

**Towards Comparability
in Evaluating the Fault-Tolerance
of Safety-Critical Embedded Software**

Vom Fachbereich Ingenieurwissenschaften der
Universität Duisburg–Essen
zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigte Dissertation

von

Stefan Freinatis

aus

Wolfsburg

Referent: Prof. Dr.-Ing. Axel Hunger
Korreferent: Prof. Dr. rer. nat. Maritta Heisel
Tag der mündlichen Prüfung: 23.02.2005

To my parents

When you think you are done, you have just begun

Acknowledgement

My sincere gratitude goes to my supervisor Prof. Dr.-Ing. Axel Hunger who gave me the opportunity to perform research in a highly interesting and present-day field. I am also grateful to my co-supervisor Prof. Dr. rer. nat. Maritta Heisel for her valuable comments on the research area in general and her constructive remarks on the thesis in particular.

Obliging thanks also go to the other members of the examination board, Prof. Dr. rer. nat. Wolfram Luther, Prof. Dr.-Ing. habil. István Erlich and Prof. Dr.-Ing. Andreas Czulwik for a well-balanced and fair examination. This of course applies to my supervisor and co-supervisor as well.

I would also like to express my warmest thanks to my colleagues at the institute for their support and their contribution to an inspiring and pleasant atmosphere. Especially I would like to thank Dipl.-Ing. Joachim Zumbrägel for our philosophic discussions about the meaning of life (and of PhD theses in particular), Dr.-Ing. Stefan Werner for guiding me through the final process by means of valuable discussions, and Dipl.-Ing. Christian Schütz for his cooperativeness and our nice evenings at the “Bürgerhof” pub. Mrs Renée Foraschick has been a gentle supporting pillar in the background. She helped me clearing up a number of perils of the English Language. Thank you very much.

Last but not least I would like to express my heartfelt gratitude to my family and good friends. They have been lenient and have accompanied me with tolerance during my doctorate. In this spirit my thanks also go to Fred Knief, Dietz Jansen and Gabi Gernholz.

Finally, and not a joke, I am thankful to Daniel Goleman for his book “Emotional Intelligence” and to Earth, Wind & Fire for their still great music. Danke schön.

Duisburg/Uelsen, May 2005.

Stefan Freinatis

This document is also available at www.freinatis.de/thesis.

Towards Comparability
in Evaluating the Fault-Tolerance
of Safety-Critical Embedded Software

Stefan Freinatis

October 2004

Abstract

This thesis deals with the problem of obtaining meaningful and comparable dependability measures of software through the method of fault-injection. The thesis is specifically dedicated to safety-critical embedded software and its dependability property ‘fault-tolerance’ with respect to random faults affecting the machine instruction execution. For enabling comparability of the fault-tolerance of different software on different hardware, a fault-injection method is derived and presented.

The method is based on the idea of the so-called *FARM* sets which was originally presented for the characterization of physical fault-injection experiments. The collection of sets is broadened and adjusted to the herein considered object of evaluation ‘software in execution’. The software is thereby conceived as a process, and the fault set F is devised accordingly by means of a universal microprocessor model. The resulting set is software-overlapping and forms a mutual basis regarding the fault input among the software. In conjunction with the other sets presented, a fault-injection method allowing for – as far as possible – comparable fault-tolerance measures is constructed.

Therewith is created a methodical fundament for fault-injection experiments that aim at evaluating the fault-tolerance of safety-critical embedded software affected by hardware faults.

Keywords: embedded system, dependability, fault-tolerance, fault-injection, *FARM*, service-provider.

Contents

List of Tables	xvii
List of Figures	xix
1 Introduction	1
1.1 Motivation	1
1.2 Problem	2
1.3 Goal of the Work	3
1.4 Background of the Work	3
1.5 Overview	4
1.6 Notions	5
1.6.1 Safety-Critical Embedded System	5
1.6.2 Software	6
1.6.3 Program	6
1.6.4 Process	6
1.6.5 Hardware	7
1.6.6 Fault Notions	8
1.6.7 Safety	10
1.6.8 Fault-Tolerance	10
2 Fault-Injection for Software Dependability Evaluation	13
2.1 Introduction	13
2.2 Injection-Techniques	14
2.2.1 Physical Fault-Injection	15
2.2.2 Software Implemented Fault-Injection (SWIFI)	18
2.2.3 Simulation-Based Fault-Injection	19
2.2.4 Summary	21
2.3 State of the Art	22
2.3.1 Related Work	23

2.3.2	Limitations	24
2.3.3	Summary	26
2.4	Comparability	27
2.4.1	The <i>FARM</i> Sets	27
2.4.2	Prominent Example	28
2.4.3	Requirement Identification	29
2.4.4	Summary Requirements	33
2.5	Summary	34
3	The Controlling Process	35
3.1	Introduction	35
3.2	Naming	35
3.3	Components	36
3.3.1	Binary Program	36
3.3.2	Processing Hardware	37
3.4	Definition	40
3.5	Fault-Tolerance Categories	41
3.5.1	Exterior Fault-Tolerance	41
3.5.2	Interior Fault-Tolerance	42
3.5.3	Distinction	44
3.5.4	Online Error Detection	46
3.6	Injection Method	47
3.6.1	Process Fault-Injection	47
3.6.2	Process Fault	47
3.6.3	Process Error	48
3.6.4	Process Failure	48
3.6.5	Summary	49
3.7	Summary	50
4	Microprocessor Modeling	51
4.1	Introduction	51
4.2	Register Model	51
4.3	The Service-Provider Model	53
4.3.1	Introduction	53
4.3.2	The Storage Space	56
4.3.3	The Services	57
4.3.4	A Z80 Example	59
4.3.5	Definitions	60

4.3.6	Gate Level Model and Service-Provider Model	63
4.3.7	Discussion	63
4.4	The Service-Provider Error Model	64
4.4.1	Storage Space	64
4.4.2	Services	65
4.4.3	Fault Propagation	66
4.4.4	Discussion	71
4.5	Summary	71
5	Service Errors	73
5.1	Introduction	73
5.2	Fault Mapping	73
5.2.1	The ‘real’ faults	75
5.2.2	Related Work	76
5.2.3	State of the Art	77
5.3	Error Behavior	78
5.3.1	Error Classes	78
5.3.2	Timing Errors	79
5.3.3	Multiple Affections	79
5.4	Error Behavior of Combinational Circuits	80
5.4.1	Retrospective	81
5.4.2	Introduction	82
5.4.3	Gate Level Fault-Simulation	84
5.4.4	Error Model	85
5.4.5	The Circuits	94
5.4.6	Single-Fault Error Behavior	100
5.4.7	Double-Fault Error Behavior	106
5.4.8	Summary	109
5.5	Creating Realistic Service Errors	112
5.6	Urging Manufacturers	114
5.7	Summary	115
6	Mutant-Injection	117
6.1	Introduction	117
6.2	Fault Set F	118
6.2.1	Services	119
6.2.2	Mutants	119
6.3	Notions N	120

6.3.1	Fault, Error, Failure	120
6.3.2	Error Detection	122
6.3.3	Fault-Tolerance	122
6.3.4	Fault Scenario	125
6.3.5	Error Scenario	126
6.4	Activations A and Readouts R	126
6.5	Predicates P	127
6.5.1	Fault Activation	127
6.5.2	Fault Effectiveness	128
6.5.3	Error Activation	129
6.5.4	Error Masking	130
6.5.5	Error Blanking	130
6.5.6	Error Propagation	131
6.5.7	Error Releasing	132
6.5.8	Error Detection	133
6.5.9	Error Signaling	134
6.5.10	Fault-Tolerance	135
6.6	Valuation Rules V	135
6.6.1	Period of Grace	135
6.6.2	Distinct Error Scenarios	136
6.6.3	Redundant Experiments	137
6.6.4	Relevance of Errors	137
6.7	General Issues	140
6.7.1	Observation interval	140
6.7.2	Error Proliferation	140
6.7.3	Golden Run	142
6.7.4	Divide and Conquer	144
6.8	Measures M	145
6.8.1	Coverage Proportion	145
6.8.2	Time Intervals	146
6.8.3	Interior Hardware-Fault Fault-Tolerance	146
6.9	Fault-Injection Environment	148
6.9.1	Simulation-Based Fault-Injection	148
6.9.2	A Simulator Concept	149
6.9.3	Requirements	151
6.9.4	Benefits	152
6.9.5	Feasibility and Availability	154

6.10	Summary	155
6.11	Discussion	156
6.11.1	Summary Mutant-Injection	156
6.11.2	Existing Approaches	158
6.11.3	Remaining Problems	160
7	Summary	163
7.1	Problem Recall	163
7.2	Review of Chapters	164
7.3	Summary of the Work	166
7.4	Prospects	170
A	Error Distribution Figures	171
A.1	Single-Fault Error Distributions	171
A.2	Double-Fault Error Distributions	188
	References	201
	Index	208

List of Tables

1.1	Program abstraction levels	7
1.2	Microprocessor abstraction levels	7
2.1	Attributes of hardware fault-injection techniques	22
3.1	Fault-tolerance categories of the controlling process	45
4.1	Operands of the Z80 service ADD A,B	60
5.1	Summary fault-simulation procedure	85
5.2	Bit-flips in the output of the 4-Bit Adder	87
5.3	Circuit data ADDER4	100
5.4	Circuit data	102
5.5	Summary simulation results (Part 1 of 2)	110
5.6	Summary simulation results (Part 2 of 2)	111

List of Figures

1.1	Stylized plot of a safety-critical embedded system	6
1.2	Exemplary microcontroller	8
1.3	Fault set F_0	9
1.4	Exterior and interior origin of fault effects to be tolerated	11
1.5	Fault-tolerance and safety	12
2.1	The <i>FARM</i> sets in fault-injection	28
2.2	The two processes in fault-injection	30
2.3	Accumulating fault effects in a common F set	31
3.1	Processing hardware involved in <code>ADD mem1,mem2</code>	38
3.2	Random fault set R	39
3.3	The controlling process (stylized)	41
3.4	On the exterior fault-tolerance	42
3.5	On the interior fault-tolerance	43
3.6	Path of impact	44
4.1	Traditional view onto microprocessor activities	53
4.2	Register model partly intercepting the path of impact	54
4.3	The notion of a Service	55
4.4	Service oriented view onto a microprocessor	55
4.5	Storage space and services	57
4.6	Service modifying the storage space	58
4.7	Sample specification of the Z80 service <code>ADD A,B</code>	61
4.8	Controlling process operating on the storage space	62
4.9	Structural elements at gate level and service level	63
4.10	Service-provider model intercepting the path of impact	66
4.11	Hardware faults during service <code>ADD mem1,mem2</code>	67
4.12	Manifestation of hardware faults	69

4.13	Masking and blanking a data fault	69
4.14	Service errors are the fault effect entry points	70
5.1	Propagation path towards controlling process	74
5.2	Transformations within a service	78
5.3	Multiple affection example	80
5.4	Addition $R:=A+B$	82
5.5	Circuit constellation	83
5.6	Circuit diagram 4-bit-Adder ADDER4	88
5.7	The arithmetic error e on the number circle.	90
5.8	Arithmetic error distribution ADDER4	91
5.9	Error distribution ADDER4 (with embedded table)	93
5.10	Schematic 4-Bit Carry-Look-Ahead Adder CLA-ADDER4	95
5.11	Schematic 4-Bit ALU ALU4	96
5.12	Schematic ALU74181	97
5.13	Schematic Barrelshifter BS4	98
5.14	Schematic Multiplier MUL4	99
5.15	Arithmetic error distribution ADDER4 (final version)	101
5.16	Single-fault error distribution CLA-ADDER4	103
5.17	Single-fault error distribution ALU4 ($R := A + B$)	103
5.18	Single-fault error distribution ALU74181 ($R := A + B$)	104
5.19	Single-fault error distribution ALU8 ($R := A + B$)	104
5.20	Single-fault error distribution ALU8 ($R := A \wedge B$)	105
5.21	Single-fault error distribution MUL4 ($R := A \cdot B$)	105
5.22	Double-fault error distribution 4-bit Adder	107
5.23	Double-fault error distribution 4-bit Carry-Look-Ahead Adder	107
5.24	Double-fault error distribution ALU4 ($R := A + B$)	108
5.25	Double-fault error distribution ALU74181 ($R := A + B + 1$)	108
5.26	Service error on MUL A,B	113
5.27	Service error on PUSH A	114
6.1	Mutant-injection for evaluating the controlling process	118
6.2	The fault set F (Mutants)	121
6.3	Masking analogy	123
6.4	Blanking	124
6.5	Activation of mutant	128
6.6	Non-effective mutant	128
6.7	Effective mutant	129

6.8	Error dormant over two periods	129
6.9	Masking	130
6.10	Blanking	130
6.11	Error propagation analogy	131
6.12	Propagation of error e_i	131
6.13	Another propagation example	132
6.14	Error propagating into output area	133
6.15	Error detection analogy	133
6.16	Period of grace	136
6.17	Two relevant temperature states	138
6.18	Error relevances	139
6.19	Characterization of an experiment [Ar90]	140
6.20	Error proliferation	141
6.21	Parallel fault-simulation	142
6.22	Concurrent processing	143
6.23	Investigation of partial process z_i	144
6.24	Sample distribution (cumulative)	146
6.25	Simulator concept	150
6.26	Shadow program through misalignment	154
7.1	Summary Mutant-injection	167

Chapter 1

Introduction

1.1 Motivation

The father of a three-month old child went to his authorized garage in order to get the front-seat passenger airbag of his car disabled. There, the airbag was electronically deactivated in the airbag control device. Some weeks later, with the child seated in a child seat on the front passenger's seat, the man had a rather insignificant rear-end collision. The front passenger's airbag went off and killed the child [MW99 p.22]. As it turned out later, the airbag had been correctly disabled by the garage. The disabling indicators could still be read from the control-unit after the accident. Somehow, the software of the control-unit must have ignored the indicators and fired the airbag inadmissibly.

To date, many mechanical control systems have been replaced by micro-processor controlled safety-critical embedded systems. A malfunction of the system may cause harm to the users, to the public or to the environment. The prior goal in system development therefore is to assure a high degree of safety during operation. Safety is achieved through fault-tolerance. For cost reasons and lack of space, hardware-based fault-tolerance mechanisms have been more and more replaced by software-based mechanisms. A large portion of the responsibility has now shifted to the software part. Safety-critical embedded software therefore is facing high demands on its fault-tolerance.

Software may fail on account of several causes. Software faults can be one of the causes. The Therac-25 case [Le95 p.515] and the Ariane 5 [Nu97 p.15]

accident are prominent warnings of incorrect software. Another reason for a failure can be random hardware faults. These may occur in any hardware component of the system. Of special concern are those random faults that occur in the *processing hardware* which is the hardware that is directly involved in the execution of the individual machine instructions. Since the application level functionality of the software depends on a program-conform execution, the fault-tolerance of the software with respect to these particular faults is vital for safety.

The development of safety-critical embedded software is not the only challenging task, but also putting the fault-tolerance capabilities into concrete terms. According to [St96], software must not only *be* fault-tolerant, it must be *shown* to be fault-tolerant. This is especially true when it comes to legal aspects or to the process of gaining certification. For that, the fault-tolerance of the software needs to be evaluated and put into meaningful and comparable measures. Fault-tolerance evaluation, a subject within the scope of dependability evaluation, can be done through static and dynamic analyses. With respect to software and random faults, the assessment is performed through a dynamic analysis method. In contrast to dynamic analyses, static analyses evaluate the program rather than the software in execution. The common dynamic analysis method used in dependability evaluation is hardware fault-injection.

1.2 Problem

Hardware fault-injection is an approved method for evaluating – and literally showing – the fault-tolerance capabilities of software. However, when related to malfunctions of the microprocessor, fault-injection faces problems. Often there is no suitable error model of the target processor available that describes its malfunctioning from a strict software application perspective. From there, the approaches taken in the fault-injection experiments are rather customized solutions. In most cases the approaches are not compatible with each other. The measures obtained from one fault-injection approach cannot be used for a meaningful comparison with those obtained by other approaches. One problem lies in the often vague or missing specification of the object of investigation ‘software in execution’ and its clear demarcation from the remaining system. Another problem is the different fault input used in the experiments. If the fault input is not comparable, the results are not comparable. Diverse perceptions of what a representative fault is, what the presence of a fault actually

means, and how the readings finally shall enter evaluation represents another problem. The varied interpretations on common notions moreover complicate the rating of the results. A mutual basis that allows for comparability is missing. This thesis takes a step towards comparability in fault-injection, however with special focus on safety-critical embedded software and random faults in the processing hardware.

1.3 Goal of the Work

Fault-injection, strictly spoken, is not concerned with why certain fault events may occur or how representative the injected faults are [cf. Vo98 p.25]. Regardless of how realistic the injected faults may be considered, by following a standardized evaluation method no further blurring should be added to the results. A clear fault input, irrespective of its plausibility, should result in a clear and comparable output.

The goal of this thesis is to provide a methodical fundament for those fault-injection experiments that aim at evaluating the fault-tolerance of safety-critical embedded software with respect to random hardware faults affecting the machine instruction execution. The silent background question is: “How can – by means of fault-injection – software for different safety-critical embedded systems be evaluated for its fault-tolerance, such that the obtained measures are – as far as possible – comparable?”. The herein presented fault-injection method tries to give an answer. Some problems, however, cannot be solved in this work. This thesis therefore does not end up in a ready-to-go evaluation manual, but is also considered as a basis for discussion towards more conformance in fault-injection based fault-tolerance evaluation of safety-critical embedded software.

1.4 Background of the Work

The basis of this thesis was build when the author was involved in a dependability evaluation project of safety-critical embedded software. The software had been developed for microprocessor based engine-controls to be used in the latest drive-by-wire generation of automobiles. One subject within the project was the evaluation of the fault-tolerance of the software with respect to hardware faults affecting the machine instruction execution. The analysis method

was defined as fault-injection. Of the microcontroller (the Infineon 80C167) only the conventional documentation was available. No low-level models were at disposal. The author developed a register model simulator for the microprocessor core [Fr98]. Fault-injection was then performed by injecting state errors into the memory – following similar approaches published in the fault-injection community. However, since there was no standardized evaluation procedure, the approach as well resulted in a customized solution, and the derived measures were merely of in-house expressiveness [Fr99a, Fr99b].

This thesis is not a report of the project. It is the further development of the understandings obtained during the project.

1.5 Overview

In the remainder of this chapter the type of considered safety-critical embedded systems is outlined and basic notions are restated. In particular the term fault-tolerance in its significance for safety-critical systems is closer specified.

Chapter 2 is dedicated to fault-injection for software dependability evaluation. The customary hardware fault-injection techniques are presented and discussed. Related publications are reviewed and the state of the art is concluded. Comparability aspects are addressed thereafter. Based on the *FARM* sets of [Ar90], the major requirements for more comparability in evaluating safety-critical embedded software through fault-injection are identified. The pivotal point in the fault-injection method is a common and object-appropriate fault set F .

In Chapter 3 the software in execution is specified as the *controlling process*. The two components establishing the controlling process – the processing hardware and the binary program – are outlined. A distinction of the fault-tolerance categories of the controlling process is given, and the *interior hardware-fault* fault-tolerance is identified as the category of concern. The conceptual injection method is then determined as *process* fault-injection.

Chapter 4 presents the *service-provider* model. Its components, the *storage space* and the *services* are discussed. In the error model section, the principal effects of random faults onto the model components are presented. It is demonstrated that the service-provider model fully intercepts the fault propagation path from the hardware to the controlling process. The services are identified as the entry points of hardware fault effects into the controlling pro-

cess. Service errors are in particular the representatives of the random faults in the processing hardware.

Chapter 5 deals with *service* errors. The field of fault mapping is looked at, and publications concerned with microprocessor error behavior modeling are reviewed. The fundamental error behavior of services is addressed then. In the major part of the chapter, the error behavior of selected combinational circuits is investigated through fault-simulation, and is presented by means of *arithmetic* error distributions. It is shown that the *power-of-two* errors are the most realistic in the presence of single and double-faults. An application example of these findings for the creation of representative service errors is given.

In Chapter 6, the requirements from Chapter 2 are put in concrete terms, resulting in the herein presented method *mutant-injection*. Mutant-injection encompasses the actual injection process and the valuation of the results obtained from the experiments. The sets characterizing the method are discussed subsequently. The concept of a simulation-based fault-injection environment for real-time execution, injection and observation is presented then. Finally, the method is summarized and discussed.

Chapter 7 restates the problem and summarizes the work.

1.6 Notions

1.6.1 Safety-Critical Embedded System

The following definition depicts the type of safety-critical embedded system that is considered throughout this thesis.

Definition 1.1: A *safety-critical embedded system* is a controlling system that is operated mainly by a single microcontroller. Its is small in size and low in cost. Its presence is largely invisible to the outside world and its correct behavior is vital.

Safety-critical embedded systems have no user interface (keyboard, display). There is no supervising operator or maintenance personnel during operation. The major application field is that of real-time applications. Figure 1.1 shows a stylized hardware view of a typical safety-critical embedded system. The microcontroller μC is the main hardware component, often these are COTS (Commercial Off The Shelf) components, such as the Infineon 80C167, 80C51 or the Motorola 68HC05.

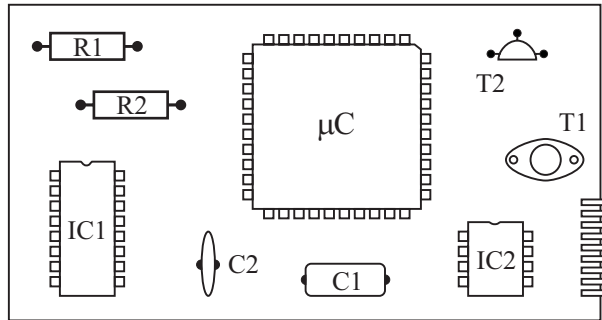


Figure 1.1: Stylized plot of a safety-critical embedded system

1.6.2 Software

The term software has at least two meanings. In expressions such as “writing software” or “software fault”, the term software actually means *program*. Software then is the conceptual counterpart of the notion of hardware. In expressions such as “software failure” or “fault effects on software” the term software depicts a *process*. The term software will be used in this thesis with both meanings, the particular meaning depends on the context.

1.6.3 Program

A program is a somehow ordered sequence of instructions. A program is static, and technically speaking it cannot ‘crash’ nor can it be safe or unsafe [cf. Vo98 p.161]. Any considerations on the behavior of a program are shifted here to the notion of a *process*. A program is denoted by P . In particular the binary machine program loaded into the safety-critical embedded system is denoted by P_M .

1.6.4 Process

According to the dictionary, a process is a connected series of actions, changes, or functions bringing about a result. In [Kaw95] a process is defined as “a set of identifiable, repeatable actions which are ordered in some way and contribute to the fulfillment of an objective”. Creating a process requires some executing device and some kind of program. In this thesis the safety-critical embedded

Level	Structural Primitives
Application	Behavioral rules, functions and functionality
Conceptual	Closed and open loops, algorithms, formulas, data structures
High level language	Statements, procedures, classes, data types
Assembly	Mnemonics
Machine	Binary code words, bit patterns, bits

Table 1.1: Program abstraction levels

software in action will be referred to as *controlling process* (Chapter 3). Contrary to a program, a process can ‘crash’, and it can be attributed to be safe or unsafe — respectively to be fault-tolerant or not.

1.6.5 Hardware

The processors used in safety-critical embedded systems are modern microcontrollers which are incorporating a variety of peripheral subsystems on the same chip. An exemplary block diagram of a typical microcontroller is shown in [Figure 1.2](#). The CPU contains the central control-logic and the common functional units, such as the ALU and the multiplication unit. State variables as well as important registers are stored in a small local RAM area. The main memory comprises the internal RAM and ROM, and can be extended through external memory components. The binary program P_M usually is stored in the ROM. The processing hardware (specification in Chapter 3) denotes those hardware areas that are transforming the binary machine instructions into their associated tasks. The peripheral subsystems shown in the figure are

Level	Structural Primitives
Functional	Constraints
Register	Registers, flags, memory
Register Transfer Level (RTL)	Buses, functional units (e.g. ALU)
Gate	Gates, flip-flops
Device	Transistors, R, L, C
Physical	Atoms, crystal structures

Table 1.2: Microprocessor abstraction levels

examples of customary subsystems found in many microcontrollers. These include AD-converters, timers, pulse-width-modulators and serial communication interfaces. The hardware abstraction levels are listed in [Table 1.2](#).

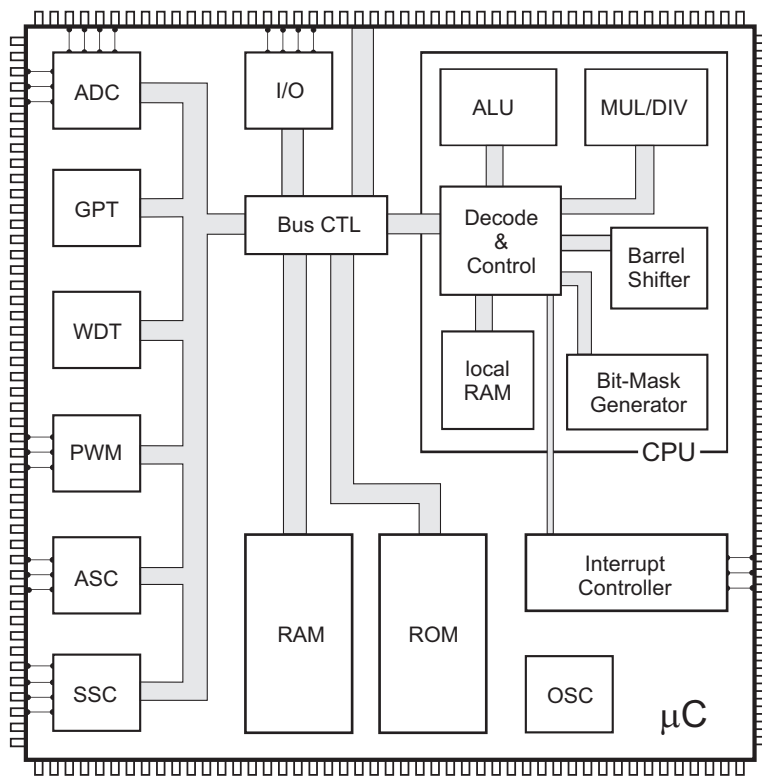


Figure 1.2: Exemplary microcontroller

1.6.6 Fault Notions

A fault is a defect in the program [Vo98 p.37] or in the system [St96 p.12]. In [CI95 p.48] a fault is considered a deviation of a component from its intended function. A fault may also be a preanomaly event [Vo98 p.40]. Interaction faults and design faults have as well joined the concept of faults [Av97 p.52]. The following definition, slightly modified from [St95 p.187], tries to be universal.

Definition 1.2: A *fault* is the adjudged or hypothesized cause of a problem.

The collection of faults that a safety-critical system may be up against is denoted by the set F_0 . The faults can be categorized into systematic faults

and random faults. Systematic faults are permanently inherent to a system. They are caused by mistakes or shortcomings in the development process. Software faults and design flaws belong to the systematic faults.

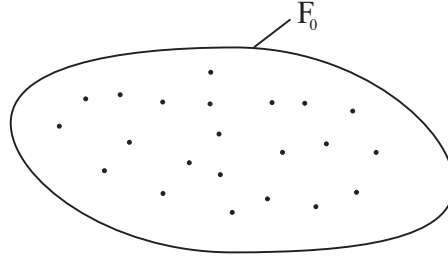


Figure 1.3: Fault set F_0

Random faults are those faults whose presence, location and duration are not predictable. Human interaction faults might have to be added to the type of random faults from a strictly objective point of view. However, the term random fault has become a common expression that is associated with hardware defects.

Definition 1.3: A *random fault* is a physical malfunction in the hardware, caused by a natural phenomenon of internal or external origin. Its location and moment of occurrence is unpredictable. Its duration may be permanent, intermittent or transient.

The presence of a fault may lead to an error. In [St96 p.12] an error is defined as a deviation from the required operation, and is considered the mechanism by which a fault becomes apparent. In this sense an error is behavioral and denotes a perceptible activity that is caused by a fault. Other authors consider an error to be a “static condition” [Le95 p.172]. For example, in [Wa78 p.9] an error is an incorrect output. In this thesis the term error refers to both — the static result of a fault as well as the dynamic consequence of a fault. In any case, an error is the manifestation of an *active* fault and usually implies that a constraint (e.g. required operation, correct output) is violated.

Definition 1.4: An *error* is the static or dynamic manifestation of an active fault. An error violates a given constraint.

Sometimes the term *fault effect* is used as a synonym for an error. The difference between an error and a fault effect is that the latter does not violate

a constraint. A fault effect is the silent and unjudged effect of a fault. Fault propagation often means the propagation of fault effects. A fault effect may vanish or turn into an error. Errors may induce failures which are deviations from the appropriate behavior of a system or object. A failure is commonly defined to be an event.

Definition 1.5: A *failure* is the nonperformance or inability of a system or object to perform its intended function for a specified time.

1.6.7 Safety

An inherent goal of any safety-critical system is that the occurrence of a fault may not result in a dangerous situation. The system must be safe, at least to a certain degree.

Definition 1.6: *Safety* is the property of a system, not to harm humans or objects that are not intended to be harmed.

Safety is not an absolute and naturally defined property, and there is no such thing as a totally safe system. The safety of a system is always related to a certain fault set F_0 . The safety measure $S_{sys}(F_0)$ is the measure of fulfillment of the system with respect to the faults F_0 . During operation safety is achieved through fault-tolerance.

1.6.8 Fault-Tolerance

In order to maintain safe behavior, the system must be able to somehow cope with faults that occur during operation. The property of an operational system to tolerate faults (or fault effects) is described by its fault-tolerance.

Fault-tolerance is the ability of a system or component to continue normal operation despite the presence of hardware or software faults [IEEE90].

The key to the above definition lies in what is considered ‘normal’. Fault-tolerance contributes to both the availability and the reliability of a system. In the sense of availability, fault-tolerance aims at keeping the system in service when faults become apparent. In the sense of reliability, fault-tolerance aims at keeping the system safe, with or without continuation of the regular service. It is the reliability aspect that matters with safety-related systems. A

fault-tolerant system or subsystem, at which faults are present, is supposed to either balance the faults or to signal their presence, such that safety can be ensured. In the former case, the system is hiding problems; in the latter case the system is alerting its outside world (the other systems or the environment). This is considered here the ‘normal’ operation and corresponds to a similar notion from [Vo98 p.42]: “Fault tolerance traditionally refers to the level of assurance that the failure of one subsystem will not cascade (propagate) causing the failure of others.”. The pinpoint lies on the unnoticed propagation of fault effects. A system that is exposed to faults can be said to behave fault-tolerant, if it successfully hinders fault effects from leaving the system without notice. The system must compensate the fault effects or must signal their presence, but no fault effect should escape silently.

Faults may enter a system from its outside via the interfaces, or may come into being within the system [Av97 p.52], as depicted in Figure 1.4.

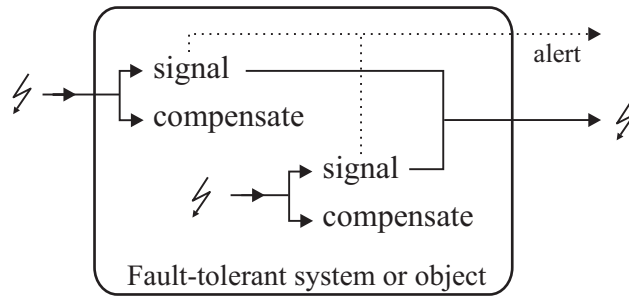


Figure 1.4: Exterior and interior origin of fault effects to be tolerated

The following definition focuses on the essence that a fault-tolerant system should neither be the emitter of faults nor be a bearer of such, unless the system is able to signal their presence. A system may thus become unavailable but still may be attributed to be fault-tolerant.

Definition 1.7: *Fault-tolerance* is the property of a system or an object, while being in the presence of faults, to prevent fault effects from leaving that system or object unnoticed.

The safety and the fault-tolerance of a system (or object) are closely related to each other when referring to the same set of faults F_0 . At system level the overall safety of a system is proportional to the overall fault-tolerance of the system, $S_{sys} \sim FT_{sys}$. Fault-tolerance can be viewed as a shield that intends to hinder faults from developing and propagating towards a location at

which they may cause harm (usually the environment of the considered object). Whether a non-tolerated fault actually causes harm or not depends on the current conditions at that location and is a matter of luck.

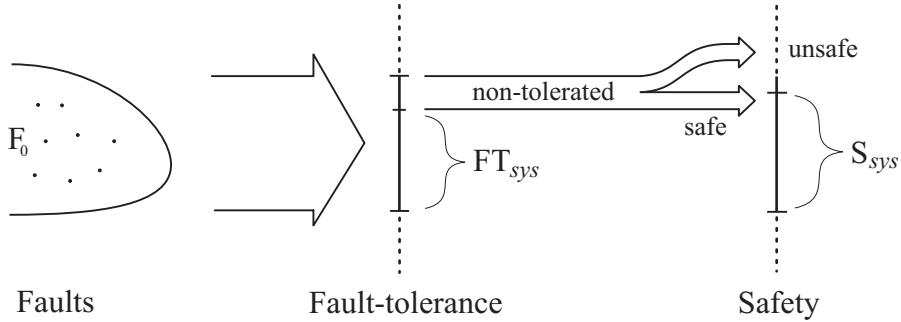


Figure 1.5: Fault-tolerance and safety

Fault-tolerance is not concerned with luck; it is concerned with preventing human life or the environment from being dependent on luck. If the set F_0 includes all faults that could ever affect the safety of a system, then FT_{sys} determines the lower bound of the system safety.

$$FT_{sys}(F_0) \leq S_{sys}(F_0) \quad (1.1)$$

The safety of a system then is at least as great as the fault-tolerance of the system. Fault-tolerance is a matter of hardware and software. As mentioned, much of the responsibility is shifted to the software. Its fault-tolerance during operation is vital and therefore has to be evaluated and put into meaningful and comparable measures. Meaningful measures require the evaluation procedure to be methodically comprehensive as well as traceable in practice (credibility through reproducibility). Comparable measures require the evaluation(s) to follow some common procedure. Fault-tolerance evaluation is one subject within the scope of dependability evaluation.

Chapter 2

Fault-Injection for Software Dependability Evaluation

2.1 Introduction

Dependability is defined as “The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers” [IFIP88]. For software there is no dependability definition universally accepted and employed. Dependability is often regarded as a set of properties such as reliability, availability, safety, fault-tolerance, robustness, and security [Co03]. It is the property ‘fault-tolerance’ of safety-critical embedded software, respectively its evaluation through fault-injection, that is considered in this work.

Fault-injection is a method for the deliberate insertion of faults into a target to determine its response. Fault-injection has become a valuable asset in dependability evaluation. Two distinct fields in fault-injection can be distinguished: *software* fault-injection and *hardware* fault-injection. With the former type, faults are implanted into a program. With the latter type, faults are introduced into hardware. Software fault-injection is unsuited to simulate the effects of random faults onto the machine instructions since this would involve massive manipulations of the original program. The executed program under test would not be the program used in the final product (there are other reasons as well). Software fault-injection is therefore not further considered.

This chapter first gives a survey on the hardware fault-injection techniques. The approaches taken in software dependability evaluation using hardware fault-injection are reviewed then. After noting the state of the art, compa-

rability aspects are addressed. Based on the *FARM* sets from [Ar90], the requirements for enabling more comparability in evaluating the fault-tolerance of software through fault-injection are determined.

2.2 Injection-Techniques

Three different injection-techniques are used in hardware fault-injection. The difference among the techniques lies in the physical manifestation of the investigated system. In *physical* fault-injection the faults are caused in the real hardware through physical disturbances. In *software implemented* fault-injection, emulated faults are injected into the real microprocessor via its regular programming interface. *Simulation based* fault-injection uses a simulation model of the target hardware; the injected faults are simulated hardware faults.

Attributes

Fault-injection is carried out through performing so-called fault-injection experiments. These are characterized by the following attributes.

Injected Faults: The outward form of the faults actually injected. The faults may be true physical faults, emulated errors at some higher abstraction level, or simulated faults.

Fault Duration: Hardware-faults may be transient, intermittent or permanent.

Operating Speed: The speed at which the investigated system or object is operating. A system may be operated in real time or may be operated (or simulated) at lower execution speed.

Real-time operation doubtlessly is the most desirable, because the system can remain connected to its real environment. The system receives real-world input and can respond through the real actuators.

Tool Intrusion: The tools necessary to inject the faults and to monitor the effects may have to be inserted directly into the investigated system, thereby changing the system away from its original state.

The inserted tools should not cause interferences. “In real-time systems where time is the most precious resource, fault injection and data collection must be performed with minimum overhead to the target system.

Otherwise, the correctness of the validation itself becomes questionable” [Ha95 p.204].

Controllability: The ability to fine-tune the injection of faults, that is, to select the fault location, and to determine the moment of injection and the fault duration, is summed up as controllability.

Controllability enables selective fault-injection, which allows to focus on a sub set of the considered faults or on a certain part of the system.

Observability: The ability to monitor the effects of the injected faults and their propagation (geographic propagation, propagation towards higher abstraction levels) is noted as observability.

For the final evaluation of the experiment results it is necessary to have knowledge about the effectiveness of the injected faults. It needs to be known whether the injected faults were effective, whether they caused errors (where, how much) and whether or not the errors have been tolerated (and by which mechanisms).

Reproducibility: Controllability and observability contribute to reproducibility which is the ability to repeat fault-injection experiments under identical conditions as in previous experiments (same fault input, same system state).

Reproducibility allows to evaluate modifications or improvements applied to the system. A convincing comparison of different versions of a system is only possible if the systems are operated under the same conditions. Reproducibility generally contributes to the trustworthiness of the obtained results since the experiments can be repeated on demand. This is especially important when it comes to the process of gaining certification. A short essay on “How reproducible should fault experiments be?” can be found in [St98].

2.2.1 Physical Fault-Injection

Physical fault-injection takes place at the physical level of the target hardware. The faults are caused through short-term variations of the physical characteristics of the hardware. The hardware is tried not to be damaged permanently, that is, physical fault-injection intends to be non-destructive.

2.2.1.1 Permanent Overranging

Permanent overranging (or overdriving) can be considered the oldest approach in non-destructive physical fault-injection. The microprocessor is purposely operated in a parameter space outside its physical tolerance limits. This approach is traditionally used to accelerate the occurrence of dynamic faults that are caused by fabrication defects. In [Hg82] and [Hg89] permanent overranging was used on various fault-free 8085 microprocessors in order to demonstrate a presented self-test approach. The parameter space has been defined by the power supply voltage, the clock frequency and the temperature.

2.2.1.2 Heavy Ion Radiation

With heavy ion radiation (HIR), a processor chip is bombed with heavy ions from a californium-252 source. The ions penetrate the depletion region of a reversed-biased pn-junction, thereby only affecting stored information by changing bit values from 0 to 1 or vice versa. The majority of heavy ions is noticed to affect only single bits [Mi95 p.442]. The fault-injection experiments must be performed in a vacuum chamber with the lid of the target IC removed, because ions are attenuated by air. The radiation flux is distributed uniformly over the chip, and error rates can be adjusted by changing the distance from the ion source. HIR allows to introduce faults at internal locations in integrated circuits. However, there is no direct control over where and when the injections actually occur [Cl95 p.49]. HIR was extensively used at the Chalmers University in Gothenburg (Sweden). Most of the publications on HIR and its application in fault-injection experiments come from there. In [Jo94], for example, HIR was used to investigate the first error-manifestation of particles within a NMOS Motorola 68000 and a CMOS Philips 68070. The application of HIR in validating fault-handling mechanisms is reported of in [Ka94].

2.2.1.3 Electromagnetic Interference

This approach uses electromagnetic interference (EMI) generated by a burst generator. Plates are connected to the generator and placed above the target circuit of the tested system. The bursts are similar to those that arise when switching inductive loads with relays or mechanical circuit breakers. Although antennas may give some controllability of the target location of fault-injection, it is difficult to determine exactly when and where a fault is actually being injected [Fo99 p.15, 30].

2.2.1.4 Power Supply Disturbances

Short, pulsed voltage drops are inserted at the power supply pin of the target processor. The voltage drops can increase propagation delays and discharge nodes. Unlike HIR, power supply disturbances affect many nodes in the target IC, thereby producing multiple transient bit-faults. The location of these faults can however not be precisely controlled. This injection technique is quite sensitive to the width and amplitude of the voltage drops. Effects can also vary widely with different circuit families [Cl95 p.49].

2.2.1.5 Pin Level Fault-Injection

With pin level fault-injection, faults are injected at the pins of the target IC. A variety of fault models is used; e.g. *stuck-at-0* or *stuck-at-1*, in which the faulted pins are set to a logic 0 or 1, *bridging*, when several pins of a circuit are interconnected, *inverted signal*, in which the level of the faulted pin is inverted, or *open connection*, when the faulted pin is essentially tri-stated. The duration of the fault can be adjusted to simulate transient, intermittent and permanent faults [Fo96 p.11]. Pin level fault-injection was used in various experiments at the CNRS (Centre National de la Recherche Scientifique) in Toulouse (France). A special tool named MESSALINE had been developed for this purpose [Ar89, Ar90].

2.2.1.6 Discussion

The most prominent advantage of physical fault-injection is that the target hardware is not functionally modified and is operated in real time. The target hardware can remain embedded in its original environment. This allows to perform fault-injection experiments while the system is connected to its external sensors and actuators. Physical fault-injection is thus especially useful for real-time systems. There is no intrusion into hardware or software. The system under test is identical to the system used in the field. Physical fault-injection has the advantage of causing actual hardware faults, but there are some disadvantages.

Physical fault-injection causes transient faults only. Permanent overranging and pin level fault-injection may be an exception to this. Detailed fault-localizing and selective fault-injection is almost impossible, therefore the experiments are usually not reproducible. Monitoring and data collection is difficult

as it requires additional hardware to be connected to the processor buses. The internal components and the interconnecting buses usually cannot be monitored at all. In general, physical fault-injection incurs high effort and costs, especially when different approaches are used in order to expose the system to as many different faults as possible, as encouraged in [Fo99 p.62].

2.2.2 Software Implemented Fault-Injection (SWIFI)

As in physical fault-injection, the SWIFI technique is applied to the real target system. The SWIFI approach consists in repeatedly interrupting the application process and executing specific fault-injection code that injects emulated fault effects into software-accessible memory locations or registers. The injection code may be embedded in the application program or may be a separate process on the processor. SWIFI was applied, for example, in DOCTOR [Ha95] for the purpose of dependability evaluation of distributed real-time systems. The actual fault-injection has been performed by so-called fault-injection agents which were separate processes on the target hosts.

SWIFI can become very powerful if the target microprocessor provides special functions, such as built-in debugging- and performance features. Either from a remote computer or from a separate software process on the target host, the processor can be stopped and fault-injection is then performed through special microprocessor commands. This happens transparent to the application software under test. The FEST technique presented in [Lo95], which uses the trace-mode feature of the 68000 processor, and the tool Xception from [Ca98], which uses the more advanced features provided by the PowerPC 601 processor, are prominent examples.

2.2.2.1 Discussion

SWIFI is a flexible and cost-effective fault-injection technique. It offers better controllability of the injection and higher observability of the effects than physical fault-injection. The SWIFI technique allows for reproducible fault-injection experiments. The experiments can be carried out near to real-time, although the actual operation speed of the target system depends on the type of faults injected. Usually, injecting transient faults with SWIFI causes the least overhead. Because the injection mostly is performed by additional software on the target system, either as a separate process or as a piece of code added to the original program, SWIFI causes some intrusion compared to the original

constellation. Monitoring-hardware or monitoring-software, if any, may also pose an alien element to the original system.

A disadvantage of SWIFI is the fact, that faults cannot be injected inside the circuits. Instead, representative errors have to be injected at the register level. SWIFI thus demands a suitable fault-and-error model of the particular target processor. Another drawback of SWIFI is the monitoring expense. Monitoring, if performed by software and not by additional hardware, happens to the cost of the execution time. High observation details may require the execution of the target application to be carried out step by step.

2.2.3 Simulation-Based Fault-Injection

With the simulation-based approach, faults are injected into a simulation model of the target hardware. This approach can support various abstraction levels of the hardware. Often the simulation model of the hardware is not limited to one abstraction level but incorporates several abstraction levels (multi level, mixed mode).

2.2.3.1 Device Level

At this level, fault-injection is used to study the impact of electrical anomalies. The fault model mainly is electrical. Fault-injection is usually performed through mathematical modifications of voltage and current time curves, thereby simulating the response of the electronic devices to electrical charges intruding the system. Simulation-based fault-injection at device level was used for example in [Du88, Ch91] to study the susceptibility of a microprocessor-based jet engine controller to voltage and current transients. The microprocessor investigated was the HS1602, and the simulations were carried out using the mixed-mode simulator SPLICE1. Device level fault-injection not only requires a precise model of the target hardware, but is also very time-consuming. Simulating a modern microcontroller at device level is infeasible.

2.2.3.2 Gate Level

Gate level fault-injection enables the simulation of larger circuits, which would be too time consuming at the device level. Also the fault model is simpler than with device level simulation. Commonly, stuck-faults and bit-flips are used. Gate level fault-injection, although never termed this way, has also been used

in classical fault-simulation on combinational or sequential circuits. The goal in fault-simulation was to find test-patterns and to verify the fault-coverage of given test-vector sets. Within the scope of dependability evaluation, gate level fault-injection is also used to observe the fault propagation towards higher levels. In [Yo96], fault-injection at gate level has been used to model the fault effects at the register transfer level of the ROMP microprocessor. In [Cz90] the effects of transient gate level faults onto program behavior (original wording) were investigated through gate level fault-injection. Gate level fault-injection nevertheless is time-consuming. This certainly is true for simulating a microcontroller completely.

2.2.3.3 Register Transfer Level

RTL fault-injection uses a register transfer model of the target microprocessor. The faults are state-errors being injected into buses and registers, but may also be behavioral errors in a more general sense. For example, the ASPHALT tool used in [Yo93] induced faults through applying state-mutation to the language statements of the RTL model description. This caused a change in the behavior of the microprocessor, which could not have been achieved as easily through just injecting bit-flips or similar into memory locations or registers. As with the device level and the gate level, an accurate model of the microprocessor is required as well as the appropriate simulation tools. RTL fault-injection is much faster than the aforementioned simulation-based injection-techniques.

2.2.3.4 Register Level

The register model usually is the best documented model of a given microcontroller. Simulation-based fault-injection at register level is the injection of faults or fault effects into memory locations, registers and public-visible state elements of a microprocessor. Register level fault-injection is similar to SWIFI, however much slower in execution but much more precise in monitoring the propagation of the injected faults. Of the four simulation-based techniques, register level fault-injection doubtless is the fastest, but it is the least accurate in modeling real fault effects. As in SWIFI, an appropriate fault-and-error model of the microprocessor is desired for the injection of realistic fault representatives.

2.2.3.5 Discussion

Simulation-based fault-injection allows a reproducible and very selective injection of faults. Depending on the modeled level of the hardware, faults can be precisely injected at internal components or nodes of large circuits. Monitoring is easy because of the all-time accessibility to the components. The fault effects can be traced in greater detail than with the other two techniques. Simulation-based fault-injection allows the injection of transient, permanent and intermittent faults. It offers high controllability and high observability. In addition, this technique is also applicable in the early phases of system development, long before a prototype is available.

The major disadvantage of simulation-based fault-injection certainly lies in its simulation-time overhead. Fault-injection experiments using the simulation-based approach are usually carried out far from real-time. Another drawback is that appropriate models of a particular hardware may not be available, at least not publicly. Especially low-level models (e.g. device level, gate level) of the microprocessor are difficult to obtain. Appropriate simulation tools are also required. The only exception may be the register model. Register model simulators are available for most microcontrollers. They are usually included in the software-development suites. The existing simulators however provide almost no fault-injection support.

2.2.4 Summary

Physical fault-injection introduces real transient hardware-faults into the system, but offers no reproducibility owing to low controllability and low observability. The system can be operated in real time, and the fault-injection tools are mostly not intruding the system. SWIFI allows the system to be executed close to real-time. The injection-tools are intruding the system to a little or medium extent. Reproducibility is provided. Controllability and observability is middling because the injection and the monitoring is limited to the software-accessible interfaces of the target microprocessor. The injected faults are data faults (state mutations) in memory locations or registers. Injecting transient faults needs the least time-overhead; injecting permanent faults however incurs more time-overhead owing to the repeatedly invocation of the injection routines. Simulation-based fault-injection is the slowest but most accurate of all. There is no intrusion. The injected faults – being simulated – can be physical (e.g. ions), logical (e.g. stuck-faults, bit-errors) or structural (e.g.

modification of RTL model). The faults which may be transient, intermittent or permanent, can be controlled and monitored precisely. Reproducibility of the experiments therefore is assured. Table 2.1 summarizes the attributes of the three injection-techniques.

	Fault-Injection Techniques		
Attributes	Physical	Software-Implemented	Simulation-Based
<i>Injected Faults</i>	True physical	Emulated errors in data (memory, registers, flags)	Physical, logical, structural. All simulated.
<i>Fault-Duration</i>	Transient	Predominantly transient, intermittent, permanent	Transient, intermittent, permanent
<i>Operating Speed</i>	Real-time	Close to real-time	Mostly below real-time
<i>Tool Intrusion</i>	None or little	Little to medium	None
<i>Controllability</i>	None or low	Medium	High
<i>Observability</i>	Low	Medium	High
<i>Reproducibility</i>	No	Yes	Yes

Table 2.1: Attributes of hardware fault-injection techniques

Concluding, for real-time execution, injection and observation, the simulation-based technique is the favorite — given that a suitable model exists. The SWIFI technique ranks second, but on principle is slower than real-time. Even overclocking the target processor cannot compensate the time needed for concurrent injection and observation.

2.3 State of the Art

Hardware fault-injection for dependability evaluation has been used in various approaches for a long time. Many approaches aimed at predominantly investigating hardware fault-tolerance mechanisms. The following survey on the related work is limited to those representative injection experiments in which software was involved or in which embedded systems were considered. A discussion on the limitations follows.

2.3.1 Related Work

Software-implemented fault-injection was used by [Ch89] to check for the response of a commercial transaction processing system on the IBM 3081-KX mainframe. The injected faults used to be corrupted memory pages that were filled with an illegal instruction code. The software failures were classified as *Crash*, *Impairment*, *non-application-related*, and *ineffective* (original wording: Nothing Happened). In [Ba90], fault-injection experiments carried out on the FIAT (Fault Injection-based Automated Testing) system are presented. FIAT consists of four interconnected IBM RT PC's. Three types of faults were injected into the memory task image of a matrix multiplication workload and a selection sort workload: zero-a-byte, set-a-byte and two-bit-compensating faults. Five distinct error behaviors of the workloads were classified: *Machine Crash*, *Task Stop*, *Response Too Late*, *Invalid Output* and *No Error*. Another automated fault-injection tool is FERRARI (Fault and ERRor Automatic Realtime Injection), reported of in [Ka92, Ka95]. The tool was first implemented on a SPARC workstation and has been ported also to IBM RISC-6000 and VAX machines. FERRARI uses traps and system calls of the operating system for the fault-injection. The fault types supported are XORing a bit, resetting a bit, setting a byte and resetting a byte. Three workloads (a matrix multiplication using checksums, a quicksort with assertions, and robust data structures in a modular robust binary tree) were used in the experiments. The response of the workload software was observed in terms of *time out*, *program exit*, *checksum detected*, and *undetected*. In [Ca98] a software fault-injection and monitoring environment called Xception is presented. Xception is built on a SPARC workstation and uses the advanced debugging features of the PowerPC 601 microprocessor. The reported fault-injection experiments aimed at evaluating the impact of faults in parallel applications (here on a Parsytec Xplorer with four nodes). The injected fault-types at register level were: stuck-at-zero, stuck-at-one, bit-flip and bridging. Three classes of impacts onto the software were categorized: *undetected*, *no error*, *error*. A technique called FEST (Fault Effect Simulation by Tracing) is presented in [Lo95]. The technique uses the trace mode facility of the Motorola 68000. FEST is implemented into the ProFi (Processor Fault injection) tool which was said to built a framework for large fault-injection experiments for the evaluation of fault-detection coverages. The paper does not present an injection experiment. Physical fault-injection was used in [Ar90]. The fault-injection tool MESSALINE is a pin level fault-injector, consisting of a management software

hosted on a MAC II and the experiment setup hosted by an INTEL 310. One of the experiments aimed at testing a self-test program which was designed on the stuck-fault hypothesis. The response of the test program was decomposed into four classes: *complete diagnosis* (error detection and report), *muteness* (no message), *garbage* (message consists of random symbols), and *no detection*. It is the more formalized approach presented in the paper that makes this publication distinct from many others. The approach will be addressed again in [Section 2.4](#). In [Re99] embedded systems were addressed in particular. Fault-injection was performed by making use of the Background Diagnostic Mode (BDM) of the Motorola MC68332 microcontroller. The faults were chosen to be transient single bit-flips which were applied to three workloads: a bubble sort, a parser for arithmetic expressions, and a Dhrystone benchmark. The behavior of the system was classified into *fail-silent* (no effect onto system behavior), *detected* (by hardware or software), *fail-silent violation* (faulty output without detection), and *time-out*.

2.3.2 Limitations

For the following limitations the reported approaches are not directly applicable for achieving comparable measures on the fault-tolerance of safety-critical embedded software, in particular regarding the fault-tolerance with respect to random faults affecting the machine instruction execution. In defense of the publications it is to remark that none of the presented approaches was claimed to allow for comparable results.

2.3.2.1 Technical Limitations

Injection Environment: In most cases the target software used to be surrounded by an operating system, and a separate injection-process was then used to perform the injection. The early approaches focused on mainframes or interconnected computers, later approaches were carried out on single multi-tasking machines (UNIX, MVS). Safety-critical embedded software usually is executed bare on the host microcontroller, and for reasons of available memory and execution time there will be no space left for additional software. Also, with safety-critical embedded software free memory space often is filled with so-called error-capturing instructions (ECI) which are redirecting the software to a specific program point in case of control-flow problems. In any case, adding non-application related software poses a distortion to the original system.

Host Processor: More recent approaches make use of the special debugging features found in some microprocessors such as the PowerPC 601 or the MC68332. Not all safety-critical embedded systems are however equipped with such sophisticated processors. Software for these systems is excluded from the presented approaches.

Execution Speed: Many of the SWIFI approaches have been stated to be fast, but the time necessary for tracing the fault effects has often been disregarded. Seeing the bundle consisting of injection *and* observation, SWIFI becomes much slower than real-time. This problem was also directly noted in [Re99].

2.3.2.2 Conceptual Limitations

Target Boundary: The boundary between the target software (the object of investigation) and the rest of the system has not been specified clearly in the publications. It is unclear where exactly the communication interface between the software and its environment was considered located. Without knowledge about the target boundary, there is no knowledge about whether a fault occurs inside or outside the target, and about how the faults (or fault effects) cross the boundary.

Fault-tolerance, as most of the dependability properties, is a feature of a certain target system or object. Therefore, when evaluating a property, a dividing line must be drawn around the target, and the location of the interface must be identified.

Fault Input: Because the behavior of the target software was stated with respect to the injected faults, but not with respect to their effects onto the software at some appropriate abstraction level, and because the fault input spectrum was varying from physical level to register level, the results are not comparable since the fault input is not comparable. The diverse fault input used and the absence of a common and target-appropriate fault set is a major reason hindering the comparability of the results.

Fault–Error Relationship: The fault–error relationship appears to have been kept quite plain in the experiments. Often the fault-injection location was considered to be also the location of the occurrence of the error. Any divergence between fault-location and error-location has been excluded right from the start (no fault propagation). Also, in most of the experiments the

injected faults were considered to be immediately effective (no fault latency), for example when the content of a manipulated cell (memory or register) became different from its original value. The problem of ineffective faults has been addressed only in a few publications. In [Ka95 p.255] a detailed example of a benign error is given. The error was benign in that it “did not cause the program to deviate from its normal execution”. The authors of [Ar90] point out that an experiment environment should provide a means to discriminate activated faults from non-activated faults.

Because software (generally any system, object or entity) cannot be tested for its response to something that does not become apparent, it is mandatory to ensure that the injected faults have indeed reached the target and have caused an error.

Golden Run: To some exceptions, as in [Ar90] and [Du88], the target software used to be artificial workload that was operating on deterministic input (e.g. matrix multiplication or sorting algorithms). The output of the fault-injected software has been compared against the output of the *golden run* through a byte-by-byte comparison. Safety-critical embedded software faces a rather non-foreseeable and complex input. Golden runs therefore are difficult to obtain since exact reproducibility of the input is required. Even in a simulated environment where states are reproducible, the comparison of the output signals likely is more difficult, because it is not the physical appearance of the output (in terms of bits and bytes) to be compared, but the meaning (the information contained).

Fault Models: Although fault-injection, as mentioned in Chapter 1, is not concerned with how close to reality the injected faults are, the injected faults seem to have been chosen quite at haphazard in the approaches. Fault models that are mapping random faults in the processing hardware onto the injected faults have not been presented or used to be quite simple (e.g. a stuck-fault on the internal bus will cause a bit-flip in a register).

2.3.3 Summary

Regarding the state of the art in using hardware fault-injection for evaluating the fault-tolerance capabilities of software, the following statement from 1995 (referring to both software fault-injection and hardware fault-injection) still holds.

Most fault-injection experiments were not designed around a formalized methodology. Experimenters typically developed customized approaches to validate each new system. This makes it difficult to apply specific results from different studies when analyzing other systems [C195 p.54].

Next to the technical limitations, the major obstacle encountered certainly is the lack of a common procedure, not only particularly for evaluating safety-critical embedded software with respect to random faults, but generally in applying fault-injection for software dependability evaluation. The various types of faults used in the experiments and the sometimes different naming of similar faults are just examples for the absence of a mutual basis. What is missing is a certain amount of comparability, that is, comparability of the procedures (conformance among the experiments) and – in the end – comparability of the obtained measures.

2.4 Comparability

From the previous it is apparent that comparability – respectively the absence of it – is an occasionally noted topic in fault-injection. This section tries to identify the major requirements necessary for enabling comparability in evaluating the fault-tolerance of software through fault-injection. The starting point has already been laid by Arlat, Crouzet and Laprie in [Ar89, Ar90] through the introduction of the so-called *FARM* sets.

2.4.1 The *FARM* Sets

A formalized methodology regarding physical fault-injection for dependability validation was presented in [Ar90]. The paper addresses the fundamental theory behind fault-injection and the interpretation of the results, and presents concrete application examples for pin level fault-injection experiments. Although the publication focuses on physical fault-injection, the ideas are worth to be considered as well for evaluating the fault-tolerance of safety-critical embedded software.

According to [Ar90], fault-injection is characterized by a collection of sets: the *FARM* sets. The set F corresponds to the input domain and denotes the set of faults in general. Dependent on the abstraction level of fault-injection,

the F set may be described by stochastic processes or may contain very specific faults. In the application example presented in the publication, the F set consisted of physical pin level faults. The A set denotes a set of activations. Activations are actions that are required to “functionally exercise the target system”. A test pattern that is applied to the inputs of a circuit is an example of an activation. Another example is the workload software that is executed on a microprocessor-based system to trigger the injected faults. The F set and the A set together determine the creation of errors. The responses of the system to the injected faults form a set of readouts R . The R set is the output domain in the fault-injection experiments. Upon these readouts a number of predicates (Boolean assertions) regarding the behavior of the system are defined. These are the predicates that the target is tested for. The M set refers to the measures that are finally derived from the outcomes of the experiments. At first instance the M set consists of coverages regarding the predicates and of corresponding distribution functions. At further instance the M set may contain the customary reliability metrics. The $FARM$ sets form the basic entities present in any fault-injection experiment.

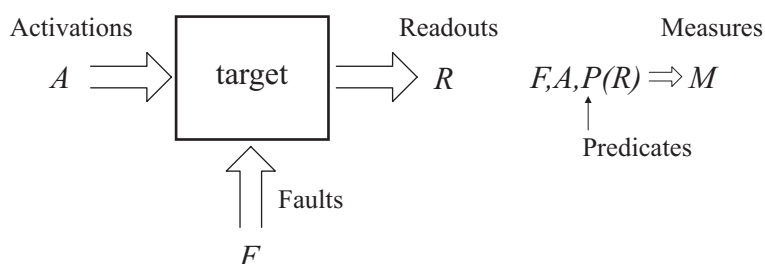


Figure 2.1: The $FARM$ sets in fault-injection

2.4.2 Prominent Example

A prominent example for conformance and comparability in fault-injection is gate level stuck-fault simulation. Everything is quite clear from the beginning. The object of investigation (the target) is a gate level model of some circuit. The subject of investigation is, for example, to assess the fault-tolerance of the circuit, or to evaluate the fault-detection coverage of a given set of test patterns. The F set may be specified to be of the class ‘single stuck-faults’, the individual faults and their locations then automatically arise from a circuit

analysis. The input patterns form the set of activations A , and the readouts refer to the node values and output patterns observed. Finally, the predicates are plain (e.g. fault masked, fault detected, fault handled) and so are the derived measures. Next to these technical aspects, there is a known terminology and a common notion of the used terms. For example, it is clear what is termed a fault (the stuck-fault) and what is termed an error (the inverse of the node's logical good value, respectively the change of the logic function of the gates). It is clear what is meant by masking, detection, latency and coverage. For these reasons the assessment and comparison of different circuits regarding their fault-tolerance capabilities is possible and feasible. Simulating large circuits or highly sequential circuits certainly poses practical problems, but the theoretical fundament for a meaningful comparison does exist: the almost standardized *FARM* sets and a standardized interpretation of the terminology.

As emerged from the state of the art, things are different when it is to evaluate and compare the fault-tolerance of software. This is especially true for safety-critical embedded software in the presence of random faults.

