# Computer Systems

Introduction

Jakub Yaghob

# Literature and slides

- Web page and slides
  - https://www.ksi.mff.cuni.cz/teaching/nswi170-web/
- Books
  - Silberschatz A.: Operating Systems Concepts, Willey
  - Hennessy J.L.: Computer Architecture: A Quantitative Approach, Morgan Kaufmann

# Course

- Lectures
  - Weekly
  - Exam
    - Short written test
    - Programming task for Arduino
- Labs
  - Playing with Arduino with an added shield
  - Biweekly, assignments, home assignment
  - Upload your assignment to the SIS module Study Group Roster
  - **Borrow your Arduino in the library!**

# Course content

- Content
  - C language
  - CPU
    - Architecture
    - Instruction set
    - Interrupt, DMA
  - Memory
    - Addressing, alignment
    - Memory hierarchy, cache
  - Programming languages
    - Compilation, linking, memory organization
    - Function calls, parameter passing
    - Heap, runtime, JIT
  - Operating systems
    - Architecture, process, thread, scheduling
    - Virtual memory
  - Parallel programming
    - Synchronization

# Computer Systems

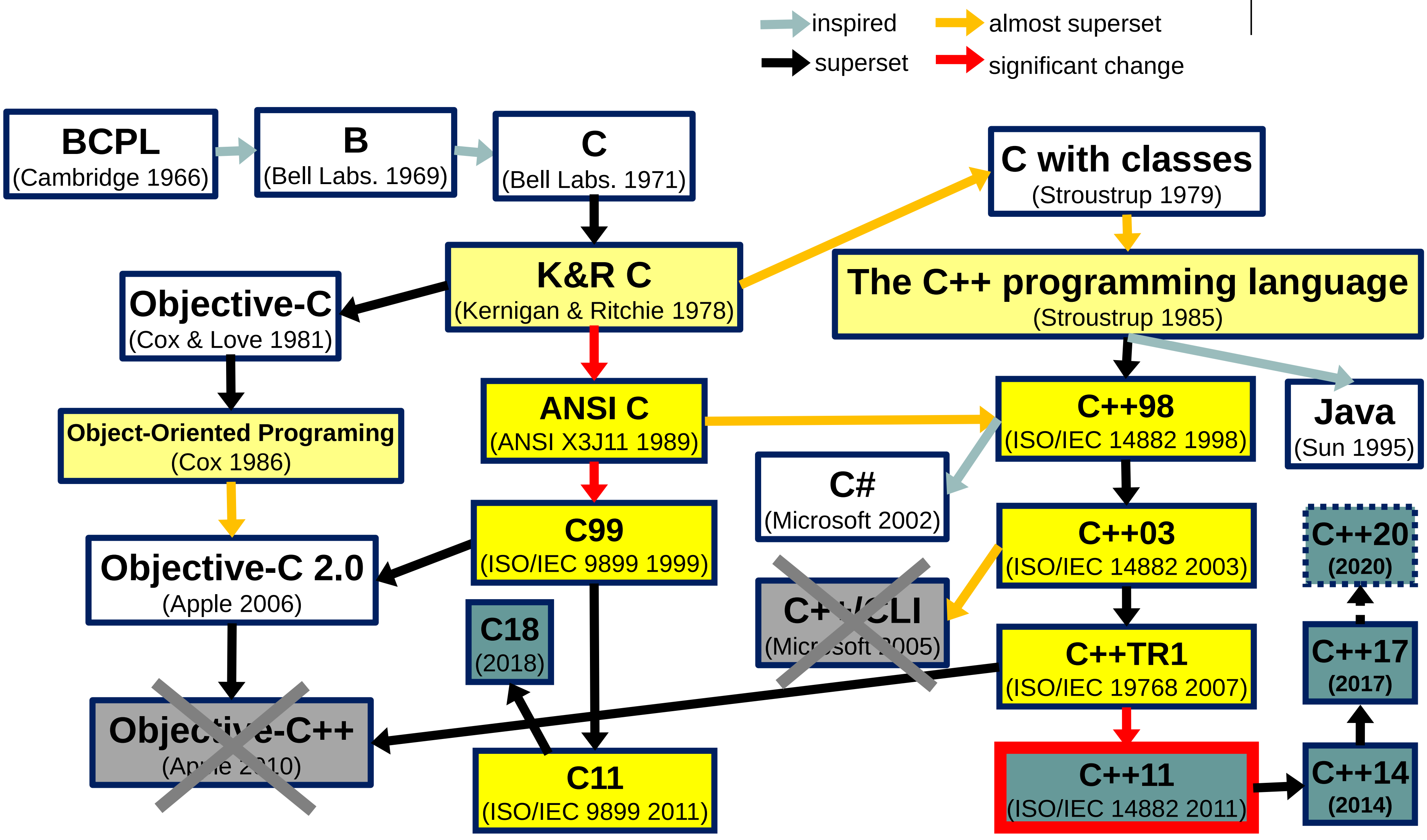
C/C++ language

Jakub Yaghob

# Features

- Procedural programming language
- Structured, imperative programming
- Recursion
- Static type system
- C constructs map efficiently to machine instructions
  - Operating systems
  - HPC
  - Embedded systems
- Case sensitive, ignore all whitespaces

# History



inspired
almost superset
superset
significant change
BCPL (Cambridge 1966)
B (Bell Labs. 1969)
C (Bell Labs. 1971)
C with classes (Stroustrup 1979)
K&R C (Kernigan & Ritchie 1978)
Objective-C (Cox & Love 1981)
The C++ programming language (Stroustrup 1985)
ANSI C (ANSI X3J11 1989)
Object-Oriented Programing (Cox 1986)
C++98 (ISO/IEC 14882 1998)
Java (Sun 1995)
C# (Microsoft 2002)
C99 (ISO/IEC 9899 1999)
Objective-C 2.0 (Apple 2006)
C++03 (ISO/IEC 14882 2003)
C++20 (2020)
C18 (2018)
C++/CLI (Microsoft 2005)
C++TR1 (ISO/IEC 19768 2007)
C++17 (2017)
Objective-C++ (Apple 2010)
C11 (ISO/IEC 9899 2011)
C++11 (ISO/IEC 14882 2011)
C++14 (2014)

# Example

```
/* this is my best program */
#include <stdio.h>

int x;      // global variable

int f(int p) {  // function
  int q = p+x;// local variable
  return q;
}
```

# Constants

- Integer numbers
  - Decimal number

    `123`, `-18`
  - Hexadecimal number

    `0x7a`
- Floating point number

  `-1.234e-5`
- String

  `"foo bar"`
- Char

  `'a'`
- Escape sequence
  - `\n` – LF
  - `\r` – CR
  - `\t` – TAB
  - `\\` – \
  - `\'` – ’
  - `\"` – ”
  - `\xab` – char 0xab

# Basic types

- Integer types
  - Base

    **`char`**, **`int`**
  - Modifiers

    **`short`**, **`long`**

    **`signed`**, **`unsigned`**
  - Auxiliary

    **`size_t`**
- Floating point types

  **`float`**, **`double`**
- Other types

  **`void`**, **`bool`**
- Implicit conversion
  - Conversion rank

# Statements

- Compound statement (block)

```
{ }
```

- Expression statement

```
expr ;
```

- If statement

```
if (expr) stmt
if (expr) stmt else stmt
```

- Return from a function

```
return expr;
```

# Statements – switch

```
switch (expr) {
case 0:
  // something
  break;
case 1:
  // something else
  break;
case 2:
case 3:
  // common code for 2 and 3
  break;
default:
  // do something else otherwise
  break;
}
```

# Statements – loops

- While

```
while (expr) stmt
```

- Do-while

```
do stmt while (expr);
```

- For

```
for(expri ; exprt ; exprs) stmt
```

- Jumps

```
break;
continue;
```

# Expression

- Arithmetic
  `+`, `-`, `*`, `/`, `%`
  - No `//`

  `++`, `--`
- Comparison
  `<`, `<=`, `>`, `>=`, `==`, `!=`
- Bitwise
  `~`, `&`, `|`, `^`, `<<`, `>>`
- Logical
  `&&`, `||`, `!`
- Pointers
  `&`, `*`
- Assignment
  `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`
- Variable/type size
  `sizeof`
- Ternary expr
  `test ? e1 : e2`

# Variables

- A named value stored in a memory
- Must be declared before initialization and using
- Variable scope
- Storage class

```
int i, j;
int c = 42;
static int s = 0;
```

# Array

- Collection of elements each identified by at least one index
- Contiguous area of memory
- Constant size
- Correct alignment
- Row-major order
- Zero based index

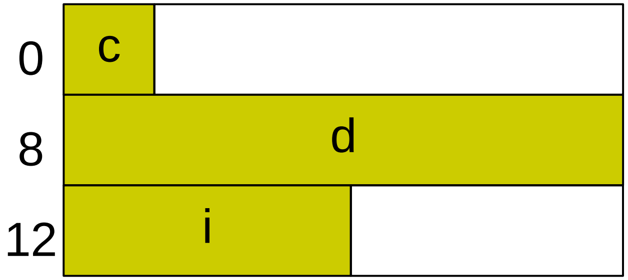
```c
int u[4];
int p[] = { 1, 2, 3 };
int a[2][3] =  { { 1, 2, 3 }, { 4, 5, 6 } };
```

| 0, 0 | 0, 1 | 0, 2 | 1, 0 | 1, 1 | 1, 2 |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

# Structure

- Collection of fields (members)
- Inner alignment (padding)
- Outer alignment (padding)

```
struct point2d { int x, y; }
struct data {
    char c;
    double d;
    int i;
};
```



# Remnants

- Constants

```
#define C 13
constexpr int C = 13;
```

- Enumerated type

```
enum e { RED, BLUE, GREEN };
```

- Automatic type
  - Type inferred from an initialization expression

```
auto a = 3;
```

- Importing a module

```
#include <system.h>
```

# Starting point

- Always function main
  - Return value is an exit code
  - Without return 0 exit code assumed
- Basic version

```
int main() { }
```

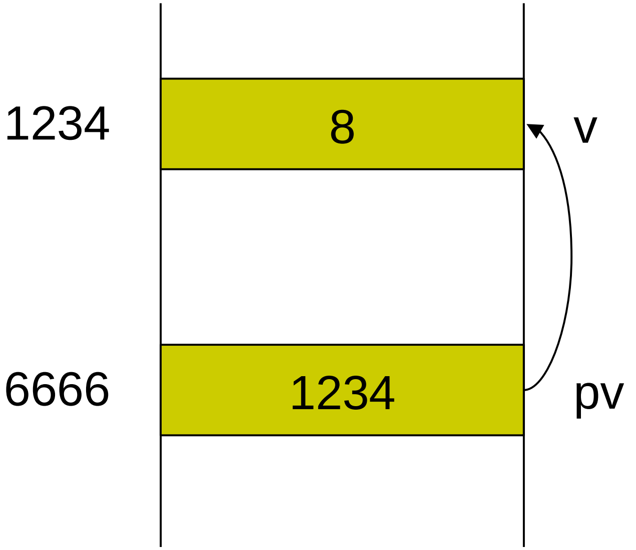
- Advanced version

```
int main(int argc, char **argv) { }
```

# Pointer

- Each variable somewhere in memory
- Address
- A variable holding an address = pointer

```
int v = 8;
int *pv = &v;
*pv = 4;
```



# Functions, parameter passing – C

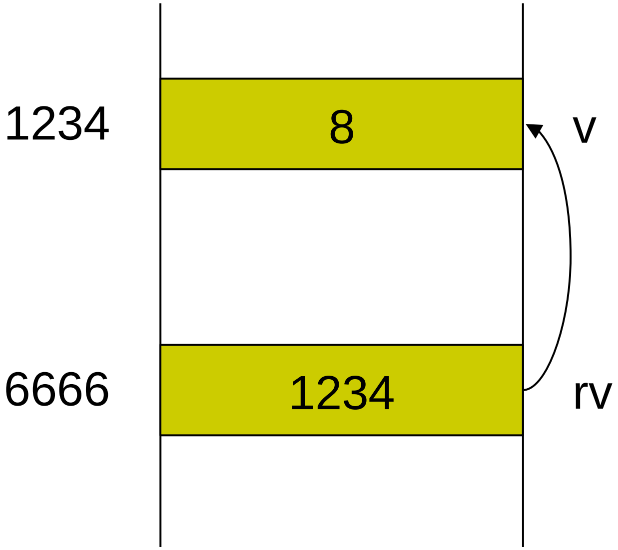
- Parameters in C always passed by value
- Output parameters use pointer

```
void pvec(point2d in, point2d *out)
{
  out->x = in.y;
  out->y = in.x;
}
```

# Reference

- Fixed pointer
  - Address not reassignable

```
int v = 8;
int &rv = v;
rv = 4;
```



# Functions, parameter passing – C++

- Parameters in C++ passed by value or by reference
- Output parameters by reference

```
void pvec(point2d in, point2d &out)
{
  out.x = in.y;
  out.y = in.x;
}
```

