Høgskolen i Bergen
Avdeling for ingeniørutdanning
Institutt for data- og realfag

Eksamen i	: DAT103/TOD077 – Datamaskiner og operativsystemer
Klasse	: 2. klasse Data / Inf
Dato	: 17. desember 2013

Antall oppgaver	: 5
Antall sider	: 12 sider inkludert forside og vedlegg
Vedlegg	: Intel assemblyfunksjoner DOS int21h Utvalgt bsd-kommando
Hjelpemidler	: Geir Maribu: "Praktisk Linux"
Tid	: 09.00-13.00 (4 timer)
Målform	: Norsk - bokmål
Sensor	: Ingen
Faglærer	: Atle Geitung

Postboks 7030, 5020 Bergen. Tlf. 55 58 75 00, Fax 55 58 77 90
Besøksadr.: Nygårdsgt. 112, Bergen

Oppgave 1

a) Hva er et operativsystem?

b) Anta at vi på et gitt tidspunkt har tre prosesser P_1, P_2 og P_3 kjørende. De trenger henholdsvis 5ms, 7ms og 3ms mer prosessor-tid (CPU-tid) for å bli fullført. Prosessoren kan kun kjøre en prosess om gangen, men prosessene kan avbrytes og prosessoren vil kjøre en prosess i 2ms hver gang en prosess slipper til.

   Forklar hvordan planleggingsalgoritmene Round Robin (RR) og Shortest Job First (SJF) fungerer og vis hvordan prosessene P_1, P_2 og P_3 blir utført ferdig ved bruk av de to planleggingsalgoritmene.

c) Hva er en kritisk region og hva er hensikten med kritiske regioner (critical sections)? Forklar også hvordan man kan implementere en kritisk region.

d) Hvilke fire kriterier må være oppfylt for at det skal oppstå en vranglås (deadlock)? Forklar hvordan Bankierens (Banker's) algoritme kan hjelpe oss å unngå vranglås.

Oppgave 2

a) Oppdeling av hukommelse gjøres ved segmentinndeling (segmentation) og sideinndeling (paging). Forklar disse to inndelingsmetodene og hvilke ulemper og fordeler de har. Fragmentering er et sentralt begrep her.

b) Forklar hva som menes med virtuell hukommelse og hvorfor virtuell hukommelse er nødvendig i dagens operativsystem.

c) La oss anta at vi har et kjørende program som skal hente data. Adressen til dataene er en adresse i den virtuelle hukommelsen som operativsystemet tilbyr. Forklar hvordan prosessoren får hentet dataene fra den fysiske primærhukommelsen (main memory). Eller sagt med andre ord, hva skjer når adresserte data som kan være lagret i sekundærlager blir gjort tilgjengelig i primærhukommelse slik at prosessoren kan bruke dem?

Oppgave 3

a) Gitt følgende skall-program:

```
#!/bin/bash

E=1
B=16

if [ -z "$1" ]
then
  echo "Melding"
  exit $E
fi

echo ""$1" "$B" o p" | dc
exit 0
```

Hva gjør programmet?

b) Koden legges inn i en tekstfil med navnet program.sh. Hva må vi gjøre for å kunne kjøre programmet fra kommandolinjen?

c) Lag et skallprogram som heter listbrukere.sh. Dette skal lese filen `/etc/passwd` og hente ut brukerne i systemet fra denne filen. `/etc/passwd` er en tekstfil og inneholder en del kolonner som er adskilt med kolon, en linje for hver bruker og første kolonne er brukernavnet. Eksempel på en linje:

```
root:x:0:0:root:/root:/bin/bash
```

Skallprogrammet som du skal lage, skal først skrive ut alle brukernavnene og til slutt antall brukere til `stdio`.

Oppgave 4

Til høyre er assemblykoden til et program listet. Linjenummer er tatt med for å forenkle kommentering av programmet

a) Gå gjennom linjene i programkoden og forklar hva de gjør.

b) Forklar hva programmet gjør.

