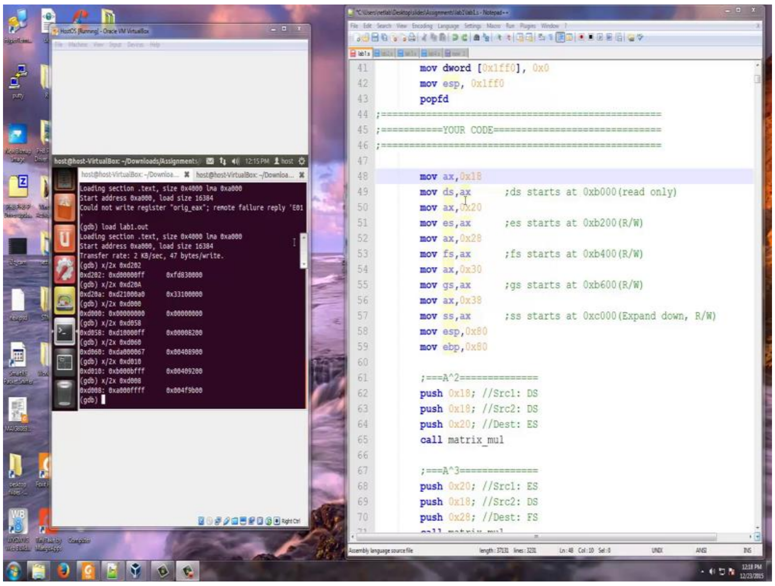
Information Security – II
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 23
Lab1 Part 3 - Week 4

(Refer Slide Time: 00:09)

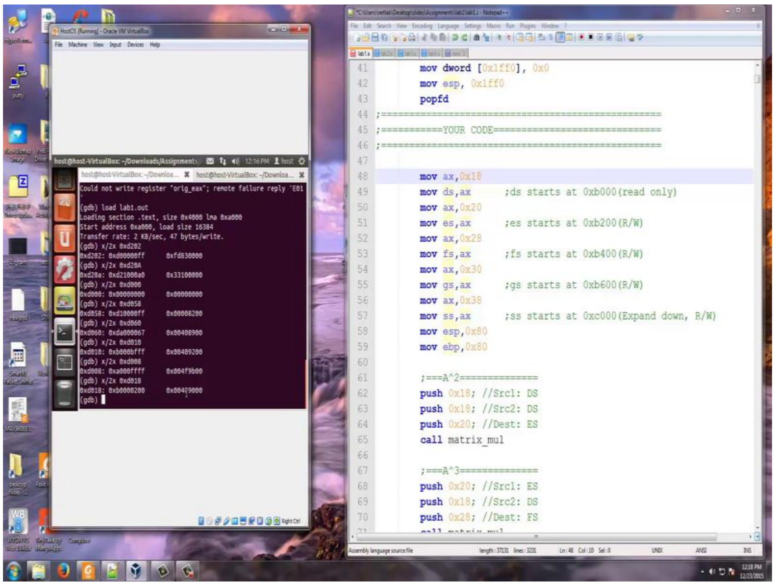


```
Loading section .text, size 0x4000 lma 0xa000
Start address 0xa000, load size 16384
Could not write register "orig_eax"; remote failure reply 'E01
'
(gdb) load lab1.out
Loading section .text, size 0x4000 lma 0xa000
Start address 0xa000, load size 16384
Transfer rate: 2 KB/sec, 47 bytes/write.
(gdb) x/2x 0xd202
0xd202: 0xd00000ff      0xfd830000
(gdb) x/2x 0xd20A
0xd20a: 0xd21000a0      0x33100000
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb)
```

```
41          mov dword [0x1ff0], 0x0
42          mov esp, 0x1ff0
43          popfd
44  ;==================================================
45  ;==========YOUR CODE==============================
46  ;==================================================
47
48          mov ax,0x18
49          mov ds,ax       ;ds starts at 0xb000(read only)
50          mov ax,0x20
51          mov es,ax       ;es starts at 0xb200(R/W)
52          mov ax,0x28
53          mov fs,ax       ;fs starts at 0xb400(R/W)
54          mov ax,0x30
55          mov gs,ax       ;gs starts at 0xb600(R/W)
56          mov ax,0x38
57          mov ss,ax       ;ss starts at 0xc000(Expand down, R/W)
58          mov esp,0x80
59          mov ebp,0x80
60
61          ;===A^2=============
62          push 0x18; //Src1: DS
63          push 0x18; //Src2: DS
64          push 0x20; //Dest: ES
65          call matrix_mul
66
67          ;===A^3=============
68          push 0x20; //Src1: ES
69          push 0x18; //Src2: DS
70          push 0x28; //Dest: FS
```

We start with the section, name your code that is line number 48, now we say first we initialize ds. So, ds will have a read only segment which is 0x, which is stored in 0x18. So, let us say where will 0x18 descriptor we stored we know that the GDT starts at d000.

(Refer Slide Time: 00:37)



```
Could not write register "orig_eax"; remote failure reply 'E01
'
(gdb) load lab1.out
Loading section .text, size 0x4000 lma 0xa000
Start address 0xa000, load size 16384
Transfer rate: 2 KB/sec, 47 bytes/write.
(gdb) x/2x 0xd202
0xd202: 0xd00000ff      0xfd830000
(gdb) x/2x 0xd20A
0xd20a: 0xd21000a0      0x33100000
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) 
```

```
        mov dword [0x1ff0], 0x0
        mov esp, 0x1ff0
        popfd
;=====================================================
;===========YOUR CODE==============================
;=====================================================

        mov ax,0x18
        mov ds,ax       ;ds starts at 0xb000(read only)
        mov ax,0x20
        mov es,ax       ;es starts at 0xb200(R/W)
        mov ax,0x28
        mov fs,ax       ;fs starts at 0xb400(R/W)
        mov ax,0x30
        mov gs,ax       ;gs starts at 0xb600(R/W)
        mov ax,0x38
        mov ss,ax       ;ss starts at 0xc000(Expand down, R/W)
        mov esp,0x80
        mov ebp,0x80

        ;===A^2=============
        push 0x18; //Src1: DS
        push 0x18; //Src2: DS
        push 0x20; //Dest: ES
        call matrix_mul

        ;===A^3=============
        push 0x20; //Src1: ES
        push 0x18; //Src2: DS
        push 0x28; //Dest: FS
```

So, we will do x slash 2x0xd018, there you see that there is a descriptor starting at b000 of limit 512 bytes and you see here it is 409. So, it is privilege zero, but it is a and the type is 0. This is privilege zero read only data descriptor. For details of this we can actually see here for the segment descriptor.

(Refer Slide Time: 01:31)

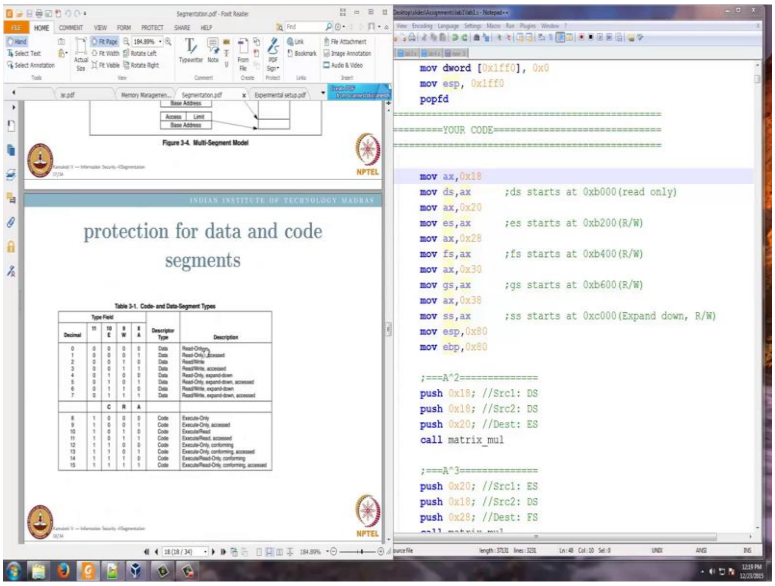


Figure 3-4. Multi-Segment Model

protection for data and code segments

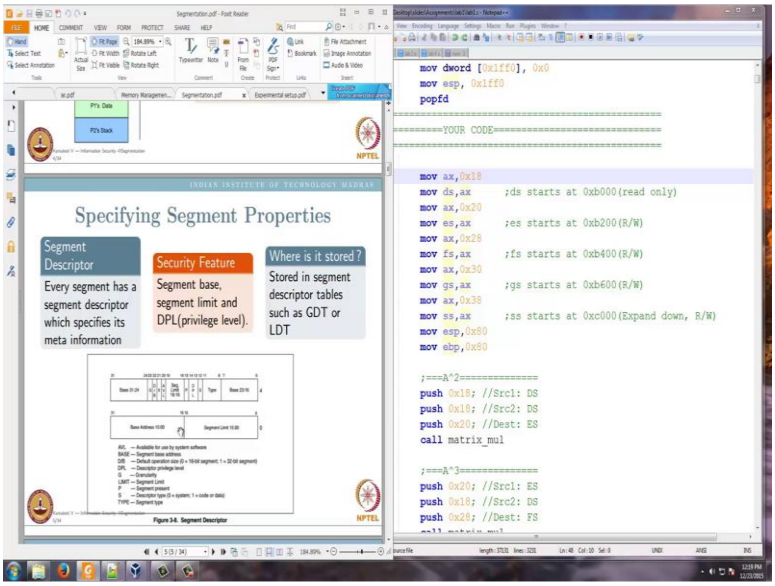
Table 3-1. Code- and Data-Segment Types

Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read-Only, conforming
15	1	1	1	1	Code	Execute/Read-Only, conforming, accessed

```
        mov dword [0x1ff0], 0x0
        mov esp, 0x1ff0
        popfd
=====================================================
=========YOUR CODE==============================
=====================================================

        mov ax,0x18
        mov ds,ax       ;ds starts at 0xb000(read only)
        mov ax,0x20
        mov es,ax       ;es starts at 0xb200(R/W)
        mov ax,0x28
        mov fs,ax       ;fs starts at 0xb400(R/W)
        mov ax,0x30
        mov gs,ax       ;gs starts at 0xb600(R/W)
        mov ax,0x38
        mov ss,ax       ;ss starts at 0xc000(Expand down, R/W)
        mov esp,0x80
        mov ebp,0x80

        ;===A^2=============
        push 0x18; //Src1: DS
        push 0x18; //Src2: DS
        push 0x20; //Dest: ES
        call matrix_mul

        ;===A^3=============
        push 0x20; //Src1: ES
        push 0x18; //Src2: DS
        push 0x28; //Dest: FS
```

So, type 0 is a read only data descriptor, if you see here and for you to understand again just going back to what had covered earlier. For you to understand, how the descriptor looks like, this is the structure of the descriptor.

