Argo Programming Guide

Evangelia Kasapaki, Rasmus Bo Sørensen

February 9, 2015

Preface

This guide describes how the Argo NOC can be programmed through a description of the APIA and code examples. This document should evolve to be the documentation on writing multicore application for the T-CREST platform.

The most recent version of this guide is contained as LaTeX source in the Patmos repository in directory `patmos/doc/noc` and can be built with `make`.

Acknowledgment

This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST). And partially funded by: The Danish Council for Independent Research | Technology and Production Sciences (FTP) project (Hard Real-Time Embedded Multiprocessor Platform - RTEMP)

Contents

1 Introduction

Argo is a time-predictable Network-on-Chip (NOC) designed to be used in hard real-time multi-core platforms. The main functionality of Argo is to provide time-predictable core-to-core communication in the form of message passing. Argo provides time-predictable message passing using Virtual Circuits. Virtual Circuits (point-to-point communication channels) are implemented using time-division-multiplexing (TDM). According to TDM, shared resources are allocated to Virtual Circuits based on a static TDM schedule. The static TDM schedule can be constructed to fit an application requirements by allocating different amounts of bandwidth to different communication channels.

The steps to build the Patmos tools on a Linux/Ubuntu system are presented in the Patmos Handbook [3]. In this report we assume that the reader is familiar with C and has skimmed through the Patmos Handbook.

1.1 The Architecture of Argo

The Argo NOC is made up of two different components, network interfaces (NI) and routers. The NI converts the transaction based communication from the processor core to stream based communication towards other processor cores in the network. The router is routing the stream of packets that are injected from the NIs through the network according to the static TDM schedule. The routers are connected in a 2D bi-torus structure. A diagram of the architecture of Argo is shown in Figure 1.1(a).

The communication that goes through the network is controlled by the TDM schedule. A direct memory access (DMA) that is placed in the NI, as seen in Figure 1.1(b), is controlling the communication from that NI, enforcing the TDM schedule. There is one DMA controller per communication channel. The TDM schedule divides time in time-slots and during one time-slot one communication channel, i.e. one DMA, is active.

The NI of Argo has two OCP [2] interfaces. The first is a 32-bit interface, connected to the processor core for configuring of TDM schedule and DMA controllers. The second is 64-bit interface, connected to the local memory of the processor, i.e. scratchpad memory (SPM), for accessing the data. The DMA and the SPM of each processor are mapped in the local address space and can be accessed by the processor through specific load and store instructions. The exact address space can be found in [3]. The DMA is setup by the processor core to initiate a message transmission, creating a direct connection from the local SPM of one processor to the local SPM of a remote processor.

Details of the architecture and implementation of the NI can be found in [5]. More details on the router design and the timing characteristics of the NOC can be found in [1].

1.2 Static Time Division Multiplexing Scheduler

The TDM scheduler used to generate schedules for Argo is called Poseidon. The source of Poseidon is placed at `https://github.com/t-crest/poseidon.git` and it is described in [4].

The scheduler is mapping an application onto an application-independent multi-processor platform with the following steps, as illustrated in Figure 1.2. An application is modeled as a *task graph* where nodes represent tasks and edges represent communication channels (end-to-end circuits). The first step is to assign tasks to processors and as part of this to decide which tasks will share a processor. The result of this is a *core communication graph* where the nodes represent processors and the edges represent communication flows and their required bandwidth between the processors. The second step is the binding of processors to specific processor cores in the platform. The third step is to generate the TDM schedule for Argo.

