

Processor HC680 fictive

documentation of simulation

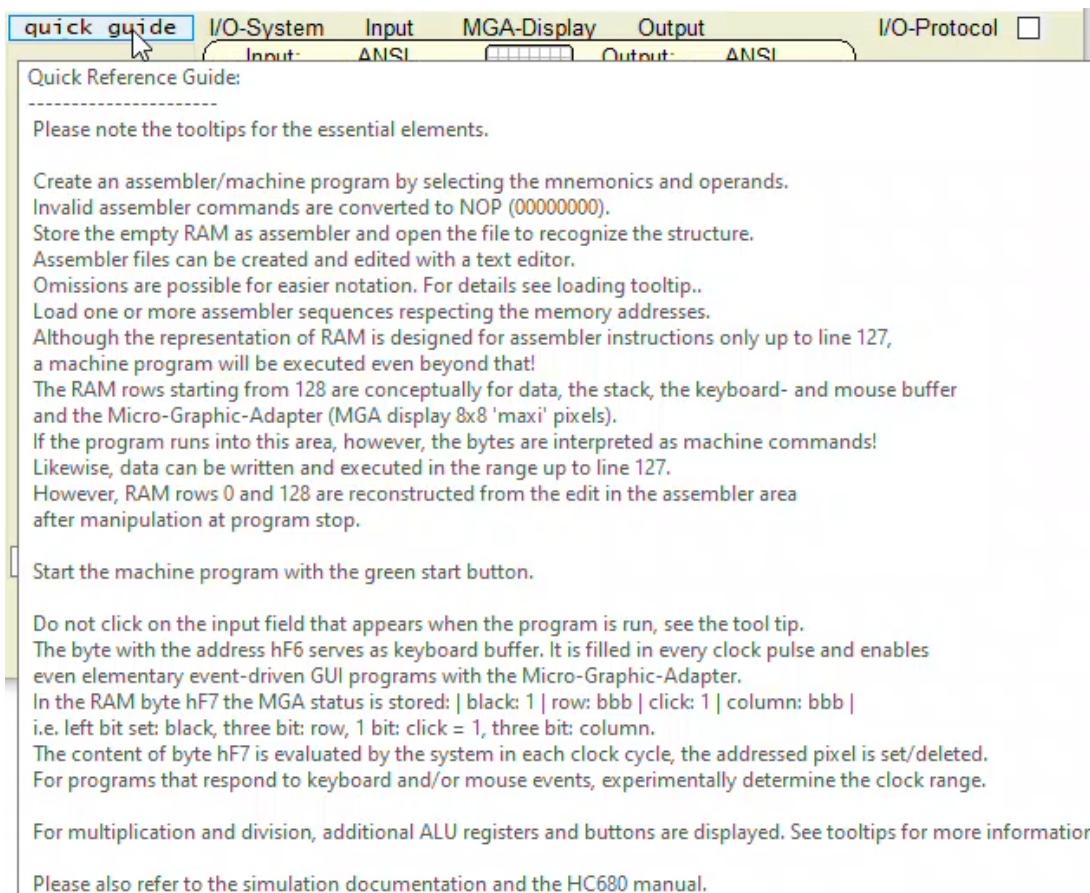


The simulation includes the structure and functionality of the processor and important assemblies of the system. Thereby both the execution of the machine program and its creation by the assembler are realized in a simulation window. Assembler programs can be stored and loaded.

data of the system

processor:	HC 680 <small>fictive</small>
processing width:	8 bit data and addresses
command length:	1 byte
address range:	256 byte
clock pulse:	Clock: single step, 0.1Hz to 255Hz, unbraked
registers:	4 general registers with shadow register, start address (ST) instruction register (IR), instruction counter (IC), status register (SR), stack pointer (SP)
flags:	Negative (N), Zero (Z), oVerflow (V), Carry (C) and I/O Bits
arithmetic unit:	Arithmetic Logic Unit (ALU), 8+1 bit sign extension for addition, multiplication by Booth, division with remainder, ALU bitwise representable
numberspace:	decimal from -128 to 127
graphic:	8x8 maxi pixel display, 8 byte shared memory
in and output:	by I/O bits: keyboard characters, decimal, hexadecimal, – output also binary
files:	by I/O bits: keyboard characters, decimal, hexadecimal, binary
breakpoints:	3 – changeable during program execution
debug:	any section, into selectable file
assembler:	40 mnemonics, 57 commands by addressing modes – see manual

The user assistance is realized via extensive, tooltips (can be switched off).



