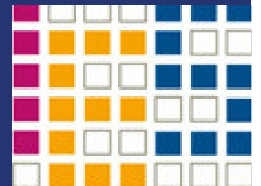
Development and Integration of a Hardware Oriented NoC Transfer Protocol Specialized for Software Parallelization

Bachelor thesis submitted by Manuel Bied
Supervising Tutor: Dipl.-Ing. Alex Schönberger
Begin: 04/06/2014 | Submission: 03/11/2014
Integrated Electronic Systems Lab
Prof. Dr.-Ing. Klaus Hofmann

TECHNISCHE
UNIVERSITÄT
DARMSTADT



Declaration

Herewith I declare, that I have made the presented paper myself and solely with the aid of the means permitted by the examination regulations of the Darmstadt University of Technology. The literature used is indicated in the bibliography. I have indicated literally or correspondingly assumed contents as such.

Ich versichere hiermit, die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt zu haben. Die verwendete Literatur und sonstige Hilfsmittel sind vollständig angegeben.

Darmstadt, 03. November 2014 ....
Manuel Bied

Contents

List of Figures iv

List of Listings vi

Used Abbreviations viii

1 Introduction 1
1.1 Motivation . . . 1

2 Fundamentals 2
2.1 Exceptions and Interrupts . . . 2
2.1.1 Exception Handler . . . 2
2.2 Build Process of a C Program . . . 2
2.3 Fundamentals of Parallelism in Hardware and Software . . . 3
2.3.1 Processor Architecture . . . 3
2.3.1.1 Pipelines . . . 4
2.3.1.2 Hardware Multithreading . . . 5
2.3.1.3 Coprocessors and Superscalar CPUs . . . 5
2.3.1.4 Multicore Processors & Multiprocessors . . . 5
2.3.2 Memory Architectures . . . 6
2.3.3 Processes and Threads . . . 6
2.4 Approaches to Parallelization . . . 7
2.4.1 Parallel Compilers . . . 7
2.4.2 Parallel Libraries . . . 7
2.4.3 PCAM Method . . . 7
2.5 Synchronization . . . 8
2.5.1 Communication Protocols . . . 9
2.5.1.1 Polling . . . 9
2.5.1.2 Stop-and-wait Protocol . . . 9
2.5.2 Mechanisms for Synchronization of competing Accesses . . . 10
2.6 Performance Metrics . . . 12

3 Description of the Environment 15
3.1 Plasma CPU Core . . . 15
3.1.1 Control Unit . . . 16
3.1.2 Datapath . . . 17
3.1.3 Delay Slot . . . 17
3.2 Specifications of the Network-on-Chip . . . 17
3.2.1 tb_platform.vhd . . . 20
3.3 Communication . . . 20
3.3.1 Router . . . 20

4 Implementation **22**

4.1 Changes in the Build Process . . . 22
4.1.1 GCC Compiler Attributes . . . 22
4.1.2 Makefile . . . 23
4.1.3 Linker Script . . . 23
4.1.4 Script lst2files.pl . . . 23
4.2 Implementation of Coprocessor0 . . . 24
4.3 Implementation of an Interrupt Functionality . . . 27
4.4 Implementation of additionally required Instructions . . . 29
4.4.1 mfc0 . . . 29
4.4.2 mtc0 . . . 31
4.4.3 di/ei . . . 33
4.4.4 ehb . . . 35
4.4.5 ins . . . 36
4.4.6 eret . . . 38
4.5 Software protocol . . . 40
4.6 The Interrupt Service Routine . . . 42

5 Conclusion **45**

5.1 Evaluation . . . 45
5.2 Outlook . . . 45
5.2.1 Scalability . . . 45
5.2.2 Delay of Interrupt . . . 45
5.2.3 Correct Implementation of mfc0 and mtc0 . . . 46
5.2.4 Acknowledgements . . . 46
5.2.4.1 Advanced Features for Interrupts . . . 46
5.2.5 Full Portation to MIPS32r2 . . . 46

6 Bibliography **47**

List of Figures

2.1 Build Process of a C program 3

2.2 Instruction pipeline. 4

2.3 Schematic of a multi-core processor. 5

2.4 Schematic of a multiprocessor system. 6

2.5 PCAM-model 8

2.6 Successful Stop-and-Wait Protocol 9

2.7 Stop-and-Wait Protocol with the two possible timeout reasons 10

2.8 Basic functionality of a Mutex 10

2.9 Basic functionality of Reader-writer locks 11

2.10 Basic functionality of a Semaphore 12

2.11 Basic functionality of a Barrier 12

2.12 Speedup according to Amdahl's law 13

3.1 Block diagram of the PLASMA CPU core 15

3.2 Memory. 19

3.3 Communication between the cores. 20

3.4 Registers of a router. 21

4.1 Description of the instruction mfc0. 30

4.2 Description of the instruction mtc0. 32

4.3 Description of the instruction ei. 34

4.4 Description of the instruction di. 34

4.5 Description of the instruction ehb. 35

4.6 Description of the instruction ins. 36

4.7 Symbolic description of the instruction ins. 36

4.8 Description of the instruction eret. 39

List of Listings

3.1 Aliases for the decoding process 16
3.2 General status information 18
3.3 Memory addresses and length 18
3.4 core0 18
3.5 core1 19
3.6 Router address 19
3.7 Definition of a router 21
3.8 Sending with a router 21

4.1 compilerflags, CFLAGS of makefile 23
4.2 compilerflags, CFLAGS_32r2 of makefile 23
4.3 compilation of isr.c 23
4.4 linking of the section .isr 23
4.5 Modification of the lst2files script. 24
4.6 Modified input- and output port list of the reg bank. 25
4.7 Representation of the CP0 register bank. 25
4.8 Signals for write enable detection. 25
4.9 Write enable detection. 25
4.10 Initial state of the register banks when a reset occurs. 26
4.11 Modified write process:regular register bank. 26
4.12 Write process:CP0 register bank. 26
4.13 Write Process: IE-field of the status register. 26
4.14 Write Process: epc 26
4.15 Read Process of the CP0 register bank. 27
4.16 Modified port list of the router 27
4.17 Setting of the interrupt signal(router) 27
4.18 New added port *intr* 28
4.19 Assigning the *interrupt* signal to the *intr* port 28
4.20 Interrupt process (Control Unit) 28
4.21 Interrupt Service Routine Address 28
4.22 Interrupt Service Routine Address 29
4.23 Passing the pc value onto the EPC register 29
4.24 Encoding of opcode **COP0** 30
4.25 Encoding of **“MF”** 30
4.26 Modification of the result MUX(control signal) 30
4.27 Modification of the result MUX(data path) 31
4.28 Decoding of the instruction **“mfc0”** 31
4.29 Default value of c0_rd_index 31
4.30 Modification of source MUX(control signal) 31
4.31 Modification of source a MUX(data path) 32
4.32 Encoding of **“MT”** 32
4.33 Decoding of the instruction **“mtc0”** 33
4.34 Modification of the forward logic for rs 33
4.35 Encoding of **“MFMC0”** 34
4.36 Passing bit 5 of the instruction to the register bank 35

4.37 Decoding of the instruction **“di”** and **“ei”** 35
4.38 Pseudoimplementation of *ehb* 35
4.39 Encoding of opcode **SPECIAL3** 37
4.40 Encoding of the function **INS** 37
4.41 Modification of the result MUX(control signal) 37
4.42 Modified port list of the ALU 37
4.43 Casting into the integer values *msb* and *lsb* 38
4.44 Implementation of the (*ins*)-operation performed by the ALU 38
4.45 Decoding of the instruction **“ins”** 38
4.46 Encoding of the function **ERET** 38
4.47 Decoding of the instruction **“eret”** 38
4.48 Returning to the epc address 39
4.49 The function main_core2() 40
4.50 The function f_noc_bmptoimage() 40
4.51 The function f_noc_bmptoimage() 40
4.52 Initialization of the structure v_noc_bmtoimage 41
4.53 Core1 waiting for acknowledgement of core2 41
4.54 The function ISR() 41
4.55 The function f_noc_t1_encode_cblks() 42
4.56 The function f_noc_tcd_rateallocate() 42
4.57 Moving the cause and the exception cause address to *k0* and *k1*. 42
4.58 Storing the exception cause address to the memory 42
4.59 Computation of the recent CP0 status. 43
4.60 Register saving and call of the received function. 43
4.61 Restoring of the non-preserving register. 44
4.62 Returning to the regular program flow. 44

Used Abbreviations and Acronyms

ACK	**Ack**nowledgment
ALU	**A**rithmetic **L**ogic **U**nit
AMP	**A**ccelerated **M**assive **P**arallelism
CP0	**C**o**p**rocessor 0
CPU	**C**entral **P**rocessing **U**nit
DSM	**D**istributed **S**hared **M**emory
EPC	**E**xception **P**rogram **C**ounter
EX	pipleine stage **Ex**ecute
FPU	**F**loating **P**oint **U**nit
GPU	**G**raphics **P**rocessing **U**nit
ID	pipeline stage **I**nstruction Fetch&**D**ecode
IE	**I**nterrupt **E**nable
I/O	**I**nput/**O**utput
ISA	**I**nstruction **S**et **A**rchitecture
ISR	**I**nterrupt **S**ervice **R**outine
MIPS	**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
MEM	**Mem**ory
MIPS I	MIPS instruction set version I
MIPS32r2	MIPS instruction set version 32 revision 2
MUX	**Mu**ltiple**x**er
NoC	**N**etwork **o**n a **C**hip
OS	**O**perating **S**ystem.
PC	**P**rogram **C**ounter
PCAM	**P**artitioning **C**ommunication **A**gglomeration **M**apping
RAW	**R**ead **A**fter **W**rite
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
WAR	**W**rite **A**fter **R**ead
WAW	**W**rite **A**fter **W**rite
WB	**W**rite**b**ack

1 Introduction

For a long time a common approach to achieve higher performance in computer technology was to increase the clock frequency of the processing unit in a processor. For more than 20 years the frequency has increased by 30% per year. This approach still improves processor performance but it gets more difficult and less efficient due to the heat dissipation and high power consumption.
A new approach is to design processors which use more than one core. To exploit more than one core, parallel programming is needed. The concept of parallel programming is not new, it has been explored since the 1960's, but was only spread in a scientific context, like simulation of scientific phenomena. For desktop computers used at home, the common method wasto design sequential implemented algorithms. Nowadays most computers in professional and home use include a multi-core processor. But the old algorithms which were designed for single-core processors can not exploit the full capability of a multi-core processor. There are two main approaches to design parallel algorithms, which differ significant in their effort and outcome. Those approaches are **auto parallelization** and **manual parallelization**. To realize parallelism there are different memory models which can be used, like the shared memory model and the distributed memory model. Most parallel algorithms require the exchange of data or at least minimal information between the cores due to data dependencies or simply to synchronize them. The regulation of the communication between the cores is itself an interesting field. Therefore exist many protocols and mechanisms to manage the communication and synchronization[13, 19].

1.1 Motivation

Using benchmarks to test and compare different multi-core systems is a common approach. But it implies a major weakness: because the input parameters are fixed, the results apply only to one special case.
Previous to this work, Manuel Bied and Felix Pels worked on a "Projektseminar" at the TU Darmstadt parallelizing parts of a JPEG2000-algorithm which was implemented in C. They worked with a Network-on-Chip system with two cores.
The perspective was to automatize the parallelization for any given number of cores and any given algorithm that could be parallelized manually. A common benchmark could only evaluate the performance with fixed input data. With this approach a broad variety of pictures can be used to compare the performance. If an automation works on any kind of algorithm that can be parallelized, the inputs are not limited to pictures. Other inputs could be audio sequences or videos.
The result of the "Projektseminar" was a manually parallelized C code. The parallelization implemented the synchronization between the two cores by software. The first core set a variable to true, when the second core was supposed to start a parallelized loop. The second core was constantly reading the memory to check whether the variable is set to true. This leads to increased memory bus traffic. This approach will not work regarding scalability of the system. The memory bus will not be able to handle the traffic created by multiple cores continuously reading the memory.
The goal of this work is to develop and integrate a hardware oriented protocol to avoid the permanent reading of the memory. This requires the possibility of interrupts provided by hardware. The solution should be as general as possible and independent from the running algorithm[13].

2 Fundamentals

2.1 Exceptions and Interrupts

The most challenging aspect of processor design is the control logic. To get the logic to function properly on the one hand and to compute fast on the other hand, are both aspects not easy to implement. One of the hardest part regarding control is the implementation of exceptions and interrupts, which change the regular flow of the program. **Exceptions** are unexpected events, which occur within the state of the processor, like a division by zero. They occur synchronously and are also called precise exceptions. **Interrupts**, also called imprecise exceptions, are externaly caused hardware exceptions triggered by an I/O-device such as a mouse.[8, 11]

2.1.1 Exception Handler

When an exception occurs, the processor saves the address and cause of the exception and jumps to a certain code to handle the exception. This code is called **exception handler**. To be able to handle interrupts, processors need to implement some register to store the exception address **(EPC)** and the **cause**. The exception handler examines the cause and reacts accordingly to the cause. After handling the exception the processor returns to the saved address. Exception handler that are designed for interrupt handling are called interrupt handler or interrupt service routine (**ISR**)[8, 11].

2.2 Build Process of a C Program

The building process of a C program consists of four steps: Preprocessing, Compilation, Assembly and Linking. It results in one executable file.

1. **Preprocessing**: The preprocessor allows the inclusion of header files, macros and conditional compilation. It takes the source code as input and produces modified source code as output.

2. **Compilation**: The compiler uses the modified source code as input and generates an assembler source code as output.

3. **Assembly**: The assembler source code is translated into object code, which is relocatable machine code.

4. **Linker**: The linker combines one or more object files. It resolves external symbols and assigns final addresses to functions and variables.

Technically speaking step 2 and 3 are different steps, although they are often referred to as one step called "compilation". The schematic process can is shown in figure 2.1. Instructions for the preprocessor are directly inserted in the C code. Instructions for the compiler and assembler can be implemented in the makefile. Instructions for the linker are implemented in the linker script[3, 8, 17].

