The Argo software perspective

A multicore programming exercise

Rasmus Bo Sørensen

Updated by Luca Pezzarossa

April 4, 2018

Preface

This exercise manual is written for the course '02211 Advanced Computer Architecture' at the Technical University of Denmark, but is intended as a stand-alone document for anybody interested in learning about multicore programming with the Argo Network-on-Chip.

This document is subject to continuous development along with the platform it describes. In case you have suggestions for improvement or find that the text is unclear and needs to be elaborated, please write to rboso@dtu.dk or lpez@dtu.dk. The latest version of this document is contained as LaTeX source in the Patmos repository in directory `patmos/doc` and can be built with `make noc`.

Contents

1. Introduction

This document presents the background required to write a multicore program utilizing the Argo NoC [1] for intercore communication in the T-CREST plaform [2]. The exercises should give the reader a good understanding of how the Argo NoC can be utilized in a multicore application. The reader will get experience in how to write a multicore application that uses message passing. In the exercises, we assume that the reader is familiar with the C programming language and multi-threaded programming in general. Furthermore, we assume that the reader has already run a single core application on a Patmos processor in an FPGA, refer to the Patmos handbook [3] for details on the Patmos processor.

An example of the multicore platform is shown in Fig. 1.1. Core P_0 is referred to as the master core, and the rest of the cores are referred to as slave cores. The reason that P_0 is the master core is that when an application is downloaded to the platform, the application starts executing `main()` on the master core; also the serial console is connected to the master core. All cores are connected to the shared external memory, but the bandwidth towards the external memory is quite low. Therefore, the programmer should utilize the NoC as much as possible for core-to-core communication.

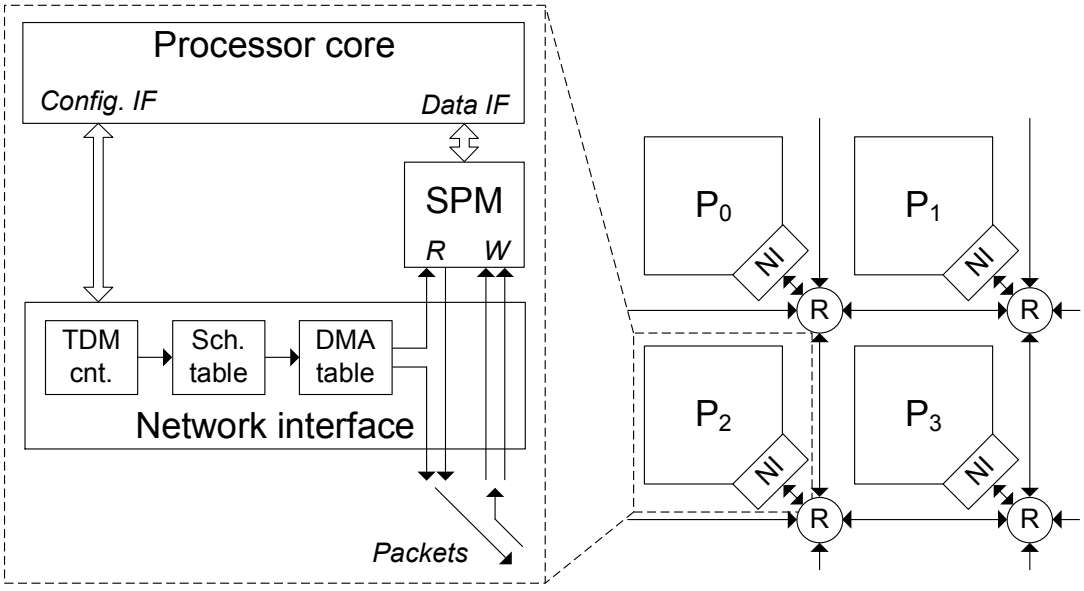

Figure 1.1.: The Patmos multicore platform with the Argo NoC for intercore communication. The core with id 0 is referred to as the master core, and the rest of the cores are referred to as the slave cores.

