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

Talet på oppgåver: 5
Talet på sider: 12 sider inkludert forside og vedlegg
Vedlegg: Intel assemblyfunksjonar
          DOS int21h
          Utvalt bsd-kommando
Hjelpemiddel: Geir Maribu: “Praktisk Linux”
Tid: 09.00-13.00 (4 timar)
Målform: Nynorsk
Sensor: Ingen
Faglærar: Atle Geitung

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

Side 1
Oppgåve 1

a) Kva er eit operativsystem?

b) Anta at vi på eit gitt tidspunkt har tre prosesser P₁, P₂ og P₃ køyrande. Dei treng høvesvis 5ms, 7ms og 3ms meir prosessor-tid (CPU-tid) for å bli fullført. Prosessoren kan berre køyre ein prosess om gangen, men prosessane kan brytast og prosessoren vil køyre ein prosess i 2ms kvar gang ein prosess slepp til.

Forklar korleis planleggingsalgoritmane Round Robin (RR) og Shortest Job First (SJF) fungerer og vis korleis prosessane P₁, P₂ og P₃ blir utført ferdig ved bruk av dei to planleggingsalgoritmane.

c) Kva er ein kritisk region og kva er føremålet med kritiske regioner (critical sections)? Forklar og korleis ein kan implementere ein kritisk region.

d) Kva fire kriterier må vere oppfylte før det skal oppstå ein vranglås (deadlock)? Forklar korleis Bankierens (Banker’s) algoritme kan hjelpe oss å unngå vranglås.

Oppgåve 2

a) Oppdeling av hukommelse gjera ved segmentinndeling (segmentation) og sideinndeling (paging). Forklar dei to inndelingsmetodane og kva ulemper og fordeler dei har. Fragmentering er eit sentralt omgrep her.

b) Forklar kva ein meiner med virtuell hukommelse og korfor virtuell hukommelse er nødvendig i dagens operativsystem.

c) La oss anta at vi har eit køyrande program som skal hente data. Adressa til dataa er ein adresse i den virtuelle hukommelsen som operativsystemet tilbyr. Forklar korleis prosessoren får henta data frå den fysiske primærhukommelsen (main memory). Eller sagt med andre ord, kva skjer når adresserte data som kan vere lagra i sekundærlager blir gjort tilgjengeleg i primærhukommelse slik at prosessoren kan bruke dei?
Oppgave 3

a) Gitt følgjande skalprogram:

```bash
#!/bin/bash
E=1
B=16
if [ -z "$1" ]
then
echo "Melding"
exit $E
fi
echo """$1" "$B" o p" | dc
exit 0
```

Kva gjer programmet?

b) Koden blir lagt inn i ein tekstfil med namnet program.sh. Kva må vi gjere for å kunne køyre programmet frå kommandolina?

c) Lag eit skalprogram som heiter listbrukere.sh. Dette skal lese filen /etc/passwd og hente ut brukerane i systemet frå denne filen. /etc/passwd er ein tekstfil og inneholder ein del kolonnar som er skilt med kolon, ein line for kvar bruker og første kolonne er brukarnamnet. Eksempel på ei line:

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

Skalprogrammet som du skal lage, skal først skrive ut alle brukarnamna og til slutt talet på brukara til stdout.
Oppgave 4

Til høgre er assemblykoden til eit program lista. Linenummer er tatt med for å forenkle kommentering av programmet

a) Gå gjennom linene i programkoden og forklar kva dei gjør.

b) Forklar kva programmet gjør.

c) Oversett følgjande Java-liknande algoritme til assembly-kode:

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

Variabelen ax er registeret ax.

cr equ 13 ;Carriage return
lf equ 10 ;Line feed

segment stack stack
resb 64
stacktop:

segment data
msg1 db "Inndata: ","\$"
msg2 db cr,lf,"ja","\$"
msg3 db cr,lf,"nei","\$"

segment code

..start:
mov ax,data
mov ds,ax

main:
mov ah,09h
mov dx,msg1
int 21h

mov ah,01h
int 21h
and al,01h
cmp al,00h
je Nei

Ja:
mov ah,09h
mov dx,msg2
int 21h
jmp Ferdig

Nei:
mov ah,09h
mov dx,msg3
int 21h

Ferdig:
mov ah,4Ch
int 21h
end main
Oppgave 5

Figuren til høyre viser ein digital terning bestående av 7 diodar som blir styrt av ein dekodar. Den har tre innganger, A, B og C. Dekodaren skal dekode verdiene 1, 2, 3, 4, 5 og 6 slik at riktige diodar lyser og viser verdien til terningen på same måte som ein vanleg terning, sjå figuren under fordekoding. Talet representeras digitalt med sifra CBA og andre verdier enn 1, 2, 3, 4, 5 og 6 vil ikkje førekome. Til dømes vil ein einar vere representert som CBA = 001 og dioden g i figuren vil lyse.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

a) Sett opp sanningstabellen for heile dekoderan (for både a, b, c, d, e, f og g).

