Maynooth University
National University of Ireland Maynooth

CS433 Modern Architectures

Video 3

Assembly language

Created for streaming using Panopto within MU Moodle only.

Topic 1.8: Introduction x64 Assembly Language

CD FC 59 C8

$B = A^2$

Assembly Language: is any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions.

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

Address	Machine Code	Assembly Language	
00007FF7408D1D84	C4 A1 7C 10 04 81	vmovups	ymm0,ymmword ptr [rcx+r8*4]
00007FF7408D1D8A	C5 FC 59 C8	vmulps	ymm1,ymm0,ymm0
00007FF7408D1D8E	C5 F4 59 D0	vmulps	ymm2,ymm1,ymm0
00007FF7408D1D92	C5 FC 58 C1	vaddps	ymm0,ymm0,ymm1
00007FF7408D1D96	C5 FC 58 C2	vaddps	ymm0,ymm0,ymm2
00007FF7408D1D9A	C4 A1 7C 11 04 82	vmovups	ymmword ptr [rdx+r8*4],ymm0
00007FF7408D1DA0	49 83 E8 08	sub	r8,8
00007FF7408D1DA4	7D DE	jge	loop1 (07FF7408D1D84h)
00007FF7408D1DA6	C5 FC 77	vzeroall	
00007FF7408D1DA9	C9	leave	
00007FF7408D1DAA	C3	ret	

Almost 1:1 mapping

Why look at Assembly Language?

Assembly language is instructive for those wishing to understand the operation of hardware. It represents the actual instructions that are being executed by the microprocessor during run-time.

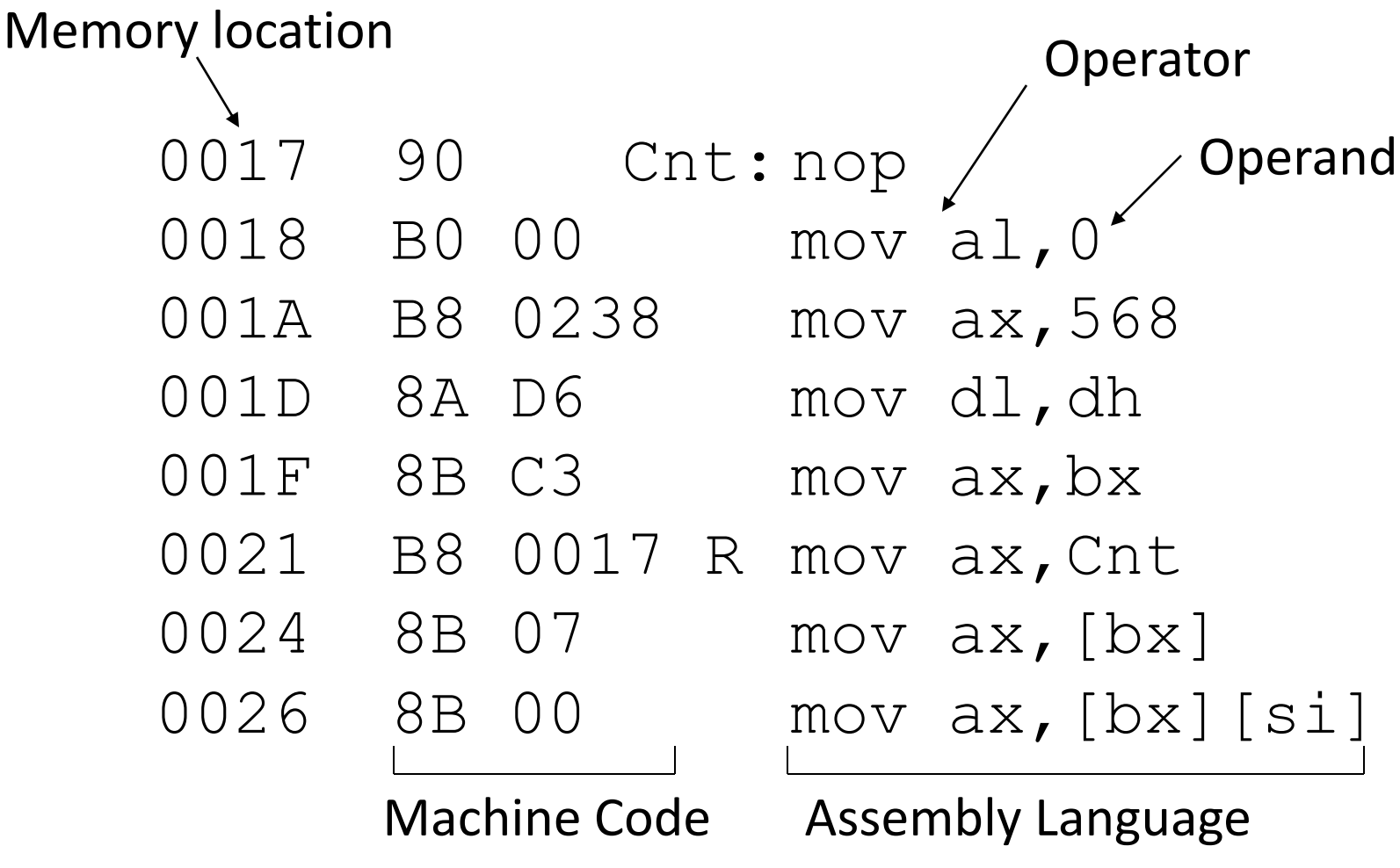
High level compilers generate machine language represented by assembly codes.

Well written assembly language programs can be very fast. There is no overhead introduced by a high level compiler or interpreter.

Programs can be written that are very efficient on memory (e.g. Hello world, 20-30bytes).

Summary: Fast, Uses small amounts of memory, but unsuitable for complex problems.

A Compiled Assembly Language Program



Assembly language, Byte code – Real processors and virtual machines

```
#include "stdafx.h"
using namespace System;
int main(array<System::String ^> ^args)
{
  unsigned int x=123;
  unsigned int y=456;
  unsigned int z;
  z=x+y;
}
```

Compile and run

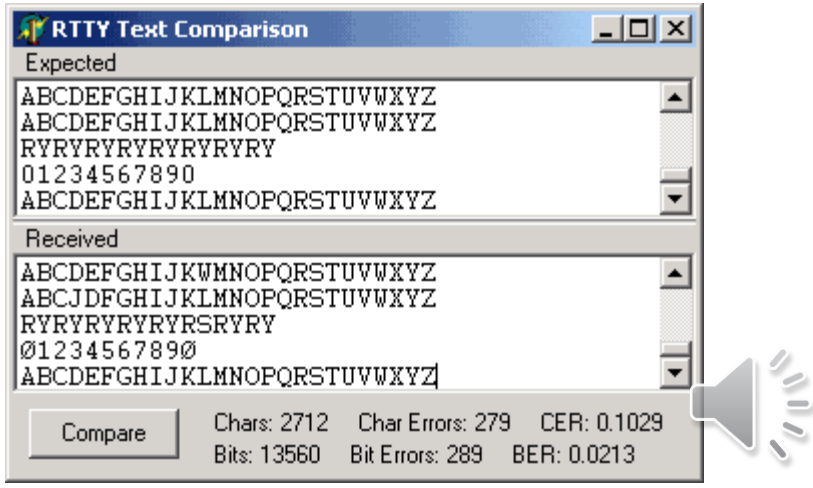
```
; 9    :        unsigned int x=123;

  0000b 16               ldc.i.0 0      ; i32 0x0
  0000c 0a               stloc.0        ; $T8928
  0000d 1f 7b            ldc.i4.s 123   ; u32 0x7b
  0000f 0c               stloc.2        ; _x$

; 10   :        unsigned int y=456;

  00010 20 c8 01 00 00   ldc.i4 456     ; u32 0x1c8
  00015 0b               stloc.1        ; _y$

; 11   :        unsigned int z;
; 12   :        z=x*y;

  00016 08               ldloc.2        ; _x$
  00017 07               ldloc.1        ; _y$
  00018 5a               mul
  00019 0d               stloc.3        ; _z$
```

Note: This is not strictly assembly language it is byte code (Common Interface Language) running on a virtual machine called the common library runtime CLR. This is how windows runs the .NET framework.

Amateur radio – my introduction assembly language

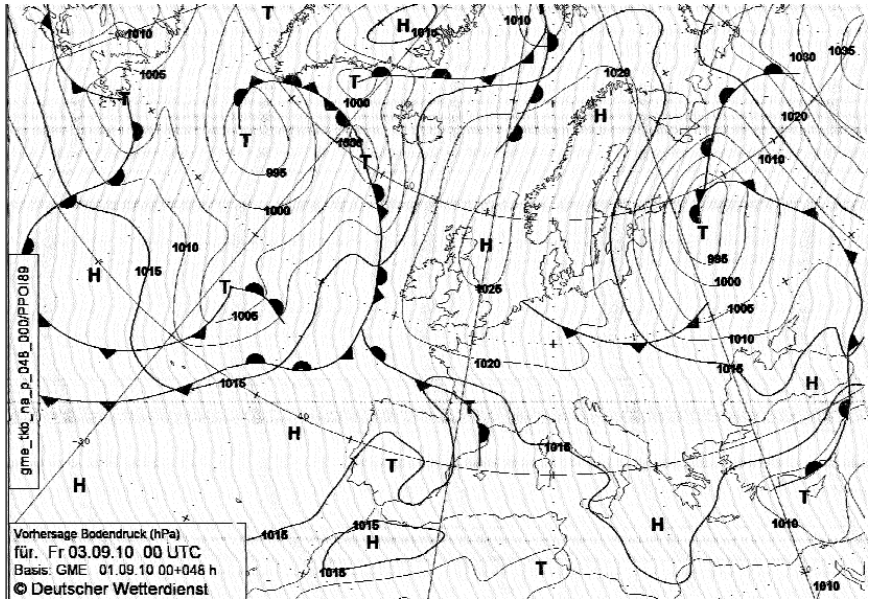
Nr.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Buchstabenreihe	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	<	≡	Bu	Zi	ZWR	
Zeichenreihe	-	?	:	Wer da?	3				8	Klingel	(	)	.	,	9	0	1	4	'	5	7	=	2	/	6	+						
Anlaufschritt																																
5er Schrittgruppe 1	●	●		●	●	●				●	●						●		●		●		●	●	●	●			●	●		
5er Schrittgruppe 2	●		●				●		●	●	●	●				●	●	●			●	●	●					●	●	●		
5er Schrittgruppe 3			●			●		●	●		●		●	●		●	●		●		●	●		●	●				●		●	
5er Schrittgruppe 4		●	●	●		●	●			●	●		●	●	●			●				●		●		●		●	●			
5er Schrittgruppe 5		●					●	●				●	●			●	●			●		●	●	●	●	●			●	●		
Sperrschritt 1½ fach	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

□ Pausenschritt
● Stromschritt
< Wagenrücklauf
≡ Zeilenvorschub
Wer da?
Frei für den internen Betrieb eines jeden Landes, aber im zwischenstaatl. Verkehr nicht zugelassen.
Klingel
Zi Ziffernumschaltung
Bu Buchstabenumschaltung
ZWR Zwischenraum



