Z80, the 8-bit Number Cruncher

Author: Andre Adrian
Version: 04.Jun.2011

The Z80 was a 8-bit CPU presented by Zilog in 1976. It was like the Intel 8085 (1976) an improved Intel 8080 design. The Intel 8080 (1974) was an improved Intel 8008. The 8008 (1972) was the MOS version of the Datapoint 2200 CPU (1971) which was implemented with 100 TTL chips. The market prefered the Z80 over the 8085. Home computers like Tandy Radio Shack TRS-80, Sinclair ZX-81 and Amstrad CPC464 used it. The common operating system of that time, CP/M from Digital Research needed an 8080, 8085 or Z80 to run. Today an enhanced 8080 lives on in the Nintendo Gameboy.
The Z80 had many registers. The more successful 6502 had only few registers, but a page zero addressing mode that allowed fast memory access to compensate for the missing registers.
But why is the Z80 the number cruncher compared to the 6502? Well, after the values were loaded into Z80 registers the data-bus was only used to load the op-codes. The 6502 needed much more loads and stores in doing 16-bit or 32-bit arithmetric.
Important note: All examples can be evaluated with Z80DT the "Z80 Development Tool for the MPF-1B" from the Dutch Open university. This Z80 Editor/Assembler/Simulator runs with MS-DOS or an MS-DOS emulator (you run a MS-DOS emulator to run a CPU simulator...).

Table of Contents

The Z80 registers

The Z80 register set is not very orthogonal, that is one specific register can only perform some operations. The registers are:

A	8-bit accumulator
HL	16-bit accumulator and address register, can be used as separate 8-bit registers H, L
DE	16-bit data register, can be used as separate 8-bit registers D, E
BC	16-bit data register, can be used as separate 8-bit registers B, C
IX	16-bit address register
IY	16-bit address register

With one op-code the registers HL, DE, BC could be exchanged with the alternate registers H'L', D'E', B'C'. This was much quicker then load/store from memory. The accumulator A and flags F could be exchanged with alternate A'F' with another op-code. There was an op-code exchange register HL with DE, too.
The data registers B, C, D, E could shift, rotate and bit-test their contents. These registers were no general purpose registers - the result of an arithmetic operation could only be in A or HL register. **Accumulators and data registers behave like integer variables in the programming language C**.
Address registers (index registers) use their contents as a memory reference (pointer). Their purpose is to load/store data from/to an address that is not a constant address. The IX, IY registers can add an offset (displacement) to the address. **Address registers behave like pointer variables in the programming language C**.

Z80 16-bit (short) add

A 16-bit add is the first example. This is the assembler equivalent to "hello world":

```
; LOAD TEST VALUES
        LD      HL,12345        ; LOAD DECIMAL VALUE 12345 INTO REGISTER HL
        LD      DE,7            ; LOAD DECIMAL VALUE 7 INTO REGISTER DE
        CALL    ADD16           ; CALL SUBROUTINE
        HALT                    ; HALT THE Z80 OR THE SIMULATOR

;================================================
; ADD ROUTINE 16+16BIT=16BIT
; HL = HL + DE
; CHANGES FLAGS
;
ADD16:
        ADD     HL,DE   ; 16-BIT ADD OF HL AND DE

; RESULT IS IN HL
        RET             ; RETURN FROM SUBROUTINE

        END             ; PSEUDO OP-CODE FOR ASSEMBLER
```

Z80 32-bit (long) add

Normally 32-bit values were stored upper 16-bit in the alternate register like H'L' and lower 16-bit in the normal register HL. The EXX op-code does exchange the registers B, C, D, E, H, L.

```
; LOAD TEST VALUES
        LD      HL,01213H       ; LOAD HEXADECIMAL 1213 TO HL (LOWER 16-BIT)
        LD      DE,0F000H       ; LOAD HEXADECIMAL F000 TO DE (LOWER 16-BIT)
        EXX
        LD      HL,01011H       ; LOAD HEXADECIMAL 1011 TO H'L' (UPPER 16-BIT)
        LD      DE,0            ; LOAD VALUE 0 TO D'E' (UPPER 16-BIT)
        EXX
        CALL    ADD32
        HALT

;================================================
; ADD ROUTINE 32+32BIT=32BIT
; H'L'HL = H'L'HL + D'E'DE
; CHANGES FLAGS
;
ADD32:
        ADD     HL,DE   ; 16-BIT ADD OF HL AND DE
        EXX
        ADC     HL,DE   ; 16-BIT ADD OF HL AND DE WITH CARRY
        EXX

; RESULT IS IN H'L'HL
        RET

        END
```

The EXX op-code was ugly to write and read, but was the reason why everybody doing number crunching in 8-bit liked the Z80 over the 8085.

Z80 IF-ELSE

To convert a C language if() statement into Z80 assembler is quite nasty in detail:

- First, there is only a 8-bit compare op-code in the Z80. To compare larger numbers a subtraction has to be done.
- Second, the conditional jump op-code is different for unsigned and signed numbers.
- Third, the jump goes to the "else" part of the statement, therefore the jump op-code is the logical opposite of the compare operator
- Fourth, at the end of the if-part an unconditional jump is needed to the end of the IF-ELSE block.

All these nasty details fit into one table. Never write assembler programs without this table!

C test	as subtraction	unsigned numbers "else jump"	signed numbers "else jump"
if (a >= b)	if (a-b >= 0)	JP C or JR C	JP PO
if (a < b)	if (a-b < 0)	JP NC or JR NC	JP PE
if (a == b)	if (a-b == 0)	cascaded JP NZ or JR NZ	cascaded JP NZ or JR NZ

Note: JP PO is jump on overflow. The overflow flag is the carry flag for signed numbers. In the Z80 the overflow and parity flag are the same. The official name of JP PO is jump on parity odd. As they say "just to confuse the russians".
As example see the assembler listing for the following C code fragment.

```
  short HL, DE;
  char A;
```

```
  HL = -2000;
  DE = -1000;
  if (HL >= DE) {
    A = 1;
  } else {
    A = 0;
  }
```

Before you run the assembler code below, what is the value of A at the end of the C language listing? Even if you are sure, you may run a C program to confirm your believes. In the assembler code we use registers A, DE and HL instead of memory locations. The C source is given as comment.

```
;================================================
; IF-ELSE CODE FRAGMENT
;
        LD      HL,-2000        ; HL = -2000;
        LD      DE,-1000        ; DE = -1000;

        AND     A               ; if (HL >= DE) {
        SBC     HL,DE
        JP      PO,ELSE
        LD      A,1             ;   A = 1;
        JR      ENDIF
ELSE:                           ; } else {
        LD      A,0             ;   A = 0;
ENDIF:                          ; }
        HALT

        END
```

Note: The AND A op-code clears the carry flag. Because of this, the SBC HL,DE op-code works like SUB HL,DE - an op-code the Z80 does not have.

