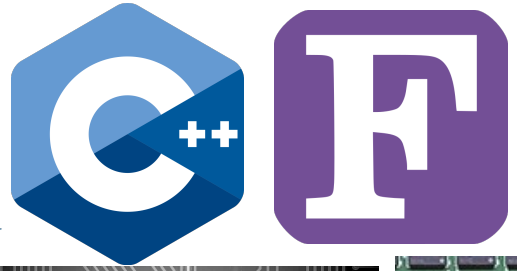
Programming Languages Primer (assembler onwards)

Koustav Pal

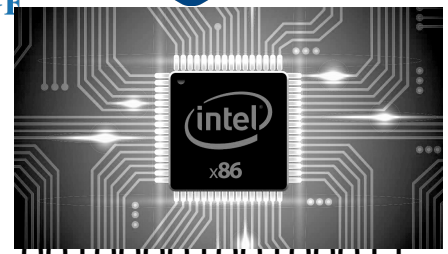
Language stacks

4th gen languages

3rd gen languages



Assembly language (human readable machine code)



↓

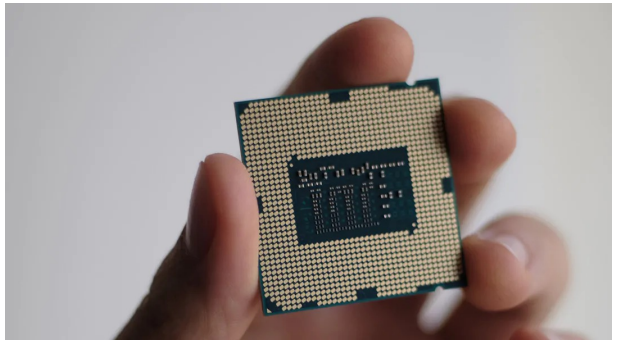
Machine code

↓

Hardware

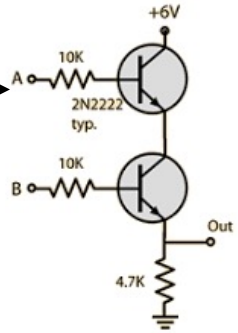
→

Chip = trillions of transistors

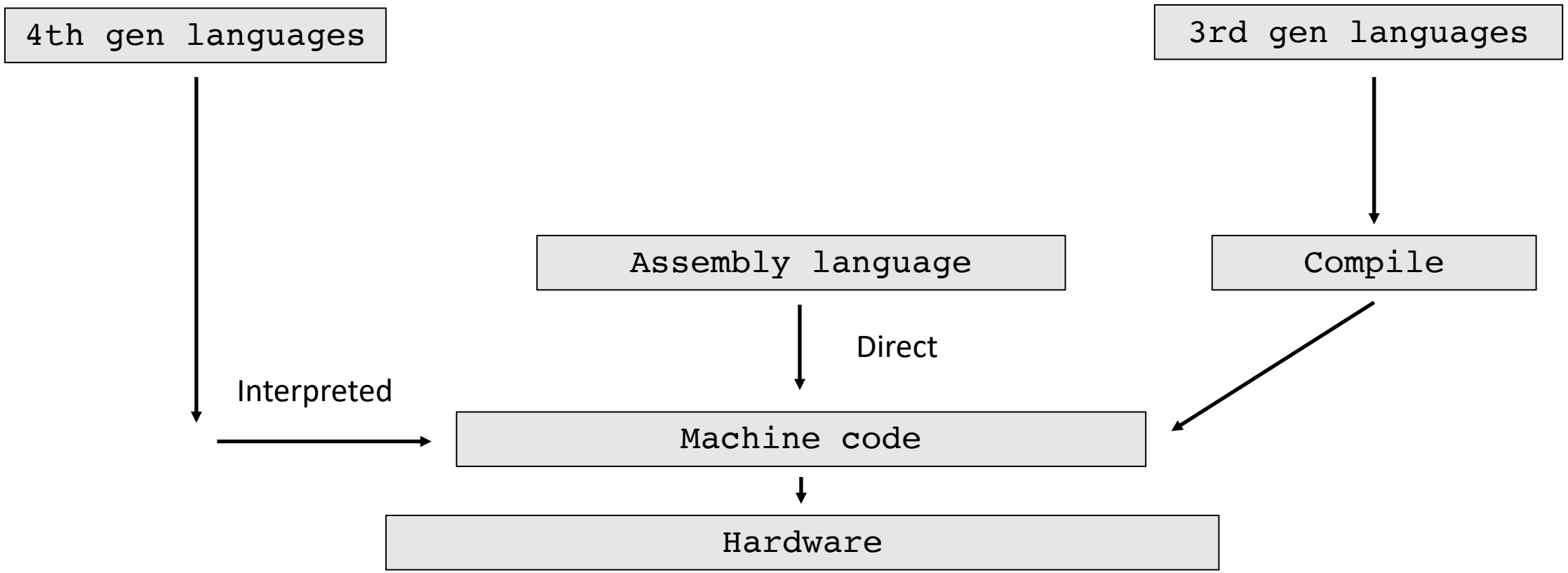


→

Logic gates for operations



Each language has a different way of communicating with the hardware



1st & 2nd gen languages

- **Architecture specific language design. Eg. x86, ARM, x64**
- Primarily created from a hardware perspective
- Designed to make hardware manipulation relatively simple for a user.
- Bit level data structures
- Instructions are always relayed and processed line-by-line

3rd gen languages

- Compiled language.
- Errors reported during compile
- Designed to abstract away machine level mumjo jumbo away from a user. E.g. No registry handling
- Users can access much more complex data structures (arrays, list).
- Procedural programming paradigm
- **Users are responsible for memory management**

4th gen languages

- Interpreted language. Just in time execution. An interpreter converts the program to machine code and executes.
- Errors reported during execution
- Complete abstraction. Users do not have to worry about memory management or computer architecture.
- Functional and OOPs paradigm
- **Comparatively much slower than previous generations**

Examples of deeper layers of abstraction?

- R's approach towards serialization and vector cycling
- Object oriented programming paradigm provide higher level abstraction.
  - GenomicRanges classes
  - Julia ODE solvers in DifferentialEquations.jl, ModelingToolkit.jl, native mathematical equations.

Assembly level languages directly manipulates machine code to turn on or turn off a set of transistors

```
;Copyright (c) 1999 Konstantin Boldyshev <konst@linuxassembly.org>
;
;"hello, world" in assembly language for Linux
;
;to build an executable:
;       nasm -f elf hello.asm
;       ld -s -o hello hello.o

section .text
; Export the entry point to the ELF linker or loader.  The conventional
; entry point is "_start". Use "ld -e foo" to override the default.
  global _start

section .data
msg db 'Hello, world!',0xa ;our dear string
len equ $ - msg      ;length of our dear string

section .text

; linker puts the entry point here:
_start:

; Write the string to stdout:

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

; Exit via the kernel:

  mov ebx,0   ;process' exit code
  mov eax,1   ;system call number (sys_exit)
  int 0x80    ;call kernel - this interrupt won't return
```

