Sistem multiprocesor cu microcontroler Blackfin 561

Descrierea lucrării

Lucrarea urmărește modul de realizare a unui sistem multiprocesor cu microcontrolerul "dual core" Blackfin 561. Sistemul multiprocesor este descris la nivel hardware – modul de comunicare intre cele doua nuclee de procesor si la nivel software – sincronizarea activităților celor doua procesoare după modelul "producător-consumator". Canalul de comunicație intre cele doua procesoare este un port DMA (direct memory access) denumit IMDMA (Internal Memory DMA). La nivel software se utilizează primitive ale sistemului de operare de timp real VDK (Visual DSP Kernel) pentru sincronizarea intre procesele executate de cele 2 procesoare si asigurarea accesului exclusive la resurse. Se ilustrează modul de implementare si funcționalitatea unui astfel de sistem, cu ajutorul simulatorului Visual DSP ++ 5.x (Analog Devices).

1. Arhitectura unui microcontroler Blackfin 561. Modelul producător- consumator. Descrierea sistemului multiprocesor

Microcontrolerul Blackfin 561 (BF561) are doua nuclee de procesor si are arhitectura prezentata in figura 1. Aceste doua nuclee (*cores*) au memorie privata, partajează o memorie comuna si pot executa in paralel procese. Cele doua nuclee pot sa interacționeze prin schimburi de informații (mesaje) intre memoriile private (de nivel L1) sau memoria comuna (de nivel L2). Comunicarea intre memoriile private se poate face printr-un controler DMA (Internal Memory DMA- IMDMA).

Detalii despre arhitectura BF561 se pot studia la https://www.analog.com/en/products/adsp-bf561.html#. Aplicațiile pentru familia de procesoare Blackfin se pot dezvolta cu suportul unui sistem de operare de timp real denumit VDK (https://www.analog.com/media/en/dsp-documentation/software-manuals/50_vdk_mn_rev_3.5.pdf). Prin utilizarea mediului de dezvoltare Visual DSP ++ 5.x pentru procesoare Balckfin (https://www.analog.com/en/design-center/evaluation-hardware-and-software/software/vdsp-bf-sh-ts.html) se creează proiecte cu suport VDK, in mod automatizat.

Scopul acestei lucrări este acela de a ilustra un sistem multiprocesor bazat pe cele doua nuclee Blackfin, pornind de la un proiect VDK dezvoltat după modelul Producător- Consumator (Producer-Consumer).

Modelul producător consumator este ilustrat in figura 2. Exista doua procese (thread-uri), **Producer** si **Consumer** in cele doua nuclee ale microcontrolerului (COREA si respectiv COREB), care interacționează pentru îndeplinirea unei sarcini comune. Procesul **Producer** va transmite comenzi către procesul **Consumer**, acesta executa comenzile si va transmite un rezultat către **Producer**. In figura 3 se detaliază activitățile cele doua procese si mesajele transmise intre ele. Figura 4 indica modul de comunicare la nivel hardware, cu ajutorul unui controler DMA, IMDMA (Internal Memory DMA).

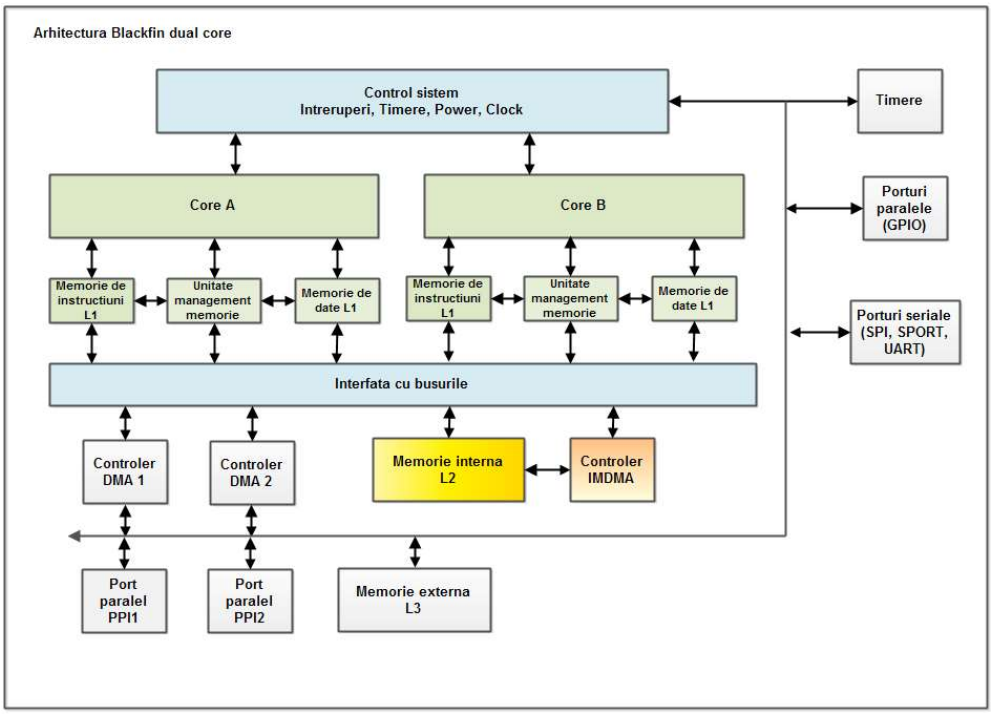


Figura 1. Arhitectura Blackfin 561

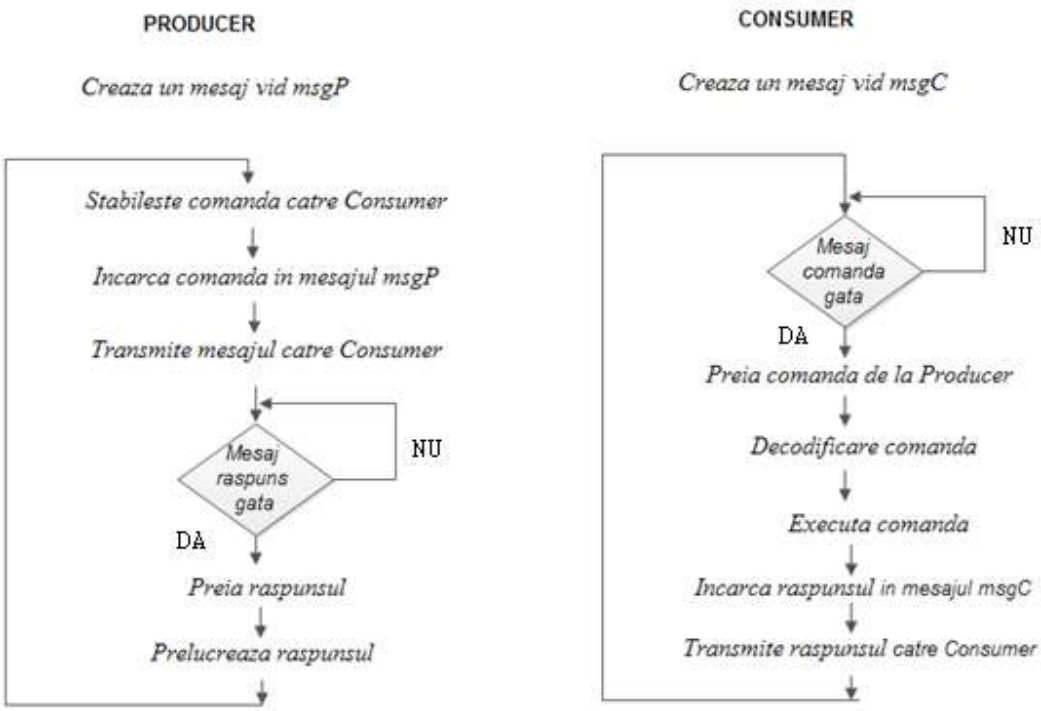


Figura 2. Organigrama proceselor (conform modelului producător consumator).

