The Argo software perspective

A multicore programming exercise

Rasmus Bo Sørensen

Updated by Luca Pezzarossa

March 2, 2017

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 the 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

Preface iii

1. Introduction 1

2. The Architecture of Argo 3

3. Application Programming Interface 5
3.1. Corethread Library . . . 5
3.2. NoC Driver . . . 5
3.3. Message Passing Library . . . 6

4. Exercises 9
4.1. Circulating tokens . . . 9
4.1.1. Task 1 . . . 9
4.1.2. Task 2 . . . 10
4.1.3. Task 3 . . . 11
4.1.4. Task 4 . . . 11
4.1.5. Extensions . . . 11

A. Build And Execute Instructions 13
A.1. Build and configure the hardware platform . . . 13
A.2. Compile and execute a multicore program . . . 13

B. Library Documentation 15
B.1. Module Documentation . . . 15
B.1.1. Libcorethread . . . 15
B.1.2. Coreset . . . 16
B.1.3. Libnoc . . . 18
B.1.4. Libmp . . . 23
B.2. Class Documentation . . . 27
B.2.1. coreset_t Struct Reference . . . 27
B.2.2. qpd_t Struct Reference . . . 27
B.2.3. spd_t Struct Reference . . . 27
B.3. File Documentation . . . 27
B.3.1. coreset.h File Reference . . . 27
B.3.2. corethread.h File Reference . . . 28
B.3.3. mp.h File Reference . . . 29
B.3.4. noc.h File Reference . . . 31

Bibliography 35

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 quiet low. Therefore, the programmer should utilize the NoC as much as possible for core-to-core communication.

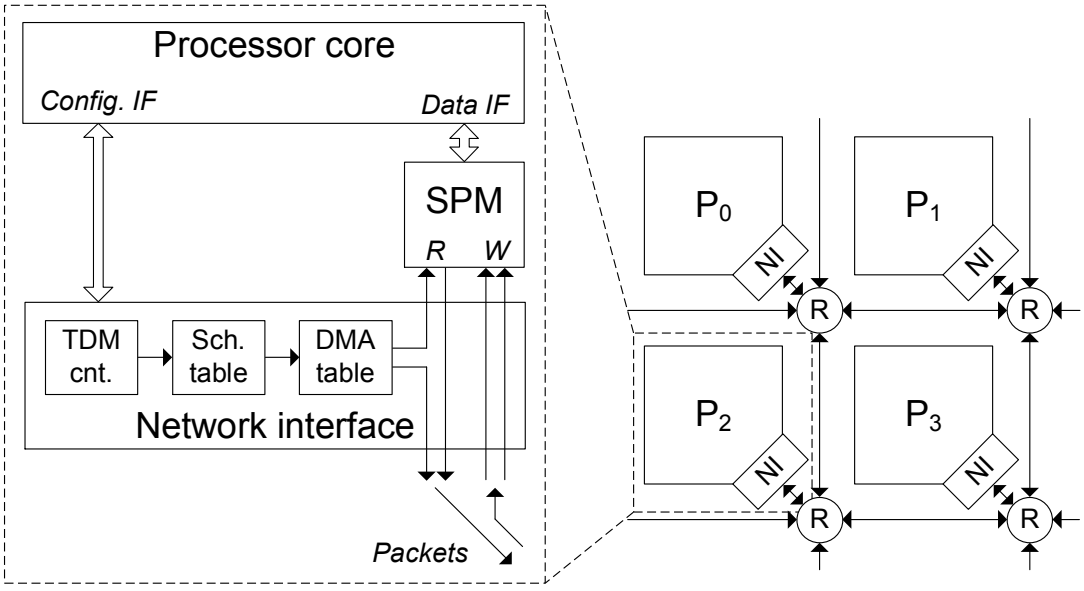

Figure 1.1.: The Patmos multicore platform with the Argo NoC for intercore communication. The core with id 0 is referred to as the master core and the rest of the cores are referred to as the slave cores.

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. Appendix A describes the practical aspects of loading the program into the platform running in an FPGA. Appendix B contains the Doxygen documentation of the C libraries.

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 run time 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

