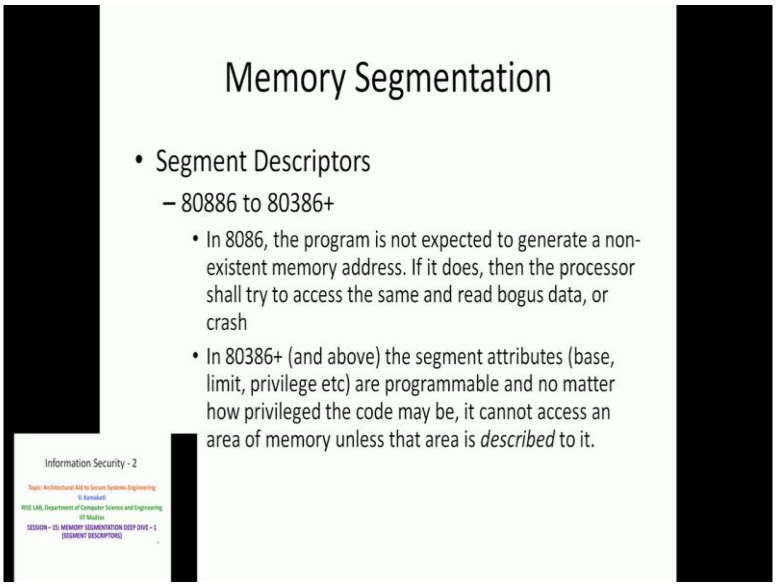
Information Security - II
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 16
Topic: Architectural Aid to Secure Systems Engineering
Session-15: Memory Segmentation Deep Dive-1

(Refer Slide Time: 00:12)



Memory Segmentation

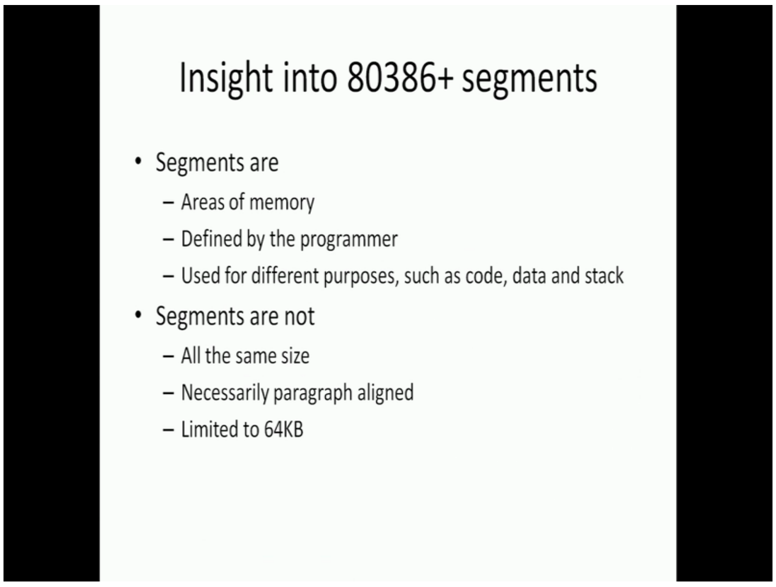
- Segment Descriptors
  - 80886 to 80386+
    - In 8086, the program is not expected to generate a non-existent memory address. If it does, then the processor shall try to access the same and read bogus data, or crash
    - In 80386+ (and above) the segment attributes (base, limit, privilege etc) are programmable and no matter how privileged the code may be, it cannot access an area of memory unless that area is *described* to it.

