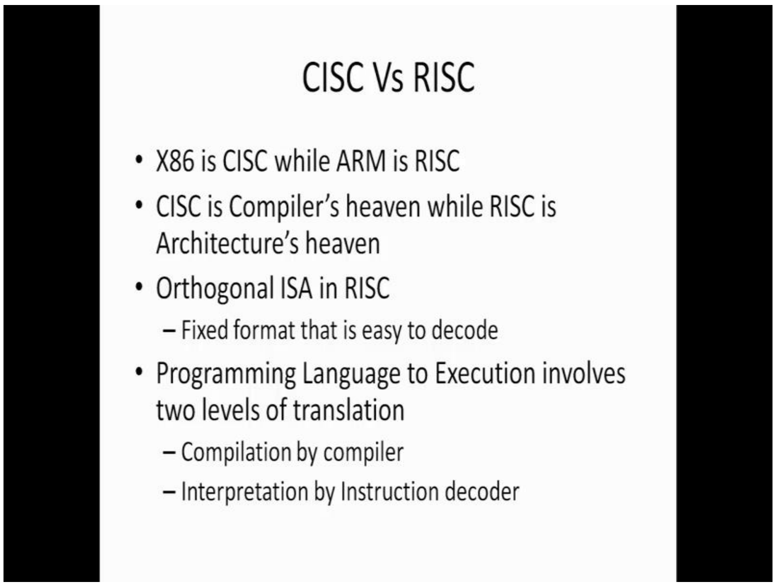
Information Security - II

Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 09
Architectural Aid to Secure Systems Engineering
Session – 8: X86 ISA – Part 1

We will do 2 sessions today, before we wind up. So, this current session is the x86 Instruction Set Architectural session. As I told the instruction set architecture is the interface between the hardware and the compiler. So, the compiler in the last session we saw there was C code and there was an equivalent machine language realization of the C code, who does this translation? The compiler, so the compiler needs to have knowledge about the instruction set architectural for it to translate from a C code to the corresponding instructions. Now, what we will be dealing now is the x86 instruction set architecture. What are all the instruction available in the x86 Intel hardware? And how do you use these instructions set architecture of this?

(Refer Slide Time: 01:13)



CISC Vs RISC

- X86 is CISC while ARM is RISC
- CISC is Compiler's heaven while RISC is Architecture's heaven
- Orthogonal ISA in RISC
  – Fixed format that is easy to decode
- Programming Language to Execution involves two levels of translation
  – Compilation by compiler
  – Interpretation by Instruction decoder

Let me give you some philosophy about instruction set architecture. There are 2 philosophies about instruction, one is what we call as the Complex Instruction Set

Computer and another is called the Reduced Instruction Set Computer, CISC versus RISC. The x86 that we are going to study now is a complex instructions set computer. The ARM which will be studying towards the end of the course is also is a reduced instruction set computer. In a reduced instruction set computer, I will be giving some simple instructions not really simple but limited amount of instruction, and then the compiler has to use these instructions into realize certain functionalities. In the CISC, I will give you a lot of instructions. CISC is a compilers heaven, but the architectures hell because compiler whatever it wants it has an instruction which is available so it can quickly translated, but the architectural will struggle to basically execute them, because there so many instructions which the architecture has to understand and execute. While RISC is architectures heaven, I have only simple set of instruction; compiler has to break it set.

Computer science is also communist science, what is the rationale of communism? You lose something you get something; you get something you lose something. That is the law of averages is very good. Nature for example is very, very communist, you start exploiting her she gives back. So, computer science is also like nature. You lose something you gain something; you gain something you lose something. So, in CISC what you gain? Compiler will be happy, but the architecture will be breaking it is head. In the RISC, architectural will be happy compiler will be breaking his head.