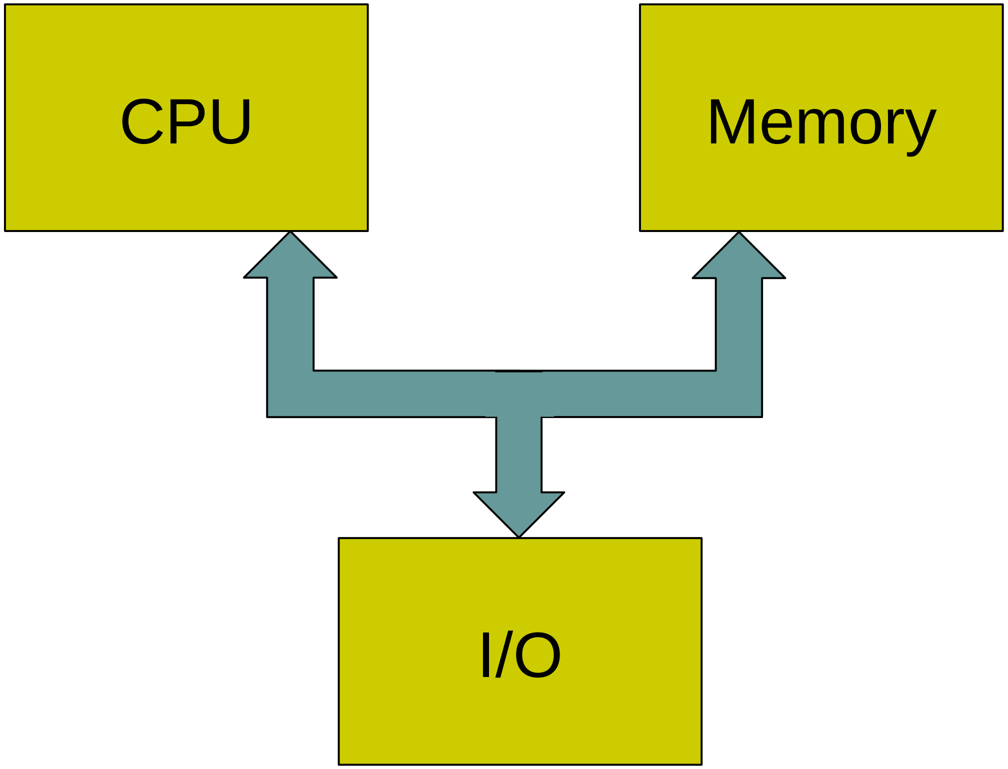
# Computer Systems

CPU

Jakub Yaghob

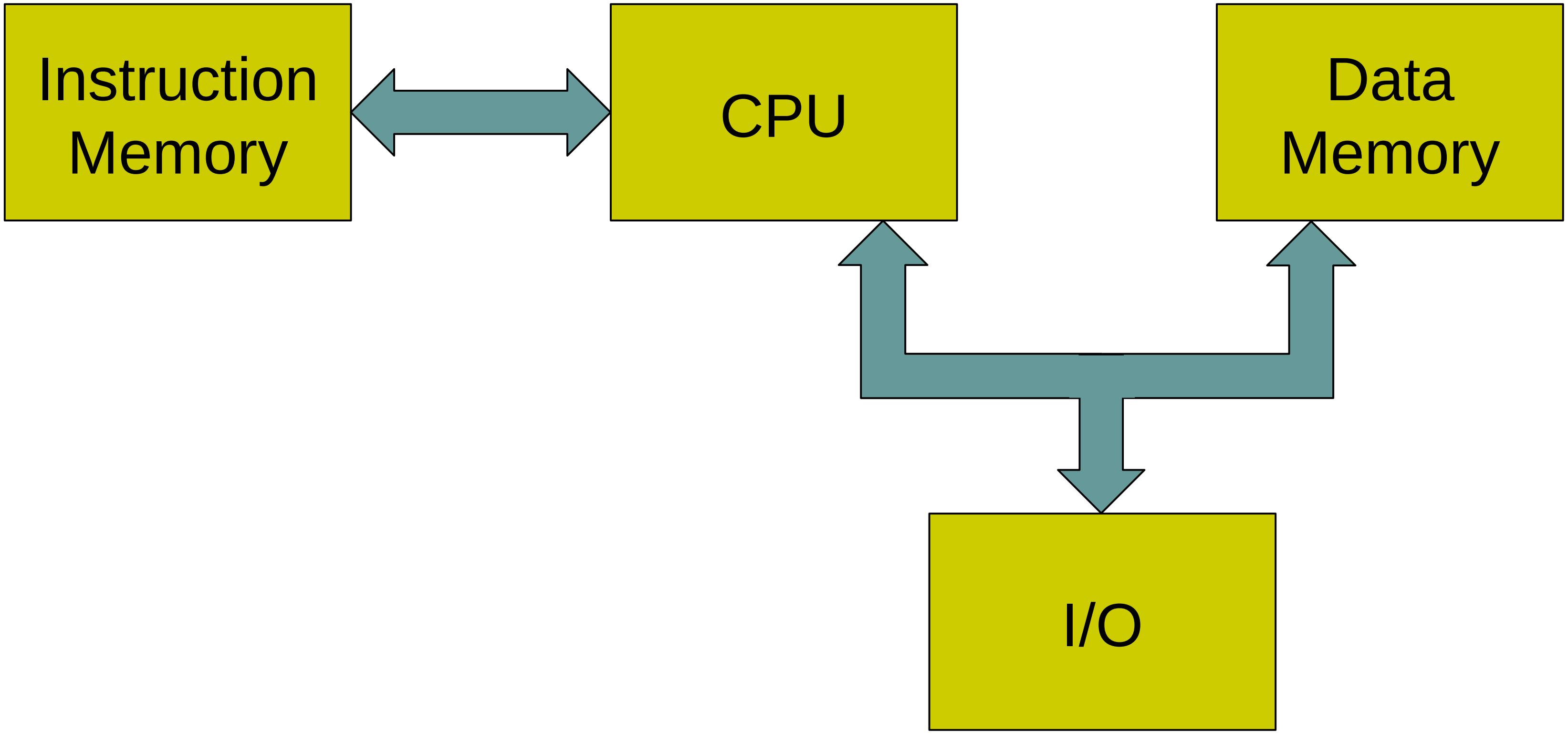
# Von Neumann architecture

- Simple, slower



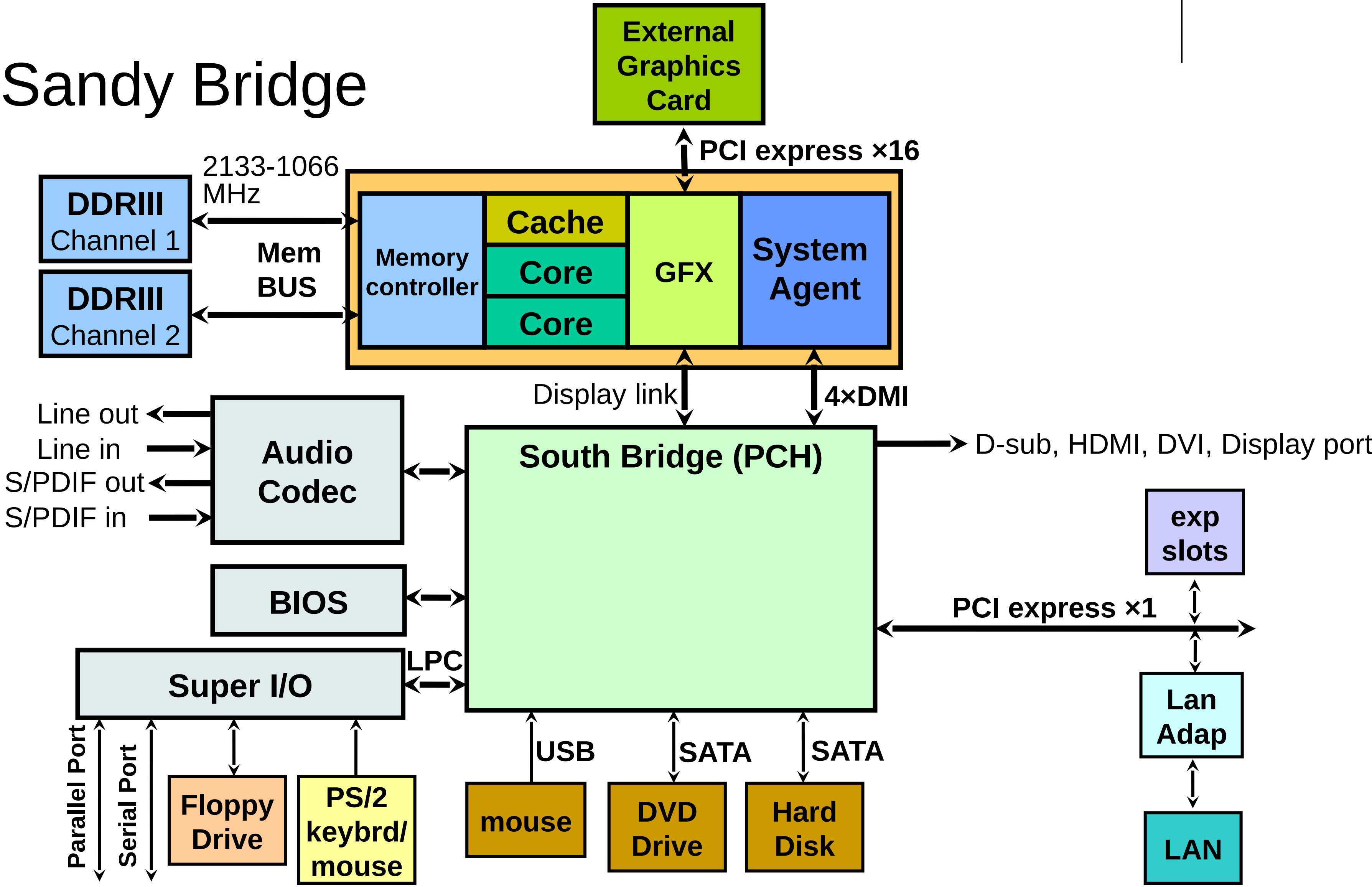
# Harvard architecture

- Microcontrollers
- Multiple address spaces



# Real PC architecture

- Sandy Bridge



# CPU

- Architecture
  - HW
  - ISA
- "Simple" machine
  - Executes instructions
    - Instruction – simple command

# Instructions - motivation

- How can we execute the following code?

```
if(a<3) b = 4; else c = a << 2;

for(int i=0;i<5;++i) a[i] = i;

int f(int p) { return p+1; }
void g() { auto r = f(42); }
```

# Instruction classes

- Load instructions
- Store instructions
- Move instruction
- Arithmetic and logic instructions
- Jumps
  - Unconditional x conditional
  - Direct x indirect x relative
- Call, return
- …

# Registers

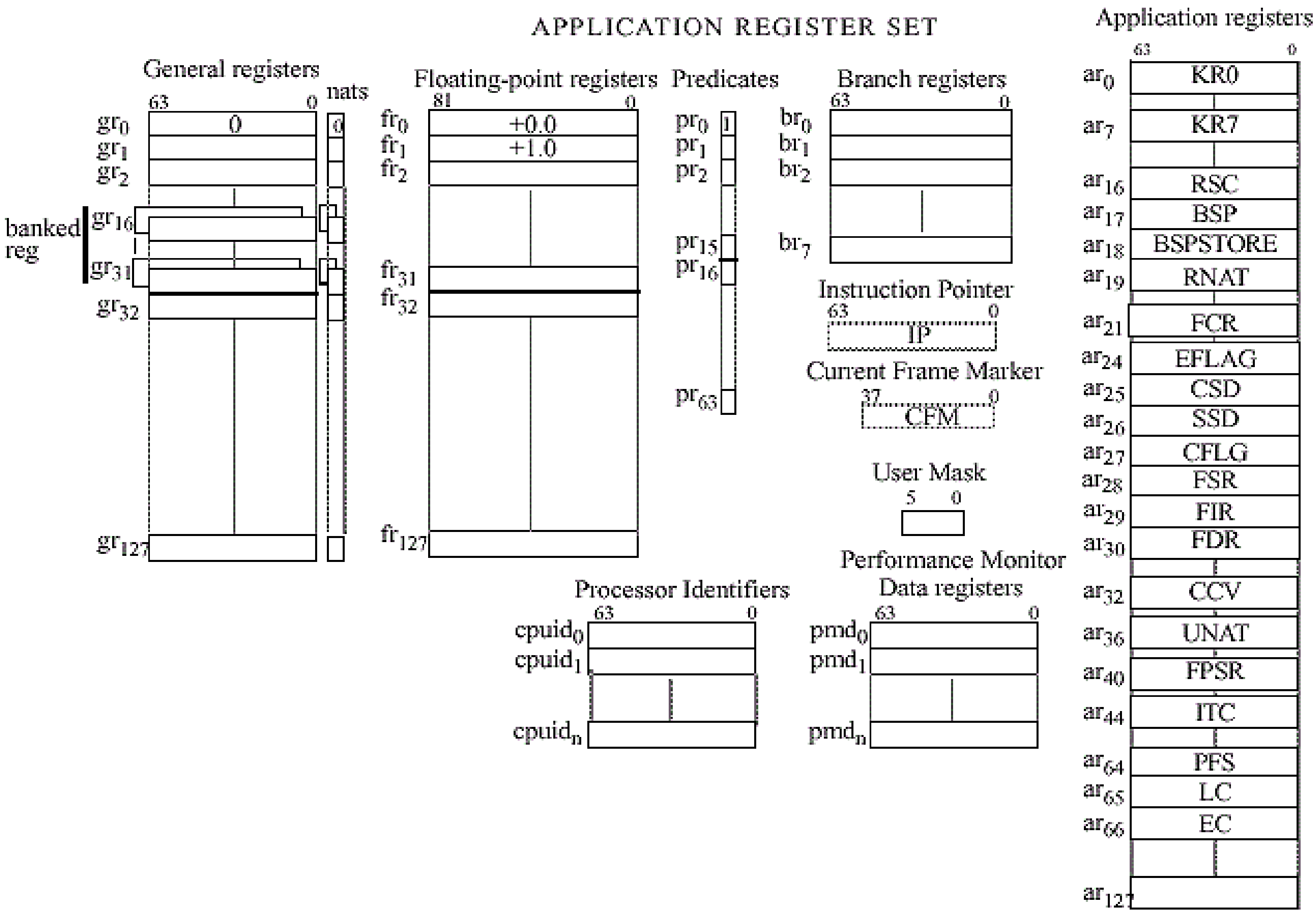
- Types
  - General, integer, floating point, address, branch, flags, predicate, application, system, vector, …
- Naming
  - Direct x stack
- Aliasing

# Registers – example 32-bit x86

| | | | |
|---|---|---|---|
| EAX | AX | AH | AL |
| EBX | BX | BH | BL |
| ECX | CX | CH | CL |
| EDX | DX | DH | DL |
| ESI | | SI | |
| EDI | | DI | |
| EBP | | BP | |
| ESP | | SP | |

| | | |
|---|---|---|
| CS | | |
| DS | | |
| ES | | |
| SS | | |
| FS | | |
| GS | | |
| EFLAGS | | FLAGS |
| EIP | | IP |

# Registers – example IA-64



# MIPS – simple assembler

- Execution environment
  - 32-bit registers r0-r31
    - r0 is always 0, writes are ignored
    - r31 is a link register for the `jal` instruction
  - No stack
  - No flags
  - PC register

# MIPS – register aliases

| Register | Name | Purpose | Preserve |
|---|---|---|---|
| $r0 | $zero | 0 | N/A |
| $r1 | $at | Assembler temporary | No |
| $r2-$r3 | $v0-$v1 | Return value | No |
| $r4-$r7 | $a0-$a3 | Function arguments | No |
| $r8-$r15 | $t0-$t7 | Temporaries | No |
| $r16-$r23 | $s0-$s7 | Saved temporaries | Yes |
| $r24-$r25 | $t8-$t9 | Temporaries | No |
| $r26-$r27 | $k0-$k1 | Kernel registers – DO NOT USE | N/A |
| $r28 | $gp | Global pointer | Yes |
| $r29 | $sp | Stack pointer | Yes |
| $r30 | $fp | Frame pointer | Yes |
| $r31 | $ra | Return address | Yes |

# MIPS – instructions

- Arithmetic
  - `add $rd,$rs,$rt`
    - R[rd] = R[rs]+R[rt]
  - `addi $rd,$rs,imm16`
    - R[rd] = R[rs]+signext(imm16)
  - `sub $rd,$rs,$rt`
  - `subi $rd,$rs,imm16`

# ISA comparison

**MIPS**

```
ADD  $t1,$t1,$t0
ADDI $t1,$t1,1

ADD  $t2,$t0,$t1
```

**x86**

```
ADD eax,ebx
ADD eax,1

MOV eax,ebx
ADD eax,ecx
```

# MIPS – instructions

- Logic operations
  - `and`/`or`/`xor`/`nor $rd,$rs,$rt`
  - `andi`/`ori`/`xori $rd,$rs,imm16`
    - R[rd] = R[rs] and/or/xor zeroext(imm16)
  - No **not** instruction, use `nor $rd,$rs,$rs`
- Shifts
  - `sll`/`slr $rd,$rs,shamt`
    - R[rd] = R[rs] << / >> shamt
  - `sra $rd,$rs,shamt`

# ISA comparison

| MIPS | x86 |
|---|---|
| `NOR $t1,$t2` | `MOV eax,ebx` |
| | `NOT eax` |
| `SLL $t1,$t1,3` | `SHL eax,3` |

# MIPS – instructions

- Memory access
  - `lw $rd,imm16($rs)`
    - R[rd] = M[R[rs] + signext32(imm16)]
  - `sw $rt,imm16($rs)`
    - M[R[rs] + signext32(imm16)] = R[rt]
  - `lb $rd,imm16($rs)`
    - R[rd] = signext32(M[R[rs] + signext32(imm16)])
  - `lbu $rd,imm16($rs)`
    - R[rd] = zeroext32(M[R[rs] + signext32(imm16)])
  - `sb $rt,imm16($rs)`
    - M[R[rs] + signext32(imm16)] = R[rt]
- Moves
  - `li $rd,imm32`
    - R[rd] = imm32
  - `move $rd,$rs`
    - R[rd] = R[rs]

# ISA comparison

**MIPS**

```
LW   $t1,1234($t0)
SW   $t1,1234($t0)
LB   $t1,1234($t0)
LI   $t1,5678
MOVE $t1,$t0
```

**x86**

```
MOV eax,[ebx+1234]
MOV [ebx+1234],eax
MOV al,[ebx+1234]
MOV eax,5678
MOV eax,ebx
```

# MIPS – instructions

- Jumps
  - **`j addr`**
    - PC = addr
  - **`jr $rs`**
    - PC = R[rs]
  - **`jal addr`**
    - R[31] = PC+4; PC = addr

# ISA comparison

**MIPS**

```
J    label
JR   $ra

JAL fnc
```

**x86**

```
JMP label1
JMP [ebx]

CALL fnc
```

# MIPS – instructions

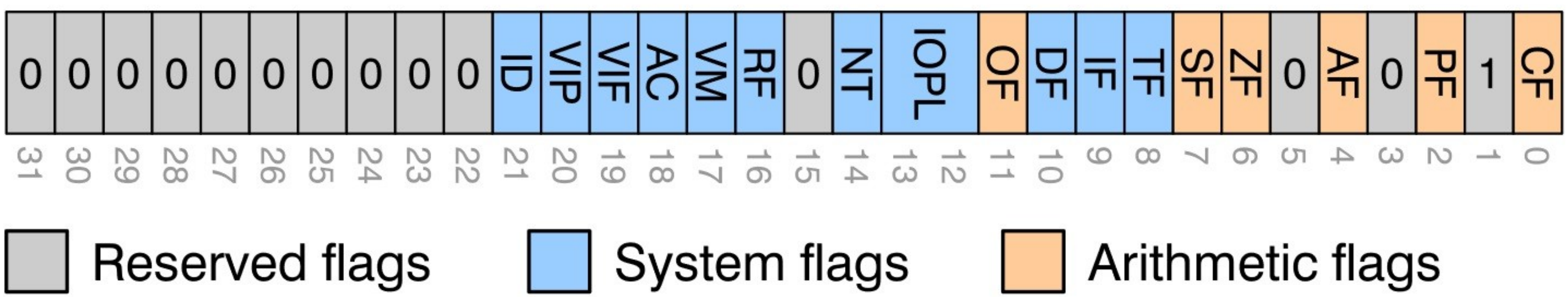
- Conditional jumps
  - **`beq $rs,$rt,addr`**
    - If R[rs]=R[rt] then PC=addr else PC=PC+4
  - **`bne $rs,$rt,addr`**
- Testing
  - **`slt $rd,$rs,$rt`**
    - If R[rs]<R[rt] then R[rd] = 1 else R[rd] = 0
  - **`sltu $rd,$rs,$rt`**
    - Unsigned version
  - **`slti $rd,$rs,imm16`**
    - If R[rs]<signext(imm16) then R[rd] = 1 else R[rd] = 0
  - **`sltiu $rd,$rs,imm16`**
    - If R[rs]<zeroext(imm16) then R[rd] = 1 else R[rd] = 0

# ISA comparison

| MIPS | x86 |
|---|---|
| `BEQ $t0,$t1,label` | `CMP eax,ebx`<br>`JZ  label` |
| `SLT $t2,$t1,$t0`<br>`BNE $t2,$zero,label` | `CMP eax,ebx`<br>`JL label` |
| `SLTI $t2,$t1,5`<br>`BNE $t2,$zero,label` | `CMP eax,5`<br>`JL label` |

# Flags

- Only used by some ISA
- Control execution
- Check status of the last instruction
- Usual flags
  - Z – zero flag
  - S – sign flag
  - C – carry flag



# CPU

- Architecture
  - Memory controller
  - Cache hierarchy
  - Core
  - Registers
    - Types
  - Logical processor
    - Hyper threading
  - Instructions

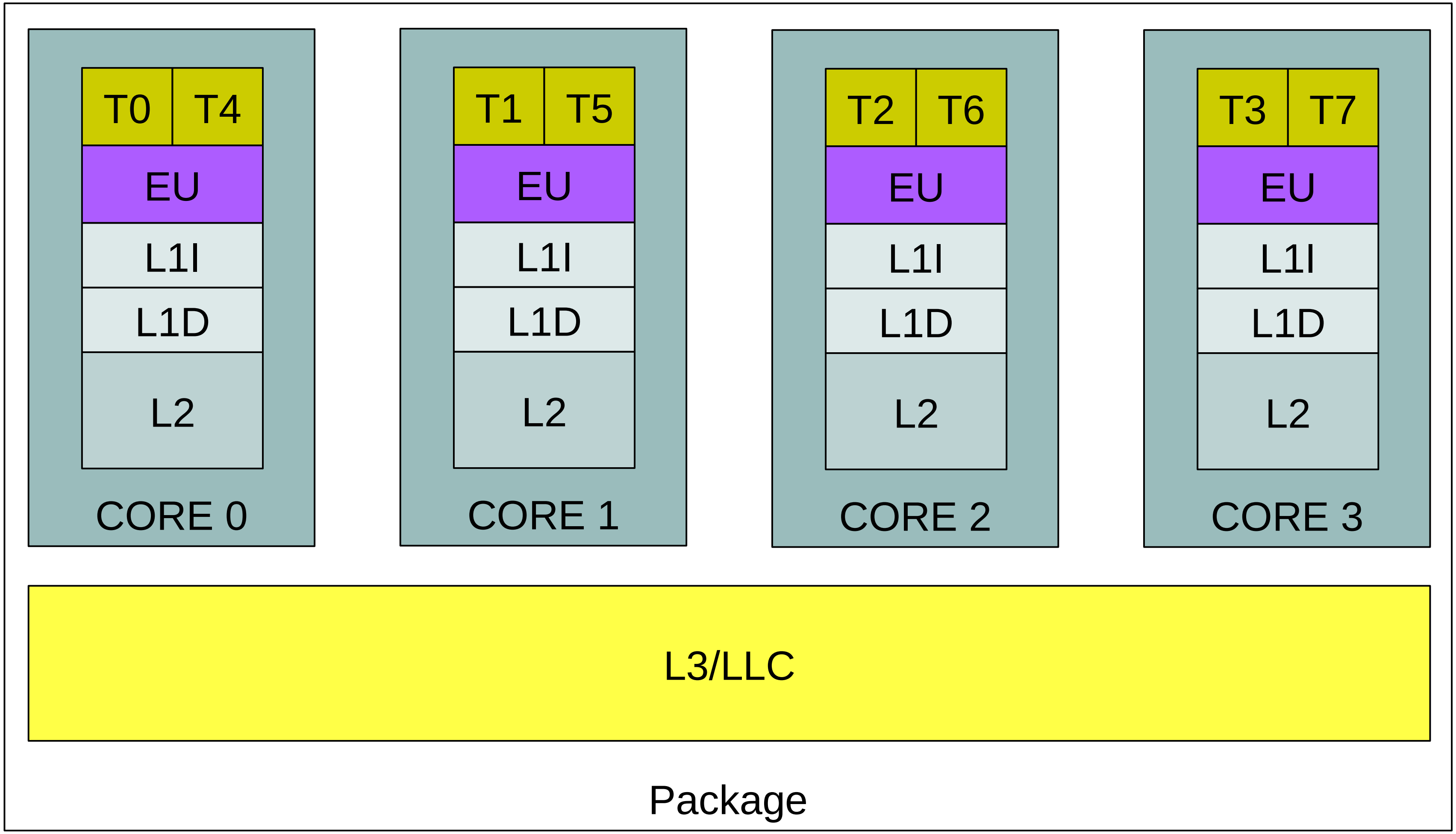
# Instruction

- Simple command to the CPU
- Encoding
- Assembler
- Operands
- Instruction flow
  - PC
- Stack?
  - SP