The following assembler listing shows the cascaded if (a == b) translation. Because the state of the zero flag does not propagate like the carry and overflow flag, after each subtraction a conditional jump is necessary. The C listing is:

```
  long HLHL, DEDE;
  char A;

  HLHL = 0x12345678;
  DEDE = 0x12340000;
  if (HLHL == DEDE) {
    A = 1;
  } else {
    A = 0;
  }
```

The assembler listing is:

```
;================================================
; IF-ELSE CODE FRAGMENT 2
;
        LD      HL,05678H       ; H'L'HL = 0x12345678;
        LD      DE,00000H       ; D'E'DE = 0x12340000;
        EXX
        LD      HL,01234H
        LD      DE,01234H
        EXX

        AND     A               ; if (HLHL == DEDE) {
```

```
        SBC     HL,DE
        JR      NZ,ELSE
        EXX
        SBC     HL,DE
        EXX
        JR      NZ,ELSE
        LD      A,1             ;   A = 1;
        JR      ENDIF
ELSE:                           ; } else {
        LD      A,0             ;   A = 0;
ENDIF:                          ; }
        HALT

        END
```

Note: What happens if the first JR NZ,ELSE is deleted?

Z80 32-bit (long) shift right 1 bit

As introduction to multiplication which is done as shifts with conditional additions, we look at arithmetic shift. The special thing about arithmetic shift is that the sign bit (the top most bit) has to be shifted down. A special op-code SRA was available in the Z80 for this purpose. As mentioned above the registers B, C, D, E, H and L could do shift and rotate. Let's do a shift right with BCDE registers.
Some 32-bit 2ers complement numbers:

```
000000000H = 0
000000001H = 1
07FFFFFFH = 2147483647    ; most positive (largest) number
080000000H = -2147483648    ; most negative (smallest) number
080000001H = -2147483647
0FFFFFFFFH = -1
```

The shift right arithmetic of 080000001h is 0C0000000h. The sign bit (1 for negative sign) is preserved.

```
; LOAD TEST VALUES
        LD      DE,4321H        ; HEXADECIMAL 4321 TO DE (LOWER 16-BIT)
        LD      BC,8765H        ; HEXADECIMAL 8765 TO BC (UPPER 16-BIT)
        CALL    SRA32
        HALT

;================================================
; 1 BIT SHIFT RIGHT ARITHMETRIC ROUTINE 32BIT = 32BIT
; BCDE >>= 1
; CHANGES FLAGS
;
SRA32:
        SRA     B
        RR      C
        RR      D
        RR      E

; RESULT IS IN BCDE.
; THE LOWEST BIT WAS SHIFTED INTO CARRY.
        RET

        END
```

Z80 32-bit (long) shift right N bits

The CORDIC algorithm needs N bits shift. A N bit shift can be done with N times a 1 bit shift. It can be improved by the observation that 8 bits are 1 byte and that the Z80 can handle data in 8 bit blocks. The following N bit shift first tries to shift 8 bits blocks, then tries to shift 1 bit blocks.

```
; LOAD TEST VALUES
        LD      DE,4321H        ; HEXADECIMAL 4321 TO DE (LOWER 16-BIT)
        LD      BC,8765H        ; HEXADECIMAL 8765 TO BC (UPPER 16-BIT)
        LD      A,18            ; SHIFT 18 BITS
        CALL    SRBCDE
        HALT

;================================================
; N BIT ARITHMETRIC SHIFT RIGHT ROUTINE 32BIT = 32BIT
; BCDE >>= A
; CHANGES FLAGS
;
SRBCDE:
    SUB     8               ; SET CARRY FLAG IF A < 8
    JR      C,SRBEBT        ; NO MORE BYTES TO SHIFT
    LD      E,D             ; SHIFT BITS 8..15 TO 7..0
    LD      D,C             ; SHIFT BITS 16..23 TO 8..15
    LD      C,B             ; SHIFT BITS 24..31 TO 16..23
    LD      B,0             ; ASSUME POSITIVE NUMBER
    BIT     7,C             ; SET ZERO FLAG IF BIT 7 == 0
    JR      Z,SRBEPS
    DEC     B               ; CHANGE + TO - SIGN
SRBEPS:
    JR      SRBCDE
SRBEBT:
    ADD     A,8             ; UNDO SUB 8, SET ZERO FLAG IF A == 0
SRBELP:
    JR      Z,SRBERT        ; NO MORE BITS TO SHIFT
    SRA     B
    RR      C
    RR      D
    RR      E
    DEC     A               ; SET ZERO FLAG IF A == 0
    JR      SRBELP
SRBERT:
    RET                     ; RESULT IS IN BCDE.

    END
```

Z80 32-bit (long) multiply, 32-bit result

The 6502 and Z80 were on equal terms for add and sub. Starting with multiplication the Z80 did benefit from all the registers. Within the multiply loop it is not necessary to load/store values. A 32-bit * 32-bit multiplication has a 64-bit result. But in 32-bit integer arithmetric only the lower 32-bit can be stored.
The multiplier MPR and multiplicand MPD go into B'C'BC and D'E'DE registers. The result is in H'L'HL. In Z80 assembler programming it was common to have parameter passing in registers - something that the C compiler did learn, too.
The actual MPR is AC'BC. This frees the B' register as loop counter. The DJNZ op-code which combined decrement and conditional jump instruction was only possible with B or B' register.

```
; LOAD TEST VALUES
        LD      DE,01213H
        LD      BC,3
        EXX
        LD      DE,01011H
```

```
        LD      BC,0
        EXX
        CALL    MUL32
        HALT

;================================================
; MULTIPLY ROUTINE 32*32BIT=32BIT
; H'L'HL = B'C'BC * D'E'DE
; NEEDS REGISTER A, CHANGES FLAGS
;
MUL32:
        AND     A               ; RESET CARRY FLAG
        SBC     HL,HL           ; LOWER RESULT = 0
        EXX
        SBC     HL,HL           ; HIGHER RESULT = 0
        LD      A,B             ; MPR IS AC'BC
        LD      B,32            ; INITIALIZE LOOP COUNTER
MUL32LOOP:
        SRA     A               ; RIGHT SHIFT MPR
        RR      C
        EXX
        RR      B
        RR      C               ; LOWEST BIT INTO CARRY
        JR      NC,MUL32NOADD
        ADD     HL,DE           ; RESULT += MPD
        EXX
        ADC     HL,DE
        EXX
MUL32NOADD:
        SLA     E               ; LEFT SHIFT MPD
        RL      D
        EXX
        RL      E
        RL      D
        DJNZ    MUL32LOOP
        EXX

; RESULT IN H'L'HL
        RET

        END
```

Z80 32-bit (long) multiply, 64-bit result

For floating point calculation a 32-bit multiply with 64-bit result is needed. The reason is in the normalization of the floating point mantissa. The 32-bit mantissa m is the nominator of the fraction m / 2^31 or m / 80000000h for 32-bit mantissa. The normalized fraction is >= 1.0 and < 2.0, or in the range 080000000h to 0FFFFFFFFh. The mantissa itself is always positive - there is a sign bit somewhere else in the floating point number. Because the highest bit of a normalized mantissa is always 1, this bit is not stored in most floating point formats. For details read the article about IEEE754 floating point format.
Another way to describe the mantissa is to call it a fixed-point number - an integer number with shifted decimal point. A normal 32-bit integer with decimal point after the last bit is called 32.0 fixed-point. A 32-bit normalized mantissa is called unsigned 1.31 fixed-point. One bit before the decimal point, 31 bits after the decimal point. Some 1.31 fixed point mantissa values:

```
0FFFFFFFFH = 1.9999999995   ; largest normalized mantissa (fraction) value
0FFFFFFFEH = 1.9999999991
0C0000000H = 1.5
080000001H = 1.0000000005
080000000H = 1.0            ; smallest normalized mantissa (fraction) value
```