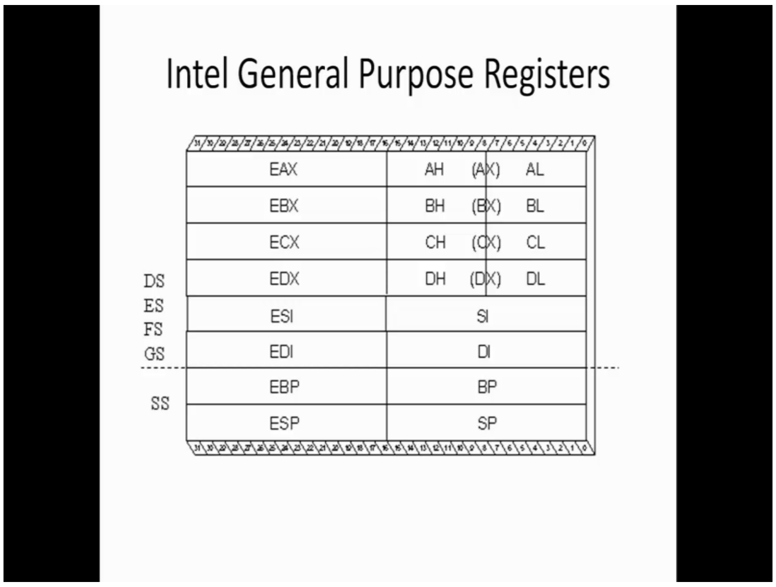
If you take courses in advanced architecture there is something called executing a CISC instruction in a RISC frame work. There is something called Code Morphing, we will not through this course it is an architecture course. But, we need to have an understanding between CISC and RISC though it does not have any impact on security, but a good understanding of it is much important from this point of view. Now, an instruction set is called an Orthogonal instruction set, if every instruction will have an operand and then which is called an opcode and an operand. See add, we said, right? Compare EAX comma EBX, compare is an opcode, EAX and EBX are operand. We had MOV EAX comma 0, so MOV is an operation or an opcode, EAX and 0 are operand. Now, there are 2 different types of operands, one is a register operand which is stored in the register like compare EAX comma EBX both EAX and EBX are register operands, but if we say MOV EAX comma 0, EAX is a register operands 0 is a memory operand. An orthogonal instruction set is 1, in which independent of the operation I can go and access any operand in any form, though there is no connection between the operands and the opcode. I cannot say for this particular opcode you should use only this type of operands, I cannot use some type of operand or when I use this opcode some operands are already implied by it. There are instructions will see as we proceed, If possible we will see, that there are instruction were some of the operands are already implied. For example, the string instructions in the x86 architectural, MOV string. MOV string is just a 1 byte instruction or 2, 1 or 2 bytes instruction, but then what will it MOV it will MOV a bunch of bytes from 1 location to another. So, the source location, the number of bytes will be in a register called ECX and where should I MOV will be in some EIX, EDX and something. So, there is lot of things. At least 4 to 5 registers have to be set before I execute this MOV set. This instruction depends upon the values of those registers which are implicit. Before I execute MOV SB have to go and set many registers and then execute this. That type of instructions set architecture is not orthogonal.

An orthogonal instruction set is 1 where my opcode I have to add and wherever that location I should be in a position to use it. So, the opcode is independent of the operand, you got this? What happens here? When I want to decode it becomes very, very easy from me to decode, if I have an orthogonal instruction set. Otherwise, which register should I go and fetch etcetera, etcetera. If it dependence, then I have to finish of the opcode and after seeing what the opcode is then I have to start fetching what the registers are. That means your decoding becomes much more complex in a non-orthogonal instruction set architecture. All the RISC architectures that we see especially ARM, they are all orthogonal instructional are set architecture to a large extent that is your opcode and the operand can be handled concurrently, where the operand lies is independent of the opcode.

Now, please understand that we have talked about that communist stuff right, the law of averages. So, now let us look at this when I look at programming language, the execution of the programming language there are 2 translations involved. The first translation is your C Program gets translated to machine language using your compiler that is an explicit translation in the sense you see the complied code. The next translation is your complied code is actually interpreted by your hardware and executed, so there is one implicit translation happening internally that is what people forget or books fail to explain. There is another translation that is happening, where the hardware accepts your instruction decodes it and then executes it. The first one is called Compilation, what is compilation? Take a code, completely translate it into hardware. What is Interpretation? take instruction by instruction execute it, then take the next instruction again execute it one by one, take an instruction decode it execute it next take another instruction decode and execute. So, in the first level translation is a compilation for us, the second level translation were each assembly instruction is taken by the hardware understand what does it trying to do and then executing which the hardware does, that is an interpretation. So, there is some common question you will be asked, what is the difference between compilation and interpretation? This is the difference between compilation and interpretation.

Compilation is done by the compiler; interpretation is done by the instruction decoder in your hardware. Please understand that there are 2 important levels of translation that happens. One is a programming language when I talk of a programming language to execution, so there is a compilation that is done by the compiler is 1 translation, there is an interpretation that is done by the interpreter, there is a translation done by the instruction decoder which is interpretation, OK.

(Refer Slide Time: 09:59)



