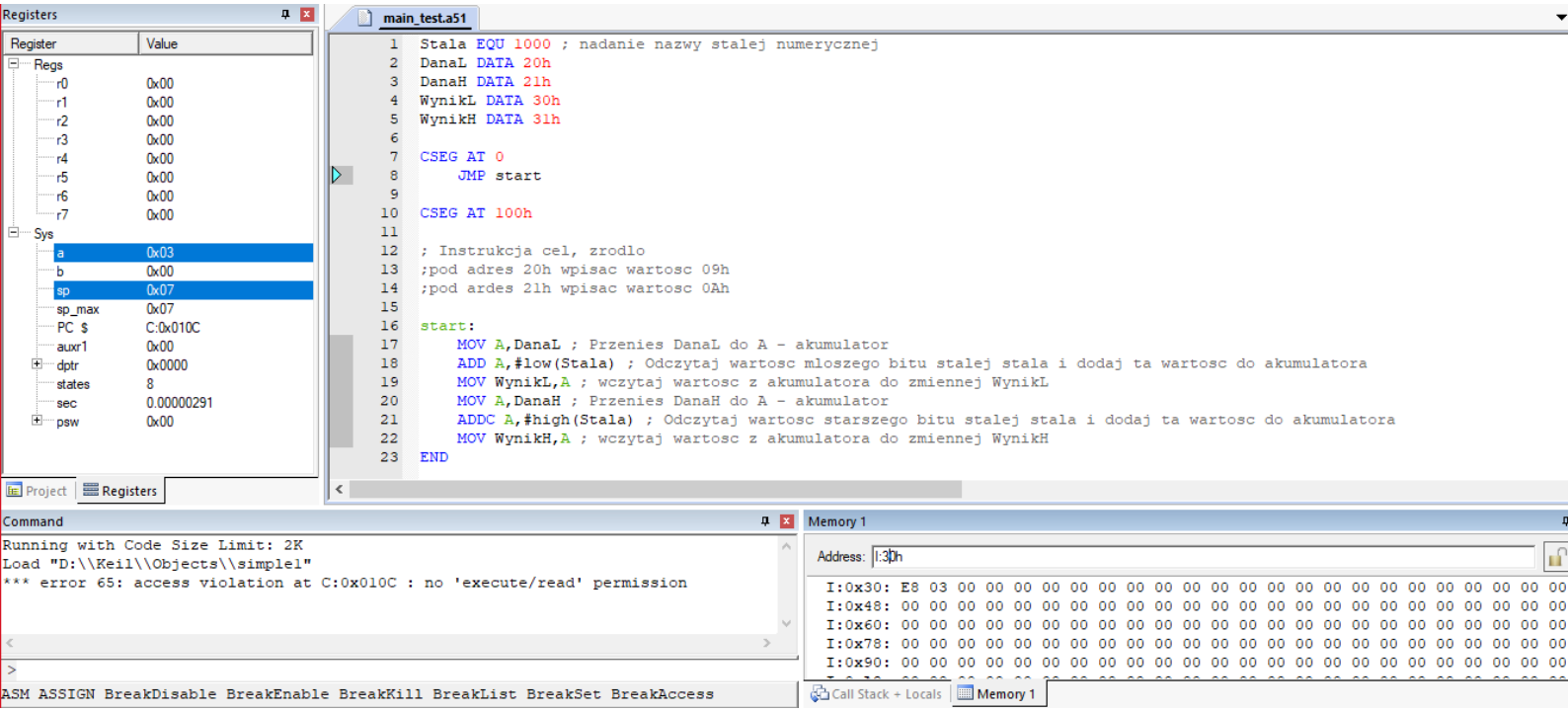
Mikroprocesory I mikrokontrolery	Temat: Wstęp do programowania w asemblerze 8051
Grupa: 21b	Michał Lechowicz Mateusz Moneta

Cel ćwiczenia

Celem ćwiczenia było zapoznanie się z podstawami assemblera dla platformy 8051, oraz środowiska programowania Keil uVision.

Zadanie 1

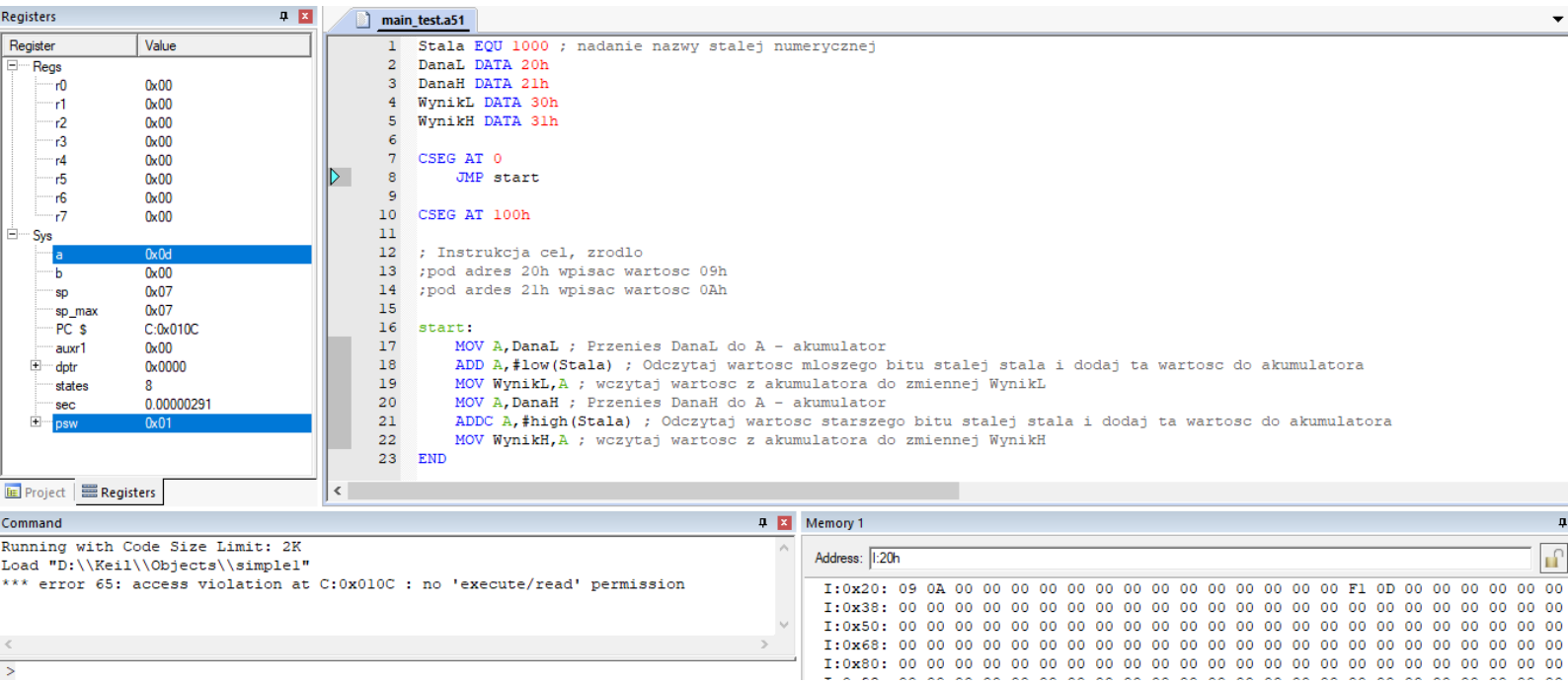
1. Analiza programu:



Rys. 1 Dodawanie 16-bitowe

Powyższy przykład dodaje najpierw mniej znaczący bajt liczby, zapisuje go do pamięci RAM, a następnie dodaje z przeniesieniem starszy bajt, również zapisując wynik do RAM. Architektura 8051 jest autorstwa firmy Intel, więc wykorzystuje kodowanie **Little Endian**.