The upper 32-bit of the 64-bit result were used with mantissa multiply. Rounding of the result can be done with the highest bit of the lower 32bit result. In most cases rounding was not done, leading to jokes like "the soccer match between Intel and AMD finished with 0 to 1.9999999".

```
; LOAD TEST VALUES
        LD      DE,0FFFEH
        LD      BC,00001H
        EXX
        LD      DE,0FFFFH       ; 1.9999999991
        LD      BC,08000H       ; 1.0000000005
        EXX
        CALL    LMUL
        HALT

;================================================
; MULTIPLY ROUTINE 32*32BIT=64BIT
; H'L'HLB'C'AC = B'C'BC * D'E'DE
; NEEDS REGISTER A, CHANGES FLAGS
;
; THIS ROUTINE WAS WITH TINY DIFFERENCES IN THE
; SINCLAIR ZX81 ROM FOR THE MANTISSA MULTIPLY
;
LMUL:
        AND     A               ; RESET CARRY FLAG
        SBC     HL,HL           ; RESULT BITS 32..47 = 0
        EXX
        SBC     HL,HL           ; RESULT BITS 48..63 = 0
        EXX
        LD      A,B             ; MPR IS B'C'AC
        LD      B,33            ; INITIALIZE LOOP COUNTER
        JR      LMULSTART
LMULLOOP:
        JR      NC,LMULNOADD
        ADD     HL,DE           ; RESULT += MPD
        EXX
        ADC     HL,DE
        EXX
LMULNOADD:
        EXX
        RR      H               ; RIGHT SHIFT UPPER
        RR      L               ; 32BIT OF RESULT
        EXX
        RR      H
        RR      L
LMULSTART:
        EXX
        RR      B               ; RIGHT SHIFT MPR/
        RR      C               ; LOWER 32BIT OF RESULT
        EXX
        RRA                     ; EQUIVALENT TO RR A
        RR      C
        DJNZ    LMULLOOP

; RESULT IN H'L'HLB'C'AC

; ROUNDING (WAS NOT INCLUDED IN ZX81)
        EXX
        SLA     B               ; RESULT BIT 31 TO CARRY
        EXX
        LD      BC,0            ; LD DOES NOT CHANGE CARRY
        ADC     HL,BC           ; HL = HL + 0 + CARRY
        EXX
        LD      BC,0
```

```
        ADC     HL,BC           ; HL = HL + 0 + CARRY
        EXX

; ROUNDED RESULT IN H'L'HL (UPPER 32BITS)
        RET

        END
```

Floating Point Formats

The fixed point numbers are fine for the sin() and cos() calculations, but not for all floating point calculations. A floating point number is a data reccord (bit struct) with the elements: exponent sign, exponent value, mantissa sign and mantissa value. Another name for mantissa is fraction. Sign and value of exponent can be combined to a two's complement integer or to an integer with bias. In the IEEE 754 format the exponent uses a bias and not the two's complement. Next to the IEEE 754 format there are other floating point formats. One possible coding for floating point is mantissa in two's complement and exponent in two's complement. This 40 bit coding is convinient for computation. The 32 bit mantissa is stored in one or more registers, the 8 bit exponent in another register. For storage the IEEE 754 format with 32 bit

IBM 1130, IBM 1800 32 Bit single precision floating point

```
 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
+-+-----------------------------------------------+---------------+
|S|                mantissa                       |exp. with bias |
+-+-----------------------------------------------+---------------+
```

GE-635 36 Bit single precision floating point

```
 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3
+-+-------------+-+-------------------------------------------------------+
|S|  Two's exp. |S|            Two's complement mantissa                  |
+-+-------------+-+-------------------------------------------------------+
```

Intel 8008, 8080 floating point UCRL-51940

```
 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
+-+-------------+-----------------------------------------------+
|S|  Two's exp. |                 positive mantissa             |
+-+-------------+-----------------------------------------------+
```

Altair BASIC 8080 floating point

```
 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
+---------------+-+---------------------------------------------+
|exp. with bias |S|       positive mantissa, MSB = 2^-1        |
+---------------+-+---------------------------------------------+
```

Z80 40 Bit floating point

```
 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
+-+-------------+-+-------------------------------------------------------------+
|S|  Two's exp. |S|            Two's complement mantissa, MSB = 2^1             |
+-+-------------+-+-------------------------------------------------------------+
```

```
IEEE 754 single precision

 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
+-+---------------+-----------------------------------------------+
|S| exp. with bias|       positive mantissa, MSB = 2^-1          |
+-+---------------+-----------------------------------------------+
```

The internal 40 bits floating point format for the Z80 uses 8 bits two's complement for the exponent and 32 bit two's complement for the mantissa. The mantissa uses a 3.29 fixed point number for numbers between -3.9999999 to 3.9999999.

IEEE 754 Floating Point Format

The external floating point format is the IEEE 754 with 8 bits with bias for the exponent, 1 bit for the sign and 23 (24) bits for the mantissa. Because the mantissa of a normalized number is in the range from 1.0000000 to 1.9999999, the topmost mantissa bit 2^0 is not stored in the IEEE 754 number.
The IEEE 754 format can represent normalized numbers in the range $2 * 10^{-38}$ to $2 * 10^{38}$. Numbers below $2 * 10^{-38}$ are represented as denormalized numbers. The IEEE 754 standard describes in section 3.2.1 the details of the single format number:

1. If e = 255 and f != 0 , then v is NaN regardless of s
2. If e = 255 and f = 0 , then $v = (-1)^s$ INFINITY
3. If 0 < e < 255 , then $v = (-1)^s 2^{e-127} (1 . f)$
4. If e = 0 and f != 0 , then $v = (-1)^s 2^{-126} (0 . f)$ (denormalized numbers)
5. If e = 0 and f = 0 , then $v = (-1)^s 0$ (zero)

The IEEE 754 standard uses the name fraction instead of mantissa. For normal numbers the mantissa has the form 1.mantissa, for denormalized numbers the form is 0.mantissa. The special values NaN (not a number) and INF (infinity) represent the results of illegal operations like division through zero.

