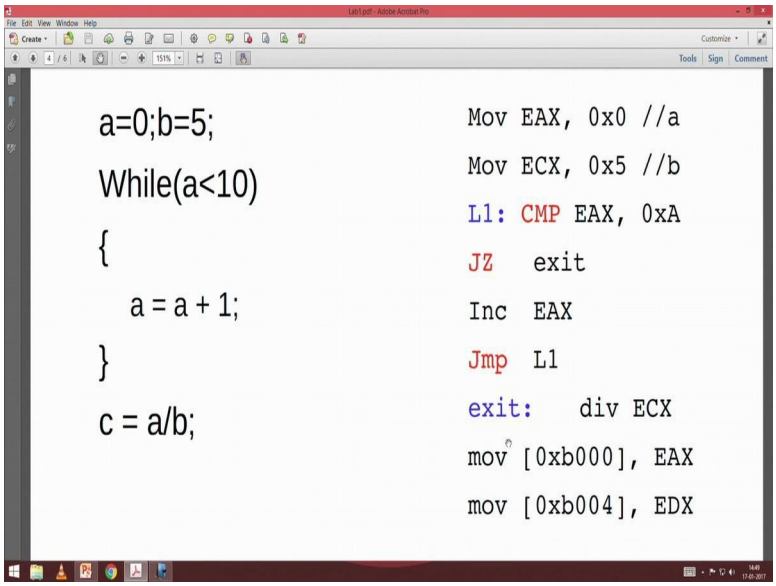
Computer Organization and Architecture
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 07
(Part – II)
Programming using X86 ISA - Addressing Modes

Now when we come to exit, now we do a division.

(Refer Slide Time: 00:27)



Now I say div ECX right; div ECX because b is stored in a b is mapped on to ECX div ECX now what are we done. So, we have only given whatever the denominator here where is the numerator? Where is EAX is you have to do EAX by ECX right, EAX is storing a, ECX is storing b I should do EAX by ECX. But here, I do not do div EAX by ECX, because div is a complex instruction always the numerator will be in the EAX resistor right div is a complex instruction. So, I do not explicitly give two different operators for division whenever I want to do division right the numerator will should be in the EAX resistor, I should I will only specify the denominator able to see that fine.

Why you should ask Intel, but this is how it is the set the many reasons for why it is should it be there. So, this address addressing mode one minute, this addressing mode where I am not telling explicitly where the numerator should; if I want to do division there are two operands that are necessary one is the numerator another is the

denominator, but the numerator I do not explicitly state that this is the numerator. So, this addressing mode is called implicit addressing mode right implicit addressing mode. So, we are already seen three addressing modes what are that; immediate addressing mode where the value of the operand is available with the instruction like move EAX 0 x 0 the 0 x 0 is a immediate operand, then we have seen a resistor addressing mode where we use a resistor as a destination or source then we are now we are seeing what is called an implicit addressing mode, where you know the thing is hidden you do not write it its implicit EAX being a numerator is implicit.

Now, I do div EAX by ECX. Now the result of this where will the result of this gets stored? See if you say move EAX comma 0, x 0 the result is basically EAX getting initialized to 0. Move ECX comma 0 x y the result is ECX gets initialized to 5; increment EAX the result is stored back in EAX itself where the source and destination operand are same, but if I say div ECX where will it be store? The answer will be stored in two resistors namely EAX and EDX. EAX will store; you know the quotient sorry the quotient and EDX will store the remainder. So, when we do integer division the quotient comes out and I can also have a remainder. So, EAX will store the quotient and EDX can store the remainder.

Student: (Refer Time: 03:50).

Right, so now, what I am saying that once I do div ECX, your numerator is lost the original numerator is lost the original numerator should be 10 right that is lost correct; now instead I will store the quotient and here. So, you get EDX in this case when I do 10 by 5 your EAX will be 2 and EDX will become 0. Now you move the content of EAX into some memory location which I give by 0 x b 1000, you move the content of EDX into another memory location say give it as 0 x b 1004.

So, 0 x b 1000 is an address in the memory 0 x b 0 0 4 is an address in the memory. So, I go and store the value of EAX and b 1000 and the value of the remainder in b 1004 correct. So, what is this addressing mode, this is called memory addressing right. So, now, we have seen four addressing modes immediate, resistor, implicit and now memory addressing correct are you able to follow yes or no; why are you respiring yes man say yes: so this addressing again memory addressing.

Student: (Refer Time: 05:20).