(Refer Slide Time: 01:48)



INDIAN INSTITUTE OF TECHNOLOGY MADRAS

Specifying Segment Properties

Segment Descriptor: Every segment has a segment descriptor which specifies its meta information

Security Feature: Segment base, segment limit and DPL(privilege level).

Where is it stored?: Stored in segment descriptor tables such as GDT or LDT

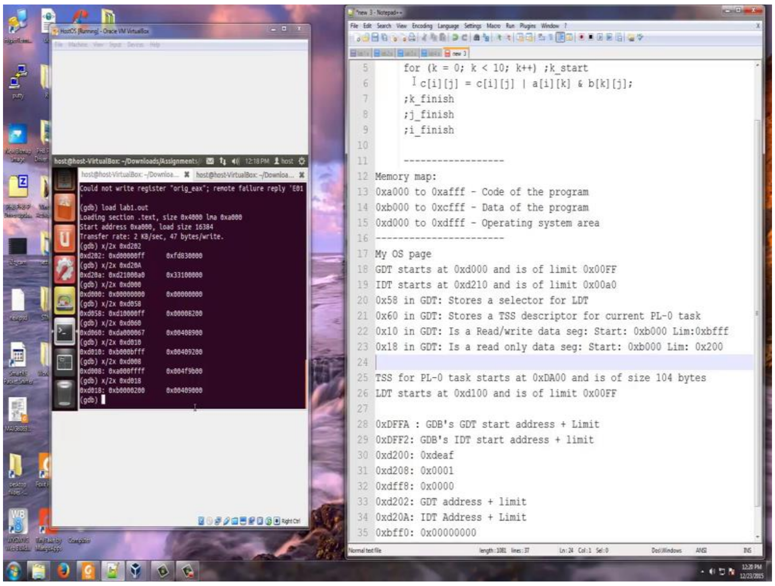
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

```
    mov dword [0x1ff0], 0x0
    mov esp, 0x1ff0
    popfd
==============================================
========YOUR CODE=============================
==============================================

    mov ax,0x18
    mov ds,ax       ;ds starts at 0xb000(read only)
    mov ax,0x20
    mov es,ax       ;es starts at 0xb200(R/W)
    mov ax,0x28
    mov fs,ax       ;fs starts at 0xb400(R/W)
    mov ax,0x30
    mov gs,ax       ;gs starts at 0xb600(R/W)
    mov ax,0x38
    mov ss,ax       ;ss starts at 0xc000(Expand down, R/W)
    mov esp,0x80
    mov ebp,0x80

    ;===A^2=============
    push 0x18; //Src1: DS
    push 0x18; //Src2: DS
    push 0x20; //Dest: ES
    call matrix_mul

    ;===A^3=============
    push 0x20; //Src1: ES
    push 0x18; //Src2: DS
    push 0x28; //Dest: FS
```

So, the base address here and the segment address limit here, part of segment limit here part of base address across three places. So, just remember this or try to note this, so that you could interpret this. Now, we have a read only descriptor at b000 of size 0x200 and privilege zero. So, let us go and make a note of it here.

(Refer Slide Time: 02:15)

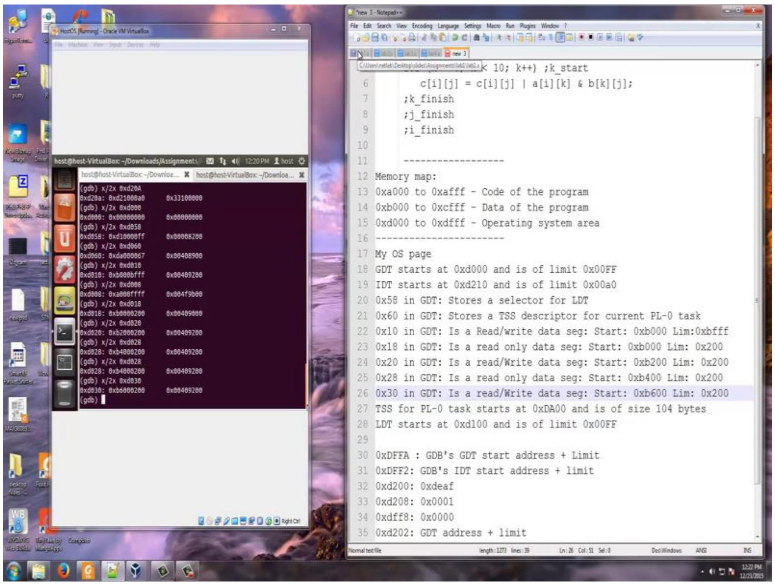


```
Could not write register "orig_eax"; remote failure reply 'E01
'
(gdb) load lab1.out
Loading section .text, size 0x4000 lma 0xa000
Start address 0xa000, load size 16384
Transfer rate: 2 KB/sec, 47 bytes/write.
(gdb) x/2x 0xd202
0xd202: 0xd00000ff      0xfd830000
(gdb) x/2x 0xd20A
0xd20a: 0xd21000a0      0x33100000
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb)
```

```
        for (k = 0; k < 10; k++) ;k_start
         c[i][j] = c[i][j] | a[i][k] & b[k][j];
        ;k_finish
        ;j_finish
        ;i_finish

        ------------------
Memory map:
0xa000 to 0xafff - Code of the program
0xb000 to 0xcfff - Data of the program
0xd000 to 0xdfff - Operating system area
-----------------------
My OS page
GDT starts at 0xd000 and is of limit 0x00FF
IDT starts at 0xd210 and is of limit 0x00a0
0x58 in GDT: Stores a selector for LDT
0x60 in GDT: Stores a TSS descriptor for current PL-0 task
0x10 in GDT: Is a Read/write data seg: Start: 0xb000 Lim:0xbfff
0x18 in GDT: Is a read only data seg: Start: 0xb000 Lim: 0x200

TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes
LDT starts at 0xd100 and is of limit 0x00FF

0xDFFA : GDB's GDT start address + Limit
0xDFF2: GDB's IDT start address + limit
0xd200: 0xdeaf
0xd208: 0x0001
0xdff8: 0x0000
0xd202: GDT address + limit
0xd20A: IDT Address + Limit
0xbff0: 0x00000000
```

So, 0x18 in GDT is a read only data seg starting at b000, its limit 0x200.

(Refer Slide Time: 02:52)

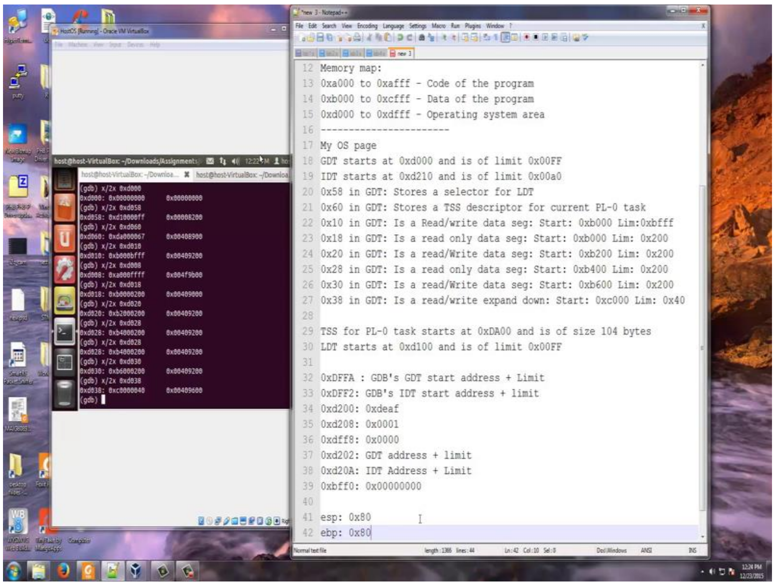


```
(gdb) x/2x 0xd20A
0xd20a: 0xd21000a0      0x33100000
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) 
```

```
                      < 10; k++) ;k_start
        c[i][j] = c[i][j] | a[i][k] & b[k][j];
      ;k_finish
      ;j_finish
      ;i_finish


      ------------------
Memory map:
0xa000 to 0xafff - Code of the program
0xb000 to 0xcfff - Data of the program
0xd000 to 0xdfff - Operating system area
-----------------------
My OS page
GDT starts at 0xd000 and is of limit 0x00FF
IDT starts at 0xd210 and is of limit 0x00a0
0x58 in GDT: Stores a selector for LDT
0x60 in GDT: Stores a TSS descriptor for current PL-0 task
0x10 in GDT: Is a Read/write data seg: Start: 0xb000 Lim:0xbfff
0x18 in GDT: Is a read only data seg: Start: 0xb000 Lim: 0x200
0x20 in GDT: Is a read/Write data seg: Start: 0xb200 Lim: 0x200
0x28 in GDT: Is a read only data seg: Start: 0xb400 Lim: 0x200
0x30 in GDT: Is a read/Write data seg: Start: 0xb600 Lim: 0x200
TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes
LDT starts at 0xd100 and is of limit 0x00FF

0xDFFA : GDB's GDT start address + Limit
0xDFF2: GDB's IDT start address + limit
0xd200: 0xdeaf
0xd208: 0x0001
0xdff8: 0x0000
0xd202: GDT address + limit
```

So, similarly let us go and say what is 0x20, line number 50 here and you see that this is again b200 but note that, this is now of type 2, this is a read write segment starting at b000 200 of size 0x200. So, here 0x2020 in GDT is a read write data segments starting at b200 and of size 200. Now, what is b28, again I am doing a slash 2x here. Note that it is again read write segment of privilege 0 starting at b400 and of size. Let us do the same thing again, here 400 now, what is 30 now, 0x30 is b000 600 again is a read write segment and what is b038, please note that this starts at c000 in this of size 0x40 hexadecimal 40 which is 64 bytes and it is type 6, what is type 6? Let us go to this and see what is type 6, the type 6 segment is read write expand down. This is type 6, its starts at c000 and of size 40.