2. Zmiana danych w przestrzeni adresowej 20h i 21h

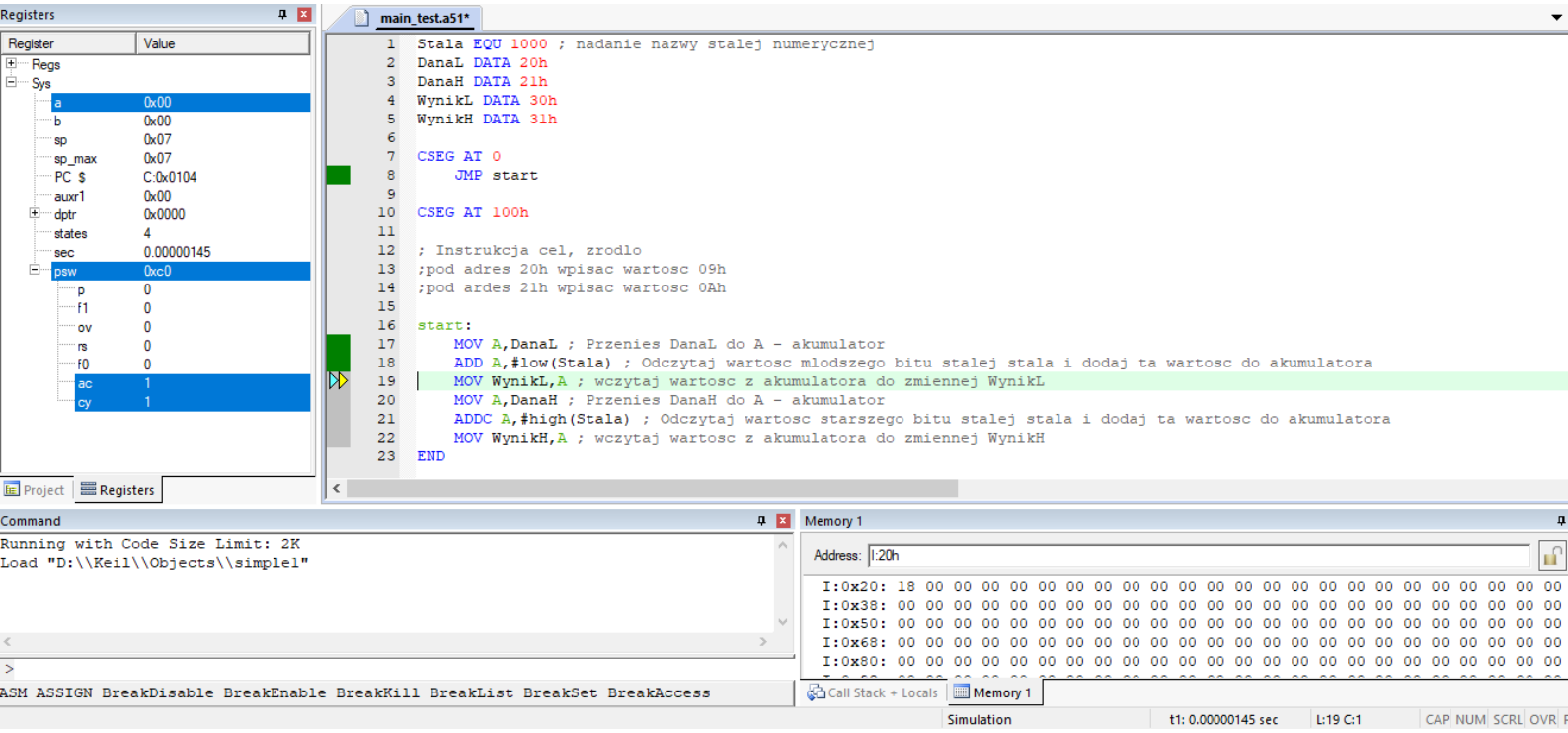


Rys. 2 Dodawanie 16-bitowe z wartościami początkowymi 09h i 0Ah

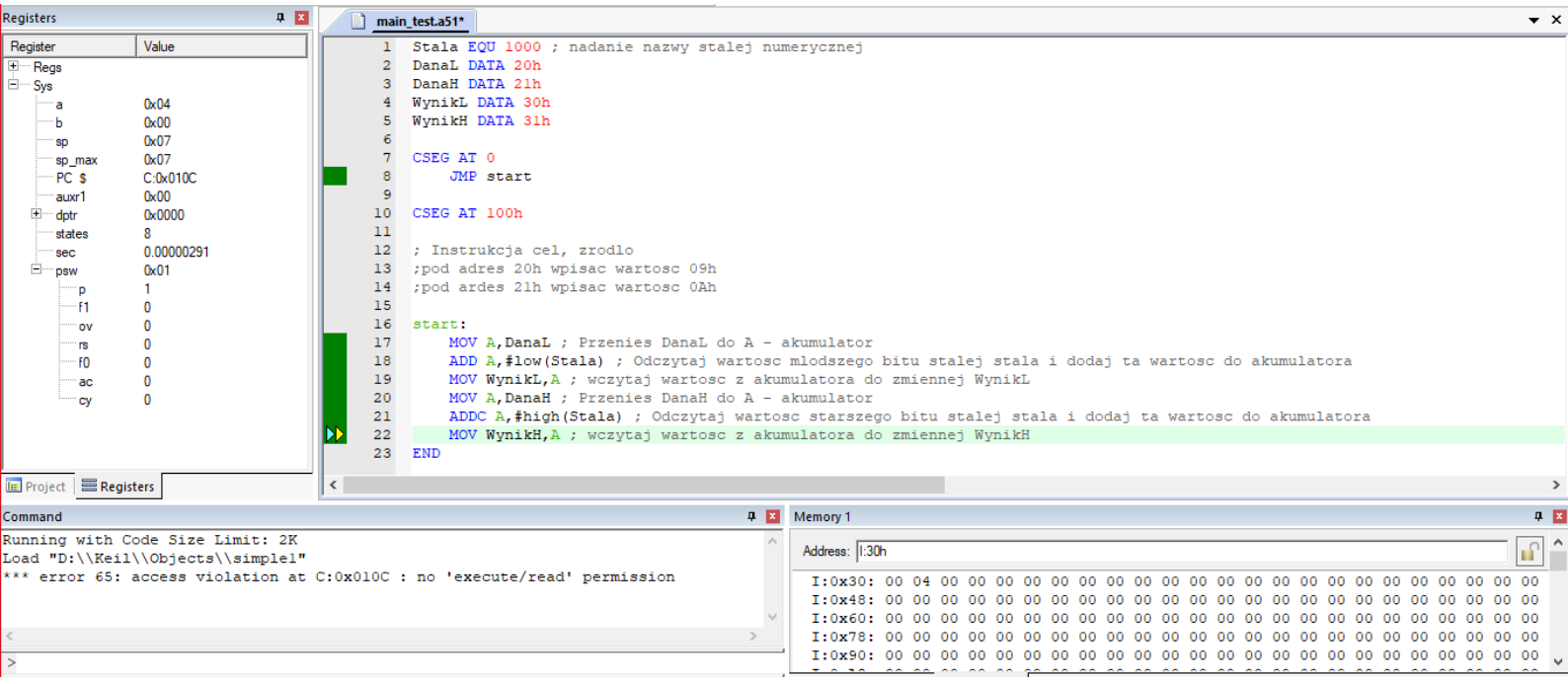
3. Wnioski

Liczby 16-bitowe są zapisywane w dwóch częściach (dwóch osobnych 8-bitowych komórkach pamięci).