It is much more difficult to compare two programs in execution, each of, say, a thousand machine instructions in size, than it is to compare two gate level circuits having each a thousand gates.

The evaluation of the fault-tolerance of software in the presence of hardware faults, in particular to those faults affecting the machine instruction execution, likely never becomes as straight and easy as the evaluation of the fault-tolerance of gate-level circuits. Nevertheless, a basis for more conformance among the fault-injection experiments and – as a consequence – for more comparability of the derived measures can be laid by porting the *FARM* sets to the target ‘software in execution’ and the concerned random faults.

2.4.3 Requirement Identification

For the following discussion, fault-injection is considered as two processes: the actual injection and observation process (the experiments), and the process of inspecting and analyzing the readouts R . The sets involved and discussed next are shown in [Figure 2.2](#).

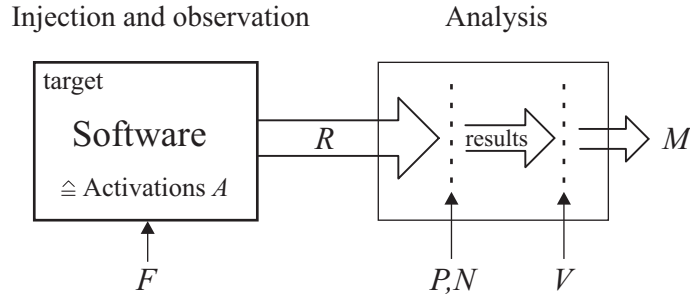


Figure 2.2: The two processes in fault-injection

2.4.3.1 The F Set

The major reason hampering comparability in fault-injection is the fact that the effects of the faults onto the target software often are not tangible (e.g. with heavy ions, pin level faults, electromagnetic interference). There is no common fault input domain and thus no common reference point. The starting point for enabling comparability therefore is some standardized and target-appropriate fault input domain.

What makes gate level stuck-fault simulation exemplary, is the fact that the injected faults not only cause sharply outlined and standardized errors, but that the abstraction level of these errors is also in close proximity to the abstraction level of the target (the circuit). The induced errors directly affect the structural elements that a gate level circuit is made of: the nodes and the gates. The errors have an immediate meaning to the object, that is, the errors and the object correlate. Such a correlation is imperative for obtaining comparable measures from an injection-experiment. Therefore the fault set F should not only be standardized, but also should be settled at an abstraction level that is close to the level of the target software. Preferably the faults in F directly affect the structural elements of the software.

The elements in F are both faults and errors: From the bottom-up view the elements are fault effects (or errors), from the top-down view the elements are faults since the input domain in fault-injection is defined to be faults. The F set should be an accumulating interface in which the effects of low-level faults can be collected (Figure 2.3). The fault effects can then be used as fault input in new experiments.

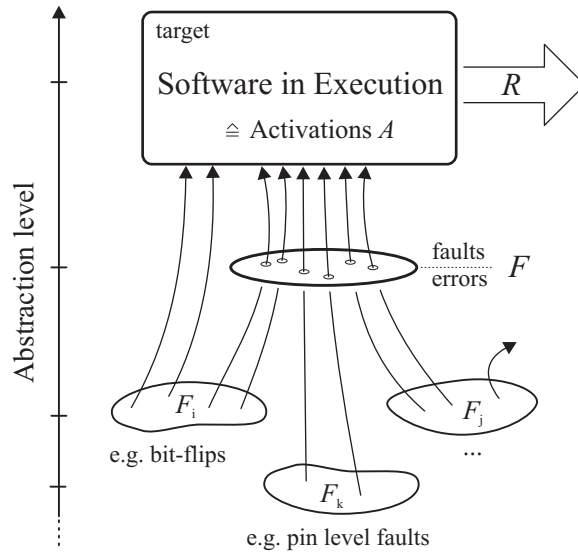


Figure 2.3: Accumulating fault effects in a common F set

2.4.3.2 The A Set

The target software in conjunction with the input from the environment forms the activations A . The sets A and F together determine the creation of errors, as well as the subsequent effects. For obvious reasons there will be no unified environment among different safety-critical embedded systems, and thus the input streams to the software are likely not to be standardized. The only comparability requirement that can be attained is to have the software operating in their real-life environment (or in a close-to-real simulated environment).

2.4.3.3 The RPM Sets

The physical appearance of the readouts R from the software usually is specific to the host system. Although any software operates on memory locations, and thus the readouts can always be expressed in terms of bits and logical signals, the great variety of possible readouts on different systems will not allow any attempt in unifying. As one is interested in the information that is carried by the readouts rather than in the particular readouts themselves, the focus in comparability rather lies on the predicates P . The readouts can remain

customized, the predicates strip and canalize the required information. Since all safety-critical software share at least some standard requirements regarding their fault-tolerance, there is a shared set of predicates P and measures M against which any safety-critical software can be assessed.

2.4.3.4 Valuation Rules V

The results obtained from the individual experiments must be inspected and classified before entering the derivation of the final measures. For example, redundant experiments or experiments in which the injected faults were not activated may have to be discarded. Citing from [SI98], “. . . even when results have been gathered, researches are still uncertain or divided as to exactly what the results mean, and how they should be used”. Therefore a common set of rules for valuating the results needs to be specified.

2.4.3.5 Notions N

Another reason hampering comparability seems to be the sometimes varying notion of common terms, such as the term ‘masking’ for instance. Even though there would be a standardized collection of *FARM* sets, a differing notion of the used terms unintentionally makes a comparison to be unfair. The term *fairness* here primarily relates to the fairness towards the investigated target, but indirectly also touches on the fairness between the software development teams. Fairness is a prerequisite for comparability. A common notion N is a prerequisite for fairness.

2.4.3.6 Target Definition

Finally, the object of investigation (the target software) needs to be defined in order to give it a clear shape. The boundary between the software and the hosting system, and in particular the location of the communication interface must be identified. The observation of fault propagation or the measuring of latencies – for instance – makes little sense if it is unclear when, where, and how the faults are entering or leaving the considered target. Also the fault-tolerance capabilities of a target may very much depend on whether a fault enters from the outside or occurs inside the target.

2.4.4 Summary Requirements

Summarizing, for enabling comparability in evaluating the fault-tolerance of software through fault-injection, the major requirements identified are:

1. A common and object-appropriate fault set F ,
 - located in close proximity to the abstraction level of the target software,
 - serving as accumulating interface in existing fault-injection experiments,
 - serving as artificial fault input in new experiments,
 - and serving as reference point among the experiments.
2. Natural activations A through operation of the software in its real environment.
3. Common predicates P , valuation rules V and measures M ,
 - serving as mutual basis regarding the effects of the injected faults,
 - and allowing for conformity in the analysis of the readouts R .
4. A common notion N of the used terminology, assuring fairness in the evaluation procedure.
5. A demarcation of the target software from the remaining system, disclosing the location of the communication interface.

Most of these requirements certainly apply for evaluating the dependability properties of any software through fault-injection, not just safety-critical embedded software and not just with respect to the property ‘fault-tolerance’. However, the requirements regarding the F set are particularly dedicated to those hardware faults that affect the execution of the machine instructions.

These requirements form the basis of the fault-injection method. In Chapter 6, the activations A , the predicates P , and the valuation rules V , as well as selected notions N and measures M are taken up again and put in concrete terms. The pivotal point in achieving comparability however is the fault set F .

In order to have the set located in close proximity to the abstraction level of the target, preferably directly affecting the structural elements, the nature of the target is to be identified first. What is the abstraction level of software and what are its structural elements? For that, the target ‘software in execution’ is to be specified more precisely. This is one of the subjects of the next chapter, Chapter 3.

2.5 Summary

This chapter was dedicated to hardware fault-injection for software dependability evaluation. The three injection techniques were discussed for their general suitability in investigating safety-critical embedded software. Physical fault-injection offers no selective injection and no reproducibility. Software-implemented fault-injection is a choice, but requires the microcontrollers to be equipped with special debugging features. Simulation-based fault-injection was found the most suitable. It is manifold in the choice of faults, and there is no intrusion into the target. However, an appropriate model of the hardware is required.

The approaches taken in various fault-injection experiments were then reviewed, and the technical and conceptual limitations were noted. The approaches are not directly applicable for evaluating the fault-tolerance capabilities of safety-critical embedded software with respect to random faults affecting the machine instruction execution. Also appearing from the publications is the lack of a mutual basis among the fault-injection experiments. Most experiments used to be customized solutions and the results are not comparable.

Therefore, in the last section of this chapter, comparability aspects were considered. Based on the *FARM* sets from [Ar90], the major requirements for enabling comparability of the obtained measures were identified. Prior point is a standardized set of faults that serves as comparable input domain among the experiments. This fault set F should be located close to the abstraction level of the software. Preferably the faults directly affect the components that the target consists of. For that, the nature of the target is to be specified more clearly.

Chapter 3

The Controlling Process

3.1 Introduction

The controlling process which is the embedded software in action, is the principal entity of a safety-critical embedded system. It is the intellectual part and reflects the capacity that the software developers have put into the program. Because the controlling process is the organizing and supervising entity of the embedded system, it significantly determines the behavior of the system and the connected actuators. The process is responsible for maintaining safety through performing fault-tolerant reactions upon faults. Its fault-tolerance is decisive for safety.

3.2 Naming

The object of investigation is termed the *controlling process* rather than *software*. One reason for this is to pinpoint on the nature of the target. The major reason however is to avoid terminological confusion. For example, a *process fault* is different from a *software fault*. The former depicts some problem in a course of activities while the latter term is reserved to depict a systematic fault in the program. Also the fault-tolerance of the controlling process is not necessarily the same as what commonly seems to be meant by *software fault-tolerance*.

Software is deemed as fault-tolerant if and only if [Vo98 p.160] [Cig04]:

1. The program is able to compute an acceptable result even if the program itself suffers from incorrect logic.

2. The program, whether correct or incorrect, is able to compute an acceptable result even if the program itself receives corrupted incoming data during execution.

The focus in the definition lies on the *program* and usually refers to the measures taken in terms of procedures and functions, language statements, or the binary machine instructions in the end. Software fault-tolerance then denotes the explicitly programmed fault-tolerance mechanisms in a program which are ought to make up the fault-tolerance of the resulting process. However, the execution of a program may be defective.

| A *good* program may be executed *badly*, resulting in a bad process.

The process may, at least for a certain period of time, be no more synchronous to the program. Control flow errors are one example. The program being executed may be the *shadow program* which is the program ‘behind’ the regular sequence of machine instructions when the program counter is misaligned (discussed in Section 6.9.4.2). If the shadow program contains hazardous instructions it likely produces a non-acceptable result. Similar applies to data accidentally being interpreted as code. Whether the definition of software fault-tolerance also covers such irregular programs is a matter of interpretation. In any case, in order to avoid adding ambiguities to the term software fault-tolerance, the software in action is termed the controlling process. After all, it is a technical process that controls the system, and independent of what may be understood of ‘hardware fault-tolerance’ and ‘software fault-tolerance’, what finally matters during operation of a safety-critical embedded system is the fault-tolerance of the resulting process.

| With safety-critical systems, what is important in the end is the fault-tolerance of the controlling process.

3.3 Components

The two components establishing the controlling process are the binary program and the processing hardware.

3.3.1 Binary Program

The binary program P_M is the final result of the software development process. It is usually stored in the ROM-area within the microcontroller. Parts of it may

also be stored in external RAM or ROM. The elements of the binary program P_M are the individual machine instructions. By means of the instructions the task of the binary program is to control the system through the on-chip and off-chip peripheral devices. Each machine instruction specifies a particular action on a particular set of memory locations, such as the general purpose registers for quick access, the condition code and interrupt flags, the special function registers connected to the peripherals, and the conventional memory space. The task of a machine instruction is to become the associated memory locations served in the specified manner. The hardware that is instructed to perform this service is the *processing hardware*.

3.3.2 Processing Hardware

3.3.2.1 Definition

The microprocessor or microcontroller sometimes is considered *the* hardware device that executes the binary program. At a closer look, however, only particular hardware areas are directly involved in the execution of the machine instructions. Other hardware areas are not involved in this process. The on-chip peripherals for instance do not execute machine instructions. All hardware areas that contribute to the transformation of a machine instruction into its associated task are denoted here by the term *processing hardware*. The processing hardware depicts those hardware areas, within or outside a microcontroller, that are essential for the correct execution of a machine instruction. The central processing unit (CPU) plays a major role. Parts of it are always involved in code fetching, decoding, and initiating the requested actions. Also parts of the bus-system, internal or external, and the clock-generation belong to the processing hardware for a given machine instruction. As different machine instructions may need different hardware areas for their execution, the areas depicted by the notion of the processing hardware may vary by instruction — and thus by time.

Definition 3.1: The *processing hardware* is the hardware that is directly involved in the execution process of a machine instruction. The processing hardware depicts those hardware areas – within or outside a microcontroller – that are essential to the fulfillment of the tasks assigned to a machine instruction.

Figure 3.1 symbolizes the hardware areas participating in the execution of the exemplary machine instruction `ADD mem1, mem2` (add the content of `mem2` onto

the content of mem1). These areas, shown in dark gray, belong to the processing hardware for this instruction.

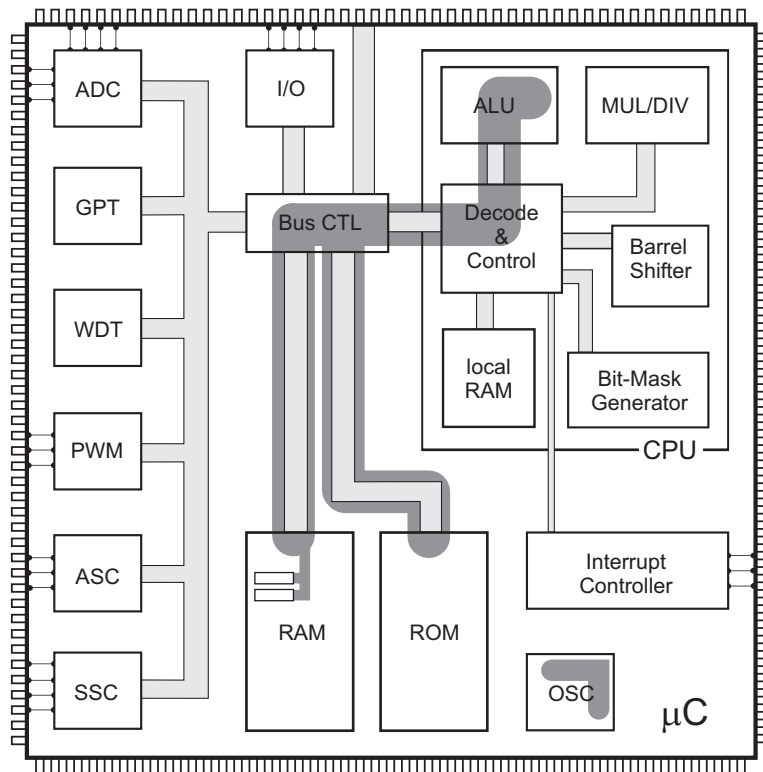


Figure 3.1: Processing hardware involved in `ADD mem1, mem2`

The difference between the processing hardware and the other hardware areas of the microcontroller (as well as of the entire embedded system) lies in the effects of random faults onto the controlling process.

3.3.2.2 Random Faults

Random faults may occur in any hardware component. Of special concern are those random faults that occur in the processing hardware. These faults are denoted by the fault set R . The set R is a subset of the faults that the controlling process has to cope with during operation (Figure 3.2).

A random fault in the processing hardware, if effective, causes an error in the execution of the current machine instruction. A fault in the other hardware does not, although the result after execution of a machine instruction can differ from the fault-free case. The following examples illustrate the difference.

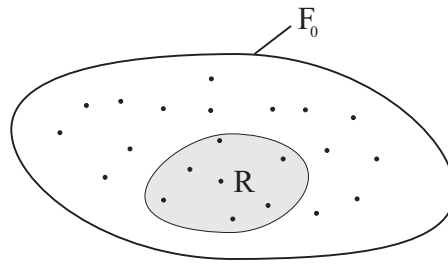


Figure 3.2: Random fault set R

Example 3.1: A fault shall be assumed to be present and effective in the adder circuit of the ALU. When the processor is executing an **ADD** instruction, the processing hardware includes the adder circuit since it is essential for the correct execution of this particular instruction. The task of the **ADD** instruction in execution is – somewhat simplified – to fetch the operands correctly, to perform the addition correctly and to put back the results correctly. All hardware involved in this execution process belongs to the processing hardware.

Example 3.2: A fault shall be present (and effective) somewhere in the I/O unit of a microcontroller. The fault causes executed **IN** instructions to receive a value that does not reflect the true data at the corresponding input pin. Here, the fault location does not belong to the processing hardware. This is because the **IN** instruction is executed correctly. The associated task is to correctly read data from the correct input port, the task certainly is not to read *correct* data. The fact that the addressed I/O-port hardware is delivering incorrect data to the corresponding port is not attributable to the executed instruction.

The effects of random faults in the processing hardware are execution errors, while the effects of random faults in other hardware areas manifest themselves in a different manner. The propagation of random faults and their fundamental effects are addressed in more detail in Chapter 4.

3.3.2.3 Fault-Tolerance

The processing hardware of a microcontroller used in safety-critical embedded systems is ought to be fault-tolerant with respect to random faults, at least to a certain degree (error-correcting circuitry as an example). With perfectly fault-tolerant processing hardware the controlling process is an exact image

of the program P_M since each machine instruction is always being executed correctly and in full accordance to the program. Any incorrect behavior of the process, such as a crash or missing fault-tolerance at moments where it was expected to be put into practice, is then owing to some kind of fault or deficiency already present in the regular program P_M . This fault may have entered the program at any development stage, from the initial idea down to the binary executable and is a clear software problem.

If the processing hardware is perfectly fault-tolerant, then the controlling process is an exact image of the program P_M .

With perfectly fault-tolerant processing hardware, software developers would be released from any concerns about execution problems and could focus on the application relevant problems.

3.4 Definition

The object of investigation is the controlling process which is, for the moment, defined as follows (a more precise definition will be given in Chapter 4).

Definition 3.2: The *controlling process* Z is the timely sequence of actions, where an action denotes the execution process of one binary machine instruction. The components establishing the controlling process are the binary program P_M and the processing hardware.

The boundary between the controlling process and its environment follows from the input and output operations. The communication interface of the controlling process is the collection of actions that perform an exchange of data across the input and output locations in the memory space. Input data is data that may carry information and that is being processed. Output data is data whose destination is outside the process boundary. The subsequent executions of the individual machine instructions are actions within the process boundary. Actions that are initiated by output data fall outside the controlling process (e.g. peripheral on-chip devices). Generally, all activities in the microcontroller or system that are not strictly contributing to the transformation of the machine instructions into their associated actions fall outside the controlling process.

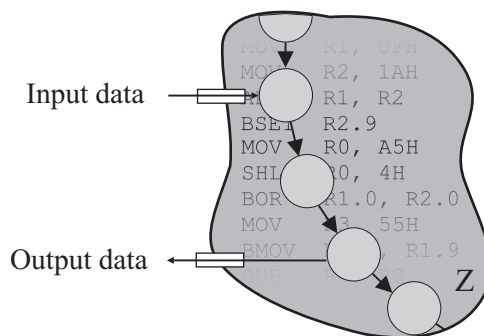


Figure 3.3: The controlling process (stylized)

When speaking about the fault-tolerance of the controlling process (or software in general), it must be closer specified what category of fault-tolerance exactly is considered.

3.5 Fault-Tolerance Categories

Corresponding to the origin of faults (or fault effects), two main categories of the fault-tolerance of the controlling process are to be distinguished. The first category is concerned with faults that enter the process from the outside. This will be referred to as the *exterior* fault-tolerance. The second category is concerned with faults that come into being within the process.

3.5.1 Exterior Fault-Tolerance

Within the scope of supervision, the controlling process must be aware of the current state of the system and of the environment. When the process notes the presence of faults in its environment, it must take care of these problems. Examples are human errors (interaction errors, thoughtless actions), technical malfunctions outside the embedded system (defect sensors or actuators, communication problems), defects within the embedded system, or faults in the microcontroller hardware (excluding the processing hardware). The effects of these faults pass the interface of the controlling process and are supposed to be recognized and tolerated then. The ability of the controlling process to be fault-tolerant with respect to faults inbound through the interface is called here the *exterior* fault-tolerance. The exterior fault-tolerance can be said to be the *concern of the controlling process about the problems of others*.

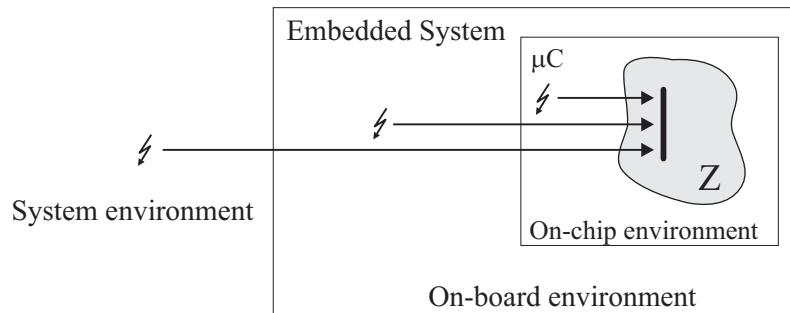


Figure 3.4: On the exterior fault-tolerance

Much of the exterior fault-tolerance originates from the hazard and risk analysis and is predominantly related to application level problems (the control tasks). A typical example from the automotive field is the practiced fault-tolerance of the controlling process when it recognizes the gas pedal and the brake pedal being pushed simultaneously. The exterior fault-tolerance is the fault-tolerance regarding the input space.

3.5.2 Interior Fault-Tolerance

In order to fulfill its application level tasks properly, the controlling process should be a course of activities in accordance with the program P_M . There are however fault events that may cause the process to deviate from the programmed activities (e.g. control-flow error). Some of these events are caused by faults that have entered the process through its interface beforehand. In this case the check-in of the faults and their further treatment within the process was just improper. This is an external fault-tolerance matter. Other fault events however may arise within the process, that is, without having passed the interface at all. Their cause is some fault in P_M or in the processing hardware. The ability of the controlling process to be fault-tolerant with respect to these internal fault effects is called here the *interior* fault-tolerance. The interior fault-tolerance can be said to be the *concern of the controlling process about its inner problems*.

The interior fault-tolerance plays a hidden but important role, and can be seen the technical equivalent of what is termed the mindfulness of a human, or what is called internal security in state politics. Corresponding to the causes of the interior fault events, two subcategories of the interior fault-tolerance can be distinguished.

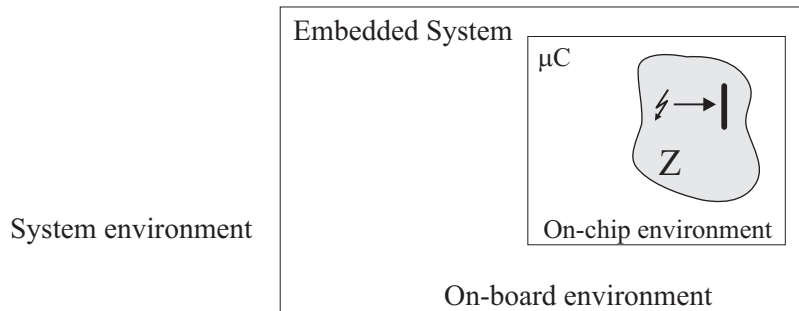


Figure 3.5: On the interior fault-tolerance

3.5.2.1 Interior Software-Fault Fault-Tolerance

The category of interior fault-tolerance that is particularly related to software faults can be closer specified as the interior *software-fault* tolerance of the controlling process (precisely: interior *software-fault* fault-tolerance). Software faults may arise from programming errors, such as the usage of non-initialized pointers, incorrect loop boundaries or wrong array indexing.

3.5.2.2 Interior Hardware-Fault Fault-Tolerance

Random faults in the processing hardware may cause execution errors. These faults address what is closer specified here as the interior *hardware-fault* tolerance of the controlling process (precisely: the interior *random hardware-fault* fault-tolerance of the process).

Definition 3.3: The *interior hardware-fault* fault-tolerance is the property of the controlling process to be fault-tolerant with respect to fault effects evolving from random faults in the processing hardware.

Because the application level functionality of the controlling process depends on a program-conform execution, the interior hardware-fault fault-tolerance is vital. It is this fault-tolerance category that the fault-injection method aims at. The propagation path of the random faults in the processing hardware towards the controlling process is termed the *path of impact* (Figure 3.6). Somewhere on this path must it be possible to create a comparable set of injectable faults F according to the requirements identified in Chapter 2.

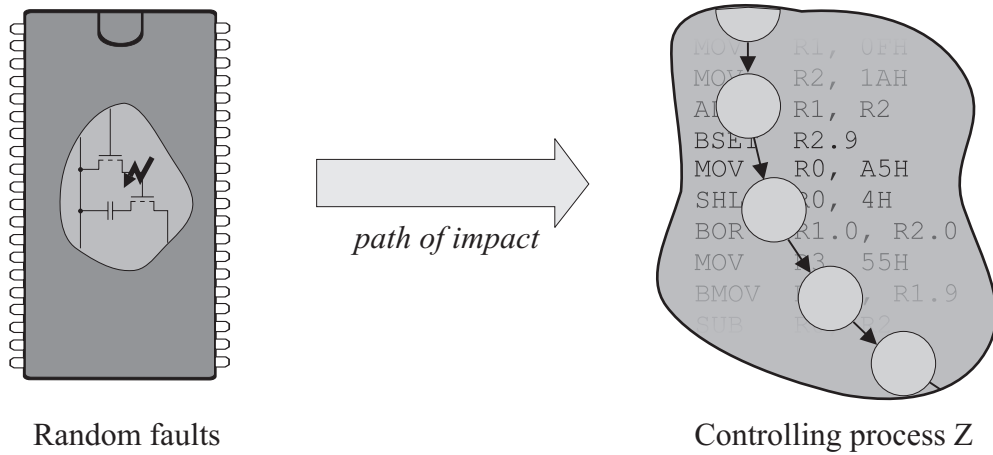


Figure 3.6: Path of impact

3.5.3 Distinction

Different from the exterior fault-tolerance and the interior software-fault tolerance, the interior hardware-fault fault-tolerance has to deal with fault effects originating from the actual technology of the processing hardware. What makes the interior fault-tolerance distinct from the other categories, is that it has to cope with sudden *asynchronous* fault effects — to be outlined in the following.

The exterior fault-tolerance relates to the proper treatment of fault effects entering the controlling process through its interface. These fault effects are not sudden in the sense of ‘surprising’ since they appear at well-known interface locations in the program (respectively at the corresponding actions of the process) and since erroneous input is just to be expected when designing safety-critical software. The space of erroneous input to be considered may be vast, but in principle, because all faults show up in form of data, this can be dealt with. After all, the faults enter the process in synchronism to the program. Regarding the interior software-fault fault-tolerance almost similar applies. Software faults are systematic and thus are not sudden, although their activation may happen to a surprise. Again, the fault events resulting from software faults arise within the controlling process at specific (but obviously unknown) locations, that is, in synchronism to the program P_M . Both the exterior fault-tolerance and the interior software-fault fault-tolerance are therefore concerned with faults or fault effects whose location of occurrence in

the process is already defined in the program. These faults (respectively their effects) enter the process synchronous to the program.

The interior hardware-fault fault-tolerance, in contrast, has to cope with faults that may affect the controlling process at any time and, figuratively, at any code location of the program. The affection is asynchronous to the program (at a random code location, at a random point in time), and therefore is considered sudden. [Table 3.1](#) lines up the fault-tolerance categories.

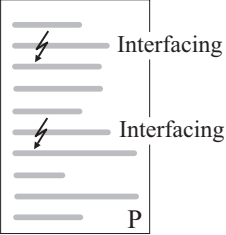
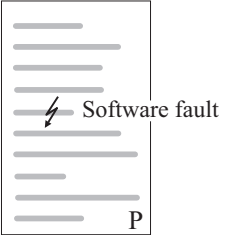
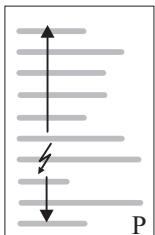
	Fault-Tolerance Category		
	Exterior	Interior Software	Interior Hardware
Concerned with:	Fault effects inbound- ing through the interface, caused by other systems or subsystems (hardware, software, user).	Fault events arising inside the process, caused by software faults in the program.	Fault events arising inside the process, caused by random faults in the processing hardware.
Fault type:	systematic, random	systematic	random
Location of oc- currence of fault effect in the pro- cess:	Defined by the program and well known. 	Defined by the program, but unknown. 	Anywhere, not defined by the program. 

Table 3.1: Fault-tolerance categories of the controlling process

Both for equipping the controlling process with interior hardware-fault fault-tolerance during the software development phase and for evaluating the interior hardware-fault fault-tolerance capabilities of the controlling process through fault-injection, it needs an error behavior model of the microcontroller (more precisely: of the processing hardware). The other two fault-tolerance categories are not concerned with such a model.

3.5.4 Online Error Detection

Online error detection seeks to detect faults during service so that their effects can be minimized. There are several fault-tolerance mechanisms that can be used to improve the interior (hardware-fault) fault-tolerance of the controlling process. The mechanisms can be purely hardware-based, they can be based on the cooperation of hardware and software, and can be purely software-based. Hardware-based mechanisms are for example self-checking circuits [Wa78], error detecting codes, and register constraint checking (e.g. invalid instruction or improper address). Signature monitoring and watchdogs are examples of mechanisms that need both hardware and software for detecting an error. Software-based mechanisms are Block-Entry-Exit Checking (BEEC), Error Capturing Instructions (ECI) and plausibility checks. The former two mechanisms have been evaluated in [Mi95] through physical fault-injection.

Some of the hardware-based mechanisms intend to prevent hardware faults from an unnoticed propagation into the register level or the functional level of the microprocessor. Self-checking circuits in conjunction with error-correcting codes can be an effective measure to protect software from fault effects originating in the processing hardware. Signature-monitoring and watchdogs are measures to pick up fault effects after the software has already been impacted. Similar applies to the software-based mechanisms. Plausibility-checking (reasonability-checking) is the only mechanism above the hardware level that is able to detect fault effects in data without preceding control-flow errors. Much of the exterior fault-tolerance is based on plausibility-checks.

As mentioned in Section 3.3.2.3, the embedded software would be released from any concerns about faulty processing hardware if in particular the hardware-based mechanisms would be applied massively to the hardware. There are microcontrollers that come close to the ideal case. An example is the AE11 (compatible to Intel 8051) which incorporates parity code checking throughout the controller, IDDQ, and built-in self-tests (BIST). The AE11 is designed to detect and respond to hardware faults within milliseconds [Boe98, As98]. For cost reasons, many microcontrollers are however sparsely equipped with such fault-tolerance mechanisms.

The common fault-tolerance mechanisms of most microcontrollers used in safety-critical embedded systems do not intercept the propagation of faults into the controlling process, but earliest take effect after the process has been impacted.

Especially with safety-critical real-time systems valuable time may have been passed until the interior fault-tolerance mechanisms grasp. Meanwhile the process may be undertaking hazardous activities, like sending malicious data to the outside world. Therefore, within the scope of software validation, the evaluation of interior hardware-fault fault-tolerance is an important subject.

3.6 Injection Method

After having specified the object and the subject of investigation in this chapter, the injection method can now be closer determined.

3.6.1 Process Fault-Injection

The target investigated through fault-injection is the controlling process which is the timely sequence of the individual machine instructions in execution. The subject of the investigation is the evaluation of the interior hardware-fault fault-tolerance of the controlling process by means of fault-injection. Fault-injection is the deliberate introduction of a fault into a target.

Since the considered target is a process (neither hardware nor a program), a fault is to be injected into a process. The conceptual injection method therefore is *process* fault-injection. This deduction may seem pedantic, but is hitting the point. Software fault-injection is the injection of faults into a program, as for instance done at great length in [Vo98]. Hardware fault-injection, of which an early stage was fault-simulation, is the injection of faults into hardware. In the beginning, hardware fault-injection solely focused on the target ‘hardware’. Later it was also used to evaluate the behavior of software. In the latter case, however, hardware fault-injection is more a technique (a means to an end) than a method, and also does not terminologically nominate the object of investigation. Therefore the injection method is specified here as process fault-injection. The actual injection technique may be hardware fault-injection. Inserting a fault into a process is done through causing a process fault.

3.6.2 Process Fault

A process is a series of actions. A process fault is a fault in a series of actions. Random faults in the processing hardware, if effective, cause execution errors.

Execution errors are faulty actions in the course of the controlling process, that is, random faults in the processing hardware cause process faults.

Definition 3.4: A *process fault* is a faulty action within a series of actions.

The process fault is defined with respect to the binary program P_M and relates to the specification-conform execution of the individual machine instructions. In the absence of process faults the controlling process precisely acts as programmed. However, even if the process is an exact image of P_M it may perform operations that may be classified undesired from a higher level perspective, that is, a process error may occur.

3.6.3 Process Error

By taking a series of actions as an *operation* that is required for serving a purpose, a process error can be defined according to [St96 p.12] as a deviation from the required operation.

Definition 3.5: A *process error* is a deviation in the operation of a correctly executed series of actions from the required operation.

The process error is not defined with respect to the binary program P_M but with respect to higher requirements imposed on the operation. The definition demands that the machine instructions are executed correctly, because otherwise the notions of fault and error would be dissolving into each other. Process faults certainly are not ‘required’ (same with stuck-faults) but they are the first order effects of the hardware faults at the level of the controlling process and thus are termed faults. A process error is a second order effect. It denotes the fact that the controlling process is correctly (with respect to the program execution) doing something undesired (with respect to higher requirements).

3.6.4 Process Failure

As the interest is on the interior hardware-fault fault-tolerance of the process, the *failure* of the process is defined straightforward. The requirements are clear from the definition of fault-tolerance. The process can be said to be failing if a random fault in the processing hardware manages to propagate into the output stream of the controlling process without somehow being signaled in good time.

Definition 3.6: The *failure* of the controlling process Z is the inability to be fault-tolerant to process faults evolving from random faults in the processing hardware.

3.6.5 Summary

So far, the fault-injection method for evaluating the fault-tolerance of safety-critical embedded software with respect to random faults in the processing hardware can be summarized as follows.

- The target investigated is neither hardware nor a program, but is a series of actions, named the controlling process.
- The subject of investigation is the evaluation of the interior hardware-fault fault-tolerance of the controlling process.
- Random faults in the processing hardware cause process faults.
- The conceptual injection method is process fault-injection. The input domain consists of process faults.

The starting point for a common fault set F are process faults, that is, faults in the individual actions. For both a closer specification of the ‘actions’ as well as for the ability to model the principal effects of random faults onto these actions, a model of the processing hardware from the software point of view is required. This is the subject of Chapter 4.

REMARK

For the evaluation of the exterior fault-tolerance no process fault-injection is required because it is not the controlling process to be fault-injected but the input stream (usually done through state mutation). Faulty input causes no process faults since input cannot affect the transformation of the machine instructions into their associated actions. Faulty input may cause process errors, though (e.g. control-flow error).

As mentioned in Chapter 2, software fault-injection is unsuited to evaluate the interior hardware-fault fault-tolerance. Software faults do not cause process faults since software faults do not affect the execution of the individual machine instructions. Software faults may however result in process errors.

3.7 Summary

In this chapter the software in execution was closer specified as the *controlling process*. This was done in order to avoid terminological ambiguities and to pinpoint on the nature of the target. The controlling process is the principal entity supervising the system and the environment, and its fault-tolerance is decisive. The two components establishing the controlling process were identified as the binary program P_M and the *processing hardware*. The processing hardware denotes those hardware areas, within the microcontroller or outside, that are directly involved in the execution of the individual machine instructions. Random faults in the processing hardware, if effective, do affect the execution process. These faults address the *interior hardware-fault* fault-tolerance of the controlling process, which is one of three fault-tolerance categories that were discriminated and discussed. It is the interior hardware-fault fault-tolerance of the controlling process that the fault-injection method aims at.

In the last section the fault-injection method was deduced as process fault-injection. Because the target is a process (neither hardware nor a program) and because the random faults cause process faults, that is, faulty actions within a series of actions, the injection method is *process* fault-injection. For both, a closer specification of the individual actions and of the fundamental effects of the faults on these actions, a model of the microprocessor is required. In order to allow the method to be applied to different software on different hardware, the model should be a generalized model of a microprocessor.

Chapter 4

Microprocessor Modeling

4.1 Introduction

In Chapter 2 the pivotal point towards comparability in fault-injection was identified as the fault set F . This set is to be adjusted to the nature of the investigated target which was identified in the previous chapter as controlling process. For investigating the path of impact which is the propagation path of the random faults from the processing hardware into the controlling process, a suitable model of a microprocessor is needed. The model must allow to represent the entry points of the random fault effects into the process in a unified manner.

In this chapter the traditional register model is first discussed. Then a more generalized model, named the *service-provider* model is presented. The basic concepts of its components, the *storage space* and the *services* are outlined. In the error model section, the principal effects of random faults, not only those resulting from the processing hardware, onto the model components are derived. Finally the entry points of random faults into the controlling process are identified.

4.2 Register Model

The register model, also known as programming model when combined with the instruction set, usually is the best documented model that is supplied with a microprocessor or microcontroller. The model denotes the memory elements at the programmer's disposal (registers, flags, input and output ports, RAM

and ROM areas), and describes the operations upon these memory elements in terms of the machine instructions.

Although the register model is the best *documented* model available, it often is not the *best* model of a microprocessor, particularly when related to the effects of hardware faults.

With modern microprocessors or microcontrollers the register model often hides essential state elements. These state elements are necessary for a correct modeling of the operations carried out upon the machine instructions.

Example 4.1: The microcontroller 80C167 supplies an instruction called EXTR (begin EXTENDED Register sequence). The instruction takes a single number n as argument. The intention of the instruction is, somewhat simplified, to redirect future memory accesses onto a particular memory area to another memory area for the next n machine instructions. Consequently there must be in the hardware some counter-variable holding n . As long as n has a value greater than zero, the redirection is active and every instruction in execution has to consult, to react correspondingly, and to decrement this variable. Although this counter-variable belongs to the operands of many 80C167 instructions, the manuals of the microcontroller do not designate this state element.

Since some essential state elements are not mentioned in the manuals, the transactions performed on these elements during the execution of a machine instruction are as well not specified, and thus the effects of random faults in the corresponding hardware cannot be modeled. Generally the activities in the microprocessor hardware during the execution of a program are left opaque to the programmer. It is not very clear which hardware activities contribute to the execution of the current machine instruction, which activities are associated with the execution of the next instruction (e.g. pipelining), and which activities are independent of the machine instructions at all. This knowledge however is of interest for safety-critical embedded software, especially when evaluating its fault-tolerance capabilities. [Figure 4.1](#) symbolizes the activities in a microprocessor from the common perspective.

Fault-tolerance is defined to be a property in the *presence* of faults, that is, the faults must first be present to the controlling process before any considerations start.

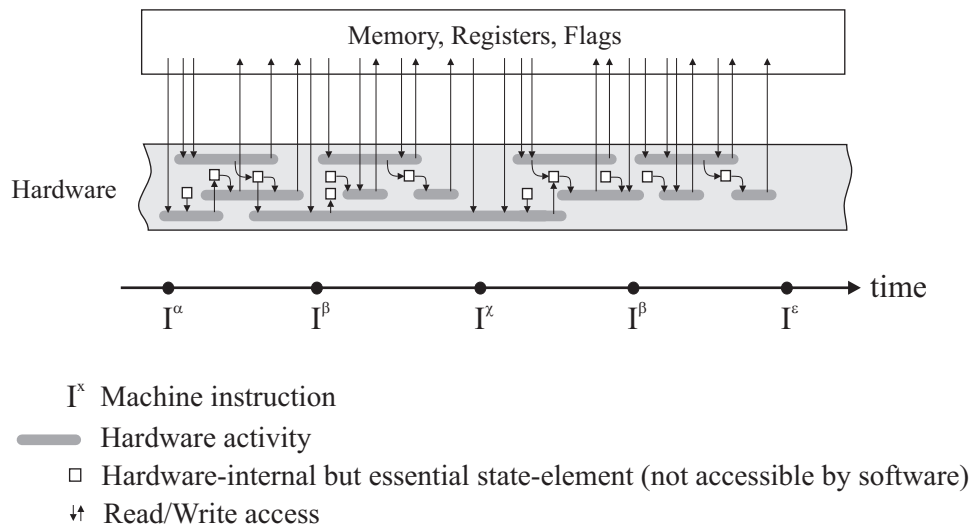


Figure 4.1: Traditional view onto microprocessor activities

Therefore it is not relevant where and when the random faults do occur in the microprocessor hardware, but through which mechanism they enter the controlling process.

Consequently, as the register model does not clearly outline the essential state elements and actions involved in the execution of a machine instruction, it cannot model the effects of random faults onto the individual execution processes, at least not entirely. There are gaps on the path from the hardware towards the controlling process since the individual actions that the controlling process is made of are not specified completely in the processor's documentation. The register model does not link hardware to software very well. The path of impact is intercepted only in parts (Figure 4.2).

4.3 The Service-Provider Model

4.3.1 Introduction

The herein introduced service-provider model is not a *new* model of a microprocessor. It is a generalized register model, however focusing on the processing hardware and emphasizing the use of a microprocessor from a strict software application perspective. The latter, the view onto a processor from out of software (the controlling process) is put into words through the introduction

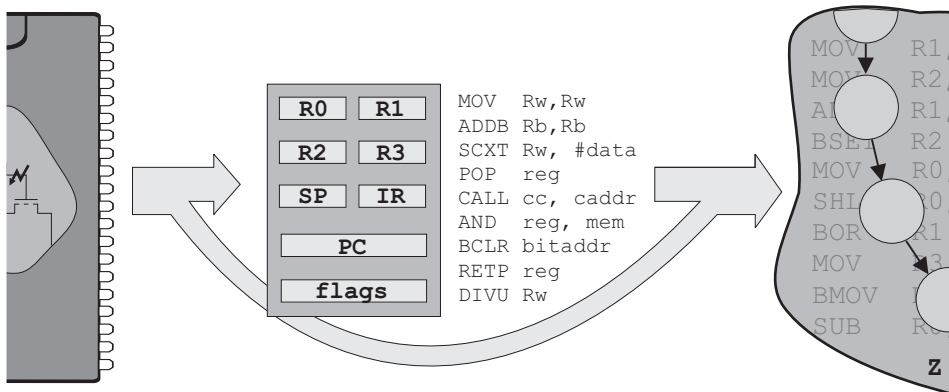


Figure 4.2: Register model partly intercepting the path of impact

of the notion of *services*. To software a microprocessor is a provider of services on request. The requests are the individual machine instructions, the services are the activities carried out on certain memory locations in response.

Before continuing, an introductory note on *instruction* and *service* is given.

INSTRUCTION AND SERVICE

The herein introduced notion of a service fills a noticeable gap in literature when it comes to referring to the *activities* of a microprocessor (respectively the processing hardware) upon a binary machine instruction. Occasionally, these activities are also termed instruction, as for example done in [Mi95 p.448] by mentioning that “. . . some instructions might be more susceptible than others to being affected by faults”. Although it becomes clear from the publication that the statement refers to instructions in execution, it is nevertheless somewhat confusing to have assigned a second but fundamentally different meaning to the notion of an instruction. An instruction is, in accordance with the original non-technical sense, just a command or order. An instruction is static and passive, while a service (the execution process of an instruction) is dynamic and active. An instruction may be coded some way (instruction code) and can be made understandable to the microprocessor (binary machine instruction). An instruction is the smallest element of the binary program P_M . Instructions have no behavior and no execution time.

A service refers to what is sometimes being paraphrased by expressions such as “instruction being executed” or “instruction in execution”, and what had to be circumscribed as ‘action’ or ‘task’ in the previous chapter. A service

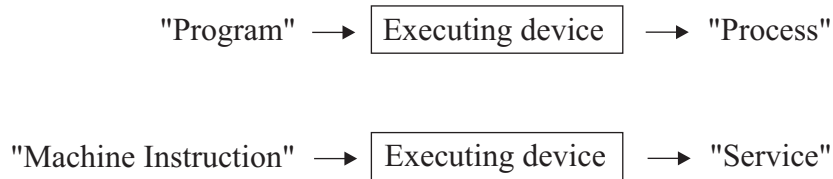


Figure 4.3: The notion of a Service

is the behavioral answer on a static instruction. A service can read and write, it can transform data, and it can fail due to hardware faults in the processing hardware. Services are the basic elements that the controlling process is made of.

CONCEPTUAL DIVISION

Within the scope of the service-provider model, the activities in the processing hardware during the execution of the machine instructions are chopped into intervals and are mapped onto separate *services*. All essential state elements are collected in the *storage space*. The storage space is the input- and output domain of the processing hardware. Figure 4.4 symbolizes the service oriented view onto a microprocessor.

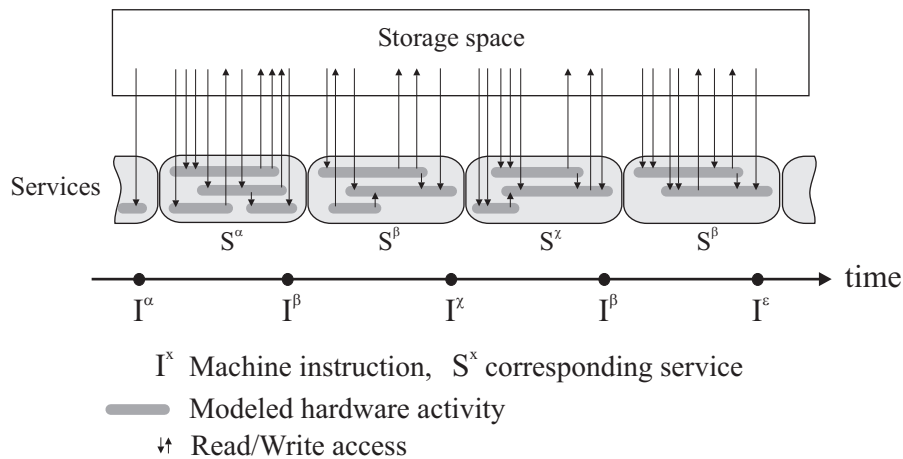


Figure 4.4: Service oriented view onto a microprocessor

The two components of the service-provider model thus are the *storage space* and the set of services. The storage space is named like this because

it encompasses more than what is usually understood as *memory*. The term service is to be understood in the sense of rendering a service. Although used here particularly for a microprocessor, the service-provider model is a model of any executing device (including a human).

4.3.2 The Storage Space

The storage space incorporates all abstractions that are needed to reflect the state of the microprocessor and of the environment from the software perspective. Most of its elements are known from the register model. These include the registers, internal and external memory, condition flags, and input- and output ports. The storage space also contains those state variables of the microprocessor hardware that are essential for the next machine instruction in execution (the next service). These variables often are not mentioned in the processor documentation. Implementation specific state elements, such as internal flip-flops and buffers, are not covered by the storage space but belong to the processing hardware. The storage space is similar to what is called the UVS (user visible space) in [Ri94], in that it contains “all state information which is passed from one instruction to the next” [p.77].

PARTITIONING

The storage space is denoted by L , its elements are called *locations*. Every location l is accessible via an address, a symbolic name, or both. Two main areas are to be distinguished within L : the working area L^W and the interface area L^F (Figure 4.5). The working area contains the program code, registers, state and condition variables, and the random access memory. The interaction between the controlling process and the environment takes place via the interface area L^F of the storage space. This area contains input- and output locations which are connected to the peripheral devices (on-chip or off-chip). The interface area represents the communication channel between the controlling process and its environment.

INTERFACE AREA

The interface area L^F of the storage can be partitioned into an input area L^I and an output area L^O . The input area reflects the status of the environment and thus acts as an information desk for the controlling process. Hardware interrupts that carry information from the environment to the software belong

to this area, as an example. They are represented as condition flags in L^I . Hardware interrupts that signal special conditions of the processing hardware, such as a divide-by-zero trap, do however belong to the working area L^W . The same applies to software interrupts. They do not import information from the environment but they represent certain internal processing states. The output area L^O passes on data or control-commands from the process Z to the environment. Both L^O and L^I are unidirectional. If a microprocessor allows input and output through a single address, that particular address is represented in the storage space as two distinct storage locations. One is located in L^I and the other in L^O . Both locations are named differently. Distinguishable naming of the locations generally applies to the storage space on the whole.

PRIVATE AND PUBLIC LOCATIONS

Because the locations $l \in L^I$ can be modified by the environment only, they are termed *public* locations. All other locations in the storage space, $l \in L^W \cup L^O$, can only be modified by the services. These are the *private* locations as the controlling process has the sole right of use.

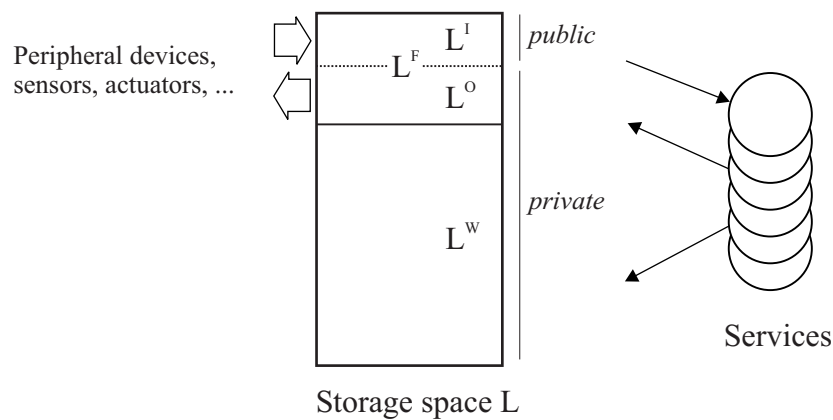


Figure 4.5: Storage space and services

4.3.3 The Services

The services depict the individual tasks that are carried out by the processing hardware on request of the software. Services are regarded in the context of this model as being atomic, subsequent, and state-less. Atomic means that a

service represents the lowest unity of inspection from the software perspective. Subsequent means that there is only one service at a time. There is no extra party present in program execution. State-less means that a service obtains all information needed for the fulfillment of its tasks from the storage space. There is no persistent information stored within the services, and information among the individual services is shared only via the storage space.

SCOPE

The task of the services is to modify the storage space in a particular manner. However, the tasks of a service are more extensive than what is usually documented in the instruction set manual of a microprocessor. Once a service is active, its task is to localize and access all relevant source and destination operands in the storage space and to perform the required transformation. In order to keep up the program flow, the final task of a service is to fetch and decode the next machine instruction and to pass on control to the successor service. Control is given from service to service.

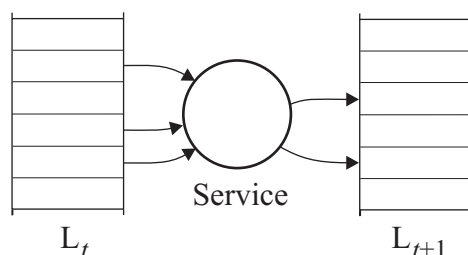


Figure 4.6: Service modifying the storage space

ON/OFF SERVICES

The very first service of a controlling process is usually called after powering-on the host processor. In order to model the power-on and power-off of the processor, the pseudo services ON and OFF are introduced. They cannot be called directly from the software and they are introduced just for completeness. The ON service activates the very first instruction of the program P_M . The controlling process ends with the OFF service which may be the successor of any service. The next service after the OFF service is the ON service.

SPECIFICATION

The description of the transformations (operations) carried out by a service actually is a description of the input-to-output function of the processing hardware involved. Although a service might be indirectly specified by means of a gate level model or a VHDL-model of the corresponding processing hardware, the focus is not on the exact structure of the hardware but on the transformation function(s). The most comprehensive and from the software point of view relevant description is a RTL description. A service preferably is specified by an enclosed set of RTL statements.

4.3.4 A Z80 Example

Figure 4.7 shows an example specification of the Z80 service `ADD A,B`. The Z80 was chosen for this example because it is not as complex as more typical microcontrollers, it is however complex enough to demonstrate how internal state elements must be named and incorporated in order to fully describe the execution of this particular Z80 instruction. The language used in the example is pseudo. It is similar to C, but uses the more comprehensive assignment operator `:=` and the single `=` for comparisons.

Given the rather simple function `A:=A+B`, the listing of the specification seems to be fairly long. The explicit task of the service – to perform the addition and to adjust the flags – is specified in a few lines. It are the implicit tasks that require most of the specification. Like with any machine instruction of the Z80, pending interrupts must be checked for. The selection of the next machine instruction depends on this, after all. Non-maskable interrupts cause the PC to be reloaded with the fixed value `0x66`, while maskable interrupts need more extensive treatment depending on the current interrupt mode of the processor. In the case of no interrupt, the PC is just advanced by 1. Once the instruction code of the next service is located, fetched, inspected for validity, and then decoded, the control is finally handed over to the next service. The listing exactly specifies (although shortened for clarity) what a software developer expects the Z80 to do when executing the instruction `ADD A,B`.

In the traditional view onto the machine instruction `ADD A,B` the operands accessed are just the registers `A` and `B`, and the flags `F` (cf. Z80 instruction-set manual). In the service-provider view however, the set of operands accessed by the service `ADD A,B` is more extensive. The set contains the explicit operands to be processed (here `A`, `B`, `F`) but also contains the implicit operands which

are those storage locations that must be evaluated or processed in order to put the hardware into the correct initial state for the next service. Table 4.1 lists the operands of the Z80 service ADD A,B. Some operands are mandatory. They are always accessed by the service. Conditional operands are those operands that may be accessed in option.

Mandatory Operands	Comment
A	Register A (Akku)
B	Register B
F	Condition Code Flags (S, Z, H, P/V, N, C)
PC	Program counter
NMI	Flag indicating the presence of a non-maskable interrupt
IFF1	Interrupt Flip-Flop I, a flag indicating whether maskable interrupts are enabled or not
Conditional Operands	
SP	Stack-Pointer
<i>Stack</i>	The storage locations referred to as 'stack'
INT	Flag indicating the presence of a maskable interrupt
INTA	Storage location in L^O , generating the physical interrupt-acknowledge signal
IFF2	Interrupt Flip-Flop II
IM	Selector, indicating the current setting of the interrupt mode (0, 1, 2)
I	Interrupt register

Table 4.1: Operands of the Z80 service ADD A,B

4.3.5 Definitions

Definition 4.1: The *storage space* L is the collection of data sources and data destinations for the individual services. It contains those information entities that are essential for the services to perform their specified tasks, and it serves as information carrier among the services. The interface area L^F is the information desk between the services and the remaining system. The storage space is a static abstraction of the microprocessor from the software perspective.

Service "ADD A,B"{	Addition: A = A + B
a := L[A]	localize operand and fetch content
b := L[B]	localize operand and fetch content
s := a + b	perform Addition
f := L[F]	get the contents of the flags F
if (s = 0) f.Z = 0	set the ZERO-bit in F
...	do so accordingly for CARRY C, SIGN S,
...	HALF-CARRY H, OVERFLOW V and SUBTRACT N.
L[A] := s	write back result
L[F] := f	write back updated flags
pc := L[PC]	get the value of the program counter
pc := pc + 1	increment pc
/* Before finishing, check for pending interrupts. */	
nmi := L[NMI]	check for a non-maskable interrupt
if (nmi = 1) {	if a NMI was signaled
L[INTA] := 1	send acknowledge
sp := L[SP]	get value of stack pointer
sp := sp - 1	decrement
L[sp] := highbyte(pc)	put high byte of PC onto stack
sp := sp - 1	decrement
L[sp] := lowbyte(pc)	put low byte of PC onto stack
L[SP] := sp	put back new value of stack pointer
L[PC] := 0x0066	set PC to fixed address 0x0066
next := decode(0x66)	find the corresponding service
jump(next)	pass on control, no return
}	end NMI treatment
iff1 := L[IFF1]	check the Interrupt-Enabled flag
if (iff1 = 1) {	if enabled
int := L[INT]	check for a maskable interrupt
if (int = 1) {	if an interrupt is pending
L[IFF1] := 0	clear the interrupt flip-flops
L[IFF2] := 0	
L[INTA] := 1	send acknowledge
im := L[IM]	get interrupt-mode (can be 0, 1 or 2)
/* Not shown for space reasons: Depending on the current interrupt mode, the service	
now has to adjust the PC to either the fixed address 0x0038 or has to fetch	
the interrupt-vector and to treat it as either an address or an instruction code.	
The pc then has to be put onto the stack and L[PC] has to be updated.*/	
...	
jump(next)	pass on control, no return
}	end interrupt treatment
}	end test for interrupt enabled
/* There were no interrupts when reaching this point. Major work is done.	
Now the PC must be updated and the regular successor service is to be	
identified and given control. */	
L[PC] := pc	update PC
ic := L[pc]	retrieve instruction code of next service
if invalid(ic) {	if code is invalid
...	take appropriate action
} else {	
next := decode(ic)	decode (extract service identification)
jump(next)	pass on control, no return
}	
}	end service specification

Figure 4.7: Sample specification of the Z80 service ADD A,B

Definition 4.2: A *service* is a well-defined and self-contained transformation-process of the storage space L , such that the service correctly reflects the execution of a particular machine instruction. The collection of services is the dynamic abstraction of the functioning of the processing hardware from the software point of view.

Definition 4.3: The *service-provider model* is an essential functional model of the processing hardware from a software application perspective. The model consists of a storage space and a set of services.

Although the service-provider model is used here particularly to model a micro-processor, it is a model of any executing device (including a human processor) that is performing a task through following a given guideline. The performance is split into elementary steps, and each step is a self-contained operation on certain input- and output locations (e.g. shelf, oven, refrigerator – with a cook being the processing hardware).

By means of the services the controlling process can now be defined more precisely than in the previous chapter.

Definition 4.4: The controlling process is the timely sequence of services evolving from the binary program P_M in the storage space. The process senses and controls the environment through the interface area L^F .

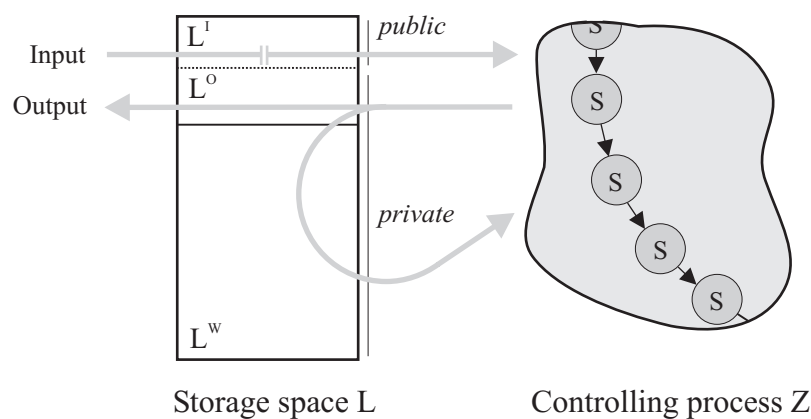


Figure 4.8: Controlling process operating on the storage space

4.3.6 Gate Level Model and Service-Provider Model

The gate level model and the service-provider model follow a similar methodology. A logic gate is a behavioral abstraction of the functioning of the device level components. The gate emphasizes the Boolean-logic point of view. In the service-provider model, a service is the behavioral abstraction of the functioning of the processing hardware. Services emphasize the application point of view onto a processor. Both gates and services are active elements. The nodes in the gate level model correspond to the storage locations in the service-provider model. Both pass information and both are passive elements.



Figure 4.9: Structural elements at gate level and service level

Gates and services are hard-coded. Both work on their input and produce output. The transformation function of a gate is simple while the transformation function of a service is rather complex. A gate has a single output, whereas a service has multiple outputs. Inputs and outputs at gate level are hardwired, whereas the source- and destination addresses of a service are both hard-coded (e.g. PC) and soft-coded. It is worth to note that incorrect input does not influence the basic functioning of neither a gate nor a service. The input is, after all, logic. As long as the hardware is fault-free, both gates and services precisely act in accordance with their specification. The basic functioning can only be affected by hardware faults.

4.3.7 Discussion

Early microprocessors, like the Motorola 6502 or the Intel 8085, came very close to the service-provider model. The processor documentation listed all relevant state elements (with minor exceptions) and the instruction set manuals outlined the corresponding operations on these elements. The function of the microprocessor from the software application perspective could be described almost completely by the instruction set. Such a model not only was comprehensive, but also used to be the mandatory starting point for several investigations (e.g. the functional self-test approaches of [Ab79, Hg82]).

Modern microcontrollers, as mentioned, often are not fully described by the provided documentation. The service-provider model therefore gives a framework for building a complete and comprehensive model. Actually, the debuggers included in the integrated software development suites for modern microcontrollers (e.g. HITECH, KEIL) are in fact service provider models (since they are fully functioning simulators of the processing hardware), however these are developed customized and show to the user only the essential state elements known from the processor documentation (the traditional register model view). It would be very helpful to both developers of embedded software as well to developers of processor simulators (or emulators) if the manufacturers of microprocessors would make available service-provider models of their processors. The models should use a common naming scheme. This would aid in a better understanding of the working of the processor in general (even if fault considerations play no role) and of the individual instructions in execution. Furthermore, since the service-provider model can easily be turned into a full-functioning simulator, the market acceptance of the microcontroller might be sped up as well.

As will be shown in Chapter 5, none of the approaches on (error-) behavior modeling of a microprocessor has considered the abstraction level of the service provider model.

4.4 The Service-Provider Error Model

By means of the essential state elements collected in the storage space and by means of the individual services acting on the storage space, the service provider model fully describes a microprocessor (the processing hardware) from the software point of view. Consequently the model must also be able to describe the fundamental effects of hardware faults from the software perspective.

4.4.1 Storage Space

The storage space L serves as an information carrier among the services and between the services and the environment. Its only function is to store information. This function may be affected by faults in the storage hardware, so that contents are altered. The storage hardware is the hardware that is used to preserve information. Static memory cells and dynamic memory cells,

as well as the refreshing mechanisms, belong to the storage hardware. Any access-circuitry does not belong to the storage hardware because it is used by the services to fetch or to deliver data, but it has no preserving function. The spurious alteration of storage locations, caused by faults in the storage hardware, is called retention errors here.

