

20 Aug 2025

Department of Computer Science
and Engineering



International Institute of
Information Technology,
Bhubaneswar

Introduction & Background

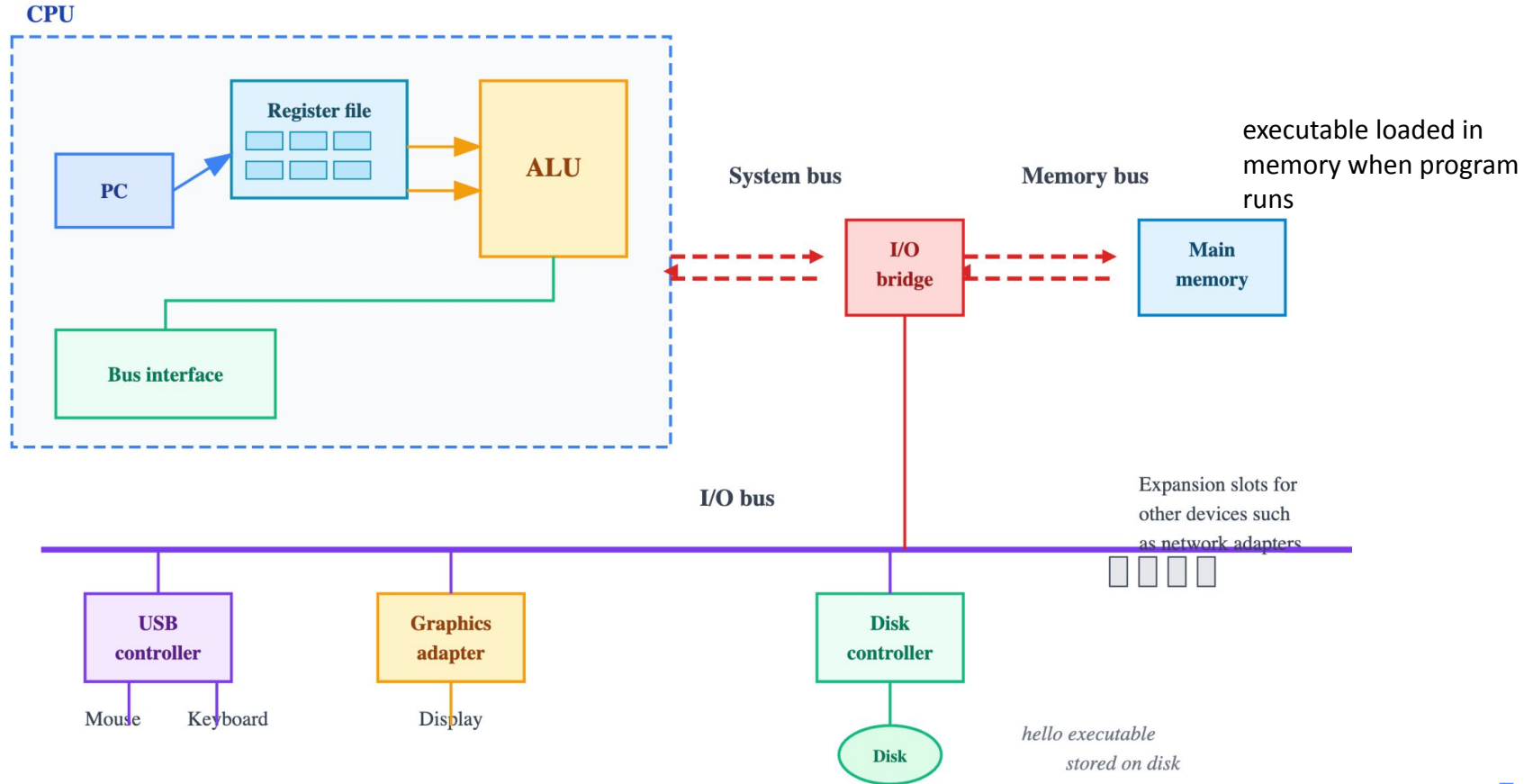


Utkalika Satapathy
utkalika@iiit-bh.ac.in

Topics to be covered

- 01 - Hardware Organization
- 02 - CPU ISA (Instruction Set Architecture)
- 03 - Memory/Storage Hierarchy
- 04 - Parts of Program Memory
- 05 - Memory Allocation
- 06 - Pointers and addresses
- 07 - Stack vs Heap