Registry addresses on Intel x86

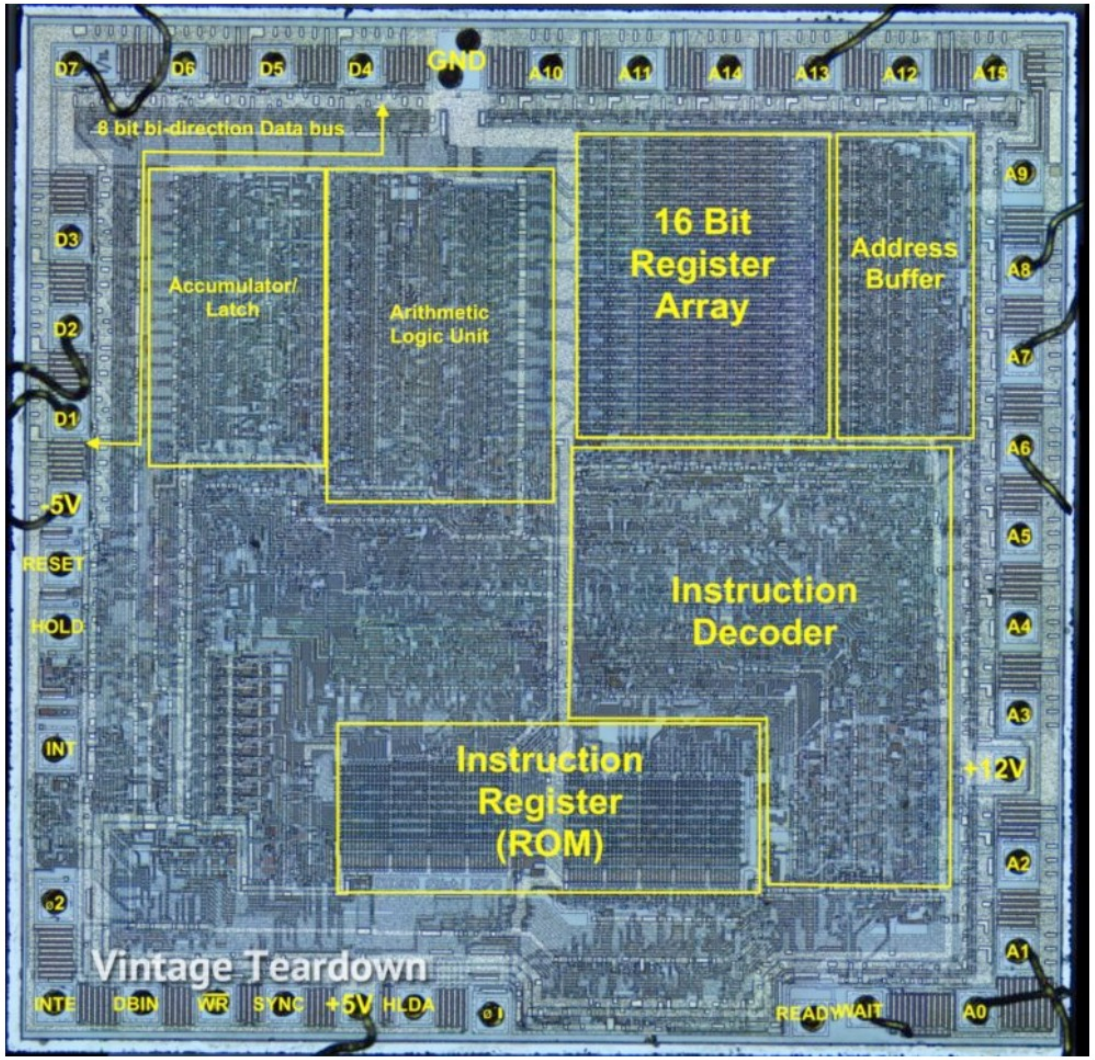
https://montcs.bloomu.edu/Information/LowLevel/Assembly/hello-asm.html#helloLinux

Machine code Hello World on Intel x86

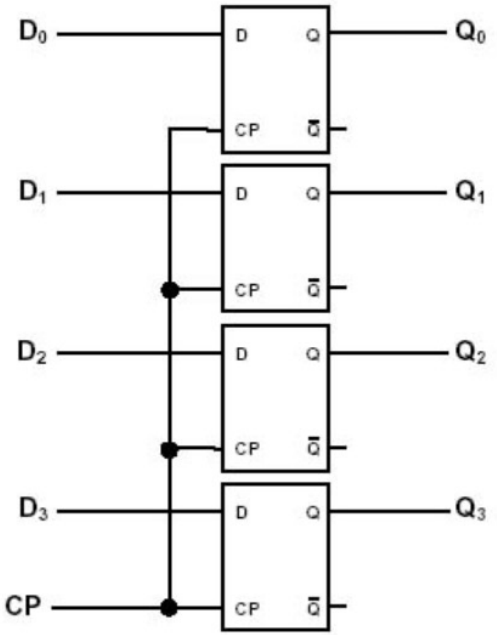
```
b8    21 0a 00 00   #moving "!\n" into eax
a3    0c 10 00 06   #moving eax into first memory location
b8    6f 72 6c 64   #moving "orld" into eax
a3    08 10 00 06   #moving eax into next memory location
b8    6f 2c 20 57   #moving "o, W" into eax
a3    04 10 00 06   #moving eax into next memory location
b8    48 65 6c 6c   #moving "Hell" into eax
a3    00 10 00 06   #moving eax into next memory location
b9    00 10 00 06   #moving pointer to start of memory location into ecx
ba    10 00 00 00   #moving string size into edx
bb    01 00 00 00   #moving "stdout" number to ebx
b8    04 00 00 00   #moving "print out" syscall number to eax
cd    80            #calling the linux kernel to execute our print to stdout
b8    01 00 00 00   #moving "sys_exit" call number to eax
cd    80            #executing it via linux sys_call
```

