

Assembly Language Tutorial

Table of Contents

Overview.....	2
Language Processing System:	6
Buses.....	7
Architecture of x64 Processor	11
Systemcalls	25
Instruction in Assembly	31
Let's write Assembly Program	46
Take input from user.....	51
Basic Maths in x64 Assembly Lanugage and output show in Debugger:.....	58
Conditional Branching(If else program)	68
Loop in Assembly	74
Final Project Calculator:	80
References:	85

Overview

Assembly language is a low-level programming language that provides a way to write programs in terms of the architecture-specific instructions of a computer's processor. It serves as a bridge between machine language (binary code) and high-level programming languages like Python, Java, or C. Each assembly language is specific to a particular CPU architecture, such as x86, ARM, or MIPS.

Assembly language uses mnemonics, which are human-readable representations of machine instructions, along with additional features such as labels and directives to make programming easier. For example:

```
MOV AX, 5    ; Move the value 5 into the AX register
ADD AX, 2    ; Add 2 to the value in the AX register
```

This code corresponds to binary machine instructions but is much easier for humans to read and write.

Importance of Assembly Language in Programming

1. Hardware Control and Optimization:

- Assembly allows programmers to control hardware directly, making it ideal for tasks requiring precision and efficiency, such as device drivers and embedded systems programming.
- It enables fine-tuned optimization for performance-critical applications.

2. Understanding System Architecture:

- Learning assembly helps programmers understand how a CPU works, including registers, memory management, and instruction execution.
- It provides insight into how high-level languages interact with hardware, aiding in debugging and optimization.

3. Performance and Resource Management:

- Assembly language allows for precise control over system resources, such as CPU cycles and memory, which is crucial for applications with strict performance requirements (e.g., real-time systems).

4. Debugging and Reverse Engineering:

- Knowledge of assembly is essential for debugging low-level issues and analyzing compiled code during reverse engineering or security analysis.

5. Embedded Systems:

- Assembly is commonly used in embedded systems where hardware constraints demand highly efficient code.

6. Legacy Systems Maintenance:

- Some legacy systems and applications are written in assembly, and maintaining them requires proficiency in the language.

7. Bootloaders and Operating Systems:

- Critical parts of operating systems and bootloaders are often written in assembly to interact directly with hardware during the initial stages of system startup.

Why it is important for Cybersecurity Purpose.

Assembly language plays a crucial role in **cybersecurity** because of its ability to interact directly with hardware and system resources. Here's how it is used in various cybersecurity domains:

1. Malware Analysis and Reverse Engineering

- **Purpose:** Understanding how malware operates and uncovering its behavior.
 - **How Assembly is Used:**
 - Disassembled malware code is often in assembly language.
 - Security analysts examine the assembly code to identify malicious behavior, such as payloads, exploits, and data exfiltration methods.
 - **Example:** Analyzing a Trojan's assembly code to understand its encryption mechanism or how it propagates.
-

2. Exploitation Development

- **Purpose:** Crafting or analyzing exploits to identify vulnerabilities.
- **How Assembly is Used:**
 - Exploits target low-level vulnerabilities like buffer overflows or return-oriented programming (ROP) attacks, which require knowledge of assembly and CPU architecture.

- Security researchers write shellcode in assembly to exploit system weaknesses.
 - **Example:** Developing a custom exploit to test a system's protection against stack overflow vulnerabilities.
-

3. Incident Response and Forensics

- **Purpose:** Investigating and mitigating cyberattacks.
 - **How Assembly is Used:**
 - Analyzing crash dumps, memory snapshots, or system logs often involves interpreting assembly instructions to determine the origin and impact of an attack.
 - Detecting and understanding rootkits or low-level malware hidden in the operating system.
-

4. Secure Software Development

- **Purpose:** Writing secure code and identifying vulnerabilities in software.
 - **How Assembly is Used:**
 - Identifying low-level security issues, such as buffer overflows or improper use of system calls.
 - Writing secure assembly code for sensitive operations like cryptography or authentication routines.
 - **Example:** Developing secure bootloaders or cryptographic libraries in assembly.
-

5. Operating System and Kernel Security

- **Purpose:** Hardening the operating system and detecting vulnerabilities.
 - **How Assembly is Used:**
 - Operating systems and kernels rely on assembly for tasks like interrupt handling and memory management.
 - Security professionals analyze kernel-level assembly to uncover zero-day vulnerabilities or ensure secure handling of system resources.
-

6. Binary Exploitation and Patching

- **Purpose:** Modifying binaries to fix vulnerabilities or alter behavior.
 - **How Assembly is Used:**
 - Analysts use assembly to understand compiled programs and apply patches to correct vulnerabilities without access to source code.
 - **Example:** Modifying a binary to disable a vulnerable function or bypass a malware's anti-analysis technique.
-

7. Firmware Analysis

- **Purpose:** Ensuring the security of hardware-level code.
 - **How Assembly is Used:**
 - Firmware is often written in assembly or includes assembly code. Analyzing and validating firmware security requires understanding its assembly instructions.
 - **Example:** Identifying backdoors in IoT device firmware.
-

8. Cryptanalysis

- **Purpose:** Breaking or analyzing encryption mechanisms.
 - **How Assembly is Used:**
 - Assembly is used to analyze cryptographic implementations at the hardware level.
 - It helps identify weaknesses in algorithms or their implementation, such as side-channel attacks.
 - **Example:** Analyzing the assembly code of an encryption function to detect timing leaks.
-

9. Learning from Vulnerabilities

- **Purpose:** Understanding real-world attacks.
- **How Assembly is Used:**
 - Many real-world exploits, such as privilege escalation and remote code execution, involve assembly.
 -

Tools Used Alongside Assembly in Cybersecurity

1. Disassemblers:

- Tools like IDA Pro, Ghidra, and Radare2 translate machine code into assembly for analysis.

2. Debuggers:

- Tools like GDB and WinDbg allow step-by-step execution of assembly instructions.

3. Emulators:

- Tools like QEMU and Bochs simulate hardware environments for testing and analysis.

4. Hex Editors:

- Tools like HxD help analyze and manipulate binary files.

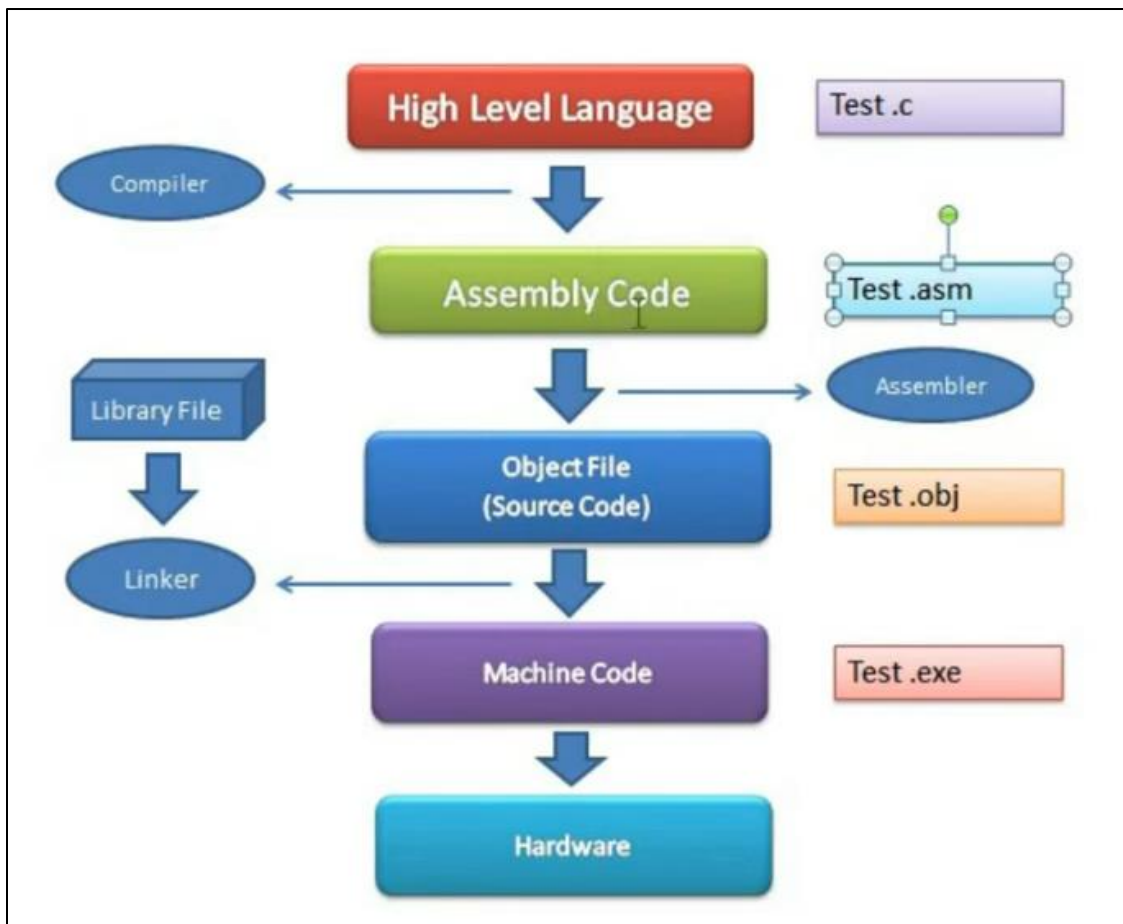
Why Assembly is Vital for Cybersecurity Professionals

- Cybersecurity operates on both offensive and defensive fronts, requiring a deep understanding of how systems and software interact at the most fundamental level.
- Knowledge of assembly language empowers professionals to analyze and secure systems effectively, making it an essential skill in the field.

#####

Language Processing System:

A **language processing system** refers to the set of tools and components that enable a computer to understand, interpret, generate, and manipulate human languages (natural languages). It is a key component in fields like natural language processing (NLP), artificial intelligence (AI), and computational linguistics.



#####

Buses

A **bus** in computer architecture is a communication system that transfers data between components of a computer or between computers. It is essentially a pathway or set of wires that allows various hardware parts to communicate with one another.

Key Features of Buses:

1. **Data Transmission:** Facilitates the transfer of data between the CPU, memory, and peripheral devices.
2. **Communication Medium:** Acts as a shared medium for multiple components to send and receive data.
3. **Types of Signals:** Transmits data, control signals, and memory addresses.

Transmission Mode

Mode	Direction	Simultaneous Communication	Examples
Simplex	One-way only	No	TV broadcast, keyboard input
Half-Duplex	Both ways (one at a time)	No	Walkie-talkies
Full-Duplex	Both ways	Yes	Telephones, Ethernet

Components of a Bus:

A bus generally consists of three main parts:

1. Data Bus:

- Transfers actual data between components (e.g., CPU to memory).
- Example: Sending a number to be added in the CPU.

2. Address Bus:

- Carries memory addresses to indicate where data is to be read or written.
- Example: Telling memory which location to fetch data from.

3. Control Bus:

- Sends control signals to coordinate operations (e.g., read/write signals).
- Example: Notifying memory that the CPU wants to write data.

Types of Buses:

1. Internal Bus:

- Connects internal computer components like the CPU, memory, and motherboard.
- Examples: System bus, memory bus.

2. External Bus:

- Connects external devices and peripherals to the computer.
- Examples: USB, PCI, SATA.

Characteristics of Buses:

1. **Width:** Determines how much data can be transmitted at once (e.g., 32-bit, 64-bit).
2. **Speed:** Measured in MHz or GHz, representing how fast data can be transferred.
3. **Topology:** The arrangement of the bus (parallel or serial).

Importance of Buses:

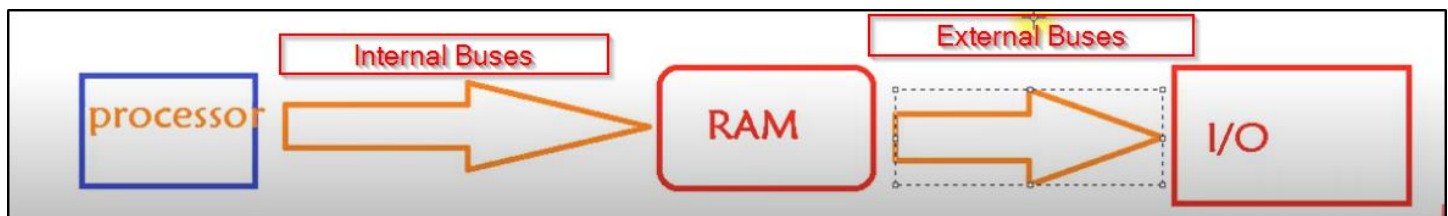
- Ensure seamless communication within the computer.
- Allow the integration of new hardware components.
- Optimize the overall performance and speed of a system.

In summary, buses are the backbone of data communication in computer systems, ensuring that all components can effectively interact with one another.

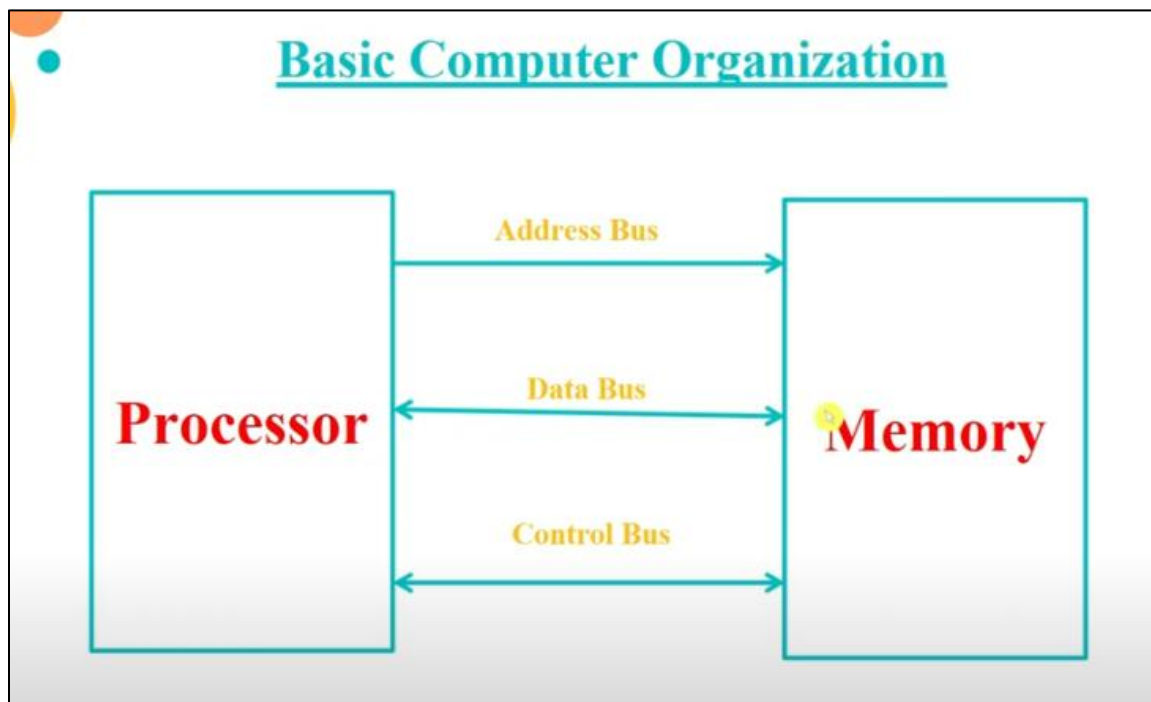
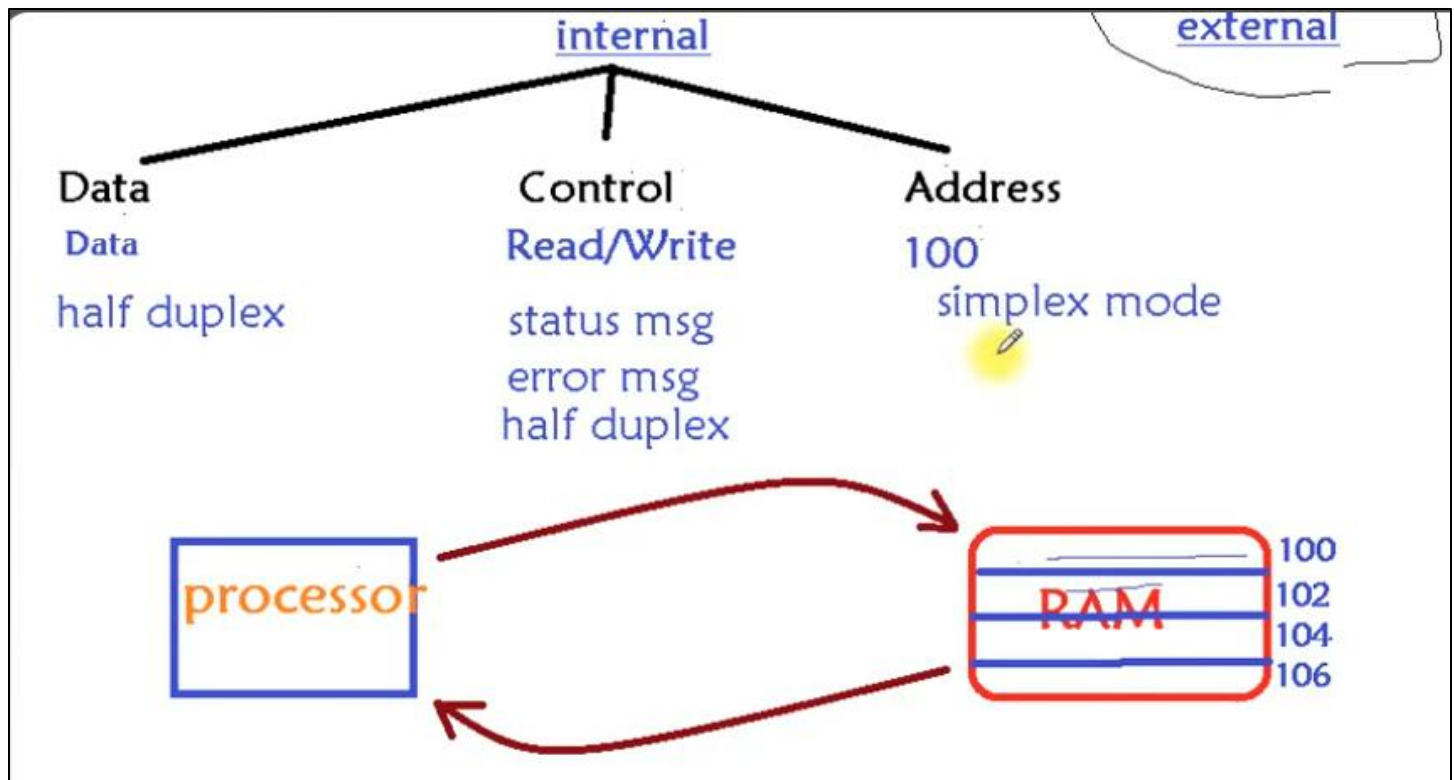
There are two types of Buses.