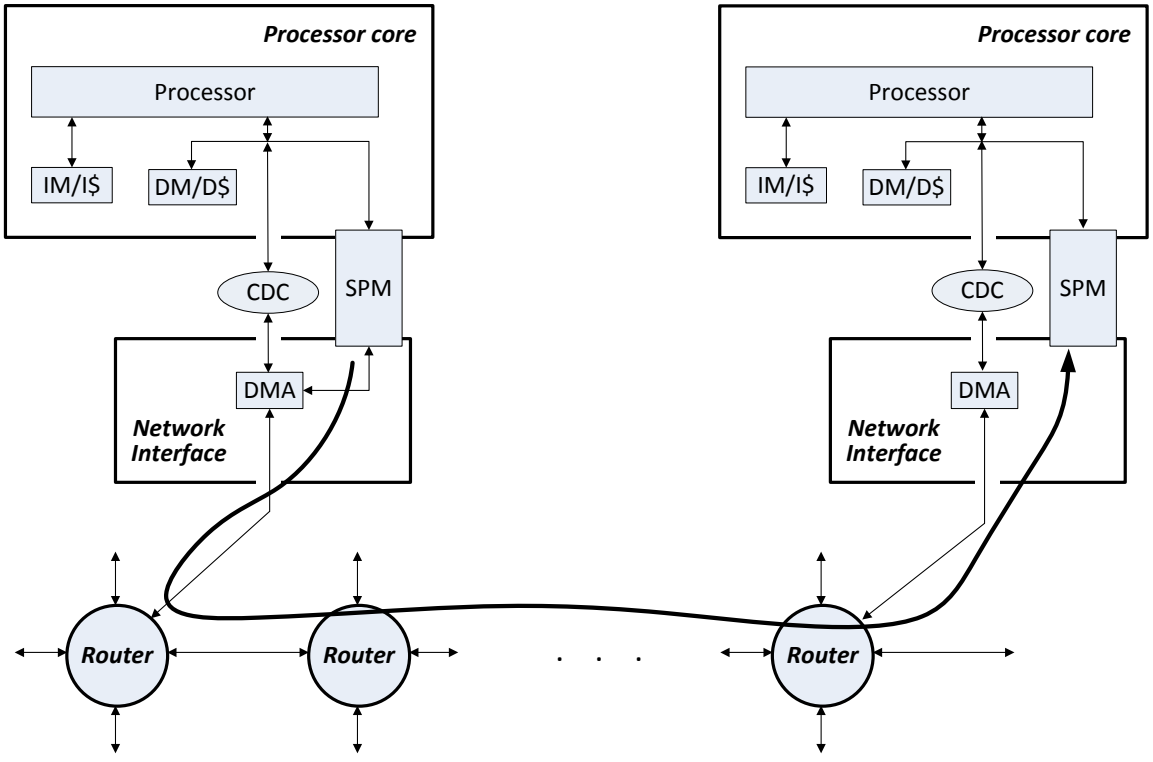

Figure 2.1.: The Argo architecture from a software perspective. A DMA write transaction moves the specified block of data from the communication SPM of the processor on the left to the specified location in the communication SPM of the processor on the right.

entries, each entry describes a DMA controller that can send to a remote processor. Each DMA controller is paired with a communication channel when the network in 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 transfered 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`. We give an overview of the three libraries in the following three section. For detailed documentation refer to Appendix B, which contains the doxygen documentation of the three C 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( corethread_t *thread, void(*start_routine)(void *), void *arg ) )`
The create function will start the execution of the `start_routine` function on the core specified by `thread`, 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( corethread_t thread, void ** retval )`
The join function will join the program flow of the master core with the program flow of the core specified by `thread`, 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 om 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 has 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 that he/she does not allocate more local memory than 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 into the write_buf buffer.

void mp_send(mpd_t* mpd_ptr)
The send function calls the nb_send function in a loop, until it returns success.

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)
The recv function calls the nb_recv function in a loop, until it returns success.

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)
The ack function calls the nb_ack function in a loop, until it returns success.