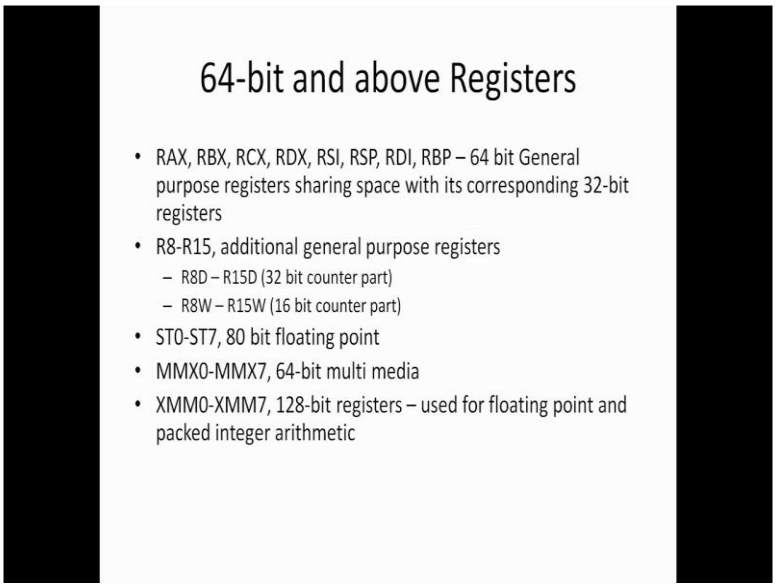
Intel General Purpose Registers

Now there is memory, accessing memory is very very time consuming. So, what I do from the memory? I take data and put internally do all calculation and once in a while go and update the memory, this is how the code gets compiled. Every time I do not go to the memory, from memory I take the data and put it inside the processor in some locations and use it and finally, I write it back to the memory. Those locations as called as General Purpose Registers. From now on, I will be actually talking about some of the complexities in Intel and let us try and appreciate. Now, Intel originally was 8-bit architecture, then it became a 16-bit architecture, then it has become a 32-bit now, 64-bit. In the 8-bit architecture there were 8 registers, registers are scratchpad memory I bring some the look from the memory I bring some value and I store it in the register. As you see in the slide there were 8 general purpose registers in the 8-bit architecture and they were AL, BL, CL, DL, AH, BH, CH, DH there were 8-bit registers.

Now, as it became a 16-bit architecture, what they did? They merged AL and AH to form AX. Today, when I say AX, AX is a 16-bit register. If I say AL, it is the first's 8-bits of that 16-bit register. If I say AH is a next 8-bit of that 16-bit register. Why I am doing this? Because I want to maintain legacy across code; still my original 80 85 code should run in today's context and that is the reason why have this sharing of registers. Similarly, BH, BL, CH, CL, DH, DL becomes BX, CX, DX. And then the 16-bit introduced 4 more registers namely SI, DI, BP and SP, where SP actually starts for the stack point. Now this 16-bit got extended to 32-bit, so they added 16 more bits. Once you see on the left hand side and name this new registers as EAX, extended AX, extended BX, extended CX, DX, SI, DI, BP, SP. So, the extended AX extended whatever EAX has the lowest.

There are 4 bytes in it the lowest of the byte is AL, the second lowest is the AH. The second and the first lowest is form what you call as AX of 16-bit and then the entire thing become CX. Why I need this type of general purpose register organization is specifically to maintain my legacy. What it means to maintain the legacy? It means to maintain the legacy from this point that whatever code I have written should still execute, and how much ever memory I have allocated at that point of time should be the one that is allocated. So, these are some of the reasons of why an Intel general purpose register this called a Register Field; because it has field collection of registers why should it be complex and what is the complexity is what you see in this file. In addition to this there are 6 general purpose registers namely CS, DS, ES, FS, GS, SS as you see on the left most side of this slide which are all different segment registers views. The segment register is one which stores the base of the segment as I explained earlier, and you have the 6 segment registers here.

(Refer Slide Time: 14:11)



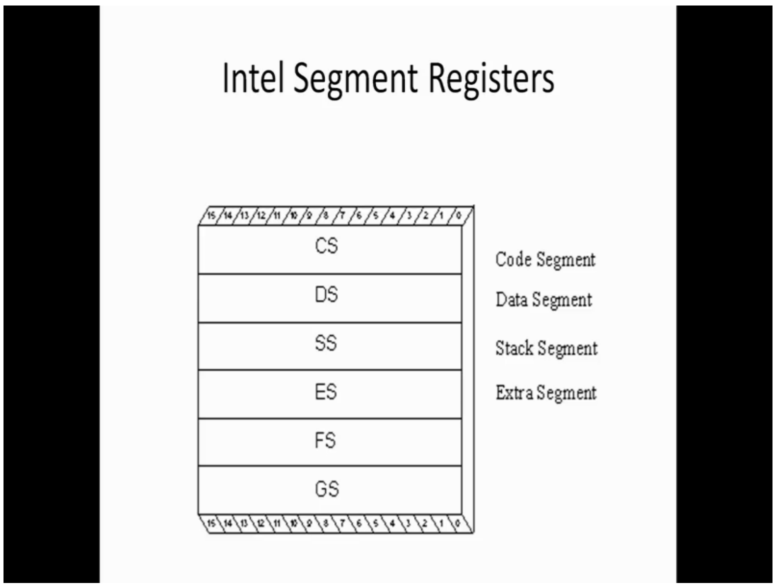
64-bit and above Registers

- RAX, RBX, RCX, RDX, RSI, RSP, RDI, RBP – 64 bit General purpose registers sharing space with its corresponding 32-bit registers
- R8-R15, additional general purpose registers
  - R8D – R15D (32 bit counter part)
  - R8W – R15W (16 bit counter part)
- ST0-ST7, 80 bit floating point
- MMX0-MMX7, 64-bit multi media
- XMM0-XMM7, 128-bit registers – used for floating point and packed integer arithmetic

Now, this particular 32-bit architecture has know becomes 64-bit and above. So RAX, RBX, RCX RDX, RSI, RSP, RDI, RBP these are all the 64-bit general purpose registers sharing space with it is corresponding 32-bit register. So, RAX is 64-bit in which the first 32-bits is called EAX, in which the first 16-bit is called AX, in which the first 8-bit is called AH and the next 8-bit is called AL. Similarly, I can go for EBX, ECX. In ESI, RSI is a 64-bit version of ESI in which the first 100 bits sorry, the last 32-bits will be ESI and the remaining first 32-bit will be the extension of that. So, in addition there were some more general purpose registers that were added which was R8D to R15D which was 32-bit counterpart and R8W to R15W which is the 16-bit counter. In additional to these 8 general purpose registers there were 8 more 32-bit registers and 16 more registers at a general purpose registers are added. So, the 32-bit are R8D to r15D and 16-bit are R8W to R15W. Then you have floating point registers 8 of them ST0 to ST7. Then you have multimedia registers which are 64-bit.

