Assembly Language Programming

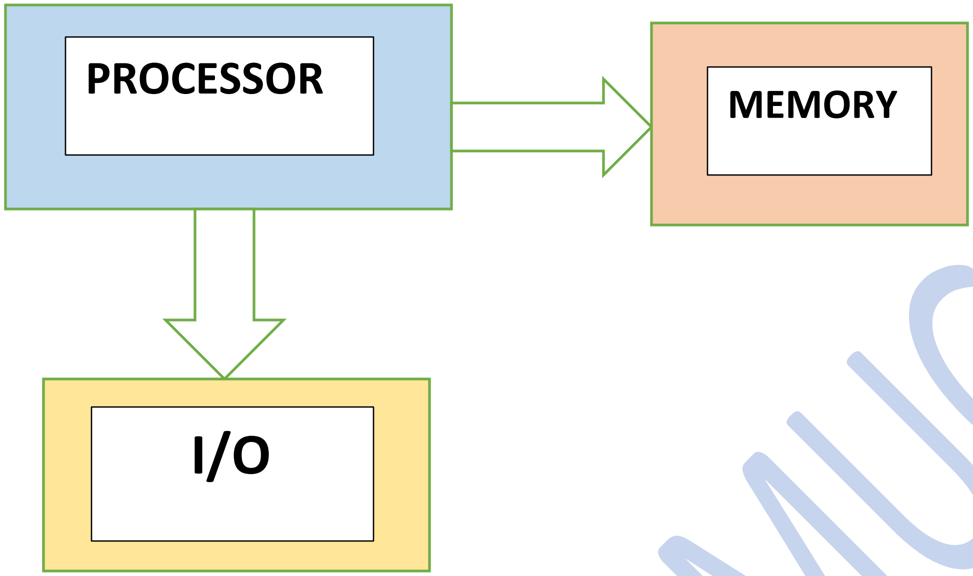
Lecture Notes CS401

MC180402118

MIT DEPARTMENT

what is Basic Computer Architecture?

2 **Basic Computer Architecture**. The main components in a typical **computer** system are the processor, memory, input/output devices, and the communication channels that connect them. ... In a typical system there will be only one processor, known at the central processing unit, or CPU.



What is a computer program?

A Program is a set of instructions compiled together in a file to perform some specific task by the CPU (Central Processing Unit). It is a series of binary numbers (0s and 1s) arranged in a sequence, which when given to the computer performs some task.

Computer is a dumb machine with expeditious computational speed. It can give quick results to many of the complex scientific calculations but it can’t perform a task on its own. A computer needs set of instructions to do some task. This set of instructions is contained in a computer program. Computer program is basically in binary language i.e. series of 0s and 1s. A large bunch of programs makes the computer functional without which the computer would be like a paralyzed machine.

You may think computer as an idiot person who does not know to cook. If you provide ingredients for cooking *Pasta* to that idiot person, you cannot expect a delicious dish. However, if you provide ingredients along with the full step-by-step recipe of cooking *Pasta* then you may expect a real *Pasta* from that idiot person. Same is the concept with computers, for computers the ingredients are data (might be an integer, string, images, videos or anything) and the recipe is a program.

What is Programming?

Programming is the process of writing an algorithm into a sequence of computer instructions. Or you can simply say it is the process of writing programs. We generally transform the solution of a specific problem into computer language. It is the only way through which we can create our own programs and can execute them on a computer. Programming requires skill, logical thinking and lots of experience.

The world's first programmer was Ada Lovelace. She was widely known for her work on Charles Babbage's Analytical Engine (general-purpose mechanical computer).

World's first programmer - Ada Lovelace

What is assembly language example?

Assembly language is an example of low-level **language**. In an **assembly language**, each machine **language** instruction is assigned a **code**. So, instead of having to remember a string of 0's and 1's, the programmer would only need to remember short codes like ADD, SUB, DIV, JMP, MOV, HALT, GO called mnemonics.

What is machine code or language?

Machine code is a computer program written in machine language instructions that can be executed directly by a computer's central processing unit.

a computer programming language consisting of binary or hexadecimal instructions which a computer can respond to directly.

What is machine language example?

Below is an **example** of **machine language** (binary) for the text "Hello World". 01001000 01100101 01101100 01101100 01101111 00100000 01010111 01101111 01110010 01101100 01100100. Below is another **example** of **machine language**(non-binary), which will print the letter "A" 1000 times to the computer screen.

what is low level and high-level programming language?

A **high-level language** (HLL) is a **programming language** such as C, FORTRAN, or Pascal that enables a **programmer** to write programs that are more or less independent of a particular type of **computer**. ... In contrast, assembly **languages** are considered **low-level** because they are very close to machine **languages**.

DIFFERENCE BETWEEN THE HIGH AND THE LOW-LEVEL LANGUAGE:

Low level language is machine readable form of program. Whereas the **high level language** will be **in** human readable form... ... **High level language** uses compilers and interpreters which requires **large** memory space.

What are the advantages and disadvantages of low level and high-level programming languages?

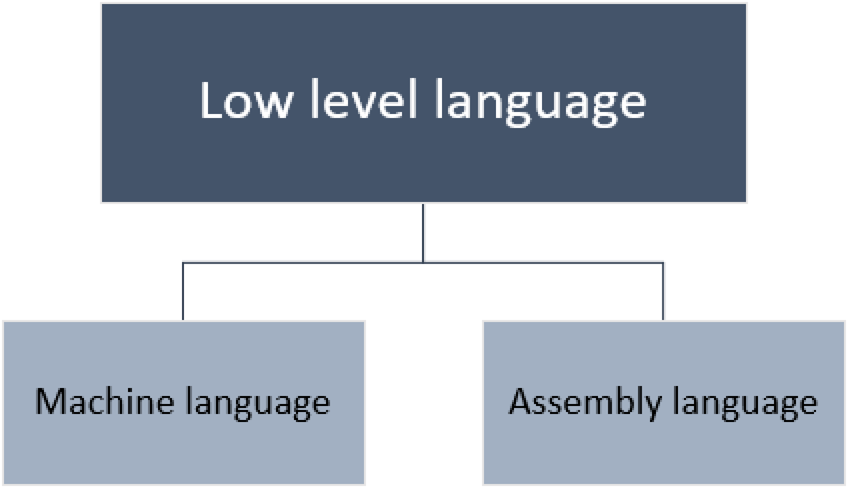
Advantages of low-level languages. Programs developed using low level languages are **fast** and **memory efficient**. Programmers can utilize processor and **memory** in better way using a low-level language. There is no need of any compiler or interpreters to translate the source to machine code.

Low level language abbreviated as **LLL**, are languages close to the machine level instruction set. They provide less or no abstraction from the hardware. A low-level programming language interacts directly with the registers and memory. Since, instructions written in low level languages are machine dependent. Programs developed using low level languages are machine dependent and are not portable.

Low level language does not require any compiler or interpreter to translate the source to machine code. An assembler may translate the source code written in low level language to machine code.

Programs written in low level languages are fast and memory efficient. However, it is nightmare for programmers to write, debug and maintain low-level programs. They are mostly used to develop operating systems, device drivers, databases and applications that requires direct hardware access.

Low level languages are further classified in two more categories – Machine language and assembly language.



Machine language

Machine language is closest language to the hardware. It consists set of instructions that are executed directly by the computer. These instructions are a sequence of binary bits. Each instruction performs a very specific and small task. Instructions written in machine language are machine dependent and varies from computer to computer.

Example: SUB AX, BX = **00001011 00000001 00100010** is an instruction set to subtract values of two registers **AX** and **BX**.

In the starting days of programming, program was only written in machine language. Each and every program were written as a sequence of binaries.

A Programmer must have additional knowledge about the architecture of the particular machine, before programming in machine language. Developing programs using machine language is tedious job. Since, it is very difficult to remember sequence of binaries for different computer architectures. Therefore, nowadays it is not much in practice.

Assembly language

Assembly language is an improvement over machine language. Similar to machine language, assembly language also interacts directly with the hardware. Instead of using raw binary sequence to represent an instruction set, assembly language uses **mnemonics**.

Mnemonics are short abbreviated English words used to specify a computer instruction. Each instruction in binary has a specific mnemonic. They are architecture dependent and there is a list of separate mnemonics for different computer architectures.
Examples of mnemonics are – ADD, MOV, SUB etc.

Mnemonics gave relief to the programmers from remembering binary sequence for specific instructions. As English words like **ADD, MOV, SUB** is easy to remember, than binary sequence 10001011. However, programmer still have to remember various mnemonics for different computer architectures.

Assembly language uses a special program called **assembler**. Assembler translates mnemonics to specific machine code.

Assembly language is still in use. It is used for developing operating systems, device drivers, compilers and other programs that requires direct hardware access.

Advantages of low-level languages

1. Programs developed using low level languages are fast and memory efficient.
2. Programmers can utilize processor and memory in better way using a low-level language.
3. There is no need of any compiler or interpreters to translate the source to machine code. Thus, cuts the compilation and interpretation time.
4. Low level languages provide direct manipulation of computer registers and storage.
5. It can directly communicate with hardware devices.

Disadvantages of low-level languages

1. Programs developed using low level languages are machine dependent and are not portable.
2. It is difficult to develop, debug and maintain.
3. Low level programs are more error prone.
4. Low level programming usually results in poor programming productivity.
5. Programmer must have additional knowledge of the computer architecture of particular machine, for programming in low level language.

High level languages – advantages and disadvantages

High level language is abbreviated as **HLL**. High level languages are similar to the human language. Unlike low level languages, high level languages are programmers friendly, easy to code, debug and maintain.

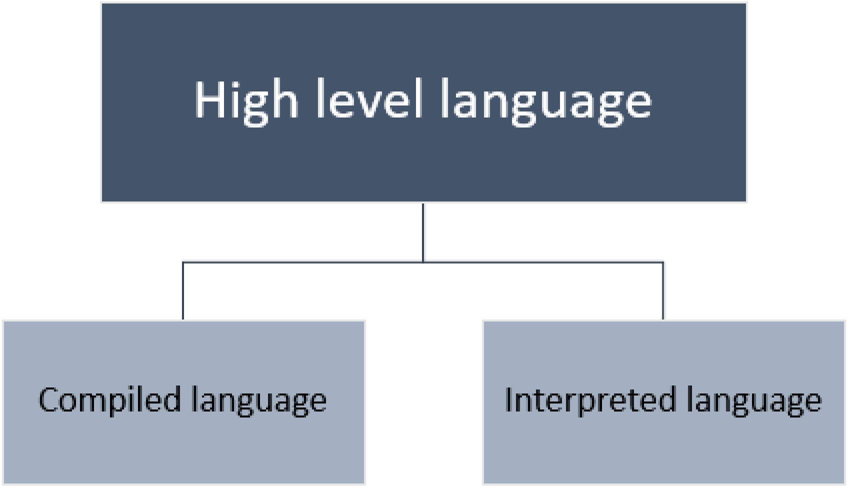
High level language provides higher level of abstraction from machine language. They do not interact directly with the hardware. Rather, they focus more on the complex arithmetic operations, optimal program efficiency and easiness in coding.

Low level programming uses machine friendly language. Programmers writes code either in binary or assembly language. Writing programs in binary is complex and cumbersome process. Hence, to make programming more programmers friendly. Programs in high level language is written using English statements.

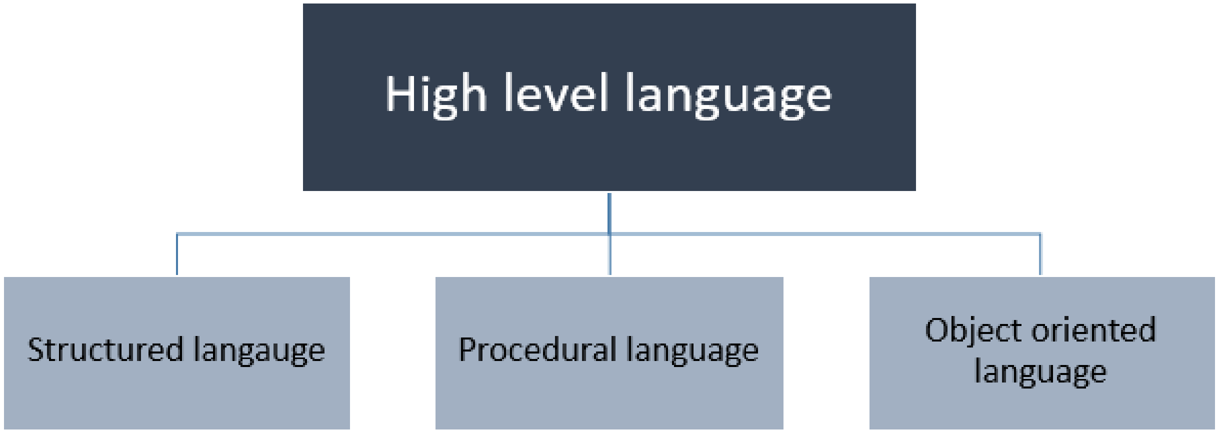
High level programs require compilers/interpreters to translate source code to machine language. We can compile the source code written in high level language to multiple machine languages. Thus, they are machine independent language.

Today almost all programs are developed using a high-level programming language. We can develop a variety of applications using high level language. They are used to develop desktop applications, websites, system software's, utility software's and many more.

High level languages are grouped in two categories based on execution model – compiled or interpreted languages.



We can also classify high level language several other categories based on programming paradigm



Advantages of High-level language

1. High level languages are programmer friendly. They are easy to write, debug and maintain.
2. It provides higher level of abstraction from machine languages.
3. It is machine independent language.
4. Easy to learn.
5. Less error prone, easy to find and debug errors.
6. High level programming results in better programming productivity.

Disadvantages of High-level language

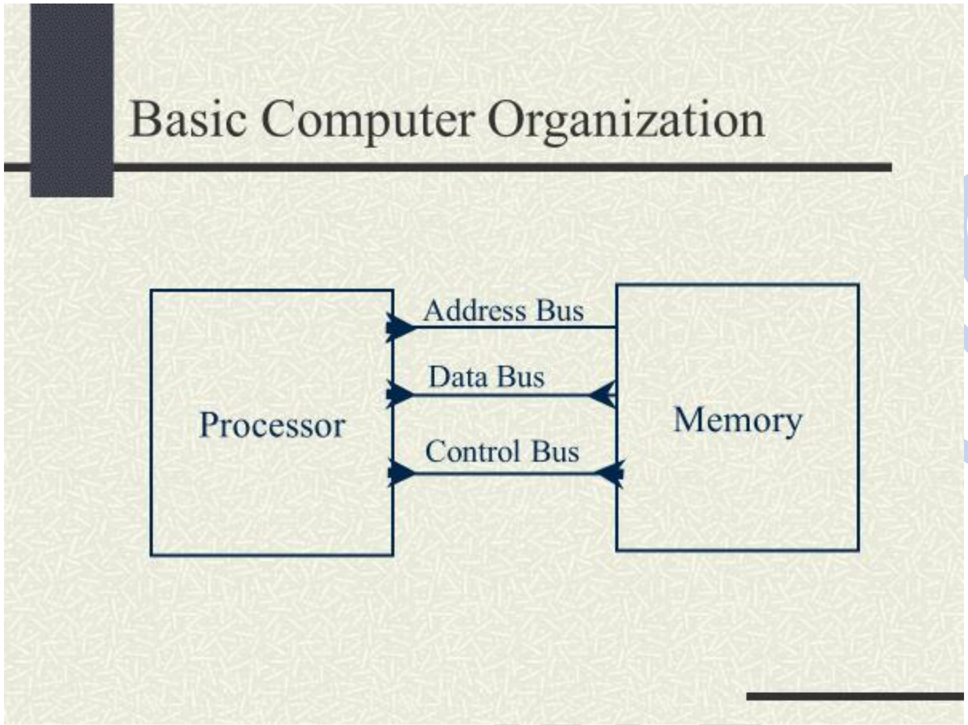
1. It takes additional translation times to translate the source to machine code.
2. High level programs are comparatively slower than low level programs.
3. Compared to low level programs, they are generally less memory efficient.
4. Cannot communicate directly with the hardware.

Differences between low level and high-level programming language.

Summing up the differences between low level and high-level programming language.

Low level language	High level language
They are faster than high level language.	They are comparatively slower.
Low level languages are memory efficient.	High level languages are not memory efficient.
Low level languages are difficult to learn.	High level languages are easy to learn.
Programming in low level requires additional knowledge of the computer architecture.	Programming in high level do not require any additional knowledge of the computer architecture.
They are machine dependent and are not portable.	They are machine independent and portable.

They provide less or no abstraction from the hardware.	They provide high abstraction from the hardware.
They are more error prone.	They are less error prone.
Debugging and maintenance is difficult.	Debugging and maintenance is comparatively easier.
They are generally used for developing system software's (Operating systems) and embedded applications.	They are used to develop a variety of applications such as – desktop applications, websites, mobile software's etc.



Basic Computer Organization

WHAT IS REGISTER?

a **register** is a small bit of memory that sits inside the CPU. and is used by **assembly language** to perform various tasks. ... Well, you have general purpose **registers**, then you have **registers** which have special usage (for **example**, the program counter **registers**), and you have various others (memory/segment **registers**, SSE)

How many registers are in assembly language?

A register is a part of the processor that can hold a bit pattern. On the MIPS, a register holds **32** bits. There are many registers in the processor, but only some of them are visible in assembly language. The others are used by the processor in carrying out its operations.

What is the purpose of registers in CPU?

A **processor register** (**CPU register**) is one of a small set of data holding places that are part of the **computer processor**. A **register** may hold an instruction, a storage address, or any kind of data (such as a bit sequence or individual characters). Some instructions specify **registers** as part of the instruction.

How many bytes can a register hold?

a 64-bit processor has 8-byte-wide registers and a **32-**bit processor has **4-**byte-wide registers, but when those registers are copied into RAM it's just bytes in memory.

What are the uses of registers?

General purpose register are used to arithmetic (+, -, *,) and logic (>, <=,) operations. Typically, these register are 8-32 bit registers. Accumulator is general purpose register and is used by CPU for performing arithmetic and logic operations and to hold the result of those operations.

What is the purpose of registers?

Purpose of registers. As stated on the previous page, Definition: A **register** is a discrete memory location within the CPU designed to hold temporary data and instructions. When a **register** is being used to move data /instructions from one part of the system to another, this is called a buffer.

Are registers memory?

Registers are the **memory** locations that are directly accessible by the processor. The **registers** hold the instruction or operands that is currently being accessed by the CPU. **Registers** are the high-speed accessible storage elements. The processor accesses the **registers** within one CPU clock cycle

What are the four general purpose registers and their functions?

The **registers** inside the 8086 are all 16 bits. They are split up into **four** categories: **General Purpose**, Index, Status & Control, and Segment. The **four general purpose registers** are the AX, BX, CX, and DX **registers**. AX - accumulator, and preferred for most operations.

What is special purpose register?

Special purpose registers (SPR) hold program state; they usually include the program counter (aka instruction pointer), stack pointer, and status **register** (aka processor status word). In embedded microprocessors, they can also correspond to specialized hardware elements

Types:

Processor **register**: **Special purpose registers** (SPR) hold program state; they usually include the program counter (aka instruction pointer), stack pointer, and status **register** (aka processor status word). In embedded microprocessors, they can also correspond to **specialized** hardware elements

What is the total size of general-purpose registers?

These **registers** may be 8/16/32/64/128-bit depending on its **use** and architecture of an electronic part where it is used (like microprocessor or RAM). The **general-purpose register** can store a data or a memory location address. Hence called as **General-purpose register**. It is a multipurpose **register**.

What is the difference between main memory and registers?

The **primary difference between register** and **memory** is that **register** holds the data that the CPU is currently processing whereas, the **memory** holds the data the that will be required for processing. ... On the other hands, **memory** is referred as the **main memory** of the computer which is RAM.

What are the 4 general and 4 special purpose registers and their function for x86?

The registers viewed as "general-purpose" are EAX, ECX, EDX, and EBX; the registers viewed as "special purpose" are ESP, EBP, ESI, and EDI.

EAX: Basically, the "main register" for everything. It also forms the lower 32 bits of instructions that deal with 64-bit integers. Also, this register is hard-coded into instruction forms like 05 Id: add eax, Id, which is only five bytes long, as compared to instructions that deal with other registers, which can be six to seven bytes long.
ECX: Loop counter! Some instructions like JECXZ and LOOP deal with the ECX register exclusively, and this can be a boon if you have some flags state that you don't want to trash.
EDX: The upper 32 bits of the 64-bit number returned in those 64-bit instructions.

EBX: The "base register".

- *Note*: This is the only "general-purpose register" addressing mode available in 16-bit mode.
  - **ESP**: Stack pointer. Messing with this one can lead to disaster.
  - **EBP**: Base pointer. Unlike other any other register, no instruction specifically modifies this register, unless it's ENTER or LEAVE, which are not used a lot these days. However, this register is often used to get arguments off the stack.
  - **ESI**: String Source index. Used with the string instructions. Other than that, it can be used as general purpose
  - **EDI**: String Destination index. See above.

FROM OTHER SOURCE

general purpose registers

General purpose registers can be used as either data or address registers.

- **DEC VAX:** 16 word (32 bit) general purpose registers; named R0 through R15
- **IBM 360/370:** 16 full word (32 bit) general purpose registers; named 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (or 10), B (or 11), C (or 12), D (or 13), E (or 14), and F (or 15)
- **Intel 8086/80286:** 8 word (16 bit) general purpose registers; named AX, BX, CX, DX, BP, SP, SI, and DI (high order bytes of the AX, BX, CX, and DX registers have the names AH, BH, CH, and DH and low order bytes of the AX, BX, CX, and DX registers have the names AL, BL, CL, and DL)

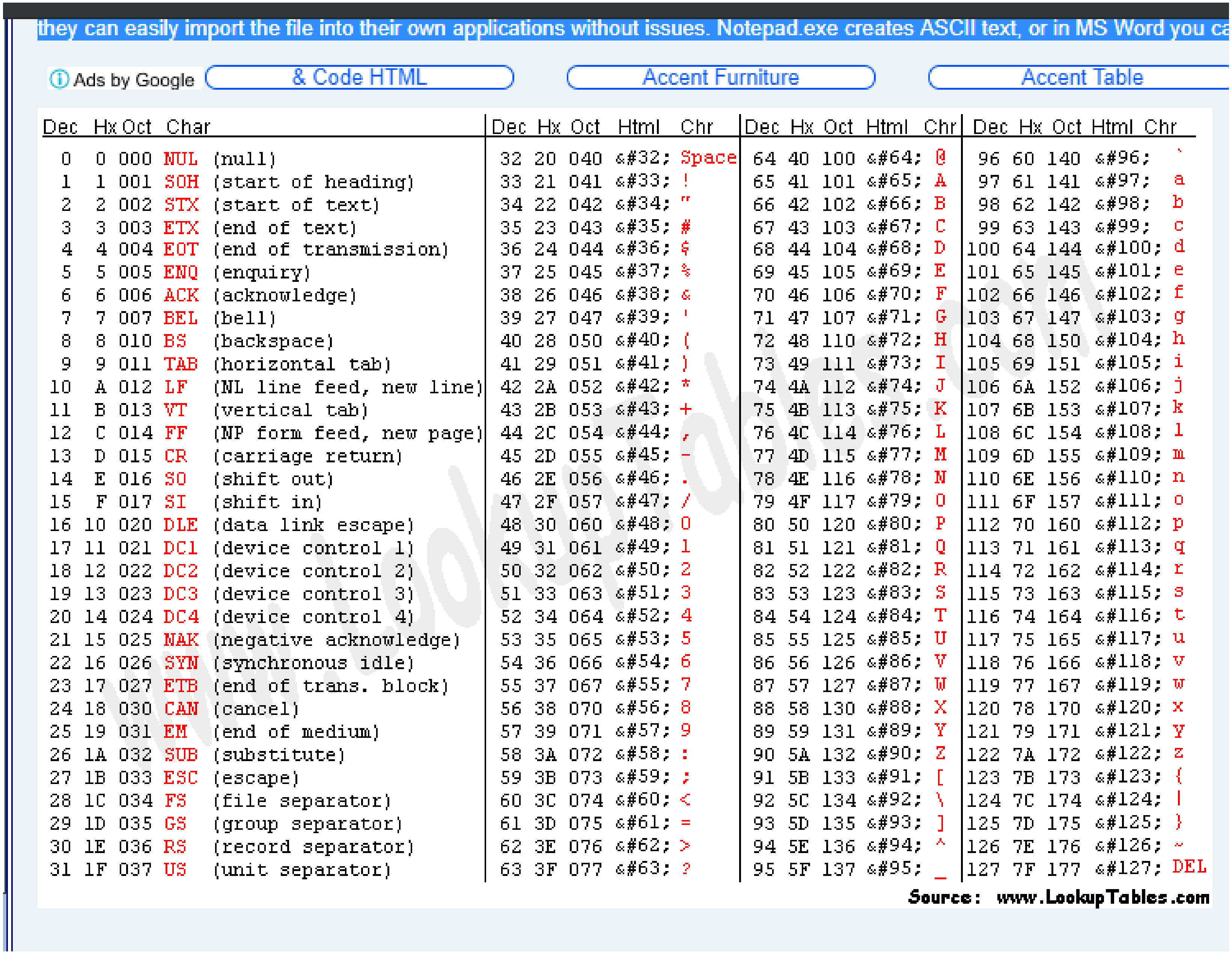
- **Intel 80386:** 8 doubleword (32 bit) general purpose registers; named EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI (low order words use the same names as the general purpose registers on the Intel 8086 and 80286 and low order and high order bytes of the low order words of four of the registers use the same names as the general purpose registers on the Intel 8086 and 80286)
- **Motorola 88100:** 32 word (32 bit) general purpose registers; named r0 through r31

SHORT NOTE

An **assembly language** is a low-level **programming language** designed for a specific type of processor. It may be produced by compiling source **code** from a high-level **programming language** (such as C/C++) but can also be written from scratch. **Assembly code** can be converted to machine **code** using an **assembler**.

ASCII Table and Description

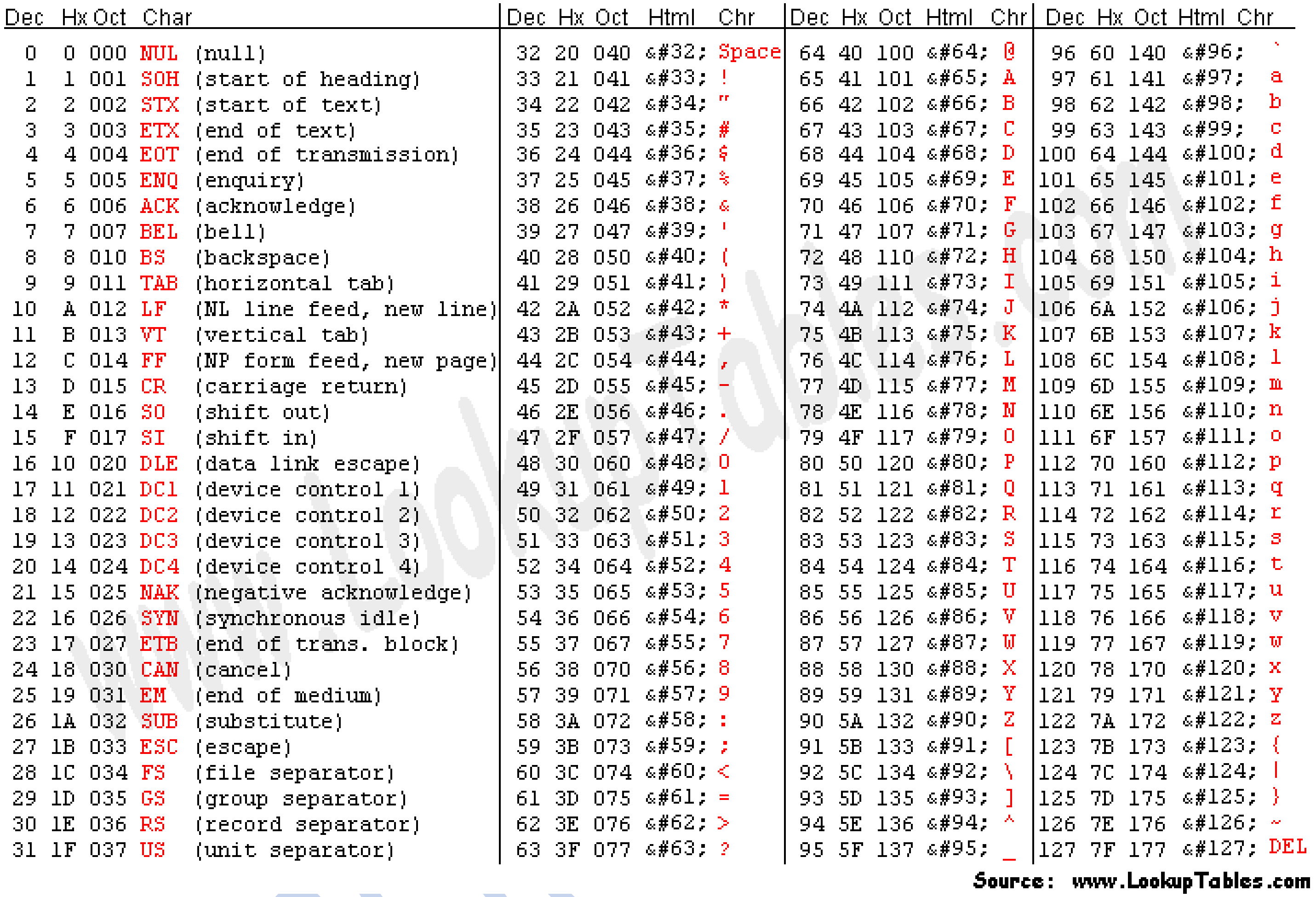
ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort. ASCII was developed a long time ago and now the non-printing characters are rarely used for their original purpose. Below is the ASCII character table and this includes descriptions of the first 32 non-printing characters. ASCII was actually designed for use with teletypes and so the descriptions are somewhat obscure. If someone says they want your CV however in ASCII format, all this means is they want 'plain' text with no formatting such as tabs, bold or underscoring - the raw format that any computer can understand. This is usually so they can easily import the file into their own applications without issues. Notepad.exe creates ASCII text, or in MS Word you can save a file as 'text only'



they can easily import the file into their own applications without issues. Notepad.exe creates ASCII text, or in MS Word you ca

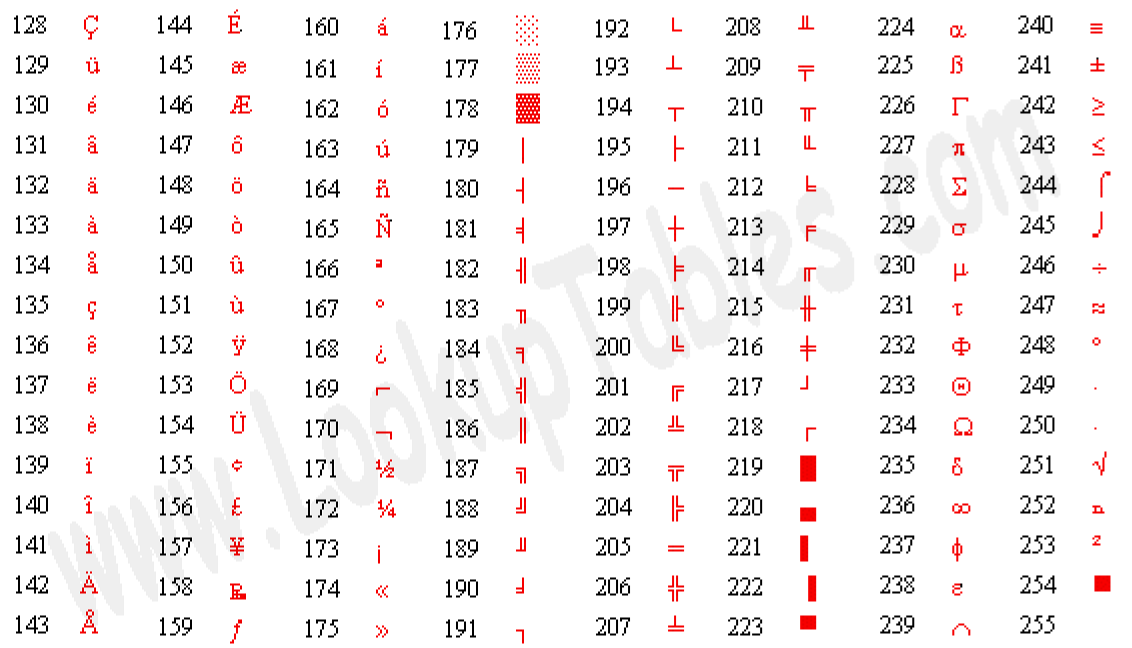
Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com



128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

Lecture # 3

Types of Register

General purpose	**Special purpose**
Used for more than one Purposes.	used for only one purpose/specific purpose

1. AX Accumulator
2. Bx base AX = 16 Bits
3. CX Count EAX =32 Bits
4. DX destination RAX =64 Bits

   X= SIZE

a "**bit**" is atomic: the smallest unit of storage; A **bit** stores just a 0 or 1; "In the ... One **byte** = collection of 8 **bits**; e.g. 0 1 0 1 1 0 1 0; One **byte** can store one ...

