ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester Thesis

Design and Implementation of an Emulator for the Intel IXP 2400 Network Processor

Author
Christian Kummer

supervising assistant
Lukas Ruf

supervising professor
Prof. Bernhard Plattner

SA-2006-17

Computer Engineering and Networks Laboratory
ETH Zurich

SS 2006

Abstract

The goal of this thesis was the completion of an emulator for the Intel IXP 2400 network processor. An emulator behaves exactly like the real system but with the advantage to inspect all data and states inside the machine. This is a great benefit for debugging software to be run on the target system.
The Intel IXP 2400 consists of an XScale Core processor and an array of 8 fully programmable multi-threaded microengines. The core processor as well as the microengines have access to several external units such as different kind of memories or a hashing unit. This thesis presents the design and implementation of the microengines and the external units. One of the principal aims of the project was that the emulation is cycle accurate. This means that the most part of the internals of the processor had to be imitated.

Kurzfassung

Ziel dieser Semesterarbeit war die Vervollständigung eines Emulators für den Intel IXP 2400 Netzwerk Prozessor. Ein Emulator verhält sich genau gleich wie das reale System, bietet aber den entscheidenenden Vorteil, dass sämtliche Daten und Zustände des Prozessors sichtbar sind. Dies ist für das Debugging von Software für das Zielsystem enorm hilfreich.
Der Intel IXP 2400 besteht aus einem XScale Core Prozessor sowie einem Array von 8 vollständig programmierbaren Microengines. Jede Microengine hat Zugriff auf verschiedene externe Einheiten wie Speicher oder eine Hash Einheit.
In dieser Arbeit ging es insbesondere um die Implementierung der Microengines sowie den externen Funktionseinheiten.
Eines der Hauptziele der Arbeit bestand darin, den Emulator so zu implementieren, dass die Emulation Zyklen genau mit dem realen Prozessor ist. Das bedeutet, dass ein Grossteil der internen Funktionsweise des Prozessors nachgebildet werden musste.

Contents

1 Introduction

1.1 Motivation

A software emulator is a program that runs on a specific platform and tries to mimic the behavior of another platform, either software or hardware. In other words, it presents an environment to a binary which behaves exactly the same way as the original one.
An emulator has the following main uses in practice:

- Software written for systems which aren't in use anymore can still be run.
- In case software is not developed on the actual target platform, as it is for example the case for embedded systems, software can be tested using an emulator running on the development platform.
- Emulators can be used to model and test new architectures before the first prototype is available.
- An emulator provides a very good testing environment with increased debugging capabilities.

This thesis deals with the emulation of another computer architecture, namely the Intel IXP 2400 network processor ([1], [2], [3], [4]).
There exist basically two approaches in building such an emulator. The first method, the interpreter, makes advantage of the fact that almost all of today's computers are based on the same model, namely on the von Neumann architecture. This architecture principally consists of a processor and one shared memory for instructions and data.
The processor fetches an instruction from this memory, decodes this instruction and then carries out the corresponding operation. These steps are performed in a repetitive manner.
An interpreter basically imitates these same steps in software using a infinite loop, as it is roughly sketched below.

```
repeat
    fetch the next instruction from the emulated memory
    decode the instruction
    execute the instruction
until stopped
```

The second approach to design an emulator is called binary translation [10]. A binary translator converts binary data written for the original machine to binary data for the host system.
A binary translator is rather difficult to implement but results in better execution speed compared to an interpreting emulator. A disadvantage of binary translation is portability. Whereas an interpreter is not bound to any platform, a binary translator generally only works for one environment.

The emulator presented in this thesis is an interpreter.

1.2 Related Work

- PowerNP 4GS3 [7] Emulator [9]. This Emulator was designed and implemented by Hanspeter Hug. It is based on the PCE (Personal Computer Emulator) [8] emulator Framework developed by the same author. Despite its name, this framework is very flexible and allows the emulation of almost any system.

- NePSim [11]. A Network Processor Simulator with Power Evaluation Framework. This is an open-source simulation infrastructure for analyzing and optimizing NP design and power dissipation at architecture-level.
  The nepsim simulator does not directly work on binaries but rather on the immediate list file, containing all the assembly instructions in text format.

1.3 Problem statement

The purpose of this project is the completion of an emulator for the Intel IXP 2400 Network Processor. The existing emulator is based on the PCE Emulator Framework and was developed by Hanspeter Hug at the Computer Engineering and Networks Laboratory at ETH Zurich.
This basic implementation is capable of running the Linux Operating System on the XScale core processor but it completely lacks the emulation of the packet processors and of most external units.
This project presents the design and implementation of most of these missing parts. In particular the packet processors as well as most of the external units have been developed in the course of this thesis.
One of the primary goals of the project was that the emulation is binary compatible as well as cycle accurate. This means that real code can be run on the emulator without any modification. Another goal was to choose the architecture in such a way, that it can be integrated into the existing framework.
Beyond that, the Intel IXP 2400 poses the following challenges for the implementation of an emulator:

- The Intel IXP 2400 is a highly parallel architecture. There are eight microengines which run completely in parallel. Furthermore, each microengine has eight hardware-assisted threads. This means that the whole inter-thread communication as well as the arbitration of the threads needs to be imitated in software.

- The proper handling of instructions which have a certain latency. Any instruction accessing functional units outside of the execution datapath has a visible delay until the result is actually written or the operation is performed. Thus, the emulator must provide the means to postpone function calls.

- The implementation of the so-called generalized Thread Signaling which is unique to the microengines. After a memory request, a new thread can be scheduled, while the operation is still pending. As soon as the operation is complete, the functional unit has to assert a signal.
  Many details of the bus architecture need to be imitated for handling this signaling between the microengines and the external units.

1.4 Outline

Section 2 describes the most important parts of the Intel IXP 2400 Network Processor.
The architecture and implementation of the emulator is discussed in section 3.
Section 4 then explains how the emulator was tested and evaluated.
The final section summarizes the achievements and gives an overview of further work to be done in order to complete the IXP Emulator.

1.5 Acknowledgments

First of all I would like to express my gratitude to Lukas Ruf for giving me the opportunity to carry out this thesis and his support during the work.
Then I would also like to thank Hanspeter Hug for his introduction to the PCE framework and for the suggestions afterward.

2 The Intel IXP 2400 Network Processor

This section gives a short overview of the Intel IXP 2400 Network Processor. A more in-depth discussion of the Intel IXP 2400 and its programming model is given in [5] and [6].
Figure 2.1 shows the main components of the Intel IXP architecture and their interaction. Each major component is then briefly described in the following sections.

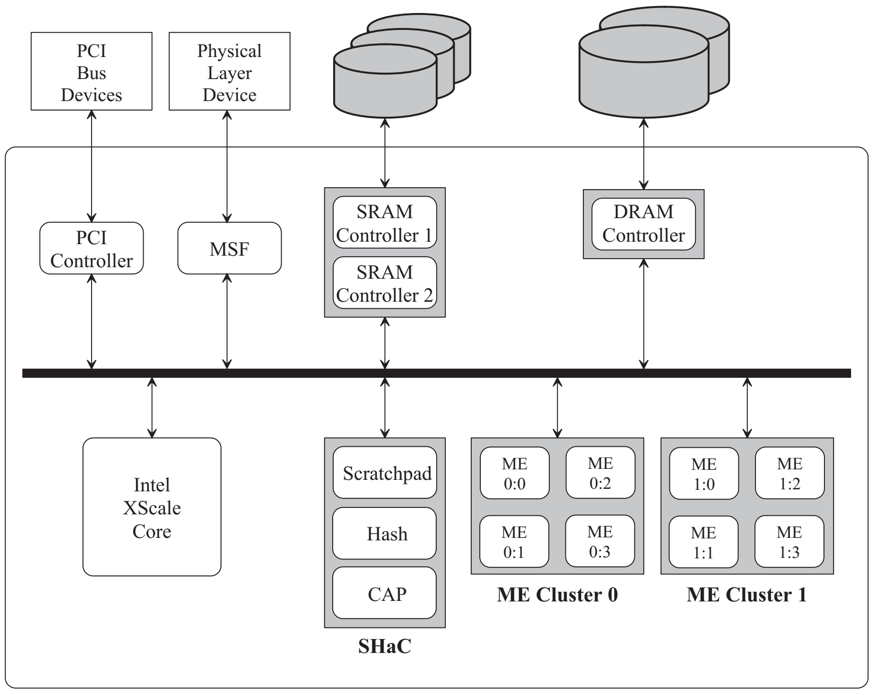


Figure 2.1: The Functional Units of the Intel IXP2400.

2.1 The Intel XScale Core Processor

The Intel XScale core processor is an embedded general-purpose RISC processor and is fully compatible with the ARM V5 architecture. It normally runs a conventional operating system such as Linux and is responsible for the initialization of the microengines. The core processor is also used for slow-path packet processing.

2.2 The Microengines

The Intel IXP2400 has eight fully programmable multi-threaded microengines which are grouped into two clusters. These microengines form the basis of the fast-path packet processing. Their instruction set is specifically designed for processing network data. Typical tasks which are handled by the microengines are:

- Packet reception from physical layer hardware

- Checksum calculation
- Header validation and classification
- Pattern matching and table lookup
- Queue Management
- Packet transmission to physical layer hardware

Figure 2.2 shows the functional blocks contained in each Microengine.