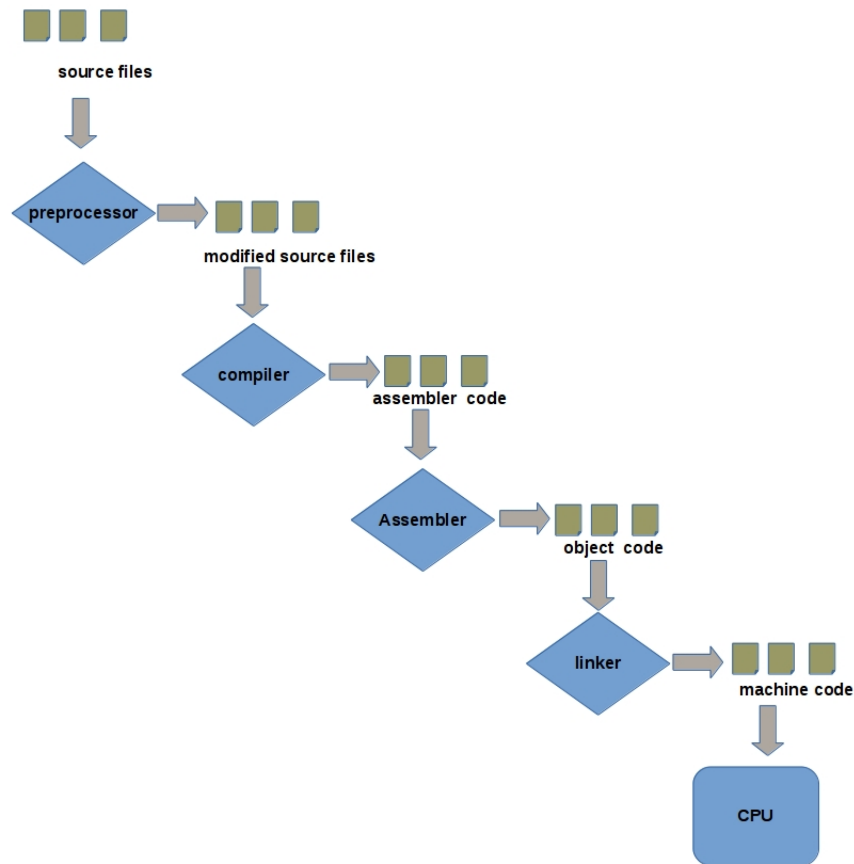


Figure 2.1: Build process of a C program.[12]

2.3 Fundamentals of Parallelism in Hardware and Software

2.3.1 Processor Architecture

The are two kinds of processor architectures: **Homogeneous** and **heterogeneous** architectures. A heterogeneous architecture uses processors of different kinds, whereas a homogeneous architecture consists of multiple processors of the same kind.
Architectures also differ from each other regarding which assembler instructions can be executed. The native commands which are implemented by a specific processor are defined by the *instruction set architecture* (**ISA**). Two approaches to ISAs are *reduced instruction set computer* (**RISC**) and *complex instruction set computer* (**CISC**). RISC architectures implement up to a few hundred instructions whereas CISC architectures implement a few thousand. Special operations like “string move”, which need hundreds of instructions on a RISC machine, need only one instruction on a CISC machine. This increases the hardware effort and reduces the performance of simple instructions down. Even before the introduction of multi-core processors there was a certain level of parallelism. Processors executed instructions or parts of instructions in parallel.
Some mechanisms will be explained below[16, 19].

2.3.1.1 Pipelines

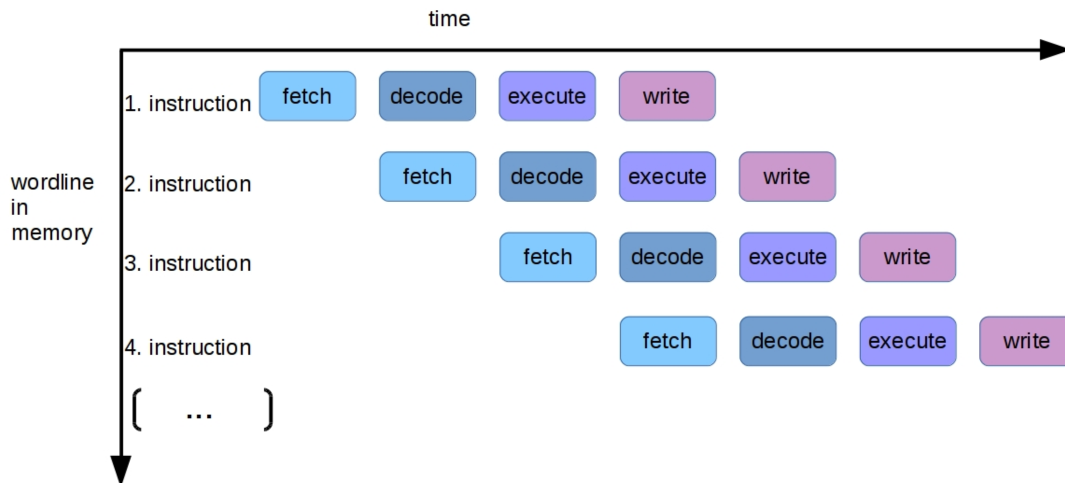


Figure 2.2: Instruction pipeline. [19]

The maximal clock rate of a processor is determined by the longest path, also called critical path. In order to achive higher clock rates the critical path needs to be shortened. Analogous to a production line, the processor is divided into different stages. Different parts of one single instruction are executed in different stages. Those are called stages of the pipeline. This shortens the critical path by a significant factor. The factor depends on how many stages are used. The downside of this is that one instruction needs multiple clock times to be finished. But there is still one instruction finished each clock cycle. Figure 2.2 illustrates a simple pipeline with four stages:

- **Fetch**: The first stage fetches the next instruction from the memory.
- **Decode**: The second stage decodes the instruction and sets the control signals accordingly.
- **Execute**: The third stage executes the instruction.
- **Write**: The fourth stage writes the the result of the instruction to a register or to the memory.

There are some possibilities to enhance the effectivity of a pipeline:

- **"out-of-order-execution"**: The processor reorders the instructions to achive a better workload of the instructions.
- **"superpipelining"**: in commercial processors there are 10-20 pipeline stages used, but there also exist processors with up to 28 pipeline stages (Pentium 5 with Hyperthreading).

The following data dependencies can occur in a pipeline, but their concepts are not constricted to pipelines:

- Read-After-Write dependence (**RAW**): An instruction reads data that has been written by a previous instruction.

- Write-After-Read dependence (**WAR**): An instruction overwrites data that has been read by a previous instruction.

- Write-After-Write dependence (**WAW**): Two instructions write the same memory cell, therefore the order of the writing process effects the result.

The importance of handling those dependencies increases when approaching parallelism.[11, 19]

2.3.1.2 Hardware Multithreading

From the viewpoint of a user of a single core processor, hardware multithreading appears like a processor with multiple cores. While waiting for main memory access, processors that support hardware multithreading execute instructions from another thread. Hardware multithreading increases the performance of a processor, but is not efficient as a processor with real additional cores. [19]

2.3.1.3 Coprocessors and Superscalar CPUs

Coprocessors are computing units which are implemented in addition to the regular processor core. They are optimized to perform certain tasks. Examples are:

- **Graphical Processing Units** (**GPU**) are optimized for graphical computing.

- **Floating Point Units (FPUs)** are optimized to perform floating point operations.

Superscalar CPUs are CPUs which contain the datapath hardware multiple times. This way they can fetch and execute more than one instruction per clock cycle. [16, 19]

2.3.1.4 Multicore Processors & Multiprocessors

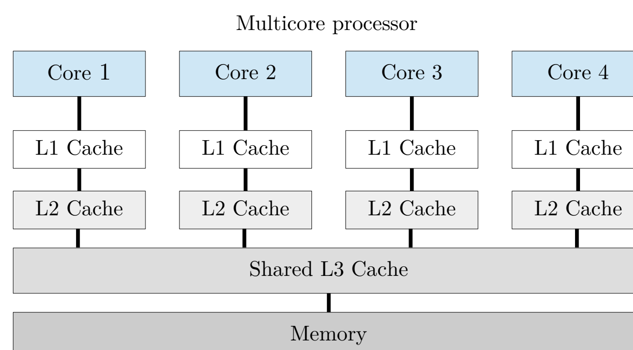


Figure 2.3: Schematic of a multi-core processor.[19]

Multi-core processors are processors that contain two or more cores on a single chip. The cores may share cache and main memory but they do not need to.

Figure 2.3 shows a homogeneous architecture which shares the main memory and L3-cache. Processor systems that consist of multiple (multi-core) processors are called **multiprocessor systems**. Each processor has its own cache. The processors are connected via some kind of network, like a **Network-on-Chip (NoC)** for example. This is illustrated in Figure 2.4[19].

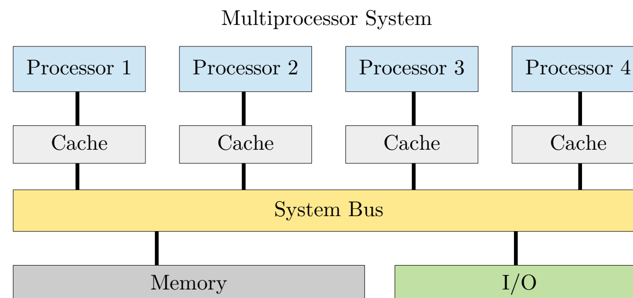


Figure 2.4: Schematic of a multiprocessor system.[13]

2.3.2 Memory Architectures

In the context of parallelism, there exist the following different memory models:

- **Shared memory model**: The processes of the multi(-core) processor systems share the same logical address space.
- **Distributed memory model**: Each processor has a memory of its own. The processors communicate through messages. This is called message passing.
- **Distributed shared memory model**: There are multiple physically separated memories. They are logically combined through virtual address space.

Programming on a shared memory system is considered easier than on a distributed memory because the effort of load balancing and data partitioning is reduced. [19]

2.3.3 Processes and Threads

The smallest unit of parallel activity are processes and threads.

- **Processes**: They are instances of computer programs and do not share the main memory of the computer. They have access to defined parts of the memory only and cannot access memory parts that are used by other processes.
- **Threads**: They are program parts inside a process. Threads share the resources used by the process like operation system resources and memory. They can be used to efficiently use available processor resources in multi-core systems. In single-core systems, threads can be used to divide a task in

different subtasks or to maintain responsiveness of a user interfaces in the foreground and still compute in the background.

The easiest way to exploit the possibilities of multi-core processors is to start a process on each core. This requires the data of each processes to be independent from other processes. Data dependencies require communication between the processes, in such cases threads are the better and faster choice. [19]

2.4 Approaches to Parallelization

Two main approaches in parallelization are auto parallelization and manual parallelization.

- **Auto parallelization**: This approach requires marginal manual effort, but the gain in performance with this approach is also marginal. Two typical approaches are the use of parallel libraries and the use of parallel compilers.

- **Manual parallelization**: Existing code is modified or new code is designed to support parallel execution. The required effort is very high, but the resulting speedup compared to a sequential algorithm is accordingly higher. The design of a parallel algorithm is a parallel process itself because a lot of concerns have to be taken into account at the same time. The process contains multiple stages and sometimes returning to a previous stage is inevitable. A typical design method is PCAM.[9]

2.4.1 Parallel Compilers

This approach requires little effort. An existing code is recompiled with a parallel compiler. The dependencies in the program are required to be known, therefore this approach is still narrowed down to parallel loops. [19, 10]

2.4.2 Parallel Libraries

There are multiple libraries which supply parallelized algorithm for numeric analysis and image processing. With the use of parallel libraries, it is possible to reduce the development effort of a program that supports parallelism. An example for a library that exploits data-parallel hardware like GPUs is the C++ library **AMP** developed by Microsoft.[1, 19]

2.4.3 PCAM Method

PCAM is an acronym for ***P****artitioning,* ***C****ommunication,* ***A****gglomeration and* ***M****apping*.

1. **Partitioning**: The problem is split into multiple small tasks. Practical matters[1] are not taken into account.

2. **Communication**: Dependencies are determined and communication topologies are defined.

[1] Like the hardware on which the algorithm should be executed.

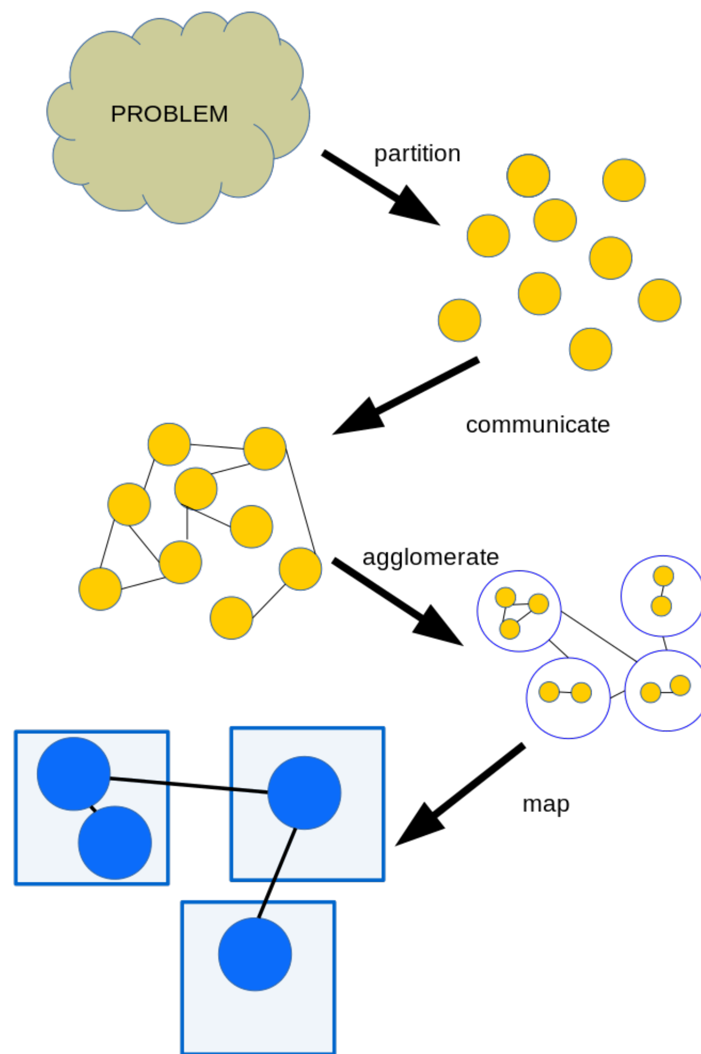


Figure 2.5: PCAM-model [9]

3. **Agglomeration**: To decrease the communication effort the subtasks are combined to larger tasks. In this stage practical matters are taken into account.

4. **Mapping**: The tasks are assigned to a processor, dependent tasks are assigned to one processor, independent tasks are assigned to different processors. Sometimes it is not possible to comply with both rules.

Figure 2.5 illustrates the four steps of the PCAM-process.[9]

2.5 Synchronization

Working with multiprocessor systems or multi-core processors requires synchronization. To regulate the program flow communication is required. To prevent conflicts while accessing resources different mechanisms are used depending on the program and resource properties.