Hardware Organization



CPU ISA (Instruction Set Architecture)

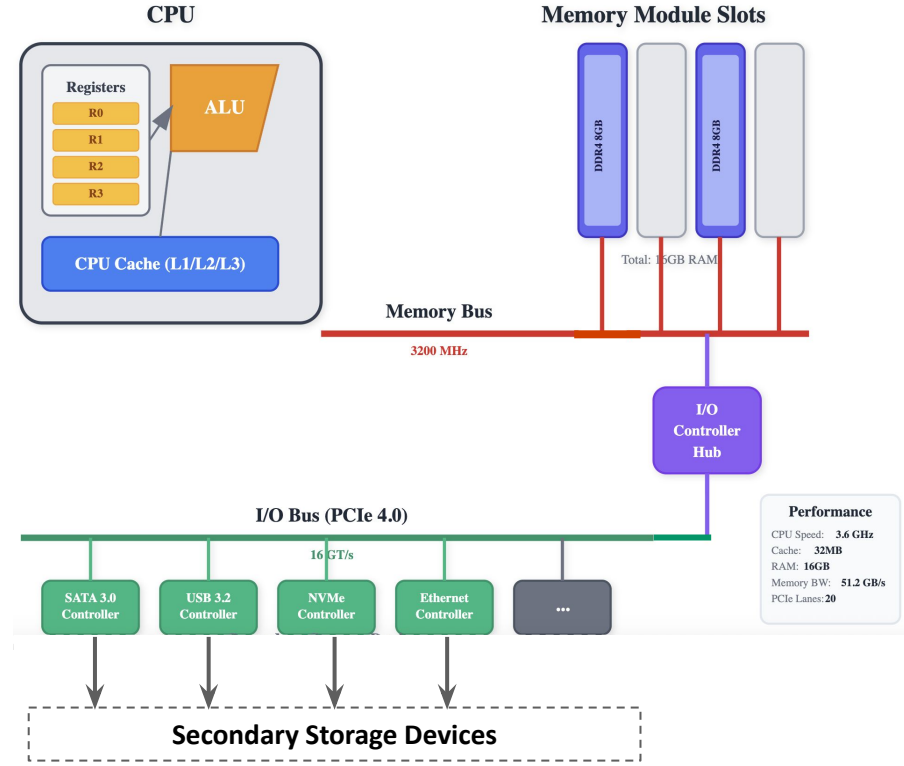
- Every CPU has a well-defined set of
 - **Instructions** that the hardware can execute
 - **Registers** for temporary storage of data within the CPU
- Instructions and registers specified by **ISA = Instruction Set Architecture**
 - Specific to CPU manufacturer (e.g., Intel CPUs follow x86 ISA)
- Registers: special registers (specific purpose) or general purpose
 - **Program counter (PC)** is special register, has memory address of the next instruction to execute on the CPU
 - General purpose registers can be used for anything, e.g., operands in instructions
- Size of registers defined by architecture (32 bit / 64 bit)

CPU instructions

- Some common examples of CPU instructions
 - **Load**: copy content from memory location register
 - **Store**: copy content from register memory location
 - **Arithmetic** and **logical** operations like add: $\text{reg1} + \text{reg2} \rightarrow \text{reg3}$, compare, ..
 - **Jump**: change value of PC
 - **Call**: invoke a function
- Simple model of CPU
 - Each clock cycle, **fetch** instruction at PC, **decode**, access required data, **execute**, **update** PC, **repeat**
 - PC increments to next instruction, or jumps to some other value
- Many optimizations to this simple model
 - **Pipelining**: run multiple instructions concurrently in a pipeline
 - Many more in modern CPUs to optimize #instructions executed per clock cycle

