<table>
<thead>
<tr>
<th>Item</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eksamen i</td>
<td>DAT103/TOD077 – Datamaskiner og operativsystemer</td>
</tr>
<tr>
<td>Klasse</td>
<td>2. klasse Data / Inf</td>
</tr>
<tr>
<td>Dato</td>
<td>17. desember 2013</td>
</tr>
<tr>
<td>Antall oppgaver</td>
<td>5</td>
</tr>
<tr>
<td>Antall sider</td>
<td>12 sider inkludert forside og vedlegg</td>
</tr>
<tr>
<td>Vedlegg</td>
<td>Intel assemblyfunksjoner</td>
</tr>
<tr>
<td></td>
<td>DOS int21h</td>
</tr>
<tr>
<td></td>
<td>Utvalgt bsd-kommando</td>
</tr>
<tr>
<td>Hjelpemidler</td>
<td>Geir Maribu: “Praktisk Linux”</td>
</tr>
<tr>
<td>Tid</td>
<td>09.00-13.00 (4 timer)</td>
</tr>
<tr>
<td>Målform</td>
<td>Norsk - bokmål</td>
</tr>
<tr>
<td>Sensor</td>
<td>Ingen</td>
</tr>
<tr>
<td>Faglærer</td>
<td>Atle Geitung</td>
</tr>
</tbody>
</table>

Postboks 7030, 5020 Bergen. Tlf. 55 58 75 00, Fax 55 58 77 90
Besøksadr.: Nygårdsdt. 112, Bergen
Side 1
Oppgave 1

a) Hva er et operativsystem?

b) Anta at vi på et gitt tidspunkt har tre prosesser P₁, P₂ og P₃ 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 P₁, P₂ og P₃ 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:

```bash
#!/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 stdout.
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:

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

Variabelen ax er registeret ax.
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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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!

Åke Geitang

Side 5
**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 ‘-’ or ‘-’ 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 [1s*Ssqd0>x [256=5sd0<x] dsxxsx[q]S[lsdq>qaPLxx] dsxxsx0qLqslqL+L+L+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 lnd/ 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 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, "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 num-
ber 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 lp
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>r will invoke
register a's contents and 2 1>r 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.

? Read 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.

O

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 OSa 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>Op1=Op2, Op2=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>DEG SP, [SP]=Source</td>
<td></td>
</tr>
<tr>
<td>PUSHF</td>
<td>Push flags</td>
<td>PUSHF</td>
<td>O, D, I, T, S, A, P, C, 286+: also NT, I/OPL</td>
<td></td>
</tr>
<tr>
<td>PUSHA</td>
<td>Push all 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, A, P, C, 286+: also NT, I/OPL</td>
<td></td>
</tr>
<tr>
<td>POPA</td>
<td>Pop all 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>Conv word extended double</td>
<td>CWDE</td>
<td>EAX=AX (signed)</td>
<td>±</td>
</tr>
</tbody>
</table>

### IN & Output

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Input</td>
<td>IN Dest, Port</td>
<td>AL/AX/EAX = byte/word/double of specified port</td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td>Output</td>
<td>OUT Port, Source</td>
<td>Byte/word/double of specified port := AL/AX/EAX</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-CF)</td>