2.5.1 Communication Protocols

To implement communication in general, there are basic protocol concepts. Two simple ones, which are used in different context are explained subsequently.

2.5.1.1 Polling

Polling is the easiest way for an external source, like an I/O device, to communicate with the processor. The processor periodically checks a status bit, whether the I/O device needs to be handled. If the I/O device has new input data, it sets the bit. Since the processor has full control, how often the bit is checked, the I/O data rates are predictable. The downside of this is the possibility of waste of resources, if the processor checks the status bit multiple times only to find the bit not set. An alternative is to drive I/O devices via interrupts. [8]

2.5.1.2 Stop-and-wait Protocol

The stop-and-wait protocol, also referred to as "positive acknowledgement with retransmission", is a basic concept to provide reliable transport for an unreliable transmission system.

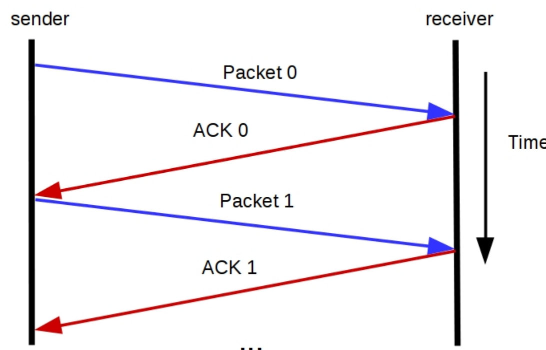


Figure 2.6: Successful Stop-and-Wait Protocol[5]

There exists a sender and a receiver. The sender sends each packet with a unique sequence number. After receiving, the receiver sends an **acknowledgement (ACK)**. The sender does not send the next packet until it receives an ACK with the corresponding sequence number. This behavior can be seen in figure 2.6. There are two possible ways how this protocol would end in a deadlock: Either the packet or the ACK is lost. To resolve this, the sender sets a timer. If it does not receive an ACK within a predefined time, it resends the previous packet. This behavior can be seen in figure 2.7.[5, 7]

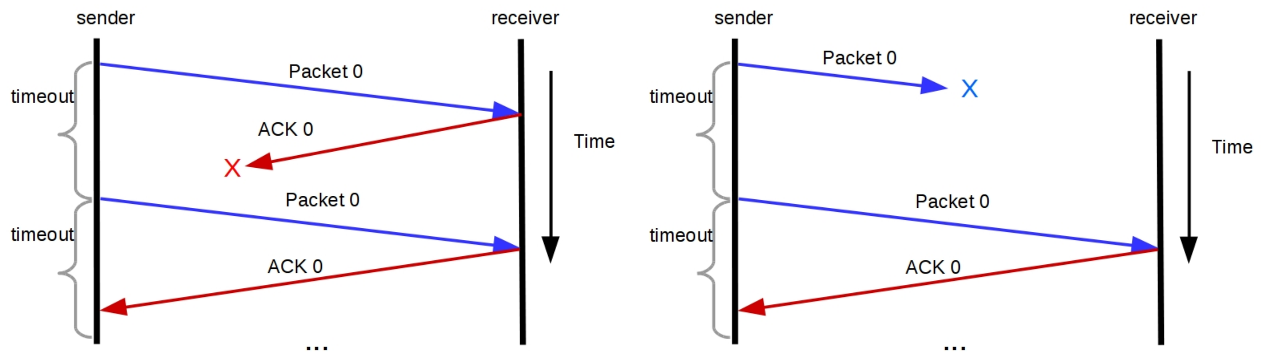


Figure 2.7: Stop-and-Wait Protocol with the two possible timeout reasons[5]

2.5.2 Mechanisms for Synchronization of competing Accesses

A challenging task in parallel programming is to synchronize concurrent accesses on resources in order to prevent conflicts. Because multiple threads are executed concurrently and not sequential, this can lead to unintended results. The reading and writing access on data has to be managed. Typical mechanisms are explained as follows:

- **Mutexes**: Mutexes are the most simple mechanism to protect critical sections of a program. A mutex has two states **locked** and **unlocked**. A thread can only access a critical section if the section is unlocked. When a thread enters a critical section it locks the mutex. This mechanism fails for recursive method calls. Figure 2.8 illustrates the behavior.

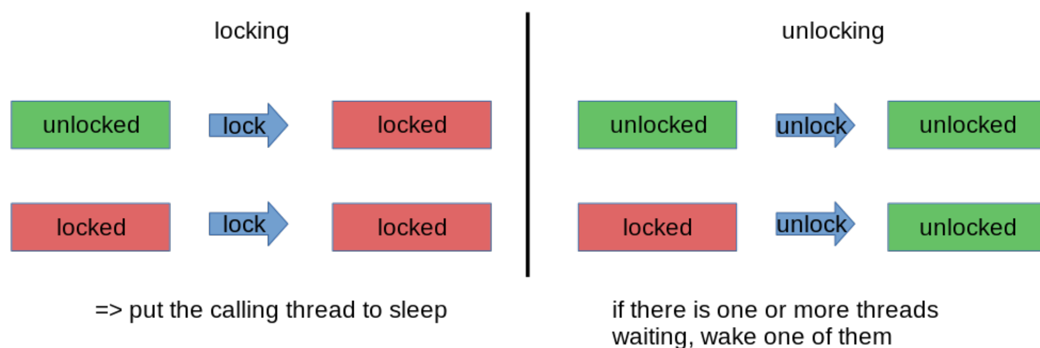


Figure 2.8: Basic functionality of a Mutex[19]

- **Scoped Locking**: Scoped locking is a simpler method than a mutex, but can only be used in programming languages where local objects are automatically destroyed. A mutex is locked in the constructor and unlocked in the destructor of an object. This has the advantage that the release of a mutex cannot be forgotten.

- **Monitors**: Monitors are mutexes that cannot be directly accessed by the user. The locking and unlocking is managed by the compiler. Therefore the programmer cannot forget to unlock a mutex.

- **Reader-writer locks**: There is no conflict in multiple threads to read the same data. But when the data is read by one or more threads, no other thread may be allowed to write the data at the same time. Vice-versa when a thread is writing the data, no other thread may be allowed to read it. The function of reader-writer locks is illustrated in figure 2.9.

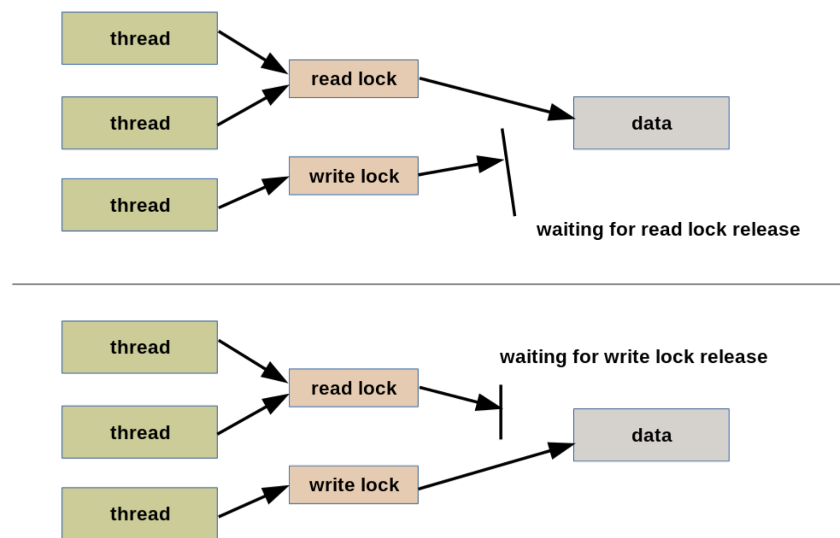


Figure 2.9: Basic functionality of a Reader-writer locks[13]

- **Spinlocks**: Spinlocks are used when the waiting time to enter a critical section is shorter than the time to switch to another thread. A spinlock actively checks whether the lock is cleared.

- **Call once**: A call once method is exactly executed once to initialize a data structure.

- **Atomic operations**: An atomic operation is an operation which includes multiple instructions. It is either executed completly or not at all. Therefore it is impossible that only a few instructions are executed while the rest is not.

- **Semaphores**: Semaphores restrict the number of threads that can access certain data to a predefined number n. The mechanism is like a mutex, but instead of locking the semaphore threads decrease a counter when accessing the data. If the counter is zero a new thread cannot access it. If a thread does not use the data anymore it increases the counter. Figure 2.10 illustrates the mechanism.

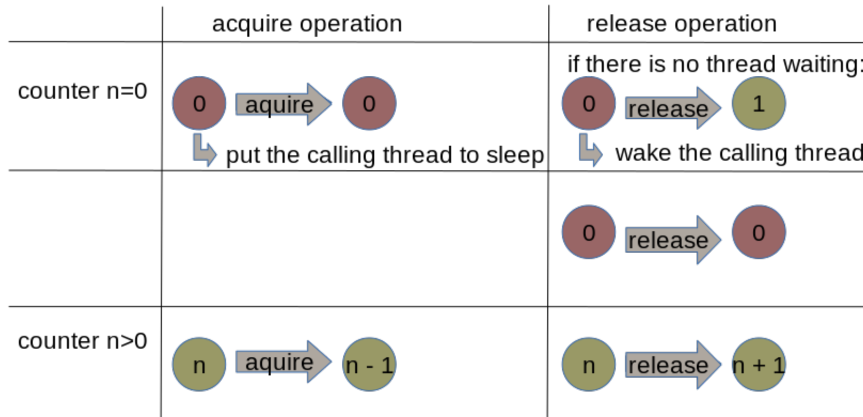


Figure 2.10: Basic functionality of a Semaphore[19]

- **Condition variables**: They contain information about which thread is waiting for which mutex. When a thread unlocks a mutex, it can lookup which other thread is waiting for the mutex and wake it.

- **Barriers**: Barriers are used to synchronize threads. A thread can only continue to execute the program, when all other threads have reached the barrier. Figure 2.11 illustrates the mechanism of a barrier[19].

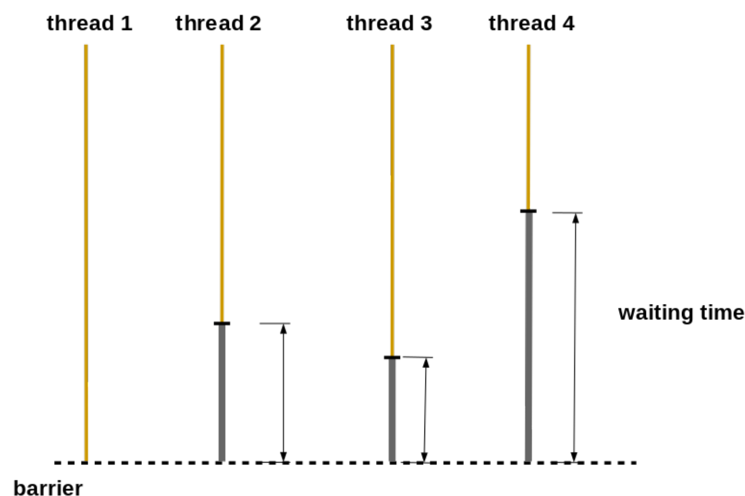


Figure 2.11: Basic functionality of a Barrier[19]

2.6 Performance Metrics

There exist a variety of performance metrics to evaluate the quality of a parallelized algorithm. Some definitions are given as follows:

p : number of used processors

n : size of the input

$T_p(n)$: parallel runtime - measured time between start and end of a parallel program

$C_p(n)$: cost[1] - measurement for the tasks carried out by all processors, defined by:

$$C_p(n) = T_p(n) \cdot p.$$

$T^*(n)$: runtime of the quickest sequential algorithm

$S_p(n)$: speedup of the parallel implement compared to the sequential implementation, defined[2] by:

$$S_p(n) = \frac{T^*(n)}{T_p(n)},$$

The speedup describes a relative speed gain which is achieved by the use of p processors. In most cases, due to overhead effects like synchronization and communication *linear speedup* $S_p(n) = p$ is not achieved. Because the best sequential algorithm might not be known or the effort to implement it might be too high, this definition comes with some shortcomings. Taking into account that a certain part of a

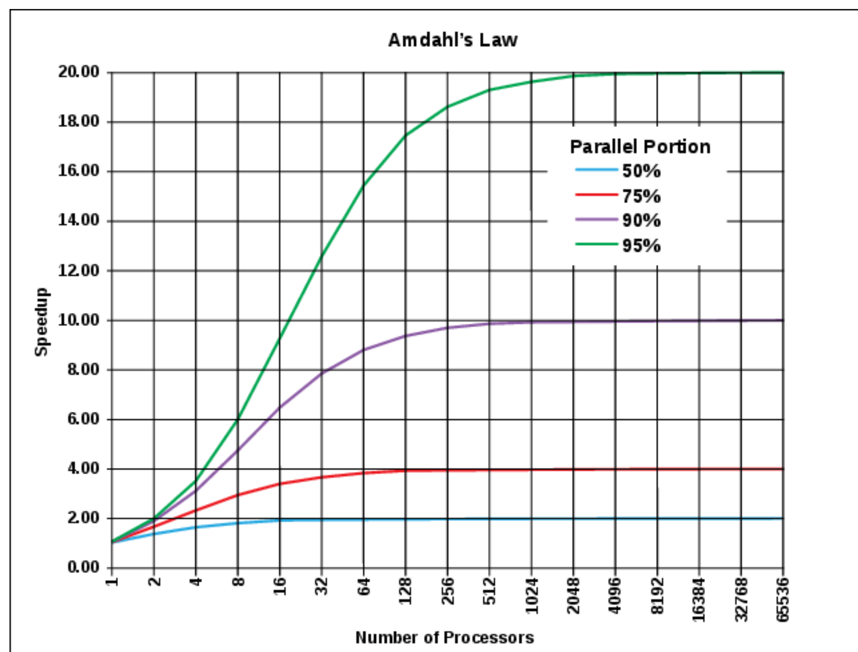


Figure 2.12: Maximum speedup according to Amdahl's law [6].

program has to executed sequential and cannot be parallelized, there is a modified version of the equation. Let f $(0 \leq f \leq 1)$ be a constant fractional part of a parallel implementation that has to be executed sequentially.

The modified Amdahl's law then yields:

$$S_p(n) = \frac{T^*(n)}{f \cdot T^*(n) + \frac{(1-f)}{p} \cdot T^*(n)} = \frac{1}{f + \frac{1-f}{p}}.$$

[1] It is also known as process time product.
[2] This definition is known as Amdahl's law.