(Refer Slide Time: 06:04)

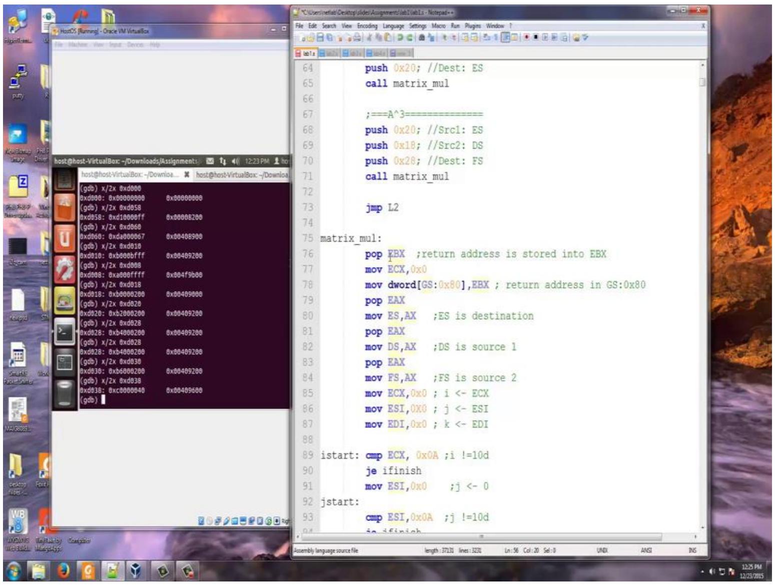


```
Memory map:
0xa000 to 0xafff - Code of the program
0xb000 to 0xcfff - Data of the program
0xd000 to 0xdfff - Operating system area
-----------------------
My OS page
GDT starts at 0xd000 and is of limit 0x00FF
IDT starts at 0xd210 and is of limit 0x00a0
0x58 in GDT: Stores a selector for LDT
0x60 in GDT: Stores a TSS descriptor for current PL-0 task
0x10 in GDT: Is a Read/write data seg: Start: 0xb000 Lim:0xbfff
0x18 in GDT: Is a read only data seg: Start: 0xb000 Lim: 0x200
0x20 in GDT: Is a read/Write data seg: Start: 0xb200 Lim: 0x200
0x28 in GDT: Is a read only data seg: Start: 0xb400 Lim: 0x200
0x30 in GDT: Is a read/Write data seg: Start: 0xb600 Lim: 0x200
0x38 in GDT: Is a read/write expand down: Start: 0xc000 Lim: 0x40

TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes
LDT starts at 0xd100 and is of limit 0x00FF

0xDFFA : GDB's GDT start address + Limit
0xDFF2: GDB's IDT start address + limit
0xd200: 0xdeaf
0xd208: 0x0001
0xdff8: 0x0000
0xd202: GDT address + limit
0xd20A: IDT Address + Limit
0xbff0: 0x00000000

esp: 0x80
ebp: 0x80
```

```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb)
```

Then you make the esp as 80, which is very important, ebp is also 80. So, we have come to line number 59.

(Refer Slide Time: 07:27)

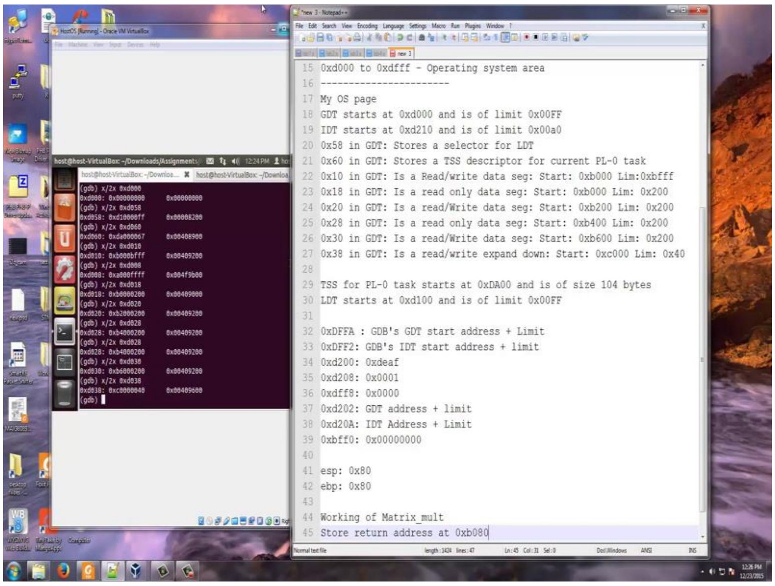


```
64          push 0x20; //Dest: ES
65          call matrix_mul
66
67          ;===A^3==============
68          push 0x20; //Src1: ES
69          push 0x18; //Src2: DS
70          push 0x28; //Dest: FS
71          call matrix_mul
72
73          jmp L2
74
75  matrix_mul:
76          pop EBX  ;return address is stored into EBX
77          mov ECX,0x0
78          mov dword[GS:0x80],EBX ; return address in GS:0x80
79          pop EAX
80          mov ES,AX   ;ES is destination
81          pop EAX
82          mov DS,AX   ;DS is source 1
83          pop EAX
84          mov FS,AX   ;FS is source 2
85          mov ECX,0x0 ; i <- ECX
86          mov ESI,0X0 ; j <- ESI
87          mov EDI,0x0 ; k <- EDI
88
89  istart: cmp ECX, 0x0A ;i !=10d
90          je ifinish
91          mov ESI,0x0    ;j <- 0
92  jstart:
93          cmp ESI,0x0A  ;j !=10d
```

```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb)
```

Now, before we start working on the line 61 and above. Let us now look at this code starting at 75, 75 is basically the matrix multiplication algorithm. So, when I call, when I execute a call matrix multiplication, what will be there on the stack is the return address. So, what we are doing here is, first we will pop the return address and store it in EBX and we will make ECX as 0 and what we are going to store is, we are storing the return address into the location GS colon 0x80. So, what is GS? GS starts at b000 600. So, in b000 680, I am going to store the written address. So, every time I call matrix mult.

(Refer Slide Time: 08:52)

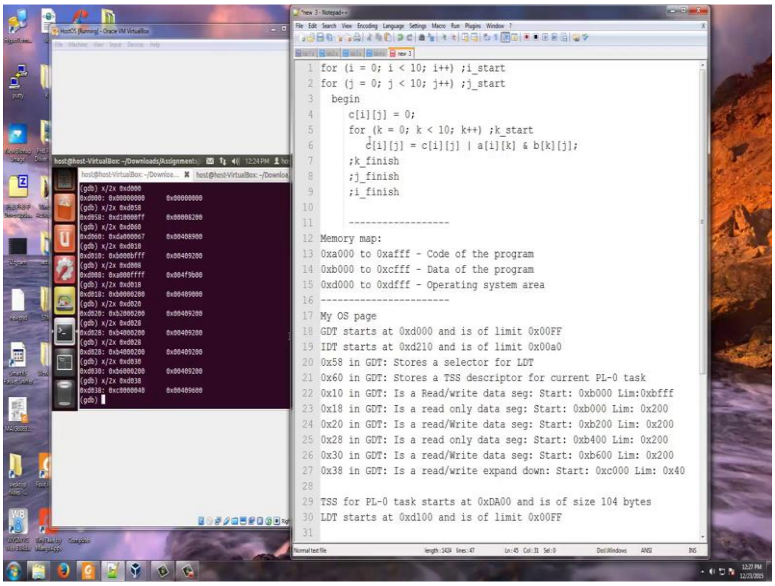


```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb)
```

```
0xd000 to 0xdfff - Operating system area
-----------------------
My OS page
GDT starts at 0xd000 and is of limit 0x00FF
IDT starts at 0xd210 and is of limit 0x00a0
0x58 in GDT: Stores a selector for LDT
0x60 in GDT: Stores a TSS descriptor for current PL-0 task
0x10 in GDT: Is a Read/write data seg: Start: 0xb000 Lim:0xbfff
0x18 in GDT: Is a read only data seg: Start: 0xb000 Lim: 0x200
0x20 in GDT: Is a read/Write data seg: Start: 0xb200 Lim: 0x200
0x28 in GDT: Is a read only data seg: Start: 0xb400 Lim: 0x200
0x30 in GDT: Is a read/Write data seg: Start: 0xb600 Lim: 0x200
0x38 in GDT: Is a read/write expand down: Start: 0xc000 Lim: 0x40

TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes
LDT starts at 0xd100 and is of limit 0x00FF

0xDFFA : GDB's GDT start address + Limit
0xDFF2: GDB's IDT start address + limit
0xd200: 0xdeaf
0xd208: 0x0001
0xdff8: 0x0000
0xd202: GDT address + limit
0xd20A: IDT Address + Limit
0xbff0: 0x00000000

esp: 0x80
ebp: 0x80

Working of Matrix_mult
Store return address at 0xb080
```

So, I will just put a session like this working of matrix mult store return address at 0xb080. Now, before going to this part of the code, we will just finish off the easier part which is i start. So, let us look at the matrix multiplication code.