<td>±</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide (unsigned)</td>
<td>DIV Op</td>
<td>O:byte=AX/Op A1=Rest</td>
<td>±</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide (unsigned)</td>
<td>DIV Op</td>
<td>O:word=DX:AX/Op DX=Rest</td>
<td>±</td>
</tr>
<tr>
<td>DIV 386</td>
<td>Divide (unsigned)</td>
<td>DIV Op</td>
<td>O:double=EX:EDX:AX/Op EDX=Rest</td>
<td>±</td>
</tr>
<tr>
<td>IDIV</td>
<td>Signed Integer Divide</td>
<td>IDIV Op</td>
<td>O:byte=AL/Op A1=Rest</td>
<td>±</td>
</tr>
<tr>
<td>IDIV</td>
<td>Signed Integer Divide</td>
<td>IDIV Op</td>
<td>O:word=DX:AX/Op DX=Rest</td>
<td>±</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply (unsigned)</td>
<td>MUL Op</td>
<td>O:byte=AL*Op if AH=0</td>
<td>±</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply (unsigned)</td>
<td>MUL Op</td>
<td>O:word=DX:AX*AX/Op if DX=0</td>
<td>±</td>
</tr>
<tr>
<td>MUL 386</td>
<td>Multiply (unsigned)</td>
<td>MUL Op</td>
<td>O:double=EDX:EX:AX*AX/Op if EDX=0</td>
<td>±</td>
</tr>
<tr>
<td>IMUL</td>
<td>Signed Integer Multiply</td>
<td>IMUL Op</td>
<td>O:byte=AL*Op if AH sufficient</td>
<td>±</td>
</tr>
<tr>
<td>IMUL</td>
<td>Signed Integer Multiply</td>
<td>IMUL Op</td>
<td>O:word=DX:AX*AX/Op if AX sufficient</td>
<td>±</td>
</tr>
<tr>
<td>IMUL 386</td>
<td>Signed Integer Multiply</td>
<td>IMUL Op</td>
<td>O:double=EDX:EX:AX*AX/Op if EAX sufficient</td>
<td>±</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>
</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 if Op=0 then CF=0 else CF=1</td>
<td>±</td>
</tr>
<tr>
<td>NOT</td>
<td>Invert each bit</td>
<td>NOT Op</td>
<td>Op:=Op (inverted bit)</td>
<td>±</td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
<td>AND Dest,Source</td>
<td>Dest=Dest+Source</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>Logical or</td>
<td>OR Dest,Source</td>
<td>Dest=Dest-Source</td>
<td>0</td>
</tr>
<tr>
<td>XOR</td>
<td>Logical exclusive or</td>
<td>XOR Dest,Source</td>
<td>Dest=(dest xor Source)</td>
<td>0</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>
### CodeTable 2/2

#### MISC
<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>NOP</td>
<td>No operation</td>
<td>O D I T S Z A P C</td>
</tr>
<tr>
<td>LEA</td>
<td>Load effective address</td>
<td>LEA Dest, Source</td>
<td>Dest := address of Source</td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>Interrupt</td>
<td>INT Nr</td>
<td></td>
<td>interrupts current program, runs spec. int-program</td>
</tr>
</tbody>
</table>

#### JUMPS (flags remain unchanged)
<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Call subroutine</td>
<td>CALL Proc</td>
<td></td>
<td>RET</td>
<td>Return from subroutine</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
<td>JMP Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JE</td>
<td>Jump if Equal</td>
<td>JE Dest</td>
<td>(e JC)</td>
<td>JNE</td>
<td>Jump if not Equal</td>
<td>JNE Dest</td>
<td>(e JNZ)</td>
</tr>
<tr>
<td>JZ</td>
<td>Jump if Zero</td>
<td>JZ Dest</td>
<td>(e JE)</td>
<td>JNZ</td>
<td>Jump if not Zero</td>
<td>JNZ Dest</td>
<td>(e JNE)</td>
</tr>
<tr>
<td>JCCXZ</td>
<td>Jump if CX Zero</td>
<td>JCCXZ Dest</td>
<td></td>
<td>JCCXZ</td>
<td>Jump if ECX Zero</td>
<td>JCCXZ Dest</td>
<td>3rb</td>
</tr>
<tr>
<td>JPE</td>
<td>Jump if Parity (Parity Even)</td>
<td>JPE Dest</td>
<td>(e JPE)</td>
<td>JPO</td>
