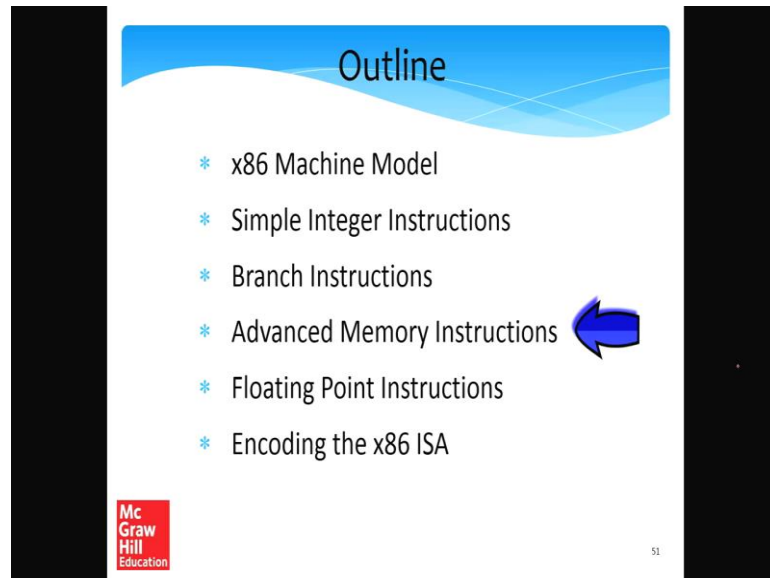


Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 12
x86 Assembly Language Part- III

(Refer Slide Time: 00:25)



The slide features a blue header with the word "Outline" in white. Below the header is a list of six topics, each preceded by a blue asterisk. A blue arrow points to the fourth item, "Advanced Memory Instructions". In the bottom left corner, there is a red logo for "Mc Graw Hill Education". In the bottom right corner, the number "51" is displayed.

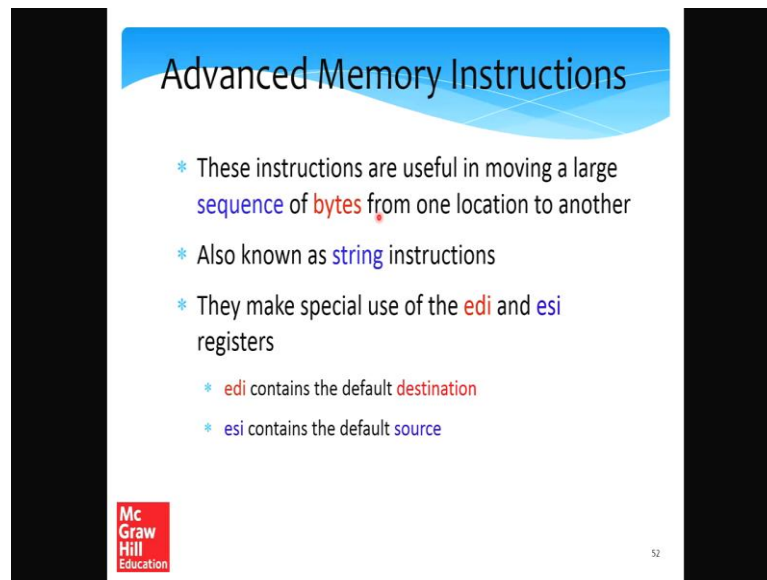
- * x86 Machine Model
- * Simple Integer Instructions
- * Branch Instructions
- * Advanced Memory Instructions
- * Floating Point Instructions
- * Encoding the x86 ISA

Mc Graw Hill Education

51

Now let us discuss advanced memory instructions. So, an advanced memory instruction is where we can actually see the power of the x86 instruction set. So, what good is a complex instruction set? Unless it gives us some instructions that make our life really easy; so you know there has to be some justification for the c in CISC.

(Refer Slide Time: 00:51)



The slide features a blue header with the title "Advanced Memory Instructions". Below the header, there are five bullet points. The first bullet point states that these instructions are useful for moving a large sequence of bytes. The second bullet point notes they are also known as string instructions. The third bullet point mentions their special use of the edi and esi registers. The fourth and fifth bullet points specify that edi is the default destination and esi is the default source. In the bottom left corner, there is a red logo for "Mc Graw Hill Education". In the bottom right corner, the number "52" is displayed.

Advanced Memory Instructions

- * These instructions are useful in moving a large sequence of bytes from one location to another
- * Also known as string instructions
- * They make special use of the edi and esi registers
 - * edi contains the default destination
 - * esi contains the default source

Mc Graw Hill Education

52

So, we will see that with advance memory instructions. So, let me illustrate a typical case, typically in some instructions will prove to be useful, if we look at certain common programming patterns. So, a typical programming pattern is that, we move a large amount of data, like a large sequence of bytes from one memory location to another memory location. So, instructions that help us in moving is long sequence of bytes, from 1 memory location a to memory location b, these are known as string instructions. The string instructions make special use of the edi and esi registers, as we had discussed in 1 of the earlier slides. edi contains the default destination, and esi contains the default source.

(Refer Slide Time: 01:50)

The slide is titled "The lea instruction" in a blue header. It contains the following text:

- * The *lea* (load effective address) inst. is used to load an address in to the *edi* and *esi* registers
- * In general, *lea* can be used to load an address in to any register
- * *lea ebx, [ecx + edx*2 + 16]* (does not access memory)
- * $ebx \leftarrow ecx + 2 * edx + 16$ (address)

Handwritten annotations include: a red arrow pointing from the *lea* instruction to the register *ebx*; a red bracket under the expression $[ecx + edx*2 + 16]$ with the note "(does not access memory)"; and a red arrow pointing from the expression $ecx + 2 * edx + 16$ to the word "address".

Mc Graw Hill Education logo is visible in the bottom left corner, and the number "53" is in the bottom right corner.

So, the first instruction that we need to introduce in this category, is the lea instruction, because this is what will be used in, you know in the subsequent discussion. So, this is very important, it is like the bedrock of all string instructions. So, lea means load effective address, and it is used to load an address into any registered, in particular the edi and the esi registers. So, in general lea can be used to load an address into any register, but we will mostly be interested in edi and esi in the later discussion, but let us look at a generic case first.

So, the standard format of lea is this. So, standard format of lea is register. So, it takes two operands, it is a two address format instruction, it takes a register and a memory. Also for example, in this case, we have a displacement, we have a scaled index, and we have a base register. So, instead of the how does this differ from the move instruction? So, what this does is, instead of actually loading something from memory, it considers this memory address, and simply computes the address part, does not access memory right.

So, this is very important, does not, this is very important, it does not access memory per say right by itself, a memory is not accessed. What it alternatively does, is that it considers the memory operand gets the address in memory, computes the address right, if you rather be called compute effective address. So, it computes the address, does not access memory, and transfers the value of the address which in this case is this right, the

address, transfers the value of the address to the destination register which is ebx right. It just helps us compute the address.

(Refer Slide Time: 04:19)

The slide is titled "stosd instruction" in a blue header. It contains the following text and annotations:

- * The `stosd` instruction does not have any operands
- * It saves the value in `(eax)` to `[edi]` (memory location in `edi`)
 - Handwritten: "56" with an arrow pointing to "(eax)"*
- * If the value of the `DF` flag in the flags register is 1
 - Handwritten: "bit in the flags register" with an arrow pointing to "DF flag"*
 - $edi \leftarrow edi - 4$
- * If the value in the `DF` flag in the flags register is 0
 - $edi \leftarrow edi + 4$
- * It is a `post-indexed` addressing mode

Mc Graw Hill Education logo is in the bottom left, and the number 54 is in the bottom right.

So, let us look at our first instruction, which makes our life slightly easy. It is the `stosd` instruction. It does not have any operands. So, what it does is this. it saves the value, that is there in `eax` to `edi` the memory location in `edi`. So, basically it treats `eax` a sort of as a default source, essentially it is a store instruction right. So, `eax` is the register that has a default source.