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 needs 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 works 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 for 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 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 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 the your program is working as expected by compiling and downloading the program 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 slaves 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() {
  corethread_t 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 tokens. 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 mechanisme that terminates the executution 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=default-altde2-115-9core platform synth
```

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

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

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 advice you to do so. This will ensure that the hardware platform is probably initialized before your download a program.

B. Library Documentation

B.1. Module Documentation

B.1.1. Libcorethread

Files

- file corethread.h

  Corethread library for the T-CREST platform.

Macros

- #define EAGAIN 1

  Resource unavailable.
- #define EINVAL 2

  Invalid argument.
- #define EPERM 3

  Operation not permitted.
- #define ESRCH 4

  No such resource.
- #define EDEADLK 5

  Resource deadlock avoided.

Typedefs

- typedef size_t corethread_t

  An type to describe a corethread.

Functions

- int corethread_create (corethread_t ∗thread, void(∗start_routine)(void ∗), void ∗arg)

  Creates a corethread on the core with the COREID equal to the id specified by thread.
- void corethread_exit (void ∗retval)

  The last function to be called by a terminating thread.
- int corethread_join (corethread_t thread, void ∗∗retval)

  The caller waits for the corethread, specified by thread, to write a return value and terminate exection.

Variables

- const int NOC_MASTER

  The master core, which governs booting and startup synchronization.

Detailed Description

Function Documentation

int corethread_create (corethread_t ∗ *thread,* void(∗)(void ∗) *start_routine,* void ∗ *arg*) Creates a corethread on the core with the COREID equal to the id specified by thread.

Parameters

thread	A pointer to the corethread_t specifying the thread to start
start_routine	A function pointer to the function that the created thread should start executing
arg	A pointer to the argument to pass to the start_routine function

Return values

0	The thread was created
EAGAIN	The corethread is already allocated
EINVAL	The attribute value is invalid
EPERM	The caller does not have appropriate permissions the set the required scheduling parameters or scheduling policy

void corethread_exit (void * *retval*) The last function to be called by a terminating thread.

The running corethread terminates with the given returnvalue

Parameters

retval	A pointer to the return value that the calling thread shall return.

int corethread_join (corethread_t *thread,* void ** *retval*) The caller waits for the corethread, specified by thread, to write a return value and terminate exection.

Parameters

thread	The thread struct belonging to the terminating corethread.
retval	A pointer to the pointer that is to point to the return value of the terminating thread.

Return values

0	The specified thread was joined.
EINVAL	The given corethread can not be joined.
ESRCH	No corethread exist with the specified corethread ID.
EDEADLK	A deadlock was detected or the specified corethread is the calling thread

B.1.2. Coreset

Files

- file coreset.h

  Functions to manipulate sets of cores.

Classes

- struct coreset_t

  An opaque type to describe a set of cores.

Macros

- #define CORESET_SIZE 32

  The maximum number of cores supported by the library. May be changed by defining it before including coreset.h. Should be a power of 2.

Functions

- static void coreset_clearall (coreset_t *set)

  Remove all cores from the set.
- static void coreset_add (unsigned core, coreset_t *set)

  Add a core to the set.
- static void coreset_remove (unsigned core, coreset_t *set)

  Remove a core from the set.
- static int coreset_contains (unsigned core, const coreset_t *set)

  Determins whether the set contains the core.
- static int coreset_empty (const coreset_t *set)

  Determins whether the set is empty.

Detailed Description

Function Documentation

static void coreset_add (unsigned *core,* coreset_t * *set*) [inline], [static] Add a core to the set.
Parameters

core	A core number.
set	A set of cores.

static void coreset_clearall (coreset_t * *set*) [inline], [static] Remove all cores from the set.
Parameters

set	A set of cores.

static int coreset_contains (unsigned *core,* const coreset_t * *set*) [inline], [static] Determins whether the set contains the core.
Parameters

core	A core number.
set	A set of cores.

Returns

Non-zero if a core is in the set, zero otherwise.

static int coreset_empty (const coreset_t * *set*) [inline], [static] Determins whether the set is empty.
Parameters

set	A set of cores.

Returns

Non-zero if the coreset is empty, zero otherwise.

static void coreset_remove (unsigned *core,* coreset_t * *set*) [inline], [static] Remove a core from the set.

Parameters

core	A core number.
set	A set of cores.

B.1.3. Libnoc

Files

- file noc.h

  Low-level NoC communication library for the T-CREST platform.

Macros

- #define DEBUGGER(...)

  Print message if (a) DEBUG is defined and (b) executing on master core.
- #define DEBUG_CORECHECK(x)

  Abort if (a) DEBUG is defined and (b) condition is true.
- #define **DATA_PKT_TYPE** 0
- #define **DATA_IRQ_PKT_TYPE** 2
- #define **CONFIG_PKT_TYPE** 1
- #define **IRQ_PKT_TYPE** 3
- #define **NOC_PTR_WIDTH** 14
- #define 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.
- #define NOC_ACTIVE_BIT 0x80000000

  The flag to mark a DMA entry as valid.
- #define **OFFSET_WIDTH** (11+2)
- #define **BANK**(ID) (ID<<OFFSET_WIDTH)
- #define **DMA_BANK** BANK(0)
- #define **SCHED_BANK** BANK(1)
- #define **TDM_BANK** BANK(2)
- #define **MC_BANK** BANK(3)
- #define **IRQ_BANK** BANK(4)
- #define NOC_DMA_BASE ((volatile unsigned int _IODEV *)(0xE0000000+DMA_BANK))

  The base address for DMA entries.
- #define NOC_SCHED_BASE ((volatile unsigned int _IODEV *)(0xE0000000+SCHED_BANK))

  The base address for DMA routing information.
- #define NOC_TDM_BASE ((volatile unsigned int _IODEV *)(0xE0000000+TDM_BANK))

