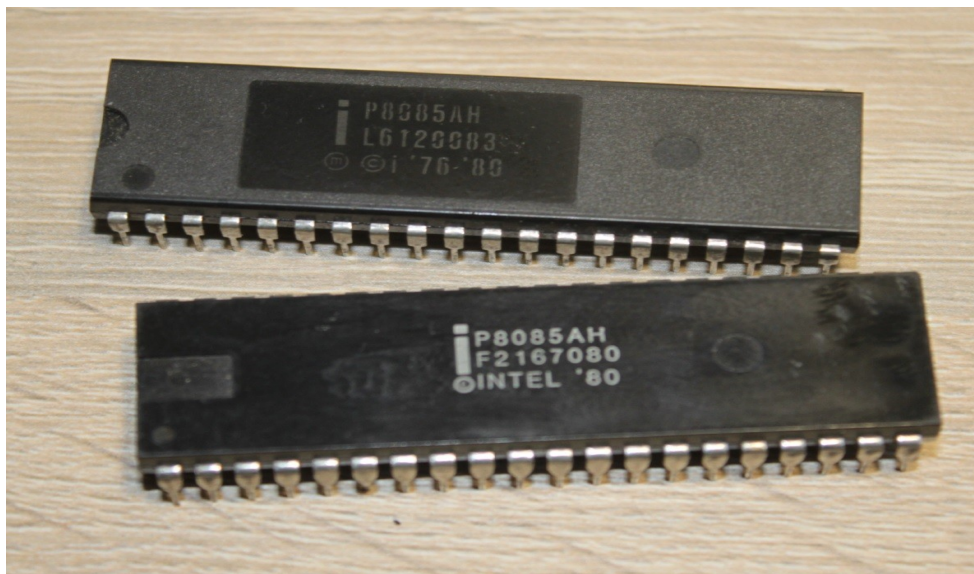


Andrzej Pawluczuk



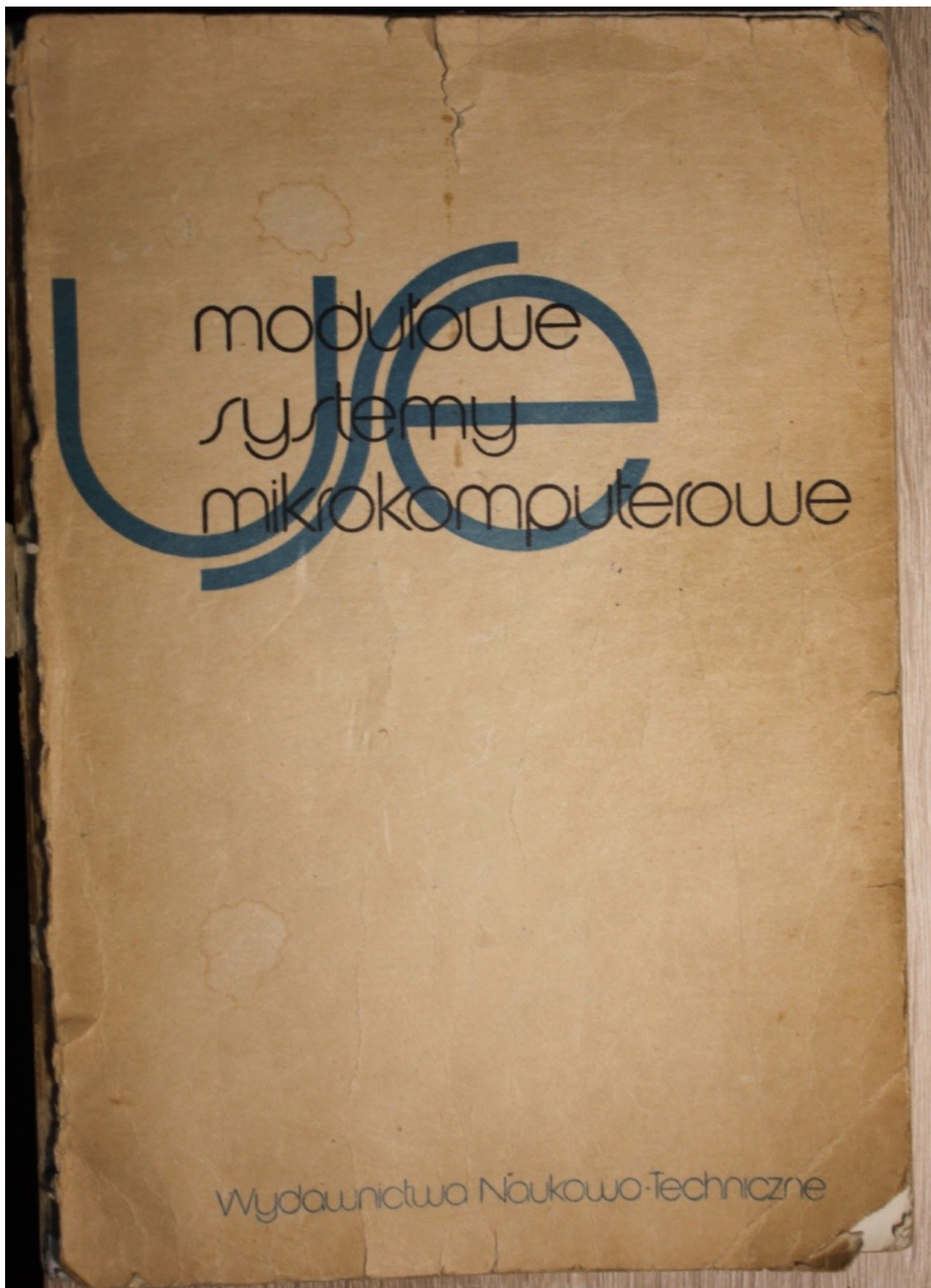
Emulator procesora i8085

Białystok, wrzesień 2023 (rev. 1.01)



Nadeszła w końcu ta chwila, by zająć się stworzeniem softu dla generatora opartego na procku i8085. Ot wala mi się kilka takich w szufladzie, więc uznałem, że to dobry pomysł, by zatrudnić ich do jakiegoś działania. Sprzęciora już nawet zrobiłem, więc pora na kolejny krok do celu. Jakiś czas tam napisałem sobie program, który kompiluje program w asm i generuje kod binarny w formacie hex. Być może i są jakieś niekomercyjne narzędzia pozwalające na tworzenie softu dla tych procków. Prawdę mówiąc to specjalnie nie szukałem. Zacząłem nawet tworzyć kawałek softu, jednak zaczęło mi się trochę kaszanić w programie. Bo to człowiek już dawno nie pisał w assemblerze, a tu w dodatku jeszcze takim trochę dziwnym, gdyż będąc przyzwyczajony do składni zilogowej, która jest jasna i klarowna, to należy wrócić do składni inteloskiej. Ta już nie jest tak elegancka, ale nie ma co narzekać i wytykać intelowi jakieś uwagi. Chłopcy zrobili to jak zrobili i tyle.

Rozwiązaniem tego problemu jest posiadanie emulatora działania procka: czy to sprzętowego czy programowego. W kwestii emulatorów sprzętowych, to ... są absolutnie poza zasięgiem. Ten procek był na topie jakieś 50 lat temu i dzisiaj, nawet jeżeli takowe istnieją to będą kosztowały majątek. Pozostaje rozwiązanie w wariacie emulatora programowego. Generalnie nie jest to jakiś wielki problem, ot program jak program, jest to napisania. Pozostaje kwestia dostępu do właściwej wiedzy. Tu staje się przydatna moja stara książka, która wniosła wiele w rozumienie techniki mikroprocesorowej: *praca zbiorowa, „Modułowe systemy mikrokomputerowe”*.



W owej książce jest podana lista instrukcji procka i8080 i rozszerzenia dla procka i8085 (oczywista, że najwięcej miejsca jest poświęcone Z80). Ogólnie wiadomo, że Z80 jest kompatybilny pod względem kodów binarnych z prockami intela dysponując jednak znacząco większą liczbą instrukcji. Ano zostało to zrobione poprzez wykorzystanie „dziur” w kodach intelowych (tych od i8080). Procek i8085 również wykorzystał wolne miejsca, gdyż doszły dwie instrukcje (RIM i SIM). No więc będąc wyposażony w szczegółową wiedzę, postanowiłem stworzyć program emulatora.

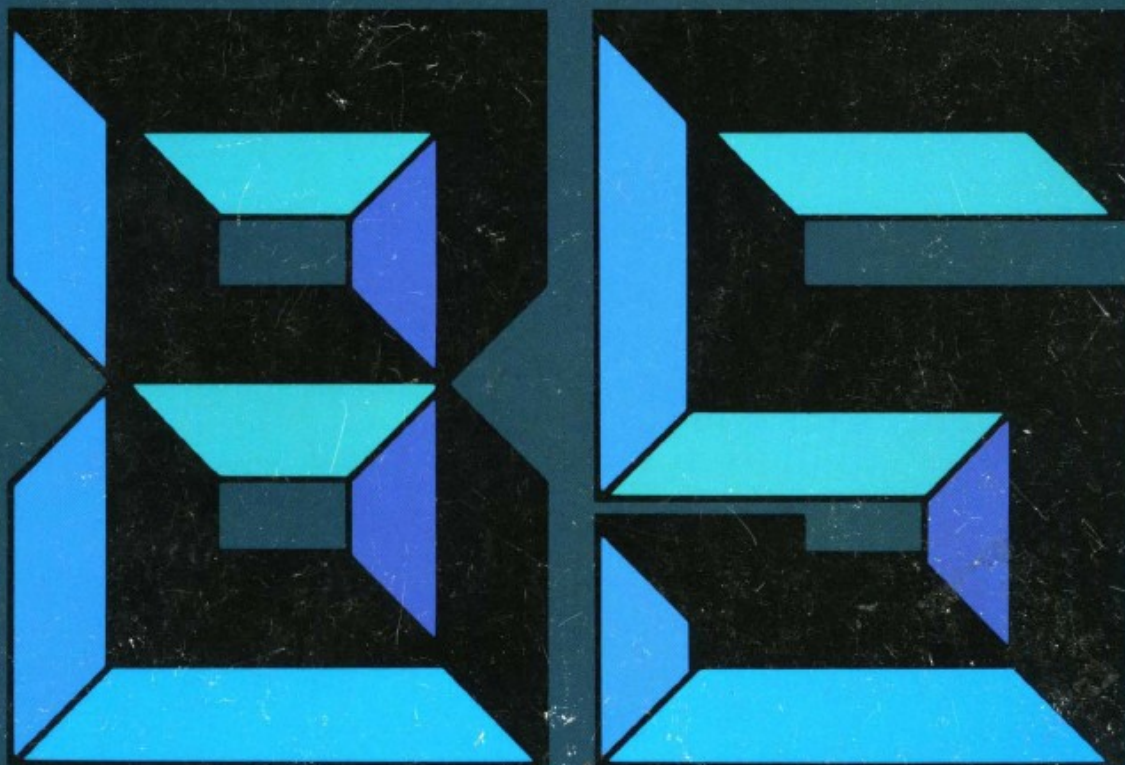
Kod mnemoniczny	Kod binarny	Liczba bajtów	Znaczenie
1	2	3	4
0 SIM	00110000 30	1 ✓	Ustaw rejestr maskowania przerwań, zeruj przerwanie 7.5 oraz dokonaj szeregowej transmisji danych
0 RIM	00100000 20	1 ✓	Odczytaj stan rejestru przerwań, rejestru maskowania przerwań i stan szeregowego wejścia danych
0 SHLX	11011001 D9	1 ✓	Prześlij do pamięci zawartość pary rejestrów H □ L
0 LHLX	11101101 FD	1 ✓	Prześlij zawartość pamięci do pary rejestrów H □ L
0 LDHI n	00101000 ←n→ 28	2 ✓	Dodaj do zawartości pary rejestrów H □ L wartość n i wynik umieść w parze rejestrów D □ E
0 LDSI n	00111000 ←n→ 28	2 ✓	Dodaj do zawartości rejestru wskaźnika stosu wartość n i wynik umieść w parze rejestrów D □ E
0 DSUB	00001000 08	1 ✓	Odejmij zawartości par rejestrów
ARHL	00010010 12	1 ✓	Przesuń arytmetycznie w prawo zawartość pary rejestrów H □ L

376

Jednak coś mnie tknęło i postanowiłem podane informacje zweryfikować. Dotarłem do oryginalnej dokumentacji tego procka:

intel

The MCS[®]-80/85 Family User's Manual



Order Number: 205775-002

gdzie jasno jest postawiona sprawa, że praca zbiorowa jest polskim nadużyciem i mija się trochę z prawdą. Poza tym jest więcej istotnych szczegółów, których znajomość jest niezbędna do stworzenia poprawnie działającego programu.

THE INSTRUCTION SET

LXI *rp*, *data 16* (Load register pair immediate)

(*rh*) ← (byte 3),

(*rl*) ← (byte 2)

Byte 3 of the instruction is moved into the high-order register (*rh*) of the register pair *rp*. Byte 2 of the instruction is moved into the low-order register (*rl*) of the register pair *rp*.

0	0	R	P	0	0	0	1
low-order data							
high-order data							

Cycles: 3

States: 10

Addressing: immediate

Flags: none

LHLD *addr* (Load H and L direct)

(*L*) ← ((byte 3)(byte 2))

(*H*) ← ((byte 3)(byte 2) + 1)

The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register *L*. The content of the memory location at the succeeding address is moved to register *H*.

0	0	1	0	1	0	1	0
low-order addr							
high-order addr							

Cycles: 5

States: 16

Addressing: direct

Flags: none

LDA *addr* (Load Accumulator direct)

(*A*) ← ((byte 3)(byte 2))

The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register *A*.

0	0	1	1	1	0	1	0
low-order addr							
high-order addr							

Cycles: 4

States: 13

Addressing: direct

Flags: none

SHLD *addr* (Store H and L direct)

((byte 3)(byte 2)) ← (*L*)

((byte 3)(byte 2) + 1) ← (*H*)

The content of register *L* is moved to the memory location whose address is specified in byte 2 and byte 3. The content of register *H* is moved to the succeeding memory location.

0	0	1	0	0	0	1	0
low-order addr							
high-order addr							

Cycles: 5

States: 16

Addressing: direct

Flags: none

STA *addr* (Store Accumulator direct)

((byte 3)(byte 2)) ← (*A*)

The content of the accumulator is moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.

0	0	1	1	0	0	1	0
low-order addr							
high-order addr							

Cycles: 4

States: 13

Addressing: direct

Flags: none

LDAX *rp* (Load accumulator indirect)

(*A*) ← ((*rp*))

The content of the memory location, whose address is in the register pair *rp*, is moved to register *A*. Note: only register pairs *rp* = *B* (registers *B* and *C*) or *rp* = *D* (registers *D* and *E*) may be specified.

0	0	R	P	1	0	1	0
---	---	---	---	---	---	---	---

Cycles: 2

States: 7

Addressing: reg. indirect

Flags: none

No i oczywista, jest lista instrukcji wraz z ich kodami. No i nawet bez okularów widać, że procek nie realizuje instrukcji opisanych w polskiej książce.

8085A

8085A CPU INSTRUCTIONS IN OPERATION CODE SEQUENCE
Table 5-2

OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC
00	NOP	28	DCX H	56	MOV D,M	81	ADD C	AC	XRA H	D7	RST 2
01	LXI B,D16	2C	INR L	57	MOV D,A	82	ADD D	AD	XRA L	D8	RC
02	STAX B	2D	DCR L	58	MOV E,B	83	ADD E	AE	XRA M	D9	—
03	INX B	2E	MVI L,D8	59	MOV E,C	84	ADD H	AF	XRA A	DA	JC Adr
04	INR B	2F	CMA	5A	MOV E,D	85	ADD L	B0	ORA B	DB	IN D8
05	DCR B	30	SIM	5B	MOV E,E	86	ADD M	B1	ORA C	DC	CC Adr
06	MVI B,D8	31	LXI SP,D16	5C	MOV E,H	87	ADD A	B2	ORA D	DD	—
07	RLC	32	STA Adr	5D	MOV E,L	88	ADC B	B3	ORA E	DE	SBI D8
08	—	33	INX SP	5E	MOV E,M	89	ADC C	B4	ORA H	DF	RST 3
09	DAD B	34	INR M	5F	MOV E,A	8A	ADC D	B5	ORA L	E0	RPO
0A	LDAX B	35	DCR M	60	MOV H,B	8B	ADC E	B6	ORA M	E1	POP H
0B	DCX B	36	MVI M,D8	61	MOV H,C	8C	ADC H	B7	ORA A	E2	JPO Adr
0C	INR C	37	STC	62	MOV H,D	8D	ADC L	B8	CMP B	E3	XTHL
0D	DCR C	38	—	63	MOV H,E	8E	ADC M	B9	CMP C	E4	CPO Adr
0E	MVI C,D8	39	DAD SP	64	MOV H,H	8F	ADC A	BA	CMP D	E5	PUSH H
0F	RRC	3A	LDA Adr	65	MOV H,L	90	SUB B	BB	CMP E	E6	ANI D8
10	—	3B	DCX SP	66	MOV H,M	91	SUB C	BC	CMP H	E7	RST 4
11	LXI D,D16	3C	INR A	67	MOV H,A	92	SUB D	BD	CMP L	E8	RPE
12	STAX D	3D	DCR A	68	MOV L,B	93	SUB E	BE	CMP M	E9	PCHL
13	INX D	3E	MVI A,D8	69	MOV L,C	94	SUB H	BF	CMP A	EA	JPE Adr
14	INR D	3F	CMC	6A	MOV L,D	95	SUB L	C0	RNZ	EB	XCHG
15	DCR D	40	MOV B,B	6B	MOV L,E	96	SUB M	C1	POP B	EC	CPE Adr
16	MVI D,D8	41	MOV B,C	6C	MOV L,H	97	SUB A	C2	JNZ Adr	ED	—
17	RAL	42	MOV B,D	6D	MOV L,L	98	SBB B	C3	JMP Adr	EE	XRI D8
18	—	43	MOV B,E	6E	MOV L,M	99	SBB C	C4	CNZ Adr	EF	RST 5
19	DAD D	44	MOV B,H	6F	MOV L,A	9A	SBB D	C5	PUSH B	F0	RP
1A	LDAX D	45	MOV B,L	70	MOV M,B	9B	SBB E	C6	ADI D8	F1	POP PSW
1B	DCX D	46	MOV B,M	71	MOV M,C	9C	SBB H	C7	RST 0	F2	JP Adr
1C	INR E	47	MOV B,A	72	MOV M,D	9D	SBB L	C8	RZ	F3	DI
1D	DCR E	48	MOV C,B	73	MOV M,E	9E	SBB M	C9	RET Adr	F4	CP Adr
1E	MVI E,D8	49	MOV C,C	74	MOV M,H	9F	SBB A	CA	JZ	F5	PUSH PSW
1F	RAR	4A	MOV C,D	75	MOV M,L	A0	ANA B	CB	—	F6	ORI D8
20	RIM	4B	MOV C,E	76	HLT	A1	ANA C	CC	CZ Adr	F7	RST 6
21	LXI H,D16	4C	MOV C,H	77	MOV M,A	A2	ANA D	CD	CALL Adr	F8	RM
22	SHLD Adr	4D	MOV C,L	78	MOV A,B	A3	ANA E	CE	ACI D8	F9	SPHL
23	INX H	4E	MOV C,M	79	MOV A,C	A4	ANA H	CF	RST 1	FA	JM Adr
24	INR H	4F	MOV C,A	7A	MOV A,D	A5	ANA L	D0	RNC	FB	EI
25	DCR H	50	MOV D,B	7B	MOV A,E	A6	ANA M	D1	POP D	FC	CM Adr
26	MVI H,D8	51	MOV D,C	7C	MOV A,H	A7	ANA A	D2	JNC Adr	FD	—
27	DAA	52	MOV D,D	7D	MOV A,L	A8	XRA B	D3	OUT D8	FE	CPI D8
28	—	53	MOV D,E	7E	MOV A,M	A9	XRA C	D4	CNC Adr	FF	RST 7
29	DAD H	54	MOV D,H	7F	MOV A,A	AA	XRA D	D5	PUSH D		
2A	LHLD Adr	55	MOV D,L	80	ADD B	AB	XRA E	D6	SUI D8		

D8 = constant, or logical/arithmetic expression that evaluates to an 8-bit data quantity.

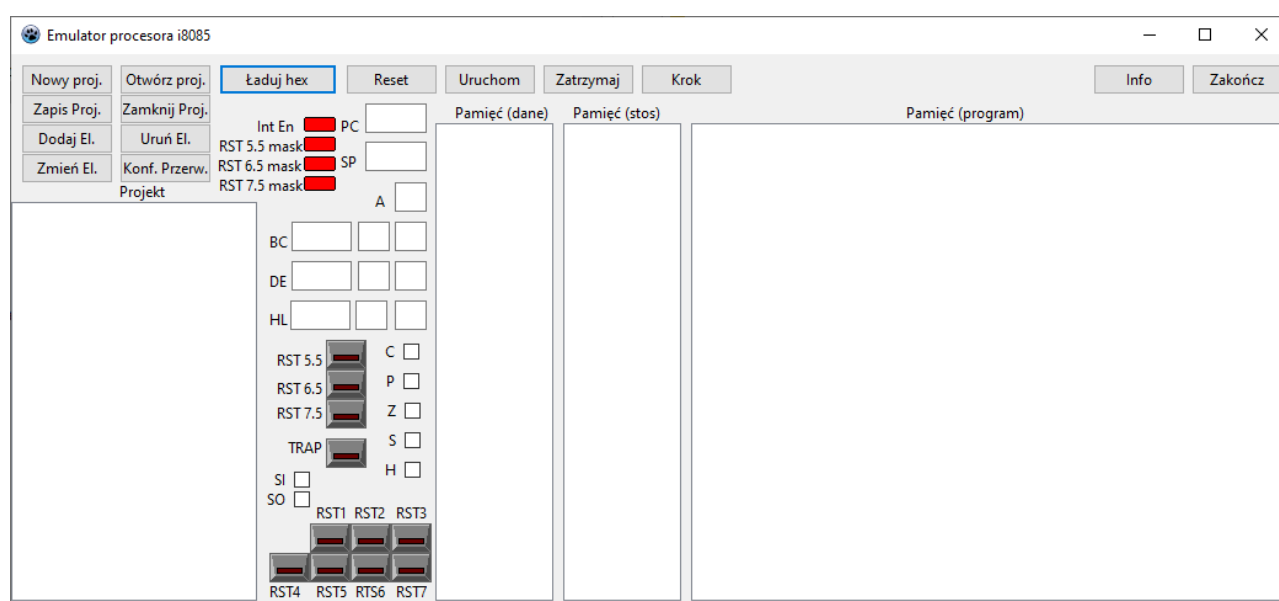
Adr = 16-bit address.

D16 = constant, or logical/arithmetic expression that evaluates to a 16-bit data quantity.