(Refer Slide Time: 09:33)

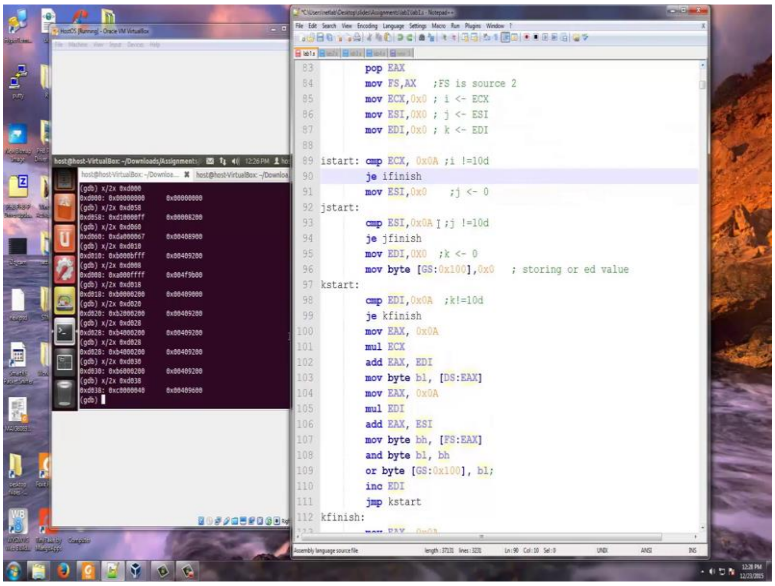


```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb)
```

```
for (i = 0; i < 10; i++) ;i_start
for (j = 0; j < 10; j++) ;j_start
  begin
     c[i][j] = 0;
     for (k = 0; k < 10; k++) ;k_start
        c[i][j] = c[i][j] | a[i][k] & b[k][j];
     ;k_finish
     ;j_finish
     ;i_finish

     -------------------
Memory map:
0xa000 to 0xafff - Code of the program
0xb000 to 0xcfff - Data of the program
0xd000 to 0xdfff - Operating system area
-----------------------
My OS page
GDT starts at 0xd000 and is of limit 0x00FF
IDT starts at 0xd210 and is of limit 0x00a0
0x58 in GDT: Stores a selector for LDT
0x60 in GDT: Stores a TSS descriptor for current PL-0 task
0x10 in GDT: Is a Read/write data seg: Start: 0xb000 Lim:0xbfff
0x18 in GDT: Is a read only data seg: Start: 0xb000 Lim: 0x200
0x20 in GDT: Is a read/Write data seg: Start: 0xb200 Lim: 0x200
0x28 in GDT: Is a read only data seg: Start: 0xb400 Lim: 0x200
0x30 in GDT: Is a read/Write data seg: Start: 0xb600 Lim: 0x200
0x38 in GDT: Is a read/write expand down: Start: 0xc000 Lim: 0x40

TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes
LDT starts at 0xd100 and is of limit 0x00FF
```

Matrix multiplication code is for i equal to 0 to 10, j equal to 0 less than 10 makes a j equal to 0 for k equal to k less than 10 c i j equal to c i j or a i k into b k j. So, there is a i variable, there is a j variable, there is a k variable. So, for all these i, j and k, we need to have central purpose registers. So, when we look at it, my ECX, this is i, so that is, i made a comment, i start, j start, then k start. All this c code is the c code starting from i start going till I finish. So, now, let us see what the c code essentially means before we go back to the code before the i start.

(Refer Slide Time: 10:29)

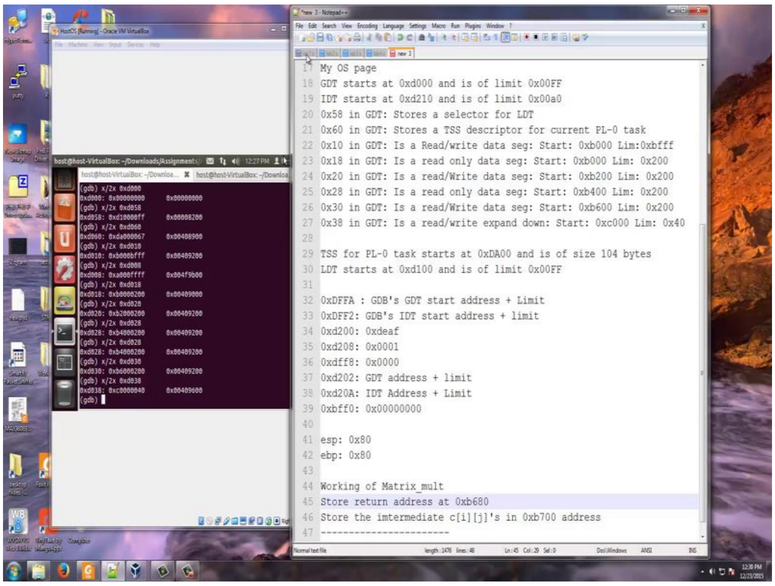


```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb) 
```

```
        pop EAX
        mov FS,AX   ;FS is source 2
        mov ECX,0x0 ; i <- ECX
        mov ESI,0X0 ; j <- ESI
        mov EDI,0x0 ; k <- EDI

istart: cmp ECX, 0x0A ;i !=10d
        je ifinish
        mov ESI,0x0    ;j <- 0
jstart:
        cmp ESI,0x0A ;j !=10d
        je jfinish
        mov EDI,0X0  ;k <- 0
        mov byte [GS:0x100],0x0    ; storing or ed value
kstart:
        cmp EDI,0x0A  ;k!=10d
        je kfinish
        mov EAX, 0x0A
        mul ECX
        add EAX, EDI
        mov byte bl, [DS:EAX]
        mov EAX, 0x0A
        mul EDI
        add EAX, ESI
        mov byte bh, [FS:EAX]
        and byte bl, bh
        or byte [GS:0x100], bl;
        inc EDI
        jmp kstart
kfinish:
```

So, ECX is a loop variable which is representing i which stores the value of i ESI is loop variable it represents the value of j and EDI is a loop variable it represents the value of k. So, initially I have made ECX as 0 long back here itself, I made ECX as 0. Now, what I do here is every time I compare ECX with 10 if it is equal this is called j e, j e means if jump and equal if it is equal i is at the end of the code. This is something like I finish the entire program, I finish the matrix multiplication but if I am not if ECX is not equal to 10 then I start the j loop ESI equal to 0. I make j equal to 0 here again. This is j start every time I compare j with 10, if j becomes equal to 10 jump one equal to I got to j finish j finish is here otherwise what I do I move EDI 0. I make k as 0, EDI stands for k and I have to make c i j equal to 0 that is what this code says c i j is equals to 0. Initially, I store c i j in some temporary location, again it is b000 700, GS is b000 600. So, this will be b000 700.

(Refer Slide Time: 12:10)

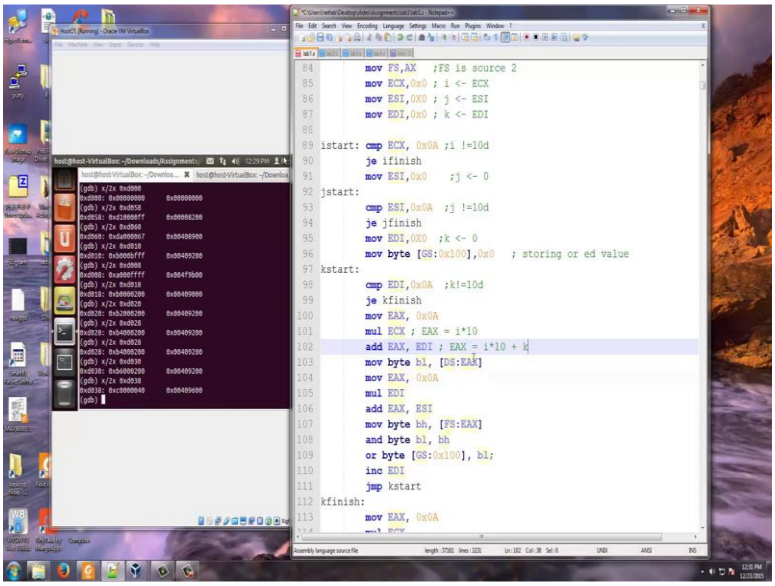


```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb) 
```

```
My OS page
GDT starts at 0xd000 and is of limit 0x00FF
IDT starts at 0xd210 and is of limit 0x00a0
0x58 in GDT: Stores a selector for LDT
0x60 in GDT: Stores a TSS descriptor for current PL-0 task
0x10 in GDT: Is a Read/write data seg: Start: 0xb000 Lim:0xbfff
0x18 in GDT: Is a read only data seg: Start: 0xb000 Lim: 0x200
0x20 in GDT: Is a read/Write data seg: Start: 0xb200 Lim: 0x200
0x28 in GDT: Is a read only data seg: Start: 0xb400 Lim: 0x200
0x30 in GDT: Is a read/Write data seg: Start: 0xb600 Lim: 0x200
0x38 in GDT: Is a read/write expand down: Start: 0xc000 Lim: 0x40

TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes
LDT starts at 0xd100 and is of limit 0x00FF

0xDFFA : GDB's GDT start address + Limit
0xDFF2: GDB's IDT start address + limit
0xd200: 0xdeaf
0xd208: 0x0001
0xdff8: 0x0000
0xd202: GDT address + limit
0xd20A: IDT Address + Limit
0xbff0: 0x00000000

esp: 0x80
ebp: 0x80

Working of Matrix_mult
Store return address at 0xb680
Store the imtermediate c[i][j]'s in 0xb700 address
----------------------
```

In the matrix multiplication, store the local the intermediate c i j s in 0xb6. So, this should also be stored it in address at b680 that is what we do because GS is b600. So, this would be b680.