So, it takes the value in the `eax`, takes, assumes that `edi` contains a memory address, accesses memory at that address and does a store right. So, `edi` is assumed to contain by default a memory address, and the value in `eax` is stored over there. if the value of the `DF` flag well. So, in the flags register is essentially the `DF` bit. It is not the `DF` flag; it is the `DF` bit in the flags registers. So, maybe I should make it slightly more specific.

If the value of the `DF` bit in the flags register is said to one, what the `stosd` instruction subsequently does, is that it will subtract 4 from the `edi` register, because we are considering a 32 bit instruction set. So, it will subtract 4 from the `edi` register, if the value in the `DF` flag is 0 instead of subtracting 4 it will add 4 to `edi` register. So, think of it as a post indexed addressing mode, where we do store to memory, the address being in `edi` and the source of the data being in `eax`. after doing that ,we either increment `edi` by 4 or subtracted, sorry we either decremented by 4 or incremented by 4. Why 4, because 4

means 4 bytes, and we are considering a 32 bit instruction set, and whether we subtract 4 or add 4, depends on the value of the DF flag.

(Refer Slide Time: 06:48)

The slide is titled "lods instruction" in a blue header. It contains the following text:

- * The `lods` instruction does not have any operands
- * It saves the value in `[esi]` to `eax` (memory location in `esi`)
- * If the value of the `DF` flag in the flags register is 1
 - $esi \leftarrow esi - 4$
- * If the value in the `DF` flag in the flags register is 0
 - $esi \leftarrow esi + 4$
- * It is a post-indexed addressing mode

A diagram shows a red arrow pointing from `[esi]` to `eax`. Another red arrow points from `[esi]` to `esi` with a small `ed` next to it, indicating the update of the register.

Mc Graw Hill Education logo is in the bottom left corner. The number 55 is in the bottom right corner.

The `lods`, this instruction is a load string instruction, it also does not have any operands. What it does is essentially the reverse of the previous instruction `stosd`. It saves the value in `esi`. So, `esi` is considered as a source register. So, `esi` is considered to have an address, and the address within square brackets becomes a memory address. So, it saves the value in memory whose address is in `esi` to `eax`. So, this is this is like loading, you are loading from memory with the address being the contents of `esi`, the memory location is in `esi`. And here also we change the value of `esi`. So, the value of the `DF` flag in the flags register is 1, and then we subtract 4 from `esi`.

Otherwise if the value in the `DF` flag in the flags register is 0, we add 4 to `esi`. So, this is a post indexed addressing mode. Fair what we do is that we first access memory, in this case perform a load, and right perform a load from the memory operand `esi`. `esi` contains the address, and then we subsequently modify the contents of `esi`, we either subtract 4 or add 4 depending upon the value of the `DF` bit in the flags register.

(Refer Slide Time: 08:26)


Summary of Memory Instructions

Semantics	Example	Explanation
→ lea reg, mem	lea ebx, [esi + edi*2 + 10]	ebx ← esi + edi*2 + 10
b → byte w → word d → 32 bit q → 64 bit	stos(b/w/d/q)	stosd [edi] ← eax; edi ← edi + 4 * (-1) ^{DF}
	lods(b/w/d/q)	eax ← [esi]; esi ← esi + 4 * (-1) ^{DF}
	movs(b/w/d/q)	[edi] ← [esi]; esi ← esi + 4 * (-1) ^{DF} edi ← edi + 4 * (-1) ^{DF}
	std	DF ← 1
	cld	DF ← 0

DF → Direction Flag

- * **movsd** : [edi] ← [esi]
 - * Auto increments esi, and edi based on the DF flag
- * **std** : Sets the DF flag to 1
- * **cld** : Sets the DF flag to 0

56



So, let me summarize the memory instructions. So, the first instruction that we discussed in this interesting set, is lea, say lea is load effective address. So, it has a simple form, where the first and the only source operand is a memory address, is a memory operand, but we do not access memory, we compute the value of the address, and store it in the register. Subsequently we introduce stosd. So, stosd is a part of you know that many kinds of instructions can be there. So, it is a part of a set of four instructions, which are s t o s b s t o s w d and q. So, you already know what this stands for; b stands for byte. So, a byte is 8 bits, w stands for a word. So, a word is 16 bits. Means that transfer 16 bits, d is we have been using d, if it is a double word which is essentially 32 bits, because our example is in 32 bits, q is 64 bits. So, this will be useful in an instruction set with 64 bits.

So, depending upon the type of instruction that we are using, we add the right suffix. So, since we are looking at a 32 bit instruction set, we use stosd, if it is 64 bit use s t o s q. So, what does this do, essentially it reads the value in eax, save set in the location, the memory locations stored in edi. Subsequently it augments edi it sets edi equals edi plus 4 times minus 1 to the power DF, a DF is the flag spit. So, if DF is 0 this is edi plus 4 if DF is 1 it is edi minus 4. So lodsd is a same thing same format as s t o s we used lodsd, and because it is a 32 bit ISA. l o d s q could have been used if it is a 64 bit ISA.

So, here what was done what was done, is that this is the reverse of stosd here the source address in memory is stored in esi. So, we read the contents of that, store it in eax, then

we augment esi. So, we can either add 4 to esi or subtract 4, depending upon the value of the DF bit, if the DF bit is 0 we add 4. Let us now introduce the movsd instruction; this is also part of the same family of instructions. So, in this case we what we do is, that we read the value, we read a memory operand and we transfer it to another memory location. So, movsd also works, without any operands, without any source or destination operands. Sorry this should have a square bracket. So, what we do is, that we read the contents of esi and. So, basically esi contains the source address. So, you go to memory access the source address, read the contents, and we save it in the memory address pointed 2 by edi.

So, think of it as a loadsd and a subsequent store s t, fused into 1 right. So, this is exactly what it does. Subsequently we augment esi. So, we add 4 to it or subtract 4 depending upon the flag bit. Sorry one more mistake. So, this DF should be at the top. And similarly we do the same with edi; we either add 4 to it or subtract 4 to it, depending upon the value of the DF flag bit. We have two more instructions std and cld; std sets DF to 1, and cld which is like clear d, it sets the direction flag to 0, depending upon what we want to do, whether you want to add 4 or whether we want to subtract 4.

(Refer Slide Time: 13:13)

What is the value of eax after executing this code snippet?

```

mov dword [esp+4], 192
lea esi, [esp+4]
lea edi, [esp+8]
movsd
mov eax, [esp+8]

```

Handwritten annotations:
- Above the first line: "32 bits"
- Next to "192": "eax ← 192"
- Next to "[esp+4]": "esi → [esp+4] ... 104"
- Next to "[esp+8]": "edi → [esp+8] ... 108"
- Next to "[esp+8]": "100"
- Under the last line: "(192)" and "(100%)", with an arrow pointing from the value 192 in the first line to the register eax in the last line.

Answer: The movsd instruction transfer 4 bytes from the memory address specified in esi to the memory address specified in edi. Since we write 192 to the memory address specified in esi, we shall read back the same value in the last line.

Mc Graw Hill Education

57

So, let us consider an example. So let us see; what is the value of eax after executing this code snippet right. So, basically this code snippet is fairly complicated. So, we need to take a look at each of these instructions and see; what is the value of eax after at the end.

So, let us see; first we have a mov instruction. So, we mov the value 192 to esp plus 4 and we mov 32 bit is. So, we mov a dword, a double word 32 bit is to the location esp plus 4. So, this is the stack pointer, the stack pointer is pointing to a location, to the location on top of that we save 192. Then we load the effective address into esi which is esp plus 4.

So, let us assume that esp was 1000. So, at this point esi is equal to 1004 right. So, which is this location 1004? Similarly we load effective address of esp plus 8. So, this is edi will be equal to 1008, which is actually the address just above it 1008. Then we execute the movsd instruction. So, the movsd instruction will transfer 4 bytes from the memory address specified in esi, to the memory address specified in edi fine.

So, what is there in esi? In esi points to 1004 which has 192. So, these contents will get transferred over here. So, 192 will get return over here. Subsequently if we access esp plus 8 which is a location 1008, which is there in edi and we transfer the contents to eax, what do we get. We get the same, so these contents get transferred to eax. So, what do we get, we get 192. So, since we write 192 the memory address specified in esi we shall read back the same value in the last line.