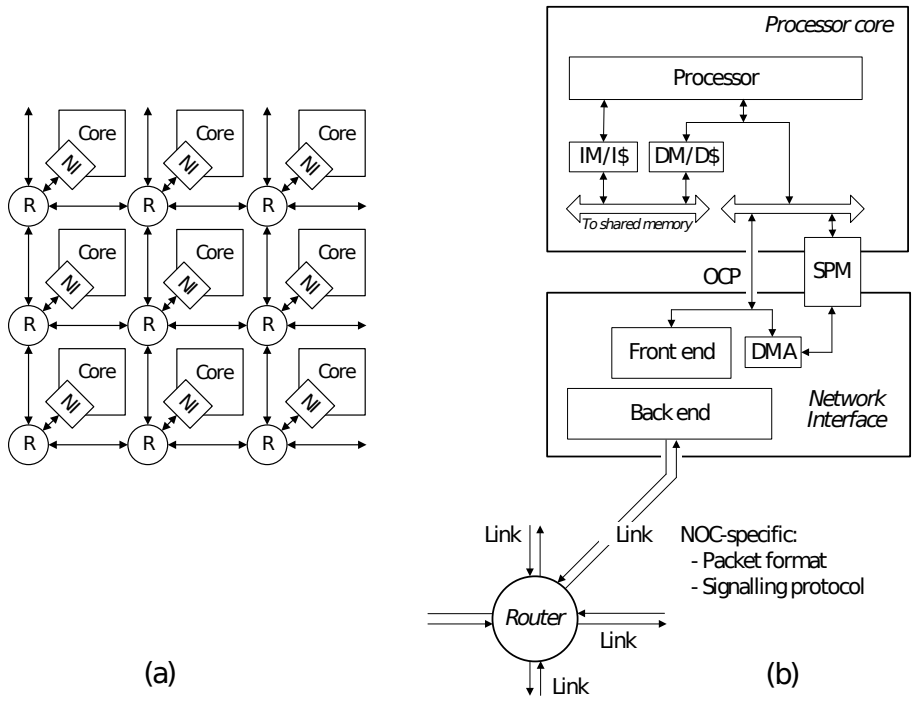

Figure 1.1: The architecture of a single Argo tile.

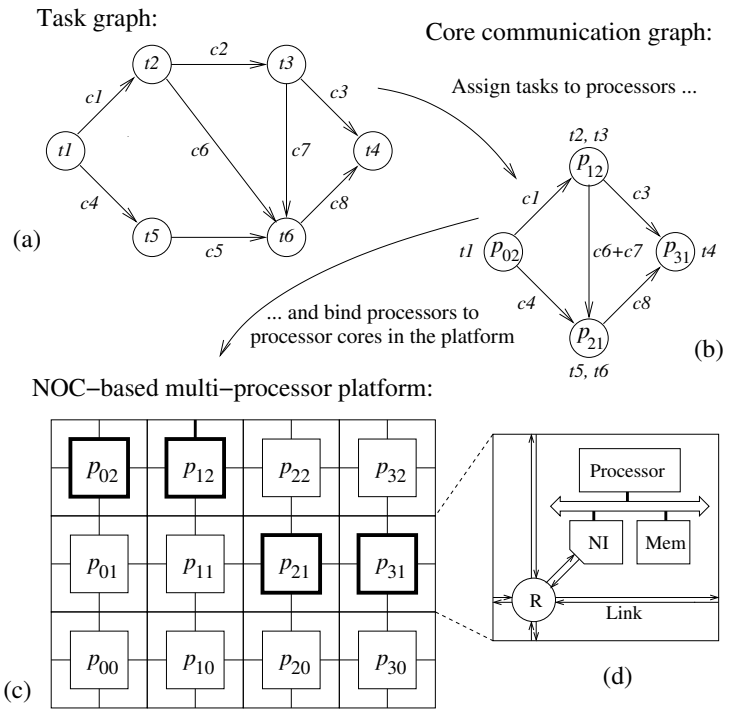

Figure 1.2: Mapping of an application onto a multi-processor platform: (a) Task graph for application, (b) core communication graph, (c) multi-processor platform, and (d) details of a node in the platform (router (R), links, network interface (NI), processor and local memory).

1.3 Argo Usage

Argo can be used in three different levels. In the first level (application programing level) a standard platform and a static predefined schedule are provided. The platform is an NxM platform,operating on an all-to-all schedule with the same bandwidth assigned to every channel. The details on platform initialization, TDM schedule creation and channel configuration are hidden from an application programmer. Instead, library tools are provided for sending messages from processing cores to remote processing cores. The current report focuses on this level of usage.

At a second level (task scheduling level) a fixed platform is provided but a different schedule can be produced based on the task graph and the communication requirements. The details of the schedule production are not presented in the current documentation.