Overall view, example recursive quicksort of the file (german Datei) input.hcx, output to output.hcx.

The screenshot displays the HC6800 simulator interface. On the left, the 'Random Access Memory' window shows a list of memory addresses (00011010 to 00111100) with their corresponding hex values and command values. The central 'Processor HC6800' window shows the CPU's internal state, including registers (D0, D1, A0, A1), the Arithmetic Logic Unit (ALU), and various status registers (SR, IR, IC, SP). The rightmost 'I/O-System' window shows the input and output files (input.hcx, output.hcx) and the program being executed (Quicksort recursive).

The background color can be changed in the INI.TXT file in the folder. If the file is missing, the default applies.

The screenshot shows a dialog box titled 'delete RAM/Flags'. It contains a 'CAUTION!' message: 'Delete the entire RAM, the flag column and the comments after confirmation prompt.' There are checkboxes for 'delete RAM/Flags', 'delete Flags', and 'CACHE'.

cleanup

assembler

The screenshot shows the 'assembler / programming' window. It displays a list of mnemonics (NOP, CMP, SWD, SWM, MUL, DIV, PSA, POA, JSR, RET, JIN, JIZ, JMP, LDC, ##, INP, OUT, PSH, POP, SSR, GSR, BTS, SWN, SHL, SHR, ROL, ROR, CLR, INC, DEC, NOT, AND, OR, ADD, SUB, MOV, LOD, STO, RCL, CPY) and a list of assembly commands (JMP RG, IC <- Reg xx, JuMP). The window also shows the 'ASSEMBLER-COMMAND' section with fields for address, operation code, mnemonic, and comment.

adress
op code
flags

data

The screenshot shows a dialog box titled 'program | data'. It contains a 'Name of the program to save or after loading:' field with the value 'Quicksort recursive'.

operands

display

example with click areas
(numeric input here with keyboard)

stackpointer, keyboard byte,

pxel row, MGA-RAM

Address	Hex	Value	Comment
11110011	F3	00000000	0 nuu
11110100	F4	00000000	0 h00 <-SP
11110101	F5	00000000	0 h00
11110110	F6	00110001	1 h31 ->Ta
11110111	F7	00000000	0 h00 >Pix
11111000	F8	00100000	32 h20 row1
11111001	F9	01100111	103 h67 row2
11111010	FA	00100000	32 h20 row3
11111011	FB	00100111	39 h27 row4
11111100	FC	00100000	32 h20 row5
11111101	FD	00000010	2 h02 row6
11111110	FE	11100111	-25 hE7 row7
11111111	FF	00000010	2 h02 row8

The MGA bit pattern is clearly visible.

The screenshot shows a dialog box titled 'mnemonics'. It contains a list of mnemonics (NOP, JIN, JIZ, JMP, LDC, ##, INP, OUT, PSH, POP, SSR, GSR, BTS, SWN, SHL, SHR, ROL, ROR, CLR, INC, DEC, NOT, AND, OR, ADD, SUB, MOV, LOD, STO, RCL, CPY) and a list of assembly commands (JMP RG, IC <- Reg xx, JuMP).

comments to
assembler commands,
list with explanation in short

The screenshot shows a dialog box titled 'RAM navigation'. It contains a list of memory addresses (11110011 to 11111111) and a list of assembly commands (JMP RG, IC <- Reg xx, JuMP).

comments command

comment to the command

Enter a comment to the command - even longer than input field, will be visible in the tooltip.

data data comment

Enter a comment to the constant - even longer than input field, will be visible in the tooltip.