# ISA

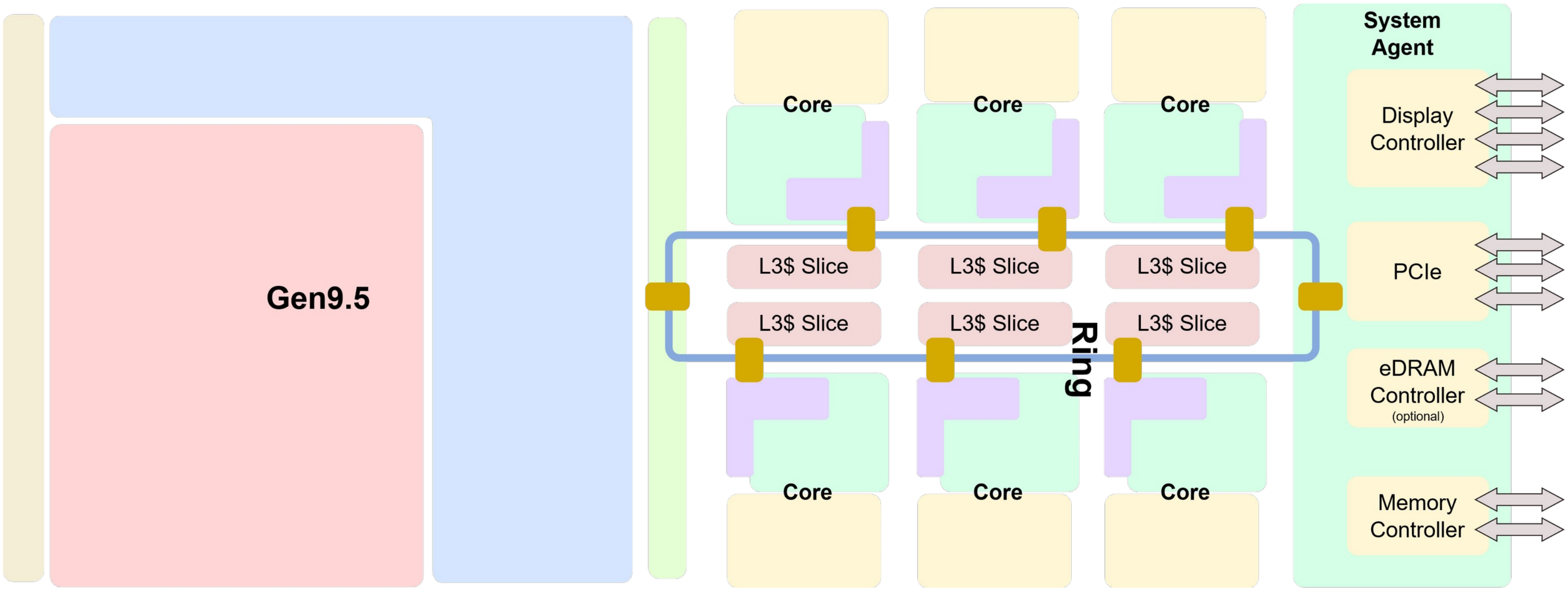
- Instruction set architecture
  - Abstract model of CPU
- Classification
  - CISC – Complex Instruction Set Computer
  - RISC – Reduced Instruction Set Computer
  - VLIW – Very Long Instruction Word
  - EPIC – Explicitly Parallel Instruction Computer
- Orthogonality
  - Accumulator
- Load-Execute-Store