4. Analiza zachowania programu dla przepełnienia



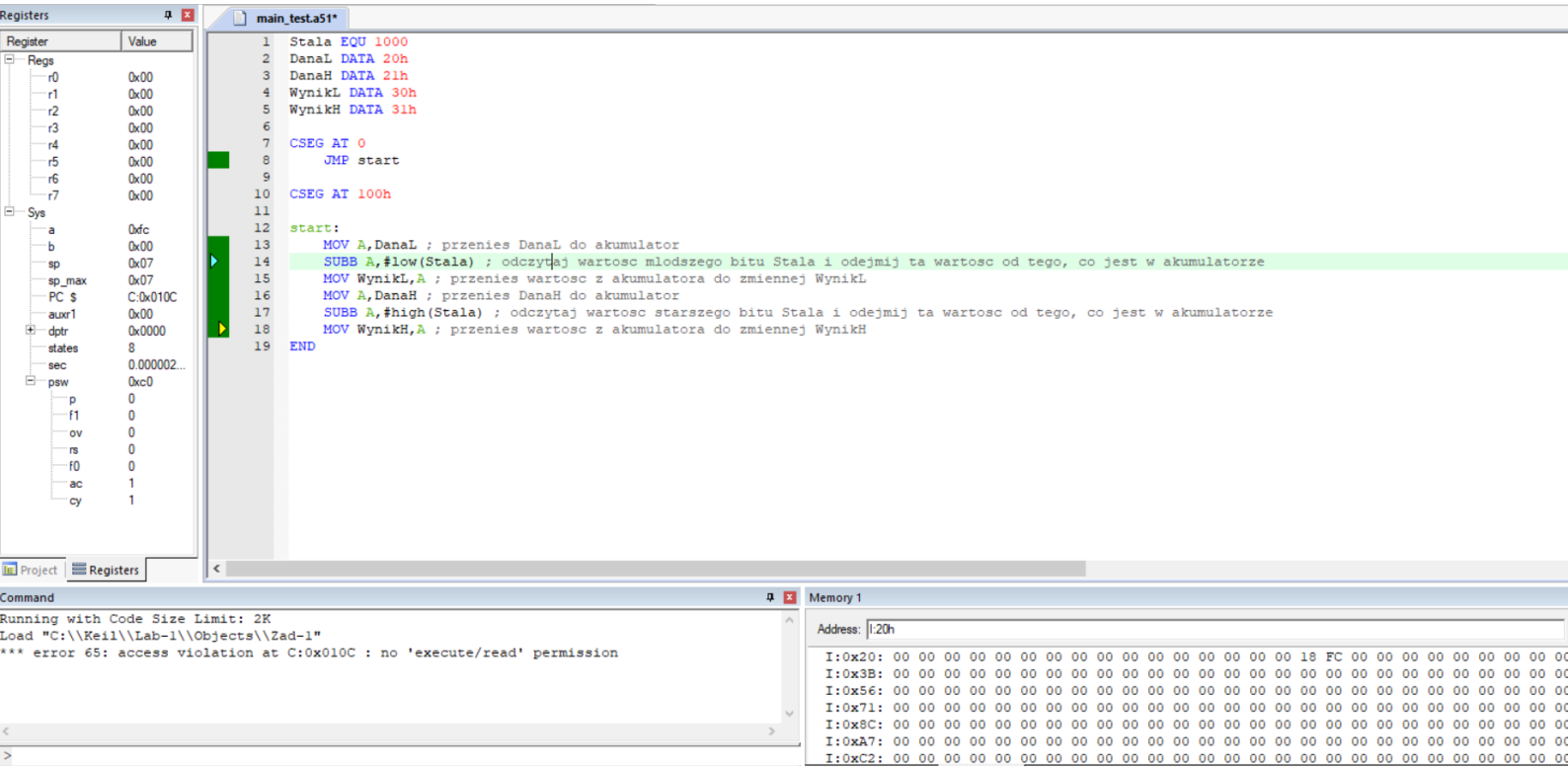
Rys.3 Dodawanie z przepełnieniem



Rys.4 Wynik dodawania z przeniesieniem

Ze względu na to, iż: E8 + 18 = 100 (a jedna komórka 8-bitowa, może pomieścić tylko 2 cyfrowe liczby), następuje przeniesienie wartości do wyższej komórki pamięci, co daje wynik 4 (3 + 1).

5. Odejmowanie liczb 16-bitowych

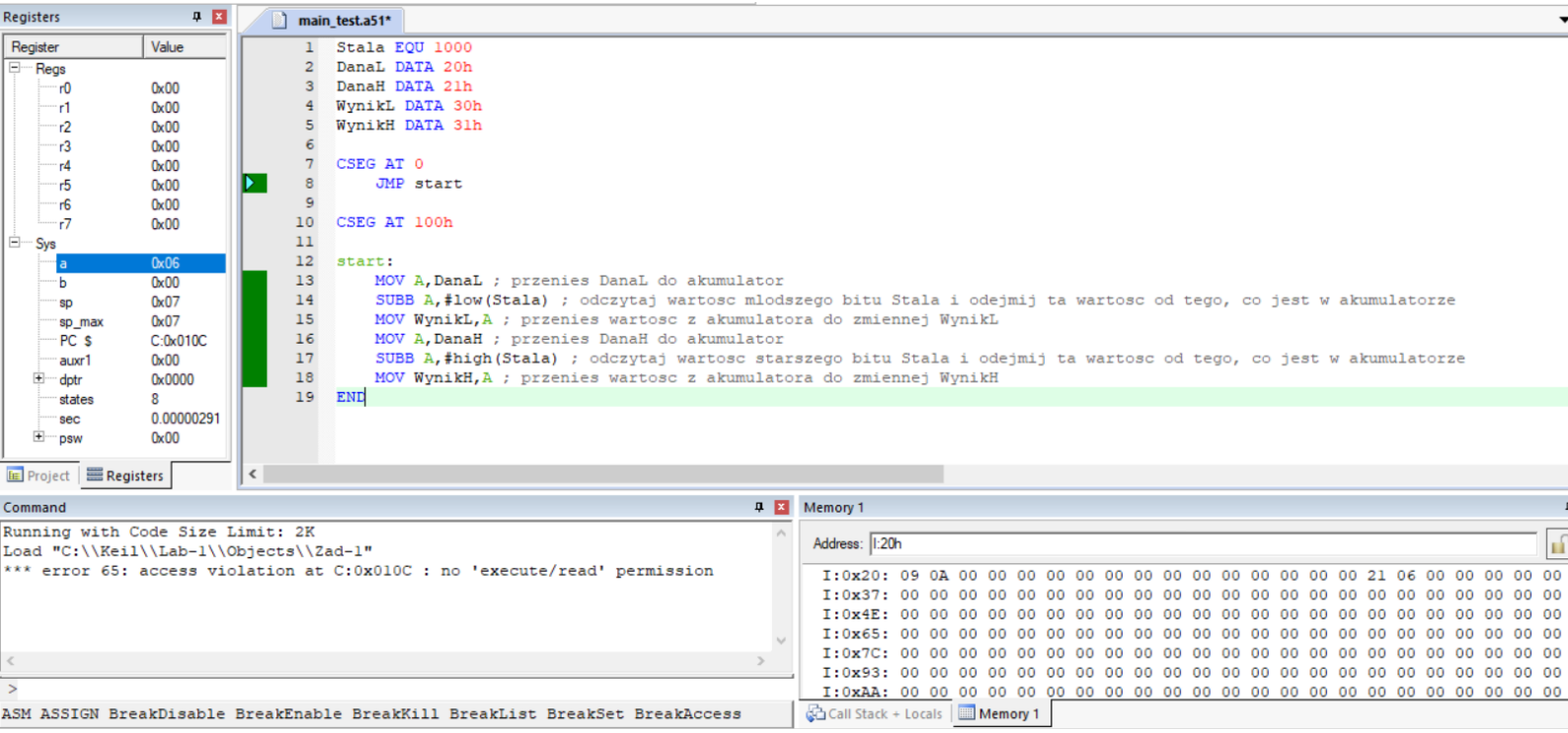


Rys. 5 Odejmowanie 16-bitowe

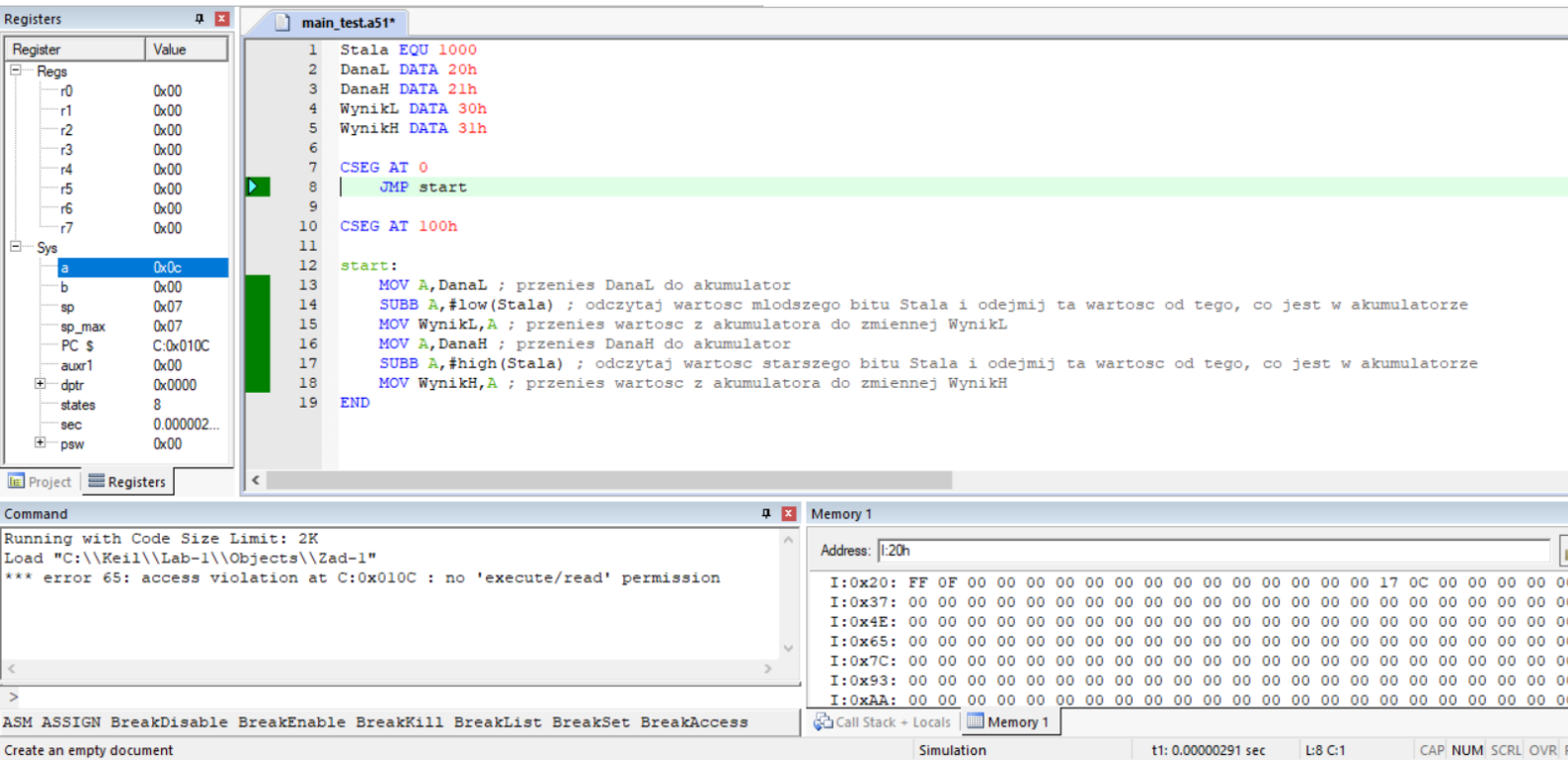
Powyższy przykład odejmuje najpierw mniej znaczący bajt liczby, zapisuje go do pamięci RAM, a następnie odejmuje z przeniesieniem starszy bajt, również zapisując wynik do RAM.

