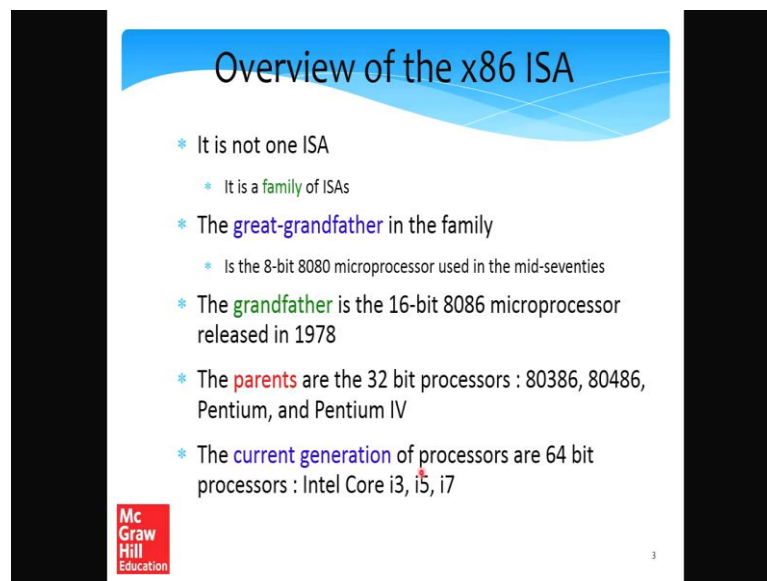


Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 10
Assembly Language Part-1

Welcome to the chapter on x86 Assembly Languages. So, as we shall see x86 is just not one assembly language, it is a family of assembly languages. They are used by the Intel and AMD processors which at least as of 2016 dominate the laptop, the desktop and the server processor markets. So, these slides are for chapter-5 in the book computer organization architecture published by McGraw Hill in 2015. The book should be available in almost all geographies around the world, you can always let the author or the publisher know, if a book is not available in certain geography.

(Refer Slide Time: 01:43)



Overview of the x86 ISA

- * It is not one ISA
 - * It is a family of ISAs
- * The great-grandfather in the family
 - * Is the 8-bit 8080 microprocessor used in the mid-seventies
- * The grandfather is the 16-bit 8086 microprocessor released in 1978
- * The parents are the 32 bit processors : 80386, 80486, Pentium, and Pentium IV
- * The current generation of processors are 64 bit processors : Intel Core i3, i5, i7

3

So, let us come to this very interesting family of ISAs. So, the first point is that it is actually not one ISA it is a family of ISAs, but all of them end with a x86. So, that is the regions x86, where x can mean a lot of things. So, the great-grandfather in the family is the 8-bit 8080 microprocessor used in the mid-seventies and say the 8080 microprocessor used to have an assembly language of its own, it was a 8-bit processor. The grandfather is the 16-bit 8086 microprocessor which was released in 1978; and for those days, it was a fairly revolutionary microprocessor because 16-bit was also there in

those days; and the microprocessor pretty much started the era of microprocessors and gradually large server computers started being built of much smaller microprocessors. So, the parents in this generation are the 32-bit processors namely 80386, 80486, Pentium 1 till Pentium 4.

So, all of these the processors used to dominate in the 90s and early 2000. So, pretty much for 20 years these processors dominated. Finally, Intel and AMD, so they move to 64-bit processors, so again 32-bits or 64-bits basically means that the smallest unit of storage inside the processor is either 32-bits or 64-bits. Alternatively, this means that the size of a register is 32-bits in a 32-bit processor, and it is 64-bits in a 64-bit processor. And even you know the size of memory addresses that can be issued by the processor and so on in one case is 32-bits, in other case is 64-bits

So, examples of 64-bit Intel processors should be Intel core i3, i5, and i7 the latest Intel processors that we use and similarly examples of AMD machines would be AMD opteron and AMD bulldozer which are the latest AMD processors that are used. So, the main features, so again why is the name of this architecture x86, because as you can see if in a previous slide there are lot of micro processors with the suffix 86, 8086, 80386, 486, Pentiums, Intel core i3, i5, i7, so all of them pretty much use the same kind of assembly language. And the nice thing is that assembly language for i5 is also a super set of the assembly language of the earlier micro processors, so which is great. So, since 86 is at the end of every assembly language of processor line at least till 486, the name of the ISA per se is x86.

(Refer Slide Time: 05:02)

The slide features a blue header with the title 'Main Features of the x86 ISA'. Below the header, there is a list of five bullet points, each starting with an asterisk. The text is color-coded: 'CISC' is red, 'instructions' is green, 'Instructions' is teal, 'stack' is black, 'return addresses' is black, and 'segmented' is blue. At the bottom left of the slide is the McGraw Hill Education logo, and at the bottom right is a small number '4'.

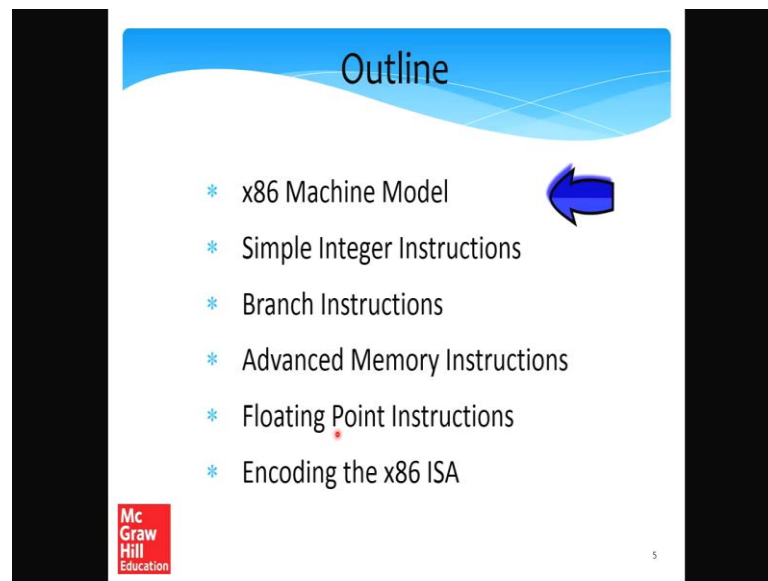
- * It is a **CISC** ISA
- * Has more than 300+ **instructions**
- * **Instructions** can have a source/destination memory operand
- * Uses the **stack** for passing arguments, and return addresses
- * Uses **segmented** memory

Say x86 is the CISC ISA see unlike RM and simple RISC. So, it has 300 plus instructions and so that is the basic ISA at least the one that we are concerned with, but if you consider all extensions and so on then out then as of 2016, the number of instructions are roughly a 1000. So, instructions can have a source or a destination memory operand. So, it is a CISC instruction set. So, the operands themselves can be far more complicated and instead of so the numbers of registers are actually few and another major difference as compared to simple RISC is that it uses the stack for passing arguments and return addresses unlike registers. So, in other RISC ISAs with a lot of registers, so we were using the registers for passing arguments and return addresses and similar kinds of information, but this does not happen over here, and it uses a different memory model called segmented memory.

So, we shall look at all of these points in detail over the next few slides, but an important point that I want to make at this stage is that the x86 ISA is per se and advanced ISA. See if you are reading this chapter or listening to this video without a basic understanding of assembly languages, you will find it somewhat difficult, because as I have been stressing all through walking in chewing gum at the same time or trying to understand a certain assembly language as well as the fundamentals of assembly languages. Both are not possible at the same time. So, it is possible to do only one, but not the other.

