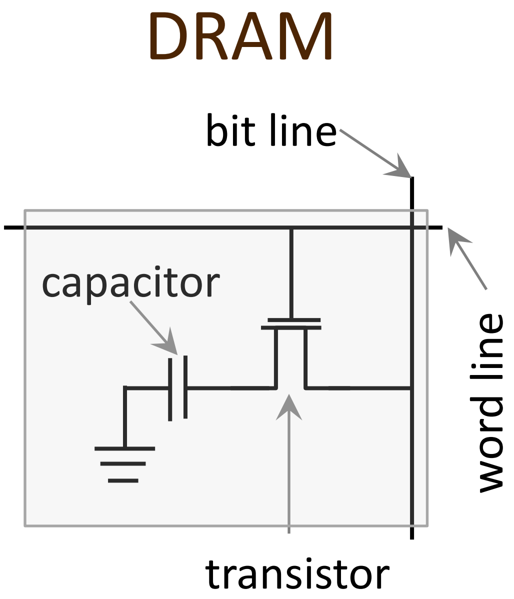
Memory

- Random (direct) Access Memory
  - SRAM or DRAM difference in bit cell
- Bit cell: stores 1 bit
- The address determines which cell content goes to the output



Bit cell

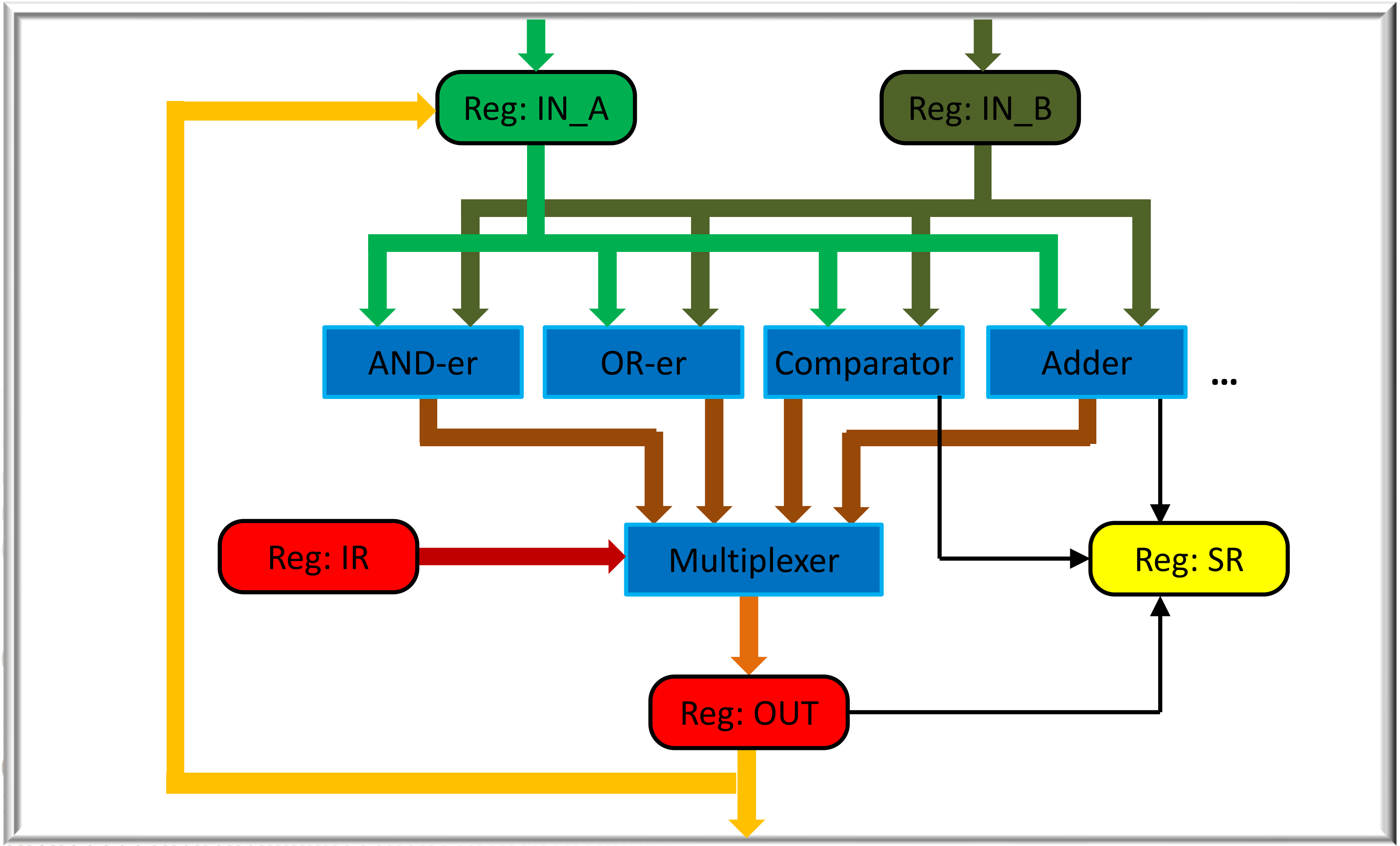




- 5-6 transistors
- Fast output
- „Permanent" store

- Simple structure
- Slow output
- Leakage → update

Primitive ALU outline



Schematic structure of computers

Architecture levels

Processor

Bus system

Memory

Peripheral

Layered computer model

- Layer 0:
  digital logical circuits (logical gates, flip-flop)
- Layer 1:
  micro-architecture (concrete ALU, inner „tricks”)
- Layer 2:
  Instruction-Set Architecture (ISA)
- Layer 3:
  operating system (schedule, storage handling)
- Layer 4:
  programming languages (applications)

Von Neumann architecture

Princeton architecture (John Neumann)

Neumann’s principles (1945):

- Electronic operation
- Application of binary numbers
- Sequential execution
- Universal availability
- Stored program principle
  - Data and program stored in the same place/way

Harvard architecture

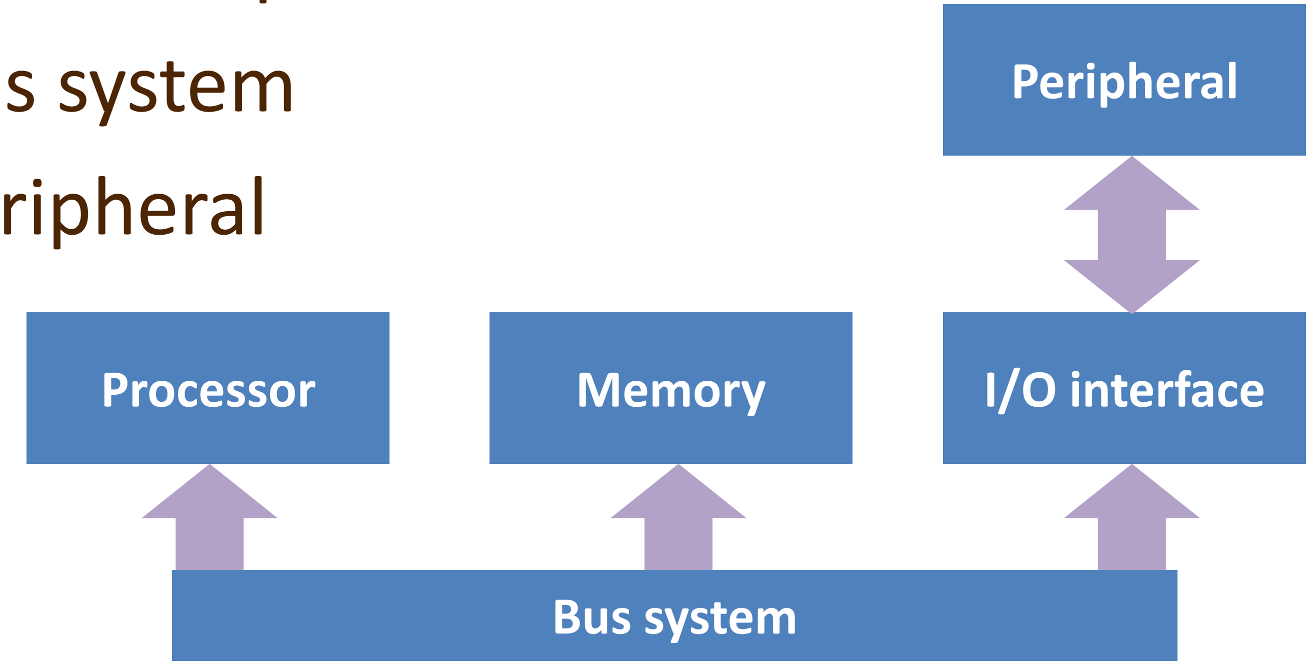
- Data and program stored in different memory
  - Different structure, address domain, type
- Higher performance due to the parallel data and instruction access
- Mainly used in microcontrollers and in DSP
- First issue: Mark I (1944)

Modified Harvard architecture

- Separate data/instruction cache, common memory
- Current in modern CPUs

Schematic structure of computers

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



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

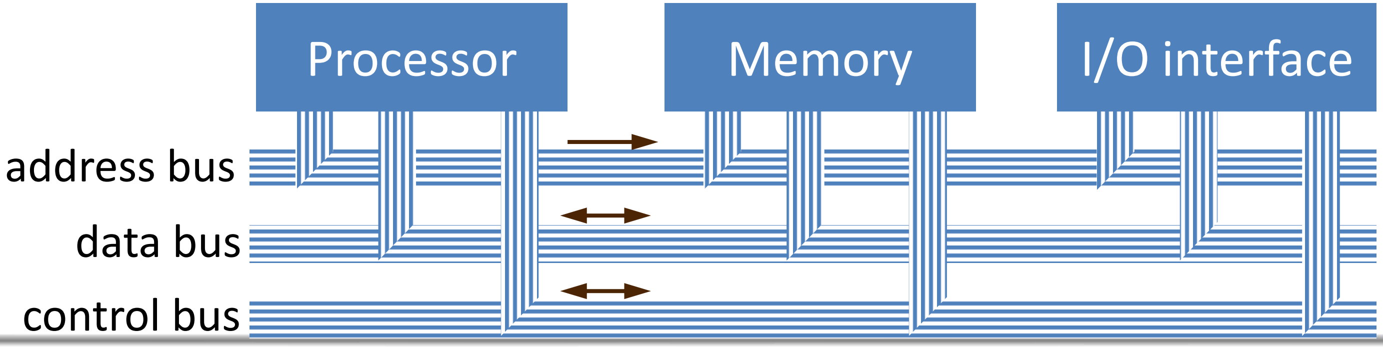
Memory

- 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

Bus system

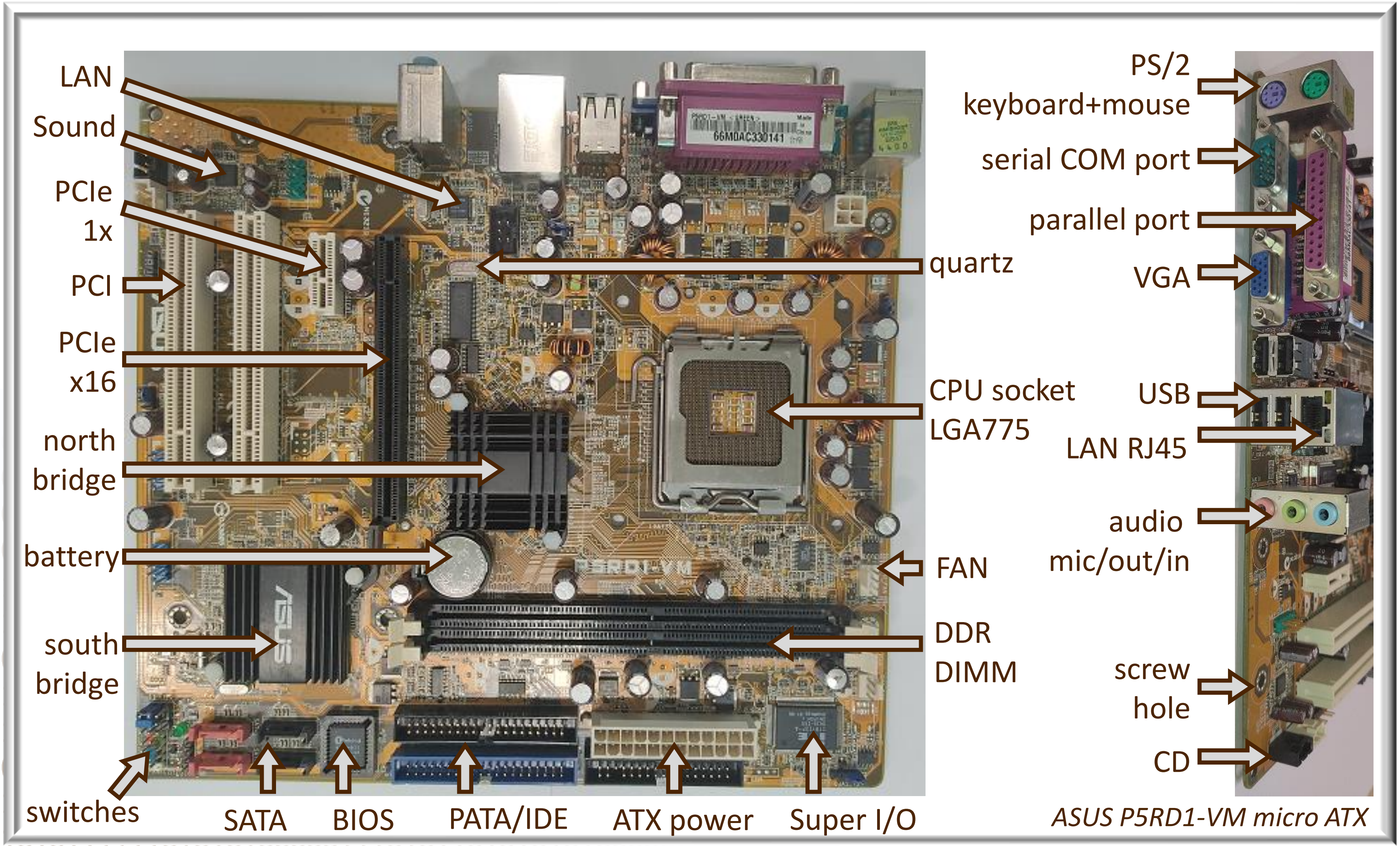
- Bus: a group of wires to transmit digital signals
- Logical component of bus systems
  - Address bus: to transmit addresses
  - Data bus: carries data bits
  - Control bus: transmit signals to synchronize the operation of components



Motherboard

- In general-purpouse computers, it is a printed circuit board to which other devices are connected:
  - Processor
  - Memory
  - Expansion cards (video, sound, network, etc.)
  - Data storage
  - Power supply
- It contains buses, connectors/sockets, control chips (BIOS, south/north bridge), battery, configuration jumpers, etc.

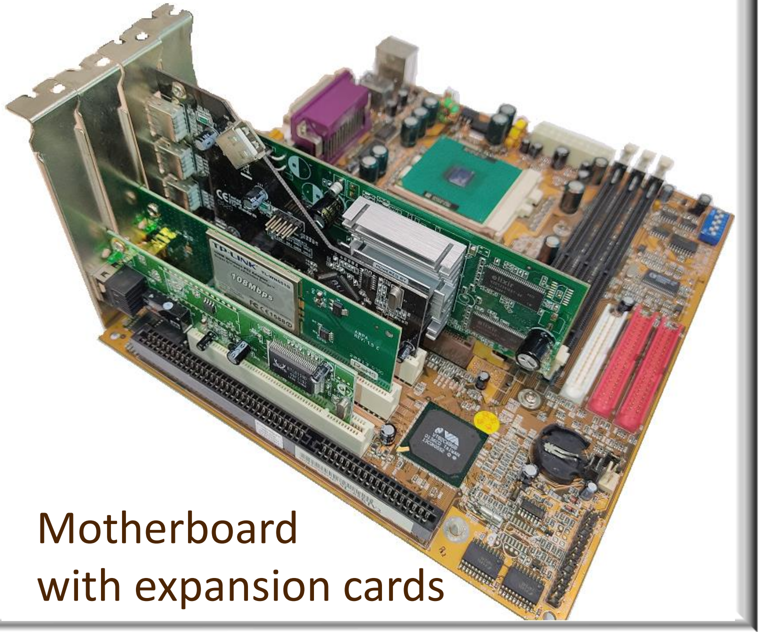
Motherboard



ASUS P5RD1-VM micro ATX

Expansion cards

- Separated integrated circuit boards
- Can be connected to the motherboard
- Extends the abilities of the computer or improve the performance
- For example
  - Video/graphics card
  - Sound card
  - Network card
  - Special hardware connection



Motherboard with expansion cards

I/O interface

- ✦ Provides data transmission between the central and peripheral parts of a computer
- ✦ The input/output devices are connected to **ports**
- ✦ Ports have addresses (similar to memory)
- ✦ During data transmission they often use buffers for temporary storage
- ✦ On software level they need drivers

Peripherals

Input

- Keyboard
- Mouse
- Scanner
- Camera

Output

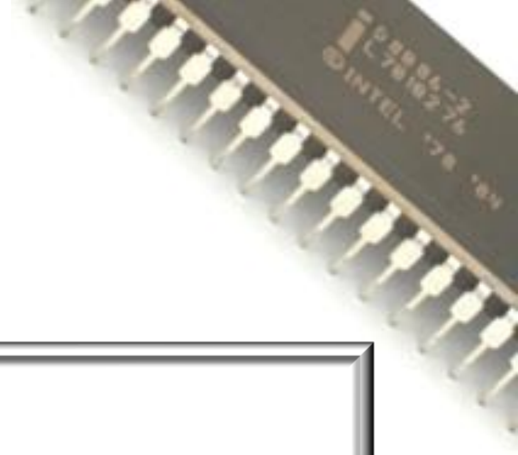
- Monitor/display
- Printer
- Projector
- Speaker system

Storage

- HDD, SSD
- CD/DVD/Blu-ray
- USB storage
- Memory card

Network

- Ethernet card
- Wi-Fi card
- Bluetooth



The processor: structure and operation

CPU

Registers

Fetch-execute cycle

RISC / CISC processors

Thought-provoking questions

- How can a processor (which is a big cuircit) execute instructions (which are bitsequences)?
- Does the processor store data, subresults? Where and how much?
- The program and datas are in the memory. How does the processor know where and when to execute what instructions?

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)

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.

Registers

Program Counter register (PC or IP)

- ✦ Stores the address of the next instruction
- ✦ During the execution automatically updated (usually) to the next address, so incremented by the size of instruction
- ✦ Tool of sequential execution
- ✦ Some instruction can overwrite its value (control passing)
- ✦ Related to control unit (CU)

Registers

Instruction register (IR)

- It contain the operation code of the actual instruction read from memory
- Related to control unit
- CU makes decision based on its content
  - Which operation have to be executed
  - Which/where is the data working on
- Programmer can’t access it directly

Registers

Accumulator (ACC, W, AX)

- ✦ General working register in the ALU
- ✦ Operands of arithmetic and logic operations are stored here
- ✦ Generally, the results of these operation are also stored in accumulator
- ✦ Most processors have more similar registers for this purpose (to reduce memory operations)
- ✦ Generally, contains integer type data

Registers

Status register (SR, FLAGS)

- Marker/control bit about the processor status, operation results
- Important in conditional control pass (as well)
  - Carry (CF)
  - Nibble carry (AF)
  - Overflow (OF)
  - Sign (SF)
  - Interrupt mask (IF)
  - Zero (ZF)
  - Parity (PF)
  - …

Registers

Stack pointer register (SP)

- For the use of (hardware) system stack
- Stores the memory address of top of stack
  - Stack usually grows toward lower addresses
  - Often SP refers to the last item in stack (x86)
- Special purpose register
- The „push” and „pop” operations automatically updated its value

Registers

Special registers related to addressing

- Determining the address of operands (E.g., relative to a base address)
- Related to AU (and BIU)
- Useful in case of arrays, string and local variables or handling memory segments
- Examples: BP, X, SI, DI, CS, DS, SS, ES

Registers

Memory address register (MAR)

- ✦ Connects the (outer) system bus to the inner address bus
- ✦ Contains the memory address for the next operation (reading and writing)
- ✦ Driven by the addressing unit
- ✦ Size is equal to the width of address bus

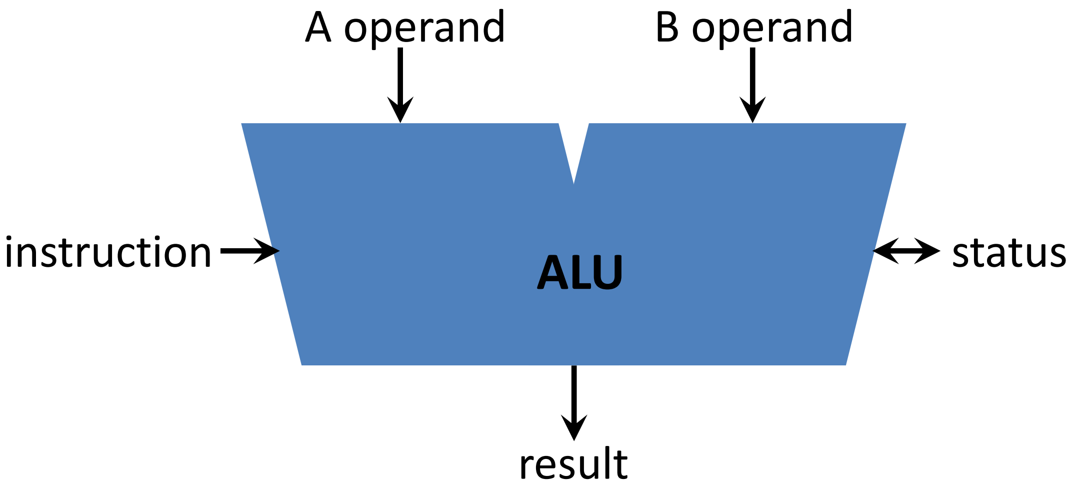
Registers

Memory data register (MDR)

- Connects the (outer) system bus to the inner data bus
- Contains the data
  - for next memory writing
  - from last memory reading
- Two-way register
- Size is the width of data bus

Arithmetic logic unit

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



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)

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

Clock signal

- Periodical electronical square wave
- Clock generator (oscillator) produces it
- Synchronize the operation of units
- Frequency is proportional to the heat produced by the CPU
- Period is greater than signal propagation time
- 1 MHz – 4 GHz
- Often variable (turbo, powersave)
- Today not proper for speed characteristic (FLOPS)

CISC processor

Complex Instruction Set Computer

- Many complex instructions (large code density)
- Lot of addressing modes
- Variable length instructions
- Several clock cycle for execution of instruction
- Microprogrammed (μops)
- Few registers
- Instructions can access RAM
- E.g.: IA-32, Motorola 68k

RISC processor

Reduced Instruction Set Computer

- Few, simple instructions
- Few addressing modes
- Fixed-length instructions
- Execution within one clock cycle
- Hard-wired or microprogrammed
- Several registers
- Only Load/Store instructions access RAM
- E.g.: ARM, MIPS, AVR

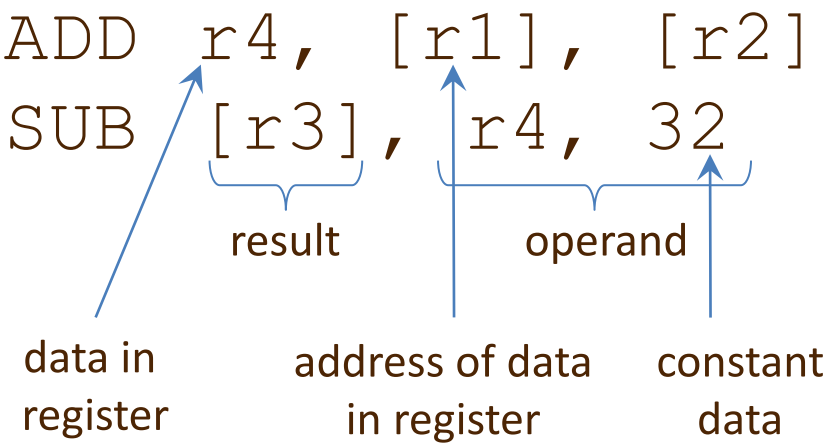
CISC vs RISC

- Example:

  `x = a+b-32;`

  r1: address of `a`; r2: address of `b`; r3: address of `x`

CISC



```
ADD  r4, [r1], [r2]
SUB  [r3], r4, 32
```

RISC

```
LDR  r5, [r1]
LDR  r6, [r2]
ADD  r4, r5, r6
SUB  r7, r4, 32
STR  [r3], r7
```

Pseudo-assembly

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

Fetch-Execute cycle

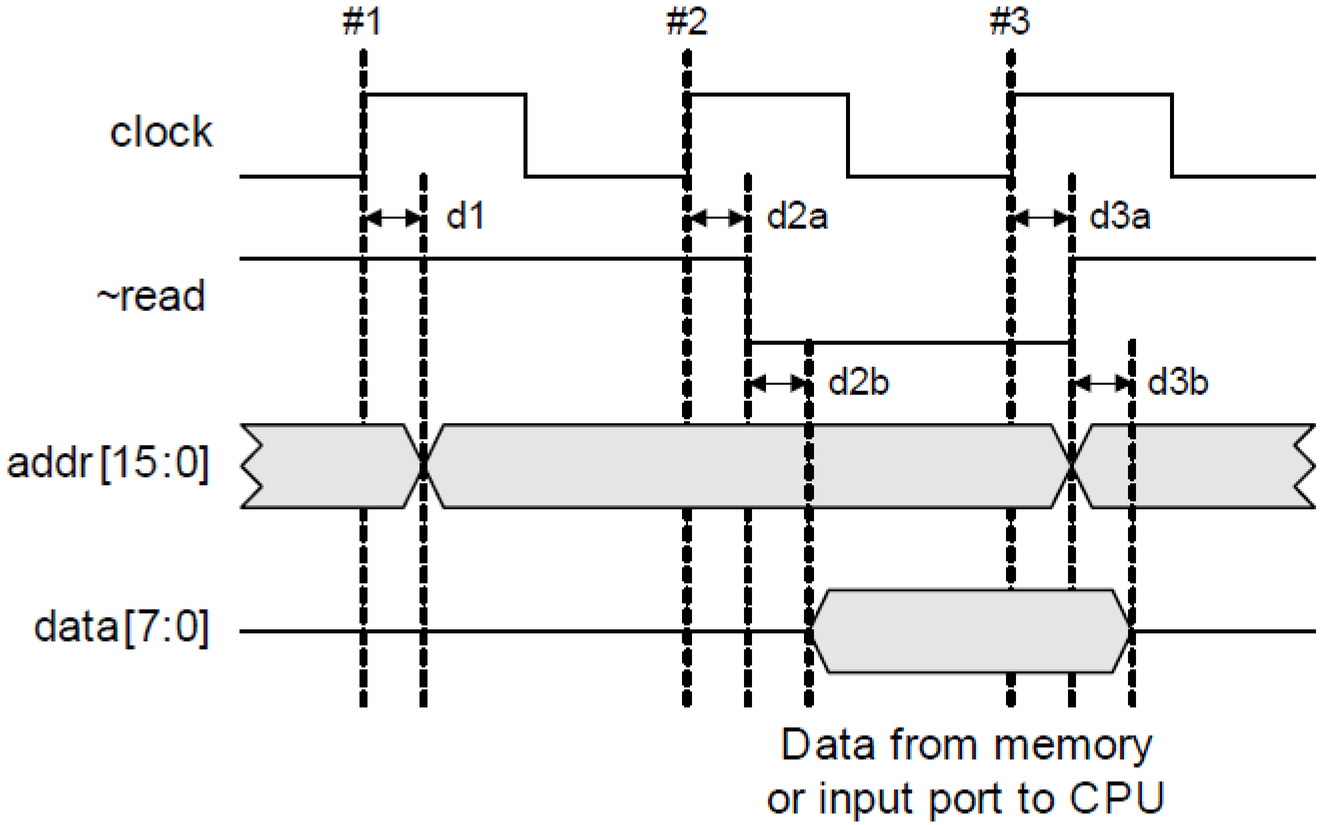
1. Instruction fetch (IF)
   PC register refers to the memory address of the next instruction. Reading from here to IR.
   PC is updated to the address of next instruction
2. Instruction decoding (ID)
   Interpretation of the opcode. What kind of operation? What is the input data? Where to save the result?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

Fetch-Execute cycle

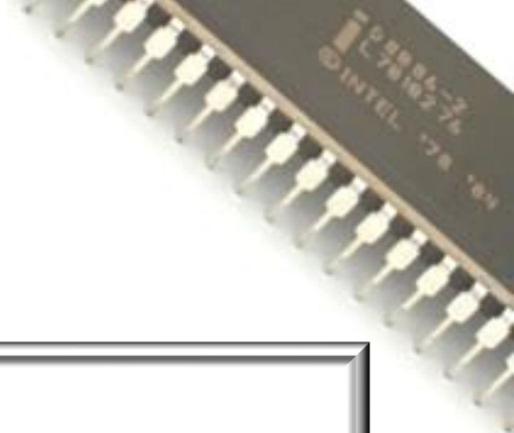
3. Execution or address calculation (EX)
   ALU works, result into internal temporal register, in case of Load/Store instruction calculation of the proper memory address.
4. Memory access (MEM)
   In case of Load/Store instruction reading/writing of given data memory address.
5. Writing back (WB)
  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

General reading cycle

- Example: in case of DIY Calculator architecture



C. Maxfield, A. Brown: The Official DIY Calculator Data Book



Interrupts and input/output

Interrupt, exception

Interrupt handling

Input/output

Bus systems, peripherals

Thought-provoking questions

- The processor continuously repeats the fetch-execute cycle during the run of the instructions of the given program.
  In the meantime, how can the processor respond to the environment, expected or unexpected events?
  What if a keystroke occurs or an IP packet arrives?
- Everithing is done by the processor in the computer? For example, does it load a whole file from HDD to RAM?