Powstał mały dylemat: kto mówi prawdę? Tu nie wahałem się zbyt i zaufałem oryginalnym publikacjom. Przykładowo kod CB hex będący dla ziloga prefiksem

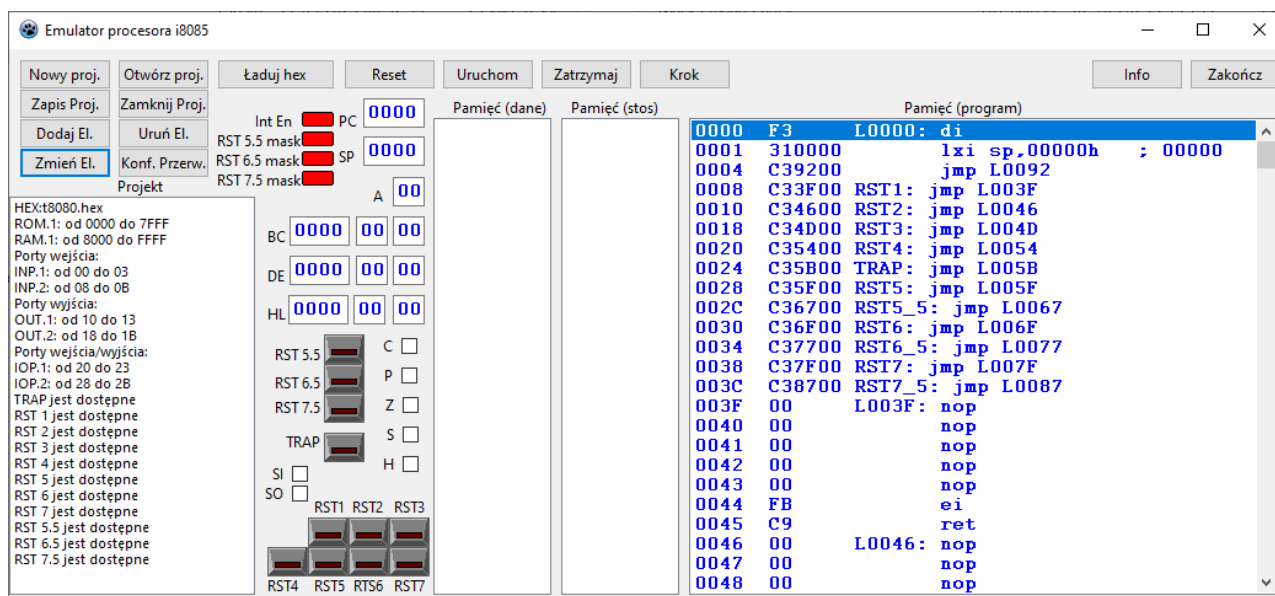
rozszerzenia do operacji bitowych, czy ED hex jako rozszerzenie do wielu ciekawych i przydatnych instrukcji czy DD hex i FD hex jako prefiksy do instrukcji operujących na rejestrach indeksowych, których w prockach intelowych nie ma. Te kody (i kilka innych) to są dziury dla procka i8085. Swoją drogą ciekawe co robi procek i8085, jak go się przymusi do wykonania tych instrukcji. To może być ciekawy temat badawczy, bo jak coś takiego zbadać? Procek może to olać, zawisnąć a może zrobić coś innego i tajemniczego. No może to kiedyś sprawdzę.

No więc stworzyłem sobie program do śledzenia wykonania programu dla procka i8085. Napisałem go sobie w pascalu.



Moim celem jest stworzenie narzędzia pozwalającego na diagnostykę w szerokim zakresie. Wymaga to określenia dla programu wielu informacji, gdyż program wyłapuje przykładowo zapis do pamięci EPROM, odczyt/zapis z pamięci nieistniejącej (gdyż nie ma obowiązku, by do procka była przypięta pamięć wypełniająca całą przestrzeń adresową). Podobnie są monitorowane użycia portów we/wy. Należy wcześniej określić jakie adresy portów są używane i symulacja wykonania programu dla procka i8085 się zatrzyma, jeżeli nie zostaną wyspecyfikowane „legalne” adresy portów. By nie wklepywać tego za każdym razem, jest plik opisujący projekt (symulacji). Oczywiście, wszystkie szczegóły można w dowolnej chwili zmodyfikować.

Przykładowy projekt: w okienku projektu wyświetlone są wszystkie szczegóły dotyczące środowiska, w którym jest uruchamiany program.



System prockowy składa się z pamięci EPROM (lub innej nieulotnej) lokowanej w przestrzeni od 0000 hex do 7FFF hex oraz pamięć RAM lokowana od 8000 hex do FFFF hex. W sumie obie pamięci wypełniają całą przestrzeń. Dodatkowo występują porty wejścia, porty wyjścia (jak przykładowo i8212) czy porty wyjścia/wejścia (jak przykładowo i8255). Dla jednych dostępne są jedynie instrukcje IN, dla innych jedynie OUT lub dla tych trzech zarówno IN jak i OUT. Również program emulatora wspiera obsługę przerw. Może to być kontroler zbudowany na bazie i8214, który generuje instrukcje RST 1 ... RST 7 lub przerwania połówkowe, które są zintegrowane w samym procku. No i oczywiście przerwanie niemaskowalne TRAP. Te przerwania (niemaskowalne oraz połówkowe) mogą być nieużywane w systemie (ze stałym sygnałem wymuszonym przez odpowiednie przyłączenie wejścia do masy lub VCC), toteż istnieje możliwość, że klikanie na przycisk do zgłaszania przerwania pozostanie bez reakcji.

Deasemblacja

Dysponując programem binarnym (w zapisie intel-hex), fajnie by było, by na ekranie było widać co się dzieje: co program robi, gdzie jest. Pierwszą czynnością po wczytaniu kodu hex, jest jego deasemblacja. Generalnie taka operacja nie jest jednoznaczna, gdyż trudno jest wyrokować o sensie zawartości przykładowo pary rejestrów HL. To może być adres skoku jak w instrukcji PCHL [w wersji zilogowej JP (HL)], to może być adres operandu w instrukcji arytmetycznej lub wręcz może pełnić rolę akumulatora w arytmetyce 16-bitowej. Ładując program intelowski do pamięci program zaczyna proces deasemblacji. Początek jest wiadomo w którym miejscu: adres 0. I tak instrukcja po instrukcji tłumaczony jest kod na prawdopodobną wersję źródłową. Napisałem prawdopodobną, gdyż nie ma gwarancji, że ta czynność będzie zgodna z intencjami autora, a te są tu zawsze nadrzędne. Algorytm deasemblacji jest rekurencyjny (to najprostsza implementacja). I posiłkuje się stałą tablicą opisującą kody instrukcji. Jest wykorzystywana do symulacji działania oraz do deasemblacji.

Jeżeli w projekcie są wskazania na obsługę przerwań, to inicjowana jest deasemblacja na stałe adresy związane z obsługą tych przerwań. Ponieważ nie ma obowiązku, by program obsługiwał przerwania, gdyż to może być trenowany jakiś fragment programu, to nie ma potrzeby deasemblować punktów wejścia do obsługi przerwań. To nawet może okazać się szkodliwe, gdyż istnieje szansa powstania konfliktów: obsługa przerwania może trafić w środek wielobajtowej instrukcji i wtedy to powstaje już kaszana.

Ten algorytm „przechodzi” każde rozgałęzienie programu intelowego, więc to co zostało „nie tykane” może stanowić obszar danych stałych. Program konwertuje te obszary do postaci *define byte*. Oczywiście w tym wszystkim jest jeden wyjątek: instrukcja PCHL, które realizuje skok, tylko nie wiadomo dokąd (na etapie deaseblacji). W ten oto sposób mogą być „wykryte” fragmenty martwego kodu, który nigdy nie ma szans na wykonanie. Przykładowo (fragment mojego generatora losowego):

Improved compiler for INTEL 8080/8085 processor, version 2.0, licence: CC BY 3.0

L.NO MEMO CODE SRC

Open file: D:\lazarus_project\I8080\T8080.ASM

```
1. ;
2. .i8085
3. 0000 ResetEntry .equ 0000h
4. .org ResetEntry
5. 0000 F3 di
6. 0001 310000 lxi sp,0
7. 0000 Stack_top .equ 0h
8. 0004 F3 di ;
9. 0005 310000 lxi sp , Stack_top ;
10. ;
11. 0008 CD1F00 call Init_random ;
12. 000B CD2C00 call CreateNewRandom ;
13. ;
14. 000E CD5900 call Init_variable ;
15. 0011 CD5A00 call Init_hardware ;
```

```

16. 0014 CD5B00          call    Init_envir          ;
17.                      ;.
18. 0017                Stop_prg                    ;
19. 0017 C31700          jmp     Stop_prg            ;
20.                      ;*****
21. 001A                Martwy_kod
22. 001A 00              nop                        ;
23. 001B 00              nop                        ;
24. 001C 00              nop                        ;
25. 001D 00              nop                        ;
26. 001E 00              nop                        ;
27.                      ;*****
28. 001F                Init_random                  ;
29. 001F 210080          lxi     h , RandomSeed      ;
30. 0022 110280          lxi     d , RandomV         ;
31. 0025 7E              mov     a , m              ;
32. 0026 12              stax    d                  ;
33. 0027 23              inx     h                  ;
34. 0028 13              inx     d                  ;
35. 0029 7E              mov     a , m              ;
36. 002A 12              stax    d                  ;
37. 002B C9              ret                        ;
38.                      ;*****
39.                      ; RandomV = ( ( MultConst * RandomV ) + AddConst ) ;
40.                      ; MultConst = 129, AddConst = 9
41. 002C                CreateNewRandom
42.                      ;function CreateNewRandom ( ) : word [HL]
43. 002C 110280          lxi     d , RandomV         ;
44. 002F 1A              ldax    d                  ;
45. 0030 6F              mov     l , a              ;
46. 0031 13              inx     d                  ;
47. 0032 1A              ldax    d                  ;
48. 0033 67              mov     h , a              ;
49. 0034 E5              push    h                  ;
50. 0035 C1              pop     b                  ;
51. 0036 CD5100          call    ShiftLBC            ;
52. 0039 CD5100          call    ShiftLBC            ;
53. 003C CD5100          call    ShiftLBC            ;
54. 003F CD5100          call    ShiftLBC            ;
55. 0042 CD5100          call    ShiftLBC            ;
56. 0045 CD5100          call    ShiftLBC            ;
57. 0048 CD5100          call    ShiftLBC            ;
58. 004B 09              dad     b                  ;
59. 004C 010900          lxi     b , 09h            ;
60. 004F 09              dad     b                  ;
61. 0050 C9              ret                        ;
62.                      ;*****
63. 0051                ShiftLBC                      ;
64.                      ;function ShiftLBC ( Arg [BC] : word ) : word [BC]
65. 0051 79              mov     a , c              ;
66. 0052 B7              ora     a                  ;
67. 0053 17              ral     ;                  ;
68. 0054 4F              mov     c , a              ;
69. 0055 78              mov     a , b              ;
70. 0056 17              ral     ;                  ;
71. 0057 47              mov     b , a              ;
72. 0058 C9              ret                        ;
73.                      ;*****
74. 0059                Init_variable                  ;
75. 0059 C9              ret                        ;
76.                      ;*****
77. 005A                Init_hardware                  ;
78. 005A C9              ret                        ;
79.                      ;*****
80. 005B                Init_envir                      ;
81. 005B C9              ret                        ;
82.                      .org      08000h            ;
83. 8000                RandomSeed      .word      1 ;

```



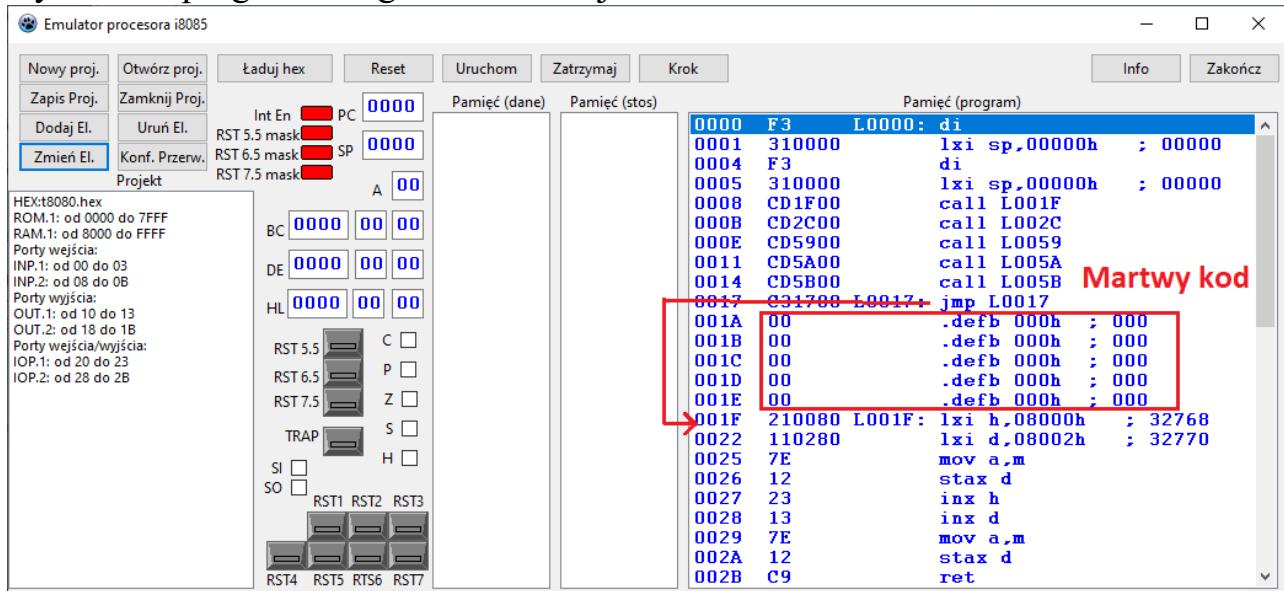
```

84. 8002          RandomV          .word    1          ;
85.              .end              ;
Close file: D:\lazarus_project\I8080\T8080.ASM

```

Compilation :successful

Zawiera fragment kodu, do którego nie prowadzi żadna droga w trakcie wykonania programu. Jego deasemblacja to:



Jednak czasami jest to informacja nie do końca prawdziwa. Przykładowo:

Improved compiler for INTEL 8080/8085 processor, version 2.0, licence: CC BY 3.0

L:NO MEMO CODE SRC

Open file: D:\lazarus_project\I8080\T8080.ASM

```

1.          ;
2.          .i8085
3. 0000      ResetEntry      .equ 0000h
4. 007B      Data8           .equ 123
5.          .org ResetEntry
6. 0000 F3          di
7. 0001 310000      lxi sp,0
8. 0004 210A00      lxi h,subr
9. 0007 E9          pchl
10. 0008 00         nop ; to jest martwy kod
11. 0009 00         nop ; to jest martwy kod
12. 000A          subr:
13. 000A CE7B      ACI Data8
14. 000C 8F        ADC A
15. 000D 88        ADC B
16. 000E 89        ADC C
17. 000F 8A        ADC D
18. 0010 8B        ADC E
19. 0011 8C        ADC H
20. 0012 8D        ADC L
21. 0013 EB        XCHG
22. 0014 8E        ADC M
23. 0015 87        ADD A
24. 0016 80        ADD B
25. 0017 81        ADD C
26. 0018 82        ADD D
27. 0019 83        ADD E
28. 001A 84        ADD H

```

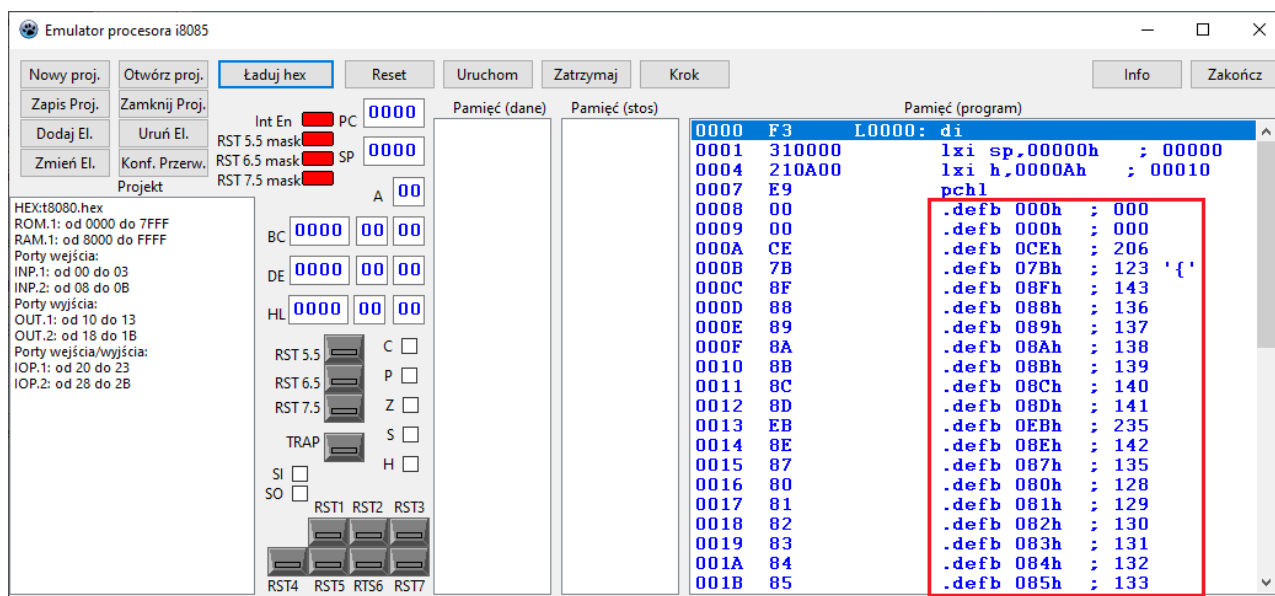
```

29. 001B 85      ADD L
30. 001C 86      ADD M
31. 001D C67B    ADI Data8
32. 001F A7      ANA A
33. 0020 A0      ANA B
34. 0021 A1      ANA C
35. 0022 A2      ANA D
36. 0023 A3      ANA E
37. 0024 A4      ANA H
38. 0025 A5      ANA L
39. 0026 A6      ANA M
40. 0027 E67B    ANI Data8
41. 0029 2F      CMA
42. 002A 3F      CMC
43. 002B BF      CMP A
44. 002C B8      CMP B
45. 002D B9      CMP C
46. 002E BA      CMP D
47. 002F BB      CMP E
48. 0030 BC      CMP H
49. 0031 BD      CMP L
50. 0032 BE      CMP M
51. 0033 C33300  Loop JMP Loop
52.              .end
Close file: D:\lazarus_project\I8080\T8080.ASM

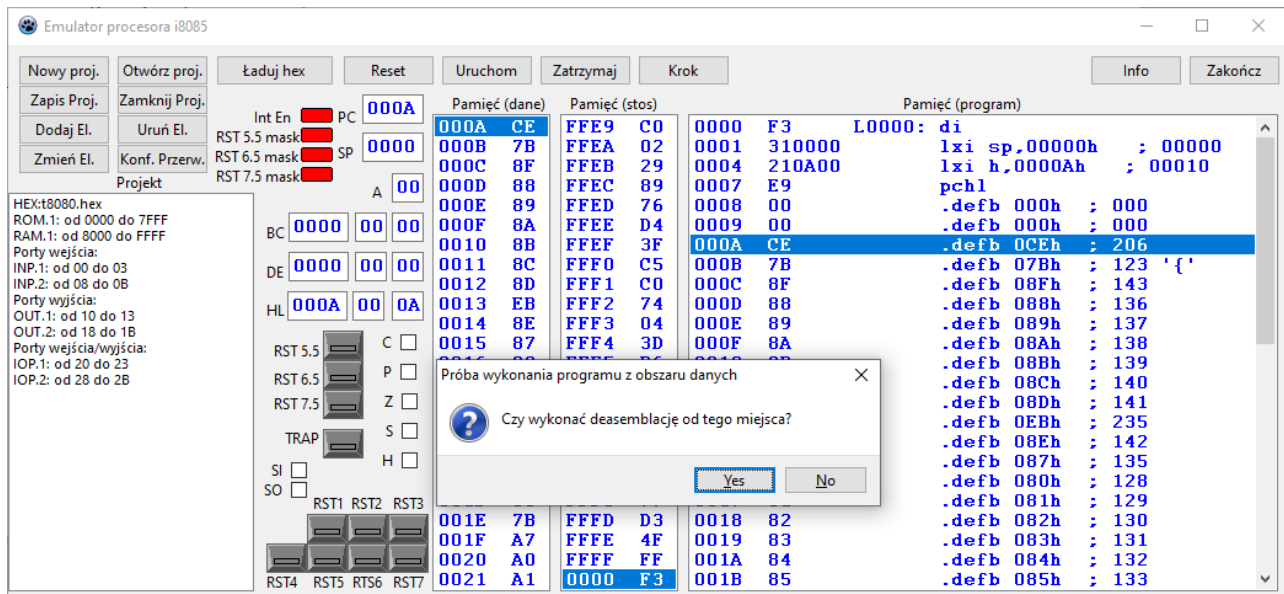
```

Compilation :successful

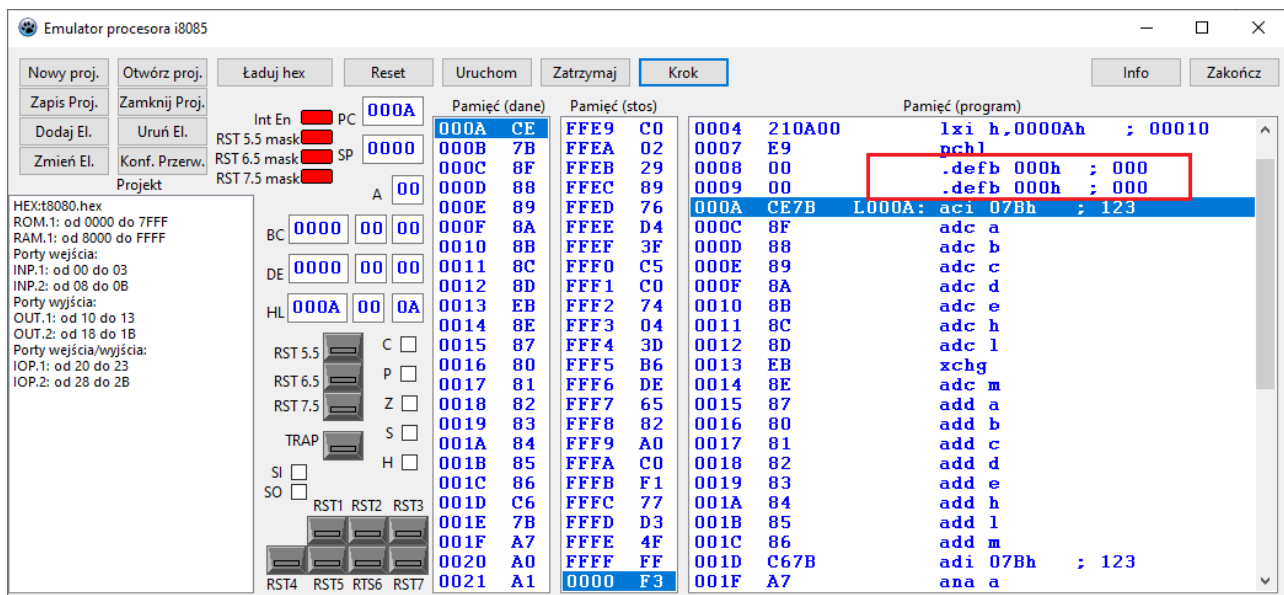
Daje następujący wynik:



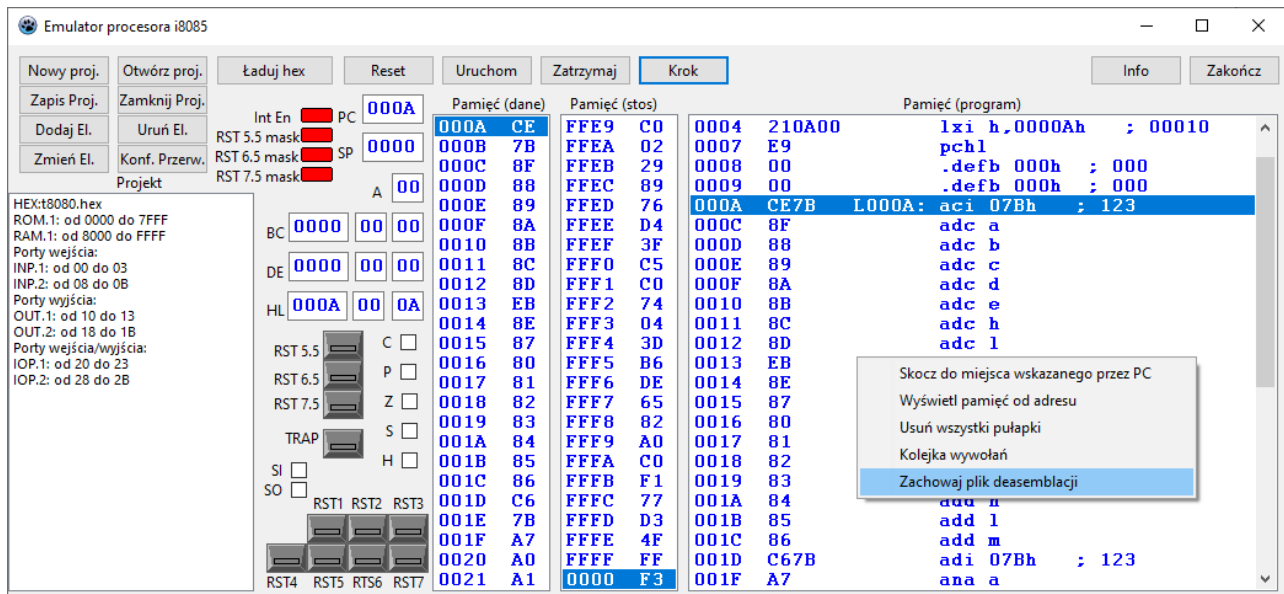
Do zaznaczonego fragmentu nie prowadzi żadna jawna ścieżka wywołania a prawdziwa rzeczywistość wyjaśni się dopiero w trakcie symulacji wykonania programu. Zaistnienie zdarzenia, gdzie następuje próba wykonania czegoś, co zostało zakwalifikowane jako blok stałych danych, generuje zatrzymanie i pytanie, czy „mamy ochotę to naprawić”.



