Programming RISC-V using assembly and C language

by: Tassadaq Hussain

Director Centre for AI and BigData

Professor Department of Electrical Engineering

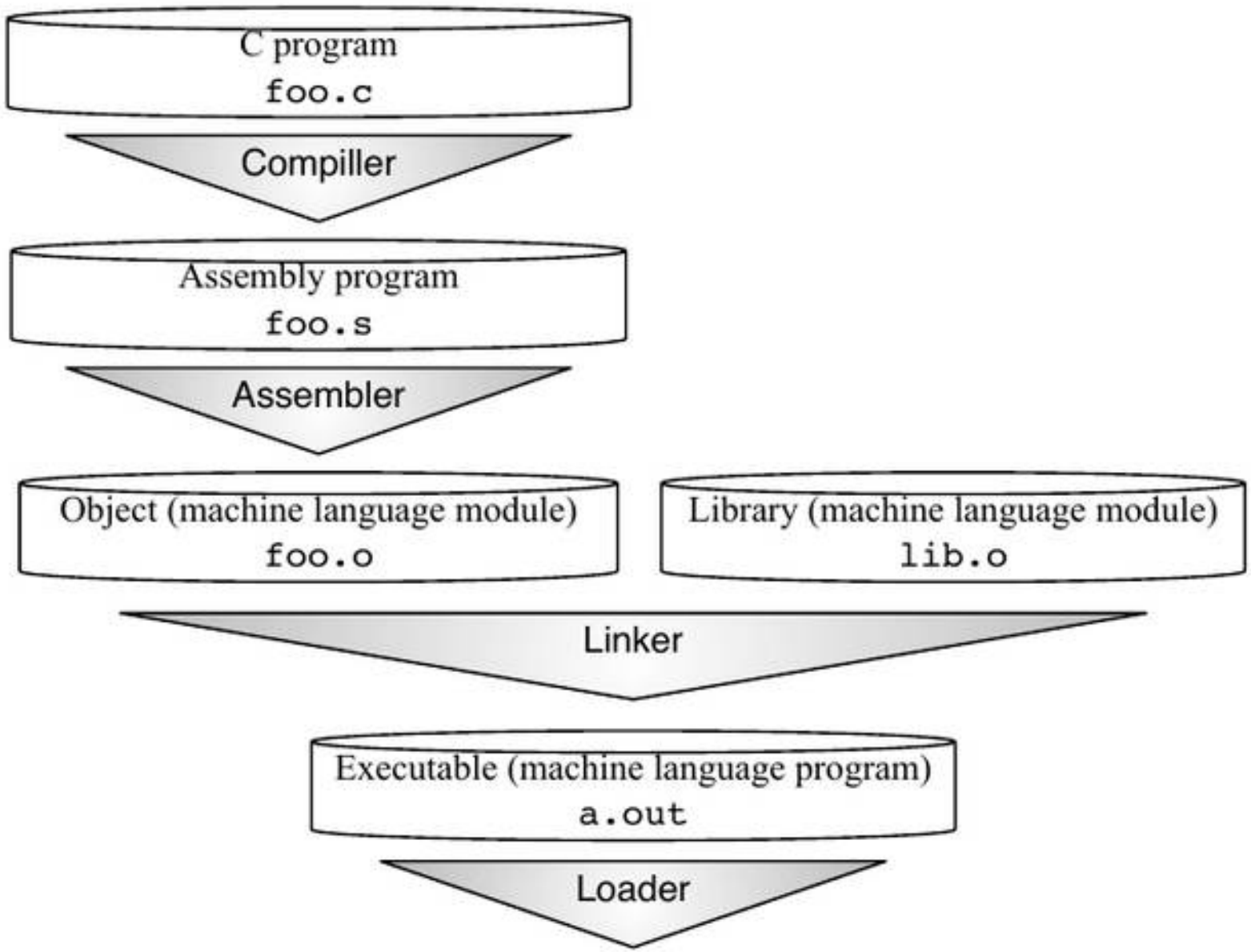
Namal University Mianwali

Collaborations:

Barcelona Supercomputing Center, Spain

European Network on High Performance and Embedded Architecture and Compilation

Pakistan Supercomputing Center


C program
foo.c
Compiller
Assembly program
foo.s
Assembler
Object (machine language module)
foo.o
Library (machine language module)
lib.o
Linker
Executable (machine language program)
a.out
Loader

RISCV GCC Assembler

_start:

ld s3 0x001121

Ld rs2 0x0022233

add rd, rs1, rs2

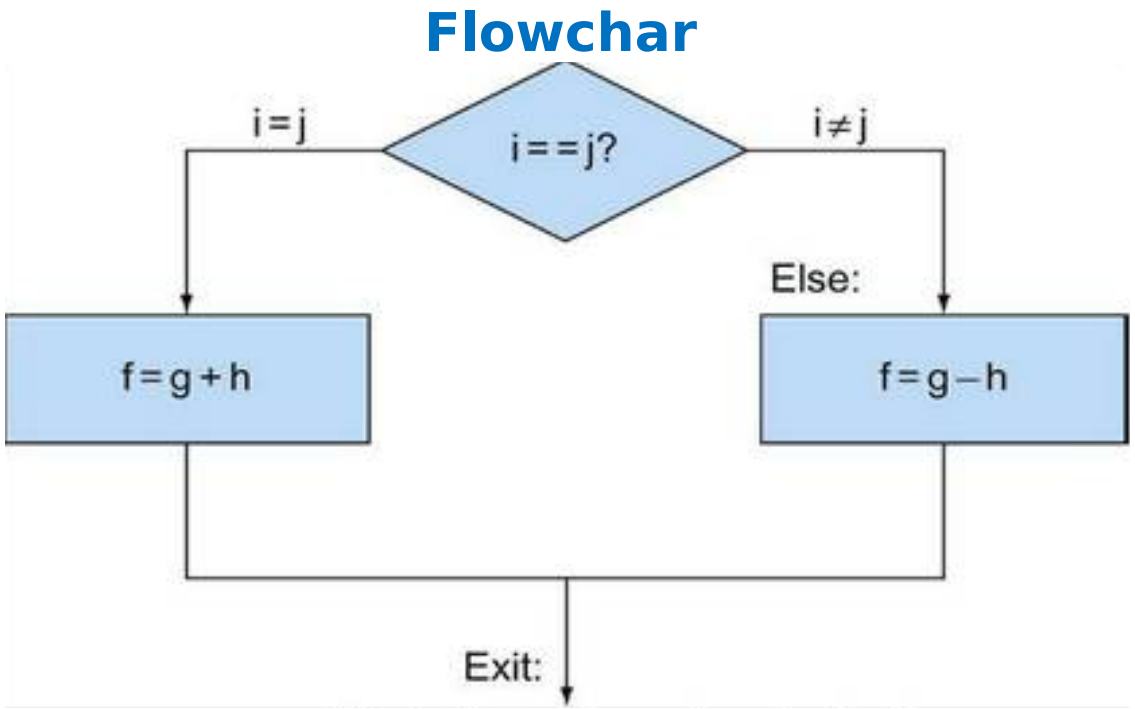
St rd 0x0000001

Assembly or C/C++

- Write Efficient Code
- Secure Application
- Multi-Threaded and Complex Program to run multiple devices (OS)
- Real-Time Applications for Real world Problems

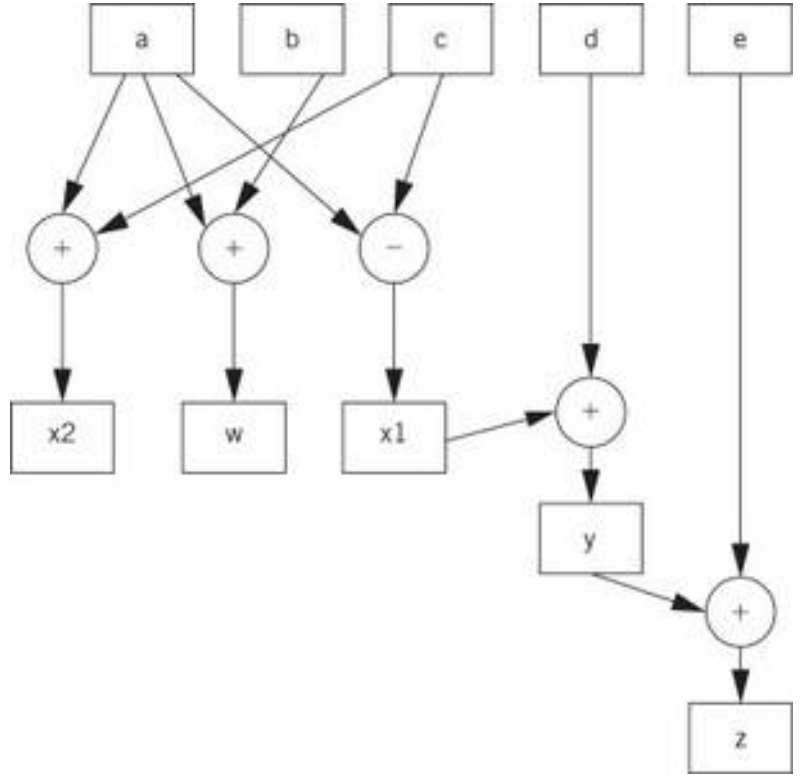
Programming RISC-V

- Problem
- Write it in your own words
- Make Pseudo Code
- Create Control and Data-flow Graph
- Program (C/C++, ASM)
- Debug
- Profile
- Optimize/Fine Tune
- Execute
- Test



Hazards

- Data Hazards: Instructions are waiting for data from other instructions.
- Control Hazards: Changes in instruction flow cause delays.
- Structural Hazards: Limited hardware resources cause delays.



```
// example.c
int global_var = 10;
int main() {
    int local_var = 5;
    int result = global_var +
local_var;
    return result;
}
```

riscv32-unknown-elf-gcc example.o -o example

- The compiler generates an object file in ELF format. This object file contains machine code, data, and metadata, organized into different sections like .text (code), .data (initialized data), and .bss (uninitialized data).

