AchtBit Assembler

Inhaltsverzeichnis

1 .asm Syntax

1.1 Comments

```
// this is a comment
```

1.2 Constants

Decimal integer: `12345`

Hexadecimal integer: `0x7f`

Binary integer: `@--**---*` or `@00110001`

String: `"Hello world!"`

Char: `'x'`

1.3 Labels

Labels can store adresses and constants. They have the form „**labelName:**“. All labels are global and must be defined somewhere (no matter if before or after usage).

Examples:

```
label1: LD A,(HL) // label1: is defined here
        CALL label1: // label1 is used
.def label1: 0xff33 // label1: is defined here directly (0xff33)
dw label1: 0x00 // label1: will be stored as a constant
```

1.4 Adresses

.org adress, adress.: defines where the program will be stored from here on.

Example: .org 0x4000
0x4000.

.run adress: after the assembler program is completely loaded, the program pointer (PC) will be set to adress.

1.5 Import

.import modulName: will include „modulName_label.asm“. This file will be created each time modulName.asm is loaded into achtBit.

Example: the program „achtbit_rom.asm“ is loaded on startup. So the file „achtbit_rom_label.asm“ has already been stored. It contains all achtbit_rom-labels and their corresponding adresses:

```
.def alist_append1: 0x00D7
.def alist_append: 0x00D2
.def alist_clear: 0x00C6
...
.def zeichen: 0x057E
.def zeilenumbruch: 0x002A
```

Another program imports these labels and can use them like its own:

```
.import achtbit_rom
...
CALL alist_clear:
```

1.6 Include

.include modulName: inserts the program modulName.asm as if it was part of the current program.

Example:

```
.include snake_highscore // insert the program snake_highscore.asm at this point
```

1.7 Defining Data

.db data... : store constants and labels as bytes (8 bit)

.dw data... : store constants and labels as words (16 bit)

`.db „Hello“ 0x0a „world!“ 0`	`48 65 6C 6C 6F 0A 57 6F 72 6C 64 21 00`
`.dw 0xff32 sys_screen: 23662 @1001110001`	`32 FF 00 E0 6E 5C 71 02`

`.db @--****--`
`.db @-*----*-`
`.db @-*----*-`
`.db @--****--`
	`3C`
	`42`
	`42`
	`3C`

2 Instructions

2.1 Storage Types

R_D,R_S: get data from register R_S and store the result in register R_D

R_D,n: take the constant n and store the result in register R_D

r,(nn): take the data from memory at nn and and store the result in register R_D

R_D,(R_S): get data from memory at register R_S and store the result in register R_D

Only valid with the LD-Command:

(nn),R_S: get data from register R_S and store the result in memory at nn

(R_D),R_S: get data from register R_S and store the result in memory at register R_D

(R_D),n: take the constant n and store the result in memory at register R_D

2.2 Registers

These registers can be used as double (16 bit) and as single registers (8 bit).

- **AF (A**: the accumulator, **F**: this register stores the flags)
  - **BC (B,C)**
  - **DE (D,E)**
  - **HL (H,L)**
  - **SP (SpH, SpL)**: the stack pointer

IX is a 16 bit register. Used as a pointer it has a constant offset (-128 to 127). (f.e. LD (IX-8),DE)

PC is the 16 bit program pointer.

2.3 Flags

Flags give extra information after several operations and affect some.

Z-Flag (Zero-Flag): it will be set after some operation gives a zero result.

C-Flag (Carry Flag): it will be set after there is an aritmetic overflow and has an effect on some aritmetic instructions (→ ADC, RHL, ...)

P-Flag (Parity-Flag): it will be set after some instruction returns an even result.

Z, **C**, **P** tests if a flag is set; **NZ**, **NC, NP** tests if a flag is not set.

2.4 Commands

NOP: no operation; just uses time.