Information Security - 2
Topic: Architectural Aid to Secure Systems Engineering
V. Kamakoti
RISE LAB, Department of Computer Science and Engineering
IIT Madras
SESSION – 15: MEMORY SEGMENTATION DEEP DIVE – 1
(SEGMENT DESCRIPTORS)

So, now we start today we will do lot more of what we have seen so far. What have done in the previous, still now is to give you a glimpse of all that we have going to teach in this course till the last session 55 or 52. We are in session-15. Next 20, 30 sessions what we are going to do, I have given a nutshell. So, we are going to talk about segmentation, we are going implement segmentation tomorrow, we are going to talk about paging, we are implementing paging, we are going to talk about task switching, we going to implement task switching, we are going to talk about interrupt service routine, we going to implement interrupt service routine. All the basic things that are necessary to have some understanding of this I have covered. Now, what I am going to do next is I am going to do a deep dive in to each one of this topic. So, today we will have a more in depth understanding in this session about Memory Segmentation specifically segment descriptors, we will have a better understanding of this segment descriptors. Segment descriptors came in to existence, right from 80386 above and it actually stores some of the attributes of the segment namely base limit privilege etcetera.

Please note that, whatever (Refer Time: 02:00) privilege level I am, I will not be allowed to access memory without having a proper segment and that is 1 fundamental strength of the Intel architecture. If I want to access memory, it should be defined as a part of some segment and that should be privilege level associated with it and all the checks that are needed to go. However, big I may be the Real Kernel or the Micro Kernel or the Nano Kernel inside I may the super boss there, but the movement segmentation is evoke everything has to go through this particular segmentation principle. And segmentation is there by default in the protected mode, I cannot have a protected mode without segmentation and that makes development on the Intel architecture and the AMD architecture is very interesting.

(Refer Slide Time: 02:55)



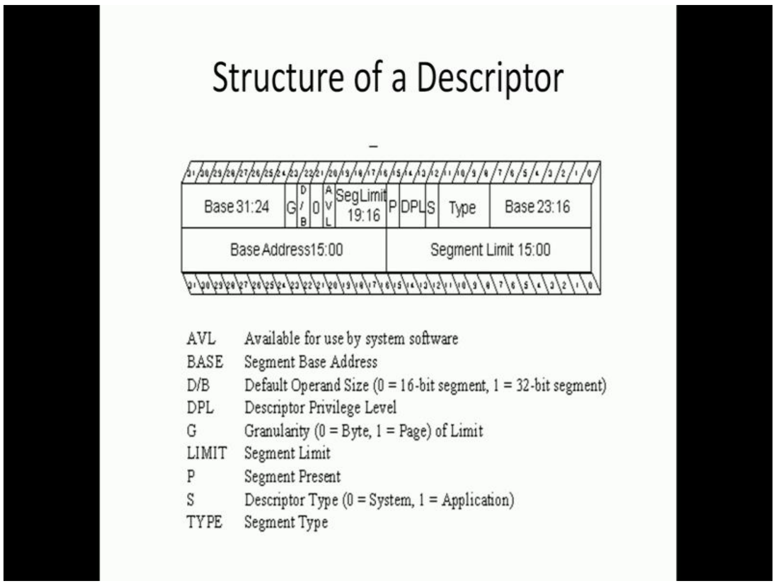
Insight into 80386+ segments

- Segments are
  - Areas of memory
  - Defined by the programmer
  - Used for different purposes, such as code, data and stack
- Segments are not
  - All the same size
  - Necessarily paragraph aligned
  - Limited to 64KB

So, what are all the different type of segments? What is a segment? Segment is actually a area of memory that is defined by the programmer, and it can be used for different purposes store different things depending upon what is store the segment is basically called a Code Segment or Data Segment or Stack Segment.

Please note that, segments are not all the same size they need not be paragraph wise aligned, and on the segments I can draft 64 kilo bytes of segments. The number of segments that I could have can be up to 64 kilo bytes. Why 64 kilo bytes? 2 power 16 is 64 is kilo bytes. Then, so the total amount off. Why is a 64 kilo bytes segments? Are you watching this, so this is 80386 we are starting off. Now, this segment size can go up to even 4 GB today. So we are talking of the old segment sizes in real mode 80386. This is where people started before protected mode when 80386 executing in it real mode, we started something call real mode which will not now discuss that it is meaningless, because when we start we need to have legacy of x86 itself. Can we a course by itself? So, we will not going to that detail this is not useful as of now. But, the old segments, real mode segments they were of 64 KB size, maximum 64 KB.