Definition 4.5: A *retention* error is the unintentional change of an information entity within the storage space, and is caused by a random fault in the storage hardware.

In the fault-free case, the contents of the storage space can only be modified by either the services (private locations) or by the peripherals (input area L^I). Retention errors model the fault case where contents change independently of any access. A retention error is a state error. It causes a state change of the storage space. Because the storage space is passive, a retention error is irrelevant as long as the corresponding storage location is not read-accessed by a service. The only exception to this are retention errors in the output area L^O , which is discussed shortly in the fault propagation section, Section 4.4.3.

4.4.2 Services

Once a service is activated, it determines its operands, performs certain operations upon the operands and finally passes control to the successor service. These basic tasks must be carried out according to the specification of the particular service. A random fault in the *processing hardware* may cause a malfunction of these tasks. The manifestation of these errors are termed service errors.

Definition 4.6: A *service error* is a deviation of the behavior of a service from its specification. It is caused by random faults in the processing hardware that become effective during the activity time of the service.

DATA-INSENSITIVITY

Service errors are the consequence of hardware problems and are not caused by incorrect input data. Services are hard-wired (includes microprogramming) and their fundamental working does not depend on the state of the storage space. To make this more clear: The rendered transformations of a service of course depend on the storage space (otherwise fetching the operands would

be useless), but the regulations of what to do with the inputs are hard-coded. A conditional branch, like `JMP NZ dest` for example, reloads the program counter to *dest* only in case the Zero-Bit is found to be cleared, but the regulations behind this service are always the same: “get the Zero-Bit, check for zero, do *this* if the test is positive or do *that* if not”. The regulations are independent of the state of the storage space, therefore errors in the storage space do not cause service errors. After all, the operands consulted by a service are just plain data. This is similar to logic gates whose function also is defined in hardware. Incorrect data does not influence the function of a logic gate. If it appears that a service acts different from what is known from the specification, then either the specification is incomplete or the input data has *activated* a hardware fault.

4.4.3 Fault Propagation

As will be shown, the service-provider model fully intercepts the path of impact. The propagation path from the hardware to the controlling process falls apart into three sections.

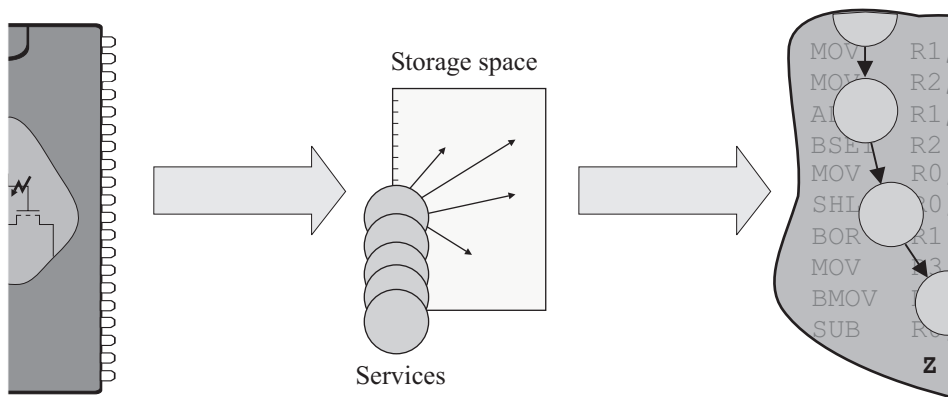


Figure 4.10: Service-provider model intercepting the path of impact

4.4.3.1 From Hardware to Service-Provider Model

Problems in the storage hardware are covered by retention errors. Problems in the processing hardware are covered by service errors. There are also hardware faults that neither cause retention errors nor service errors, but whose effects

show up as erroneous data in L^1 . Examples are defect on-chip peripherals, such as timers or A/D converters. The faults may also originate outside the microcontroller or even outside the embedded system. In any case, if these faults do not affect the processing hardware or the storage hardware, their effects either vanish or propagate into the input area of the storage space.

Figure 4.11 shows a microcontroller with six exemplary fault locations. The service currently busy shall be something like `ADD mem1,mem2` (add the content of one memory-location onto the content of the other). The faults shall be single-faults, that is, there is only one fault considered at a time. Also the faults are assumed to be effective in the end.

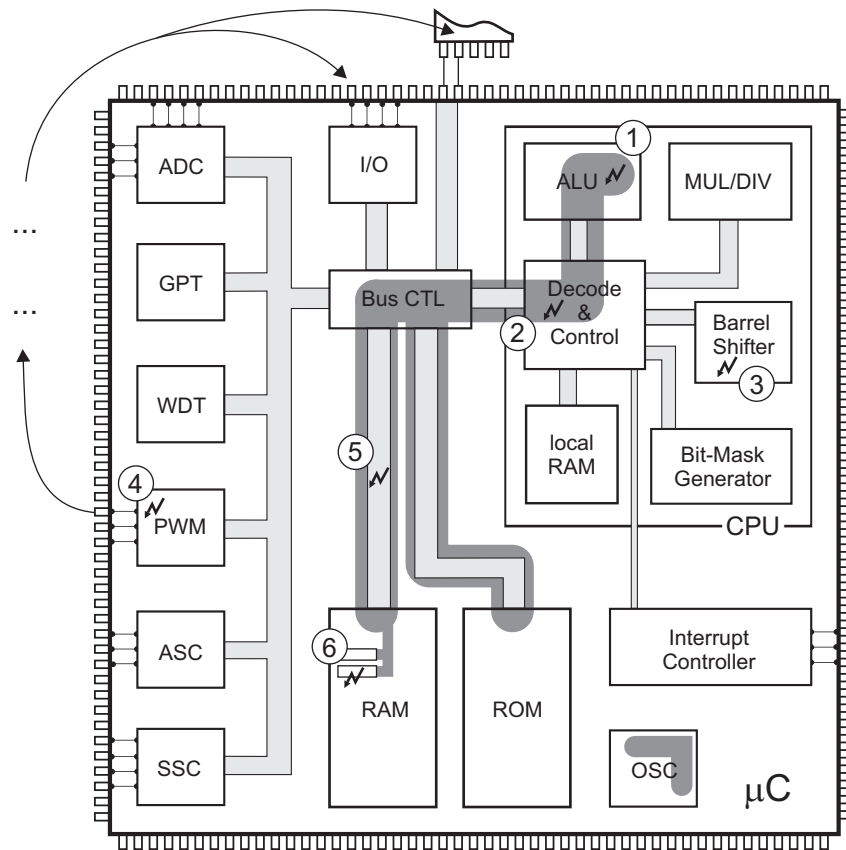


Figure 4.11: Hardware faults during service `ADD mem1,mem2`

Fault 1 is a fault in the ALU. The fault will lead to a service error. There is one exception. If the fault directly affects a memory element in the ALU that belongs to the storage space (in which case it is an essential state element),

then the fault causes a retention error. The similar applies to fault 2 in the Decoding-Unit. It causes a service error or a retention error, depending on its location in the circuit. Fault 3 in the Barrel-Shifter unit is irrelevant for the current service. Still, if the Barrel-Shifter contains a memory element that is mirrored in the storage space, and if the fault directly affects the storage hardware of that memory element, then the fault causes a retention error. Because the current service does not access that storage location (otherwise the processing hardware would have also included the path to the memory element in the barrel-shifter), this fault has no meaning to the current service. The same applies to fault 4 in the Pulse-Width-Modulator. This fault shall denote a defect in the output-stage of the unit. Assuming that the PWM currently is active, the fault causes an error in the output signal. The fault may then propagate across the circuit-board into the environment of the embedded system. Since safety-critical embedded systems are feedback-systems (closed-loop control), the fault effect is likely to return to the microprocessor at a later point in time. The effect will then either show up at the regular input-interfaces that are used to retrieve the condition of the controlled environment (e.g. ADC, I/O), in which case it will end up in the input-area L^I of the storage space as just data, or the fault effect becomes more serious by affecting the processing hardware external to the microcontroller, in which case the service busy at that time will become erroneous. Fault 5 is a fault affecting the bus. This fault may change data during transportation or it may affect the address signals. It causes a service error. Finally, fault 6 denotes a problem in the storage hardware of a memory-cell. This fault turns into a retention error.

Summarizing, the effects of hardware-faults first manifest themselves

1. as retention errors,
2. as service errors,
3. or wrapped in some input data in L^I .

This is depicted in [Figure 4.12](#). Hardware faults that neither cause retention errors nor service errors, nor show up in the input stream to the controlling process, are irrelevant from the software perspective. Software cannot do anything about the fault effects, nor can software be tested against these faults.

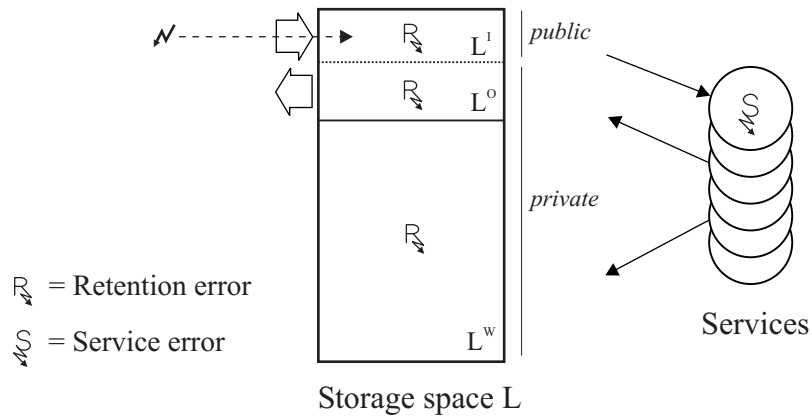


Figure 4.12: Manifestation of hardware faults

4.4.3.2 Among Storage Space and Services

Service errors usually cause the state of the storage space to be different from the fault-free case, that is, service errors produce data faults in the storage space. Service errors may be ineffective such that the storage space is left identical to the good case. The ineffectiveness of service errors will be addressed more thoroughly in Chapter 6. Service errors thus have two consequences. They are either

1. ineffective
2. or produce data faults in the storage space.

Data faults in the storage space are either the result of a retention error or they have been deposited there by a service beforehand (propagation). Since the storage space is passive, a data fault needs to be picked up by a service in order to potentially pose a problem for the controlling process. Data faults may be masked, in which case the output of the service is unaffected. Data faults may

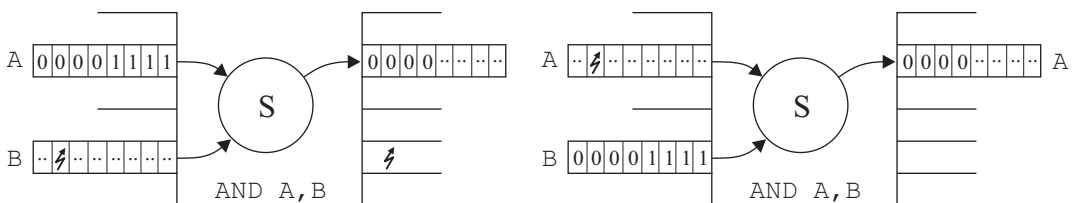


Figure 4.13: Masking and blanking a data fault

be blanked, in which case the fault is overwritten with the (or a) good value. Data faults may propagate unnoticed through a service or they may be detected by a service (e.g. invalid address, invalid instruction code). Figure 4.13 shows an example of masking and blanking.

Summing up, data faults in the storage space may be

1. dormant (no read access)
2. masked (service output unaffected)
3. blanked (fault nullified)
4. detected (constraints at functional microprocessor level)
5. or propagate through a service (output affected).

The propagation of fault effects (or errors) in the storage space and the corresponding predicates are addressed in Chapter 6.

4.4.3.3 Towards the Controlling Process

Random faults in the processing hardware manifest themselves as service errors. Since the controlling process is a timely succession of services, the random faults do affect the controlling process by means of affected services. Services therefore are the natural entry points of random fault effects propagating into the controlling process.

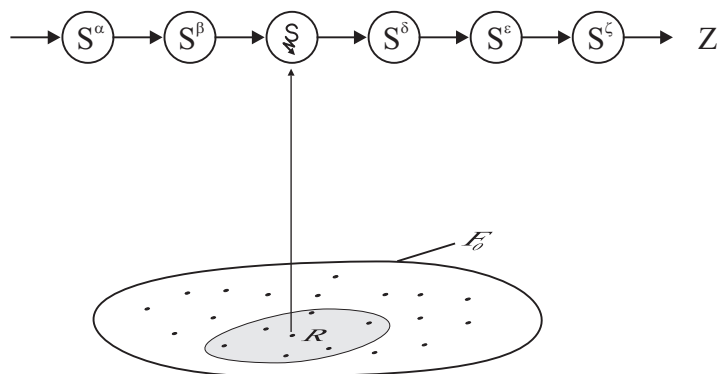


Figure 4.14: Service errors are the fault effect entry points

This not only applies for random faults in the processing hardware, but for all fault events causing data faults in the storage space (e.g. software faults or corrupted and carelessly checked input data). In any case must the faults propagate through the services, either from input to output (data fault propagation) or in terms of a service error, in order to have a potential meaning for the controlling process.

4.4.4 Discussion

Although the subject of investigation is the interior hardware-fault fault-tolerance of the controlling process (thus focusing on the processing hardware), the error model describes the effects of any random fault for the reason of completeness. The effects of random hardware faults manifest themselves in the service-provider model

- as *retention errors* (malfunctioning storage hardware),
- in form of erroneous data or error-information in the input area L^I ,
- or as *service errors* (malfunctioning processing hardware),

otherwise they are irrelevant to the controlling process. The service provider model thus fully intercepts the path of impact.

Because the fault effects always must propagate through the services in order to pose a potential problem, services are the natural inlets into the controlling process.

Services therefore are the suitable starting point for fault-injection when it is to evaluate the fault-tolerance of the controlling process. This is especially true for random faults in the processing hardware since these faults, on principle, cause ‘action-problems’, that is *process* faults. The effects of random faults in the processing hardware are represented by service errors. The interior hardware-fault fault-tolerance of the controlling process is thus the ability to be fault-tolerant to service errors. In Chapter 6 the intentional falsification of services for the purpose of fault-injection is called *mutation*.

4.5 Summary

This chapter started with a discussion on the register model and its limitations. The register model often hides essential state elements and associated

activities that are involved in the execution of the individual machine instructions. Therefore a more generalized model, named the *service-provider* model, was presented. The model consists of a *storage space* and a set of *services*. The introduced notion of a service not only fills a terminological gap, but also conceptualizes the link between hardware and software. Services are well-specified, enclosed, behavioral answers upon the static machine instructions. They emphasize the software point of view onto a processor. The services are successive, that is, there is only one service active at a time. Information among the services is exchanged only via the storage space. The storage space incorporates, next to the traditional memory and registers, the essential state elements that are needed for the services to perform operation and continuation. The hardware constituting the services is the processing hardware. The service-provider model is a unified abstraction of a microprocessor.

By means of the services, the controlling process was then more properly redefined as a series of services. The interface area L^F in the storage space is the geographical location of the communication interface between the controlling process and the remaining system. It was also pointed out that the gate level model and the service-provider model follow a similar methodology. As with logic gates, the principal functioning of services can only be affected by random faults in the processing hardware.

In the error model section, the fundamental effects of hardware faults onto the model components were outlined. Hardware faults manifest themselves as *retention errors* (malfunctioning storage hardware), as *service errors* (malfunctioning processing hardware) or affect the input stream to the controlling process. It was shown that the service-provider model fully intercepts the *path of impact* which is the fault propagation path from the hardware into the controlling process. Finally it was deduced that the services are the pivotal point in the fault-injection method. Any hardware fault, not only those from the processing hardware, must first propagate through a service in order to pose a potential problem for the controlling process. Services are the link between hardware and software. Service errors in particular are the representatives of random faults in the processing hardware. They form the representative fault set F in the fault-injection method.

In the next chapter service errors are being looked at closer.

Chapter 5

Service Errors

5.1 Introduction

In the previous chapter services were identified the conceptual link between hardware and software. Services are the structural components that the controlling process is made of. Random faults in the processing hardware must propagate along the path of impact in order enter the controlling process by means of service errors (Figure 5.1). Service errors are the representatives of the considered random faults.

This chapter intends to obtain a more concrete insight into realistic service errors. Work related to the field of fault mapping is reviewed first. After giving the state of the art, some service error classes are identified, and multiple affections are discussed. The major part of this chapter is dedicated to investigations on the error behavior of combinational circuits in the presence of faults. The circuits investigated are typical customary circuits used in microprocessors (e.g. Adder, ALU). These findings allow for the creation of realistic and representative service errors, which is exemplary shown then. The chapter ends with an appeal to the manufacturers of microprocessors.

5.2 Fault Mapping

Although fault-injection, by principle, is not concerned with why certain adverse events occur or how representative the injected faults are, most fault-injection methods stand or fall with the availability of a plausible set of injectable faults. *Fault mapping* is the process of achieving an artificial set of

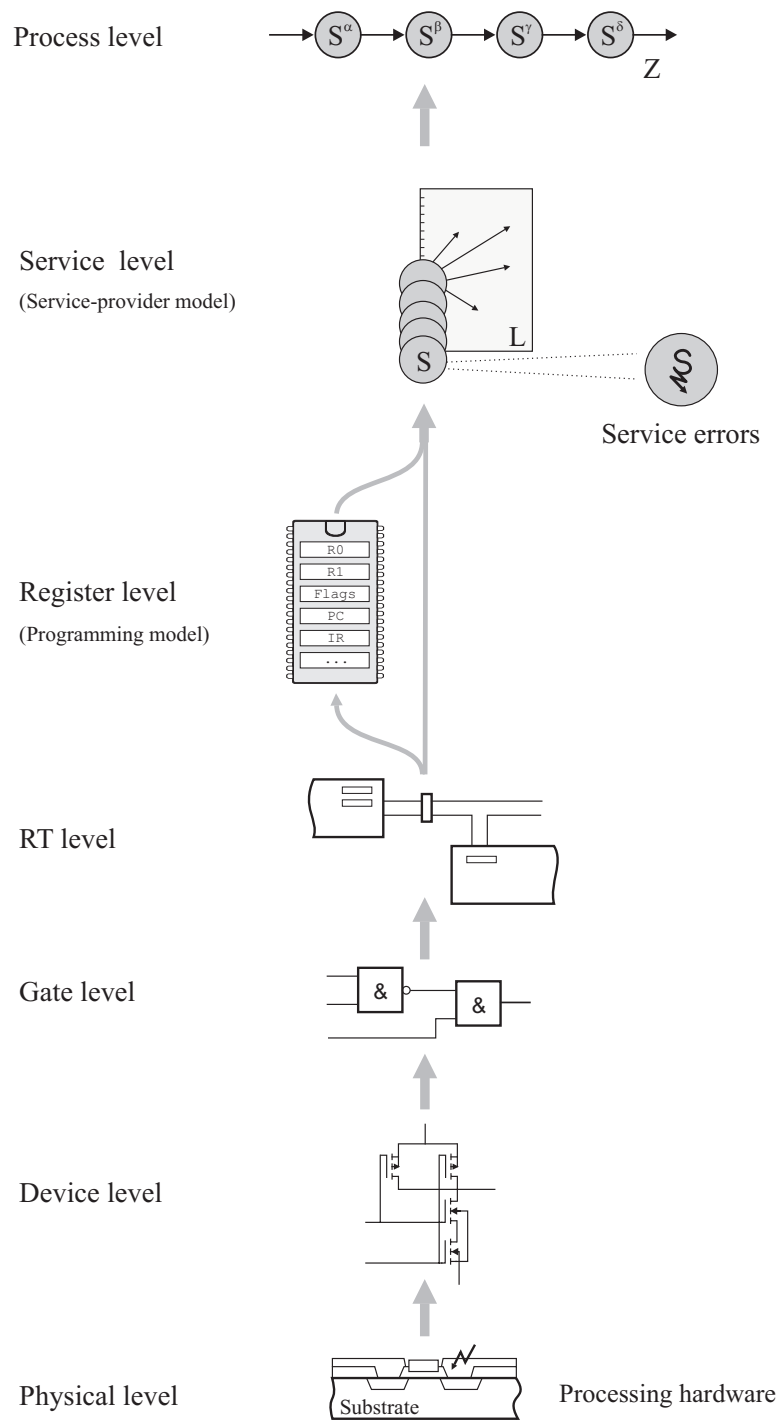


Figure 5.1: Propagation path towards controlling process

representative faults for the purpose of fault-injection. Fault mapping therefore plays an important role. There are however differences in the assessment of its significance in the end. This is because fault mapping is a difficult process. The main problem of practical fault mapping is the lack of detailed information: information about the so-called real faults, about the affected system, and especially about the propagation paths of the faults towards higher abstraction levels. Although representative fault sets are desirable, they cannot be achieved sometimes. Therefore some authors even encourage to “... avoid the trap of spending all of our time worrying about how realistic certain anomalies may be, and simply observe how those anomalies impact the software” [Vo98 p.25].

5.2.1 The ‘real’ faults

There is a consensus of opinion among the publications, that transient faults occur more frequently than permanent faults. Ratings however vary. In [Ma90 p.513], transient faults are estimated to occur 10 to 30 times more frequently than permanent faults. In [Cz90 p.238], the rate of occurrence of transient faults is assessed to be 20 to 100 times more likely than with permanent faults. As reported in [Du88 p.272, Ch92 p.1515, Ma90 p.513], experience has shown, that approximately 85% of the system errors were caused by transient faults. However, the experience that is being referred to by the above papers – as well as by other fault-injection publications – dates back to the early 80’s and predominantly originates from investigations on mainframe computers, as done for example in [Iy86]. Although some statistics on transients were collected, no particular detailed information about them (cause, duration, energy) is available. Because the chips are getting more densely packed and the timing margins are decreasing, transient faults are likely to continue to play a major role among the random faults. On the other hand, as the dense chip-structures have become more susceptible, the chance of permanent faults increases as well. A disturbance that caused a transient fault in a circuit twenty years ago might now inflict a permanent damage. Little can be said about the real-life proportion between permanents and transients for today’s hardware, not least because this also depends on the current environment of the system. In addition, the trend towards deeper sub-micron geometries tends to alter the statistical distribution of defects and also creates new classes of timing-related defects which currently have no adequate fault models [Ga01].

Real faults thus are elusive and can hardly be encircled. Because they are a “moving target” [Ca99 p.50], and the fault models at device level have to be rethought, there is a general need for a higher representative fault model for the purpose of fault-injection. A short review on related work in fault mapping is given next.

5.2.2 Related Work

There are some publications from the fault-injection community that in particular attempt to obtain a more distant error behavior portrayal of a microprocessor. The approaches differ in the analysis method and in the abstraction level of the error model. An early approach, presented in [So86], is the investigation of program-flow disturbances (original wording) caused by transients in standard microprocessors. By way of a statistical analysis of the instruction sets of six microprocessors (8085, Z80, 8086, 8048, 8051, 8096) the probability of a microprocessor deadlock was determined. In [Du88] and [Ch91] the susceptibility of the HS1602 microprocessor to upsets caused by current- and voltage transients was analyzed through simulation-based fault-injection at device level and gate level. Focus was put on error propagation, however, not towards higher abstraction levels but towards the microprocessor pins. On the “effects of transient gate-level faults on program behavior” was reported in [Cz90]. Gate level transients had been injected into a simulation model of the IBM PC RT. The effects were not presented in terms of instruction execution errors, but in terms of the completion states of the workload software. A microprocessor *error behavior function* (EBF) that maps faults onto errors at the functional level was presented in [Ri94]. Two state elements were assigned to the processor model: a user visible state (UVS) and an internal state (IS). The UVS includes the registers that are defined by the instruction set, and it contains all state information that is passed “from one instruction to the next”. The outputs of the processor are defined by the type of bus access states (BAS). Seven error classes were then specified “by their effect or function on the UVS and BAS”. Fault-injection (single bit-flips and simulated pin level faults) was used on models of the TRIP processor for obtaining two EBFs. Concrete error examples, such as the error(s) caused by a faulty adder circuit, have however not been presented. Although the functional level chosen in the paper is stated to correspond to the programmer’s view of a processor, the model still is too implementation oriented. The model incorporates the

buses and their states (fetch, read, write, idle) as well as the execution phases (fetch, decode, execute, read, write). From a strict software application perspective these details should be invisible. Another publication concerned with fault mapping is [Yo93, Yo96]. The work tries to bridge the gap between accurate but time-consuming gate level fault-injection, and rapid but inaccurate high-level fault-injection. Fault-injection experiments were carried out on a RTL model and a gate level model of the IBM ROMP processor. The RTL model was called “naïve” because it had been constructed without detailed knowledge of the hardware implementation. In total 91 different machine instructions in execution have been separately investigated. The results showed that the RTL fault model covered 97% of the gate level faults in the mean.

5.2.3 State of the Art

The approaches taken in [Ri94] and [Yo96] certainly are the most far-reaching from a strict software perspective. The UVS is similar to the storage space but does not contain all essential state elements. The RTL statements from [Yo96] are similar to service specifications, however, the essential and the implementation-specific state elements of the ROMP microprocessor are not clearly separated. Although all approaches try to obtain a more distant view onto a microprocessor, the presented models contain – at least in parts – remnants of the implementation, such as the bus states. Concrete examples of errors in the execution of the individual machine instructions (service errors) were not given. This also applies to similar publications not reviewed here. Most of the publications are nevertheless encouraging in that the used hardware models, the injection techniques and the simulation environments would have allowed to obtain concrete service errors. Because the observation of the fault effects onto the workload software did belong to the objectives of many experiments, the injected faults have actually passed the service level in these experiments, though without being noticed or captured. Some authors also reviewed their work from an inter-processor portability perspective. Error models at RTL and register level were indicated or proposed to be suitable to form a portable set of injectable errors. Summarizing, it can be stated that there is a demand for a high level error behavior model of a microprocessor. The model should also provide some inter-processor portability. The service level has not yet been considered, though.

5.3 Error Behavior

Any service modifies the storage space according to hardware-defined rules. It does so in terms of transformations. A service can be considered as an enclosed set of transformations. These are performing according to what is stipulated by the service specification.

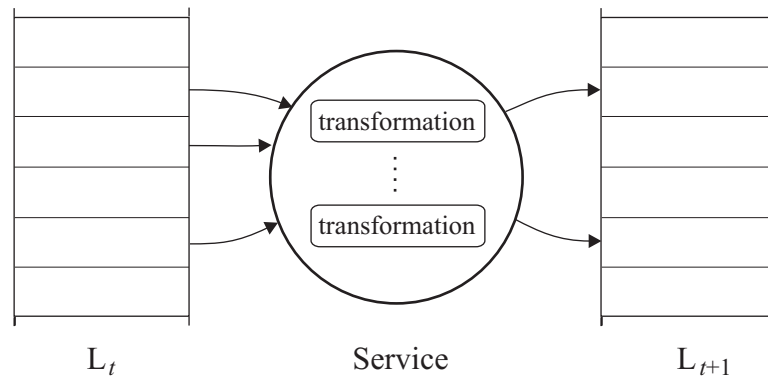


Figure 5.2: Transformations within a service

The term transformation is kept general here. A transformation can be the process of calculating the sum of two input operands, it can be the process of addressing the input and output locations, it can also be the process of instantiating the next service. In any case, random faults in the processing hardware influence the transformations and thus cause service errors.

5.3.1 Error Classes

A service error, if effective, produces data faults in the storage space. Service errors may be classified by their effects onto the storage space.

Placement Errors: Placement errors denote those service errors in which the output data is correct, but is placed in the wrong storage locations. Placement errors are due to random faults in the addressing mechanisms. Note that incorrect input data (e.g. used for indirect addressing) does not cause a placement error since the service is doing right, through on wrong data.

Processing Errors: Processing errors denote those service errors in which the output data is incorrect, but is placed in the right output locations. Again, the cause are random faults in the corresponding circuitry involved.

Indication Errors: One of the tasks of a service is to update the PC according to the input (e.g. interrupts) in order to indicate and identify the successor service about to be instantiated. Indication errors (more precisely: successor indication errors) denote those errors in which the PC is updated incorrectly. These errors result in control-flow errors at process level.

Instantiation Errors: Although the PC may be updated correctly, when finally fetching and decoding the instruction code, the successor service instantiated may be different from what is requested by the instruction code. Random faults affecting the decoding unit are one cause. In this case the controlling process still is on the correct program path (no control-flow error), but is being delivered a wrong service by the provider.

5.3.2 Timing Errors

Some hardware faults can cause the services to deviate from their specified timings. Timing errors may be related to certain transformations within a service or to the service on the whole. A service may also become stuck in which case all activity is stopped. The controlling process then implodes into a halt. Timing problems usually result in a too late delivery of service (delay) rather than in a shortened delivery. However, no statement on concrete timing errors can be made here. Timing problems certainly are not to be neglected, but given the fact that obtaining concrete models on the error behavior of a microprocessor already is difficult enough (Section 5.2.3), timing problems legitimately can be postponed for future consideration. Anyhow, if a slight variation in the timing of a service causes the controlling process or the embedded system to be non-fault-tolerant, then something fundamental is wrong with the system from the start.

5.3.3 Multiple Affections

Depending on the location in the processing hardware, a permanent fault is likely to affect several services, that is, a single hardware fault can cause a number of services to become erroneous. A defective address unit will cause errors in most of the services. A permanent fault in the ALU may at least affect the services such as ADD, SUB, AND, OR and similar. Owing to the fact that the details of the processing hardware are kept confidential by the microprocessor manufacturers, the relationship between a given random fault and the set of

services potentially affected often remains vague. Sometimes it is possible to pick out from the processor documentation an idea of which services depend on which functional units. Again, no statement on concrete relationships can be made here.

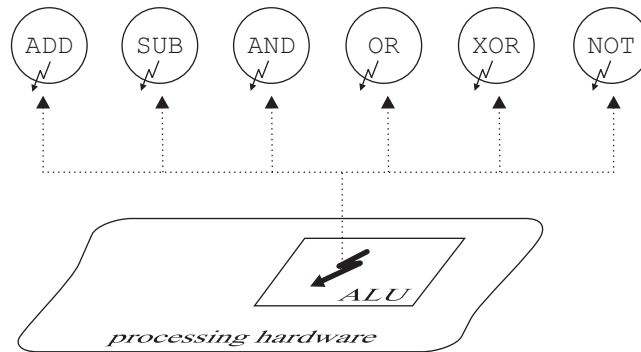


Figure 5.3: Multiple affection example

Multiple affections are not necessarily limited to permanent faults. A transient may latch up in some flip-flop in the processing hardware and may influence the subsequent services for a certain period of time. In [Ch91] the induced transients were observed to almost disappear after 9 clock-cycles. This finding however cannot be generalized for obvious reasons.

The only statement that can be made is that it is not unlikely that the controlling process is not only affected by a single service error, but is affected by a collection of service errors over a certain period of time. The controlling process thus may be facing a timely spectrum of faults. Although the F set is defined in [Ar90] to be the input domain the fault-injection experiments, which is true for each individual injection, the input domain in evaluating the fault-tolerance of the controlling process must be generalized to be what will be called in Chapter 6 a *fault scenario*. A fault scenario is a constellation of service errors that is applied to the controlling process. A fault scenario may of course contain just one particular service error.

5.4 Error Behavior of Combinational Circuits

In this section fault mapping investigations on customary combinational circuits used in functional units of microprocessors are presented. The obtained

findings can be used for describing or creating realistic service errors to be used in fault-injection experiments.

5.4.1 Retrospective

Many fault-injection experiments were conducted by applying bit-flips to the registers of a microprocessor. For a n -bit register, there are $2^n - 1$ choices to falsify the content distinctly. In order to avoid combinatorial explosion, the experimenters have limited the amount of actually injected faults. The number of simultaneous bit-flips to be applied as well as the choice which bits to be affected however was chosen ad hoc in most cases. In [Yo96], as an example, single bit-flips for each register bit, zero-all-bits and set-all-bits were used. This reduced the fault space to $n + 2$. In [Ba90], two-bit-compensating faults were applied, and in [Ca98] the injected bits seem to have been selected rather randomly.

Undoubtedly, the number of faults to be injected must be somehow reduced. The question arises, whether the reduction process can be guided by some à-priori knowledge or whether there is no way other than to choose the faults arbitrarily. The answer certainly is important to those aiming at injecting realistic representatives of random faults.

Example 5.1: It shall be assumed that a microprocessor has just delivered the service $R := A \text{ ADD } B$, where A and B are registers holding the numerical values 2 and 3, and where R is a register holding the sum 5. If the register R is about to get fault injected for the purpose of emulating an effective stuck-fault in the adder circuit involved in the delivery of the ADD -service, then the question is, which bits of R have to be modified, or, in arithmetic terms, which value shall R be assigned to in order to represent a realistic fault situation? Are all combinations of bit-flips, thus all values other than 5 even likely or do stuck-faults in the adder circuit produce a certain error pattern (Figure 5.4)?

The answer is, that stuck-faults indeed produce a certain error pattern in the output of the adder circuit. For the above example, assigning the value 7 to the register R is a good choice, while using 8 or 10 is not favorable. The reason for this decision will be shown.

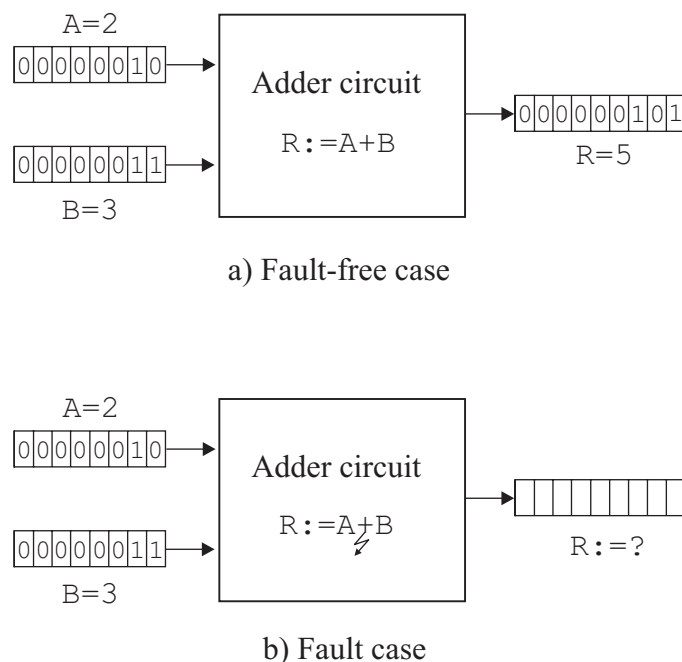


Figure 5.4: Addition $R := A + B$

5.4.2 Introduction

The execution of a particular machine instruction involves particular functional units of the microprocessor. These units contribute to the delivery of the requested service. A fault that has become effective in such a functional unit is likely to cause a service error, that is, at least one of the transformations of the service may get affected and will then differ from the specification. For the example given previously, the **ADD** service is likely not to perform $R := A + B$ in case of a faulty adder circuit, but some other function $R := g(A, B)$. The function g is a function of the affected circuit and thus it must be possible to give some shape to g by investigating the error behavior of the corresponding circuit.

Many circuits used in functional processor units are pure combinational circuits or have at least combinational equivalents. This certainly applies to common adders, multipliers, barrel shifters or multiplexers, for instance. Although it is difficult to obtain specific gate level models for current microprocessors, the fundamental functioning and the structure of these circuits is nevertheless well known. It is assumed here that these basic principles still apply to most of the current processors used in safety-critical embedded systems. In

any case, no statement will be made here about a particular microprocessor. The intention of this section is not to present an extensive research on existing microprocessor circuits, but to demonstrate that the circuits used in this investigation show a typical error behavior, and that there is a strong indication, that the findings are universal and thus can be carried over to other circuits.

TERMINOLOGY

Figure 5.5 shows an arbitrary combinational circuit with two data inputs, one control input and one data output. The data inputs are connected to the registers A and B, and the output is connected to the register R. The control input is fed by some control register C whose content determines the operation to be carried out on the inputs. While A, B, C and R denote registers, thus storage locations, the lower-case letters a, b, c, r denote their corresponding content — a value that will be expressed either in terms of an unsigned decimal integer or as binary representation. For the output register R, the lower-case letter r depicts the value in the fault-free case, while r' depicts the value when a fault is present in the circuit. The number of used bits of a register, also termed *size*, is denoted by a Greek letter as follows.

$$\text{Size A} = \alpha, \text{ size B} = \beta, \text{ size C} = \chi, \text{ size R} = \rho.$$

Register A, for example, can hold 2^α distinct values. The sizes of the registers are determined to be equal to the number of input or output lines associated, that is, no bit of a register is left unconnected.

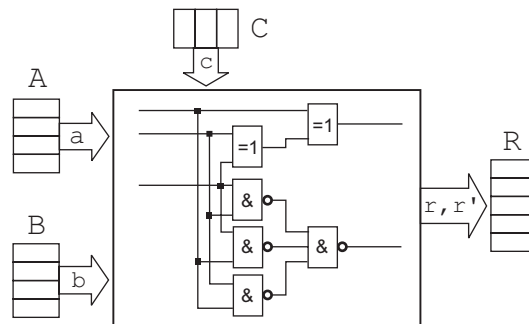


Figure 5.5: Circuit constellation

The circuit consists of basic gates as shown in Figure 5.6 (page 88) for the 4-bit Adder circuit which will be the first simulation example to be discussed

straightaway in Section 5.4.4.1. The inputs and outputs of the gates are called *ports* .

5.4.3 Gate Level Fault-Simulation

The investigations are carried out through fault-simulation at gate level. The injected faults are single stuck-faults, that is, there is only one fault in the circuit per simulation. A *good-simulation* is the fault-free logic simulation with fixated inputs \mathbf{a} and \mathbf{b} for the purpose of obtaining \mathbf{r} . A *fault-simulation* is the logic simulation with fixated inputs \mathbf{a} and \mathbf{b} , and a particular fault f for the purpose of obtaining $\mathbf{r}'(f)$. A *simulation run* denotes the combined sequence of one good simulation followed by one fault-simulation, while the inputs \mathbf{a} and \mathbf{b} remain fixated during the complete run. The run unravels the error evoked in \mathbf{r}' , if any. A *simulation section* denotes a set of simulation runs, where all runs use the same input values \mathbf{a} and \mathbf{b} , but each run uses a different fault.

The faults are injected into the input and output ports of the gates. However, injecting a stuck-at-0 fault into a port that already has the logic value 0 makes no sense because this procedure poses no error to the circuit. The similar applies to stuck-at-1 faults when injected into ports already set to logical 1. It is to emphasize that the primary goal of this investigation is to force a logical fault into the circuit in order to observe the effects onto the output \mathbf{R} . This goal is achieved when at least one gate behaves different from the fault-free case. Therefore the stuck-faults are chosen to be the inverse of the actual logic port value. The port values are known from the preceding good-simulation and thus the input values do not change during a simulation run. After fault-simulation the fault is retracted. The next run starts with a good simulation in which the nodes of the circuit are brought back to their fault-free values in accordance to the applied input. Another port is then selected and the injected fault will force the port value to its logic inverse during the fault-simulation. When all fault locations have been visited, the next input combination of \mathbf{a} and \mathbf{b} is applied to the circuit, and the next section with p simulation runs follows. The letter p denotes the number of ports that the faults are injected into (the visited ports). Some of the investigated circuits have output lines that are not used during a particular simulation. All gates driving these unused outputs are removed from fault-injection, therefore the number of visited ports p sometimes is less than the total number of ports of the circuit.

For a given circuit having the inputs A and B, and p fault locations, the maximum number of distinguishable simulation runs is

$$N_o = 2^\alpha \cdot 2^\beta \cdot p. \quad (5.1)$$

The size of the control input C is not taken into account because the investigation of each circuit was carried out separately for each of its operation modes, that is, the value of c was *one* in each investigation. Some circuits anyway have no control input since they perform only one specific function. The number of simulation runs that have been carried out on a circuit and a given operation mode is depicted by N . For all investigations presented here, is $N = N_0$. The simulation software has been written by the author. The circuits are taken mainly from textbooks.

Fault model:	Single stuck-at-0 and stuck-at-1 faults, adjusted to the logic value of the fault location (acting like inverters).
Fault locations:	Input and output ports of the gates driving the output under observation. Total number = p .
α, β, ρ :	The size (number of bits) of the input registers A, B, and of the output register R.
<i>Simulation run</i> :	Sequential combination of one good simulation and one fault-simulation, the latter with one particular fault, while the inputs remain constant ($\mathbf{a} = \mathbf{a}_i, \mathbf{b} = \mathbf{b}_j$).
N :	Total number of simulation runs for a given circuit, such that all input combinations are applied onto all faults.
Feature:	None of the injected faults is ineffective at gate level, that is, each fault causes the affected gate to act different from the good case. Therefore the core error behavior of the circuit is revealed.

Table 5.1: Summary fault-simulation procedure

5.4.4 Error Model

The goal of the investigations is to find out of what kind the effects of the injected faults onto the output register R are. More precisely, if the output of the circuit is in error, to which degree will \mathbf{r}' differ from the correct value \mathbf{r} and how can this error best be described? Obviously, the fault effects should not be itemized for each particular input combination and each individual fault,

which would sum up to N_0 information entries in total. Two reasons forbid such a procedure anyway. First, the circuits investigated are exemplary circuits which could have been constructed using another internal structure, thus other gates and different wiring while maintaining the same logical function. Second, and in consequence of the first, focusing on a very particular fault is much too detailed in this situation as this particular fault and its location may not exist in the real hardware that the error model may be used for. Furthermore, usually the least of all is known about very individual faults. For the purpose of portability, the error model should average over all faults and all input combinations. The model thus becomes a probability distribution function of the particular errors observed.

ERROR PROBABILITY DISTRIBUTION

In statistical terminology, each simulation run is an experiment. The outcome of the experiment is the observation of one particular error $e_k(\mathbf{r}, \mathbf{r}')$ out of a set of possible errors. This set of errors forms a *sample space* Ω defined on \mathbb{R} and it includes the null-error which depicts the outcome $\mathbf{r} = \mathbf{r}'$. The errors $e(\mathbf{r}, \mathbf{r}')$ in Ω are scalar quantities which are derived through some measure-function mapping the difference between \mathbf{r} and \mathbf{r}' onto an ordered scale. Counting the number of flipped bits in \mathbf{r}' with respect to \mathbf{r} is a commonly used measure to describe $e(\mathbf{r}, \mathbf{r}')$, for instance. A random variable E shall now be defined on Ω . For a single simulation run, E takes exactly one error from the set. For a simulation section, thus $\mathbf{a} = \mathbf{a}_i$ and $\mathbf{b} = \mathbf{b}_j$ and all p faults, E is assigned a distribution of the particular errors $e_k(\mathbf{r}, \mathbf{r}')$ observed. A complete simulation, thus all input combinations onto all faults, yields an error distribution function in E for a given circuit and operation mode. This function describes each individual error in \mathbf{r} and its likelihood of occurrence, independent of the input values, and under the presumption that a stuck-fault is present at the very moment at which the circuit performs an operation on the inputs, while all faults are to occur with the same probability. It should be noted, that the error model applies to both permanent and transient stuck-faults. The only requirement is, that a single fault is active at the very moment at which the input signals propagate through the circuit. The model thus describes the worst case, namely that either a permanent fault (which acts like an inverter) is present or that a transient fault occurs and flips the logic value of some port whenever the circuit is used.

Before continuing on the error model, the 4-Bit-Adder, ADDER4, shall be introduced in brief. The circuit serves as exemplary basis for the subsequent discussion.

ADDER4: The function of the 4-bit Adder shown in Figure 5.6 is to perform an addition on the inputs **A** and **B** such that $R:=A+B$. The inputs are each of 4-bit size, yielding 256 distinct input combinations to the circuit. The output **R** is of 5-bit size. The number of ports (= fault locations) is 96. In total, $N = 24576$ simulation runs were carried out on the circuit.

5.4.4.1 Bit-flip Distribution

As mentioned beforehand, a commonly used measure to describe the error $e(\mathbf{r},\mathbf{r}')$ is to count the number of bits flipped in \mathbf{r}' with respect to \mathbf{r} . For the 4-bit Adder ADDER4 the distribution of bit-flips obtained from simulation is shown in Table 5.2. Since the output **R** is of 5 bits, the number of bits to flip ranges from 0 to 5.

Bit-flips	%
0	25.68
1	55.60
2	12.11
3	4.72
4	1.69
5	0.20

Table 5.2: Bit-flips in the output of the 4-Bit Adder

In 25.68% of an effective fault present, its effects did not propagate to the output of the circuit, that is, no bits in **R** flipped and \mathbf{r} was identical to \mathbf{r}' . A fraction of 55.6% of the fault-injections caused one bit to flip, while in 18.72% two or more bits flipped. The following finding is not new.

Not every stuck-fault effective at the logic level produces an error in the output of the circuit.

From the bit-flip distribution it however cannot be drawn, which *particular* error is the most likely. Obviously, focusing on the number of bits flipped is not very expressive for practical fault-injection experiments. First, it is

not known which bits actually were affected and thus which bits should be manipulated for the purpose of injecting an error. Second, as bits are a means to an end rather than the means, it should be observed what the fault effects mean to the *integer* represented by the collection of bits. Therefore a more descriptive error is to join the traditional bit-flip error.

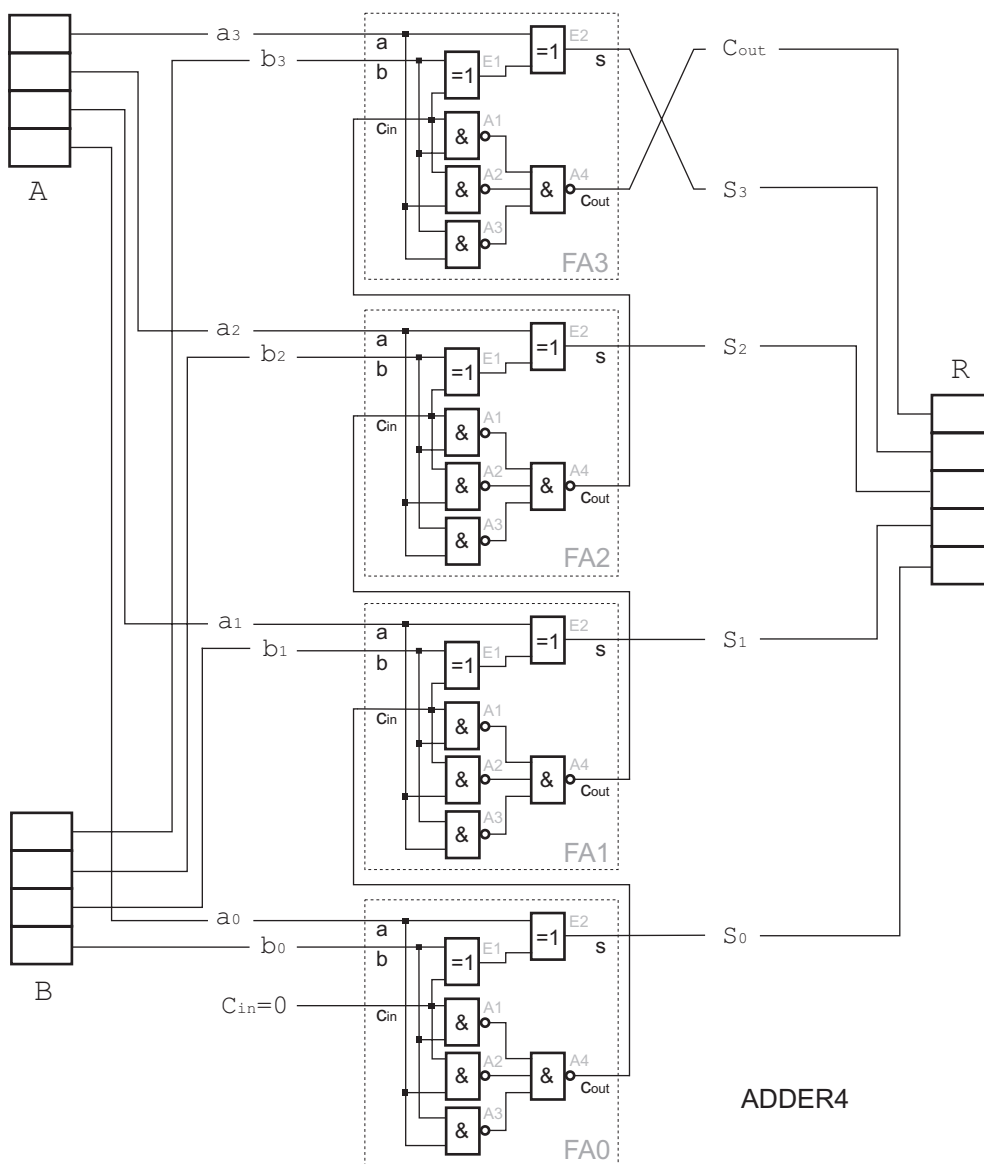


Figure 5.6: Circuit diagram 4-bit-Adder ADDER4

5.4.4.2 The Arithmetic Error

In coding theory, especially with arithmetic codes, an error can be described by the *arithmetic error*. The following informal definition is taken from [Wa78 p.35] with slight adjustments.

In the arithmetic error model, instead of viewing an error as changing a certain number of bits in a binary vector, we consider its effect on the integer represented by the vector. Thus the effect of an arithmetic error on the binary representation of an integer r is to change it to the binary representation of $r + e$, where e is called the error value.

The error value, for convenience just termed error here, is a signed integer-value representing the difference between the two unsigned integers \mathbf{r} and \mathbf{r}' such that $\mathbf{r}' = \mathbf{r} + e$. Because the range of values of \mathbf{r} is limited by the size of the register \mathbf{R} , arithmetic addition and subtraction must be performed within the scope of the number circle spanned by \mathbf{R} (modulus- 2^ρ calculus). The size of e is equal to the size of \mathbf{R} , hence e may take any value from the integer range

$$E = [-(2^{\rho-1}), \dots, 0, \dots, +(2^{\rho-1} - 1)] \quad (5.2)$$

and thus depicts the shortest distance between \mathbf{r} and \mathbf{r}' on the circle, as shown in Figure 5.7. Demanding by definition, that e **must** take a value from E avoids two potential problems. The first problem, owing to the modulus arithmetic, is that e could be substituted by $(e + n \cdot 2^\rho)$, where n is an arbitrary integer. The error mapping would become ambiguous then. The second problem addresses the question, which way to take from \mathbf{r} to \mathbf{r}' on the number circle when both are located exactly opposite of each other, thus when $\mathbf{r}' = \mathbf{r} + 2^{\rho-1}$ (see Figure 5.7c). The answer is to go counter-clockwise, because the distance $-2^{\rho-1}$ between \mathbf{r} and \mathbf{r}' can be represented by e whereas the same distance going clockwise, $+2^{\rho-1}$, can not.

Definition 5.1: The arithmetic error $e(\mathbf{r}, \mathbf{r}')$ is a signed integer of size ρ , defined on two unsigned integers \mathbf{r} and \mathbf{r}' , each of size ρ , taking values from the set

$$E = \{-(2^{\rho-1}), \dots, 0, \dots, +(2^{\rho-1} - 1)\}$$

for the purpose of unambiguously describing the error in \mathbf{r}' with respect to \mathbf{r} , such that $\mathbf{r}' = \mathbf{r} + e(\mathbf{r}, \mathbf{r}')$ in modulus- 2^ρ calculus. The set E contains 2^ρ elements.

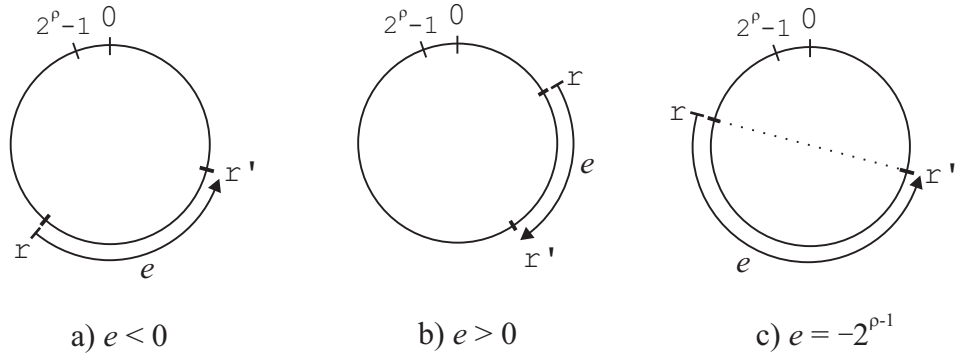


Figure 5.7: The arithmetic error e on the number circle.

For simplicity, the error $e(\mathbf{r}, \mathbf{r}')$ will be notated just as e . The error $e = 0$ is the *null-error*, depicting the outcome $\mathbf{r} = \mathbf{r}'$. In order to ease the distinction between the null-error and the remaining errors $e \neq 0$ in the further discussion, the set E is split into a set E_0 and a set E_E . The former contains the arithmetic null-error and the latter contains all remaining arithmetic errors.

$$E = E_0 \cup E_E = \{0\} \cup \{-(2^{p-1}), \dots, -1, 1, \dots, +(2^{p-1} - 1)\} \quad (5.3)$$

The null-error often is not termed as such, rather it is said that *no error occurred*. This terminology will be used in the following figures when indicating the frequency of occurrence of elements originating from either set. In the figures shown shortly, NOERR is the percentage of occurrence of the null-error and ERR is the percentage of the remaining errors $e \neq 0$.

5.4.4.3 Arithmetic Error Distribution

The distribution in the random variable E for the 4-bit Adder ADDER4, when the particular errors have been collected in terms of arithmetic errors, is shown in Figure 5.8. The x -axis lists the particular arithmetic errors $e(\mathbf{r}, \mathbf{r}')$ observed. Because the output of the circuit is of size 5, the errors may range from -16 to $+15$, thus taking $2^5 = 32$ distinct values. The y -axis indicates the frequency of occurrence of each possible error. The frequency is given in percent of the total number of fault-simulations N . It is to note that each of the N fault-injections used to be effective.

Figure 5.8 shows a discrete error distribution function of the errors in the output of the 4-bit Adder for all input combinations when single stuck-faults

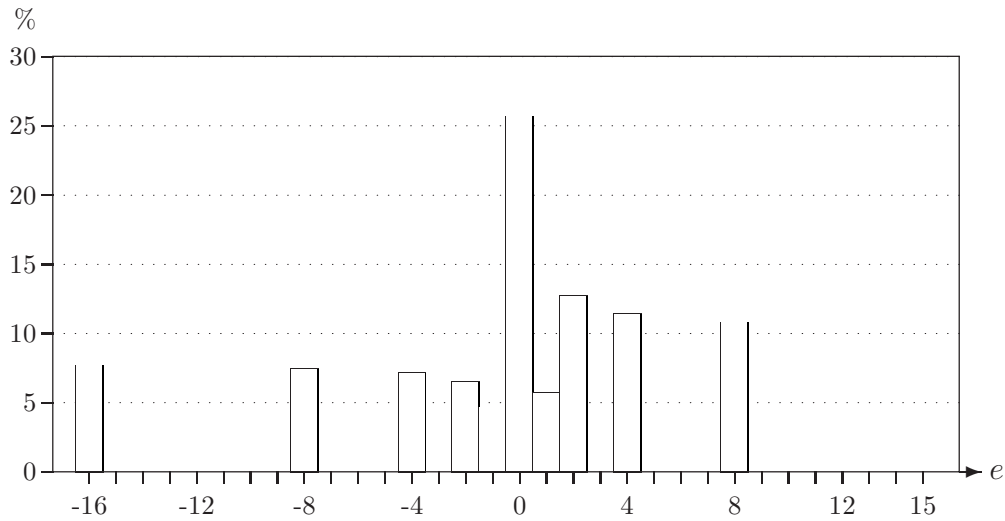


Figure 5.8: Arithmetic error distribution ADDER4

are present and effective at gate level. From the distribution function it is evident that some errors appear more often than others. The null-error, $e = 0$, is the most frequent with a percentage of 25.68. This matches with the percentage of no bit-flips from Table 5.2. However, for the purpose of fault-injection, the remaining errors $e \neq 0$ are of more importance. For fault-injection it is instantly recognizable from the distribution, which particular errors should be selected preferably. As may be noticed, the arithmetic error distribution represents the error behavior of a circuit in a more concrete manner than does the bit-flip distribution

Now the initial question from Section 5.4.1 (Figure 5.4, page 82), how to manipulate \mathbf{r} in order to inject a most likely error, can be answered for this particular circuit: If there is a single fault present in the circuit, then adding an error $e = 2$ to the content of \mathbf{R} is a good choice because this error is the most likely to occur. Thus, if $\mathbf{r} = 5$, as assumed in the example, then the value 7 should be assigned to \mathbf{R} . However, given the fact that the probability of occurrence of the individual stuck-faults is likely not to be equal for the considered faults in reality, the likelihood of the errors e will vary slightly. Thus, taking an error other than $e = 2$, for example $e = 4$ or $e = 8$, is a good choice as well. In any case, there are errors that do not occur, regardless of the real probability of occurrence of the stuck-faults. For example, assigning the values 8 or 10 to \mathbf{R} , as already discouraged in Section 5.4.1, is indeed not favorable because the corresponding errors $e = 3$ and $e = 5$ have the like-

likelihood *zero* for this circuit and this fault model. These errors will never appear. For the ADDER4 circuit, the following finding can be formulated.

⌊ Despite the presence of an arbitrary and effective single-stuck fault in the circuit, there are some arithmetic errors that never occur.

Also it appears from the figure that the arithmetic error distribution follows a certain pattern, which is discussed next.

5.4.4.4 The Power-of-two Errors

Next to the finding that some errors do occur, while others do not occur at all, the set of occurring errors $e \neq 0$ can be specified more precisely for this circuit. From the simulation results it reveals that all errors $e \neq 0$ solely are *powers of two*, regardless of the number of flipped bits.

$$E_E = \{-16, -8, -4, -2, -1, 1, 2, 4, 8\}, \forall e \in E_E : e = \pm(2^n).$$

Thus for the ADDER4, the number of possible arithmetic errors in the output amounts to just 9 (excluding the null-error) and these error values can be summarized by a simple formula. This insight very much eases the selection of errors to inject. Given that the circuit is affected by an arbitrary single stuck-fault, the following statement applies.

⌊ If it is to inject realistic errors, then error values that are to a *power-of-two* are the first choice.

According to [Wa78 p.140], indications on power-of-two errors have already been given in the late 60's and early 70's, as for instance in [Se68]. The findings resulted from analytical considerations of specific circuits ([Se68 p.101] used the Boolean Difference on selected adders). The considerations were however not concerned with any error frequencies.

It should be noted, that power-of-two errors are not identical to single bit-flips. Single bit-flips always cause an arithmetic power-of-two error, but not vice versa. For example, adding $e = 8$ to the value 9 needs two bit-flips, and adding $e = 2$ to the value 14 requires 4 bit-flips.

⌊ Single bit-flips do not fully describe the error behavior of a circuit.

For later use, the powers-of-two contained in the value-range of a signed integer of size ρ will be denoted by P_ρ . The number of elements in P_ρ is $2 \cdot \rho - 1$.

COMPLETING THE ERROR MODEL

Because of the noticeable presence of the power-of-two errors, the errors in E_E will be further categorized by whether they are to a power-of-two or not. The percentage of the former will be denoted by POW2, the percentage of the latter by \neg POW2 as shown in the new error distribution figure for the ADDER4, Figure 5.9. The values are given with respect to both E and E_E , thus describing the absolute fraction (with respect to N) of the power-of-two errors and the relative fraction (with respect to all $e \neq 0$). As can be seen from the figure, the absolute frequency of power-of-two errors for the circuit is 74.32%, which makes a relative fraction of 100%. Thus, if there is an error in the output register R, its value is to the power of two.

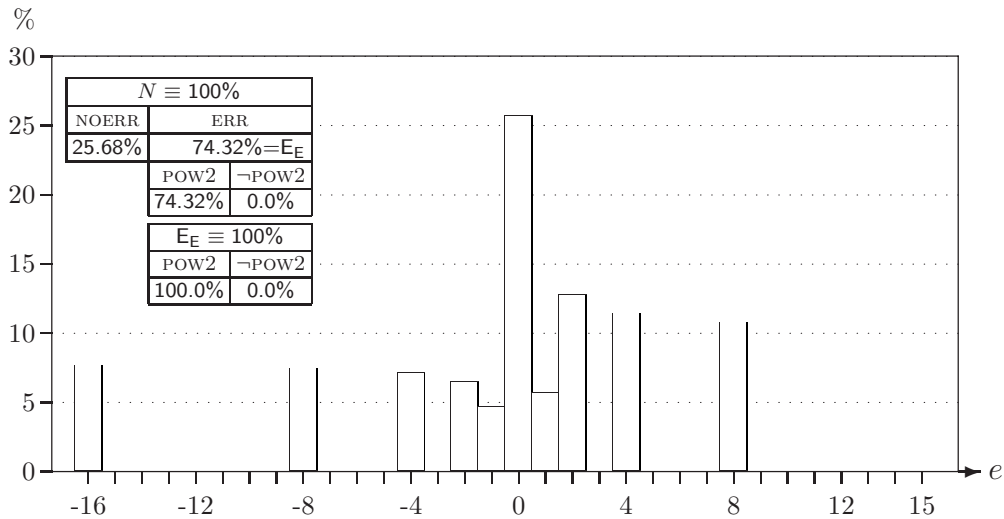


Figure 5.9: Error distribution ADDER4 (with embedded table)

5.4.4.5 Summary

The error behavior of a circuit, while a fault is present and effective at gate level, can be given in form of the traditional bit-flip distribution and in form of an arithmetic error distribution. While the bit-flip focuses on the physical representation of data, the arithmetic error emphasizes the meaning of the stuck-faults to the data's numerical values. The distributions are obtained through fault-simulation. They are frequency distributions, describing the error behavior of a circuit when all stuck-faults occur with the same likelihood. The distribution functions are also probability-density functions, indicating the

probability of occurrence of each error in the output when an arbitrary but unknown single-stuck-fault is present in the circuit. Compared to the traditional bit-flip distribution, the arithmetic error distribution is more expressive.

