Gdańsk, 26.01.2006

Zastosowanie procesorów sygnałowych

Projekt

Przykładowe programy asemblerowe do zestawu firmy ANALOG DEVICES

1.Cel projektu.

Głównym zagadnieniem związanym z realizacją projektu było zapoznanie się z architekturą procesora SHARC firmy ANALOG DEVICES i stworzenie na tej podstawie kilku przykładowych wstawek asemblerowych demonstrujących wykorzystanie typowych rozkazów asemblera tego procesora.

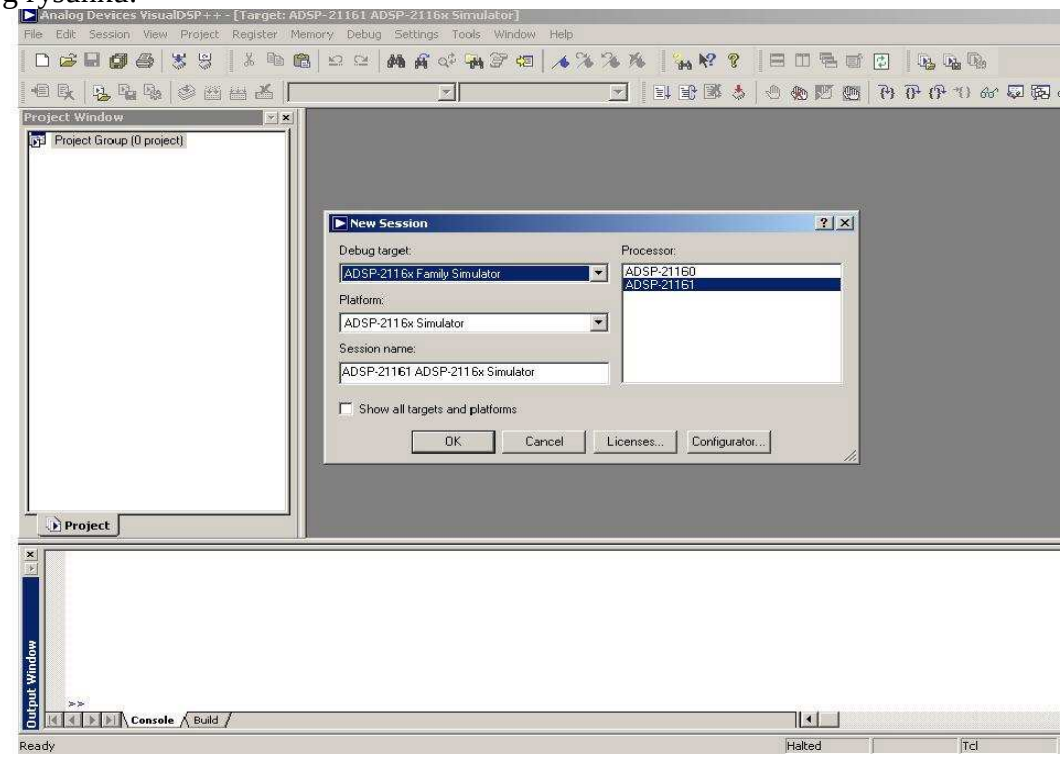
Prace prowadzone były w symulatorze pakietu *VisualDSP ++* firmy Analog Devices. Poniżej zostaną przedstawione przykładowe programy.

2.Przykładowe programy.

Aby rozpocząć pracę z programem należy z menu *Session* wybrać *New Session* i następnie w otwartym oknie skonfigurować opcje:

- Debug Target: ADSP-2116x Family Simulator
- Platform: ADSP-2116x Simulator
- Processor: ADSP-21161

wg rysunku:



Powyższa konfiguracja pakietu umożliwia pracę bez konieczności podłączania zewnętrznego układu z procesorem SHARC.

W skład przykładowych programów wchodzą:

- Potęgowanie poprzez podanie podstawy i wykładnika potęgi
- Obliczanie transpozycji zadanej macierzy
- Sortowanie elementów zadanego wektora