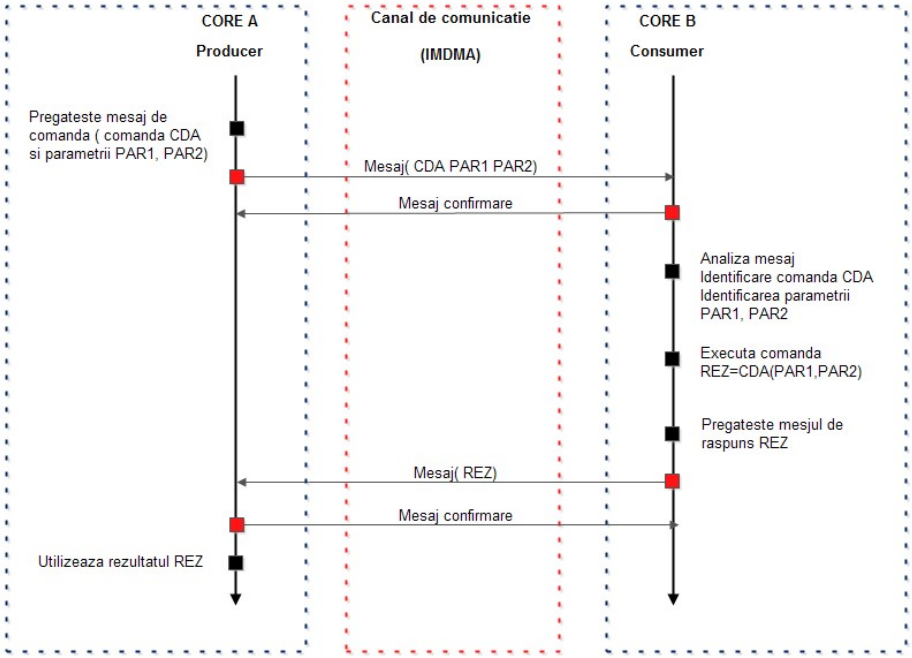


Figura 3. Diagrama de mesaje dintre **Producer** si **Consumer**

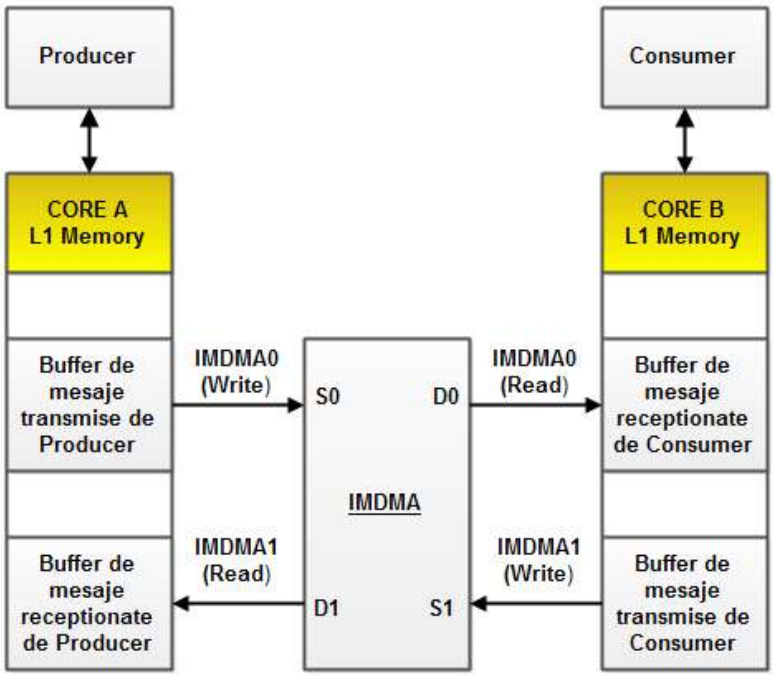


Figura 4. Realizarea comunicatei, la nivel hardware, dintre **Producer** si **Consumer**

Portul IMDMA are 4 canale DMA ce asigura transferuri de tip memorie-memorie intre cele două memorii private ale celor doua nuclee (COREA si COREB). Exista doua fluxuri de comunicație: Stream0 si Stream1. Fiecare flux are doua capete (reprezentate fiecare de catre un canal DMA) : Stream0 este definit de canalele S0 si D0 si asigura comunicația de la **Producer** la **Consumer** (COREA către COREB) si Stream1 este definit de canalele S1 si D1 si asigura comunicația in sens invers, de la **Consumer** la **Producer** (COREB către COREA). Programarea canalelor DMA implica stabilirea adreselor sursa si destinație pentru zonele de memorie intre care se realizează transferul si a numărului de cuvinte transferate. Încheierea transferului va fi semnalizata printr-o întrerupere. Se pot genera doua întreruperi – una când se încheie un transfer de intrare (citire) si alta când se încheie un transfer de ieșire (scriere). Pentru întreruperea asociata transferului de intrare – rutina de servire trebuie sa preia informația transferata, iar pentru întreruperea de ieșire – rutina de servire trebuie sa pregătească o informație noua ce va fi transmisa. Transferul DMA va fi controlat de un driver asociat portului IMDMA.

2. Comunicarea intre nucleele Blackfin 561

Când un mesaj este postat de un thread, se examinează ID-ul nodului (porcesorului) de destinație (încorporat în ID-ul thread-ului de destinație). Dacă ID-ul nodului de destinație este identic cu ID-ul nodului pe care rulează thread-ul care postează un mesaj, atunci acest mesaj este plasat direct în coada de mesaje a thread-ului de destinație (este o comunicație in același nod (core) intre doua thread-uri.

Dacă ID-urile nodurilor sursa si destinație nu sunt identice, atunci mesajul este transmis către un thread de rutare (RThread), care este responsabil pentru etapa următoare în procesul de transmitere a mesajului la destinație. Un thread de tip RThread poate fi configurat ca intrare sau ca ieșire (acest rol este fixat în momentul creării sale). Un thread de tip RThread folosește un driver asociat unui port de intrare/ieșire (IO device diver), care gestionează transmiterea la nivel fizic a mesajelor între noduri distincte. Un RThread de ieșire are portul IO activ pentru scriere, în timp ce un RThread de intrare are portul IO activ pentru citire. Sistemul de operare VDK construiește un tabel de rutare care conține thread-urile Rthread active la scriere (de ieșire).

Când un mesaj trebuie trimis către un alt nod, ID-ul nodului de destinație este utilizat ca index in tabelul de rutare pentru a selecta thread-ul RThread de ieșire care va gestiona transmiterea mesajului. Fiecare nod trebuie să conțină cel puțin un thread RThread de intrare și unul de ieșire, împreună cu driverele IO corespunzătoare.

Un thread de tip RThread activ ca ieșire, când este inactiv, așteaptă ca mesajele să fie plasate pe orice canal al cozii sale de mesaje și apoi transmite acel mesaj efectuând unul sau mai multe apeluri ale unei primitive VDK *SyncWrite()* către driverul IO asociat. Primitiva *SyncWrite ()* așteaptă ca datele de transmis sa fie pregătite (thread-ul Rthread de ieșire intra in starea de blocare daca datele nu sunt pregătite).

Un thread de tip RThread activ ca intrare apelează primitiva VDK *SyncRead()* care așteaptă, prin intermediul driverului IO asociat, ca datele de intrare sa fie pregătite. Daca datele nu sunt pregătite thread-ul Rthread de intrare trece in starea de blocare. Când datele de intrare sunt primite thread-ul de tip RThread de intrare va transmite aceste date la destinație.

In figura 5 este ilustrat modul de comunicație dintre doua noduri din nuclee diferite.

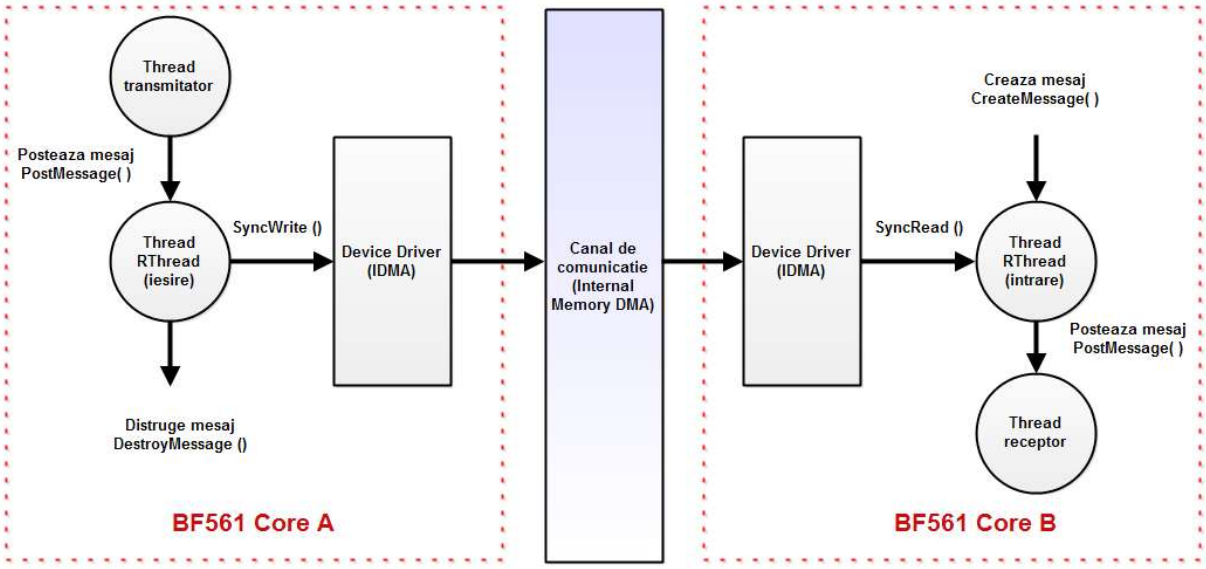


Figura 5. Comunicația dintre doua thread-uri din doua noduri (procesoare) distincte

3. Descrierea software a sistemului

Implementarea software are următoarele componente:

- procesele (thread-urile) **Producer** si **Consumer**.
- driver-ul IO (pentru implementarea transferurilor DMA)

Un thread definit in VDK este caracterizat de o funcție de execuție (*Run()*) care este implementata ca o bucla infinita ce conține codul asociat thread-ului. Planificarea thread-urilor este dictat de prioritatea acestora si de disponibilitatea resurselor necesare execuției. Se utilizează primitive de sistem de operare (VDK) pentru mesaje si dispozitive IO.