5.4.5 The Circuits

The following are the circuits investigated. The schematics use the BS3939 standard for logic symbols.

- The 4-bit adder already introduced (ADDER4, schematic shown in [Figure 5.6](#)). The adder is constructed of 4 full-adders (FA).
- A 4-bit carry-look-ahead adder (CLA-ADDER4, schematic shown in [Figure 5.10](#)).
- A 4-bit ALU ALU4, schematic shown in [Figure 5.11](#). The ALU uses the same Full-Adder as the ADDER4 circuit. For space reasons these are shown as modules in the schematic.
- An 8-Bit ALU (ALU8, no schematic) which is of the same structure as the ALU4. This ALU is the 8-bit version of ALU4, having 8 stages instead of 4.
- The 4-bit ALU 74181 (ALU74181, schematic in [Figure 5.12](#)).
- A 4-bit barrel shifter (BS4, schematic in [Figure 5.13](#)).
- A 4-bit multiplier (MUL4, schematic in [Figure 5.14](#)). The Full-Adders used in the circuit are the same as in the ADDER4 and the ALU4.

A table summarizing the essential data of the circuits is given in the following section ([Table 5.4](#) on page 102).

The error distribution functions of the investigated circuits are presented and discussed as well in the following section (page 100 – after the schematics).

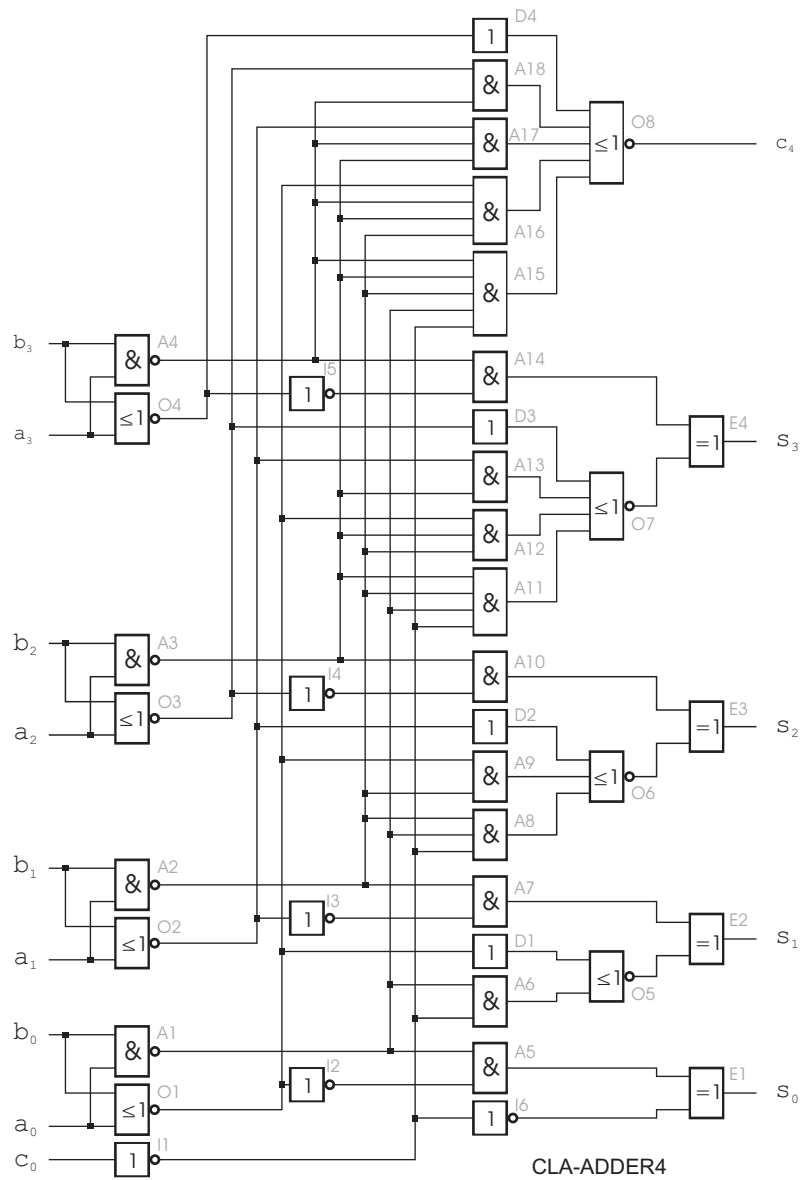


Figure 5.10: Schematic 4-Bit Carry-Look-Ahead Adder CLA-ADDER4

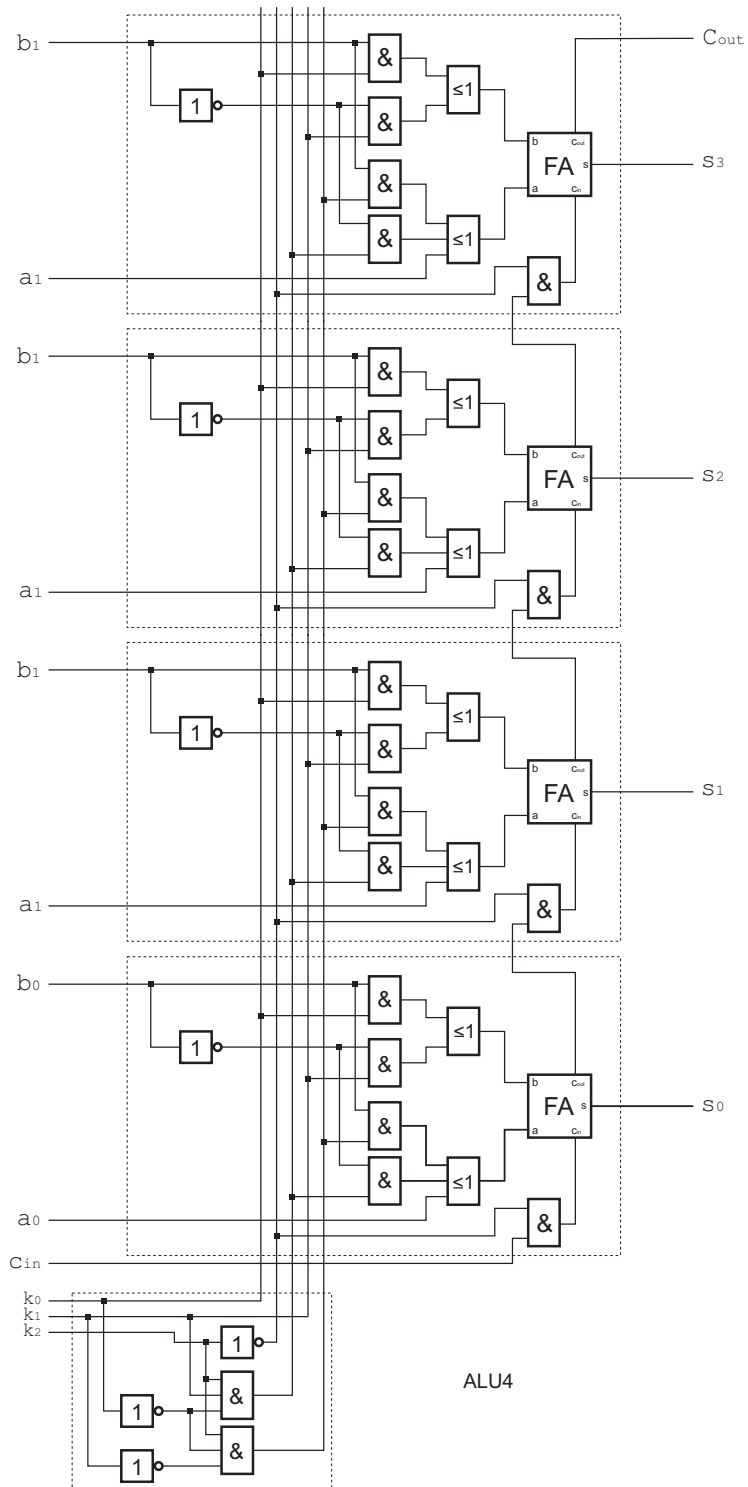


Figure 5.11: Schematic 4-Bit ALU ALU4

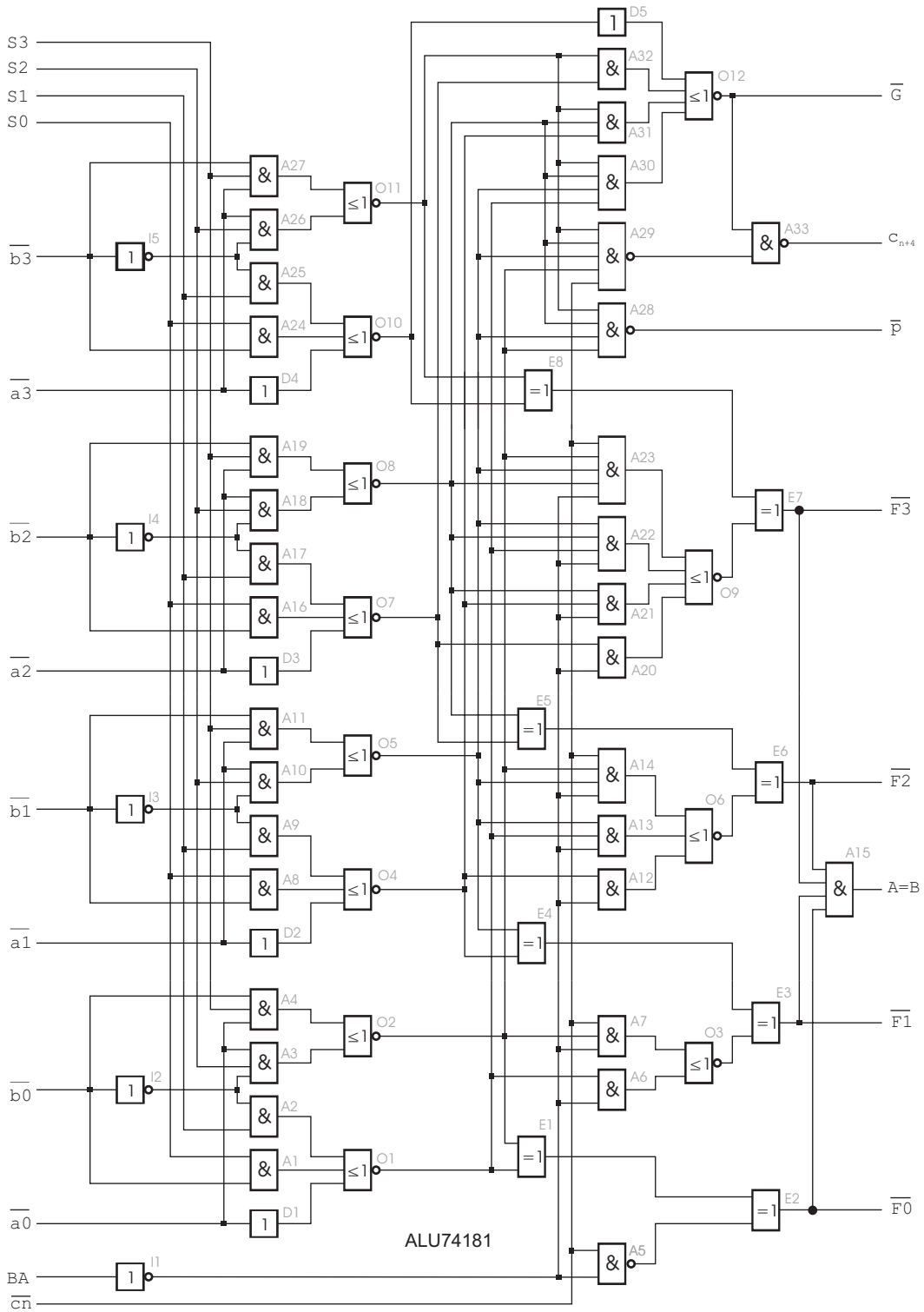


Figure 5.12: Schematic ALU74181

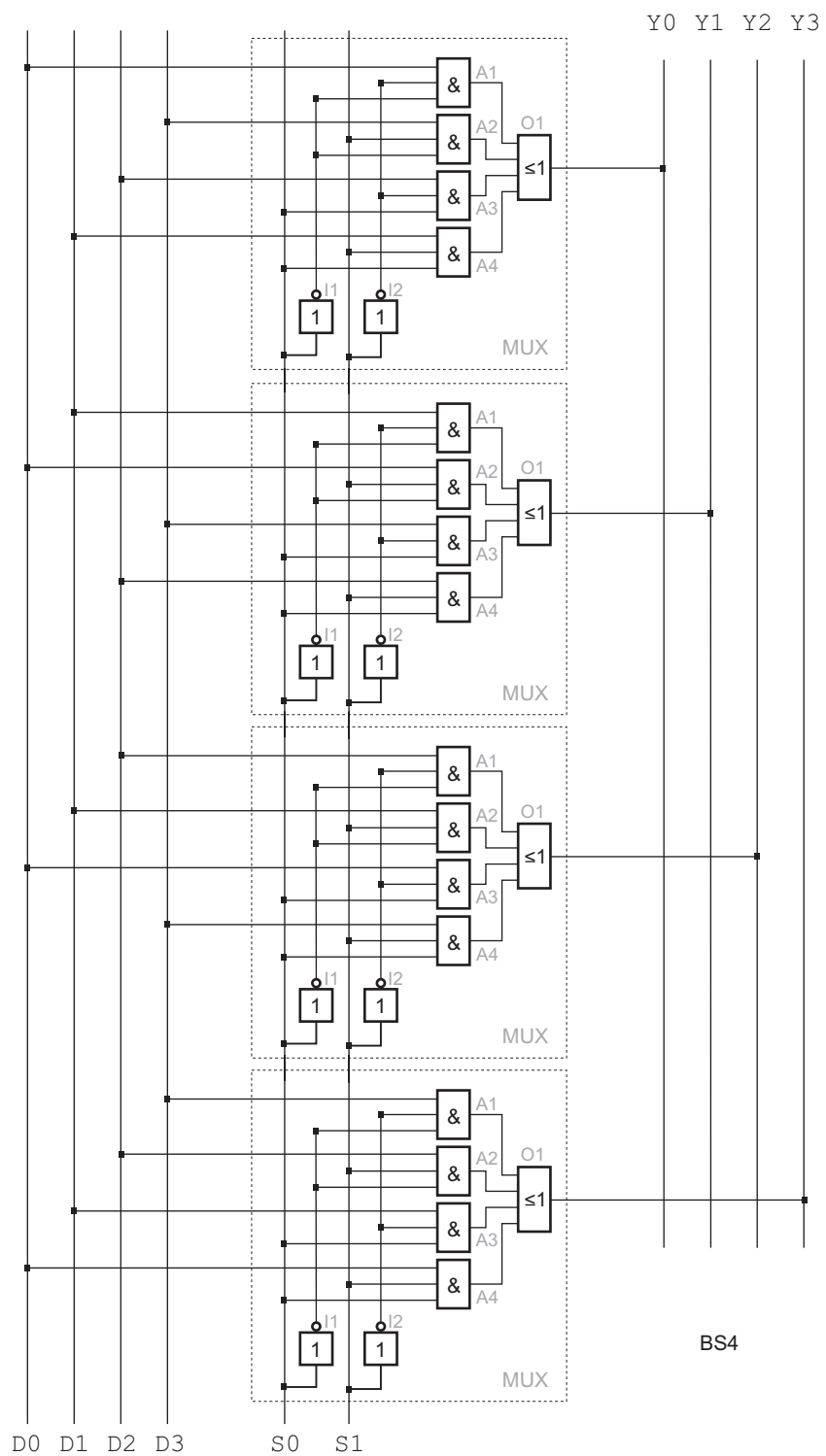


Figure 5.13: Schematic Barrelshifter BS4

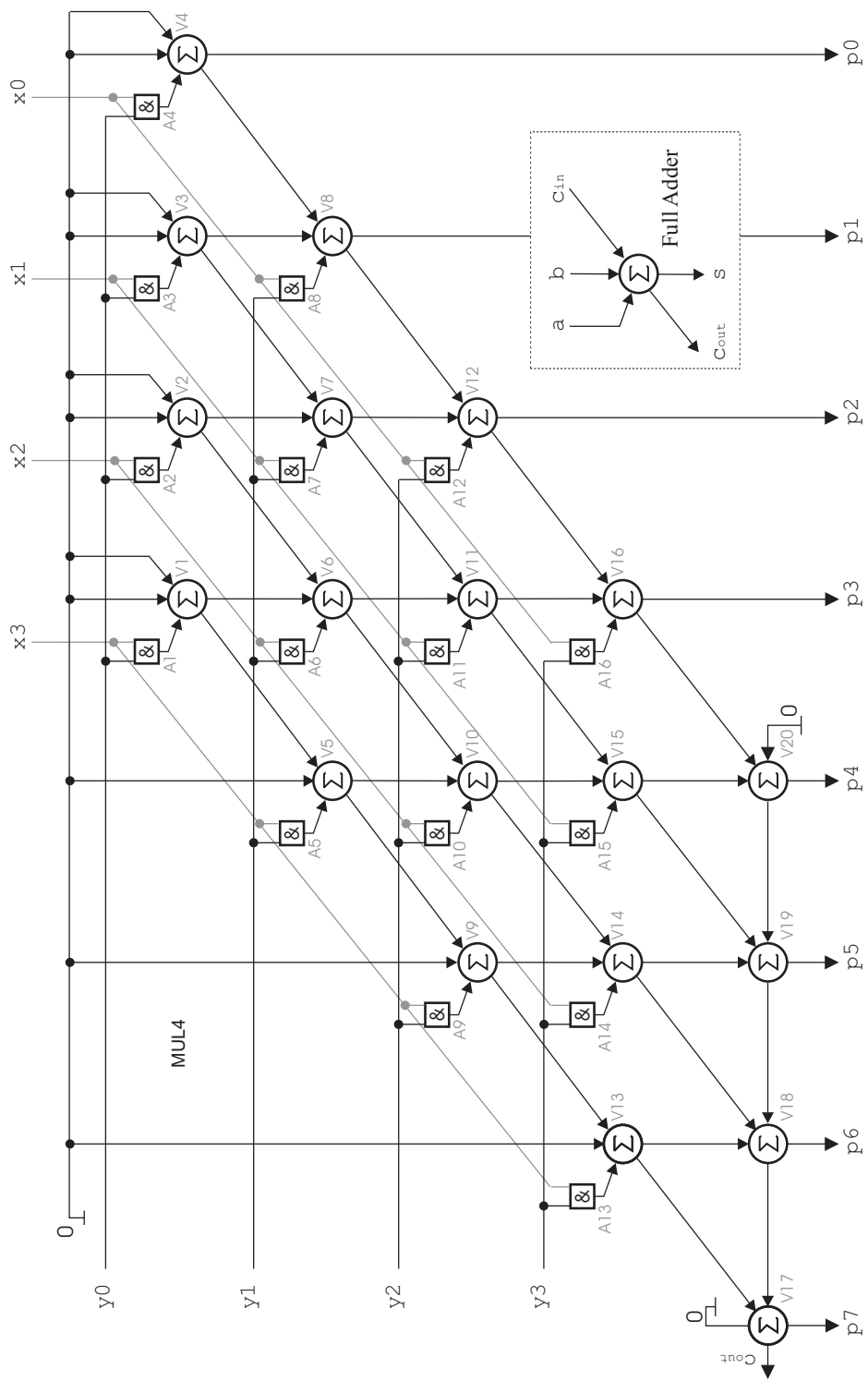


Figure 5.14: Schematic Multiplier MUL4

5.4.6 Single-Fault Error Behavior

In this section, the error behavior of the investigated circuits in the presence of single-stuck-faults is presented. Both the bit-flip distribution and the arithmetic error distribution will be shown. Standing in for all the investigated circuits, the ADDER4 is used to first outline the figures and tables in this section.

THE 4-BIT ADDER ADDER4

The function of the 4-bit Adder from [Figure 5.6](#) is to perform an addition on the inputs **A** and **B**, thus $R := A + B$. The inputs are each of 4-bit size, therefore yielding 256 distinct input combinations. The output **R** is of size 5, yielding up to 32 arithmetic errors from the range $[-16 \dots +15]$ as well as up to six bit-flip errors within the range $[0 \dots 5]$. The number of ports that faults are injected into amounts to $p = 96$ for this circuit. [Table 5.3](#) summarizes the essential circuit information.

Circuit	Mode	Function	α	β	ρ	p	N	Reference
ADDER4	ADD	$R := A + B$	4	4	5	96	24576	fig. 5.15

Table 5.3: Circuit data ADDER4

In total, $N = 24576$ simulations have been carried out on the circuit. The obtained error distributions, in terms of bit-flips and in terms of arithmetic errors, are shown in [Figure 5.15](#) which is the final type of figure used for presenting the circuits' error behavior. The arithmetic errors are displayed in form of a histogram while the bit-flips are embedded in tabular form. Focus will however be given to the arithmetic errors and the term *distribution* will mainly refer to the arithmetic error distribution. The header of the distribution figure identifies the circuit through a short name (here ADDER4) and a mnemonic of its actual operation mode (ADD). The circuit function for the particular operation mode (here $R := A + B$) and the number of simulations that the error distribution is based on, N , is given as well in the header.

THE OTHER CIRCUITS

[Table 5.4](#) summarizes the essential circuit data of the investigated circuits. Again, α and β denote the number of bits of each the inputs **A** and **B**. Some circuit functions, for example $R := A + 1$, only use input **A**. In these cases the input **B** is unused and $\beta = 0$. The size of the output ρ also depends on the circuit

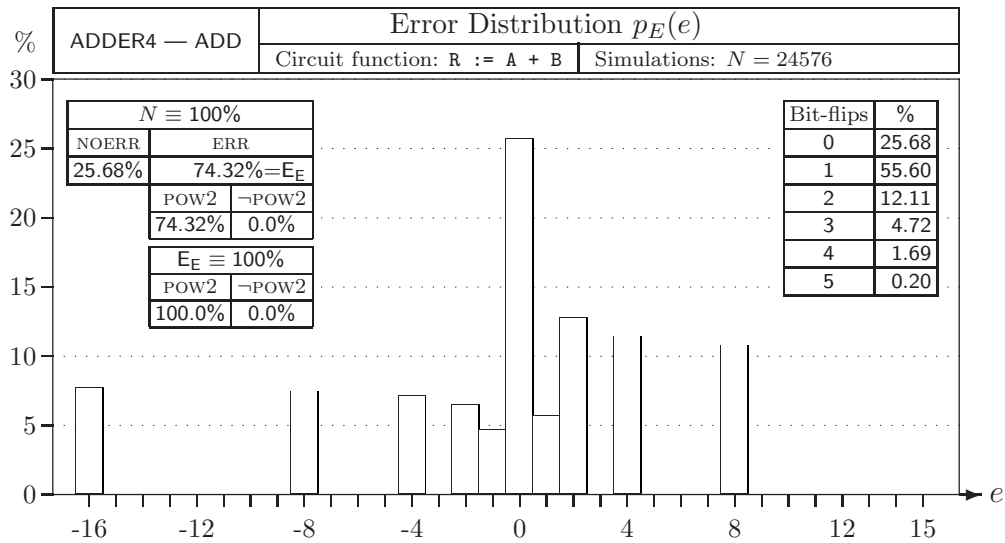


Figure 5.15: Arithmetic error distribution ADDER4 (final version)

function. With some functions, the carry-bit belongs to the output, while with other functions the carry-bit is irrelevant. In consequence, the number of fault locations p varies for a given circuit because all gates driving unused outputs are removed from the simulation. N denotes the number of simulation runs that were carried out on the circuit and particular operation mode. The corresponding distribution figures are referenced in the last column.

ERROR DISTRIBUTION FIGURES

For space reasons, only six error distributions are shown in the following (Figure 5.16 – 5.21). The other figures are shifted to appendix A (page 171).

As appears from the figures, the circuits show specific error patterns in response to the injected single-faults. None of the distributions comes close to an equal-probability distribution, they rather contain a predominant group of errors whose likelihood of occurrence is much higher than that of the remaining errors. This predominant group are the *power-of-two* errors. Of all errors $e \neq 0$, the power-of-two errors make up a proportion between 93.57% and 100%, independent of the circuit and its operation mode (see also column POW2 in Table 5.5 on page 110). Thus, given one of the circuits presented here, and given an arbitrary single-stuck fault present in the circuit, the error produced in the output (if not the null-error) will be a power-of-two error by a chance higher than 93%. Consequently, if it is to emulate realistic errors in

fault-injection experiments, the power-of-two errors are far the most realistic ones when single stuck-faults have affected a circuit.

Circuit	Mode	Function	α	β	ρ	p	N	Reference
CLA-ADDER4	ADD	$R := A + B$	4	4	5	126	32256	fig. 5.16 pg. 103
ALU4	ADD	$R := A + B$	4	4	5	254	65024	fig. 5.17 pg. 103
	ADDINC	$R := A + B + 1$	4	4	5	254	65024	fig. A.1
	SUB	$R := A - B$	4	4	5	254	65024	fig. A.2
	SUBDEC	$R := A - B - 1$	4	4	5	254	65024	fig. A.3
	INC	$R := A + 1$	4	0	4	241	3856	fig. A.4
	DEC	$R := A - 1$	4	0	4	241	3856	fig. A.5
	AND	$R := A \wedge B$	4	4	4	241	61696	fig. A.6
	OR	$R := A \vee B$	4	4	4	241	61696	fig. A.7
	NOT	$R := \neg A$	4	0	4	241	3856	fig. A.8
	XOR	$R := A \oplus B$	4	4	4	241	61696	fig. A.9
THRU	$R := A$	4	0	4	241	3856	fig. A.10	
ALU8	ADD	$R := A + B$	8	8	9	486	31850496	fig. 5.19 pg. 104
	ADDINC	$R := A + B + 1$	8	8	9	486	31850496	fig. 5.20 pg. 105
	SUB	$R := A - B$	8	8	9	486	31850496	fig. A.11
	SUBDEC	$R := A - B - 1$	8	8	9	486	31850496	fig. A.12
	INC	$R := A + 1$	8	0	8	473	121088	fig. A.13
	DEC	$R := A - 1$	8	0	8	473	121088	fig. A.14
	AND	$R := A \wedge B$	8	8	8	473	30998528	fig. A.15
	OR	$R := A \vee B$	8	8	8	473	30998528	fig. A.16
	NOT	$R := \neg A$	8	0	8	473	121088	fig. A.17
	THRU	$R := A$	8	0	8	473	121088	fig. A.18
ALU74181	ADD	$R := A + A$	4	0	5	208	3328	fig. A.19
	ADDINC	$R := A + A + 1$	4	0	5	208	3328	fig. A.20
	ADD	$R := A + B$	4	4	5	208	53248	fig. 5.18 pg. 104
	ADDINC	$R := A + B + 1$	4	4	5	208	53248	fig. A.21
	SUB	$R := A - B$	4	4	5	208	53248	fig. A.22
	SUBDEC	$R := A - B - 1$	4	4	5	208	53248	fig. A.23
	INC	$R := A + 1$	4	0	4	181	2896	fig. A.24
	DEC	$R := A - 1$	4	0	4	181	2896	fig. A.25
	AND	$R := A \wedge B$	4	4	4	181	46336	fig. A.26
	NAND	$R := \neg(A \wedge B)$	4	4	4	181	46336	fig. A.27
	OR	$R := A \vee B$	4	4	4	181	46336	fig. A.28
	NOR	$R := \neg(A \vee B)$	4	4	4	181	46336	fig. A.29
	XOR	$R := A \oplus B$	4	4	4	181	46336	fig. A.30
	NOT	$R := \neg A$	4	0	4	181	2896	fig. A.31
THRU	$R := A$	4	0	4	181	2896	fig. A.32	
BS4	ROL	$R := A \lll B$	4	2	4	128	8192	fig. A.33
MUL4	MUL	$R := A \cdot B$	4	4	8	528	135168	fig. 5.21 pg. 105

Table 5.4: Circuit data

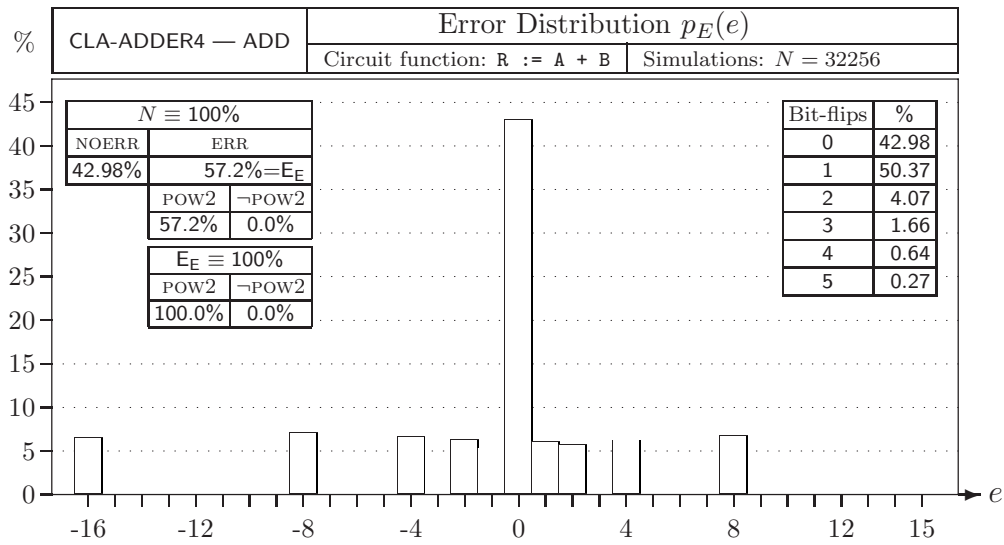


Figure 5.16: Single-fault error distribution CLA-ADDER4

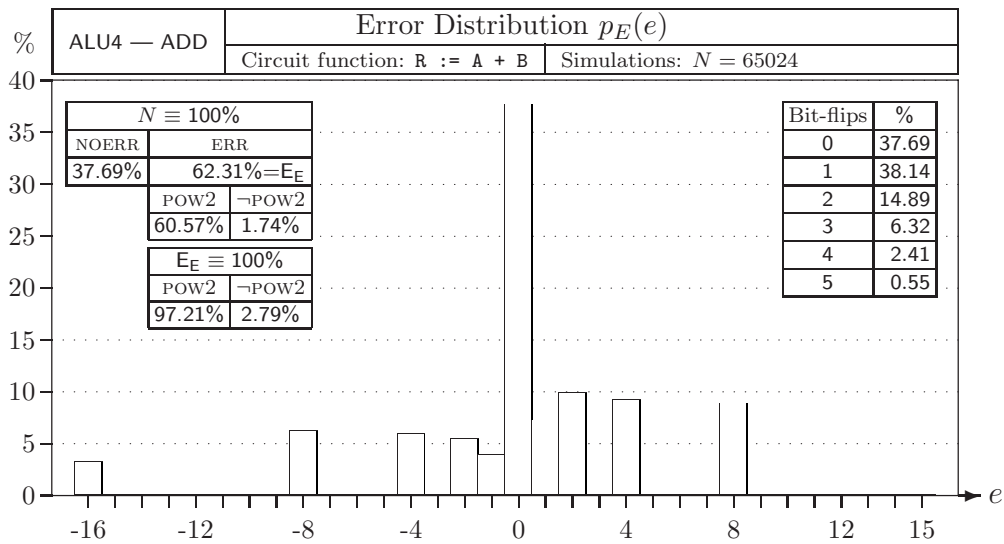


Figure 5.17: Single-fault error distribution ALU4 ($R := A + B$)

Assuming a single-fault affected circuit, the power-of-two errors are the primary choice for fault-injection. They are to occur with a likelihood of 98% on the average.

Because the arithmetic error e is a signed integer and because any signed integer of size ρ can take exactly $2\rho - 1$ values that are to a power-of-two, the error set reduces from $2^\rho - 1$ potential errors to $2\rho - 1$ realistic errors.

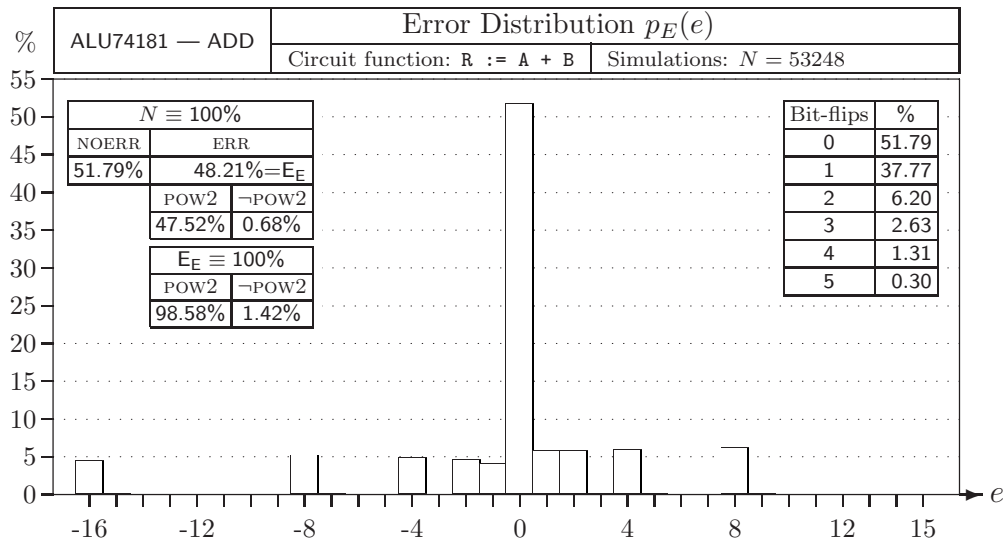


Figure 5.18: Single-fault error distribution ALU74181 ($R := A + B$)

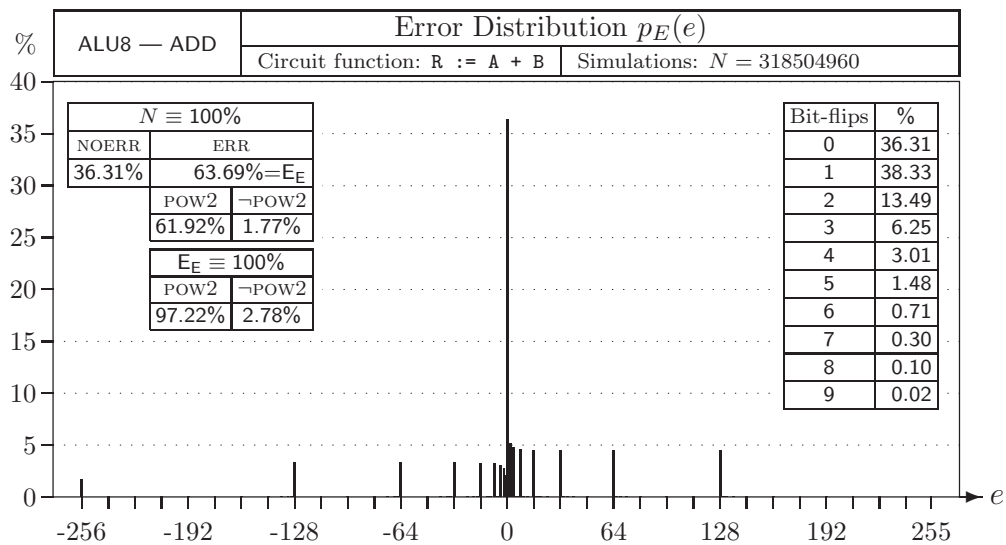


Figure 5.19: Single-fault error distribution ALU8 ($R := A + B$)

DOUBLE-FAULTS

There may arise the objection, that the single stuck-fault model used in these investigations might not be appropriate to certain real circuits or to some extremely rough environments a processor may reside in. On one hand, the single-stuck fault model is a widely accepted and adequate model. Despite all doubts, it serves at least to take a first approach in error modeling and fault-

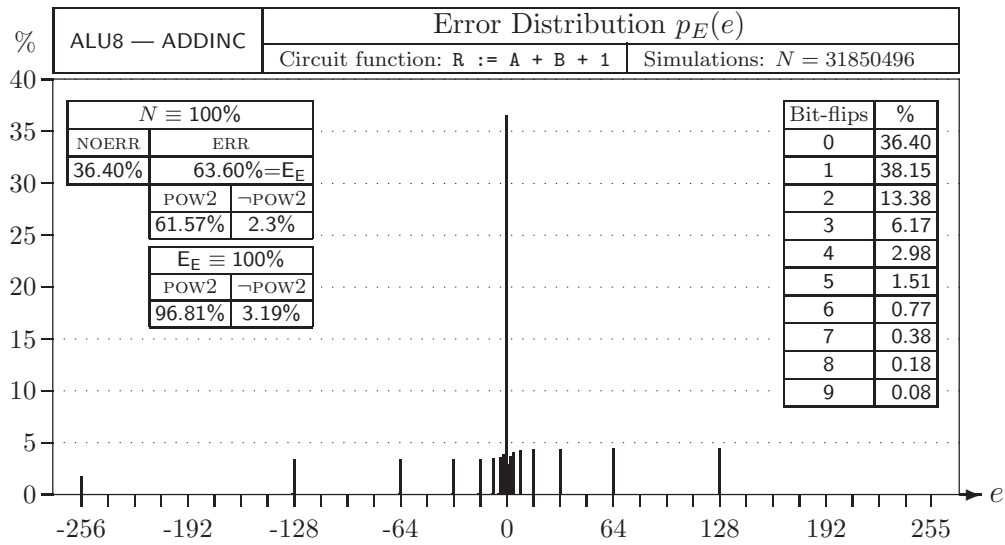


Figure 5.20: Single-fault error distribution ALU8 ($R := A \wedge B$)

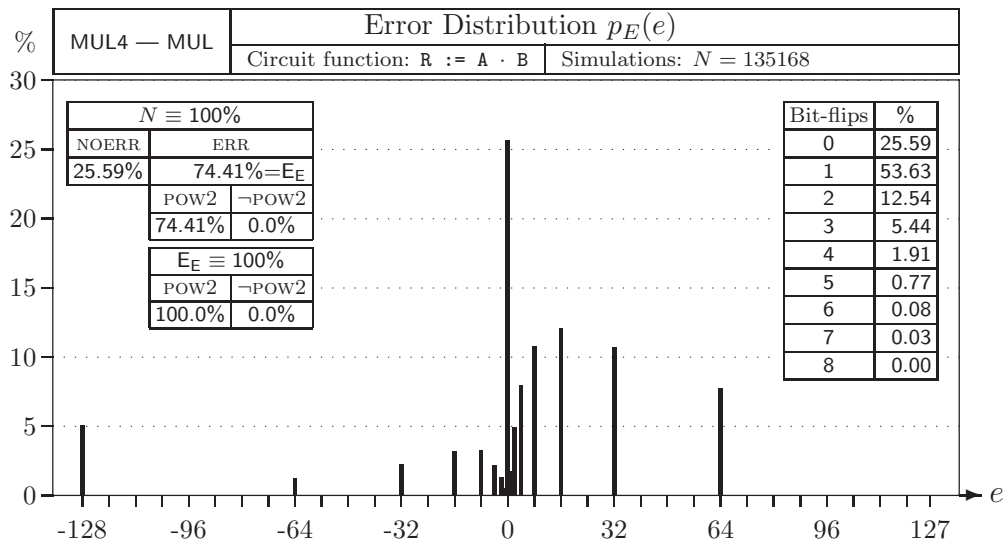


Figure 5.21: Single-fault error distribution MUL4 ($R := A \cdot B$)

mapping. On the other hand, given the tiny layout dimensions of nowadays circuits, radiation and wear-out certainly may cause more than one fault at a time. Therefore the investigations on the error behavior of the circuits have also been carried out using *two* simultaneous stuck-faults.

5.4.7 Double-Fault Error Behavior

In the second investigation presented here, two simultaneous stuck-faults have been injected into the circuits. For a circuit having p fault locations, the number of useful unique combinations of the two faults across all locations is

$$c = \binom{p}{2} \quad (5.4)$$

and the total number of simulation runs for the circuit sums up to

$$N_0 = 2^\alpha \cdot 2^\beta \cdot c. \quad (5.5)$$

Compared to the single-fault injection, the number of simulation runs for a circuit and operation mode using double-fault injection multiplies by the factor

$$\frac{c}{p} = \frac{1}{p} \cdot \frac{p!}{2!(p-2)!} = \frac{(p-1)!}{2!(p-2)!} = \frac{p-1}{2}. \quad (5.6)$$

For the 4-Bit Adder ADDER4, having $p = 96$ fault locations, the number of simulation runs increases from 24,576 to 1,167,360. Simulation took about 2 hours. Though, just simulating the circuit function $R:=A+B$ of the 8-Bit ALU ALU8 requires $N_0 = 7,723,745,280$ runs for double-fault injection, which would have taken 8 months with the equipment available to the author. The ALU8 circuit and the Multiplier MUL4 are therefore not considered in the double-fault investigation.

Four error distributions are shown in the following. These are the distributions of both the adders and of the two ALUs performing the function ADD. The other distribution figures are shown on page 188 in the appendix.

The figures show, that double-faults cause more errors to appear at the output, both in quantity and in value. All individual errors from the set E_E can be found in the distributions, however to the exception of the circuits ADDER4 and CLA-ADDER4, where the particular error values $e = \pm 11$ and $e = \pm 13$ still have the likelihood zero. For these two circuits even double-faults never raise these errors. For all other circuits, the error values $e = \{\pm 11, \pm 13\}$ appear to have the least likelihood of occurrence among all errors. This finding indicates that some arithmetic errors seem keeping to occur with a low likelihood among all circuits.

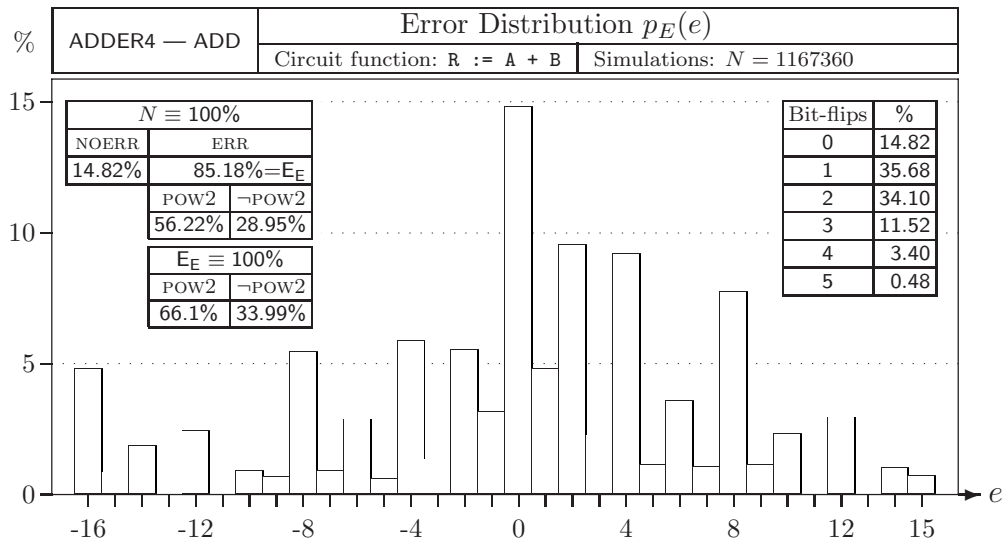


Figure 5.22: Double-fault error distribution 4-bit Adder

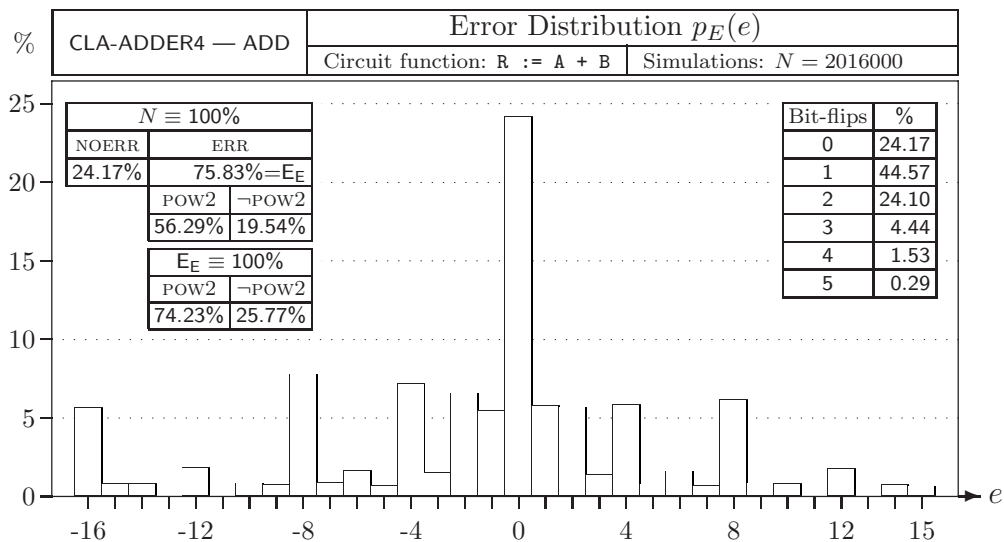


Figure 5.23: Double-fault error distribution 4-bit Carry-Look-Ahead Adder

The similar, but the other way round, still applies to the power-of-two errors. Regardless of the circuit and operation mode, their likelihood of occurrence ranks between 66.1% (ADDER4) and 86.49% (BS4). The power-of-two errors are not as predominant as they were in the single-fault investigations, but they are still dominant. Roughly, their relative proportion among all arith-

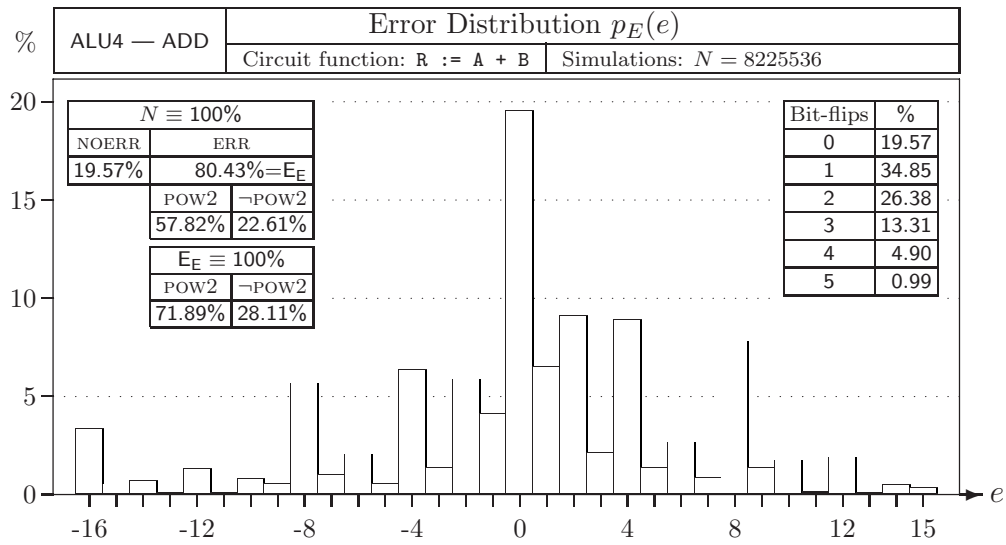


Figure 5.24: Double-fault error distribution ALU4 ($R := A + B$)

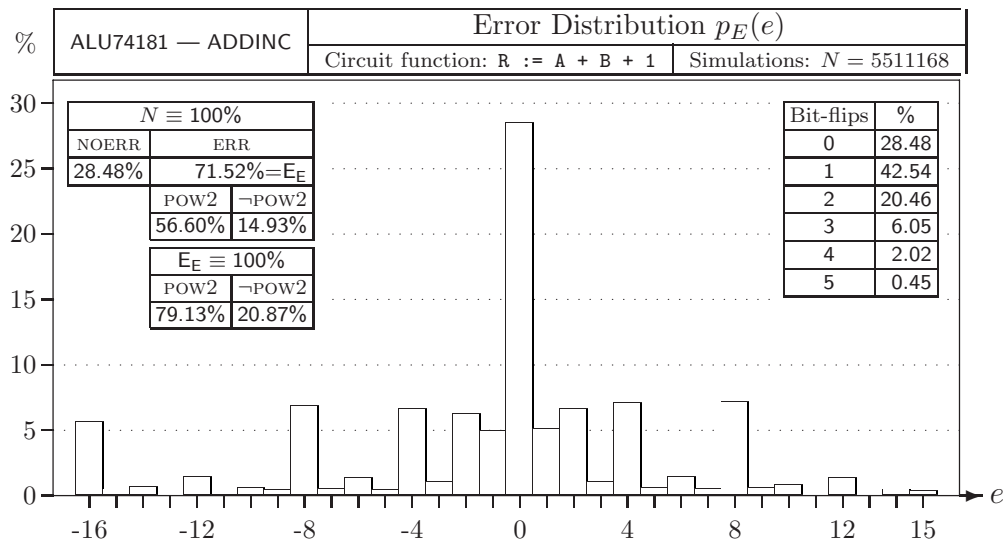


Figure 5.25: Double-fault error distribution ALU74181 ($R := A + B + 1$)

metric errors is 76% on the average. For the investigated circuits the following statement can be made.

If a circuit is assumed to be affected by double-faults, the *power-of-two* errors are still the primary choice in fault-injection experiments. They are to occur with a likelihood of roughly 76%.

As may be seen from the figures, the power-of-two error patterns from the single-fault error distributions of a given circuit and operation mode are

repeating themselves in the corresponding double-fault error distribution. The shapes are almost identical, that is, the particular power-of-two errors seem to keep their relative proportions among each other. There are only slight variations. With negligible loss of accuracy the single-fault error distribution is forecasting the appearance of the power-of-two errors in the double-fault case. A double-fault circuit simulation may be dropped if concentration is on power-of-two errors.

In a double-fault affected circuit, when focusing on the power-of-two errors only, it is sufficient to use the single-fault error distribution as a guideline for double-fault-based error-injection.

5.4.8 Summary

In total 288,836,352 simulation runs were carried out on selected combinational circuits. [Table 5.5](#) on page 110 summarizes the major attributes of the distribution functions for both the single-fault model and the double-fault model. The simulation results demonstrate that the circuits respond to the faults with a specific error pattern. These patterns clearly point out the errors that are most to be expected for a given circuit and operation mode. Across all circuits and modes investigated, the power-of-two errors, P_ρ , have the highest likelihood of occurrence among the errors.

If the output of a *single-fault* affected circuit is in error other than $e = 0$, then this error is to a power-of-two by a chance of 98% on the average (lowest value is 93.57%, highest is 100%).

If the output of a *double-fault* affected circuit is in error other than $e = 0$, then this error is to a power-of-two by a chance of roughly 76% (lowest value is 66.10%, highest is 86.49%).

Because these errors are the most realistic errors in case of a faulty circuit, the error set can be reduced from $2^\rho - 1$ to $2\rho - 1$ errors.

The dominance of the power-of-two errors does not depend on the structure of the herein investigated circuits nor on their size. Therefore it is reasonable to assume that other circuits of same functionality but different structure behave in a similar way. This assumption holds as long as the fundamental mode of

Circuit	Mode	Function	Single-fault				Double-fault			
			N	ERR	POW2	Reference	N	ERR	POW2	Reference
ADDER4	ADD	$R := A + B$	24576	74.32%	100.00%	fig. 5.15 pg. 101	1167360	85.18%	66.10%	fig. 5.22 pg. 107
	CLA-ADDER4	$R := A + B$	32256	57.20%	100.00%	fig. 5.16 pg. 103	2016000	75.83%	74.23%	fig. 5.23 pg. 107
ALU4	ADD	$R := A + B$	65024	62.32%	97.21%	fig. 5.17 pg. 103	8225536	80.43%	71.89%	fig. 5.24 pg. 108
	ADDINC	$R := A + B + 1$	65024	62.60%	96.16%	fig. A.1	8225536	80.70%	70.75%	fig. A.34
	SUB	$R := A - B$	65024	66.20%	96.11%	fig. A.2	8225536	82.58%	68.48%	fig. A.35
	SUBDEC	$R := A - B - 1$	65024	65.85%	97.50%	fig. A.3	8225536	82.43%	69.44%	fig. A.36
	INC	$R := A + 1$	3856	61.36%	99.32%	fig. A.4	462720	78.95%	77.24%	fig. A.37
	DEC	$R := A - 1$	3856	60.43%	98.63%	fig. A.5	462720	77.88%	77.42%	fig. A.38
	AND	$R := A \wedge B$	61696	50.10%	95.16%	fig. A.6	7403520	69.65%	77.83%	fig. A.39
	OR	$R := A \vee B$	61696	45.80%	95.97%	fig. A.7	7403520	66.55%	80.78%	fig. A.40
	NOT	$R := \neg A$	3856	49.10%	95.34%	fig. A.8	462720	68.86%	78.84%	fig. A.41
	XOR	$R := A \oplus B$	61696	48.17%	96.93%	fig. A.9	7403520	77.10%	76.36%	fig. A.42
THRU	$R := A$	3856	58.87%	99.30%	fig. A.10	462720	77.10%	76.36%	fig. A.43	
ALU8	ADD	$R := A + B$	31850496	63.69%	97.21%	fig. 5.19 pg. 104	-	-	-	-
	ADDINC	$R := A + B + 1$	31850496	63.59%	96.81%	fig. 5.20 pg. 105	-	-	-	-
	SUB	$R := A - B$	31850496	67.07%	96.74%	fig. A.11	-	-	-	-
	SUBDEC	$R := A - B - 1$	31850496	67.19%	97.05%	fig. A.12	-	-	-	-
	INC	$R := A + 1$	121088	62.57%	99.36%	fig. A.13	-	-	-	-
	DEC	$R := A - 1$	121088	60.65%	98.10%	fig. A.14	-	-	-	-
	AND	$R := A \wedge B$	30998528	48.57%	93.57%	fig. A.15	-	-	-	-
OR	$R := A \vee B$	30998528	44.73%	94.59%	fig. A.16	-	-	-	-	
NOT	$R := \neg A$	121088	47.98%	95.08%	fig. A.17	-	-	-	-	
THRU	$R := A$	121088	61.30%	99.35%	fig. A.18	-	-	-	-	

Table 5.5: Summary simulation results (Part 1 of 2)

Circuit	Mode	Function	Single-fault				Double-fault			
			N	ERR	POW2	Reference	N	ERR	POW2	Reference
ALU74181	ADD	$R := A + A$	3328	52.70%	96.58%	fig. A.19	344448	74.15%	75.94%	fig. A.44
	ADDINC	$R := A + A + 1$	3328	52.67%	95.44%	fig. A.20	344448	74.04%	75.68%	fig. A.45
	ADD	$R := A + B$	53248	48.21%	98.58%	fig. 5.18 pg. 104	5511168	69.86%	79.02%	fig. A.46
	ADDINC	$R := A + B + 1$	53248	49.98%	99.40%	fig. A.21	5511168	71.52%	79.12%	fig. 5.25 pg. 108
	SUB	$R := A - B$	53248	53.82%	99.11%	fig. A.22	5511168	74.74%	76.97%	fig. A.47
	SUBDEC	$R := A - B - 1$	53248	52.50%	98.69%	fig. A.23	5511168	73.21%	76.97%	fig. A.48
	INC	$R := A + 1$	2896	50.62%	99.45%	fig. A.24	260640	71.17%	81.76%	fig. A.49
	DEC	$R := A - 1$	2896	54.49%	99.75%	fig. A.25	260640	73.55%	80.20%	fig. A.50
	AND	$R := A \wedge B$	46336	49.2%	100.00%	fig. A.26	4170240	68.45%	81.48%	fig. A.51
	NAND	$R := \neg(A \wedge B)$	46336	59.26%	98.60%	fig. A.27	4170240	75.59%	75.05%	fig. A.52
	OR	$R := A \vee B$	46336	55.77%	98.89%	fig. A.28	4170240	72.90%	77.01%	fig. A.53
	NOR	$R := \neg(A \vee B)$	46336	52.76%	98.82%	fig. A.29	4170240	71.41%	79.62%	fig. A.54
	XOR	$R := A \oplus B$	46336	57.46%	98.92%	fig. A.30	4170240	74.31%	77.45%	fig. A.55
	NOT	$R := \neg A$	2896	56.91%	98.54%	fig. A.31	260640	74.23%	77.89%	fig. A.56
THRU	$R := A$	2896	46.89%	98.23%	fig. A.32	260640	67.90%	80.56%	fig. A.57	
BS4	$R := A \ll B$	8192	40.62%	100.00%	fig. A.33	520192	60.60%	86.49%	fig. A.58	
MUL4	$R := A \cdot B$	135168	74.41%	100.00%	fig. 5.21 pg. 105	-	-	-	-	

Table 5.6: Summary simulation results (Part 2 of 2)

operation of the circuits are comparable. The findings are considered portable. If it is to emulate realistic errors that are caused by a random fault affecting a combinational circuit of which the function is known but its structure is not, then it is legitimate to use power-of-two errors.

Owing to the similarities of the error distribution functions, the error distribution of a circuit is likely not to reveal the precise structure of a particular circuit. The manufacturers of microprocessors may thus, without giving away secrets, frankly hand out such error distribution functions to those concerned.

5.5 Creating Realistic Service Errors

Following the previous section, if a microprocessor service performs a transformation function $R := g(A,B)$ and the transformation g uses a fault-affected circuit similar to those investigated, then the transformation function turns into

$$g' = g(A,B) + p_E(e),$$

where $p_E(e)$ is the circuit-specific distribution of arithmetic errors in R , assuming that all faults occur with the same likelihood. The question how to manipulate a transformation function in order create a most likely (and thus representative) error in the output of a service can now be answered more precisely.

- If the error distribution $p_E(e)$ is known, the answer is at hand instantly. The power-of-two errors are the most realistic ones, and the error distribution shows which among them have highest precedence.
- If $p_E(e)$ of the particular circuit is not known, but error distributions of similar circuits are available, then these are a legitimate guideline.
- If nothing is known, choosing arbitrary power-of-two errors is justified.

Example 5.2: The service MUL A,B is assumed to be active (Figure 5.26). The multiplication circuitry in the microprocessor shall be affected by a random fault. Using the arithmetic error distribution from Figure 5.21 (page 105), the transformation function of the input operands A and B turns from $A := A * B$ into

$A := (A*B) + 16$ with highest probability. Other transformation tasks within the scope of the MUL service are not affected as the multiplication circuit likely is not involved in other operations, such as incrementing the program counter or decoding and instantiating the next service. The service error results in a data fault in the storage space at location A. This is a typical processing error of a service.

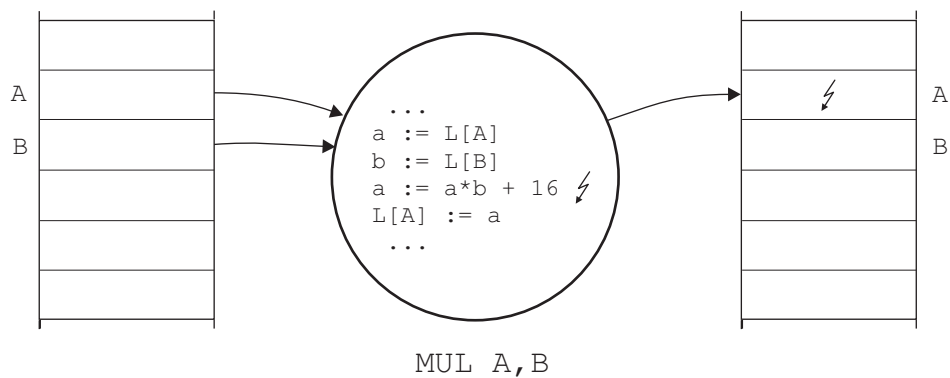


Figure 5.26: Service error on MUL A,B

Example 5.3: The adder circuit located in the address unit shall be affected by a random fault while the service PUSH A (push register A onto the stack) is active (Figure 5.27). Register A shall be 16-bit. In the fault-free case the transformation of the stack pointer is $SP := SP - 2$. Taking the error distribution from Figure 5.19 as a guideline, the arithmetic errors 2, 4, 8 and 16 are the most likely (null error excluded). Using the error $e = 16$ for instance, the transformation turns into $SP := SP + 14$. In any case will the content of the register A be stored at a wrong memory location. Depending on the contents of the storage locations in the fault-free case, the service error may cause up to 3 data faults in the storage space. One fault occurs in the location SP. One fault occurs in the original stack where the content of register A should have been put originally (missed upload), but only again in case the content is different from that of register A. The third data fault occurs in the storage location where A is being put now, but also only in case the previous content was different from register A. Because the address unit is also used for incrementing the program counter, the PC will be faulty as well (not shown in the figure).