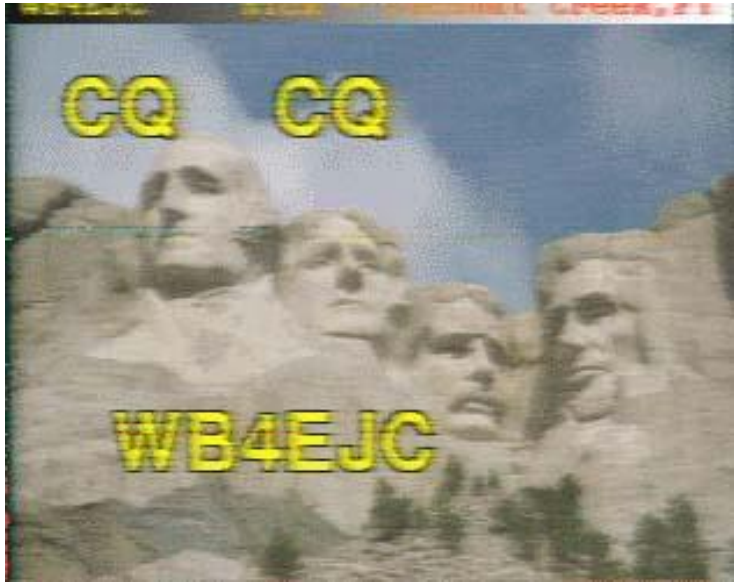
RTTY Radio-teletype

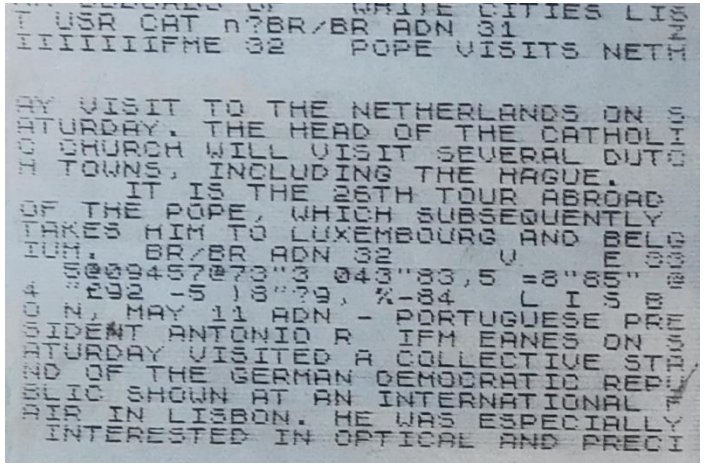
Amateur radio



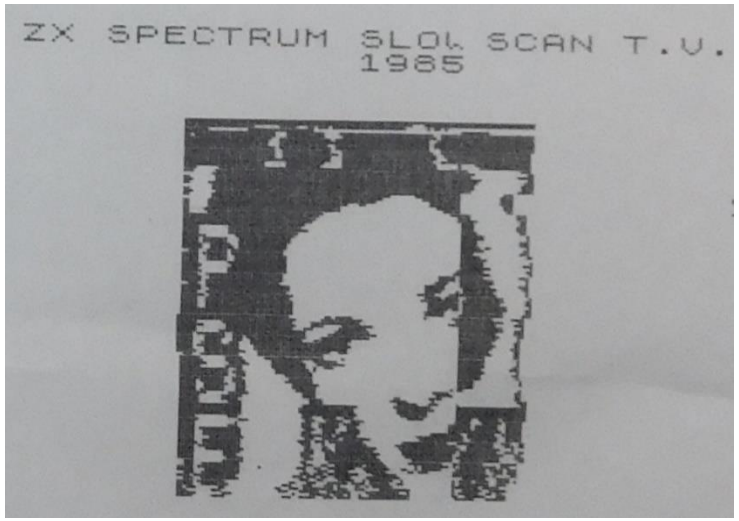
Wefax – Weather fax

Shortwave



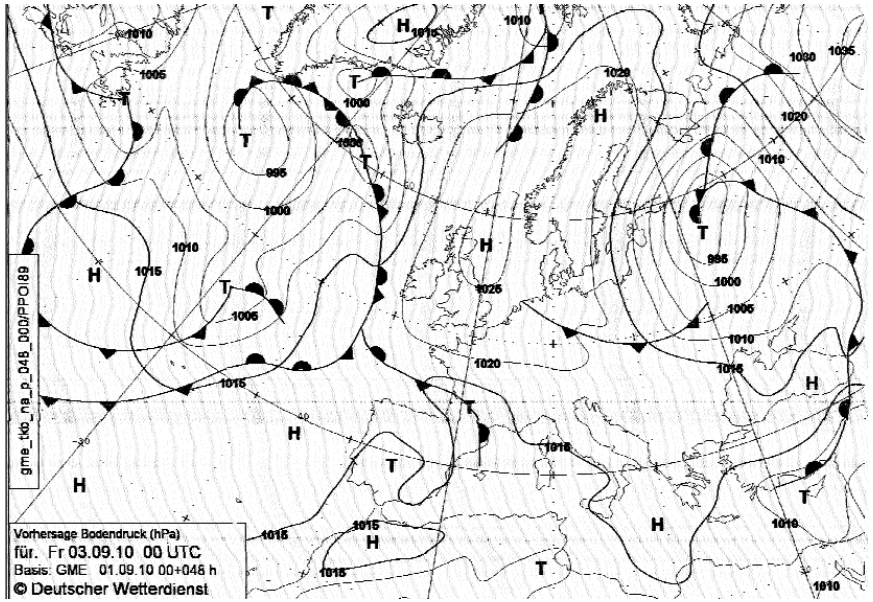


RTTY– Radio Teletype



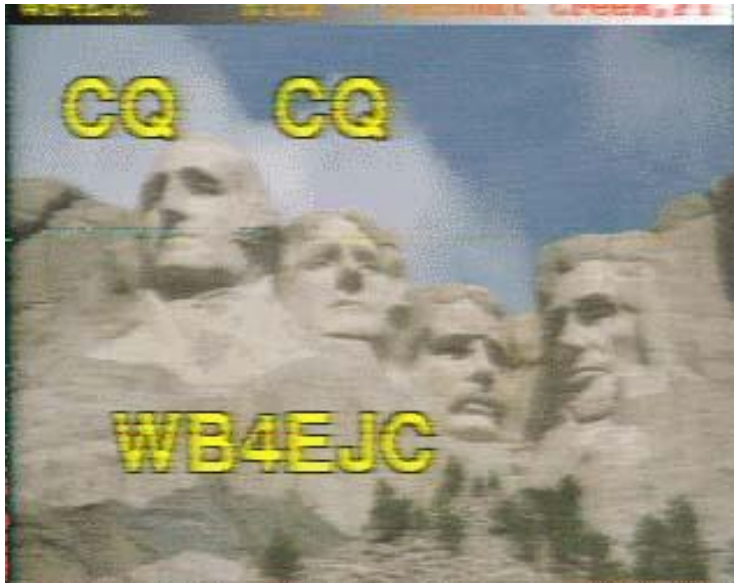
SSTV– Slow scan TV

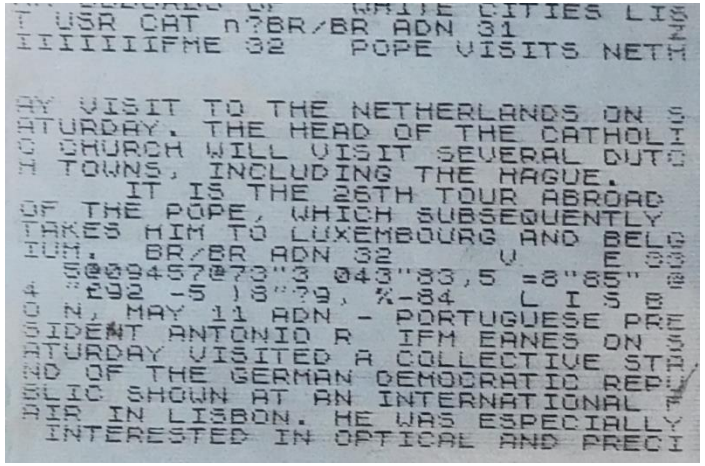
Amateur radio



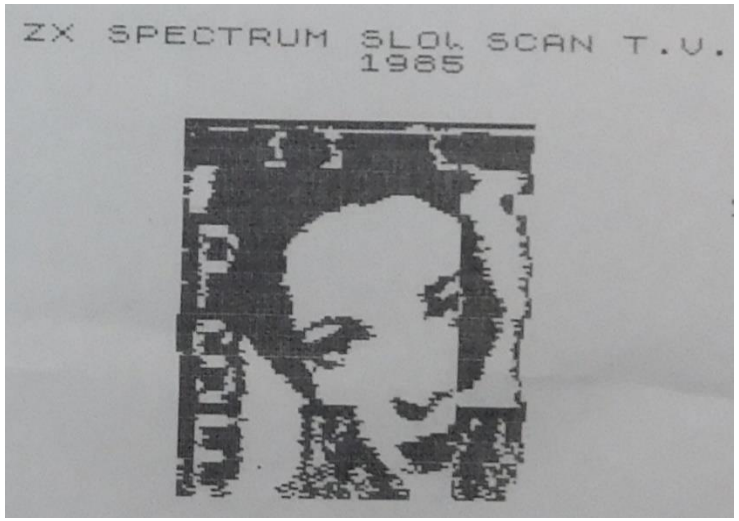
Wefax – Weather fax

Shortwave





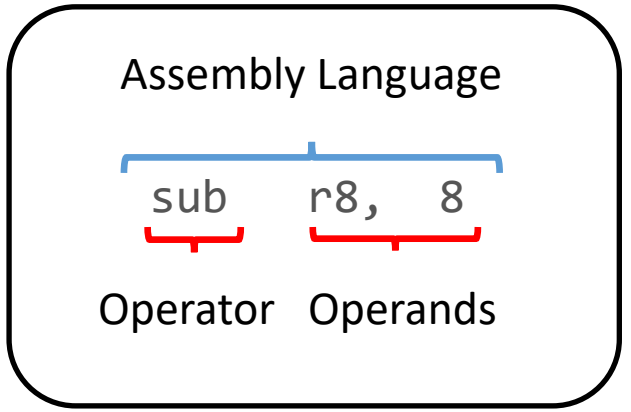
RTTY– Radio Teletype



SSTV– Slow scan TV

Operator/Instruction types

Binary Arithmetic	:	Adding and Subtracting
Decimal Arithmetic	:	Adjust binary to BCD
Logical	:	AND, OR, Shifting
Data Transfer	:	Move (mov), Load (ld)
Stack	:	Storing on the stack
Control Transfer	:	Function calls, jumps
String	:	Text arrays
Pointer	:	Stack and program counter
Input/Output	:	Reading and writing to ports
Prefix	:	Modifies all other commands
System	:	CPU configuration settings
Floating point	:	The 80837 co-processor commands
Extensions	:	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AESNI, A



General Purpose Registers

The 8086 has eight general purpose registers. As a programmer you can think of them as eight variables. AX is the accumulator and there are more instructions associated with it than any other register. Each register is eight bits wide.

However some instructions can uses registers in pairs, giving 16 bit operation.

When used as a register pair they are given the collective name AX, BX, CX, DX.