# CPU – simplified scheme

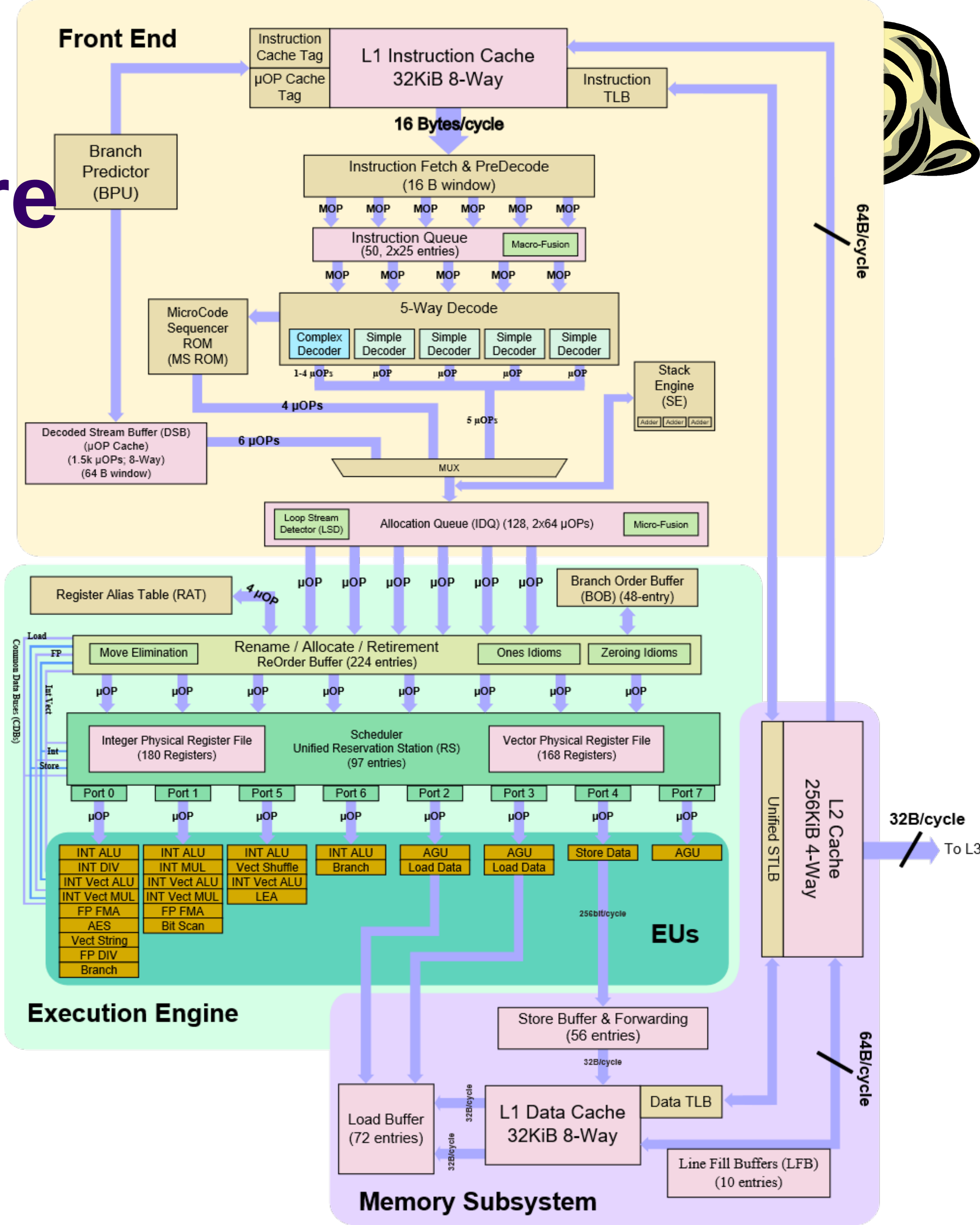


# Real CPU scheme – package

- Intel Coffee Lake

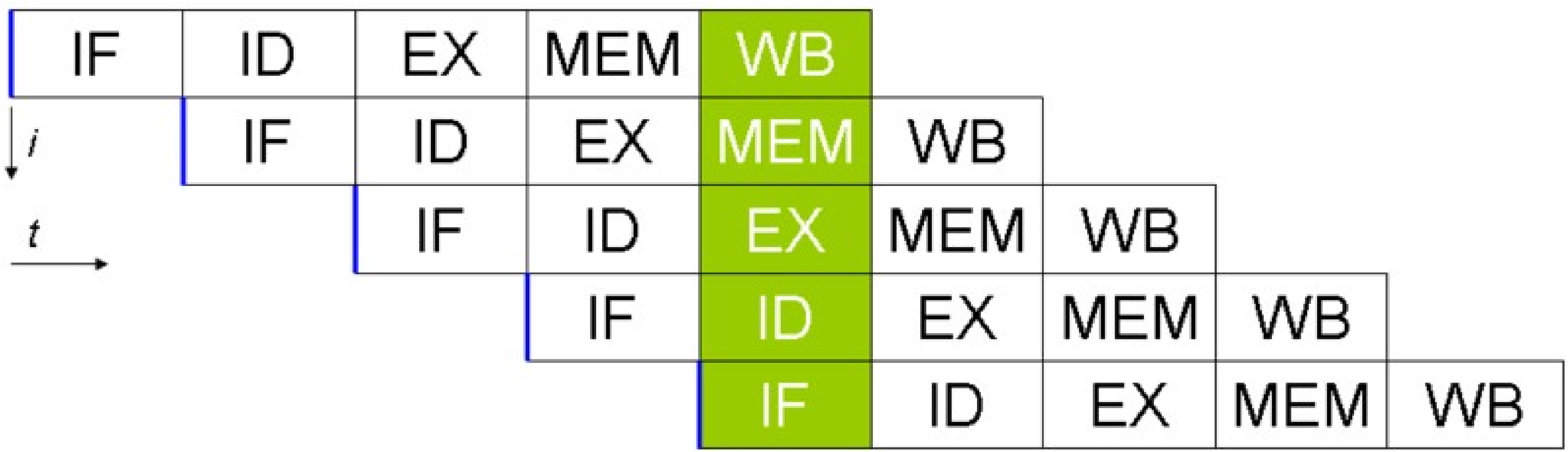


# Real CPU scheme – core



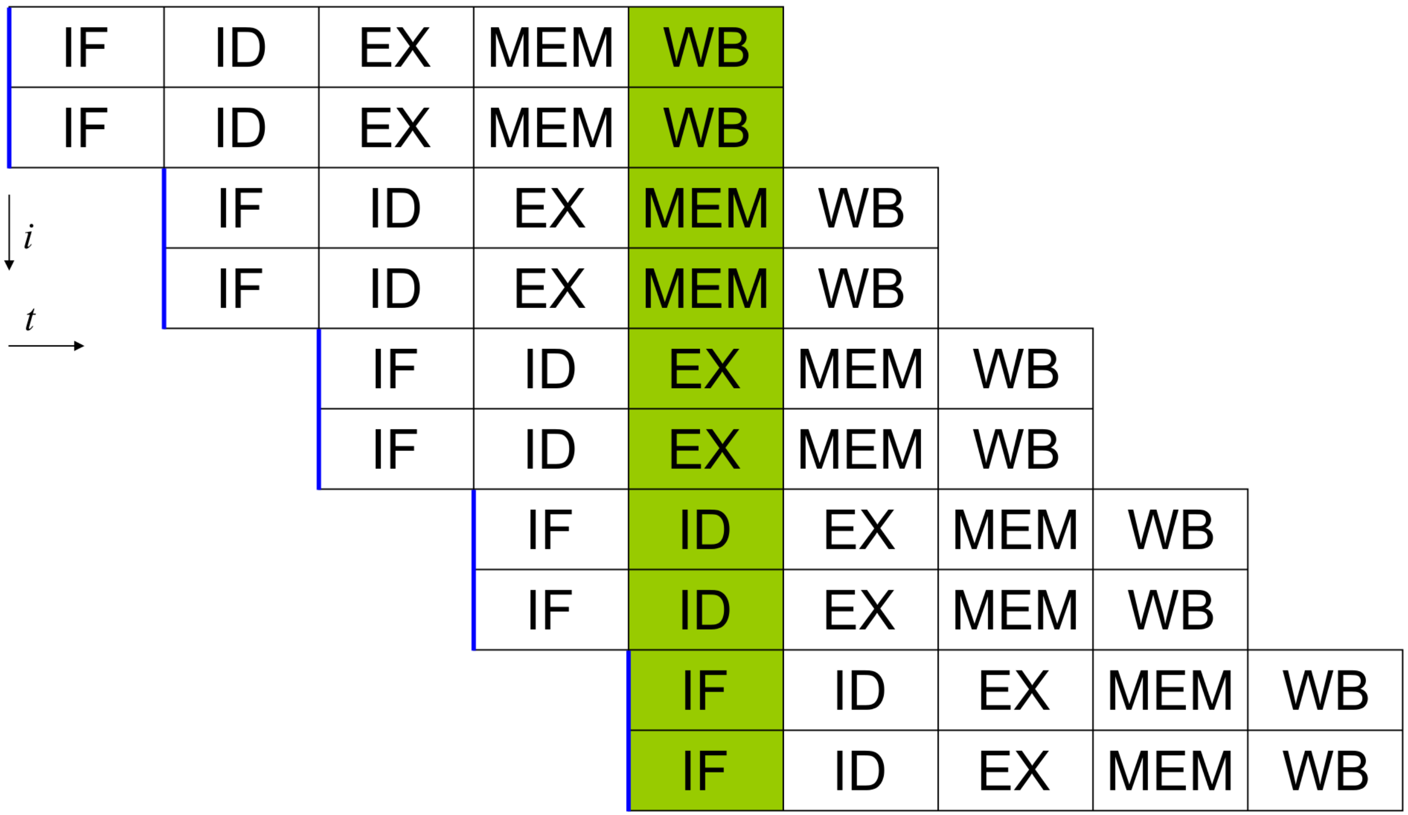
# CPU architecture – pipeline

- Current CPU
  - 14-19 stages

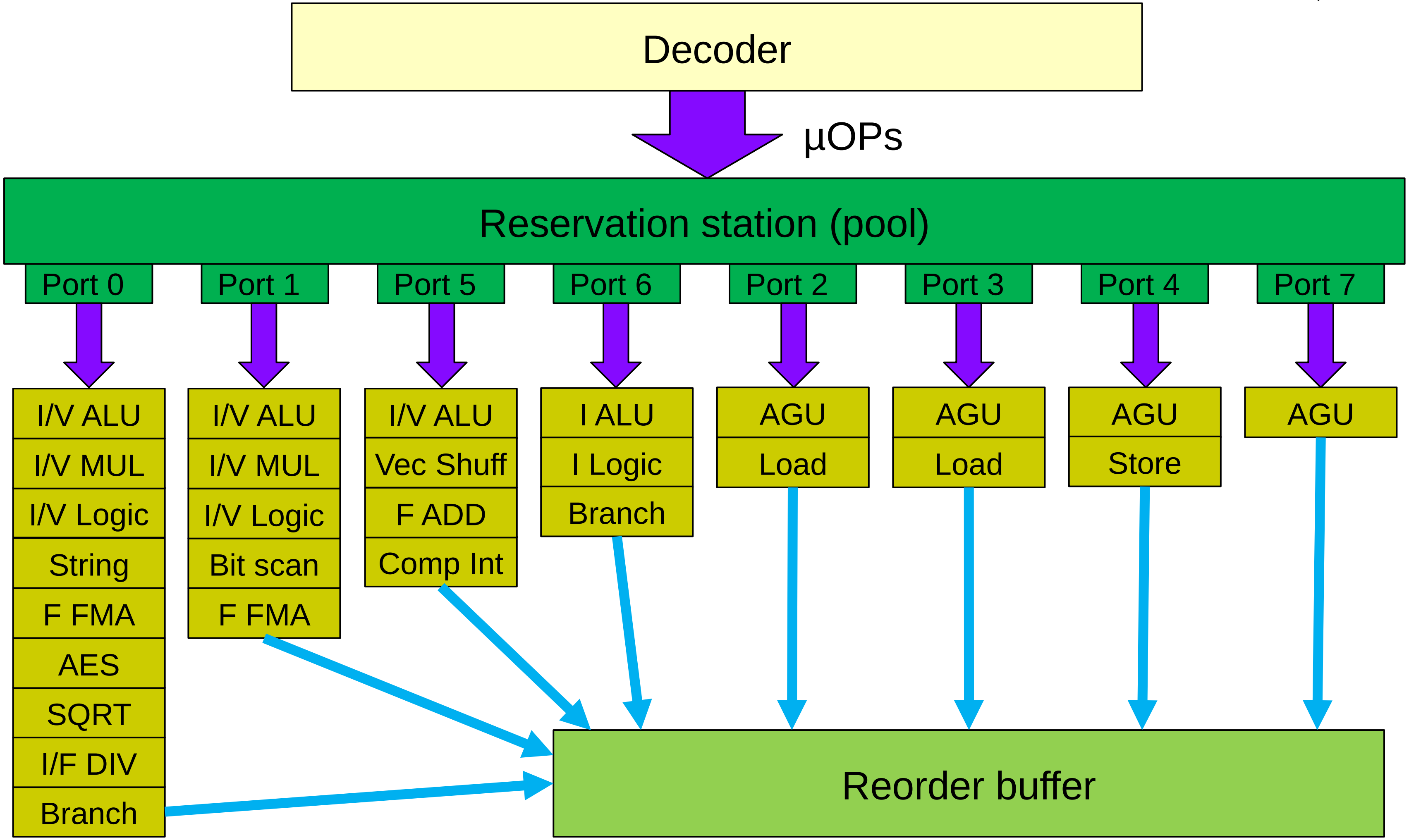


# CPU architecture – superscalar processor

- Current CPU
  - 5-way, asymmetric



# CPU architecture – out-of-order execution



# Computer Systems

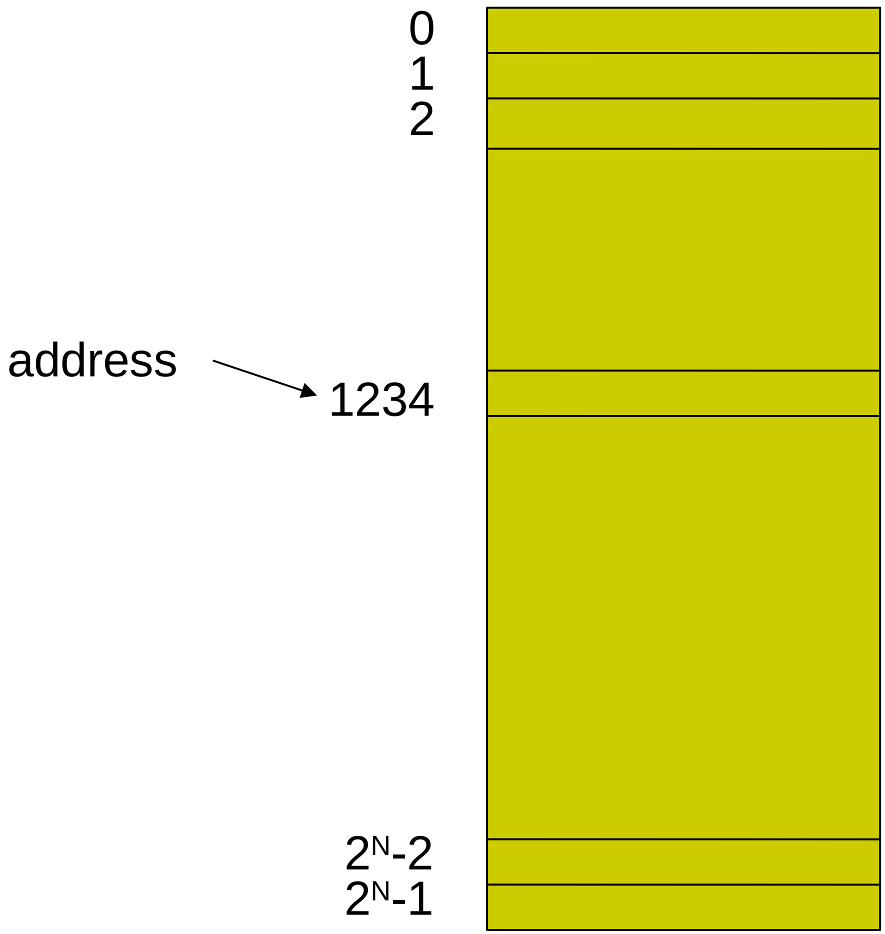
Memory

Jakub Yaghob

# Memory

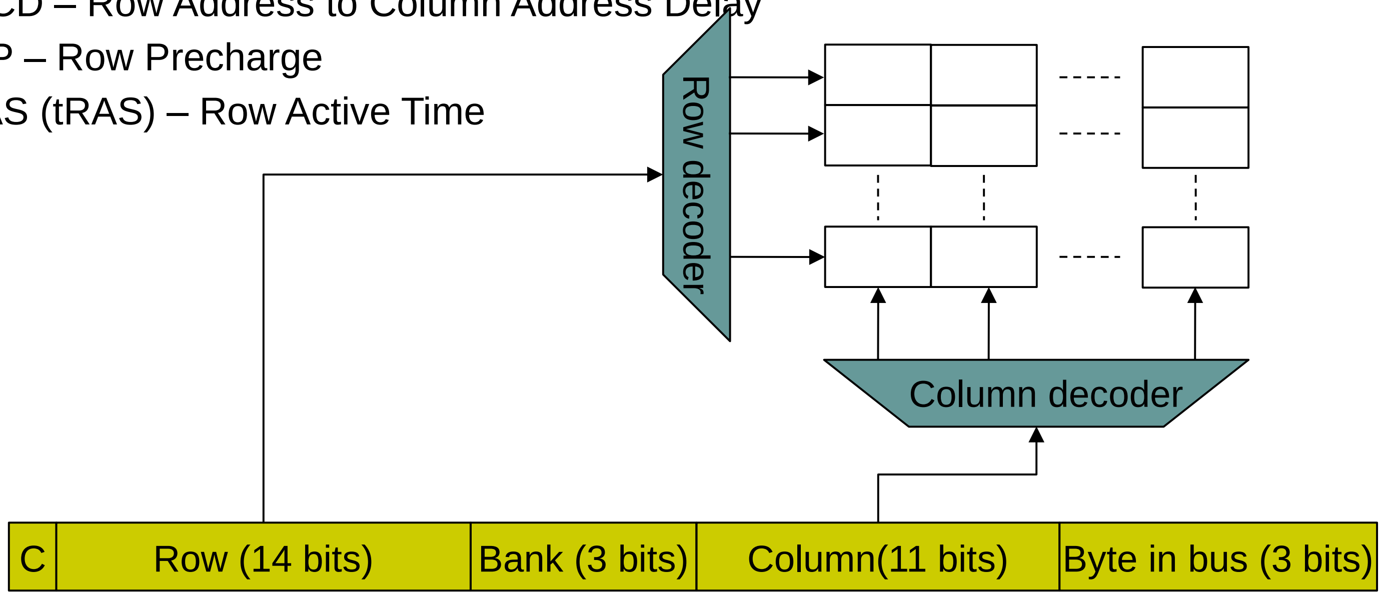
- Definition
  - Each memory organized into memory cells – bits
  - Bits are grouped into words of fixed length
    - 1, 2, 4, 8, 16, 32, 64, and 128 bits
  - Each word can be accessed by a binary address
    - N bits
    - We can store $2^N$ words in the memory
  - Today, the 8-bit word is used exclusively
    - Byte

# Memory – address space



# Memory – physical view

- 2D array
  - Row x column
  - Select, access, deselect row
  - Timing
    - CAS (tCL) – Column Access Strobe
    - tRCD – Row Address to Column Address Delay
    - tRP – Row Precharge
    - RAS (tRAS) – Row Active Time

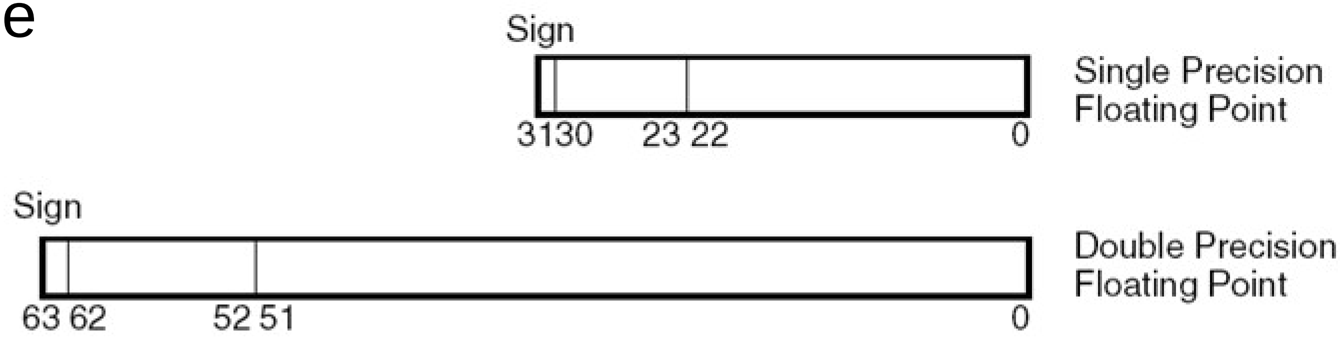


# Data representation – integer numbers

- Unsigned numbers
  - Simple binary representation of a number
  - Usual sizes
    - 1, 2, 4, 8 bytes
  - Represented range
    - $[0; 2^{N}-1]$
- Signed numbers
  - Two’s complement
  - Bitwise negation + 1
  - One 0
  - Compatible with unsigned arithmetic
  - Asymmetric range
    - $[-2^{N-1};2^{N-1}-1]$

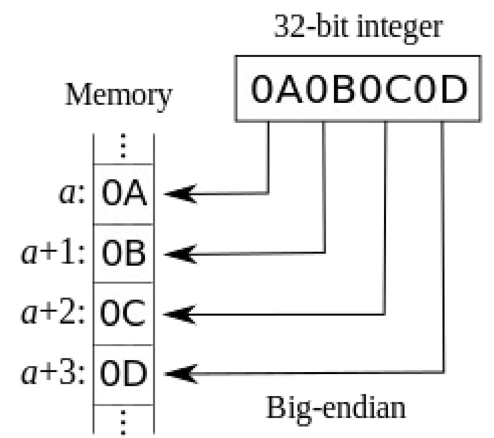
# Data representation – floating point numbers

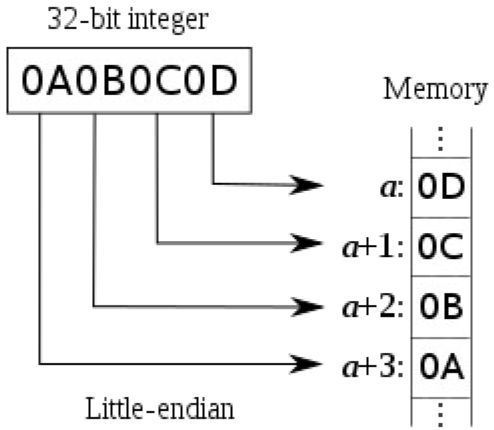
- IEEE 754
- Hidden bit convention
  - Memory representation for SP, DP
  - Use the smallest representable exponent
    - Hide leading bit of significand, it is always 1
- Exponent
  - Bias (FP=127, DP=1023)
  - Special values
- Value



# Data representation - endianess

- How to store multi-byte numbers?
- Big endian
  - MSB first, LSB last
  - PowerPC
- Little endian
  - LSB first, MSB last
  - Intel
- Example
  - Store 32-bit number 0x0A0B0C0D

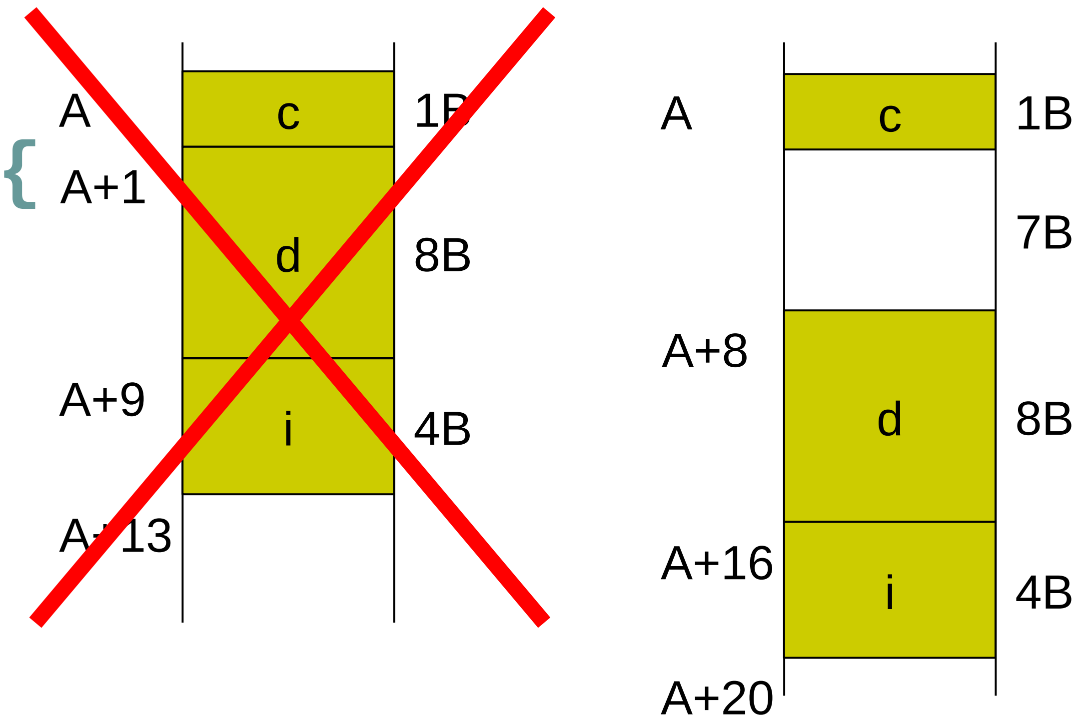


# Data alignment – inner padding

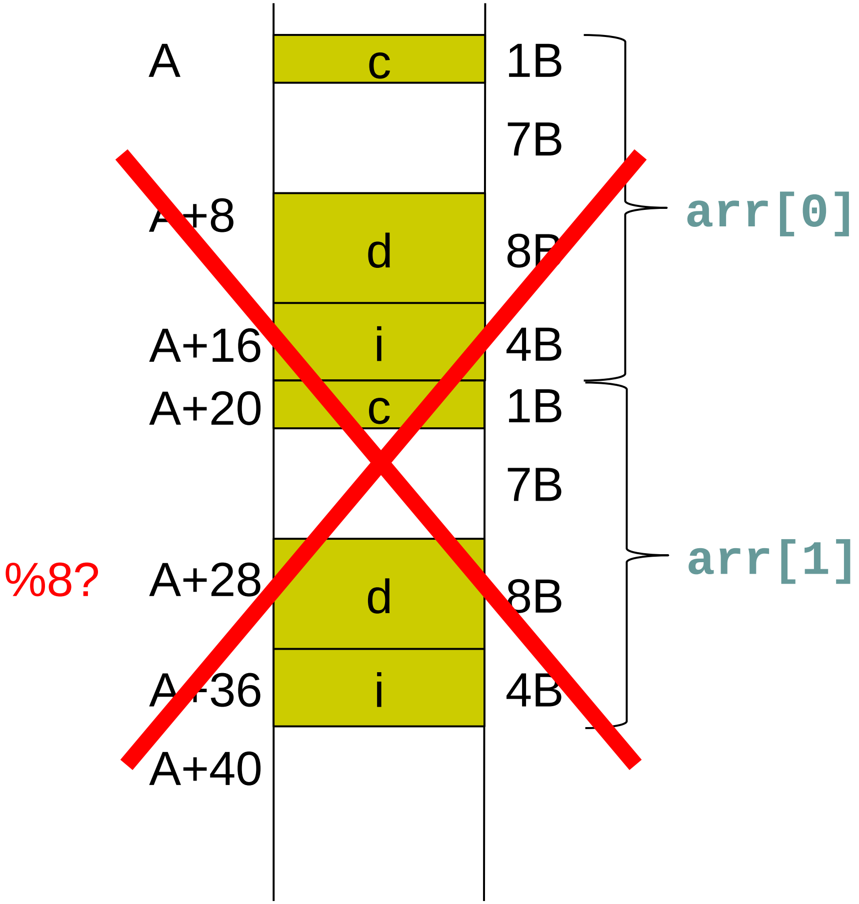
- Modern CPUs require data in memory aligned to their size
  - E.g. integer (4B) must have address aligned to 4

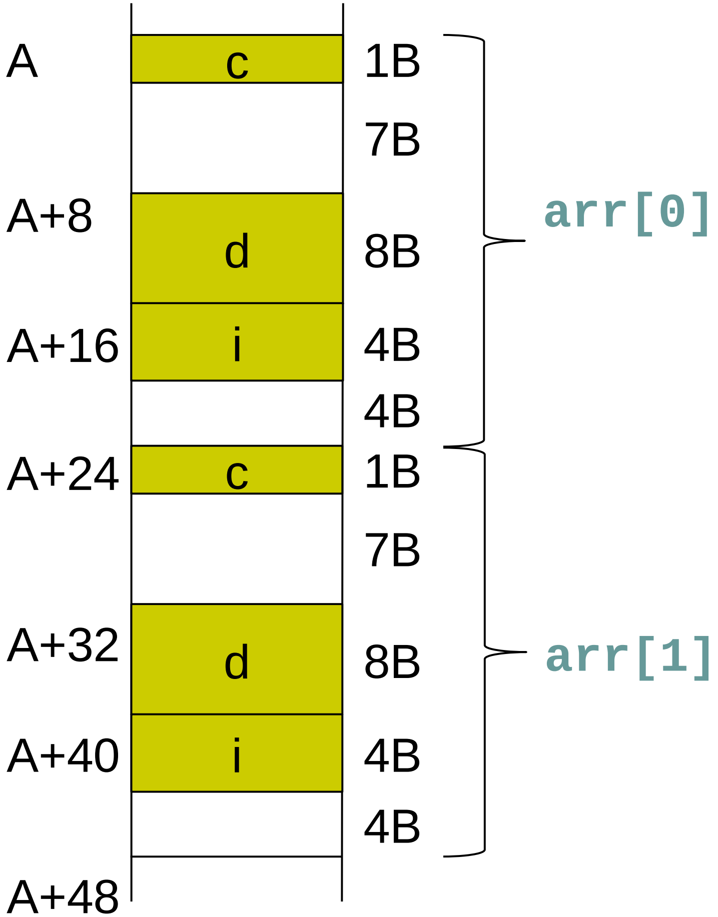
```
struct dem {
  char c;
  double d;
  int i;
};
```



# Data alignment – outer padding

```
dem arr[2];
```



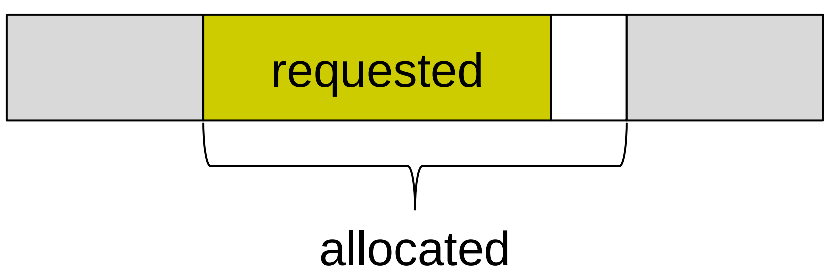



# Memory allocation

- Task
  - Locate a block of unused memory of sufficient size
  - Allocate portions from a large pool of memory
    - Heap, memory arena/pool
- Lifecycle
  - Allocate a block
    - Different strategies, allocators
  - Use the block
  - Free the block
    - Explicitly,  garbage collector

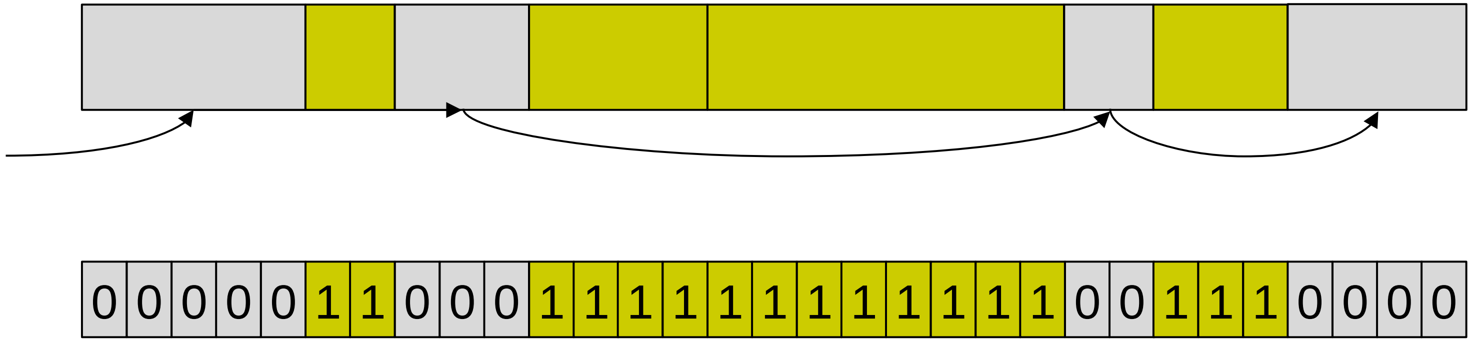
# Fragmentation

- Internal
  - Allocated more memory than needed in a block
- External
  - Free memory separated into small blocks and interspersed by allocated memory




# Dynamic memory allocation

- Contiguous allocation of variable size
- Free blocks evidence
  - Linked list
  - Bitmap
    - Each bit represents a block of a fixed size