As I told you, the multimedia operations involve bit (Refer Time: 16:07) I can do 64-bit in 1 shot and that improves my graphic speed etcetera. And that is why is called multimedia instructions. Then packed integer and floating point arithmetic there are 128-bit registers which are named XMM0 to XMM7, and these are the 128-bit registers.

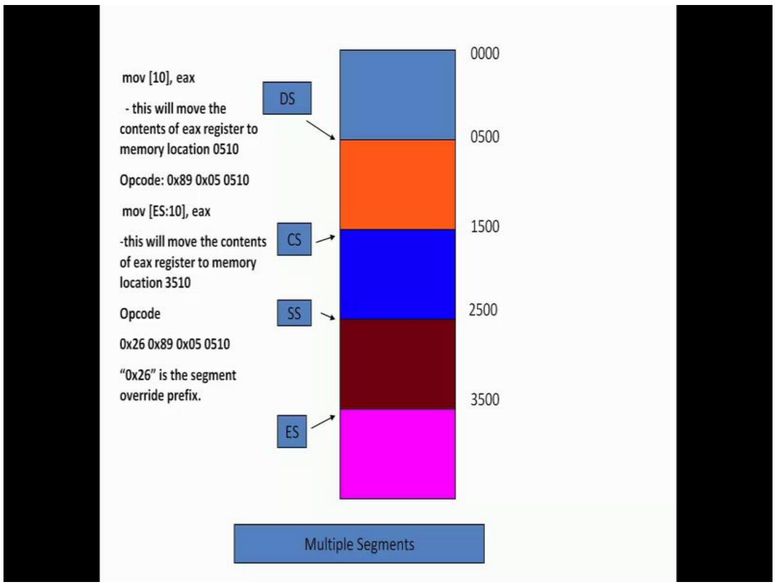
(Refer Slide Time: 16:29).



Intel Segment Registers

Then there are 6 segments register namely CS, DS, SS, ES, FS, GS. The CS is for the code segment, DS is for data segment, SS is for the stack segment and then there are three additional segment registers ES, FS, GS which are the extra segment registers.

(Refer Slide Time: 16:59)


mov [10], eax

- this will move the contents of eax register to memory location 0510

Opcode: 0x89 0x05 0510

mov [ES:10], eax

-this will move the contents of eax register to memory location 3510

Opcode

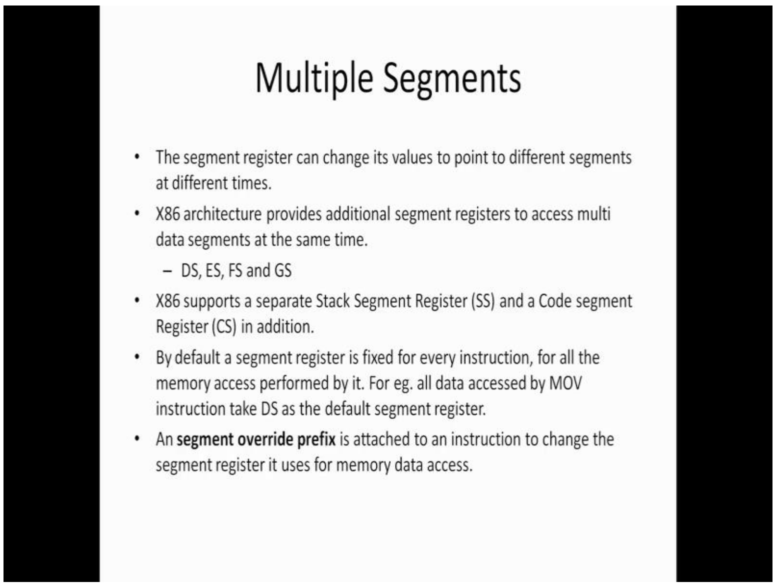
0x26 0x89 0x05 0510

"0x26" is the segment override prefix.

I will just give you this code. Now, in this set up there are 4 segments namely, DS 1 pointed 2 by DS, 1 pointed 2 by SS and 1 pointed 2 by ES. So there are 4 segments in which 3 are data segments, 1 is DS another is ES another is SS. SS is essentially a stack segment which is a data segment because data has stored in the stack. And there is a another segment called CS which is the code segment which points, so as you see DS is pointing to 500, SS is pointing to 2500, ES is pointing to 3500 while your CS the code segment is pointing to 1500. Now when I say mov 10 comma EAX, what will happen? The value stored where DS stored, DS is stored in 500. So, the value in 510 the value of EAX will be moved to the 510 memory address. And the opcode which of compiling mov 10 comma EAX is 0x89, 0x05, 0510. These x are decimal 89 5 5 10. This is the opcode for if I compile mov 10 EAX, this is the opcode I get which I give it to the architecture for executing. Now every time DS is added, but now I have 2 more segments namely SS and ES, what is SS, what is ES? They are also data segment. How will I access data in that SS and ES? If I say mov 10 comma EAX it is only using DS, so the instruction say mov ES colon 10 comma EAX if I put that ES then it will no access from where is ES pointing to 3500, it will now start accessing from 3510.