- Instruction Section: Contains the compiled machine code instructions (text section).
- Data Section: Contains initialized data (data section).
- The linker combines the code and data sections, resolves symbols, and sets up memory addresses.
- The linker script defines how different sections are mapped into the memory of the microcontroller.
- It specifies memory regions and assigns addresses to different sections of the code and data.

```
MEMORY
{
    ROM (rx) : ORIGIN = 0x08000000, LENGTH = 512K
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
}

SECTIONS
{
    .text : {
        *(.text)
    } > ROM

    .data : {
        *(.data)
    } > RAM
}
```

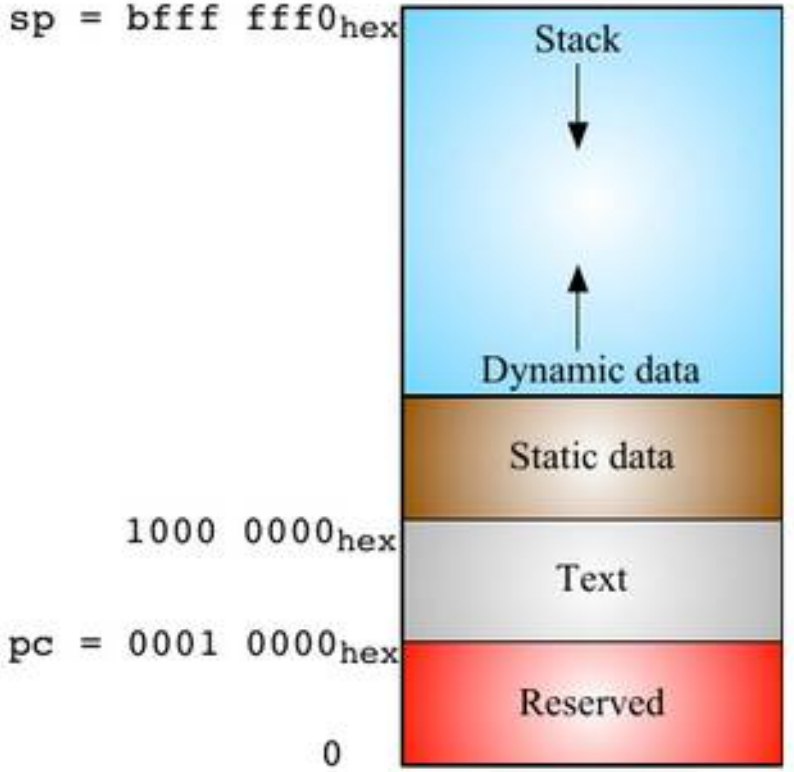
- Next step involves using a programmer or debugger tool to flash the firmware into the RISCV System.
  - Instruction Memory: The code from the .text section is loaded into the system instruction memory.
  - Data Memory: The initialized data from the .data section is loaded into the system data memory.

Linker Script: Program and Data Memory Allocation

The high addresses are the top of the figure and the low addresses are the bottom.

The stack pointer (sp) starts at BFFF FFF0 hex and grows down toward the Static data. The text (program code) starts at 0001 0000hex and includes the statically-linked libraries.

The Static data starts immediately above the text region; in this example, we assume that address is 1000 0000hex . Dynamic data, allocated in C by malloc(), is just above the Static data. Called the heap, it grows upward toward the stack. It includes the dynamically-linked libraries.



Topics

- **Storage Classes in C**
- Functions in C
- Memory Layout in C
- Arrays
- Operators in C

Decimal, Hexadecimal, Octal, and Character Values in C

- **Decimal is the default number format**

```
int m,n;                    //16-bit signed numbers
m = 453; n = -25;
```

- **Hexadecimal: preface value with 0x or 0X**

```
m = 0xF312; n = -0x12E4;
```

- **Octal: preface value with zero (0)**

```
m = 0453; n = -023;
```

Don't use leading zeros on "decimal" values. They will be interpreted as octal.

- **Character: character in single quotes, or ASCII value following "slash"**

```
m = 'a'; //ASCII value 0x61
n = '\13'; //ASCII value 13 is the "return" character
```

- **String (array) of characters:**

```
unsigned char k[7];
strcpy(m,"hello\n");
        //k[0]='h', k[1]='e', k[2]='l', k[3]='l', k[4]='o',
        //k[5]=13 or '\n' (ASCII new line character),
        //k[6]=0 or '\0' (null character – end of string)
```

Syntax => No Prefix

Syntax => Prefix 0x

Syntax => Prefix 0

Single quotes for character literals, or ASCII value with a backslash

Double quotes for Strings with null Terminator \n

Program Variables in C Programming

Definition:

A variable is an addressable storage location used to hold information that can be referenced and manipulated by the program.

Declaration:

Purpose: To specify the size, type, and name of the variable.

Example:

```
int x, y, z;  // Declares 3 variables of type “int” (integer)
char a, b;    // Declares 2 variables of type “char” (character)
```

Storage Allocation:

- **Registers:** Fast, limited storage for frequently accessed variables.
- **RAM:** Dynamic memory for variables that change during program execution.
- **ROM/Flash:** Permanent storage, typically for constants or read-only data.

Variable Declaration in C

Basic syntax for variable declaring in C is as follows

data_type variable name = value;

Example:

```
int z = 35; // declare and initialize variable z with value
35.
```

Note 1: The Data type and the Value used to store in the Variable must match.

Note 2: All declaration statements must end with a semi-colon (;)

The refined syntax for declaring variables in C can be quite comprehensive, incorporating storage classes, type qualifiers, type modifiers, data types, pointers, arrays, and initial values. Adding these parameter the syntax will look like

storage-class type-qualifier type-modifier data-type *pointer variable-name[size] = initial-value;

Example:

```
static const unsigned int *configFlagPtr = (int
*)0x40021000;
```

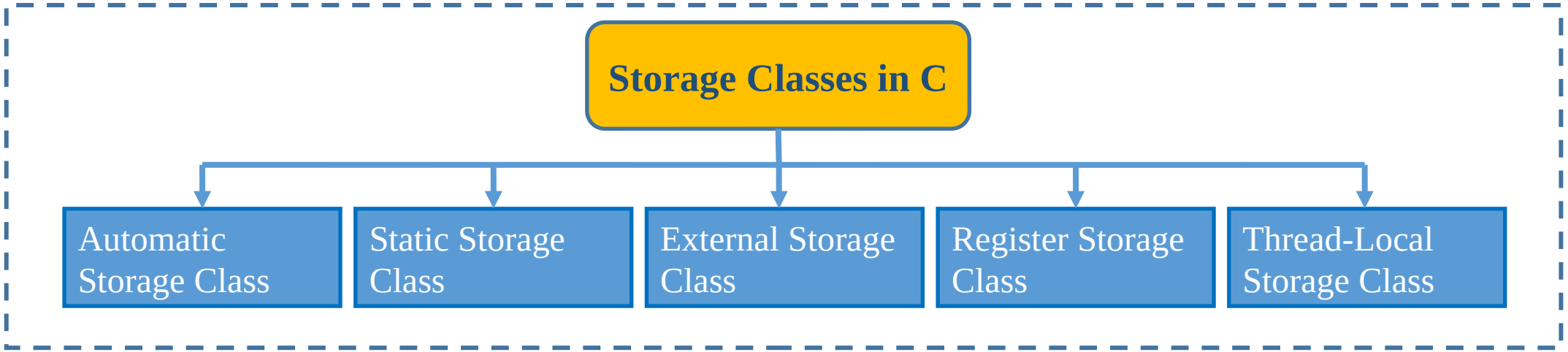
Storage-class, type-qualifier, type-modifier, pointer, array-size are all optional.