This a memory direct addressing because I give the address directly there to which memory I will location should I write to as a 0 x b 1000, I give the address directly and I put those two square brackets which is very important which says that I am writing into that memory right. So, this is called memory direct addressing wait you ask some doubt.

Student: Sir, are we able to see c is equal to b by a (Refer Time: 05:50).

C equal to b by a.

Student: Yeah.

It is very simple right move b should be in EAX and say yes.

Student: Sir, instruction is executed the value of EAX can be changed.

Yeah.

Student: So, if we have to use that value (Refer Time: 06:08).

Again that is a problem you have to be careful. So, that is why I said no general purpose resistors are scratch patch. So, once you scratch on it you through it off suddenly what I want to get it.

Student: (Refer Time: 06:22).

So, you should be careful while compilation what if.

Student: (Refer Time: 06:28).

After c equal to a by b, c equal to b by a, so that is why we have no EAX, ECX EDX EBX and all we have right. So, in another resistor you use.

Student: (Refer Time: 06:45).

EBX, EDX, and all other.

Student: (Refer Time: 06:50).

Then you put it in memory and get it back again, that is called resistance fielding I will talk about it good. Now let me ask you a very interesting question why b 1000 then b 1004.

Student: (Refer Time: 07:05).

Why b 1000 and then b 1004 their addresses.

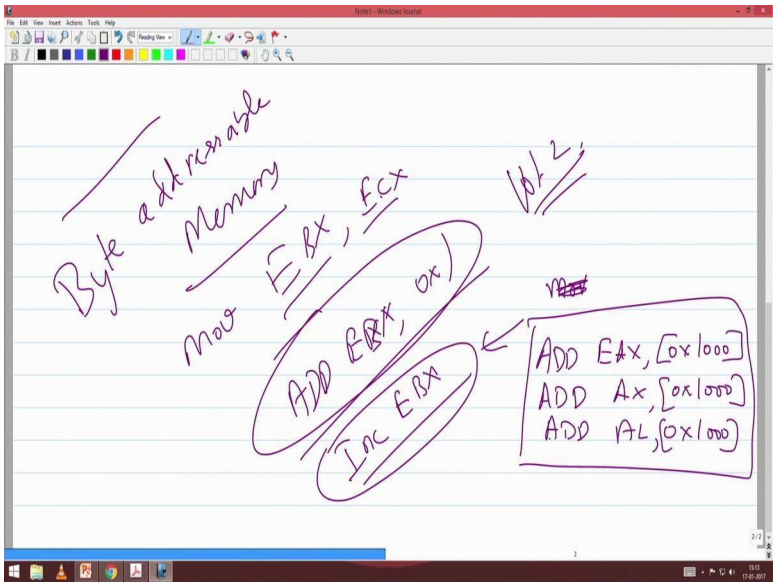
Student: (Refer Time: 07:11).

Yes.

Student: (Refer Time: 07:14).

Size of: so we are talking of integer. So, in every memory a location I can store only one byte. So, this is called. So, x 86 has what we call as a byte addressable memory.

(Refer Slide Time: 07:28)



Byte addressable memory; that means, for every byte I have an address. So, if I have a byte addressable memory then integer is 4 bytes in normal compilers today. So, at least the compiler I am using here is at 4 byte integer. So, b 1000, b 1001, b 10002, b 1003 will store the quotient b 1004, b 1005, b 1006, b 1007 will store the denominator sorry remainder.

Student: Sir, the second instruction in actually b executed when we on this 0 (Refer Time: 08:23).

Which second instruction?

Student: When we do c equals a by b it is unnecessary to store the remainder right.

But in this case it will store right you can if you want you can.

Student: But.

In these days you do not need it, but if you can you want you can. So, let me say some.

Student: (Refer Time: 08:48).

Some m; some m.

Student: (Refer Time: 08:50).

It depends no, what I am saying is that EDX will have the remainder.

Student: (Refer Time: 08:58) actual protocol is followed right like emergency.

This last instruction may not be dumped by the company. So, this is just an attention seeking for you to tell you that remainder also is generated and importantly your EDX gets screwed, it gets changed right if you used EDX for some value and you go and execute div ECX these two resistors get completely changed that is the most important point that we are right. So, it is not that the source operand is implicit even the destination operands can be also be implicit right; the source was EAX that was implicit the destination is EDX that is also implicit here, but interestingly the denominator is not changing right; why? No I could have made the denominator store the remainder no.