(Refer Slide Time: 15:55)

```
.cld ; DF = 0
mov ebx, 0 ; initialisation of the loop index
.loop:
movsd ; [edi] <-- [esi]
inc ebx ; increment the index
cmp ebx, 10 ; loop condition
jne .loop
```

- * Copy a 10 element array
- * Starting address of source array in esi
- * Starting address of destination array in edi

McGraw Hill Education

58

So, this is a small and simple example, let us now do something. So, where is the power of the string instruction? So, what we have seen is, we have seen the lodsd, stosd and movsd kind of instructions, which are essentially shortcuts of load instructions, the

movsd instruction actually, fuses a load and store into one instruction, but that is hardly any power we are looking for something clearly much more.

So, let us consider an example, and let us see what can be done. So, let us assume that we have a ten element array of integers. The starting address of the source array is an esi, and the starting address of the destination that is an edi, and essentially we want to copy the source to the destination. So, the first thing that we do, which we should always do for such kind of codes, is that we should set the value of DF, the data flag, either 0 or 1 depending upon what we want to do. In our case we want to count upwards, start at the lowest element and keep going up, up means it was higher addresses. So, we set DF equal to 0 by calling the cld instruction.

Then we initialize the loop index of the array. So, we set ebx equal to 0 and start a loop. So, we call movsd. So, what it does is? That it transfers the contents of the memory, contents from esi to edi. So, this it does right. Subsequently we increment ebx, which is the index. We compare the ebx index with 10 as long as the index has not reached then we keep on iterating right.

So, basically if the index is not equal to 10, we do 1 more iteration and mind you since the movsd has a built in post increment operation, both edi and esi are incremented with 4 bytes. We do not need to explicitly increment or in some other cases decrement, edi and esi. So, they are being incremented automatically. So, nothing needs to be done. So, we just keep on calling movsd in a loop that is all that needs to be done to transfer one array from a source location to a destination location. This sounds simple enough in a sense it takes six instructions; otherwise it was taken slightly more, because we would have had a load and a store. It turns out that, this is not the end, we can do something better.

(Refer Slide Time: 18:41)

The rep prefix

Semantics	Example	Explanation
<code>rep inst</code>	<code>rep movsd</code>	<code>val</code> ← ecx; Execute the movsd instruction <code>val</code> times; <code>ecx</code> ← 0

* Repeats a given instruction *n* times

* *n* is the value stored in `ecx`

```
cld ; DF = 0
mov ecx, 10 ; Set the count to 10
rep movsd ; Execute movsd 10 times
```

`(edi) ← [esi]`
`esi ← esi + 4`
`edi ← edi + 4`

Mc
Graw
Hill
Education

59

So, let me introduce the rep prefix. So, rep basically means to repeat the repetition, repeat prefix. So, the semantics is that we put rep. So, rep is the starting of the instruction, and then we specify the instruction. any instruction we put rep before it example would be rep movsd. So, what we do. So, it will consider the value in the register ecx, and let us say it is value is Val. It will execute the movsd instruction val times right. So, basically this is a very powerful instruction, because 1 instruction incorporates a loop inside it. So, what will this do again? It will repeat the instruction inst, the same, you know val times, or val is the contents of ecx.

So, for example, if ecx contains 10, the rep prefix will ensure that the instruction after it gets repeated ten times, and every time the value of ecx will get decremented till, and at the end it will become equal to 0. So, what do we do? Here is a simple piece of code. So, we call the cld instruction first, we set DF to 0, which is something we need to do. In this case we are looking at 10 iterations. So, we set ecx to 10, and all that we do is rep move while, we are done, it is as simple as that.

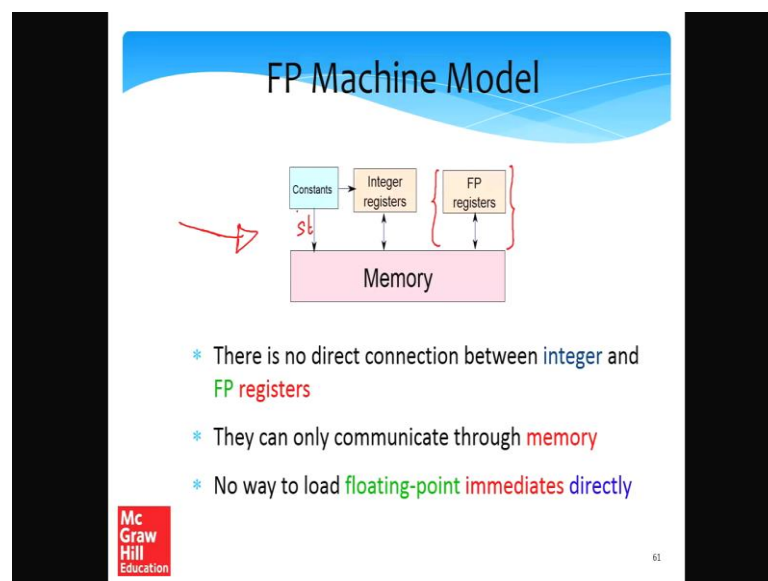
So, let us look at the previous. So, let us look at this piece of code, which is six instructions, and let us now look at this piece of code which is only three instructions. And the fantastic thing is that we have effectively incorporated an entire loop into 1 instruction, and we are all done. So, transferring a huge amount of data from point a to

point b is not a big deal at all. We will just have to execute three instructions in sequence. Let me go through these three magic instructions.

So, we have `cld` which essentially indicates the direction in which we will traverse upwards or downwards in memory. In this case we decide to go upwards we set the DF flag to 0. Subsequently set the number of iterations in `ecx` which is 10, and then we execute `movsd` 10 times. So, `movsd` will, what will it do. It will essentially read the contents of `esi`, access the memory location transfer them to the memory location pointed by `edi`. Since DF is 0 `esi` will be set to `esi plus 4` `edi` will be set to `edi plus 4`, `esi` get set to `esi plus 4` and `edi` will get set to `edi plus 4`, and we just need to execute it ten times and we are all done. So, now that we have seen the power of some advanced memory instructions, branch instructions.

Our understanding of integer instructions is per say more or less complete. So, we will not talk about it anymore. So, we will now move to floating point instructions.

(Refer Slide Time: 22:31)



So, all the students that have talked to regarding this issue of floating point instructions in x86, have consistently indicated that floating point instructions are fairly complicated, and this is a part that they would rather not study, and the you know they want to focus on something else, but that is not the case. So, x86 floating point instructions are not complicated at all, as long as you understand the basic philosophy behind them.

So, the machine model is like this memory. So, we can directly load constants into memory via the, let me write it. So, we can directly store constants into memory, we can directly store constants and registers. The registers can then be stored into memory, and memory values can also be stored into registers; however, floating point registers are sort of their connections are weak. Floating point registers can only talk to memory, and loading constants and so on into them is slightly difficult we shall see how, but then primary interface is via memory. So, there is no direct connection between integer and floating point registers, is point number 1, FP stands for floating point. They can only communicate via memory, and there is also no way to load floating point immediate directly, so that also needs to be done via memory.

So, once this sorry basic diagram is understood, we are in good shape.

(Refer Slide Time: 24:17)

FP Load Instructions

Semantics	Example	Explanation
<i>fld mem</i>	<i>fld dword [eax]</i>	Pushes an FP number stored in [eax] to the FP stack
<i>fld reg</i>	<i>fld st1</i>	Pushes the contents of st1 to the top of the stack
<i>fld mem</i>	<i>fld dword [eax]</i>	Pushes an integer stored in [eax] to the FP stack after converting it to a 32 bit floating point number

* The **fld** instruction pushes the value of the **first operand** (register/mem) to the **FP stack**
 * The **fld** instruction pushes an **integer** stored in **memory** to the **FP stack**

62

So let us understand; what are some of the basic floating point instructions? How are they to be invoked? And what is to be done to learn them correctly. So, this is basically one major problem that happens while writing floating point programs is that is students do not understand the machine model, they can make mistakes, and this is something we want to avoid. So, let us consider the fld instruction; fld's is very simple floating point load. So, this instruction pushes the value of the first operand, which can either be a register or memory.