  The base address for the slot table.
- #define NOC_MC_BASE ((volatile unsigned int _IODEV *)(0xE0000000+MC_BANK))

  The base address for the slot table.
- #define NOC_IRQ_BASE ((volatile unsigned int _IODEV *)(0xE0000000+IRQ_BANK))

  The base address for the slot table.
- #define NOC_SPM_BASE ((volatile unsigned int _SPM *)0xE8000000)

  The base address of the communication SPM.

Functions

- void noc_sched_set (unsigned char config)

  Loads the configuration data from the noc_init_array.
- void noc_enable (void)

  Enables the TDM schedule in the NoC.
- void noc_disable (void)

  Disables the TDM schedule in the NoC.
- int noc_configure (void)

  Configure network interface according to initialization information in noc_init_array.
- int noc_fifo_irq_read (void)

  Reads out an element of the fifo with remote irq's.
- int noc_fifo_data_read (void)

  Reads out an element of the fifo with data irq's.
- void noc_init (void)

  Configure network-on-chip and synchronize all cores.
- void **__remote_irq_handler** (void)
- void **__data_recv_handler** (void)
- void **__noc_trap_handler** (void)
- int noc_conf (unsigned dma_id, volatile void _SPM ∗dst, volatile void _SPM ∗src, size_t size)

  Start a NoC configure transfer.
- int noc_irq (unsigned dma_id, volatile void _SPM ∗dst, volatile void _SPM ∗src)

  Start a NoC interrupt.
- int noc_dma_done (unsigned dma_id)

  Check if a NoC transfer has finished.
- void noc_dma_clear (unsigned dma_id)

  Stops a NoC transfer and clear the DMA entry.
- int noc_nbwrite (unsigned dma_id, volatile void _SPM ∗dst, volatile void _SPM ∗src, size_t size, unsigned irq_enable)

  Attempt to transfer data via the NoC (non-blocking).
- void noc_write (unsigned dma_id, volatile void _SPM ∗dst, volatile void _SPM ∗src, size_t size, unsigned irq_enable)

  Transfer data via the NoC (blocking).
- void noc_multisend (unsigned cnt, unsigned dma_id[ ], volatile void _SPM ∗dst[ ], volatile void _SPM ∗src, size_t size, unsigned irq_enable)

  Multi-cast transfer of data via the NoC (blocking).
- void noc_multisend_cs (coreset_t ∗receivers, volatile void _SPM ∗dst[ ], unsigned offset, volatile void _SPM ∗src, size_t size, unsigned irq_enable)

  Multi-cast transfer of data like noc_multisend(), but with coreset and a single destination address.
- void noc_wait_dma (coreset_t receivers)

  Wait until all transfers to a set of receivers have finished.

Variables

- const int NOC_CORES

  The number of cores on the platform.
- const int NOC_CONFS

  The number of configurations in the noc_init_array.
- const int NOC_TABLES

  The number of tables for NoC configuration.
- const int NOC_SCHEDULE_ENTRIES

  The number of entries in the schedule table.
- const int noc_init_array [ ]

  The array for initialization data.
- const int NOC_MASTER

  The master core, which governs booting and startup synchronization.

Detailed Description

Function Documentation

int noc_conf (unsigned *dma_id,* volatile void _SPM ∗ *dst,* volatile void _SPM ∗ *src,* size_t *size*) Start a NoC configure transfer.

The addresses and the size are in words and relative to the communication SPM base NOC_SPM_BASE.

Parameters

dma_id	The core id of the receiver.
write_ptr	The address in the receiver's communication SPM, in words, relative to NOC_SPM_BASE.
read_ptr	The address in the sender's communication SPM, in words, relative to NOC_SPM_BASE.
size	The size of data to be transferred, in words.

Return values

1	Sending was successful.
0	Otherwise.

void noc_disable (void) Disables the TDM schedule in the NoC.

When the function returns, all NIs in the platform are quiet and do not transmit any packets through the NoC.

void noc_dma_clear (unsigned *dma_id*) Stops a NoC transfer and clear the DMA entry.

Parameters

dma_id	The core id of the receiver.

int noc_dma_done (unsigned *dma_id*) Check if a NoC transfer has finished.

Parameters

dma_id	The core id of the receiver.

Return values

1	The transfer has finished.
0	Otherwise.

void noc_enable (void) Enables the TDM schedule in the NoC.
When the function returns all NIs in the platform execute the configured schedule.

void noc_init (void) Configure network-on-chip and synchronize all cores.
noc_init is a static constructor and not intended to be called directly.

int noc_irq (unsigned *dma_id,* volatile void _SPM * *dst,* volatile void _SPM * *src*) Start a NoC interrupt.
The addresses and the size are in words and relative to the communication SPM base NOC_SPM_BASE.
Parameters

dma_id	The core id of the receiver.
write_ptr	The address in the receiver's communication SPM, in words, relative to NOC_SPM_BASE.
read_ptr	The address in the sender's communication SPM, in words, relative to NOC_SPM_BASE.

Return values

1	Sending was successful.
0	Otherwise.

void noc_multisend (unsigned *cnt,* unsigned *dma_id[],* volatile void _SPM * *dst[],* volatile void _SPM * *src,* size_t *size,* unsigned *irq_enable*) Multi-cast transfer of data via the NoC (blocking).
The addresses and the size are absolute and in bytes.
Parameters

cnt	The number of receivers.
dma_id	An array with the core ids of the receivers.
dst	An array with pointers to the destinations of the transfer.
src	A pointer to the source of the transfer.
size	The size of data to be transferred, in bytes.

void noc_multisend_cs (coreset_t * *receivers,* volatile void _SPM * *dst[],* unsigned *offset,* volatile void _SPM * *src,* size_t *size,* unsigned *irq_enable*) Multi-cast transfer of data like noc_multisend(), but with coreset and a single destination address.
The addresses and the size are absolute and in bytes.
Parameters

receivers	The set of receivers.
dst	An array with pointers to the destinations of the transfer.
offset	Common offset for the destination addresses.
src	A pointer to the source of the transfer.
size	The size of data to be transferred, in bytes.

int noc_nbwrite (unsigned *dma_id,* volatile void _SPM * *dst,* volatile void _SPM * *src,* size_t *size,* unsigned *irq_enable*) Attempt to transfer data via the NoC (non-blocking).
The addresses and the size are absolute and in bytes.

Parameters

dma_id	The core id of the receiver.
dst	A pointer to the destination of the transfer.
src	A pointer to the source of the transfer.
size	The size of data to be transferred, in bytes.
irq_enable	If irq_enable is 1 an interrupt will be triggered at the receiver when the whole transfer is complete, of it is zero no interrupt will be triggered at the receiver.

Return values

1	Sending was successful.