Conversion IEEE 754 to Z80 40bits format

The 32 bits IEEE 754 format is expanded to a 40 bits Z80 format. The suppressed mantissa bit 2^0 is restored to 1 for normalized numbers and 0 for denormalized numbers. Left to the decimal point one more bit is added. This bit is needed for addition and subtraction. The result of 1.9999999 plus 1.9999999 is 3.9999998. This mantissa value is no longer a normalized IEEE 754 mantissa.

```
    LD      HL,0
    EXX
    LD      HL,03f80h
    EXX
    CALL    ToZ80FP
    CALL    ToIEEE
    HALT

NEGHLHL:        ; calculate two's complement with H'L'HL = 0 - H'L'HL
                ; changes A, H'L'HL, Flags
    XOR     A   ; A = 0
    SUB     L
    LD      L,A
```

```
    LD     A,0
    SBC    A,H
    LD     H,A
    EXX
    LD     A,0
    SBC    A,L
    LD     L,A
    LD     A,0
    SBC    A,H
    LD     H,A
    EXX
    RET

; Conversion IEEE 754 32 bit in Z80 registers to Z80 40bit
; IEEE 754 single precision, Z80

;  7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
; +-+---------------+-----------------------------------------------+
; |S| exp. with bias|   positive mantissa, MSB = 2^-1               |
; +-+-------------+-+-------------+---------------+---------------+
; |      H'       |        L'     |       H       |       L       |
; +---------------+---------------+---------------+---------------+

; conversion to internal 40 bit two's complement
; exponent: 8 bits two's complement integer
; mantissa: Sign + 2.29 bits two's complement fixed point

ToZ80FP:
; BH'L'HL << 8
    LD    B,H        ; mantissa byte 2
    EXX
    LD    A,H        ; sign+exp-high
    LD    H,L        ; mantissa byte 1
    LD    L,B        ; mantissa byte 2
    EXX
    LD    H,L        ; mantissa byte 3
    LD    B,A        ; sign+exp-high
    LD    L,0        ; mantissa byte 4

;  7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
; +-+---------------+-------------------------------------------+---------------+
; |S| exp. with bias|      positive mantissa, MSB = 2^-1        |0 0 0 0 0 0 0 0|
; +-+-------------+-+-------------+---------------+---------------+---------------+
; |       B       |      H'       |        L'     |       H       |       L       |
; +---------------+---------------+---------------+---------------+---------------+

; H'L'HL >> 2
    LD    A,2
SRLHLHL:
    EXX
    SRL   H
    RR    L
    EXX
    RR    H
    RR    L
    DEC   A
    JR    NZ,SRLHLHL

;  7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
; +-+-------------+---+-+-----------------------------------------+-----------+
; |S| exp. w. bias|0 0|e|      positive mantissa, MSB = 2^-1       |0 0 0 0 0 0|
; +-+-------------+---+-+---------+---------------+---------------+---+-----------+
; |       B       |      H'       |        L'     |       H       |       L       |
; +---------------+---------------+---------------+---------------+---------------+
```

```
    LD     A,B
    ADD    A,A        ; A << 1
    EXX
    BIT    5,H
    JR     Z,ExpIsEven
    INC    A
ExpIsEven:
                       ; Exponent with bias in A
    OR     A
    JR     Z,DeNorm
    SET    5,H        ; Normalized mantissa 2^0 bit is 1
    JR     NormEnd
DeNorm:
    RES    5,H        ; Denormalized mantissa 2^0 bit is 0
NormEnd:
    EXX
    SUB    127        ; Two's complement exponent in A
    SLA    B          ; B << 1, Carry = S
    LD     B,A
    JR     NC,PosMan
    CALL   NEGHLHL
PosMan:
    RET

;  7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
; +-+-------------+-+-----------------------------------------------+
; |S| Two's exp.  |S|            Two's complement mantissa, MSB = 2^1              |
; +-+-------------+-+-------------+---------------+---------------+---------------+
; |       B       |      H'       |        L'     |       H       |       L       |
; +---------------+---------------+---------------+---------------+---------------+
```

Conversion Z80 40bits to IEEE 754 format

The conversion from Z80 40bits format to IEEE 754 normalizes the mantissa. The sub routine can not handle denormalized numbers.

```
ToIEEE:
    EXX
    LD     A,H
    EXX
    LD     C,A        ; C stores Sign
    BIT    7,C
    JR     Z,PosMan2
    CALL   NEGHLHL
PosMan2:
    LD     A,L
    OR     H
    EXX
    OR     L
    OR     H
    EXX
    JR     Z,IEEEEND    ; H'L'HL is zero, no more conversion
SLBEGIN:
    SLA    L
    RL     H
    EXX
    RL     L
    RL     H
    BIT    7,H
    EXX
    JR     NZ,SLEND    ; Bit 2^0 is 1, normalized number
```

```
    INC    B           ; ++exponent
    JR     SLBEGIN
SLEND:
    DEC    B           ; Z80 mantissa is signed 2.29; IEEE is unsigned 1.23

;  7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
; +---------------+-----------------------------------------------+
; | Two's exponent|           normalized positive mantissa, MSB = 2^0             |
; +---------------+---------------+---------------+---------------+---------------+
; |       B       |      H'       |       L'      |       H       |       L       |
; +---------------+---------------+---------------+---------------+---------------+

    EXX
    LD     A,L
    LD     L,H
    EXX
    LD     L,H
    LD     H,A         ; 24 bit mantissa is in L'HL
    LD     A,127
    ADD    A,B         ; A = exponent with bias
    SLA    C           ; Carry = Sign
    EXX
    LD     H,A
    RR     H           ; Sign is Bit 7, Exponent Bit 0 is Carry
    JR     C,OddExp
    RES    7,L
OddExp:
    EXX

;  7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
; +-+---------------+-----------------------------------------------+
; |S| exp. with bias|      positive mantissa, MSB = 2^-1            |
; +-+---------------+-+-------------+---------------+---------------+
; |      H'       |        L'     |       H       |       L       |
; +---------------+---------------+---------------+---------------+

IEEEEND:
    RET
```

Floating point add

The Floating point add subroutine is presented als programming language C pseudocode. The 8 bit two's complement exponents are in register B and C. The 32 bit two's complement mantissas are in registers H'L'HL and D'E'DE. In Z80 assembler implementation the exchange of HL and DE register will be done with the EX DE,HL instruction.

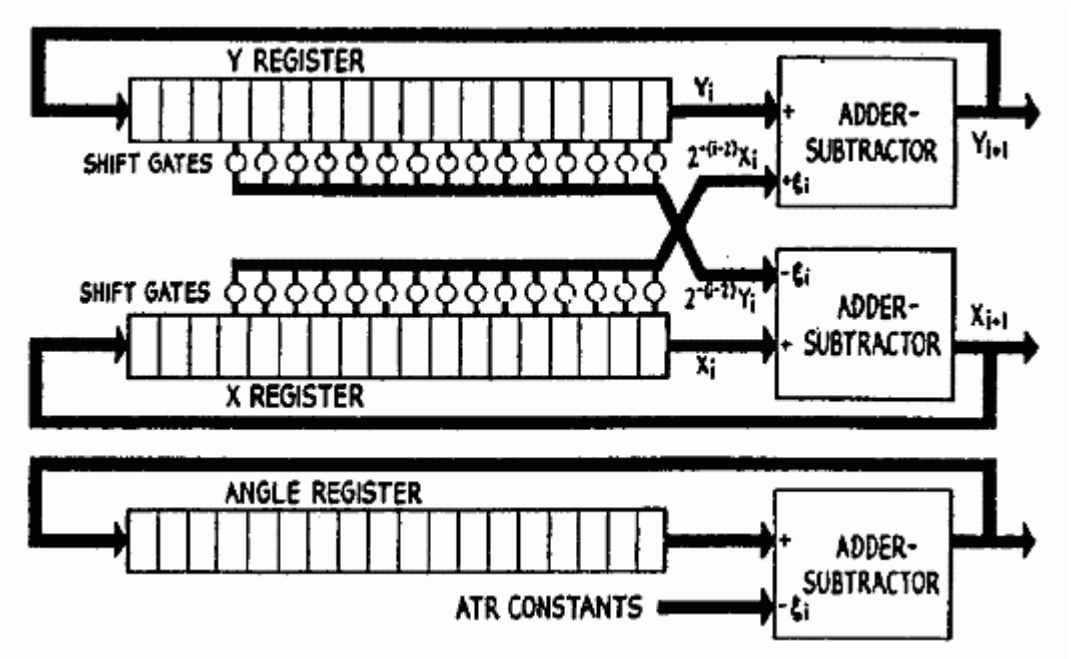
```
long hlhl, dede, ltmp;    // mantissa
signed char b, c, a;      // exponent

// B,H'L'HL += C,D'E'DE
void fadd_z80() {
    a = b;
    if (a < c) {
        // larger number to B,H'L'HL
        a = c;
        c = b;
        b = a;
        ltmp = dede;
        dede = hlhl;
        hlhl = ltmp;
```

```
    }
    a = b;
    a -= c;
    if (a > 23) return;
    dede >>= a;        // mantissa alignment
    hlhl += dede;
}
```