Interrupt

- Ability of general-purpose computers to interact outer world (keyboard, mouse, network interface, monitor)
- Devices need „prompt” response from processor independently of its actual activity
- CPU must respond to unexpected events
- **Interrupt** draw the attention of CPU

Interrupt

- ✦ Special signal for CPU
- ✦ Indicate an event supervening
- ✦ Prompt reaction necessary
- ✦ The „normal” execution have to be interrupted
  - ➥ Later it have to be resumed
- ✦ Sources
  - ➥ Hardware
  - ➥ Processor
  - ➥ Software

Types of interrupts

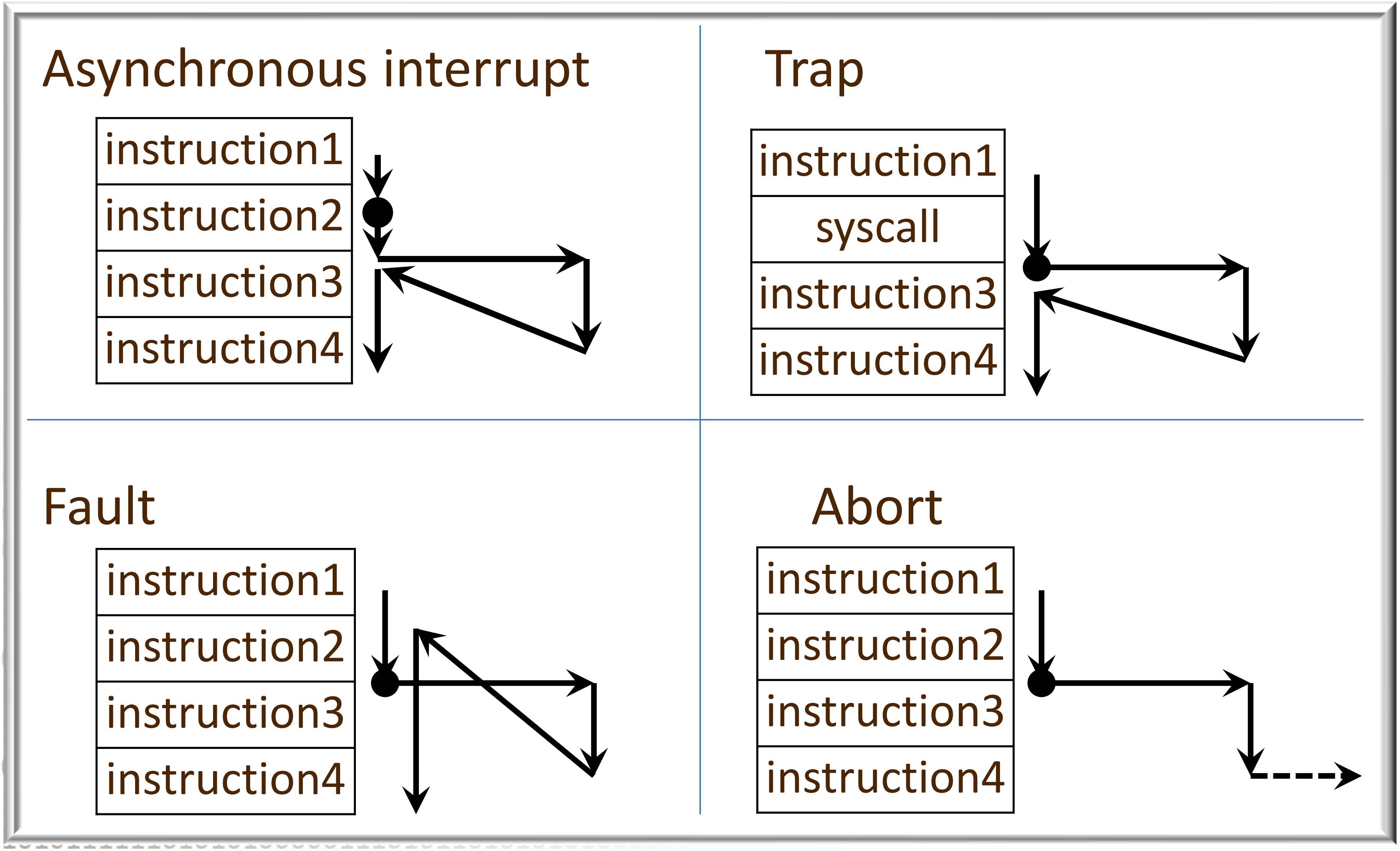
Asynchronous

- From external sources (E.g., from I/O device)
- Independent of recently executed instruction

Synchronous (sometimes „exception”)

- Triggered by the processor
  - Fault: „corrigible” problem, trying again
  - Abort: significant, „incorrigible” (hardware) error
  - Trap: E.g., debugging, programmed interrupt (system calls)

Types of interrupts



Asynchronous interrupt

instruction1
instruction2
instruction3
instruction4

Trap

instruction1
syscall
instruction3
instruction4

Fault

instruction1
instruction2
instruction3
instruction4

Abort

instruction1
instruction2
instruction3
instruction4

Concepts

- Interrupt line
  - One (or more) CPU control bus wire
- Interrupt request (IRQ)
  - Interrupts can be identified by numbers (0-255)
  - They can have priority
- Programmable Interrupt Controller (PIC, APIC)
  - Identifies the source of interrupt, activates the interrupt line, handles priority

Concepts

Interrupt service routine (ISR)

- Program code activated by IRQ, that handles the special situation
- Similar to subroutine „without call”

Context

- To resume the „normal” execution after the interrupt handling, environment (register content) have to be saved and restored

Concepts

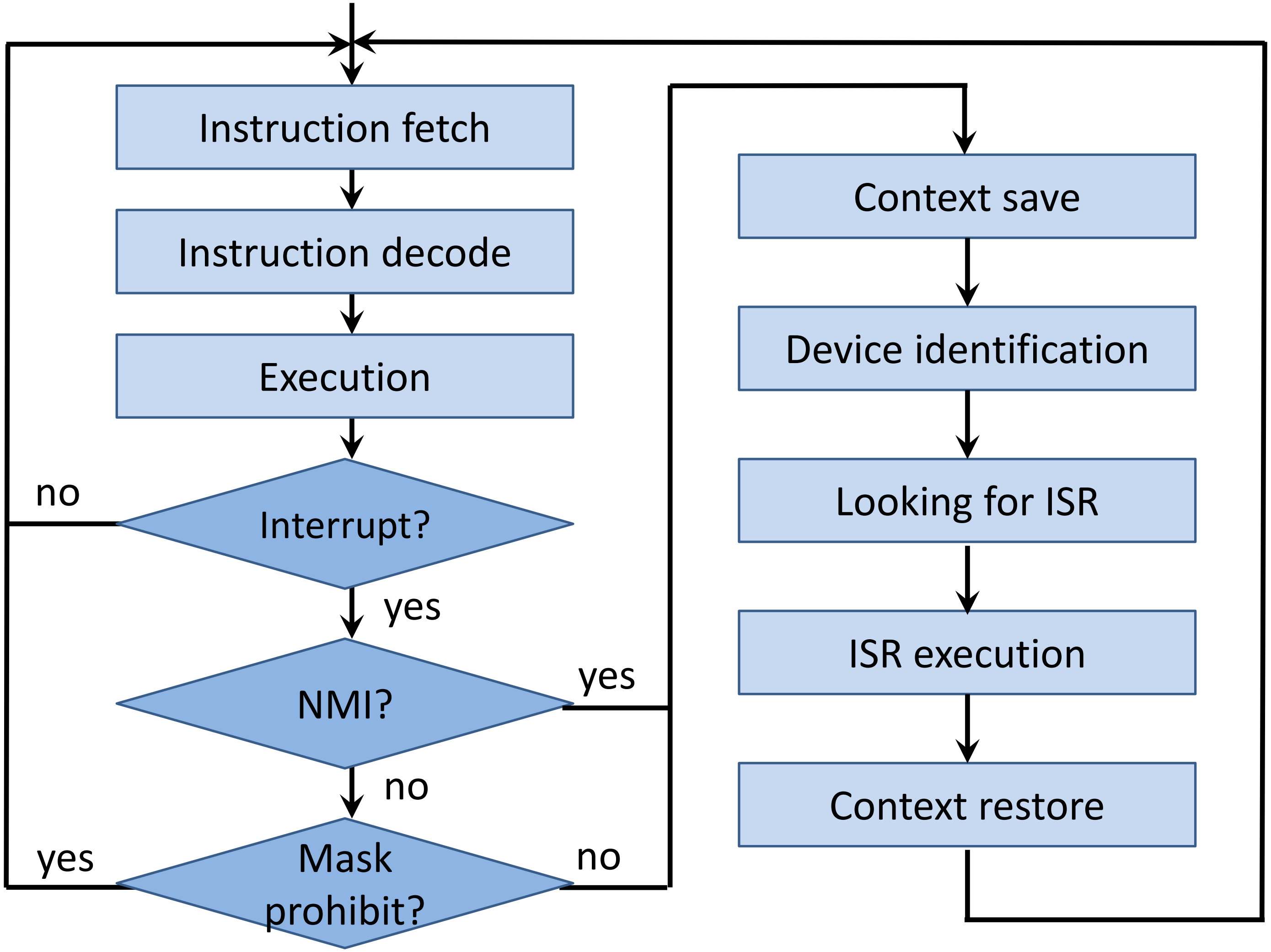
Masking

- Listening to some interrupt can be turned off
  - At the same time, globally in the processor
  - Separately, in PIC with the help IMR
- There are Non-Maskable Interrupts (NMI)

Nested interrupt

- Interrupt service routine can be interrupted by another IRQ
- Interrupt in interrupt handling

Interrupt handling



Interrupt handling techniques

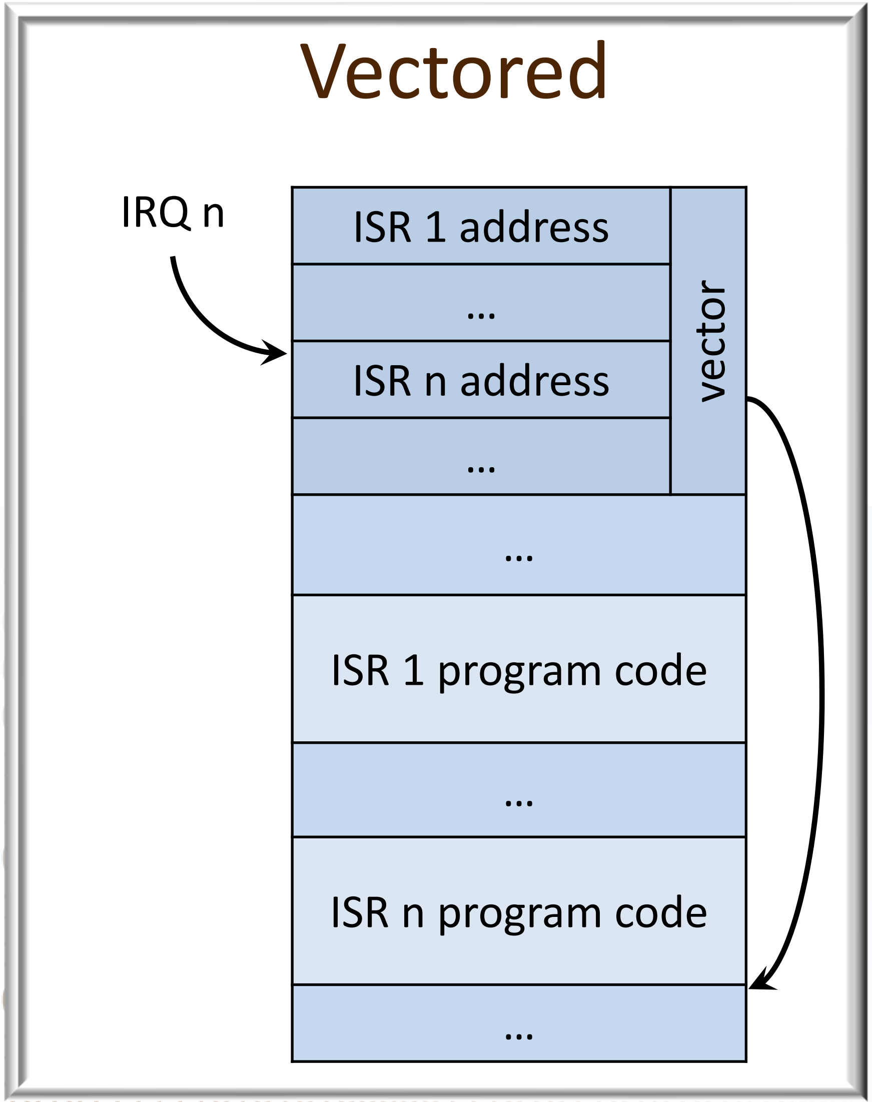
Vectored

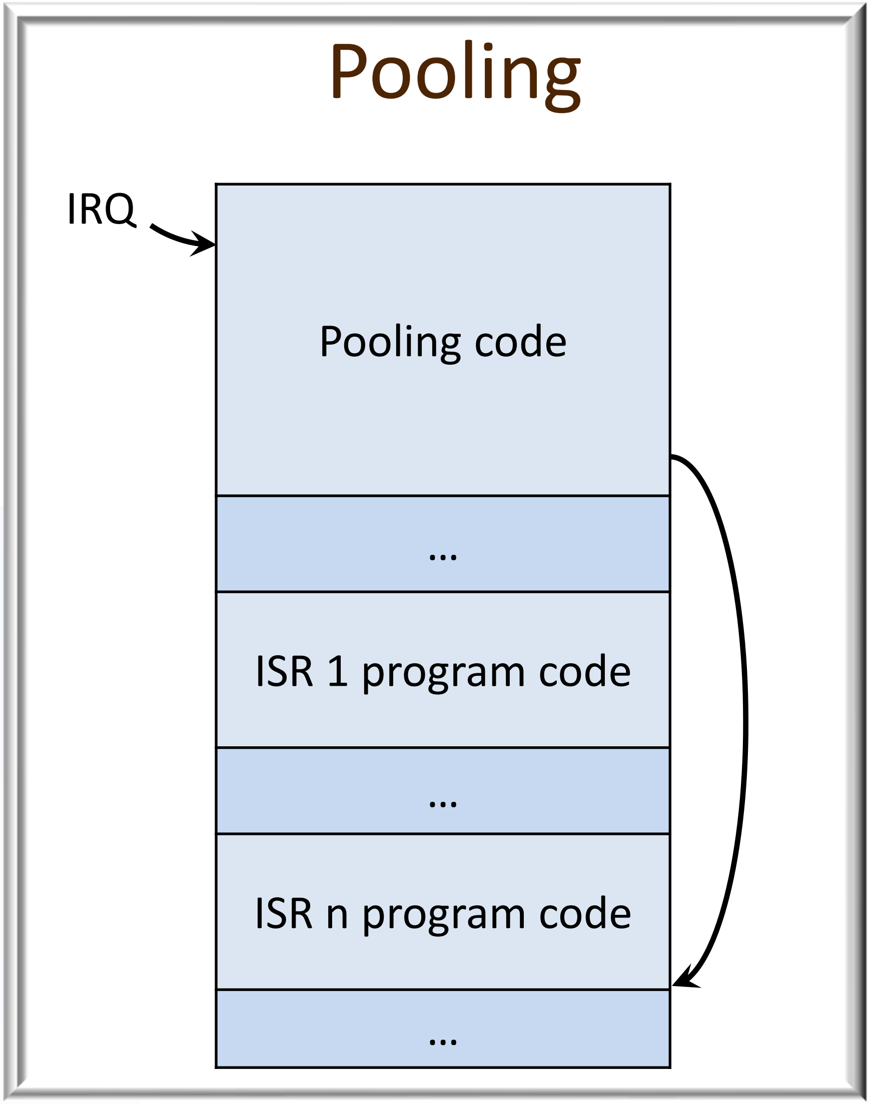
- Interrupt line activated
- Source of IRQ identifies itself
    - Special code/address is sent on bus system
- ISR addresses in an table

Polling

- Interrupt line activated
- CPU sequentially ask devices to find the device with demand

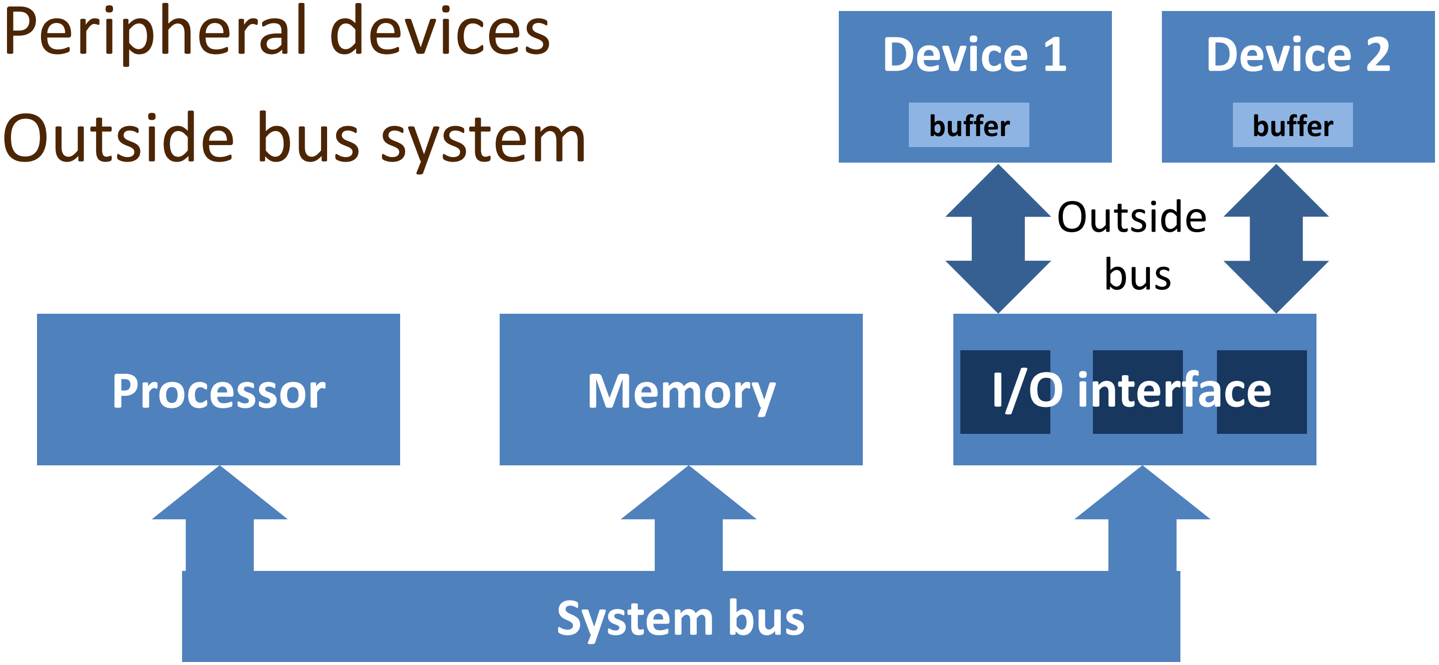
Interrupt handling techniques





Input/Output

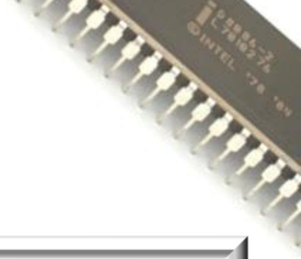
- Connection to outside world
  - Hardware I/O interfaces
  - Software interfaces (device drivers)
  - Peripheral devices
  - Outside bus system



Input/output interfaces

- Connection between CPU and peripherals
- Addressable devices
  - Memory-mapped I/O (MMIO)
    one address space, E.g. RAM: 0x0000-0xEFFF, I/O: 0xF000-0xFFFF
  - Port-mapped I/O (PMIO)
    separate address spaces, E.g. RAM: 0x0000-0xFFFF, I/O: 0x0000-0x03FF
- Different transfer speed
- Different physical signals
- Serial and parallel transmission
- Device control is not the task of CPU

I/O techniques



Programmed I/O (PIO)

- Special I/O instruction (in, out)
- The program tells when to apply to peripheral (independently of its state)
- Transmission of a word per instruction between I/O buffer and CPU
- I/O data and address registers in CPU
- Continuous device state check of CPU is time consuming

I/O techniques

Interrupt controlled I/O

- Peripheral applies interrupt to tell to CPU…
  - If new input is available
  - If ready with output
- CPU not continuously checks devices
  - It deals with other things
- Data transfer between CPU and the I/O module

I/O techniques

Direct Memory Access (DMA)

- With separate controller direct RAM-I/O connection (RAM-RAM, RAM-I/O)
- CPU ask DMA controller to transfer data
- When DMA controller is ready sends IRQ
- CPU don’t participate in data movements
- Blocked transmission mode

I/O processor (IOP)

- Extended DMA with more separateness

Bus systems

Parallel

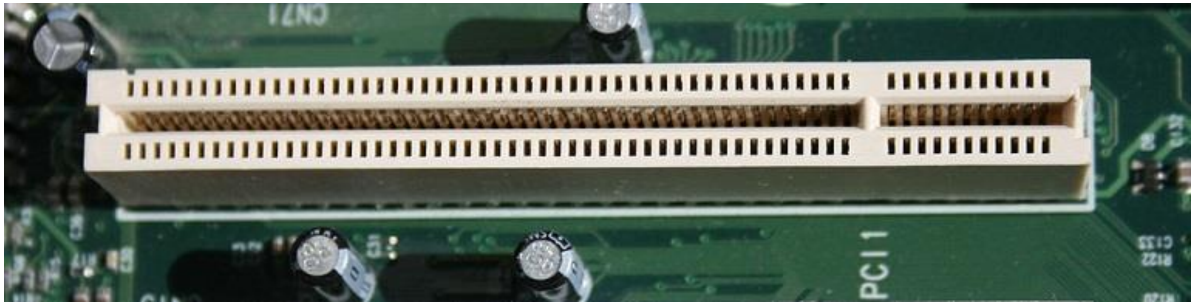
- ISA
- PCI
- ATA (ATAPI, UATA, IDE)
- VESA Local Bus (VLB)
- SCSI
- AGP
- …

Serial

- I^{2}C
- PCI Express
- Serial ATA
- USB
- Serial Attached SCSI
- FireWire
- …

PCI bus

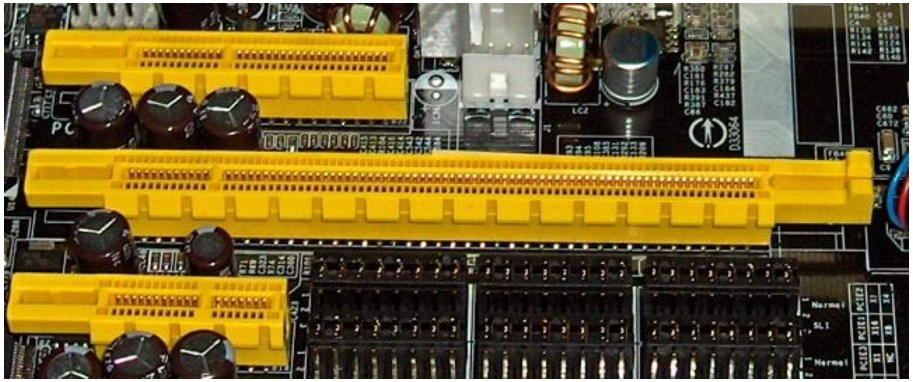
- Peripheral Component Interconnect
- Parallel internal bus
- Width: 32 or 64 bits
- Transmission speed: 133-533 MB/s
- Development: 1992-2002
- Separate control circuit is necessary



Source: en.wikipedia.org/wiki/Conventional_PCI

PCIe bus

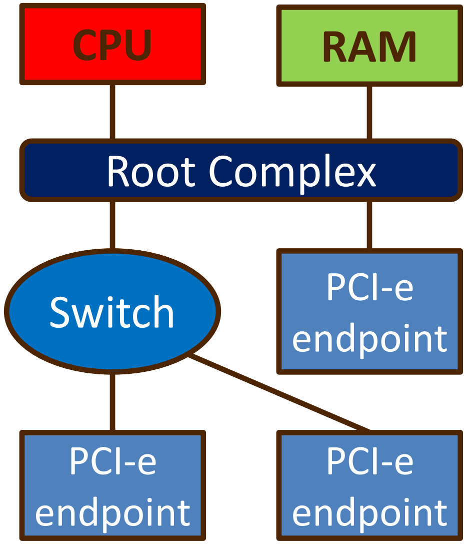
- Peripheral Component Interconnect Express
- **Serial** successor of old parallel PCI
- Covers multiple layers of OSI model
- Development: from 2003
- Application:
  - Graphics card
  - SSD/HDD
  - Network card
  - etc.



Source: en.wikipedia.org/wiki/PCI_Express

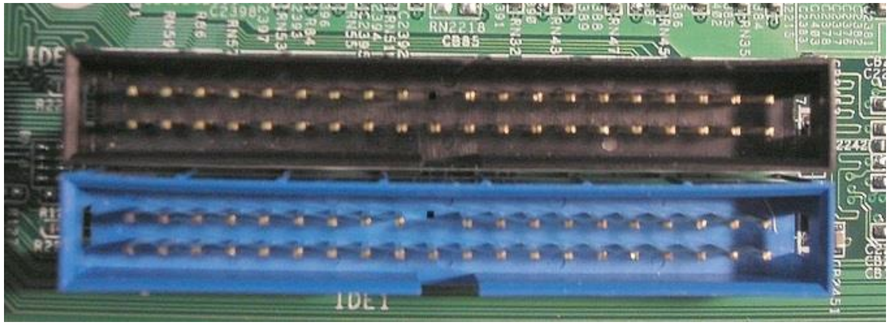
PCIe bus

- Point-to-point connection (link) between devices
  - ➥ Each device has own link (switching)
- 1 link consists of 1, 2, 4, 8, 16 or 32 lanes
- All lanes contains 4 wires
- Dual simplex lanes
- Transmission speed: 250-126000 GB/s (v6.0)
- Hot-swapping



(Parallel) ATA bus

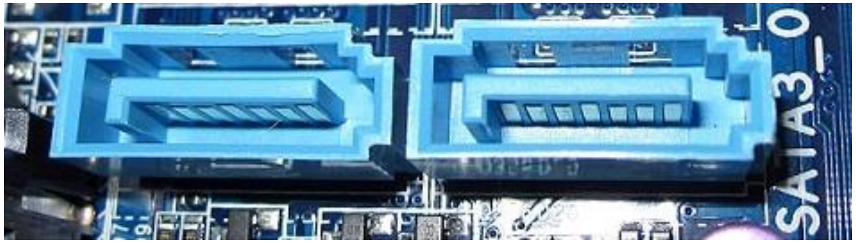
- (Parallel) AT Attachment
- Parallel interior bus
- „Synonyms”: PATA, ATAPI, UATA, IDE, EIDE
- Transmission speed : 3,3-166 MB/s
- Development : 1986-2003
- HDD, CD and DVD drive connection



Source: en.wikipedia.org/wiki/Parallel_ATA

SATA bus

- Serial ATA
- Serial interior bus
- Point-point connection
- Transmission speed : 150-1969 MB/s (v3.5)
- Development : 2003-
- Hot swapping
- HDD, SSD, CD and DVD drive connection



Source: hu.wikipedia.org/wiki/Serial_ATA

USB bus

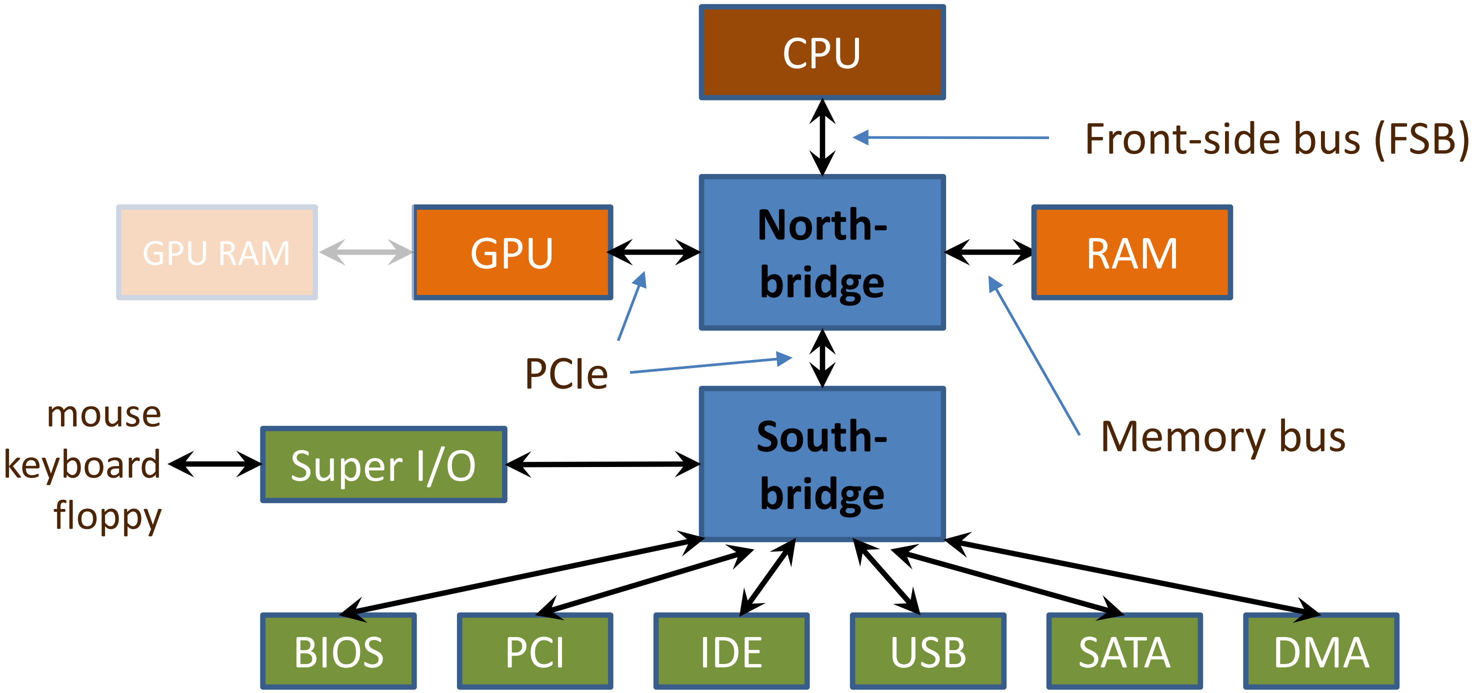
- Universal Serial Bus
- Serial outside bus
- Plug-and-Play, hub
- Transmission speed : 0,18-5000 MB/s (v4.0)
- Development : 1996-
- Connections: standard-A/B/C, mini-A/B/AB, micro-A/B/AB/BSuperSpeed

Source: en.wikipedia.org/wiki/USB

Control hubs

Important chips on motherboard

- ✦ Northbridge (memory controller hub)
- ✦ Southbridge (I/O controller hub)



Device driver

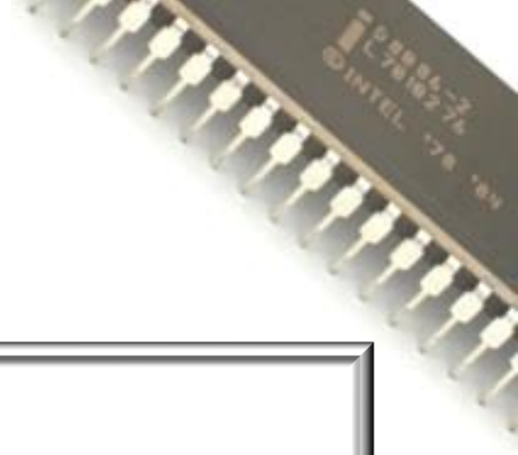
- Software interface to hardware
- Types of devices
  - Character, block, network, other
- Programs apply hardware independent **system calls**
  - open, read, write, close, …
- Device drivers „fill” this with concrete hardware-specific content

BIOS

- Basic Input Output System
- Between Operating System and hardware
- Stored in ROM/Flash on main board
- Initialize and test hardwares
  - Power-on Self Test
- Starts Bootstrap Loader
- Contains I/O handler interrupt service routines
- Text user interface

UEFI

- Unified Extensible Firmware Interface
- Between and operating system and platform firmware
- Goal is repleace BIOS (from 2005)
- Flexible pre-OS environment (GUI, network, 32/64bit, large storege device, security, CPU-independency, compatibility to BIOS, etc.)
- Boot- and runtime services
- Cooperation with GPT (see later)



Storing data

Operative memory

Cache

Storage

Thought-provoking questions

- SRAM is more effective than DRAM (e.g., DDR). Why we use DRAM as mainmemory?
- What is cache? Why is it advantageous?
- How large disc space is allocated for a text file containing only one character?
- A large file is stored on disk as a set of small fragments. How can the operating system tell which fields belongs to which file?