, there's a big **difference between** a **bit**and a **byte**. A **byte** is much bigger — eight times bigger, to be exact, with eight **bits**in every **byte**. By extension, there are eight megabits in every megabyte, and one gigabyte is eight times bigger than one gigabit.

Bit. A **bit** (short for "binary digit") is the smallest unit of measurement used to quantify computer data. It contains a single binary value of 0 or 1. ... For **example**, a small text file that is 4 KB in size contains 4,000 bytes, or 32,000 **bits**

What is the biggest byte size?

Kilobyte (1024 Bytes)

Megabyte (1024 Kilobytes)

Gigabyte (1,024 Megabytes, or 1,048,576 Kilobytes)

Terabyte (1,024 Gigabytes)

Petabyte (1,024 Terabytes, or 1,048,576 Gigabytes)

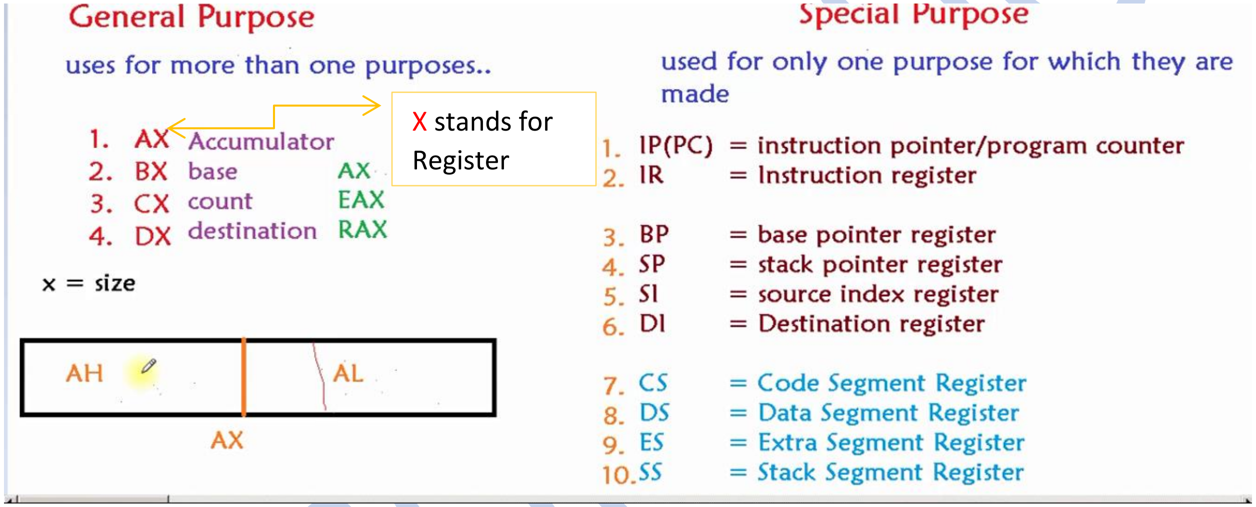
Exabyte (1,024 Petabytes)

Zettabyte (1,024 Exabytes)

Yottabyte (1,204 Zettabytes, or 1,208,925,819,614,629,174,706,176 bytes

We divide general purpose registers in two parts

AH(HIGHER)16BIT	AL(LOWER)16 BIT
AX=32	



General Purpose

uses for more than one purposes..

X stands for Register

1. AX Accumulator
2. BX base
3. CX count
4. DX destination

AX
EAX
RAX

x = size

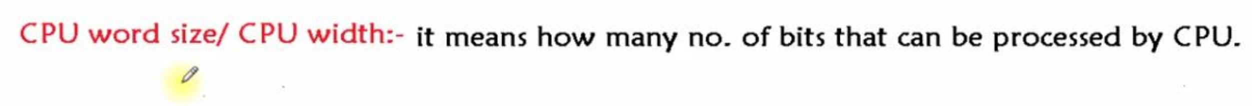
Special Purpose

used for only one purpose for which they are made

1. IP(PC) = instruction pointer/program counter
2. IR = Instruction register

3. BP = base pointer register
4. SP = stack pointer register
5. SI = source index register
6. DI = Destination register

7. CS = Code Segment Register
8. DS = Data Segment Register
9. ES = Extra Segment Register
10. SS = Stack Segment Register



CPU word size/ CPU width:- it means how many no. of bits that can be processed by CPU.

AccumulatorAX: 32BITS

It involves in every operation.

If accumulator is 32 bits, then CPU also will be 32 bits and wise versa.

Accumulators

Accumulator

There is a central register in every processor called the accumulator. Traditionally all mathematical and logical operations are performed on the accumulator. The word size of a processor is defined by the width of its accumulator. A 32bit processor has an accumulator of 32 bits

Accumulators are registers that can be used for arithmetic, logical, shift, rotate, or other similar operations. The first computers typically only had one accumulator. Many times there were related special purpose registers that contained the source data for an accumulator. Accumulators were replaced with data registers and general purpose registers. Accumulators reappeared in the first microprocessors.

- **Intel 8086/80286:** one word (16 bit) accumulator; named AX (high order byte of the AX register is named AH and low order byte of the AX register is named AL)
- **Intel 80386:** one doubleword (32 bit) accumulator; named EAX (low order word uses the same names as the accumulator on the Intel 8086 and 80286 [AX] and low order and high order

bytes of the low order words of four of the registers use the same names as the accumulator on the Intel 8086 and 80286 [AH and AL])

- **MIX:** one accumulator; named A-register; five bytes plus sign

Base RegisterBX: 16 BITS

Pointer / Index / Base

Function:

Holds the Address of Operands

Base register is used to store starting address of data structure.

Suppose we have an array, and array name represents starting address which is also called **Base address.**

base registers

Base registers or **segment registers** are used to segment memory. Effective addresses are computed by adding the contents of the base or segment register to the rest of the effective address computation. In some processors, any register can serve as a base register. In some processors, there are specific base or segment registers (one or more) that can only be used for that purpose. In some processors with multiple base or segment registers, each base or segment register is used for different kinds of memory accesses (such as a segment register for data accesses and a different segment register for program accesses).

- **IBM 360/370:** any of the 16 general purpose registers may be used as a base register
- **Intel 80x86:** 6 dedicated segment registers: CS (code segment), SS (stack segment), DS (data segment), ES (extra segment, a second data segment register), FS (third data segment register), and GS (fourth data segment register)
- **Motorola 680x0, 68300:** any of the 8 address registers may be used as a base register

Counter registerCX: 32 BITS

Counter remembers how many times a loop will run,

Example:

For(int i ;i< 10 i++)

In this example counter register will remember that this loop is going to be run 9 times.

A **counter** is a special case of a **register**. Usually, it can only be loaded, stored, or incremented, or used for the stack or as the program **counter**. A **register** can hold data, and it can be used for temporary storage or, **in the** case of an accumulator, it can participate in arithmetic or logical operations.

Destination RegisterDX: 64 BITS

In this register we store final output of our operations.

Let’s suppose we are adding two numbers and it gives answer of 38 bits as following:

Num1 + Num2 = 38 bits answer

But our accumulator can save only 32 bits, remaining 6bits will be carried by destination register to save our data.

The DI (**destination** Index) **register** performs the same functions as SI. There is a class of instructions, called string operations, that use DI to access memory locations addressed by ES.

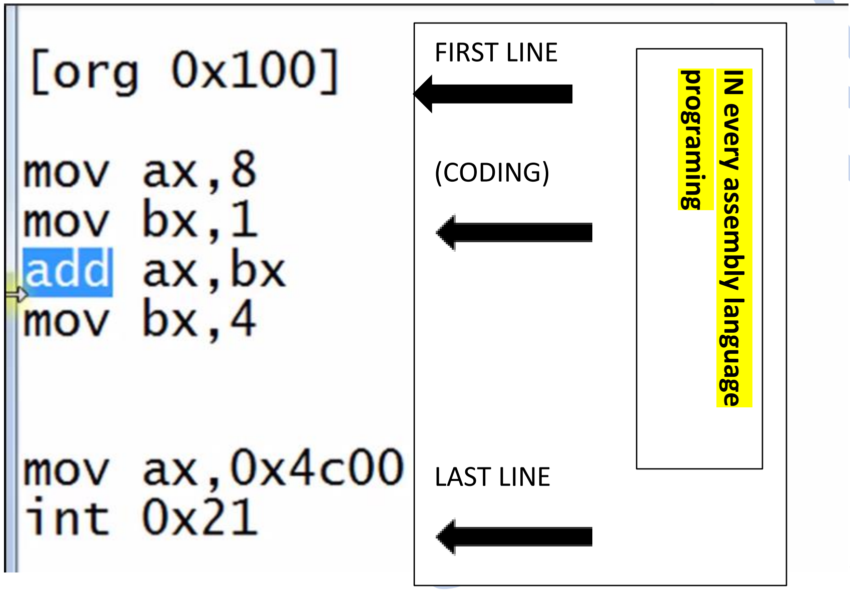
AGAIN

What are registers in assembly language?
a **register** is a small bit of memory that sits inside the CPU. and is used by**assembly language** to perform various tasks. ... Well, you have general purpose**registers**, then you have **registers** which have special usage (for **example**, the program counter **registers**), and you have various others (memory/segment**registers**, SSE)

What is ax in assembly language?
AX is the primary accumulator; it is used in input/output and most arithmetic instructions.
For **example**, in multiplication operation, one operand is stored in EAX or **AX** or AL register according to the size of the operand.

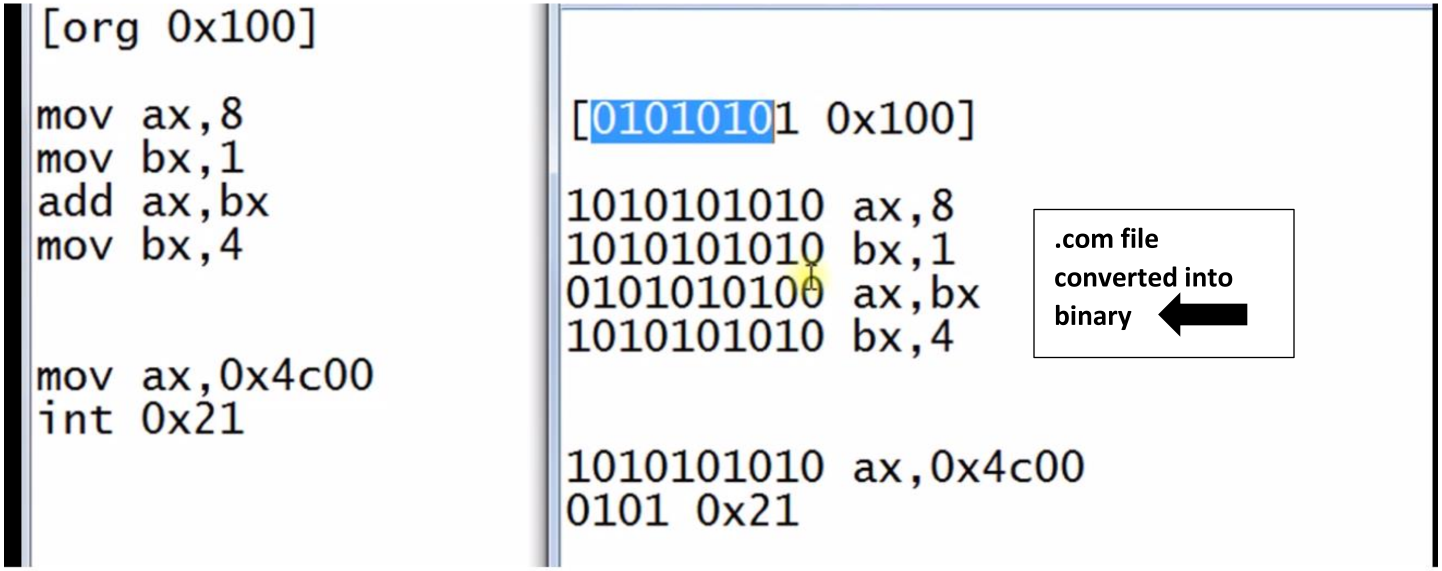
A register is a part of the processor that can hold a bit pattern. On the MIPS, a register holds **32** bits. There are many registers in the processor, but only some of them are visible in assembly language. The others are used by the processor in carrying out its operations.

The **type of register** that hold data and addresses is called general purpose**register**. General purpose **register** are used to arithmetic (+,-,*,) and logic (>, <=,) operations. Typically theses **register** are 8-32 bit **registers**.



```
[org 0x100]

mov  ax,8
mov  bx,1
add  ax,bx
mov  bx,4


mov  ax,0x4c00
int  0x21
```

assemble convert code into binary



```
[org 0x100]

mov ax,8
mov bx,1
add ax,bx
mov bx,4


mov ax,0x4c00
int 0x21
```

```
[01010101 0x100]

1010101010 ax,8
1010101010 bx,1
0101010100 ax,bx
1010101010 bx,4


1010101010 ax,0x4c00
0101 0x21
```

.com file converted into binary

Nsam converted code into binary

Lecture 4

Programing Overview

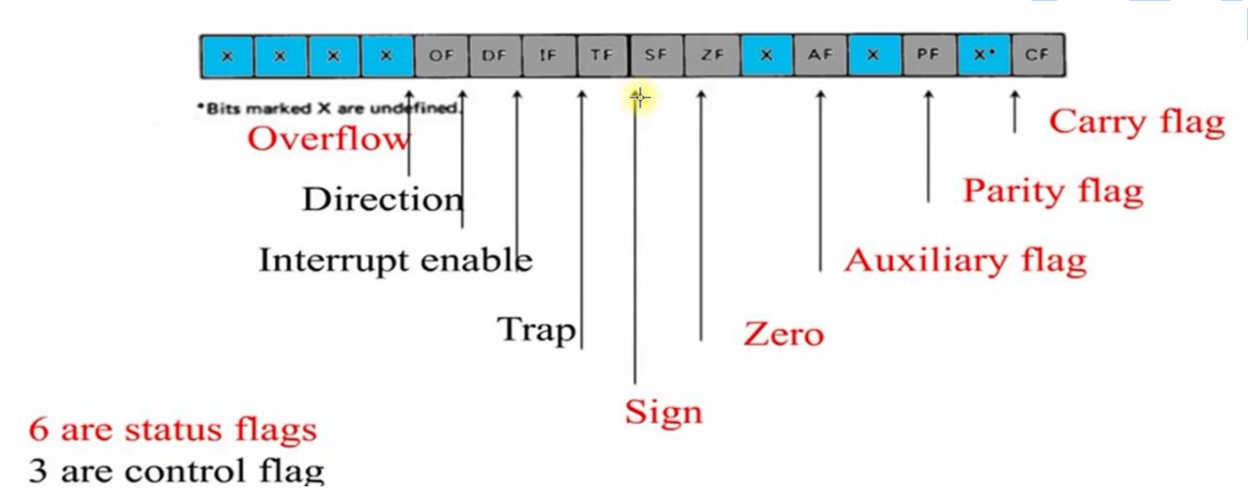
Lecture 5

Flag Register

Flag / Program Status Word

Function:

Collection of different boolean information each bit has an independent meaning



Carry Flag:

In eighty, eighty-six architecture, flag registers have 16 bits.

In this case Flag register has **9 bits**and there are also some **empty bits**.

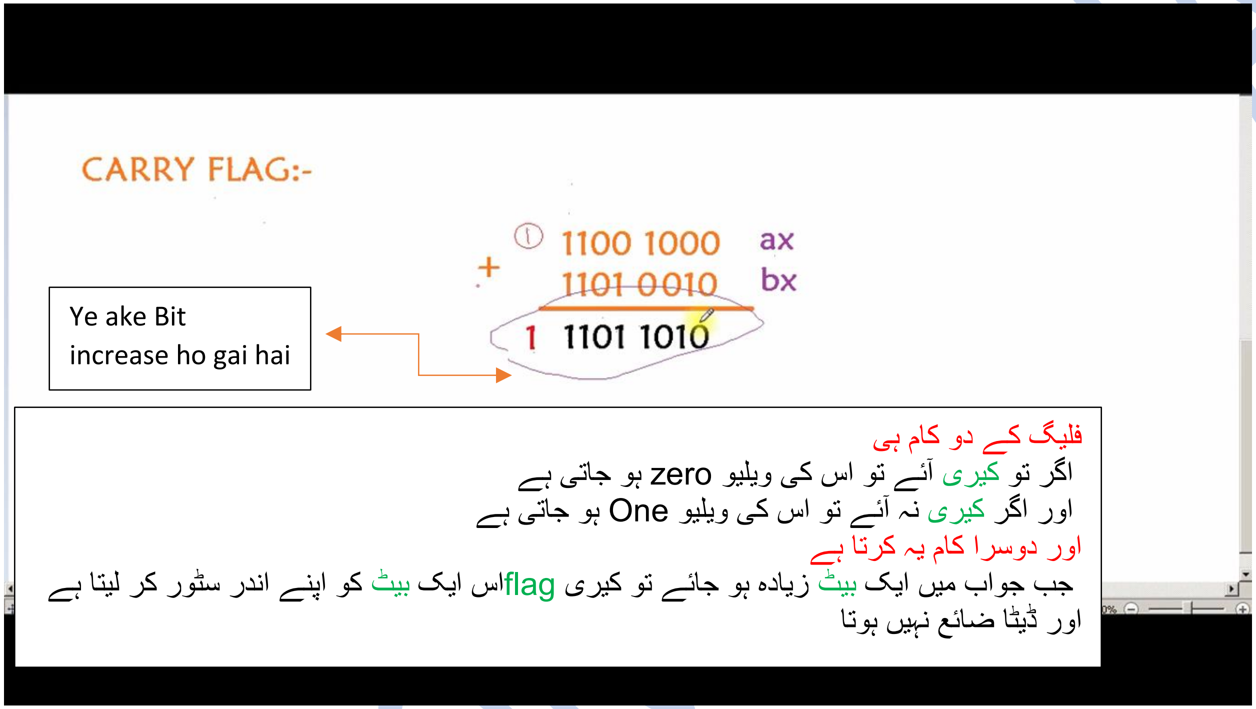
اس میں نیلے رنگ والی خالی بٹس ہیں

Overall flag register represents state of microprocessor.

It contains bits, and every bit has its own meaning.

POSITIVE=0

NEGATIVE=1



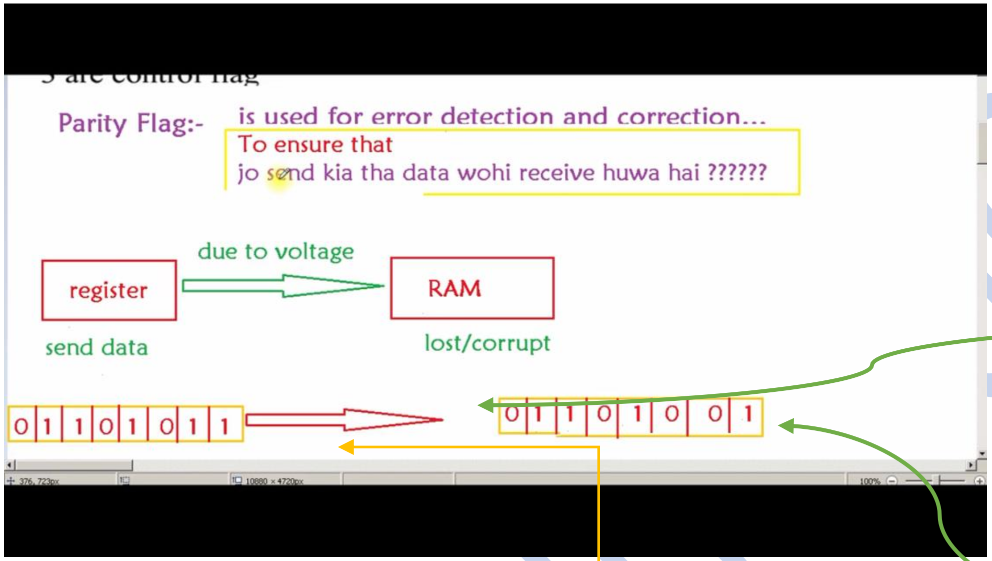
CARRY FLAG:-

```
   ①  1100 1000   ax
 +    1101 0010   bx
   ─────────────
   1  1101 1010
```

Ye ake Bit increase ho gai hai

فلیگ کے دو کام ہی
اگر تو کیری آئے تو اس کی ویلیو zero ہو جاتی ہے
اور اگر کیری نہ آئے تو اس کی ویلیو One ہو جاتی ہے
اور دوسرا کام یہ کرتا ہے
جب جواب میں ایک بیٹ زیادہ ہو جائے تو کیری flagاس ایک بیٹ کو اپنے اندر سٹور کر لیتا ہے
اور ڈیٹا ضائع نہیں ہوتا

Parity Flag:

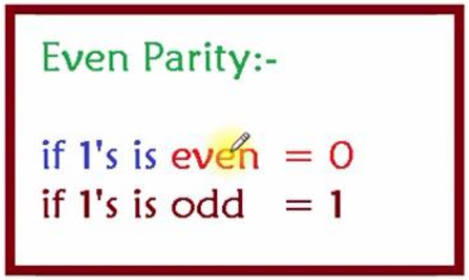


Parity Flag:- is used for error detection and correction...

To ensure that jo send kia tha data wohi receive huwa hai ??????

Is jaga number of 1’s, 5 hain yani odd hain to parity ‘1’ hogi

Is jaga number of 1’s, 4 hainyani even hain to parity ‘0’ hogi



Even Parity:-

if 1's is even = 0

if 1's is odd = 1

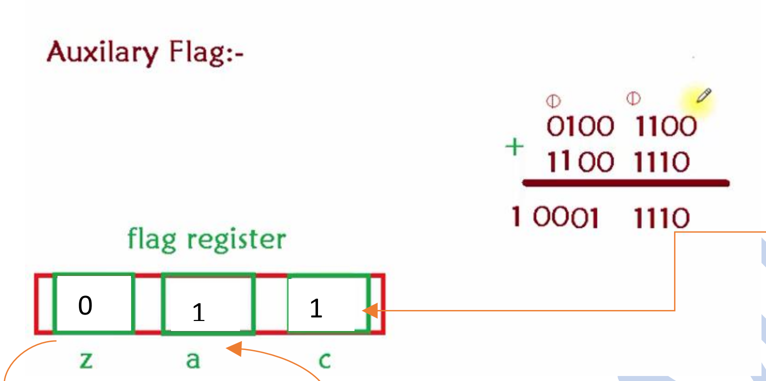
Donoki parity value same nahihai ,issiliye message send hoga k data galataya hi dobara send karo.

Auxiliary Flag:

It is used very rare,

Group of 4 bits is called nibble. Eg:- 1100

Agar nibble k baad carry a jaye to auxiliary flag ki value One ho jatihai,

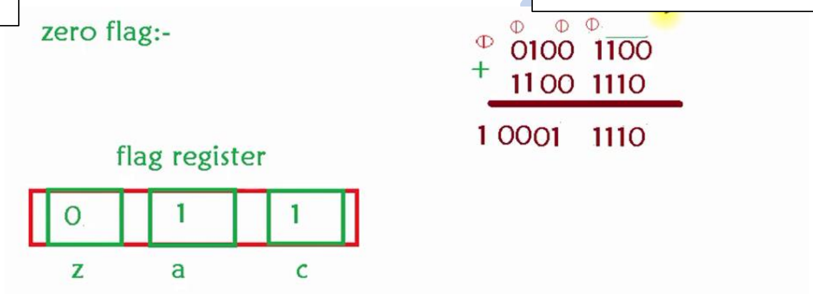


Jab last ma carry a jaye to carry flag ki value 1 ho jayegi

Zero flag ki value 0 hogi q k answer ma 0 nahiaya.

Jab nibble k baad carry a jaye to Auxilary flag ki value 1 ho jatihai.

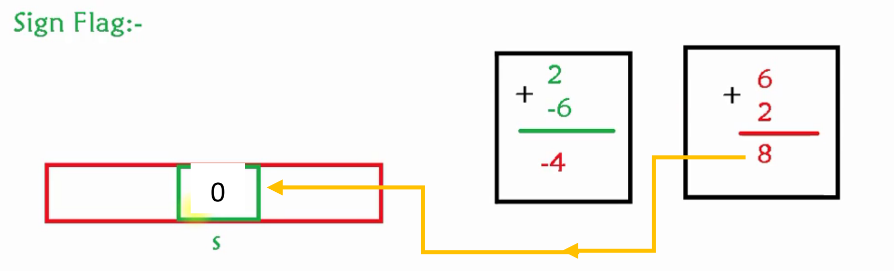
Zero Flag:



If we do any operation and it gives a value =‘0’ then zero flag will become 1.

And if it gives a value =‘1’ then zero flag will become 0.

Sign Flag:



If we do any operation and it gives positive value, then sign flag will become 0.

And if it gives negative value, then sign flag will become 1.

Trap Flag:

Trap flag is used to facilitate debuggers.

Further detail will be shown practically.

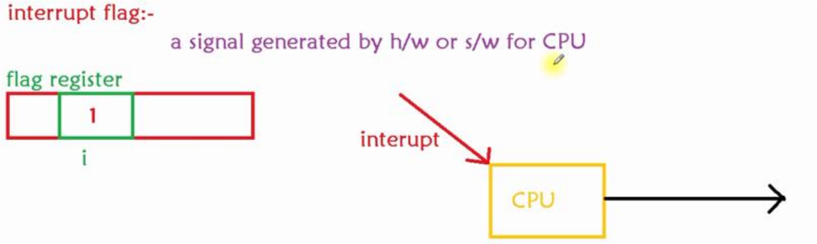
If we set trap flag at 1 it will execute all line of code one by one.

Interrupt Flag:

It is a signal which is generated by hardware or software

When any problem occurs, this signal is also generated if any software needs a service.

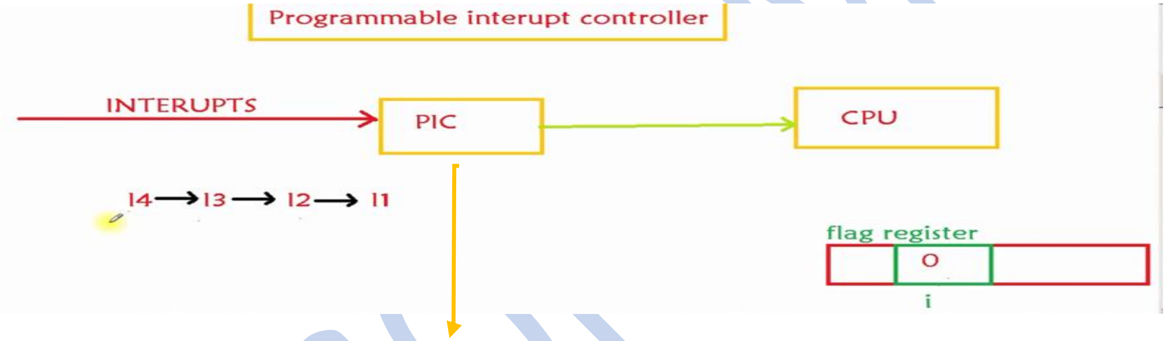
These signals are generated for CPU.



Jab koi interrupt signal ata hai to CPU uski value check kartahi ,

Agar to value 1 hai to CPU apny wala kaam chor k interrupt wala kaam karna shuru kardayga,

Or agar iski value 0 hai to apna kaam kart arahy ga interrupt signal ko ignore kardayga.



