

Diploma Thesis

Formal Specification of the x87 Floating-Point Instruction Set



Christoph Baumann

baumann@wjpservers.cs.uni-sb.de

Saarbrücken, 13. Februar 2008

Supervisor: Prof. Dr. Wolfgang J. Paul

Advisor: M. Sc. Ulan Degenbaev

Reviewers: Prof. Dr. Wolfgang J. Paul
Prof. Dr. Michael Backes

Ich versichere hiermit an Eides Statt, dass ich die von mir eingereichte Diplomarbeit selbständig verfasst und ausschließlich die angegebenen Quellen und Hilfsmittel verwendet habe. Ich habe diese Arbeit keinem anderen Prüfungsamt vorgelegt.

Saarbrücken, den 13. Februar 2008

“Цыплят по осени считают.”
(Russian proverb)

Acknowledgments

I want to express my gratitude to Prof. Dr. Wolfgang J. Paul for his helpful assistance during my research. Also throughout my studies i constantly benefited from his expertise as the knowledge and experience i gained from his lectures enabled me to write this thesis in the first place. Moreover his collaborative work with Silvia M. Müller on the formalization of floating-point arithmetics was an important basis for this document.

Furthermore I am thankful to Ulan Degenbaev for his considerable contributions to this thesis. I always appreciated his valuable suggestions and corrections during my work and it was a pleasure working together with him.

Last but not least i would like to thank my family and friends for their love and support in the past months and years.

Contents

1	Introduction	1
1.1	Goals	1
1.2	Related Work	2
1.3	Scope of this Document	2
1.4	Model Overview	3
1.5	Outline	5
2	Notation	7
2.1	Types and Domains	7
2.1.1	Basic Domains	7
2.1.2	Tuples and Records	8
2.1.3	Data Types	9
2.1.4	Functions	9
2.2	Source Code	10
2.3	Number Representations	12
2.3.1	Bit Strings	12
2.3.2	Binary Numbers	14
2.3.3	Binary Fractions	14
2.3.4	Two's Complement Numbers	15
2.3.5	Hexadecimal Numbers	16
2.4	Abbreviations	16
2.5	Conventions	17

2.5.1	Empty and Undefined Results	17
2.5.2	Quotations	17
2.5.3	Signed Zero	18
3	Interface	19
3.1	x87 Model Extensions	19
3.2	Decoding x87 Instructions and Operands	21
3.3	Executing x87 Instructions	22
3.4	XMM state Save and Restore	24
3.5	Handling Exceptions	25
4	Configuration	27
4.1	x87 Stack Registers	28
4.2	x87 Tag Word	29
4.3	x87 Status Word	29
4.4	x87 Control Word	30
4.5	x87 Instruction Information	32
4.5.1	Instruction components	32
4.5.2	Image Formats	33
4.5.3	Flags and Data	34
4.5.4	Memory Pointers	34
4.6	Initial Values	35
5	Numbers	37
5.1	x87 Floating-Point Data Representations	37
5.2	Factorings and Rounding	42
5.3	Converting Factorings and Bit Strings	44
5.4	Binary-Coded Decimal Numbers	46
5.5	x87 Exceptions	46
6	Operations	49

6.1	x87 Instruction Groups	49
6.1.1	Grouping by Functions	50
6.1.2	Grouping by Operand and Result Scheme	53
6.1.3	Operand and Result Predicates	54
6.1.4	Sources and Destinations	55
6.1.5	Grouping by Stack Manipulation Behaviour	58
6.2	x87 Format Converting Functions	58
6.2.1	Converting Precision	59
6.2.2	Converting to Binary Integer Format	59
6.2.3	Converting to BCD Format	60
6.2.4	Maximal and Minimal Values	60
6.3	x87 Mathematical Functions	61
6.3.1	Addition	63
6.3.2	Subtraction	64
6.3.3	Multiplication	64
6.3.4	Division	65
6.3.5	Rounding	65
6.3.6	Extracting Significand and Exponent	65
6.3.7	Partial Remainder	66
6.3.8	Square Root	67
6.3.9	Sine and Cosine	67
6.3.10	Tangent and Arc Tangent	68
6.3.11	Exponentiation	69
6.3.12	Logarithm	69
6.4	x87 Comparison Functions	70
7	Execution	73
7.1	x87 Pre-Computation Exception Checks	73
7.1.1	Detecting Pre-Computation Exceptions	74

7.1.2	Transition Function	76
7.2	x87 Result Computation and Storage	77
7.2.1	Transition Function	78
7.2.2	Stack Operations	78
	Pushing Values to the Stack	78
	Saving Values in the Stack	79
7.2.3	Computing the Result	80
	Calculation Results	81
	Sign Operation Results	82
	Comparison Results	82
	Constant Results	83
	Conversion Results	83
	Load Results	83
	Control Results	84
	Special Results	86
7.2.4	Comparison, Stack Management and Control Instructions	87
	Image Components	88
	Effects on the Configuration	89
7.3	x87 Post-Computation Exceptions Checks	92
7.3.1	Detecting Post-Computation Exceptions	92
7.3.2	Impact on the FPU Environment	92
7.3.3	Adjusting the Result	94
7.3.4	Transition Function	95
7.4	Returning the x87 Result	95
7.4.1	Storing a Result to Memory	95
7.4.2	Popping the Stack	96
7.4.3	Transition Function	96

8 Conclusion

99

A Opcode Table	101
B Instruction Predicates	105
C FPU Image Formats	109
D Next State FPU Configuration	113

List of Figures

1.1	old CPU Model vs. new CPU Model	4
3.1	New CPU model with FPU	20
4.1	FPU configuration components	28
4.2	x87 Physical and Stack Registers	28
5.1	The floating-point formats of the FPU	38
5.2	Transferring real numbers into floating-point representation	45
5.3	Structure of 80-bit BCD numbers	46
6.1	Structured x87 instruction set	52
7.1	Effect of a push operation on the x87 stack	79
7.2	Effect of a pop operation on the x87 stack	96
C.1	x87 Environment image format for 32-bit protected mode	109
C.2	x87 Environment image format for 32-bit real/virtual mode	109
C.3	x87 Environment image format for 16-bit protected mode	110
C.4	x87 Environment image format for 16-bit real/virtual mode	110
C.5	XMM/x87/Media image format for non-64-bit mode	111
C.6	XMM/x87/Media image format for 64-bit mode	112

Chapter 1

Introduction

In this thesis we will define the semantics of x87 floating-point instructions. These instructions are part of the Intel IA32 and AMD64 Instruction Set Architectures for modern processors. They implement floating-point functionalities in hardware, benefitting numeric and multimedia-related calculations that make extensive use of floating-point arithmetics. Floating-point numbers represent real numbers with a limited precision and enable more complex calculations than plain integers. We will describe how the particular instructions are executed within the framework of an IA32/AMD64-compliant processor. For that purpose we will produce a formal model of the Floating-Point Unit (FPU) and specify the effects of x87 instructions on its various components. A CPU model was already established by Ulan Degenbaev for general purpose instructions (cf. [Deg07]) and we will extend it to support the x87 instruction subset.

The formalization of the complete x86-64 ISA is part of the VerisoftXT project, which aims among other things at verifying a modern hypervisor. A hypervisor is a system software that runs several operating systems in parallel as so-called guest partitions. For each guest an entire CPU with system and user mode is simulated. To prove the correctness of a hypervisor a formal model of that CPU is needed and the x87 part of it is contributed by this thesis.

1.1 Goals

Our main goal in this document is to translate the informal specification of the instruction set into a formal model that is complete and well-defined. It shall comprise the behaviour of all x87 instructions in an efficient and coherent way. A major difficulty will be the sheer abundance and depth in form and content of the x87 instruction subset. This set of instructions sums up a vast diversity of different functions and routines. Furthermore for many operations several variant instructions exist and for every instruction a multitude of special and exceptional cases must be distinguished. In addition we will have to deal with different number formats and precisions, rounding modes, image formats etc. This confusingly high level of details will possibly result in a quite complex model of the FPU. Nevertheless it is our objective to create a model that is as abstract and generalized as possible but still incorporates all the special cases and details of the

specification. In the end the reader shall be able to effortlessly comprehend the impact of floating-point instructions on the CPU.

1.2 Related Work

Of course this thesis relies on the specification of the x87 instruction set. It is taken from the “Intel 64 and IA-32 Architectures Software Developer’s Manual”[IA07] and the “AMD64 Architecture Programmer’s Manual”[AMD07]. In total about 400 pages from these specifications are compressed in this document. Furthermore this thesis is based on the general purpose instruction CPU model created by Ulan Degenbaev in his Master thesis “Formal Specification of Parts of the x86-64 Instruction Set Architecture”[Deg07]. Although we give an introduction to his model later on we expect the reader to be familiar with his work as we use and extend some of his definitions to build our model. Floating-point numbers are defined in the IEEE 754 Standard for Binary Floating-Point Arithmetic [IE85]. To formalize these numbers we revert to the book “Computer Architecture, Complexity and Correctness”[MP00] written by Silvia M. Müller and Wolfgang J. Paul. There we find a complete formalization of the IEEE 754 standard and useful definitions to define rounding and format conversion. However we will have to customize them to fit into our model. Also at other subjects we are influenced by Prof. Wolfgang J. Paul’s books and lectures.

Besides the formalization of the IEEE floating-point standard in [MP00] there have been many projects about specification and verification of IEEE 754 compliant hardware components. However to our knowledge up to now there have been no attempts to specify the semantics of the entire x87 instruction set for a modern processor. Thus we literally have to start from scratch for our model.

1.3 Scope of this Document

As stated before this document deals with the x87 floating-point instruction subset. A table of all x87 instructions apart from *FWAIT*, *FXSAVE* and *FXRSTOR* can be found in Appendix B. All in all we consider 83 distinct instructions.¹ To state the effects of these instructions we present a formal model for the FPU. It is embedded into the CPU model of Ulan Degebaev described in the next section. Therefore we have to develop an interface between both models. Moreover the instructions *FXSAVE* and *FXRSTOR* access components of the XMM instruction environment. Although this environment is not formally defined yet, we have to specify the semantics of the two instructions mentioned above. To this end we will have to presume a reasonable definition of the respective XMM components. Apart from these subjects the remaining parts of the CPU are not discussed in this thesis. We particularly consider the following issues to be beyond the scope of this document.

- instruction fetch of x87 instructions (part of [Deg07])

¹Note that there may be aliases for some x87 instructions. For instance *WAIT* is an alias for *FWAIT* and *FSAVE* is a compiler macro for *FWAIT* followed by *FNSAVE*. There are aliases like this for every x87 control instruction beginning with *FN*.

- loading x87 memory operands from memory (part of [Deg07])
- storing x87 results to memory (part of [Deg07])
- 128-bit/XMM and 64-bit/MMX media instructions besides *FXSAVE* and *FXRSTOR* (not defined yet)
- FERR# and IGNNE# signals for handling x87 exceptions by external logic (external processor signals not defined yet)
- handling of non-floating-point exceptions raised by x87 instructions outside of the FPU (not defined yet)

The last point is very important. In this thesis we assume that exceptions occurring for x87 instructions after the FPU has finished computation, do not cause a repetition of the instruction execution as the FPU is not able to reverse an instruction execution on its own. The CPU model has to provide means to restore the old FPU configuration in case of exceptions causing a repeated instruction execution.

Having stated what is not part of this thesis we also want to list the capabilities of our model. It includes the following features.

- x87 instruction and operand decoding
- an interface between FPU and CPU model ensuring full compatibility to [Deg07]
- execution of the XMM-related *FXSAVE* and *FXRSTOR* instructions
- formalization of the FPU components
- a complete formalism for the x87 floating-point and integer number formats
- a general definition of all mathematical operations defined in the x87 instruction set
- calculation, conversion and storage of x87 instruction results
- execution of x87 stack management and control instructions
- detection and treatment of x87 exceptions
- overview of the FPU next state configuration

1.4 Model Overview

In his Master's Thesis [Deg07] Ulan Degenbaev established a model for general purpose and system instructions. This model is based on several concurrently running units which communicate with each other using requests. A CPU contains the following units in this model.

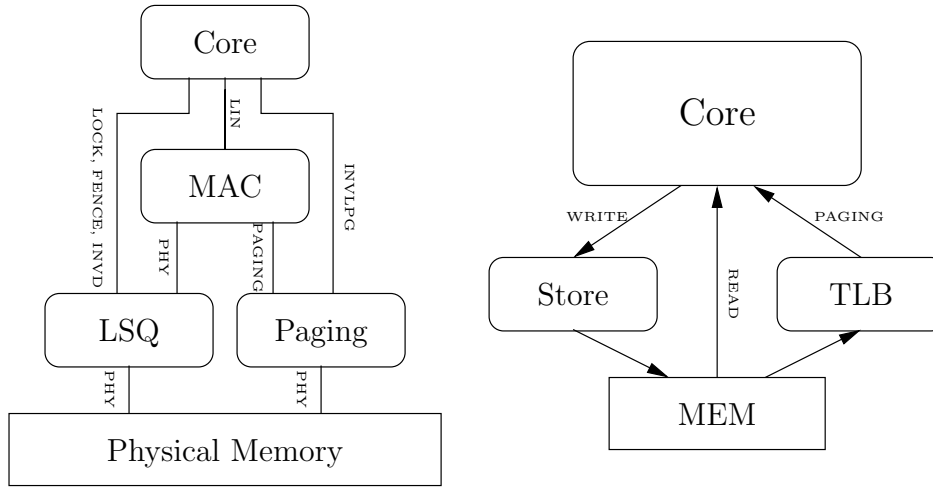


Figure 1.1: old CPU Model vs. new CPU Model

- Core - the central unit containing all registers, being responsible for instruction fetch, decoding and execution.
- MAC - the MAC unit is called by the Core to perform linear memory accesses using the remaining two units.
- TLB - the translation lookaside buffer unit manages paging, address translation and caching.
- LSQ - the load/store queue unit is used to collect and reorder memory accesses. Also out-of-order memory accesses are simulated by this unit.

Following the idea of a hypervisor the model assumes several CPUs sharing one physical memory. For clarity we assume from now on that there is only a single CPU to be considered. The units are run by turns. They may send requests to other units and answer the requests they received.

However, recently the model was altered. The rather complicated functional and unit-based approach was changed to a more comfortable centralised one where all the necessary subroutines are executed atomically.

We still have the Core and the TLB paging unit. The LSQ was replaced by a pure store queue because the core may now read data directly from memory once it received the address translation from the TLB. Values to be stored in memory are handed over to the store unit which non-deterministically performs the write accesses to memory. Apart from that only the Core unit is running cycles. Figure 1.1 illustrates both CPU models.

The execution of instructions is still similar to the old model. Instructions are processed sequentially. First Core checks for interrupts from the preceding instruction. When exceptions were raised the core jumps to the interrupt service routine. Otherwise a new instruction is fetched. The function *exec(cpu)* is called to decode and execute the particular instruction. To access memory the Core unit can directly utilize the functionalities of the TLB and Store units.

Within this framework the FPU environment is a component of the Core unit. We will extend the $exec(cpu)$ function by a case for x87 instructions and define an interface function $execFPU(cpu)$ that connects both models with each other and computes the next FPU state.

1.5 Outline

We choose a deductive, top-down approach to define the semantics of floating-point instructions.

Chapter 2 introduces the basic **Notation** used in this document. We will define the main types and sets of numbers that represent the domains for the components and functions we will use to model the floating-point unit. As we also use source code to establish the interface between CPU and FPU, we will explain the corresponding functional language in the subsequent section. Because number formats play a central role in this thesis, we will also introduce the fundamental number representations in this chapter. In the end we establish several abbreviations and conventions.

Chapter 3 starts the definition of our FPU model. We extend the given CPU framework with the floating-point unit and develop the **Interface** between both models. We will also manipulate the respective functions from [Deg07] to enable decoding of x87 instructions. Finally the exception signaling as well as the implementation of XMM state saving and restoring are parts of this chapter.

Chapter 4 defines the **Configuration** of the floating-point unit comprising all of its components. This configuration describes the current state of the x87 environment. The main goal of this thesis will be to define the next state configuration f' resulting from executing an x87 instruction on an arbitrary configuration f . To this end we provide functionality to comfortably access or manipulate the FPU components and current instruction information.

Chapter 5 introduces a formalism for Floating-Point **Numbers**. Many definitions are therefor taken from [MP00], however we must modify the respective functions to fit our needs. In this chapter we will also discuss the issues of rounding and conversion between real numbers and floating-point bit strings. Additionally we introduce BCD numbers and x87 exceptions.

Chapter 6 deals with the various **Operations** implemented by the x87 instruction set. In the first part we structurize this instruction set by dividing the instructions into different functional groups, establishing predicates for certain instruction types and investigating on several instruction characteristics that are appropriate for distinguishing instruction variants. In these sections we also provide definitions for the x87 instruction operands. The second part of this chapter concerns the semantics of the various operations. We define the exact results for conversions, calculations and comparisons in a generalized way.

Chapter 7 eventually implements the FPU state transition function σ_{fpu} which simulates the **Execution** of x87 instructions. We divide this simulation into four steps, namely the detection of pre-computation exceptions, the computation of the result followed by the post-computation exception detection and finally the returning of the result back to the CPU. By the end of this chapter our FPU model will be complete.

We conclude in chapter 8.

Chapter 2

Notation

In this chapter we will cover notation issues and establish necessary conventions for the following thesis.

2.1 Types and Domains

In order to give the syntax and semantics of functions we need to formally define the types and domains of arguments and results.

2.1.1 Basic Domains

The basic domains contain boolean, natural and integer numbers.

$$\begin{aligned}\mathbb{B} &= \{0, 1\} \\ \mathbb{N} &= \{0, 1, 2, 3, \dots\} \\ \mathbb{Z} &= \{\dots, -2, -1, 0, 1, 2, \dots\}\end{aligned}$$

Observe that 0 and 1 occur in all three sets. One can determine the original set of these numbers from the context of their occurrences. In addition to the domains above the set of real numbers $\mathbb{R}$ may be referenced by functions we define. For some more comfort we denote certain subsets of the natural numbers by the following shorthand notations.

$$\begin{aligned}\mathbb{N}^+ &= \mathbb{N} \setminus \{0\} \\ \mathbb{N}_n &= \{0, 1, 2, \dots, n-1\}\end{aligned}$$

$\mathbb{N}^+$ denotes the strictly positive natural numbers, while $\mathbb{N}_n \subset \mathbb{N}$ comprises the the first $n \in \mathbb{N}^+$ natural numbers. For the well-known boolean operations we use the following notation.

- $\wedge$ - logical AND
- $\vee$ - logical OR

- $\oplus$ - exclusive OR

The negation of a boolean variables x is given by $\bar{x}$. Accordingly $\bar{\wedge}$ represents the boolean NAND operation. In addition we make use of the following mathematical operations on real and integer numbers x, y .

- $x + y$ - Addition
- $x - y$ - Subtraction
- $x \cdot y$ - Subtraction
- x/y - Division
- $x \bmod y$ - modulo computation (only for integers)
- x^y - Exponentiation
- $\log_x y$ - Logarithm
- $\sqrt{x}$ - Square Root
- $\sin x, \cos x$ - Sine and Cosine
- $\tan x, \arctan x$ - Tangent and Arc Tangent
- $\text{sgn}(x)$ - Signum (returns the sign of a number)
- $|x|$ - Absolute value

These operations are defined as known from mathematics, however we will state the results for special arguments when necessary.

2.1.2 Tuples and Records

In this document we produce a model for the floating-point unit of an IA-32 compliant processor. Similar to the main part of the processor formalization written by Ulan Degenbaev our model shall be translated to a programming language later on. That is why we would like to base our functions on data types rather than on mathematical domains. Therefore we introduce tuples and records as well as data type definitions in the following.

Tuples and records are our main means to define the type of our model components and functions. An n -tuple with $n \in \mathbb{N}^+$ consists of n variables $x_1, \dots, x_n$ while x_i has type T_i . Then the tuple is represented as $(x_1, x_2, \dots, x_n)$ with type $T_1 \times T_2 \times \dots \times T_n$. For our type declarations we use a simplified notation for type tuples.

$$(x_1, x_2, \dots, x_n) \in (T_1, T_2, \dots, T_n)$$

In addition we want to comprise all our model components in a data structure. To this end we adopt records. These are data types similar to tuples with the difference that here the components of the record have names. A record type T with is obtained through the following type definition.

$$T = \{x_1 : T_1, x_2 : T_2, \dots, x_n : T_n\}$$

Here x_i is the identifier of the i -th record component with type T_i . A component x_i of record r may be accessed via:

$$r.x_i$$

Note that a record may contain further complex data types.

2.1.3 Data Types

Accessorily to the basic types we may want to construct auxiliary data types. The basic types serve as the atomic elements for type construction. We may define a new type T from available types T_i by

$$T = T_1 \mid T_2 \mid \dots \mid T_i \mid \dots \mid T_n .$$

The “ $\mid$ ” works as a type union operator here. Apart from this mere joining of basic types we may use type constructor functions that take an arbitrary number of type arguments to define more complex types. A type T resulting from a type construction is identified by the name of the constructor function and the argument types.

$$T = \text{CONSTR } T_1 \mid T_2 \mid \dots \mid T_i \mid \dots \mid T_n$$

An instance of this type with arguments x_i is then represented by $\text{CONSTR } x_1, \dots, x_n$. We mostly use type joining and constructor functions without arguments to define enumeration types. Additionally we may extend a type T with the empty value ϵ via $T \mid \epsilon$. For more information about ϵ see subsection 2.5.1.

2.1.4 Functions

To define the semantics of the x87 instruction subset mathematically, we have to use functions of course. A function maps arguments of one type S to a result of type T in a well-defined manner. However functions may be interpreted as types, too. Therefore the type of function f is given by

$$f : S \rightarrow T ,$$

while the body of the function is represented in the usual way for an $s \in S$.

$$f(s) = \text{expr}$$

Here expr is a logical or arithmetic expression defining the result of f . Only arguments contained in s may be used to determine the result. However sometimes we may omit common arguments in

the notation and reference them implicitly for the sake of simplicity. From a programmer's point of view this may be interpreted as dependencies on the values of global variables. Nevertheless we will always make remarks, when dealing with implicate arguments.

Observe that we keep using the “ $\in$ ” operator for stating type affiliation. Only for record and function declarations the “ $:$ ” is used.

2.2 Source Code

In [Deg07] a functional programming language is used to define the general purpose CPU model. In this thesis we want to restrict ourselves to mathematical definitions as far as possible, however at least concerning the interface between CPU and FPU we will not be able to avoid providing source code in the aforementioned language. A description of its syntax and semantics can be found in [Deg07], sections 2.5 and 4.4.5. Nevertheless we want to give a short introduction on several specialities of the language. As a functional language, all procedures are defined like mathematical functions. At first a type declaration is given. It is followed by the body definition for the respective function. The body consists mainly of ordinary expressions, case distinctions and let statements.

- Ordinary expressions are valid terms of type T for a function $f : S \rightarrow T$.
- A simple case distinction is the *if-then-else* expression.

```
1 if cond then expr1
2 else expr2
```

Here *cond* is a logical expression of type $\mathbb{B}$. The expression is interpreted in the well-known way. For *cond* = 1 it results in *expr₁*, else *expr₂* is returned. *if-then-else* expressions may be nested.

```
1 if cond1 then expr1
2 else
3     if cond2 then expr2
4     else
5         if cond3 then expr3
6         else
7             ⋮
8             if condn then exprn
9             else exprn+1
```

This may be written in a simplified way.

```
1 |> cond1  → expr1
2 |> cond2  → expr2
3 |> cond3  → expr3
4 ⋮
5 |> condn  → exprn
6 |> otherwise → exprn+1
```

- Another case distinction is the *case* expression for distinguishing different values of an expression x .

```

1 case x of
2   x1 → expr1
3   x2 → expr2
4   ⋮
5   xn → exprn
6 default → exprn+1

```

This is just the same as:

```

1 |> x = x1 → expr1
2 |> x = x2 → expr2
3 |> x = x3 → expr3
4   ⋮
5 |> x = xn → exprn
6 |> otherwise → exprn+1

```

- The *let* expression is used to define temporary variables and abbreviations that can be used in the expression following the *in* keyword.

```

1 let x1 = a
2     x2 = b
3     ⋮
4     xn = abc in
5 expr1

```

All appearances of x_i in $expr_1$ are replaced by the assigned values according to the *let* statement. Note that pattern matching may be used to define the abbreviation.

In addition to these expressions the syntax is extended by a special type extension and a *let?* statement. In the old model the execution was interrupted whenever an exception occurred or a request was raised in the course of the next state computation. In these cases the *Core* component had to be returned to the CPU updating only the respective exception or request components. To this end the source code would have been spoiled with a lot of *if-then-else* case distinctions of the form:

```

1 A: Core → Core
2 A (c) =
3   let c' = B(c) in
4   |> c'.req ≠ ε → c'
5   |> c'.excp ≠ ε → c'
6   |> otherwise → // do some computation with c'

```

In this example $B : Core \rightarrow Core$ returns a modified version of the core and we have to check whether exceptions or requests occurred. To find a shorter notation for this Ulan Degenbaev defined a special type $?T$ for any given typ T as follows.

$$?T = \text{NORMAL } T | \text{PASS } Core$$

Thus a value of $?T$ could either be NORMAL t or PASS c with $t \in T$ and $c \in Core$. So now every function of type $X \rightarrow T$ that might have to pass the core at some point may be defined as $X \rightarrow ?T$ to catch these exceptional cases. Then the example from above may be written as:

```

1 A: Core  $\rightarrow$  ?Core
2 A (c) =
3   let r = B(c) in
4   case r of
5     PASS c'  $\rightarrow$  PASS c'
6     NORMAL c'  $\rightarrow$  // do some computation with c'

```

This combination of *let* and *case* may be abbreviated by the following *let?* statement.

```

1 A: Core  $\rightarrow$  ?Core
2 A (c) =
3   let? c' = B(c) in
4   // do some computation with c'

```

Accordingly

```

1   let? y = f(x) in
2   // do some computation with y

```

stands for

```

1   let r = f(x) in
2   case r of
3     PASS c  $\rightarrow$  PASS c
4     NORMAL y  $\rightarrow$  // do some computation with y

```

Please refer to [Deg07] for more explanations and examples on this subject. Also note that in the new model the whole *Cpu* type is passed to all functions so that all memory accesses can be executed atomically. This means for our document that type $?T$ has a new definition.

$$?T = \text{NORMAL } T | \text{PASS } Cpu$$

2.3 Number Representations

This section deals with basic representations of integer numbers and binary fractions. Additional formats for floating-point numbers and binary coded decimal numbers will be discussed in chapter 5.

2.3.1 Bit Strings

In the processor all data is saved in bit format. Therefore also all numbers are stored as strings of bits. Before examining the different representations we want to investigate bit strings first.

Fixed length bit strings of length n may be modelled as boolean n -tuples.

$$\underbrace{\mathbb{B} \times \mathbb{B} \times \cdots \times \mathbb{B}}_{n \text{ times}} = \mathbb{B}^n$$

Sometimes functions will result in bit strings of different lengths. For instance the binary representation of an FPU operation's result varies in its size with respect to the format of the particular result. Instead of defining bit string type unions for all the special cases, we denote the type for variable-length bit strings by $\mathbb{B}^*$. This is a common notation from set theory which might be interpreted as

$$\mathbb{B}^* \subseteq \bigcup_{n=1}^{\infty} \mathbb{B}^n$$

Now let $x \in \mathbb{B}^m$ and $y \in \mathbb{B}^n$ be bit strings. We want to introduce methods to access and manipulate them.

- $x[i]$ is the i -th element of the string for some $i \in \mathbb{N}_m$. We use Little Endian notation, i.e. the bits are arranged from left to right in descending order and the leftmost bit contains $x[m-1]$. As an abbreviation $x_i = x[i]$ can be used.
- $x \circ y$ is the concatenation of two bit strings. With $i \in \mathbb{N}_{n+m}$ the following statement holds for the concatenated string.

$$(x \circ y)[i] = \begin{cases} x[i-n] & n \leq i < n+m \\ y[i] & \text{else} \end{cases}$$

That means y is appended to the rightmost bit of x . Apparently x can be represented as $x[m-1] \circ x[m-2] \circ \cdots \circ x[1] \circ x[0]$. For constructing bit strings from boolean constants we allow leaving out the concatenation symbols for clarity. For example $1 \circ 0 \circ 1^3 \circ 0^3$ can be written as 101^30^3 .

- x^k concatenates $k \in \mathbb{N}$ copies of x yielding a new string with type $\mathbb{B}^{m \cdot k}$.

$$x^k = \underbrace{x \circ x \circ \cdots \circ x}_{k \text{ times}}$$

When a string is copied using zero or a negative value for k , it is omitted and replaced with an empty string respectively.

- $x[j : i]$ represents the substring $x[j] \circ x[j-1] \circ \cdots \circ x[i+1] \circ x[i]$ for some $i, j \in \mathbb{N}$ and $0 \leq i \leq j < m$. Depending on the context substring selections with $i > j$ may be omitted or they may produce a substring with reversed bit order. $x[j : i]$ with $j = i$ naturally returns $x[i]$.

These definitions allow us to comfortably handle bit strings.

2.3.2 Binary Numbers

Bit strings may be interpreted in several ways to determine the number they represent. The most common format is that of unsigned binary numbers. The value $\langle x \rangle$ of a number's binary representation $x \in \mathbb{B}^n$ can be obtained by applying the following definition from [MP00].

$$\langle x[n-1:0] \rangle = \sum_{i=0}^{n-1} x_i \cdot 2^i$$

This just complies with the well-known concept of binary numbers. An n -bit binary number can encode values from range $\{0, \dots, 2^n - 1\}$.

To return the binary representation of length n for a given natural number a we introduce the conversion function bin_n .

$$\begin{aligned} bin_n : \{0, \dots, 2^n - 1\} &\rightarrow \mathbb{B}^n \\ bin_n(a) = x[n-1:0] &\Leftrightarrow \langle x \rangle = a \end{aligned}$$

Sometimes we need to increase the length of a bit string without changing the value of the number it represents. This is called *zero-extension*. Basically it is just appending zeroes in front of the bit string. For a bit string $x \in \mathbb{B}^m$ the zero-extended bit string with length n is denoted by $zxt_n(x)$ as long as $n \geq m$ holds.

$$\begin{aligned} zxt_n : \mathbb{B}^m &\rightarrow \mathbb{B}^n \\ zxt_n(x) &= 0^{m-n} \circ x[m-1:0] \end{aligned}$$

One can easily prove that $\langle zxt_n(x) \rangle = \langle x \rangle$. At last we want to care about addition and subtraction of binary numbers. We would like to comfortably obtain the bit strings representing the result of the addition or subtraction of two integers in binary format. Therefore we establish the modulo- n -bit binary addition and subtraction $+_n$ and $-_n$. They implement the following characteristics for some binary numbers $a, b, c \in \mathbb{B}^n$.

$$\begin{aligned} a[n-1:0] +_n b[n-1:0] &= c[n-1:0] \Leftrightarrow \langle a \rangle + \langle b \rangle = \langle c \rangle \bmod 2^n \\ a[n-1:0] -_n b[n-1:0] &= c[n-1:0] \Leftrightarrow \langle a \rangle - \langle b \rangle = \langle c \rangle \bmod 2^n \end{aligned}$$

Observe that overflowing or negative results will be wrapped around to a representable number due to the modulo computation. One can show that calculating modulo 2^n with binary numbers can be emulated by computing the exact result with a higher number of bits and then truncating the upmost bits to receive a bit string of length n .

2.3.3 Binary Fractions

Besides binary integers we will be confronted with binary fractions. Luckily they follow the same rules as decimal fractions and have a limited precision, so we can simply handle them as binary natural numbers that are scaled by some power of two with a negative exponent. From [MP00] we learn that a binary fraction consists of two bit strings $a \in \mathbb{B}^n$ and $b \in \mathbb{B}^p$ separated by a point, so

$$a[n-1:0].b[1:p]$$

represents the corresponding binary fraction. Note that the fractional part b is kept in reversed bit order. This reflects the magnitude of the negative exponent the respective bits are scaled with. According to [MP00] the value of a binary fraction $a.b$ can be obtained by the following formula.

$$\langle a[n-1:0].b[1:p] \rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i + \sum_{j=1}^p b_j \cdot 2^{-j}$$

Observe that the fraction's value can be split into the sum $\langle a.b \rangle = \langle a \rangle + \langle b \rangle \cdot 2^{-p}$ which uses only the common definition of the binary representation for natural numbers. Similarly for some $x \in [0, 1)$ with the property $\lfloor x \cdot 2^p \rfloor = x \cdot 2^p$ the binary fractional representation $0.b$ with $b \in \mathbb{B}^p$ can be determined via

$$b[1:p] = \text{bin}_p(x \cdot 2^p).$$

It can be proven that $\langle 0.b \rangle = x$ holds.

2.3.4 Two's Complement Numbers

Up to now we only considered positive numbers. The common format for negative integers is the *two's complement representation*. [MP00] gives the definition of a bit vector $a[n-1:0]$'s two's complement value.

$$[a] = -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle$$

Thus the range of representable numbers is defined as

$$[a] \in \{-2^{n-1}, \dots, 2^{n-1} - 1\}$$

For an integer x from this range we can produce the n -bit two's complement bit string via $\text{twoc}_n(x)$.

$$\begin{aligned} \text{twoc}_n : \{-2^{n-1}, \dots, 2^{n-1} - 1\} &\rightarrow \mathbb{B}^n \\ \text{twoc}_n(x) = a[n-1:0] &\Leftrightarrow [a] = x \end{aligned}$$

An important observation is that the upmost bit a_{n-1} of the two's complement number determines the sign of its value. That's why this bit is also called *sign bit*. Comparable with the zero-extension for binary numbers, two's complement numbers can be enlarged while conserving their values via *sign-extension*. The sign-extended n -bit version of $a \in \mathbb{B}^m$ is given by $\text{sxt}_n(a)$ for $n \geq m$.

$$\begin{aligned} \text{sxt}_n : \mathbb{B}^m &\rightarrow \mathbb{B}^n \\ \text{sxt}_n(a) &= (a_{m-1})^{m-n} \circ a[m-1:0] \end{aligned}$$

Sign-extension assures that $[\text{sxt}_n(a)] = [a]$ holds true.

At first sight two's complement numbers might appear to be a somewhat unintuitive approach to implement integers. However their structure bears some great benefits compared to other solutions, as for instance the equation $[a] = \langle a \rangle \bmod 2^n$ holds for any bit string $a \in \mathbb{B}^n$. See [MP00] for more information on this number format.

2.3.5 Hexadecimal Numbers

Besides binary format we also use hexadecimal numbers in this document. They are mainly used to state opcodes or bit masks because one hexadecimal digit is able to encode four bits. Thus longer bit strings become more clear and memorable when given in hex format. We denote the set of hexadecimal digits by $\mathbb{Q}$.

$$\mathbb{Q} = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$$

Then any bit string $x \in \mathbb{B}^{4n}$ can be encoded by a hexadecimal number $h \in \mathbb{Q}^n$. The value of a hex number h can be easily computed as follows.

$$\langle h[n-1:0] \rangle = \sum_{i=0}^{n-1} \langle h_i \rangle \cdot 16^i$$

Here the digits $0, \dots, F$ are evaluated by the overloaded $\langle \cdot \rangle$ -brackets as the natural numbers from 0 to 15. Though, we are rather interested in the corresponding bit vector than in the actual value of a hexadecimal number. Hence we establish the “0x” prefix for these numbers. For $h \in \mathbb{Q}^n$ it yields the bit vector $x \in \mathbb{B}^{4n}$ such that

$$\begin{aligned} 0xh &= x \\ \langle h[n-1:0] \rangle &= \langle x[4n-1:0] \rangle \end{aligned}$$

holds. This way we are able to comfortably include hex-codes in our formulae.

2.4 Abbreviations

Throughout this document we utilize several abbreviatory notations to promote clarity and avoid tedious repetitions of similar definitions. For instance when defining various functions with the same type scheme and related names, we allow to abbreviate the several similar declarations by a single one containing a placeholder x that can be replaced by the appropriate letters to form the declaration for a particular function.

Another measure of combining definitions are the $\pm$ and $\mp$ signs. They are used to distinguish the two cases of positive and negative arguments - or addition and subtraction respectively - in a single definition. An expression $expr(\pm, \mp)$ containing the $\pm$ and $\mp$ signs stands for two definitions where the respective signs are symbolically replaced by $+$ or $-$ according to the following rules.

$$expr(\pm, \mp) \equiv expr(+, -) \wedge expr(-, +)$$

For example the equations $\sin(\frac{3}{2}\pi) = -1$ and $\sin(-\frac{3}{2}\pi) = 1$ maybe combined to the statement

$$\sin(\pm\frac{3}{2}\pi) = \mp 1$$

We can extract the two equations back from this by applying the rule from above. $\pm$ and $\mp$ are replaced by $+$ and $-$ for the first equation and vice versa for the second one.

Another abbreviatory notation can be found in table B.1 of Appendix B. There we list the sets of

opcodes in hexadecimal format which belong to a certain instruction. Many of these opcodes are succeeding each other, that means they only differ in the last hexadecimal digits. For clarity and to reduce the size of the definitions we allow to define hexadecimal ranges in the following way. Let $i, j \in \mathbb{Q}^m$ be some hex digit strings with $\langle j \rangle > \langle i \rangle$ and $h \in \mathbb{Q}^n$ a hexadecimal number with $n \geq m$, then $h[n-1:m][i:j]$ denotes the following range of hex numbers.

$$h[n-1:m][i:j] = h[n-1:m] \circ i, h[n-1:m] \circ i+1, \dots, h[n-1:m] \circ j$$

For example the opcodes for the instruction *FXCH*, namely 0xD9C8, 0xD9C9, 0xD9CA, ..., 0xD9CF may be comprised by the hex-code range 0xD9C[8:F]. Note that we overload the selection brackets here with contrary semantics. However as we use the range abbreviation brackets only for constants and the substring selection brackets only for variables there is no conflict between both notations.

2.5 Conventions

There are some general subjects left we want to discuss.

2.5.1 Empty and Undefined Results

The x87 part of the IA-32 specification as well as the whole instruction set itself comprises a multitude of different functionalities. For each instruction we have to consider various special cases. In addition we want to keep our functions as general and comprehensive as possible. This however will make it hard for us to give reasonable results for these functions considering all possible FPU configurations. That is why we introduce the empty result ϵ which is returned in all the cases where a well-defined result is neither available nor required from a certain function. It just assures that we have exhaustive case distinctions and definitions. In addition the functions of our model may accept empty arguments when necessary, that means they are also defined for the cases that their inputs may become ϵ .

On the contrary the Intel IA-32 and AMD64 Architecture specifications often leave particular issues undefined. Whenever this is the case the components under consideration will be updated with the value “undefined”. Note that ϵ will never be assigned to an FPU component. It is in a way the “undefined” value of our internal model. We apply this distinction to point out where the specification itself is unclear on certain subjects. Moreover an undefined bit is represented by an “X”.

2.5.2 Quotations

This document is based on the general purpose CPU model in [Deg07]. Therefore we will at least for the interface need to recall definitions and functions from Ulan Degenbaev’s thesis. Additionally the IEEE floating-point number format was already formalized in [MP00]. Thus we will also cite definitions from this book. Whenever we refer to ideas and formulae from external sources we will

state their origin in brackets close to the quoted material. For the interface we will have to quote and extend sourcecode from [Deg07]. All cited lines will be held in a grey shade to mark which portions of code were added by us and which were taken from that thesis. We also may leave out certain parts of definitions and code listings by “...” when they are irrelevant for our model.

2.5.3 Signed Zero

A famous speciality of floating-point numbers is that they have two representations for zero, namely positive and negative zero. Reading this one will surely wonder about the purpose of signed zeroes. One main reason is the rounding of very small numbers. When a floating-point number is rounded to zero one may determine whether the exact result was a positive or negative number from the sign of zero. Also when finite values are divided by infinities the sign of the resulting zero reflects the signs of the operands. Of course in mathematics zero has no explicit sign, however in this document we assume that $+0$ and -0 may be distinguished not only for floating-point but also for real and integer numbers. Still these representations have the same value and are considered equal by comparisons. We want to set up conventions to handle this ambiguous subject.

Within this model the sign of zero is stated explicitly for all definitions. Whenever a variable x 's value shall be just zero regardless of the sign, we apply an absolute value notation. Thus the following expressions may be found.

$$x = +0 \quad x = -0 \quad |x| = 0 \quad |x| \neq 0$$

In the third case the value of the variable may either be $+0$ or -0 , its sign is undefined. When necessary we interpret these zeroes as positive by default. In the fourth case x contains neither signed nor unsigned zeroes at all. We never use the $\pm$ symbol to state that a zero's sign is undefined.

For the various mathematical operations we will identify special cases and state the results for computations involving signed zeroes. As mentioned before the x87 comparison instructions ignore the sign of zero. On the contrary the sign of zeroes is always taken into account when converting numbers to IEEE floating-point or BCD format. For these formats the sign may be stored explicitly in a sign bit, so we must not omit it in the course of our result computation.