Storage Classes in C

In C programming, storage classes determine following characteristics of variables and functions.

1. **Scope:** Refers to variables or functions declared in another file or elsewhere in the same file.
2. **Lifetime:** Exists for the duration of the program.
3. **Visibility:** Visibility determines where a variable or function can be referenced within the program.
4. **Memory location:** The actual variable or function is defined elsewhere, usually in a different file.

Storage Classes control how variables are stored, accessed, and managed throughout the program. The key storage classes in C are:



Storage Classes in C: Automatic

Automatic Variable

- It is declared inside the function where it is used
- It are created when function is called and destroyed when the function is exited
- It is local to function and also called private variables
- It is also called as local or internal variables

Auto is Default storage class for all the local variables therefore, no need to use keyword auto

Example:

```
void function1 (void)
main()
{           int m =1000;
            function2();
            prinf(“%d\n”, m) }
Void function1(void)
{           int m =10;
            printf(“%d”\n,m) }
```

Storage Classes in C: Automatic

Static Variable

- It persists at the function until the end of the program
- The keyword Static is used for declaration static int x;
- Static may be internal type or external type.
- Internal means it is declared inside the function
- The scope is up to end of the function
- It is used to retain the values between functions calls

Example:

```
void counterFunction() {
    static int count = 0;  // Static variable retains its
value between function calls
    count++;
    printf("Count: %d\n", count);
}
int main() {
    counterFunction();  // Output: Count: 1
    counterFunction();  // Output: Count: 2
    counterFunction();  // Output: Count: 3
    return 0;
}
```

Storage Classes in C

Scope

The scope of a variable or function refers to the region of the program where the variable or function can be accessed or used.

Types of Scope:

1. **Local Scope**: The region within a function or block where a variable or function is defined. Example: Variables declared inside a function or a block are local to that function or block.

Code Example:

```
void func() {
 int x = 10;  // x has local scope within func }
```

2. **Global Scope**: The region of the program where a variable or function is accessible throughout the entire program, typically from its point of declaration until the end of the file.

Example: Variables and functions declared outside of all functions.

Code Example:

```
int globalVar = 20;  // globalVar has global scope
void func() { // can use globalVar here }
```

Storage Classes in C

Visibility

Visibility determines where a variable or function can be referenced within the program. It specifies the extent to which a variable or function is accessible.

Types of Visibility:

1. **Internal Visibility:** Refers to variables or functions that are only accessible within the file they are declared. This is typically controlled using the static keyword.

Example:

```
static int internalVar = 30;  // Only visible within the same file
```

2. **External Visibility:** Definition: Refers to variables or functions that are accessible across different files. This is typically achieved using the extern keyword.

Example:

```
// File1.c

int externalVar = 40;  // Visible to other files

// File2.c

extern int externalVar;  // Reference to externalVar defined in File1.c
```

Storage Classes in C

Lifetime

The lifetime of a variable or function refers to the duration of time that the variable or function exists in memory and retains its value.

Types of Lifetime:

1. Automatic Lifetime: Variables with automatic lifetime are created when a function or block is entered and destroyed when it is exited. They are usually stored on the stack.

Example:

```
void func()
{ int autoVar = 50;  // Lifetime is limited to the duration of func }
```

2. Static Lifetime: Variables with static lifetime are created when the program starts and destroyed when the program ends. They retain their value between function calls or across files.

Example:

```
void func() {
static int staticVar = 60;  // Lifetime is the entire program duration }
```

3. Dynamic Lifetime: Variables with dynamic lifetime are allocated and deallocated manually using functions like malloc() and free(). Their lifetime is controlled by the programmer.

```
void func() {
int* dynamicVar = (int*)malloc(sizeof(int));  // Dynamic allocation
free(dynamicVar);  // Manual deallocation
}
```

Topics

- Storage Classes in C
- **Functions in C**
- Memory Layout in C
- Arrays
- Operators in C

Functions in C

- A function in C is a set of statements that when called perform some **specific task**.
- It is the basic building block of a C program that provides **modularity** and code **reusability**.
- The programming statements of a function are enclosed within { } braces, having certain meanings and performing certain operations.
- They are also called subroutines or procedures in other languages.

Syntax of Functions in C

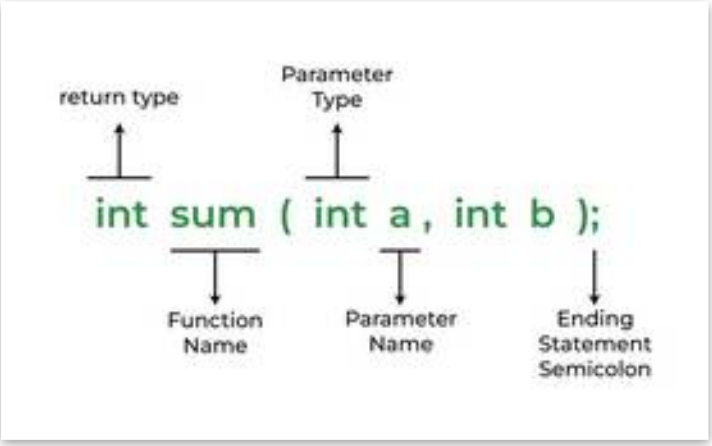
The syntax of function can be divided into 3 aspects:

- **Function Declaration**

```
return_type name_of_the_function (parameter_1, parameter_2);
```

Example:

```
int sum(int a, int b);  // Function declaration with parameter names
int sum(int , int);     // Function declaration without parameter names
```



Functions in C

➢ Function Definition

The function definition consists of actual statements which are executed when the function is called (i.e. when the program control comes to the function).

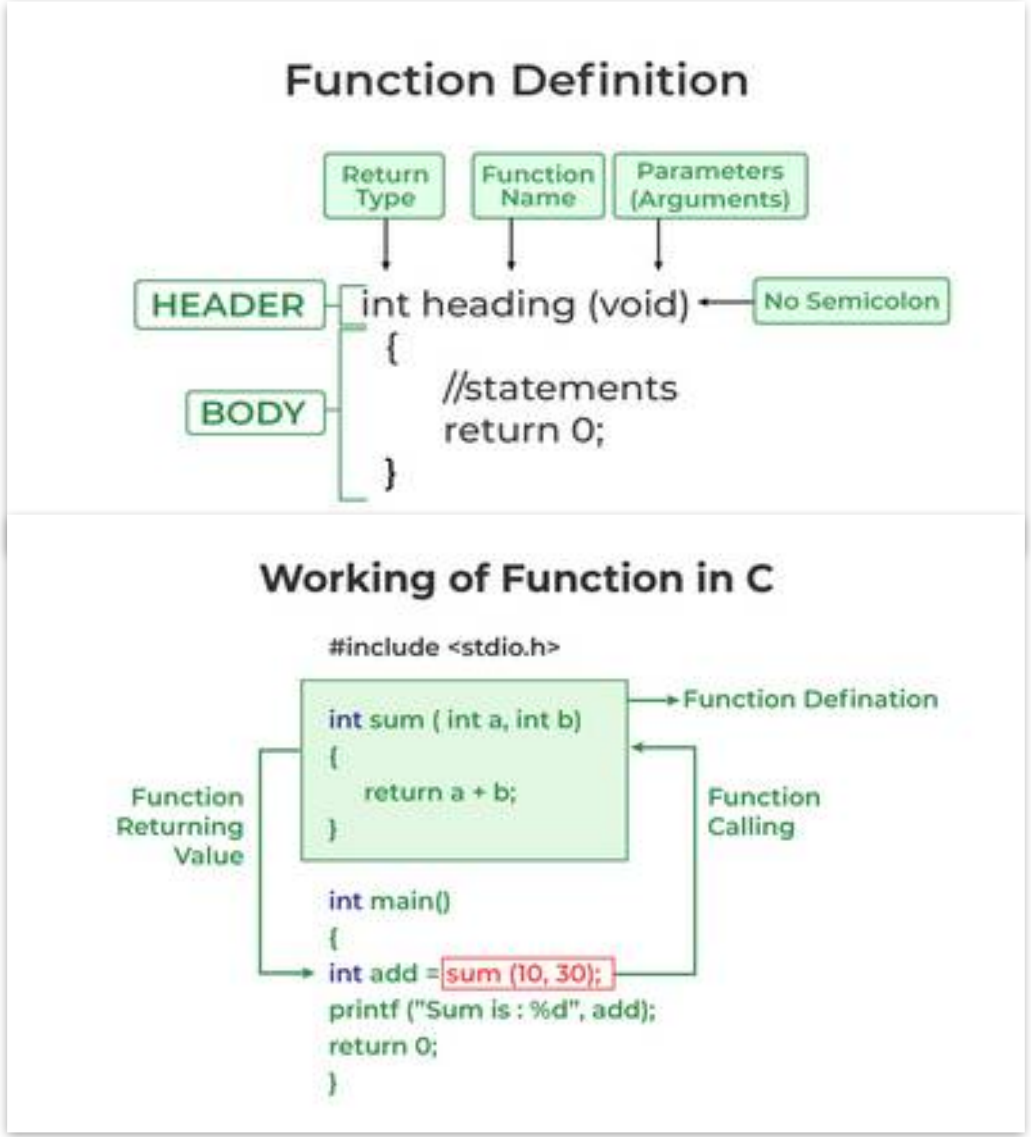
```
return_type function_name (parameter_1, parameter_2)
{
        // body of the function
}
```

➢ Function Calls

A function call is a statement that instructs the compiler to execute the function. We use the function name and parameters in the function call.