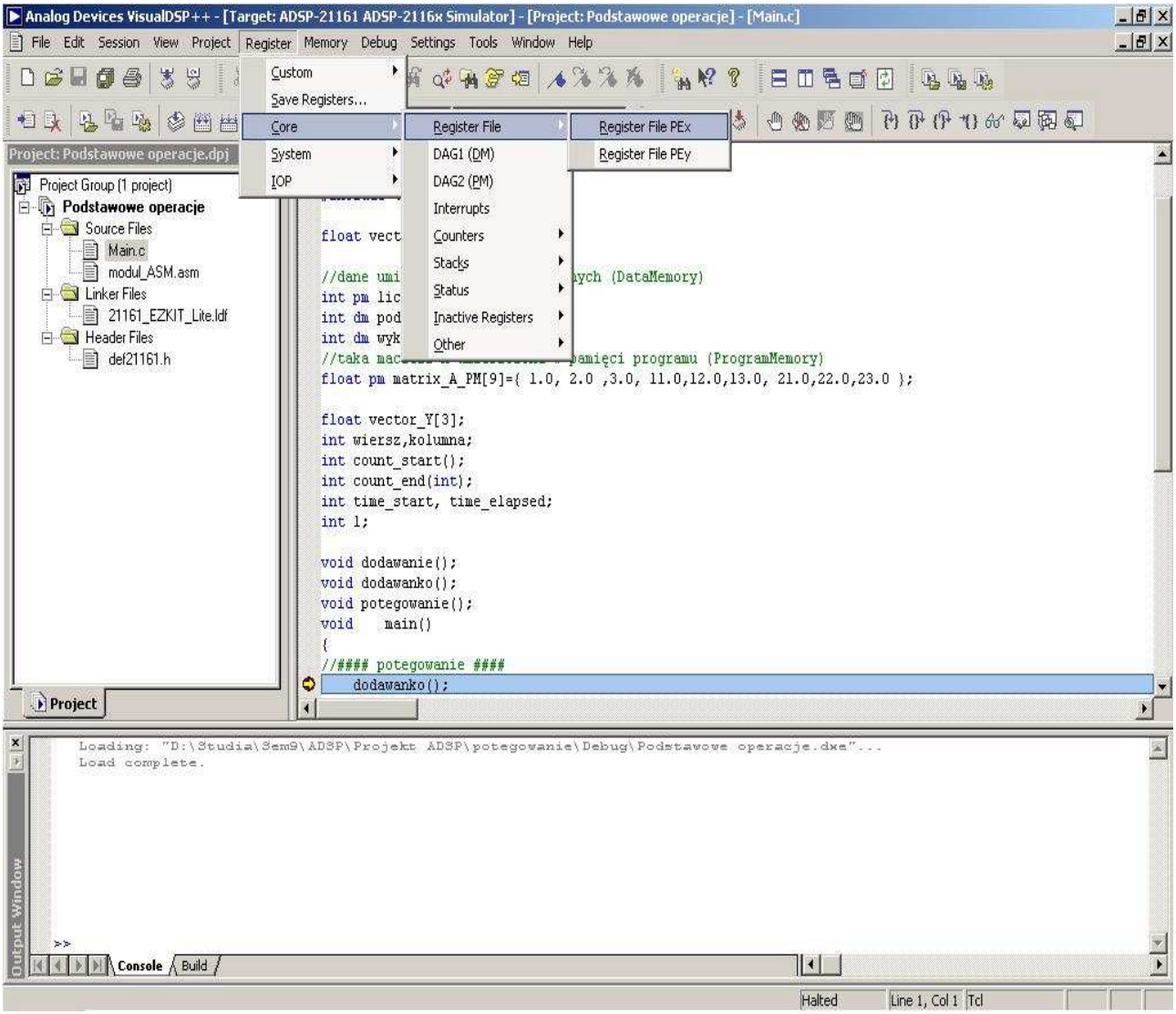
Zadanie 1.

Należy otworzyć projekt *Podstawowe operacje.dpj* (folder Potęgowanie) z menu File->Open->Projekt… i przeanalizować kod programu napisany w asemblerze *modul_ASM.asm* oraz plik *main.c* , który służy do demonstracji na ekranie terminala wyników działania programu asemblerowego.

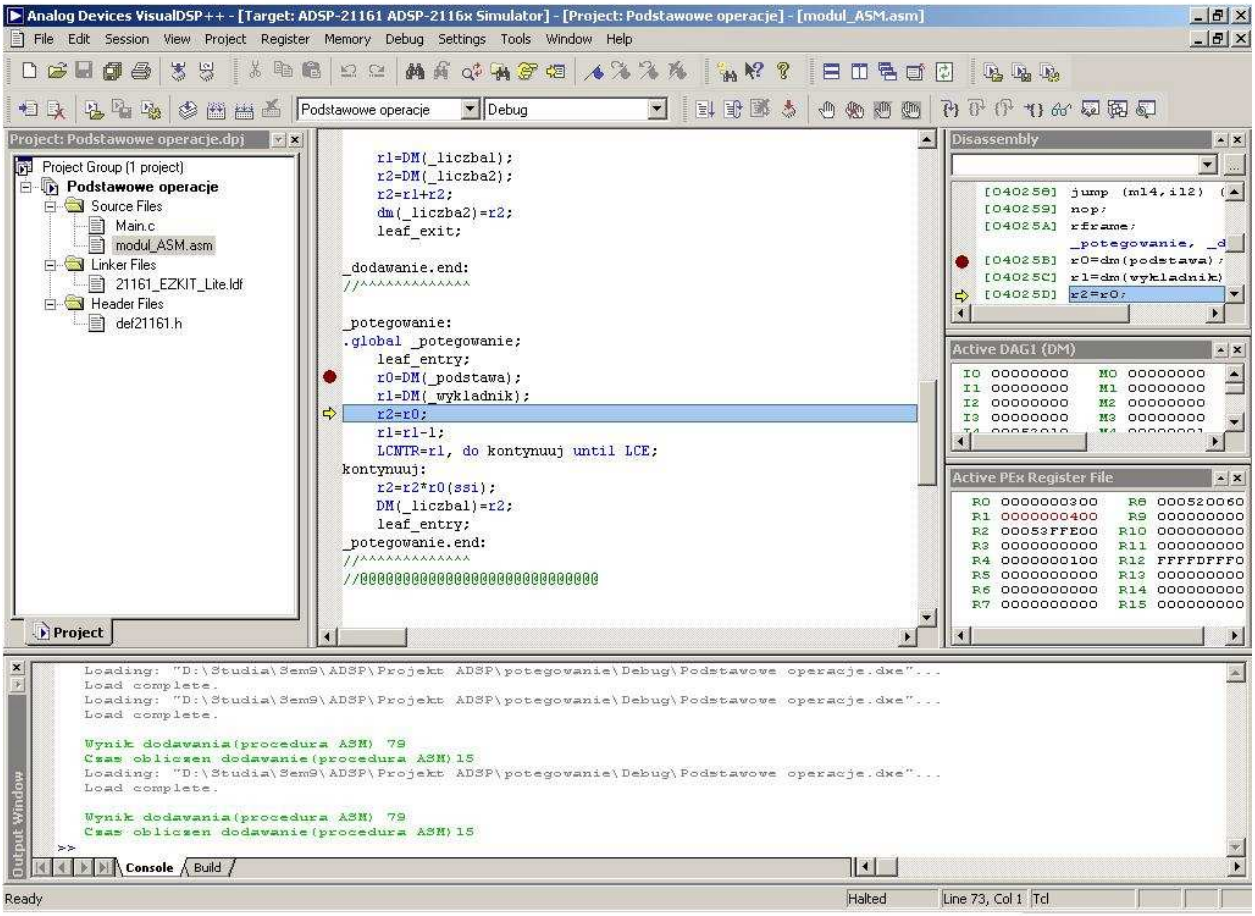
Funkcja napisana w asemblerze realizująca potęgowanie:

```
                _potegowanie:
                .global _potegowanie;
                leaf_entry;
                r0=DM(_podstawa);       //przesłanie do rejestru r0 danej z
                            pamięci danych zaadresowanej etykietą podstawa

                r1=DM(_wykladnik);      //przesłanie do rejestru r1 danej z
                            pamięci danych zaadresowanej etykietą wykladnik
                r2=r0;
                r1=r1-1;
                LCNTR=r1, do kontynuuj until LCE;
                                        //pętla wykonująca r1-krotnie fragment
                                        programu zaadresowany etykietą
                                        kontynuuj
   kontynuuj:
                r2=r2*r0(ssi);
                DM(_liczba1)=r2;        //przesłanie danej z r2 do zmiennej
                                        oznaczonej etykietą liczba1
                leaf_entry;
                _potegowanie.end:
```