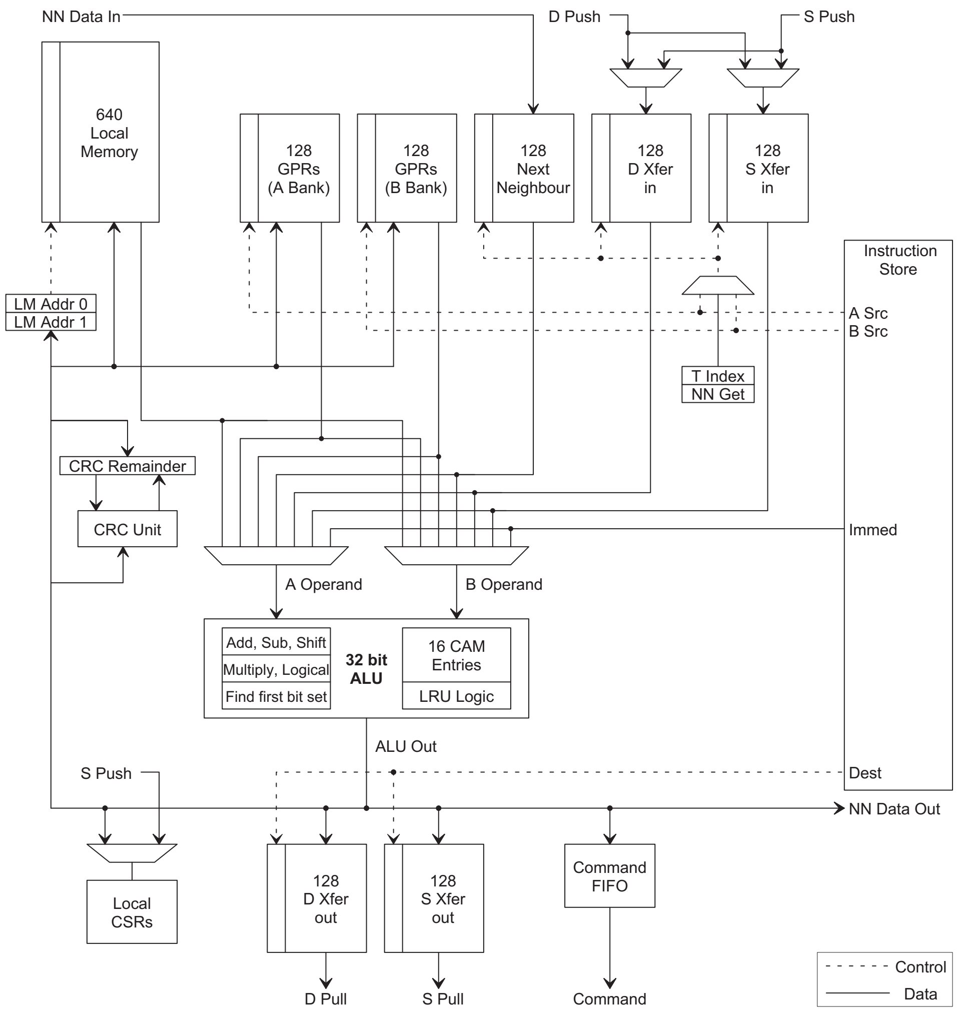


Figure 2.2: Block diagram of a Microengine.

2.2.1 Execution Pipeline

The execution pipeline of a microengine consists of six stages:

- Instruction fetch

- Instruction fetch
- Instruction decode
- Extract the operands
- Perform the necessary operations (ALU, ...)
- Write back the result to the destination register

The instruction fetch is divided into two steps since an instruction is 40-bit wide and not as usual only 32-bit. This instruction size is needed in order to have enough bits to represent all possible register combinations.

2.2.2 Instruction store

Each microengine has an own instruction store since it would need too much time to fetch an instruction from external memory. The instruction store has room for 4096 40-bit wide instructions and is initialized by the Intel XScale core processor.

2.2.3 Local Memory

Local Memory is addressable storage located in the Microengine and can therefore be accessed much faster than the external memory. It can store 640 32-bit words.

2.2.4 Registers

A Microengine has 256 General Purpose Registers (GPR), 512 Transfer Registers and 128 Next Neighbor Registers.
The GPRs are used for general programming purposes. They can either be accessed in thread-local or absolute mode.
The Transfer Registers are used for transferring data to and from the external units. Since the microengine accesses the external units in an asynchronous manner, these registers are needed to buffer the data to be transferred until the request is complete.
The Next neighbor registers provide a dedicated datapath for moving data from one microengine to an adjacent microengine.

2.2.5 Content Addressable Memory (CAM)

The content addressable memory is located in the ALU of each microengine. The CAM maintains an LRU Logic for 16 entries. Each entry has a 32-bit tag and a 4-bit state value.
This unit can be used to implement a cache or to provide a locking mechanism for the microengine threads.

2.2.6 Threads

There are eight hardware-assisted threads available in each of the Microengine. Any of these Threads has its own register set and program counters. This means that there

is no need to save any state information when another thread is scheduled. A context switch is therefore very efficient.
The main purpose of having several threads of execution per microengine is to hide memory latencies. A thread can do other work while another is waiting for an I/O operation to complete.

2.3 SRAM

The IXP 2400 has two SRAM Controllers, each of which supports up to 64 MB of medium-latency memory.
Each SRAM controller also includes a 64-element queue array. An element of this array can be used to implement a queue or a ring.
In addition to random reads and writes, the SRAM controller provides several atomic operations, such as atomic increment and decrement. These operations can be used to synchronize the access to data structures between the microengines.

2.4 DRAM

There is one DRAM controller located on the IXP which supports sizes up to one GB. The DRAM unit does neither provide atomic operations nor a queue array.

2.5 SHaC

SHaC (Scratchpad, Hash and CAP) is a multipurpose unit containing the following controls:

- Scratchpad. The Scratchpad provides 16KB of low-latency memory for general purpose use. Like SRAM, the Scratchpad supports hardware for the implementation of rings and atomic operations.

- Hash unit. This unit can be used to offload hash calculations. It generates 48, 64-, and 128-bit hashes.

- CAP (Control and Status Register Access Proxy) Unit. This unit provides the means to access all system-wide registers. Furthermore, this unit contains the controls for the inter-microengine communication. In addition, the CAP unit also implements the register reflection interface which allows one microengine to write or read the transfer registers of another microengine.

2.6 MSF

The Media and Switch Fabric (MSF) Interface is used to connect the IXP 2400 to a physical layer device or to a switch fabric. It provides the interface for packet reception and transmission.

3 Architecture of the Emulator

This chapter describes the architecture and implementation of the emulator. The first two sections describe the implementation concept and the design criteria, respectively. Section 3.3 then describes the design and architecture of the microengine emulator. The emulation of the external units is discussed in section 3.4.

3.1 Implementation concept

There are basically two approaches in implementing an emulator, a software friendly and a hardware friendly approach. The first one is oriented towards the programming language in which the emulator is implemented. In this approach the data structures are chosen as to be best implemented in the given language. This concept generally results in simpler and more efficient code. The drawback of this design is that a small change in the hardware may invalidate the chosen implementation.
The latter approach is oriented towards the hardware. The functionality of the hardware is implemented as it works in the real system although it may be solved more efficient or simpler in software. But this design has the advantage that a change in the functioning of the hardware has a similar change in software.
The implementation of the IXP 2400 emulator follows the second approach.

3.2 Design criteria

There are several design issues which must be taken into account in implementing an emulator. In this implementation, the criteria were considered in the following order of decreasing priority:

- Flexibility. The same implementation concepts should be applicable to any hardware device. For the IXP 2400 Emulator this means that only a few extensions and changes should be necessary in order to also emulate the IXP 2850 or other successors.
- Accuracy. There should be no difference in executing code on the real machine and on the emulator.
- Portability. The Emulator was implemented in standard compliant C. This means that compiler-dependent facilities such as bit-fields are not used. Further, only the Standard C Library was used.
- Efficiency. From the implementation alternatives, which fulfill the above requirements, the most efficient one has to be chosen.

3.3 Emulation of the Microengines

3.3.1 Processor status

A processor must always keep information of the current execution status. This also applies to a microengine. For example, a microengine must know which thread is the one

which is currently executing. This information is saved in the registers of the processor. The full state of the processor must also be maintained by the emulator. This is done using a so-called context struct. This struct is kept up to date during the whole execution. Any register of the real machine must be stored in this struct.
The context struct for a Microengine is shown in Listing 3.1. Some fields which are not essential for the discussion are omitted.

```c
typedef struct me_s
{
    uint32_t            gpr[2][ME_GPR_CNT];
    uint32_t            sxfer[2][ME_SXFER_CNT];
    uint32_t            dxfer[2][ME_DXFER_CNT];
    uint32_t            nnreg[ME_NNREG_CNT];

    uint32_t            lm[ME_LM_SIZE];

    uint32_t            local_csr[ME_CSR_CNT];

    unsigned char       istore[ME_ISTORE_SIZE][5];

    me_ctx_t            *ctx[ME_CTX_CNT];
    me_ctx_t            *cur_ctx;

    cmd_fifo_t          *fifo;
    me_cam_t            *cam;

    int                 code_loaded;
    int                 idle;
    int                 stalls;

    uint64_t            instr;

    uint32_t            cc;

    me_delayed_t        *delayed_list;

    struct me_cluster_s *cluster;
    struct me_s         *prev, *next;

    me_opcode_f         *opcodes;
} me_t;
```

Listing 3.1: Microengine context struct.

The meaning of each field is shortly described in the list below. Most of these fields are explained in more detail during the subsequent discussion.

- `gpr`: The 256 general purpose registers (128 on Bank A, 128 on Bank B).
- `sxfer`: The 256 SRAM Transfer Registers (128 read-only, 128 write-only).