So, what my suggestion would be is that readers can go to chapter three, get a broad idea of what assembly languages are, they need not read chapter four which is about ARM assembly languages, so that is not required. But at least my assumption at this stage would be that for all the readers or listeners who want to move forward they have a basic idea of what assembly languages are. So, this will really help them, otherwise they will find this chapter very difficult to understand because the fundamentals would pretty much not be there yet.

(Refer Slide Time: 07:30)



So, the way that the chapter will proceed is that first we will discuss the machine model, then we discuss simple integer and branch instructions. X86 has fantastic memory instructions or advanced memory instructions, so we will discuss that we will discuss floating-point instructions because floating-point instructions are fairly complicated in the x86 ISA. And lastly as we did for ARM and simple RISC, we will discuss methods to encode the x86 ISA.

(Refer Slide Time: 08:03)

The slide is titled "View of Registers" and features a blue header with a wavy pattern. The main content consists of four bullet points. The first bullet point states that modern Intel machines are still ISA compatible with the 16-bit 8086 processor. The second bullet point explains that due to market requirements, a 64-bit processor must be ISA compatible with all 32-bit and 16-bit ISAs. The third bullet point asks what to do with registers. The fourth bullet point asks if a new set of registers should be defined for each x86 ISA, with the answer being "NO". The McGraw Hill Education logo is located in the bottom left corner of the slide, and a small number "6" is in the bottom right corner.

View of Registers

- * Modern Intel machines are still ISA compatible with the **arcane** 16 bit 8086 processor
- * In fact, due to **market** requirements, a **64 bit processor** needs to be ISA compatible with all 32 bit, and 16 bit ISAs
- * What do we do with **registers**?
- * Do we define a new set of **registers** for each type of x86 ISA? **ANSWER : NO**

McGraw Hill Education

6

So, let us take a look at the view of registers. Modern Intel machines are ISA compatible with even the oldest 8086 processor. So, what this essentially means that any kind of program which was meant for the 8086 processor can still run on a modern processor. So, this is mainly done due to market requirements. So, market requirements suggest that a 64-bit processor has to be ISA compatible, what means that is a compatible is that program is written for earlier processors, older processors which had 32-bit and 16-bit ISAs should still be able to run with a 64-bit processor.

And a market requirement is as follows. Let us assume that a certain company has a lot of programs written for 32-bit processors, and because they need some additional performance, they buy a newer processor from Intel. And this newer processor is a 64-bit ISA. It would be a very bad idea for the company to discard all of its code, and essentially rewrite or recompile the code for the newer processor. There are several reasons for this; first is that the original developers might be not there, the source code might not be available, it might be a lot of work, and there are issues with robustness and reliability of the code.

So, because of so many issues, most companies most customers of Intel and AMD would want that whatever code they had that should still run with the advanced processors, advanced processors can add more features, but they should not break something which was already running. So, keeping this requirement in mind both Intel as well as the AMD

ensures that the latest processors can still run code, which was written we can still run let us say a binary which was written 30 years ago.

So, the question is what do we do with our set of registers, do we have a separate set of 16-bit registers a separate set of 32, and a separate set of 64-bit registers, is this good idea? It is probably not a good idea, because we do not have that much of space inside the processor and because of performance constraints it is not what we should be doing. So, we should definitely not define a new set of registers for each kind of ISA and that is a bad idea so we have to look for a better idea.

(Refer Slide Time: 10:43)

The slide is titled "View of Registers - II" and contains the following text:

- * Consider the 16 bit x86 ISA – It has 8 registers: ax, bx, cx, dx, sp, bp, si, di. *Handwritten notes: "stack pointer" under sp, "source" under si, "destination" under di.*
- * Should we keep the old registers, and create a new set of registers in a 32 bit processor? **NO** *Handwritten note: "Registers Functional Unit" in a red box.*
- * **NO** – Widen the 16 bit registers to 32 bits.
- * If the processor is running a 16 bit program, then it uses the lower 16 bits of every 32 bit register.

Mc Graw Hill Education logo is visible in the bottom left corner.

So, let us consider the 16-bit x86 ISA, is the 16-bit ISA had 8 registers they were called ax, bx, cx, and dx, sp, bp, si and di. So, actually the idea of these registers had actually come from the 8-bit processor whose registers were of the form a b c and d, but in any case when they made a 16-bit processor they named the register. So, which was way back in nineteen seventy 8 ax, bx, cx, dx, sp, bp, si and di. So, sp is the stack pointer. So, recall from chapter three that the idea of the stack pointer was introduced stack pointer bp is actually the base pointer.

So, we will discuss what it is. So, essentially when we enter a function we save the initial value of the stack pointer in the base pointer. So, we will discuss this when we come to how functions are implemented in x86 then we have the registers si and di say si actually stands for source and d for destination. So, this will be clearer when we discuss

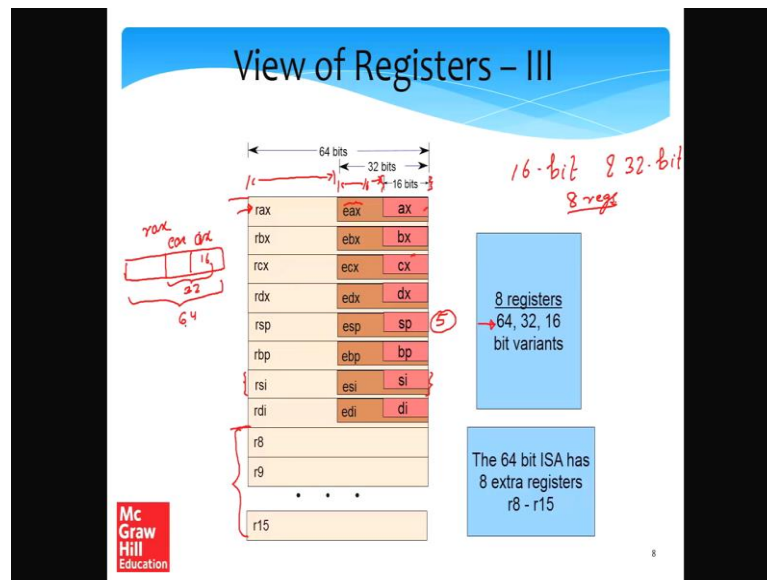
instructions that use the si and di registers with you know you know in a special form. So, for the time being it is sufficient to just simply remember that s stands for source and d for destination.

So, in the 16-bit processor, we had only these 8 registers and see if you would recall both ARM and simple RISC add more registers at 16-bit x86 in that sense is fairly conservative in terms of the number of registers in 8. We have four you know general purpose registers ax, bx, cx, and dx. SP has a special meaning it is a stack pointer; BP also has the special meaning it is called the frame pointer or base pointer depending upon how it is implemented. And si and di can be used for regular computation that they have a special meaning. S stands for source and D for destination. The some instructions assume that si is a default source and some other instructions assume that di is a default destination.

So, we shall look at these instructions later in this lecture. So, the question is should we keep the old registers and create a new set of registers in a 32-bit processor we have already realized that is a bad idea that is not what we should be doing the main thing being that the area in which the registers are there, it is fairly cramped. So, if you consider that area of the processor you have registers you will have functional units like adders and multipliers and you will have many other kinds of structures. So, you can consider it just think of it as a downtown or a processor.