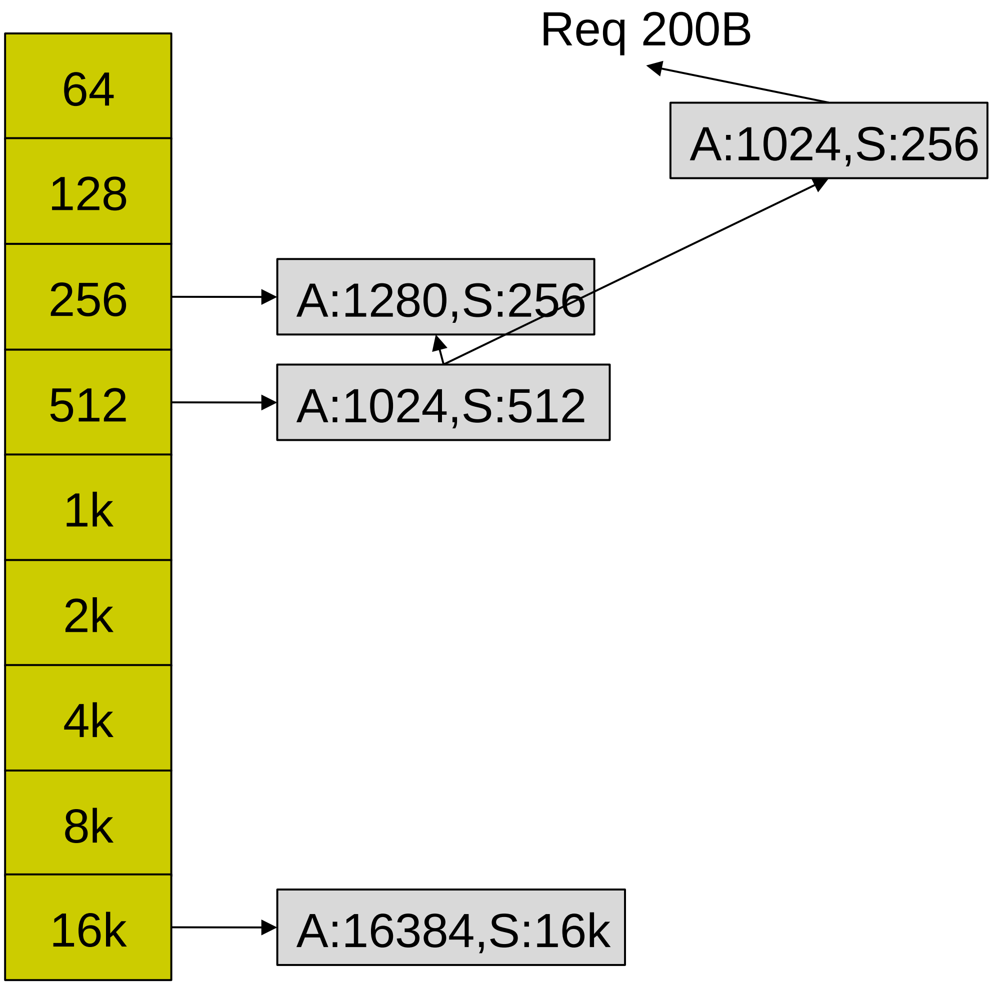
# Allocation algorithms

- First fit
  - Start from the beginning
  - Find the first free space big enough to accommodate required block size
  - Pros: fast, simple; Cons: can divide larger blocks
- Next fit
  - Like the first fit, but starts from the last position
  - Pros: fast, doesn't make fragmentation on the start of the heap
- Best fit
  - Start from the beginning, find the smallest space big enough
  - Pros: keeps large blocks; Cons: slower, creates many tiny blocks
- Worst fit
  - Start from the beginning, find the largest space
  - Cons: divides large blocks

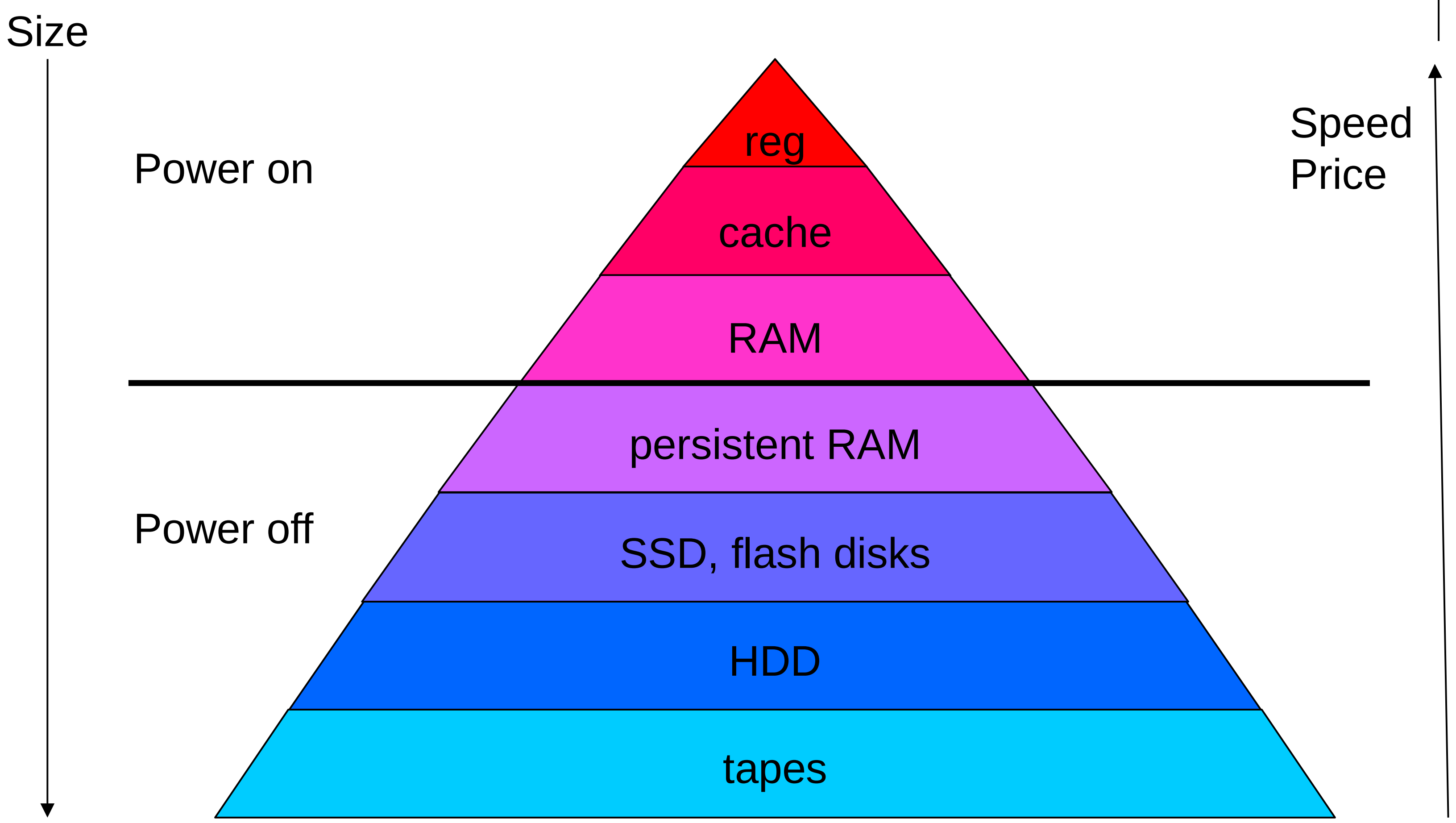
# Buddy memory allocation

- Blocks of $2^N$ size
  - Address aligned to $2^N$
- Find the smallest $2^N$ block fitting the required size
  - "List" of free blocks lists with fixed sizes $2^N$
- If there are no small blocks, create them dividing larger blocks
  - Buddies
    - Find the buddy address by XORing my address with the block size
- Merge blocks back when both buddies are free
- Significant internal fragmentation

# Buddy memory allocation



# Computer memory hierarchy



# Cache

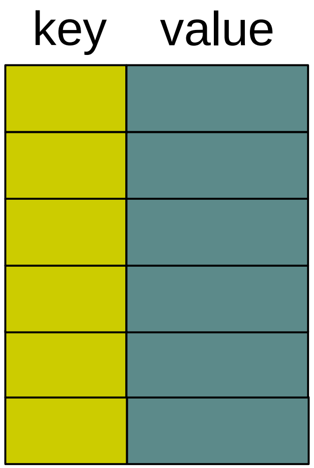
- HW or even SW
  - A structure holding data
  - Future requests for that data can be served faster
  - Generic cache operation
    - Make a request for data
    - Are data placed in the cache?
    - If they are, return them, otherwise do a slow calculation/access
- Cache in CPU
  - Hides memory latency
  - Based on locality of reference
  - CPU cache operation
    - Make a request for data in the memory
    - Are data placed in the cache? Look in all levels of cache in the CPU from the fastest L1 to the slowest LLC
    - If they are, return them to the execution unit in a CPU core, otherwise do a full memory access

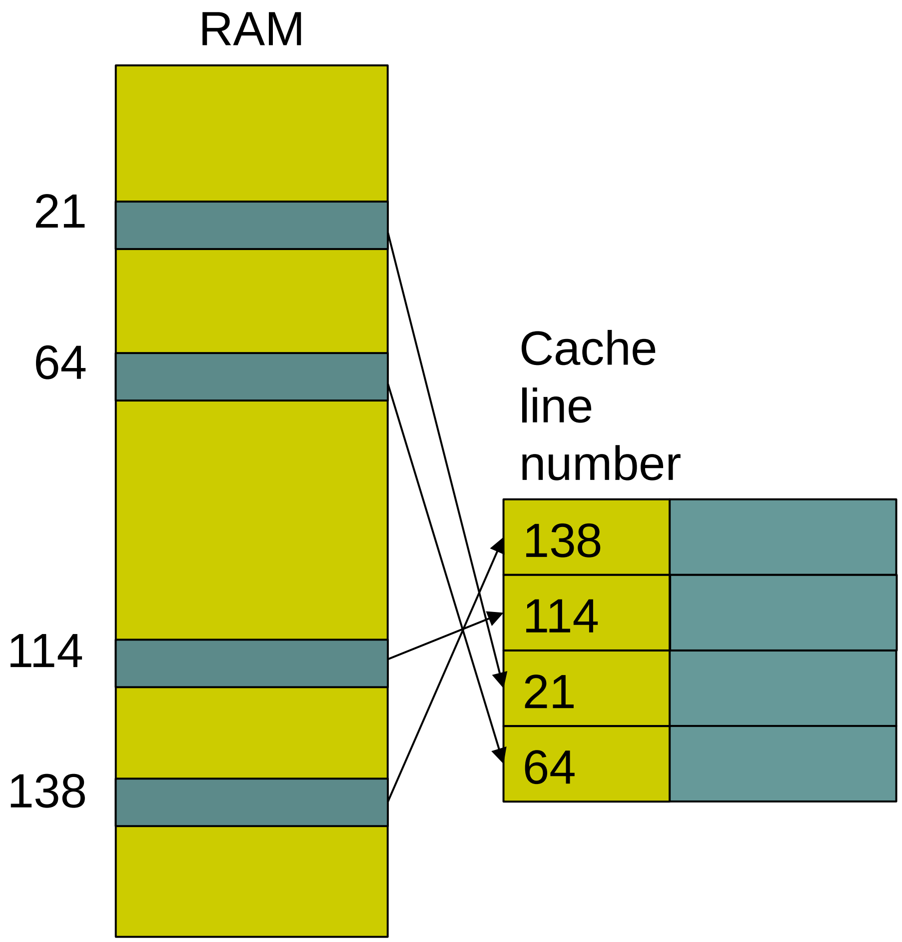
# Cache terminology

- Cache line/entry
  - Caches are organized in lines
    - Usual size is 64B
- Cache hit
  - Request served from the cache
  - Success rate around 97%
- Cache miss
  - Data not found in a cache hierarchy, do a full memory access
  - Load data from the memory to a cache line
    - Select either a free cache line or select a victim cache line
    - Store modified cache lines back to the memory
- Cache line state
  - MESI

# Associative memory

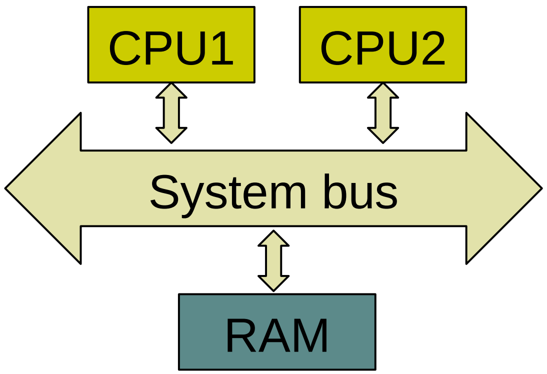
- Associative memory
  - Very fast
  - Content based addressing
  - Used in CPU caches

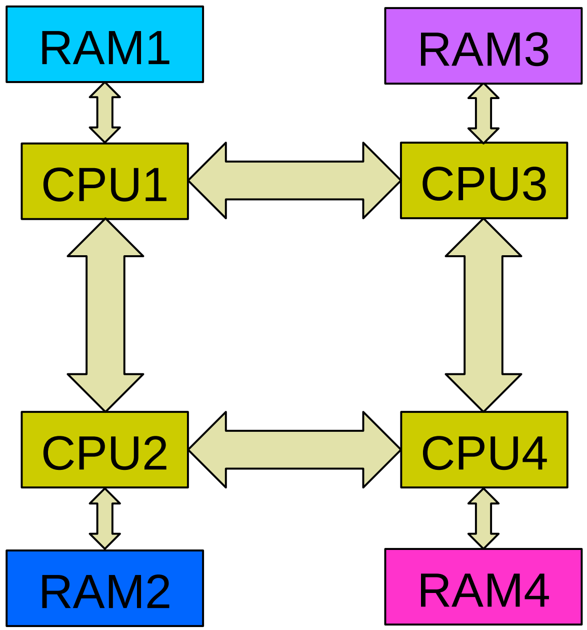


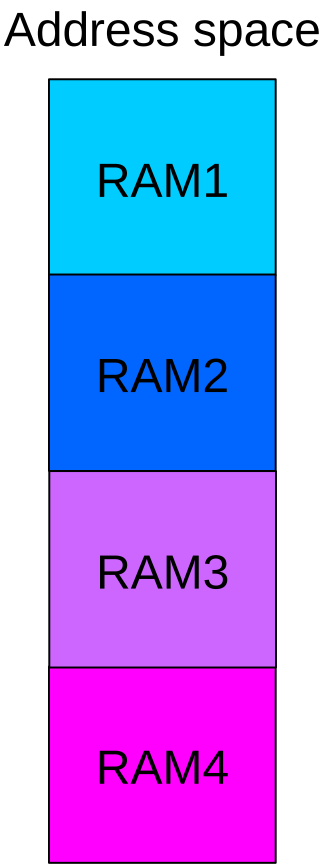



# NUMA

- Multiprocessors
  - SMP – Symmetric multiprocessing
  - NUMA – Non-uniform memory access



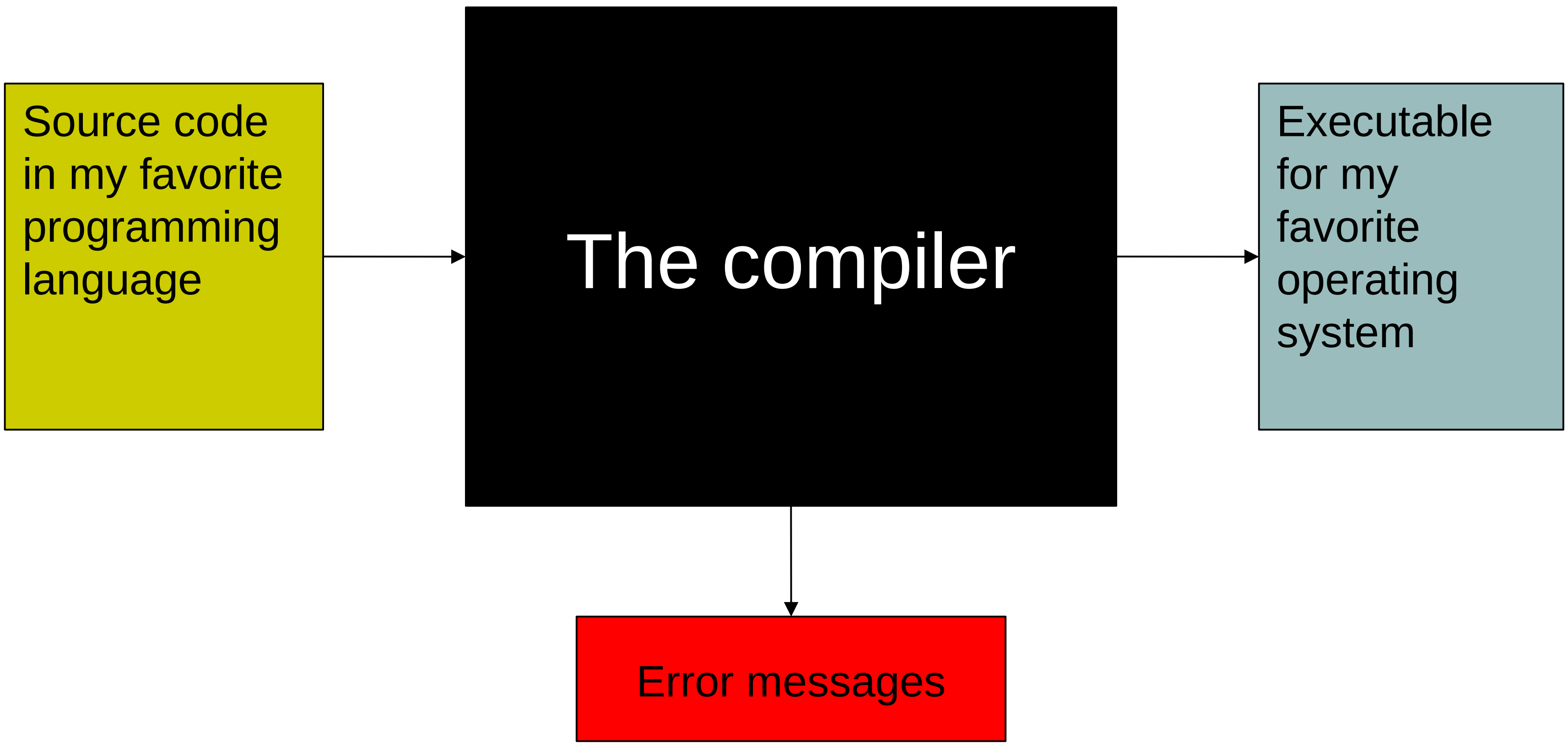



# Computer Systems

Programming languages

Jakub Yaghob

# Naïve view of a compiler



# Formal view of a compiler

- From slides of the course Compiler Principles
  - Let's have an input language $L_{in}$ generated by a grammar $G_{in}$
  - Let's have an output language $L_{out}$ generated by a grammar $G_{out}$ or accepted by an automaton $A_{out}$
  - The compiler is a mapping $L_{in} \rightarrow L_{out}$, where for all $w_{in}$ in $L_{in}$ exist $w_{out}$ in $L_{out}$. The mapping does not exist for $w_{in}$ not in $L_{in}$
- Don't worry!
  - You have to visit Automata and Grammars (NTIN071) course (obligatory) and then Compiler Principles (NSWI098) course (elective)