Chapter 3

Interface

Defining our model deductively, we begin our investigation in the CPU model. As explained in the introduction we want to extend the new centralised model invented by Ulan Degenbaev to support floating-point instructions. The old model was defined using a functional language with full documentation, while the new model just exist as C source code. However it would be rather useless to base this thesis on an outdated foundation. Thus we stick to the new model but we keep the old syntax as it is mathematically precise, type consistent and it allows convenient integration of our formal FPU specification. Doing so also enables us to refer to several useful definitions from [Deg07]. Anyway we must extend the given model to provide floating-point functionality.

3.1 x87 Model Extensions

To describe the execution of floating-point instructions by the x87 coprocessor we introduce an additional component to the CPU model: the floating-point configuration. The *Cpu* type of the new model is defined in the following way.

$$Cpu = \{core : Core, store : Store, tlb : Tlb, mem : Mem\} \quad [Deg07]$$

One could suggest that the FPU should be a separate component of the CPU as it also used to be a separate hardware component in the real world. On the other hand floating-point instructions are integrated into the core's fetching and decoding routine and should be treated in the same way as general purpose and system instructions within our sequential model. Furthermore there are links between x87 and the 64-bit media and XMM instruction subsets and the latter ones are incorporated within the core. At last the FPU environment consists merely of registers, which customarily reside inside the core component. On account of these facts we include the floating-point configuration as a component of the core and establish functions to manipulate this environment according to the floating-point operations under consideration.

$$Core = \{\dots, fpu : FPU\} \quad [Deg07]$$

The FPU environment contains all necessary data to perform one execution step of the FPU. Figure 3.1 depicts the new CPU containing the FPU. It is encapsulated from the rest of the core, so that

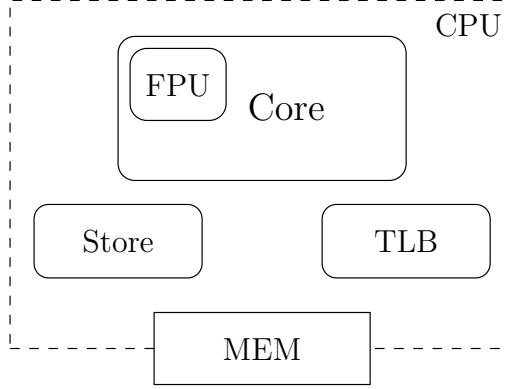


Figure 3.1: New CPU model with FPU

we can define the semantics of x87 instructions independently of the overall model. To this end we have interface components inside the FPU.

$$FPU = \{\dots, acc : FpuRequest|\epsilon, ans : \mathbb{B}^*|\epsilon\};$$

The field *acc* accumulates all data affiliated with the x87 instruction to be executed. In the field *ans* the result of an operation will be saved. A full definition of *FPU* will be given in chapter 4. We also need to state the *FpuRequest* declaration to enable FPU access.

$$FpuRequest = \{I : Instruction, flags : \mathbb{B}^{16}, data : \mathbb{B}^*, ds : \mathbb{B}^{16}, cs : \mathbb{B}^{16}, ip : \mathbb{B}^{64}, D : \mathbb{B}, m : Mode\};$$

Accordingly an FPU access contains the following parameters:

- *I* - the instruction to be executed including opcode, memory operand offset, prefixes etc.
- *flags* - the current content of *RFLAGS*[15 : 0]
- *data* - the value of the memory operand
- *ds* - the data segment selector of the memory operand
- *cs* - the code segment selector of the current instruction
- *ip* - the instruction pointer offset of the current instruction, zero-extended to 64 bits
- *D* - the current code segment descriptor's *D* bit for determining default address and data width
- *m* - the current mode of the CPU

Whenever an FPU instruction is decoded, these parameters must be set to the appropriate values. Then the FPU transition function $\sigma_{fpu} : FPU \rightarrow FPU$ can be applied on the current x87 configuration to simulate one execution step. Our present model does not recognize floating-point instructions yet, so we have to apply further modifications.

3.2 Decoding x87 Instructions and Operands

Before the FPU can be called, the core must be able to recognize x87 instructions. It also needs to know whether there are memory operands to be fetched. Therefore we modify definitions from the "Instruction Fetch and Decode" chapter in [Deg07] to suit floating-point instructions. First of all we introduce two new operand types *FPM* and *FPS* for x87 operands.

$$\begin{aligned} optype : (\mathbb{B}^{24}, \mathbb{B}^8 | \epsilon, \mathbb{B}, \{1, 2, 3\}) &\rightarrow \{\dots, FPM, FPS\} & [\text{Deg07}] \\ optype(opc, modrm, is64mode, index) &= \dots \end{aligned}$$

As in [Deg07] we will compute the operand type using a table given in Appendix A. All x87 related instruction are listed there as an extension to the opcode tables in [Deg07]. Note that all floating-point operations except one¹ have either operand types *FPM* or *FPS*. Their default operand size is 80 bit. However there are memory operands with deviating sizes that were not considered yet. Thus we extend *opsiz*e as well.

$$\begin{aligned} opsiz : (\mathbb{B}^{24}, \mathbb{B}^8 | \epsilon, \{1, 2, 3\}, \{16, 32, 64\}) &\rightarrow \{\dots, 112, 224, 752, 864, 4096\} & [\text{Deg07}] \\ opsiz(opc, modrm, is64mode, index, dataw) &= \dots \end{aligned}$$

The operand bitsize can be determined from the opcode table by checking the subscripts of the operand type alias. There are x87 state-saving instructions that allow to save the entire floating-point environment to memory and to restore the state based on this data later on. The contiguous bit string containing the information for the FPU components is called *image*. The image size is variable and dependent on the current data width. We denote three different image categories by the aliases *iS* (small image - 14/28 byte), *iM* (medium image - 94/108 byte) and *iL* (large image - 512 byte). They stand for the following bit sizes.

$$\begin{aligned} iS &= \begin{cases} 112 & : \quad dataw = 16 \\ 224 & : \quad \text{else} \end{cases} \\ iM &= \begin{cases} 752 & : \quad dataw = 16 \\ 864 & : \quad \text{else} \end{cases} \\ iL &= 4096 \end{aligned}$$

To learn more about images and state saving/restoring please confer to sections 3.4, 7.2.3 and Appendix C. It is worth mentioning that x87 operand size ranges from 16 to 4096 bit. All x87 instructions need a ModR/M-byte, hence the *needModRM* predicate is updated.

$$\begin{aligned} needModRM : \mathbb{B}^{24} &\rightarrow \mathbb{B} & [\text{Deg07}] \\ needModRM(opc) &= opc2(opc) \neq \epsilon \vee \exists i \in \{1, 2, 3\} : (optype(opc, i) \in \{\dots, FPM, FPS\}) \end{aligned}$$

The function *opdecode* : (*Core*, {1, 2, 3}) → (*Operand*, $\mathbb{B}$) from [Deg07] relates operand type and the actual operand description. Floating-Point operands may be x87 registers, or data residing in memory. As the core shall not access FPU registers directly we only need to assign meaningful decodings to the memory operands. On account of this we add *FPM* (Floating-Point Memory) and *FPS* (Floating-Point Stack) operand types to *opdecode*. Note that Instructions *I* with *I.modrm.mod* = 11 can not reference memory operands.

¹Solely an obsolete version of FNSTSW allows to store the x87 status word to the AX register.

```

1  opdecode : (Core, {1, 2, 3}) → (Operand, ℬ)
2  opdecode (c, i) =
3    let size' = opsize(c.I.opc, c.I.modrm, x64mode(c), i, dataw(c)) in
4    case optype(c.I.opc, c.I.modrm, i) of
5      A → (MEM {sreg=opseg(c), addr=c.I.imm, size=size'}, 1)
6      ...
7      Y → (MEM {sreg=ES, addr=c.R[DI], size=size'}, 1)
8      FPM ∧ c.I.modrm.mod = 11 → (ϵ, 1)
9      FPM ∧ c.I.modrm.mod ≠ 11 → (MEM {sreg=opseg(c), addr=opaddr(c), size=size'}, 1)
10     FPS → (ϵ, 1)
11     SS → (REG {type=SEG, reg=0◦SS, size= 16}, 1)
12     ...
13     ϵ → (ϵ, 1)

```

Finally in the fetch phase the instruction field's operand components are filled with the respective contents gained from decoding. Also the opcode fields are updated with the appropriate data. Therefore we henceforth are able to recognize x87 instructions by examining the opcode. The necessary predicates are defined in section 6 and table B.1 in Appendix B. As we included floating-point operations into the regular operand decoding scheme, floating-point memory operands can be fetched and updated without extra effort using the *readOp* and *writeOp* functions from [Deg07]. All functionality to fetch and decode x87 instructions and operands and to write back result to memory resides in the core unit hence the FPU unit is only required for the actual floating-point calculations.

3.3 Executing x87 Instructions

As described above, the core must deliver all required data to the FPU environment whenever an x87 instruction is decoded. The FPU operation may involve a memory operand, which can be loaded utilising the *readOp* function. After decoding an x87 instruction the suitable function *execFPU* : *Cpu* → ?*Cpu* is called for the execution. Recognizing x87 instructions by the opcode is easy using table A.1 or the predicates defined in chapter 6. These predicates identify the current x87 instruction. They are written in capital letters and are named after the mnemonic of the corresponding operation. Now we want to give the definition for the function *execFPU* that represents the interface between CPU and FPU environment. For information on the syntax see section 2.2 in the notations chapter.

```

1  execFPU : Cpu → ?Cpu
2  execFPU(cpu) =
3    let c = cpu.core in
4    let MP = c.CR(0)[1] in      // 'monitor coprocessor' control bit
5    let EM = c.CR(0)[2] in      // 'emulate coprocessor' control bit
6    let TS = c.CR(0)[3] in      // 'task switched' control bit
7    let mf = es(cpu.fpu) in
8    |> (EM=1 ∧ WAIT(c.I)) ∨ (TS=1 ∧ (WAIT(c.I) ∨ (WAIT(c.I) ∧ MP=1))) →
9      PASS cpu{core = core{excp = eNM}}
10   |> (mf=1 ∧ nonctrl(instr)) → PASS cpu{core = core{excp = eMF}}
11   |> WAIT(c.I) → if mf=1 then PASS cpu{core = core{excp = eMF}}
12                      else NORMAL cpu

```

```

13 |> otherwise →
14   let? (cpu1,op) = if xread(c.I) then readOp(cpu, c.I.op1)
15                     else NORMAL (cpu, 0) in
16   let? (cpu2,op1) = if FXRSTOR(c.I) then xmmrstor(cpu1,op)
17                     else NORMAL (cpu1, op) in
18   let c1 = cpu2.core in
19   let fpu1 = reqFPU(c1,op1) in
20   let fpu2 =  $\sigma_{fpu}$ (fpu1) in
21   let res = fpu2.ans in
22   let (c2,res1) = if FXSAVE(c.I) then xmmsave(c1,res) else (c1,res) in
23   let cpu3 = cpu2{core = c2{fpu = fpu2}} in
24   |> xupdm(c.I) → writeOp(cpu3, c.I.op1, res1)
25   |> xupdf(c.I) → NORMAL cpu3{core = core{RFLAGS = RFLAGS[63:16] ◦ res1}}
26   |> otherwise → NORMAL cpu3

```

At first we have to catch some exceptions affiliated with the EM (emulate coprocessor), TS (task switched) and MP (monitor coprocessor) bits in the CR0 control register.

FPU requests can not always be served because there may be pending floating-point exceptions from the last non-control x87 instruction executed. Therefore the CPU must check for exceptions before the execution of another non-control x87 instruction. To this end we test for unmasked x87 exceptions. $es : FPU \rightarrow \mathbb{B}$ is used to determine the FPU exception status. $nonctrl : Instruction \rightarrow \mathbb{B}$ is a predicate which identifies non-control x87 instructions by examining the two x87-opcode bytes. The exact definitions of these functions can be found in sections 4.4 and 6.1.1 respectively. Furthermore $WAIT : Instruction \rightarrow \mathbb{B}$ identifies a WAIT instruction, which forces the processor to check for pending floating-point exceptions and to handle them.

Then - when necessary - the above function loads a memory operand for the FPU operation. From the last section we can deduce that field *op1* of *Instruction* is either a memory operand MEM x or ϵ for x87 instructions. The predicate *xread* states whether a memory operand must be read. It is defined in section 6.1.2. Predicates *FXSAVE*/*FXRSTOR* being defined in section 6.1.1 identify special cases which are discussed in the next subsection.

To prepare the FPU access, *reqFpu* is called. This function determines the required parameters of the request and sets them accordingly in the field *acc*. Memory operands are saved in the *data* component. A copy of the lower 16 bits of the *RFLAGS* register is stored as well as the x87 exception pointers (data and instruction addresses), D bit and the current mode of the CPU. Now the state transition function σ_{fpu} can be applied on the floating-point unit. Its complete semantics is defined in section 7. The operation's result is then contained in the FPU configuration's field *ans*.

The predicates *xupdf*, *xupdm* : *Instruction* $\rightarrow \mathbb{B}$ signal whether an x87 instruction updates the *RFLAGS* register or a memory operand respectively. Those predicates' exact definitions can be found in section 6.1.2.

Accordingly the memory or the flags are updated when required.

```

1 reqFPU : (Cpu,  $\mathbb{B}^*$ )  $\rightarrow$  FPU
2 reqFPU (cpu, op) =
3   let c = cpu.core in
4   let (ddesc, dex, dsel) = if (c.I.op1  $\neq \epsilon$ ) then segment(cpu, c.I.op1.sreg)
5                           else ( $\epsilon$ ,  $\epsilon$ ,  $\epsilon$ ) in
6   let (idesc, iex, isel) = segment(cpu, CS) in
7   let ioff = zeroext64(c.oldRIP[ipw(c)-1:0]) in

```

```

8      let d = idesc.D in
9      let M = mode(c) in
10     let f = c.RFLAGS[15:0] in
11     let fpuacc = FPU {I=c.I, flags=f, data=op, ds=dsel, cs=isel, ip=ioff, D=d, m=M} in
12     fpu{acc = fpuacc}

```

Here $segment : (Cpu, \mathbb{B}^3) \rightarrow (\mathbb{B}^{128}, \mathbb{B}^8, \mathbb{B}^{16})$ returns the segment descriptor, the affiliated exception and the segment selector for a certain segment register. $ipw(c)$ stands for the instruction pointer address width as defined in [Deg07].

3.4 XMM state Save and Restore

For FXSAVE and FXRSTOR instructions we must consider not only x87 components, but also the XMM state. The respective instructions save or restore this information together with the FPU state to or from memory using a joined image format. However inside the FPU we can not access the XMM components, thus in *execFPU* the data transmitted to or received from the FPU must be further manipulated. At a save of the configuration we must merge the XMM data with the FPU data to build the final image. When the whole floating-point and media state shall be restored we have to strip the respective XMM data from the image first, update the corresponding XMM components and transmit the remaining floating-point state to the FPU. These image manipulations are executed using $xmmsave : (Core, \mathbb{B}^{1280}) \rightarrow (Core, \mathbb{B}^{4096})$ and $xmmrstor : (Cpu, \mathbb{B}^{4096}) \rightarrow ?(Cpu, \mathbb{B}^{1280})$. Being called from *execFPU* these functions extract FPU information from or add non-FPU information to the memory images and also carry out the XMM part of the instruction execution. For clarity, illustrations of the XMM image formats are given in Appendix C.

```

1  xmmrstor : (Cpu,  $\mathbb{B}^{4096}$ )  $\rightarrow$  ?(Cpu,  $\mathbb{B}^{1280}$ )
2  xmmrstor(cpu, image) =
3      let c = cpu.core in
4      let fpuim = image[1279:0] in
5      let mask = if (mxcsr_mask(c) = 0x00000000) then 0x0000FBFF else mxcsr_mask(c) in
6      |> c.CR(4)[9] = 0  $\rightarrow$  NORMAL (cpu, fpuim)
7      |> (mask ^ image[223:192]  $\neq$  image[223:192])  $\rightarrow$  PASS cpu{core = c{excp = eGP}}
8      |> otherwise  $\rightarrow$ 
9          let c1 = c{MXCSR = image[223:192]} in
10         let c2 = |> c.EFER[14]=1  $\wedge$  cpl(c)=0  $\wedge$  x64mode(c)  $\rightarrow$  c1
11             |>  $\overline{x64mode(c)}$   $\rightarrow$ 
12                 c1{XMM = XMM[  $\forall i \in \mathbb{N}_8 : i \rightarrow$  image[128 $\cdot$ (i+1)+1279:128 $\cdot$ i+1280] ]}
13             |> otherwise  $\rightarrow$ 
14                 c1{XMM = XMM[  $\forall i \in \mathbb{N}_{16} : i \rightarrow$  image[128 $\cdot$ (i+1)+1279:128 $\cdot$ i+1280] ]} in
15         NORMAL (cpu{core = c2}, fpuim)

```

Here it is assumed, that $XMM : \mathbb{N}_{16} \rightarrow \mathbb{B}^{128}$ and $MXCSR \in \mathbb{B}^{32}$ represent the 16 XMM registers and the XMM control/status register of the core. The FPU relevant information is contained in the lower 160 bytes of the image. $CR(4)[9]$ contains the OSFXSR bit which toggles the saving/restoring of the XMM registers. We must consider some special characteristics of the *MXCSR* register. Several bits in this register are reserved and may only be written with zeroes. These zero bits

signal that the associated features are not available. To gain information, which bits are reserved, software may examine the field `MXCSR_MASK` in the `FXSAVE` memory image at bytes 28-31. We assume function $mxcsr_mask : Core \rightarrow \mathbb{B}^{32}$ to compute this mask bit field on the basis of the current configuration and machine-specific information. When there is an attempt to write ones to reserved bits in the `MXCSR`, a General Protection exception must be raised. To this end the current `MXCSR_MASK` is compared with the new data for the `MXCSR`. Furthermore the XMM restoration depends on bit 14 of the `EFER` register (FFXSR bit) and on the current mode. The FFXSR bit enables fast save and restore of the XMM state as it omits restoring the XMM registers when `FXRSTOR` is executed in 64-bit mode with privilege level 0. Note that regardless of FFXSR bit only in 64-bit mode all 16 XMM registers can be restored. According to the same rules the `FXSAVE` image is built using the x87 environment image received from the FPU.

```

1 xmmsave : (Core,  $\mathbb{B}^{1280}$ )  $\rightarrow$  (Core,  $\mathbb{B}^{4096}$ )
2 xmmsave(c, image) =
3   let img = image[1279:256]  $\circ$  mxcsr_mask(c)  $\circ$  c.MXCSR  $\circ$  image[191:0] in
4   |> c.CR(4)[9]=0  $\rightarrow$  (c,  $0^{2816} \circ$  image)
5   |> c.EFER[14]=1  $\wedge$  cpl(c)=0  $\wedge$  x64mode(c)  $\rightarrow$  (c,  $0^{2816} \circ$  img)
6   |> x64mode(c)  $\rightarrow$  (c,  $0^{1792} \circ$  c.XMM(7)  $\circ$  c.XMM(6)  $\circ$  ...  $\circ$  c.XMM(1)  $\circ$  c.XMM(0)  $\circ$  img)
7   |> otherwise  $\rightarrow$  (c,  $0^{768} \circ$  c.XMM(15)  $\circ$  c.XMM(14)  $\circ$  ...  $\circ$  c.XMM(1)  $\circ$  c.XMM(0)  $\circ$  img)

```

Note that for `FXSAVE` and `FXRSTOR` the x87 part is located in the image's lower 160 bytes or 1280 bits respectively. Now the interface of the Floating-Point Unit is complete. We have obtained a framework to decode floating-point instructions and generate the corresponding x87 requests. The next chapters describe how these requests are processed by the FPU in order to return a result to the core.

3.5 Handling Exceptions

As one can already see above, exception handling is somewhat special for x87 instructions. The floating-point exceptions are not reported to the CPU immediately after their occurrence. Instead they are not detected by the processor before executing the next `WAIT` or non-control floating-point instruction. That means that control instructions may manipulate the FPU configuration in the meantime before the exception is recognized. We implemented this behaviour in the beginning of the `execFPU` function. Besides the numerical eMF exceptions FPU instructions may cause the following general purpose exceptions.

- eDB - Debug
- eBP - Breakpoint
- eUD - Invalid-Opcode
- eNM - Device-Not-Available
- eDF - Double-Fault
- eSS - Stack

- eGP - General-Protection
- ePF - Page-Fault
- eAC - Alignment-Check
- eMC - Machine-Check

The origins of these exceptions lie outside of the FPU. They mostly occur when accessing memory to read operands or write results or due to improper use of the instructions. The checking and handling of them is already given in [Deg07]. However we have to consider a speciality for floating-point instructions.

Whenever an exception causing a repetition of the respective x87 instruction occurs after the FPU finished computation and returned the result to the CPU, the old FPU configuration has to be restored before the instruction's execution can be repeated. Everything else would result in a corrupted FPU state and a likely erroneous result. Therefore arrangements to save and restore the floating-point unit component have to be incorporated in the core's exception handling routines.

Chapter 4

Configuration

In this section we will define the data structures needed to describe the semantics of floating-point instructions. Besides the parameters, operands, etc. submitted through the core's request this environment will also include the content of the x87 data registers as well as the non-data FPU state. To describe a current configuration of the floating-point unit we complete the definition of *FPU*.

$$FPU = \{fpr : \mathbb{N}_8 \rightarrow \mathbb{B}^{80}, opc : \mathbb{B}^{11}, dp : \mathbb{B}^{64}, ip : \mathbb{B}^{64}, fsw : \mathbb{B}^{16}, fcw : \mathbb{B}^{16}, ftw : \mathbb{B}^{16}, acc : Request|\epsilon, ans : Result|\epsilon\};$$

In the *Cpu* type an *FPU* typed unit with the name *fpu* is defined. For comfort we abbreviate this configuration by *f*. Its components in detail are:

- $f.fpr : \mathbb{N}_8 \rightarrow \mathbb{B}^{80}$ - the *Floating-Point Register File* represents the eight physical registers available for x87 data in the FPU. Register indexes are mapped on the respective current 80-bit register values.
- $f.ip \in \mathbb{B}^{64}$ - The *x87 Instruction Pointer* stores the logical or linear memory address of the last non-control x87 instruction executed, depending on the current mode and effective operand size.
- $f.dp \in \mathbb{B}^{64}$ - The *x87 Data Pointer* stores the logical or linear address of the last memory operand accessed by a non-control x87 instruction, depending on the current mode and effective operand size.
- $f.opc \in \mathbb{B}^{11}$ - The *x87 Opcode* is an extract of the last non-control x87 instruction's original first two x87-opcode bytes. It saves all the necessary bits to determine the floating-point function to be executed.
- $f.fsw \in \mathbb{B}^{16}$ - The *x87 Status Word Register* holds information to describe the current state of the FPU. This includes status flags, condition code bits and the stack pointer.
- $f.fcw \in \mathbb{B}^{16}$ - The *x87 Control Word Register* holds bits to control FPU behaviour, i.e. to set exception mask, rounding mode and precision control bits.

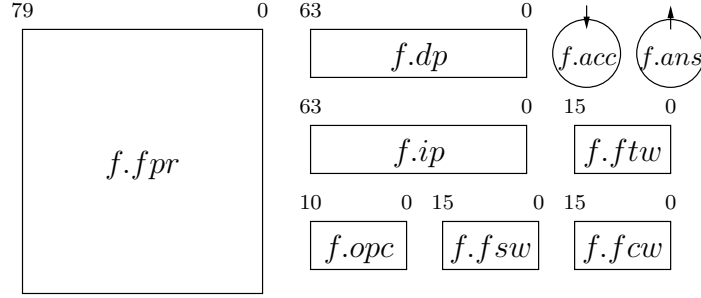


Figure 4.1: FPU configuration components

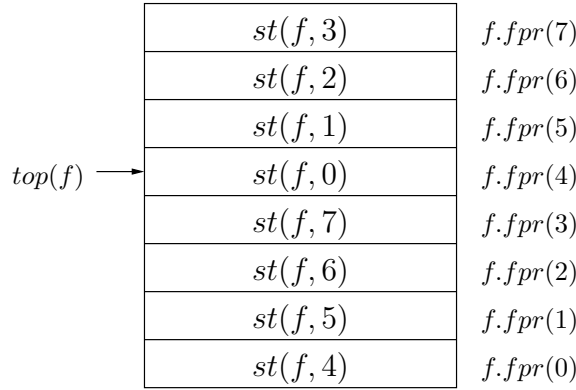


Figure 4.2: x87 Physical and Stack Registers

- $f.ftw \in \mathbb{B}^{16}$ - The *x87 Tag Word Register* provides means to manage floating-point registers. Two Bits for each of the eight floating-point registers signal whether the respective register contains meaningful data.
- $f.acc \in FpuRequest|\epsilon$ - The FPU field to store current instruction information.
- $f.ans \in \mathbb{B}^*|\epsilon$ - The FPU field to store the result of requested operations.

Figure 4.1 shows the FPU components at a glance.

4.1 x87 Stack Registers

The floating-point registers are rather used as a stack than being addressed directly by their index. That means there is a top-of-stack pointer $top(f)$ stored in the x87 status word referring to the element at the top of the stack.

$$top(f) = f.fsw[13 : 11]$$

These three bits encode the register index of the top-of-stack floating-point register. See figure 4.2 for an illustration of the stack organisation.

We denote the register index associated with stack element $j \in [0 : 7]$ in FPU state f by stack register index

$$sri(f, j) = (j + \langle top(f) \rangle) \bmod 8.$$

Hence the content of stack element j is defined by

$$st(f, j) = f.fpr(sri(f, j)).$$

These functions supply us with the possibility to switch between stack and register context easily. The stack-top register is denoted by $st(f, 0)$ for instance.

4.2 x87 Tag Word

With the help of the x87 tag word one of four different tags are assigned to each floating-point data register. These tags specify attributes for the registers' contents, namely:

- *empty* - The register contains no meaningful data.
- *valid* - The register contains a normal floating-point number.
- *zero* - The register contains zero.
- *special* - The register contains special data, such as denormal numbers, NaNs, infinity or unsupported number formats.

The tag bits for register $i \in [0 : 7]$ are saved in $f.ftw[2 \cdot i + 1 : 2 \cdot i]$. To refer to stack elements we denote the tag of stack register $j \in [0 : 7]$ by

$$tag(f, j) = f.ftw[2 \cdot sri(f, j) + 1 : 2 \cdot sri(f, j)].$$

Consequently we define the following predicates for stack registers j :

$$\begin{aligned} empty(f, j) = 1 &\Leftrightarrow tag(f, j) = 11 \\ valid(f, j) = 1 &\Leftrightarrow tag(f, j) = 00 \\ zero(f, j) = 1 &\Leftrightarrow tag(f, j) = 01 \\ special(f, j) = 1 &\Leftrightarrow tag(f, j) = 10 \end{aligned}$$

4.3 x87 Status Word

The status word register signals information about the current state of the Floating-Point Unit. This comprises exception signalling, condition code bits, the top-of-stack pointer and the busy bit.

$rc(f)$	Rounding Mode	$pc(f)$	Precision Mode
00	Round to nearest	00	Single precision
01	Round down	01	<i>undefined</i>
10	Round up	10	Double precision
11	Round toward zero	11	Double-extended precision

Table 4.1: Rounding and Precision Modes

We define aliases for the particular status word bits as given below.

$busy(f) = f.fsw[15]$	(busy bit)
$cc(f)[3 : 0] = f.fsw[14] \circ f.fsw[10 : 8]$	(condition code)
$top(f) = f.fsw[13 : 11]$	(top-of-stack pointer)
$es(f) = f.fsw[7]$	(exception status)
$sf(f) = f.fsw[6]$	(stack fault)
$pe(f) = f.fsw[5]$	(precision exception)
$ue(f) = f.fsw[4]$	(underflow exception)
$oe(f) = f.fsw[3]$	(overflow exception)
$ze(f) = f.fsw[2]$	(zero exception)
$de(f) = f.fsw[1]$	(denormalized operand exception)
$ie(f) = f.fsw[0]$	(invalid operation exception)

4.4 x87 Control Word

The control word register is used to customize floating-point operations to certain extents. Software can influence rounding modes and precision settings for the calculations as well as mask occurring floating exceptions. There are four different rounding and three precision modes as defined in table 4.1 using the following functions.

$rc(f) = f.fcw[11 : 10]$	(rounding control)
$pc(f) = f.fcw[9 : 8]$	(precision control)

Rounding control enables the user to choose the way floating-point numbers are rounded by the FPU. For some instructions the actual rounding mode is fixed, so the rounding control has no effect. We denote the current rounding mode by $rm(f) : FPU \rightarrow \mathbb{B}^2$.

$$rm(f) = \begin{cases} 00 & : FPREM1(f) \\ 11 & : FISTTP(f) \vee FPREM(f) \\ rc(f) & : \text{otherwise} \end{cases}$$

$FISTTP(f)$, $FPREM(f)$ and $FPREM1(f)$ are instruction predicates. They are defined in chapter 6 and identify the instructions corresponding to their names.

Under certain circumstances precision control governs the precision of the significand which we denote by $p(f)$.

$$p(f) = \begin{cases} 24 & : (precctrl(f) \wedge pc(f) = 00) \vee ((FST(f) \vee FSTP(f)) \wedge ressize(f) = 32) \\ 53 & : (precctrl(f) \wedge pc(f) = 10) \vee ((FST(f) \vee FSTP(f)) \wedge ressize(f) = 64) \\ 64 & : \text{otherwise} \end{cases}$$

The predicate $precctrl : FPU \rightarrow \mathbb{B}$ checks for these circumstances, using instruction predicates from section 6.1, where also $FST(f)$, $FSTP(f)$ and $ressize(f)$ are defined.

$$precctrl(f) = add(f) \wedge \overline{FIADD(f)} \vee sub(f) \wedge \overline{FISUB(f)} \vee mul(f) \wedge \overline{FIMUL(f)} \vee div(f) \wedge \overline{FIDIV(f)} \vee FSQRT(f)$$

We can not directly control the size of the exponent $n(f)$. It is determined in the following way.

$$n(f) = \begin{cases} 8 & : (FST(f) \vee FSTP(f)) \wedge ressize(f) = 32 \\ 11 & : (FST(f) \vee FSTP(f)) \wedge ressize(f) = 64 \\ 15 & : \text{otherwise} \end{cases}$$

That means that the x87 specification allows computations that deviate from the IEEE floating-point standard by mixing small significands with exponents in extended formats.

The exception mask bits are aligned in the control word according to their status word counterparts. Though there are corresponding mask bits in the control word, exception status and stack fault bits can not be masked. On account of these facts we define the floating-point exception masks as

$$masks(f) = f.fcw[5 : 0].$$

We introduce aliases for the particular mask bits.

$$\begin{aligned} pm(f) &= masks(f)[5] && \text{(precision exception mask bit)} \\ um(f) &= masks(f)[4] && \text{(underflow exception mask bit)} \\ om(f) &= masks(f)[3] && \text{(overflow exception mask bit)} \\ zm(f) &= masks(f)[2] && \text{(zero exception mask bit)} \\ dm(f) &= masks(f)[1] && \text{(denormalized operand exception mask bit)} \\ im(f) &= masks(f)[0] && \text{(invalid operation exception mask bit)} \end{aligned}$$

Inspired by the DLX exception handling part of [MP00], we define corresponding functions for the floating-point exceptions.

$$\begin{aligned} eca(f) &= pe(f) \circ ue(f) \circ oe(f) \circ ze(f) \circ de(f) \circ ie(f) \\ &= f.fsw[5 : 0] \end{aligned}$$

stands for the floating-point exception cause vector, which collects all occurring exceptions sorted by their cause. All of these exceptions may be masked, that is they are not reported to the processor's exception handler. To determine occurring unmasked exceptions a masked cause is calculated.

$$\forall i \in [0 : 5] : mca(f)[i] = eca(f)[i] \wedge masks(f)[i]$$

An x87 stack fault is signalled together with an invalid operation exception. It can not be masked directly, however no stack fault is reported to the core when $im(f) = 1$. The exception status bit indicates for a given FPU state f that the last non-control x87 instruction caused an unmasked exception. This is specified by the following statement:

$$es(f) \equiv \bigvee_{i=0}^5 mca(f)[i]$$

In addition we denote the case that only exceptions occurred whose mask bits were active by the *masked exception* predicate.

$$\begin{aligned} mexcp : FPU &\rightarrow \mathbb{B} \\ mexcp(f) &= \overline{es(f)} \wedge (\bigvee_{i=0}^5 eca(f)[i] = 1) \end{aligned}$$

When there are only masked exceptions, that means that there are active exception flags but $es(f) = 0$, the FPU returns a default result for the given situation.

The busy bit of the FPU is obsolete and serves only backward compatibility purposes for old 8087 coprocessors. It holds that

$$busy(f) \equiv es(f).$$

Note that all exception flags and the stack fault flag are sticky bits. That means they will keep there values until they are altered by software.

4.5 x87 Instruction Information

Besides the x87 environment the *FPU* configuration also contains the fields *acc* and *ans*. The latter one is only used to store the answer to the core's request. In the *FpuRequest*-type *acc* field we find all the information for the current instruction. To access them easily we define shorthand functions in the following. Note that all these functions exist provided that

$$f.acc \neq \epsilon$$

holds as a precondition.

4.5.1 Instruction components

The instruction that caused a floating-point unit request is included in the respective data. I incloses information about the operation to be executed, its operands and activated prefixes. We denote the current instruction of the FPU by

$$\begin{aligned} I : FPU &\rightarrow Instruction \\ I(f) &= f.acc.I \end{aligned}$$

For the sake of comfort we also define an alias for the current opcode which is included in $I(f)$.

$$\begin{aligned} opc : FPU &\rightarrow \mathbb{B}^{16} \\ opc(f) &= I(f).opc[7 : 0] \circ I(f).modrm \end{aligned}$$

The original x87-opcode is composed of the instruction's first opcode byte and the *modr/m* byte. As $opc[15 : 8] \in \{0xD8, \dots, 0xDF\}$ and thus $opc(f)[15 : 11] = 11011$ for all FPU related instructions except FWAIT and FXSAVE/FXRSTOR, only $opc(f)[10 : 0]$ needs to be saved in $f.opc$ in case of non-control floating-point instructions.

Instructions' corresponding operand data and address widths depend on the current mode and the default widths of the current code segment. They may be modified by prefixes as well. Address and effective operand sizes must be considered at several points in the FPU. That's why we define the following predicates with type $FPU \rightarrow \mathbb{B}$. They imply that a certain prefix was set for the active instruction.

$$\begin{aligned} RexW(f) &= \begin{cases} I(f).rex[3] & : I(f).rex \neq \epsilon \\ 0 & : \text{otherwise} \end{cases} && (\text{REX.W prefix}) \\ 66h(f) &= (0x66 \in I(f).legacy) && (0x66 \text{ prefix}) \\ 67h(f) &= (0x67 \in I(f).legacy) && (0x67 \text{ prefix}) \end{aligned}$$

REX.W and 0x66 prefixes exclusively affect the choice of image formats when saving or restoring x87 environment. A 0x67 prefix solely impacts on x87 instructions which are accessing memory operands. See the following subsections for details. The *FpuRequest* type also contains the *D*-bit of the instructions code segment descriptor, which toggles the default address and data width.

$$\begin{aligned} D : FPU &\rightarrow \mathbb{B} \\ D(f) &= f.acc.D \end{aligned}$$

The current mode of the CPU is denoted by $m(f)$.

$$\begin{aligned} m : FPU &\rightarrow Mode \\ m(f) &= f.acc.m \end{aligned}$$

4.5.2 Image Formats

Several x87 instructions may save the floating-point environment to memory or restore it from there. To this end memory image formats are used. Their structure depends on the current mode $m : Mode$, default bitwidth bit D as well as on REX.W and 0x66 prefixes. By FXSAVE/FXRSTOR not only x87 environment but also the 64-bit MMX and 128-bit XMM instruction environment states are saved or restored alternatively. For this purpose two formats may be chosen via REX.W prefix. For FLDENV, FNSTENV, FNSAVE, and FRSTOR four different formats are available in each case. The structure of these formats is depicted in Appendix B. We determine which format shall be used by the following predicates with type $FPU \rightarrow \mathbb{B}$, referring to the current mode and effective operand size.

$$\begin{aligned} prot32(f) &= \overline{x16_mode(m(f))} \wedge ((D(f) \oplus 66h(f)) \vee RexW(f)) \\ prot16(f) &= \overline{x16_mode(m(f))} \wedge ((D(f) \oplus 66h(f)) \vee RexW(f)) \\ real32(f) &= x16_mode(m(f)) \wedge \overline{66h(f)} \\ real16(f) &= x16_mode(m(f)) \wedge \overline{66h(f)} \end{aligned}$$

The mnemonics *prot* and *real* stand for Protected/Compatibility Mode and Real/Virtual-8086 Mode. The functions for determining mode are defined in [Deg07]. Figures of the corresponding formats may be found in Appendix C.

4.5.3 Flags and Data

In the field *acc* also the components *flags* and *data* are included. *flags* : $\mathbb{B}^{16}$ holds the lower 16 bits of the core's *RFLAGS* register. From this word only three bits are relevant to the FPU.

$$\begin{aligned} xf : FPU &\rightarrow \mathbb{B} \\ zf(f) &= f.acc.flags[6] && \text{(zero flag)} \\ pf(f) &= f.acc.flags[2] && \text{(parity flag)} \\ cf(f) &= f.acc.flags[0] && \text{(carry flag)} \end{aligned}$$

To conveniently alter these bits we define the following function, which returns the field *f.acc.flags* with changed values for zero, parity and carry flag.

$$\begin{aligned} chflg : (FPU, \mathbb{B}, \mathbb{B}, \mathbb{B}) &\rightarrow \mathbb{B}^{16} \\ chflg(f, z, p, c) &= f.acc.flags[15 : 7] \circ z \circ f.acc.flags[5 : 3] \circ p \circ f.acc.flags[2 : 1] \circ c && \text{(change flags)} \end{aligned}$$

If an instruction references a memory operand, its content will be transmitted using the field *data* of the *FpuRequest* type.

$$\begin{aligned} mop : FPU &\rightarrow \mathbb{B}^* \\ mop &= f.acc.data && \text{(memory operand)} \end{aligned}$$

When there is no memory operand *data* contains 0.

4.5.4 Memory Pointers

An FPU request must also contain the memory addresses of the instruction and of a potential x87 data memory operand, because this information is saved in the x87 environment for exception handling purposes. To this end the *FpuRequest* type comprises field *ds* for storing the data segment selector, field *cs* for storing the code segment selector and *ip* for storing the current instruction's instruction pointer. The offset for the memory operand's data pointer does not need to be saved in an extra field as it is already contained in field *I(f).op1*. For *I(f).op1* $\neq \epsilon$ we compose instruction and data pointer in the subsequent way.

$$\begin{aligned} xp : FPU &\rightarrow (\mathbb{B}^{16}, \mathbb{B}^{64}) \\ dp(f) &= (dp.s, dp.o) \\ &= (f.acc.ds, I(f).op1.addr) && \text{(data pointer)} \\ ip(f) &= (ip.s, ip.o) \\ &= (f.acc.cs, f.acc.ip) && \text{(instruction pointer)} \end{aligned}$$

Selectors and offsets of the pointers can be accessed via components *xp(f).s* and *xp(f).o* respectively. Depending on current mode *m*, *D*-bit and 0x67 prefix we can adjust the data and instruction pointers to the right format which is applied to save them in the corresponding x87 environment

registers $f.dp$ and $f.ip$.

$$adjustdp : FPU \rightarrow \mathbb{B}^{64}$$

$$adjustdp(f) = \begin{cases} dp(f).o & : x64_mode(m(f)) \\ zeroext_{64}(dp(f).s \circ dp(f).o[31 : 0]) & : x32_mode(m(f)) \\ & \wedge (D(f) \oplus 67h(f)) \\ zeroext_{64}(dp(f).s \circ dp(f).o[15 : 0]) & : x32_mode(m(f)) \\ & \wedge \overline{(D(f) \oplus 67h(f))} \\ zeroext_{64}(dp(f).s \circ 0^4 +_{32} dp(f).o[31 : 0]) & : x16_mode(m(f)) \\ & \wedge 67h(f) \\ zeroext_{64}(dp(f).s \circ 0^4 +_{20} dp(f).o[15 : 0]) & : x16_mode(m(f)) \\ & \wedge \overline{67h(f)} \end{cases}$$

In 64-bit Mode the 64 offset bits are saved and the data segment selector is omitted. In Protected and Compatibility Mode the concatenation of 16-bit selector and 16 or 32-bit offset are saved. In real mode the 20 or 32-bit linear address is saved. In all modes however only pointers pertaining to non-control x87 instructions are stored in $f.dp$. We adjust the instruction pointer in a similar way.

$$adjustip : FPU \rightarrow \mathbb{B}^{64}$$

$$adjustip(f) = \begin{cases} ip(f).o & : x64_mode(m(f)) \\ zeroext_{64}(ip(f).s \circ ip(f).o[31 : 0]) & : x32_mode(m(f)) \wedge D(f) \\ zeroext_{64}(ip(f).s \circ ip(f).o[15 : 0]) & : x32_mode(m(f)) \wedge \overline{D(f)} \\ zeroext_{64}(ip(f).s \circ 0^4 +_{20} ip(f).o[16 : 0]) & : x16_mode(m(f)) \end{cases}$$

Since the instruction pointer width can not depend on prefixes, only mode and default address width influence the instruction pointer format. As above just non-control x87 instructions cause the FPU to update $f.ip$ with a new instruction pointer.