Student: (Refer Time: 09:58).

Why poor some other totally uninvolved resistors why I am going.

Student: (Refer Time: 10:03) because 0 integer (Refer Time: 10:05). We use it again for division later it becomes 0.

No what this source across and.

Student: (Refer Time: 10:16) we are getting the same resisters all the time right (Refer Time: 10:18) can be everything.

Right, if I say div EDX, EDX will change anyway right that is the question. So, it need not be ECX always, but the two operands where I store that, that has to be told in advance a when I execute div EAX and EDX will change right and if you use EDX as your denominator anyway it goes for your toss right do you understand this. So, this is this is. So, slowly this is a complex instruction set computer that complexity slowly will try and understand.

Student: Sir, when an addressing program is (Refer Time: 11:08).

This is an actual C program.

Student: Sir, (Refer Time: 11:12) are the value stored in the memory or (Refer Time: 11:15).

It all depends if you say a equal to 0, b equal to 5 and all right then it will compile like this.

Student: (Refer Time: 11:24).

Then you mapped on to a, if you want to here also I am reusing a right I am reusing a several times in that.

Student: No after that (Refer Time: 11:35).

After that you want to then you have to if you can, if you want to use reuse a after division then are this point you have to move a.

Student: To some memory.

To some memory location and some other register.

Student: (Refer Time: 11:51).

Right.

Student: (Refer Time: 11:53).

Yeah.

Student: Sir, can we 0 as b 1000 for instead of EAX (Refer Time: 11:58) some constant for division what is the parameter.

Division you can I you can put div 0 within bracket 0 x b 1000.

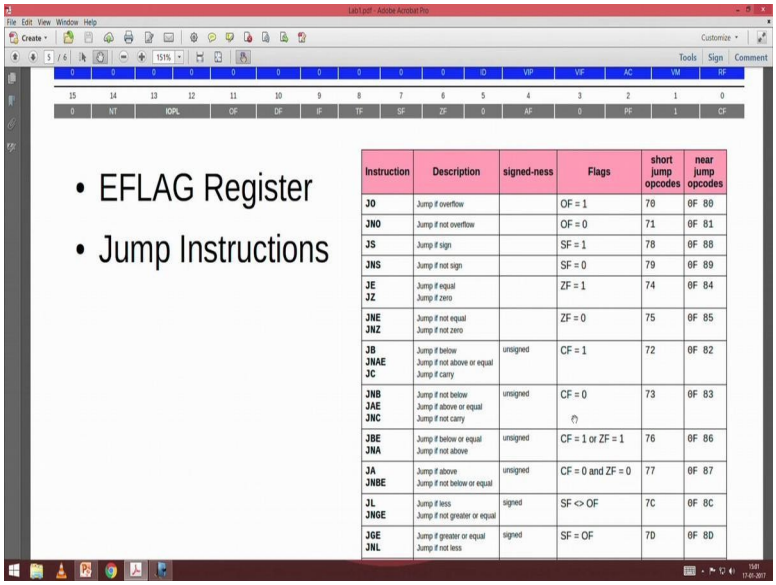
Student: (Refer Time: 12:09) sir, resistor (Refer Time: 12:10).

We can give a denominator as a memory operand we can I do not know. So, that is what you can go to the Intel manual.

Student: (Refer Time: 12:19) working about constant (Refer Time: 12:21).

I can give a constant as denominator. So, all these things you can see now this question has come up like.

(Refer Slide Time: 12:42)