The complete list of assembler commands with explanation in short form.

Mnem	OP	OP	- Meaning Command/Addressing -	FLAG
NOP			No Operation	...
CMP			CoMPare D0, D1	NZVC
SWD			SWap D0, D1	...
SWM			SWap Memory Adr(D0), Adr(D1)	...
MUL			MULTiplication D0 <- D0*D1	NZVC
DIV			DIVision D0 <- D0/D1 Rest SDO	NZVC
PSA			PUsh All Stack <- A,D,SR Reg.	...
POA			POp All SR,D,A reg. <- Stack	NZVC
JSR			Jump SubR. S<-IC+1,IC<-IC+A1/IC<-A0+A1+ST	...
JIN +A			N=1: IC <- IC + A1 Jump If Negative	...
JIZ +A			Z=1: IC <- IC + A1 Jump If Zero	...
JMP +A			IC <- IC + A1 JuMP	...
RET			RETTurn subroutine IC <-Stack	...
LDC Rg			LoaD Constant reg xx <- next Byte ##	NZ..
## CCCC CCCC			## Constant Byte	...
JIN IA			N=1: IC <- A0 +A1 +ST Jump If Negative	...
JIZ IA			Z=1: IC <- A0 +A1 +ST Jump If Zero	...
JMP IA			IC <- A0 +A1 +ST JuMP	...
JIN RG			N=1: IC <- Reg xx Jump If Negative	...
JIZ RG			Z=1: IC <- Reg xx Jump If Zero	...
JMP RG			IC <- Reg xx JuMP	...
INP RG			reg xx <- INPut	NZ..
OUT RG			OUTput <- reg xx	...
PSH RG			PUsh Stack <- reg xx	NZ..
POP RG			POP reg xx <- Stack	NZ..
SSR RG			Set Shadow Register reg xx -> Sxx	NZ..
GSR RG			Get Shadow Register reg xx <- Sxx	NZ..
BTS RG			BitTest reg xx with Shadow register	NZ..
SWN RG			SWap Nibble reg xx	NZ..
SHL RG			Shift Left reg xx	NZ..C
SHR RG			Shift Right reg xx	.Z.C
ROL RG			ROtate Left reg xx	NZ..C
ROR RG			ROtate Right reg xx	NZ..C
CLR RG			CLeaR Register reg xx <- 0	.Z..
INC RG			INCRement reg xx	NZVC
DEC RG			DECRement reg xx	NZVC
NOT RG			Reg xx <- NOT reg xx (bitwise)	NZ..
AND RG RG			Reg yy <- reg yy AND reg xx	NZ..
OR RG RG			Reg yy <- reg yy OR reg xx	NZ..
ADD RG RG			Reg yy <- reg yy + reg xx	NZVC
ADD RG [RG]			Reg yy <- reg yy + Adr(reg xx)	NZVC
SUB RG RG			Reg yy <- reg yy - reg xx	NZVC
SUB RG [RG]			Reg yy <- reg yy - Adr(reg xx)	NZVC
MOV[RG] RG			MOVE Adr(reg yy) <- reg xx	NZ..
MOV RG [RG]			MOVE reg yy <- Adr(reg xx)	NZ..
MOV RG RG			MOVE reg yy <- reg xx	NZ..
MOV RG SR			MOVE reg yy <- SR	NZ..
MOV RG SP			MOVE reg yy <- SP	NZ..
MOV SR RG			MOVE SR <- reg xx	NZ..
MOV SP RG			MOVE SP <- reg xx	NZ..
MOV SR CC			MOVE SR <- CC.. (2bit IO)	...
MOV RG IA			MOVE reg xx <- Adr(A0+A1+ST)	NZ..
MOV IA RG			MOVE Adr(A0+A1+ST) <- reg xx	NZ..
LOD RG			LoaD Data reg xx <- Adr(h80+A1+ST)	NZ..
STO (IO)			STOre file <- Adr(h80+A1+ST) D0 byte /D1	.Z..
RCL (IO)			ReCaLL Adr(h80+A1+ST)<- file /D1 read D0	.ZV.
CPY			CoPY Adr(A1)<-Adr(A0) D0 byte via D1	.ZV.
STP			StoP P. stop, flags from pre-command	vvvv

I/O-Protocol ☒

>In Out>

€ €

-128 -128

h80 h80

>1000 0000.