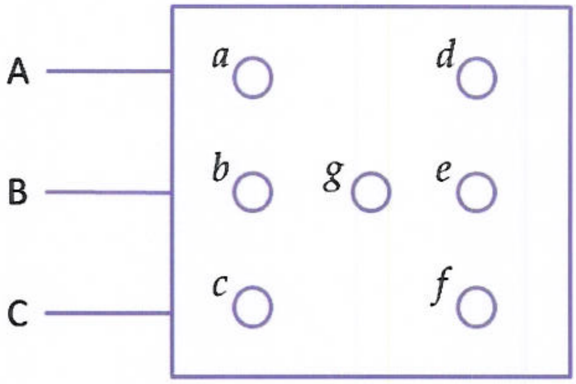
c) Oversett følgende Java-lignende algoritme til assembly-kode:

```
ax = 10;
while (ax > 0) {
    // kode
    ax--;
}
```

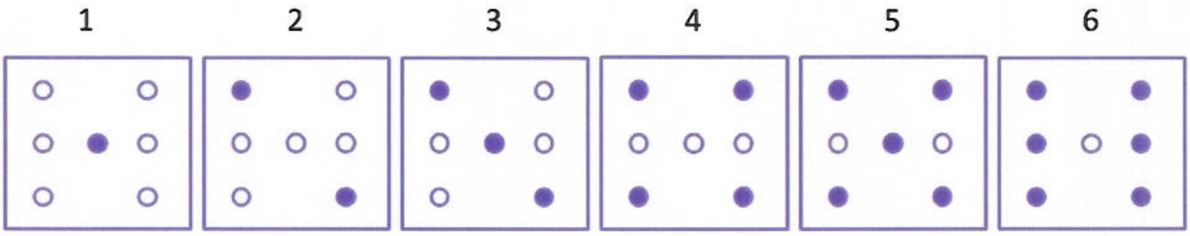
Variabelen ax er registeret ax.

```
1       cr equ 13   ;Carriage return
2       lf equ 10   ;Line feed
3
4   segment stack stack
5       resb 64
6   stacktop:
7
8   segment data
9       msg1 db "Inndata: ","$"
10      msg2 db cr,lf,"ja","$"
11      msg3 db cr,lf,"nei","$"
12
13  segment code
14
15  ..start:
16      mov ax,data
17      mov ds,ax
18
19  main:
20      mov ah,09h
21      mov dx,msg1
22      int 21h
23
24      mov ah,01h
25      int 21h
26      and al,01h
27      cmp al,00h
28      je Nei
29  Ja:
30      mov ah,09h
31      mov dx,msg2
32      int 21h
33      jmp Ferdig
34  Nei:
35      mov ah,09h
36      mov dx,msg3
37      int 21h
38  Ferdig:
39      mov ah,4Ch
40      int 21h
41  end main
42
```

Oppgave 5



Figuren til høyre viser en digital terning bestående av 7 dioder som styres av en dekoder. Den har tre innganger, A, B og C. Dekoderen skal dekode verdiene 1, 2, 3, 4, 5 og 6 slik at riktige dioder lyser og viser terningens verdi på samme måte som en vanlig terning, se figuren under for dekoding. Tallet representeres digital med sifrene CBA og andre verdier enn 1, 2, 3, 4, 5 og 6 vil ikke forekomme. For eksempel vil en ener være representert som CBA = 001 og dioden g i figuren vil lyse.



a) Sett opp sannhetstabellen for hele dekoderen (for både a, b, c, d, e, f og g).

b) Sett opp karnaughdiagrammet for a, b, c og g.

c) Finn enklest mulig boolsk uttrykk for a, b, c og g.

d) Tegn kretsløsningen for de boolske uttrykkene fra c).

Lykke til og God Jul!

Atle Geitung

Vedlegg – bash (kommando som ikke er i læreboken)

NAME

```
       dc - an arbitrary precision calculator
```

SYNOPSIS

```
       dc [-V] [--version] [-h] [--help]
          [-e scriptexpression] [--expression=scriptexpression]
          [-f scriptfile] [--file=scriptfile]
          [file ...]
```

DESCRIPTION

dc is a reverse-polish desk calculator which supports unlimited precision arithmetic. It also allows you to define and call macros. Normally dc reads from the standard input; if any command arguments are given to it, they are filenames, and dc reads and executes the contents of the files before reading from standard input. All normal output is to standard output; all error output is to standard error.