So, this particular area is very, very cramped. So, it is not a good idea to sort of increase the set of registers. So, this is definitely a bad idea. So, what we can do is that we can widen the set of 16-bit registers to 32-bits. So, we shall see in a next few slides how this is done. So, if the processor is running a 16-bit program then it can sort of think that every register is 16-bits wide and how will it think it will use the lower 16-bits of every 32-bit register to essentially saves its value.

(Refer Slide Time: 14:27)



So, let us take a look at this particular figure. So, this figure is very important we will be keep on referring to this figure. So, is very important that readers get a very, very good idea of this figure before proceeding. So, let us first can just consider the first 8 registers which are there in the 64, 32 and 16-bit variants see in these registers let us assume that the processor is running a 16-bit program. So, the registers that it sees are ax, bx, cx, dx, sp, bp, si and di. So, ax is actually the lower 16-bits of the eax register. So, consider e as the extended ax. So, thus the eax register is a 32-bit register which is seen by a program when it is running in 32-bit mode. So, essentially it has we can divide a 32-bit field into two bits field 16-bits wide each. So, ax part is the lower 16-bits of the eax register. Similarly, dx is the lower 16-bits of the edx register and likewise.

So, we just extend this idea to 64-bits as well. So, instead of eax we define rax which is 64-bits wide it the upper 32-bits are essentially part of the 64-bit mode with a lower 32-bits store eax and out of this again the lower 16-bits store ax. So, if you consider this, so this is a same picture for all the 64-bit registers rax, rbx, rcx, rdx, rsp, rbp, rsi and rdi. So, what is exactly happening till this point, what is happening is that a 64-bit processor, we will have these 8 registers which are 64-bits wide from rax till rdi. So, it will have some more. So, we will discuss them, but let us focus on the top 8 first. So, the top 8 registers will lit will be 64-bits wide and it will have only these top 8 for you know with these names now if we have an instruction that accesses eax eax in this case would basically mean the lower 32-bits of rax.

So, what the processor will do is that it will read the `rax` register, discard the upper 32-bits and take the lower 32-bits and return the value the same is true for a right. So, the question, so let us assume that you know 64-bit processor, we try to access `si`. So, we try to access `si`, what the processor would do a it will first and let us say that in a read aside. So, it will first write a read the `rsi` registers. So, out of the 64-bits, it will only consider the bottom 16-bits this is what `si` corresponds to. Similarly, if you want to write to `si` then what the processor will do is it will write to the lower 16-bits least significant 16-bits of the `rsi` register. So, essentially it all depends on how we view this set of registers, if you view the set of registers as 16-bit quantities then we access the lower 16-bits of every register.

So, let us consider one more example, let us assume that we have a 32-bit processor. In a 32-bit processor, clearly the `rax`, `rbx` will not be there. So, what is basically means is that if I write a program with let us say `rax` in a 32-bit processor, this will clearly be an error, but if I write `eax` it is say, it is fine. So, it is essentially the first register. And I can always write `ax`, so `ax` will be the lower 16-bits of the first register.

So, what we are doing you know in a nutshell to summarize is that every register is being viewed in multiple ways the storage area is the same, but it is just that our view is changing. So, we can when we say that let us say the name of a register is `sp`, what we mean is the lower 16-bits of pretty much the first, second, third, fourth, fifth register, so the lower 16-bits of the fifth register. Similarly, when we say `esp` is the lower 32-bits of the fifth register; and when we say `rsp`, it is essentially this fifth register whose length is 64-bits.

So, it is possible for any processor that implements the x86 instruction set to define these set of registers, and it all depends on how many bits is a processor you know how wide is the processor. If it is a 64-bit processor it needs to define `rsp`, `esp` and `sp`; if it is a 32-bit processor clearly every register is 32-bits. So, it is meaningless to define something like `rx`, `rax` or `rbx`, `rsp`, because 64-bits cannot be supported, but at least we can say that the full 32-bits will let us say be referred with the `e` suffix and for the 16-bit register we consider the lower 16-bits. So, the advantage here is that we are not wasting any space the code is pretty much compatible. So, the code of let us say 16-bit Intel processor and easily run on a 64-bit Intel processor, because the set of registers are all there. And we have not created any extra space further all we are saying is given you know.

Let us assume that an assembly code for a 16-bit processor at `cx` what we will do is that we generate code which will tell the processor to access `r cx` and then consider the lower 16-bits. So, in this case, it is just a question, this is actually a very neat way of designing the set of registers, it keeps the clients happy, it keeps the customers happy. Well, it does not keep the designers very happy, because it introduces makes their life slightly difficult, but actually not that difficult as we shall see in chapter 6, 7 and 8 and, but the most important thing is as the compatibility of code is there.

So, tomorrow if Intel releases a new processor, then we will be sure that all the code that we have right all the binaries that we have will actually run. In addition the 64-bit ISA defines 8 extra registers `r 8` to `r 15`. And mind your `r 8` to `r 15`, do not have any of these sub fields because the 16 and 32-bit ISA, the 16-bit and 32-bit Intel ISA is only defined 8 registers. So, for these 8 registers, they are appropriately mapped to different fields inside large 64-bit registers, but `r 8` to `r 14` are 8 extra registers, we should not have any sub fields and they can be used by 64-bit Intel processors and Intel binaries. But mind you any binary generated for a 64-bit processor will not run on a 32-bit processor, but neither should it be. See always want whatever programs you write to be compatible with future processors you do not want any program that you write today to be compatible with processors of the past. So, this is sort of a general rule that most companies follow and that is the reason this particular design decision has been made.

So, before I leave this slide, I would again like to underscore the importance of this particular slide is very, very important that readers and listeners understand this slide. And basically the imp important takeaway point is that the register is the same given a program if the register will just be interpreted in different ways. So, if we write `ax`, we will only interpret the bottom 16-bits; if we write `eax`, then only interpret the bottom 32-bits; and if we write `r ax`, we will interpret essentially all 64-bits in a 64-bit processor.

(Refer Slide Time: 24:02)

x86 can even Support 8 bit Registers

ax	ah	al
bx	bh	bl
cx	ch	cl
dx	dh	dl

- * For the first four 16 bit registers
 - * The lower 8 bits are represented by : al, bl, cl, dl
 - * The upper 8 bits are represented by : ah, bh, ch, dh

Mc Graw Hill Education

9

So, x86 can actually takes the arguments slightly further. So, it can even support 8-bit registers. So, the ax, bx, cx, and dx fields are further subdivided into two 8-bit fields. So, this is to ensure that we are compatible at some of the earliest 8-bit processors. So, in this case, also you know it helps us write slightly more efficient code if we need to access values at a byte level. So, for because you know compatibility with 8080 which was a 8-bit processor is not really that important, but this allows us to access information at a bite level which is much more important today at least.

So, here the lower 8-bits of the lower one byte is represented by al, bl, cl and dl, l for lower; similarly the upper 8-bits or byte is represented by ah, bh, ch and dh. So, so in this case of the idea is the same it is exactly similar as this particular figure where we take a register and divide it into several sub fields and interpret the fields based on our instruction. So, in this case, if we further divide the top four 16-bit registers into two sub fields each ah al, bh bl, ch cl and dh dl.