(Refer Slide Time: 13:05)



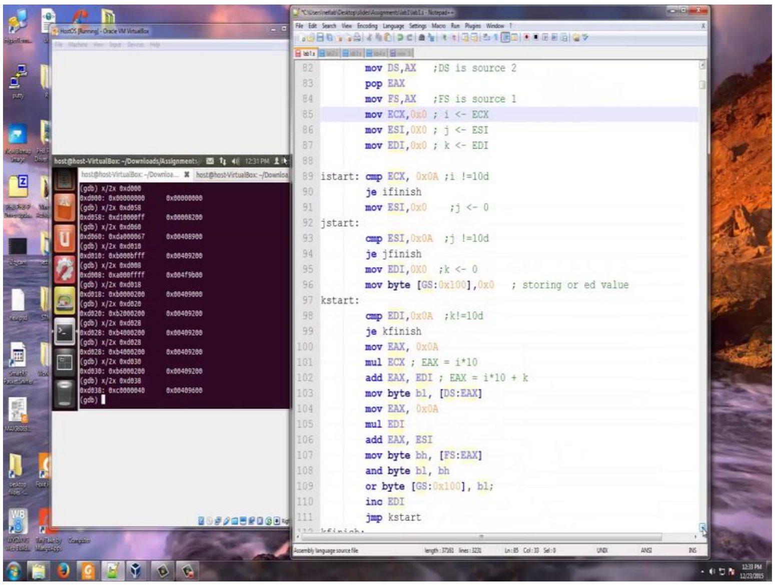
```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb) 
```

```
        mov FS,AX    ;FS is source 2
        mov ECX,0x0 ; i <- ECX
        mov ESI,0X0 ; j <- ESI
        mov EDI,0x0 ; k <- EDI

istart: cmp ECX, 0x0A ;i !=10d
        je ifinish
        mov ESI,0x0     ;j <- 0
jstart:
        cmp ESI,0x0A  ;j !=10d
        je jfinish
        mov EDI,0X0  ;k <- 0
        mov byte [GS:0x100],0x0    ; storing or ed value
kstart:
        cmp EDI,0x0A  ;k!=10d
        je kfinish
        mov EAX, 0x0A
        mul ECX ; EAX = i*10
        add EAX, EDI ; EAX = i*10 + k
        mov byte bl, [DS:EAX]
        mov EAX, 0x0A
        mul EDI
        add EAX, ESI
        mov byte bh, [FS:EAX]
        and byte bl, bh
        or byte [GS:0x100], bl;
        inc EDI
        jmp kstart
kfinish:
        mov EAX, 0x0A
```

So, I make a j 0, there is that address as 0 now I am doing the k loop I compare k with 10, if it is equal I go to k finish here jump on equal to I go to k finish here. Otherwise I do this move EAX as 10 mul ECX, mul is a command which will take the value in ECEAX and multiply with the content of ECX. Basically, this gives me i into 10 and add

EAX to EDI. So, what is EDI it is a EDI is k. So, this will give me. So, I into 10 will be stored in EAX now this will give me EAX is equal to I into 10 plus j plus k. Now, d s colon EAX, ds is the point let us go of here when I am calling the matrix multiplication I push before I call the matrix multiplication; I push the two source segments and the data destination segment into the stack. Please note that my initially the matrix is stored in ds my matrix is again stored in 18, 18 is where we have stored the matrix. So, let us go here 0x18 in GDT is a read only segment where I have stored my matrix. So, these two are the source matrices and 0x20 is the segment in which I need to store the return multiplied matrix. So, these things I am pushing into the stack. At this point of time, I pop when I call matrix mul, I pop the EBX that is the written address and then I pop the destination matrix because I pushed in this order first the destination will be there then the source two and then the source one. So, I pop.

(Refer Slide Time: 15:45)

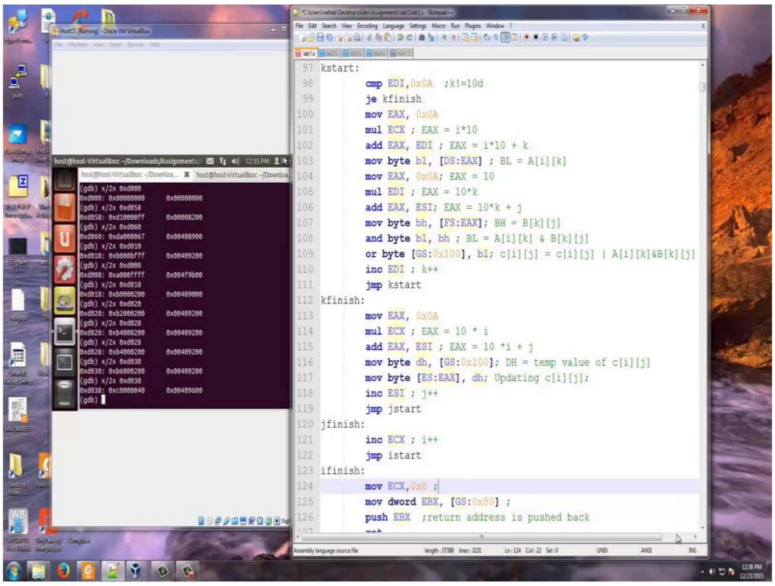


```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb)
```

```
        mov DS,AX   ;DS is source 2
        pop EAX
        mov FS,AX   ;FS is source 1
        mov ECX,0x0 ; i <- ECX
        mov ESI,0X0 ; j <- ESI
        mov EDI,0x0 ; k <- EDI

istart: cmp ECX, 0x0A ;i !=10d
        je ifinish
        mov ESI,0x0    ;j <- 0
jstart:
        cmp ESI,0x0A  ;j !=10d
        je jfinish
        mov EDI,0X0  ;k <- 0
        mov byte [GS:0x100],0x0   ; storing or ed value
kstart:
        cmp EDI,0x0A  ;k!=10d
        je kfinish
        mov EAX, 0x0A
        mul ECX ; EAX = i*10
        add EAX, EDI ; EAX = i*10 + k
        mov byte bl, [DS:EAX]
        mov EAX, 0x0A
        mul EDI
        add EAX, ESI
        mov byte bh, [FS:EAX]
        and byte bl, bh
        or byte [GS:0x100], bl;
        inc EDI
        jmp kstart
```

So, e s will store the destination segment address then again I pop ax and move it to ds, ds will store the source two and f s will store source one. So, whenever I am executing matrix multiplication, if I say ds inside that is the source one matrix.

(Refer Slide Time: 16:21)



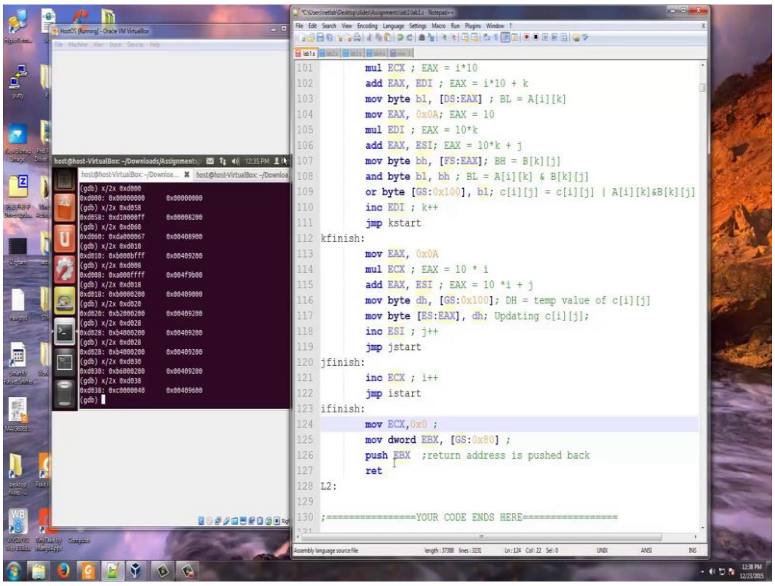
```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0008200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2008200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4008200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4008200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6008200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb)
```

```
kstart:
        cmp EDI,0x0A  ;k!=10d
        je kfinish
        mov EAX, 0x0A
        mul ECX ; EAX = i*10
        add EAX, EDI ; EAX = i*10 + k
        mov byte bl, [DS:EAX] ; BL = A[i][k]
        mov EAX, 0x0A; EAX = 10
        mul EDI ; EAX = 10*k
        add EAX, ESI; EAX = 10*k + j
        mov byte bh, [FS:EAX]; BH = B[k][j]
        and byte bl, bh ; BL = A[i][k] & B[k][j]
        or byte [GS:0x100], bl; c[i][j] = c[i][j] | A[i][k]&B[k][j]
        inc EDI ; k++
        jmp kstart
kfinish:
        mov EAX, 0x0A
        mul ECX ; EAX = 10 * i
        add EAX, ESI ; EAX = 10 *i + j
        mov byte dh, [GS:0x100]; DH = temp value of c[i][j]
        mov byte [ES:EAX], dh; Updating c[i][j];
        inc ESI ; j++
        jmp jstart
jfinish:
        inc ECX ; i++
        jmp istart
ifinish:
        mov ECX,0x0 ;
        mov dword EBX, [GS:0x80] ;
        push EBX  ;return address is pushed back
        ret
```

So, here what time moving to the register b l, it is of i k is now in b l in register b l, b l is the 8 byte of register. First 8 bytes of raise the EB, now again I move EAX as ten, EAX becomes I into 10 EAX becomes 10 and mul EDI. What is EDI? EDI is k. So, EDI is now EAX becomes mul EDI will multiply the content of EAX with EDI and store the answer. So, EAX now will become 10 into k add EAX with ESI, ESI is j. So, EAX is now 10 into k plus j.

