THM | CAMPUS GIESSEN | MNI
Mathematik, Naturwissenschaften und Informatik
TECHNISCHE HOCHSCHULE MITTELHESSEN

Master thesis

Development of a Simulator for the 64-Bit RISC Processor MMIX

June 14, 2011

Author:
Nils Asmussen

Referee:
Prof. Dr. Hellwig Geisse
Second referee:
Prof. Dr. Thomas Karl Letschert

Contents

1 Introduction **7**
- 1 Current Status . . . 7
- 2 Motivation . . . 8

2 The MMIX Architecture **9**
- 1 General Terms and Notations . . . 9
- 2 Instruction Format . . . 10
- 3 Registers . . . 11
  - 3.1 Special Registers . . . 11
  - 3.2 Global and Local Registers . . . 13
- 4 Integer Arithmetic . . . 13
  - 4.1 Addition and Substraction . . . 13
  - 4.2 Multiplication and Division . . . 14
- 5 Bit Fiddling . . . 15
  - 5.1 Basic Bit Operations . . . 15
  - 5.2 Wyde Operations . . . 15
  - 5.3 Exotic Bit Operations . . . 16
- 6 Comparisons . . . 17
- 7 Branches and Jumps . . . 18
- 8 Memory . . . 19
  - 8.1 Load Instructions . . . 20
  - 8.2 Store Instructions . . . 20
  - 8.3 Virtual and Physical Address Space . . . 21
    - 8.3.1 Address Translation . . . 22
    - 8.3.2 Translation Caches . . . 25
    - 8.3.3 Physical Memory Caches . . . 26
- 9 Floating Point Operations . . . 27
  - 9.1 Representation of Floating Point Numbers . . . 27
  - 9.2 Arithmetic . . . 28
  - 9.3 Comparison . . . 29
  - 9.4 Neighborhood Comparison . . . 29
  - 9.5 Conversion between Float and Integer . . . 31
  - 9.6 Short Floats . . . 31
- 10 Register Stack . . . 32
  - 10.1 Subroutine Linkage . . . 32
    - 10.1.1 Programmers View . . . 32
    - 10.1.2 Internal View . . . 34
  - 10.2 Saving and Restoring the State . . . 36
- 11 Interrupts and Exceptions . . . 37

11.1 Triggering of Trips and Traps . . . 37
11.1.1 Triggering Trips . . . 37
11.1.2 Triggering Traps . . . 38
11.2 Handling of Trips and Traps . . . 39
11.2.1 Resuming . . . 40
11.2.2 Handling Trips . . . 41
11.2.3 Handling Traps . . . 41
11.3 Interruptibility . . . 42

3 The Implementation of GIMMIX 43
1 Basic Design Decisions . . . 43
1.1 No Pipelining . . . 43
1.2 Uninterruptible Instructions . . . 43
1.3 Programming Language . . . 44
1.4 Host Platform . . . 44
2 Overview . . . 45
2.1 Exception . . . 45
2.2 Event . . . 46
2.3 Config . . . 46
3 Simulator Core . . . 47
3.1 Structure . . . 47
3.2 CPU . . . 47
3.2.1 Executing an Instruction . . . 47
3.2.2 Execution Functions . . . 49
3.2.3 Arithmetic . . . 51
3.2.4 Register . . . 52
3.2.5 Trips and Traps . . . 60
3.3 MMU . . . 63
3.3.1 Reading from Memory . . . 64
3.3.2 Writing to Memory . . . 65
3.3.3 Address Translation . . . 65
3.3.4 Translation Cache . . . 67
3.4 Cache . . . 67
3.4.1 Organisation . . . 67
3.4.2 Reading and Writing . . . 68
3.4.3 Implementation of Caching Instructions . . . 69
3.5 Bus . . . 70
4 Devices . . . 71
5 Command Line Interface . . . 72
5.1 The Language . . . 72
5.2 Commands . . . 74
5.2.1 Print and Set . . . 74
5.2.2 Execution of Instructions . . . 74
5.2.3 Examining the State . . . 75
5.3 Implementation . . . 76
5.3.1 General Structure . . . 76
5.3.2 Command Infrastructure . . . 77
5.3.3 Command Implementation . . . 77

4 Test System **81**
1 Program Tests . . . 81
1.1 Test Infrastructure . . . 81
1.2 Test Programs . . . 82
1.2.1 User Tests . . . 82
1.2.2 Kernel Tests . . . 83
1.2.3 Diff Tests . . . 83
1.2.4 CLI Tests . . . 83
1.3 Code Coverage . . . 83
2 Unit Tests . . . 84

5 Future Possibilities **85**
1 Missing Parts for an Operating System Port . . . 85
1.1 `TRAP 0,0,0` halts the Simulator . . . 85
1.2 Toolchain . . . 85
1.3 Startup and Tools . . . 86
2 Extensions and Enhancements . . . 86
2.1 More Devices . . . 86
2.2 Working with Symbols in the CLI . . . 86
2.3 Interface to GDB . . . 86
2.4 Infrastructure for MMIX Programs without OS . . . 87
2.5 Acid Test Mode . . . 87
2.6 Graphical User Interface . . . 87
2.7 Provide Information about Hardware for Software . . . 88
2.8 Mapping of the I/O Space . . . 88

Glossary **89**

Listings **91**

Bibliography **93**

Eidesstattliche Erklärung **97**

Chapter 1

Introduction

This master thesis is about MMIX (pronounced "em-mix") and the implementation of a simulator for it. MMIX is a computer architecture designed by Donald Knuth as a successor of MIX, which is used as an abstract machine in The Art of Computer Programming. The name has been determined by averaging the identifying numbers of 14 similar machines:

(Cray I + IBM 801 + RISC II + Clipper C300 + AMD 29K + Motorola 88K
+ IBM 601 + Intel i960 + Alpha 21164 + POWER 2 + MIPS R4000
+ Hitachi SuperH4 + StrongARM 110 + Sparc 64) / 14
= 28126 / 14 = 2009

The representation of 2009 in roman numerals is MMIX, which is the reason for the name. [1, pg. 2] The name MIX has been chosen analogously. Apart from the name, MMIX has not much in common with MIX. MMIX is a 64-bit big-endian binary computer with 8 bits per byte, that has a RISC instruction set. On the other hand, MIX is a hybrid binary-decimal computer with 6 bits per byte and a CISC-like instruction set [2], [3, pg. 124,125].

1 Current Status

At the beginning of this project, two simulators for MMIX were already available, called MMIX-SIM and MMIX-PIPE. Both were developed by Donald Knuth himself and are published with MMIXware[1], which contains the simulators, the full documentation and example programs. The simulators have been written with the literate programming system CWEB, also designed by Donald Knuth, that is a mixture of TEX and C, from which both compilable C code and documentation can be generated. MMIX-SIM is a simple, instruction-level simulator, that does only support user-programs, i. e. no operating system kernel. That is, all user mode instructions are available, but no interrupts or exceptions can be handled, no paging is supported and no caches are present. It is an instruction-level simulator in the sense, that each instruction takes exactly one cycle and one can thus step through a program instruction per instruction. Its main goal is to be able to run and analyze example programs published in The Art of Computer Programming. [4, pg. 1] On the other hand, MMIX-PIPE is a highly configurable meta-simulator, that supports all features the MMIX architecture has in mind. In contrary to MMIX-SIM, it uses pipelining and the instructions take an

[1] It can be downloaded at http://www-cs-faculty.stanford.edu/~uno/mmix-news.html.

arbitrary and varying number of cycles. Due to the degree of configurability regarding registers, caches, functional units and so on, which is also the reason for the name "meta-simulator", it can for example be utilized to explore what settings are the most suitable ones for building a hardware implementation of MMIX. Furthermore, the user interface of MMIX-PIPE is not designed to analyze a program – like MMIX-SIM – but rather the machine it runs on.

2 Motivation

The longterm goal of this project is to port an operating system to MMIX. Unfortunatly, neither MMIX-SIM nor MMIX-PIPE are well suited for that task. Because, as just mentioned, MMIX-SIM does not support OS kernels at all and MMIX-PIPE would only appropriate if, for example, one liked to explore what cache-configuration or how many functional units are ideal for a hardware implementation. It does not fit well when one would like to develop, port or debug an operating system for MMIX.

Of course, it would be possible to change MMIX-SIM or MMIX-PIPE to fit our needs. But MMIX-SIM does not implement most of the complicated mechanisms MMIX offers, such as paging, interrupt and exception handling or caching. Therefore, it would require many changes to its code, which is quite difficult to understand and especially to adjust or extend. MMIX-PIPE has all these mechanisms, but uses pipelining and is highly configurable, which increases the complexity of the code by an order of magnitude, compared to MMIX-SIM. Additionally, the implications of the mentioned goals of MMIX-PIPE, do not fit well for this project.

For these reasons, it has been decided to write a new simulator from scratch[2]. This way, the system is easier to understand for people who want to learn how MMIX works, is completely in our control, which makes changes simple, and can be designed so that it perfectly matches with our needs. The conformity to the MMIX architecture specification is ensured by a sophisticated test system, that tries to test every possible case and compares it with MMIX-SIM and MMIX-PIPE.

This thesis explains at first the architecture MMIX in general and describes afterwards the implementation of the simulator, called GIMMIX. The name stands for "Gießen Implementation of MMIX", because Gießen is the home town of our university. Both GIMMIX and this thesis are based on the version 20110305 of MMIXware. That means, GIMMIX is implemented to match with the specification of that version, uses MMIX-SIM and MMIX-PIPE published with it and this thesis describes the mentioned version.

[2] A first approach had already been started a few years ago, but for design reasons it has been decided to start on a green field again.

Chapter 2

The MMIX Architecture

As already mentioned in the introduction, MMIX is a 64-bit big-endian RISC machine. It provides 256 32-bit wide instructions, at least 256 general purpose registers and 32 special registers. Both are 64-bit wide. Additionally it has a 64-bit virtual and physical address space and supports both integer arithmetic and floating point arithmetic. Donald Knuth described the goals of MMIX with "I strove to design MMIX so that its machine language would be simple, elegant, and easy to learn. At the same time I was careful to include all of the complexities needed to achieve high performance in practice, so that MMIX could in principle be built and even perhaps be competitive with some of the fastest general-purpose computers in the marketplace." [5, pg. v].

This chapter splits the features of MMIX in categories and describes them one after another. In each category the concept is explained, if necessary, and the associated instructions are introduced. It will mostly resemble the MMIX specification [6], but of course, this thesis has a different purpose, i. e. some parts will be described in more detail, some in less. Especially, this thesis will try to give more examples to the difficult chapters of MMIX. But after all, it is of course not meant to be a replacement for the specification. Thus, whenever a concept or instruction is explained, the end of it will link to the corresponding page of the specification.

1 General Terms and Notations

Before MMIX is described in further detail, a few terms and notations that are used throughout this thesis should be introduced.

At first, numbers without any prefix or other qualification should be read in decimal base, whereas numbers prefixed with '`#`' should be read in hexadecimal base. General purpose registers are named `$X`, where `X` is between 0 and 255. Special registers are named `rX`, where `X` is any of `A`, `B`, ..., `Z`, `BB`, `TT`, `WW`, `XX`, `YY`, `ZZ`. The quantities in MMIX are:

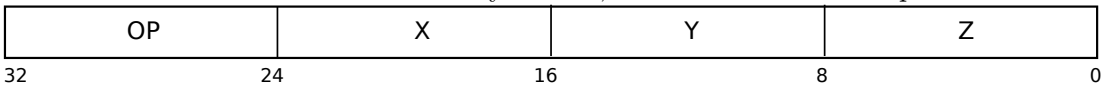
Name	Bits	Unsigned and signed integer range
Byte	8	$0 \ldots 255$ $-128 \ldots 127$
Wyde	16	$0 \ldots 65535$ $-32768 \ldots 32767$
Tetra	32	$0 \ldots 4,294,967,295$ $-2,147,483,648 \ldots 2,147,483,647$
Octa	64	$0 \ldots 18,446,744,073,709,551,615$ $-9,223,372,036,854,775,808 \ldots 9,223,372,036,854,775,807$

Table 1: Quantities in MMIX [6, pg. 3]

The virtual memory is an array called M. An access of 2^t consecutive bytes at location k is written as $M_{2^t}[k]$, where k is 2^t-byte aligned (the least significant t bits are zero). That means, for example M_1[#1234] denotes the byte at location #1234 and M_8[#100] the octabyte at location #100. The virtual memory is divided in two halfs. The memory space #0000 0000 0000 0000 ... #7FFF FFFF FFFF FFFF is called *user space* and #8000 0000 0000 0000 ... #FFFF FFFF FFFF FFFF is called *privileged space*. Furthermore, the location of the instruction pointer, called @, determines the mode in which MMIX operates. If it is in user space, it is in *user mode*, otherwise in *privileged mode*. Finally, MMIX distinguishes between *arithmetic exceptions* (AE) like division by zero or integer overflow, which are handled by the user application, *program exceptions* (PE) such as privileged instruction or protection fault, which are handled by the operating system, and *machine exceptions* (ME) like power failure, which are as well handled by the OS.

2 Instruction Format

Each MMIX instruction is described by a tetra, which consists of four parts:



The first byte is the *opcode* of the instruction and the other three bytes specify the *operands*. Thus, MMIX allows (and uses) 256 instructions with 3 operands, each having 256 possible values. The typical instruction has the meaning "Set register X to the result of Y OP Z". [6, pg. 2]

In this thesis, all instructions are at first outlined with a box that contains the mnemonic, the operands of the instruction and the its effect. Afterwards the instruction is illustrated in further detail – including special cases and raised AEs or PEs. The box looks like the following:

Name:	ADD|SUB $X,$Y,$Z|Z
Effect:	$X $\leftarrow s(\$Y) + \mid - s(\$Z) \mid$Z

This describes the four instructions ADD, ADDI, SUB and SUBI. As in this example, many MMIX instructions come in two forms: In the first one, the Z-operand is a register, in the second one it is an immediate value. Since there is no other difference, these are handled at once by saying "$Z|Z". Furthermore, ADD and SUB are very similar, so that they are grouped together. The notation $s(...)$ denotes, that MMIX interprets it as signed and uses two's complement arithmetic. Otherwise it means that MMIX treats the value as unsigned. Thus, the effect in the box shown above can be read as

- `ADD` sets `$X` to the result of the addition of `$Y` and `$Z`, interpreting both as signed values and hence, using signed arithmetic,
- `ADDI` sets `$X` to the result of the addition of the signed value `$Y` and unsigned immediate value `Z`,
- `SUB` sets `$X` to the result of the substraction of the signed value `$Y` and signed value `$Z` and
- `SUBI` sets `$X` to the result of the substraction of the signed value `$Y` and unsigned immediate value `Z`.

Immediate values are always interpreted unsigned in MMIX. Additionally, if one of the operands `X`, `Y` and `Z` is not mentioned in the name, it means implicitly that the corresponding byte has to be zero. If it is not, MMIX will raise a *breaks rules* PE.

Although the effects description will be straight forward for most of the instructions, it is not meant to be always complete or self-explaining, because that would be too verbose and would require a formal language definition for some instructions. Rather, it should be seen as a quick overview of what the instruction does. Thus, if necessary, the text below the box will clarify the effects description or adds further information.

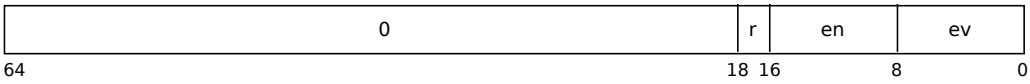
3 Registers

Since the registers are one of the most important entities in MMIX, they are explained at first. MMIX has special, global and local registers, which will be described one after another in this section.

3.1 Special Registers

MMIX provides 32 special registers in an array called *sp* in this thesis[1]. Most of them will be introduced later as soon as the associated concept or instruction is described. The other ones, that do not fit into a certain category or are very important, are explained here.

- `rA` - Arithmetic status register:
  Since `rA` affects many instructions, it is explained first. Its layout is:



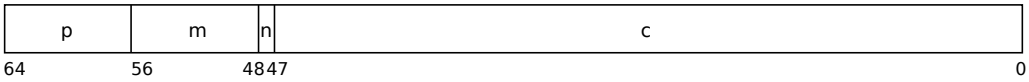
  The fields *en* and *ev* contain *enable* and *event* bits, both called abbreviated "DVWIOUZX from left to right, where D stands for integer divide check, V for integer overflow, W for float-to-fix overflow, I for invalid operation, O for floating overflow, U for floating underflow, Z for floating division by zero, and X for floating inexact." [6, pg. 26] The enable bits control whether an AE is raised as soon as the corresponding exceptional condition occurs, while the event bits will be set if no AE is raised. The field *r* specifies the rounding mode that is used for floating point numbers, where 00_2 means round near, 01_2 round toward zero, 10_2 round toward $+\infty$ and 11_2 round toward $-\infty$. All other bits are defined to be zero. [6, pg. 15 and 26]

[1] The special registers may be put in the first 32 slots of the global register array, which is the reason why `rG` is always at least 32, as it will be mentioned in the next section. But actually, this is not enforced.

- `rC` - Cycle counter:
  As the name suggests, MMIX increases this special register on every cycle by 1. It can be used to measure the performance of a code snippet, for example. [6, pg. 32]

- `rI` - Interval counter:
  The special register `rI` is decreased by 1 on every cycle and causes an *interval interrupt* as soon as it reaches zero. It can also be used for runtime analysis. [6, pg. 32]

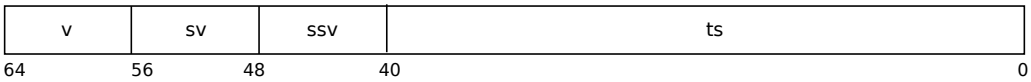
- `rU` - Usage counter:
  Register `rU` is structured in the following way:



  The usage count c is increased whenever $op \ \& \ m = p$, where op is the opcode of an instruction. The bit n indicates whether it should also be done when the instruction pointer is in the privileged space. [6, pg. 32]

- `rF` - Failure location register:
  This register holds the physical memory address when a parity error or other kinds of memory faults occur. Since an MMIX implementation may use caching, the instruction that caused this error might be long gone before it is detected. [6, pg. 40]

- `rN` - Serial number:
  Register `rN` identifies the particular MMIX implementation and is structured as follows:



  The fields v, sv and ssv specify the MMIX architecture version. This thesis describes version 1, subversion 0 and subsubversion 0, i. e. 1.0.0. The field ts holds the number of seconds from 01/01/1970, 00:00:00 GMT to the date the particular instance of MMIX was built on. [6, pg. 32]

MMIX provides two instructions to read from or write to a special register.

Name:	`GET $X,Z`
Effect:	`$X` $\leftarrow$ *sp*`[Z]`

The instruction `GET` sets `$X` to the value of special register `Z`. MMIX does not keep secrets from the user, i. e. all special registers are readable – even in user mode. [6, pg. 34]

Name:	`PUT X,$Z|Z`
Effect:	*sp*`[X]` $\leftarrow$ `$Z|Z`

`PUT` sets special register `X` to either `$Z` or the immediate value `Z`. The registers `rC`, `rN`, `rO` and `rS` are not writeable in general. `rI`, `rT`, `rTT`, `rK`, `rQ`, `rU` and `rV` are writeable in privileged mode only. Additionally, in `rA` all bits except `#3FFFF` have to be zero, `rG` can't be less than $max(\texttt{rL}, 32)$ and not greater than 255. Furthermore, `rL` can't be increased with `PUT` and bits in the *interrupt request register* `rQ`, that have been set by

MMIX since the last execution of `GET $X,rQ`, can not be unset. In this way, no PE, ME or interrupt bit can be lost by accident.[2] [6, pg. 34]

3.2 Global and Local Registers

MMIX maintains two banks of registers. One for global registers, called g, which has at most 256 slots. The other one for local registers, called l, which has 2^n slots, where n is between 8 and 10. Both can be accessed by the so called *dynamic registers* `$0`, `$1`, ..., `$255`. MMIX uses the special register `rG` to separate them. `rG` is always at least 32 at at most 255. When saying `$X`, it denotes a global register whenever `X` is greater or equal to `rG`. It will denote a local register if it is less than `rG`. [6, pg. 22]

Additionally, `rL` splits the local registers in two categories. The registers `$0`, ..., `$(rL - 1)` are the local registers that are currently in use. The other ones, `$(rL)`, ..., `$(rG - 1)`, are called *marginal*. If such a register is read, it will always yield zero. If `$X` is written, whereas $\texttt{rL} \leq \texttt{X} < \texttt{rG}$, the registers `$(rL)`, ..., `$(X - 1)` will be set to zero, `$X` will be set to the desired value and `rL` will be set to $\texttt{X} + 1$. [6, pg. 22]

For example, if `rL` is 4 and `rG` is 64,

- reading `$3` will yield the value of $l[3]$,
- reading `$4` will yield 0,
- writing 12 to `$5` will set $l[4]$ to 0, $l[5]$ to 12 and `rL` to 6,
- reading `$64` will yield the value of $g[64]$ and
- writing 100 to `$70` will set $g[70]$ to 100.

4 Integer Arithmetic

Of course, MMIX provides some instructions to perform integer arithmetic. That is, addition, substraction, multiplication, division and some more. For most of these instructions, MMIX has an unsigned and a signed version. The only difference when adding or substracting is, that the signed versions raise arithmetic exceptions – if necessary – while the others will not. When multiplying or dividing, the rules for signed or unsigned arithmetic have to be considered.

4.1 Addition and Substraction

Name:	`ADD|SUB $X,$Y,$Z|Z`
Effect:	$\texttt{\$X} \leftarrow s(\texttt{\$Y}) + \mid -s(\texttt{\$Z})\mid \texttt{Z}$

The sum or difference of `$Y` and `$Z|Z` is put into `$X`. An integer overflow AE will be raised if the result is $\geq 2^{63}$ or $< -2^{63}$. [6, pg. 6]

Name:	`ADDU|SUBU $X,$Y,$Z|Z`
Effect:	$\texttt{\$X} \leftarrow (\texttt{\$Y} + \mid -\texttt{\$Z}\mid \texttt{Z}) \bmod 2^{64}$

The sum or difference of `$Y` and `$Z|Z` is put into `$X`. [6, pg. 6]

[2] The restrictions will become more clear as soon as the concepts and instructions working with these registers have been explained.

Name:	`NEG $X,Y,$Z|Z`
Effect:	$\$X \leftarrow Y - s(\$Z)\|Z$

MMIX provides a separate instruction for negation to save the programmer from having to put a constant into a register first. This way, e. g. `$Z` can be negated by saying `NEG $X,0,$Z`. It can also be used for building the value -1: `NEG $X,0,1`. The instruction will throw an overflow AE if the result is $> 2^{63} - 1$. [6, pg. 6]

Name:	`NEGU $X,Y,$Z|Z`
Effect:	$\$X \leftarrow (Y - \$Z\|Z) \bmod 2^{64}$

This instruction has the same effect as `NEG`, but does not raise an AE. [6, pg. 6]

4.2 Multiplication and Division

The more expensive integer arithmetic operations are multiplication and division. MMIX supports 64-bit signed multiplication and division and 128-bit unsigned multiplication and division. The division instructions perform a division and modulo calculation at once. Additionally it is worth noting that MMIX uses the so called *floored division*. This means that division rounds towards negative infinity and that the sign of the modulus is always the same as the sign of the divisor [7, pg. 2]. For example, this differs from the x86 architecture, which uses *truncated division* [8, pg. 560], that rounds towards zero and gives the modulus the sign of the dividend [7, pg. 2]. The following table illustrates the differences:

Y,Z	$trunc(Y/Z)$	$trunc(Y\%Z)$	$\lfloor Y/Z \rfloor$	$\lfloor Y\%Z \rfloor$
$+8, +3$	$+2$	$+2$	$+2$	$+2$
$+8, -3$	-2	$+2$	-3	-1
$-8, +3$	-2	-2	-3	$+1$
$-8, -3$	$+2$	-2	$+2$	-2

Table 2: Comparison of truncated and floored division [7, pg. 3]

The differences occur for numbers with different signs only, i. e. the unsigned division does behave in the same way regardless of using the floored or truncated algorithm.

Name:	`MUL $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow s(\$Y) * s(\$Z)\|Z$

The `MUL` instruction sets `$X` to the result of the multiplication. It will raise an integer overflow AE if the result is $\geq 2^{63}$ or $< -2^{63}$. [6, pg. 14]

Name:	`MULU $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow (\$Y * \$Z\|Z) \bmod 2^{64}, \quad \texttt{rH} \leftarrow (\$Y * \$Z\|Z) \gg 64$

This instruction basically does the same as `MUL`, but treats the operands as unsigned and places the upper 64 bit of the result into the special *himult register* `rH` and does not raise an overflow AE. [6, pg. 14]

Name:	`DIV $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow \lfloor s(\$Y) \ / \ s(\$Z)\|Z \rfloor, \quad \texttt{rR} \leftarrow s(\$Y) \bmod s(\$Z)\|Z$

The instruction `DIV` sets `$X` to the result of the division and the *remainder register* `rR` to the result of the modulo operation. If $s(\$Z)\|Z$ is zero, a division by zero AE will be

raised, $X will be set to zero and rR will be set to $Y. An integer overflow AE will occur if and only if -2^{63} is divided by -1. [6, pg. 14]

Name:	DIVU $X,$Y,$Z|Z
Effect:	$X $\leftarrow \lfloor$ rD$Y / $Z|Z $\rfloor$, rR $\leftarrow$ rD$Y mod $Z|Z

Analogous to MULU, DIVU prefixes the *dividend register* rD to $Y, resulting in a 128-bit number, and divides it by $Z|Z, using unsigned arithmetic. If rD $\geq$ $Z|Z (this includes the case that $Z|Z is zero), $X will be set to rD and rR will be set to $Y. Additionally, no AE is raised. [6, pg. 14]

Name:	2ADDU|4ADDU|8ADDU|16ADDU $X,$Y,$Z|Z
Effect:	$X $\leftarrow ((2|4|8|16 * $Y) + $Z|Z) mod 2^{64}

As usual, if a number should be divided or multiplied by a power of 2, shifts are much more efficient, which will be described later. MMIX goes even further by providing instructions that multiply a number by 2, 4, 8 or 16 and adding the result to another value. In this way, one can easily e.g. multiply by 3 by saying 2ADDU $X,$Y,$Y. [6, pg. 6]

5 Bit Fiddling

MMIX has quite a few instructions for manipulating bits. At first, the well known bit operations AND, OR, NOR, ... and shifts are described, because they will not be a surprise.

5.1 Basic Bit Operations

Name:	AND|OR|XOR|ANDN|ORN|NAND|NOR|NXOR $X,$Y,$Z|Z
Effect:	$X $\leftarrow$ $Y $\wedge \mid \vee \mid \oplus \mid \wedge \sim \mid \vee \sim \mid \overline{\wedge} \mid \overline{\vee} \mid \overline{\oplus}$ $Z|Z

These instructions set $X to the result of the corresponding bit operation with operands $Y and $Z|Z. The instructions that end with 'N' logically negate $Z|Z first and apply the operation without 'N' (OR or AND) afterwards. [6, pg. 7]

Name:	SL|SLU|SR|SRU $X,$Y,$Z|Z
Effect:	$X $\leftarrow$ $Y $\ll \mid \ll \mid \gg \mid \ggg$ $Z|Z

The instructions for shifting left, SL and SLU, have the same behaviour, except that SL will raise an integer overflow AE, if the result is $\geq 2^{63}$ or $< -2^{63}$. SR performs an arithmetic right shift, i. e. it shifts in copies of the sign bit from the left, and SRU performs a logical right shift, i. e. it shifts in zeros from the left. Since $Z|Z is treaten unsigned, one can not use SL to shift right or similar. Additionally it is worth mentioning, that a logical left or right shift of 64 or more will set $X to zero, whereas an arithmetic right shift of 64 or more will set $X to -1, if $Y is negative and to zero otherwise. [6, pg. 10]

5.2 Wyde Operations

If one liked to put an arbitrary 64-bit constant into a register or manipulate individual wydes of a registers, one could use one of the following 16 instructions.

Name:	SETH|SETMH|SETML|SETL $X,YZ
Effect:	$X ← YZ ≪ 48|32|16|0

These instructions set the corresponding wyde of $X to the 16-bit constant YZ and all other wydes to zero [6, pg. 7]. That means, for example SETMH $X,#1234 sets $X to #0000 1234 0000 0000.

Name:	ORH|ORMH|ORML|ORL $X,YZ
Effect:	$X ← $X ∨ (YZ ≪ 48|32|16|0)

Similarly, these instructions OR the 16-bit constant YZ into the corresponding wyde of $X [6, pg. 7]. For example, if $X is #0000 F0F0 FF00 0000, ORML $X,#0FF0 will result in #0000 F0F0 FFF0 0000.

Name:	ANDNH|ANDNMH|ANDNML|ANDNL $X,YZ
Effect:	$X ← $X ∧ ∼ (YZ ≪ 48|32|16|0)

Analogous to the ORX family, these instructions remove the bits set in the 16-bit constant YZ from the corresponding wyde of $X [6, pg. 7]. For example, if $X is #0000 F0F0 FF00 0000, ANDNML $X,#F000 will result in #0000 F0F0 0F00 0000.