(Refer Slide Time: 25:39)

x86 Flags Registers and PC

Fields in the flags register

Field	Condition	Semantics
OF	Overflow	Set on an overflow
CF	Carry flag	Set on a carry or borrow
ZF	Zero flag	Set when the result is a 0, or the comparison leads to an equality
SF	Sign flag	Sign bit of the result

- * Similar to the SimpleRisc flags register (ARM CPSR) (eoi := ebr)
- * It has 16 bit, 32 bit, and 64 bit variants
- * The PC is known as IP (instruction pointer) (PC)

10

So, along with the general purpose registers which can be used for all the assembly programs, and they can be used to do regular arithmetic computation and everything. So, these are pretty much in the multipurpose general purpose registers. X86 defines two more registers the first, but these registers cannot be used by the assembly programmer and cannot be manipulated by the assembly programmer. So, it can be read, but it cannot be manipulated. So, these registers are manipulated by the system. So, first we have the r flags register.

So, similar to our terminology in the previous two slides the r flags register is 64-bits its lower 32-bits can be interpreted as the e flags register, and the lower 16-bits can be interpreted as the flags register. So, flags, e flags and r flags, same way as ax, eax, rax. So, the flags register is exactly similar to the Simple RISC flags register or the ARM CPSR register. So, what it does is that it contains the results of the latest comparison. So, comparison, so there are many bits any many sub fields in this flags register. So, one of them is overflow OF. So, this is set if the previous operation can be in compare and there are some other operations which can set the flags. So, readers who are interested can take a look at the x86 manual and they will find out all the flags that any instruction can set. So, the overflow flag is set on an overflow, and so this can be read from the flags register.

Similarly, we have a C F for a carry flag which is set on a carry or borrow. So, let us assume I am doing a doing an addition or a subtraction. So, the carry flag C F is set if the addition results in a carry or the subtraction results in a borrow. Then we have the 0 flag the ZF which is set when the result of a comparison or result of any other instruction which can set a flag has led to equality. So, let us say we are comparing a and b or let us say you know in terms of assembly language, we are comparing registers eax and ebx. So, let us say the values if eax is you know equal to I am writing equal to equal to in C style if eax and ebx are the same value then the 0 flag will be set.

And then we have SF for the sign flag which is the sign bit of the result. So, the sign bit of the result will essentially indicate if the result is negative or nonnegative, non negative means positive or 0. So, these 4 bits can be set in the flags register. So, there are other bits as well, readers can take a look at Intel's manuals or the other bits, but these are the four main bits. So, the idea over here is that this flags register or e flags or r flags depending upon the processor in assembly language, stores the flags in exactly the same way as in the simple RISC flags register used to or the ARM CPSR register used to. So, the idea here is that any instruction like a compare which can set the flags will essentially set the flags based on its result. And then the flags can be used by a later instruction such as a branch to base its the decision and what exactly this register contains. So, this also has 16, 32 and 64-bit variants.

Similarly, we the PC the program counter is known as IP or the instruction pointer in x86 you know this is exactly what the PC is. So, this cannot be set by a user, but user means the programmer or the program, but the program can read its value. So, there are instructions actually to read its value it cannot be read or accessed directly, but using instructions you know there are methods to read the value of the program counter. So, the program counter is called the IP register in 16-bits, EIP registered in 32-bits and the r I p register extended instruction pointer register for 64-bits, it is called RIP. So, the idea is the same that the same register is divided into multiple fields and depending upon the nature of the instruction the instruction set in the processor we choose which field should be returned as the answer.

(Refer Slide Time: 30:54)

Floating-point Registers

- * x86 has 8 (80 bit) floating-point registers
- * st0 – st7
- * They are also arranged as a stack
- * st0 is the top of the stack
- * We can perform both register operations, as well as stack operations

McGraw Hill Education

11

So, now let us take a look at the floating-point registers. So, ARM, so we did not discuss the floating-point registers in x86 and ARM; mainly because, ARM has array floating-point support somewhat late and so it was not a good idea to have it at the part of a basic course, but floating-point support has been there in x86 for a long time now. So, the floating-point unit was sort of going back to history in a 386 processor right, 80386 the 386 processor is like series in the mid 80s, it did not have floating-point support. So, it was only possible to have integers. So, an additional core processor was used called a 387 to give it floating-point support, but finally when the 486 processor came it had floating-point support.

So, in the those days because not a lot of transistors were available in hardware, the architectures were somewhat more complicated they were not very simple and pretty much x86 has not been able to change that for the floating-point for the set of floating-point registers. So, they are fairly complicated. So, but I say I hope that after explaining how it works, after this slide which is slide number 11, it will not look that complicated. So, the complexity arises from the following fact, it arises from this fact that the set of registers can be addressed independently, so they can be addressed. So, there are 8 floating-point registers, so they can be addressed as st0, st1 or till st7. So, they can be addressed in that way. And the other is that they can also be addressed as registers that are part of a stack a separate stack called a floating-point register stack.

So, the way that we can think of this is that inside the processor what we actually have is we have 8 locations arranged as a stack. A stack is any data structure which are the last in first out semantics, something exactly similar to a stack in programming languages. So, consider a stack of books. So, the book that we add at the end is on the top. Now, if we need to remove any book we start removing books from the top, so pretty much any book which was added the last will also be the first to be taken out see the same for register values as well. So, in this case, let us assume that we have a stack of 8 floating-point values. So, this is called the floating-point stack or the FP stack.

Now, another interesting part is that these floating-point registers each floating-point number actually has more precision than a double precision number. So, instead of 64-bits, each of these registers is actually 80 bits with a huge mantissa it gives Intel processors a lot of precision. So, we refer to the stack top as register `st 0`, and we refer to the bottom of the stack as register `st 7`. So, there are instructions to push a value on the floating-point stack. So, once this is done, so let us assume that the numbers were initially of this form 3, 4.5, 6 we just had three numbers.

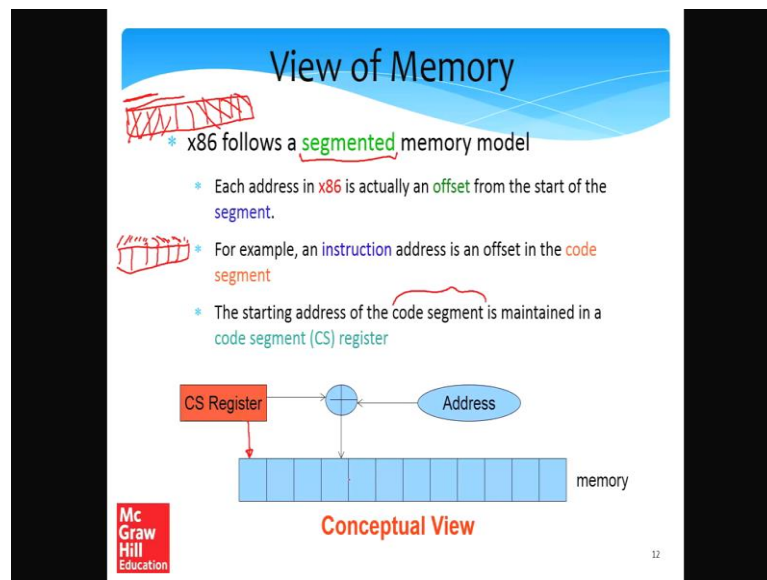
Now, let us say we push two on these floating-points stack. So, the new state of the stack would be 3 would come here, 4.5 would come here, 6 would come down and 2 will get pushed on the top. So, now we print the value of `st 0`, it will actually print 2, `st 1` will print 3, `st 2` will print 4.5, and `st 3` will print 6. So, this is where the complexity lies that we can look at the set of floating-point registers in two ways; we can either look at them in the set of registers or as a stack, it is better to look at them as a stack where essentially the register gives you the position on the stack.