In the example,

- The first sum function is called and 10,30 are passed to the sum function.
- After the function call sum of a and b is returned and control is also returned back to the main function of the program.



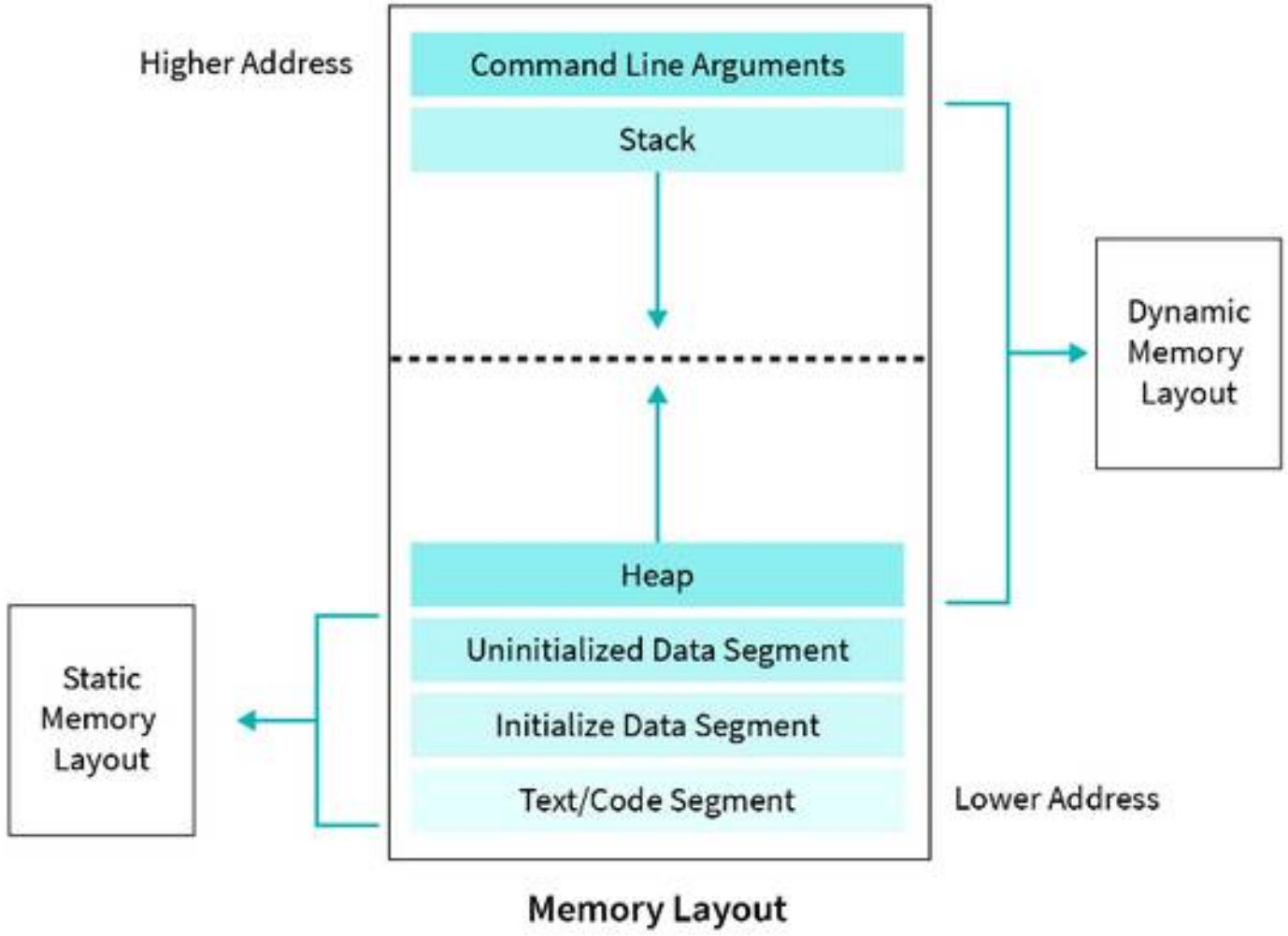
Topics

- Introduction to Embedded C Programming
- Storage Classes in C
- Functions in C
- **Memory Layout in C**
- Arrays
- Operators in C

Memory Layout in C

A typical memory representation of a C program consists of the following sections.

- **Text/Code segment (i.e. instructions)**
- **Initialized data segment**
- **Uninitialized data segment (bss)**
- **Heap**
- **Stack**



Memory Layout in C

Text/Code Segment

- A text segment, also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.
- As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.
- Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on.
- The text segment is often read-only, to prevent a program from accidentally modifying its instructions.

Example:

```
 int global_var = 5
// A function (text segment)
void print_message()
{
printf("Hello, World!\n");
}
int main()
{
print_message();          // Calls the function in the text segment return 0;
}
```

Segment	
Text Segment	void print_message() {...}
Initialized Data Segment	- int global_var = 5
Uninitialized Data Segment	-
Stack/Heap	-

Memory Layout in C Code

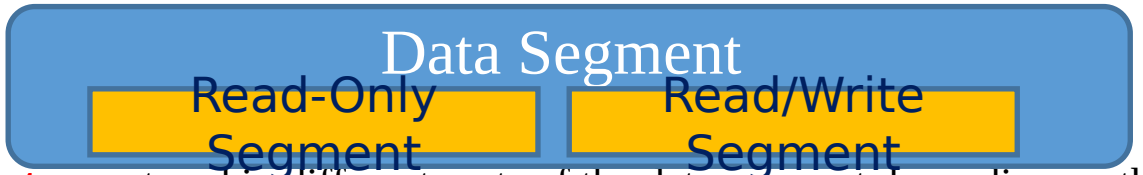
Initialized Data Segment

A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

```
int global_var = 10;     // Global variable initialized with 10
static int static_var = 20; // Static variable initialized with 20
```

Note that, the data segment is not read-only, since the values of the variables can be altered at run time.

Lifetime: Variables in the data segment exist for the lifetime of the program. They are initialized at program startup and persist until the program terminates.



In C, **variables** and **constants** are stored in different parts of the data segment depending on their initialization and attributes.

Type	Example	Memory Segment
Global Variable:	int debug = 1;	initialized read-write area
Global Constants:	const char* string = "hello world";	initialized read-only area
Global Static Variables	static int globalStatic = 20;	initialized read-write area
Static Variables in	void myFunction() {	initialized read-write

Memory Layout in C Code

Uninitialized Data Segment (bss)

- Also called the “BSS” segment (Block Started by Symbol).
- Contains global and static variables that are either:
  - Not explicitly initialized in the source code.
  - Initialized to zero.

Characteristics

- **Initialization:** The compiler initializes all variables in the BSS segment to zero before the program starts executing.
- **Memory Allocation:** The BSS segment occupies space in memory but does not store actual values; instead, it reserves space and initializes it to zero.
- **Memory Layout:** Comes after the initialized data segment in memory.

Examples:

```
static int i; // Static variables uninitialized
int j;                    // Global variables uninitialized
```

Memory Layout in C Code

Stack:

- The stack is a region of memory that stores temporary data, following a Last In, First Out (LIFO) structure.
- Traditionally, it adjoined the heap and grew in the opposite direction.

Characteristics:

- **Memory Layout:**
  - The stack is typically located in the higher parts of memory and grows towards lower addresses.
  - In modern systems with large address spaces and virtual memory, the stack and heap can be placed almost anywhere, but they still generally grow in opposite directions.

- **Stack Pointer:**
  - A stack pointer register keeps track of the top of the stack.
  - Adjusted each time a value is pushed onto or popped from the stack.

- **Stack Frame:** The data associated with a function call is stored in a stack frame.

  Stack Frame at minimum includes:
  - Return Address: Address to return to after the function call is complete.
  - May also include local variables, function parameters, etc.

Memory Layout in C Code

The **size command** is used to check the sizes (in bytes) of these different memory segments.