Programmable interrupt controller: kaam esay karta hai k jitney b interrupt atay hain unko receive karta hai or phir chek karta hi k flag register ki value kia hai,

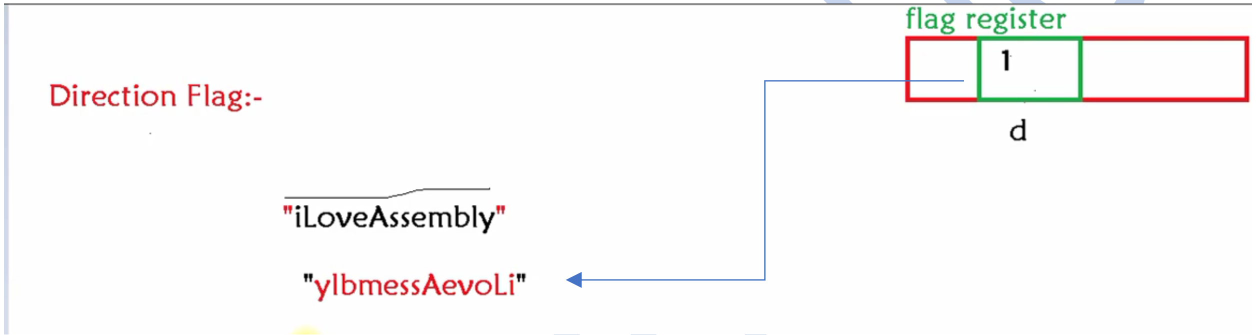
Agar to value 1 hai to iska matlab CPU free hai to interrupt signals CPU ko processing k liye bhaij diye jatay hain.

Or agar flag register ki value 0 ho iska matlab CPU abhi free nahi hai koi oor kaam kar raha hai issi liye interrupt signals rok liye jatay hain.

Programmable interrupt controller ye kaam b karta hai k jo sab say zaroori interrupt signal hota hai usay sab say pehly CPU ko Bhaijta hai.

Direction Flag:

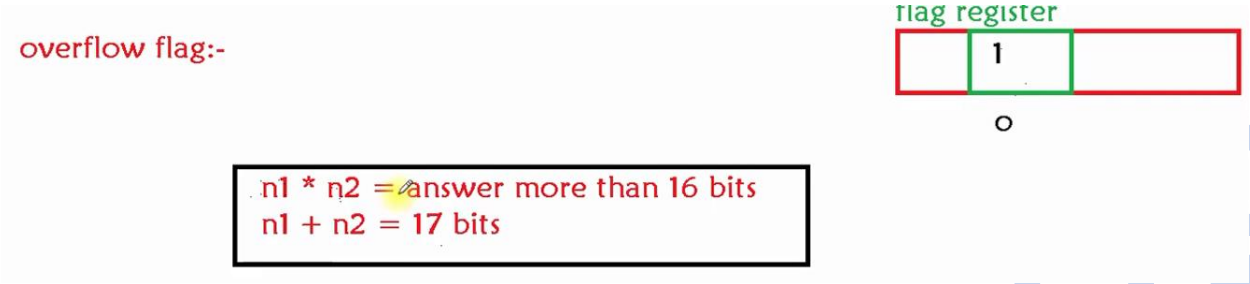
It controls string processing. left to right or right to left



If direction flag is 0 then it will process string from left to right.

And if direction flag is 1 it will process string from right to left.

Overflow Flag:

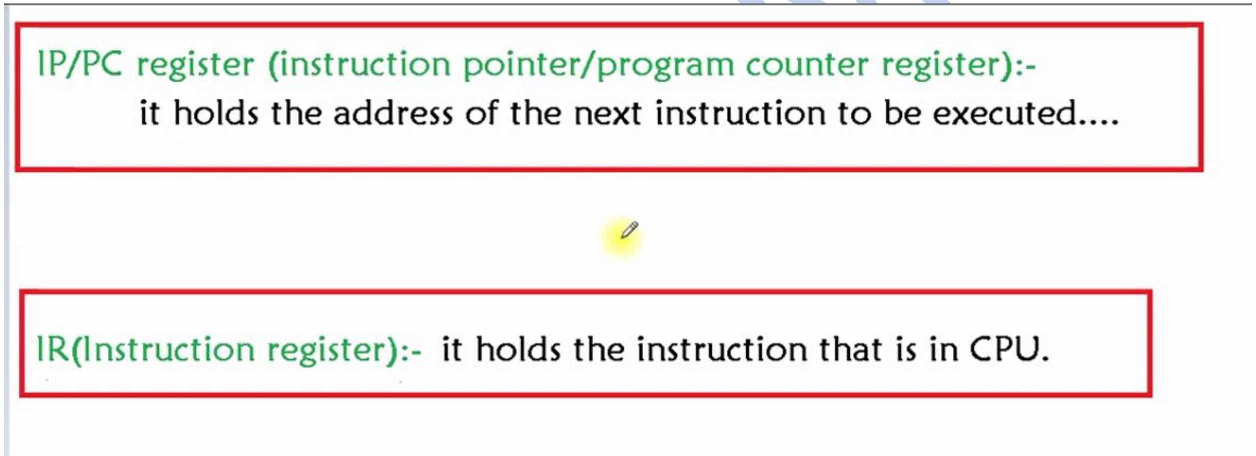


overflow flag:-

n1 * n2 = answer more than 16 bits
n1 + n2 = 17 bits

Agar ham kisi numer ko multiply ya add karen or uska answer uski saving capacity say ziada a jaye to flag register ki value 1 ho jatihai , jo show kartahai k answer over flow ho rahahai .

Agar answer saving capacity k andar andar aye to value 0 ho jatihai.



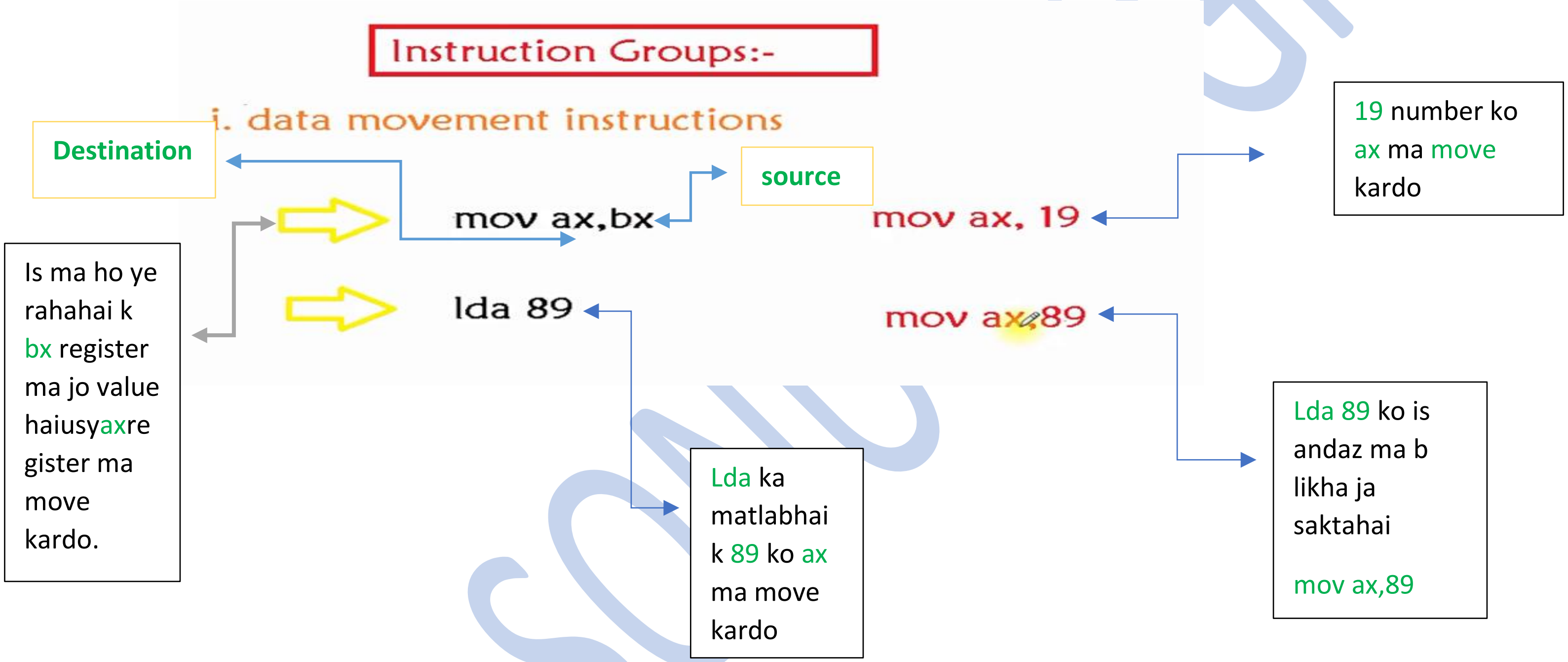
IP/PC register (instruction pointer/program counter register):-
it holds the address of the next instruction to be executed....

IR(Instruction register):- it holds the instruction that is in CPU.

Computer ma virus is tarha ata hai, k jab ham koi USB computer ko lagaty hain to virus chupkay say IP k andar apni virus wali file ka address rakh deta hai ,

Computer ka kaam to just address file ko execute karna hota hai to wo virus wali file b execute ho jati hai jis say computer ma virus a jata hai.

Instructions Groups:



Instruction Groups:-

i. data movement instructions

mov ax,bx

mov ax, 19

19 number ko ax ma move kardo

Is ma ho ye rahahai k bx register ma jo value haiusyaxregister ma move kardo.

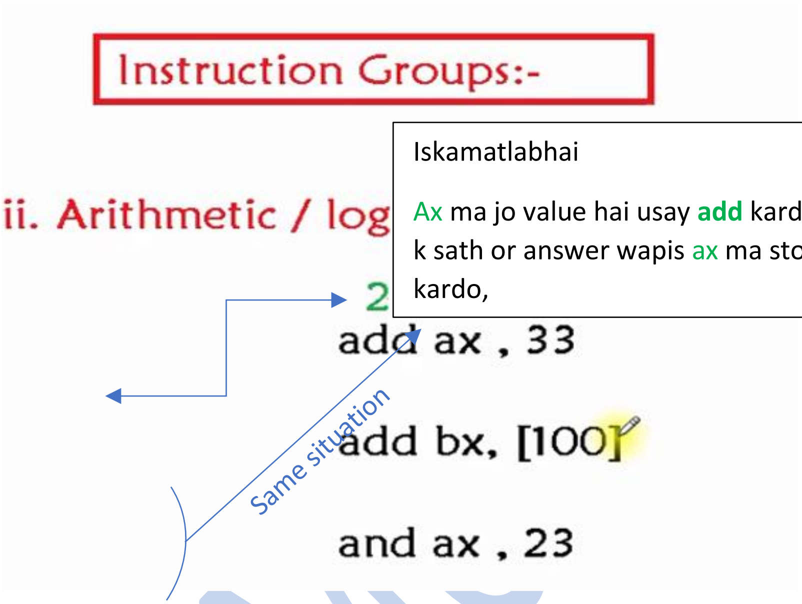
lda 89

mov ax,89

Lda ka matlabhai k 89 ko ax ma move kardo

Lda 89 ko is andaz ma b likha ja saktahai

mov ax,89



Instruction Groups:-

ii. Arithmetic / log

2

add ax , 33

add bx, [100]

and ax , 23

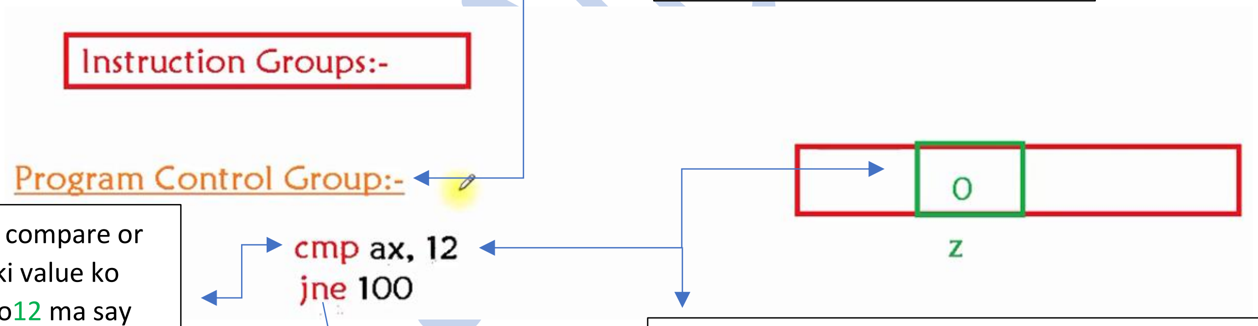
Iskamatlabhai

Ax ma jo value hai usay **add** kardo 33 k sath or answer wapis ax ma store kardo,

Same situation

Is ma tarteeb ka khyal rakhna hai

Those instructions which control program execution



Instruction Groups:-

Program Control Group:-

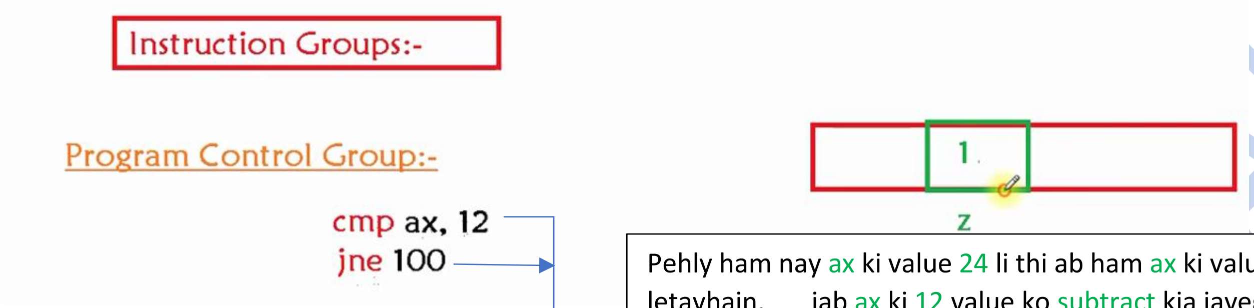
cmp ax, 12

jne 100

Cmp matlab compare or subtract, axki value ko subtract karo12 ma say or registers ko updatekaro.

IskamatlabhaiJump Not Equal, mtlab agar zero flag ki value 0haii to 100 address pay jumpkarjao

Eg:-axki value 24 thi, 24 ko 12 say subtract kia to 12 answer aya jo k zeronahihai, issiliye zero flag ki value 0 ho jayegi.



Instruction Groups:-

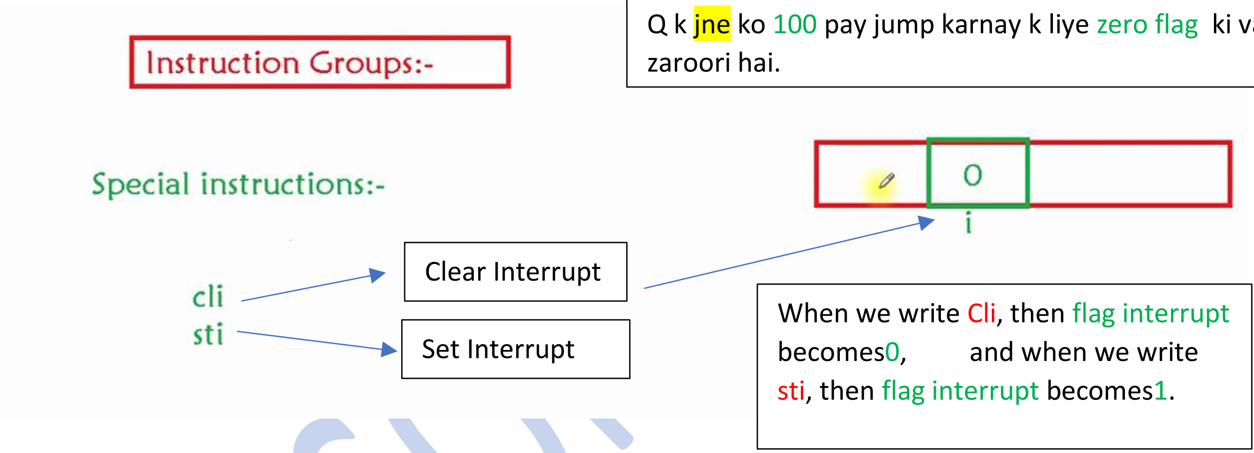
Program Control Group:-

cmp ax, 12
jne 100

Pehly ham nay ax ki value 24 li thi ab ham ax ki value 12 letayhain, jab ax ki 12 value ko subtract kia jayega bahirwali 12 value k sath to answer 0 aye ga.

Jab answer 0 aye ga to zero flag ki value 1 ho jayegi or zero flag ki 1 value pay jne100 pay jump nah ikaray ga ,

Q k jne ko 100 pay jump karnay k liye zero flag ki value 0 hona zaroori hai.



Instruction Groups:-

Special instructions:-

cli → Clear Interrupt
sti → Set Interrupt

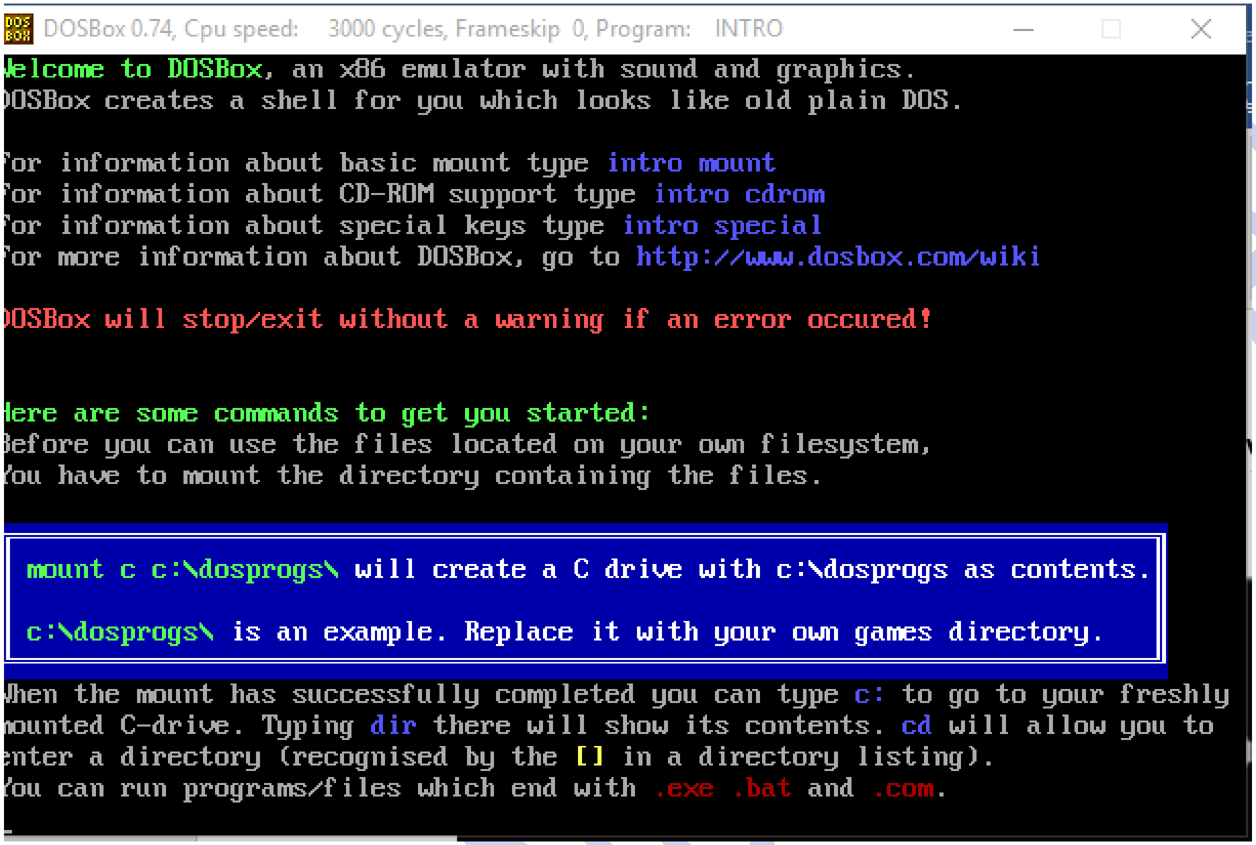
When we write Cli, then flag interrupt becomes0, and when we write sti, then flag interrupt becomes1.

These special instruction changes the CPU behavior.

Lecture 6

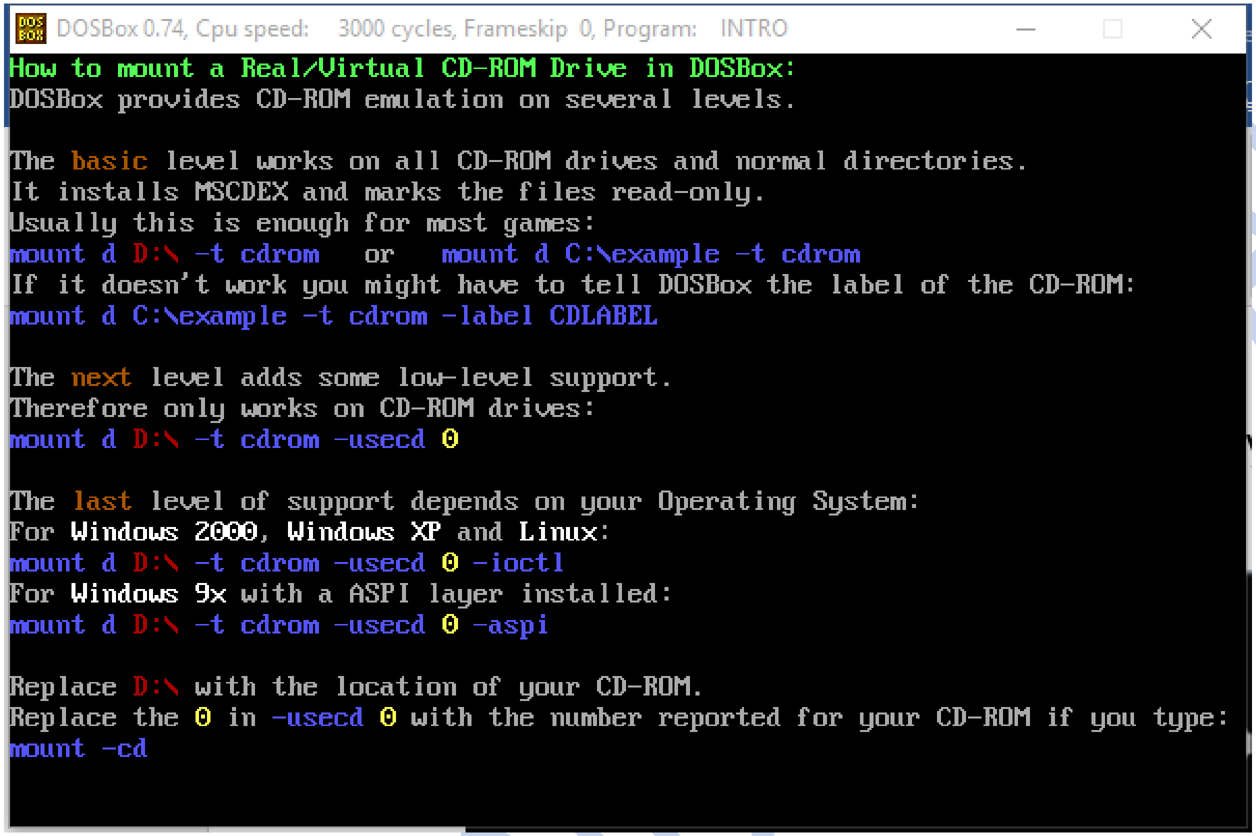
Program execution

Some informative s.s are following

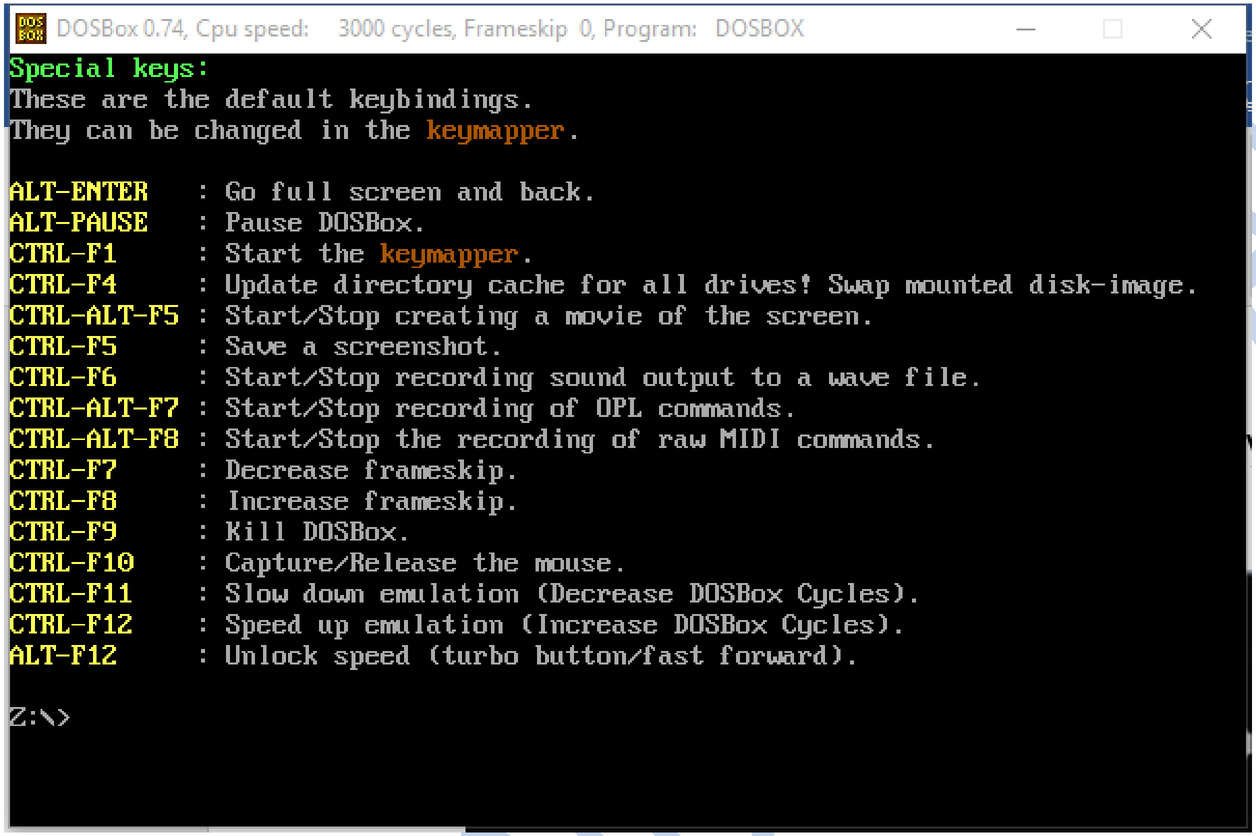


DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: INTRO

```
Welcome to DOSBox, an x86 emulator with sound and graphics.
DOSBox creates a shell for you which looks like old plain DOS.

For information about basic mount type intro mount
For information about CD-ROM support type intro cdrom
For information about special keys type intro special
For more information about DOSBox, go to http://www.dosbox.com/wiki

DOSBox will stop/exit without a warning if an error occured!


Here are some commands to get you started:
Before you can use the files located on your own filesystem,
You have to mount the directory containing the files.

 mount c c:\dosprogs\ will create a C drive with c:\dosprogs as contents.

 c:\dosprogs\ is an example. Replace it with your own games directory.

When the mount has successfully completed you can type c: to go to your freshly
mounted C-drive. Typing dir there will show its contents. cd will allow you to
enter a directory (recognised by the [] in a directory listing).
You can run programs/files which end with .exe .bat and .com.
```

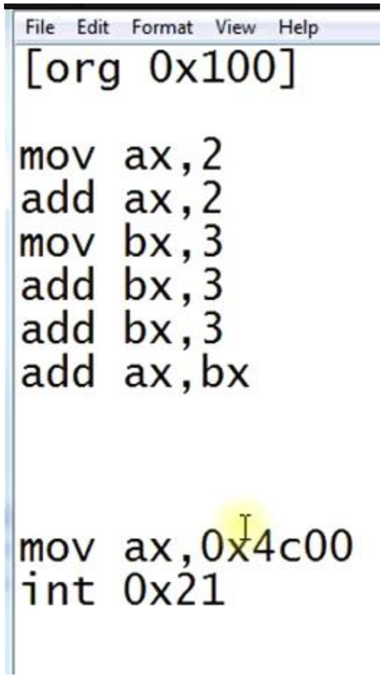

DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: INTRO

```
How to mount a Real/Virtual CD-ROM Drive in DOSBox:
DOSBox provides CD-ROM emulation on several levels.

The basic level works on all CD-ROM drives and normal directories.
It installs MSCDEX and marks the files read-only.
Usually this is enough for most games:
mount d D:\ -t cdrom   or   mount d C:\example -t cdrom
If it doesn't work you might have to tell DOSBox the label of the CD-ROM:
mount d C:\example -t cdrom -label CDLABEL

The next level adds some low-level support.
Therefore only works on CD-ROM drives:
mount d D:\ -t cdrom -usecd 0

The last level of support depends on your Operating System:
For Windows 2000, Windows XP and Linux:
mount d D:\ -t cdrom -usecd 0 -ioctl
For Windows 9x with a ASPI layer installed:
mount d D:\ -t cdrom -usecd 0 -aspi

Replace D:\ with the location of your CD-ROM.
Replace the 0 in -usecd 0 with the number reported for your CD-ROM if you type:
mount -cd
```


DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

```
Special keys:
These are the default keybindings.
They can be changed in the keymapper.

ALT-ENTER     : Go full screen and back.
ALT-PAUSE     : Pause DOSBox.
CTRL-F1       : Start the keymapper.
CTRL-F4       : Update directory cache for all drives! Swap mounted disk-image.
CTRL-ALT-F5   : Start/Stop creating a movie of the screen.
CTRL-F5       : Save a screenshot.
CTRL-F6       : Start/Stop recording sound output to a wave file.
CTRL-ALT-F7   : Start/Stop recording of OPL commands.
CTRL-ALT-F8   : Start/Stop the recording of raw MIDI commands.
CTRL-F7       : Decrease frameskip.
CTRL-F8       : Increase frameskip.
CTRL-F9       : Kill DOSBox.
CTRL-F10      : Capture/Release the mouse.
CTRL-F11      : Slow down emulation (Decrease DOSBox Cycles).
CTRL-F12      : Speed up emulation (Increase DOSBox Cycles).
ALT-F12       : Unlock speed (turbo button/fast forward).

Z:\>
```

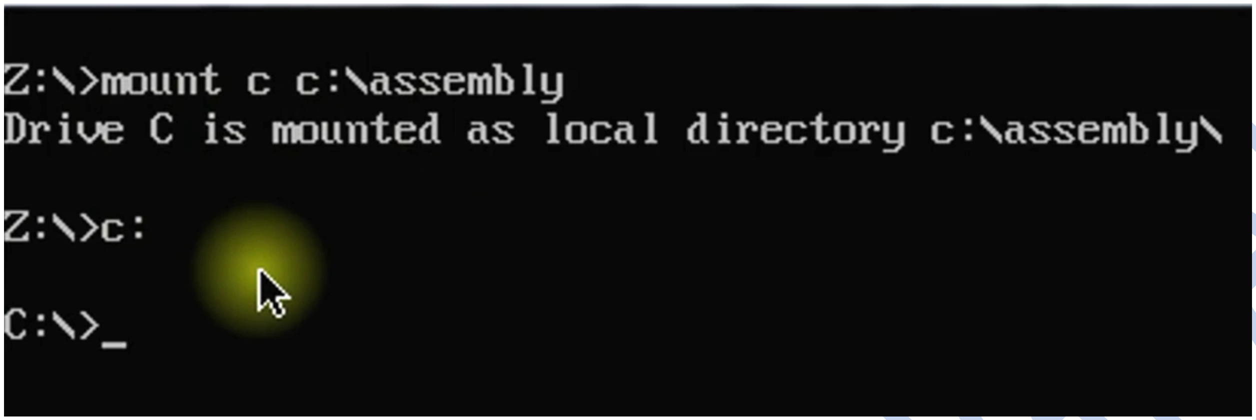
Our code in .txt file



```
[org 0x100]

mov ax,2
add ax,2
mov bx,3
add bx,3
add bx,3
add ax,bx



mov ax,0x4c00
int 0x21
```

To convert our assembly code in binary language we will write following code in DoxBox

First type cls to clear the screen.



```
Z:\>mount c c:\assembly
Drive C is mounted as local directory c:\assembly\

Z:\>c:

C:\>_
```

Type dir to show all files in assembly folder.