So, let me repeat the floating-point register actually gives you the position on the floating-point stack. Say floating-point stack is essentially a stack of 8 values. I am just drawing it once again, because this is a very important concept and students typically do not get it. So, I am trying my best to explain it in as many ways possible. So, consider a floating-point stack as a sub stack of 8 values 8 floating-point values. So, let me draw 8 rectangles; let me draw one more at the top to make it 8 all. So, we have 8 rectangles. So, what `st 0` basically means is that it is a register and its value is whatever is at the top of the floating-point stack. Similarly, `st 7` can also be treated as a register, and its value is what is the last element in the floating-point stack.

So, if I push or pop the stack then essentially the elements move, so the value of these registers will also change right. So, essentially it is not a register in a registered sense, it is only something which tells you that for this let us say for s t 3, it tells you that for let us say the third position from the top of the stack or the fourth position in the stack what is the value. So, it is essentially you can think of it as a numbered stack, where it is possible to access the elements of the stack given its position from the top.

So, there are historical reasons why this has been done. So, this is sort of reminiscent if you still recall what was taught in chapter 1 that there are stack based machines. So, those days programming and writing code for stack based machines was somewhat easy, so that is the reason this is this was done in a 20 or 30 years ago, but because of reasons of maintaining compatibility it has become a very, very essential to sort of keep the models the programming models that used to be there. Still you know there is a need to keep them alive; otherwise a programs that were written for let say the 486 processors will not work on I seven processors. And unnecessarily companies that make x86 processors will lose business. So, if let say the set of if let say floating-point registers were written as a stack, it is a good idea to maintain it.

(Refer Slide Time: 38:28)



So, the view of memory is also not very easy. So, ARM and simple RISC viewed memory has an infinite array of bytes. So, basically the memories the first location was at location 0s next at location 1 and so on. So, the x86 memory model is slightly

different. So, it is a Von Neumann model no doubt. So, the instruction and data are saved in the same memory, but it is what is called a segmented memory model. So, the idea is that each address in x86 is actually an offset from the start of a segment.

So, what we do is that if we consider this is memory. So, any kind of a memory address, so we actually think is that we sort of divided this into multiple segments. So, this is one segment this is one more segment. So, it is an extension of a von Neumann model, it is more like a Harvard model in that sense a logical hardware model, but Howard model I am sorry, but the idea is that given a large array, we are interested only in parts of the array called segments. So, maybe you know this part and these parts. So, the way that this works is like this.

So, let us consider the code segment, which contains the instructions. So, the starting address of the code segment is contained in a code segment register. So, when we have the address of an instruction. So, maybe let say the starting of the code segment register points over here in memory. So, when we have an address, what we do is that every address is specified as an offset from its corresponding segment register. So, in this case, we take the value of the segment register, we take the value of the address, and we add them, and then the result once the memory address in which we will find the data.

(Refer Slide Time: 40:37)

Segmentation in x86

16 bit segment registers

cs	es
ss	fs
ds	gs

- * x86 has 6 different segment registers
 - * Each register is 16 bits wide
 - * Code segment (cs), data segment (ds), stack segment (ss), extra segment (es), extra segment 1 (fs), extra segment 2 (gs)

Mc Graw Hill Education

13

So, this can be extended as follows. There is a x86 support six different segments registers. So, each register is 16-bits wide. So, we are essentially starting from the

earliest days when segmentation was started or was introduced. So, we will have a code segment. So, what the code segment basically means is like this. So, let us consider a conceptual view of memory where we treated as the large in almost semi infinite area bytes. So, where I am drawing a conceptual view of memory where each box is a byte. So, the code segment register might let say start over here. Say every instruction in the program is specified as an offset from the code segment register. So, maybe let say there is an instruction those addresses 4. So, we will start from the code segment register and count 1, 2, 3, 4. So, this instruction would begin from here.

Similarly, we can have a data segment that contains all our data. So, maybe this can start from here. So, the data segment would be this wide, let us call it the data segment ds and similarly this might be the code segment. We can then have a stack segment to say to store the stack of a function. So, mind you this is different from the floating-point stack, the floating-point FP stack is stored in hardware, but the stack is a programming language concept which was elaborately introduced in the second half of chapter 3. So, you the reader should definitely go back to that those set of lectures or those videos and understand what the stack is all about.

So, the stack essentially stores the activation block and function, what is an activation block it stores all the arguments, the return address, all the temporary variables of the function this is called the stack. So, the stack segment also has its segment register; and along with that we have three extra segment registers cs, fs and gs which are not used, but they can be defined if need be.

(Refer Slide Time: 42:54)

Segmented vs Linear Memory Model

- In a **linear memory model** (e.g. SimpleRisc, ARM) the address specified in the instruction is sent to the memory system
 - There are no segment registers
- What are the advantages of a **segmented memory model**?
 - The **contents** of the segment registers can be changed by the **operating system** at runtime.
 - Can map the **text section (code)** to another part of memory, or in principle to other devices also (discussed in **Chapter 10**)
 - **Stores** cannot modify the instructions in the text section.
REASON : Stores use the **data segment**, and instructions use the **code segment**

Mc Graw Hill Education

14

So, ARM, simple RISC and so on, have a linear memory model where the address that is specified in the instruction is sent to the memory system. So, segmentation is not required. So, there are no segment registers; however, x86 does not have a linear memory model, it has a segmented memory model where let say we define divide our addresses into three types code which is regular program instructions data which is all the data that we would use and then a stack. So, data is pretty much global variables and you can think of stack of local variables. So, given a conceptual view of memory, we map these segments to different locations, and the segments, and every single address is specified as an offset from its segment register.

So, we will see this kind of a model has some advantages. We will not be able to discuss a lot at this point of time. We will actually discuss more of this in chapter 10, the chapter on memory system. But the idea basically is that this gives us some amount of security, because what a computer virus typically does is that the computer virus tries to change the code of a program such that instead of doing its normal job it does what the virus asks it to do.

With a segmented memory model this is somewhat difficult, but will appreciate all of this in chapter 10, but what can be understood at this point of time is that it will give us some advantages in terms of security. There were other advantages as well, for example, in early processor that were really memory constrained it was possible to use this

mechanism to actually work with a smaller amount of memory and keep on moving data between the hard disk and memory to support a larger programs. Those reasons are not very relevant today, but still there are issues of security that this model handles very well.

So, we look at them in chapter 10, where we will show that for most viruses, it is not possible to modify you know instructions in the text section of the program that is because any stores are by default supposed to use the data segment and the instructions are in the code segment. So, this gives us some amount of separation, so between the code and the data segment. So actually you know I was not very accurate when I said this follows the Von Neumann model the physically yes physically is the same memory, but logically it is more like a Harvard architecture where we are thinking of the code part of the memory and the data part of the memory is actually separate. So, the code part and the data part we are thinking of them as actually separate.