Simple Program

```c
#include<stdio.h>

int main() {
    return 0;
}
```

```
~$ gcc file_1.c -o file_1
~$ size file_1
   text    data     bss     dec     hex filename
   1418     544       8    1970     7b2 file_1
```

Adding one global variable in program

```c
#include<stdio.h>

int global_variable = 5;

int main() {
    return 0;
}
```

```
~$ gcc file_1.c -o file_1
~$ size file_1
   text    data     bss     dec     hex filename
   1418     548       4    1970     7b2 file_1
```

Adding one global variable increased memory allocated by data segment (Initialized data segment) by 4 bytes, which is the actual memory size of 1 variable of type integer (sizeof(global_variable)).

Task: Day 3 Embedded C Programming

Prepare a brief report explaining the operation of stack memory with respect to function calls and the phenomenon of stack overflow.

Topics

- Introduction to Embedded C Programming
- Storage Classes in C
- **Functions in C**
- Memory Layout in C
- Arrays
- Operators in C

Arrays in C Programming

Definition:

An array is a collection of data elements stored in consecutive memory locations. The array begins at a named address and contains a fixed number of elements.

----------One-Dimensional Arrays----------

Declaration:

Syntax: type arrayName[size];

Example Code:

```
int n[5];          // Declares an array of 5 integers
n[3] = 5;      // Sets the value of the 4th element (index 3) to 5
```

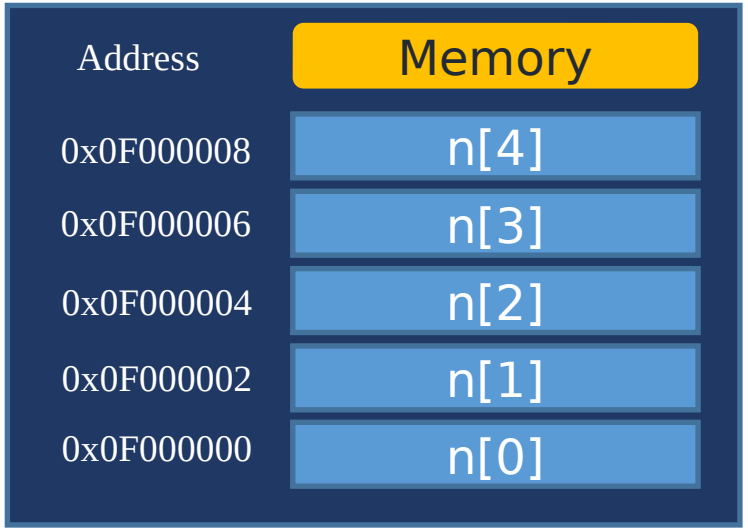
Array Indexing:

Indices: Start from 0 to N-1 where N is the number of elements.

Element Access: Access elements using arrayName[index].

Memory Layout: Array Elements: n[0] | n[1] | n[2] | n[3] | n[4]

Address	Value
A= (base Address)	n[0]
A+2	n[1]
A+4	n[2]
A+6	n[3]
A+8	n[4]



Arrays in C Programming

Address	Value
A(base Address)	n[0][0]
A+2	n[0] [1]
A+4	n[0] [2]
A+6	n[0] [3]
A+8	n[1] [0]
A+10	n[1] [1]
A+12	n[1] [2]

----------**Two-Dimensional Arrays**----------

Declaration:

Syntax: type arrayName [rows][columns];

Example Code:

```
int matrix[3][4];  // Declares a 2D array with 3 rows and 4 columns
matrix[1][2] = 7;  // Sets the value of the element in the 2nd row and 3rd column to 7
```

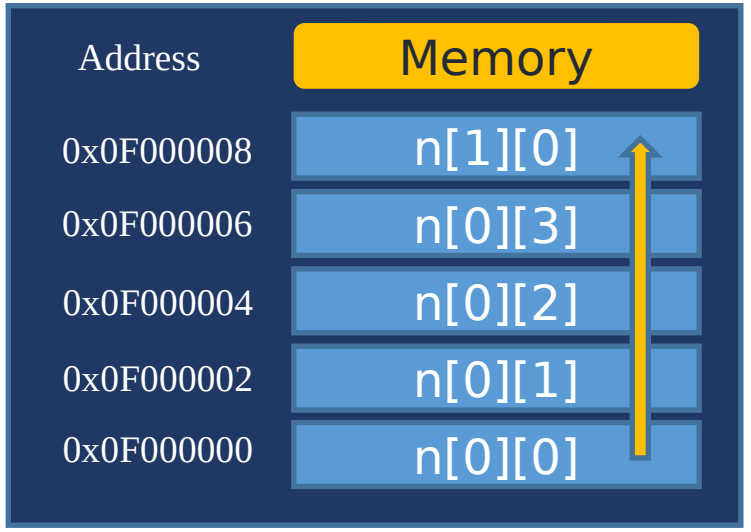
Array Indexing:

Indices: Start from 0,0 to N-1,N-1 where N is the number of elements.

Element Access: Access elements using arrayName[index].

Memory Layout: Array Elements (for matrix[3][4]):

matrix[0][0] | matrix[0][1] | matrix[0][2] | matrix[0][3]
matrix[1][0] | matrix[1][1] | matrix[1][2] | matrix[1][3]
matrix[2][0] | matrix[2][1] | matrix[2][2] | matrix[2][3]



Operators in C

An operator in C can be defined as the symbol that helps us to perform some specific mathematical, relational, bitwise, conditional, or logical computations on values and variables. The values and variables used with operators are called operands. So we can say that the operators are the symbols that perform operations on operands.

Types of Operators in C

C language provides a wide range of operators that can be classified into 6 types based on their functionality:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Other Operators

OPERATOR	TYPE	ASSOCIAVITY
() [] . ->		left-to-right
++ -- + - ! ~ (type) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

Arithmetic Operators

The arithmetic operators are used to perform arithmetic/mathematical operations on operands.

There are 9 arithmetic operators in C language:

S. No.	Symbol	Operator	Description	Syntax
1	**+**	**Plus**	Adds two numeric values.	**a + b**
2	**–**	**Minus**	Subtracts right operand from left operand.	**a – b**
3	*	**Multiply**	Multiply two numeric values.	**a * b**
4	/	**Divide**	Divide two numeric values.	**a / b**
5	**%**	**Modulus**	Returns the remainder after diving the left operand with the right operand.	**a % b**
6	**+**	**Unary Plus**	Used to specify the positive values.	**+a**
7	**–**	**Unary Minus**	Flips the sign of the value.	**-a**
8	**++**	**Increment**	Increases the value of the operand by 1.	a++
9	**—**	**Decrement**	Decreases the value of the operand by 1.	a–

Arithmetic Operators Example

Example Code:

```
int main()
{
        int a = 25, b = 5;
 // using operators and printing results
    printf("a + b = %d\n", a + b);
    printf("a - b = %d\n", a - b);
    printf("a * b = %d\n", a * b);
    printf("a / b = %d\n", a / b);
    printf("a % b = %d\n", a % b);
    printf("+a = %d\n", +a);
    printf("-a = %d\n", -a);
    printf("a++ = %d\n", a++);
    printf("a-- = %d\n", a--);

    return 0;
}
```

Output

```
a + b = 30
a - b = 20
a * b = 125
a / b = 5
a % b = 0
+a = 25
-a = -25
a++ = 25
a-- = 26
```

Relational Operators in C

Relational Operators in C

The relational operators in C are used for the comparison of the two operands. All these operators are binary operators that return true or false values as the result of comparison.

These are a total of 6 relational operators in C:

S. No.	Symbol	Operator	Description	Syntax
1	**<**	**Less than**	Returns true if the left operand is less than the right operand. Else false	**a < b**
2	**>**	**Greater than**	Returns true if the left operand is greater than the right operand. Else false	**a > b**
3	**<=**	**Less than or equal to**	Returns true if the left operand is less than or equal to the right operand. Else false	**a <= b**
4	**>=**	**Greater than or equal to**	Returns true if the left operand is greater than or equal to right operand. Else false	**a >= b**
5	**==**	**Equal to**	Returns true if both the operands are equal.	**a == b**
6	**!=**	**Not equal to**	Returns true if both the operands are NOT equal.	**a != b**

Relational Operators in C

Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either **true** or **false**.