So, that is the reason I actually have written it twice, or it in a register form and a memory form, on to the floating point stack. So, let us see. So, recall that in the floating point stack, you actually have 8 floating point registers right. So, this is like the FP stacks, I am not drawing 8 boxes, but that should be the connotation. So, the top of the stack is always st0. So, a floating point register is actually a pointer to the floating point stack, and the bottom of the stack is st7; that is the idea. So, essentially the values of registers will change. So, if I let us say push something onto the stack, let us say push 3.5. So, previously there were 4.5 over here, and then I push 3.5. So, 4.5 goes over here, it becomes st1 and 3.5 becomes in st0.

So, when I call something like fld mem. So, first thing I need to do is, I need to specify how many bytes am I interested in. So, here this is the memory operand, stored the address stored in eax. If I am interested in four bytes, I specify the dword modifier. So, it pushes a floating point number. Well and FP number fine grammatically it is still. So, it pushes an FP floating point number, stored in eax right in square brackets, to their floating point stack.

So, there if the number stored is 3.5 it gets stored on the floating point stack. Similarly there is a register form as well, but mind you this is not an integer register, this register form of the instruction, this is a floating point register. So, what we can do is, we can do fld stl the floating point registers are st0 to st7. So, it will push the contents of stl to the top of the stacks, it will read the contents of stl. So, let us say push 4.5, this four point five is stl. So, then this will again get pushed to the top of the stack right. So, these are the two most common variants of the fld instruction that we can either read directly from memory, or we can read another floating point register, but we read the contents and simply push them on the floating point stack. Similarly we have the fild instruction, which basically what this does is, that i stands for integer. So, this pushes an integer stored in memory to the floating point stacks, essentially it converts it does an int to float conversion, think of it that way.

So, in this case we read the value of eax, which is an address in memory and we read in 32 bit is, but the 32 bit is are not in the floating point format, they are in integer format. So, this integer which is stored in eax, is first converted to a 32 bit floating point number, and then pushed to the floating point stack. So, what are the basic instructions fldFloating point load and fild, where actually load an integer, but we convert it to the

floating point format and we push it onto the stack. So, this is how we load memory values right. So, there are two ways to load a floating point immediate the first is that we first load it is hex representation to memory and. So, what do we do let us take a number 3.5. So, we will first, well this technically should not be load, it is better, so actually call it store right, it is a better idea to actually call it store.

(Refer Slide Time: 28:48)

The slide is titled "Assembler Directives" and contains the following content:

- * There are two ways to load an FP **immediate**
 - * ^{5 byte (IEEE 754 format)} Load its **hex** representation to **memory**, and use the **fld** instruction to bring the value to a FP register.
 - * Use an **assembler directive** to store the **immediate** as a **constant** before the **program** starts. Then use the **fld** instruction to transfer the **value** to the **FP stack**.
- * In NASM :

```
section .data
num dd 2.392
```

Declares a 32 bit floating-point constant : 2.392 in the data section

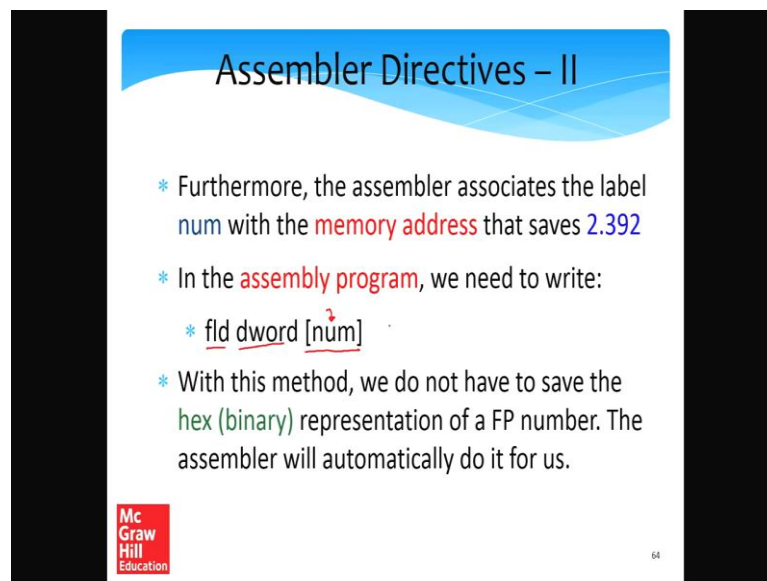
McGraw Hill Education logo is visible in the bottom left corner of the slide.

So, what we can do is, that we can store it is hex representation, write the representation and IEEE, I should also maybe call it the IEEE 754 format. We can store it to memory and use the fld instruction, to bring the value into a floating point register. Otherwise we can use an assembler directive, to store the immediate of the constant before the program starts. So, what would the assembler do, the assembler will essentially insert code, to basically convert it into hex, and store it in a certain area of memory, and then we can use the fld instruction, to transfer the value to the FP stacks, in a sense both these instructions, you know both of these points are the same, or the points basically say that either the program, assembly programmer manually converts a floating point number to it is hex representation, and stores it to memory and then reads it to the floating point stack.

The other is we let the assembler do exactly the same thing. So, in nasm, we can have it, there is a data section in which all of these constants can be declared, section dot data. So, what we do is that we can define a 32 bit floating point constant called num, which

is. So, so this specifies 32 bit is, and this is it is value 2.392. So, this is something that can be done, that we can specify a number in a decimal format and. So, the num the label here is actually the name of the variable, the memory variable which can be referenced later, and this is the value. So, what the assembler would do is it will take 2.392, convert it to its hex representation, store it somewhere in memory right, and associate that memory address right, mem address with the label num.

(Refer Slide Time: 31:36)



The slide is titled "Assembler Directives - II" and contains the following text:

- * Furthermore, the assembler associates the label num with the memory address that saves 2.392
- * In the assembly program, we need to write:
 - * `fld dword [num]`
- * With this method, we do not have to save the hex (binary) representation of a FP number. The assembler will automatically do it for us.

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


So, that is exactly what will happen the assembler will associate the label num with the memory address that saves 2.392. In the assembly program all that we need to write in the case of nasm, but all similar other assemblers have a similar format as well, is that we will write fld, dword for 32 bit is and num. So, the assumption here is that num will get replaced, by the actual value of the memory address by the assembler.

(Refer Slide Time: 32:19)

FP Exchange

Semantics	Example	Explanation
<u>fxch reg</u>	fxch st3	Exchange the contents of <u>st0</u> and <u>st3</u>
fxch	fxch	Exchange the contents of <u>st0</u> and <u>st1</u>

- * Exchanges the contents of two floating point registers
- * *st0* is always one of the FP registers



Mc Graw Hill Education

65

So, with this method we do not have to save the hex or binary representation of a floating point number. This will automatically be taken care of by the assembler for us. , so now, we will look at another instruction which, works exclusively on floating point registers, it is called floating point exchange or fxch right, f for floating point x c h for exchange. So, we can think we can call it fxch.

So, here the it has two variants, and in both the variants it exchanges the contents of two floating point registers, an *st0* which is the floating point stack top, is always one of the FP registers, if I write `fxch st3`, it will exchange the contents of *st0* and *st3*. If I just write `fxch` without any arguments, it will exchange the contents of *st0* and *st1*. So, pretty much if this is the floating point stack and the top is referred as *st0*, and the second is *st1*, then essentially their contents will be exchanged, interchanged.