Now, move byte b h f s colon EAX. So, what is fs? fs is the source two. So, this will store b of k k j. So, b h is equal to b of k j now and byte b l and b h. So, I will get ah. So, so the answer would be in bl. So, bl will be equal to now a i k and b k j byte wise and amperes. Now, I would byte in or this value this stores the temporary result that is the temporary location for c i j, this is bb700. So, c i j now will become c i j plus a i k and b k j. So, this plus is nothing but now I am increment EDI this is nothing, but k plus and then it comes to k stack again it completes. So, this explains how the matrix works and after a finish one k loop again I move EAX as 0 a multiply ECX. So, what is ECX? So, now, EAX becomes 10 into i and add EAX comma ESI. So, this becomes EAX actually becomes 10 into i plus j and GS stores the destination segment. So, i moving into d h the temporary value of c i j and actually I should say a temp value of, now, I am updating c i j right. So, this is the way I do and also note that I cannot directly move byte e x colon EAX to GS colon 0x100, I cannot do two memory operations in an instruction that is a restriction of the x-axis instruction at architecture. So, there can be only one memory operand I can do. So, essentially I am doing it as two instructions here. Then I am incrementing ESI, ESI is nothing, but j plus then I jump to j start, once j is finish I am incrementing i plus and I start after this is over I make my counter i as 0.

(Refer Slide Time: 20:41)



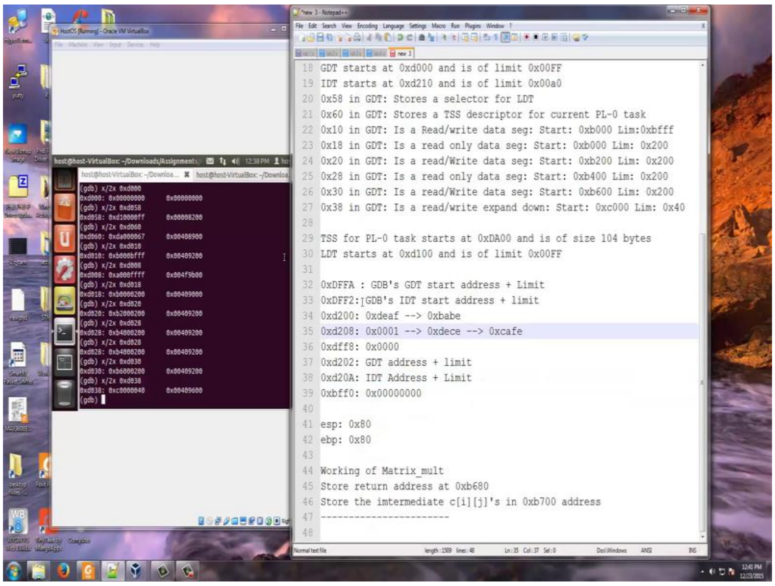
```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb)
```

```
        mul ECX ; EAX = i*10
        add EAX, EDI ; EAX = i*10 + k
        mov byte bl, [DS:EAX] ; BL = A[i][k]
        mov EAX, 0x0A; EAX = 10
        mul EDI ; EAX = 10*k
        add EAX, ESI; EAX = 10*k + j
        mov byte bh, [FS:EAX]; BH = B[k][j]
        and byte bl, bh ; BL = A[i][k] & B[k][j]
        or byte [GS:0x100], bl; c[i][j] = c[i][j] | A[i][k]&B[k][j]
        inc EDI ; k++
        jmp kstart
kfinish:
        mov EAX, 0x0A
        mul ECX ; EAX = 10 * i
        add EAX, ESI ; EAX = 10 *i + j
        mov byte dh, [GS:0x100]; DH = temp value of c[i][j]
        mov byte [ES:EAX], dh; Updating c[i][j];
        inc ESI ; j++
        jmp jstart
jfinish:
        inc ECX ; i++
        jmp istart
ifinish:
        mov ECX,0x0 ;
        mov dword EBX, [GS:0x80] ;
        push EBX  ;return address is pushed back
        ret
L2:

;================YOUR CODE ENDS HERE=================
```

Then you know that GS colon 0x80 is storing the written address as you see here store return address as b680 that is GS colon 080. So, I move it to EBX and I push that value into the stack and I do a return. So, I actually jump to the written address.

I jump to the written address. Initially what I do, I stored the matrix is stored in the segment given by the selectors 0x18. I multiply and a second source is also 0x18. The same matrix now, the destination matrix 0x20 has a square after this at this point. Now, again I am pushing a square as the source one and as source two and in 0x28. I will have the return matrix, which is a cube and after I finish this I jump l 2 that is I jump this entire routine and come to this location then what I do here is I go back to my original 0x10 segment, what is that 10 segment? 0x10 is a start b00 and codes to limit bff, I go and make b plus 2d208 as 0xd ECE.

(Refer Slide Time: 22:30)

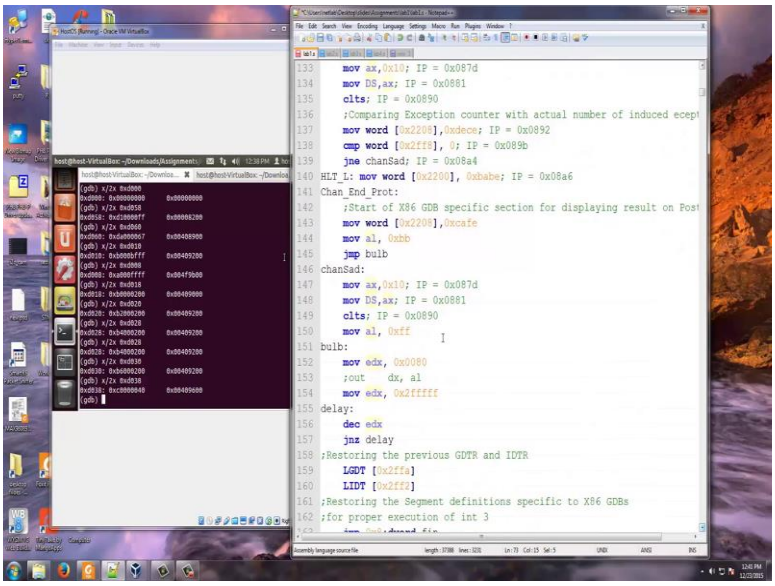


```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb) 
```

```
GDT starts at 0xd000 and is of limit 0x00FF
IDT starts at 0xd210 and is of limit 0x00a0
0x58 in GDT: Stores a selector for LDT
0x60 in GDT: Stores a TSS descriptor for current PL-0 task
0x10 in GDT: Is a Read/write data seg: Start: 0xb000 Lim:0xbfff
0x18 in GDT: Is a read only data seg: Start: 0xb000 Lim: 0x200
0x20 in GDT: Is a read/Write data seg: Start: 0xb200 Lim: 0x200
0x28 in GDT: Is a read only data seg: Start: 0xb400 Lim: 0x200
0x30 in GDT: Is a read/Write data seg: Start: 0xb600 Lim: 0x200
0x38 in GDT: Is a read/write expand down: Start: 0xc000 Lim: 0x40

TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes
LDT starts at 0xd100 and is of limit 0x00FF

0xDFFA : GDB's GDT start address + Limit
0xDFF2:GDB's IDT start address + limit
0xd200: 0xdeaf --> 0xbabe
0xd208: 0x0001 --> 0xdece --> 0xcafe
0xdff8: 0x0000
0xd202: GDT address + limit
0xd20A: IDT Address + Limit
0xbff0: 0x00000000

esp: 0x80
ebp: 0x80

Working of Matrix_mult
Store return address at 0xb680
Store the imtermediate c[i][j]'s in 0xb700 address
-----------------------

```

So, this now becomes 0xdece to start with and 2ff8 as 0. So, this is b plus 2 is ddff8 as 0 it remains 0 and then d200 as 0xbabed200 as b and d208 as cafe.

(Refer Slide Time: 23:34)

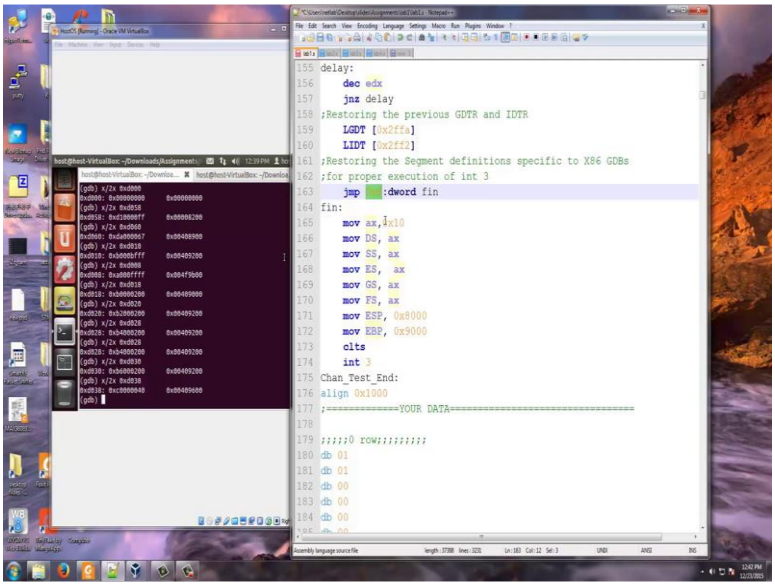


```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb) 
```

```
    mov ax,0x10; IP = 0x087d
    mov DS,ax; IP = 0x0881
    clts; IP = 0x0890
    ;Comparing Exception counter with actual number of induced ecept
    mov word [0x2208],0xdece; IP = 0x0892
    cmp word [0x2ff8], 0; IP = 0x089b
    jne chanSad; IP = 0x08a4
HLT_L: mov word [0x2200], 0xbabe; IP = 0x08a6
Chan_End_Prot:
    ;Start of X86 GDB specific section for displaying result on Post
    mov word [0x2208],0xcafe
    mov al, 0xbb
    jmp bulb
chanSad:
    mov ax,0x10; IP = 0x087d
    mov DS,ax; IP = 0x0881
    clts; IP = 0x0890
    mov al, 0xff
bulb:
    mov edx, 0x0080
    ;out    dx, al
    mov edx, 0x2fffff
delay:
    dec edx
    jnz delay
;Restoring the previous GDTR and IDTR
    LGDT [0x2ffa]
    LIDT [0x2ff2]
;Restoring the Segment definitions specific to X86 GDBs
;for proper execution of int 3
```

Then it goes and so these are all some statement, which we need not worry about now.