More recent processor extend the registers to 32 bit (EAX), 64 bit (RAX).

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

Very similar to the concept of unions in C.

Note registers are memory locations in the processor and are very fast to access.

The Flag Register

The 8086 keeps track of the result of certain calculations in a special 16 bit flag register.

15MSB 0LSB

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

U: Undefined

OF: Overflow flag

DF: String direction flag

IF: Interrupt enable flag

TF: Single step trap flag

SF: Sign flag MSB of result

ZF: Zero flag, set if result=0

AF: BCD Carry flag

PF: Parity flag

CF: Carry flag, checking magnitude

Conditional Flags: CF, PF, ZF, SF, OF

Control Flags: CF, PF, ZF, SF, OF

Decisions

jz pass – jump if zero flag set

Logical Instructions

AND:	Boolean AND
OR:	Boolean OR
XOR:	Exclusive OR
BT:	Bit test result in CF
BTC:	Bit test and then complement bit tested
BTR:	Bit test and then reset bit tested
BTS:	Bit test and set the bit tested
BSF:	Bit scan forward until bit set
BSR:	Bit scan reverse until bit set

MSB(15) LSB(0)
AX=00010000-00000010

BSF AX,DX

Result : DX=1

MSB(15) LSB(0)
AX=00010000-00000010

BSR AX,DX

Result : DX=3

OR Function used to convert BCD to ASCII

AL=00001001, 9 Decimal

OR AL,30H ; Quicker than add al,30h

AL=0011 1001, 57 Decimal code for '9'

ASCII

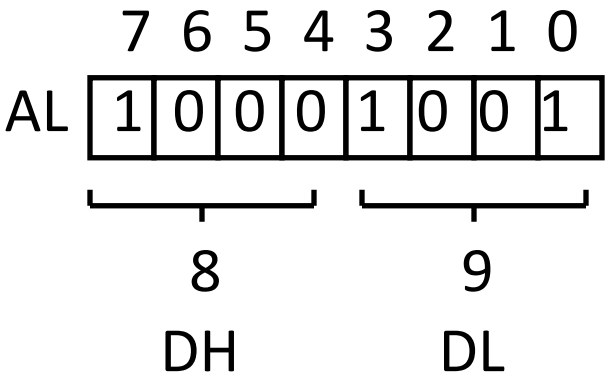
Characters are represented by a 7 bit binary code known as ASCII.

ASCII American Standard Code For Information Interchange.

Nm	Ch	Nm	Ch	Nm	Ch	Nm	Ch	Nm	Ch	Nm	Ch
32	sp	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		

Note: To convert a number in the range (0,9) from BCD (unpacked) to ASCII add 30H or 48Decimal.

AND can be used to mask bits




```
AL=10001001

MOV   DL,AL

AND   DL,00001111b

;DL Equals 9


MOV   DH,AL

AND   DH,11110000b

SHR   DH,4

;DH Equals 8
```

Logical Shift

SHL: Shift Left Logical

SHR: Shift Right Logical

SAL: Shift Arithmetic Left

SAR: Shift Arithmetic Right

ROL: Rotate left

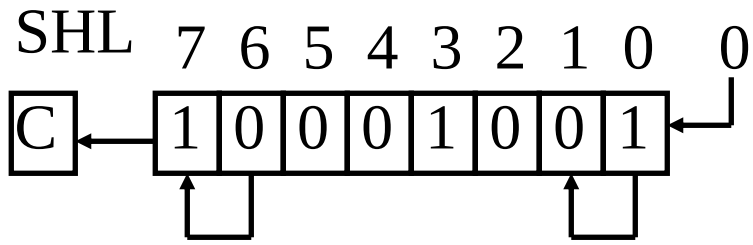
ROR: Rotate right

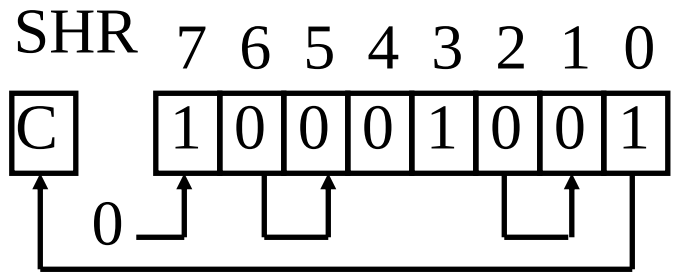
RCL: Rotate through carry left

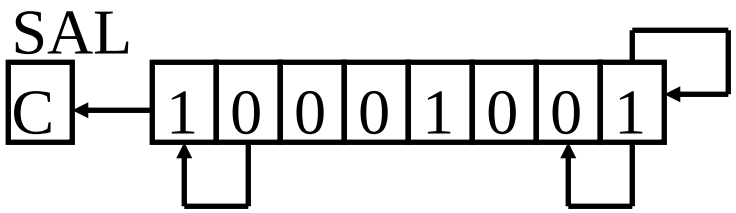
RCR: Rotate through carry right

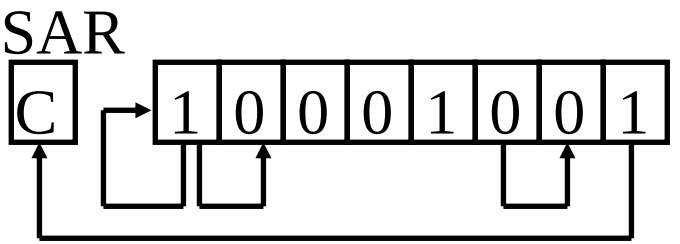
SHLD: Shift left double

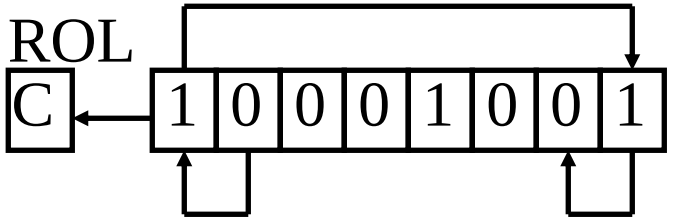
SHRD: Shift right double

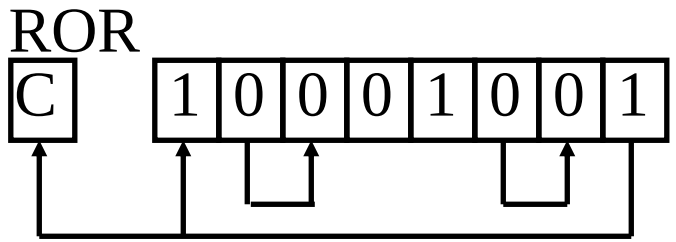


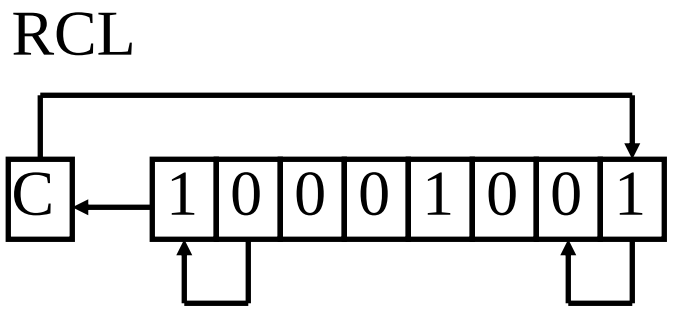


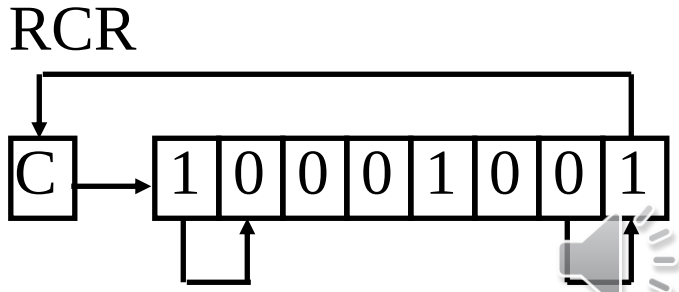












Control Transfer

Control Transfer functions affect the flow of program execution.

Normally the IP pointer incremented as each instruction is executed.

Branch Instructions: Change the IP (Instruction pointer), (e.g. jmp, jz, jc)

Call Instructions: Store the current IP on the stack then change the IP (e.g. call, ret).

Note: The instruction pointer could be considered as the line of code currently being executed. It is incremented automatically by the microprocessor after each instruction is executed so as execute the next instruction.

Branch Control Transfer Functions

Jump *offset*

JA	Jump above CF=0,ZF=0
JAE	Jump above or equal CF=0
JB	Jump below CF=1
JBE	Jump below or equal CF=1 ZF=1
JC	Jump if carry CF=1
JCXZ	Jump if CX=0
JZ	Jump if zero ZF=1
JG	Jump Greater SF=0, ZF=0
JGE	Jump greater or equal SF=0
JL	Jump less SF!=0F, ZF=1
JLE	Jump less or equal SF!=0F, ZF=1
JNA	Jump not above
JNAE	Jump not above or equal
JNB	Jump not below
JNBE	Jump not below or equal
JCXZ	Jump if CX=0
JNC	Jump if no carry CF=0
JNE	Jump not zero ZF=1
JNG	Jump not greater SF!=0, ZF=1
JNGE	Jump not greater or equal (JL)

Conditional jumps are short jumps, the operand is a single byte that allows a jump back of -128 or forward of +127. Thus you can't jump very far using conditional jumps. Offsets for jumps are counted from the byte after the jump.

The unconditional jump

The unconditional jump can jump much bigger distances.

The compiler will used code with an offset address for small jumps (-128,+127).

A direct address jump is possible allowing a jump of +/-32K

Inter segment jumps are possible.

```
JMP     Label

JMP     Label.seg
```

JMP AX where — Jumps using registers are allowed (be careful you jump!, this needs checking!!!!)

Branch Control Transfer Functions

These jumps normally follow a CMP command

Compare cmp

cmp ax,bx evaluate ax-bx, set the flags but no change to ax or bx

```
        mov ax,20
        mov bx,10
        cmp ax,bx
        jnc       over
        ;print bx greater than or equal to ax
        jmp  pass
over    ;print bx less   than ax
pass    ;next bit of code
```

No Carry

Carry

Compiled Branch Code

If Al<10, print C
AL=10 or above print NC

```
0017  B4 02 start:      mov ah,02h    ; ax=02 Print
0019        3C 0A       cmp al,10
001B        72 04       jc  lab1
001D        B2 4E       mov dl,'N'
001F        CD 21       int 021h;
0021        B2 43 lab1: mov dl,'C'
0023        CD 21       int 021h
0025        EB 04       jmp quit
0027        90          nop
0028        90          nop
0029        74 F6       jz  lab1
002B        90    quit: nop
```

+4 Bytes

-9 Bytes
255-9=246, F6h

Conditional jumps are short jumps, the operand is a single byte that allows a jump back of -128 or forward of +127.

You can not jump very far using conditional jumps.

Offsets for jumps are counted from the byte after the jump. Jump 1, jump 2….