0	Otherwise.

void noc_wait_dma (coreset_t *receivers*) Wait until all transfers to a set of receivers have finished.

Parameters

receivers	The set of receivers.

void noc_write (unsigned *dma_id,* volatile void _SPM * *dst,* volatile void _SPM * *src,* size_t *size,* unsigned *irq_enable*) Transfer data via the NoC (blocking).

The addresses and the size are absolute and in bytes.

Parameters

dma_id	The core id of the receiver.
dst	A pointer to the destination of the transfer.
src	A pointer to the source of the transfer.
size	The size of data to be transferred, in bytes.
irq_enable	If irq_enable is 1 an interrupt will be triggered at the receiver when the whole transfer is complete, of it is zero no interrupt will be triggered at the receiver.

Variable Documentation

const int NOC_CONFS The number of configurations in the noc_init_array.

Typically generated by Poseidon.

AUTO-Generated file DO NOT EDIT!!! Loads the pre calculated schedule into the Slot and Route tables.

const int NOC_CORES The number of cores on the platform.

Typically generated by Poseidon.

AUTO-Generated file DO NOT EDIT!!! Loads the pre calculated schedule into the Slot and Route tables.

const int noc_init_array[] The array for initialization data.

Typically generated by Poseidon.

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

Typically defined by the application.

const int NOC_SCHEDULE_ENTRIES The number of entries in the schedule table.

Typically generated by Poseidon.

const int NOC_TABLES The number of tables for NoC configuration.

Typically generated by Poseidon.

B.1.4. Libmp

Files

- file mp.h

  Message passing library for the T-CREST platform.

Classes

- struct _SPM_LOCK_T
- struct _qpd_t
- struct _spd_t
- struct LOCK_T

  Lock type placed in local scratchpad memory.
- struct qpd_t

  Queuing port descriptor.
- struct spd_t

  Sample port descriptor.

Macros

- #define MAX_CHANNELS 32

  Aligns X to word size.
- #define **INLINING** __attribute__ ((always_inline))

Typedefs

- typedef char coreid_t

  A type to identify a core. Supports up to 256 cores in the platform.
- typedef struct _SPM_LOCK_T _SPM **LOCK_T**
- typedef struct _qpd_t _SPM **qpd_t**
- typedef struct _spd_t _SPM **spd_t**

Enumerations

- enum **direction_t** { **SOURCE**, **SINK** }

Functions

- LOCK_T * **initialize_lock** (unsigned remote)
- void **acquire_lock** (LOCK_T *lock) INLINING
- void **release_lock** (LOCK_T *lock) INLINING
- void mp_init (void)

  Initialize message passing library.
- void _SPM * mp_alloc (const size_t size)

  Static memory allocation on the communication scratchpad. No mp_free function.
- qpd_t * mp_create_qport (const unsigned int chan_id, const direction_t direction_type, const size_t msg_size, const size_t num_buf)

  Initialize the state of a communication channel.
- spd_t * mp_create_sport (const unsigned int chan_id, const direction_t direction_type, const size_t sample_size)

  Initialize the state of a communication channel.

- int mp_init_ports ()
- int mp_nbsend (qpd_t ∗qpd_ptr) INLINING

  Non-blocking function for passing a message to a remote processor under flow control. The data to be passed by the function should be in the local buffer in the communication scratch pad before the function is called.
- int mp_send (qpd_t ∗qpd_ptr, const unsigned int time_usecs) INLINING

  A function for passing a message to a remote processor under flow control. The data to be passed by the function should be in the local buffer in the communication scratch pad before the function is called.
- int mp_nbrecv (qpd_t ∗qpd_ptr) INLINING

  Non-blocking function for receiving a message from a remote processor under flow control. The data that is received is placed in a message buffer in the communication scratch pad, when the received message is no longer used the reception of the message should be acknowledged with the mp_ack()
- int mp_recv (qpd_t ∗qpd_ptr, const unsigned int time_usecs) INLINING

  A function for receiving a message from a remote processor under flow control. The data that is received is placed in a message buffer in the communication scratch pad, when the received message is no longer used the reception of the message should be acknowledged with the mp_ack()
- int mp_nback (qpd_t ∗qpd_ptr) INLINING

  Non-blocking function for acknowledging the reception of a message. This function should be used with extra care, if no acknowledgement is sent the communication channel will be blocked until an acknowledgement is sent. This function shall be called to release space in the receiving buffer when the received data is no longer used. It is not necessary to call mp_ack() after each mp_recv() call. It is possible to work on 2 or more incoming messages at the same time with out them being overwritten.
- int mp_ack (qpd_t ∗qpd_ptr, const unsigned int time_usecs) INLINING

  A function for acknowledging the reception of a message. This function shall be called to release space in the receiving buffer when the received data is no longer used. It is not necessary to call mp_ack() after each mp_recv() call. It is possible to work on 2 or more incoming messages at the same time with out them being overwritten.
- int **mp_ack_n** (qpd_t ∗qpd_ptr, const unsigned int time_usecs, unsigned int num_acks) INLINING
- int mp_write (spd_t ∗sport, volatile void _SPM ∗sample)

  A function for writing a sampled value to the remote location at the receiving end of the channel.
- int mp_read (spd_t ∗sport, volatile void _SPM ∗sample)

Detailed Description

Function Documentation

int mp_ack (qpd_t ∗ *qpd_ptr,* const unsigned int *time_usecs*) A function for acknowledging the reception of a message. This function shall be called to release space in the receiving buffer when the received data is no longer used. It is not necessary to call mp_ack() after each mp_recv() call. It is possible to work on 2 or more incoming messages at the same time with out them being overwritten.

Parameters

qpd_ptr	A pointer to the message passing data structure for the given message passing channel.
time_usecs	The time out time in microseconds, if parameter is 0 the timeout is infinite

Return values

0	The function timed out.
1	The function succeeded acknowledging the message.

qpd_t∗ mp_create_qport (const unsigned int *chan_id,* const direction_t *direction_type,* const size_t *msg_size,* const size_t *num_buf*) Initialize the state of a communication channel.

Parameters

qpd_ptr	A pointer the the message passing descriptor
remote	The core id of the remote processor