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

c) Finn enklast mogeleg boolsk uttrykk for a, b, c og g.

d) Teikn kretsløysninga for dei boolske uttrykka frå c).

Lykke til og God Jul

Ale Godtung
Vedlegg – bash (kommando som ikkje er i læreboka)

NAME
dc - an arbitrary precision calculator

SYNOPSIS
dc [-V] [--version] [-h] [--help]
    [-e scriptexpression] [-e 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 Z 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.
Pops off the value on top of the stack. If it is 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]dxssxx[qs]sq[Lsd0>qaplxx] dxssxx0sqLqsxLxLk+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 d1d/ 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 SdSm_lnd/ lndd% 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.

l 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 exponentiated, which should be an integer. For small integers this is like the sequence Sm^Im%, 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, "^d*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 SaSaBlAb.)

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, [lp]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; [lp]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
## TRANSFER

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move (copy)</td>
<td>MOV Dest, Source</td>
<td>Dest=Source</td>
<td></td>
</tr>
<tr>
<td>XCHG</td>
<td>Exchange</td>
<td>XCHG Op1,Op2</td>
<td>O1:Op2, O2:Op1</td>
<td></td>
</tr>
<tr>
<td>STC</td>
<td>Set Carry</td>
<td>STC</td>
<td>CF:1</td>
<td>1</td>
</tr>
<tr>
<td>CLC</td>
<td>Clear Carry</td>
<td>CLC</td>
<td>CF:0</td>
<td>0</td>
</tr>
<tr>
<td>CMC</td>
<td>Complement Carry</td>
<td>CMC</td>
<td>CF: -CF</td>
<td></td>
</tr>
<tr>
<td>STD</td>
<td>Set Direction</td>
<td>STD</td>
<td>DF:1 (string op's downwards)</td>
<td>1</td>
</tr>
<tr>
<td>CLD</td>
<td>Clear Direction</td>
<td>CLD</td>
<td>DF:0 (string op's upwards)</td>
<td>0</td>
</tr>
<tr>
<td>STI</td>
<td>Set Interrupt</td>
<td>STI</td>
<td>IF:1</td>
<td>1</td>
</tr>
<tr>
<td>CLI</td>
<td>Clear Interrupt</td>
<td>CLI</td>
<td>IF:0</td>
<td>0</td>
</tr>
<tr>
<td>PUSH</td>
<td>Push onto stack</td>
<td>PUSH Source</td>
<td>DEC SP, [SP]=Source</td>
<td></td>
</tr>
<tr>
<td>PUSHF</td>
<td>Push flags</td>
<td>PUSHF</td>
<td>O, D, I, T, S, Z, A, P, C</td>
<td>286+ also NT, IOPL</td>
</tr>
<tr>
<td>PUSHA</td>
<td>Push all general registers</td>
<td>PUSHA</td>
<td>AX, CX, DX, BX, SP, BP, SI, DI</td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>Pop from stack</td>
<td>POP Dest</td>
<td>Dest=[SP], INC SP</td>
<td></td>
</tr>
<tr>
<td>POPF</td>
<td>Pop flags</td>
<td>POPF</td>
<td>O, D, I, T, S, Z, A, P, C</td>
<td>286+ also NT, IOPL</td>
</tr>
<tr>
<td>POPA</td>
<td>Pop all general registers</td>
<td>POPA</td>
<td>DI, SI, BP, SP, BX, DX, CX, AX</td>
<td></td>
</tr>
<tr>
<td>CBW</td>
<td>Convert byte to word</td>
<td>CBW</td>
<td>AX=AL (signed)</td>
<td></td>
</tr>
<tr>
<td>CWD</td>
<td>Convert word to double</td>
<td>CWD</td>
<td>DX:AX=AX (signed)</td>
<td></td>
</tr>
<tr>
<td>CWDE</td>
<td>Convert word extended double</td>
<td>CWDE 386</td>
<td>EAX=AX (signed)</td>
<td></td>
</tr>
</tbody>
</table>