Po wstępnej analizie należy skompilować projekt  i uruchomić  i zaobserwować na ekranie terminala wyniki. W celu lepszego zrozumienia działania programu można uruchomić projekt w pracy krokowej i obserwować zmiany w poszczególnych rejestrach procesora. Aby zrealizować tą opcję należy wybrać odpowiednie rejestry procesora, które chcemy obserwować. Z menu Register -> Core… uaktywniamy podgląd rejestrów Rx i DAG1(DM) procesora wg. poniższego rysunku:



Po wstawieniu Breakpoint'ów w kodzie programu w odpowiednich miejscach (np. początek funkcji asemblerowej) należy ponownie skompilować i uruchomić projekt. Istnieje możliwość zaobserwowania momentów wpisywania danych do poszczególnych rejestrów co jest uwidocznione na kolejnym rysunku. Wygląd okna pakietu VisualDSP++ przy pracy krokowej:



Po zakończeniu symulacji należy zamknąć projekt wybierając z menu File->Close->Projekt…

Zadanie 2.

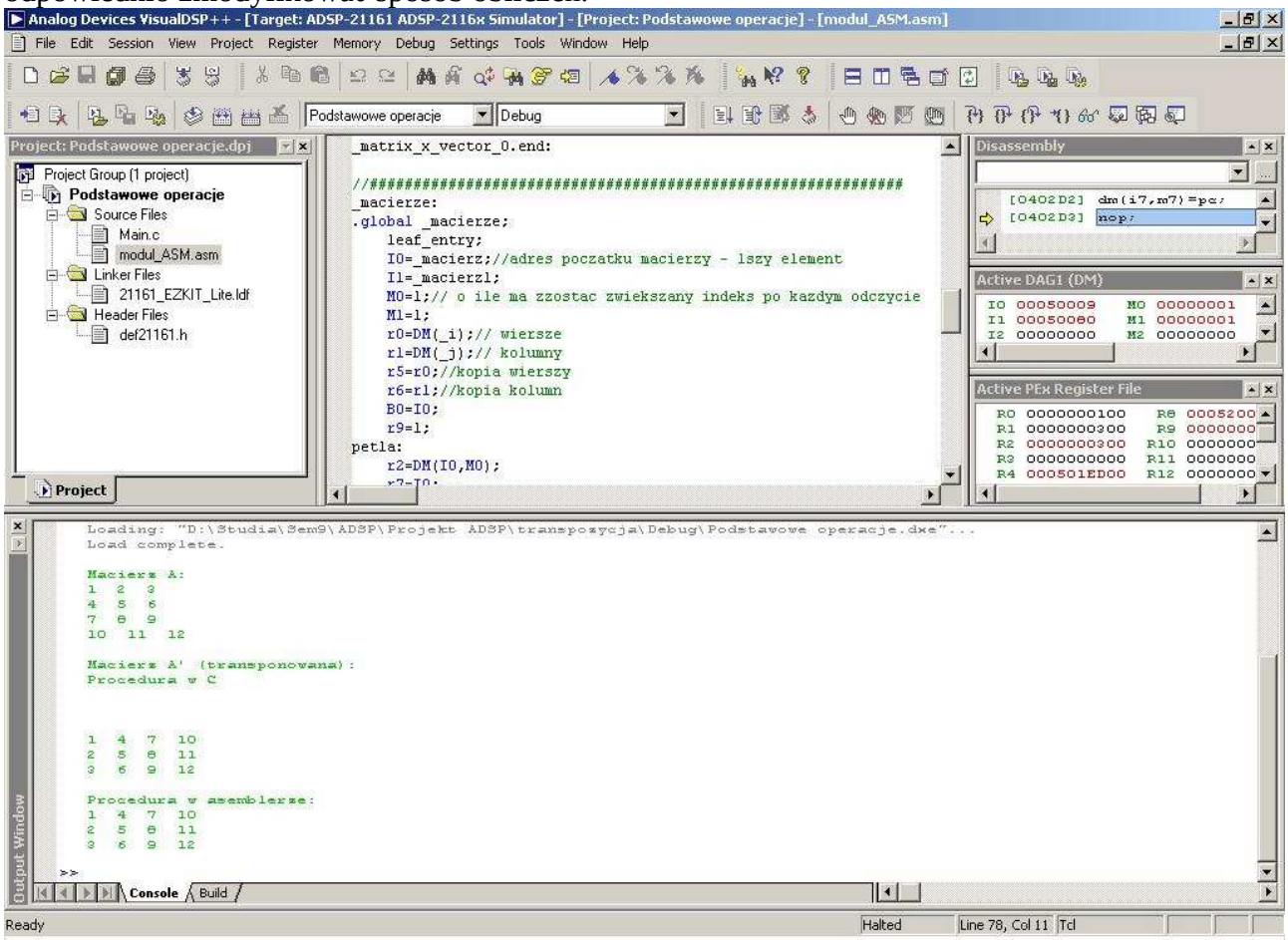
Podobnie jak poprzednio należy otworzyć projekt *Podstawowe operacje.dpj* (tym razem z folderu Transpozycja) z menu File->Open->Projekt… i przeanalizować kod programu napisany w asemblerze *modul_ASM.asm* oraz plik *main.c* , który służy do demonstracji na ekranie terminala wyników działania programu asemblerowego.