HLT: halt (pause) the system

CLC: clear the carry flag

SETC: set the carry flag

LD Destination, Source: load data from source into the destination

ADD Destination, Source: add source to destination

ADC Destination, Source: add with carry flag

ADC B,C:	**C-Flag**	1	1
	B	0000011	3
	+C	0000100	4
	=B	0001000	8
	C-Flag	0	0

ADC B,C:	**C-Flag**	0	0
	B	11000011	195
	+C	01000110	70
	=B	0001001	9
	C-Flag	1	1

SUB Destination, Source: substract source from destination

SBC Destination, Source: substract with carry flag

MUL Destination, Source: multiply destination by source

DIV Destination, Source: divide destination by source

MOD Destination, Source: get the rest of the division destination by source

AND Destination, Source: aritmetic and

OR Destination, Source: aritmetic or

XOR Destination, Source: aritmetic exclusive or

CMP Destination, Source: do a virtual „SUB Destination, Source“, that only affects the flags

CMP B,C	B>C	B<C	B==C
Flags	NC, NZ	C, NZ	NC, Z

SHL Destination: shift bitwise to the left and clear bit 0. The bit falling out is stored into the carry flag.

SHR Destination: shift bitwise to the right and clear bit 7 / 15. The bit falling out is stored into the carry flag.

RHL Destination: roll bitwise to the left and store the carry flag in bit 0. The bit falling out is stored into the carry flag.

RHR Destination: roll bitwise to the right and clear bit 7 / 15. The bit falling out is stored into the carry flag.

SHL										**RHL**									
C	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**C**	**C**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**C**
	1	0	0	1	1	0	0	1	1		1	0	0	1	1	0	0	1	1
				←	←	←								←	←	←			
1	0	0	1	1	0	0	1	0		1	0	0	1	1	0	0	1	1	

INC Destination: increment the destination.

DEC Destination: decrement the destination.

JP [flag] adress: jump to the adress, when given test if the flag is set / unset

CALL [flag] adress: store PC on stack and jump to the adress

JR [flag] offset: add the offset (-128 to 127) to PC; this is a relative jump

RET [flag]: restore PC from stack (return from a call).

RETI: return from interrupt (get PC from stack and enable further interrupts)

PUSH Source: put the source onto the stack

POP Destination: get data from stack and store it into destination.

IN register,(n): get data from port n and store it in the register.

OUT (n),register: get data from the register and send it to port n.

3 Ports

Device	**Port**	**IN register,(port)**	**OUT (port),register**
Keyboard	0	get next char	255: clear buffer
Random	1	get random number [0..255]	-
Sound	2	get loudness of next note	Set loudness of next note
	3	get length of next note	set length of next note
	4	get length unit of next note (1/n)	set length unit of next note (1/n)
	5	1 = playback finished 0 = still playing	0 = stop playback and clear queue 1..255 = note to append for playback

4 Example program „helloworld.asm“


```
.import roms/achtBit_rom

.org 0x4000
.run 0x4000

helloworld:
      CALL clear_scr:
      CALL helloworld_colors:
      LD HL,0x0900
      LD (sys_cursor_x:),HL
      LD HL,helloworld_str:
      CALL print_str:
helloworld_loop:
      JR helloworld_loop:

helloworld_colors:
      LD A,0 // color
      LD B,8 // lines
      LD HL,sys_screen_color:
      CALL helloworld_colors_lines:
      LD A,0x80 // color
      LD B,8 // lines
      LD HL,sys_screen_color:
      ADD HL,480 // 12 line x 40 chars
      CALL helloworld_colors_lines:
      RET

helloworld_colors_lines:
helloworld_colors_outerLoop:
      LD C,40
helloworld_colors_innerLoop:
      LD (HL),A
      INC HL
      DEC C
      JR NZ helloworld_colors_innerLoop:

      ADD A,0x10
      DEC B
      JR NZ helloworld_colors_outerLoop:

      RET


helloworld_str:
      db "        H E L L O" 0x0a
      db "                   W O R L D !" 0
```