https://excelwithbusiness.com/blogs/news/say-hello-world-in-53-different-programming-languages
https://www.cs.uaf.edu/2016/fall/cs301/lecture/09_28_machinecode.html

What does a registry look like? Teardown of an Intel 8080A



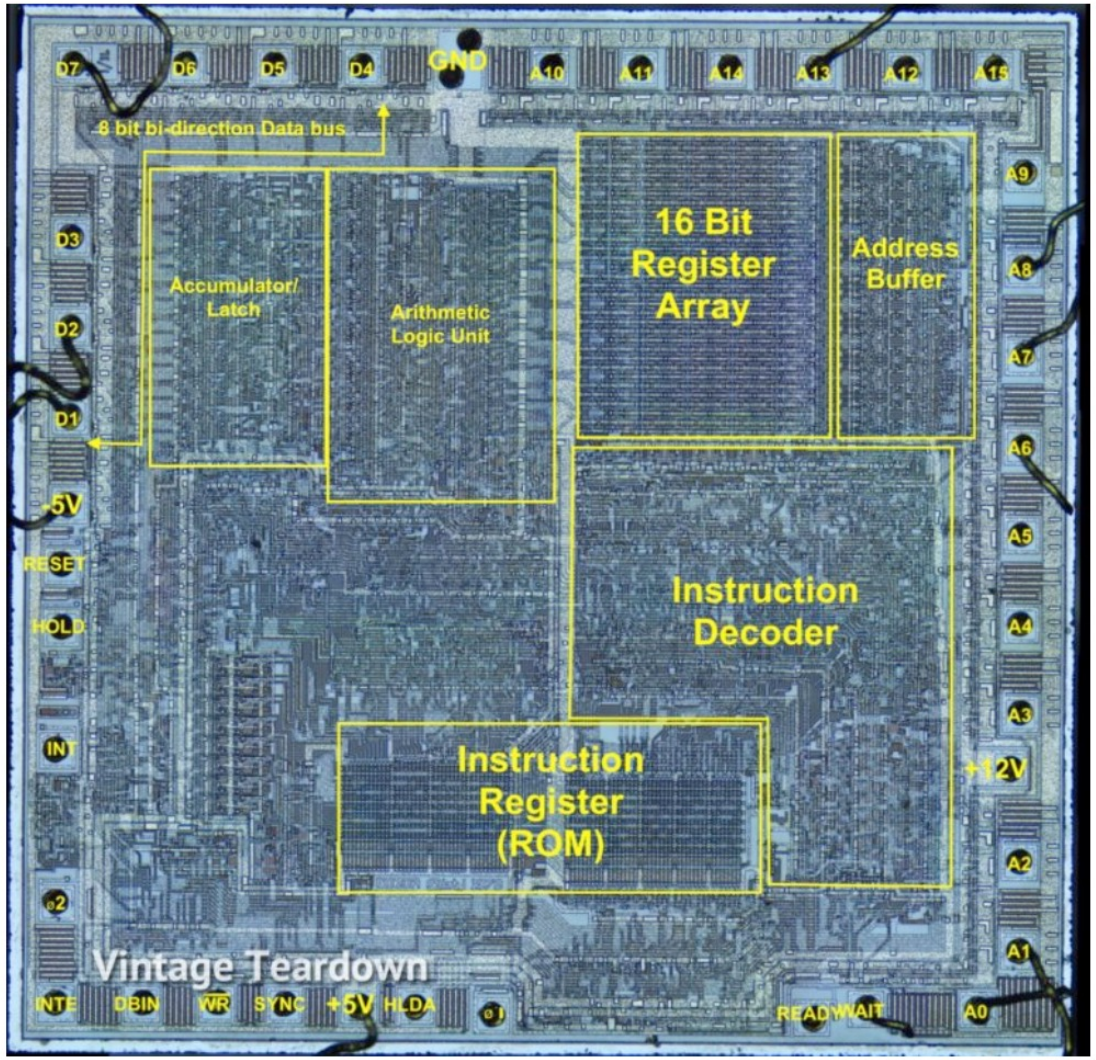
4-bit register constructed from logic gates