Sincronizarea comunicației dintre procese este asigurata de o structura de date speciala a VDK, numita *device flag* – care va indica daca portul IMDMA (device-ul) are date pregătite. Inițial acest flag este 0 (IMDMA nu are date pregătite de transfer).

Funcțiile driver-lui sunt definite într-o funcție de dispecerizare (dispatch) apelata de unul dintre thread-rile RThread (atunci când se transmit mesaje intre noduri distincte). Funcțiile driver-ului sunt următoarele:

1. *Init*
   - creează device flag
   - validare întreruperi IMDMA Stream 0 si IMDMA Stream 1

2. *Open*
   - stabilire mod de lucru (Read sau Write)

3. *SyncRead* sau *SyncWrite*

- configurare IMDMA (access protejat, având in vedere ca registrele de control se afla într-o zona de memorie comuna; protecția se realizează utilizând soluția Peterson – vezi cursul de Sisteme de Operare)
- *PendDeviceFlag* (așteaptă ca device flag sa fie 1; thead-ul poate intra in starea de blocare)

4. *IOCtl* – controlul device-lui (nu se utilizează in acest caz)

5. *Activate* - este apelata din rutina de servire a întreruperii (transfer DMA încheiat):
VDK_ISR_ACTIVATE_DEVICE

- *PostDeviceFlag* (device flag este setat in 1; datele sunt pregătite se pot utiliza).

6. *Close*
- șterge modul de lucru (IMDMA așteaptă noi comenzi)

Ordinea de apelare a acestor funcții in thead-ul RThread este: *Init, Open, SyncRead* (*SyncWrite)* si *Close.* Funcția *Activate* este apelata ca urmare a servirii cererii de întrerupere la încheierea transferului DMA. In figura 5 s-a reprezentat doar apelul funcției principale *SyncRead* (*SyncWrite)* care realizează de fapt transferul propriu-zis.

Programarea transferului DMA este realizata astfel:

- un singur transfer (oprire după transfer)
- cuvinte de 32 biți (4 octeți)
- validare întrerupere
- validare DMA
- direcția transferului
- transfer unidimensional (1D) - adresa de start, număr de cuvinte, modificator

Configurarea DMA(scrierea unor cuvinte de programare in registrele de control ale portului IMDMA care specifica adresele de start pentru sursa si destinație , numarul de cuvinte transferate si modul de lucru) se face astfel:

- înainte de transfer (transfer de intrare)

```
DST_START_ADDR(m_chNum) = m_pData;
DST_X_COUNT(m_chNum)    = m_dataSize >> 2;
DST_X_MODIFY(m_chNum)   = 4;
DST_CONFIG(m_chNum)     = DI_EN | WDSIZE_32 | WNR | DMAEN;
```

- după transfer (transfer de intrare)

```
DST_CONFIG(m_chNum) = DI_EN | WDSIZE_32 | WNR;
```

- înainte de transfer (transfer de ieșire)

```
SRC_START_ADDR(m_chNum) = m_pData;
SRC_X_COUNT(m_chNum)    = m_dataSize >> 2;
SRC_X_MODIFY(m_chNum)   = 4;
SRC_CONFIG(m_chNum)     = DI_EN | WDSIZE_32 | DMAEN;
```

- după transfer (transfer de ieșire)

```
SRC_CONFIG(m_chNum) = DI_EN | WDSIZE_32;
```

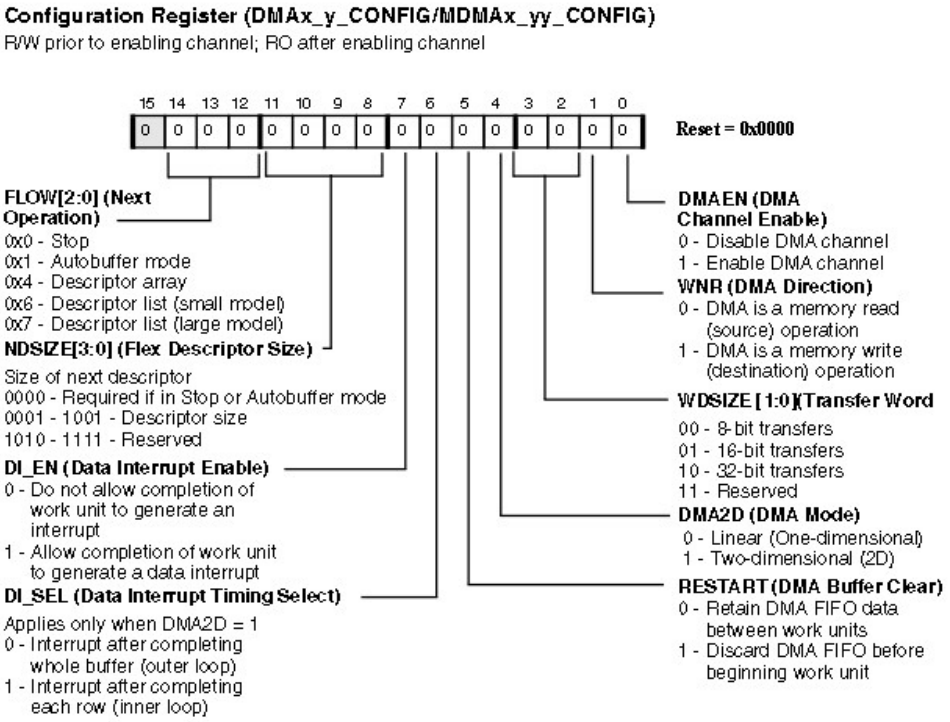
Notațiile sunt următoarele :

DST_START_ADDR, DST_X_COUNT, DST_X_MODIFY, DST_CONFIG si SRC _START_ADDR, SRC _X_COUNT, SRC _X_MODIFY, SRC _CONFIG reprezintă macro definiții ce indica denumirea registrelor de control IMDMA.

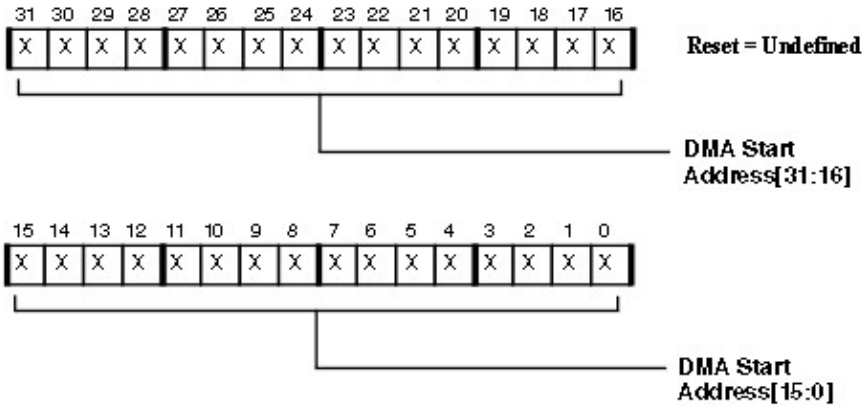
m_chNum – numărul canalului DMA programat
m_pData – pointer la bufferul de date
m_dataSize – dimensiunea zonei de memorie (in octeți)

DI_EN - DMA Interrupt enable, activ in 1
WDSIZE_32 - dimensiune cuvânt 32 biți
WNR - DMA direction : 1 scriere, 0 citire
DMAEN - DMA Enable, activ in 1

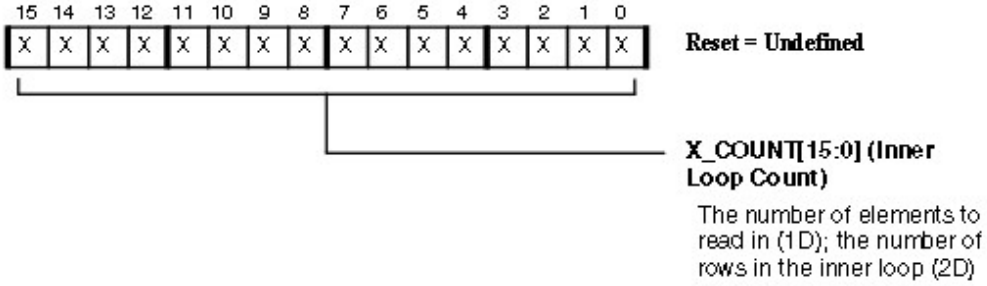
Registre de control IMDMA (Blackfin 561)