Classification

- Writing capability
  - Read only, readable-writeable
- Place in hierarchy
  - Primary, secondary, tertiary
- Storage media
  - semiconductor, magnetic, optical
- Data preservation
  - Persistent, non-persistent
- Access
  - Serial access, direct access

Primary memory

- Actively used by processor and executed programs
- Fast access
- Non-persistent storage
- Limited size
- Types
  - Registers
  - Cache
  - Operative memory

Secondary memory

- Storage
- „Extension” of primary memory
- Large capacity
- Slow access
- Persistent storage
- Types
  - HDD, SSD
  - CD-, DVD-drive
  - Memory card

Tertiary memory

- Based on removable storage
- Robotic arm connects the storage media („juke-box”)
- Rare access
- Huge capacity
- Long-term backup
- Types
    - Magnetic tapes
    - Optical disks

Semiconductor memory

Read-Only Memory (ROM)

- Content is stored during production, later not changeable
- Content preserved for long time without voltage

Programmable Read-Only Memory (PROM)

- „Empty” after production. Content can be „burned” only once with special devices.
- Writing is slow (comparing to RAM)

Semiconductor memory

Erasable Programmable ROM (EPROM)

- ✦ Slowly erasable content with strong UV light then again writeable

Electrically Erasable Programmable ROM (EEPROM)

- ✦ Electronically, byte-by-byte erasable content

Flash memory

- ✦ Special improved EEPROM
- ✦ Quickly erasable in blocks (quickly, many times)

Semiconductor memory

Random Access Memory (RAM)

- ✦ Direct access
  - ➥ Access time independent of location
- ✦ Readable/writeable arbitrarily
- ✦ Lose content without voltage
- ✦ Fast reading/writing

Static Random Access Memory (SRAM)

- ✦ Uses flip-flops (pl. cache)

Semiconductor memory

Dynamic Random Access Memory (DRAM)

- ✦ Periodical update is necessary, because used capacitors lose voltage
- ✦ Slower than SRAM (E.g., operative memory)
- ✦ Response is as fast as possible

Synchronous Dynamic RAM (SDRAM)

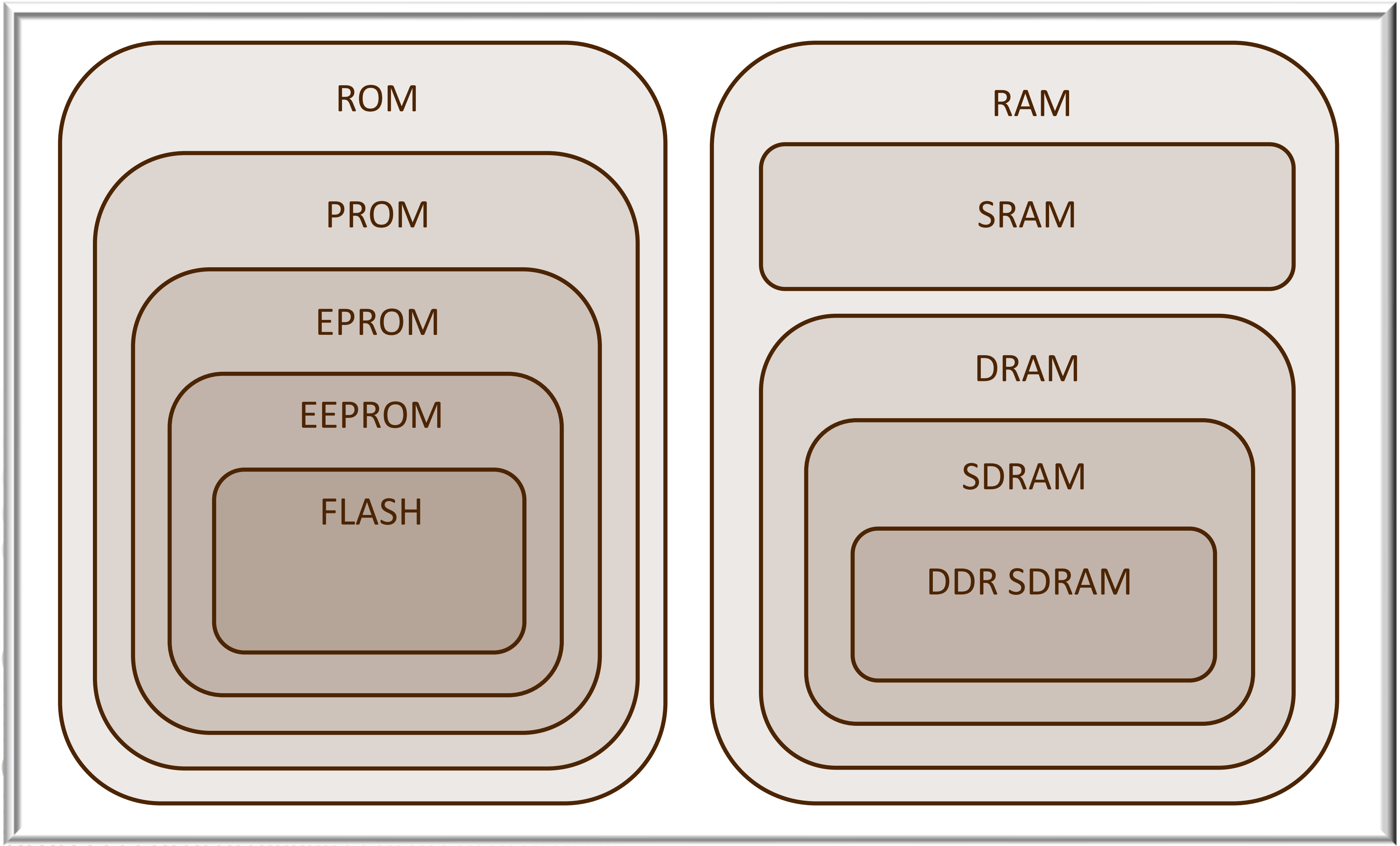
- ✦ Response is synchronized by the clock
- ✦ Faster than asynchronous DRAM (useful in pipeline)

Semiconductor memory

Double Data Rate (DDR)

- Speeded SDRAM
- Synchronized by rising- and falling edge as well
  DDR2, DDR3, DDR4, DDR5
  - Increasing speed
  - Increasing data density
  - Decreasing voltage
- Incompatible DIMM
  (SDRAM, DDR, DDR2, DDR3, DDR4)

Semiconductor memory



Cache

- Devices exchange data regularly
- Transfer speed of devices are very different
- Slow device „slows down” the faster
- E.g., CPU circa 10 times faster than RAM
- CPU cache, GPU cache, Web cache, DNS cache
- Solution idea
  - Intermediate storage (cache)
  - Faster than the slower device (SRAM)
  - Slow device not replaceable by faster due to price
  - Reason of existence: locality principle

Cache

Spatial locality principle

- ✦ If a program refers to a memory address (data or instruction), then probably soon refers to neighboring addresses as well
  - ➥ Sequential execution, arrays

Temporal locality principle

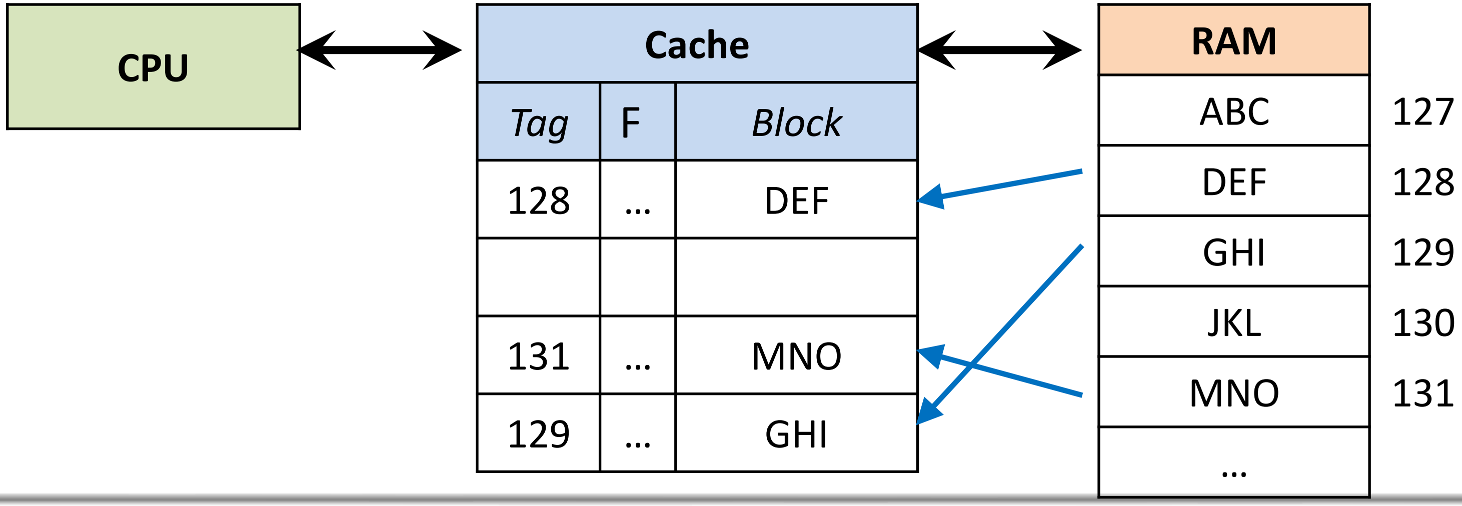
- ✦ If a program refers to a memory address, then probably soon refers to this address again
  - ➥ Loops

Cache

- Storing frequently used data
- Smaller capacity, faster access (SRAM) than the operative memory
- Transparent to programmers/users
- Can contain associative (CAM) memory
- CPU cache
  - Multiple level: L1, L2, L3 cache
  - On-chip or off-chip

Structure of cache

- Storage unit „**Line**” (or „Block”)
- Line is extended by „Tag” and „Flags”
- Block contains a copy of memory fragment
- Tag belongs to the memory address of Block



Cache		
Tag	F	*Block*
128	…	DEF
131	…	MNO
129	…	GHI

RAM	
ABC	127
DEF	128
GHI	129
JKL	130
MNO	131
…	

Operation of cache

- CPU look up necessary data in cache, giving the address to cache controller
- If the address is in a Tag (cache hit), then cache responses based on Block content
- If address is not stored in any Tag (cache miss), then cache reads the Block from RAM saving it (overwriting a line), then responses based on Block content

Operation of cache

Associativity: How many lines can be used to store a data from a RAM address?

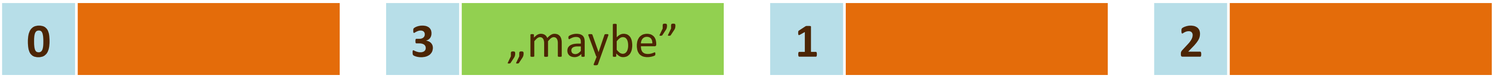
- Direct-mapped
  - From 1 RAM address to only 1 line.
- Fully associative
  - From any address to any cache line.
- Set-associative (N-way)
  - From 1 RAM address to one of a few lines.
  - Special cases: direct-mapped, fully associative

Operation of cache

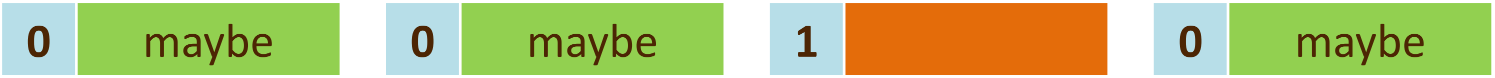
- Replacement policy in case of overwriting
  - Random
    Fast, but not effective

maybe
maybe
maybe
maybe

  - Least recently used (LRU)
    efficient, but complicated



  - Not most recently used
    Effective and simple



Operation of cache

- During writing take care of consistency of memory and cache
- Solutions
  - Wright through
  - Write back

Write through

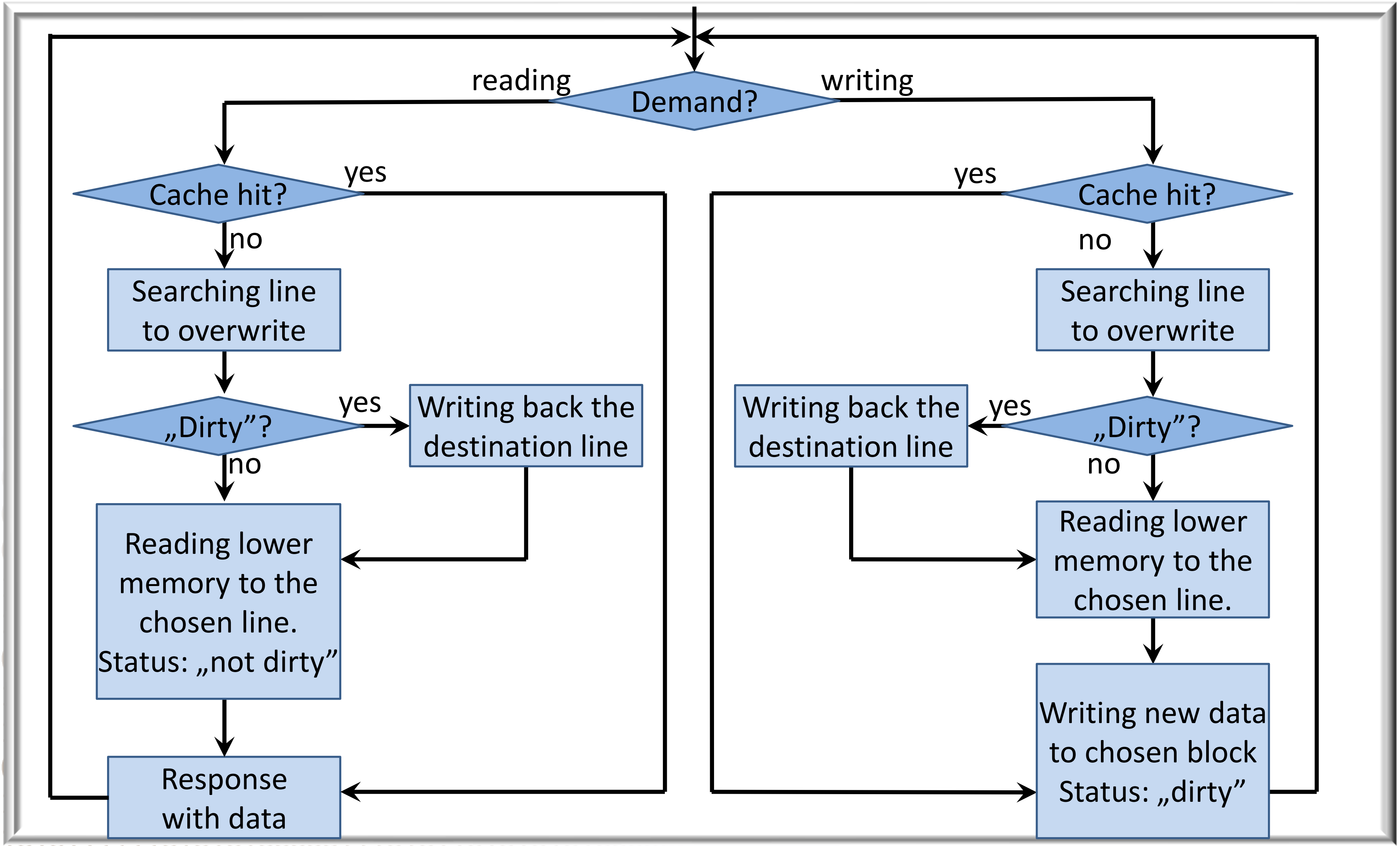
- Cache writing simultaneously to RAM
- Cache does not speed up writing

Operation of cache

Write back

- Cache updated after each write operation
- It is indicated by a „dirty” bit of Tag
- If necessary, to overwrite a „dirty” cache line, the line have to be written back to memory. After than the line can be updated with a new line.
- In multiprocessor system separate caches can lead to problems (inconsistency). There are problems with DMA as well.

Operation of cache



Demand?

reading

writing

Cache hit?

yes

no

Searching line to overwrite

„Dirty”?

yes

no

Writing back the destination line

Reading lower memory to the chosen line. Status: „not dirty”

Response with data

Cache hit?

yes

no

Searching line to overwrite

„Dirty”?

yes

no

Writing back the destination line

Reading lower memory to the chosen line.

Writing new data to chosen block Status: „dirty”

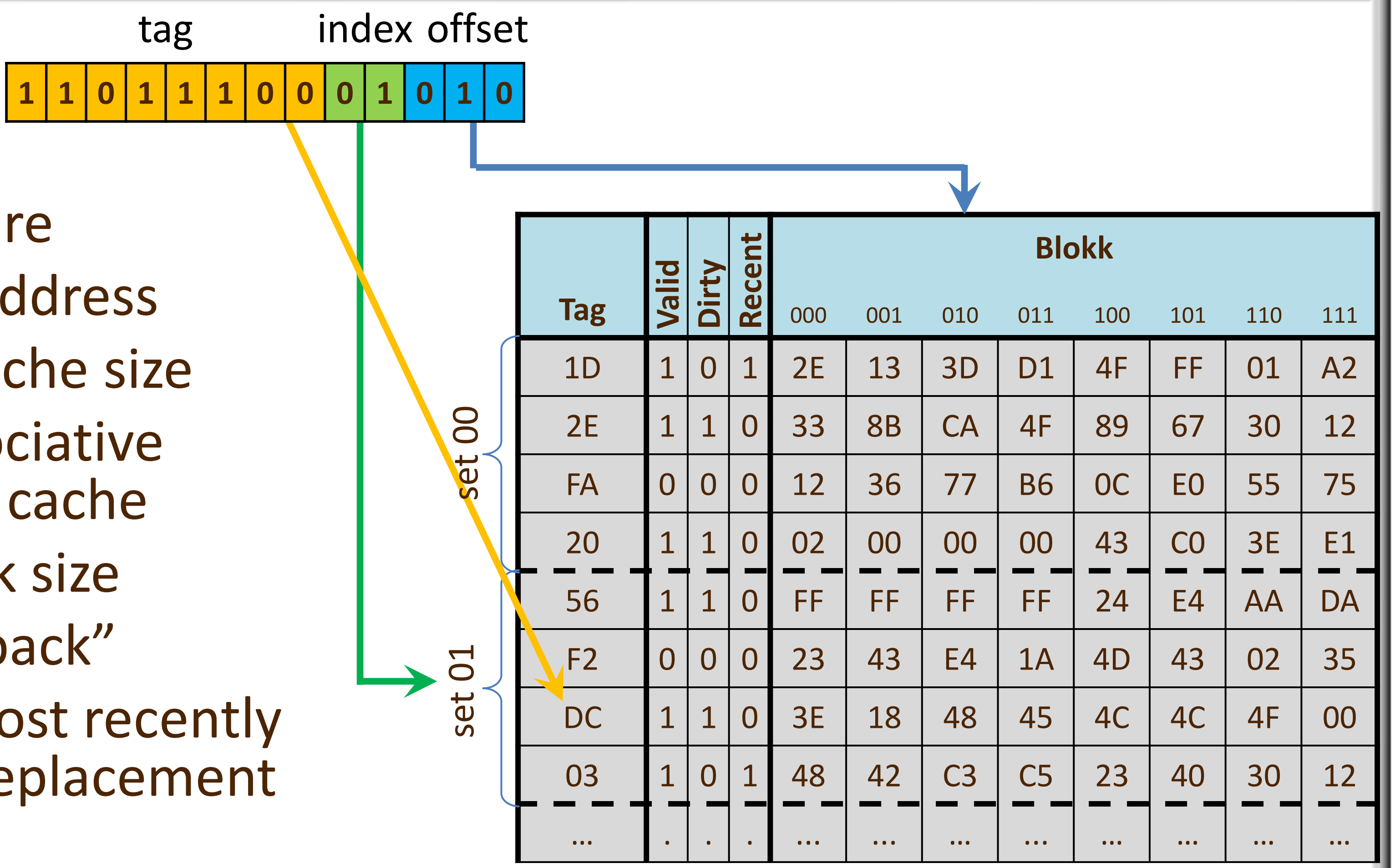
Characteristics of cache

- Size of cache
- Size of Block (E.g., 64 byte)
- Lookup time of a block
- Updating time (in case of write back)
- Replacement strategy (incase of line update)
- Hit rate
  - Generally, over 90%
  - Depending on size, replacement, etc.

Structure of addresses

- Address = Tag & Index & Offset
- Offset: position of the byte in the block
  - In case of 2^S size blocks it is S bit wide
- Index: identifies cache line sets
  - In case of 2^N sets it is N bit wide
- Tag: most significant bits of a RAM address
  - In case of L bit wide address, it is L-N-S bit wide
- Example: Pentium 4, 8kB 4-way L1 cache, 64B block
  - $\log_2(64)=$**6** offset bit; $\log_2(8192/64/4)=$**5** index bit; 32-5-6=**21** tag bit

Addressing example



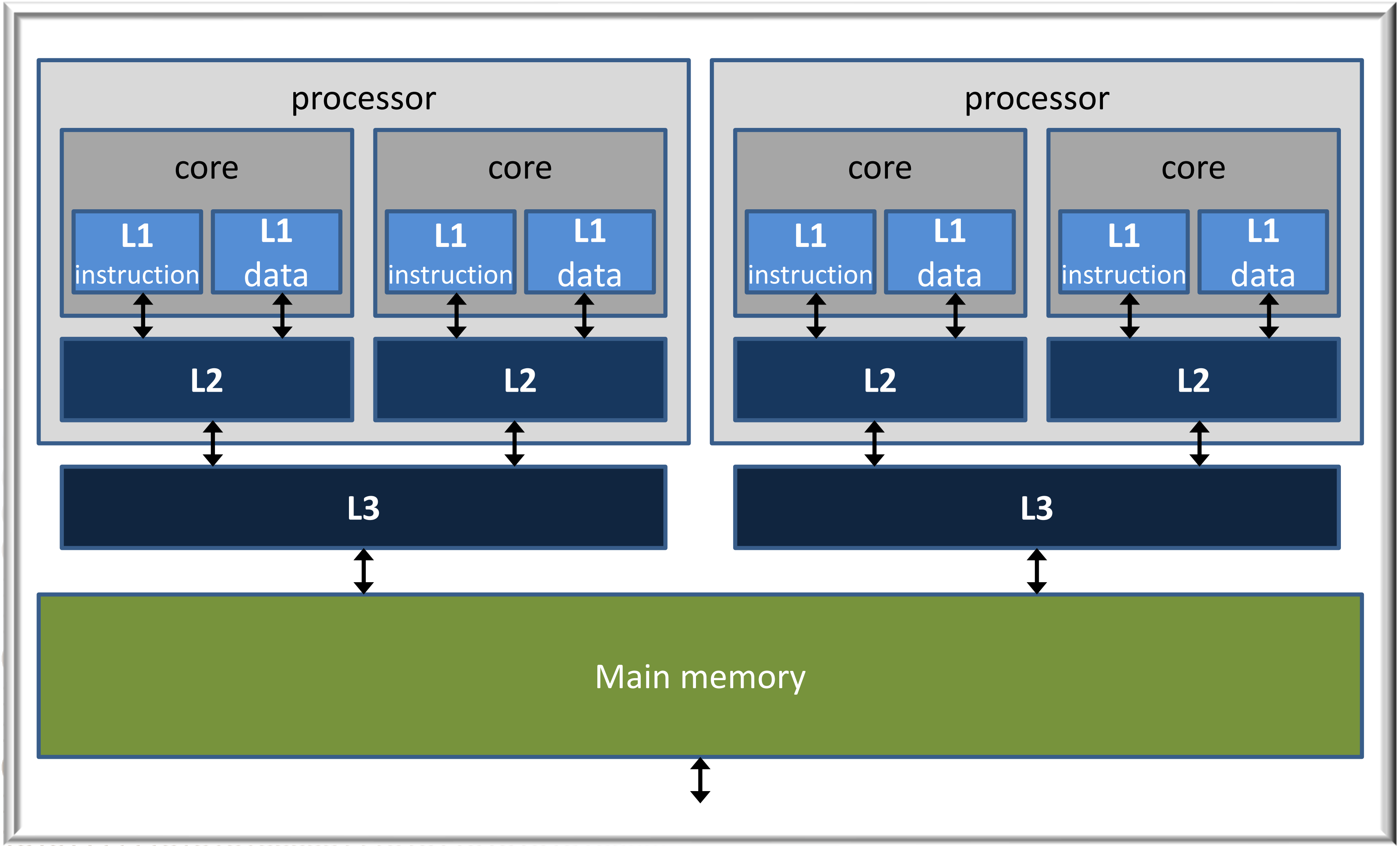
	Tag	Valid	Dirty	Recent	Blokk							
					000	001	010	011	100	101	110	111
set 00	1D	1	0	1	2E	13	3D	D1	4F	FF	01	A2
	2E	1	1	0	33	8B	CA	4F	89	67	30	12
	FA	0	0	0	12	36	77	B6	0C	E0	55	75
	20	1	1	0	02	00	00	00	43	C0	3E	E1
set 01	56	1	1	0	FF	FF	FF	FF	24	E4	AA	DA
	F2	0	0	0	23	43	E4	1A	4D	43	02	35
	DC	1	1	0	3E	18	48	45	4C	4C	4F	00
	03	1	0	1	48	42	C3	C5	23	40	30	12
	…	.	.	.	…	…	…	…	…	…	…	…

Architeture

- 13-bit address
- 128B cache size
- set associative (4-way) cache
- 8B block size
- „write-back"
- „ not most recently used" replacement

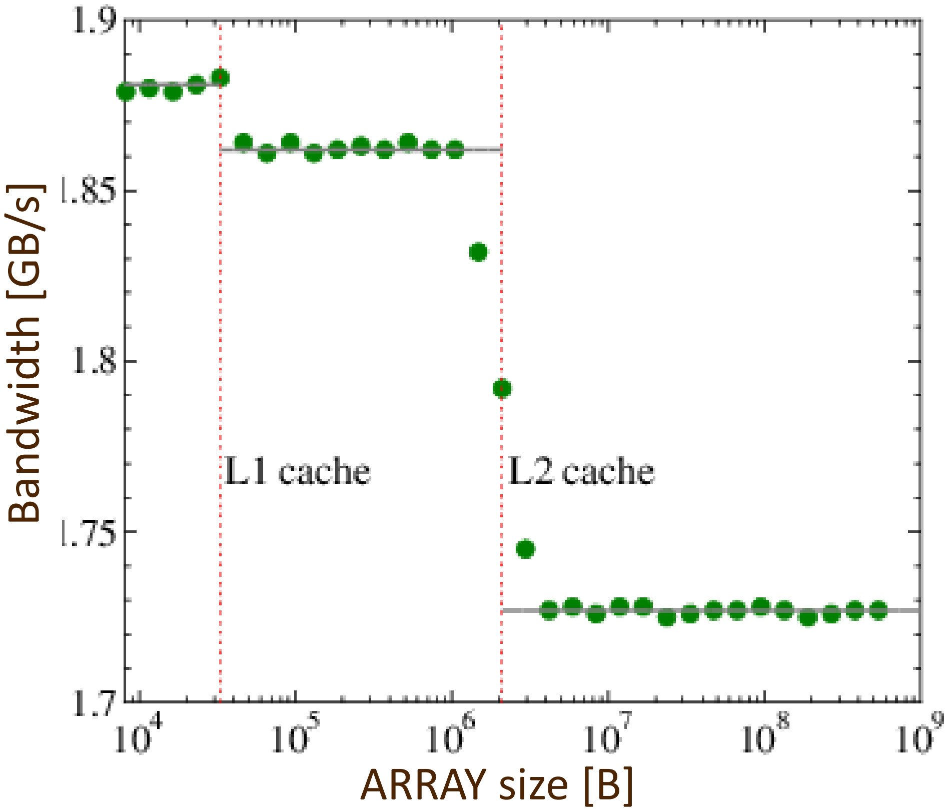
Cache hit? What is there? Overwritable? Need save?

Cache hierarchy



Effect of cache to programming

```
...
ARRAY = (double*)malloc(SIZE*sizeof(double));
N_Rep = 1000000000/SIZE;
for(j=0; j<N_Rep; j++)
   for(i=0; i<SIZE; i++)
      sum += ARRAY[i];
...
```



If the data/program fit into the cache, then the program will be faster!