(Refer Slide Time: 33:35)

FP Store Instruction

Semantics	Example	Explanation
<i>fst mem</i>	<code>fst dword [eax]</code>	$[eax] \leftarrow (st0)$
<i>fst reg</i>	<code>fst st4</code>	$st4 \leftarrow st0$
<i>fstp mem</i>	<code>fstp dword [eax]</code>	$[eax] \leftarrow st0$; pop the FP stack
<i>fist mem</i>	<code>fist dword [eax]</code>	$[eax] \leftarrow \text{int}(st0)$
<i>fistp mem</i>	<code>fistp dword [eax]</code>	$[eax] \leftarrow \text{int}(st0)$; pop the FP stack

- * The *fst* instruction saves the value of `st0` to memory
- * The *fist* instruction converts the FP value to an integer, and then saves it in memory.
- * With the 'p' suffix, the inst. also pops the FP stack

fld, fld, f2h, fist, fistp

Mc Graw Hill Education

66

So, now that we have taken a look at load instructions, let us take a look at floating point store instructions. So, similar to an `fld` we have an `fst`, it does exactly the same thing. So, let us take a look at the two variants are `fst` that we have. So, the `fst mem` instruction takes a memory address, a memory operand and it is argument and. So, it can save it in multiple formats, but let us only stick to the 32 bit format right now `dword`. So, since you are not specifying which floating point register `st0` is assumed to be king the stack top. So, it takes the contents of the stack top, and saves it at the memory address specified in `eax`.

Alternatively it is possible, to specify a certain floating point register. In this case what we do, what happens, is that the contents of `st0` are. So, these acts like a floating point register move pretty much. So, it takes the contents of `st0` and saves it in this case in `st4` right. So, basically `st0` is always the source, we can either save it in memory, we can either save it is contents in memory, or we can save it is contents in another floating point register.

So, let us now look at this variant over here `fst p`, which is exactly the same as `fst`, but in addition `p` stands for pop. So, let me maybe write here, the `p` over here stands for pop. So, it does exactly the same thing as `fst`, but the additional `p` after doing, what `fst` would have done it pops the floating point stack. So, `fist` is an analogue of `fild`. So, what this does is, that see here also we specify a memory address, and of course, the size how

many bit is need to be saved. So, what this does, is that let us just compare fst mem and fist mem. So, it is this line over here and this line over here. Say what this instruction does is that it first converts the floating point number into an integer representation, pretty much by truncating everything after the decimal point. So, that is what it does, that it first converts it into an integer.

After converting it to an integer, what does it do, what it does, is that it saves it in memory. And the fist p instruction is the same that we read in the value of the floating point register st0. So, st0 is almost always the default source or the default destination, and, so st0 sort of as a special place, it is a top of the floating point stack and if an argument is not specified, most likely it is st0. So, in this case the fist p mem, an argument is not specified which register. So, that is st0. So, we convert it to an integer and save it in memory, and because that p suffix is there we pop the floating point stack.

So, what are we looked at? So, till now we have looked at fld, and it is variants like fld. We have looked at floating point exchange which exchanges the contents of two registers. we have looked at fst which stores the value of a floating point register, either in memory or in another floating point register, and again in a different variance with a p suffix pops the stag and ever fist for integer conversion and so on. So, the main instructions are fld and fst that we need to bother about right; fld and fst.

(Refer Slide Time: 37:38)

Example

A 32 bit floating point number is loaded in st0. Convert it to an integer and save its value in eax.

fp reg → memory → int reg (st)

Answer:

```
1) 32-bit fld dword [esp+4] ; save st0 to [esp+4]
2) mov eax, [esp+4]
```

McGraw Hill Education

67

Let us now consider an example. So, the 32 bit floating point number is loaded in st0. So, it is present in st0. Let us convert it to an integer, and let us save its value in eax. So, to do this; one thing that is clear is that from the set of floating point registers. So, the direction of data flow is from the floating point registers to memory. So, this is a store. So, we need to use some variant of the fst instruction, and since conversion to an integer is also there, the instruction that we need to use, is fist, fist for store and i for the conversion to integer.

So, we want to save this in a memory location, first and then transfer it from memory to a register, to an integer register, because we cannot directly transfer a floating point value to an integer register. So, what we do, is that we assume that let us say this location on the stack esp plus 4 is available is free. So, there we transfer a dword or 32 bit is, because it is a 32 bit floating point number. So, the contents of st0 get returned to this memory location, this is what the first statement achieves, then the second statement is a simple register mov which we have seen, you know in great detail in the previous sections. So, here what we do is that we specify a memory address, and from this memory address we essentially read in 4 bytes into the register eax.

(Refer Slide Time: 39:49)

Variants of the FP *add* instruction

Semantics	Example	Explanation
<i>fadd mem</i>	<i>fadd dword [eax]</i>	$st0 \leftarrow st0 + [eax]$
<i>fadd reg, reg</i>	<i>fadd st0, st1</i>	$st0 \leftarrow st0 + st1$ (one of the registers has to be <i>st0</i>)
<i>fadd reg, reg</i>	<i>faddp st1, st0</i>	$st1 \leftarrow st0 + st1$; pop the FP stack
<i>fiadd mem</i>	<i>fiadd dword [eax]</i>	$st0 \leftarrow st0 + \text{float}([eax])$
<i>fadd reg</i>	<i>fadd st1</i>	$st0 \leftarrow st0 + st1$

general

specific

- * *fadd* adds two FP numbers
- * *faddp* additionally pops the stack
- * *fiadd* adds an integer in the first memory operand to *st0*

Mc Graw Hill Education

So, let us now look at the floating point add instruction, which is in a sense similar to some of the earlier instructions that we have seen, and so, but let us take a look it also has different forms, and some of these forms are slightly involved. So, let us start taking

a look at least the most basic form. So, the 1 operand forms of the fadd instruction takes a memory operand as the single source. So, for example, fadd, dword, eax; so in this case, the contents of the register eax are taken, memory is accessed and from there we read in a dword or a double word, which is 32 bit is. So, the assumption is that a floating point number is present over there, and we take it and we add it add it is contents to st0, and save the results in st0 as well.

So, st0 can be thought of as the default floating point register for this instruction. The other form is where the instruction takes two registers, to floating point registers to be specific, as the operands, and this the form is exactly similar to the add instruction to the integer add instruction, the only caveat here is that one of the registers has to be st0. So, so most versions of x86 of the x86 ISA, actually required this that 1 of the registers has to be st0. So, in this case the add instruction is similar to the integer add instruction, where we set st0 is equal to st0 plus stl. I mean this instead of stl this can be st 5 it is still fine. So, we can say st0 is st0 plus st 5. So, it does not matter. Then the add instruction also has a mode, where we can pop the stack. So, we use the p suffix, p is for pop.

So, here what we do. So, before I explain this we need to consider 1 thing that, we cannot store the result in st0, because st0 is going to get popped. So, the typical form in this case, is that we have some floating point register which is not st0, typically stl, but can be others also, and then we store the result in stl and. So, in this case also, it is typically the case at st0 is 1 of the operands. So, we add. So, in this case we can think of this as st i where you know it can be any st, and any st where i is not equal to 0 since this is stl the explanation is that the way this instruction will work, we will compute st0 plus stl and save the result in stl; that is important, because we cannot save it in st0 it is going to get popped, subsequently we pop the floating point stack.

So, this means that whatever are the contents of stl become the contents of st0, after the pop this instruction also has a mode, where we can read in an integer from memory, converted to floating point and do an addition. For example, if I have the fi if i have this i the character i after f. So, the instruction will become fi add, and an example of this instruction would be fi word, if I add dword eax and. So, here this contains the memory address. So, the computation that is being done is that to st0, we add the contents of the memory address. So, that is assumed to an integers they convert it to a floating point, we add it to the contents of st0, and save the results in st0.

So, since all our code has been specific to the nasm assembler. The nasm assembler also provides one more variant of the fadd instruction. So, this variant has also been discussed in a book and is there in several examples as well, but this is not a part of the general x86 specification, even though a lot of assemblers do implement, it because it is very convenient, but users should keep in mind that the table on top, these are like generic formats which you will find in Intel manuals, but this is nasm specific. So, can be used, if nasm is being you used as the assembler, and if there are other assemblers suppose this particular format, this can be used is nothing wrong in it. So, this format is a a 1 address format instruction, where the only operand that fadd takes, is a floating point register.

So, for example, fadd stl. So, here the default source, the default second source is st0 and the default destination is st0. So, we could have very well written fadd s t four. So, the effect that this would have had is st0 is st0 plus st4. So, these are very convenient when writing assembly programs, but as I said this is not a general format of the fadd instruction. So, it is possible that in some assemblers this might not work and. So, this possibility is always there that in some assemblers it might not accept this format, but wherever this format is being accepted it can be used. See here what needs to be remembered is that the second source operand is st0, and the destination is also st0.