sincos() CORDIC Algorithm

The COordinate Rotation Digital Computer was developed by Jack E. Volder to calculate Sine and Cosine with a digital computer. In the paper "The CORDIC Trigonometric Computing Technique" Volder wrote: "the trigonometric operations in the CORDIC computer can be functionally described as the digital equivalent of an analog resolver". A resolver is a transducer that converts the angular position and/or velocity of a rotating shaft to an electrical signal. It delivers signals proportional to the Sine and/or Cosine of the shaft angle. A resolver needs the rotation of the shaft to build up the magnetic field that gets converted into a voltage by the transducer coil. The CORDIC algorithm calculates the Sine and Cosine at the same time. A good sign for "CORDIC inside" are devices that can compute sin() and cos() in one command, like the floating point chip Intel 80387 with its FSINCOS op-code.
Before the work of Volder, sin() and cos() in digital computers were calculated with Taylor series expansion (was used in Commodore C64) or the Chebyshev series expansion (was used in Sinclair ZX-81). The series expansions need multiplications. The CORDIC needs additions and shifts, numerical operations that are easier to perform then multiplications for a digital computer. The CORDIC I device was build with three shift gates and three adder/subtractors as seen in Fig. 2 in Volder's paper. The angle input value is stored in the angle register. The Y register contains the Sine and the X register contains the Cosine value after the run of the CORDIC algorithm. The Arc Tangent Radix (ATR) constants are the true magic of the CORDIC algorithm. These constants are the arcus tangens values of 2^{-i}. The CORDIC algorithm uses a loop for the calculation. In every iteration of the loop one bit of result is created. For a result that is correct for 7 digits decimal, which is the accuracy of 32 Bit IEEE754 floating point numbers, we need 22 iterations.



The ATR constants for i >= 0 are:

$$a_i = \arctan\left(2^{-i}\right)$$

The following C language source code snippet creates the ATR constants:

```
#define N 22

double atr[N];

void ATR_constants() {
  int i;
  for (i = 0; i < N; ++i) {
    atr[i] = atan(pow(2.0, -i));
  }
}
```

These ATR constants are calculated offline and are stored in the ATR constants array. The CORDIC algorithm iterates for i = 0 .. N over the following formulas:

$$Y_{i+1} = Y_i \pm 2^{-i} \cdot X_i$$ The new Y value is the current Y value plus (if angle is positive) or minus (if angle is negative) the shifted current X value.

$$X_{i+1} = X_i \mp 2^{-i} \cdot Y_i$$ The new X value is the current X value minus (if angle is positive) or plus (if angle is negative) the shifted current Y value.

$$\theta_{i+1} = \theta_i \mp a_i$$ And the new angle θ value is the current angle θ value minus (if angle is positive) or plus (if angle is negative) an ATR constant.

sincos() CORDIC implementation in C

The following C language source code snippet contains the CORDIC algorithm for floating point numbers:

```
/* returns Sine of angle and Cosine of angle */
double cordic(double *cos, double angle) {
    double y = 0.0;
    double x = 0.607252935009;
    double *pATR = atr;
    int i;
    for (i = 0; i < N; ++i) {
        double yold = y;
        y += ((angle > 0)? pow(2.0, -i): -pow(2.0, -i)) * x;
        x -= ((angle > 0)? pow(2.0, -i): -pow(2.0, -i)) * yold;
        angle -= (angle > 0)? *pATR++: -*pATR++;
    }
    *cos = x;
    return y;
}
```

Up to now the CORDIC algorithm is no great help for a CPU without floating point unit. The floating point numbers have to be replaced by fixed point numbers. The valid range for angle is between -π/2 and π/2. With a 2.30 signed fixed point number the range is -1.999999999 to 1.999999999. Here is the CORDIC algorithm with fixed point numbers:

```
#define N           22        /* iterations */
#define FRACTION    30
#define FIXED(X)    ((long)((X) * (1 << FRACTION)))
#define FLOAT(X)    ((X) / (double)(1 << FRACTION))

typedef long fixed;   /* signed 2.30 fixed-point */

fixed atr_fixed[N];

void ATR_constants_fixed() {
    int i;
    for (i = 0; i < N; ++i) {
        atr_fixed[i] = FIXED(atan(pow(2.0, -i)));
    }
}

/* returns Sine of angle and Cosine of angle */
fixed cordic_fixed(fixed *cos, fixed angle) {
    fixed y = 0;
    fixed x = FIXED(0.607252935009);
    fixed *pATR = atr_fixed;
    int i = 0;
    do {
        fixed yold = y;
        y += (angle > 0)? x >> i: -(x >> i);
        x -= (angle > 0)? yold >> i: -(yold >> i);
        angle -= (angle > 0)? *pATR++: -*pATR++;
    } while (++i < N);
    *cos = x;
    return y;
}
```

The magic number 0.607.. still needs explanation. Well, all I can say is: please read the paper of Volder or the papers "A unified algorithm for elementary functions" by J. S. Walther from 1971 or "Pseudo Division and Pseudo Multiplication Processes" by J. E. Meggitt from 1962. The Walther paper is part of the US patent 3766370. Another good source is US patent 4001569 which contains the complete ROM Listing of the HP45 pocket calculator. One of the inventors is David S. Cochran, the programmer behind the HP9100 and HP35, the first electronic calculators that used the CORDIC algorithm.

sincos() CORDIC improved implementation in C

The ATR array of the CORDIC algorithm can be optimized. If we compare the values of arctan(2^{-i}) and 2^{-i}, we see that for i >= 8 these values are the same. The ATR array needs only the 9 entries from 0 to 8. If we need the ATR array entry 9 we shift the ATR array entry 8 one bit to the right, which is the same as divide by two. The other ATR array entries are created "on the fly" with the same method. The space optimized signed fixpoint CORDIC is:

```
#define N           22       /* iterations */
#define M           8        /* length of ATR array */
#define FRACTION    30
#define FIXED(X)    ((long)((X) * (1 << FRACTION)))
#define FLOAT(X)    ((X) / (double)(1 << FRACTION))

typedef long fixed;   /* signed 2.30 fixed-point */

fixed atr[M+1];
```

```
void ATR_constants_fixed() {
  int i;
  for (i = 0; i <= M; ++i) {
    atr_fixed[i] = FIXED(atan(pow(2.0, -i)));
  }
}

/* returns Sine of angle and Cosine of angle */
fixed cordic_fixed(fixed *cos, fixed angle) {
    fixed y = 0;
    fixed x = FIXED(0.607252935009);
    fixed *pATR = atr_fixed;
    unsigned int i = 0;
    do {
        fixed yold = y;
        y += (angle >= 0)? x >> i: -(x >> i);
        x -= (angle >= 0)? yold >> i: -(yold >> i);
        fixed a = (i < M+1)? *pATR++: atr_fixed[M] >> (i-M);
        angle -= (angle >= 0)? a: -a;
    } while (N-1 >= ++i);
    *cos = x;
    return y;
}
```

The cordic function uses only the >= and < compare operators, because only these operators are directly available on the Z80.