At a third level (platform configuration level) the user can configure a different hardware platform. The configured parameters that can be defined for the platform can be the size of the NOC in an NxM structure, the connecting topology (bi-torus/mesh) and the sizes of memories used (main memory, SPM, cache). The details on different platform configurations are not presented in the current documentation.

2 Argo Programming Guide

This chapter refers to the use of Argo on application programing level.

2.1 Operation of Argo

Argo implements message passing functionality. A processor can push data to a remote processor, i.e. a task running on a processor can send messages to a task running on a remote processor. Each processor has an SPM that is mapped to its local address space. The data to be sent should be placed in the SPM of the local processor. From there the data is sent through a channel in the network and placed in a specified address in the remote SPM.

The sending is controlled by the DMA controller and based on the TDM schedule. The initialization of the TDM schedule is done automatically through initialization of the Aegean platform. The configuration of the DMA is done by the processor. The details of this operation are hidden from the application. However, the application programmer is offered a library, *libnoc* library, with the necessary variables and functions. The full list of variables in libnoc can be seen in Section 2.3.

While the initialization of SPM and configuration of DMA are not transparent to the programmer, the programmer is responsible for the explicit handling of data and the communication in terms of source and destination and SPM addresses. Thus, the programmer should explicitly place data to an address in the local SPM. From each core there is one communication channel to every other core. A sending function is offered in *libnoc* where the programmer should specify the sending channel in terms of destination core as well as the source and destination SPM address. The path information and routing through the network as well as details of the sending operation are hidden from the programmer.

2.2 Build Instructions

In the following we present the build instructions on a Linux/Ubuntu system, for a 2x2 Aegean platform. The default hardware platform includes 4 Patmos processors connected by the Argo NOC and memory arbiter giving access to the shared memory.

2.2.1 Aegean Platform Build Instructions

Information on how to build a Patmos processor and how to run the compiler for Patmos and some application examples can be found in [3].

The platform can be checked out from GitHub as part of the T-CREST project. The T-CREST project will live in `$HOME/t-crest` and the Aegean platform with Argo NOC can be checked out and build with the same commands as for Patmos and the compiler.

Several packages and tools need to be installed and setup. The list of packages and detailed setup instructions can be found in [3]. Additionally, [3] describes a simple Hello world application to be run either by simulation or on FPGA board, for a single Patmos processor.

The whole build process of an Aegean platform,[1] applications in C, configuration of the FPGA, and downloading an application is `Makefile` based. The build of the platform is done in the aegean folder, therefore, the following descriptions assumes you have changed to:

```
t-crest/aegean
```

[1] Get the source from GitHub with: `git clone git@github.com:t-crest/aegean`

The default platform is 4 Patmos cores connected by a 2x2 bi-torus Argo NOC and a memory arbiter connecting to main memory, with a memory controller and pin definitions suitable for the Altera DE2-115 board. The platform is generated by the command:

```
make platform
```

To synthesize the platform for the FPGA board, run:

```
make synth
```

The synthesized platform is configured on the FPGA with the command:

```
make config
```

The configuration can be changed by setting the `AEGEAN_PLATFORM` variable in the above make command. The currently supported boards are:

- Altera DE2-115 (`default-altde2-115`)
- Altera DE2-70 (`default-altde2-70`)
- Xilinx ML605 (`ml605_4core_oc`, only on-chip memory)

More configurations can be found in the directory `config`, where it is also possible to add custom configurations.

2.2.2 Applications

Applications for the multicore platform can be developed in C code, either compiled with the hardware platform, and synthesized on the the on-chip ROM of the FPGA or compiled in ELF binaries, and booted by a bootloader application on FPGA. The bootloader is compiled and synthesized as the default application along with the platform when `make synth` is run.

The application is written in C and should be placed under /t-crest/patmos/c. In both cases the application is build through `Makefile` using the following variables:

- `BOOTAPP` for the built-in on-chip ROM applications.
- `APP` for the ELF binary applications to be executed by the bootloader.