(Refer Slide Time: 46:01)

**Subtraction, Multiplication,
Division**

have the same variants also

Semantics	Example	Explanation
fsub mem	fsub dword [eax]	st0 ← st0 - [eax]
fmul mem	fmul dword [eax]	st0 ← st0 * [eax]
fdiv mem	fdiv dword [eax]	st0 ← st0 / [eax]

* *fsub, fmul, fdiv* have exactly the same form (variants) as the *fadd* instruction

Mc Graw Hill Education

So, similar to add we have subtracts multiplication and division instructions, which work in exactly the same way, and have the same variance also. So, let me write it down.

So, exactly the variance that we have seen over here, say they can take a single memory operand as you know the arguments of the instruction or they can take to floating point registers they also have the additional p mode where we pop the floating point stack, and also similar to f i add you can have an f i sub, where we read in an integer from memory converted to floating point and use it in a subtraction. So, here I am just showing examples with a default memory operand.

So, in this case, if this is the memory operand what we will do is that we will read in the value and subtract it from st0 or multiply it to st0 or divide it to st0, whatever the case may be and the rest of the variants are the same at the cost of. So, basically to avoid repeating, I am not talking about all of the variants once again, but essentially to look at what variants are supported, let us go back to this table which shows the variance for the fadd instruction, and reuse the same variance for the other instructions f sub f mul and f div.

(Refer Slide Time: 47:51)

Example: Arithmetic Mean

Compute the arithmetic mean of two integers stored in `eax` and `ebx`. Save the result (in 64 bits) in `esp+4`. Assume that the memory address, `two`, contains the constant 2.

Answer:

```

; load the inputs to the FP stack
mov [esp], eax
mov [esp+4], ebx
fild dword [esp]
fild dword [esp+4]

fadd st0, stl                ; compute the sum
fdiv dword [two]            ; arithmetic mean

fstq qword [esp+4]          ; save the result to [esp+4]
                             ; used the qword identifier
                             ; for specifying 64 bits

```

Handwritten annotations: Red circles around `eax`, `ebx`, `stl`, and `st0`. Red arrows point from `stl` to `fadd` and from `st0` to `fdiv`. A red bracket under `fstq` is labeled "64".

Mc Graw Hill Education 70

So, let us show an example, let us compute the arithmetic mean of 2 integers stored in `eax` and `ebx`, let us save the result. So, here there is a catch. So, we do not want to save a 32 bit floating point, we want to save a 64 bit floating point. a 64 bit floating point number is a double precision numbers. So, we want to save this in the address `esp` plus four. Let us assume that the memory address `two` contains the constant 2. So, basically a loading value into floating point registers is complicated, loading constants particularly.

So, they need to be saved in some memory location, and that memory location needs to be, the contents of that memory location need to be transferred to the floating point stack. So, let us do one thing, let us first start by loading the inputs to the floating point stack. So, since `eax` and `ebx` contain the two integers. We need to transfer them to memory first, because direct communication between integer and floating point registers is not there. So, let us transfer them to memory.

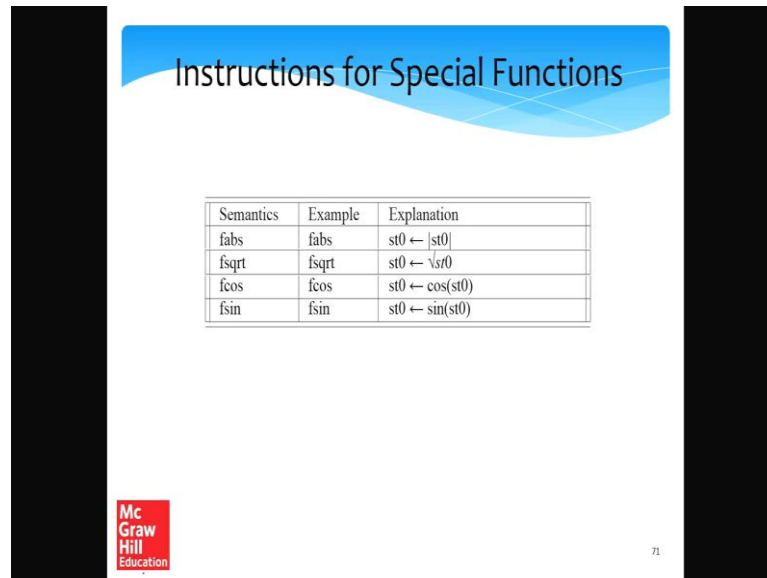
So, let us assume that these two memory locations at `esp` and `esp plus 4` are free. Subsequently let us load them onto the spoof floating point stack. We will use the `fild` instruction, basically meaning that a conversion will happen from integers to floating point. So, this is what the `i` means over here. So, then we will convert them, and we load them onto the stack. So, basically this value will be at the stack top `st0`, and this will be at the second from the stack top `st1`. Let us go ahead let us compute the sum of `st0` and `st1`, and, so the result will get saved in `st0` and let us divide the contents of `st0` by, whatever is stored in the memory location 2. So, what is stored there, what is stored there is 2 point 0. So, let us divide it, and the final result is there in `st0`.

So, basically let us consider an example, if `eax` can contain eight and `ebx` contain 9. So, arithmetic of 8 and 9 is 8.5. So, 8.5 is stored in `st0`. What needs to be done next? Well, what needs to be done next is that we need to save the result to `esp plus 4` to this memory address `esp plus 4`, and we want to save a 64 bit result. So, that is the catch, to save a 64 bit result I am using the `fstp` instruction, and to just clean up the stack I am using `fstp`, but it is technically not required. So, `fst` would have been just fine. And then the important point to note is that the `q` word identifier is being used. The `q` word identifier basically means quad word or 64 bit is. So, we are saving the result of the 64 bit quantity, and this is being stored at the memory location `esp plus 4`.

So, this is all that we need to write this program. So, the program contains seven assembly instructions which is not much. Given the fact that a lot is being achieved we are taking integers, converting them to floating point, putting them on the floating point, stack, computing their arithmetic mean and writing them back. So, it is a fair amount of work that is being done. and, but it is important to understand that what is being used the `fild` instruction being is being used to achieve the conversion, from integer to floating point; that is point number one, second we are doing a division, and a constant 2 is saved

at a certain memory address called `tw0`. And then we are saving the result back which is a 64 bit quantity, using the `q` word identifier to `esp` plus 4.

(Refer Slide Time: 52:13)



The slide features a blue header with the title "Instructions for Special Functions". Below the header is a table with three columns: "Semantics", "Example", and "Explanation". The table lists four instructions: `fabs`, `fsqrt`, `fcos`, and `fsin`. Each instruction is associated with a specific semantics and an example of its operation on the `st0` register. The `fsin` instruction is noted as being the same as `fsqrt`. The slide also includes the McGraw Hill Education logo in the bottom left corner and the number "71" in the bottom right corner.

Semantics	Example	Explanation
<code>fabs</code>	<code>fabs</code>	<code>st0 ← st0 </code>
<code>fsqrt</code>	<code>fsqrt</code>	<code>st0 ← √st0</code>
<code>fcos</code>	<code>fcos</code>	<code>st0 ← cos(st0)</code>
<code>fsin</code>	<code>fsin</code>	<code>st0 ← sin(st0)</code>

Now, let us take a look at instructions for some special functions. So, basically you know; what is the advantage of having a CISC instruction set, unless these special functions are supported. So, there are many more special functions which listeners can find, readers can find in Intel's manual, but some of the 1s that are most commonly used are `fabs` for an absolute value square root that computes a square root of `st0`, and saves it in `st0`. Then `cos` and `sin`; 1 second, then `cos` and `sin`.

So, again the cost of `st0` is computed and saved in `st0` and same for `sin`. So, there are many functions, other functions for other transcendentals and logarithms and so on. So, the listener is the listener the reader the user right. All these three are interchangeable terms, can take a look at the Intel's manual, to find out what are the other instructions that are supported.

