ASSEMBLY TUTORIAL

x86-64 instructions

Data Transfer Instructions

mov S D: This instruction moves data from source(Src) to destination(Dest)

```
Source and destination can be: 1) Memory(Mem) 2) Register(Reg) 3) Immediate
Value(Imm) Following 6 instructions are included in x86-64 insstruction set:
| Source | Destination | Instruction | C Analog | |--------|-------------|--
---------------|--------- | | Imm | Reg | mov $0x2,%rax | temp=0x2 | | Imm |
Mem | mov $-115,(%rax)| *p=-115 | | Reg | Reg | mov %rax,%rdx | temp1=temp2|
| Reg | Mem | mov %rax,(%rdx) | *p=temp | | Mem | Reg | mov (%rax),%rdx |
temp=*p |
```

Arithmetic and Logical operators

Two operand instructions:

- add Src, Dest Dest = Dest + Src
- sub Src, Dest Dest = Dest - Src
- imul Src, Dest Dest = Dest*Src
- sal Src, Dest Dest = Dest << Src (Also called shl)
- sar Src, Dest Dest = Dest >> Src (Arithmetic)
- shr Src, Dest Dest = Dest >> Src (Logical)

- For sal, sar, shl and shr instructions Src must be either an Immediate value or byte register %cl.

- xor Src, Dest Dest = Dest ^ Src
- and Src, Dest Dest = Dest & Src
- or Src, Dest Dest = Dest | Src

Single operand instructions:

- inc Dest Dest = Dest + 1
- dec Dest Dest = Dest - 1
- neg Dest Dest = -Dest
- not Dest Dest = ~Dest

This embedded cheat-sheet is extremely useful for reference,

Using Assembly Code for some Functions

- In the coming assignments, you will be required to implement specific functions using assembly code.
- To do this, you will first create a wrapper C file.
- This is just a normal C file, where the only difference is that instead of defining the required function in C code, you will just leave the function at declaration.
- You must then create a separate .s file, where you will write the function in Assembly code.
- During compilation, you will compile the C file along with the .s file. This ensures that when the function is called in the C file, the definition written in the .s file is used.

Here is an example of code to add, subtract or multiply two numbers.

The wrapper C code is like this :

```
#include <stdio.h> long long int add(long long int a, long long int b); long long int subtract(long long int a, long long int b); long long int multiply(long long int a, long long int b); long long int isgreaterthan10(long long int a); long long int shift(long long int num, long long int sh); int main() { long long int x, y; printf("Enter two numbers 'x' and 'y' for operations : "); scanf("%lld %lld", &x, &y); long long int ans = add(x, y); printf("x + y = %lld\n", ans); ans = subtract(x, y); printf("x - y = %lld\n", ans); ans = multiply(x, y); printf("x * y = %lld\n", ans); ans = isgreaterthan10(x); printf("x > 10 = %lld\n", ans); ans = isgreaterthan10(y); printf("y > 10 = %lld\n", ans); printf("Enter the number to be shifted : "); scanf("%lld", &x); printf("Enter the shift : "); scanf("%lld", &y); ans = shift(x, y); printf("%lld << %lld is : %lld\n", x, y, ans); return 0; }
```

Here, we can see that the definitions for the add, subtract and multiply functions is not present.

The code for these functions is written in operations.s :

```
.global add .global subtract .global multiply .global isgreaterthan10
.global shift .text # generally, the first argument is in %rdi, the second
in %rsi, and the result is in %rax add: mov %rdi, %rax # copy the first
argument into the result (i.e rax = rdi) add %rsi, %rax # add the second
argument to the result (i.e rax = rax + rsi) ret subtract: mov %rdi, %rax #
copy the first argument into the result (i.e rax = rdi) sub %rsi, %rax #
subtract the second argument from the result (i.e rax = rax - rsi) ret
multiply: mov %rdi, %rax # copy the first argument into the result (i.e rax
= rdi) imul %rsi, %rax # multiply the second argument by the result (i.e rax
= rax * rsi) # mul %rsi ret isgreaterthan10: cmp $10, %rdi # compare the
first argument to 10 jg .yes # if the first argument is greater than 10,
jump to .yes mov $0, %rax # otherwise, set the result to 0 ret .yes: mov $1,
%rax # if the first argument is greater than 10, set the result to 1 ret
shift: mov %rsi, %rcx # rcx = rsi sal %cl, %rdi # rdi = rdi << cl mov %rdi,
%rax # rax = rdi ret # return
```

- Functions to be called by the C file must be declared global. Other routines need not be declared global like these.
- As mentioned in the doc, the arguments to a function are stored in some set registers by default. %rdi and %rsi store the first two arguments (useful here).
- The return value is always stored in %rax.