(Refer Slide Time: 05:05)

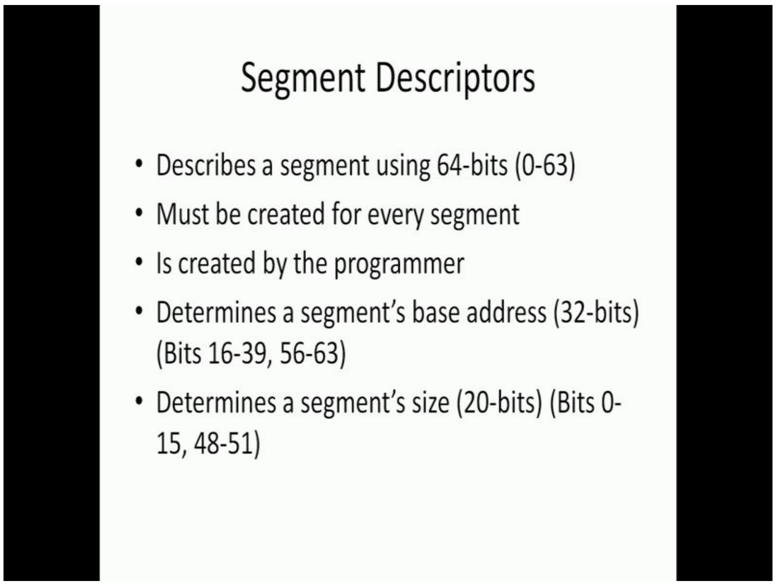


Structure of a Descriptor

AVL	Available for use by system software
BASE	Segment Base Address
D/B	Default Operand Size (0 = 16-bit segment, 1 = 32-bit segment)
DPL	Descriptor Privilege Level
G	Granularity (0 = Byte, 1 = Page) of Limit
LIMIT	Segment Limit
P	Segment Present
S	Descriptor Type (0 = System, 1 = Application)
TYPE	Segment Type

But now, thing have changed and this is now the structure of every descriptors, we had gone through this in great detail. So, some of the things that we need to keep is the Granularity byte when it is 0 the segment limit is measured in bytes, when it is 1 the segment limits is measure in 4 KB pages. And then there is a Descriptor Privilege Level, which we need to understand.

(Refer Slide Time: 05:36)

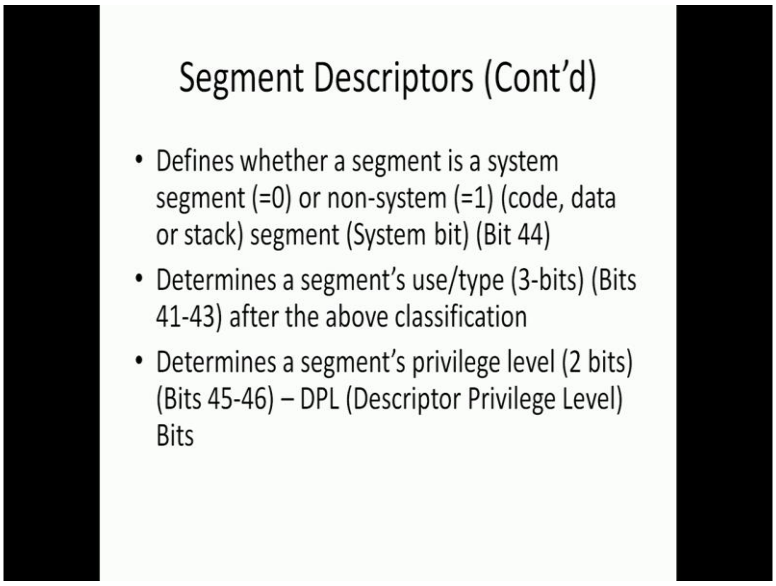


Segment Descriptors

- Describes a segment using 64-bits (0-63)
- Must be created for every segment
- Is created by the programmer
- Determines a segment's base address (32-bits) (Bits 16-39, 56-63)
- Determines a segment's size (20-bits) (Bits 0-15, 48-51)

So, what does the segment descriptor do? It describes the segment and for every segment that I am using I need to have a segment descriptor and it is created actually not by the programmer in the sense, it is created by the operating system not by the architecture. And it determines a segments base address; there are 32-bits which 16 to 39 and 56 to 63 because it is totally 60 to 63 bit segment description. It also determines the segment size by 20-bits, 0 to 15 and 48 to 51.