### ARITHMETIC

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
<td>ADD Dest, Source</td>
<td>Dest=Dest+Source</td>
<td></td>
</tr>
<tr>
<td>ADC</td>
<td>Add with Carry</td>
<td>ADC Dest, Source</td>
<td>Dest=Dest+Source+CF</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>SUB Dest, Source</td>
<td>Dest=Dest-Source</td>
<td></td>
</tr>
<tr>
<td>SBB</td>
<td>Subtract with borrow</td>
<td>SBB Dest, Source</td>
<td>Dest=Dest-Source-Source</td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>Divide (unsigned)</td>
<td>DIV Op</td>
<td>Op:byte: AL=AX / Op</td>
<td>Art=Rest ?</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide (unsigned)</td>
<td>DIV Op</td>
<td>Op:word: AX=DX AX / Op</td>
<td>DX=Rest ?</td>
</tr>
<tr>
<td>DIV 386</td>
<td>Divide (unsigned)</td>
<td>DIV Op</td>
<td>Op:doubleword: EAX=DX EAX / Op</td>
<td>EDX=Rest ?</td>
</tr>
<tr>
<td>IDIV</td>
<td>Signed Integer Divide</td>
<td>IDIV Op</td>
<td>Op:byte: AL=AX / Op</td>
<td>Art=Rest ?</td>
</tr>
<tr>
<td>IDIV</td>
<td>Signed Integer Divide</td>
<td>IDIV Op</td>
<td>Op:word: AX=DX AX / Op</td>
<td>DX=Rest ?</td>
</tr>
<tr>
<td>IDIV 386</td>
<td>Signed Integer Divide</td>
<td>IDIV Op</td>
<td>Op:doubleword: EAX=DX EAX / Op</td>
<td>EDX=Rest ?</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply (unsigned)</td>
<td>MUL Op</td>
<td>Op:byte: AX=AL*Op</td>
<td>%A=0</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply (unsigned)</td>
<td>MUL Op</td>
<td>Op:word: DX:AX=AX*Op</td>
<td>DF=0</td>
</tr>
<tr>
<td>MUL 386</td>
<td>Multiply (unsigned)</td>
<td>MUL Op</td>
<td>Op:doubleword: EAX=DX EAX*Op</td>
<td>ifDF=0</td>
</tr>
<tr>
<td>IMUL</td>
<td>Signed integer Multiply</td>
<td>IMUL Op</td>
<td>Op:byte: AX=AL*Op</td>
<td>%A=0</td>
</tr>
<tr>
<td>IMUL</td>
<td>Signed integer Multiply</td>
<td>IMUL Op</td>
<td>Op:word: DX:AX=AX*Op</td>
<td>%A=0</td>
</tr>
<tr>
<td>IMUL 386</td>
<td>Signed integer Multiply</td>
<td>IMUL Op</td>
<td>Op:doubleword: EAX=DX EAX*Op</td>
<td>ifDF=0</td>
</tr>
<tr>
<td>INC</td>
<td>Increment</td>
<td>INC Op</td>
<td>Op:Op+1 (Carry not affected)</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>Decrement</td>
<td>DEC Op</td>
<td>Op:Op-1 (Carry not affected)</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>CMP Op1,Op2</td>
<td>Op1:Op2</td>
<td></td>
</tr>
<tr>
<td>SAL</td>
<td>Shift arithmetic left (=SHL)</td>
<td>SAL Op, Quantity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR</td>
<td>Shift arithmetic right</td>
<td>SAR Op, Quantity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL</td>
<td>Rotate left through Carry</td>
<td>RCL Op, Quantity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCR</td>
<td>Rotate right through Carry</td>
<td>RCR Op, Quantity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>Rotate left</td>
<td>ROL Op, Quantity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate right</td>
<td>ROR Op, Quantity</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## LOGIC

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG</td>
<td>Negate (two-complement)</td>
<td>NEG Op</td>
<td>Op=0:Op</td>
<td>ifOp=0 then CF=0 else CF=1</td>
</tr>
<tr>
<td>NOT</td>
<td>Invert each bit</td>
<td>NOT Op</td>
<td>Op= ~Op (invert each bit)</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
<td>AND Dest, Source</td>
<td>Dest=Dest, Source</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>Logical or</td>
<td>OR Dest, Source</td>
<td>Dest=Dest, Source</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>Logical exclusive or</td>
<td>XOR Dest, Source</td>
<td>Dest=Dest (exor) Source</td>
<td></td>
</tr>
<tr>
<td>SHL</td>
<td>Shift logical left (=SAL)</td>
<td>SHL Op, Quantity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR</td>
<td>Shift logical right</td>
<td>SHR Op, Quantity</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* for more information see instruction specifications