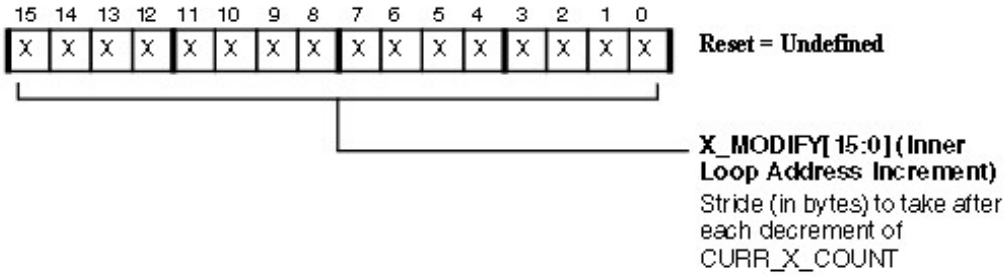
Start Address Register (DMAx_y_START_ADDR/ MDMAx_yy_START_ADDR)
R/W prior to enabling channel; RO after enabling channel



Inner Loop Count Register (DMAx_y_X_COUNT/MDMAx_yy_X_COUNT)
R/W prior to enabling channel; RO after enabling channel



Inner Loop Address Increment Register (DMAx_y_X_MODIFY/MDMAx_yy_X_MODIFY)
R/W prior to enabling channel; RO after enabling channel



4. Funcționarea sistemului

Execuția celor doua thread-uri poate fi explicata prin diagramele de planificare asociate.

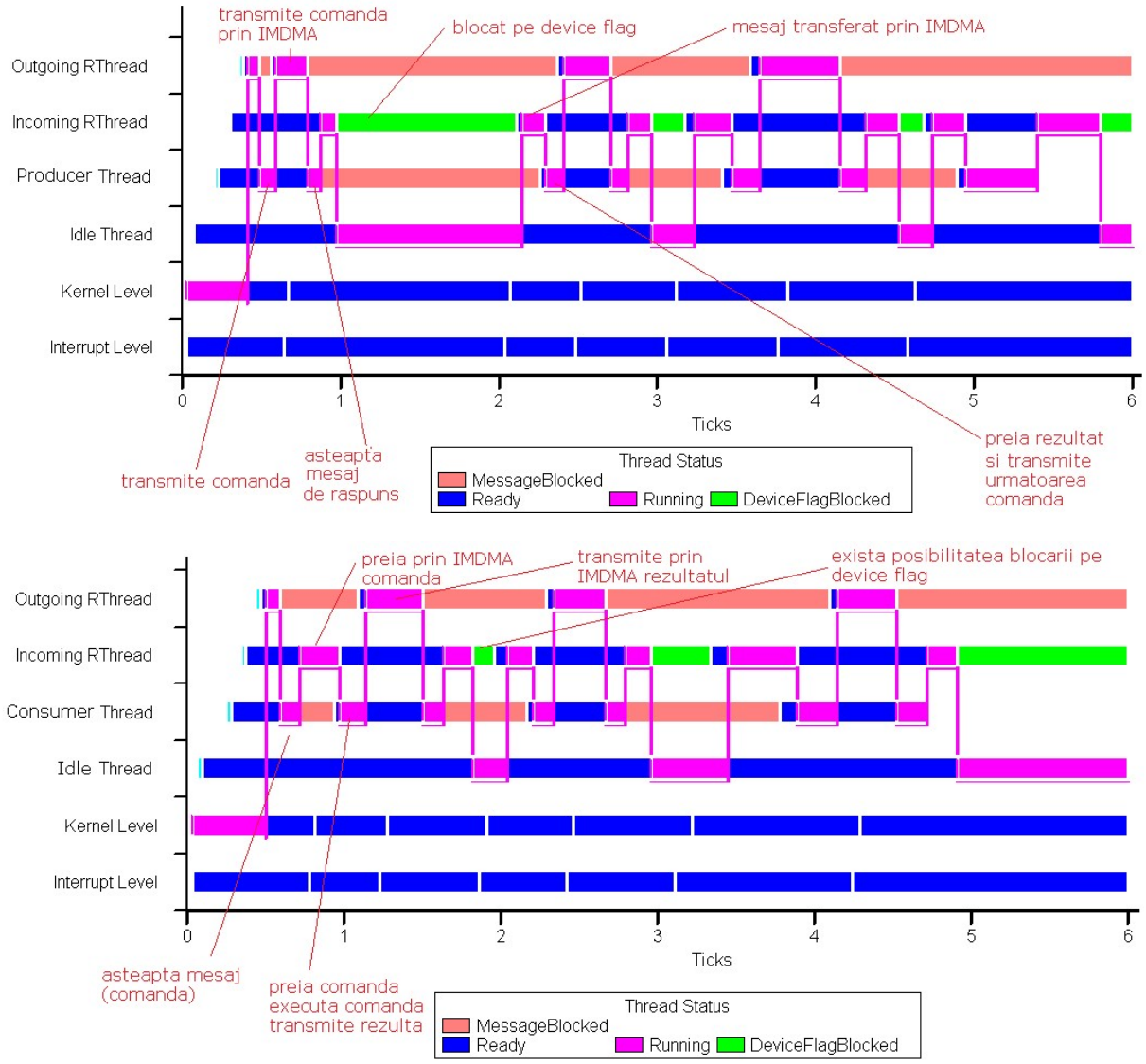


Figura 6. Diagramele de planificare pentru cele doua nuclee P0 si P1

Funcționarea este in conformitate cu organigramele din figura 2, cu deosebirea ca nu se transmit decât un număr de 3 comenzi (si se primesc doar 3 răspunsuri) dupa care ambele procesoare executa doar procesul Idle. Cele doua procesoare lucrează in parale, iar sincronizarea accesului la memoriile private ale acestora se realizează de către driverul IMDMA.

Protejarea accesului la zona de registre de programare ale IMDMA (MMR – Memory Mapped Registers) se face folosind soluția lui Peterson (cu variabila *turn*), ilustrata in figura 7. Nu este necesara utilizarea unui semafor (care ar asigura execuția unui alt thread) deoarece in acest caz

este necesar sa se aștepte terminarea programării portului IMDMA.

```
extern volatile int turn;
extern volatile int interested[2];

static void enter_region(int node)
{
 int other = 1 - node;

 interested[node] = true;
 turn = node;
 while (turn == node && interested[other]) ; // spin
}

static void leave_region(int node)
{
 interested[node] = false;
}
```

Figura 7. Soluția lui Peterson (cu variabila *turn*) pentru asigurarea accesului exclusiv

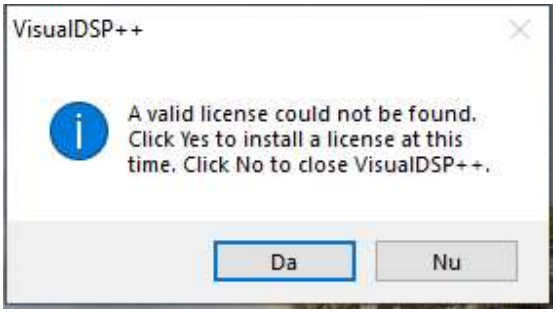
5. Desfășurarea lucrării

Se vor parcurge următoarele etape:

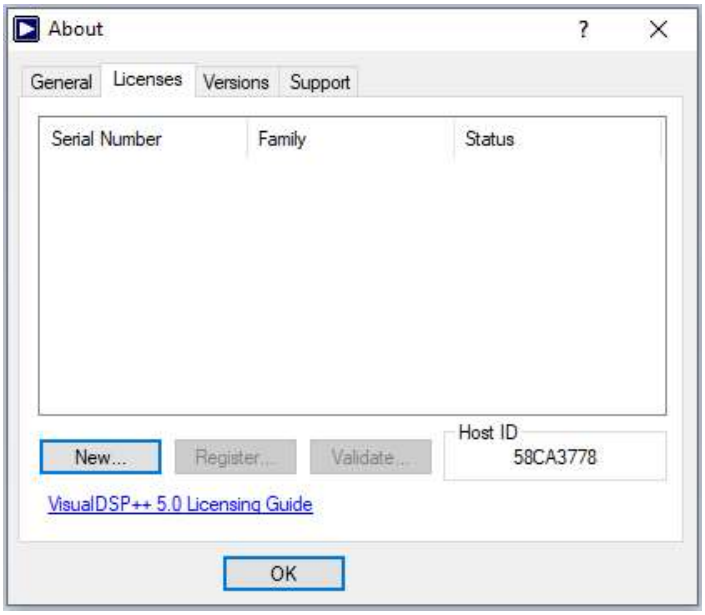
a) Instalare Visual DSP ++ 5.x
Se descarcă kitul Visual DSP++ 5.0 (Blackfin) de la adresa
https://www.analog.com/en/design-center/evaluation-hardware-and-software/software/vdsp-bf-sh-ts.html#software-overview

Licența de test (valabila 90 de zile) este TST-178-346-6228484-85 si trebuie introdusa la prima lansare in execuție a simulatorului.