Key Differences:

- **Internal Buses:** Operate inside the system to connect components within the CPU, motherboard, and memory.
- **External Buses:** Provide connections to external devices or peripherals for input/output and additional functionality.



Category	Bus Type	Purpose
Internal Buses	Data Bus	Transfers data between internal components (e.g., CPU, RAM).
	Address Bus	Carries memory addresses from the CPU to RAM or other components.
	Control Bus	Sends control signals (e.g., read/write) between CPU and internal devices.
	Backplane Bus	Connects components on the motherboard, such as PCI or system buses.
External Buses	USB (Universal Serial Bus)	Connects peripherals (e.g., keyboards, mice, storage devices).
	PCI/PCIe (Peripheral Component Interconnect)	Connects external hardware like GPUs and network cards.
	SATA (Serial ATA)	Connects storage devices like SSDs and HDDs.
	Ethernet Bus	Handles network communication.
	Thunderbolt	High-speed external connection for storage, displays, etc.
	I²C (Inter-Integrated Circuit)	Connects low-speed peripherals (e.g., sensors, small devices).
	SPI (Serial Peripheral Interface)	Connects microcontrollers to small devices like displays and sensors.



#####

Architecture of x64 Processor

Registers:

A **register** is a small, high-speed storage location directly within a processor (CPU). Registers are used to temporarily hold data, instructions, and addresses that the CPU is currently processing or accessing. They are the fastest type of memory in a computer system because they are located within the CPU chip.

Key Characteristics of Registers:

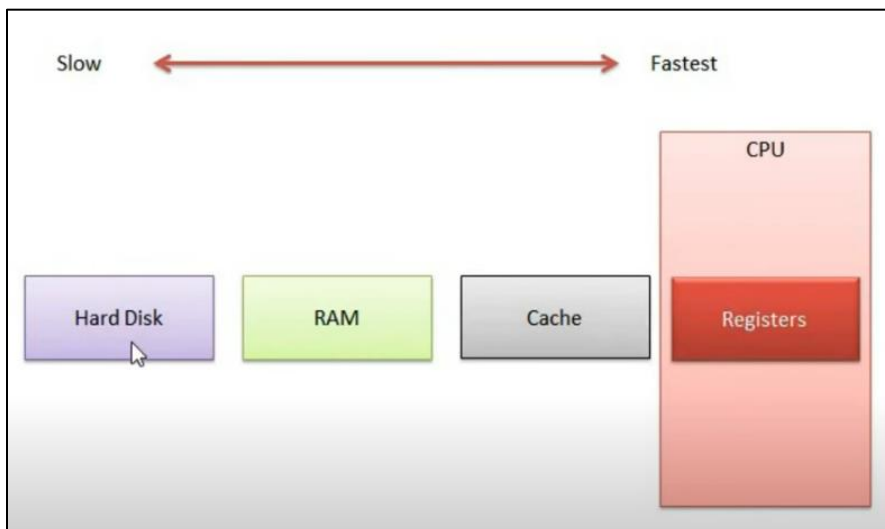
1. **High-Speed:** Faster than RAM or cache as they are part of the CPU.
2. **Temporary Storage:** Used for immediate operations and not for long-term data retention.
3. **Fixed Size:** Typically 8, 16, 32, or 64 bits, depending on the CPU architecture.
4. **Specialized Purpose:** Some registers are general-purpose, while others serve specific functions (e.g., instruction pointer, stack pointer).

Role in CPU Operation:

Registers play a critical role in the execution of instructions:

1. **Fetch:** The CPU fetches an instruction from memory, using registers to store the instruction temporarily.
2. **Decode:** Registers hold operands for the instruction during decoding.
3. **Execute:** The CPU performs the operation, often using registers to store intermediate or final results.

Registers enable efficient processing by providing the CPU with quick access to the data it needs.



Types of Register:--

General Purpose

uses for more than one purposes..

1. **AX** Accumulator
2. **BX** base **AX**
3. **CX** count **EAX**
4. **DX** destination **RAX**

x = size

Special Purpose

used for only one purpose for which they are made

1. **IP(PC)** = instruction pointer/program counter
2. **IR** = Instruction register
3. **BP** = base pointer register
4. **SP** = stack pointer register
5. **SI** = source index register

The x64 processor architecture has several types of registers, each with specific purposes. Here's a breakdown:

1. General-Purpose Registers (GPRs)

- **Purpose:** Used for arithmetic, logical, data movement, and address calculations.
- **Registers:**
 - **RAX:** Accumulator for arithmetic operations.
 - **RBX:** Base register for addressing memory.
 - **RCX:** Counter for loops and string operations.
 - **RDX:** Data register for I/O and multiplications/divisions.
 - **RSI:** Source index for string operations.
 - **RDI:** Destination index for string operations.
 - **RBP:** Base pointer for stack frames.
 - **RSP:** Stack pointer for the top of the stack.
 - **R8 to R15:** Additional general-purpose registers introduced in x64.
- **Sizes:**
 - 64-bit (e.g., RAX) [R stands for Rich]
 - 32-bit (e.g., EAX) [E stands for Extended]
 - 16-bit (e.g., AX)
 - 8-bit (e.g., AL and AH).

2. Segment Registers

- **Purpose:** Hold segment selectors for memory segmentation.
- **Registers:**
 - **CS:** Code Segment.
 - **DS:** Data Segment.
 - **ES:** Extra Segment.
 - **FS:** Additional segment often used for thread-local storage.
 - **GS:** Another additional segment for specialized data.
 - **SS:** Stack Segment.
- **Note:** Segmentation is less significant in x64 due to flat memory models.

3. Control Registers

- **Purpose:** Control the processor's operation.
- **Registers:**
 - **CR0:** Enables or disables features (e.g., protected mode).
 - **CR2:** Holds the faulting address during a page fault.
 - **CR3:** Contains the base address of the page directory (used in paging).
 - **CR4:** Controls extended features (e.g., SSE, PAE).
 - **CR8:** Task priority register (controls IRQ priority).

4. Instruction Pointer

- **Purpose:** Points to the next instruction to execute.
- **Register:**
 - **RIP:** 64-bit instruction pointer.

5. Flags Register (RFLAGS)

- **Purpose:** Holds status flags and control flags.
- **Key Flags:**
 - **CF:** Carry Flag.
 - **ZF:** Zero Flag.
 - **SF:** Sign Flag.
 - **OF:** Overflow Flag.

- **IF**: Interrupt Enable Flag.
- **DF**: Direction Flag.

6. Floating-Point and SIMD Registers

- **Purpose**: Handle floating-point operations, SIMD operations, and cryptographic operations.
- **Registers**:
 - **XMM0-XMM15**: 128-bit SIMD registers.
 - **YMM0-YMM15**: 256-bit SIMD registers (AVX).
 - **ZMM0-ZMM31**: 512-bit SIMD registers (AVX-512).
 - **ST(0)-ST(7)**: Floating-point stack registers for legacy FPU operations.

7. Debug Registers

- **Purpose**: Debugging and setting hardware breakpoints.
- **Registers**:
 - **DR0-DR3**: Debug address registers.
 - **DR6**: Debug status register.
 - **DR7**: Debug control register.

8. Model-Specific Registers (MSRs)

- **Purpose**: Configuration and monitoring of processor-specific features.
- **Access**: Using instructions like RDMSR and WRMSR.

9. Task State Segment (TSS) Registers

- **Purpose**: Manage task-related information in multitasking.

These registers collectively allow the x64 processor to perform complex computations, control program flow, manage memory, and interface with hardware effectively.

Types of Register in x64 Processor:

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl

rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

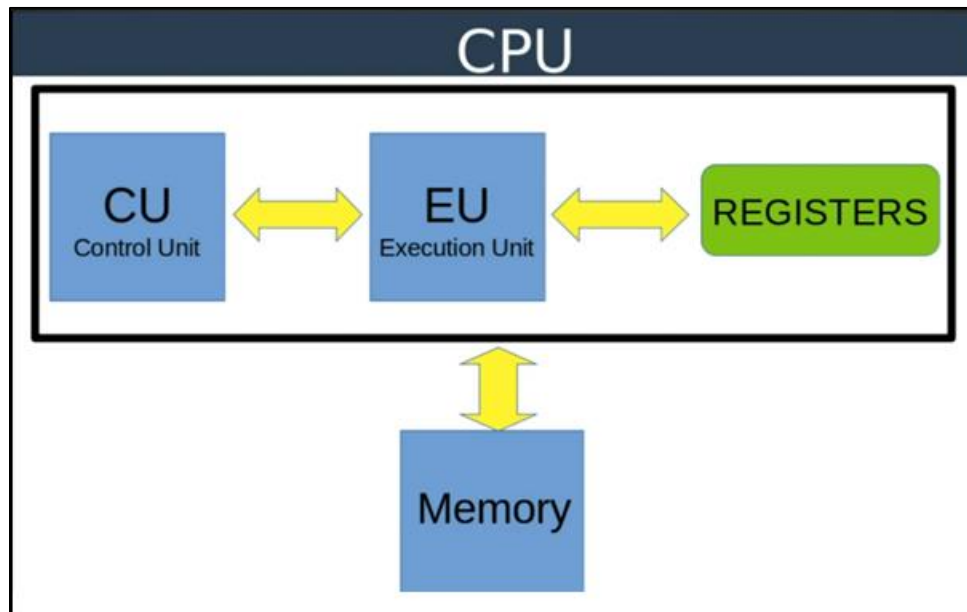
Registers names, Full name, Description.

Register Name	Full Name	Description
RAX	Accumulator Register	Used for arithmetic and logical operations, function return values, and data storage.
RBX	Base Register	Used as a pointer to memory locations; often for addressing data in memory.
RCX	Count Register	Used for loop counters and string operations.
RDX	Data Register	Stores data for I/O operations and extended arithmetic (e.g., division and multiplication).
RSI	Source Index Register	Holds the source address for string operations and memory data transfers.
RDI	Destination Index Register	Holds the destination address for string operations and memory data transfers.
RBP	Base Pointer Register	Points to the base of the current stack frame; used for stack-based addressing.
RSP	Stack Pointer Register	Points to the top of the stack; used for stack operations (e.g., push, pop).
RIP	Instruction Pointer	Points to the next instruction to be executed by the CPU.
RFLAGS	Flags Register	Holds condition flags (e.g., Zero, Carry, Overflow) and control flags for the CPU.

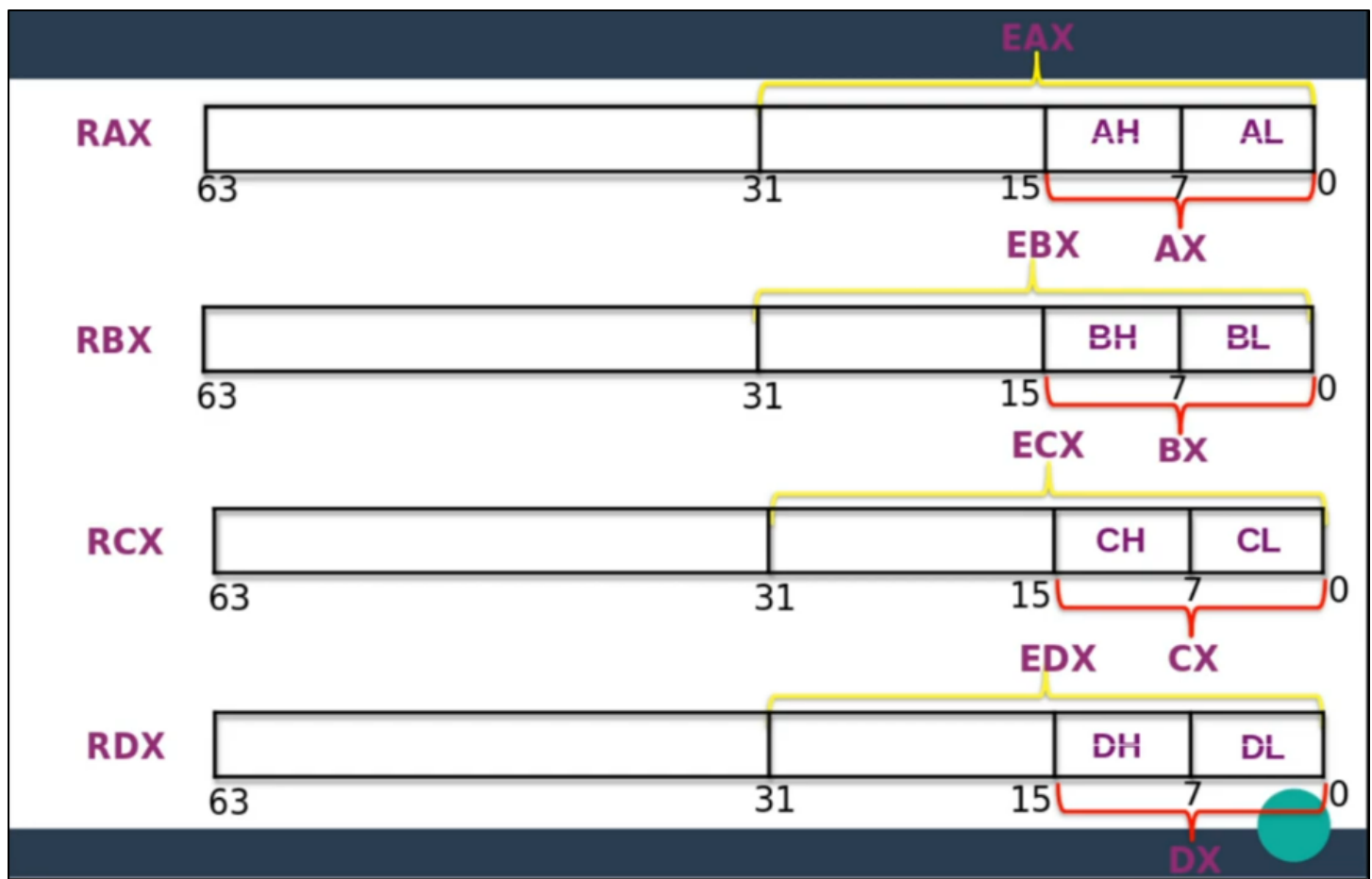
R8 to R15	General-Purpose Registers	Additional general-purpose registers available in the x64 architecture.
CS	Code Segment Register	Holds the segment address for code.
DS	Data Segment Register	Holds the segment address for data.
ES	Extra Segment Register	Holds an additional segment address.
FS	FS Segment Register	Used for thread-local storage or specialized memory areas.
GS	GS Segment Register	Similar to FS, often used for thread-local data or special-purpose memory areas.
SS	Stack Segment Register	Holds the segment address for the stack.
XMM0–XMM15	SIMD Registers	128-bit registers used for floating-point and vector operations (SSE instructions).
YMM0–YMM15	Extended SIMD Registers	256-bit registers for vector operations (AVX instructions).
ZMM0–ZMM31	AVX-512 SIMD Registers	512-bit registers for advanced vector operations (AVX-512 instructions).
ST(0)–ST(7)	Floating-Point Registers	80-bit stack registers for floating-point arithmetic (legacy FPU).
CR0	Control Register 0	Controls CPU operations (e.g., enabling protected mode, paging).
CR2	Control Register 2	Holds the faulting address during a page fault.
CR3	Control Register 3	Holds the address of the page table base (used for virtual memory).
CR4	Control Register 4	Enables advanced processor features (e.g., PAE, SSE).
CR8	Control Register 8	Controls interrupt priority levels in x64.
DR0–DR3	Debug Registers	Used to set hardware breakpoints for debugging.