<td>Jump if no Parity (Parity Odd)</td>
<td>JPE Dest</td>
<td>(e JPO)</td>
</tr>
<tr>
<td>JPE</td>
<td>Jump if Parity Even</td>
<td>JPE Dest</td>
<td>(e JPE)</td>
<td>JPO</td>
<td>Jump if Parity Odd</td>
<td>JPE Dest</td>
<td>(e JPO)</td>
</tr>
</tbody>
</table>

#### JUMPS Unsigned (Cardinal)
<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>Jump if Above</td>
<td>JA Dest</td>
<td>(e JNB)</td>
<td>JG</td>
<td>Jump if Greater</td>
<td>JG Dest</td>
<td>(e JNL)</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if Above or Equal</td>
<td>JAE Dest</td>
<td>(e JNC)</td>
<td>JGE</td>
<td>Jump if Greater or Equal</td>
<td>JGE Dest</td>
<td>(e JNL)</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if Below</td>
<td>JB Dest</td>
<td>(e JNAE = JC)</td>
<td>JL</td>
<td>Jump if Less</td>
<td>JL Dest</td>
<td>(e JNGE)</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if Below or Equal</td>
<td>JBE Dest</td>
<td>(e JNAE = JE)</td>
<td>JLE</td>
<td>Jump if Less or Equal</td>
<td>JLE Dest</td>
<td>(e JNGE)</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if not Above</td>
<td>JNA Dest</td>
<td>(e JBE = JC)</td>
<td>JNG</td>
<td>Jump if not Greater</td>
<td>JNG Dest</td>
<td>(e JLE)</td>
</tr>
<tr>
<td>JNAE</td>
<td>Jump if not Above or Equal</td>
<td>JNAE Dest</td>
<td>(e JBE = JC)</td>
<td>JNGE</td>
<td>Jump if not Greater or Equal</td>
<td>JNGE Dest</td>
<td>(e JLE)</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not Below</td>
<td>JNB Dest</td>
<td>(e JNAE = JNC)</td>
<td>JNL</td>
<td>Jump if Less</td>
<td>JNL Dest</td>
<td>(e JGE)</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not Below or Equal</td>
<td>JNB Dest</td>
<td>(e JNAE = JNC)</td>
<td>JNLE</td>
<td>Jump if Less or Equal</td>
<td>JNLE Dest</td>
<td>(e JGE)</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if Carry</td>
<td>JNC Dest</td>
<td></td>
<td>JO</td>
<td>Jump if Overflow</td>
<td>JO Dest</td>
<td></td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if no Carry</td>
<td>JNC Dest</td>
<td></td>
<td>JNO</td>
<td>Jump if no Overflow</td>
<td>JNO Dest</td>
<td></td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if Sign (negative)</td>
<td>JNC Dest</td>
<td></td>
<td>JS</td>
<td>Jump if Sign (negative)</td>
<td>JS Dest</td>
<td></td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if no Sign (positive)</td>
<td>JNC Dest</td>
<td></td>
<td>JNS</td>
<td>Jump if no Sign (positive)</td>
<td>JNS Dest</td>
<td></td>
</tr>
</tbody>
</table>

#### JUMPS Signed (Integer)

#### Example:
```plaintext
.DOSSEG
.MOBELSEG SMALL
.STACK 1024
.Two
.EQU 2
.DATA
.VarB DB ?
.VarW DW 1010b
.VarW2 DW 257
.VarD DD 00000000H
.S DB "Hello", 0
.CODE
.main:
.MOV AX, DXGROUP
.MOV AX, DS:AX
.MOV AX, VarB
.MOV AX, VarW
.MOV AX, VarW2
.MOV AX, VarD
.MOV AX, Offset[VarD]
.MOV AX, AX
.ADD AX, VarW2
.MOV AX, AX
.MOV AX, 4000h
.INT 21h
.END main
```

#### Flags:
- **C**: Carry
- **O**: Overflow
- **S**: Sign
- **Z**: Zero
- **A**: Auxiliary carry
- **P**: Parity

#### Control Flags (how instructions are carried out):
- **D**: Direction
- **I**: Interrupt
- **T**: Trap

#### 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**: Auxiliary carry: similar to Carry but restricted to the low nibble only
- **P**: Parity: 1 = result has even number of set bits

---

**KISS**

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