4.6 Initial Values

To complete the configuration definition we will state the initial values of the FPU components after reset. We denote this first configuration by f^0 . For the several components we have:

$$\begin{aligned} \forall i \in \mathbb{N}_8 : \\ f^0.fpr(i) &= \text{undefined} \\ f^0.opc &= 0^{11} \\ f^0.dp &= 0^{64} \\ f^0.ip &= 0^{64} \\ f^0.fsw &= 0^{16} \\ f^0.fcw &= 0x0300 \\ f^0.ftw &= 1^{16} \\ f^0.acc &= \epsilon \\ f^0.ans &= \epsilon \end{aligned}$$

That means, that rounding control is set to nearest even mode, precision control to double-extended precision mode and all stack registers are tagged as empty. Note also that after reset all floating-point exceptions are unmasked.

Chapter 5

Numbers

The *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE 754-1985, [IE85]) defines data formats and representations used by x87 instructions respectively the floating-point unit. The FPU sticks mostly to the standard but there are also precision modes available which do not conform to it. In the following we will formalize the floating-point number representations defined in the IA-32 instruction set. Here we will profit from the work of Silvia M. Müller and Wolfgang J. Paul who already established a formalization of IEEE numbers (cf. [MP00]). However we will need to extend these definitions to make them fit into our x87 model. We will also care about rounding and conversion of real numbers into floating-point numbers in this chapter as well as BCD numbers and FPU exceptions.

5.1 x87 Floating-Point Data Representations

In general floating-point numbers according to the IEEE standard consist of three parts:

- a sign-bit s - This bit toggles the sign of the number, while a 1 selects negativity.
- an exponent e - This integer is taken to the power of two and scales the floating-point number.
- a significand f - This is a fractional number between zero and two.

The significand scaled with the exponent's power of two and signed according to the sign bit forms the value of a floating-point number. Numbers inside the FPU are represented by bit strings containing binary representations of the three parts of floating-point numbers. Of course the size of these strings is limited so one must decide on a proper distribution of bits to the three components, respectively to exponent and significand as the sign always needs only one bit. In addition there are several precision modes defined in the x87 instruction subset. Therefore we will keep the formalization as general as possible, allowing to extend it to arbitrary bit widths and partitions at a later time. Figure 5.1 shows the three floating-point formats of the FPU. Sticking to [MP00] we

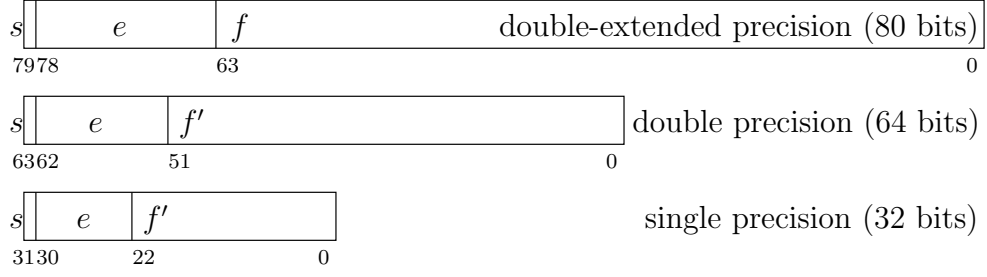


Figure 5.1: The floating-point formats of the FPU

call the size of the exponent n and the size of the significand p .¹ Additionally we denote the total bit width of a floating-point number by m .

$$m = \begin{cases} 80 & : (n, p) = (15, 64) \\ n + p & : \text{otherwise} \end{cases}$$

Obviously the double-extended precision with 15-bit exponent and 64-bit significand plays an extra role compared to all other precision modes. This has to do with the way significands are normally stored in IEEE 754 number bit strings. For a p -bit significand only p' bits are saved.

$$p' = \begin{cases} 64 & : (n, p) = (15, 64) \\ p - 1 & : \text{otherwise} \end{cases}$$

That is for most bit patterns one significand bit is not saved. We call this the *hidden bit* $h : \mathbb{B}^m \rightarrow \mathbb{B}$ for a floating-point representation bit string $x \in \mathbb{B}^m$. Strangely this hidden bit is explicit for the 80-bit double-extended format. For all other formats it is defined by the exponent. We can easily give shorthand functions for exponent and sign of x .

$$\begin{aligned} s : \mathbb{B}^m &\rightarrow \mathbb{B} \\ s(x[m-1:0]) &= x[m-1] \\ e : \mathbb{B}^m &\rightarrow \mathbb{B}^n \\ e(x[m-1:0]) &= x[m-2:p'] \end{aligned}$$

Then the hidden bit is defined as:

$$h(x[m-1:0]) = \begin{cases} x[63] & : (n, p) = (15, 64) \\ 1 & : (n, p) \neq (15, 64) \wedge e(x) \neq 0^n \\ 0 & : \text{otherwise} \end{cases}$$

Now let $f'(x)$ be the significand bits saved explicitly in the binary floating-point number representation x

$$\begin{aligned} f' : \mathbb{B}^m &\rightarrow \mathbb{B}^{p'} \\ f'(x[m-1:0]) &= x[p'-1:0] \end{aligned}$$

¹Note that for the x87 model many definitions depend on the particular values of n or p . However for the sake of simplicity and because mostly double-extended precision is used, we want to keep the dependance on these parameters implicit as long as it is obvious which precision values should be used. We will make the dependance explicit and declare n and p as arguments to the respective function only when results depend on the $n(f)$ or $p(f)$ settings.

Then the hidden bit is the most significant bit of the significand $f(x)$ which is composed in the following way.

$$\begin{aligned} f : \mathbb{B}^m &\rightarrow \mathbb{B}^p \\ f(x[m-1:0]) &= h(x) \circ f'(x)[p-2:0] \end{aligned}$$

Apart from floating-point numbers also special values can be encoded with x87 numbers. These are:

- positive and negative infinity
- Not-a-Number symbolic values (NaNs)

To be able to handle infinities we explicitly extend the real numbers by the infinity symbol.

$$\mathbb{R}_\infty = \mathbb{R} \cup \{+\infty, -\infty\}$$

Calculations with these numbers are done in the intuitive way known from mathematics. One could replace the ∞ symbols with variables tending to infinity and then obtain the result by computing the limit of the term under consideration. Nevertheless some of these limits are undefined. Hence we will be confronted with exceptional situations and special cases later on.

Not-a-Number symbolic values are introduced to handle these and other exceptions because under several cases it is necessary to return a result although the actual operation generates an undefined result respectively an error. Programmers can embed an error code in the Not-a-Number value. This is useful for identifying the type of error that occurred. There are also predefined indefinite value NaNs for every data format. Furthermore two types of NaNs exist:

- Signaling NaNs (SNaN) - These cause an x87 exception when they are used as floating-point operands.
- Quiet NaNs (QNaN) - These do not cause an x87 exception when they are used as floating-point operands. Floating-point operations involving QNaNs generate another QNaN as the result.²

To represent NaNs we define the *NaN* data type that comprises all SNaN and QNaN values for a significand precision p .

$$NaN = QNaN \mathbb{B}^{p-1} \mid SNaN \mathbb{B}^{p-1}$$

The $p-1$ -bit binary value accompanying the NaN contains the sign bit followed by the $p-2$ -bit error code belonging to the NaN.

Now we would like to define the value of a binary floating-point representation $x \in \mathbb{B}^m$. For this purpose we first have to evaluate the values of exponent and significand. While the significand is in plain binary fractional format exhibiting one integer bit, the exponent is stored in a special *biased*

²Also operations with SNaNs generate further NaNs when the FPU invalid operation exception is masked.

integer format. This number representation allows to introduce negative number while avoiding two's complement numbers. The value of an exponent e is then given by $\llbracket e \rrbracket_{bias}$.

$$\llbracket \cdot \rrbracket_{bias} : \mathbb{B}^n \rightarrow \mathbb{Z}$$

Note that this representation is defined only for exponents $e[n-1:0] \notin \{0^n, 1^n\}$. Then the biased value is computed by subtracting the bias $2^{n-1} - 1$ from the binary value of the bit string.

$$\llbracket e[n-1:0] \rrbracket_{bias} = \langle e[n-1:0] \rangle - (2^{n-1} - 1)$$

As $e[n-1:0] \notin \{0^n, 1^n\}$ the minimal and maximal values for the exponent are:

$$\begin{aligned} e_{min} &= -2^{n-1} + 2 \\ e_{max} &= 2^{n-1} - 1 \end{aligned}$$

Thusly we can now define the value $\llbracket x \rrbracket$ of x87 floating-point representations x . We want to give the definition for both the complete bit-string as well as the decomposed version.

$$\begin{aligned} \llbracket \cdot \rrbracket : \{\mathbb{B}, \mathbb{B}^n, \mathbb{B}^p\} &\rightarrow \mathbb{R}_\infty | NaN \\ \llbracket \cdot \rrbracket : \mathbb{B}^m &\rightarrow \mathbb{R}_\infty | NaN \end{aligned}$$

Upgrading the definition from [MP00] we yield the floating-point value for sign $s \in \mathbb{B}$, biased exponent $e \in \mathbb{B}^n$ and significand $f \in \mathbb{B}^p$ in the following way

$$\llbracket s, e, f \rrbracket = \begin{cases} (-1)^s \cdot 2^{\llbracket e \rrbracket_{bias}} \cdot \langle f[p-1].f[p-2:0] \rangle & : e \notin \{0^n, 1^n\} \\ (-1)^s \cdot 2^{e_{min}} \cdot \langle f[p-1].f[p-2:0] \rangle & : e = 0^n \\ (-1)^s \cdot \infty & : e = 1^n \wedge f[p-2:0] = 0^{p-1} \\ QNaN \ s \circ f[p-3:0] & : e = 1^n \wedge f[p-2:0] \neq 0^{p-1} \wedge f[p-2] = 1 \\ SNaN \ s \circ f[p-3:0] & : e = 1^n \wedge f[p-2:0] \neq 0^{p-1} \wedge f[p-2] = 0 \end{cases}$$

Note that this definition does not check for $e = 1^n \Rightarrow f[p-1] = 1$. Thus also triples in an unsupported format are assigned to their pseudo-values.³ For contiguous bit strings $x \in \mathbb{B}^m$ we simply set

$$\llbracket x[m-1:0] \rrbracket = \llbracket s(x), e(x), f(x) \rrbracket.$$

The first row of the case distinction above stands for the standard case of normal numbers. These are floating-point numbers with $e \notin \{0^n, 1^n\}$ and the hidden integer bit being 1. That means the significand is always in the range of $[1, 2)$ for normal numbers.⁴ In contradiction there are denormal numbers to represent tiny values. For denormal numbers the exponent is minimal due to $e = 0^n$

³We do not include cases which cause undefined results for unsupported formats as they only occur for the double-extended precision with $(n, p) = (15, 64)$. Furthermore we would not like to depend the general definition of the floating-point value on the support preferences of the FPU. Instead we care about unsupported formats later by a separate predicate.

⁴Actually according to the definition the value may also be in range $[0, 1)$ for $(n, p) = (15, 64)$ when $f[p-1] = 0$, but these so-called unnormal numbers are not supported by the floating-point unit.

and the hidden integer bit is 0.⁵ This allows a range of $[0, 1)$ for the significand and thus adds an area of very small numbers to the set of representable values. We detect denormal numbers with the following predicate.

$$\begin{aligned} dnrml : \mathbb{B}^m | \epsilon &\rightarrow \mathbb{B} \\ dnrml(x[m-1:0]) = 1 &\Leftrightarrow e(x) = 0^n \wedge f'(x) \neq 0^{p'} \end{aligned}$$

Note that this definition excludes zero from being denormal. Furthermore $dnrml(\epsilon) = 0$ holds. Another special case of the floating point value definition is infinity. The predicate $infly(x)$ holds true for all bit strings that represent floating-point infinity. For ϵ it is false.

$$\begin{aligned} infly : \mathbb{B}^m | \epsilon &\rightarrow \mathbb{B} \\ infly(x[m-1:0]) = 1 &\Leftrightarrow e(x) = 1^n \wedge f(x)[p-1:0] = 10^{p-1} \end{aligned}$$

The last exceptional case for floating-point values are NaNs. A bit string contains a quiet NaN when the exponent is all ones and the hidden integer bit is 1 as well as the most significant bit of the significand's fractional part.

$$\begin{aligned} QNaN : \mathbb{B}^m &\rightarrow \mathbb{B} \\ QNaN(x) = 1 &\Leftrightarrow e(x) = 1^n \wedge f(x)[p-1:p-2] = 11 \end{aligned}$$

SNaNs have the same structure like QNaNs besides that the significand bit following the hidden bit is 0. Also the representation of infinity is not an SNaN.

$$\begin{aligned} SNaN : \mathbb{B}^m &\rightarrow \mathbb{B} \\ SNaN(x) = 1 &\Leftrightarrow e(x) = 1^n \wedge f(x)[p-1:p-2] = 10 \wedge f(x) \neq 10^{p-1} \end{aligned}$$

Finally a bit string encodes a NaN if it contains either a QNaN or an SNaN.

$$\begin{aligned} NaN : \mathbb{B}^m &\rightarrow \mathbb{B} \\ NaN(x) = 1 &\Leftrightarrow QNaN(x) \vee SNaN(x) \\ &\Leftrightarrow e(x) = 1^n \wedge h(x) = 1 \wedge f(x) \neq 10^{p-1} \end{aligned}$$

A special case of QNaNs are the predefined x87 indefinite values. They are generated as results when invalid operation exceptions are masked. The indefinite values for each floating-point, integer and BCD format are listed below.

$$\begin{aligned} \text{indefp32} &= 1^{10} \circ 0^{22} & \text{indefint16} &= 1 \circ 0^{15} & \text{indefbcd} &= 1^{18} \circ 0^{62} \\ \text{indefp64} &= 1^{13} \circ 0^{51} & \text{indefint32} &= 1 \circ 0^{31} \\ \text{indefp80} &= 1^{18} \circ 0^{62} & \text{indefint64} &= 1 \circ 0^{63} \end{aligned}$$

At last we have a look at unsupported encodings. These may only occur in the double-extended precision mode as here the hidden bit is not implicit anymore. That is why its value can be zero when it originally was supposed to be one and vice versa. This leads to unsupported floating-point encodings such as pseudo-NaN, pseudo-infinities and unnormal numbers. We identify these encodings via the *unsup* predicate.

$$\begin{aligned} unsup : \mathbb{B}^{80} | \epsilon &\rightarrow \mathbb{B} \\ unsup(x[79:0]) = 1 &\Leftrightarrow e(x) \neq 0^{15} \wedge x[63] = 0 \end{aligned}$$

Now that we learned about floating-point representation the next section is about how to transfer real numbers and infinitely precise results into our discrete number model.

⁵Actually because of the “unhidden” hidden bit for extended-double precision the integer bit of the significand may also be 1. Such numbers are called *pseudo-denormal* numbers. They are accepted as operands by the FPU but never generated as a result.

5.2 Factorings and Rounding

Naturally we can only represent a subset of the real numbers with floating-point bit strings. This follows from finite precision on the one hand and a limited exponent on the other hand. Therefore we must provide means to round real numbers to a representable floating-point number. Luckily most of the necessary definitions can already be found in chapter 7 of [MP00]. All we have to do is adapting these definitions to our x87 model and the various precision modes.

First of all let us investigate the set of representable numbers. In [MP00] this was introduced as $\mathcal{R}$ so we stick to the notation. We extend it with the infinite values. Not-a-Number values are not numbers so they are omitted here.

$$\begin{aligned}\mathcal{R}_\infty &= \mathcal{R} \cup \{+\infty, -\infty\} & [\text{MP00}] \\ &\subsetneq \mathbb{R} \cup \{+\infty, -\infty\}\end{aligned}$$

With the definition of floating-point representation one can easily state the contents of this set for some bit size and partition values m and (n, p) as defined in the preceding section.

$$\mathcal{R}_\infty = \{ \llbracket s, e, f \rrbracket \mid s = s(x), e = e(x), f = f(x), x \in \mathbb{B}^m \wedge \overline{NaN(x)} \wedge \overline{unsup(x)} \}$$

Now we search for a way to assign infinitely precise numbers to representable ones. This reduction of precision is commonly known as rounding. Rounding of floating-point numbers is fully defined in [MP00] so we can rely on the several rounding functions from there.

$$r_{ne,u,d,z} : \mathbb{R} \rightarrow \mathcal{R}_\infty \quad [\text{MP00}]$$

The different rounding modes are defined as follows for $x \in \mathbb{R}$.

$$\begin{aligned}r_u(x) &= \min\{y \in \mathcal{R}_\infty \mid x \leq y\} && \text{(rounding up)} \\ r_d(x) &= \max\{y \in \mathcal{R}_\infty \mid x \geq y\} && \text{(rounding down)} \\ r_z(x) &= \begin{cases} r_d(x) & : x \geq 0 \\ r_u(x) & : x < 0 \end{cases} && \text{(rounding towards zero)} \end{aligned} \quad [\text{MP00}]$$

The definition of rounding to nearest even mode is not as trivial as those above. A real number is rounded to the nearest representable one and when there are two candidates, that means when the number is lying exactly between two representable numbers the even one of both is chosen. [MP00] defines $X_{max}^* = 2^{e_{max}} \cdot (2 - 2^{-p})$ as the smallest number which would be rounded to $2^{e_{max}+1}$ according to these rules, given that $2^{e_{max}+1}$ would be representable at all. Then we can define $r_{ne}(x)$ as follows.

$$r_{ne}(x) = \begin{cases} r_u(x) & : -X_{max}^* < x < X_{max}^* \wedge |x - r_u(x)| < |x - r_d(x)| \\ r_d(x) & : -X_{max}^* < x < X_{max}^* \wedge |x - r_u(x)| > |x - r_d(x)| \\ z & : -X_{max}^* < x < X_{max}^* \wedge |x - r_u(x)| = |x - r_d(x)| \wedge \\ & \quad \exists z \in \mathbb{R}, q \in \mathbb{Z} : z = 2q \cdot 2^{-(p-1)} \wedge z \in \{r_u(x), r_d(x)\} \\ \infty & : X_{max}^* \leq x \\ -\infty & : x \leq -X_{max}^* \end{cases}$$

In the FPU concurring rounding modes and precisions may be selected. Nevertheless perpetually needing to choose the right rounding function and setting the precision parameters to their respective values might get tedious. Thus we introduce a master rounding function r which takes all the necessary information as arguments and selects the right rounding function automatically.

$$r : (\mathbb{R}_\infty, \mathbb{B}^2, \mathbb{N}^+, \mathbb{N}^+) \rightarrow \mathcal{R}_\infty$$

$$r(x, y, u, v) = \begin{cases} r_{ne}(x) & : y = 00 \\ r_d(x) & : y = 01 \\ r_u(x) & : y = 10 \\ r_z(x) & : y = 11 \end{cases} \quad (n := u, p := v)$$

The various inputs are:

- $x \in \mathbb{R}$ - the number to be rounded
- $y \in \mathbb{B}^2$ - the rounding control bits
- $u, v \in \mathbb{N}^+$ - the exponent size and precision controlling parameters.

Observe that all rounding functions implicitly are functions of (n, p) . Setting these parameters to (u, v) therefore changes the results of the rounding functions accordingly. For further comfort we define shorthand rounding functions which allow control of the precision while the exponent size is fixed to 15. This reflects the fact that for some FPU operations the destination precision can be selected while the exponent size is fixed. For $y \in \mathbb{N}^+$ we have:

$$\begin{aligned} r_{ne}^y(x) &\equiv r(x, 00, 15, y) \\ r_d^y(x) &\equiv r(x, 01, 15, y) \\ r_u^y(x) &\equiv r(x, 10, 15, y) \\ r_z^y(x) &\equiv r(x, 11, 15, y) \end{aligned}$$

In addition we supply a function $r_f(x)$ which rounds x according to the settings of the current x87 configuration f .

$$r_f(x) \equiv r(x, rm(f), n(f), p(f))$$

So we have the machinery now to round real numbers to representable ones with respect to a certain precision and rounding control. Anyway these numbers are still real and not in floating-point representation. To this end - as an intermediate step between real numbers and floating-point representation - Silvia M. Müller and Wolfgang J. Paul established factorings in [MP00]. These are triples of numbers that imitate the floating-point number structure. However only the sign is in bit format while exponent and significand are unbounded and infinitely precise integers or real numbers respectively. We comprise all possible factorings in the set $\mathbb{F}$.

$$\mathbb{F} : (\mathbb{B}, \mathbb{Z} \cup \{\infty\}, [0, 2))$$

Similar to earlier definitions we included infinity for the exponent here as well. The value of a factoring (s, e, f) is yielded from $\llbracket s, e, f \rrbracket$.

$$\begin{aligned} \llbracket \rrbracket : \mathbb{F} &\rightarrow \mathbb{R}_\infty \\ \llbracket s, e, f \rrbracket &= (-1)^s \cdot 2^e \cdot f \end{aligned}$$

Note that we overload the $\llbracket \cdot \rrbracket$ -brackets for the floating-point representation and thus mix notation of factorings and bit strings value. However as one can determine the type of the arguments from the context this is more comfortable than establishing several different notations for related subjects. Now in [MP00] we can find the function η which yields an IEEE-normal factoring with a bounded exponent for a given real number $x \in \mathbb{R}_\infty$.

$$\eta : \mathbb{R}_\infty \rightarrow \mathbb{F} \quad [\text{MP00}]$$

IEEE-normality means that for all $x \in \mathbb{R}_\infty$, a certain exponent size n with the corresponding minimal exponent $e_{\min}$ and $\eta(x) = (s, e, f)$ holds:

$$e \geq e_{\min} \wedge f \in [0, 1) \Rightarrow e = e_{\min} \wedge e > e_{\min} \Rightarrow f \in [1, 2) \wedge \llbracket s, e, f \rrbracket = x$$

There are special factorings for zero and infinity. Note that η considers the sign of zero.

$$\begin{aligned} \eta((-1)^s \cdot 0) &= (s, e_{\min}, 0) \\ \eta((-1)^s \cdot \infty) &= (s, \infty, 0) \end{aligned}$$

When we yield the IEEE-normal factoring of a representable number it is only another small step to gain the floating-point representation bit string.

5.3 Converting Factorings and Bit Strings

In the last step of transferring real numbers into IEEE 754 representation we need to build the floating-point bit string from a given IEEE-normal factoring of the rounded number under consideration. We call this *packing* the factoring into a bit string.

$$\begin{aligned} \text{pack} &: (\mathbb{F}, (\mathbb{N}^+, \mathbb{N}^+)) \rightarrow \mathbb{B}^* \\ \text{pack} &: (\mathbb{F}, \{32, 64, 80\}) \rightarrow \mathbb{B}^* \end{aligned}$$

Once again we have to consider various precision modes. We will thus define the packing function for general bit partitions (n, p) but we will also define a shorthand version for the common precision modes of the FPU. We assume that our input factoring (s, e, f) is an IEEE-normal factoring regarding (n, p) and that it is representable. That implies $\llbracket s, e, f \rrbracket \in \mathcal{R}_\infty$ and thus $e_{\min} \leq e \leq e_{\max} \vee e \in \{+\infty, -\infty\}$. Furthermore $\lfloor f \cdot 2^{p-1} \rfloor = f \cdot 2^{p-1}$ holds. Then the packing function has the succeeding definition.

$$\text{pack}((s, e, f), (n, p)) = \begin{cases} s \circ 1^n \circ 10^{63} & : (n, p) = (15, 64) \wedge e = \infty \\ s \circ \text{bin}_{15}(e + 2^{14} - 1) \circ \text{bin}_1(\lfloor f \rfloor) \circ \text{bin}_{63}((f - \lfloor f \rfloor) \cdot 2^{p-1}) & : (n, p) = (15, 64) \wedge e \neq \infty \\ s \circ \text{bin}_n(e + 2^{n-1} - 1) \circ \text{bin}_{p-1}((f - 1) \cdot 2^{p-1}) & : (n, p) \neq (15, 64) \wedge e \notin \{\infty, e_{\min}\} \\ s \circ 0^n \circ \text{bin}_{p-1}(f) & : (n, p) \neq (15, 64) \wedge e = e_{\min} \\ s \circ 1^n \circ 0^{p-1} & : (n, p) \neq (15, 64) \wedge e = \infty \end{cases}$$

We must take the special cases of infinity into account. For the double-extended precision mode we have to explicitly save the integer significand bit. The exponents are converted back to biased

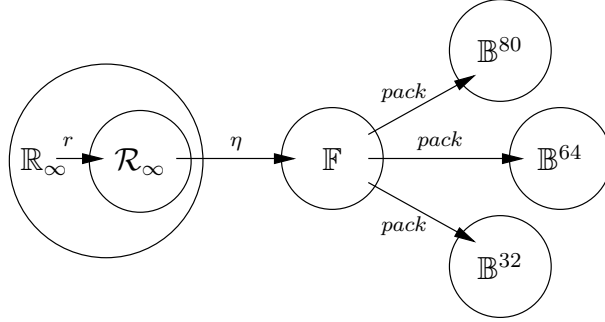


Figure 5.2: Transferring real numbers into floating-point representation

binary format via addition of the bias and transformation to binary representation. For the fractional parts we also utilize the bin_n function as defined in chapter 2 after scaling the fractions to integers. The pack function for the common 32-bit, 64-bit or 80-bit floating-point bit strings just forwards the result from the pack function above.

$$pack(x, size) = \begin{cases} pack(x, (8, 24)) & : size = 32 \\ pack(x, (11, 53)) & : size = 64 \\ pack(x, (15, 64)) & : size = 80 \end{cases}$$

Converting a real number x into floating-point representation can now be done by successive rounding, factoring and packing.

$$x \mapsto pack(\eta(r_f(x)), 80)$$

This particular mapping for instance rounds the number according to the current FPU configuration f and packs it into 80-bit double-extended precision floating-point format. Anyway the rounding and precision modes can be easily altered. Figure 5.2 depicts the overall conversion process.

Besides transforming factorings into bit strings we also may need to convert bit strings from a low precision to a higher one. This is necessary when loading floating-point memory operands of sizes smaller than 80 bits into the FPU. We denote the function which executes this bit string extension by *extend*.

$$extend : (\mathbb{B}^m, FPU) \rightarrow \mathbb{B}^{80}$$

Again we want to define the function for arbitrary bit partitions and precisions. For a bit string $x \in \mathbb{B}^m$ of length $m = n + p$, with $n \leq 15$, $p \leq 64$ but $(n, p) \neq (15, 64)$ we yield:

$$extend(x[m-1:0], f) = \begin{cases} s(x) \circ sext_{15}(e(x)) \circ 1 \circ f(x)[p-1:0] \circ 0^{64-p} & : NaN(x) \vee infty(x) \\ pack(\eta(\llbracket x \rrbracket), 80) & : \text{otherwise} \end{cases}$$

For special values we sign and zero extend the respective number components. Ordinary numbers are reconverted to the higher precision from their real value using the factoring and packing routines. When $(n, p) = (15, 64)$ the extension is trivial and

$$extend(x, f) = x$$

holds. With the completion of the bit string extension we have gathered all necessary tools to handle x87 floating-point numbers.

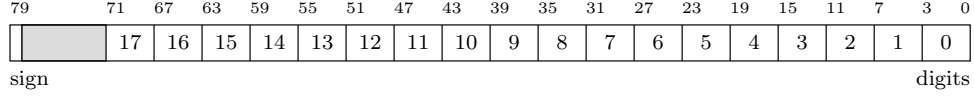


Figure 5.3: Structure of 80-bit BCD numbers

5.4 Binary-Coded Decimal Numbers

In addition to integers and floating-point numbers the x87 instruction subset supports an 80-bit packed binary-coded decimal format. However the FPU does not calculate with these numbers. As with integers it just supplies us with instructions to load/store numbers from/to BCD format⁶. We need a method to interpret 80-bit BCD bit strings as numbers. For the purpose of our model it would be enough to care only about 80-bit BCD numbers. Nevertheless we will state definitions for arbitrary bit lengths. The structure of 80-bit BCD numbers is shown in figure 5.3. Every four-bit nibble holds the binary representation of a decimal digit. As the smallest data unit in the IA-32 instruction set is a byte, we assume that each byte of the BCD number encodes two decimals. Overlaying bits which do not make up an additional byte do not contain any digits, even when a nibble would fit in. In addition there is a sign bit located at the most significant bit position. It must not be part of a digit byte. Take into account that from all this it follows that there might be unused bits between the highest decimal nibble and the sign bit. With respect to these facts we denote the value of a BCD number n -bit string $x[n-1:0]$ by $\langle x \rangle_{BCD}$.

$$\begin{aligned} \langle \cdot \rangle_{BCD} &: \mathbb{B}^n \rightarrow \mathbb{Z} \\ \langle x[n-1:0] \rangle_{BCD} &= (-1)^{x[n-1]} \cdot \sum_{i=0}^{\lfloor \frac{n-1}{8} \rfloor \cdot 2 - 1} \langle x[4i+3:4i] \rangle \cdot 10^i \end{aligned}$$

Observe that $\lfloor \frac{n-1}{8} \rfloor$ represents the number of bytes fitting into the bit string. The number of encoded decimals is given by $\lfloor \frac{n-1}{8} \rfloor \cdot 2$. Not surprisingly the value of the BCD number is computed by the sum of decimals weighed with their corresponding powers of ten.

5.5 x87 Exceptions

During floating-point calculations chances are that some exceptional situations will occur. That is for instance operations which are mathematically undefined, like division by zero etc. There are also unsupported data types and due to the variety of possible operand types some combinations may be problematic. At last it may also happen that the stack is not accessed properly. Such circumstances may force the FPU to stop computation and report an error to the CPU. At least they demand a special treatment and cannot be dealt with in a standard way. To handle these situations the x87 instruction set introduces seven⁷ floating-point exception types.

$$FPU\text{EXCP} : \text{OVF}|\text{UNF}|\text{DBZ}|\text{INX}|\text{DEN}|\text{INV}|\text{STK}$$

⁶FBLD loads an 80-bit BCD number, converts it to double-extended floating point format and pushes it on the FPU stack. FBSTP pops a floating-point number from the stack, rounds it to an integer and stores it in BCD format to a memory destination.

⁷Actually there is only six types, while stack fault is a special case of invalid operation exception.

Pre-Computation	Post-Computation
STK	OVF
INV	UNF
DBZ	INX
DEN	

Table 5.1: Pre- and Post-Computation Floating Point Exceptions

These are in detail:

- OVF - Overflow exception
- UNF - Underflow exception
- DBZ - Division-by-Zero exception
- INX - Inexact result exception (precision exception)
- DEN - Denormal operand exception
- INV - Invalid operation exception
- STK - Floating-point stack fault

They describe the occurrence of problematic cases and signal these to the CPU. However the exceptions may be concealed from the CPU by masking the corresponding exception flags. In these cases exceptions are handled in a predefined default manner and special results are returned. Floating-point exceptions are divided into two types, namely pre- and post-computation as shown in table 5.5. They are ordered according to their priority from most important down to least important. We will define the causes and effects of these exceptions later on.

Chapter 6

Operations

At this point we have all the information available to proceed to the execution routine to compute the result for an FPU instruction. What we have not discussed yet, is the computation of the result itself. We need to investigate the instruction set as well as the mathematical aspects of floating-point operations. Also operation-specific floating-point exceptions are taken into consideration.

6.1 x87 Instruction Groups

There are more than 80 floating point instructions which compute many different kinds of functions - mathematical and non-mathematical ones. However it would be inexpedient to make case distinctions with 80 cases. Thus we need to further examine the instruction set and divide instructions into functional groups. There are also variant instructions of the same function which differ in stack manipulation and operand scheme. We will define predicates which allow us to distinguish between instructions and between several instruction groups.

In general we determine the type of an FPU operation by analysing the corresponding opcode. We define predicates to report the group membership of an instruction. The predicates will be labelled with a certain mnemonic associated with the respective instruction. For more convenience we overload these mnemonic(X)-predicates to be applied within several contexts and to several arguments X.

$$\begin{array}{llll} \text{mnemonic} : & \textit{Instruction} & \rightarrow & \mathbb{B} \\ \text{mnemonic} : & \textit{FPU} & \rightarrow & \mathbb{B} \\ \text{mnemonic} : & \mathbb{B}^{16} & \rightarrow & \mathbb{B} \\ \text{mnemonic} : & \mathbb{B}^{11} & \rightarrow & \mathbb{B} \end{array}$$

If not defined otherwise these are always computed on the following way.

$$\begin{array}{ll} \text{mnemonic}(I) & = \text{mnemonic}(I.opc[7 : 0] \circ I.modrm) \\ \text{mnemonic}(f) & = \text{mnemonic}(opc(f)) \\ \text{mnemonic}(opc[15 : 0]) & = (opc[15 : 11] = 11011) \wedge \text{mnemonic}(opc[10 : 0]) \end{array}$$

Following this scheme mostly just the predicate $\text{mnemonic}(opc[10 : 0])$ for 11-bit opcode fields must be defined individually. E.g. the predicates for the *FNOP* instruction are defined according to the rules given above by exchanging mnemonic with *FNOP*.

$$\begin{aligned} \text{FNOP}(I) &= \text{FNOP}(I.opc[7 : 0] \circ I.modrm) \\ \text{FNOP}(f) &= \text{FNOP}(opc(f)) \\ \text{FNOP}(opc[15 : 0]) &= (opc[15 : 11] = 11011) \wedge \text{FNOP}(opc[10 : 0]) \end{aligned}$$

All $\text{mnemonic}(opc[10 : 0])$ instruction predicates are defined in Appendix B. From now on by *I* we denote *Instruction*-type arguments and *f* represents an *FPU* configuration. $opc[.]$ stands for an opcode string of either 16 or 11-bit length. A single *opc* shall be interpreted as an 11-bit x87 opcode string.

6.1.1 Grouping by Functions

We can classify x87 instructions into nine general groups. Predicate $x87(X)$ signals that the instruction or opcode under consideration belongs to the x87 instruction subset. Here we have to adopt individual definitions to incorporate the special cases of FXSAVE/FXRSTOR and WAIT instructions which deviate from the standard x87 opcode format.

$$\begin{aligned} x87(I) &= \begin{cases} x87(I.opc[7 : 0] \circ I.modrm) & : I.opc[23 : 3] = 0^{16}11011 \\ 1 & : I.opc[15 : 0] = 0x0FAE \\ & \wedge \langle I.modrm[5 : 3] \rangle \in \{0, 1\} \\ & \vee I.opc[15 : 0] = 0x009B \\ 0 & : \text{otherwise} \end{cases} \\ x87(f) &= \begin{cases} x87(opc(f)[7 : 0] \circ I(f).modrm) & : opc(f)[23 : 3] = 0^{16}11011 \\ 1 & : opc(f)[15 : 0] = 0x0FAE \\ & \wedge \langle I(f).modrm[5 : 3] \rangle \in \{0, 1\} \\ & \vee opc(f)[15 : 0] = 0x009B \\ 0 & : \text{otherwise} \end{cases} \\ x87(opc[15 : 0]) &= (opc[15 : 11] = 11011) \wedge x87(opc[10 : 0]) \\ x87(opc[10 : 0]) &= data(opc) \vee arith(opc) \vee sign(opc) \vee const(opc) \vee cmp(opc) \\ &\quad \vee trans(opc) \vee stack(opc) \vee fnop(opc) \vee ctrl(opc) \end{aligned}$$

Accordingly we need to define the predicates for the nine function classes.

- *data*(X) - Data Transfer
- *arith*(X) - Arithmetic Operations
- *sign*(X) - Sign Changing Operations
- *const*(X) - Load Constants

- *cmp*(X) - Compare and Test
- *trans*(X) - Transcendental Functions
- *stack*(X) - Stack Management
- *fnop*(X) - No Operation
- *ctrl*(X) - Control Operations

Many of these groups even consist of function subclasses. Those subclasses comprise instruction predicates at last. See table B.1 for their exact definitions. Note that one instruction may be referred to by more than one opcode. The different opcode variants are discussed later in this section. For FXSAVE and FXRSTOR which are not pure x87 instructions we define the following predicates for an instruction *I*.

$$\begin{aligned} FXSAVE(I) = 1 &\Leftrightarrow (I.opc[15 : 0] = 0x0FAE) \wedge \langle I.modrm[5 : 3] \rangle = 0 \\ FXRSTOR(I) = 1 &\Leftrightarrow (I.opc[15 : 0] = 0x0FAE) \wedge \langle I.modrm[5 : 3] \rangle = 1 \end{aligned}$$

The IA32 instruction set also specifies waiting control instructions *FINIT*, *FCLEX*, *FSTCW*, *FSTSW*, *FSTENV* and *FSAVE*. These *Fx* mnemonics do not refer to self-contained instructions. They are rather compiler macros which are translated to a pair of instructions. The first one is an *FWAIT*/*WAIT* to handle any pending x87 exceptions. It is followed by the counterpart *FNx* non-waiting control instruction. We detect a wait instruction by using the following predicate.

$$WAIT(I) = (I.opc[24 : 16] = 0x9B)$$

Now we can define the function classes as shown in figure 6.1. We begin with *data*(X).

$$\begin{aligned} data(opc) &= load(opc) \vee store(opc) \vee FCMOV(opc) \vee FXCH(opc) \vee FXTRACT(opc) \\ load(opc) &= FLD(opc) \vee FILD(opc) \vee FBLD(opc) \\ store(opc) &= FST(opc) \vee FSTP(opc) \vee FBSTP(opc) \vee \\ &\quad FIST(opc) \vee FISTP(opc) \vee FISTTP(opc) \end{aligned}$$

Arithmetic Operations are divided into several subclasses.

$$\begin{aligned} arith(opc) &= add(opc) \vee sub(opc) \vee mul(opc) \vee div(opc) \vee \\ &\quad prem(opc) \vee FRNDINT(opc) \vee FSQRT(opc) \\ add(opc) &= FADD(opc) \vee FADDP(opc) \vee FIADD(opc) \\ sub(opc) &= FSUB(opc) \vee FSUBP(opc) \vee FISUB(opc) \vee \\ &\quad FSUBR(opc) \vee FSUBPR(opc) \vee FISUBR(opc) \\ mul(opc) &= FMUL(opc) \vee FMULP(opc) \vee FIMUL(opc) \\ div(opc) &= FDIV(opc) \vee FDIVP(opc) \vee FIDIV(opc) \vee \\ &\quad FDIVR(opc) \vee FDIVPR(opc) \vee FIDIVR(opc) \\ prem(opc) &= FPREM(opc) \vee FPREM1(opc) \end{aligned}$$

Sign Changing Operations manipulate the sign bit of floating-point numbers.

$$sign(opc) = FABS(opc) \vee FCHS(opc)$$

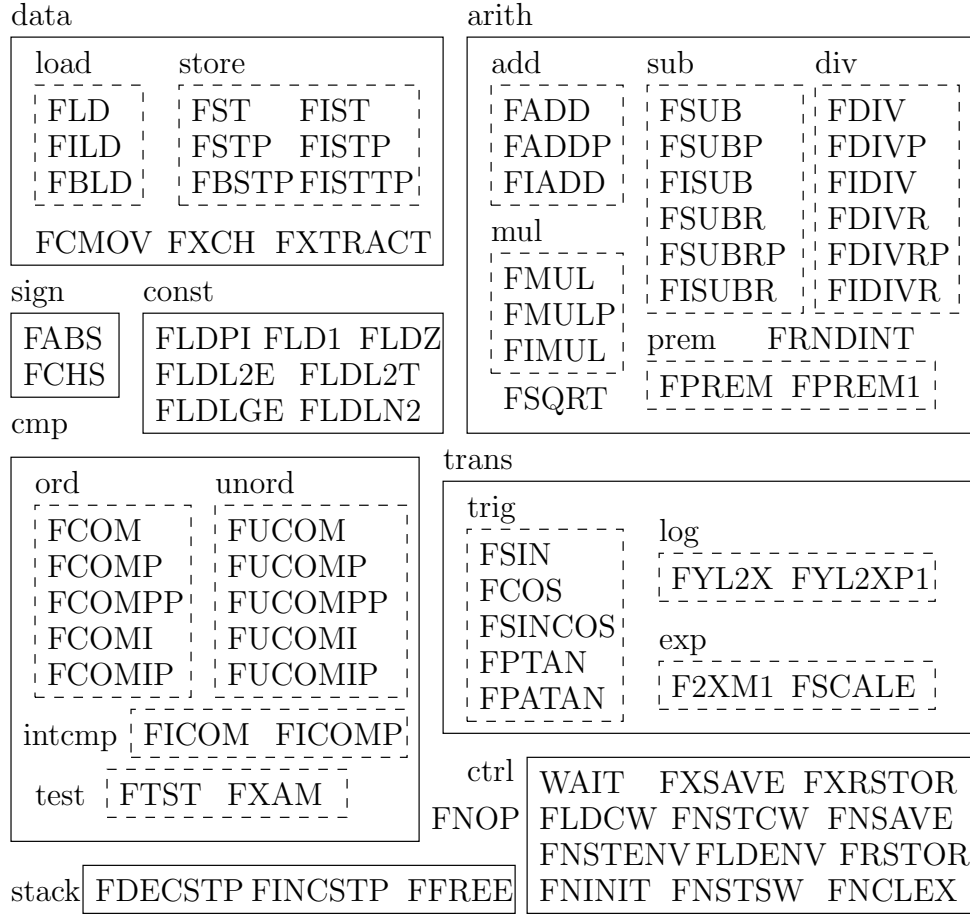


Figure 6.1: Structured x87 instruction set

Load Constants instructions push either 0.0, +1.0, π or logarithmic constants on the stack.

$$\begin{aligned}
const(opc) = & FLDZ(opc) \vee FLD1(opc) \vee FLDPI(opc) \vee \\
& FLDL2E(opc) \vee FLDL2T(opc) \vee FLDLG2(opc) \vee FLDLN2(opc)
\end{aligned}$$

There are ordered and unordered Compare instructions as well as integer compare and x87 test operations.

$$\begin{aligned}
cmp(opc) &= ord(opc) \vee unord(opc) \vee intcmp(opc) \vee test(opc) \\
ord(opc) &= FCOM(opc) \vee FCOMP(opc) \vee FCOMPP(opc) \vee \\
&\quad FCOMI(opc) \vee FCOMIP(opc) \\
unord(opc) &= FUCOM(opc) \vee FUCOMP(opc) \vee FUCOMPP(opc) \vee \\
&\quad FUCOMI(opc) \vee FUCOMIP(opc) \\
intcmp(opc) &= FICOM(opc) \vee FICOMP(opc) \\
test(opc) &= FTST(opc) \vee FXAM(opc)
\end{aligned}$$

Transcendental functions are either trigonometric, logarithmic or exponential.

$$\begin{aligned}
trans(opc) &= trig(opc) \vee log(opc) \vee exp(opc) \\
trig(opc) &= FSIN(opc) \vee FCOS(opc) \vee FSINCOS(opc) \vee \\
&\quad FPTAN(opc) \vee FPATAN(opc) \\
log(opc) &= FYL2X(opc) \vee FYL2XP1(opc) \\
exp(opc) &= F2XM1(opc) \vee FSCALE(opc)
\end{aligned}$$

Only three instructions may directly influence the stack management.

$$stack(opc) = FDECSTP(opc) \vee FINCSTP(opc) \vee FFREE(opc)$$

At last we set $fnop(opc) = FNOP(opc)$ and define the control instruction class.

$$\begin{aligned}
ctrl(I) &= WAIT(I) \vee FXSAVE(I) \vee FXRSTOR(I) \vee ctrl(I.opc[7:0] \circ I.modrm) \\
ctrl(f) &= FXSAVE(I(f)) \vee FXRSTOR(I(f)) \vee ctrl(opc(f)[7:0] \circ I(f).modrm) \\
ctrl(opc[15:0]) &= (opc[15:11] = 11011) \wedge ctrl(opc[10:0]) \\
ctrl(opc[10:0]) &= FLDENV(opc) \vee FNSTENV(opc) \vee FNINIT(opc) \vee \\
&\quad FNSTSW(opc) \vee FNCLEX(opc) \vee FNSTCW(opc) \vee \\
&\quad FLDCW(opc) \vee FNSAVE(opc) \vee FRSTOR(opc)
\end{aligned}$$

Finally we can give the definition for non-control instructions. The corresponding predicate is needed for the interface when reporting unmasked floating-point exceptions to the CPU.

$$nonctrl(I) = \overline{ctrl(I)}$$

Furthermore we consider all instructions which calculate floating-point type results with the *calc* predicate.

$$calc(opc) = arith(opc) \vee trans(opc) \vee FXTRACT(opc)$$

6.1.2 Grouping by Operand and Result Scheme

Now we are able to distinguish between all kinds of FPU instructions. However even a single instruction may have different variants concerning number and type of source and destination operands. We want to shed light on this circumstance below. Source operands are argument inputs to the various x87 instructions. Mostly they are floating-point or integer numbers, but there may also be image inputs. They may reside in memory or in some x87 stack register. Floating-point instructions can have one, two or no operands. The following schemes can be distinguished for the several instructions.