Name:	INCH|INCMH|INCML|INCL $X,YZ
Effect:	$X ← ($X + (YZ ≪ 48|32|16|0)) mod 2^{64}

Last but not least, the INCX family adds the 16-bit constant YZ to the corresponding wyde of $X, ignoring overflow [6, pg. 7]. For example, if $X is #0000 F0F0 FF00 0000, INCML $X,#0101 will result in #0000 F0F1 0001 0000 (as shown with the example, other wyde may be affected as well, in contrast to the other wyde instructions).

5.3 Exotic Bit Operations

Apart from the simple bit operations just described, MMIX does also support more exotic ones that probably will not be used very often, but allow to do complicated computations in hardware instead of in software, as it would be necessary with other architectures.

Name:	MUX $X,$Y,$Z|Z
Effect:	$X ← ($Y ∧ rM) ∨ ($Z|Z ∧ $\overline{\texttt{rM}}$)

The first rather exotic operation is the *bitwise multiplexer* MUX. For each bit position i, it sets bit Y_i, if rM$_i$ is 1, and bit Z_i$\|Z$_i$, if rM$_i$ is 0 [6, pg. 7]. For example, if rM is #FFFF 0000 FFFF 0000, $0 is #1234 5678 90AB CDEF and $1 is #FFFF FFFF FFFF FFFF, a MUX $X,$0,$1 will set $X to #1234 FFFF 90AB FFFF.

Name:	BDIF|WDIF|TDIF|ODIF $X,$Y,$Z|Z
Effect:	X_i ← $max(0, Y_i − Z_i$\|Z$_i$) for each byte|wyde|tetra|octa i

The second family in this category is the byte, wyde, tetra and octa difference. For example, BDIF takes the byte i of $Y, namely Y_i, and substracts the byte Z_i$\|Z$_i$ from it. If the difference is less than zero, X_i will be set to zero. Otherwise it will be set to the difference. This is done individually for every byte pair. WDIF, TDIF and ODIF behave analogous using wydes, tetras and octas, respectively. These instructions are for example useful, when a graphical application wants to calculate the "pixel difference", i. e. the absolute difference of colors, each color component represented as a byte. [6, pg. 8]

Name:	`SADD $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow countbits(\$Y \setminus \$Z|Z)$

The *sideways addition* `SADD` performs at first the complement of `$Z|Z` and logically ands the result with `$Y`. Afterwards the number of set bits in this value is put into `$X`. [6, pg. 9] So, for example, if `$0` is `#8642` and `$1` is `#8002`, `SADD $X,$0,$1` will put 3 into `$X`. Because the difference, `#0640`, has 3 bits set.

Name:	`MOR|MXOR $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow mat(\$Y) \vee | \oplus mat(\$Z|Z)$

The last exotic bit operation, called *multiple or/exclusive-or*, is the most complicated one. It treats `$Y` and `$Z|Z` as 8×8 bit matrices, using one byte for each column, and performs a kind of matrix product using `OR` or `XOR` instead of the multiplication. More precisely, when the bits of `$Y` and `$Z|Z` are numbered as

$$y_{00}y_{01}\dots y_{07}y_{10}y_{11}\dots y_{17}\dots y_{70}y_{71}\dots y_{77} \quad z_{00}z_{01}\dots z_{07}z_{10}z_{11}\dots z_{17}\dots z_{70}z_{71}\dots z_{77},$$

each bit x_{ij} of `$X` is set to

$$(y_{0j} \wedge z_{i0}) \vee (y_{1j} \wedge z_{i1}) \vee \dots \vee (y_{7j} \wedge z_{i7}).$$

When using `MXOR` instead of `MOR`, the `OR`s are replaced by `XOR`s. `MOR` can be used for example to convert between big-endian and little-endian. [6, pg. 9] If `$0` is `#0123 4567 89AB CDEF` and `$1` is `#0102 0408 1020 4080`, a `MOR $X,$0,$1` will perform the following operation:

$$\begin{pmatrix} 0&0&0&0&1&1&1&1\\ 0&0&1&1&0&0&1&1\\ 0&1&0&1&0&1&0&1\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&1&1&1&1\\ 0&0&1&1&0&0&1&1\\ 0&1&0&1&0&1&0&1\\ 1&1&1&1&1&1&1&1 \end{pmatrix} \vee \begin{pmatrix} 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0 \end{pmatrix} = \begin{pmatrix} 1&1&1&1&0&0&0&0\\ 1&1&0&0&1&1&0&0\\ 1&0&1&0&1&0&1&0\\ 0&0&0&0&0&0&0&0\\ 1&1&1&1&0&0&0&0\\ 1&1&0&0&1&1&0&0\\ 1&0&1&0&1&0&1&0\\ 1&1&1&1&1&1&1&1 \end{pmatrix}$$

Analogous to the matrix product, the highlighted cell in the result matrix is built by `AND`ing each cell in the highlighted row of `$Y` with the corresponding highlighted cell of `$Z|Z` individually, starting on the left and top, respectively, and performing an `OR` of all these values. In this case, there is no pair of bits in which both are 1 and thus, the highlighted cell in the result is 0. Doing that for all cells leads to the value `#EFCD AB89 6745 2301`, i. e. the bytes of `$0` in the opposite order. [9, pg. 192]

6 Comparisons

MMIX has four instructions to compare numbers, which for example can be used for branching. Additionally, it has instructions to conditionally set and zero or set a register.

Name:	`CMP $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow (s(\$Y) > s(\$Z|Z) - (s(\$Y) < s(\$Z|Z)$

The instruction `CMP` compares `$Y` with `$Z|Z` using signed arithmetic and puts the result into `$X`. If `$Y` is less than `$Z|Z`, `$X` will be set to -1, if they are equal, `$X` will be set to 0 and if `$Y` is greater than `$Z|Z`, `$X` will be set to 1. [6, pg. 11]

Name:	`CMPU $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow (\$Y > \$Z|Z) - (\$Y < \$Z|Z)$

This instruction behaves like `CMPU`, but uses unsigned arithmetic. [6, pg. 11]

Name:	`CSN|CSZ|CSP|CSOD|CSNN|CSNZ|CSNP|CSEV $X,$Y,$Z|Z`
Effect:	if $s(\$Y) < 0 | = 0 | > 0 | odd | \geq 0 | \neq 0 | \leq 0 | even$: $\$X \leftarrow \$Z|Z$

The family of conditional set instructions will set `$X` to `$Z|Z`, if `$Y` is negative, zero, positive, odd, nonnegative, nonzero, nonpositive or even. Otherwise nothing happens. [6, pg. 11]

Name:	`ZSN|ZSZ|ZSP|ZSOD|ZSNN|ZSNZ|ZSNP|ZSEV $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow (s(\$Y) < 0 | = 0 | > 0 | odd | \geq 0 | \neq 0 | \leq 0 | even)\ ?\ \$Z|Z\ :\ 0$

Very similar to the conditional set instructions, the zero or set instructions set `$X` either to `$Z|Z` or zero, depending on whether `$Y` is negative, zero, positive, odd, nonnegative, nonzero, nonpositive or even. [6, pg. 11]

MMIX does also provide an atomic *compare and swap* instruction. It can be used for interprocess communication with shared memory or to synchronize threads in the same process. Since MMIX is not only designed to work on a single processor, this instruction might also be helpful when independent computers are sharing the same memory. [6, pg. 25]

Name:	`CSWAP $X,$Y,$Z|Z`

Effect:	if $M_8[\$Y + \$Z|Z] = \texttt{rP}$:
	$M_8[\$Y + \$Z|Z] \leftarrow \$X, \quad \$X \leftarrow 1$
	else:
	$\texttt{rP} \leftarrow M_8[\$Y + \$Z|Z], \quad \$X \leftarrow 0$

The *compare and swap octabytes* instruction compares $M_8[\$Y + \$Z|Z]$ with the special *prediction register* `rP` and either replaces the octa in memory with `$X` or `rP` with the octa in memory, depending on whether `rP` is equal to the octa. `$X` indicates whether the octa in memory has been replaced. [6, pg. 25] For example, one could set `$0` to 1, `rP` to 0 and do a `CSWAP $0,$Y,$Z|Z`, assuming that $\$Y + \$Z|Z$ denotes the memory location that is desired for synchronization. If `$0` has been set to 1, the lock has been aquired successfully. If not, the whole procedure will be repeated. That means, $M_8[\$Y + \$Z|Z]$ being 1 or 0 would indicate that someone currently has the lock or not, respectively.

7 Branches and Jumps

Of course, MMIX does also need instructions to change the course of computation. To allow programs to use a pipeline implementation of MMIX in an efficient way, it provides both ordinary branches and probable branches. For consistency, the different kinds of comparisons offered by branches are the same as those existing for the conditional set and zero or set instructions.

Similarly to the fact, that the typical "set register `X` to the result of `Y` OP `Z`" instructions come in two versions – one with `Z` as a register, one with `Z` as an immediate value – the branch and jump instructions also come in two versions. The first one branches or jumps forward, while the second one branches or jumps backwards. The backward versions are suffixed with a '`B`'.

Name:	`BN|BZ|BP|BOD|BNN|BNZ|BNP|BEV $X,@+4*(YZ[-2^16])`
Effect:	if $s(\$X) < 0 \mid = 0 \mid > 0 \mid odd \mid \geq 0 \mid \neq 0 \mid \leq 0 \mid even$: $@ \leftarrow @ + 4 * (YZ[-2^{16}])$

If `$X` is negative, zero, positive, odd, nonnegative, nonzero, nonpositive or even, the branch will be taken. The forward versions increase the instruction pointer by $4 * \texttt{YZ}$, i. e. the value of the unsigned 16-bit immediate value `YZ` multiplied with the number of bytes of an instruction. The backward versions increase it by $4 * (\texttt{YZ} - 2^{16})$. Thus, these instructions allow to change @ to any instruction in $(@ - 4 * 2^{16}) \ldots (@ + 4 * (2^{16} - 1))$. It should be noted, that this category of branches tell MMIX that the branch will probably not be taken. This may affect the runtime on some implementations of MMIX. [6, pg. 12]

Name:	`PBN|PBZ|PBP|PBOD|PBNN|PBNZ|PBNP|PBEV $X,@+4*(YZ[-2^16])`
Effect:	if $s(\$X) < 0 \mid = 0 \mid > 0 \mid odd \mid \geq 0 \mid \neq 0 \mid \leq 0 \mid even$: $@ \leftarrow @ + 4 * (YZ[-2^{16}])$

These instructions behave exactly in the same way as the previously introduced ones. The only difference is, that these tell MMIX that the branch will probably be taken. [6, pg. 12]

Name:	`JMP @+4*(XYZ[-2^24])`
Effect:	$@ \leftarrow @ + 4 * (XYZ[-2^{24}])$

Of course, MMIX has also an instruction to change the instruction pointer unconditionally: the jump. It simply sets @ to $(@ + 4 * (\texttt{XYZ}[-2^{24}]))$, i. e. the forward version increases @ by the unsigned 24-bit constant `XYZ`, multiplied by 4. The backward version substracts 2^{24} from `XYZ` before multiplying. Thus, one can jump to any instruction in the range $(@ - 4 * 2^{24}) \ldots (@ + 4 * (2^{24} - 1))$. [6, pg. 13]

Name:	`GO $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow @ + 4, \quad @ \leftarrow \$Y + \$Z \mid Z$

To be able to jump to any location in the virtual address space, MMIX has the instruction `GO`. It simply changes the instruction pointer to $\$Y + \$Z \mid Z$. Additionally, `$X` is set to the location that ordinary would have been executed next. That allows using `GO` for a simple type of subroutine linkage by not overwriting `$X` in the subroutine and returning via `GO $X,$X,0`. But MMIX provides another mechanism, that is much better suited for that task, because it makes subroutines independent of each other, as will be described later in this chapter. An interesting corner case is, that `GO` permits it to jump to addresses that are not tetra-aligned. MMIX will simply set the desired instruction pointer, but this is not going to be a problem because when loading $M_4[@]$, the least significant 2 bits of @ are ignored. [6, pg. 13]

Name:	`GETA $X,@+4*(YZ[-2^16])`
Effect:	$\$X \leftarrow @ + 4 * (YZ[-2^{16}])$

This instruction does not change the instruction pointer, but is related because it builds an absolute address from the instruction pointer and the `YZ` field. `GETA` comes in two versions for forward and backwards calculations and uses the same rules for that as the branches do. [6, pg. 13]

8 Memory

The memory hierarchy is one of the more complicated concepts in MMIX. This section starts by explaining the load and store instructions, which are rather simple. Afterwards

the structure of the virtual and physical address space is described, followed by the translation mechanism and the translation caches. Finally, the instructions interesting for physical memory caches, which may be present in a particular implementation of MMIX, are introduced.

8.1 Load Instructions

MMIX has two kinds of load instructions for each quantity: a signed version and an unsigned version. The difference is that the signed versions treat the number in memory as signed and thus sign-extend the value to 64 bit.

Name:	`LDB|LDW|LDT|LDO $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow s(M_{1\|2\|4\|8}[\$Y + \$Z\|Z])$

These are the signed load instructions, which set `$X` to the byte, wyde, tetra or octa at the specified location in memory. `LDB`, `LDW` and `LDT` will sign-extend the number, i. e. the bits in the upper 7, 6 and 4 bytes, respectively, are set to copies of the sign-bit of the number in memory. [6, pg. 4]

Name:	`LDBU|LDWU|LDTU|LDOU $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow M_{1\|2\|4\|8}[\$Y + \$Z\|Z]$

As already mentioned, the unsigned versions have the same behaviour, except that they do not sign-extend the values. `LDOU` and `LDO` are completely identical and only exist both for consistency. [6, pg. 4]

Name:	`LDHT $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow M_4[\$Y + \$Z\|Z] \ll 32$

The last load instruction, *load high tetra*, puts the tetra $M_4[\$Y + \$Z|Z]$ into the higher half or `$X`. The other half is set to zero. [6, pg. 4]

8.2 Store Instructions

Analogous to the load instructions, MMIX provides two store instructions for every quantity. In this case, the signed versions throw an integer overflow AE, if the number can not be represented with the corresponding quantity, while the unsigned versions do not.

Name:	`STB|STW|STT|STO $X,$Y,$Z|Z`
Effect:	$M_{1\|2\|4\|8}[\$Y + \$Z\|Z] \leftarrow s(\$X)$

These instructions write the signed number in `$X` to the corresponding location in memory. `STB` will throw an integer overflow AE, if `$X` is not between -128 and $+127$, `STW` if `$X` is not between $-32,768$ and $+32,767$ and `STT` if `$X` is not between $-2,147,483,648$ and $+2,147,483,647$. `STO` will not throw an integer overflow AE. [6, pg. 5]

Name:	`STBU|STWU|STTU|STOU $X,$Y,$Z|Z`
Effect:	$M_{1\|2\|4\|8}[\$Y + \$Z\|Z] \leftarrow \$X$

The unsigned store instructions are the same as their signed correspondence, but do not test for overflow. [6, pg. 5]

Name:	`STHT $X,$Y,$Z|Z`
Effect:	$M_4[\$Y + \$Z\|Z] \leftarrow \$X \gg 32$

Similarly to load high tetra, *store high tetra* stores the most significant four bytes of `$X` to $M_4[\$Y + \$Z|Z]$. [6, pg. 5]

Name:	`STCO X,$Y,$Z|Z`
Effect:	$M_8[\$Y + \$Z|Z] \leftarrow X$

For convenience and efficiency, MMIX provides another store instruction, *store constant octabyte*, which stores the unsigned immediate value `X` as an octa to the desired location in memory. This saves the programmer from having to put the constant into a register first. [6, pg. 5]

8.3 Virtual and Physical Address Space

As already mentioned at the beginning, MMIX has both a 64-bit virtual and physical address space. Their layout is illustrated by the following figure:

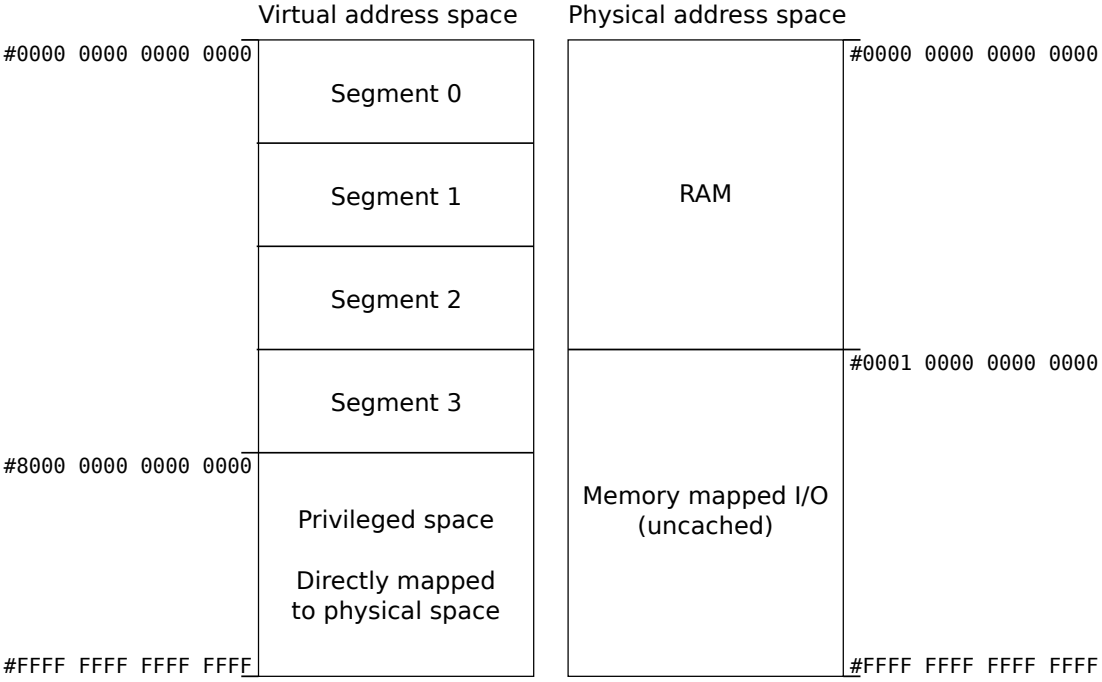


Figure 1: Virtual and physical address space layout [6, pg. 35]

The virtual address space is divided in user space and privileged space. The privileged space is directly mapped to the physical address space, called m. That means, $M[X] = m[X \wedge \#7FFF\,FFFF\,FFFF\,FFFF]$, if $X \geq 2^{63}$. The user space is divided into four segments, determined by the most significant 3 bits using 000_2, 001_2, 010_2 and 011_2 for segment 0, 1, 2 and 3, respectively. The use of these segments is not restricted by the hardware, but each segment is translated separately, as will be described in the next section.

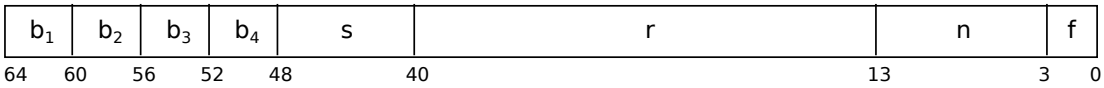
The first 256 terabyte of the physical space are used for RAM. The remaining space is reserved for I/O devices. The layout of the I/O space is implementation dependent, but MMIX defines that the I/O space is always uncached, regardless of whether the particular MMIX implementation uses caching or not. [6, pg. 35]

8.3.1 Address Translation

Besides the privileged space, which is directly mapped to the physical space, the user space is translated to the physical one via a quite complicated scheme. This section describes how this translation works in detail.

The Virtual Translation Register

The translation is defined by register `rV`, which has the following layout:



The first two bytes of `rV` specify the number of pages in the four segments. Segment i has at most $1024^{b_{i+1}-b_i}$ pages, where b_0 is defined to be zero. If $b_i = b_{i+1}$, segment i must have at most one page and if $b_i > b_{i+1}$, segment i must be empty. For example,

- if $b_1 = 1$, $b_2 = 2$, $b_3 = 3$ and $b_4 = 4$, all segments will have at most 1024 pages,
- if $b_1 = 3$, $b_2 = 2$, $b_3 = 1$ and $b_4 = 0$, segment 0 will have at most 1024^3 pages and all other segments will be empty and
- if $b_1 = 1$, $b_2 = 0$, $b_3 = 0$ and $b_4 = 0$, segment 0 will have at most 1024 pages, segment 1 will be empty and segments 2 and 3 will have both at most 1 page.

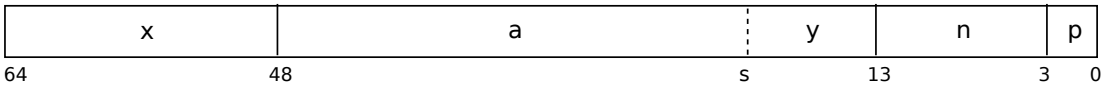
The next field, called s, specifies that the page size is 2^s, where s has to be at least 13 and at most 48. The field r tells MMIX the *root location*, which will be described in further detail shortly. The field n holds the *address space number* and last but not least, f is the *function field*, which specifies whether virtual address translation will be done by software ($f = 1$) or by hardware ($f = 0$). Other values are illegal. If translation by software is requested, MMIX will ignore b_1, b_2, b_3, b_4 and r of `rV` and let the software decide how the actual translation mechanism works. That means, the following structures and concepts only apply if hardware translation is used. [6, pg. 36]

The Root Location

The field r specifies an area in memory that holds the paging structures for the current virtual address space. For each segment i it holds $b_{i+1} - b_i$ page tables with either *page table entries* (PTEs) or *page table pointers* (PTPs), which are described in the next paragraphs. The page tables in the root location are, one could say, the "first layer" of the translation, because PTPs point to other page tables that reside in a different location in memory.

Page Table Entries

A PTE defines to which page in physical memory a page in virtual memory is mapped to. Additionally it specifies the access permissions for that page. A PTE looks like the following:

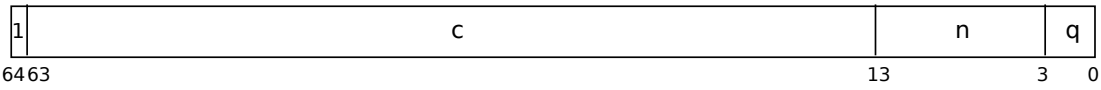


The field a holds the physical address divided by the page size, i. e. the physical address is $a * 2^s$. The field n is the address space number, which has to be equal to n in `rV`. The access permissions are defined by p with bit 0 for executing, bit 1 for writing and bit 2

for reading. That means, for example $p = 101_2$ makes the page readable and executable. The fields x and y are ignored by the hardware, which allows the operating system to use them for any purpose. [6, pg. 36] It is noteworthy, that y would be empty, if the page size were 2^{13}, and a would be empty, if the page size were 2^{48}. Additionally, the layout of a PTE clarifies the reason for the page size restrictions.

Page Table Pointers

PTPs are pointers to other page tables that may either hold PTPs as well or hold PTEs. They have the following layout:



Similarly to the field a of a PTE, the field c of a PTP specifies the address of the page table this PTP links to, divided by the page size. The field n has to match n in `rV` as well, while q is ignored and the most significant bit has to be 1. This forces the operating system to put a privileged address into a PTP, which is – as already mentioned – directly mapped. [6, pg. 36]

The Translation Process

Finally, the actual translation process should be described. If address A should be translated, the segment is determined by $i = \lfloor A/2^{61} \rfloor$. The page number in this segment is $A_p = \lfloor (A \wedge \texttt{\#1FFF FFFF FFFF FFFF})/2^s \rfloor$. Assuming that A_p is equal to $(a_4 a_3 a_2 a_1 a_0)_{1024}$ (in the number system with base 1024), the translation works as follows:

- if $a_4 = a_3 = a_2 = a_1 = 0$, the PTE e is $\mathrm{m}_8[2^{13}(r + b_i) + 8a_0]$ and thus, $\mathrm{M}[A]$ corresponds to $\mathrm{m}[2^s * e.a + (A \bmod 2^s)]$.
- if $a_4 = a_3 = a_2 = 0$, the auxiliary PTP p is used, i. e. $\mathrm{m}_8[2^{13}(r + b_i + 1) + 8a_1]$. In this case, the PTE is $\mathrm{m}_8[2^{13} * p.c + 8a_0]$. Thus, one level of indirection is involved.
- if $a_4 = a_3 = 0$, two levels of indirection are used. That means, the first PTP $p1$ is $\mathrm{m}_8[2^{13}(r + b_i + 2) + 8a_2]$ and determines the next PTP. This one, $p2$, is $\mathrm{m}_8[2^{13} * p1.c + 8a_1]$. And finally the PTE is $\mathrm{m}_8[2^{13} * p2.c + 8a_0]$.
- ...

[6, pg. 36] It is noteworthy, that when using the minimum page size of 2^{13}, four levels of indirection are sufficient to cover a whole segment. Because $2^{13} * 1024 * 1024^4 = 2^{63}$ covers even more than one segment. Additionally, it is worth mentioning that the first slot in PTP page tables in the root location is actually never used. Because this slot is covered by the previous page table in the root location.

Example

To clarify the just explained concepts and to show the layout in memory, which they imply, the following goes through an example. Supposed that `rV` and the paging structures in memory are filled as:

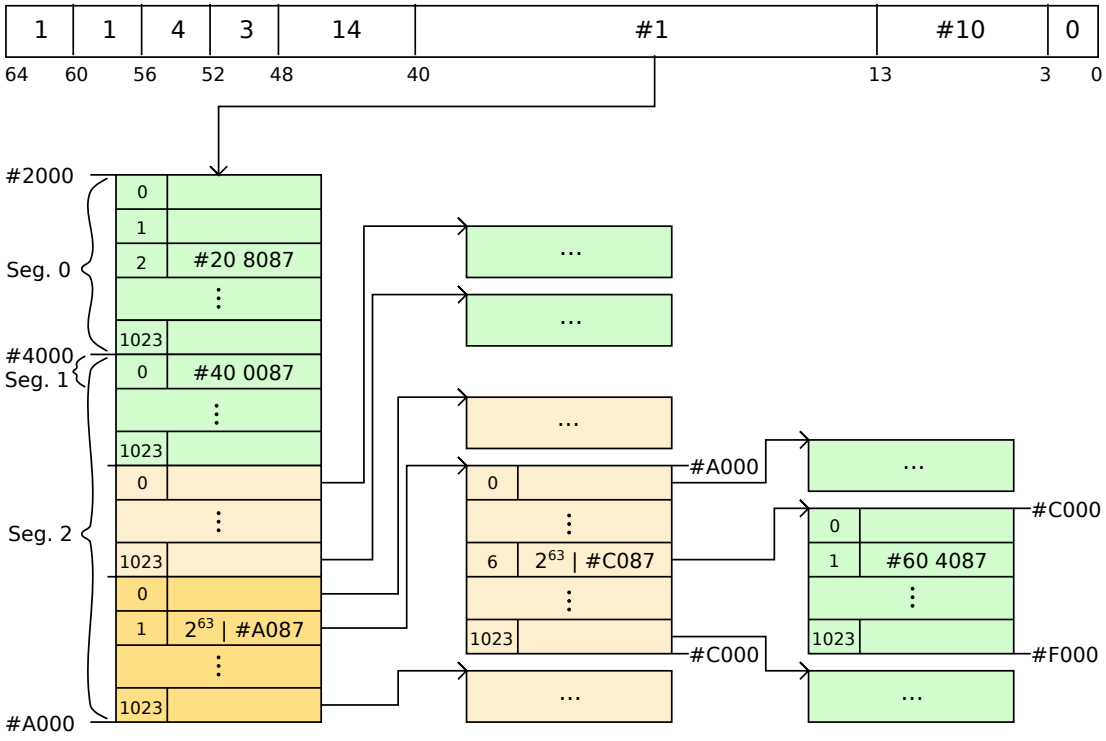


Figure 2: Paging example

Hence, segment 0 has 1024 pages, segment 1 has only one page, segment 2 has 1024^3 pages and segment 3 has no pages at all. Additionally, the page size is 2^{14}, the address space number is `#10` and the root location is `#2000` (`#1` $\ll$ 13). The root location is displayed on the left, while auxiliary page tables are on the right. Furthermore, the green cells contain PTEs, the light orange cells PTPs of level 1 and the orange cells PTPs of level 2. For demonstration purposes, the cells display the page table index on the left and some show the content as an octa on the right. All PTEs and PTPs end with `#87`, because they have to match the n of `rV` and have read, write and execute permissions (which actually has no special reason here).

Ignoring the special case that segment 1 and 2 overlap for a while, one can see that the number of page tables in the root location for segment i corresponds to $b_{i+1} - b_i$. The first page table does always contain PTEs, the second one PTPs of level 1 and so on. Each page table has 1024 slots and therefore it is 8192 bytes large. As the arrows show, PTPs do always point to another page table.

Since in this case b_1 is equal to b_2, segment 1 has only one page. An additional consequence of the interpretation of the segment sizes and the translation mechanism is, that – as the figure shows – the page in segment 1 is mapped by the first PTE of the page table responsible for segment 2. Thus, this PTE is used for the first page both in segment 1 and segment 2.