SINCOS in Z80 assembler

As we can see in the C sincos() function the CORDIC algorithm needs variable storage for X, Y, Angle, Yold, pATR and Step. The function argument Angle is passed by register. Because registers are a valueable resource Angle is stored in variable storage, too. Constant storage for the ATR array in fixed point format is needed, too. The assembler pseudo op-code DEFW defines 16-bit values. The memory layout for SINCOS is:

```
;================================================
; Constants
;
CN:         EQU         22
CM:         EQU         8

;================================================
; variable storage (RAM)
;
Angle:
    DEFW    0,0
Y:
    DEFW    0,0
X:
    DEFW    0,0
Yold:
    DEFW    0,0
pATR:
    DEFW    0
Step:
    DEFB    0

;================================================
; constant storage (ROM)
;
ATR:
```

```
    DEFW 0F6A8H,03243H      ; atan(1)
    DEFW 06705H,01DACH      ; atan(1/2)
    DEFW 0BAFCH,00FADH      ; atan(1/4)
    DEFW 06EA6H,007F5H      ; atan(1/8)
    DEFW 0AB76H,003FEH      ; ...
    DEFW 0D55BH,001FFH
    DEFW 0FAAAH,000FFH
    DEFW 0FF55H,0007FH
    DEFW 0FFEAH,0003FH
```

Note: More sophisticated assembler can separate between RAM storage and ROM storage. The constant values for Angles were created with the cordic.c program.

SINCOS initialization

The SINCOS initialization is equivalent to the C function between the function start and the do loop. The C source is given as comment. The sin() result is in D'E'DE registers, the cos() result is in B'C'BC.

```
;================================================
; SIN() COS() ROUTINE 2 * 32BIT = 32BIT
; B'C'BC, D'E'DE = f(BCDE)
; NEEDS REGISTERS A BC DE HL, CHANGES FLAGS
;
SINCOS:
    LD      (Angle),DE      ; store argument
    LD      (Angle+2),BC
    SUB     A               ; A = 0
    LD      (Step),A        ; i = 0
    LD      L,A
    LD      H,A
    LD      (Y),HL          ; y = 0;
    LD      (Y+2),HL
    LD      HL,03B6AH       ; x = FIXED(0.607252935..);
    LD      (X),HL
    LD      HL,026DDH
    LD      (X+2),HL
    LD      HL,ATR          ; pATR = ATR;
    LD      (pATR),HL
```

SINCOS loop

This is the longest assembler listing in the article up to now. But it was easy to write. Just compile (by hand) the C source statements into assembler statements. Very important is the quality of the pseudo code. You should not develop your algorithms at the primitive assembler abstration level. Use C or MathCad, Matlab, Mathematica to evaluate, test and proof your work.
Note: The angle >= 0 compare is done with a test of the sign bit.

```
SNCSDO:                         ; do {
    LD      HL,(Y)          ;   yold = y;
    LD      (Yold),HL
    LD      HL,(Y+2)
    LD      (Yold+2),HL
                            ;   y += (angle >= 0)? x >> i: -(x >> i);
    LD      DE,(X)          ;   x >> i
    LD      BC,(X+2)
```

```
    LD      A,(Step)
    CALL    SRBCDE
    LD      A,(Angle+3)     ;   (angle >= 0)?
    RLA                     ;   Bit 7 to Carry
    LD      HL,(Y)
    JR      C,SNCSL1
    ADD     HL,DE           ;   y += x >> i
    LD      (Y),HL
    LD      HL,(Y+2)
    ADC     HL,BC
    JR      SNCSE1
SNCSL1:
    AND     A               ;   y -= x >> i
    SBC     HL,DE
    LD      (Y),HL
    LD      HL,(Y+2)
    SBC     HL,BC
SNCSE1:
    LD      (Y+2),HL
                            ;   x -= (angle >= 0)? yold >> i: -(yold >> i);
    LD      DE,(Yold)       ;   yold >> i
    LD      BC,(Yold+2)
    LD      A,(Step)
    CALL    SRBCDE
    LD      A,(Angle+3)     ;   (angle >= 0)?
    RLA
    LD      HL,(X)
    JR      C,SNCSL2
    AND     A               ;   x -= yold >> i
    SBC     HL,DE
    LD      (X),HL
    LD      HL,(X+2)
    SBC     HL,BC
    JR      SNCSE2
SNCSL2:
    ADD     HL,DE           ;   x += yold >> i
    LD      (X),HL
    LD      HL,(X+2)
    ADC     HL,BC
SNCSE2:
    LD      (X+2),HL
                          ;   a = (i < M+1)? *pATR++: atr_fixed[M] >> (i-M);
    LD      A,(Step)      ;   (i < M+1)?
    SUB     CM+1          ;   i-(M+1)
    JR      NC,SNCSL3
    LD      HL,(pATR)     ;   *pATR++
    LD      E,(HL)
    INC     HL
    LD      D,(HL)
    INC     HL
    LD      C,(HL)
    INC     HL
    LD      B,(HL)
    INC     HL
    LD      (pATR),HL
    JR      SNCSE3
SNCSL3:
    LD      DE,(ATR+32)     ;   atr[M] >> i-M
    LD      BC,(ATR+34)
    INC     A               ;   i-M
    CALL    SRBCDE
SNCSE3:
                            ;   angle -= (angle >= 0)? a: -a;
    LD      A,(Angle+3)     ;   (angle >= 0)?
    RLA
```

```
    LD      HL,(Angle)
    JR      C,SNCSL4
    AND     A               ;   angle -= a
    SBC     HL,DE
    LD      (Angle),HL
    LD      HL,(Angle+2)
    SBC     HL,BC
    JR      SNCSE4
SNCSL4:
    ADD     HL,DE           ;   angle += a
    LD      (Angle),HL
    LD      HL,(Angle+2)
    ADC     HL,BC
SNCSE4:
    LD      (Angle+2),HL
                            ; } while (++i <= N-1);
    LD      HL,Step         ; ++i
    INC     (HL)
    LD      A,CN-1          ; N-1 >= ++i
    CP      (HL)
    JP      NC,SNCSDO
                            ; RESULT IS IN MEM(X) AND MEM(Y)
    LD      DE,(X)
    LD      BC,(Y)
    EXX
    LD      DE,(X+2)
    LD      BC,(Y+2)
    EXX
    RET                     ; RESULT IS IN D'E'DE AND B'C'BC
```

Complete SINCOS source code

The complete SINCOS module is 340 bytes long in Z80 machine code. 19 bytes are RAM, the other bytes are ROM. Here is a link to sincos.z80 that contains the complete Z80 assembler source code for SINCOS.

sincos() CORDIC reentrant solution

The C and assembler programs above did use variable storage. Another solution is to use stacks. The CORDIC stack algorithm needs three 32 Bit registers for angle, x and y, an index register for the ATR constants and a 8 Bit register for i. The 32 Bits registers are H'L'HL, D'E'DE and B'C'BC, index register is IY and loop counter register is A. Because there are so many variables on the stack, the parameter stack and the return stack is used for easy ordering of the variables on the stacks. The parameter stack pointer is SP, the Forth threaded code return stack pointer is IX. For details see Forth Register. The C program is an intermediate step for a CORDIC implementation in Z80 assembler. The macros push() and pop() simulate the parameter stack op-codes, rpush() and rpop() do the same for the return stack. The variable names describe which Z80 registers will be used. The stack solution is harder to read then the variable solution, but the author hopes that the assembler program will be faster and shorter then the variable solution.