The part that cannot be parallelized has a significant impact on the maximum speedup. If this part amounts 1/10, the maximum speedup[1] is 10. The speedup for different portions that can be parallelized with increasing number of processors is illustrated in figure 2.12. It shows that the speedup converges to a maximum available speedup. [11, 18]

[1] The maximum speedup can be calculated with p going to infinity.

3 Description of the Environment

The platform consists of a simulated Network-on-Chip(NoC) with two cores. The cores communicate through a router module. The specifications of the NoC are given in the data file **noc.h**. Modelsim is used as simulator. The code that is executed, is an implementation of the JPEG2000-algorithm written in C by openJPEG[1]. The two cores use a shared memory model. Two simultaneous read operations are allowed , whereas two simultaneous write operations at the same address are denied and result the simulation to stop with an error.

3.1 Plasma CPU Core

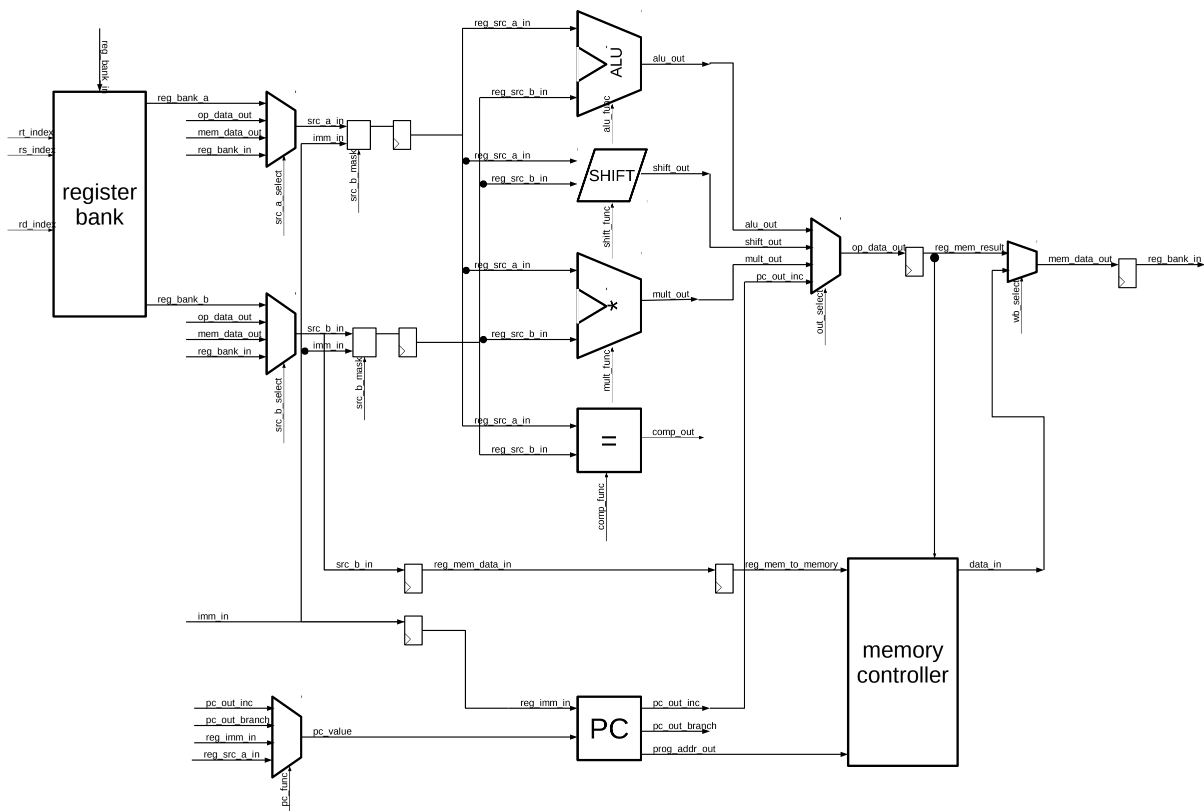


Figure 3.1: Block diagram of the PLASMA CPU core

The Plasma CPU core is implemented in vhdl. The central parts are the data path and the control unit. The datapath performs data processing operations, the control unit determines how the data is processed. It implements the four pipeline stages **Instruction Fetch&Decode (ID)**, **Execute(EX)**, **Memory (MEM)**

[1] OpenJPEG is an open-source library. For further information see [2].

and **Writeback (WB)**. It implements the **MIPS I**^{TM} - instruction set. A detailed documentation can be found in [14]. The connection of input and output signals of the control unit and the datapath are defined in the file **plasma.vhd**. The block diagram of the core can be seen in figure 3.1.

3.1.1 Control Unit

The Control Unit sets the control signals depending on the decoded instruction and the states of the pipeline. It determines the registers which are used as in- and output according to the **opcode** and the **rt-**, **rs** and **rd-field**. It also detects if a pipeline stage has to be stalled or if values can be forwarded. For example, if an instruction saves its result in register $s0 and the next instruction uses $s0 as input, the pipeline needs to be stalled because the result of the first instruction is not available in the next clock (**RAW**-problem). Thus, the second instruction needs the first instruction to pass the WB-stage, because the register $s0 is written in the WB-stage. If for example, the first instruction is *add*, the result will already be available after the EX-stage and can be forwarded. The Control Unit implements a stall- and forward-logic and sets the control signals for the input- and output multiplexers accordingly.
To represent the pipeline stages, the control signals have a representation in each stage if needed. The signal names have the following scheme:

- **ID**: i_control_signal
- **EX**: control_signal_ex
- **MEM**: control_signal_mem
- **WB**: control_signal_wb

Each clock cycle the signal is passed on to the corresponding signal in the next stage, provided that it is still needed there. For the decoding process the aliases in listing 3.1 are used to represent the different fields of the fetched instruction.

```
138  -- -----------------------------------------------------
139  -- ___  ____ ____ ____ ___  ____ ____
140  -- |  \ |___ |    |  | |  \ |___ |__/
141  -- |__/ |___ |___ |__| |__/ |___ |  \
142  -- -----------------------------------------------------
143  -- ---------- 1. STAGE: FETCH AND DECODE ---------------
144  -- operation decode
145  alias i_opcode_dec           : t_mips_opcode   is instr_dec(31 downto 26);
146  alias i_format_dec           : t_mips_format   is instr_dec(20 downto 16);
147  alias i_func_dec             : t_mips_function is instr_dec( 5 downto  0);
148
149  -- register addresses
150  alias i_rs_dec               : t_mips_reg_addr is instr_dec(25 downto 21);
151  alias i_rt_dec               : t_mips_reg_addr is instr_dec(20 downto 16);
152  alias i_rd_dec               : t_mips_reg_addr is instr_dec(15 downto 11);
```

Listing 3.1: Aliases for the decoding process

3.1.2 Datapath

The datapath performs the data processing operations depending on the signals it gets from the control unit. The datapath contains the following units:

- **The program counter (pc)**: It contains the next instruction that is to be fetched.
- **The register bank**: Storing and loading never occurs directly from the memory, values are always loaded from and stored in the register bank first.
- **The operation units:** The operation units have ***source a*** and ***source b*** as input. Each source is controlled by a multiplexer to choose which input data is passed onto the operation unit.
  - **Arithmetic logic unit:** Provides several functions like **ADD** and **OR**.
  - **Shifter**: Operates bit-shifting operations.
  - **Multiplier**: Operates multiplications.
  - **Comparator**: Checks for equality.
  - **Input- and output multiplexers**: The input and output is chosen accordingly to the control unit signals.
  - **The pipeline stage registers**: They store the data of the pipeline stages.
  - **The memory controller**: It regulates the communication with the memory.

  Subsequent to the operation units the output multiplexer chooses from whose operation unit the result is passed on.

3.1.3 Delay Slot

The implemented pipeline of the MIPSI-ISA contains a delay slot, which means that the next instruction that follows a jump or branch instructions like ***j, jr, jal, bne*** and ***beq*** are executed independently, if the condition of the previous instruction (if it has one) is true. If there is no instruction that can be performed before the jump is executed, the compiler has to insert a **nop**-instruction in the delay slot.

3.2 Specifications of the Network-on-Chip

The specifications of the NoC are described in the data file ***noc.h***. These are the informations for the preprocessor. Below the source code is described step by step. The equivalent specifications for the hardware are defined in the corresponding vhdl files. The memory model can be seen in figure 3.2.

Listing 3.2 contains general information about the NoC for the simulation platform. Line 2 defines the number of cores to two. Line 4 and 5 define the data- and address width to 32 bits. Line 7 defines the name of the main file to “openJPEG_core2’.

```
1  /*** GENERAL STATUS INFORMATION FOR SIMULATION PLATFORM */
2  #define NOC_INFO_CORE_NUMBER    2                          // number of cores in the noc
3
4  #define NOC_INFO_DATA_WIDTH     32                         // data width
5  #define NOC_INFO_ADDR_WIDTH     32                         // address width
6
7  #define NOC_INFO_MAIN       "openJPEG_core2"               // main name of files
```

Listing 3.2: General status information

Listing 3.3 contains the addresses and length of different parts of the memory. The program starts at address **0x0** and has a maximum length of 65536. The input data starts at address **0x40000** and has a maximum length of 197632. The output data starts at address **0xC1000** and has also a maximum length of 132096. The stack has a maximum length of 131072 and starts at address **0x182000**. This is not the initial address, each core has its own initial stack pointer value. The heap starts at address **0x202000** and has a maximum length of 16777216. The last address within the range of the heap is **0x1202000**.

```
31 /********  ADDRESSES AND LENGTH *************************/
32 #define NOC_MEM_PROG_START                         0x0
33 #define NOC_MEM_DATA_START    (NOC_MEM_PROG_START  + 0x40000)   // 65536 - maximal length
       of program
34 #define NOC_MEM_RESL_START    (NOC_MEM_DATA_START  + 0xC1000)   // 197632 - maximal length
       of input data
35 #define NOC_MEM_STACK_START   (NOC_MEM_RESL_START  + 0x81000)   // 132096 - maximal length
       of output data
36 #define NOC_MEM_HEAP_START    (NOC_MEM_STACK_START + 0x80000)   // 131072 - maximal length
       of stack
37
38 #define NOC_MEM_HEAP_LENGTH                        0x1000000    // 16777216 - maximal length
       of heap
```

Listing 3.3: Memory addresses and length

Listing 3.4 defines the information about core0. It starts with the function **main_core0**. The initial value of the first core's stack pointer is **0x202000**. Note that the stacks are growing from a higher to a lower valued address.

```
53 /**************  CORE 0 **********************/
54 #define NOC_CORE0_START      "main_core0"                    // start function name
55
56 #define NOC_CORE0_STACK_INIT                     0x202000     // initial stack pointer
     value
```

Listing 3.4: core0

In analogy to the previous section, listing 3.5 defines the information about core1. It starts with the function **main_core1**. The initial value of the second core's stack pointer is **0x1c2000**.

```
53  /**************   CORE 1 *********************/
54
55  #define NOC_CORE1_START      "main_core1"                    // start function name
56
57  #define NOC_CORE1_STACK_INIT                    0x1c2000        // initial stack pointer
        value
```

Listing 3.5: core1

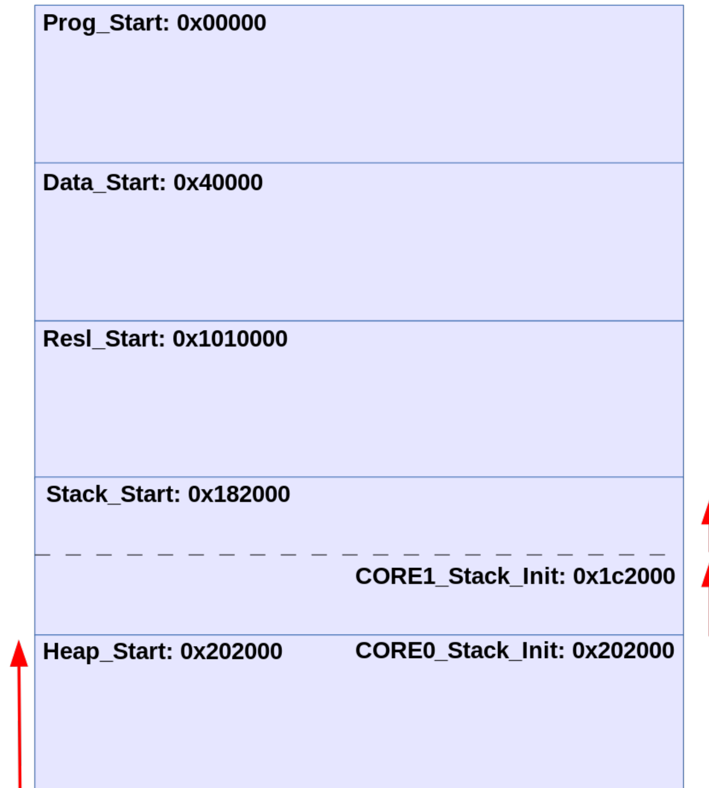


Figure 3.2: Model of the memory.[1]

Listing 3.6 sets the address of the router to **0x1202004**. This address lies beyond the normal range of the memory. Note that there is only one address defined. When a core tries to access an address the splitter checks whether this address is located in the normal accessible range of the memory or is located within the router addresses. If the core tries to access an address beyond the memory range, the splitter redirects the core to the corresponding router.

```
41  #define NOC_MEM_ROUTER_START                    0x1202004       // first address of NoC
        router module
42  #define NOC_MEM_ROUTER_END                      0x1202008       // limit address of NoC
        router module
```

Listing 3.6: Router address

3.2.1 tb_platform.vhd

The file tb_platform.vhd is used as a test bench. There are multiple flags which can be set to configure the setting. The test bench is used to create instances of routers, cores, cache and memory dependent on the set flags. The test bench defines how the components are connected.

3.3 Communication

The communication between the cores is realized by routers. Each core is connected to a splitter. Dependent on the accessed addresses, the splitter connects to the router or to the memory model. The router is connected to the other router via a data bus. The communication model is shown in figure 3.3.