Istnieje taka możliwość i realizowana jest deasemblacja „w biegu”. Fragment nadal pozostaje oznaczony jako martwy kod (tym razem już prawdziwy).



Istnieje możliwość zapisania do pliku wyniku deasemblacji po kliknięciu prawym klawiszem myszki.

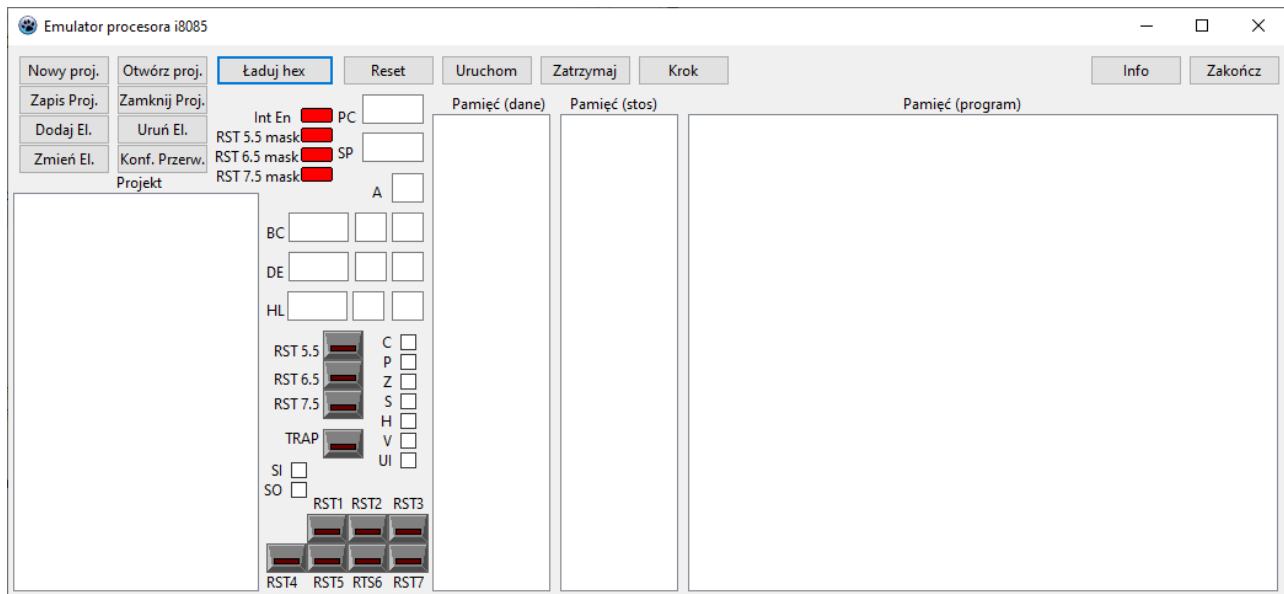


Powstała postać „źródłowa” jest następująca:

```
; *****
; ** Deasemblacja kodu procesora i8085 **
; ** przez program emulatora I8085 **
; ** Licencja: CC BY 3.0 **
; *****
.org 0000h
L0000: di
      lxi sp,0000h    ; 0000h
      lxi h,000Ah     ; 0001h
      pchl
      .defb 000h      ; 000
      .defb 000h      ; 000
L000A: aci 07Bh       ; 123
      adc a
      adc b
      adc c
      adc d
      adc e
      adc h
      adc l
      xchg
      adc m
      add a
      add b
      add c
      add d
      add e
      add h
      add l
      add m
      adi 07Bh        ; 123
      ana a
      ana b
      ana c
      ana d
      ana e
      ana h
```

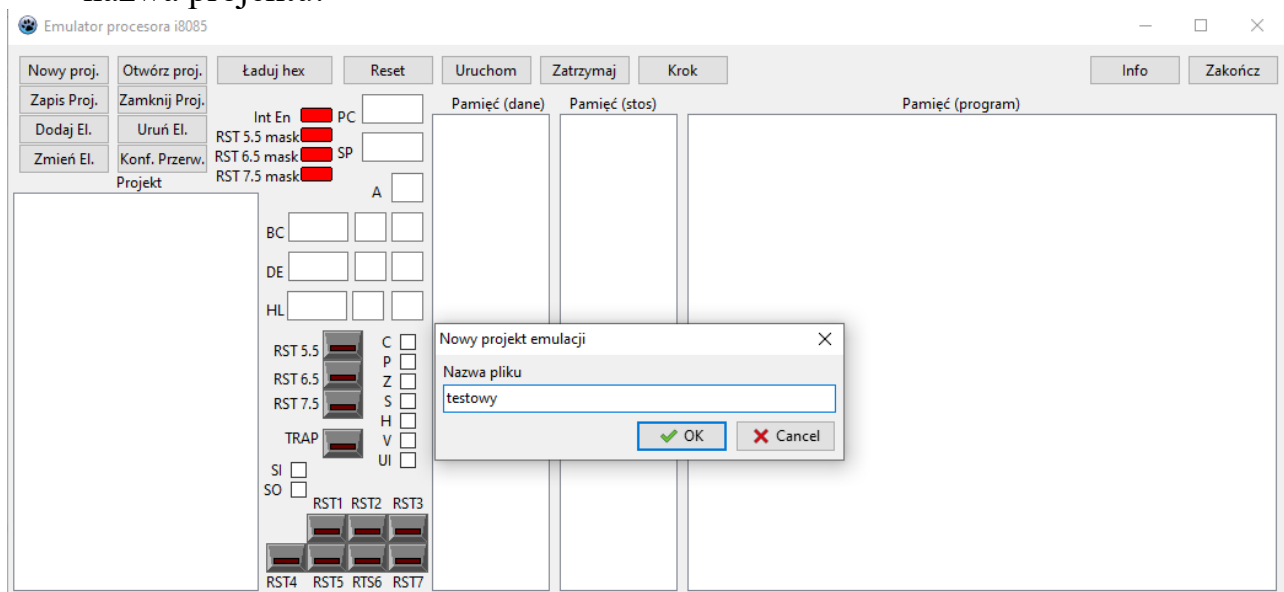


```
ana l
ana m
ani 07Bh ; 123
cma
cmc
cmp a
cmp b
cmp c
cmp d
cmp e
cmp h
cmp l
cmp m
L0033: jmp L0033
.end
```

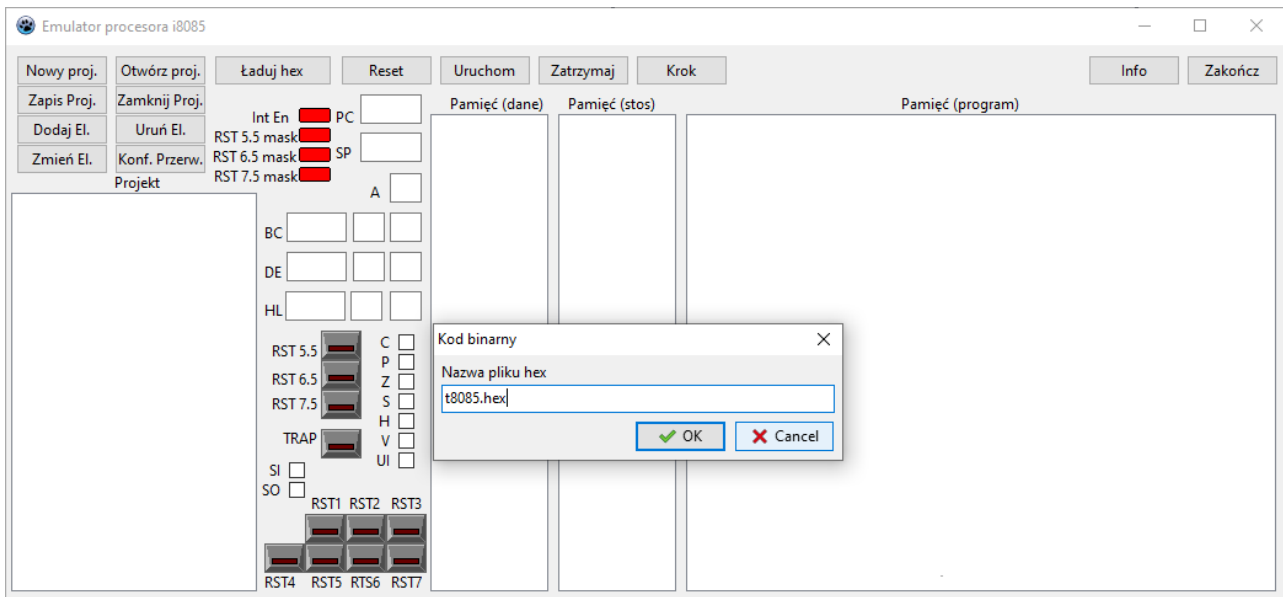


Po uruchomieniu programu emulatora, mamy możliwość otwarcia istniejącego projektu lub utworzenie nowego. Po kliknięciu na „Nowy proj.” program poprosi o wszystkie niezbędne informacje. Należą do nich:

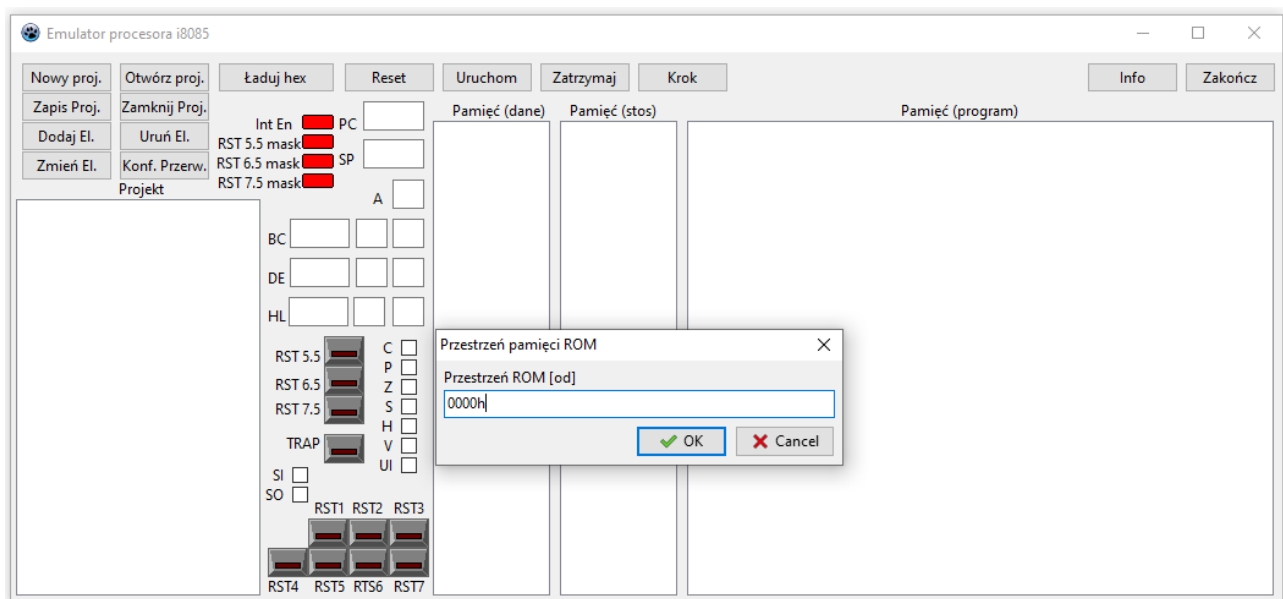
* nazwa projektu:

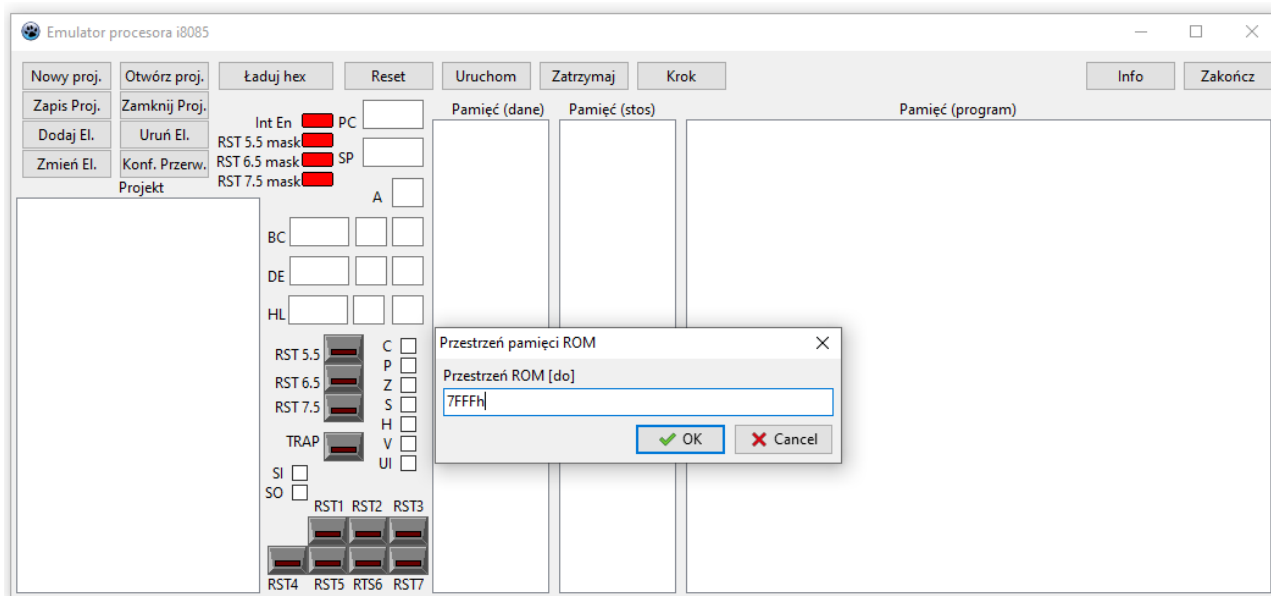


* nazwa pliku z kodem binarnym programu:

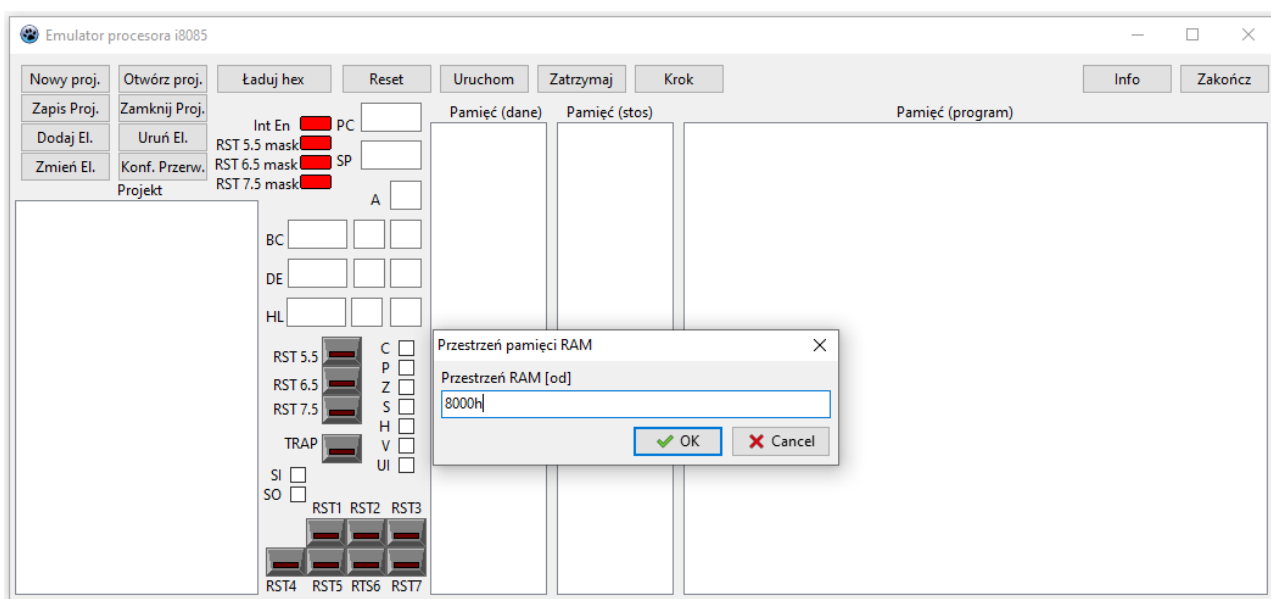


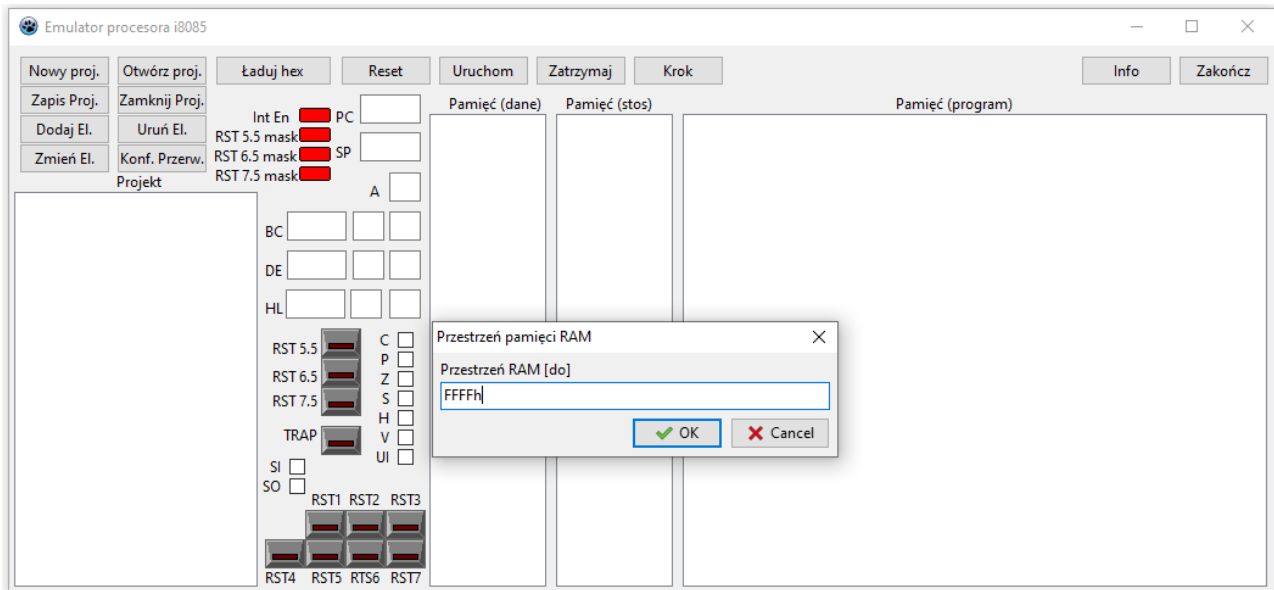
* określenie przestrzeni adresowej pamięci ROM (na program) – od .. do