sincos() reentrant C Simulation

```
/* returns Sine of angle and Cosine of angle, rentrant */
/* C simulation of Z80 assembler */
fixed sincosr(fixed *cos, fixed bcbc) {
  fixed hlhl = 0;                   // y
```

```
  fixed dede = FIXED(InvK);    // x
  fixed *iy = atr_fixed;
  signed char af = 0;              // i
  do {
    push(hlhl);        // y
    push(dede);        // y x
    push(bcbc);        // y x angle
    push(af);          // y x angle i
    dede >>= af;
    if (bcbc >= 0) {
      hlhl += dede;
    } else {
      hlhl -= dede;
    }
    rpush(hlhl);      // R: y+1
    pop(af);          // y x angle (i)
    pop(bcbc);        // y x (angle)
    pop(hlhl);        // y (x)
    pop(dede);        // nil (y)
    push(bcbc);       // angle
    push(af);         // angle i
    dede >>= af;
    if (bcbc >= 0) {
      hlhl -= dede;
    } else {
      hlhl += dede;
    }
    dede = hlhl;
    rpop(bcbc);       // R: nil (y+1)
    pop(af);          // angle (i)
    pop(hlhl);        // nil (angle)
    push(bcbc);       // y+1
    push(dede);       // y+1 x+1
    push(af);         // y+1 x+1 i
    af -= M+1;
    if (af < 0) {
      dede = *iy++;
    } else {
      ++af;
      dede = atr_fixed[M];
      dede >>= af;
    }
    if (hlhl >= 0) {
      hlhl -= dede;
    } else {
      hlhl += dede;
    }
    pop(af);      // y+1 x+1 (i)
    bcbc = hlhl;
    pop(dede);    // y+1 (x+1)
    pop(hlhl);    // nil (y+1)
    ++af;
  } while (af-CN < 0);
  printf("HL=%8lX DE=%8lX BC=%8lX\n", hlhl, dede, bcbc);
  printf("|angle|=%1.8f (%08X)\n", abs(FLOAT(bcbc)), bcbc);
  *cos = dede;
  bcbc = hlhl;
  return bcbc;
}
```

SINCOS reentrant Z80 Assembler

This solution is reentrant. The solution with variables was not. Because of the limited numbers of Z80 registers, two stacks are needed for temporary storage of the variables. The hardware stack pointer SP handles the parameter and subroutine return stack. The index register IX handles the second stack, the Forth return stack. The author is quite proud on this implementation of the CORDIC algorithm in Z80 assembler. You should not find many implementations that are faster and/or smaller then this subroutine. The accuracy is 7 digits decimal as it is needed for the IEEE 754 standard.

```
; sincosr.z80
; CORDIC for sin() and cos() for the Zilog Z80 and Assembler Z80DT
; reentrant version
; Author: Andre Adrian
; Version: 12Apr2011
; length is 333 bytes

; LOAD TEST VALUES
    LD      IX,1E80H        ; init second stack pointer
    LD      BC,12AFH        ; Angle = -1.57079632679 (-90°)
    EXX
    LD      BC,9B78H
    EXX
    CALL    SINCOS
                            ; expected sin = B'C'BC = 0C0000000H (-1.0)
                            ; expected cos = D'E'DE = 000000000H (0.0)
                            ; delivered sin = B'C'BC = 0BFFFFFFFH
                            ; delivered cos = D'E'DE = 000000031H
    HALT

;================================================
; Constants
;
CN:         EQU         22
CM:         EQU         8

;================================================
; constant storage (ROM)
;
ATR:
    DEFW     0F6A8H,03243H    ; atan(1)
    DEFW     06705H,01DACH    ; atan(1/2)
    DEFW     0BAFCH,00FADH    ; atan(1/4)
    DEFW     06EA6H,007F5H    ; atan(1/8)
    DEFW     0AB76H,003FEH    ; ...
    DEFW     0D55BH,001FFH
    DEFW     0FAAAH,000FFH
    DEFW     0FF55H,0007FH
    DEFW     0FFEAH,0003FH

;================================================
; N BIT ARITHMETRIC SHIFT RIGHT ROUTINE 32BIT = 32BIT
; D'E'DE >>= A
; CHANGES A, AF' and FLAGS
;
SRDEDE:
    SUB     8           ; SET CARRY FLAG IF A < 8
    JR      C,SRDEBT    ; NO MORE BYTES TO SHIFT
    LD      E,D         ; SHIFT BITS 8..15 TO 7..0
    EXX
    EX      AF,AF'
    LD      A,E         ; SHIFT BITS 16..23 TO 8..15
    LD      E,D         ; SHIFT BITS 24..31 TO 16..23
    LD      D,0         ; ASSUME POSITIVE NUMBER
    BIT     7,E         ; SET ZERO FLAG IF BIT 7 == 0
    JR      Z,SRDEPL
```

```
    DEC     D           ; CHANGE + TO - SIGN
SRDEPL:
    EXX
    LD      D,A
    EX      AF,AF'
    JR      SRDEDE
SRDEBT:
    ADD     A,8         ; UNDO SUB 8, SET ZERO FLAG IF A == 0
SRDELP:
    JR      Z,SRDERT    ; NO MORE BITS TO SHIFT
    EXX
    SRA     D
    RR      E
    EXX
    RR      D
    RR      E
    DEC     A           ; SET ZERO FLAG IF A == 0
    JR      SRDELP
SRDERT:
    RET                 ; RESULT IS IN D'E'DE.

;================================================
; SIN() COS() ROUTINE 2 * 32BIT = 32BIT
; B'C'BC, D'E'DE = f(B'C'BC)
; NEEDS REGISTERS A A' BC B'C' DE D'E' HL H'L' IX IY, CHANGES FLAGS
;
SINCOS:
    SUB    A            ; unsigned char af = 0;     // i
    LD     L,A          ; fixed hlhl = 0;           // y
    LD     H,A
    LD     DE,03B6AH    ; fixed dede = FIXED(InvK); // x
    EXX
    LD     L,A
    LD     H,A
    LD     DE,026DDH
    EXX
    LD     IY,ATR       ; fixed *iy = atr_fixed;
SNCSDO:                 ; do {
    EXX                 ; push(hlhl);       // y
    PUSH   HL
    EXX
    PUSH   HL
    EXX                 ; push(dede);       // y x
    PUSH   DE
    EXX
    PUSH   DE
    EXX                 ; push(bcbc);       // y x angle
    PUSH   BC
    EXX
    PUSH   BC
    PUSH   AF           ; push(af);         // y x angle i
    CALL   SRDEDE       ; dede >>= af;
    EXX
    BIT    7,B          ; if (bcbc >= 0) {
    EXX
    JR     NZ,SNCSL1
    ADD    HL,DE        ;     hlhl += dede;
    EXX
    ADC    HL,DE
    JR     SNCSE1
SNCSL1:                 ; } else {
    AND    A            ;     clear Carry
    SBC    HL,DE        ;     hlhl -= dede;
    EXX
    SBC    HL,DE
```

```
SNCSE1:                 ; }
    DEC     IX          ; rpush(hlhl);      // R: y+1
    LD      (IX+0),H
    DEC     IX
    LD      (IX+0),L
    EXX
    DEC     IX
    LD      (IX+0),H
    DEC     IX
    LD      (IX+0),L
    POP     AF          ; pop(af);          // y x angle (i)
    POP     BC          ; pop(bcbc);        // y x (angle)
    EXX
    POP     BC
    EXX
    POP     HL          ; pop(hlhl);        // y (x)
    EXX
    POP     HL
    EXX
    POP     DE          ; pop(dede);        // nil (y)
    EXX
    POP     DE
    PUSH    BC          ; push(bcbc);       // angle
    EXX
    PUSH    BC
    PUSH    AF          ; push(af);         // angle i
    CALL    SRDEDE      ; dede >>= af;
    EXX
    BIT     7,B         ; if (bcbc >= 0) {
    EXX
    JR      NZ,SNCSL2
    AND     A
    SBC     HL,DE       ;    hlhl -= dede;
    EXX
    SBC     HL,DE
    EXX
    JR      SNCSE2
SNCSL2:                 ; } else {
    ADD     HL,DE       ;    hlhl += dede;
    EXX
    ADC     HL,DE
    EXX
SNCSE2:                 ; }
    LD      E,L         ; dede = hlhl;
    LD      D,H
    EXX
    LD      E,L
    LD      D,H
    EXX
    LD      C,(IX+0)    ; rpop(bcbc);       // R: nil (y+1)
    INC     IX
    LD      B,(IX+0)
    INC     IX
    EXX
    LD      C,(IX+0)
    INC     IX
    LD      B,(IX+0)
    INC     IX
    EXX
    POP     AF          ; pop(af);          // angle (i)
    POP     HL          ; pop(hlhl);        // nil (angle)
    EXX
    POP     HL
    PUSH    BC          ; push(bcbc);       // y+1
    EXX
```

```
    PUSH   BC
    EXX                ; push(dede);       // y+1 x+1
    PUSH   DE
    EXX
    PUSH   DE
    PUSH   AF          ; push(af);         // y+1 x+1 i
    SUB    CM+1        ; af -= M+1;
    JR     NC,SNCSL3   ; if (af < 0) {
    LD     E,(IY+0)    ;    dede = *iy++;
    INC    IY
    LD     D,(IY+0)
    INC    IY
    EXX
    LD     E,(IY+0)
    INC    IY
    LD     D,(IY+0)
    INC    IY
    EXX
    JR    SNCSE3
SNCSL3:                ; } else {
    INC    A           ;    ++af;
    LD     DE,(ATR+32) ;    dede = atr_fixed[M];
    EXX
    LD     DE,(ATR+34)
    EXX
    CALL   SRDEDE      ;    dede >>= af;
SNCSE3:                ; }
    EXX                ; if (hlhl >= 0) {
    BIT    7,H
    EXX
    JR     NZ,SNCSL4
    AND    A           ;    hlhl -= dede;
    SBC    HL,DE
    EXX
    SBC    HL,DE
    EXX
    JR    SNCSE4
SNCSL4:                ; } else {
    ADD    HL,DE       ;    hlhl += dede;
    EXX
    ADC    HL,DE
    EXX
SNCSE4:                ; }
    POP    AF          ; pop(af);      // y+1 x+1 (i)
    LD     C,L         ; bcbc = hlhl;
    LD     B,H
    EXX
    LD     C,L
    LD     B,H
    EXX
    POP    DE          ; pop(dede);    // y+1 (x+1)
    EXX
    POP    DE
    EXX
    POP    HL          ; pop(hlhl);    // nil (y+1)
    EXX
    POP    HL
    EXX
    INC    A           ; ++af;
    CP     CN          ; } while (af-CN < 0);
    JP     C,SNCSDO    ;
                       ; *cos = dede;
    LD     C,L         ; bcbc = hlhl;
    LD     B,H
    EXX
```

```
    LD      C,L
    LD      B,H
    EXX
    RET                 ; return bcbc;

    END
```