(Refer Slide Time: 23:48)

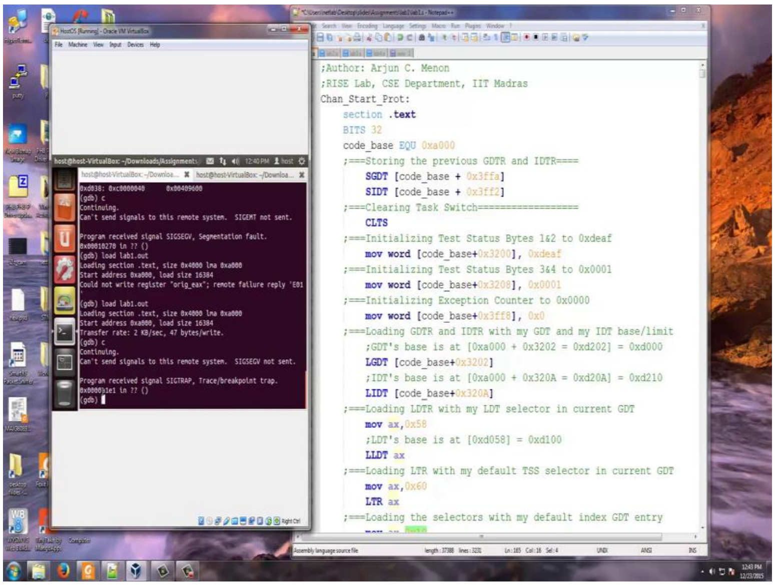


```
(gdb) x/2x 0xd000
0xd000: 0x00000000      0x00000000
(gdb) x/2x 0xd058
0xd058: 0xd10000ff      0x00008200
(gdb) x/2x 0xd060
0xd060: 0xda000067      0x00408900
(gdb) x/2x 0xd010
0xd010: 0xb000bfff      0x00409200
(gdb) x/2x 0xd008
0xd008: 0xa000ffff      0x004f9b00
(gdb) x/2x 0xd018
0xd018: 0xb0000200      0x00409000
(gdb) x/2x 0xd020
0xd020: 0xb2000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd028
0xd028: 0xb4000200      0x00409200
(gdb) x/2x 0xd030
0xd030: 0xb6000200      0x00409200
(gdb) x/2x 0xd038
0xd038: 0xc0000040      0x00409600
(gdb) 
```

```
155 delay:
156     dec edx
157     jnz delay
158 ;Restoring the previous GDTR and IDTR
159     LGDT [0x2ffa]
160     LIDT [0x2ff2]
161 ;Restoring the Segment definitions specific to X86 GDBs
162 ;for proper execution of int 3
163     jmp [illegible]:dword fin
164 fin:
165     mov ax,0x10
166     mov DS, ax
167     mov SS, ax
168     mov ES,  ax
169     mov GS, ax
170     mov FS, ax
171     mov ESP, 0x8000
172     mov EBP, 0x9000
173     clts
174     int 3
175 Chan_Test_End:
176 align 0x1000
177 ;=============YOUR DATA==================================
178 
179 ;;;;;0 row;;;;;;;;;
180 db 01
181 db 01
182 db 00
183 db 00
184 db 00
```

Let us go to 159, it loads GDT from b plus 2 dffa. So, what was dffa dffa was GDT. So, that old GDT is loaded similarly 2ff2 that is dff2 stored IDT and. So, it is load the old IDT of the GDT. So, now, it says jump 0xeight this corresponds to the course segment of the GDT because already I am loaded the GDT of that and d word fin is next. So, it comes here and then it initializes all d s and this these are all the descriptors in the original GDB and then it says in three fine now we should know what are these code segment and ah you know the data segment in the GDT of the GDB carnal right that we will see once we execute it. So, this is how the program basically executes now this is loaded now how do you execute the program.

(Refer Slide Time: 25:02)

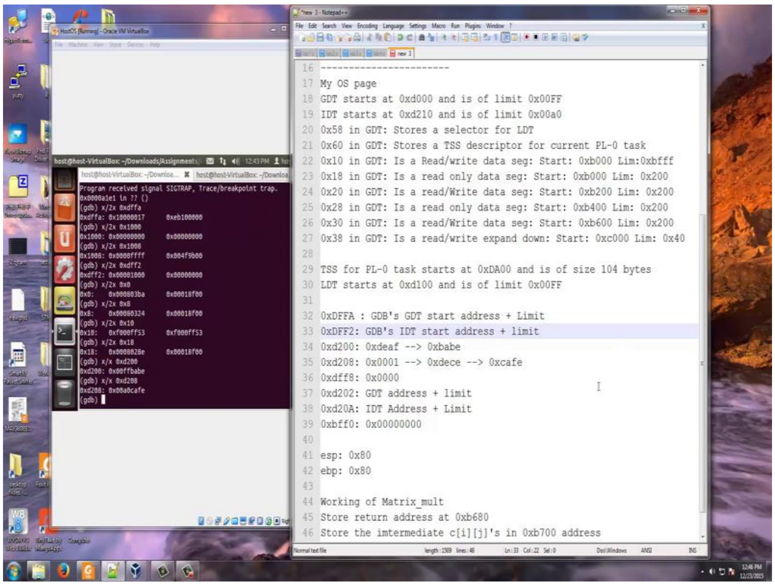


```
0xd038: 0xc0000040      0x00409600
(gdb) c
Continuing.
Can't send signals to this remote system.  SIGEMT not sent.

Program received signal SIGSEGV, Segmentation fault.
0x00010270 in ?? ()
(gdb) load lab1.out
Loading section .text, size 0x4000 lma 0xa000
Start address 0xa000, load size 16384
Could not write register "orig_eax"; remote failure reply 'E01
'
(gdb) load lab1.out
Loading section .text, size 0x4000 lma 0xa000
Start address 0xa000, load size 16384
Transfer rate: 2 KB/sec, 47 bytes/write.
(gdb) c
Continuing.
Can't send signals to this remote system.  SIGSEGV not sent.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000a1e1 in ?? ()
(gdb) 
```

```
;Author: Arjun C. Menon
;RISE Lab, CSE Department, IIT Madras
Chan_Start_Prot:
    section .text
    BITS 32
    code_base EQU 0xa000
    ;===Storing the previous GDTR and IDTR====
        SGDT [code_base + 0x3ffa]
        SIDT [code_base + 0x3ff2]
    ;===Clearing Task Switch==================
        CLTS
    ;===Initializing Test Status Bytes 1&2 to 0xdeaf
        mov word [code_base+0x3200], 0xdeaf
    ;===Initializing Test Status Bytes 3&4 to 0x0001
        mov word [code_base+0x3208], 0x0001
    ;===Initializing Exception Counter to 0x0000
        mov word [code_base+0x3ff8], 0x0
    ;===Loading GDTR and IDTR with my GDT and my IDT base/limit
        ;GDT's base is at [0xa000 + 0x3202 = 0xd202] = 0xd000
        LGDT [code_base+0x3202]
        ;IDT's base is at [0xa000 + 0x320A = 0xd20A] = 0xd210
        LIDT [code_base+0x320A]
    ;===Loading LDTR with my LDT selector in current GDT
        mov ax,0x58
        ;LDT's base is at [0xd058] = 0xd100
        LLDT ax
    ;===Loading LTR with my default TSS selector in current GDT
        mov ax,0x60
        LTR ax
    ;===Loading the selectors with my default index GDT entry
```

Till now we have just seen how it is loaded now we just press c now you get some signal sigsegv this is a segmentation fault this is again some problem with the you know the mimicking of the thing. So, again we load it no there is an error again we load it now we continue you should a sigtrap; that means, the program has executed correctly the last instruction executed was a1 e1. Right now, we will go and see what are all the things happening here first and foremost.

(Refer Slide Time: 25:56)



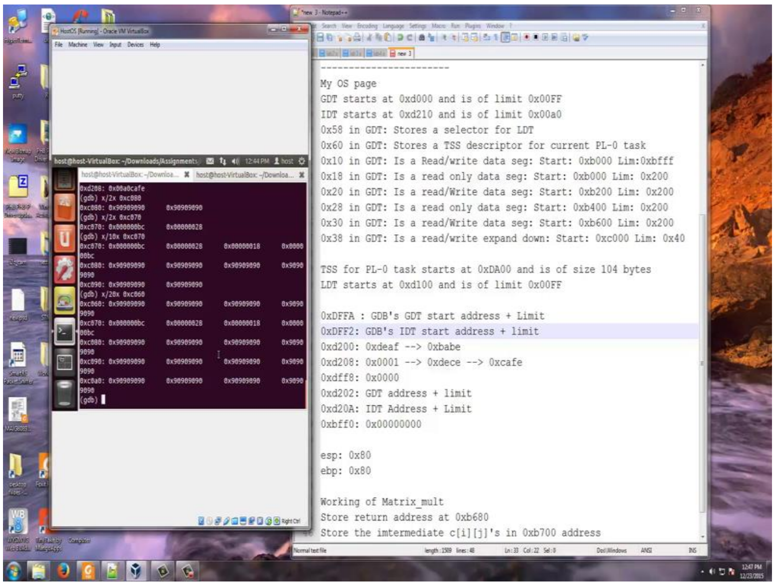
```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000a1e1 in ?? ()
(gdb) x/2x 0xdffa
0xdffa: 0x10000017      0xeb100000
(gdb) x/2x 0x1000
0x1000: 0x00000000      0x00000000
(gdb) x/2x 0x1008
0x1008: 0x0000ffff      0x004f9b00
(gdb) x/2x 0xdff2
0xdff2: 0x00001000      0x00000000
(gdb) x/2x 0x8
0x8:    0x000083ba      0x00018f00
(gdb) x/2x 0x8
0x8:    0x00008324      0x00018f00
(gdb) x/2x 0x10
0x10:   0xf000ff53      0xf000ff53
(gdb) x/2x 0x18
0x18:   0x0000828e      0x00018f00
(gdb) x/x 0xd200
0xd200: 0x00ffbabe
(gdb) x/x 0xd208
0xd208: 0x00a0cafe
(gdb) 
```

```
-----------------------
My OS page
GDT starts at 0xd000 and is of limit 0x00FF
IDT starts at 0xd210 and is of limit 0x00a0
0x58 in GDT: Stores a selector for LDT
0x60 in GDT: Stores a TSS descriptor for current PL-0 task
0x10 in GDT: Is a Read/write data seg: Start: 0xb000 Lim:0xbfff
0x18 in GDT: Is a read only data seg: Start: 0xb000 Lim: 0x200
0x20 in GDT: Is a read/Write data seg: Start: 0xb200 Lim: 0x200
0x28 in GDT: Is a read only data seg: Start: 0xb400 Lim: 0x200
0x30 in GDT: Is a read/Write data seg: Start: 0xb600 Lim: 0x200
0x38 in GDT: Is a read/write expand down: Start: 0xc000 Lim: 0x40

TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes
LDT starts at 0xd100 and is of limit 0x00FF

0xDFFA : GDB's GDT start address + Limit
0xDFF2: GDB's IDT start address + limit
0xd200: 0xdeaf --> 0xbabe
0xd208: 0x0001 --> 0xdece --> 0xcafe
0xdff8: 0x0000
0xd202: GDT address + limit
0xd20A: IDT Address + Limit
0xbff0: 0x00000000

esp: 0x80
ebp: 0x80

Working of Matrix_mult
Store return address at 0xb680
Store the imtermediate c[i][j]'s in 0xb700 address
```