Now convert assembly code file to binary code file using following code:

nasm first.asm -o fir.com

When code has been converted we will write following code to execute our first.asm file. Type:

first.com

how to generate listing file

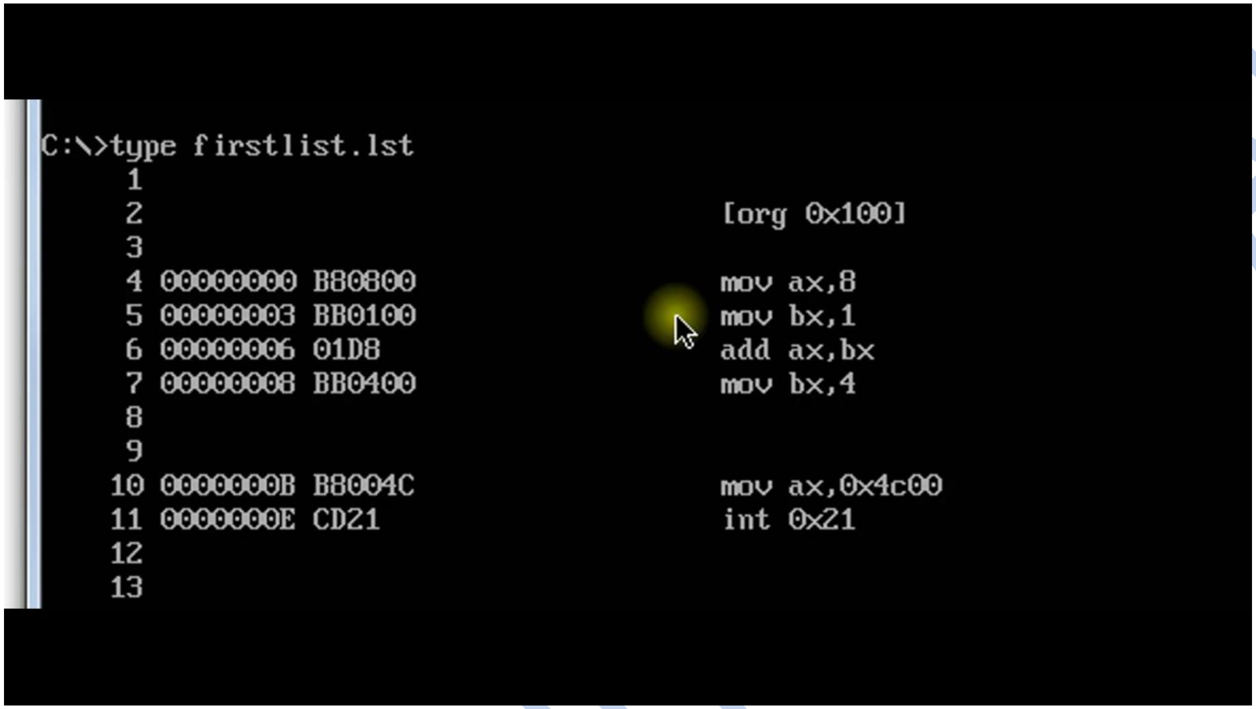
nasm first.asm -l firstlist.lst

now list has been generated, and we are going to execute this list file.

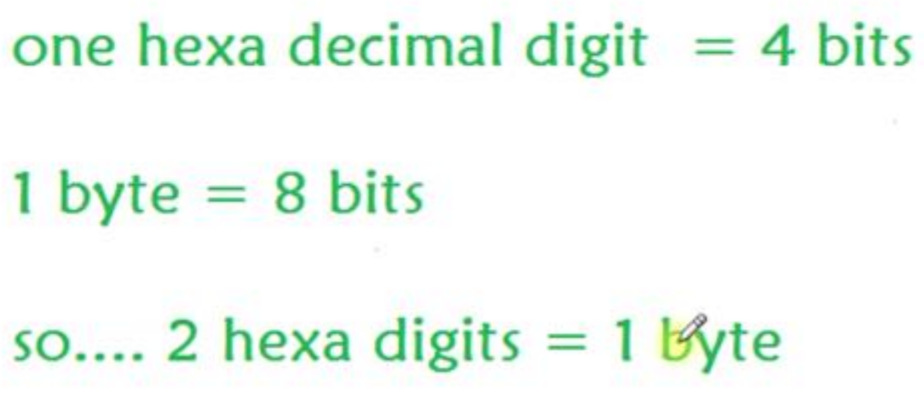
type firstllist.lst

How to Read list file

Take lecture



```
C:\>type firstlist.lst
     1
     2                                      [org 0x100]
     3
     4 00000000 B80800                      mov ax,8
     5 00000003 BB0100                      mov bx,1
     6 00000006 01D8                        add ax,bx
     7 00000008 BB0400                      mov bx,4
     8
     9
    10 0000000B B8004C                      mov ax,0x4c00
    11 0000000E CD21                        int 0x21
    12
    13
```



one hexa decimal digit = 4 bits

1 byte = 8 bits

so.... 2 hexa digits = 1 byte

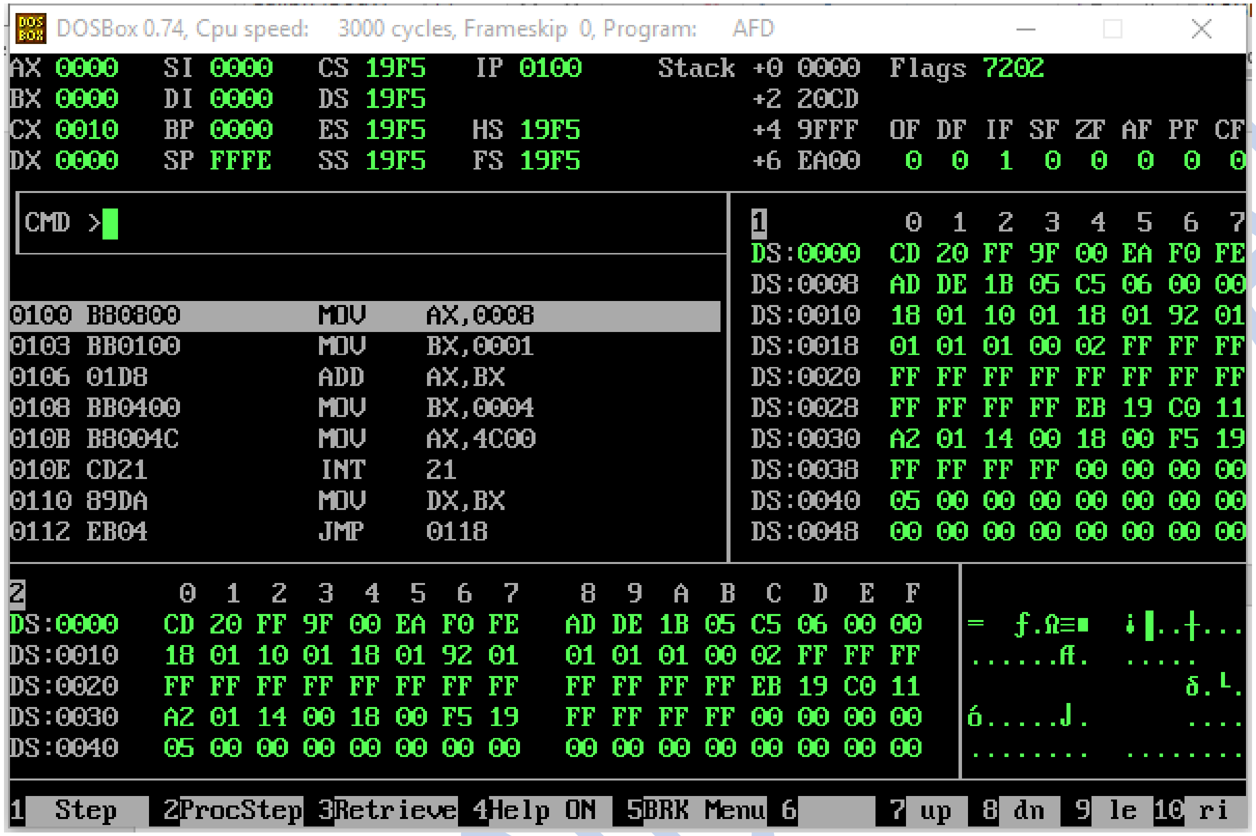
Using Debugger:

In assembly language we use afd(advance full screen debugger)

Debugger only supports .com file

Now type following code to execute your program in debugger format.

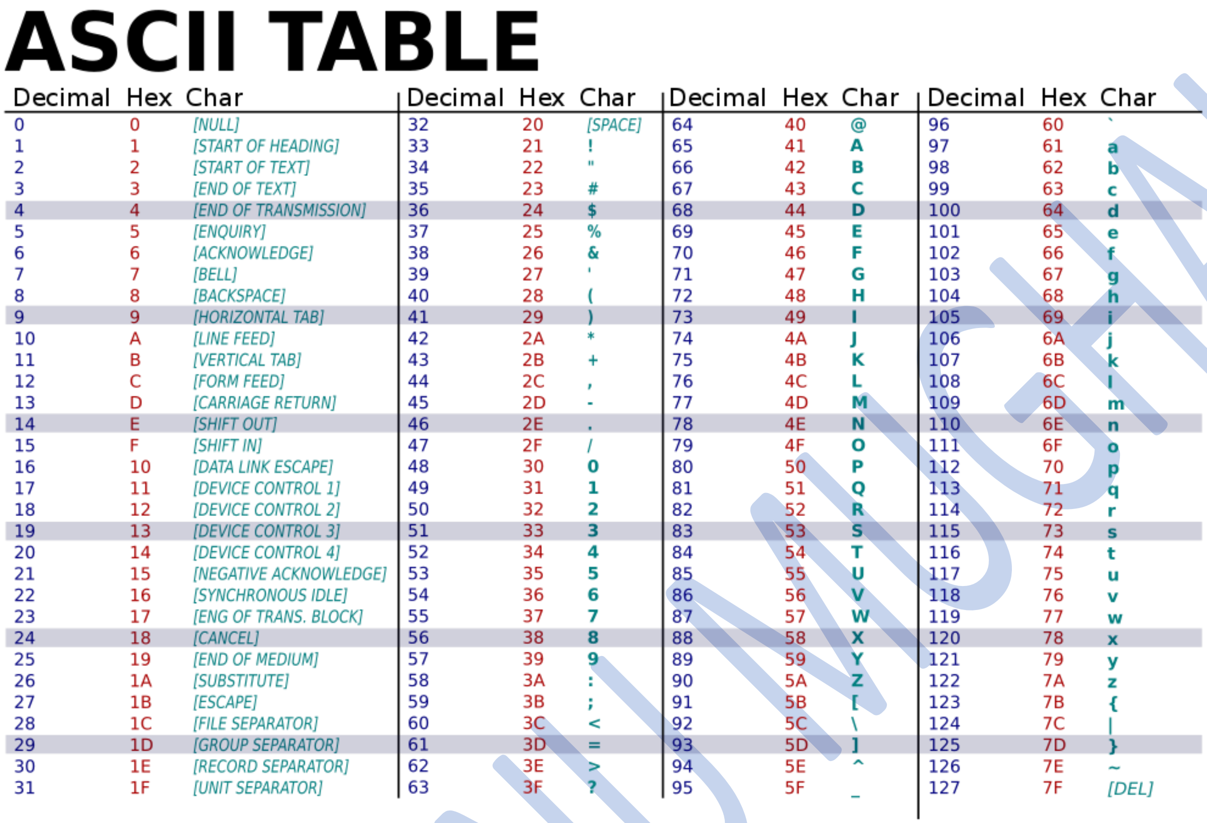
afd fir.com



DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: AFD

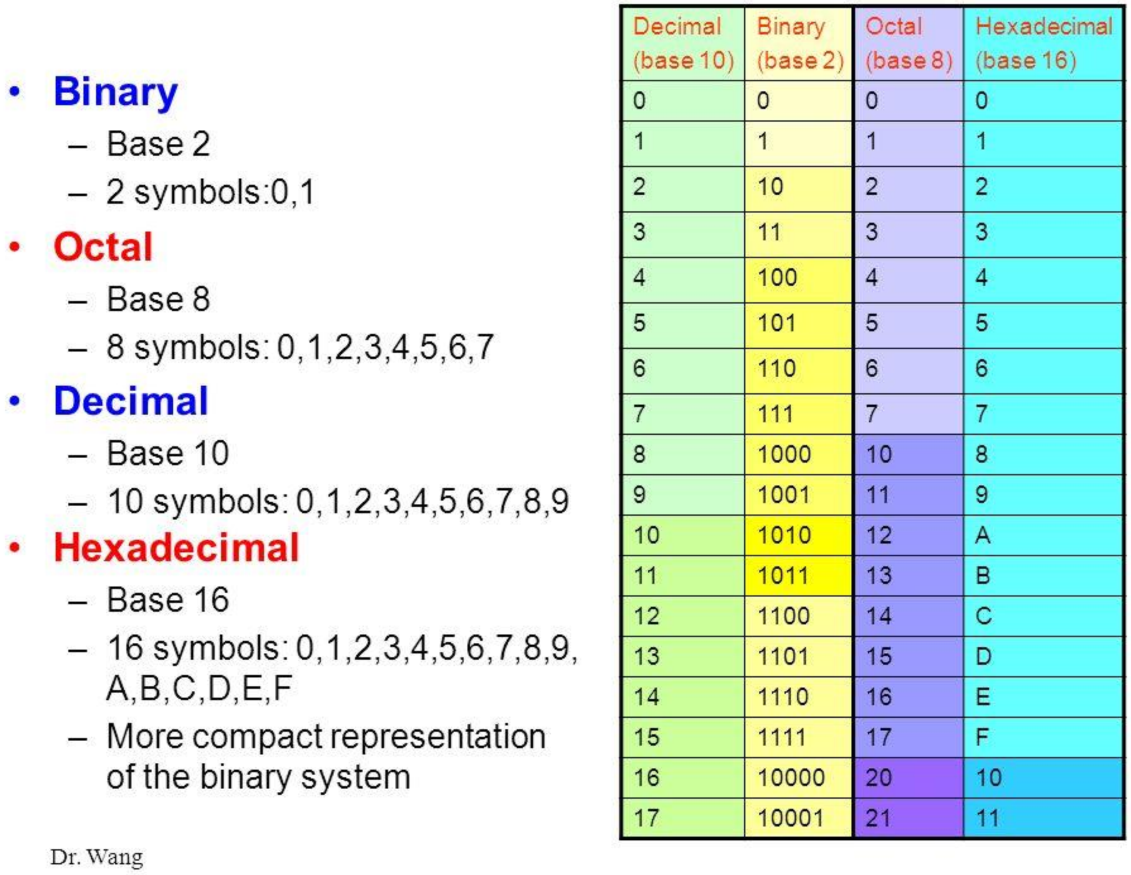
```
AX 0000   SI 0000   CS 19F5   IP 0100      Stack +0 0000  Flags 7202
BX 0000   DI 0000   DS 19F5                      +2 20CD
CX 0010   BP 0000   ES 19F5   HS 19F5            +4 9FFF  OF DF IF SF ZF AF PF CF
DX 0000   SP FFFE   SS 19F5   FS 19F5            +6 EA00   0  0  1  0  0  0  0  0

 CMD >                                      1          0  1  2  3  4  5  6  7
                                            DS:0000    CD 20 FF 9F 00 EA F0 FE
                                            DS:0008    AD DE 1B 05 C5 06 00 00
0100 B80800         MOV    AX,0008          DS:0010    18 01 10 01 18 01 92 01
0103 BB0100         MOV    BX,0001          DS:0018    01 01 01 00 02 FF FF FF
0106 01D8           ADD    AX,BX            DS:0020    FF FF FF FF FF FF FF FF
0108 BB0400         MOV    BX,0004          DS:0028    FF FF FF FF EB 19 C0 11
010B B8004C         MOV    AX,4C00          DS:0030    A2 01 14 00 18 00 F5 19
010E CD21           INT    21               DS:0038    FF FF FF FF 00 00 00 00
0110 89DA           MOV    DX,BX            DS:0040    05 00 00 00 00 00 00 00
0112 EB04           JMP    0118             DS:0048    00 00 00 00 00 00 00 00

2          0  1  2  3  4  5  6  7    8  9  A  B  C  D  E  F
DS:0000    CD 20 FF 9F 00 EA F0 FE   AD DE 1B 05 C5 06 00 00   =  ƒ.Ω≡■   ı▌..┼...
DS:0010    18 01 10 01 18 01 92 01   01 01 01 00 02 FF FF FF   ......Æ.   .....
DS:0020    FF FF FF FF FF FF FF FF   FF FF FF FF EB 19 C0 11                δ.└.
DS:0030    A2 01 14 00 18 00 F5 19   FF FF FF FF 00 00 00 00   ó.....J.       ....
DS:0040    05 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ........   ........

1 Step  2ProcStep 3Retrieve 4Help ON  5BRK Menu 6      7 up  8 dn  9 le 10 ri
```

F1 and F2 these keys are used to execute program one by one in debugger.



ASCII TABLE

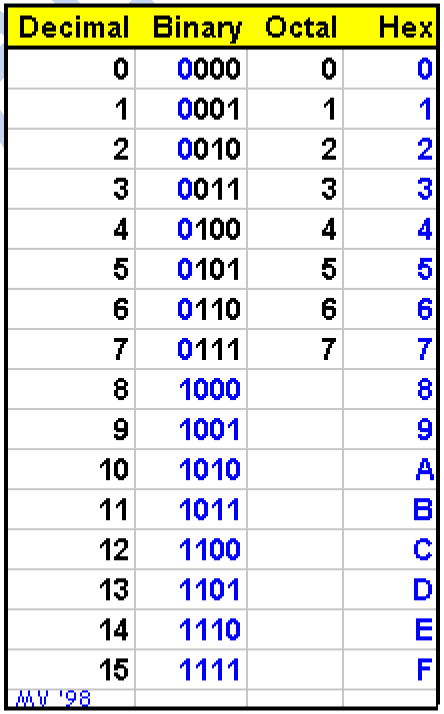
Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]



- **Binary**
  - Base 2
  - 2 symbols:0,1
- **Octal**
  - Base 8
  - 8 symbols: 0,1,2,3,4,5,6,7
- **Decimal**
  - Base 10
  - 10 symbols: 0,1,2,3,4,5,6,7,8,9
- **Hexadecimal**
  - Base 16
  - 16 symbols: 0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F
  - More compact representation of the binary system

Decimal (base 10)	Binary (base 2)	Octal (base 8)	Hexadecimal (base 16)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11

Dr. Wang



Decimal	Binary	Octal	Hex
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000		8
9	1001		9
10	1010		A
11	1011		B
12	1100		C
13	1101		D
14	1110		E
15	1111		F

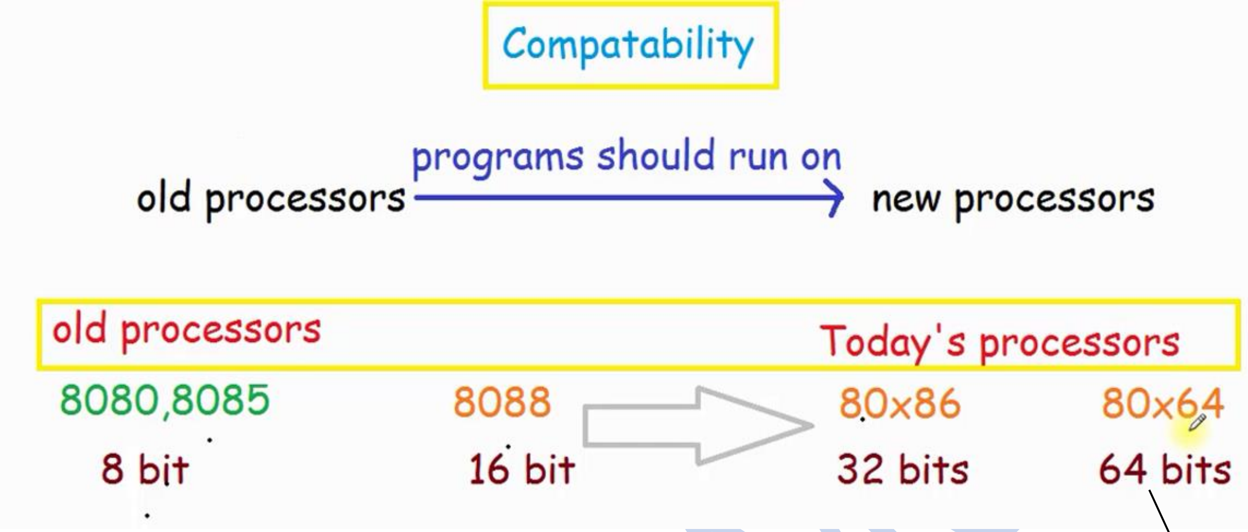
MV '98

Lecture 6

Assembly ka matlab hai assembly say number ya upcode ma lay k jana

Diss assembly ka matlab hai number yani upcode say assembly ma lay jana

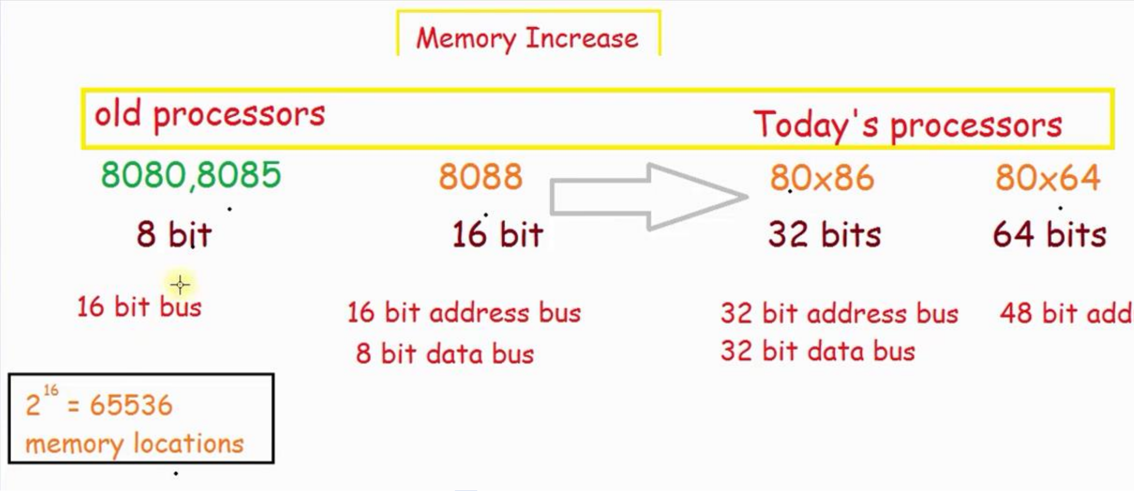
Lecture 9



Compatability

old processors $\xrightarrow{\text{programs should run on}}$ new processors

old processors		Today's processors	
8080,8085	8088	80x86	80x64
8 bit	16 bit	32 bits	64 bits



Memory Increase

old processors		Today's processors	
8080,8085	8088	80x86	80x64
8 bit	16 bit	32 bits	64 bits
16 bit bus	16 bit address bus 8 bit data bus	32 bit address bus 32 bit data bus	48 bit address bus

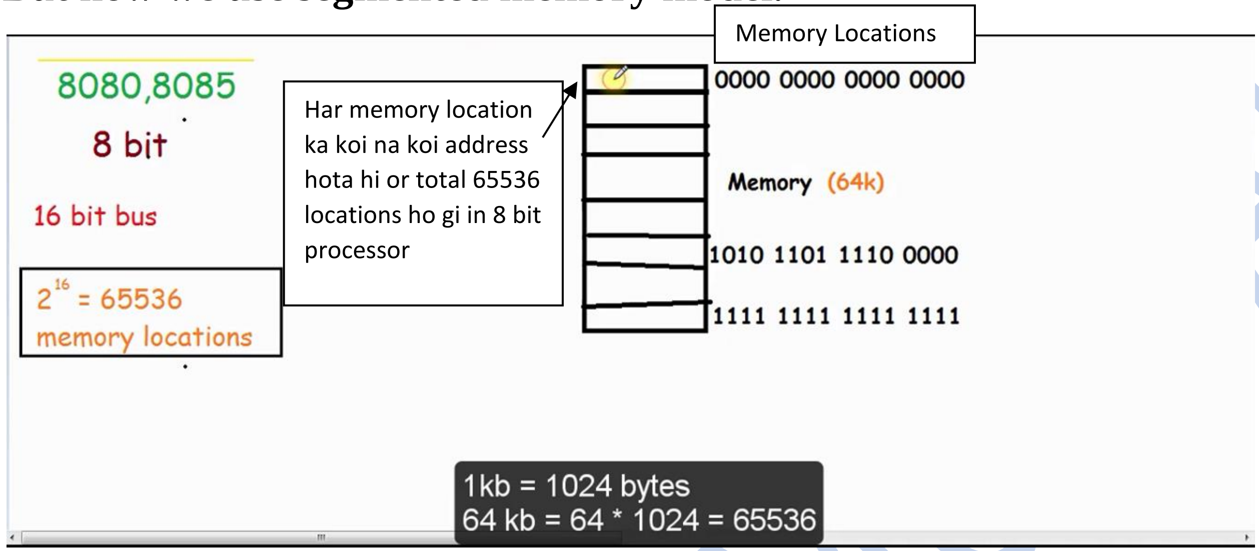
2^{16} = 65536 memory locations

Iska matlab yai ye processor ake time ma 64 bit say ziada data store nai kar saktay.

Vise versa

In old processors linear memory model was used

But now we use segmented memory model.