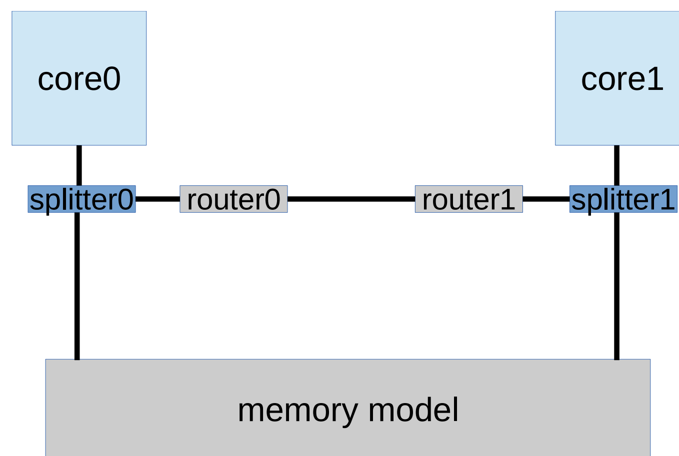


Figure 3.3: Communication between the cores.[1]

3.3.1 Router

The definition of the router is located in *noc.h*. It consists of two 32-bit registers. The first register can be accessed by ***ctrl*** and **CTRL** in the C source code. The control register consists of five fields: ***my_addr, send_addr, reserved, recv*** and ***send***. The field *my_addr* contains the address of the router itself, *send_addr* contains the address of the router to which data is to be send, *reserved* is not used, *recv* is set to **1** when the router is receiving and *send* is set to **1** when the router is sending. The second register contains the address of the received data[2]. The definition can be seen in listing 3.7, figure 3.4 illustrates the registers of a router.

[2] There is no sending of data itself, only addresses to data are being sent.

```
typedef struct{
  unsigned int my_addr      : 4;
  unsigned int send_addr    : 4;
  unsigned int reserved     : 22;
  unsigned int recv         : 1;
  unsigned int send         : 1;
} router_ctrl_reg_t;

typedef struct{
  union{
    unsigned int         ctrl;
    router_ctrl_reg_t    CTRL;
  } __attribute__((aligned (4)));

  unsigned int           data;
} router_t;
```

Listing 3.7: Definition of a router

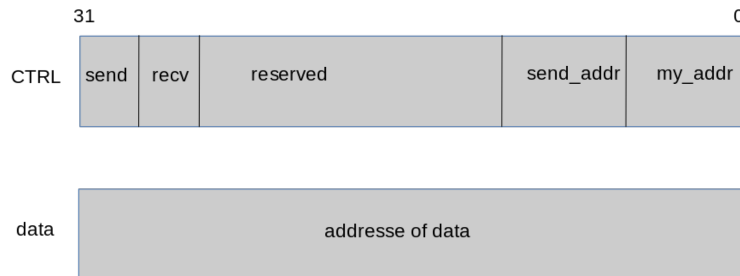


Figure 3.4: Registers of a router.[1]

The function *router_send(unsigned int addr, void * data)* is used for data transmission. As long as the router is processing a previous sending process, a new sending process waits for the previous one to finish. When the router starts to send, the data-field is set to the data address argument, the field *send_addr* is set to the addr argument and the *send* bit is set. The definition of the sending process can be seen in listing 3.8.

```
router_return_t router_send(unsigned int addr, void * data){

  while( router->CTRL.send ){

  }

  router->data = (unsigned int) data;

  router->CTRL.send_addr = addr;
  router->CTRL.send = 1;

  return ROUTER_OK;
}
```

Listing 3.8: Sending with a router

4 Implementation

The hardware oriented protocol requires implementations and changes on different levels, i.e. implementation and changing in hardware, implementation in software, changing of the makefile and linker script. The implementation cannot be understood step-by-step but rather has to work in its entirety. In order to implement a protocol that relies on hardware provided interrupts instead of software managed polling the following steps are required:

1. Since the gcc compiler provides support for interrupts[1], some changes in the build process are required.
2. An implementation of the MIPS coprocessor0 is required.
3. Additional instructions are required to be implemented.
4. The interrupt functionality itself has to be implemented in the hardware.
5. The software protocol needs to be adjusted.

4.1 Changes in the Build Process

4.1.1 GCC Compiler Attributes

The gcc compiler provides different attributes[2] to support interrupts. These attributes have to be inserted in the function header in the C source code. The attributes are documented in [4]. For the MIPS architecture the interrupt attribute looks like this:

```
__attribute((interrupt))__
```

The interrupt attribute tells the compiler to create an assembler routine that automatically handles register saving, fetching of the interrupt cause and returning to the interrupted code. This routine is wrapped around the code that is created by the marked function. The last instruction of a function marked as interrupt is always an *eret*. A more detailed description can be found in chapter 4.6. The interrupt attribute requires a **MIPS32r2** or higher implementation, which requires a manipulation of the makefile. The gcc compiler also provides an attribute to place a function into a defined section in the memory.

```
__attribute((.sectionname))__
```

The marked function will be located into the section ***.sectionname***. In order to get a fixed address for interrupt handling, which can be implemented in hardware, it is necessary to modify the linker script to locate this section to a fixed address.

[1] For MIPS this support is only provided for implementations of version MIPS32r2 or higher, this results in the steps 2&3.
[2] depending on the architecture

4.1.2 Makefile

The original flag list **CFLAGS** used for compilation was declared like in listing 4.1 in the makefile. The flag of interest is the **-mips1** flag, which tells the compiler to compile for MIPSI architecture.

```
CFLAGS      = -EB -O2 -Wall -Wfatal-errors -mips1 -c -s -std=c99 -msoft-float -I$(INCLUDE)
```

Listing 4.1: compilerflags, CFLAGS of makefile

There was another flag list **CFLAGS_32R2** added, which uses the flag **-mips32r2** instead of -mips1. This can be seen in listing 4.2.

```
CFLAGS_32R2 = -EB -O2 -Wall -Wfatal-errors -mips32r2 -c -s -std=c99 -msoft-float -I$(
    INCLUDE)
```

Listing 4.2: compilerflags, CFLAGS_32r2 of makefile

The function to handle an interrupt was implemented in C in the file isr.c, which will be explained later on. The code seen in listing 4.3 was added to the makefile to make the compiler compile the file isr.c for an MIPS32r2 architecture.

```
isr.o: isr.c
	$(SIM_MESSAGE) $(MAIN) $<
	$(MIPS_GCC) $(CFLAGS_32R2) $< -o $@
```

Listing 4.3: compilation of isr.c

The compiled code includes instructions which are not part of the MIPS-I instruction set. These instructions were implemented additionally.

4.1.3 Linker Script

The linker script is the file **slow_memory.lds**. The code seen in figure 4.4 results in declaring the start of the section **.isr** to the address **0x35000**.

```
  .isr 0x35000 :
    {
      *(.isr)
    }
```

Listing 4.4: linking of the section .isr

The address was chosen because it is precedes the data, which starts at **0x40000**, but is located after the original jpeg2000 program code, which ends at **0x305c7**.

4.1.4 Script lst2files.pl

There are several scripts running in the background of the compilation process to create and provide data needed for the simulation by modelsim. One of these scripts is **lst2files.pl** which had to be modified because of the added section. In order to recognize the new section, the expression **($1 eq "isr")** was

added to line 280 of the script (see listing 4.5). This is necessary to provide the correct data for the simulation.

```
280        if( ($1 eq "text") || ($1 eq "rodata") || ($1 eq "data") || ($1 eq "sdata") || ($1
               eq "sbss") || ($1 eq "isr") || ($1 eq "simple") ){
281            $code_flag = 1;
282        }
```

Listing 4.5: Modification of the lst2files script.

4.2 Implementation of Coprocessor0

To handle interrupts, the MIPS-ISA requires the implementation of coprocessor0, which is one of four coprocessors that extend the ISA functionality. Therefore the **CP0 registers** and the interactions with them need to be implemented. The full definition of the CP0 register set can be found in [15]. For interrupt handling, following registers are of interest:

- **$12 - Status Register:** The status register stores the states of the processor. The value at bit 0 represents the **IE**-field, containing the status if **i**nterrupts are **e**nabled.
- **$13 - Cause Register:** The cause register contains the cause of an exception, interrupts are a special case of exceptions (asynchronous exceptions).
- **$14 - Exception Program Counter:** The **EPC** contains the address where to return after an exception has occurred and has been handled.

The implementation of the interaction is described in chapter 4.4. The implementation of the CP0 registers was done in the file ***plasma_reg_bank.vhd***. To interact with the CP0 register set the following input- and output ports were added: ***c0_rd_index, c0_source_out, c0_write, c0_read, status_ie_en, interrupt_enable_sc, epc_out. epc_in*** and ***epc_we***. The modified input- and output port list can be seen in listing 4.6. Their use will be described subsequently. Note that the input port **c0_read** was added for an earlier implementation, the recent version does not use it.

```
43 entity plasma_reg_bank is
44     generic(
45       DEBUG_FLAG                : string := "OF"
46     );
47     port(
48       clk                       : in  std_logic;
49       reset                     : in  std_logic;
50       rs_index                  : in  t_mips_reg_addr;
51       rt_index                  : in  t_mips_reg_addr;
52       rd_index                  : in  t_mips_reg_addr;
53       reg_dest_new              : in  t_plasma_word;
54       reg_source_out            : out t_plasma_word;
55       reg_target_out            : out t_plasma_word;
56       c0_rd_index               : in  t_mips_reg_addr;
57       c0_source_out             : out t_plasma_word;
58       c0_write                  : in  std_logic;
59       c0_read                   : in  std_logic;
60       status_ie_en              : in std_logic;
61       interrupt_enable_sc       : in std_logic;
62       epc_out                   : out t_plasma_word;
63       epc_in                    : in  t_plasma_word;
64       epc_we                    : in std_logic
65     );
66 end entity plasma_reg_bank;
```

Listing 4.6: Modified input- and output port list of the reg bank.

To implement the register bank the signal **c0_reg_bank** of type **t_reg_bank** was added as shown in listing 4.7. t_reg_bank is defined in *plasma_pack.vhd* and represents a register bank consisting of 32 registers.

```
74    -- CP0 register bank
75    signal c0_reg_bank              : t_reg_bank;
```

Listing 4.7: Representation of the CP0 register bank.

To modify the write signal detection, the signals, ***c0_write_en*** and ***epc_we_signal*** were added like in listing 4.8. They are connected to the input ports *c0_write* and *epc_we*. The signal ***write_en*** is set if the destination register is not register $0, which is never a legal destination because its value always contains zero as value and cannot be changed. This can be seen in listing 4.9.

```
79    -- write signal detection
80    signal dest_zero                : std_logic;
81    signal write_en                 : std_logic;
82    signal c0_write_en              : std_logic;
83    signal epc_we_signal           : std_logic;
```

Listing 4.8: Signals for write enable detection.

```
95 -- -------- WRITE ENABLE DETECTION ------------------------
96 dest_zero <= '1' when rd_index = b"0_0000" else '0';        -- rs0 is always zero
97 write_en  <= not dest_zero;
98 c0_write_en <= c0_write;
99 epc_we_signal <= epc_we;
```

Listing 4.9: Write enable detection.

The write process, which is a synchronous process, is triggered by rising edges of the clock signal. Like the regular register bank, the CP0 register bank is initialized with zeros when a reset occurs (see listing 4.10).

```
108          if reset = '1' then
109             for i in 2**PLASMA_REG_ADDR_WIDTH - 1 downto 0 loop
110                mem_reg_bank( i ) <= PLASMA_ZERO_WORD;
111                c0_reg_bank( i ) <= PLASMA_ZERO_WORD;
112             end loop;
```

Listing 4.10: Initial state of the register banks when a reset occurs.

Because *write_en* does not contain any information about which register bank should be accessed, the original write process for regular registers was modified with the condition, that *c0_write_en* has to be zero. The condition that *write_en* needs to be true remains unchanged. This is shown in listing 4.11.

```
115             if write_en = '1' and c0_write_en = '0' then
116                mem_reg_bank( to_integer(unsigned(rd_index)) ) <= reg_dest_new_sig;
117             end if;
```

Listing 4.11: Modified write process:regular register bank.

To write the CP0 register bank, both signals *write_en* and *c0_write_en* have to be true.

```
119             if write_en = '1' and c0_write_en = '1' then
120                c0_reg_bank( to_integer(unsigned(rd_index)) ) <= reg_dest_new_sig;
121             end if;
```

Listing 4.12: Write process:CP0 register bank.

The IE-field of the status register ($12) can be accessed directly, therefore the signal ***status_ie_en*** has to be set to true. Then the value of the signal ***interrupt_enable_sc*** is stored into the IE-field. This can be seen in listing 4.13

```
123             if status_ie_en = '1' then
124                c0_reg_bank(12)(0) <= interrupt_enable_sc;
125             end if;
```

Listing 4.13: Write Process: IE-field of the status register.

If the write signal of the EPC ***epc_we_signal*** is set to true, the EPC register($14) stores the value of the incoming signal ***epc_in***. This can be seen in listing 4.14.

```
123             if epc_we_signal = '1' then
124             c0_reg_bank(14) <= epc_in;
125             end if;
```

Listing 4.14: Write Process: epc

Like the reading process of the regular register bank, the reading process of the CP0 register bank is implemented as asynchronous process. The value of the destination register is assigned to the output port **c0_source_out**. To be able to use the EPC value anytime, it was assigned to the output port **epc_out**. This can be seen in listing 4.15

```
138 -- CP0 read access is asynchronous
139 c0_source_out  <= c0_reg_bank(to_integer(unsigned(c0_rd_index)) );
140 epc_out  <= c0_reg_bank(14);
```

Listing 4.15: Read Process of the CP0 register bank.