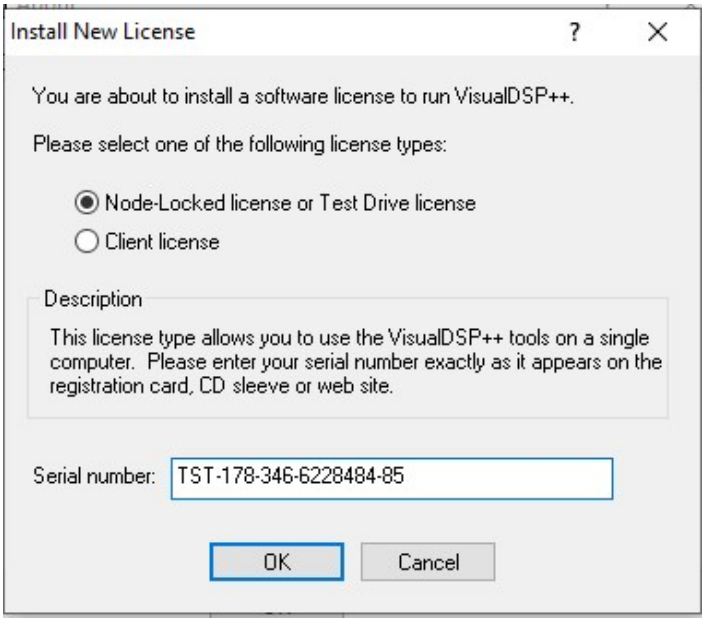
La prima punere in execuție se va genera următorul mesaj:



Se va instala licența de test: Da (Yes) - > New

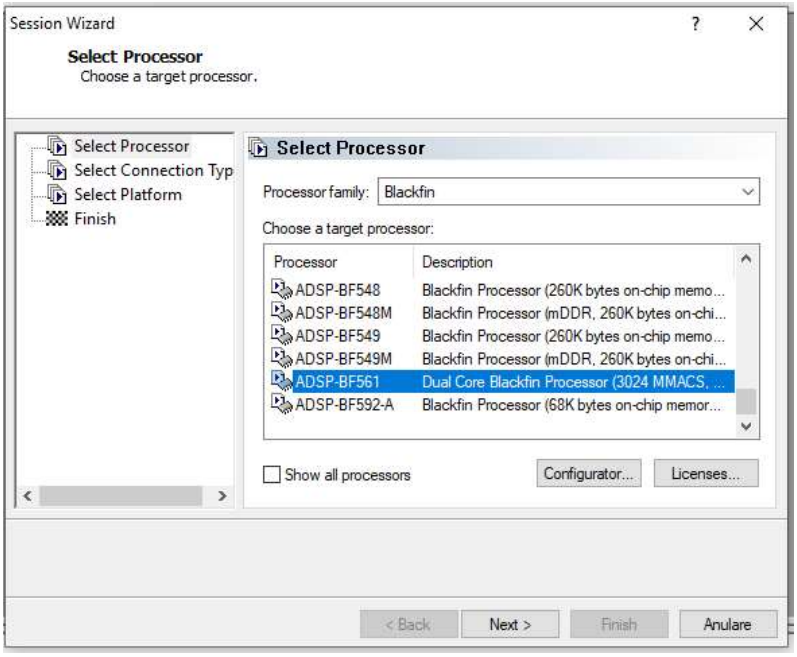


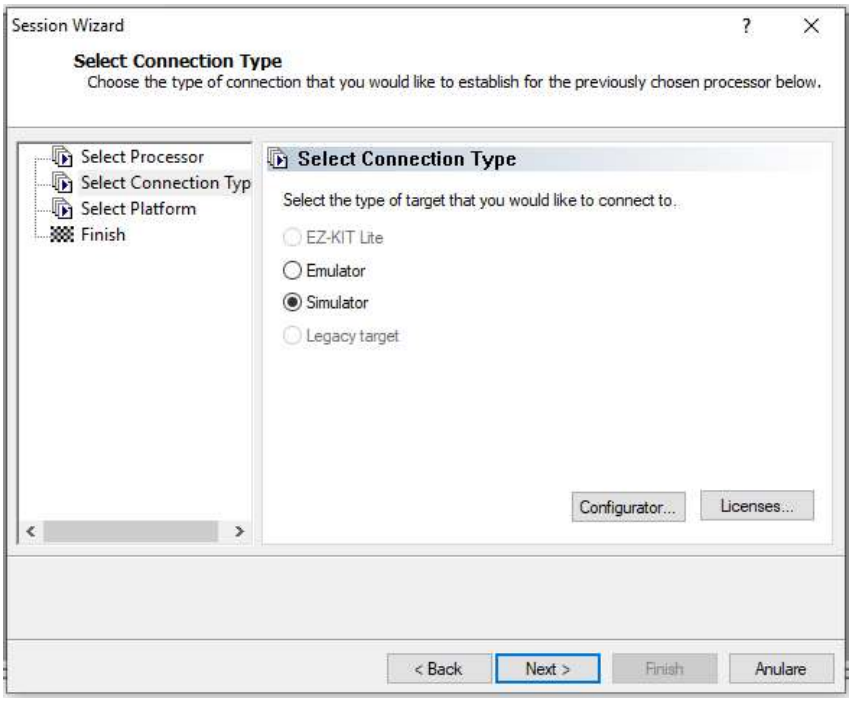
Licența se va introduce ca in fereastra următoare, apoi se da click pe OK.

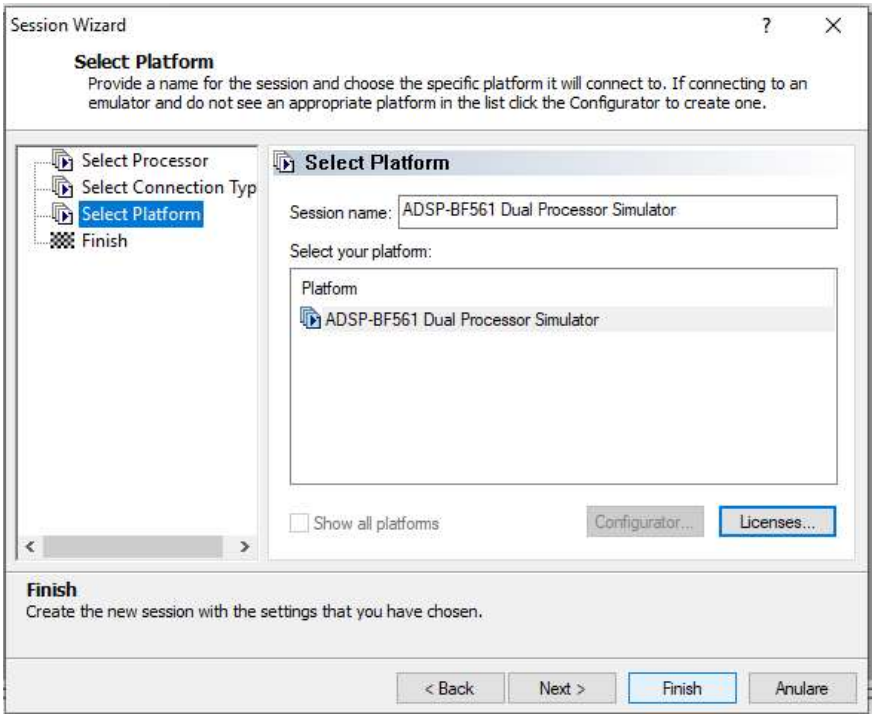


Se lansează in execuție VisualDSP ++ 5.x si se deschide o sesiune noua conform pașilor de mai jos: New Session, Select processor - ADSP BF561, Connection Type – Simulator, Select platform – ADSPBF561 Dual Processor Simulator , Finish


Analog Devices VisualDSP++ - [Target: (Not Connected)]
File
Edit
Session
View
Project
Register
Memory
Debug
New Session...
Select Session
Session List...
Connect to Target
Project Window
Project Group (0 project)