Forwards: 0 1 2 3 4 5 6 7 8 9
Backwards: FF, FE, FD, FC FB, FA F9 ,F8, F7, F6

Requires DOS-Box to access int 21h

LOOPS

The for(x=0; x<3; x++) of the assembly world

LOOP	Loop if CX not zero
LOOPNZ	Loop while not equal (ZF=0) and CX not zero
LOOPZ	Loop while zero (ZF=1) and CX not zero

On Z80A this command is djnz (decrease and jump if not zero)

REP Modifier

Using loops as delays

```
        MOV CX, N          ; 4      C0=4
WASTE:  NOP                ; 3
        NOP                ; 3      CBK=23
        LOOP WASTE; 17^ or 5V
```

Clock cycles required to complete the instruction.

$$C_T = C_0 + N(C_{BK}) - C_{CR}$$

C_T is the desired time delay in clock cycles,

C_0 is the overhead

N is the number of times to go around the loop

C_{CR} =-12 since 17-5=12 cycles are saved last time through.

Calculating the delay

On a 1GHz machine $T_{Clock}=10^{-9}S$

To create a delay of T_{Delay} =2.5uS=$2.5\times10^{-6}S$

$$N = \frac{C_T - C_0 + 12}{C_{BK}}$$

$$C_T = \frac{T_{Delay}}{T_{Clock}} = \frac{2.5 \times 10^{-5}}{10^{-9}} = 2500$$

T_{clock} = Time for one Clock Cycle

$$N = \frac{2500 - 4 + 12}{23} = 109$$

mov cx,109 or mov cx,6Ch

Longer delays

```
        mov   bx,count1
lp1:    mov   cx,count2
lp2:    loop  lp2
        dex   bx
        jnz   lp1
```

Use double or triple nested loops to create a long delay.

dec or dex ?

Consider the delay introduced by the inner loop first. Treat this inner loop as a single instruction in your calculation of the delay constant *count1.*

Speed Test

```
        .STARTUP
        mov ah,02          ;Print S
        mov dl,'S'
        int 021h

        mov   bx, 30000    ;4

back2:  mov   cx, 30000    ;4
back1:  nop                ;3
        loop  back1        ;17 or 5

        dec   bx           ;2
        jnz   back2        ;16 or 4

        mov ah,02          ;Print F
        mov dl,'F'
        int 021h
        .EXIT
```

Inner loop

$$C_T = C_0 + N(C_{BK}) - 12$$
$$C_T = 4 + 30000(599992 + 2 + 16) - 12$$
$$C_T = 1.8 \times 10^{10} \text{ Clock cycles}$$

Inner loop

$$C_T = C_0 + N(C_{BK}) - 12$$
$$C_T = 4 + 30000(20) - 12$$
$$C_T = 599992$$

Program took 15 seconds to run

$$Clock\_rate = \frac{1.8 \times 10^{10} cycles}{15s}$$

$$Clock\_rate = 1.2 \times 10^9 = 1200MHz$$

$$Rated\_speed = 300MHz$$

How is computer doing four instructions at a time?

Data format (unsigned integer) in PC's

So far we have used the data stored in the registers in the following way, the following formats are very similar to that used in C/C++.

unsigned char

Unsigned, $0 \le X \le 255$

1 byte, 8bits, (al,dl,bh etc)

Reminder of c

unsigned int

Unsigned, $0 \le X \le 65535$

2 bytes, 16bits, (ax,bx,dx etc)

signed char

Twos Complement, $-128 \le X \le 127$

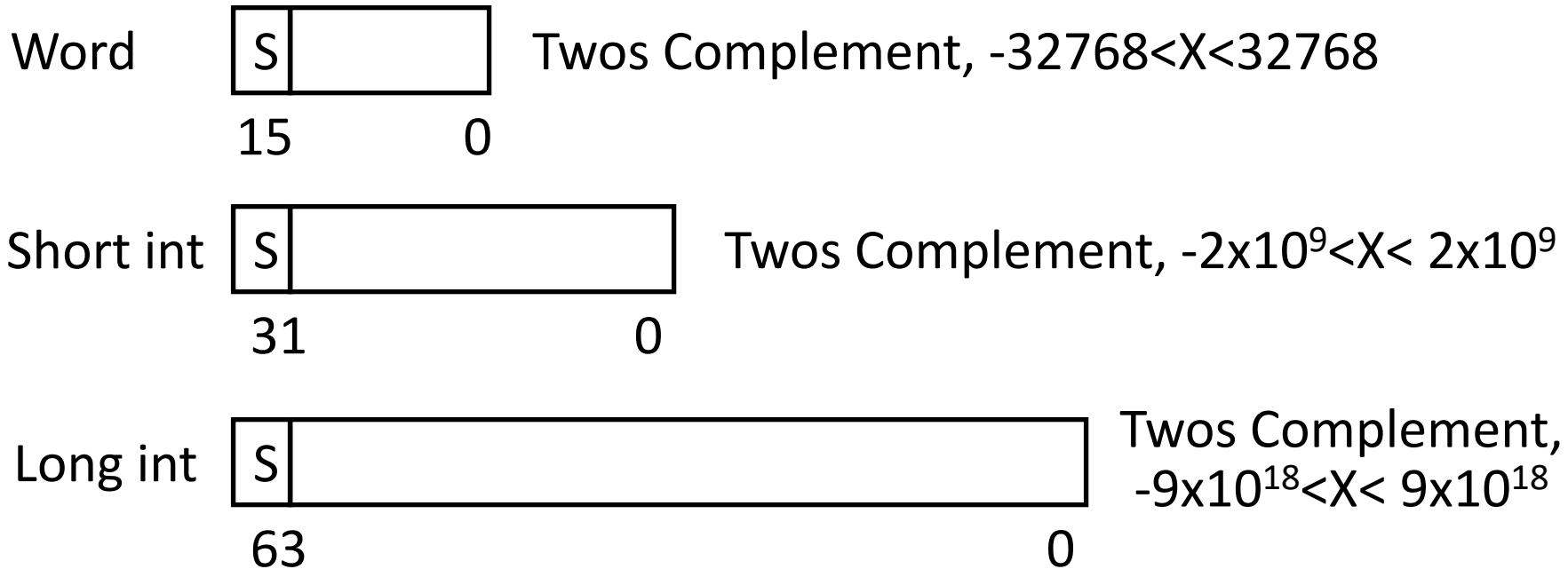
1 byte, 8bits, (al,dl,bh etc)

signed int

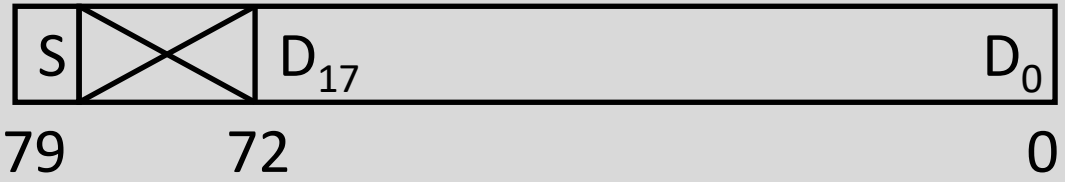
Twos Complement, $-32768 \le X \le 32767$

2 bytes, 16bits, (ax,bx,dx etc)

Data format (signed integer) in 8086



Packed Decimal



-99…99<X<99…99 (18digits)

Each byte stores two BCD digits, each digit requires 4 bits

Twos Complement

You could use a single bit to set the sign of the number. However if you do this then you need separate hardware/software for addition and subtraction. Two's complement avoids the problem.

By inverting each bit in a number and adding one to the result. A new number is formed that has the same effect when added to a number as subtracting the original number. If the MSB is 1 then the number is negative, twos complement, otherwise positive ordinary binary.

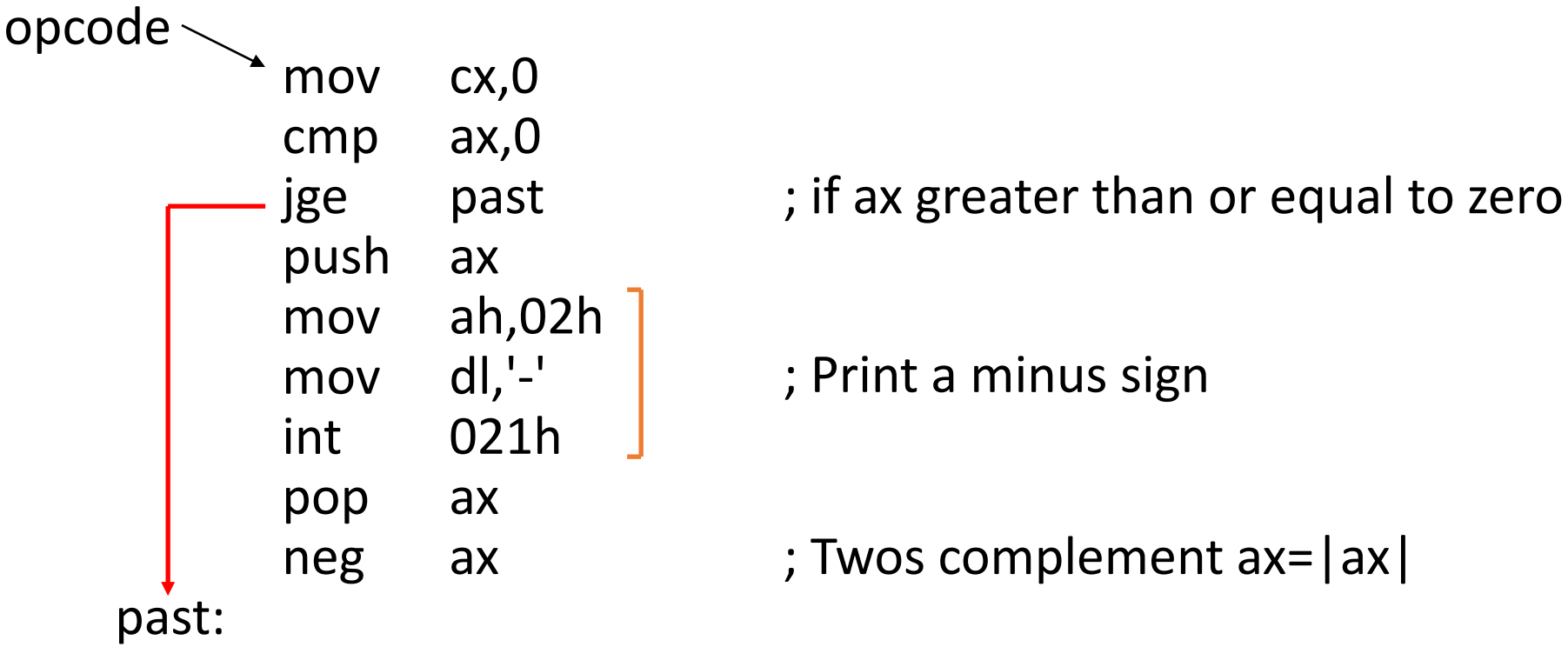
Subtract 65 from 120 using the twos complement method.

120:	0 1 1 1 1 0 0 0
65:	0 1 0 0 0 0 0 1
Invert bits:	1 0 1 1 1 1 1 0
Add 1:	1 0 1 1 1 1 1 1
Add to 120	0 0 1 1 0 1 1 1 : 55

128, 64, 32, 16, 8, 4, 2, 1

Negative numbers

The following code segment converts negative numbers to their absolute (positive) value. Add it to the start of the print algorithm to print signed numbers, it assumes the number is stored in AX.