Jeśli wartość jest ujemna to dochodzi do zjawiska przepełnienia licznika, czyli na przykład:

0 – E8 = 18



Rys.6 Odejmowanie 16-bitowe z wartościami początkowymi 09h i 0Ah



Rys. 7 Odejmowanie 16-bitowe z wartościami początkowymi FFh i 0Fh

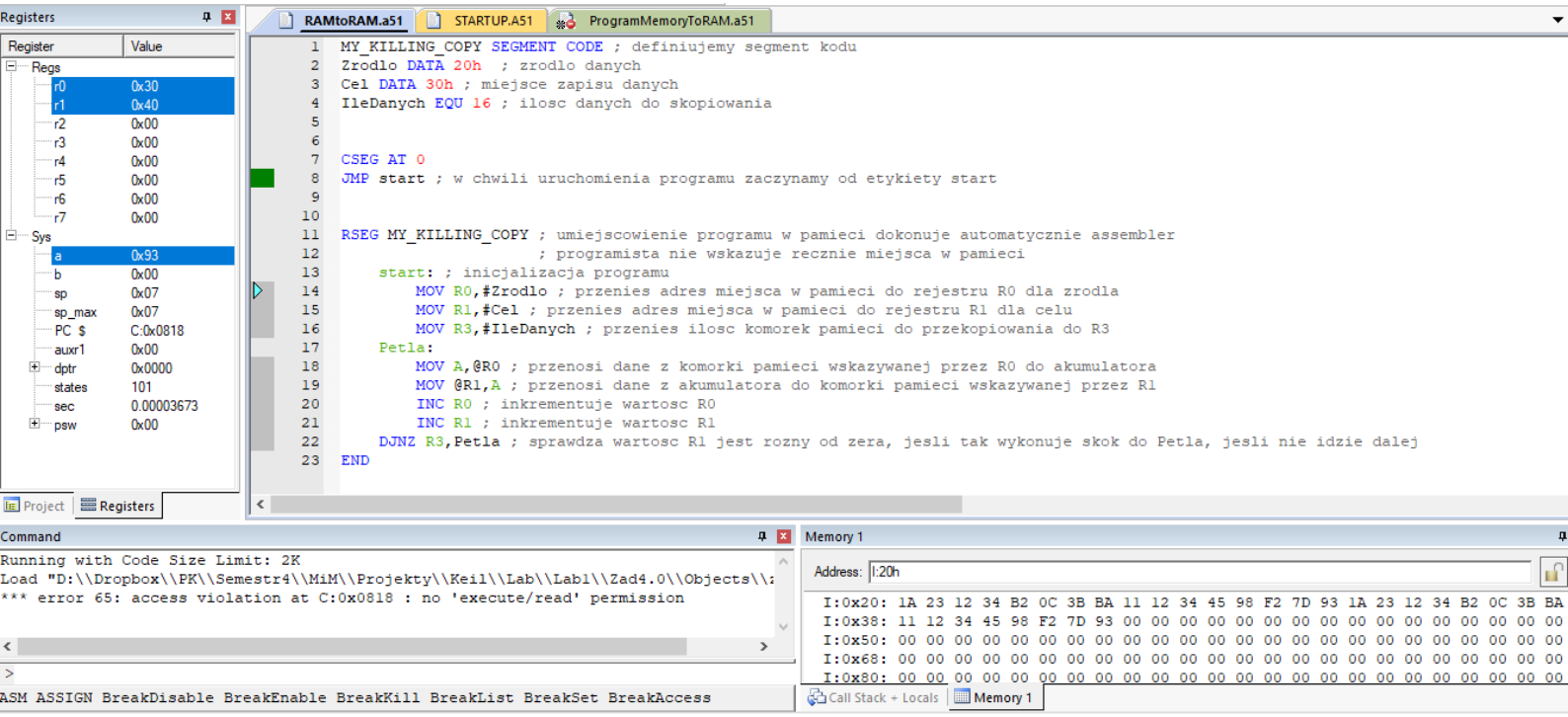
Zadanie 2

2.1. Z RAM do RAM:

Zadanie drugie polegało na przeanalizowaniu działania programu kopiującego dane z jednego obszaru pamięci RAM do innego.

```
MY_KILLING_COPY SEGMENT CODE ; definiujemy segment kodu
Zrodlo DATA 20h  ; zrodlo danych
Cel DATA 30h ; miejsce zapisu danych
IleDanych EQU 16 ; ilosc danych do skopiowania


CSEG AT 0
JMP start ; w chwili uruchomienia programu zaczynamy od etykiety start


RSEG MY_KILLING_COPY ; umiejscowienie programu w pamieci dokonuje automatycznie assembler
 ; programista nie wskazuje recznie miejsca w pamieci
start: ; inicjalizacja programu
MOV R0,#Zrodlo ; przenies adres miejsca w pamieci do rejestru R0 dla zrodla
MOV R1,#Cel ; przenies adres miejsca w pamieci do rejestru R1 dla celu
MOV R3,#IleDanych ; przenies ilosc komorek pamieci do przekopiowania do R3
Petla:
MOV A,@R0 ; przenosi dane z komorki pamieci wskazywanej przez R0 do akumulatora
MOV @R1,A ; przenosi dane z akumulatora do komorki pamieci wskazywanej przez R1
INC R0 ; inkrementuje wartosc R0
INC R1 ; inkrementuje wartosc R1
DJNZ R3,Petla ; sprawdza wartosc R1 jest rozny od zera, jesli tak wykonuje skok do Petla, jesli nie idzie dalej
END
```

Kod programu 2.1



Rys. 8 Program kopiujący komórki pamięci RAM

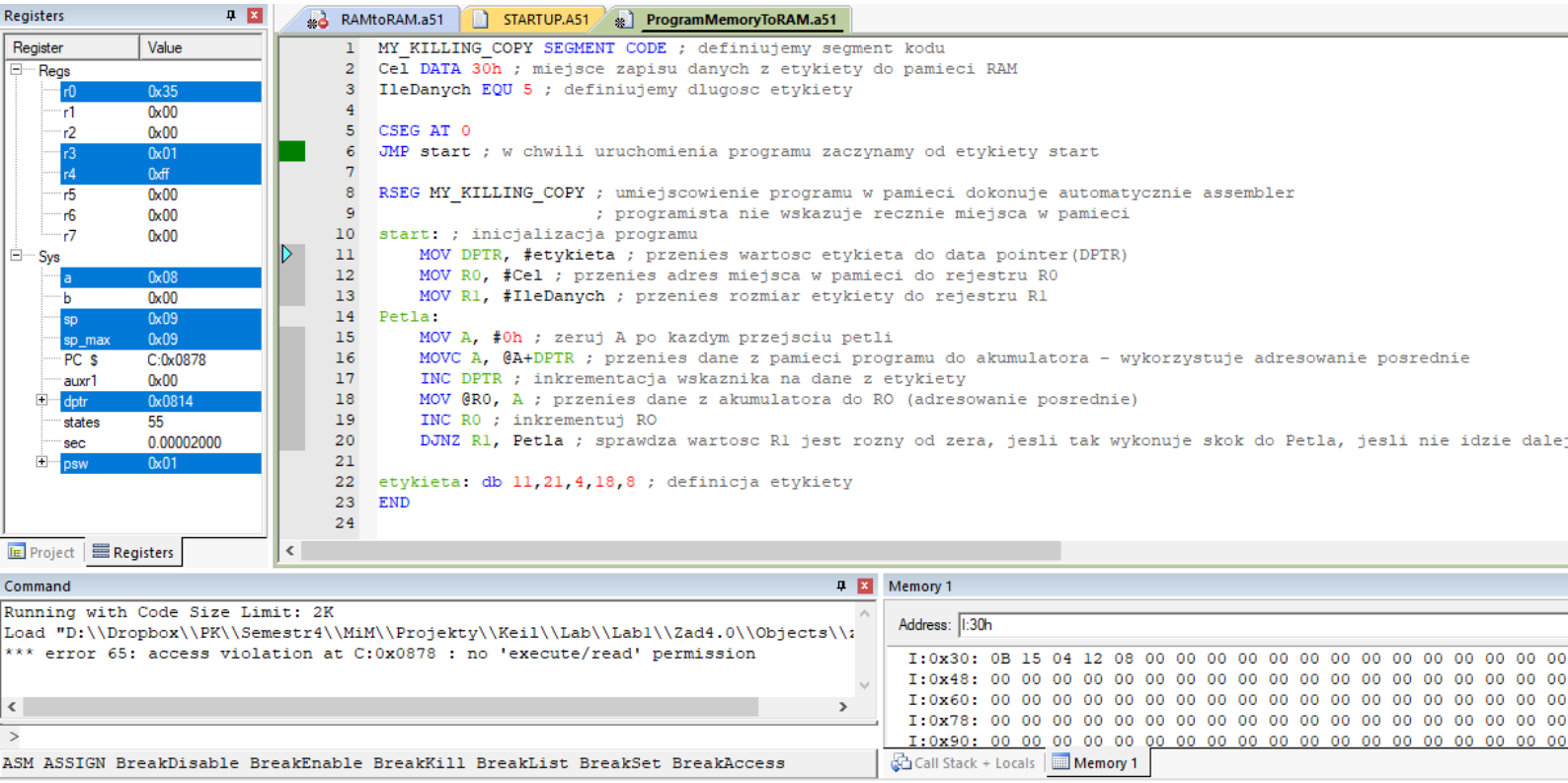
2.2 Z pamięci programu do RAM:

Następnie należało zmodyfikować program, aby dokonywał kopiowania z pamięci programu do pamięci RAM.

```
MY_KILLING_COPY SEGMENT CODE ; definiujemy segment kodu
Cel DATA 30h ; miejsce zapisu danych z etykiety do pamieci RAM
IleDanych EQU 5 ; definiujemy dlugosc etykiety

CSEG AT 0
JMP start ; w chwili uruchomienia programu zaczynamy od etykiety start

RSEG MY_KILLING_COPY ; umiejscowienie programu w pamieci dokonuje automatycznie assembler
 ; programista nie wskazuje recznie miejsca w pamieci
start: ; inicjalizacja programu
MOV DPTR, #etykieta ; przenies wartosc etykieta do data pointer(DPTR)
MOV R0, #Cel ; przenies adres miejsca w pamieci do rejestru R0
MOV R1, #IleDanych ; przenies rozmiar etykiety do rejestru R1
Petla:
MOV A, #0h ; zeruj A po kazdym przejsciu petli
MOVC A, @A+DPTR ; przenies dane z pamieci programu do akumulatora
                                ;- wykorzystuje adresowanie posrednie
INC DPTR ; inkrementacja wskaznika na dane z etykiety
MOV @R0, A ; przenies dane z akumulatora do RO (adresowanie posrednie)
INC R0 ; inkrementuj RO
DJNZ R1, Petla ; sprawdza wartosc R1 jest rozny od zera, jesli tak wykonuje skok do Petla,
                            ; jesli nie idzie dalej

etykieta: db 11,21,4,18,8 ; definicja etykiety
END
```

Kod programu 2.2

Powyższy przykład wykorzystuje **adresowanie pośrednie [@]. MOVC** umożliwia kopiowanie danych z pamięci programu do akumulatora. Rejestr **R0** wskazuje na miejsce zapisu w pamięci RAM. Jest on inkrementowany każdorazowo, po przekopiowaniu każdego bajtu. Rejestr **R1** wskazuje na to, ile danych należy kopiować. **DJNZ** dekrementuje **R1** i jeśli nie wyniesie ona zero, to skacze do etykiety **PETLA**. Należy też zwrócić uwagę na wskaźnik DPTR, który należy ręcznie inkrementować, aby odczytać kolejne elementy w **etykiety**.



Rys. 9 Program kopiujący dane z pamięci programu do RAM

Zadanie 3

1. Opis zadania

Zadanie polegało na napisaniu generatora liczb pseudolosowych w oparciu o opóźniony ciąg Fibonacciego.

2. Teoria

2.1. Generator liczb pseudolosowych (*ang. PRNG - Pseudo-Random Number Generator*) pozwala na generowanie ciągu liczb, który:

- jest deterministyczny - zainicjowany tą samą wartością dają zawsze taki sam ciąg pseudolosowych liczb;
- pod pewnymi względami jest nieodróżnialny od ciągu uzyskanego z prawdziwie losowego źródła.

2.2. Standardowy ciąg Fibonacciego definiowany jest jako:

$$x_n = x_{n-1} + x_{n-2}$$

A więc kolejne wyrazy ciągu powstają poprzez dodanie dwóch poprzednich. Przy czym zakłada się, że dwa początkowe wyrazy wynoszą 1.

2.3. Uogólniony wzór wygląda następująco:

$$x_n = (x_{n-j} \,@\, x_{n-k}) \bmod m,\ 0 < j < k$$

Uzyskamy w ten sposób Opóźniony Generator Fibonacciego (*ang. LFG - Lagged Fibonacci Generator*). Znaczenie poszczególnych symboli jest następujące:

- x_n: *n*-ta wygenerowana liczba pseudolosowa;
- *m:* współczynnik określający zakres generowanych liczb pseudolosowych (od 0 do *m-1*).

Zmianie, w stosunku do oryginalnej definicji ciągu, uległo przede wszystkim wprowadzenie dzielenia modulo *m*, które zapewnia nam generowanie liczb w zakresie *<0; m-1>*. Uogólniono też operację wykonywaną na dwóch wyrazach ciągu, @ może być zdefiniowane jako np. dodawanie, odejmowanie, mnożenie. Zmieniono też definicję wyrazów ciągu poddawanych operacji - nie są to już dwa bezpośrednio poprzednie wyrazy.

2.4. Zgodnie z poleceniem, wybraną operacją jest dodawanie:

- ALFG - Addytywny Opóźniony Generator Fibonacciego (*ang. Additive Lagged Fibonacci Generator*). Otrzymamy wówczas następujący wzór:

$$x_n = (x_{n-j} + x_{n-k}) \bmod m,\ 0 < j < k$$

Generator ten musi pamiętać *k* ostatnich wygenerowanych słów. W praktyce wykonuje się to poprzez zastosowanie tablicy o *k* elementach. Tablicę taką traktuje się jak bufor cykliczny, tzn. iterując po kolejnych elementach, a dochodząc do końca tablicy przechodzimy z powrotem na jej początek. Łączymy niejako koniec i początek tej tablicy, tworząc cykl. Zatem odwołanie do elementu *n-j*, dla *k*-elementowej tablicy będzie pod indeksem: ***(n-j+k) mod k***. Kolejne wywołania funkcji obliczają wartości w tablicy od indeksu 0 do *k-1* po czym znów zaczynają obliczać wartości dla indeksu 0. Początkowo tablica ta musi zostać zainicjowana jakimiś losowymi wartościami.

2.5. W pseudokodzie cały algorytm można przedstawić następująco. Niech *x* będzie wspomnianą tablicą, natomiast *i* będzie wewnętrznym wskaźnikiem określającym aktualne przesunięcie w tablicy:

```
x[i] = (x[(k + i - j) mod k] @ x[i]) mod m;
result = x[i];
i = (i + 1) mod k;
```

2.6. Należy jeszcze ustalić wartości *j* oraz *k*.

By generator miał maksymalny możliwy cykl, wartości te należy dobrać tak by wielomian: $x^k + x^j + 1$ był wielomianem prostym. Poniżej w tabeli zestawiono popularne wartości tych współczynników:

j	**k**
5	17
6	31
7	10
24	55
31	63
65	71
97	127
128	159
168	521
273	607
334	607
353	521
418	1279

3. Założenia:

- (j, k) = (7, 10);
- Operacja - @ - dodawanie;
- M = 4 – generuje liczby z przedziału od 0 do 3;
- Tablica startowa = 4, 5, 3, 3, 3, 4, 6, 2, 6, 5 - powinna być zapisana do pamięci programu i skopiowana do pamięci RAM, przez program inicjalizujący (w postaci bufora cyklicznego);
- Procedura generująca liczby powinna zwracać liczbę pseudolosową do rejestru A;
- Wynik losowania powinien być zapisany do bufora cyklicznego.