# Naïve view of a grammar

- Formal description of a language
  - Rules
  - Lexical elements

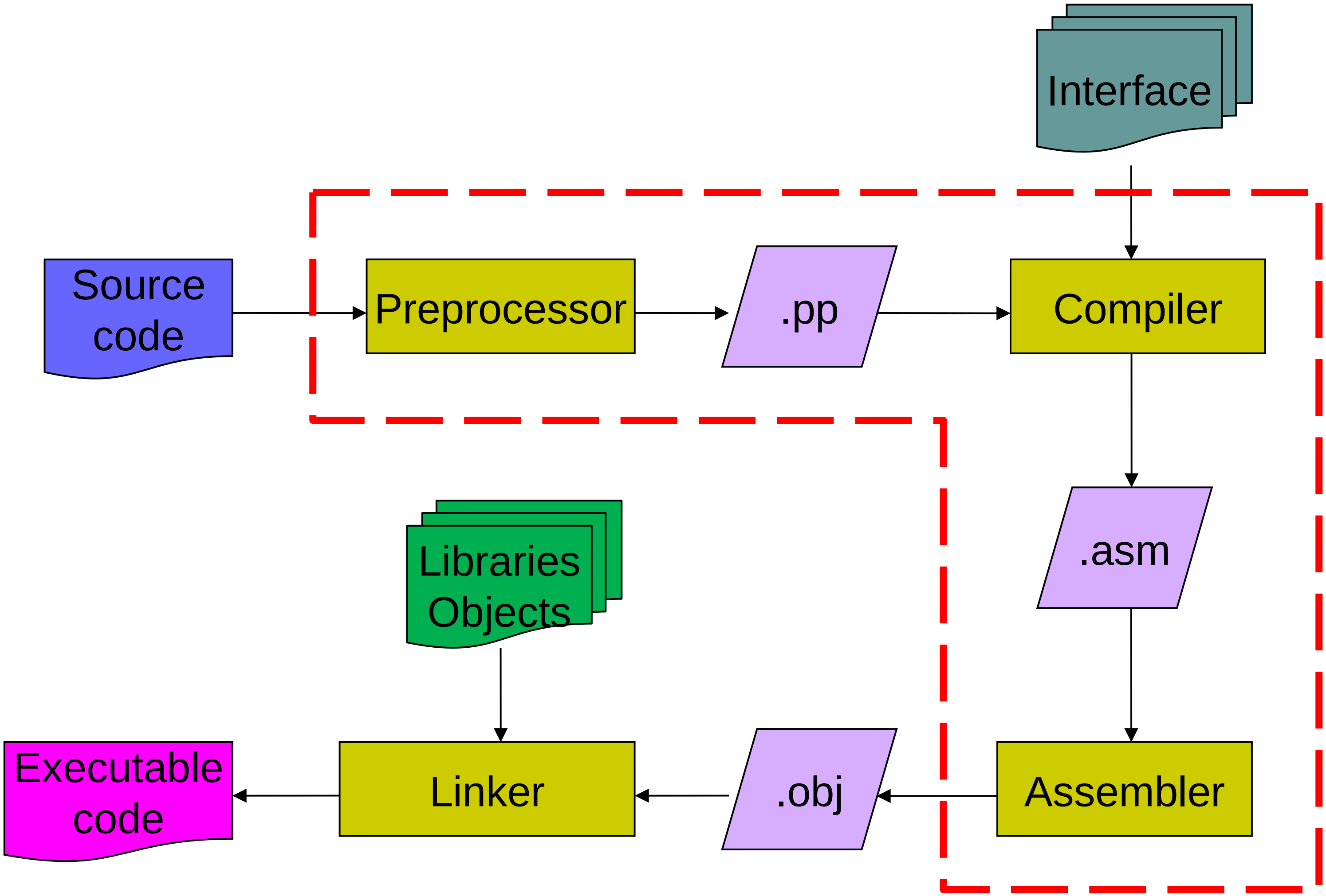
*iteration-statement*:

**while (** *expression* **)** *statement*

**do** *statement* **while (** *expression* **) ;**

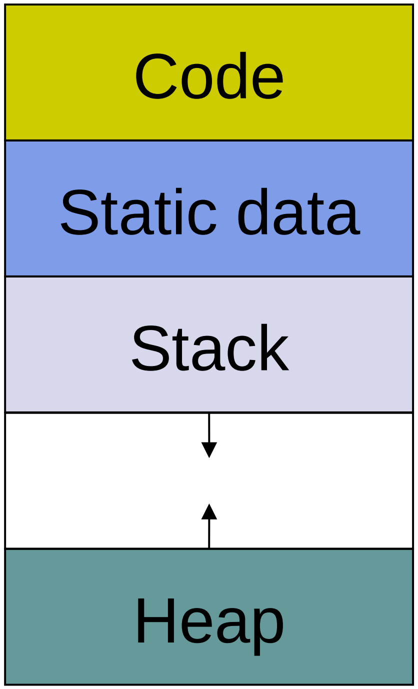
**for (** *expression*$_{opt}$ **;** *expression*$_{opt}$ **;** *expression*$_{opt}$ **)** *statement*

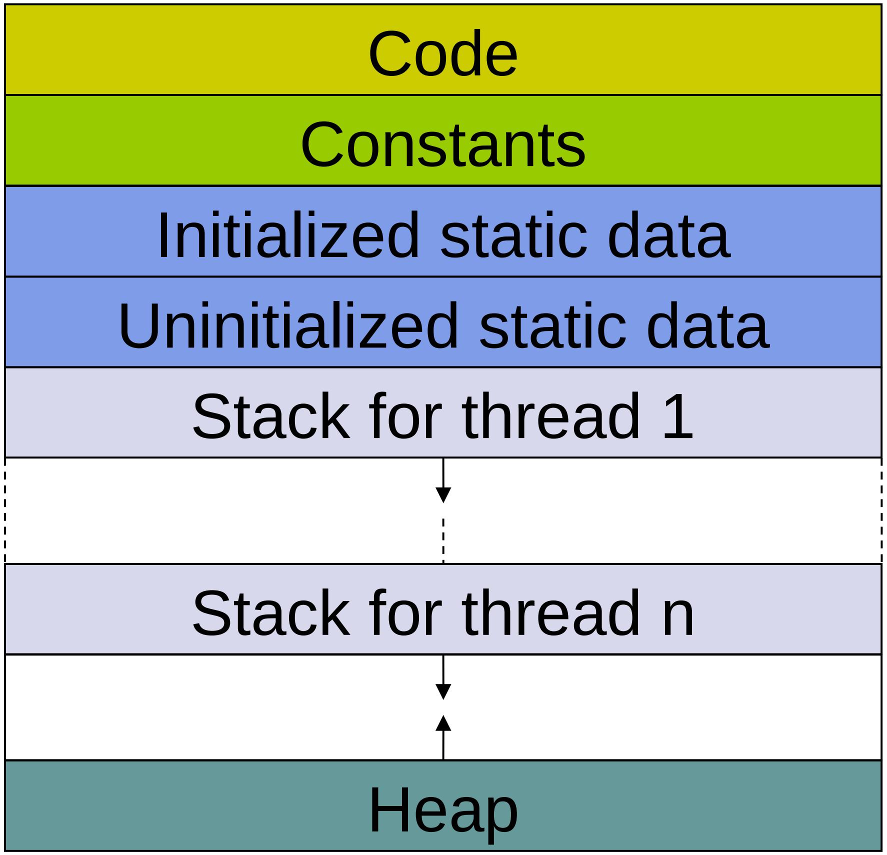
# More practical view of a translation



# Memory organization

- Memory organization during procedural program execution

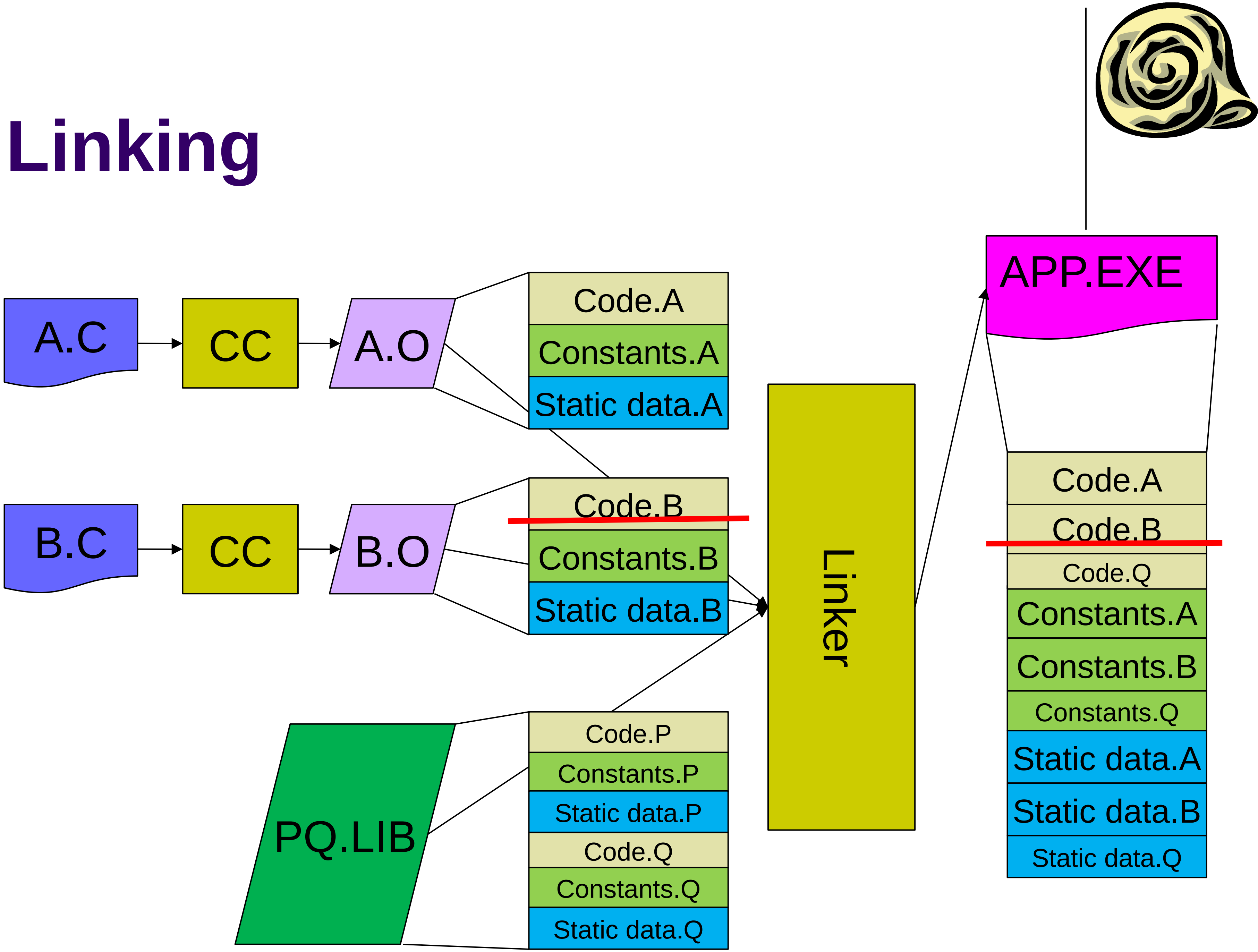


# Linker/librarian/loader

- Library
  - A collection of compiled source modules and other resources
  - Static, dynamic
- Linking
  - “Gluing” the results of the different translations and libraries together into one executable for given OS
  - Relocations
  - Positions independent code
- Loader
  - Part of OS, loads the executable into memory
  - Relocation again

# Linking



A.C
CC
A.O
Code.A
Constants.A
Static data.A
B.C
CC
B.O
Code.B
Constants.B
Static data.B
PQ.LIB
Code.P
Constants.P
Static data.P
Code.Q
Constants.Q
Static data.Q
Linker
APP.EXE
Code.A
Code.B
Code.Q
Constants.A
Constants.B
Constants.Q
Static data.A
Static data.B
Static data.Q

# Run-time

- Static language support
  - Compiler
  - Library interface
    - Header files
- Dynamic language support
  - Run-time program environment
    - Storage organization
    - Memory content before execution
    - Constructors and destructors of global objects
  - Libraries
  - Calling convention

# Function call – activation record (stack frame)

| |
|---|
| Return value |
| Actual parameters |
| Return address |
| Control link |
| Saved machine status |
| Local data |
| Temporaries |

- Saved machine status
  - Return address to the code
  - Registers
- Control link
  - Activation record of the caller

# Calling convention

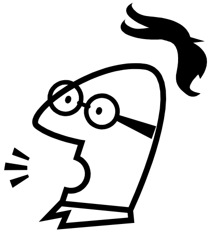
- Calling convention
  - Public name mangling
  - Call/return sequence for functions and procedures
    - Housekeeping responsibility
  - Parameter passing
    - Registers, stack
    - Order of passed parameters
  - Return value
    - Registers, stacks
  - Registers role
    - Parameter passing, scratch, preserved

# Public name mangling

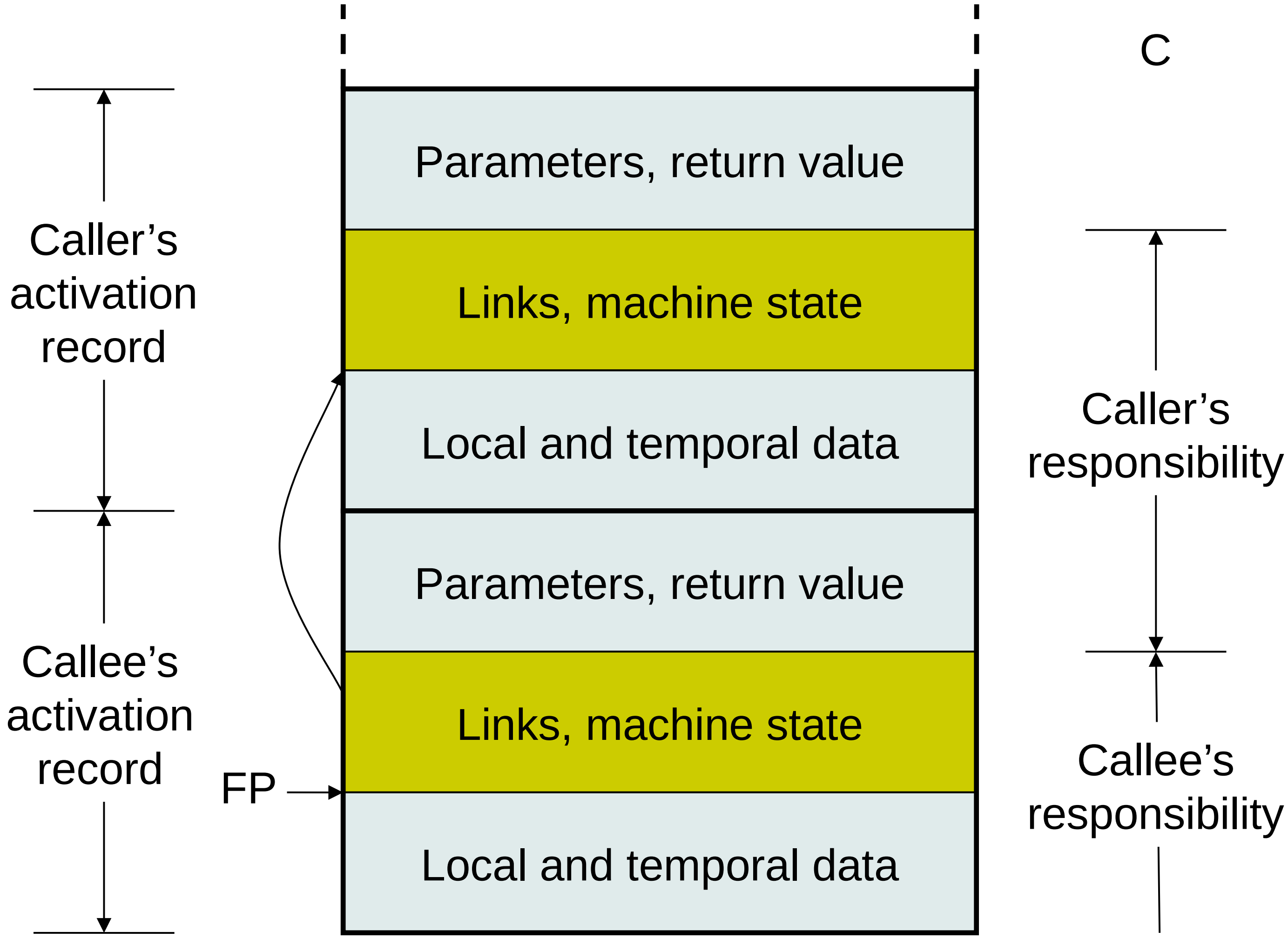
- Real meaning
  - mangle
    - mandlovat
    - rozsekat, roztrhat, rozbít, rozdrtit, těžce poškodit, potlouci, pohmožditi
    - *přen*. pokazit, znetvořit, k nepoznání změnit, překroutit, zkomolit
- Examples:

```
long f1(int i, const char *m, struct s *p)
```

| | |
|---|---|
| `_f1` | MSVC IA-32 C __cdecl |
| `@f1@12` | MSVC IA-32 C __fastcall |
| `_f1@12` | MSVC IA-32 C __stdcall |
| `?f1@@YAJHPBDPAUs@@@Z` | MSVC IA-32 C++ |
| `_f1` | GCC IA-32 C |
| `__Z2f1iPKcP1s` | GCC IA-32 C++ |
| `f1` | MSVC IA-64 C |
| `?f1@@YAJHPEBDPEAUs@@@Z` | MSVC IA-64 C++ |



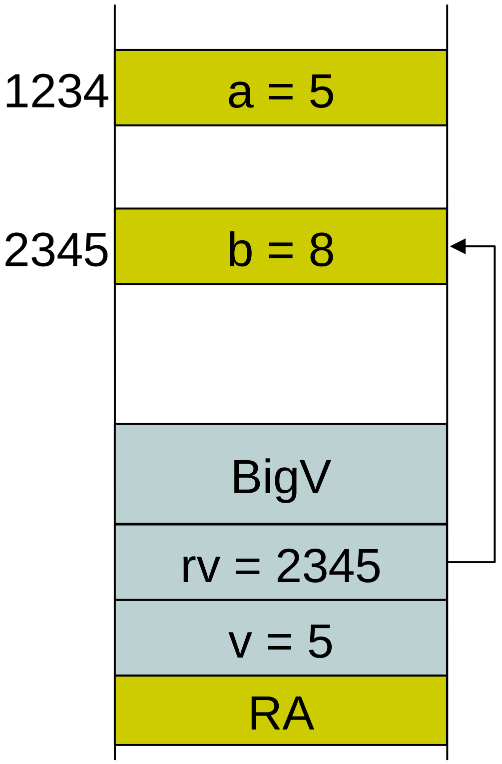
# Call/return sequence



# Parameter passing

- Call by value
  - Actual parameter is evaluated and the value is passed
  - Input parameters, the parameter is like a local variable
  - C
- Call by reference
  - The caller passes a pointer to the variable
  - Input/output parameters
  - & in C++

```
BigV fnc(int v, int &rv);
BigV r = fnc(a, b);
```



# Variables

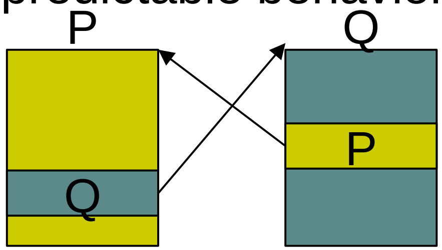
- Named memory holding a value
- Has a type
- Storage
  - Static data
    - Global variables in C
  - Stack
    - Local variables in C
  - Heap
    - Dynamic memory in C/C#
  - Dictionary
    - In Python
    - Not a storage, it is a dynamic structure

# Heap

- Storage for dynamic memory
- Allocate
  - Use all features from dynamic memory allocation
    - Free blocks evidence
    - Allocation algorithms
      - Extremely simple and fast incremental allocation
- Deallocate
  - Explicit action in some languages
    - C, C++
  - Automatic deallocation by garbage collection
    - Remove burden and errors
    - Works only with good knowledge of live objects and references