32-bit equals register created from 32 logic gates

https://electronics.stackexchange.com/questions/476334/where-are-registers-and-what-do-they-look-like

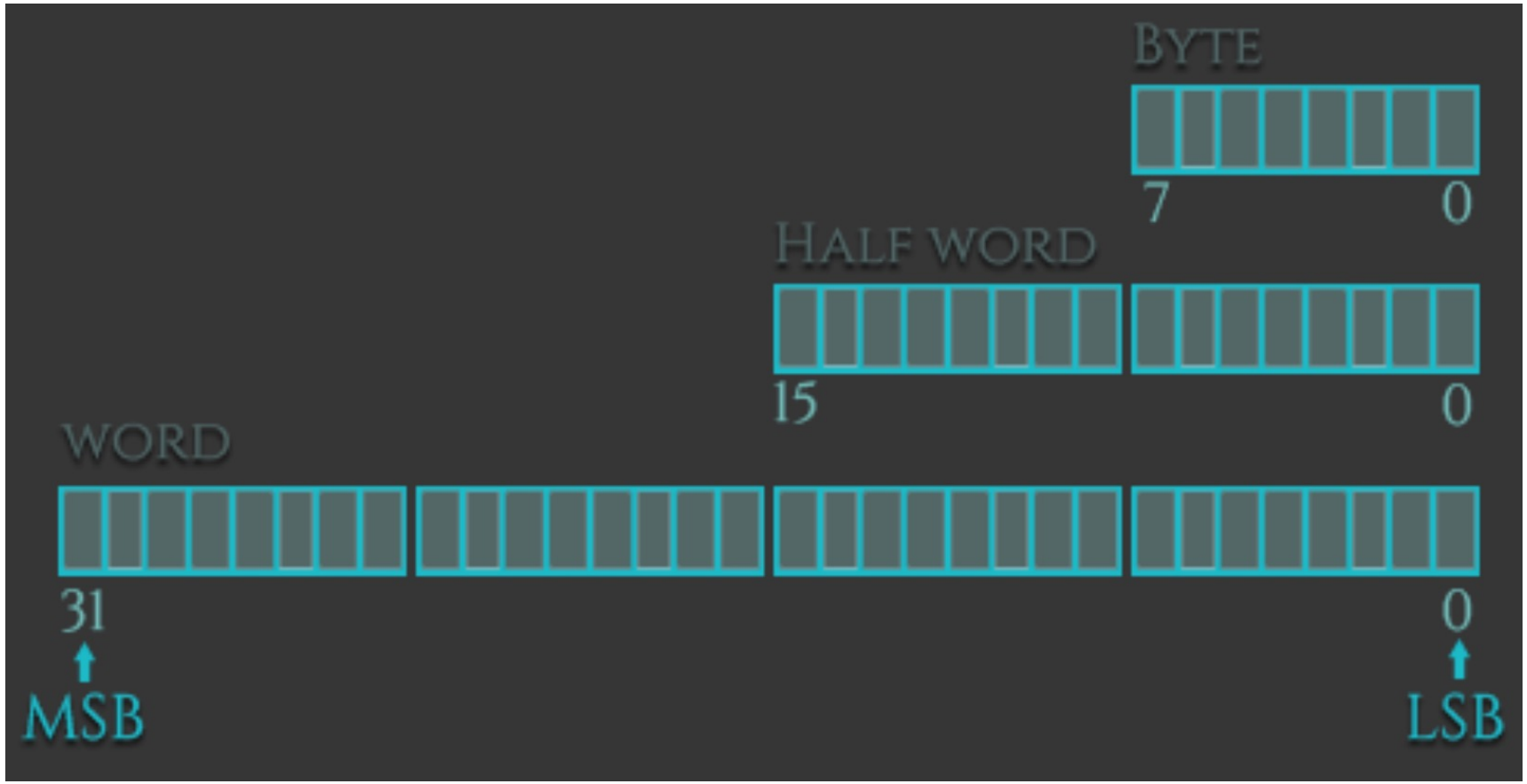
Similarly, arithmetic operations require the usage of arithmetic logic unit




```
SECTION .data

    extern printf
    global main

fmt:
    db "%d", 10, 0

SECTION .text

main:
    mov     eax, 14
    mov     ebx, 10
    add     eax, ebx

    push    eax
    push    fmt
    call    printf

    mov     eax, 1
    int     0x80
```