.1000 0000>

input - output protocol

In- and Output protocol commands INP and OUT - display of values

interpretation according to IO bits in the Status Register

00 keyboard character

01 decimal numbers -128 ... 127

10 hexadecimal numbers two digits xx

11 binary numbers 8 bit (over both columns!)

>Input.

.Output>

... visibility

I/O-Protocol ☒

>In Out>

Visibility of the input and output protocol.

... clear protocol

I/O-Prot. delete ☒

Delete input and output protocol without asking.

The 9-bit ALU at work.

View the individual bits at clock speed.

Arithmetic Logic Unit

111110100

111111101

111000

10001

bitwise ☒

Display the calculation process in the ALU bitwise.

Status Register

0100

IOXY

Stack Pointer

During the bitwise representation in the ALU:

- * no running keyboard input
- * no mouse handling
- * no stopping possible with start button

assembler command MUL

Arithmetic Logic Unit

000000000 00000100 A

111111100 11111100 S

000000 x

11100 1000001 0

bitwise ☒ 4» Booth

Booth multiplication

(bitwise in the ALU with additional temporary registers)

--- multiplication according to Booth ---

Status Register

00000001

IOXYNZVC

FLAG

Stack Pointer

11110101

ST

0010000100

program | data

artadr. from to | from

hex

Op. 1 Op. 2

cc

ANSI

Register D0 is temporarily stored in temporary ALU register A (for addition). The two's complement -D0 is formed and stored in the temporary register S. The 16-bit result is calculated step by step in the bottom line. For this purpose, the left byte is initialized with 0 and D1 is stored on the right and the auxiliary bit x is set to 0 next to it.

Sequence control takes place after the last two bits of the result line (with auxiliary bit x):

>> The result line is shifted 1 bit to the right, the auxiliary bit is dropped out, on the left, the bit identical to the bit shifted away is added as a sign extension.

00, 11 >> only >>

01 add >> D0 is added, then >>

10 sub >> -D0 is added, then >>

The left byte of the last line is copied to the top line as a subtotal. (with sign expansion)

After 8 steps, the 16bit result (without auxiliary bit x) is obtained in the lower double register. The right byte (low byte) is brought to D0.

Then check for displayability in the range -128 ... 127 and set the flags.

Division bitwise in the ALU with additional temporary registers.

assembler command DIV

Arithmetic Logic Unit 100

Vz D0 D1 00000000 S

00111100 --- integer division with remainder ---

00000100

bitwise

Status 0

Stack 1

ST 1

000111100

program

rtadr. from

hex

Op. 1

ANSI

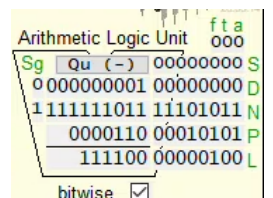
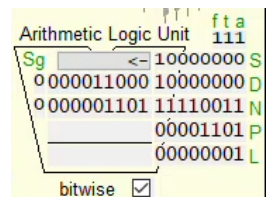
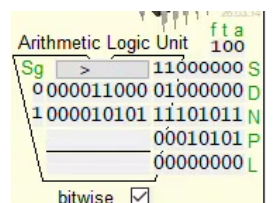
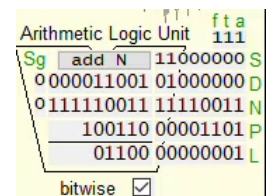
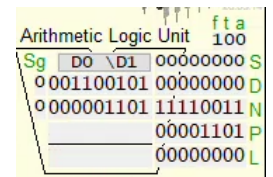
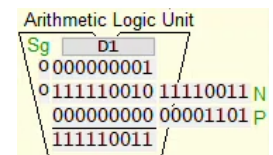
The signs of the dividend and divisor are stored in temporary ALU memory bits (Vz).
The two's complement is formed from the divisor D1 (temporary ALU registers: N negative, P positive).
If the dividend in D0 is negative, its two's complement is formed first.
The division of the positive operands is realized with ALU control bits via the ALU shift registers S, D.
For this purpose, the positive dividend in the ALU is moved 'fittingly' over the positive divisor.
This is achieved by bit comparison of the bits of the operands standing on top of each other.
The difference is then formed by adding the negative divisor.
The relevant shifts and additions generate the quotient in the temporary solution register L.
Based on the stored signs, the sign may now be changed (solution, remainder).