Funkcja napisana w asemblerze realizująca transpozycję zadanej macierzy:

```
_macierze:
.global _macierze;
     leaf_entry;
     I0=_macierz; //adres początku macierzy – 1szy element
     I1=_macierz1; // adres I-szego elementu macierzy
     M0=1;//o ile ma zostac zwiekszany indeks po 4stry4
odczycie
     M1=1;
     r0=DM(_i);// wiersze
     r1=DM(_j);// kolumny
     r5=r0;//kopia wierszy
     r6=r1;//kopia kolumn
     B0=I0;
     r9=1;
petla:
     r2=DM(I0,M0); //pobranie danej spod adresu wskazywanego
     przez I0 zmodyfikowanego o M0
```

```
        r7=I0;
        r7=r7+r1;
        r7=r7-1;
        I0=r7;
        r5=r5-1;
        DM(I1,M1)=r2;
        if ne jump petla; // sprawdzanie czy wynik ostatniej
        operacji arytmetycznej lub logicznej jest rowny 0 (zero,
        sprawdzana jest flaga zera) – jeśli tak to skok do
        rozkazu opatrzonego etykietą petla

        r7=B0;
        r7=r7+r9;
        I0=r7;
        r5=r0;
        r9=r9+1;
        r6=r6-1;
        if ne jump petla;

        leaf_entry;
_macierze.end:
```

Podczas analizy kodu programu należy zauważyć, że macierz wielowymiarowa w pliku asemblerowym jest widziana jako tablica jednowymiarowa, więc powinno się zadbać o to, aby odpowiednio zmodyfikować sposób obliczeń.



Okno prezentujące wyniki działania programu na terminalu.

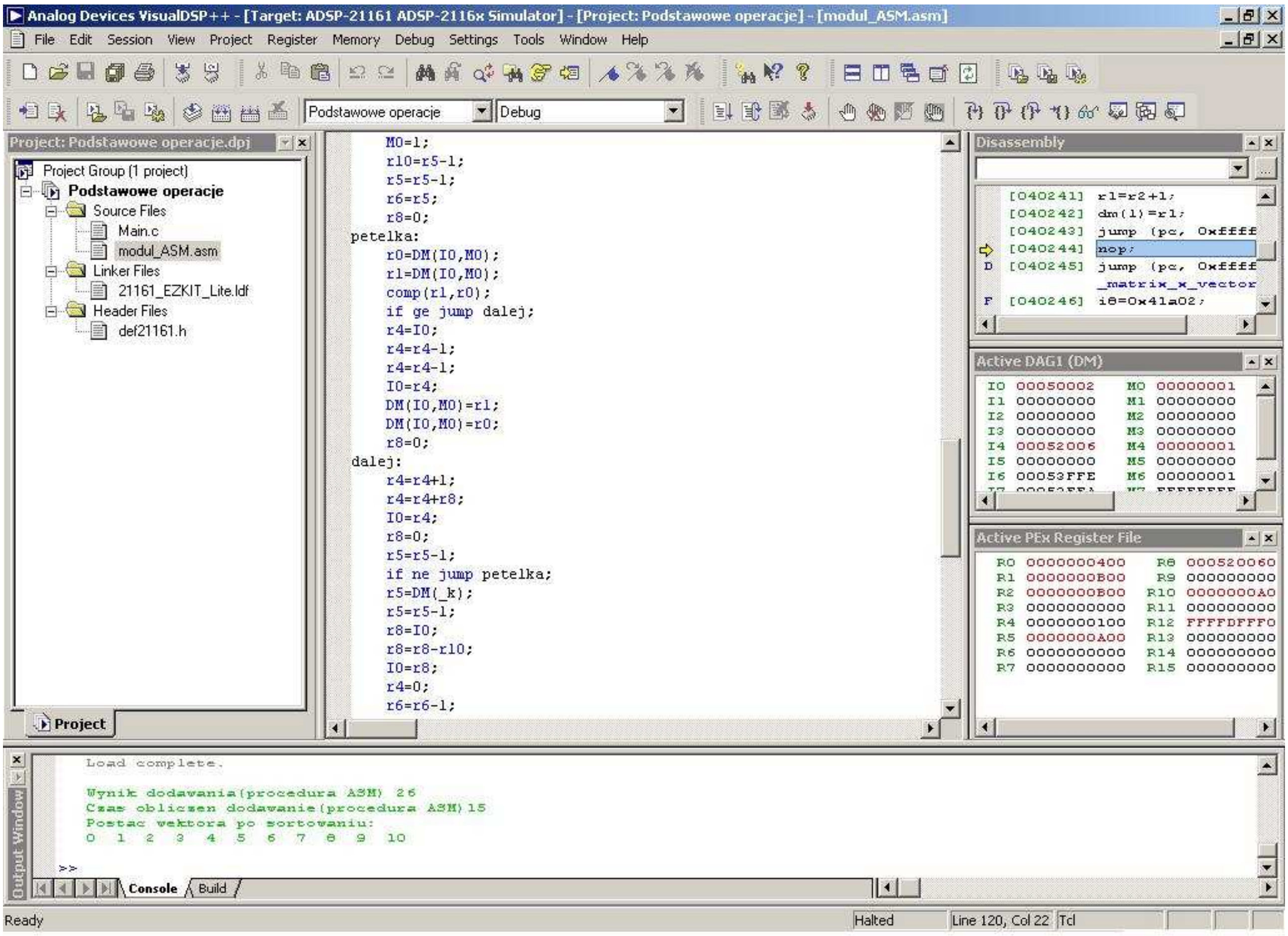
Po wykonaniu programu należy podobnie jak w poprzednim ćwiczeniu zamknąć projekt.