https://electronics.stackexchange.com/questions/476334/where-are-registers-and-what-do-they-look-like

Assembly level languages have very simple data types

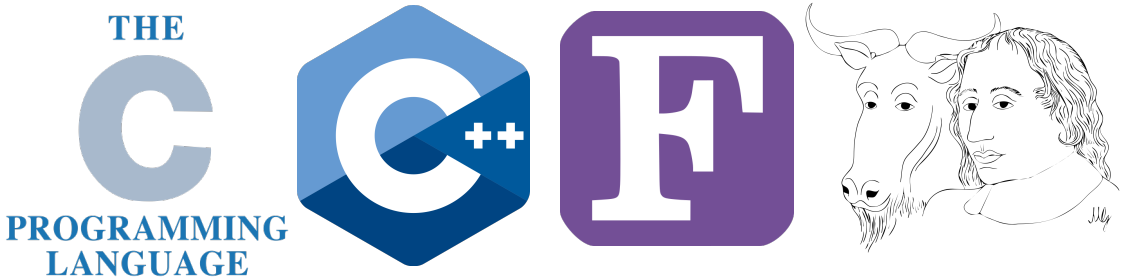


https://azeria-labs.com/arm-data-types-and-registers-part-2/

Assembly - Summary

- Architecture specific language
  - Even different versions within the same architecture will have variations in the underlying assembly code
- Machine level execution of user defined commands which must be defined from beginning to end
- First level of abstraction already visible as user does not write a program in machine code

Third generation languages

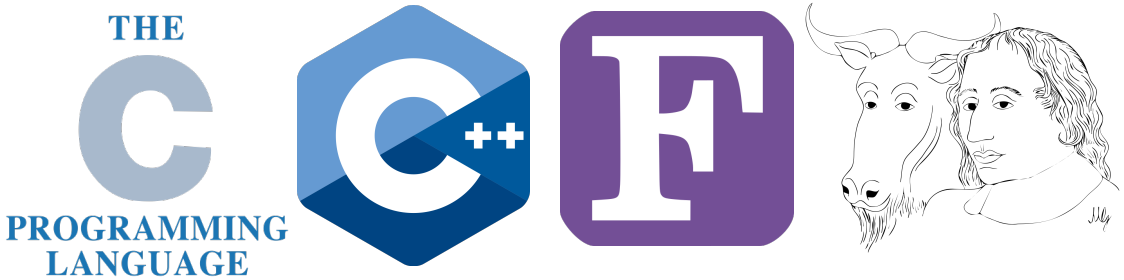


- C, C++, C#, Pascal, Fortran, Java
- Memory management of variables in C during declaration
  - Pre-defined memory allocation for a variable, stack
  - Undefined/max memory allocation for a variable, heap
- Any memory associated to stack variables is automatically freed up. However, stack memory is limited
- Memory allocation to heap variables must be manually freed up.
- C++ allocates some heap memory towards automatic garbage collection

Much more complex data types are available from here

- Arrays,
- Lists

Third generation languages



- C, C++, C#, Pascal, Fortran, Java
- Memory management of variables in C during declaration
  - Pre-defined memory allocation for a variable, stack
  - Undefined/max memory allocation for a variable, heap
- Any memory associated to stack variables is automatically freed up. However, stack memory is limited
- Memory allocation to heap variables must be manually freed up.
- C++ allocates some heap memory towards automatic garbage collection