- `dxfer`: The 256 DRAM Transfer Registers (128 read-only, 128 write-only).
- `nnreg`: The 128 next Neighbor registers.
- `lm`: Local Memory.
- `local_csr`: The local Control and Status Registers.
- `istore`: Instruction store
- `ctx`: The array containing the eight thread context structs.
- `cur_ctx` A pointer to the current executing thread.
- `fifo`: A pointer to the command FIFO.
- `cam`: A pointer to the CAM unit.
- `code_loaded`: Indicates whether a binary was loaded to this microengine or not.
- `idle`: Indicates that the microengine is in idle state.
- `stalls`: The number of stall cycles until the next instruction is executed.
- `instr`: The instruction which is currently executing.
- `cc`: The condition codes. There is only one set of Condition Codes, not a set per thread.
- `delayed_list`: This list contains operations which are performed in a later cycle.
- `cluster`: A pointer to the cluster on which this microengine is located.
- `next`, `prev`: Pointers to the next and previous microengine, respectively. These pointers are required to access the next neighbor registers of an adjacent microengine.

The context struct for a thread on the other hand must save the current state for one thread. Above all, a thread must know which instruction will get executed next. This information is saved in the program counter. The whole context struct is shown in listing 3.2.

```
typedef struct me_ctx_s
{
    int             state;

    uint32_t        pc;

    uint32_t        *gpr_map[2];
    uint32_t        *sxfer_map[2];
    uint32_t        *dxfer_map[2];
    uint32_t        *nnreg_map;

    uint32_t        ctx_csr[ME_CSR_CNT];
    struct me_s      *uengine;
} me_ctx_t;
```

Listing 3.2: Thread context struct.

The meaning of the fields are as follows:

- `state`: The current state of the thread (Inactive, Sleep, Ready, Executing).
- `gpr_map`, `sxfer_map`, `dxfer_map`, `nnreg_map`: The thread-local registers. These registers are a subset of the register file of the microengine.
- `ctx_csr`: The per-thread Control and Status Registers.
- `uengine`: A pointer to the microengine, this thread is belonging to.

3.3.2 Clocking

As already briefly described in the introduction, most of today's computer are based on the von Neumann architecture. One of the major component of such an architecture is the internal clock. This clock regulates the work of all hardware devices.
As in hardware, the most basic task of an emulator is to clock all the components. This is done in the main loop of the emulator. Each microengine cluster is clocked in this loop and a cluster in turn clocks each microengine, unless there is no code loaded.

3.3.3 Instruction Execution

Instruction execution in the PCE Emulator Framework is performed using an opcode table.
This opcode table contains function pointers for each opcode. Whenever the emulator fetches a new instruction, part of the instruction opcode is extracted and used as an index into this table. The corresponding function found at the index is then called with the microengine context struct as a parameter.
The part of the opcode which is used as an index should contain enough bits to have a different index for each instruction. The microengine emulator uses the first 8 bit of the 40-bit instruction to achieve this requirement.

3.3.4 Instructions with Latency

Any instruction accessing functional units which are external to the execution datapath have a certain delay. This delay is fully visible to a programmer.
The `local_csr_wr` instruction is an example of such a delayed operation. This instruction is used to write to the local control and status registers (CSR) of a Microengine. It always takes 3 cycles until the value in the register has actually changed.
The code sequence shown in listing 3.3 sets the index of the local memory to 1 by writing to the corresponding CSR. The local memory can then be accessed at the specified address through the use of the indexed register `*l$index`.

```
local_csr_wr[INDIRECT_LM_ADDR_0, 1]
nop
nop
nop
alu[dest, *l$index0,  +,  8]
```

Listing 3.3: Writing to a CSR has a latency of three cycles.

The indexed register can therefore be referenced only 3 cycles later. That this delay cannot simply be ignored by the emulator becomes even more obvious with the following example with maximum pipeline usage:

```
local_csr_wr[INDIRECT_LM_ADDR_0, 1]
local_csr_wr[INDIRECT_LM_ADDR_0, 2]
local_csr_wr[INDIRECT_LM_ADDR_0, 3]
local_csr_wr[INDIRECT_LM_ADDR_0, 4]
alu[dest1, *l$index0,  +,  1]
alu[dest2, *l$index0,  +,  2]
alu[dest3, *l$index0,  +,  3]
alu[dest4, *l$index0,  +,  4]
```

Listing 3.4: Maximum pipeline usage.

The first ALU instruction in listing 3.4 operates on the memory address loaded by the first `local_csr_wr` instruction, the next on the address loaded by the second `local_csr_wr` instruction and so on.
Whereas the first example would still be executed correctly on an emulator ignoring this delay, this is no longer the case with the second example.

Instructions with a delay are handled by the emulator using a dynamic linked list. Whenever an operation needs to be delayed, a new node is inserted to this list. The node stores all the necessary information such as arguments and function pointer as well as the clock cycle when the delayed operation is to be handled.
The reason for using a dynamic linked list instead of a static array is that the list needs to maintain a chronological order of all nodes. So insertion must be possible at any position since the delay is not always the same.
In each cycle, the front of the list is checked by the emulator to see if there is one or more delayed instructions which take place in the current clock cycle. If so, the corresponding node is removed and handled just before any new instruction is fetched.

The example in figure 3.1 shows the contents of the list after the last `local_csr_wr` instruction has been executed in listing 3.4. It is assumed that the list is empty at the beginning. Clock cycle n is the cycle when the first ALU instruction is executed.

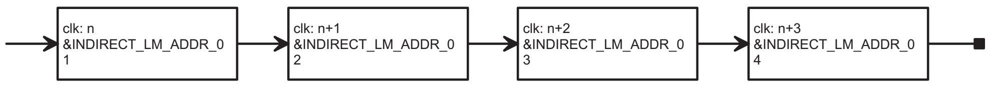


Figure 3.1: Contents of the delayed list for Listing 3.1.

3.3.5 Branch Instructions

Branch instructions are used to control the program flow. In case a branch is taken, the value of the program counter is changed accordingly and the execution continues at an other place in the code.
Any taken branch causes two or more instructions in the execution pipeline to be aborted, depending on the stage where the branch decision was made.

An unconditional branch, for example, causes two instructions to be aborted, as can be seen in figure 3.2. The unconditional branch instruction `br[10]` is decoded at stage 3. At this time, two new instructions (opX and opY) have already been fetched. Assuming the branch is taken, these two instruction must be aborted.

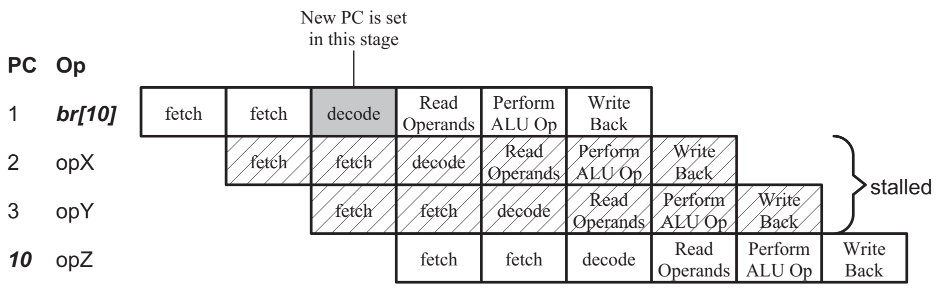


Figure 3.2: Execution Pipeline.

In order for the microengine emulator to be cycle accurate, these pipeline stalls must be taken into account. Whenever a branch instruction causes a stall of the pipeline, the `stalls` variable of the microengine context struct is set to the corresponding number of stall cycles. This variable tells the emulator not to fetch and execute any instruction for the given number of cycles.

Since each stall of the pipeline causes a cycle to be lost, a microengine includes hardware which allows a branch to be deferred by a programmer. This means that one or more instructions which immediately follow the branch instruction are executed independent of whether the branch was taken or not. Instructions which are already fetched do not need to be aborted any longer in case the branch was actually taken.

In case of a deferred branch, the update of the program counter as well as the update of the `stalls` variable must be delayed by the emulator. This is realized using the same concept as discussed in the previous section.
The following code snippet shows how the emulator handles an unconditional branch which is delayed by one cycle.

```
me_delay_assign32(c, &c->stalls, 1, 1);
me_delay_assign32(c, &c->cur_ctx->pc, pc, 2);
```

Listing 3.5: Deferred branch.

The implementation can actually be optimized since no other branch can occur in the defer shadow of a branch instruction. This optimization is discussed in the section about performance.

3.3.6 Thread Arbitration

There are eight hardware threads available in each microengine. Each of these threads is always in one of the following states:

- Inactive: A thread which is in this state will never be scheduled.
- Executing: The thread has been scheduled by the arbiter and is currently executing code. There is at most one thread in this state.
- Ready: The thread is ready to be scheduled. When the current executing thread is put to sleep state, the thread arbiter chooses the next thread from among all the threads in ready state.
- Sleep: The context is waiting for an external event to occur. It will not be scheduled by the arbiter.

The events a thread in sleep state is waiting for are stored in a control and status register (CSR) of the corresponding thread. Another CSR saves the events which have already been occurred. These two registers are used by the thread arbiter of the emulator to decide whether a thread can be scheduled or not.
As soon as any or all (depending on the mode) of the events a thread is waiting for have been occurred, the corresponding thread is put into ready state. This procedure is illustrated in figure 3.3.