(Refer Slide Time: 06:17)

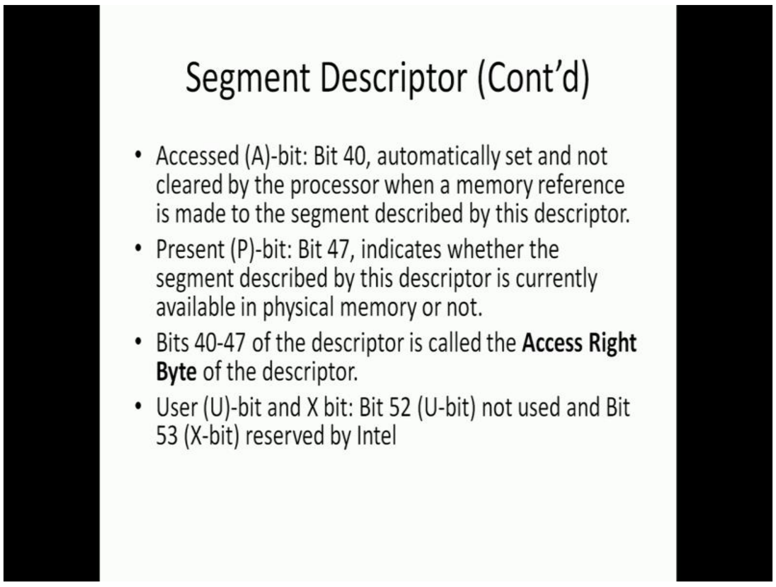


Segment Descriptors (Cont'd)

- Defines whether a segment is a system segment (=0) or non-system (=1) (code, data or stack) segment (System bit) (Bit 44)
- Determines a segment's use/type (3-bits) (Bits 41-43) after the above classification
- Determines a segment's privilege level (2 bits) (Bits 45-46) – DPL (Descriptor Privilege Level) Bits

It also defines whether a segment is a system segment that is 0 or a non-system segment that is user segment which is code, data, stack and that is called system bit, bit 44. 1 means non-system, o means system. It also determines the segments use slash type there are 3-bits, we will see what those 3-bits stand for this type of segment. And it also determines a segments privileged level there is 2-bits which 45 or 46 is talk about descriptors privilege level right DPL. Then there is an Accessed bit, as I told you whenever the segment is accessed, accessed means read or written to it is set by the processor that is accessed and this can be use by the operating system to do some more analysis of the data, how frequently something is accessed, etcetera.

(Refer Slide Time: 07:06)

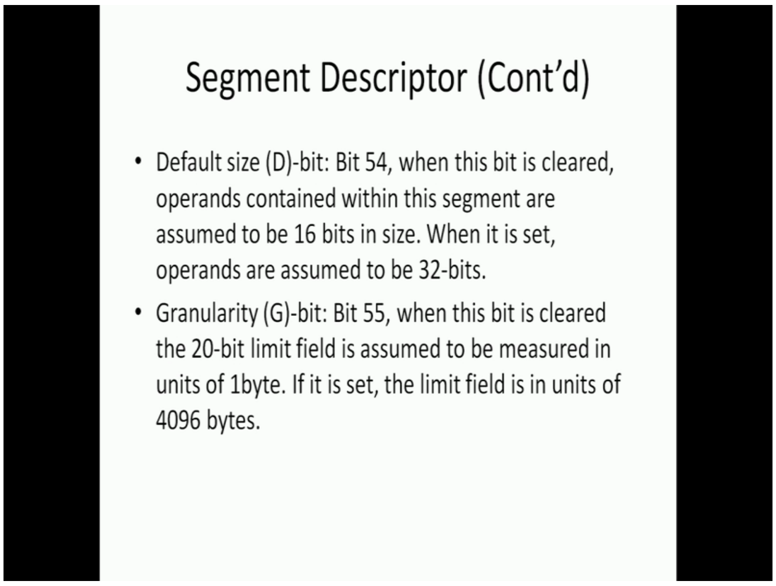


Segment Descriptor (Cont'd)