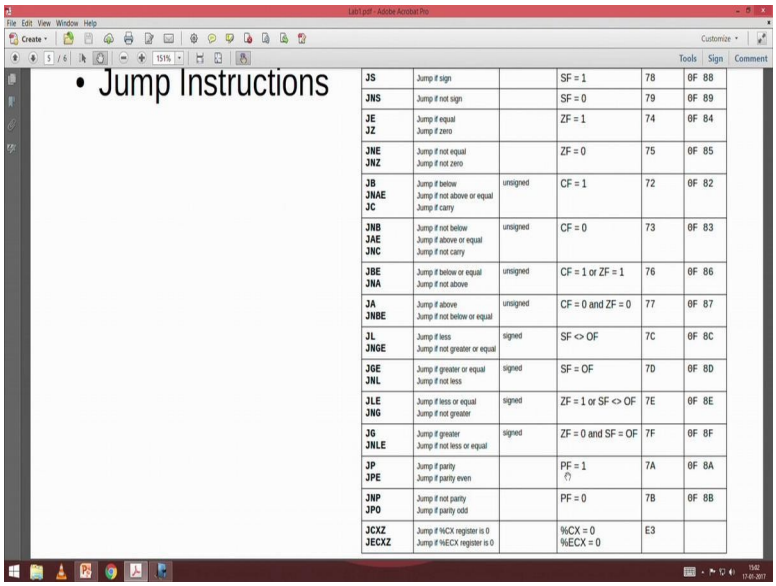
Instruction	Description	signed-ness	Flags	short jump opcodes	near jump opcodes
JO	Jump if overflow		OF = 1	70	0F 80
JNO	Jump if not overflow		OF = 0	71	0F 81
JS	Jump if sign		SF = 1	78	0F 88
JNS	Jump if not sign		SF = 0	79	0F 89
JE JZ	Jump if equal Jump if zero		ZF = 1	74	0F 84
JNE JNZ	Jump if not equal Jump if not zero		ZF = 0	75	0F 85
JB JNAE JC	Jump if below Jump if not above or equal Jump if carry	unsigned	CF = 1	72	0F 82
JNB JAE JNC	Jump if not below Jump if above or equal Jump if not carry	unsigned	CF = 0	73	0F 83
JBE JNA	Jump if below or equal Jump if not above	unsigned	CF = 1 or ZF = 1	76	0F 86
JA JNBE	Jump if above Jump if not below or equal	unsigned	CF = 0 and ZF = 0	77	0F 87
JL JNGE	Jump if less Jump if not greater or equal	signed	SF <> OF	7C	0F 8C
JGE JNL	Jump if greater or equal Jump if not less	signed	SF = OF	7D	0F 8D

Let us go I will just finish this slide and introduce you to the manual specifically with the div instruction. Manual is here, these are all the jump instruction the conditional jump instructions right JO jump on over flow, jump if there is no over flow, jump if the sign bit is one, jump is the sign bit is not is 0, jump when equal, jump if 0, both means the same. JE and J is the task same jump when not equal to is equal to jump not in not 0 jump if below, jump if not above or equal jump if carry, jump if not below, jump if above or equal jump if not carry, jump if below or equal jump if not above, jump if above jump if

not below or equal, jump if less jump if not greater or equal all these things a very quiet easy JLE jump if less than or equal to, JNG jump if not grater both means same.

So, all these acronyms you can use right these are all assembly language instruction and these should be followed by n arithmetic instruction the no this should follow an arithmetic instruction and this will touch those flags. So, what are all those flags OF is the overflow flag, SF is the sign flag, ZF is the 0 flag CF is the carry flag and PF is a parity flag you know what is parity.

(Refer Slide Time: 14:10)