- One operand is always $st(f, 0)$ and the other one is either empty, 0.0 or $st(f, 1)$.
- One operand is chosen from the stack and the other one is either empty or $st(f, 0)$.
- One operand is read from memory. The other operand may be empty or $st(f, 0)$.
- The instruction does not need any source operands.

Concerning results we can distinguish four cases.

- A result from a calculation, sign changing, constant or data transfer operation is saved at or pushed to the x87 stack.
- A result from a data transfer or control operation is stored in memory.
- The instruction does not produce any result data. Components of the FPU environment may be updated.
- The *RFLAGS* bits are updated after certain comparison instructions.

Observe that one instruction can have zero, one or two results. However only instructions with one result may save it to a memory destination. Note also that especially the result schemes are partly linked to the different stack manipulation schemes. We will focus on these matters in the next subsection.

6.1.3 Operand and Result Predicates

For now we care merely about where source operands come from, their number and size as well as where the results are stored in the end. Therefore we begin with defining three general predicates on the basis of the opcode table in Appendix A.

$$\begin{aligned}
xread(opc) = 1 & \Leftrightarrow x87(I) \wedge (r?(opc[15 : 8], opc[7 : 0]) = 1) \\
xread(I) & = xread(I.opc \circ I.modrm) \\
xupdf(opc) = 1 & \Leftrightarrow x87(I) \wedge (uf?(opc[15 : 8], opc[7 : 0]) = 1) \\
xupdf(I) & = xupdf(I.opc \circ I.modrm) \\
xupdm(opc) = 1 & \Leftrightarrow x87(I) \wedge (um?(opc[15 : 8], opc[7 : 0]) = 1) \\
xupdm(I) & = xupdm(I.opc \circ I.modrm)
\end{aligned}$$

xread identifies functions which read an operand from memory. These are already marked with a 1 in the *r?* column of the opcode table. Accordingly *xupdf* recognizes instructions that update the *RFLAGS* register and *xupdm* perceives instructions storing a result to memory¹. The x87 results can have various sizes. We denote the size of the result of the current instruction regarding configuration *f* by *resize(f)*.

$$\begin{aligned}
resize : FPU & \rightarrow \{16, 32, 64, 80, 112, 224, 752, 864, 4096\} \\
resize(f) & = \begin{cases} opsize(I(f).opc, I(f).modrm, x64_mode(m(f)), 1, 32) & : xupdm(I(f)) \\ 80 & : \text{otherwise} \end{cases}
\end{aligned}$$

Once again the opcode table is utilized for the definition. We can also refer to it to determine the number of operands pertaining to a specific instruction. However this definition is partly inexact as the table mixes source and destination operands. That is why instructions without source operands

¹This also incorporates the case when *FNSTSW* stores the x87 status word to the *AX* register. Although *AX* is not a part of memory we can embed this special case here, because the *writeOp* function in the interface can be used for all kinds of non-FPU destination operands.

may be assigned to the one-operand group. Nevertheless in all cases of instructions featuring source operands, we can trust in $nops(f)$.

$$\begin{aligned} nops : \mathbb{B}^{16} &\rightarrow \mathbb{N}_3 \\ nops(opc[15:0]) &= \begin{cases} 2 & : \quad op1(opc[15:8], opc[7:0]) \neq \epsilon \wedge op2(opc[15:8], opc[7:0]) \neq \epsilon \\ 1 & : \quad op1(opc[15:8], opc[7:0]) \neq \epsilon \wedge op2(opc[15:8], opc[7:0]) = \epsilon \\ 0 & : \quad \text{otherwise} \end{cases} \end{aligned}$$

In addition we want to cover the case of instructions having two floating-point results.

$$\begin{aligned} 2results : FPU &\rightarrow \mathbb{B} \\ 2results(f) &= FXTRACT(f) \vee FPTAN(f) \vee FSINCOS(f) \end{aligned}$$

Furthermore we define predicates signalling when images have to be loaded or saved.

$$\begin{aligned} imgsave(f) &= FNSTENV(f) \vee FNSAVE(f) \vee FXSAVE(f) \\ imgload(f) &= FLDENV(f) \vee FRSTOR(f) \end{aligned}$$

Actually also for $FXRSTOR$ an image must be restored but this instruction differs in many details from the $imgload$ functions, so we exclude it from this predicate to the benefit of a more convenient notation later on.

6.1.4 Sources and Destinations

In this subsection we accomplish some essential definitions, namely those of the two floating-point operands and the target stack register indices of instructions as well. We begin with the operands, where we focus on operands containing numbers. That means we exclude images as control instruction operands for the sake of simplification. Image saving and restoring will be handled later.

$$\begin{aligned} op1 &: FPU \rightarrow \mathbb{B}^{80}|_{\epsilon} \\ op2 &: FPU \rightarrow \mathbb{B}^{80}|_{\epsilon} \end{aligned}$$

All numbers are translated into double-extended precision floating-point numbers for usage in the FPU. Thus it is convenient to have not only a reference to the source operand but also already the format conversion included into the operand functions.

$$op1(f) = \begin{cases} pack(\eta(r_f([mop(f)])), 80) & : \quad intop1(f) \\ extend(mop(f), f) & : \quad fpmop1(f) \\ pack(\eta(r_f(\langle mop(f) \rangle_{BCD})), 80) & : \quad FBLD(f) \\ st(f, \langle opc(f)[2:0] \rangle) & : \quad srop1(f) \\ \epsilon & : \quad ctrl(f) \vee fnop(f) \vee stack(f) \\ st(f, 0) & : \quad \text{otherwise} \end{cases}$$

In case of certain integer loading arithmetic instructions and $FILD$ the first operand is the memory integer operand's value rounded and transformed into a floating-point number.

$$intop1(f) = FISUBR(f) \vee FIDIVR(f) \vee FILD(f)$$

When a floating-point number is loaded from memory as the first operand it may have to be extended to a higher precision.

$$fpmop1(f) = xread(I(f)) \wedge (FSUBR(f) \vee FDIVR(f) \vee FLD(f))$$

For FBLD the memory operand in BCD representation is translated into IEEE 754 format. There are also instructions which allow the programmer to choose the source x87 stack register for the first operand by the last three bits of the x87 opcode.

$$srop1(f) = \overline{xread(I(f))} \wedge (opc(f)[15:8] = 0xD8 \wedge (FSUBR(f) \vee FDIVR(f)) \vee FLD(f)) \vee FSUBP(f) \vee FDIVP(f)$$

However mostly the stack-top register contains the first operand. For control, NOP and stack manipulation instructions no floating-point operands are needed. Concerning the second floating-point operand we distinguish seven cases.

$$op2(f) = \begin{cases} pack(\eta(r_f([mop(f)])), 80) & : intop2(f) \\ extend(mop(f), f) & : fpmop2(f) \\ st(f, \langle opc(f)[2:0] \rangle) & : srop2(f) \\ st(f, 1) & : st1op2(f) \\ st(f, 0) & : st0op2(f) \\ 0^{80} & : FTST(f) \\ \epsilon & : otherwise \end{cases}$$

Certain instructions demand an integer memory operand as second operand. They are identified by $intop2(f)$ predicate.

$$intop2(f) = FISUB(f) \vee FIDIV(f) \vee FIADD(f) \vee FIMUL(f) \vee intcmp(f)$$

Also floating-point numbers in memory may represent the second operand. Using the *extend* function they are brought into the right precision format.

$$fpmop2(f) = xread(I(f)) \wedge (FCOM(f) \vee FCOMP(f) \vee FADD(f) \vee FSUB(f) \vee FMUL(f) \vee FDIV(f))$$

Some variants of the arithmetic operations allow to choose the second operand from the x87 register stack via the last three x87 opcode bits. Predicate $srop2(f)$ specifies which opcodes cause this behaviour.

$$\begin{aligned} srop2(f) = & \overline{xread(I(f))} \wedge (opc(f)[15:8] = 0xD8 \\ & \wedge (FADD(f) \vee FSUB(f) \vee FMUL(f) \vee FDIV(f)) \\ & \vee (cmp(f) \wedge \overline{pop2(f)}) \vee (opc(f)[15:8] = 0xDC \\ & \wedge (FADD(f) \vee FSUBR(f) \vee FMUL(f) \vee FDIVR(f))) \\ & \vee FADDP(f) \vee FSUBRP(f) \vee FMULP(f) \vee FDIVRP(f)) \end{aligned}$$

Further on there are sets of instructions which always use $st(f, 0)$ or $st(f, 1)$ as the second operand. We introduce $st0op2$ and $st1op2$ to detect them.

$$\begin{aligned} st0op2(f) = & FSUBP(f) \vee FISUBR(f) \vee FDIVP(f) \vee FIDIVR(f) \\ & \vee ((opc(f)[15:8] = 0xD8 \wedge \overline{xread(I(f))}) \wedge (FSUBR(f) \vee FDIVR(f))) \\ & \vee (opc(f)[15:8] = 0xDC \wedge \overline{xread(I(f))}) \wedge (FSUB(f) \vee FDIV(f)) \\ st1op2(f) = & FYL2X(f) \vee FYL2XP1(f) \vee FPATAN(f) \vee FSCALE(f) \\ & \vee FUCOMPP(f) \vee FCOMPP(f) \vee prem(f) \end{aligned}$$

At last *FTST* compares a number with 0.0. All other instructions do not possess a second operand.

Now we want to investigate the destination operands. That is the target components which are updated with the respective result of an x87 instruction. Considering the aforementioned result schemes, x87 stack registers are the only destinations which reside inside the FPU. The others are either the *RFLAGS* register bits or a memory destination. We already established means to distinguish between the instructions which update memory operands or flags by the predicates $xupdf(I(f))$ and $xupdm(I(f))$. They are defined implicitly based on a table search. With our new knowledge we can replace them by explicitly defined predicates $xupdf(f)$ and $xupdm(f)$.

$$\begin{aligned} xupdf : FPU &\rightarrow \mathbb{B} \\ xupdf(f) &= FCOMI(f) \vee FCOMIP(f) \vee FUCOMI(f) \vee FUCOMIP(f) \\ xupdm : FPU &\rightarrow \mathbb{B} \\ xupdm(f) &= store(f) \vee imgsave(f) \vee FNSTCW(f) \vee FNSTSW(f) \end{aligned}$$

Now we want to determine which stack elements are affected by a floating-point operation. Therefore we define the targets of the current configuration's instruction $trgt(f)$.

$$\begin{aligned} trgt : FPU &\rightarrow (\mathbb{N}_8 | \epsilon, \mathbb{N}_8 | \epsilon) \\ trgt(f) &= (trgt_1(f), trgt_2(f)) \end{aligned}$$

Obviously these targets may be empty, when an instruction does not save or push a value to the stack. The targets define the stack element indices which identify the registers the results shall be stored to. We begin with the target for the first result.

$$trgt_1(f) = \begin{cases} \epsilon & : \overline{push?(f)} \wedge \overline{save?(f)} \\ 7 & : \overline{push?(f)} \wedge \overline{save?(f)} \\ 1 & : FPATAN(f) \vee FYL2X(f) \vee FYL2XP1(f) \\ \langle opc(f)[2 : 0] \rangle & : opc(f)[15 : 8] = 0xDC \wedge \overline{xread(I(f))} \wedge arith(f) \\ & \vee arith(f) \wedge pop?(f) \\ & \vee \overline{xread(I(f))} \wedge (FST(f) \vee FSTP(f)) \\ 0 & : \text{otherwise} \end{cases}$$

In most cases this is the stack-top register. For a separate push we aim at the stack register above the stack-top, which is stack register 7 following the modulo-8 computation. Several instructions may also choose the target via opcode bits or they always store the result to stack element 1. A second target only exists when there is a push and a save or for the *FXCH* instruction.

$$trgt_2(f) = \begin{cases} 7 & : \overline{push?(f)} \wedge \overline{save?(f)} \\ \langle opc(f)[2 : 0] \rangle & : FXCH(f) \\ \epsilon & : \text{otherwise} \end{cases}$$

At a push and a save the second result is pushed to stack register 7. *FXCH* allows to choose the target using the last three x87 opcode bits. Note that the predicates $push?(f)$, $save?(f)$ and $pop?(f)$ are defined in the following subsection.

6.1.5 Grouping by Stack Manipulation Behaviour

Besides the function that instructions implement, the type and number of operands they utilize or where they store their results, we can also group instructions regarding the way they manipulate the floating-point register stack. Instructions may save data to the stack, that is they just overwrite the content of an existing stack element with a new value without changing the stack structure. In addition data may be pushed on top of the stack. This implies updating the top-of-stack pointer as a side effect. After saving the results instructions have also the possibility to pop the stack-top element off the stack. Thereby the stack-top pointer will be adjusted and the respective register will be tagged empty. Some instructions even pop two elements off the stack at once. Further information about the execution of these stack operations can be found in sections 7.2.2 and 7.4.2.

Most x87 instructions including all arithmetic and sign changing operations save a value to the stack. we can identify these instructions by the $save? : FPU \rightarrow \mathbb{B}$ predicate.

$$save?(f) = sign(f) \vee calc(f) \vee ((FST(f) \vee FSTP(f)) \wedge \overline{xupdm(f)})$$

All load and constant instructions as well as arithmetic instructions which produce two results may push results onto the x87 stack. In case they are the current instructions of an FPU configuration f the $push? : FPU \rightarrow \mathbb{B}$ predicate becomes true.

$$push?(f) = load(f) \vee const(f) \vee 2results(f)$$

A variety of instructions may pop the FPU stack. The predicate $pop? : FPU \rightarrow \mathbb{B}$ comprises all operations which pop one element off the stack.

$$\begin{aligned} pop?(f) = & FSTP(f) \vee FISTP(f) \vee FISTTP(f) \vee FBSTP(f) \vee FCOMP(f) \vee \\ & FCOMIP(f) \vee FUCOMP(f) \vee FUCOMIP(f) \vee FICOMP(f) \vee FADDP(f) \vee \\ & FMULP(f) \vee FSUBP(f) \vee FSUBRP(f) \vee FDIVP \vee FDIVRP(f) \vee \\ & FPATAN(f) \vee FYL2X(f) \vee FYL2XP1(f) \end{aligned}$$

There are only two instructions which pop the stack twice. They are detected by $pop2? : FPU \rightarrow \mathbb{B}$.

$$pop2?(f) = FCOMPP(f) \vee FUCOMPP(f)$$

6.2 x87 Format Converting Functions

After having structurized the instruction set we want to have a look at actual computation issues. Computation is done on double-extended precision floating-point numbers and inside the FPU only this kind of numbers is stored. However there are certain instructions such as FST, FIST, FBSTP etc., which may convert floating-point data to different number formats or precisions. Some instructions also need to round floating-point numbers to integers, thus before dealing with the mathematical functions of the FPU we have to establish means to convert numbers to different formats.

6.2.1 Converting Precision

Some variants of FSTP and all FST instructions store floating-point data to 32- or 64-bit memory destinations. For this purpose their formats must be converted to single or double precision. To this end we define a precision conversion function prc which returns the first operand converted to the suitable precision in case the predicate $cvtprc(f)$ holds.

$$\begin{aligned}
cvtprc : FPU &\rightarrow \mathbb{B} \\
cvtprc(f) &= FST(f) \vee FSTP(f) \\
prc : (\mathbb{B}^{80}, \mathbb{B}^2, \{32, 64, 80\}) &\rightarrow \mathbb{B}^* \\
prc(x, rm, size) &= \begin{cases} pack(\eta(r(\lfloor x \rfloor), rm, 8, 24)), 32 & : \quad size = 32 \wedge \overline{NaN}(x) \\ x[79 : 71] \circ 1 \circ x[62 : 41] & : \quad size = 32 \wedge NaN(x) \\ pack(\eta(r(\lfloor x \rfloor), rm, 11, 53)), 64 & : \quad size = 64 \wedge \overline{NaN}(x) \\ x[79 : 68] \circ 1 \circ x[62 : 12] & : \quad size = 64 \wedge NaN(x) \\ x & : \quad \text{otherwise} \end{cases}
\end{aligned}$$

The particular numbers are rounded according to the current rounding mode. In case a number's magnitude exceeds the limits of representable numbers it is rounded to infinity or the biggest representable number respectively.

6.2.2 Converting to Binary Integer Format

For integer storing instructions FIST, FISTP and FISTTP but also several other operations we must be able to convert floating-point numbers to integers. The Predicate $cvtint(f)$ signals whether a number shall be stored in integer format.

$$\begin{aligned}
cvtint : FPU &\rightarrow \mathbb{B} \\
cvtint(f) &= FIST(f) \vee FISTP(f) \vee FISTTP(f)
\end{aligned}$$

This conversion of a floating-point number naturally forces us to apply rounding. The same rounding modes as for floating-point rounding are available, however we can not utilize our floating-point rounding functions in this case. For rounding x to the nearest even integer, we introduce the function $rne(x)$.

$$\begin{aligned}
rne : \mathbb{R} &\rightarrow \mathbb{Z} \\
rne(x) &= \begin{cases} \text{sgn}(x) \cdot \lfloor |x| \rfloor & : \quad \lceil x \rceil - x < 0.5 \vee \lceil x \rceil - x = 0.5 \wedge \exists n \in \mathbb{Z} : \lceil x \rceil = 2 \cdot n \\ \text{sgn}(x) \cdot \lfloor |x| \rfloor & : \quad \text{otherwise} \end{cases}
\end{aligned}$$

Then depending on a given rounding mode rm the rounded integer number regarding a real number x is denoted by $rndint(x, rm)$.

$$\begin{aligned}
rndint : (\mathbb{R}, \mathbb{B}^2) &\rightarrow \mathbb{Z} \\
rndint(x, rm) &= \begin{cases} rne(x) & : \quad rm = 00 & \text{(round to nearest even)} \\ \text{sgn}(x) \cdot \lfloor |x| \rfloor & : \quad rm = 01 & \text{(round down)} \\ \text{sgn}(x) \cdot \lceil |x| \rceil & : \quad rm = 10 & \text{(round up)} \\ \text{sgn}(x) \cdot \lfloor |x| \rfloor & : \quad rm = 11 & \text{(round to zero)} \end{cases}
\end{aligned}$$

We do not need to care about infinities as these result in invalid operation exceptions for the functions under consideration. The factor $\text{sgn}(x)$ ensures that the number's sign is kept, when the number is rounded to zero. Thus from the sign of zero one can determine from which direction the number was rounded to zero. Keep in mind that we keep the sign of zero explicit also for integers.

6.2.3 Converting to BCD Format

We already introduced the BCD number format and postulated a method to compute the value of a number's binary-coded decimal representation in section 5.4. For the FBSTP instruction we also must be able to convert an integer a to BCD format. Therefore we define the following conversion function with $n \in \mathbb{N}, n > 8$ and $\lfloor \frac{n-1}{8} \rfloor \cdot 2 > \log_{10} a$.

$$\begin{aligned} bcd_n : \mathbb{Z} &\rightarrow \mathbb{B}^n \\ bcd_n(a) &= x[n-1 : 0], \quad \text{with} \quad \langle x \rangle_{BCD} = a \end{aligned}$$

So we implicitly defined the BCD representation of an integer. We can implement this function recursively as follows.

$$bcd_n(a) = \begin{cases} 1 \circ 0^{n-1-\lfloor \frac{n-1}{8} \rfloor \cdot 8} \circ bcd'(|a|, \lfloor \frac{n-1}{8} \rfloor \cdot 2 - 1) & : a \leq -0 \\ 0 \circ 0^{n-1-\lfloor \frac{n-1}{8} \rfloor \cdot 8} \circ bcd'(|a|, \lfloor \frac{n-1}{8} \rfloor \cdot 2 - 1) & : \text{else} \end{cases}$$

At first the sign bit must be set. Waste bits between sign and the first digit nibble are cleared to zero. Then the actual digits are computed from the number's absolute value.

$$\begin{aligned} bcd' : (\mathbb{N}, \mathbb{N}) &\rightarrow \mathbb{B}^* \\ bcd'(a, n) &= \begin{cases} bin_4(a \bmod 10) & : n = 0 \\ bcd'(\lfloor a/10 \rfloor, n-1) \circ bin_4(a \bmod 10) & : \text{else} \end{cases} \end{aligned}$$

One can prove that $\langle bcd_n(a) \rangle_{BCD} = a$ holds for this implementation.

6.2.4 Maximal and Minimal Values

The configuration-specific maximal and minimal values $X_{max}(f)$ and $X_{min}(f)$ are defined in the following way for the different formats.

$$\begin{aligned} X_{max}(f) &= \begin{cases} 2^{63} - 1 & : cvtint(f) \wedge ressize(f) = 64 \\ 2^{31} - 1 & : cvtint(f) \wedge ressize(f) = 32 \\ 2^{15} - 1 & : cvtint(f) \wedge ressize(f) = 16 \\ 10^{18} - 1 & : FBSTP(f) \\ 2^{e_{max}} \cdot (2 - 2^{-(p(f)-1)}) & : \text{otherwise} \end{cases} \\ X_{min}(f) &= \begin{cases} -2^{63} & : cvtint(f) \wedge ressize(f) = 64 \\ -2^{31} & : cvtint(f) \wedge ressize(f) = 32 \\ -2^{15} & : cvtint(f) \wedge ressize(f) = 16 \\ -(10^{18} - 1) & : FBSTP(f) \\ 2^{e_{min}} \cdot 2^{-(p(f)-1)} & : \text{otherwise} \end{cases} \end{aligned}$$

Note that e_{max} and e_{min} are also depending on the current configuration via $n(f)$. In addition these values represent the maximal positive or minimal negative numbers for integer data types, while they stand for the maximal or minimal non-zero absolute values for floating-point numbers.

6.3 x87 Mathematical Functions

Now we will focus on the mathematical functions behind the different instructions. We want to abstract this from the FPU and the several x87 instructions as far as possible. Thus we define a set of FPU functions by declaring the $FPUF$ type as follows.

$$FPUF : \text{ADD|SUB|MUL|DIV|PREM|SQRT|RND|XTRACT|} \\ \text{SIN|COS|SINCOS|TAN|ATAN|2XM1|SCALE|YL2X|YL2XP1}$$

All these functions can be computed by the FPU and we later on want to state for each of them separately how the result is calculated and which exceptions may occur. The particular function corresponding to the current floating-point configuration is denoted by $fpuf(opc[15:0])$ or $fpuf(f)$ respectively.

$$\begin{aligned} fpuf & : \mathbb{B}^{16} \rightarrow FPUF|\epsilon \\ fpuf & : FPU \rightarrow FPUF|\epsilon \\ fpuf(opc[15:0]) & = \begin{cases} \text{ADD} & : add(opc) \\ \text{SUB} & : sub(opc) \\ \text{MUL} & : mul(opc) \\ \text{DIV} & : div(opc) \\ \text{PREM} & : prem(opc) \\ \text{SQRT} & : FSQRT(opc) \\ \text{RND} & : FRNDINT(opc) \\ \text{XTRACT} & : FXTRACT(opc) \\ \text{SIN} & : FSIN(opc) \end{cases} \\ fpuf(f) & = fpuf(opc(f)) \end{aligned}$$

COS	$: FCOS(opc)$
SINCOS	$: FSINCOS(opc)$
TAN	$: FPTAN(opc)$
ATAN	$: FPATAN(opc)$
2XM1	$: F2XM1(opc)$
SCALE	$: FSCALE(opc)$
YL2X	$: FYL2X(opc)$
YL2XP1	$: FYL2XP1(opc)$
ϵ	$: \text{otherwise}$

In addition it is important to know how many operands a function needs to operate correctly. We already introduced a $nops(opc)$ function to determine the number of operands for particular instructions. Using $fpuf$ function we can easily specify the number of operands associated with particular mathematical functions.

$$\begin{aligned} nops & : FPUF|\epsilon \rightarrow \mathbb{N}_3 \\ nops(fn) & = \begin{cases} 2 & : \exists x \in \mathbb{B}^{16} : fpuf(x) = fn \wedge nops(x) = 2 \\ 1 & : \exists x \in \mathbb{B}^{16} : fpuf(x) = fn \wedge nops(x) = 1 \\ 0 & : \text{otherwise} \end{cases} \end{aligned}$$

Thus we do not have to consider opcodes any longer. Our abstract computation model becomes independent of the actual instruction set opcode specification. Now we can define the precise result of a calculating floating-point operation. In the style of the ALU result $aluop$ of the DLX in [MP00] we denote this by $fpuop$.

$$fpuop : (FPUF, \mathbb{B}^{80}|\epsilon, \mathbb{B}^{80}|\epsilon, \mathbb{B}^2) \rightarrow (\mathbb{R}_\infty|NaN|\epsilon, \mathbb{R}_\infty|NaN|\epsilon)$$

Note that this function is only dependent of the specific function to be computed, two operands and a rounding mode indicator. Under certain circumstances it will return an empty result.

$$fpuop(fn, op1, op2, rm) = (\epsilon, \epsilon) \Leftrightarrow fn = \epsilon \vee (op1 = \epsilon \wedge nops(fn) \neq 0 \vee op2 = \epsilon \wedge nops(fn) = 2)$$

Furthermore we are confronted with NaN operands, which produce NaN results. The following equations show how the NaNs are converted. Note that SNaNs in contrary to QNaNs cause invalid operation exceptions for mathematical operations and the results stated below are only valid when INV exceptions are masked. We also need to care about those functions producing two results, that is $fn2 \in \{\text{SINCOS}, \text{TAN}, \text{XTRACT}\}$. With $fn \notin \{\text{SINCOS}, \text{ATAN}, \text{XTRACT}\}$, $qnanx \in \mathbb{B}^{80}$, $x \in \mathbb{B}^{63}$, $\llbracket qnanx \rrbracket = \text{QNaN } x$ and $\llbracket op1 \rrbracket, \llbracket op2 \rrbracket \notin \text{NaN}$ we get:

$$\begin{aligned} fpuop(fn, qnanx, op2, rm) &= (\text{QNaN } x, \epsilon) \\ fpuop(fn, op1, qnanx, rm) &= (\text{QNaN } x, \epsilon) \\ fpuop(fn2, qnanx, op2, rm) &= \begin{cases} (\text{QNaN } x, 1) & : \quad fn2 = \text{TAN} \\ (\text{QNaN } x, \text{QNaN } x) & : \quad \text{otherwise} \end{cases} \end{aligned}$$

Now for $snanx \in \mathbb{B}^{80}$, $\llbracket snanx \rrbracket = \text{SNaN } x$, $\overline{\text{SNaN}(op1)}$ and $\overline{\text{SNaN}(op2)}$ we have:

$$\begin{aligned} fpuop(fn, snanx, op2, rm) &= (\text{QNaN } x, \epsilon) \\ fpuop(fn, op1, snanx, rm) &= (\text{QNaN } x, \epsilon) \\ fpuop(fn2, snanx, op2, rm) &= \begin{cases} (\text{QNaN } x, 1) & : \quad fn2 = \text{TAN} \\ (\text{QNaN } x, \text{QNaN } x) & : \quad \text{otherwise} \end{cases} \end{aligned}$$

At last we have a look at the combination of two equal NaN operands. Let $qnanx', snanx' \in \mathbb{B}^{80}$, $x' \in \mathbb{B}^{63}$, $\llbracket qnanx' \rrbracket = \text{QNaN } x'$ and $\llbracket snanx' \rrbracket = \text{SNaN } x'$, then:

$$\begin{aligned} fpuop(fn, qnanx, qnanx', rm) &= \begin{cases} (\text{QNaN } x, \epsilon) & : \quad \langle x[62:0] \rangle > \langle x'[62:0] \rangle \\ (\text{QNaN } x', \epsilon) & : \quad \langle x[62:0] \rangle < \langle x'[62:0] \rangle \\ (\text{QNaN } x, \epsilon) & : \quad x = x' \\ \text{undefined} & : \quad \text{otherwise} \end{cases} \\ fpuop(fn, snanx, snanx', rm) &= \begin{cases} (\text{SNaN } x, \epsilon) & : \quad \langle x[62:0] \rangle > \langle x'[62:0] \rangle \\ (\text{SNaN } x', \epsilon) & : \quad \langle x[62:0] \rangle < \langle x'[62:0] \rangle \\ (\text{SNaN } x, \epsilon) & : \quad x = x' \\ \text{undefined} & : \quad \text{otherwise} \end{cases} \end{aligned}$$

We do not have to consider $fn2$ here because all x87 instructions which generate two results reference only one operand. The undefined cases identify the situations that both NaNs exhibit the same error codes but have different signs.

In the following subsections we will give the exact definition of $fpuop$ by a case split on $FPUF$. That means for each function $fpuop$ will be defined separately. Accordingly we set up the detection of pre-computation exceptions and the definition of the respective masked exception results. Here we will focus on function specific exceptions which are mostly invalid operation exceptions, but also division-by-zero ones. The function $prechk$ checks for pre-computation exceptions based on the respective function and operands.

$$prechk : (FPUF, \mathbb{B}^{80}|\epsilon, \mathbb{B}^{80}|\epsilon) \rightarrow FPUEXCP^*$$

It returns a set of floating-point exceptions which were provoked by the given operands for the particular floating-point function. We will complete its definition step by step for each function type. For an empty function type we define:

$$prechk(\epsilon, op1, op2) = \emptyset$$

In addition we must state the masked exception result, that is the result of a computation returned in case of a masked pre-computation exception.

$$\begin{aligned} mexcpres : (FPUF, FPUXCP | \epsilon, \mathbb{B}^{80} | \epsilon, \mathbb{B}^{80} | \epsilon) &\rightarrow (\mathbb{B}^{80} | \epsilon, \mathbb{B}^{80} | \epsilon) \\ mexcpres(fn, excp, op1, op2) = (\epsilon, \epsilon) &\Leftrightarrow fn = \epsilon \vee excp = \epsilon \vee op1 = \epsilon \vee (op2 = \epsilon \wedge nops(fn) = 2) \end{aligned}$$

Once again we will specify these results individually for each function type.

From the IA-32 specification we learn that

$$\begin{aligned} \forall fn \in FPUF : \forall f \in FPU : \\ mexcpres(fn, DEN, op1(f), op2(f)) &= pack(\eta(r_f(fpuop(fn, op1(f), op2(f), rm(f))))), 80) \end{aligned}$$

holds. I.e. in case of denormal operands, the default result for masked exceptions is the ordinary *fpuop* result rounded and converted to double-extended floating-point format just like no exception occurred at all.

6.3.1 Addition

Not surprisingly the FPU is able to add two floating-point numbers. The result is the sum of the operands' values. In case of an invalid exception the 80-bit floating-point indefinite value is returned as a default response.

$$\begin{aligned} fpuop(ADD, op1, op2, rm) &= \\ &\begin{cases} (+0, \epsilon) & : s(op1) \neq s(op2) \wedge \llbracket op1 \rrbracket + \llbracket op2 \rrbracket = 0 \wedge rm \neq 01 \\ (-0, \epsilon) & : s(op1) \neq s(op2) \wedge \llbracket op1 \rrbracket + \llbracket op2 \rrbracket = 0 \wedge rm = 01 \\ (\llbracket op1 \rrbracket + \llbracket op2 \rrbracket, \epsilon) & : \text{otherwise} \end{cases} \\ mexcpres(ADD, INV, op1, op2) &= (indeffp80, \epsilon) \end{aligned}$$

An INV exception is caused when one operand exhibits an unsupported data format or when the operands are infinities with opposing signs.

$$INV \in prechk(ADD, op1, op2) \Leftrightarrow unsup(op1) \vee unsup(op2) \vee SNaN(op1) \vee SNaN(op2) \vee (infty(op1) \wedge infty(op2) \wedge (s(op1) \neq s(op2)))$$

Addition of infinities is exact, as long as the signs are equal. For these cases and for addition of equally signed zeroes we have:

$$\begin{aligned} \pm\infty + \pm\infty &= \pm\infty & \pm 0 + \pm 0 &= \pm 0 \\ \forall x \in \mathcal{R} : \pm\infty + x &= \pm\infty & x + \pm\infty &= \pm\infty \end{aligned}$$

6.3.2 Substraction

The FPU is also capable of substracting two floating-point numbers. The result is the difference of the operands' values, while the first operand serves as minuend and the second one as subtrahend. In case of an invalid exception the 80-bit floating-point indefinite value is returned as a default response.

$$\begin{aligned}
 fpuop(\text{SUB}, op1, op2, rm) &= \\
 &\begin{cases} (+0, \epsilon) & : s(op1) = s(op2) \wedge \llbracket op1 \rrbracket - \llbracket op2 \rrbracket = 0 \wedge rm \neq 01 \\
 (-0, \epsilon) & : s(op1) = s(op2) \wedge \llbracket op1 \rrbracket - \llbracket op2 \rrbracket = 0 \wedge rm = 01 \\
 (\llbracket op1 \rrbracket - \llbracket op2 \rrbracket, \epsilon) & : \text{otherwise} \end{cases} \\
 mexcpres(\text{SUB}, \text{INV}, op1, op2) &= (\text{indefp80}, \epsilon)
 \end{aligned}$$

An INV exception is caused when one operand exhibits an unsupported data format or when the operands are infinities with identical signs.

$$\text{INV} \in \text{prechk}(\text{SUB}, op1, op2) \Leftrightarrow \text{unsup}(op1) \vee \text{unsup}(op2) \vee \text{NaN}(op1) \vee \text{NaN}(op2) \vee (\text{inf}(op1) \wedge \text{inf}(op2) \wedge (s(op1) = s(op2)))$$

Subtraction of infinities is exact, as long as the signs are opposites. For these cases and for subtraction of oppositely signed zeroes we have:

$$\begin{aligned}
 \pm\infty - \mp\infty &= \pm\infty & \pm 0 - \mp 0 &= \pm 0 \\
 \forall x \in \mathcal{R} : \pm\infty - x &= \pm\infty & x - \pm\infty &= \mp\infty
 \end{aligned}$$

6.3.3 Multiplication

The floating-point unit provides multiplication of two floating-point numbers. The result is the product of the operands' values. In case of an invalid exception the 80-bit floating-point indefinite value is returned as a default response.

$$\begin{aligned}
 fpuop(\text{MUL}, op1, op2, rm) &= (\llbracket op1 \rrbracket \cdot \llbracket op2 \rrbracket, \epsilon) \\
 mexcpres(\text{MUL}, \text{INV}, op1, op2) &= (\text{indefp80}, \epsilon)
 \end{aligned}$$

An INV exception is caused when one operand exhibits an unsupported data format or when the operands are infinity and zero

$$\text{INV} \in \text{prechk}(\text{MUL}, op1, op2) \Leftrightarrow \text{unsup}(op1) \vee \text{unsup}(op2) \vee \text{NaN}(op1) \vee \text{NaN}(op2) \vee (\text{inf}(op1) \wedge (\llbracket op2 \rrbracket = 0) \vee \text{inf}(op2) \wedge (\llbracket op1 \rrbracket = 0))$$

Multiplication of infinities is always exact when there is no factor of zero:

$$\begin{aligned}
 \pm\infty \cdot \pm\infty &= +\infty & \pm\infty \cdot \mp\infty &= -\infty \\
 \forall x \in \mathcal{R} : |x| \neq 0 &\Rightarrow & \forall x \in \mathcal{R} : |x| \neq \infty &\Rightarrow \\
 \pm\infty \cdot |x| &= \pm\infty & \pm 0 \cdot |x| &= \pm 0 \\
 \pm\infty \cdot -|x| &= \mp\infty & \pm 0 \cdot -|x| &= \mp 0 \\
 |x| \cdot \pm\infty &= \pm\infty & |x| \cdot \pm 0 &= \pm 0 \\
 -|x| \cdot \pm\infty &= \mp\infty & -|x| \cdot \pm 0 &= \mp 0
 \end{aligned}$$

6.3.4 Division

Also division of two floating-point numbers is available in the FPU. The result is the quotient of the operands' values, while the first operand serves as dividend and the second one as divisor. In case of an invalid exception the 80-bit floating-point indefinite value is returned as a default response.

$$\begin{aligned} fpuop(\text{DIV}, op1, op2, rm) &= (\llbracket op1 \rrbracket / \llbracket op2 \rrbracket, \epsilon) \\ mexcpres(\text{DIV}, \text{INV}, op1, op2) &= (\text{indeffp80}, \epsilon) \\ mexcpres(\text{DIV}, \text{DBZ}, op1, op2) &= ((s(op1) \oplus s(op2)) \circ 1^{15} \circ 10^{63}, \epsilon) \end{aligned}$$

For a divide-by-zero exception, infinity is returned with a sign corresponding to the operands' sign bits. An INV exception is caused when one operand exhibits an unsupported data format or when the operands are both infinities or zeroes.

$$\begin{aligned} \text{INV} \in \text{prechk}(\text{DIV}, op1, op2) &\Leftrightarrow \text{unsup}(op1) \vee \text{unsup}(op2) \vee \text{NaN}(op1) \vee \text{NaN}(op2) \vee \\ &\quad \text{inf}(op1) \wedge \text{inf}(op2) \vee (\llbracket op1 \rrbracket = 0 \wedge \llbracket op2 \rrbracket = 0) \\ \text{DBZ} \in \text{prechk}(\text{DIV}, op1, op2) &\Leftrightarrow \llbracket op2 \rrbracket = 0 \end{aligned}$$

A DBZ exception occurs when the divisor is zero. Division of infinities is exact as long as not both operands are infinity:

$$\begin{aligned} \forall x \in \mathcal{R} : |x| \neq \infty \Rightarrow \pm\infty/|x| &= \pm\infty \\ \pm\infty/-|x| &= \mp\infty \\ |x|/\pm\infty &= \pm 0 \\ -|x|/\pm\infty &= \mp 0 \end{aligned}$$

6.3.5 Rounding

Apart from the inevitable rounding of exact results the FPU offers the possibility to explicitly round a floating-point number to an integer. However the result is stored in floating-point format. In case of an invalid exception the 80-bit floating-point indefinite values are returned as a default response.

$$\begin{aligned} fpuop(\text{RND}, op1, op2, rm) &= \begin{cases} (\llbracket op1 \rrbracket, \epsilon) & : \text{inf}(op1) \\ (\text{rndint}(\llbracket op1 \rrbracket, rm), \epsilon) & : \text{otherwise} \end{cases} \\ mexcpres(\text{RND}, \text{INV}, op1, op2) &= (\text{indeffp80}, \epsilon) \end{aligned}$$

Infinities can not be rounded to an integer. An INV exception is caused when one operand exhibits an unsupported data format.

$$\text{INV} \in \text{prechk}(\text{RND}, op1, op2) \Leftrightarrow \text{unsup}(op1) \vee \text{NaN}(op1)$$

6.3.6 Extracting Significand and Exponent

In some application it may be useful to split a floating point number into two parts, namely its exponent and signed significand. The FPU's FXTRACT instruction supplies us with this

functionality. In case of an invalid exception the 80-bit floating-point indefinite value is returned as a default response.

$$\begin{aligned} fpuop(\text{XTRACT}, op1, op2, rm) &= (\lfloor \log_2 \lfloor \lceil op1 \rceil \rfloor \rfloor, \lfloor \lceil op1 \rceil \rfloor / 2^{\lfloor \log_2 \lfloor \lceil op1 \rceil \rfloor \rfloor}) \\ mexcpres(\text{XTRACT}, \text{INV}, op1, op2) &= (\text{indefp80}, \text{indefp80}) \\ mexcpres(\text{XTRACT}, \text{DBZ}, op1, op2) &= (op1, 1 \circ 1^{15} \circ 10^{63}) \end{aligned}$$

For a divide-by-zero exception, zero and negative infinity are returned. An INV exception is caused when one operand exhibits an unsupported data format. A DBZ exception occurs, when the source operand is zero.

$$\begin{aligned} \text{INV} \in \text{prechk}(\text{XTRACT}, op1, op2) &\Leftrightarrow \text{unsup}(op1) \vee \text{SNaN}(op1) \\ \text{DBZ} \in \text{prechk}(\text{XTRACT}, op1, op2) &\Leftrightarrow \lfloor \lceil op1 \rceil \rfloor = 0 \end{aligned}$$

6.3.7 Partial Remainder

The FPU offers the possibility to calculate the partial remainder of a division of two floating-point numbers. In fact there are even two instructions² computing the partial remainder, but their algorithms only differ in the rounding modes they use. In case of an invalid exception the 80-bit floating-point indefinite value is returned as a default response. Let $ediff = \langle e(op1) \rangle_{bias} - \langle e(op2) \rangle_{bias}$ and $N = 32 + (ediff \bmod 32)$, then the result is computed as follows.

$$\begin{aligned} fpuop(\text{PREM}, op1, op2, rm) &= \begin{cases} (\pm 0, \epsilon) & : \lfloor \lceil op1 \rceil \rfloor = \pm 0 \\ (\lfloor \lceil op1 \rceil \rfloor, \epsilon) & : ediff < 0 \vee \text{infity}(op2) \\ (\lfloor \lceil op1 \rceil \rfloor - (\lfloor \lceil op2 \rceil \rfloor \cdot \text{rndint}(\frac{\lfloor \lceil op1 \rceil \rfloor}{\lfloor \lceil op2 \rceil \rfloor}, rm)), \epsilon) & : ediff < 64 \\ (\lfloor \lceil op1 \rceil \rfloor - (\lfloor \lceil op2 \rceil \rfloor \cdot \text{rndint}(\frac{\lfloor \lceil op1 \rceil \rfloor / 2^{ediff-N}}{\lfloor \lceil op2 \rceil \rfloor}, 11) \cdot 2^{ediff-N}), \epsilon) & : \text{otherwise} \end{cases} \\ mexcpres(\text{PREM}, \text{INV}, op1, op2) &= (\text{indefp80}, \epsilon) \\ mexcpres(\text{PREM}, \text{DBZ}, op1, op2) &= (\lfloor \lceil op1 \rceil \rfloor, \epsilon) \end{aligned}$$

In the third case the exponent difference of the two operands is larger than the number of fractional digits. Thus the quotient may have too many digits to be represented exactly. Then a computation according to the second case would not guarantee that the resulting remainder is smaller than the divisor. We want to exemplify the method of the operation for integers x and y . Analysing the formula of the third case the exponent difference between x and $y \cdot 2^{ediff-N}$ is then limited to the range $\{32, \dots, 63\}$. Thus an exact partial remainder

$$x - \lfloor \frac{x}{2^{ediff-N}y} \rfloor \cdot 2^{ediff-N}y = x \bmod 2^{ediff-N}y$$

can be computed. This partial remainder can be further processed to the actual remainder by successive application of the PREM instructions. Because of $\lfloor \frac{x}{2^{ediff-N}y} \rfloor \cdot 2^{ediff-N}y \in \mathbb{Z}$ we can follow that $(\lfloor \frac{x}{2^{ediff-N}y} \rfloor \cdot 2^{ediff-N}y) \bmod y = 0$ and thus:

$$\begin{aligned} (x - \lfloor \frac{x}{2^{ediff-N}y} \rfloor \cdot 2^{ediff-N}y) \bmod y &= ((x \bmod y) - (\lfloor \frac{x}{2^{ediff-N}y} \rfloor \cdot 2^{ediff-N}y \bmod y)) \bmod y \\ &= ((x \bmod y) - 0) \bmod y \\ &= x \bmod y \end{aligned}$$

²FPREM and FPREM1, while the latter one computes the partial remainder according to IEEE 754 standard.