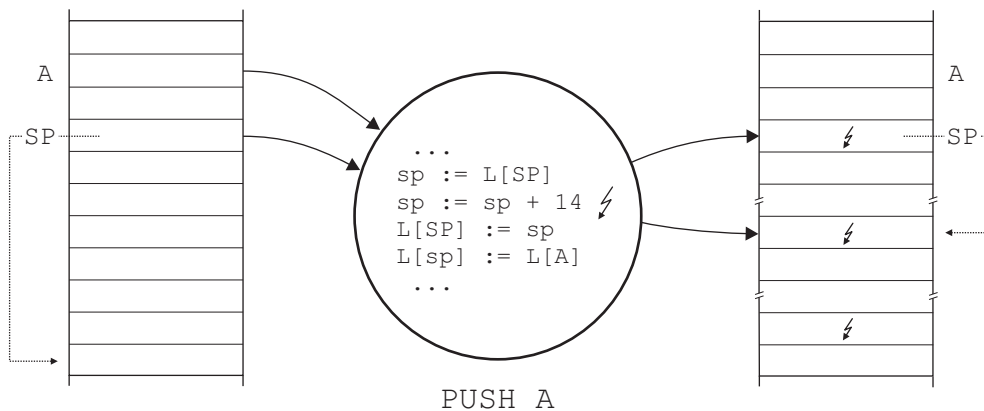


Figure 5.27: Service error on PUSH A

DISCUSSION

The findings from Section 5.4 certainly can be applied onto services to more extent than done in the examples shown. With slight exceptions, all services likely make use of circuits similar to those investigated. With the knowledge of the power-of-two errors a manifold collection of realistic service errors can be created if the specification of a service is known. The service errors obtained are more realistic than are the unbound injections of single bit-flips into memory locations, as done in many SWIFI approaches. This is because a service (its specification) tightens the coupling between real random faults and the creation of data faults in the storage space.

Without doubt, there are service errors that cannot be modeled using the power-of-two errors. Although not verified, examples may be faults affecting the pipeline or the decoding unit. Nothing can be said here about how the corresponding service errors will look like in particular.

5.6 Urging Manufacturers

The manufacturers of microprocessor certainly can say little about the likelihood of occurrence of random hardware faults in their products. However, the manufacturers principally can make statements about the effects of random faults, at least when the product is well-understood. Rather than feeling compelled of handing out any low-level models of their microcontrollers (which often are ranked secret), manufacturers should supply their products with an

appropriate error behavior model. Arithmetic error distributions are a good start, concrete service errors are a better continuation. It is not the complete microcontroller to be error-modeled, but merely the processing hardware. In any case, it is the author's opinion that the manufacturers should (be able to) explain their products comprehensively, both for the good-case and for the fault case.

5.7 Summary

This chapter was dedicated to service errors which form the injectable representatives of random faults in the processing hardware at the process level.

Publications concerned with fault mapping which is the process of achieving a set of representative errors at a higher abstraction level for the purpose of fault-injection, were discussed first. The publications substantiate the demand for a high-level error behavior portrayal onto a microprocessor. However, the approaches taken have not reached the service level, but are encouraging in that the used injection-techniques and available hardware models principally allow for obtaining concrete service errors. The work of [Yo93, Yo96] moreover indicates that even with a "naïve" service specification a high percentage of low-level faults can be error-modeled.

In the error behavior section, some error classes were discussed, and timing errors as well as the multiple affection of services was broached.

In the major part of this chapter investigations on the error behavior of combinational circuits in the presence of single and double faults, carried out by the author through gate level fault-simulation, were presented. The circuits investigated are customary circuits used in functional units of the processing hardware. Instead of using bit-flips for describing the error behavior of a circuit, the *arithmetic error* was introduced. The arithmetic error was demonstrated to be more descriptive than the traditional bit-flip. Almost 300 million fault-simulations were carried out. It appeared from the obtained error distributions that the *power-of-two* errors have the highest likelihood of occurrence ($\approx 98\%$ with single faults, $\approx 76\%$ with double faults). The dominance of these errors was found independent of the structure and the size of the circuits. It is therefore concluded that other circuits of same functionality but different structure behave likewise. When it is to create representative service errors, assuming that the service makes use of a similar circuit affected by a random

fault, the power-of-two errors are the preferable choice for manipulating the corresponding transformation functions of a service.

The findings from the circuit investigations were then used for the exemplary creation of representative service errors. These service errors are more realistic than are ad-hoc injections of bit-flips into memory locations. However, as outlined, the power-of-two errors are not capable of modeling all service errors possible. The chapter closed with a request to the microprocessors manufacturers to provide error distribution functions or service errors of their products to those concerned.

This chapter ends the search for an appropriate fault input domain. In the next chapter, following the requirements identified in Chapter 2, the sets involved in the fault-injection method are instantiated, resulting in the method of mutant-injection.

Chapter 6

Mutant-Injection

6.1 Introduction

In Chapter 2 the requirements for enabling comparability in evaluating the fault-tolerance of software through fault-injection were identified. With respect to random faults in the processing hardware the set of faults F was desired to directly affect the structural elements of the target. In Chapter 3 the target was specified as the controlling process, and the subject of the investigation was stated more precisely as the evaluation of the interior hardware-fault fault-tolerance of the controlling process. In Chapter 4 the structural elements of the process were defined as services. Examples for the creation of realistic service errors, however limited to faults in customary combinational circuits, were given in Chapter 5.

This chapter presents *mutant-injection* which is a fault-injection method for the evaluation of the interior hardware-fault fault-tolerance of safety-critical embedded software in execution, and which forms a fundament for more conformance in fault-injection in order to achieve – as far as possible – comparable measures. The method is characterized by a collection of sets — according to the requirements from Chapter 2. In preview of the chapter, the method is summarized as follows (see also [Figure 6.1](#)).

The fault set F consists of mutants which are deliberate service errors. Depending on the hardware fault assumptions, one or more mutants form a fault-scenario. The fault scenarios are then applied to the controlling process. The readouts R are filtered through the predicates P , and the resulting data describes the error scenarios

that the controlling process has gone through. The set of error scenarios is then valuated and revised. The remaining error scenarios are the basis for the derivation of the fault-tolerance measures.

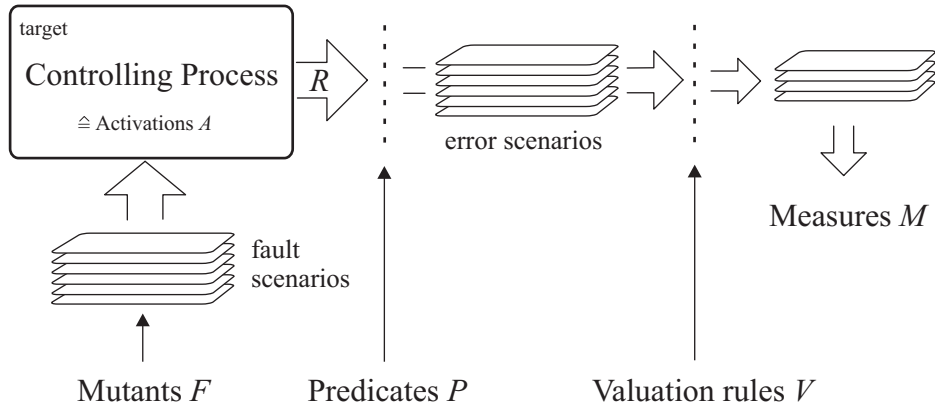


Figure 6.1: Mutant-injection for evaluating the controlling process

The sets involved in mutant-injection are addressed first. Starting with the fault set F , some notions N are defined and proposals for the predicates P and the valuation rules V are given. General issues, such as the golden run or time-censored data, are discussed then. In the section on the measures M the coverage proportion is selected as a measure of the fault-tolerance. After presenting the concept of a simulation-based fault-injection environment, the chapter is summarized and the fault-injection method is discussed. Remaining problems are outlined.

6.2 Fault Set F

For enabling comparability in fault-injection experiments, the used fault set F should consist of a type of faults that is transferable to other systems or objects as well. Also the abstraction level of the faults should be close to the abstraction level of the investigated object, such that the injected faults do not blur or vanish during their propagation along the path of impact. Preferably the injected faults directly affect the structural elements of the target.

6.2.1 Services

For evaluating the interior hardware-fault fault-tolerance capabilities of the controlling process through fault-injection, services are the obvious starting point for such a fault set. First, most microprocessors share the same services (e.g. addition, subtraction, multiplication, bit-operations, logical operations, data movement). Second, services have the same (or at least a similar) meaning among different software. The meaning of a service is independent of the microprocessor architecture (e.g. Harvard, von Neumann) and of the data formats used (little endian, big endian). It is also independent of the coding format of the binary machine instructions and of the location of the binary program P_M within the storage space. The meaning of services to software is invariant of the particular microprocessor, certainly with the exceptions of some very specific services. The third reason pleading for services is more fundamental. Random faults in the processing hardware, if effective, cause service errors. By means of service errors the random faults propagate into the controlling process. Services therefore are the natural inlet into the object of investigation. Service errors directly affect the structural elements that the controlling process is made of. Finally, service errors cause process faults (actually they *are* process faults), such that the fault–error–failure relationship at process level can be maintained and is also in accordance with [Ar90] in that the input in fault-injection experiments is ought to be called faults. Service errors therefore are the faults the interior hardware-fault fault-tolerance finally has to cope with.

6.2.2 Mutants

The deliberate corruption of memory or registers is also called *state mutation*. Similarly, the deliberate corruption of services for the purpose of fault-injection is called *service mutation* here. State mutation is the manipulation of states, service mutation is the manipulation of motion.

Definition 6.1: *Service mutation* is the deliberate cause of a service error for the purpose of fault-injection. Service mutation is the manipulation of motion.

The result of mutating a service is called a mutant. A *mutant* is a service that somehow differs in its transformation activities from the specification.

Definition 6.2: A *mutant* is an intentional service error, representing the effects of random faults in the processing hardware at the service level for the purpose of fault-injection at the process level.

Mutants form the desired fault set F in fault-injection experiments regarding the interior hardware-fault fault-tolerance of the controlling process.

Instead of evaluating the fault-tolerance of the controlling process with respect to the random faults in the processing hardware, the process is evaluated with respect to mutants.

Mutants form an object-appropriate and therefore comparable input domain in the fault-injection experiments. By using mutants, it is now possible to investigate different software with respect to equivalent or even identical fault input. For example, different software may be investigated for their response to a particular mutation of the ADD service (e.g. sum is always wrong by +2). The space of possible mutations that a service principally may undergo doubtless is vast, but mutants at least enable to collect the effects of low level faults at a common abstraction level. Moreover, following the philosophy from [Vo98 p.25], not to worry about how realistic certain anomalies may be, but to simply observe how those anomalies impact the software, mutants allow for the creation of artificial dependability benchmarks which any software can be tested against.

6.3 Notions N

As emerged from Chapter 2, one problem in comparing the results of the fault-injection experiments lies in the sometimes differing interpretation of common notions. In this section the fundamental notions used in mutant-injection are discussed and defined. In particular it will be distinguished between *true* fault-tolerance, masking and blanking.

6.3.1 Fault, Error, Failure

The role of the faults in mutant-injection is taken over by the service mutants.

Definition 6.3: A *fault* is a particular mutation of a particular service. The mutant always causes a process fault.

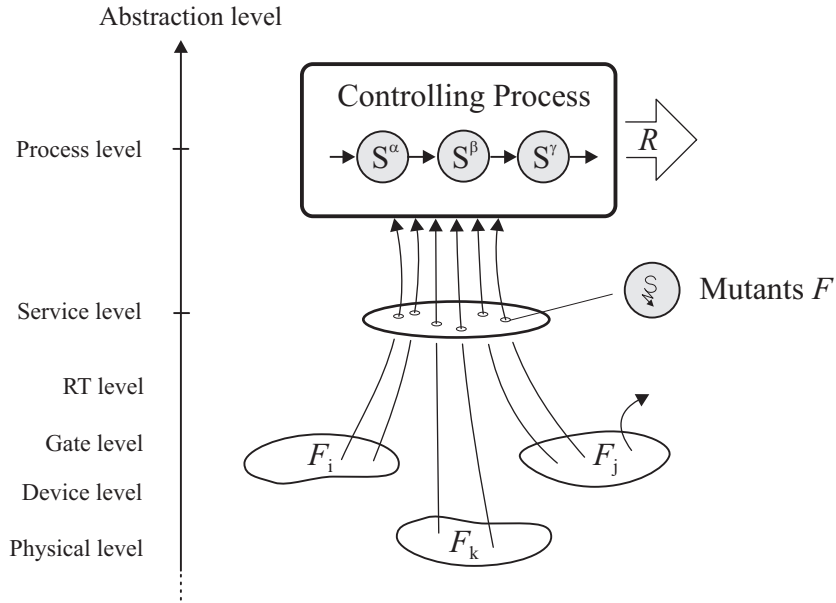


Figure 6.2: The fault set F (Mutants)

If effective, a mutant causes one or more data faults in the storage space. Following the customary practice in fault-injection experiments, the data faults are considered the errors induced. An error then denotes the fact that the content of a storage location is somehow differing from its fault-free value, that is, each bit is considered significant. A comment on this opinion follows in Section 6.6.4.

Definition 6.4: An *error* is a data fault in a storage location.

An error in the storage space is not to be confused with a process error. The latter is an error in (or of) a process, thus a motion error, while a data fault in the storage space is a state error. State errors in the storage space may cause process errors – if a process exists and if the error is read-accessed – but they are not identical.

A particular error is denoted by e_i . The collection of errors present in the storage space at some point in time is denoted by E . Owing to error propagation, an error e_i may found an error family e_i^+ which is the collection of the root error e_i and its descendants. If the controlling process is perfectly fault-tolerant, no error will propagate into the output area L^O without being signaled at the time of output latest. Definition 3.6 can be reformulated as follows.

Definition 6.5: The *failure* of the controlling process Z is the inability to tolerate mutants.

6.3.2 Error Detection

Sometimes the moment of error detection is already considered the relieving moment. The term error detection then encompasses error handling as well. Here, the term error detection will be used in its original meaning. According to [Ar90 p.170], where the error detection latency is defined as “the time interval between an error and its first perception by the FT mechanisms”, detection is understood as recognition and does not imply any subsequent error treatment or signaling. A detected error therefore is not yet a safe error, it still has to be dealt with by the controlling process.

Definition 6.6: *Error detection* is the process or moment of recognizing the presence of a fault effect. A detected error is not yet a safe error.

6.3.3 Fault-Tolerance

Fault-tolerance stems from three fundamental mechanisms: from *tolerance* in the narrow sense of the word, from *masking* and from *blanking*. As far as safety is concerned, there may be no need for further detailing. The higher the fault-tolerance of a system, the better for safety. However, two reasons urge a distinction between *true* fault-tolerance and the other two mechanisms. The reasons are quoted after a closer specification of what is meant here by true fault-tolerance, masking and blanking.

6.3.3.1 True Fault-Tolerance

Fault-tolerance in the technical sense is the quality of compensating the effects of faults. Tolerance in the original sense is the quality of tolerating opinions, beliefs, customs, behavior or similar different from one’s own. Tolerance then implies that the differing opinions etc. have been taken notice of beforehand. The term *true fault-tolerance* will be used here when a system or entity is performing fault-tolerance through appropriate reaction on a fault effect. True fault-tolerance implies error detection. The mechanisms of a voter and the mechanisms behind error-correcting codes work this way, for instance. True fault-tolerance is *reactive*.

Definition 6.7: *True* fault-tolerance applies when a system or entity knowingly tolerates the presence of an error. True fault-tolerance implies prior detection.

6.3.3.2 Masking

Masking is related to the propagation of faults (here: data faults in the storage space). A fault is said to be masked if its propagation is gated out. Masking can be the result of a system’s minded act or can happen by chance. In the former case masking bases on (re)cognition. For example, the voter of a TMR system is sometimes attributed to mask the effects of a fault, as stated for instance in [St96 p.131]. Fault masking in that sense refers to the utilization of static redundancy and is a fault triggered action. Within the scope of mutant-injection however, masking refers to the non-cognitive process of gating out fault effects. Masking then is not a function of the fault and thus happens incidentally. An example is the masking of fault effects in gate level fault-simulation. The propagation of a fault effect may be stopped by some gate, as shown on the left in Figure 6.3. Similarly, at the process level the propagation of a data-fault may be stopped by a service.

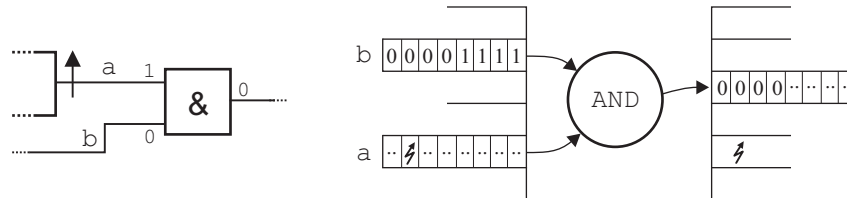


Figure 6.3: Masking analogy

Although the actual process of masking out a fault effect is incidental, masking is nevertheless a quality characteristic of an entity. Increasing the chance of these incidents through appropriate design or implementation is one measure to make an entity more fault-tolerant.

Definition 6.8: *Masking* is the non-cognitive process of gating out fault effects. A masked error is hindered in its propagation.

6.3.3.3 Blanking

A fault effect in the storage space (a data fault) may become overwritten with correct data. Correct data is data that is not directly or indirectly affected by a fault, that is, data that is considered to be correct in the current situation. The data fault can be said to be blanked out. Blanking, as understood here, is different from fault-removal. Fault-removal (actually error-removal) is a deliberate act which follows a previous detection of an error. Blanking, like masking, is considered to happen incidentally. On the left of Figure 6.4 a service is blanking out a data fault. The data fault does not belong to the input operands of the service. On the right of the figure a data fault does belong to the source operands. Owing to the other input operand and to the current operation, the data fault becomes overwritten.

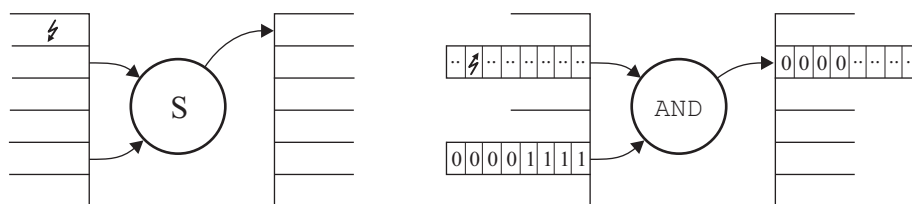


Figure 6.4: Blanking

Another example for blanking is the periodical reset of the microprocessor, followed by a re-initialization of the memory. Although this action may be a precautionary measure and thus is not incidental, the actual clearing of data faults in the storage space still is incidental and non-cognitive.

Definition 6.9: *Blanking* is the non-cognitive process of overwriting an error such that the error is extinguished.

6.3.3.4 Distinction

True fault-tolerance bases on the cognition of fault effects and implies some form of systematic redundancy to be present in the system. It is a design issue which requires additional effort and costs. Masking and blanking are based on non-systematic, coincidental redundancy. It may also be a design issue, but very often masking and blanking are side effects that are achieved without additional effort. Through masking and blanking a system becomes to some extent naturally robust [cf. Ar90 p.169].

A system that gains its fault-tolerance purely from masking and blanking, may, in sloppy wording, be termed *dumb but good-natured* (regarding faults, not regarding its application task). A system that gains its fault-tolerance purely from true fault-tolerance, may, in similar terms, be called *intelligently good-natured*. All systems can be classified somewhere in-between these two extremes. This naturally applies to software as well.

The other reason to distinguish between true fault-tolerance, masking and blanking is because of their different contribution to reliability and availability. A system that draws its fault-tolerance from masking and blanking provides both high reliability and high availability. A system whose fault-tolerance stems from true tolerance provides high reliability, but usually some lower availability. The fault-handling mechanisms of the system may work smoothly (error recognition and immediate nullifying), they may work in an interruptive manner (error recognition and recovery), or even the beseeching way (error recognition, fail safe stop, signaling the operator). The duration of suspension of the primary service certainly depends on the implemented fault-handling strategy, but in general the availability obtained through true fault-tolerance is lower or at the most equal to the availability that is achieved when the same faults would have been masked or blanked. Therefore, a distinction should be present between true fault-tolerance and the other two mechanisms.

6.3.4 Fault Scenario

As mentioned in Chapter 5, a single random fault in the processing hardware may affect multiple services, either because the random fault is permanent or because its effects have latched somewhere in the hardware for a certain period of time. In addition, the processing hardware may be affected by more than one random fault at the same time. The controlling process may thus be facing more than a single mutant during an experiment run. The constellation of which services are assumed to be affected and in which way they are affected is defined as *fault scenario* here.

Definition 6.10: A *fault scenario* is a set of particular mutants provided for a fault-injection experiment run.

A fault scenario may be based on assumptions, it may result from field experience, or it may be derived from a low-level hardware-model investigation

(fault mapping). A scenario may contain just a single mutant (e.g. transient random fault assumption without latching), or may contain several mutants. The result of a fault scenario is an error scenario.

6.3.5 Error Scenario

The effect of a given fault depends on the system activity at the moment of (and following) the sensitization of the fault, so the input space consists not only of the injected fault but also of the subsequent activity. An activity is a trajectory in the system's state space [Cu99 p.708]. The combination of an effective fault (or a series of effective faults) and the resulting activity poses an *error scenario*. For mutant-injection an error scenario can be defined as follows.

Definition 6.11: An *error scenario* is the process starting from, and following, the effectiveness of the first mutant.

Each error scenario, respectively the corresponding data obtained from the experiment run, contains information about which mutants of the applied fault scenario have actually been used by the controlling process (activation, effectiveness) and about the consequences.

6.4 Activations A and Readouts R

Activations are actions required to activate and functionally exercise the target system [Ar90]. The controlling process is guided by the input stream (respectively its history in the storage space) and itself forms the activations. In [Cu99] the term *activations* has been replaced by the more appropriate term *activities*. As noted in Chapter 2, the only comparability requirement that can be made on the set A is to have the controlling process operating in its real environment.

The readouts R form the output domain in fault-injection experiments. They are the responses of the system or entity being investigated. For the controlling process, the readouts are obtained through observing the storage space L , especially the output area L^O . The physical manifestation of the readouts R is identical among different microprocessors and different software applications since they are always in form of bits or combinations of bits in L^O . The contents of the storage locations in L^O and their change over time

are however specific to each application software. More important than the technical readouts are the predicates P .

6.5 Predicates P

Many fault-injection approaches did not use a unified classification scheme of the observed effects of the injected faults (e.g. activation, effectiveness, detection). Arlat et. al. recommend the use of predicates to describe the observations [Ar90 p.170]. This section proposes an exemplary collection of predicates which allow for comparable statements on the properties of the induced errors in the storage space.

Predicates are Boolean assertions that are made on the readouts. A predicate leads to two observable states (*true*, *false*). Some predicates may assert more than once during a fault-injection experiment (e.g. fault activation). For the reason of simplification, it is assumed that a predicate asserts once at maximum. In extension of [Ar90 p.170], it is to remark that some predicates actually are tri-state, incorporating the state *n.a.* (not applicable). For example, the predicate on the effectiveness of a given mutant is *n.a.* until the mutant is first activated.

6.5.1 Fault Activation

The faults in mutant-injection are the injected mutants. The mutants are initialized in the fault-injection environment and are activated at the moment the controlling process is using the mutated services (Figure 6.5). The following predicate is taken over from [Ar90 p.170] and indicates whether a provided fault is activated or not.

`fault_activated(M)`: Indicates whether the mutant M is used by the controlling process.

If the predicate remains *false* the mutant must not enter evaluation since injection did not take place. A time variable t_i (injection) may be associated that holds the point in time at which the predicate becomes *true*. A distribution function of t_i helps in optimizing the injection-process.

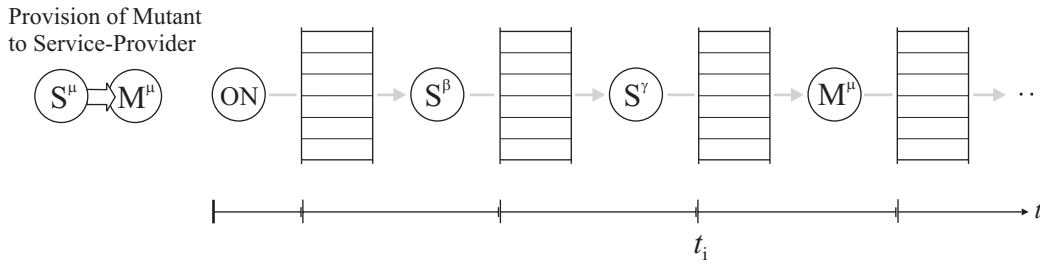


Figure 6.5: Activation of mutant

6.5.2 Fault Effectiveness

As with any fault, an injected mutant may be ineffective in its consequences, that is, the storage space may still be identical to the fault-free case.

Example: The service `CMP A,B` is mutated such that the `N`-bit (the *Negative* bit) is always set, independent of the operands to be compared. If the current source operands are such that the `N`-bit would have been set anyway, the mutant is ineffective.

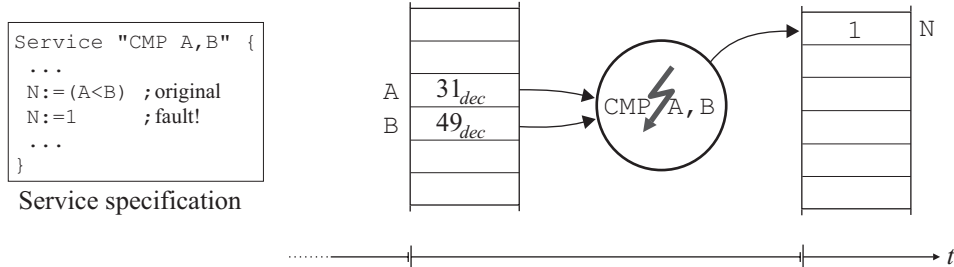


Figure 6.6: Non-effective mutant

If an injected mutant is ineffective then there is nothing left to be detected by the controlling process. The mutant may not enter evaluation of the fault-tolerance capabilities of the controlling process. The following predicate asserts to *true* if the mutant is effective.

`fault_effective(M)`: Indicates whether the injected mutant M is effective, that is, whether an error is induced in the storage space.

The associated time variable is t_e which is $t_i + 1$ when the mutant is effective. As errors in the storage space do not only arise through mutants but also through propagation (proliferation), each error in the storage space can be assigned

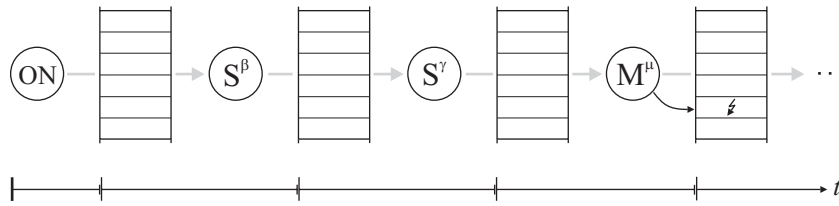


Figure 6.7: Effective mutant

an individual time variable t_o in order to record its time of occurrence. The time interval between the occurrence of an error and its detection is the error detection latency [Ar90 p.170].

6.5.3 Error Activation

An error may lay dormant for a certain period of time. Dormant errors are harmless. They cannot be detected earliest before a service is read-accessing the corresponding storage location. The moment of a read-access onto the error activates the error.

`error_activated(e)`: Asserts to *true* when the error e_i belongs to the input operands of a service.

The time variable t_a denotes the point in time of activation. The time interval between the occurrence of an error t_o and its use by a service (activation) is the error dormancy.

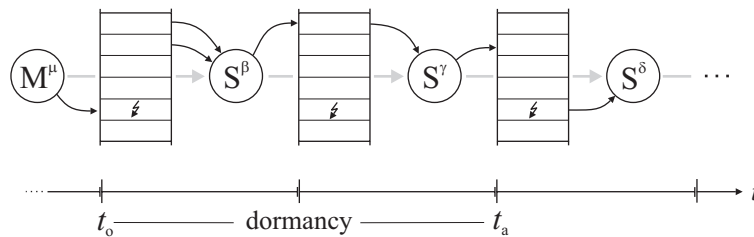


Figure 6.8: Error dormant over two periods

Errors that are never activated do not serve to evaluate the fault-tolerance capabilities of the controlling process (except blanking) since the process never can get notice of their presence.

6.5.4 Error Masking

The following predicate becomes *true* if an error e_i is being read-accessed by a service but does not affect the output.

`error_masked(e_i)`: Indicates whether a particular error e_i is masked by a service.

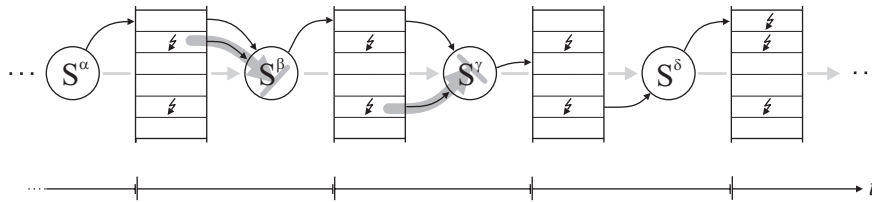


Figure 6.9: Masking

Masking implies the prior activation of an error. Non-activated errors cannot be masked, but they can be blanked.

6.5.5 Error Blanking

An error in the storage space may be overwritten with correct data. If the process of overwriting happens by coincidence, then the error is blanked out.

`error_blanked(e_i)`: Indicates whether a particular error e_i is inadvertently overwritten with correct data.

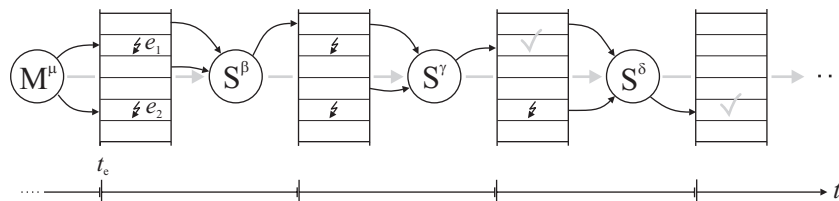


Figure 6.10: Blanking

The blanking of a non-activated error might be termed the *silent death* of an error. An example is error e_1 in Figure 6.10. Silent deaths certainly are the most preferred events.

6.5.6 Error Propagation

Propagation is the locomotion of fault effects. At gate level, a logic error occurring at some node may propagate through the circuit. While doing so, the logic error is conducted along the nodes and passes the logic gates. Owing to the fan-outs the error may multiply from gate to gate. In the same way the errors in the storage space may propagate and proliferate. Propagation implies that the error belongs to the input operands of a service.

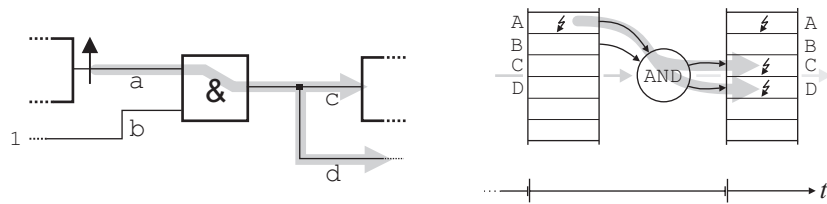


Figure 6.11: Error propagation analogy

The following propagation predicate yields *true* if the error e_i propagates, that is, when it causes at least one descendant in a different storage location.

error_propagate(e_i): Asserts to *true* when the error e_i traverses a service and causes one or more errors in the storage space.

The propagation predicate may also be applied to an error family, in which case the predicate yields *true* when at least one member propagates.

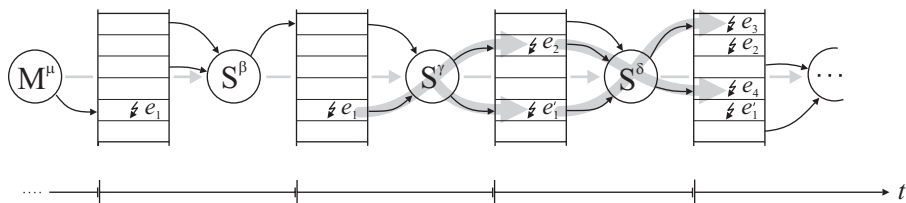


Figure 6.12: Propagation of error e_i

The example shown in [Figure 6.12](#) follows the common view on propagation, in that the errors seem to be transported along the lines of action (here: the read and write accesses of the services). As a matter of fact, an error may also be created in a storage location that does not belong to the output operands of a service. An example is shown in [Figure 6.13](#). The error e_1 is

assumed to cause the service to write into a wrong storage location. Two new errors arise: the error e_2 in the now addressed location which should not have been updated, and the error e_3 in the original location which should have been updated. Here propagation applies as well, although it may be less obvious from the figure. The absence of some activity in the fault case is after all a fault effect. Propagation is related to the causal connection of fault effects and is not restricted to the corporeal movement of errors.

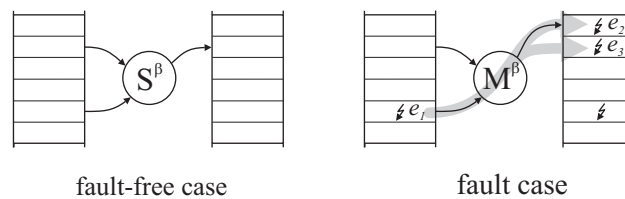


Figure 6.13: Another propagation example

6.5.7 Error Releasing

One predicate has to be assigned to the event of having an error propagated into the output area L^O . At that particular moment the controlling process has lost grip on the error. The error is likely to proceed into the environment, dependent of the type of peripheral devices connected and on their hardware fault-tolerance mechanisms. The following predicate applies when an error occurs in the output area.

error_released(): Asserts to *true* when the output area L^O contains an error.

The predicate is general and relates to the appearance of any error in L^O . As a matter of fact the predicate can also be assigned to a particular error family. The period of time between the occurrence of an error e_i and a descendant propagating into the output area (denoted by t_r) may be called the *transit time* T_t .

If the transit time appears to be zero then a mutant has directly caused a data fault in L^O . For reasons of fairness the injected mutant must be excluded from the evaluation since the controlling process cannot do anything about this situation. Such failure injections are addressed in Section 6.6.1.

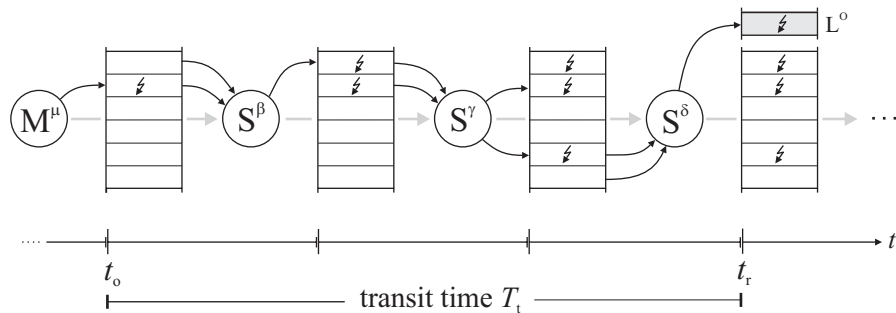


Figure 6.14: Error propagating into output area

6.5.8 Error Detection

At a certain point in time an error may be recognized by the controlling process. As outlined, a detected error is not yet a safe error. However, what is exactly the point in time at which an error can be considered *detected*? One condition is, that something must happen that would not have happened in the absence of the error. The second condition is, that the activities triggered by the error are some kind of error handling procedure.

In Figure 6.15 a simple error recognition analogy is shown. At gate level the error can be said to be recognized if the output of the XNOR-gate is assigned a special meaning among all nodes because some error handling circuitry is

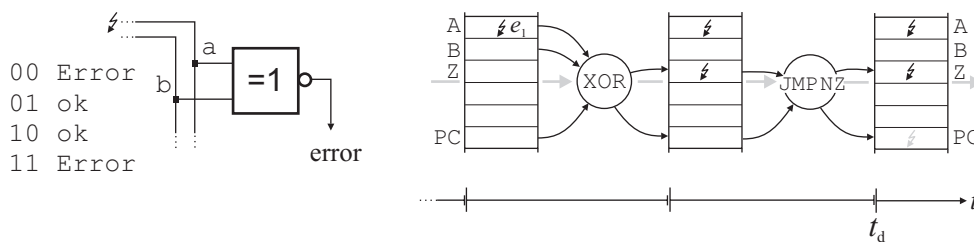


Figure 6.15: Error detection analogy

connected to its output. Similar applies to the controlling process. The XOR service is setting the Zero-bit Z since the contents of A and B are equal (assumption in the fault case), that is, the error propagates into the location Z. This is no detection yet, this is still the normal flow of activities. The next service is checking Z and because Z is found set, the service reroutes the process course to a new code address. If the now addressed sequence of services is an error handling routine, then the error can be said to have been detected at

$t = t_d$. Otherwise the upcoming activities are just fault effects caused by ordinary error propagation. In the figure it is assumed that some error handling is about to start, for that reason the PC contains the error symbol in gray color in the figure. The current value of the PC certainly is a fault effect, but it is supposed to be a good fault effect, that is, the PC is not considered erroneous in the current situation.

Whether or not an error can be said to have been detected by the controlling process depends on the meaning of the error-triggered activities. Usually the sequences of machine instructions for error detection and error handling are known from the program sources, therefore the corresponding code locations in the storage space can be flagged accordingly.

The following predicate indicates whether an error is detected. The associated point in time is t_d .

`error_detected(e_i)`: Asserts to true when the error e_i is detected by the controlling process.

The predicate can of course also be applied to an error family, denoting that a member of the family is detected.

6.5.9 Error Signaling

If the controlling process cannot compensate the effects of the injected mutant(s), it must notify the environment. The only way to report is through the output area L^O of the storage space. Notification can be the presence of some signal or the absence of a specified signal.

`error_signaled()`: Asserts to *true* when the output area L^O contains notification about the presence of problems in the controlling process.

The predicate is the most global and refers to the signaling of the effects of the injected mutant(s). The predicate may also be assigned to a particular error family. Similar to error detection, the notification must trigger some subsequent error (or failure) handling mechanisms (on-chip, on-board, off-board) connected to the output area.

A time variable t_s can be assigned which denotes the moment of occurrence of a notification in the output. The period of time between the occurrence of an error and its signaling is the error signaling latency $T_s = t_s - t_o$.

6.5.10 Fault-Tolerance

The predicate on fault-tolerance evolves from the definition of fault-tolerance and informally reads as follows.

`fault_tolerant()`: No error escapes the controlling process without being signaled by the process at the same time or earlier.

The predicate may be assigned to a mutant M or to an error family. The validity of the predicate starts at the point in time at which a mutant (respectively the first mutant out of multiple mutants) becomes effective. The predicate asserts to true when either no error is released or when $T_s \leq T_t$ (signaling latency \leq transit time). The associated time variable is t_t . However, owing to the fact that the event of “no error released” occurs at the end of the observation interval and thus is artificial, the variable t_t may be time-censored. The problem of time-censored data is addressed in Section 6.7.1.

6.6 Valuation Rules V

After completion of the fault-injection experiments, and after filtering the required information from the readouts R by means of the predicates P , a collection of error scenarios is obtained. Each error scenario describes the living of the errors in the storage space and the corresponding behavior of the controlling process. Before the error scenarios enter the derivation of the final fault-tolerance measures, the scenarios must be valuated and, if necessary, be purged. What is to bear in mind when revising the error scenarios is discussed in this section.

6.6.1 Period of Grace

Injecting a fault directly into the output of the investigated object, and immediately stating the failure of the object in tolerating the injected fault, clearly is inappropriate. Such injections, where the system or object of investigation à priori is given no chance to tolerate the fault, are failure injections. Failure injections are adverse to fairness. The questions thus arises how ‘far’ from the output may faults be injected into an object without being principally unfair. The answer depends on the intelligence of the structural elements and on the complexity of the signals or data being processed by these elements. For a fault-tolerant gate level circuit the distance between the injection location and

the primary output likely should not be closer than one or two levels (the number of gates to pass) — from an intuitive guess.

Similar, safety-critical embedded software must be granted a *period of grace* after injection in order to give the controlling process a principal chance to cope with the problem. The period of grace can be given in number of services. A rough guess of an appropriate period is 2 – 4 services. Expecting fault-tolerance, at least true fault-tolerance, within this period is doubtful. For the reason of fairness the period of grace has to be considered when valuating the experiment results.

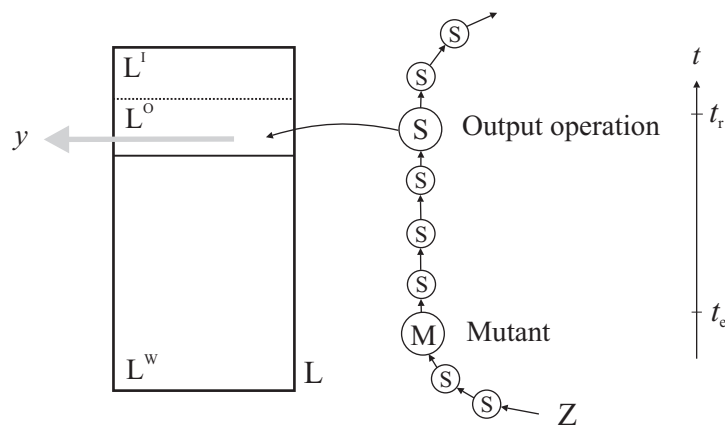


Figure 6.16: Period of grace

6.6.2 Distinct Error Scenarios

It is common sense in fault-tolerance evaluation through fault-injection that the object of investigation should not only be exposed to a reasonable high number of faults (quantity), but to as many *different* faults as possible (quality). This is especially true when the obtained measures are used for predicting future behavior. Therefore as many different fault scenarios as possible should be applied to the controlling process. Although this is true and desirable, it is not yet the finalized request. Different fault scenarios may result in identical or equivalent error scenarios, that is, different fault scenarios may pose the same ‘fault’ (in a generalized sense) to the controlling process. Therefore the request for different faults (or fault scenarios) more accurately reads as the request for obtaining distinct error scenarios.

Next to the hope that a reasonable number of distinct error scenarios covers future scenarios in the field, another reason for requiring distinct scenarios is to avoid redundant experiments.

6.6.3 Redundant Experiments

Identical or equivalent error scenarios result in redundant fault-injection experiments. A redundant experiment reveals no new information. Redundant experiments, if not recognized as such, may lead to incorrect measures. The measures are not incorrect in the mathematical sense, but in their expressiveness. They are either too optimistic or too pessimistic and thus are not reflecting the truth. If n experiments have been carried out and if k experiments used to be redundant then the statement on the considered predicate(s), such as the predicate on fault-tolerance, must be related to $n - k$ experiments, otherwise the measures are bogus.

Redundant fault-injection experiments therefore are to be noted and excluded from the final derivation of the measures. This is decisive for maintaining trustworthiness of the evaluation procedure and the final results. As can be imagined, through intentionally incorporating the results of redundant experiments, the final measures can be manipulated at will (e.g. Example 6.1 in the next section).

6.6.4 Relevance of Errors

Safety-critical embedded software is not a test program aiming at detecting low-level faults. Safety-critical embedded software, strictly spoken, also is not interested in *bit-flips* (data faults), but should be interested in *errors*. An error, besides being the consequence of a fault, is associated with a change of meaning of the affected data (information view). Often, what is finally important is the information carried by the data, not the technical representation.

Data faults that cause no information change with respect to the fault-free case are termed benign errors. Benign errors are errors whose presence makes no difference to the software (cf. Ka95 p.255]). Benign errors cannot be detected.

Example 6.1: The Boolean variables `true` and `false` are often implemented the following way: `true` is represented by any arbitrary value greater 0 (in

the unsigned representation), and `false` is represented by 0. If, for example, a memory location contains `true` (11111111) in the fault-free case, then none of all possible bit-combinations in the fault case, except the combination 00000000, poses an error to the process. The memory location will still be evaluated by the controlling process as `true`. To the process there is no change of meaning and thus no difference to the fault-free case. Of the 255 bit-manipulations possible, 254 will be resulting in benign errors.

Benign errors may lead to redundant experiments and finally to incorrect measures. Similar applies to data faults resulting in identical errors (fault equivalence). Errors are identical if they have the same meaning, that is, if they hold the same information. Identical errors also can result in redundant experiments.

Example 6.2: The controlling process is reading the temperature from some sensor in order to check whether the temperature is above a certain threshold (Figure 6.17). The temperature value which is coded 8-bits in this example, is thus used for a binary decision. From the sight of the controlling process the environment has only *two* relevant states regarding the temperature: either below (or equal to) the threshold, or above. Consequently, when it is to simulate a defect of the sensor through state mutating the storage location in L^I (e.g. input testing), there is only *one* error, namely that the mutated data reflects the converse of the real state of the environment. All remaining bit-manipulations of the storage location are either equivalent to the good-case (benign errors) or to the fault case.

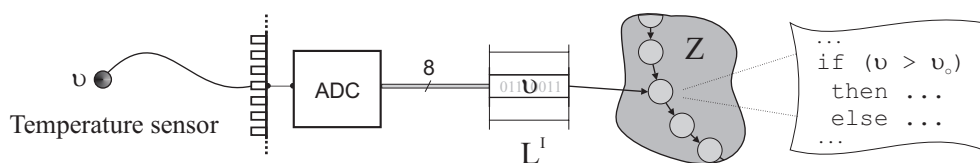


Figure 6.17: Two relevant temperature states

For the sake of trustworthiness, the relevance of the errors must be taken into account in the experiments.

An error can be relevant or irrelevant to both the controlling process, and – only when propagated into L^O – to the environment. The judgment whether

or not an error in L^O is considered relevant to the environment is, as a matter of fact, beyond the scope of the controlling process. It is one of the major subjects of software design to make these errors relevant to the process too. The fault-tolerance capabilities can only be judged with respect to the meaning that the errors have to the controlling process.

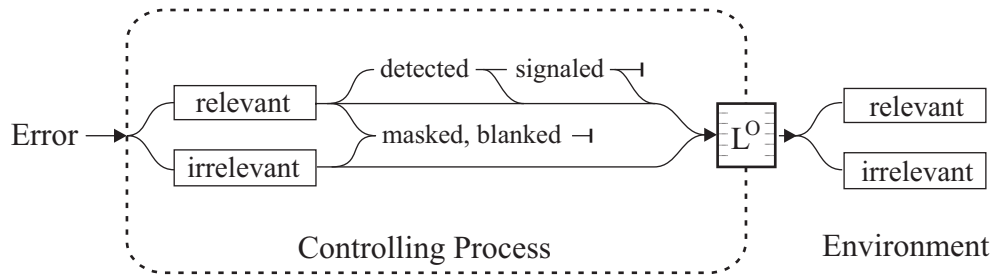


Figure 6.18: Error relevances

Errors that are irrelevant to the controlling process (benign errors) do not serve to test error detection mechanisms for obvious reasons. These errors however, if propagated into L^O and if then ranked relevant to the environment, can serve to identify the *absence* of fault-tolerance. Errors that are relevant to the controlling process are principally detectable. An error is relevant to the controlling process if the data is evaluated by the process, that is, if the information is used for some decision (e.g. conditional branches, data-indexed calls or storage accesses) *and* the consequence is different from the fault-free case (cf. examples). A relevant error thus has some form of control character and influences the process. A relevant error may cause a process error, that is, the services may be correctly operating on incorrect data, however resulting in an operation that is deviating from the requirements.

REMARK

Instead of speaking about the relevance of errors, one may from the very beginning distinguish by dint of the predicates between *data faults* and *errors*. A data fault then is just a fault effect and has no meaning to the controlling process (irrelevant error). At the moment at which the data fault however becomes relevant to the controlling process, it turns into an error. Definition 6.4 then is to be adjusted, and some predicates dedicated to data faults are to be added to the proposed predicates. For the reason of compatibility with the

existing SWIFI approaches, as well as for space reasons, this separation was not made.

6.7 General Issues

6.7.1 Observation interval

Each fault-injection experiment is assigned a certain observation interval $[0, T]$. During that interval a given predicate P may assert at time t_p or may remain unasserted. For obvious reasons the interval cannot be $[0, \infty]$. Some

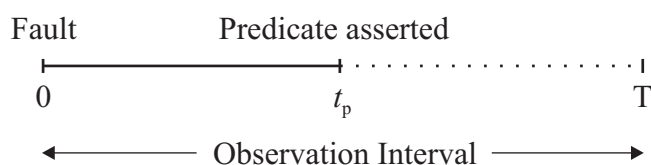


Figure 6.19: Characterization of an experiment [Ar90]

measurements may therefore result in time-censored data. The predicate on fault-tolerance is a predicate that may be bound to the artificial event of a *time-out*. A time-out occurs when no error is released and no signal is given and there are still errors present in the storage space. The controlling process can be attributed to have been fault-tolerant so far, but there is no natural point in time bound to this fault-tolerance. Therefore it might be necessary to construct two auxiliary predicates.

P_0 : asserts when all errors in the storage space are removed or blanked, or when a signal is given in time.

P_T : asserts at time-out, that is, there are errors remaining, but none has been signaled nor has propagated into L^O .

The predicate P_0 defines a natural point in time which is specific to the controlling process. In case of P_T asserting, although the process has been fault-tolerant so far, problems still could arise for $t > T$.

6.7.2 Error Proliferation

While error propagation focuses on the fact that an error causes descendants at other storage locations, error proliferation focuses on the amount of subsequent

errors. A proliferation factor may be assigned to an error family, giving the multiplication rate per propagation step. The rate may either indicate the number of new errors caused, or the total number of descendants present after propagation. For the latter case the proliferation factor would be defined as

$$f_p(e_i^+, t) = \frac{n_f(e_i^+, t + 1)}{n_f(e_i^+, t)}, \quad (6.1)$$

where $n_f(e_i^+, t)$ is the number of family members at time t . A very high proliferation factor over a short period of time is identifying the event of an *error explosion*. There is no proliferation without propagation. Diminution is due to fault-removal or blanking, in which cases $f_p < 1$.

The total number of errors $n_f(e_i^+, t)$ can be sketched over time, as done in [Figure 6.20](#). During the time interval t_1 none of the errors is propagating. Since the number of errors remains constant, the errors are dormant or are masked. In the time interval t_2 the number of errors is decreasing. Since propagation is *false*, the only reason for the diminution is blanking or removal. During the interval t_3 as many errors are produced as are blanked or removed. Also shown in the figure is the point in time at which detection is taking place.

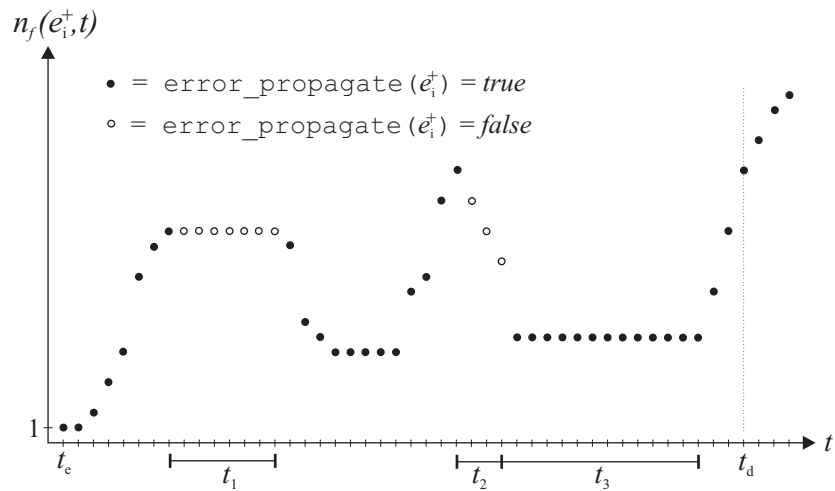


Figure 6.20: Error proliferation

The diagram gives valuable insight into the relationship between the controlling process and the living of the errors in the storage space. The grade of infection at any time is visible, and the reactions of the process may be judged also

with respect to the number of errors present at a particular moment. For example, it might be interesting to know, how many errors – on the average or in particular – are present in the storage space at the moment of detection or signaling. The answer gives an indication on how many errors are needed to trigger the detection mechanisms or to cause signaling.

6.7.3 Golden Run

In the fault-injection experiments discussed in Chapter 2, the golden runs were carried out separately from the injection process. This conception requires reproducibility of the input, which was no problem in the reported experiments since the input used to be deterministic (e.g. matrix multiplication). Safety-critical embedded software however faces a rather non-foreseeable and complex input that may not be perfectly reproducible (e.g. engine temperature, air pressure, oxygen content).

PARALLEL FAULT-SIMULATION

A solution could be to perform the golden run in parallel to the injection run (parallel fault-simulation). Both runs are separate but synchronous processes. During each time slot, the service-provider (a simulator) is executing two machine instructions concurrently. One instruction belongs to the injection run (the fault-injected process) and the other belongs to the golden run (the fault-free process). The storage space must be available twice.

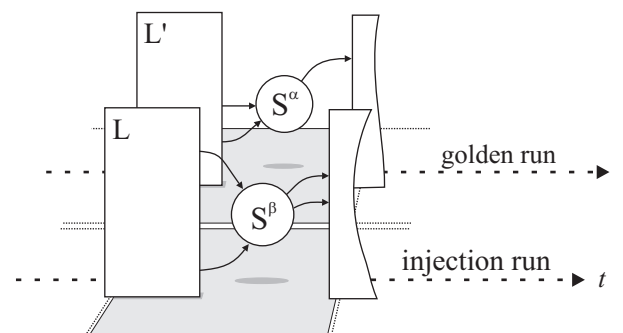


Figure 6.21: Parallel fault-simulation

Both processes are starting with identical input from the environment. The introduction of an error or a deviation in program flow can be recognized

immediately. However there is one major drawback. Since two independent processes are operating in parallel, both of them need to interact with individual environments. There is only one environment which of course must be dedicated to the fault-injected process. The golden process therefore cannot control anything and may fall down after a period of time because of the lack of appropriate responses from the environment. Therefore, executing the golden run in parallel makes sense only when the fault-injected process is reacting upon the mutants much quicker than does the environment on the generated control-signals (quasi static environment).

A MUST?

The golden run is a guideline of how the controlling process behaves in the general absence of faults. In the presence of faults, the controlling process likely takes other routes through the program (error detection and handling), that is, the golden run and the fault-injected run become diverse qua program paths and corresponding activities. In consequence, the storage space L' no longer represents the time-synchronous fault-free case of the affected storage space L . This is a general issue often missing in the publications where the golden run is considered the only basis for comparison. As soon as the controlling process has reacted upon the injected mutant, the situation is a new one: from this very moment it is not the task of the controlling process Z to realign with the golden run, but to compensate the fault effects its own way.

The golden run is not the ultimate criterion and therefore is not compulsory. Nevertheless it must be possible to differentiate between fault-free data and affected data. For that, each storage location can be assigned two contents: one for the fault-free data and one for the actual data (whether erroneous or not). The services are transforming both contents, that is, a service concurrently processes the actual data value and the unaffected value.

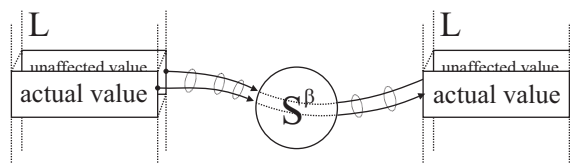


Figure 6.22: Concurrent processing

Whether the actual value (the foreground data) is erroneous or not can be

determined from comparison with the unaffected data (the background data). It is not necessary to perform the comparison each time. If the input of a service is known to be erroneous, often the output is erroneous as well (e.g. bitwise complement, negation, bitwise rotation).

6.7.4 Divide and Conquer

Similar to testing sub-circuits of a circuit, the injection-experiments can focus just on particular parts of the program. Preferably these program parts perform a dedicated task, such that their input, actions, and output are known and comprehensive. If the program P_M is created from a high-level language, such as ADA or C, certain subroutines (modules, procedures, functions) may be advisable candidates. At assembly language level, certain program paths may be candidates for an individual investigation. In any case, each part of the program P_M is mapped by the service-provider into a corresponding process which forms a sequence within the overall process Z . Each partial process, denoted here as z_i , can be evaluated the same way as the controlling process. Like in gate level fault-simulation, where the sub-circuit under investigation is assigned pseudo-primary inputs and outputs (PPI, PPO), the process z_i defines its input and output locations. These may be located in L^I and L^O , but usually they are located somewhere in the working area L^W of the storage space (e.g. stack, local variables).

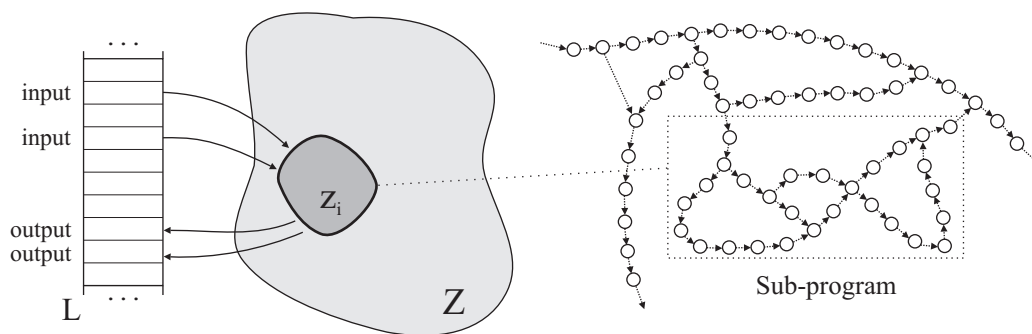


Figure 6.23: Investigation of partial process z_i

The fault-tolerance capabilities of z_i can then be assessed through injecting mutants. The results indicate how and in which way the other parts of the controlling process can rely on the tested component. A process z_i may

also be tested just for its behavior regarding faulty input (local input-testing) or may be subjected to a data-sensitivity analysis (input variation-analysis). An interesting approach for analyzing the propagation paths of data faults in modularized software is given in [Hi01]. The modules are assigned *error permeability* values, which are defined as the conditional probabilities that an error propagates from a module input to a module output. In any case, the results obtained from a partial analysis describe the behavior of a functional component of the controlling process. The ‘functional block’ z_i may then statistically be combined with other blocks in order to derive final measures for the controlling process Z .

6.8 Measures M

A well-known measure used in fault-injection is the *coverage*. A coverage is a statement that is made on the assertion(s) of a certain predicate or a combination of predicates. The coverage *factor* is a probability parameter. It requires the conditional probability of each error scenario – out of all scenarios ever possible – to be known (exhaustive testing). The coverage *proportion* is a frequency parameter. It is based on those error scenarios that are obtained from the experiments. Each error scenario is considered equally likely. Since little can be said about the probability of the ‘real’ faults, the coverage proportion is proposed as the suitable measure. A short recall on the coverage proportion and on the corresponding time intervals is given next.

6.8.1 Coverage Proportion

Following [Cu99], the *coverage proportion* can be defined in terms of the error scenarios. Let G denote the set of error scenarios that enter the derivation of the measures. Let H denote a discrete random variable characterizing the assertion of a particular predicate, such that $H = 1$ if the predicate asserts, otherwise $H = 0$ (or vice versa). If $h(g)$ denotes the value of H for a given error scenario g , then the coverage proportion is defined as

$$c = \frac{1}{n} \sum_{i=1}^n h(g_i), \quad (6.2)$$

and gives the frequency of having a certain predicate asserted with respect to the total number of n error scenarios.

6.8.2 Time Intervals

According to [Ar90 p.169] an experiment can be considered a Bernoulli trial regarding a given predicate P . The associated time variable is t_p . Each t_p can be assigned a random variable T_P . From the experiments a frequency distribution of T_P is then obtained. The distribution can be a density function $c(t)$ or can be a cumulative function $C(t)$ as used in [Ar90]. Note that $C(t=T)$, where T is the observation interval, can be smaller than 1 owing to time-outs.

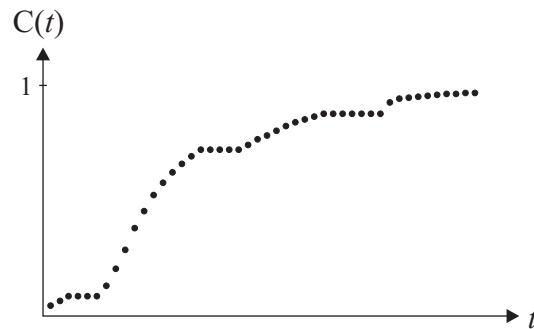


Figure 6.24: Sample distribution (cumulative)

Each function fully characterizes the coverage with respect to the associated predicate. For estimating future behavior, the error scenarios G need to be representative of future scenarios. In [Po95] and [Cu99] the estimation of coverage through statistical processing is addressed in detail.

As different software likely is executed at different clock speeds, the timing measurements preferably should not be taken in terms of the real time passed, but in terms of executed machine instructions (services). This allows for direct comparability of the time interval distributions.

6.8.3 Interior Hardware-Fault Fault-Tolerance

The interior hardware-fault fault-tolerance capabilities of the controlling process can be assigned various attributes and corresponding measures. For example, a measure may be dedicated to the average error proliferation factor (Section 6.7.2) or to the number of errors present when the controlling process is giving signal. The latter is an indication of how important the signaling is to be classified. Another measure may be dedicated to the number of control-flow errors, either in sum or separated into severe and non-severe errors. There

is certainly a variety of attributes and corresponding measures that serve to elucidate the behavior of the controlling process in the presence of mutants, however, in the following only the fault-tolerance coverage and the error detection coverage is discussed.

For the discussion it is assumed that the valuation rules have been applied, that is, the set G contains only distinct error scenarios in which also the period of grace is granted.

6.8.3.1 Fault-Tolerance Coverage

The fault-tolerance coverage, for short FTC, is the coverage proportion regarding the predicate on fault-tolerance (Section 6.5.10). It is a measure of the fault-tolerance in the quantity domain. The corresponding distribution function of the time intervals gives a measure regarding the time domain. Note that the measures may contain time-censored data (Section 6.7.1). The FTC is the most important dependability measure in safety-critical embedded systems.

The fault-tolerance coverage does not give any details about which fault-tolerance mechanisms contributed to mastering the applied fault scenarios. For a more detailed information about the interior hardware-fault fault-tolerance capabilities of the controlling process, it may be further distinguished between signaling, blanking and masking. The corresponding coverages then are *relative* coverage proportions, that is, they are related to those fault scenarios that were successfully tolerated.

RELATIVE SIGNALING COVERAGE: The relative signaling coverage, RSC, gives a measure of how many fault-scenarios were tolerated through giving signal in good time.

RELATIVE BLANKING COVERAGE: If the fault-tolerance was achieved only through blanking, then the corresponding error scenarios contribute to the measure of the relative blanking coverage, RBC. Having the effects of a fault scenario blanked out doubtlessly is the most desired form of fault-tolerance (full availability).