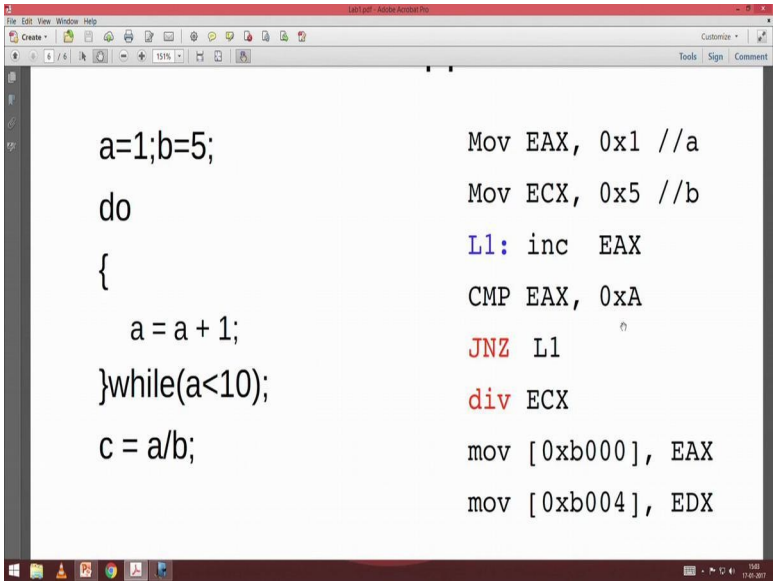


JS	Jump if sign		SF = 1	78	0F 88
JNS	Jump if not sign		SF = 0	79	0F 89
JE JZ	Jump if equal Jump if zero		ZF = 1	74	0F 84
JNE JNZ	Jump if not equal Jump if not zero		ZF = 0	75	0F 85
JB JNAE JC	Jump if below Jump if not above or equal Jump if carry	unsigned	CF = 1	72	0F 82
JNB JAE JNC	Jump if not below Jump if above or equal Jump if not carry	unsigned	CF = 0	73	0F 83
JBE JNA	Jump if below or equal Jump if not above	unsigned	CF = 1 or ZF = 1	76	0F 86
JA JNBE	Jump if above Jump if not below or equal	unsigned	CF = 0 and ZF = 0	77	0F 87
JL JNGE	Jump if less Jump if not greater or equal	signed	SF <> OF	7C	0F 8C
JGE JNL	Jump if greater or equal Jump if not less	signed	SF = OF	7D	0F 8D
JLE JNG	Jump if less or equal Jump if not greater	signed	ZF = 1 or SF <> OF	7E	0F 8E
JG JNLE	Jump if greater Jump if not less or equal	signed	ZF = 0 and SF = OF	7F	0F 8F
JP JPE	Jump if parity Jump if parity even		PF = 1	7A	0F 8A
JNP JPO	Jump if not parity Jump if parity odd		PF = 0	7B	0F 8B
JCXZ JECXZ	Jump if %CX register is 0 Jump if %ECX register is 0		%CX = 0 %ECX = 0	E3	

Student: (Refer Time: 14:11).

Odd, odd, even parity if in a byte if you have even number of ones then C 1 parity, odd number of ones it is a odd parity. So, if this will check parity flag is for even parity; that means, it will be set to one if the result is even parity arrays are parity and then we have some conditional resistors CX; if the CX is 0 ECX is 0. So, all these different types of jump will jump conditional jump instructions are there; and this you can use these combinations to model any loop any type of condition within a loop. So, that is very very important.

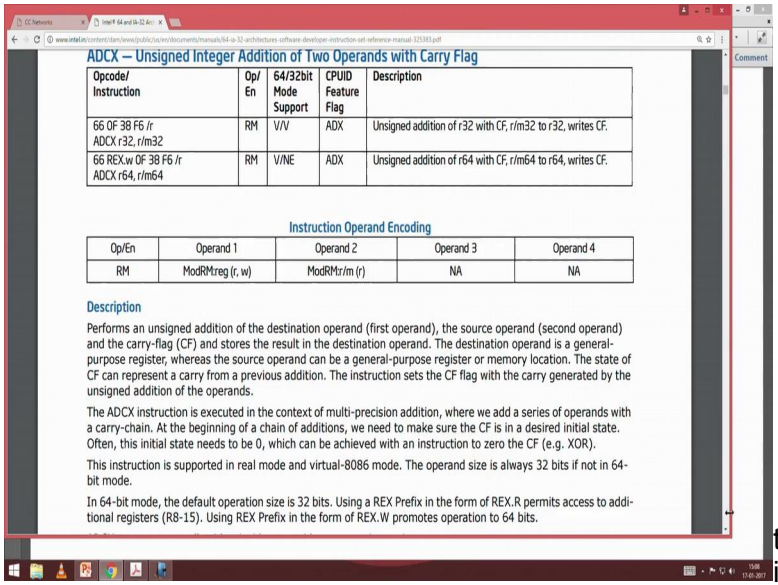
(Refer Slide Time: 14:51)



Now this is another snippet where a is 1 and the ECX is 5, but I go and do I do a while rather than a while loop; that means, surely it will be executed at once. So, I just do increment EAX then I compare, then if it is non 0 I go up if it is 0 I just come down and do again div ECX and so on. So, this is how a do while loop can be translated, and you see there is a one two one translation here whenever I encounter a loop put a label there right and then check the condition there itself if it is a while loop like what we saw earlier if the while loop we immediately check that condition and do the necessary jumping, if it is a do while then put the body of the loop here then go and check the condition and do in necessary jumping. So, that is it now we will just go into the manual.

Now, about this manual you can read in your summer vacation, instruction format will take quite long time for you to understand instruction to understand that is not necessary let us go to chapter three in instructions set reference.

(Refer Slide Time: 16:38)



ADCX — Unsigned Integer Addition of Two Operands with Carry Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F6 /r ADCX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with CF, r/m32 to r32, writes CF.
66 REX.w 0F 38 F6 /r ADCX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with CF, r/m64 to r64, writes CF.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the carry-flag (CF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of CF can represent a carry from a previous addition. The instruction sets the CF flag with the carry generated by the unsigned addition of the operands.