DR6	Debug Status Register	Provides information about debug exceptions.
DR7	Debug Control Register	Controls debug operations and sets conditions for breakpoints.

=====



These are common registers. Which is common in 32 and 64 bit machine.



AX → 16 bits

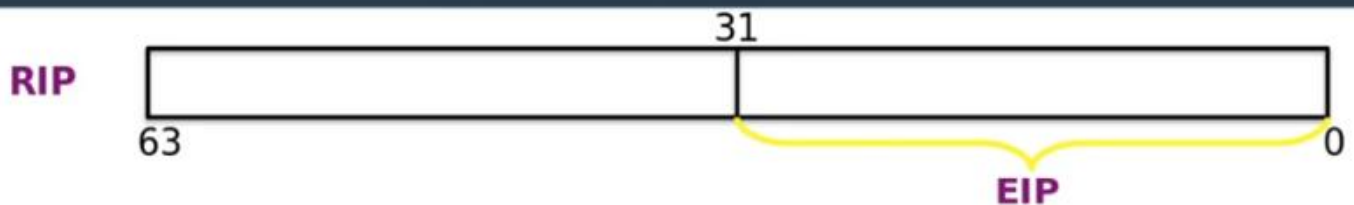
EAX → 32 bit

RAX → 64 bit

AH → Higher

AL → Lower

INSTRUCTION PONTER OR PROGRAM COUNTER



It tells what instruction will be executed next.

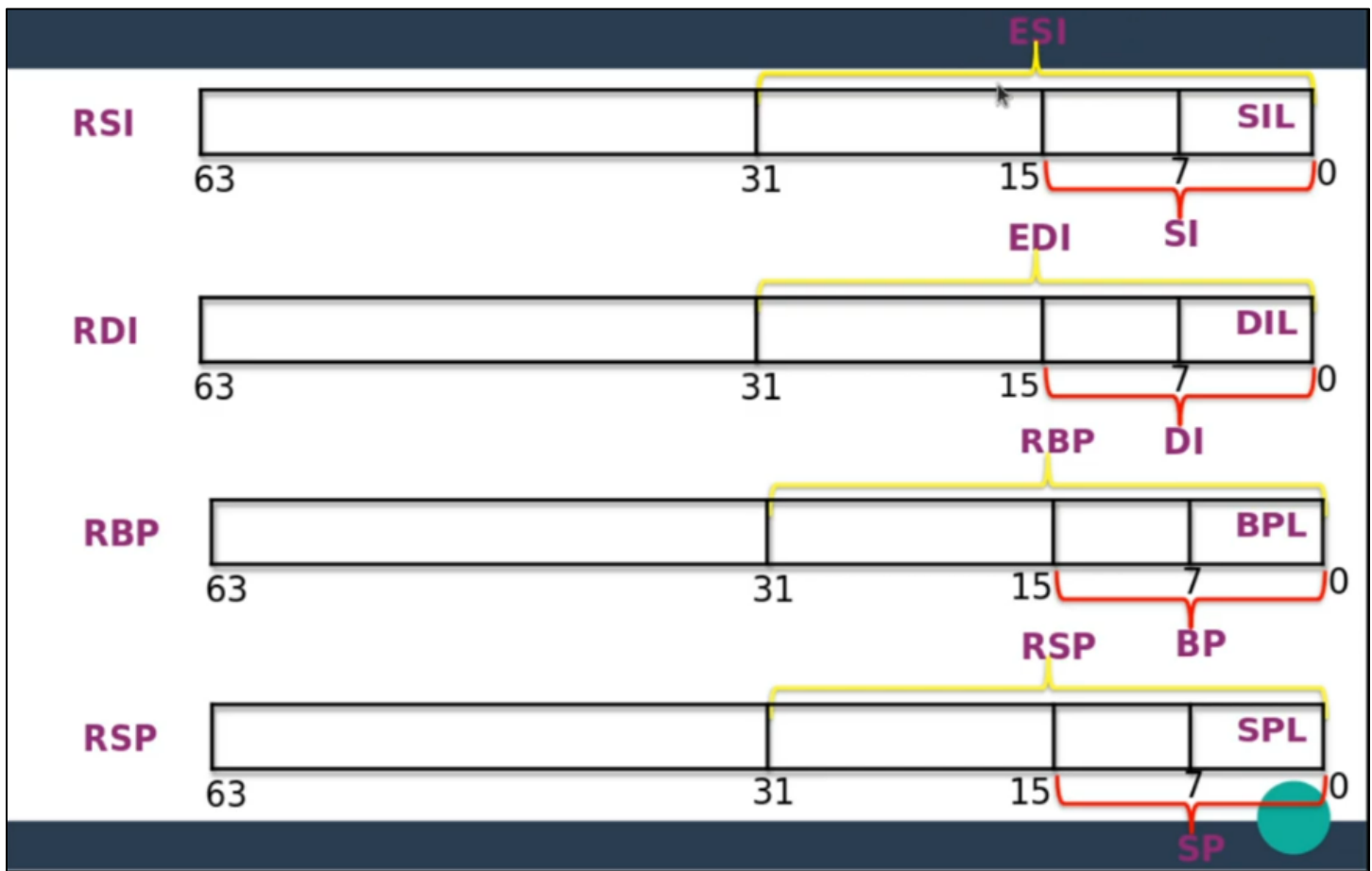
IP → 16 bits

EIP → 32 bit

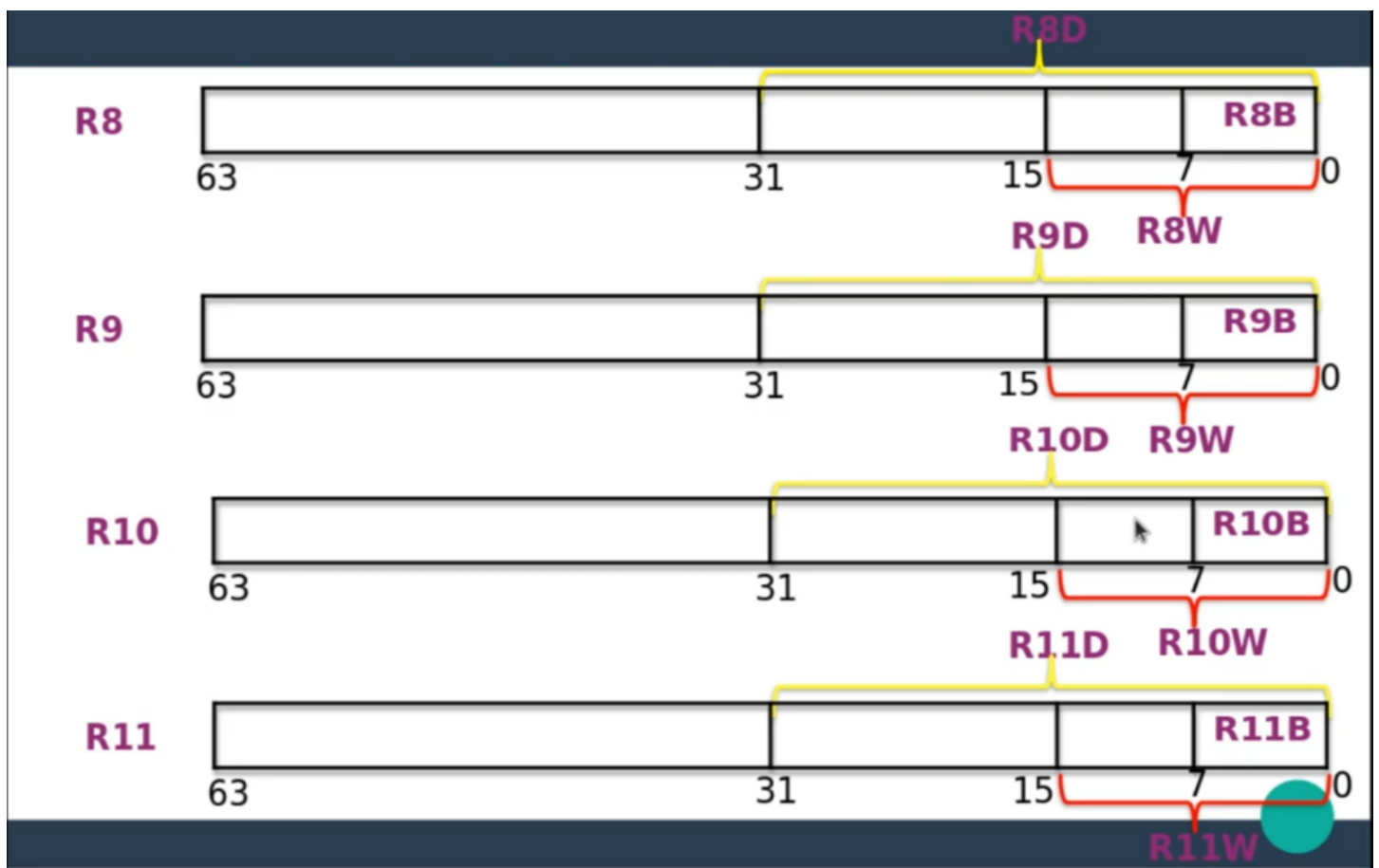
RIP → 64 bit

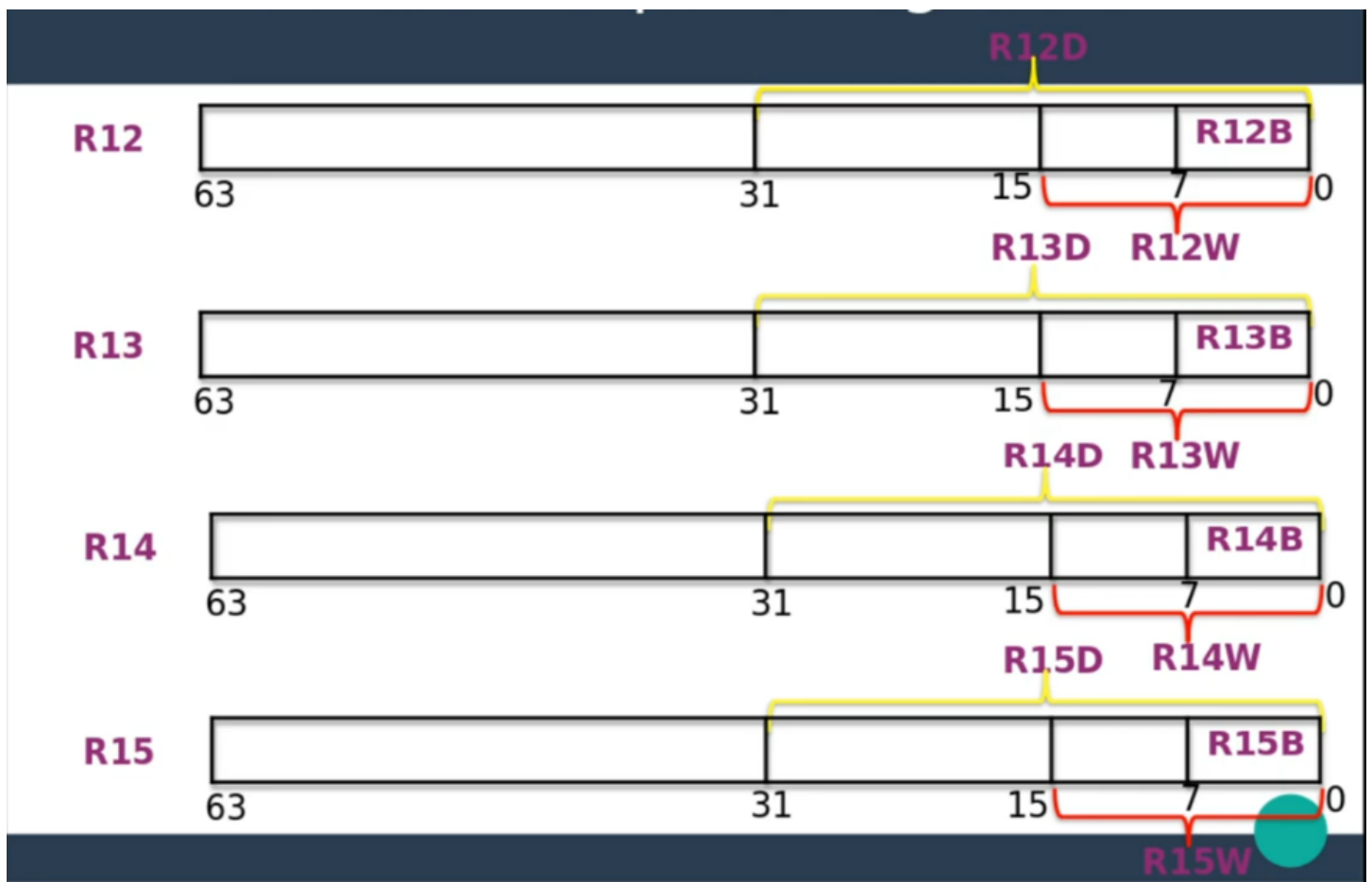
Another Registers:

We can't use its higher part.



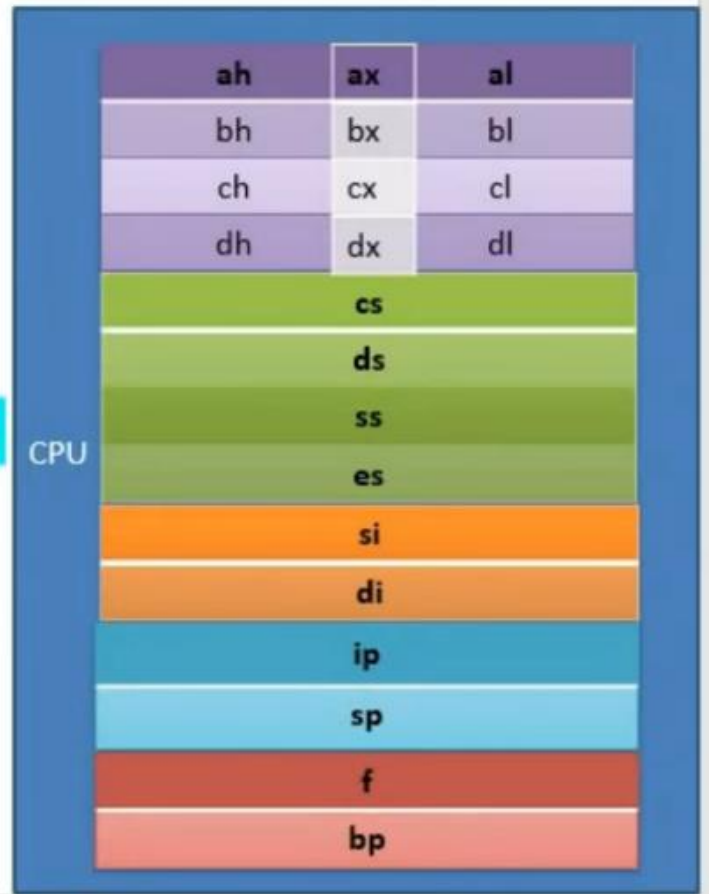
Additional registers (Only used in 64 bit processor).