S. No.	Symbol	Operator	Description	Syntax
1	**&&**	**Logical AND**	Returns true if both the operands are true.	**a && b**
2	**||**	**Logical OR**	Returns true if both or any of the operand is true.	**a || b**
3	**!**	**Logical NOT**	Returns true if the operand is false.	**!a**

Example

```
int main()
{
        int a = 25, b = 5;
        // using operators and printing
        results
            printf("a && b : %d\n", a && b);
            printf("a || b : %d\n", a || b);
            printf("!a: %d\n", !a);
        return 0;}
```

Output

```
a && b : 1
a || b : 1
!a: 0
```

Relational Operators Example

```
int main()
{

    int a = 25, b = 5;

    // using operators and printing results
    printf("a & b: %d\n", a & b);
    printf("a | b: %d\n", a | b);
    printf("a ^ b: %d\n", a ^ b);
    printf("~a: %d\n", ~a);
    printf("a >> b: %d\n", a >> b);
    printf("a << b: %d\n", a << b);

    return 0;
}
```

Output

```
a < b  : 0
a > b  : 1
a <= b: 0
a >= b: 1
a == b: 0
a != b : 1
```

Bitwise Operators in C

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing. There are 6 bitwise operators in C:

S. No.	Symbol	Operator	Description	Syntax
1	**&**	**Bitwise AND**	Performs bit-by-bit AND operation and returns the result.	**a & b**
2	**|**	**Bitwise OR**	Performs bit-by-bit OR operation and returns the result.	**a | b**
3	**^**	**Bitwise XOR**	Performs bit-by-bit XOR operation and returns the result.	**a ^ b**
4	**~**	**Bitwise First Complement**	Flips all the set and unset bits on the number.	**~a**
5	**<<**	**Bitwise Leftshift**	Shifts the number in binary form by one place in the operation and returns the result.	**a << b**
6	**>>**	**Bitwise Rightshilft**	Shifts the number in binary form by one place in the operation and returns the result.	**a >> b**

Bitwise Operators: AND, OR, XOR, ~

```
C = A & B;          A   0 1 1 0 0 1 1 0
(AND)               B   1 0 1 1 0 0 1 1
                    C   0 0 1 0 0 0 1 0

C = A | B;          A   0 1 1 0 0 1 0 0
(OR)                B   0 0 0 1 0 0 0 0
                    C   0 1 1 1 0 1 0 0

C = A ^ B;          A   0 1 1 0 0 1 0 0
(XOR)               B   1 0 1 1 0 0 1 1
                    C   1 1 0 1 0 1 1 1

B = ~A;             A   0 1 1 0 0 1 0 0
(COMPLEMENT)        B   1 0 0 1 1 0 1 1
```

Bitwise Operators: Bit Masking

```
C = A & 0xFE;    A     a b c d e f g h
               0xFE    1 1 1 1 1 1 1 0
                 C     a b c d e f g 0

C = A & 0x01;    A     a b c d e f g h
               0xFE    0 0 0 0 0 0 0 1
                 C     0 0 0 0 0 0 0 h

C = A | 0x01;    A     a b c d e f g h
               0x01    0 0 0 0 0 0 0 1
                 C     a b c d e f g 1

C = A ^ 0x01;    A     a b c d e f g h
               0x01    0 0 0 0 0 0 0 1
                 C     a b c d e f g h'
```

Clear selected bit of A

Clear all but the selected bit of A

Set selected bit of A

Complement selected bit of A

Bitwise Operators: Shift Operator

```
B = A << 3;                A   1 0 1 0 1 1 0 1
(Left shift 3 bits)        B   0 1 1 0 1 0 0 0

B = A >> 2;                A   1 0 1 1 0 1 0 1
(Right shift 2 bits)       B   0 0 1 0 1 1 0 1

B = '1';                   B = 0 0 1 1 0 0 0 1 (ASCII 0x31)
C = '5';                   C = 0 0 1 1 0 1 0 1 (ASCII 0x35)
D = (B << 4) | (C & 0x0F);
       (B << 4)            = 0 0 0 1 0 0 0 0
     (C & 0x0F)            = 0 0 0 0 0 1 0 1
             D             = 0 0 0 1 0 1 0 1 (Packed BCD 0x15)
```

Lab Task: LED Follower Logic

Generate a code to Print a number in binary and decimal format, then apply left shift operator 3 times then print number in binary and decimal

Bitwise Operators Example

```
int main()
{

    int a = 25, b = 5;

    // using operators and printing results
    printf("a & b: %d\n", a & b);
    printf("a | b: %d\n", a | b);
    printf("a ^ b: %d\n", a ^ b);
    printf("~a: %d\n", ~a);
    printf("a >> b: %d\n", a >> b);
    printf("a << b: %d\n", a << b);

    return 0;
}
```

Output

```
a & b: 1
a | b: 29
a ^ b: 28
~a: -26
a >> b: 0
a << b: 800
```

Assignment Operators in C

S. No.	Symbol	Operator	Description	Syntax
1	=	Simple Assignment	Assign the value of the right operand to the left operand.	a = b
2	+=	Plus and assign	Add the right operand and left operand and assign this value to the left operand.	a += b
3	-=	Minus and assign	Subtract the right operand and left operand and assign this value to the left operand.	a -= b
4	*=	Multiply and assign	Multiply the right operand and left operand and assign this value to the left operand.	a *= b
5	/=	Divide and assign	Divide the left operand with the right operand and assign this value to the left operand.	a /= b
6	%=	Modulus and assign	Assign the remainder in the division of left operand with the right operand to the left operand.	a %= b
7	&=	AND and assign	Performs bitwise AND and assigns this value to the left operand.	a &= b
8	|=	OR and assign	Performs bitwise OR and assigns this value to the left operand.	a |= b
9	^=	XOR and assign	Performs bitwise XOR and assigns this value to the left operand.	a ^= b
10	>>=	Rightshift and assign	Performs bitwise Rightshift and assign this value to the left operand.	a >>= b
		Leftshift and	Performs bitwise Leftshift and assign this value to the left	

Assignment Operators Example

```
int main()
{
    int a = 25, b = 5;

    // using operators and printing results
    printf("a = b: %d\n", a = b);
    printf("a += b: %d\n", a += b);
    printf("a -= b: %d\n", a -= b);
    printf("a *= b: %d\n", a *= b);
    printf("a /= b: %d\n", a /= b);
    printf("a %%= b: %d\n", a %= b);
    printf("a &= b: %d\n", a &= b);
    printf("a |= b: %d\n", a |= b);
    printf("a >>= b: %d\n", a >>= b);
    printf("a <<= b: %d\n", a <<= b);
return 0;
}
```

Output

```
a = b: 5
a += b: 10
a -= b: 5
a *= b: 25
a /= b: 5
a %= b: 0
a &= b: 0
a |= b: 5
a >>= b: 0
```

Testing and Executing the Code

RIPES

https://ripes.me/

https://github.com/mortbopet/Ripes/releases/download/v2.2.6/Ripes-v2.2.6-win-x86_64.zip

Next:

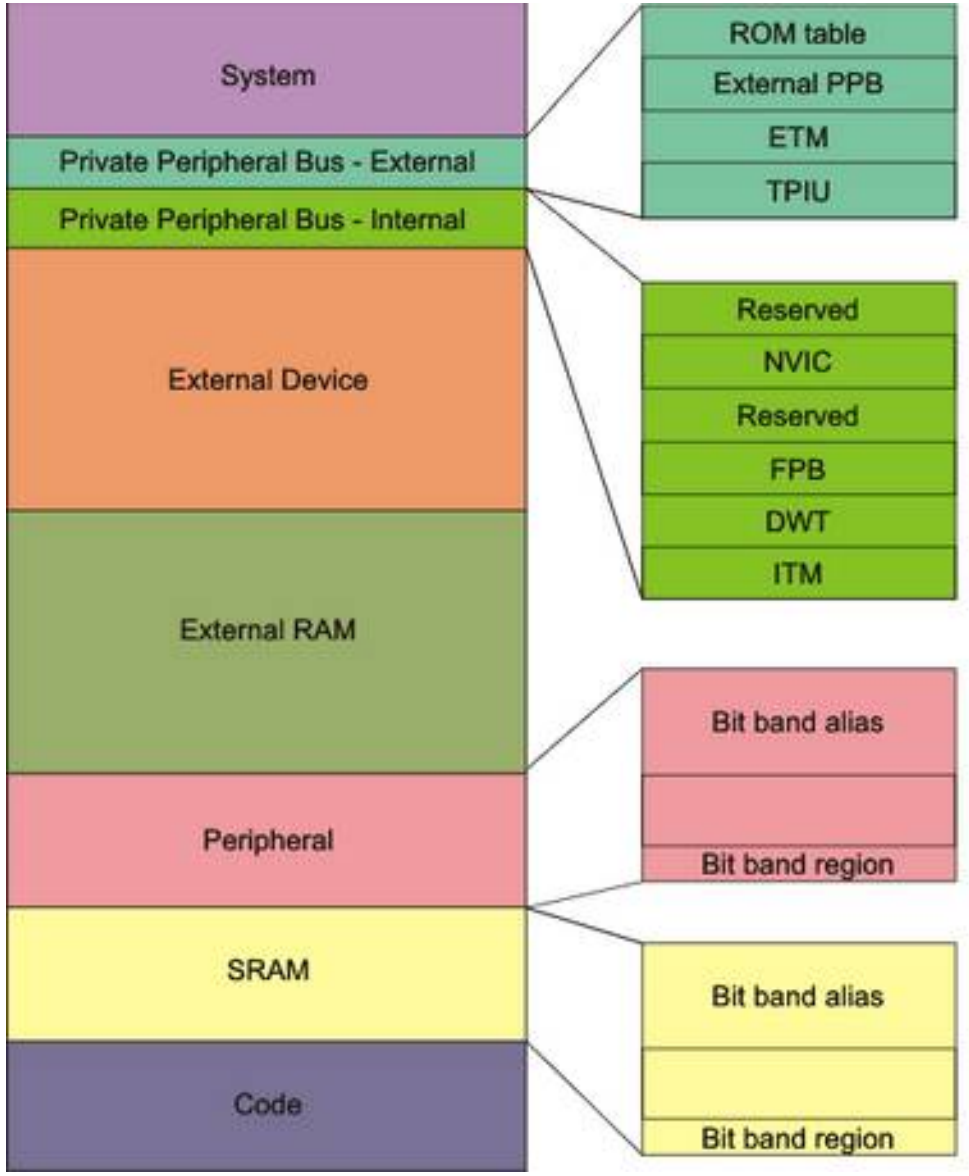
RISCV Micro Controller

RISCV Simulator and Emulators

RISCV Single Board Computer

Microcontroller Memory Architecture

- Flash Memory
  - Stores the program code (firmware) and constant data.
- SRAM (Static RAM)
  - **Program Memory:**
  - Temporary storage used during execution, holding variables, the stack, and the heap.
  - **Data Memory:**
  - Used for dynamic data storage, including global variables, static variables, and temporary data.
- DRAM (Dynamic RAM)
  - Typically used in more complex microcontrollers or embedded systems where larger data storage is needed.
  - Stores variable data, buffers, and other dynamic data structures.



Memory Architecture Overview:

- Harvard Architecture: Many microcontrollers use this architecture, where program and data memories are separate, allowing simultaneous access to both.
- Von Neumann Architecture: Some microcontrollers use this unified memory architecture, where program and data are stored in the same memory space, but this can introduce bottlenecks since program and data fetches compete for the same bus.

Pointers in Embedded C

- Pointers in C are variables that store the memory address of another variable. In embedded C, pointers are particularly important because they provide a way to directly access and manipulate hardware registers, memory locations, and peripheral devices.
- Direct Hardware Access: Pointers allow you to interact with specific memory-mapped registers and peripherals.
- Memory Management: They are used for dynamic memory allocation, accessing arrays, and structures efficiently.
- Function Arguments: Pointers can be passed to functions to modify variables or return multiple values.

```
int a = 10;
int *ptr = &a;  // ptr is a pointer that stores the address of variable 'a'
// Access the value of 'a' using the pointer
printf("Value of a: %d\n", *ptr);  // Outputs 10
// Modify the value of 'a' using the pointer
*ptr = 20;
printf("New value of a: %d\n", a);  // Outputs 20


int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;  // Points to the first element of the array
// Accessing array elements using the pointer
for(int i = 0; i < 5; i++) {
    printf("arr[%d] = %d\n", i, *(ptr + i));
}
```

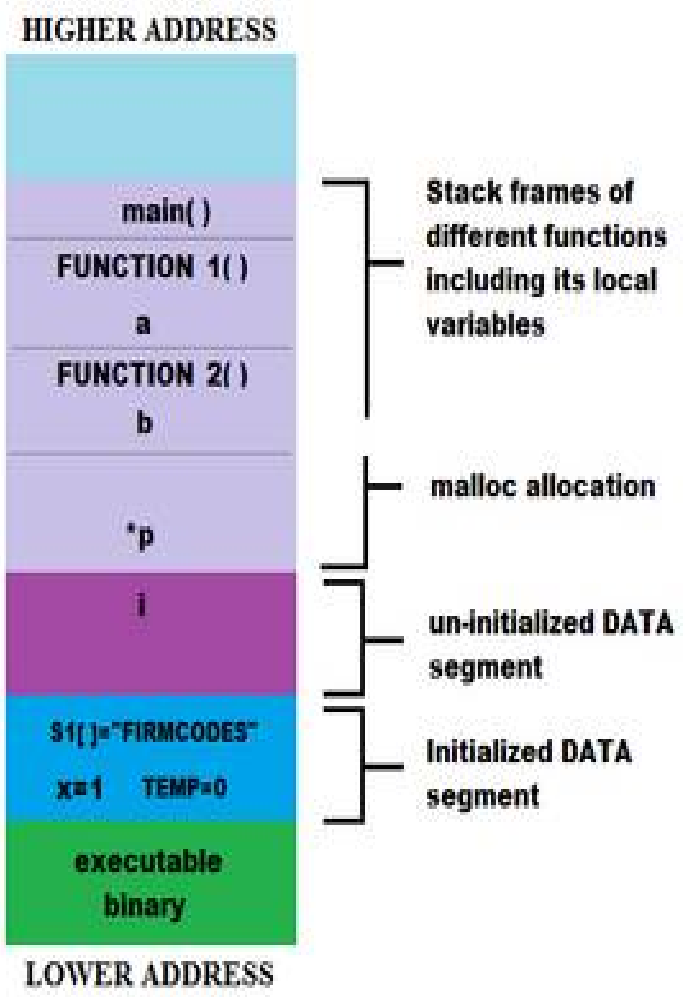
```
void myFunction(int x) {
    printf("Value is: %d\n", x);
}


int main() {
    void (*funcPtr)(int) = myFunction;  // Declare a function pointer
    funcPtr(10);  // Call the function using the pointer
    return 0;
}


#define GPIO_PORTA_BASE 0x40004000
volatile unsigned int *gpioData = (volatile unsigned int *)(GPIO_PORTA_BASE + 0x3FC);
*gpioData = 0xFF;  // Set all bits of PORTA to 1
```

Program Memory

- Stack and Heap: Typically located in SRAM.
- Global/Static Variables: Located in SRAM.
- Constant Data: Located in Flash.




```c
#include<stdio.h>
#include<malloc.h>

void FUNCTION_1();
void FUNCTION_2();

char S1[]="FIRMCODES";  //initialized read-write area of DATA segment
int i;                  //uninitialized DATA segment
const int x=1;          //initialized read-only area of DATA segment

int main()
{
    static int TEMP=0;  //uninitialized DATA segment

    char *p=(char*)malloc(sizeof(char)); //Heap segment

    FUNCTION_1();       //FUNCTION_1 stack frame

    return 0;
}

void FUNCTION_1()
{
    int a;          //initialized in stack frame of FUNCTION_1

    FUNCTION_2();   //FUNCTION_2 stack frame
}

void FUNCTION_2()
{
    int b;          //initialized in stack frame of FUNCTION_2
}
```

Generate Instruction and Data Memory

- riscv32-unknown-elf-gcc -march=rv32i -S -o riscv.s ./code.c
- riscv32-unknown-elf-as -march=rv32i -S -o riscv.o ./riscv.s
- riscv32-unknown-elf-as -march=rv32i -o riscv.o ./riscv.s
- riscv32-unknown-elf-ld -o riscv ./riscv.o
- riscv32-unknown-elf-objcopy -O binary --only-section=.text riscv instr.mem
- riscv32-unknown-elf-objcopy -O binary --only-section=.data riscv data.mem
- riscv32-unknown-elf-objdump -D -b binary -m riscv:rv32i instr.mem

Managing Local Memory (SRAM)

```
#include <stdio.h>
void printArray() {
    // Local array stored in stack memory (SRAM)
    int a[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
    for(int i = 0; i < 10; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int main() {
    printArray();
    return 0;
}
```

Main Memory

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    // Allocate memory for an array of 10 integers in the heap
    int *a = (int *)malloc(10 * sizeof(int));

    // Check if memory allocation was successful
    if (a == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Initialize the array with values
    for(int i = 0; i < 10; i++) {
        a[i] = i + 1;
    }