So, I can access from any of the data segments not necessarily only DS, but ES. SS, FS, GS. By just adjusting that instead of DS if I put default it is DS, but if I put ES colon then it comes from ES. When I go and compile this the opcode is again 0x89, 0x05, 0510. This is the same opcode, but then in addition I have 1 prefix which is called 0x26 which tells for this instruction it is a mov instruction ok, this is a mov instruction but mov it from what ES, use the segment register ES and not DS. That 0x26 in front of 0x89, 0x05, 0510 basically tells me what mov it from 0x26 move it from ES, use the ES at the segment register rather than SS or DS. So, this is the basic thing.

(Refer Slide Time: 20:39)



Multiple Segments

- The segment register can change its values to point to different segments at different times.
- X86 architecture provides additional segment registers to access multi data segments at the same time.
  - DS, ES, FS and GS
- X86 supports a separate Stack Segment Register (SS) and a Code segment Register (CS) in addition.
- By default a segment register is fixed for every instruction, for all the memory access performed by it. For eg. all data accessed by MOV instruction take DS as the default segment register.
- An **segment override prefix** is attached to an instruction to change the segment register it uses for memory data access.

So, with this as a back ground let us quickly go through this slide. I have multiple segments, I need multiple segments for some reason I will ask at the end of this slide, I will ask you why do we need multiple segments, let us see how many of you answer that. The segment register can change it is value to point to different segments at different times. So, x86 architecture provides additional segment registers to access multiples data segments at the same time like DS ES FS and GS. X86 supports a separate stack segment register and a code segment CS in addition to this. So, there are 6 segment registers.

By default, a segment register is fixed for every instruction, for all the memory access performed by it. For example, all data access by mov instruction takes DS as the default segment register, Now, if I want to change that DS to ES then I put something called a segment override prefix and if I use the segment override prefix then your DS becomes whatever other segment you want. The 0x26 is a segment override prefix, why it is called prefix? Because, it is happening before the instruction. And why it is segment override? Because, it changes the segment which you want to add. So, by using segmentation I have multiple segments and I can access all these segments as a part of the code. Why do I need multiple segments? Why do I need multiple data segments? I need multiple segments one for code, one for data, one for stack. Why do I need multiple data segments? Can you just go back to your C programming fundamentals and tell me why do I need?

Student: (Refer Time: 22:23)

No, why do I need multiple data segments?

Student: To perform same operations (Refer Time: 22:43) different data.

No, I can put it in the same thing and the 1 data segment I can put all the data and access it no. Why should I need multiple, can you give me one reason?

Student: (Refer Time: 22:54) Contiguous space.

Contiguous space will not be available. Why contiguous space will not be available? See everything is in the stack, all the local variables are in the stack, so everything is with respect to the stack segment itself I can access it.

Student: Usually, you need access a multiple data to the same instruction.

Multiple data with the same instruction, but they will all be within 1 data segment right. Why do I need multiple data segments? Can you go to some programming construction and give me an answer? Programming constructs with.

Student: Separate the input data is from the process.

No, there is nothing like separation when I give a data to the in process it becomes process data, right?

Student: (Refer Time: 23:54) inputs may be used with some other program.

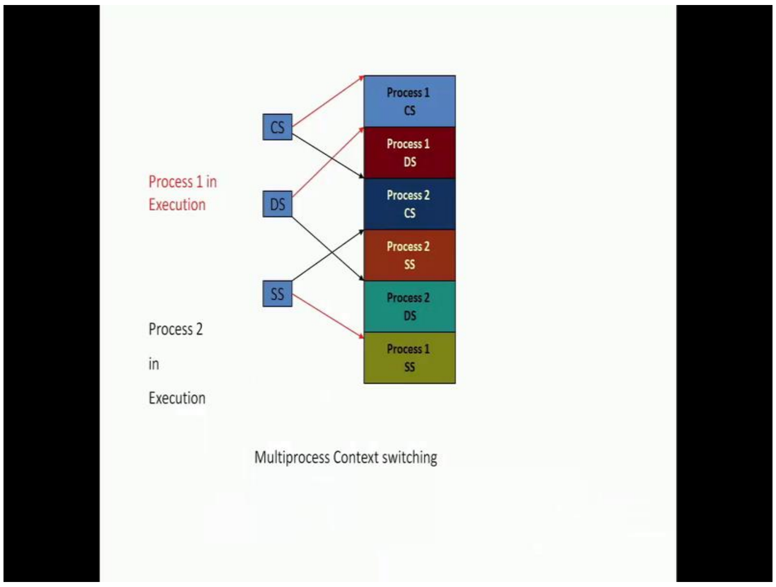
Then it is for that data segment. Why in a particular in as a single program? Why should I use 2 data segments? One thing is clear that we need contiguous space. Yes, I agree that we need contiguous space; so, because of that what? Please note that we have a call melock, what does melock do? It creates new set of addresses. Now, when I do a melock, I have already a data segment given to me. When I compile I know these are all the data I need, so there is a data segment given in which all data I could realize or I could interpret at the time of compilation. I say I have 100 integers in this program, so 400 bits are given to me, but then there is something like melock which I do not know.