In a similar way³ the exact remainder of $\frac{\llbracket op1 \rrbracket}{\llbracket op2 \rrbracket}$ is contained in the partial remainder.

Moreover an invalid operation exception is generated when one of the operands is in an unsupported format, the dividend is infinity or both operands are zero.

$$\begin{aligned} INV \in prechk(PREM, op1, op2) &\Leftrightarrow unsup(op1) \vee unsup(op2) \vee SNaN(op1) \vee SNaN(op2) \vee \\ &\quad infty(op1) \vee (\llbracket op1 \rrbracket = 0 \wedge \llbracket op2 \rrbracket = 0) \\ DBZ \in prechk(PREM, op1, op2) &\Leftrightarrow infty(op1) \wedge (\llbracket op1 \rrbracket \neq 0) \wedge (\llbracket op2 \rrbracket = 0) \end{aligned}$$

When only the divisor is zero it depends on the instruction if an INV or DBZ exception is caused. FPREM1 provokes a divide-by-zero exception in this case, while FPREM considers it as an invalid operation⁴

6.3.8 Square Root

For many applications it is necessary to compute the square root of floating-point numbers. Hence the FPU provides a hardware implementation of this function. In case of an invalid exception the 80-bit floating-point indefinite value is returned as a default response. Otherwise the result is computed as follows.

$$\begin{aligned} fpuop(SQRT, op1, op2, rm) &= (\sqrt{\llbracket op1 \rrbracket}, \epsilon) \\ mexcpres(SQRT, INV, op1, op2) &= (indefp80, \epsilon) \end{aligned}$$

An invalid operation exception occurs for non-zero negative numbers.

$$INV \in prechk(SQRT, op1, op2) \Leftrightarrow unsup(op1) \vee (\llbracket op1 \rrbracket < -0) \vee SNaN(op1)$$

For positive infinity $\sqrt{\infty} = \infty$ holds and for signed zeroes we have $\sqrt{\pm 0} = \pm 0$.

6.3.9 Sine and Cosine

Also trigonometric functions are supported by the x87 instruction subset. At first we have a look at sine and cosine. There are three FPU functions which return either sine, cosine or both sine and cosine of a floating-point number⁵. Only when the source operand is in the range $[-2^{63}, 2^{63}]$ a computation is performed. In case of an invalid exception the 80-bit floating-point indefinite value is returned as a default response. For $fn \in \{SIN, COS, SINCOS\}$ we have:

$$fpuop(fn, op1, op2, rm) = \begin{cases} (\sin \llbracket op1 \rrbracket, \epsilon) & : fn = SIN \wedge -2^{63} \leq \llbracket op1 \rrbracket \leq 2^{63} \\ (\cos \llbracket op1 \rrbracket, \epsilon) & : fn = COS \wedge -2^{63} \leq \llbracket op1 \rrbracket \leq 2^{63} \\ (\sin \llbracket op1 \rrbracket, \cos \llbracket op1 \rrbracket) & : fn = SINCOS \wedge -2^{63} \leq \llbracket op1 \rrbracket \leq 2^{63} \\ (\text{undefined}, \epsilon) & : \text{otherwise} \end{cases}$$

³We can conclude from integers on floating-point numbers because they have only a finite precision and can be scaled to integers. In a way floating-point numbers can be seen as rational numbers with a maximal denominator of 2^p .

⁴The Intel IA-32 Instruction Set Reference is unclear on this subject, thus we rely on the AMD64 Architecture Programmer's Manual.

⁵FSIN, FCOS and FSINCOS

$$\begin{aligned}
mexcpres(\text{SIN}, \text{INV}, op1, op2) &= (\text{indefp80}, \epsilon) \\
mexcpres(\text{COS}, \text{INV}, op1, op2) &= (\text{indefp80}, \epsilon) \\
mexcpres(\text{SINCOS}, \text{INV}, op1, op2) &= (\text{indefp80}, \epsilon)
\end{aligned}$$

An INV exception is triggered by infinities being held in the source operand.

Let $fn \in \{\text{SIN}, \text{COS}, \text{SINCOS}\}$, then:

$$\text{INV} \in \text{prechk}(fn, op1, op2) \Leftrightarrow \text{unsup}(op1) \vee \text{infity}(op1) \vee \text{NaN}(op1)$$

Observe the following definitions for signed zeroes.

$$\sin \pm 0 = \pm 0 \quad \cos \pm 0 = 1$$

6.3.10 Tangent and Arc Tangent

Additionally the FPU facilitates calculation of tangent as well as arc tangent of floating-point numbers. For the tangent the computation is executed only when the operand lies in the range $[-2^{63}, 2^{63}]$. Furthermore 1.0 is pushed on the stack.⁶

$$\begin{aligned}
fpuop(\text{TAN}, op1, op2, rm) &= \begin{cases} (\pm 0, 1) & : \llbracket op1 \rrbracket = \pm 0 \\ (\tan \llbracket op1 \rrbracket, 1) & : -2^{63} \leq \llbracket op1 \rrbracket \leq 2^{63} \\ (\text{undefined}, 1) & : \text{otherwise} \end{cases} \\
fpuop(\text{ATAN}, op1, op2, rm) &= \begin{cases} (\arctan(\frac{\llbracket op2 \rrbracket}{\llbracket op1 \rrbracket}), \epsilon) & : \llbracket op1 \rrbracket > 0 \wedge \llbracket op2 \rrbracket \neq 0 \\ (\arctan(\frac{\llbracket op2 \rrbracket}{\llbracket op1 \rrbracket}) + \pi, \epsilon) & : \llbracket op1 \rrbracket < 0 \wedge \llbracket op2 \rrbracket > 0 \\ (\arctan(\frac{\llbracket op2 \rrbracket}{\llbracket op1 \rrbracket}) - \pi, \epsilon) & : \llbracket op1 \rrbracket < 0 \wedge \llbracket op2 \rrbracket < 0 \\ (\frac{\pi}{2}, \epsilon) & : \llbracket op1 \rrbracket = 0 \wedge \llbracket op2 \rrbracket > 0 \\ (-\frac{\pi}{2}, \epsilon) & : \llbracket op1 \rrbracket = 0 \wedge \llbracket op2 \rrbracket < 0 \\ (\pm \pi, \epsilon) & : \llbracket op1 \rrbracket \leq -0 \wedge \llbracket op2 \rrbracket = \pm 0 \\ (\pm 0, \epsilon) & : \llbracket op1 \rrbracket \geq +0 \wedge \llbracket op2 \rrbracket = \pm 0 \end{cases} \\
mexcpres(\text{TAN}, \text{INV}, op1, op2) &= (\text{indefp80}, \epsilon) \\
mexcpres(\text{ATAN}, \text{INV}, op1, op2) &= (\text{indefp80}, \epsilon)
\end{aligned}$$

For invalid operation exceptions the floating-point indefinite value is returned as default. Besides operands in unsupported formats only infinities can cause an INV exception for FPTAN instruction.

$$\begin{aligned}
\text{INV} \in \text{prechk}(\text{TAN}, op1, op2) &\Leftrightarrow \text{unsup}(op1) \vee \text{infity}(op1) \vee \text{NaN}(op1) \\
\text{INV} \in \text{prechk}(\text{ATAN}, op1, op2) &\Leftrightarrow \text{unsup}(op1) \vee \text{NaN}(op1)
\end{aligned}$$

The ATAN operation's result is well-defined for all possible arguments as the polar coordinate system angle of a point with ordinate $\llbracket op2 \rrbracket$ and abscissa $\llbracket op1 \rrbracket$. The following equations show the behaviour of arctan for special operands.

$$\begin{aligned}
\arctan(\frac{\pm\infty}{-\infty}) &= \mp\frac{\pi}{4} & \arctan(\frac{\pm\infty}{+\infty}) &= \pm\frac{\pi}{4} \\
\forall x \in \mathcal{R} : \arctan(\frac{\pm\infty}{\pm|x|}) &= \mp\frac{\pi}{2} & \forall x \in \mathcal{R} : \arctan(\frac{\pm\infty}{\pm|x|}) &= \pm\frac{\pi}{2} \\
\forall x \in \mathcal{R} : \arctan(\frac{\pm|x|}{-\infty}) &= \mp 0 & \forall x \in \mathcal{R} : \arctan(\frac{\pm|x|}{+\infty}) &= \pm 0
\end{aligned}$$

⁶This is done to ensure comfortable reversibility of the FPTAN instruction by the FPATAN instruction.
 $\arctan(\frac{\tan x}{1.0}) = x$

These special results shall be inserted for the arctan terms in the above definitions.

6.3.11 Exponentiation

There are two exponential functions available in the FPU. The first 2XM1 one directly exponentiates 2 with the value of the first operand and then subtracts 1. Only for operands in the range $[-1.0, 1.0]$ the computation is executed⁷. This is a restriction from the specification. The second function SCALE, scales the first operand with the power of two, gained from the truncated integer value of the second operand.

$$\begin{aligned}
fpuop(2XM1, op1, op2, rm) &= \begin{cases} (2^{\llbracket op1 \rrbracket} - 1, \epsilon) & : \quad -1 \leq \llbracket op1 \rrbracket \leq 1 \\ (\text{undefined}, \epsilon) & : \quad \text{otherwise} \end{cases} \\
fpuop(SCALE, op1, op2, rm) &= \begin{cases} (\text{QNaN } 1 \circ 0^{62}, \epsilon) & : \quad (\llbracket op1 \rrbracket, \llbracket op2 \rrbracket) \in \{(0, +\infty), (\infty, -\infty)\} \\ (\llbracket op1 \rrbracket \cdot 2^{rndint(\llbracket op2 \rrbracket, 11)}, \epsilon) & : \quad \text{otherwise} \end{cases} \\
mexcpres(2XM1, INV, op1, op2) &= (\text{indefp80}, \epsilon) \\
mexcpres(SCALE, INV, op1, op2) &= (\text{indefp80}, \epsilon)
\end{aligned}$$

In case of invalid operation exceptions the floating-point indefinite value is returned. This exception occurs only because of unsupported operand data formats for these functions.

$$\begin{aligned}
INV \in prechk(2XM1, op1, op2) &\Leftrightarrow unsup(op1) \vee SNaN(op1) \\
INV \in prechk(SCALE, op1, op2) &\Leftrightarrow unsup(op1) \vee SNaN(op1)
\end{aligned}$$

However there is a speciality about the FSCALE instruction, as it is the only one which creates a NaN result, without causing an exception and without any operands containing a NaN. This happens for the cases that $\llbracket op1 \rrbracket = 0 \wedge \llbracket op2 \rrbracket = +\infty$ or $\llbracket op1 \rrbracket = \infty \wedge \llbracket op2 \rrbracket = -\infty$, due to the fact that the product of zero and infinity is undefined.

6.3.12 Logarithm

As a counterpart to exponentiation the FPU is also able to compute the dual logarithm of floating-point numbers. There are two functions available to this end, YL2X and YL2XP1. For YL2X the first operand must be greater than zero, while YL2XP1 demands the first operand to be in range

⁷When the operand is outside the range, the result is undefined.

$[\sqrt{\frac{1}{2}} - 1, \sqrt{2} - 1]$.⁸ Then the results are computed as follows.

$$\begin{aligned}
fpuop(YL2X, op1, op2, rm) &= \begin{cases} ((-1)^{\overline{s(op2)}} \cdot \infty, \epsilon) & : \llbracket op1 \rrbracket = 0 \wedge \text{infity}(op2) \\ (\llbracket op2 \rrbracket \cdot \log_2 \llbracket op1 \rrbracket, \epsilon) & : \text{otherwise} \end{cases} \\
fpuop(YL2XP1, op1, op2, rm) &= \begin{cases} (\pm 0, \epsilon) & : \llbracket op1 \rrbracket = \pm 0 \wedge \llbracket op2 \rrbracket \geq +0 \\ (\mp 0, \epsilon) & : \llbracket op1 \rrbracket = \pm 0 \wedge \llbracket op2 \rrbracket \leq -0 \\ (\llbracket op2 \rrbracket \cdot \log_2(\llbracket op1 \rrbracket + 1), \epsilon) & : \sqrt{\frac{1}{2}} - 1 \leq \llbracket op1 \rrbracket \leq \sqrt{2} - 1 \\ \text{undefined} & : \text{otherwise} \end{cases} \\
mexcpres(YL2X, INV, op1, op2) &= (\text{indefp80}, \epsilon) \\
mexcpres(YL2X, DBZ, op1, op2) &= (\overline{s(op2)} \circ 1^{15} \circ 10^{63}, \epsilon) \\
mexcpres(YL2XP1, INV, op1, op2) &= (\text{indefp80}, \epsilon)
\end{aligned}$$

In case of invalid operation exceptions the floating-point integer value is returned. For a divide-by-zero exception during computation of YL2X infinity is returned with a sign in opposition to the one of the second operand. In the following cases exceptions are triggered.

$$\begin{aligned}
INV \in \text{prechk}(YL2X, op1, op2) &\Leftrightarrow \text{unsup}(op1) \vee \text{NaN}(op1) \vee \llbracket op1 \rrbracket < 0 \vee \\
&\quad (\llbracket op1 \rrbracket = 1 \wedge \text{infity}(op2)) \vee \\
&\quad (\llbracket op1 \rrbracket \in \{0, \infty\} \wedge \llbracket op2 \rrbracket = 0) \\
DBZ \in \text{prechk}(YL2X, op1, op2) &\Leftrightarrow \llbracket op1 \rrbracket = 0 \wedge \llbracket op2 \rrbracket \notin \{0, \infty\} \\
INV \in \text{prechk}(YL2XP1, op1, op2) &\Leftrightarrow \text{unsup}(op1) \vee \text{NaN}(op1) \vee (\llbracket op1 \rrbracket = 0 \wedge \text{infity}(op2))
\end{aligned}$$

For YL2X there are a lot of exceptional situations. The logarithm of negative numbers is undefined and thus interpreted as an invalid operation. For the first operand being zero the logarithm tends to negative infinity and a DBZ exception is raised as long as the other operand is a finite and non-zero number. Whenever zero is multiplied with infinity an INV exception is provoked.

6.4 x87 Comparison Functions

Now that we finished the mathematical floating-point functions only comparison functions are left to be defined. Comparisons in the floating-point unit are related to the values of the *RFLAGS* or x87 condition code zero, carry and parity bits. These flags can be altered using *cmp(f)*-instructions according to the comparison of two floating-point numbers. Conditional moves may rely on these results as they reference the respective *RFLAGS* bits. At first we introduce the various conditions that may be checked for by conditional move instructions. The condition type is denoted by *COND*.

$$COND : B|BE|E|NB|NBE|NE|NU|U$$

These comparison result types stand for:

⁸Here the Intel IA-32 Instruction Set Reference and AMD64 Architecture Programmer's Manual contradict themselves. For Intel IA-32 processors the upper interval boundary is already at $1 - \sqrt{\frac{1}{2}}$ while for AMD64 processors the given range is valid.

- B - Below
- BE - Below or Equal
- E - Equal
- NB - Not Below
- NBE - Neither Below nor Equal
- NE - Not Equal
- NU - Not Unordered
- U - Unordered

They are associated with a conditional move in case of $FCMOV(f) = 1$ using the following function.

$$cond : FPU \rightarrow COND$$

$$cond(f) = \begin{cases} B & : FCMOV(f) \wedge opc(f)[7 : 0] = 0xDA \wedge \langle I(f).modrm[5 : 3] \rangle = 0 \\ BE & : FCMOV(f) \wedge opc(f)[7 : 0] = 0xDA \wedge \langle I(f).modrm[5 : 3] \rangle = 2 \\ E & : FCMOV(f) \wedge opc(f)[7 : 0] = 0xDA \wedge \langle I(f).modrm[5 : 3] \rangle = 1 \\ NB & : FCMOV(f) \wedge opc(f)[7 : 0] = 0xDB \wedge \langle I(f).modrm[5 : 3] \rangle = 4 \\ NBE & : FCMOV(f) \wedge opc(f)[7 : 0] = 0xDB \wedge \langle I(f).modrm[5 : 3] \rangle = 6 \\ NE & : FCMOV(f) \wedge opc(f)[7 : 0] = 0xDB \wedge \langle I(f).modrm[5 : 3] \rangle = 5 \\ U & : FCMOV(f) \wedge opc(f)[7 : 0] = 0xDA \wedge \langle I(f).modrm[5 : 3] \rangle = 3 \\ NU & : FCMOV(f) \wedge opc(f)[7 : 0] = 0xDB \wedge \langle I(f).modrm[5 : 3] \rangle = 7 \end{cases}$$

To compute the values of the zero, carry and parity flags in the condition code and *RFLAGS* bits we define three *compxf*(*x*, *y*)-functions. They determine for two floating-point operands whether the particular flag is set to one or cleared to zero.

$$\begin{aligned} compzf & : (\mathbb{B}^{80}, \mathbb{B}^{80}) \rightarrow \mathbb{B} \\ compcf & : (\mathbb{B}^{80}, \mathbb{B}^{80}) \rightarrow \mathbb{B} \\ comppf & : (\mathbb{B}^{80}, \mathbb{B}^{80}) \rightarrow \mathbb{B} \end{aligned}$$

The zero flag is set to one for equal operands or an unordered exception.

$$compzf(x, y) = 1 \Leftrightarrow (\llbracket x \rrbracket = \llbracket y \rrbracket) \vee NaN(x) \vee NaN(y) \vee unsup(x) \vee unsup(y) = 1$$

The carry flag is set to one when the first operand is less than the second one or there is an unordered exception.

$$compcf(x, y) = 1 \Leftrightarrow (\llbracket x \rrbracket < \llbracket y \rrbracket) \vee NaN(x) \vee NaN(y) \vee unsup(x) \vee unsup(y) = 1$$

Keep in mind that equality and less relations ignore the sign of zero.

The parity flag is only set to one for unordered exceptions.

$$comppf(x, y) = 1 \Leftrightarrow NaN(x) \vee NaN(y) \vee unsup(x) \vee unsup(y) = 1$$

For a given FPU Configuration we yield the flag values from the following functions. Note that x stands for z , c and p here. The three functions are defined under the condition that $\overline{cmp(f) \wedge FXAM(f)} = 1$ holds.

$$\begin{aligned} compxf : FPU &\rightarrow \mathbb{B} \\ compxf(f) &= compxf(op1(f), op2(f)) \end{aligned}$$

No more functions are necessary to handle floating-point comparisons.

There are also no floating-point operations left to take care of. Remaining functionalities such as stack manipulation or control instructions will be handled in the next chapter along with the complete x87 execution routine and the rounding and packing of results.

Chapter 7

Execution

Finally we have collected all the important ingredients necessary to build our transition function $\sigma_{fpu} : FPU \rightarrow FPU$ on the set of floating-point configurations. For relevant cases we decompose this function into four subfunctions which are applied on the FPU configuration sequentially. This allows us to describe the floating-point execution routine in a structural and orderly way.

$$\sigma_{fpu}(fpu) = \begin{cases} \sigma_{fpu}^{store}(\sigma_{fpu}^{postchk}(\sigma_{fpu}^{exec}(\sigma_{fpu}^{prechk}(fpu)))) & : \quad fpu.acc \neq \epsilon \\ fpu & : \quad \text{otherwise} \end{cases}$$

The various functions serve the following purposes.

- $\sigma_{fpu}^{prechk} : FPU \rightarrow FPU$ - Checking for pre-computation exceptions and setting the x87 exception pointers and flags.
- $\sigma_{fpu}^{exec} : FPU \rightarrow FPU$ - Computing the instruction's result and storing it on the stack when required.
- $\sigma_{fpu}^{postchk} : FPU \rightarrow FPU$ - Checking for post-computation exceptions, setting the corresponding flags and adjusting the computed result when required.
- $\sigma_{fpu}^{store} : FPU \rightarrow FPU$ - Returning the instruction's result and popping the stack when required.

In the following sections we will define each function separately, taking care of all different function classes and variants. After finishing these tasks our model to describe the effect of x87 instructions will be complete.

7.1 x87 Pre-Computation Exception Checks

The first transition subfunction serves solely pre-computation exception related purposes. On the one hand it detects these exception and sets the corresponding flags on the other hand it saves the

floating-point exception pointers, that is information for the exception handler to recover from an exception. We already introduced x87 exceptions and investigated the possible pre-computation exceptions for mathematical floating-point instructions. Now we want to complete the overall picture by defining all possible circumstances under which the particular exceptions occur. Afterwards we will state the effect of transition function $\sigma_{fpu}^{prechk}(f)$.

7.1.1 Detecting Pre-Computation Exceptions

The detection of pre-computation exceptions is quite complex, as there is a huge amount of special cases for each floating-point instruction which will result in an exception. We already covered a lot of these with the $prechk : (FPUF, \mathbb{B}^{80}|\epsilon, \mathbb{B}^{80}|\epsilon) \rightarrow FPUEXCP^*$ function, yet there are still some instructions left which deserve our attention. Thus we introduce the following function which subsumes all possible pre-computation exceptions in a set of floating-point exceptions.

$$prechk : FPU \rightarrow FPUEXCP^*$$

The $prechk(f)$ function comprises all x87 pre-computation exceptions which may occur for a floating-point operation. As there may occur multiple exceptions for one instruction it results in a set of FPU exceptions. The $prechk$ -function for calculating instructions was defined accordingly. Before we examine the particular exceptions we define the following predicates covering NaN operands.

$$\begin{aligned} NaNs : FPU &\rightarrow \mathbb{B} \\ NaNs(f) = 1 &\Leftrightarrow (NaN(op1(f)) \vee NaN(op2(f))) = 1 \\ SNaNs : FPU &\rightarrow \mathbb{B} \\ SNaNs(f) = 1 &\Leftrightarrow (SNaN(op1(f)) \vee SNaN(op2(f))) = 1 \\ QNaNs : FPU &\rightarrow \mathbb{B} \\ QNaNs(f) = 1 &\Leftrightarrow (QNaN(op1(f)) \vee QNaN(op2(f))) = 1 \end{aligned}$$

Now we will define the conditions for every kind of pre-computation exception. We begin with stack faults as they are the most important exceptions. Stack faults may occur for stack overflows or underflows.

$$STK \in prechk(f) \Leftrightarrow stkovf(f) \vee stkunf(f)$$

These predicates are defined as follows.

$$\begin{aligned} stkovf : FPU &\rightarrow \mathbb{B} \\ stkovf(f) &= \overline{push?(f) \wedge save?(f) \wedge \overline{empty(f, trgt_1(f))}} \vee \\ &\quad \overline{push?(f) \wedge save?(f) \wedge \overline{empty(f, trgt_2(f))}} \\ stkunf : FPU &\rightarrow \mathbb{B} \\ stkunf(f) &= (\overline{srop1(f) \vee srop2(f)}) \wedge \overline{empty(f, \langle opc(f)[2 : 0] \rangle)} \vee \\ &\quad (\overline{xread(f) \wedge op1(f) \neq \epsilon} \wedge \overline{srop1(f) \vee st0op2(f)}) \wedge \overline{empty(f, 0)} \vee \\ &\quad \overline{st1op2(f) \wedge empty(f, 1)} \end{aligned}$$

That means a stack overflow occurs when a value is pushed to a non-empty register location. A stack underflow on the contrary is caused by referencing an empty FPU register. Observe that the

term $\overline{xread(f)} \wedge op1(f) \neq \epsilon \wedge \overline{srop1(f)}$ identifies the situation that the first operand is located in the stack-top register.

We continue with invalid operation exceptions which may be raised in various situations. For the sake of clarity we define a predicate for each individual situation. The most common case is that a mathematical operation is invalid.

$$INVcalc(f) = calc(f) \wedge INV \in prechk(fpu f(f), op1(f), op2(f))$$

We already covered these cases with the *prechk* function. Furthermore a divide-by-zero exception for the *FPREM* instruction results in an invalid operation exception.

$$INVprem(f) = FPREM(f) \wedge (DBZ \in prechk(fpu f(f), op1(f), op2(f)))$$

When floating-point numbers shall be rounded to integers and stored to memory their magnitude may not exceed the limits of the destination format. Otherwise an INV exception is raised. This happens also if one or both operands are infinity, NaNs or in an unsupported format. In the latter two cases the precision converting floating-point store instructions produce the invalid operation exception, too.

$$\begin{aligned} INVcvt(f) = & (FBSTP(f) \vee cvtint(f)) \wedge (NaNs(f) \vee infity(op1(f)) \vee unsup(op1(f)) \vee \\ & rndint(\llbracket op1(f) \rrbracket) > X_{max}(f) \vee rndint(\llbracket op1(f) \rrbracket) < X_{min}(f)) \vee \\ & cvtprc(f) \wedge (SNaN(op1(f)) \vee unsup(op1(f))) \end{aligned}$$

Also comparison instructions cause invalid operations for NaNs or unsupported operands, depending on the kind of comparisons. Unordered compare instructions ignore whether the operands are QNaNs.

$$\begin{aligned} INVcmp(f) = & (ord(f) \vee FTST(f) \vee intcmp(f)) \wedge NaNs(f) \vee unord(f) \wedge SNaNs(f) \vee \\ & cmp(f) \wedge (unsup(op1(f)) \vee unsup(op2(f))) \end{aligned}$$

Finally we can state the conditions for an INV exception.

$$INV \in prechk(f) \Leftrightarrow INVcalc(f) \vee INVprem(f) \vee INVcvt(f) \vee INVcmp(f) = 1$$

Besides we have also divide-by-zero exceptions. These occur only for mathematical operations.

$$DBZ \in prechk(f) \Leftrightarrow calc(f) \wedge \overline{FPREM(f)} \wedge (DBZ \in prechk(fpu f(f), op1(f), op2(f)))$$

Again we have to consider that a DBZ exception while executing *FPREM* is signalled as an invalid operation. The least significant pre-computation exception is the denormalized operation exception. It is called if a computation or comparison involves a denormal operand. For *FXAM* no DEN exception is generated because it examines the content of a register for all types of numbers.

$$\begin{aligned} DEN \in prechk(f) \Leftrightarrow & (calc(f) \vee cmp(f) \wedge \overline{FXAM(f)}) \wedge \\ & (dnrml(op1(f)) \vee dnrml(op2(f))) \end{aligned}$$

Thusly we accomplished the detection of pre-computation exceptions in a quite intuitive manner. In the following we care about the according changes in the FPU configuration.

7.1.2 Transition Function

The $\sigma_{fpu}^{prechk}(f)$ transition subfunction results in a new FPU configuration f' .

$$\sigma_{fpu}^{prechk}(f) = f'$$

This configuration shall embody the changes due to the detected pre-computation exception. It also contains the x87 exception pointers for non-control x87 instructions. Accordingly the components of f' are defined as follows.

$$\begin{aligned} sf(f') &= \begin{cases} 1 & : \text{STK} \in prechk(f) \\ sf(f) & : \text{otherwise} \end{cases} \\ ie(f') &= \begin{cases} 1 & : \text{STK} \in prechk(f) \vee \text{INV} \in prechk(f) \\ ie(f) & : \text{otherwise} \end{cases} \\ ze(f') &= \begin{cases} 1 & : \text{DBZ} \in prechk(f) \\ ze(f) & : \text{otherwise} \end{cases} \\ de(f') &= \begin{cases} 1 & : \text{DEN} \in prechk(f) \\ ie(f) & : \text{otherwise} \end{cases} \end{aligned}$$

Because the exception flags are not cleared automatically, masked exception flags are accumulated during consecutive x87 calculations. The exception status flag and busy bit are set according to their definitions.

$$\begin{aligned} es(f') &\equiv \bigvee_{i=0}^5 mca(f')[i] \\ busy(f') &\equiv es(f') \end{aligned}$$

Only for non-control instructions the x87 exception information is saved.

$$\begin{aligned} f'.dp &= \begin{cases} adjustdp(f) & : nonctrl(I(f)) \\ f.dp & : \text{otherwise} \end{cases} \\ f'.ip &= \begin{cases} adjustip(f) & : nonctrl(I(f)) \\ f.ip & : \text{otherwise} \end{cases} \\ f'.opc &= \begin{cases} opc(f)[10:0] & : nonctrl(I(f)) \\ f.opc & : \text{otherwise} \end{cases} \end{aligned}$$

The precomputation exception check phase may also affect condition code bits 1 and 2.

$$\begin{aligned}
cc(f')[1] &= \begin{cases} 0 & : \quad sign(f) \vee FDECSTP(f) \vee FINCSTP(f) \vee FXCH(f) \vee \\ & \quad FUCOMI(f) \vee FUCOMIP(f) \vee \\ & \quad (STK \in prechk(f) \wedge stkunf(f)) \vee \\ & \quad (STK \notin prechk(f) \wedge (load(f) \vee const(f))) \\ 1 & : \quad (STK \in prechk(f) \wedge stkovf(f)) \\ cc(f)[1] & : \quad \text{otherwise} \end{cases} \\
cc(f')[2] &= \begin{cases} 0 & : \quad (FSIN(f) \vee FCOS(f) \vee FSINCOS(f) \vee FPTAN(f)) \wedge \\ & \quad \llbracket op1(f) \rrbracket \in [-2^{63}, 2^{63}] \\ 1 & : \quad (FSIN(f) \vee FCOS(f) \vee FSINCOS(f) \vee FPTAN(f)) \wedge \\ & \quad \llbracket op1(f) \rrbracket \notin [-2^{63}, 2^{63}] \wedge \overline{NaN(op1(f))} \\ cc(f)[2] & : \quad \text{otherwise} \end{cases}
\end{aligned}$$

The first condition code bit signals in case of a stack fault whether a stack underflow or overflow occurred. For some instructions it is always zero in case no stack fault was raised. The second code bit is set to one in case an argument was out of range for the trigonometric functions. All other components remain unchanged by $\sigma_{fpu}^{prechk}(f)$.

$$\begin{aligned}
&\forall comp \in f' : \\
&comp \in \{fsw[14 : 11], fsw[8], fsw[5 : 3], fpr, fcw, ftw, acc, ans\} \Rightarrow f'.comp = f.comp
\end{aligned}$$

This completes the definition of the first transition subfunction.

7.2 x87 Result Computation and Storage

The second transition subfunction $\sigma_{fpu}^{exec}(f)$ computes the result of an instruction and saves it to the floating-point register stack when necessary. This happens only when no unmasked pre-computation exceptions occurred. As the execution stage is based on the result of the $\sigma_{fpu}^{prechk}(f)$ function, these checks were already performed and we can use their results. To comfortably determine the active pre-computation exception with top priority we define the following function.

$$\begin{aligned}
prexcp &: FPU \rightarrow FPUEXCP|\epsilon \\
prexcp(f) &= \begin{cases} STK & : \quad ie(f) \wedge sf(f) \\ INV & : \quad ie(f) \wedge \overline{sf(f)} \\ DBZ & : \quad ze(f) \wedge \overline{ie(f)} \\ DEN & : \quad de(f) \wedge \overline{ze(f) \vee ie(f)} \\ \epsilon & : \quad \text{otherwise} \end{cases}
\end{aligned}$$

For the case of masked exceptions $\sigma_{fpu}^{exec}(f)$ incorporates the necessary default routines and results.

7.2.1 Transition Function

To define the execution stage transition function we must distinguish several cases. The changes to the configuration naturally depend on the type of instruction and its stack manipulation behaviour. When there are unmasked pre-computation exceptions no changes take place.

$$\sigma_{fpu}^{exec}(f) = \begin{cases} push(res_2(f), save(res_1(f), trgt_1(f), f)) & : push?(f) \wedge save?(f) \wedge \overline{es(f)} \\ save(res_1(f), trgt_1(f), f) & : save?(f) \wedge \overline{push?(f)} \wedge \overline{es(f)} \\ push(res_1(f), f) & : push?(f) \wedge \overline{save?(f)} \wedge \overline{es(f)} \\ save(op2(f), trgt_2(f), save(op1(f), trgt_1(f), f)) & : FXCH(f) \wedge \overline{es(f)} \\ set(f) & : set?(f) \wedge \overline{es(f)} \\ f & : otherwise \end{cases}$$

Here several new functions appear. These are stack operations such as *push* and *save*, the computation results res_1 and res_2 as well as the *set* function which handles control, stack and comparison instructions. We will discuss them in the following sections.

7.2.2 Stack Operations

Floating-point instructions differ in the way they manipulate the stack. We already established predicates which signal whether an instruction saves or pushes a value to the stack. Now we want to define atomic functions which save or push a value to the stack of an arbitrary FPU configuration. These are:

$$\begin{aligned} push & : (\mathbb{B}^{80}|_{\epsilon}, FPU) \rightarrow FPU \\ save & : (\mathbb{B}^{80}|_{\epsilon}, \mathbb{N}_8, FPU) \rightarrow FPU \end{aligned}$$

Pushing Values to the Stack

We start with implementing the *push* functionality, by defining the resulting configuration $f' = push(x, f)$ with $x \in \mathbb{B}^{80}|_{\epsilon}$. Figure 7.1 displays the effects of a push operation on the stack. The stack-top pointer must be decremented because the stack indices follow the ascending order of registers in the floating-point register file.

$$\langle top(f') \rangle = (\langle top(f) \rangle - 1) \bmod 8$$

In the new stack-top element x shall be saved, such that $st(f', 0) = x$ holds. All other stack registers stay unchanged. Observe that the new stack-top register is the old stack register 7. Let $i \in \mathbb{N}_8$, then:

$$f'.fpr(i) = \begin{cases} x & : i = sri(f, 7) \wedge x \neq \epsilon \\ f.fpr(i) & : otherwise \end{cases}$$

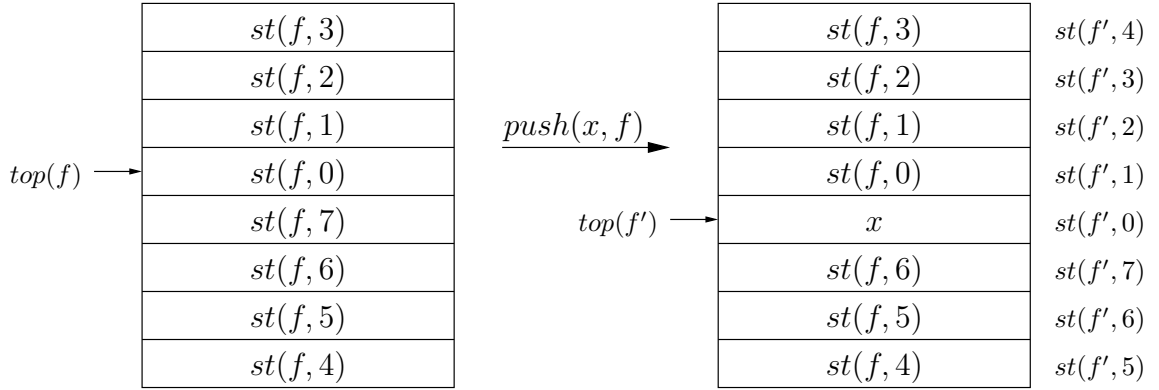


Figure 7.1: Effect of a push operation on the x87 stack

At last we examine x and set the tag bits accordingly.

$$f'.ftw[2 \cdot sri(f, 7) + 1 : 2 \cdot sri(f, 7)] = \begin{cases} 11 & : x = \epsilon \\ 01 & : |\llbracket x \rrbracket| = 0 \\ 10 & : NaN(x) \vee unsup(x) \vee dnrml(x) \vee infty(x) \\ 00 & : \text{otherwise} \end{cases}$$

The remaining components are kept unaltered.

$$\begin{aligned} f'.opc &= f.opc \\ f'.dp &= f.dp & f'.fsw[15 : 14] &= f.fsw[15 : 14] \\ f'.ip &= f.ip & f'.fsw[10 : 0] &= f.fsw[10 : 0] \\ f'.fcw &= f.fcw & f'.ftw[15 : 2 \cdot sri(f, 7) + 2] &= f.ftw[15 : 2 \cdot sri(f, 7) + 2] \\ f'.acc &= f.acc & f'.ftw[2 \cdot sri(f, 7) - 1 : 0] &= f.ftw[2 \cdot sri(f, 7) - 1 : 0] \\ f'.ans &= f.ans \end{aligned}$$

Saving Values in the Stack

Saving a value in a stack register is even more simple. Once again we have $f' = save(x, i, f)$ with $x \in \mathbb{B}^{80} \setminus \epsilon$ and $i, j \in \mathbb{N}_8$. As the stack-top pointer is not changed all we have to do is updating the register and its tag bits.

$$\begin{aligned} f'.fpr(j) &= \begin{cases} x & : j = i \wedge x \neq \epsilon \\ f.fpr(j) & : \text{otherwise} \end{cases} \\ f'.ftw[2i + 1 : 2i] &= \begin{cases} 11 & : x = \epsilon \\ 01 & : |\llbracket x \rrbracket| = 0 \\ 10 & : NaN(x) \vee unsup(x) \vee dnrml(x) \vee infty(x) \\ 00 & : \text{otherwise} \end{cases} \end{aligned}$$

Accordingly the other components do not change.

$$\begin{array}{llllll}
f'.opc & = & f.opc & f'.fsw & = & f.fsw & f'.fcw & = & f.fcw \\
f'.dp & = & f.dp & f'.acc & = & f.acc & f'.ftw[15 : 2i + 2] & = & f.ftw[15 : 2i + 2] \\
f'.ip & = & f.ip & f'.ans & = & f.ans & f'.ftw[2i - 1 : 0] & = & f.ftw[2i - 1 : 0]
\end{array}$$

Apart from these stack operations instructions may also pop the stack. This is handled in section 7.4.2.