Effect of cache to programming

- Row-mayor order matrix

```
int a[N][M];
```

Row-by-row:

```
for(i=0;i<N;i++)
   for(j=0;j<M;j++)
      sum+=a[i][j];
```

Column-by-column:

```
for(j=0;j<M;j++)
   for(i=0;i<N;i++)
      sum+=a[i][j];
```

In case of huge matrix:

- Frequent cache miss
- Slow run

Virtual memory

- Transparent extension of the limits of physical memory with the help of storage device
- Physical memory is divided into **frames**
- Logical memory divided into **pages** (size: 4kB)
- Separate physical and virtual addressing
  - All processes have own virtual address space
- Virtual addresses are mapped into physical due to the **page table**
- Unused pages are on HDD/SDD
- Memory Management Unit (MMU) governs

Virtual memory

- Referring a page actually not in the RAM cause page fault
  - ➥ Swapping pages is necessary
- The page swap strategy is important (FIFO, LRU, …)
- Access time of HDD is ca. 100 000 times larger than access of RAM
  - ➥ Frequent swap slows down the program
- Implementation
  - ➥ Linux: swap partition
  - ➥ Windows: C:\pagefile.sys file

Page table

- Content: which page is in which frame
  - plus „valid” and „shared” bits
- Stored in the memory
- Starting address in Page Table Base Reg. (PTBR)
- **Address translation** also requires RAM access
  - Problem of 2 memory access
- Translation Lookaside Buffer (TLB)
  - Special cache for page table in MMU
- Page table can be multilevel hierarchically

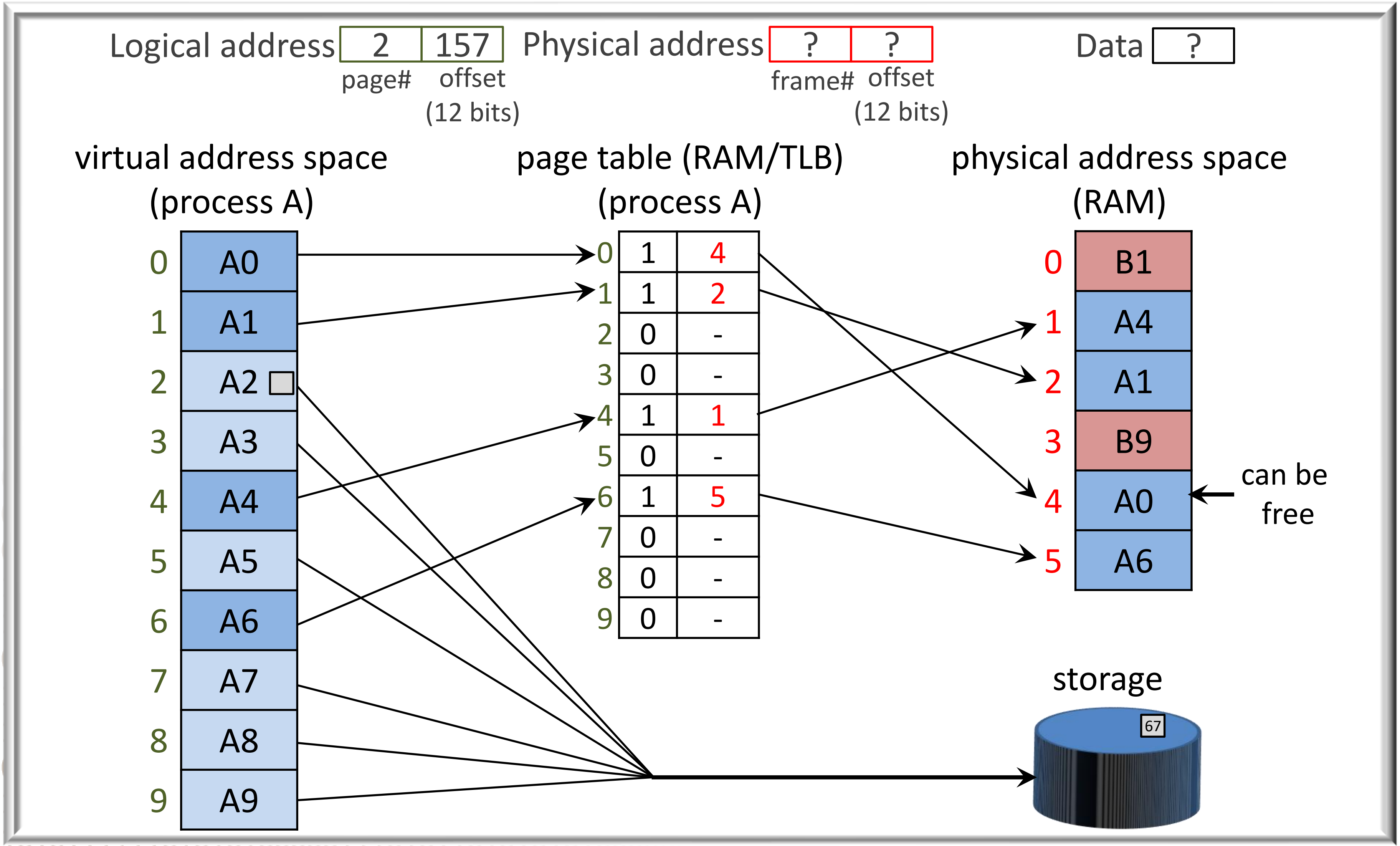
Virtual memory handling

1. CPU demand to logical memory (page# + offset)
2. Checking TLB: If no proper entry in TLB go to step 3, else go to step 10
3. Checking page table: If the page is in RAM go to step 10, else go to step 4
4. Chose a frame in RAM as destination of read
5. In case of need, start the swap out of page in frame to HDD, then start loading needed page into the free frame (by the help of DMA)

Virtual memory handling

6. During this the process is in „waiting” state, scheduled launches another process
7. In the background DMA swaps in/out pages between RAM and HDD and sends an „I/O ready” IRQ, if it is ready
8. Process get into „ready” state
9. When scheduler resumes the process repeat the memory demand
10. Based on the page table, the physical address is ready (frame# + offset), RAM sends back the data

Virtual memory handling



Logical address 2 | 157 (page#, offset (12 bits))

Physical address ? | ? (frame#, offset (12 bits))

Data ?

virtual address space (process A)

0	A0
1	A1
2	A2 □
3	A3
4	A4
5	A5
6	A6
7	A7
8	A8
9	A9

page table (RAM/TLB) (process A)

0	1	4
1	1	2
2	0	-
3	0	-
4	1	1
5	0	-
6	1	5
7	0	-
8	0	-
9	0	-

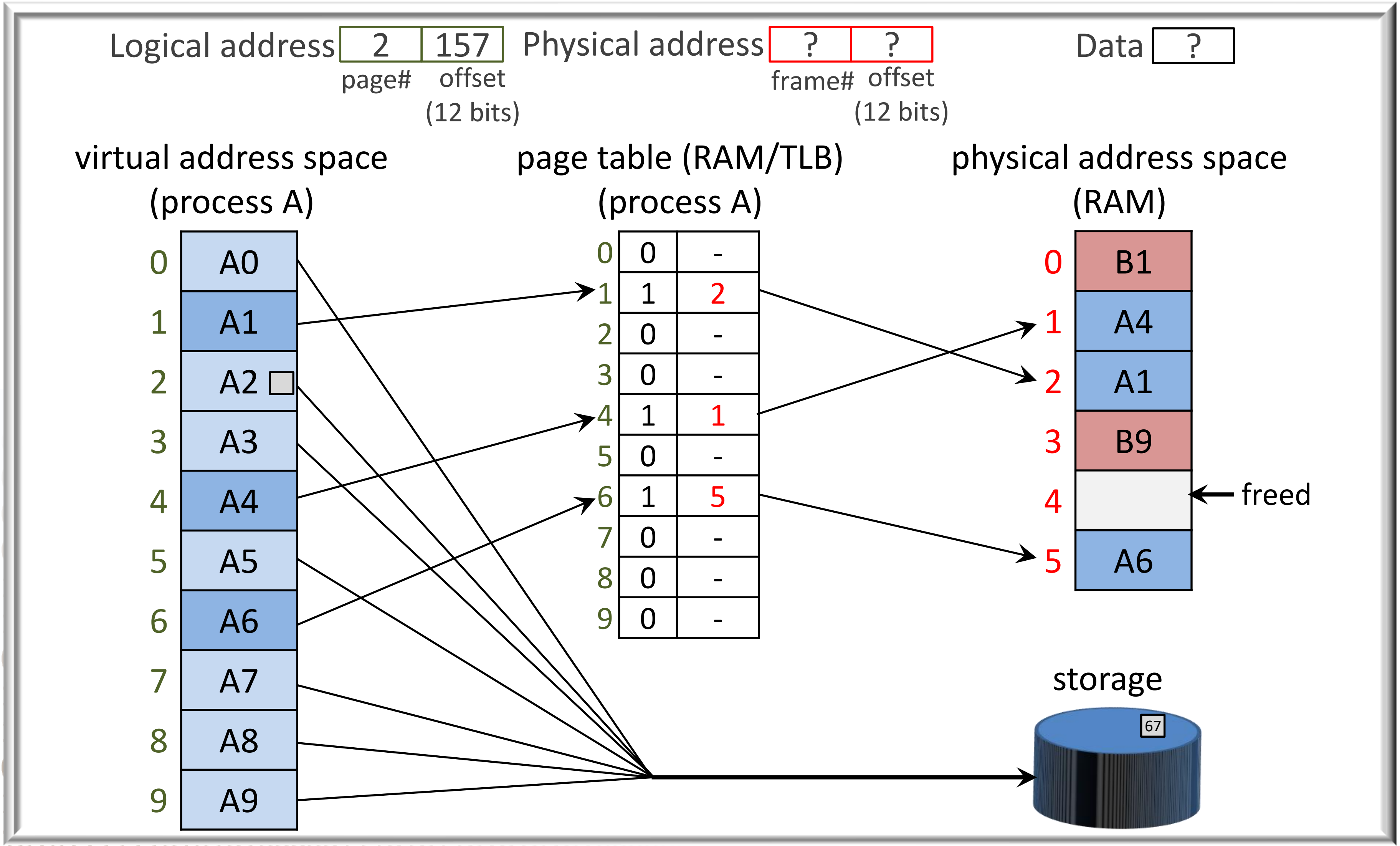
physical address space (RAM)

0	B1
1	A4
2	A1
3	B9
4	A0
5	A6

can be free

storage

Virtual memory handling



Logical address | 2 | 157
page# offset (12 bits)

Physical address | ? | ?
frame# offset (12 bits)

Data | ?

virtual address space (process A)

0	A0
1	A1
2	A2
3	A3
4	A4
5	A5
6	A6
7	A7
8	A8
9	A9

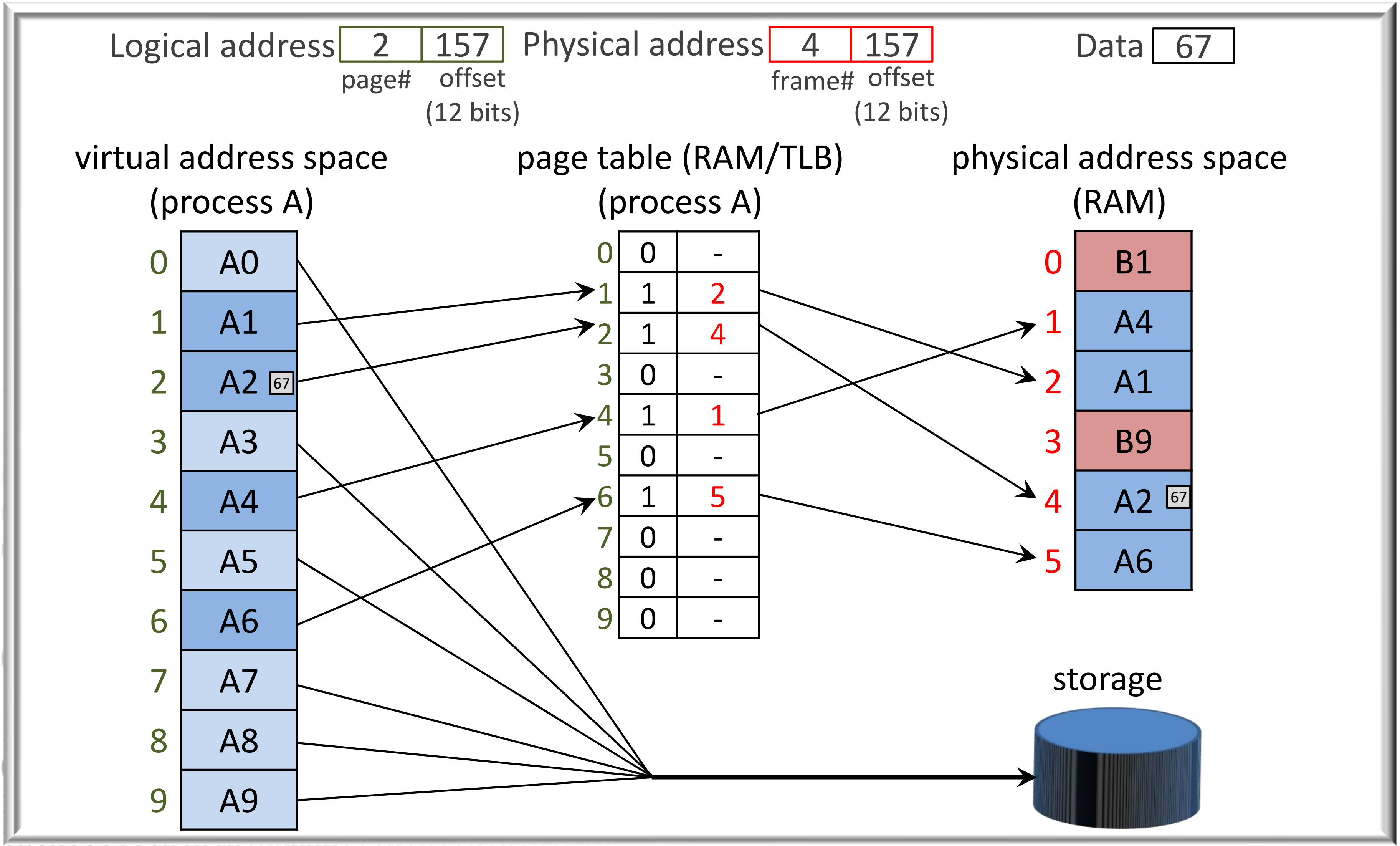
page table (RAM/TLB) (process A)

0	0	-
1	1	2
2	0	-
3	0	-
4	1	1
5	0	-
6	1	5
7	0	-
8	0	-
9	0	-

physical address space (RAM)

0	B1
1	A4
2	A1
3	B9
4	
5	A6

Virtual memory handling



Logical address 2 | 157 (page#, offset (12 bits))

Physical address 4 | 157 (frame#, offset (12 bits))

Data 67

virtual address space (process A)

0	A0
1	A1
2	A2 67
3	A3
4	A4
5	A5
6	A6
7	A7
8	A8
9	A9

page table (RAM/TLB) (process A)

0	0	-
1	1	2
2	1	4
3	0	-
4	1	1
5	0	-
6	1	5
7	0	-
8	0	-
9	0	-

physical address space (RAM)

0	B1
1	A4
2	A1
3	B9
4	A2 67
5	A6

storage

Advantages for programmers

- ✦ Ability to use more data in a program than the available RAM size
- ✦ All programs/processes have own virtual address space without access to memory content of others (see also shared bit)
- ✦ When we write a program, not necessary to know which memory areas are occupied (by others)
- ✦ It supports multi-programmed systems

Storage

Problems

- Operative memory is not enough large
- RAM content is volatile (no power, no data)

Solution

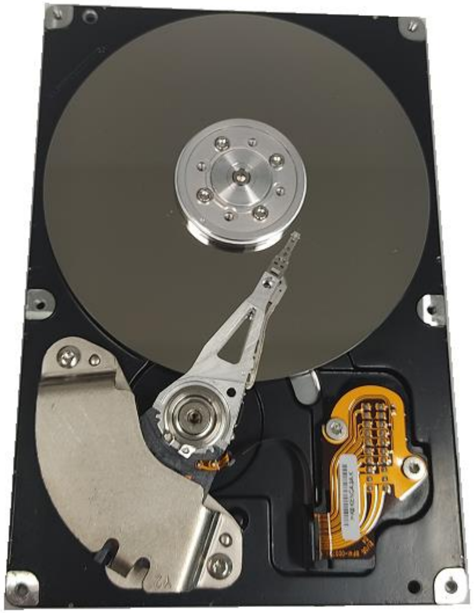
- Storage: slow access, but large capacity

Storage mechanism

- Magnetic: **Hard Disc Drive** (HDD)
- Electronic: **Solid-State Drive** (SSD)
- Optical: CD, DVD, Blu-ray

Hard Disc Drive

- Winchester
- Storing data on magnetic rotating discs
- 2 read/write head per disc
- Properties
  - Storage capacity: <18TB
  - Read/write speed: <200MB/s
  - Rotational speed: 5400 – 15000 rpm
  - Connector: ATA, SATA, SCSI, USB
  - Disc cache size: 8MB – 512MB



Opened HDD

Storing data on HDD

- HDD can contain several **platters**
- Platter build up concentric **tracks**
- Tracks with same radius on platters under each other calls **cylinder**
- Tracks are divided into **sectors**
  - Smallest unit of storage
- **Clusters** are group of sectors
- All read/write **heads** moves together
  - At a given moment all head are in a cylinder

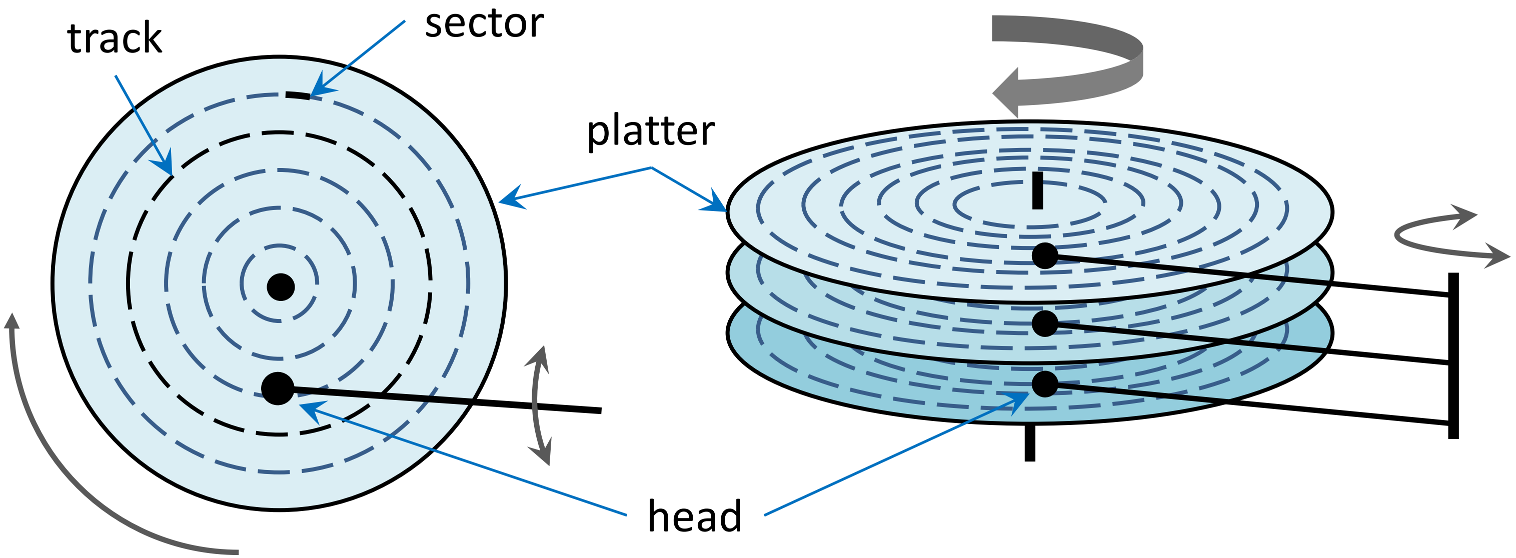
Storing data on HDD

- Average access time

$$T_{access} = T_{seek} + T_{rotation} + T_{other}$$

$$T_{access} \approx T_{seek} + T_{rotation} \; (\approx 10\, ms)$$

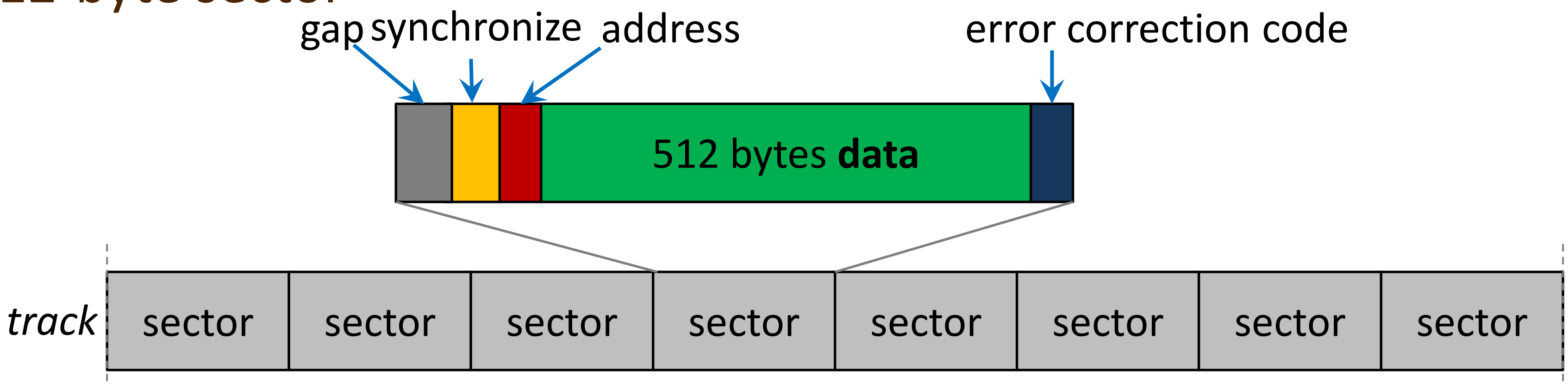
➥ First byte (in sector) is „expensive”, rest is „cheap”



Sector

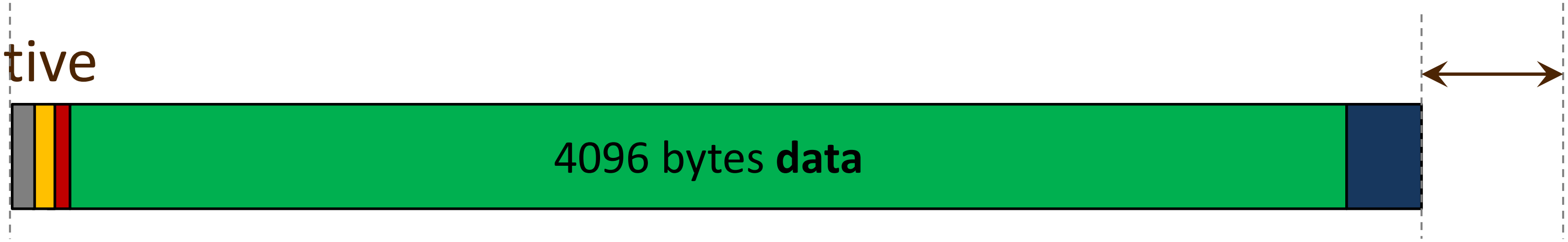
Traditional structure

512-byte sector



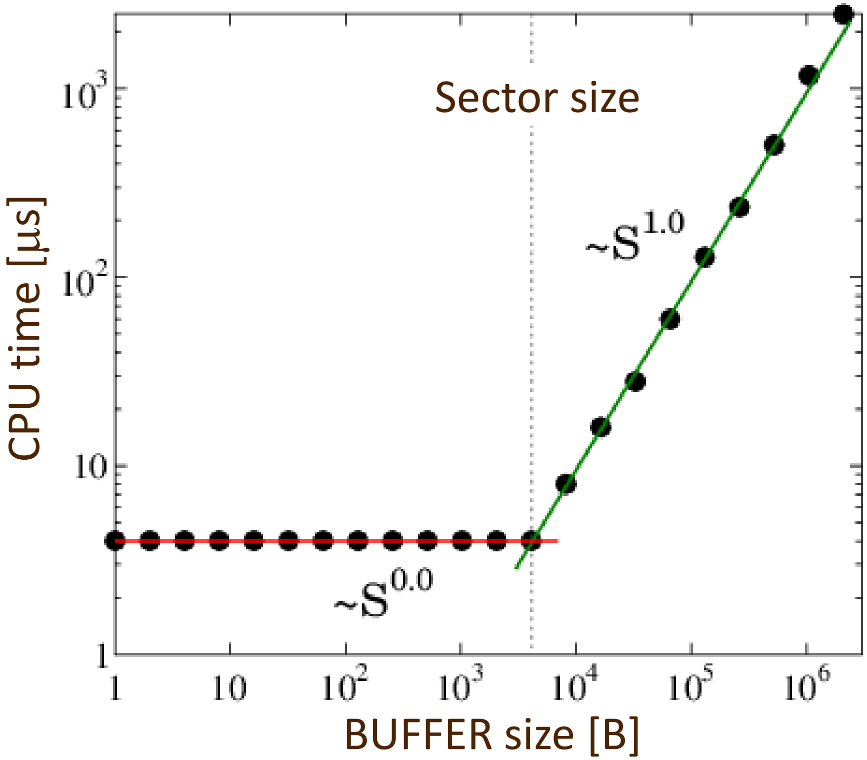
Advanced Format (AF) (2010)

4K native



Effect of sectors to programming

```
...
#define SIZE 2048
char BUFFER[SIZE];
out=open("output.txt",O_WRONLY);
write(out, BUFFER, SIZE);
close(out);
...
```

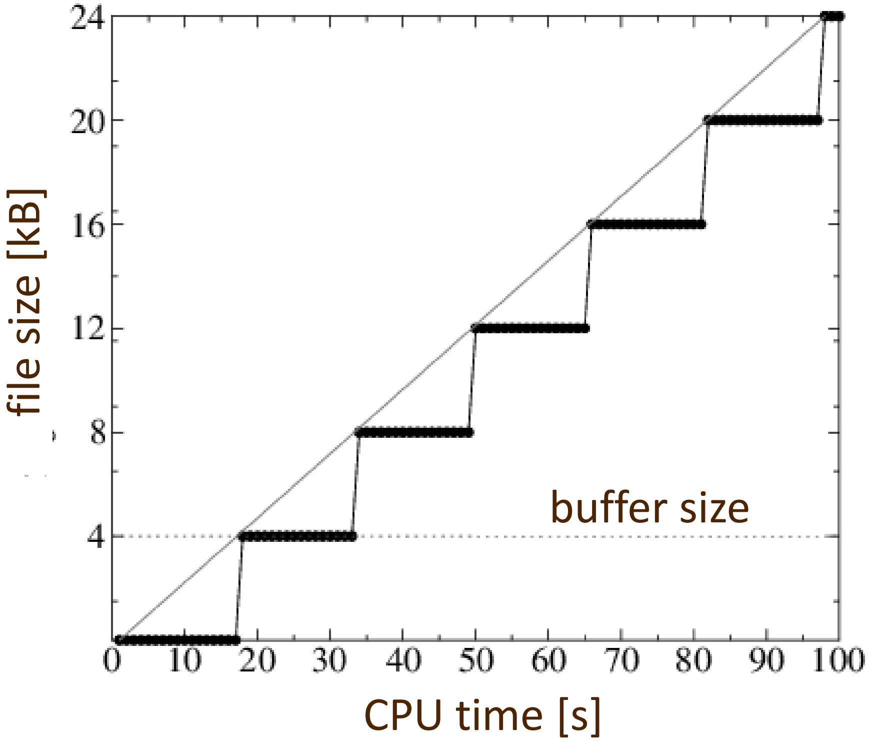


Writing smaller than sector is not effective!

Buffered data stream

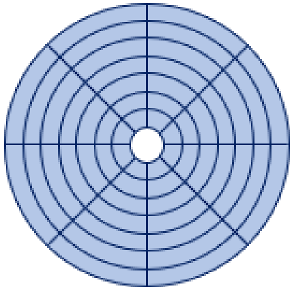
- The system often buffering output stream and writing data later in blocks.
- Less system call, less disc operation, faster program.

```
…
while(i<100000){
   fprintf(f,”x”);
   sleep(t);
   i++;
   }
…
```



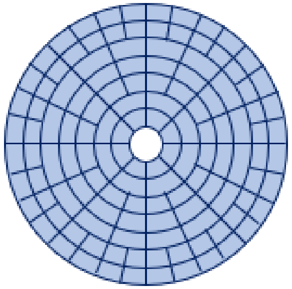
Number of sectors in tracks

Old-fashioned HDD



- ✦ All tracks have same number of sectors
- ✦ Data density in outside track is lower, than inside (low efficiency)

Modern HDD



- ✦ Zone Bit Recording (ZBR)
- ✦ Data density is similar anywhere
- ✦ More sector in outside tracks than inside
- ✦ Complex addressing

Data access on HDD

How to refer to a data on HDD?