The ADCX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we need to make sure the CF is in a desired initial state. Often, this initial state needs to be 0, which can be achieved with an instruction to zero the CF (e.g. XOR).

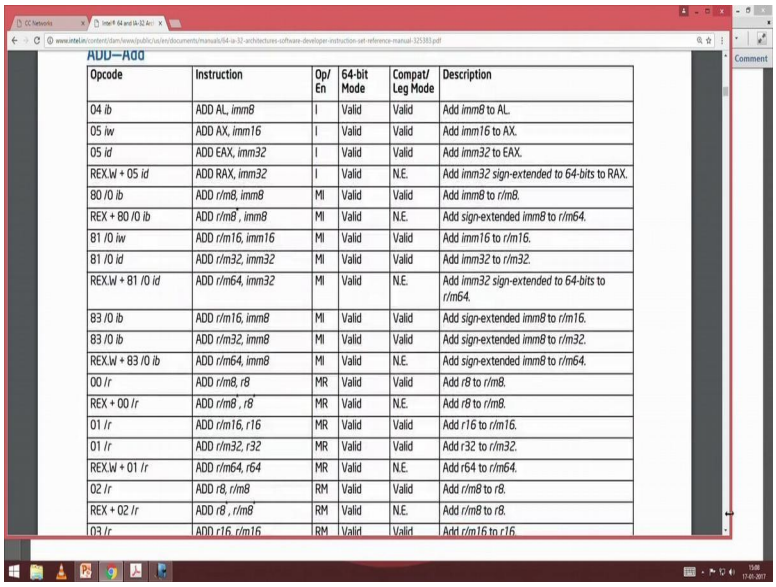
This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64 bits.

So, our whole things starts with AAA instruction ASCII adjust after addition that it 318.

So, we can go to 318. So, let us take the add instructions add.

(Refer Slide Time: 16:49)



ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8*.
REX + 80 /0 *ib*	ADD *r/m8*[*], imm8*	MI	Valid	N.E.	Add *sign-extended imm8* to *r/m64*.
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16*.
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32*.
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64*.
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign-extended imm8* to *r/m16*.
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign-extended imm8* to *r/m32*.
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign-extended imm8* to *r/m64*.
00 /r	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8*.
REX + 00 /r	ADD *r/m8*[*], r8*[*]	MR	Valid	N.E.	Add *r8* to *r/m8*.
01 /r	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16*.
01 /r	ADD *r/m32, r32*	MR	Valid	Valid	Add *r32* to *r/m32*.
REX.W + 01 /r	ADD *r/m64, r64*	MR	Valid	N.E.	Add *r64* to *r/m64*.
02 /r	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8*.
REX + 02 /r	ADD *r8*[*], r/m8*[*]	RM	Valid	N.E.	Add *r/m8* to *r8*.
03 /r	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16*.

So, these are all the varieties of add you can have, we can just say add AL with an immediate 8 AL you know I explain what is AL right add AL with immediate 8 immediate 8 is an 8 bit operand 8 bit value right immediate operand; add AX with immediate 16,

add EAX with immediate 32, add RAX, RAX is for the 64 bit extension there also we can immediate 32 you can add r slash m 8 with immediate 8, r slash m 8 means a the left hand side operand can be a resister of 8 bit or a memory location of 8 bits r slash m 8 with immediate 8 r add r slash m 8 with immediate.

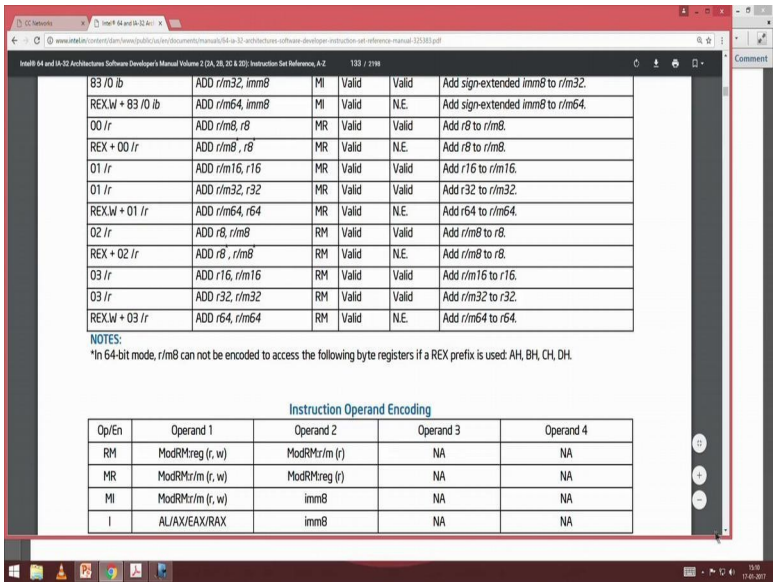
Student: Destination of the r (Refer Time: 17:53).