So, it is very difficult for instructions such as a store which modifies data to actually go and modify the code, and this actually makes it much, much harder to implement a malware such as a virus.

(Refer Slide Time: 46:14)

The slide is titled "How does Segmentation Work" and contains the following text and annotations:

- * The **segment registers** nowadays contain an offset into a segment descriptor table
- * Because, 16 bits are not sufficient to store a memory address
- * Modern x86 processors have two kinds of **segment descriptor tables**
- * **LDT** (Local Descriptor Table), 1 per process, typically not used nowadays X
- * **GDT** (Global Descriptor Table), contains 8191 entries
- * Each **entry** in these **tables** contains the starting address of the segment

Handwritten annotations include:

- A diagram showing a 16-bit register (CS) pointing to a memory address.
- A note: "code seg., data seg., stack seg. 8192 = 1024 * 8 = 2^13"
- A red arrow pointing to the GDT entry.

Mc Graw Hill Education logo is visible in the bottom left corner. The slide number 15 is in the bottom right corner.

So, how does segmentation work? So, in the 16-bit version, it had six segment registers, but nowadays with 64-bit you know with 64-bit systems, the segment registers will not be able to hold a 64-bit memory address. So, what they actually do nowadays is that they

contain an offset into a large segment descriptor table and this is again because 16-bits are not sufficient to store a memory address. So, again you know just to show an example. So, let us say the cs registers which contains the code segment instead of pointing to a memory address actually points to a location in a large system wide table you can think call it a segment descriptor table.

So, in this case, so since it is a 16-bit quantity, this table can have at the most 2^{16} entries. So, this entry will actually point to a memory address. And think of this that this entry it will actually be a memory address. So, this is how modern processor works that we still have 6 segment registers which can be modified by the operating system. What is an operating system it is a special program whose job is to take care of the rest of the programs. So, we will again discuss what an operating system is in slightly more detail in chapter 10 and chapter 10, 11 and 12, not 11, but chapter 10 and 12.

Mainly, but at the moment we can think of each segment register still containing 16-bits, but in this case 16-bits refer to an entry in a large table called the segment descriptor table. So, Intel processors have two kinds of segment descriptor tables, there is something called an LDT segment descriptor table, and a GDT - a global descriptor table. So, the LDT is actually meant to be one per process; what is the process a process is running instance of a program. So, this is typically not used nowadays if it is not used we will not discuss it the GDT is a global descriptor tables. So, the GDT is what we will discuss and so pretty much this does not have 2^{16} entries because we will typically not need that many segments, this is this is slightly smaller it contains 8191 entries. So, 8192 is actually a very special number, it is some time to guess, it is 1024×8 which is nothing but 2^{13} .

So, we will actually use 13-bits from the contents of the register to access this particular GDT table to read the starting memory address of the segment. And what are the three segments that we are interested in the code segment which contains all the instructions the data segment which contains all the global variables, and the constants. And the stack segment which contains all the local variables and function arguments and return addresses and so on. So, these are the primarily the three segments that we are interested in.

So, the starting address of these three segments will be different for different programs in the system, because they shared the memory the large memory is being shared and where these segments actually begin will definitely be different for all the programs in the system. But in any case we do not care about this at the moment all that we are concerned is that we will have a single large table we will be shared by all the running programs in the systems. And pretty much every program can access this to find out where its segments begin. And each entry in these tables in the GDT table in for say will contain the starting address of the segment.

(Refer Slide Time: 50:41)

The slide features a diagram at the top illustrating the memory access process. It shows a 'Store or load' operation leading to a 'mem addr.' (memory address). This address is then processed through an 'SDC' (Segment Descriptor Cache) and a 'GDT' (Global Descriptor Table). The diagram includes handwritten annotations: 'mem addr.' is circled in red, 'SDC' is boxed in red, and 'GDT' is boxed in red. A red arrow points from 'mem addr.' to a '+' sign, which then points to 'SDC'. Another red arrow points from 'SDC' to 'GDT'. A red arrow points from 'GDT' to 'mem addr.', with a handwritten note 'mem addr. + segment base' below it. A red arrow points from 'GDT' to 'fault', with a handwritten note 'seg. base' below it. A red arrow points from 'GDT' to 'low', with a handwritten note '79' below it.

- * Every memory access needs to access the GDT or LDT : **VERY SLOW**
- * Use a **segment descriptor cache (SDC)** at each processor that stores a copy of the relevant entries in the GDT
 - * Lookup the **SDC** first
 - * If an **entry** is not there, send a **request** to the **GDT**
 - * **Quick, fast, and efficient**

Mc Graw Hill Education logo is visible in the bottom left corner, and the number 16 is in the bottom right corner.

So, now the question is for every memory access what do we need to do. For every memory access let it be a store or a load. So, we will pretty much generate an address. So, we will have a base register depending upon the addressing mode, we can use base offset or base index or does not matter whatever is the addressing mode, we will generate a memory address. In a 64-bit processor, the memory address will be a 64-bit number; in a 32-bit processor, the memory address will be a 32-bit number. So, it does not matter. So, with this memory address, what we need to do is we need to do something else.

So, let us say that we need to access the segment register right we need to find the entry in the GDT which for corresponding to the value stored in the segment register. We need to read that entry add it to the memory address, compute the new memory address and send it to, so compute the new memory address and send it to the memory system right.

So, I am just writing MEMS is over here. So, this is pretty much the main idea that if for any store or a load instruction, which will have a memory address that the processor will compute we need to add it with the contents of the relevant segment, so we need to add it with a number which is obtained as follows. We read the relevant segment register, we access the GDT; from the GDT, we get the starting address of the segment in memory we add both to get the final memory address and this is sent to the memory system.

So, does it mean that for every memory access we need to access the GDT or LDT that would make things very slow? So, let us instead have a have the same mechanism we try to speed it up. So, let us use a segment descriptor cache - a SDC at each processor that stores a copy of the relevant entries in the GDT, which is in my opinion our opinion everybody's opinion a much faster route. So, what we do is that we define a shortcut what we do over here is that we define a segment descriptor cache we access that first with a very, very small and fast hardware structures. So, this will keep the memory addresses of the segments that we are interested in. So, we are typically interested in three segments code, data and stack; and given the fact that a processor at any point of time executes only one program, it cannot do more.

So, we just need to access a small hardware structure which is processor specific access the value and add it to the memory address. If you do not find it, the value is still then we will have to go through this longer route. So, I will sort of you know cut this and say that this is this going viral segment descriptor cache is a default method. Just in case, we do not find an entry we need to take the much slower method, which is this method. So, let me call this the slow method. So, I actually you know ultimately they produce the same information's. So, I can put it over here and cut it right here. So, this is the slower method this is the faster method.

So, what is the idea we look up the SDC first, if an entry is not there then only we get it from the GDT, and we populate SDC, otherwise we take the value from the SDC. So, this method is very quick fast and efficient and almost hides the fact that we are adding an additional level of work to every memory access. So, this fact although gets hidden with modern hardware. And we will discuss how in chapter nine, but the important point to note that is that with a segment descriptor cache things become really fast you do not have to access the GDT for every single access for every single memory access.