- Accessed (A)-bit: Bit 40, automatically set and not cleared by the processor when a memory reference is made to the segment described by this descriptor.
- Present (P)-bit: Bit 47, indicates whether the segment described by this descriptor is currently available in physical memory or not.
- Bits 40-47 of the descriptor is called the **Access Right Byte** of the descriptor.
- User (U)-bit and X bit: Bit 52 (U-bit) not used and Bit 53 (X-bit) reserved by Intel

Then there is Present bit as I told you, every not fully filled table should have a present bit to distinguish between valid and invalid entries. So, there is present bit here. And then there is bits 40 to 47 of the descriptor this are called Access Right Byte. Then we will see, what is the Access Right Byte, shortly. Then, there is a user bit and X-bit, bit 52 user bit which is not used and bit 53 is a X-bit which reserved by Intel.

(Refer Slide Time: 08:09)

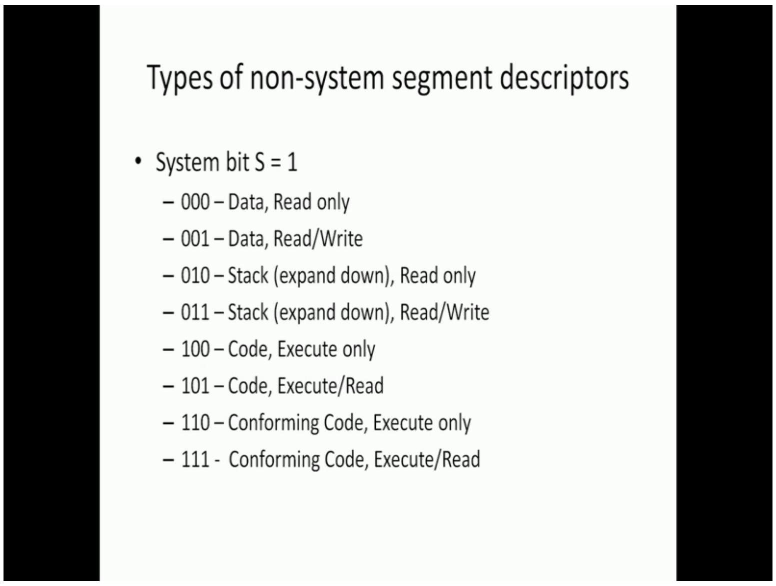


Segment Descriptor (Cont'd)

- Default size (D)-bit: Bit 54, when this bit is cleared, operands contained within this segment are assumed to be 16 bits in size. When it is set, operands are assumed to be 32-bits.
- Granularity (G)-bit: Bit 55, when this bit is cleared the 20-bit limit field is assumed to be measured in units of 1byte. If it is set, the limit field is in units of 4096 bytes.

Then there is default size is D-bit, bit 54 when this bit is cleared, that means operands containing within segment are assumed to be 16 bits in size. When it is set operands are assumed to be 32-bits in size. So this is very useful in code segment, when I compile a program for 16 bit then I make the code segment have D value as 0 here, that means it is a 16 bit code. When I make it 1 then it becomes a 32-bit code. So, this very, very important for me to execute legacy codes in this new step up, it begins extremely important. Then there is a Granularity or G-bit with 55, when is bit is cleared the 20-bit limit field is assumed to be measured in units of 1 byte. If it is set the limit field is in units of 4096 bytes. This is a Granularity bit. So, I could have a segment which is as small as 0 byte if it 1 megabyte with a limit as slow as 1 mega byte or as high as 4 gigabytes.