Zadanie 3.

Podobnie jak poprzednio należy otworzyć projekt *Podstawowe operacje.dpj* (z folderu Sortowanie) z menu File->Open->Projekt… i przeanalizować kod programu napisany w asemblerze *modul_ASM.asm* oraz plik *main.c* , który służy do demonstracji na ekranie terminala wyników działania programu asemblerowego.

Listing programu asemblerowego realizującego procedurę sortowania bąbelkowego:

```
_sortowanie:
.global _sortowanie;
     leaf_entry;
     I0=_wektor;     //adres pierwszego elementu
     tablicy wektor

     r5=DM(_k);      //wartość zmiennej k
     M0=1;
     r10=r5-1;
     r5=r5-1;
     r6=r5;
     r8=0;
petelka:
     r0=DM(I0,M0);
     r1=DM(I0,M0);
     comp(r1,r0);
     if ge jump dalej;     //skok do etykiety
     opatrzonej napisem dalej jeżeli wynik ostatniej
     operacji jest większy lub równy 0

     r4=I0;
     r4=r4-1;
     r4=r4-1;
     I0=r4;
     DM(I0,M0)=r1;
     DM(I0,M0)=r0;
     r8=0;
dalej:
     r4=r4+1;
     r4=r4+r8;
     I0=r4;
     r8=0;
     r5=r5-1;
     if ne jump petelka; //sprawdzanie czy wynik
     ostatniej operacji arytmetycznej lub logicznej
     jest rowny 0 (zero, sprawdzana jest flaga zera) –
     jeśli tak to skok do rozkazu opatrzonego etykietą
     petelka

     r5=DM(_k);
     r5=r5-1;
     r8=I0;
     r8=r8-r10;
     I0=r8;
     r4=0;
     r6=r6-1;
     if ne jump petelka;
     leaf_entry;
_sortowanie.end:
```



Okno prezentujące wyniki działania programu na terminalu.

Poniżej zamieszczone są kompletne listingi programów:

- ✓ Potęgowanie:

Plik *modulASM.asm*

```
#include <def21161.h>
#include "adds_21161_ezkit.h"
#include <asm_sprt.h>
.extern            _vector_X;
.extern            _vector_Y;
.extern            _matrix_A_PM;
//---------
.extern            _liczba1;
.extern            _wykladnik,_podstawa;

.segment /pm seg_pmco;
//###########################################################
##################
//# pusta procedura - w celu obliczenia czasu wejscia i
wyjscia z procedury #
//###########################################################
##################

//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
_matrix_x_vector_0:
.global _matrix_x_vector_0;
```

```
   leaf_entry;

   I8=_matrix_A_PM;
   M8=1;
   B8=I8;
   L8=9;

   I1=_vector_X;
   M1=1;
   B1=I1;
   L1=3;

   I2= _vector_Y;
   B2=I2;
   L2=3;
   M2=1;

//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
// wyjscie z procedury  (przywrócenie wartości zerowych w
rejestrach)
   i0=0;
   i1=0;
   b0=i0;
   b1=i0;
   l0=0;
   l1=0;
   m0=0;
   m1=0;

   I2=0;
   B2=0;
   M2=0;
   L2=0;

   leaf_exit;
_matrix_x_vector_0.end:

//##########################################################
####
_potegowanie:
.global _potegowanie;
   leaf_entry;
   r0=DM(_podstawa);
   r1=DM(_wykladnik);
   r2=r0;
   r1=r1-1;
   LCNTR=r1, do kontynuuj until LCE;
kontynuuj:
   r2=r2*r0(ssi);
   DM(_liczba1)=r2;
   leaf_entry;
_potegowanie.end:
//^^^^^^^^^^^^^^
//@@@@@@@@@@@@@@@@@@@@@@@@@@@@


//##########################################################
#######
```

```
//# procedura _count_start  - do pomiaru czasu trwania
procedury #
//############################################################
#######

_count_start: //* call this to start cycle count
.global _count_start;
r1=mode1;
bit clr mode1 IRPTEN;
r0=emuclk;
mode1=r1;
exit;
_count_start.end:

//############################################################
######
//# procedura _count_end   - do pomiaru czasu trwania
procedury #
//############################################################
######

_count_end: //* call this to end cycle count
.global _count_end;
r2=mode1;
bit clr mode1 IRPTEN;
r0=emuclk;
r0=r0-r4;
r1=14;     //* fudge factor to compensate for overhead
r0=r0-r1;
mode1=r2;
exit;
_count_end.end:

.endseg;
```