(Refer Slide Time: 55:19)

Memory Addressing Mode

32-bit

$$\text{address} = \begin{matrix} \text{optional} \\ \text{cs:} \\ \text{ds:} \\ \text{ss:} \\ \text{es:} \\ \text{fs:} \\ \text{gs:} \end{matrix} \begin{matrix} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esp} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{matrix} + \begin{matrix} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{matrix} * \begin{matrix} 1 \\ 2 \\ 4 \\ 8 \end{matrix} + \text{[displacement]}_{\text{offset}}$$

base register (32-bit) *index (32-bit)* *scale*

32-bit → 700
64-bit → 800
64-bit → 700
32-bit → 800

- * x86 supports a **base**, a **scaled index** and an **offset** (known as the **displacement**)
- * Each of the fields is optional ✓

17

So, x86 addressing modes given the fact that is a CISC instruction set are also somewhat complicated. So, it is very important for us to get a general idea of what the addressing mode looks like before actually we move forward. So, up till now listeners must have gotten a feel of the fact that x86 is definitely more complicated than most other instruction sets and well. So, that is where the see in the CISC comes from right complex, but it is not that complex there is some amount of there is a large amount of elegance to it. So, it is not that difficult.

So, let us take a look at this particular figure say any memory address this is the way that we actually specify it in assembly language and this specification is translated into machine code as well then we define a base register that is the first thing that we do. So, the base register we specify the segment, but mind you this is optional this field. So, actually all of these fields individually are optional. So, this field is definitely optional.

So, let me write optional over here. So, typically you will not find this being done unless is a special case. So, in any case all of these fields are actually optional. So, so we will discuss that part later. So, the then we have a base register which can be any of the sta any of the 8 general purpose registers and in 64-bit mode it can be the additional registers as well right there is no problem in that, but we will not discuss that for the time being. So, let us proceed with the rest of the lecture assuming that we are talking about the 32-bit instruction set right. So, there are different variants of x86.

So, let us assume that we are talking about the 32-bit instruction set. And with that assumption let us proceed, but clearly if you know how to write assembly code for a 32-bit ISA, you can always write assembly code for a 64-bit ISA all that you need to do is convert everyday register from `ax` to `rax` right. So, you will move from 32 to 64 that is pretty much all that you need to do and there are some other instructions which will also change, but we will you know discuss them when a time comes similarly to move from let us say 16 to 64 `ax` will become `rax`. So, this is a very simple transformation that we need to apply.

Similarly, going from 64 to let say 32, what we need to do is that all registers of the form let say `rax` or `rbx` does not matter will become `eax`. So, that is the reason it is not necessary to discuss every variant of the x86 instruction set separately for example, sixteen 32 and 64. So, we sort of take the middle path and we discuss the 32-bit instruction set, and then the readers can do the simple transformation which is just in the register name throughout the `e` and put in the `r`, and it will become 64-bit compliant as simple as that it is not harder than that at all.

So, coming back to our present discussion here we have a base register, for the base register can be any of the first 8 registers plus we can. So, this is scaled indexed addressing mode that we can have, here again the scale is optional. So, in this case the index can be any of the first 8 registers other than `esp`. So, the `esp` the stack pointer cannot be used as an index.


So, let me write that other than `esp` or the stack pointer right, all the other seven registers can be used on index and optionally they can be scaled by a factor of 1 or 2 or 4 or 8 and then we can add a displacement. So, displacement is an offset it is a constant. So, we can add any displacement that we want; and in 32-bit instruction set the displacement maximum size is limited to 32-bits. So, what does x86 do, it supports the base scaled index and an offset. And the offset is known as the displacement and mind you each of these fields is optional all right this is something which has to be kept in mind.

(Refer Slide Time: 60:28)

Examples of Addressing Modes

Memory operand	Value of the address (in register transfer notation)	Addressing mode
[eax]	eax	register-indirect
[eax + ecx*2]	eax + 2 * ecx	base-scaled-index
[eax + ecx*2 - 32]	eax + 2 * ecx - 32	base-scaled-index-offset
[edx - 12]	edx - 12	base-offset
[edx*2]	edx * 2	scaled-index
[0xFFE13342]	0xFFE13342	memory-direct

- * x86 supports **memory direct addressing**
- * The address can just be the **index**
- * It can be a combination of the **base**, **scaled index**, and **displacement**



18

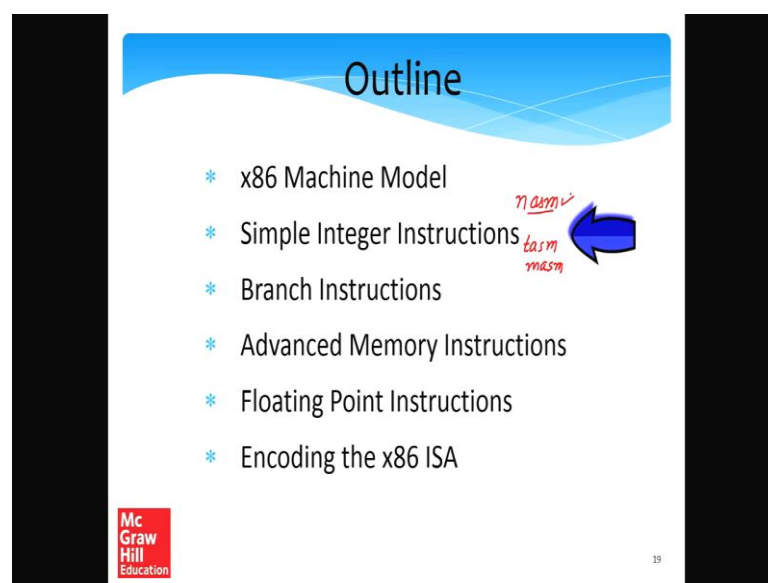
So, let me now give an example of addressing modes. So, let us consider the memory operand as it is specified in the x86 assembly language. The value of the address in register transfer notation. So, for this go back to chapter 3. So, it will tell you of what it is. So, this basically we are just mathematically trying to represent what the address is like and then the addressing mode. So, x86 supports many kinds of addressing modes, so one of the basic modes is a register indirect addressing mode where the value of the memory address is stored in eax. So, we represent it as eax with a square bracket on both sides.

Then we can have a base and scaled index addressing mode, where it can be the address can be represented as ax plus e cx multiplied by 2. So, the value of the address is eax plus 2 times e cx. So, this is base scaled index. Similarly, we have a base scaled index offset mode where the address can be other type eax plus e cx into 2 plus a displacement which in this case is minus 32. Similarly, we can have base offset just edx minus twelve we can just have scaled index which is just edx into 2. And we can just have a displacement which becomes memory direct. So, we can directly specify the memory address. So, this is also allowed in this form.

So, let us just go back to the previous slide and look at this once again. Now, that we have seen examples. See, you will be able to appreciate how and why this happens. So, in this particular case, as you see the base the index the scaled index rather and offset all

of these fields are optional. So, the memory address can be any combination of these fields as we see in this particular table over here that we can have all kinds of combinations of memory addresses. And this makes the entire thing very interesting and slightly complicated. But if you understand the main idea again I am going back to this figure where we have a base register and another index register which can be scaled by a factor from 1 to 8, 1 to 4 or 8 plus a fixed offset or a displacement which is a constant. So, basically if this is a general format, we can start fitting any kind of an address in this format where one or more of the fields are optional, and sort of not present. For example, in this case only the displacement is present, but the base and the index are not present.