Let us go and see x slash 2 x 0xdffa there we note that this is the place where your GDB is GDT is there. So, 00 one seven is the GDT start is a size of the GDT of GDB and it starts at 0xone000. So, suppose. So, x slash x 2 x 0xone000 one000 note this is a null descriptor one000 8 is the code descriptor please note that the codes should stands starts at 0 and it is privilege level is 0 and it has a highest limit offfff and so this is the course segment descriptor of the GDT.

Similarly, IDT where does it starts x slash 2 x that IDT value is stored in dff2 please note the screen here. So, the IDT again had a limit of000 or the IDT actually started at 0. So, x slash ah 2 x 0x0. So, this is the first IDT for divide by 00x8 is for the first interrupt one, this is for interrupt two. This is how it looks into three that we are executing this is by restart. So, these are all the incorruptible for at the GDB's IDT now let us see what this happening whether the code has work correctly 0xd200 bcbabed208, we should see cafe.

(Refer Slide Time: 28:20)

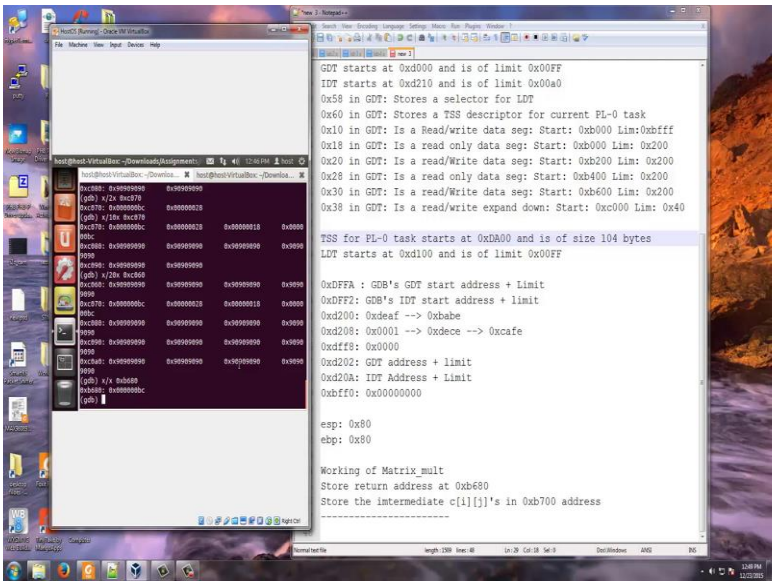


```
0xd208: 0x00a0cafe
(gdb) x/2x 0xc080
0xc080: 0x90909090      0x90909090
(gdb) x/2x 0xc070
0xc070: 0x000000bc      0x00000028
(gdb) x/10x 0xc070
0xc070: 0x000000bc      0x00000028      0x00000018      0x0000
00bc
0xc080: 0x90909090      0x90909090      0x90909090      0x9090
9090
0xc090: 0x90909090      0x90909090
(gdb) x/20x 0xc060
0xc060: 0x90909090      0x90909090      0x90909090      0x9090
9090
0xc070: 0x000000bc      0x00000028      0x00000018      0x0000
00bc
0xc080: 0x90909090      0x90909090      0x90909090      0x9090
9090
0xc090: 0x90909090      0x90909090      0x90909090      0x9090
9090
0xc0a0: 0x90909090      0x90909090      0x90909090      0x9090
9090
(gdb)
```

```
-----------------------
My OS page
GDT starts at 0xd000 and is of limit 0x00FF
IDT starts at 0xd210 and is of limit 0x00a0
0x58 in GDT: Stores a selector for LDT
0x60 in GDT: Stores a TSS descriptor for current PL-0 task
0x10 in GDT: Is a Read/write data seg: Start: 0xb000 Lim:0xbfff
0x18 in GDT: Is a read only data seg: Start: 0xb000 Lim: 0x200
0x20 in GDT: Is a read/Write data seg: Start: 0xb200 Lim: 0x200
0x28 in GDT: Is a read only data seg: Start: 0xb400 Lim: 0x200
0x30 in GDT: Is a read/Write data seg: Start: 0xb600 Lim: 0x200
0x38 in GDT: Is a read/write expand down: Start: 0xc000 Lim: 0x40

TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes
LDT starts at 0xd100 and is of limit 0x00FF

0xDFFA : GDB's GDT start address + Limit
0xDFF2: GDB's IDT start address + limit
0xd200: 0xdeaf --> 0xbabe
0xd208: 0x0001 --> 0xdece --> 0xcafe
0xdff8: 0x0000
0xd202: GDT address + limit
0xd20A: IDT Address + Limit
0xbff0: 0x00000000

esp: 0x80
ebp: 0x80

Working of Matrix_mult
Store return address at 0xb680
Store the imtermediate c[i][j]'s in 0xb700 address
```

We will also see something that has happened to a stack. The stack is 0xe000, so x slash 2x0xc080 because the stack point row set to 80, it will expand down segment. So, we should see c070 as you see here from c080, there as the stack has been populated bc18 28 bc in stuff like that. So, you can even say x slash 20xc060 then we can see much more here. So, the 10bc2818bc, these are all the stack values that have been pushed into this. So, we had been pushing 28 18 as you see here into the stack. When we were calling this calling these stack here please note we have being pushing 18 and 20, etcetera but then we also pushed finally, we pushed 28 18 and this bc is the written address if you just calculate it will be a 0bc is where you come back here and. So, this is all that and we can also see that the written address was stored in some place.

(Refer Slide Time: 30:35)



```
0xc080: 0x90909090      0x90909090
(gdb) x/2x 0xc070
0xc070: 0x000000bc      0x00000028
(gdb) x/10x 0xc070
0xc070: 0x000000bc      0x00000028      0x00000018      0x0000
00bc
0xc080: 0x90909090      0x90909090      0x90909090      0x9090
9090
0xc090: 0x90909090      0x90909090
(gdb) x/20x 0xc060
0xc060: 0x90909090      0x90909090      0x90909090      0x9090
9090
0xc070: 0x000000bc      0x00000028      0x00000018      0x0000
00bc
0xc080: 0x90909090      0x90909090      0x90909090      0x9090
9090
0xc090: 0x90909090      0x90909090      0x90909090      0x9090
9090
0xc0a0: 0x90909090      0x90909090      0x90909090      0x9090
9090
(gdb) x/x 0xb680
0xb680: 0x000000bc
(gdb)
```

```
GDT starts at 0xd000 and is of limit 0x00FF
IDT starts at 0xd210 and is of limit 0x00a0
0x58 in GDT: Stores a selector for LDT
0x60 in GDT: Stores a TSS descriptor for current PL-0 task
0x10 in GDT: Is a Read/write data seg: Start: 0xb000 Lim:0xbfff
0x18 in GDT: Is a read only data seg: Start: 0xb000 Lim: 0x200
0x20 in GDT: Is a read/Write data seg: Start: 0xb200 Lim: 0x200
0x28 in GDT: Is a read only data seg: Start: 0xb400 Lim: 0x200
0x30 in GDT: Is a read/Write data seg: Start: 0xb600 Lim: 0x200
0x38 in GDT: Is a read/write expand down: Start: 0xc000 Lim: 0x40

TSS for PL-0 task starts at 0xDA00 and is of size 104 bytes
LDT starts at 0xd100 and is of limit 0x00FF

0xDFFA : GDB's GDT start address + Limit
0xDFF2: GDB's IDT start address + limit
0xd200: 0xdeaf --> 0xbabe
0xd208: 0x0001 --> 0xdece --> 0xcafe
0xdff8: 0x0000
0xd202: GDT address + limit
0xd20A: IDT Address + Limit
0xbff0: 0x00000000

esp: 0x80
ebp: 0x80

Working of Matrix_mult
Store return address at 0xb680
Store the imtermediate c[i][j]'s in 0xb700 address
-----------------------
```

You can just say x slash x0xb680, Now, that this is the loop return address and that is also bc that was stored there and the matrix of course, is going to be stored in the final segment starting at b400 is where it is stored.

This basically explains you the working of segmentation, how do you set of segment registers? How do you convert a c code into assembly code? And how you execute this entire frame work? I hope you enjoyed. So, you can now start working on this, I want you to do expand this code and do a power 5 power 10. What it means, I just want you to do more exercises on this probably increase the size of the matrix. So, these are something that you can play with, so that you get a grip over the entire assembly handling of x86.

Thank you.