RELATIVE MASKING COVERAGE: Similar, if the effects of a fault-scenario (the errors in the storage space) were masked, but not tolerated some other way, the corresponding error scenarios may enter the measure of the relative masking coverage, RMC. However, although the controlling process has proved to be resistant to the induced errors in the storage

space, this form of fault-tolerance certainly is the least desired since the errors are still present at the end of the observation interval.

6.8.3.2 Error Detection Coverage

Another common coverage proportion is the error detection coverage (EDC). It gives a measure of the detection capabilities of the controlling process. Because a detected error is not yet a tolerated error, the EDC may be greater than the FTC, that is, more errors may have been detected than tolerated in the end. If so, the effectiveness of the true fault-tolerance mechanisms is imperfect.

The EDC may principally be based on all error scenarios G . However, a portion of the fault-scenarios may have produced only irrelevant errors in the storage space. These errors are undetectable on principle. Therefore it is advisable to remove the corresponding error scenarios from G and to relate the EDC to the remaining set of error scenarios. This might be termed the REDC (relevant-error detection coverage). The measure then gives the detection capability with respect to those fault-scenarios that, on principle, can be detected since relevant errors influence the controlling process (e.g. decisions) while irrelevant errors do not. The measure has more expressiveness as the basis is more clear.

6.9 Fault-Injection Environment

6.9.1 Simulation-Based Fault-Injection

The most suitable fault-injection technique to be applied in mutant-injection is the simulation-based approach. Several reasons account for using a simulation-model of the target microprocessor. First and foremost, the service-provider model of a microcontroller can only be simulated since the real hardware hides essential storage locations — both for the controlling process and for the instrumentation. Even if the hardware would not be hiding essential locations to the software, real microcontrollers (SWIFI approach) do not allow a concurrent and transparent all-time access to the storage space from outside. The only workaround would be to use the advanced debugging features provided by some modern microcontrollers, but then the experiment would have to be conducted as a continuous stop-and-go of the controlling process in order to constantly check for the predicates (especially propagation). The execution

speed would be far from real-time, such that the controlling process cannot be connected to its real environment. Also, as mentioned in Chapter 2, only a few microcontrollers used in safety-critical embedded systems do provide these special debugging features. Software for other microcontrollers would be excluded from this assessment. Moreover, a real microprocessor hardly allows for simulating timing errors. A simulation model of the target microcontroller therefore is the most appropriate solution. While a real microcontroller cannot be operated much faster than with its specified clock frequency, and even overclocking would not catch up the loss of time needed for monitoring, a simulation-model in principle can be operated at real-time. It depends on the particular implementation of the simulator (hardware, software) and is a technological issue.

In addition, a simulator can provide special features for the dynamic analysis (the fault-injection experiment) as well as for a static analysis of the binary program P_M . The major requirements on such a simulator, its benefits and its feasibility are discussed in the following.

6.9.2 A Simulator Concept

The evaluation of the interior hardware-fault fault-tolerance of the controlling process is preferably carried out with having the process connected to its destination environment. Therefore the peripheral on-chip devices may not be missing in a simulator. The simulator should consist of two parts which together simulate the real microcontroller in real-time. One part, the service-provider simulator (SPS), implements the service-provider model. The other part, the communication channel simulator (CCS), covers the peripheral on-chip devices and the signal generation.

6.9.2.1 The Communication Channel Simulator

The CCS simulates the communication channels between the I/O area of the storage space and the circuit board of the embedded system. Preferably, as with in-circuit emulators, the CCS physically connects to the microcontroller socket on the board. The CCS generates the necessary signals to the outside and to the service-provider simulator. The main task however is to provide the peripheral devices that are used in the real microcontroller. For reasons of execution speed the CCS should be implemented in hardware. Another task of the CCS is to record the input-stream to the L^1 . The recorded input stream

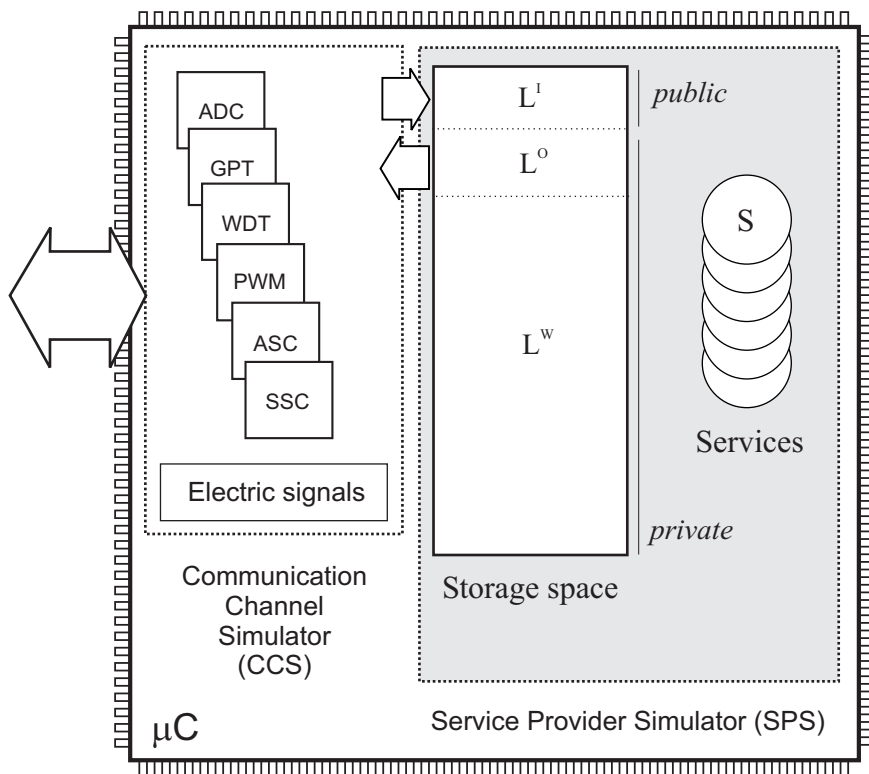


Figure 6.25: Simulator concept

may be used at a later time – completely or partly – to simulate a certain state of the environment. Although the stream will then be static and for the most part non-reacting upon the outputs in L^0 , some injection-experiments may still be reproduced. This is especially true for those experiments in which the injected mutants cause a rapid reaction of the controlling process, such that the real environment appears to be static meanwhile. A more sophisticated version of the CCS could be collecting statistical information on the individual signals from the environment (e.g. distribution functions on signal values), as well on the signals generated by the controlling process. The correlations between the signals can then be used to build a fully-functioning environment-simulator.

6.9.2.2 The Service-Provider Simulator

The SPS provides the individual services of the microcontroller. At the first step, the SPS is nothing else than the well-known and common register model simulators that are shipped with the microcontrollers' software-development

suites. The only difference lies in the broadening of the memory space to the storage space (incorporating formerly unnamed but essential state elements) and in the corresponding extensions of the activities of the services (adding transformations). Because the existing register model simulators are fully functioning models of the real target processor, all essential state elements and activities are already implemented. These just need to be named and shifted to the right locations, that is, into the storage space and into the services.

Principally the SPS may be implemented in software. The microcontrollers used in safety-critical embedded systems usually operate at clock-frequencies of some decades of MHz (the Infineon 80C167 is clocked with 20 MHz, similar applies to others). Modern high-performance computers are operating at more than 100 times these frequencies. The application of parallel computing and the usage of specially designed hardware-boards for the simulator (dual-ported memory for example) can extend the time margin much further. There is no technological problem to overcome for the construction of a real-time SPS.

6.9.3 Requirements

The basic requirements on a fault-injection environment and on the instrumentation tools are known and common. The SPS, like any processor simulator, must of course allow to upload the binary program P_M into the storage space. Preferably, the simulator should also be able to extract from the assembly listing or from a map-file the local names that are given to certain storage locations (variable names). These are assigned to the storage locations in addition to the regular addresses, which eases very much the identification and meaning of certain storage locations when inspecting the controlling process step by step, or when examining the storage space. Similar applies to the locations in the code area. The mutants are either hard-coded manipulations of the services or, certainly favorable, given as configuration file to the SPS.

A storage location in the simulator is not just a plain memory location, but is likely to be a rather complex object in the sense of object-oriented modeling. Each storage location is to be flagged with various attributes. These attributes control the access-methods and the monitoring. For example it needs to be specified whether a storage location is read-only, whether it belongs to the L^O or to the L^I , whether it has a special meaning to the application software (user definable attributes), whether special simulation-events are bound to it

on read or write access (breakpoints for example), whether it belongs to the code section of the program P_M , whether and which predicates are bound to the location, and similar customary attributes. Also the number of accesses onto the storage location may be counted for statistics.

6.9.4 Benefits

Most of the benefits of using a simulator are known from the simulators used in the common software-development suites. Stepping through the process while observing the storage locations of interest as well as identifying the current position in the source code is one acknowledged feature. A built in online-disassembler may not only aid in reengineering a binary program P_M but also elucidates what is about to happen in case the controlling process undergoes a control flow problem and resumes execution in the data area (similar applies when the binary code of the program is changing). Collecting statistics on the program P_M , such as the number of machine instructions contained, the number of instruction classes used, and other information can as well be carried out by the SPS.

6.9.4.1 Automatic Program Path Analysis

Of special interest may be what is called here the automatic program path analysis. No input data is required for this semi-dynamic procedure. In this operation mode the simulator walks along the program paths (beginning at some initial point) and executes all services as if they were NOP-services (no operation), that is, the simulator only advances the PC according to the current service but performs no other transformations (except stack operations on calls and similar). Conditional branches (conditional jumps, calls and returns) are both taken *and* crossed. The simulator just remembers which paths have already been paced and which not (backtracking principle). Through this procedure the simulator can identify most of the legal paths of the program and can mark the corresponding storage locations with a *legal-path*-attribute. This attribute can serve in later mutant-injection experiments to identify control-flow errors immediately.

Although most of the legal program paths can be found through this analysis, the simulator might need some human assistance. Interrupt routines, for example, are usually located off the main code separately. The simulator can try all interrupts but it cannot always decide whether the code found is to be

considered legal or not. When uploading the program P_M into the simulator, the simulator can mark all storage locations that is being written into, and thus can decide which locations are to be considered empty and which not. From there it may be more obvious for the simulator whether certain locations do contain code or not (depending on the address in the memory space, may be data as well). Another problem in the path analysis are implicit conditional branches, such as `jmp[index]`. The conditions lie in the index data, that is, the data decides where to proceed. Without assistance the simulator must check all possible destinations. Generally these instructions should be avoided as they give too much control-power to data. The only exception is when all destinations possible are indeed legal destinations. The simulator may produce warnings on such suspect machine instructions.

6.9.4.2 Shadow Program Investigation

Another semi-dynamic analysis is the shadow program investigation. A shadow program is the unintended and not necessarily consecutive program ‘behind’ the original program P_M . The shadow program is revealed when the PC is misaligned and the misalignment is not detected by a service (no functional error of the microprocessor). Some microcontrollers allow, for example, both 2-byte machine instructions and 4-byte machine instructions to be mixed in the program P_M (so does the Infineon 80C167). A misaligned PC pointing into the middle of a 4-byte instruction will not be detected by the microprocessor hardware. The remaining two bytes, and perhaps also the following two bytes, will be interpreted as one instruction code. Execution may cause an immediate error (e.g. illegal instruction), but may also be performed without complaints. The execution may then be continued for several shadow instructions until either an error occurs or the shadow sequence evolves back into the regular program P_M . The major thing that may not happen during a shadow path is that the sequence contains instructions that send data into the output area L^O of the storage space. In the shadow program investigation, the simulator therefore purposely jumps into all ‘long’ machine instructions of the program (these are known from the previous program path analysis) and executes the facing shadow sequence. Spurious machine instructions are then reported.

For this analysis no input is required. The only problem will be indirect addressing where the final destinations depend on some input data. A warning may be given on these services in order to have this situation repeated in a

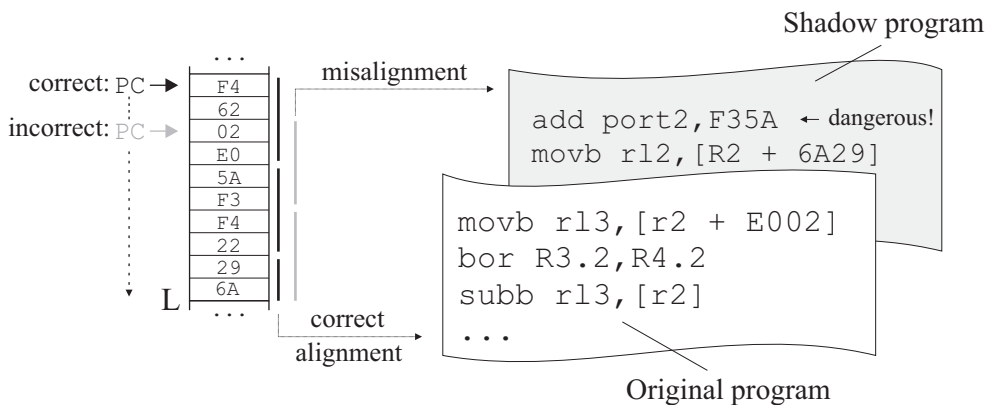


Figure 6.26: Shadow program through misalignment

regular simulation. The example shown in Figure 6.26 is taken from a 80C167 program. The shadow program is sending data to `port2` in the output area, which is considered dangerous.

6.9.5 Feasibility and Availability

Such a simulator certainly is technologically feasible. There are no conceptual problems that are not yet known and solved. The simulator certainly will not fit into a pocket, but is likely to occupy a rack. Technological feasibility is not the problem, the problem is the practical availability to those concerned about dependability evaluation of embedded software. For obvious reasons, the purchasers of the microcontrollers cannot be asked to develop the simulator on their own. Although the required hardware may be affordable, the costs for the development are not negligible. Also would it make little sense to have different customers constructing perhaps the same simulator – not only for cost reasons, but also because of compatibility aspects. The results might be customized solutions again. Furthermore, some of the information needed for developing the simulator may not be available to the public, but remain with the manufacturer of a microcontroller.

The ideal solution would be to have an independent consortium that constructs and releases these simulators. There would be a common and standardized scheme on the service specifications, on the error behavior model, and on the evaluation method. Regarding the simulators, given that most microcontrollers used in safety-critical embedded systems feature similar func-

tionalties, it may even be possible to develop a universal simulator which is then adjusted to emulate a particular microcontroller through the application of corresponding configuration files provided the microprocessor manufacturer. One file is dedicated to the service provider model (containing the services and the naming of the storage space locations), the other file contains the timings and features of the communications channels to the outside. This would, as a further benefit, allow manufacturers to pre-release new microprocessors (or variants of existing processors) for acceptance tests without having them produced yet. Depending on which parties are involved, the consortium might also be authorized to finally certify the fault-tolerance or the safety of embedded systems.

6.10 Summary

In Chapter 2, after outlining the limitations of the existing approaches, the *FARM* sets presented in [Ar90] were recalled and discussed. Based on the idea of characterizing fault-injection through a collection of sets, the collection was then broadened and adjusted to the herein considered object and subject of investigation, and the requirements for enabling more comparability in the evaluation of the fault-tolerance of safety-critical embedded software through fault-injection were set up. In this chapter, following these requirements, the sets were given shape, resulting in the method of mutant-injection.

The F set consists of mutants which are deliberate service errors. Mutants directly affect the controlling process and thus form an object-appropriate fault input. The sets A and R were recalled to be given life through having the controlling process operating in its destination environment. Then definitions on common notions N were given. It was distinguished between cognitive fault-tolerance mechanisms (detection and handling) and non-cognitive mechanisms (masking, blanking). Also was stated that a detected error is not yet a safe error, which especially is true for software being impacted by hardware faults. Following [Cu99], the notions of a fault scenario and an error scenario were introduced. In the predicate section, the major predicates P for the observation and classification of the fault effects were presented and discussed. The section was followed by a collection of valuation rules V . Major topics were the period of grace and the problem of redundant experiments. It was alluded that redundant experiments may water down the expressiveness of the fault-tolerance measures. General issues, such as time-censored data and the golden run were

addressed as well. The golden run was argued not to be the ultimate basis for comparison, because it is not the task of the controlling process to align with the golden run in the presence of problems, but to cope with the problem its own way. In the section on the measures M , a recall on the coverage proportion and on the time interval distributions was given. The fault-tolerance coverage and the error detection coverage were discussed in particular. Finally a proposal of a simulation-based fault-injection environment was presented. Its benefits in general, but in particular the program-path analysis and the shadow program investigation, were accentuated.

6.11 Discussion

6.11.1 Summary Mutant-Injection

THE PURPOSE

Mutant-injection is a fault-injection method for the purpose of measuring the interior hardware-fault fault-tolerance of safety-critical embedded software in a – as far as possible – comparable manner. The goal is to achieve meaningful and comparable measures through a standardized evaluation procedure. Mutant-injection predominantly aims at real-time execution, injection, and observation. This is because the only comparability requirement that can be met regarding the input space of different software on different systems is to have the software operating in their real environment. Mutant-injection encompasses the actual injection and observation process, as well as the process of valuating and revising the results obtained from the experiments. The principle of the method was summarized in the introduction of this chapter and was depicted in [Figure 6.1](#) (pg. 118).

THE BASIS

Mutant-injection is based on the idea of the *FARM* sets of [Ar90]. According to the publication, fault-injection can be characterized by a collection of sets, which had been exemplary shown for physical fault-injection experiments. In this work these sets were taken up and were ported to the herein considered subject and object of the evaluation: the interior hardware-fault fault-tolerance of the controlling process. The most essential set characterizing the method is the fault set F . Distinct from other fault-injection approaches in this field, the

fault set consists neither of hardware faults nor of software faults, but consists of process faults. The faults in F are mutants (deliberate service errors) which form the representatives of the random faults at the level of the controlling process. Because mutants are action errors and because mutants – respectively the services – are the structural components of the controlling process, mutants are the most object-appropriate fault input. Since all software share similar services – at least to some extent – and since the meaning of a service to the controlling process is invariant of the actual hardware — within limits, mutants allow for a comparable fault input domain among different experiments. Comparable output requires comparable input. Mutants are the smallest common denominator that can be achieved among different software. The faults traditionally used in this field are for the most part specific to the target hardware, and therefore are hardly comparable. This is certainly true for low-level faults, whether simulated or physical, or for the faults injected into the memory locations (SWIFI technique). The meaning of a particular corrupted memory location, for instance, cannot easily be transferred to another software on another system, the meaning of a corrupted service however is transferable — with certain limitations, as a matter of fact. Mutant-injection allows for comparable measures of the fault-tolerance, based on a comparable fault input and a collection of sets for a common evaluation procedure (notions N , predicates P , valuation rules V , and the coverage proportion as the measure M). With mutant-injection it is possible to design fault-injection based dependability benchmarks for software.

THE SOFTWARE

Mutant-injection is predominantly dedicated to safety-critical embedded software, but may on principle be applied to any software. However, only safety-critical embedded software has the chance to be executed real-time in a simulator, this is hardly possible for customary computer software, such as the conventional operating systems or software applications used on home computers or workstations. Moreover, the microcontrollers used in safety-critical embedded systems are much closer to each other qua instruction sets (and thus services) than are the high performance processors of computer systems. It will be more intricate to build service-provider models for the latter type of processors. Finally, the notion of fault-tolerance has another significance for safety-critical embedded systems – it is just vital – than it has for ordinary computer systems. This also holds for safety-critical systems used in indus-

tries (e.g. power plants) or in aerospace as there usually is a high amount of hardware redundancy present, such that the responsibility of the software, at least regarding its interior hardware-fault fault-tolerance, is much lower than it is with safety-critical embedded systems.

THE HARDWARE FAULTS

Mutant-injection focuses on random faults in the processing hardware. These faults cause mutations of services. Retention errors which have been introduced in Chapter 4 for the reason of completeness do not cause process faults. Similar to the exterior fault-tolerance of the controlling process, the investigation of the effects of retention errors onto the controlling process does not require the injection of mutants. Nevertheless, since retention errors are only relevant when being read-assessed by a service, services can be used to simulate the occurrence of retention errors in the storage space. In any case, the procedure following the occurrence of a retention error may follow the guidelines that were proposed in this chapter (predicates, valuation rules, final measures). Similar applies to faults entering the controlling process through the interface area.

APPLICABILITY

Although the mutants, which are action errors, can only be injected by means of simulation-based fault-injection, the philosophy behind the method may principally be applied to other fault-injection approaches — however with the limitations imposed by the corresponding injection technique. Mutant-injection then is a methodology, requesting the other approaches to observe and register the effects of the injected low-level faults at the service level (mutants) in order to obtain a comparable input, and to follow a common procedure in the derivation of the final measures. Mutant-injection therefore does not make other injection techniques obsolete, on the contrary, physical fault-injection and low-level simulation-based fault-injection experiments are the most important means to obtain a notion of realistic service errors.

6.11.2 Existing Approaches

The existing approaches for dependability evaluation of software through fault-injection, as discussed in Chapter 2, are customized solutions. None of the

publications however claimed to have presented unbiased and comparable measures. On the contrary, some authors annotated the lack of portability and comparability of their approach. Without any judgement, the major differences between mutant-injection and the existing approaches are as follows.

- Mutant-injection emphasizes the software point of view on an executing device. A microprocessor is regarded as a provider of enclosed individual services (service-provider model). Services are the dynamic responses to the static machine instructions. Data exchange among services, and between services and the environment, takes place only via the storage space. The hardware constituting the services upon request of the machine instructions is the processing hardware.
- In mutant-injection the object of investigation (the software) is defined as a process, termed the controlling process. The controlling process is a sequence of services. Services are the basic components of the controlling process. The process communicates with its environment through well-identified input and output locations in the storage space (the interface area). There is a clear boundary between the object and its environment, which allows for a precise determination of when, where and how the hardware faults enter the controlling process.
- Mutant-injection uses purposive faults that are common to most safety-critical embedded software, and that form a junction among the different fault-injection techniques. The injected faults are neither hardware faults nor software faults, but are object-adequate action-errors (process faults). The fault input domain is not specific to the hardware and thus allows for comparability and portability of the injected faults.
- Mutant-injection encompasses a procedure for the evaluation of the interior hardware-fault fault-tolerance — from the initial fault scenarios to the final measures.
- Masking and blanking are acknowledged as fault-tolerance mechanisms of software. Both masking and blanking are non-negligible software qualities to be considered in a fair evaluation process.
- Because the only comparability requirement that can be satisfied regarding the input space of different software is to have the software operating in their destination environments, mutant-injection proposes the

simulation-based fault-injection technique in order to allow for real-time execution, injection and observation. Both physical fault-injection and SWIFI do not allow for real-time injection and monitoring.

6.11.3 Remaining Problems

The creation of a service-provider model of a microcontroller surely poses some difficulties if the manufacturer withholds information. However, from the conventional microcontroller documentation and from engineer’s intuition, a service provider model can be constructed to a certain extent (respectively accuracy). In any case, mutant-injection can also be used on a traditional register-model simulator (if it allows service manipulation), but with the restrictions outlined in Chapter 4. A real-time fault-injection environment, such as the one proposed in this chapter, certainly poses a financial problem, but no technical one. Obtaining realistic mutants may be intricate, but is feasible (Chapter 5 — related work in fault-mapping). Anyway, following the recommendation of [Vo98 p.25] to “...avoid the trap of spending all of our time worrying about how realistic certain anomalies may be, and simply observe how those anomalies impact the software”, one may start with artificial mutants anyway.

One of the major problems to be solved, not only for mutant-injection but for all methods aiming at evaluating the dependability properties of software, lies in the categorization and standardization of equivalent error scenarios (the behavior of the controlling process after fault-injection). Clearly, if two scenarios are physically identical (same sequence of services, same errors and same propagation in the storage space) then there is no doubt. But the controlling process may, as an example, accidentally branch to another path and may then be delivering the same sequence of services as on the original path. Both paths shall be assumed to merge again at some later point. From the core implementation perspective the scenarios are different (wrong program path), from the application point of view the scenarios are equal since the process successfully renders the requested series of services. The fundamental question behind the equivalence problem is: Must the controlling process exactly do as programmed, or must it finally do as required from some higher perspective? It is to remind – also following from the service-provider point of view onto any kind of processor – that mutant-injection is not necessarily limited to software, but to any kind of process. Regarding software, research must be carried out

for classifying distinguishable abstraction levels within the controlling process as well as for identifying the corresponding ‘errors’ of the process. The lowest process level is the level of the individual services, the highest level is the application level, but there are intermediate levels within. These levels need to be identified and standardized, such that conformity can be achieved about the (mis)behavior of the object of investigation. In any case, the equivalence of error scenarios must be taken into account because otherwise the measures may not be credible, as was pointed out in this chapter.

Strongly connected to the latter problem certainly is to generally achieve a common agreement about the complete evaluation procedure. This problem can only be solved by a joint community of the concerned, that is, the software developers, the microprocessor manufacturers and the researchers from the field of dependability evaluation and validation. This work intends to point onto the problems as well as to indicate a solution. In any case, there must be some standardized and commonly accepted evaluation method, otherwise the obtained measures are more or less elusive. This can be dangerous, especially for safety-critical embedded systems, as several accidents have shown in the past.

Chapter 7

Summary

7.1 Problem Recall

Safety-critical embedded software can be affected during operation by random faults in the processing hardware. These faults influence the execution process of the machine instructions. The fault-tolerance of the software with respect to these faults is significant for maintaining system safety.

Software must not only be fault-tolerant, it must also prove to be fault-tolerant. Therefore the fault-tolerance is to be evaluated and put into expressive measures. Fault-injection is an accepted method for dependability evaluation. Several approaches for evaluating the dependability of software through fault-injection were presented in the past. Most of the approaches are however customized solutions. The measures obtained from one approach often cannot be compared against those of the other approaches. A mutual basis allowing for conformance among the experiments and thus for comparability of the results is missing. The approaches also do not address safety-critical embedded software, in particular the evaluation of its fault-tolerance in the presence of random faults in the processing hardware.

This thesis tried to proceed towards comparability in fault-injection for software dependability evaluation, however specifically dedicated to safety-critical embedded software and its dependability property ‘fault-tolerance’ in the presence of random faults affecting the machine instruction execution. For that, a fault-injection method was derived and presented in this work.

7.2 Review of Chapters

Before summarizing the work, the major steps taken throughout the chapters towards the presented fault-injection method are itemized.

Chapter 1 — Introduction

- Introduction into the type of safety-critical embedded system considered, and specification of *fault-tolerance* as the property of an object to prevent fault effects from leaving that object unnoticed.

Chapter 2 — Fault-Injection for Software Dependability Evaluation

- Comparison of the three hardware fault-injection techniques, concluding that only the simulation-based technique on principle allows real-time execution, injection and observation — given that a suitable model of the microprocessor exists.
- Discussion of related work in software dependability evaluation through fault-injection and derivation of comparability requirements, starting from the *FARM* sets originally introduced by [Ar90] for pin level fault-injection.

Chapter 3 — The Controlling Process

- Definition of the ‘software in execution’ as *controlling process*, thereby avoiding terminological confusion and realizing the nature of the object as well as its boundary.
- Specification of the *processing hardware* as the hardware areas relevant for the execution of the machine instructions.
- Identification of the *interior hardware-fault* fault-tolerance of the controlling process as the fault-tolerance category of concern.
- Deduction of *process* fault-injection as the injection method following the nature of both the object and the effects of the faults.

Chapter 4 — Microprocessor Modeling

- Introduction of the notion *service* as well-specified behavioral answer upon a machine instruction, thereby also filling a gap in terminology.

- Presentation of the *service-provider* model, a generalized model of a processor from a strict software point of view, consisting of a *storage space* and a set of enclosed services.
- Presentation of the error model outlining the principal effects of random faults on the model components, and identification of services as natural link between hardware and software on the *path of impact*.
- Determination of service errors as the representatives of the considered random faults, forming the fault input domain in the method.

Chapter 5 — Service Errors

- Discussion of publications related to fault mapping and error behavior modeling, concluding that the approaches are still implementation oriented and that the service level has not yet been considered.
- Presentation of the error behavior of customary combinational circuits, based on almost 300 million fault-simulations at gate level, for the application of creating realistic service errors, thereby also showing the advantage of the *arithmetic* error over the traditional bit-flip.
- Exemplary derivation of realistic and representative service errors, but also noting that the *power-of-two* errors cannot model all service errors principally possible.

Chapter 6 — Mutant-Injection

- Definition of deliberately caused service errors as *mutants*, serving as comparable fault input domain F at the process level.
- Definition of notions N , and distinction between cognitive and non-cognitive fault-tolerance mechanisms.
- Determination of *fault scenarios* as input in the injection experiments, resulting in *error scenarios* as the output — following the ideas of [Cu99].
- Presentation of predicates P for a consistent classification of the fault effects, and presentation of valuation rules V for the revision of the obtained error scenarios.
- Proposal of a simulation-based fault-injection environment for real-time execution, injection and observation of the controlling process.

7.3 Summary of the Work

In this work a fault-injection method, called mutant-injection, was presented. The method focuses on the evaluation of the *interior hardware-fault* fault-tolerance of safety-critical embedded software. This fault-tolerance category is concerned with the effects arising from random faults in the processing hardware. Mutant-injection allows for comparable measures of the fault-tolerance of different software on different hardware. [Figure 7.1](#) summarizes the method in brief.

The presented method is built upon the so-called *FARM* sets which had initially been introduced by [Ar90] for physical fault-injection experiments. These sets form the basic entities present in any fault-injection experiment. The collection of sets was expanded and adjusted in this work to the herein considered object and subject of investigation, resulting in a collection of requirements for more comparability of the final measures. The pivotal point identified is a common and object-appropriate fault set F . Preferably, the faults directly affect the structural elements of the software.

The object of investigation – the software in execution – was specified as controlling process. This reveals the nature of the object, its structural elements, and its boundary, and also avoids terminological ambiguities. The two components establishing the controlling process were determined as the processing hardware and the binary program. The controlling process represents the amalgamation of hardware of software, and allows the embedded system to be conceptually partitioned into two tangible entities: The controlling process as the organizing and supervising entity (the intellectual part), and the remaining hardware as the expedient entity (the servant part). It is the controlling process that is predominantly responsible for maintaining safety through fault-tolerant reactions upon faults. Its fault-tolerance is decisive.

According to the locations of occurrence of faults, three fault-tolerance categories of the controlling process were distinguished. The interior hardware-fault fault-tolerance was identified as the category of concern (the subject of investigation). It denotes the ability of the controlling process to tolerate internal fault-events arising from random faults in the processing hardware. The other two categories are not concerned with any hardware defects impacting the execution of the machine instructions. Corresponding to the nature of the object and to the first-order effects of the random faults on the object,

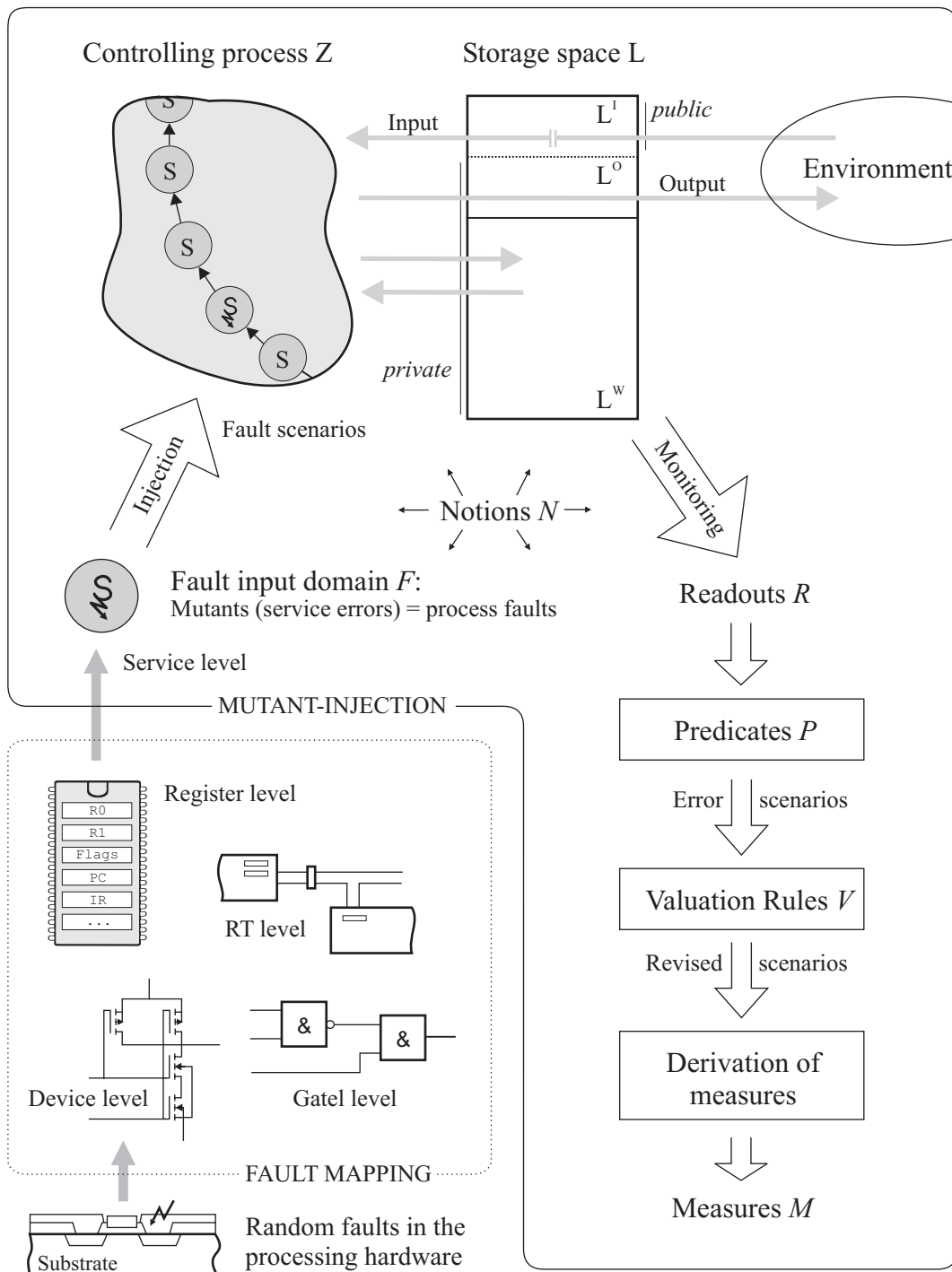


Figure 7.1: Summary Mutant-injection

the conceptual injection method was determined as process fault-injection. In contrast to other approaches, the fault input domain consists neither of hardware faults nor of software faults, but of process faults. Any considerations on the fault-tolerance capabilities of the controlling process start with the occurrence of a process fault. Therewith a clear boundary exists as to when and where the random faults become relevant for the controlling process. This, the conceptual isolation of the object of investigation from the surrounding system, and also the clear specification of the location of the communication interface is one shortcoming noticed in other software dependability evaluation approaches.

Within the scope of the presented service-provider model which is a generalized model of a microprocessor from a strict software perspective, the controlling process was defined as a sequence of independent services. The introduced notion of a service not only fills a terminological gap, but also conceptualizes the link between hardware and software. A service is the dynamic response to a static machine instruction. For any hardware fault propagating into the controlling process, services form the inlets into the controlling process. Service errors in particular are the representatives of random faults in the processing hardware.

As a review on publications related to fault mapping and microprocessor error behavior modeling has shown, none of the investigations had considered the service level, although in many experiments the faults have traversed the service level while propagating into the software. In order to obtain some insight into realistic service errors, the results of gate level fault-simulations carried out on customary combinational circuits used in microprocessors were presented. It was demonstrated by means of the arithmetic error distribution, that the power-of-two errors are the most likely in the presence of single and double faults. It was also shown that the arithmetic error is more expressive than the traditional bit-flip. The findings were then used for the exemplary creation of realistic service errors, however also noting, that the power-of-two errors cannot model all service errors possible.

Following the requirements identified, the fault-injection method was then assembled. The hub of the method is the fault set F . The set consists of mutants which are deliberate service errors. Mutants are process faults and thus form an object-appropriate fault input. Because all safety-critical em-

bedded software share a certain set of equivalent services (e.g. ADD, SUB, MUL, MOV, CMP), and because the meaning of a service to the software is independent of the actual microprocessor architecture (Harvard, von Neumann), of the data formats (little endian, big endian), and also independent of the size of the machine instructions and their location in memory, mutants form the smallest common denominator that can be attained among different software on different hardware. Mutants are predestined to form a – as far as possible – comparable fault set among different software. Low level faults (e.g. pin level faults, stuck-faults) or state mutations applied to the memory, as used in other approaches, are far less comparable since they are specific to the hardware that the software is executed on.

Another problem hampering comparability is the sometimes varying interpretations of common terms. Therefore relevant notions N were defined, in particular it was distinguished between cognitive (detection and handling) and non-cognitive fault-tolerance mechanisms (masking, blanking). Following the ideas of [Ar90], a collection of predicates P for a consistent classification of the monitored fault effects was presented. Predicates are Boolean assertions that strip the required information from the experiment readouts. Also some valuations rules V , to be taken into account when revising and purging the experiment results, were given. General issues, in particular the golden run, were addressed as well. The golden run was argued not to be the reference criterion because the controlling process usually follows other program paths in the fault-case (e.g. detection and handling) than in the good-case. It is not the task of the controlling process to align with the golden run, but to hinder the fault effects from propagating into the output stream. For rating the fault-tolerance, the coverage proportion was proposed as a suitable measure M .

Since the only comparability requirement that can be met regarding the input space of different software on different hardware is to have the software operating in their real world, the concept of a simulation-based fault-injection environment was presented. Its benefits in general, and in particular its usefulness for an automatic program path analysis and a shadow program investigation were discussed. Finally, remaining problems were addressed. The major problem to be solved is a standardized categorization of process errors in order to recognize the occurrence of equivalent error scenarios. Equivalent error scenarios result in redundant experiments. Redundant experiments may lead to bogus fault-tolerance measures.

Mutant-injection predominantly aims at real-time execution, injection and observation of the controlling process. However, mutant-injection is not per se a real-time injection method and moreover, mutant-injection does not make other fault-injection techniques obsolete, as was discussed. Mutant-injection is also considered a methodology, requesting other approaches in this field to observe and name the effects of the injected low-level faults at the service level (mutants) in order to obtain a comparable input, and to follow a common procedure in deriving the final measures.

Therewith a methodical fundament is built that allows for comparability of the obtained measures — as far as this is even possible in fault-injection.

7.4 Prospects

The herein presented fault-injection method is not a ready-to-go manual for achieving unbiased and comparable fault-tolerance measures. There are still problems to be dealt with, most of them are solvable only by a joint community of the concerned (software developers, microprocessor manufacturers, dependability researchers). One step to be taken is to have the manufacturers providing service provider models of their products. Another step is to create a standardized and comprehensible description for qualifying service errors. Then a centralized library of realistic service errors and fault scenarios should be called into being. The contributions to the library may result from field experience, from low-level fault-injections or from model simulations. As shown in this work, many of the past researches on microprocessor error behavior modeling could have contributed to such a library. The major step certainly is to establish a standard for the evaluation of the fault-tolerance of safety-critical embedded software, that is, to proceed forward to commonly accepted dependability benchmarks and certifications in this particular field. This is especially important for safety-critical embedded software. This thesis intended to take a step towards this direction.

Appendix A

Error Distribution Figures

Following are shown the error distribution functions obtained from the gate level fault-simulations presented in Chapter 5 (page 80). For space reasons these figures have been moved to this appendix. The figures show the distributions of the arithmetic error in the output of the circuit,

- a) while a single fault is present in the circuit (this page),
- b) and while a double fault is present in the circuit (page 188).

A.1 Single-Fault Error Distributions

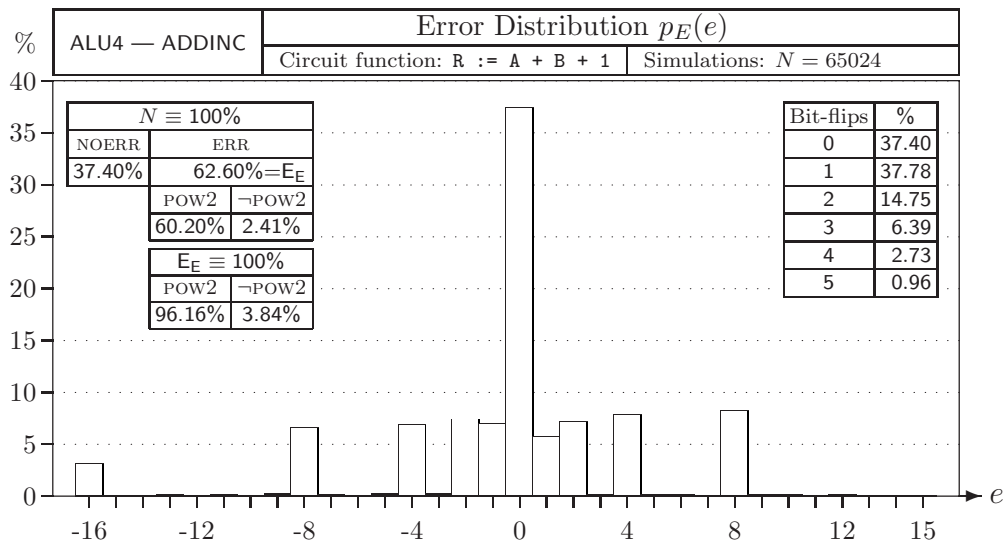


Figure A.1: Single-fault error distribution ALU4 ($R := A + B + 1$)

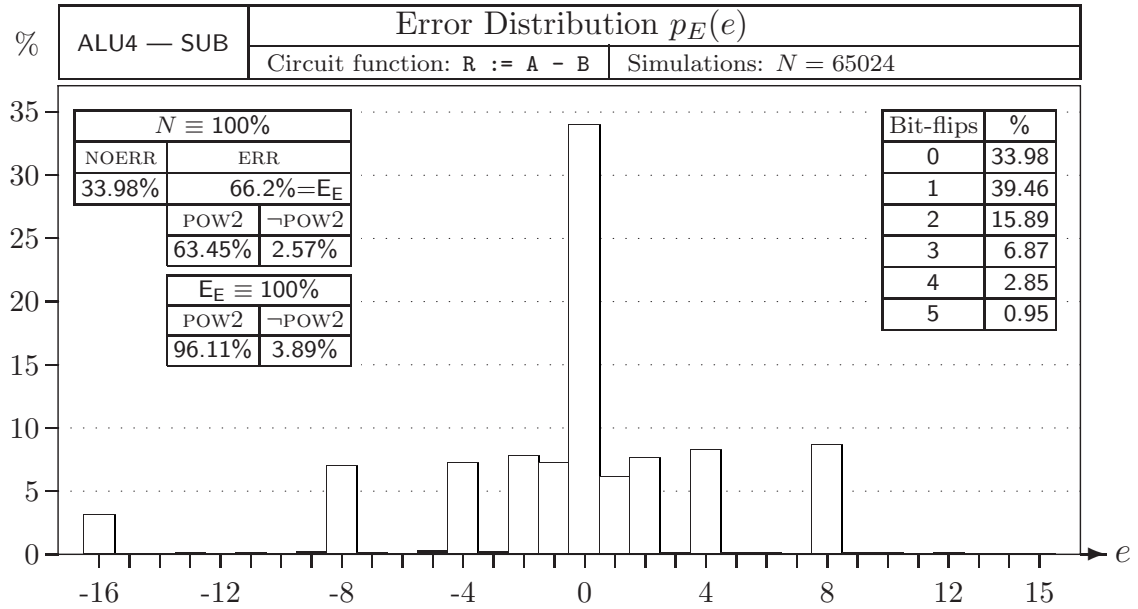


Figure A.2: Single-fault error distribution ALU4 ($R := A - B$)

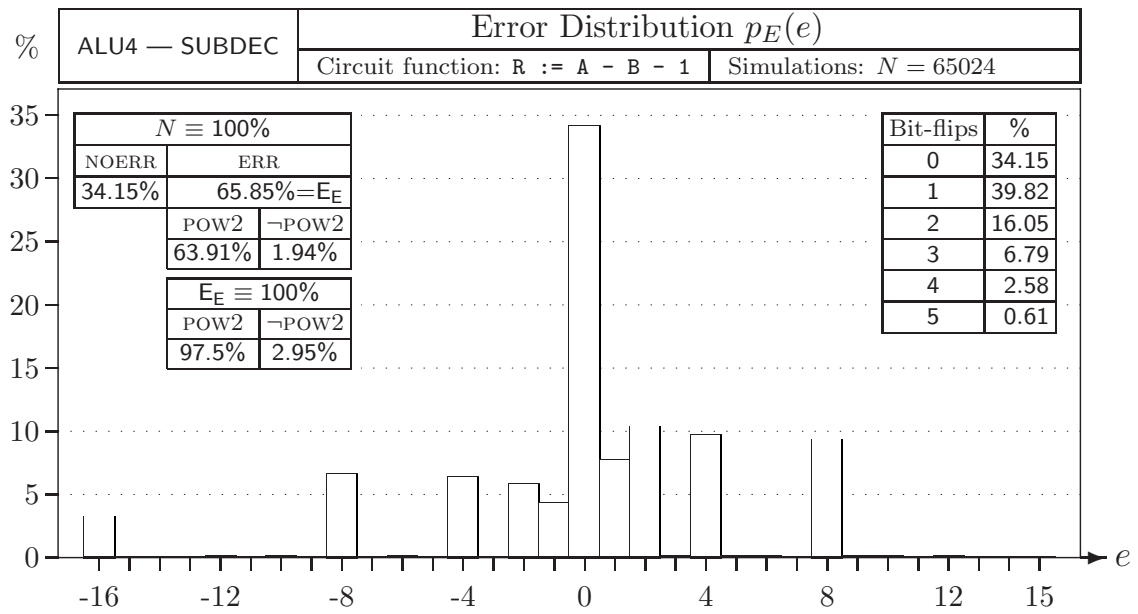


Figure A.3: Single-fault error distribution ALU4 ($R := A - B - 1$)

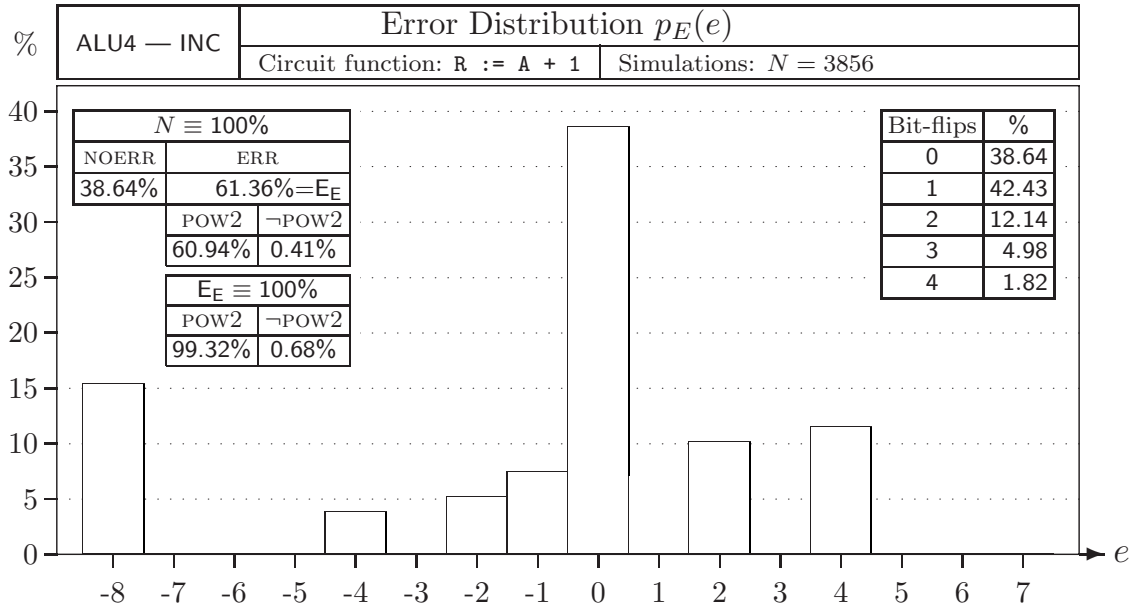


Figure A.4: Single-fault error distribution ALU4 ($R := A + 1$)

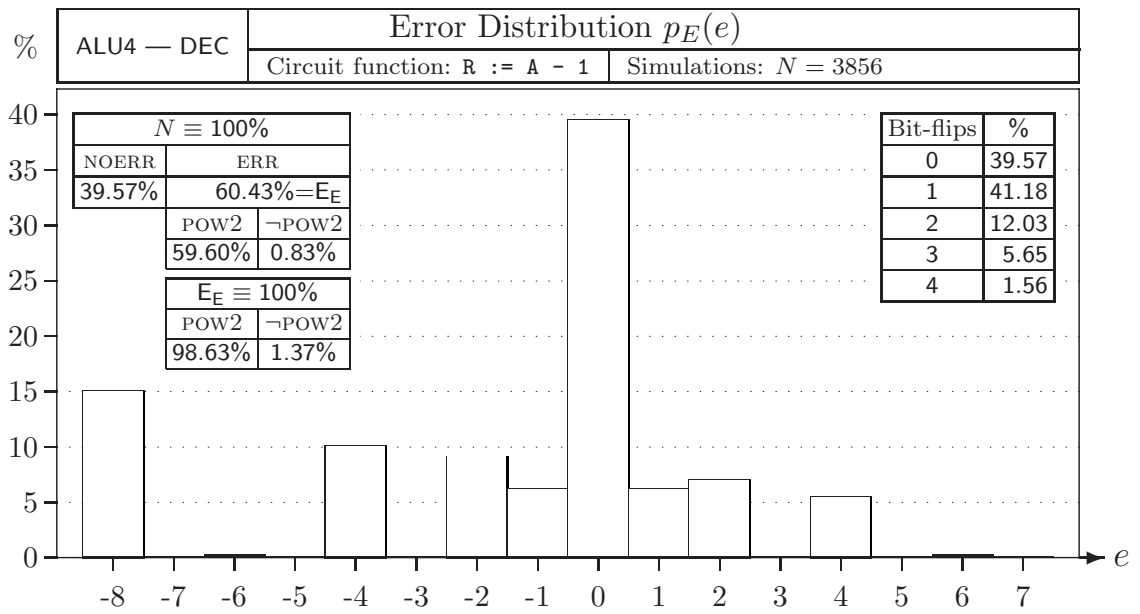


Figure A.5: Single-fault error distribution ALU4 ($R := A - 1$)

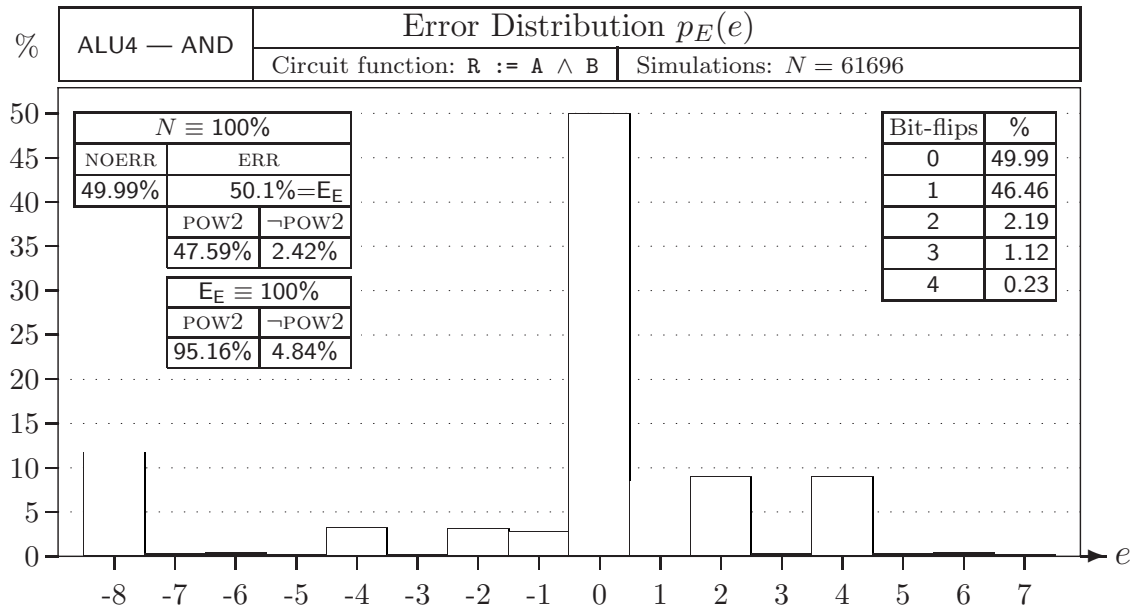


Figure A.6: Single-fault error distribution ALU4 ($R := A \wedge B$)

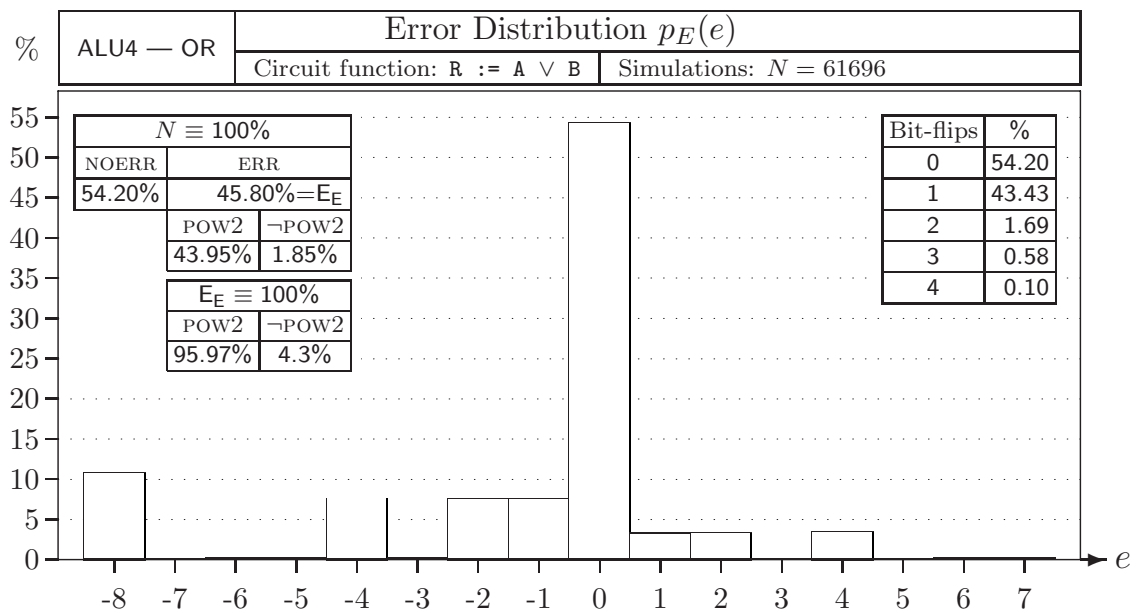


Figure A.7: Single-fault error distribution ALU4 ($R := A \vee B$)

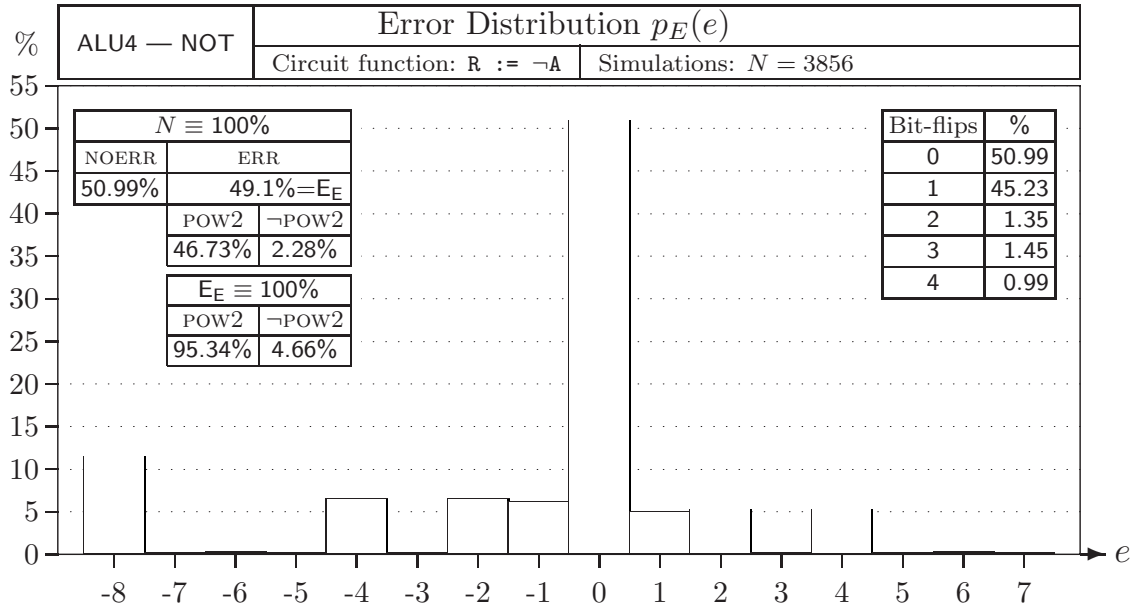


Figure A.8: Single-fault error distribution ALU4 ($R := \neg A$)

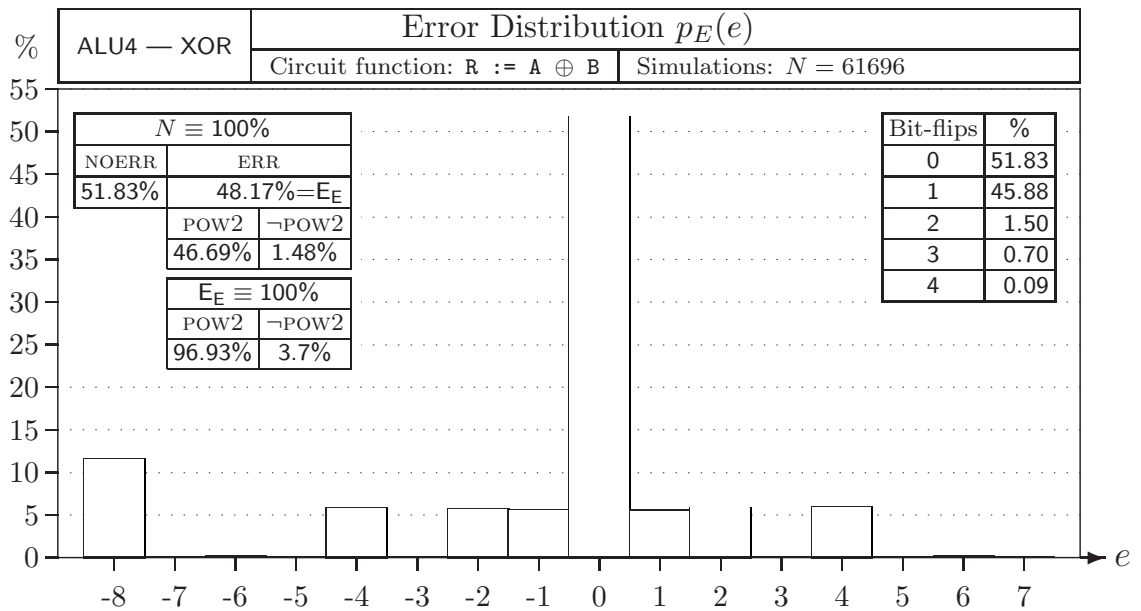


Figure A.9: Single-fault error distribution ALU4 ($R := A \oplus B$)

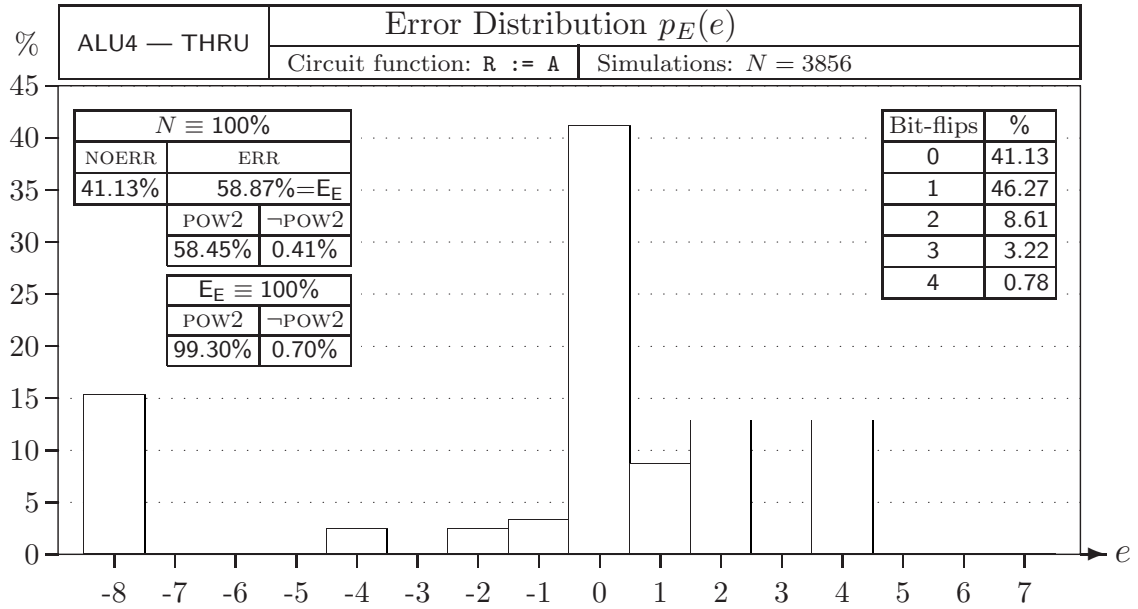


Figure A.10: Single-fault error distribution ALU4 (R := A)