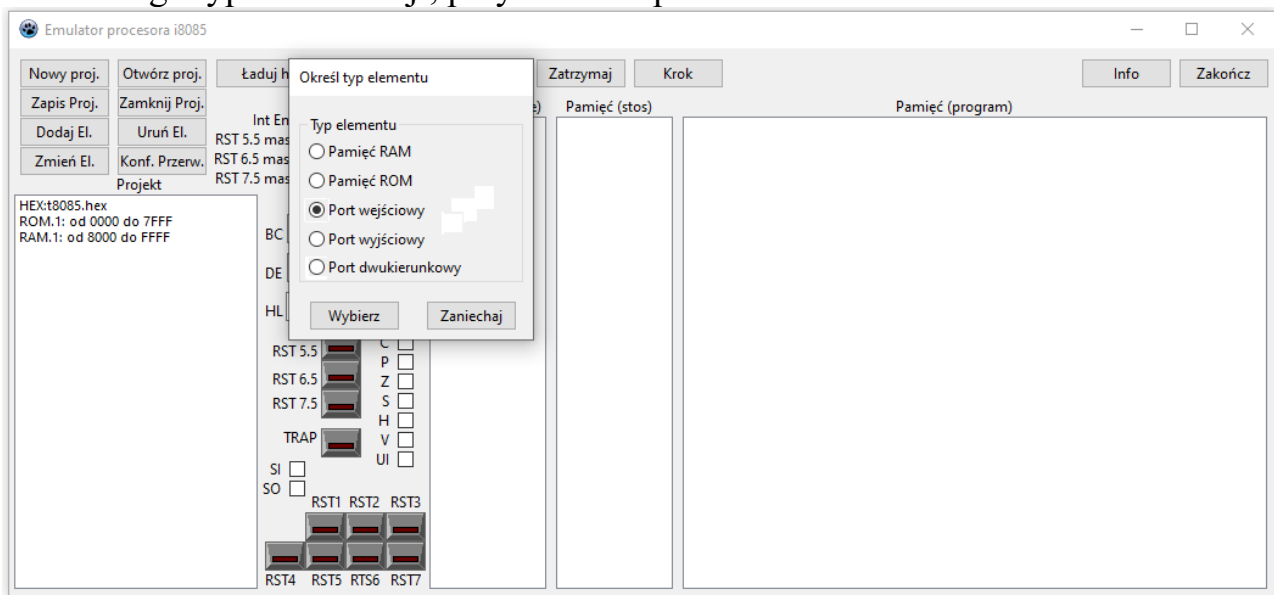


* określenie przestrzeni adresowej pamięci RAM – od .. do

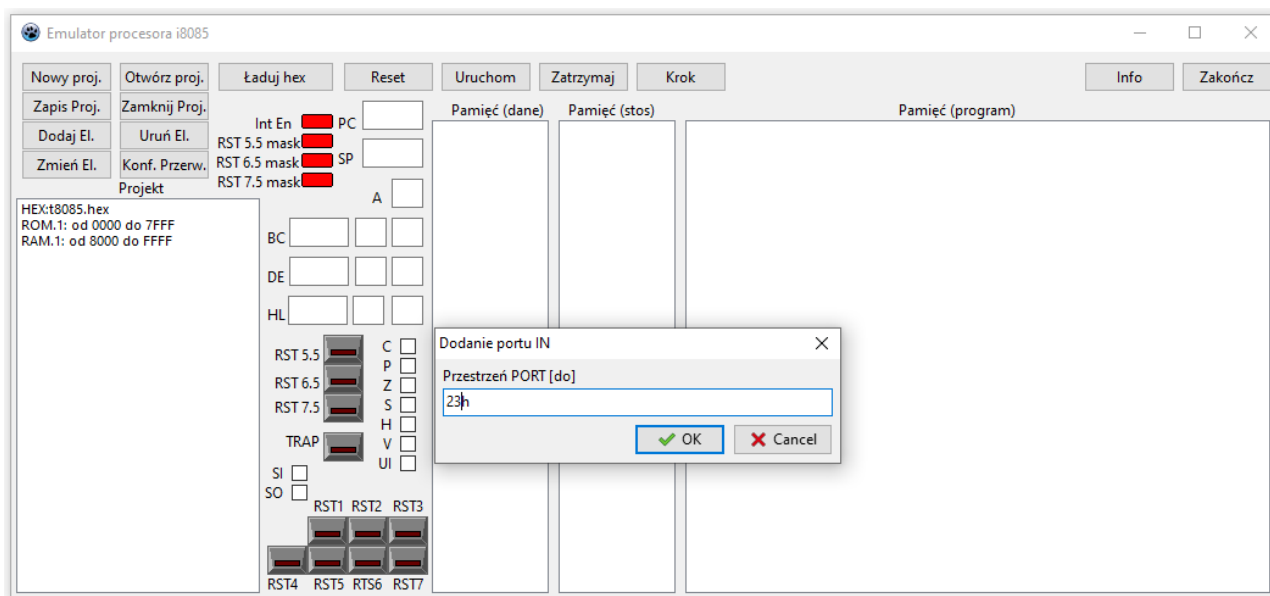
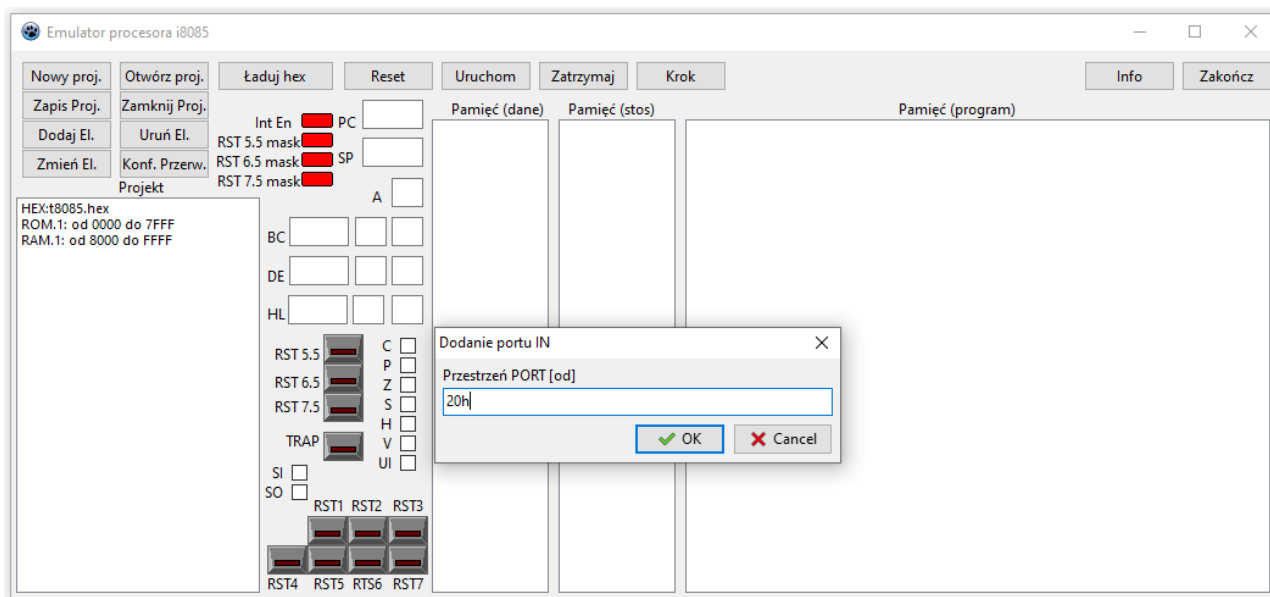




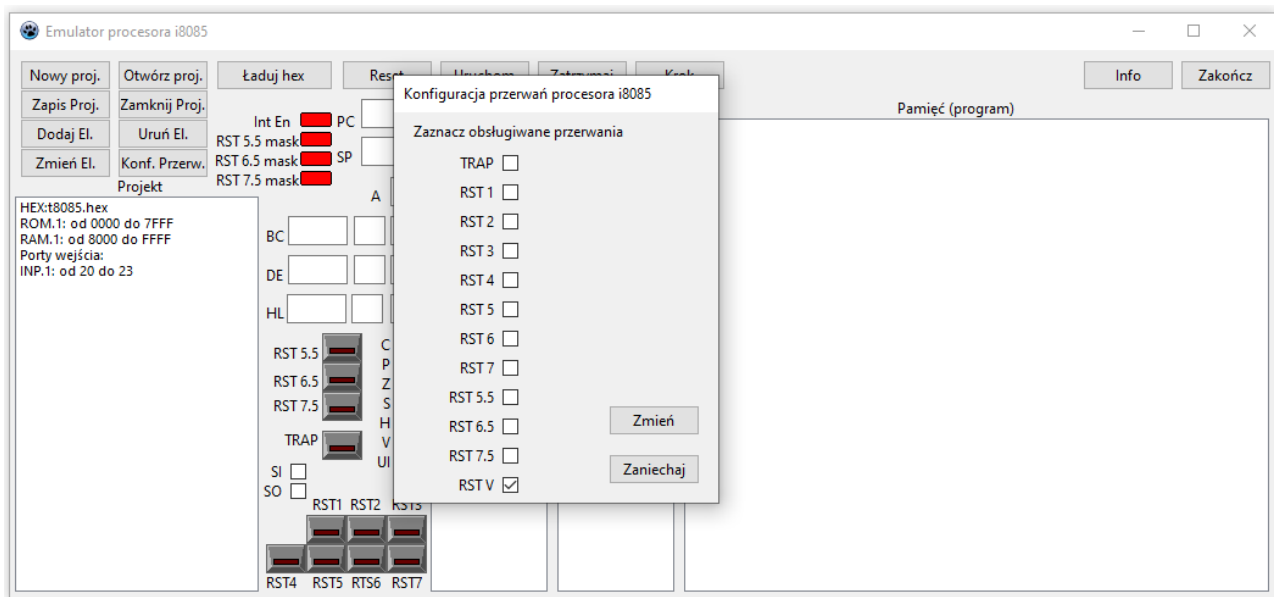
Daje to minimalną konfigurację środowiska. Dodatkowo można dodać zestaw portów we, wy lub we/wy klikając na „Dodaj el.”, gdzie istnieje możliwość dodania określonego typu informacji, przykładowo portów we:



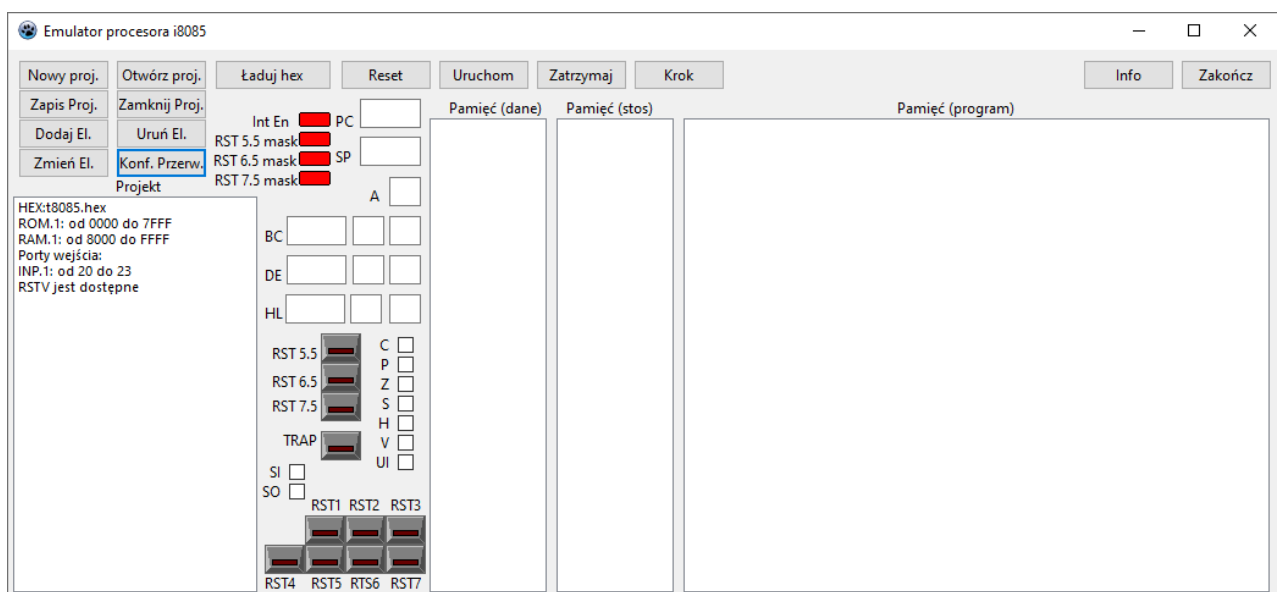
i w dalszej kolejności zakresu adresowego portów:



Przycisk „Konf. Przerw.” służy do określenia przerw, jakie są obsługiwane w programie. Wśród znanych dotychczas przerw pojawia się nowe. Program został rozbudowany o kolejną pozycję: przerwania od nadmiaru arytmetycznego: RSTV.

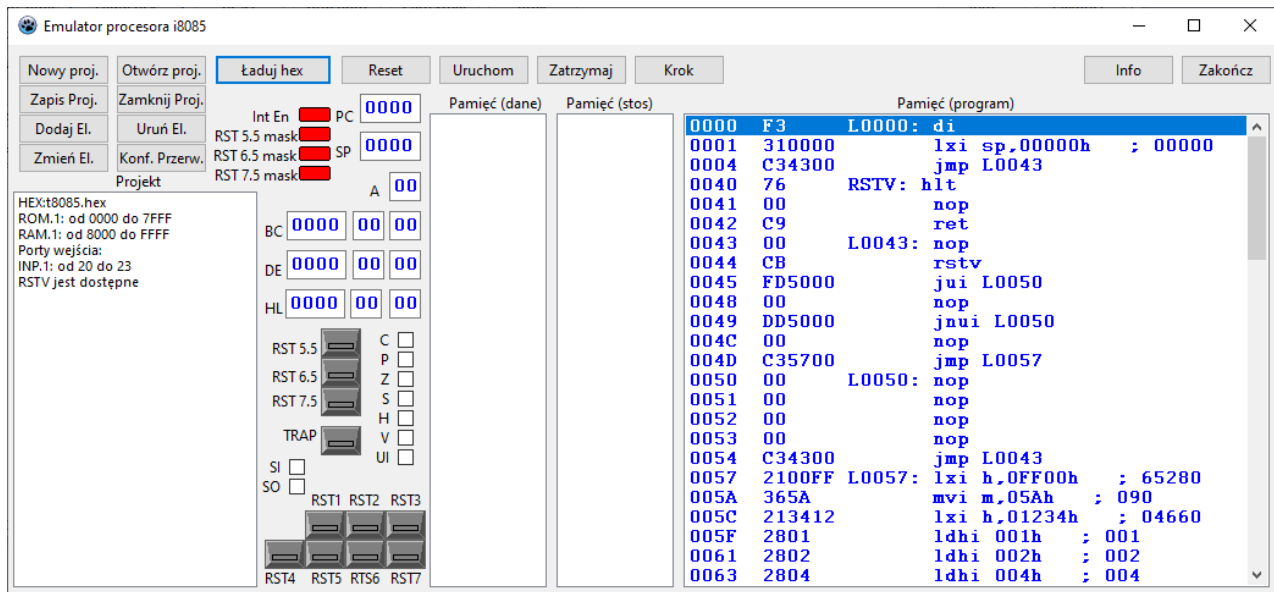


Wszystkie wprowadzone dane środowiskowe są wyświetlone w okienku:



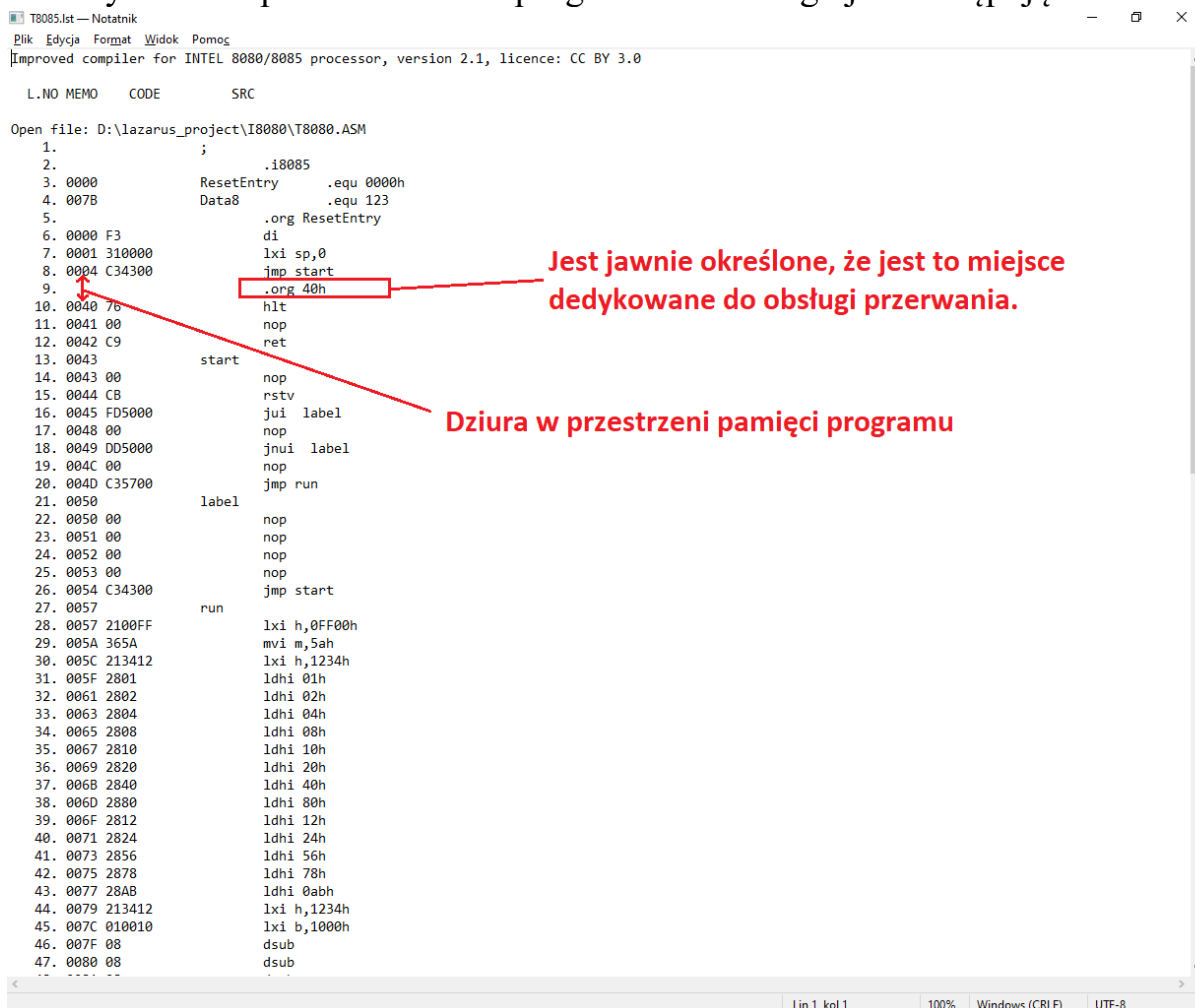
Można je zapisać do pliku i posłużyć się tym plikiem na przyszłość.

Teraz pozostało załadować kod programu (znana pliku jest już znana). Przy tej operacji sprawdzana jest zgodność wklepanych danych z załadowanym kodem. Głównie chodzi tu o obsługę przerwań, gdzie każde z nich ma swój indywidualny adres obsługi – w ładowanym kodzie na adresach specyfikowanych przerwań musi wystąpić kod programu.



Oczywiście program nie jest w stanie sprawdzić na tym etapie, czy rzeczywiście jest to kod obsługi przerwań. Program emulatora jest jedynie narzędziem wsparcia i nie zwalnia „od myślenia” użytkownika.

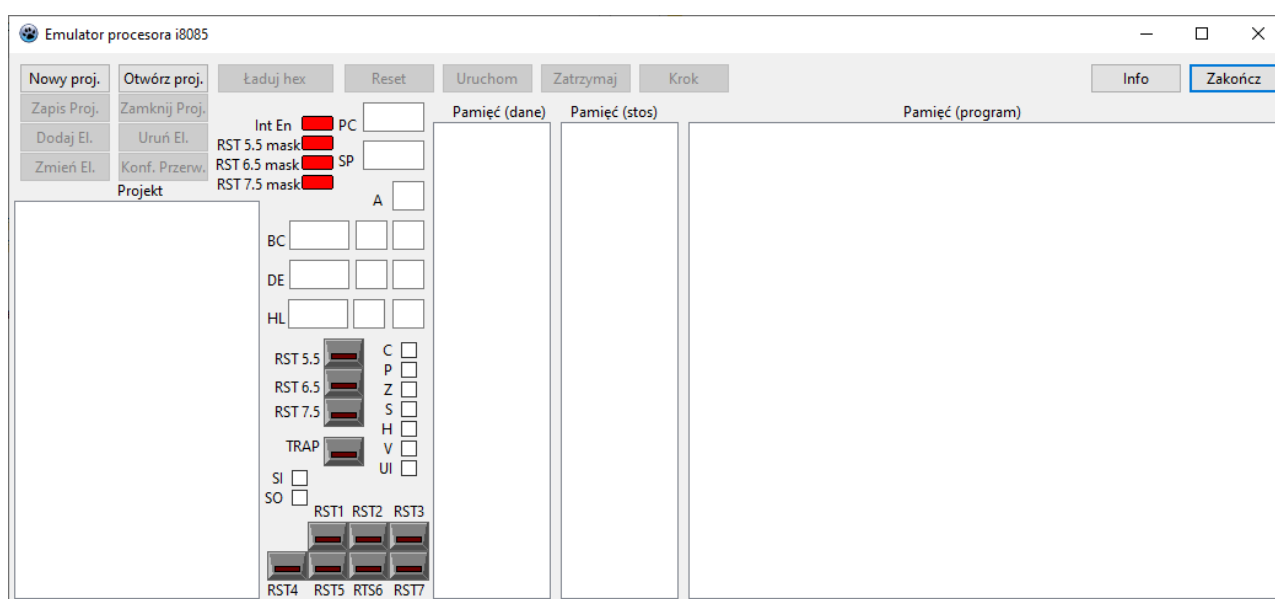
Przykładowa postać źródłowa programu intelowego jest następująca:



Emulator „zauważył”, że w kodzie binarnym programu w lokacji adresowej przewidzianej do obsługi przerwania coś jest.

Emulacja programu

Program ciągle się rozwija i ulega zmianom. Jednocześnie przechodzi testy. Ponieważ pewne działania wymagają określonej kolejności (w sensie klikania na przyciski, gdyż przykładowo trudno kliknąć na „wykonaj krok” jeżeli nie jest załadowany program), to jego działanie zostało rozbudowane o funkcjonalność „wskazującą na możliwy kolejny krok”. Obecnie po uruchomieniu programu mamy dostępne przyciski pozwalające na utworzenie lub otwarcie pliku projektu. Istnieje również możliwość upuszczenia nazwy pliku projektu na formę programu, co jest równoznaczne z otwarciem tego konkretnego pliku.