Types of Registers

1. Accumulator (ah,al): Input, Output
2. Base(bh,bl): Holds address of data
3. Counter(ch,cl): Counting, Looping
4. Data(dh,dl): Holds Output Data
5. Code Segment(cs): Holds address of Code
6. Data Segment(ds): Holds address of data
7. Stack Segment(ss): Holds address of stack
8. Extra Segment(es): Holds address of Data
9. Source Index(si): Point the source operand
10. Destination Index(di): Point the destination operand
11. Instruction Pointer(ip): Holds next Instruction
12. Stack Pointer(sp): Point current of Stack
13. Flag Registers(f): Holds Overflow, Underflow
14. Base Pointer(bp): Base of top of stack



ASCII Code:

```

Dec Hex    Dec Hex    Dec Hex    Dec Hex    Dec Hex    Dec Hex    Dec Hex    Dec Hex
 0 00 NUL   16 10 DLE   32 20      48 30 0    64 40 @    80 50 P    96 60 `   112 70 p
 1 01 SOH   17 11 DC1   33 21 !    49 31 1    65 41 A    81 51 Q    97 61 a   113 71 q
 2 02 STX   18 12 DC2   34 22 "    50 32 2    66 42 B    82 52 R    98 62 b   114 72 r
 3 03 ETX   19 13 DC3   35 23 #    51 33 3    67 43 C    83 53 S    99 63 c   115 73 s
 4 04 EOT   20 14 DC4   36 24 $    52 34 4    68 44 D    84 54 T   100 64 d   116 74 t
 5 05 ENQ   21 15 NAK   37 25 %    53 35 5    69 45 E    85 55 U   101 65 e   117 75 u
 6 06 ACK   22 16 SYN   38 26 &    54 36 6    70 46 F    86 56 V   102 66 f   118 76 v
 7 07 BEL   23 17 ETB   39 27 '    55 37 7    71 47 G    87 57 W   103 67 g   119 77 w
 8 08 BS    24 18 CAN   40 28 (    56 38 8    72 48 H    88 58 X   104 68 h   120 78 x
 9 09 HT    25 19 EM    41 29 )    57 39 9    73 49 I    89 59 Y   105 69 i   121 79 y
10 0A LF    26 1A SUB   42 2A *    58 3A :    74 4A J    90 5A Z   106 6A j   122 7A z
11 0B VT    27 1B ESC   43 2B +    59 3B ;    75 4B K    91 5B [   107 6B k   123 7B {
12 0C FF    28 1C FS    44 2C ,    60 3C <    76 4C L    92 5C \   108 6C l   124 7C |
13 0D CR    29 1D GS    45 2D -    61 3D =    77 4D M    93 5D ]   109 6D m   125 7D }
14 0E SO    30 1E RS    46 2E .    62 3E >    78 4E N    94 5E ^   110 6E n   126 7E ~
15 0F SI    31 1F US    47 2F /    63 3F ?    79 4F O    95 5F _   111 6F o   127 7F DEL
ubuntu@ip-172-31-8-118:~$ ascii -t

```

dec	hex	oct	char	dec	hex	oct	char	dec	hex	oct	char	dec	hex	oct	char
0	0	000	NULL	32	20	040	space	64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	\$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051)	73	49	111	I	105	69	151	i
10	a	012	LF	42	2a	052	*	74	4a	112	J	106	6a	152	j
11	b	013	VT	43	2b	053	+	75	4b	113	K	107	6b	153	k
12	c	014	FF	44	2c	054	,	76	4c	114	L	108	6c	154	l
13	d	015	CR	45	2d	055	-	77	4d	115	M	109	6d	155	m
14	e	016	SO	46	2e	056	.	78	4e	116	N	110	6e	156	n
15	f	017	SI	47	2f	057	/	79	4f	117	O	111	6f	157	o
16	10	020	DLE	48	30	060	0	80	50	120	P	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	131	Y	121	79	171	y
26	1a	032	SUB	58	3a	072	:	90	5a	132	Z	122	7a	172	z
27	1b	033	ESC	59	3b	073	;	91	5b	133	[123	7b	173	{
28	1c	034	FS	60	3c	074	<	92	5c	134	\	124	7c	174	
29	1d	035	GS	61	3d	075	=	93	5d	135]	125	7d	175	}
30	1e	036	RS	62	3e	076	>	94	5e	136	^	126	7e	176	~
31	1f	037	US	63	3f	077	?	95	5f	137	_	127	7f	177	DEL

www.alpharithms.com

#####

Systemcalls

A **system call** is a mechanism that allows a user program to request a service from the operating system (OS). It acts as an interface between the application software and the OS, enabling programs to access hardware resources or perform tasks that require privileged operations.

How Does a System Call Work?

1. **Request Initiation:** A user program issues a system call when it needs a service (e.g., reading a file or allocating memory).
2. **Switch to Kernel Mode:** The system switches from **user mode** to **kernel mode** (a privileged mode with direct access to hardware and system resources).
3. **Execution:** The OS performs the requested operation.
4. **Return to User Mode:** The system switches back to user mode and returns the result to the user program.

Examples of System Calls

1. File Operations:

- `open()`: Open a file.
- `read()`: Read from a file.
- `write()`: Write to a file.
- `close()`: Close a file.

2. Process Control:

- `fork()`: Create a new process.
- `exec()`: Execute a new program.
- `exit()`: Terminate a process.
- `wait()`: Wait for a child process to finish.

3. Memory Management:

- `mmap()`: Map a file or device into memory.
- `brk()`: Change the size of the data segment.

4. Device Management:

- `ioctl()`: Control a device.
- `read()/write()`: Interact with device drivers.

5. Network Operations:

- `socket()`: Create a socket.
- `bind()`: Bind a socket to an address.
- `connect()`: Connect to a remote server.

6. Information Maintenance:

- `getpid()`: Get the process ID.
- `gettimeofday()`: Get the current time.
- `uname()`: Get system information.

Why Are System Calls Important?

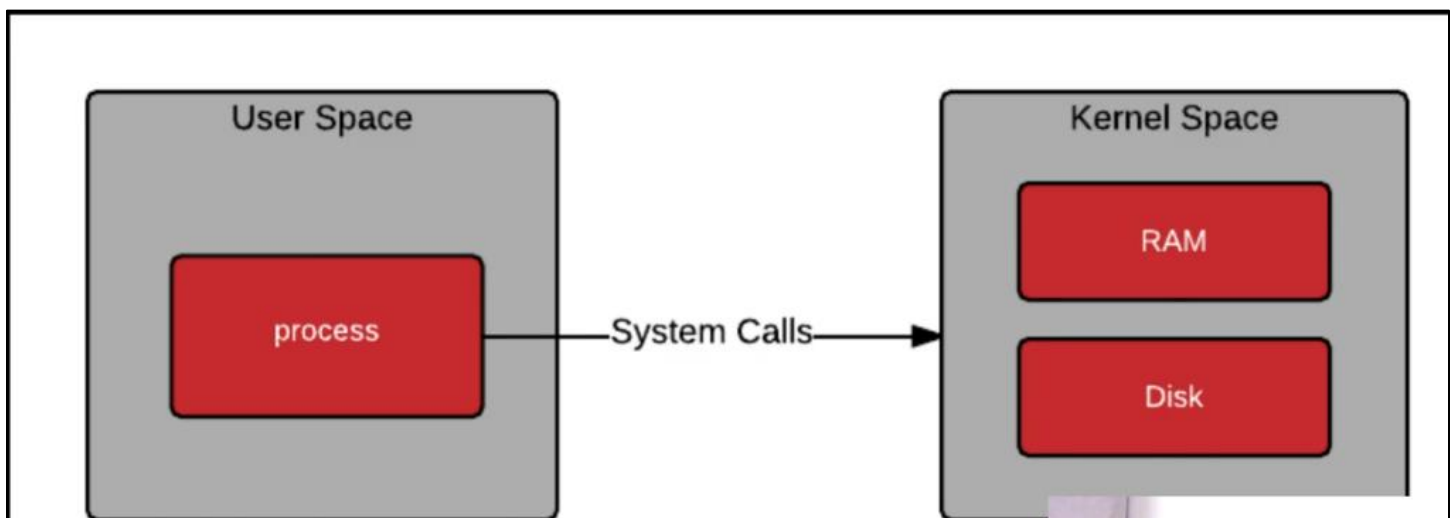
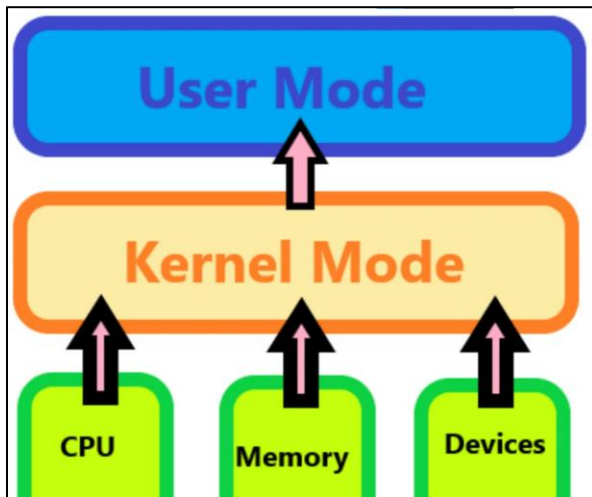
- **Resource Access:** They provide controlled access to system resources (e.g., CPU, memory, devices).
- **Abstraction:** Hide low-level hardware details, offering a simpler API for developers.
- **Security:** Ensure that only the OS has direct hardware access, protecting the system from malicious or erroneous programs.
- **Portability:** Abstracting hardware-specific operations allows applications to run on different platforms without modification.

System Call vs. Function Call

Aspect	System Call	Function Call
Mode of Execution	Switches to kernel mode.	Remains in user mode.
Execution Speed	Slower due to context switching.	Faster as no mode switching occurs.
Purpose	Requests OS services.	Performs operations within a program.
Access Level	Accesses hardware and system resources.	Limited to program's resources.

How to communicate User mode and Kernal Mode:

User mode and Kernal Mode communicate with the help of System calls.



Path to see all systemcalls in linux:

```
cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h
```

```
cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h
```

To see about any syscall:

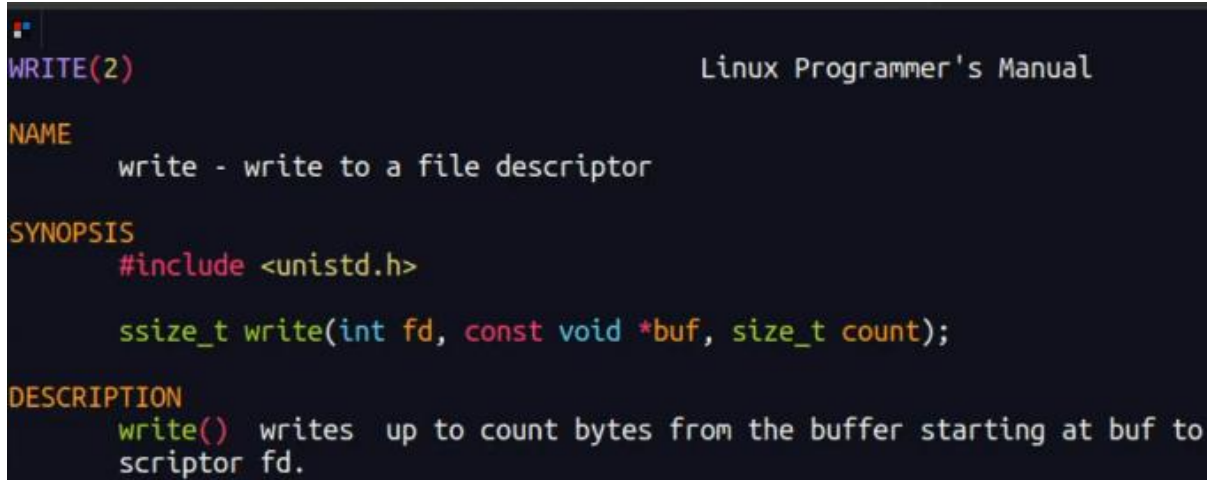
```
man 2 <syscall_name>
```

Ex.

```
man 2 write
```

Here, 2 for see syscall **or**

```
man fwrite
```



WRITE(2) Linux Programmer's Manual

NAME

write - write to a file descriptor

SYNOPSIS

```
#include <unistd.h>
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

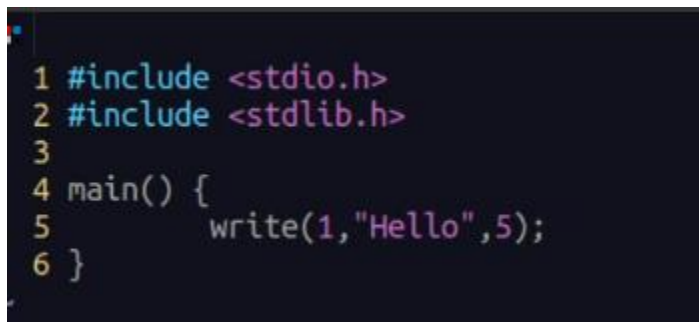
DESCRIPTION

write() writes up to count bytes from the buffer starting at buf to scriptor fd.

It takes 3 arguments first **file descriptor**, buffere(data) and length of data.

Here output number is 1, input number is 0 and error number is 2.

Example of syscall with a C program.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 main() {
5     write(1,"Hello",5);
6 }
```

There are 3 types of file descriptor.

- 1) Input
- 2) Output
- 3) Error

```

→ gcc syscalls.c -o syscalls
syscalls.c:4:1: warning: return type defaults to 'int' [-Wimplicit-int]
  4 | main() {
    | ^~~~~~
syscalls.c: In function 'main':
syscalls.c:5:2: warning: implicit declaration of function 'write'; did you mean
'atexit' [-Wimplicit-declaration]
  5 |     write(1,"Hello",5);
    |     ^~~~~~
    |     fwrite
→

```

Ignore the warning and run it.

Output

```

~ → ./syscalls
Hello~ →

```

Exit syscall

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 main() {
5     write(1,"Hello",5);
6     exit(11);
7 }
~
~

```

man 2 exit

```
_EXIT(2) Linux Programmer's Manual
NAME
    _exit, _Exit - terminate the calling process
SYNOPSIS
    #include <unistd.h>

    void _exit(int status);

    #include <stdlib.h>
    void _Exit(int status);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

    _Exit():
        _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
DESCRIPTION
```

It takes only one argument 'status'. We can give any number as status if it is successfully exit then return that number otherwise return different number.

Ok, run it.

Output

```
~ → ./syscalls
Hello~ →
```

Only hello show. And to see status code.

Write

echo \$?

```
~ → ./syscalls
Hello~ → echo $?
11
~ →
```

Here, we put write directly in C programming. But in assembly we use syscall number to use syscall.

#####

Instruction in Assembly

In computer architecture, an **instruction** refers to a command given to the processor to perform a specific operation. Instructions are stored in memory and executed by the processor, with registers playing a crucial role in facilitating this execution.

Role of Registers in Instruction Execution

Registers are used during instruction execution for the following purposes:

1. Instruction Fetch:

- The **Instruction Pointer (RIP)** register holds the address of the next instruction to execute.
- The processor fetches the instruction from memory and updates the instruction pointer.

2. Instruction Decode:

- The fetched instruction is decoded into its operation and operands.
- Decoded data may involve values in general-purpose registers (e.g., RAX, RBX) or immediate values.

3. Operand Fetch:

- If the instruction involves registers, the processor reads or writes the operands from/to the specified registers (e.g., MOV RAX, RBX moves the value from RBX to RAX).

4. Execution:

- Arithmetic and logical operations are performed directly on registers (e.g., ADD RAX, RBX adds the values in RAX and RBX).

5. Instruction Result Storage:

- The result of an operation is typically stored back in a register or written to memory.

Types of Instructions Using Registers

1. Data Transfer Instructions:

- MOV RAX, RBX - Transfers data from RBX to RAX.
- PUSH RAX - Pushes the value of RAX onto the stack.

2. Arithmetic Instructions:

- ADD RAX, RBX - Adds RBX to RAX.
- SUB RAX, RBX - Subtracts RBX from RAX.

3. Logical Instructions:

- AND RAX, RBX - Performs a bitwise AND on RAX and RBX.
- OR RAX, RBX - Performs a bitwise OR.

4. Control Transfer Instructions:

- JMP address - Jumps to the specified address (uses RIP).
- CALL address - Calls a function at the specified address.

5. Comparison and Branching:

- CMP RAX, RBX - Compares RAX with RBX.
- JE address - Jumps if equal.

6. Shift and Rotate:

- SHL RAX, 1 - Shifts RAX left by 1 bit.
- ROR RBX, 1 - Rotates RBX right by 1 bit.

Key Registers Used in Instructions

Register	Purpose in Instructions
RAX, RBX, RCX, RDX	General-purpose, used for arithmetic, logic, and data transfer.
RIP	Holds the address of the next instruction to execute.
RFLAGS	Stores flags that affect control flow (e.g., Zero Flag, Carry Flag).
RSI, RDI	Used for string and memory operations (source and destination index).
RSP, RBP	Stack pointer and base pointer for stack operations.

All Instruction name, Full name, Description.

Instruction	Full Name	Description
MOV	Move	Transfers data from a source to a destination (e.g., MOV RAX, RBX).
ADD	Add	Adds the source operand to the destination operand (e.g., ADD RAX, RBX).
SUB	Subtract	Subtracts the source operand from the destination operand (e.g., SUB RAX, RBX).
MUL	Multiply	Multiplies the source operand with the accumulator (e.g., MUL RBX).
DIV	Divide	Divides the accumulator by the source operand (e.g., DIV RBX).
INC	Increment	Increases the value of the operand by 1 (e.g., INC RAX).
DEC	Decrement	Decreases the value of the operand by 1 (e.g., DEC RAX).
CMP	Compare	Compares two operands and sets flags (e.g., CMP RAX, RBX).
JMP	Jump	Unconditionally jumps to a specified address (e.g., JMP LABEL).
JE/JZ	Jump if Equal/Zero	Jumps to a specified address if the Zero Flag is set (e.g., JE LABEL).
JNE/JNZ	Jump if Not Equal/Not Zero	Jumps if the Zero Flag is not set (e.g., JNE LABEL).
JG/JA	Jump if Greater/Above	Jumps if the destination is greater than the source (e.g., JG LABEL).
JL/JB	Jump if Less/Below	Jumps if the destination is less than the source (e.g., JL LABEL).
CALL	Call	Transfers control to a subroutine (e.g., CALL FUNCTION).
RET	Return	Returns control to the calling procedure (e.g., RET).
PUSH	Push	Places data onto the stack (e.g., PUSH RAX).
POP	Pop	Removes data from the stack (e.g., POP RAX).

NOP	No Operation	Does nothing; often used for timing or alignment (e.g., NOP).
INT	Interrupt	Triggers a software interrupt (e.g., INT 0x80).
HLT	Halt	Stops processor execution until the next interrupt (e.g., HLT).
AND	Logical AND	Performs a bitwise AND operation (e.g., AND RAX, RBX).
OR	Logical OR	Performs a bitwise OR operation (e.g., OR RAX, RBX).
XOR	Logical XOR	Performs a bitwise XOR operation (e.g., XOR RAX, RBX).
NOT	Logical NOT	Inverts all the bits of the operand (e.g., NOT RAX).
SHL	Shift Left	Shifts the bits of the operand to the left (e.g., SHL RAX, 1).
SHR	Shift Right	Shifts the bits of the operand to the right (e.g., SHR RAX, 1).
LEA	Load Effective Address	Loads the address of a memory operand into a register (e.g., LEA RAX, [RBX+4]).
TEST	Test	Performs a bitwise AND without modifying the operands; sets flags based on the result.
LOOP	Loop	Decrements the RCX register and jumps if it is not zero (e.g., LOOP LABEL).
STOS	Store String	Stores a value from AL/AX/EAX/RAX into a memory location addressed by RDI.
SCAS	Scan String	Compares a string value in memory with AL/AX/EAX/RAX.
CLD	Clear Direction Flag	Sets the string operation direction to increment (default).
STD	Set Direction Flag	Sets the string operation direction to decrement.
IN	Input	Reads data from an I/O port into the accumulator (e.g., IN AL, DX).
OUT	Output	Writes data from the accumulator to an I/O port (e.g., OUT DX, AL).

=====

Example

Here, sequence matters first destination after that source.

MOV Instruction

MOV [DESTINATION],[SOURCE]

REGISTERS	VALUES
RAX	0x0
RBX	0x0
RCX	0x0
RDX	0x0

MOV rax,0x1

MOV rbx,0xa

MOV rcx,rbx

MOV rax,rdx

Here we move 1 to rax register.

REGISTERS	VALUES
RAX	0x1
RBX	0xa
RCX	0xa
RDX	0x0

after last operation

REGISTERS	VALUES
RAX	0x0
RBX	0xa
RCX	0xa
RDX	0x0

SUB Instruction

REGISTERS	VALUES
RAX	0x10
RBX	0x20
RCX	0x30
RDX	0x40