buf_size	The size of the message buffer
num_buf	The number of buffers in the receiving scratchpad

Returns

The function returns a pointer to the created message passing descriptor qpd_t. If the function fails, the pointer is NULL. Remember to check if the returned pointer is different from NULL.

spd_t* mp_create_sport (const unsigned int *chan_id,* const direction_t *direction_type,* const size_t *sample_size*) Initialize the state of a communication channel.

Parameters

qpd_ptr	A pointer the the message passing descriptor
remote	The core id of the remote processor
buf_size	The size of the message buffer
num_buf	The number of buffers in the receiving scratchpad

Returns

The function returns a pointer to the created sampling port descriptor spd_t. If the function fails, the pointer is NULL. Remember to check if the returned pointer is different from NULL.

void mp_init (void) Initialize message passing library.

mp_init is a static constructor and not intended to be called directly.

int mp_init_ports ( ) Initializing all the channels that have been registered.

Return values

0	The initialization of one or more communication channels failed.
1	The initialization of all the communication channels succeeded.

int mp_nback (qpd_t * *qpd_ptr*) Non-blocking function for acknowledging the reception of a message. This function should be used with extra care, if no acknowledgement is sent the communication channel will be blocked until an acknowledgement is sent. This function shall be called to release space in the receiving buffer when the received data is no longer used. It is not necessary to call mp_ack() after each mp_recv() call. It is possible to work on 2 or more incoming messages at the same time with out them being overwritten.

Parameters

qpd_ptr	A pointer to the message passing data structure for the given message passing channel.

Return values

0	No acknowledgement has been sent.
1	An acknowledgement has been sent.

int mp_nbrecv (qpd_t * *qpd_ptr*) Non-blocking function for receiving a message from a remote processor under flow control. The data that is received is placed in a message buffer in the communication scratch pad, when the received message is no longer used the reception of the message should be acknowledged with the mp_ack()

Parameters

qpd_ptr	A pointer to the message passing data structure for the given message passing channel.

Return values

0	No message has been received yet.
1	A message has been received and dequeued. The call has to be followed by a call to mp_ack() when the data is no longer used.

int mp_nbsend (qpd_t * *qpd_ptr*) Non-blocking function for passing a message to a remote processor under flow control. The data to be passed by the function should be in the local buffer in the communication scratch pad before the function is called.

Parameters

qpd_ptr	A pointer to the message passing data structure for the given message passing channel.

Return values

0	The send did not succeed, either there was no space in the receiving buffer or there was no free DMA to start a transfer
1	The send succeeded.

int mp_read (spd_t * *sport,* volatile void _SPM * *sample*) A function for reading a sampled value from the remote location at the sending end of the channel

int mp_recv (qpd_t * *qpd_ptr,* const unsigned int *time_usecs*) A function for receiving a message from a remote processor under flow control. The data that is received is placed in a message buffer in the communication scratch pad, when the received message is no longer used the reception of the message should be acknowledged with the mp_ack()

Parameters

qpd_ptr	A pointer to the message passing data structure for the given message passing channel.
time_usecs	The time out time in microseconds, if parameter is 0 the timeout is infinite

Return values

0	The function timed out.
1	The function succeeded receiving the message.

int mp_send (qpd_t * *qpd_ptr,* const unsigned int *time_usecs*) A function for passing a message to a remote processor under flow control. The data to be passed by the function should be in the local buffer in the communication scratch pad before the function is called.

Parameters

qpd_ptr	A pointer to the message passing data structure for the given message passing channel.
time_usecs	The time out time in microseconds, if parameter is 0 the timeout is infinite

Return values

0	The function timed out.

1	The function succeeded sending the message.

B.2. Class Documentation

B.2.1. coreset_t Struct Reference

An opaque type to describe a set of cores.
`#include <coreset.h>`

Detailed Description

An opaque type to describe a set of cores.

The documentation for this struct was generated from the following file:

- coreset.h

B.2.2. qpd_t Struct Reference

Queuing port descriptor.
`#include <mp.h>`

Detailed Description

Queuing port descriptor.

The struct is used to store the data describing the massage passing channel. This struct is used to describe both the sending and receiving ends of a communication channel.

The documentation for this struct was generated from the following file:

- mp.h

B.2.3. spd_t Struct Reference

Sample port descriptor.
`#include <mp.h>`

Detailed Description

Sample port descriptor.

The struct is used to store the data describing the sampling port. This struct is used to describe both the writer and reader of the sampling port.

The documentation for this struct was generated from the following file:

- mp.h

B.3. File Documentation

B.3.1. coreset.h File Reference

Functions to manipulate sets of cores.

Classes

- struct coreset_t

  An opaque type to describe a set of cores.

Macros

- #define CORESET_SIZE 32

  The maximum number of cores supported by the library. May be changed by defining it before including coreset.h. Should be a power of 2.

Functions

- static void coreset_clearall (coreset_t *set)

  Remove all cores from the set.
- static void coreset_add (unsigned core, coreset_t *set)

  Add a core to the set.
- static void coreset_remove (unsigned core, coreset_t *set)

  Remove a core from the set.
- static int coreset_contains (unsigned core, const coreset_t *set)

  Determins whether the set contains the core.
- static int coreset_empty (const coreset_t *set)

  Determins whether the set is empty.

Detailed Description

Functions to manipulate sets of cores. Definitions for sets of cores as used by libnoc.

Author

Wolfgang Puffitsch `wpuffitsch@gmail.com`

B.3.2. corethread.h File Reference

Corethread library for the T-CREST platform.