Memory Locations

8080,8085

8 bit

16 bit bus

2^{16} = 65536 memory locations

Har memory location ka koi na koi address hota hi or total 65536 locations ho gi in 8 bit processor

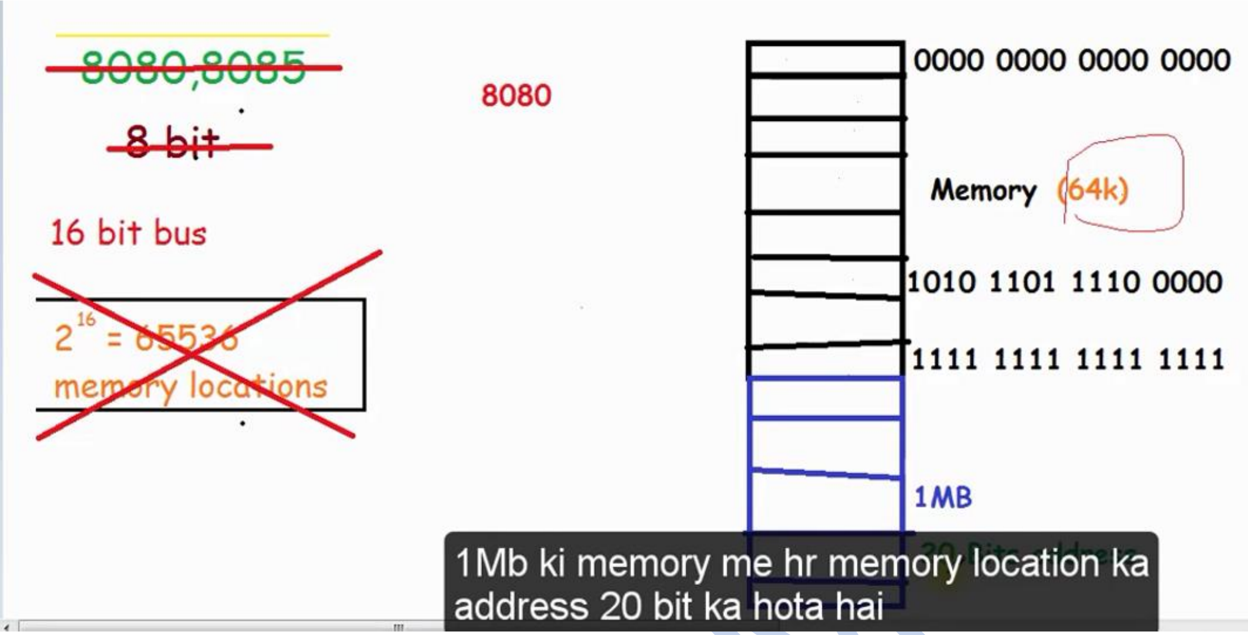
0000 0000 0000 0000

Memory (64k)

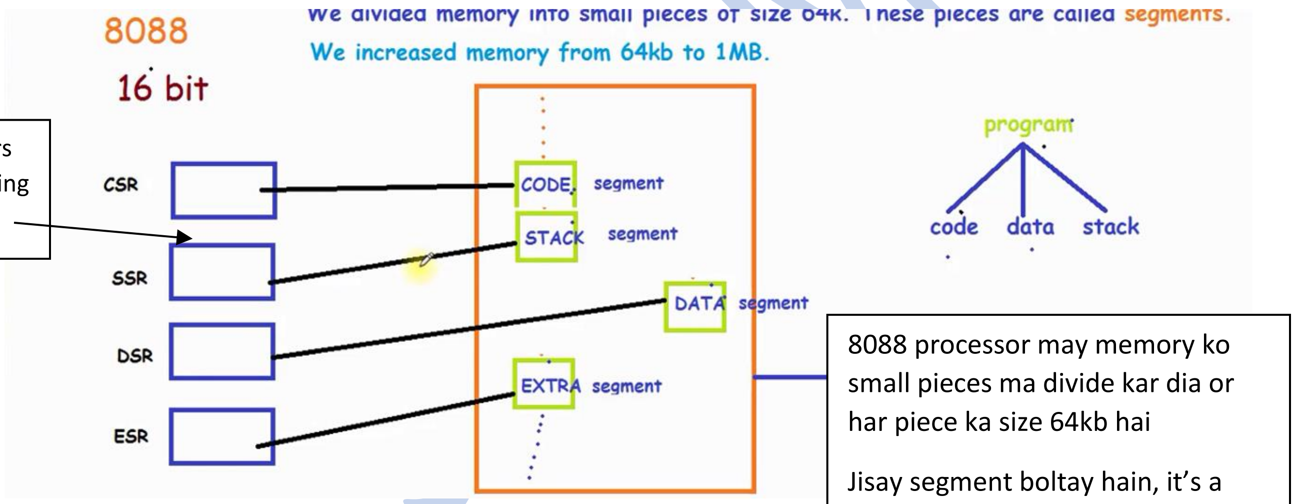
1010 1101 1110 0000

1111 1111 1111 1111

1kb = 1024 bytes
64 kb = 64 * 1024 = 65536



1Mb ki memory me hr memory location ka address 20 bit ka hota hai

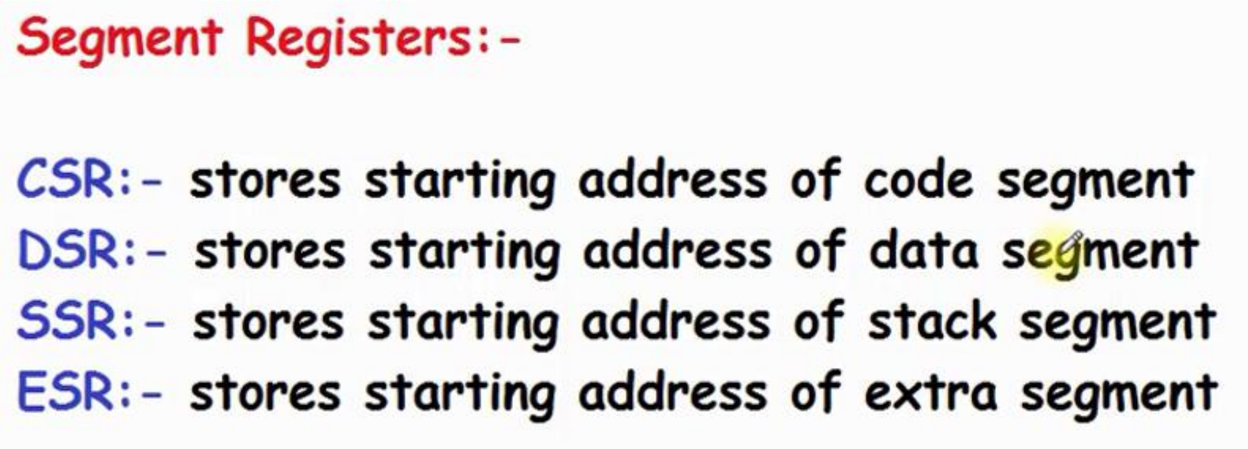

We divided memory into small pieces of size 64k. These pieces are called segments.

We increased memory from 64kb to 1MB.

These registers contains starting addresses

8088 processor may memory ko small pieces ma divide kar dia or har piece ka size 64kb hai

Jisay segment boltay hain, it’s a small piece of memory with size of 64kb.



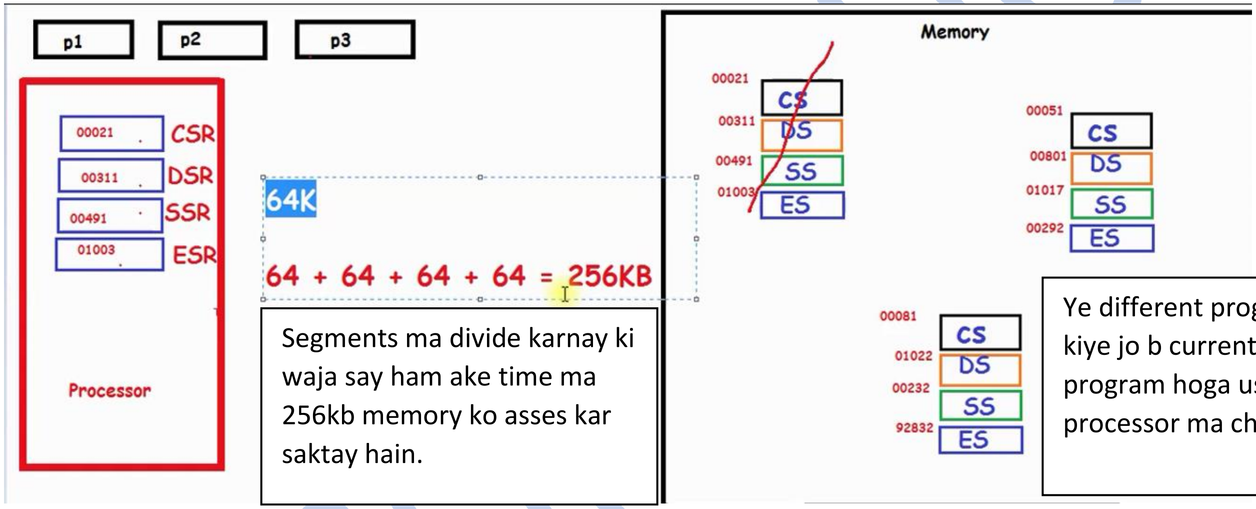
Segment Registers:-

CSR:- stores starting address of code segment

DSR:- stores starting address of data segment

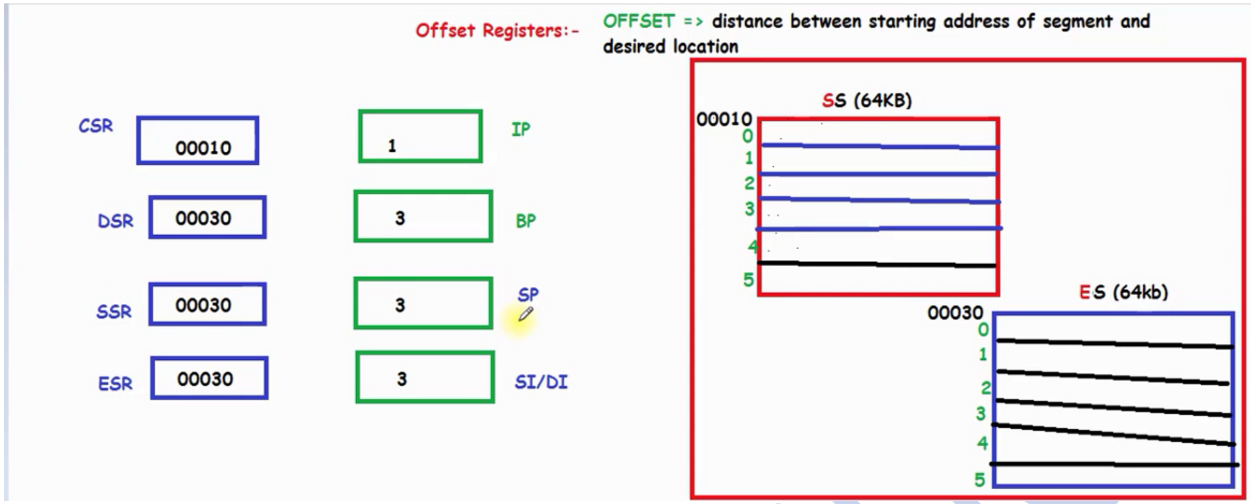
SSR:- stores starting address of stack segment

ESR:- stores starting address of extra segment



Segments ma divide karnay ki waja say ham ake time ma 256kb memory ko asses kar saktay hain.

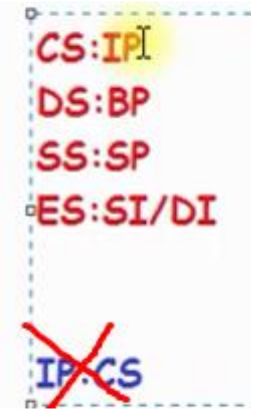
Ye different programs run kiye jo b current running program hoga uska address processor ma chala jayega



Offset Registers:-

OFFSET => distance between starting address of segment and desired location

Stating address say jis address pay ham nay jana hota hai in dono starting or destination address k difference ko offset boltay hain.



CS:IP

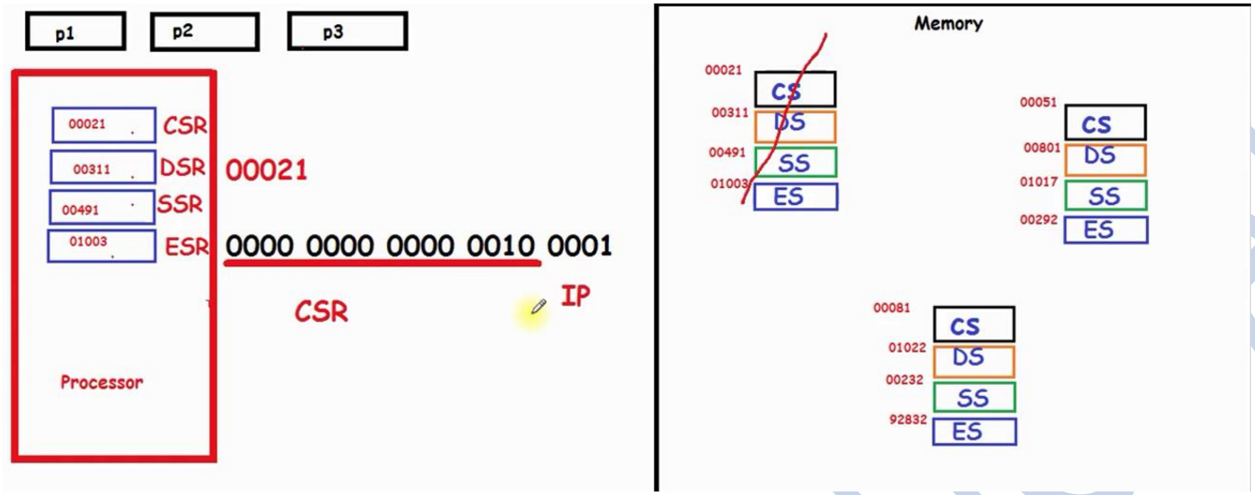
DS:BP

SS:SP

ES:SI/DI

~~IP:CS~~

in segments k sath fixed registers hotay hain

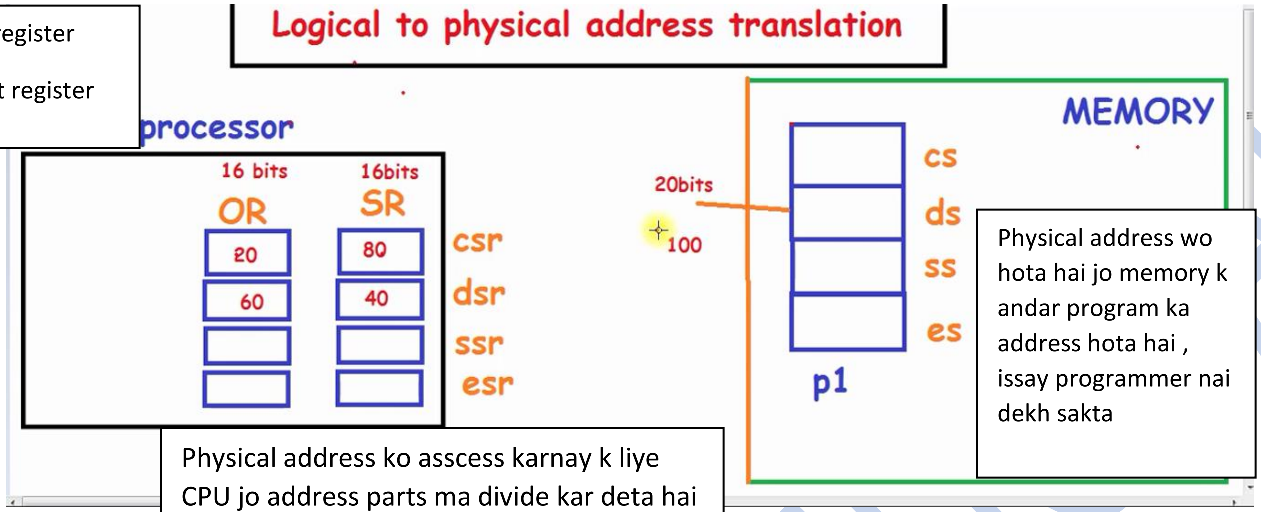


Hexa decimal ko binary ma Karen to 20bit banti hain jabky hamara system 16bit hai to remaining 4bits offset register ma store kar detay hain .

Lecture #10

OR= offset register

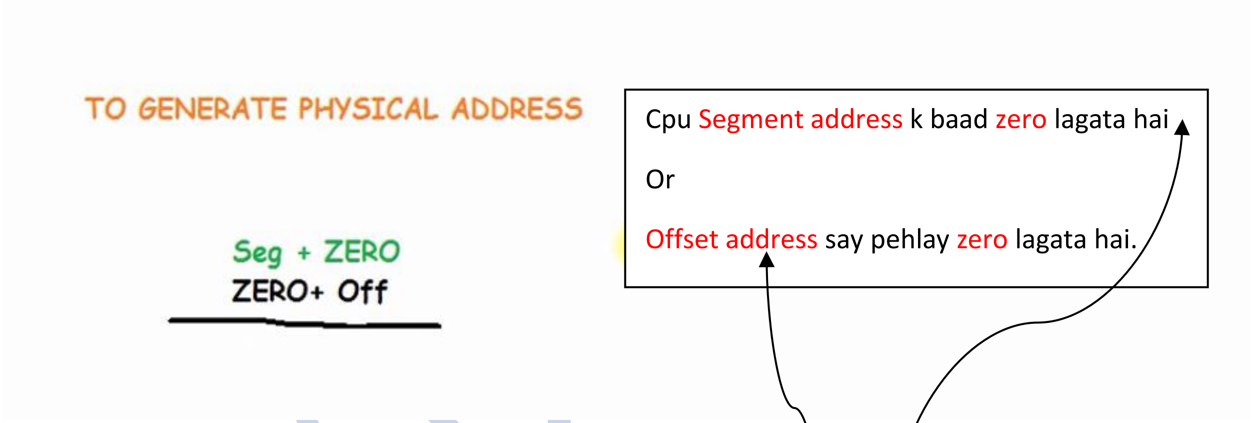
SR=segment register



Logical to physical address translation

Physical address wo hota hai jo memory k andar program ka address hota hai , issay programmer nai dekh sakta

Physical address ko asscess karnay k liye CPU jo address parts ma divide kar deta hai wo logical address kehlata hi or programmer isay dekh sakta hai.



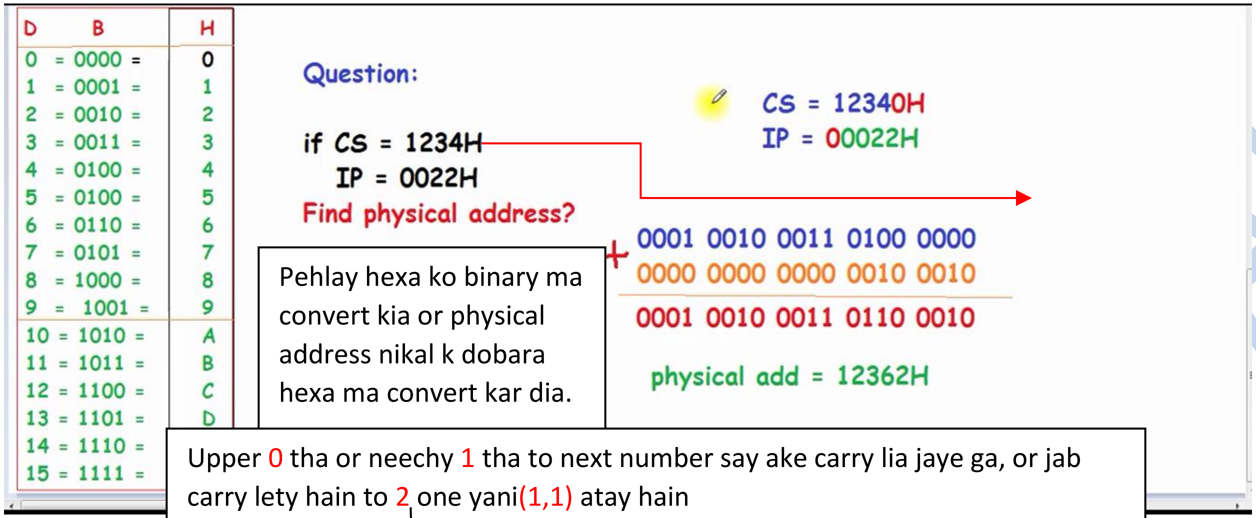
TO GENERATE PHYSICAL ADDRESS

Cpu Segment address k baad zero lagata hai

Or

Offset address say pehlay zero lagata hai.

Segment **Registers**. **Code** Segment − It contains all the instructions to be executed. A 16-bit **Code**Segment **register** or **CS register** stores the starting address of the **code** segment. Data Segment − It contains data, constants and work areas



D	B	H
0	= 0000 =	0
1	= 0001 =	1
2	= 0010 =	2
3	= 0011 =	3
4	= 0100 =	4
5	= 0100 =	5
6	= 0110 =	6
7	= 0101 =	7
8	= 1000 =	8
9	= 1001 =	9
10	= 1010 =	A
11	= 1011 =	B
12	= 1100 =	C
13	= 1101 =	D
14	= 1110 =	
15	= 1111 =	

Question:

if CS = 1234H
IP = 0022H
Find physical address?

CS = 12340H
IP = 00022H

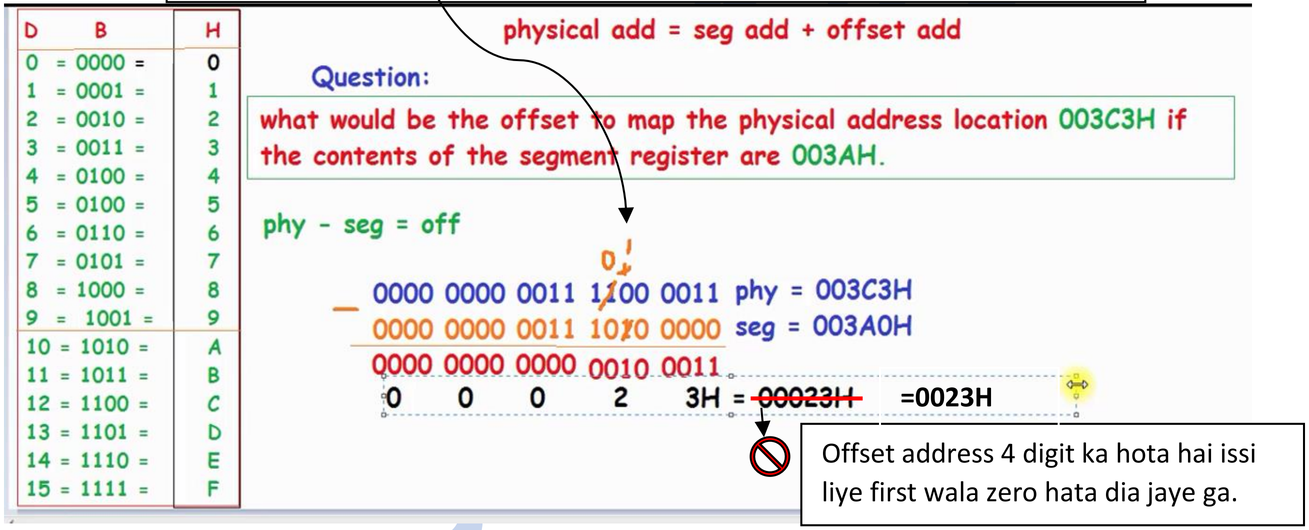
```
  0001 0010 0011 0100 0000
+ 0000 0000 0000 0010 0010
  ------------------------
  0001 0010 0011 0110 0010
```

physical add = 12362H

Pehlay hexa ko binary ma convert kia or physical address nikal k dobara hexa ma convert kar dia.

Upper 0 tha or neechy 1 tha to next number say ake carry lia jaye ga, or jab carry lety hain to 2 one yani(1,1) atay hain

No we are



D	B	H
0	= 0000 =	0
1	= 0001 =	1
2	= 0010 =	2
3	= 0011 =	3
4	= 0100 =	4
5	= 0100 =	5
6	= 0110 =	6
7	= 0101 =	7
8	= 1000 =	8
9	= 1001 =	9
10	= 1010 =	A
11	= 1011 =	B
12	= 1100 =	C
13	= 1101 =	D
14	= 1110 =	E
15	= 1111 =	F

physical add = seg add + offset add

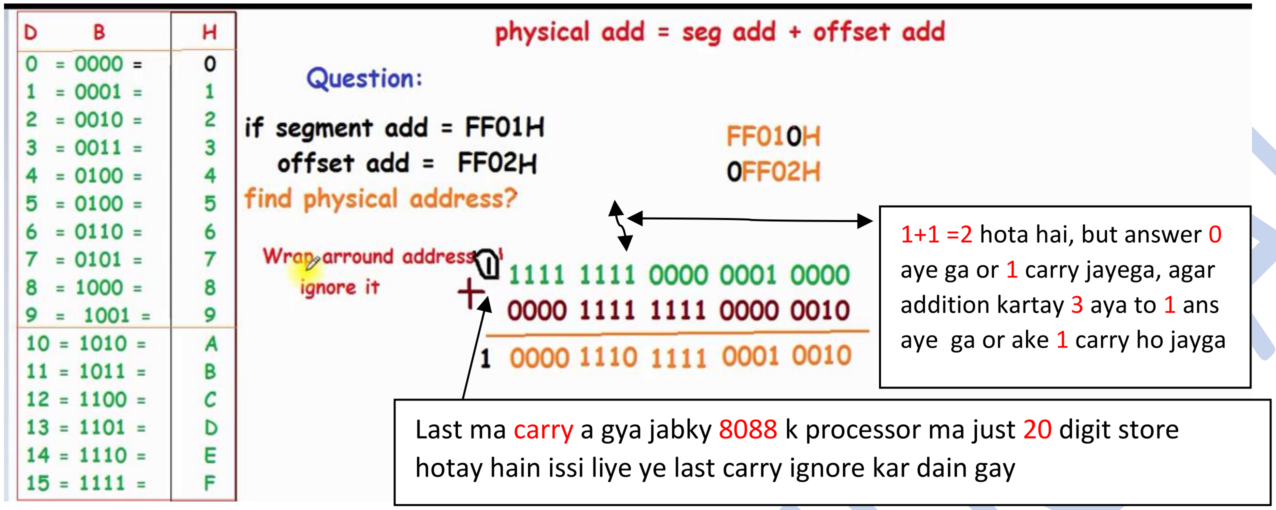
Question:

what would be the offset to map the physical address location 003C3H if the contents of the segment register are 003AH.

phy - seg = off

```
                 0 1
  0000 0000 0011 ~~1~~100 0011   phy = 003C3H
- 0000 0000 0011 10~~1~~0 0000   seg = 003A0H
  ------------------------
  0000 0000 0000 0010 0011
  0    0    0    2    3H = ~~00023H~~   =0023H
```

Offset address 4 digit ka hota hai issi liye first wala zero hata dia jaye ga.



D	B	H
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0100	5
6	0110	6
7	0101	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

physical add = seg add + offset add

Question:

if segment add = FF01H
offset add = FF02H
find physical address?

FF010H
0FF02H

Wrap arround address
ignore it

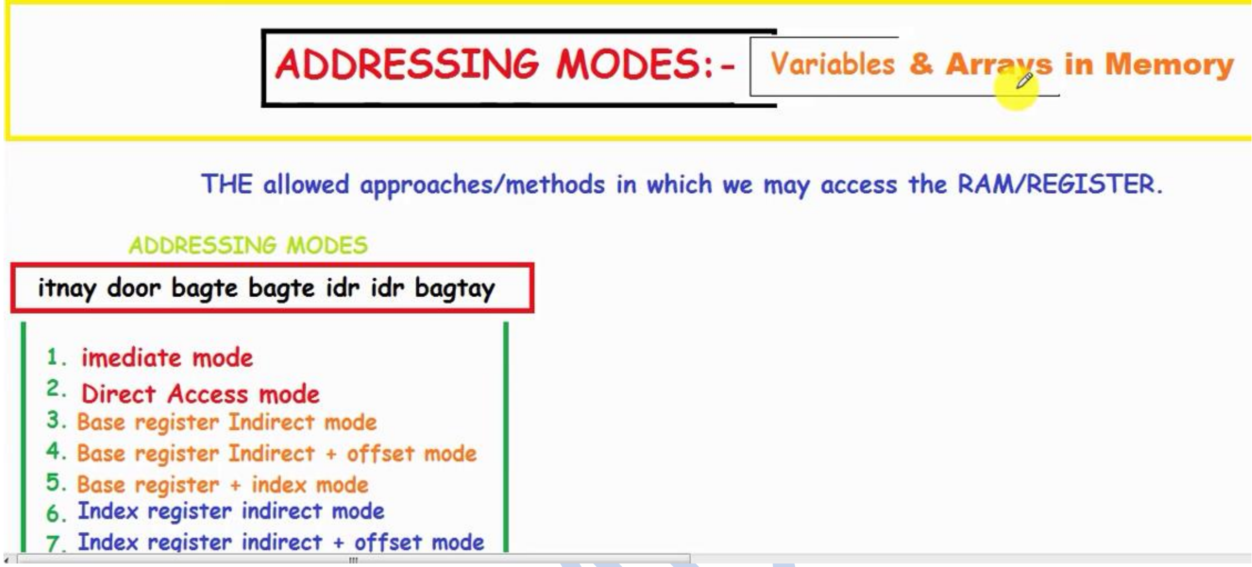
```
  1111 1111 0000 0001 0000
+ 0000 1111 1111 0000 0010
1 0000 1110 1111 0001 0010
```

1+1 =2 hota hai, but answer 0 aye ga or 1 carry jayega, agar addition kartay 3 aya to 1 ans aye ga or ake 1 carry ho jayga

Last ma carry a gya jabky 8088 k processor ma just 20 digit store hotay hain issi liye ye last carry ignore kar dain gay

Effective Address or Offset **Address**: The offset for a memory operand is called the operand's **effective address** or EA. It is an unassigned 16 bit number that expresses the operand's distance in bytes from the beginning of the segment in which it resides. In **8086** we have base registers and index registers

Lecture 11



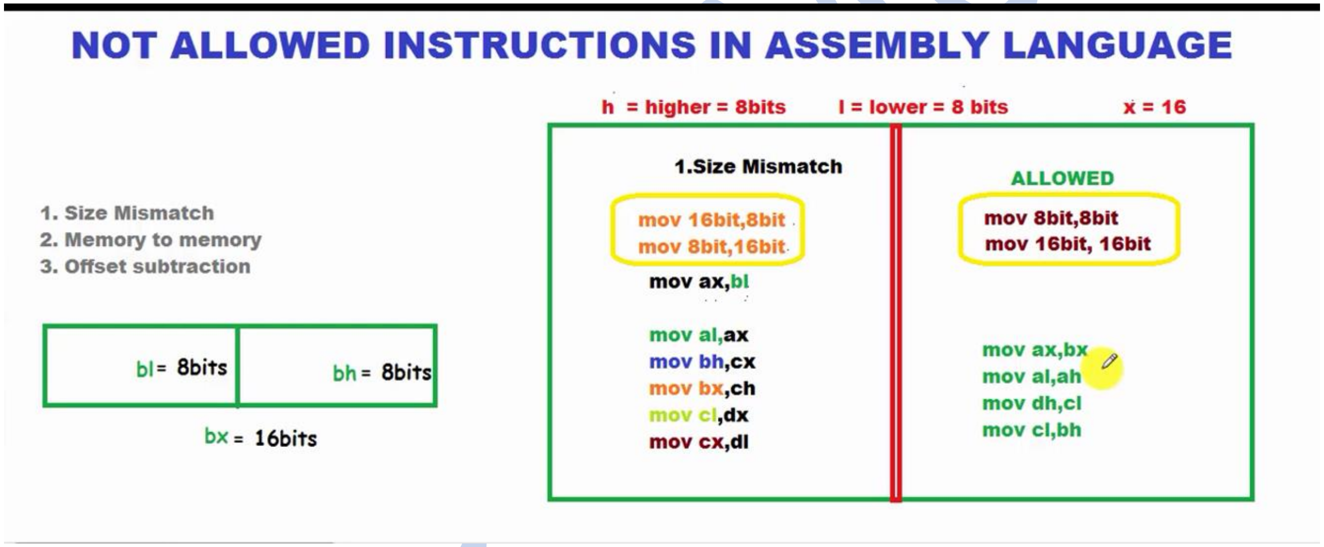
ADDRESSING MODES:- Variables & Arrays in Memory

THE allowed approaches/methods in which we may access the RAM/REGISTER.