4.3 Implementation of an Interrupt Functionality

To implement the interrupt functionality, the port list of the router was extended by the output port ***recv_interrupt***. The modified port list can be seen in listing 4.16.

```
entity router is
    generic (
      addr_intern      : t_noc_addr := b"0000"
          );
    port(
      -- GENERAL
      clk              : in  std_logic;
      reset            : in  std_logic;
      -- PLASMA INTERFACE
      address          : in  std_logic;
      we               : in  std_logic;
      rd               : in  std_logic;
      data_w           : in  t_plasma_word;
      data_r           : out t_plasma_word;
      -- INPUT BUS INTERFACE
      addr_in          : in  t_noc_addr;
      acc_in           : in  std_logic;
      ack_in           : out std_logic;
      data_in          : in  t_plasma_word;
      -- OUTPUT BUS INTERFACE
      addr_out         : out t_noc_addr;
      acc_out          : out std_logic;
      ack_out          : in  std_logic;
      data_out         : out t_plasma_word;
      recv_interrupt   : out std_logic
    );
end entity router;
```

Listing 4.16: Modified port list of the router

The output port *recv_interrupt* is set to true, when the signal **recv_flag** is set, which is the case as soon as the router receives any data. This can be seen in listing 4.17.

```
  recv_interrupt <= recv_flag;
```

Listing 4.17: Setting of the interrupt signal(router)

The output port *recv_interrupt* was passed on through the test bench *tb_platform* onto the new input port of the control unit ***intr***(listing 4.18).

```
    intr                          : in std_logic;
```

Listing 4.18: New added port *intr*

The new signal ***interrupt*** is then assigned to the port *intr* (listing 4.19).

```
    interrupt <= intr;
```

Listing 4.19: Assigning the *interrupt* signal to the *intr* port

There was a process implemented to set the signal ***epc_we_out*** to **1** when an interrupt occurs, and to **0** when the interrupt signal is false. *epc_we_out* is passed through the datapath onto the register bank port *epc_we* which enables the writing of the incoming address to the EPC register. The process can be seen in listing 4.20.

```
interrupt_process:
  process(  interrupt )
  begin

    if interrupt = '1' then
      epc_we_out <= '1';
    else
      epc_we_out <= '0';
    end if;

  end process;
```

Listing 4.20: Interrupt process (Control Unit)

Correspondingly, there was the input port ***intr*** added to the datapath, which is assigned to the signal ***interrupt***. When the interrupt is triggered, the signal ***epc_out*** is set to the value of the signal *pc_val* which contains the next address of the regular program flow. To jump to a fixed address, the address **0x35000** was encoded as **ISR_ADDR** in *plasma_pack.vhd* as shown in listing 4.21.

```
  constant ISR_ADDR                         : std_logic_vector         := x"0003_5000"; --
     addresse for the interrupt service routine
```

Listing 4.21: Interrupt Service Routine Address

To jump to the interrupt service routine (ISR) address, the pc multiplexer of the datapath was modified as shown in listing 4.22. The signal ***pc_val*** contains the regular value of the next instruction, for the case that no interrupt occurred[1]. If the interrupt signal is set the ISR address is assigned to the signal ***pc_value_with_delay_slot***. Otherwise the signal *pc_value_with_delay_slot* is assigned the regular pc value. If the instruction is not *eret*, the *pc_value_with_delay_slot* is passed on to the pc unit. The further details are explained in the description of the instruction *eret*.

[1] The condition that the signal pc_func contains the value PLASMA_PC_EPC does not occur, this option was added due to a previous implementation of *eret*, the latest implementation does not require it.

```
-- PC VALUE MUX
--
with pc_func select
  pc_val          <=  reg_imm_in      when PLASMA_PC_IMM,
                        reg_src_a_in    when PLASMA_PC_REG,
                        pc_out_branch   when PLASMA_PC_BRANCH,
                        epc_in          when PLASMA_PC_EPC,
                        pc_out_inc      when others;

with interrupt select
  pc_value_with_delay_slot          <=  ISR_ADDR when '1',
                        pc_val when others;

with no_delay_slot select
    pc_value                    <= epc_in when '1',
                             pc_value_with_delay_slot when others;
```

Listing 4.22: Interrupt Service Routine Address

Thus, when an interrupt occurs the regular pc address (signal *pc_val*) must to be stored to the EPC. A process, which is triggered by a rising interrupt signal edge, was implemented to assign the value of *pc_val* to the signal ***epc_out***. The signal *epc_out* is wired to the input port *epc_in* of the register bank. The process can be seen in listing 4.23.

```
interrupt_process:
  process(interrupt )
  begin
    if rising_edge( interrupt ) then
        epc_out <= pc_val;
    end if;
  end process;
```

Listing 4.23: Passing the pc value onto the EPC register

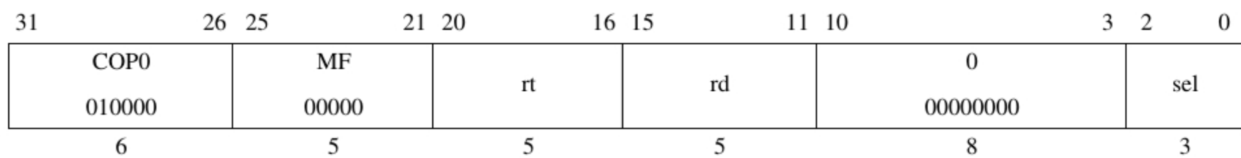
4.4 Implementation of additionally required Instructions

If the flag is set to compile for a MIPS32r2-ISA, the assembler code contains the following additional instructions **mfc0, mtc0, di, ei, ins, ehb** and **eret**. Their descriptions are extracted from [14]. The implementation is described in the following. The signals that are set in the *ID* stage are passed on after each clock to the signal representation of the next stage.

4.4.1 mfc0

The instruction **"mfc0"** is used to move a value from a CP0 register to a register of the regular register bank. Precisely the value of the regular register specified by the instruction field **rt** is copied to the CP0 register specified by the instruction field **rd**. The description of mfc0 can be seen in figure 4.1. Note that the **sel** field is not supported and must be **zero**.

Move from Coprocessor 0 **MFC0**



Format: `MFC0 rt, rd` **MIPS32**
`MFC0 rt, rd, sel` **MIPS32**

Purpose:

To move the contents of a coprocessor 0 register to a general register.

Description: `rt ← CPR[0,rd,sel]`

The contents of the coprocessor 0 register specified by the combination of rd and sel are loaded into general register rt. Note that not all coprocessor 0 registers support the sel field. In those instances, the sel field must be zero.

Figure 4.1: Description of the instruction mfc0.[14]

First the opcode **COP0** encoded by **"010000"** had to be implemented, this was done in the file **mips_instruction_set.vhd** as shown in listing 4.24, the instruction field **"MF"** was encoded as shown in listing 4.25. The opcode COP0 is used for all coprocessor0 instructions, MF can also be used for *move from* instructions of other coprocessors if they are implemented.

```
constant MIPS_OPCODE_COP0      : t_mips_opcode            := b"01_0000";        --
    coprocessor0 = interrupt
```

Listing 4.24: Encoding of opcode **COP0**

```
constant MIPS_FMT_MFC          : t_mips_reg_addr          := b"0_0000";         -- rt =
    fs
```

Listing 4.25: Encoding of **"MF"**

To be able to use values coming from CP0 as output source, the option **SRC_OUT_C0** was added to the result multiplexer of the datapath. When the control signal of the result multiplexer **"src_out_select"** holds the value *SRC_OUT_C0*, the output of the result multiplexer is set to the output of CP0 *c0_out*. The implementation can be seen in line **183** of listing 4.26 and line **437** of listing 4.27.

```
174  -- RESULT MUX
175  --
176  type t_src_out_select is (
177          SRC_OUT_ALU,
178          SRC_OUT_MULT,
179          SRC_OUT_SHIFT,
180          SRC_OUT_PC,
181          SRC_OUT_MEM_DATA,
182          SRC_OUT_C0);
```

Listing 4.26: Modification of the result MUX(control signal)

```
429  -- OUTPUT MUX
430  --
431  with src_out_select select
432    op_data_out       <= pc_out_inc          when SRC_OUT_PC,
433                         shift_out           when SRC_OUT_SHIFT,
434                         mult_out            when SRC_OUT_MULT,
435                         reg_mem_data_in     when SRC_OUT_MEM_DATA,
436                         c0_out              when SRC_OUT_C0,
437                         alu_out             when others;
```

Listing 4.27: Modification of the result MUX(data path)

When the **"mfc0"** is recognized in the decoding process, only the signal ***i_src_out_select*** has to be set to *SRC_OUT_C0*. The signal ***c0_read_en*** is set to **"1"** and passed on to the register bank, but it is not used anymore.

```
    when MIPS_OPCODE_COP0      =>
      case i_rs_dec is
        when MIPS_FMT_MFC      =>  i_src_out_select <= SRC_OUT_C0;
                                   c0_read_en  <= '1';
```

Listing 4.28: Decoding of the instruction **"mfc0"**

The default value of *c0_rd_index* was implemented to be *i_rd_index* (see listing 4.29 and the default value of *i_rd* is *i_rt_dec*. Therefore the required format is matched and the value is moved correctly.

```
c0_rd_index <= i_rd_dec;
```

Listing 4.29: Default value of c0_rd_index

4.4.2 mtc0

The next instruction **mtc0** is used to move the value of the regular register specified by the instruction field **rt** to the CP0 register specified by the instruction field **rd**. The description of mfc0 can be seen in figure 4.2 Note that the **sel** field is not supported and must be **zero**.

The opcode COP0 has already been implemented for the instruction *mfc0*, the instruction field **"MT"** was encoded like in listing 4.32 in mips_instruction_set.vhd. *MT* can also be used for *move to* instructions of other coprocessors if they are implemented. For the implementation of *mtc0* the *source a* multiplexer of the ALU was modified. The option SRC_ZERO_SEL was added like in listing 4.30. When the control signal of the *source a* multiplexer holds SRC_ZERO_SEL the output is set to *PLASMA_ZERO_WORD*(seen listing 4.31).

```
158  type t_src_select is (
159          SRC_REG,
160          SRC_OP_OUT,
161          SRC_MEM_OUT,
162          SRC_WB_OUT,
163          SRC_ZERO_SEL);
```

Listing 4.30: Modification of source MUX(control signal)

Move to Coprocessor 0 **MTC0**

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MT 00100	rt	rd	0 0000 000	sel
6	5	5	5	8	3

Format: `MTC0 rt, rd` **MIPS32**
`MTC0 rt, rd, sel` **MIPS32**

Purpose:

To move the contents of a general register to a coprocessor 0 register.

Description: `CPR[0, rd, sel]` ← `rt`

The contents of general register rt are loaded into the coprocessor 0 register specified by the combination of rd and sel. Not all coprocessor 0 registers support the the sel field. In those instances, the sel field must be set to zero.

Figure 4.2: Description of the instruction mtc0.[14]

```
266  -- SOURCE A MUX
267  --
268  with src_a_select select
269    src_a_in        <=  op_data_out        when SRC_OP_OUT,
270                        mem_data_out       when SRC_MEM_OUT,
271                        reg_bank_in        when SRC_WB_OUT,
272                        PLASMA_ZERO_WORD   when SRC_ZERO_SEL,
273                        reg_bank_a         when others;
```

Listing 4.31: Modification of source a MUX(data path)

```
    constant MIPS_FMT_MTC          : t_mips_reg_addr           := b"0_0100";         -- fs =
        rt
```

Listing 4.32: Encoding of **"MT"**

The **ALU** input a is set to be **0**, which was done by setting the control signal of the *source a* multiplexer ***src_a_select*** to ***SRC_ZERO_SEL***. The default value for the *source a* input is the value that comes from the register defined by the **rs**-field. The rs-field is not defined for this instruction, the MT-field (00100) is at its place instead. So it is always interpreted as register $4. Because the original forward logic does not recognize this, it is possible that it forwards a value written to register $4. To avoid this, the signal ***no_rs_forward*** is set to true, the forward logic was modified not to forward rs if the signal is set. This can be seen in listing 4.34. The *source b* input is by default set to the register defined by the **rt-field**. The **ALU** operates an unsigned add operation on the input values. The output value is the original value of input coming from *source b*. The signal result is saved in the register defined by the **rd**-field in the CP0 register bank.

The **ALU** operates an unsigned add operation on the value coming from the regular register defined by rt and adds **zero** from the *source a* input. The result, which is the original value, is stored in the register defined by the **rd-field**. Since the b input is not an immediate value, the signal ***imm_command.src_b*** was set to **"0"**.

```
    when MIPS_OPCODE_COP0     =>
      case i_rs_dec is
        when MIPS_FMT_MTC     =>        i_alu_func              <= PLASMA_ALU_ADDU;
                                        src_a_select   <=    SRC_ZERO_SEL;
                                        imm_command.src_b           <= '0';
                                        i_rd                       <= i_rd_dec;
                                        reg_addr.rt                   <= i_rt_dec;
                                        i_c0_write_en  <= '1';
                                        no_rs_forward  <= '1';
```

Listing 4.33: Decoding of the instruction **"mtc0"**

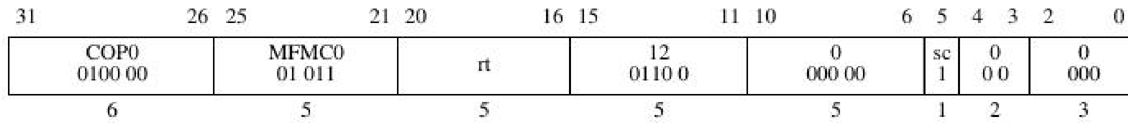
```
-- -------- A SOURCE FORWARDING -----------------
--

   if forward_flags.ex.rs   = '1' and no_rs_forward = '0'  then  src_a_select  <=
       SRC_OP_OUT;
elsif forward_flags.mem.rs  = '1' and no_rs_forward = '0'  then  src_a_select  <=
    SRC_MEM_OUT;
elsif forward_flags.wb.rs   = '1' and no_rs_forward = '0'  then  src_a_select  <=
    SRC_WB_OUT;
end if;
```

Listing 4.34: Modification of the forward logic for rs

4.4.3 di/ei

The instructions ***di*** and ***ei*** are alike except in one detail, *di* disables the IE-bit of the status register, *ei* enables the IE-bit of the status register. Their machine code differs only at bit 5, for the *ei* instruction this bit is set to **1** and for the *di* instruction this bit is set to **"0"**. Both instructions return the previous value of the status register to the regular register with the index determined by the rt-field. The descriptions can be found in listing 4.3 and 4.4.

Enable Interrupts **EI**



Format: `EI` **MIPS32 Release 2**
`EI rt` **MIPS32 Release 2**

Purpose:

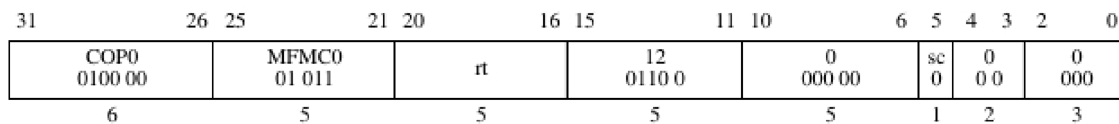
To return the previous value of the *Status* register and enable interrupts. If EI is specified without an argument, GPR r0 is implied, which discards the previous value of the Status register.

Description: `rt ← Status; Status`$_{IE}$ `← 1`

The current value of the *Status* register is loaded into general register rt. The Interrupt Enable (IE) bit in the *Status* register is then set.

Figure 4.3: Description of the instruction ei.[14]

Disable Interrupts **DI**



Format: `DI` **MIPS32 Release 2**
`DI rt` **MIPS32 Release 2**

Purpose:

To return the previous value of the *Status* register and disable interrupts. If DI is specified without an argument, GPR r0 is implied, which discards the previous value of the Status register.

Description: `rt ← Status; Status`$_{IE}$ `← 0`

The current value of the *Status* register is loaded into general register *rt*. The Interrupt Enable (IE) bit in the *Status* register is then cleared.