```
  #include <machine/patmos.h>
#include <machine/spm.h>
#include <stdlib.h>
#include <machine/rtc.h>
#include <machine/boot.h>
```

Macros

- #define EAGAIN 1

  Resource unavailable.
- #define EINVAL 2

  Invalid argument.
- #define EPERM 3

  Operation not permitted.
- #define ESRCH 4

  No such resource.
- #define EDEADLK 5

  Resource deadlock avoided.

Typedefs

- typedef size_t corethread_t

  An type to describe a corethread.

Functions

- int corethread_create (corethread_t ∗thread, void(∗start_routine)(void ∗), void ∗arg)

  Creates a corethread on the core with the COREID equal to the id specified by thread.
- void corethread_exit (void ∗retval)

  The last function to be called by a terminating thread.
- int corethread_join (corethread_t thread, void ∗∗retval)

  The caller waits for the corethread, specified by thread, to write a return value and terminate exection.

Variables

- const int NOC_MASTER

  The master core, which governs booting and startup synchronization.

Detailed Description

Corethread library for the T-CREST platform. Definitions for libcorethread.

Author

Rasmus Bo Soerensen `rasmus@rbscloud.dk`

B.3.3. mp.h File Reference

Message passing library for the T-CREST platform.

```
  #include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <machine/patmos.h>
#include <machine/spm.h>
#include <machine/boot.h>
#include <machine/rtc.h>
#include "libnoc/noc.h"
#include "libnoc/coreset.h"
#include "include/debug.h"
```

Classes

- struct _SPM_LOCK_T
- struct _qpd_t
- struct _spd_t

Macros

- #define MAX_CHANNELS 32

  Aligns X to word size.
- #define **INLINING** __attribute__ ((always_inline))

Typedefs

- typedef char coreid_t

  A type to identify a core. Supports up to 256 cores in the platform.
- typedef struct _SPM_LOCK_T _SPM **LOCK_T**
- typedef struct _qpd_t _SPM **qpd_t**
- typedef struct _spd_t _SPM **spd_t**

Enumerations

- enum **direction_t** { **SOURCE**, **SINK** }

Functions

- LOCK_T ∗ **initialize_lock** (unsigned remote)
- void **acquire_lock** (LOCK_T ∗lock) INLINING
- void **release_lock** (LOCK_T ∗lock) INLINING
- void mp_init (void)

  Initialize message passing library.
- void _SPM ∗ mp_alloc (const size_t size)

  Static memory allocation on the communication scratchpad. No mp_free function.
- qpd_t ∗ mp_create_qport (const unsigned int chan_id, const direction_t direction_type, const size_t msg_size, const size_t num_buf)

  Initialize the state of a communication channel.
- spd_t ∗ mp_create_sport (const unsigned int chan_id, const direction_t direction_type, const size_t sample_-size)

  Initialize the state of a communication channel.
- int mp_init_ports ()
- int mp_nbsend (qpd_t ∗qpd_ptr) INLINING

  Non-blocking function for passing a message to a remote processor under flow control. The data to be passed by the function should be in the local buffer in the communication scratch pad before the function is called.
- int mp_send (qpd_t ∗qpd_ptr, const unsigned int time_usecs) INLINING

  A function for passing a message to a remote processor under flow control. The data to be passed by the function should be in the local buffer in the communication scratch pad before the function is called.
- int mp_nbrecv (qpd_t ∗qpd_ptr) INLINING

  Non-blocking function for receiving a message from a remote processor under flow control. The data that is received is placed in a message buffer in the communication scratch pad, when the received message is no longer used the reception of the message should be acknowledged with the mp_ack()
- int mp_recv (qpd_t ∗qpd_ptr, const unsigned int time_usecs) INLINING

  A function for receiving a message from a remote processor under flow control. The data that is received is placed in a message buffer in the communication scratch pad, when the received message is no longer used the reception of the message should be acknowledged with the mp_ack()
- int mp_nback (qpd_t ∗qpd_ptr) INLINING

  Non-blocking function for acknowledging the reception of a message. This function should be used with extra care, if no acknowledgement is sent the communication channel will be blocked until an acknowledgement is sent. This function shall be called to release space in the receiving buffer when the received data is no longer used. It is not necessary to call mp_ack() after each mp_recv() call. It is possible to work on 2 or more incoming messages at the same time with out them being overwritten.
- int mp_ack (qpd_t ∗qpd_ptr, const unsigned int time_usecs) INLINING

  A function for acknowledging the reception of a message. This function shall be called to release space in the receiving buffer when the received data is no longer used. It is not necessary to call mp_ack() after each mp_recv() call. It is possible to work on 2 or more incoming messages at the same time with out them being overwritten.

- int **mp_ack_n** (qpd_t *qpd_ptr, const unsigned int time_usecs, unsigned int num_acks) INLINING
- int mp_write (spd_t *sport, volatile void _SPM *sample)

  A function for writing a sampled value to the remote location at the receiving end of the channel.
- int mp_read (spd_t *sport, volatile void _SPM *sample)

Detailed Description

Message passing library for the T-CREST platform. Definitions for libmp.

Author

Rasmus Bo Soerensen `rasmus@rbscloud.dk`

B.3.4. noc.h File Reference

Low-level NoC communication library for the T-CREST platform.