- CHS
  cylinder-head-sector triplet (old)
- LBA
  logical block addressing (linear, new)
- Conversion (in simple case):
  $LBA = (C*N_{head}+H)*N_{sector}+(S-1)$
- Disc controller maps logical and physical addresses

SSD

- Solid-State Drive
- Semiconductor memory without mobile parts
- Data access in blocks
- SATA, SCSI, USB connection
- Advantage comparing with HDD
  - Fast data access ($\approx 0.1\, ms$)
  - Uniform access for all data
  - Less noise, consumption and heat
  - Mechanical reliability

Storage in high level

- Formatting
  - Low level (creation of sectors)
  - High level (creation of file system)
- Partitioning
  - Breaking up HDD into small logical units
  - MBR, GPT formation
- File systems
  - FAT32, NTFS
  - ext2, ext3, ext4

MBR

Master Boot Record

- LBA 0 (first sector)
  - Boot loader code (E.g., GRUB) + Partition table
- Max 4 partition with max 2 TB size
  - 4 primary position
  - 3 primary partition + 1 extended partition (with possible logical partitions)
- BIOS launches the loader
  - After the Power On Self Test

GPT

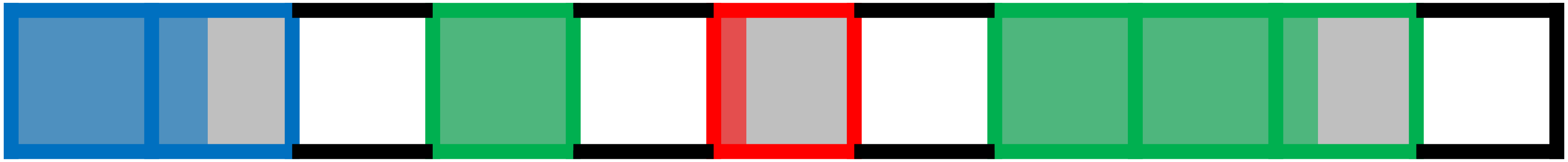
GUID Partition Table

- LBA 0: MBR preservation;
- LBA 1: primary GPT header
- LBA 2-33: primary GPT partition table
- Last sector: safety spare
  - Secondary GPT header and partition table
- UEFI Standard (replacing BIOS)
- Even 256 partitions, max disc size 75.6 ZB

Storing files on HDD

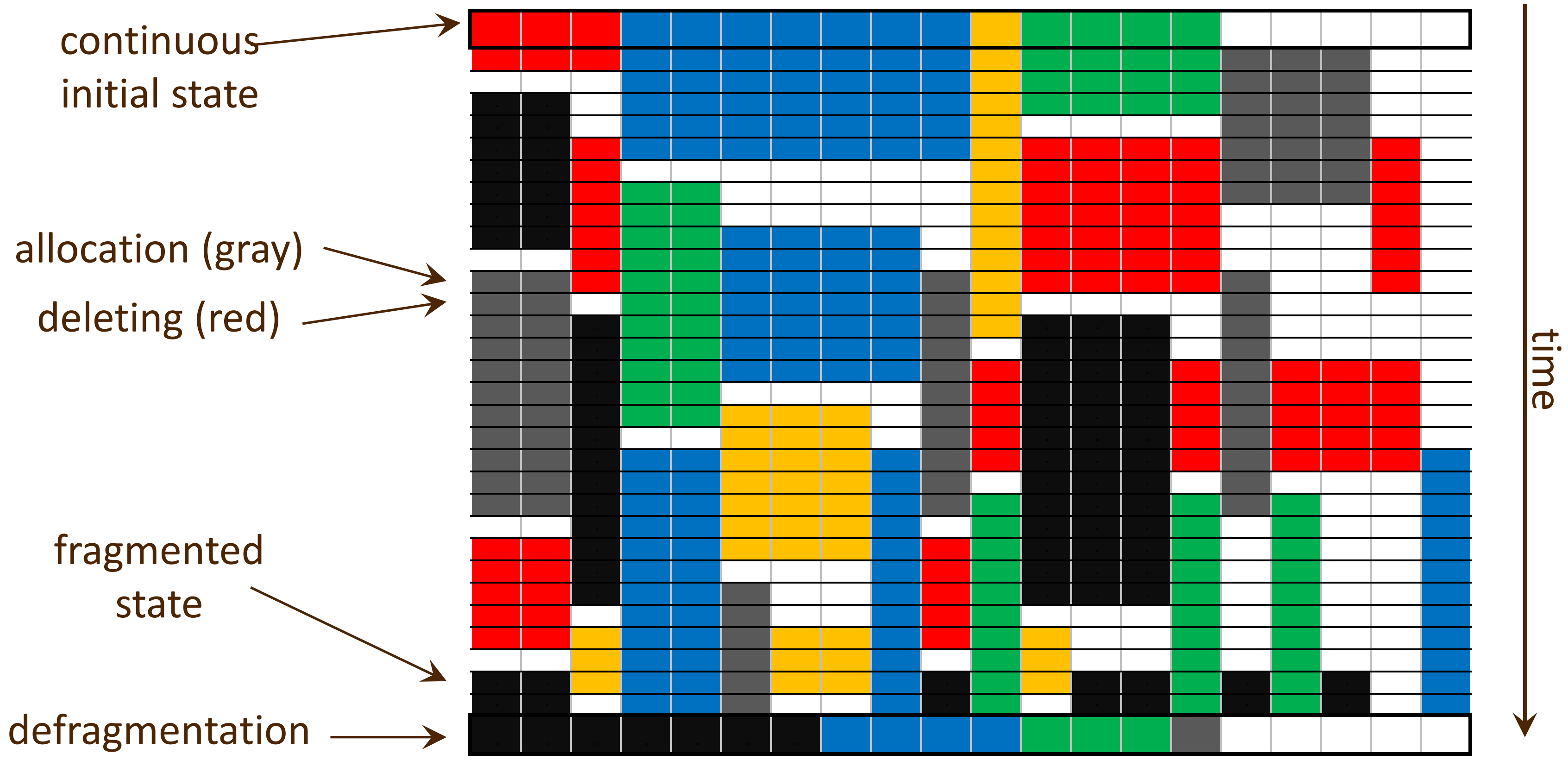
- Smallest logical (software) access unit
  - Block (1 or more continuous sectors)
- A file can be stored in more blocks
  - Sometimes not continuously
- A block can belong to only one file

Blue file:	Red file:	Green file:	Total:
Space: 2.0 block	Space: 1.0 block	Space: 4.0 block	**Space: 7.0 block**
Size: 1.4 block	Size: 0.2 block	Size: 3.3 block	**Size: 4.9 block**

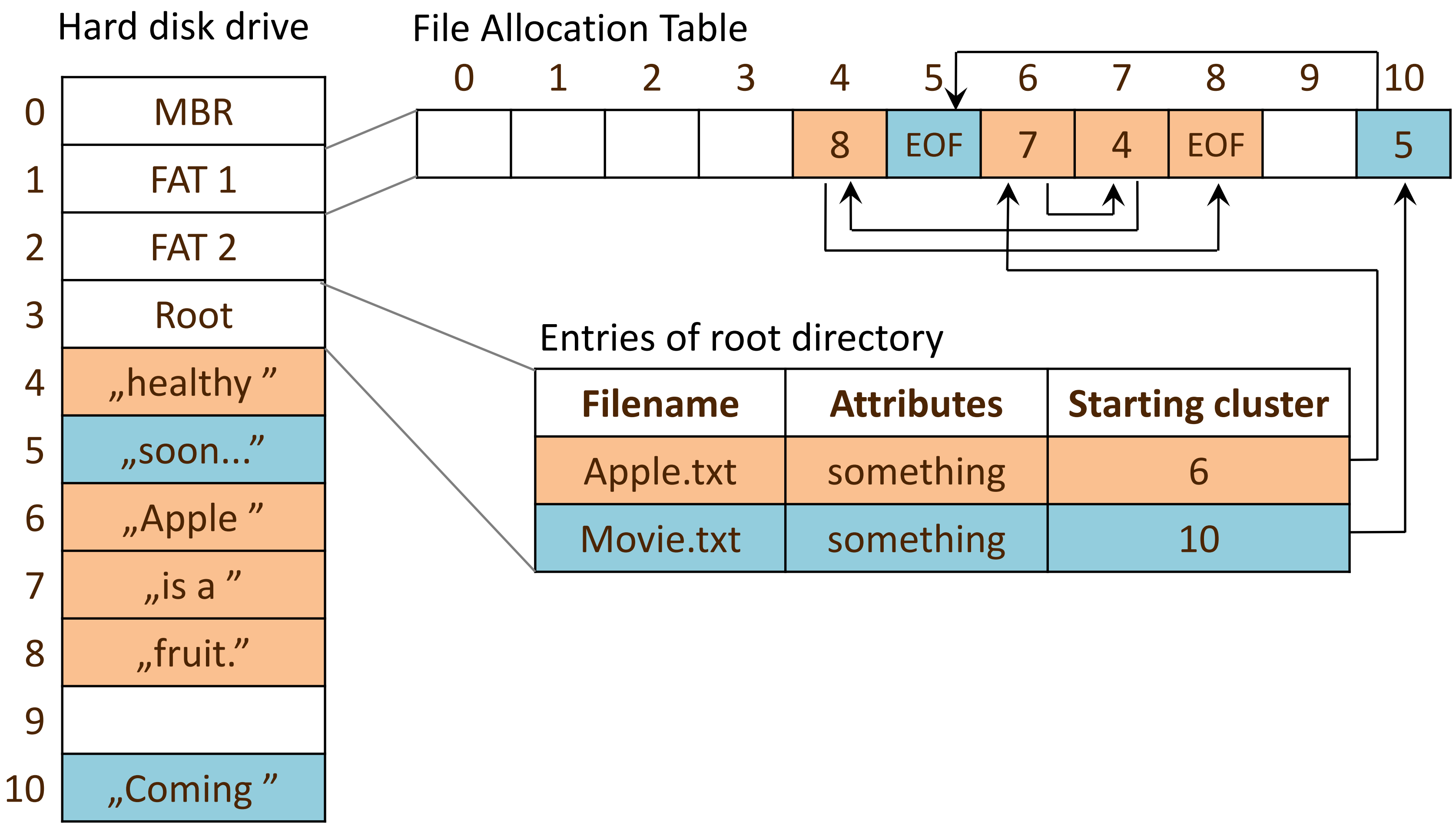


Disk fragmentation

- Fragmented structure due to allocations/deletes
- Slow file access due to lots of head positioning



FAT file system



Hard disk drive

0	MBR
1	FAT 1
2	FAT 2
3	Root
4	„healthy ”
5	„soon...”
6	„Apple ”
7	„is a ”
8	„fruit.”
9	
10	„Coming ”

File Allocation Table

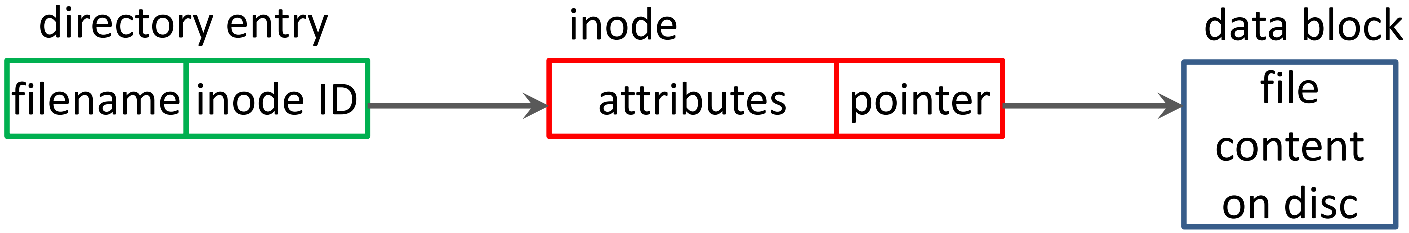
0	1	2	3	4	5	6	7	8	9	10
				8	EOF	7	4	EOF		5

Entries of root directory

Filename	Attributes	Starting cluster
Apple.txt	something	6
Movie.txt	something	10

Linux file system

- E.g.: Ext2, Ext3, Ext4
- File storage



directory entry

filename | inode ID

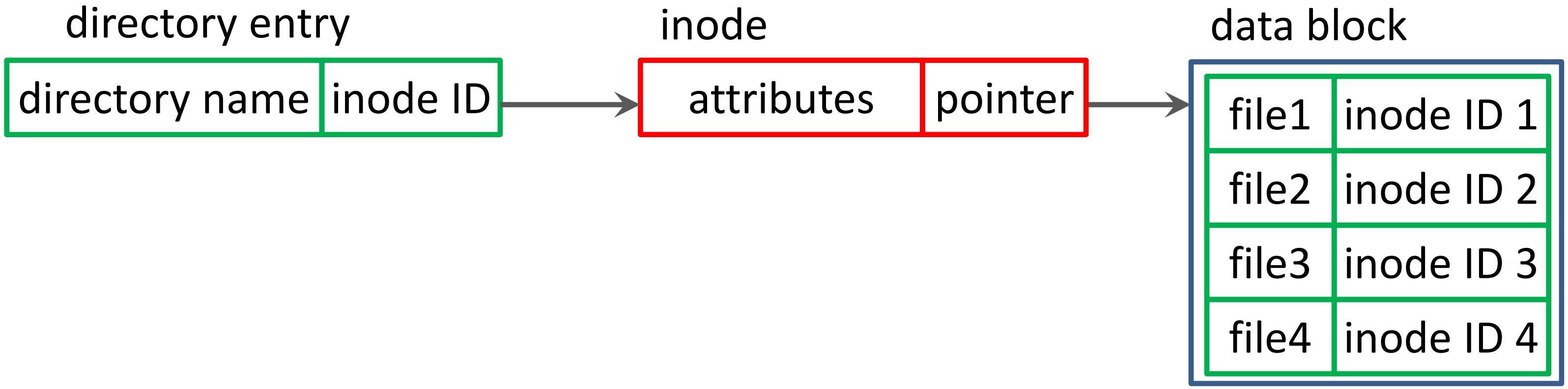
inode

attributes | pointer

data block

file content on disc

- Directory storage



directory entry

directory name | inode ID

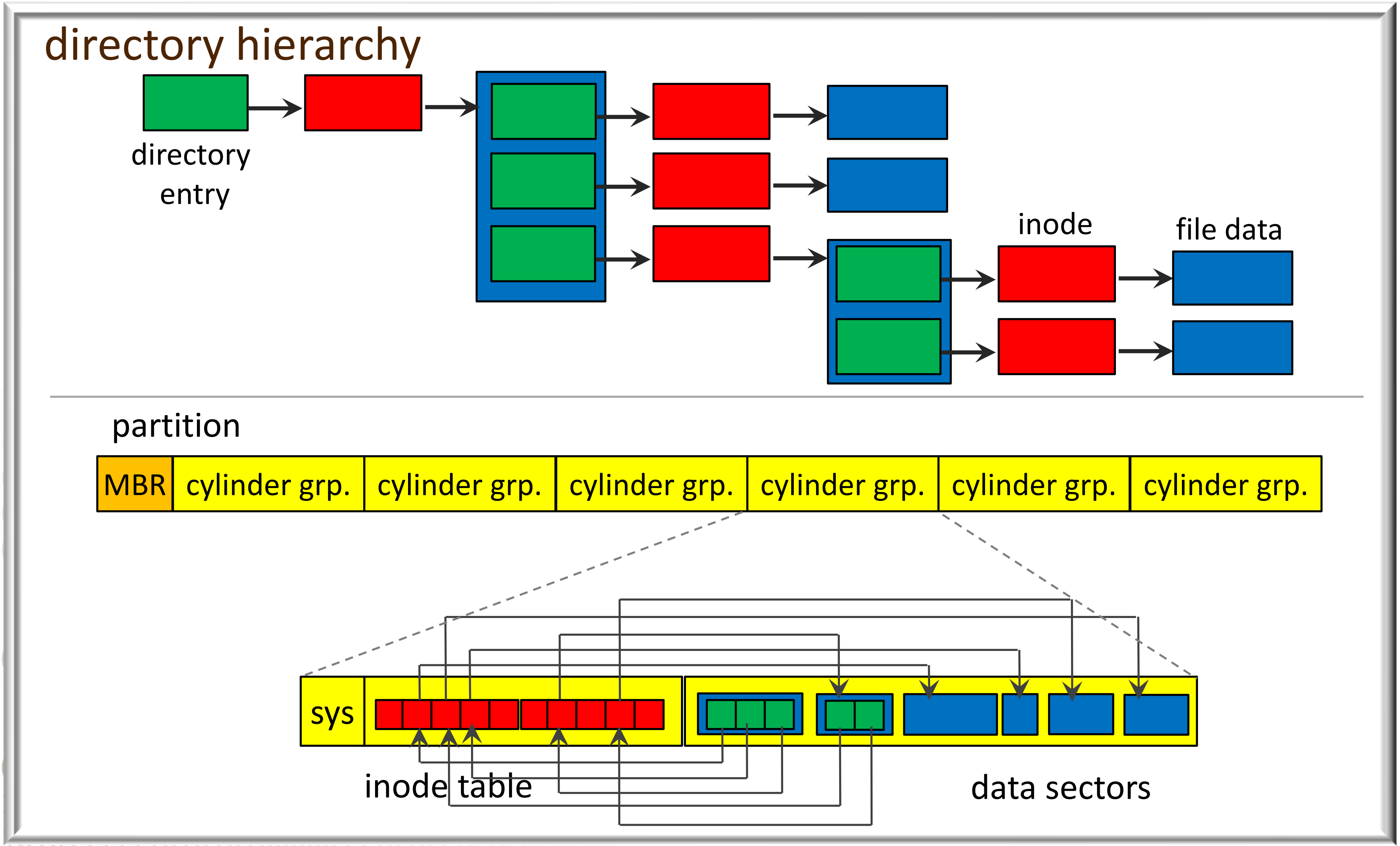
inode

attributes | pointer

data block

file1	inode ID 1
file2	inode ID 2
file3	inode ID 3
file4	inode ID 4

Linux file system

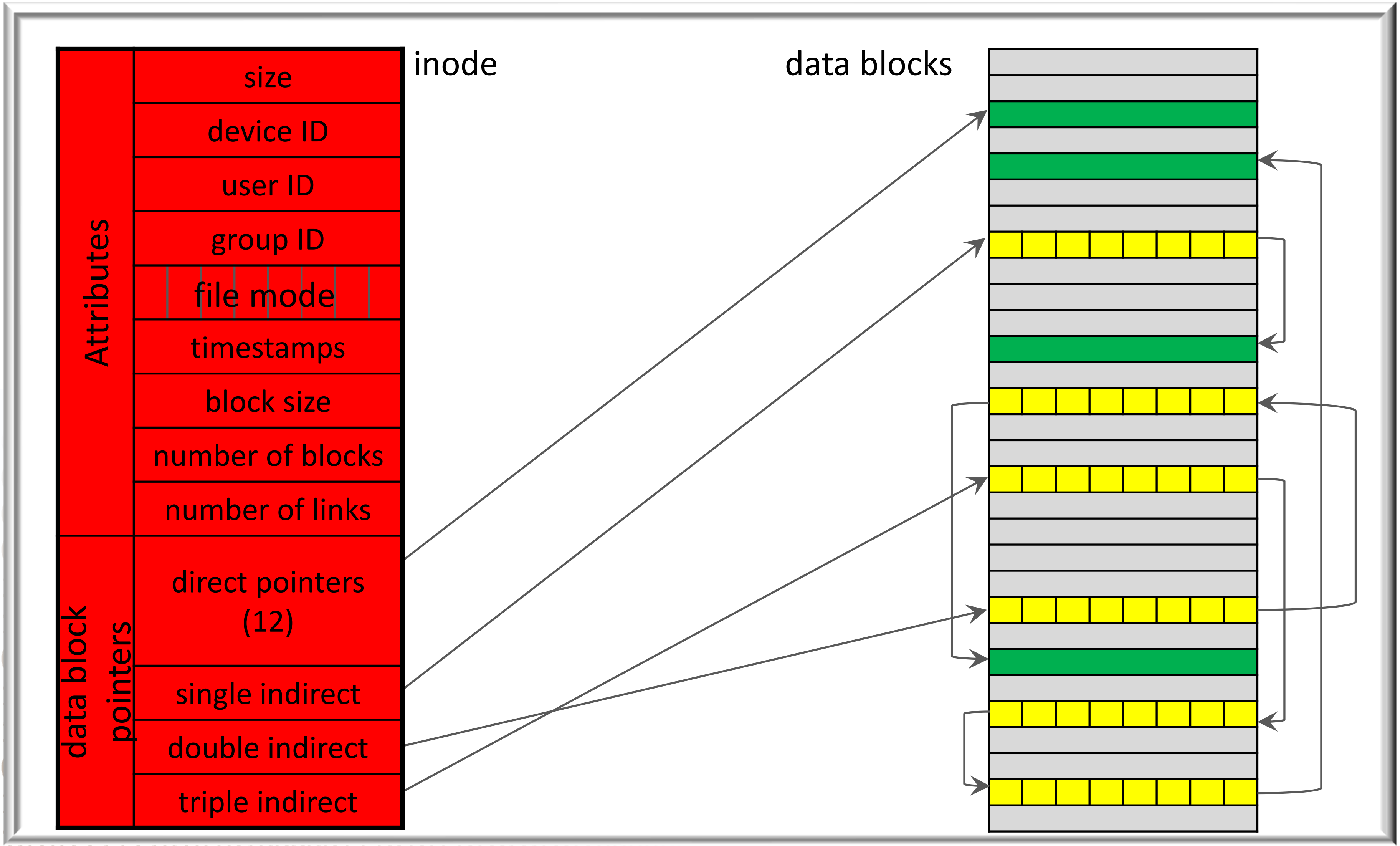


directory hierarchy

partition

MBR	cylinder grp.	cylinder grp.	cylinder grp.	cylinder grp.	cylinder grp.	cylinder grp.

inode



inode
data blocks
Attributes
size
device ID
user ID
group ID
file mode
timestamps
block size
number of blocks
number of links
data block pointers
direct pointers (12)
single indirect
double indirect
triple indirect

Example calculations for limits

Block size (BS): 4096B

Block-address width (AW): 4B

Number of pointers per block (PpB=BS/AS): 1024

#blocks referred by direct pointers: 12

#blocks referred by single indirect pointers (PpB): 1024

#blocks referred by double indirect pointers (PpB^2): 1048576

#blocks referred by triple indirect pointers (PpB^3): 1073741824

Maximal file size ($MFS=BS*PbB^3$): **4398046511104 (4TB)**

#blocks can be addressed ($NB=2^{AW*8}$): 4294967296

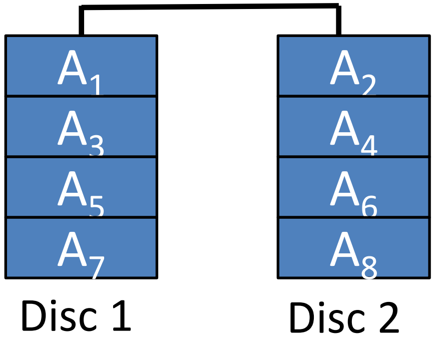
Maximal file system size (NB*BS): **17592186044416 (16TB)**

RAID

- Redundant Array of Independent Discs
- Data distribution and replication on several discs
- Goal of replication
  - Increasing reliability
  - Increasing transmission speed
- More „level” (RAID 0-6)
  - Even hierarchically (RAID 1+0)
- Software and hardware support

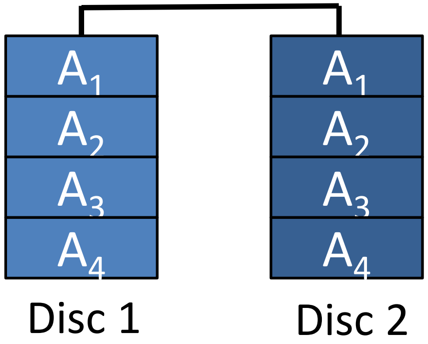
RAID examples

RAID 0 (striping)



Physical capacity: 2 unit
Logical capacity: 2 unit
Speed (R/W): 2/2 unit
Error protection: no

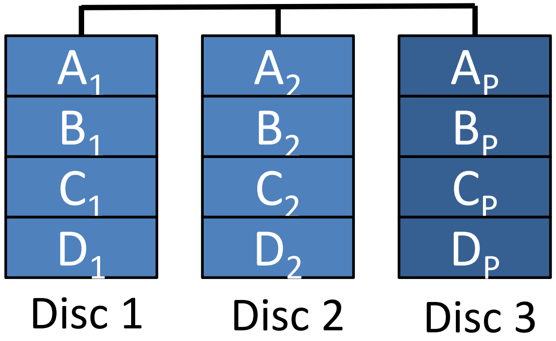
RAID 1 (mirroring)



Physical capacity: 2 unit
Logical capacity: 1 unit
Speed (R/W): 2/1 unit
Error protection: yes

RAID examples

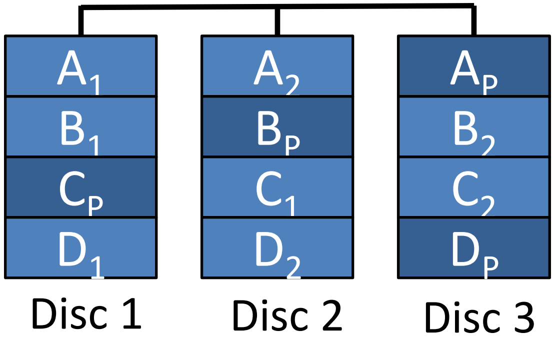
RAID 4 (parity disc)



Disc 1	Disc 2	Disc 3
A_1	A_2	A_P
B_1	B_2	B_P
C_1	C_2	C_P
D_1	D_2	D_P

Physical capacity: 3 unit
Logical capacity: 2 unit
Speed (R/W): 2/1 unit
Error protection: yes (slow)

RAID 5 (rotating parity)

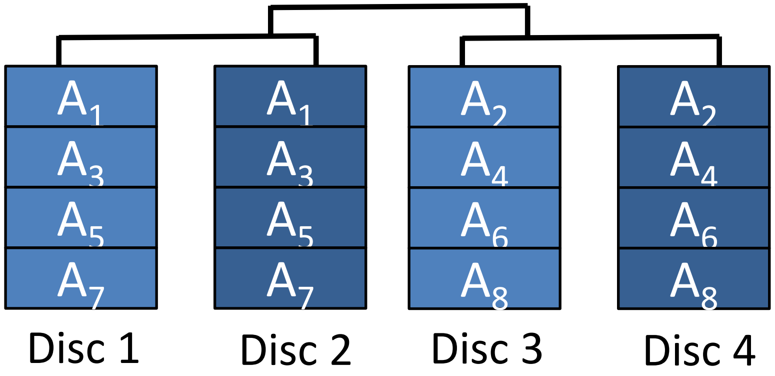


Disc 1	Disc 2	Disc 3
A_1	A_2	A_P
B_1	B_P	B_2
C_P	C_1	C_2
D_1	D_2	D_P

Physical capacity: 3 unit
Logical capacity: 2 unit
Speed (R/W): 2/1 unit
Error protection: yes (fast)

RAID examples

RAID 1+0 (mirroring + striping)



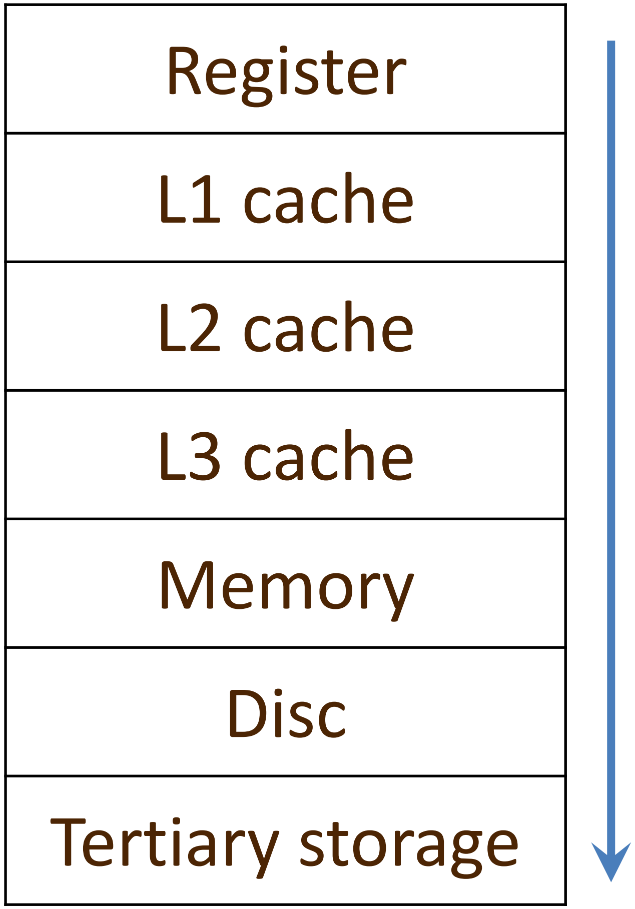
Physical capacity: 4 unit

Logical capacity: 2 unit

Speed (R/W): 2/2 unit

Error protection: yes

Memory hierarchy



Register

L1 cache

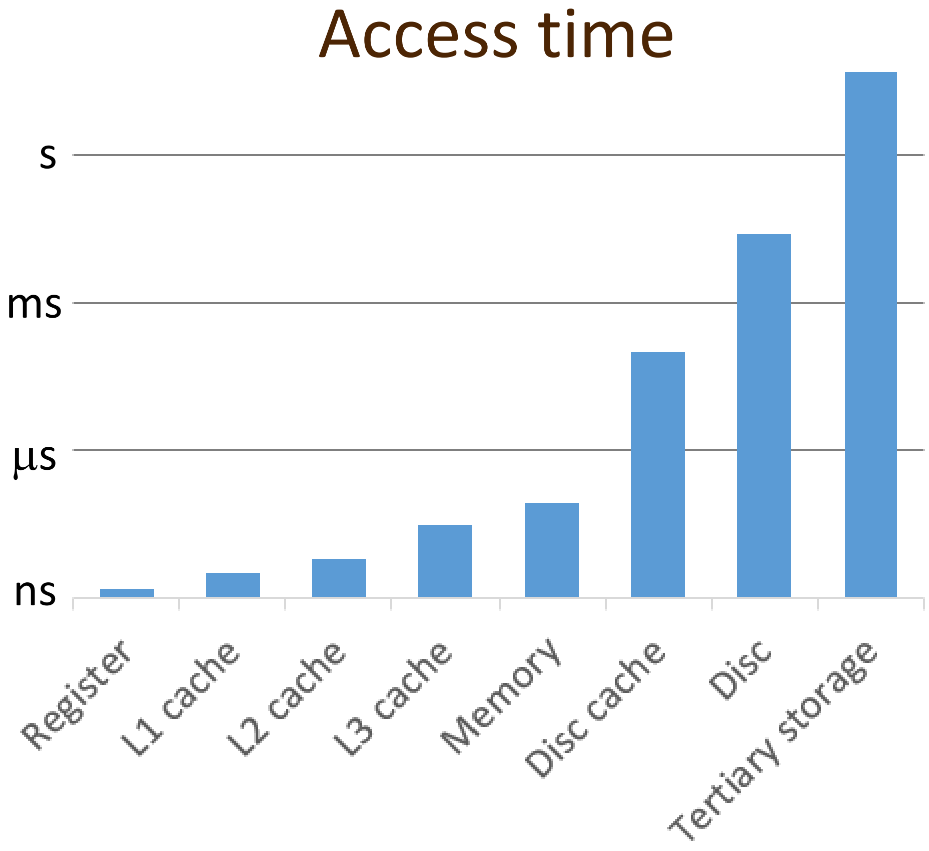
L2 cache

L3 cache

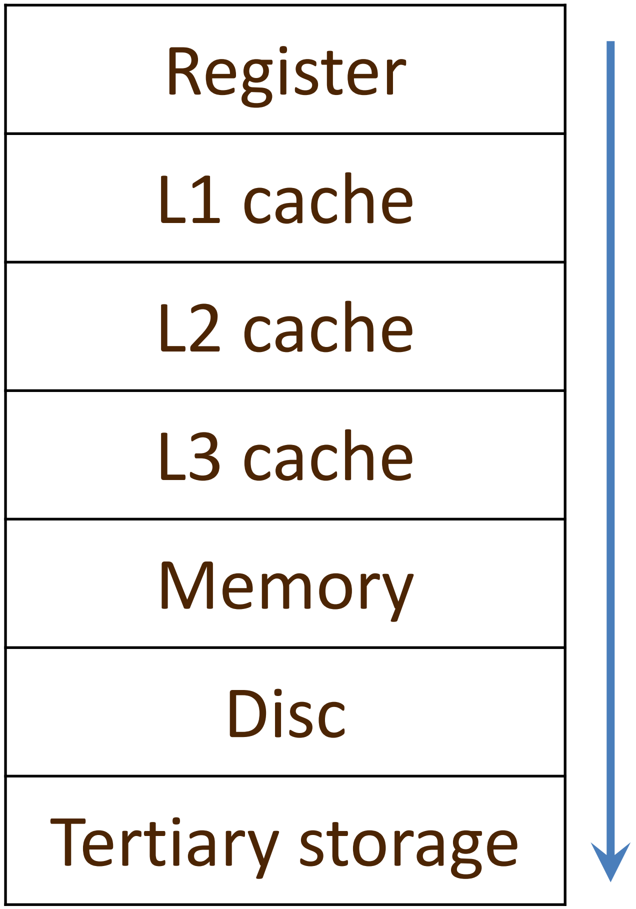
Memory

Disc

Tertiary storage



Memory hierarchy



Register

L1 cache

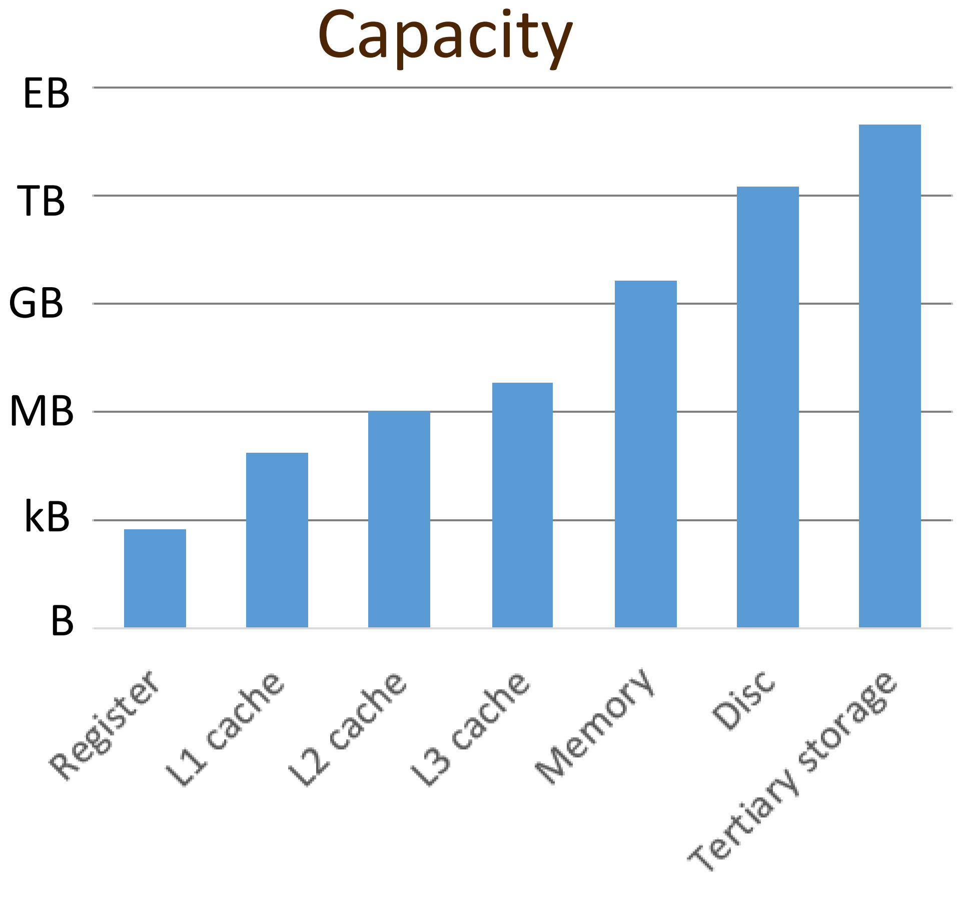
L2 cache

L3 cache

Memory

Disc

Tertiary storage



The Intel x86 and the ARM architectures

Structure of the processor

Register set

Memory handling

Assembly language

Thought-provoking questions

- Nowadays many kind of processors can be used in our personal computers.
  Can all of them be programmed in different ways in assembly level?
- What kind of processors are in our smart devices?
- What is necessary to know if we want to write programs for them? What kind of instruction set architecture are used?