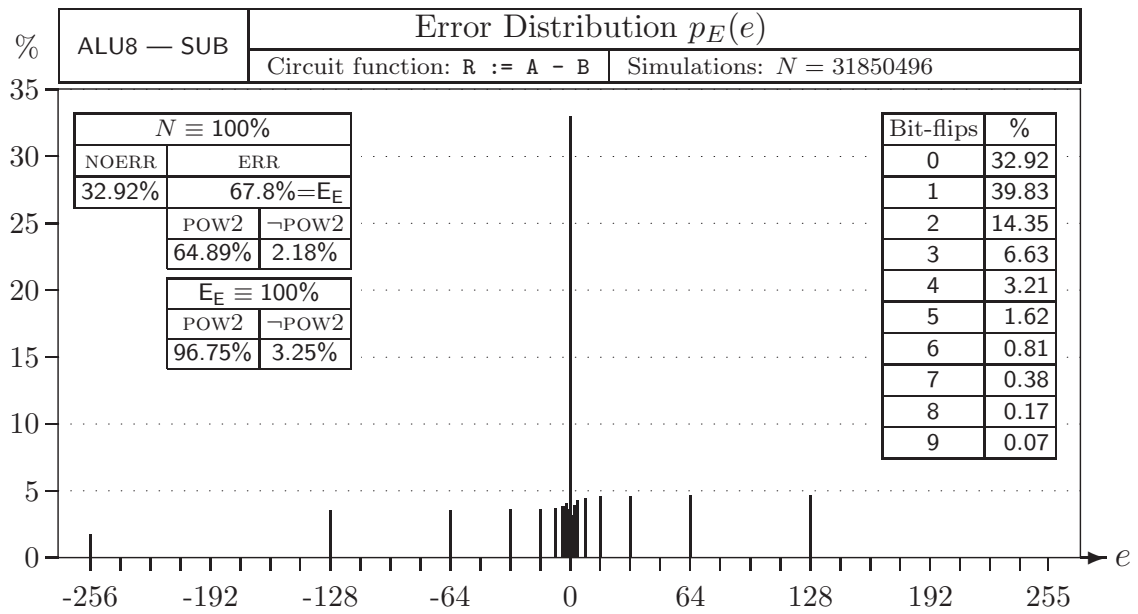


Figure A.11: Single-fault error distribution ALU8 (R := A - B)

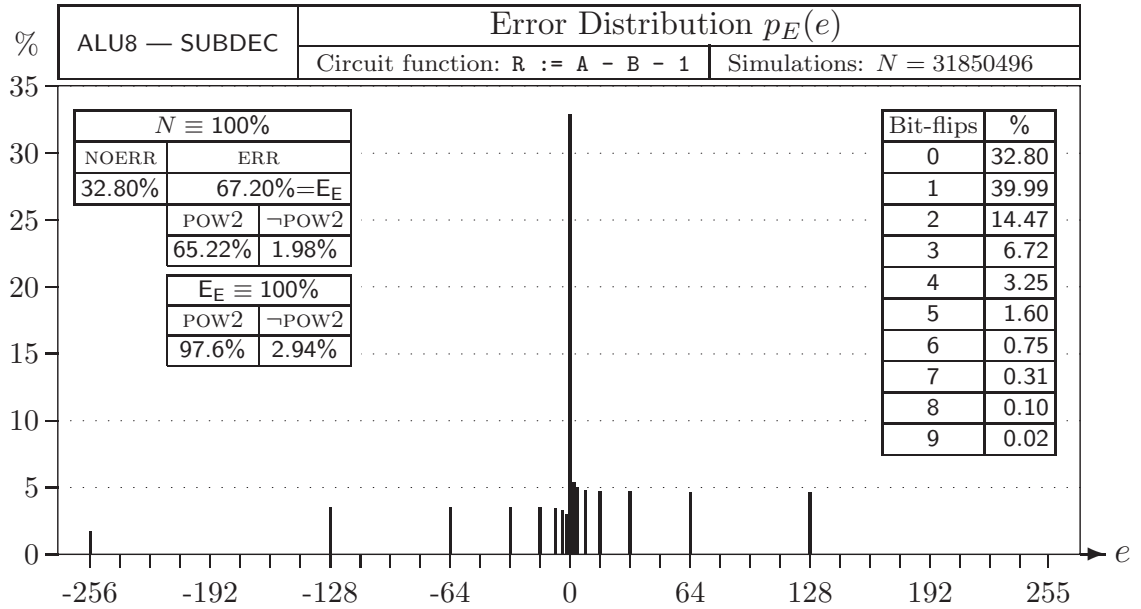


Figure A.12: Single-fault error distribution ALU8 ($R := A - B - 1$)

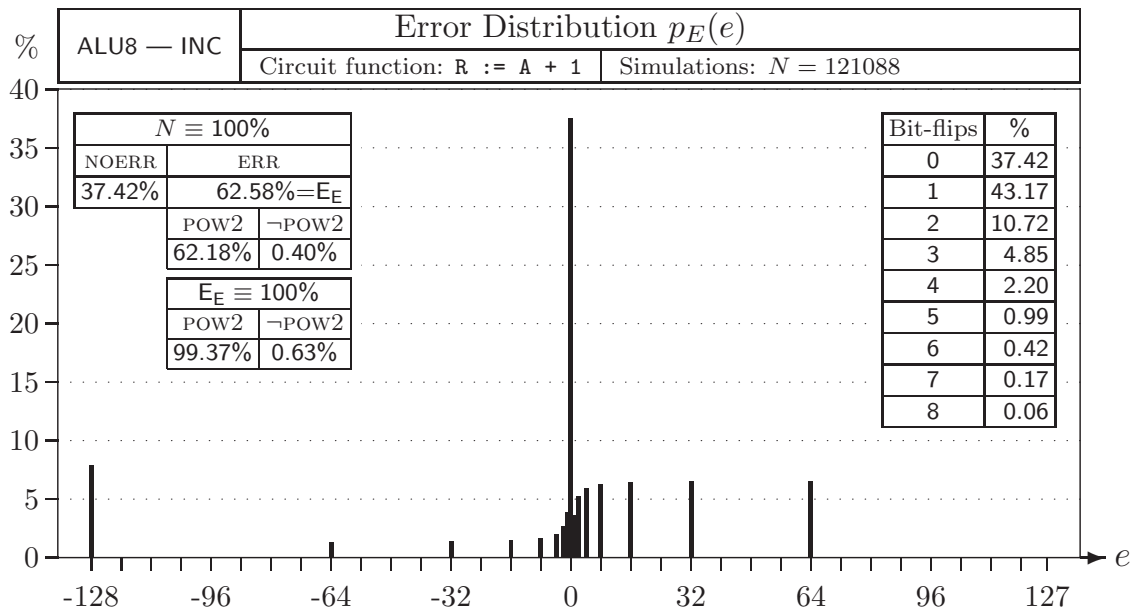


Figure A.13: Single-fault error distribution ALU8 ($R := A + 1$)

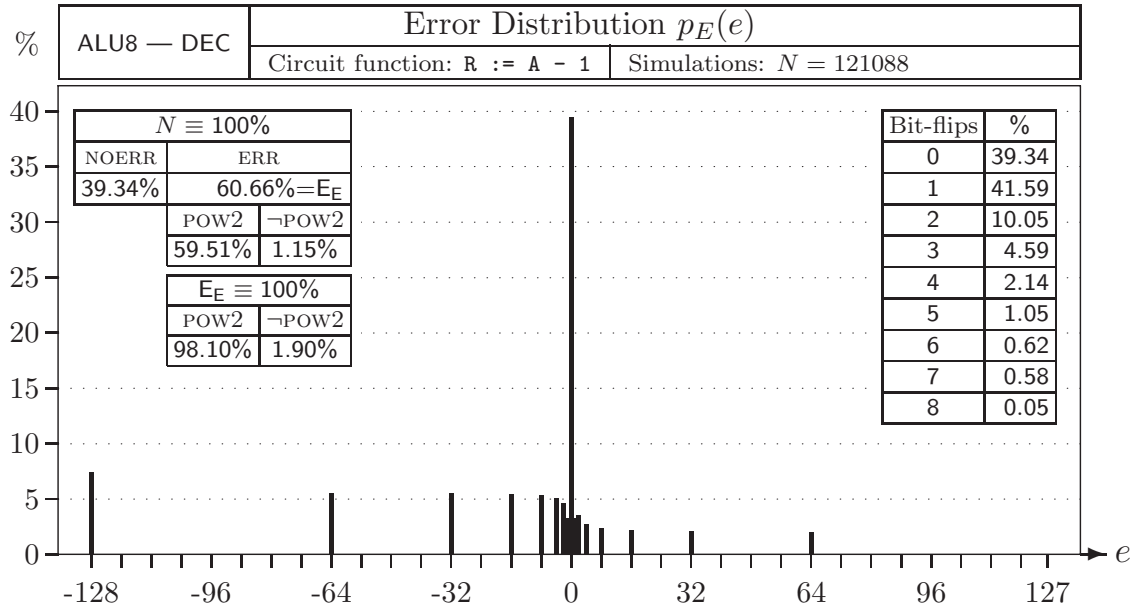


Figure A.14: Single-fault error distribution ALU8 ($R := A - 1$)

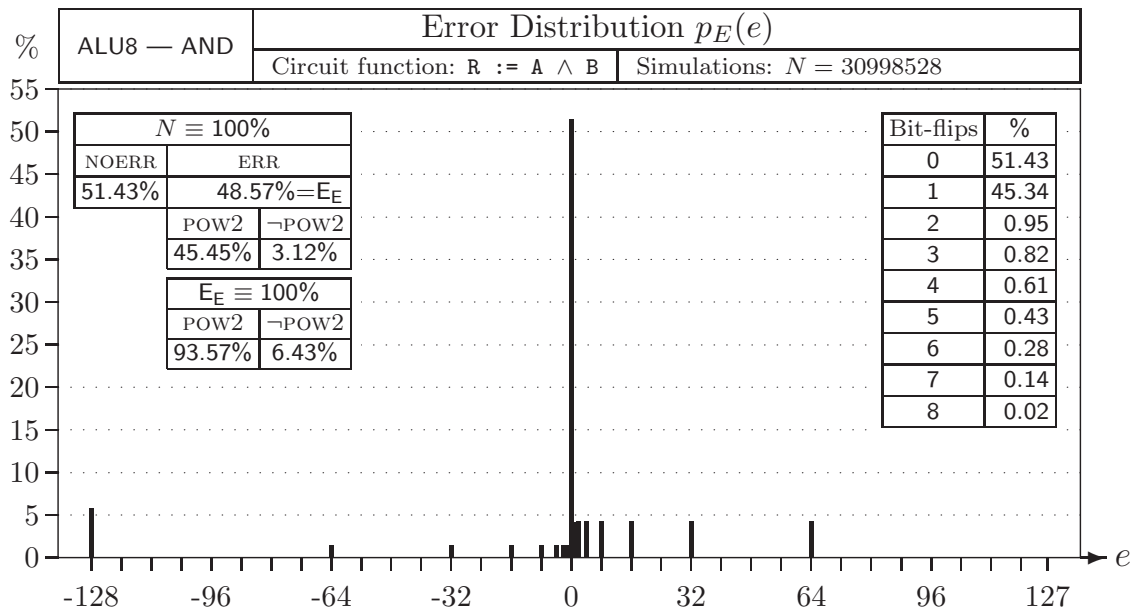


Figure A.15: Single-fault error distribution ALU8 ($R := A \wedge B$)

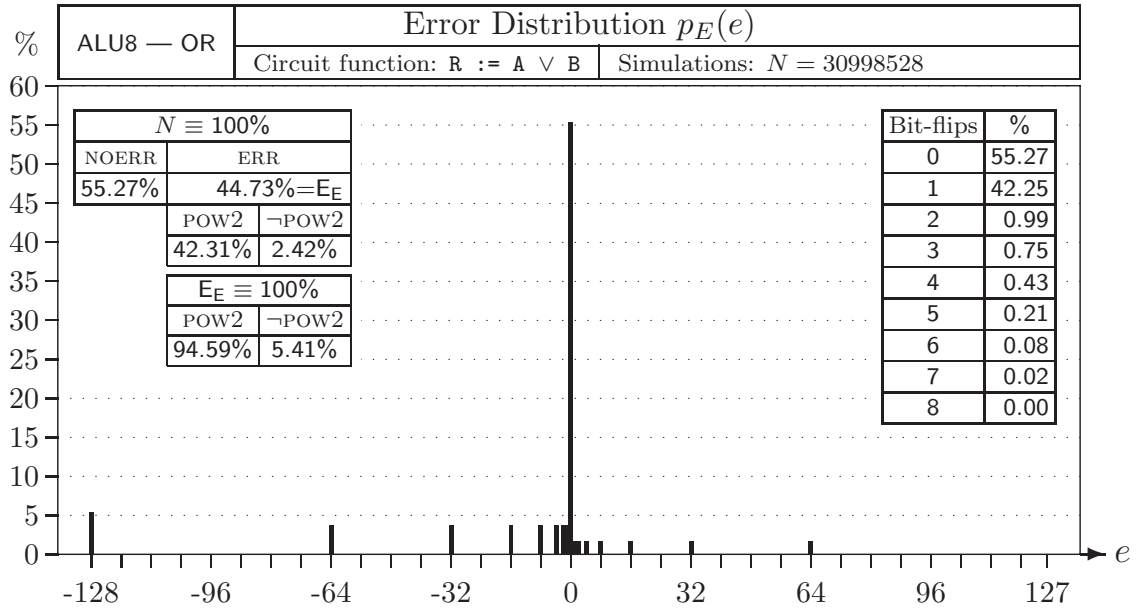


Figure A.16: Single-fault error distribution ALU8 ($R := A \vee B$)

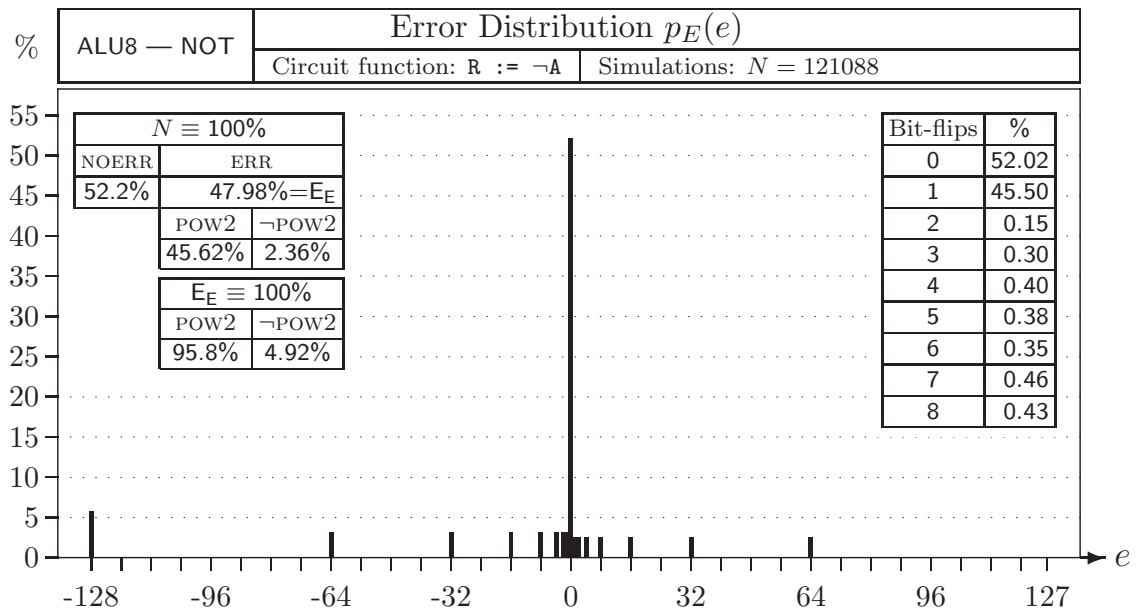


Figure A.17: Single-fault error distribution ALU8 ($R := \neg A$)

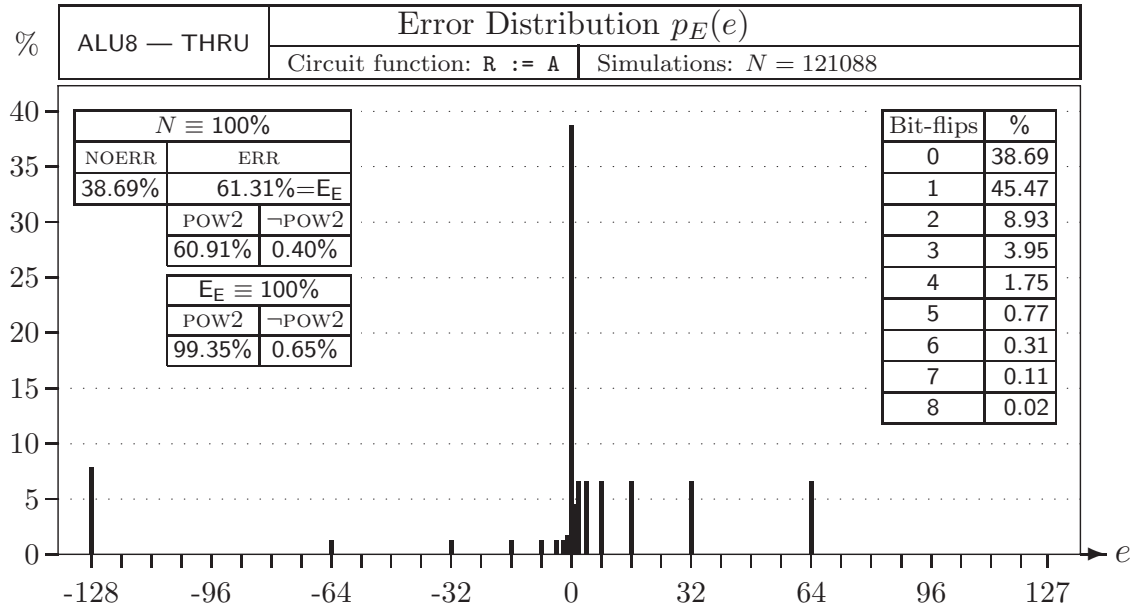


Figure A.18: Single-fault error distribution ALU8 (R := A)

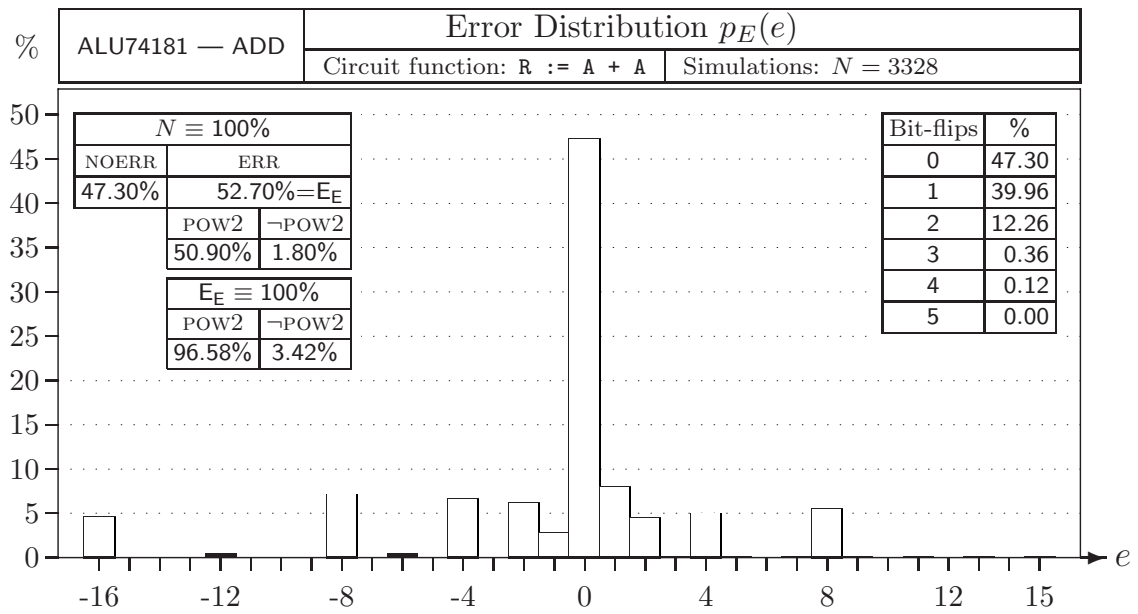


Figure A.19: Single-fault error distribution ALU74181 (R := A + A)

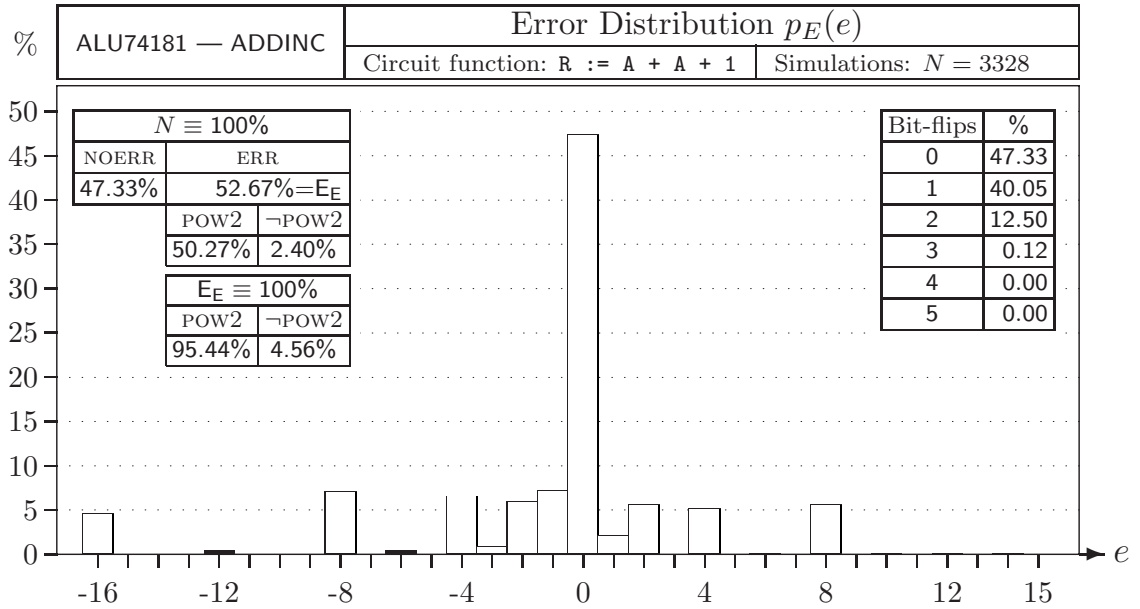


Figure A.20: Single-fault error distribution ALU74181 ($R := A + A + 1$)

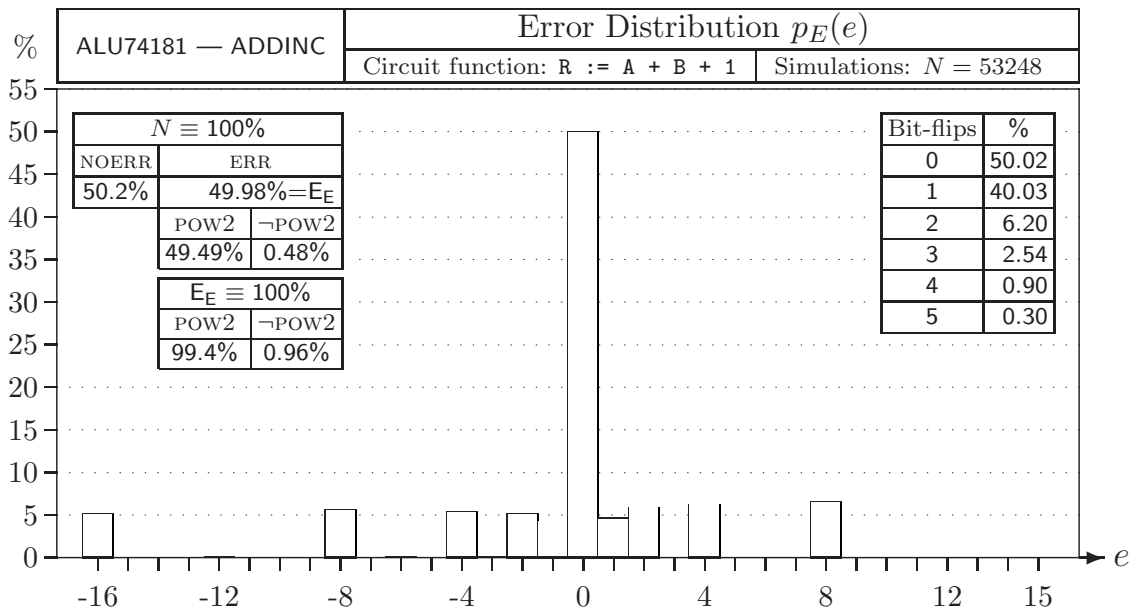


Figure A.21: Single-fault error distribution ALU74181 ($R := A + B + 1$)

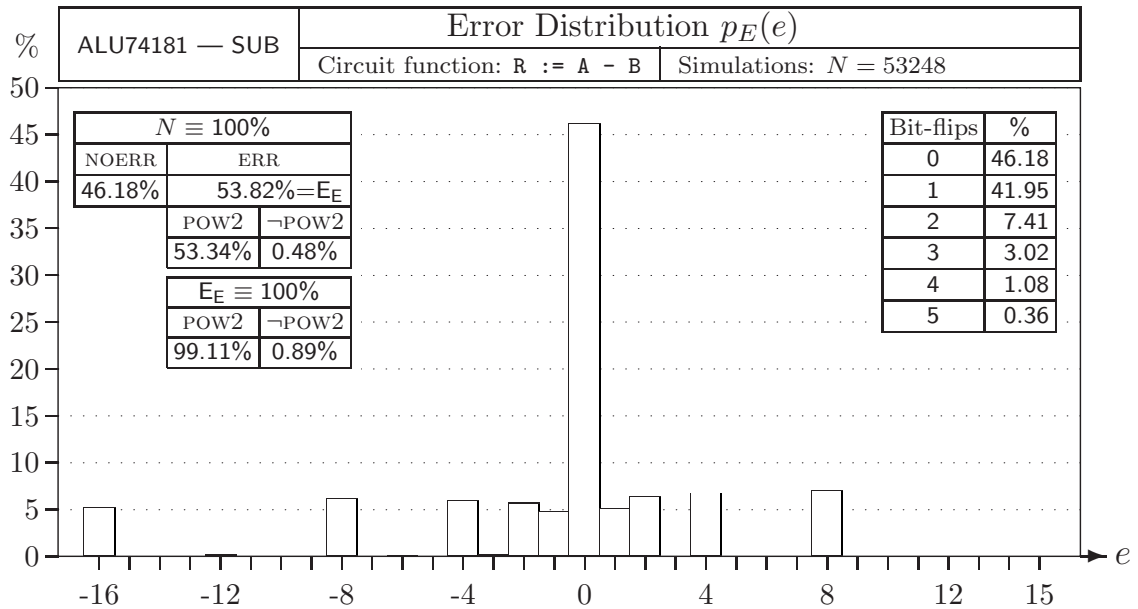


Figure A.22: Single-fault error distribution ALU74181 ($R := A - B$)

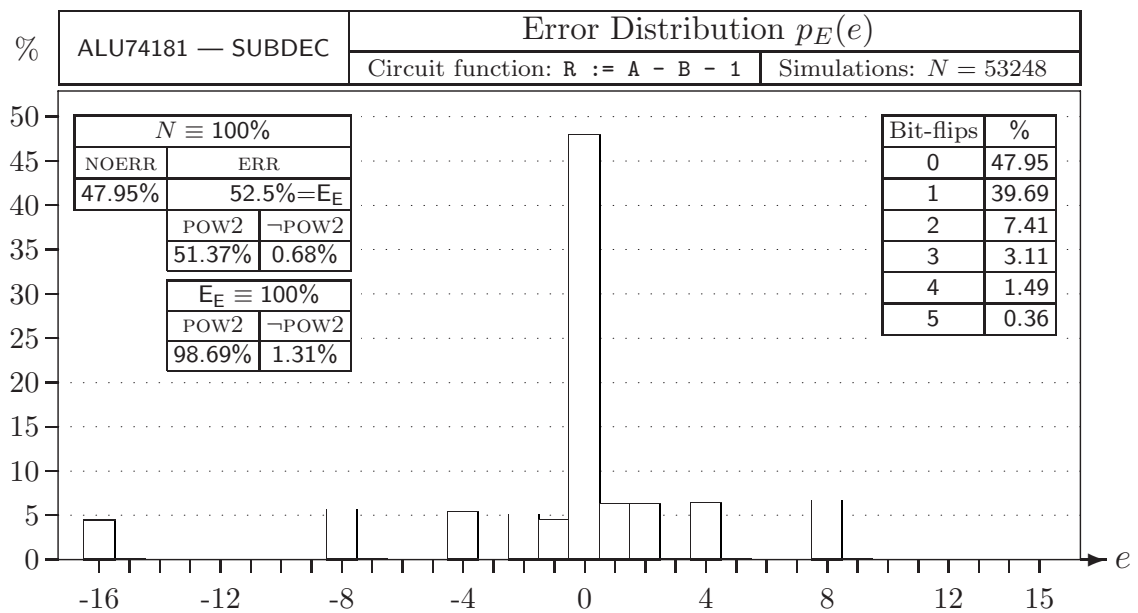


Figure A.23: Single-fault error distribution ALU74181 ($R := A - B - 1$)

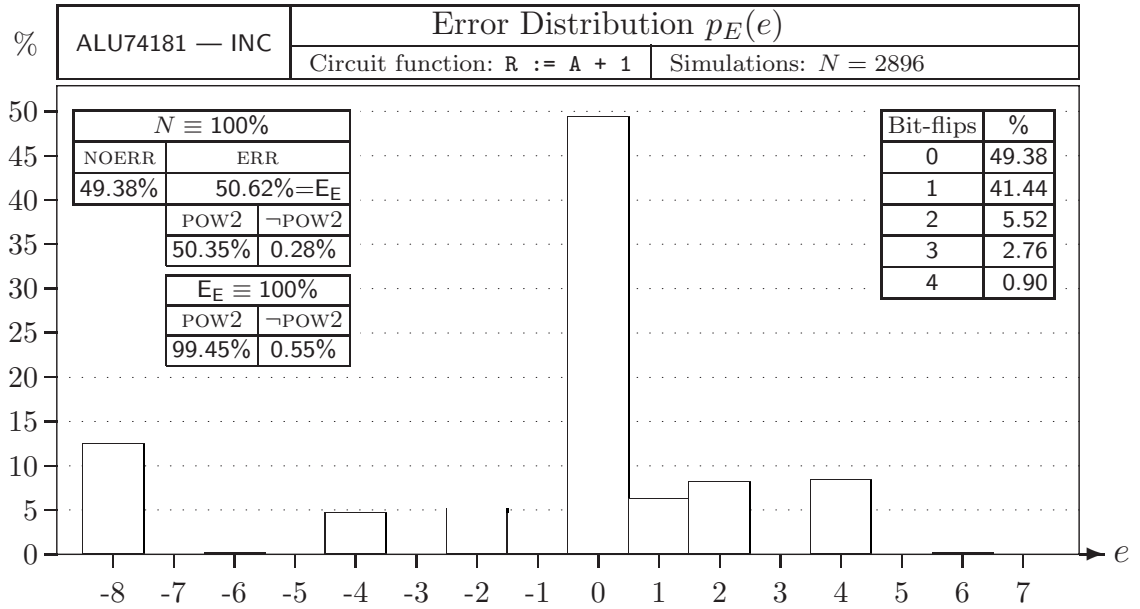


Figure A.24: Single-fault error distribution ALU74181 ($R := A + 1$)

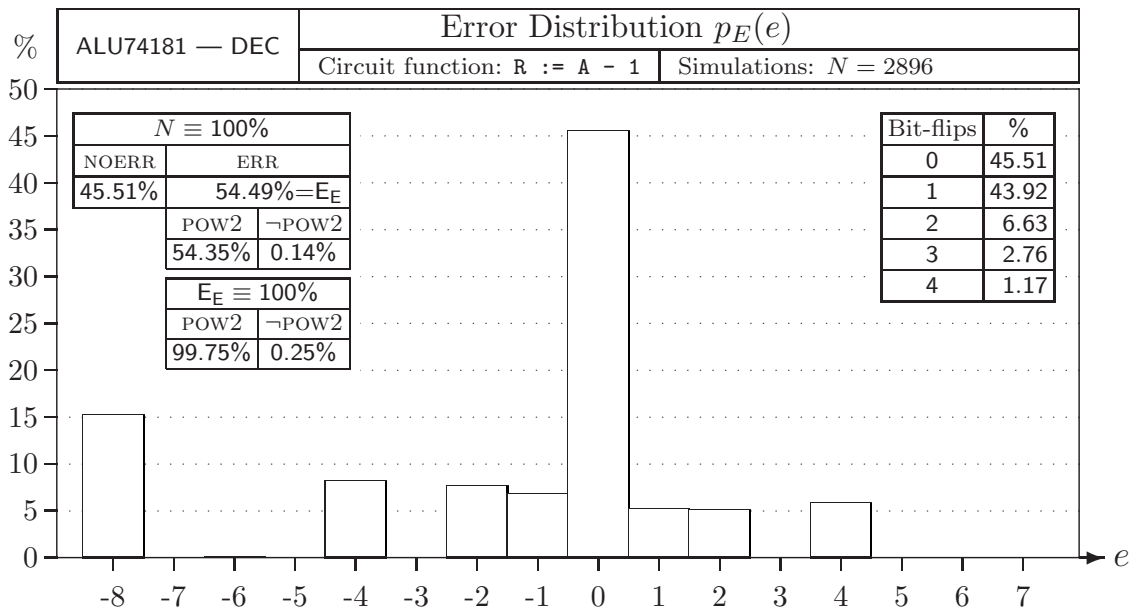


Figure A.25: Single-fault error distribution ALU74181 ($R := A - 1$)

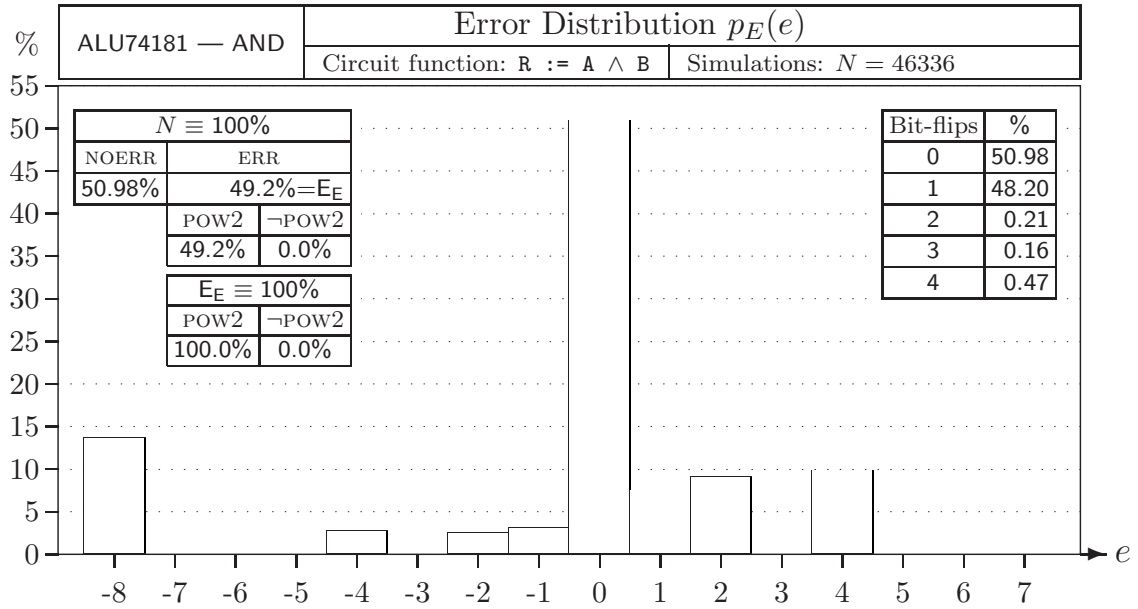


Figure A.26: Single-fault error distribution ALU74181 ($R := A \wedge B$)

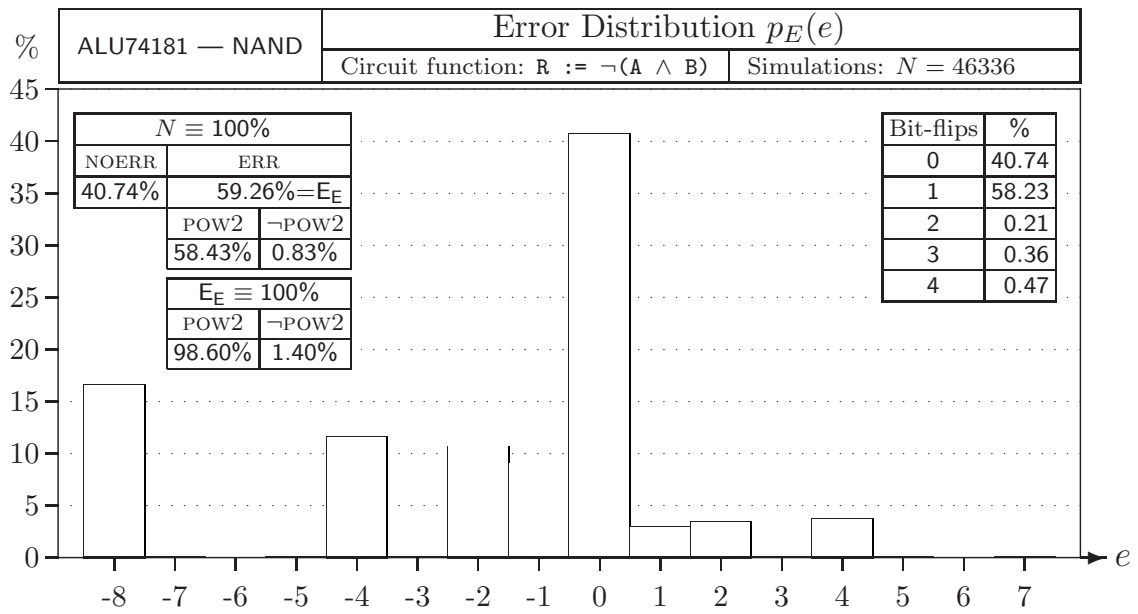


Figure A.27: Single-fault error distribution ALU74181 ($R := \neg(A \wedge B)$)

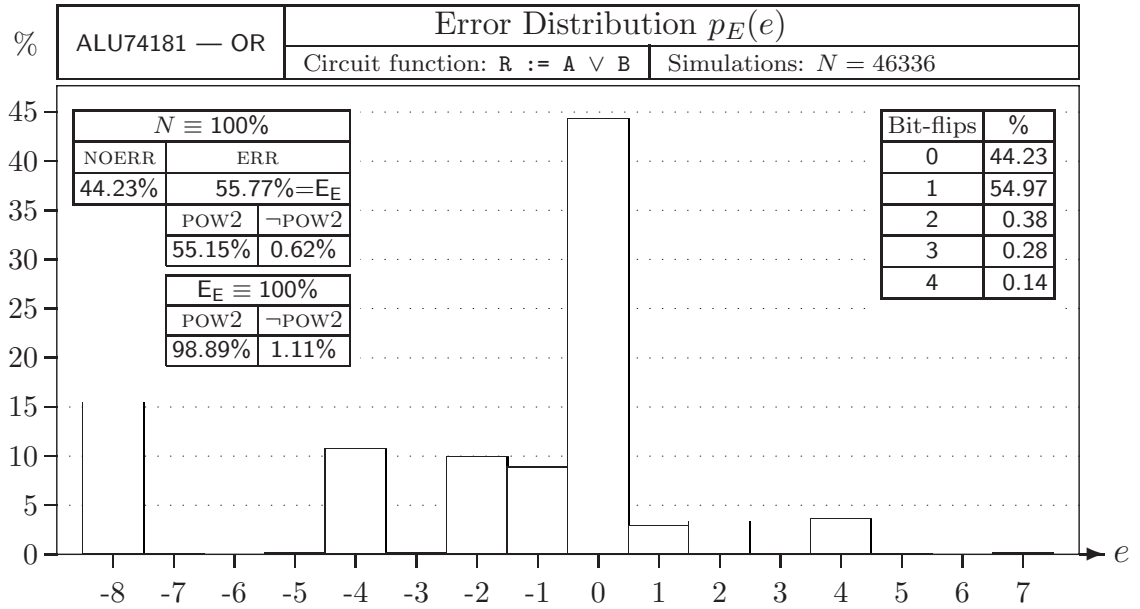


Figure A.28: Single-fault error distribution ALU74181 ($R := A \vee B$)

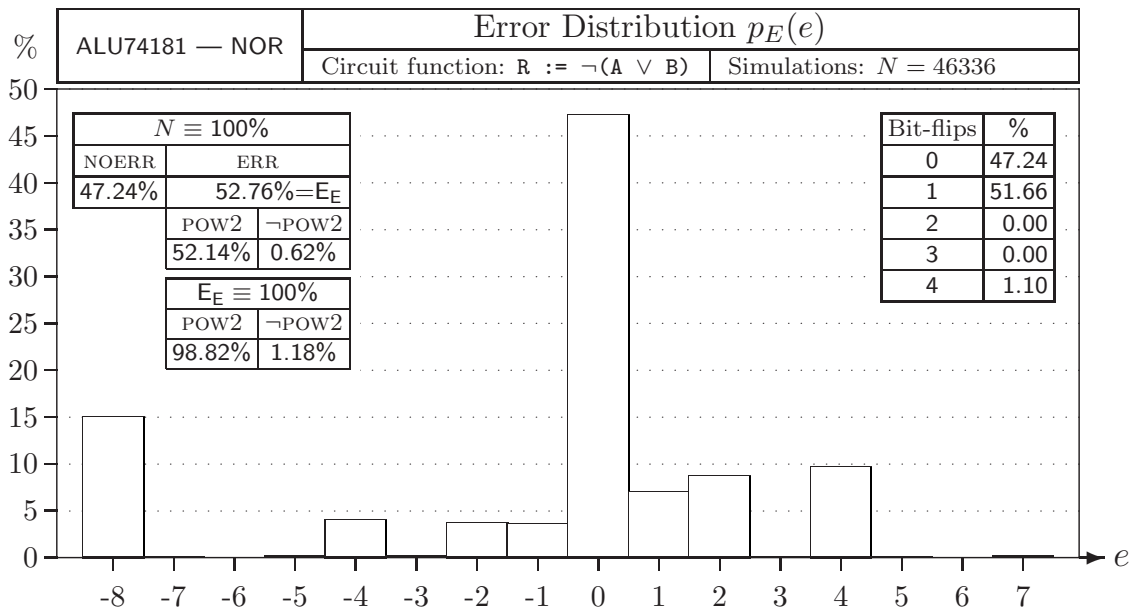


Figure A.29: Single-fault error distribution ALU74181 ($R := \neg(A \vee B)$)

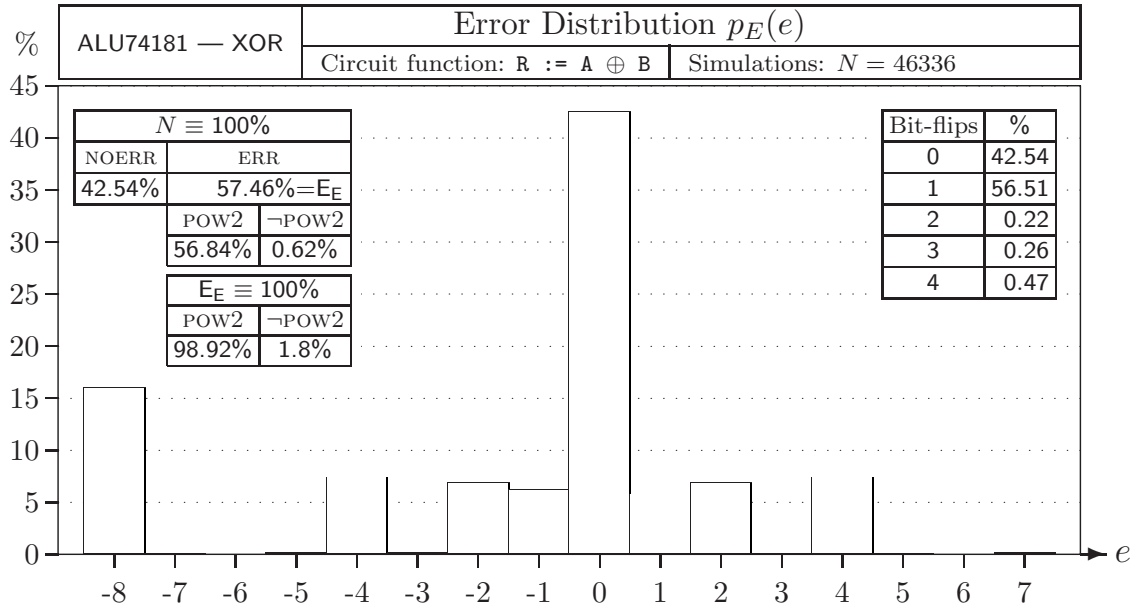


Figure A.30: Single-fault error distribution ALU74181 ($R := A \oplus B$)

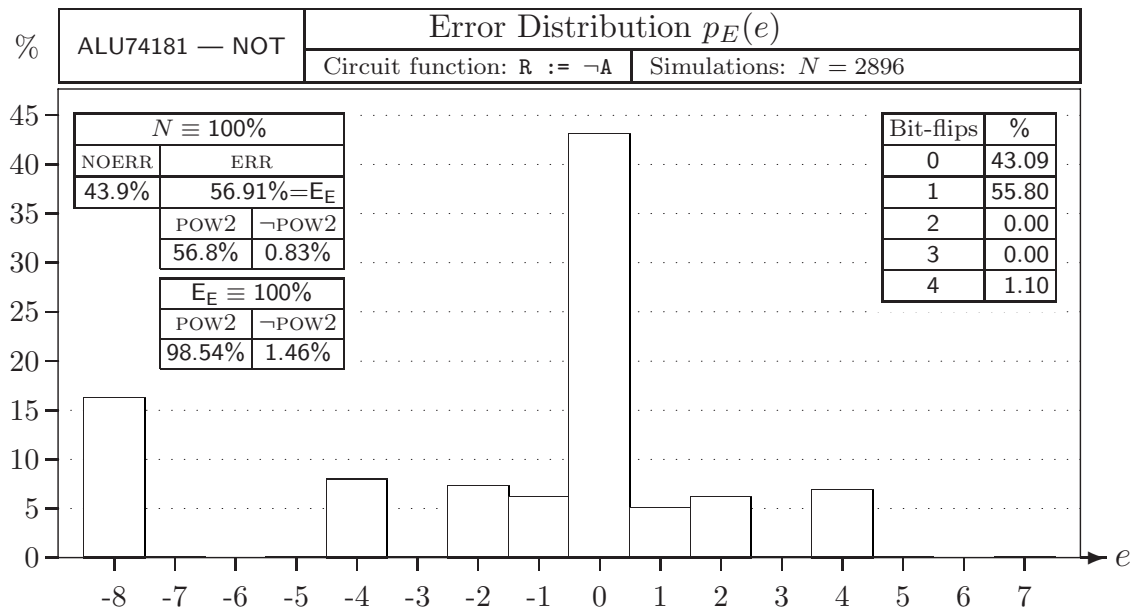


Figure A.31: Single-fault error distribution ALU74181 ($R := \neg A$)

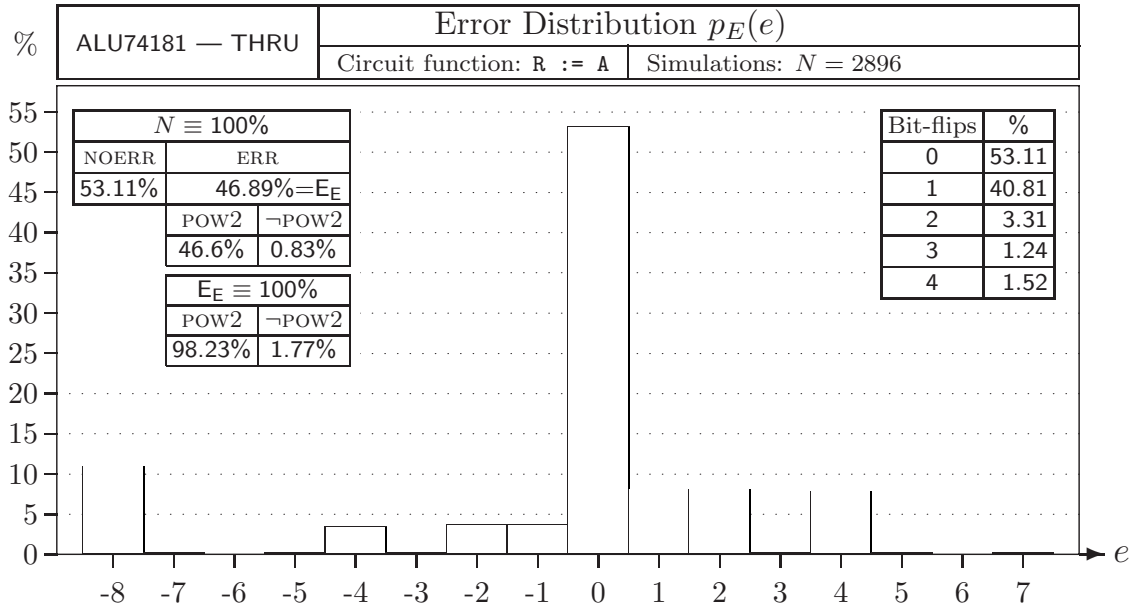


Figure A.32: Single-fault error distribution ALU74181 ($R := A$)

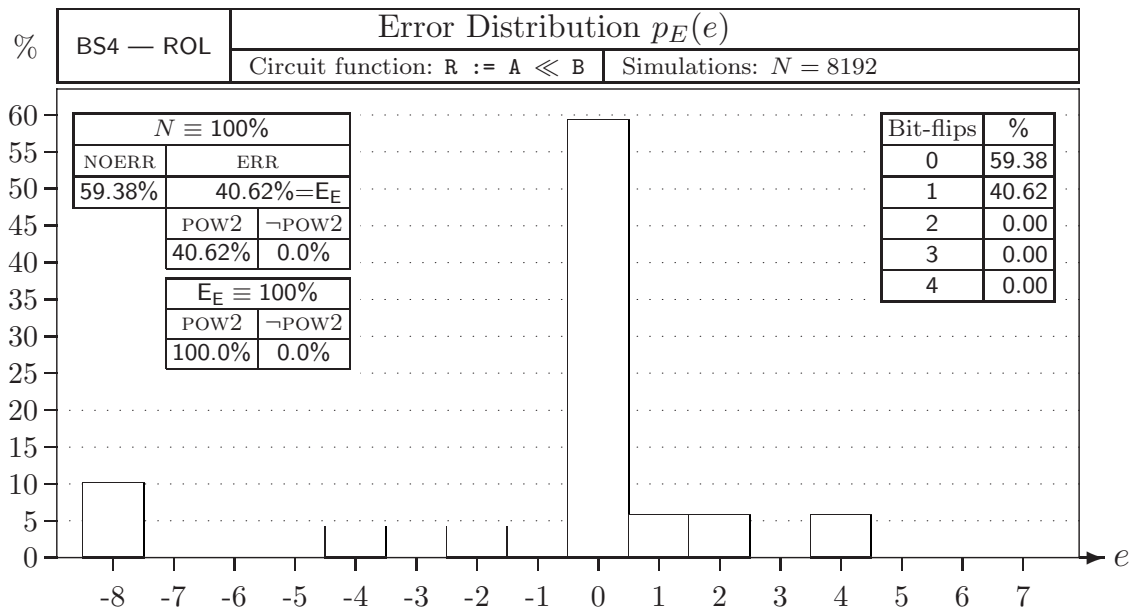


Figure A.33: Single-fault error distribution BS4 ($R := A \ll B$)

A.2 Double-Fault Error Distributions

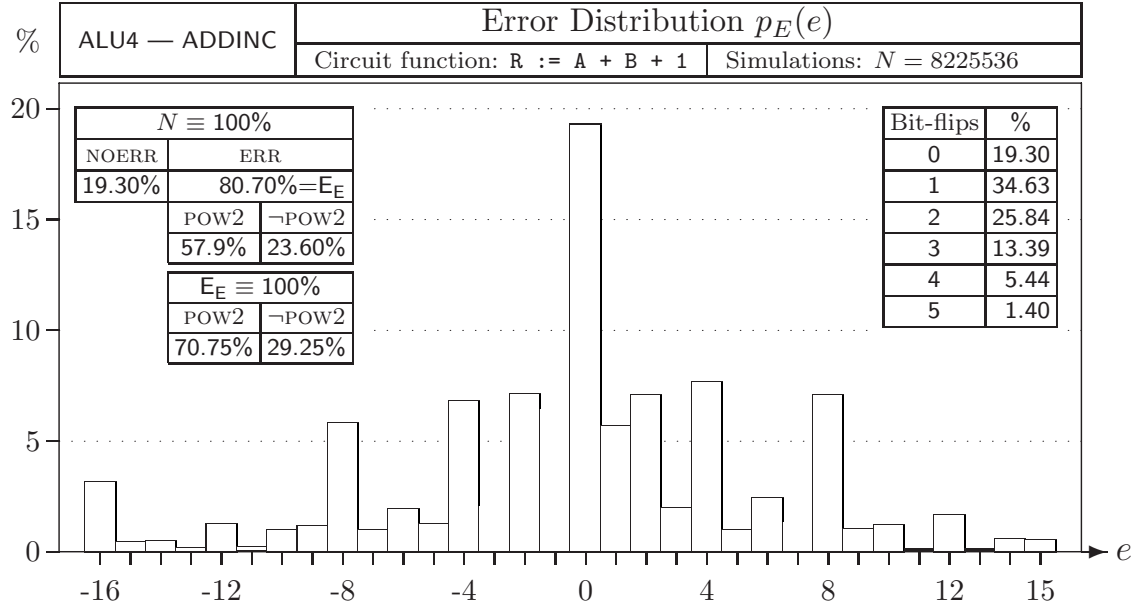


Figure A.34: Double-fault error distribution ALU4 ($R := A + B + 1$)

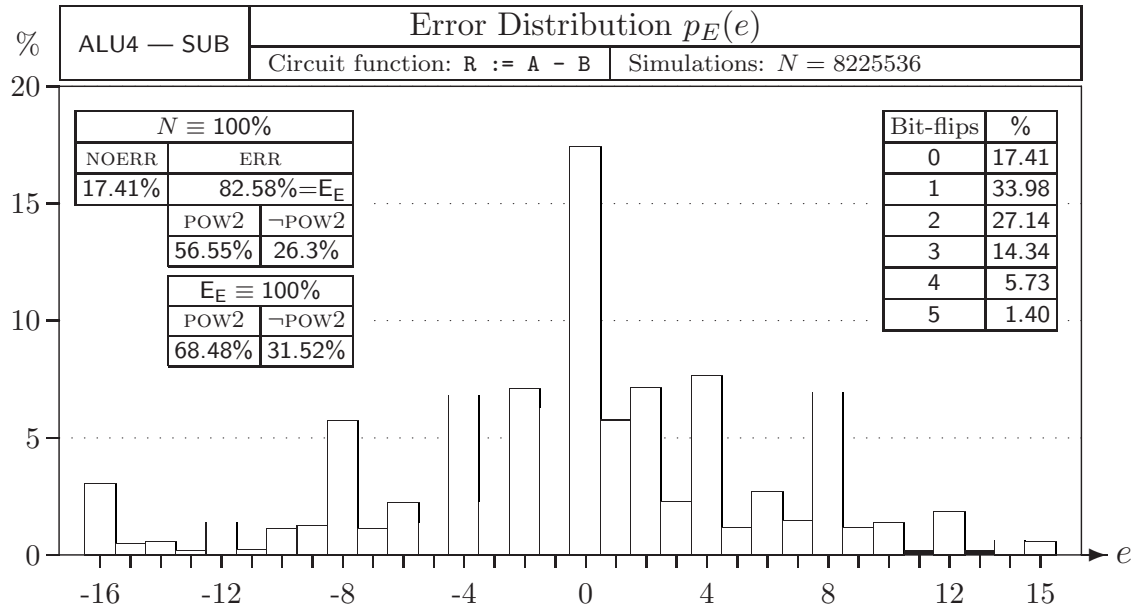


Figure A.35: Double-fault error distribution ALU4 ($R := A - B$)

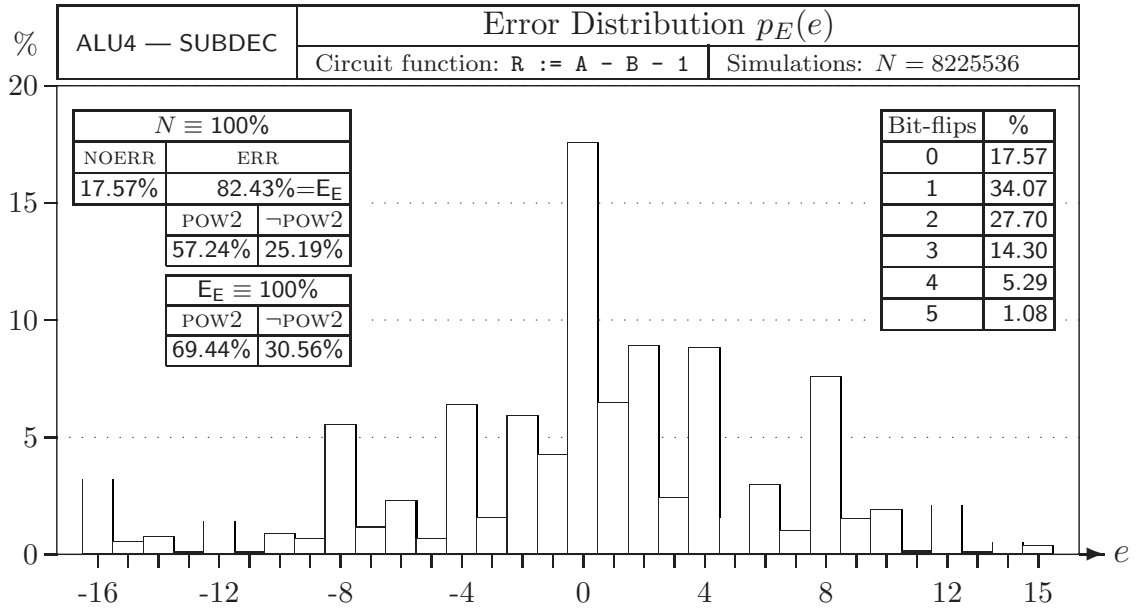


Figure A.36: Double-fault error distribution ALU4 ($R := A - B - 1$)

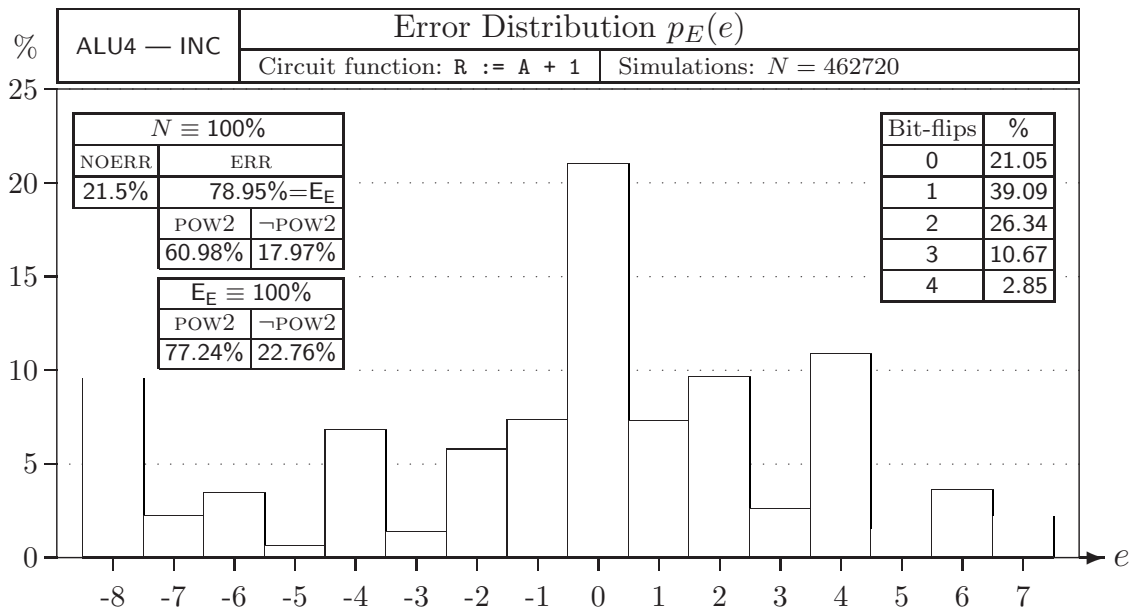


Figure A.37: Double-fault error distribution ALU4 ($R := A + 1$)