Fourth generation languages

- R, Python, Julia
- Characterised by functional and object oriented programming paradigm
- Garbage collection is a built-in functionality
- R OOPs is built-in through S4 and R6 classes

High level language characteristics

- Encapsulation and immutability
- Abstractions
- Inheritance

Encapsulation

```
x <- c(1, 7)
tracemem(x)
```

```
## [1] "<0x55d1fa6f6bb8>"
```

```
x[2] <- 2
```

```
## tracemem[0x55d1fa6f6bb8 -> 0x55d1faa37298]: eval eval withVisible withCal
```

https://blog.r-hub.io/2020/01/22/mutable-api/

Why is encapsulation and immutability useful?

- Once initiated an object of class GenomicMatrix cannot be changed
- This object is used as a basis for working with complex data structures and contains definitions expected to remain constant

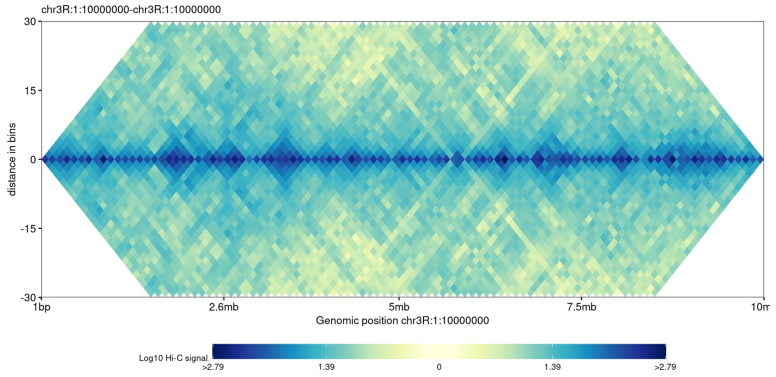
```
GenomicMatrix <- R6Class("GenomicMatrix",
    public = list (
        initialize = function(){

        },
        hdf.matrices.root = "Base.matrices",
        hdf.ranges.root = "Base.ranges",
        hdf.metadata.root = "Base.metadata",
        metadata.chrom.dataset = "chrominfo",
        hdf.matrix.name = "matrix",
        hdf.matrix.coverage = "chr1_bin_coverage",
        hdf.matrix.coverage.t = "chr2_bin_coverage",
        hdf.matrix.rowSums = "chr1_row_sums",
        hdf.matrix.colSums = "chr2_col_sums",
        hdf.matrix.sparsity = "sparsity",
        hdf.bintable.ranges.group = "Bintable",
        hdf.ranges.dataset.name = "ranges",
        hdf.ranges.offset.name = "offset",
        hdf.ranges.chr.name = "chr.names",
        hdf.ranges.lengths.name = "lengths",
        mcool.resolutions.name = "resolutions",
        cache.basename = "HiCBricks",
        cache.metacol.cols = "HiCBricks_MetaInfo",
        cache.metacol.dbid = "rid",
        cache.metacol.hashname = "hashname",
        cache.metacol.dirpath = "dirpath",
        brick.extension = "brick",
        Max.vector.size=104857600,
        brick.config.name = "HiCBricks_builder_config.json",
        mcool.available.normalisations = function(){
            Names <- c("Knight-Ruitz","Vanilla-coverage",
                "Vanilla-coverage-square-root","Iterative-Correction")
            Vector <- c("KR","VC","VC_SQRT","weight")
            names(Vector) <- Names
            return(Vector)
        },
        genomic.ranges.protected.names = c("seqnames", "strand", "seqlevels",
            "seqlengths", "isCircular", "start", "end", "width", "element"),
        genomic.ranges.metadata.functions = function(name = NULL){
            transformlist <- list(
                "strand" = strand,
                "seqlevels" = seqlevels,
                "seqlengths" = seqlengths,
                "width" = width)
            if(is.null(name)){
                return(names(transformlist))
            }
            return(transformlist[[name]])
        },
```

https://bioconductor.org/packages/devel/bioc/html/HiCBricks.html

Abstraction

- The ability to hide or remove complexity from re-used pieces of code
- Create or abstract away shared attributes or methods

- A visualization function implemented across 134 lines of code with many internal functions being used.