(Refer Slide Time: 53:24)

Example: Geometric Mean


Compute the geometric mean of two integers stored in eax and ebx. Save the result (in 64 bits) in esp+4.

Answer:

```
; load the inputs to the FP stack
mov [esp], eax
mov [esp+4], ebx
fild dword [esp]
fild dword [esp+4]

fmul st0, st1 ; compute the product
fsqrt        ; geometric mean

fstp qword [esp+4] ; save the result to [esp+4]
                ; used the qword identifier
                ; for specifying 64 bits
```



72

So, let us do 1 thing, let us now compute the geometric mean of two integers store in eax and ebx, and let us save the result again in 64 bit is, at the location esp plus 4. So, the only change that is being done is that from arithmetic mean we have gone to geometric mean that is all. So, most of the code will remain in the same, and it genuinely does. So, this part remains exactly the same where we read in the floating point numbers from these two registers.

So, these two registers contain integers they are converted to floating point numbers, and stored on the floating point stack. Subsequently instead of an fadd we multiply them, stay st0 contains the product of st0 and st1. We compute the square root. So, basically st0 contains the square root of st0 right. and then the last line is exactly the same, where we again store it, we use the q word identifier to ensure that the result is 64 bit is, and we store it at the memory address esp plus 4.

(Refer Slide Time: 54:43)

Compare Instructions

Semantics	Example	Explanation
<code>fcomi reg, reg</code>	<code>fcomi st0, st1</code>	compare <code>st0</code> and <code>st1</code> , and set the <code>eflags</code> register (first register has to be <code>st0</code>)
<code>fcomip reg, reg</code>	<code>fcomi st0, st1</code>	compare <code>st0</code> and <code>st1</code> , and set the <code>eflags</code> register; pop the FP stack

- * The `fcomi` instruction compares the values of two FP registers and sets the flags
- * **NOTE:** It sets the flags for unsigned comparison
below (b) jb ~~ja~~
above (a) ja

Mc Graw Hill Education

73

So, now let us take a look at comparing floating point numbers that is important, because that is you know a lot of word that has branches and. So, on depends on the results of comparison. So, compare also has 2 variants, 2 simple variants. So, first variant is `fcomi`. So, here we compare 2 registers, let us say `st0` and `st1`. So, we compare `st0` and `st1`, and we set the flags register appropriately. So, here the caveat in most variants of the x86 instruction set is, but the first register has to be `st0`. So, that is typically the caveat that is placed; another variant of the `fcomi` the compare instruction. So this compare instruction is very similar to the `cmpl` instruction for integers. So, essentially we compare registers and we set the flags that are a simple basic idea.

So, we can additionally have the `p` suffix, where we compare two registers, and we set the flags register, and additionally at the end, we pop the floating point stack. So, let us say we do not need a value, we do not need `st0` for example, we can do the comparison and then at the end of the instruction, we can also pop the floating point stack. So, here is a very important point, a lot of students make mistakes in this point, and this is a very common source of bugs, while writing assembly programs with floating point operands. So, the source of the bugs is as follows.

So, let me draw a circle over here, the important point is that the flags are set for unsigned comparison not for signed comparison. So, recall that we had different kinds of flags. So, basically the condition codes that we need to use are essentially below, and

above. So, these are the condition codes for unsigned comparison, not greater than less than you know anything else right. So, definitely not in a greater than and less than and greater than equals and so on, not these ones. We need to use the condition codes basically b for below a for above n b for not below and so on. So, these are the condition codes that need to be used, and these are the condition codes for unsigned comparison and this is definitely what it is, that needs to be used.

So, this is a very, you know very common source of mistakes, because typically students would use, the condition codes for signed comparison, and they get the wrong answer. So, this needs to be kept in mind that the condition codes that need to be used, are the ones for unsigned comparisons, essentially we need branch instructions of the form j b or j a.

(Refer Slide Time: 57:58)

Example

Compare $\sin(2\theta)$ and $2\sin(\theta)\cos(\theta)$. Verify that they have the same value for any given value of θ . Assume that θ is stored in the data section at the label theta, and the threshold for floating point comparison is stored at label threshold. Save the result in eax (1 if equal, and 0 if unequal).

Answer: $|f_1 - f_2| < \epsilon \quad f_1 \quad f_2$
 $(3.0) \quad (3.0 + 10^{-20})$

```

; compute sin(2*theta), and save in [esp]
fld dword [theta]
fadd st0, st0           ; st0 = theta + theta
fsin                   ; st0 = sin(2θ)
fst dword [esp]      ; store the value

; compute (2*sin(theta)*cos(theta))
fld dword [theta]    ; st0
fst st1                ; st1 = st0 = theta
fsin                   ; st0 = sin(theta)
fxch                   ; swap st0 and st1 (st1=sin(theta)) st0 = θ
fcos                   ; st0 = cos(theta)
fmul st0, st1          ; st0 = sin(theta) * cos(theta)
fadd st0, st0          ; st0 = st0 + st0
                       ; = 2 * st0
                       ; = 2 * sin(θ) * cos(θ)
  
```

74

So, now let us take a look at difficult examples, it is not that difficult, but it pretty much uses all the concepts that we have learnt up till now. So, let us compare $\sin 2\theta$, and $2\sin\theta\cos\theta$. So, those who remember, still remember trigonometry will quickly guess that both of these expressions are the same. So, $\sin 2\theta$ and $2\sin\theta\cos\theta$ are the same expressions. Let us verify that they have the same value for any given value of θ , and assume that θ is stored in the data section data section of the assembly file, at the label theta right at the memory address theta, and the threshold. So, here is the important thing. Let me stop here right.

So, how do you compare two floating point variables, let us say f_1 and f_2 . So, the nature of real numbers or floating point numbers is in exact right. So, this we have discussed in chapter 2, that comparing floating point numbers makes little sense, basically because the representation is not exact. So, essentially you know we have a basic question here that, let us say a number of the form 3.0, and 3.0 plus, are they equal or not equal. So, this is a broad philosophical issue, so theoretically speaking they are not equal, but practically they might be equal and the reason being that we might not have enough bit is to specify 3 plus 10 to the power minus 30 right. So, we might not have that precision, that is required to specify f_2 , and as a result the hardware might compute f_1 to be equal to f_2 right, this is theoretically not correct, but real number arithmetic is always in exact.

So, the best way and the standard method, the standard approach of comparing floating point numbers is like this, that we take 2 numbers f_1 and f_2 , we subtract them, take their absolute value, and see if this is less than a predefined threshold like epsilon the. Threshold can be arbitrarily small; it depends upon what are tolerances. So, maybe like ten to the power minus 5 or 10 to the power minus 10, it all depends on what f_1 and f_2 are, but the main idea is, that the way we find out equality is basically by using the fact that the difference, the absolute value of the difference has to be less than a given threshold.

So, in this case let us assume that this threshold is saved at the label threshold, and finally, after the comparison let us save the result in eax 1 if there is equality, and 0 if there is lack of equality. So, we expect 1 as an answer all the time. So, let us now gradually. Let us break this example; because the example is kind of long and complicated as you see, it is a two page example. So, nevertheless try to understand it slowly gradually. So, let us do one thing, let us first compute $\sin 2\theta$, and let us save it in memory.

So, the first thing that we do is that, we load the constant theta from memory. So, what we do is, that we assume that the constant theta. In any value of the angle theta is stored in the memory label theta. So, we load the contents or onto the stack top, which is st0 and then we add st0 with st0, essentially compute 2θ , then we compute the sin of this. So, this basically at this point st0 is equal to \sin of 2θ . Subsequently we save $\sin 2\theta$; into the memory location pointed to by esp, where we assume that that particular

memory location is available and can be used. So, since all our quantities are 32 bit is, we use the dword identifier, and in addition we actually use the fstp instruction. So, basically we do not need the value anymore. So, we pop the stack also right in addition since. So, this is in general a good idea, that when we do not need the value. We can additionally pop the stack as I said; the floating point stack remains fairly clean.