A reverse-polish calculator stores numbers on a stack. Entering a number pushes it on the stack. Arithmetic operations pop arguments off the stack and push the results.

To enter a number in dc, type the digits (using upper case letters A through F as "digits" when working with input bases greater than ten), with an optional decimal point. Exponential notation is not supported. To enter a negative number, begin the number with ``_''. ``-'' cannot be used for this, as it is a binary operator for subtraction instead. To enter two numbers in succession, separate them with spaces or newlines. These have no meaning as commands.

OPTIONS

dc may be invoked with the following command-line options:

-V
--version
: Print out the version of dc that is being run and a copyright notice, then exit.

-h
--help
: Print a usage message briefly summarizing these command-line options and the bug-reporting address, then exit.

-e script
--expression=script
: Add the commands in script to the set of commands to be run while processing the input.

-f script-file
--file=script-file
: Add the commands contained in the file script-file to the set of commands to be run while processing the input.

If any command-line parameters remain after processing the above, these parameters are interpreted as the names of input files to be processed. A file name of - refers to the standard input stream. The standard input will processed if no script files or expressions are specified.

Printing Commands

p
: Prints the value on the top of the stack, without altering the stack. A newline is printed after the value.

n
: Prints the value on the top of the stack, popping it off, and does not print a newline after.

P Pops off the value on top of the stack. If it it a string, it is simply printed without a trailing newline. Otherwise it is a number, and the integer portion of its absolute value is printed out as a "base (UCHAR_MAX+1)" byte stream. Assuming that (UCHAR_MAX+1) is 256 (as it is on most machines with 8-bit bytes), the sequence KSK0k1/_1Ss [ls*]Sxd0>x [256~Ssd0<x]dsxxsx[q]Sq[Lsd0>qaPlxx] dsxxsx0sqLqsxLxLK+k could also accomplish this function. (Much of the complexity of the above native-dc code is due to the ~ computing the characters backwards, and the desire to ensure that all registers wind up back in their original states.)

f Prints the entire contents of the stack without altering anything. This is a good command to use if you are lost or want to figure out what the effect of some command has been.

Arithmetic

+ Pops two values off the stack, adds them, and pushes the result. The precision of the result is determined only by the values of the arguments, and is enough to be exact.

- Pops two values, subtracts the first one popped from the second one popped, and pushes the result.

* Pops two values, multiplies them, and pushes the result. The number of fraction digits in the result depends on the current precision value and the number of fraction digits in the two arguments.

/ Pops two values, divides the second one popped from the first one popped, and pushes the result. The number of fraction digits is specified by the precision value.

% Pops two values, computes the remainder of the division that the / command would do, and pushes that. The value computed is the same as that computed by the sequence Sd dld/ Ld*- .

~ Pops two values, divides the second one popped from the first one popped. The quotient is pushed first, and the remainder is pushed next. The number of fraction digits used in the division is specified by the precision value. (The sequence SdSn lnld/ LnLd% could also accomplish this function, with slightly different error checking.)

^ Pops two values and exponentiates, using the first value popped as the exponent and the second popped as the base. The fraction part of the exponent is ignored. The precision value specifies the number of fraction digits in the result.

| Pops three values and computes a modular exponentiation. The first value popped is used as the reduction modulus; this value must be a non-zero number, and should be an integer. The second popped is used as the exponent; this value must be a non-negative number, and any fractional part of this exponent will be ignored. The third value popped is the base which gets exponenttiated, which should be an integer. For small integers this is like the sequence Sm^Lm%, but, unlike ^, this command will work with arbitrarily large exponents.

v Pops one value, computes its square root, and pushes that. The precision value specifies the number of fraction digits in the result.

Most arithmetic operations are affected by the ``precision value'', which you can set with the k command. The default precision value is zero, which means that all arithmetic except for addition and subtraction produces integer results.