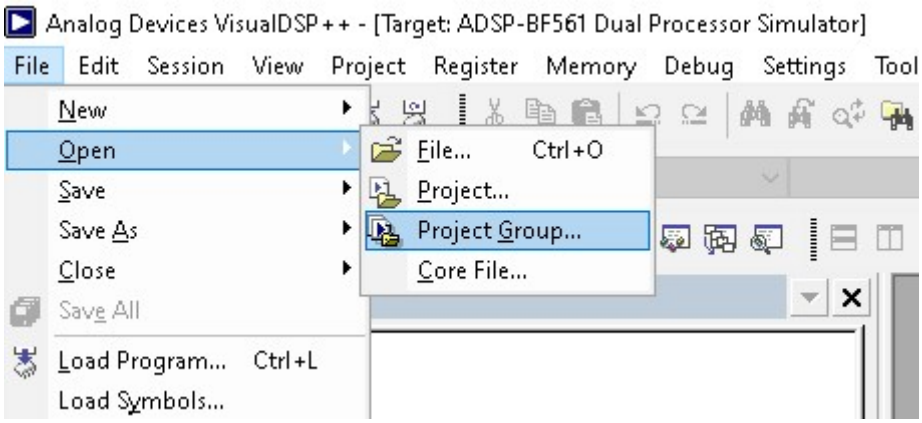

Session Wizard
Select Processor
Choose a target processor.
Select Processor
Select Connection Typ
Select Platform
Finish
Select Processor
Processor family:
Blackfin
Choose a target processor:
Processor
Description
ADSP-BF548
Blackfin Processor (260K bytes on-chip memo...
ADSP-BF548M
Blackfin Processor (mDDR, 260K bytes on-chi...
ADSP-BF549
Blackfin Processor (260K bytes on-chip memo...
ADSP-BF549M
Blackfin Processor (mDDR, 260K bytes on-chi...
ADSP-BF561
Dual Core Blackfin Processor (3024 MMACS, ...
ADSP-BF592-A
Blackfin Processor (68K bytes on-chip memor...
Show all processors
Configurator...
Licenses...
< Back
Next >
Finish
Anulare



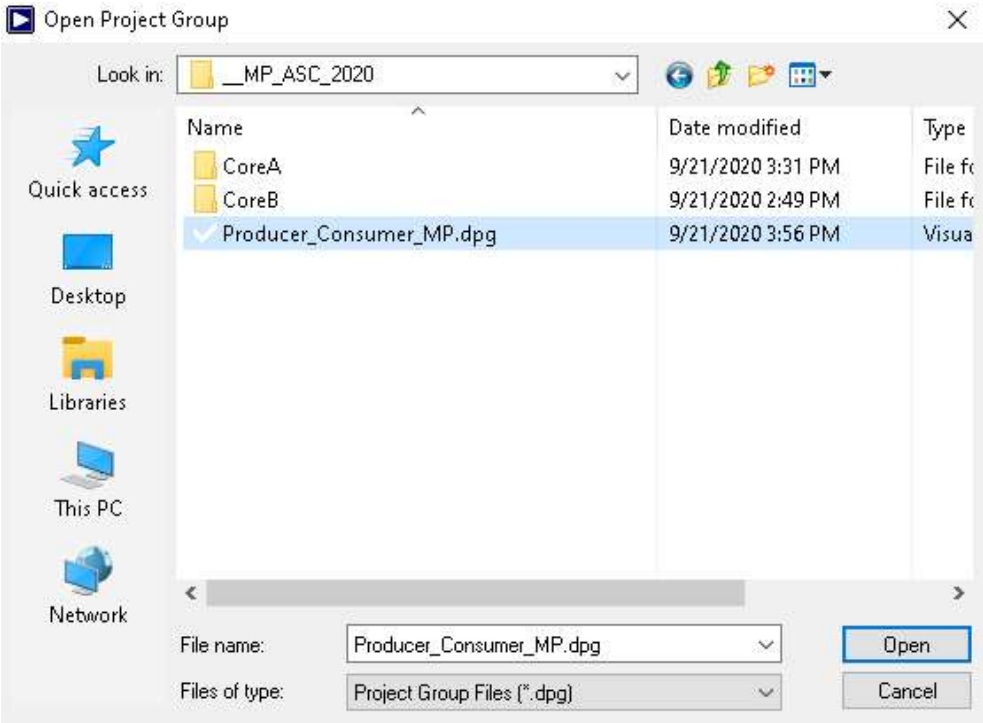


b) Încărcarea grupului de proiecte

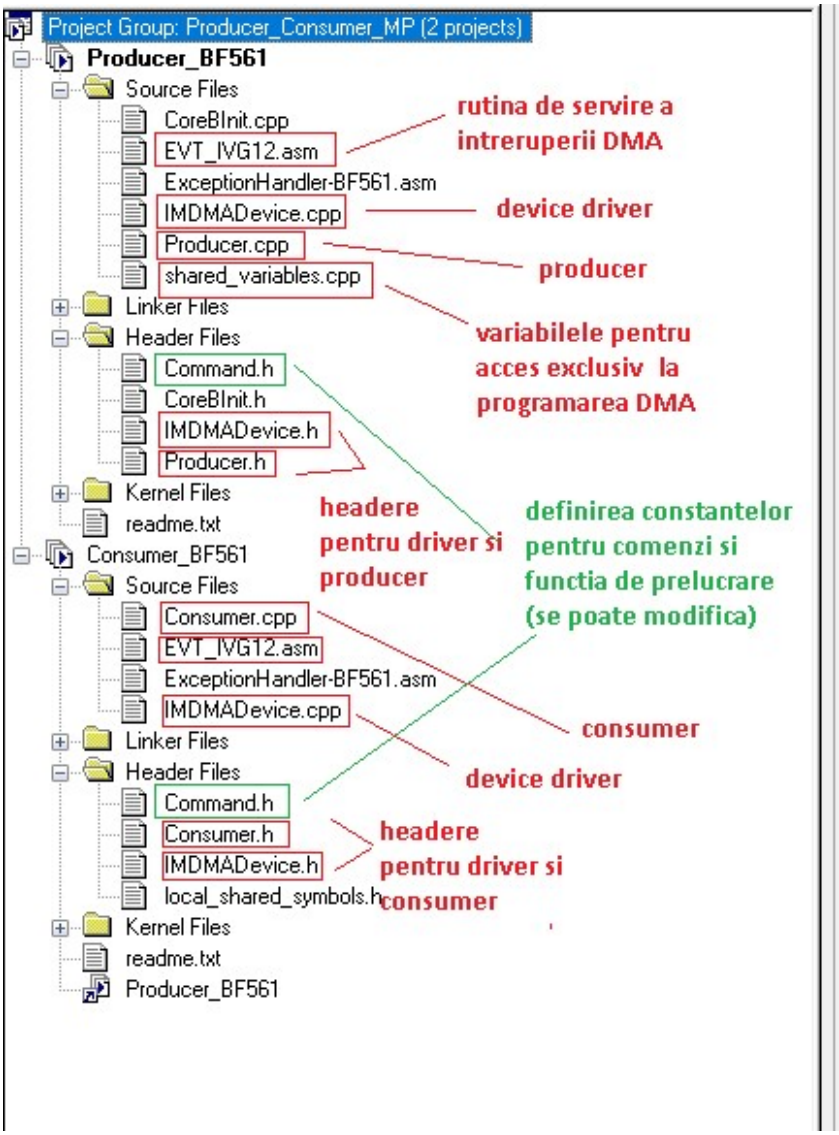
Din meniul principal se da comanda File-> Open -> Project Group



Se va alege grupul de proiecte **Producer_Consumer_MP**



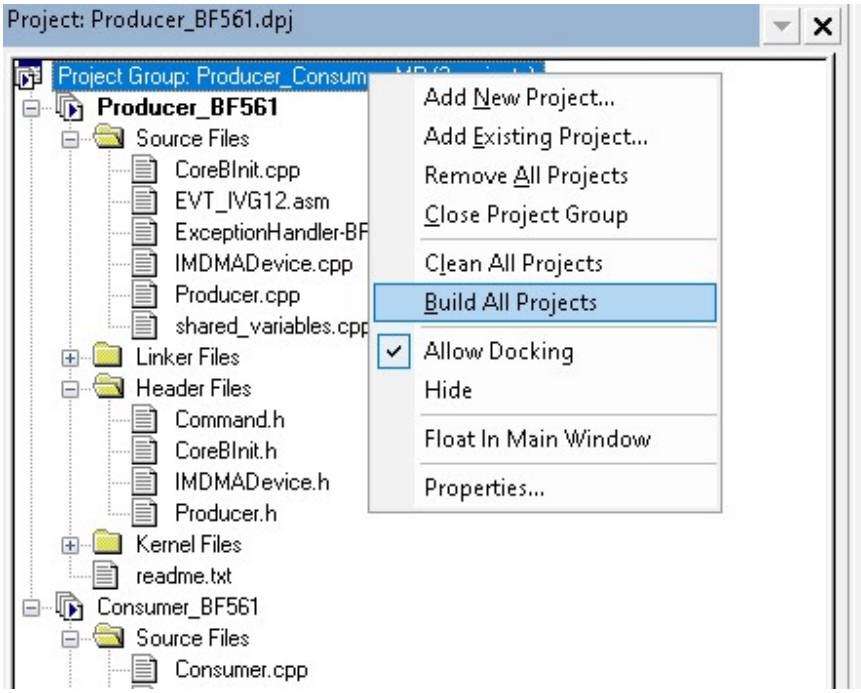
cu structura de mai jos (fișierele încadrate cu roșu sunt fișiere de studiat iar fișierul încadrat cu verde este fișierul unde se pot face modificări).