Chapter 2 presents the architecture of the Argo Network-on-Chip. Chapter 3 describes the programming interface of the multicore platform, including the thread library and the high-level message passing. Chapter 4 contains the practical exercises to give the reader a practical introduction to the platform. Finally, Appendix A describes the practical aspects of loading the program into the platform running in an FPGA.

2. The Architecture of Argo

The Argo network-on-chip (NoC) is a time-predictable core-to-core interconnect. Argo can provide communication channels that have a guaranteed minimum bandwidth and maximum latency. Argo uses direct memory access (DMA) controllers to perform write transactions through the NoC that is interleaved with the TDM schedule. When Argo performs a write transaction through the NoC, it moves a block of data from the local scratchpad memory (SPM) to the SPM of another core in the network.

The guarantees on bandwidth and latency are enforced by a static time division multiplexing (TDM) schedule, where the network resources are allocated to communication channels. A TDM schedule is generated by the Poseidon TDM scheduler, based on some bandwidth requirements that are given in XML format. The statically allocated TDM schedule is loaded into hardware tables in the network interface when the platform boots. It is possible to reconfigure a new schedule at runtime with the reconfiguration capabilities of the Argo NoC, but since the reconfiguration capabilities are not needed for these exercises, they are not described in this document. In these exercises, we assume the default all-to-all schedule where all cores have communication channels to all other cores.

Figure 2.1 shows the architecture of the Argo NoC. The DMA block in the figure contains a table of DMA entries, each entry describes a DMA controller that can send to a remote processor. Each DMA controller is

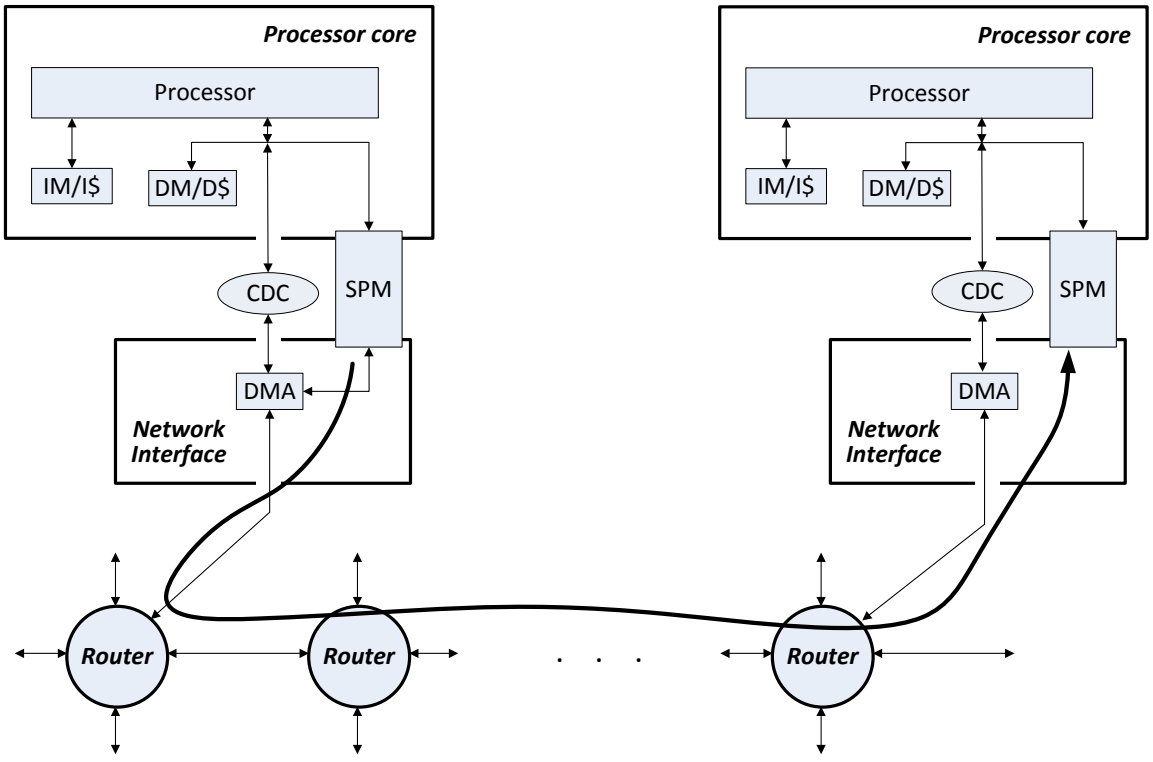

Figure 2.1.: The Argo architecture from a software perspective. A DMA write transaction moves the specified block of data from the communication SPM of the processor on the left to the specified location in the communication SPM of the processor on the right.