7.2.3 Computing the Result

The main task of the execution phase is to deliver the result of a floating-point instruction. Not every operation produces such a result bit string, on the other hand some instructions even return two results. We denote these results by $res_1(f)$ and $res_2(f)$ respectively.

$$\begin{array}{ll}
res : FPU & \rightarrow (\mathbb{B}^*|\epsilon, \mathbb{B}^*|\epsilon) \\
res(f) & = (res_1(f), res_2(f))
\end{array}$$

The pair of results $res(f)$ is a non-trivial definition as it compiles a multitude of instruction classes and types. We distinguish eight different result types.

- Calculation result - a result obtained from a floating-point calculation
- Sign operation result - a result obtained from manipulating the sign of a floating-point number
- Comparison result - the resulting *EFLAGS* register content corresponding to a floating-point comparison
- Constant result - a floating-point number from a fixed set of predefined constants which is to be pushed on the stack
- Conversion result - a floating-point number converted to a different number format or precision which shall be stored to memory
- Load result - a floating-point number which is loaded from memory or is involved in a conditional move operation
- Control result - a bitstring which contains parts or even a whole image of the floating-point environment
- Special result - a default result generated by a masked pre-computation exception

Correspondingly we define result subfunctions for each type. For $res(f)$ we apply the following case distinction

$$res(f) = \begin{cases} calcres(f) & : calc(f) \wedge \overline{spec(f)} \\ (signres(f), \epsilon) & : sign(f) \wedge \overline{spec(f)} \\ (cmpres(f), \epsilon) & : intcmp(f) \wedge \overline{spec(f)} \\ (constres(f), \epsilon) & : const(f) \wedge \overline{spec(f)} \\ (cvtres(f), \epsilon) & : store(f) \wedge \overline{spec(f)} \\ (loadres(f), \epsilon) & : (load(f) \vee FCMOV(f)) \wedge \overline{spec(f)} \\ (ctrlres(f), \epsilon) & : ctrl(f) \wedge \overline{spec(f)} \\ specres(f) & : spec(f) \\ (\epsilon, \epsilon) & : otherwise \end{cases}$$

Special results can occur for every result type, thus we need a predicate $spec(f)$ to distinguish whether the result must be computed in the ordinary way or whether the special result is returned instead.

$$spec(f) = (mexcp(f) \wedge prexcp(f) \neq \text{DEN})$$

In the following each result function is discussed separately.

Calculation Results

At first we care about the main purpose of the FPU: floating-point calculations. Luckily we already defined the semantics of the several floating-point operations. All we have left to do, is integrating this general description into our model. The calculation result consists of one or two double-extended precision floating-point number bit strings.

$$calcres : FPU \rightarrow (\mathbb{B}^{80}, \mathbb{B}^{80} | \epsilon)$$

However we base these numbers on the infinitely precise results $pres_1(f)$ and $pres_2(f)$. They can be obtained using the $fpuop$ function for the current FPU function and operands.

$$\begin{aligned} pres : FPU &\rightarrow (\mathbb{R}_\infty | NaN | \epsilon, \mathbb{R}_\infty | NaN | \epsilon) \\ pres(f) &= (pres_1(f), pres_2(f)) \\ &= \begin{cases} fpuop(fpu f(f), op1(f), op2(f), rm(f)) & : calc(f) \\ \llbracket op1(f) \rrbracket & : store(f) \end{cases} \end{aligned}$$

This precise result is also relevant for detecting post-computation exceptions like INX. Exception detection for format conversion operations benefits from this definition, too.

Naturally hardware can not store infinitely precise numbers, therefore the result must be rounded and transformed into a double-extended floating-point number bit string. To this end we define

the following function.

$$\begin{aligned}
\text{packres} : (FPU, \mathbb{R}_\infty |NaN| \epsilon) &\rightarrow \mathbb{B}^{80} | \epsilon \\
\text{packres}(f, pres) &= \begin{cases} x[62] \circ 1^{15} \circ 11 \circ x[61 : 0] & : \exists x \in \mathbb{B}^{63} : pres = QNaN \ x \\ x[62] \circ 1^{15} \circ 10 \circ x[61 : 0] & : \exists x \in \mathbb{B}^{63} : pres = SNaN \ x \\ \epsilon & : pres = \epsilon \\ \text{pack}(\eta(r_f(pres)), 80) & : \text{otherwise} \end{cases}
\end{aligned}$$

It regards NaNs and uses the *pack* function from 5.3. Observe that although all numbers are packed into an 80-bit string, results can be rounded using smaller precisions via the r_f function¹. Finally we can state the definition of the calculation result for $\text{calc}(f) = 1$.

$$\text{calcres}(f) = (\text{packres}(f, pres_1(f)), \text{packres}(f, pres_2(f)))$$

Sign Operation Results

The effect of sign operations is trivial. Depending on the instruction the sign of a floating-point register content is either switched or cleared to zero. For $\text{sign}(f) = 1$ we have:

$$\begin{aligned}
\text{signres} : FPU &\rightarrow \mathbb{B}^{80} \\
\text{signres}(f) &= \begin{cases} 0 \circ \text{op1}(f)[78 : 0] & : FABS(f) \\ \overline{s(\text{op1}(f))} \circ \text{op1}(f)[78 : 0] & : FCHS(f) \end{cases}
\end{aligned}$$

Comparison Results

In most cases floating-point comparisons do not produce a result in the sense of a bit string that ought to be saved to some register oder memory destination. However there are four x87 instructions² which update the the lower 16-bit part of the *RFLAGS* register with the resulting flags of a floating-point comparison. With the flag computation functions and *chflg* the resulting bit string for $\text{cmp}(f) = 1$ is easily defined.

$$\begin{aligned}
\text{cmpres} : FPU &\rightarrow \mathbb{B}^{16} \\
\text{cmpres}(f) &= \text{chflg}(f, \text{compzf}(f), \text{comppf}(f), \text{compcf}(f))
\end{aligned}$$

¹This means on the other hand that we always use 15 bits to represent the exponent of a floating-point number. Thus the results of x87 operations may deviate from implementations that stick to the IEEE standard more strictly.

²FComI, FComP, FUCOMI and FUCOMIP

Constant Results

The FPU offers seven different constants which can be pushed on the stack. For $const(f) = 1$ the chosen constant is denoted by $constant(f)$.

$$constant(f) = \begin{cases} 0.0 & : FLDZ(f) \\ 1.0 & : FLD1(f) \\ \pi & : FLDPI(f) \\ \log_2 10 & : FLDDL2T(f) \\ \log_2 e & : FLDDL2E(f) \\ \log_{10} 2 & : FLDLG2(f) \\ \ln 2 & : FLDLN2(f) \end{cases}$$

This value is rounded and packed to 80-bit double-extended format.

$$\begin{aligned} constres : FPU &\rightarrow \mathbb{B}^{80} \\ constres(f) &= pack(\eta(r_f(constant(f))), 80) \end{aligned}$$

Conversion Results

Inside the FPU all computations are done in double-extended floating-point format. Nevertheless there are store instructions to export numbers to deviating precisions and formats. We already introduced this fact along with the corresponding conversion functions. Thus the conversion result is a simple application of these utilities.

$$\begin{aligned} cvtres : FPU &\rightarrow \mathbb{B}^* \\ cvtres(f) &= \begin{cases} twoc_{resize(f)}(rndint(\llbracket op1(f) \rrbracket, rm(f))) & : cvtint(f) \\ bcd_{80}(rndint(\llbracket op1(f) \rrbracket, rm(f))) & : FBSTP(f) \\ prc(op1(f), rm(f), resize(f)) & : cvtprc(f) \\ op1(f) & : otherwise \end{cases} \end{aligned}$$

Note that for BCD and integer conversion the floating-point numbers are first rounded to integers and then transformed into the right representation.

Load Results

As stated before, load results comprise the results from load instructions and conditional moves. For loads actually the result is nothing but the first operand containing the converted memory operand. In the latter case a value is only moved, if a certain condition holds. We denote this condition by $cmove(f)$.

$$cmove : FPU \rightarrow \mathbb{B}$$

Using this predicate the conditional move may be emulated by switching the actual source with the destination content. This assures that the destination's value is not altered in case of a missed condition.

$$\begin{aligned} loadres : FPU &\rightarrow \mathbb{B}^* \\ loadres(f) &= \begin{cases} st(f, trgt_1(f)) & : FCMOV(f) \wedge \overline{cmove(f)} \\ op1(f) & : \text{otherwise} \end{cases} \end{aligned}$$

Now we have to define $cmove(f)$. The condition is determined with regard to the current conditional move instruction and the values of zero, carry and parity flags in the *RFLAGS* register. For $FCMOV(f)$ we yield:

$$cmove(f) = \begin{cases} cf(f) & : cond(f) = B \\ cf(f) \vee zf(f) & : cond(f) = BE \\ zf(f) & : cond(f) = E \\ \overline{cf(f)} & : cond(f) = NB \\ \overline{cf(f) \vee zf(f)} & : cond(f) = BE \\ \overline{zf(f)} & : cond(f) = E \\ pf(f) & : cond(f) = U \\ \overline{pf(f)} & : cond(f) = NU \end{cases}$$

The respective flags may have been set by preceding *FCOMI*, *FCOMIP* or *FUCOMI* as well as *FUCOMIP* instructions.

Control Results

Several control instructions may save FPU configuration data to memory to enable restoring the FPU state at a later time. Thus control results are FPU components or image bit strings of different length.

$$ctrlres : FPU \rightarrow \mathbb{B}^* | \epsilon$$

There are three different image types:

- x87 Environment - contains only the floating-point environment registers but not the content of the FPU stack
- x87/MMX State - a complete image of the x87 environment and data register contents - As the FPU registers are shared by x87 and MMX media instructions this image also saves the MMX state.
- XMM/x87/Media state - contains the complete XMM and x87/MMX state

For $imgsave(f) = 1$ the image to be saved is denoted by $image(f)$. Its definition reflects the image type separation above.

$$\begin{aligned}
image : FPU &\rightarrow \mathbb{B}^* \\
image(f) &= \begin{cases} x87env(f) & : FNSTENV(f) \\ x87img(f) & : FNSAVE(f) \\ xmmimg(f) & : FXSAVE(f) \end{cases}
\end{aligned}$$

The figures in Appendix C visualize the various image formats. In the following we will describe mathematically how these images are composed. It is coherent that their structure depends on the current mode and data width of the CPU. We begin with defining the complete 94/108-byte x87 configuration image.

$$\begin{aligned}
x87img : FPU &\rightarrow \mathbb{B}^* \\
x87img(f) &= \begin{cases} st(f, 7) \circ \dots \circ st(f, 0) \circ 0^{16} \circ f.dp[47 : 0] \circ 0^5 \circ f.opc & : prot32(f) \\ \circ f.ip[47 : 0] \circ 0^{16} \circ f.ftw \circ 0^{16} \circ f.fsw \circ 0^{16} \circ f.fcw & \\ st(f, 7) \circ \dots \circ st(f, 0) \circ 0^4 \circ f.dp[31 : 16] \circ 0^{28} & : real32(f) \\ \circ f.dp[15 : 0] \circ 0^4 \circ f.ip[31 : 16] \circ 0 \circ f.opc \circ 0^{16} & \\ \circ f.ftw \circ 0^{16} \circ f.fsw \circ 0^{16} \circ f.fcw & \\ st(f, 7) \circ \dots \circ st(f, 0) \circ f.dp[31 : 0] \circ f.ip[31 : 0] & : prot16(f) \\ \circ f.ftw \circ f.fsw \circ f.fcw & \\ st(f, 7) \circ \dots \circ st(f, 0) \circ f.dp[19 : 16] \circ 0^{12} \circ f.dp[15 : 0] & : real16(f) \\ \circ f.ip[19 : 16] \circ 0 \circ f.opc \circ f.ftw \circ f.fsw \circ f.fcw & \end{cases}
\end{aligned}$$

It contains the eight FPU stack registers followed by the exception pointers and the x87 status, control and tag word. Depending on the mode varying portions of the data and instruction pointer fields are saved in the image. This affects also the order and position of the other components. The 14/28-byte x87 environment is then a part of the complete image.

$$\begin{aligned}
x87env : FPU &\rightarrow \mathbb{B}^* \\
x87env(f) &= \begin{cases} x87img(f)[223 : 0] & : prot32(f) \vee real32(f) \\ x87img(f)[111 : 0] & : otherwise \end{cases}
\end{aligned}$$

The XMM/x87/Media state image is the biggest one. It spans 512 bytes, while most of the space is needed for the 16 128-bit XMM register contents. This XMM information is not available to the FPU, thus only the x87 part of the image is returned to the CPU, while there are routines in the interface which insert the missing data and compile the full image. These routines are defined in section 3.4.

$$\begin{aligned}
xmmimg : FPU &\rightarrow \mathbb{B}^{1280} \\
xmmimg(f) &= \begin{cases} zeroext_{128}(st(f, 7)) \circ \dots \circ zeroext_{128}(st(f, 0)) \circ 0^{64} & : es(f) = 1 \\ \circ f.dp \circ f.ip \circ 0^5 \circ f.opc \circ 0^{16} \circ ftw8(f) \circ f.fsw \circ f.fcw & \\ zeroext_{128}(st(f, 7)) \circ \dots \circ zeroext_{128}(st(f, 0)) \circ 0^{192} & : otherwise \\ \circ 0^5 \circ f.opc \circ 0^{16} \circ ftw8(f) \circ f.fsw \circ f.fcw & \end{cases}
\end{aligned}$$

As a speciality *FXSAVE* only stores the FPU exception data and instruction pointers when an unmasked exception is pending. Also it only saves eight bits for the x87 tag word. These eight bits just signal whether the corresponding stack registers are full or empty. They are computed in the following way.

$$\begin{aligned} ftw8 : FPU &\rightarrow \mathbb{B}^{16} \\ ftw8(f) &= (f.ftw[15] \bar{\wedge} f.ftw[14]) \circ (f.ftw[13] \bar{\wedge} f.ftw[12]) \circ (f.ftw[11] \bar{\wedge} f.ftw[10]) \circ \\ &\quad (f.ftw[9] \bar{\wedge} f.ftw[8]) \circ (f.ftw[7] \bar{\wedge} f.ftw[6]) \circ (f.ftw[5] \bar{\wedge} f.ftw[4]) \circ \\ &\quad (f.ftw[3] \bar{\wedge} f.ftw[2]) \circ (f.ftw[1] \bar{\wedge} f.ftw[0]) \circ \end{aligned}$$

Thus a 1 signals an empty register while a 0 signals some valid data in the corresponding stack register.

At last we incorporate the images into our control result function. Besides images also the x87 status and control word may be stored to memory.

$$ctrlres(f) = \begin{cases} image(f) & : \quad imgsav(f) \\ fsw(f) & : \quad FNSTSW(f) \\ fcw(f) & : \quad FNSTCW(f) \\ \epsilon & : \quad \text{otherwise} \end{cases}$$

Special Results

Finally only the special results are left. They play an exceptional role among the different result types as they embody not really another self-contained type of results. Instead they represent the default results for all instruction and result types in the event of masked exceptions.

$$specres : FPU \rightarrow (\mathbb{B}^*|\epsilon, \mathbb{B}^*|\epsilon)$$

In section 6.3 we did define the masked exception results by the *mexcpres* function for mathematical operations. However exceptions may occur for all kind of instructions so we must investigate further. The most common default response to masked exceptions - especially masked INV exceptions - are indefinite values. We define the cases in which indefinite values are returned - apart from the cases of mathematical functions - with the following predicate.

$$\begin{aligned} indefinite : FPU &\rightarrow \mathbb{B} \\ indefinite(f) = 1 &\Leftrightarrow ((prexcp(f) = STK) \vee ((prexcp(f) = INV \vee QNaNs(f)) \wedge \\ &\quad (FIST(f) \vee FISTP(f) \vee FISTTP(f) \vee FBSTP(f))) \vee \\ &\quad (prexcp(f) = INV \wedge unsup(op1(f)) \wedge (FST(f) \vee FSTP(f)))) = 1 \end{aligned}$$

In these cases one of the seven predefined indefinite values is stored as a result. Which one of them is chosen depends on the instruction and the destination size. We denote the indefinite result by

$indefres(f)$.

$$indefres : FPU \rightarrow \mathbb{B}^*$$

$$indefres(f) = \begin{cases} \text{indefbcd} & : FBSTP(f) \\ \text{indefint16} & : cvtint(f) \wedge ressize(f) = 16 \\ \text{indefint32} & : cvtint(f) \wedge ressize(f) = 32 \\ \text{indefint64} & : cvtint(f) \wedge ressize(f) = 64 \\ \text{indeffp32} & : cvtprc(f) \wedge ressize(f) = 32 \\ \text{indeffp64} & : cvtprc(f) \wedge ressize(f) = 64 \\ \text{indeffp80} & : \text{otherwise} \end{cases}$$

Furthermore there may be NaN results as a default response to masked invalid operation exceptions. Most of the cases are already covered by $fpuop$, nevertheless there is another case for precision converting functions that deserves some attention. When a NaN operand shall be stored to some memory destination it may be necessary to convert the NaN to a lower precision format. To this end the floating-point precision conversion function can be used. Note that SNaNs are not converted to QNaNs in this case.

Another exception which requires special treatment is when unordered operands are used in a comparison and the *RFLAGS* register shall be updated. The corresponding circumstances are detected by the predicate $unordexcp(f)$.

$$unordexcp : FPU \rightarrow \mathbb{B}$$

$$unordexcp(f) = 1 \Leftrightarrow \overline{(indefinite(f) \wedge (SNaNs(f) \wedge (FUCOMI(f) \vee FUCOMIP(f)) \vee NaNs(f) \wedge (FCOMI(f) \vee FCOMIP(f))))} = 1$$

In these cases the respective *RFLAGS* bits must be set to unordered state.

Considering all these facts we define the special result function.

$$specres(f) = \begin{cases} (indefres(f), \epsilon) & : indefinite(f) \\ (prc(op1(f), rm(f), ressize(f)), \epsilon) & : \overline{indefinite(f)} \wedge prexcp(f) = INV \wedge (FST(f) \vee FSTP(f)) \wedge NaN(op1(f)) \\ (chflg(f, 1, 1, 1), \epsilon) & : unordexcp(f) \\ calcrec(f) & : calc(f) \wedge prexcp(f) = INV \wedge SNaNs(f) \\ mexcpres(fpu(f), prexcp(f), op1(f), op2(f)) & : \text{otherwise} \end{cases}$$

This completes our general result definition as well.

7.2.4 Comparison, Stack Management and Control Instructions

Apart from the instructions which deliver a direct result there are also instructions which affect the x87 environment. These are floating-point comparisons, stack and control instructions. To grasp these effects we define another transition subfunction called $set(f)$.

$$set : FPU \rightarrow FPU$$

It manages all the changes applied to the FPU components due to the aforementioned instruction types. Anyway this function results in a new configuration $f' = \text{set}(f)$ and in the following we will define the new values of the particular components according to the several instructions we have to consider. Before we can define these components we should have a look at image operands as these contain the new values for our components in case of restore instructions.

Image Components

As with image saving functions also at FPU restoration we are confronted with the multitude of image formats and instructions. Therefore we define functions that deliver the desired FPU component from whatever image format in case of image restoration instructions $\text{imgload}(f) \vee \text{FXRSTOR}(f)$. Confer Appendix C for illustrations of the various image formats.

$$\begin{aligned} \text{imgfsw}(f) &= \begin{cases} \text{mop}(f)[47 : 32] & : \text{imgload}(f) \wedge (\text{prot32}(f) \vee \text{real32}(f)) \\ \text{mop}(f)[31 : 16] & : \text{imgload}(f) \wedge (\text{prot16}(f) \vee \text{real16}(f)) \vee \text{FXRSTOR}(f) \end{cases} \\ \text{imgfcw}(f) &= \text{mop}(f)[15 : 0] \\ \text{imgftw}(f) &= \begin{cases} \text{mop}(f)[79 : 64] & : \text{imgload}(f) \wedge (\text{prot32}(f) \vee \text{real32}(f)) \\ \text{mop}(f)[47 : 32] & : \text{imgload}(f) \wedge (\text{prot16}(f) \vee \text{real16}(f)) \\ \text{mop}(f)[39 : 32] & : \text{FXRSTOR}(f) \end{cases} \end{aligned}$$

The functions for control, status and tag word are quite comprehensible. For the x87 tag word at FXRSTOR only eight bits are saved. That is why imgftw has type $\text{FPU} \rightarrow \mathbb{B}^*$ they have to be extended to the two bit tags later. Moreover the images contain the x87 exception pointers.

$$\begin{aligned} \text{imgdp}(f) &= \begin{cases} 0^{16} \circ \text{mop}(f)[207 : 160] & : \text{imgload}(f) \wedge \text{prot32}(f) \\ 0^{32} \circ \text{mop}(f)[219 : 204] \circ \text{mop}(f)[175 : 160] & : \text{imgload}(f) \wedge \text{real32}(f) \\ 0^{32} \circ \text{mop}(f)[111 : 80] & : \text{imgload}(f) \wedge \text{prot16}(f) \\ 0^{44} \circ \text{mop}(f)[111 : 108] \circ \text{mop}(f)[95 : 80] & : \text{imgload}(f) \wedge \text{real32}(f) \\ \text{mop}(f)[191 : 128] & : \text{FXRSTOR}(f) \end{cases} \\ \text{imgip}(f) &= \begin{cases} 0^{16} \circ \text{mop}(f)[143 : 96] & : \text{imgload}(f) \wedge \text{prot32}(f) \\ 0^{32} \circ \text{mop}(f)[155 : 140] \circ \text{mop}(f)[111 : 96] & : \text{imgload}(f) \wedge \text{real32}(f) \\ 0^{32} \circ \text{mop}(f)[79 : 48] & : \text{imgload}(f) \wedge \text{prot16}(f) \\ 0^{44} \circ \text{mop}(f)[79 : 76] \circ \text{mop}(f)[63 : 48] & : \text{imgload}(f) \wedge \text{real32}(f) \\ \text{mop}(f)[127 : 64] & : \text{FXRSTOR}(f) \end{cases} \\ \text{imgopc}(f) &= \begin{cases} \text{mop}(f)[154 : 144] & : \text{imgload}(f) \wedge \text{prot32}(f) \\ \text{mop}(f)[138 : 128] & : \text{imgload}(f) \wedge \text{real32}(f) \\ \text{undefined} & : \text{imgload}(f) \wedge \text{prot16}(f) \\ \text{mop}(f)[74 : 64] & : \text{imgload}(f) \wedge \text{real32}(f) \\ \text{mop}(f)[58 : 48] & : \text{FXRSTOR}(f) \end{cases} \end{aligned}$$

Note that the last non-control instruction opcode is not contained in the x87 image for 16-bit protected mode.

The last image component relevant for the FPU is the stack register contents. They are not included in the x87 environment. Thus $imgstk : FPU \rightarrow \mathbb{B}^{640}$ is only defined for *FRSTOR* or *FXRSTOR* instruction.

$$\begin{aligned} imgstk(f) &= \begin{cases} mop(f)[863 : 224] & : FRSTOR(f) \wedge (prot32(f) \vee real32(f)) \\ mop(f)[767 : 128] & : FRSTOR(f) \wedge (prot16(f) \vee real16(f)) \\ xmmimgstk(f) & : FXRSTOR(f) \end{cases} \\ xmmimgstk(f) &= mop(f)[1231 : 1152] \circ mop(f)[1103 : 1024] \circ mop(f)[975 : 896] \circ \\ &\quad mop(f)[847 : 768] \circ mop(f)[719 : 640] \circ mop(f)[591 : 512] \circ \\ &\quad mop(f)[463 : 384] \circ mop(f)[335 : 256] \end{aligned}$$

Effects on the Configuration

Now we have all the ingredients to define the configuration which results from $set(f)$. We start with the stack registers. They are only affected by state restoring instructions.

$$\begin{aligned} \forall j \in \mathbb{N}_8 : \\ f'.fpr(sri(f, j)) &= \begin{cases} imgstk(f)[80 \cdot j + 79 : 80 \cdot j] & : FRSTOR(f) \vee FXRSTOR(f) \\ f.fpr(sri(f, j)) & : \text{otherwise} \end{cases} \end{aligned}$$

Also the x87 exception pointers an opcode fields are changed during execution phase by instructions that load the FPU environment from memory. In addition they are also cleared to zero in case of instructions that reinitialize the FPU configuration. For *FXRSTOR* instructions these components are only restored when there are pending unmasked exceptions with respect to the restored status word content.

$$\begin{aligned} f'.dp &= \begin{cases} imgdp(f) & : imgload(f) \vee (FXRSTOR(f) \wedge imgfsw(f)[7]) \\ 0^{64} & : FNINIT(f) \vee FNSAVE(f) \\ f.dp & : \text{otherwise} \end{cases} \\ f'.ip &= \begin{cases} imgip(f) & : imgload(f) \vee (FXRSTOR(f) \wedge imgfsw(f)[7]) \\ 0^{64} & : FNINIT(f) \vee FNSAVE(f) \\ f.ip & : \text{otherwise} \end{cases} \\ f'.opc &= \begin{cases} imgopc(f) & : imgload(f) \vee (FXRSTOR(f) \wedge imgfsw(f)[7]) \\ 0^{11} & : FNINIT(f) \vee FNSAVE(f) \\ f.opc & : \text{otherwise} \end{cases} \end{aligned}$$

Keep in mind that the x87 exception information is regularly updated in the pre-computation phase. Thus the values for the current instruction do not appear in this case distinction.

The x87 tag word may be updated by state restoring, initializing and *FFREE* instructions. In the first case for *FXRSTOR* the 2-bit tags must be restored from the image 1-bit tags³ examining the

³In the XMM/x87/media images only a eight-bit version of the tag word is saved, where for each register a 0 marks empty registers and a 1 marks registers that are occupied by a value. That is why we must examine the particular register contents to determine the corresponding 2-bit tag. For state restoring instructions other than *FXRSTOR* we can immediately copy the tag values from the image.

contents of the stack register fields in the image. Therefore we define the new tag word content separately for each register. In the case of initialization all tags are marked empty. In the case of *FFREE* only one register is tagged empty. Let $j \in \mathbb{N}_8$ be a floating-point register index and $i = ((j - \langle \text{imgfsw}(f)[13 : 11] \rangle) \bmod 8)$ be the corresponding stack element index, then:

$$f'.ftw[2j+1:2j] = \begin{cases} \text{imgftw}(f)[2j+1:2j] & : \text{imgload}(f) \\ 11 & : \text{FNINIT}(f) \vee \text{FNSAVE}(f) \vee \\ & \text{FXRSTOR}(f) \wedge \overline{\text{imgftw}(f)[j]} \vee \\ & \text{FFREE}(f) \wedge j = \langle \text{opc}(f)[2:0] \rangle \\ 01 & : \text{FXRSTOR}(f) \wedge \text{imgftw}(f)[j] \wedge \\ & \llbracket \text{imgstk}(f)[80i+79:80i] \rrbracket = 0 \\ 10 & : \text{FXRSTOR}(f) \wedge \text{imgftw}(f)[j] \wedge \\ & (\text{NaN}(\text{imgstk}(f)[80i+79:80i]) \vee \\ & \text{unsup}(\text{imgstk}(f)[80i+79:80i]) \vee \\ & \text{dnrml}(\text{imgstk}(f)[80i+79:80i]) \vee \\ & \text{infty}(\text{imgstk}(f)[80i+79:80i])) \\ f.ftw & : \overline{\text{imgload}(f)} \wedge \overline{\text{FXRSTOR}(f)} \wedge \\ & \overline{\text{FNSAVE}(f)} \wedge \overline{\text{FNINIT}(f)} \\ 00 & : \text{otherwise} \end{cases}$$

The else case represents the situation that the respective stack register contains a normal non-zero floating-point number and *FXRSTOR* is executed.

Next we define the changes to the x87 control word evoked during the execution phase of the FPU state transition. It is affected by image loading or initializing operations as well as the *FLDCW* and the *FNSTENV* instructions.

$$f'.fcw = \begin{cases} \text{imgfcw}(f) & : \text{imgload}(f) \vee \text{FXRSTOR}(f) \\ \text{mop}(f) & : \text{FLDCW}(f) \\ 0x037F & : \text{FNINIT}(f) \vee \text{FNSAVE}(f) \\ f.fcw[15:7] \circ 1^7 & : \text{FNSTENV}(f) \\ f.fcw & : \text{otherwise} \end{cases}$$

At initialization all exceptions are masked, precision control is set to double-extended precision and rounding mode to nearest-even rounding. *FNSTENV* set all exception mask bits to one as well⁴. At last only the x87 status word is left to be considered.

$$f'.fsw = \begin{cases} \text{imgfsw}(f) & : \text{imgload}(f) \vee \text{FXRSTOR}(f) \\ 0^{16} & : \text{FNINIT}(f) \vee \text{FNSAVE}(f) \\ 0 \circ X \circ \text{top}(f) \circ XXX \circ 0^8 & : \text{FNCLEX}(f) \\ f.fsw[15] \circ \text{cmpcc}(f)[3] \circ \text{stktop}(f) \circ \\ \text{cmpcc}(f)[2:0] \circ f.fsw[7:0] & : \text{otherwise} \end{cases}$$

⁴This is useful for implementing floating-point exception handlers as this instruction saves the FPU environment to the memory for further examination and at the same time masks all further x87 exceptions which may interrupt the exception handling routine.

In the event of an FNCLEX instruction the exception flags are cleared to zero. The stack-top pointer stays unchanged while the condition code bits are undefined. Furthermore besides control operations, comparison and stack management instructions can affect the condition code bits or the stack-top pointer. We introduce these effects by $cmpcc(f)$ and $stktop(f)$.

$$\begin{aligned} stktop : FPU &\rightarrow \mathbb{B}^3 \\ stktop(f) &= \begin{cases} top(f) -_3 001 & : FDECSTP(f) \\ top(f) +_3 001 & : FINCSTP(f) \\ top(f) & : \text{otherwise} \end{cases} \end{aligned}$$

The stack management functions solely increment or decrement the stack-top pointer. For the new condition code bits we must distinguish between comparison instructions that affect $cc(f)$ and the FXAM instruction.

$$cmpcc : FPU \rightarrow \mathbb{B}^4$$

$$cmpcc(f) = \begin{cases} compzf(f) \circ comppf(f) \circ 0 \circ compcf(f) & : cmp(f) \wedge \overline{FXAM(f)} \wedge \overline{xupdf(I(f))} \wedge \overline{stkunf(f)} \\ 1101 & : cmp(f) \wedge \overline{FXAM(f)} \wedge \overline{xupdf(I(f))} \wedge stkunf(f) \\ xam(op1(f), f) & : FXAM(f) \\ cc(f) & : \text{otherwise} \end{cases}$$

Either the condition code bits are associated with their corresponding flag values or they contain a special code for the type of the first operand according to the definition of $xam(x, f)$. This represents the type of a 80-bit floating-point number x and is defined as follows below. In case of a masked stack underflow the unordered condition code 1101 is returned by $cmpcc(f)$.

$$xam : (\mathbb{B}^{80}, FPU) \rightarrow \mathbb{B}^4$$

$$xam(x, f) = \begin{cases} 00 \circ s(x) \circ 0 & : unsup(x) \wedge \overline{empty(f, 0)} \\ 00 \circ s(x) \circ 1 & : NaN(x) \wedge \overline{empty(f, 0)} \\ 01 \circ s(x) \circ 1 & : infty(x) \wedge \overline{empty(f, 0)} \\ 10 \circ s(x) \circ 0 & : \llbracket x \rrbracket = 0 \wedge \overline{empty(f, 0)} \\ 11 \circ s(x) \circ 0 & : dnrml(x) \wedge \overline{empty(f, 0)} \\ 10 \circ s(x) \circ 1 & : empty(f, 0) \\ 01 \circ s(x) \circ 0 & : \text{otherwise} \end{cases}$$

This function completes the semantics of control, stack management and comparison instructions. The sign of the value in the stack-top register is assigned to condition code bit $cc(f')[1]$. In the else case the stack-top register is not empty and it contains a non-zero normal floating-point number.

Finally we finished the core piece of the FPU. In the execution phase the results and main effects of all instructions are computed and adopted. The following phases consider the consequences these results imply. There may be additional changes due to exceptions and other side effects to care about.

7.3 x87 Post-Computation Exceptions Checks

Apart from pre-computation exceptions there may also be exceptions based on the rounded result of the computation. These are called post-computation exceptions, as they may only be discovered after the computation routine. We already introduced the function $\sigma_{fpu}^{postchk}(f) : FPU \rightarrow FPU$ which shall serve to check for and apply all the effects of post-computation exceptions to the post-computation FPU configuration. Before we give the exact definition of this function we must supply means to detect these exceptions.

7.3.1 Detecting Post-Computation Exceptions

Basically there are three types of post-computation exceptions:

- OVF - computation caused an *overflow* exception
- UNF - computation caused an *underflow* exception
- INX - computation caused a precision exception (*inexact* result)

As there cannot occur an overflow and an underflow simultaneously, only two post-computation exceptions may be caused by a single instruction at a time. Also only calculating and certain precision converting store instructions may produce such exceptions. Thus we choose a more simple and straight forward approach than for the more complex pre-computation exceptions. We just define the following three predicates to determine whether a computation caused the corresponding exception. Note that these definitions are only valid for the case that $(calc(f) \vee store(f)) = 1$ and $prexcp(f) \in \{\epsilon, DEN\} \wedge \overline{es(f)}$, that means that there must have occurred no exceptions but masked denormalized operand exceptions. In any other case there is no definite result that could result in a post-computation exception.

$$\begin{aligned}
ovf(f) = 1 &\Leftrightarrow (|pres_1(f)| \neq \infty) \wedge (\llbracket res_1(f) \rrbracket > X_{max}(f)) \wedge \overline{(cvtint(f) \vee FBSTP(f))} \\
unf(f) = 1 &\Leftrightarrow dnrml(res_1(f) \wedge (cvtint(f) \vee FBSTP(f))) \vee \\
&\quad dnrml(res_2(f)) \wedge FSINCOS(f) \\
inx(f) = 1 &\Leftrightarrow (\llbracket res_1(f) \rrbracket \neq pres_1(f)) \vee FSINCOS(f) \wedge (\llbracket res_2(f) \rrbracket \neq pres_2(f))
\end{aligned}$$

Instruction FSINCOS demands a special case treatment as it is the only instruction that produces two floating-point results which might be underflowing or inexact. Note that it is not considered as an underflow when a very tiny exact result is rounded to zero. This just results in an INX exception.

7.3.2 Impact on the FPU Environment

Similar to pre-computation exceptions also post-computation exceptions affect the floating-point unit environment. To cover these changes we define the function $postxcp(f)$, which returns the altered FPU configuration.

$$\begin{aligned}
postxcp : FPU &\rightarrow FPU \\
postxcp(f) &= f'
\end{aligned}$$

We describe the altered configuration f' separately for each component. At first we have a look at the post-computation exception flags for overflow, underflow and precision exceptions. Once again we only consider cases where $(\text{calc}(f) \vee \text{store}(f)) = 1$ and $\text{prexcp}(f) \in \{\epsilon, \text{DEN}\} \wedge \text{es}(f)$ holds.

$$\begin{aligned} \text{oe}(f') &= \begin{cases} 1 & : \text{ovf}(f) \\ \text{oe}(f) & : \text{otherwise} \end{cases} \\ \text{ue}(f') &= \begin{cases} 1 & : \text{unf}(f) \wedge (\text{inx}(f) \wedge \text{um}(f) \vee \overline{\text{um}(f)}) \\ \text{ue}(f) & : \text{otherwise} \end{cases} \\ \text{pe}(f') &= \begin{cases} 1 & : \text{inx}(f) \wedge \overline{(\text{xupdm}(f) \wedge (\text{ovf}(f) \wedge \overline{\text{om}(f)} \vee \text{unf}(f) \wedge \overline{\text{um}(f)}))} \\ \text{pe}(f) & : \text{otherwise} \end{cases} \end{aligned}$$

Observe also here how masked exception flags are accumulated by assigning the old flag values. The underflow flag is only set when the underflow exception is unmasked or when the result is also inexact. Precision exceptions are only raised when they were not provoked by instructions that generated unmasked overflows or underflow and update memory.

The condition code bits in the status word are affected by post-computation exceptions as well. Whenever a result is inexact $\text{cc}(f')[1]$ will contain information, whether the result was rounded upwards or downwards. For this purpose we define a helper function which compares rounded and exact results x and y respectively.

$$\begin{aligned} \text{roundup?} : (\mathbb{R}_\infty, \mathbb{R}_\infty) &\rightarrow \mathbb{B} \\ \text{roundup?}(x, y) &= \begin{cases} 1 & : |x| > |y| \\ 0 & : \text{otherwise} \end{cases} \end{aligned}$$

Note that this definition does not comply with the round up mode we already introduced. Instead of checking whether a number was rounded towards positive infinity, the function tests if the rounding increased the number's magnitude.

Besides rounding we may also need to consider the three least significant bits of the quotient $q : FPU \rightarrow \mathbb{B}^3$ and exponent difference $\text{Diff} : FPU \rightarrow \mathbb{Z}$ for partial remainder instructions.

$$\begin{aligned} q(f) &= \text{bin}_3(|\text{rndint}(\lfloor \frac{\text{op1}(f)}{\text{op2}(f)} \rfloor, \text{rm}(f))| \bmod 8) \\ \text{Diff}(f) &= \langle e(\text{op1}(f)) \rangle_{\text{bias}} - \langle e(\text{op2}(f)) \rangle_{\text{bias}} \end{aligned}$$

Now the new value of the condition code flag after applying $\text{postxcp}(f)$ will be:

$$\text{cc}(f')[1] = \begin{cases} \text{roundup?}(\lfloor \text{res}_1(f) \rfloor, \text{pres}_1(f)) & : \text{inx}(f) \wedge \overline{\text{ovf}(f)} \wedge \overline{\text{unf}(f)} \wedge \\ & (\text{calc}(f) \wedge \overline{\text{prem}(f)} \vee \text{store}(f)) \\ \text{roundup?}(\lfloor \text{adjres}_1(f) \rfloor, \lfloor \text{res}_1(f) \rfloor) & : \overline{\text{xupdm}(f)} \wedge \overline{\text{prem}(f)} \wedge \\ & (\text{ovf}(f) \wedge \overline{\text{om}(f)} \vee \text{unf}(f) \wedge \overline{\text{um}(f)}) \\ \text{cc}(f)[1] & : \overline{\text{inx}(f)} \wedge \overline{\text{ovf}(f)} \wedge \overline{\text{unf}(f)} \wedge \\ & (\text{FST}(f) \vee \text{FSTP}(f)) \\ q(f)[0] & : \text{prem}(f) \wedge (0 \leq \text{Diff}(f) < 64) \\ 0 & : \text{otherwise} \end{cases}$$

The function $adjres_1(f)$ returns an adjusted result for masked overflow or underflow exceptions. This is called *exponent wrapping* and will be defined in the next subsection. The other condition code bits are only affected by partial remainder instructions during post-computation exception check phase.

$$cc(f')[0] = \begin{cases} q(f)[2] & : \text{prem}(f) \wedge (0 \leq Diff(f) < 64) \\ 0 & : \text{prem}(f) \wedge (Diff(f) < 0 \vee Diff(f) \geq 64) \\ cc(f)[0] & : \text{otherwise} \end{cases}$$

$$cc(f')[2] = \begin{cases} 0 & : \text{prem}(f) \wedge (Diff(f) < 64) \\ 1 & : \text{prem}(f) \wedge (Diff(f) \geq 64) \\ cc(f)[2] & : \text{otherwise} \end{cases}$$

$$cc(f')[3] = \begin{cases} q(f)[1] & : \text{prem}(f) \wedge (0 \leq Diff(f) < 64) \\ 0 & : \text{prem}(f) \wedge (Diff(f) < 0 \vee Diff(f) \geq 64) \\ cc(f)[3] & : \text{otherwise} \end{cases}$$

For partial remainder instructions the condition code bits contain the last three bits of the quotient $q(f)$ in a quite confusing order. The quotient bits $q(f)[2], q(f)[1]$ and $q(f)[0]$ are assigned to the condition code bits $cc(f)[0], cc(f)[3]$ and $cc(f)[1]$.