Strings

A string is a group of ASCII characters used to describe text.

Two formats are used

String terminated with a NULL *byte=0* character

Charles Markham /0

String uses first byte to set length (rare)

/15 Charles Markham

Registers SI (Source) and DI(Destination) are used for string manipulation.

LEA	SI,Label	Load effective address in to SI
MOVSB		Move string byte
CMPSB		Compare string

movsb
Copies memory pointed to by DS:SI to the address specified in ES:DI

If DF=0 SI,DI are incremented (DF Direction Flag),
If DF=1 SI,DI are decreased.

Note a modifier 'REP' can be placed in front of any string instruction and repeats the instruction CX times.

CMPSB
Compares memory byte pointed to by SI to the memory byte pointed to by DI. The command is normally used with a REPE modifier as part of a large string test loop.

REPE: Repeat while equal
RENE: Repeat while not equal

REPE CMPSB
JNE Strings are not equal

Anatomy of an assembly language instruction (overview)

Address	Machine Code	Assembly Language
00007FF7408D1DA0	49 83 E8 08	sub r8, 8
Unused	Prefix Opcode ModR/M Operand	Operator Operands

x64 has a 48 bit address space
2^{48}=256 TBytes

49H prefix Switch to 64 bit mode REX.W 49H = 0100 1001

64 bit

Extension to r/m

83H SUB r/m16, imm8 Subtract sign-extended imm8 from r/m16

100000 1(REG Destination) 1(Word)Source is R/M
Subtract

E8H = 11 101 000 (Addressing Mode 11 = no offsets, Reg = 101 (3 further bits of opcode info), R/M = 1000 r8)

sub - opcode "83h /101b" reg_mem $v, imm 8

08H = 0000 1000 Operand

x86 Instructions can vary in length.

They all contain an operator (something to do).

Most contain operands, a reference to the data required to complete the operation

The machine code instruction can be up to 15 bytes long.

There is a nearly 1:1 correspondence between assembly language and machine code.

Prefixes

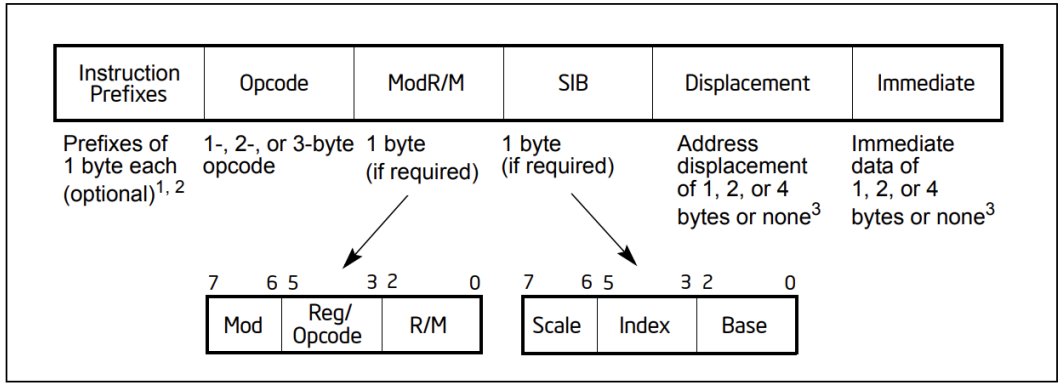


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

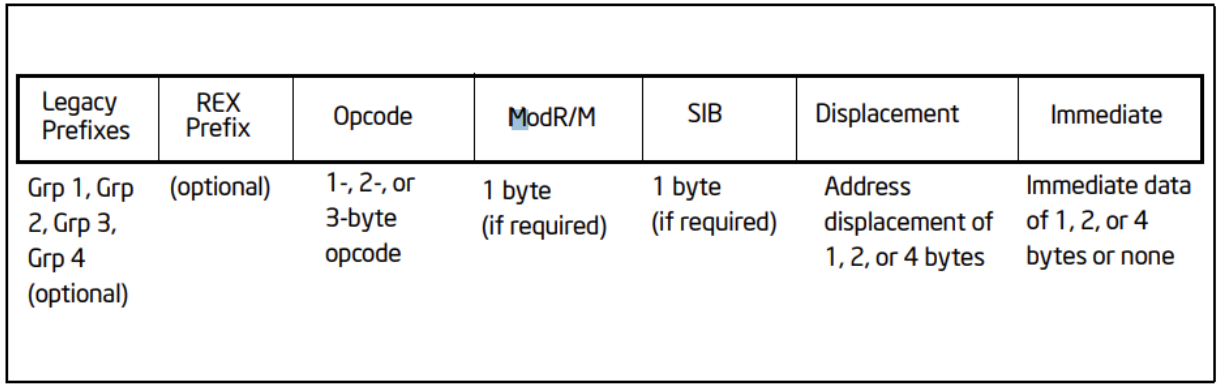


Figure 2-3. Prefix Ordering in 64-bit Mode

Legacy Prefix
Lock F0H.
Segment Override (ES,DS,CS) FS, GS (36, 26, 64, 2E,3E)
REPNE/REPNZ F2H
REP/REPE/REPZ F3H.
REX 0x44-0x4F
Operand override switch16 to 32 bit 0x67, 0X66

49H

0	1	0	0	1	0	0	1
Rex.W				64			E

SUB—Subtract

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
83 /5 *ib*	SUB *r/m16, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m16.*

4-ia-32-architectures-software-developer-

REX Prefixes

(32 bit) AX,BX,CX,DX,SI,DI etc (64 bit) adds R8, R9, R10 etc

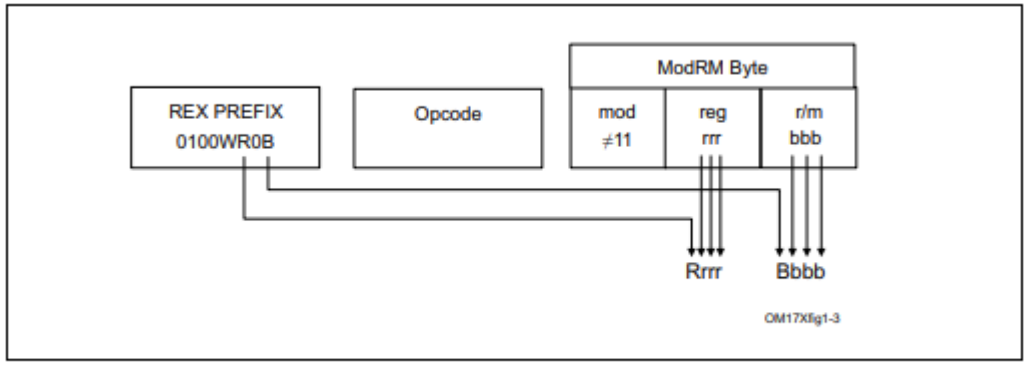


Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used

Operand size 64 bit

REX.W 49H = 0100 1001

Mod REG R/M 11 0101 1000

https://www.intel.com/content/dam/support/us/en/documents/processors/pentium4/sb/25366621.pdf

SUB—Subtract

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
83 /5 *ib*	SUB *r/m16, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m16.*

83H E8H

1	0	0	0	0	0	1	1
opcode						D	W

1	1	1	0	1	0	0	0
Mod		Reg			R/M		

Opcode for “Sub” contained in the two bytes.
First byte 32 and second byte 5.

Opcode

64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf Pg. 1306

SUB—Subtract

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2C *ib*	SUB AL, *imm8*	I	Valid	Valid	Subtract *imm8* from AL.
2D *iw*	SUB AX, *imm16*	I	Valid	Valid	Subtract *imm16* from AX.
2D *id*	SUB EAX, *imm32*	I	Valid	Valid	Subtract *imm32* from EAX.
REX.W + 2D *id*	SUB RAX, *imm32*	I	Valid	N.E.	Subtract *imm32* sign-extended to 64-bits from RAX.
80 /5 *ib*	SUB *r/m8, imm8*	MI	Valid	Valid	Subtract *imm8* from *r/m8*.
REX + 80 /5 *ib*	SUB *r/m8*, imm8*	MI	Valid	N.E.	Subtract *imm8* from *r/m8*.
81 /5 *iw*	SUB *r/m16, imm16*	MI	Valid	Valid	Subtract *imm16* from *r/m16*.
81 /5 *id*	SUB *r/m32, imm32*	MI	Valid	Valid	Subtract *imm32* from *r/m32*.
REX.W + 81 /5 *id*	SUB *r/m64, imm32*	MI	Valid	N.E.	Subtract *imm32* sign-extended to 64-bits from *r/m64*.
83 /5 *ib*	SUB *r/m16, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m16*.
83 /5 *ib*	SUB *r/m32, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m32*.
REX.W + 83 /5 *ib*	SUB *r/m64, imm8*	MI	Valid	N.E.	Subtract sign-extended *imm8* from *r/m64*.
28 /r	SUB *r/m8, r8*	MR	Valid	Valid	Subtract *r8* from *r/m8*.
REX + 28 /r	SUB *r/m8*, r8**	MR	Valid	N.E.	Subtract *r8* from *r/m8*.
29 /r	SUB *r/m16, r16*	MR	Valid	Valid	Subtract *r16* from *r/m16*.
29 /r	SUB *r/m32, r32*	MR	Valid	Valid	Subtract *r32* from *r/m32*.
REX.W + 29 /r	SUB *r/m64, r64*	MR	Valid	N.E.	Subtract *r64* from *r/m64*.
2A /r	SUB *r8, r/m8*	RM	Valid	Valid	Subtract *r/m8* from *r8*.
REX + 2A /r	SUB *r8*, r/m8**	RM	Valid	N.E.	Subtract *r/m8* from *r8*.
2B /r	SUB *r16, r/m16*	RM	Valid	Valid	Subtract *r/m16* from *r16*.
2B /r	SUB *r32, r/m32*	RM	Valid	Valid	Subtract *r/m32* from *r32*.
REX.W + 2B /r	SUB *r64, r/m64*	RM	Valid	N.E.	Subtract *r/m64* from *r64*.

NOTES:

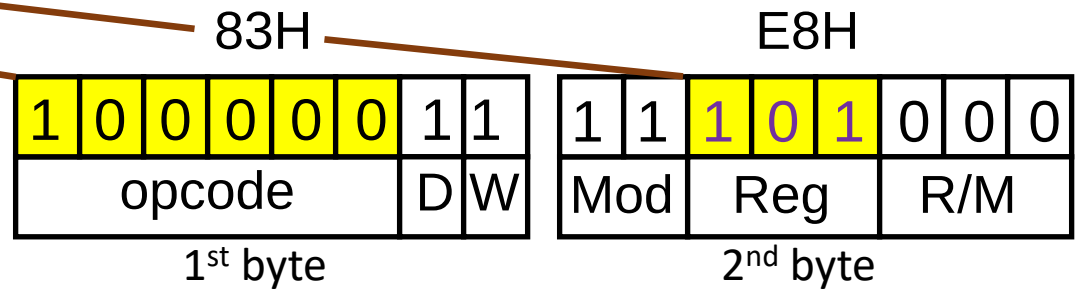
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

The opcode is the portion of a machine language instruction that specifies the operation to be performed.