So, when I execute a melock what happens, another set of data I need to allocate say another 1,000 bits of data that I will know only dynamically. Now that, for that 1,000 bits I cannot have it contiguous because all data should be contiguous, the segment should be contiguous memory locations I may not have contiguous memory location to store that 1,000 bits. So, what I do? I create another area and put 1,000 bits there and use another segment register to point it. If I want to access that data, which is got out of this melock then I use that particular segment. This is one reason why I need multiple data segments, because there is a necessity for us to do dynamic memory allocation and when I do dynamic memory allocation, I need lot more data which I may not get it in consecutive location. Else what I should do to get it in consecutive location, I should push data away and creates space. Every time I execute a melock can I do your garbage collection, I cannot do a garbage collection because it is garbage collection is really time consuming.

Sometimes you boot your system it takes lot of time for you to know basically execute right suddenly becomes slow, what it does it? Does lot of garbage collection, that is why even in normal you know big services like your financial services they advice at regular intervals shut down your server and then reboot it, why? Because sometimes suddenly what happen in the mid of the day when real business is happening suppose your 640 gigabyte server starts garbage collection right now gone, will kill your performance. So, that is why they say no at regular occasions please do garbage collection and keep your garbage away. That is one of the reasons why I need; why I am giving in this as a case study here, because I give this I believe that we should discuss about this here because we need to have a clear mapping of why certain things are there in the architecture. That is one big difference in the education that we see everywhere now.

Lot of things, how it is done? What is done and how it is done? is very very clearly explained, but why it is done is not basically explained. And you as teachers or students or (Refer Time: 27:42) I think it is very, very important that you understand why certain things are there right. Please keep asking why? Why? Why? Why? Why? And that will give you lot more answers and for information security it is why that is going to solve the problem, you should question everything that comes there and try to get an convincing answer for that.

(Refer Slide Time: 28:08)



Student: Sir (Refer Time: 28:10)

Yes.

Student: (Refer Time: 28:13) have single index and if you want to access something different (Refer Time: 28:19) in the case of string copy or something we may need multiple segment.

Exactly.

Student: (Refer Time: 28:28)

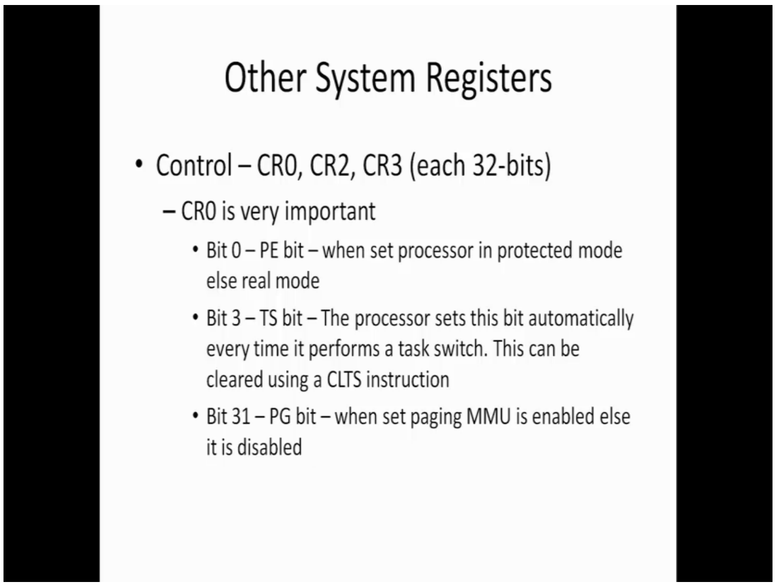
Exactly, for example, if you go to the Intel developer manual which I want you to go and start looking at this evening, and probably have it as a bed time story today for all the attendees. There are instructions called mov SB, mov SD, mov SW comma mov string bytes, mov sting word, mov string double word. All these mov strings are basically to copy, they are basically intended to do mass copying from 1 set of memory locations to another and they had been introduced in my opinion. I do not know we have to ask the instructions at architect of Intel to get an idea of that, but I think Intel came out with that first. I think it was done with a view to do process migration specifically from a point of view know either memory banking and performance issues when I go from one process to another, and I have a non-uniform memory architecture things like that would have been very useful and garbage collection from a operating system point of view.

There when you look at that instruction, there is source and there is a destination block. The source block is given by one segment register and a counter, he says in this block from this address which is another register and this many bytes that is a counter move it to this segment this block pointed to by this segment register and within that segment this address offset. So, there are 2 segment registers, 2 offset registers within that segment and 1 counter involved in each of this instruction implied, so that mov SB is just 1 byte, but then there are 6 register, 5 registers used 2 segment registers, 2 general purpose registers, 2 for telling the offset and 1 for giving the byte. So, what Vidya told is correct, that we will also use this multiple segments to move chunks of data from one part to another part. Specifically, this is needed in the case of process migration and problems like garbage collection that we see.