# Garbage collection

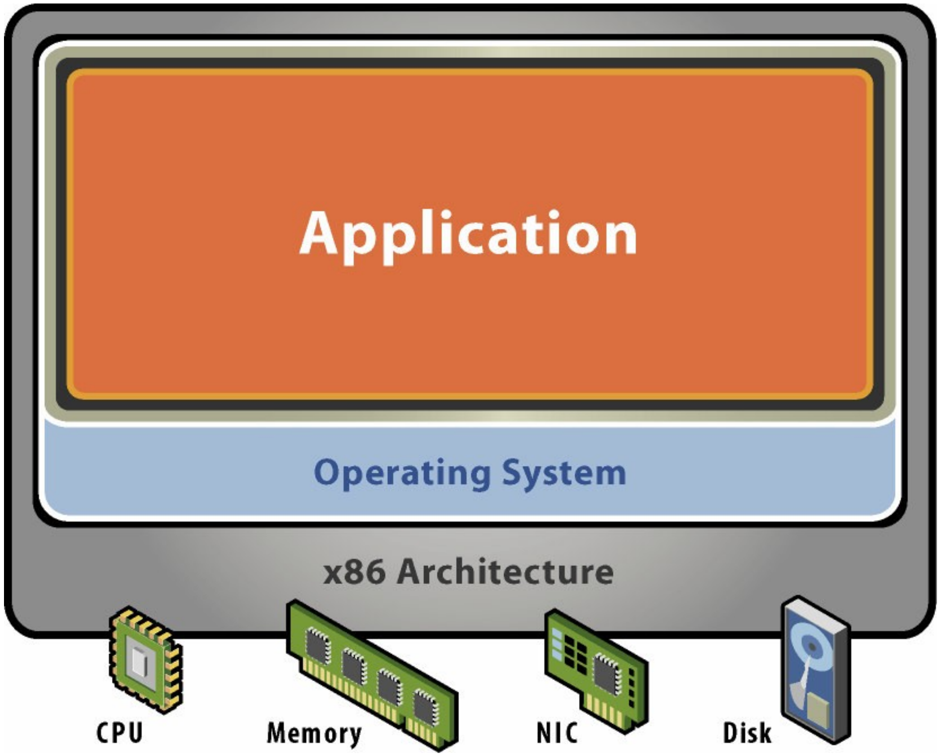
- Automatic removal of unused memory blocks
  - Advantages
    - No dangling pointers, no double free, no memory leaks, allows heap consolidation and fast allocation
  - Disadvantages
    - Performance impact, even execution stall, unpredictable behavior
- GC strategies
  - Tracing
    - Reachable objects from live objects
  - Reference counting
    - Problems with cycles, space and speed overhead
  - Advanced versions for languages with heavy use



# Virtual machine and containers

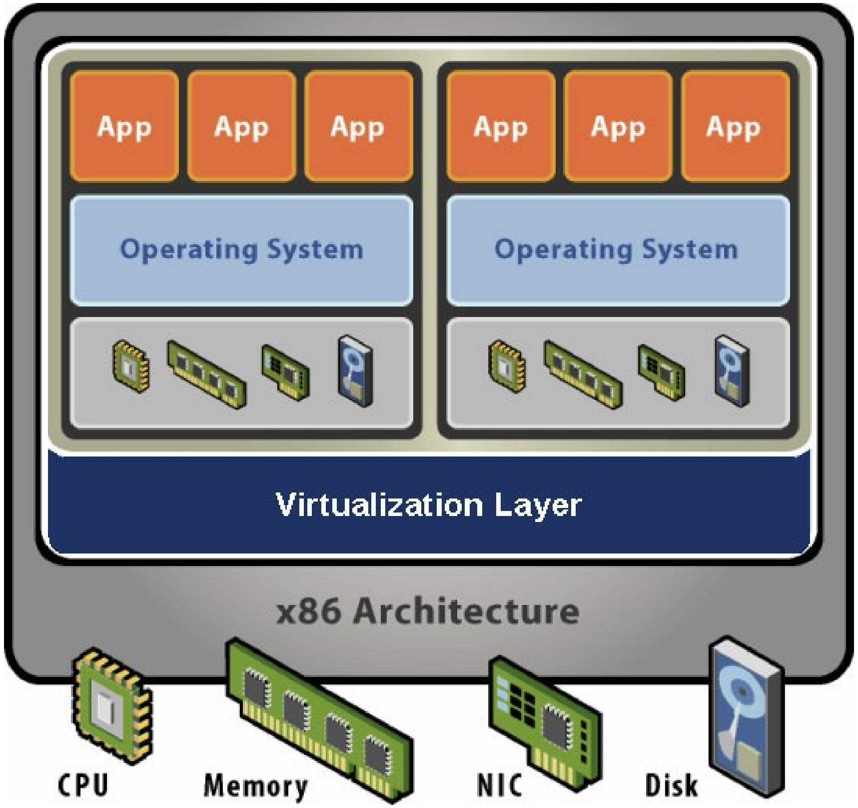
- VM = Emulation of a computer system
  - Full virtualization
    - Substitute for a real machine, allows execution of entire OS
    - Hypervisor shares real HW, native execution, virtual HW
    - Isolation, encapsulation, compatibility
  - Process VM
    - Runs as an application inside OS
    - Provides platform-independent programming environment
      - Abstract machine (instructions, memory, registers, …)
      - Java VM, .NET CLR
    - Slow execution
      - JIT, AOT
- Container = OS-level virtualization
  - OS kernel allows existence of multiple isolated user space instances

# Physical machine




- Physical HW
  - CPU, RAM, disks, I/O
  - Underutilized HW
- SW
  - Single active OS
  - OS controls HW

# Virtual machine




- HW-level abstraction
  - Virtual HW: CPU, RAM, disks, I/O
- Virtualization SW
  - Decouples HW and OS
  - Multiplexes physical HW across multiple guest VMs
  - Strong isolation between VMs
  - Manages physical resources, improves utilization

# Portability

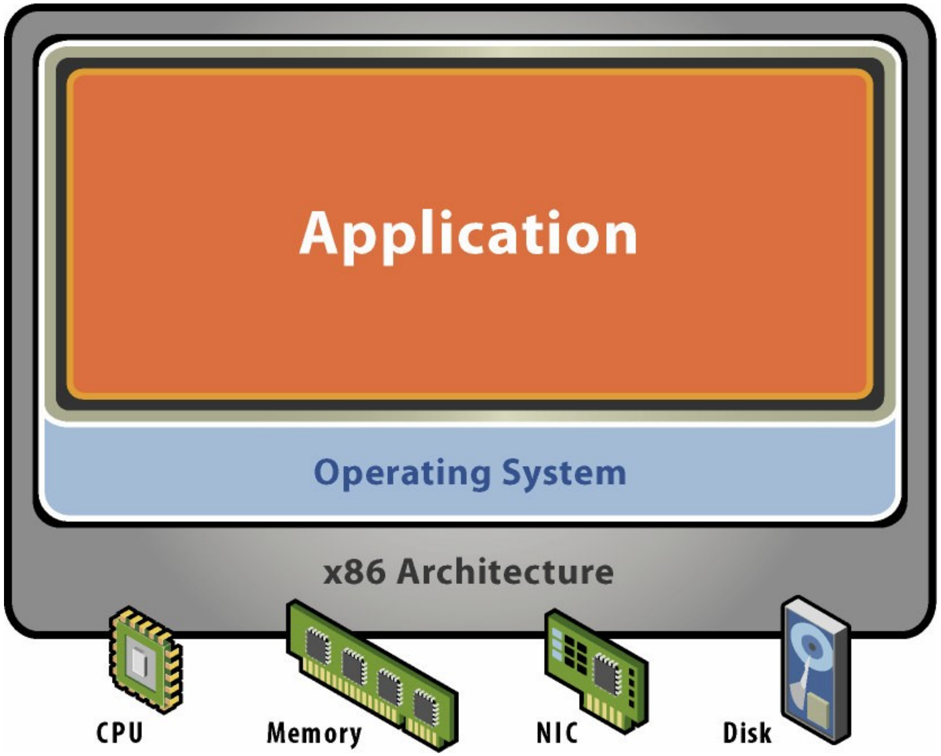
- Source code portability
  - CPU architecture
    - Different type sizes
      - C, C++
    - Fixed type sizes
      - C#, Java
  - Compiler
    - Different language “flavors”
      - C++ - gcc, msvc, clang, …
    - Use only syntax and library from a language standard
  - OS
    - Different system/library calls
      - Linux, Windows
    - Sometimes easy
      - BSD sockets

# Computer Systems

Operating systems

Jakub Yaghob

# Operating system – role



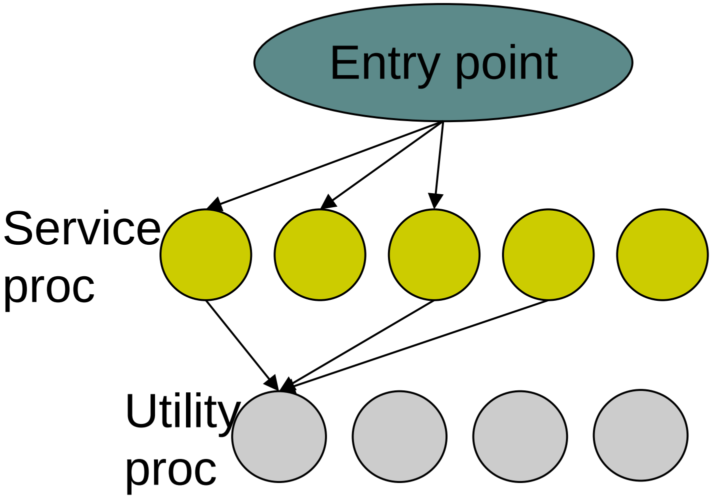

- Abstract machine
  - Presented by kernel API
    - System calls
  - Hide HW complexity
- Resource manager
  - All HW managed by OS
  - Sharing HW among applications

# CPU modes

- User mode
  - Available to all application
  - Limited or no access to some resources
    - Registers, instructions
- Kernel (system) mode
  - More privileged
  - Used by OS or by only part of OS
  - Full access to all resources

# Architecture – monolithic

- Monolithic systems
  - Big mess – no structure
  - “Early days”
    - Linux
  - Collection of procedures
    - Each one can call another one
  - No information hiding
  - Efficient use of resources, efficient code
  - Originally no extensibility
    - Now able to load modules dynamically

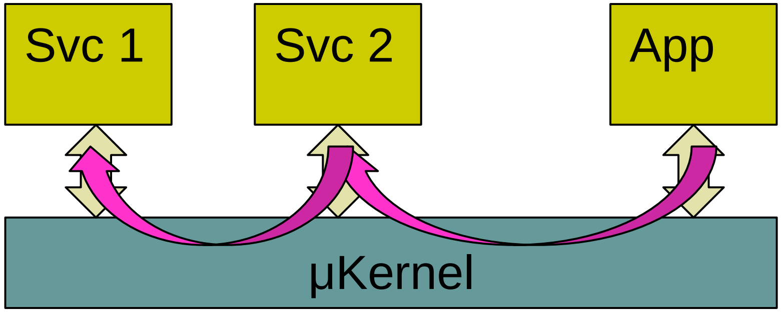


# Architecture – layered

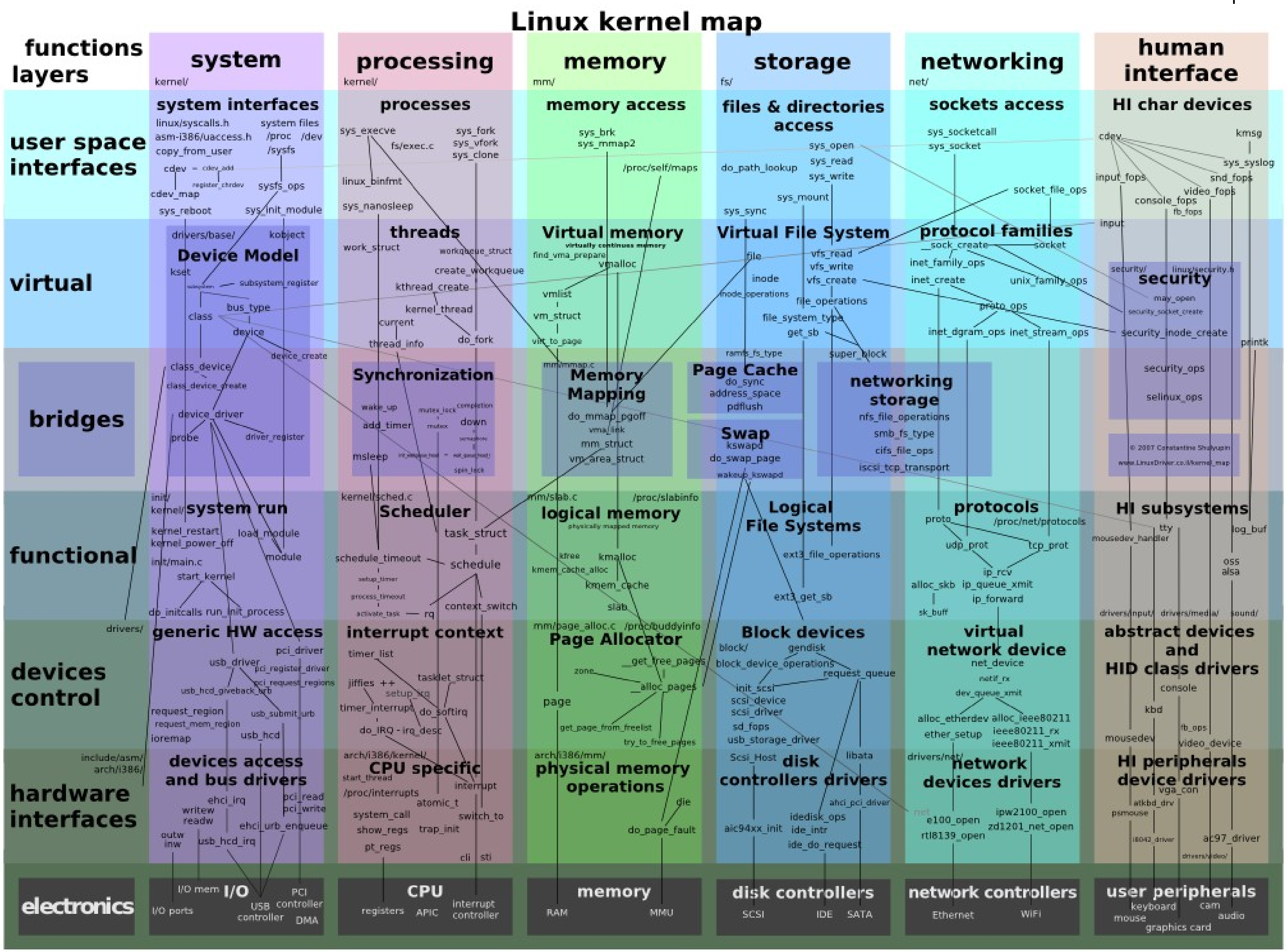
- Evolution of monolithic system
  - Organized into hierarchy of layers
  - Layer n+1 uses exclusively services supported by layer n
  - Easier to extend and evolve

# Architecture – microkernel

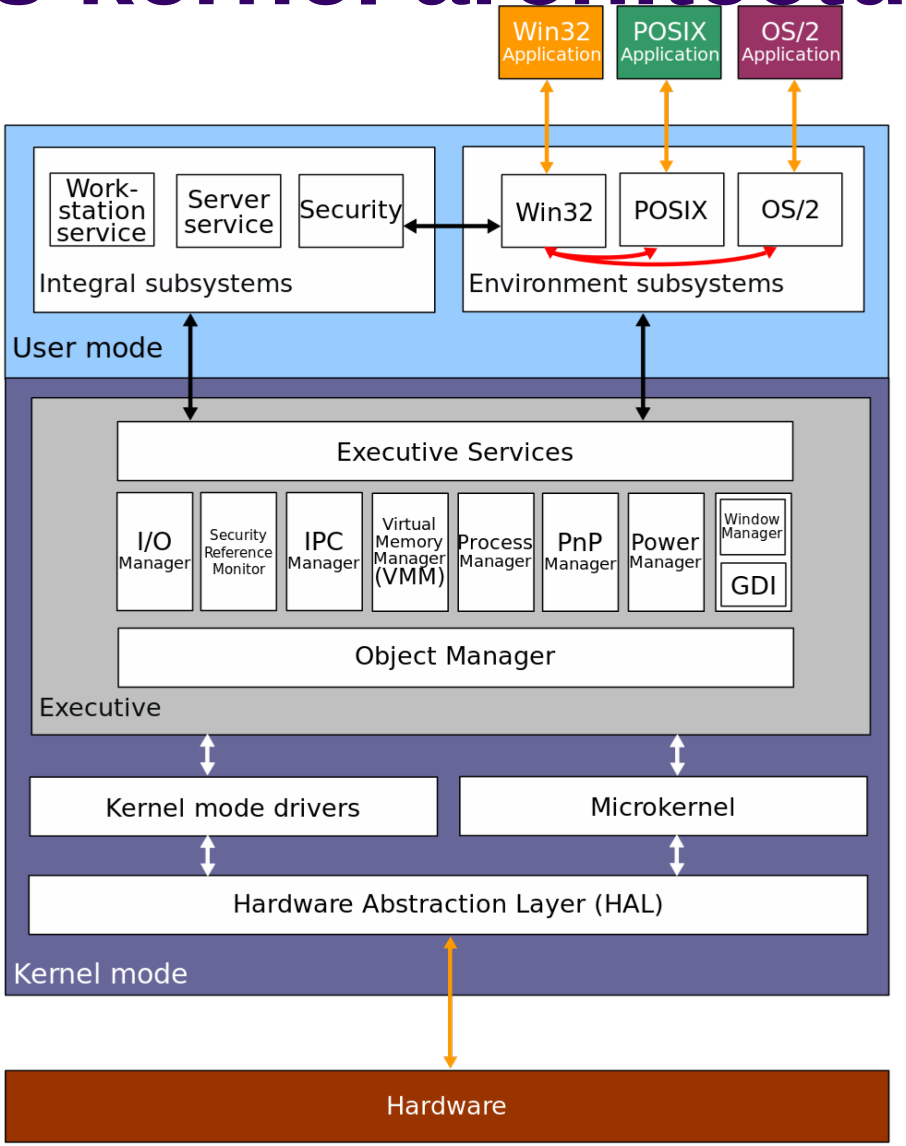
- Microkernel architecture
  - Move as much as possible from the kernel space to the user space
  - Communication between user modules
    - Message passing
    - Client/server
  - Extendable
  - Secure
  - Reliable



# Linux kernel architecture



# Windows kernel architecture

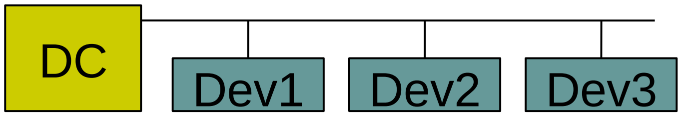


# Devices

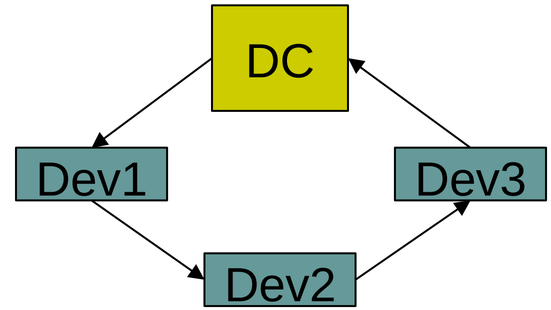
- Terminology
  - Device
    - “a thing made for a particular purpose”
  - Device controller
    - Handles “electrically” connected devices
      - Signals on a “wire”, A/D converters
    - Devices connected in a topology
  - Device driver
    - SW component, part of OS
    - Abstract interface to the upper layer in OS
    - Specific for a controller or a class/group of controllers