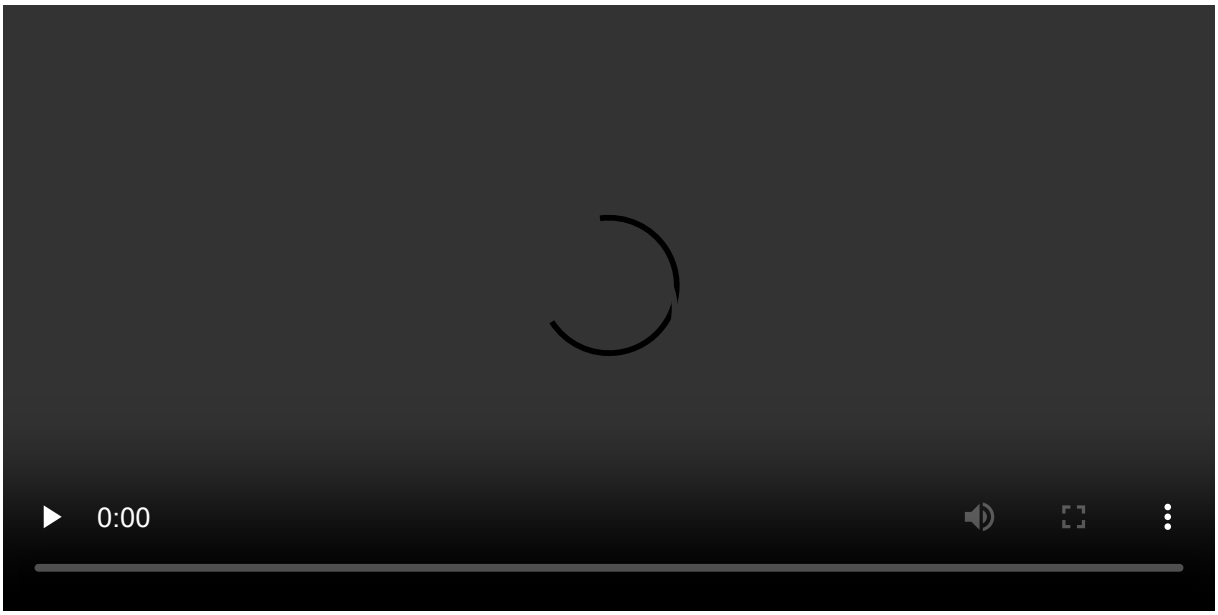
Compile and run :

```
-> gcc main.c operations.s -> ./a.out
```