So, before we wind up the session the next important thing that segmentation will teach us is this Multi-Process Context switching, please understand this is very very important. We have multi all the servers today that are working or multi user operating system. Even our mobile phone has multi-process is running there. It is not just user even a single user can spawn multiple processes, so this is a multi process operating system. That means, several processes will be coexisting in the same RAM memory at the same point of time. Now, what will happen is that, please note this is the memory that we have shown here that is CS that is there are 2 processes, process 1, process 2 and each process has a code segment, a data segment, and a stack segment.

So, EC 6 segments here process 1 CS, process 1 DS, process 1 SS, process 2 CS, process 2 DS and process 2 SS the segments need not be consecutive, but internally each segments should be consecutive contiguous. It is not necessary that CS, DS and SS all should be in contiguous locations, but CS individually should be contiguous, DS individually should be contiguous etcetera. Now when process 1 is executing the CS, DS and SS will point to that, when process 2 is executing it will now point to that (Refer Time: 32:33). So, what I mean by switching from process 1 to process 2 is to just go and change the value of that segment register and immediately my context switch happens. I need not do anything else. So, these are all some of the important things that come out of this segmentation. We have already seen that there is ease of compilation, there is a process movement, process mobility is ensured, then in the previous slide we saw that we could have multiple segments and we also motivated several things for that and then now we could have what we call as this multi process context switching.

(Refer Slide Time: 33:19)

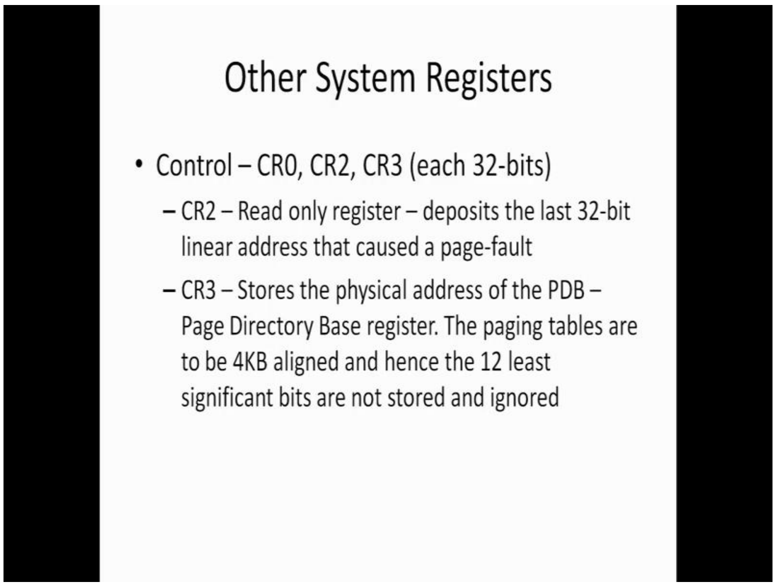


Other System Registers

- Control – CR0, CR2, CR3 (each 32-bits)
  - CR0 is very important
    - Bit 0 – PE bit – when set processor in protected mode else real mode
    - Bit 3 – TS bit – The processor sets this bit automatically every time it performs a task switch. This can be cleared using a CLTS instruction
    - Bit 31 – PG bit – when set paging MMU is enabled else it is disabled

In addition to this, the x86 architecture basically has controlled registers. There are 3 controlled registers that are importance as of now. Each of these control registers are 32-bit in length, the CR0 is very important. The CR0 has some very interesting bits. The bit 0 is one which will make the processor to protected mode. The bit 3 is one which will indicate whether there is a task switching that is happening. The bit 31 is one where you enable paging. So, there are 3 bits, please remember these bits that we will be using them extensively down the line; bit 0 is the protected mode enable bit; bit 3 is a task switch you will understand whether a task switch has happened or not and bit 31 is a bit which will enable virtual memory in your stuff.

(Refer Slide Time: 34:19)

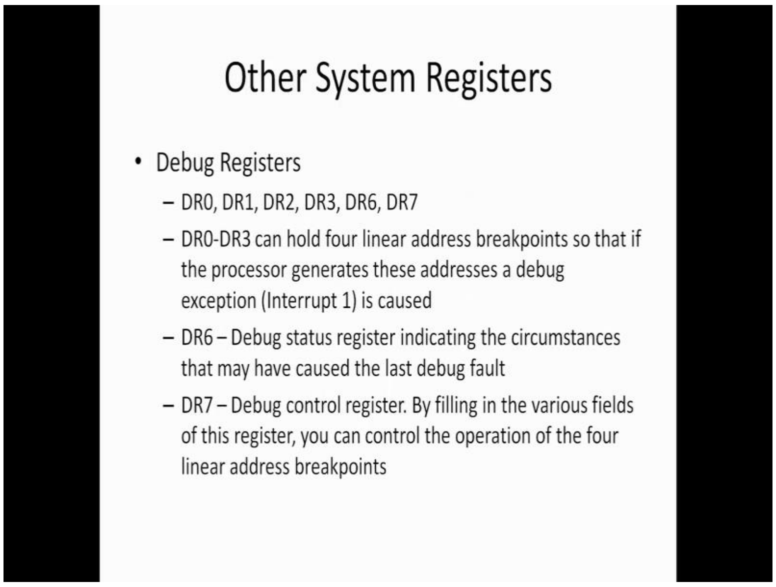


Other System Registers

- Control – CR0, CR2, CR3 (each 32-bits)
  - CR2 – Read only register – deposits the last 32-bit linear address that caused a page-fault
  - CR3 – Stores the physical address of the PDB – Page Directory Base register. The paging tables are to be 4KB aligned and hence the 12 least significant bits are not stored and ignored

The CR2 is a read only register from the users prospective; it will give you which was the last address which created a page fault. And, the CR3 will page fault is for paging for virtual memory and the CR3 will store the page directory address for the paging. Just remember that, CR2 will give you address for page faults, CR3 will store the base address for directory. And as we go on to paging etcetera, we will be using these controlled registers.