The control bits are shown in the upper right corner of the ALU in the following order:
f (first loop) t (two) a (add)

f: First total pass in bit comparison of dividend and divisor (f = 1).
The dividend is moved to the right into the auxiliary register D of the dividend if required.
In the shifting pointer register S, a 1 is inserted from the left.
t: In the first subsequence the bits are compared from the left (symbol |), possibly shifted (symbols < >). If this means that the subsequent bits in the second subprocess must also be checked to determine the size (symbol :), t = 1 is set.
a: Solution bit 1 is appended to the quotient, then addition of the negative divisor (current remainder), a bit from the auxiliary register D of the dividend is shifted after (symbol < -).
This is only done if there is still a 1 in the slider pointer register S above it on the left.

Then, depending on the sign bit in Vz, a sign reversal may occur.
The quotient is brought to D0, the remainder to SD0 (sign remainder like sign dividend).

Excerpts:



data,
store in file

Out output.hcx

entra

Output to data file with the STO command.

File structure: Text file, one value per line.
Content of the file - interpretation according to IO bits in the Status Register

arithm

00 keyboard character

01 decimal numbers -128 ... 127

10 hexadecimal numbers two digits xx

11 binary numbers 8 bit

If no data was written, the Z flag is set.
If no file name is entered, the file selection is displayed.

data, read in from file

file input.hcx In - Out output.hcx

ccess

Data re

File structure: Text file, one value per line.
Content of the file - interpretation according to IO bits in the Status Register

0000

00 keyboard character

01 decimal numbers -128 ... 127

10 hexadecimal numbers two digits xx

11 binary numbers 8 bit

Address

0000

If no data was read, the Z flag is set.
If there is too much data in the file, the program terminates at the end of RAM and sets the V flag.
It is also set by values that do not respect the range.
If no file name is entered, the file selection is displayed.

input (continuously a keyboard character or with command INP)

HC680 binary code

input ...

clock: 1.0 Hz

Breakpoints

Input: ANSI

-128 128 €

10000000 h80 OK

Output: ANSI

>In Out>

file inp

Processor

Data reg

D0 00000

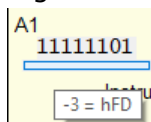
D1 00000

Address

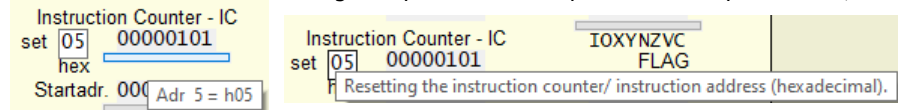
Continuous input of a keyboard character. Input possible with Alt xxxx.
The input takes place to each clock also without clicking into the input field!
The binary ANSI code is stored at address hF6.

With the command INP alternatively also input of a decimal number in the range -128 ... 127 or enter a hexadecimal number in the form hxx or Hxx or \$xx.
Hexadecimal digits 0..9, A..F or a..f - invalid digits are converted to 0.
For INP, the input must be acknowledged with OK.
The input is then stored in the register and at address hF6 and displayed in the input/output log. The display there is according to the content of the IO bits in the status register: 00 character, 01 decimal, 10 hexadecimal, 11 binary.