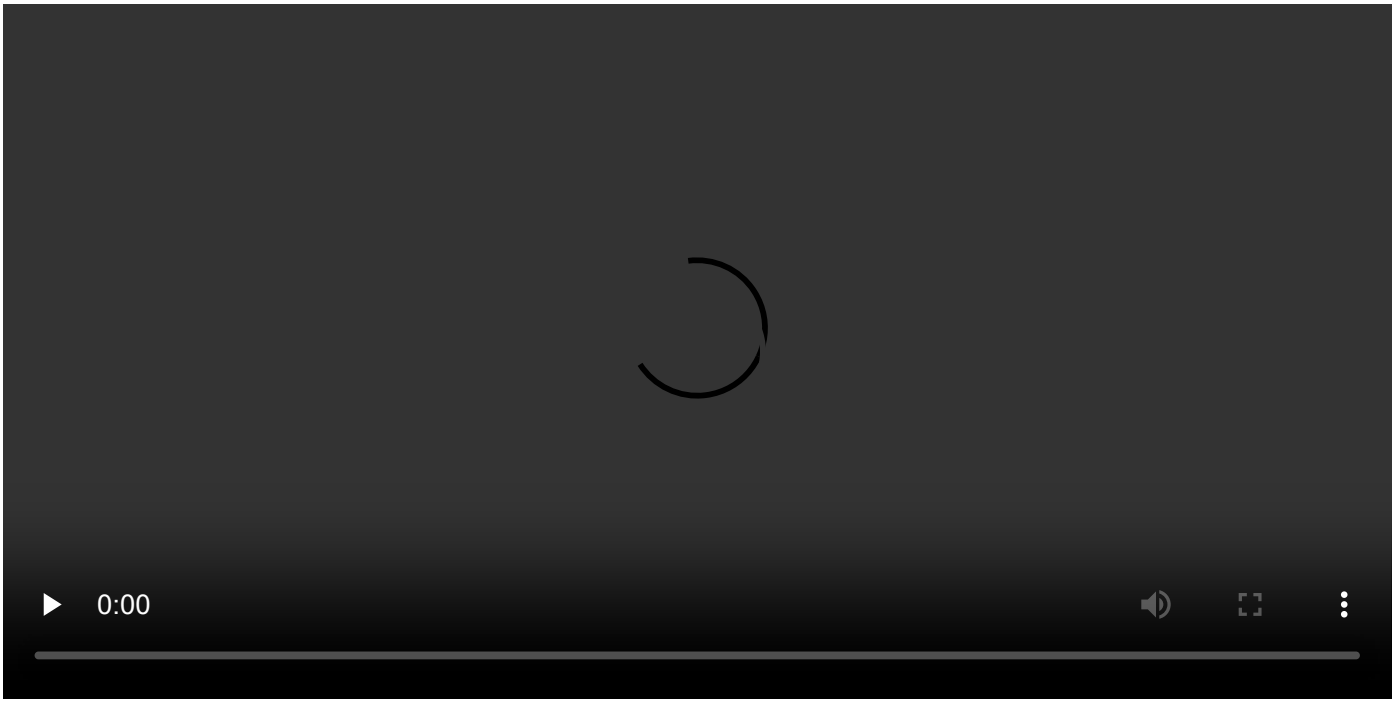
Debugging Assembly code using VS Code

Steps to run the debugger

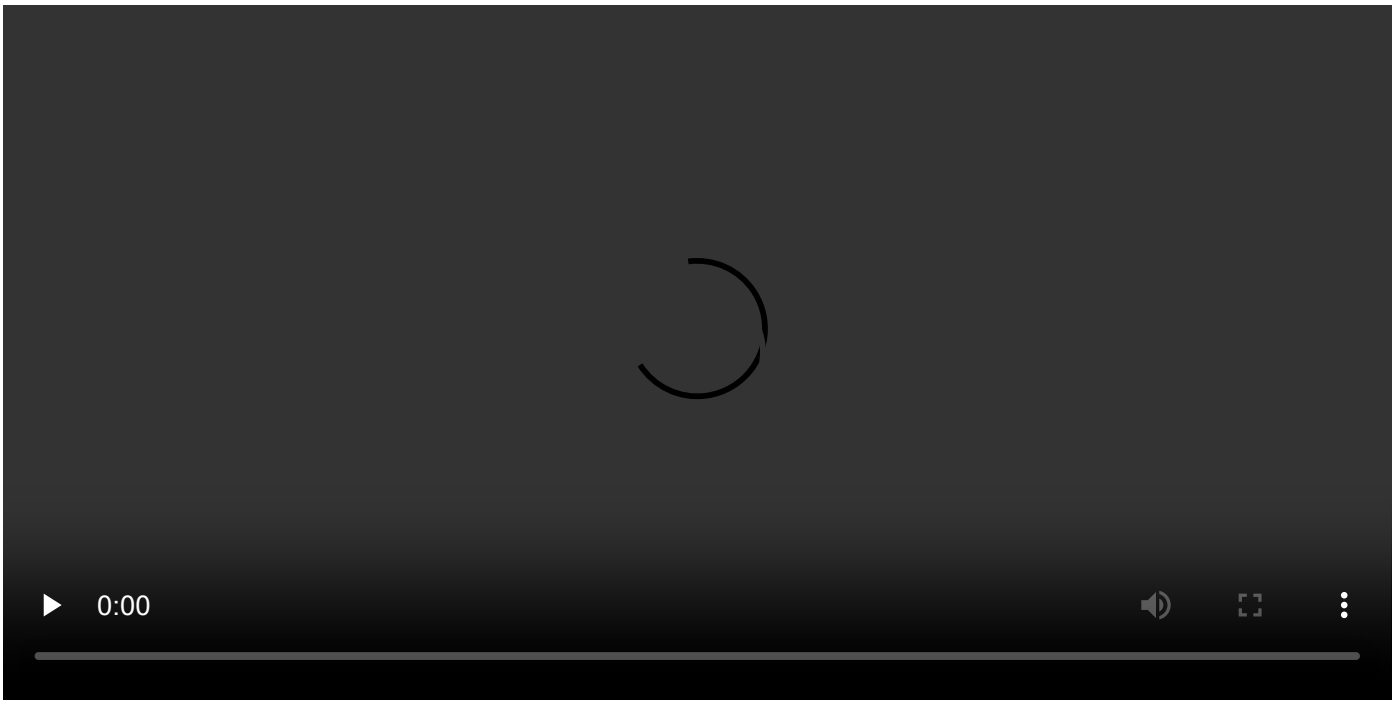
Have the C file as your current active file in VS Code, and use the UI to select ‘Run and Debug’. The first time, it might give you an error, just as in this video.



This is because, the definition of the functions defined in main.c are present in operations.s, so it must also be compiled along with it. Once you fail, you will find a file called tasks.json in a folder called .vscode in your directory (the directory opened in vs code). We need to add the path to the .s file to the command used by the VS Code debugger to compile the code, as shown in the video.



Now, trying to run the debugger shouldn't give any problems! You can place breakpoints where you call the function in the C file, and then use 'step into', to go to the line-by-line code in the .s file. You can also look at the current values of all the registers as shown in the video.



Note : If you are unable to find '.vscode' folder, then try opening the required directory in VS Code using File → Open Folder , and select the folder with the required files. Repeat the steps and you'll see the .vscode folder.