paired with a communication channel when the network is configured. To transfer a block of data from a local SPM to a remote SPM, there are 2 steps:

1. Store the block of data in the local SPM
2. Through the network interface set up the DMA controller that is paired with the correct communication channel by:
    - Writing the local address of the block of data and the remote address to which the block of data should be moved
    - Writing the size of the block of data
    - Setting the 'active' bit in the DMA entry to 1

After step 2 the DMA will start to transfer data in each TDM slot that is allocated to the specified communication channel. When the DMA has transferred all packets through the network the 'active' bit is reset by the NI for that DMA entry. The 'active' bit can be pooled to wait for the DMA to finish.

Conflicts of reading and writing to the same addresses in the dual ported SPMs has to be handled by software; there is no protection in hardware.

3. Application Programming Interface

This chapter describes the Argo application programming interface (API). The Argo API is made up of three libraries, the thread library `libcorethread`, the NoC Driver library `libnoc`, and the message passing library `libmp`. In the following three sections, we give an overview of the three libraries.

3.1. Corethread Library

When an application starts executing on the platform, `main()` is executed only on the master core with core ID 0. From the `main()` function the programmer can start the execution of a function on the slave cores using the functions in the `libcorethread` library. The functions of the `libcorethread` library are:

`int corethread_create( int core_id, void(*start_routine)(void *), void *arg ) )`
The create function will start the execution of the `start_routine` function on the core specified by `core_id`, an argument can be given to the started function via the `arg` pointer. The start function should only be called by the master core during the initialization phase of the application.

`void corethread_exit( void * retval )`
The exit function can be called in the `start_routine` functions if they need to return a value to the master core. The exit function should be called as the last thing before the return statement.

`int corethread_join( int core_id, void ** retval )`
The join function will join the program flow of the master core with the program flow of the core specified by `core_id`, and the join function should only be called from the master core. The join function will point the `retval` pointer to the return value allocated by the thread on the slave core. Be aware, the return value should not be allocated on the stack of the slave core!

3.2. NoC Driver

The NoC driver `libnoc` provides direct access to the hardware functionality and only abstracts the low-level accesses to hardware registers away. There are driver functions for initialization of the NoC and for setting up DMA transfers. The `libnoc` library is linked together with the auto-generated c file from the Poseidon scheduler. The auto-generated c file contains the schedule data. The initialization of the NoC is done automatically before the `main()` function starts executing, if the compiler sees that the application uses any functions from the NoC driver. If the application requires direct control over data movement through the NoC the following functions can be used, but it is very advisable to use the message passing library presented in Section 3.3 to reduce the amount of manual memory allocation.

`int noc_dma_done( unsigned dma_id )`
The done function is used to tell whether a local DMA transfer has finished.

`int noc_nbwrite( unsigned dma_id, volatile void _SPM *dst, volatile void _SPM *src, size_t size )`
The nbwrite function is a non-blocking function for writing a block of data at the address `src` of size `size` to the core with the core id `dma_id` and the remote address `dst`. The nbwrite function will fail if the DMA controller is still sending the previous block of data.

`void noc_write( unsigned dma_id, volatile void _SPM *dst, volatile void _SPM *src, size_t size )`
The write function is calling the nbwrite in a while loop until it returns success.

3.3. Message Passing Library

The libmp adds flow control, buffering and memory management on top of the libnoc. libmp implements two different concepts of message passing, queuing message passing and sampling message passing. Queuing message passing implements a first-in-first-out queue where all messages have to be consumed by the receiver. Sampling message passing implements atomic updated of a sample value, this sample value can be read multiple times or not read at all before the next update.