On x86 machines it can be spread across 1 or bytes.

The Subtract example uses bits in the first two bytes to specify the opcode.

Reg can be used to specify a register but in this example it just contains additional opcode bits 101b=5.



https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf (local copy on Moodle)

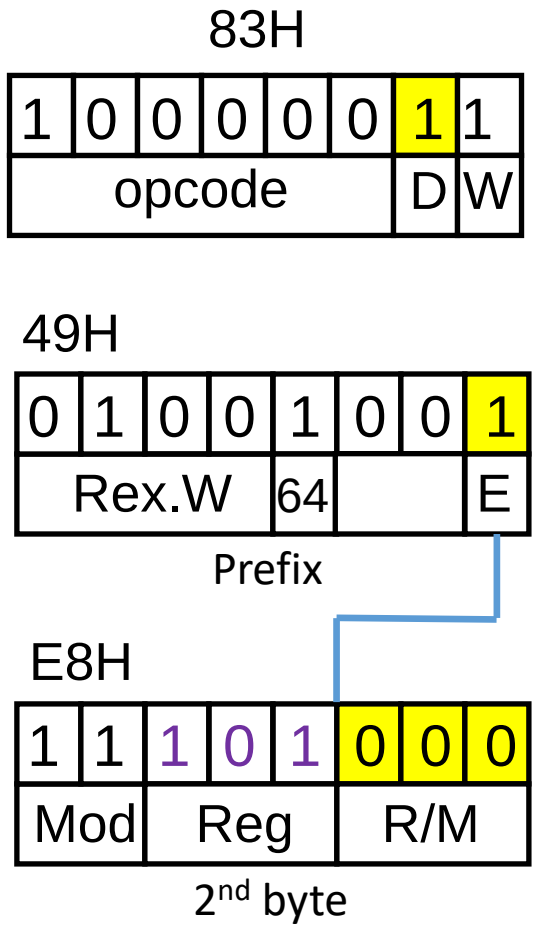
Mod RegR/M

#	Reg	64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
0	0000	rax	eax	ax	al
1	0001	rbx	ebx	bx	bl
2	0010	rcx	ecx	cx	cl
3	0011	rdx	edx	dx	dl
4	0100	rsi	esi	si	sil
5	0101	rdi	edi	di	dil
6	0110	rbp	ebp	bp	bpl
7	0111	rsp	esp	sp	spl
8	1000	r8	r8d	r8w	r8b
9	1001	r9	r9d	r9w	r9b
10	1010	r10	r10d	r10w	r10b
11	1011	r11	r11d	r11w	r11b
12	1100	r12	r12d	r12w	r12b
13	1101	r13	r13d	r13w	r13b
14	1110	r14	r14d	r14w	r14b
15	1111	r15	r15d	r15w	r15b

Mode Register Memory

ModR/M This byte follows the opcode and specifies the operand register(s).

R/M is typically the destination and Reg the source (direction bit D can reverse this).



Anatomy of an assembly language instruction (Operators)

Assembly Language

```
sub   r8,  8
```

Operator Operands

ADC	CMOVB	CMPSS	DIV	FDECSTP	FNENI	FUCOMIP	INVEPT	JP	MAXPD	MOVSB	PADDQ	PMINUB	PSUBUSB	REX.RXB	SETG	SQRTPD	VMLAUNCH
ADD	CMOVBE	CMPSW	DIVPD	FDIV	FNINIT	FUCOMP	INVLPG	JPE	MAXPS	MOVSD	PADDSB	PMOVMSKB	PSUBUSW	REX.W	SETGE	SQRTPS	VMPTRLD
ADDPD	CMOVC	CMPXCHG	DIVPS	FDIVP	FNOP	FUCOMPP	INVVPID	JPO	MAXSD	MOVSHDUP	PADDSW	PMULHUW	PSUBW	REX.WB	SETL	SQRTSD	VMPTRST
ADDPS	CMOVE	CMPXCHG16B	DIVSD	FDIVR	FNSAVE	FWAIT	IRET	JRCXZ	MAXSS	MOVSLDUP	PADDUSB	PMULHW	PUNPCKHBW	REX.WR	SETLE	SQRTSS	VMREAD
ADDSD	CMOVG	CMPXCHG8B	DIVSS	FDIVRP	FNSETPM	FXAM	IRETD	JS	MFENCE	MOVSQ	PADDUSW	PMULLW	PUNPCKHDQ	REX.WRB	SETNA	STC	VMRESUME
ADDSS	CMOVGE	COMISD	DPPD	FFREE	FNSTCW	FXCH	IRETQ	JZ	MINPD	MOVSS	PADDW	PMULUDQ	PUNPCKHQDQ	REX.WRX	SETNAE	STD	VMWRITE
ADDSUBPD	CMOVL	COMISS	DPPS	FFREEP	FNSTENV	FXCH4	JA	LAHF	MINPS	MOVSW	PALIGNR	POP	PUNPCKHWD	REX.WRXB	SETNB	STI	VMXOFF
ADDSUBPS	CMOVLE	CPUID	EMMS	FIADD	FNSTSW	FXCH7	JAE	LAR	MINSD	MOVSX	PAND	POPCNT	PUNPCKLBW	REX.WX	SETNBE	STMXCSR	VMXON
AND	CMOVNA	CQO	ENTER	FICOM	FPATAN	FXRSTOR	JB	LDDQU	MINSS	MOVSXD	PANDN	POPF	PUNPCKLDQ	REX.WXB	SETNC	STOS	WAIT
ANDNPD	CMOVNAE	CRC32	EXTRACTPS	FICOMP	FPREM	FXSAVE	JBE	LDMXCSR	MONITOR	MOVUPD	PAUSE	POPFQ	PUNPCKLQDQ	REX.X	SETNE	STOSB	WBINVD
ANDNPS	CMOVNB	CVTDQ2PD	F2XM1	FIDIV	FPREM1	FXTRACT	JC	LEA	MOV	MOVUPS	PAVGB	POR	PUNPCKLWD	REX.XB	SETNG	STOSD	WRMSR
ANDPD	CMOVNBE	CVTDQ2PS	FABS	FIDIVR	FPTAN	FYL2X	JE	LEAVE	MOVAPD	MOVZX	PAVGW	PREFETCHNTA	PUSH	ROL	SETNGE	STOSQ	XADD
ANDPS	CMOVNC	CVTPD2DQ	FADD	FILD	FRNDINT	FYL2XP1	JECXZ	LFENCE	MOVAPS	MPSADBW	PBLENDW	PREFETCHT0	PUSHF	ROR	SETNL	STOSW	XCHG
BLENDPD	CMOVNE	CVTPD2PI	FADDP	FIMUL	FRSTOR	GETSEC	JG	LFS	MOVBE	MUL	PCMPEQB	PREFETCHT1	PUSHFQ	ROUNDPD	SETNLE	STR	XGETBV
BLENDPS	CMOVNG	CVTPD2PS	FBLD	FINCSTP	FS	GS	JGE	LGDT	MOVD	MULPD	PCMPEQD	PREFETCHT2	PXOR	ROUNDPS	SETNO	SUB	XLAT
BSF	CMOVNGE	CVTPI2PD	FBSTP	FINIT	FSAVE	HADDPD	JL	LGS	MOVDDUP	MULPS	PCMPEQW	PSADBW	RCL	ROUNDSD	SETNP	SUBPD	XLATB
BSR	CMOVNL	CVTPI2PS	FCHS	FIST	FSCALE	HADDPS	JLE	LIDT	MOVDQ2Q	MULSD	PCMPESTRI	PSHUFD	RCPPS	ROUNDSS	SETNS	SUBPS	XOR
BSWAP	CMOVNLE	CVTPS2DQ	FCLEX	FISTP	FSIN	HINT_NOP	JMP	LLDT	MOVDQA	MULSS	PCMPESTRM	PSHUFHW	RCPSS	RSM	SETNZ	SUBSD	XORPD
BT	CMOVNO	CVTPS2PD	FCMOVB	FISTTP	FSINCOS	HLT	JMPE	LMSW	MOVDQU	MWAIT	PCMPGTB	PSHUFLW	RCR	RSQRTPS	SETO	SUBSS	XORPS
BTC	CMOVNP	CVTPS2PI	FCMOVBE	FISUB	FSQRT	HSUBPD	JMPF	LOCK	MOVHLPS	NEG	PCMPGTD	PSHUFW	RDMSR	RSQRTSS	SETP	SWAPGS	XRSTOR
BTR	CMOVNS	CVTSD2SI	FCMOVE	FISUBR	FST	HSUBPS	JNA	LODS	MOVHPD	NOP	PCMPGTW	PSLLD	RDPMC	SAHF	SETPE	SYSCALL	XSAVE
BTS	CMOVNZ	CVTSD2SS	FCMOVNB	FLD	FSTCW	ICEBP	JNAE	LODSB	MOVHPS	NOT	PCMPISTRI	PSLLDQ	RDTSC	SAL	SETPO	SYSENTER	XSETBV
CALL	CMOVO	CVTSI2SD	FCMOVNBE	FLD1	FSTENV	IDIV	JNB	LODSD	MOVLHPS	OR	PCMPISTRM	PSLLQ	RDTSCP	SAR	SETS	SYSEXIT	
CALLF	CMOVP	CVTSI2SS	FCMOVNE	FLDCW	FSTP	IMUL	JNBE	LODSQ	MOVLPD	ORPD	PEXTRB	PSLLW	REP	SBB	SETZ	SYSRET	
CBW	CMOVPE	CVTSS2SD	FCMOVNU	FLDENV	FSTP1	IN	JNC	LODSW	MOVLPS	ORPS	PEXTRD	PSRAD	REPE	SCAS	SFENCE	TEST	
CDQ	CMOVPO	CVTSS2SI	FCMOVU	FLDL2E	FSTP8	INC	JNE	LOOP	MOVMSKPD	OUT	PEXTRQ	PSRAW	REPNE	SCASB	SGDT	UCOMISD	
CDQE	CMOVS	CVTTPD2DQ	FCOM	FLDL2T	FSTP9	INS	JNG	LOOPE	MOVMSKPS	OUTS	PEXTRW	PSRLD	REPNZ	SCASD	SHL	UCOMISS	
CLC	CMOVZ	CVTTPD2PI	FCOM2	FLDLG2	FSTSW	INSB	JNGE	LOOPNE	MOVNTDQ	OUTSB	PINSRB	PSRLDQ	REPZ	SCASQ	SHLD	UNPCKHPD	
CLD	CMP	CVTTPS2DQ	FCOMI	FLDLN2	FSUB	INSD	JNL	LOOPNZ	MOVNTI	OUTSD	PINSRD	PSRLQ	RETF	SCASW	SHR	UNPCKHPS	
CLFLUSH	CMPPD	CVTTPS2PI	FCOMIP	FLDPI	FSUBP	INSERTPS	JNLE	LOOPZ	MOVNTPD	OUTSW	PINSRQ	PSRLW	RETN	SETA	SHRD	UNPCKLPD	
CLI	CMPPS	CVTTSD2SI	FCOMP	FLDZ	FSUBR	INSW	JNO	LSL	MOVNTPS	PACKSSDW	PINSRW	PSUBB	REX	SETAE	SHUFPD	UNPCKLPS	
CLTS	CMPS	CVTTSS2SI	FCOMP3	FMUL	FSUBRP	INT	JNP	LSS	MOVNTQ	PACKSSWB	PMADDWD	PSUBD	REX.B	SETB	SHUFPS	VERR	
CMC	CMPSB	CWD	FCOMP5	FMULP	FTST	INT1	JNS	LTR	MOVQ	PACKUSWB	PMAXSW	PSUBQ	REX.R	SETBE	SIDT	VERW	
CMOVA	CMPSD	CWDE	FCOMPP	FNCLEX	FUCOM	INTO	JNZ	MASKMOVDQU	MOVQ2DQ	PADDB	PMAXUB	PSUBSB	REX.RB	SETC	SLDT	VMCALL	
CMOVAE	CMPSQ	DEC	FCOS	FNDISI	FUCOMI	INVD	JO	MASKMOVQ	MOVS	PADDD	PMINSW	PSUBSW	REX.RX	SETE	SMSW	VMCLEAR	