    // Print the array values
    for(int i = 0; i < 10; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    // Free the allocated memory
    free(a);
    return 0;
}
```

Debugging and Analysis Techniques

Embedded systems are specialized computer systems designed for specific purposes. They

- **Control**
- **Monitor**
- **Assist**

in the operation of equipment, machinery, or a larger system. These systems are present in various industries, such as automotive, consumer electronics, aerospace, and medical devices.

- **Debugging** is a crucial aspect of embedded systems development. As these systems are responsible for critical operations, any error or malfunction can have severe consequences.
- Debugging helps identify and fix errors, ensuring the system functions as expected.
- Moreover, it contributes to the overall quality, reliability, and performance of the embedded system

Debugging and Analysis Techniques

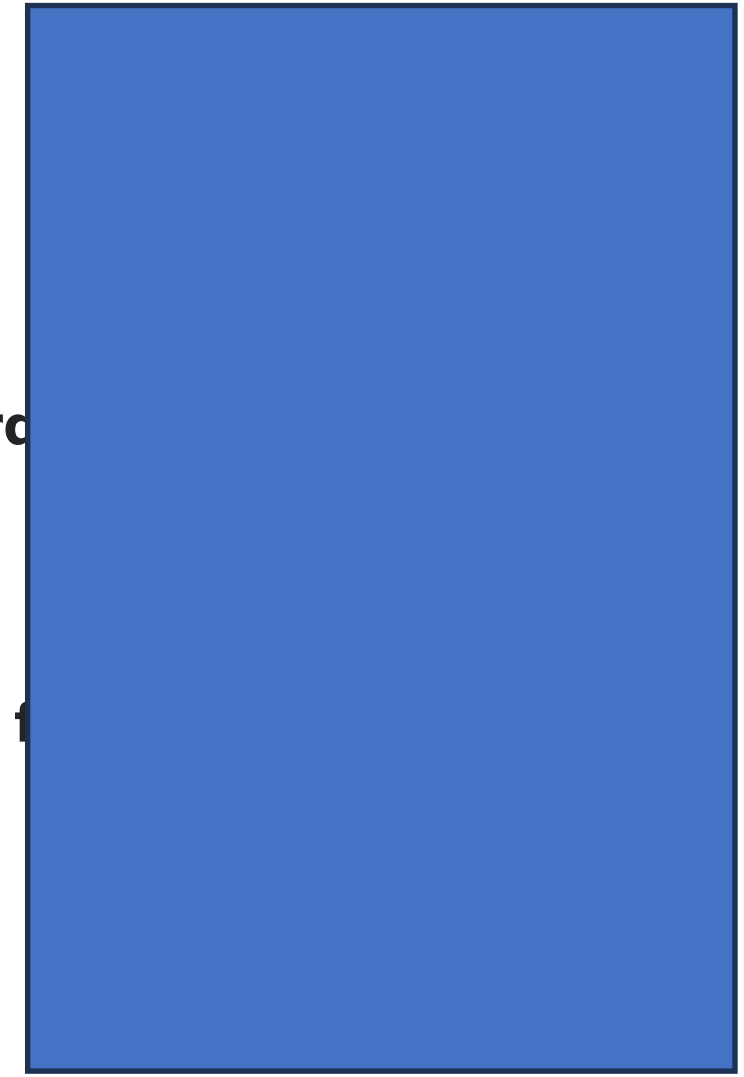
Understanding Debugging

Debugging is the process of

- **Identifying,**
- **Analyzing,**
- **Resolving issues within a software or hard**

It involves

- **finding the root cause of problems,**
- **understanding their impact, and**
- **implementing solutions to ensure proper f**



Debugging and Analysis Techniques

Goals and Objectives of Debugging

- The primary goal of debugging is to ensure that a system functions as intended. This involves identifying and fixing errors, optimizing performance, and enhancing stability. Debugging aims to:
- Locate and resolve software bugs and hardware issues.
- Improve system performance and efficiency.
- Enhance the user experience by fixing usability issues.
- Ensure compliance with industry standards and best practices.
- Maintain system stability and reliability.

Debugging and Analysis Techniques

Importance of Debugging in Embedded Systems

Debugging plays a vital role in embedded systems development. Due to the specialized nature of these systems, errors can lead to severe consequences, such as equipment malfunction or even safety hazards.

- Debugging helps ensure the proper functioning of embedded systems by:
- Eliminating errors that can compromise system performance and safety.
- Optimizing resource usage, which is crucial in systems with limited resources.
- Enhancing system stability and reliability.
- Improving overall system quality and user satisfaction.

By thoroughly understanding and mastering debugging techniques, embedded systems developers can create high-quality, reliable, and efficient systems that meet the demands of various industries.

Debugging and Analysis Techniques

Common Debugging Challenges in Embedded Systems

Limited Resources and Processing Power

Embedded systems often operate under strict resource constraints, such as limited memory, processing power, and power consumption.

Debugging in such environments can be challenging, as developers must balance the need for debugging tools and techniques with the available resources. This may require creative approaches and careful planning to ensure effective debugging without impacting system performance.

Real-Time Constraints

Many embedded systems operate in real-time, meaning they must respond to events and inputs within strict time constraints. Debugging real-time systems can be challenging, as developers must not only identify and resolve issues but also ensure that the system continues to meet its real-time requirements.

This often involves analyzing and optimizing the timing and synchronization aspects of the system.

Debugging and Analysis Techniques

Complex Hardware and Software Interactions

Embedded systems typically involve complex interactions between hardware and software components. Debugging these systems requires a deep understanding of both domains, as well as the ability to analyze and trace issues across the hardware-software boundary. This can be challenging, particularly when dealing with proprietary or custom hardware.

Concurrency Issues

Many embedded systems rely on concurrent processing to achieve their goals, whether through multi-threading, multi-processing, or other parallel processing techniques.

Debugging concurrent systems introduces additional complexity, as developers must identify and resolve issues related to synchronization, race conditions, and other concurrency-related challenges.

Debugging and Analysis Techniques

Debugging Techniques for Embedded Systems

1. Static Code Analysis

Static code analysis involves examining the source code of a system without executing it. It helps identify potential issues such as syntax errors, memory leaks, and coding standard violations. The benefits of static code analysis include early detection of errors, improved code quality, and reduced development time.

Some popular static code analysis tools for embedded systems include:

PC-Lint: A widely used tool for analyzing C and C++ code

Cppcheck: An open-source tool for detecting bugs in C and C++ code

CodeSonar: A commercial tool for analyzing C, C++, Java, and Ada code

MISRA-C: A set of coding standards for embedded systems development in C

Debugging and Analysis Techniques

2. Dynamic Analysis

Dynamic analysis involves monitoring the behavior of a system during runtime. It helps identify issues such as memory corruption, race conditions, and performance bottlenecks. The benefits of dynamic analysis include real-time error detection, improved system performance, and increased reliability.

Some popular dynamic analysis tools for embedded systems include:

Valgrind: An open-source tool for detecting memory management issues

GDB: The GNU Debugger, a widely used debugger for various programming languages

JTAG: A hardware debugging interface used for on-chip debugging and programming

Tracealyzer: A commercial tool for visualizing and analyzing real-time system behavior

Debugging and Analysis Techniques

4. In-Circuit Debugging

In-circuit debugging involves connecting a debugger directly to a running embedded system, allowing developers to monitor and control its execution. Benefits include real-time debugging capabilities, improved system visibility, and the ability to debug hardware-related issues.

Some popular in-circuit debugging tools for embedded systems include:

JTAG: A widely used hardware debugging interface, as mentioned in the Dynamic Analysis section

Segger J-Link: A popular JTAG/SWD debugger for ARM-based systems

P&E Micro: A provider of in-circuit debugging solutions for various microcontroller platforms

Atmel-ICE: An in-circuit debugger and programmer for Atmel microcontrollers

Debugging and Analysis Techniques

5. Hardware Debugging

Hardware debugging involves diagnosing and fixing issues related to the physical components of an embedded system, such as circuitry, sensors, and actuators. Benefits include improved system reliability, reduced development time, and the ability to identify and resolve hardware-specific issues.

Some popular hardware debugging tools for embedded systems include:

- Oscilloscopes: Essential tools for analyzing and troubleshooting electrical signals
- Logic Analyzers: Devices used for monitoring and analyzing digital signals
- Protocol Analyzers: Tools for capturing and analyzing communication data between system components
- Power Analyzers: Instruments for measuring and analyzing power consumption in embedded systems

Performance Profiling

What do we means by Performance ?

An analogy with passenger airplanes shows how subtle the question of performance can be. For example

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers × m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

The capacity, range, and speed for a number of commercial airplanes.

Which of the planes in this table had the best performance ?

We have possible performance criterias' as

Performance Criterias	Plane
Highest cruising speed	Concorde
highest cruising speed, taking a single passenger from one point to another in the least time.	Being 747

Performance Profiling

Computer Performance

Similarly, we can define computer performance in several distinct ways.

Your personal performance Criteria:

If you were running a program on two different desktop computers, you'd say that the faster one is the desktop computer that gets the job done first.

As an individual computer user, you are interested in reducing response time—the time between the start and completion of a task—also referred to as **execution time**.

Data Centre's Criteria of performance

The faster computer was the one. that completed the most jobs during a day.

Datacenter managers often care about increasing **throughput** or **bandwidth**—the total amount of work done in a given time.

Performance Profiling

Response time /Execution time. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

Maximizing the Performance of Computer

To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

That is, the execution time on Y is longer than that on X, if X is faster than Y.

Performance Profiling

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase

“X is n times faster than Y”—or equivalently “X is n times as fast as Y”—to mean

If X is n times as fast as Y, then the execution time on Y is n times as long as it is on X:

Relative Performance

Example If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

Answer We know that A is n times as fast as B if

Thus the performance ratio is and A is therefore 1.5 times as fast as B.

Performance Profiling

Measuring Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest.

*Program **execution time** is measured in **seconds per program**.*

Also wall clock time, response time, or elapsed time.

These terms mean the total time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead—everything.