Plik *main.c*

```
#include "ADDS_21161_EzKit.h"
#include <def21161.h>

#include <signal.h>
#include <math.h>
#include <stdio.h>

float vector_X[3]={1,2,3};

//dane umieszczone w pamięci danych (DataMemory)
int dm liczba1;
int dm podstawa=3;
int dm wykladnik=4;
//taka macierz A umieszczona w pamięci programu
(ProgramMemory)
float pm matrix_A_PM[9]={ 1.0, 2.0 ,3.0, 11.0,12.0,13.0,
21.0,22.0,23.0 };

float vector_Y[3];

int count_start();
int count_end(int);
int time_start, time_elapsed;
```

```
int l;

void potegowanie();
void    main()
{
//#### potegowanie ####
   potegowanie();
   printf("\nWynik potegowania %d^%d=%d
",podstawa,wykladnik,liczba1);

//############################
   while (1)  //pętla nieskonczona
   {
        asm("nop;");
   }
}
```

✓ Traznspozycja macierzy

Plik *modulASM.asm*

```
#include <def21161.h>
#include "adds_21161_ezkit.h"
#include <asm_sprt.h>
.extern           _vector_X;
.extern           _vector_Y;
.extern           _matrix_A_PM;
//---------
.extern           _macierz,_macierz1;
.extern           _i,_j;

.segment /pm seg_pmco;
//#############################################################
##################
//# pusta procedura - w celu obliczenia czasu wejscia i
wyjscia z procedury #
//#############################################################
##################

//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
_matrix_x_vector_0:
.global _matrix_x_vector_0;
   leaf_entry;

   I8=_matrix_A_PM;
   M8=1;
   B8=I8;
   L8=9;

   I1=_vector_X;
   M1=1;
   B1=I1;
   L1=3;

   I2= _vector_Y;
   B2=I2;
   L2=3;
   M2=1;

//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

```
// wyjscie z procedury  (przywrócenie wartości zerowych w
rejestrach)
   i0=0;
   i1=0;
   b0=i0;
   b1=i0;
   l0=0;
   l1=0;
   m0=0;
   m1=0;

   I2=0;
   B2=0;
   M2=0;
   L2=0;

   leaf_exit;
_matrix_x_vector_0.end:

//#############################################################
####
_macierze:
.global _macierze;
   leaf_entry;
   I0=_macierz;//adres poczatku macierzy - 1szy element
   I1=_macierz1;
   M0=1;// o ile ma zzostac zwiekszany indeks po kazdym
odczycie
   M1=1;
   r0=DM(_i);// wiersze
   r1=DM(_j);// kolumny
   r5=r0;//kopia wierszy
   r6=r1;//kopia kolumn
   B0=I0;
   r9=1;
petla:
   r2=DM(I0,M0);
   r7=I0;
   r7=r7+r1;
   r7=r7-1;
   I0=r7;
   r5=r5-1;
   DM(I1,M1)=r2;
   if ne jump petla;
   r7=B0;
   r7=r7+r9;
   I0=r7;
   r5=r0;
   r9=r9+1;
   r6=r6-1;
   if ne jump petla;

   leaf_entry;
_macierze.end:

//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

```
//#############################################################
#######
//# procedura _count_start  - do pomiaru czasu trwania
procedura #
//#############################################################
#######

_count_start: //* call this to start cycle count
.global _count_start;
r1=mode1;
bit clr mode1 IRPTEN;
r0=emuclk;
mode1=r1;
exit;
_count_start.end:

//#############################################################
######
//# procedura _count_end   - do pomiaru czasu trwania
procedura #
//#############################################################
######

_count_end: //* call this to end cycle count
.global _count_end;
r2=mode1;
bit clr mode1 IRPTEN;
r0=emuclk;
r0=r0-r4;
r1=14;     //* fudge factor to compensate for overhead
r0=r0-r1;
mode1=r2;
exit;
_count_end.end:

.endseg;
```

Plik *main.c*

```
#include "ADDS_21161_EzKit.h"
#include <def21161.h>

#include <signal.h>
#include <math.h>
#include <stdio.h>

float vector_X[3]={1,2,3};
//dane te nalezy uaktualniac przy okazji dostawiania
wierszy/kolumn
// do macierzy
int i=4;//ilosc wierszy macierzy
int j=3;//ilosc kolumn macierzy
//dane umieszczone w pamięci danych (DataMemory)
int dm macierz[4][3]={1,2,3,4,5,6,7,8,9,10,11,12};
int dm macierz1[3][4];
//taka macierz A umieszczona w pamięci programu
(ProgramMemory)
float pm matrix_A_PM[9]={ 1.0, 2.0 ,3.0, 11.0,12.0,13.0,
21.0,22.0,23.0 };
```

```
float vector_Y[3];
int wiersz,kolumna;
int count_start();
int count_end(int);
int time_start, time_elapsed;

void dodawanko();
void macierze();
void transpozycja();
void    main()
{
//#### transpozycja macierzy ####
   transpozycja();

//############################
   while (1)  //pętla nieskonczona
   {
        asm("nop;");
   }
}

void transpozycja()
{
   printf("\nMacierz A: \n");
   for(wiersz=0; wiersz<i; wiersz++)
   {
        for(kolumna=0; kolumna<j;kolumna++)
        {
             printf("%d  ",macierz[wiersz][kolumna]);
        }
        printf("\n");
   }
   printf("\nMacierz A' (transponowana):\n");
   printf("Procedura w C \n ");
   for(kolumna=0; kolumna<j;kolumna++)
   {
        for(wiersz=0; wiersz<i; wiersz++)
        {

   macierz1[kolumna][wiersz]=macierz[wiersz][kolumna];

        }
        printf("\n");
   }
   for(wiersz=0; wiersz<j; wiersz++)
   {
        for(kolumna=0; kolumna<i;kolumna++)
        {
             printf("%d  ",macierz1[wiersz][kolumna]);
        }
        printf("\n");
   }

   macierze();
   printf("\nProcedura w asemblerze: \n");
   for(wiersz=0; wiersz<j; wiersz++)
   {
        for(kolumna=0; kolumna<i;kolumna++)
```

```
            {
                printf("%d  ",macierz1[wiersz][kolumna]);
            }
            printf("\n");
        }
    }
```