2.2.3 Download of ELF files

An application executed in the multicore platform can be compiled in an ELF file and downloaded on the FPGA. The compilation and downloading is make-based and is done in folder /t-crest/patmos/ with the following make commands:

```
make APP=<app_name> comp
make APP=<app_name> download
```

For a new application the corresponding dependencies should be added in the Makefile that lies under /t-crest/patmos/c.

2.3 Application Programming Guide

2.3.1 Application Programming Interface

Library libnoc is providing an application programming interface. A number of functions and variables are provided through libnoc library. and they are used for two different purposes. The first is to initialize the network interface (NI) and the second is the setup DMA transfers for sending messages to other processing cores. The variables and the functions defined in libnoc are set automatically by the initialization process of the Aegean platform and only a subset of them are used by the application programmer. The complete list is presented in the following for reference purposes.

Variables:

NOC_CORES The number of cores on the platform.

NOC_TABLES The number of tables for the NOC configuration.

NOC_TIMESLOTS The number of timeslots in the TDM schedule.

NOC_DMAS The number of DMAs in the configuration.

noc_init_array The array for initialization data.

NOC_MASTER The master core, which governs booting and startup synchronization.

Functions:

noc_configure Configure network interface according to initialization information in noc_init_array.

noc_init Configure network-on-chip and synchronize all cores.

noc_dma Starts a NoC transfer.

noc_send Transfer data via the NoC.

Defines:

NOC_INIT Define this before including noc.h to force the use of noc_init as constructor. NOC_INIT does not need to be defined if any functions from libnoc are used.

NOC_VALID_BIT The flag to mark a DMA entry as valid.

NOC_DONE_BIT The flag to mark a DMA entry as done.

NOC_DMA_BASE The base address for DMA entries.

NOC_DMA_P_BASE The base address for DMA routing information.

NOC_ST_BASE The base address for the slot table.

NOC_SPM_BASE The base address of the communication SPM.

2.3.2 Hello World Multicore

An application running on the T-CREST multicore platform consists of a main function that is running on all cores. Each core is assigned a `CORE_ID` through the initialization process. The distribution of the workload is decided by the programmer defining a function with the workload for each specific core. The corresponding function is called through main by checking the `CORE_ID`. In the T-CREST platform one of the cores has the role of the master which is the only one that has access to IO. Thus, it can be used as the point of synchronizing and collecting overall data, as well as outputing the data for viewing purposes.

To demonstrate the use of Argo, a multicore Hello World application *hello_sum.c* is placed in /patmos/c. The application is written in C and can be compiled and downloaded on the FPGA board. In *hello_sum* application, a round trip "hello" message is sent from the master core passing through all the slave cores. Each core is adding its ID to the sum of the IDs and eventually the master will receive the sum of all core IDs.

Compile and Run

For any application to be compiled and run, it needs to be placed in /patmos/c and the target and dependencies should be added in the Makefile in the same directory. To compile and download *hello_sum* application on the board, the following make commands are used:

```
make APP=hello_sum comp
make APP=hello_sum download
```

SPM Access

To develop an application that runs on T-CREST platform, data memory and communication should be handled explicitly by the programmer. Argo NOC and the TDM schedule are initialized automatically through Aegean platform but the communication should be handled by the application programmer. An application is provided access to the main memory (shared), to a number of local SPMs as well as to a communication SPM. In the current report, when SPM is mentioned, it refers to the communication SPM used by Argo NOC.

To send data from one core to another, the sending application should specify an address in the local SPM of the sending core and place the data there. To receive data from another core, the receiving application should specify an address in the local SPM where the data are expected. Thus, the destination address in the receiving SPM should be known by both sending and receiving applications. The base address of the communication SPM is `NOC_SPM_BASE` as defined in the libnoc library and any address of the SPM can be used for either sending or receiving purposes or even for directly processing data. Pointers to access SPM space are defined as:

```
volatile _SPM <type> *<name>
```

The copying of the message should be done manually, copying one-by-one character or integer. The platform operates as *bigendian*. Copying of strings through string.h is not available.