* then CF=0, OF=0 else CF=1, OF=1

---

This page may be freely distributed without cost provided it is not changed. Send comments to roger@jegerlehner.ch - www.jegerlehner.ch/intel/
## CodeTable 2/2

### JUMPS (flags remain unchanged)

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Call subroutine</td>
<td>CALL Proc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
<td>JMP Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JE</td>
<td>Jump if Equal</td>
<td>JE Dest ((= JZ))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JZ</td>
<td>Jump if Zero</td>
<td>JZ Dest ((= JE))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JCXZ</td>
<td>Jump if CX Zero</td>
<td>JCXZ Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JP</td>
<td>Jump if Parity (Parity Even)</td>
<td>JP Dest ((= JPE))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPE</td>
<td>Jump if Parity Even</td>
<td>JPE Dest ((= JP))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNE</td>
<td>Jump if Not Equal</td>
<td>JNE Dest ((= JNZ))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump if Not Zero</td>
<td>JNZ Dest ((= JNE))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JECXZ</td>
<td>Jump if ECX Zero</td>
<td>JECXZ Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNP</td>
<td>Jump if No Parity (Parity Odd)</td>
<td>JNP Dest ((= JPO))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPO</td>
<td>Jump if Parity Odd</td>
<td>JPO Dest ((= JNP))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### JUMPS Unsigned (Cardinal)

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>Jump if Above</td>
<td>JA Dest ((= JNBE))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if Above or Equal</td>
<td>JAE Dest ((= JNB = JNC))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JB</td>
<td>Jump if Below</td>
<td>JB Dest ((= JNAE = JC))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if Below or Equal</td>
<td>JBE Dest ((= JNE))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if Not Above</td>
<td>JNA Dest ((= JBE))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNAB</td>
<td>Jump if Not Above or Equal</td>
<td>JNAB Dest ((= JB = JC))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if Not Below</td>
<td>JNB Dest ((= JAE = JNC))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNBL</td>
<td>Jump if Not Below or Equal</td>
<td>JNBL Dest ((= JA))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JC</td>
<td>Jump if Carry</td>
<td>JC Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if No Carry</td>
<td>JNC Dest</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### JUMPS Signed (Integer)

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>JG</td>
<td>Jump if Greater</td>
<td>JG Dest ((= JNLE))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JGE</td>
<td>Jump if Greater or Equal</td>
<td>JGE Dest ((= JNL))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if Less</td>
<td>JLE Dest ((= JNGE))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JLS</td>
<td>Jump if Less</td>
<td>JLS Dest ((= JJE))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if Not Less</td>
<td>JNL Dest ((= JGE))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNLE</td>
<td>Jump if Not Less or Equal</td>
<td>JNLE Dest ((= JG))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JO</td>
<td>Jump if Overflow</td>
<td>JO Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if No Overflow</td>
<td>JNO Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JS</td>
<td>Jump if Sign ((= negative))</td>
<td>JS Dest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if No Sign ((= positive))</td>
<td>JNS Dest</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### General Registers:

- **EAX**: Accumulator
  - AX: 31 24 16 8 2 0
  - AH, AL
- **EDX**: Data mul, div, IO
  - DX: 31 24 16 8 2 0
  - DH, DL
- **ECX**: Count loop, shift
  - CX: 31 24 16 8 2 0
  - CH, CL
- **EBX**: BaseX data ptr
  - BX: 31 24 16 8 2 0
  - BH, BL

### Flags:

- C: Carry Flags (how instructions are carried out):
  - C: Carry
  - O: Overflow
  - S: Sign
  - Z: Zero
  - A: Aux. carry
  - P: Parity

- Status Flags (result of operations):
  - C: Carry
  - O: Overflow
  - S: Sign
  - Z: Zero
  - A: Aux. carry
  - P: Parity

### Example:

- `DOSSEG MODEL SMALL .STACK 1024` : Demo program
- `EQU 2 ; Const` : Const
- `DATA` : Define Byte, any value
- ` VarB DB ? ` : Define Word, binary
- ` VarW DW 1010b ` : Define Word, decimal
- ` VarW2 DW 257 ` : Define Doubleword, hex
- ` S DB "Hello",0 ` : Define String
- ` CODE` : Code
- ` MOV AX,DXGROUP` : Resolved by linker
- ` MOV DS,AX` : init defsegment 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
- ` END main` : End main

This page may be freely distributed without cost provided it is not changed. Send comments to roger@jegerlehner.ch - www.jegerlehner.ch/intel/