ADDRESSING MODES

itnay door bagte bagte idr idr bagtay

1. imediate mode
2. Direct Access mode
3. Base register Indirect mode
4. Base register Indirect + offset mode
5. Base register + index mode
6. Index register indirect mode
7. Index register indirect + offset mode



NOT ALLOWED INSTRUCTIONS IN ASSEMBLY LANGUAGE

1. Size Mismatch
2. Memory to memory
3. Offset subtraction

bl= 8bits | bh= 8bits

bx= 16bits

h = higher = 8bits l = lower = 8 bits x = 16

1.Size Mismatch

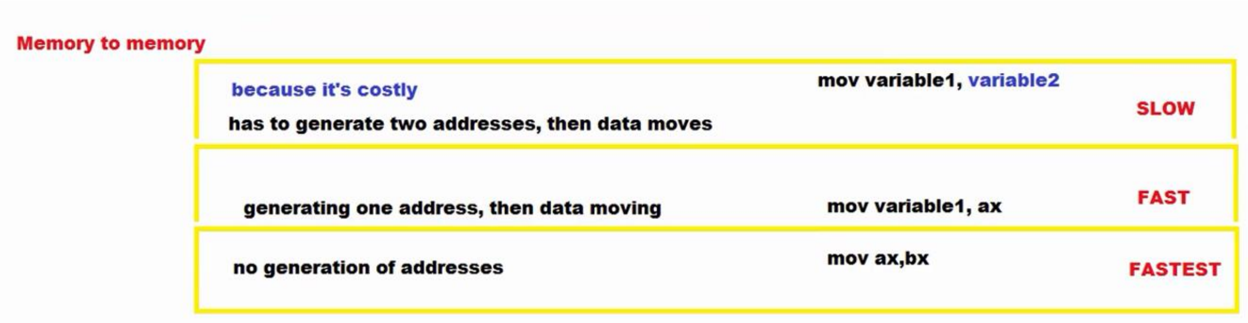
mov 16bit,8bit
mov 8bit,16bit

```
mov ax,bl

mov al,ax
mov bh,cx
mov bx,ch
mov cl,dx
mov cx,dl
```

ALLOWED

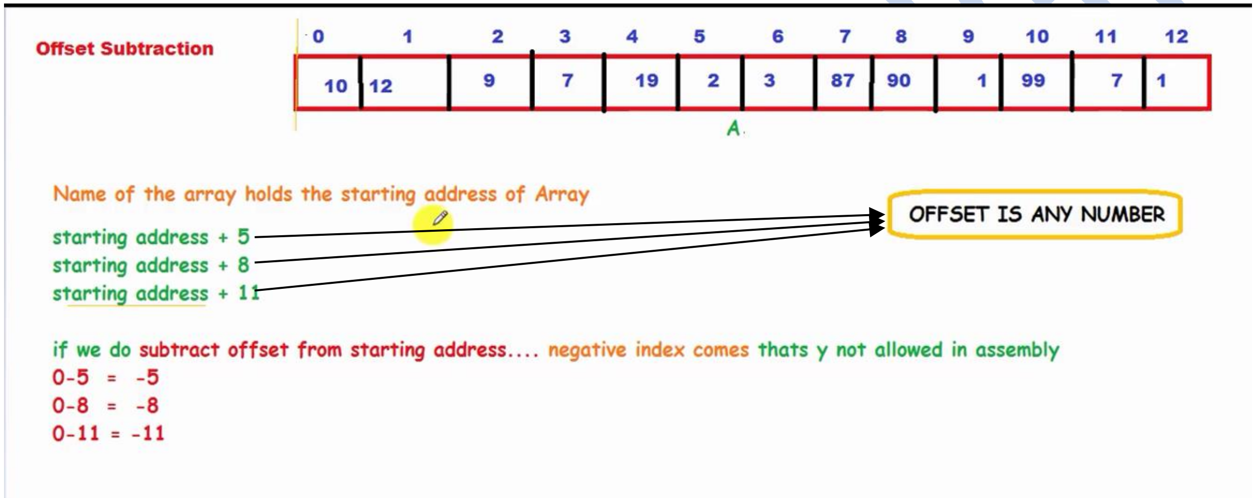
mov 8bit,8bit
mov 16bit, 16bit

```
mov ax,bx
mov al,ah
mov dh,cl
mov cl,bh
```



Memory to memory

because it's costly has to generate two addresses, then data moves	mov variable1, variable2	SLOW
generating one address, then data moving	mov variable1, ax	FAST
no generation of addresses	mov ax,bx	FASTEST



Offset Subtraction

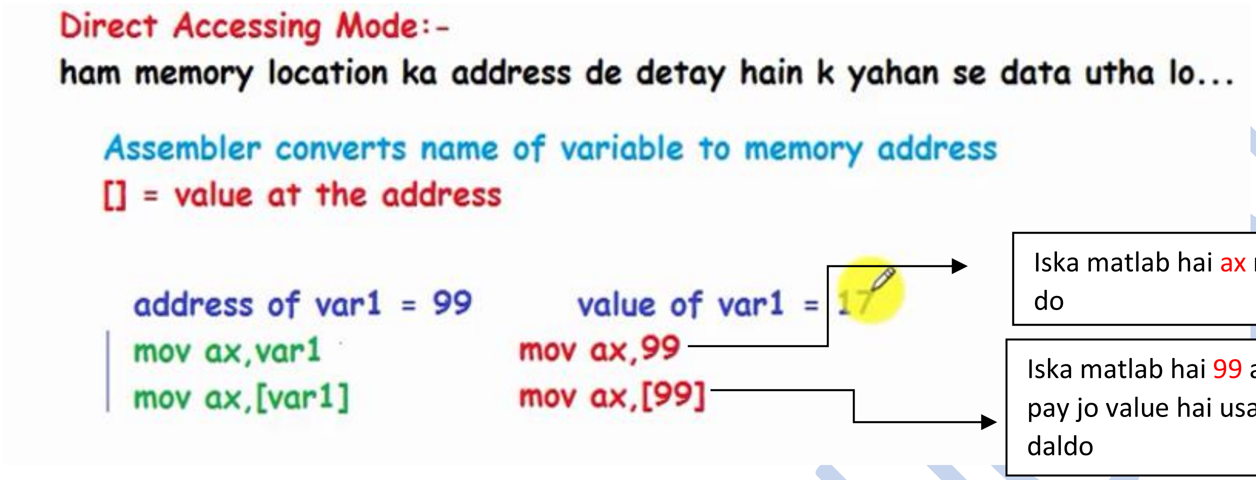
0	1	2	3	4	5	6	7	8	9	10	11	12
10	12	9	7	19	2	3	87	90	1	99	7	1

Name of the array holds the starting address of Array

starting address + 5
starting address + 8
starting address + 11

if we do subtract offset from starting address.... negative index comes thats y not allowed in assembly

0-5 = -5
0-8 = -8
0-11 = -11



Direct Accessing Mode:-

ham memory location ka address de detay hain k yahan se data utha lo...

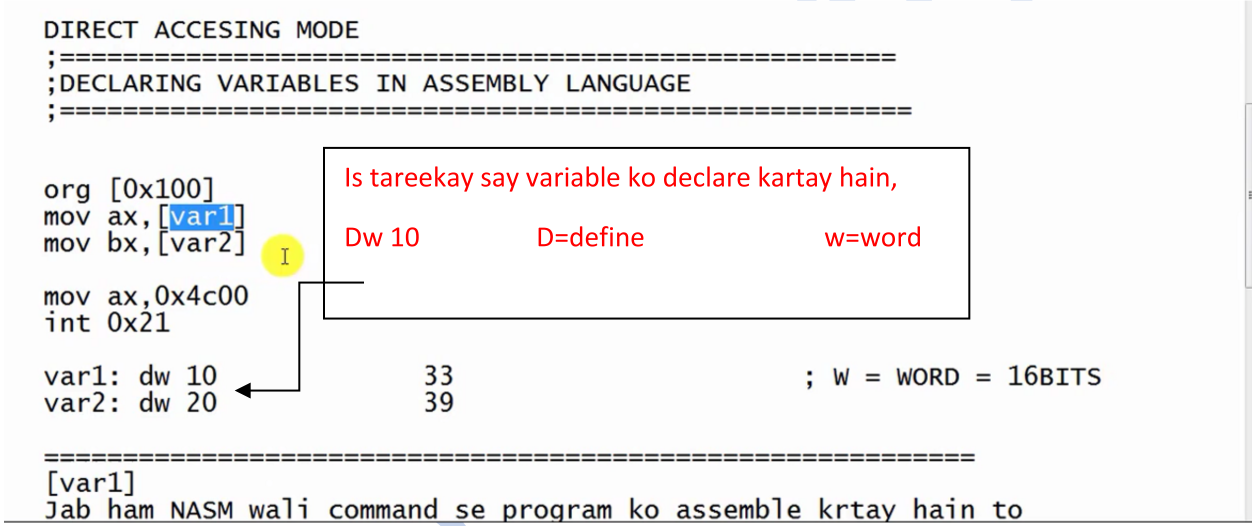
Assembler converts name of variable to memory address

[] = value at the address

```
address of var1 = 99        value of var1 = 17
mov ax,var1                 mov ax,99
mov ax,[var1]               mov ax,[99]
```

Iska matlab hai ax ma 99 dall do

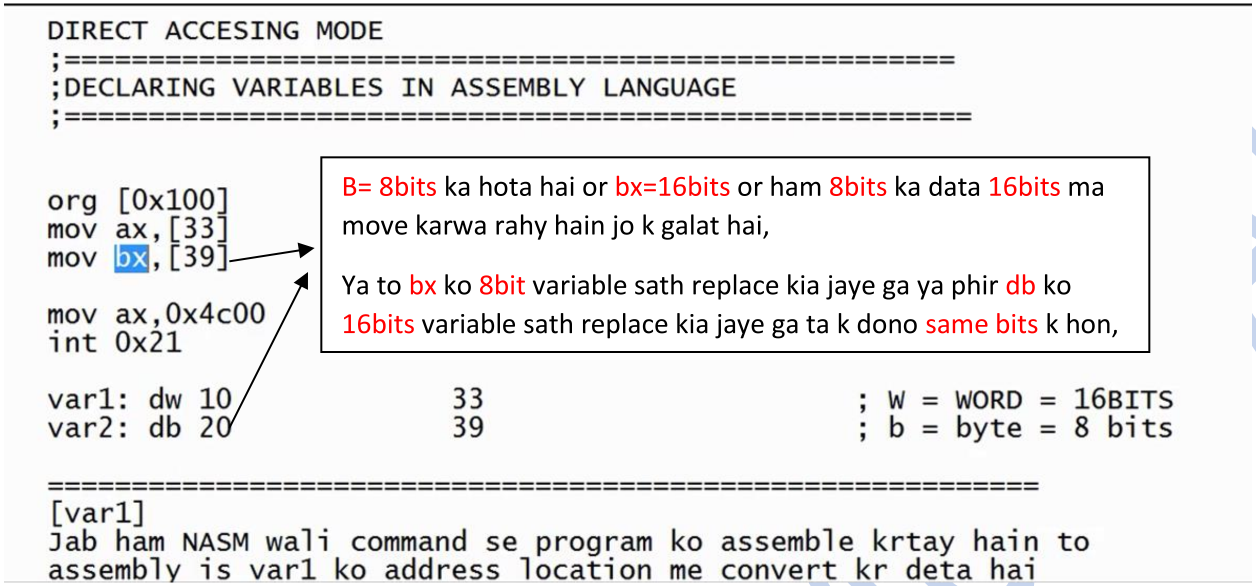
Iska matlab hai 99 address pay jo value hai usay ax ma daldo



```
DIRECT ACCESING MODE
;=====================================================
;DECLARING VARIABLES IN ASSEMBLY LANGUAGE
;======================================================

org [0x100]
mov ax,[var1]
mov bx,[var2]

mov ax,0x4c00
int 0x21

var1: dw 10             33                        ; W = WORD = 16BITS
var2: dw 20             39

===========================================================
[var1]
Jab ham NASM wali command se program ko assemble krtay hain to
```

Is tareekay say variable ko declare kartay hain,

Dw 10 D=define w=word



```
DIRECT ACCESING MODE
;=====================================================
;DECLARING VARIABLES IN ASSEMBLY LANGUAGE
;======================================================

org [0x100]
mov ax,[33]
mov bx,[39]

mov ax,0x4c00
int 0x21

var1: dw 10             33                      ; W = WORD = 16BITS
var2: db 20             39                      ; b = byte = 8 bits

===========================================================
[var1]
Jab ham NASM wali command se program ko assemble krtay hain to
assembly is var1 ko address location me convert kr deta hai
```

B= 8bits ka hota hai or bx=16bits or ham 8bits ka data 16bits ma move karwa rahy hain jo k galat hai,

Ya to bx ko 8bit variable sath replace kia jaye ga ya phir db ko 16bits variable sath replace kia jaye ga ta k dono same bits k hon,

See the table for the list of **data type 8086 architecture** offers.

Pseudo-operation	Stand For
DB	Define Byte
DW	Define Word
DD	Define Doubleword
DQ	Define Quadword
DT	Define Ten Byte

Data types or Data sizes:
In Assembly language, there are no distinct data types like char,string,int,float,double,etc Instead there are very basic data types according to their sizes. They are:

1. byte – 8 bits
2. word – 16 bits
3. dword – 32 bits
4. qword – 64 bits
5. Real4 – 32 bit float
6. Real8 – 64 bit float
7. Real10 – 80 bit float

General purpose registers in 8086 microprocessor

General purpose registers are used to store temporary data within the microprocessor. There are 8 general purpose registers in 8086 microprocessor.

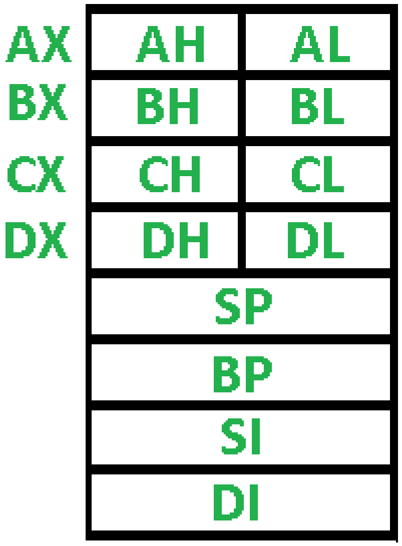


Figure – General purpose registers

1. **AX** – This is the accumulator. It is of 16 bits and is divided into two 8-bit registers AH and AL to also perform 8-bit instructions.
   It is generally used for arithmetical and logical instructions but in 8086 microprocessor it is not mandatory to have accumulator as the destination operand.

   Example:

   ```
   ADD AX, AX (AX = AX + AX)
   ```

2. **BX** – This is the base register. It is of 16 bits and is divided into two 8-bit registers BH and BL to also perform 8-bit instructions.
   It is used to store the value of the offset.
   Example:

   ```
   MOV BL, [500] (BL = 500H)
   ```

3. **CX –** This is the counter register. It is of 16 bits and is divided into two 8-bit registers CH and CL to also perform 8-bit instructions.
   It is used in looping and rotation.

   Example:

   ```
   MOV CX, 0005
   LOOP
   ```

4. **DX –** This is the data register. It is of 16 bits and is divided into two 8-bit registers DH and DL to also perform 8-bit instructions.
   It is used in multiplication an input/output port addressing.
   Example:

   ```
   MUL BX (DX, AX = AX * BX)
   ```

5. **SP –** This is the stack pointer. It is of 16 bits.
   It points to the topmost item of the stack. If the stack is empty the stack pointer will be (FFFE)H. It’s offset address relative to stack segment.
6. **BP –** This is the base pointer. It is of 16 bits.
   It is primary used in accessing parameters passed by the stack. It’s offset address relative to stack segment.
7. **SI –** This is the source index register. It is of 16 bits.
   It is used in the pointer addressing of data and as a source in some string related operations. It’s offset is relative to data segment.
8. **DI –** This is the destination index register. It is of 16 bits.
   It is used in the pointer addressing of data and as a destination in some string related operations.It’s offset is relative to extra segment.

Computer Architecture and Assembly Language Programming CS401
Lecture No: 1

Address, Data, and Control Buses

A computer system comprises of a processor, memory, and I/O devices. I/O is used for interfacing with the external world, while memory is the processor’s internal world. Processor is the core in this picture and is responsible for performing operations.

□ There must be a mechanism to inform memory that we want to do the read operation
□ There must be a mechanism to inform memory that we want to read precisely which element
□ There must be a mechanism to transfer that data element from memory to processor

The group of bits that the processor uses to inform the memory about which element to read or write is collectively known as the *address bus*. (**address bus generate binary numbers**)

Another important bus called the *data bus* is used to move the data from the memory to the processor in a read operation and from the processor to the memory in a write operation.

The third group consists of miscellaneous independent lines used for control purposes. For example, one line of the bus is used to inform the memory about whether to do the read operation or the write operation. These lines are collectively known as the *control bus*. (**precise synchronization between the processor and the memory is the responsibility of**

the control bus. the control bus is a bidirectional bus and can carry information from processor to memory as well as from memory to processor.)

The address bus is unidirectional and address always travels from processor to memory.

The number of bits in a cell is called the *cell width*.

Registers:

Operands are the data on which we want to perform a certain operation.

There are temporary storage places inside the processor called *registers*.

Registers are like a scratch pad ram inside the processor and their operation is very much like normal memory cells. They have precise locations and remember what is placed inside them. They are used when we need more than one data element inside the processor at one time. Registers are more than one in number r0, r1, r2

Lecture No: 2

Accumulator

There is a central register in every processor called the accumulator. Traditionally all mathematical and logical operations are performed on the accumulator. The word size of a processor is defined by the width of its accumulator. A 32bit processor has an accumulator of 32 bits.

Pointer, Index, or Base Register

The name varies from manufacturer to manufacturer, but the basic distinguishing property is that it does not hold data but holds the address of data.

Flags Register or Program Status Word

This is a special register in every architecture called the flags register or the program status word. Like the accumulator it
is an 8, 16, or 32 bits register but unlike the accumulator it is meaningless as a unit, rather the individual bits carry
different meanings.
The bits of the accumulator work in parallel as a unit and each bit mean the same thing. The bits of the flags register work
independently and individually, and combined its value is meaningless.

Program Counter or Instruction Pointer

Everything must translate into a binary number for our dumb processor to understand it, be it an operand or an operation
itself. Therefore, the instructions themselves must be translated into numbers. (the symbols are called instruction
Mnemonic)
For the processor 152 might be the add instruction. Just this one number tells it that it has to add, where its operands are,
and where to store the result. This number is called the ***opcode***.

INSTRUCTION GROUPS

Usual opcodes in every processor exist for moving data, arithmetic and logical manipulations etc. However, their
mnemonics vary depending on the will of the manufacturer. Some manufacturers name the mnemonics for data
movement instructions as “move,” some call it “load” and “store” and still other names are present. But the basic set of

instructions is similar in every processor. A grouping of these instructions makes learning a new processor quick and easy.

Just the group an instruction belongs tells a lot about the instruction.

Data Movement Instructions

These instructions are used to move data from one place to another. These places can be registers, memory, or even inside

peripheral devices. Some examples are: mov ax, bx lad 1234

Arithmetic and Logic Instructions

Arithmetic instructions like addition, subtraction, multiplication, division and Logical instructions like logical and, logical

or, logical xor, or complement are part of this group. Some examples are: and ax, 1234 add bx, 0534 add bx,

[1200]

Program Control Instructions

The instruction pointer points to the next instruction and instructions run one after the other with the help of this register.

Special Instructions

Another group called special instructions works like the special service commandos. They allow changing specific

processor behaviors and are used to play with it.

Lecture No: 3

REGISTER ARCHITECTURE

The iAPX88 architecture consists of **14 registers.**

General Registers (AX, BX, CX, and DX)

The registers AX, BX, CX, and DX behave as general purpose registers in Intel architecture and do some specific
functions in addition to it. X in their names stand for extended meaning 16bit registers. For example, AX means we are
referring to the extended 16bit "A" register. Its upper and lower byte are separately accessible as AH (A high byte) and
AL (A low byte). All general purpose registers can be accessed as one 16bit register or as two 8bit registers. Any change
in AH or AL is reflected in AX as well. The **A of AX** stands for Accumulator. **The B of BX** stands for Base because of its
role in memory addressing The **C of CX** stands for Counter
as there are certain instructions that work with an automatic count in the CX register. The **D of DX** stands for Destination
as it acts as the destination in I/O operations.

Index Registers (SI and DI)

SI and DI stand for source index and destination index respectively. These are the index registers of the Intel architecture
which hold address of data and used in memory access. SI and DI are 16bit and cannot be used as 8bit register pairs like
AX, BX, CX, and DX.

Instruction Pointer (IP)

This is the special register containing the address of the next instruction to be executed. No mathematics or memory
access can be done through this register. It is out of our direct control and is automatically used.

Stack Pointer (SP)

It is a memory pointer and is used indirectly by a set of instructions. This register will be explored in the discussion of the
system stack.

Base Pointer (BP)

It is also a memory pointer containing the address in a special area of memory called the stack and will be explored
alongside SP in the discussion of the stack.

Flags Register

The flags register is not meaningful as a unit rather it is bit wise significant and accordingly each bit is named separately.

The Intel FLAGS register has its bits organized as follows:

15 14 13 12 11 10 9 8 7 6 5 3 2 1 0

O D I T S Z A P C

Overflow Flag - Direction Flag - Interrupt Flag - Trap Flag - Sign Flag - Zero Flag - Auxiliary Carry -Parity –
Carry

Segment Registers (CS, DS, SS, and ES)

The code segment register, data segment register, stack segment register, and the extra segment register are special
registers related to the Intel segmented memory model.

the Intel way of writing things is: **operation destination, source - operation destination - operation source – operation**

Lecture No: 4

SEGMENTED MEMORY MODEL

Rationale

In earlier processors like 8080 and 8085 the linear memory model was used to access memory. In linear memory model
the whole memory appears like a single array of data. 8080 and 8085 could access a total memory of 64K using the 16
lines of their address bus. When designing iAPX88 the Intel designers wanted to remain compatible with 8080 and 8085
however 64K was too small to continue with, for their new processor. To get the best of both worlds they introduced the
segmented memory model in 8088.

Mechanism

The segmented memory model allows multiple functional windows into the main memory, a code window, a data window
etc. The processor sees code from the code window and data from the data window. The size of one window is restricted
to 64K. 8085 software fits in just one such window. It sees code, data, and stack from this one window, so downward
compatibility is attained.
CS holds the zero or the base of code. DS holds the zero of data. Or we can say CS tells how high code from
the floor is, and DS tells how high data from the floor is, while SS tells how high the stack is. One extra segment ES can
be used if we need to access two distant areas of memory at the same time that both cannot be seen through the same
window. ES also has special role in string instructions. ES is used as an extra data segment and cannot be used as an extra

code or stack segment.
The IP register cannot work alone. It needs the CS register to open a 64K window in the 1MB memory
If the window is moved i.e. the CS register is changed, IP will change its behavior accordingly and start selecting from the
new window.

Physical Address Calculation

Now for the whole megabyte we need 20 bits while CS and IP are both 16bit registers. We need a mechanism to make a
20bit number out of the two 16bit numbers. Consider that the segment value is stored as a 20 bit number with the lower
four bits zero and the offset value is stored as another 20 bit number with the upper four bits zeroed. The two are added to
produce a 20bit absolute address.

Paragraph Boundaries

As the segment value is a 16bit number and four zero bits are appended to the right to make it a 20bit number, segments
can only be defined a 16byte boundaries called paragraph boundaries. The first possible segment value is 0000 meaning a
physical base of 00000 and the next possible value of 0001 means a segment base of 00010 or 16 in decimal. Therefore
segments can only be defined at 16 byte boundaries.

Overlapping Segments

We can also observe that in the case of our program CS, DS, SS, and ES all had the same value in them. This is called

overlapping segments so that we can see the same memory from any window. This is the structure of a COM file. The
segment, offset pair is called a logical address, while the 20bit address is a physical address which is the real thing.
Logical addressing is a mechanism to access the physical memory. As we have seen three different logical addresses
accessed the same physical address.

Lecture No: 5

DATA DECLARATION

The first instruction of our first assembly language program was “mov ax, 5.” Here MOV was the opcode; AX was the
destination operand, while 5 was the source operand. The value of 5 in this case was stored as part of the instruction
encoding. In the opcode B80500, B8 was the opcode and 0500 was the operand stored immediately afterwards. Such an
operand is called an immediate operand. It is one of the many types of operands available.
To declare a part of our program as holding data instead of instructions we need a couple of very basic but special
assembler directives. The first directive is “define byte” written as “db.” db somevalue
The other directive is “define word” or “dw” with the same syntax as “db” but reserving a whole word of 16 bits instead
of a byte.
There are directives to declare a double or a quad word as well but we will restrict ourselves to byte and word declarations
for now. For single byte we use db and for two bytes we use dw.

a symbol associated to a point in the program is called a label and is written as the label name followed by a colon.

Lecture No: 6

SIZE MISMATCH ERRORS

If we change the directive in the last example from DW to DB, the program will still assemble and debug without errors,
however the results will not be the same as expected. To keep the declarations and their access synchronized is the
responsibility of the programmer and not the assembler. The assembler allows the programmer to do everything he wants
to do, and that can possibly run on the processor. The assembler only keeps us from writing illegal instructions which the
processor cannot execute. This is the difference between a syntax error and a logic error.
the processor knew the size of the data movement
operation from the size of the register involved, for example in “mov ax, [num1]” memory can be accessed as byte or as
word, it has no hard and fast size, but the AX register tells that this operation has to be a word operation. Similarly in
“mov al, [num1]” the AL register tells that this operation has to be a byte operation. However in “mov ax, bl” the AX
register tells that the operation has to be a word operation while BL tells that this has to be a byte operation. The
assembler will declare that this is an illegal instruction.
The instruction “mov [num1], [num2]” is illegal as previously discussed not because of data movement size but because

memory to memory moves are not allowed at all. The instruction “mov [num1], 5” is legal but there is no way for the
processor to know the data movement size in this operation. The variable num1 can be treated as a byte or as a word. Such
instructions are declared ambiguous by the assembler. Therefore to resolve its ambiguity we clearly tell our intent to the
assembler in one of the following ways.
mov byte [num1], 5
mov word [num1], 5

REGISTER INDIRECT ADDRESSING

We have done very elementary data access till now. Assume that the numbers we had were 100 and not just three. This
way of adding them will cost us 200 instructions. There must be some method to do a task repeatedly on data placed in
consecutive memory cells. The key to this is the need for some register that can hold the address of data.
There are four registers in iAPX88 architecture that can hold address of data and they are BX, BP, SI, and DI.
Labels can be used on code as well. Just like data labels they remember the address at which they are used. The
assembler does not differentiate between code labels and data labels. The programmer is responsible for using a data label
as data and a code label as code. The label l1 in this case is the address of the following instruction.
JNZ stands for “jump if not zero.” NZ is the condition in this instruction. So the instruction is read as “jump to the

location l1 if the zero flag is not set.” And revisiting the zero flag definition “the zero flag is set if the last mathematical or logical operation has produced a zero in its destination.”

a register is used to reference memory so this form of access is called register indirect memory access. We used the BX register for it and the B in BX and BP stands for base therefore we call register indirect memory access using BX or BP, “based addressing.” Similarly, when SI or DI is used we name the method “indexed addressing.” They have the same functionality, with minor differences because of which the two are called base and index.

Lecture No: 7

REGISTER + OFFSET ADDRESSING

Direct addressing and indirect addressing using a single register are two basic forms of memory access. Another possibility is to use different combinations of direct and indirect references.

This form of access is of the register indirect family and is called base + offset or index + offset depending on whether BX or BP is used or SI or DI is used.

SEGMENT ASSOCIATION: All the addressing mechanisms in iAPX88 return a number called *effective address*. For example in base + offset addressing, neither the base nor the offset alone tells the desired cell in memory to be accessed. It is only after the addition is done that the processor knows which cell to be accessed. This number which came as the result of addition is called the effective address. But the effective address is just an offset and is meaningless without a segment.

Only after the segment is known, we can form the physical address that is needed to access a memory cell.

Segmentation is there and it’s all happening relative to a segment base. We saw DS, CS, SS, and ES inside the debugger.

CS is associated to IP by default; rather it is tied with it. It cannot access memory in any other segment.

In case of data, there is a bit relaxation and nothing is tied. Rather there is a default association which can be overridden.

In the case of register indirect memory access, if the register used is one of SI, DI, or BX the default segment is DS. If however the register used is BP the default segment used is SS. The stack segment has a very critical and fine use and there is a reason why BP is attached to SS by default. IP is tied to CS while SP is tied to SS. The association of these registers cannot be changed; they are locked with no option. Others are not locked and can be changed.

To override the association for one instruction of one of the registers BX, BP, SI or DI, we use the segment override prefix. For example “mov ax, [cs:bx]” associates BX with CS for this one instruction. For the next instruction the default association will come back to act. The processor places a special byte before the instruction called a prefix.

The important thing to note is that CS, DS, SS, and ES all had the same value. The value itself is not important but the fact that all had the same value is important. All four segment windows exactly overlap. Whatever segment register we use the same physical memory will be accessed.

ADDRESS WRAPAROUND: There are two types of wraparounds. One is within a single segment and the other is inside the whole physical memory. Segment wraparound occurs when during the effective address calculation a carry is generated. This carry is dropped giving the effect that when we try to access beyond the segment limit, we are actually wrapped around to the first cell in the segment. For example if BX=9100, DS=1500 and the access is [bx+0x7000] we form the effective address 9100 + 7000 = 10100.

The same can also happen at the time of physical address calculation. For example BX=0100, DS=FFF0 and the access under consideration is [bx+0x0100]. The effective address will be 0200 and the physical address will be 100100.

This is a 21bit answer and cannot be sent on the address bus which is 20 bits wide. The carry is dropped and just like the segment wraparound our physical memory has wrapped around at its very top. When we tried to access beyond limits the actual access is made at the very start. This second wraparound is a bit different in newer processor with more address lines but that will be explained in later chapters.

ADDRESSING MODES SUMMARY: The iAPX88 processor supports seven modes of memory access. Remember that immediate is not an addressing mode but an operand type. Operands can be immediate, register, or memory. If the operand is memory one of the seven addressing modes will be used to access it. The memory access mechanisms can also be written in the general form “base + index + offset”

Direct: A fixed offset is given in brackets and the memory at that offset is accessed. For example “mov [1234], ax” stores the contents of the AX registers in two bytes starting at address 1234 in the current data segment. The instruction “mov [1234], al” stores the contents of the AL register in the byte at offset 1234.