Ilustracja 1: em8085-61.png

Jako przykład będę posługiwał się następującym plikiem źródłowym (który należy skompilować by uzyskać program binarny). Zawiera on obsługę przerwań sprzętowych i jakiś tam zestaw instrukcji (które właściwie nie tworzą jakiegoś sensownego programu, na to przyjdzie czas później).

```

;
; *****
; *
; *   zbior instrukcji mikroprocesora 8080
; *
; *****
;
;
;   .i8080
;   .i8085
ResetEntry    .equ 0000h
RST1Entry     .equ 0008h
RST2Entry     .equ 0010h
RST3Entry     .equ 0018h
RST4Entry     .equ 0020h
TRAPEntry     .equ 0024h
RST5Entry     .equ 0028h
RST55Entry    .equ 002Ch
RST6Entry     .equ 0030h
RST65Entry    .equ 0034h
RST7Entry     .equ 0038h

```

```
RST75Entry      .equ 003Ch
;
Data8           .equ 56h
                .org ResetEntry
                di
                lxi sp,0
                jmp StartProg
                .org RST1Entry
                jmp RST1Service
                .org RST2Entry
                jmp RST2Service
                .org RST3Entry
                jmp RST3Service
                .org RST4Entry
                jmp RST4Service
                .org TRAPEntry
                jmp TRAPService
                .org RST5Entry
                jmp RST5Service
                .org RST55Entry
                jmp RST55Service
                .org RST6Entry
                jmp RST6Service
                .org RST65Entry
                jmp RST65Service
                .org RST7Entry
                jmp RST7Service
                .org RST75Entry
                jmp RST75Service
RST1Service
    nop
    nop
    ei
    ret
RST2Service
    nop
    nop
    ei
    ret
RST3Service
    nop
    nop
    ei
    ret
RST4Service
    nop
    nop
    ei
    ret
TRAPService
    jmp TRAPService
    ret
RST5Service
    nop
    nop
    ei
    ret
RST55Service
    nop
    nop
    ei
    ret
RST6Service
    nop
    nop
    ei
    ret
RST65Service
```

```

        nop
        nop
        ei
        ret
RST7Service
        nop
        nop
        ei
        ret
RST75Service
        nop
        nop
        mvi a,10h
        sim
        ei
        ret
StartProg
        di
        ei
loop
        call subr
        jmp loop
subr
        di
        ei
        in 123
        in 0
        in 1
        in 2
        in 3
        in 038h
        out 123
        out 0
        out 1
        out 2
        out 3
        out 1Ah
        lhld lhldop
        shld shldop
        lxi h,5aa5h
        xthl
        xthl
        lxi h,089abh
        rim
        mvi a,08h
        sim
        rim
        mov a,m
        inx h
        mov b,m
        inx h
        mov c,m
        inx h
        mov d,m
        inx h
        mov e,m
        xchg
        ACI Data8
        ADC A
        ADC B
        ADC C
        ADC D
        ADC E
        ADC H
        ADC L
        xchg
        ADC M
        ADD A

```

```

; CE56
; 8F
; 88
; 89
; 8A
; 8B
; 8C
; 8D
; 8E
; 87

```

ADD B	; 80
ADD C	; 81
ADD D	; 82
ADD E	; 83
ADD H	; 84
ADD L	; 85
ADD M	; 86
ADI Data8	; C656
ANA A	; A7
ANA B	; A0
ANA C	; A1
ANA D	; A2
ANA E	; A3
ANA H	; A4
ANA L	; A5
ANA M	; A6
ANI Data8	; E656
CMA	; 2F
CMC	; 3F
CMP A	; BF
CMP B	; B8
CMP C	; B9
CMP D	; BA
CMP E	; BB
CMP H	; BC
CMP L	; BD
CMP M	; BE
DAA	; 27
DAD B	; 09
DAD D	; 19
DAD H	; 29
DAD SP	; 39
DCR A	; 3D
DCR B	; 05
DCR C	; 0D
DCR D	; 15
DCR E	; 1D
DCR H	; 25
DCR L	; 2D
DCX B	; 0B
DCX D	; 1B
DCX H	; 2B
DCX SP	; 3B
DI	; F3
EI	; FB
IN Data8	; DB56
OUT Data8	; D356
INR A	; 3C
INR B	; 04
INR C	; 0C
INR D	; 14
INR E	; 1C
INR H	; 24
INR L	; 2C
INX B	; 03
INX D	; 13
INX H	; 23
INX SP	; 33
LDA Data16	; 3A3412
LDAX B	; 0A
LDAX D	; 1A
LHLD Data16	; 2A3412
LXI B,Data16	; 013412
LXI D,Data16	; 113412
LXI H,Data16	; 213412
MOV A,A	; 7F
MOV A,B	; 78
MOV A,C	; 79

MOV A,D	; 7A
MOV A,E	; 7B
MOV A,H	; 7C
MOV A,L	; 7D
MOV A,M	; 7E
MOV B,A	; 47
MOV B,B	; 40
MOV B,C	; 41
MOV B,D	; 42
MOV B,E	; 43
MOV B,H	; 44
MOV B,L	; 45
MOV B,M	; 46
MOV C,A	; 4F
MOV C,B	; 48
MOV C,C	; 49
MOV C,D	; 4A
MOV C,E	; 4B
MOV C,H	; 4C
MOV C,L	; 4D
MOV C,M	; 4E
MOV D,A	; 57
MOV D,B	; 50
MOV D,C	; 51
MOV D,D	; 52
MOV D,E	; 53
MOV D,H	; 54
MOV D,L	; 55
MOV D,M	; 56
MOV E,A	; 5F
MOV E,B	; 58
MOV E,C	; 59
MOV E,D	; 5A
MOV E,E	; 5B
MOV E,H	; 5C
MOV E,L	; 5D
MOV E,M	; 5E
MOV H,A	; 67
MOV H,B	; 60
MOV H,C	; 61
MOV H,D	; 62
MOV H,E	; 63
MOV H,H	; 64
MOV H,L	; 65
MOV H,M	; 66
MOV L,A	; 6F
MOV L,B	; 68
MOV L,C	; 69
MOV L,D	; 6A
MOV L,E	; 6B
MOV L,H	; 6C
MOV L,L	; 6D
MOV L,M	; 6E
MOV M,A	; 77
MOV M,B	; 70
MOV M,C	; 71
MOV M,D	; 72
MOV M,E	; 73
MOV M,H	; 74
MOV M,L	; 75
MVI A,Data8	; 3E56
MVI B,Data8	; 0656
MVI C,Data8	; 0E56
MVI D,Data8	; 1656
MVI E,Data8	; 1E56
MVI H,Data8	; 2656
MVI L,Data8	; 2E56
MVI M,Data8	; 3656

NOP	; 00
ORA A	; B7
ORA B	; B0
ORA C	; B1
ORA D	; B2
ORA E	; B3
ORA H	; B4
ORA L	; B5
ORA M	; B6
ORI Data8	; F656
PUSH B	; C5
PUSH D	; D5
PUSH H	; E5
PUSH PSW	; F5
POP PSW	; F1
POP H	; C1
POP D	; D1
POP B	; E1
RAL	; 17
RAR	; 1F
RC	; D8
RLC	; 07
RM	; F8
RNC	; D0
RNZ	; C0
RP	; F0
RPE	; E8
RPO	; E0
RRC	; 0F
RST 0	; C7
RST 1	; CF
RST 2	; D7
RST 3	; DF
RST 4	; E7
RST 5	; EF
RST 6	; F7
RST 7	; FF
RZ	; C8
SBB A	; 9F
SBB B	; 98
SBB C	; 99
SBB D	; 9A
SBB E	; 9B
SBB H	; 9C
SBB L	; 9D
SBB M	; 9E
SBI Data8	; DE56
SHLD Data16	; 223412
SPHL	; F9
STA Data16	; 323412
STAX B	; 02
STAX D	; 12
STC	; 37
SUB A	; 97
SUB B	; 90
SUB C	; 91
SUB D	; 92
SUB E	; 93
SUB H	; 94
SUB L	; 95
SUB M	; 96
SUI Data8	; D656
XCHG	; EB
XRA A	; AF
XRA B	; A8
XRA C	; A9
XRA D	; AA
XRA E	; AB

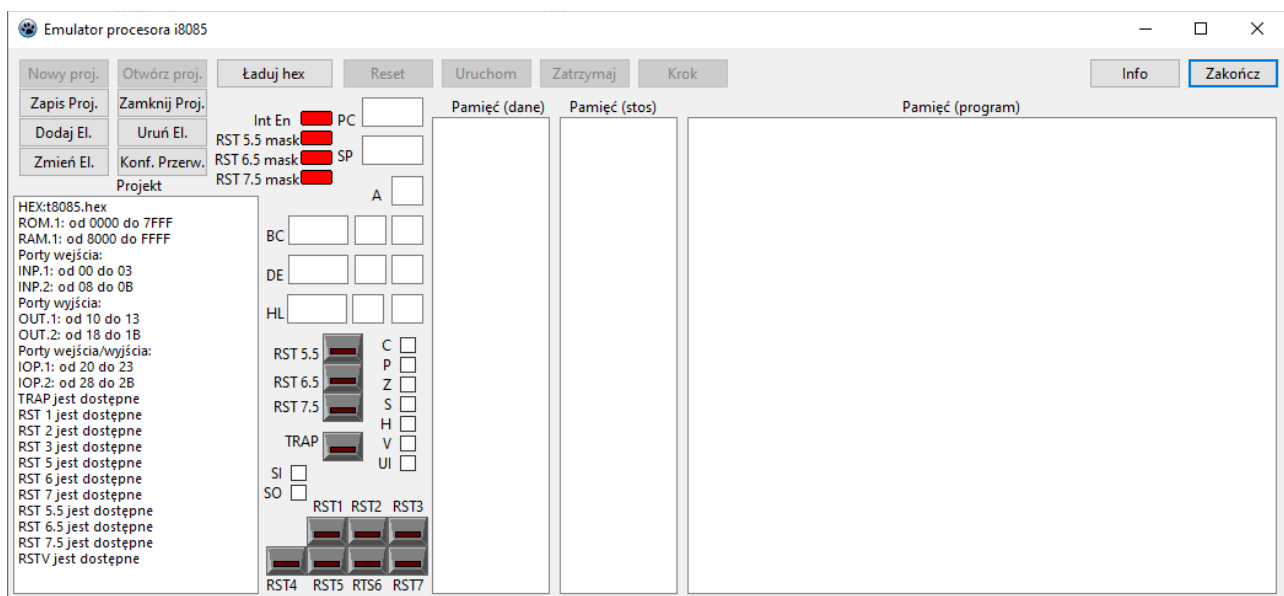

```

XRA H           ; AC
XRA L           ; AD
XRA M           ; AE
XRI Data8       ; EE56
XTHL           ; E3
RIM            ; 20
SIM            ; 30
RET            ; C9

lhldop .defw1234h
      .org 08000h
Data16 .word
shldop .word
.end

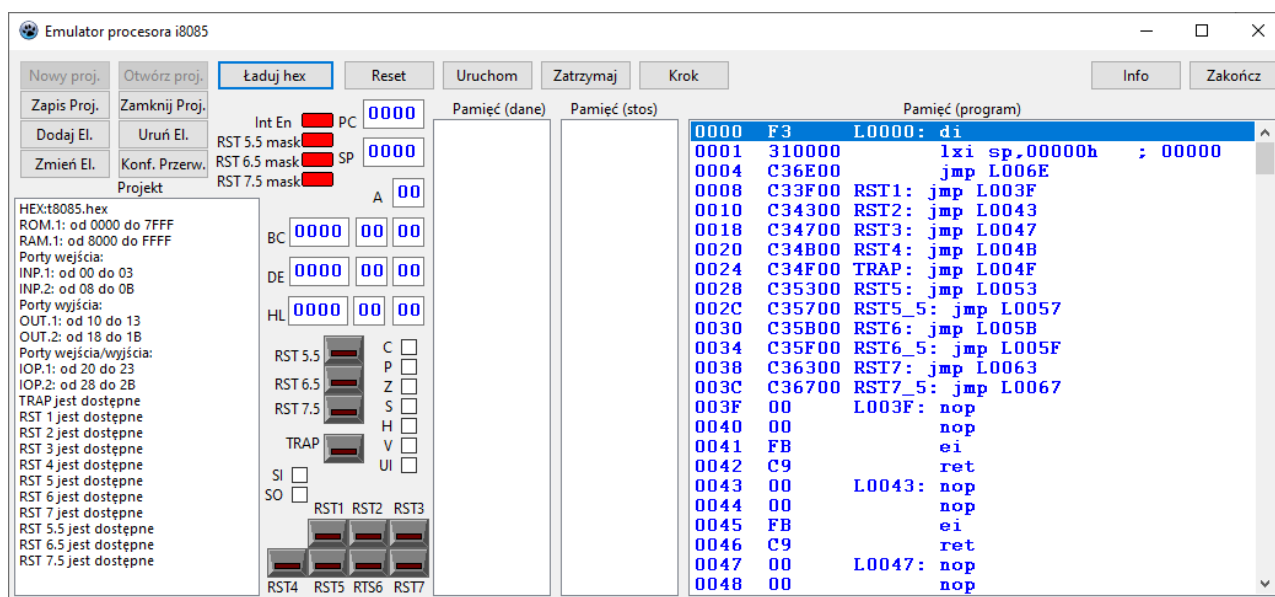
```

Po otwarciu pliku projektu istnieje możliwość jego modyfikacji lub załadowania kodu programu binarnego.



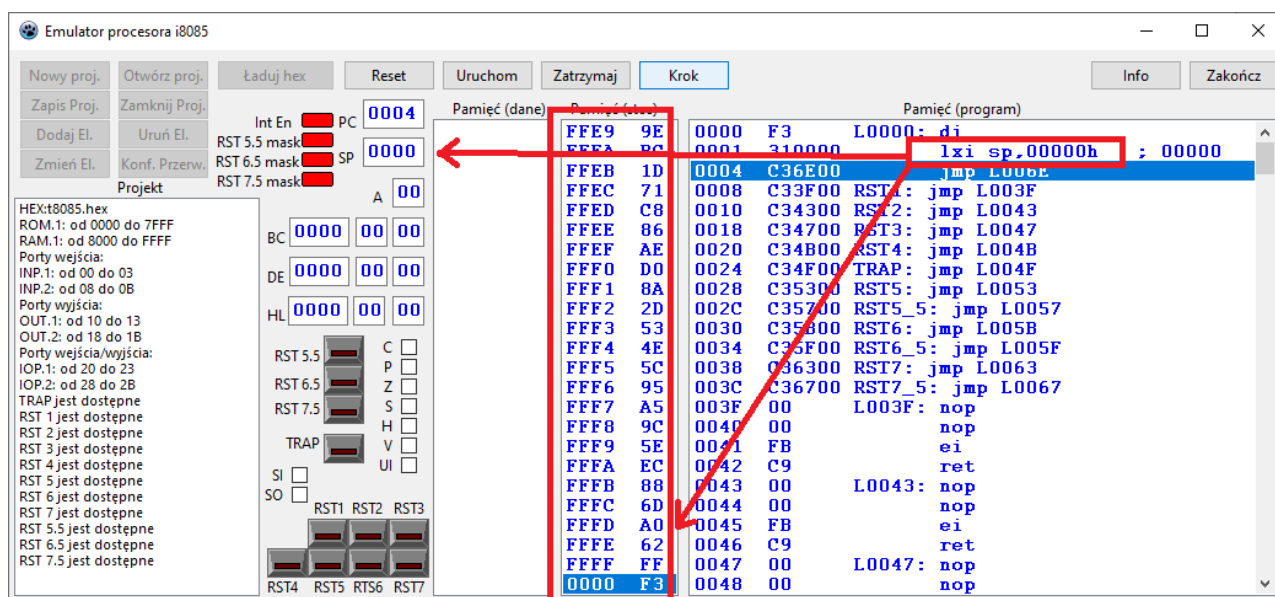
Ilustracja 2: em8085-62.png

Oczywiście zmienia się zestaw dostępnych operacji. Jeżeli projekt nie jest modyfikowany można załadować program binarny klikając na „Ładuj hex” (jego nazwa jest już wpisana z plik projektu). Jeżeli w trakcie symulacji zostaną dostrzeżone własne błędy, to można zmodyfikować program źródłowy (ten dla intel), skompilować i ponownie kliknąć na „Ładuj hex” (nie ma konieczności zamykania programu emulatora lub ponownego otwierania projektu). Po załadowaniu kodu, program jest deasemblowany i jest postać jest pokazana w okienku.



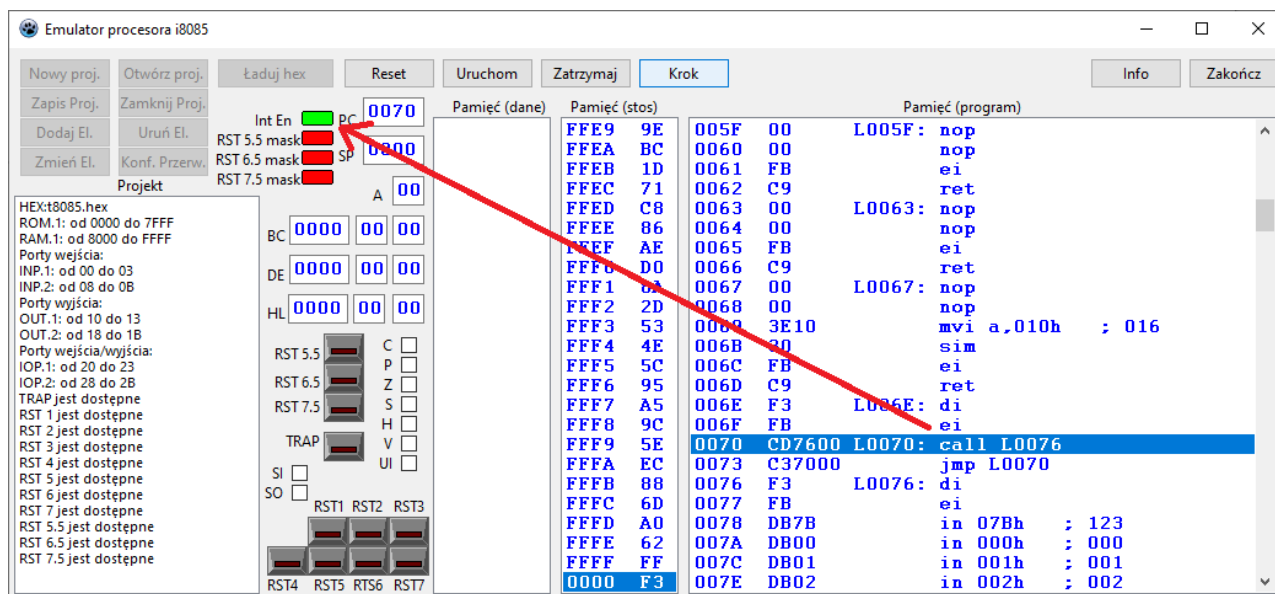
Ilustracja 3: em8085-63.png

W tym przykładzie wszystkie skoki i wywołania są „jawne”, więc nie ma „potencjalnych” martwych kodów. Po kliknięciu na „Krok” lub „Uruchom” program staruje i nie ma już możliwości „grzebania” w konfiguracji projektu. By wrócić do tej możliwości program należy zatrzymać klikając na „Zatrzymaj” (to nic, że program „stoi”, gdyż został puszczoney krokowy, ale jest w kontekście <uruchomiony>, więc będzie wymagał zatrzymania, by stały się dostępne pewne jego funkcje). Załadowanie wskaźnika stosu powoduje, że zostaje wyświetlony stos – zawartość pamięci wskazana przez rejestr SP. Stos jest pokazany oczywiście „od tyłu” i widać tu dosyć przypadkowe wartości. Funkcją programu jest wypełnienie wszystkich pamięci RAM losową zawartością po każdym kliknięciu na „Reset”. Załadowanie kodu programu (Ładuj hex) również jest połączone z automatycznym resetem.



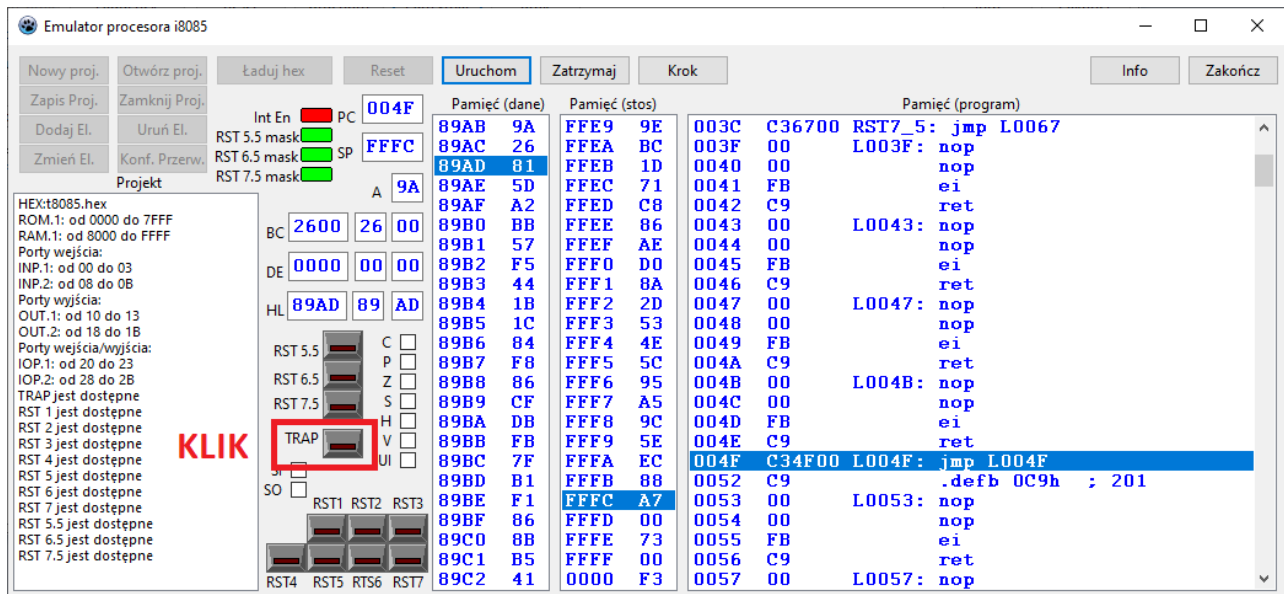
Ilustracja 4: em8085-64.png

Wykonanie instrukcji DI (odblokowanie przerwań), sygnalizuje możliwość przyjęcia klasycznych (w sensie procka i8080) przerwań sprzętowych.



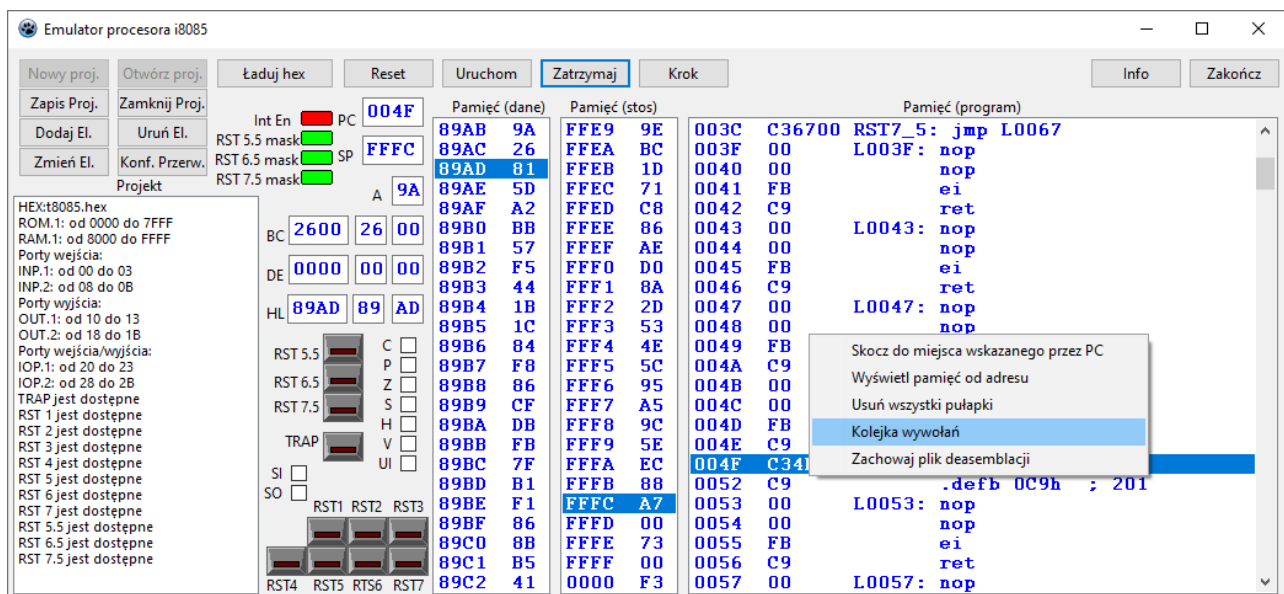
Ilustracja 5: em8085-65.png

Kliknięcie na przerwanie niemaskowalne (TRAP) powoduje skod do jego obsługi, która zatrzymuje program w ciasnej pętli (skok do samego siebie).



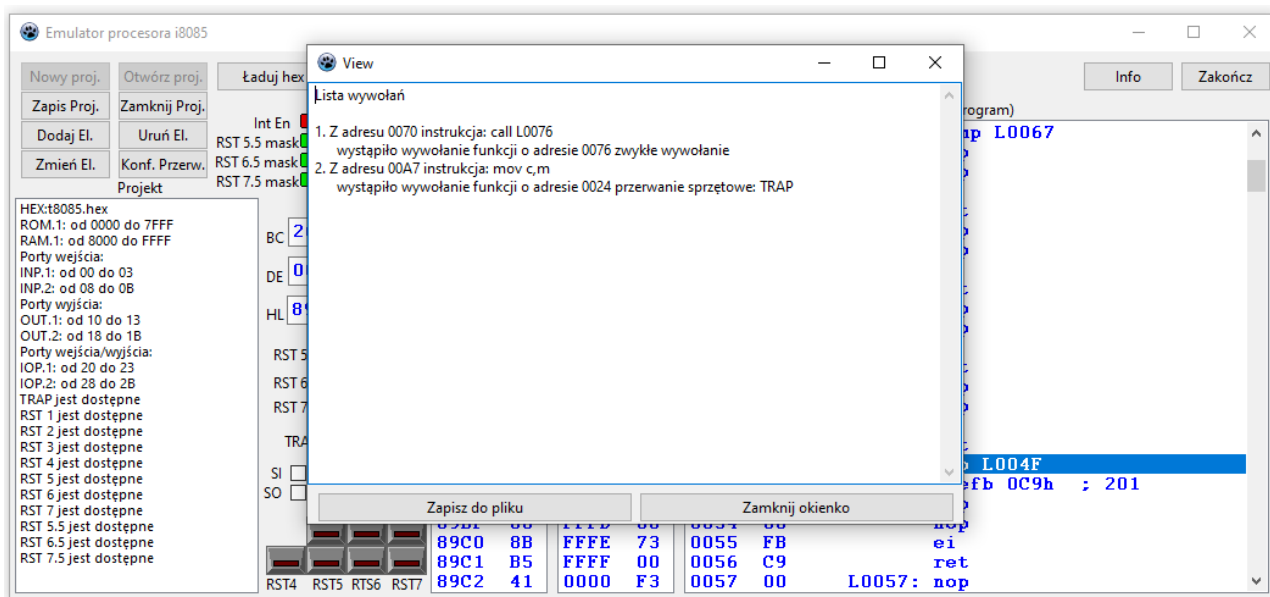
Ilustracja 6: em8085-66.png

Fajnie jest wiedzieć, co i gdzie zaszło? Co to wiadomo: TRAP, ale gdzie? Po kliknięciu prawym klawiszem z powstałego menu wybieramy „Kolejka wywołań”.