X64 Assembly language operators (>618)

CISC

General x64 Operators/Instructions

Arithmetic	
ADC	Add with Carry
ADD	Add
CMP	Compare Two Operands
DEC	Decrement by 1
DIV	Unsigned Divide
IDIV	Signed Divide
IMUL	Signed Multiply
INC	Increment by 1
MUL	Unsigned Multiply
NEG	Two's Complement Negation
SBB	Integer Subtraction with Borrow
SUB	Subtract

Logical	
AND	Logical AND
NOT	One's Complement Negation
OR	Logical OR
TEST	Logical Compare
XOR	Logical Exclusive OR

Bit	
BSF	Bit Scan Forward
BSR	Bit Scan Reverse
BT	Bit test
BTC	Bit Test and Complement
BTR	Bit Test and Reset
BTS	Bit Test and Set
POPCNT	Bit Population Count

Move/Artihmetic	
CMPXCHG	Compare and Exchange
CMPXCHG16B	Compare and Exchange
CMPXCHG8B	Compare and Exchange Bytes
XADD	Exchange and Add

String	
CMPS	Compare String Operands
CMPSB	Compare String Operands
CMPSD	Compare String Operands
CMPSQ	Compare String Operands
LODS	Load String
LODSB	Load String
LODSD	Load String
LODSQ	Load String
LODSW	Load String
MOVS	Move Data from String to String
MOVSB	Move Data from String to String
MOVSD	Move Data from String to String
MOVSQ	Move Data from String to String
MOVSW	Move Data from String to String
SCAS	Scan String
SCASB	Scan String
SCASD	Scan String
SCASQ	Scan String
SCASW	Scan String
STOS	Store String
STOSB	Store String
STOSD	Store String
STOSQ	Store String
STOSW	Store String

Convert	
CBW	Convert
CDQ	Convert
CDQE	Convert
CWD	Convert
CWDE	Convert
MOVSX	Move with Sign-Extension
MOVSXD	Move with Sign-Extension
MOVZX	Move with Zero-Extend
CQO	Convert

Branch	
JA	Jump if above
JAE	Jump above or equal
JB	Jump below
JBE	Jump short if below or equal/not above (CF=1 OR ZF=1)
JC	Jump if carry
JE	Jump if equal
JECXZ	Jump short if rCX register is 0
JG	Jump greater than
JGE	Jump greater or equal
JL	Jump if less
JLE	Jump short if less or equal/not greater ((ZF=1) OR (SF!=OF))
JMP	Jump
JMPE	Jump to IA-64 Instruction Set
JMPF	Jump
JNA	Jump not above
JNAE	Jmp not above equal
JNB	Jump short if not below/above or equal/not carry (CF=0)
JNBE	Jump short if not below or equal/above (CF=0 AND ZF=0)
JNC	Jump if nor carry
JNE	Jump if not equal
JNG	Jump not greater
JNGE	Jump not greater or equal
JNL	Jump short if not less/greater or equal (SF=OF)
JNLE	Jump short if not less nor equal/greater ((ZF=0) AND (SF=OF))
JNO	Jump short if not overflow (OF=0)
JNP	Jump short if not parity/parity odd (PF=0)
JNS	Jump short if not sign (SF=0)
JNZ	Jump short if not zero/not equal (ZF=0)
JO	Jump if odd
JP	Jump parity even
JPE	Jump parity even
JPO	Jump if parity odd
JRCXZ	Jump if CX is 0
JS	Jump if sign
JZ	Jump if zero
LOOP	Decrement count; Jump short if count!=0, ZF=1
LOOPE	Decrement count; Jump short if count!=1
LOOPNE	Decrement count; Jump short if count!=0 and ZF=0
LOOPNZ	Decrement count; Jump short if count!=0 and ZF=0
LOOPZ	Decrement count; Jump short if count!=0 and ZF=1

Stack	
ICEBP	Call to Interrupt Procedure
CALL	Call Procedure
CALLF	Call Procedure
RETF	Return from procedure
RETN	Return from procedure
INT	Call to Interrupt Procedure
INT1	Call to Interrupt Procedure
INTO	Call to Interrupt Procedure
IRET	Interrupt Return
IRETD	Interrupt Return
IRETQ	Interrupt Return
ENTER	Make Stack Frame for Procedure Params
LEAVE	High Level Procedure Exit
POP	Pop Word
PUSH	Push Word
POPF	Pop Stack into rFLAGS Register
POPFQ	Pop Stack into rFLAGS Register
PUSHF	Push rFLAGS Register onto the Stack
PUSHFQ	Push rFLAGS Register onto the Stack

Control	
CPUID	CPU Identification
HINT_NOP	Hintable NOP
NOP	No Operation

Shift/Rotate	
RCL	Rotate
RCR	Rotate
ROL	Rotate
ROR	Rotate
SAL	Shift
SAR	Shift
SHL	Shift
SHLD	Double Precision Shift Left
SHR	Shift
SHRD	Double Precision Shift Right

General x64 Operators/Instructions

Move	
BSWAP	Byte Swap
CMOVB	Conditional Move - below/not above or equal/carry (CF=1)
CMOVBE	Conditional Move - below or equal/not above (CF=1 OR ZF=1)
CMOVC	Conditional Move - carry
CMOVE	Conditional Move - equal
CMOVG	Conditional Move - greater
CMOVGE	Conditional Move - greater or equal
CMOVL	Conditional Move - less/not greater (SF!=OF)
CMOVLE	Conditional Move - less or equal/not greater ((ZF=1) OR (SF!=OF))
CMOVNA	Conditional Move - below or equal/not above
CMOVNAE	Conditional Move - not below or equal/not above
CMOVNB	Conditional Move - not below/above or equal/not carry (CF=0)
CMOVNBE	Conditional Move - not below or equal/above (CF=0 AND ZF=0)
CMOVNC	Conditional Move - no carry
CMOVNE	Conditional Move - not equal
CMOVNG	Conditional Move - not greater``
CMOVNGE	Conditional Move - not greater`equal
CMOVNL	Conditional Move - not less/greater or equal (SF=OF)
CMOVNLE	Conditional Move - not less nor equal/greater ((ZF=0) AND (SF=OF))
CMOVNO	Conditional Move - not overflow (OF=0)
CMOVNP	Conditional Move - not parity/parity odd (PF=0)
CMOVNS	Conditional Move - not sign (SF=0)
CMOVNZ	Conditional Move - not zero/not equal (ZF=0)
CMOVO	Conditional Move - overflow (OF=1)
CMOVP	Conditional Move - parity/parity even (PF=1)
CMOVPE	Conditional Move - parity/parity even (PF=1)
CMOVPO	Conditional Move - parity/parity even (PF=0)
CMOVS	Conditional Move - sign (SF=1)
CMOVZ	Conditional Move - zero/equal (ZF=1)
LEA	Load Effective Address
MOV	Move
MOVBE	Move Data After Swapping Bytes
MOVD	Move Doubleword/Quadword
SETA	Set byte above
SETAE	Set byte above or equal
SETB	Set Byte on Condition - below/not above or equal/carry (CF=1)

SETBE	Set Byte on Condition - below or equal/not above (CF=1 OR ZF=1)
SETC	Set byte on carry
SETE	Set byte on equal
SETG	Set byte greater than
SETGE	Set byte greater or equal
SETL	Set Byte on Condition - less/not greater (SF!=OF)
SETLE	Set Byte on Condition - less or equal/not greater ((ZF=1) OR (SF!=OF))
SETNA	Set byte not above
SETNAE	Set byte not above or equal
SETNB	Set Byte on Condition - not below/above or equal/not carry (CF=0)
SETNBE	Set Byte on Condition - not below or equal/above (CF=0 AND ZF=0)
SETNC	Set byte no carry
SETNE	Set byte not equal
SETNG	Set byte not greater
SETNGE	Set byte not greater or equal
SETNL	Set Byte on Condition - not less/greater or equal (SF=OF)
SETNLE	Set Byte on Condition - not less nor equal/greater ((ZF=0) AND (SF=OF))
SETNO	Set Byte on Condition - not overflow (OF=0)
SETNP	Set Byte on Condition - not parity/parity odd (PF=0)
SETNS	Set Byte on Condition - not sign (SF=0)
SETNZ	Set Byte on Condition - not zero/not equal (ZF=0)
SETO	Set Byte on Condition - overflow (OF=1)
SETP	Set Byte on Condition - parity/parity even (PF=1)
SETPE	Set byte parity even
SETPO	Set parity parity odd
SETS	Set Byte on Condition - sign (SF=1)
SETZ	Set Byte on Condition - zero/equal (ZF=1)
XCHG	Exchange Register/Memory with Register
XLAT	Table Look-up Translation
XLATB	Table Look-up Translation
LAHF	Load Status Flags into AH Register
SAHF	Store AH into Flags
LFS	Load Far Pointer
LGS	Load Far Pointer
LSS	Load Far Pointer

Input/Output	
IN	In from port
INS	Input from Port to String
INSB	In to string
INSD	In to string
INSW	In to string
OUT	Output to Port
OUTS	Output String to Port
OUTSB	Output String to Port
OUTSD	Output String to Port
OUTSW	Output String to Port

Flag control	
CLC	Clear Carry Flag
CLD	Clear Direction Flag
CLI	Clear Interrupt Flag
CMC	Complement Carry Flag
CMOVA	Conditional Move - not above or equal
CMOVAE	Conditional Move - below/not above
STC	Set Carry Flag
STD	Set Direction Flag
STI	Set Interrupt Flag