```
SUB    rax,0x2
```

```
SUB    rbx,0xa
```

```
SUB    rcx,rbx
```

```
SUB    rdx,rax
```

Will be

REGISTERS	VALUES
RAX	0x08
RBX	0x16
RCX	0x30
RDX	0x38

CMP Instruction

Here, we can't store data in that register that holds data then we use flag register. We use zero flag register if condition is true then zero flag value will be 1. If false then zero flag value will be 0.

REGISTERS	VALUES
RAX	0x1
RBX	0x2
RCX	0x3
RDX	0x4

`CMP rax,0x1`

`CMP rbx,0x3`

`CMP rcx,rbx`

`CMP 0x4,rdx`

- 1) zero flag = 1
- 2) zero flag = 0
- 3) zero flag = 0
- 4) zero flag = 1

TEST Instruction

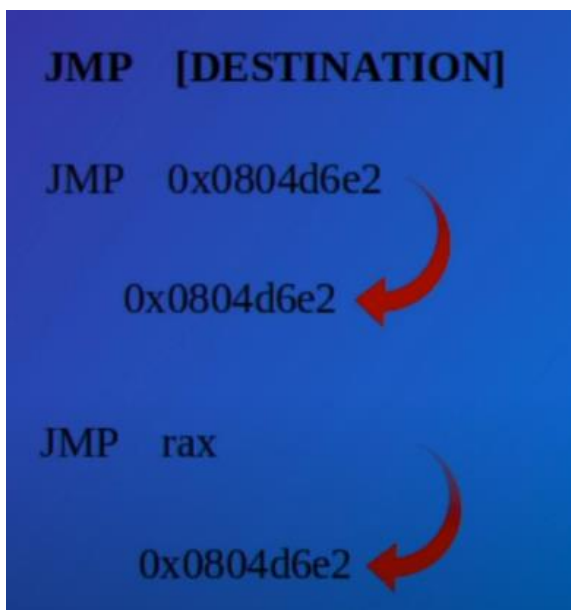
REGISTERS	VALUES
RAX	0x0
RBX	0x2
RCX	0x0
RDX	0x0

```
TEST    rax,rax  
  
TEST    rbx,rbx
```

Here, we use zero flag as above. True then it will be 1, false then it will be 0.

JUMP Instruction

REGISTERS	VALUES
RAX	0x0804d6e2
RBX	0x0
RCX	0x0
RDX	0x0



We can give memory address or register name.

JMP is also called unconditional jump.

JE/JZ (conditional Jump) Jump if it equal:

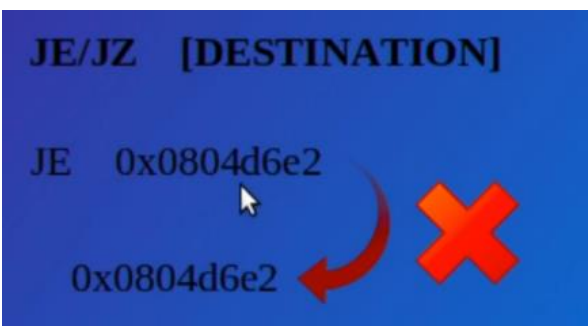
REGISTERS	VALUES
RAX	0x0804d6e2
RBX	0x0
RCX	0x0
RDX	0x0

JE/JZ [DESTINATION]

JE 0x0804d6e2

ZF	0
----	---

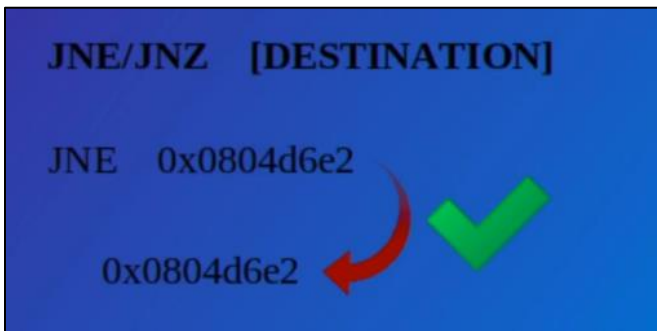
Here, it will go to given address and check zero flag value. If it get 1 then jump otherwise not.



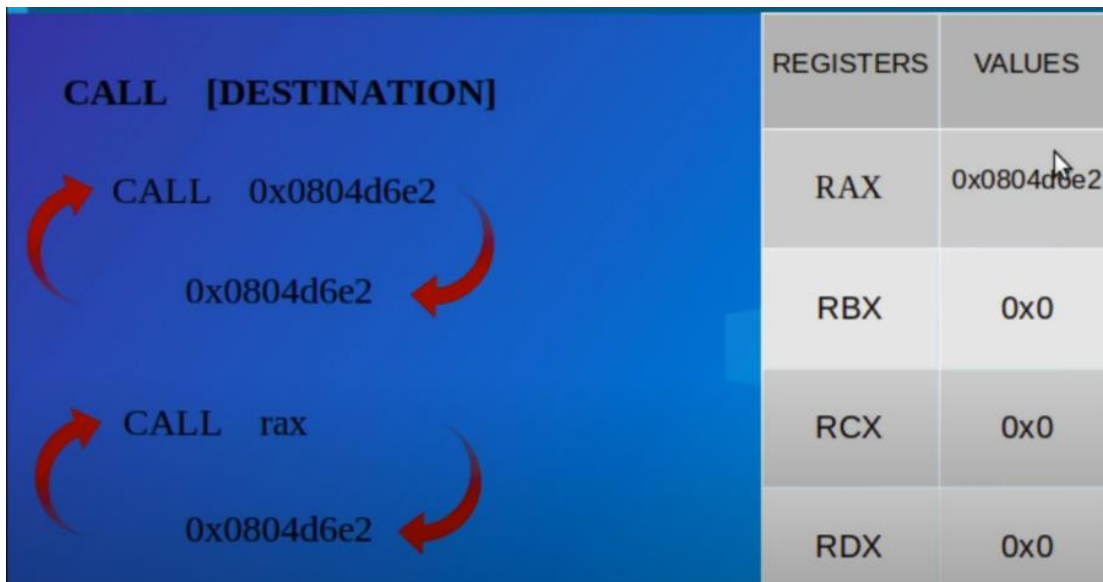
JNE/JNZ (conditional jump) Jump not equal to

It is opposite of JE/JZ. It will jump if zero flag value 0. If 1 then it will not jump.

ZF 0



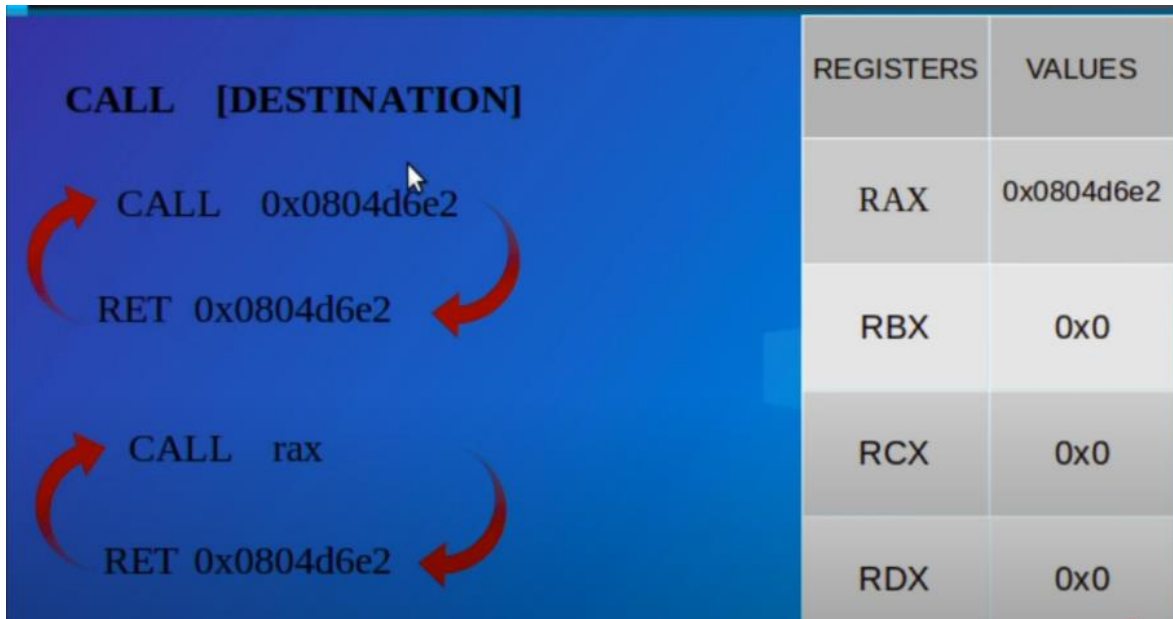
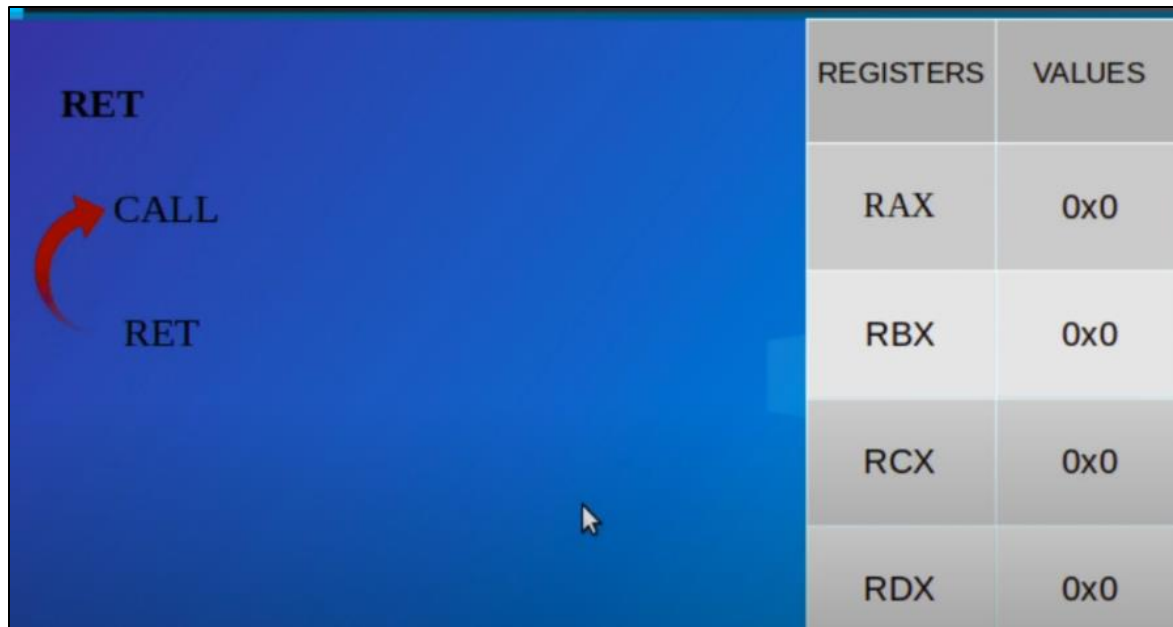
CALL Instruction



Call is like jump but jump exit after excution but call return again on that point from where it jumped.

RET Instruction

RET is used to return the call instruction as soon CALL get RET it returns back.



SYSCALL Instruction

Syscall is very important instruction. It can perform lots of operations such as read, write, open, close, execute, exit etc.

0x3c means **60**

SYSCALL SYSCALL	57	sys_fork	REGISTERS	VALUES
	58	sys_vfork		
	59	sys_execve	RAX	0x3c
	60	sys_exit	RDI	0xb
	61	sys_wait4	RSI	0x0
	62	sys_kill		
	63	sys_uname	RDX	0x0

```

Man 2 _exit
_EXIT(2)
Linux Programmer

NAME
    _exit, _Exit - terminate the calling process

SYNOPSIS
    #include <unistd.h>
    void _exit(int status);
    #include <stdlib.h>
    void _Exit(int status);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```

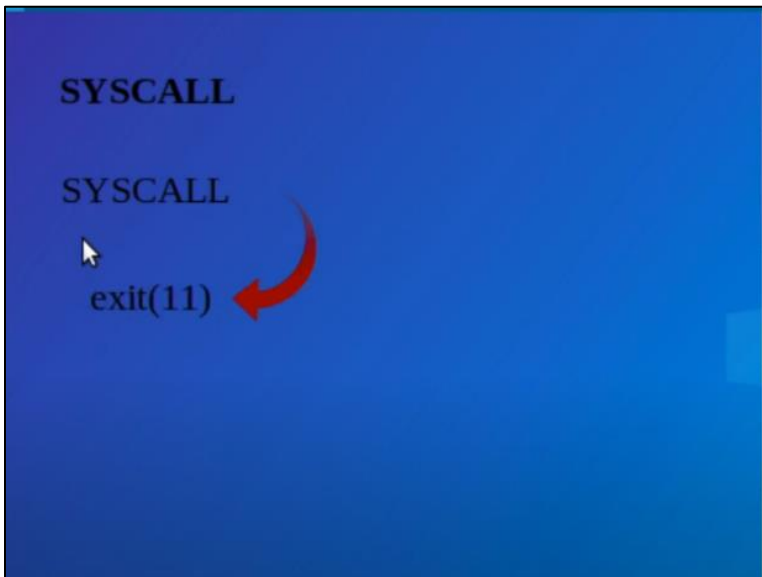
Exit takes a argument.

How can pass argument in registers

If rax is used then use RDI and rdi is used then RSI register as repectively.

Register	Value
RAX	System call number
RDI	1st argument
RSI	2nd argument
RDX	3rd argument
RCX	4th argument
R8	5th argument
R9	6th argument

So on **R8-to-R15**

	REGISTERS	VALUES
	RAX	0x3c
	RDI	0xb
	RSI	0x0
	RDX	0x0

Here, argument is only one then it will go to RDI. Rest all are empty.