Stack Control

c Clears the stack, rendering it empty.

d Duplicates the value on the top of the stack, pushing another copy of it. Thus, ``4d*p'' computes 4 squared and prints it.

r Reverses the order of (swaps) the top two values on the stack. (This can also be accomplished with the sequence SaSbLaLb.)

Registers

dc provides at least 256 memory registers, each named by a single character. You can store a number or a string in a register and retrieve it later.

sr Pop the value off the top of the stack and store it into register r.

lr Copy the value in register r and push it onto the stack. This does not alter the contents of r.

Each register also contains its own stack. The current register value is the top of the register's stack.

Sr Pop the value off the top of the (main) stack and push it onto the stack of register r. The previous value of the register becomes inaccessible.

Lr Pop the value off the top of register r's stack and push it onto the main stack. The previous value in register r's stack, if any, is now accessible via the lr command.

Parameters

dc has three parameters that control its operation: the precision, the input radix, and the output radix. The precision specifies the number of fraction digits to keep in the result of most arithmetic operations. The input radix controls the interpretation of numbers typed in; all numbers typed in use this radix. The output radix is used for printing numbers.

The input and output radices are separate parameters; you can make them unequal, which can be useful or confusing. The input radix must be between 2 and 16 inclusive. The output radix must be at least 2. The precision must be zero or greater. The precision is always measured in decimal digits, regardless of the current input or output radix.

i Pops the value off the top of the stack and uses it to set the input radix.

o Pops the value off the top of the stack and uses it to set the output radix.

k Pops the value off the top of the stack and uses it to set the precision.

I Pushes the current input radix on the stack.

O Pushes the current output radix on the stack.

K Pushes the current precision on the stack.

Strings

dc has a limited ability to operate on strings as well as on numbers; the only things you can do with strings are print them and execute them as macros (which means that the contents of the string are processed as dc commands). All registers and the stack can hold strings, and dc always knows whether any given object is a string or a number. Some commands such as arithmetic operations demand numbers as arguments and print errors if given strings. Other commands can accept either a number or a string; for example, the p command can accept either and prints the object according to its type.

[characters]
: Makes a string containing characters (contained between balanced [and] characters), and pushes it on the stack. For example, [foo]P prints the characters foo (with no newline).

a
: The top-of-stack is popped. If it was a number, then the low-order byte of this number is converted into a string and pushed onto the stack. Otherwise the top-of-stack was a string, and the first character of that string is pushed back.

x
: Pops a value off the stack and executes it as a macro. Normally it should be a string; if it is a number, it is simply pushed back onto the stack. For example, [1p]x executes the macro 1p which pushes 1 on the stack and prints 1 on a separate line.

Macros are most often stored in registers; [1p]sa stores a macro to print 1 into register a, and lax invokes this macro.

>r
: Pops two values off the stack and compares them assuming they are numbers, executing the contents of register r as a macro if the original top-of-stack is greater. Thus, 1 2>a will invoke register a's contents and 2 1>a will not.

!>r
: Similar but invokes the macro if the original top-of-stack is not greater than (less than or equal to) what was the second-to-top.

<r
: Similar but invokes the macro if the original top-of-stack is less.

!<r
: Similar but invokes the macro if the original top-of-stack is not less than (greater than or equal to) what was the second-to-top.

=r
: Similar but invokes the macro if the two numbers popped are equal.

!=r
: Similar but invokes the macro if the two numbers popped are not equal.

?
: Reads a line from the terminal and executes it. This command allows a macro to request input from the user.

q
: exits from a macro and also from the macro which invoked it. If called from the top level, or from a macro which was called directly from the top level, the q command will cause dc to exit.

Q
: Pops a value off the stack and uses it as a count of levels of

macro execution to be exited. Thus, 3Q exits three levels. The Q command will never cause dc to exit.

Status Inquiry

Z Pops a value off the stack, calculates the number of digits it has (or number of characters, if it is a string) and pushes that number. The digit count for a number does not include any leading zeros, even if those appear to the right of the radix point.

X Pops a value off the stack, calculates the number of fraction digits it has, and pushes that number. For a string, the value pushed is 0.

z Pushes the current stack depth: the number of objects on the stack before the execution of the z command.