Figure 4.4: Description of the instruction di.[14]

The field *MFMC0* was encoded with **"010110"**, like in listing 4.35. The implementation of the return value is exactly like a *mfc0* instruction.

```
constant MIPS_FUNC_MFMC0       : t_mips_format             := b"0_1011";        --
    implements di and ei depending on the sc field
```

Listing 4.35: Encoding of **"MFMC0"**

The implementation of the return value is exactly like a *mfc0* instruction. The signal ***i_src_select*** was set to ***SRC_OUT_C0*** to implement the *move to* part of the instruction. The signal ***c0_read_enable*** was set to true, but the signal is not needed anymore. The logic to set the IE-field to the value of an incoming signal was already described in chapter 4.2, to enable the writing the signal ***i_status_ie_en*** was set to true. The

value of bit 5 of the instruction is passed onto the register banks. This can be seen in listing 4.36. The decoding of the instruction is shown in listing 4.37.

```
i_interrupt_enable_sc <= instr_dec(5);
```

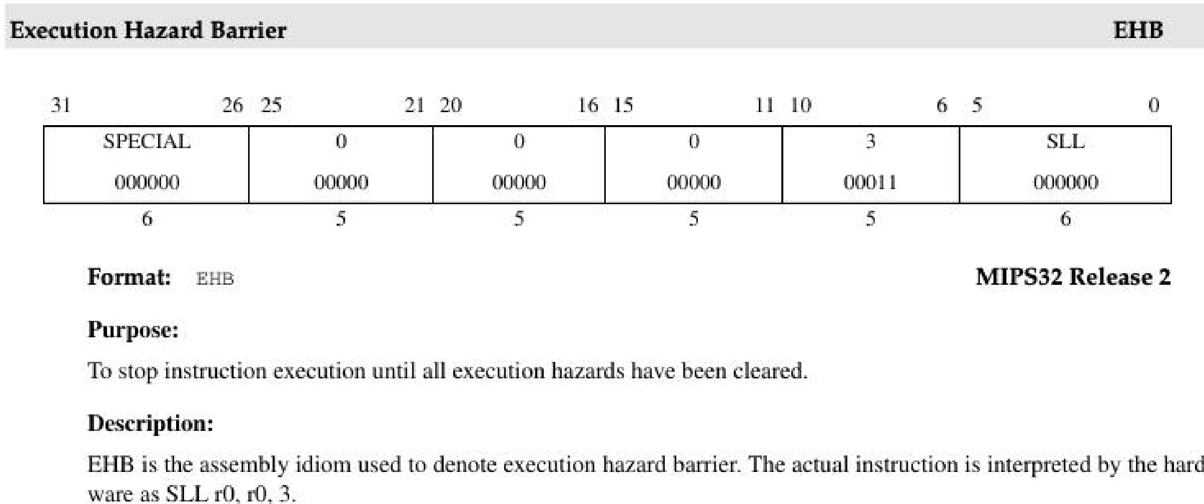
Listing 4.36: Passing bit 5 of the instruction to the register bank

```
    when MIPS_OPCODE_COP0      =>
      case i_rs_dec is
       when MIPS_FUNC_MFMC0    =>        c0_read_en   <= '1';
                                         i_src_out_select <= SRC_OUT_C0;
                                         i_status_ie_en  <= '1';
```

Listing 4.37: Decoding of the instruction **"di"** and **"ei"**

4.4.4 ehb

The MIPS documentation, seen in listing 4.38, states that the instruction ***ehb*** requires a MIPS32r2 (or higher) ISA, nevertheless MIPSI is able to interpret the instruction. *ehb* is an assembler idiom interpreted as **SLL r0,r0, 3** which does not change any register since the value of register **r0** (also called $0) is always **zero** and cannot be modified. Therefore, values written to **$0** cannot be forwarded. *ehb* stops the excecution of instrcutions until all excecution dependencies have been resolved. There was only a comment added to mips_instruction_set.vhd to indicate that the instruction *ehb* can be interpreted. This can be seen in listing 4.38.



Execution Hazard Barrier **EHB**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	0 00000	0 00000	0 00000	3 00011	SLL 000000
6	5	5	5	5	6

Format: EHB **MIPS32 Release 2**

Purpose:

To stop instruction execution until all execution hazards have been cleared.

Description:

EHB is the assembly idiom used to denote execution hazard barrier. The actual instruction is interpreted by the hardware as SLL r0, r0, 3.

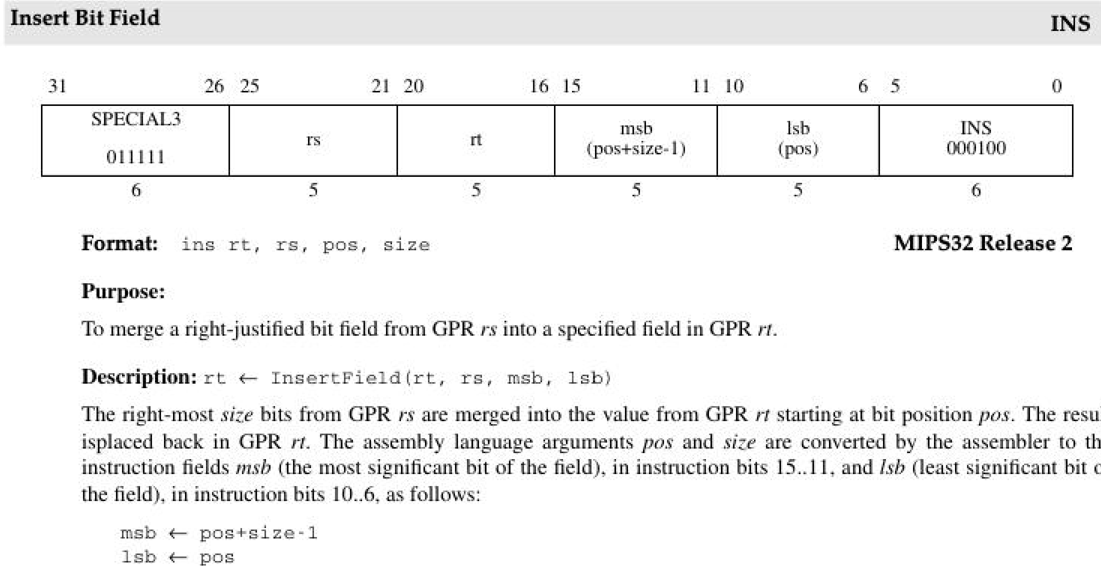
Figure 4.5: Description of the instruction ehb.[14]

```
-- constant MIPS_FUNC_EHB        : t_mips_function           := b"00_0000";      -- assembly
    idiom interpreted as SLL r0,r0,3
```

Listing 4.38: Pseudoimplementation of *ehb*

4.4.5 ins

The instruction ***ins*** is used to merge a right-justified bit from the register defined by the rs field into the register defined by the rt field. The exact description can be seen in figure 4.6. The more intuitive symbolic description can be seen in figure 4.7.



Insert Bit Field **INS**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL3 011111	rs	rt	msb (pos+size-1)	lsb (pos)	INS 000100
6	5	5	5	5	6

Format: `ins rt, rs, pos, size` **MIPS32 Release 2**

Purpose:

To merge a right-justified bit field from GPR *rs* into a specified field in GPR *rt*.

Description: `rt ← InsertField(rt, rs, msb, lsb)`

The right-most *size* bits from GPR *rs* are merged into the value from GPR *rt* starting at bit position *pos*. The result isplaced back in GPR *rt*. The assembly language arguments *pos* and *size* are converted by the assembler to the instruction fields *msb* (the most significant bit of the field), in instruction bits 15..11, and *lsb* (least significant bit of the field), in instruction bits 10..6, as follows:

```
msb ← pos+size-1
lsb ← pos
```

Figure 4.6: Description of the instruction ins.[14]

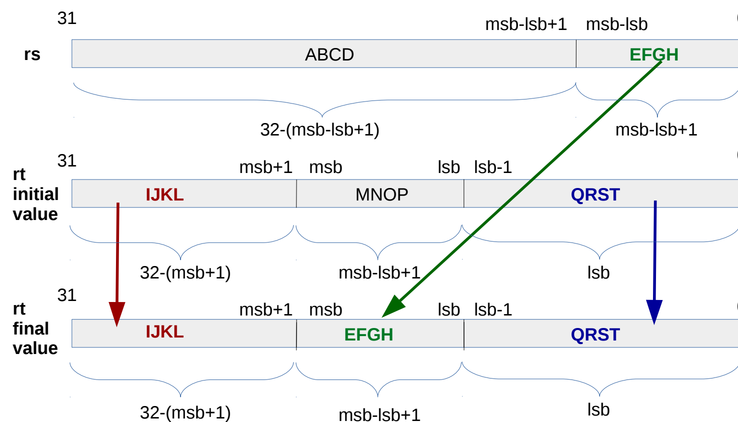


Figure 4.7: Symbolic description of the instruction ins.[14]

The opcode **SPECIAL3** encoded by **“011111”** was added to mips_instruction_set.vhd. This can be seen in listing 4.39.

```
constant MIPS_OPCODE_SPEC3     : t_mips_opcode             := b"01_1111";
```

Listing 4.39: Encoding of opcode **SPECIAL3**

The function **“INS”** was encoded with **“00100”** as seen in listing 4.40.

```
constant MIPS_FUNC_INS         : t_mips_function           := b"00_0100";
```

Listing 4.40: Encoding of the function **INS**

To operate the instruction *ins*, there was the new function control signal *PLASMA_ALU_INS* added to the ALU control. This can be seen in the last line of listing 4.41.

```
239   -- ALU CONTROL
240   --
241   type t_alu_function is (
242           PLASMA_ALU_NOTHING,
243           PLASMA_ALU_ADD,
244           PLASMA_ALU_ADDU,
245           PLASMA_ALU_SUB,
246           PLASMA_ALU_SUBU,
247           PLASMA_ALU_SLT_SIGNED,
248           PLASMA_ALU_SLT_UNSIGNED,
249           PLASMA_ALU_OR,
250           PLASMA_ALU_AND,
251           PLASMA_ALU_XOR,
252           PLASMA_ALU_NOR,
253           PLASMA_ALU_INS);
```

Listing 4.41: Modification of the result MUX(control signal)

Additionally there were the input ports ***alu_msb*** and ***alu_lsb*** added to the alu, as shown in listing 4.42.

```
entity plasma_alu is
    generic(
      FPGA_FLAG               : string := "OF"
    );
    port(
      alu_a_in                : in  t_plasma_word;
      alu_b_in                : in  t_plasma_word;
      alu_func                : in  t_alu_function;
      alu_msb                 : in  std_logic_vector;
      alu_lsb                 : in  std_logic_vector;
      alu_busy                : out std_logic;
      alu_out                 : out t_plasma_word
    );
end entity plasma_alu;
```

Listing 4.42: Modified port list of the ALU

To be able to use the values from the ports *alu_msb* and *alu_lsb*, the values were casted into the integer values ***msb*** and ***lsb*** as seen in listing 4.43. The required function was implemented like in the description and can be seen in figure 4.44.

```
msb := to_integer(unsigned(alu_msb));
lsb := to_integer(unsigned(alu_lsb));
```

Listing 4.43: Casting into the integer values *msb* and *lsb*

```
case alu_func is
 when PLASMA_ALU_INS        => alu_out <= alu_b_in(31 downto msb+1) &
                              alu_a_in( (msb - lsb) downto 0) &
                              alu_b_in( (lsb -1) downto 0);
```

Listing 4.44: Implementation of the (*ins*)-operation performed by the ALU

The decoding of the instruction *ins* can be seen in listing 4.45.

```
when MIPS_OPCODE_SPEC3             =>
  case i_func_dec is
    when MIPS_FUNC_INS     => report "INS used";
                              i_alu_func <= PLASMA_ALU_INS;
                              imm_command.src_b          <= '0';
                              reg_addr.rt                  <= i_rt_dec;
```

Listing 4.45: Decoding of the instruction **"ins"**

For debug purposes the reporting of the string **"INS used"** was implemented. To perform the *ins* instruction, the ALU function is set to *PLASMA_ALU_INS*. Since the *source b* input comes from a register and is not an immediate value, the signal *imm_command.src_b* is set to **"0"**. Since the used rt register (signal ***reg_addr.rt*** is by default set to ***register $0***, the signal is set to the value contained by the instruction rt-field (represented by the signal ***i_rt_dec***). The pipeline ensures a correct performance of the instruction.

4.4.6 eret

The instruction ***eret*** is used to return from an exception, interrupt or error trap. Despite the fact that it is a jump instruction, it has no delay slot, which means the next instruction following *eret* is not executed. The description can be found in figure 4.8. The opcode of *eret* is *COP0*, the function code is ***ERET***. The function code *ERET* was encoded with **"011000"** as shown in listing 4.46.

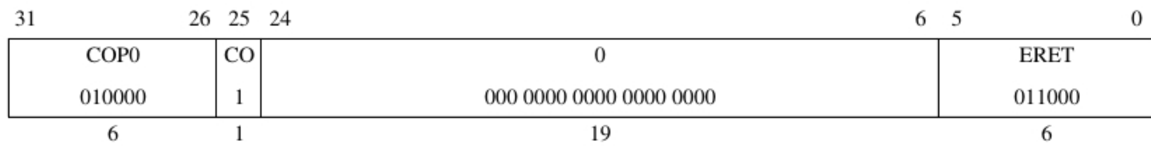
```
constant MIPS_FUNC_ERET        : t_mips_function           := b"01_1000";
```

Listing 4.46: Encoding of the function **ERET**

```
when MIPS_OPCODE_COP0      =>
  case i_func_dec is
    when MIPS_FUNC_ERET    => report "eret";
                              i_pc_func             <= PLASMA_PC_EPC;
                              i_no_delay_slot       <= '1';
```

Listing 4.47: Decoding of the instruction **"eret"**

Exception Return **ERET**



Format: `ERET` **MIPS32**

Purpose:

To return from interrupt, exception, or error trap.

Description:

ERET clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ in a Release 2 implementation, and returns to the interrupted instruction at the completion of interrupt, exception, or error processing. ERET does not execute the next instruction (i.e., it has no delay slot).