General purpose registers

8-bit GP	16-bit GP	32-bit GP	64-bit GP	80-bit x87	64-bit MMX	128-bit XMM	256-bit YMM	16-bit Segment	32-bit Control	32-bit Debug
AL	AX	EAX	RAX	ST0	MMX0	XMM0	YMM0	ES	CR0	DR0
CL	CX	ECX	RCX	ST1	MMX1	XMM1	YMM1	CS	CR1	DR1
DL	DX	EDX	RDX	ST2	MMX2	XMM2	YMM2	SS	CR2	DR2
BL	BX	EBX	RBX	ST3	MMX3	XMM3	YMM3	DS	CR3	DR3
AH, SPL1	SP	ESP	RSP	ST4	MMX4	XMM4	YMM4	FS	CR4	DR4
CH, BPL1	BP	EBP	RBP	ST5	MMX5	XMM5	YMM5	GS	CR5	DR5
DH, SIL1	SI	ESI	RSI	ST6	MMX6	XMM6	YMM6	-	CR6	DR6
BH, DIL1	DI	EDI	RDI	ST7	MMX7	XMM7	YMM7	-	CR7	DR7
R8L	R8W	R8D	R8	-	MMX0	XMM8	YMM8	ES	CR8	DR8
R9L	R9W	R9D	R9	-	MMX1	XMM9	YMM9	CS	CR9	DR9
R10L	R10W	R10D	R10	-	MMX2	XMM10	YMM10	SS	CR10	DR10
R11L	R11W	R11D	R11	-	MMX3	XMM11	YMM11	DS	CR11	DR11
R12L	R12W	R12D	R12	-	MMX4	XMM12	YMM12	FS	CR12	DR12
R13L	R13W	R13D	R13	-	MMX5	XMM13	YMM13	GS	CR13	DR13
R14L	R14W	R14D	R14	-	MMX6	XMM14	YMM14	-	CR14	DR14
R15L	R15W	R15D	R15	-	MMX7	XMM15	YMM15	-	CR15	DR15

Addressing modes

Immediate: the value is stored in the instruction. ADD EAX, 14 ; add 14 into 32-bit EAX

Register: data transferred from register to register ADD R8L, AL ; add 8 bit AL into R8L

Indirect: Use a 8, 16, or 32 bit displacement (offset number)
General purpose registers for base and index,
Scale of 1, 2, 4, or 8 to multiply the index.
Prefixed with segment FS: or GS:

MOV R8W, 1234[8*RAX+RCX] ; move word at address 8*RAX+RCX+1234 into R8W

The dword ptr (32 bit pointer) tells the assembler how to encode the MOV instruction.

x86 32 bit

Direct Addressing

In x64 it is possible but rarely used

mov rax,[qword address]

Register Indirect Addressing

mov ax,CS[BX]

RIP-relative addressing: this is new for x64 and allows accessing data tables and such in the code relative to the current instruction pointer, making position independent code easier to implement.

```
mov al,[RIP]    ; This returns the opcode for nop (RIP: Relative to the current instruction pointer)
nop
```

https://software.intel.com/content/www/us/en/develop/articles/introduction-to-x64-assembly.html

Anatomy of an assembly language instruction (general purpose registers)

Assembly Language

```
sub   r8,  8
```

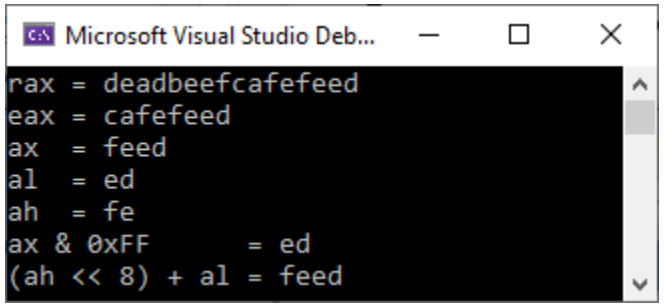
Operator Operands

	(63)MSB								(31)				(15)		(7) LSB(0)	
64-bit	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000 **RAX**
32-bit									0000	0000	0000	0000	0000	0000	0000	0000 **EAX**
16-bit													0000	0000	0000	0000 **AX**
8-bit															0000	0000 **AL**
8-bit															0000	0000 **AH**

#	Reg	64-bit register	Lower 32 bits	Lower 16 bits	Upper 8 bits	Lower 8 bits	Name
0	0000	rax	eax	ax	ah	al	Accumulator
1	0001	rbx	ebx	bx	bh	bl	Base
2	0010	rcx	ecx	cx	ch	cl	Counter
3	0011	rdx	edx	dx	dh	dl	Data
4	0100	rsi	esi	si	sih	sil	Source
5	0101	rdi	edi	di	dih	dil	Destination
6	0110	rbp	ebp	bp	dph	bpl	Base pointer
7	0111	rsp	esp	sp	sph	spl	Stack pointer
8	1000	r8	r8d	r8w		r8b	register 8
9	1001	r9	r9d	r9w		r9b	
10	1010	r10	r10d	r10w		r10b	
11	1011	r11	r11d	r11w		r11b	
12	1100	r12	r12d	r12w		r12b	
13	1101	r13	r13d	r13w		r13b	
14	1110	r14	r14d	r14w		r14b	
15	1111	r15	r15d	r15w		r15b	register 15

al, ah, ax, eax and rax share common bits, similar to a union in C.

```cpp
// https://stackoverflow.com/questions/27336734/how-to-tie-variables-in-c
#include <iostream>
#include <cstdint>
using namespace std;

union reg_t
{
    uint64_t rx;
    uint32_t ex;
    uint16_t x;
    struct {
        uint8_t l;
        uint8_t h;
    };
};

int main()
{
    reg_t a;

    a.rx = 0xdeadbeefcafefeed;

    cout << "rax = " << hex << a.rx << endl;
    cout << "eax = " << hex << a.ex << endl;
    cout << "ax  = " << hex << a.x << endl;
    cout << "al  = " << hex << (uint16_t)a.l << endl;
    cout << "ah  = " << hex << (uint16_t)a.h << endl;
    cout << "ax & 0xFF      = " << hex << (a.x & 0xFF) << endl;
    cout << "(ah << 8) + al = " << hex << (a.h << 8) + a.l << endl;
}
```



Using the disassembler to test an idea

```
int main()
{
 unsigned int x = 9;

 x++;

 x += 10;

}
```

Optimization	Optimizations (Favor Speed) (/Ox)
Inline Function Expansion	Default
Enable Intrinsic Functions	No
Favor Size Or Speed	**Favor fast code (/Ot)**
Omit Frame Pointers	
Enable Fiber-Safe Optimizations	No
Whole Program Optimization	No

Optimised for speed

```
00E91D98 C7 45 F8 09 00 00 00 mov         dword ptr [x],9
00E91D9F 8B 45 F8             mov         eax,dword ptr [x]
00E91DA2 83 C0 01             add         eax,1
00E91DA5 89 45 F8             mov         dword ptr [x],eax
00E91DA8 8B 45 F8             mov         eax,dword ptr [x]
00E91DAB 83 C0 0A             add         eax,0Ah
00E91DAE 89 45 F8             mov         dword ptr [x],eax
```

Optimization	Maximum Optimization (Favor Size) (/O1)
Inline Function Expansion	Default
Enable Intrinsic Functions	No
Favor Size Or Speed	**Favor small code (/Os)**
Omit Frame Pointers	
Enable Fiber-Safe Optimizations	No
Whole Program Optimization	**Yes (/GL)**

Optimised for size (it is the same)

```
00E91D98 C7 45 F8 09 00 00 00 mov         dword ptr [x],9
00E91D9F 8B 45 F8             mov         eax,dword ptr [x]
00E91DA2 83 C0 01             add         eax,1
00E91DA5 89 45 F8             mov         dword ptr [x],eax
00E91DA8 8B 45 F8             mov         eax,dword ptr [x]
00E91DAB 83 C0 0A             add         eax,0Ah
00E91DAE 89 45 F8             mov         dword ptr [x],eax
```

May have expected ->

```
00E91DA2 40                   inc         eax
```

The INC and DEC instructions modify only a subset of the bits in the flag register. This creates a dependence on all previous writes of the flag register. This is especially problematic when these instructions are on the critical path because they are used to change an address for a load on which many other instructions depend. Assembly/Compiler Coding Rule 33. (M impact, H generality) *INC and DEC instructions should be replaced with ADD or SUB instructions*, because ADD and SUB overwrite all flags, whereas INC and DEC do not, therefore creating false dependencies on earlier instructions that set the flags.

Anatomy of an assembly language instruction (overview)

Prefixes (0, 1, 2, 3, 4) VEX (0, 2, 3) Opcode (1) ModR/M (1) SIB (0,1) DISP (0, 1, 2, 4) IMM (0, 1, 2, 4)

```
48 8D 9C DB 01 01 00 00          lea rbx,[rbx+rbx*8+101h]
```

Can we decode this statement?

Prefix: Group 1:LOCK/REP, 2:Segement override CS, SS, DS, ES, FS, GS, 3:Operand size 66H, 4: Address size
VEX (0, 2, 3) - AVX extension
Opcode - instruction
ModR/M – Mode register/memory
SIB – Scale, Index, Base
Displacement – and immediate value added to address
Immediate – immediate (operand) value within instruction

General x64 Operators/Instructions – code example

```
// x86 (not x64) asm code inline
#include <iostream>
unsigned int test(unsigned int);

unsigned int num;   // 64 bit

int main()
{
    num = 999983;
    if (test(num)==1) printf("Prime"); else printf("Not prime");
}

unsigned int test(unsigned int testPrime)
{
    unsigned int testNum=2;

    __asm
    {
    retest:  mov eax, testPrime
             cmp eax, testNum
             je  prime        // if we reach the number it is prime

             mov edx, 0
             mov eax, testPrime
             mov ebx, testNum
             div ebx          // eax=(edx:eax)/ebx, remainder in edx

             inc testNum      // 2,3,4....,testPrime

             mov eax, edx
             cmp eax, 0
             jnz  retest      // if remainder not 0 then try again

 not_prime : mov eax, 0
             jmp over
     prime : mov eax, 1

     over:   mov testNum, eax
    }

    return(testNum);
}
```

```
Microsoft Visual Studio Debug Console
Prime
C:\Users\Alien\source\repos\MASM1\Debug\MASM1.exe (process 3552) exited
 with code 0.
To automatically close the console when debugging stops, enable Tools->
Options->Debugging->Automatically close the console when debugging stop
s.
Press any key to close this window . . .
```

Disassembly

```
__asm
{          1181D – 117FF = 1E Hex -> FF-1E = E1
00D117F8 C7 45 F4 02 00 00 00 mov         dword ptr [testNum],2
retest:
00D117FF 8B 45 08             mov         eax,dword ptr [testPrime]
00D11802 3B 45 F4             cmp         eax,dword ptr [testNum]
00D11805 74 1E                je          test+55h (0D11825h)
00D11807 BA 00 00 00 00       mov         edx,0
00D1180C 8B 45 08             mov         eax,dword ptr [testPrime]
00D1180F 8B 5D F4             mov         ebx,dword ptr [testNum]
00D11812 F7 F3                div         eax,ebx
00D11814 FF 45 F4             inc         dword ptr [testNum]
00D11817 8B C2                mov         eax,edx
00D11819 83 F8 00             cmp         eax,0
00D1181C 75 E1                jne         test+2Fh (0D117FFh)
not_prime:
00D1181E B8 00 00 00 00       mov         eax,0
00D11823 EB 05                jmp         over (0D1182Ah)
prime:
00D11825 B8 01 00 00 00       mov         eax,1
over:
00D1182A 89 45 F4             mov         dword ptr [testNum],eax
00D1182D 8B 45 F4             mov         eax,dword ptr [testNum]
return(testNum);
}
```

31 bytes back

32 bit value 0

5 bytes forward

Offset jump