Miscellaneous

! Will run the rest of the line as a system command. Note that parsing of the !<, !=, and !> commands take precedence, so if you want to run a command starting with <, =, or > you will need to add a space after the !.

Will interpret the rest of the line as a comment.

:r Will pop the top two values off of the stack. The old second-to-top value will be stored in the array r, indexed by the old top-of-stack value.

;r Pops the top-of-stack and uses it as an index into the array r. The selected value is then pushed onto the stack.

Note that each stacked instance of a register has its own array associated with it. Thus 1 0:a 0Sa 2 0:a La 0;ap will print 1, because the 2 was stored in an instance of 0:a that was later popped.

BUGS

Email bug reports to bug-dc@gnu.org.

Vedlegg – int21h

AH = 01h - READ CHARACTER FROM STANDARD INPUT, WITH ECHO

Return: AL = character read

AH = 09h - WRITE STRING TO STANDARD OUTPUT

Entry: DS:DX -> '$'-terminated string

Return: AL = 24h

AH = 4Ch - "EXIT" - TERMINATE WITH RETURN CODE

Entry: AL = return code

Return: never returns

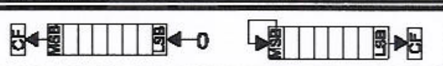
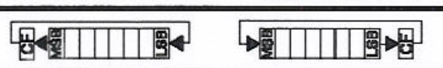
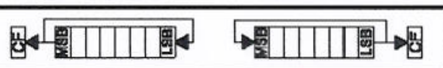
CodeTable 1/2

TRANSFER

Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
MOV	Move (copy)	MOV Dest,Source	Dest:=Source									
XCHG	Exchange	XCHG Op1,Op2	Op1:=Op2 , Op2:=Op1									
STC	Set Carry	STC	CF:=1									1
CLC	Clear Carry	CLC	CF:=0									0
CMC	Complement Carry	CMC	CF:= ¬CF									±
STD	Set Direction	STD	DF:=1 (string op's downwards)		1							
CLD	Clear Direction	CLD	DF:=0 (string op's upwards)		0							
STI	Set Interrupt	STI	IF:=1			1						
CLI	Clear Interrupt	CLI	IF:=0			0						
PUSH	Push onto stack	PUSH Source	DEC SP, [SP]:=Source									
PUSHF	Push flags	PUSHF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL									
PUSHA	Push all general registers	PUSHA	AX, CX, DX, BX, SP, BP, SI, DI									
POP	Pop from stack	POP Dest	Dest:=[SP], INC SP									
POPF	Pop flags	POPF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL	±	±	±	±	±	±	±	±	±
POPA	Pop all general registers	POPA	DI, SI, BP, SP, BX, DX, CX, AX									
CBW	Convert byte to word	CBW	AX:=AL (signed)									
CWD	Convert word to double	CWD	DX:AX:=AX (signed)	±				±	±	±	±	±
CWDE	Conv word extended double	CWDE 386	EAX:=AX (signed)									
IN *i*	Input	IN Dest, Port	AL/AX/EAX := byte/word/double of specified port									
OUT *i*	Output	OUT Port, Source	Byte/word/double of specified port := AL/AX/EAX									

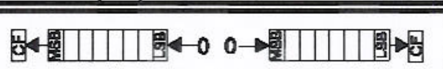
i for more information see instruction specifications   Flags: ±=affected by this instruction ?=undefined after this instruction