So, now let us compute 2 sin theta times cos theta. So, let us do the same thing, let us load theta into st0, and let us create one more copy of theta in st1 by the fst instructions. So, fst st1, basically sets st1 equal to st0 which is equal to theta right. So, both at this point at this sorry, at this particular point st0 and st1 both of them contain theta. Now let us issue the fsign instruction. So, the fsign instruction has computes st0 is equal to sin theta, let us exchange. So, f x f exchange exchanges swaps st0 and st1.

So, at this particular point after this instruction st1 will contain the previous contents of st0, which is sin theta, and st0 will contain the old contents of st1 which is just theta. Then let us issue fcos. So, fcos will compute st0 as cos theta, then let us multiply st0 and st1. So, st0 will then contain sin theta times cos theta right, and then let us add st0 with st0. So, then st0 will be 2 times st0, which is 2 sin theta cos theta.

(Refer Slide Time: 64:53)

Example - II

```


; compute the modulus of the difference
push → fld dword [esp] ; load (sin(2*theta))
fsub st0, st1 ; st0 = sin(2*theta) - 2*sin(theta)cos(theta)
fabs →
; st0 = |sin(2θ) - 2sin(θ)cos(θ)|
; compare
fld dword [threshold] ; (10^-ε, 10^-d)
fcomi st0, st1 ; compare ε > diff
ja .equal ; threshold > difference (a for above)
mov eax, 0 ; else not equal
jmp .exit

.equal:
mov eax, 1 ; values are equal
.exit:

```

①

- 1) FP stack
- 2) FP insts.
- 3) conditionals (above & below)


75

So, where are we at the end of this page, we are saved sin 2 theta at the memory location esp right, point whose address is in esp, and we have we have 2 sin theta cos theta and st0 . Now that we have this let us do one thing, let us again load the value that was stored

in `esp`, which is $\sin 2\theta$. So, let us load this value into the stack top. So, the previous stack top goes into `st1`. So, at this particular point `st0` contains $\sin 2\theta$, and `st1` which used to be `st0` before we actually did a push operation on the stack. So, recall, that any `fld` instruction or any `fload` instruction pushes the value on the stack. So, the `st0` becomes `st1`, `st4` becomes `st5`, `st6` becomes `st7` and so on. So `st1` will be the previous contents of `st0` which is $2 \sin \theta \cos \theta$ right.

So, at this point let us subtract them. So, let us subtract this value from this value, and the difference between these two values will be saved in `st0`. And let us also then compute using the `fabs` instruction, it is absolute value. So, what is `st0` at this particular point may be. Let me draw a line here what does `st0` contain. `st0` contains the absolute value of the difference between $\sin 2\theta$, and $2 \sin \theta \cos \theta$. So, now, let us do 1 thing, let us read in the value of the threshold, which is the value `epsilon`. The maximum possible deviations between two quantities are again to be equal.

So, let us read value. So, so when we load it is value it will get loaded to `st0` right. So, `st0` will contain `epsilon`, and the new `st1` will actually contain the previous contents of `st0`, because we just pushed the stack which is this difference over here. So, let us compare `st0` which is `epsilon`, and `st1` which is the difference right, let us just compare. So, ideally what is it that we want? We want `epsilon` to be greater than the difference right; the threshold to be greater than the difference. So, which means that the threshold is above the difference, because it is an unsigned comparison, that is an important point to note, that will only use above and below. So, we use the branch operation `jg`; the conditional branch `jg`. So, `j` is for jump and `g` is for greater. So, if `jg` is true, we jump to `dot equal`, and at this particular point; since we have found out equality that the difference is less than the threshold, we set 1 to `eax` and we exit.

Otherwise just if this is not the case, we will set to say 0 to `eax`, and then we will exit, but the important point is this should not happen, because the trigonometric equality; that is a trigonometric identity, they should hold all the time. Unless you know there is some bug in the code. So, thankfully there is no bug and you can copy paste the code as is written, on to an awesome window and test. So, it should work just fine. And, so this is where and all the time the output should be equal to 1, which means that equality between $\sin 2\theta$ and $2 \sin \theta \cos \theta$ exists.

So, one thing that we get to understand over here: you are writing a program with regular x86 floating point instructions, is certainly not difficult at all. We just need to understand the nature of the floating point stack. So, let me just write down what are the concepts that we need to understand. We need to understand the nature of the floating point stack; that is the first thing that we need to do. So, we need to understand that when a new value is pushed into the stack, the rest of the values go down by one position that is point number one.

We need to understand the nature of the floating point instructions right. So, basically the fact that they treat st0 in a special way and how they work, then we need to understand the nature of floating point conditionals, that always the above and the below flags are to be used right, below condition codes are to be used, and the rest is straightforward, the rest is pretty easy. So, that is not difficult at all, and we can write fairly efficient floating point code with this. So, let me now add an afterword, now that we have seen this actually.

(Refer Slide Time: 70:18)

The slide is titled "Stack Cleanup Instructions". It contains a table with the following content:

Semantics	Example	Explanation
<u>f free reg</u>	f free st4	Free st4
<u>f init</u>	f init	Reset the status of the FP unit including the FP stack and registers

Handwritten notes in red ink include:

- A box labeled "empty" with an arrow pointing to the "Free st4" entry in the table.
- A list of instructions: "MMX", "SSE", and "AVX".
- A diagram of a square with an 'X' inside, representing a stack.
- A list of bit widths and their corresponding years: "16-bit - 8086", "32-bit - 286 (i386)", and "64-bit - x86-64".
- Additional notes: "1987" and "co-processor" next to the 32-bit entry.

The McGraw Hill Education logo is visible in the bottom left corner, and the number "76" is in the bottom right corner.

Let me add the afterward after the next slide, because that is when I will introduce two more instructions. So, these instructions are typically uses cleanup operations. So, for the stack cleanup; so the f free instruction which takes a floating point register as an argument, like f free st4. Essentially frees is the content of st4 and marks it as clean.

So, typically it is good programming practices that at the end of a program or at the end of certain points in the program, clean up the floating point stack. The floating point stack is a part of the program state. So, let us say that another program wants to execute, and the current program needs to be suspended. It is necessary to save the complete state of the floating point stack and then restore it. So, that is the reason that it is advisable that if floating point code is not used and we can keep the stack out of floating point clean. So, the instruction that helps us clean the floating point stack is `fldt`.

So, what does `fldt` do, it resets the status of the floating point unit including the stack and registers, which basically means that it cleans up all the registers, if there is it essentially says that the floating point stack is empty right, and it resets the flags and condition codes of all the floating point. I am sorry I should not use the word flags and condition codes, but it resets the status of all the floating point hardware, which includes adders and multipliers and all of that, and also similar kinds of hardware. It marks the floating point registers and the stack is empty, it pretty much clean setup.

Now, let me add my afterward. So, my after word is that this is the basic floating point support, that x86 processors have. So, x86 processors are primarily you know sixteen bit types, which are the 8086 type. So, these are not used very frequently these days, maybe in some embedded applications, but otherwise not very frequently, 32 bit is still around also in the embedded domain, it is gradually going away, but 32 bit is definitely around and. So, this is what is referred as x86 or x86 32 bit or sometimes i386. Then we have 64bit, 64 bit is definitely around and will be around for a long time. So, this is called x86 64.

So, these are all the variants and versions of the x86 instruction set. The 8 bit instruction set is also there, but that is too old. So, the floating point stack has been there since pretty much 1987, to the best of my knowledge at least with a 387 coprocessor, whose job was to do floating point operations. So, it is been there for a long time. So, this is a basic and you know the architecture is somewhat slightly old and primitive. So, maybe as an addendum to this book at a later point of time I will add the more recent architecture.

So, subsequently Intel has introduced many more floating point registered sets actually; 1 of them is MMX, the other are the SSE the registers, then recently the AVX registers. So, it is possible to use particularly say MMX defines the x m m registers right, that are x

m m registers y m m registers and so on, but with the x m m registers a lot of this code can actually get far more simplified. So, we are not covering them in at least this version of the book, but my advice to readers would be that if they want to know more, they can definitely read more about these extensions of the x86 architecture set ISA.