Based Register Indirect: A base register is used in brackets and the actual address accessed depends on the value contained in that register. For example “mov [bx], ax” moves the two byte contents of the AX register to the address contained in the BX register in the current data segment. The instruction “mov [bp], al” moves the one byte content of the AL register to the address contained in the BP register in the current stack segment.

Indexed Register Indirect: An index register is used in brackets and the actual address accessed depends on the value contained in that register. For example “mov [si], ax” moves the contents of the AX register to the word starting at address contained in SI in the current data segment. The instruction “mov [di], ax” moves the word contained in AX to the offset stored in DI in the current data segment.

Based Register Indirect + Offset: A base register is used with a constant offset in this addressing mode. The value contained in the base register is added with the constant offset to get the effective address. For example “mov [bx+300], ax” stores the word contained in AX at the offset attained by adding 300 to BX in the current data segment. The instruction “mov [bp+300], ax” stores the word in AX to the offset attained by adding 300 to BP in the current stack segment.

Indexed Register Indirect + Offset: An index register is used with a constant offset in this addressing mode. **n**The value contained in the index register is added with the constant offset to get the effective address. For example “mov [si+300], ax” moves the word contained in AX to the offset attained by adding 300 to SI in the current data segment and the instruction “mov [di+300], al” moves the byte contained in AL to the offset attained by adding 300 to DI in the current data segment.

Base + Index: One base and one index register is used in this addressing mode. The value of the base register and the index register are added together to get the effective address. For example “mov [bx+si], ax” moves the word contained in the AX register to offset attained by adding BX and SI in the current data segment. The instruction “mov [bp+di], al”

moves the byte contained in AL to the offset attained by adding BP and DI in the current stack segment. Observe that the
default segment is based on the base register and not on the index register.

Base + Index + Offset: This is the most complex addressing method and is relatively infrequently used. A base register,
an index register, and a constant offset are all used in this addressing mode. The values of the base register, the index
register, and the constant offset are all added together to get the effective address. For example "mov [bx+si+300], ax"
moves the word contents of the AX register to the word in memory starting at offset attained by adding BX, SI, and 300 in
the current data segment. Default segment association is again based on the base register. It might be used with the array
base of a two dimensional array as the constant offset, one dimension in the base register and the other in the index
register. This way all calculation of location of the desired element has been delegated to the processor.

Lecture No: 8

COMPARISON AND CONDITIONS: Conditional jump was introduced in the last chapter to loop for the addition of a
fixed number of array elements. The jump was based on the zero flag. There are many other conditions possible in a
program. For example an operand can be greater than another operand or it can be smaller. We use comparisons and

boolean expressions extensively in higher level languages. They must be available is some form in assembly language, otherwise they could not possibly be made available in a higher level language. In fact they are available in a very fine and purified form. The basic root instruction for all comparisons is CMP standing for compare. The operation of CMP is to subtract the source operand from the destination operand, updating the flags without changing either the source or the destination. CMP is one of the key instructions as it introduces the capability of conditional routing in the processor.

A closer thought reveals that with subtraction we can check many different conditions. For example if a larger number is subtracted from a smaller number then borrow is needed. The carry flag plays the role of borrow during the subtraction operation. And in this condition the carry flag will be set. If two equal numbers are subtracted the answer is zero and the zero flag will be set. Every significant relation between the destination and source is evident from the sign flag, carry flag, zero flag, and the overflow flag. CMP is meaningless without a conditional jump immediately following it. Another important distinction at this point is the difference between signed and unsigned numbers. In unsigned numbers only the magnitude of the number is important, whereas in signed numbers both the magnitude and the sign are

important. For example -2 is greater than -3 but 2 is smaller than 3. The sign has affected our comparisons.
However for signed numbers JG and JL will work properly and for unsigned JA and JB will work properly and
not the other way around.
CONDITIONAL JUMPS: For every interesting or meaningful situation of flags, a conditional jump is there. For
example JZ and JNZ check the zero flag. If in a comparison both operands are same, the result of subtraction will be zero
and the zero flag will be set. Thus JZ and JNZ can be used to test equality. That is why there are renamed versions JE and
JNE read as jump if equal or jump if not equal. They seem more logical in writing but mean exactly the same thing with
the same opcode. Many jumps are renamed with two or three names for the same jump, so that the appropriate logic can
be conveyed in assembly language programs. This renaming is done by Intel and is a standard for iAPX88. JC and JNC
test the carry flag. For example we may need to test whether there was an overflow in the last unsigned addition or
subtraction. Carry flag will also be set if two unsigned numbers are subtracted and the first is smaller than the second.
Therefore the renamed versions JB, JNAE, and JNB, JAE are there standing for jump if below, jump if not above or
equal, jump if not below, and jump if above or equal respectively. The operation of all jumps can be seen from the

following table.
The CMP instruction sets the flags reflecting the relation of the destination to the source. This is important as
when we say jump if above, then what is above what. The destination is above the source or the source is above the
destination. The JA and JB instructions are related to unsigned numbers. That is our interpretation for the destination and
source operands is unsigned. The 16th bit holds data and not the sign. In the JL and JG instructions standing for jump if
lower and jump if greater respectively, the interpretation is signed. The 16th bit holds the sign and not the data. The
difference between them will be made clear as an elaborate example will be given to explain the difference. One jump is
special that it is not dependant on any flag. It is JCXZ, jump if the CX register is zero. This is because of the special
treatment of the CX register as a counter. This jump is regardless of the zero flag. There is no counterpart or not form of
this instruction.

Lecture No: 9

UNCONDITIONAL JUMP: Till now we have been placing data at the end of code. There is no such restriction and we
can define data anywhere in the code. Taking the previous example, if we place data at the start of code instead of at the
end and we load our program in the debugger. We can see our data placed at the start but the debugger is intending to start

execution at our data. The COM file definition said that the first executable instruction is at offset 0100 but we have placed data there instead of code. So the debugger will try to interpret that data as code and showed whatever it could make up out of those opcodes. We introduce a new instruction called JMP. It is the unconditional jump that executes regardless of the state of all flags. So we write an unconditional jump as the very first instruction of our program and jump to the next instruction that follows our data declarations. This time 0100 contains a valid first instruction of our program.

RELATIVE ADDRESSING: Inside the debugger the instruction is shown as JMP 0119 and the location 0119 contains the original first instruction of the logic of our program. This jump is unconditional, it will always be taken. Now looking at the opcode we see F21600 where F2 is the opcode and 1600 is the operand to it. 1600 is 0016 in proper word order.

0119 is not given as a parameter rather 0016 isgiven.

This is position relative addressing in contrast to absolute addressing. It is not telling the exact address rather it is telling how much forward or backward to go from the current position of IP in the current code segment. So the instruction means to add 0016 to the IP register. At the time of execution of the first instruction at 0100 IP was pointing to the next instruction at 0103, so after adding 16 it became 0119, the desired target location. The mechanism is important to know, however all calculations in this mechanism are done by the assembler and by the processor. We just use a label with the JMP instruction and are ensured that the instruction at the target label will be the one to be executed.

TYPES OF JUMP: The three types of jump, near, short, and far, differ in the size of instruction and the range of memory they can jump to with the smallest short form of two bytes and a range of just 256 bytes to the far form of five bytes and a range covering the whole memory.

Near Jump: When the relative address stored with the instruction is in 16 bits as in the last example the jump is called a near jump. Using a near jump we can jump the lower part. A negative number actually is a large number and works this way using the wraparound behavior.

Short Jump: If the offset is stored in a single byte as in 75F2 with the opcode 75 and operand F2, the jump is called a short jump. F2 is added to IP as a signed byte. If the byte is negative the complement is negated from IP otherwise the byte is added. Unconditional jumps can be short, near, and far. The far type is yet to be discussed. Conditional jumps can only be short. A short jump can go +127 bytes ahead in code and -128 bytes backwards and no more. This is the limitation of a byte in singed representation.

Far Jump: Far jump is not position relative but is absolute. Both segment and offset must be given to a far jump. The previous two jumps were used to jump within a segment. Sometimes we may need to go from one code segment to another, and near and short jumps cannot take us there. Far jump must be used and a two byte segment and a two byte offset are given to it. It loads CS with the segment part and IP with the offset part. Execution therefore resumes from that location in physical memory. The three instructions that have a far form are JMP, CALL, and RET, are related to program control. Far capability makes intra segment control possible.

Lecture No: 10

Inside the debugger we observe that the JBE is changed to JNA due to the same reason as discussed for JNE and JNZ. The passes change the data in the same manner as we presented in our illustration above. If JBE in the code is changed to JAE the sort will change from ascending to descending. For signed numbers we can use JLE and JGE respectively for ascending and descending sort.

To clarify the difference of signed and unsigned jumps we change the data array in the last program to include some negative numbers as well. When JBE will be used on this data, i.e. with unsigned interpretation of the data and an ascending sort, the negative numbers will come at the end after the largest positive number. However JLE will bring the negative numbers at the very start of the list to bring them in proper ascending order according to a signed interpretation,

even though they are large in magnitude.
The JBE instruction will treat this data as an unsigned number and will cater only for the magnitude ignoring the
sign. If the program is loaded in the debugger, the numbers will appear in their hexadecimal equivalent.
MULTIPLICATION ALGORITHM: With the important capability of decision making in our repertoire we move on to
the discussion of an algorithm, which will help us uncover an important set of instructions in our processor used for bit
manipulations.
Multiplication is a common process that we use, and we were trained to do in early schooling. Remember multiplying by
a digit and then putting a cross and then multiplying with the next digit and putting two crosses and so on and summing
the intermediate results in the end. Very familiar process but we never saw the process as an algorithm, and we need to
see it as an algorithm to convey it to the processor.
SHIFTING AND ROTATIONS: The set of shifting and rotation instructions is one of the most useful set in any
processor's instruction set. They simplify really complex tasks to a very neat and concise algorithm. The following
shifting and rotation operations are available in our processor.
Shift Logical Right (SHR): The shift logical right operation inserts a zero from the left and moves every bit one position
to the right and copies the rightmost bit in the carry flag.

Shift Logical Left (SHL) / Shift Arithmetic Left (SAL): The shift logical left operation is the exact opposite of shift logical right. In this operation the zero bit is inserted from the right and every bit moves one position to its left with the most significant bit dropping into the carry flag. Shift arithmetic left is just another name for shift logical left.

Shift Arithmetic Right (SAR): A signed number holds the sign in its most significant bit. If this bit was one a logical right shifting will change the sign of this number because of insertion of a zero from the left. The sign of a signed number should not change because of shifting. The operation of shift arithmetic right is therefore to shift every bit one place to the right with a copy of the most significant bit left at the most significant place. The bit dropped from the right is caught in the carry basket. The sign bit is retained in this operation.

Rotate Right (ROR): In the rotate right operation every bit moves one position to the right and the bit dropped from the right is inserted at the left. This bit is also copied into the carry flag. The operation can be understood by imagining that the pipe used for shifting.

Rotate Left (ROL): In the operation of rotate left instruction, the most significant bit is copied to the carry flag and is inserted from the right, causing every bit to move one position to the left. It is the reverse of the rotate right instruction.

Rotation can be of eight or sixteen bits.

Rotate Through Carry Right (RCR): In the rotate through carry right instruction, the carry flag is inserted from the left,
every bit moves one position to the right, and the right most bit is dropped in the carry flag. Effectively this is a nine bit or
a seventeen bit rotation instead of the eight or sixteen bit rotation as in the case of simple rotations. Imagine

Rotate Through Carry Left (RCL): The exact opposite of rotate through carry right instruction is the rotate through
carry left instruction. In its operation the carry flag is inserted from the right causing every bit to move one location to its
left and the most significant bit occupying the carry flag.

Lecture No: 11

MULTIPLICATION IN ASSEMBLY LANGUAGE: The multiplication with the new algorithm
We try to identify steps of our algorithm. First we set the result to zero. Then we check the right most bit of multiplier. If
it is one add the multiplicand to the result, and if it is zero perform no addition. Left shift the multiplicand before the next
bit of multiplier is tested. The left shifting of the multiplicand is performed regardless of the value of the multiplier's right
most bit. Just like the crosses in traditional multiplication are always placed to mark the ones, tens, thousands, etc. places.
Then check the next bit and if it is one add the shifted value of the multiplicand to the result. Repeat for as many digits as
there are in the multiplier, 4 in our example. Formulating the steps of the algorithm we get:

- Shift the multiplier to the right.

- If CF=1 add the multiplicand to the result.
- Shift the multiplicand to the left.

EXTENDED OPERATIONS: We performed a 4bit multiplication to explain the algorithm however the real advantage of the computer is when we ask it to multiply large numbers, Numbers whose multiplication takes real time. If we have an 8bit number we can do the multiplication in word registers, but are we limited to word operations? What if we want to multiply 32bit or even larger numbers? We are certainly not limited. Assembly language only provides us the basic building blocks. We build a plaza out of these blocks, or a building, or a classic piece of architecture is only dependent upon our imagination. With our logic we can extend these algorithms as much as we want. Our next example will be multiplication of 16bit numbers to produce a 32bit answer. However for a 32bit answer we need a way to shift a 32bit number and a way to add 32bit numbers. We cannot depend on 16bit shifting as we have 16 significant bits in our multiplicand and shifting any bit towards the left may drop a valuable bit causing a totally wrong result. A valuable bit means any bit that is one. Dropping a zero bit doesn't cause any difference. So we place the 16it number in 32bit space with the upper 16 bits zeroed so that the sixteen shift operations don't cause any valuable bit to drop. Even though the numbers were 16bit we need 32bit operations to multiply correctly.

Extended Shifting: Using our basic shifting and rotation instructions we can effectively shift a 32bit number in memory
word by word. We cannot shift the whole number at once since our architecture is limited to word operations. The
algorithm we use consists of just two instructions and we name it extended shifting.

```
num1: dd 40000
shl word [num1], 1
rcl word [num1+2], 1
```

The DD directive reserves a 32bit space in memory, however the value we placed there will fit in 16bits. So we can safely
shift the number left 16 times. The least significant word is accessible at num1 and the most significant word is accessible
at num1+2.

Extended Addition and Subtraction: We also needed 32bit addition for multiplication of 16bit numbers. The idea of
extension is same here. However we need to introduce a new instruction at this place. The instruction is ADC or "add with
carry." Normal addition has two operands and the second operand is added to the first operand. However ADC has three
operands. The third implied operand is the carry flag. The ADC instruction is specifically placed for extending the
capability of ADD. Numbers of any size can be added using a proper combination of ADD and ADC. All basic building
blocks are provided for the assembly language programmer, and the programmer can extend its capabilities as much as

needed by using these fine instructions in appropriate combinations.

Further clarifying the operation of ADC, consider an instruction “ADC AX, BX.” Normal addition would have just added BX to AX, however ADC first adds the carry flag to AX and then adds BX to AX. Therefore the last carry is also included in the result.

The algorithm should be apparent by now. The lower halves of the two numbers to be added are first added with a normal addition. For the upper halves a normal addition would lose track of a possible carry from the lower halves and the answer would be wrong. If a carry was generated it should go to the upper half. Therefore the upper halves are added with an addition with carry instruction.

```
dest: dd 40000
src: dd 80000
mov ax, [src]
add word [dest], ax
mov ax, [src+2]
adc word [dest+2], ax
```

For subtraction the same logic will be used and just like addition with carry there is an instruction to subtract with borrows called SBB. Borrow in the name means the carry flag and is used just for clarity. Or we can say that the carry flag holds the carry for addition instructions and the borrow for subtraction instructions. Also the carry is generated at the 17th bit and the borrow is also taken from the 17th bit. Also there is no single instruction that needs borrow and carry in their independent meanings at the same time. Therefore it is logical to use the same flag for both tasks. We extend subtraction with a very similar algorithm. The lower halves must be subtracted normally while the upper halves must be subtracted with a subtract with borrow instruction so that if the lower halves needed a borrow, a one is subtracted from the upper halves. The algorithm is as under.

```
dest: dd 40000
src: dd 80000
mov ax, [src]
sub word [dest], ax
mov ax, [src+2]
sbb word [dest+2], ax
```

Extended Multiplication: We use extended shifting and extended addition to formulate our algorithm to do extended multiplication. The multiplier is still stored in 16bits since we only need to check its bits one by one. The multiplicand however cannot be stored in 16bits otherwise on left shifting its significant bits might get lost. Therefore it has to be stored in 32bits and the shifting and addition used to accumulate the result must be 32bits as well.

Lecture No: 12

BITWISE LOGICAL OPERATIONS: The 8088 processor provides us with a few logical operations that operate at the bit level. The logical operations are the same as discussed in computer logic design; however our perspective will be a little different. The four basic operations are AND, OR, XOR, and NOT. The important thing about these operations is that they are bitwise. This means that if “and ax, bx” instruction is given, then the operation of AND is applied on corresponding bits of AX and BX. There are 16 AND operations as a result; one for every bit of AX. Bit 0 of AX will be set if both its original value and Bit 0 of BX are set, bit 1 will be set if both its original value and Bit 1 of BX are set, and so on for the remaining bits. These operations are conducted in parallel on the sixteen bits. Similarly the operations of other logical operations are bitwise as well.

AND operation: AND performs the logical bitwise *and* of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the bit is cleared as shown in the truth table. Examples are “and ax, bx” and “and byte [mem], 5.” All possibilities that are legal for addition are also legal for the AND operation. The different thing is the bitwise behavior of this operation.

OR operation: OR performs the logical bitwise “inclusive or” of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if either or both corresponding bits in the original operands are set otherwise the result bit is cleared as shown in the truth table. Examples are “or ax, bx” and “or byte [mem], 5.”

XOR operation: XOR (Exclusive Or) performs the logical bitwise “exclusive or” of the two operands and returns the
result to the destination operand. A bit in the result is set if the corresponding bits of the original operands contain
opposite values (one is set, the other is cleared) otherwise the result bit is cleared as shown in the truth table. XOR is a
very important operation due to the property that it is a reversible operation. It is used in many cryptography algorithms,
image processing, and in drawing operations. Examples are “xor ax, bx” and “xor byte [mem], 5.”

NOT operation: NOT inverts the bits (forms the one’s complement) of the byte or word operand. Unlike the other logical
operations, this is a single operand instruction, and is not purely a logical operation in the sense the others are, but it is still
traditionally counted in the same set. Examples are “not ax” and “not byte [mem]”.

MASKING OPERATIONS

Selective Bit Clearing: Another use of AND is to make selective bits zero in its destination operand. The source operand
is loaded with a mask containing one at positions which are retain their old value and zero at positions which are to be
zeroed. The effect of applying this operation on the destination with mask in the source is to clear the desired bits. This
operation is called masking. For example if the lower nibble is to be cleared then the operation can be applied with F0 in

the source. The upper nibble will retain its old value and the lower nibble will be cleared.

Selective Bit Setting: The OR operation can be used as a masking operation to set selective bits. The bits in the mask are cleared at positions which are to retain their values, and are set at positions which are to be set. For example to set the lower nibble of the destination operand, the operation should be applied with a mask of 0F in the source. The upper nibble will retain its value and the lower nibble will be set as a result.

Selective Bit Inversion: XOR can also be used as a masking operation to invert selective bits. The bits in the mask are cleared at positions, which are to retain their values, and are set at positions, which are to be inverted. For example to invert the lower nibble of the destination operand, the operand should be applied with a mask of 0F in the source. The upper nibble will retain its value and the lower nibble will be set as a result. Compare this with NOT which inverts everything. XOR on the other hand allows inverting selective bits.

Selective Bit Testing: AND can be used to check whether particular bits of a number are set or not. Previously we used shifting and JC to test bits one by one. Now we introduce another way to test bits, which is more powerful in the sense that any bit can be tested anytime and not necessarily in order. AND can be applied on a destination with a 1-bit in the desired position and a source, which is to be checked. If the destination is zero as a result, which can be checked with a JZ

instruction, the bit at the desired position in the source was clear.
However the AND operation destroys the destination mask, which might be needed later as well. Therefore Intel
provided us with another instruction analogous to CMP, which is non-destructive subtraction. This is the TEST instruction
and is a non-destructive AND operation. It doesn’t change the destination and only sets the flags according to the AND
operation. By checking the flags, we can see if the desired bit was set or cleared. We change our multiplication algorithm
to use selective bit testing instead of checking bits one by one using the shifting operations.

Lecture No: 13

PROGRAM FLOW: Instruction are tied to one another forming an execution thread, just like a knitted thread where
pieces of cotton of different sizes are twisted together to form a thread. This thread of execution is our program. The jump
instruction breaks this thread permanently, making a permanent diversion, like a turn on a highway. The conditional jump
selects one of the two possible directions, like right or left turn on a road. So there is no concept of returning. However
there are roundabouts on roads as well that take us back from where we started after having traveled on the boundary of
the round. This is the concept of a temporary diversion. Two or more permanent diversions can take us back from where
we started, just like two or more road turns can take us back to the starting point, but they are still permanent diversions in

their nature. We need some way to implement the concept of temporary diversion in assembly language. We want to create a roundabout of bubble sort, another roundabout of our multiplication algorithm, so that we can enter into the roundabout whenever we need it and return back to wherever we left from after completing the round.

Key point in the above discussion is returning to where we left from, like a loop in a knitted thread. Diversion should be temporary and not permanent. The code of bubble sort written at one place, multiply at another, and we temporarily divert to that place, thus avoiding a repetition of code at a 100 places.

CALL and RET: In every processor, instructions are available to divert temporarily and to divert permanently. The instructions for permanent diversion in 8088 are the jump instructions, while the instruction for temporary diversion is the CALL instruction. The word call must be familiar to the readers from subroutine call in higher level languages. The CALL instruction allows temporary diversion and therefore reusability of code. Now we can place the code for bubble sort at one place and reuse it again and again. This was not possible with permanent diversion. Actually the 8088 permanent diversion mechanism can be tricked to achieve temporary diversion. However it is not possible without getting into a lot of trouble. The key idea in doing it this way is to use the jump instruction form that takes a register as argument.

Therefore this is not impossible but this is not the way it is done.

The natural way to do this is to use the CALL instruction followed by a label, just like JMP is followed by a label.

Execution will divert to the code following the label. Till now the operation has been similar to the JMP instruction. When the subroutine completes we need to return. The RET instruction is used for this purpose. The word return holds in its meaning that we are to return from where we came and need no explicit destination. Therefore RET takes no arguments and transfers control back to the instruction following the CALL that took us in this subroutine. The actual technical process that informs RET where to return will be discussed later after we have discussed the system stack.

Parameters

We intend to write the bubble sort code at one place and CALL it whenever needed. An immediately visible problem is that whenever we call this subroutine it will sort the same array in the same order. However in a real application we will need to sort various arrays of various sizes. We might sometimes need an ascending sort and descending at other times.

Similarly our data may be signed or unsigned. Such pieces of information that may change from invocation to invocation and should be passed from the caller to the subroutine are called parameters. There must be some way of passing these parameters to the subroutine. Revising the subroutine temporary flow breakage mechanism, the most straight forward way is to use registers. The CALL mechanism breaks the thread of execution and does not change registers, except IP which must change for processor to start executing at another place, and SP whose change will be discussed in detail later. Any of the other registers can hold for the subroutine.

Lecture No: 14

STACK: Stack is a data structure that behaves in a first in last out manner. It can contain many elements and there is only one way in and out of the container. When an element is inserted it sits on top of all other elements and when an element is removed the one sitting at top of all others is removed first. To visualize the structure consider a test tube and put some balls in it. The second ball will come above the first and the third will come above the second. When a ball is taken out only the one at the top can be removed. The operation of placing an element on top of the stack is called pushing the element and the operation of removing an element from the top of the stack is called popping the element. The last thing pushed is popped out first; the last in first out behavior.

This top of stack is contained in the SP register. The physical address of the stack is obtained by the SS:SP combination. The stack segment registers tells where the stack is located and the stack pointer marks the top of stack inside this segment.

Whenever an element is pushed on the stack SP is decremented by two as the 8088 stack works on word sized elements.

Single bytes cannot be pushed or popped from the stack. Also it is a decrementing stack. Another possibility is an incrementing stack. A decrementing stack moves from higher addresses to lower addresses as elements are added in it while an incrementing stack moves from lower addresses to higher addresses as elements are added.

The basic use of the stack is to save things and recover from there when needed. For example we discussed the shortcoming in our last example that it destroyed the caller's registers, and the callers are not supposed to remember which registers are destroyed by the thousand routines they use. Using the stack the subroutine can save the caller's value of the registers on the stack, and recover them from there before returning. Meanwhile the subroutine can freely use the registers. From the caller's point of view if the registers contain the same value before and after the call, it doesn't matter if the subroutine used them meanwhile.

Similarly during the CALL operation, the current value of the instruction pointer is automatically saved on the stack, and the destination of CALL is loaded in the instruction pointer. Execution therefore resumes from the destination of CALL. When the RET instruction is executed, it recovers the value of the instruction pointer from the stack. The next instruction executed is therefore the one following the CALL. Observe how playing with the instruction pointer affects the program flow.

Apart from CALL and RET, the operations that use the stack are PUSH and POP. Two other operations that will be discussed later are INT and IRET. Regarding the stack, the operation of PUSH is similar to CALL however with a register other than the instruction pointer. For example "push ax" will push the current value of the AX register on the stack. The operation of PUSH is shown below.

SP SP – 2

[SP] AX

The operation of POP is the reverse of this. A copy of the element at the top of the stack is made in the operand, and the top of the stack is incremented afterwards. The operation of "pop ax" is shown below.

AX [SP]

SP SP + 2

SAVING AND RESTORING REGISTERS: The subroutines we wrote till now have been destroying certain registers and our calling code has been carefully written to not use those registers. However this cannot be remembered for a good number of subroutines. Therefore our subroutines need to implement some mechanism of retaining the callers' value of any registers used.

The trick is to use the PUSH and POP operations and save the callers' value on the stack and recover it from there on return. Our swap subroutine destroyed the AX register while the bubble sort subroutine destroyed AX, CX, and SI. BX was not modified in the subroutine. It had the same value at entry and at exit; it was only used by the subroutine. Our next example improves on the previous version by saving and restoring any registers that it will modify using the PUSH and POP operations.

PUSH

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

POP

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

Observe that the operand of PUSH is called a source operand since the data is moving to the stack from the operand, while the operand of POP is called destination since data is moving from the stack to the operand.

CALL

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the
procedure to transfer control back to the instruction following the CALL. For an intra segment direct CALL, SP is
decremented by two and IP is pushed onto the stack. The target procedure's relative displacement from the CALL
instruction is then added to the instruction pointer. For an inter segment direct CALL, SP is decremented by two, and CS
is pushed onto the stack. CS is replaced by the segment word contained in the instruction. SP again is decremented by
two. IP is pushed onto the stack and replaced by the offset word in the instruction. The out-of-line procedure is the
temporary division, the concept of roundabout that we discussed. Near calls are also called intra segment calls, while far
calls are called inter-segment calls. There are also versions that are called indirect calls; however they will be discuss later
when they are used.

RET

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure.
RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by
two. If RETF (inter segment RET) is used the word at the top of the stack is popped into the IP register and SP is

incremented by two. The word at the new top of stack is popped into the CS register, and SP is again incremented by two.

If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.

Lecture No: 15

PARAMETER PASSING THROUGH STACK

Due to the limited number of registers, parameter passing by registers is constrained in two ways. The maximum parameters a subroutine can receive are seven when all the general registers are used. Also, with the subroutines are themselves limited in their use of registers, and this limited increases when the subroutine has to make a nested call thereby using certain registers as its parameters. Due to this, parameter passing by registers is not expandable and generalizable. However this is the fastest mechanism available for passing parameters and is used where speed is important. Considering stack as an alternate, we observe that whatever data is placed there, it stays there, and across function calls as well. For example the bubble sort subroutine needs an array address and the count of elements. If we place both of these on the stack, and call the subroutine afterwards, it will stay there. The subroutine is invoked with its return address on top of the stack and its parameters beneath it.

To handle this using PUSH and POP, we must first pop the return address in a register, then pop the operands, and push the return address back on the stack so that RET will function normally. However so much effort doesn’t seem to pay back the price.

Recall that the default segment association of the BP register is the stack segment and the reason for this association had been deferred for now. The reason is to peek inside the stack using the BP register and read the parameters without removing them and without touching the stack pointer. The stack pointer could not be used for this purpose, as it cannot be used in an effective address. It is automatically used as a pointer and cannot be explicitly used. Also the stack pointer is a dynamic pointer and sometimes changes without telling us in the background.

When the bubble sort subroutine is called, the stack pointer is pointing to the return address. Two bytes below it is the second parameter and four bytes below is the first parameter. The stack pointer is a reference point to these parameters. If the value of SP is captured in BP, then the return address is located at [bp+0], the second parameter is at [bp+2], and the first parameter is at [bp+4]. This is because SP and BP both had the same value and they both defaulted to the same segment, the stack segment.

This copying of SP into BP is like taking a snapshot or like freezing the stack at that moment. Even if more pushes are made on the stack decrementing the stack pointer, our reference point will not change. The parameters will still be accessible at the same offsets from the base pointer. If however the stack pointer increments beyond the base pointer, the references will become invalid. The base pointer will act as the datum point to access our parameters. However we have destroyed the original value of BP in the process, and this will cause problems in nested calls where both the outer and the inner subroutines need to access their own parameters. The outer subroutine will have its base pointer destroyed after the call and will be unable to access its parameters.

To solve both of these problems, we reach at the standard way of accessing parameters on the stack. The first two instructions of any subroutines accessing its parameters from the stack are given below.

```
push bp
mov bp, sp
```

As a result our datum point has shifted by a word. Now the old value of BP will be contained in [bp] and the return address will be at [bp+2]. The second parameters will be [bp+4] while the first one will be at [bp+6]. We give an example of bubble sort subroutine using this standard way of argument passing through stack.

Stack Clearing by Caller or Callee

Parameters pushed for a subroutine are a waste after the subroutine has returned. They have to be cleared from the stack.

Either of the caller and the callee can take the responsibility of clearing them from there. If the callee has to clear the stack it cannot do this easily unless RET n exists. That is why most general processors have this instruction. Stack clearing by the caller needs an extra instruction on behalf of the caller after every call made to the subroutine, unnecessarily increasing instructions in the program. If there are thousand calls to a subroutine the code to clear the stack isrepeated a thousand times. Therefore the prevalent convention in most high level languages is stack clearing by the callee; even though the other convention is still used in some languages.