To demonstrate the translation process, the following shows a few examples:

- Virtual address `#80FF`:
  Obviously, `#80FF` belongs to segment 0 and the page number is $(00002)_{1024}$. As described, in this case the PTE is $\mathrm{m}_8[2^{13}(\texttt{\#1}+0) + 8 * 2] = \mathrm{m}_8[\texttt{\#2010}]$. The third slot of the first page table in segment 0 contains `#20 8087`, which means that the physical base address for that page is `#20 8000` and thus, the resulting physical

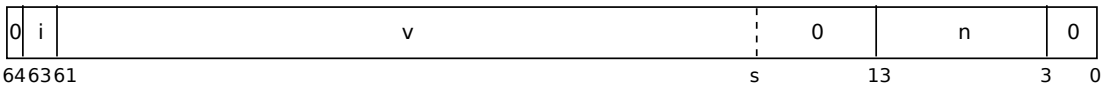
address #20 80FF.

- Virtual address #2000 0000 0000 1234:
  This address belongs to segment 1 and has the page number $(00000)_{1024}$. The PTE is $m_8[2^{13}(\#1{+}1) + 8 * 0] = m_8[\#4000] =$ #40 0087. Hence, the resulting physical address is #40 1234.

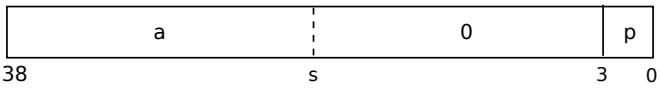
- Virtual address #4000 0004 0600 4000:
  Last but not least, this address belongs to segment 2 and has the page number $(00161)_{1024}$. Thus, the level 2 PTP is $m_8[2^{13}(\#1{+}3) + 8 * 1] = m_8[\#8008]$, which loads #8000 0000 0000 A087. Therefore it links to the PTP $m_8[\#A000{+}8 * 6]$, i. e. $m_8[\#A030]$, which in turn is #8000 0000 0000 C087. The last step loads the PTE $m_8[\#C000{+}8 * 1] = m_8[\#C008] =$ #60 4087, so that the final address is #60 4000.

8.3.2 Translation Caches

To prevent that every memory access requires this lengthy translation from virtual to physical addresses, MMIX uses a *translation cache*. This is also known as *translation lookaside buffer* (TLB). However, MMIX calls it translation cache or short TC. The exact behaviour or whether separate caches for instructions and data exist, is not enforced by the architecture. But MMIX defines that the TC contains *translation keys*, which are associated with *translations*. The translation key is basically the virtual address, whereas the translation is more or less the physical address. The key is structured as follows:



As usual, the field n holds the address space number. The field i is the segment number and v is the virtual address in that segment, divided by the page size. The other parts are defined to be zero. The layout of a translation is:



Similarly to a PTE, the field a holds the physical address, divided by the page size. The last three bits contain the protection bits, whereas the other bits are defined to be zero. [6, pg. 37]

Of course, the operating system needs a way to keep the TC up to date, when for example removing PTEs or changing their protection bits. Therefore, MMIX provides the instruction LDVTS.

Name:	LDVTS $X,$Y,$Z|Z
Effect:	$X ← $updateTC$($Y + $Z\|Z$)

The instruction *load virtual translation status* updates the translation cache for $Y+$Z|Z, which should have the form of a translation key, except that the least significant three bits need not be zero. $X will be set to 0 if the key is not in the TC, 1 if it is present for instructions, 2 if it is present for data and 3 if it is present for both. If this key is present in the TC, the protection bits will be replaced with ($Y + $Z|Z) ∧ #7. If these are zero, the key will be removed. [6, pg. 37]

8.3.3 Physical Memory Caches

As already mentioned, a particular implementation of MMIX may use caches for the RAM space of the physical memory. Which caches are present and how they are organized, is completely implementation dependent. But MMIX provides several instructions to allow a more efficient use of the caches and to keep them up to date. Similarly to the translation caches, MMIX has in mind that an implementation may provide separate caches for instructions and data.

Name:	`LDUNC $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow s(M_8[\$Y + \$Z|Z])$

The first instruction in this category is `LDUNC`, *load octa uncached*. It has the same behaviour as `LDO`, but tells MMIX "that the loaded octabyte (and its neighbors in a cache block) will probably not be read or written in the near future" [6, pg. 24].

Name:	`STUNC $X,$Y,$Z|Z`
Effect:	$M_8[\$Y + \$Z|Z] \leftarrow s(\$X)$

The instruction *store octa uncached* has the same meaning as `STO` and tells MMIX the same as `LDUNC` does. [6, pg. 24]

Name:	`PRELD|PREGO|PREST X,$Y,$Z|Z`
Effect:	-

These instructions have no (visible) effect, but inform MMIX that the $X + 1$ bytes $M_1[\$Y + \$Z|Z], \ldots, M_1[\$Y + \$Z|Z + X]$ will probably be loaded/stored, used as instructions or stored before loaded for `PRELD`, `PREGO` or `PREST`, respectively. That means, if `PRELD` is used, it might make sense to load these bytes into the data cache. If `PREGO` is used, MMIX might put these bytes into the instruction cache and if `PREST` is used, MMIX may ignore the current bytes in memory. Therefore, if these bytes are requested and are not yet in cache, MMIX does not need to load them from memory, because they will be written before they are read anyway. MMIX does also define, that no protection fault occurs for these instructions. [6, pg. 24]

Name:	`SYNCD X,$Y,$Z|Z`
Effect:	$caches[\$Y + \$Z|Z : X + 1] \rightarrow m[\$Y + \$Z|Z : X + 1]$ if in privileged mode: $caches[\$Y + \$Z|Z : X + 1] \leftarrow \varnothing$

The instruction *synchronize data* forces the hardware to make sure that all data for the $X+1$ bytes $M_1[\$Y + \$Z|Z], \ldots, M_1[\$Y + \$Z|Z + X]$ is present in memory (and not only in cache). If executed in the privileged space, it does additionally force MMIX to remove these bytes from the data cache. Again, no protection fault will occur if the memory is not accessible. [6, pg. 24]

Name:	`SYNCID X,$Y,$Z|Z`

Effect:	if in user mode:
	$IC[\$Y + \$Z|Z : X + 1] \leftrightarrow DC[\$Y + \$Z|Z : X + 1]$
	else:
	$caches[\$Y + \$Z|Z : X + 1] \leftarrow \varnothing$

When executed in user space, *synchronize instructions and data* forces the hardware to make sure that the $X+1$ bytes $M_1[\$Y + \$Z|Z], \ldots, M_1[\$Y + \$Z|Z + X]$ will be interpreted correctly when used as instructions. That means, MMIX should synchronize its data

cache with its instruction cache (e. g. because instructions might have been manually fabricated and might thus only be present in the data cache yet). When `SYNCID` is executed in privileged space, the hardware has to remove these bytes from all caches *without* writing it to memory. As with `SYNCD`, no protection faults can occur. [6, pg. 24,25]

Name:	`SYNC XYZ`
Effect:	if `XYZ` $= 0$: drain pipeline
	if `XYZ` $= 1$: drain stores
	if `XYZ` $= 2$: drain loads
	if `XYZ` $= 3$: drain loads and stores
	if `XYZ` $= 4$: go into power-saver mode
	if `XYZ` $= 5$: flush caches to memory
	if `XYZ` $= 6$: clear TCs
	if `XYZ` $= 7$: clear caches

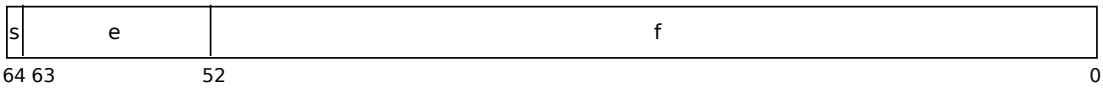
The last instruction in this category is *synchronize*, which is used for various purposes, whereas the 24-bit constant `XYZ` determines what action is performed. The first four actions drain the pipeline, in the sense that it stalls until all preceding instructions are finished (or all stores, loads, loads and stores for 1, 2, 3, respectively, are finished before the corresponding instructions after them). The fifth action tells MMIX to go into a power-saver mode, i. e. MMIX is allowed to execute instructions slower or not at all until some kind of signal arrives. The next one writes all cache content to main memory, while the last two simply remove all entries from the TCs or the instruction and data caches. Using `SYNC` with `XYZ` > 3 is allowed in privileged mode only. [6, pg. 25]

9 Floating Point Operations

Besides integer arithmetic, MMIX does also provide instructions to work with floating point numbers. The floating point arithmetic respects the IEEE/ANSI Standard 754. Since 64-bit quantities are the words of MMIX, arithmetic does always work with 64-bit floats, i. e. "doubles". But MMIX does also support some instructions to convert from 64-bit floats to 32-bit floats and the other way around.

9.1 Representation of Floating Point Numbers

A 64-bit floating point number has the following structure:



That means, it has a sign-bit s, an 11-bit exponent e and a 52-bit fraction f. Taking e as an unsigned integer and f as a fraction between 0 and $(.111\ldots1)_2 = 1 - 2^{-52}$, an octabyte has the following significance:

$$\begin{aligned}
\pm 0.0, &\quad \text{if } e = f = 0 \text{ (zero);}\\
\pm 2^{e-1023}(1+f), &\quad \text{if } 0 < e < 2047 \text{ (normal);}\\
\pm 2^{-1022} f, &\quad \text{if } e = 0 \text{ and } f > 0 \text{ (subnormal);}\\
\pm \infty, &\quad \text{if } e = 2047 \text{ and } f = 0 \text{ (infinite);}\\
\pm \mathrm{NaN}\,(f), &\quad \text{if } e = 2047 \text{ and } 0 < f < 1/2 \text{ (signaling NaN);}\\
\pm \mathrm{NaN}\,(f), &\quad \text{if } e = 2047 \text{ and } f \geq 1/2 \text{ (quiet NaN).}
\end{aligned}$$

As shown, there are two representations for zero - a positive and a negative one. There are *normal* numbers, that can range from approximately $\pm 10^{-308}$ to $\pm 10^{308}$. Additionally *subnormal* numbers (also called *denormal* or *denormalized* numbers) are supported, which can range from approximately $\pm 10^{-324}$ to $\pm 10^{-308}$, but have fewer bits of precision. If the exponent has the maximum value, i. e. 2047, it encodes infinity or *not a number* (NaN). The latter distinguishes between *signaling* and *quiet* NaN. Signaling NaNs raise an invalid AE when they are used, while quiet NaNs will not. [6, pg. 15]

Furthermore, the standard defines that four rounding modes should be available: round to nearest (and to even in case of ties, i. e. the least significant bit should be zero), round off (toward zero), round up (toward $+\infty$) and round down (toward $-\infty$). MMIX uses the special register `rA` to specify the rounding mode. Additionally, all instructions having only one operand, specify the rounding mode with operand `Y` using `Y` = 0 for the round mode in `rA`, `Y` = 1 for round off, `Y` = 2 for round up, `Y` = 3 for round down and `Y` = 4 for round near. [6, pg. 15 and 21]

Last but not least, there are five kinds of arithmetic exceptions, that can occur when working with floating point numbers:

1. Floating overflow (value too large to be representable),
2. Floating underflow (value too small to be representable),
3. Floating divide by zero,
4. Floating inexact (exact result not representable) and
5. Floating invalid (square root of negative number, using signaling NaN, ...).

As already said, all of these will either raise an AE or set the corresponding *event bit* in `rA`, depending on whether the corresponding *enable bit* in `rA` is set or not. [6, pg. 15]

9.2 Arithmetic

The first category of floating point operations are the arithmetic instructions. The floating point instructions don't have an immediate version, because it does not make much sense to specify a float with a single byte. Additionally, this section uses the notation $f(\dots)$ to indicate that a value is interpreted as a 64-bit floating point number.

Name:	`FADD|FSUB $X,$Y,$Z`
Effect:	$\$X \leftarrow f(\$Y) + \mid - f(\$Z)$

`FADD` computes the sum of `$Y` and `$Z`, treating them as floating point numbers and puts the result in `$X`. `FSUB` performs the same operation, but switches the sign of `$Z` first, if `$Z` is not NaN. If the sum of $(+\infty) + (-\infty)$ or $(-\infty) + (+\infty)$ is computed, an invalid AE will be raised. [6, pg. 16]

Name:	`FMUL|FDIV $X,$Y,$Z`
Effect:	$\$X \leftarrow f(\$Y) * \mid / f(\$Z)$

These instructions multiply or divide the floating point numbers `$Y` and `$Z`. Several cases result in an invalid AE, like $(\pm 0.0) * (\pm\infty)$, $(\pm 0.0)/(\pm 0.0)$ or $(\pm\infty)/(\pm\infty)$. Of course, dividing by (± 0.0) raises a floating divide by zero AE. [6, pg. 16]

Name:	`FREM $X,$Y,$Z`
Effect:	$\$X \leftarrow f(\$Y) \bmod f(\$Z)$

The *floating remainder* instruction computes the remainder and puts it into `$X`. This is defined "to be $\$Y - n * \$Z$, where n is the nearest integer to $\$Y/\$Z$, and n is an even integer in case of ties" [6, pg. 16]. If `$Y` is infinite and/or `$Z` is zero, an invalid AE will be raised. [6, pg. 16]

Name:	`FSQRT $X,Y,$Z`
Effect:	$\$X \leftarrow \sqrt{f(\$Z)}$, using round-mode `Y`

The last one in this family is *floating square root*. It puts the square root of `$Z` into `$X`. An invalid AE will be raised if `$Z` is negative, except for -0.0. [6, pg. 17]

9.3 Comparison

Of course, besides doing arithmetic, one has to be able to compare floating point numbers. This does not work well using the integer comparison instructions, because one would have to take care of negative numbers, $+0.0$, -0.0 and NaN manually. Therefore, the floating point comparisons simplify that task.

Name:	`FCMP $X,$Y,$Z`
Effect:	$\$X \leftarrow (f(\$Y) > f(\$Z)) - (f(\$Y) < f(\$Z))$

As the effect description shows, `FCMP` is basically the same as `CMP`, but treats the operands as floating point numbers. Thus, `$X` will be set to -1, if `$Y` is less than `$Z`, 0 if `$Y` is equal to `$Z` and 1 if `$Y` is greater than `$Z`. It will raise an invalid AE and set `$X` to zero, if `$Y` or `$Z` is NaN . [6, pg. 17]

Name:	`FEQL $X,$Y,$Z`
Effect:	$\$X \leftarrow (f(\$Y) = f(\$Z))\ ?\ 1\ :\ 0$

Floating equal to sets `$X` to 1, if `$Y` and `$Z` are equal. But it is noteworthy, that NaN is not equal to anything and -0.0 is equal to $+0.0$. [6, pg. 17]

Name:	`FUN $X,$Y,$Z`
Effect:	$\$X \leftarrow (f(\$Y) = \text{NaN} \vee f(\$Z) = \text{NaN})\ ?\ 1\ :\ 0$

The last comparison instruction is *floating unordered* and sets `$X` to 1, if `$Y` and `$Z` are considered *unordered*, i. e. at least one of them is NaN . [6, pg. 17]

9.4 Neighborhood Comparison

Because of the limited precision of floating point numbers, operations might produce inexact results. The larger the numbers, the larger the potential difference of the produced result to the exact result. For that reason, an absolute comparison of floats, as the last section described, is not always desired. Therefore, MMIX offers another category of instructions that allow comparisons with respect to an *epsilon* and depending on the magnitude of the floating point numbers in question.

At first, MMIX defines a so called *neighborhood* of a number. Assuming that epsilon is a float called ϵ, the float u with fraction f and exponent e has the neighborhood:

$$N_\epsilon(u) = \begin{cases} \{x \mid |x-u| \le 2^{e-1022}\epsilon\} & \text{if } u \text{ is normal} \\ \{x \mid |x-u| \le 2^{-1021}\epsilon\} & \text{if } u \text{ is subnormal} \\ \{0\} & \text{if } u \text{ is zero} \\ \{\pm\infty\} & \text{if } u \text{ is } \pm\infty \text{ and } \epsilon < 1 \\ \{\text{everything except } \mp\infty\} & \text{if } u \text{ is } \pm\infty \text{ and } 1 \le \epsilon < 2 \text{ and} \\ \{\text{everything}\} & \text{if } u \text{ is } \pm\infty \text{ and } \epsilon \ge 2. \end{cases}$$

[6, pg. 19] Without going into the details of this definition, it basically means that the neighborhood of a normal float depends on its exponent. That is, the larger the exponent, the larger the neighborhood.

Displayed graphically (and a bit exaggerated for demonstration purposes), the neighborhoods of a few numbers might look like the following:

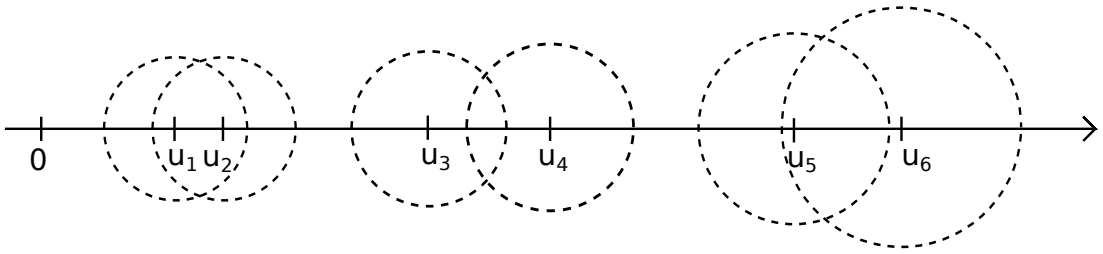


Figure 3: Example neighborhoods, demonstrating float relationships

MMIX distinguishes four cases when comparing floats u and v with respect to ϵ:

1. $u \approx v$, if $u \in N_\epsilon(v)$ and $v \in N_\epsilon(u)$.
   That means, two numbers will be considered *equivalent*, if both are in the neighborhood of the corresponding other number. In the example, only $u_1 \approx u_2$, because $u_1 \in N_\epsilon(u_2)$ and $u_2 \in N_\epsilon(u_1)$.

2. $u \sim v$, if $u \in N_\epsilon(v)$ or $v \in N_\epsilon(u)$.
   Thus, two numbers will be considered *similar*, if only one of them belongs to the neighborhood of the corresponding other number. For example, $u_5 \sim u_6$, because $u_5 \in N_\epsilon(u_6)$, but $u_6 \notin N_\epsilon(u_5)$.

3. $u \prec v$, if $u < N_\epsilon(v)$ and $N_\epsilon(u) < v$.
   For example, $u_3 \prec u_4$ because u_3 is less than all numbers in $N_\epsilon(u_4)$ and all numbers in $N_\epsilon(u_3)$ are less than u_4.

4. $u \succ v$, if $u > N_\epsilon(v)$ and $N_\epsilon(u) > v$.
   Analogous, $u_4 \succ u_3$.

[6, pg. 19] The following instructions are based on this definition and use the special *epsilon register* `rE` for ϵ.

Name:	`FCMPE $X,$Y,$Z`
Effect:	$\$X \leftarrow (f(\$Y) \succ f(\$Z)\ (\texttt{rE})) - (f(\$Y) \prec f(\$Z)\ (\texttt{rE}))$

Analogous to `FCMP`, `FCMPE` – called *floating compare with respect to epsilon* – compares `$Y` with `$Z` according to the definition above and sets `$X` to -1, 0 or 1. It should be noted, that `$X` will be set to zero, if `$Y` is similar *or* equivalent to `$Z`. An invalid AE will be raised, if `$Y`, `$Z` or `rE` is NaN or `rE` is negative. [6, pg. 19]

Name:	`FEQLE $X,$Y,$Z`
Effect:	$\$X \leftarrow (f(\$Y) \approx f(\$Z)\ (\texttt{rE}))\ ?\ 1\ :\ 0$

Similarly to `FEQL`, `FEQLE` will set `$X` to 1, if `$Y` is equivalent to `$Z`, depending on `rE`. It raises the same arithmetic exceptions as `FCMPE`. [6, pg. 19]

Name:	`FUNE $X,$Y,$Z`
Effect:	$\$X \leftarrow (f(\$Y) = \text{NaN} \vee f(\$Z) = \text{NaN} \vee \texttt{rE} = \text{NaN} \vee \texttt{rE} < 0)\ ?\ 1\ :\ 0$

The last one in this group is `FUNE`, which will set `$X` to 1, if `$Y`, `$Z` or `rE` are exceptional as described for `FCMPE` and `FEQLE`. [6, pg. 19]

9.5 Conversion between Float and Integer

MMIX offers three groups of instructions to convert an integer to a floating point number and the other way around.

Name:	`FIX|FIXU $X,Y,$Z`
Effect:	$\$X \leftarrow (int)f(\$Z) \bmod 2^{64}$, using round-mode `Y`

The instructions *convert floating to fixed* and *convert floating to fixed unsigned* take `$Z` as a float, convert it to an integer and put it into `$X`. Only when using `FIX`, an invalid AE will be raised if `$Z` is infinite or NaN and a float-to-fix AE will occur, if the result is less than -2^{63} or greater than $2^{63} - 1$. [6, pg. 20]

Name:	`FINT $X,Y,$Z`
Effect:	$\$X \leftarrow f((int)f(\$Z))$, using round-mode `Y`

The instruction *floating integer* rounds the float `$Z` to a floating integer and places it in `$X`. Infinity and NaN are not changed. The difference to `FIX` is, that `FINT` writes a floating point number to `$X`, while `FIX` writes a signed integer to `$X`. [6, pg. 17]

Name:	`FLOT|FLOTU $X,Y,$Z|Z`
Effect:	$\$X \leftarrow f(s(\$Z)\|Z)$, using round-mode `Y`

Finally, the instructions *convert fixed to floating* and *convert fixed to floating unsigned* treat `$Z|Z` as an integer and convert it to the nearest floating point number. Only if using `FLOT`, an floating inexact AE will be raised, if rounding is necessary. [6, pg. 20]

9.6 Short Floats

Although MMIX is a 64-bit architecture and thus, works with 64-bit floating point values by default, it does also provide some instructions to use 32-bit floating point numbers, called *short floats*. But MMIX has no separate arithmetic or comparison instructions for them. Instead it offers instructions to load a short float from memory into a float and store a float as a short float to memory.

Name:	`LDSF $X,$Y,$Z|Z`
Effect:	$\$X \leftarrow f(sf(\mathrm{M}_4[\$Y + \$Z\|Z]))$

The first one, called *load short float*, loads the tetra $\mathrm{M}_4[\$Y + \$Z|Z]$, treating it as a 32-bit float, converts it to a 64-bit float and puts it into `$X`. [6, pg. 20]

Name:	`STSF $X,$Y,$Z|Z`
Effect:	$\mathrm{M}_4[\$Y + \$Z|Z] \leftarrow sf(f(\$X))$

`STSF` goes the other way: it treats `$X` as a 64-bit float, converts it to a 32-bit float and stores that into $\mathrm{M}_4[\$Y + \$Z|Z]$. It may trigger a floating overflow, underflow, inexact and invalid AE. [6, pg. 20]

Name:	`SFLOT|SFLOTU $X,Y,$Z|Z`
Effect:	$\$X \leftarrow f(sf(s(\$Z)\|Z))$, using round-mode `Y`

These instructions behave like `FLOT` and `FLOTU`, but convert $s(\$Z)|Z$ to a 32-bit float first, which ensures that no AE will be raised if `$X` is stored with `STSF` afterwards. [6, pg. 20]

10 Register Stack

Actually, the local registers in MMIX are more complicated than explained at the beginning. Because MMIX uses a combination of registers and memory for the stack. This way, subroutine linkage is realized. Additionally, MMIX offers instructions to save or restore the complete state of a running program using the stack. Both are explained in detail in this section.

10.1 Subroutine Linkage

Because of the complexity of the subroutine linkage mechanism, it is explained in two steps. At first, it is shown from the perspective of the programmer. Afterwards the internal functional principle is described.

10.1.1 Programmers View

The programmer has `rG` local registers named `$0`, ..., `$(rG - 1)` at his hand. The registers `$0`, ..., `$(rL - 1)` are the currently used ones, while `$(rL)`, ..., `$(rG - 1)` are the marginal registers. Furthermore he can think of the stack as an potentially unbounded list S. The stack pointer, i. e. the pointer that indicates the slot in S that is going to be written next, is called τ, which is initially zero. [6, pg. 22]

Calling and Returning

MMIX provides two instruction families to call subroutines and return from them.

Name:	`PUSHJ $X,@+4*(YZ[-`2^{16}`]),  PUSHGO $X,$Y,$Z|Z`
Effect:	$S[\tau] \leftarrow \$0, S[\tau+1] \leftarrow \$1, \ldots, S[\tau + \mathtt{X} - 1] \leftarrow \$(\mathtt{X}-1)$
	$S[\tau + \mathtt{X}] \leftarrow \mathtt{X}$
	$\tau \leftarrow \tau + \mathtt{X} + 1$
	$\$0 \leftarrow \$\mathtt{X}+1, \$1 \leftarrow \$\mathtt{X}+2, \ldots, \$(\mathtt{rL}-\mathtt{X}-2) \leftarrow \$(\mathtt{rL}-1)$
	$\mathtt{rL} \leftarrow \mathtt{rL} - \mathtt{X} - 1$
	$\mathtt{rJ} \leftarrow @ + 4$
	$@ \leftarrow (@ + 4 * (\mathtt{YZ}[-2^{16}])) \mid (\$\mathtt{Y} + \$\mathtt{Z}\|\mathtt{Z})$

At first, `PUSHJ` (*push registers and jump*) and `PUSHGO` (*push registers and go*) are essentially the same, except that MMIX determines the new value of the instruction pointer in different ways. The first action they perform is to push the current local registers `$0`, ..., `$(X - 1)` onto the stack S. Afterwards the number of registers, that have been *pushed down*, is saved in $S[\tau + \mathtt{X}]$ and τ is increased correspondingly. The next step is to rename the current registers `$(X + 1)`, ..., `$(rL - 1)` to `$0`, ..., `$(rL - X - 2)`. That means, all used registers above `X` are passed as arguments to the subroutine, where they appear as `$0`, `$1` and so on. Finally, `rL` is adjusted, so that only the arguments are currently in use, the *return-jump register* `rJ` is set to the instruction that would have been executed normally and the instruction pointer is changed. [6, pg. 22]

Name:	`POP X,YZ`
Effect:	$x \leftarrow S[\tau - 1] \bmod 256$
	$S[\tau - 1] \leftarrow \$(\texttt{X} - 1)$
	$\texttt{rL} \leftarrow min(x + \texttt{X}, \texttt{rG})$
	$\$(\texttt{rL} - 1) \leftarrow \$(\texttt{rL} - \texttt{x} - 2), \ldots \$(\texttt{x} + 1) \leftarrow \$0$
	$\$\texttt{x} \leftarrow S[\tau - 1], \$(\texttt{x} - 1) \leftarrow S[\tau - 2], \ldots, \$0 \leftarrow S[\tau - x - 1]$
	$\tau \leftarrow \tau - x - 1$
	$@ \leftarrow \texttt{rJ} + 4 * \texttt{YZ}$

Of course, `POP` (*pop registers and return from subroutine*) basically behaves in the opposite way as `PUSHJ` and `PUSHGO` do. At first, the number of registers that have been pushed down by the associated `PUSHJ` or `PUSHGO` are loaded from $S[\tau - 1]$. It can not be more than 255, because MMIX has only 256 dynamic registers. In the next step, MMIX sets $S[\tau - 1]$ to the "main return value" `$(X - 1)`, which is used later. Next, `rL` is adjusted to be the number of local registers the caller wanted to keep plus the number of return values, denoted by `X`. Of course, that should not be more than `rG`. Subsequently, the registers holding the return values (except the main return value) are renamed, so that they appear in `$(x + 1)`, ..., `$(rL - 1)` for the caller. Finally, the previously saved values are restored from the stack, τ is adjusted correspondingly and MMIX jumps back to the location stored in `rJ`. Optionally, some instructions can be skipped with `YZ` > 0. It is noteworthy that the main return value appears in the so called *hole* `$x`, i. e. the register that has stored the number of registers that have been pushed down. [6, pg. 22,23]

Example

To make the just described instructions more clear, the following goes through an example. It supposes, that the first four local registers have some values and a `PUSHJ $1,Sub` is executed. The following figure illustrates the state before the `PUSHJ` and the state afterwards, i. e. the initial state in subroutine *Sub*:

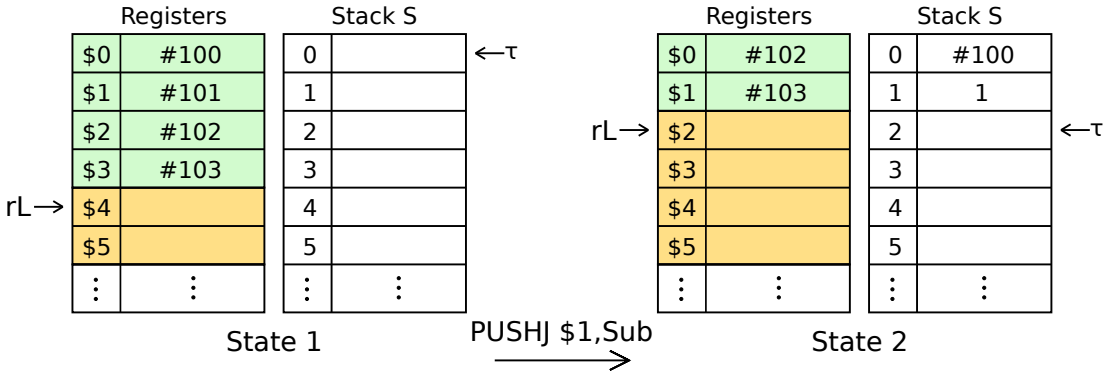


Figure 4: Register stack: User perspective, state 1 to 2

The figure displays the used local registers green and the marginal ones orange. At first `$0` and `$1` are saved on the stack, whereas `$1` has been set to the number of pushed down values. Afterwards `$2` and `$3` are pushed as arguments to *Sub*, appearing as `$0` and `$1`. Thus, `rL` is 2 and τ is 2 as well.