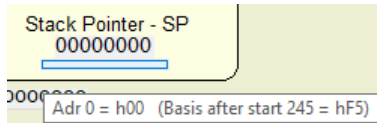
register content (binary, plus decimal, hexadecimal in tooltip)



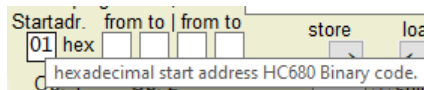
Instruction Counter (at single step –also at breakpoint– manually resettable)



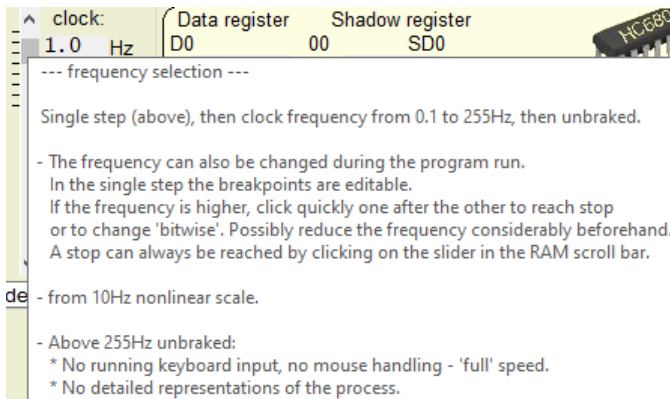
base address Stack Pointer



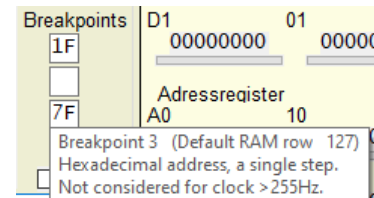
Start address



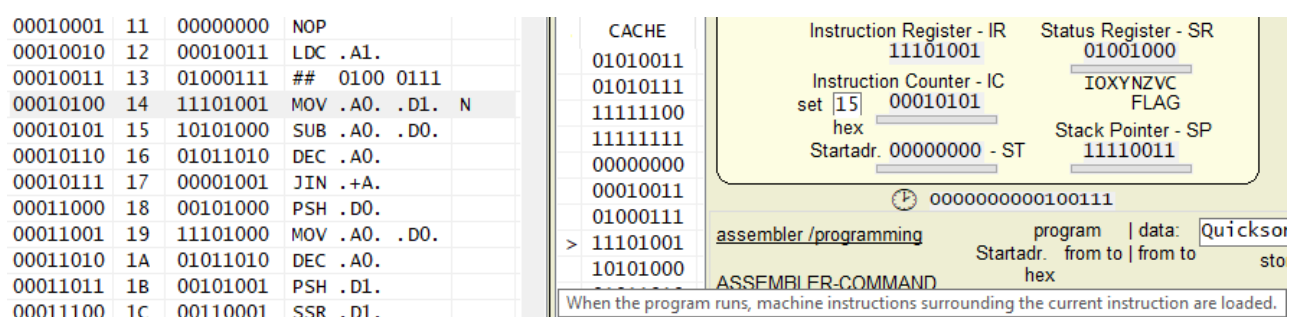
takt



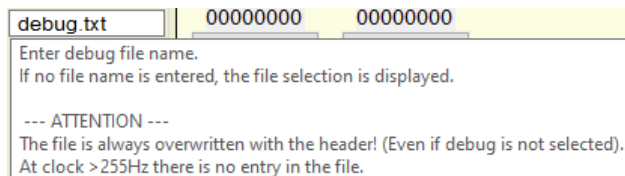
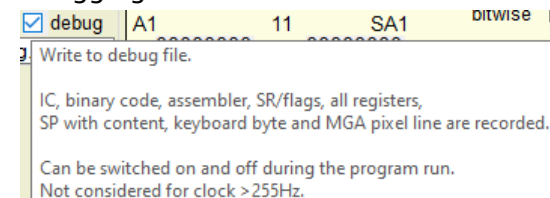
Breakpoints