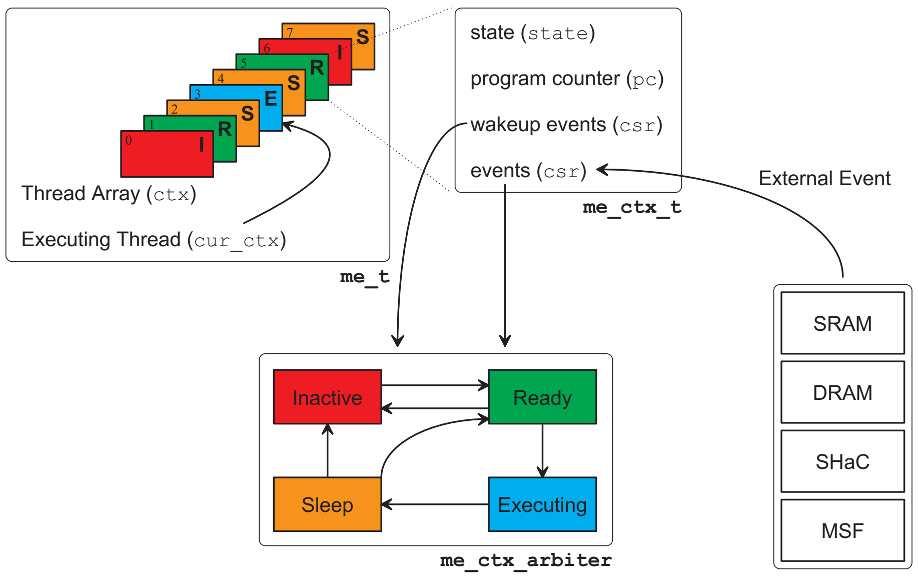


Figure 3.3: Thread Arbitration.

The execution of a context switch instruction stalls the pipeline for two cycles. The reason is the same as for branch instructions. The program counter of the new context is set only in pipeline stage 3. As for branch instructions, the stalls can be eliminated using the defer optional token.
Each time a context switch operation is executed on the emulator, the corresponding opcode function swaps out the running thread and then calls the thread arbiter:

```
me_delay_func(c, me_ctx_swap_out, 2);
me_ctx_arbiter();
```

Listing 3.6: The current executing thread is swapped out.

The call to `me_ctx_swap_out` is delayed by two cycles. This means that the `cur_ctx` variable of the microengine context struct is set to `NULL` only 2 cycles later. So one or two instruction still get executed from the old thread in case the context swap instruction was deferred by one or two cycles, respectively.

The thread arbiter checks if there is a ready thread and if so, this thread is scheduled. This operation is also delayed by two cycles, as can be seen in Listing 3.7.

```
me_delay_func(c, me_ctx_swap_in,  2);
```

Listing 3.7: The thread arbiter swaps in a new thread in case there is one in ready state.

Otherwise the arbiter sets the `idle` variable of the microengine context struct to true. This flag indicates that the emulator now has to call the arbiter in each cycle until a ready thread is found.

3.3.7 Condition Codes

The CPU maintains a set of single-bit condition code variables or registers describing the attributes of the most recent arithmetic or logical operation. These codes or flags are then used to perform conditional branches. The condition codes used by the Microengines are:

- C: Carry Flag. The most recent operation resulted in a carry out of the most significant bit.
- Z: Zero Flag. The most recent ALU operation yielded zero.
- N: Negative Flag. The most recent result was negative, i.e. Bit 31 was 1.
- V: Overflow Flag. The most recent operation caused a signed overflow.

These conditions are easy to set in hardware but more involved in software, especially the test for carry out.
A simple approach to test whether there was a carry out in an 32-bit addition would be as follows:

```
carry = ( ((uint64_t) s1 + (uint64_t) s2) & 0x100000000ULL) ? 1 : 0;
```

Listing 3.8: Test for carry out using 64 bit addition.

But this requires a 64 bit addition and is not applicable any longer if the operands were of 64-bit.
A more flexible and even slightly more efficient way is based on the following properties of two-complement Integer arithmetic:

- If we add two positive numbers, there will never be a carry out of the sign.
- If we add two negative numbers, there will always be a carry out of the sign.
- If we add two numbers of unlike sign, there is a carry out of the sign if and only if there is a carry in to the sign, i.e. the two carries will always be the same.

These properties lead to the following test for carry out:

```
if ((((long) s1 < 0) && ((long) s2 < 0))
    || (((s1 ^ s2) & BIT31) && !(d & BIT31)))
{
    // carry out
}
```

Listing 3.9: Test for carry out using Two-Complement Integer Arithmetic properties.

Where `d` is the 32-bit result of the addition of `s1` and `s2`.

3.3.8 Execute Function

The above discussions lead to the following execute function which is called in each cycle:

```
void me_execute (me_t *c)
{
    if (c->idle) {
        me_ctx_arbiter(c);
    }

    if (c->stalls > 0) {
        c->stallcnt++;
        c->stalls--;
    }
    else if (c->cur_ctx != NULL) {
        me_ifetch (c, c->cur_ctx->pc, &c->instr);
        c->opcodes[(c->instr >> 32) & 0xff] (c);
    }

    me_exec_delay_op(c);

    if (cmd_fifo_full(c->fifo) && (c->stalls < 1)) {
```

```
19          c->stalls = 1;
20      }
21  }
```

Listing 3.10: Execute function of a microengine.

Some non-essential lines have been omitted. The code is shortly explained in the list below.

- lines 3-5: It is checked if the microengine is in idle state. If so, the thread arbiter is called to see if there is a thread which can be scheduled.
- lines 7- 10: The last branch or context swap operation caused a stall of the pipeline. Hence no operation is performed in this cycle.
- lines 11 - 14: In case there is a context in executing state, the next instruction is fetched from the instruction store. The index to the opcode table is extracted and the corresponding opcode function is called.
- lines 16: Delayed operations are executed.
- lines 18-19: The command FIFO is tested for fullness. In case the FIFO is actually full, the processor is stalled until there is room for a further command. This can only be the case if the last operation was an I/O instruction. The command FIFO is explained in more detail in the next section.

3.3.9 Differences

The following list summarizes the differences between the real processor and the micro-engine emulator:

- The MSF specific instructions are not yet implemented. This is due to the fact that the Media Switch Fabric is not yet emulated.
- The integration of the microengines with the XScale core processor is not yet complete. Especially the microengines are not yet initialized by the XScale core processor. Instead the instruction store is directly initialized by the loader.
- No interrupt on the XScale is yet caused by the Microengines. This difference also arises from the incomplete integration of the Microengines with the Core Processor.

3.4 Emulation of the External Units

3.4.1 Command Bus

The Microengines communicate with the external devices through a command bus. This command bus notifies the external units that a microengine is requesting service. The architecture of this bus is shown in figure 3.4.

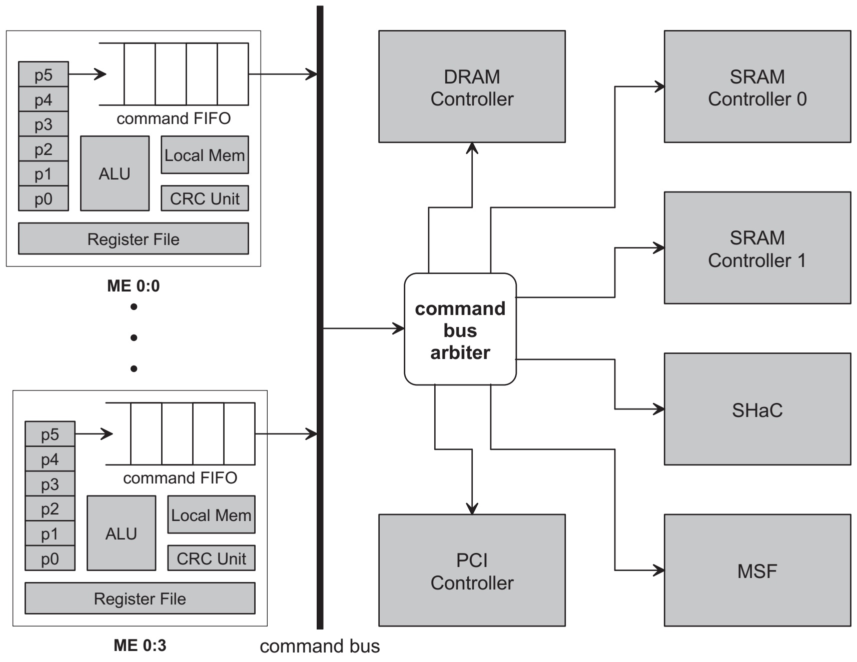


Figure 3.4: Command Bus Architecture.

Whenever a thread of a microengine executes a memory or I/O instruction, a command is generated. A command is represented by the following structure:

```
typedef struct mem_cmd_s
{
    int               target_id;

    int               func;

    uint32_t          addr;

    uint32_t          *xfer_rd;
    uint32_t          *xfer_wr;

    uint32_t          immed;
    int               refcnt;
    int               wmask;

    event_handler_f event_handler;
    void*           event_dst;
    int             sig;
} mem_cmd_t;
```

Listing 3.11: Command struct.

Each field is briefly described in the following list:

- `target_id`: The ID of the unit this command is sent to.
- `func`: The function to be executed on the target device. Each device has its own set of functions such as read, write and so on.
- `addr`: The target address.
- `xfer_rd`: The address of the destination transfer register, where the results of the operation are to be stored.
- `xfer_wr`: The address of the source transfer register, where the operands for an operation are read.
- `immed`: This field is used by so called fast write commands. Such commands don't buffer the value in a transfer register. Instead the operand is delivered directly with the command. This saves the cycles to pull the operand from a transfer register. The reference count for such a command must always be one.
- `refcnt`: The Reference Count. This is the number of operations to be performed in the same request. The operands must be placed in continuous transfer registers in case this number is greater than one.
- `wmask`: Some targets support a write mask indicating the bytes which should be modified. The other bytes are not affected by a write command.
- `event_handler`: The function pointer of the method which gets called as soon as the request is complete. This is actually always the `me_assert_signal` function.
- `event_dst`: A pointer to the context which has to be informed when the operation is complete. The `event_handler` is called with this pointer as first argument.
- `sig`: The signal which must be asserted on completion. This is the second argument to the `event_handler` function.

After the command has been created and initialized, it is put onto the command FIFO of the corresponding microengine. Each Microengine has an own command FIFO which has room for 4 commands. In case the FIFO is full the microengine stalls until there is room for a further command.
This command FIFO is registered on the bus when the microengine Emulator is initialized:

```
void me_init(me_t *c)
{
   ...
```

```
    cmd_bus_register_fifo(c->cluster->bus, c->fifo);
    ...
}
```

Listing 3.12: Registration of the command FIFO.