In the next step of the example *Sub* performs some calculations, leading to state 3, and executes a `POP 4,0` to return to the caller, whose state is shown on the right:

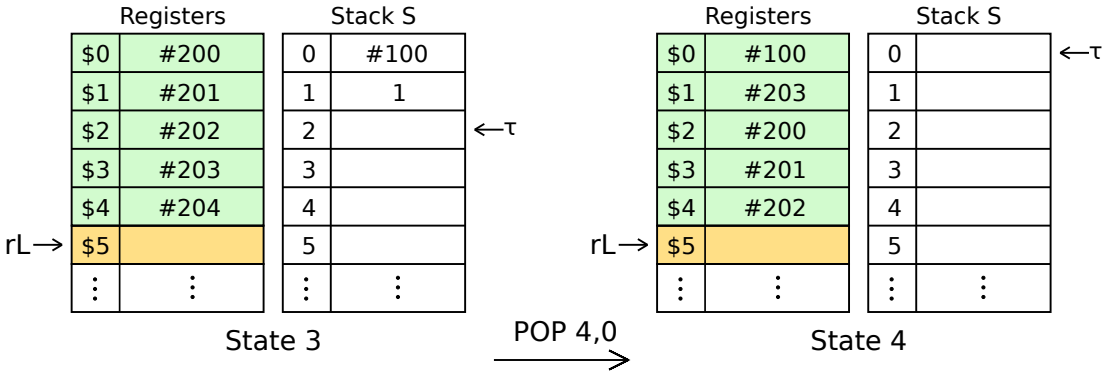


Figure 5: Register stack: User perspective, state 3 to 4

As the figure shows, `$0`, ..., `$4` have been set to some arbitrary values. The subsequent `POP 4,0` at first loads the number of pushed down registers from the stack, i. e. $S[\tau-1] = 1$. Based on that, it restores `#100` from $S[0]$ into `$0`, sets the hole (`$1`) to the last return value `#203` and `$2`, ..., `$4` to the other return values in the original order. Thus, `rL` is $1 + \texttt{X} = 5$ and τ is zero again.

At this point it might look strange that the last return value appears first for the caller, followed by the other ones in the same order they were in the registers of *Sub*. The section about the internal view will show the reason for that behaviour.

Special Cases

Unfortunatly it is even more complicated as just described, because some special cases have been suppressed. The instructions `PUSHJ|PUSHGO $X,...` have the following special cases:

- If $\texttt{rL} \leq \texttt{X} < \texttt{rG}$, the value of `rL` will be increased to $\texttt{X} + 1$ first.
- If $\texttt{X} \geq \texttt{rG}$, all local registers `$0`, ..., `$(rL - 1)` will be saved, followed by `rL` and `rL` will be reset to zero.

On the other hand, `POP X,YZ` has to take care of:

- If $\texttt{X} > \texttt{rL}$, `X` will be replaced by $\texttt{rL} + 1$ first and the hole will be set to zero.
- If $\texttt{X} = 0$, the hole will disappear, i. e. it will become marginal.

[6, pg. 22,23]

10.1.2 Internal View

After having described the procedure of calling subroutines and returning from them from the perspective of the programmer, it should be explained how MMIX does actually achieve that.

As mentioned previously, MMIX has a local register array l with either 256, 512 or 1024 slots. It is used as a ring, which means that if we have the local registers $l[0]$, $l[1]$, ..., $l[255]$, register $l[256]$ will be the same as $l[0]$, $l[257]$ the same as $l[1]$ and so on.

To realize the register stack, MMIX has to manage the registers and the stack in memory. To do so, the register ring is divided into three parts by α, β and γ. $l[\alpha]$, $l[\alpha+1]$, ..., $l[\beta-1]$ are corresponding to `$0`, `$1`, ..., `$(rL - 1)`. The registers $l[\beta]$, ...,

$l[\gamma - 1]$ are currently unused and $l[\gamma], \ldots, l[\alpha - 1]$ are the registers that are currently not accessible, called *hidden*, but have not yet been stored to memory. The situation on the stack in memory is described by the special registers `rO` and `rS`. The former is the offset of `$0` in memory, i. e. the location it would be written to. The latter is the location the next register is going to be written to. α, β and γ relate to `rO` and `rS` in the following way (when 2^n is the number of slots in l):

$$\alpha = (\mathrm{rO}/8) \bmod 2^{\mathrm{n}}, \qquad \beta = (\alpha + \mathrm{rL}) \bmod 2^{\mathrm{n}}, \qquad \text{and} \qquad \gamma = (\mathrm{rS}/8) \bmod 2^{\mathrm{n}}$$

To make sure that no value is lost, MMIX has to take care that α, β and γ never move past each other. Whenever a `PUSHJ|PUSHGO` is done, α moves towards β. Setting `$X` with `X` $\geq$ `rL` means that `rL` is increased, i. e. β is moved towards γ. If β moved past γ, registers would have to be written to memory first to free as many slots as required. To do so, $l[\gamma]$ is written to M_8[`rS`] and γ is increased by one and thus `rS` by 8, until the new β is less than γ. When doing a `POP`, α moves backwards to γ. If this moved α past γ, we would have to load values back from memory first. That means, γ and `rS` are decreased and M_8[`rS`] is put into $l[\gamma]$ until the new α is less than γ. [6, pg. 33]

The following figures illustrate the actions with a similar example as in the previous section. For simplicity it is assumed that only 4 local registers are present[3]:

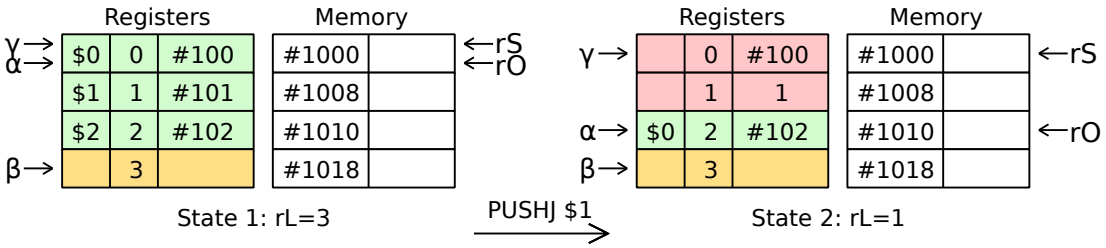


Figure 6: Register stack: Internal perspective, state 1 to 2

The green and yellow cells have the same meaning as in figures 4 and 5, while the red cells are the hidden registers. The register block on the left shows the current assignment of dynamic registers, the local register index and the value in the register. The right block displays the stack in memory with the address and the value. Additionally the values of α, β, γ, `rO` and `rS` are indicated by pointing to the corresponding register or memory slot.

In the first state, three registers are used, α and γ are zero, β is $\alpha +$ `rL` $= 3$ and `rO` and `rS` have the value `#1000`. State 2 is reached by doing a `PUSHJ $1`. Hence, the offset α in the register ring is increased by 2 and the offset `rO` in memory is increased by 16. Additionally, the hole ($l[1]$) is set to the number of registers that have been pushed down.

[3]This is not specification conform, as explained previously, but simplifies the example.

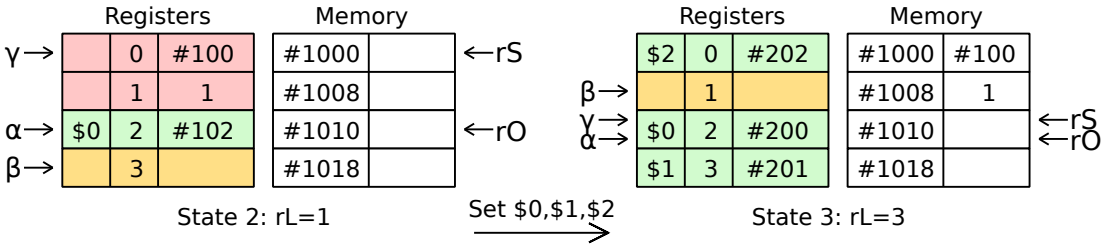


Figure 7: Register stack: Internal perspective, state 2 to 3

The third state, shown in figure 7, is reached by setting `$0`, `$1` and `$2` to `#200`, `#201` and `#202`, respectively. Setting `$0` does not increase `rL`, so that only $l[2]$ is changed. The other two both increase `rL` by one, i. e. move β towards γ. Thus, in each case γ and `rS` are increased and a value is written to memory.

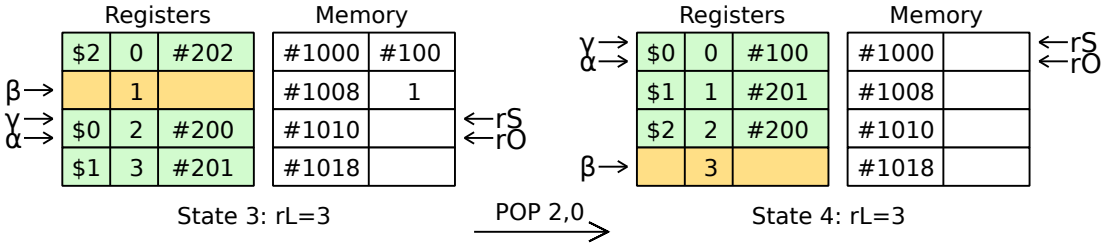


Figure 8: Register stack: Internal perspective, state 3 to 4

The last transition, illustrated in figure 8, is achieved by doing a `POP 2,0`. As described, `POP` moves α back. The hole tells MMIX how many registers have been pushed down by the preceding `PUSHJ` or `PUSHGO`. In this case, the hole has been written to memory, so that it has to be loaded back first. Afterwards MMIX has to move α two steps up because one register has been saved and the hole has been created. To be able to decrease α and `rO`, one further value has to be loaded from memory. To return the two values, a move of $l[3]$ to $l[1]$ is sufficient because the other value is already in the desired slot (as all other return values would be, if there were more than two). This is the reason for the – at a first glance – strange order of the return values.

10.2 Saving and Restoring the State

Another concept, that works with the register stack as well, is the procedure of storing and restoring the state of the running program. It is intended both for the operating system and user applications. The former may use it for example to implement process switching or to save the state when handling an interrupt. The latter can use it to implement thread switching in user space, for example.

Name:	`SAVE $X`
Effect:	$S[\tau{+}{+}] \leftarrow \$0, \ldots, S[\tau{+}{+}] \leftarrow \$(\texttt{rL}-1)$
	$S[\tau{+}{+}] \leftarrow \texttt{rL}, \quad \texttt{rL} \leftarrow 0$
	$S[\tau{+}{+}] \leftarrow g[\texttt{rG}], \ldots, S[\tau{+}{+}] \leftarrow g[255]$
	for $s \in \{\texttt{rB}, \texttt{rD}, \texttt{rE}, \texttt{rH}, \texttt{rJ}, \texttt{rM}, \texttt{rR}, \texttt{rP}, \texttt{rW}, \texttt{rX}, \texttt{rY}, \texttt{rZ}, (\texttt{rG} \ll 56) \vee \texttt{rA}\}$
	$\quad S[\tau{+}{+}] \leftarrow s$
	$\$\texttt{X} \leftarrow \tau$

Put simply, SAVE (*save process state*) stores all registers that might affect the computation on the stack and writes the address of the topmost octabyte on the stack into $X. In detail, it means that at first all hidden registers are written to memory (which is not listed in the effect description, because it shows it from the programmer perspective). Afterwards all used local registers are written behind them, followed by rL and rL is set to zero. In the next step, all global registers are written to memory, followed by the special registers that affect the computation, whose last value is an octabyte containing rG in the most significant byte and rA in the least significant ones. As will be described shortly, the value of $X after executing SAVE $X can be used for UNSAVE. [6, pg. 34]

Name:	UNSAVE $Z
Effect:	$\tau \leftarrow \$\mathtt{Z}$
	for $s \in \{(\mathtt{rG} \ll 56) \vee \mathtt{rA}, \mathtt{rZ}, \mathtt{rY}, \mathtt{rX}, \mathtt{rW}, \mathtt{rP}, \mathtt{rR}, \mathtt{rM}, \mathtt{rJ}, \mathtt{rH}, \mathtt{rE}, \mathtt{rD}, \mathtt{rB}\}$
	$\quad s \leftarrow S[--\tau]$
	$g[255] \leftarrow S[--\tau], \ldots, g[\mathtt{rG}] \leftarrow S[--\tau]$
	$\mathtt{rL} \leftarrow S[--\tau]$
	$\$(\mathtt{rL} - 1) \leftarrow S[--\tau], \ldots, \$0 \leftarrow S[--\tau]$

Consequently, UNSAVE (*restore process state*) goes the other way. It first sets the stack pointer to $Z and restores all special registers in the opposite order from the stack, followed by the global ones. Afterwards rL is loaded, that tells MMIX the number of local registers to restore from stack, which is done in the last step. [6, pg. 34] It should be noted, that the hidden registers, that had been stored on the stack during SAVE before the used local ones, are *not* restored by UNSAVE. This is done by the next POP, i. e. as soon as they are needed.

11 Interrupts and Exceptions

As already mentioned a few times in this thesis, MMIX does of course have a concept for interrupts and exceptions as well. It distinguishes between four different kinds: *Forced trips*, *dynamic trips*, *forced traps* and *dynamic traps*.[4] The first two are simply called *trips*, while the other two are called *traps*. The main difference is, that trips are handled by the user application, while traps are handled by the operating system.

11.1 Triggering of Trips and Traps

At first, the procedure of triggering a trip or trap is described. The forced trips and traps are requested explicitly by an instruction, whereas dynamic trips and traps are either raised because an exceptional condition occurred (synchronous) or an interrupt occurred (asynchronous).

11.1.1 Triggering Trips

All trips make use of the special registers rB, rW, rX, rY and rZ. Register rB is called *bootstrap register* and is used to save $255. The *where interrupted register* rW indicates the location the interruption occurred at, *execution register* rX holds the 4 instruction bytes and some other information. The registers rY and rZ (*Y operand* and *Z operand*) are used to pass operands to the trip handler. [6, pg. 28]

[4] Actually, the MMIX specification speaks of three kinds, because forced trips and dynamic trips are put together. [6, pg. 28] But taking it as a whole, forced trips and dynamic trips conceptually differ in the same way as forced traps and dynamic traps do. Therefore, this thesis speaks of four kinds.

A forced trip can be triggered with the following instruction:

Name:	`TRIP X,Y,Z`
Effect:	`rX` $\leftarrow 2^{63} \vee M_4[@]$
	`rW` $\leftarrow @ + 4$
	`rY` $\leftarrow$ `$Y`, `rZ` $\leftarrow$ `$Z`
	`rB` $\leftarrow$ `$255`, `$255` $\leftarrow$ `rJ`
	$@ \leftarrow 0$

As the effect description shows, `TRIP` puts various information into the special registers that can be used by the handler. The `X` operand of the instruction is not used by the instruction itself, but since its bytes are put into `rX`, the handler may utilize it. Forced trips are always handled at address 0. [6, pg. 28] The meaning of 2^{63} in `rX` and why `rJ` is saved, will be explained when the handling of trips and traps is described.

Dynamic trips are raised for arithmetic exceptions, which are controlled by `rA`. The only differences to forced trips are the location they are handled at and that `rY` and `rZ` will be set to the decoded operands of the instruction that caused the AE. Each arithmetic exception has its own location:

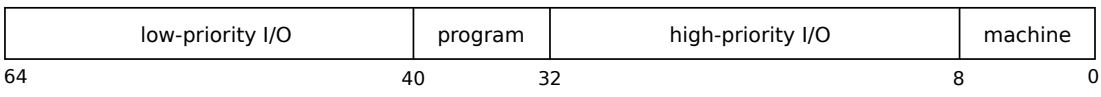
`#10` :	Integer divide check (D)
`#20` :	Integer overflow (V)
`#30` :	Float-to-fix overflow (W)
`#40` :	Invalid operation (I)
`#50` :	Floating overflow (O)
`#60` :	Floating underflow (U)
`#70` :	Floating division by zero (Z)
`#80` :	Floating inexact (X)

Forced and dynamic trips are triggered in user mode only, i. e. they are ignored in privileged mode. [6, pg. 28]

11.1.2 Triggering Traps

Similarly to trips, traps use the special registers `rBB`, `rWW`, `rXX`, `rYY` and `rZZ`, with the same purposes as their trip correspondences. The reason for the separate registers is, that a trap may of course be triggered while a trip is handled. Additionally, register `rT` specifies the location at which forced traps are handled, whereas `rTT` specifies the location for dynamic traps. [6, pg. 28,29]

MMIX uses the special *interrupt mask register* `rK` to control which dynamic traps are enabled. As soon as a bit in the special *interrupt request register* `rQ` is 1 and the corresponding bit in `rK` is 1 as well, a dynamic trap is triggered. These registers have the following layout:



In general, the bits on the right have a higher priority than the bits on the left. Therefore, high-speed devices like network cards should get bits on the right, while slow devices like terminals should get bits on the left. But MMIX does not define the meanings of the I/O bits. Only the program bits (PEs) and some of the machine bits (MEs)

are specified. The program bits are called `rwxnkbsp` with the following meanings:

- `r` bit: instruction tries to load from a page without read permission;
- `w` bit: instruction tries to store to a page without write permission;
- `x` bit: instruction appears in a page without execute permission;
- `n` bit: instruction refers to a privileged (negative) address;
- `k` bit: instruction is privileged, for use by the kernel only;
- `b` bit: instruction breaks the rules of MMIX;
- `s` bit: instruction violates security (see below);
- `p` bit: instruction comes from a privileged address.

The four specified machine bits from right to left stand for power failure, memory parity error, nonexistent memory and rebooting. MMIX defines, that the program bits have to be set in `rK` when executing in user mode (otherwise a security PE is raised) and that program bit `p` has to be zero in `rK` when executing in privileged mode (otherwise a privileged PE is raised). [6, pg. 29]

Analogous to the forced trip, the forced trap instruction is defined as:

Name:	`TRAP X,Y,Z`
Effect:	$\texttt{rXX} \leftarrow 2^{63} \vee \mathrm{M}_4[@]$
	$\texttt{rWW} \leftarrow @ + 4$
	$\texttt{rYY} \leftarrow \$\texttt{Y}, \quad \texttt{rZZ} \leftarrow \$\texttt{Z}$
	$\texttt{rBB} \leftarrow \$255, \quad \$255 \leftarrow \texttt{rJ}$
	$\texttt{rK} \leftarrow 0$
	$@ \leftarrow \texttt{rT}$

That means, besides the different handler location, the different set of special registers and the fact that `TRAP` clears `rK`, `TRIP` and `TRAP` are equivalent. By clearing `rK`, dynamic traps are disabled. [6, pg. 28] The operating system may even use instructions in a way that would raise a PE, if the corresponding bit in `rK` were set. Because MMIX defines that "an instruction that traps with bits `x`, `k` or `b` does nothing; a load instruction that traps with `r` or `n` loads zero; a store instruction that traps with any of `rwxnkbsp` stores nothing" [6, pg. 29]. The meaning of the operands `X`, `Y` and `Z` can be defined by the operating system for any purpose. But two settings are predefined by MMIX:

1. `XYZ` $= 0$ should terminate the user process and
2. `XYZ` $= 1$ should offer a default action for a trip, for which the user program has not provided a handler (thus, instead of handling the trip, it can do a `TRAP 0,0,1`).

Analogous to forced and dynamic trips, the differences between forced and dynamic traps are, that for dynamic traps, the operands in `rYY` and `rZZ` correspond to the operands of the interrupted instruction and the handler location is different. Additionally, MMIX defines that if "the interrupted instruction contributed 1s to any of the `rwxnkbsp` bits of `rQ`, the corresponding bits are set to 1 also in `rXX`" [6, pg. 29]. More precisely, these bits occur in the first byte of the upper tetra of `rXX`.

11.2 Handling of Trips and Traps

After having described the mechanisms of triggering trips and traps, it should be explained how they can be handled. This section starts with the instruction that resumes an interrupted computation, followed by the handling of trips and traps.

11.2.1 Resuming

Of course, MMIX has to be able to resume the ordinary execution after a trip or trap has been handled. To do so, it provides a quite sophisticated instruction that can be used for various purposes.

Name:	`RESUME Z`

Effect:	if `Z` = 1:
	  `rK` ← `$255`, `$255` ← `rBB`
	@ ← `rW`|`rWW`
	ropcode ← (`rX`|`rXX`) ≫ 56
	if ropcode = 0: repeat(`rX`|`rXX`)
	if ropcode = 1: continue(`rX`|`rXX`)
	if ropcode = 2: set(`rX`|`rXX`)
	if `Z` = 1 and ropcode = 3: trans(`rX`|`rXX`)

The instruction `RESUME` comes in two versions: `RESUME 0` resumes the computation after a trip, while `RESUME 1` resumes it after a trap. Thus, `RESUME 0` uses `rW` and `rX`, while `RESUME 1` uses `rWW` and `rXX`. That does also mean, that the latter is prohibited in user mode. The default behaviour of `RESUME`, when the so called *ropcode* is `#80` (see `TRIP` and `TRAP`), is quite simple. The trip-version continues the execution at `rW`, while the trap-version restores `rK` and `$255` first and continues at `rWW` afterwards. But as the effect description shows, there are four other defined values of ropcode, which are more complicated. The four sketchy described actions have the following meaning:

- repeat(`rX`|`rXX`):
  MMIX interprets the lower four bytes of `rX`|`rXX` as an instruction and executes it. This is allowed for all instructions except `RESUME` itself.

- continue(`rX`|`rXX`):
  Continue is similar to repeat. It does also interpret the lower four bytes of `rX`|`rXX` as an instruction. But it does not use the operands provided in the instruction, but takes `rY`|`rYY` and `rZ`|`rZZ` instead. It is allowed for all "Set `$X` to the result of `$Y` OP `$Z|Z`" and "Set `$X` to the result of OP `$Z|Z`" instructions and for `TRAP` as well. Some implementations of MMIX may also allow `SYNCD` and `SYNCID`. Another restriction is, that the instruction can not increase `rL`, i. e. the `X` operand of the instruction has to be less than `rL`.

- set(`rX`|`rXX`):
  This ropcode tells MMIX to set the register, denoted by the third least significant byte of `rX`|`rXX`, to `rZ`|`rZZ`. Additionally, the third most significant byte is used to raise AEs. Again, `rL` can not be increased.

- trans(`rX`|`rXX`):
  Last but not least, this ropcode can be used to put a translation into the translation cache. It uses `rYY` as the virtual address and `rZZ` as the PTE and puts it into a TC. If the opcode of the instruction in the lower half of `rXX` is `SWYM` (*sympathize with your machinery*; the NOP instruction of MMIX, that does nothing), the translation will be put into the instruction TC, otherwise in the data TC. Additionally, if this opcode is not `SWYM`, the action repeat(`rXX`) will be performed.

All these actions behave as if they appeared as an instruction at location `rW`|`rWW`−4, i. e. as if they have been inserted into the instruction stream at that position. [6, pg. 30] It

will be described shortly for what reasons the different actions, depending on ropcode, are offered.

11.2.2 Handling Trips

As said in the last section, all different kinds of trips have their own handler location, which are 16 bytes away from each other. That means, each handler has 4 instructions available to do what ever is necessary. For example, it could do something like:

```
PUSHJ $255,Handler; PUT rJ,$255; GET $255,rB; RESUME 0
```

This way, the actual handler is called, saving all local registers on the stack. Before resuming, `rJ` and `$255` have to be restored, because – as described previously – `RESUME` will not do that. [6, pg. 28]

Another way is to let the operating system perform the default actions for a trip by:

```
TRAP 0,0,1; GET $255,rB; RESUME 0
```

In this case, no subroutine is called and thus, `rJ` has not to be restored. [6, pg. 28] Additionally, the reason why `RESUME` does not restore the values saved by `TRIP`, is that MMIX pursues the goal to perform only the minimum set of actions that are required.

When handling a dynamic trip, the operands of the instruction that caused the AE are put into `rY` and `rZ`. For example, if `DIV $0,$1,0` is executed, MMIX will raise a division by zero AE and set `rY` to $1 and `rZ` to 0. The handler could simply recognize that an AE occurred. But it could also replace `rY` and `rZ` with something else and change the ropcode in `rX` from `#80` to `#01` (continue). This way, the instruction would be repeated with operands `rY` and `rZ`. Another way would be to set the ropcode to `#02`, which would set `$X` to the value specified in `rZ`. [6, pg. 28]

11.2.3 Handling Traps

Similarly to dynamic trips, dynamic traps that are triggered because of a PE, save the operands of the instruction that caused it in `rYY` and `rZZ`. MMIX does not define the meaning of these registers for all instructions. But it is defined that all "Set `$X` to the result of `$Y` OP `$Z|Z`" instructions put `$Y` in `rYY` and `$Z|Z` into `$Z`. Additionally, load instructions put the virtual address into `rYY`[5], while store instructions do additionally put the octa to be stored (including unchanged bytes for cases like `STB`) into `rZZ`. [6, pg. 27]

Additionally it is noteworthy, that the OS has not many choices when considering PE handling. Because the only PEs, which are well defined, are the protection faults. For all others, i. e. when refering to a negative address, using a privileged instruction, breaking the rules of MMIX, violating security or jumping to a privileged address, the MMIX specification does not define the values of `rWW` and `rXX`. Thus, the operating system is unable to resume the computation of the user program in a reliable way. But of course, these PEs indicate a malicious or faulty user program anyway, so that the best option is a kill (and perhaps providing feedback to the user in form of a log entry or similar).

[5] Unfortunatly, the specification does not mention that explicitly for loads. But MMIX-PIPE does behave as explained and it would make no sense to do it for store instructions, but not for load instructions. After all, both may trigger a protection fault, for which the operating system has to know the virtual address. Having to calculate it from the `Y` and `Z` operand of the instruction for loads and simply read it from `rYY` for stores, would be very strange.

MMIX does also allow cheaper implementations of it, that let the software realize some instructions. In this case, these instructions cause a forced trap. For example, expensive operations like `DIV` or `FREM` could be implemented in software by checking the opcode in `rXX` when handling a forced trap. If it is one of these instructions, the result will be computed, put into `rZZ` and the ropcode will be set to `#02`. A subsequent `RESUME 1` will place `rZZ` into the desired register. As already said, even arithmetic exceptions could be raised by specifying them in the third most significant byte of `rXX`. [6, pg. 28]

A similar feature is, that address translation can be done in software. This can be requested by setting field f of `rV` to 1, but some implementations of MMIX might even require that. As soon as a translation is necessary, i. e. it is not already in the corresponding TC, a forced trap with ropcode `#03` is triggered. The virtual address will be available in `rYY`. When the translation is done, the handler should put the physical address into `rZZ`. A subsequent `RESUME 1` will put this translation into the TC and repeat the instruction, which will succeed this time. It should be noted, that if an instruction fetch fails, MMIX will put the instruction `SWYM` into `rXX`. This way, the translation will be put into the instruction TC and the fetch will be repeated when `RESUME 1` is executed. [6, pg. 28]

11.3 Interruptibility

Because of the complexity of some instructions (like `SAVE`, `POP`, `DIV`, `FREM` and so on), MMIX allows them to be interruptible. The specification words it as:

> Non-catastrophic interrupts in MMIX are always precise, in the sense that all legal instructions before a certain point have effectively been executed, and no instructions after that point have yet been executed. The current instruction, which may or may not have been completed at the time of interrupt and which may or may not need to be resumed after the interrupt has been serviced, is put into the special execution register `rX`, and its operands (if any) are placed in special registers `rY` and `rZ`.[6] [6, pg. 27]

That means, if an interrupt occurs while executing an instruction, a particular MMIX implementation may wait briefly until an interruption is possible and then put the state of computation of the current instruction into `rXX`, `rYY` and `rZZ`. In this case, the meaning of the registers is completely implementation dependent. For example, `FREM $X,$Y,$Z` may set `rYY` and `rZZ`, such that `rYY` mod `rZZ` = \$Y mod \$Z, but the actual values of `rYY` and `rZZ` may be different than `$Y` and `$Z`. [6, pg. 27 and 29] If MMIX sets ropcode to `#01` (continue), the operating system will not be required to care about that, because a `RESUME 1` will simply execute that instruction again with `rYY` and `rZZ`, which will lead to the same result.

It should also be mentioned, that `rXX` might not hold the instruction found at address `rWW` − 4. Not only because of jumps, but also because MMIX might put an instruction into `rXX`, that has been inserted internally. For example, if `ADD $X,$Y,$Z` is executed with `X` ≥ `rL`, the hardware may insert an instruction to increase `rL` first.

[6] Actually, `rXX`, `rYY` and `rZZ` are meant (which were introduced later in the specification), because the other ones are only used for trips, which are either explicitly requested or raised for AEs. Thus, they do not occur at arbitrary points of time during a computation, as interrupts from I/O devices may.

Chapter 3

The Implementation of GIMMIX

Now that the entire MMIX architecture has been explained, the simulator GIMMIX should be described. As already mentioned in the introduction, GIMMIX has been developed with the goal to be able to port an operating system to it. Therefore GIMMIX pursues to be correct (which implies to be not more complex than necessary), implement MMIX completely and offer a convenient and productive user interface, so that an OS can be debugged.