To communicate from one core to another, each core must create a port of the same type, either sampling or queuing. There must be one source port and one sink port. Furthermore, the unique channel identifier for the two ports must be the same.

void _SPM * mp_alloc(coreid_t id, unsigned size)
: The alloc function will allocate a block of memory of size size in the SPM local to the core with the id id. The alloc function can only be called from the master core executing main() and once the memory block is allocated, it cannot be freed. In the current version of the software, the alloc function will not give an out of memory error, so the programmer should be aware to not allocate more local memory than what is present.

qpd_t * mp_create_qport(unsigned int chan_id, direction_t direction_type, size_t msg_size, size_t num_buf)
: The create_qport function allocates the static buffer structures of a communication channel and initializes the queuing port descriptor qpd_ptr. The communication channel is set up between the sending core sender and the receiving core receiver. The communication channel will transfer messages of size msg_size and buffer a number of num_buf messages in the receiver SPM.

int mp_nbsend(mpd_t* mpd_ptr)
: The nbsend function checks if there is a free buffer in the receiver and if the DMA controller for the given communication channel is free. If both are free, it will set up the DMA to transfer the new block of data. The nbsend function assumes that the user/application already wrote the data to be sent to the write_buf buffer.

void mp_send(mpd_t* mpd_ptr, const unsigned int timeout_usecs)
: The send function calls the nb_send function in a loop until it returns success. Put timeout_usecs at 0 if not used.

int mp_nbrecv(mpd_t* mpd_ptr)
: The nbrecv function checks if the next buffer, in the buffer queue, has received a complete message. If a message is received, it will move the read_buf pointer to the beginning of the message, such that the user/application can read the received data.

void mp_recv(mpd_t* mpd_ptr, const unsigned int timeout_usecs)
: The recv function calls the nb_recv function in a loop until it returns success. Put timeout_usecs at 0 if not used.

int mp_nback(mpd_t* mpd_ptr)
: The nback function increment the number of messages that has been acknowledged and sends the updated value to the sender core, if the send does not succeed the number of acknowledged messages is decremented.

void mp_ack(mpd_t* mpd_ptr, const unsigned int timeout_usecs)
: The ack function calls the nb_ack function in a loop until it returns success. Put timeout_usecs at 0 if not used.

spd_t * mp_create_sport(unsigned int chan_id, direction_t direction_type, size_t sample_size)
: The create_sport function allocates the static buffer structures of a communication channel and initializes the sampling port descriptor spd_ptr. The communication channel is set up between the writer core and the reader core. The communication channel will transfer messages of size sample_size.

int mp_write(spd_t * sport, volatile void _SPM * sample)
The write function writes the sample to the specified sampling port.

int mp_read(spd_t * sport, volatile void _SPM * sample)
The read function reads a sample from the specified sampling port and places the sample according to the sample pointer.

int mp_init_ports()
The init_ports function initializes all the created ports. All ports shall be created in the initialization phase of the program, and all cores need to call the init_ports function to initialize its local ports.

4. Exercises

The following exercises are made to run on the default 9 core platform for the Altera DE2-115 board. Please refer to the Appendix A.1 for instructions on how to build an up-to-date hardware platform.

4.1. Circulating tokens

By creating an application that mimics streaming behavior between a number of processors, this exercise will illustrate to the reader how the basics of message passing work on Argo. In this exercise, we will make an application that circulates a number of tokens in a ring of 8 slave processors. The number of tokens should be configurable, but always less than the number of processors. Each of the processors in the ring shall repeatedly execute the following 4 steps:

1. Receive a token from the previous processor
2. Turn on the processor LED to indicate that the token is being processed
3. Wait for a random amount of time in the interval [100 ms; 1 s]
4. Send the token to the next processor, when the send is complete Turn off the processor LED to indicate the token has been processed

Looking at the LEDs when the application runs, the reader should see tokens move from one LED to the other. This behavior should be easy to observe with only a few tokens. This exercise is split into 4 tasks:

1. Create a function that blinks an LED and create a thread on each slave core that executes the blink function
2. Extend the blink function to turn the LED on and off at random times
3. Extend the blink function to receive a message from the previous core in the ring and send a message to the next core in the ring
4. Change the blink function such that it sends the random seed value along with the token

In each task you should verify that your program is working as expected by compiling and downloading it to the platform. Figure 4.1 shows the libraries to include in your program and the definition of the NoC master core as core 0. Moreover, it shows some useful functions to get information related to the multicore platform.