Beginnings of x86 architecture

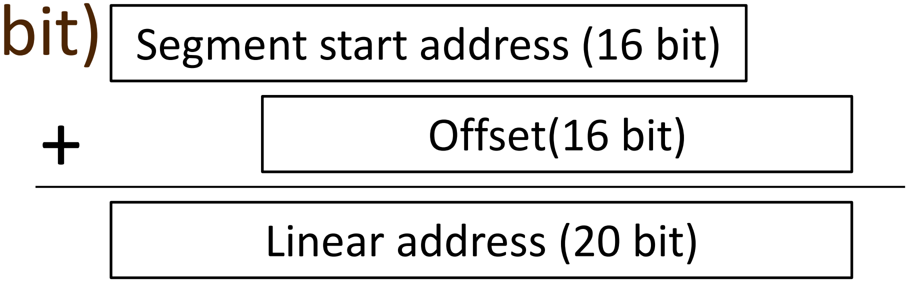
- Intel developed a „new” CISC processor between 1976-78 called **Intel 8086**
- Later it was improved
  - Intel 80186 (1982)
  - Intel 80286 (1982)
  - Intel 80386 (1986)
  - Intel 80486 (1989)
  - …, still go on
  - New processors are backward compatible
- The processor family is referred as **x86**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

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

Memory segmentation

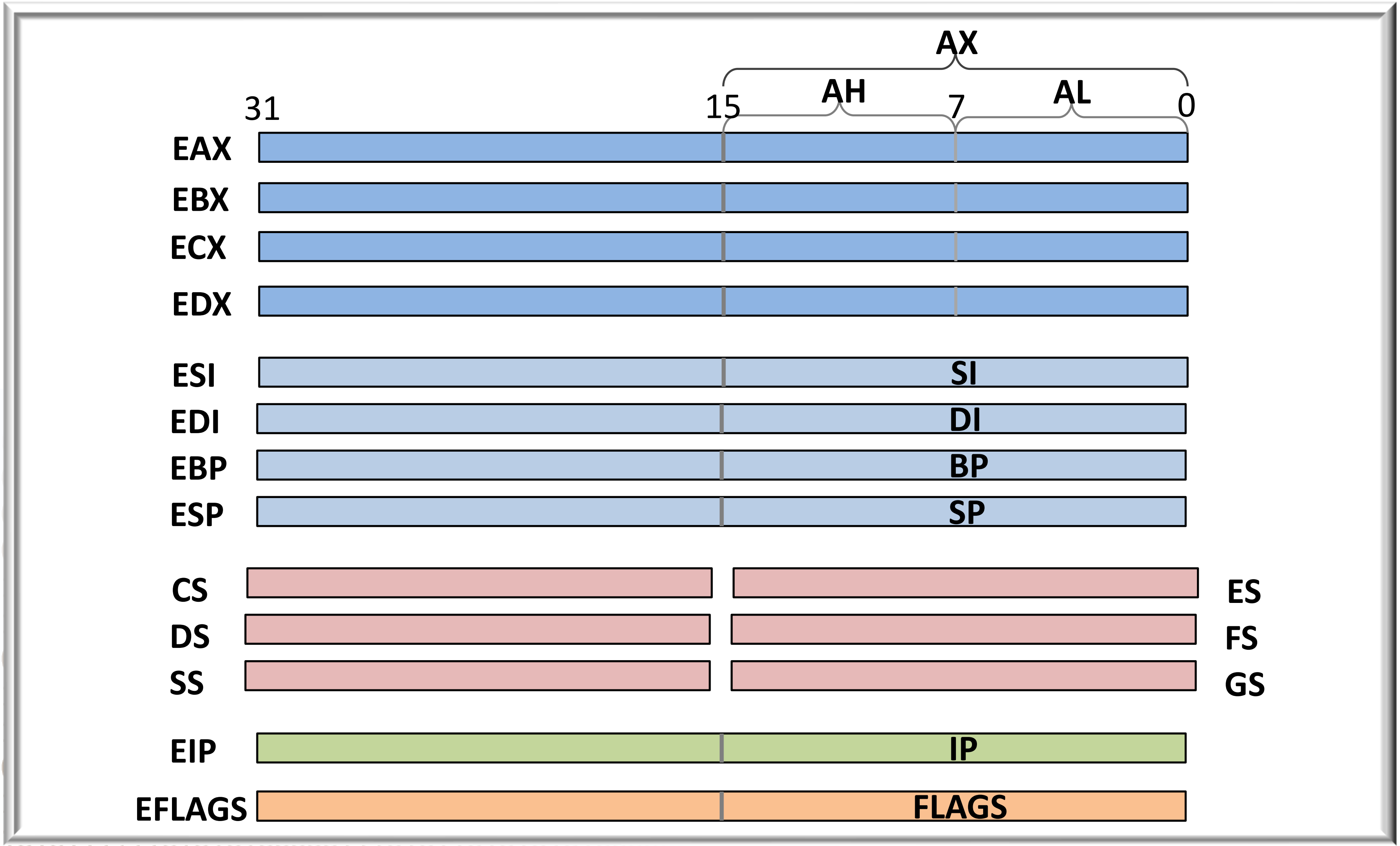
- Real mode
  - In 8086 the only one
  - 20 bits address bus (1MB)
  - All memory address available without restriction
  - The processor starts in this mode
  - Linear address = segment address*16 + offset
  - Linear address = Physical address
  - Segment size 64kB (16 bit)



Memory segmentation

- ✦ Protected mode
  - ➥ Introduced by 80386
  - ➥ Limited memory access
  - ➥ 32-bit address space (4GB)
  - ➥ Virtual memory support
  - ➥ Linear address (32) = segment address + offset
  - ➥ Linear address → paging → physical address
- ✦ Virtual mode (quasi real mode from 80386)
- ✦ Long mode (64 bit, no segmentation)

x86 register set



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

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

x86 register set

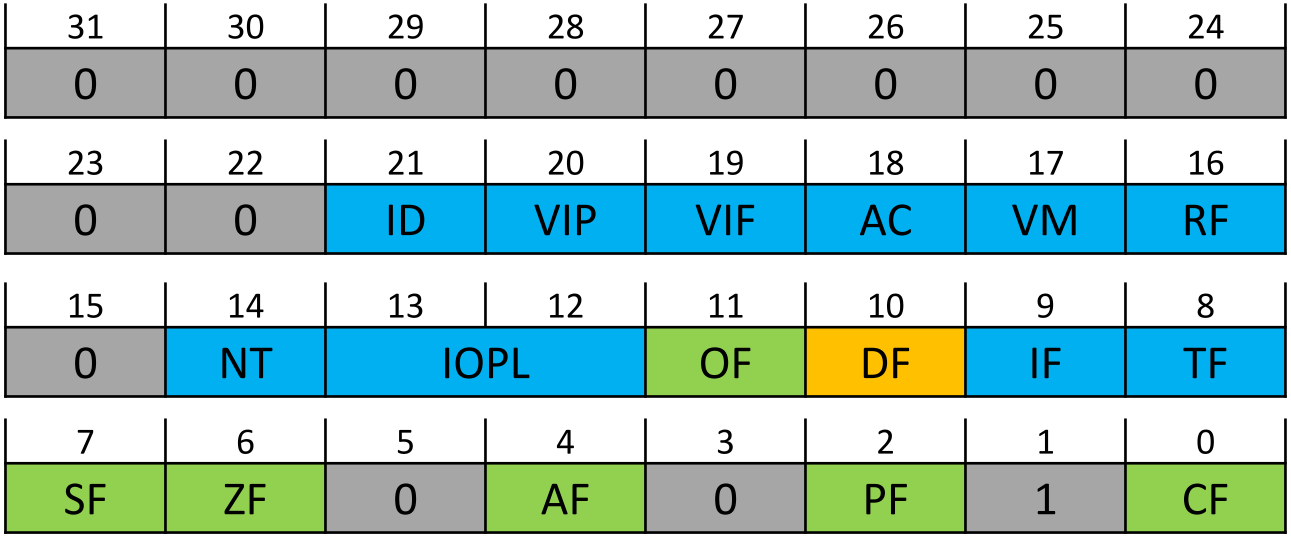
Segment registers

- CS
  - Address of code segment, IP works with it
- DS
  - Address of data segment (static variables)
- SS
  - Address of stack segment, ESP and EBP use it
- ES, FS, GS
  - Address of extra segment, base of EDI is ES

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

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.

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

x87 register set

- Floating point unit (mathematical co-processor)
- 8 pcs of 64 (80) bit registers (ST(0)-ST(7))
- Stack-based operation
- Double precision floating point representation
- 2 more bits per registers
  - 00 valid, 01 null, 10 special (Inf, NaN), 11 empty
- 16-bit status registers (E.g., OE, UE, ZE, TOS, B, DE)
- 16-bit control registers (E.g., RC, PC)
- 48-bit program counter and data pointer

Input-Output

- Both port-mapped and memory-mapped I/O
- 16 bit I/O addresses (0h-FFFFh)
- Separate instructions (in, ins, out, outs, etc.)
- More ports belongs to a device
  - Data, Instruction, Status
- Some device available via I/O ports:
  - DMA controller, programmable interrupt handler (8259A), timer (8254), keyboard (8042), real time clock, mathematical co-processor, PATA controller, etc.
- Linux: /proc/ioports

x86 instructions, operands

Several hundred instructions

Instructions has 0, 1 or 2 operand(s)

- Register (8, 16, 32 bits)
- Constant (8, 16, 32 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]*
```

*x86-64

x86 addressing modes

Summary of effective address (EA) forms

$$\left[\underbrace{\begin{Bmatrix} CS: \\ SS: \\ DS: \\ ES: \\ FS: \\ GS: \\ none \end{Bmatrix}}_{\text{Segment selector}} \underbrace{\begin{Bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESI \\ EDI \\ EBP \\ ESP \\ none \end{Bmatrix}}_{\text{Base}} \underbrace{\begin{Bmatrix} + \begin{matrix} EAX \\ EBX \\ ECX \\ EDX \\ ESI \\ EDI \\ EBP \\ none \end{matrix} \end{Bmatrix}}_{\text{Index}} \underbrace{\begin{Bmatrix} * \begin{matrix} 1 \\ 2 \\ 4 \\ 8 \\ none \end{matrix} \end{Bmatrix}}_{\text{Scale factor}} \underbrace{\begin{Bmatrix} + \begin{matrix} offset \\ nothing \end{matrix} \end{Bmatrix}}_{\text{Offset}}\right]$$

Example:

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

The same instruction in different form:

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

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
```

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

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)

ARM architecture

- ✦ Advanced/Acorn **RISC** Machine
- ✦ Since 1985
- ✦ Continuous development
  - ➥ ARMv1 (1985) – ARMv9 (2021) architectures
- ✦ The ARM Ltd sells license, no production
- ✦ Manufacturers: Apple, Samsung, Qualcomm, …
- ✦ Application: smartphones, media players, game consoles, routers, navigation systems, cameras, on-board systems (billions of CPUs per year)

ARM register set

In case of ARMv7 (AArch32):

- R0-R15: sixteen 32-bit general registers (in some modes further private registers)
  - Frame pointer (FP=R11)
  - Stack pointer (SP=R13)
  - Link register (LR=R14)
  - Programcounter (PC=R15)
- Statusregister (CPRS)

31								16
N	Z	C	V	Q	IT	J	DNM	GE

15						0
IT	E	A	I	F	T	Mode

ARM instructions

- Uniform 32-bit long instructions
- Only load/store instructions access memory
  - Other instructions use registers/constants only
  - Powerful indexed addressing modes
- Numerous CPU modes
  - E.g., User, System, IRQ, FIQ, Supervisor, Abort, …
- 2-priority-level interrupt subsystem
- Hardwired control unit (no microcodes)
- Bi-endiannes (default Little-endian)

ARM instructions, operands

- Arithmetic and logic instructions
  - 3 operands (register or constant)

```
add R1, R2, R3    ; R1=R2+R3
sub R1, R1, #1    ; R1=R1-1
```

- Load/Store instructions
  - 2 operands (register, constant or memory reference)

```
ldr R1, [R0] ; register indirect
str R2, [R1,#4]! ; pre-indexed
```

- Control-flow instructions
  - 1 operand (label)

```
b mylabel ; jump to a somewhere
```

ARM conditional execution

- 4-bit conditioncode in each machine instruction
- Instructions are not always executed, sometimes they are ignored
- No brancing instructions in short `if` statements
- Example: greatest common divisor

ARM assembly

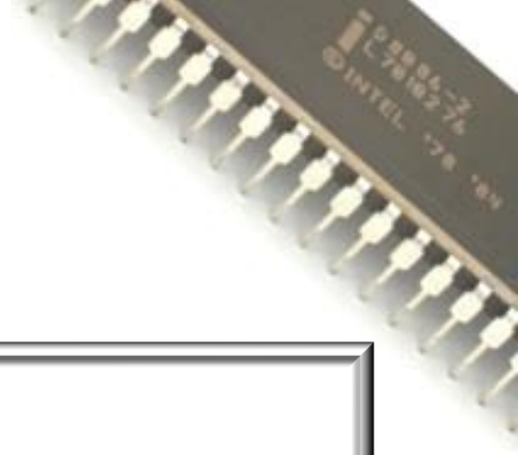
```
Loop   cmp    R0, R1
       subgt  R0, R0, R1
       sublt  R1, R1, R0
       bne    Loop
```

x86 assembly (intel)

```
Loop:  cmp    eax, ebx
       je     End
       jl     Less
       sub    eax, ebx
       jmp    Loop
Less:  sub    ebx, eax
       jmp    Less
End:   nop
```

ARM subroutine calling convention

- Parameters into the `R0−R3` registers
- Return address into the `LR` register („leaf” call)
- Control-flow by brach-and-link (`bl`) instruction
- The R4-R11 and LR registers saved into the stack (`push, pop`) by caller, if necessary
- Return address in `R0` register
- Return by `mov PC, LR` or `bx LR` instructions



Modern architectures

Parallel execution

Pipeline operation

Superscalar and vector processors

Thought-provoking questions

- ✦ Our computers usually have the same instruction set architecture.
  Why we find an order of magnitude difference between the prices of an Intel Celeron® G1610 and an AMD Ryzen™ 7 5700G processor?
- ✦ During the execution of a sequential (one-thread) program written in C, can we talk about parallelism?
- ✦ Using the same language, can we write more effective programs for advanced processors?

„Classical” architecture

In-order (serial) execution (von Neumann)

Possible performance improvements of „classical” architectures

- ✦ Increasing CPU clock frequency
- ✦ Application of co-processors (FPU)
- ✦ Direct Memory Access (DMA)
- ✦ Larger word size (larger address space/registers)
- ✦ Application of cache ✓
- ✦ Faster bus system ✓

Dynamic clock setting

Higher clock frequency leads to higher produced heat and higher consumption (limiting factor)

- Intel SpeedStep and AMD PowerNow!
  If the core is „idle” cock frequency and power is reduced saving energy.
- Intel Turbo Boost and AMD Turbo Core
  A core can use higher frequency if rest of cores are not loaded. (Heat remains below threshold.)

Floating-Point Unit

- FPU: mathematical co-processor
  - E.g., Intel x87
- Integer and floating-point arithmetic is architecturally different
- Earlier separate co-processor, now integrated into CPU
- Stack-based register set
  - ST(0)-ST(7)
- Separate instruction set
  - E.g., FADD, FMUL, FDIV, FSQRT, FSIN, …

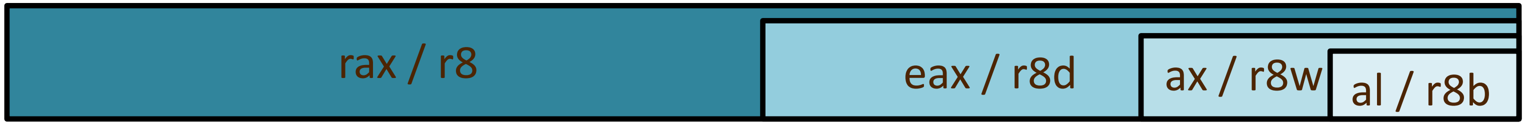
DMA and Cache

- Direct Memory Access (DMA)
  - Not all memory operation is controlled by the CPU
  - Direct RAM-RAM or RAM-I/O blocked data movement without CPU
- Cache
  - Intermediate storage between CPU and RAM
  - Contains a copy of recently/frequently used data
  - Low access latency
  - Small capacity

64-bit architectures

x86-64 (AMD64, Intel 64)

- ✦ 64-bit general purpose registers (rax, rbx, rcx, rdx, rbp, rsp, rip, rsi, rdi, r8-r15)



- ✦ Backward compatibility with x86 (IA-32)
- ✦ 64-bit virtual address (in implementation 48bit)
- ✦ 48-bit physical address (256TB) (extendable to 52 bit)
- ✦ Operation modes
  - ➥ Long mode (no memory segmentation), Legacy mode

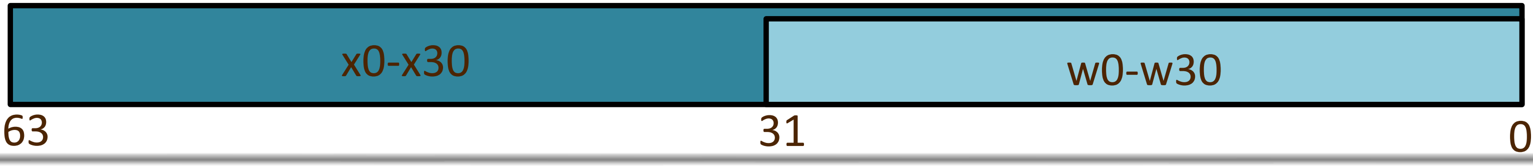
64-bit architectures

IA-64

- 2001-2021
- Itanium processor family
- 128 general 64-bit register
  - ➥ 32 static + 96 as stack

AArch64

- 2011-
- ARMv8 processor family
- 31 general 64-bit register (x0-x30), +PC



Parallelism

- Bit level parallelism
  - Operations on all bits at the same time
- Data level parallelism (DLP)
  - Same instruction on several data at the same time
- Instruction level parallelism (ILP)
  - Run of assembly instructions „next to each other”
- Task (thread) level parallelism (TLP)
  - Parallel execution of instruction groups
- Process level parallelism
  - More running process (multiprogammed OS)

Flynn’s taxonomy

Classification from the point of view of parallelism

- ✦ SISD (Single Instruction Single Data)
  - ➥ E.g., classical, early one core PCs
- ✦ SIMD (Single Instruction Multiple Data)
  - ➥ E.g., vector processors, GPU
- ✦ MISD (Multiple Instruction Single Data)
  - ➥ E.g., fault tolerant systems (space shuttle)
- ✦ MIMD (Multiple Instruction Multiple Data)
  - ➥ E.g., multicore- or superscalar processors

Instruction level parallelism

ILP solutions

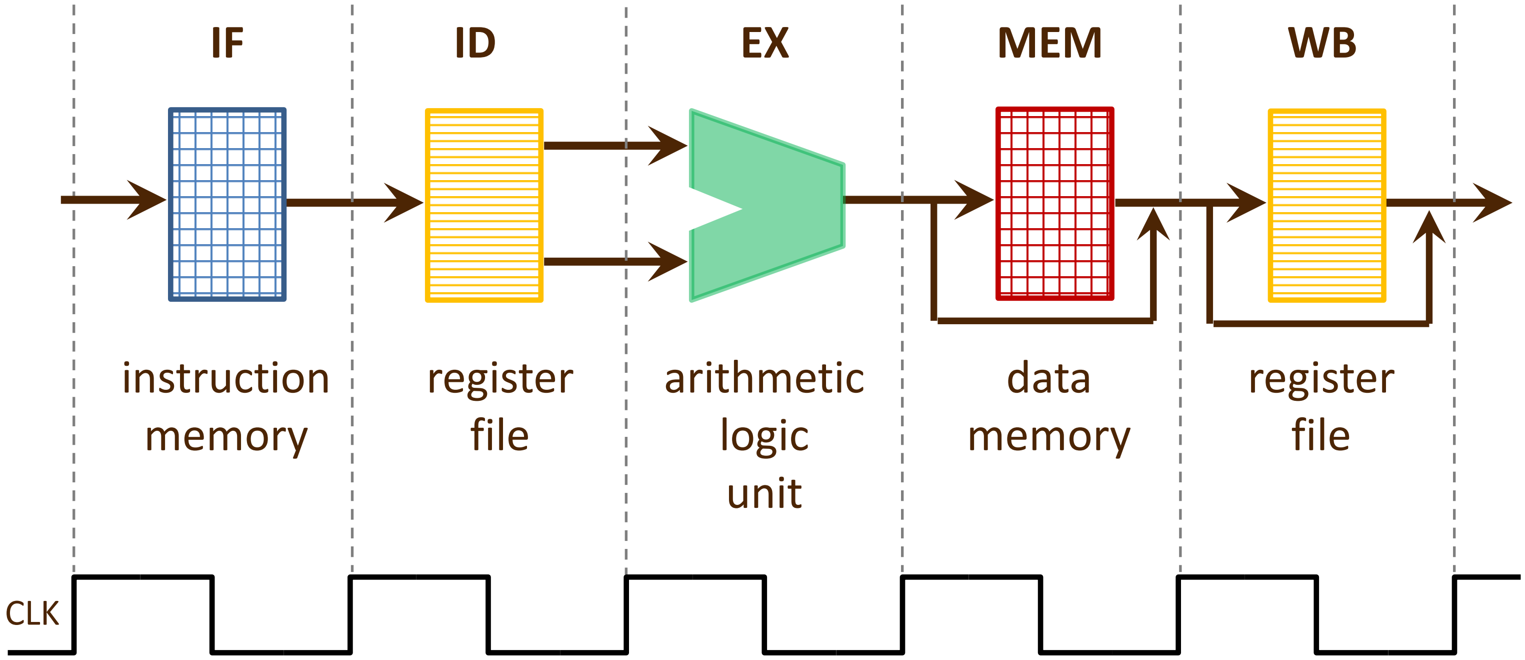
- Pipeline execution
- Out-of-Order Execution (OoOE)
  - Register renaming
- Speculative execution
  - Branch prediction
- Superscalar execution
- Using Very Long Instruction Word (VLIW)

Execution of an instruction

- Fetch-execute cycle (RISC logic)
- All instructions have the following phases:
  - Instruction fetch (IF)
  - Instruction decode (ID)
  - Execution (EX)
  - Memory access (MEM)
  - Write back (WB)
- An instruction is executed in a few (more than one) clock cycle

Execution of an instruction

- Simple RISC datapath



Pipelining

- **Pipelined architecture** of execution
- More instruction are under execution at the same time, but all of them in distinct phases
- In some processor even 30 phases/instruction
- Execution time of an instruction is not reduced
- However, the number of finished instructions in a time unit (throughput) is increases
- Program run is faster

Pipelining

CPU time [clock]	pipeline phases				
	IF	**ID**	**EX**	**MEM**	**WB**
1.	instruction 1				
2.	instruction 2	instruction 1			
3.	instruction 3	instruction 2	instruction 1		
4.	instruction 4	instruction 3	instruction 2	instruction 1	
5.	instruction 5	instruction 4	instruction 3	instruction 2	instruction 1
6.	instruction 6	instruction 5	instruction 4	instruction 3	instruction 2
7.	instruction 7	instruction 6	instruction 5	instruction 4	instruction 3
8.	instruction 8	instruction 7	instruction 6	instruction 5	instruction 4

Theoretical execution time (latency): 5 clock / instruction

Theoretical throughput: 1 instruction / clock

Hazard

- Sequential execution principle (von Neumann)
  - Supposes, all instruction is finished before the next instructions are started
- It is no longer true for modern processors
- Hazard: a situation when it leads to a problem
- Types
  - Data hazard
  - Structural hazard
  - Control hazard

Data hazard

- Instructions under execution uses/modifies the same data
  - RAW (Read After Write, data dependency)
    - add **r3**, r1, r2
    - add r4, **r3**, r2
  - WAR (Write After Read, name dependency)
    - add r3, r1, **r2**
    - add **r2**, r1, r4
  - WAW (Write After Write, name dependency)
    - add **r3**, r1, r2
    - add **r3**, r4, r1

ARM assembly examples

Structural and control hazard

Structural hazard

- If the processor hardware is not able to execute given instructions at the same time
- E.g., if instructions are in IF and MEM phase, reading memory simultaneously

Control hazard

- In case of conditional jumps CPU does not know in advance which instruction must be placed into the pipeline

Handling hazards

- Data hazard
  - Pipeline bubble/stall
  - Operand/result forwarding (bypassing)
  - Out-of-Order Execution (OoOE)
  - Register renaming
- Structural hazard
  - Pipeline bubble
- Control hazard
  - Pipeline bubble
  - Speculative execution

Pipeline bubble/stall

- If control unit realize hazard after IF phase inserts a NOP instruction delaying the next instruction in pipeline
- When the instruction is ready with a result (after the delay) the next instruction can use it
- **Bubble** („idle” state) in pipeline
- Running time is increasing, but still faster, then without pipeline
- Also known as pipeline stall

Pipeline bubble/stall

CPU time [clock]	pipeline phases				
	IF	**ID**	**EX**	**MEM**	**WB**
1.	ADD r1,r2,r3				
2.	SUB r4,r5,r1	ADD r1,r2,r3			
3.		SUB r4,r5,r1	ADD r1,r2,r3		
4.		SUB r4,r5,r1	(bubble)	ADD r1,r2,r3	
5.		SUB r4,r5,r1	(bubble)	(bubble)	ADD r1,r2,r3
6.			SUB r4,r5,r1	(bubble)	(bubble)
7.				SUB r4,r5,r1	(bubble)
8.					SUB r4,r5,r1

Data dependency caused 2 clock cycle delay

ARM assembly examples

Result forwarding

- The output of the EX phase of an instruction directly (earlier than WB) forwarded to the EX phase of the next instruction (no bubble)

CPU time [clock]	**pipeline phase**				
	IF	**ID**	**EX**	**MEM**	**WB**
1.	ADD r1,r2,r3				
2.	SUB r4,r5,r1	ADD r1,r2,r3			
3.		SUB r4,r5,r1	ADD r1,r2,r3		
4.			SUB r4,r5,r1	ADD r1,r2,r3	
5.				SUB r4,r5,r1	ADD r1,r2,r3
6.					SUB r4,r5,r1
7.					