(Refer Slide Time: 09:33)

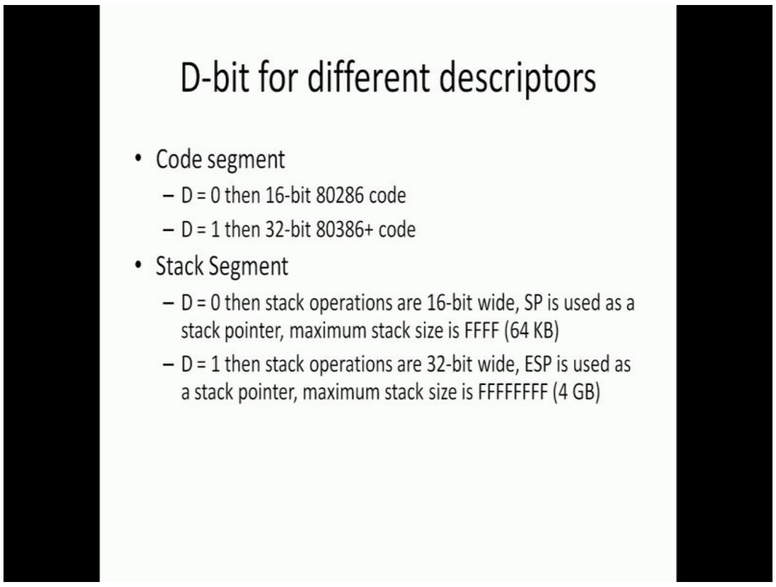


Types of non-system segment descriptors

- System bit S = 1
  - 000 – Data, Read only
  - 001 – Data, Read/Write
  - 010 – Stack (expand down), Read only
  - 011 – Stack (expand down), Read/Write
  - 100 – Code, Execute only
  - 101 – Code, Execute/Read
  - 110 – Conforming Code, Execute only
  - 111 - Conforming Code, Execute/Read

Now we will go to type of segment. This is S equal to 1 means they are non-system segment or user segment. So, if I put a type of 000, it is data, read only. 001 it is data, read write. 010 it is stack, expand down, read only. We will now explain what is stack, expand down and expand up. 011 stack, expand down, read write. 100 is the code, execute only. 101 is the code, execute with read. 110 is called a conforming code, execute only. 111 is conforming code, execute read. What is Conforming Code? We need to explain you down the line. And what is Stack expand down, expand up? We need to read that also, we need to specify that I will do it over the next few lectures. So, there is a type bit in this segment as you see bits number 8 to 11 on the top, that type bit is filled with these values for the different stacks.

(Refer Slide Time: 10:55)

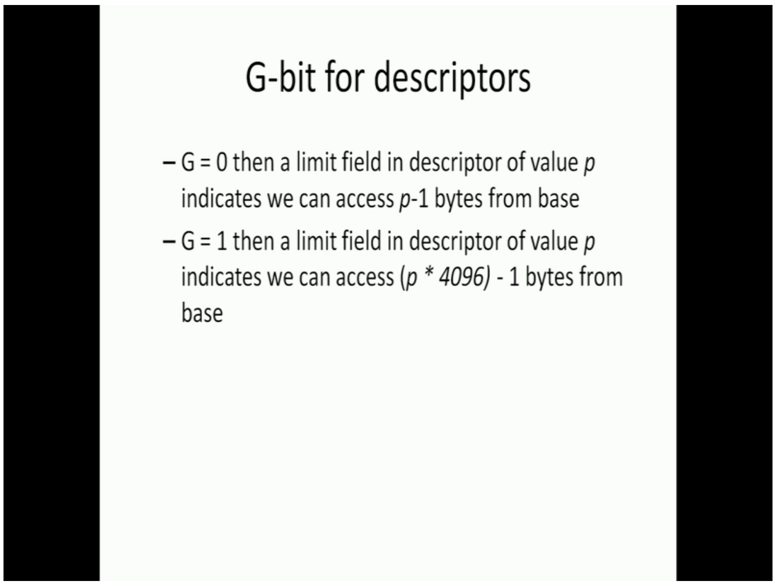


D-bit for different descriptors

- Code segment
  - D = 0 then 16-bit 80286 code
  - D = 1 then 32-bit 80386+ code
- Stack Segment
  - D = 0 then stack operations are 16-bit wide, SP is used as a stack pointer, maximum stack size is FFFF (64 KB)
  - D = 1 then stack operations are 32-bit wide, ESP is used as a stack pointer, maximum stack size is FFFFFFFF (4 GB)