Due to its definition in section 4.4, $es(f') = 1$ and $busy(f') = 1$ holds in case of unmasked post-computation exceptions. All remaining configuration components are kept unmodified by $postxcp(f)$.

$$\begin{array}{llll} f'.fpr & = & f.fpr & f'.opc & = & f.opc & f'.fsw[13:11] & = & f.fsw[13:11] \\ f'.dp & = & f.dp & f'.fcw & = & f.fcw & f'.fsw[6] & = & f.fsw[6] \\ f'.ip & = & f.ip & f'.ftw & = & f.ftw & f'.fsw[2:0] & = & f.fsw[2:0] \\ f'.acc & = & f.acc & f'.ans & = & f.ans & & & \end{array}$$

7.3.3 Adjusting the Result

As stated above, there may be cases where the rounded result of a computation is exchanged with an adjusted one. For unmasked overflows or underflows upon instructions which do not affect memory, the result is divided by or multiplied with a certain constant, thus yielding a new result which is inside the range of normal representable numbers. Basically this multiplication/division is only an addition to/subtraction from the exponent, i.e. the significant is kept unchanged and the exception handler later on can restore the original value of the result by reversing the constant multiplication/division. The new exponent is called *wrapped exponent* (cf. [MP00]). We establish

the adjusted results by the following functions.

$$\begin{aligned}
adjres_1(f) &= \begin{cases} pack(\eta(r_f(pres_1(f))/2^{24576}), 80) & : \quad ovf(f) \wedge \overline{om(f)} \wedge \overline{xupdm(f)} \\ pack(\eta(r_f(pres_1(f) \cdot 2^{24576}), 80) & : \quad unf(f) \wedge \overline{um(f)} \wedge \overline{xupdm(f)} \\ res_1(f) & : \quad otherwise \end{cases} \\
adjres_2(f) &= \begin{cases} pack(\eta(r_f(pres_2(f))/2^{24576}), 80) & : \quad ovf(f) \wedge \overline{om(f)} \wedge 2results(f) \\ pack(\eta(r_f(pres_2(f) \cdot 2^{24576}), 80) & : \quad unf(f) \wedge \overline{um(f)} \wedge 2results(f) \\ res_2(f) & : \quad otherwise \end{cases}
\end{aligned}$$

To save the adjusted result we also need to define an adjusted target, because pushes to the stack may have antiquated the old target index.

$$adjtgt(f) = \begin{cases} (tgt_1(f) + 1) \bmod 8 & : \quad push?(f) \\ tgt_1(f) & : \quad otherwise \end{cases}$$

7.3.4 Transition Function

At last we produce the definition of the post-computation exception checking transition function. We utilise our atomic stack functions to this end.

$$\sigma_{fpu}^{postchk}(f) = \begin{cases} \underbrace{save(adjres_1(f), adjtgt(f), postxcp(f))}_{f'} & : \quad (calc(f) \vee store(f)) \wedge \\ & : \quad (prexcp(f) \in \{\epsilon, \underline{DEN}\}) \wedge \\ & : \quad FSINCOS(f) \wedge es(f) \\ save(adjres_2(f), 0, f') & : \quad (prexcp(f) \in \{\epsilon, \underline{DEN}\}) \wedge \\ & : \quad FSINCOS(f) \wedge es(f) \\ f & : \quad otherwise \end{cases}$$

Again we have to consider the special case that FSINCOS could produce two underflowing results. Note that an adjusted result is only saved for unmasked overflows or underflows at mathematical instructions which store their results to a register destination. In the absence of post-computation exceptions or if there were pre-computation exceptions other than a masked DEN, no changes to the configuration are evoked by $\sigma_{fpu}^{postchk}(f)$.

7.4 Returning the x87 Result

Finally the computation of results is done and all checks for exceptions are complete. Now we must return the result to the CPU in case the instruction's destination is a memory operand. At last the floating-point stack is popped when necessary.

7.4.1 Storing a Result to Memory

Due to the interface between CPU and FPU which we already established, writing back a result to memory is easy. All we have to do is to update the field *ans* of the floating-point configuration

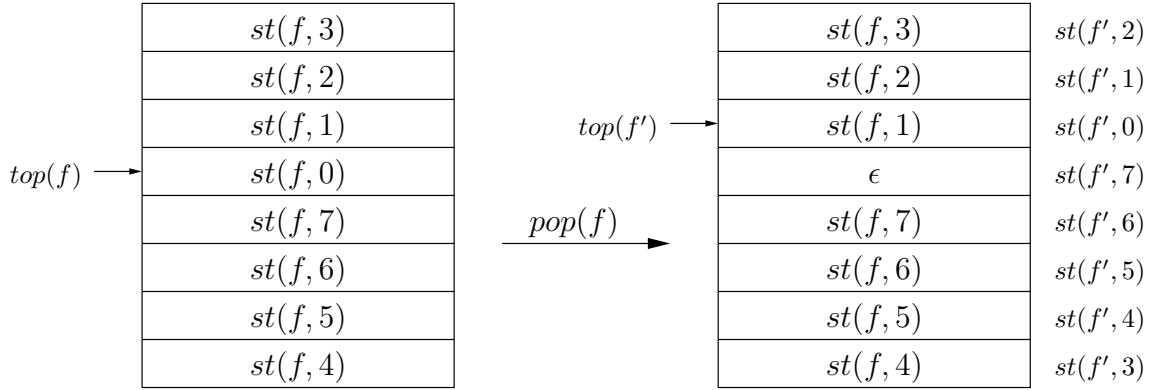


Figure 7.2: Effect of a pop operation on the x87 stack

with the result data. To this end we adopt another atomical function, which alters the FPU state accordingly.

$$\begin{aligned}
store : (\mathbb{B}^*, FPU) &\rightarrow FPU \\
store(x, f) &= f' \\
f'.ans &= x \\
\forall comp \in f' : comp \neq ans &\Rightarrow f'.comp = f.comp
\end{aligned}$$

Using this function we can comfortably update the field *ans*, leaving all else components untouched.

7.4.2 Popping the Stack

Many x87 instructions may not only save computed results in x87 stack registers repectively to memory locations, they may also remove the stack-top data element in parallel. This removal is called *popping the stack*. To introduce this funtionality to our model according to *push(f)* we define another atomical stack operation *pop(f)*, which returns the configuration after detaching the stack top element. The operation is demonstrated in figure 7.2. Let $i = sri(f, 0)$ be the stack-top register index in configuration f .

$$\begin{aligned}
pop : FPU &\rightarrow FPU \\
pop(f) &= f' \\
\langle top(f') \rangle &= (\langle top(f) \rangle + 1) \bmod 8 \\
empty(f', 7) &= 1
\end{aligned}$$

All other components remain unmodified. For a picture of the stack organisation see section 4.1.

$$\begin{aligned}
&\forall comp \in f' : \\
&comp \notin \{fsw[13], fsw[12], fsw[11], ftw[2 \cdot i + 1], ftw[2 \cdot i]\} \Rightarrow f'.comp = f.comp
\end{aligned}$$

7.4.3 Transition Function

Ultimately we will define the last step of our overall FPU transition function. This employs a case distinction on the different storing/popping behaviour of instructions. Confer sections 6.1.2 and

6.1.5 to identify the corresponding instructions.

$$\sigma_{fpu}^{store}(f) = \begin{cases} store(res_1(f), f) & : \quad xupdm(f) \wedge \overline{pop?(f)} \wedge \overline{es(f)} \\ pop(store(res_1(f), f)) & : \quad xupdm(f) \wedge pop?(f) \wedge \overline{es(f)} \\ pop(f) & : \quad \overline{xupdm(f)} \wedge pop(f) \wedge \overline{es(f)} \\ pop(pop(f)) & : \quad pop2?(f) \wedge \overline{es(f)} \\ f & : \quad \text{otherwise} \end{cases}$$

If a floating-point instruction does neither returns its result to memory nor pops the stack, no changes are applied to the configuration.

At last we completed the definition of the FPU state transfer function. Based on the floating-point interface it describes the effects of the multitude of x87 instructions on the FPU in a formal mathematical way. In the interface *execFPU* function the resulting FPU configuration is saved as the new value of the *fpu* component. This next state FPU configuration is wrapped up in Appendix D.

As we already explained in section 3.5, any further exception after this point⁵ lies outside the scope of the FPU. However in case of a repetition of the instruction under consideration the old floating-point configuration must be restored.

⁵for instance a page fault while storing the result to memory

Chapter 8

Conclusion

In the preceding chapters we established a formal model of the x86-64 Floating-Point Unit. Starting from a given general purpose CPU model, we constructed an interface and a floating-point environment, encapsulating the execution of x87 instructions from the rest of the processor. Thus we were able to describe the semantics of floating-point instructions in a precise and self-contained manner. In doing so we benefited from the formalization of floating-point arithmetics in [MP00]. Nevertheless we had to extend these definitions to meet the complexity of the x87 instruction set. By generalizing the FPU operations and subdividing the execution process into four consecutive phases we were finally able to define the resulting floating-point configuration in a structured, step-by-step way.

However the presented model is quite sophisticated and it is rather complicated to extract the next state of the FPU after executing an x87 instruction. Hence we wrapped up the next state configuration based on our model, specifying the available cases for every component of the FPU. This summary can be found in Appendix D. It tops off this thesis and allows a clear view on the variety of functionalities and features incorporated in our model.

In the future this thesis will contribute to the completion of the x86-64 CPU model and thus to the verification of a modern hypervisor as aspired by the Verisoft XT project. We will try to further simplify the model and check its correctness via automated tools. In addition parts of our thesis may be reused for defining the semantics of XMM and MMX multimedia instructions. Eventually a formal model of the x87 instruction subset may also complement and to a certain extent substitute the existing architecture manuals regarding floating-point instructions.

Appendix A

Opcode Table

The following table lists all x87 opcodes and links to their associated floating-point instructions. With the help of this table we can among others decode types and sizes of operands. Note that the operand information does not reflect how certain functions are called in assembler language. It is rather a hint how many floating point operands are necessary for the calculation and whether they reside in memory or floating point registers. In the Comments we sometimes use aliases for $ST(0)$ and $ST(1)$.

$$\begin{aligned} x &= ST(0) \\ y &= ST(1) \end{aligned}$$

One should also mention, that values which are saved on the FPU stack are always converted to IEEE double-extended precision format. This conversion is assumed implicit for all formats of memory operands.

The columns $r?$, $uf?$ and $um?$ signal whether the x87 instruction reads memory, updates flags or updates memory. Apparently at most one of these predicates may be true for one FPU instruction.

Table A.1 – FPU opcodes

Opc	mod	reg/Opc2	rm	Mnemonic	Class	r?	uf?	um?	Op 1	Op 2	Comment
0x0FAE	00	000		FXSAVE	fpu	0	0	1	FPM_{iL}	ϵ	store XMM/MMX/x87 state
0x0FAE	01	000		FXSAVE	fpu	0	0	1	FPM_{iL}	ϵ	store XMM/MMX/x87 state
0x0FAE	10	000		FXSAVE	fpu	0	0	1	FPM_{iL}	ϵ	store XMM/MMX/x87 state
0x0FAE	00	001		FXRSTOR	fpu	1	0	0	FPM_{iL}	ϵ	load XMM/MMX/x87 state
0x0FAE	01	001		FXRSTOR	fpu	1	0	0	FPM_{iL}	ϵ	load XMM/MMX/x87 state
0x0FAE	10	001		FXRSTOR	fpu	1	0	0	FPM_{iL}	ϵ	load XMM/MMX/x87 state
0x9B	ϵ	ϵ	ϵ	WAIT	fpu	0	0	0	ϵ	ϵ	handle pending MF exceptions
0xD8	00	000		FADD	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) + Op1$
0xD8	01	000		FADD	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) + Op1$
0xD8	10	000		FADD	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) + Op1$
0xD8	11	000		FADD	fpu	0	0	0	FPS	FPS	$ST(0) = ST(0) + Op1$
0xD8	00	001		FMUL	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) \cdot Op1$
0xD8	01	001		FMUL	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) \cdot Op1$
0xD8	10	001		FMUL	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) \cdot Op1$
0xD8	11	001		FMUL	fpu	0	0	0	FPS	FPS	$ST(0) = ST(0) \cdot Op1$
0xD8	00	010		FCOM	fpu	1	0	0	FPM_{32}	FPS	$CC = ST(0) \geq Op1$
0xD8	01	010		FCOM	fpu	1	0	0	FPM_{32}	FPS	$CC = ST(0) \geq Op1$
0xD8	10	010		FCOM	fpu	1	0	0	FPM_{32}	FPS	$CC = ST(0) \geq Op1$
0xD8	11	010		FCOM	fpu	0	0	0	FPS	FPS	$CC = ST(0) \geq Op1$
0xD8	00	011		FCOMP	fpu	1	0	0	FPM_{32}	FPS	$CC = ST(0) \geq Op1$
0xD8	01	011		FCOMP	fpu	1	0	0	FPM_{32}	FPS	$CC = ST(0) \geq Op1$
0xD8	10	011		FCOMP	fpu	1	0	0	FPM_{32}	FPS	$CC = ST(0) \geq Op1$
0xD8	11	011		FCOMP	fpu	0	0	0	FPS	FPS	$CC = ST(0) \geq Op1$
0xD8	00	100		FSUB	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) - Op1$
0xD8	01	100		FSUB	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) - Op1$
0xD8	10	100		FSUB	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) - Op1$

Continued on next page

Table A.1 – FPU opcodes, continued from previous page

Opc	mod	reg/Opc2	rm	Mnemonic	Class	r?	uf?	um?	Op 1	Op 2	Comment
0xD8	11	100		FSUB	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$ST(0) = ST(0) - Op1$
0xD8	00	101		FSUBR	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = Op1 - ST(0)$
0xD8	01	101		FSUBR	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = Op1 - ST(0)$
0xD8	10	101		FSUBR	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = Op1 - ST(0)$
0xD8	11	101		FSUBR	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$ST(0) = Op1 - ST(0)$
0xD8	00	110		FDIV	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = ST(0)/Op1$
0xD8	01	110		FDIV	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = ST(0)/Op1$
0xD8	10	110		FDIV	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = ST(0)/Op1$
0xD8	11	110		FDIV	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$ST(0) = ST(0)/Op1$
0xD8	00	111		FDIVR	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = Op1/ST(0)$
0xD8	01	111		FDIVR	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = Op1/ST(0)$
0xD8	10	111		FDIVR	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = Op1/ST(0)$
0xD8	11	111		FDIVR	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$ST(0) = Op1/ST(0)$
0xD9	00	000		FLD	fpu	1	0	0	<i>FPM₃₂</i>	ϵ	<i>push(Op1)</i>
0xD9	01	000		FLD	fpu	1	0	0	<i>FPM₃₂</i>	ϵ	<i>push(Op1)</i>
0xD9	10	000		FLD	fpu	1	0	0	<i>FPM₃₂</i>	ϵ	<i>push(Op1)</i>
0xD9	11	000		FLD	fpu	0	0	0	<i>FPS</i>	ϵ	<i>push(Op1)</i>
0xD9	11	001		FXCH	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$ST(0) = Op1, Op1 = ST(0)$
0xD9	00	010		FST	fpu	0	0	1	<i>FPM₃₂</i>	ϵ	$Op1 = ST(0)$
0xD9	01	010		FST	fpu	0	0	1	<i>FPM₃₂</i>	ϵ	$Op1 = ST(0)$
0xD9	10	010		FST	fpu	0	0	1	<i>FPM₃₂</i>	ϵ	$Op1 = ST(0)$
0xD9	11	010	000	FNOP	fpu	0	0	0	ϵ	ϵ	no operation
0xD9	00	011		FSTP	fpu	0	0	1	<i>FPM₃₂</i>	ϵ	$Op1 = ST(0), pop$
0xD9	01	011		FSTP	fpu	0	0	1	<i>FPM₃₂</i>	ϵ	$Op1 = ST(0), pop$
0xD9	10	011		FSTP	fpu	0	0	1	<i>FPM₃₂</i>	ϵ	$Op1 = ST(0), pop$
0xD9	00	100		FLDENV	fpu	1	0	0	<i>FPM₁₆</i>	ϵ	load x87 environment
0xD9	01	100		FLDENV	fpu	1	0	0	<i>FPM₁₆</i>	ϵ	load x87 environment
0xD9	10	100		FLDENV	fpu	1	0	0	<i>FPM₁₆</i>	ϵ	load x87 environment
0xD9	11	100	000	FCHS	fpu	0	0	0	<i>FPS</i>	ϵ	$ST(0) = -ST(0)$
0xD9	11	100	001	FABS	fpu	0	0	0	<i>FPS</i>	ϵ	$ST(0) = ST(0) $
0xD9	11	100	100	FTST	fpu	0	0	0	<i>FPS</i>	ϵ	$CC = ST(0) \geq 0.0$
0xD9	11	100	101	FXAM	fpu	0	0	0	<i>FPS</i>	ϵ	$CC = \text{type of } ST(0)$
0xD9	00	101		FLDCW	fpu	1	0	0	<i>FPM₁₆</i>	ϵ	$FCW = Op1$
0xD9	01	101		FLDCW	fpu	1	0	0	<i>FPM₁₆</i>	ϵ	$FCW = Op1$
0xD9	10	101		FLDCW	fpu	1	0	0	<i>FPM₁₆</i>	ϵ	$FCW = Op1$
0xD9	11	101	000	FLD1	fpu	0	0	0	ϵ	ϵ	<i>push(+1.0)</i>
0xD9	11	101	001	FLDL2T	fpu	0	0	0	ϵ	ϵ	<i>push(log₂ 10)</i>
0xD9	11	101	010	FLDL2E	fpu	0	0	0	ϵ	ϵ	<i>push(log₂ e)</i>
0xD9	11	101	011	FLDPI	fpu	0	0	0	ϵ	ϵ	<i>push(π)</i>
0xD9	11	101	100	FLDLG2	fpu	0	0	0	ϵ	ϵ	<i>push(log₁₀ 2)</i>
0xD9	11	101	101	FLDLN2	fpu	0	0	0	ϵ	ϵ	<i>push(ln 2)</i>
0xD9	11	101	110	FLDZ	fpu	0	0	0	ϵ	ϵ	<i>push(0.0)</i>
0xD9	00	110		FNSTENV	fpu	0	0	1	<i>FPM₁₆</i>	ϵ	store x87 environment
0xD9	01	110		FNSTENV	fpu	0	0	1	<i>FPM₁₆</i>	ϵ	store x87 environment
0xD9	10	110		FNSTENV	fpu	0	0	1	<i>FPM₁₆</i>	ϵ	store x87 environment
0xD9	11	110	000	F2XM1	fpu	0	0	0	<i>FPS</i>	ϵ	$ST(0) = 2^{ST(0)} - 1$
0xD9	11	110	001	FY2LX	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$y = y \cdot \log_2 x, pop$
0xD9	11	110	010	FPTAN	fpu	0	0	0	<i>FPS</i>	ϵ	$x = \tan(x), push(1.0)$
0xD9	11	110	011	FPATAN	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$y = \arctan(y/x), pop$
0xD9	11	110	100	FXTRACT	fpu	0	0	0	<i>FPS</i>	ϵ	$x = \lfloor \log_2 x \rfloor, push(\frac{x}{2^{\lfloor \log_2 x \rfloor}})$
0xD9	11	110	101	FPREM1	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$x = x - r_{ne}(x/y) \cdot y$
0xD9	11	110	110	FDECSTP	fpu	0	0	0	ϵ	ϵ	$top = top + 1$
0xD9	11	110	111	FINCSTP	fpu	0	0	0	ϵ	ϵ	$top = top - 1$
0xD9	00	111		FNSTCW	fpu	0	0	1	<i>FPM₁₆</i>	ϵ	$Op1 = FCW$
0xD9	01	111		FNSTCW	fpu	0	0	1	<i>FPM₁₆</i>	ϵ	$Op1 = FCW$
0xD9	10	111		FNSTCW	fpu	0	0	1	<i>FPM₁₆</i>	ϵ	$Op1 = FCW$
0xD9	11	111	000	FPREM	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$x = x - trunc(x/y) \cdot y$
0xD9	11	111	001	FY2LXP1	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$y = y \cdot \log_2(x + 1), pop$
0xD9	11	111	010	FSQRT	fpu	0	0	0	<i>FPS</i>	ϵ	$ST(0) = \sqrt{ST(0)}$
0xD9	11	111	011	FSINCOS	fpu	0	0	0	<i>FPS</i>	ϵ	$x = \sin x, push(\cos x)$
0xD9	11	111	100	FRNDINT	fpu	0	0	0	<i>FPS</i>	ϵ	$ST(0) = int(ST(0))$
0xD9	11	111	101	FSCALE	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$ST(0) = ST(0) \cdot 2^{int(ST(1))}$
0xD9	11	111	110	FSIN	fpu	0	0	0	<i>FPS</i>	ϵ	$ST(0) = \sin ST(0)$
0xD9	11	111	111	FCOS	fpu	0	0	0	<i>FPS</i>	ϵ	$ST(0) = \cos ST(0)$
0xDA	00	000		FIADD	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = ST(0) + Op1$
0xDA	01	000		FIADD	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = ST(0) + Op1$
0xDA	10	000		FIADD	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = ST(0) + Op1$
0xDA	11	000		FCMOVB	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$< \Rightarrow ST(0) = Op1$
0xDA	00	001		FIMUL	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = ST(0) \cdot Op1$
0xDA	01	001		FIMUL	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = ST(0) \cdot Op1$
0xDA	10	001		FIMUL	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$ST(0) = ST(0) \cdot Op1$
0xDA	11	001		FCMOVE	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$= \Rightarrow ST(0) = Op1$
0xDA	00	010		FICOM	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$CC = ST(0) \geq Op1$
0xDA	01	010		FICOM	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$CC = ST(0) \geq Op1$
0xDA	10	010		FICOM	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$CC = ST(0) \geq Op1$
0xDA	11	010		FCMOVBE	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	$< \Rightarrow ST(0) = Op1$
0xDA	00	011		FICOMP	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$CC = ST(0) = Op1, pop$
0xDA	01	011		FICOMP	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$CC = ST(0) = Op1, pop$
0xDA	10	011		FICOMP	fpu	1	0	0	<i>FPM₃₂</i>	<i>FPS</i>	$CC = ST(0) = Op1, pop$
0xDA	11	100		FCMOVU	fpu	0	0	0	<i>FPS</i>	<i>FPS</i>	Unordered $\Rightarrow ST(0) = Op1$

Continued on next page

Table A.1 – FPU opcodes, continued from previous page

Opc	mod	reg/Opc2	rm	Mnemonic	Class	r?	uf?	um?	Op 1	Op 2	Comment
0xDA	00	100		FISUB	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) - Op1$
0xDA	01	100		FISUB	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) - Op1$
0xDA	10	100		FISUB	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0) - Op1$
0xDA	00	101		FISUBR	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = Op1 - ST(0)$
0xDA	01	101		FISUBR	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = Op1 - ST(0)$
0xDA	10	101		FISUBR	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = Op1 - ST(0)$
0xDA	11	101	001	FUCOMPP	fpu	0	0	0	FPS	FPS	$CC = ST(0) \geq ST(1), pop, pop$
0xDA	00	110		FIDIV	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0)/Op1$
0xDA	01	110		FIDIV	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0)/Op1$
0xDA	10	110		FIDIV	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = ST(0)/Op1$
0xDA	00	111		FIDIVR	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = Op1/ST(0)$
0xDA	01	111		FIDIVR	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = Op1/ST(0)$
0xDA	10	111		FIDIVR	fpu	1	0	0	FPM_{32}	FPS	$ST(0) = Op1/ST(0)$
0xDB	00	000		FILD	fpu	1	0	0	FPM_{32}	ϵ	$push(Op1)$
0xDB	01	000		FILD	fpu	1	0	0	FPM_{32}	ϵ	$push(Op1)$
0xDB	10	000		FILD	fpu	1	0	0	FPM_{32}	ϵ	$push(Op1)$
0xDB	11	000		FCMOVNB	fpu	0	0	0	FPS	FPS	$\nless \Rightarrow ST(0) = Op1$
0xDB	00	001		FISTTP	fpu	0	0	1	FPM_{32}	ϵ	$Op1 = trunc(ST(0)), pop$
0xDB	01	001		FISTTP	fpu	0	0	1	FPM_{32}	ϵ	$Op1 = trunc(ST(0)), pop$
0xDB	10	001		FISTTP	fpu	0	0	1	FPM_{32}	ϵ	$Op1 = trunc(ST(0)), pop$
0xDB	11	001		FCMOVNE	fpu	0	0	0	FPS	FPS	$\neq \Rightarrow ST(0) = Op1$
0xDB	00	010		FIST	fpu	0	0	1	FPM_{32}	ϵ	$Op1 = int(ST(0))$
0xDB	01	010		FIST	fpu	0	0	1	FPM_{32}	ϵ	$Op1 = int(ST(0))$
0xDB	10	010		FIST	fpu	0	0	1	FPM_{32}	ϵ	$Op1 = int(ST(0))$
0xDB	11	010		FCMOVNBE	fpu	0	0	0	FPS	FPS	$\nless \Rightarrow ST(0) = Op1$
0xDB	00	011		FISTP	fpu	0	0	1	FPM_{32}	ϵ	$Op1 = int(ST(0)), pop$
0xDB	01	011		FISTP	fpu	0	0	1	FPM_{32}	ϵ	$Op1 = int(ST(0)), pop$
0xDB	10	011		FISTP	fpu	0	0	1	FPM_{32}	ϵ	$Op1 = int(ST(0)), pop$
0xDB	11	011		FCMOVNU	fpu	0	0	0	FPS	FPS	$\neg Unordered \Rightarrow ST(0) = Op1$
0xDB	11	100	010	FNCLEX	fpu	0	0	0	ϵ	ϵ	clear FSW exception bits
0xDB	11	100	011	FNINIT	fpu	0	0	0	ϵ	ϵ	initialize FPU
0xDB	00	101		FLD	fpu	1	0	0	FPM_{80}	ϵ	$push(Op1)$
0xDB	01	101		FLD	fpu	1	0	0	FPM_{80}	ϵ	$push(Op1)$
0xDB	10	101		FLD	fpu	1	0	0	FPM_{80}	ϵ	$push(Op1)$
0xDB	11	101		FUCOMI	fpu	0	1	0	FPS	FPS	$rFlags = ST(0) \geq Op1$
0xDB	11	110		FCOMI	fpu	0	1	0	FPS	FPS	$rFlags = ST(0) \geq Op1$
0xDB	00	111		FSTP	fpu	0	0	1	FPM_{80}	ϵ	$Op1 = ST(0), pop$
0xDB	01	111		FSTP	fpu	0	0	1	FPM_{80}	ϵ	$Op1 = ST(0), pop$
0xDB	10	111		FSTP	fpu	0	0	1	FPM_{80}	ϵ	$Op1 = ST(0), pop$
0xDC	00	000		FADD	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0) + Op1$
0xDC	01	000		FADD	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0) + Op1$
0xDC	10	000		FADD	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0) + Op1$
0xDC	11	000		FADD	fpu	0	0	0	FPS	FPS	$Op1 = ST(0) + Op1$
0xDC	00	001		FMUL	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0) \cdot Op1$
0xDC	01	001		FMUL	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0) \cdot Op1$
0xDC	10	001		FMUL	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0) \cdot Op1$
0xDC	11	001		FMUL	fpu	0	0	0	FPS	FPS	$Op1 = ST(0) \cdot Op1$
0xDC	00	010		FCOM	fpu	1	0	0	FPM_{64}	FPS	$CC = ST(0) \geq Op1$
0xDC	01	010		FCOM	fpu	1	0	0	FPM_{64}	FPS	$CC = ST(0) \geq Op1$
0xDC	10	010		FCOM	fpu	1	0	0	FPM_{64}	FPS	$CC = ST(0) \geq Op1$
0xDC	00	011		FCOMP	fpu	1	0	0	FPM_{64}	FPS	$CC = ST(0) \geq Op1, pop$
0xDC	01	011		FCOMP	fpu	1	0	0	FPM_{64}	FPS	$CC = ST(0) \geq Op1, pop$
0xDC	10	011		FCOMP	fpu	1	0	0	FPM_{64}	FPS	$CC = ST(0) \geq Op1, pop$
0xDC	00	100		FSUB	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0) - Op1$
0xDC	01	100		FSUB	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0) - Op1$
0xDC	10	100		FSUB	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0) - Op1$
0xDC	11	100		FSUBR	fpu	0	0	0	FPS	FPS	$Op1 = ST(0) - Op1$
0xDC	00	101		FSUBR	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = Op1 - ST(0)$
0xDC	01	101		FSUBR	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = Op1 - ST(0)$
0xDC	10	101		FSUBR	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = Op1 - ST(0)$
0xDC	11	101		FSUB	fpu	0	0	0	FPS	FPS	$Op1 = Op1 - ST(0)$
0xDC	00	110		FDIV	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0)/Op1$
0xDC	01	110		FDIV	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0)/Op1$
0xDC	10	110		FDIV	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = ST(0)/Op1$
0xDC	11	110		FDIVR	fpu	0	0	0	FPS	FPS	$Op1 = ST(0)/Op1$
0xDC	00	111		FDIVR	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = Op1/ST(0)$
0xDC	01	111		FDIVR	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = Op1/ST(0)$
0xDC	10	111		FDIVR	fpu	1	0	0	FPM_{64}	FPS	$ST(0) = Op1/ST(0)$
0xDC	11	111		FDIV	fpu	0	0	0	FPS	FPS	$Op1 = Op1/ST(0)$
0xDD	00	000		FLD	fpu	1	0	0	FPM_{64}	ϵ	$push(Op1)$
0xDD	01	000		FLD	fpu	1	0	0	FPM_{64}	ϵ	$push(Op1)$
0xDD	10	000		FLD	fpu	1	0	0	FPM_{64}	ϵ	$push(Op1)$
0xDD	11	000		FFREE	fpu	0	0	0	ϵ	ϵ	set FP register tag to empty
0xDD	00	001		FISTTP	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = trunc(ST(0)), pop$
0xDD	01	001		FISTTP	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = trunc(ST(0)), pop$
0xDD	10	001		FISTTP	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = trunc(ST(0)), pop$
0xDD	00	010		FST	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = ST(0)$
0xDD	01	010		FST	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = ST(0)$
0xDD	10	010		FST	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = ST(0)$
0xDD	11	010		FST	fpu	0	0	0	FPS	ϵ	$Op1 = ST(0)$
0xDD	00	011		FSTP	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = ST(0), pop$

Continued on next page

Table A.1 – FPU opcodes, continued from previous page

Opc	mod	reg/Opc2	rm	Mnemonic	Class	r?	uf?	um?	Op 1	Op 2	Comment
0xDD	01	011		FSTP	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = ST(0), pop$
0xDD	10	011		FSTP	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = ST(0), pop$
0xDD	11	011		FSTP	fpu	0	0	0	FPS	ϵ	$Op1 = ST(0), pop$
0xDD	00	100		FRSTOR	fpu	1	0	0	FPM_{iM}	ϵ	load x87/MMX state
0xDD	01	100		FRSTOR	fpu	1	0	0	FPM_{iM}	ϵ	load x87/MMX state
0xDD	10	100		FRSTOR	fpu	1	0	0	FPM_{iM}	ϵ	load x87/MMX state
0xDD	11	100		FUCOM	fpu	0	0	0	FPS	FPS	$CC = ST(0) \geq Op1$
0xDD	11	101		FUCOMP	fpu	0	0	0	FPS	FPS	$CC = ST(0) \geq Op1, pop$
0xDD	00	110		FNSAVE	fpu	0	0	1	FPM_{iM}	ϵ	store x87/MMX state
0xDD	01	110		FNSAVE	fpu	0	0	1	FPM_{iM}	ϵ	store x87/MMX state
0xDD	10	110		FNSAVE	fpu	0	0	1	FPM_{iM}	ϵ	store x87/MMX state
0xDD	00	111		FNSTSW	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = FSW$
0xDD	01	111		FNSTSW	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = FSW$
0xDD	10	111		FNSTSW	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = FSW$
0xDE	00	000		FIADD	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0) + Op1$
0xDE	01	000		FIADD	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0) + Op1$
0xDE	10	000		FIADD	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0) + Op1$
0xDE	11	000		FADDP	fpu	0	0	0	FPS	FPS	$Op1 = ST(0) + Op1, pop$
0xDE	00	001		FIMUL	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0) \cdot Op1$
0xDE	01	001		FIMUL	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0) \cdot Op1$
0xDE	10	001		FIMUL	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0) \cdot Op1$
0xDE	11	001		FMULP	fpu	0	0	0	FPS	FPS	$Op1 = ST(0) \cdot Op1$
0xDE	00	010		FICOM	fpu	1	0	0	FPM_{16}	FPS	$CC = ST(0) \geq Op1$
0xDE	01	010		FICOM	fpu	1	0	0	FPM_{16}	FPS	$CC = ST(0) \geq Op1$
0xDE	10	010		FICOM	fpu	1	0	0	FPM_{16}	FPS	$CC = ST(0) \geq Op1$
0xDE	00	011		FICOMP	fpu	1	0	0	FPM_{16}	FPS	$CC = ST(0) \geq Op1, pop$
0xDE	01	011		FICOMP	fpu	1	0	0	FPM_{16}	FPS	$CC = ST(0) \geq Op1, pop$
0xDE	10	011		FICOMP	fpu	1	0	0	FPM_{16}	FPS	$CC = ST(0) \geq Op1, pop$
0xDE	11	011		FCOMPP	fpu	1	0	0	FPS	FPS	$CC = ST(0) \geq ST(1), pop, pop$
0xDE	00	100		FISUB	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0) - Op1$
0xDE	01	100		FISUB	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0) - Op1$
0xDE	10	100		FISUB	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0) - Op1$
0xDE	11	100		FSUBRP	fpu	0	0	0	FPS	FPS	$Op1 = ST(0) - Op1, pop$
0xDE	00	101		FISUBR	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = Op1 - ST(0)$
0xDE	01	101		FISUBR	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = Op1 - ST(0)$
0xDE	10	101		FISUBR	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = Op1 - ST(0)$
0xDE	11	101		FSUBP	fpu	0	0	0	FPS	FPS	$Op1 = Op1 - ST(0), pop$
0xDE	00	110		FIDIV	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0)/Op1$
0xDE	01	110		FIDIV	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0)/Op1$
0xDE	10	110		FIDIV	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = ST(0)/Op1$
0xDE	11	110		FIDIVR	fpu	0	0	0	FPS	FPS	$Op1 = ST(0)/Op1, pop$
0xDE	00	111		FIDIVR	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = Op1/ST(0)$
0xDE	01	111		FIDIVR	fpu	1	0	0	FPM_{16}	FPS	$ST(0) = Op1/ST(0)$
0xDE	11	111		FIDIVR	fpu	0	0	0	FPS	FPS	$Op1 = Op1/ST(0), pop$
0xDF	00	000		FILD	fpu	1	0	0	FPM_{16}	ϵ	push(Op1)
0xDF	01	000		FILD	fpu	1	0	0	FPM_{16}	ϵ	push(Op1)
0xDF	10	000		FILD	fpu	1	0	0	FPM_{16}	ϵ	push(Op1)
0xDF	00	001		FISTTP	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = trunc(ST(0)), pop$
0xDF	01	001		FISTTP	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = trunc(ST(0)), pop$
0xDF	10	001		FISTTP	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = trunc(ST(0)), pop$
0xDF	00	010		FIST	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = int(ST(0))$
0xDF	01	010		FIST	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = int(ST(0))$
0xDF	10	010		FIST	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = int(ST(0))$
0xDF	00	011		FISTP	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = int(ST(0)), pop$
0xDF	01	011		FISTP	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = int(ST(0)), pop$
0xDF	10	011		FISTP	fpu	0	0	1	FPM_{16}	ϵ	$Op1 = int(ST(0)), pop$
0xDF	00	100		FBLD	fpu	1	0	0	FPM_{16}	ϵ	push(Op1)
0xDF	01	100		FBLD	fpu	1	0	0	FPM_{16}	ϵ	push(Op1)
0xDF	10	100		FBLD	fpu	1	0	0	FPM_{16}	ϵ	push(Op1)
0xDF	11	100	000	FNSTSW	fpu	0	0	1	AX_{16}	ϵ	$Op1 = FSW$
0xDF	00	101		FILD	fpu	1	0	0	FPM_{64}	ϵ	push(Op1)
0xDF	01	101		FILD	fpu	1	0	0	FPM_{64}	ϵ	push(Op1)
0xDF	10	101		FILD	fpu	1	0	0	FPM_{64}	ϵ	push(Op1)
0xDF	11	101		FUCOMIP	fpu	0	1	0	FPS	FPS	$rFlags = ST(0) \geq Op1, pop$
0xDF	00	110		FBSTP	fpu	0	0	1	FPM_{80}	ϵ	$Op1 = bcd(ST(0), pop$
0xDF	01	110		FBSTP	fpu	0	0	1	FPM_{80}	ϵ	$Op1 = bcd(ST(0), pop$
0xDF	10	110		FBSTP	fpu	0	0	1	FPM_{80}	ϵ	$Op1 = bcd(ST(0), pop$
0xDF	11	110		FCOMIP	fpu	0	1	0	FPS	FPS	$rFlags = ST(0) \geq Op1, pop$
0xDF	00	111		FISTP	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = int(ST(0)), pop$
0xDF	01	111		FISTP	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = int(ST(0)), pop$
0xDF	10	111		FISTP	fpu	0	0	1	FPM_{64}	ϵ	$Op1 = int(ST(0)), pop$

Appendix B

Instruction Predicates

The following table defines all x87 instruction predicates, except for *FXSAVE*, *FXRSTOR* and *WAIT*, whose definitions can be found in section 6.1.1. There is also only one predicate *FCMOV* for all Conditional Moves. We also omit all waiting control instructions as these are rather assembler macros than actual independent instructions. Note, that all predicates are defined for $opc \in \mathbb{B}^{11}$. One can easily deduce the forms for other arguments, using the scheme already given in section 6.1.

$$\begin{aligned} \text{instr}(I) &= \text{instr}(I.opc[7:0] \circ I.modrm) \\ \text{instr}(f) &= \text{instr}(opc(f)) \\ \text{instr}(opc[15:0]) &= (opc[15:11] = 11011) \wedge \text{instr}(opc[10:0]) \end{aligned}$$

Table B.1 – x87 instruction set

instr(<i>opc</i>)	instr(<i>opc</i>)=1 $\Leftrightarrow$ 11011$\circ$<i>opc</i> $\in \dots$	function
<i>data(opc) - Data Transfer</i>		
<i>FLD</i>	{0xD90[0:7],0xD94[0:7],0xD98[0:7],0xD9C[0:7], 0xD9C[0:7],0xDB2[8:F],0xDB6[8:F],0xDBA[8:F], 0xDD0[0:7],0xDD4[0:7],0xDD8[0:7]}	FP Load
<i>FST</i>	{0xD91[0:7],0xD95[0:7],0xD99[0:7], 0xDD1[0:7],DD5[0:7],DD9[0:7],DDD[0:7]}	FP Store Stack Top
<i>FSTP</i>	{0xD91[8:F],0xD95[8:F],0xD99[8:F], 0xDB3[8:F],0xDB7[8:F],0xDBB[8:F]}	FST And Pop
<i>FILD</i>	{0xDB0[0:7],0xDB4[0:7],0xDB8[0:7], 0xDF0[0:7],0xDF4[0:7],0xDF8[0:7], 0xDF2[8:F],0xDF6[8:F],0xDFA[8:F]}	FP Load Integer
<i>FIST</i>	{0xDB1[0:7],0xDB5[0:7],0xDB9[0:7], 0xDF1[0:7],0xDF5[0:7],0xDF9[0:7]}	FP Integer Store
<i>FISTP</i>	{0xDB1[8:F],0xDB5[8:F],0xDB9[8:F], 0xDF1[8:F],0xDF5[8:F],0xDF9[8:F], 0xDF3[8:F],0xDF7[8:F],0xDFB[8:F]}	FIST And Pop