58 +128 lines

9 lines

```
Brick_vizart_plot_heatmap = function(File, Bricks, resolution,
    x_coords, y_coords, FUN = NULL, value_cap = NULL,
    distance = NULL, rotate = FALSE, x_axis = TRUE, x_axis_title = NULL,
    y_axis = TRUE, y_axis_title = NULL, title = NULL, legend_title = NULL,
    return_object=FALSE, x_axis_num_breaks = 5, y_axis_num_breaks = 5,
    palette, col_direction = 1, extrapolate_on = NULL,
    x_axis_text_size = 10, y_axis_text_size = 10, text_size = 10,
    legend_title_text_size = 8, legend_text_size = 8, title_size = 10,
    tad_ranges = NULL, group_col = NULL, tad_colour_col = NULL, colours = NULL,
    colours_names = NULL, cut_corners = FALSE, highlight_points = NULL,
    width = 10, height = 6, line_width = 0.5, units = "cm",
    legend_key_width = unit(3,"cm"), legend_key_height = unit(0.5,"cm")){

    if(!is.list(Bricks)){
        stop("Bricks expects an argument of type list.",
             " Please refer to the vignette to understand the parameter.")
    }
    Matrix.df <- Get_one_or_two_brick_regions(Bricks = Bricks,
        resolution = resolution, x_coords = x_coords,
        y_coords = y_coords, distance = distance,
        value_cap = value_cap, FUN = FUN)
    if(nrow(Matrix.df)==0){
        stop("The matrix was empty!")
    }
    list_of_coords <- list("x_coords" = x_coords, "y_coords" = y_coords)

    Parsed_string <- ._Parse_genomic_coordinates(list_of_coords)
    x.coord.parsed <- Parsed_string[["x_coords"]]
    y.coord.parsed <- Parsed_string[["y_coords"]]
    x.coord.breaks <- make_axis_coord_breaks(from = min(Matrix.df$row),
        to = max(Matrix.df$row), how.many = x_axis_num_breaks,
        two.sample = FALSE)
    x_axis.coord.labs <- Make_axis_labels(Brick = Bricks[[1]],
        resolution = resolution, chr = x.coord.parsed['chr'],
        positions = x.coord.breaks)
```

Inheritance

- Objects containing methods and attributes make available their public methods and attributes to any object that extends their functionality.

- Data.table is a package that extends the data.frame class.
- Therefore, the imported object can be manipulated using data.frame methods
- However, these methods are not explicitly defined within data.table

```
55:  NA
56:  NA
57:  NA
58:  NA
59:  NA
60:  NA
61:  NA
62:  NA
63:  NA
64:  NA
65:  NA
66:  NA
67:  NA
68:  NA
69:  NA
70:  NA
71:  NA
    V13
> class(fread("Downloads/f.1622124668.861.txt"))
[1] "data.table" "data.frame"
> class(data.table::fread("Downloads/f.1622124668.861.txt"))
[1] "data.table" "data.frame"
```

What about language?

- Languages are encode using standard ASCII table
- ASCII is maintained by Unicode

Dec	Hex	Binary	HTML	Char	Description
97	61	01100001	&#97;	a	
98	62	01100010	&#98;	b	
99	63	01100011	&#99;	c	
100	64	01100100	&#100;	d	
101	65	01100101	&#101;	e	
102	66	01100110	&#102;	f	
103	67	01100111	&#103;	g	
104	68	01101000	&#104;	h	
105	69	01101001	&#105;	i	
106	6A	01101010	&#106;	j	
107	6B	01101011	&#107;	k	
108	6C	01101100	&#108;	l	
109	6D	01101101	&#109;	m	
110	6E	01101110	&#110;	n	
111	6F	01101111	&#111;	o	
112	70	01110000	&#112	p	
113	71	01110001	&#113;	q	

There’s a lot more love and joy in the world than sadness

- Unicode also maintains all of the current emojis
- Current emojis – 3,663 emojis
- Top ten emojis
  - 😂❤️🤣👍😭🙏😘🥰😍😊.

https://home.unicode.org/emoji/emoji-frequency/