(Refer Slide Time: 63:35)



So, after taking a look at the machine model which was binomial in simple we have done most of the hard and difficult and dirty work. So, now, we can appreciate the greatness of the x86 instruction set slightly better. So, again you know as I have been pointing out over and over its reading the book, listening to a video is great, but unless students actually sit down and practice and physically write assembly code, they will actually not learn or be effective programmers.

So, what I would suggest is that they can write assembly code. So, an assembler and a runtime are there not a runtime, but an assembler you can find on the books website. So, you can just go to the website of the book and type my name go to the home page of the author and go to the website of the book. So, you will find a link to an assembler called

naam - n a s m. And this can be used there are many other assemblers x86 assemblers as well such as tasm or nasn and say any assembler is fine right, there is no problem. But we will be using the national assembler format at least the format of the nasm assembler in this lecture, but other assemblers have similar format. So, the format per say is not that important.

(Refer Slide Time: 65:09)

The slide is titled "Basic x86 Assembly" and contains the following text:

- * We shall use the **NASM** assembler in this book
- * Available at : <http://www.nasm.us>
- * Generic **structure** of an assembly statement
 - * `<label>;<assembly instruction>;<comment>`
 - * **Comments** are preceded by a ;
- * x86 assembly instructions (*AT&T format and Intel format*)
 - * Typically in the 1 and 2 address format
 - * **2 address format** : `<instruction> <operand 1> <operand 2>`
 - * `<operand 1>` is typically both the **source** and **destination**

Handwritten notes on the slide include:

- 1) *RAM* (with a diagram of a box labeled "RAM")
- 2) *segmented memory model* (with arrows pointing to "seg. register", "GDT", "SS", and "SP stack pointer")
- AT&T format and Intel format* (written in red)

The slide also features the McGraw Hill Education logo in the bottom left corner and the number "20" in the bottom right corner.

So, the url of the nasm assembler is www.nasm.us. So, this is a fairly easy assembler to use, so you will find a lot of examples on the website of this assembler and at least on a linux on the Linux system and on a MAC OS system this assembler seems to be work seems to work this fine. So, the generic structure of an assembly statement is very similar to the generic structure of an assembly statement in both simple RISC as well as ARM. Any assembly statement can have an optional label followed with a colon then the assembly instruction. And comments here start in nasm comments start with a semicolon not with an as has been the case earlier.

So, x86 assembly instructions typically of two formats one is the AT&T format and the other is the Intel format and the other is the Intel format. So, in this book, we shall actually be using the Intel format. So, the formats are more or less the same, but in AT&T format that the destination comes at the end, which is not what we are used to, we are used to the destination operand coming at the beginning. So, we will use the Intel format. So, there is something important to keep in mind because we you know the

readers, listeners should know that both of these formats are out there, but you have to verify that the Intel format is being used.

So, typically most x86 instructions use a one and two address format. So, basically they can have one operand or two operands. So, the two address format which is of this type instruction operand 1 and operand 2, if we are using the Intel style operand 1 is typically both the source as well as the destination. So, essentially we will perform an operation on operand 1 and operand 2, subsequently we will save the result in operand one. So, operand one will get over it. So, this is standard format of x86 instructions for the uninitiated this can look as something new, but once you get used to it, I think its fine.

So, in general you know people get scared or the fact that you know the Intel x86 assembly is very complicated that is not the case. So, there is a lot of elegance in the instruction set and it is actually very easy to learn one sort of the basics are in place. So, what are the basics let me just repeat. So, I will just keep on repeating it several times during the slides I said readers do not forget the first basic is the way that we define registers. So, essentially ax, eax and rax are the same registers, but we are just using you know is the same register, but we are using different parts of it in a 64-bit system, that is point number one, Intel has a segmented memory model.

So, let me just write it. So, in a 64-bit system, rax, eax and ax are actually the same location in hardware is a different parts of it is being used. The next is a segmented memory model this is something that we all need to get used to. What are the main components of the segmented memory model well the main components are the segment registers, which give the start of the segment and the GDT table. So, well the segment registers previously used to give the start of the segment. Now, they point to an entry in the GDT table which gives us start of the segment to make this process faster. And we have a segment descriptor cache, which is a small piece of hardware, which stores some of the mappings that this particular processor would be interested in, so that a table need not be accessed all the time.

And we have the floating-point stack which is slightly difficult to understand, but hopefully we have understood it by now, where the idea basically is that all the floating-point values are saved in an 8 entry stack. And a register just gives the position of a

register let us say `stI` refers to the 8h entry of the stack. So, once these three ideas were cleared, we can proceed with the rig with the rest of the lecture very easily.

(Refer Slide Time: 70:24)

The slide is titled "Basic x86 Assembly - II" and contains the following content:

- * Rules for operands (for most instructions)
 - * Both the operands can be a register
 - * At most one of them can be an immediate
 - * At most one of them can be a memory location
 - * A memory operand is encapsulated in `[]`
- * Rules for immediates
 - * The size of an immediate is equal to the size of the memory address
 - * For example, for a 32 bit machine, the maximum size of an immediate is 32 bits

Mc Graw Hill Education logo is visible in the bottom left corner of the slide. The number 21 is in the bottom right corner.

So, let us now look at some of the basic rules for operands for most x86 instructions. So, both the operands can be registers. So, which is fine this is also this rule was also there in our earlier instruction sets ARM and simple RISC. So, at most so here are some additional points. So, at most one of them can be an immediate. So, both cannot be an immediate or a constant right that is meaningless. So, one of them has to be a variable which means stored in a register or memory, but at most one of them, one of them can be, but not two, at most one of them can be an immediate.

Utmost one of them sorry this should be m over here; that means, at most one of them can be a memory location. So, which means that you cannot have two memory locations as operands and a memory operand is encapsulated in square brackets which we have already seen. So, what are the rules for immediate, the size of an immediate is equal to the size of the memory address, for example, in a 32-bit machine, the maximum size of an immediate is 32-bits.

(Refer Slide Time: 71:48)

Basic x86 Assembly - III

16-bit, 32-bit, 64-bit

- * We shall use the 32 bit flavour of x86 in this book
- * Readers can seamlessly write 16 bit x86 programs
- * Simply use the registers : ax, bx, cx, dx, sp, bp, si, di
- * Readers can also write 64 bit programs by using the registers :
rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi, and ~~r9~~ r8 - r15

McGraw Hill Education

22

So, what are some of the other rules? So, we shall use the 32-bit flavor of x86 in this book in this lecture. So, the readers can seamlessly write as I have said 16-bit x86 programs no difficulty at all. Simply use the registers ax, bx, cx, dx, sp, bp, si and di. So, all the programs that we write can easily be converted into a 16-bit programs, sometimes some additional suffixes need to be changed in the instructions, but when that arises essentially I will tell you what needs to be done, to write the 16-bit or the 64-bit variants of those instructions. But in general converting a program between any of the three flavors and the three flavors being 16-bit, 32-bit and 64-bit is easy, only the register names have to be changed.

So, the readers can also write 64-bit programs by using the same set of registers, the r series r ax, r bx, r cx, r dx; and the 8 new registers that are provided by the which are essentially provided by the 64-bit system. And these registers are actually should be sorry r 8 to r 15 right not r 9 to r 15, but r 8 to r 15. So, next we will discuss what the individual instructions are like. So, this particular part of a lecture, we will end here; and the next part will begin with the move instruction and what it is like.