#####

Let's write Assembly Program

To check any file type in linux

```
~/yt → ls
final
~/yt → file final
final: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
~/yt → ./final
hello world~/yt →
```

Create a assembly file with .asm extension

```
~/yt → subl helloworld.asm
```

So, there are some sections.

_Start section – it is a initial point from where our assembly program starts.

Data section – this section contains variable (initialized variable) jiski value pta ho.

Text section – iske andar humara program code aata hai.

Bss section – iske andar uninitialized variables aate hai jinki value hume nhi pta ho jaise ki hum user se lenge.

; - semicolon is used to comment our in assembly program.

Here, our task is print "hello world". So here we have to perform write operation. Who can write 'hello world' that is kernel. So we will say to kernel through syscall write 'hello world'.

Note.: in assembly, we used syscall number instead of name.

Let's see number of a syscall.

```

~/yt → cat /usr/include/x86_64-linux-gnu/asm/unistd_32.h  unistd_64.h  unistd.h  unistd_x32.h
~/yt → cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h
#ifndef _ASM_X86_UNISTD_64_H
#define _ASM_X86_UNISTD_64_H 1

#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7

```

So write syscall number is **1**.

Syscall ka number hum humesa **rax** ke andar store krte hai. And first argument rdi and so on.

Register	Value
RAX	System call number
RDI	1st argument
RSI	2nd argument
RDX	3rd argument
RCX	4th argument
R8	5th argument
R9	6th argument

Ok, let's see write syscall manuals.

```
man 2 write
```

write - write to a file descriptor

```
#include <unistd.h>
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

```
write() writes up to count bytes from the buffer starting at buf to the
```

File descriptors

`write()` writes up to count bytes from the buffer starting at buf to the scriptor fd.

The number of bytes written may be less than count if, for example, the underlying physical device or the LIMIT_FSDATA endpoint limit is encoded and was interrupted by a signal handler after having written less than

1 = stdout -> output

0 = stdin -> input

1 = stdout -> output

2 = stderr -> error

```
global _start
```

```
section .text
```

```
_start:
```

```
; print hello world
```

```
mov rax, 1 ; write syscall
```

```
mov rdi, 1 ;fd -> 1 (output)
```

```
mov rsi, hello ;buffer -> hello -> 'hello world'
```

```
mov rdx, 11 ;count -> 11 (size)
```

syscall

```
section .data
```

```
hello: db 'hello world'
```


Yha **hello** 'hello world' ko point kr rha hai.

Aur hum jb bhi data section me write krte hai to hume datatype batana pdta hai. **db** means **data bytes**.

Assemble the Code:

Use **nasm** to assemble the code into an object file:

```
nasm -f elf64 program.asm -o program.o
```

- **-f elf64**: Specifies the output format (64-bit ELF for Linux).
- **-o program.o**: Specifies the output file name.

Link the Object File:

Use **ld** (the linker) to create an executable: (linker add some necessary libraries code that is make it runnable)

```
ld program.o -o program
```

```
~/yt → file helloworld
helloworld: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
```

Run the Executable:

Run the program using:

```
./program
```

Output-

```
hello world
```

Debugging (Optional):

To debug the assembly program, use **gdb**:

```
gdb ./program
```

Ensure that your assembly code uses the correct calling **conventions** and **system calls** for your platform (e.g., Linux x86-64).

=====

When we run our program...

```
~/yt → ./helloworld
hello worldSegmentation fault (core dumped)
```

It printed hello world but an error was shown segmentation fault (core dumped).

Yahan program jaise hi hello world print kiya fir ise samjh nhi aaya ki aage kya krna hai. Because hmne exit nhi likha aur assembly me sb kuchh batana pdta hai.

Let's implement exit

```
global _start

section .text

_start:
    ; print hello world
    mov rax, 1 ; write syscall
    mov rdi, 1 ; fd -> 1 (output)
    mov rsi, hello ; buffer -> hello -> 'hello world'
    mov rdx, 11 ; count -> 11 (size)
    syscall

    mov rax, 60 ; exit syscall
    mov rdi, 11 ; status code -> 22
    syscall

section .data

hello: db 'hello world'
```

Assemble and link

Output:

```
~/yt → ./helloworld
hello world~/yt →
```

#####

Take input from user

Program looks like-

```
~/yt → ./final1
Enter Your Name : Harshit
hello, Harshit
~/yt →
```

```
global _start

section .text

_start:

    mov rax,1
    mov rdi,1
    mov rsi,hello
    mov rdx,

section .data

hello: db 'Enter Your Name :|'
hello_length: equ $-hello
```

Here, \$ is point out last word and – the whole string from hello variable to get the exact **length of data bytes** and store in **hello_length** variable.

user_input.asm

```
global _start

section .text
```

```

_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, hello
    mov rdx, hello_length
    syscall

    mov rax, 60
    mov rdi, 22
    syscall

section .data

    hello: db 'Enter Your Name : '
    hello_length: equ $ - hello

```

Assemble and link

Output:

```

~/yt → ./user_input
Enter Your Name : ~/yt →

```

So, now we will take read syscall to get input from user.

```

Linux Programmer's Manual

NAME
    read - read from a file descriptor

SYNOPSIS
    #include <unistd.h>

    ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
    read() attempts to read up to count bytes from file descriptor fd.

```

user_input.asm

```

global _start

section .text

```

```

_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, hello
    mov rdx, hello_length
    syscall

user_input:
    mov rax, 0
    mov rdi, 0
    mov rsi, input
    mov rdx, 100
    syscall

exiting_program:
    mov rax, 60
    mov rdi, 22
    syscall

section .data

    hello: db 'Enter Your Name : '
    hello_length: equ $ - hello

section .bss
    input: resb 100

```

Here, **resb** means it reserve 100 characters so we can give 100 or less than 100. Otherwise it will cut your input.

Jiska value nhi pta ya jise hum user se lene wale hai use hum .bss section me rkhte hai.

Ouput

```

~/yt → ./user_input
Enter Your Name : harshit

```

user_input.asm

```

global _start

section .text

_start:
    mov rax, 1

```

```

    mov rdi, 1
    mov rsi, wlc_mssg
    mov rdx, wlc_length
    syscall

user_input:
    mov rax, 0
    mov rdi, 0
    mov rsi, input
    mov rdx, 100
    syscall

printing_hello:
    mov rax, 1
    mov rdi, 1
    mov rsi, hello
    mov rdx, hello_length
    syscall

exiting_program:
    mov rax, 60
    mov rdi, 22
    syscall

section .data

    wlc_mssg: db 'Enter Your Name : '
    wlc_length: equ $ - wlc_mssg
    hello: db 'hello, '
    hello_length: equ $ - hello

section .bss
    input: resb 100

```

Output

```

~/yt → ./user_input
Enter Your Name : harshit
hello, ~/yt →

```

Jaisa ki hum jante hai ki jb bhi syscall hoti hai to wh kuchh na kuchh value return krta hai aur **rax** ke andar store kr deta hai. jaise hum **exit syscall** krte hai to wh jo bhi value dete hai such 11 to 11 return krta hai. Theek usi tarah **read syscall** input length return krta hai.

So, yha **rax** ki value kisi dusre register me save kr lete hai ki usse phle program me kahi aur overwrite na kr diya jaye. To pure program me **rbx** kahi nhi use hua hai to hum **rax** ki value **rbx** me mov kr dete hai.

```
user_input:
    mov rax, 0
    mov rdi, 0
    mov rsi, input
    mov rdx, 100
    syscall
    mov rbx, rax
```

```
global _start

section .text

_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, wlc_mssg
    mov rdx, wlc_length
    syscall

user_input:
    mov rax, 0
    mov rdi, 0
    mov rsi, input
    mov rdx, 100
    syscall
    mov rbx, rax

printing_hello:
    mov rax, 1
    mov rdi, 1
    mov rsi, hello
    mov rdx, hello_length
    syscall

printing_userinput:
    mov rax, 1
    mov rdi, 1
    mov rsi, input
    mov rdx, rbx                ; use return value (mov rbx, rax)
    syscall

exiting_program:
    mov rax, 60
```

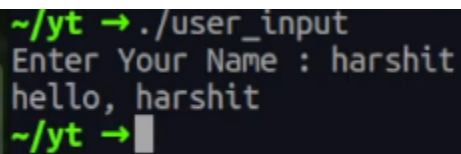
```
mov rdi, 22
syscall

section .data

wlc_mssg: db 'Enter Your Name : '
wlc_length: equ $ - wlc_mssg
hello: db 'hello, '
hello_length: equ $ - hello

section .bss
input: resb 100
```

Output

A terminal window with a dark background. The prompt is ~/yt →. The first line shows the program output: Enter Your Name : harshit. The second line shows: hello, harshit. The prompt is repeated as ~/yt → followed by a cursor.

```
~/yt → ./user_input
Enter Your Name : harshit
hello, harshit
~/yt →
```

Yha cursor hello harshit print krne ke bad next line me kyu aa gya.

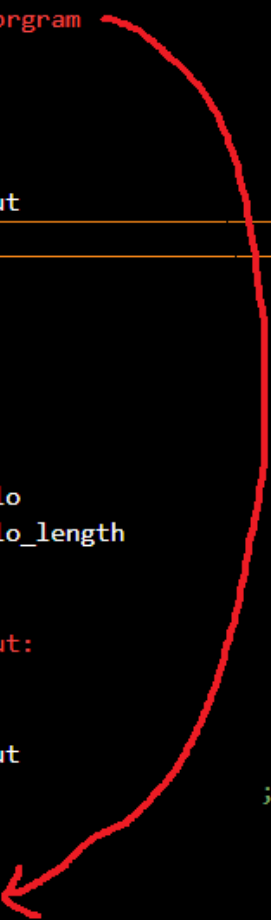
Because we hit enter after type harshit. Then there is not only harshit, yor give total input **'harshit\n'** because enter key consider as **'\n'**.

Lable is can be used in jmp instruction


```

4
5  _start:
6      mov rax, 1
7      mov rdi, 1
8      mov rsi, wlc_mssg
9      mov rdx, wlc_length
10     syscall
11     jmp exiting_prgram
12
13  user_input:
14      mov rax, 0
15      mov rdi, 0
16      mov rsi, input
17      mov rdx, 100
18      syscall
19      mov rbx, rax
20
21  printing_hello:
22      mov rax, 1
23      mov rdi, 1
24      mov rsi, hello
25      mov rdx, hello_length
26      syscall
27
28  printing_userinput:
29      mov rax, 1
30      mov rdi, 1
31      mov rsi, input
32      mov rdx, rbx          ; use return value (mov rbx, rax)
33      syscall
34
35  exiting_prgram:
36      mov rax, 60
37      mov rdi, 22
38      syscall
39

```



Output

```

~/yt → ./user_input
Enter Your Name : ~/yt →

```

It will not execute code between exiting_program and _start .

#####

Basic Maths in x64 Assembly Language and output show in Debugger:

- 1) Addition
- 2) Subtraction
- 3) Multiplication
- 4) Division

1) Addition

```
mov rax, 2  
add rax, 3
```

Here, first we **mov 2** into **rax** and **add 3** into **2** and after that output stored in **rax**.

Here, one thing is mandatory that we have to store **ouput value** in **rax** with addtion and subtraction.

2) Subtraction

```
mov rax, 2  
sub rax, 2
```

Here, first we **mov (put) 2** into **rax** and **subtract 2** from **2**. After that value of **rax** will be **0**.

3) Multiplication

```
mov rax, 6  
imul rax, rax, 2
```

It works like addtion and substruction but use **imul** for multiplication and we need to store output in any register. We can store in any register such as **rax, rbx, rdi** etc.

4) Division

$$\begin{array}{r}
 \text{Quotient} \swarrow \\
 72 \\
 \text{Divisor} \rightarrow 5 \overline{) 361} \leftarrow \text{Dividend} \\
 \underline{-35} \downarrow \\
 11 \\
 \underline{-10} \\
 1 \leftarrow \text{Remainder}
 \end{array}$$

Divident = 100

Divisor = 2

```

mov rax, 100
mov rbx, 2
idiv rbx

```

```

rdx = 1
rax = 1

```

```

11 / 2 |

```

```

mov rdx, 0
mov rax, 100
mov rbx, 2
idiv rbx

```

Yha pr **quotient rax** me store hota hai aur **reminder rdx** me store hota hai.

Here, we can't store **divisor** in **rax and rdx** and we can't write **divisor** value directly we need to write it in any register except **rax and rdx**.

And here adx and rax register aapas me concatnate hokr ek single number banate hai fir divide hote hai. jaisa ke upar ke image me hai.

One more example

```
mov rdx,0
mov rax,100
mov rbx,2
idiv rbx

0100 / 2
```

math.asm

```
global main

section .text

main:

    mov rax, 2
    add rax, 3

    mov rax, 2
    sub rax, 2

    mov rax, 6
    imul rax, rax, 2

    mov rdx, 0
    mov rax, 100
    mov rbx, 2
    idiv rbx

_exit:

    mov rax, 60
    mov rdi, 0
    syscall
```

```
~/yt → nasm -f elf64 math.asm -o math.o
~/yt → ld mat
math.asm math.o
~/yt → ld math.o -o math
ld: warning: cannot find entry symbol start; defaulting to 0000000000401000
~/yt →
```

It raised an error. It couldn't find `_start`.

```
~/yt → ld -e main math.o -o math
~/yt → ./math
~/yt →
```

Write **-e main**

Let's see output in debugger.

Download and install cutter from github.

The screenshot shows a web browser window with the address bar containing `https://github.com/rizinorg/cutter`. The page displays the GitHub repository for `cutter`. The main content area shows a list of recent commits, including:

- Remove freenode mentions in remaining files (#2703) - 2 hours ago
- Enable CUTTER_USE_BUNDLED_RIZIN by default (#2622) - 3 months ago
- Update Rizin version (#2698) - 11 days ago

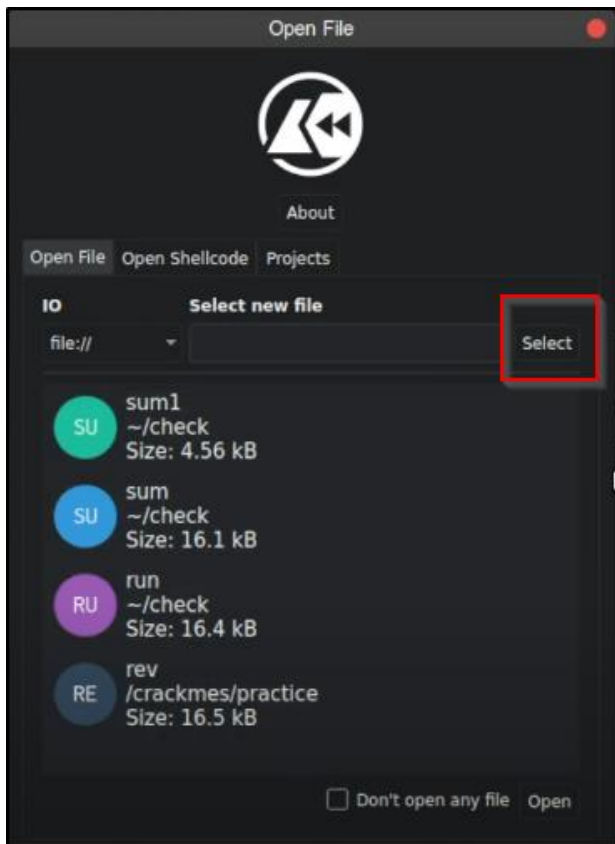
On the right sidebar, the **Releases** section is highlighted with a red box, showing version **2.0.2** as the latest release on 25 Apr.