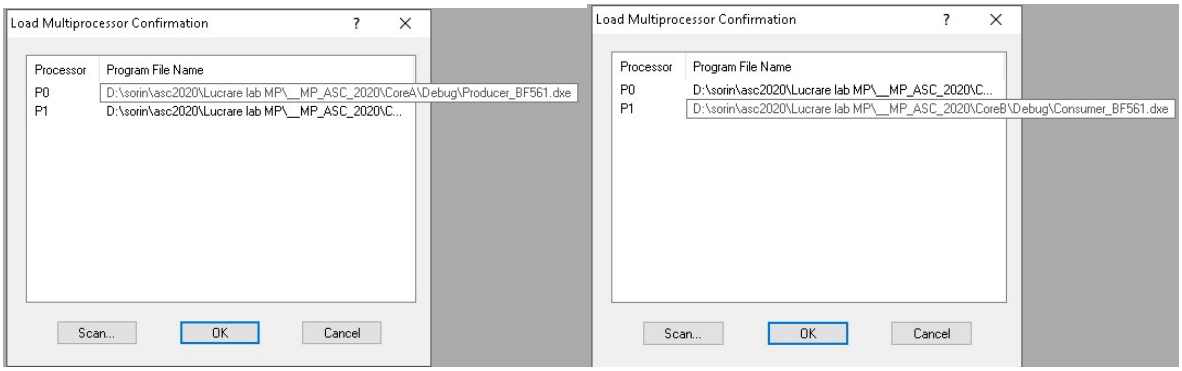
c) Analiza programelor **Producer** si **Consumer** in cele doua core-uri. Se analizează codul si organigramele din figura 2.

d) Compilarea si rularea programelor pe cele 2 nuclee de procesor

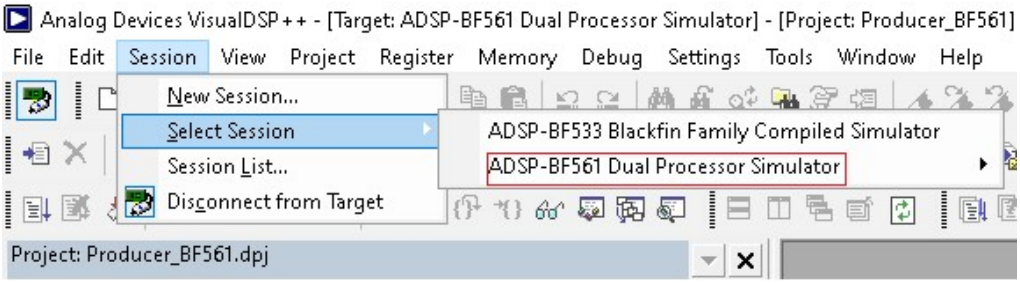
- construirea proiectului (click dreapta pe numele grupului de proiecte apoi Build All Projects)



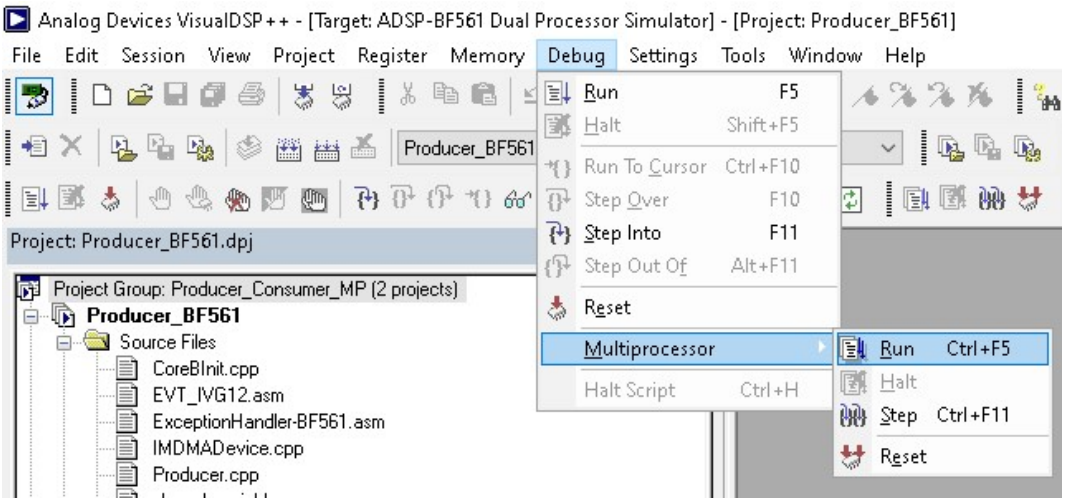
- încărcarea thread-urilor Producer_BF561 in core P0 (COREA) si Consumer_BF561 in core P1 (COREB)



Sesiunea de lucru trebuie sa fie ADSP-BF561 Dual Processor Simulator (daca nu exista se va crea urmărind meniul Session).



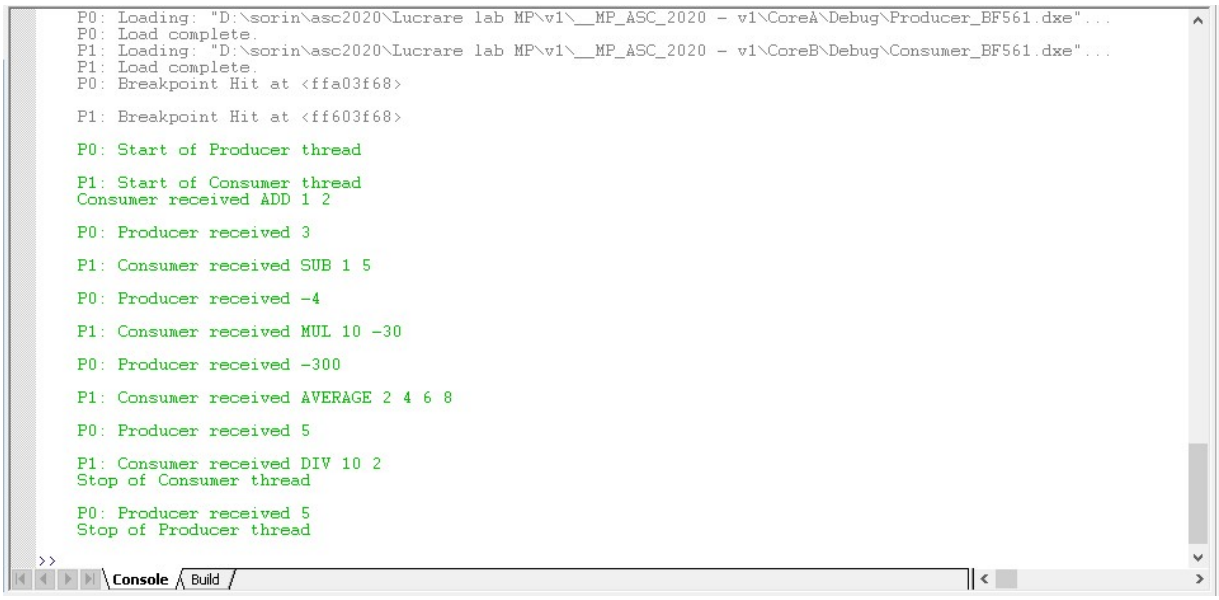
- execuție thread-uri in ambele nuclee simultan (Ctrl F5)



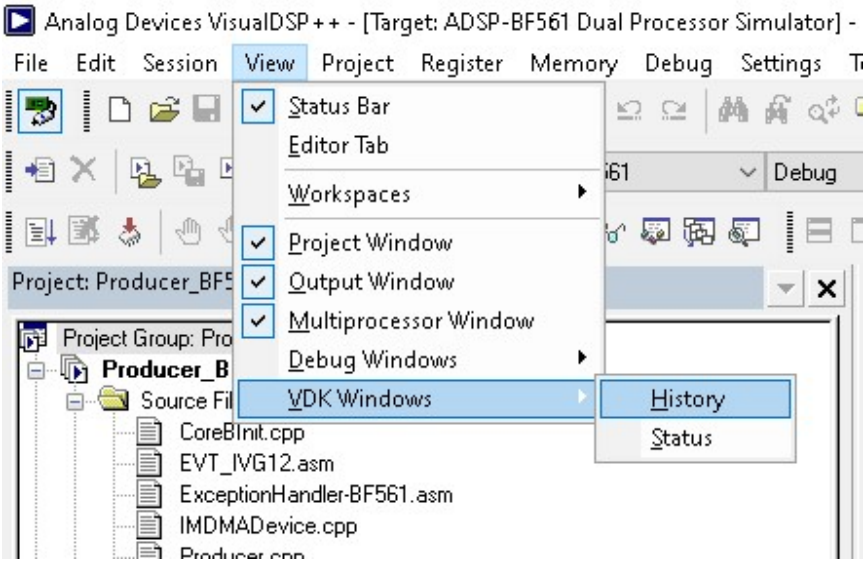
Programul se oprește inițial intra-un punct de oprire (breakpoint) setat de sistem. Reluarea execuției se va face din butonul iar oprirea din butonul

Se vizualizează in fereastra *Console* execuția programelor.