4.1.1. Task 1

In this task, you should create a function that blinks the LED and execute the function on the slave processors. The frequency of blinking the LED should be in the order of 1 - 10 Hz so that it is visible to the eye. Figure 4.2 shows an example of how to blink an LED, where the frequency of the blinking is set through a parameter of the blinking function. To turn the LED on and off, write a 1 and 0, respectively to the hardware address of the LED.

```
const int NOC_MASTER = 0;
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <machine/patmos.h>
#include "libcorethread/corethread.h"
#include "libmp/mp.h"

get_cpucnt(); // returns the number of cores
get_cpuid(); // returns the core ID
```

Figure 4.1.: Libraries to include, definition of the NoC master, and some useful functions.

```
//blink function, period=0 -> ~10Hz, period=255 -> ~1Hz
void blink(int period) {
    // The hardware address of the LED
    #define LED ( *( ( volatile _IODEV unsigned * ) 0xF0090000 ) )

    for (;;)
    {
        for (int i=400000+14117*period; i!=0; --i){LED = 1;}
        for (int i=400000+14117*period; i!=0; --i){LED = 0;}
    }
    return;
}
```

Figure 4.2.: An example of a function to blink a LED with the period as parameter.

To execute the `blink()` function on the slave core there is an example of how to call the `corethread_create()` function in Figure 4.3. Section 3.1 explains in further detail, how corethreads are started on slave processors and how a parameter can be passed to the function.

Expected output The 8 LEDs on the board should all blink with the specified frequency.

4.1.2. Task 2

In task 2 you shall extend the blink function from task 1 to turn the LED on and off at random times. We suggest to use the `rand_r()` function to generate a random number, `rand_r()` takes a pointer to a seed value in order to generate a random number. Do not use the `rand()` function as it is not thread-safe. The seed value in each core should be different; otherwise, all cores have the same sequence of pseudo-random numbers.

Use the lower bits of the random number to generate a number on the desired range. The `get_cpu_usecs()` function returns the value of the microsecond counter as an `unsigned long long`.

Expected output The 8 LEDs should now independently blink with random varying frequencies.

```
void loop(void* arg) {
  int num_tokens = *((int*)arg);
  /*
    Write code in the slave loop
  */
}

int main() {
  int worker_id = 1; // The core ID
  int parameter = 42;
  corethread_create( worker_id, &loop, (void*) &parameter );

  int* res;
  corethread_join( worker_id, &res ); // No return value is returned

  return *res;
}
```

Figure 4.3.: An example of how to create a corethread.

4.1.3. Task 3

In this task, you will start sending messages in order to move the tokens between the slave cores. The use of the message passing function is described in Section 3.3. The initialization of the message passing channels shall be done in the slave threads, and before messages can be sent or received, each slave needs to initialize the message passing channels with the `mp_chan_init()` function. Figure 4.4 shows an example of how slave core 1 opens a source port (to send) towards core 2 and how slave core 2 opens a sink port (to receive) form core 1, creating a communication channel identified by the id 1 (first parameter in the function).

Expected output It should now be observable that the tokens move between cores.

4.1.4. Task 4

For the sake of the example, you should now pair a seed value to each token. To send the seed value along with the message, you need to write the seed value into the `write_buf` before sending the message, and read out the seed value from the `read_buf` after receiving a message. Figure 4.5 shows an example of how to receive, send, acknowledge reception, read, and write message data.

Expected output It should now be observable that the tokens move between cores, like task 3 but with random intervals.