This chapter splits the depiction of GIMMIX into five parts. At first, the design decisions for the implementation are explained, followed by an overview of the simulator. Subsequently, the implementation of the MMIX architecture (the "core") is described in detail and yet existing devices are introduced shortly. Finally, the command line interface (CLI), which allows it to debug an OS or arbitrary other programs for MMIX, is presented.

1 Basic Design Decisions

Before starting with the description of the simulator, a few general design decisions that have been made should be explained.

1.1 No Pipelining

At first, it has been decided to abandon pipelining. Since it is a simulator, i. e. implemented in software, and has not the goal to explore the difficulties for a potential hardware pipelining implementation or similar, using pipelining would not bring advantages. Without it, the simulator is much simpler and thus the correctness is easier to achieve.

1.2 Uninterruptible Instructions

Similarly to the previous one and a bit related to it, it has been decided to make instructions uninterruptible, i. e. an instruction is either executed completely or not at all, as far as no program or machine exception occurs. As mentioned in 11.3 of chapter 2, MMIX allows it to make complex instructions like `POP` or `SAVE` interruptible by encoding the current state of computation in `rXX`, `rYY` and `rZZ`. But it does not require implementations to do that, which means that it is specification conform to make all instructions uninterruptible. [6, pg. 27]

Without this decision, GIMMIX would have required a concept that allows it to execute instructions in multiple steps, pause them for interrupts and resume them later.

This would have made it a lot more complex, without leading to real advantages for similar reasons as in the previous decision. Thus, for simplicity all instructions in GIMMIX are uninterruptible.

1.3 Programming Language

Obviously, when one wants to run an operating system in a simulator, i. e. a program that implements the whole simulated machine in software, efficiency is very important. Additionally, a lot of control over the produced machine code is necessary to be able to imitate the exact behaviour of the simulated machine. Therefore it has been decided to use the programming language C, which both allows to build efficient programs and offers a lot of control. Additionally, MMIX-SIM and MMIX-PIPE are implemented in C as well[1], so that it is easier to use code from them, such as the floating point algorithms.

Since an octa is the word quantity of MMIX, it will be used at nearly all places in the simulator. Therefore, its representation affects more or less the whole code. Using C89 on a 32-bit platform would mean, that no 64-bit type is available [10]. Thus, a struct would have to be created, that contains two 32-bit integers; one for the upper half of the octa and one for the lower half. Of course, every operation that is done with an octa, would have to be broken down to operations with the two 32-bit integers. In other words, the whole code of the simulator would get a lot more difficult to read and write, just because of the fact that there is no 64-bit type.

But there is an alternative. C99 provides the so called *exact-width integer type* `uint64_t`, which does always correspond to an unsigned 64-bit integer, independent of the underlying platform [11]. That is, the compiler will arrange things, such that this type (including all possible operations with it) is available. Although most currently available compilers offer no complete implementation of C99 yet [12], even the old version 3.0 (released 2001 [13]) of the GNU C Compiler, which is used in this project, supports this feature including many other useful ones [14]. For these reasons, C99 has been selected, but GIMMIX will only utilize some basic features of it such as exact-width integer types, `"// comments"`, mixed declarations and code, initialization in for-loops and `snprintf`, which are available in gcc and should be in almost all other compilers as well.

1.4 Host Platform

The host platform, i. e. the platform that runs the simulator, is expected to be a Linux system. Although most parts of GIMMIX are platform independent, some of them – like the simulated terminals for example – assume Linux. Furthermore, the simulator core (excluding e. g. some test generation programs, that expect x86) uses only standard C99, which means it should be hardware independent (but it has not been tested with different hardware).

Since both 32 and 64-bit platforms are widely spread nowadays, GIMMIX has the effort to support both of them. This is mostly achieved behind the scenes by adding a layer on top of the `printf`, `scanf` and `strtol` families. For the first two families, the layer lets the user specify the size of an argument with 'O','T','W' or 'B' (additionally to 'h', 'l' and 'L') for octa, tetra, wyde or byte, respectively. For `strtol`, the layer calls simply either `strtol` or `strtoll`, depending on the platform. But there are a few other things to take care of. For example, when using logical or arithmetical negation, the result depends on the size of the operand. That means, e. g. `-sizeof(octa)`

[1] As mentioned, they are written with CWEB, which produces C code in the end.

could lead to different results. If the type of `sizeof(octa)` is 32 bit wyde, the logical negation would produce `#FFFF FFF8`, instead of the perhaps intended `#FFFF FFFF FFFF FFF8`. Thus, whenever such operators are used with this intention, the operand has to be casted to an octa first.

2 Overview

Before explaining the main parts of GIMMIX, this section gives an overview and introduces the basic components, that are used throughout the code.

Conceptually, GIMMIX consists of several *modules* that interact with each other. The name of a module corresponds to a source file or folder in the directory **src** and to the associated header file or folder in the directory **include**. The following FMC diagram illustrates the most important modules and their relationships, whereas the CLI, the core and the devices are only shown as black boxes for now:

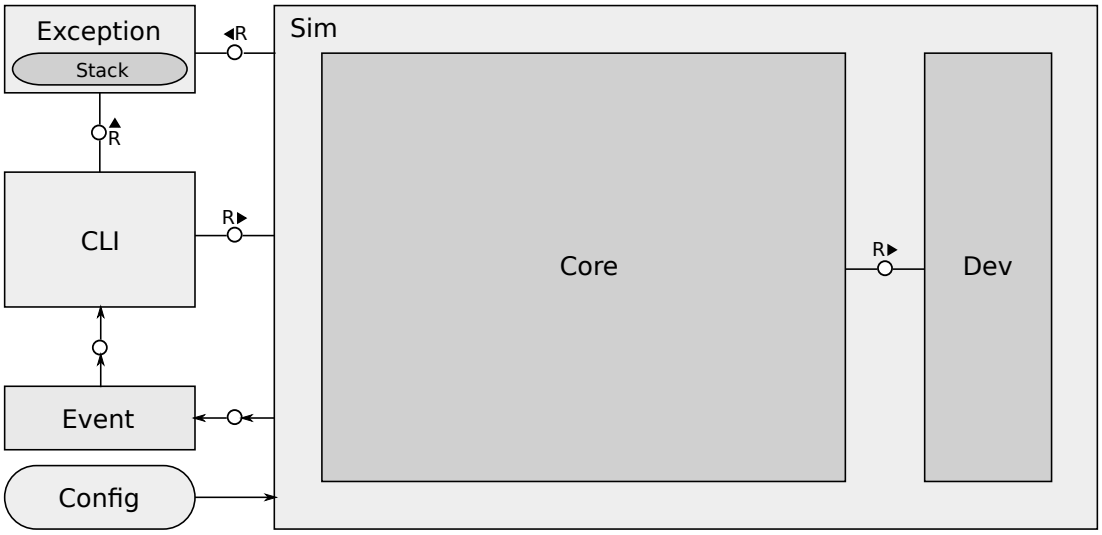


Figure 1: Architecture of GIMMIX in FMC notation

That means, the modules *core* and *dev*, to which the core sends requests for tasks like reading an octa from a specific physical address, are in a sense enveloped in *sim*. Sim supports some basic operations like initializing the whole system, resetting it or shutting it down. This is controlled by the module *CLI* (when running in interactive mode). The following sections describe the other general modules in the figure.

2.1 Exception

Although exception handling in C via `setjmp` and `longjmp` is not as convenient and powerful as for example in Java, it offers some advantages. That is, it separates "normal" code from exceptional code and provides the possibility to handle an exception only at places, where it can be handled in a sensible way. To achieve that and allow nested "try and catches", the module *exception* manages a stack of `jmp_bufs`, which hold the state created by `setjmp`. It is used for trips and traps in the core and also for exceptions in the CLI. The most important functions are:

- `ex_push(jmp_buf *environment)`:
  This function pushes the given environment onto the stack and is thus the equivalent to a `try`. The environment *has to* be created by the caller, because one can not jump back to a stack frame, that has been already destroyed.

- `ex_throw(int exception,octa bits)`:
  As the name suggests, it throws the exception with given number and additional information in `bits`, which is used for example to specify what trip or trap should be triggered. That is, it calls `longjmp` with the topmost `jmpbuf` and `exception` as arguments, so that the caller of `setjmp` receives `exception` as return-value.

- `ex_pop()`:
  This function removes the topmost environment from the stack. Hence, it is the equivalent to the end of a try-catch-construct.

In this thesis, the term "exception" without the prefix "arithmetical", "program" or "machine", refers to this concept, instead of the different exceptions of MMIX.

2.2 Event

As already mentioned, GIMMIX offers a sophisticated CLI that should allow it to debug an operating system or other programs in a convenient way. To do so, the CLI needs of course access to the core, i. e. it has to read the current state and should also be able to display the effect of an instruction or set breakpoints. Integrating all those facilities into the core would mean, that it gets more difficult to read and understand and it would also introduce strong coupling between the CLI and the core. Since the core is quite stable (the architecture is not expected to change much), but it may indeed be imaginable to provide a graphical user interface (GUI) to GIMMIX some day, it has been decided to work with events to supply the user interface with information. This way, the core is completely independent of the CLI and thus, assuming that a GUI is ready, it could replace the CLI within minutes, without affecting the core.

The module *event* implements this idea by providing `ev_register` to register a callback function for a specific event and `ev_unregister` to unregister it again. Additionally, events can be fired by calling `ev_fire`, `ev_fire1` or `ev_fire2` to raise an event with 0, 1 or 2 arguments, respectively. As figure 1 suggests, the core fires events and the event module forwards them to the CLI (and any other interested modules), which has registered the corresponding callbacks at the beginning. It is noteworthy, that if GIMMIX is used in non-interactive mode, the CLI will not be used at all and thus no callback will be registered, which will allow GIMMIX to be as performant as possible.

2.3 Config

Of course, GIMMIX should be configurable to some extent, which is realized by the module *config*. It consists of two categories of settings: the settings in the first one are configurable during compile time (i. e. constants are used) and the other ones are configurable during runtime. The latter can be passed to GIMMIX as command line arguments. In a sense, the compile time options are those that would be fixed when GIMMIX were a solid piece of hardware (number of local registers, cache size, device addresses, ...), while the runtime options could be changed more easily (amount of main memory, number of terminals, disk image, ...). But of course, there are also practical reasons, why one has to be able to specify the disk image or the rom image via command line argument.

3 Simulator Core

This section describes the most important part of the simulator, i. e. the implementation of the MMIX architecture.

3.1 Structure

The following figure breaks the core, that has been displayed as a black box in the previous FMC diagram, further down into pieces:

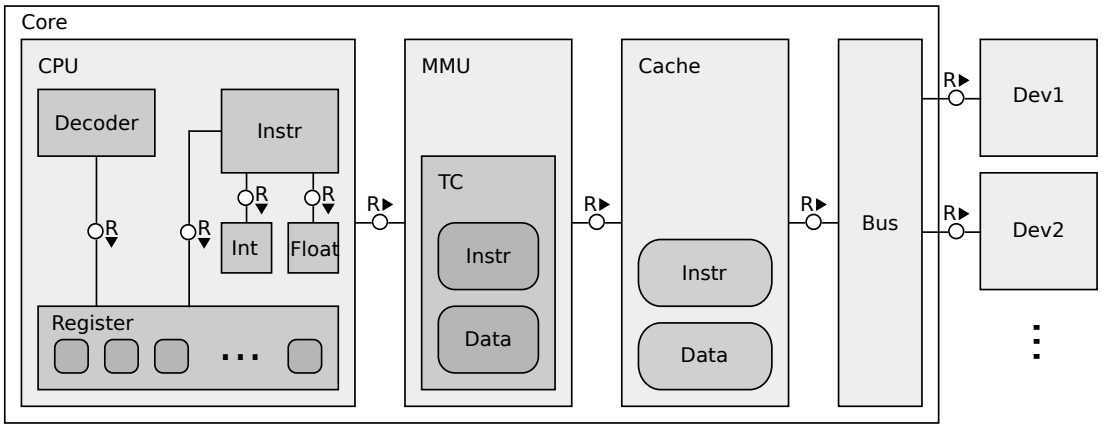


Figure 2: Architecture of the core in FMC notation

That means, it consists of four parts: *CPU*, *MMU*, *cache* and *bus*, whereas requests are always done from left to right. For example, when using `LDO` to read an octa from main memory (RAM, which is a device as well), the load instruction will request it from the MMU, which might have to translate the virtual address to a physical one first. Subsequently, the MMU will send a request to the cache to read an octa from the obtained physical address. If it is not in the data cache yet, the cache will send a request to the bus, which in turn will find the associated device and forward the request to it. Finally, the RAM device reads the octa from main memory and the result is passed back, ending at the load instruction.

The CPU consists of several parts as well. It has a module named *decoder*, which obviously is responsible for decoding an instruction. Additionally, the module *instr* implements all MMIX instructions, using *int* and *float* for integer and floating point arithmetic, respectively. Finally, the module *register* realizes the register stack.

The different parts shown in the diagram are explained in more detail in the following sections.

3.2 CPU

As the name suggests, the CPU is the central unit in GIMMIX, that is responsible for executing the instructions, using the help of several other modules.

3.2.1 Executing an Instruction

The execution of instructions is separated into three phases:

1. fetching the instruction, i. e. loading the tetra from memory,

2. decoding the instruction, which means that the operands are extracted from the tetra and translated into *arguments*, depending on the opcode and

3. executing the instruction, taking the arguments and performing the required actions.

The code in module CPU, that realizes the described procedure, looks like the following:

```
jmp_buf env;
int ex = setjmp(env);
if(ex != EX_NONE)
    cpu_triggerException(ex,ex_getBits());
else {
    ex_push(&env);

    // fetch instruction
    ev_fire(EV_BEFORE_FETCH);
    instrRaw = mmu_readInstr(pc,MEM_SIDE_EFFECTS);

    // decode
    instr = dec_getInstr(OPCODE(instrRaw));
    cpu_checkSecurity();
    dec_decode(instrRaw,&iargs);

    // execute
    ev_fire(EV_BEFORE_EXEC);
    instr->execute(&iargs);

    // handle realtime tasks
    if(++instrCount == INSTRS_PER_TICK) {
        instrCount = 0;
        timer_tick();
    }
    cpu_checkForInterrupt();
    cpu_counterTick();
}
ex_pop();
ev_fire(EV_AFTER_EXEC);
pc += sizeof(tetra);
```

Listing 3.1: Executing an instruction (slightly shortened)

As the listing shows, the whole procedure is surrounded by a kind of try-catch-construct, because several parts of it may throw an exception. If an exception is thrown, it will jump back to the `setjmp` call, which will return the exception-number. Afterwards the requested trip or trap is triggered (this will be described in detail later). Additionally, the code fires events at several interesting places to allow the CLI to perform some actions. After the instruction has been executed, realtime tasks are done if necessary (which might result in a finished disk-command or similar), it is checked whether `rQ∧rK` is non-zero and the counters are ticked (`rC`, `rU` and `rI`). Finally, the PC is advanced to the next instruction. It should be mentioned, that the function `cpu_checkSecurity`

ensures that all program bits are set in `rK`, when running in user mode, and that the privileged PC bit is *not* set, when running in privileged mode.

The actual decoding is done by the function `dev_decode` of the module *decoder*. The decoder contains a table, that assigns to each instruction both an *instruction format* and an *argument format*. The first one defines the meaning of the three operand bytes `X`, `Y` and `Z`. The second one defines the arguments for the execution function. More precisely, those functions receive a struct that contains the fields `x`, `y` and `z`, but the meaning of the fields differs, depending on the argument format. For example:

- `ADD` (`I_RRR`, `A_DSS`): `x` = `X`, `y` = `$Y`, `z` = `$Z`
- `ADDI` (`I_RRI8`, `A_DSS`): `x` = `X`, `y` = `$Y`, `z` = `Z`
- `BNN` (`I_RF16`, `A_CT`): `x` = `$X`, `z` = @ + 4 ∗ `YZ`
- `STBI` (`I_RRI8`, `A_SA`): `x` = `$X`, `y` = `$Y` + `Z`

In other words, the decoder performs a mapping from the instruction operands to the arguments, which hide certain details from the execution function. For example, the functions that implement the branches will simply receive the address to jump to and do not have to care about jumping forward or backwards and so on. Similarly, instructions like `ADD` and `ADDI` are put together, i. e. there is only one execution function for both, because the differences are handled by the decoder.

Besides the fact that the separation of decoding and executing prevents code duplication and is arguably more elegant, there is also a technical reason for it. Because, as described in the previous chapter, the instruction `RESUME` has a ropcode that allows it to execute the typical "set `X` to the result of `Y` OP `Z`" instructions and some other, whereas the operands are taken from `rYY` and `rZZ` instead of from the instruction itself. Thus, many instructions need to be executable both in the ordinary and in this special way. Since this separation hides the origins of the operands from the execution function, this is no problem anymore, because when implementing `RESUME`, one can simply create the argument struct from the instruction in `rXX` and exchange `y` and `z` with `rYY` and `rZZ`, respectively.

3.2.2 Execution Functions

The execution functions, mentioned in the previous section, are in most cases very short and simple. Because the arguments are provided in the desired form from the decoder and because other modules like MMU, int, float and register do the hard work. The following listing shows a few examples:

```
void cpu_instr_nor(const sInstrArgs *iargs) {
    octa res = ~(iargs->y | iargs->z);
    reg_set(iargs->x,res);
}

void cpu_instr_stou(const sInstrArgs *iargs) {
    mmu_writeOcta(iargs->y,iargs->x,MEM_SIDE_EFFECTS);
}

void cpu_instr_bev(const sInstrArgs *iargs) {
    if(!(iargs->x & 1))
```

```
        jumpTo(iargs->z);
}
static void jumpTo(octa addr) {
    if(!cpu_isPCOk(addr))
        ex_throw(EX_DYNAMIC_TRAP,TRAP_PRIVILEGED_PC);
    // substract sizeof(tetra) because it will be increased
    cpu_setPC(addr - sizeof(tetra));
}
```

Listing 3.2: Examples of the execution functions

That means, the implementation of `NOR` simply performs the bit operation and sets `$X`, `STOU` uses `mmu_writeOcta` to write `iargs->x` to `iargs->y` and `BEV` jumps to the desired location if `iargs->x` is even. The latter has to check the PC first to make sure that no jump from user space to privileged space can be done.

Besides this simplicity, there is an important point every instruction has to take care of. Although all instructions are uninterruptable, i. e. an interrupt can not happen during their execution, of course program exceptions can occur. Thus, every execution function (and many other functions as well) has to pay attention to this. In most cases, as in `cpu_instr_stou` above, it is simple because there is at most one function, that might throw an exception. Therefore, calling this function at first, i. e. before the state has changed, solves the problem. But if multiple functions might throw an exception, it gets more complicated:

```
1  void cpu_instr_cswap(const sInstrArgs *iargs) {
2      octa addr = iargs->y + iargs->z;
3      octa mem = mmu_readOcta(addr,MEM_SIDE_EFFECTS);
4      if(mem == reg_getSpecial(rP)) {
5          octa val = reg_get(iargs->x);
6          reg_set(iargs->x,0);
7          reg_set(iargs->x,val);
8          mmu_writeOcta(addr,val,MEM_SIDE_EFFECTS);
9          reg_set(iargs->x,1);
10     }
11     else {
12         reg_set(iargs->x,0);
13         reg_setSpecial(rP,mem);
14     }
15 }
```

Listing 3.3: Execution function of `CSWAP`

In this case, the functions to be careful with are `mmu_readOcta`, `reg_set` (storing values on the stack might lead to an exception) and `mmu_writeOcta`. Obviously, if multiple functions change the state and might throw an exception, calling them at first will not work. The following trick solves the problem for `CSWAP`. The read in line 3 does not change the state (caches are not considered critical in this case) and can thus be called at first. The `reg_set` in line 6 sets `$X` to an arbitrary value. If this function throws, nothing will have been changed yet. If it does not, line 7 will set `$X` back to the original value, which can not throw because the previous `reg_set` would have already done that. Hence, if the write in line 8 throws, the state will not have been changed yet. If line 9 is reached, `reg_set` will not throw either. All in all, no matter which function throws,

the state has not changed (line 12 is not critical here). Additionally, all used functions are assumed to be exception-safe as well. It will be described later, that a few functions actually do not garantee to be atomic, i. e. do not change the state if an exception occurs, but only make sure that a consistent state is reached.

3.2.3 Arithmetic

As described, MMIX supports both integer and floating point arithmetic. GIMMIX uses the modules *int* for integer arithmetic and *float* for floating point arithmetic. The former only implements the operations that differ from the behaviour defined by C99. The latter handles all operations in software, i. e. uses the integer arithmetic of the underlying platform. Both – and especially float – contain very complicated algorithms, but because they are not the central points of this thesis, they are described only sketchy. Besides that, they have been inherited from MMIX-SIM/MMIX-PIPE and have only been adjusted for GIMMIX. All functions are implemented, so that they are independent of the rest of the simulator. That means, they do not access registers, change the state or similar.

Integer Arithmetic

The module int contains functions for 128-bit signed and unsigned multiplication and division. They basically break down the operation to the 16-bit or 32-bit multiplication or division of the underlying platform, because 128-bit versions are not available. Additionally, the division has to handle the differences to C99. Because MMIX requires floored division, while C99 defines that truncated division is used [15, pg. 82]. Additionally it is worth mentioning, that the signed versions of both multiplication and division are based on the unsigned versions and only adjust the result accordingly.

The three shift types of MMIX – shift left, shift right arithmetically and shift right logically – are implemented in this module as well. The reason is, that C99 says:

> If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined. [15, pg. 84]

Thus, the functions implementing the shifts ensure, that for example $1 \ll 65$ leads to zero, as MMIX defines it.[2]

Floating Point Arithmetic

The basic principle in module float is to use the function `fl_unpack` to extract the components of a float from an octa. It returns a structure called `sFloat`, that contains the sign, the exponent, the fraction and the type (`ZERO`, `NAN`, `INF` or `NUM`). If the number is denormalized, it will be normalized and the exponent gets negative to abstract away the differences. Additionally, the fraction is always shifted left by 2, leading to two zero bits at the end, that are used for rounding and detection of inexact results.

All floating point operations work with instances of `sFloat`. Most of them consist of a `switch`, that does the corresponding action or returns the corresponding value, depending on the type of the operand(s). Finally, `fl_pack` is used to build an octa from the `sFloat` structure, which has represented the result so far. Consequently, it goes the opposite way than `fl_unpack`. It also rounds the result with the requested

[2] For example, the result of $1 \ll 65$ with gcc 4.4.5 on x86_64 is 2 instead of 0, because $x \ll y$ does actually mean $x \ll (y \bmod (sizeof(y) * 8))$.

rounding mode and indicates overflow, underflow and inexact AEs, if necessary. These are not directly triggered, but passed back to the caller. As soon as short floats are used, `fl_spack` and `fl_sunpack` are chosen instead of `fl_pack` and `fl_unpack`.

3.2.4 Register

The module *register* is one of the most important and also most complicated ones. Therefore, this thesis describes it in depth. Obviously, it is responsible for managing the registers. Additionally, it implements the main parts of the instructions `PUSHJ`, `PUSHGO`, `POP`, `SAVE` and `UNSAVE`. More precisely, the parts that affect the register stack. Thus, checking the instruction operands, setting the new PC and similar tasks, are done by their execution functions, while the rest is done by the register module.

Basic Data Structures and Operations

The basic data structures and functions of register are quite simple. It has the arrays `local`, `global` and `special` to hold local, global and special registers, respectively. Furthermore, it manages the abbreviations `L` for `rL`, `G` for `rG`, `S` for `rS`/8 and `O` for `rO`/8. The last two are used as indices into `local` and – instead of what the description of MMIX said – modulo is not used all the time (i. e. in general, $\texttt{O} \neq (\texttt{rO}/8) \wedge (2^n - 1)$), but only if `local` is accessed. In code listings, $2^n - 1$ will be called `LREG_MASK`, analogously to `GREG_MASK` for the global registers.

To offer other modules access to the mentioned arrays, the following functions are provided:

- `reg_getLocal`/`reg_setLocal`:
  These functions return or set the value of the specified local register, independent of `O`. This is not used by the core, but only by the CLI at the moment.

- `reg_getGlobal`/`reg_setGlobal`:
  These ones return or set the value of the specified global register and are also only used by the CLI.

- `reg_getSpecial`/`reg_setSpecial`:
  Similarly, these functions return or set the value of the specified special register.

- `reg_get`/`reg_set`:
  Finally, these two return or set the value of the specified dynamic register, i. e. if `rno` is the desired register number, $\texttt{rno} \geq \texttt{G}$ will denote global register `rno`, $\texttt{rno} < \texttt{L}$ will denote local register $(\texttt{O} + \texttt{rno}) \wedge (2^n - 1)$ and other values denote marginals.

The most interesting function just described is `reg_set`, which is implemented as follows:

```
void reg_set(int rno,octa value) {
    if(rno >= G)
        global[rno & GREG_MASK] = value;
    else {
        while(rno >= L) {
            local[(O + L) & LREG_MASK] = 0;
            if(((S - O - (L + 1)) & LREG_MASK) == 0)
                reg_stackStore();
```

```
            special[rL] = L = L + 1;
        }
        local[(O + rno) & LREG_MASK] = value;
    }
}
```

Listing 3.4: Implementation of `reg_set`

As the listing shows, if `rno` is greater than or equal to `L`, `L` will be increased first and registers will be set to zero. If the register ring is full, i. e. if the number of used local registers (plus one to leave a slot free) is equal to the number of available slots (difference of `S` and `O`), a register will have to be written to memory first. More precisely, `reg_stackStore` writes `local[S & LREG_MASK]` to M_8[`rS`] and increases `rS` by 8 and `S` by 1.

Similarly to the implementation of `CSWAP`, `reg_set` has to take care of exceptions, that might be thrown when writing to memory. But in contrary to `CSWAP`, it does not ensure that the state does not change. Instead, it will always leave in a consistent state. That means, if `reg_stackStore` fires an exception, the function might have already increased `L` and cleared some registers. But it is no problem, because the state is ok and the repetition of the instruction will continue at the point just left. To allow that, it is important that the increasing of `L` is done *after* storing a value on the stack. Doing it in the other way around would lead to an inconsistent state.

Pushing Registers Down

The function, that performs the central tasks of `PUSHJ` and `PUSHGO` is called `reg_push` and receives the `X` operand of the instruction as argument. It is implemented in the following way:

```
void reg_push(int rno) {
    int curL = L;
    if(rno >= G) {
        rno = curL++;
        if(((S - O - curL) & LREG_MASK) == 0)
            reg_stackStore();
        local[(O + rno) & LREG_MASK] = rno; // set hole
    }
    else {
        reg_set(rno,rno);   // set hole
        curL = L;
    }

    O += rno + 1;           // push down
    special[rO] += (rno + 1) * sizeof(octa);
    L = curL - (rno + 1);   // move L down (keep arguments)
    special[rL] = L;
}
```

Listing 3.5: Implementation of `reg_push`

That means, at first the hole is set to `rno` to remember the number of pushed down registers. Afterwards, `O` is increased correspondingly and `L` is adjusted, so that the arguments for the callee are kept. If `rno` $\geq$ `G`, `rno` will be set to `L`, `L` will be increased

by 1 and a register might have to be written to memory first. Similarly to `reg_set`, the function has to take care of the exceptions thrown by `reg_stackStore`. Thus, `L` is not changed before the call to ensure that the state has not been changed if it throws.

Popping Registers

The implementation for `POP`, which receives the number of registers to return as argument, is a bit more complicated:

```
void reg_pop(int rno) {
    octa holeData;      // last return value -> hole
    if(rno != 0 && rno <= L)
        holeData = local[(O + rno - 1) & LREG_MASK];
    else
        holeData = 0;   // if rno > L, hole <- 0

    int numRets = rno <= L ? rno : L + 1;
    if(special[rS] == special[rO]) {
        reg_stackLoad();
        if(((S - O - L) & LREG_MASK) == 0)
            special[rL] = --L;
    }
    int numRegs = local[(O - 1) & LREG_MASK] & 0xFF;

    while((tetra)(O - S) <= (tetra)numRegs) {
        reg_stackLoad();
        if(((S - O - L) & LREG_MASK) == 0)
            special[rL] = --L;
    }

    L = numRegs + numRets;
    if(L > G)
        L = G;
    // set hole
    if(L > numRegs)
        local[(O - 1) & LREG_MASK] = holeData;
    O -= numRegs + 1;
    special[rO] -= (numRegs + 1) * sizeof(octa);
    special[rL] = L;
}
```

Listing 3.6: Implementation of `reg_pop`

At first, the value for the hole is calculated. If `rno` $\leq$ `L`, it will be set to the last return value, if `rno` $>$ `L`, it will be set to zero and if `rno` $= 0$, the hole will not be set at all. Afterwards, the number of local registers the caller wanted to keep, is determined. It might be necessary to load this value back from memory into a register first. Subsequently, if required, local registers will be restored until the desired `numRegs` are available. Finally, `L` is set for the caller, the hole is written and `O` is decreased correspondingly.