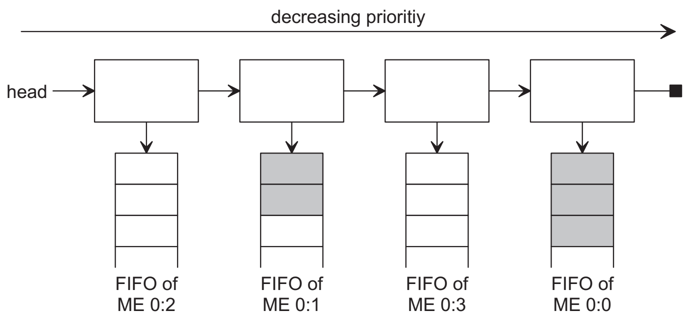
3.4.2 Command Bus Arbiter

The command bus arbiter selects one command from a command FIFO in each bus cycle and moves it on the command bus. As can be seen in figure 3.4, the four microengines of a cluster share the same command bus. Hence it may happen that more than one command FIFO contains a command to be sent to the external devices at the same time. It is the task of the command bus arbiter to decide which FIFO gets served next, so that only one command is sent in each bus cycle.

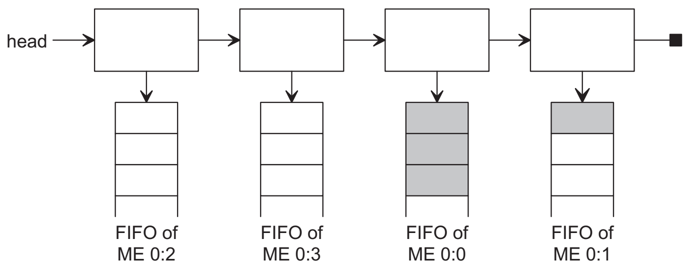
Unfortunately the exact functioning of the bus arbiter used in the IXP was not precisely explained in the hardware reference manual.
But there exist generally two arbitration schemes to achieve this exclusiveness: fixed-priority and rotating-priority scheduling. In the case of fixed priorities, each device connected to the bus is statically assigned a priority. Whenever more than one device wants access to the bus simultaneously, the one with the higher priority gets served first. Obviously this scheme is not fair but can be used in case the requests of the devices are not equally important.

Scheduling using rotating priorities on the other hand guarantees fairness. With this scheme, each device is assigned a priority at the beginning. The device with the highest priority is granted the bus first. After this bus access has finished, this device will get the lowest priority and the other priorities will change accordingly, i.e. the one with the second highest priority now has the highest priority and so on.
Using fixed priorities certainly makes no sense as all microengines connected to the bus should be treated in the same manner. Such a fair treatment can only be achieved using rotating priorities or another round-robin based algorithm. The arbitration in the IXP emulator is implemented using rotating priorities.

All command FIFOs which were registered on the bus are placed in a linear linked list. This list is sorted in descending order of priority. That is, the head of the queue points to the FIFO which has currently the highest priority.
In each bus cycle this list is traversed and the first FIFO having a command is served, as it is the one with the highest priority from all non-empty queues.
This FIFO is then put to the tail, which means that it now has the lowest priority. This procedure is depicted in figure 3.5. In clock cycle n, the FIFO of Microengine 1 has the highest priority from all non-empty FIFOs and hence gets scheduled first. After the request is complete, this FIFO is placed to the tail of the queue so that in cycle $n + 1$ the FIFO of microengine 0 is scheduled.



(a) Clock n



(b) Clock $n + 1$

Figure 3.5: Arbitration of the command FIFOs using rotating priorities.

3.4.3 Command Handler

Any external device must implement a command handler function which has to be compatible with the following function pointer:

```c
typedef void (*cmd_handler_f) (void *ext, struct mem_cmd_s *cmd);
```

Listing 3.13: Function pointer to a command handler function.

The first parameter is a pointer to the device struct of the target unit. The second parameter is the pointer to the command to be executed.
This command handler function gets called whenever a command is sent to the corresponding device. Each external device must register its own handler on the bus when the Emulator is initialized. The following lines show how this is done for one SRAM Controller:

```c
void sarm_setup_sram (simarm_t *sim)
{
    ...
    cmd_bus_register_dev(sim->bus, DEV_SRAM0, sim->sram[0],
(cmd_handler_f) sram_cmd_handler);
    ...
}
```

Listing 3.14: Registration of the command handler.

The second parameter to the registration function is a constant which uniquely identifies the device. The third parameter is a pointer to the device struct.
Each time the command bus arbiter schedules a command it looks up the pointer to the device struct in the registration table using the `target_field` of the command struct. The command handler is then called with this function pointer as first parameter.
Although the implementation of the command handler is device dependent, the typical procedure is to enqueue the command to the command queue of the device.

In hardware, once the command reaches the head of the queue the command goes through the whole controller pipeline during which the necessary steps to handle the request are performed. The steps involved depend on the command but usually include pulling the operands from the pull bus, setting the address and so on.

Since most part of this scheduling is invisible to a programmer, the IXP 2400 Emulator performs all these steps in the final clock. In order to achieve this, the emulator puts the command on the queue together with the clock cycle, when the operation is to be completed.
In each clock cycle, the emulator checks if the command at the head of the command queue is ready to be executed. If this is the case, the command is dequeued from the queue and the command execution function of the device is called with the command struct as parameter.

3.4.4 Differences

Most of the differences arise from the fact, that not every part of the bus architecture is yet implemented. The missing parts are indicated in the list below.

- The Pull and Push data buses are not yet emulated. Once a command is scheduled, the current emulator directly reads or stores the value at the register address which is specified in the command.
- Only one command bus is instantiated for all microengines as opposed to the real architecture which has two instantiations, one for each cluster.

The differences which result from the above omissions merely have an effect on the delay of commands.
The omission of the Pull and Push buses, for example, has an influence on the time when the signals are asserted. In the real architecture, the signal for a write request is asserted as soon as the value has been pulled from the write transfer register, meaning that the register can now be reused. But it takes further cycles until the value gets actually written.
On the other hand, the signal for a read request is asserted when the operation is actually complete, i.e. the memory location has been read and the corresponding value has been pushed into the push bus.

On the emulator the signals for both, read and write commands, are asserted only when the request is complete.

The omission of the second instantiation of the command bus on the other hand, has a consequence in case two microengines on different clusters are accessing external units in the same cycle. Instead of processing the two commands at the same time, the emulator sends the commands in two different cycles.
The reason why only one command bus is yet instantiated is the fact that most units can only accept one command per cycle. By using two command buses, it may happen that two microengines on different clusters access the same unit in the same clock cycle. The details of the bus architecture needed to handle this case are not yet emulated in the current implementations status.

Of course, these timing issues have no influence at all on the correct execution of programs, as the only way to determine the completion of an I/O operation is the use of signals. The queue management involved in performing such an operation is generally completely unpredictable by a programmer. This point will become more clear in section 4.2.

4 Evaluation and Testing

Section 4.1 discusses the procedures which were used to the test the implementation of the emulator. Section 4.2 first gives some performance examples and then discusses the interaction of all components with the help of the cycle statistics of an assembly program.

4.1 Test methodology

Program testing generally serves the purpose of finding errors in a program. Except for trivial programs, testing can never prove the correctness of a program.
Testing a program for correctness would mean to supply every possible input sequence and then comparing the result with the expected one. In the case of an emulator this means to generate any possible instruction combination and compare the result with the one of the real machine. In other words, any conceivable assembly program would have to be run and compared. Such a task is impossible in practice.

Nevertheless, proper use of testing reveals many of the errors by choosing a representative subset of all possible inputs.

In order to test the proper functioning of the Microengine Emulator, the debug interface of the IXP emulator has been extended to support the following features:

- Microcode Disassembler
- Inspection of any local and system-wide register
- Dumping of any memory
- Current status of any context
- Running code until a specified microengine is halted

The test procedure involved the following steps:

- Each opcode function has been tested just after implementation using some short assembly programs.
  In this stage, it has only be tested whether the implemented function performed according to the description in the manuals or not. Only the general purpose registers were implemented at this time, so the set of operands was rather restricted.
- After the implementation of the remaining registers and functional blocks of the microengine, each opcode was again tested. This time the result was compared with the one of the Intel simulator running the same program.
  At this stage each non-memory opcode was already implemented, so more complicated programs, such as prime number search, could already be run and tested.
- The memory opcodes were tested after the completion of the external units.

- As a final step, assembly programs having both, ALU and memory instructions were executed. Again, the result after each major step was compared with the Intel simulator.

As the Media Switch Fabric is not yet fully implemented, packet processing programs could not yet been executed and tested. Nevertheless, many of the steps involved in such processing, as enqueuing and dequeuing of buffers, have been tested with other types of input.

4.2 Performance

The purpose of this section is to discuss the performance of the emulator with the help of some of the assembly programs which were used for testing. Furthermore, several execution statistics are explained to make some of the above concepts more clear.
The first example is a brute force prime number search. The assembly program is shown in listing A.1. The implementation is actually rather naive but it works without the need of any memory reference. Since the microengine neither have a division nor a modulo instruction, these operations are implemented in lines 37-43 using an iterative algorithm.

Table 4.1 gives a summary of the cycles needed for the program to execute. The total number of clock cycles is subdivided into the following components:

- Operations: The number of operations which were actually performed during execution. This number is the result of the subtraction of the stall and idle count from the total number of cycles.
- Stalls: The number of instruction which were aborted due to pipeline stalls.
- Idle cycles: The number of clock cycle during which the microengine was idle, meaning no thread was in executing state.