```
  #include <machine/patmos.h>
#include <machine/spm.h>
#include "coreset.h"
#include <machine/boot.h>
#include <machine/exceptions.h>
#include <stdlib.h>
#include <stdio.h>
```

Macros

- #define DEBUGGER(...)

  Print message if (a) DEBUG is defined and (b) executing on master core.
- #define DEBUG_CORECHECK(x)

  Abort if (a) DEBUG is defined and (b) condition is true.
- #define **DATA_PKT_TYPE** 0
- #define **DATA_IRQ_PKT_TYPE** 2
- #define **CONFIG_PKT_TYPE** 1
- #define **IRQ_PKT_TYPE** 3
- #define **NOC_PTR_WIDTH** 14
- #define 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.
- #define NOC_ACTIVE_BIT 0x80000000

  The flag to mark a DMA entry as valid.
- #define **OFFSET_WIDTH** (11+2)
- #define **BANK**(ID) (ID<<OFFSET_WIDTH)
- #define **DMA_BANK** BANK(0)
- #define **SCHED_BANK** BANK(1)
- #define **TDM_BANK** BANK(2)
- #define **MC_BANK** BANK(3)
- #define **IRQ_BANK** BANK(4)
- #define NOC_DMA_BASE ((volatile unsigned int _IODEV *)(0xE0000000+DMA_BANK))

  The base address for DMA entries.
- #define NOC_SCHED_BASE ((volatile unsigned int _IODEV *)(0xE0000000+SCHED_BANK))

  The base address for DMA routing information.

- #define NOC_TDM_BASE ((volatile unsigned int _IODEV *)(0xE0000000+TDM_BANK))

  The base address for the slot table.
- #define NOC_MC_BASE ((volatile unsigned int _IODEV *)(0xE0000000+MC_BANK))

  The base address for the slot table.
- #define NOC_IRQ_BASE ((volatile unsigned int _IODEV *)(0xE0000000+IRQ_BANK))

  The base address for the slot table.
- #define NOC_SPM_BASE ((volatile unsigned int _SPM *)0xE8000000)

  The base address of the communication SPM.

Functions

- void noc_sched_set (unsigned char config)

  Loads the configuration data from the noc_init_array.
- void noc_enable (void)

  Enables the TDM schedule in the NoC.
- void noc_disable (void)

  Disables the TDM schedule in the NoC.
- int noc_configure (void)

  Configure network interface according to initialization information in noc_init_array.
- int noc_fifo_irq_read (void)

  Reads out an element of the fifo with remote irq's.
- int noc_fifo_data_read (void)

  Reads out an element of the fifo with data irq's.
- void noc_init (void)

  Configure network-on-chip and synchronize all cores.
- void **__remote_irq_handler** (void)
- void **__data_recv_handler** (void)
- void **__noc_trap_handler** (void)
- int noc_conf (unsigned dma_id, volatile void _SPM *dst, volatile void _SPM *src, size_t size)

  Start a NoC configure transfer.
- int noc_irq (unsigned dma_id, volatile void _SPM *dst, volatile void _SPM *src)

  Start a NoC interrupt.
- int noc_dma_done (unsigned dma_id)

  Check if a NoC transfer has finished.
- void noc_dma_clear (unsigned dma_id)

  Stops a NoC transfer and clear the DMA entry.
- int noc_nbwrite (unsigned dma_id, volatile void _SPM *dst, volatile void _SPM *src, size_t size, unsigned irq_enable)

  Attempt to transfer data via the NoC (non-blocking).
- void noc_write (unsigned dma_id, volatile void _SPM *dst, volatile void _SPM *src, size_t size, unsigned irq_enable)

  Transfer data via the NoC (blocking).
- void noc_multisend (unsigned cnt, unsigned dma_id[ ], volatile void _SPM *dst[ ], volatile void _SPM *src, size_t size, unsigned irq_enable)

  Multi-cast transfer of data via the NoC (blocking).
- void noc_multisend_cs (coreset_t *receivers, volatile void _SPM *dst[ ], unsigned offset, volatile void _SPM *src, size_t size, unsigned irq_enable)

  Multi-cast transfer of data like noc_multisend(), but with coreset and a single destination address.
- void noc_wait_dma (coreset_t receivers)

  Wait until all transfers to a set of receivers have finished.

Variables

- const int NOC_CORES

  The number of cores on the platform.
- const int NOC_CONFS

  The number of configurations in the noc_init_array.
- const int NOC_TABLES

  The number of tables for NoC configuration.
- const int NOC_SCHEDULE_ENTRIES

  The number of entries in the schedule table.
- const int noc_init_array [ ]

  The array for initialization data.
- const int NOC_MASTER

  The master core, which governs booting and startup synchronization.

Detailed Description

Low-level NoC communication library for the T-CREST platform. Definitions for libnoc.

Author

Wolfgang Puffitsch `wpuffitsch@gmail.com`
Rasmus Bo Soerensen `rasmus@rbscloud.dk`

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.