Again, exception-safety has to be considered. Of course, both calls of `reg_stackLoad` might throw an exception. At a first glance one might think that it is sufficient to

make sure, that all instructions can be repeated successfully, after the PE that they had triggered previously, has been handled. In this case it would mean, that we do not need to care about it, because only a few registers of the caller might have been already loaded and will not be loaded again when repeating the instruction. But unfortunatly this is not enough. Because the OS might choose not to repeat the function directly, but e. g. let the user application handle a signal first (or something else, that is asynchronous to the ordinary control flow)[3]. In this case, arbitrary other instructions may be used and thus, the state of MMIX and in particular the state of the register stack has to be consistent. To ensure it in `reg_pop`, `L` has to be decreased when a register is restored. This way, one register of the callee is exchanged for a register of the caller. If this was not been done, the condition that α, β and γ may never move past each other, would be violated.

Page Faults on the Stack

Unfortunatly, the current version of the MMIX architecture, described in this thesis, has a design flaw. Because if a process causes a page fault while accessing the stack (for example, when using `PUSHJ`, `POP`, `SAVE` or setting a register), the operating system will not be able to save the state, handle the PE and resume the process successfully. Because obviously, the OS needs at least a few registers to be able to handle a PE. But in general (i. e. if the OS does not define, that some global registers can not be used by user applications), no register is available. Thus, the OS has to save a few registers first. For example, `SAVE` could be used for that purpose to save the whole state of the running program. But `SAVE` will write the state onto the stack, which does of course not work, because the stack has caused the page fault. Using `PUSHJ` to push some registers down would be an alternative way to supply the OS with a few registers. But equally, `PUSHJ` might have to save registers on the stack. Additionally it is not possible to save the state manually, because at least one register has to be available to build the destination memory address. In sum, there is no way to save the state of the running program. The consequence is, that the OS has either to restrict the user applications to dedicate some global registers to the OS, or it has to assign a stack of constant size, that is not swapped out or similar, to the applications.

Since these limitations are not acceptable, it has been chosen to solve this problem. The basic idea is to give the operating system a different stack than the user applications. To do so, a new special register called `rSS` is introduced, which holds the desired address of the *kernel stack*. Additionally, `SAVE` and `UNSAVE` are extended to offer a second version, which switches to the kernel stack and back to the user stack, respectively. That means, that `SAVE` will first set `rS` to `rSS` and will save the whole state on this stack. Consequently, `UNSAVE` will do the opposite, i. e. it first loads all values from the kernel stack back into registers and then will switch back to the user stack. This way, a trap handler can look like:

```
SAVE    $255,1  % the 1 requests the new version of SAVE
% ... handle the trap ...
UNSAVE  1,$255
% set $255 to rK for the current process
RESUME  1
```

[3] At least, the MMIX specification does not say the opposite. Therefore it is considered valid to let the user application handle a signal first when leaving the kernel, even if the kernel has been entered because of a PE.

Listing 3.7: Trap handler, using the extended SAVE and UNSAVE

Since SAVE stores the state on a different stack (of course, the OS has to make sure that it is big enough), it does not matter for what reason the PE has been triggered. Even if the user stack is currently unusable, the state can be saved and thus, the user program can be resumed afterwards.

To allow the operating system to use nested interrupts, i. e. enable interrupts while another one is handled, a SAVE $X,1 does not always switch the stack, but only if rS is currently in user space. Thus, rSS is expected to be in privileged space. An UNSAVE will perform the switch only if the associated SAVE has done it, too. Of course, the operating system should assign a different kernel stack for every process or thread. The implementation of SAVE and UNSAVE will be described in detail shortly.

An alternative solution for this problem has been suggested by Prof. Dr. Martin Ruckert, a professor for mathematics at the university of applied sciences in Munich and the author of "Das MMIX-Buch" [9]. He proposed to add the rules, that 1. the page(s) affected by the range rS to rO are always mapped (i. e. readable and writable) and 2. rG is at most 224. The first one ensures, that the OS can begin a trap handler with PUSHJ $255,YZ, because it may write to rS, but will never move rS past rO. The second makes sure, that the OS has 32 local registers available after the PUSHJ. Because the push increases rO and might increase it to a new unmapped page. To garantee that no register is written beyond the old rO (which is mapped, because of the first rule), the second rule is necessary. If the size of the local register ring is larger than the number of local registers that can be used, the ring will always have a few slots left until a register will have to be written beyond the old rO. That is, if only 224 local registers can be used and the ring size is at least 256, at least 32 slots will be free. Of course, PUSHJ and other functions that change rO have to trigger an exception if rO should be moved to an unmapped page. The PUSHJ that starts the trap handling will not trigger another trap, because rK is zero at that point. Additionally, the OS is forced to let no process run, for which the range rS to rO is not completely mapped.

Both solutions have their advantages and disadvantages:

- Without a separate kernel stack, a security problem arises. Because every user application can see the values, that have been written by the kernel. This might cause trouble if passwords, cryptographic keys or similar occur at that place. Therefore, every serious OS would have to establish a separate kernel stack anyway, by saving the user state and using UNSAVE to change the stack. Thus, the first solution simplifies that and makes it more efficient by requiring only a single SAVE.

- On the other hand, the first solution forces the OS to use the new SAVE and UNSAVE versions, if the problem should be solved and dynamic stack extension is desired. Since these instructions require a lot of memory accesses, they are time-consuming.

- The second solution allows the OS to use PUSHJ and POP to handle a trap, which is much faster.

- But the second solution does even require to use these instructions and forces the OS to handle a page fault caused on the stack, if necessary, before the state

can be saved with `SAVE`. Having only 32 registers to do so might make it difficult for large operating systems, that come with a sophisticated machinery for virtual memory. Even if it can be arranged, so that 32 registers are sufficient, it will be more inconvenient than a simple `SAVE`.

- Additionally, the second solution restricts the operating system in two ways. All user applications have at most 224 local registers and the OS has to make sure that the pages for the range `rS` to `rO` are always mapped. This does also mean, that current programs or operating system for MMIX might not work anymore.
- On the contrary, the first solution is completely downwards compatible, i. e. every MMIX program that respects the current specifiation will run on a new MMIX with the separate kernel stack.

It would also be imaginable to implement both solutions. This would offer more flexibility for the operating system. Some OSs might choose the simple `SAVE` and `UNSAVE`, while others might select `PUSHJ` to be able to handle page faults more quickly and to use `SAVE` only if necessary.

But for simplicity and – most important – compatibility, only the first solution has been implemented in GIMMIX. Because it has not yet been decided if one of the two solutions, both or something entirely different will be selected for the next version of the MMIX architecture.

Saving the State

Now that the page fault problem on the stack has been explained, the extended version of `SAVE` should be described. As already said, `SAVE` has to store all local and global registers and all special registers that might affect the computation on the stack. Depending on whether it should switch the stack, they are stored on the current one or on the kernel stack. At first, a few preparations are necessary:

```
void reg_save(int dst,bool changeStack) {
    octa oldrS = special[rS],oldrO = special[rO];
    bool doChangeStack = changeStack && !(oldrS & MSB(64));
    if(!doChangeStack)
        <check wether all stores will succeed>

    int oldL = L;
    O += L;
    special[rO] += L * sizeof(octa);

    if(doChangeStack) {
        octa newrO = special[rSS] + L * sizeof(octa) + (
           oldrO - oldrS);
        reg_setSpecial(rO,newrO);
        special[rS] = special[rSS];
    }
```

Listing 3.8: Implementation of `reg_save`, part 1 (partially pseudo-code)

For later computations, the old `rS` and `rO` are saved and it is determined whether the stack should really be changed. If not, one has to make sure that all values can be stored, because this is done on the user stack, which might cause page faults. This

way, the function will not change the state, if the stack is not completely mapped. Afterwards L is saved and all local registers are pushed down. Finally, if necessary, the stack is switched. That means, rO is changed to point to the end of all local registers that have to be saved. rS is set to the beginning of the kernel stack, whereas S is *not* changed here, because S indicates the register to store, while rS indicates the memory location. That is, S does still point to the register to save, but rS points to the kernel stack.

After the preparations, all pushed down registers (the current locals were pushed down previously), followed by rL are stored on the stack:

```
    special[rL] = L = 0;
    while(special[rO] != special[rS])
        reg_stackStore();
    reg_stackStoreVal(oldL,RSTACK_DEFAULT,S & LREG_MASK);
```

Listing 3.9: Implementation of reg_save, part 2

The last step stores the global and special registers on the stack:

```
    if(doChangeStack) {
        reg_stackStoreVal(oldrO,RSTACK_SPECIAL,rO);
        reg_stackStoreVal(oldrO + (oldL + 1) * sizeof(octa),
            RSTACK_SPECIAL,rS);
    }
    <store global and special reg.>
    if(doChangeStack)
        S = special[rS] / sizeof(octa);
    O = S;
    special[rO] = special[rS];
    reg_set(dst,special[rO] - sizeof(octa));
}
```

Listing 3.10: Implementation of reg_save, part 3 (partially pseudo-code)

If the stack is changed, the original rO and rS will have to be stored as well. As will be described shortly, rS has to be saved, so that it points to the beginning of the local registers to be restored in UNSAVE. When all registers have been saved and the stack is changed, S has to be corrected to correspond to the current value of rS. Finally, rO is set to rS and the top of the stack is written to the destination register. Additionally it should be mentioned, that a bit in the octa containing rG and rA stores whether the stack has been switched, which is required for UNSAVE.

Restoring the State

Of course, UNSAVE goes in the opposite direction. It has to restore the state from a given stack pointer. The implementation starts with loading the value containing rG, rA and the "change-stack bit" and checking whether the whole procedure will succeed:

```
void reg_unsave(octa src,bool changeStack) {
    src &= ~(octa)(sizeof(octa) - 1);   // octa-align it

    octa rGrA = mmu_readOcta(src,MEM_SIDE_EFFECTS);
    if((rGrA >> 56) < MIN_GLOBAL)
```

```
        ex_throw(EX_DYNAMIC_TRAP,TRAP_BREAKS_RULES);
    if(((rGrA) & 0xFFFFFF) & ~(octa)0x3FFFF)
        ex_throw(EX_DYNAMIC_TRAP,TRAP_BREAKS_RULES);
    if(changeStack && !(rGrA & ((octa)1 << 32)))
        changeStack = false;
    if(!changeStack)
        <check whether all loads will succeed>
```

Listing 3.11: Implementation of `reg_unsave`, part 1 (partially pseudo-code)

As the listing shows, the stack is only changed back, if it is desired and the associated `SAVE` had done it. Afterwards `rS` is set, which points to the stack to restore, and all global and special registers are loaded back into registers:

```
    reg_setSpecial(rS,src + sizeof(octa));
    <restore global and special reg.>
    if(changeStack) {
        octa newrS = reg_stackLoadVal(RSTACK_SPECIAL,rS);
        octa newr0 = reg_stackLoadVal(RSTACK_SPECIAL,r0);
        reg_setSpecial(r0,newr0);
        S = newrS / sizeof(octa);
    }

    reg_stackLoad();                        // load L ...
    int k = local[S & LREG_MASK]&0xFF;      // ...into this slot
    for(int j = 0; j < k; j++)
        reg_stackLoad();
```

Listing 3.12: Implementation of `reg_unsave`, part 2 (partially pseudo-code)

If the stack should be switched, `rS` and `r0` are loaded from the stack as well. But `rS` is not set immediately – in contrary to `S` – because the following loads should put the local registers into the corresponding slots, but they should still be loaded from the kernel stack. Finally, the new values for `rS`, `r0`, `rL` and `rG` are set:

```
    if(changeStack) {
        while(special[rS] != special[rSS])
            reg_stackLoad();
        special[rS] = S * sizeof(octa);
    }
    else {
        0 = S;
        special[r0] = special[rS];
    }
    L = k > G ? G : k;
    special[rL] = L;
    special[rG] = G;
}
```

Listing 3.13: Implementation of `reg_unsave`, part 3

As can be seen in the listing, if the stack is changed, all values on the kernel stack, that were hidden previously, will have to be loaded back into registers. Because as soon as the program uses the user stack again, those values would be lost on the kernel stack.

3.2.5 Trips and Traps

The last chapter about the CPU module explains how GIMMIX implements the interrupt and exception facilities of MMIX. At first, it is described how they are triggered, followed by the explanation of the `RESUME` implementation.

Triggering Trips and Traps

GIMMIX uses `ex_throw` for all kinds of trips and traps. To do so, it introduces the types `EX_FORCED_TRIP`, `EX_DYNAMIC_TRIP`, `EX_FORCED_TRAP` and `EX_DYNAMIC_TRAP`, which are passed as first argument to `ex_throw`. The different kinds are raised in the following ways:

- Forced trips and forced traps are triggered by the instructions `TRIP` and `TRAP`. Thus, their execution functions call `ex_throw` directly with the corresponding arguments.
- Dynamic trips will call `cpu_setArithEx` to set `rA` correspondingly. The CPU module will call `ex_throw` later, if necessary, i. e. if the enable bit for that AE is set. This way, the instructions are executed completely, even if an AE has been caused. MMIX defines the behaviour for all these cases.
- Dynamic traps for interrupts use `cpu_setInterrupt` to set a bit in `rQ`. After each instruction, the CPU checks whether `rQ` $\wedge$ `rK` is non-zero and calls `ex_throw`, if necessary.
- Dynamic traps for memory faults use `cpu_setMemEx`, which sets `rQ`, if the bit in `rK` is zero. Otherwise it stores the fault location and optionally the value to store and tells the caller that an exception should be raised. The reason is, that if the bit in `rK` is zero, failed loads should load zero and failed stores should store nothing. Thus, similarly to dynamic trips, the execution of the instruction has to be continued. The fault location and the value is required later for `rYY` and `rZZ`.
- Dynamic traps for other reasons (e. g. if the instruction breaks the rules) are simply raised by calling `ex_throw`.

That means, finally, all kind of trips and traps use `ex_throw`, which is catched in `cpu_execInstr`. As already mentioned, the function `cpu_triggerException` will be called in this case. Besides the first argument of `ex_throw`, which indicates the kind of trip or trap, the second argument provides additional information about the dynamic types. That is, it indicates the AE for dynamic trips and the interrupt, PE or ME for dynamic traps. The function `cpu_triggerException` performs the following actions:

1. Determine the value for `rX`/`rXX`;
2. If it is a dynamic trap exception, set the bit(s) in `rQ`. If `rQ` $\wedge$ `rK` is still zero, return;
3. If it is a trip, set registers `rX`, `rW`, `rY`, `rZ`, `rB` and `$255`. If a trap should be issued, `rK` is cleared and `rXX`, `rWW`, `rYY`, `rZZ`, `rBB` and `$255` are set;
4. Set the new PC.

The first action is the most interesting one and will thus be explained in more detail. At first, GIMMIX defines two additional, internal trap bits, that are used to determine what should be done: `TRAP_SOFT_TRANS` and `TRAP_REPEAT`. The former is used whenever

an address translation should be done in software and the latter for memory faults, for which the instruction should be repeated. At first, the implementation determines the ropcode to set in `rX`/`rXX`:

```
octa rxVal;
if(ex == EX_FORCED_TRAP && (bits & TRAP_SOFT_TRANS)) {
    if(bits & TRAP_PROT_EXEC)
        pc -= sizeof(tetra);
    rxVal = (octa)3 << 56;
}
else if(ex == EX_DYNAMIC_TRAP && (bits & TRAP_REPEAT)) {
    bits &= ~TRAP_REPEAT;
    rxVal = 0;
}
else
    rxVal = MSB(64);
```

Listing 3.14: Determining `rX`/`rXX` in `cpu_triggerException`, part 1

If software translation is requested, `rxVal` will be set to 3, which will urge `RESUME` to put a translation into the corresponding TC. Additionally, if `TRAP_PROT_EXEC` is set, i. e. a protection fault occurred because of a missing execution permission, the fetch will be repeated. Therefore, the PC is decreased to set `rWW` to the old @ instead of the default $@ + 4$. The second condition is true for all memory faults, that require a repetition of the instruction. This is requested for all protection faults with missing permissions, if the n field of a PTE or PTP is not equal to n in `rV` or if the segment limit has been exceeded. That means, basically for all types of memory faults, that are theoretically resolvable by the operating system. The third ropcode in the listing is the default one, which simply skips the instruction when `RESUME` is used.

Afterwards the instruction is put into the value to be constructed:

```
if(bits & TRAP_PROT_EXEC)
    rxVal |= (octa)SWYM << 24;
else
    rxVal |= useResume ? instrRawResume : instrRaw;
```

Listing 3.15: Determining `rX`/`rXX` in `cpu_triggerException`, part 2

That means, if an execution protection fault occurs, the NOP instruction `SWYM` will be put into `rxVal`. As already mentioned, `RESUME` will put the translation into the instruction TC in this case. Otherwise the current instruction will be put into `rxVal`. If `RESUME` is executed and not `RESUME` itself but the inserted instruction has caused a trap, `useResume` will be true, so that the inserted instruction will be put into `rxVal`. Finally, the PE bits are added, which tells the operating system whether a PE has caused the interruption and if so, which one:

```
if((bits & 0xFF00000000) && ex == EX_DYNAMIC_TRAP)
    rxVal |= bits & 0xFF00000000;
```

Listing 3.16: Determining `rX`/`rXX` in `cpu_triggerException`, part 3

Implementation of RESUME

As described in the chapter about the MMIX architecture, the RESUME instruction is very complicated, because of the different ropcodes it defines to allow different kind of actions before resuming the ordinary computation. At first, the following overview describes how it is implemented in principle:

```
void cpu_instr_resume(const sInstrArgs *iargs) {
    if(iargs->y > 1)
        ex_throw(EX_DYNAMIC_TRAP,TRAP_BREAKS_RULES);

    int isPriv = cpu_isPriv();
    int rx,ry,rz,rw;
    if(iargs->y == 1 && isPriv)
        rx = rXX, ry = rYY, ...
    else
        rx = rX, ry = rY, ...

    if(!isPriv) {
        if(iargs->y != 0)
            ex_throw(EX_DYNAMIC_TRAP,TRAP_PRIVILEGED_INSTR);
        if(reg_getSpecial(rw) & MSB(64))
            ex_throw(EX_DYNAMIC_TRAP,TRAP_PRIVILEGED_PC);
    }
    octa x = reg_getSpecial(rx);
    if(!(x & MSB(64)))
        <check if the ropcode can be used in desired way>

    if(iargs->y == 1 && isPriv) {
        reg_setSpecial(rK,reg_get(255));
        reg_set(255,reg_getSpecial(rBB));
    }
    cpu_setPC(reg_getSpecial(rw) - sizeof(tetra));

    if(!(x & MSB(64)))
        <execute action, depending on ropcode>
}
```

Listing 3.17: The implementation of RESUME (partially pseudo-code)

That means, the implementation is split into four parts:

1. The special registers to use are determined,
2. it is checked whether RESUME is allowed in the way it should be executed,
3. it resumes the ordinary computation and
4. executes the desired action, if necessary.

Again, exception-safety has to be considered, because some of the actions may cause an exception. In this case, two situations are distinguished. At first, RESUME itself might cause an exception. For example, if RESUME 1 is executed in user mode or if the "resume again" action wants to insert RESUME itself. Second, the inserted instruction might

cause exceptions, such as a protection fault when accessing memory. Conceptually, this instruction is inserted into the instruction stream at position `rW|rWW` − 4. Therefore, all checks regarding `RESUME` itself are done before returning to the ordinary computation and the checks for the inserted instruction including its execution is performed afterwards. This order is not only important for exception-safety, but also to ensure a correct environment. That is, e. g. the instruction pointer has to be set before the actions to ensure that relative jumps and similar instructions work as expected.

The individual ropcode actions are implemented as separate functions. For example, "resume again" looks like the following:

```
static void resumeAgain(bool isPriv,tetra raw) {
    sInstrArgs iargs;
    dec_decode(raw,&iargs);
    const sInstr *instr = dec_getInstr(OPCODE(raw));
    cpu_setResumeInstr(raw);
    cpu_setResumeInstrArgs(&iargs);
    if(!isPriv && (cpu_getPC() & MSB(64)))
        ex_throw(EX_DYNAMIC_TRAP,TRAP_PRIVILEGED_PC);
    instr->execute(&iargs);
}
```

Listing 3.18: Implementation of the "resume again" action

As the listing shows, the instruction (`raw`) is decoded and at the end, it is executed with the corresponding execution function. The calls of `cpu_setResumeInstr` and `cpu_setResumeInstrArgs` notify the CPU module about the instruction that should be executed and about its arguments. This is necessary to allow the CPU to put, for example, the correct instruction into `rX|rXX`, if an exception is triggered (see `useResume` above). Furthermore, it has to be checked whether the CPU was in user mode previously (`!isPriv`) and has changed into the privileged mode. This is of course not allowed, but without this check it would be possible to set `rW` to 0 and perform a `RESUME 0` in user mode to execute one instruction in privileged mode. Because this instruction is executed at `rW` − 4 and the instruction pointer determines the mode the CPU runs in.

3.3 MMU

The memory management unit is the first piece of the memory hierarchy in GIMMIX. It is responsible for loading values from the next piece of the hierarchy (the cache), performing address translation and mapping byte, wyde or tetra requests to octa requests, when accessing the cache.

The module *MMU* provides three kinds of functions for the simulator core:

1. Functions to read from memory,
2. functions to write to memory and
3. functions to synchronize caches and memory.

Additionally, the whole memory hierarchy uses four flags to communicate the desired behaviour to the different parts of the hierarchy:

1. `MEM_SIDE_EFFECTS`: If disabled, the state of GIMMIX is not changed and no events are fired. This is used by the CLI, which should of course not change the state when for example a value is read from memory.

2. `MEM_UNCACHED`: Tells the cache, that if a value is not yet in the cache, the affected cache block should not be loaded, but the value should be read/written directly from/to memory.

3. `MEM_UNINITIALIZED`: If enabled, the cache will not load the affected cache block from memory, if not already present, but initialize the cache block to zero.

4. `MEM_IGNORE_PROT_EX`: If enabled, no dynamic trap will be triggered for protection faults. This is used for synchronizing cache and memory, for which MMIX defines that no protection faults occur.

3.3.1 Reading from Memory

The MMU module offers five reading functions – one for each quantity and a separate function to read an instruction (to use the instruction TC and instruction cache). All read functions use the internal function `mmu_doRead` to perform the actual reading. The octa-version will simply return the value read by that function, while others extract the quantity from the corresponding position. For example, `mmu_readTetra` is implemented as:

```
tetra mmu_readTetra(octa addr,int flags) {
    int off = (addr & (sizeof(octa) - 1)) >> 2;
    octa data = mmu_doRead(addr,MEM_READ,flags);
    return (data >> (32 * (1 - off))) & 0xFFFFFFFF;
}
```

Listing 3.19: Implementation of `mmu_readTetra`

The actual reading function looks like the following:

```
static octa mmu_doRead(octa addr,int mode,int flags) {
    octa res;
    jmp_buf env;
    int ex = setjmp(env);
    if(ex != EX_NONE) {
        mmu_handleMemEx(ex,addr,0,flags);
        // loads that cause an exception, load zero
        res = 0;
    }
    else {
        ex_push(&env);
        int exp = (mode & MEM_READ) ? MEM_READ : MEM_EXEC;
        int cache = (mode & MEM_READ) ? CACHE_DATA :
           CACHE_INSTR;
        octa phys = mmu_translate(addr,mode,exp,flags &
           MEM_SIDE_EFFECTS);
        res = cache_read(cache,phys,flags);
    }
    ex_pop();
    return res;
}
```

Listing 3.20: Implementation of `mmu_doRead`

As the listing shows, exceptions are catched here, because not all exceptions actually cause a trap. It depends on the current `rK` and whether the flag `MEM_IGNORE_PROT_EX` is set. If no trap is caused, the instruction will have to be finished, i. e. the register has to be set, for example. The decision whether to trap or not is made by `mmu_handleMemEx`, which uses `cpu_setMemEx` and calls `ex_rethrow` to throw the catched exception again, if necessary. The first step of `mmu_doRead` is to translate the virtual address to a physical one via `mmu_translate`. Afterwards the octa is read from the cache, which in turn might request it from the corresponding device.

3.3.2 Writing to Memory

Before taking a closer look at the address translation, it should be described how the write functions work. Analogous to the read functions, the MMU provides a function for each quantity of MMIX, whereas all functions except the one that writes an octa, reads an octa first using `mmu_doRead`, replaces the corresponding byte, wyde or tetra and writes the octa back to memory using `mmu_doWrite`. This function is implemented as follows:

```
static void mmu_doWrite(octa addr,octa value,int flags) {
    jmp_buf env;
    int ex = setjmp(env);
    if(ex != EX_NONE) {
        mmu_handleMemEx(ex,addr,value,flags);
        // stores that cause an exception, store nothing
    }
    else {
        ex_push(&env);
        octa phys = mmu_translate(addr,MEM_WRITE,MEM_WRITE,
           flags & MEM_SIDE_EFFECTS);
        cache_write(CACHE_DATA,phys,value,flags);
    }
    ex_pop();
}
```

Listing 3.21: Implementation of `mmu_doWrite`

Of course, the implementation is very similar to the one of `mmu_doRead`. Exceptions are catched and handled by `mmu_handleMemEx`, the address is translated first and finally, it writes to the cache.

3.3.3 Address Translation

Virtual addresses are translated to physical ones via `mmu_translate`. Without going too far into the details here, it works roughly in the following way:

```
octa mmu_translate(octa addr,int mode,int expected,bool
   sideEffects) {
    int tc = (mode & MEM_EXEC) ? TC_INSTR : TC_DATA;
    if(sideEffects)
        ev_fire2(EV_VAT,addr,mode);
    if(addr & MSB(64))
        return addr & ~MSB(64);
```

```
    <check rV, i.e. page size and f field>
    octa pte;
    sTCEntry *tce = NULL;
    if(sideEffects)
        tce = tc_getByKey(tc,tc_addrToKey(addr));
    if(tce == NULL) {
        if(sideEffects && vtr.f == 1)
            ex_throw(EX_FORCED_TRAP,TRAP_SOFT_TRANS | mode);
        <translate the address, yielding the PTE>
        pte = tc_pteToTrans(pte);
        if(sideEffects)
            tc_set(tc,tc_addrToKey(addr),pte);
    }
    else
        pte = tce->translation;

    if(!(pte & expected))
        ex_throw(EX_DYNAMIC_TRAP,TRAP_REPEAT | (mode&~pte));
    octa trans = pte & ~(octa)0x7;
    octa phys = (trans << 10) + (addr & ((1 << vtr.s) - 1));
    return phys;
}
```

Listing 3.22: Implementation of `mmu_translate` (partially pseudo-code)

That means, if the address is in privileged space, no translation will be done, but only the most significant bit will be cleared. If it is in user space, the corresponding TC will be asked whether a translation exists. If so, it will be used, otherwise the actual address translation will be performed and the resulting PTE will be put into the translation cache. Additionally, if software translation is desired, the corresponding exception will be thrown (`vtr` is a struct containing all fields of `rV`). As soon as the PTE is known, the access permissions are checked and if these are sufficient, the resulting physical address is returned.

It should be noted, that the function takes `mode` and `expected`, which are both a combination of the bits `MEM_READ`, `MEM_WRITE` and `MEM_EXEC`. The former determines the actual access type and the thrown exceptions, whereas the latter contains the flags that are required to be set in the PTE. The reason for the separation is that the functions for writing a byte, wyde or tetra have to perform a read request first. Of course, this request should fail, if the page does not allow reads. But it may have to throw both a read and write protection fault, because we have to write afterwards. More precisely, the bits that are missing have to be reported. That is, if read permission is missing, but write permission is present, a read protection fault will be thrown. If both are missing, both will be thrown. For this reason, the read requests used in the write functions, use `MEM_READ` for `mode` and `MEM_READ | MEM_WRITE` for `expected`. In the ordinary cases, `mode` and `expected` are the same. Additionally, the way the functions for writing a byte, wyde or tetra to memory have to be implemented including the thrown exceptions implies, that when a page is only writable, no bytes, wydes and tetras can be written to that page, but only octas.

Furthermore, the state will not be changed and no events will be fired, if no side effects are desired. In other words, if the CLI accesses memory, the translation will be done every time and the translation caches will be ignored.

3.3.4 Translation Cache

Since the translation procedure is handled in the MMU module, the module *TC* is very simple. It only holds two arrays with TC entries – one for the instruction TC and one for the data TC – and offers other modules access to it. The most important functions are `tc_getByKey`, `tc_set`, `tc_remove` and `tc_removeAll`. The first one searches for a given key (i. e. more or less the virtual address) and returns the translation entry containing the key and the translation, if found. The function `tc_set` puts a translation into the TC, `tc_remove` removes a specific entry and `tc_removeAll` removes all entries from a TC. Last but not least, `tc_pteToTrans` and `tc_addrToKey` can be used to transform a PTE to a translation and a virtual address to a key, respectively.

3.4 Cache

As already said, providing caches for physical memory is optional in MMIX. Therefore, it can be implemented in nearly arbitrary ways or not at all.

3.4.1 Organisation

For GIMMIX, it has been decided to keep it as simple as possible, but complicated enough to be able to implement all instructions regarding caches in a sensible way. In short, GIMMIX provides a fully associative, write-back, write-alloc cache and with a random replacement policy. Each cache consists of a specific number of *cache blocks* (also known as *cache lines*), whereas each cache block contains a specific number of octas and has a dirty flag. Additionally, each cache block has a *tag*, which corresponds to the physical address of the first octa in it. More detailed:

- *Fully associative* means, that every address can be put into every cache slot. Thus, all slots will be searched if an address is looked up. It has been chosen for simplicity.