Memory/Storage Hierarchy

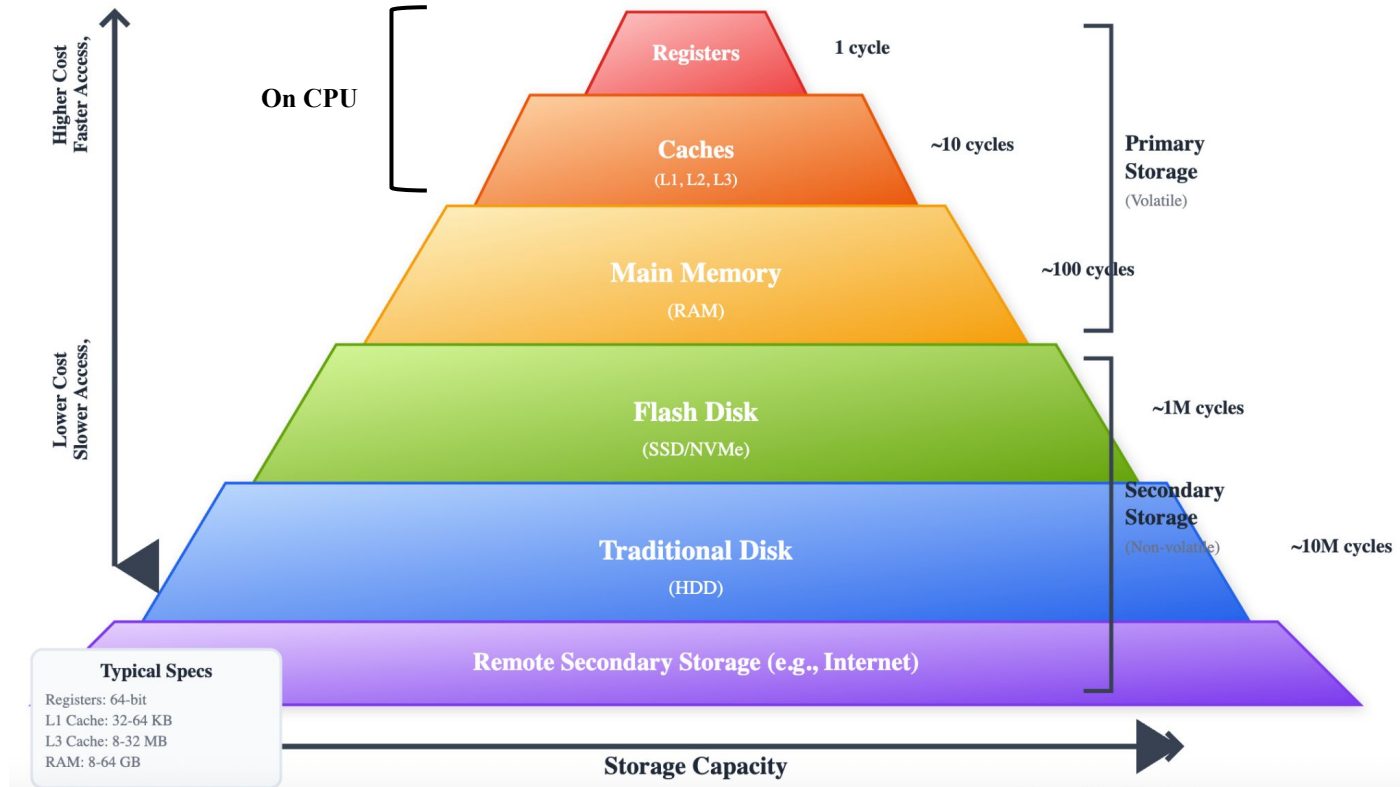
- Program executable loaded from secondary storage to main memory
- When CPU runs program, recently accessed instructions and data stored in CPU caches(faster access than DRAM)
- Registers in CPU provide temporary storage, e.g., hold operands



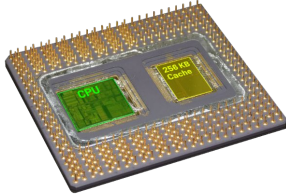
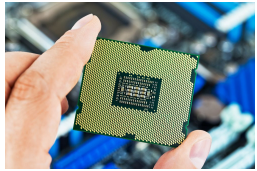
Memory/Storage Hierarchy

- Hierarchy of storage elements which store instructions and data
 - CPU registers (small number, accessed in <1 nanosec)
 - Multiple levels of CPU caches (few MB, 1-10 nanosec)
 - Main memory or RAM (few GB, ~100 nanosec)
 - Hard disk (few TB, ~1 millisec)
- Hard disk is non-volatile storage, rest are volatile
 - Hard disk stores files and other data persistently
- As you go down the hierarchy, memory access technology becomes cheaper, slower, less expensive
- CPU caches transparent to software, managed by hardware
 - Software only accesses memory, doesn't know if served from cache or DRAM