ARITHMETIC

Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
ADD	Add	ADD Dest,Source	Dest:=Dest+Source	±				±	±	±	±	±
ADC	Add with Carry	ADC Dest,Source	Dest:=Dest+Source+CF	±				±	±	±	±	±
SUB	Subtract	SUB Dest,Source	Dest:=Dest-Source	±				±	±	±	±	±
SBB	Subtract with borrow	SBB Dest,Source	Dest:=Dest-(Source+CF)	±				±	±	±	±	±
DIV	Divide (unsigned)	DIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
DIV	Divide (unsigned)	DIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
DIV 386	Divide (unsigned)	DIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
IDIV 386	Signed Integer Divide	IDIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
MUL	Multiply (unsigned)	MUL Op	Op=byte: AX:=AL*Op if AH=0 ♦	±				?	?	?	?	±
MUL	Multiply (unsigned)	MUL Op	Op=word: DX:AX:=AX*Op if DX=0 ♦	±				?	?	?	?	±
MUL 386	Multiply (unsigned)	MUL Op	Op=double: EDX:EAX:=EAX*Op if EDX=0 ♦	±				?	?	?	?	±
IMUL *i*	Signed Integer Multiply	IMUL Op	Op=byte: AX:=AL*Op if AL sufficient ♦	±				?	?	?	?	±
IMUL	Signed Integer Multiply	IMUL Op	Op=word: DX:AX:=AX*Op if AX sufficient ♦	±				?	?	?	?	±
IMUL 386	Signed Integer Multiply	IMUL Op	Op=double: EDX:EAX:=EAX*Op if EAX sufficient ♦	±				?	?	?	?	±
INC	Increment	INC Op	Op:=Op+1 (Carry not affected !)	±				±	±	±	±	
DEC	Decrement	DEC Op	Op:=Op-1 (Carry not affected !)	±				±	±	±	±	
CMP	Compare	CMP Op1,Op2	Op1-Op2	±				±	±	±	±	±
SAL	Shift arithmetic left (≡ SHL)	SAL Op,Quantity		*i*				±	±	?	±	±
SAR	Shift arithmetic right	SAR Op,Quantity		*i*				±	±	?	±	±
RCL	Rotate left through Carry	RCL Op,Quantity		*i*								±
RCR	Rotate right through Carry	RCR Op,Quantity		*i*								±
ROL	Rotate left	ROL Op,Quantity		*i*								±
ROR	Rotate right	ROR Op,Quantity		*i*								±

i for more information see instruction specifications   ♦ then CF:=0, OF:=0 else CF:=1, OF:=1

LOGIC

Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
NEG	Negate (two-complement)	NEG Op	Op:=0-Op if Op=0 then CF:=0 else CF:=1	±				±	±	±	±	±
NOT	Invert each bit	NOT Op	Op:=¬Op (invert each bit)									
AND	Logical and	AND Dest,Source	Dest:=Dest∧Source	0				±	±	?	±	0
OR	Logical or	OR Dest,Source	Dest:=Dest∨Source	0				±	±	?	±	0
XOR	Logical exclusive or	XOR Dest,Source	Dest:=Dest (exor) Source	0				±	±	?	±	0
SHL	Shift logical left (≡ SAL)	SHL Op,Quantity		*i*				±	±	?	±	±
SHR	Shift logical right	SHR Op,Quantity		*i*				±	±	?	±	±

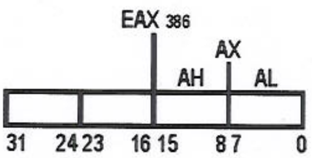
CodeTable 2/2

MISC				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
NOP	No operation	NOP	No operation									
LEA	Load effective address	LEA Dest,Source	Dest := address of Source									
INT	Interrupt	INT Nr	interrupts current program, runs spec. int-program			0	0					

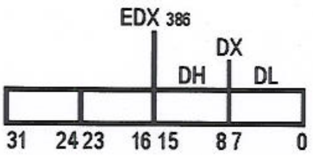
JUMPS (flags remain unchanged)							
Name	Comment	Code	Operation	Name	Comment	Code	Operation
CALL	Call subroutine	CALL Proc		RET	Return from subroutine	RET	
JMP	Jump	JMP Dest					
JE	Jump if Equal	JE Dest	(≡ JZ)	JNE	Jump if not Equal	JNE Dest	(≡ JNZ)
JZ	Jump if Zero	JZ Dest	(≡ JE)	JNZ	Jump if not Zero	JNZ Dest	(≡ JNE)
JCXZ	Jump if CX Zero	JCXZ Dest		JECXZ	Jump if ECX Zero	JECXZ Dest	386
JP	Jump if Parity (Parity Even)	JP Dest	(≡ JPE)	JNP	Jump if no Parity (Parity Odd)	JNP Dest	(≡ JPO)
JPE	Jump if Parity Even	JPE Dest	(≡ JP)	JPO	Jump if Parity Odd	JPO Dest	(≡ JNP)