```
mv cutter-v2.0.3-x64....AppImage cutter
sudo cp cutter /usr/local/bin/
```

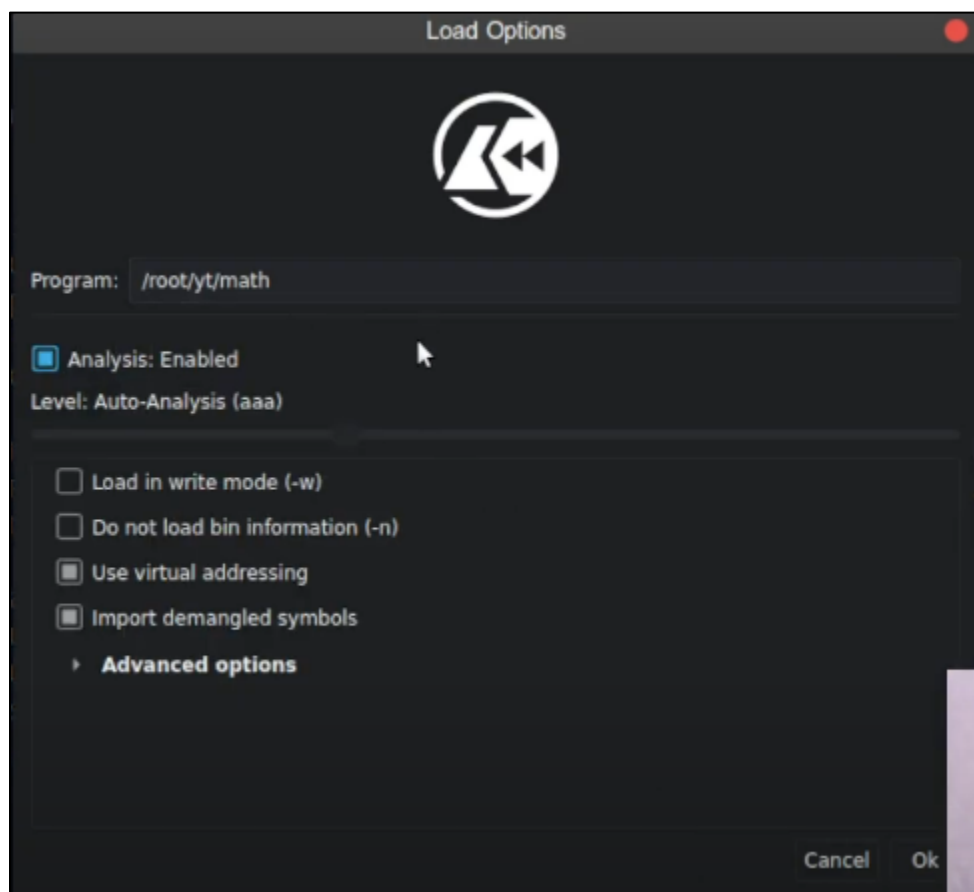
Run cutter

cutter

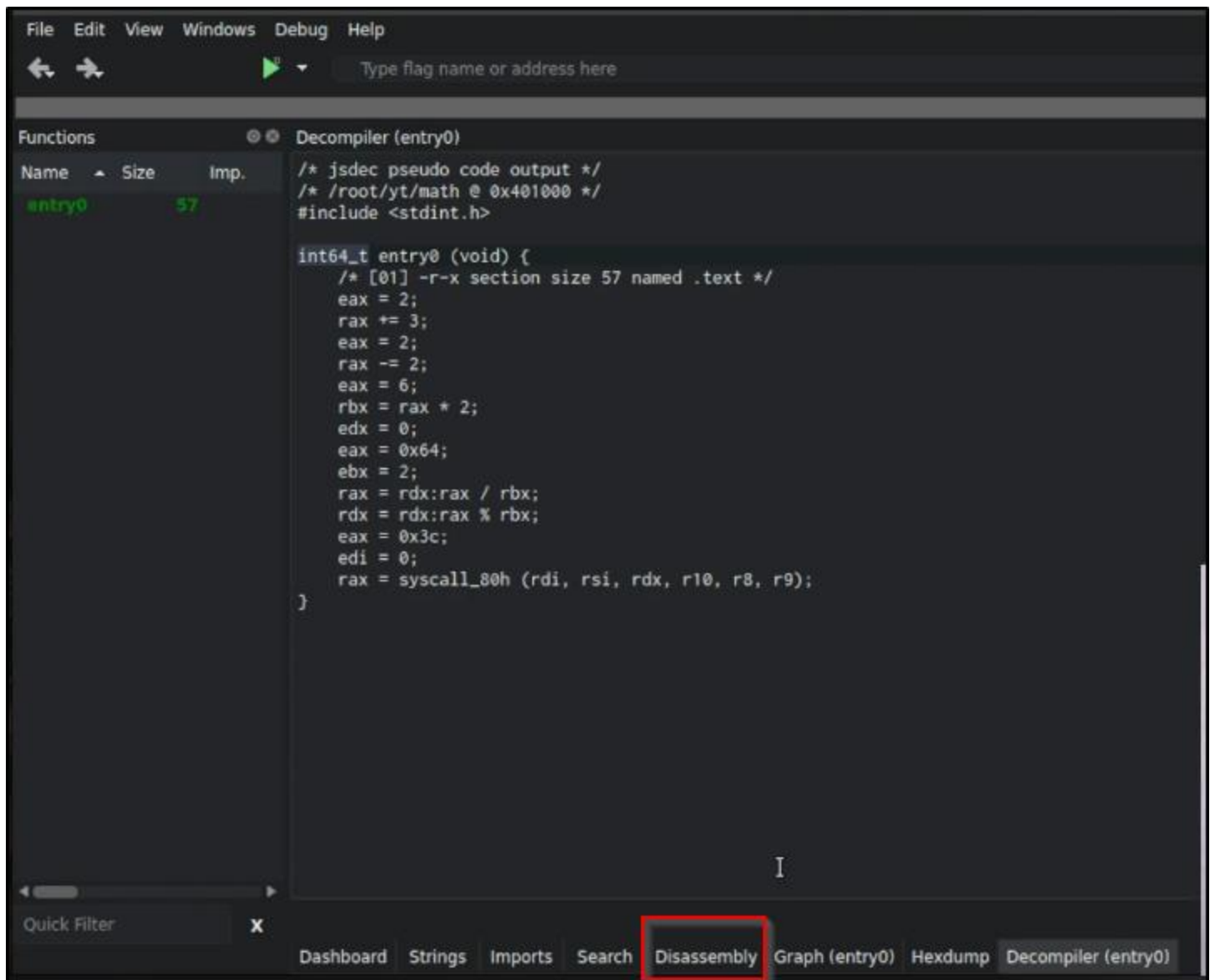


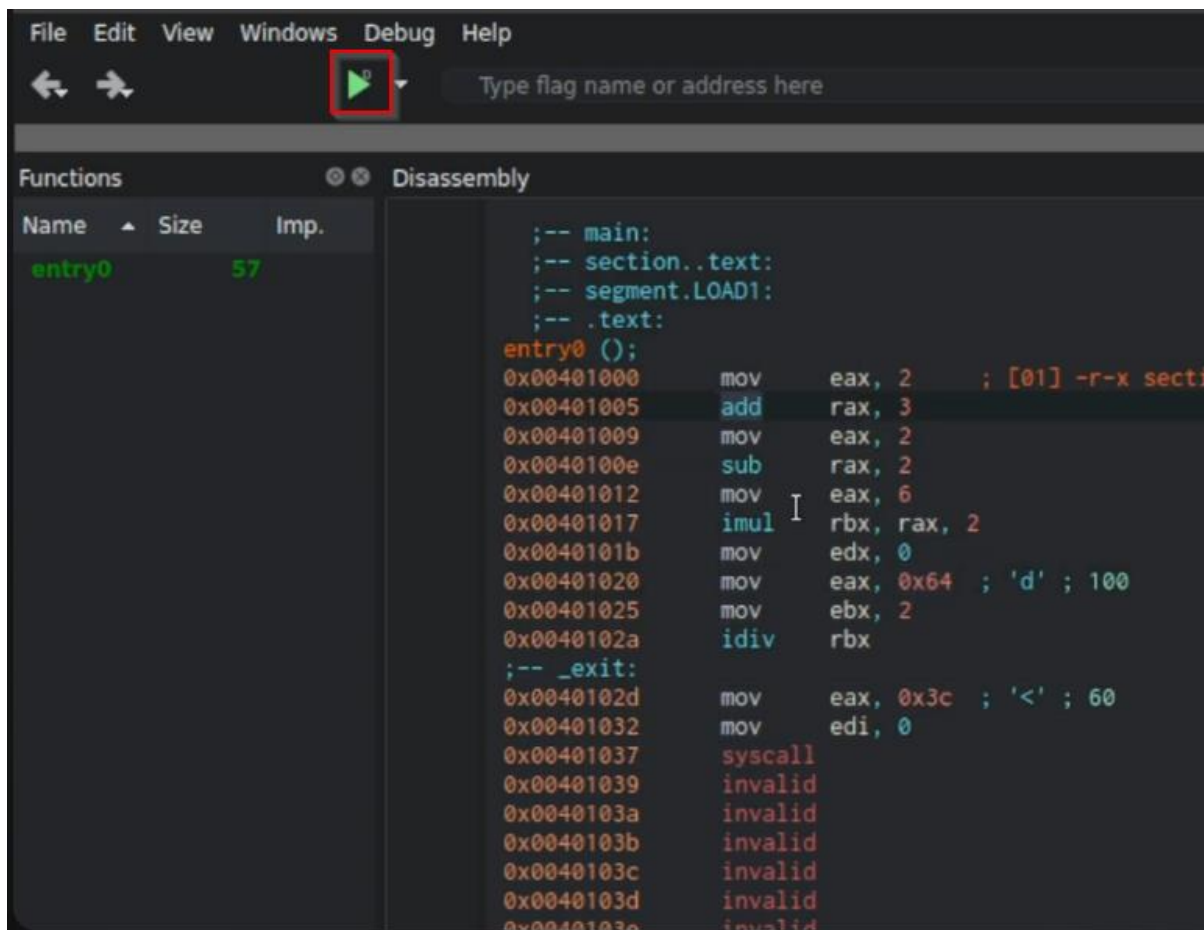
Click on select.

And select your binary file.



Click on ok





It will show popap then click ok no need to fill anything.

The screenshot shows a debugger interface with a menu bar (File, Edit, View, Windows, Debug, Help) and a toolbar. Below the toolbar is a search bar labeled "Type flag name or address here". The main window is titled "Disassembly" and shows a list of functions on the left, with "entry0" selected. The disassembly window displays the following code:

```
entry0 ();
0x00401000 mov     eax, 2      ; [01] -r-x section size 57 named .text
0x00401005 add     rax, 3
0x00401009 mov     eax, 2
0x0040100e sub     rax, 2
0x00401012 mov     eax, 6
0x00401017 imul    rbx, rax, 2
0x0040101b mov     edx, 0
0x00401020 mov     eax, 0x64 ; 'd' ; 100
0x00401025 mov     ebx, 2
0x0040102a idiv    rbx
0x0040102d mov     eax, 0x3c ; '<' ; 60
0x00401032 mov     edi, 0
0x00401037 syscall
0x00401039 add     byte [rax], al
0x0040103b add     byte [rax], al
0x0040103d add     byte [rax], al
0x0040103f add     byte [rax], al
0x00401041 add     byte [rax], al
0x00401043 add     byte [rax], al
0x00401045 add     byte [rax], al
0x00401047 add     byte [rax], al
0x00401049 add     byte [rax], al
0x0040104b add     byte [rax], al
0x0040104d add     byte [rax], al
0x0040104f add     byte [rax], al
0x00401051 add     byte [rax], al
0x00401053 add     byte [rax], al
0x00401055 add     byte [rax], al
0x00401057 add     byte [rax], al
0x00401059 add     byte [rax], al
0x0040105b add     byte [rbx], al
```

Here, we can see registers values.

```






entry0 ();
0x00401000    mov     eax, 2      ; [01] -r-x section size 57 named .text
0x00401005    add     rax, 3
0x00401009    mov     eax, 2
0x0040100e    sub     rax, 2
0x00401012    mov     eax, 6
0x00401017    imul    rbx, rax, 2
0x0040101b    mov     edx, 0
0x00401020    mov     eax, 0x64   ; 'd' ; 100
0x00401025    mov     ebx, 2
0x0040102a    idiv    rbx
0x0040102d    mov     eax, 0x3c   ; '<' ; 60
0x00401032    mov     edi, 0
0x00401037    syscall
0x00401039    add     byte [rax], al
0x0040103b    add     byte [rax], al
0x0040103d    add     byte [rax], al
0x0040103f    add     byte [rax], al
0x00401041    add     byte [rax], al
0x00401043    add     byte [rax], al
0x00401045    add     byte [rax], al
0x00401047    add     byte [rax], al
0x00401049    add     byte [rax], al
0x0040104b    add     byte [rax], al
0x0040104d    add     byte [rax], al
0x0040104f    add     byte [rax], al

```

rax	0x0	rsp	ffd966579a0
rbx	0x0	rbp	0x0
rcx	0x0	rip	0x401000
rdx	0x0	cs	0x33
r8	0x0	rflags	0x200
r9	0x0	orax	0x3b
r10	0x0	ss	0x2b
r11	0x0	fs	0x0
r12	0x0	gs	0x0
r13	0x0	ds	0x0
r14	0x0	es	0x0

Click on steps.

Debug Help

Type flag name or address here

```

sembly
entry0 ();
0x00401000    mov     eax, 2      ; [01] -r-x section size
0x00401005    add     rax, 3
0x00401009    mov     eax, 2
0x0040100e    sub     rax, 2
0x00401012    mov     eax, 6
0x00401017    imul    rbx, rax, 2
0x0040101b    mov     edx, 0
0x00401020    mov     eax, 0x64   ; 'd' ; 100
0x00401025    mov     ebx, 2
0x0040102a    idiv    rbx
0x0040102d    mov     eax, 0x3c   ; '<' ; 60
0x00401032    mov     edi, 0
0x00401037    syscall
0x00401039    add     byte [rax], al
0x0040103b    add     byte [rax], al
0x0040103d    add     byte [rax], al

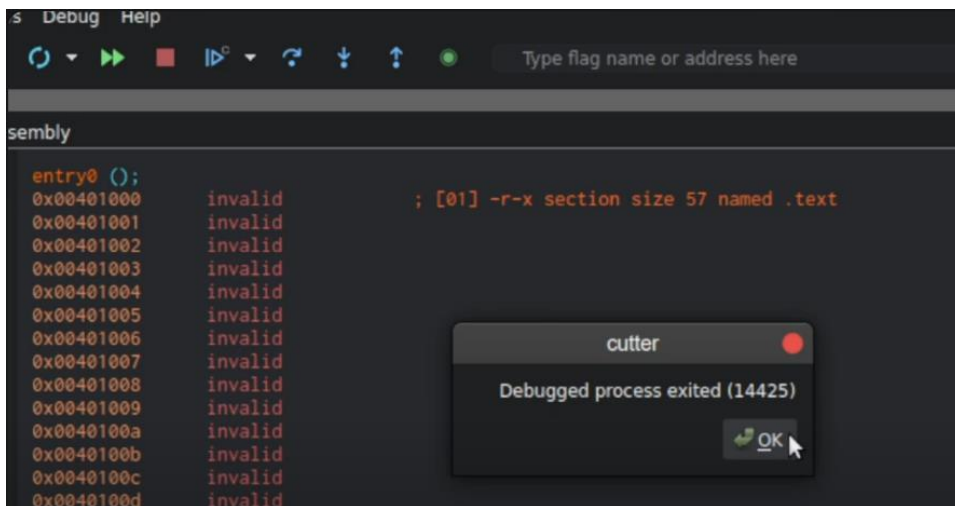
```

rax	0x2	rsp	ffd966579a0
rbx	0x0	rbp	0x0
rcx	0x0	rip	0x401005
rdx	0x0	cs	0x33
r8	0x0	rflags	0x202
r9	0x0	orax	ffffffff
r10	0x0	ss	0x2b
r11	0x0	fs	0x0
r12	0x0	gs	0x0

Here, we see step by step execution of program.

Red line is showing next instruction to be executed.

After finish program it will exit.



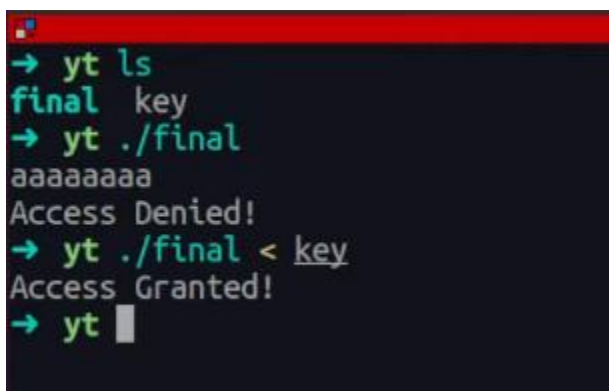
#####

Conditional Branching(If else program)

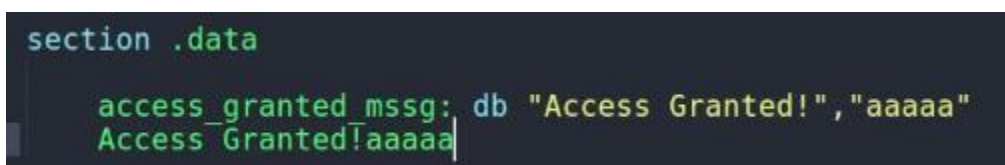
Unconditional jump

Conditional jump

We have to crate this program.



Assembly me agar aise likhe to ye niche ki line jaisa print kr dega.



To yha aaaa ki jagah hum new line print krna chahte hai. to new line ke liye koi character nhi bna to hum uska ascii value de denge.

Use man ascii command to get ascii table.

```
man ascii
```

To see ascii decimal table.