Ilustracja 7: em8085-67.png

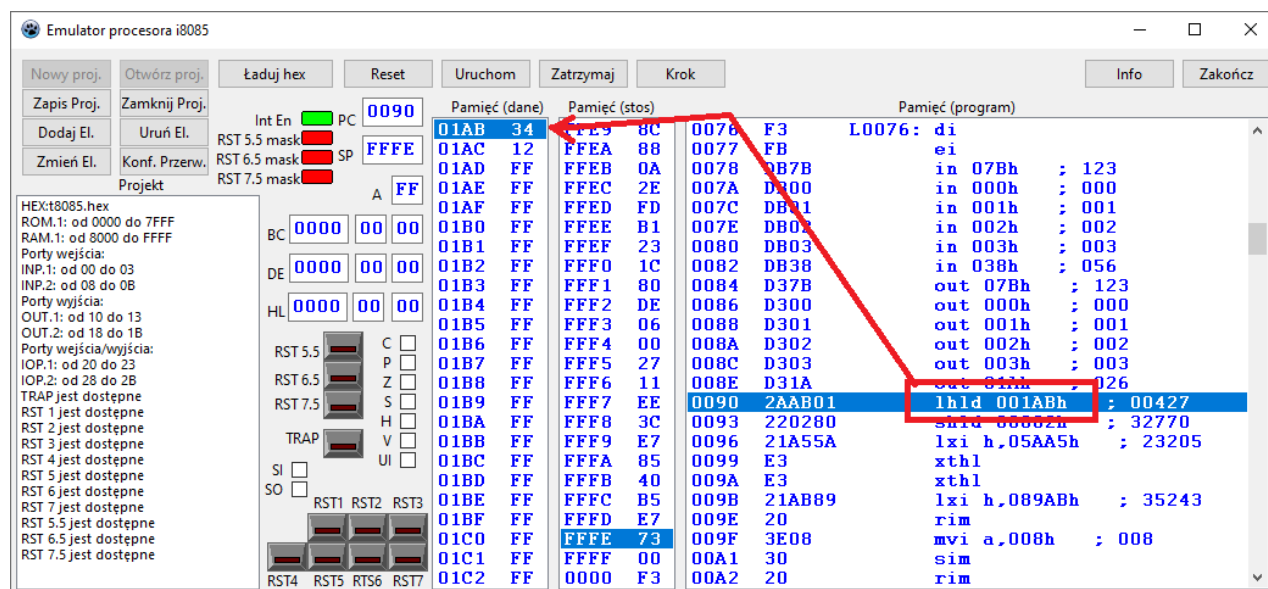
Wyświetlona jest istniejąca kolejka wywołań: normalnych wywołań oraz obsługi przerwań.



Ilustracja 8: em8085-68.png

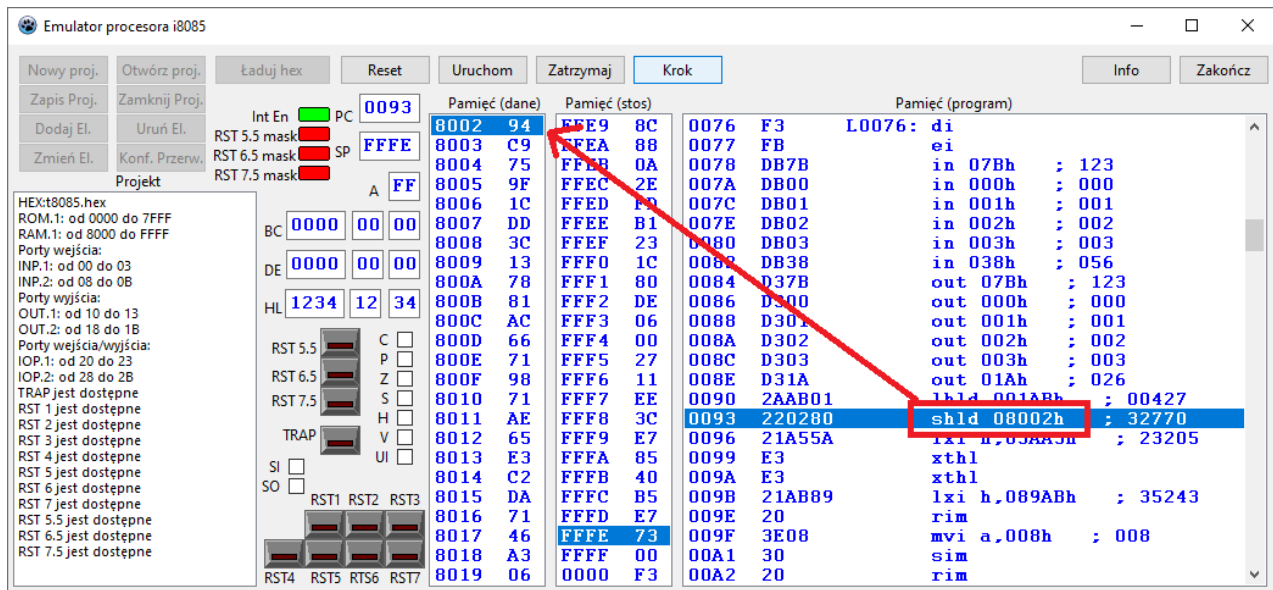
Pokazany tekst można sobie zapisać do pliku, co może być pomocne przy analizie programu. Ponieważ program został zatrzymany w obsłudze przerwań, to pozostaje jedynie przycisk „Reset” i wszystko wraca na początek.

Łaadowanie czegokolwiek z pamięci do jakiegokolwiek rejestru pokazuje docelowe lub źródłowe miejsce w pamięci w okienku danych. To wyświetlenie jest takim krokiem w „przyszłość”, stan pokazanego obszaru danych jest jeszcze przed wykonaniem instrukcji. Poniżej LHLD z podanym adresem 1AB hex pokazuje, że do pary HL zostanie wpisane 1234 hex (starsza część jest na starszym adresie).



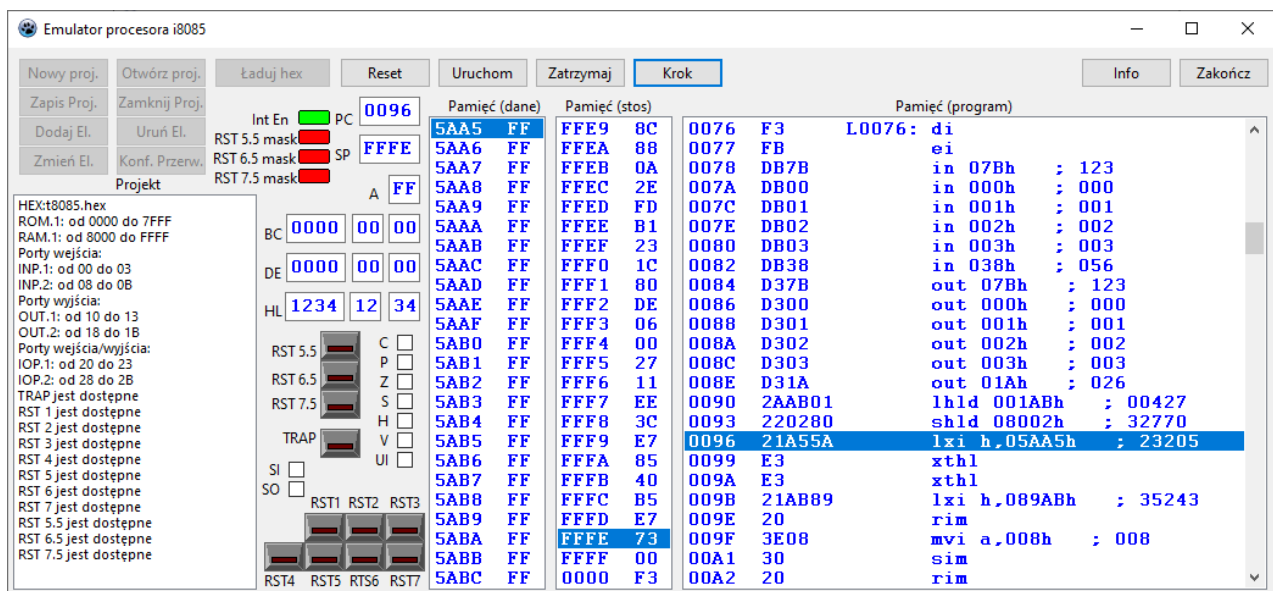
Ilustracja 9: em8085-69.png

Przed kolejną instrukcją SHLD w okienku danych pokazana jest zawartość pamięci jeszcze przed wykonaniem instrukcji, która jest losowa (bo to pamięć RAM wypełniona losowo w wyniku akcji reset).



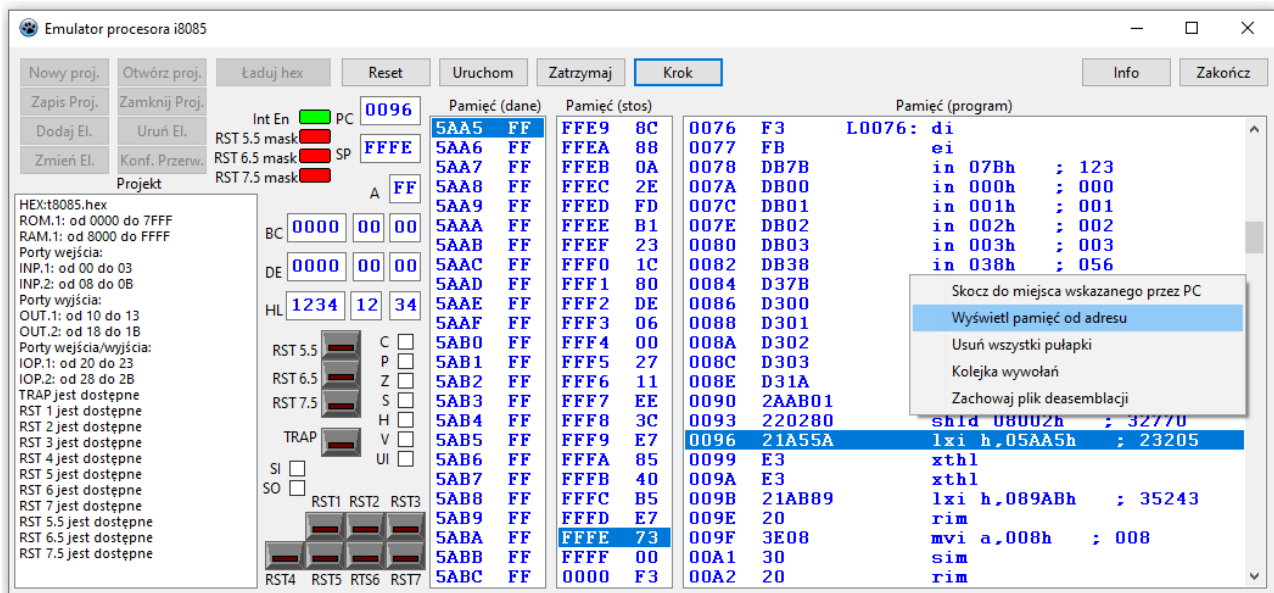
Ilustracja 10: em8085-70.png

Ponieważ kolejna instrukcja ładuje do HL stałą i program próbuje pokazać, gdzie to wskazuje, to nie widać skutku operacji zapisu.



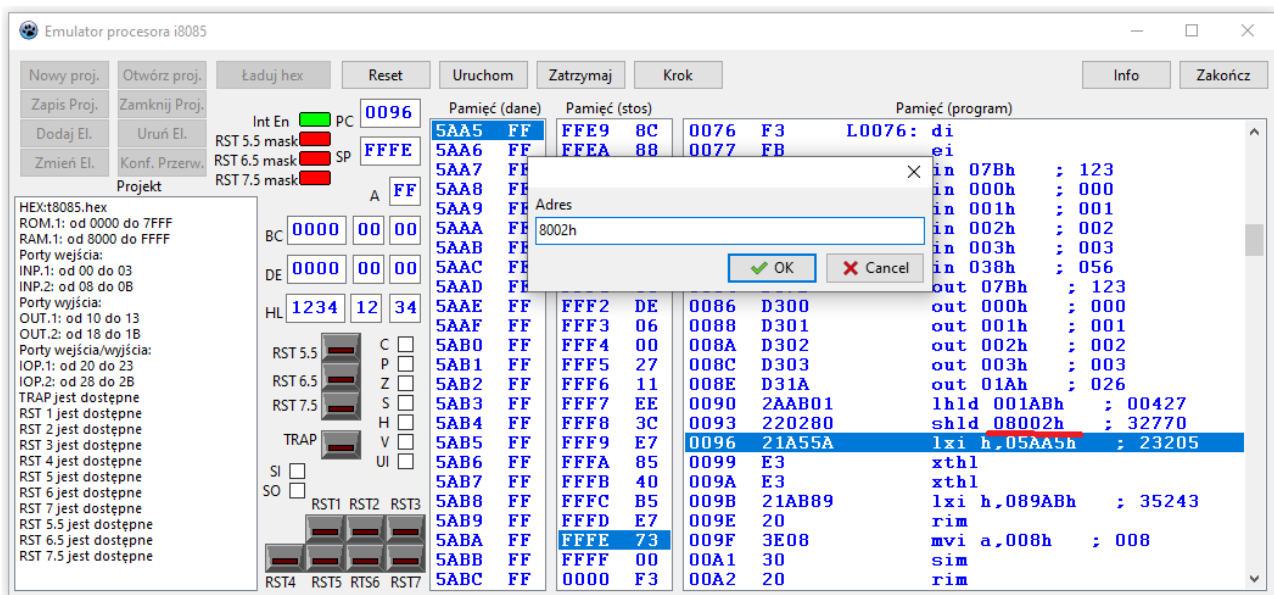
Ilustracja 11: em8085-71.png

Możliwym wyjściem jest kliknięcie prawym klawiszem i z powstałego menu wybrać:



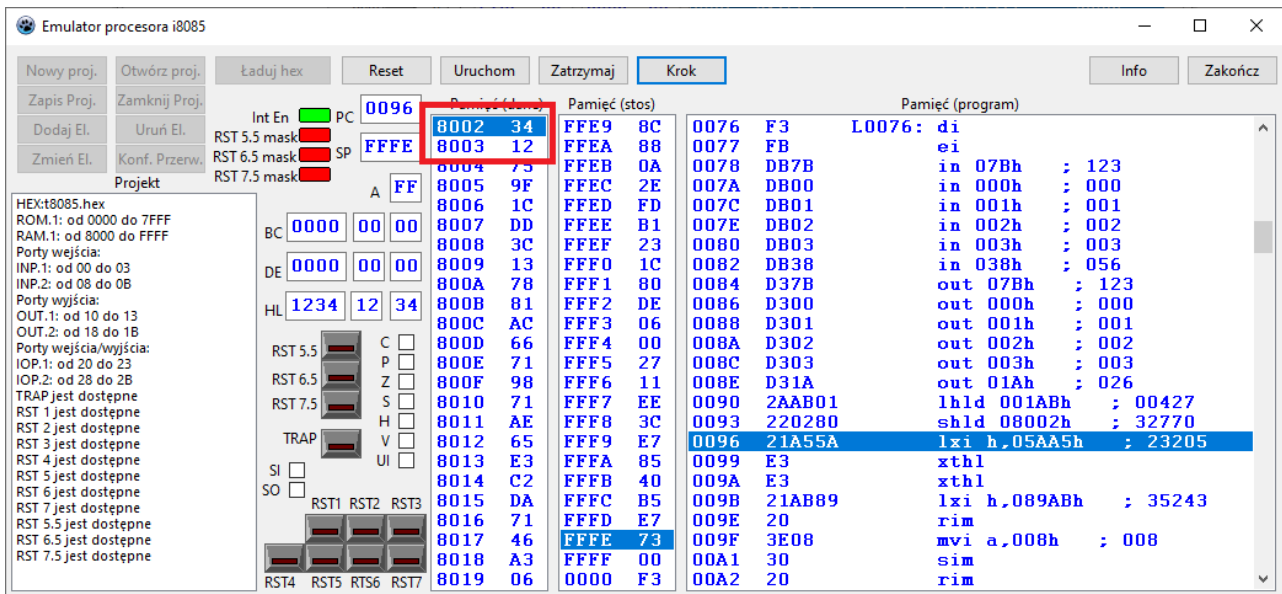
Ilustracja 12: em8085-72.png

Tam był adres 8002 hex, więc taki zostaje wpisany.

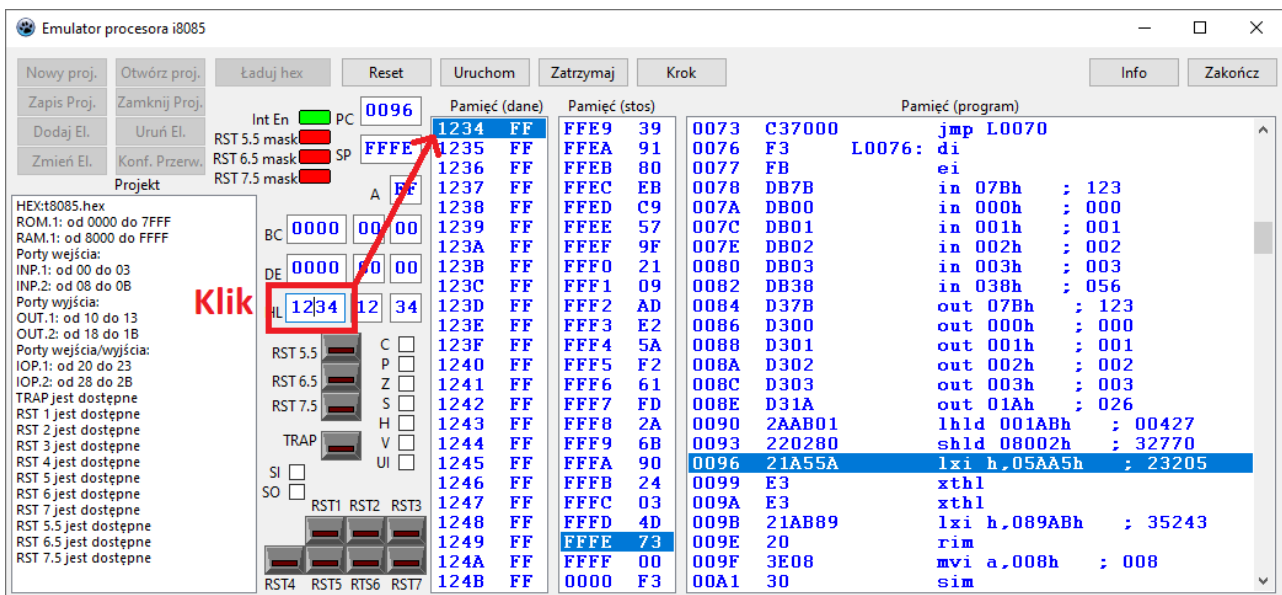


Ilustracja 13: em8085-73.png

Wcześniej było C994 hex, teraz tam jest 1234 hex.



By wrócić, wystarczy kliknąć na pole HL.

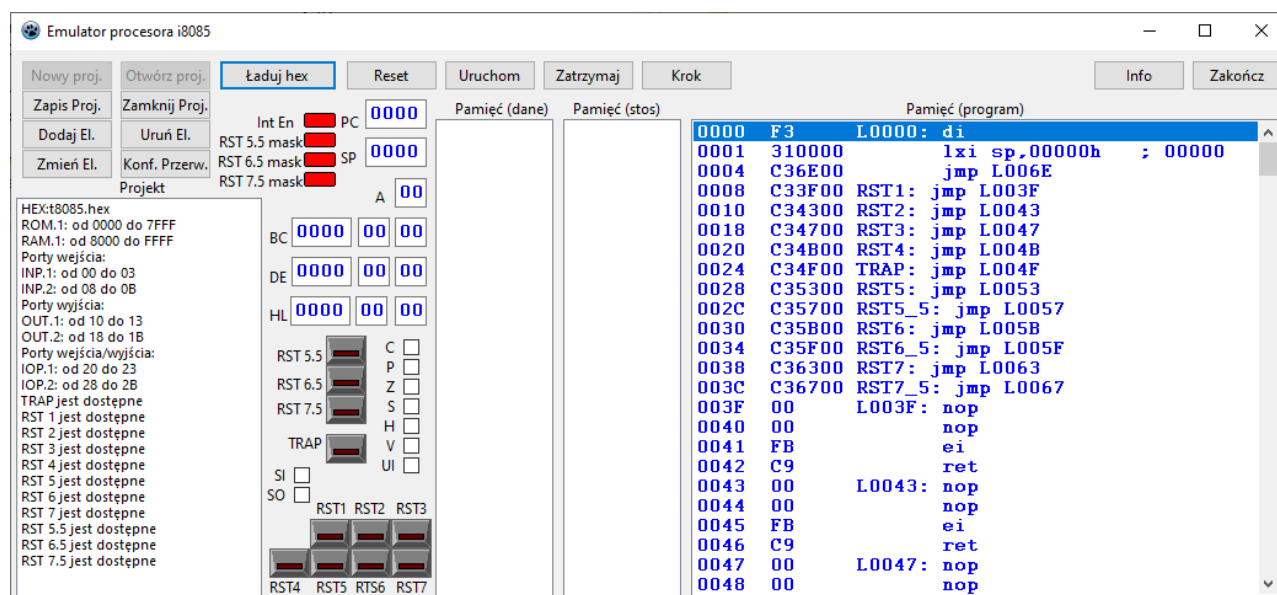


Ilustracja 14: em8085-75.png

Identycznie kliknięcie na DE lub BC również pokazuje w okienku pamięć adresowaną przez odpowiednie pary rejestrów.