Clocks	12'762'792
Operations	12'686'914
Stalls	75'875
Idle cycles	3
CPI	1.0060

Table 4.1: Cycle Summary.

The reason why the program causes almost no pipeline stalls is the use of a deferred branch instruction in the innermost loop (lines 37-43). The unconditional branch in line 40 is deferred by two cycles and hence never causes a stall of the pipeline. If the branch would not have been deferred, it would cause a penalty of two cycles in each iteration.
The three idle cycles are due to the fact, that there is no thread scheduled at the very beginning. When the microengine is clocked for the first time after reset, the thread arbiter schedules the first thread, which begins execution only two cycles later.
The execution time and the conversion to megahertz for running this program under

different configurations are listed in table 4.2. The times were measured on an Intel Pentium P4 3GHz processor.

	Only MEs	All units
malloc	2.77 s (4.61 MHz)	5.52 s (2.31 MHz)
palloc	2.10 s (6.08 MHz)	4.97 s (2.57 MHz)
opt. branches	1.76 s (7.25 MHz)	4.68 s (2.73 MHz)

Table 4.2: Execution times for running the program from listing A.1 on one microengine.

For the first part of this example, the image has been loaded to only the first microengine, while the others were inactive.
The second column contains the times in case only the microengines are clocked whereas the last column shows the corresponding values if all units, including the XScale core processor, are clocked.

Each row of a table indicates the execution time for a different configuration.
Since the branch in the innermost loop is deferred and hence a delayed operation, a new node must be allocated in each iteration. The first row shows the time, when the corresponding memory is allocated using malloc. This is rather expensive, as the times in this row show. A static list is not an alternative since the nodes need be inserted in sorted order.
But as each delayed operation gets executed after a finite time, there is more or less always the same amount of memory allocated. That's why the use of a memory pool is quite appropriate. Instead of always allocating a small amount of memory for each request, a big buffer is requested from the operating system at the beginning. The memory from this buffer is then no longer managed by the operating system but from the application program, using the calls to `palloc` and `pfree`, respectively. In case all memory from the buffer is in use, a further buffer is allocated from the operating system.
The improvement in terms of execution time using a memory pool can be seen in the second row.
A memory pool has the big advantage that apart from the calls to `malloc` and `free` nothing has to be changed. In other words, all delayed instructions can still be implemented by using the same linked list.

The last row finally shows the execution time for the case that branches are implemented in a completely different manner, that is, without the need of any additional memory. Although this solution doesn't require many changes neither, it has the disadvantage that it can only be applied to branches and context switch operations. The reason for this is that no other branch or context switch instruction can occur in the defer shadow of such an instruction.
This optimization may actually be legitimate as these two operations occur much more often than the others. Nevertheless the final release uses the dynamic list approach for reasons of flexibility.

The execution times for running the same image on all microengines are shown in table 4.3. Now the overhead imposed by the external unit is much less, as the microengines

	Only MEs	All units
malloc	17.18 s (0.74 MHz)	20.01 s (0.64 MHz)
palloc	13.38 s (0.95 MHz)	16.31 s (0.78 MHz)
opt. branches	10.53 s (1.21 MHz)	14.04 s (0.91 MHz)

Table 4.3: Execution times for simultaneously running the assembly program from listing A.1 on all eight microengines.

now need most of the resources for computation. The same reason also causes the optimizations to be of more importance. Figure 4.1 illustrate these execution times.

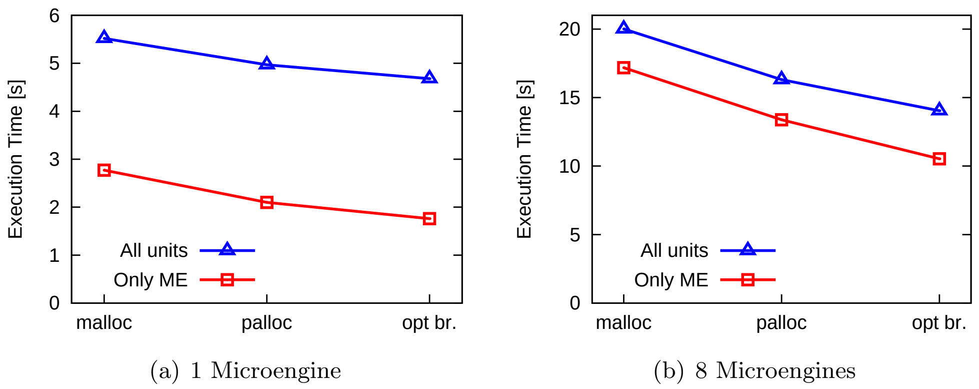


Figure 4.1: Illustration of the execution times for running the program on one and eight microengines, respectively.

The next example shows that the execution time of an instruction on the emulator depends on the particular type of the instruction.
Listing A.2 and listing A.3 demonstrate this with the help of the CRC instruction. The first program performs a CRC instruction in each iteration of a loop. The second program executes this same loop, but this time the CRC instruction was replaced by a `nop` instruction. The result is result is shown in table 4.4 and 4.5, respectively.

Clocks	33'554'447
Operations	25'165'836
Stalls	8'388'608
Idle cycles	3
CPI	1.3333

Table 4.4: Cycle Summary.

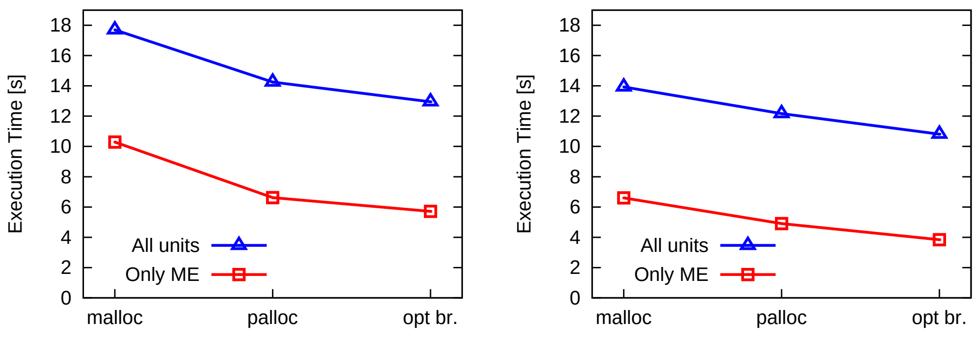
The code was each time loaded to only one microengine.
The reason why the influence of using a memory pool is more significant than the use of optimized branches in the case of the CRC example is the fact that the CRC instruction

	Only MEs	All units
malloc	10.29 s (3.26 MHz)	17.70 s (1.90 MHz)
palloc	6.62 s (5.07 MHz)	14.25 s (2.35 MHz)
opt. branches	5.71 s (5.88 MHz)	12.95 s (2.59 MHz)

Table 4.5: Execution times for running a loop performing CRC calculations.

	Only MEs	All units
malloc	6.60 s (5.08 MHz)	13.93 s (2.41 MHz)
palloc	4.91 s (6.83 MHz)	12.17 s (2.76 MHz)
opt. branches	3.84 s (8.74 MHz)	10.81 s (3.10 MHz)

Table 4.6: Execution times for running a loop of nop instructions.



(a) CRC calculation (b) Nop loop

Figure 4.2: Measured execution time.

is also a delayed instruction.

The last two examples do not deal with performance issues but rather intend to make the concepts discussed in previous sections more clear.
A.4 shows a program which uses both, ALU and memory instructions. The `init` macro initializes an array with random numbers. This array is then sorted in the `sort` macro using selection sort.
Each time the program executes a memory instruction, the current thread is put into sleep state and waits for the operation to complete. In case there is another thread in ready state, this thread is scheduled and execution continues in the new thread.

Table 4.7 shows the clock summary for eight different configurations.
The first configuration executes the program in only one thread. All other threads are killed at the very beginning. The second one uses two threads, each one executing the exactly same program but operating at a different base address. The random numbers are generated in such a way, that each thread initializes the array with the same sequence,

Number of threads	Total cycles	Operations	Stall cycles	Idle cycles	CPI
1	16’322’047	1’815’561	1’076’758	13’429’728	8.9901
2	16’322’085	3’631’073	2’153’481	10’537’531	4.4951
3	16’322’121	5’446’584	3’230’204	7’645’333	2.9968
4	16’322’155	7’262’094	4’306’927	4’753’134	2.2476
5	16’324’567	9’077’603	5’383’650	1’863’314	1.7983
6	17’392’641	10’893’111	6’460’373	39’157	1.5967
7	20’264’917	12’708’618	7’537’096	19’203	1.5946
8	23’152’946	14’524’124	8’613’819	15’003	1.5941

Table 4.7: Cycle summary for running program A.4 in a different number of threads.

resulting in exactly the same number of swap operations.
The third configuration uses three threads and so on. The image is loaded to only one microengine for each configuration.
The percentage of idle cycles and the CPI is illustrated in Figure 4.3.

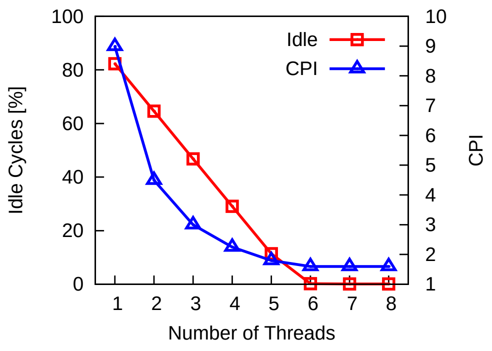


Figure 4.3: Percentage of idle cycles and CPI.

Since the first configuration has only one thread, there is no other thread which can be scheduled after this thread is swapped out. Therefore the microengine is idle until the memory operation is complete and this same thread can continue again.
With each further thread, the number of idle cycles is reduced which results in a better CPI. In other words, several arrays can be sorted with approximately the same number of clock cycles.