4.1.5. Extensions

If you have more time left or just can not get enough of programming message passing applications, you can extend your application in several ways:

- Move the calculation of random numbers to core 0. Core 0 shall act like a server replying with a new random number when it receives a message from any of the slave cores.
- Create a mechanism that terminates the execution of the blink function on the slaves when the master is signaled to stop thought the terminal.

```
#define MP_CHAN_NUM_BUF 2
#define MP_CHAN_BUF_SIZE 40

...

// Slave function running on core 1
void slave1(void* param) {
    // Create the port for channel 1
    qpd_t * chan1 = mp_create_qport(1, SOURCE,
    MP_CHAN_BUF_SIZE, MP_CHAN_NUM_BUF);
    mp_init_ports();

    // Do something
    return;
}

// Slave function running on core 2
void slave2(void* param) {
    // Create the port for channel 1
    qpd_t * chan1 = mp_create_qport(1, SINK,
    MP_CHAN_BUF_SIZE, MP_CHAN_NUM_BUF);
    mp_init_ports();

    // Do something
    return;
}
```

Figure 4.4.: An example of how to create a communication channel.

```
// Receiving, reading and acknowledge reception of
// an unsigned integer value from the channel read buffer
mp_recv(chan,0);
seed = *(( volatile int _SPM * ) ( chan->read_buf ));
mp_ack(chan,0);

// Writing an unsigned integer value to the channel
// write buffer and sending it.
*( volatile int _SPM * ) ( chan->write_buf ) = seed;
mp_send(chan,0);
```

Figure 4.5.: An example of how to receive, send, acknowledge reception, read, and write message data.

A. Build And Execute Instructions

In this chapter, we present the details on how to build and configure the hardware platform and compile and execute a multicore program on the platform.

A.1. Build and configure the hardware platform

The Aegean framework generates a hardware description from an XML description. The default XML description for the Altera DE2-115 board with 9 cores has an external shared memory and an Argo network-on-chip. To build the platform run the following commands:

```
cd ~/t-crest/aegean
make AEGEAN_PLATFORM=altde2-115-9core platform synth
```

The make command will generate a platform as described in the `config/altde2-115-9core.xml` file. When the platform description is generated, then it will be synthesised. When the synthesis is finished the multicore platform can be configured into the FPGA using the following commands:

```
cd ~/t-crest
make -C aegean AEGEAN_PLATFORM=altde2-115-9core config
```

If you experience problems in building the multicore platform, you may need to update your T-CREST repositories to the newest version and re-build the project with the following commands before re-executing the commands listed above:

```
cd ~/t-crest/
./misc/gitall pull
cd ~/t-crest/argo/
git pull
cd ~/t-crest/
./misc/build.sh -c
```

If you still experience problems, please send an email to *lpez@dtu.dk* (Luca Pezzarossa).

A.2. Compile and execute a multicore program

There is no difference between compiling a single core program and a multicore program. Furthermore, a single core program can execute in a multicore platform without any modifications. To compile a multicore program, place it in the `patmos/c/` directory and run the following commands:

```
cd ~/t-crest
make -C patmos APP=${APP_NAME} comp
```

The `comp` target will compile the C program in the file `patmos/c/${APP_NAME}.c` and output an .elf file `patmos/tmp/${APP_NAME}.elf`. When compiling a program that includes either "`libmp/mp.h`" or "`libnoc/noc.h`", the `nocinit.c`, generated by the Aegean framework, is included needed, as this contains the configuration data for the Argo NoC. To download the program to the configured FPGA, run the following commands:

```
cd ~/t-crest
make -C patmos APP=${APP_NAME} download
```

The `download` target of the Makefile depends on the comp target, therefore it is not necessary to execute the `comp` target before every download. Also, it is not strictly necessary to configure the FPGA with the hardware platform between each download of a program, but we advise you to do so. This will ensure that the hardware platform is probably initialized before you download a program.

Bibliography

[1] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, and J. Sparsø. Argo: A real-time network-on-chip architecture with an efficient GALS implementation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 24:479–492, 2016.

[2] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[3] M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, and D. Prokesch. Patmos reference handbook. Technical report, 2014.