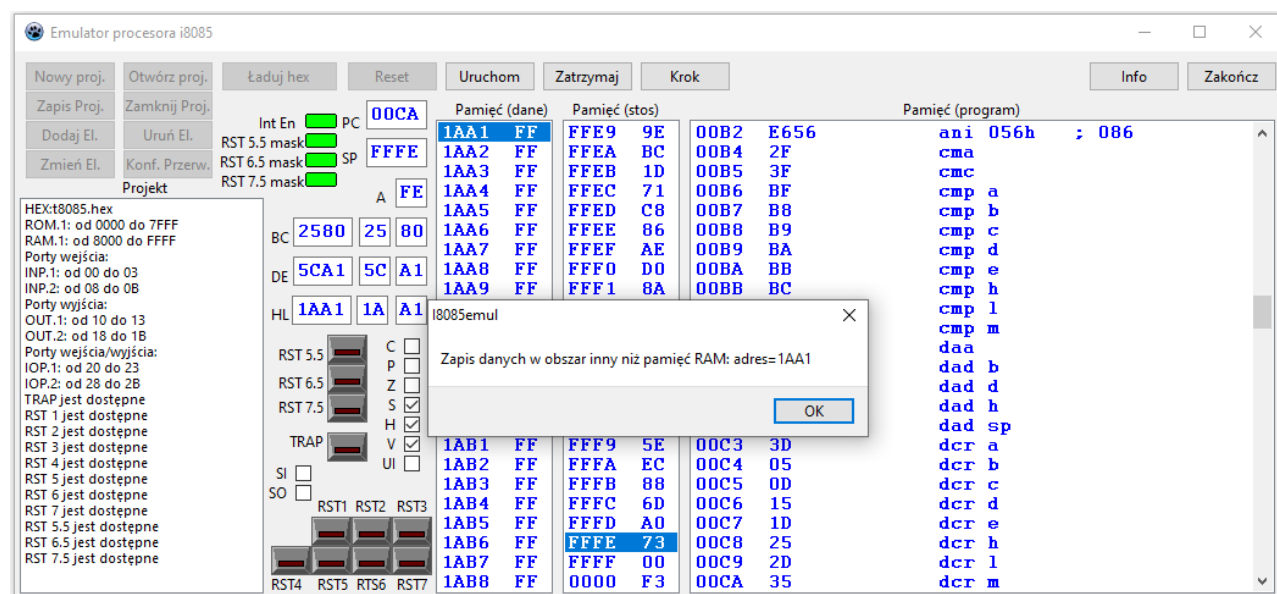
Modyfikacja stanów rejestrów

Po załadowaniu kodu programu do pamięci emulatora (po wcześniejszym otwarciu projektu), program można puścić na żywioł. Istotną funkcją emulatora jest „pilnowanie” by wszystko szło właściwie oraz zgłaszanie „nadużyć”.



Ilustracja 15: em8085-81.png

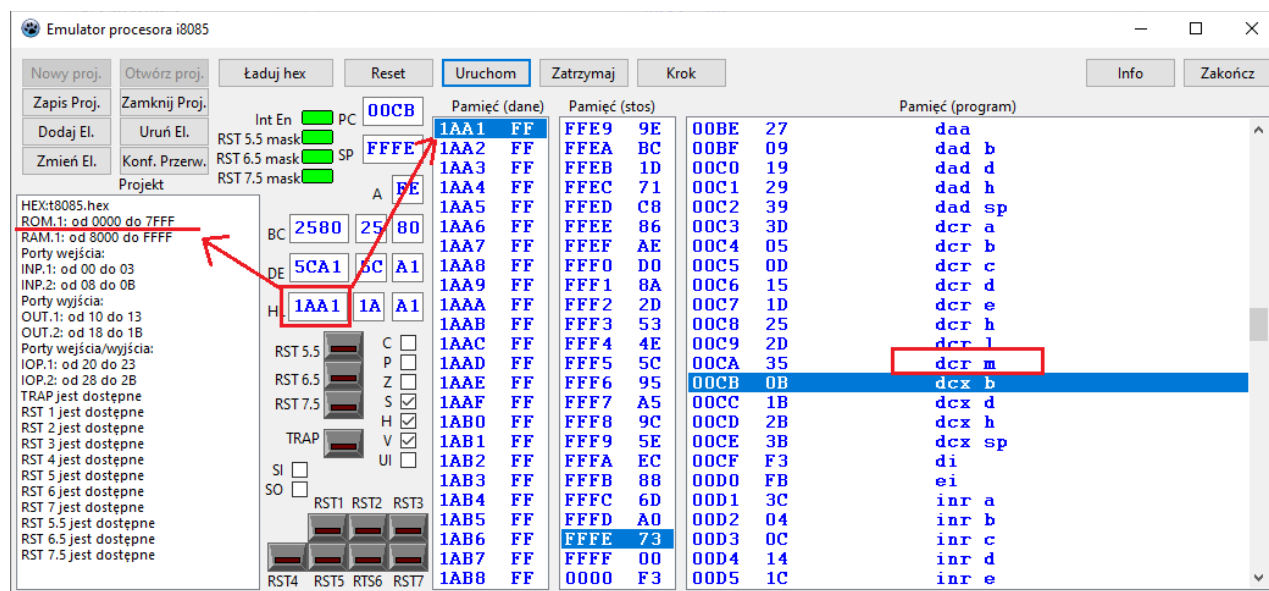
Przykładowo, zaistniało naruszenie przestrzeni pamięci ROM, gdyż zapis czegokolwiek w ten obszar jest niedopuszczalny.



Ilustracja 16: em8085-82.png

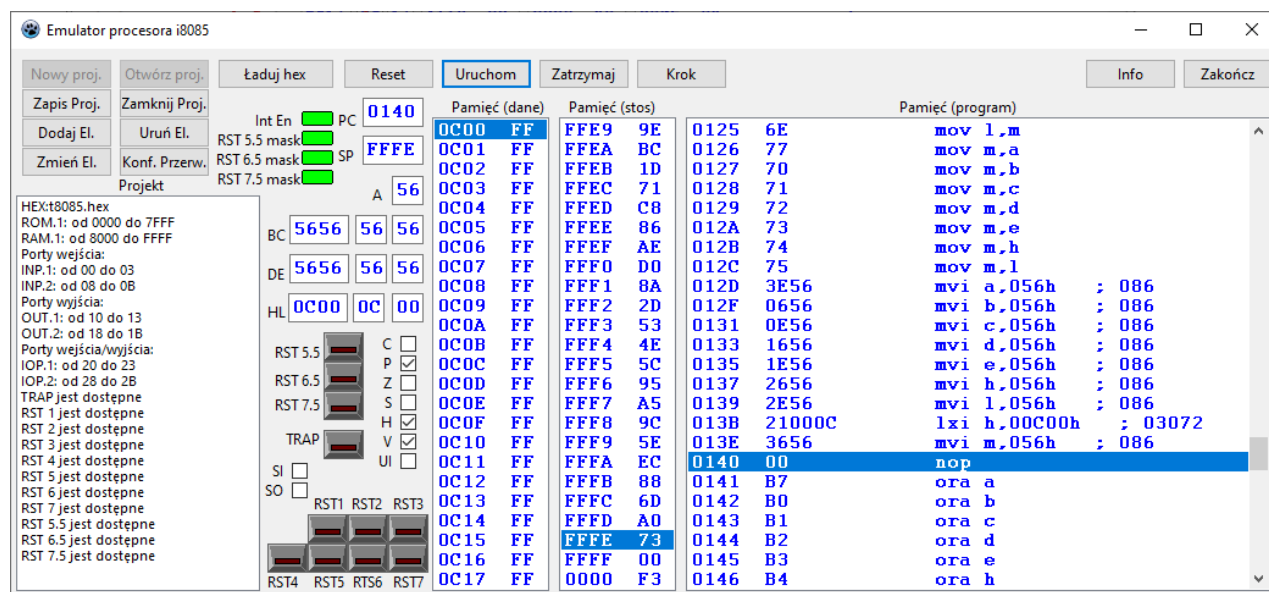
Wykonanie programu zostało zatrzymane z odpowiednim komunikatem. Problem wygenerowała instrukcja DCR M, której zadaniem jest dekrementacja bajtu

pamięci o adresie wskazanym zawartością pary rejestrów HL. Wskazuje on na obszar pamięci ROM. Odczyt z tego obszaru jest możliwy, jednak zapis tam technicznie nie jest możliwy.

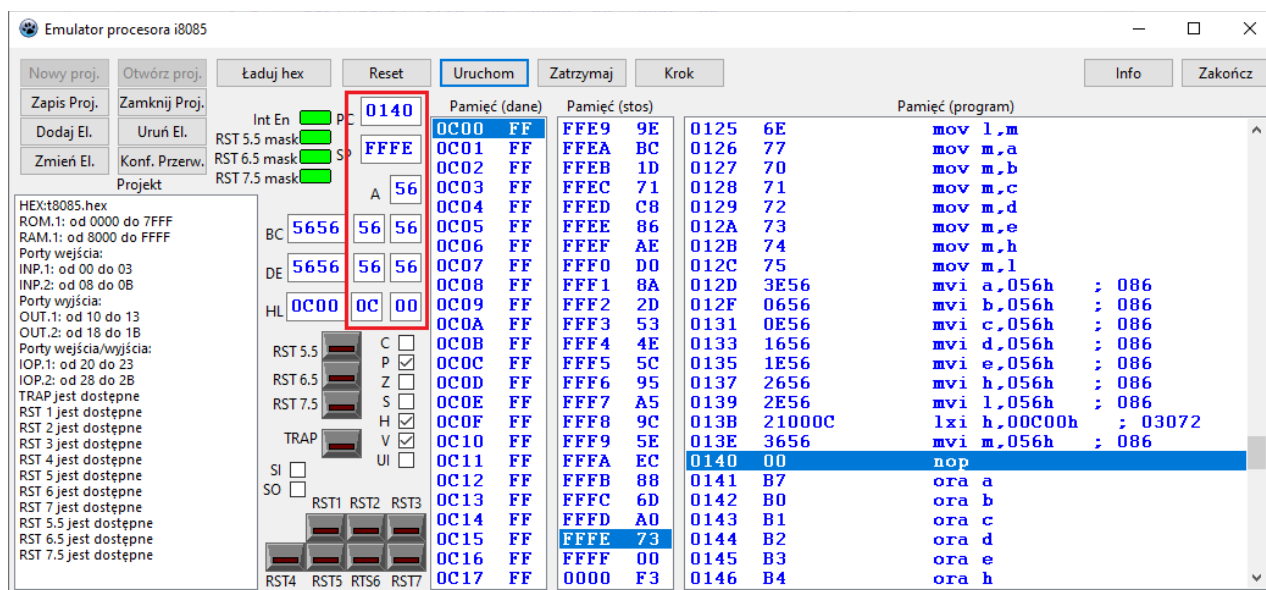


Ilustracja 17: em8085-83.png

Podobna sytuacja wystąpiła nieco później. Przed instrukcją MVI M,... została załadowana para rejestrów HL, ale w wyniku „własnej pomyłki” to nie jest właściwa wartość w tej parze rejestrów.

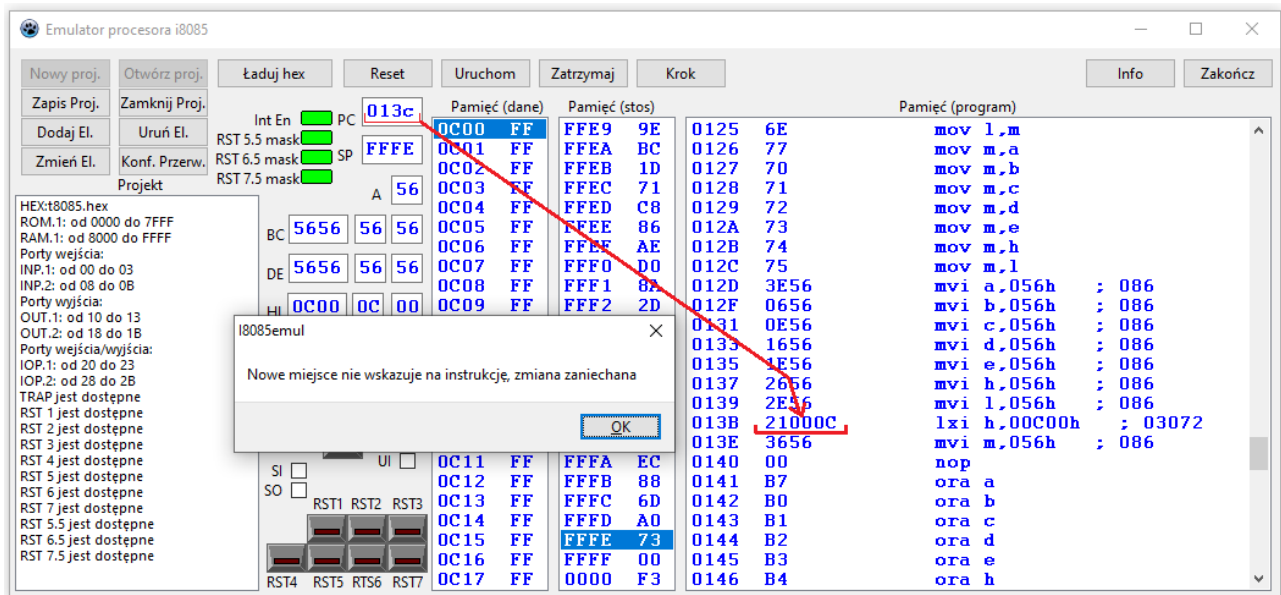


Możliwe jest dokonanie modyfikacji bez potrzeby edycji programu źródłowego i jego kompilacji (no chyba, że będzie to zachodzić wielokrotnie, to wtedy jest to bardziej uzasadnione). Można dokonać korekty każdego pokazanego niżej rejestru:



Ilustracja 18: em8085-85.png

Operacja zmiany zawartości rejestrów jest możliwa w sytuacji zatrzymanego programu lub gdy program jest puszczany krokowo. Należy kliknąć na wybrany rejestr i wprowadzić szesnastkowo nową wartość. Zmiany par rejestrów są realizowane przez dwie zmiany zawartości pojedynczych rejestrów (starsza część jest z lewej strony, młodsza z prawej strony). Zmiana rejestru BC, DE, HL jednocześnie wyświetla zawartość pamięci danych wskazaną przez modyfikację rejestrów składowych. Również możliwa jest zmiana zawartości wskaźnika stosu, jednak tu trzeba zachować czujność, bo może to doprowadzić, że program się zgubi (po przykładowo wykonaniu instrukcji RET). Pewnym ograniczeniem podlega modyfikacja rejestru PC. Istnieją instrukcje jednobajtowe, dwubajtowe oraz składające się z trzech bajtów. Wymagane jest by trafić zawsze na pierwszy bajt instrukcji (po deasemblacji wiadomo, gdzie zaczyna się każda instrukcja). W tym miejscu program nie da sobie zrobić krzywdy.

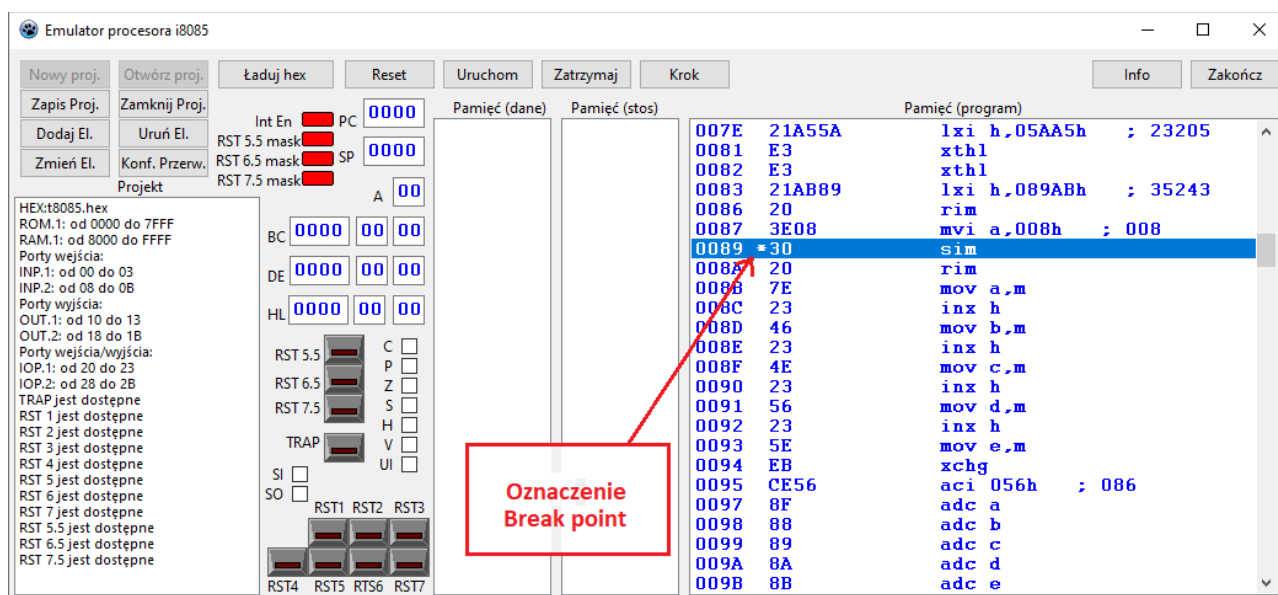


Ilustracja 19: em8085-86.png

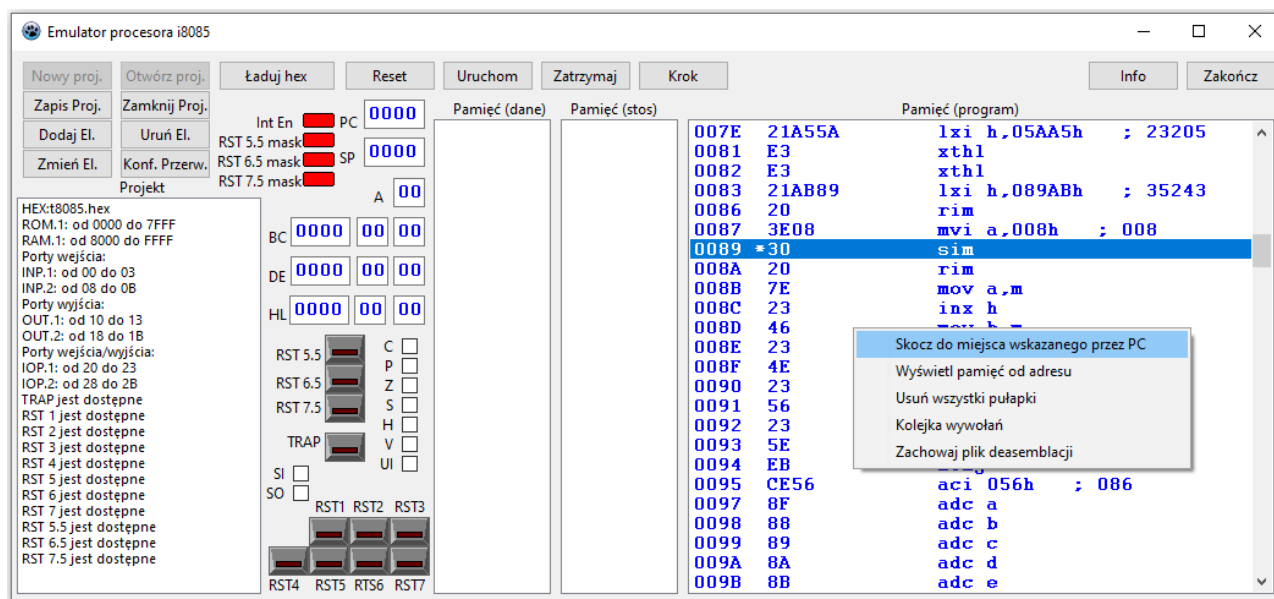
Wpisanie przykładowo do PC wartości 13C wskazuje na drugi bajt instrukcji LXI (czyli operand tej instrukcji).

Break point’y w programie

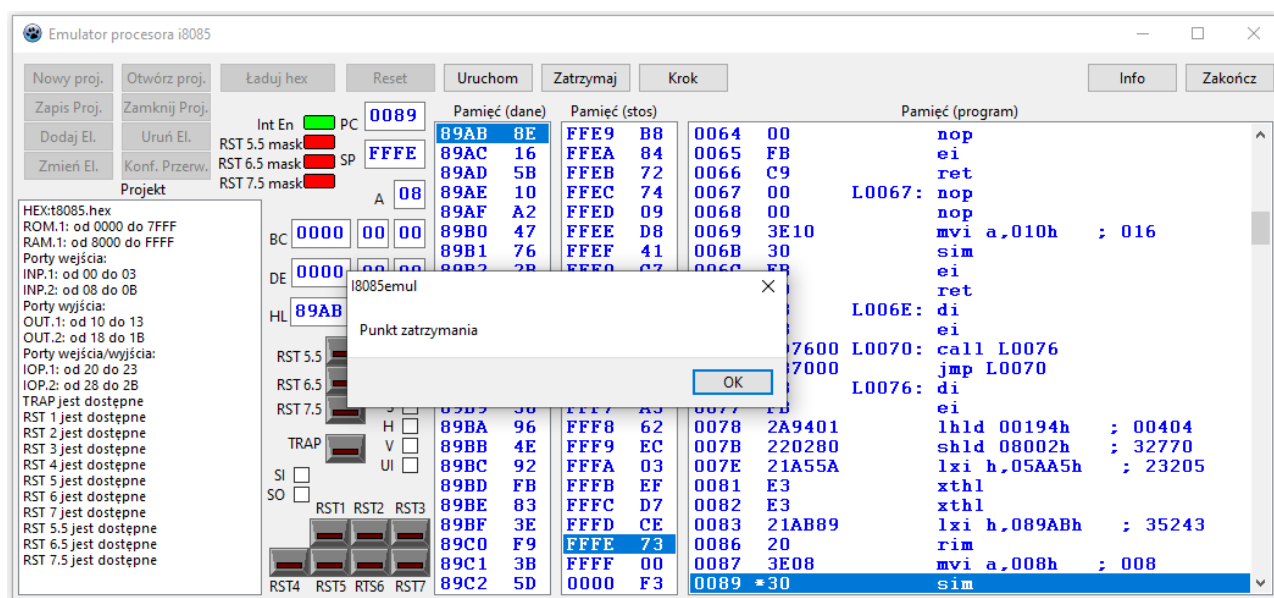
W emulatorze nie może zabraknąć funkcji związanych z pułapkami zwanymi break pointy. By to uzyskać to należy mieć załadowany program (ten w intelu).



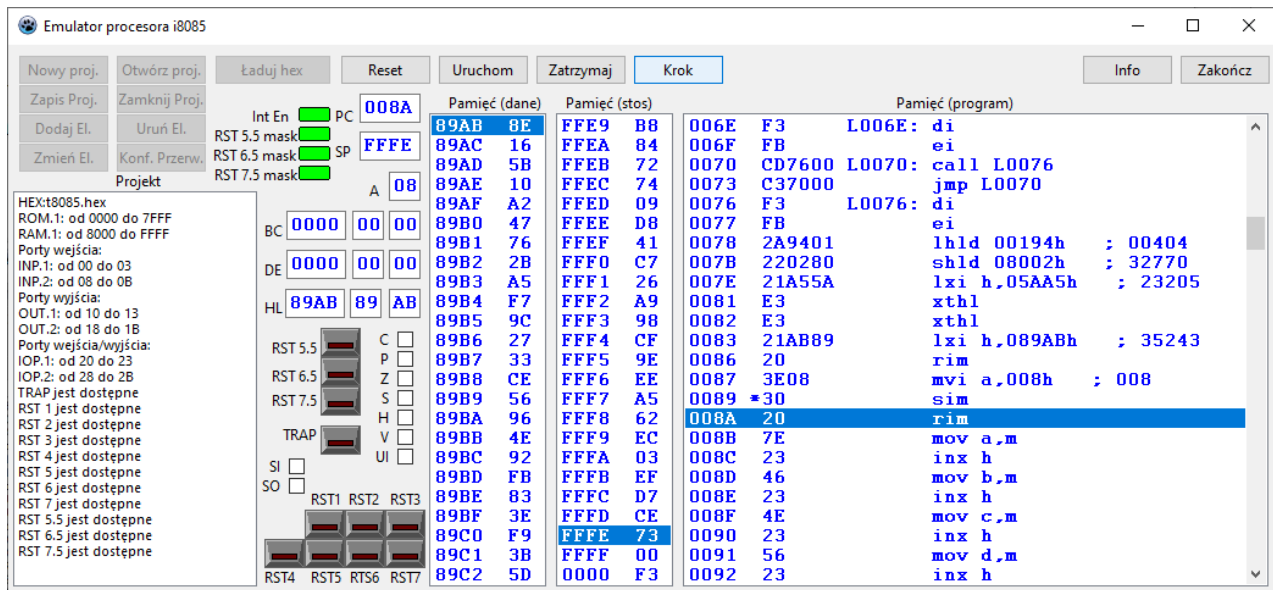
W okienku z programem „przewinąć się” do punktu, w którym wykonanie programu ma zostać wstrzymane i zrobić tam dwuklik. Miejsce to zostaje oznaczone gwiazdką. Ponowny dwuklik w tym miejscu zdejmuję z tej instrukcji pułapkę. Jeżeli wskazana instrukcja nie była oznaczona flagą pułapki, to zostanie ona ustawiona, jeżeli była oznaczona flagą pułapki, to zostaje zdjęta. Pułapek można w programie ustawić dowolną ilość. Efektem operacji pułapkowej jest przestawienie podświetlenia w okienku programu (co nie zmienia wartości zapisanej w rejestrze PC). By wrócić należy użyć prawego klawisza myszki i wybrać opcję „Skocz do miejsca wskazanego przez PC”. I wszystko wraca do normy.