4. Kod programu:

```
j EQU 7 ; definujemy stala j
kSize EQU 10 ; definiujemy stala k
m EQU 4 ; <0,3> ; definiujemy stala m odpowiedzialna za zakres generowanych liczb
indexModuloStart EQU 3 ; definiujemy pierwsza wartosc indeksu

ORG 0020H ; ustaw miejsce w kodzie na 20h
k0 EQU 4 ; definujemy wartosci poczatkowe tablicy
k1 EQU 5
k2 EQU 3
k3 EQU 3
k4 EQU 3
k5 EQU 4
k6 EQU 6
k7 EQU 2
k8 EQU 6
k9 EQU 5

poczatekTablicy DATA 20h ; definjemy poczatek tablicy
koniecTablicy DATA 29h ; definiujemy koniec tablicy
first_element DATA 32h ; definujmey pierwszy element dla wartosci modulo
second_element DATA 33h ; definujmey drugi element dla wartosci modulo
result DATA 34h
Modulo1 DATA 40h ; pierwsza czesc wyrazenia modulo
Modulo2 DATA 41h ; druga czesc wyrazenia modulo
indexModulo DATA 42h
Quotient DATA 50h ; czesc calkowita z dzielenia
Remainder DATA 51h ; reszta z dzielenia

INIT_GEN: ; wpisanie wartosci tablicy do pamieci RAM
MOV poczatekTablicy, #k0
MOV 21h, #k1
MOV 22h, #k2
MOV 23h, #k3
MOV 24h, #k4
MOV 25h, #k5
MOV 26h, #k6
MOV 27h, #k7
MOV 28h, #k8
MOV koniecTablicy, #k9

ACALL RESET_INDEXES
RET

MODULO:
MOV A, Modulo1 ; prznies pierwszy argument z Modulo1 do A
MOV B, Modulo2 ; prznies pierwszy argument z Modulo1 do B

DIV AB ; Dziel A przez B

MOV Quotient, A; Zapisz czesc calkowita do komorki RAM 50H
MOV Remainder, B; Zapisz reszte do komorki RAM 51H

RET ; powrot z funkcji

INDEKS_1:
MOV Modulo1, indexModulo ; 3 = pierwotna wartosc indeksu dla k=10, prznies z indexModulo do Modulo1
MOV Modulo2, #kSize ; mod k, gdzie k = 10; prznies wartosc kSize do Modulo2
```

```
ACALL MODULO ; wywolanie funkcji modulo

MOV A, Remainder ; zapisz reszte z dzielenia (wynik funkcji modulo) do A
ADD A, #20h ; dodaj 20h do akumulatora by uzyskac indeks
MOV R0, A ; zapisz wartosc z akumulatora do rejestru R0

RET ; powrot z funkcji


GET_FROM_ARRAY:
MOV first_element, @R0 ; pobierz pierwsza skladowa do pamieci RAM (adresowanie posrednie)
MOV second_element, @R1 ; pobierz druga skladowa do pamieci RAM (adresowanie posrednie)

RET ; powrot z funkcji

SUM: ; czy moze wystapic przepelnienie?
MOV A, first_element ; przenies pierwszy element do akumulatora
ADD A, second_element ; dodaj drugi element do akumulatora
MOV result, A ; przenies wartosc z akumulatora do pamieci RAM

RET ; powrot z funkcji

INCREMENT:
INC R1 ; podnies o 1 wartosc R1 (Indeks)
INC indexModulo ; podnies o 1 wartosc  indexModulo

RET ; powrot z funkcji

CALCULATE_RANDOM:
MOV Modulo1, result ; przenies result do Modulo1
MOV Modulo2, #m ; przenies wartosc parametru m do Modulo2

ACALL MODULO ; wywoluje funkcje modulo
MOV A, B ; przenies wynik funkcji modulo (szukana liczba losowa) do akumulatora

MOV @R1, A ; zapisz wartosc A do rejestru cyklicznego pod wartosc indeksu wskazywanego przez R1 (adresowanie
posrednie)

CJNE R1, #koniecTablicy, INCREMENT ; sprawdz czy indeks osiagnal koniecTablicy,
  ; jesli tak idz dalej, jesli nie wywolaj funkcje INCREMENT
SJMP RESET_INDEXES ; idz do funkcji RESET_INDEKS

RET ; powrot z funkcji

RESET_INDEXES:
MOV indexModulo, #indexModuloStart ; przenies warosc 3 do indexModulo
MOV R1, #20h ; ustaw 20 jako wartosc rejestru R1 (poczatek tablicy)

RET ; powrot z funkcji

CSEG AT 0
ACALL INIT_GEN ; inicjalizuj generator 10 startowymi liczbami
LJMP main ; skok do glownego programu

CSEG AT 0x100
main:
ACALL INDEKS_1 ; oblicz indeks i-j+k
ACALL GET_FROM_ARRAY ; pobierz wartosci z rejestru cyklicznego
ACALL SUM ; dodaj dwie wartosci pobrane z rejestru cyklicznego
ACALL CALCULATE_RANDOM ; oblicza wartosc losowa i przenosi ja do akumulatora
```

```
NOP
AJMP main
END
```

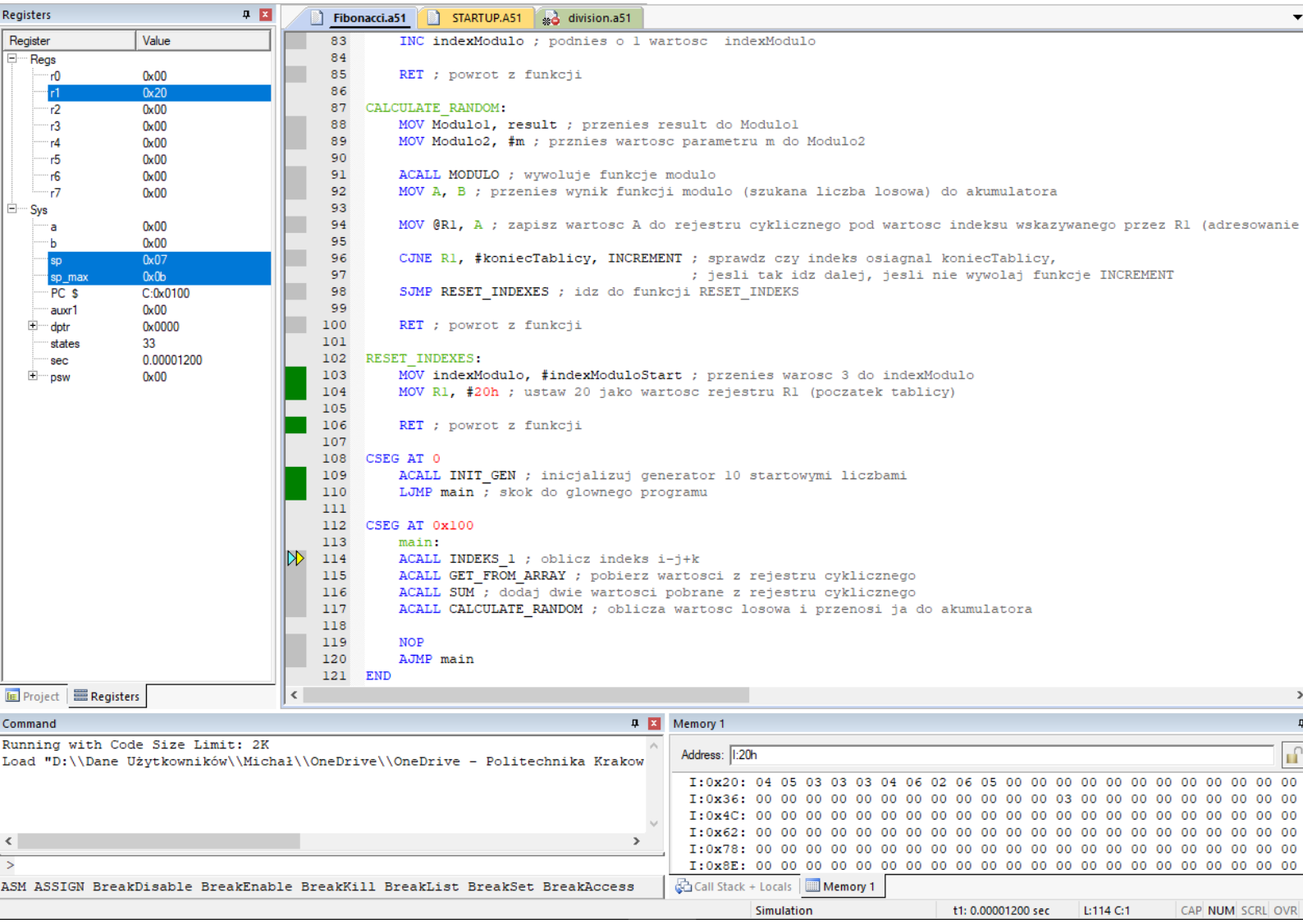
Kod programu 3

Powyższy kod przedstawia generator liczb pseudolosowych w Assemblerze dla platformy Intel 8051.