Conclusions

This paper comes 30 years late, a good Z80 floating point package was needed in 1976. But I really liked to write it. Today there are still microcontrollers around that have no floating point unit. The reason for these low performance chips are cost and power consumption. At the high end the CORDIC algorithm is used in FPGAs which want to process Giga-samples per second. Maybe I will work with CORDIC algorithm in hardware at my next project after using Kalman Filter (KF) and Least Means Square (LMS) already. Who knows?
Another thing I liked in this writing was the ability to cover the different interpretations of the bits. As integer, unsigned fixed-point format, signed fixed-point format or floating point number, which is a struct (record) for me. At the end I can only say: stay healthy and clean and share your knowledge!

References

Every serious assembler programmer should read the Altair BASIC and ZX-81 ROM listing to get an impression what can be done with dense assembler code. Another high-light is the ZX-81 Chess program that runs in 672 Bytes of RAM. No tricks were used in all cases, like use of undocumented op-codes or jumps in the middle of op-codes.

Volder, J.E.; 1959; The CORDIC Trigonometric Computing Technique; IRE Transactions on Electronic Computers, V. EC-8, No. 3, pp. 330-334

Meggitt, J. E.; 1962; Pseudo Division and Pseudo Multiplication Processes; IBM Journal, Res & Dev, April 1962, 210-226

General Electric; 1966; GE-625 / 635 Programming Reference Manual

IBM; 1966; IBM 1130 Subroutine Library; File No. 1130-30

Walther, J. S.; 1971; A unified algorithm for elementary functions; Spring Joint Computer Conf., 1971, proc., pp379-385

Laporte, Jacques; http://www.jacques-laporte.org/ has disassembled the HP-35 ROM. His page hosts the Volder, Meggitt and Walther papers. Even a CORDIC for dummies tutorial is available.

Baykov, Vladimir; 1972; Hardware implementation of the elementary functions by digit-by-digit (CORDIC) technique; PhD thesis, Leningrad (St.-Petersburg)

Walther; 1973; US Patent 3766370; Elementary floating point CORDIC function processor and shifter (HP-2116 CORDIC Coprocessor)Zaks, Rodney; 1980; Programming the Z80; Sybex

Maples, Michael; 1975; UCRL-51940 Floating-Point Package for Intel 8008 and 8080 Microprocessors; Lawrence Livermore Laboratory

Davidoff, Monte; 1975; Altair BASIC Math package; Microsoft

Baykov, Smolov; 1975; Hardware implementation of elementary functions in computers; Leningrad State University

Dickinson et al.; 1977; US Patent 4001569; General Purpose Calculator having selective data storage, data conversion and time-keeping capabilities (HP-45)

Logan, Ian, O'Hara, Frank; 1982; The Complete Timex TS1000/Sinclair ZX81 ROM Disassembly; Melbourne House

An Assembly Listing of the Operating System of the ZX81 ROM

Horn, David; 1983; Full ZX-81 Chess in 1K; Your Computer

J. Pitts Jarvis, III; 1990; A single compact routine for computing transcendental functions; Dr.Dobb's Journal

Knoppers, Peter; 1992; CORDIC Algorithm Simulation Code; Delft University of Technology

Ray Andraka, Ray; 1998; A survey of CORDIC algorithms for FPGAs; ACM/SIGDA sixth international symposium on Field programmable gate arrays

Rayappan, Christober; 2006; Software Implementation of Trigonometric Functions Using CORDIC Algorithm; Dr.Dobb's Journal

ZX-81 emulator: RPM for Linux Project Homepage

Wikipedia; CORDIC (in german)

About the author

Andre Adrian has a diplom in Computer Science from the University of Applied Science in Wiesbaden, Germany. After using a Commodore PET at high school in 1980 and a friends color TRS-80, the ZX-81 was my first own home computer. Today I work as senior engineer (mathematician, systems architect, project leader) for the German Air Traffic control agency DFS. The current project uses C++ on a 1GHz VIA Eden CPU with SSE in-line assembler code.