- *Write-back* means, that the cache content is not immediatly written back to main memory, but only as soon as necessary or explicitly requested. Without it, the OS would not have to make sure that the main memory is up to date, if for example instructions were loaded into memory (in privileged mode, a `SYNCID` removes blocks from the caches without writing it to main memory first. Thus, without write-back, a `SYNCID` would be enough; otherwise, a `SYNCD` has to be done first).

- The term *write-alloc* means, that if writing and the associated cache block of an address it not yet in cache, the block will be read from memory into the cache first and the value will be put into that block. The advantage in this case is, that it makes it more transparent and consistent, because all data is always at first in the cache (if not explicitly requested otherwise).

- A random replacement policy has been chosen to make the selection of victims simple. Random in this case means, that the module will walk through the slots in linear order, i. e. $0, 1, \ldots, n-1, 0, 1, \ldots$, if n is the number of slots.

The module *cache* provides an instruction and a data cache and offers functions for reading/writing from/to a specific physical address, removing blocks and flushing blocks to memory. All these functions refer to the cache, that has been specified via parameter (either instruction or data cache).

3.4.2 Reading and Writing

The functions for reading and writing are the most interesting ones and are thus explained in more detail. The function `cache_read` is implemented as follows:

```
octa cache_read(int cache,octa addr,int flags) {
    if(addr >= DEV_START_ADDR)
        return bus_read(addr,flags & MEM_SIDE_EFFECTS);

    sCache *c = caches + cache;
    sCacheBlock *block = cache_get(c,addr,flags,false);
    if(!block)
        return bus_read(addr,flags & MEM_SIDE_EFFECTS);
    return block->data[(addr & ~c->tagMask) / sizeof(octa)];
}
```

Listing 3.23: Implementation of `cache_read`

At first, it is checked whether the physical address refers to an I/O device, whose area starts at `DEV_START_ADDR` (defined as `#0001 0000 0000 0000`). I/O devices are always uncached and thus, `cache_read` directly reads from the bus, ignoring the cache. If main memory is requested, `cache_get` will be used to get the cache block for the specified address. By default, the value will be extracted from the corresponding position and returned. But there are cases, that require a direct access of the bus. For example, when no side effects are desired or `flags` contains `MEM_UNCACHED`.

The function to write to the cache is implemented analogously, except that it calls `bus_write` and sets the corresponding octa in the cache block to the specified value. Both functions use `cache_get` to determine the affected cache block, which should be described in more detail:

```
static sCacheBlock *cache_get(sCache *cache,octa addr,int
   flags,bool isWrite) {
    sCacheBlock *block = cache_find(cache,addr);
    if(flags & MEM_SIDE_EFFECTS)
        ev_fire2(EV_CACHE_LOOKUP,cache - caches,addr);

    if(block == NULL) {
        if(!(flags & MEM_UNCACHED) &&
                (isWrite || (flags & MEM_SIDE_EFFECTS))) {
            block = cache_findVictim(cache);
            if(block) {
                if(block->dirty)
                    cache_flushBlock(cache,block,flags);
                if(!(flags & MEM_UNINITIALIZED))
                    cache_fill(cache,block,addr,flags);
                else {
                    block->tag = addr & cache->tagMask;
```

```
                    memset(block->data,0,cache->blockSize);
                }
            }
        }
    }
    return block;
}
```

Listing 3.24: Implementation of `cache_get`

At first, `cache_find` is called, which simply iterates through all cache blocks and returns the block, if it is already present. If it is not yet present, it might have to be loaded first. As the listing shows, this will not be done, if `MEM_UNCACHED` is desired and otherwise always when writing to the cache, but for reading only, if side effects are tolerated. The idea behind it is, that writes – in contrary to reads – are considered as an explicit request to change the state of MMIX. That means, if the CLI is used to set a value in memory, for example, the state will be changed. Instead, reading in the CLI does never produce any side effect. If the block has to be loaded, a free block will be required. The function `cache_findVictim` will choose a free one or an arbitrary victim. This does always succeed, except if `cache` has no cache blocks at all (that is, the cache is disabled, which can be configured). If a block has been found, it will have to be flushed to memory, if it is dirty. Finally, it is either filled with zeros or filled with the values from the corresponding memory location.

3.4.3 Implementation of Caching Instructions

Last but not least, it should be explained how the instructions regarding caching are implemented in GIMMIX. Because the MMIX architecture does not specify that completely, but leaves some details up to the particular implementation.

- `LDUNC $X,$Y,$Z|Z` behaves as `LDO`, but passes `MEM_UNCACHED` to the memory hierarchy to request that the associated cache block is not loaded from memory, if not already present.
- `STUNC $X,$Y,$Z|Z` behaves as `STO`, but does also use the flag `MEM_UNCACHED`.
- `PRELD X,$Y,$Z|Z` ensures, that the bytes $M_1[\$Y + \$Z|Z]$, ..., $M_1[\$Y + \$Z|Z + X]$ are present in the data cache.
- `PREGO X,$Y,$Z|Z` has the same behaviour, but puts the bytes in that range into the instruction cache.
- `PREST X,$Y,$Z|Z` makes sure, that the bytes $M_1[\$Y + \$Z|Z]$, ..., $M_1[\$Y + \$Z|Z + X]$ are in cache, but passes `MEM_UNINITIALIZED` to the memory hierarchy. This way, the data is not loaded from memory, but initialized with zeros (which is of course not required, but simplifies debugging).
- `SYNCD X,$Y,$Z|Z` flushes the cache blocks affected by the range $M_1[\$Y + \$Z|Z]$, ..., $M_1[\$Y + \$Z|Z + X]$ to main memory. If running in privileged mode, they will additionally be removed from the cache.
- `SYNCID X,$Y,$Z|Z` will remove the specified range from the instruction cache and flushes the range in the data cache, if running in user mode. This way, manually

fabricated instructions will be interpreted correctly, because they have to be loaded from main memory into the instruction cache first and the content of the data cache has been flushed to main memory. If running in privileged mode, the range will be removed from both the instruction and the data cache.

The instructions `PRELD`, `PREGO` and `PREST` use `MEM_IGNORE_PROT_EX` to ignore protection faults. Analogously, `SYNCD` and `SYNCID` catch these exceptions to ensure, that no protection fault is triggered.

3.5 Bus

GIMMIX uses the module *bus* as an interface to the attached devices. The idea is to make the rest of the simulator core independent of the present devices. In this case, a compromise between simplicity and dynamic has been chosen. Because on the one hand, it is expected that some day more devices will be provided, but on the other hand, it is considered unlikely, that devices are offered by third party vendors. Thus, it is not that dynamic that no code change is necessary at all (by using shared libraries, for example), but dynamic enough to require as few code changes as possible and still keep the extension mechanism simple.

To achieve that, the bus maintains a single linked list with the attached devices. The init function of the bus, which is called during initialization of the simulator, calls the init function of all devices. These in turn will use `bus_register` to register themself to the bus. That is, they tell the bus their name, their address range in I/O space, their interrupt mask and the callback functions for reading, writing, resetting the device and shutting it down. This way, for a new device, the file implementing the device has to be added and its init function has to be called in the init function of the bus. No other changes are necessary.

The most important functions of the bus are `bus_read` and `bus_write`. Since there are no interesting differences, it is sufficient to take a look at `bus_read`:

```
octa bus_read(octa addr,bool sideEffects) {
    addr &= ~(octa)(sizeof(octa) - 1);
    const sDevice *dev = bus_getDevByAddr(addr);
    if(dev == NULL)
        ex_throw(EX_DYNAMIC_TRAP,TRAP_NONEX_MEMORY);

    octa data = dev->read(addr,sideEffects);
    if(sideEffects)
        ev_fire2(EV_DEV_READ,addr,data);
    return data;
}
```

Listing 3.25: Implementation of `bus_read`

As the listing shows, at first the desired device is searched with `bus_getDevByAddr`. If no device is found, the physical memory does not exist and thus, an exception will be thrown. Otherwise the read function of the device will be called and the value will be returned. It is noteworthy, that the main memory (RAM) and the ROM are devices as well, because the only difference to the actual I/O devices like terminal, disk and so on is the address range. That is, RAM uses `#0000 0000 0000 0000` to `#0000 FFFE FFFF FFFF` and ROM uses `#0000 FFFF 0000 0000` to `#0000 FFFF FFFF FFFF`, while the other devices

use the space beginning at `#0001 0000 0000 0000`. To prevent the introduction of a special case for RAM and ROM, they are treated like all other devices as well.

4 Devices

As indicated previously, all devices in MMIX are memory mapped. That is, each device owns a specific range in the I/O space of the physical memory to provide access to itself. It is used for reading the state of the device, changing the state and urging the device to perform some kind of action. To behave like a hardware implementation, GIMMIX has a configurable *tick mechanism*, managed by the timer device, to simulate a delay when accessing devices. It achieves that by defining the number of instructions that can be executed in a tick as `INSTRS_PER_TICK` and thus calling the tick function of the timer every `INSTRS_PER_TICK` instructions. Each device can register callbacks at the timer, that are called after a specific number of ticks. For example when doing a disk operation, reading from terminal or similar. This way, the delay of each operation can be defined in ticks to approximate the delay that would be present in a hardware implementation. This concept has been inherited from the ECO32 project. ECO32 is a simple, but still realistic 32-bit big-endian RISC processor, designed by Prof. Dr. Geisse for research and teaching purposes [16].

To be able to do something useful, GIMMIX already provides a few devices. All of them – except RAM – have been taken from ECO32 and have been adjusted to fit the needs of GIMMIX. The following devices are present so far:

- RAM:
  The RAM device obviously simulates the main memory of GIMMIX. Without going into the details here, it manages the memory in a so called *treap*, a combination of a binary tree and a heap [17], whereas each node in the tree holds 2048 bytes of the memory. As soon as a memory location is accessed, a node is created, if not already done, and inserted into the tree. This way, GIMMIX can allow large amounts of main memory and will only have to pay for it, if it is actually used. The implementation has been inherited from MMIX-SIM (see [4, pg. 12]).

- ROM:
  The ROM device contains a memory area of constant size, in which a firmware or similar can be integrated. For example, a program that allows the user of the machine to boot from the hard disk.

- Timer:
  As already said, the timer is used internally for the tick machinery. Additionally, it offers the opportunity to raise a timer interrupt after a specific number of ticks, configureable over device registers (accessible at a specific memory location).

- Terminal:
  The terminal device allows to attach a variable number of terminals to GIMMIX. To achieve that, an instance of xterm[4] is started for each terminal. Each of them has a *receiver*, that reads from xterm every few ticks and notifies about read characters by setting a flag in a device register and optionally by triggering an interrupt. Analogously, it has a *transmitter* that allows the software to write a character to the xterm terminal, which sets a flag and optionally raises an interrupt as soon as the operation is finished, as well.

[4] Xterm is a terminal emulator for the X Window System. [18]

- Disk:
  The disk device offers a hard disk for the software by using a disk image file, divided into 512 byte sectors. Disk commands can be started by putting the sector number and sector count into specific device registers and finally setting the start flag and mode (read or write). The read command reads the requested sectors from the disk image into a disk buffer, which can in turn be read by the software when reading from the corresponding memory location. Consequently, the write command writes the content of the disk buffer to the disk image. Analogously to the terminals, it sets a flag when an operation is finished and optionally raises an interrupt.

- Output:
  Last but not least, GIMMIX has an output device, that allows the software to write characters to a file. This is intented for debugging purposes only.

5 Command Line Interface

The last not yet described part of the GIMMIX simulator is the command line interface. As already mentioned, GIMMIX has the goal to provide a convenient, intuitive and productive interface, to allow it to debug an operating system. Of course, the user needs some commands to work with the simulator, such as "execute one instruction", "print a part of the state" or "disassemble instructions". Additionally, it has been decided to develop a small language, that is used by all commands. The language allows it to access the different entities of MMIX (registers, virtual memory, ...) and do calculations with them. This way, the user has a lot of flexibility when examining a program, all commands work in a common way and it is easy to add new commands.

5.1 The Language

The language is designed to be both usable interactively and non-interactively. That is, when controlling the simulator with the command line interface, it is used interactively. But it is also possible to execute scripts before entering the CLI; for example to establish an initial environment for convenience or to start and control the simulator fully automatized. To achieve that goal, the language consists of an arbitrary number of commands with one command per line. Each line looks like:

```
commandName [<arg1> <arg2> ...]
```

That is, the command name comes first, followed by an arbitrary number of arguments. The command name and the arguments are separated by whitespace. Thus, no whitespace can be used inside an argument. This has been chosen, because it is no general purpose programming language, in which complex expressions are used, but it is intended as a language to use the simulator. That means, it is expected that the user does not want to type more than necessary and whitespace in an expression is not required.

Each argument is an expression, whose grammar looks like the following:

```
expr = string | integer | float
    | expr,"+",expr | expr,"-",expr
    | expr,"*",expr | expr,"/",expr | expr,"%",expr
    | expr,"s*",expr | expr,"s/",expr | expr,"s%",expr
```

```
| expr,"&",expr | expr,"|",expr | expr,"^",expr
| expr,"<<",expr | expr,">>",expr | expr,">>>",expr
| "~",expr | "-",expr
| "@"
| expr,"..",expr | expr,":",expr
| "M","[",expr,"]"
| "M1","[",expr,"]" | "M2","[",expr,"]"
| "M4","[",expr,"]" | "M8","[",expr,"]"
| "m","[",expr,"]"
| "l","[",expr,"]" | "g","[",expr,"]"
| "sp","[",expr,"]"
| "$","[",expr,"]" | "$",integer
| "(",expr,")";
```

Listing 3.26: Grammar of expressions in EBNF

The nonterminal `string` begins with a letter (`[A-Za-z]`), followed by an arbitrary number of alphanumeric characters. An `integer` can be specified in base 2 using the prefix `0b`, octal using `0`, hexadecimal using `0x` or `#` and decimal without a prefix. Additionally, the special registers `rA`, ..., `rZ`, `rBB`, `rSS`, `rTT`, `rWW`, `rXX`, `rYY` and `rZZ` are translated into their internal number, so that they can be used at arbitrary places (e. g. `sp[rA]`). Finally, a `float` can be specified in the form accepted by the function `strtod` of the C standard library. That means, either by optionally starting with the decimal part, followed by a dot and the fraction part or by using the scientific notation. Thus, for example `1.0`, `.6`, `2.5e1` or `12E-1`.

As the grammar shows, the notation is very similar to the one generally used in this thesis. Additionally, one can see that three data types exist in the language: integers, floating point numbers and strings. Furthermore, it has the following groups of operators:

- Arithmetic:
  One can use the well known arithmetic operations addition, substraction, multiplication, division and modulo with `+`, `-`, `*`, `/` and `%`, respectively. Additionally, their signed counterpart is available for `*`, `/` and `%` by prefixing it with an "s".

- Bit and shift operators:
  The bit operations `AND`, `OR` and `XOR` are denoted by `&`, `|` and `^`. Additionally `<<` performs a left shift, `>>` an arithmetic right shift and `>>>` a logical right shift.

- Negation:
  All values can be negated arithmetically by using `-` and all integers can additionally be negated logically via `~`.

- Instruction Pointer:
  The instruction pointer can be accessed by `@`.

- Ranges:
  One of the most important and interesting concepts of the language are the so called *ranges*. The range `X..Y` corresponds to `X`, `X+1`, ..., `Y` and the range `X:Y` corresponds to `X`, `X+1`, ..., `X+Y-1`. Thus, the first one specifies the beginning and end, whereas the second one specifies the beginning and the length.

- Fetches:
  The last group of operations are the so called *fetches*. Analogous to the notation used throughout this thesis, `M` accesses virtual memory, while `m` accesses physical memory. More precisely, `M[x]` denotes the 8 unaligned bytes at location `x`, `M1[x]` denotes the byte at location `x`, `M2[x]` denotes the wyde at wyde-aligned location `x`, `M4[x]` denotes the tetra at tetra-aligned location `x` and finally, `M8[x]` denotes the octa at octa-aligned location `x`. The value of the expression `m[x]` is the octa at octa-aligned physical memory location `x`. Additionally, one can access a local, global and special register `x` via `l[x]`, `g[x]` and `sp[x]`, respectively. Finally, dynamic register `x` can be specified by either `$[x]` or `$x`.

It should be noted, that all operations behave like defined in MMIX. That is, unsigned division behaves like `DIVU`, signed division like `DIV` and so on.

5.2 Commands

GIMMIX provides several commands to allow the user of the simulator to inspect or change the current state, debug and control a program. But of course, it is expected that more commands might be added in future. The following gives an overview of the most important commands and their functional principles.

5.2.1 Print and Set

`p [<fmt>] <expr..>...`	Print expression(s) in format `<fmt>`

The print command is one of the most important ones, because it allows it to print arbitrary values or entities of GIMMIX. The argument `<fmt>` is optional and can be `u` for decimal unsigned, `d` for decimal signed, `o` for octal unsigned, `x` for hexadecimal unsigned, `lx` for hexadecimal unsigned with 16 digits (default), `f` for float or `s` for string. For example, "`p $1`" prints the value of dynamic register 1 and "`p o 42`" prints the value 42 in octal base. The ".." after `expr` indicates that ranges are supported, while the "..." at the end indicates that multiple arguments of it can be given. That is, "`p 1..4`" would print the values 1, 2, 3 and 4 and "`p 1 2 3 4`" would do that as well. Additionally, besides printing values or entities of GIMMIX, another useful application of this command is to analyse different representations of values. That is, "`p x -8`" to see the hexadecimal representation of -8, "`p x 1.4`" for the hexadecimal integer representation for the float 1.4, "`p f #3FF<<52`" for the float that is represented by `#3FF0 0000 0000 0000` and so on.

`set <obj..> <expr..>`	Set `<obj>` to `<expr>`

The set command can be used to change the state, such as a register, a value at a specific memory location or the instruction pointer. For example, "`set $[1..3] 10..12`" would set `$1` to 10, `$2` to 11 and `$3` to 12.

5.2.2 Execution of Instructions

`s [<count>]`	Execute one or `<count>` instruction(s)
`c [<count>]`	Continue one or `<count>` time(s)
`ou [<count>]`	Step out of current function one or `<count>` time(s)
`ov [<count>]`	Step over next instruction one or `<count>` time(s)

To control the running program, the CLI offers the commands *step* (s), *continue* (c), *stepout* (ou) and *stepover* (ov), whereas stepout means that GIMMIX continues until the current function is left via `POP`. If the next instruction is not `PUSHJ` or `PUSHGO`, stepover will execute only this instruction. Otherwise it will continue until the called function is left by `POP`. That is, it "steps over" a function call.

`d [<addr..>]`	Disassemble instruction(s) at `@` or `<addr>`

The disassemble command interprets either the 16 following tetras at the instruction pointer or the tetra(s) at `<addr>` as instructions and prints their name and operands. For example, "`d @:16`" would disassemble the next four instructions at the PC.

`b v <mode> <addr>`	Break if `<addr>` (virt.) is accessed for `<mode>` (rwx)
`b p <mode> <addr>`	Break if `<addr>` (phys.) is accessed for `<mode>` (rwx)
`b e <expr..>`	Break if the value of `<expr>` changed
`b <addr>`	Break if `<addr>` (virt.) is accessed for x

Of course, one has to be able to set breakpoints. GIMMIX provides a very general facility to do so. Because the break command `b` can be used to stop the CPU as soon as a specific virtual or physical address is accessed for a specific mode. That is, when specifying `x` as mode, it stops the CPU as soon as an instruction should be executed at that position; i. e. this would be an ordinary breakpoint. But one can also stop the CPU as soon as a specific memory location is read or written. Additionally, the third version of it offers the opportunity to set expression breakpoints. That means, with "`b e $1`" the CPU would stop as soon as the value of `$1` has changed and, as the command description indicates, ranges are supported as well. The final variant of this command is only an abbreviation for "`b v x <addr>`".

`e [<flags>]`	Print effects of last instruction

To be able to see all effects that an instruction caused, the command `e` can be used. It optionally takes an argument with flags, which are any combination of `s`, `d`, `c`, `v` and `f`. By default, only `s` is used. The flag `s` prints all reads and writes from/to the stack. Flag `c` prints fills and write-backs of the caches, `d` prints accesses to the devices and `v` prints virtual address translations. Finally, `f` specifies that effects, that occur during the fetch phase, are printed as well. For example, a typical output of the command could be:

```
8000000000001008: SETH    $1,#6000    : rL=2,$1=l[1] = #6000000000000000
```

That means, the last executed instruction at location `#8000 0000 0000 1008` was `SETH $1,#6000`. Behind the disassembled instruction, the effects are displayed. In this case, `$1`, currently stored in $l[1]$, has been set to `#6000 0000 0000 0000` and `rL` has been increased to 2.

5.2.3 Examining the State

`itc [<addr..>]`	Print all ITC entries or search for `<addr>`
`dtc [<addr..>]`	Print all DTC entries or search for `<addr>`
`ic [<addr..>]`	Print all IC entries or search for `<addr>`
`dc [<addr..>]`	Print all DC entries or search for `<addr>`

Of course, the user of the machine should also have the opportunity to print the current content of the caches. Therefore, `itc` prints the instruction translation cache, `dtc` the data translation cache, `ic` the instruction cache and `dc` the data cache.

dev	Print the state of the attached devices

Especially when developing or porting an operating system, it is very helpful to see the current state of all devices. Thus, GIMMIX offers this instruction, which prints all devices with their memory range, interrupt mask and the current device register contents.

tr <cmd> [<arg>...]	Execute the given command after each CPU stop

Another important command is the trace command. It can be used to execute an arbitrary command everytime the CPU stops, i. e. as soon as a step, continue or similar is finished. The most frequent usage is probably `"tr e"` to display the effect of each instruction. But it can also be used to track the values of registers `$4`, `$5` and `$6` via `"tr p $[4..6]"`, for example.

5.3 Implementation

After the conceptual description of the command line interface, the most important points of its implementation should be explained.

5.3.1 General Structure

The following FMC diagram shows the interior of the last black box, displayed in the overview diagram at the beginning:

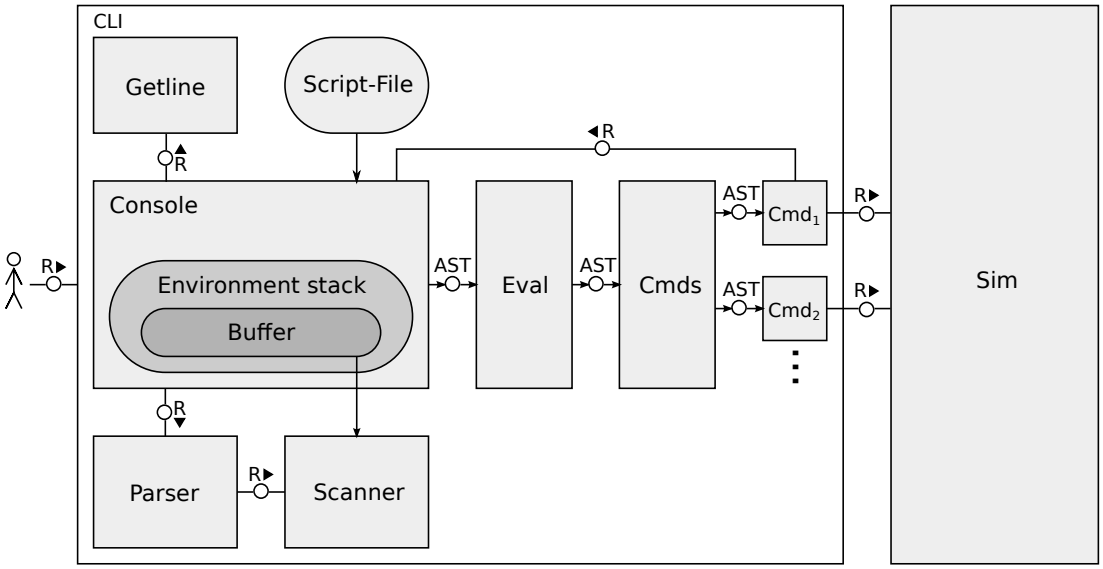


Figure 3: Architecture of the CLI in FMC notation

Conceptually, the user interacts with the CLI, whose main module is the *console*. Depending on whether the console is used interactively or not, it either uses the *getline* library[5] to read a line from the command line into the buffer or reads a specified script file into the buffer. Afterwards this input is parsed to construct an abstract syntax tree (AST). The scanner and parser for this task have been generated by flex and Bison, respectively. As soon as the AST is present, it is passed to the *eval* module, which

[5] A small library, that allows basic line editing and provides a history, written by Chris Thewalt. It can be found in the folder **lib/getline** of GIMMIX.

executes it. If a node that represents a command is found, the module *cmds* comes into play. It holds a table with all available commands and will execute the corresponding function to execute the command. The commands will typically control the simulator, read the state or manipulate it.

As the FMC diagram indicates, the console manages a stack of *environments* to allow nesting. Each line or script execution will get a new environment on the top of the stack. This way, a command may use the console to execute a line or script as well. For example, the command `tr` uses this feature by storing the desired command as a string and executing it later with the console.

5.3.2 Command Infrastructure

Because it is expected that the CLI language itself will not change much in future, but the commands will probably be changed and – most important – new ones will be added, it has been decided to separate cleanly between the language and the commands. Additionally, implementing and adding new commands should be as simple as possible. Similarly to the device infrastructure of GIMMIX, a compromise between dynamic and simplicity has been selected. That is, the module cmds keeps a table with all commands, but each command is implemented in its own file. Thus, to add a new command, a file has to be added and the table has to be extended.

Each command has an *execution function*, which receives the number of arguments and an array of AST nodes – one for each argument. Additionally, the cmds module provides the function `cmds_evalArg`:

```
sCmdArg cmds_evalArg(const sASTNode *arg,int expTypes,octa
    offset,bool *finished);
```

The function receives the argument to evaluate, the types that are supported (integer, float or string), an offset and a pointer to a boolean. The last two are used to implement ranges. Since the language is intended for scripting as well and it is imaginable that a user would like to, e. g. , execute 1000 instructions and pipe the contents of the first 64 megabytes of memory into a file to analyze it with other tools later, it has been decided not to simply convert a range to an array. Because as the example shows, it might cost a lot of time and memory. Instead, the evaluation function receives the current offset and tells the caller whether all ranges are already finished. That means, in each step one value of each range is extracted, depending on the current offset.

The return value of `cmds_evalArg` is a struct called `sCmdArg`, that provides all required information about the value of the argument for the caller. It contains the type (integer, float or string), the origin (`M8`, `l`, `$`, and so on or an arbitrary expression such as `1+2`), the location for origins like virtual memory or registers (that is, the memory address or register index) and of course the value of the argument as integer, float or string. The location and origin are for example used by `p` to print the origin of an argument or by `set` to set the corresponding entity of GIMMIX.

5.3.3 Command Implementation

A typical command, that makes use of ranges, is *disassemble*. Ignoring the details of the disassembling process, its implementation looks like the following:

```
void cli_cmd_disasm(size_t argc,const sASTNode **argv) {
    if(argc > 1)
```

```
        cmds_throwEx(NULL);
    if(argc == 0) {
        for(int i = 0; i < DEFAULT_INSTR_COUNT; i++)
            doDisasm(cpu_getPC() + i * sizeof(tetra));
    }
    else {
        sCmdArg a;
        bool fin = false;
        for(octa off = 0; ; off += sizeof(tetra)) {
            a = cmds_evalArg(argv[0],ARGVAL_INT,off,&fin);
            if(fin)
                break;
            doDisasm(a.d.integer & -(octa)sizeof(tetra));
        }
    }
}
```

Listing 3.27: Implementation of command `d`

As the listing shows, at first the number of arguments is checked and if no arguments are given, `DEFAULT_INSTR_COUNT` (16) instructions at the PC will be disassembled. If one argument is given, `cmds_evalArg` will be called until `fin` has been set to `true`, whereas in each step the offset `off` will be increased to reach the next instruction. Thus, "`d @:16`" would disassemble the instructions at addresses `@`, `@+4`, `@+8` and `@+12`.

Another interesting example, that utilizes the origin and location, is the command `set`. The slightly shortened implementation is:

```
void cli_cmd_set(size_t argc,const sASTNode **argv) {
    if(argc != 2)
        cmds_throwEx(NULL);
    for(octa off = 0; ; off++) {
        bool oFin = false, vFin = false;
        sCmdArg obj,val;
        obj = cmds_evalArg(argv[0],ARGVAL_INT,off,&oFin);
        val = cmds_evalArg(argv[1],
            ARGVAL_INT | ARGVAL_FLOAT,off,&vFin);
        if(oFin)
            break;

        switch(obj.origin) {
            case ORG_VMEM1:
                mmu_writeByte(obj.location,val.d.integer,0);
                break;
            ...
            case ORG_EXPR:
                cmds_throwEx("Can't set arbitrary expr\n");
                break;
        }
    }
}
```

Listing 3.28: Implementation of command `set`

That means, it iterates over both the objects to set and the values at the same time, until the last object has been assigned. Additionally, the origin is used to determine what entity of GIMMIX should be changed. Of course, `set` is not possible for expressions like `1*2+M[0]` as first argument. It is only allowed for the instruction pointer and for fetches, that are specified without any operator in the "outmost layer", i. e. `M4[@+4]` for example.