Comenzile date de producător sunt de forma *OPERATIE op1 op2 … opn*. Operațiile codificate prin *OPERATIE* pot fi : *ADD*, *SUB*, *MUL*, *DIV* sau *AVERAGE* cu semnificațiile: adunarea operanzilor *op1* si *op2*, scăderea operanzilor *op1* si *op2*, înmulțirea operanzilor *op1* si *op2,* împărțirea operanzilor *op1* si *op2* si media aritmetica a operanzilor *op1*, *op2, …, opn.* Numărul operanzilor e cunoscut. Comenzile pot fi modificate in fișierul *Command.h*.



e) Vizualizarea si analiza diagramelor de planificare
Comanda View->VDK Windows->History



Se vor afișa diagrame similar cu cele de la punctul 4 din lucrare (Figura 6). Alegerea procesorului pentru vizualizarea diagramei se face din fereastra *Processor* (click pe procesorul dorit). Diagramele pot fi mărite daca se debifează opțiunea Allow Docking din meniul ferestrei (click dreapta pe fereastra deschisa in urma execuției comenzii WDK Windows-> History).

f) Verificarea funcționalității: in fereastra *Console* se verifica daca comenzile scrise in fișierul *Command,h* au fost preluate si executate corect. Se pot introduce noi comenzi in acest fișier.

Codul sursa (extras)

Producer

```
#include "Producer.h"
#include <new>
#include <stdlib.h>
#include "../ProdCons.h"
#include <stdio.h>

#pragma file_attr("OS_Component=Threads")
#pragma file_attr("Threads")

//
#include "../Command.h"
//

/*****************************************************************************
 *  Functii locale ale Producer
 *****************************************************************************/

// Genereaza o comanda, cda, catre Consumer
static void produce_cda(void *pItem, unsigned int *s, char* cda)
{
 char *p;
 p=(char*)pItem;
 strcpy(p,cda);
 *s=sizeof(p);
}

// Preia raspunsul de la Consumer
static void use_rez (void *item, int type, int size)
{
 char tmp[10];
 char *p;
 p= (char*)item;
 strcpy(tmp,p);

 printf("Producer received %s\n",tmp);

}

/*****************************************************************************
 *  Producer Run Function (Producer's main{})
 */

 VDK::MessageID msgP;
 VDK::MessageID msgC;

 void *item;

 VDK::MsgChannel channel;
 VDK::ThreadID sender;

 int type1;
 unsigned int size1;
 void *item1;

 unsigned int s=0;

 unsigned int nr_c=0; // current command

void
```

```
Producer::Run()
{
 printf("Start of Producer thread\n");

 msgP= VDK::CreateMessage(EMPTY_MSG, sizeof(item), (void*)(item));

  while (1)
  {
   // Genereaza comanda - pentru a crea mesajul catre Consumer
    produce_cda(item,&s, COMANDA[nr_c]);

   // Pregatire mesaj catre Consumer
  VDK::SetMessagePayload(msgP, FULL_MSG, s, item);

  // Trimite mesajul catre Consumer
  VDK::PostMessage(kConsumer1, msgP, FULL_MSG_CHANNEL);


   // Asteapta raspunsul
   msgC = VDK::PendMessage(EMPTY_MSG_CHANNEL, 0);

   // Determina de unde a sosit mesajul
   VDK::GetMessageReceiveInfo(msgC, &channel, &sender);

   // Preia continutul mesajului
   VDK::GetMessagePayload(msgC, &type1, &size1, &item1);

   // Utilizeaza raspunsul
  use_rez (item1, type1, size1);

  // Testeza daca mai exista comenzi de transmis
  nr_c++;
  if (nr_c >= NRMAXCOMENZI)
    {
    printf("Stop of Producer thread\n");
    break;
    }
    }
}
```

Consumer

```
#include "Consumer.h"
#include <new>
#include <stdlib.h>
#include <stdio.h>
#include "../ProdCons.h"

#pragma file_attr("OS_Component=Threads")
#pragma file_attr("Threads")

//
#include "../Command.h"
//

int type;
unsigned int size;
void *item;

VDK::MessageID msgP;
VDK::MessageID msgC;
```

```
void *item1;

unsigned int s=0;

//
char cda[N][M]; // comanda

char REZULTAT[10]; // rezultatul

unsigned int nr_c=0; // nr de comenzi executate
//

// Functia de prelucrare a comenzilor
extern void computation(char c[N][M], char res[P]);
//

/*****************************************************************************
 *  Functii locale ale Consumer
 *****************************************************************************/
// Interpreteaza comanda de la Producer
static void parse_exe_cda (void *item, int type, int size)
{
 char tmp[10];
 char *p;
 char *pc;
    int i=0;
    int r;

 p= (char*)item;
 strcpy(tmp,p);

 printf("Consumer received %s\n",tmp);

 // interpreteaza comanda
   pc=strtok(tmp," ");
    while (pc != NULL)
    {
        strcpy(cda[i],pc);
        pc = strtok (NULL, " ");
        //printf("%s\n", cda[i]);
        i++;
    }

    // executa comanda curenta
        computation(cda, REZULTAT);

}

// Stocheaza rezultatul
static void store_rez(void *pItem, unsigned int *s, char* res)
{
 char *p;
 p=(char*)pItem;
 strcpy(p,res);
 *s=sizeof(p);
}
/*****************************************************************************
 *  Consumer Run Function (Consumer's main{})
 */

void
Consumer::Run()
```

```
{
 printf("Start of Consumer thread\n");

 msgC= VDK::CreateMessage(EMPTY_MSG, sizeof(item1), (void*)(item1));

  while (1)
  {
   // Asteapta mesaj de la Producer  (mesaj ce contine o comanda)
   msgP = VDK::PendMessage(FULL_MSG_CHANNEL, 0);

   // Extrage comanda din mesaj
   VDK::GetMessagePayload(msgP, &type, &size, &item);

   // Interpreteaza si executa comanda
   parse_exe_cda(item,type,size);

   // Stocheaza rezultatul in vedere transmiterii acestuia catre Producer
    store_rez(item1,&s,REZULTAT);

   // Pregateste mesajul de raspuns catre Producer
   VDK::SetMessagePayload(msgC, EMPTY_MSG, s, item1);

   // Trimite mesajul de raspuns (cu rezultatul prelucrarii) catre Producer
   VDK::PostMessage(kProducer1, msgC, EMPTY_MSG_CHANNEL);

   // Testeaza daca mai sint comenzi de executat
   nr_c++;
   if (nr_c >= NRMAXCOMENZI)
    {
    printf("Stop of Consumer thread\n");
    break;
    }

    }
}
```

Command

```
/*******************************************************************************
* O comanda e formata din o operatie si un numar cunoscut de operanzi
* Elementele comenzii sint separate prin caracterul SPACE
* Operatiile se efectueaza asupra unor numere intregi (date ca siruri de caractere)
* Rezultatul este tot un numar intreg
*******************************************************************************/

// numarul de elemente in comanda: operatia, operanzii
#define N 5
// numarul de caractere al fiecarui element al comenzii
#define M 10
// numarul de caractere al rezultatului
#define P 4

#define NRMAXCOMENZI  5

char *COMANDA[NRMAXCOMENZI] = {"ADD 1 2", "SUB 1 5", "MUL 10 -30", "AVERAGE 2 4 6 8",
"DIV 10 2"};

void computation(char c[N][M], char res[P])
{
 int i;
 int r;
```

```
    if (strcmp(c[0],"ADD") == 0)
    {
    r=atoi(c[1]) + atoi(c[2]);
    }

    if (strcmp(c[0],"SUB") == 0)
    {
    r=atoi(c[1]) - atoi(c[2]);
    }

    if (strcmp(c[0],"MUL") == 0)
    {
    r=atoi(c[1]) * atoi(c[2]);
    }

    if (strcmp(c[0],"DIV") == 0)
    {
    r=atoi(c[1]) / atoi(c[2]);
    }

    if (strcmp(c[0],"AVERAGE") == 0)
    {
     r=0;
     for (i=1; i<N; i++)
     {
     r=r + atoi(c[i]);
     }
     r = r/(N-1);

    }


    sprintf(res,"%d",r);
}
```