Figure 4.8: Description of the instruction eret.[14]

When the instruction *eret* is decoded the string **"eret"** is reported for debug purposes. The signal ***i_pc_signal*** is set to **1**, but for the latest implementation it is not needed anymore. The *pc_func* signal is controlling the pc multiplexer after the **EX-stage**. Because the *eret* instruction does not execute its suceeding instruction, it needs to control the pc multiplexer one stage earlier, meaning after the **EX-stage**. Therefore was the signal ***i_no_delay_slot*** set to **1**. This signal is directly connected to the pc multiplexer and can be evaluated after the ID-stage. If the instruction is *eret* the pc is set to the signal ***epc_in*** which contains the pc value when the interrupt occurred. This can be seen in listing 4.48.

```
with no_delay_slot select
     pc_value                    <= epc_in when '1',
                                    pc_value_with_delay_slot when others;
```

Listing 4.48: Returning to the epc address

When the program is started, the second core starts with the function *main_core2()*. It is trapped in an endless loop doing *nop* instructions. This can be seen in listing 4.49. The first core starts with the function *main_core1()* which does not differ from the original *main()* function of the jpeg2000 algorithm. The first loop that can be parallelized is located in *bmp2image.c*.

```
int main_core2() {

        while(1){
        for(int i = 0;i<100;i++){
                    __asm__("nop");
                    __asm__("nop");
                    __asm__("nop");
                    __asm__("nop");
                    __asm__("nop");
                    __asm__("nop");
                    __asm__("nop");
                    __asm__("nop");
                    __asm__("nop");
                    __asm__("nop");
                    __asm__("nop");
        }
        }
        return 0;
}
```

Listing 4.49: The function main_core2()

The parallelized loop was implemented in the function ***f_noc_bmptoimage()***. This can be seen in listing 4.50.

```
void f_noc_bmptoimage(){
  unsigned int i;                                 // loop variable
  for(i = v_noc_bmptoimage->start; i < v_noc_bmptoimage->end; i++ ){     // main loop
     v_noc_bmptoimage->RGB[i] = v_noc_bmptoimage->IN[v_noc_bmptoimage->index++];
  }
  ack_received = 1;
}
```

Listing 4.50: The function f_noc_bmptoimage()

When reaching the location of the parallelized loop, the first core sets the volatile int variable ***ack_received*** to "**0**". Then it sends a pointer to the function *f_noc_bmtoimage()* to the second core. This can be seen in listing 4.51.

```
    ack_received = 0;
    router_send(1, &f_noc_bmptoimage );                    // pointer to the function
```

Listing 4.51: The function f_noc_bmptoimage()

The structure ***v_noc_bmptoimage*** that is required by *f_noc_bmtoimage()* is initialised by core0, as shown in listing 4.52.

```
v_noc_bmptoimage = (t_noc_bmptoimage *) malloc(sizeof(t_noc_bmptoimage));

v_noc_bmptoimage->start = ((3 * W + PAD) * H) >> 1;
v_noc_bmptoimage->end   = (3 * W + PAD) * H;
v_noc_bmptoimage->index = index + (((3 * W + PAD) * H) >> 1);
v_noc_bmptoimage->IN    = IN;
v_noc_bmptoimage->RGB   = RGB;
```

Listing 4.52: Initialization of the structure v_noc_bmtoimage

Therefore, the first core computes its own share of the parallelized loop. When finished, it stays in a loop idling until ***ack_received*** is set to 1. After it received an acknowledgement from core2 it continues with the regular program flow. This can be seen in listing 4.53.

```
while(ack_received == 0){
   __asm__("nop");
};
```

Listing 4.53: Core1 waiting for acknowledgement of core2

When the second core receives data, it jumps to the function ***ISR()***. The value of the address pointer ***loop_function*** is set to the address received by the router which is the address of the function *f_noc_bmptoimage()*. The command ***"*(loop_function)();"*** executes *f_noc_bmptoimage()*. The variable *ack_received* is set to **"1"** at the end of the loop. Since the last instruction of the compiler created function is *eret*, core2 returns to it's previous address, which means it return to the nop-loop. The C source code of the function ISR() can be seen in listing 4.54.

```
void  __attribute__ ((interrupt, , section(".isr")))  ISR(){

  void (* loop_function)(void);
  loop_function = router->data;
  (*loop_function)();

   return;
 }
```

Listing 4.54: The function ISR()

The implementation for the two other functions that can be parallelized works similiar. The function f_noc_t1_encode_cblks()(listing 4.55) is located in ***t1.c***. The function f_noc_t1_encode_cblks()(listing 4.56) is located in ***tcd.c***.

```
void f_noc_t1_encode_cblks(){
   if( !opj_t1_encode_cblks( v_noc_t1_encode_cblks->t1,
                             v_noc_t1_encode_cblks->tile,
                             v_noc_t1_encode_cblks->tcp,
                             v_noc_t1_encode_cblks->mct_norms ) ){
sim_message(9, 0);  //sim_message("return FALSE",0)
    sim_stop();
  }
  ack_received = 1;
}
```

Listing 4.55: The function f_noc_t1_encode_cblks()

```
void f_noc_tcd_rateallocate(){
   if( !opj_tcd_rateallocate( v_noc_tcd_rateallocate->tcd,
                              v_noc_tcd_rateallocate->dest,
                              v_noc_tcd_rateallocate->p_data_written,
                              v_noc_tcd_rateallocate->len,
                              v_noc_tcd_rateallocate->cstr_info) ){
sim_message(9, 0);  //sim_message("return FALSE",0)
    sim_stop();
  }
  ack_received = 1;
}
```

Listing 4.56: The function f_noc_tcd_rateallocate()

4.6 The Interrupt Service Routine

To handle interrupts the assembler routine created by the compiler uses only the registers ***k0*** and ***k1***. Those registers are reserved for interrupt purposes. The other registers should not be used, because this could result in crushing registers that are still needed. First the values from the Cause and EPC register are moved to the registers *k0* and *k1* (see listing 4.57).

```
mfc0    k0,c0_cause
mfc0    k1,c0_epc
```

Listing 4.57: Moving the cause and the exception cause address to *k0* and *k1*.

The EPC value is then stored to the memory (listing 4.58).

```
addiu   sp,sp,-104
sw      k1,100(sp)
```

Listing 4.58: Storing the exception cause address to the memory

In order to compute the recent CP0 status, the code seen in listing 4.59 does the following: The status of CP0 is moved to the regular register bank. By using the instrucion *ins* the information from the cause register is merged into the preceding CP0 status. The second *ins* instruction sets some bits determining the operation mode to **zero**[1]. The recent CP0 status value is then moved back to the CP0 status register.

[1] The concrete details of the operation mode are for this implementation not relevant.

```
mfc0    k1,c0_status
srl     k0,k0,0xa
sw      k1,96(sp)
ins     k1,k0,0xa,0x6
mflo    k0
ins     k1,zero,0x1,0x4
sw      k0,92(sp)
mfhi    k0
sw      k0,88(sp)
mtc0    k1,c0_status
```

Listing 4.59: Computation of the recent CP0 status.

In the next part the non-preserving registers are saved. The isr-routine as defined by the C source code is executed. In this case it is just a *jalr* to the function received by the router. The assembler code can be seen in listing 4.60.

```
sw      v0,20(sp)
lui     v0,0x120
sw      ra,84(sp)
addiu   v0,v0,8196
sw      t9,80(sp)
sw      t8,76(sp)
sw      t7,72(sp)
sw      t6,68(sp)
sw      t5,64(sp)
sw      t4,60(sp)
sw      t3,56(sp)
sw      t2,52(sp)
sw      t1,48(sp)
sw      t0,44(sp)
sw      a3,40(sp)
sw      a2,36(sp)
sw      a1,32(sp)
sw      a0,28(sp)
sw      v1,24(sp)
sw      at,16(sp)
lw      v0,4(v0)
jalr    v0
nop
```

Listing 4.60: Register saving and call of the received function.

Before the non-preserving registers are restored, interrupts are disabled by the use of the instruction *di*. The instruction *ehb* resolves execution hazards. This is shown in listing 4.61.

```
di
ehb
lw      k0,92(sp)
lw      ra,84(sp)
lw      t9,80(sp)
mtlo    k0
lw      k0,88(sp)
lw      t8,76(sp)
lw      t7,72(sp)
lw      t6,68(sp)
lw      t5,64(sp)
lw      t4,60(sp)
lw      t3,56(sp)
mthi    k0
lw      t2,52(sp)
lw      t1,48(sp)
lw      t0,44(sp)
lw      a3,40(sp)
lw      a2,36(sp)
lw      a1,32(sp)
lw      a0,28(sp)
lw      v1,24(sp)
lw      v0,20(sp)
lw      at,16(sp)
```

Listing 4.61: Restoring of the non-preserving register.

In the last part of the routine, the original EPC and status register values are loaded from the memory and moved to CP0. Finally, the processor returns to the regular program flow by the use of the instruction *eret* (see listing 4.62).

```
lw      k0,100(sp)
mtc0    k0,c0_epc
lw      k0,96(sp)
addiu   sp,sp,104
mtc0    k0,c0_status
eret
```

Listing 4.62: Returning to the regular program flow.

5 Conclusion

5.1 Evaluation

The main goal of this work, was to implement a solution that does not require the second core to constantly read the memory. This was achived. The first core on the other hand, is still permanently reading the memory while waiting for acknowledgement of the second core. The implemented protocol does not provide a solution to this matter, because the conditions are different for the first core. The second core is trapped in a nop-loop as default state and is interrupted to call a given function. The first core on the other hand is running the jpeg2000 algorithm and idling in a nop-loop when waiting for the second core to finish. The approach to interrupt it when the second core has finished its loop had no success. The implemented solution works for the given source code, showing that the implemented concept does solve the problem in general. But the implemented solution is not optimal. With the given platform it is only possible to check whether the jpeg2000 picture is computed correctly. There is no guarantee that all implemented instructions work under any circumstances. The correct return to the interrupted address does not work properly when the instruction before the interrupted instruction was a jump or branch. The implementation of mfc0 and mtc0 does not provide a proper forward and stall logic for this instructions. Nevertheless, the implementation solves the required tasks. Additionally, it contains more functions than exploited by the modified hardware so far. The provided solution can easily be adopted to other parallelized software by adjusting the C source code only.

5.2 Outlook

5.2.1 Scalability

The implemented solution does scale well for more than two cores, just the partitioning of the parallelized loops has to be adjusted and an additional variable has to be added for every additional core to store the status of the acknowledgement. The solution can easily be adjusted for a NoC with more than two cores. To exploit parallelism further more the platform could be expanded to three or more cores.

5.2.2 Delay of Interrupt

To solve the problem that the return to an interrupted instruction does not work properly if the previous instruction was a jump can easily be solved with a simple solution. There has to be a signal implemented in the control unit which stores the information if the previous instruction was a jump or branch. The interrupt signal which is wired from the router directly to the datapath and to the control unit, has to be wired to the control unit only. The control unit evaluates whether an interrupt may take place or may be delayed and sets the signals ***interrupt_allowed*** and ***interrupt_delayed*** accordingly. Those signals are required to be forwarded to the datapath. The processes that were initially triggered by the *interrupt* signal is changed to trigger on *interrupt_allowed* and *interrupt_delayed*.

5.2.3 Correct Implementation of mfc0 and mtc0

The forward logic of mfc0 and mtc0 was not correctly implemented. A value that is moved from the CP0 register bank to the regular register bankhas to go through the ALU, which performs an operation that does not change the original value. This way, the result of the ALU is recognized by the forward logic and is forwarded properly. The value that is moved from the regular register bank to the CP0 register bank has to be wired directly to the multiplexer of the WB-stage and to be chosen by the control signal. That way it cannot occur that a false value, for example the result of the ALU, is forwarded.

5.2.4 Acknowledgements

The given platform does not support package acknowledgement. A possible way to improve the platform is to implement a package acknowledgement. If the delay of interrupt is implemented it is also useful to implement an acknowledgement for received interrupts.

5.2.4.1 Advanced Features for Interrupts

The implemented instructions *di* and *ei* disable or enable interrupts in the IE-field of the status register. The implemented solution does not check the IE-field, because in the implemented protocol, an interrupt never occurs when the IE-field is set to disable interrupts. For more functionality, the evaluation of the interrupt enable bit can be implemented. The routine created by the compiler also loads the cause of an interrupt. The implemented solution does not write any cause for an interrupt because there is only one cause that leading to an interrupt. For enhanced functionality the evaluation of the cause can be implemented. The address where to jump when an interrupt occurs can be implemented by using different addresses for different causes.

5.2.5 Full Portation to MIPS32r2

The solution implements partial functionalities of a MIPS32r2 architecture. To further improve the functionality of the platform, the platform could be changed to implement all requirements of a MIPS32r2 ISA.

6 Bibliography

[1] http://msdn.microsoft.com/en-us/library/vstudio/hh265136.aspx. [Online; accessed 16-October-2014].

[2] http://www.openjpeg.org/. [Online; accessed 20-October-2014].

[3] COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY. www.tenouk.com/ModuleW.html. [Online; accessed 16-October-2014].

[4] “6.30 Declaring Attributes of Functions”. https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html#Function-Attributes. [Online; accessed 20-October-2014].

[5] "Stop-and-Wait" protocol. http://www.isi.edu/nsnam/DIRECTED_RESEARCH/DR_HYUNAH/D-Research/stop-n-wait.html. [Online; accessed 17-October-2014].

[6] http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg, 2008. [Online; accessed 16-October-2014].

[7] Andrew S. Tanenbaum. *Computernetzwerke*. Person Studium, 4. edition, 2003.

[8] David A. Patterson and John L. Hennessy. *Computer Oranziation and Design - The Hardware/Software Interface*. Morgan Kaufmann Puplishers, United States of America, 3. edition, 2004.

[9] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[10] Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 23(3):1369–1386, august 2012.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Puplishers, United States of America, 5. edition, 2011.

[12] Roland Kluge. C/C++ Praktikum - Einführung. Presentation, 2014. Technische Universität Darmstadt.

[13] Manuel Bied and Felix Pels. Partial parallelization of the openjpeg implementation of the jpeg2000-algorithm for a network-on-a-chip with two cores. Technical report, Technische Universität Darmstadt, Fachgebiet Integrierte Elektronische Systeme.

[14] MIPS Technologies, Inc. *MIPS32TM Architecture For Programmers Volume II: The MIPS32TM Instruction Set*, 2001.

[15] MIPS Technologies, Inc. *MIPS32TM Architecture For Programmers Volume III: The MIPS32TM Privileged Resource Architecture*, 2001.

[16] Sarah Harris and David Harris. *Digital Design and Computer Architecture. From Gates to Processors*. Morgan Kaufmann Puplishers, 2007.

[17] Stephen G. Kochan. *Programming in C*. Sams Publishing, 3. edition, 2004.

[18] Thomas Rauber and Gudula Rünger. *Parallele Programmierung*. Springer, Berlin Heidelberg New York, 2. edition, 2007.

[19] Urs Gleim and Tobias Schüle. *Multicore-Software*. dpunkt.verlag, Paderborn, 1. edition, 2012.