Chapter 4

Test System

As already mentioned in the introduction, it is very important that GIMMIX behaves correctly according to the MMIX specification. Additionally, since not all details are defined by the specification and to increase the confidence that everything has been understood and implemented correctly, the behaviour of GIMMIX is compared to the one of MMIX-SIM and MMIX-PIPE.

For this reason, a sophisticated test system has been built, that consists of two parts. At first, test programs for MMIX are executed on as many of the three simulators as possible and their results are compared. Second, unit tests are used to test parts of GIMMIX, that are not visible to the software. This chapter describes the ideas behind both systems and the most important facilities that are required.

1 Program Tests

The basic idea behind the program tests is to let the simulators execute some instructions, that put values into registers and/or write to specific memory locations, and check afterwards if the registers and memory locations have the expected content.

1.1 Test Infrastructure

Each test consists of a *mms-file*, that holds the assembly code to test, and a *test-file*, that specifies the expected results. The first line of the test-file describes the values to compare, while the rest of the file is matched against the produced values. The first line is passed to the *post commands* module, which interprets it and prints the desired values. The line consists of an arbitrary number of commands, separated by `","`. Each command can either be `r:$X..$Y` to print dynamic registers `$X`, `$(X + 1)`, ..., `$Y` or `m:X..Y` to print the octas between physical memory addresses `X` and `Y`, inclusively.

MMIX-SIM, MMIX-PIPE and GIMMIX have been extended to support these post commands. They can be specified via command line argument to urge the simulator to execute them after the execution of the program is finished. This way, the contents of the specified registers or memory locations are printed to `stdout`. GIMMIX provides the shellscript **runtests.sh**, that executes all test programs on the simulators, whereas the post commands are extracted from the first line of the test-file. Afterwards, the output of GIMMIX is compared to the expected output (all lines of the test-file, except the first one), to the output of MMIX-PIPE and to the output of MMIX-SIM. If there are any differences, the tool diff[1] will be used to illustrate them. Additionally, it distinguishes

[1] Diff is a comparison utility, that finds and displays differences between two files. [19]

between the following subfolders of folder **tests**, in which all program tests reside:

- **user**: This directory contains tests, that can and should be executed in user mode. All three simulators have an option to execute a program in user mode, for which an initial environment is supplied. That is, paging is pre-configured by setting `rV` and PTEs all installed for all four segments. Furthermore, `rS`, `rO`, `rK`, `rT` and `rTT` are initialized correspondingly. Most important, all tests in this folder are executable with all three simulators, i. e. also with MMIX-SIM.

- **kernel**: This directory holds the tests, that should be executed in privileged mode. Therefore, they do not expect this initial environment, but start without any pre-configuration and in privileged mode. These tests are executed with MMIX-PIPE and GIMMIX only.

- **diff**: To test some behaviours that are implementation-defined, to test the fix for the page fault problem on the stack or to test cases that are currently erroneous in MMIX-PIPE or MMIX-SIM, the folder **diff** is used. All tests in it are executed on GIMMIX only.

- **cli**: Finally, this directory holds tests for the command line interface. Thus, the tests are typically no programs for MMIX, but scripts for the CLI. Of course, they are executed in GIMMIX only.

All other folders are ignored by **runtests.sh**. But it is noteworthy, that the folder **manual** holds some programs for GIMMIX, that make use of the yet existing devices. These are not included into the program test system, because they are not automatically testable that easily.

1.2 Test Programs

Although the tests programs can not be explained in detail, this section will give an overview about them.

1.2.1 User Tests

Since all three simulators can execute the user tests, as many cases as possible are tested in this way. Most of them simply test the behaviour of a set of instructions. While a part of the tests have been written manually, all tests, in which the individual test cases work in a common pattern, are generated by scripts or programs. All these generators are placed in the directory **testgen**, which contains two groups of generators. The first group are Ruby scripts, that are used for most of the tests. Ruby has been chosen, because it allows a quick development of the tests and provides an arbitrary-precision arithmetic, independend of the underlying platform (in other scripting languages, such as PHP, the width of integers depends on the platform, on which the interpreter runs [20]). This way, e. g. bit operations with 64-bit integers can be performed, regardless of whether the generator is run on a 32-bit or 64-bit platform. The second group are C programs, that are primarily used for the floating point arithmetic tests. Because this way, the double precision arithmetic of a well tested language and platform can be utilized to make sure, that the implementation of the floating point instructions in GIMMIX is correct. Ruby is not well suited in this case, because of the arbitrary precision arithmetic; instead C fits better, because of the amount of control and the

similarities of double precision arithmetic and 64-bit float arithmetic in MMIX. Additionally it is noteworthy, that some of these test generators expect x86 as underlying platform, because the rounding mode of the x87 floating point unit is set and some reactions on not defined details by IEEE-754 and therefore x87-specific behaviour are necessary. For example, the sign-bit of NaN results is not defined by the standard [21, pg. 17].

1.2.2 Kernel Tests

Most of the tests in the **kernel** directory have been written manually. They test the PEs of MMIX, interrupts, paging and privileged instructions. But some of them are generated automatically as well. Most important, the paging tests are generated automatically with a Ruby script, that generates PTPs, PTEs and access tests for a given value of `rV`.

1.2.3 Diff Tests

As already mentioned, MMIX-PIPE and GIMMIX differ in a few details. For example, the values of `rXX`, `rYY` and `rZZ` are not equal for some PEs, because MMIX does not define it completely. Another example is, that GIMMIX will trigger an exception if a not existing memory location in the I/O space is accessed or a bit in `rQ` is set, that is not used for an interrupt, PE or ME, while MMIX-PIPE will not do that. The second reason for this directory is, that the fix for the page fault problem on the stack has to be tested, which of course does not work in MMIX-PIPE.

1.2.4 CLI Tests

Finally, the directory **cli** contains scripts to test the language and commands of the CLI. In this case, obviously, the expected results are only compared with the actual results. Currently, the expressions of the language, handling of whitespace and the most critical and important commands such as `p`, `set`, `e`, `b`, `d`, `tr` and a few others are tested.

1.3 Code Coverage

To ensure that all lines in the important parts of the simulator are executed by the tests, a small code coverage system has been developed. The basic functional principle is to cause gcc to let the simulator generate information for gcov[2] during the execution of the simulator. The shellscript **runcov.sh** runs the simulator for a specific or for all tests and uses gcov to produce the coverage information. This information is analyzed by a few Ruby scripts to generate HTML pages, that display the results. These pages contain a list of all tests, whereas each of them displays the source files of the simulator with the percentage of executed lines. Additionally, one HTML page per source file is generated, displaying the source code and highlighting the lines correspondingly, depending on whether they have been executed or not.

[2] Gcov is a test coverage program, that can be used together with gcc to improve the performance of a program and discover untested parts of it. [22]

2 Unit Tests

The second part of the test system consists of unit tests. As already said, they are used to test the parts of GIMMIX, that are not visible to the software or would be difficult to test. To achieve that, a small unit test framework has been developed. Although existing frameworks like CUnit[3] or Check[4] have been considered, it has been decided to built an own framework, because the existing ones are quite heavy and most of their functionality would not be used anyway. Additionally, of course they do not provide special functions to compare bytes, wydes, tetras and octas. Thus, by building our own framework, it can be designed to fit our needs exactly.

The test framework found in **unittests/test.c** allows it to register and run tests, and offers functions to assert equalities. The function `test_register` is intended to register a *test suite*, which contains one or more *test cases*. Each test case has a name and a function, that runs that test case. After registering all test suites, `test_start` can be called to execute all test cases in all test suites. To support a convenient and informative facility for asserts, it offers the macros `test_assertTrue`, `test_assertInt`, `test_assertByte`, `test_assertOcta` and so on, which call the actual assert function with the current function name and line number. This way, reasons of failures can be identified and fixed more quickly.

Currently, tests suites exist for testing the physical memory (independent of the rest of the memory hierarchy), the complete memory hierarchy (focused on the different flags such as `MEM_UNCACHED` or `MEM_SIDE_EFFECTS`), the system instructions like `LDUNC`, `SYNCD`, `SYNC` and so on and the interruptibility of instructions.

[3] Available at http://cunit.sourceforge.net.
[4] Available at http://check.sourceforge.net.

Chapter 5

Future Possibilities

The current state of the GIMMIX project is, as described in the previous chapters, that the simulator itself including a few basic devices is completely finished. That is, the simulator realizes the entire MMIX architecture. Additionally, a convenient and powerful command line interface exists and the whole system has been tested as much as possible to reach the confidence, that everything works as intended. This final chapter mentions yet missing parts to reach the goal of porting an operating system to MMIX and suggests a few possible enhancements.

1 Missing Parts for an Operating System Port

At first, the yet missing tools and changes for developing or porting an operating system for MMIX are listed.

1.1 `TRAP 0,0,0` halts the Simulator

To allow automatized tests, the simulator has to be stoppable in some way. Since MMIX-SIM uses a `TRAP` with only zeros as arguments for a quit command, GIMMIX does so as well. Additionally, MMIX-PIPE has been changed, so that it stops for `TRAP 0,0,0`, too. Of course, this is only temporary, because MMIX defines that a `TRAP 0,0,0` terminates a user process. That is, the operating system should handle that trap – like all others as well – and the simulator should not stop. Hence, to be able to develop or port an operating system, this "feature" has to be removed or replaced with something else.

1.2 Toolchain

Currently, GIMMIX does not provide its own assembler and has no C compiler and linker at all. Instead, it uses the assembler *mmixal*, written by Donald Knuth, to produce special MMIX object files. These in turn are converted by a tool to an ASCII file, that specifies which values should be written to which physical memory addresses. GIMMIX interprets this format to load a program into the main memory. Of course, this is only a temporary solution. Later, a ROM should be put into the simulator, which initiates the boot process. Although it is imaginable that programs can still be loaded directly into GIMMIX for testing purposes, these will probably not be specified in that ASCII format, but in a binary format. MMIX-PIPE understands the ASCII format as well and MMIX-SIM is able to load the MMIX object files, which is one of the reasons why GIMMIX uses this solution currently.

As already mentioned, the project uses no C compiler yet. By now, a GNU toolchain (gcc cross-compiler, binutils[1] and newlib[2]) for MMIX[3] is already available due to the efforts of Hans-Peter Nilsson, which could be used in future. Another opportunity would be to build a backend for the lcc[4], which has already been started in the previous GIMMIX approach, but is not yet finished.

1.3 Startup and Tools

Although GIMMIX already provides a ROM and a disk device, tools for disk creation, partitioning, filesystem creation and so on are still missing. The previously mentioned project ECO32 offers tools for such tasks, which could be used for GIMMIX as well.

2 Extensions and Enhancements

As mentioned, the simulator is considered complete, but there are of course still imaginable improvements and additional features. These are listed in this section.

2.1 More Devices

The currently provided devices are only the basic ones, which have been added primarily to ensure that the device infrastructure is sufficient. Therefore, a few more will have to be added in future. For example, every reasonable operating system will need a screen and a keyboard, instead or additional to the already existing terminals. Furthermore, a real-time clock would be a useful device. A screen and a keyboard are for example present in the ECO32 project and could thus be ported to GIMMIX.

2.2 Working with Symbols in the CLI

Since the loading format used in GIMMIX is not considered final, the CLI does not offer the opportunity to use symbols instead of addresses at the moment. That is, for example breakpoints can only be set by specifying virtual or physical addresses, but not via symbols, that have been assigned to virtual addresses. Of course, this would simplify the debugging process.

2.3 Interface to GDB

As soon as programs for MMIX can be written in C, for example, it would be more convenient to debug such programs in the language they have been written in, instead of having to work with the generated assembly. To do so, GIMMIX could provide an interface for GDB, that offers the opportunity to control GIMMIX with GDB. Fortunatly, an alpha version of GDB for MMIX[5] is already available because of the work of Mr. Ruckert. Therefore, as soon as the GDB interface in GIMMIX is present, it should be possible to use that version of GDB to debug programs running on GIMMIX.

[1] GNU binutils is a collection of tools for analyzing binary files, building archives or stripping symbols from a file [23].

[2] Newlib is a small C library, that is intended for embedded systems [24].

[3] The GNU toolchain is available at http://www.bitrange.com/mmix.

[4] Lcc is a retarget compiler for ANSI C [25].

[5] The complete GNU toolchain for MMIX including GDB is available at http://math.cs.hm.edu/mmix/examples/MMIXonMMIX/index.html.

2.4 Infrastructure for MMIX Programs without OS

To be able to run all example programs for MMIX – especially those that are or will be listed in the volumns of the The Art of Computer Programming books about MMIX – directly in GIMMIX, i. e. without an operating system, a few additions to GIMMIX are necessary. These are:

- Those programs expect a rudimentary operating system, that provides special system calls via forced traps to perform I/O requests. That is, opening a file, reading a file, writing to a file and so on, which can for example be used to write to `stdout` or read from `stdin`. In contrary to GIMMIX, MMIX-SIM and MMIX-PIPE offer these system calls.

- Additionally, mmixal allows it to pre-allocate global registers and initialize them with specific values. This feature is currently not usable, because the ASCII file format does not support that. MMIX-SIM (and MMIX-PIPE, when started in user mode), for example, starts with the instruction `UNSAVE` to establish a part of the user environment, including the requested global registers.

- To allow more interaction with the host platform, MMIX-SIM provides the number of arguments, given to itself, in `$0` and a pointer to the first one in `$1`. Additionally, feedback can be passed back by putting a value in `$255`, which will be returned by the main function of MMIX-SIM.

These features are not present in GIMMIX at the moment, because it has a different goal than MMIX-SIM and MMIX-PIPE. Nevertheless, it might make sense to provide such facilities as well, e. g. requested by a command line argument, to allow the execution of all the programs for MMIX, that require them.

2.5 Acid Test Mode

When developing or porting an operating system, a deterministic machine is appreciated in most cases. That means, that the behaviour of the machine is always exactly the same, when the start conditions do not change. This is especially a problem when working with real hardware, because for example the timing of devices varies. Of course, a deterministic behaviour simplifies debugging, since errors are reproducible. But in some cases, it might be helpful to produce non-deterministic behaviour or random start conditions on purpose, to make sure that the OS works correctly in these cases as well or to detect errors, that would not have arisen otherwise. GIMMIX could help by providing an "acid test mode", which initializes registers, main memory and so on with random values, randomizes the timing behaviour of devices or similar.

2.6 Graphical User Interface

As already mentioned, a graphical user interface for GIMMIX could be provided. Since the simulator core is completely independent of the current command line interface, it could be easily exchanged with a GUI or both could coexist. A GUI would have the advantage, that many entities of MMIX such as the dynamic registers, special registers, main memory and so on could be displayed simultaneously. Additionally, the disassembly of the previous and next few instructions could be displayed as well. Thus, it would allow a more productive interaction with the simulator.

2.7 Provide Information about Hardware for Software

Currently, operating systems for MMIX are very dependent on the particular MMIX implementation they are designed for. Because, the devices, the amount of main memory, the cache configuration and so on, is either not investigatable at all or only with a lot of effort. Therefore, it could make sense to add a kind of "meta-device", that provides that information at specific addresses in the I/O space. This way, the OS could react dynamically on present devices, find out the amount of main memory easily and make more efficient use of caches. Of course, this meta-device should be part of the MMIX specification, so that all MMIX implementations provide it.

2.8 Mapping of the I/O Space

Unfortunatly, at the moment MMIX does not allow to map the I/O space via paging. That is, the I/O space can only be accessed in privileged mode by using the directly mapped space. The reason is, that the physical address in a PTE is only 48 bit wyde, which does not include the I/O space. This will be a problem for microkernel operating systems, because it is not possible to implement drivers as user processes. A solution might be to limit the I/O space to range from `#0001 0000 0000 0000` to `#00FF FFFF FFFF FFFF` and extend the width of the physical address in a PTE to 56 bits. This way, the upper 8 bits of a PTE would still remain for the operating system and the I/O space should still be large enough as well. Additionally, the width of a translation in the TCs would have to be extended. These changes would allow the operating system to map pages from the I/O space into the user space, so that user processes can access it.

Glossary

AST is the acronym for "abstract syntax tree", a tree representation of the syntactic structure of source code written in a language [26]. 76, 77

Bison is a general-purpose parser generator, which uses a specification of a context-free grammar to construct for example a C source file [27]. 76

C89 and C99 are both standards of the programming language C. C89 has been standardized in 1989 by the American National Standards Institute (ANSI) and is also known as ANSI C. [28] C99 on the other hand, has been published by ISO/IEC in 1999 and has been adopted as an ANSI standard in May 2000 [29]. 44, 51

CISC is the acronym for "complex instruction set computer", which can – in contrast to RISC – typically perform several operations with a single instruction and supports complex addressing modes [30]. 7, 90

Donald Knuth is a computer scientist and Professor Emeritus at Stanford University, who is famous for the creation of TEX, METAFONT, CWEB and The Art of Computer Programming [31]. 7, 9, 91

EBNF is the acronym for "Extended Backus-Naur Form", which is a family of notations for expressing context-free grammars, an extension of the basic Backus-Naur Form (BNF) [32]. 73, 93

Endianness refers to the ordering that is used to store bytes in external memory. The most important ones are *big-endian* and *little-endian*. The former stores the most significant byte at the lowest address, while the latter stores the most significant byte at the highest address [33]. 7, 9, 17, 71

Exception is a term, that has to be used for two different concepts in this thesis, due to the common understanding in both cases. At first, it refers to an extraordinary condition in MMIX, which indicates, that an instruction can not be executed at all or a special case arised. MMIX distinguishes between arithmetic exceptions (AE) like division by zero or integer overflow, which are handled by the user application, program exceptions (PE) such as privileged instruction or protection fault, which are handled by the operating system, and machine exceptions (ME) like power failure, which are as well handled by the OS. Second, the term refers to the exception concept in the code of GIMMIX via `setjmp` and `longjmp`. In this thesis, the context or the exact term that is used, will clarify what is meant. 7, 8, 10–15, 20, 27–31, 37–43, 45, 46, 48, 50, 51, 53–56, 59–64, 66, 70, 83

flex is a tool for generating scanners, which in turn are programs that recognize lexical patterns in text. The scanner to generate is described in pairs of regular expressions and C code, from which a C source file is generated [34]. 76

FMC is the acronym for "Fundamental Modeling Concepts" and is a general notation to communicate the concepts and structure of complex informational systems in an efficient way. The basic elements are *agents*, displayed as rectangular nodes, and *storages*, displayed as rounded nodes. Agents communicate with each other over *channels* and read from or write to storages [35]. 45, 47, 76

gcc is the acronym for "GNU C Compiler", which belongs to the GNU Compiler Collection (GCC), a compiler system produced by the GNU Project for various programming languages. [36]. 44, 51, 83, 85

GDB is the GNU Project debugger, that allows it to analyze the behaviour of a program, while it is executed, or its state at the moment it crashed [37]. 5, 86

Immediate value is a value utilized by an instruction, that is directly present in the bytes that encode the instruction. Thus, it has not to be loaded from a register or from memory, but is immediatly available, hence the name. 10–12, 18, 19, 21, 28

Interrupt is a asynchronous signal, that is triggered by the hardware to communicate some kind of event to the software. In MMIX, interrupts raise a dynamic trap. 7, 8, 12, 36–38, 42, 43, 50, 56, 59, 60, 70, 71, 76, 83

Paging is a memory-management mechanism, that provides a virtual space, additionally to the physical space, divides both into *pages* and allows a mapping from virtual pages to physical pages. This way, processes can be separated from each other and the physical memory used by a process has not to be contiguous. 7, 8, 22, 23, 82, 83

PC is the acronym for "program counter", also called "instruction pointer", and in GIMMIX it denotes the location in memory, from which the next instruction will be fetched. 10, 12, 19, 32, 48, 50, 52, 60–62, 73–75, 79

PHP is the recursive acronym for "PHP: Hypertext Preprocessor" and "a widely-used general-purpose scripting language that is especially suited for Web development" [38]. 82

Pipelining is a concept used in the design of computers to allow an overlapping execution of multiple instructions and thus to increase the number of instructions, that can be executed in a unit of time. That is, the execution is divided into stages, whereas each stage processes one instruction at a time [39]. 7, 8, 43

RISC is the acronym for "reduced instruction set computer", which aims – in contrast to CISC – to provide rather simple instructions, that can be executed very fast [40]. 7, 9, 71, 89

Ruby is "a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write" [41]. 82, 83

Subroutine linkage is a term used to denote the mechanism for calling subroutines (or functions; both words are used interchangeably in this thesis) and returning from them. 19, 31, 32

TEX is a typesetting system written by Donald Knuth, intended for the creation of beautiful books [42]. 7, 89

The Art of Computer Programming or short TAOCP is the famous book series, written by Donald Knuth, that covers many kinds of programming algorithms and their analysis. The examples in the book are written in the MIX assembly language, but might be expressed in MMIX assembly language in the near future, because currently, MIX is replaced by MMIX [43]. 7, 86, 89

Unit testing is a test method by which individual units of source code are tested in an automatic way. [44] Typically, a unit test framework is used to simplify the process of writing and running the tests. 81, 83

Listings

3.1 Executing an instruction (slightly shortened) . . . 48
3.2 Examples of the execution functions . . . 49
3.3 Execution function of `CSWAP` . . . 50
3.4 Implementation of `reg_set` . . . 52
3.5 Implementation of `reg_push` . . . 53
3.6 Implementation of `reg_pop` . . . 54
3.7 Trap handler, using the extended `SAVE` and `UNSAVE` . . . 55
3.8 Implementation of `reg_save`, part 1 (partially pseudo-code) . . . 57
3.9 Implementation of `reg_save`, part 2 . . . 58
3.10 Implementation of `reg_save`, part 3 (partially pseudo-code) . . . 58
3.11 Implementation of `reg_unsave`, part 1 (partially pseudo-code) . . . 58
3.12 Implementation of `reg_unsave`, part 2 (partially pseudo-code) . . . 59
3.13 Implementation of `reg_unsave`, part 3 . . . 59
3.14 Determining `rX`/`rXX` in `cpu_triggerException`, part 1 . . . 61
3.15 Determining `rX`/`rXX` in `cpu_triggerException`, part 2 . . . 61
3.16 Determining `rX`/`rXX` in `cpu_triggerException`, part 3 . . . 61
3.17 The implementation of `RESUME` (partially pseudo-code) . . . 62
3.18 Implementation of the "resume again" action . . . 63
3.19 Implementation of `mmu_readTetra` . . . 64
3.20 Implementation of `mmu_doRead` . . . 64
3.21 Implementation of `mmu_doWrite` . . . 65
3.22 Implementation of `mmu_translate` (partially pseudo-code) . . . 65
3.23 Implementation of `cache_read` . . . 68
3.24 Implementation of `cache_get` . . . 68
3.25 Implementation of `bus_read` . . . 70
3.26 Grammar of expressions in EBNF . . . 72
3.27 Implementation of command `d` . . . 77
3.28 Implementation of command `set` . . . 78

Bibliography

[1] Knuth, D.E. The Art of Computer Programming - Fascicle 1 - MMIX. http://www-cs-faculty.stanford.edu/~knuth/fasc1.ps.gz. Last Accessed: 05/12/2011.

[2] GNU MDK, Revised First Edition. http://shop.fsf.org/product/gnu-mdk. Last Accessed: 05/12/2011.

[3] Knuth, D.E. *The Art of Computer Programming Volume 1 - Fundamental Algorithms*. Addison-Wesley, third edition, 1997. ISBN 978-0-201-89683-1.

[4] Knuth, D.E. MMIX-SIM. http://www-cs-faculty.stanford.edu/~uno/programs/mmix-20110305.tar.gz. Last Accessed: 05/05/2011.

[5] Knuth, D.E. *MMIXware: a RISC computer for the third millennium*. Lecture notes in computer science. Springer, 1999. ISBN 9783540669388. URL http://books.google.de/books?id=RJmYNq0KQnQC.

[6] Knuth, D.E. Definition of architecture details. http://www-cs-faculty.stanford.edu/~uno/programs/mmix-20110305.tar.gz. Last Accessed: 04/16/2011.

[7] Leijen, Daan. Division and Modulus for Computer Scientists. http://research.microsoft.com/en-us/um/people/daan/download/papers/divmodnote-letter.pdf. Last Accessed: 04/20/2011.

[8] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M. http://www.intel.com/Assets/PDF/manual/253666.pdf. Last Accessed: 04/20/2011.

[9] Heidi Anlauff and Axel Böttcher and Martin Ruckert. *Das MMIX-Buch - Ein praxisnaher Zugang zur Informatik*. Springer-Verlag, 2002. ISBN 3-540-42408-3.

[10] GNU C Language Extensions. http://tigcc.ticalc.org/doc/gnuexts.html#SEC72. Last Accessed: 04/26/2011.

[11] Ubuntu Manpage: stdint.h - integer types. http://manpages.ubuntu.com/manpages/maverick/man7/stdint.h.7posix.html. Last Accessed: 04/26/2011.

[12] C99 - Implementations. http://en.wikipedia.org/w/index.php?title=C99&oldid=425924364#Implementations. Last Accessed: 04/26/2011.

[13] GCC Releases. http://gcc.gnu.org/releases.html. Last Accessed: 04/26/2011.

[14] Status of C99 features in GCC 3.0. http://gcc.gnu.org/gcc-3.0/c99status.html. Last Accessed: 04/26/2011.

[15] ISO/IEC 9899:TC3 - Committee Draft - September 7, 2007. http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf. Last Accessed: 04/29/2011.

[16] ECO32 Project Homepage. http://homepages.fh-giessen.de/~hg53/eco32/. Last Accessed: 05/05/2011.

[17] Treap. http://en.wikipedia.org/w/index.php?title=Treap&oldid=422108088. Last Accessed: 05/05/2011.

[18] XTERM(1) manual page. http://www.xfree86.org/current/xterm.1.html. Last Accessed: 05/06/2011.

[19] diff(1) - Linux man page. http://linux.die.net/man/1/diff. Last Accessed: 05/14/2011.

[20] PHP: Integers - Manual. http://www.php.net/manual/en/language.types.integer.php. Last Accessed: 05/09/2011.

[21] IEEE Standard for Binary Floating-Point Arithmetic. http://kfe.fjfi.cvut.cz/~klimo/nm/ieee754.pdf. Last Accessed: 05/09/2011.

[22] Gcov intro - Using the GNU Compiler Collection (GCC). http://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html. Last Accessed: 05/09/2011.

[23] GNU Binutils. http://www.gnu.org/software/binutils/. Last Accessed: 05/11/2011.

[24] The Newlib Homepage. http://www.sourceware.org/newlib. Last Accessed: 05/11/2011.

[25] lcc, A Retargetable Compiler for ANSI C. http://sites.google.com/site/lccretargetablecompiler/. Last Accessed: 05/11/2011.

[26] Abstract syntax tree. http://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=426689149. Last Accessed: 05/14/2011.

[27] Bison - GNU parser generator. http://www.gnu.org/software/bison. Last Accessed: 05/14/2011.

[28] ANSI C. http://en.wikipedia.org/w/index.php?title=ANSI_C&oldid=425328214. Last Accessed: 05/15/2011.

[29] C99. http://en.wikipedia.org/w/index.php?title=C99&oldid=428474931. Last Accessed: 05/15/2011.

[30] Complex instruction set computing. http://en.wikipedia.org/w/index.php?title=Complex_instruction_set_computing&oldid=421524452. Last Accessed: 05/14/2011.

[31] Donald Knuth. http://en.wikipedia.org/w/index.php?title=Donald_Knuth&oldid=428319430. Last Accessed: 05/12/2011.

[32] Extended Backus-Naur Form. http://en.wikipedia.org/w/index.php?title=Extended_Backus%E2%80%93Naur_Form&oldid=419163293. Last Accessed: 05/14/2011.

[33] Endianness. http://en.wikipedia.org/w/index.php?title=Endianness&oldid=433505641. Last Accessed: 06/11/2011.

[34] flex: The Fast Lexical Analyzer. http://flex.sourceforge.net. Last Accessed: 05/14/2011.

[35] FMC - Quick Introduction to Fundamental Modeling Concepts. http://www.fmc-modeling.org/quick-intro. Last Accessed: 05/14/2011.

[36] GNU Compiler Collection. http://en.wikipedia.org/w/index.php?title=GNU_Compiler_Collection&oldid=426957812. Last Accessed: 05/14/2011.

[37] GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb. Last Accessed: 05/14/2011.

[38] PHP: Hypertext Preprocessor. http://www.php.net. Last Accessed: 05/14/2011.

[39] Instruction pipeline. http://en.wikipedia.org/w/index.php?title=Instruction_pipeline&oldid=425081992. Last Accessed: 05/15/2011.

[40] Reduced instruction set computing. http://en.wikipedia.org/w/index.php?title=Reduced_instruction_set_computing&oldid=427914688. Last Accessed: 05/14/2011.

[41] Ruby Programming Language. http://www.ruby-lang.org/en. Last Accessed: 05/14/2011.

[42] TeX Frequently Asked Questions. http://www.tex.ac.uk/cgi-bin/texfaq2html?label=whatTeX. Last Accessed: 05/14/2011.

[43] The Art of Computer Programming. http://en.wikipedia.org/w/index.php?title=The_Art_of_Computer_Programming&oldid=424138281. Last Accessed: 05/12/2011.

[44] Unit testing. http://en.wikipedia.org/w/index.php?title=Unit_testing&oldid=427435117. Last Accessed: 05/15/2011.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Marburg, June 14, 2011