Destination will be first is destination second is source always.

Student: We have like r a comma b stores in.

Add AL comma immediate 8; obviously, AL should be the destination right immediate 8 is the constant. So, from that you can infer. So, always first is the destination second is the source. So, what do you mean by add r slash m 8 add; immediate 8 add the immediate 8 to the content of a resistor 8 bit resistor or a content of and 8 bit memory meaning one byte right. Similarly r slash m 16 with immediate 8 r slash m 32 all varieties of add which human being can never conceive of all these are there.

(Refer Slide Time: 18:49)



83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32*.
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64*.
00 /r	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8*.
REX + 00 /r	ADD *r/m8**, *r8**	MR	Valid	N.E.	Add *r8* to *r/m8*.
01 /r	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16*.
01 /r	ADD *r/m32, r32*	MR	Valid	Valid	Add *r32* to *r/m32*.
REX.W + 01 /r	ADD *r/m64, r64*	MR	Valid	N.E.	Add *r64* to *r/m64*.
02 /r	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8*.
REX + 02 /r	ADD *r8**, *r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8*.
03 /r	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16*.
03 /r	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32*.
REX.W + 03 /r	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64*.

NOTES:
*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

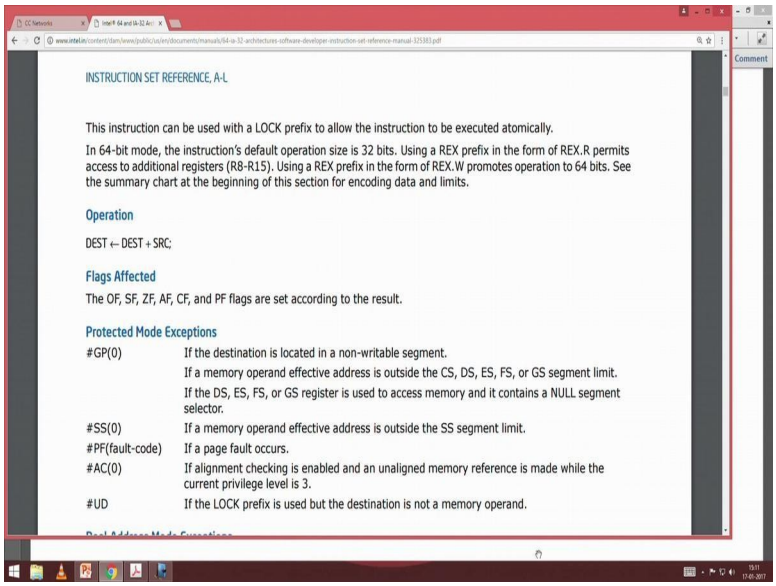
Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AX/EAX/RAX	imm8	NA	NA

I can have add r 64 comma r slash m 64 this for the 64 bit operations also then at the end please note that. So, this is the operation, operation will be destination is destination plus source. So, one will be destination, destination will be destination plus source and please note that this instruction importantly please note here it will effect these flags what are the flags it is going to effect? The overflow of flag, sing flag, 0 flag, axillary flag c is CF is somewhat.

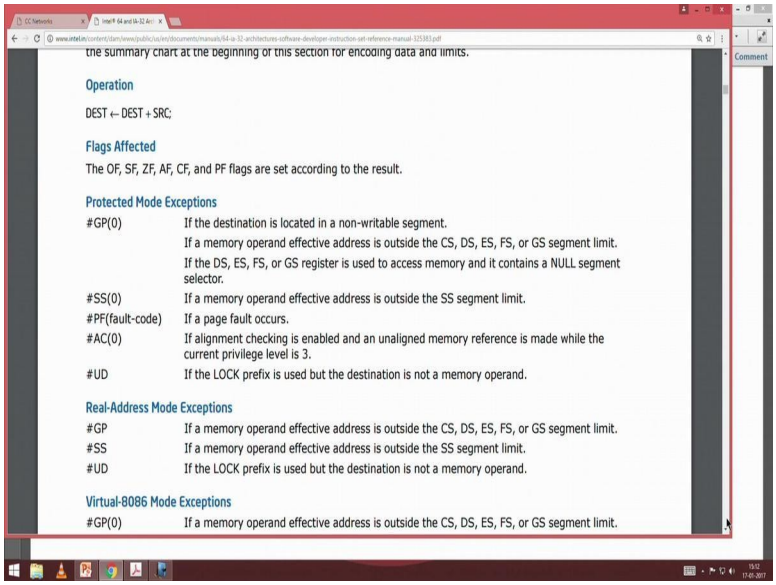
Student: Carry.