cache



debugging



example in assembler "sum exceeded until 50" – stored as a text file

```
HC680 assembler ; sum exceeded until 50
00: ST          ; Startadr.
Adr Mnm Op Op ; - comment -
00: MOV .SR. .01. ;output decimal
01: LOD .D1. ;load target sum from h80
02: INC .A1. ;incrementing the address index register A1 to load the next constant
03: LOD .A1. ;load jump difference for IC relative jump in A1
04: PSH .D0. ;put initial value 0 on the stack for cumulating
05: INP .A0. ;-INPUT- data into address register - is possible!
06: POP .D0. ;get cumulated sum from stack
07: ADD .D0. .A0. ;sum with input value
08: PSH .D0. ;put sum back on stack
09: CMP ;compare sum with target sum (D0-D1 without result)
0A: JIN .+A. ;IC-relative jump to input, if target sum is not yet reached
0B: OUT .D0. ;-OUTPUT-
0C: STP ;done!
80: 50 h32 ;sum up to 50
81: -5 hFB ;IC-relative destination address (five commands back)
```

When creating with a text editor, the colon, the dots before and after the operands and the semicolon can be omitted.

After Mnm and between the operands up to 4 spaces are allowed, before Mnm two.

Before the data up to five blanks are allowed for separation, thus right-justified notation is possible.

documentation of simulation

processor HC680

page 5 von 6

load, store, shift assembler file

load
←

Load an HC680 assembly file into the working memory.

Faulty assembler commands are interpreted as NOP.
The colon after the command address is optional, one or two spaces are sufficient for it.
Also, the dots around the operands are optional, then put one to four separator spaces.
In the data area from address h80 only the left value or character is read.
For a better overview, the data can be set right-justified.
The colon after the data address can be omitted, up to five spaces are allowed.
The 2nd column (uniform hexadecimal) is optional. It will be recalculated.
From column 21 the comment is read in, the semicolon in front of it is optional.
Please note the start address stored in the file.

store → **load** ←

☐ Store a HC680 assembler file.

o th The start address is stored.
From start address (min. 00) to end address (max. 7F) the program,
the data from (min. 80) to (max. FF) are stored.
ient If no values are entered (the part) is not stored.

shift

Shift the assembly instructions and/or the data in RAM.

Specify the offset in decimal or hexadecimal hxx with sign.
It is shifted from the entered start to the end addresses.
The existing contents are overwritten, free lines are set to zero.
The RAM section is stored as asm_tmp.txt.
The complete RAM is saved before under asm_bak.txt.

-23 | shift
-h17 | shift of program and/or data in memory, decimal or hexadecimal (-)hxx resp. Hxx, \$xx

address ranges of the assembler file

program | data: test program

dr.	from	to	from	to	store	load
hex	00	6A	80	BF	→	←

1 hexadecimal start address of the binary code to store/move in the assembly file.

program | data: test program

dr.	from	to	from	to	store	load
hex	00	6A	80	BF	→	←

1 hexadecimal end address program - for storage/movement

data: test program

from	to	store	load
80	BF	→	←

2 hexadecimal data address start - for storage/moving

data: test program

from	to	store	load
80	BF	→	←

2 hexadecimal data address end - for storage/movement

execute machine program / binary code

click -

HC680 binary code

execute

clock: 1.0 Hz

Intermediate stops possible

HC680 binary code

halt

clock: 0.2 Hz

HC680 binary code

halt again

clock: 0.2 Hz

bitwise in the ALU

HC680 binary code

... bit ...

clock: 0.2 Hz

slower is not possible

HC680 binary code

STOP

Step ...

clock: Step Hz

go on

continue please

full speed

HC680 binary code

Takt > 255

clock: Speed Hz

at INPUT

HC680 binary code

input ...

clock: 1.0 Hz

break off?

HC680 binary code

STOP

run further

input: input.hcx

file input.hcx

Cancel program immediately with x, otherwise click yellow circle.

done!

HC680 binary code

stopped