On the first sight, the number of stall cycles might seem to be rather big. But there are 6 pipeline stalls and 10 instructions executed on the average in each iteration of the innermost loop (lines 57-71):

- `ctx_arb`: 2 stalls
- `bge`: 3 stalls (this branch is taken most of the time)

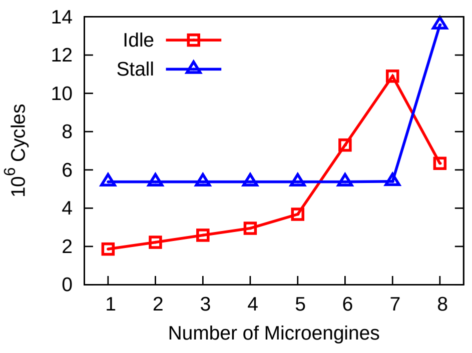
- `br`: $2 - 1 = 1$ stall (deferred)

Of course, the branch in line 66 could be changed in such a way that the branch is normally not taken. Furthermore, any delay slot could be filled with useful instructions, except for I/O and memory instructions, which must not appear in a defer shadow. But such optimizations are not relevant here.

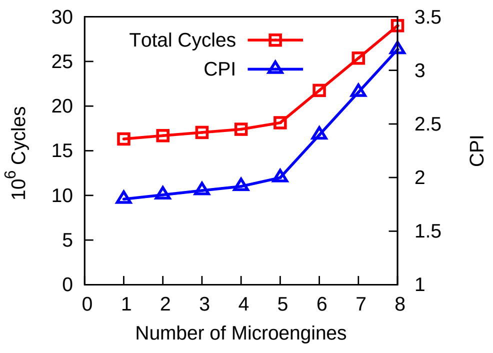
Programs which do not contain any memory or I/O operation, as it was the case for the prime number search, need always the same number of clock cycles to execute, independent on whether the image runs on one or several microengines. This is no longer the case for programs with such operations, as any external unit is shared by any microengine.
This behavior is illustrated in the final example. Now, the version of the above program using five threads is run on several microengines at the same time. Each Microengine operates on a different offset. The results are shown in table 4.8 and illustrated in figure 4.4.

Number of MEs	Total cycles	Operations	Stall cycles	Idle cycles	CPI
1	16’324’567	9’077’603	5’383’650	1’863’314	1.7983
2	16’685’177	9’077’603	5’383’650	2’223’924	1.8381
3	17’048’165	9’077’603	5’383’650	2’586’912	1.8780
4	17’411’153	9’077’603	5’383’650	2’949’900	1.9180
5	18’142’975	9’077’603	5’383’650	3’681’722	1.9987
6	21’760’965	9’077’603	5’383’650	7’299’712	2.3972
7	25’378’963	9’077’603	5’401’650	10’899’710	2.7958
8	28’996’961	9’077’603	13’577’045	6’342’313	3.1943

Table 4.8: Cycle summary for running program A.4 on a different number of microengines.



(a) Stall and idle cycles



(b) Total Cycles and CPI

Figure 4.4: Illustration of the cycle summary of Table 4.8.

Obviously the number of idle cycle increases again with each further microengine.

The reason for this is that each microengine accesses the same SRAM controller. This controller can receive at most one command per cycle. This means that in case there are two or more microengines performing a memory request in the same bus cycle, only one of them gets served. The other commands remain in the command FIFO of the corresponding microengine until they are arbitrated.
The delay of such a command is increased by the number of cycles it remains in this FIFO. That's why there are now fewer threads in ready state, so the microengine is more often idle.
The number of commands in the FIFOs increases with each further microengine until the FIFOs are finally full. This is the case for the first time when the program is executed on 7 microengines. This can be seen by the increased number of stall cycles. In case the FIFO is full, a microengine wanting to send a memory command has to wait until there is room for a further command.
The impact of the fullness of a FIFO gets fully apparent only with the eight microengine. The number of stall cycles has more than doubled.
On the other hand, the number of idle cycles has diminished. The reason for this decrease is the fact that a microengine now has to wait longer until the command can be placed into the FIFO than the command remains in the FIFO afterward.
This example makes clear that the programmer can generally make no assumption about the time when the request is completed.

5 Summary and Conclusion

5.1 Achievements

The emulator for the Microengine is complete, except for the instructions specifically used for the Media Switch Fabric. Apart from this, there exists no difference between executing code on the Microengine and on the real machine.

With the exception of the Media and Switch Fabric, all external units have been fully implemented. Each command of any of these unit is available to the microengine as well as to the XScale core processor.
Still, there are differences in the delay of memory and I/O commands, since not every detail of the bus architecture is yet emulated.
Apart from being used as a very useful debugging tool, the IXP 2400 emulator provides the means for analyzing and optimizing assembly programs with the help of the cycle statistics.

5.2 Outlook

Clearly, the most important unit which is not yet implemented, is the Media and Switch Fabric. Hence, this unit should be implemented next.
Once this unit is functioning, the integration of the Microengines and the XScale core processor must be carried out. Especially, the initialization of the Microengines must be performed by the XScale. The external units can already be accessed by the XScale, so the missing part of the integration only concerns the XScale and the Microengines.
After both, the implementation of the MSF and the integration, the IXP 2400 Emulator is actually fully functioning.
As already mentioned in the last section, there are still differences in the timings of command execution in the external units. In order to achieve full compatibility, the whole bus architecture had to be imitated. In particular, the Pull and Push Bus need be implemented.

A Assembly Programs

```
////////////////////////////////////////////////////////
#macro is_prime(num, prime)
////////////////////////////////////////////////////////
.begin

.reg divisor rem q prod tmp

immed[divisor, 2]
immed[prime, 1]

loop#:
alu[tmp, --, B, divisor]
multiply(tmp, divisor, prod)
alu[--, prod, -, num]
bgt[exit#]

divide(num, divisor, rem, q)

alu[--, --, B, rem]
bne[loop#], defer[1]
alu[divisor, divisor, +, 1]

immed[prime, 0]

exit#:

.end
#endm
////////////////////////////////////////////////////////
#macro divide(x, y, r, q)
////////////////////////////////////////////////////////
.begin

immed[q, 0]
alu[r, --, B, x]

loop#:
alu[--, y, -, r]
bgt[exit#]
br[loop#], defer[2]
alu[r, r, -, y]
alu[q, q, +, 1]
exit#:

.end

#endm
////////////////////////////////////////////////////////

////////////////////////////////////////////////////////
#macro multiply(x, y, p)
////////////////////////////////////////////////////////
.begin

mul_step[x, y], 16x16_start
mul_step[x, y], 16x16_step1
mul_step[x, y], 16x16_step2
mul_step[p, --], 16x16_last

.end
```

```

#endm
////////////////////////////////////////////////////////

////////////////////////////////////////////////////////
//main program
////////////////////////////////////////////////////////
.begin

.reg num prime cnt max

immed[cnt, 0]
alu_shf[max, --, B, 1, <<11]
immed[num, 2]

loop#:
is_prime(num, prime)
alu[--, --, B, prime]
beq[next#]
alu[cnt, cnt, +, 1]

next#:
alu[--, num, -, max]
ble[loop#], defer[1]
alu[num, num, + , 1]

exit#:
halt

.end
```

Listing A.1: Naive prime number search.

```
.reg reg0 reg1

local_csr_wr[crc_remainder, 0]
alu_shf[reg0, --, B, 1, <<23]
nop
loop#:
bne[loop#], defer[2]
crc_be[crc_32, reg1, reg0]
alu[reg0, reg0, -, 1]

halt
```

Listing A.2: CRC calculation.

```
.reg reg0 reg1

local_csr_wr[crc_remainder, 0]
alu_shf[reg0, --, B, 1, <<23]
nop
loop#:
bne[loop#], defer[2]
nop
alu[reg0, reg0, -, 1]

halt
```

Listing A.3: Nop instruction.

```
////////////////////////////////////////////////////////
#macro init(size, base)
////////////////////////////////////////////////////////
.begin

.reg val, addr, $xfer
.sig sig1

local_csr_wr[PSEUDO_RANDOM_NUMBER, 1]
immed[$xfer, 0]
alu_shf[addr, --, B, size, <<2]

loop#:
alu[addr, addr, -, 4]
blt[exit#]

local_csr_rd[PSEUDO_RANDOM_NUMBER]
immed[val, 0]
alu[$xfer, --, B, val]

local_csr_rd[PSEUDO_RANDOM_NUMBER]
immed[val, 0]

sram[write, $xfer, base, addr, 1], ctx_swap[sig1]

// make sure each thread operates on the same sequence
local_csr_wr[PSEUDO_RANDOM_NUMBER, val]
br[loop#]

exit#:

.end
#endm
////////////////////////////////////////////////////////


////////////////////////////////////////////////////////
#macro sort(size, base)
////////////////////////////////////////////////////////
.begin

.reg i, j, min, last, $t1, $t2, t
.sig sig1 sig2

init(size, base)

immed[i, 0]
alu_shf[last, --, B, size, <<2]

loop_i#:
alu[--, i, -, last]
bge[exit_i#]

alu[min, --, B, i]
alu[j, i, +, 4]

loop_j#:
alu[--, j, -, last]
bge[exit_j#]

sram[read, $t1, j, base, 1], sig_done[sig1]
sram[read, $t2, min, base, 1], sig_done[sig2]
ctx_arb[sig1, sig2]
```

```

alu[t, --, B, $t1]
alu[--, t, -, $t2]
bge[next#]
alu[min, --, B, j]

next#:
br[loop_j#], defer[1]
alu[j, j, +, 4]

exit_j#:

sram[read, $t1, min, base, 1], sig_done[sig1]
sram[read, $t2, i, base, 1], sig_done[sig2]
ctx_arb[sig1, sig2]
alu[$t1, --, B, $t1]
alu[$t2, --, B, $t2]
sram[write, $t1, i, base, 1], sig_done[sig1]
sram[write, $t2, min, base, 1], sig_done[sig2]
ctx_arb[sig1, sig2]
br[loop_i#], defer[1]
alu[i, i, + , 4]

exit_i#:

.end
#endm
//////////////////////////////////////////////////////

//////////////////////////////////////////////////////
//main program
//////////////////////////////////////////////////////
.begin

.reg size, base, @running

immed[size, 600]
immed[base, 0x00, <<16]

br=ctx[0, sort0#]
br=ctx[1, sort1#]

...

br[exit#]

sort0#:
immed[@running, 8]
sort(size, base)
br[exit#]

sort1#:
alu_shf[base, base, or, 0x1, <<16]
sort(size, base)
br[exit#]

...

exit#:
alu[@running, @running, - , 1]
beq[halt#]
ctx_arb[kill]

halt#:
```

```
halt

.end
```

Listing A.4: Selection sort.

B Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Sommersemester 2006

Semesterarbeit

für

Christian Kummer

Betreuer:	Lukas Ruf
Stellvertreter:	Daniela Brauckhoff

Ausgabe:	03.04.2006
Abgabe:	07.07.2006

Emulierung des Intel IXP 2xxx Netzwerkprozessors

1 Einführung

Für Hochgeschwindigkeitsrouter werden heutzutage vermehrt sogenannte Netzwerkprozessoren eingesetzt (NP). Gewöhnlicherweise bestehen solche NPs aus einem Kontrollprozessor, spezialisierten Paket- und Co-Prozessoren, die zusammen auf einem einzigen Chip implementiert sind. Einer ihrer Vertreter ist der Intel IXP 2400 (IXP) [6], welcher mit einem Intel XScale Kontrollprozessor und 6 der zuvor erwähnten Paketprozessoren (sog. microengines) aufgebaut ist.
NPs werden in der Regel auf Netzwerkinterface Karten eingebettet. Im Falle vom IXP wurde ein Simulator entwickelt, welcher gewisse Komponenten nachbildet, jedoch nicht das gesamte System emuliert. Die Entwicklung von Code und besonders dessen Debugging gestaltet sich somit besonders schwierig, da ausser professionellem JTAG-basierten [5] Debugging keine Hilfsmittel zur Verfügung stehen.
Am Institut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich (ETHZ) wurde im Rahmen der Diplomarbeit von Hanspeter Hug [3] ein binär kompatible Emulierung des IBM PowerNP 4GS3 [4] entwickelt. Auf den dabei gewonnenen Konzepten aufbauend wurde die Basis des Intel IXP 2400 implementiert.
Diese Basisimplementation ist in der Lage, Linux auf dem Kontrollprozessor auszuführen. Jedoch fehlt ihr weitestgehend die Emulation der Paketprozessoren, insbesondere die Inter-Prozessor und die Netzwerkkommunikation.

2 Aufgaben: Emulierung des Intel IXP 2xxx Netzwerkprozessors

Im Rahmen dieser Semesterarbeit soll der Emulator des IXPs vervollständigt werden. Ihr primäres Ziel ist die Implementierung der fehlenden Paketprozessoren und die Vervollständigung der Inter-Prozessor Kommunikation.
Die Emulation soll die Möglichkeit bieten, die (Weiter-)Entwicklung von Betriebssystemen auf geeignete Weise zu unterstützen. Wenn möglich, soll ein Interface zum weitverbreiteten gdb [1] der GNU Tools entwickelt werden.

3 Vorgehen

- Richten Sie sich eine Entwicklungsumgebung (GNU Tools) unter Linux ein.
- Machen Sie sich vertraut mit den Unterlagen (Dokumentation und Source Code) zum bestehenden Emulator.
- Erstellen Sie einen Zeitplan, in welchem Sie die von Ihnen zu erreichenden Meilensteine Ihrer Arbeit identifizieren.
- Entwickeln Sie eine Architektur für die zu emulierenden Paketprozessoren. Beachten Sie die Multiprozessoreigenschaften eines Netzwerkprozessors sowie dessen Netzwerkinterfaces im Speziellen.
- Implementieren Sie Ihre Architektur.
- Verifizieren, evaluieren und demonstrieren Sie das Erreichte durch eine Beispielapplikation.
- Dokumentieren Sie die Resultate ausführlich.

optional Entwickeln Sie ein Interface zum GDB, welches sich in den Emulator einfügt.

optional Implementieren und testen Sie dieses Interface zum GDB.

optional Dokumentieren Sie dieses Interface.

Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte und untersuchten Varianten vor dem definitiven Variantenentscheid ausführlich schriftlich festzuhalten.

4 Organisatorische Hinweise

- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Mit dem Betreuer sind regelmässige, zumindest wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen sollen die Studenten mündlich über den Fortgang der Arbeit und die Einhaltung des Zeitplanes berichten und anstehende Probleme diskutieren.
- Am Ende des ersten Monates muss eine Vorabversion des Inhaltsverzeichnis zur Dokumentation dem Betreuer abgegeben und mit diesem besprochen werden.
- Nach der Hälfte der Arbeitsdauer soll ein kurzer mündlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt. Dieser Zwischenbericht besteht aus einer viertelstündigen, mündlichen Darlegung der bisherigen Schritte und des weiteren Vorgehens gegenüber Professor Plattner.
- Am Schluss der Arbeit muss eine Präsentation von **20 Minuten** im Fachgruppen- oder Institutsrahmen gegeben werden. Anschliessend an die Schlusspräsentation soll die Arbeit Interessierten praktisch vorgeführt werden.
- Die Arbeit muss regelmässig auf dem PromethOS subversion Server gesichert werden.
- Ein einheitlicher Coding Style muss eingehalten werden.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Die Dokumentation ist mit dem Satzsystem LaTeX zu erstellen.
- Es ist ein mit Bindespiralen (am TIK vorhanden) gebundener Schlussbericht über die geleisteten Arbeit abzuliefern (4 Exemplare). Dieser Bericht besteht aus einer Zusammenfassung , einer Einleitung, einer Analyse von verwandten und verwendeten Arbeiten, sowie einer vollständigen Beschreibung der Konfiguration von den eingesetzten Programmen. Der Bericht ist in Deutsch oder Englisch zu halten. Die Zusammenfassung muss in Deutsch und Englisch verfasst werden.

- Die Arbeit muss auf CDROM archiviert abgegeben werden. Stellen Sie sicher, dass alle Programme sowie die Dokumentation sowohl in der lauffähigen, resp. druckbaren Version als auch im Quellformat vorhanden, lesbar und verwendbar sind.
  Mit Hilfe der abgegebenen Dokumentation muss der entwickelte Code zu einem ausführbaren Programm erneut übersetzt und eingesetzt werden können.
- Diese Arbeit steht unter der GNU General Public License (GNU GPL) [2].
- Diese Arbeit wird als Diplomarbeit an der ETH Zürich durchgeführt. Es gelten die Bestimmungen hinsichtlich Kopier- und Verwertungsrechte der ETH Zürich.

Literatur

[1] GDB, the GNU symbolic debugger. http://www.gnu.org, 2001.
[2] GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`, Oct. 2005.
[3] H Hug. Design and Implementation Of An Emulator For the IBM PowerNP 4GS3. DA 2004-08, ETH Zürich, Switzerland, Mar. 2004.
[4] IBM Corp. IBM PowerNP NP4GS3 databook. `http://www.ibm.com`, 2002.
[5] IEEE-1149. JTAG. http://grouper.ieee.org/groups/1149, 1997-2003.
[6] Intel Corp. Intel IXP2xxx hardware reference manual. http://www.intel.com, 2003.

Zürich, den 01.04.2006

Lukas Ruf

3

C Schedule

Schedule

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	03. 04 – 09. 04	10. 04 – 16.04	17.04 – 23.04	24.04 – 30.04	01.05 – 07.05	08.05 – 14.05	15.05 – 21.05	22.05 – 28.05	29.05 – 04.06	05.06 – 11.06	12.06 – 18.06	19.06 – 25.06	26.06 – 02.07	03.07 – 09.07	10.07 – 16.07
read docu	▨	▨				▨	▨								
microengine		▨	▨	▨	▨	▨	▨								
external units								▨	▨	▨	▨	▨			
testing													▨	▨	▨
Integration														▨	▨
presentation										▨				▨	▨
documentation			▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨

References

[1] *Intel IXP2400/IXP2800 Network Processor - Programmer's Reference Manual*, Intel, 2003.

[2] *Intel IXP2400 Network Processor - Hardware Reference Manual*, Intel, 2003.

[3] *Intel IXP2400/IXP2800 Network Processor - Development Tools User's Guide*, Intel, 2004.

[4] *IXP2400 Network Processor - Microengine Instruction Format*, http://ixp2xxx.sourceforge.net, 2005.

[5] Erik J. Johnson, Aaron R. Kunze: *IXP2400/IXP2800 Programming -The Complete Microengine Coding Guide*, Intel Press, 2003.

[6] Douglas E. Comer: *Network Systems Design using Network Processors*, Prentice Hall, 2004.

[7] *PowerNP NP4GS3 Network Processor*, http://www.ibm.com, 2003.

[8] Hanspeter Hug: *Personal Computer Emulator Framework*, http://www.hampa.ch, 2004.

[9] Hanspeter Hug: *Design And Implementation Of An Emulator For The IBM PowerNP 4GS3*, ETH Zuerich, 2003.

[10] Victor Moya del Barrio: *Study of the techniques for emulation programming*, FIB UPC, 2001.

[11] Yan Luo, Jun Yang, Laxmi N. Bhuyan, Li Zhao: *NePSim: A Network Processor Simulator with Power Evaluation Framework*, University of California, 2004.