✓ Sortowanie elementów wektora

Plik *modulASM.asm*

```
#include <def21161.h>
#include "adds_21161_ezkit.h"
#include <asm_sprt.h>
.extern          _vector_X;
.extern          _vector_Y;
.extern          _matrix_A_PM;
//---------

.extern          _k;
.extern          _wektor;

.segment /pm seg_pmco;
//#############################################################
##################
//# pusta procedura - w celu obliczenia czasu wejscia i
wyjscia z procedury #
//#############################################################
##################

//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
_matrix_x_vector_0:
.global _matrix_x_vector_0;
  leaf_entry;

  I8=_matrix_A_PM;
  M8=1;
  B8=I8;
  L8=9;

  I1=_vector_X;
  M1=1;
  B1=I1;
  L1=3;

  I2= _vector_Y;
  B2=I2;
  L2=3;
  M2=1;

//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
// wyjscie z procedury  (przywrócenie wartości zerowych w
rejestrach)
  i0=0;
  i1=0;
  b0=i0;
  b1=i0;
  l0=0;
  l1=0;
  m0=0;
  m1=0;
```

```
   I2=0;
   B2=0;
   M2=0;
   L2=0;

   leaf_exit;
_matrix_x_vector_0.end:
//############################################################
####

//^^^^^^^^^^^^^
_sortowanie:
.global _sortowanie;
   leaf_entry;
   I0=_wektor;
   r5=DM(_k);
   M0=1;
   r10=r5-1;
   r5=r5-1;
   r6=r5;
   r8=0;
petelka:
   r0=DM(I0,M0);
   r1=DM(I0,M0);
   comp(r1,r0);
   if ge jump dalej;
   r4=I0;
   r4=r4-1;
   r4=r4-1;
   I0=r4;
   DM(I0,M0)=r1;
   DM(I0,M0)=r0;
   r8=0;
dalej:
   r4=r4+1;
   r4=r4+r8;
   I0=r4;
   r8=0;
   r5=r5-1;
   if ne jump petelka;
   r5=DM(_k);
   r5=r5-1;
   r8=I0;
   r8=r8-r10;
   I0=r8;
   r4=0;
   r6=r6-1;
   if ne jump petelka;
   leaf_entry;
_sortowanie.end:
//@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


//############################################################
#######
//# procedura _count_start  - do pomiaru czasu trwania
procedury #
```

```
//#############################################################
#######

_count_start: //* call this to start cycle count
.global _count_start;
r1=mode1;
bit clr mode1 IRPTEN;
r0=emuclk;
mode1=r1;
exit;
_count_start.end:

//#############################################################
######
//# procedura _count_end   - do pomiaru czasu trwania
procedury #
//#############################################################
######

_count_end: //* call this to end cycle count
.global _count_end;
r2=mode1;
bit clr mode1 IRPTEN;
r0=emuclk;
r0=r0-r4;
r1=14;     //* fudge factor to compensate for overhead
r0=r0-r1;
mode1=r2;
exit;
_count_end.end:

.endseg;
```

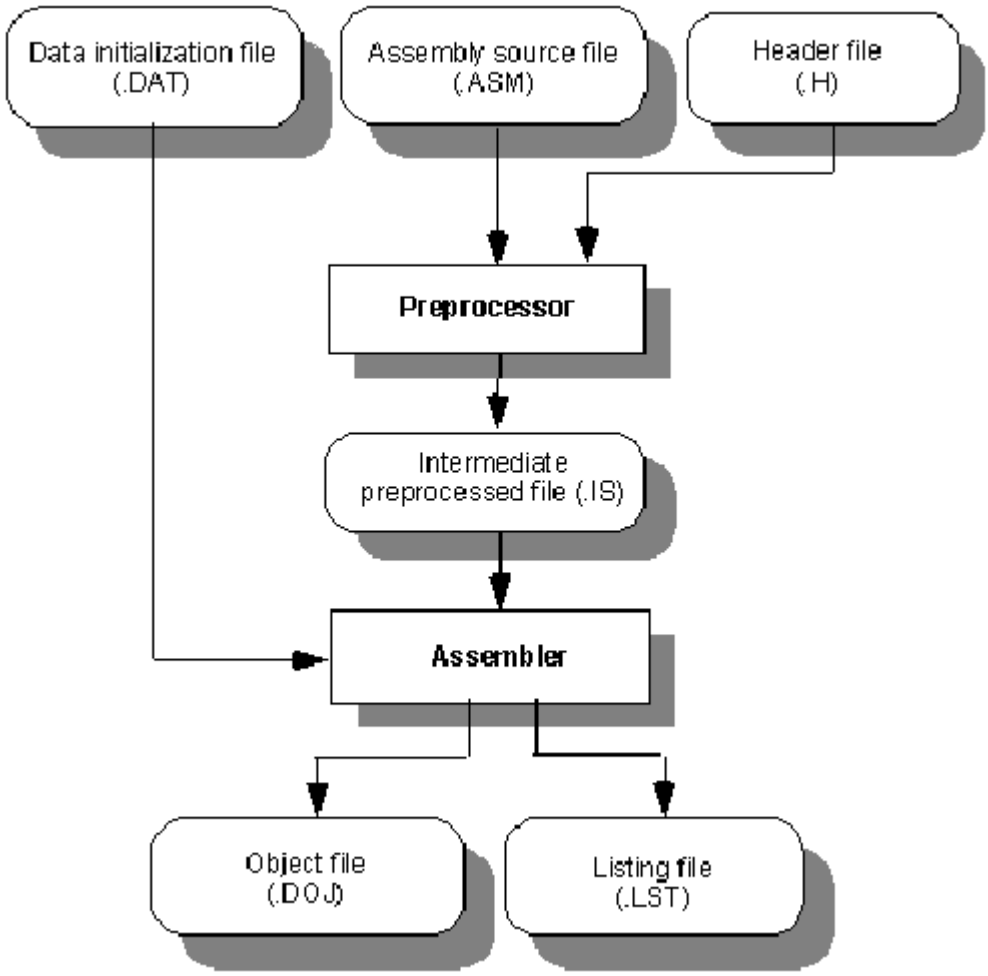
Plik *main.c*

```
#include "ADDS_21161_EzKit.h"
#include <def21161.h>

#include <signal.h>
#include <math.h>
#include <stdio.h>

int k=11;//ilosc elementow wektora
//dane umieszczone w pamięci danych (DataMemory)
//wektor nieuporzadkowany
int dm wektor[11]={8,3,6,1,2,5,10,4,9,7,0};

//taka macierz A umieszczona w pamięci programu
(ProgramMemory)
float pm matrix_A_PM[9]={ 1.0, 2.0 ,3.0, 11.0,12.0,13.0,
21.0,22.0,23.0 };

float vector_Y[3];
float vector_X[3]={1,2,3};

int count_start();
int count_end(int);
int time_start, time_elapsed;
```

```
int l;


void sortowanie();
void    main()
{
//#### sortowanie babelkowe elementow wektora ####


   printf("\nPostac wektora po sortowaniu:\n");
   sortowanie();
   for(l=0;l<k;l++)
        printf("%d  ",wektor[l]);

//############################
   while (1)  //pętla nieskonczona
   {
        asm("nop;");
   }
}
```