Carry flag and PF is parity flag; all these flags it will affect then it will also it will can also give you an assumption exception.

(Refer Slide Time: 19:25)



(Refer Slide Time: 19:32)



For example; it can give you memory divide this is add. So, if your memory operand is outside your segment, it can give you if the destination is located in a non writable

segment, if a memory operand effective address is outside segment limit if the if there. So, this is as I told you yesterday we are talking about protection right. So, we will talk about this in great detail, but it can give you a set of exceptions if you are trying to

access memory which is which you are not allowed to access both for reading and writing. So, are you able to get and understanding of this now let me go into something and say this. So, let me say I want to move sorry let us add EAX comma, 0 x 1000 what it will do what will this instruction do? It will take the it will take the 32 bits starting at 1000, 1001, 1002, 1003 stored at these four, we will take that 32 bits added with the content of EAX and store it back in the EAX.

So, now if I say add Ax comma, 0 x 1000 it will take the 16 bits that are stored in 1000 1001 and added it with A x and store the resultant A x; if I say add AL comma 0 x 1000 then it will take.

Student: (Refer Time: 21:32).

8 bits that are stored in 0 x 1000 alone add it with AL and store the result back. Now let us take these three instructions add EAX, add A x add AL see the Op code, see this line basically talks of the Op code this tells you what the operation is both for r 16 and r 32 the op code is the same right, but what it will do is there is an prefix that will be added right there will be a prefix that will be added in front of this 0 1 which will say that this is a 16 bit instruction. So, please note that the op code is same to distinguish between whether it is an r 16, m 16 comma r 16, or m 32 comma r 32, we cannot do it here because r the Op code is the same. So, before the op code there will be a prefix that will be adder which will say that this is the 32 bit instruction r 16 bit instruction.

So, for every op code there will be a prefix which will distinguish whether it is a 16 bit instruction or 32 bit instruction. In some sense Intel was too lazy it has finish the 16 bit encoding, now to convert to 32 bit it did not want to do another fresh encoding. So, what it does is in a 16 bit instruction if I put a 32 bit prefix in front of that 16, bit instruction it will start behaving like a 32 bit instruction we are able to get this right. So, this is how Intel mood from a 16 bit architecture to a 32 bit architecture you can have a 16 bit instruction.

Now, if I want to make it a 32 bit put one prefix in front of 16 bit then it becomes a 32 bit instruction. So, this is how you start reading this manual it does not looks carry. So, first two or three instructions you will struggle after that you will get used it. So, with this I stop my introduction any doubts do not tell me you did not have doubts you will have doubts.

Student: Sir.

Yeah.

Student: Where are the op codes (Refer Time: 24:10).

Where are the.

Student: Op codes (Refer Time: 24:14).

It is in the manual.

Student: (Refer Time: 24:16).

When you compile it you get Op codes right the op codes are the one which the machine can understand correct. So, when you compile the c code for example, good question. So, this is the c code this is a human readable form if I write 0 1 1 and 0 you cannot read it right what you see on the right hand side sorry what you see on the right hand side where my hand is rotating right this is a human readable form of your op code, all your move add and all will be there is an equivalent zeros and ones, but I cannot rise zeros and ones here because then you cannot understand that.

So, this is called an mnemonic which is which is a readable form of your op code, but what you see on the right hand side is basically zeros and ones it will be represented zeros and ones these are the things that will go into your system and your machine your architecture will understand only those zeros and ones, it cannot understand the C program that you see on the left hand side it will understand the assembly program that you see on the right hand side.

So, this move compile excreta or human readable versions of your op code, ultimately this are this will be in binary and this is what your system will understand system cannot understand do while loop and all, but it can answer understand this. So, this is what we will go into your computer into your architecture and architecture we will execute right are you able to follow yes or no.

Student: Yes sir.

Yes, fine enough.