Continued on next page

Table B.1 – x87 instruction set, continued from previous page

instr(<i>opc</i>)	instr(<i>opc</i>)=1 $\Leftrightarrow$ 11011$\circ$<i>opc</i> $\in$...	function
<i>FISTTP</i>	{0xDB0[8:F],0xDB0[8:F],0xDB0[8:F], 0xDD0[8:F],0xDD4[8:F],0xDD8[8:F], 0xDF0[8:F],0xDF4[8:F],0xDF8[8:F]}	Truncating FISTP
<i>FBLD</i>	{0xDF2[0:7],0xDF6[0:7],0xDFA[0:7]}	FP Load BCD
<i>FBSTP</i>	{0xDF3[0:7],0xDF7[0:7],0xDFB[0:7]}	FP Store BCD And Pop
<i>FCMOV</i>	{0xDAC[0:7],0xDAC[8:F],0xDAD[0:7], 0xDAD[8:F],0xDBC[0:7],0xDBC[8:F], 0xDBD[0:7],0xDBD[8:F]}	FP Conditional Moves
<i>FXCH</i>	{0xD9C[8:F]}	FP Exchange
<i>FTRACT</i>	{0xD9F4}	FP Extract
<i>arith(opc)</i> - Arithmetic Operations		
<i>FADD</i>	{0xD80[0:7],0xD84[0:7],0xD88[0:7],0xD8C[0:7], 0xDC0[0:7],0xDC4[0:7],0xDC8[0:7],0xDCC[0:7]}	FP Add
<i>FADDP</i>	{0xDEC[0:7]}	FADD and Pop
<i>FIADD</i>	{0xDA0[0:7],0xDA4[0:7],0xDA8[0:7], 0xDE0[0:7],0xDE4[0:7],0xDE8[0:7]}	FP Add Integer
<i>FSUB</i>	{0xD82[0:7],0xD86[0:7],0xD8A[0:7],0xD8E[0:7], 0xDC2[0:7],0xDC6[8:F],0xDCA[8:F],0xDCE[8:F]}	FP Subtract
<i>FSUBP</i>	{0xDEE[8:F]}	FSUB and Pop
<i>FISUB</i>	{0xDA2[0:7],0xDA6[0:7],0xDAA[0:7], 0xDE2[0:7],0xDE6[0:7],0xDEA[0:7]}	FP Subtract Integer
<i>FSUBR</i>	{0xD82[8:F],0xD86[8:F],0xD8A[8:F],0xD8E[8:F], 0xDC2[8:F],0xDC6[0:7],0xDCA[0:7],0xDCE[0:7]}	Reverse FSUB
<i>FSUBRP</i>	{0xDEE[0:7]}	Reverse FSUBP
<i>FISUBR</i>	{0xDA2[8:F],0xDA6[8:F],0xDAA[8:F], 0xDE2[8:F],0xDE6[8:F],0xDEA[8:F]}	Reverse FISUB
<i>FMUL</i>	{0xD80[8:F],0xD84[8:F],0xD88[8:F],0xD8C[8:F], 0xDC0[8:F],0xDC4[8:F],0xDC8[8:F],0xDCC[8:F]}	FP Multiply
<i>FMULP</i>	{0xDEC[8:F]}	FMUL and Pop
<i>FIMUL</i>	{0xDA0[8:F],0xDA4[8:F],0xDA8[8:F], 0xDE0[8:F],0xDE4[8:F],0xDE8[8:F]}	FP Multiply Integer
<i>FDIV</i>	{0xD83[0:7],0xD87[0:7],0xD8B[0:7],0xD8F[0:7], 0xDC3[0:7],0xDC7[8:F],0xDCB[8:F],0xDCF[8:F]}	FP Divide
<i>FDIVP</i>	{0xDEF[8:F]}	FDIV and Pop
<i>FIDIV</i>	{0xDA3[0:7],0xDA7[0:7],0xDAB[0:7], 0xDE3[0:7],0xDE7[0:7],0xDEB[0:7]}	FP Divide by Integer
<i>FDIVR</i>	{0xD83[8:F],0xD87[8:F],0xD8B[8:F],0xD8F[8:F], 0xDC3[8:F],0xDC7[0:7],0xDCB[0:7],0xDCF[0:7]}	Reverse FDIV
<i>FDIVRP</i>	{0xDEF[0:7]}	Reverse FDIVP

Continued on next page

Table B.1 – x87 instruction set, continued from previous page

instr(<i>opc</i>)	instr(<i>opc</i>)=1 $\Leftrightarrow$ 11011$\circ$<i>opc</i> $\in \dots$	function
<i>FIDIVR</i>	{0xDA3[8:F],0xDA7[8:F],0xDAB[8:F], 0xDE3[8:F],0xDE7[8:F],0xDEB[8:F]}	Reverse FIDIV
<i>FABS</i>	{0xD9E1}	FP Absolute Value
<i>FCHS</i>	{0xD9E0}	FP Change Sign
<i>FRNDINT</i>	{0xD9FC}	FP Round to Integer
<i>FPREM</i>	{0xD9F8}	FP Partial Remainder (Truncating)
<i>FPREM1</i>	{0xD9F5}	FP Partial Remainder (Rounding to Nearest)
<i>FSQRT</i>	{0xD9FA}	FP Square Root
<i>const(opc)</i> - Load Constants		
<i>FLDZ</i>	{0xD9EE}	FP Load +0.0
<i>FLD1</i>	{0xD9E8}	FP Load +1.0
<i>FLDPI</i>	{0xD9EB}	FP Load π
<i>FLDL2E</i>	{0xD9EA}	FP Load $\log_2 e$
<i>FLDL2T</i>	{0xD9E9}	FP Load $\log_2 10$
<i>FLDLG2</i>	{0xD9EC}	FP Load $\log_{10} 2$
<i>FLDLN2</i>	{0xD9ED}	FP Load $\ln 2$
<i>cmp(opc)</i> - Compare and Test		
<i>FCOM</i>	{0xD81[0:7],0xD85[0:7],0xD89[0:7],0xD8D[0:7], 0xDC1[0:7],0xDC5[0:7],0xDC9[0:7]}	FP Compare
<i>FCOMP</i>	{0xD81[8:F],0xD85[8:F],0xD89[8:F],0xD8D[8:F], 0xDC1[8:F],0xDC5[8:F],0xDC9[8:F]}	FCOM and Pop
<i>FCOMPP</i>	{0xDE9}	FCOM and Pop Twice
<i>FCOMI</i>	{0xDBF[0:7]}	FCOM and Set Flags
<i>FCOMIP</i>	{0xDFF[0:7]}	FCOMI and Pop
<i>FUCOM</i>	{0xDDE[0:7]}	FP Unordered Compare
<i>FUCOMP</i>	{0xDDE[8:F]}	FUCOM and Pop
<i>FUCOMPP</i>	{0xDAE9}	FUCOM and Pop Twice
<i>FUCOMI</i>	{0xDBE[8:F]}	FUCOM and Set Flags
<i>FUCOMIP</i>	{0xDFE[8:F]}	FUCOMI and Pop
<i>FICOM</i>	{0xDA1[0:7],0xDA5[0:7],0xDA9[0:7], 0xDE1[0:7],0xDE5[0:7],0xDE9[0:7]}	FP Integer Compare
<i>FICOMP</i>	{0xDA1[8:F],0xDA5[8:F],0xDA9[8:F], 0xDE1[8:F],0xDE5[8:F],0xDE9[8:F]}	FICOM and Pop
<i>FTST</i>	{0xD9E4}	FP Test with Zero
<i>FXAM</i>	{0xD9E5}	FP Examine
<i>trans(opc)</i> - Transcendental Functions		
<i>FSIN</i>	{0xD9FE}	FP Sine
<i>FCOS</i>	{0xD9FF}	FP Cosine

Continued on next page

Table B.1 – x87 instruction set, continued from previous page

instr(<i>opc</i>)	instr(<i>opc</i>)=1 $\Leftrightarrow$ 11011$\circ$<i>opc</i> $\in \dots$	function
<i>FSINCOS</i>	{0xD9FB}	FP Sine and Cosine
<i>FPTAN</i>	{0xD9F2}	FP Partial Tangent
<i>FPATAN</i>	{0xD9F3}	FP Partial Arctangent
<i>F2XM1</i>	{0xD9F0}	FP Compute $2^x - 1$
<i>FSCALE</i>	{0xD9FD}	FP Scale with 2^n
<i>FYL2X</i>	{0xD9F1}	FP Compute $y \cdot \log_2 x$
<i>FYL2XP1</i>	{0xD9F9}	FP Compute $y \cdot \log_2(x + 1)$
<i>stack(opc)</i> - Stack Management		
<i>FDECSTP</i>	{0xD9F6}	FP Decrement Stack-Top Pointer
<i>FINCSTP</i>	{0xD9F7}	FP Increment Stack-Top Pointer
<i>FFREE</i>	{0xDDC[0:7]}	Free FP Register
<i>fnop(opc)</i> - No Operation		
<i>FNOP</i>	{0xD9D0}	FP No Operation
<i>ctrl(opc)</i> - Control Operations		
<i>FNINIT</i>	{0xDBE3}	FP Initialize
<i>FNCLEX</i>	{0xDBE2}	FP Clear Exceptions
<i>FLDCW</i>	{0xD92[8:F],0xD96[8:F],0xD9A[8:F]}	FP Load x87 Control Word
<i>FNSTCW</i>	{0xD93[8:F],0xD97[8:F],0xD9B[8:F]}	FP Store x87 Control Word
<i>FNSTSW</i>	{0xDD3[8:F],0xDD7[8:F],0xDDB[8:F], DFE0}	FP Store x87 Status Word
<i>FLDENV</i>	{0xD92[0:7],0xD96[0:7],0xD9A[0:7]}	FP Load x87 Environment
<i>FNSTENV</i>	{0xD93[0:7],0xD97[0:7],0xD9B[0:7]}	FP Store x87 Environment
<i>FRSTOR</i>	{0xDD2[0:7],0xDD6[0:7],0xDDA[0:7]}	FP Restore x87 and MMX state
<i>FNSAVE</i>	{0xDD3[0:7],0xDD7[0:7],0xDDB[0:7]}	FP Save x87 and MMX state

Appendix C

FPU Image Formats

The following figures illustrate the different image formats used to save the FPU configuration. The x87 environment images for the different CPU modes are shown in figures C.1 to C.4. Figures C.5 and C.6 depict the XMM/x87/Media images for 64-bit and non-64-bit mode.

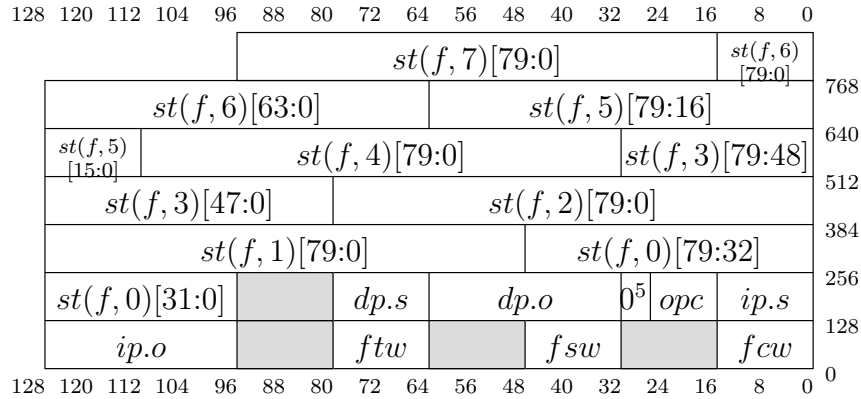


Figure C.1: x87 Environment image format for 32-bit protected mode

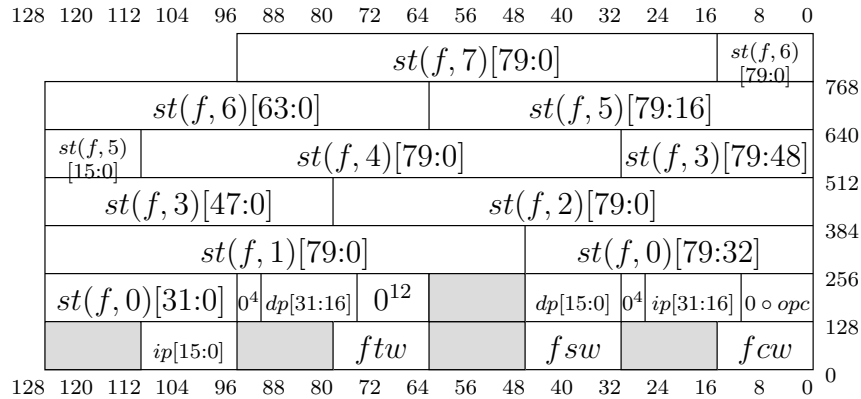


Figure C.2: x87 Environment image format for 32-bit real/virtual mode

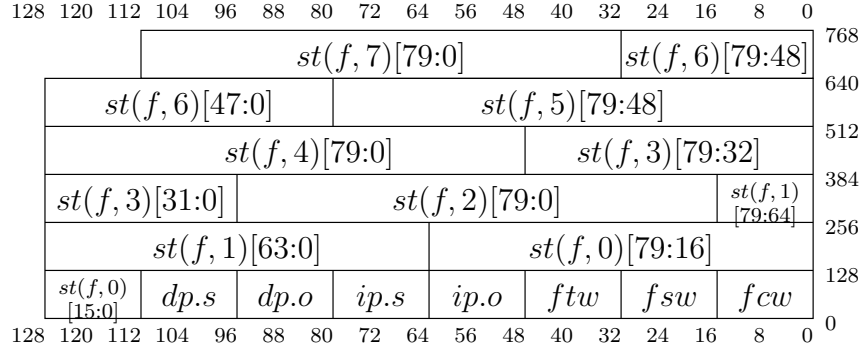


Figure C.3: x87 Environment image format for 16-bit protected mode

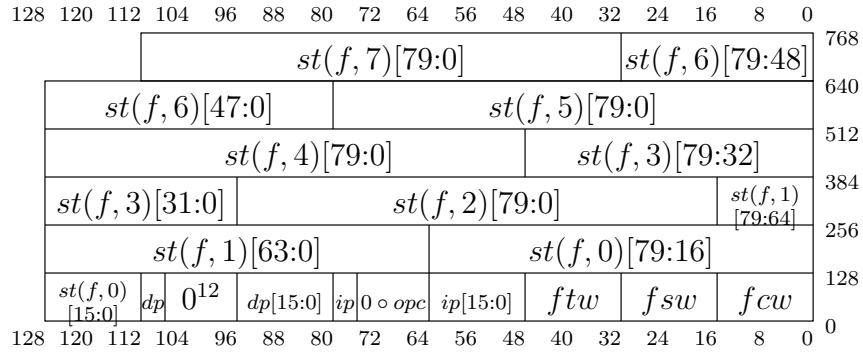


Figure C.4: x87 Environment image format for 16-bit real/virtual mode

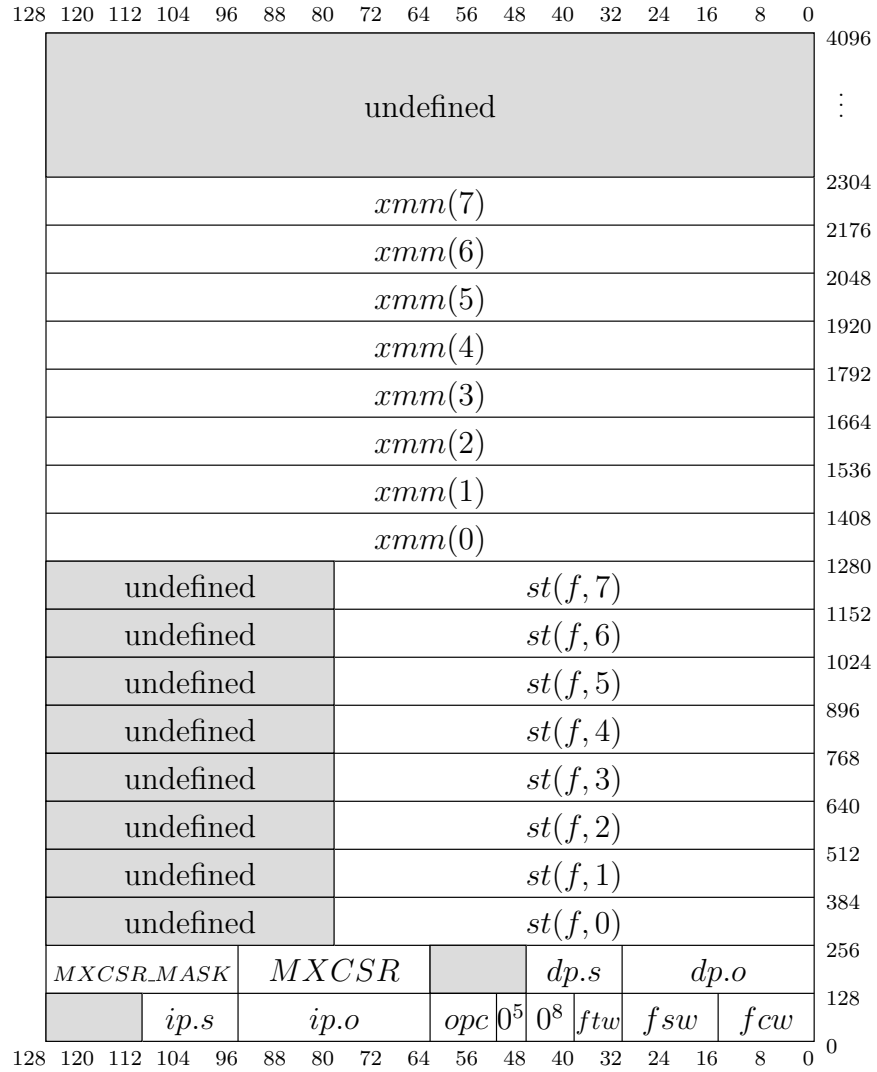


Figure C.5: XMM/x87/Media image format for non-64-bit mode

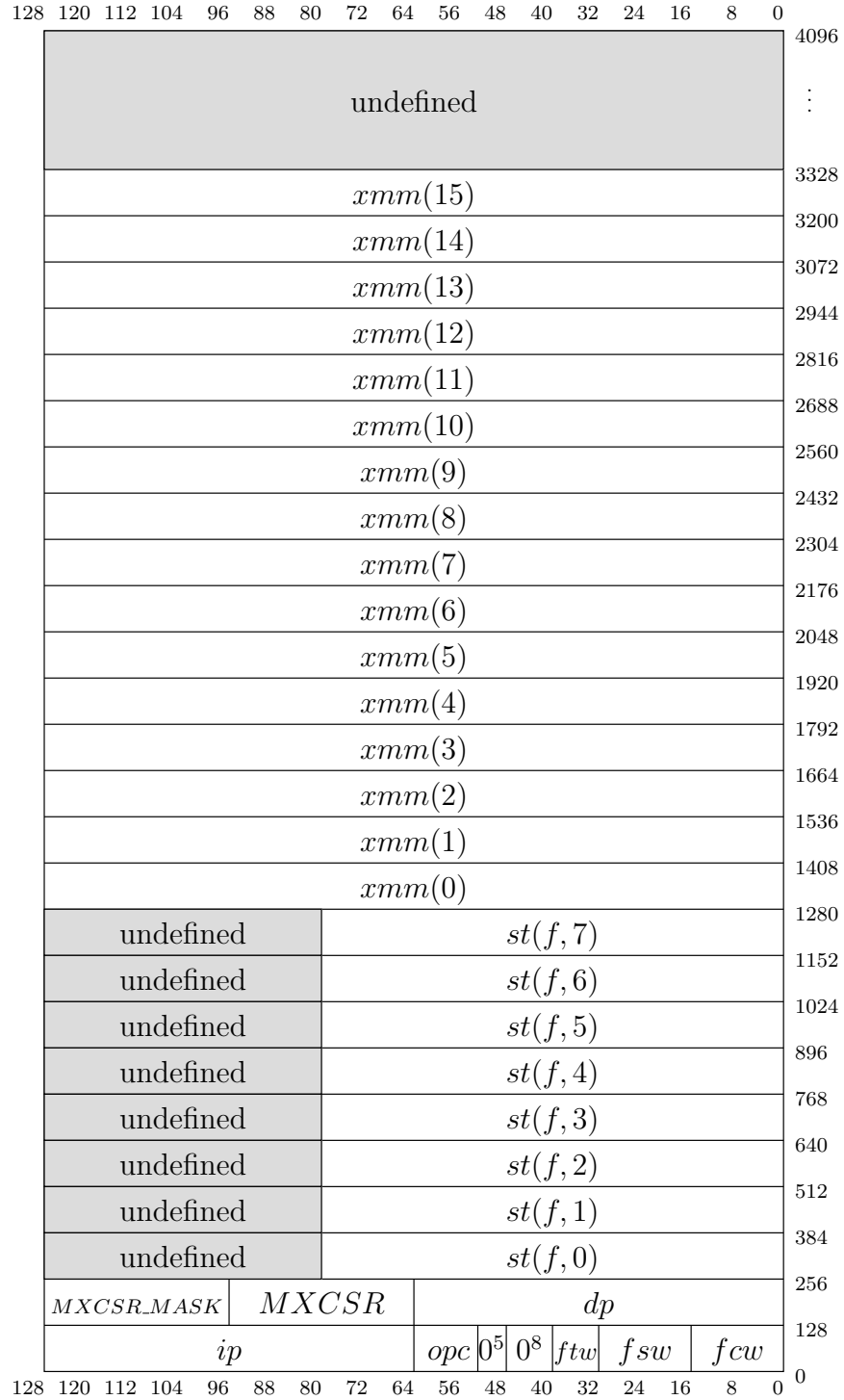


Figure C.6: XMM/x87/Media image format for 64-bit mode

Appendix D

Next State FPU Configuration

In the following we will summarise the components of the next state FPU configuration f' resulting from applying the FPU transition function to some floating-point configuration f .

$$f' = \sigma_{fpu}(f)$$

To this end we will use functions from this document. We will try to refer to the original configuration f as far as possible, nevertheless we will have to make cases distinctions on the exception status. This forces us to rely on the resulting FPU configurations of the pre- or post-computation exception checking phase.

$$f^{pre} = \sigma_{fpu}^{prechk}(f) \quad f^{post} = \sigma_{fpu}^{postchk}(\sigma_{fpu}^{exec}(f^{pre}))$$

We do not need to consider any other transition subfunctions. Additionally we want to state the initial configuration f^0 again.

$$\begin{aligned} \forall i \in \mathbb{N}_8 : \\ f^0.fpr(i) &= \text{undefined} \\ f^0.opc &= 0^{11} \\ f^0.dp &= 0^{64} \\ f^0.ip &= 0^{64} \\ f^0.fsw &= 0^{16} \\ f^0.fcw &= 0x0300 \\ f^0.ftw &= 1^{16} \\ f^0.acc &= \epsilon \\ f^0.ans &= \epsilon \end{aligned}$$

Now we want to list the contents for the several components of f' . We begin with the fields acc and ans , which are trivially defined.

$$\begin{aligned} f'.acc &= f'.acc \\ f'.ans &= \begin{cases} res_1(f^{pre}) : & xupdm(f) \wedge \overline{es(f^{post})} \\ f.ans : & \text{otherwise} \end{cases} \end{aligned}$$

The new values for the x87 exception pointers $f'.dp$, $f'.ip$ and $f'.opc$ can be found in table D.1.

The new x87 register contents $f'.fpr(i)$ for $i \in \mathbb{N}_8$ are given in table D.2.

The new register tags $f'.ftw[2j + 1 : 2j]$ for floating-point registers $j \in \mathbb{N}_8$ are listed in table D.3. Here the $i = ((j - \langle imgfsw(f)[13 : 11] \rangle) \bmod 8)$ represents the index of the stack element which shall be restored to register j from an FPU image.

The next state x87 control word is considered in table D.4.

Finally the new values for the several x87 status word components are contained in table D.5.

Table D.1 – x87 exception pointers

$f'.x$	content	conditions
$f'.dp$	$adjustdp(f)$	$nonctrl(I(f))$
	$imgdp(f)$	$imgload(f) \vee FXRSTOR(f)$
	0^{64}	$FNINIT(f) \vee FNSAVE(f)$
	$f.dp$	otherwise
$f'.ip$	$adjustip(f)$	$nonctrl(I(f))$
	$imgip(f)$	$imgload(f) \vee FXRSTOR(f)$
	0^{64}	$FNINIT(f) \vee FNSAVE(f)$
	$f.ip$	otherwise
$f'.opc$	$I(f).opc[18 : 16] \circ I(f).modrm$	$nonctrl(I(f))$
	$imgopc(f)$	$imgload(f) \vee FXRSTOR(f)$
	0^{11}	$FNINIT(f) \vee FNSAVE(f)$
	$f.opc$	otherwise

Table D.2 – x87 floating-point registers

$f'.x$	content	conditions
$f'.fpr(i)$	$res_1(f^{pre})$	$save?(f) \wedge i = sri(f, trgt_1(f)) \wedge \overline{es(f^{post})}$
		$push?(f) \wedge \overline{save?(f)} \wedge i = sri(f, 7) \wedge \overline{es(f^{post})}$
	$res_2(f^{pre})$	$push?(f) \wedge \overline{save?(f)} \wedge i = sri(f, 7) \wedge \overline{es(f^{post})}$
	$adjres_1(f^{pre})$	$ovf(f^{pre}) \wedge \overline{om(f)} \wedge \overline{xupdm(f)} \wedge \overline{es(f^{pre})} \wedge prechk(f^{pre}) \in \{\epsilon, DEN\} \wedge i = sri(f, trgt_1(f))$
		$unf(f^{pre}) \wedge \overline{um(f)} \wedge \overline{xupdm(f)} \wedge \overline{es(f^{pre})} \wedge prechk(f^{pre}) \in \{\epsilon, DEN\} \wedge i = sri(f, trgt_1(f))$
	$adjres_2(f^{pre})$	$unf(f^{pre}) \wedge \overline{um(f)} \wedge \overline{FSINCOS(f)} \wedge \overline{es(f^{pre})} \wedge prechk(f^{pre}) \in \{\epsilon, DEN\} \wedge i = sri(f, 7)$
	$imgstk(f)[80j + 79 : 80j]$	$(FRSTOR(f) \vee FXRSTOR(f)) \wedge i = sri(f, j)$
	$f.fpr(i)$	otherwise

Table D.3 – x87 tag word

$f'.x$	content	conditions
$f'.ftw[2j + 1 : 2j]$	$imgftw(f)[2j + 1 : 2j]$	$imgload(f)$
	11	$FNINIT(f) \vee FNSAVE(f)$
		$FXRSTOR(f) \wedge \overline{imgftw(f)[j]}$
		$FFREE(f) \wedge j = \langle opc(f)[2 : 0] \rangle$
		$pop?(f) \wedge j = sri(f, 0) \wedge \overline{es(f^{pre})}$
		$pop2?(f) \wedge j \in \{sri(f, 0), sri(f, 1)\} \wedge \overline{es(f^{pre})}$
	01	$FXRSTOR(f) \wedge imgftw(f)[j] \wedge$ $ imgstk(f)[80i + 79 : 80i] = 0$
		$save?(f) \wedge j = sri(f, trgt_1(f)) \wedge$ $\overline{es(f^{pre})} \wedge res_1(f^{pre}) = 0$
		$push?(f) \wedge \overline{save?(f)} \wedge j = sri(f, 7) \wedge$ $\overline{es(f^{pre})} \wedge res_1(f^{pre}) = 0$
		$push?(f) \wedge \overline{save?(f)} \wedge j = sri(f, 7) \wedge$ $\overline{es(f^{pre})} \wedge res_2(f^{pre}) = 0$
	10	$FXRSTOR(f) \wedge imgftw(f)[j] \wedge$ $(NaN(imgstk(f)[80i + 79 : 80i]) \vee$ $unsup(imgstk(f)[80i + 79 : 80i]) \vee$ $dnrml(imgstk(f)[80i + 79 : 80i]) \vee$ $infty(imgstk(f)[80i + 79 : 80i]))$
		$save?(f) \wedge j = sri(f, trgt_1(f)) \wedge \overline{es(f^{pre})} \wedge$ $(NaN(res_1(f^{pre})) \vee unsup(res_1(f^{pre})) \vee$ $dnrml(res_1(f^{pre})) \vee infty(res_1(f^{pre})))$
		$push?(f) \wedge \overline{save?(f)} \wedge j = sri(f, 7) \wedge \overline{es(f^{pre})} \wedge$ $(NaN(res_1(f^{pre})) \vee unsup(res_1(f^{pre})) \vee$ $dnrml(res_1(f^{pre})) \vee infty(res_1(f^{pre})))$
		$push?(f) \wedge \overline{save?(f)} \wedge j = sri(f, 7) \wedge \overline{es(f^{pre})} \wedge$ $(NaN(res_2(f^{pre})) \vee unsup(res_2(f^{pre})) \vee$ $dnrml(res_2(f^{pre})) \vee infty(res_2(f^{pre})))$
	00	$FXRSTOR(f) \wedge imgftw(f)[j] \wedge$ $e(imgstk(f)[80i + 79 : 80i]) \notin \{0^{15}, 1^{15}\}$
		$save?(f) \wedge j = sri(f, trgt_1(f)) \wedge$ $\overline{es(f^{pre})} \wedge e(res_1(f^{pre})) \notin \{0^{15}, 1^{15}\}$
		$push?(f) \wedge \overline{save?(f)} \wedge j = sri(f, 7) \wedge$ $\overline{es(f^{pre})} \wedge e(res_1(f^{pre})) \notin \{0^{15}, 1^{15}\}$
		$push?(f) \wedge \overline{save?(f)} \wedge j = sri(f, 7) \wedge$ $\overline{es(f^{pre})} \wedge e(res_2(f^{pre})) \notin \{0^{15}, 1^{15}\}$
	$f.ftw[2j + 1 : 2j]$	otherwise

Table D.4 – x87 control word

$f'.x$	content	conditions
$f'.fcw$	$imgfcw(f)$	$imgload(f) \vee FXRSTOR(f)$
	$mop(f)$	$FLDCW(f)$
	0x037F	$FNINIT(f) \vee FNSAVE(f)$
	$fcw[15 : 7] \circ 1^7$	$FNSTENV(f)$
	$f.fcw$	otherwise

Table D.5 – x87 status word

$f'.x$	content	conditions
$ie(f')$	0	$FNINIT(f) \vee FNSAVE(f) \vee FNCLEX(f)$
	1	$INV \in prechk(f) \vee STK \in prechk(f)$
	$imgfsw(f)[0]$	$imgload(f) \vee FXRSTOR(f)$
	$ie(f)$	otherwise
$de(f')$	0	$FNINIT(f) \vee FNSAVE(f) \vee FNCLEX(f)$
	1	$DEN \in prechk(f)$
	$imgfsw(f)[1]$	$imgload(f) \vee FXRSTOR(f)$
	$de(f)$	otherwise
$ze(f')$	0	$FNINIT(f) \vee FNSAVE(f) \vee FNCLEX(f)$
	1	$DBZ \in prechk(f)$
	$imgfsw(f)[2]$	$imgload(f) \vee FXRSTOR(f)$
	$ze(f)$	otherwise
$oe(f')$	0	$FNINIT(f) \vee FNSAVE(f) \vee FNCLEX(f)$
	1	$(calc(f) \vee store(f)) \wedge es(f^{pre}) \wedge prexcp(f^{pre}) \in \{\epsilon, DEN\} \wedge ovf(f^{pre})$
	$imgfsw(f)[3]$	$imgload(f) \vee FXRSTOR(f)$
	$oe(f)$	otherwise
$ue(f')$	0	$FNINIT(f) \vee FNSAVE(f) \vee FNCLEX(f)$
	1	$(calc(f) \vee store(f)) \wedge es(f^{pre}) \wedge prexcp(f^{pre}) \in \{\epsilon, DEN\} \wedge unf(f^{pre}) \wedge (inx(f^{pre}) \wedge um(f) \vee um(f))$
	$imgfsw(f)[4]$	$imgload(f) \vee FXRSTOR(f)$
	$ue(f)$	otherwise
$pe(f')$	0	$FNINIT(f) \vee FNSAVE(f) \vee FNCLEX(f)$
	1	$(calc(f) \vee store(f)) \wedge es(f^{pre}) \wedge prexcp(f^{pre}) \in \{\epsilon, DEN\} \wedge inx(f^{pre}) \wedge (xupdm(f) \wedge (ovf(f^{pre}) \wedge om(f) \vee unf(f^{pre}) \wedge um(f)))$
	$imgfsw(f)[5]$	$imgload(f) \vee FXRSTOR(f)$
	$pe(f)$	otherwise
$sf(f')$	0	$FNINIT(f) \vee FNSAVE(f) \vee FNCLEX(f)$
	1	$stkovf(f) \vee stkunf(f)$

Continued on next page

Table D.5 – x87 status word, continued from previous page

$x(f')$	content	conditions
	$imgfsw(f)[6]$	$imgload(f) \vee FXRSTOR(f)$
	$sf(f)$	otherwise
$es(f')$	$\bigvee_{i=0}^5 mca(f')[i]$	true
$busy(f')$	$es(f')$	true
$top(f')$	$top(f) -_3 bin_3(1)$	$push?(f) \wedge \overline{es(f^{pre})}$ $FDECSTP(f)$
	$top(f) +_3 bin_3(1)$	$pop?(f) \wedge \overline{es(f^{pre})}$ $FINCSTP(f)$
	$top(f) +_3 bin_3(2)$	$pop2?(f) \wedge \overline{es(f^{pre})}$
	0^3	$FNINIT(f) \vee FNSAVE(f)$
	$imgfsw(f)[13 : 11]$	$imgload(f) \vee FXRSTOR(f)$
	$top(f)$	otherwise
$cc(f')[0]$	0	$FNINIT(f) \vee FNSAVE(f)$ $prem(f) \wedge (Diff(f) \notin [0, 64)) \wedge \overline{es(f^{pre})}$ $prechk(f^{pre}) \in \{\epsilon, DEN\}$ $FXAM(f) \wedge \overline{empty(f, 0)} \wedge (unsup(op1(f)) \vee$ $\ op1(f)\ = 0 \vee dnrml(op1(f)) \vee e(op1(f)) \notin \{0^{15}, 1^{15}\})$
	1	$cmp(f) \wedge \overline{FXAM(f)} \wedge \overline{es(f^{pre})} \wedge \overline{xupdf(f)} \wedge \overline{stkunf(f)}$ $FXAM(f) \wedge (NaN(op1(f)) \vee infly(op1(f)) \vee \overline{empty(f, 0)})$
	$compcf(f)$	$cmp(f) \wedge \overline{FXAM(f)} \wedge \overline{es(f^{pre})} \wedge \overline{xupdf(f)} \wedge \overline{stkunf(f)}$
	$q(f)[2]$	$prem(f) \wedge (Diff(f) \in [0, 64)) \wedge \overline{es(f^{pre})}$ $prechk(f^{pre}) \in \{\epsilon, DEN\}$
	$imgfsw(f)[8]$	$imgload(f) \vee FXRSTOR(f)$
	undefined	$FNCLEX(f)$
	$cc(f)[0]$	otherwise
$cc(f')[1]$	0	$FNINIT(f) \vee FNSAVE(f)$ $stkunf(f)$ $prem(f) \wedge (Diff(f) \notin [0, 64)) \wedge \overline{es(f^{pre})} \wedge$ $prechk(f^{pre}) \in \{\epsilon, DEN\}$ $\overline{(prem(f) \wedge calc(f) \vee store(f) \wedge \overline{FST(f)} \vee \overline{FSTP(f)}) \wedge}$ $\overline{inx(f^{pre})} \wedge \overline{unf(f^{pre})} \wedge \overline{ovf(f^{pre})})$ $cmp(f) \wedge \overline{FXAM(f)} \wedge \overline{es(f^{pre})} \wedge \overline{xupdf(f)}$ $(load(f) \vee const(f)) \wedge \overline{STK} \notin prechk(f)$ $sign(f) \vee FDECSTP(f) \vee FINCSTP(f)$ $\vee FXCH(f) \vee FUCOM(f) \vee FUCOMP(f)$
	1	$stkovf(f)$
	$q(f)[0]$	$prem(f) \wedge (Diff(f) \in [0, 64)) \wedge \overline{es(f^{pre})} \wedge$ $prechk(f^{pre}) \in \{\epsilon, DEN\}$

Continued on next page

Table D.5 – x87 status word, continued from previous page

$x(f')$	content	conditions
	<i>roundup?</i> $(\llbracket res_1(f^{pre}) \rrbracket, pres_1(f^{pre}))$	$(calc(f) \wedge \overline{prem(f)} \vee store(f)) \wedge \overline{inx(f^{pre})} \wedge \overline{ovf(f^{pre})} \wedge \overline{unf(f^{pre})} \wedge \overline{es(f^{pre})} \wedge prechk(f^{pre}) \in \{\epsilon, DEN\}$
	<i>roundup?</i> $(\llbracket adjres_1(f^{pre}) \rrbracket, pres_1(f^{pre}))$	$(calc(f) \wedge \overline{prem(f)} \vee store(f)) \wedge \overline{xupdm(f)} \wedge (\overline{ovf(f^{pre})} \wedge \overline{om(f)} \vee \overline{unf(f^{pre})} \wedge \overline{um(f)}) \wedge \overline{es(f^{pre})} \wedge prechk(f^{pre}) \in \{\epsilon, DEN\}$
	$s(op1(f))$	$FXAM(f)$
	$imgfsw(f)[9]$	$imgload(f) \vee FXRSTOR(f)$
	undefined	$FNCLEX(f)$
	$cc(f)[1]$	otherwise
$cc(f')[2]$	0	$FNINIT(f) \vee FNSAVE(f)$
		$prem(f) \wedge (Diff(f) < 64) \wedge \overline{es(f^{pre})} \wedge prechk(f^{pre}) \in \{\epsilon, DEN\}$
		$(FSIN(f) \vee FCOS(f) \vee FSINCOS(f) \vee FPTAN(f)) \wedge \overline{es(f^{pre})} \wedge \llbracket op1(f) \rrbracket \in [-2^{63}, 2^{63}] \wedge \overline{stkunf(f)}$
		$FXAM(f) \wedge (NaN(op1(f)) \vee unsup(op1(f)) \vee \llbracket op1(f) \rrbracket = 0 \vee empty(f, 0))$
	1	$(FSIN(f) \vee FCOS(f) \vee FSINCOS(f) \vee FPTAN(f)) \wedge \overline{es(f^{pre})} \wedge (\llbracket op1(f) \rrbracket \notin [-2^{63}, 2^{63}] \wedge \overline{stkunf(f)} \vee stkunf(f))$
		$prem(f) \wedge (Diff(f) \geq 64) \wedge \overline{es(f^{pre})} \wedge prechk(f^{pre}) \in \{\epsilon, DEN\}$
		$FXAM(f) \wedge \overline{empty(f, 0)} \wedge (infity(op1(f)) \vee dnrml(op1(f)) \vee e(op1(f)) \notin \{0^{15}, 1^{15}\})$
		$cmp(f) \wedge \overline{FXAM(f)} \wedge \overline{es(f^{pre})} \wedge \overline{xupdf(f)} \wedge \overline{stkunf(f)}$
	$comppf(f)$	$cmp(f) \wedge \overline{FXAM(f)} \wedge \overline{es(f^{pre})} \wedge \overline{xupdf(f)} \wedge \overline{stkunf(f)}$
	$imgfsw(f)[10]$	$imgload(f) \vee FXRSTOR(f)$
$cc(f')[3]$	0	$FNINIT(f) \vee FNSAVE(f)$
		$prem(f) \wedge (Diff(f) \notin [0, 64)) \wedge \overline{es(f^{pre})} \wedge prechk(f^{pre}) \in \{\epsilon, DEN\}$
		$FXAM(f) \wedge \overline{empty(f, 0)} \wedge (unsup(op1(f)) \vee NaN(op1(f)) \vee infity(op1(f)) \vee e(op1(f)) \notin \{0^{15}, 1^{15}\})$
	1	$cmp(f) \wedge \overline{FXAM(f)} \wedge \overline{es(f^{pre})} \wedge \overline{xupdf(f)} \wedge \overline{stkunf(f)}$
		$FXAM(f) \wedge (dnrml(op1(f)) \vee \llbracket op1(f) \rrbracket = 0 \vee empty(f, 0))$
	$compzf(f)$	$cmp(f) \wedge \overline{FXAM(f)} \wedge \overline{es(f^{pre})} \wedge \overline{xupdf(f)} \wedge \overline{stkunf(f)}$
	0	$FNINIT(f) \vee FNSAVE(f)$
		$prem(f) \wedge (Diff(f) \notin [0, 64)) \wedge \overline{es(f^{pre})} \wedge prechk(f^{pre}) \in \{\epsilon, DEN\}$
		$FXAM(f) \wedge \overline{empty(f, 0)} \wedge (unsup(op1(f)) \vee NaN(op1(f)) \vee infity(op1(f)) \vee e(op1(f)) \notin \{0^{15}, 1^{15}\})$
	1	$cmp(f) \wedge \overline{FXAM(f)} \wedge \overline{es(f^{pre})} \wedge \overline{xupdf(f)} \wedge \overline{stkunf(f)}$

Continued on next page

Table D.5 – *x87 status word, continued from previous page*

$x(f')$	content	conditions
	$q(f)[1]$	$prem(f) \wedge (Diff(f) \in [0, 64)) \wedge \overline{es(f^{pre})}$ $prechk(f^{pre}) \in \{\epsilon, DEN\}$
	$imgfsw(f)[14]$	$imgload(f) \vee FXRSTOR(f)$
	undefined	$FNCLEX(f)$
	$cc(f)[3]$	otherwise

Bibliography

- [AMD07] *AMD64 Architecture Programmer's Manual*, September 2007.
- [Deg07] Ulan Degenbaev. Formal specification of parts of the x86-64 instruction set architecture. Master's thesis, Saarland University, 2007.
- [IA07] *Intel 64 and IA-32 Architectures Software Developer's Manual*, November 2007.
- [IE85] *IEEE standard for binary floating-point arithmetic*, August 1985.
- [MP00] S.M. Müller and W.J. Paul. *Computer Architecture, Complexity and Correctness*. Springer Verlag, 2000.