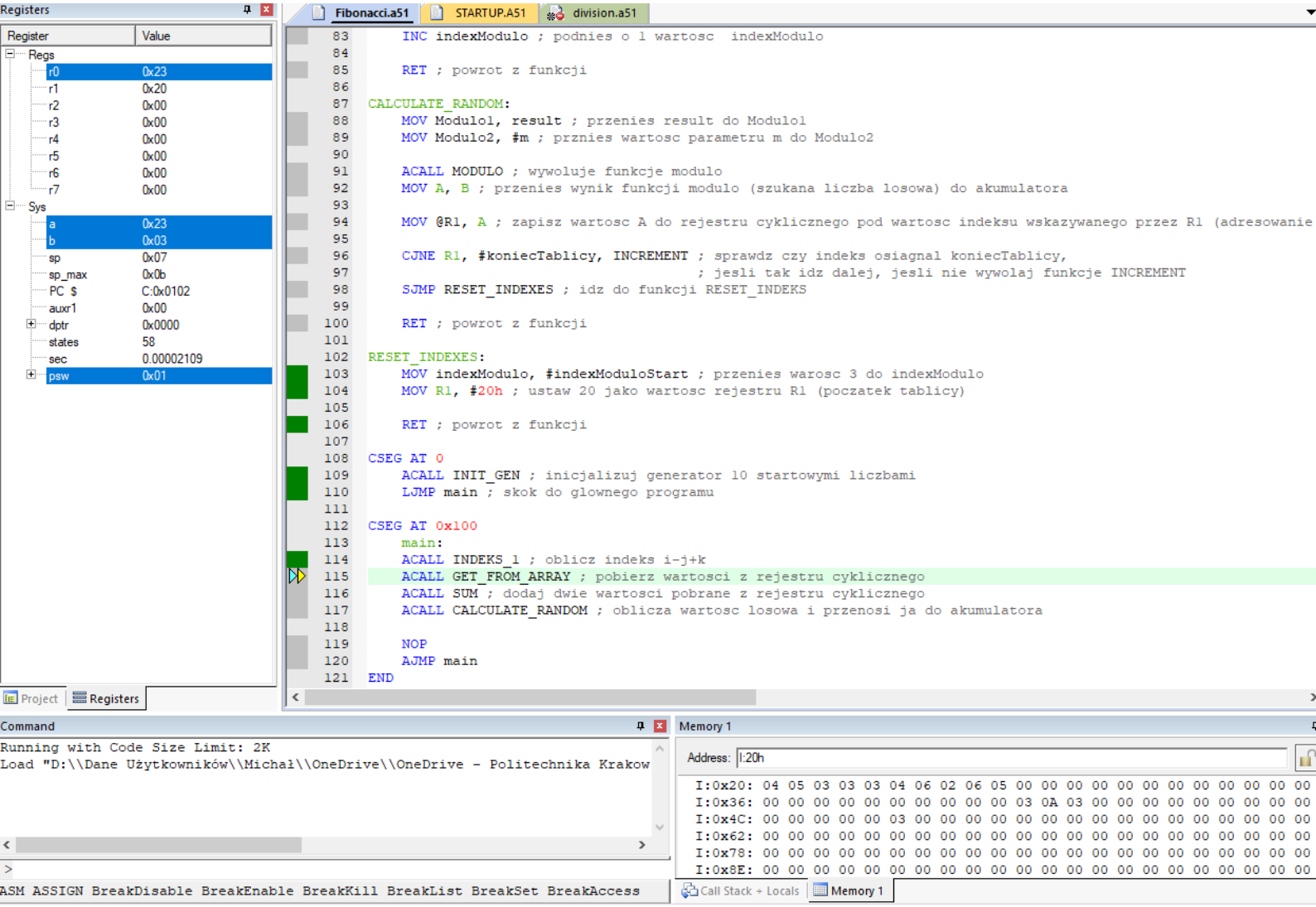
5. Opis implementacji algorytmu:

1. Na początku programu definiujemy stałe w pamięci aplikacji, wraz z wartościami początkowymi tablicy. Następnie definiujemy miejsca w pamięci RAM dla obliczeń.
2. Funkcja INIT_GEN służy do inicjalizacji tablicy początkowymi wartościami.
3. Początkowa wartość k + j –n, dla k = 10; j = 7; n = 0, wynosi 3 i jest wpisywana jako wartość *indexModulo.*
4. W funkcji MODULO wykorzystano operację DIV. Wynik zapisywany jest do komórki pamięci Remainder (51H).
5. Aby uzyskać pierwszą wartość indeksu, k+j-n mod k, użyto funkcji INDEKS_1, w wyniku której indeks jest zapisywany do rejestru R0.
6. Funkcja GET_FROM_ARRAY pobiera z bufora cyklicznego wartości według indeksów zapisanych w Rejestrach R0 i R1.
7. Funkcja CALCULATE_RANDOM oblicza docelową liczbę pseudolosową i zapisuje ją w rejestrze A. Następnie podmienia liczbę w rejestrze cyklicznym, według indeksu zapisanego w R1. Kolejno następuje sprawdzenie, czy indeks wskazuje na koniec tablicy. Jeśli tak, funkcja RESET_INDEXES ustawia wartości R0 i R1 na startowe, jeśli nie, wywoływana jest funkcja INCREMENT, która zwiększa o jeden Rejestr R1 oraz komórkę pamięci *indexModulo*.

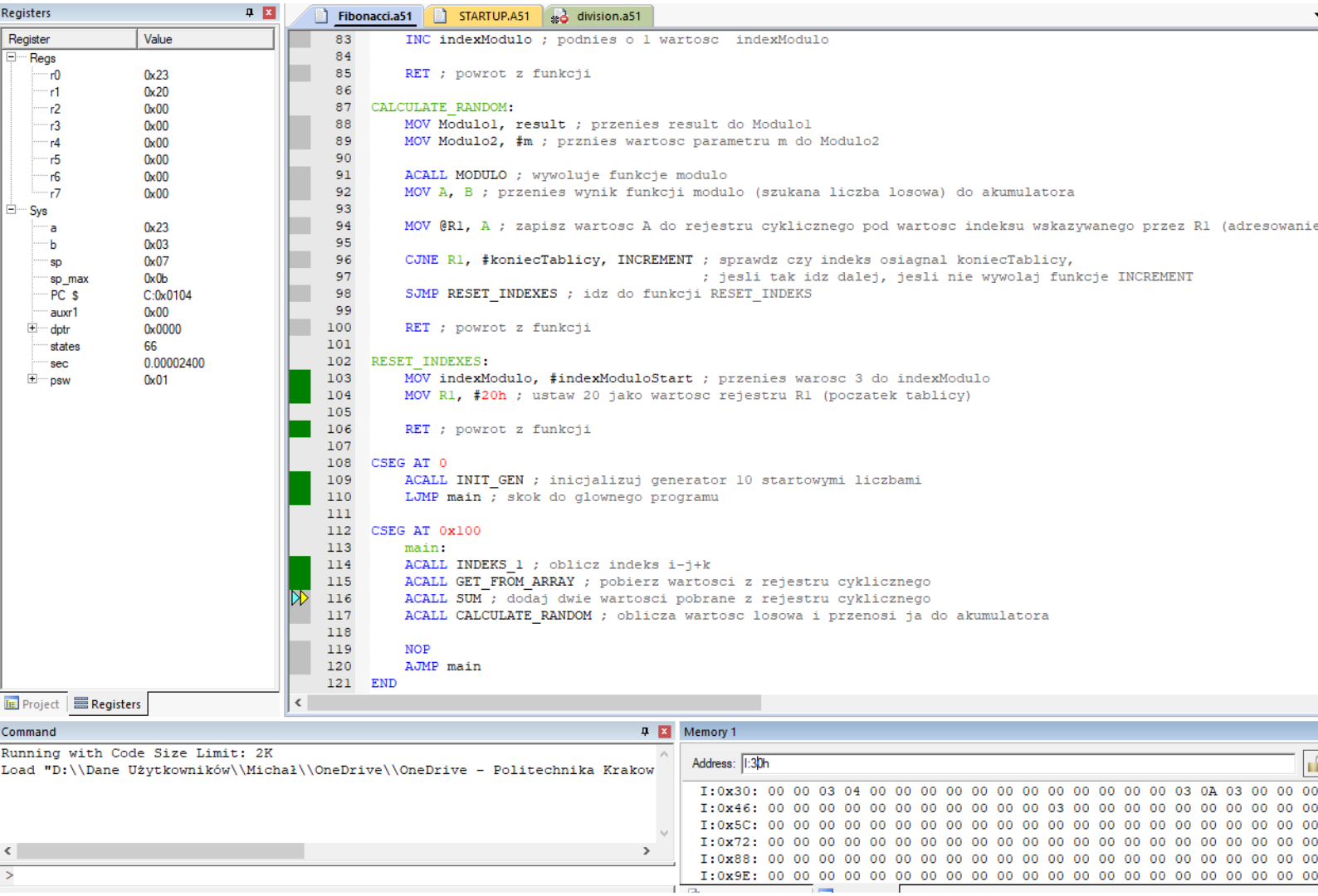
6. Demonstracja działania programu:



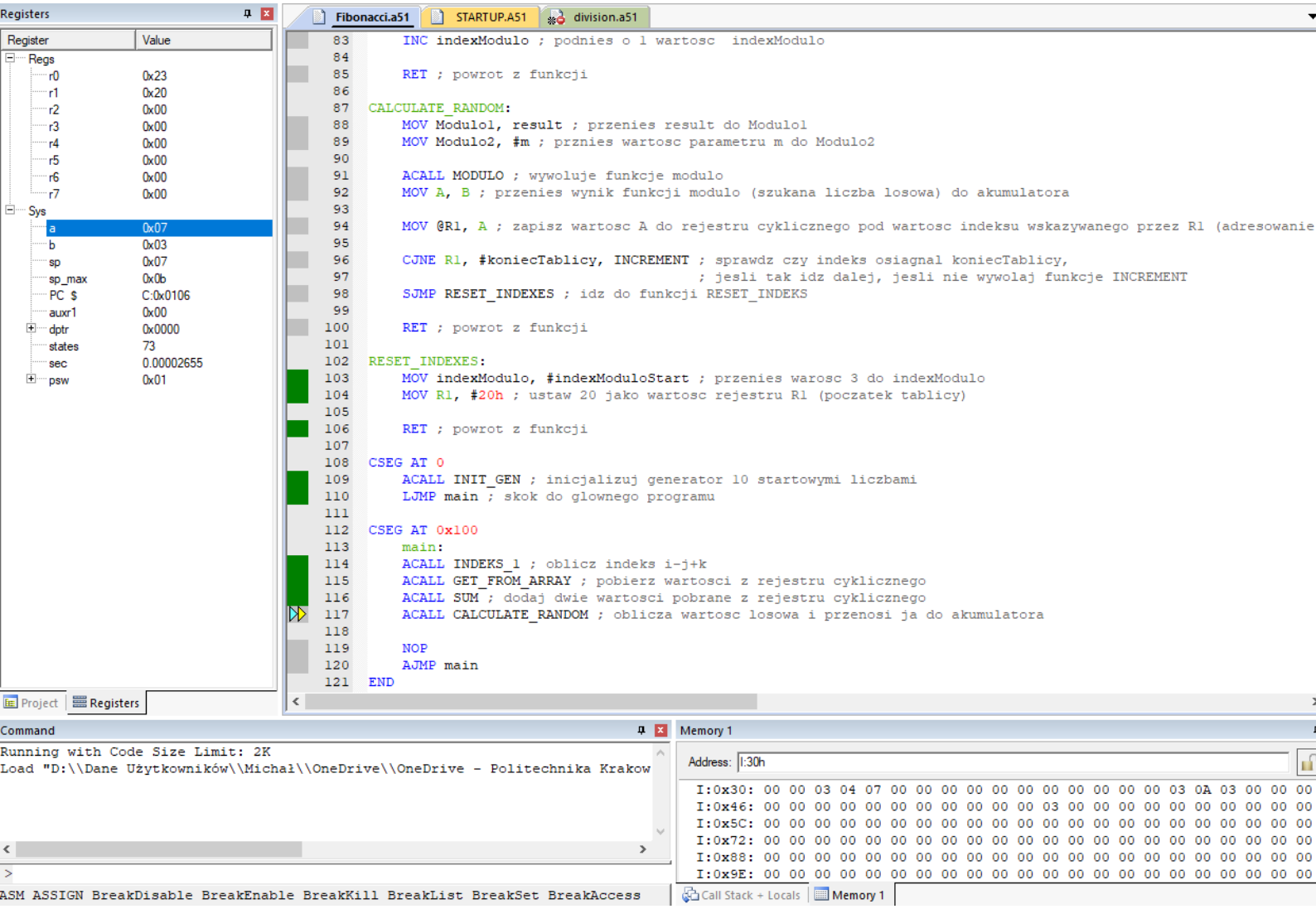
Rys. 10 Zapis tablicy do pamięci RAM



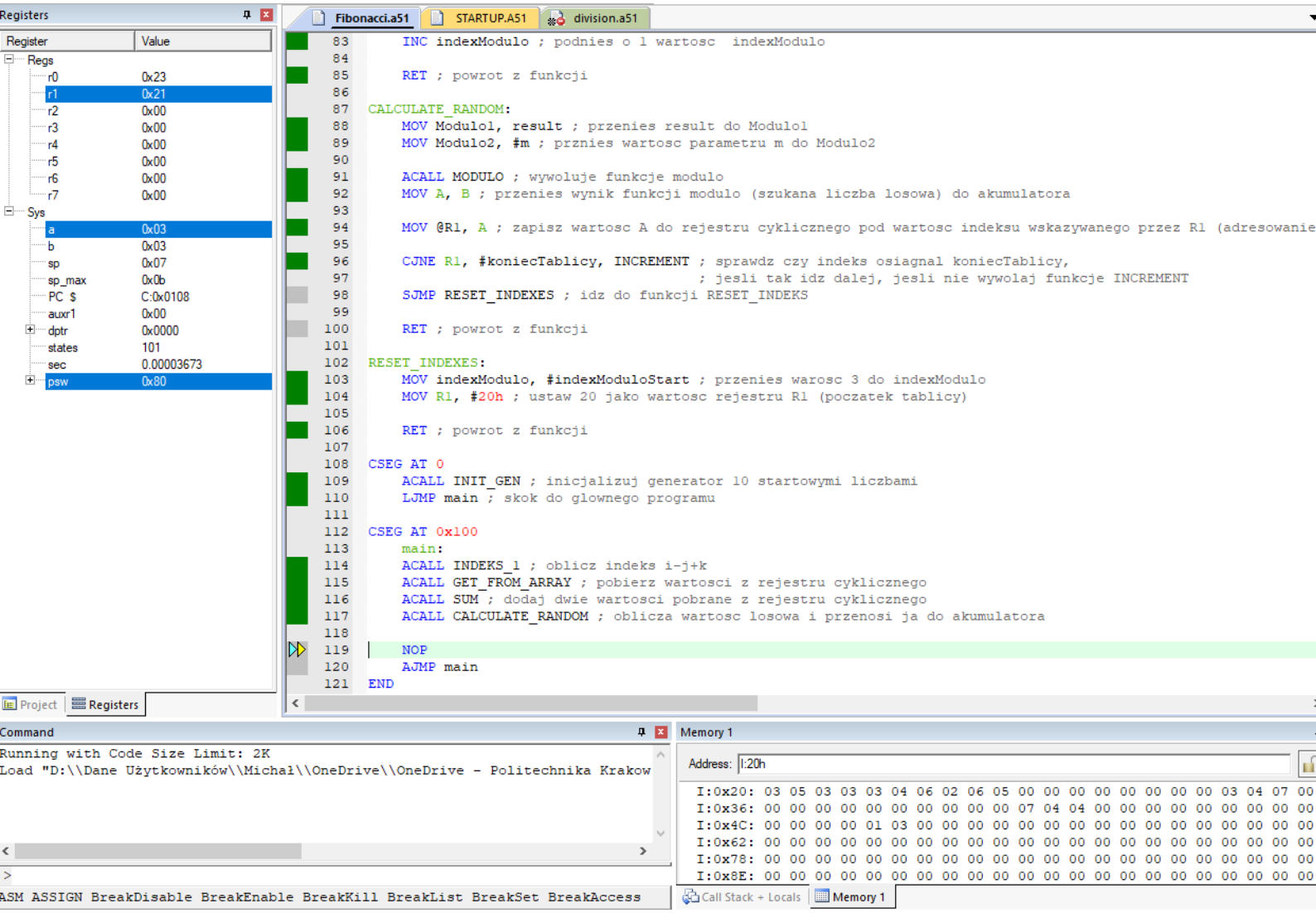
Rys. 11 Obliczenie Indeksu i-j+k



Rys. 12 Pobieranie wartości z rejestru cyklicznego



Rys. 13 Dodawanie dwóch składowych do obliczenia docelowej liczby pseudolosowej



Rys. 14 Obliczenie docelowej liczby i zapis jej do akumulatora oraz bufora cyklicznego

7. Podsumowanie i wnioski

Pierwsze ćwiczenie miało zaznajomić nas z assemblerem **8051**. Na kilku ćwiczeniach poznaliśmy, jak funkcjonuje środowisko **uVision** oraz w jaki sposób działają mikrokontrolery.