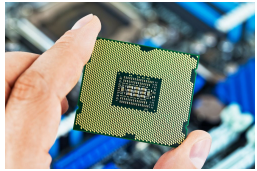
Memory/Storage Hierarchy



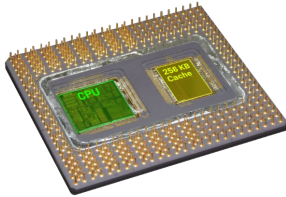
Memory/Storage Hierarchy



Memory/Storage Hierarchy



Register



Cache



RAM



Flash Disk

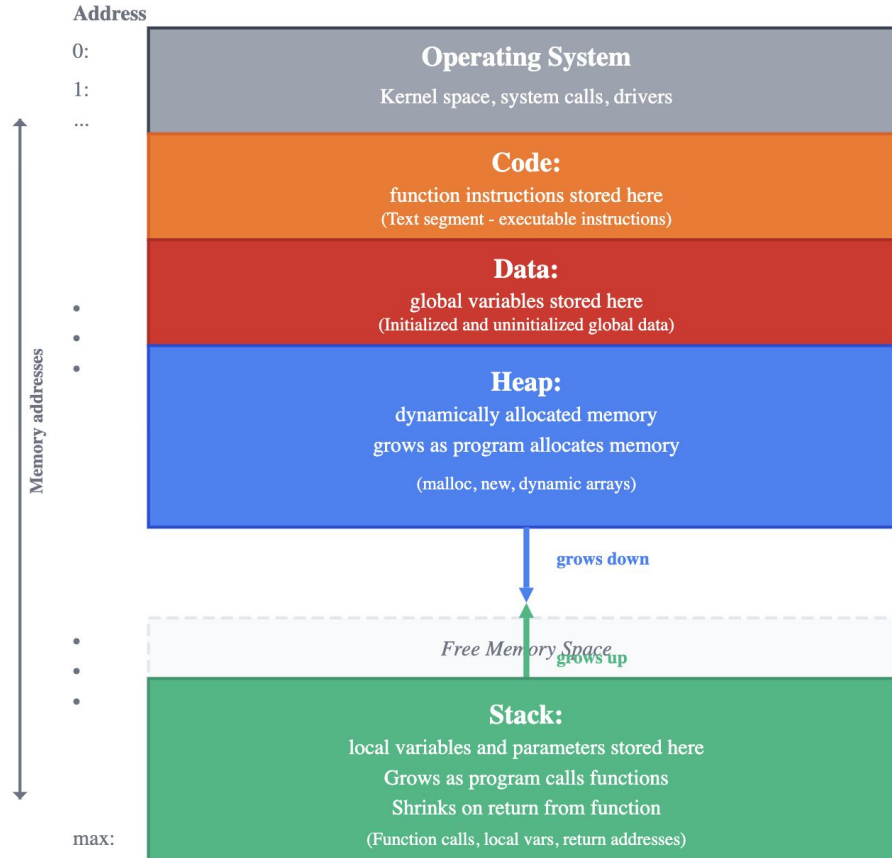


Hard Disk

Parts of Program Memory

- The memory of a running program in DRAM has the following components
 - Compiled code (instructions)
 - Compile-time data (global/static variables)
 - Runtime data on stack (function arguments, local variables, ...)
 - Runtime data on heap (dynamically allocated memory via malloc, ...)
- All instructions and data are assigned memory addresses, based on their location in memory
- Main memory contains user programs + code/data of OS

Parts of Program Memory



Memory Properties

Code:

- Read-only
- Fixed size

Data:

- Read/Write
- Fixed size

Heap:

- Dynamic size
- Manual control

Stack:

- Auto managed
- LIFO structure

Typical Sizes

Code: 1-100 MB
Data: 1-10 MB
Heap: 0-GB range
Stack: 1-8 MB

**varies by program*

Virtual Memory

Each process has
its own address
space layout

Example: Memory Allocation

- When is memory allocated for the various parts of this program?
 - Memory for global variable “g” allocated when ?
 - Memory for function arguments and local variables (a, b, x,y, z, ...) ?

```
int g;

int increment(int a){
    int b;
    b = a + 1;
    return b;
}

main (){
    int x, y;
    x = 1;
    y = increment(x);

    int *z = malloc(40);
}
```

