C code to Byte Code Example

```c
#include <stdio.h>

char s[] = "Hello World";

int main()
{
        int x = 2000, z = 21;
        printf("%s %d /n", s, x+z);
}
```

Assembly Output with:

`gcc -S helloworld_for_assembly.c`

```
        .file   "helloworld_for_assembly.c"
        .text
        .globl  s
        .data
        .align 8
        .type   s, @object
        .size   s, 12
s:
        .string "Hello World"
        .section        .rodata
.LC0:
        .string "%s %d /n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $2000, -8(%rbp)
        movl    $21, -4(%rbp)
```

```
        movl    -8(%rbp), %edx
        movl    -4(%rbp), %eax
        addl    %edx, %eax
        movl    %eax, %edx
        leaq    s(%rip), %rax
        movq    %rax, %rsi
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string "GNU"
1:
        .align 8
        .long   0xc0000002
        .long   3f - 2f
2:
        .long   0x3
3:
        .align 8
4:
```

Identifying Parts in Assembly

data section declares initialized data
bss section is used to declare variables
text section is where the actual code is

64-bit register	*Lowest 32-bits*	*Lowest 16-bits*	*Lowest 8-bits*
rax	eax	ax	al

64-bit register	*Lowest 32-bits*	*Lowest 16-bits*	*Lowest 8-bits*
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

General Purpose Registers

• ***AX***: The accumulator is designated by AX. This register consists of 16 bits, which is further split into registers such as AH and AL, which are 8 bits each. This split enables the AX register to process 8-bit instructions as well. You will find this register involved in arithmetic and logic operations.

• ***BX***: The base register is designated by BX. This 16-bit register is also split into two 8-bit registers, which are BH and BL. The BX register is leveraged to keep track of an offset value.

• ***CX***: The counter register is designated by CX. CX is split into CH and CL, which are 8 bits each. This register is involved in the looping and rotation of data.

• ***DX***: The data register is designated by DX. This register also contains two 8-bit registers, which are DH and DL. The function of this register is to address input and output functions.

Pointer Registers

• ***SP***: This stands for stack pointer. It has a bit size of 16 bits. It indicates the stack's highest item. The stack pointer will be (FFFEH) if the stack is empty. It's a relative offset address to the stack section.

• ***BP***: The base pointer is denoted by the letters BP. It has a bit size of 16 bits. It is mostly used to access stack-passed arguments. It's a relative offset address to the stack section.

• ***IP***: This determines the address of the next instruction that will be executed. The whole address of the current instruction in the code segment is given by IP in conjunction with the code segment (CS) register as (CS: IP).

Index Registers

• ***Source Index (SI)***: This is the register for the source index. It has a bit size of 16 bits. It's utilized for data pointer addressing and as a source for various string operations. It has a relative offset to the data segment.

• ***Destination Index (DI)***: This is the register used for the destination index.

Control Registers

• ***Overflow Flag (OF)***: Once a signed arithmetic operation completes, it signifies the overflow of a higher-order bit (which will be the leftmost bit) of data.

• ***Direction Flag (DF)***: This determines whether to move or compare string data in the left or right direction. When the DF value is 0, the string operation is performed left to right, and when the value is 1, the string operation is performed right to left.

• ***Interrupt Flag (IF)***: This specifies whether external interrupts, such as the input of a keyboard, should be ignored or processed. When set to 0, it inhibits external interrupts, and when set to 1, it enables them.

• ***Trap Flag (TF)***: This allows you to set the CPU to operate in single-step mode. The TF is set by the DEBUG program, which allows us to walk through the execution. This walk-through is executed as per instructions.

• ***Sign Flag (SF)***: This displays the sign of the result of an arithmetic operation. Following an arithmetic operation, this flag is set based on the sign of the data item. The most significant bit of the leftmost bit indicates the sign. A positive result resets the SF value to 0, and a negative result resets it to 1.

• ***Zero Flag (ZF)***: This denotes the outcome of a calculation or a comparison. When a result is equal to non-zero, this will result in the ZF being set to 0; conversely, when a result is zero, then the ZF will be set to 1.

• ***Auxiliary Carry Flag (AF)***: When it comes to binary coded decimal operations, or BCD as it is abbreviated, the auxiliary carry flag would come into play. It is related to math operations and is set when there is a carry from a lower bit to a higher bit, for example, from bit 3 to bit 4.