Ogólne zasady tworzenia programów w asemblerze dla procesorów AD serii SHARC

Poniższy schemat ilustruje przebieg kompilacji programu napisanego w asemblerze:



Lista wszystkich dostępnych instrukcji procesora jest zawarta w Manualu. Każda linia instrukcji musi być zakończona znakiem średnika (;). Istnieje także możliwość wyodrębnienia wybranej instrukcji poprzez nadanie etykiety w linii poprzedzającej daną instrukcję. Etykieta powinna być zakończona znakiem dwukropka (:). Należy również zaznaczyć, iż w przypadku etykiet rozróżniana jest wielkość liter, stąd np.: *etykieta:* i *Etykieta:* to dwa różne oznaczenia.

Oprócz instrukcji w składni asemblera występują też dyrektywy. Dyrektywy zaczynają się znakiem kropki (.), a kończą średnikiem (;). W tym przypadku wielkość liter nie ma znaczenia. Spis wszystkich dostępnych dyrektyw zawarty jest również w Manualu do procesora. Przykład:

```
.SECTION data_1;
.VAR  const = 0x33;
```

W pliku asemblerowym mogą się znajdować również polecenia preprocesora. Zaczynają się one od znaku (#), a kończą znakiem nowej linii. Jeśli polecenie jest dłuższe niż jedna linia, wówczas powinno się użyć znaku backslash (\) w celu kontynuacji polecenia w następnej linii. Przykład:

```
#include „string.h”
#define MAXIMUM 100
```

Struktura programu

Źródłowy plik asemblerowy definiuje zarówno instrukcje, jak i dane. W pliku LDF (Linker Description File) zawarta jest organizacja (mapowanie) pamięci programu i danych w pamięci fizycznej procesora. Dyrektywa `.SECTION` definiuje pewne grupy kodu i danych programu, które zajmują ciągłą przestrzeń adresową pamięci procesora. Nazwa sekcji zawartej w tej dyrektywie musi odpowiadać określonej sekcji pamięci w pliku LDF. Poniższa tabela zawiera listę sugerowanych dla procesora SHARC segmentów pamięci:

.SECTION Name	Description
seg_pmco	A section in Program Memory that holds code
seg_dmda	A section in Data Memory that holds data
seg_pmda	A section in Program Memory that holds data
seg_rth	A section in Program Memory that holds system initialization code and interrupt service routines

Wywoływanie procedur asemblerowych z poziomu C/C++

Procedury asemblerowe są widziane na poziomie języka C jako funkcje. Należy je uprzednio zadeklarować przed wywołaniem programu głównego w pliku *.c*. Ciało funkcji napisane w osobnym pliku asemblerowym powinno być poprzedzone odpowiednią etykietą o takiej samej nazwie, jak nazwa definicji funkcji w pliku *.c* poprzedzoną znakiem (_).

Do przekazywania zmiennych zdefiniowanych w programie .c do funkcji asemblerowej służy dyrektywa `.EXTERN` (lub `.EXTERN STRUCT` w przypadku bardziej złożonych struktur).

Opisane powyżej niektóre zagadnienia programowania w asemblerze mogą być pomocne w zrozumieniu dołączonych przykładowych programów. Szczegółowy opis (Manual) asemblera procesora AD serii SHARC został również dołączony w postaci pliku PDF.

Literatura

Płyta CD zawiera:

- materiały wykorzystane do realizacji projektu:
  1. *Visual DSP++ 4.0 asm_man.pdf* – zawiera szczegółowe informacje odnośnie pisania programów w asemblerze
  2. *SEC3-Architecture-I-V1.2.pdf*
  3. *SEC4-Architecture-II-V1.2.pdf*
  4. *SEC5-Architecture-V1.2.pdf*

- przykładowe programy:
  1. *Podstawowe operacje* – programy dostarczone podczas zajęć laboratoryjnych(silnia i mnożenie macierzy, pliki wzorcowe do realizacji powyższego projektu)
  2. *potęgowanie*
  3. *transpozycja*
  4. *sortowanie*

  Pozycje 2,3,4 to programy napisane przez grupę projektową.