Sending

A message is sent using function noc_send():

```
void noc_send (int rcv_id, volatile void _SPM *dst, volatile void _SPM *src, size_t size)
```

The function returns when the DMA transmitting the message is setup. `rcv_id` is the ID of the receiving core, `dst` the write address in the destination SPM, `src` the read address in the source SPM and `size` the size of the message to be transferred in bytes.

Receiving

Currently, libnoc library does not support a receive message function, therefore a receiver needs to wait and poll in order to know when the data has arrived. To do so, specified addresses where the data are expected to arrive are defined and initialized to 0 before every reception of new data. The receiving application is polling this address to know if the data has arrived or not and needs to clear it after the reception is complete and data are read. Since data is transmitted in packets, to safely consider the reception of the entire message completed, the polling should be done on the last unit of data (char/ int/...) of the message sent.

Printing

To print a message, functions `puts` and `printf` can be used. Data can be print only by the master core and only if it is placed in the main memory of the program. Therefore, a data value that is placed in the SPM should be copied in an application variable in main-memory and then print through `puts` or `printf`.

Master - Slave Application

The code for the *hello_sum* is shown in Listing 2.1 and 2.2, for the master application and the slave application respectively. Master is using `spm_base` address for the data to be sent out, thus is copying the characters of the message one by one to `spm_base`. Master is using `spm_slave` as a destination address to the slaves, as well as a receiving address for the expected data from the slaves. The slaves use `spm_slave` as receiving and sending address of the message. `spm_slave + 20` is the last character of the message, thus it is the address to be initialized to 0 and to be polled to detect the completion of the reception for both the master and slaves.

Matrix multiplier is another application developed for Argo. The source can be accessed at: `https://github.com/t-crest/patmos/blob/master/c/matrix_mult.c`

Listing 2.1: A 2x2 Hello World application: Master application.

```
volatile _SPM char *spm_base = (volatile _SPM char *) NOC_SPM_BASE;
volatile _SPM char *spm_slave = spm_base + 64;
*(spm_slave+20) = 0;

// message to be send
const char *msg_snd = "Hello␣slaves␣sum_id:0";
char msg_rcv[22];

// put message to spm
int i;
for (i = 0; i < 21; i++) {
        *(spm_base+i) = *(msg_snd+i);
}

// send message
noc_send(1, spm_slave, spm_base, 21); //21 bytes

puts("MASTER:␣message␣sent:␣");
puts(msg_snd);

// wait and poll
while(*(spm_slave+20) == 0) {;}

// received message
puts("MASTER:␣message␣received:");
// copy message to static location and print
for (i = 0; i < 21; i++) {
        *(msg_rcv+i) = *(spm_slave+i);
}
*(msg_rcv+i) = '\0';
puts(msg_rcv);
```

Listing 2.2: A 2x2 Hello World application: Slave application.

```
volatile _SPM char *spm_base = (volatile _SPM char *) NOC_SPM_BASE;
volatile _SPM char *spm_slave = spm_base + 64;

// initialize polling address to 0
*(spm_slave + 20) = 0;

// wait and poll until message arrives
while(*(spm_slave + 20) == 0) {;}

// PROCESSING DATA : add ID to sum_id
*(spm_slave+20) = *(spm_slave+20) + CORE_ID;

// send to next slave
int rcv_id = (CORE_ID==3)? 0 : CORE_ID+1;
noc_send(rcv_id, spm_slave, spm_slave, 21);
```

Bibliography

[1] E. Kasapaki and J. Sparsø. Argo: A time-elastic time-division-multiplexed noc using asynchronous routers. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems*, ASYNC '14, 2014.

[2] OCP International Partnership. Open Core Protocol specification, release 3.0, 2009.

[3] M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, and D. Prokesch. Patmos reference handbook.

[4] R. Sørensen, J. Sparsø, M. Pedersen, and J. Højgaard. *A Metaheuristic Scheduler for Time Division Multiplexed Network-on-Chip*. DTU Compute-Technical Report-2014. Technical University of Denmark, 2014.

[5] J. Sparsø, E. Kasapaki, and M. Schoeberl. An area-efficient network interface for a tdm-based network-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1044–1047, San Jose, CA, USA, 2013. EDA Consortium.