Out-of-Order Execution

- Abbreviated: OoOE
- Execution of instructions in a different order as they are in the program
- That instruction is executed first which’s input(s) is ready first
- CPU tries to avoid bubbles (idle state) in pipeline with rearranging the instructions
- Recompilation accelerated by hardware
- Relatively large instruction window is needed

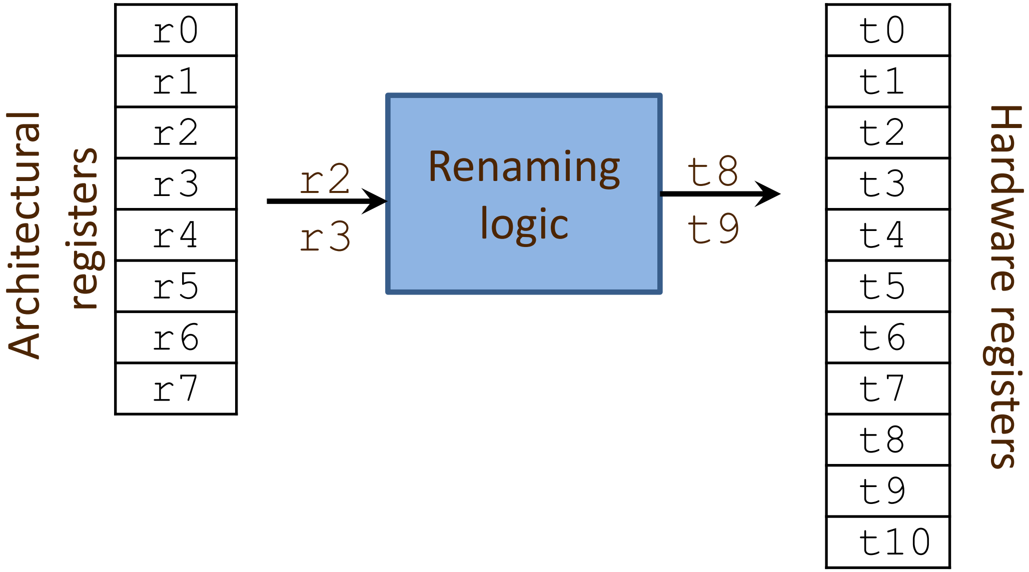
Out-of-Order Execution

- Reading instruction from memory
- Instruction goes to instruction queue and waits
- An instruction leaves the queue if its operands are available
- This instruction will be executed
- The result goes to a result queue and waits
- A given result leaves the queue (and saved to register file) if all results of previous instructions are ready/saved

Register renaming

- In case of name dependency (WAR and WAW hazard) the output of an instruction should overwrite a register, which's content is necessary later (number of empty register is limited)
- It can be avoided by renaming registers
- The instruction set refers to the items of architectural register file
- It is mapped to a greater hardware register file by a circuit

Register renaming

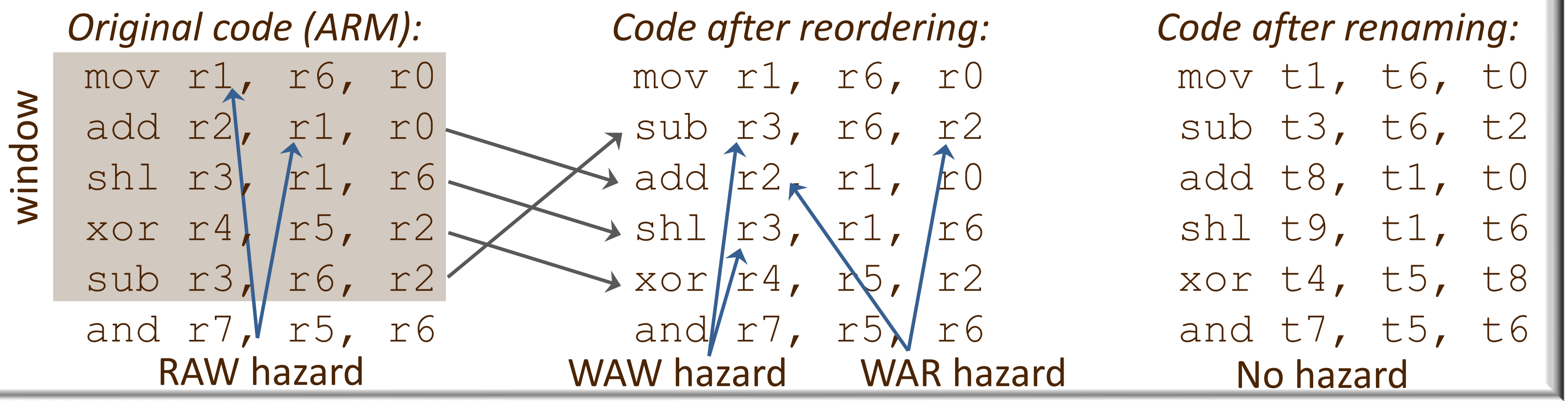


Architectural registers

r0
r1
r2
r3
r4
r5
r6
r7

Hardware registers

t0
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10



Original code (ARM):

```
mov r1, r6, r0
add r2, r1, r0
shl r3, r1, r6
xor r4, r5, r2
sub r3, r6, r2
and r7, r5, r6
```

Code after reordering:

```
mov r1, r6, r0
sub r3, r6, r2
add r2, r1, r0
shl r3, r1, r6
xor r4, r5, r2
and r7, r5, r6
```

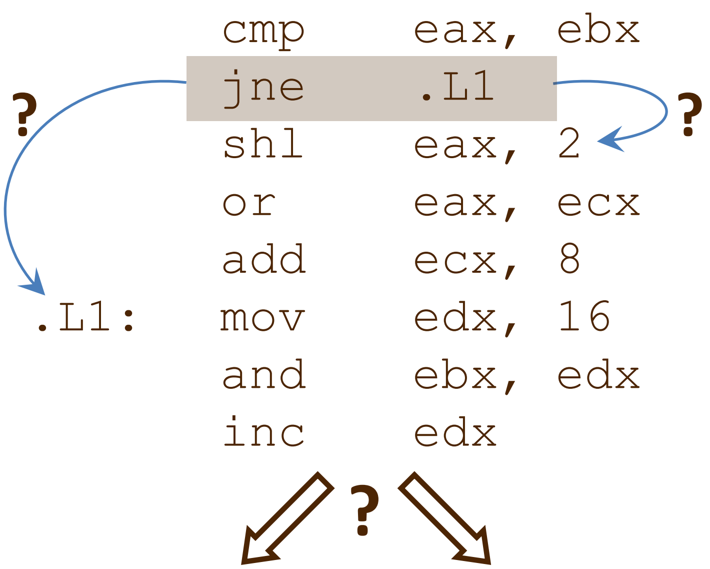
Code after renaming:

```
mov t1, t6, t0
sub t3, t6, t2
add t8, t1, t0
shl t9, t1, t6
xor t4, t5, t8
and t7, t5, t6
```

Speculative execution

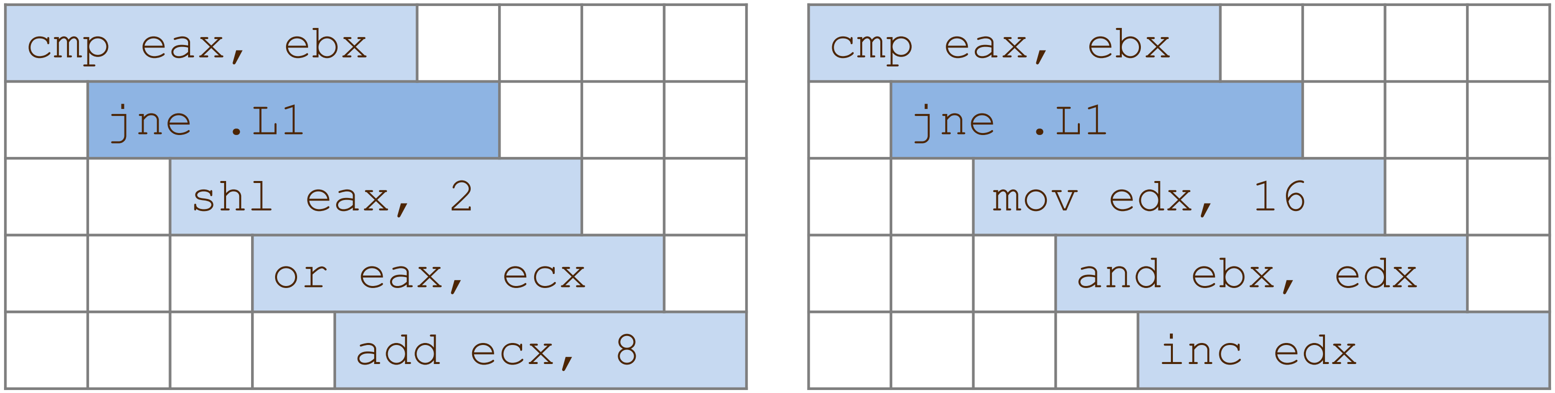
✦ Problem: control hazard

x86 assembly code:



```
       cmp    eax, ebx
       jne    .L1
       shl    eax, 2
       or     eax, ecx
       add    ecx, 8
.L1:   mov    edx, 16
       and    ebx, edx
       inc    edx
```

pipeline:



```
cmp eax, ebx
    jne .L1
        shl eax, 2
            or eax, ecx
                add ecx, 8
```

```
cmp eax, ebx
    jne .L1
        mov edx, 16
            and ebx, edx
                inc edx
```

Speculative execution

- Execution of instructions that maybe not necessary
- If CPU later realizes that execution was not necessary, it discards the result
- No idle state (bubble) in pipeline waiting for decision
- Solutions
  - Greedy prefetching (E.g., execution of both branches, if it find out which is needed CPU keep the result of only this)
  - Predictive execution (E.g., prediction of the necessary branch)

Branch prediction

- In case of conditional jump instruction, which is the next instruction to read into the pipeline?
- Separate circuit tries to predict the needed branch
- The execution of the predicted branch starts
- If later the prediction proved to be wrong, the result of these instructions are discarded, else saved time (not necessary to wait)
- The length of pipeline (it can be even 30) is proportional the need of good prediction

Superscalar processors

- More than one instruction is finished per clocks
- A processor core contains more execution unit

CPU time [clock]	pipeline phases				
	IF	ID	EX	MEM	WB
1.	instruction 1 instruction 2				
2.	instruction 3 instruction 4	instruction 1 instruction 2			
3.	instruction 5 instruction 6	instruction 3 instruction 4	instruction 1 instruction 2		
4.	instruction 7 instruction 8	instruction 5 instruction 6	instruction 3 instruction 4	instruction 1 instruction 2	
5.	instruction 9 instruction 10	instruction 7 instruction 8	instruction 5 instruction 6	instruction 3 instruction 4	instruction 1 instruction 2

Superscalar processors

- ILP and OoOE
- Instruction window
  - Set of „foreseeable” instructions
  - Scene of looking for independent instructions
  - Larger window size accelerates run
- Simple sequential programs are also executable
  - Appropriate compilation can increase performance increasing the throughput

VLIW processors

- Very Large Instruction Word
- A „big instruction” (called bundle) contains more, small (simple) instruction to be executed parallel
- Specially compiled programs are needed, which explicitly contains the instructions to be parallel
- Superscalar ILP and OoOE
- Simple hardware, complex compiler
- Intel terminology:
  Explicitly Parallel Instruction Computing (EPIC)

VLIW example

Calculation of `(x-y)*(x+y)/(z*z*8)`

$$\frac{(x-y)(x+y)}{8z^2}$$

value of `x` in `r1`, value of `y` in `r2`, value of `z` in `r3`

Scalar solution (6 instructions)

SUB r4, r1, r2	1.
ADD r5, r1, r2	2.
MUL r6, r4, r5	3.
MUL r7, r3, r3	4.
ASL r8, r7, #3	5.
DIV r9, r6, r8	6.

VLIW solution (3 instructions)

SUB r4, r1, r2	ADD r5, r1, r2	MUL r7, r3, r3
MUL r6, r4, r5	ASL r8, r7, #3	NOP
DIV r9, r6, r8	NOP	NOP

clock	IF	ID	EX			MEM	WB
1	A						
2	B	A					
3	C	B	A	A	A		
4		C	B	B	B	A	
5			C	C	C	B	A
6						C	B
7							C

ARM assembly examples

VLIW vs Superscalar

VLIW Pros

- Simple hardware
- Shorter clock cycle, faster operation
- Higher density of execution units on chips

VLIW Cons

- Slow compilation with special compiler
- Non-portable programs (software incompatibility)
- Larger size in RAM (due to NOP instructions)

Vector processors

- CPU executes an instruction on a one dimensional „array” of data (SIMD)
- Using large size registers which can store more data at the same time
- To handle them new instructions are needed
- Examples
  - MMX (Intel)
  - 3DNow! (AMD)
  - SSE
  - AVX
  - Helium
  - Neon

x86

ARM

Vector processors

- Logic of scalar processor

```
Repeat 10 times
    Read next instruction
    Read this and that numbers
    Add them together
    Save the result
Loop end
```

- Logic of vector processor

```
Read the next instruction
Read these 10 and those 10 numbers
Add them together simultaneously
Save results
```

Vector processors

MultiMedia eXtension (MMX)

- 64-bit registers (integer: 1x64, 2x32, 4x16, 8x8)
- 8 pieces (MM0-MM7)
- Aliases of FPU registers (causing concurrency)
- 3DNow! (development of AMD) using also float

Streaming SIMD Extensions (SSE)

- 128-bit registers (float: 4x32)
- 8 or 16 pieces (XMM0-XMM15)
- 70 new instructions

Vector processors

SSE2

- 128-bit registers
  (float: 2x64, 4x32; integer: 2x64, 4x32, 8x16, 16x8)

SSE3, SSE4

- Further instructions (+13, +47)

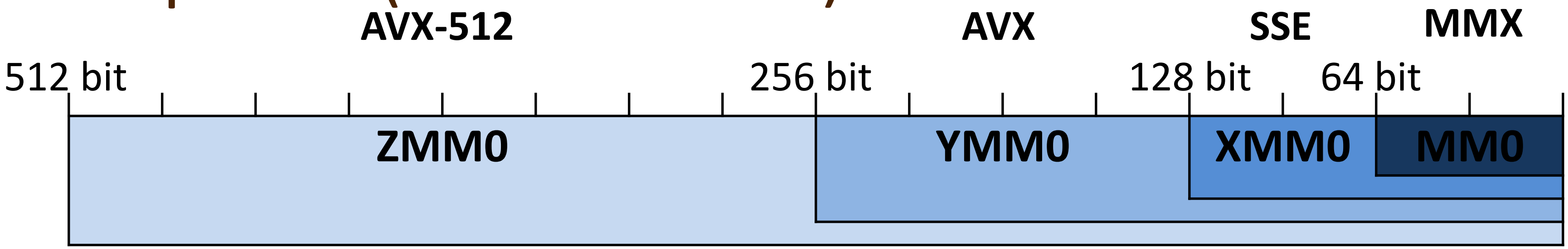
Advanced Vector eXtensions (AVX, AVX2)

- 256-bit registers (float)
- 8 or 16 pieces (YMM0-YMM15)
- Further instructions, int and float datas

Vector processors

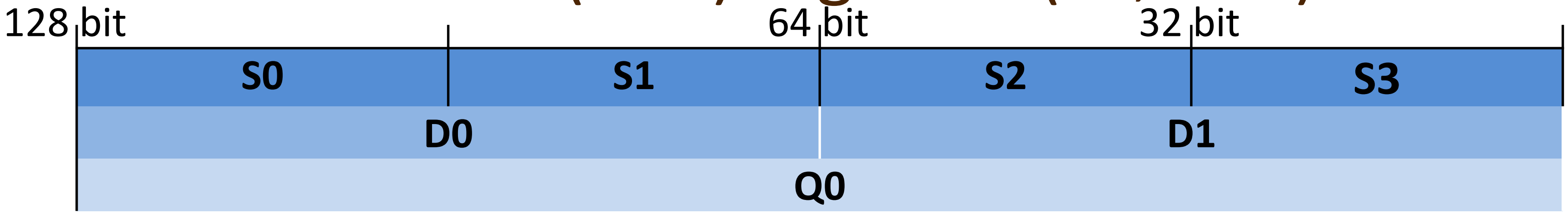
AVX-512

- 512-bit registers (integer, float)
- 32 pieces (ZMM0-ZMM31)

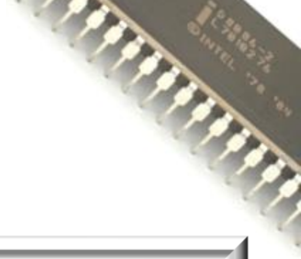


Neon (ARM)

- sixteen 128-bit (FPU) registers (int, float)



Loop unrolling



Traditional loop

```
for(i=0;i<100;i++)
     a[i]=b[i]+c[i];
```

Unrolled loop

```
for(i=0;i<100;i+=4){
   a[i]  =b[i]  +c[i];
   a[i+1]=b[i+1]+c[i+1];
   a[i+2]=b[i+2]+c[i+2];
   a[i+3]=b[i+3]+c[i+3];
   }
```

- Less loop control steps
- Performance can be improved in case of superscalar, vector and VLIW processors as well
- Easier detection of independent instructions

Hyper-threading

- Intel SMT (Simultaneous MultiThreading) solution
- More thread execution in one CPU/core
  - More „context handler”
  - One execution resource
- If a thread have to wait, the other can be executed
  - Reducing pipeline bubbles
- All „physical” core can behave as two „logical” cores
- Operating system support is important

Multi-processor systems

- Computer with more than one processors
- Shared or distributed memory
- Implementations
  - Homogeneous (symmetric)
    Same type processors with shared memory
  - Heterogeneous
    Different type processors for various tasks
  - Cluster
    Separate processors with own memory in network

Homogeneous systems

- On the same chip
  - **Multi-core** system
    some independent CPUs (namely cores) on a die
    application: personal computers
  - **Many-core** system
    a lot of (up to thousands) cores
    application: servers, supercomputers
- On separate chips
  - Computers using multisocket motherboard
    application: servers

Multi-core processors

- ✦ More independent processing unit on a chip
- ✦ In case of N cores performance is better and consumption is lower than in case of N CPUs
- ✦ Usually own L1 (maybe L2) cache, but shared L3
  - ➥ Cache coherency problems
- ✦ Parallel thread/process run (MIMD)
  - ➥ Task-level parallelism (E.g., Java Thread, C OpenMP)
  - ➥ Operating system support is needed

Heterogeneous sytems

Different CPUs

- Foe example ARM big.LITTLE architecture

Hardware accelerators

- Floating Point Unit (FPU)
- **Graphical Processing Unit** (GPU)
- Cryptography accelerators (e.g., AES-NI)
- Digital Signal Processors (DSP)
- Field-Programmable Gate Array (FPGA)
- Artificial intelligence accelerators (pl. PNN)

Graphics Processing Unit

- Abbreviated: GPU
- Task:
  - texture handling,
  - rendering images,
  - 3D graphics acceleration,
  - video decoding, …
- Integrated or separate video card
- Can have own dedicated memory
- Producers: nVIDIA and AMD (ATI)

Graphics Processing Unit

- Big computation capacity (parallel, SIMD)
- Parallel execution of hundreads of threads
- Hardware-level thread management
- Thousands of registers
- Big memory bandwidth requirement (even TB/s)
- Special instruction set

  `1/sqrt(x), a*b+c`

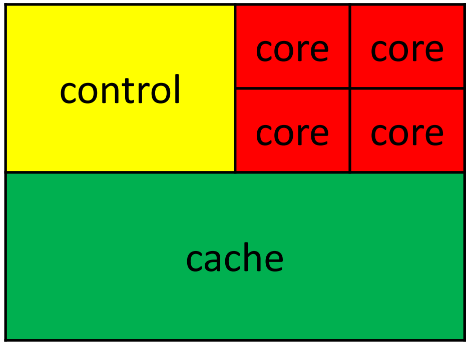
GPU programming

- General-Purpose GPU (GPGPU)
- GPU receives (non-graphics) computations from CPU
- Huge memory bandwidth requirement
- Programming
  - OpenCL
  - CUDA
  - MATLAB
- Accelerated Processing Unit (APU)
  - CPU and GPU elements in a heterogeneous system

CPU vs GPU

CPU

- Low latency
- Sellow pipeline (<30 stages)
- Optimized for serial operation
- Max few 10 cores



control

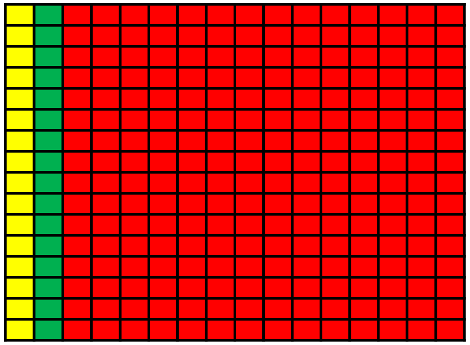
core core

core core

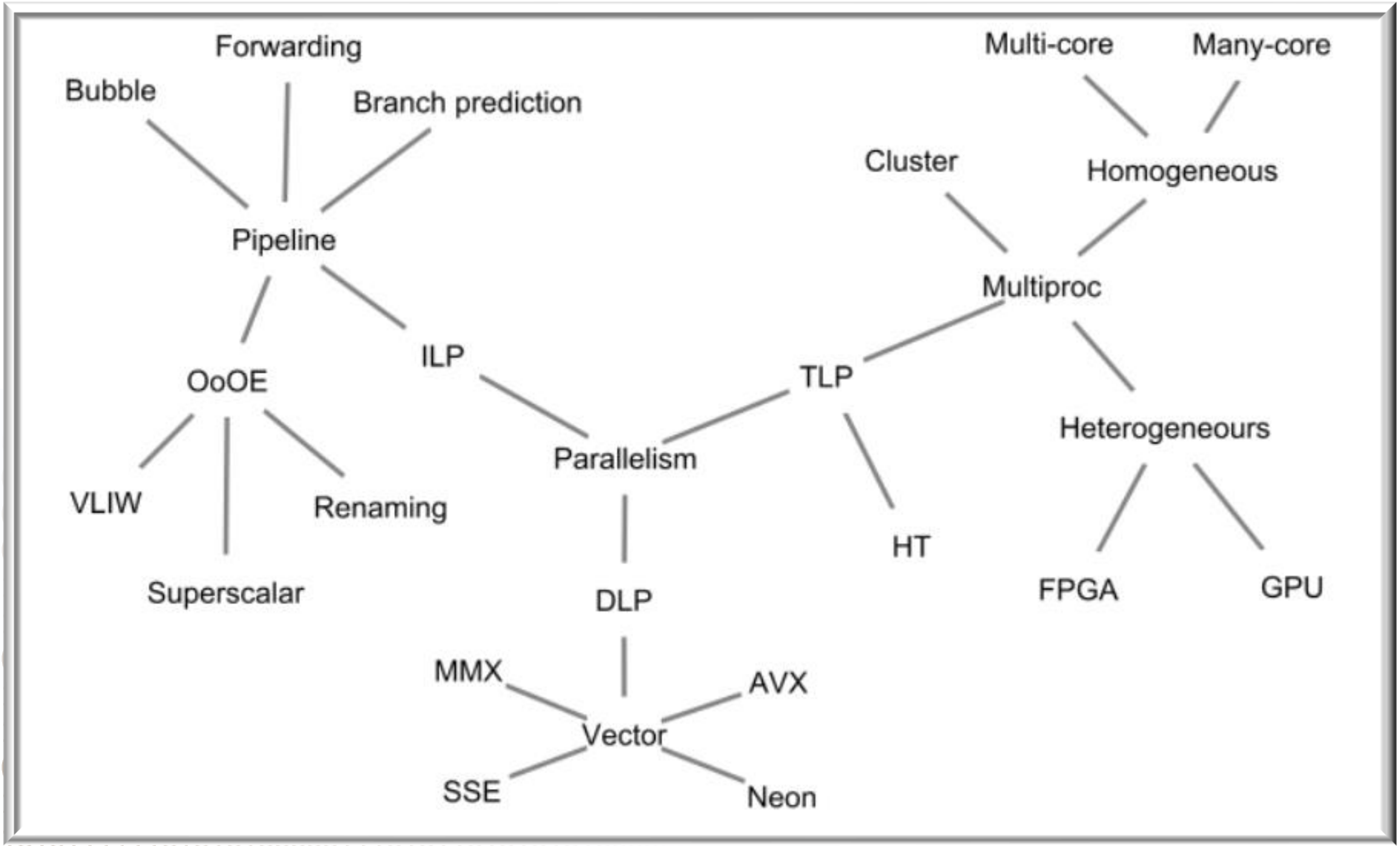
cache

GPU

- Large troughput
- Deep pipeline (>100 stages)
- Optimized for parallel operation
- Even 10000 „cores”



Mindmap: parallelism



Performance analysis

- Performance equation:

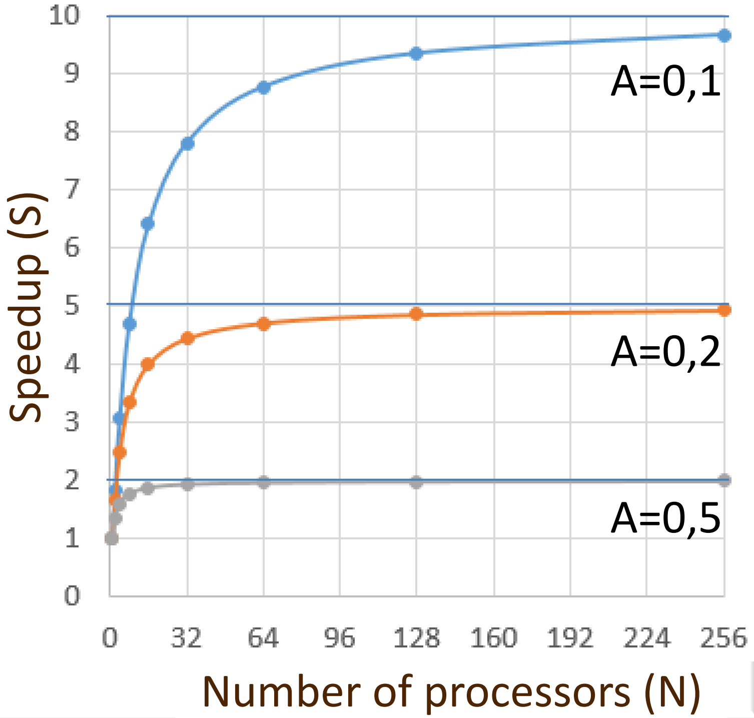
$$T = NI \times eCPI \times 1/f$$

  - T: Program execution time (s)
  - NI: number of machine code instructions (considering loops)
  - eCPI: effective number of cycles per instructions, depends on type of instruction, microarchitecture (pipeline, superscalar, RISC/CISC, etc.)
  - f: clock frequency (Hz)
- Performance measure unit:
  - FLoating point Operations Per Second (FLOPS)

Amdahl's law

- N processor (thread) does not result in N times acceleration in a process execution
- $S(N) = \frac{T_1}{T_N} = \frac{1}{A+\frac{1}{N}(1-A)}$

  Where S is the speedup,
  T is the running time and
  A is the sequential ratio of execution
  0 ≤ A ≤ 1



Intel x86 history & other architectures

History of Intel processors

Other architectures

High-Performance Computing

Thought-provoking questions

- What happened in the world of processors during the last decades?
- What kind of processors your parent could use when they were younger?
- How new is the idea of cache/pipeline/SIMD?
- Is there anything else outside the x86 and the ARM architectures? Where can we find them? What do they know?
- What is in supercomputers?

Main architecture families

- **Intel x86**
  - 8086, 80386, Pentium, Core i7, Core i9, Ryzen 9
- ARM
  - ARM2, Cortex-A72
- Motorola 68000
- MIPS
- Alpha
- PowerPC
- SPARC

CPU

- Microchip PIC
- AVR

MCU

Intel 8086

- Released in 1978
- 16-bit registers
- 20-bit physical address (1MB)
- Clock 5-10 MHz
- 3000nm technology
- 29 000 transistors
- 40 pin DIP package
- DMA support

```
mov ax, 512 ;store the value to ax register
```

Intel 80386

- Released in 1986
- 32-bit registers
- 4GB address space
- Clock 12-40 MHz
- 1500-1000nm technology
- 275 000 – 855 000 transistors
- 132 pin PGA package
- Cache, virtual memory, pipeline

Intel Pentium

- Released in 1993
- 32-bit registers
- 4GB-os address space
- Clock 60-300 MHz
- 800-250nm technology
- 3 100 000 – 4 500 000 transistors
- 273 pin Socket 4; 321 pin Socket 7
- Superscalar, MMX, L2 cache, integrated FPU

Intel Pentium 4

- ✦ Released in 2000
- ✦ 32-bit registers
- ✦ Microarchitecture: NetBurst
- ✦ Clock 1300-3800 MHz
- ✦ 180-65nm technology
- ✦ 42 – 188 million transistors
- ✦ Socket 423 or Socket 478
- ✦ SSE3, 20-31 level pipeline, branch prediction, HyperThread

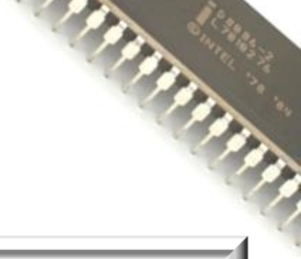
Intel Core i7

- Released in 2008
- 64-bit registers (x86-64)
- Microarchitecture: Nehalem, Sandy Bridge, Ivy Bridge, Haswell, Broadwell, …
- 45-14nm technology
- 0.7 – 3.2 billion transistors
- LGA socket (1000-2000 contacts)
- 2-8 cores, L3 cache, integrated GPU, AVX-512, integrated NorthBridge

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

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)

ARM2

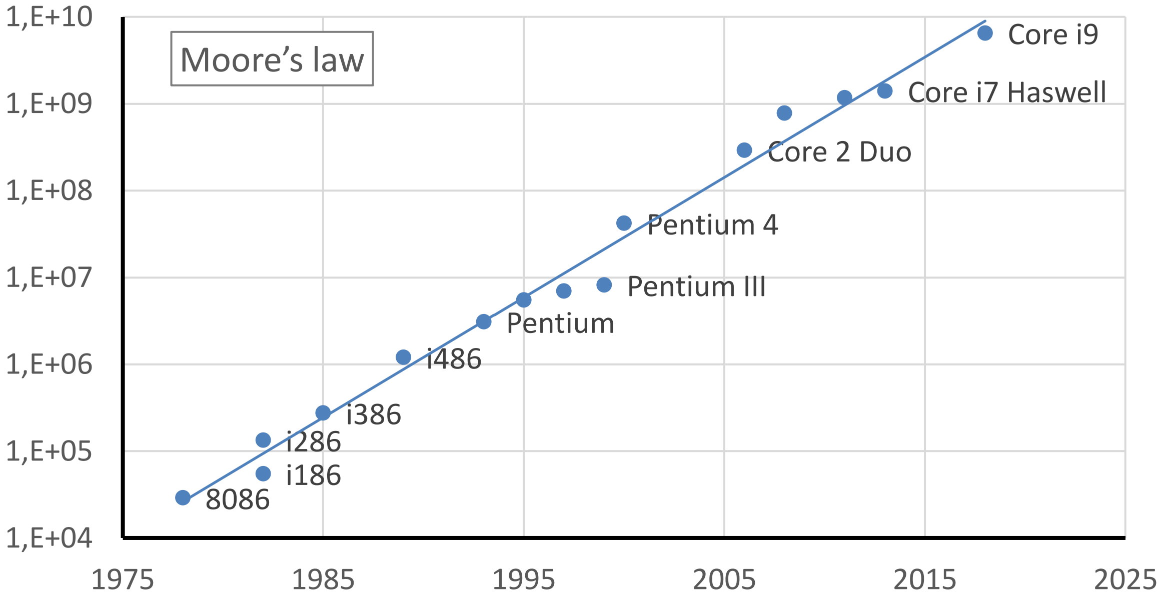
- Released in 1986
- 32-bit registers
- 64MB address space (26-bit address bus)
- 8-12 MHz clock frequency
- 2000nm technology
- 27 000 transistors
- 84-pin PLCC package
- No cache, no DMA, no microcode (RISC), 3-stage pipeline