```
ascii -d
```

To see ascii decimal and hex table

```
ascii -t
```

```
global _start

section .text

_start:
    jmp main

main:

    mov rax, 0
    mov rdi, 0
    mov rsi, user_key
    mov rdx, 64
    syscall

; cmd_key:

access_granted:
    mov rax, 1
    mov rdi, 1
    mov rsi, access_granted_mssg
    mov rdx, access_granted_mssg_len
    syscall

access_denied:
    mov rax, 1
    mov rdi, 1
    mov rsi, access_denied_mssg
    mov rdx, access_denied_mssg_len
```

```

syscall

exiting:
    mov rax, 60
    mov rdi, 0
    syscall

section .data
    access_granted_mssg: db "Access Granted!",10    ; ascii value of new line
    access_granted_mssg_len: equ $ - access_granted_mssg
    access_denied_mssg: db "Access Denied!",10    ; ascii value of new line
    access_denied_mssg_len: equ $ - access_denied_mssg

section .bss
    user_key: resb 64

```

Assemble and link

Output

```

→ yt nasm -f elf64 condition.asm -o condition.o
→ yt ld condition.o -o condition
→ yt ./condition
aaaaaaa
Access Granted!
Access Denied
→ yt

```

Simple logic:

```

cmp_key:
    cmp rax,original_key_len
    jne access_denied
    mov rsi,original_key
    mov rdi,user_key
    cmpsb
    je access_granted
    jne access_denied

```

Here, **cmp** does not work with string it only work with numbers. The we have to use **cmpsb** instruction. But cmpsb doesn't take variable directly. We have to store in registers. And register should be **rsi and rdi**.

je means jump if equal.

jne means jump is not equal.

Let's run the program

```
→ yt ./condition
aaaa
Access Denied!
→ yt ./condition < key
Access Granted!
→ yt
```

Here, we can see that program working good. But it is not.

Find the length of key and genrate first character of key in equal to key length.

```
→ yt python3
Python 3.8.5 (default, May 27 2021, 13:30:53)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> len('1789-7654-0987-4532')
19
>>> '1'*19
'1111111111111111111'
>>>
```

```
→ yt echo -n '1111111111111111111' | ./condition
Access Granted!
→ yt
```

Here, we can see that wrong key is also access granted.

Yha cmpsb sirf first byte ko hi compare karta hai.

Uske liye ek instruction hota hai **repe** means repeat.


```

cmp_key:
    cmp rax,original_key_len
    jne access_denied
    mov rsi,original_key
    mov rdi,user_key
    mov rcx,original_key_len
    repe cmpsb
    je access_granted
    jne access_denied

```

Yha pr hum ise batayenge ki kitni bar repeat karo use **rcx** register me hi store karenge.

To hume use bataya ki **original_key_len** bar repeat karo.

Yha pr ye repeat aise krta hai ye repeat kiye hue ko remove krke bache hue key ko **rdi** me store karega fir aage repeat krega. Otherwise ye **1** ko hi repeat krte rh jyega.

Final code:

```

global _start

section .text

_start:
    jmp main

main:

    mov rax, 0
    mov rdi, 0
    mov rsi, user_key
    mov rdx, 64
    syscall

cmd_key:
    cmp rax, original_key_len
    jne access_denied
    mov rsi, original_key
    mov rdi, user_key
    mov rcx, original_key_len

```



```

    repe cmpsb
    je access_granted
    jne access_denied

access_granted:
    mov rax, 1
    mov rdi, 1
    mov rsi, access_granted_mssg
    mov rdx, access_granted_mssg_len
    syscall
    jmp exiting

access_denied:
    mov rax, 1
    mov rdi, 1
    mov rsi, access_denied_mssg
    mov rdx, access_denied_mssg_len
    syscall

exiting:
    mov rax, 60
    mov rdi, 0
    syscall

section .data
    access_granted_mssg: db "Access Granted!",10    ; ascii value of
new line
    access_granted_mssg_len: equ $ - access_granted_mssg
    access_denied_mssg: db "Access Denied!",10    ; ascii value of
new line
    access_denied_mssg_len: equ $ - access_denied_mssg
    original_key: db "1289-3423-2323-9898"
    original_key_len: equ $ - original_key

section .bss
    user_key: resb 64

```

Output

```

→ yt ./condition
1789-7654-0987-4532
Access Denied!
→ yt echo 1789-7654-0987-4532 | ./condition
Access Denied!
→ yt man echo
→ yt echo -n 1789-7654-0987-4532 | ./condition
Access Granted!
→ yt

```

Here, above we can see that correct key entered but access denied. Because key ke sath new line bhi gya. Agar hum **echo** kare to wo bhi new line add kr deta hai. use remove krne ke liye hum **echo -n** use krte hai.

#####

Loop in Assembly

Final output will be.

To print table of any number.

```

→ yt ./loop
Enter Value : 2
2
4
6
8
10
12
14
16
18
20

```

```

→ yt ./loop
Enter Value : 20
20
40
60
80
100
120
140
160
180
200

```

```

section .data
    mssg: db "Enter Number : ",10,0
    mssg1: db "hello"

```

Generally, assembler consider both string as single string . to Yha 10 ke bad 0 hai wo null byte hai yh niche ke string ko differentiate krne ke kam aata hai.

Agar aap glti se bhi length me ek extra de diya to aage ka string print kr dega.

Null byte assembler ko batayega ki aage ka string iska part nhi hai ise add mt karo.

Assembler can't print ouput of integer or take input of integer. Only take or print string as input or output.

Assembly can't read and write numbers.

Here, we will not write whole code, we use **util.asm** nasm library readymade code.

Go to github link.

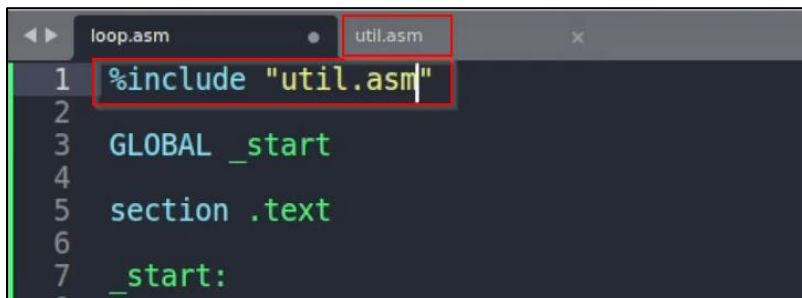
<https://github.com/mjbrusso/util.asm>

<https://raw.githubusercontent.com/mjbrusso/util.asm/refs/heads/master/util.asm>

donwload only **util.asm** file.

How to include external file.

Keep both file in same directory.

A screenshot of a code editor with two tabs: 'loop.asm' and 'util.asm'. The 'loop.asm' tab is active, showing assembly code. Line 1 contains '%include "util.asm"', which is highlighted with a red box. Below it, lines 2 through 7 show 'GLOBAL _start', 'section .text', and '_start:'.

```
1 %include "util.asm"
2
3 GLOBAL _start
4
5 section .text
6
7 _start:
```

In the current version (1.2) the following functions are available:

Function	Description
exit	If you want different exit status code then you use it.
exit0	Exit program code with exit status code 0
strlen	Get string length
itoa	Number to string convertor

atoi	String to number convertor
endl	Print new line
printstr	Write a string
printint	Write an integer
readstr	Read a string
readint	Read an interger

You can **call** any function from **util.asm** using call instruction.

```

;*****
printstr:
    push    r15                ; r15 is callee saved
    mov     r15, rdi           ; save copy (rdi should be caller saved)
    call    strlen
    mov     rdx, rax            ; string size
    mov     rsi, r15            ; string
    mov     rax, SYS_WRITE      ; system call number
    mov     rdi, STDOUT         ; file descriptor
    syscall                      ; system call
    pop     r15
    ret
;*****

```

Write in assembly using printstr

```

;*****
void printstr(char *s)

Description:
    Print a string

Arguments:
    rdi: char *s: address of a null-terminated string

Returns:
    Nothing
;*****

```

Yha hum dekh skte hai ki kaise use krna hai.

Yha **rdi** register ke andar variable ka address pass krna hai joki **null-terminated** se end hona chahiye.

Readint function

To take user input into integer.

```
; int64 readint()
;
; Description:
;   Read int64 from standard input
;
; Arguments:
;   None
;
; Returns:
;   rax: int64: The value entered
```

And return **entered value** in **int64** in rax register. Then have to store it for further use.

```
readint:
    sub     rsp, 40
    mov     rdi, rsp
    mov     rsi, 21
    call    readstr
    mov     rdi, rsp
    call    atoi
    add     rsp, 40
    ret
```

Loop.asm

```
_start:

    mov     rdi, mssg
    call    printstr
    call    readint
    mov     [user_value], rax

section .data
    mssg: db "Enter Number : ", 10, 0

section .bss
    user_value: resb 8

    0x44444444
    user_value --> -----
```

Here, we created a variable **user_value** reserve **8 bytes**.

Here, we have used `[user_value]` in `[]` because it point blank space in not **address**.

```
loop.asm  util.asm x
1  %include "util.asm"
2
3  GLOBAL _start
4
5  section .text
6
```

```
7  _start:
8
9      mov rdi,mssg
10     call printstr
11     call readint ;2
12     mov [user_value],rax
13     mov rbx,1
14
15     LOOP_START:
16
17     mov rcx,[user_value]
18     imul rcx,rbx
19     mov rdi,rcx
20     call printint
21     call endl
22     add rbx,1
23     cmp rbx,11
24     jne LOOP_START
25     call exit0
26
27     section .data
28
29     mssg: db "Enter Number : ",0
30
31     section .bss
32
33     user_value: resb 8 ;2 | I
34
```

Output

```

→ yt subl util.asm
→ yt nasm -f elf64 loop.asm -o loop.o
→ yt ld loop.o -o lop
→ yt ./lop
Enter Number : 2
2
4
6
8
10
12
14
16
18
20
→ yt

```

Why here we used square bracket.

```

LOOP_START:
    mov rcx, [user_value]
    imul rcx, rcx
    mov rdi, rcx
    call printint
    call endl
    add rbx, 1
    cmp rbx, 11
    jne LOOP_START
    call exit 0

section .data
    mssg: db "Enter Number : ", 0

section .bss
    0x444444 -> 2
    user_value: resb 8

```

user_value variable ka **address 0x444444** jo hai wo **2** ko point kr rha hai. wha 2 store hai.

Yha pr hum square bracket isliye lagaye kyoki hum 2 ko get krna chahte hai. agar nhi lagate to hume address milta.

Aur kai jagah hume address pass krna hota hai. jaise ki

```

_start:
    mov rdi, mssg
    call printstr
    call readint
    mov [user_value], rax
    mov rbx, 1

LOOP_START:
    mov rcx, [user_value]
    imul rcx, rbx
    mov rdi, rcx
    call printint
    call endl
    add rbx, 1
    cmp rbx, 11
    jne LOOP_START
    call exit0

section .data
    0x5555 ->
    mssg: db "Enter Number : ", 0

```

Yha hum util.asm file me dekh skta hai yha humse address manga ja rha hai.

```

*****
; void printstr(char *s)
;
; Description:
;   Print a string
;
; Arguments:
;   rdi: char *s: address of a null-terminated string (array of chars terminated by 0)
;
; Returns:
;   Nothing
;
*****
printstr:

```

#####

Final Project Calculator:

Output will be


```
→ yt ./final_calculator
Enter First Number : 10
Enter Second Number : 5
Enter Operator To Use(+,-,*,/) : +
15
→ yt ./final_calculator
Enter First Number : 20
Enter Second Number : 70
Enter Operator To Use(+,-,*,/) : *
1400
```

```
→ yt ./final_calculator
Enter First Number : 20
Enter Second Number : 10
Enter Operator To Use(+,-,*,/) : -
10
→ yt ./final_calculator
Enter First Number : 20
Enter Second Number : 40
Enter Operator To Use(+,-,*,/) : %
Cannot perform this operation.
```

Let's start

calculator.asm

```

calculator.asm  util.asm
1  %include "util.asm"
2
3  GLOBAL _start
4
5  section .text
6
7  _start:
8
9      mov rdi,num1
10     call printstr
11     call readint
12     mov [user_num1],rax
13     mov rdi,num2
14     call printstr
15     call readint
16     mov [user_num2],rax
17     mov rdi,operators
18     call printstr
19     mov rdi,user_operator
20     mov rsi,2
21     call readstr
22

```

Here, we label can be called function. And we can call it.

```

cmp_operators:
    mov rdi,[user_operator]
    cmp rdi,43 ; +
    je addition
    cmp rdi,45 ; -
    je subtraction
    cmp rdi,42 ; *
    je multiply
    cmp rdi,47 ; /
    je division

```

```

exception:
    mov rdi,error_mssg
    call printstr
    call endl
    call exit0

```

addition:

```
mov rdi,[user_num1]
add rdi,[user_num2]
call results
```

subtraction:

```
mov rdi,[user_num1]
sub rdi,[user_num2]
call results
```

multiply:

```
mov rdi,[user_num1]
imul rdi,[user_num2]
call results
```

division:

```
mov rdx,0
mov rax,[user_num1]
mov rbx,[user_num2]
idiv rbx
mov rdi,rax
call results
```

results:

```
call printint
call endl
call exit0
```

section .data

```
num1: db "Enter Number 1 : ",0
num2: db "Enter Number 2 : ",0
operators: db "Enter operation to perform(+,-,*,/) : ",0
error_mssg: db "Cannot perform this operation."
I
```

section .bss

```
user_num1: resb 8
user_num2: resb 8
user_operator: resb 2
```

Final Ouput

```
→ yt nasm -f elf64 calculator.asm -o calculator.o
→ yt ld calculator.o -o calculator
→ yt ./calculator
Enter Number 1 : 20
Enter Number 2 : 10
Enter operation to perform(+,-,*,/) : -
10
→ yt ./calculator
Enter Number 1 : 50
Enter Number 2 : 30
Enter operation to perform(+,-,*,/) : *
1500
→ yt ./calculator
Enter Number 1 : 20
Enter Number 2 : 40
Enter operation to perform(+,-,*,/) : $
Cannot perform this operation.␣
```

So, finally program working well.

References:

https://www.youtube.com/watch?v=Nv-GTg3uICE&list=PL-DxAN1jsRa-3KzeQeEeoL_XpUHKfPL1u

<https://www.youtube.com/watch?v=BpOV3G1-m0&list=PLAZj-jE2acZLdYT7HLFgNph190z2cjmAG>

<https://www.youtube.com/watch?v=SL--goiu7yA&list=PLR2FqYUVaFJpHPw1ExSVJZFNIXzJYGAT1>

<https://www.youtube.com/watch?v=rwtTIJMjNnM&list=PLGqyrbzgTfmRnYtSCvbBB0nHqMoop4z45&index=4>

<https://www.youtube.com/watch?v=Ehy4Cnx4Xr8&list=PL3SAXYUEnrabDbKeOiJnLDRRWzIFg9pIn>

<https://www.youtube.com/watch?v=8PML0rmIgyM&list=PLm9FYbXgpdMryPIvZKM-LXVEzOuIzySSP&index=19>

<https://www.youtube.com/watch?v=7xiPJPzcGM&list=PLduM7bkxBdOczQDpzp3R9ieJRpjZrcxj>

https://www.youtube.com/watch?v=LfXahoBkz5Y&list=PLgWOIdHQBz5t_8v6eZXxKLTV8SGj5kN&index=12

<https://youtu.be/gfmRrPjnEw4?si=nq9MMZiPWkxIi5qZ>

https://www.youtube.com/watch?v=P5JOlz7MeYg&list=PLsu3nQRCX0RIjLN0eUNzQbDFyfz8E_DJi&index=2

=====

https://www.youtube.com/watch?v=Nv-GTg3uICE&list=PL-DxAN1jsRa-3KzeQeEeoL_XpUHKfPL1u

<https://www.youtube.com/watch?v=BpOV3G1-m0&list=PLAZj-jE2acZLdYT7HLFgNph190z2cjmAG>

<https://www.youtube.com/watch?v=SL--goiu7yA&list=PLR2FqYUVaFJpHPw1ExSVJZFNIXzJYGAT1>

<https://www.youtube.com/watch?v=rwtTIJMjNnM&list=PLGqyrbzgTfmRnYtSCvbBB0nHqMoop4z45&index=4>

<https://www.youtube.com/watch?v=Ehy4Cnx4Xr8&list=PL3SAXYUEnrabDbKeOiJnLDRRWzIFg9pIn>

<https://www.youtube.com/watch?v=8PML0rmIgyM&list=PLm9FYbXgpdMryPIvZKM-LXVEzOuIzySSP&index=19>

<https://www.youtube.com/watch?v=7xiPJVPzcGM&list=PLduM7bkxBdOczQDpzp3R9ieJRpjZrcxj>

https://www.youtube.com/watch?v=LfXahobkz5Y&list=PLgWOIdHQBEz5t_8v6eZXxKLT_V8SGj5kN&index=12

<https://youtu.be/gfmRrPjnEw4?si=nq9MMZiPWkxIi5qZ>

https://www.youtube.com/watch?v=P5JOlz7MeYg&list=PLsu3nQRCX0RIjLN0eUNzQbDFyfz8E_DJi&index=2