If RET n is not available, stack clearing by the callee is a complicated process. It will have to save the return address in a register, then remove the parameters, and then place back the return address so that RET will function. When this instruction was introduced in processors, only then high level language designers switched to stack clearing by the callee. This is also exactly why RET n adds n to SP after performing the operation of RET. The other way around would be totally useless for our purpose. Consider the stack condition at the time of RET and this will become clear why this will be useless. Also observe that RET n has discarded the arguments rather than popping them as they were no longer of any use either of the caller or the callee.

The strong argument in favour of callee cleared stacks is that the arguments were placed on the stack for the subroutine, the caller did not needed them for itself, so the subroutine is responsible for removing them. Removing the arguments is important as if the stack is not cleared or is partially cleared the stack will eventually become full, SP will reach 0, and thereafter wraparound producing unexpected results. This is called stack overflow. Therefore clearing anything placed on the stack is very important.

LOCAL VARIABLES: Another important role of the stack is in the creation of local variables that are only needed while the subroutine is in execution and not afterwards. They should not take permanent space like global variables. Local variables should be created when the subroutine is called and discarded afterwards. So that the spaced used by them can be reused for the local variables of another subroutine. They only have meaning inside the subroutine and no meaning outside it.

The most convenient place to store these variables is the stack. We need some special manipulation of the stack for this task. We need to produce a gap in the stack for our variables. This is explained with the help of the swapflag in the bubble sort example.

The swapflag we have declared as a word occupying space permanently is only needed by the bubble sort subroutine and

should be a local variable. Actually the variable was introduced with the intent of making it a local variable at this time.
The stack pointer will be decremented by an extra two bytes thereby producing a gap in which a word can reside. This gap
will be used for our temporary, local, or automatic variable; however we name it. We can decrement it as much as we
want producing the desired space, however the decrement must be by an even number, as the unit of stack operation is a
word. In our case we needed just one word. Also the most convenient position for this gap is immediately after saving the
value of SP in BP. So that the same base pointer can be used to access the local variables as well; this time using negative
offsets. The standard way to start a subroutine which needs to access parameters and has local variables is as under. push bp
mov bp, sp sub sp, 2

Lecture No: 16

Display Memory: The debugger gives a very close vision of the processor. That is why every program written
till now was executed inside the debugger. Also the debugger is a very useful tool in assembly language program
development, since many bugs only become visible when each instruction is independently monitored the way the
debugger allows us to do.

ASCII CODES: An ‘A’ on any computer and any operating system is an ‘A’ on every other computer and operating system. This is because a standard numeric representation of all commonly used characters has been developed. This is called the ASCII code, where ASCII stands for American Standard Code for Information Interchange. The name depicts that this is a code that allows the interchange of information; ‘A’ written on one computer will remain an ‘A’ on another. The ASCII table lists all defined characters and symbols and their standardized numbers. All ASCII based computers use the same code. There are few other standards like EBCDIC and gray codes, but ASCII has become the most prevalent standard and is used for Internet communication as well. It has become the de facto standard for global communication. The character mode displays of our computer use the ASCII standard. Some newer operating systems use a new standard Unicode but it is not relevant to us in the current discussion.

Standard ASCII has 128 characters with numbers assigned from 0 to 127. When IBM PC was introduced, they extended the standard ASCII and defined 128 more characters. Thus extending the total number of symbols from 128 to 256 numbered from 0 to 255 fitting in an 8-bit byte. The newer characters were used for line drawing, window corners, and some non-English characters. The need for these characters was never felt on teletype terminals, but with the advent of IBM PC and its full screen display, these semi-graphics characters were the need of the day. Keep in mind that at that time there was no graphics mode available.

The important thing to observe in the ASCII table is the contiguous arrangement of the uppercase alphabets (41-5A), the lowercase alphabets (61-7A), and the numbers (30-39). This helps in certain operations with ASCII, for example converting the case of characters by adding or subtracting 0x20 from it. It also helps in converting a digit into its ASCII representation by adding 0x30 to it.

DISPLAY MEMORY FORMATION: We will explore the working of the display with ASCII codes, since it is our immediately accessible hardware. When 0x40 is sent to the VGA card, it will turn pixels on and off in such a way that a visual representation of ‘A’ appears on the screen. It has no reality, just an interpretation.

The video device is seen by the computer as a memory area containing the ASCII codes that are currently displayed on the screen and a set of I/O ports controlling things like the resolution, the cursor height, and the cursor position. The VGA memory is seen by the computer just like its own memory. There is no difference; rather the computer doesn’t differentiate, as it is accessible on the same bus as the system memory. Therefore if that appropriate block of the screen is cleared, the screen will be cleared. If the ASCII of ‘A’ is placed somewhere in that block, the shape of ‘A’ will appear on the screen at a corresponding place.

This correspondence must be defined as the memory is a single dimensional space while the screen is two dimensional
having 80 rows and 25 columns. The memory is linearly mapped on this two dimensional space, just like a two
dimensional is mapped in linear memory. There is one word per character in which a byte is needed for the ASCII code
and the other byte is used for the character's attributes discussed later. Now the first 80 words will correspond to the first
row of the screen and the next 80 words will correspond to the next row. By making the memory on the video controller
accessible to the processor via the system bus, the processor is now in control of what is displayed on the screen.
The three important things that we discussed are.

- One screen location corresponds to a word in the video memory
- The video controller memory is accessible to the processor like its own memory.
- ASCII code of a character placed at a cell in the VGA memory will cause the corresponding ASCII shape to be
displayed on the corresponding screen location.

Display Memory Base Address: memory at which the video controller's memory is mapped must be a standard, so that
the program can be written in a video card independent manner. Otherwise if different vendors map their video memory at
different places in the address space, as was the problem in the start, writing software was a headache. BIOS vendors had

a problem of dealing with various card vendors. The IBM PC text mode color display is now fixed so that system software can work uniformly. It was fixed at the physical memory location of B8000. The first byte at this location contains the ASCII for the character displayed at the top left of the video screen. Dropping the zero we can load the rest in a segment register to access the video memory. If we do something in this memory, the effect can be seen on the screen. For example we can write a virus that makes any character we write drop to the bottom of the screen.

Attribute Byte

The second byte in the word designated for one screen location holds the foreground and background colors for the character. This is called its video attribute. So the pair of the ASCII code in one byte and the attribute in the second byte makes the word that corresponds to one location on the screen. The lower address contains the code while the higher one contains the attribute. The attribute byte as detailed below has the RGB for the foreground and the background. It has an intensity bit for the foreground color as well thus making 16 possible colors of the foreground and 8 possible colors for the background. When bit 7 is set the character keeps on blinking on the screen. This bit has some more interpretations like background intensity that has to be activated in the video controller through its I/O ports.

7 6 5 4 3 2 1 0

7 – Blinking of foreground character

6 – Red component of background color
5 – Green component of background color
4 – Blue component of background color
3 – Intensity component of foreground color
2 – Red component of foreground color
1 – Green component of foreground color
0 – Blue component of foreground color

Display Examples

Both DS and ES can be used to access the video memory. However we commonly keep DS for accessing our data, and
load ES with the segment of video memory. Loading a segment register with an immediate operand is not allowed in the
8088 architecture. We therefore load the segment register via a general purpose register. Other methods are loading from a
memory location and a combination of push and pop.

```
mov ax, 0xb800
mov es, ax
```

This operation has opened a window to the video memory. Now the following instruction will print an ‘A’ on the top left
of the screen in white color on black background.

```
mov word [es:0], 0x0741
```

The segment override is used since ES is pointing to the video memory. Since the first word is written to, the character
will appear at the top left of the screen. The 41 that goes in the lower byte is the ASCII code for ‘A’. The 07 that goes in

the higher byte is the attribute with I=0, R=1, G=1, B=1 for the foreground, meaning white color in low intensity and R=0, G=0, B=0 for the background meaning black color and the most significant bit cleared so that there is no blinking. Now consider the following instruction.

```
mov word [es:160], 0x1230
```

This is displayed 80 words after the start and there are 80 characters in one screen row. Therefore this is displayed on the first column of the second line. The ASCII code used is 30, which represents a ‘0’ while the attribute byte is 12 meaning green color on blue background.

Lecture No: 17

HELLO WORLD IN ASSEMBLY LANGUAGE: To declare a character in assembly language, we store its ASCII code in a byte. The assembler provides us with another syntax that doesn’t forces us to remember the ASCII code. The assembler also provides a syntax that simplifies declaration of consecutive characters, usually called a string. The three ways used below are identical in their meaning.

```
db 0x61, 0x62, 0x63
db 'a', 'b', 'c'
db 'abc'
```

When characters are stored in any high level or low level language the actual thing stored in a byte is their ASCII code.

The only thing the language helps in is a simplified declaration.

NUMBER PRINTING IN ASSEMBLY: Another problem related to the display is printing numbers. Every high level language allows some simple way to print numbers on the screen. As we have seen, everything on the screen is a pair of ASCII code and its attribute and a number is a raw binary number and not a collection of ASCII codes. For example a 10 is stored as a 10 and not as the ASCII code of 1 followed by the ASCII code of 0. If this 10 is stored in a screen location, the output will be meaningless, as the character associate to ASCII code 10 will be shown on the screen. So there is a process that converts a number in its ASCII representation. This process works for any number in any base.

Number Printing Algorithm: The key idea is to divide the number by the base number, 10 in the case of decimal. The remainder can be from 0-9 and is the right most digit of the original number. The remaining digits fall in the quotient. The remainder can be easily converted into its ASCII equivalent and printed on the screen. The other digits can be printed in a similar manner by dividing the quotient again by 10 to separate the next digit and so on. However the problem with this approach is that the first digit printed is the right most one. For example 253 will be printed as 352. The remainder after first division was 3, after second division was 5 and after the third division was 2. We have to somehow correct the order so that the actual number 253 is displayed, and the trick is to use the stack since the stack is a Last In First Out structure so if 3, 5, and 2 are pushed on it, 2, 5, and 3 will come out in this order. The steps of our algorithm are outlined below.

- Divide the number by base (10 in case of decimal)
- The remainder is its right most digit
- Convert the digit to its ASCII representation (Add 0x30 to the remainder in case of decimal)
- Save this digit on stack
- If the quotient is non-zero repeat the whole process to get the next digit, otherwise stop
- Pop digits one by one and print on screen left to right

DIV Instruction: The division used in the process is integer division and not floating point division. Integer division gives an integer quotient and an integer remainder. A division algorithm is now needed. Fortunately or unfortunately there is a DIV instruction available in the 8088 processor. There are two forms of the DIV instruction. The first form divides a 32bit number in DX:AX by its 16bit operand and stores the 16bit quotient in AX and the 16bit remainder in DX. The second form divides a 16bit number in AX by its 8bit operand and stores the 8bit quotient in AL and the 8bit remainder in AH. For example “DIV BL” has an 8bit operand, so the implied dividend is 16bit and is stored in the AX register and “DIV BX” has a 16bit operand, so the implied dividend is 32bit and is therefore stored in the concatenation of the DX and AX registers. The higher word is stored in DX and the lower word in AX.

If a large number is divided by a very small number it is possible that the quotient is larger than the space provided for it in the implied destination. In this case an interrupt is automatically generated and the program is usually terminated as a result. This is called a divide overflow error; just like the calculator shows an –E– when the result cannot be displayed. This interrupt will be discussed later in the discussion of interrupts. DIV (divide) performs an unsigned division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the two-byte dividend assumed to be in registers AL and AH. The byte quotient is returned in AL, and the byte remainder is returned in AH. If the source operand is a word, it is divided into the two-word dividend in registers AX and DX. The word quotient is returned in AX, and the word remainder is returned in DX. If the quotient exceeds the capacity of its destination register (FF for byte source, FFFF for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined.

Lecture No: 18

SCREEN LOCATION CALCULATION: For mapping from the two dimensional coordinate system of the screen to the one dimensional memory, we need to multiply the row number by 80 since there are 80 columns per row and add the column number to it and again multiply by two since there are 2 bytes for each character.

For this purpose the multiplication routine written previously can be used. However we introduce an instruction of
the 8088 microprocessor at this time that can multiply 8bit or 16bit numbers.

MUL Instruction

MUL (multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source operand is
a byte, then it is multiplied by register AL and the double-length result is returned in AH and AL. If the source operand is
a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX.
Push and pop operations always operate on words; however data can be read as a word or as a byte. For example
we read the lower byte of the parameter y-position in this case.

STRING PROCESSING

In this chapter we will discuss a bit more powerful instructions that can process blocks of data in one go. They are
called block processing or string instructions. This is the appropriate place to discuss these instructions as we have just
introduced a block of memory, which is the video memory. The vision of this memory for the processor is just a block of
memory starting at a special address. For example the clear screen operation initializes this whole block to 0741.
There are just 5 block processing instructions in 8088. In the primitive form, the instructions themselves operate

on a single cell of memory at one time. However a special prefix repeats the instruction in hardware called the REP prefix.

The REP prefix allows these instructions to operate on a number of data elements in one instruction. This is not like a loop; rather this repetition is hard coded in the processor. The five instructions are STOS, LODS, CMPS, SCAS, and MOVS called store string, load string, compare string, scan string, and move string respectively. MOVS is the instruction that allows memory to memory moves, as was discussed in the exceptions to the memory to memory movement rules.

String instructions are complex instruction in that they perform a number of tasks against one instruction. And with the REP prefix they perform the task of a complex loop in one instruction. This causes drastic speed improvements in operations on large blocks of memory. The reduction in code size and the improvement in speed are the two reasons why these instructions were introduced in the 8088 processor.

There are a number of common things in these instructions. Firstly they all work on a block of data. DI and SI are used to access memory. SI and DI are called source index and destination index because of string instructions. Whenever an instruction needs a memory source, DS:SI holds the pointer to it. An override is possible that can change the association from DS but the default is DS. Whenever a string instruction needs a memory destination, ES:DI holds the pointer to it. No override is possible in this case. Whenever a byte register is needed, AL holds the value. Whenever a word register is used AX holds the value. For example STOS stores a register in memory so AL or AX is the register used and ES:DI points to the destination. The LODS instruction loads from memory to register so the source is pointed to by DS:SI and the register used is AL or AX.

String instructions work on a block of data. A block has a start and an end. The instructions can work from the start towards the end and from the end towards the start. In fact they can work in both directions, and they must be allowed to work in both directions otherwise certain operations with overlapping blocks become impossible. This problem is discussed in detail later. The direction of movement is controlled with the Direction Flag (DF) in the flags register. If this flag is cleared the direction is from lower addresses towards higher addresses and if this flag is set the direction is from higher addresses to lower addresses. If DF is cleared, this is called the auto-increment mode of string instruction, and if DF is set, this is called the auto-decrement mode. There are two instructions to set and clear the direction flag.

cld ; clear direction flag
std ; set direction flag

Every string instruction has two variants; a byte variant and a word variant. For example the two variants of

STOS are STOSB and STOSW. Similarly the variants for the other string instructions are attained by appending a B or a W to the instruction name. The operation of each of the string instructions and each of the repetition prefixes is discussed below.

STOS

STOS transfers a byte or word from register AL or AX to the string element addressed by ES:DI and updates DI to point to the next location. STOS is often used to clear a block of memory or fill it with a constant.

The implied source will always be in AL or AX. If DF is clear, SI will be incremented by one or two depending of whether STOSB or STOSW is used. If DF is set SI will be decremented by one or two depending of whether STOSB or STOSW is used. If REP is used before this instruction, the process will be repeated CX times. CX is called the counter register because of the special treatment given to it in the LOOP and JCXZ instructions and the REP set of prefixes. So if REP is used with STOS the whole block of memory will be filled with a constant value. REP will always decrement CX like the LOOP instruction and this cannot be changed with the direction flag. It is also independent of whether the byte or the word variant is used. It always decrements by one; therefore CX has count of repetitions and not the count of bytes.

LODS: LODS transfers a byte or word from the source location DS:SI to AL or AX and updates SI to point to the next location. LODS is generally used in a loop and not with the REP prefix since the value previously loaded in the register is overwritten if the instruction is repeated and only the last value of the block remains in the register.

SCAS

SCAS compares a source byte or word in register AL or AX with the destination string element addressed by ES:DI and updates the flags. DI is updated to point to the next location. SCAS is often used to locate equality or in-equality in a string through the use of an appropriate prefix.

SCAS is a bit different from the other instructions. This is more like the CMP instruction in that it does subtraction of its operands. The prefixes REPE (repeat while equal) and REPNE (repeat while not equal) are used with this instruction. The instruction is used to locate a byte in AL in the block of memory. When the first equality or inequality is encountered; both have uses. For example this instruction can be used to search for a 0 in a null terminated string to calculate the length of the string. In this form REPNE will be used to repeat while the null is not there.

MOVS

MOVS transfers a byte or word from the source location DS:SI to the destination ES:DI and updates SI and DI to point to the next locations. MOVS is used to move a block of memory. The DF is important in the case of overlapping blocks. For example when the source and destination blocks overlap and the source is below the destination copy must be done upwards while if the destination is below the source copy must be done downwards. We cannot perform both these copy operations properly if the direction flag was not provided. If the source is below the destination and an upwards copy is used the source to be copied is destroyed. If however the copy is done downwards the portion of source destroyed is the one that has already been copied. Therefore we need the control of the direction flag to handle this problem. This problem is further detailed in a later example.

CMPS

CMPS subtracts the source location DS:SI from the destination location ES:DI. Source and Destination are unaffected. SI and DI are updated accordingly. CMPS compares two blocks of memory for equality or inequality of the block. It subtracts byte by byte or word by word. If used with a REPE or a REPNE prefix is repeats as long as the blocks are same or as long as they are different. For example it can be used for find a substring. A substring is a string that is contained in another string. For example “has” is contained in “Mary has a little lamp.” Using CMPS we can do the operation of a complex loop in a single instruction. Only the REPE and REPNE prefixes are meaningful with this instruction.

Lecture No: 19

REP Prefix

REP repeats the following string instruction CX times. The use of CX is implied with the REP prefix. The decrement in CX doesn’t affect any flags and the jump is also independent of the flags, just like JCXZ.

REPE and REPNE Prefixes

REPE or REPZ repeat the following string instruction while the zero flag is set and REPNE or REPNZ repeat the following instruction while the zero flag is not set. REPE or REPNE are used with the SCAS or CMPS instructions. The other string instructions have nothing to do with the condition since they are performing no comparison. Also the initial state of flags before the string instruction does not affect the operation. The most complex operation of the string instruction is with these prefixes.

A space efficient way to zero a 16bit register is to XOR it with itself. Remember that exclusive or results in a zero whenever the bits at the source and at the destination are same. This instruction takes just two bytes compared to “mov di, 0” which would take three. This is a standard way to zero a 16bit register.

Inside the debugger the operation of the string instruction can be monitored. The trace into command can be used to monitor every repetition of the string instruction. However screen will not be cleared inside the debugger as the debugger overwrites its display on the screen. CX decrements with every iteration, DI increments by 2. The first access is made at

B800:0000 and the second at B800:0002 and so on. A complex and inefficient loop is replaced with a fast and simple
instruction that does the same operation many times faster.

LODS Example – String Printing

The use of LODS with the REP prefix is not meaningful as only the last value loaded will remain in the register. It is
normally used in a loop paired with a STOS instruction to do some block processing. We use LODS to pick the data, do
the processing, and then use STOS to put it back or at some other place. For example in string printing, we will use LODS
to read a character of the string, attach the attribute byte to it, and use STOS to write it on the video memory.

SCAS Example – String Length

Many higher level languages do not explicitly store string length; rather they use a null character, a character with an
ASCII code of zero, to signal the end of a string. In assembly language programs, it is also easier to store a zero at the end
of the string, instead of calculating the length of string, which is very difficult process for longer strings. So we delegate
length calculation to the processor and modify our string printing subroutine to take a null terminated string and no length.
We use SCASB with REPNE and a zero in AL to find a zero byte in the string. In CX we load the maximum possible size,
which is 64K bytes. However actual strings will be much smaller. An important thing regarding SCAS and CMPS is that

if they stop due to equality or inequality, the index registers have already incremented. Therefore when SCAS will stop DI would be pointing past the null character.

Lecture No: 20

LES and LDS Instructions

Since the string instructions need their source and destination in the form of a segment offset pair, there are two special instructions that load a segment register and a general purpose register from two consecutive memory locations. LES loads ES while LDS loads DS. Both these instructions have two parameters, one is the general purpose register to be loaded and the other is the memory location from which to load these registers. The major application of these instructions is when a subroutine receives a segment offset pair as an argument and the pair is to be loaded in a segment and an offset register. According to Intel rules of significance the word at higher address is loaded in the segment register while the word at lower address is loaded in the offset register. As parameters segment should be pushed first so that it ends up at a higher address and the offset should be pushed afterwards. When loading the lower address will be given. For example "lds si, [bp+4]" will load SI from BP+4 and DS from BP+6.

MOVS EXAMPLE – SCREEN SCROLLING

MOVS has the two forms MOVSB and MOVSW. REP allows the instruction to be repeated CX times allowing blocks of

memory to be copied. We will perform this copy of the video screen.
Scrolling is the process when all the lines on the screen move one or more lines towards the top of towards the
bottom and the new line that appears on the top or the bottom is cleared. Scrolling is a process on which string movement
is naturally applicable. REP with MOVS will utilize the full processor power to do the scrolling in minimum time.

CMPS EXAMPLE – STRING COMPARISON

For the last string instruction, we take string comparison as an example. The subroutine will take two segment
offset pairs containing the address of the two null terminated strings. The subroutine will return 0 if the strings
are different and 1 if they are same. The AX register will be used to hold the return value.

Lecture No: 21

Interrupts: Interrupts in reality are events that occurre outside the processor and the processor must be
informed about them. Interrupts are asynchronous and unpredictable. Asynchronous means that the interrupts
occur, independent of the working of the processor, i.e. independent of the instruction currently executing.
Synchronous events are those that occur side by side with another activity. Interrupts must be asynchronous as
they are generated by the external world which is unaware of the happenings inside the processor.
True

interrupts that occur in real time are asynchronous with the execution. Also it is unpredictable at which time an interrupt will come. The two concepts of being unpredictable and asynchronous are overlapping. Unpredictable means the time at which an interrupt will come cannot be predicted, while asynchronous means that the interrupt has nothing to do with the currently executing instruction and the current state of the processor.

The 8088 processor divides interrupts into two classes. Software interrupts and hardware interrupts. Hardware interrupts are the real interrupts generated by the external world as discussed above. Software interrupts on the contrary are not generated from outside the processor. They just provide an extended far call mechanism. Far all allows us to jump anywhere in the whole megabyte of memory. To return from the target we place both the segment and offset on the stack.

Software interrupts show a similar behavior. It however pushes one more thing before both the segment and offset and that is the FLAGS register. Just like the far call loads new values in CS and IP, the interrupt call loads new values in CS, IP, and FLAGS. Therefore the only way to retain the value of original FLAGS register is to push and pop as part of interrupt call and return instructions. Pushing and popping inside the routine will not work as the routine started with an already tampered value.

The discussion of real time interrupts is deferred till the next chapter. They play the critical part in control applications where external hardware must be control and events and changes therein must be appropriately responded by the processor. To generate an interrupt the INT instruction is used. The routine that executes in response to an INT instruction is called the interrupt service routine (ISR) or the interrupt handler. Taking example from real time interrupts the routine to instruct an external hardware to close the valve of a boiler in response to an interrupt from the pressure sensor is an interrupt routine.

The software interrupt mechanism in 8088 uses vectored interrupts meaning that the address of the interrupt routine is not directly mentioned in an interrupt call, rather the address is lookup up from a table. 8088 provides a mechanism for mapping interrupts to interrupt handlers. Introducing a new entry in this mapping table is called hooking an interrupt.

Syntax of the INT instruction is very simple. It takes a single byte argument varying from 0-255. This is the interrupt number informing the processor, which interrupt is currently of interest. This number correlates to the interrupt handler routine by a routing or vectoring mechanism. A few interrupt numbers in the start are reserved and we generally do not use them. They are related to the processor working. For example INT 0 is the divide by zero interrupt. A list of all reserved interrupts is given later. Such interrupts are programmed in the hardware to generate the designated interrupt when the specified condition arises. The remaining interrupts are provided by the processor for our use. Some of these were reserved by the IBM PC designers to interface user programs with system software like DOS and BIOS. This was the logical choice for them as interrupts provided a very flexible architecture. The remaining interrupts are totally free for use in user software. The correlation process from the interrupt number to the interrupt handler uses a table called interrupt vector table. Its loation is fixed to physical memory address zero. Each entry of the table is four bytes long containing the segment and offset of the interrupt routine for the corresponding interrupt number. The first two bytes in the entry contain the offset and the next two bytes contain the segment. The little endian rule of putting the more significant part (segment) at a higher address is seen here as well. Mathematically offset of the interrupt will be at nx4 while the segment will be at nx4+2. One entry in this table is called a vector. If the vector is changed for interrupt 0 then INT 0 will take execution to the new handler whose address is now placed at those four bytes. INT 1 vector occupies location 4, 5, 6, and 7 and similarly vector for INT 2 occupies locations 8, 9, 10, and 11. As the table is located in RAM it

can be changed anytime. Immediately after changing it the interrupt mapping is changed and now the interrupt will result
in execution of the new routine. This indirection gives the mechanism extreme flexibility.
The operation of interrupt is same whether it is the result of an INT instruction (software interrupt) or it is generated by an
external hardware which passes the interrupt number by a different mechanism. The currently executing instruction is
completed, the current value of FLAGS is pushed on the stack, then the current code segment is pushed, then the offset of
the next instruction is pushed. After this it automatically clears the trap flag and the interrupt flag to disallow further
interrupts until the current routine finishes. After this it loads the word at nx4 in IP and the word at nx4+2 in CS if
interrupt n was generated. As soon as these values are loaded in CS and IP execution goes to the start of the interrupt
handler. When the handler finishes its work it uses the IRET instruction to return to the caller. IRET pops IP, then CS, and
then FLAGS. The original value of IF and TF is restored which reenables further interrupts. IF and TF will be discussed in
detail in the discussion of real time interrupts. We have discussed three things till now.
1. The INT and IRET instruction format and syntax
2. The formation of IVT (interrupt vector table)
3. Operation of the processor when an interrupt in generated
Just as discussed in the subroutines chapter, the processor will not match interrupt calls to interrupt returns. If a RETF is

used in the end of an ISR the processor will still return to the caller but the FLAGS will remain on the stack which will destroy the expectations of the caller with the stack. If we know what we are doing we may use such different combination of instructions. Generally we will use IRET to return from an interrupt routine. Apart from indirection the software interrupt mechanism is similar to CALL and RET. Indirection is the major difference.

The operation of INT can be written as:

- sp ← sp+2
- [sp] ← flag
- sp ← sp+2
- if ← 0
- tf ← 0
- [sp] ← cs
- sp ← sp+2
- [sp] ← ip
- ip ← [0:N*4]
- cs ← [0:N*4+2]

The operation of IRET can be written as:

- ip ← [sp]
- sp ← sp-2
- cs ← [sp]
- sp ← sp-2
- flag ← [sp]
- sp ← sp-2

The above is the microcode description of INT and IRET. To obey an assembly language instruction the processor breaks it down into small opertions. By reading the microcode of an instruction its working can be completely understood.

The interrupt mechanism we have studied is an extended far call mechanism. It pushes FLAGS in addition to CS and IP and it loads CS and IP with a special mechanism of indirection. It is just like the table of contents that is located at a fixed position and allows going directly to chapter 3, to chapter 4 etc. If this association is changed in the table of contents the direction of the reader changes. For example if Chapter 2 starts at page 220 while 240 is written in the table of contents, the reader will go to page 240 and not 220. The table of contents entry is a vector to point to map the chapter number to page number. IVT has 256 chapters and the interrupt mechanism looks up the appropriate chapter number to reach the desired page to find the interrupt routine.

Another important similarlity is that table of contents is always placed at the start of the book, a well known place. Its physical position is fixed. If some publishers put it at some place, others at another place, the reader will be unable to find the desired chapter. Similarly in 8088 the physical memory address zero is fixed for the IVT and it occupies exactly a kilobyte of memory as the 256x4=1K where 256 is the number of possible interrupt vectors while the size of one vector is

4 bytes.

Lecture No: 22

Interrupts introduce temporary breakage in the program flow, sometimes programmed (software interrupts) and
unprogrammed at other times (hardware interrupts). By hooking interrupts various system functionalities can be
controlled. The interrupts reserved by the processor and having special functions in 8088 are listed below:

· INT 0, Division by zero

Meaning the quotient did not fit in the destination register. This is a bit different as this interrupt does not return to
the next instruction, rather it returns to the same instruction that generated it, a DIV instruction ofcourse. Here INT
0 is automatically generated by a DIV when a specific situation arises, there is no INT 0 instruction.

· INT 1, Trap, Single step Interrupt

This interrupt is used in debugging with the trap flag. If the trap flag is set the Single Step Interrupt is generated
after every instruction. By hooking this interrupt a debugger can get control after every instruction and display the
registers etc. 8088 was the first processor that has this ability to support debugging.

· INT 2, NMI-Non Maskable Interrupt

Real interrupts come from outside the processor. INT 0 is not real as it is generated from inside. For real interrupts
there are two pins in the processor, the INT pin and the NMI pin. The processor can be directed to listen or not to

listen to the INT pin. Consider a recording studio, when the recording is going on, doors are closed so that no interruption occurs, and when there is a break, the doors are opened so that if someone is waiting outside can come it. However if there is an urgency like fire outside then the door must be broken and the recording must not be catered for. For such situations is the NMI pin which informs about fatal hardware failures in the system and is tied to interrupt 2. INT pin can be masked but NMI cannot be masked.

INT 3, Debug Interrupt

The only special thing about this interrupt is that it has a single byte opcode and not a two byte combination where the second byte tells the interrupt number. This allows it to replace any instruction whatsoever. It is also used by the debugger and will be discussed in detail with the debugger working.

· INT 4, Arithmetic Overflow, change of sign bit

The overflow flag is set if the sign bit unexpectedly changes as a result of a mathemcial or logical instruction.

However the overflow flag signals a real overflow only if the numbers in question are treated as signed numbers.

So this interrupt is not automatically generated but as a result of a special instruction INTO (interrupt on overflow) if the overflow flag is set. Otherwise the INTO instruction behaves like a NOP (no operation).

These are the five interrupts reserved by Intel and are generally not used in our operations.