ARM Cortex-A72

- Released in 2016
- 64-bit registers (AArch64)
- Microarchitecture: ARMv8-A
- 16nm technology
- 1 billion transistors
- 1-4 cores, 0.5-4MB L2 cache, 3-way superscalar, speculative execution, Neon Vector FPU, DSP
- Used in Raspberry Pi 4, Samsung Galaxy A9 smart phones (big.LITTLE)

Moore’s law

- Number of transistors in processors doubles about every two years
- We are close to the limits (ca. 2025)
  - because of the atomic size



MIPS

- Microprocessor without Interlocked Pipeline Stages (1985-)
- RISC
- 32/64 bit (31 GPRs + 32 FPRs)
- 5 level pipeline
- OoOE, SIMD, superscalar
- Multilevel cache
- Application: PlayStation 2 and Nintendo 64 game consoles, CISCO routers, HPC

```
lw $t1, ($t0) #move 2 bytes from the address
               stored in t0 to t1
```

Alpha

- Developer DEC (1992-)
- Goal to relpace CISC based VAX architecture lecserélése
- RISC
- 64-bit (31 GPRs + 31 FPRs)
- multilevel cache
- OoOE, SIMD
- Application: workstations, HPC

```
ldq ra, 0(sp) # pop return address from stack
```

PowerPC

- Developer: Apple+IBM+Motorola (1992-)
- RISC
- 32/64 bit (32 GPRs + 32 FPRs)
- Superscalar, vector processor
- Runtime changeable little-endian/big-endian byte order
- Application: Xbox 360, Nintendo Wii, F35 Raptor

```
li 3, 0(5) ;move data from the address stored
            in register 5 to register 3
```

SPARC

- Scalable Processor Architecture (1993-)
- Sun Microsystems (Oracle)
- RISC
- 32/64-bit (32 GPRs + 32 FPRs)
- 1-32 cores (<=5.0GHz)
- L3 cache
- Java support
- Application: servers, HPC (K-computer)

```
add %L1,%L2,%L3 !Sum of %L1+%L2 into %L3
```

Microchip PIC

- Microcontrollers
- (Modified) Harvard architecture
- RISC
- 8/12/14/16 bit
- 1 general work register (W)
- No difference between data memory and registers
  - E.g.: PC and other registers mapped in memory

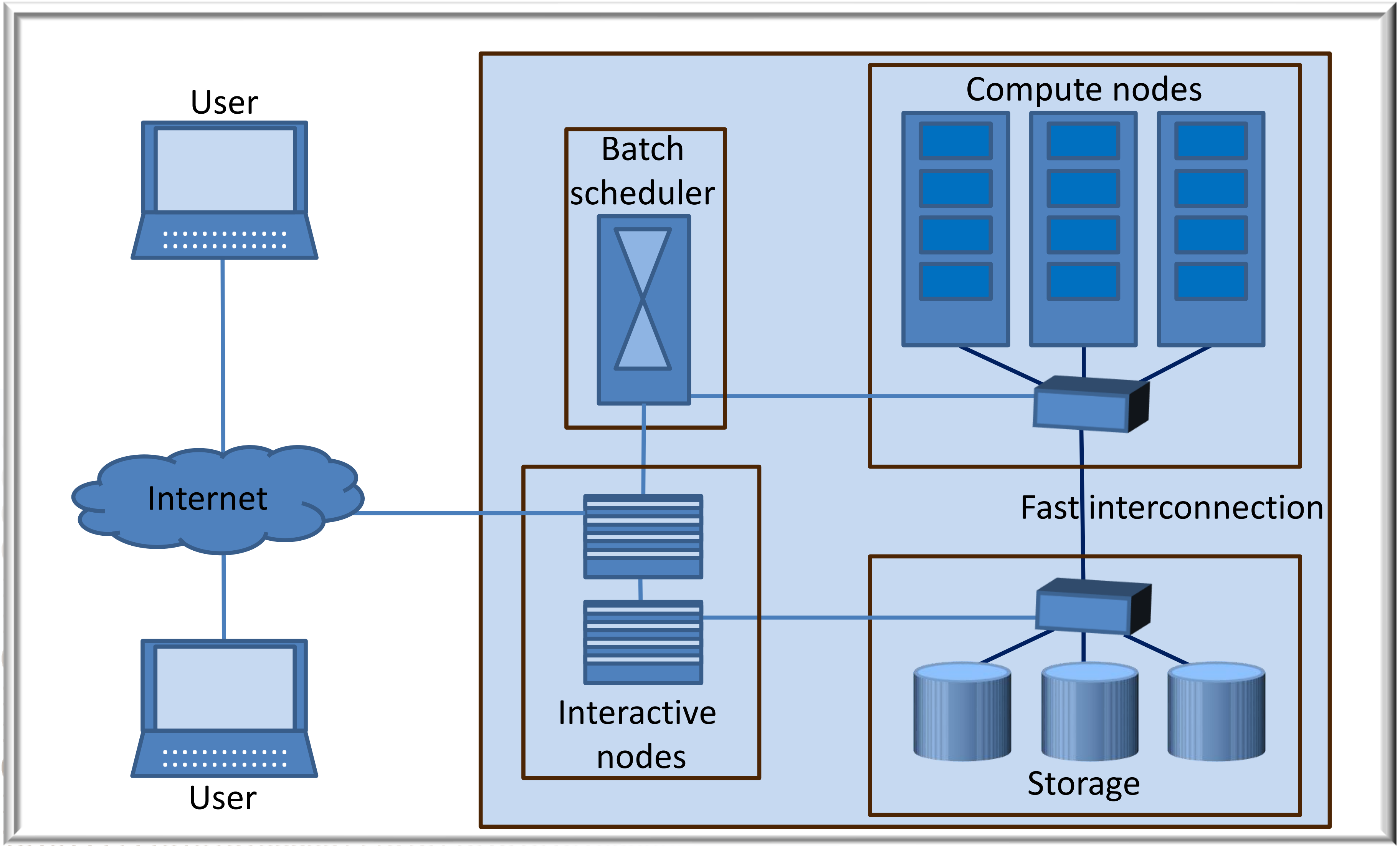
```
MOVLW d'29' ;move decimal 29 into W
```

Atmel AVR

- Microcontroller
- (Modified) Harvard architecture
- RISC
- 8/32 bit
- 32 general 8-bit work registers (R0-R31)
- Built in memory (SRAM, Flash)
  - Address space: register file + memory
- Application: Arduino

```
ldi r16, 0xF1 ;load hexadecimal 0xF1 to r16
```

Supercomputer architecture



High-Performance Computing

HPC - Top 1 in Hungary, TOP60 in world (2023)

- Komondor
- Debrecen, DE Kassai campus
- 5 PFLOPS
- 384 AMD EPYC™ 7763 CPUs (24 576 cores) + 12 Intel® Xeon® Gold 6254 CPUs (216 cores) + 200 NVIDIA A100 GPUs
- 49 664 GB RAM
- 11 900 TB storage
- 4 700 000 000 HUF (approx. 12.8 million USD)
- 500 kW consumption

High-Performance Computing

HPC - TOP 1 (2022)

- Frontier (Hewlett Packard Enterprise CRAY EX235A)
- Oak Ridge National Laboratory, USA
- 1102 PFLOPS
- AMD Epyc 64C CPUs + Instinct MI250X GPUs
- 8 730 112 cores
- *??? GB RAM*
- Ca. 700 000 TB storage
- 74 HPE Cray EX cabinets (>268 000 kg)
- 90 miles worth of HPE Slingshot-11 networking cables
- Ca. 600 000 000 USD
- 21 100 kW consumption

HPC challenges

- Limited clock frequency
- Gigantic costs
- Huge energy consumption
  - Carbon (ecological) footprint
- Increasing communication latency
- More components results in more frequent failures (low reliability)
- Application scalability problems
  - Amdahl’s law

Operating system and the programming

Processes

Scheduling

Inter-process communications

Program development

System calls

Thought-provoking questions

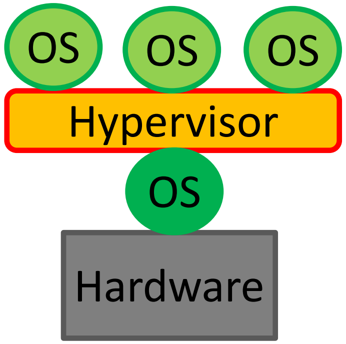
- ✦ Can we start 200 programs simultaneously on s 4 core computer?
- ✦ Can multiple operating systems run at the same time on the same hardware?
- ✦ Can we overwrite the variables of another program by inaccurate pointer operations? Why?
- ✦ How a source code stored in a text file on disk become a bit sequence under execution loaded into the memory?

Operating system

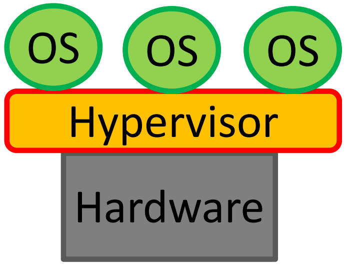
- A software that manages the hardware and software resources of computer
- OS components
  - Process management
  - Memory management
  - File management
  - I/O and device management
  - Protection management
  - User interface
  - Program development

Virtualization

- Multiple operating system running on one machine
  - One host
  - More guests
- Types
  - Hosted virtualization
  - Native virtualization
- Hypervisor (VMM)
  - Hyper-V, Vmware, VirtualBox
- Hardware support (VT-x, AMD-V)



Hosted virtualization



Native virtualization

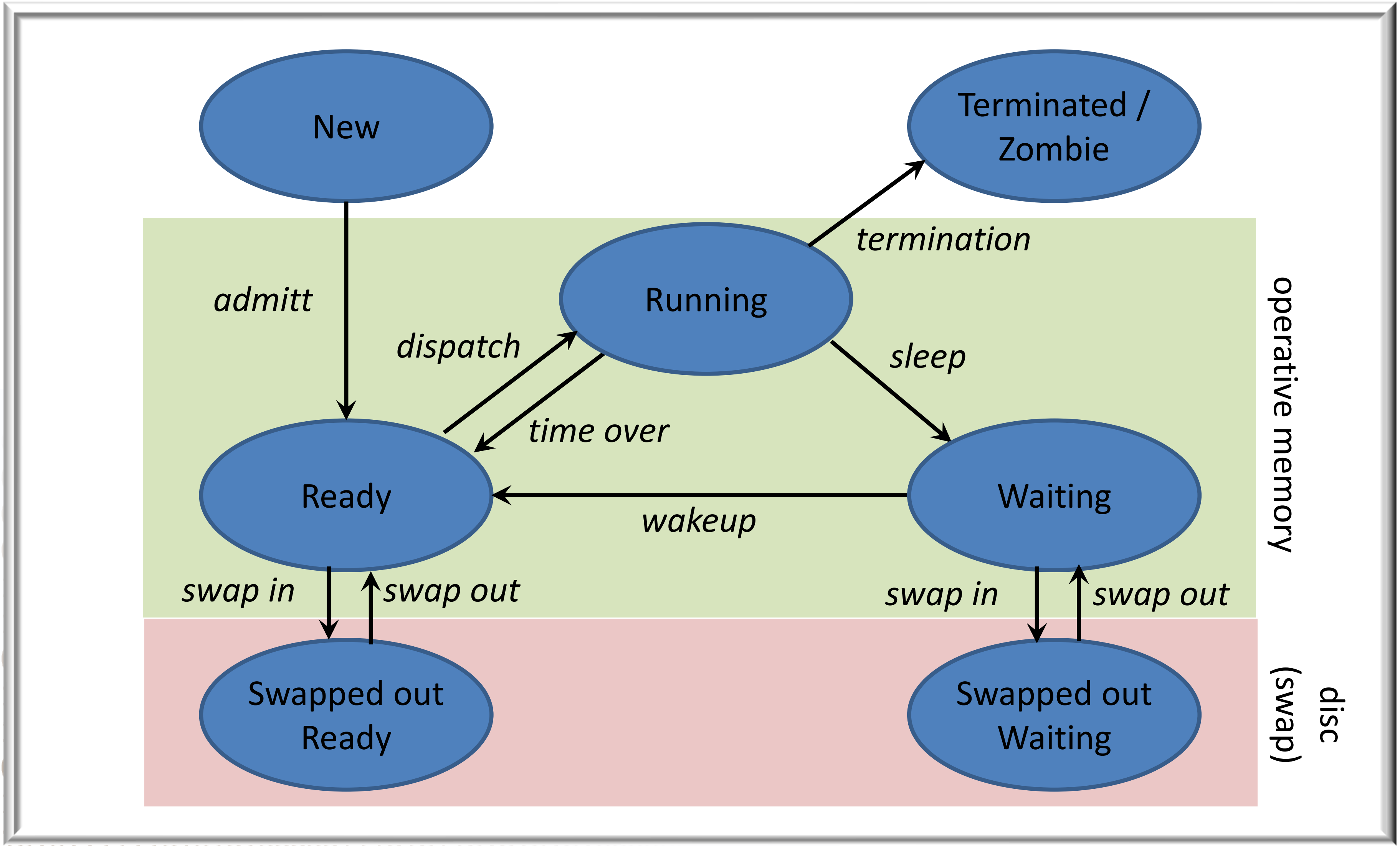
Process

- Program under execution
  - In memory
  - (not source file, not executable file)
- It has own resources
  - CPU time, memory, files, etc.
- Process Control Block (PCB) contains information about processes
- Processes can communicate with each other
  - Inter-process communication (IPC)

Process

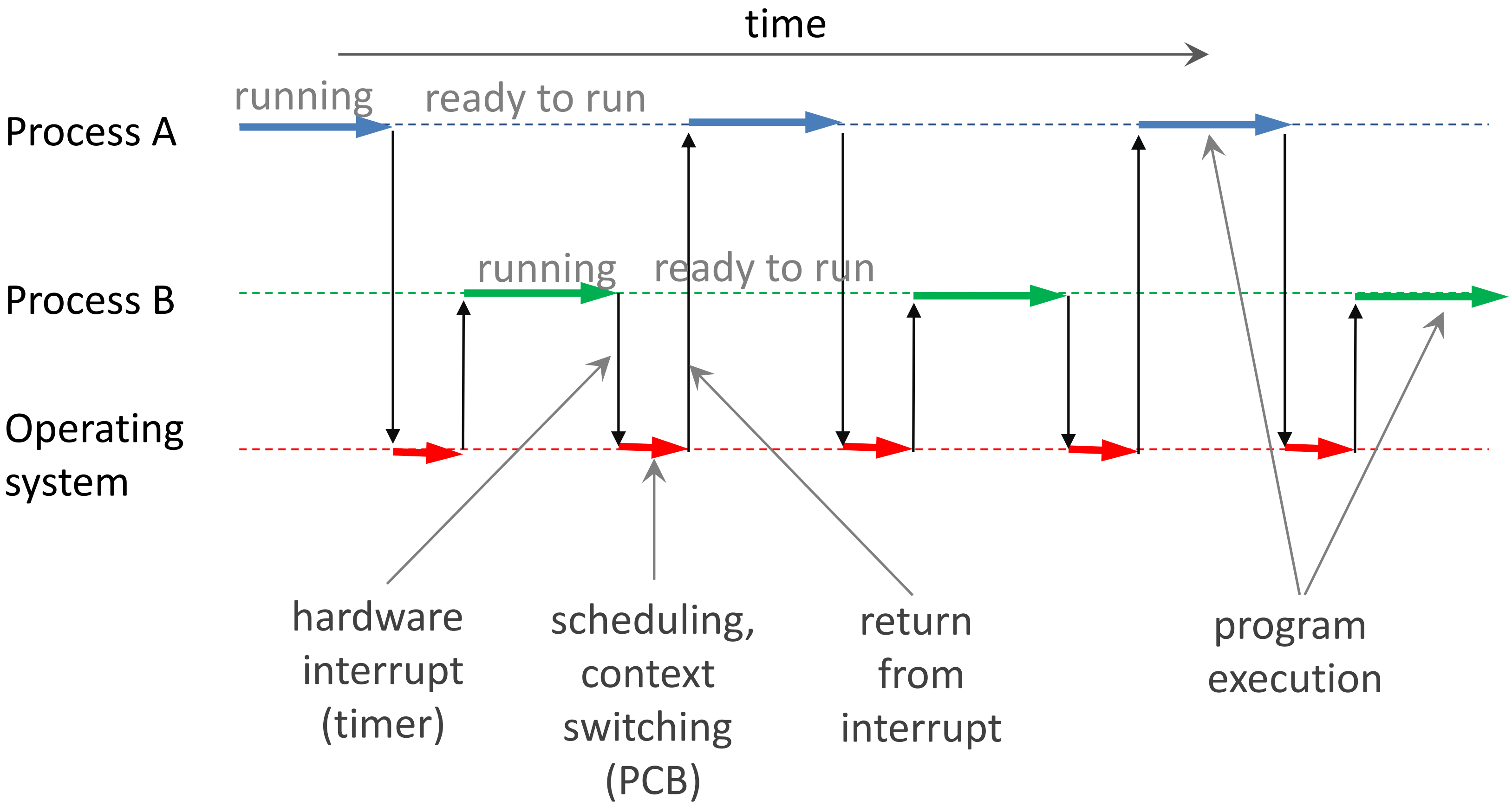
- In a multi-programmed environment there exists more processes at the same time
- Unique process ID (PID)
- All es have a parent process and can have several child processes
- A process build-up from (one or more) threads
- Processes have states
  - There are some state changes between them

State changes of a process



Scheduling

- **Time division** multiplexing of processes



Process schedule

- In multi-programmed environment (processes)
- **Scheduler**: gives the CPU to a (ready to run) process
  - Algorithms: FCFS, RR, SJF, EDF
- Preemptive scheduling
  - E.g., hardware timer runs out and request an interrupt
  - Actual process is interrupted, code of scheduler is activated
  - It sets up a new timer, chose a process, gives the control to it (modifying PCB)

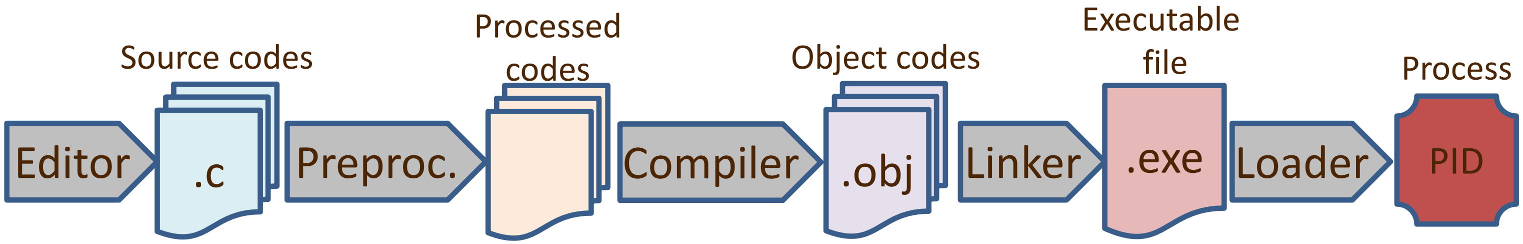
Inter-process communication

Sometimes processes need information exchange between each other

- File: more process use the same file
- Pipe: like input/output redirection
- Signal: notification about an event
- Semaphore: synchronization of shared resources
- Shared memory: multiple used RAM area
- Socket: via network protocols
- …

Programming

- Creation and execution of programs have several steps
- This is supported by several softwares:
  - Editor
  - Preprocessor
  - Compiler (assembler)
  - Linker
  - Loader



Editor

- Text editor to create the source code
- **Source code**: (text) file stored on the disk
- Probably language sensitive tools
  - Knowledge of syntax
    - Coloring, error indication
  - Automatic completion
- Offten part of an IDE (Integrated Development Environment)
- Output: source code
- For example: `gedit hello.c`

Preprocessor

- Eliminate comments
- Eliminate not necessary, multiple whitespaces
- Inserting the text of header files
  `#include<…>` or `#include”…”`
- Define macro replacement
  ```
  #define PI 3.141593
  #define avg(a,b) (((a)+(b))/2)
  ```
- Conditional compilation instructions
  `#if … #elif … #else … #endif`
- For example: `gcc -E hello.c`

Compiler

- Its aim is to convert source code to binary **machine code**
- Main tasks
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis
  - Optimization
  - Code generation
- Don't confuse it with interpreter.
- Output: **object code** (e.g., hello.o)
- For example: `gcc -c hello.c`

Linker

- A program can consist of multiple source codes
  - Modular programming
  - Separate object code for each source code
  - Libraries are also separate files
- Object codes does not contain „real” addresses
  - Only references
- Linker resolves references (symbol table)
- Some code parts are relocated to create only one **executable** file
  - Binary formats (PE, ELF; e.g., hello.exe, a.out)
- For example: `ld hello.o`

Static and dynamic linking

Object codes and libraries can be linked in two different ways

- Static linking
  - Linking happens before the execution
  - The executalbe file contains every necessary codes
  - Larger executables, faster execution
- Dynamic linking
  - At runtime (in case of need) linking
  - Effective disc/RAM management, flexible modification

Loader

- Component of operating system, to launch an executable
- Tasks
  - Creation of a **process**
  - Loading program image from disc to RAM
  - Pushing command line arguments into the stack
  - Initialization of registers
  - Jumping to an entry point
- For example: `./a.out`

Dual mode operation

Kernel mode

- High CPU privilege
- Processor can execute all instructions
- All memory address can be accessed

User mode

- Low CPU privilege
- Processor has restricted instruction set
- Not whole memory can be addressed
  - See: segmentation fault

System call

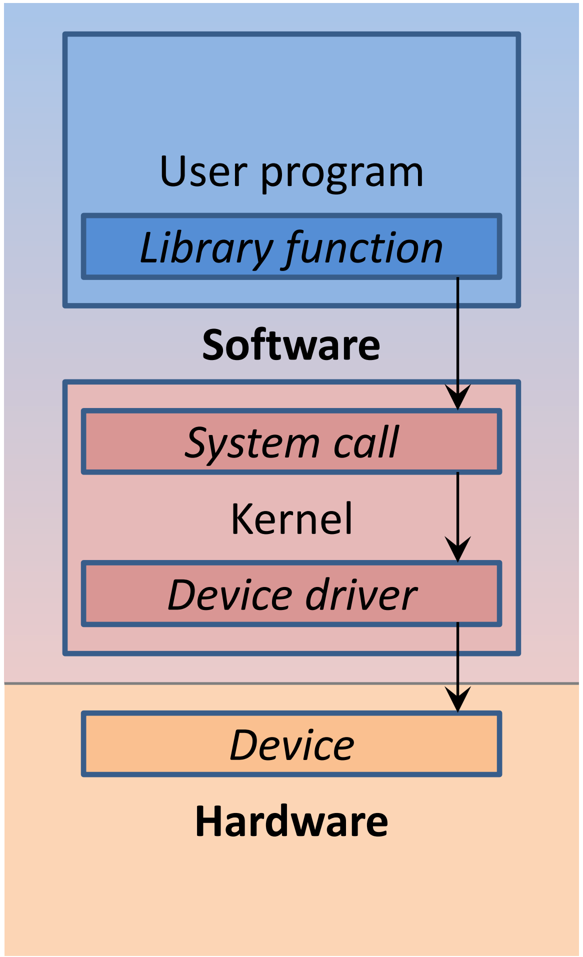
- Interface between user programs and the kernel
- The program requires a service from the operating system kernel
- Implementation with software interrupt
- Kernel mode operation
- Example
  - Process handling
  - File handling
  - Device handling

System call

Example:

Writing a file on USB drive

- Own *C* program
- *fprintf()* standard library function
- *write* system call
- USB driver
- USB drive



Address space of a program

- Due to virtual memory management all programs (processes) manage the so-called user space part of the full address space as onw area. No access to kernel space.
- This address space is devided into **segments**
  - Code segment (text)
  - Read-only data segment (rodata)
  - Writeable data segment (data, bss)
  - Dynamic memory (heap)
  - Shared memory segment (mmap)
  - Stack segment

Memory segments

- Code segment
  Contains the machine code instructions loaded from executable. Read-only during execution. The PC register points to here.
  For example: `x=a>5?5:a;`
- Read-only data segment
  Write protected data, for instance the konstant strings.
  For example: `printf(”Hello World\n”);`

Memory segments

- Writeable data segment
  - data part

    Static lifetime (e.g., global) variables which are initialized in the program.

    For example: `static int x=123;`
  - bss part

    Static lifetime (e.g., global) variables which are uninitialized at declaration. They are filled with 0 bits.

    For example: `static int x;`

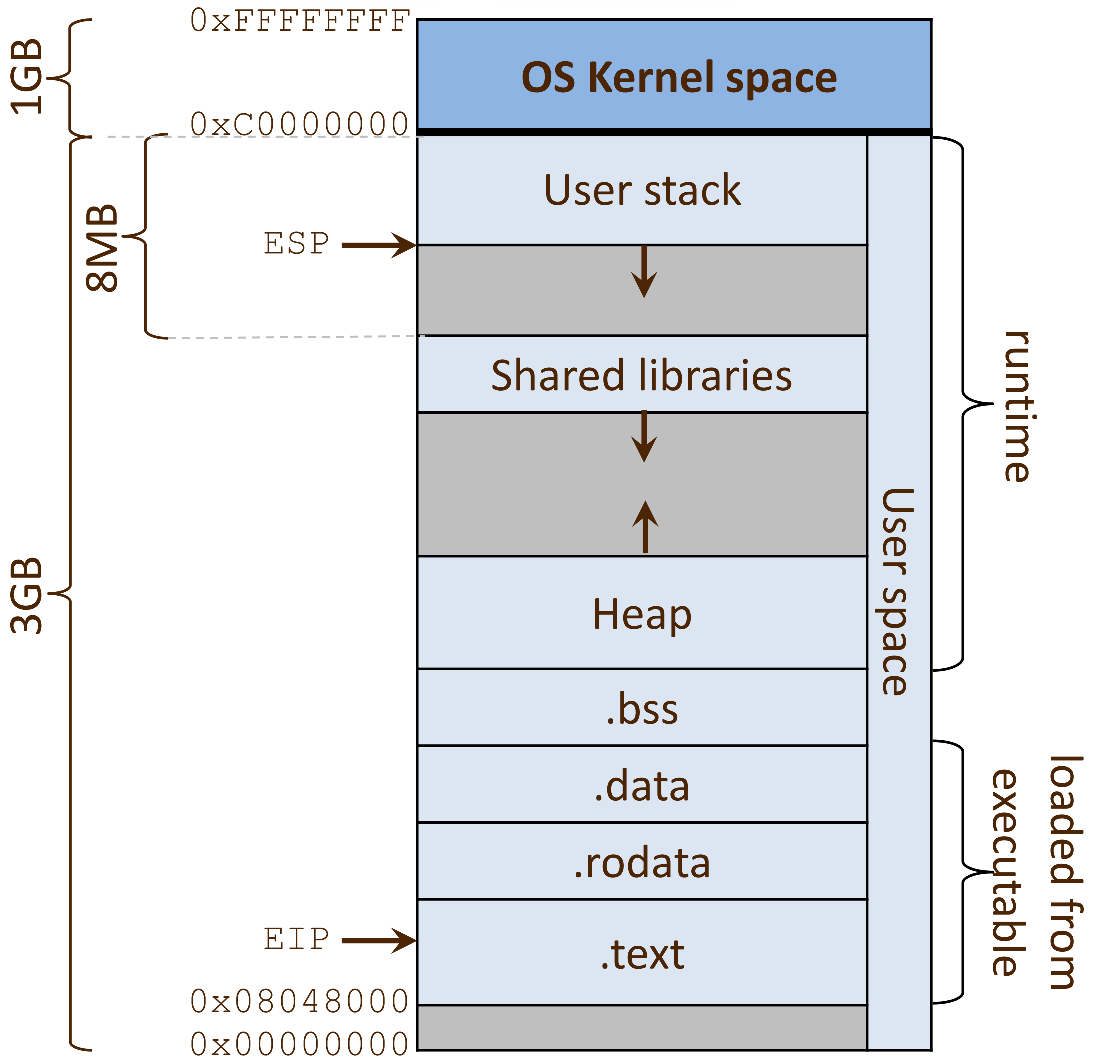
Memory segments

- Dynamic memory (heap)
  Whole runtime allocated memory fields. It grows toward the large memory addresses. See `malloc`, `calloc`, `realloc`.
- Shared memory
  Space for memory-mapped files and dynamically loaded built-in function libraries. For example, the program code of `printf`.

Memory segments

- Stack segment
  Area with LIFO access, which grows toward the lower addresses. The top of the stack (lowest address) is stored in SP register.
  For example: dynamic lifetime (local) variables, subroutine return addresses, parameters, temporarily saved refister contets, etc. Its size is limited. Frequent access (cache).
- There are further empty and reserved fields in the address space.

Address space of a program in Linux



Further thought-provoking questions

Answers are deeply discussed in
„**Assembly programming**”
subject

Thought-provoking questions

- What is the difference between the application of the two symbolic constants in C?

```
#define ABC 123
const int ABC=123;
```

- Which runtime condition evaluation is faster? Who/When does the evaluation of expressions?

```
int a=1, b=2, c=3, d=2, e=1;
if(0>a+b*c||d%e)…
if(0>a+b*c&&d%e)…
if(0>1+2*3&&2%1)…
if(0)…
```

Thought-provoking questions

- A procedure can be called at several point of a program code. How the procedure knows where to return?
- Technically, How the parameters pass from the caller to the callee subroutine?
- Where are the local variables stored in RAM? Who/when allocates space for them and who/when free it?

Thought-provoking questions

- Which assignment instruction is faster? Why?

```
double myabs1(double x){
  return (x)<0?-1*(x):(x);}

#define myabs2(x)  (x)<0?-1*(x):(x)


int main(){
  int a1,a2,b=-23;
  a1=myabs1(b);
  a2=myabs2(b);
  return 0;}
```

Thought-provoking questions

- What happens during the execution of these programs? What are their return values?

```
int main(){
   int T[10000000];
   T[0]=123;
   return T[0]; }
```

```
void one(int a){
   if(a>0) one(a-1); }
int main(){
   one(1000000);
   return 0; }
```

Thought-provoking questions

- What is the return value of the main?

```
int a=1;
void first(){
  int b=2; }
void second(){
  int c;
  a=c+3; }
int main(){
  first();
  second();
  return a-4; }
```

References

- Sarah L. Harris, David M. Harris: Digital design and computer architecture (ARM edition), Morgan Kaufman, 2016
- Nicholas Charter:
  Computer architecture, McGraw-Hill, 2001
- David A Patterson, John L Hennessy:
  Computer organization and design, Morgan Kaufman, 2014
- 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

ThanQ
for your attention!

Have a successful study!

This work was supported by the construction **EFOP-3.4.3-16-2016-00021**.
The project was supported by the European Union, co-financed by the European Social Fund.