JUMPS Unsigned (Cardinal)				JUMPS Signed (Integer)			
JA	Jump if Above	JA Dest	(≡ JNBE)	JG	Jump if Greater	JG Dest	(≡ JNLE)
JAE	Jump if Above or Equal	JAE Dest	(≡ JNB ≡ JNC)	JGE	Jump if Greater or Equal	JGE Dest	(≡ JNL)
JB	Jump if Below	JB Dest	(≡ JNAE ≡ JC)	JL	Jump if Less	JL Dest	(≡ JNGE)
JBE	Jump if Below or Equal	JBE Dest	(≡ JNA)	JLE	Jump if Less or Equal	JLE Dest	(≡ JNG)
JNA	Jump if not Above	JNA Dest	(≡ JBE)	JNG	Jump if not Greater	JNG Dest	(≡ JLE)
JNAE	Jump if not Above or Equal	JNAE Dest	(≡ JB ≡ JC)	JNGE	Jump if not Greater or Equal	JNGE Dest	(≡ JL)
JNB	Jump if not Below	JNB Dest	(≡ JAE ≡ JNC)	JNL	Jump if not Less	JNL Dest	(≡ JGE)
JNBE	Jump if not Below or Equal	JNBE Dest	(≡ JA)	JNLE	Jump if not Less or Equal	JNLE Dest	(≡ JG)
JC	Jump if Carry	JC Dest		JO	Jump if Overflow	JO Dest	
JNC	Jump if no Carry	JNC Dest		JNO	Jump if no Overflow	JNO Dest	
				JS	Jump if Sign (= negative)	JS Dest	
				JNS	Jump if no Sign (= positive)	JNS Dest	

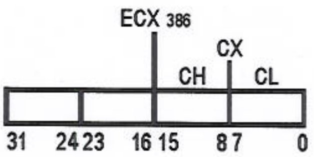
General Registers:



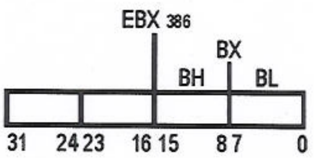
Accumulator



Data mul, div, IO



Count loop, shift



BaseX data ptr

Example:

```
        .DOSSEG              ; Demo program
        .MODEL SMALL
        .STACK 1024
Two     EQU 2                ; Const
        .DATA
VarB    DB ?                 ; define Byte, any value
VarW    DW 1010b             ; define Word, binary
VarW2   DW 257               ; define Word, decimal
VarD    DD 0AFFFFh           ; define Doubleword, hex
S       DB "Hello !",0       ; define String
        .CODE
main:   MOV AX,DGROUP        ; resolved by linker
        MOV DS,AX            ; init datasegment reg
        MOV [VarB],42        ; init VarB
        MOV [VarD],-7        ; set VarD
        MOV BX,Offset[S]     ; addr of "H" of "Hello !"
        MOV AX,[VarW]        ; get value into accumulator
        ADD AX,[VarW2]       ; add VarW2 to AX
        MOV [VarW2],AX       ; store AX in VarW2
        MOV AX,4C00h         ; back to system
        INT 21h
        END main
```

KEEP IT SIMPLE STUPID (AND SHORT!) KISS

Flags: - - - - O D I T S Z - A - P - C

Control Flags (how instructions are carried out):

- D: Direction — 1 = string op's process down from high to low address
- I: Interrupt — whether interrupts can occur. 1= enabled
- T: Trap — single step for debugging

Status Flags (result of operations):

- C: Carry — result of unsigned op. is too large or below zero. 1 = carry/borrow
- O: Overflow — result of signed op. is too large or small. 1 = overflow/underflow
- S: Sign — sign of result. Reasonable for Integer only. 1 = neg. / 0 = pos.
- Z: Zero — result of operation is zero. 1 = zero
- A: Aux. carry — similar to Carry but restricted to the low nibble only
- P: Parity — 1 = result has even number of set bits