# Devices topology

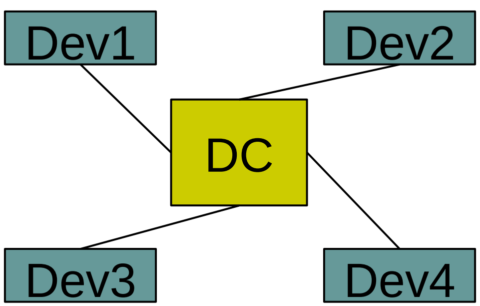
Bus



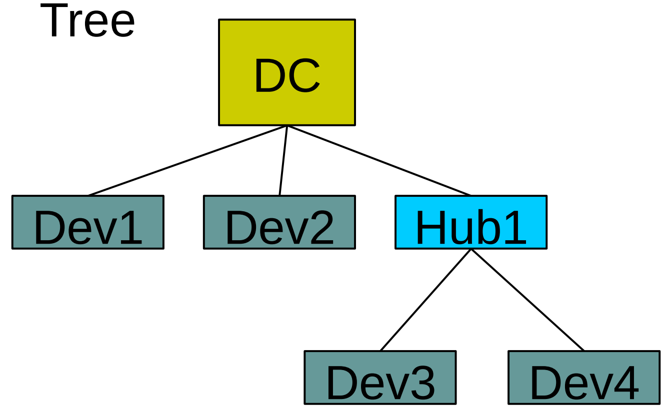

Ring




Star

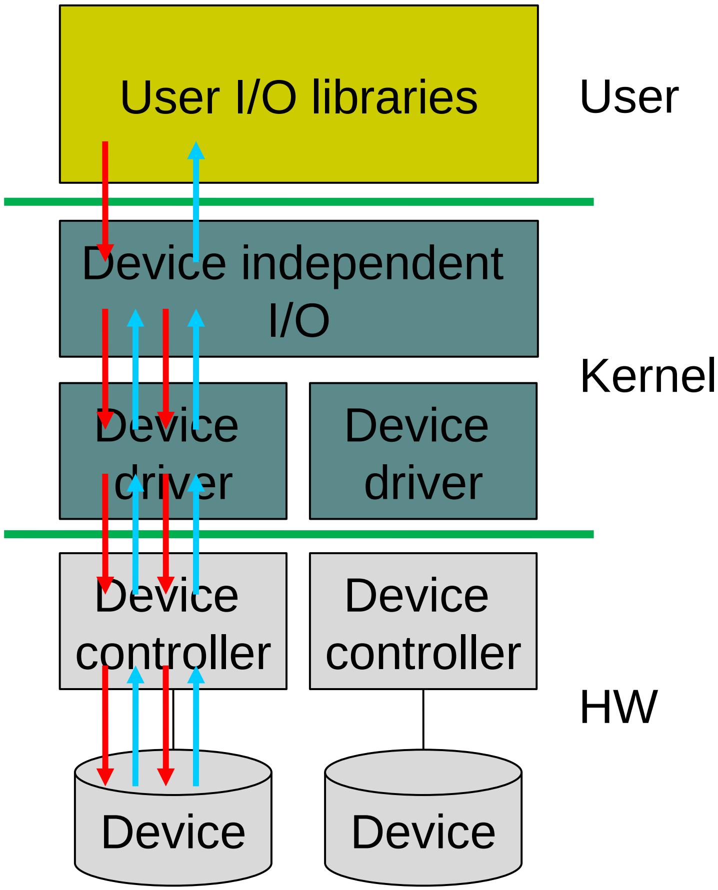



Tree



# Device handling

1. Application issues an I/O request
2. Language library makes a system call
3. Kernel decides, which device is involved
4. Kernel starts an I/O operation using device driver
5. Device driver initiates an I/O operation on a device controller
6. Device does something
7. Device driver checks for a status of the device controller
8. When data are ready, transfer data from device to the memory
9. Return to any kernel layer and make other I/O operation fulfilling the user request
10. Return to the application



# Device intercommunication

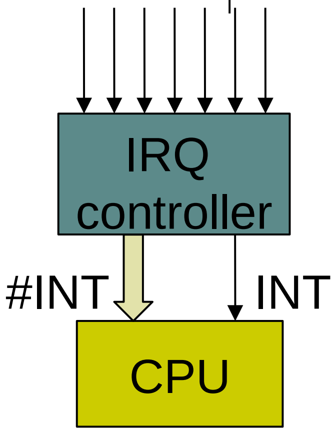
- Polling
  - CPU actively checks device status change
- Interrupt
  - Device notifies CPU that it needs attention
  - CPU interrupts current execution flow
  - IRQ handling
  - CPU has at least one pin for requesting interrupt
- DMA (Direct Memory Access)
  - Transfer data to/from a device without CPU attention
  - DMA controller
  - Scatter/gather

# Interrupt types

- External
  - HW source using an IRQ pin
  - Masking
- Exception
  - Unexpectedly triggered by an instruction
  - Trap or fault
  - Predefined set by CPU architecture
- Software
  - Special instruction
  - Can be used for system call mechanism

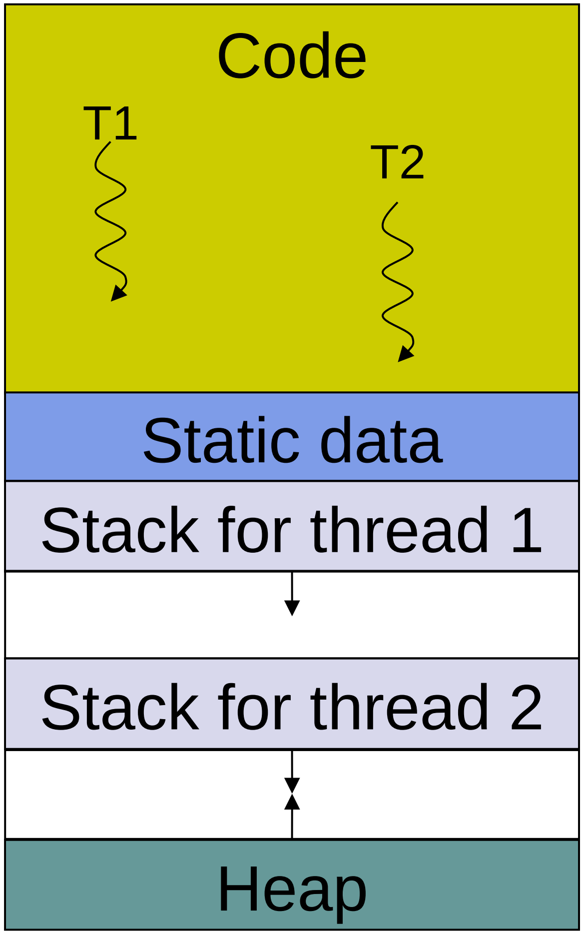
# Interrupt request handling

- What happens, when an interrupt occurs?
  - CPU decides the source of the interrupt
    - Predefined
    - IRQ controller
  - CPU gets an address of interrupt handler
    - Fixed
    - Interrupt table
  - Current stream of instructions is interrupted, CPU begins execution of interrupt handler's instructions
    - Usually between instructions
    - Privilege switch usually happens, interrupt handler is part of a kernel
  - Interrupt handler saves the CPU state
  - Interrupt handler do something useful
  - Interrupt handler restores the CPU state
  - CPU continues with original instruction stream



# Processing

- Program
  - A passive set of instruction and data
  - Created by a compiler/linker
- Process
  - An instance of a program created by OS
  - It contains program code and data
    - Process address space
  - The program is “enlivened” by an activity
    - Instructions are executed by CPU
  - Owns other resources
- Thread
  - One activity in a process
  - Stream of instructions executed by CPU
  - Unit of kernel scheduling
- Fiber
  - Lighter unit of scheduling
  - Cooperative scheduling
    - Running fiber explicitly yields



# Processing

- Scheduler
  - Part of OS
  - Uses scheduling algorithms to assign computing resources to scheduling units
- Multitasking
  - Concurrent executions of multiple processes
- Multiprocessing
  - Multiple CPUs in one system
  - More challenging for the scheduler
- Context
  - CPU (and possibly other) state of a scheduling unit
    - Registers (including PC)
- Context switch
  - Process of storing the context of a scheduling unit, which is now paused, and restoring the context of another scheduling unit, which resumes its execution

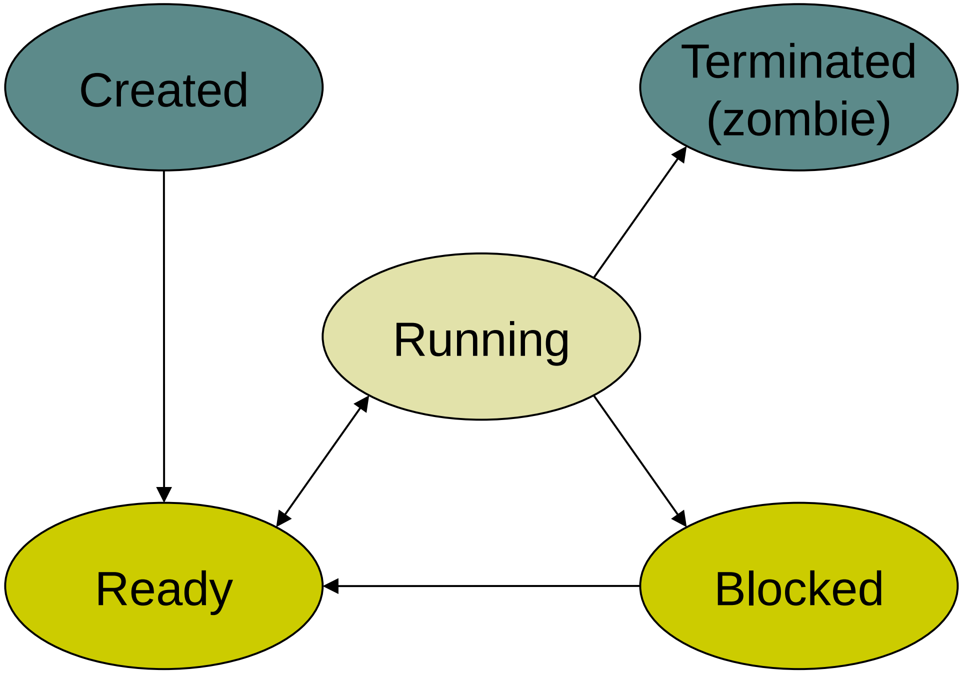
# Real-time scheduling

- Real-time scheduling
  - RT process has a start time (release time) and a stop time (deadline)
  - Release time – time at which the process must start after some event occurred
  - Deadline – time by which the task must complete
    - Hard – no value to continue computation after the deadline
    - Soft – the value of late result diminishes with time after the deadline



# Unit of scheduling state

- Created
  - Awaits admission
- Terminated
  - Until parent process waits for result
- Ready
  - Wait for scheduling
- Running
  - CPU assigned
- Blocked
  - Wait for resources



# Multitasking

- Cooperative
  - OS does not initiate context switch
  - Unit of scheduling must explicitly and voluntarily yield control
  - All processes must cooperate
  - Scheduling in OS reduced on starting the process and making context switch after the yield
- Preemptive
  - Each running unit of scheduling has assigned a time-slice
  - OS needs some external source of interrupt
    - Timer
  - If the unit of scheduling blocks or is terminated before the time-slice ends, nothing interesting will happen
  - If the unit of scheduling consumes the whole time-slice, it will be interrupted by the external source, OS will make context switch, and the unit of scheduling is moved to the READY state

# Scheduling

- Objectives
  - Maximize CPU utilization
  - Fair allocation of CPU
  - Maximize throughput
    - Number of processes that complete their execution per time unit
  - Minimize turnaround time
    - Time taken by a process to finish
  - Minimize waiting time
    - Time a process waits in READY state
  - Minimize response time
    - Time to response for interactive applications

# Scheduling – priority

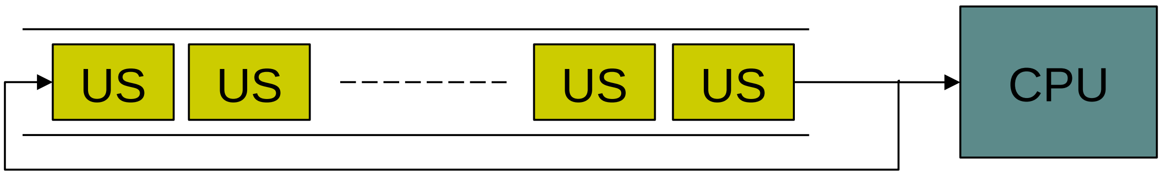
- Priority
  - A number expressing the importance of the process
  - Unit of scheduling with greater priority should be scheduled before unit of scheduling with lower priority
  - Static priority
    - Assigned at the start of the process
      - Users hierarchy or importance
  - Dynamic priority
    - Adding fairness to the scheduling
    - The priority of the process is the sum of a static priority and dynamic priority
    - Once in a time the dynamic priority is increased for all READY units of scheduling
    - The dynamic priority is initialized to 0 and is reset to 0 after the unit of scheduling is scheduled for execution

# Scheduling algorithms – non-preemptive

- First Come, First Serve (FCFS)
  - Simple queue, process enters the queue on the tail, the head process has CPU assigned and runs, then is removed from the queue
- Shortest Job First
  - Maximizes throughput
  - Expected job execution time must be known in advance
- Longest Job first

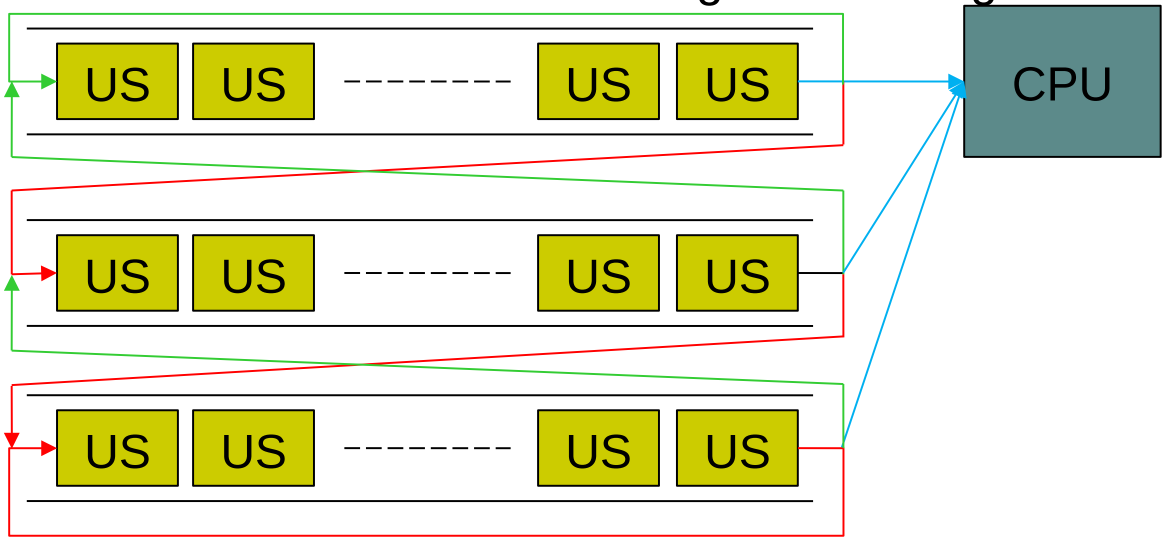
# Scheduling algorithms – preemptive

- Round Robin
  - Like FCFS, there is a queue
  - Each unit of scheduling has assigned time-slice
  - If the unit of scheduling consumes whole time-slice or is blocked, it will be assigned to the tail of the queue



# Scheduling algorithms – preemptive

- Multilevel feedback-queue
  - Multiple queues
    - Each level has assigned greater time-slice
  - If the unit of scheduling consumes the whole time-slice, it will be assigned to the lower queue
  - If the unit of scheduling blocks before consuming the whole time-slice, it will be assigned to the higher queue
  - Schedule head unit of scheduling from the highest non-empty queue



# Scheduling algorithms - preemptive

- Completely fair scheduler (CFS)
  - Implemented in Linux kernel
  - Processes are in red-black tree
    - Indexed by execution time
  - Maximum execution time
    - Time-slice calculated for each process
    - The time waiting to run divided by the total number of processes
  - Scheduling algorithm
    - The leftmost node is selected (lowest execution time)
    - If the process completes its execution, it is removed from scheduling
    - If the process reaches its maximum execution time or is somehow stopped or interrupted, it is reinserted into the tree based on its new execution time

# File

- File
  - Collection of related information
    - Stored on secondary storage (?)
    - Abstract stream of data
  - Operations
    - Open, close, read, write, seek
  - Access
    - Sequential, random
  - Type
    - Extension
  - Attributes
    - Name, timestamps, size, access, …

# File directory

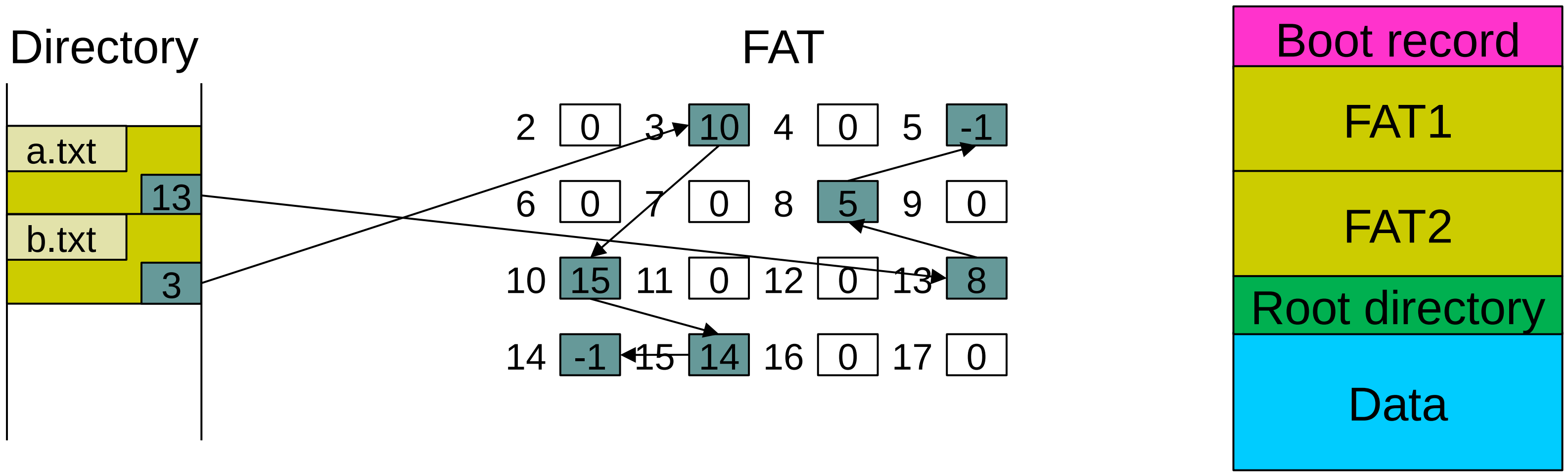
- Directory
  - Collection of files
    - Efficiency – a file can be located more quickly
    - Naming – better navigation for users
    - Grouping – logical grouping of files
  - Usually represented as a file of a special type
  - Store file attributes
  - Hierarchy or structure
    - Root
  - Operations
    - Create/delete/rename file/subdirectory
    - Search for a name
    - List members

# File system

- File system
  - Controls, how and where data are stored
  - Creates an abstraction for files and directories
  - Responsibility
    - Name translation
    - File data location
    - Free blocks management
      - Bitmap, linked list
  - Local file system
    - Stored on HDD, SSD, removable media
    - FAT, NTFS, ext234, XFS, …
  - Network file system
    - Access to files/directories over a network stack
    - NFS, CIFS/SMB, …

# FAT

- File Allocation Table (FAT)
  - Simple, old, MS-DOS, many variants used today
  - One structure (FAT) for managing free blocks and file data location
  - Directory
    - Sequence of entries with fixed size and attributes
      - Starting cluster, name+ext, size, timestamps, attributes
    - Root in fixed position



# ext2

- Second extended file system (ext2)
  - Simple, old, Linux
  - Inode (index node)
    - Represents one file/directory
  - Directory
    - Sequence of entries with fixed structure
      - Inode, name