For a Code segment, D equals to 0 then it is a 16-bit 80286 code as I told you, if a D equal to 1 then 32-bit 80386. For a Stack Segment if D is equal to 0, then stack operations are 16-bit wide. So, it uses only Sp not the ESP, SP is the 16-bit (Refer Time: 11:17). When D equal to 1 then the stack operations are 32-bit wide, so ESP is used as a stack pointer and you can go up to 4 GB of stack. These are the things that we need to understand.

(Refer Slide Time: 11:36)

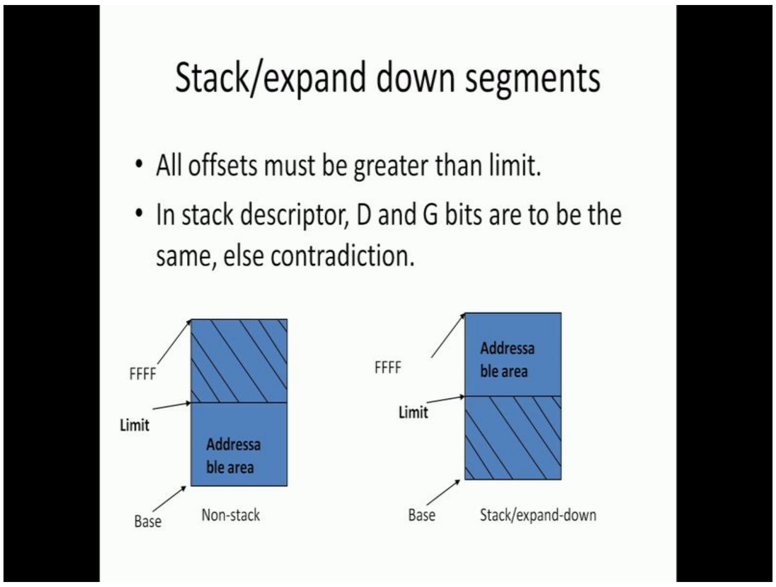


G-bit for descriptors

- G = 0 then a limit field in descriptor of value *p* indicates we can access *p*-1 bytes from base
- G = 1 then a limit field in descriptor of value *p* indicates we can access *(p * 4096)* - 1 bytes from base

And, G is equal to 0 then a limit field in descriptor of value p indicates we can access p minus 1 bytes on base, G is equal to 1 means then it is p into 4096 bytes the limit.

(Refer Slide Time: 11:51)



Stack/expand down segments

- All offsets must be greater than limit.
- In stack descriptor, D and G bits are to be the same, else contradiction.

Now, what is expand down and expand up segments? When a stack is expand down then all offsets must be greater than limit. In stack descriptor D and G bits are to be the same else contradiction. If I go and see a stack descriptor the D and G bits have to be the same. So, in expand down stack what happens? I fix a limit so your base is here this is the limit, if you just expand down stack your thing would be FFFF to the limit to be the addressable area. So, limit and above is expand down, limit and below it is expand up you are getting this. So, this is how I we can have stacks, and why do we need stacks? Because, I will say it is very easy for me to say you go and expand in this area. I can put a limit and say use beyond that limit and that is much more easy for me to say rather than putting a 2 limits and limit and limit and then. So, if I say use everything above that limit then it becomes easy for us to maintain the stack. That is why the reason, expand up and expand down stack segments have come.

To look at into all these things what we have done so far is that we have just understood in this part of the lecture some of the important things about the segment descriptors. Some understanding, a deeper understanding of the different fields that you see in this segment descriptor and that is what we have finally got to in this session.

.