(Refer Slide Time: 34:55)



Other System Registers

- Debug Registers
  - DR0, DR1, DR2, DR3, DR6, DR7
  - DR0-DR3 can hold four linear address breakpoints so that if the processor generates these addresses a debug exception (Interrupt 1) is caused
  - DR6 – Debug status register indicating the circumstances that may have caused the last debug fault
  - DR7 – Debug control register. By filling in the various fields of this register, you can control the operation of the four linear address breakpoints

In addition there are several registers basically DR0, DR1, DR2, DR3, DR6, DR7. Then DR0 to DR3 can hold 4 linear addresses breakpoints, so you do that GDB, right? You say break at this, so where we want to break? I can put those addresses here and when you reach those addresses automatically the architecture will do a break and it will transfer it to a interrupt service routine in which will serve that break. When you break then you go and query what is the variable address etcetera, variable value etcetera. Then there is a DR6 which a debug status register indicating the circumstances that may have cause the last debug fault. And DR7 is a debug control register by filling in the various fields of this register you can control the operation of the 4 linear address breakpoints. Lot of things can be done here, so these are registers.

So why are these control registers very important to realize certain functionalities? To enable and disable certain functionalities; the debug registers are gives you much more insight into the working of the program. So, later part when we want to develop certain utilities that will go and monitor a program I think these are the registers, at least the debug register or once which can be extensively used. Down the course not only this course but the subsequent courses we may try and use these debug registers. So, with this we end this session.

Student: (Refer Time: 36:31) I was asking about, I mean you asked us question mark why we should have multiple segment registers.

Ok.

Student: (Refer Time: 36:40) saying.

Yes, right. So, yeah now I remember. So the thing is that, if I want a very huge segment, for example if I take a 32-bit architecture I need segment which is as large as 4 gigabyte. Now, if you look at the segment size it is the segment there is something called at segment descriptor as we go down we will see that. The segment at descriptor is only 20-bits in size. If I say it is 20-bits that mean I can; what the 20-bits? 1 MB, but then Intel because it is a complex instruction set it has to prove it is complicity every time. IT has 1 bit called a Granularity bit in the segment descriptor, if I set that bit to 1 then every integer that I store should be multiplied by 4 KB. 4 KB is 12 bits. So, if that that bit is 0 and I put limit as 10, it is then 10 bytes. If I make that bit 1 and store 10 here it is 10 into 4 KB that is 40 KB. If I make that bit 1 and I store all the 20 bits, that means it is 2 power 20 into 12, 2 power 12 which is 4 GB. So I could have a segment which is as large as 4 GB or as small as 1 MB meaning in terms of limits.

Now, this is how Intel has addressed large segment versus small segment. But this is sort of very binary because 1 MB to 4GB is too huge as space. If I say it is 4GB then what happens, essentially the entire address page is available to me. If I say 1 MB is very restrictive. Why I am answering Vidya's question in this big way is that you understand this has become a very big problem, if you look at some of the OS none of the OS actually use segmentation for protection. The OS where we have seen the source code available, if you carefully look at the source code the operating system does not use segmentation for protection. The operating system uses paging for protection. Why it does not use segmentation for protection is? Since it does not use segmentation for protection it just creates 14GB memory, 14GB segment and say it is open to all. Anybody can do anything because it is 1 least privilege level 14GB is given. The reason I also thought why is this reason because, it has either segments of size 1 MB or it has segments of size 4GB and this is too restrictive for. Because today, program can have segments which are more than 1 MB in size, the execute doubles more than 1. And maybe this is the reason why you know Intel type of this set of feature is not being used in practice.

So one reason yeah, this could also be a reason Vidya well I wanted 2 MB memory segment I cannot offered to create a 4 GB for that, so I will use 2 segment registers to create 1 MB plus 1 MB rather than you know 14 GB segment so that could also be a valid reason. And this is also a reason why segmentation is not being used you know prevalently in for protections. But segmentation essentially has so much good things and we can get lot of process isolation done through segmentation, probably because of this particular reason.

The second reason would be portability. Each architecture has it is own wimps fences about segmentation. So I cannot have a compiler compile for the segment (Refer Time: 41:02) I need to have an operating system so I make a Linux, I cannot make Linux for Intel Linux for this. I want 1 Linux which is portable. So, people did not try and do things for each architecture specific things, and that is also precisely some of the reason why you know, the operating systems do not use these hardware infrastructure that is available. And that it does not use these hardware infrastructures positively and that is precisely, some of the reasons why you miss some of the security features that are made available. This is a sort of an elaborate answer, that is a wonderful question and I think this answer throws lot of light on architecturally aid to security.