Example: Memory Allocation

- When is memory allocated for the various parts of this program?
 - Memory for global variable “g” allocated when executable loaded into memory at start of execution
 - Memory for function arguments and local variables (a, b, x,y, z, ...) allocated (“pushed”) on stack when the corresponding function is called
- Why not allocate memory at start of program?

```
int g;  
  
int increment(int a){  
    int b;  
    b = a + 1;  
    return b;  
}  
  
main (){  
    int x, y;  
    x = 1;  
    y = increment(x);  
  
    int *z = malloc(40);  
}
```

Example: Memory Allocation

- When is memory allocated for the various parts of this program?
 - Memory for global variable “g” allocated when executable loaded into memory at start of execution
 - Memory for function arguments and local variables (a, b, x,y, z, ...) allocated (“pushed”) on stack when the corresponding function is called
- Why not allocate memory at start of program?
 - Because we do not know if/how many times the function will be called at runtime
 - Function variables “popped” from stack when function returns
- Memory requested dynamically via malloc is allocated on the heap at runtime, when malloc is invoked

```
int g;  
  
int increment(int a){  
    int b;  
    b = a + 1;  
    return b;  
}  
  
main (){  
    int x, y;  
    x = 1;  
    y = increment(x);  
  
    int *z = malloc(40);  
}
```

Pointers and addresses

- A pointer variable contains the memory address of another variable
- Note that these addresses are only logical addresses, and not the actual physical addresses in DRAM (why? Will cover later)
- Pointer variables contain space to only store the address, and the variable being pointed to must be declared/allocated separately
- Ensure pointer contains valid address before accessing it

```
/* Assuming an integer named x has already been declared, this code sets the value  
of x to 8. */
```

```
ptr = &x; /* initialize ptr to the address of x (ptr points to variable x) */  
*ptr = 8; /* the memory location ptr points to is assigned 8 */
```



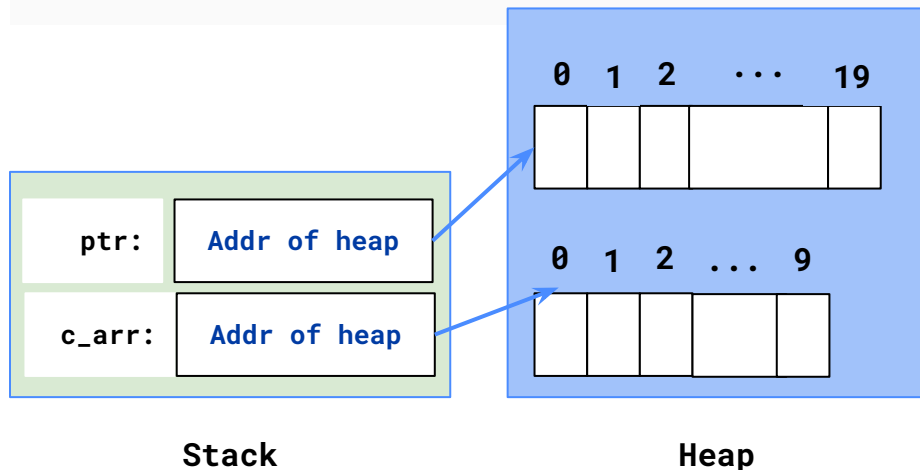
Stack vs Heap

- Functions like malloc allocate memory on heap and return start address of allocated chunk
- This heap address is stored in a pointer variable, which may be a local variable in a function, and hence located on the stack
- Dynamically allocated memory on heap must be explicitly freed up (in languages like C), else memory leak
 - Stack memory automatically popped when function returns

```
int *arr;  
char *c_arr;
```

```
/* allocate an array of 20 ints on the heap*/  
arr = malloc(sizeof(int) * 20);
```

```
/* allocate an array of 10 char on the heap*/  
c_arr = malloc(sizeof(int) * 10);
```



Next Class We Will Talk About

- What happens on a function call ?
- Role of OS in running a process
- Concurrent execution & CPU virtualization
- Context Switching
- User Mode and Kernel Mode
- System Calls
- Interrupts

Happy Learning !

We're Done.



Questions?

quibunnies.com