• ***Parity Flag (PF)***: When an arithmetic operation takes place and the resulting bits are even, then the parity flag gets set. If the result is not even, the parity flag will be set to 0.

• ***Carry Flag (CF)***: Upon completion of an arithmetic operation, the CF reflects the carry of 0 or 1 from a high-order bit (leftmost). It also saves the contents of a shift or rotates the operation's last bit.

Segment Registers

• ***Code Segment***: This covers all of the directions that must be carried out. The CS register is used to store the starting address of the code segment.

• ***Data Segment***: Data, constants, and work areas are all included. The DS register is used to store the starting address of the data segment.

• ***Stack Segment***: This contains information on procedures and subroutines, as well as their return addresses. The implementation of this is in the form of a data structure known as a stack. The stack's starting address is stored in the stack segment register, or SS register.

Debugging /bin/bash with gdb

```
┌──(kali㉿kali)-[~]
└─$ gdb /bin/bash
GNU gdb (Debian 10.1-2) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /bin/bash...
(No debugging symbols found in /bin/bash)
(gdb) break main
Breakpoint 1 at 0x2ee90
(gdb) run
Starting program: /usr/bin/bash

Breakpoint 1, 0x0000555555582e90 in main ()
(gdb) 
```

Getting Information from the Registers

```
(gdb) info registers
rax            0x555555582e90      93824992423568
rbx            0x0                 0
rcx            0x7ffff7f75738      140737353570104
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x55555563c1a0      0x55555563c1a0 <__libc_csu_init>
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe22f0      140737354015472
r10            0x69682ac           110527148
r11            0x202               514
r12            0x555555584670      93824992429680
r13            0x0                 0
r14            0x0                 0
r15            0x0                 0
rip            0x555555582e90      0x555555582e90 <main>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) 
```

RAX

```
(gdb) display /x $rax
8: /x $rax = 0x555555585340
(gdb) display /x $eax
9: /x $eax = 0x55585340
(gdb) display /x $ax
10: /x $ax = 0x5340
(gdb) display /x $ah
11: /x $ah = 0x53
(gdb) display /x $al
12: /x $al = 0x40
(gdb) 
```

Data Movement Instructions

• ***MOV***: This is a command that moves data from one operand to another. This data can be in the form of a byte, word, or even a double word. Any of these pathways can be used with the MOV instruction to transfer data. There are also MOV variations that work with segment registers.

• ***MOVS*** for memory to memory ***MOV*** operations

• ***XCHG*** exchange content between two operands

Stack Manipulation Instructions

• ***POP***: This transfers a value that is currently at the top of the stack to a destination operand. Once this is done, the ESP register is incremented to point to the new stack value. POP can also be used with segment registers.

• ***POPA***: POPA means to pop all registers. This instruction is used to restore the general-purpose registers. POPA on its own is a 16-bit register. This means that the first register to be popped would be DI, followed by SI, BP, BX, DX, CX, and AX. POPAD, on the other hand, is a 32-bit register. Essentially, POPAD is referring to a double word, so in this case, the first register to be popped would be EDI, followed by ESI, EBP, EBX, EDX, ECX, and EAX.

• ***PUSHA***: PUSHA, which means push all registers, saves the contents of the stacks registers. The POPA instruction is used in conjunction with PUSHA, and the same applies to PUSHAD in relation to POPAD.

• ***PUSH***: PUSH is commonly used to store parameters on the stack; it is also the primary method of storing temporary variables on the stack. Memory operands, immediate operands, and register operands are all affected by the PUSH instruction (including segment registers).

Arithmetic Instructions

- Addition (***add***)
- Subtraction (***sub***)
- Division (***div***)
- Multiplication (***mul***)

Conditional Instructions

Conditional Jump

Instruction	Description	Flags
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump Not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater/Equal or Jump Not Less	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

Unconditional Jump

• A ***short*** jump is a 2-byte instruction that allows access or jumps to memory locations that are defined within a certain memory byte range. This memory byte range is 127 bytes ahead of the jump or 128 bytes behind the jump instruction.

• A ***near*** jump is a 3-byte jump that allows access to +/- 32K bytes from the jump instruction.

• A ***far*** jump works with a specified code segment. In the case of a far jump, the value is absolute, meaning that the instruction will jump to a defined instruction.