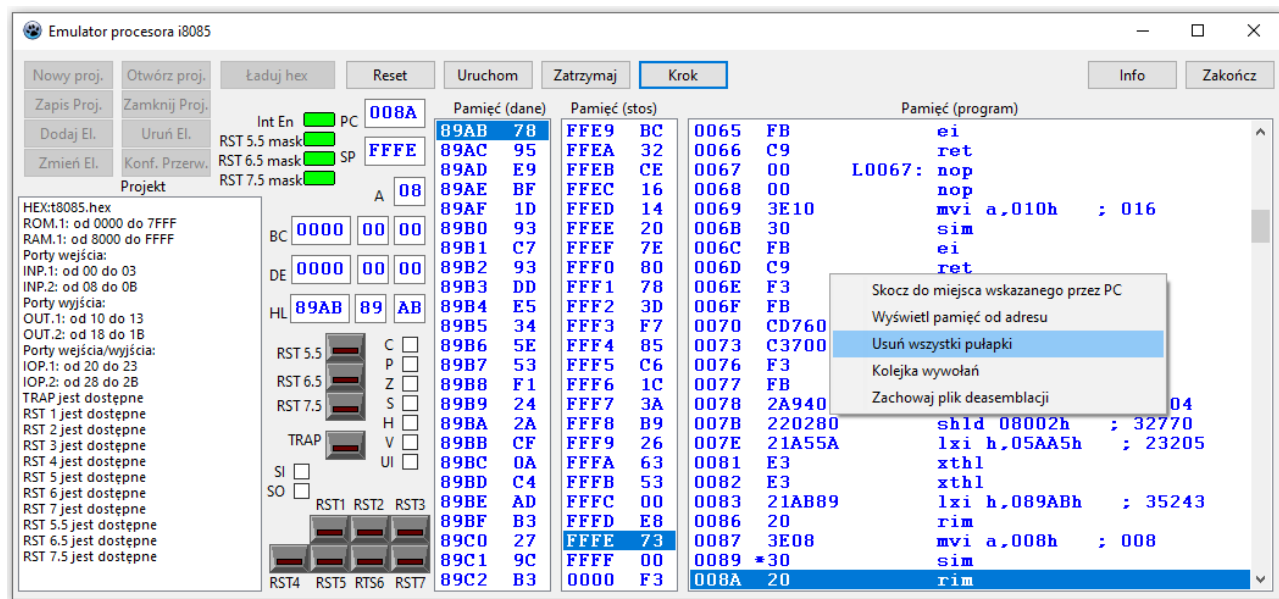
Uruchomienie programu „na żywo” (przycisk <Uruchom>) zatrzymuje wykonanie programu we wskazanym miejscu (program jest przed wykonaniem instrukcji). By przejść przez ten punkt należy kliknąć przycisk <Krok>.



Przy pracy krokowej wykonanie zawsze przechodzi przez pułapki bez zatrzymania wykonania.

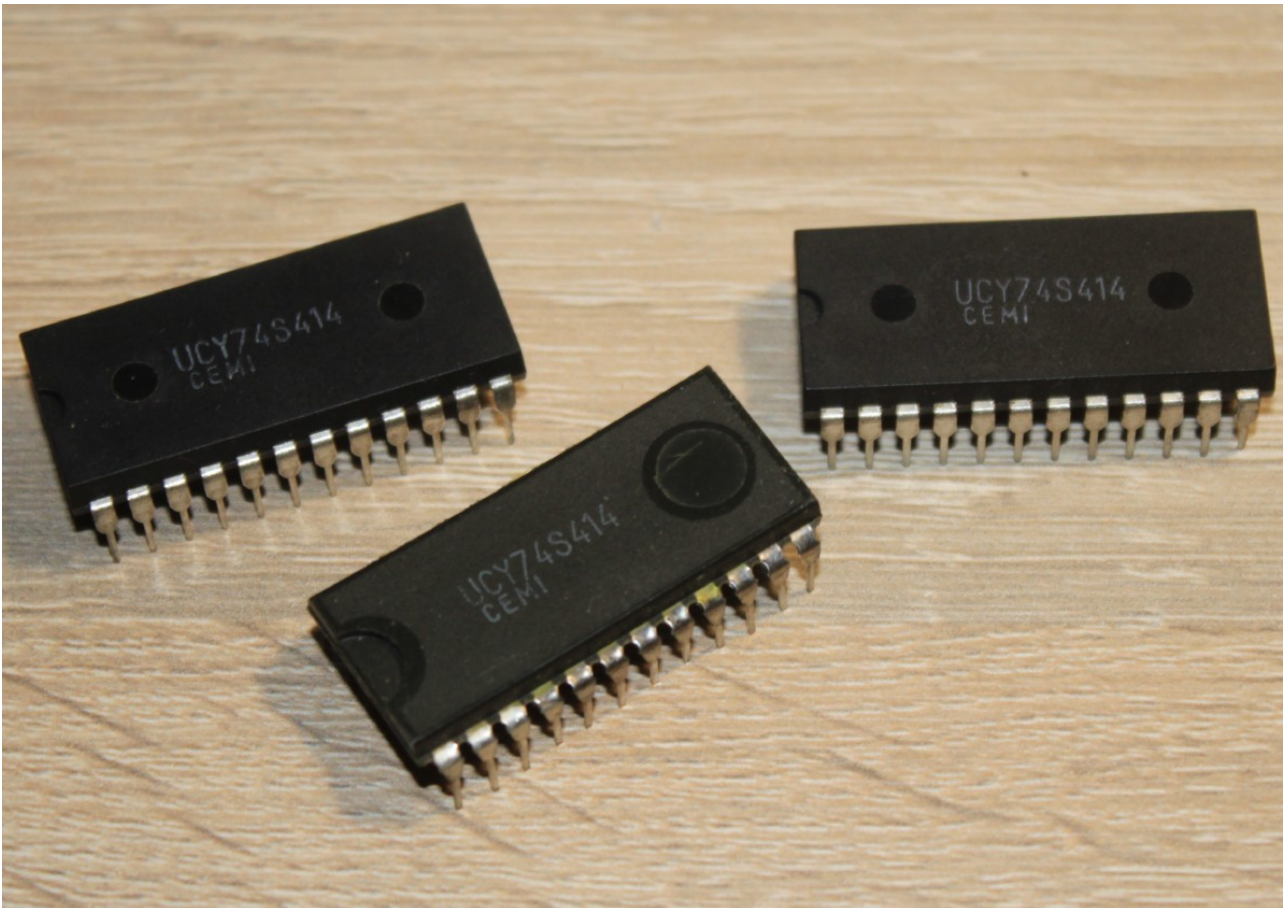


By usunąć pułapki należy kliknąć prawym klawiszem myszki i wybrać <Usuń wszystkie pułapki>.



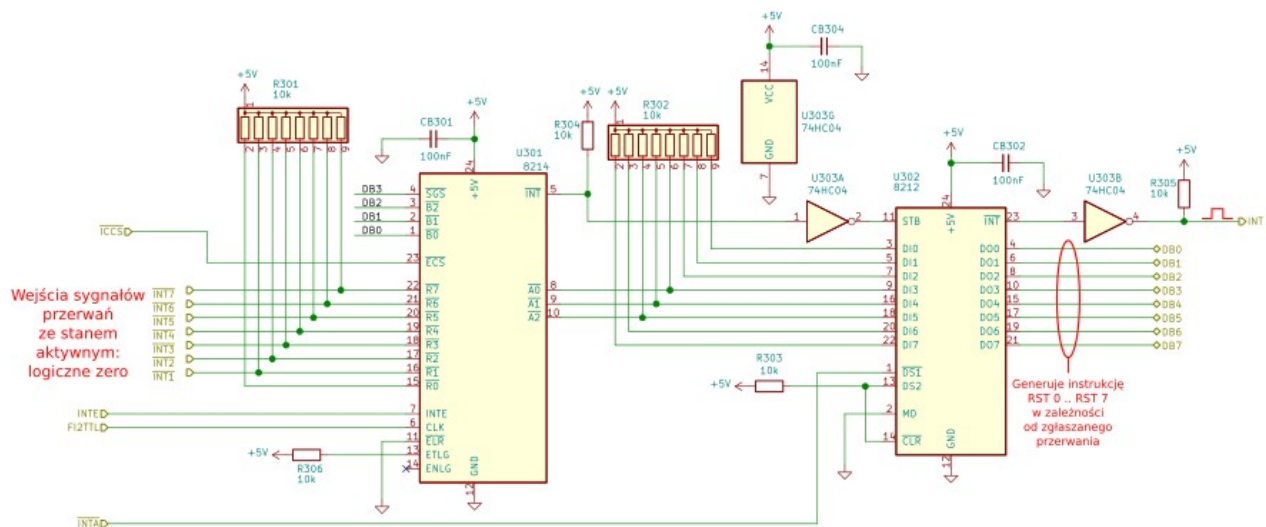
Jeżeli ktoś ma ochotę poeksperymentować z programem, to ... zapraszam.

Obsługa przerwań w emulatorze



Ilustracja 20: em8085_90.png

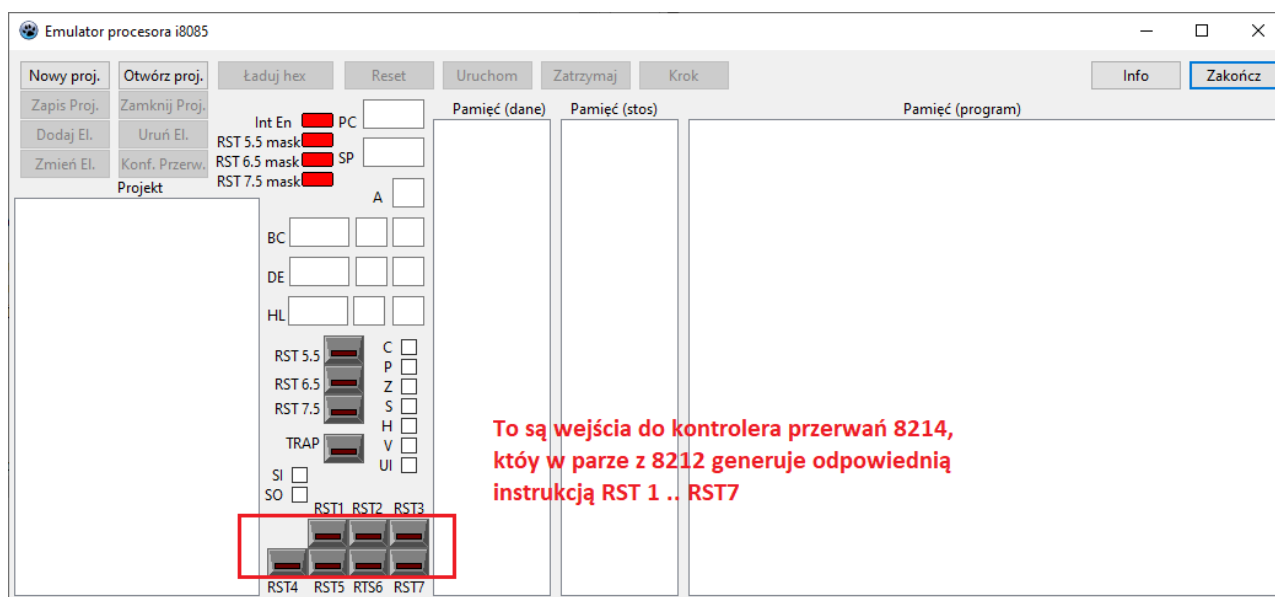
Emulator procka realizuje obsługę przerwań opartą o prosty (ale skuteczny) kontroler i8214. Jego zadaniem (w kooperacji w parze z i8212) jest wypracowanie odpowiedniej instrukcji, która jest zgłaszana w chwili akceptacji przyjęcia przerwania przez procka. Schemat takiego rozwiązania jest następujący:



Ilustracja 21: em8085_91.png

Akceptacja przez procka przerwań prowadzi do wystawienia na szynie danych jednej z możliwych instrukcji *RST n* (no może z wyjątkiem *RST 0*, bo to będzie tożsame z resetem).

I dokładnie taka filozofia działania jest zawarta w emulatorze. Kliknięcie na jeden z przycisków zgłoszenia przerwań generuje sygnał do kontrolera.



Ilustracja 22: em8085_92.png

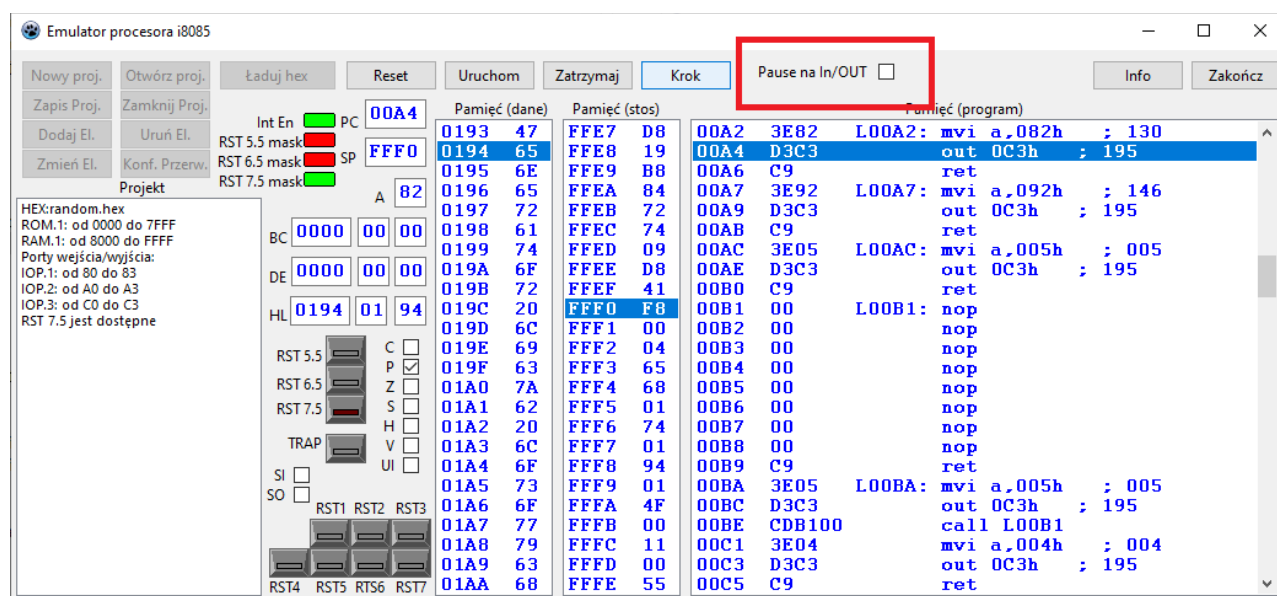
Jeżeli jest zezwolenie na obsługę przerwań, to ta obsługa sprowadza się do „zasymulowania” wykonania odpowiedniego wariantu instrukcji RST n (dokładnie jak na schemacie wyżej). Także przycisk RST1 generuje instrukcję RST 1, i tak dalej aż do przycisku RST7, który generuje instrukcję RST 7. Obsługa przerwań połówkowych jest zrealizowana trochę odmiennie: na stos wędruje bieżąca zawartość rejestru PC i następuje wpisanie do rejestru PC odpowiedniego adresu wynikającego z obsługiwanego przerwania. Każde przerwanie RST1 .. RST7 oraz połówkowe (z wyjątkiem RST 7.5) gasi „lampkę” sygnalizującą zgłoszenie przerwania na przycisku), RST 7.5 wymaga potwierdzenia (odpowiednie zakłucie w instrukcji SIM). Za każdym razem również emulator sam z siebie blokuje przyjęcie kolejnych przerwań (robi DI).

Symulacja bardziej rozbudowanego układu obsługi przerwań i8259 nie jest zrealizowana.

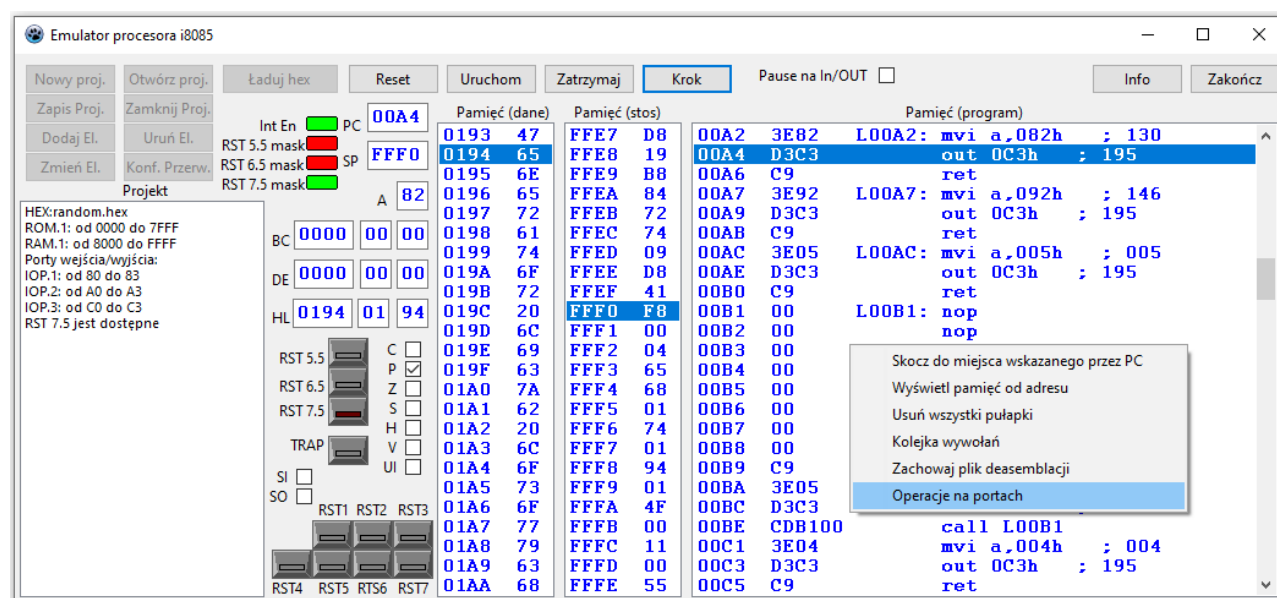
Obsługa przerwania niemaskowalnego TRAP (w nomenklaturze Ziloga: NMI) przebiega podobnie do przerwań połówkowych. Na stos wędruje stan rejestru PC i następnie do jego wpisywany jest odpowiednia adres związany z obsługą tego przerwania.

Operacje na portach

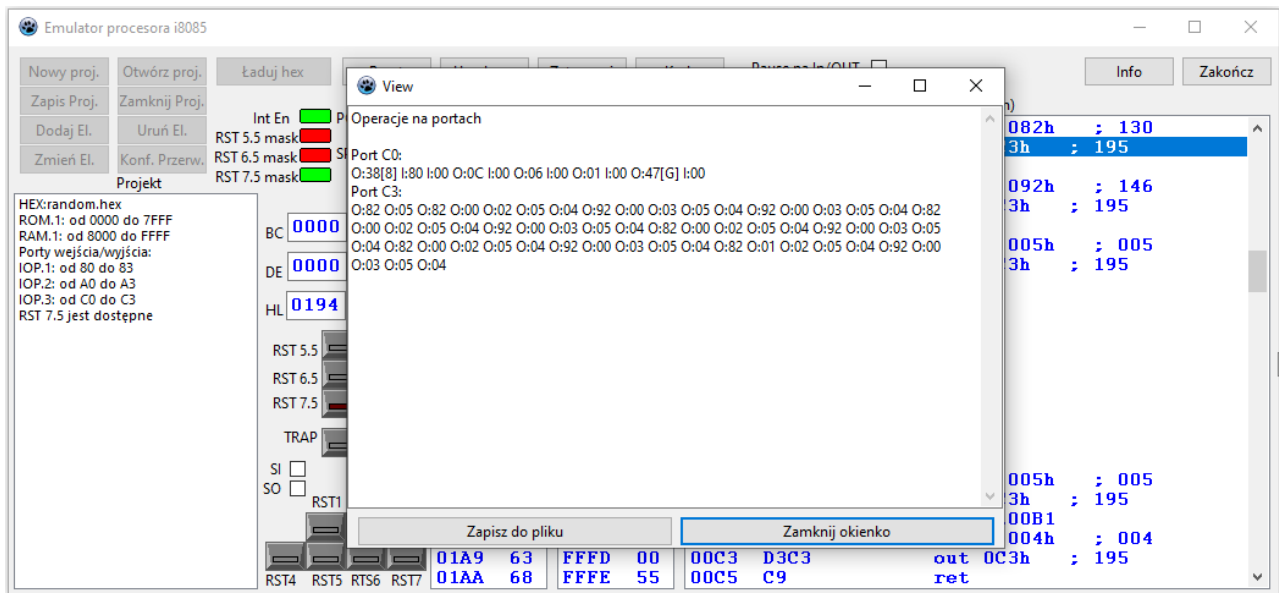
Używając emulatora, wyszło mi, że warto go trochę poprawić. Otóż, operacje na portach są sygnalizowane komunikatem w okienku, gdzie należy kliknąć <OK>. Na dłuższą metę to trochę nużące, więc postanowiłem trochę udoskonalić swój program. Na formie doszedł CheckBox określający, czy program ma informować o dokonanej transmisji na port (jak dotychczas), czy nie informować o tym zdarzeniu.



Jeżeli „ptaszek” nie jest zaznaczony, to wszystko jedzie po cichu. Oczywiście istnieje możliwość sprawdzenia, co takiego poszło na porty. W tym celu należy kliknąć prawym klawiszem i wybrać właściwą pozycję:



W reakcji zostają wyświetlone wszystkie dokonane dotychczas operacje na portach, przykładowo:



gdzie jest podany adres portu i co latało na styku do portu. Literka „O” sygnalizuje operację wyjścia, literka „I” dotyczy operacji wejścia. Dalej po znaku ‘:’ jest transmitowana wartość w zapisie szesnastkowym. W przypadku, gdy dane są w zakresie znaków pisarskich (ISO-7), to w nawiasach kwadratowych jest pokazany znak. Istnieje możliwość zapisu tych danych do pliku.