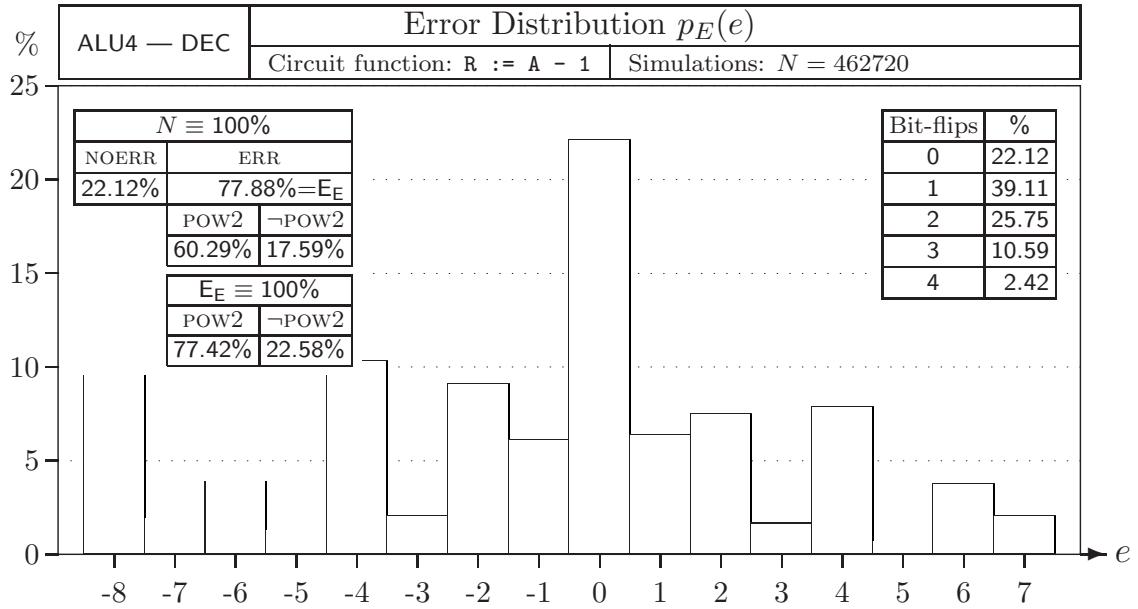


Figure A.38: Double-fault error distribution ALU4 ($R := A - 1$)

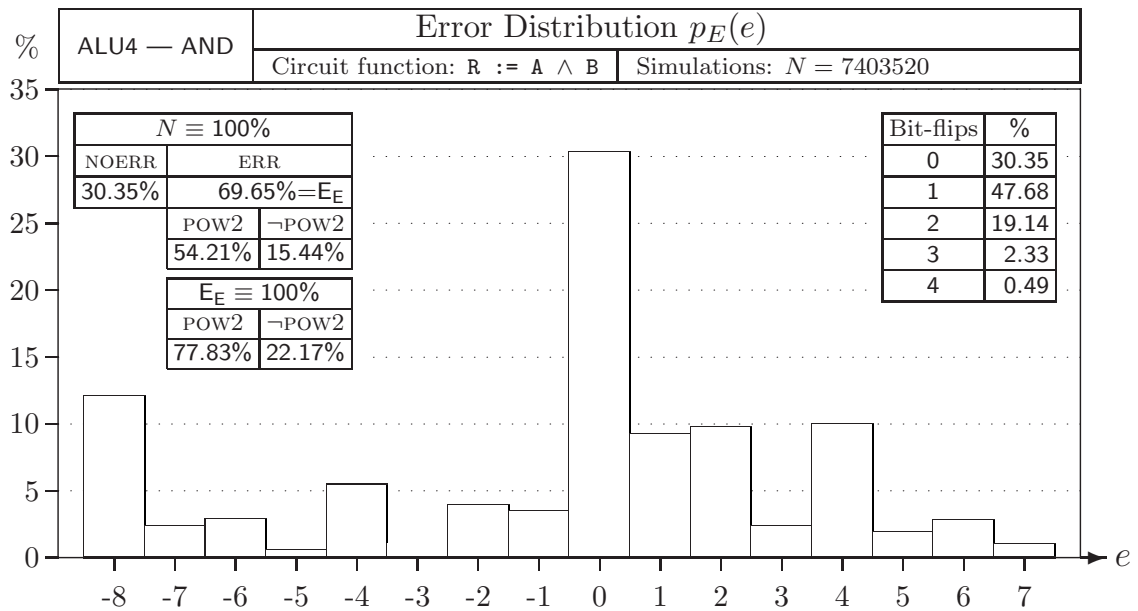


Figure A.39: Double-fault error distribution ALU4 ($R := A \wedge B$)

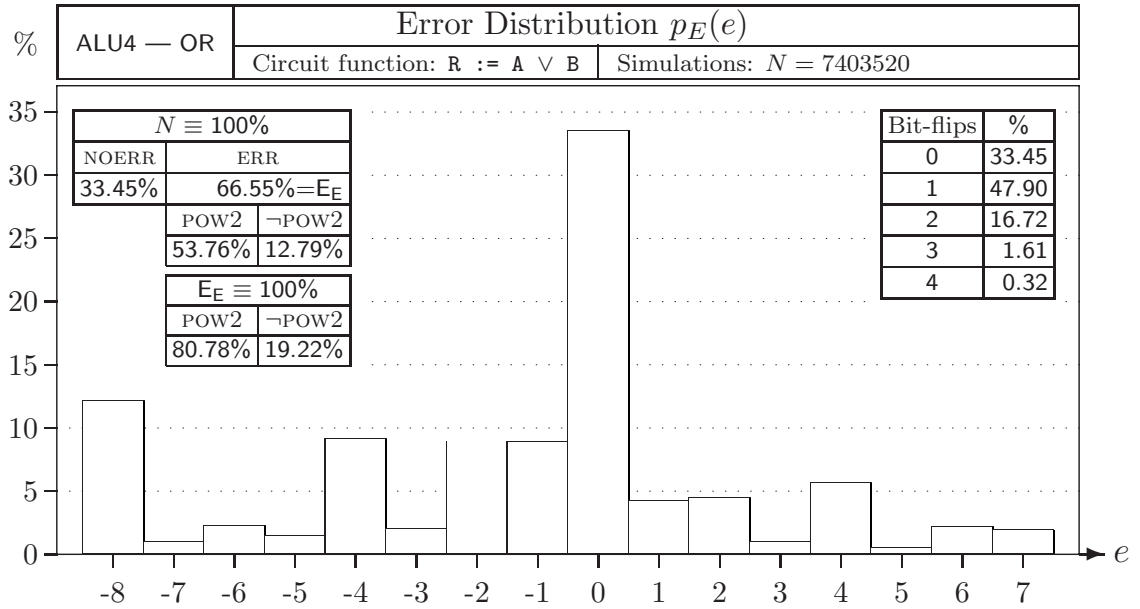


Figure A.40: Double-fault error distribution ALU4 ($R := A \vee B$)

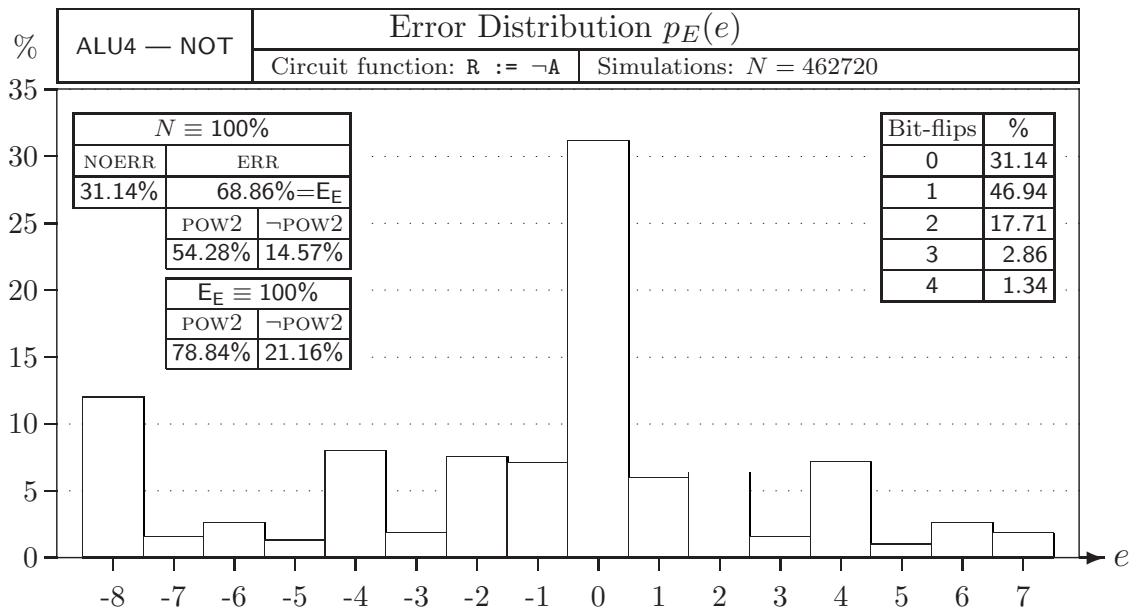


Figure A.41: Double-fault error distribution ALU4 ($R := \neg A$)

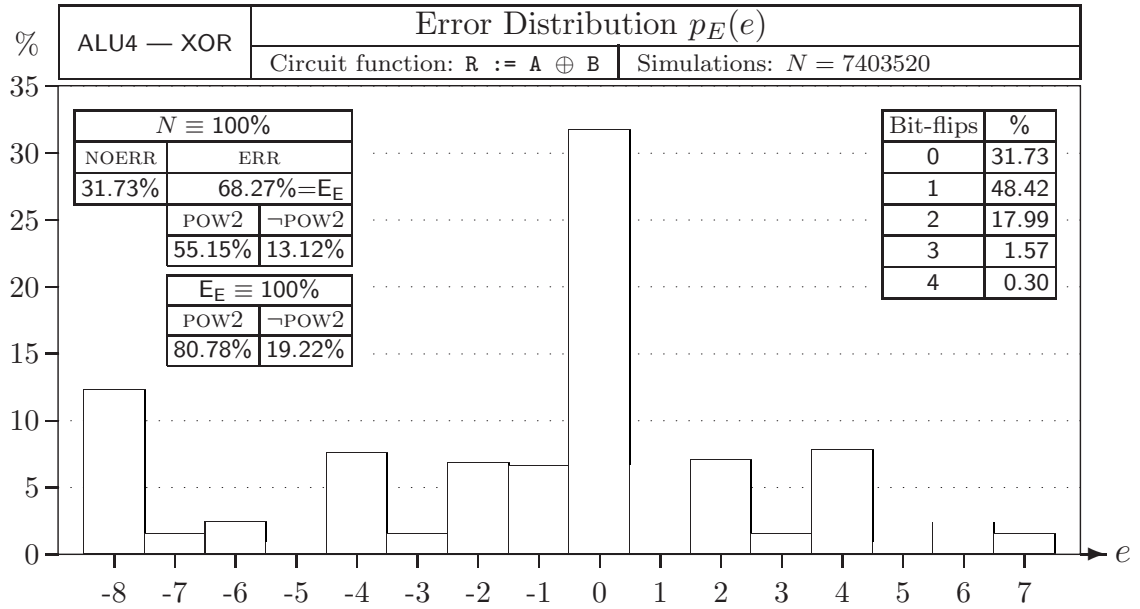


Figure A.42: Double-fault error distribution ALU4 ($R := A \oplus B$)

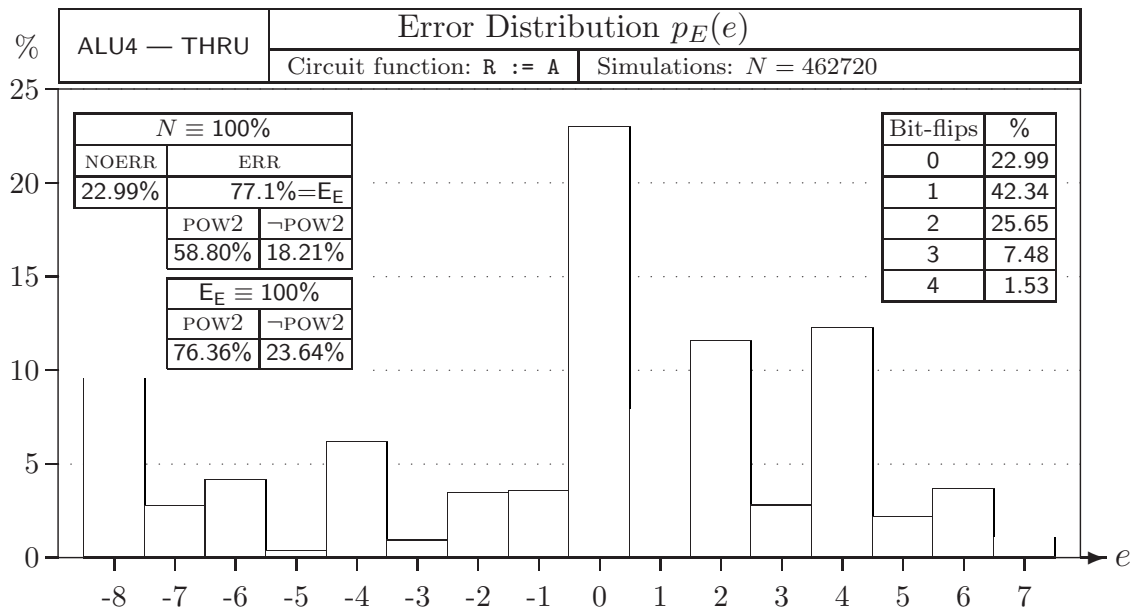


Figure A.43: Double-fault error distribution ALU4 ($R := A$)

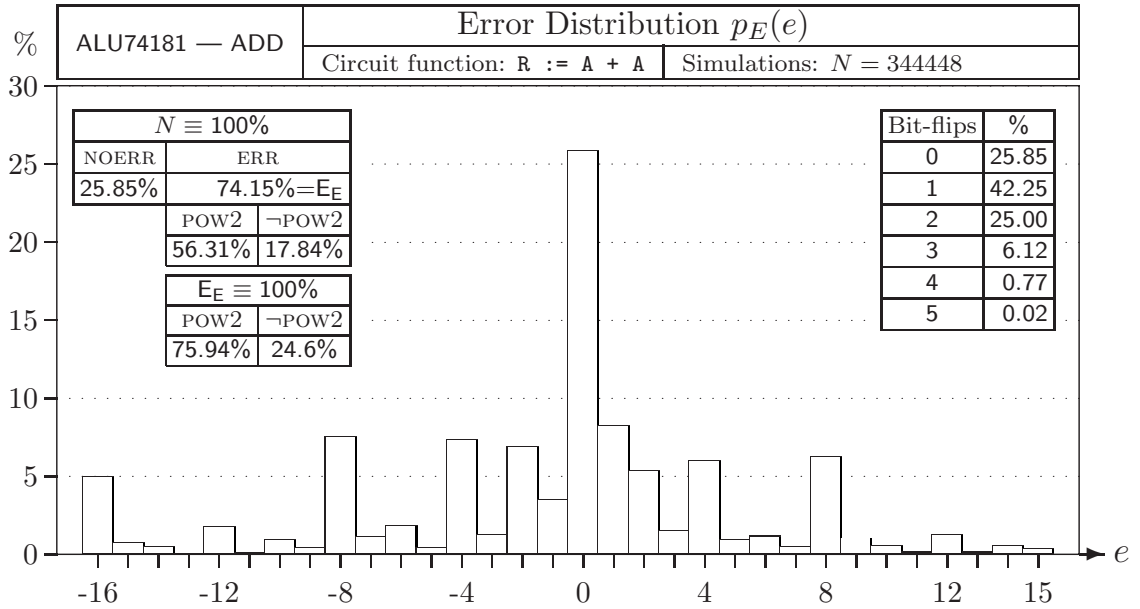


Figure A.44: Double-fault error distribution ALU74181 ($R := A + A$)

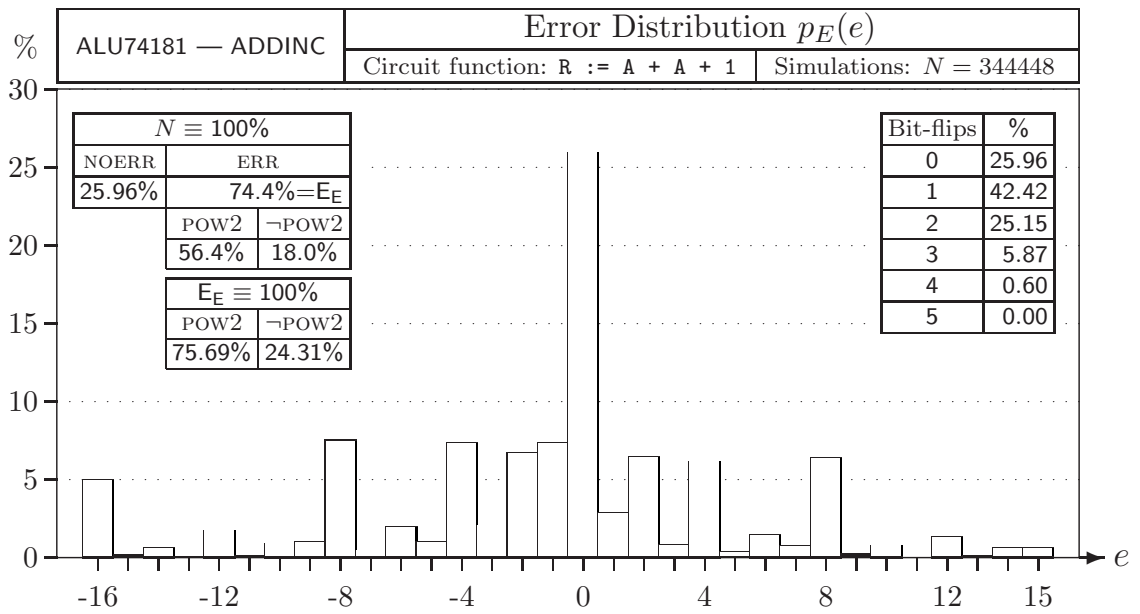


Figure A.45: Double-fault error distribution ALU74181 ($R := A + A + 1$)

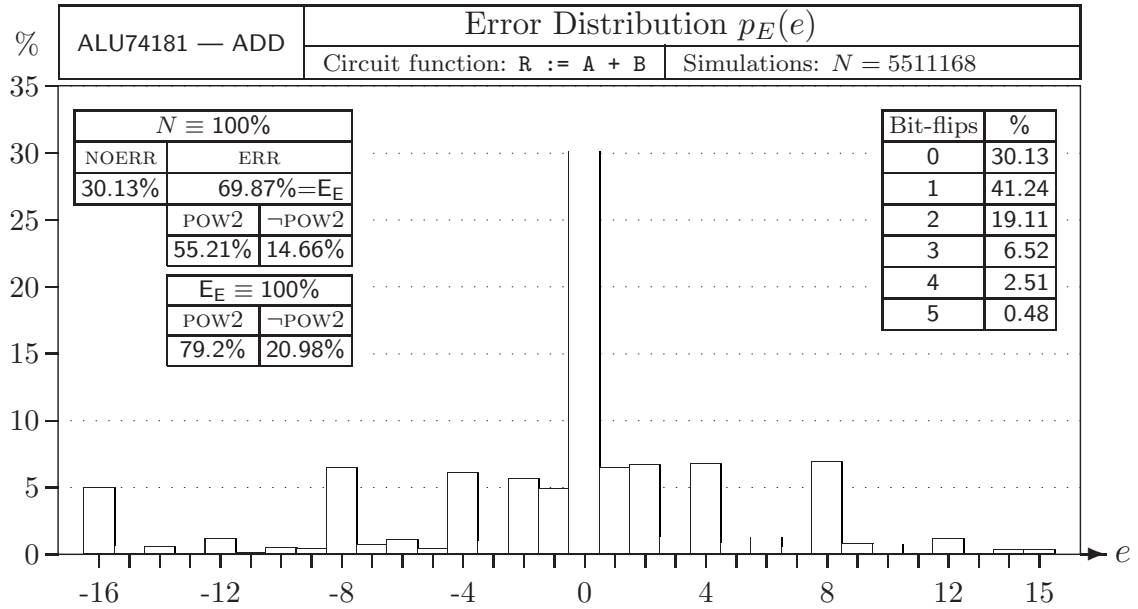


Figure A.46: Double-fault error distribution ALU74181 ($R := A + B$)

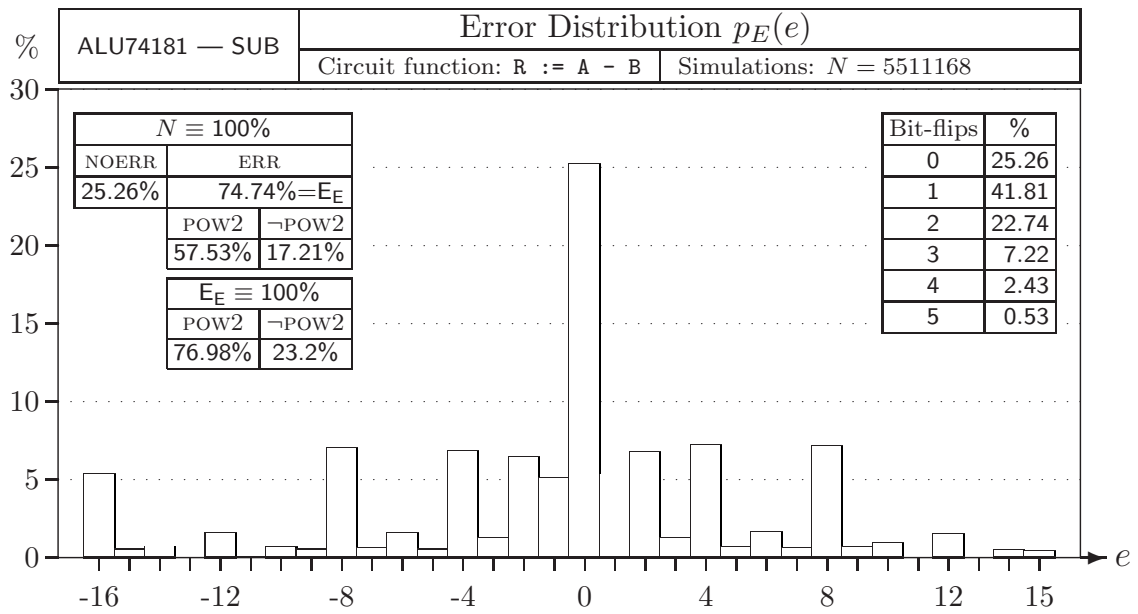


Figure A.47: Double-fault error distribution ALU74181 ($R := A - B$)

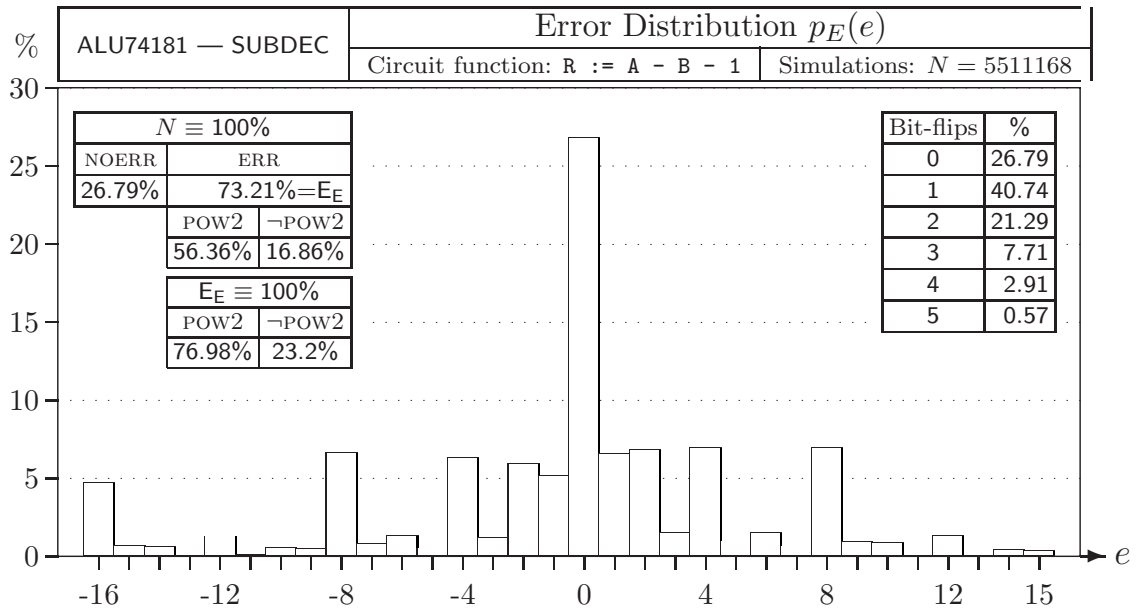


Figure A.48: Double-fault error distribution ALU74181 ($R := A - B - 1$)

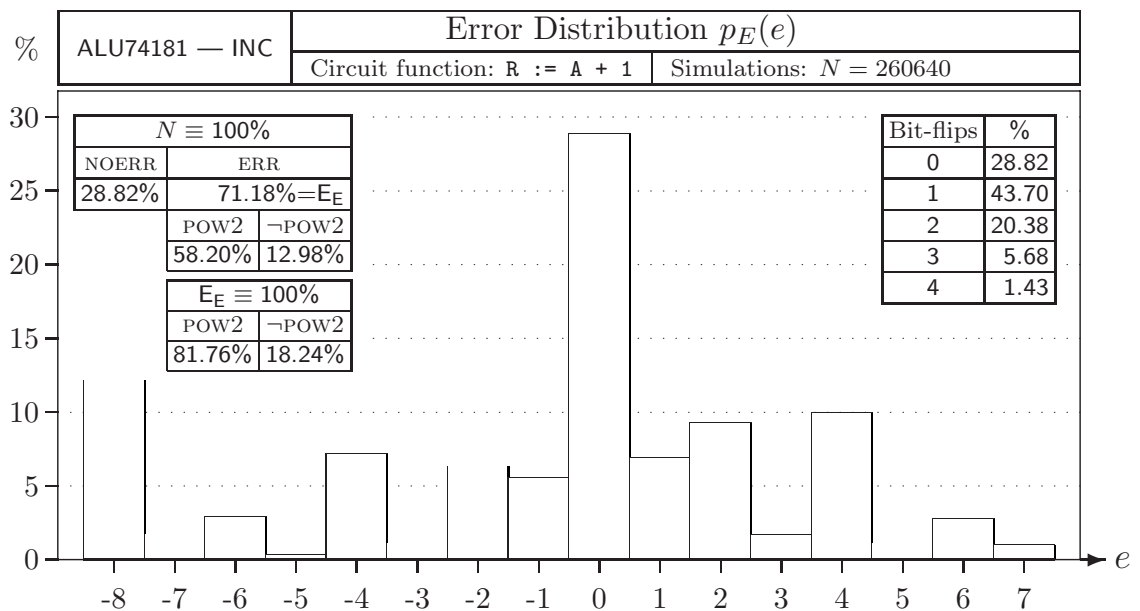


Figure A.49: Double-fault error distribution ALU74181 ($R := A + 1$)

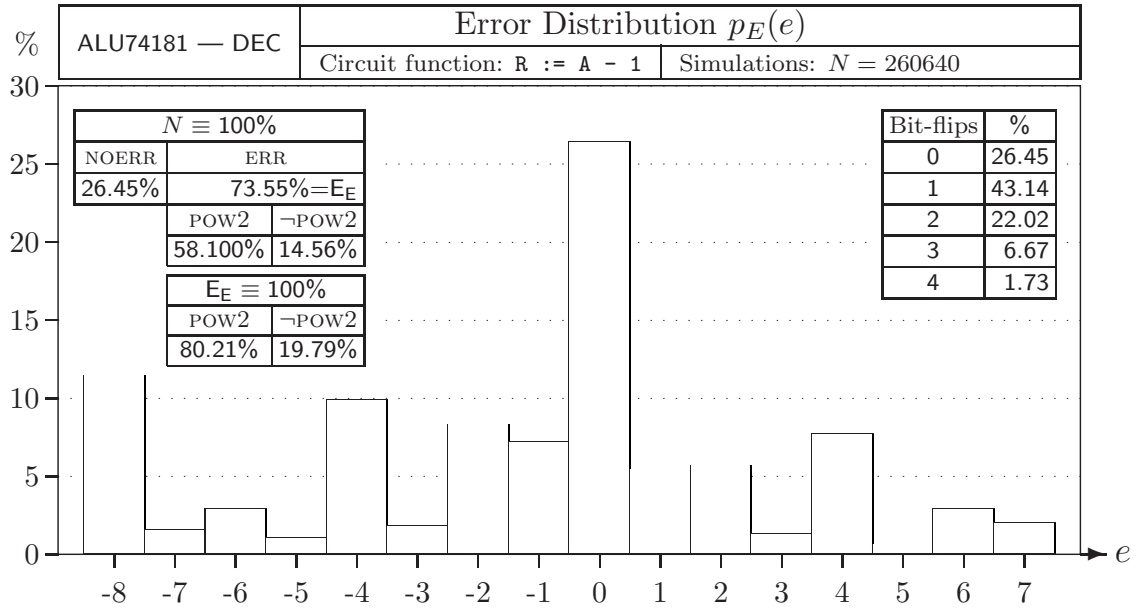


Figure A.50: Double-fault error distribution ALU74181 ($R := A - 1$)

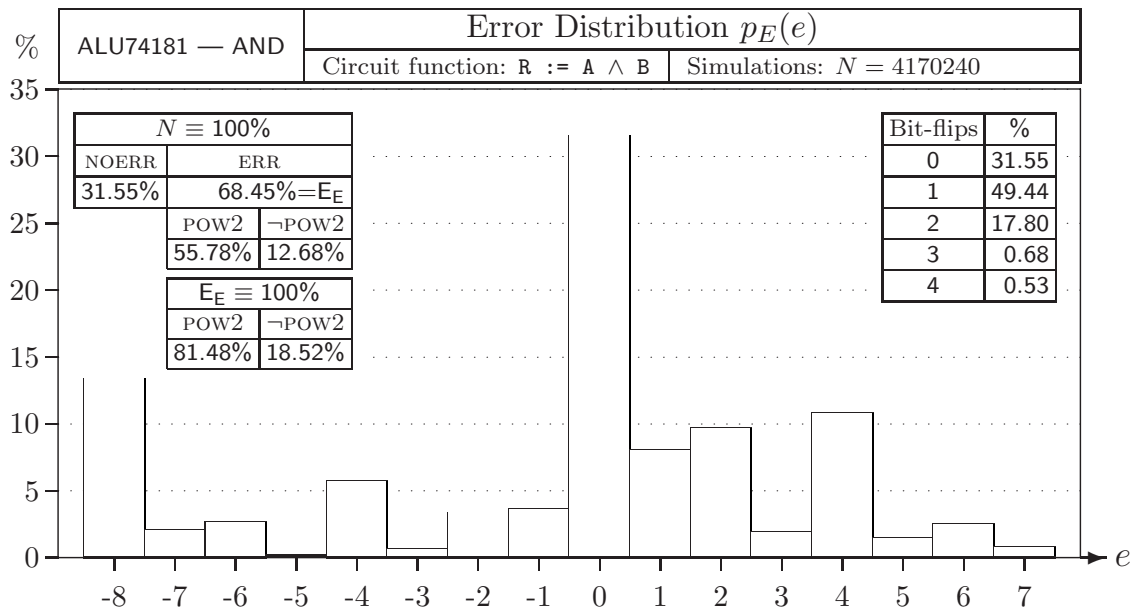


Figure A.51: Double-fault error distribution ALU74181 ($R := A \wedge B$)

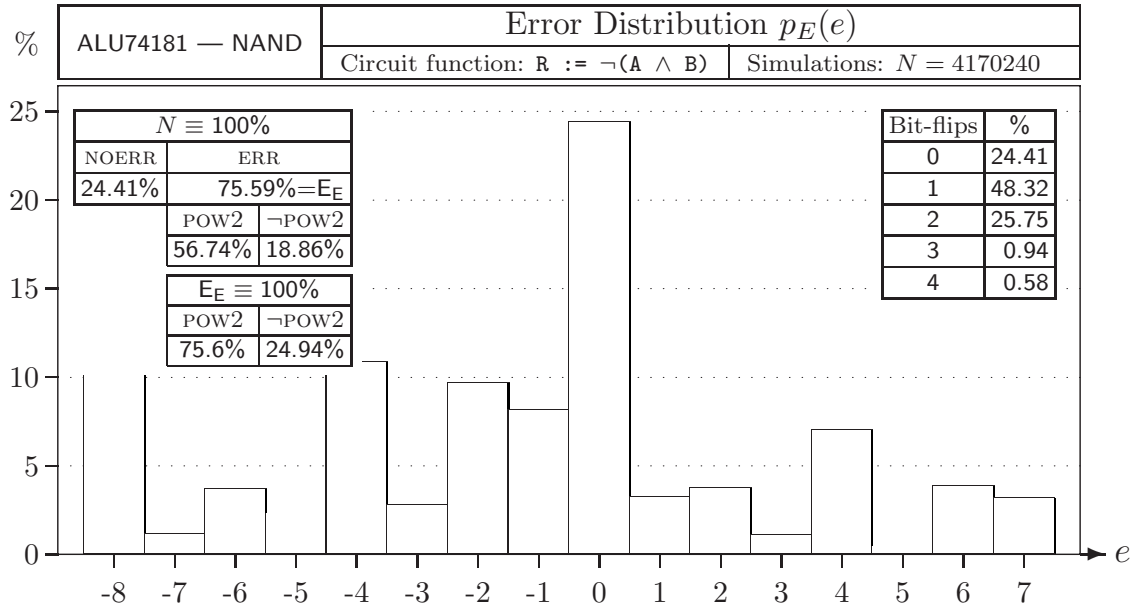


Figure A.52: Double-fault error distribution ALU74181 ($R := \neg(A \wedge B)$)

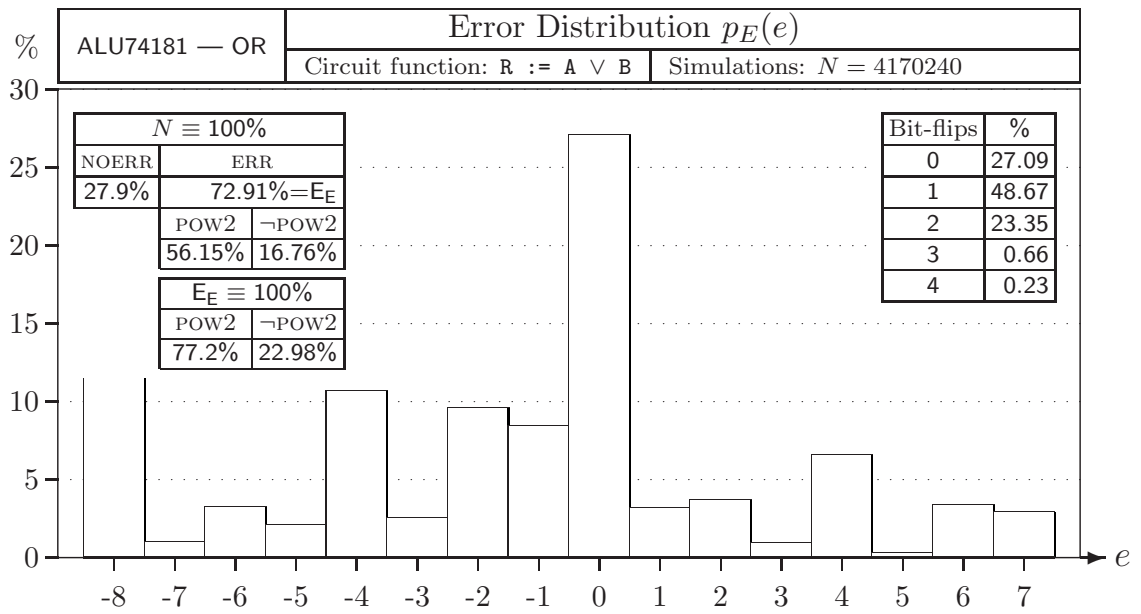


Figure A.53: Double-fault error distribution ALU74181 ($R := A \vee B$)

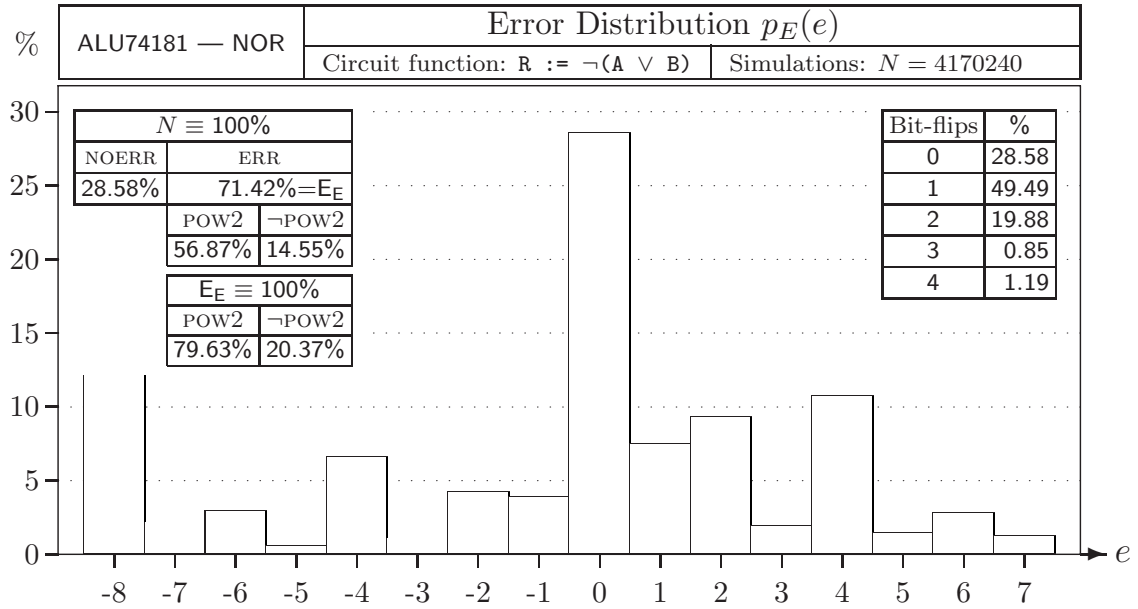


Figure A.54: Double-fault error distribution ALU74181 ($R := \neg(A \vee B)$)

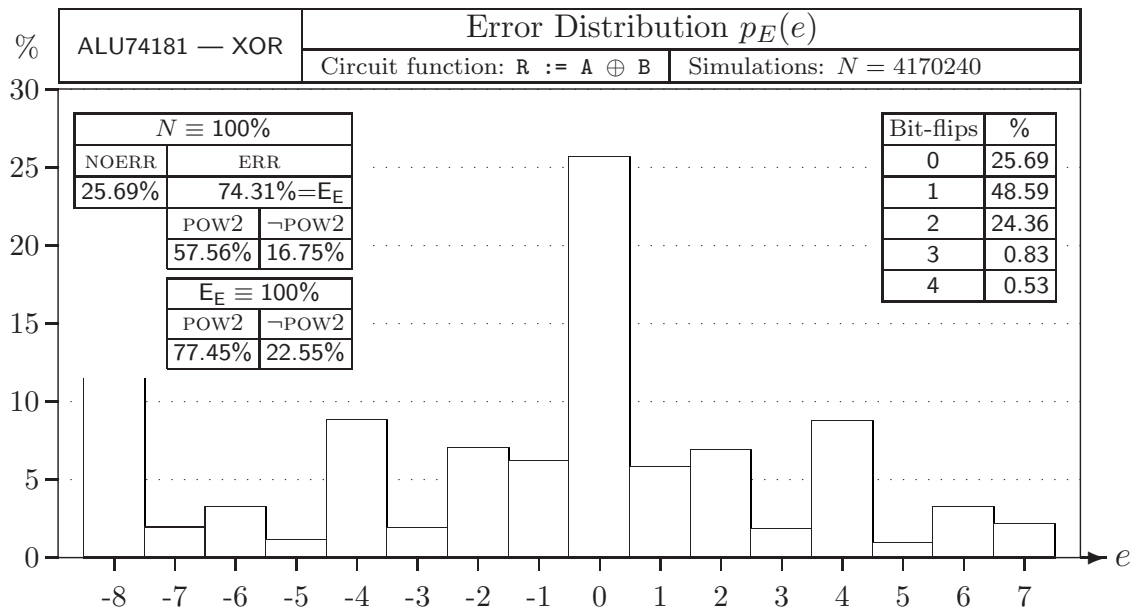


Figure A.55: Double-fault error distribution ALU74181 ($R := A \oplus B$)

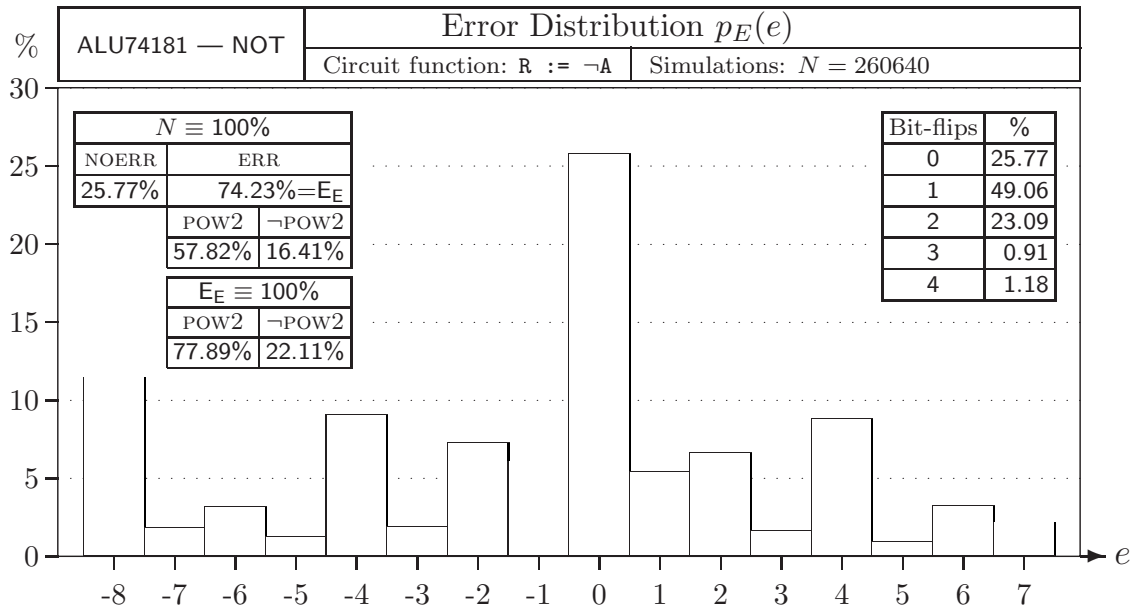


Figure A.56: Double-fault error distribution ALU74181 ($R := \neg A$)

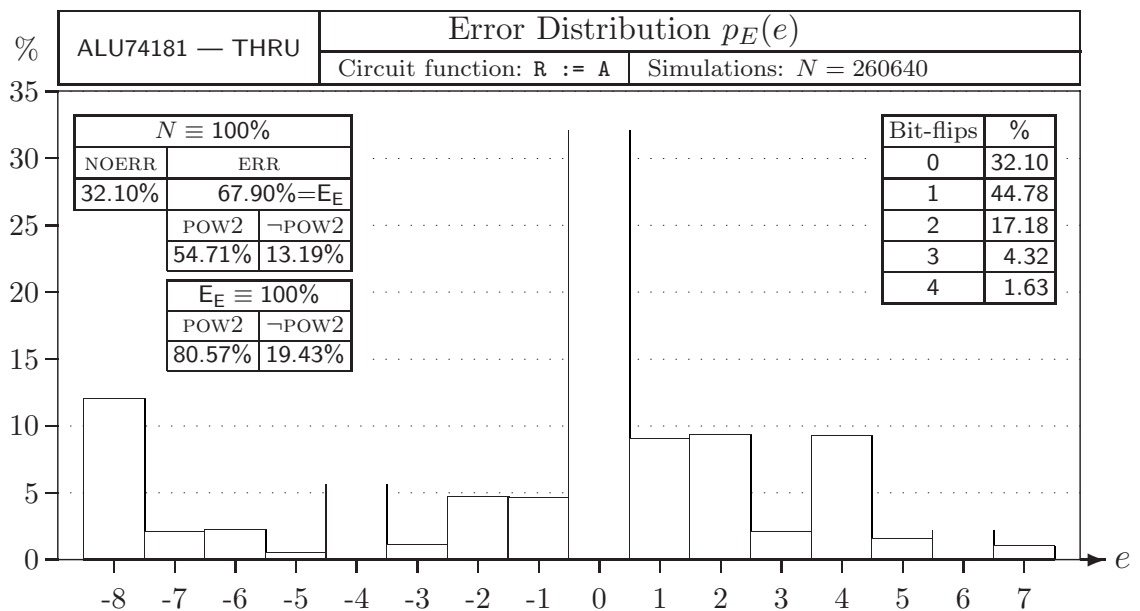


Figure A.57: Double-fault error distribution ALU74181 ($R := A$)

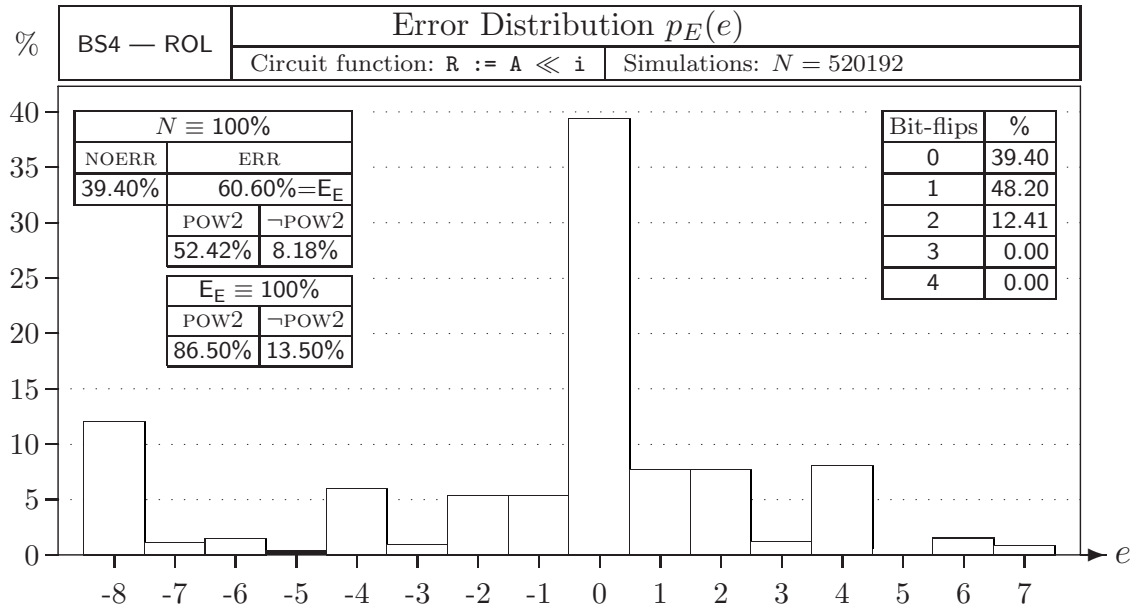


Figure A.58: Double-fault error distribution BS4 ($R := A \ll B$)

References

- [Ab79] J.A. **Abraham** and S.M. Thatte. Fault Coverage of Test Programs for a Microprocessor. In *Proc. IEEE International Test Conference, Cherry Hill, New Jersey, Oct., 1979*, pages 18–22. IEEE, 1979.
- [Ar89] Jean **Arlat**, Yves Crouzet, and Jean-Claude Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *Proc. FTCS-19, Chicago, Jun., 1989*, pages 348–355. IEEE, 1989.
- [Ar90] Jean **Arlat** et al. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.
- [As98] Hussain Al-**Asaad**, Brian T. Murray, and John P. Hayes. Online BIST for embedded systems. *IEEE Design & Test of Computers*, 15(4):17–24, Oct–Dec 1998.
- [Av97] Algirdas **Avizienis**. Torward systematic design of fault-tolerant systems. *IEEE Computer*, pages 51–58, Apr 1997.
- [Ba90] James H. **Barton** et al. Fault injection experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, Apr 1990.
- [Boe98] Eberhard **Böhl**, Thomas Lindenkreuz, and Matthias Meerwein. On-chip IDDQ testing in the AE11 fail-stop controller. *IEEE Design & Test of Computers*, 15(4):57–65, Oct–Dec 1998.
- [Ca98] João **Carreira**, Henrique Madeira, and João Gabriel Silva. Xception: Software fault injection and monitoring in processor functional units. In Ravishankar K. Iyer et al., editors, *Dependable Computing for Critical Applications*, volume 10 of *Dependable Computing and Fault-Tolerant Systems*, pages 245–265. IEEE, 1998. ISBN 0-8186-7803-8.

- [Ca99] João **Carreira**, Henrique Madeira, and João Gabriel Silva. Fault injection spot-check computer system dependability. *IEEE Spectrum*, 36(8):50–55, Aug 1999.
- [Ch89] Ram **Chillarege** and Nicholas S. Bowen. Understanding large system failures – a fault injection experiment. In *Proc. FTCS-19, Chicago, Jun 1989*, pages 356–363. IEEE, 1989.
- [Ch91] Gwan S. **Choi** et al. A fault behavior model for an avionic microprocessor: A case study. In *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, pages 177–195. Springer, 1991.
- [Ch92] Gwan S. **Choi** and Ravishankar K. Iyer. FOCUS: An experimental environment for fault sensitivity analysis. *IEEE Transactions on Computers*, 41(12):1515–1526, Dec 1992.
- [Cig04] Cigital Labs, Dulles, VA, USA. Web page, 2004. www.cigitallabs.com/resources/definitions/fault_tolerance.html.
- [Cl95] Jeffrey A. **Clark** and Dhiraj K. Pradhan. Fault injection, a method for validating computer-system dependability. *IEEE Computer*, 28(6):47–56, Jun 1995.
- [Co03] Patricia **Costa** and Ioana Rus. Characterizing software dependability from multiple stakeholder’s perspective. *Software Tech News*, 6(2), Dec 2003. DoD online newsletter: www.softwaretechnews.com/stn6-2/dependability.html.
- [Cu99] Michel **Cukier**, David Powell, and Jean Arlat. Coverage estimation methods for stratified fault-injection. *IEEE Transactions on Computers*, 48(7):707–723, Jul 1999.
- [Cz90] Edward W. **Czeck** and Daniel P. Siewiorek. Effects of transient gate-level faults on program behavior. In *Proc. FTCS-20*, pages 236–243. IEEE, 1990.
- [Du88] P. **Duba** and R.K. Iyer. Transient fault-behavior in a microprocessor – a case study. In *Proc. ICCD: VLSI in Computers & Processors, Rye Brook, NY, Oct 1988*, pages 272–276. IEEE, 1988.

- [Fo96] Peter **Folkesson**. Experimental validation of a fault-tolerant system using physical fault-injection. Technical report, Chalmers University of Technology, Göteborg (Gothemburg), Sweden, 1996. Report no. 239L.
- [Fo99] Peter **Folkesson**. *Assessment and Comparison of Physical Fault Injection Techniques*. PhD thesis, Chalmers University of Technology, Göteborg (Gothemburg), Sweden, 1999.
- [Fr98] Stefan **Freinatis** and Axel Hunger. Using a register model fault simulator to assess the fault detection coverage of fault tolerant embedded software for automotive systems. In *Proc. ESS'98, Nottingham*, pages 490–494. SCS, 1998. ISBN 1-5655-147-8.
- [Fr99a] Stefan **Freinatis** and Axel Hunger. Assessing the fault tolerance of embedded software through application of machine instruction mutations. In *Proc. Conference on Modeling and Simulation (MS'99), Philadelphia*, pages 541–549. IASTED, 1999. ISBN 0-88986-247-8.
- [Fr99b] Stefan **Freinatis** and Axel Hunger. Bewertung sicherheitskritischer Steuerungssoftware durch Simulation fehlerhaften Prozessorverhaltens. In *11. ITG Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen, Potsdam, Feb./Mar. 1999*, pages 58–61. BTU Cottbus and Universität Potsdam (Germany), 1999. ISBN 3-9806494-1-5.
- [Ga01] Rudy **Garcia**. Rethink fault models for submicron-IC test. *Test & Measurement World*, Oct 2001. Reed Business Information, Waltham, MA, USA.
- [Ha95] Sejungjae **Han**, Kang G. Shin, and Harold A. Rosenberg. DOCTOR: An integrated software fault injection environment for distributed real-time systems. In *Proc. International Computer Performance and Dependability Symposium, Erlangen, Apr 1995*, pages 204–213. IEEE, 1995.
- [Hg82] Axel **Hunger**. *Neues Verfahren zum Selbsttest von Mikroprozessoren*. PhD thesis, Rheinisch-Westfälische Technische Hochschule Aachen, Aachen, Germany, 1982. ISBN 3-88585-078-8.

- [Hg89] Axel **Hunger**. Reliability measurement of microprocessors based on functional testing. *Microelectronics Reliability*, 29(3):349–355, 1989. Pergamon Press, ISSN 0026-2714.
- [Hi01] Martin **Hiller**, Arshad Jhumka, and Neeraj Suri. An approach for analysing the propagation of data errors in software. In *Proc. DSN 2001, Göteborg (Gothemburg), Jul 2001*, pages 161–172. IEEE, 2001.
- [IEEE90] Institute of Electrical and Electronics Engineers. IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries, 1990.
- [IFIP88] International Federation for Information Processing – Working Group 10.4. Aim of the working group, 1988.
www.dependability.org/wg10.4.
- [Iy86] Ravishankar K. **Iyer** and David J. Rossetti. A measurement-based model for workload dependence of CPU errors. *IEEE Transactions on Computers*, 35(6):511–519, 1986.
- [Jo94] Rolf **Johansson**. On single event upset error manifestation. In Klaus Ehtle, Dieter K. Hammer, and David Powell, editors, *Proc. 1st European Dependable Computing Conference*, volume 852 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 1994. ISBN 3-540-58426-9.
- [Ka92] Ghani A. **Kanawati**, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A tool for the validation of system dependability properties. In *Proc. FTCS-22, Boston 1992*, pages 336–344. IEEE, Jul 1992.
- [Ka94] Johan **Karlsson** et al. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14(1):8–23, Feb 1994.
- [Ka95] Ghani A. **Kanawati**, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Reliability*, 44(2):248–260, Feb 1995.

- [Kaw95] P. **Kawalek** and D.G. Wastell. The development of a process modelling method. In *Proc. 2nd BCS Information Systems Methodology Conference, Edinburgh 1995*. British Computer Society/Heriot-Watt University, 1995.
- [Le95] Nancy G. **Leveson**. *Safeware: System Safety and Computers*. Addison Wesley, 1995. ISBN 0-201-11972-2.
- [Lo95] Tomislav **Lovrić**. Processor fault simulation with ProFI. In *Proc. 7th European Simulation Symposium, Erlangen-Nürnberg 1995*, pages 353–357, 1995.
- [Ma90] Henrique **Madeira**, Gonçalo Quadros, and João Gabriel Silva. Experimental evaluation of a set of simple error detection mechanisms. In *Proc. Euromicro Symposium on Microprocessing and Microprogramming (The EUROMICRO Journal), Amsterdam 1990*, volume 30, pages 513–520. North Holland, August 1990.
- [Mi95] Ghassem **Miremadi** and Jan Torin. Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection. *IEEE Transactions on Reliability*, 44(3):441–454, Sep 1995.
- [MW99] Klaus Brieter. Skandal um VW-Airbag. *ADAC Motorwelt*, (2):22, Feb 1999. ADAC (German Automobile Club), Munich, Germany.
- [Nu97] Bashar **Nuseibeh**. Ariane 5: Who dunnit? *IEEE Software*, pages 15–16, May/June 1997.
- [Po95] David **Powell**, Eliane Martins, Jean Arlat, and Yves Crouzet. Estimators for fault-tolerance coverage evaluation. *IEEE Transactions on Computers*, 44(2):261–274, Feb 1995.
- [Re99] M. **Rebaudengo** and M. Sonza Reorda. Evaluating the fault tolerance capabilities of embedded systems via BDM. In *Proc. 17th IEEE VLSI Test Symposium (VTS'99), San Diego, Apr 1999*, pages 452–457. IEEE, 1999.
- [Ri94] Marcus **Rimén**, Joakim Ohlsson, and Jan Torin. On microprocessor error behavior modeling. In *Proc. FTCS-24*, pages 76–85. IEEE, 1994.

- [Se68] Frederik. F. **Sellers**, Mu-Yue Hsiao, and Leroy W. Bearnson. *Error Detecting Logic for Digital Computers*. McGraw-Hill, 1968.
- [Sl98] Robert **Slater**. Fault injection, 1998. www-2.cs.cmu.edu/~koopman/des_s99/fault_injection.
- [So86] Janusz **Sosnowski**. Transient fault effects in microprocessor controllers. In *Reliability Technology – Theory & Applications*, pages 239–248. Elsevier Science Publishers B.V. (North-Holland), 1986.
- [St95] A. **Steininger** and H. Schweinzer. A model for the analysis of the fault injection process. In *Proc. FTCS-25*, pages 186–195. IEEE, 1995.
- [St96] Neil **Storey**. *Safety Critical Computer Systems*. Addison Wesley Longman, 1996. ISBN 0-201-42787-7.
- [St98] Andreas **Steininger**. How reproducible should fault injection experiments be? In *Proc. FTCS-28, Munich*. IEEE, 1998. Fast Abstract.
- [Vo98] Jeffrey M. **Voas**. *Software Fault Injection*. John Wiley & Sons, 1998. ISBN 0-471-18381-4.
- [Wa78] John **Wakerly**. *Error Detecting Codes, Self-Checking Circuits and Applications*. Elsevier North-Holland, 1978. ISBN 0-444-00256-1.
- [Yo93] Charles R. **Yount**. *The Automatic Generation of Instruction-Level Error Manifestations of Hardware Faults*. PhD thesis, Carnegie Mellon University, May 1993.
- [Yo96] Charles R. **Yount** and Daniel P. Siewiorek. A methodology for the rapid injection of transient hardware errors. *IEEE Transactions on Computers*, 45(8):881–891, Aug 1996.

BIBLIOGRAPHY ABBREVIATIONS

BCS	British Computer Society, www.bcs.org
DSN	Dependable Systems and Networks, www.dsn.org
ESS	European Simulation Symposium (organized by SCS)
FTCS	Fault Tolerant Computing Symposium
IASTED	International Association of Science and Technology for

Development, www.iasted.org
ICCD International Conference on Computer Design,
www.iccd-conference.org
ITC International Test Conference, www.itctestweek.org
SCS Society for Modeling and Simulation, www.scs.org

Index

Symbols

L — storage space, 55, **56**, 60
 L^F — interface area, 56
 L^I — input area, 56
 L^O — output area, 56
 L^W — working area, 56
 P_M — binary program, 6, **36**
 A — activations, **31**, 126
 F — fault set, 30, 118, 121
 M — measures, **31**, 145
 N — notions, **32**, 120
 P — predicates, **31**, 127
 R — readouts, **31**, 126
 V — valuation rules, **32**, 135
 Z — controlling process, **35**, 40, 62

A

activation, 127, 129
activations A , **31**, 126
arithmetic error, 89
– distribution, 90

B

bit-flip, 81, 92
– distribution, 87
blanking, 70, 122, **124**, 130

C

combinational circuit, 80
comparability, 27

controllability, 15

controlling process, 35, 40, 62
coverage, 145

D

data fault, 69, 113, 121, 139
dependability, 2, **13**, 22
– properties, 13
detection, **122**, 133

E

error, **9**, 121, 139
– activation, 129
– arithmetic, 89
– benign, 26, 138
– blanking, 70, 130
– detection, **122**, 133
– dormancy, 129
– explosion, 141
– family, 121
– masking, 130
– null, 90
– online detection, 46
– power-of-two, 92
– proliferation, 140
– propagation, 121, 131
– releasing, 132
– relevance, 137–139
– retention, 65
– service, **65**, 78, 112, 120

- signaling, 134
- error detection, **122**, 133
 - coverage, 148
 - latency, 122, 129
- error scenario, **126**
 - distinct, 136

F

- FARM*, 27
- failure, 10, 122
- failure injection, 135
- fault, 8, 120
 - activation, 127
 - data fault, 69, 113, 121, 139
 - double, 104, 106
 - effect, 9, 139
 - effectiveness, 128
 - mapping, 73
 - permanent, 75
 - process fault, 35, **47**, 119
 - random, **9**, 38
 - set F , 30, 118, 121
 - single, 100
 - software fault, 35
 - systematic, 9
 - transient, 75
- fault scenario, 80, **125**
- fault-injection, 13, 47
 - physical, 15
 - simulation based, 19, 148
 - software implemented, 18
- fault-simulation, 84
 - gate level, 28, 84
 - parallel, 142
- fault-tolerance, **11**, 52, 122, 125, 135
 - categories, 41, 45
 - coverage, 147
 - exterior, 41

- interior, 42
- interior hardware, **43**, 71, 146
- interior software, 43
- mechanisms, 46
- processing hardware, 39
- software, 35
- true tolerance, 122

G

- gate level, 7, 63
 - fault-simulation, 28, 84
- golden run, 26, 142

H

- hardware, 7
 - fault-injection, 2, 13
 - processing, 2, **37**, 55, 71
- hardware fault-injection, 13
 - attributes, 14, 22
 - techniques, 14

I

- input area L^I , 56
- instruction, 54
- interface area L^F , 56
- intrusion, 14

M

- manufacturer, 64, 79, 114, 155
- masking, 69, 122, **123**, 130
- measures M , **31**, 145
- mutant, 119
- mutant-injection, 117, 156
- mutation, 119

N

- nature, 35
- notions N , **32**, 120
- null-error, 90

O

observability, 15
output area L^O , 56

P

path of impact, 43, 51, 53, 66, 73
period of grace, 135
port, 84
power-of-two error, 92
predicates P , 31, 127
process, 6, 55
– components, 36
– controlling process, 35, 40, 62
– error, 48
– failure, 48
– fault, 35, 47, 119
– fault-injection, 47
processing hardware, 2, 37, 55, 71
program, 6, 36, 55
– binary P_M , 6, 36
– path analysis, 152
– shadow, 153
programming model, 51
proliferation, 140
propagation, 10, 46, 66, 71, 121, 123, 131

R

readouts R , 31, 126
redundant experiments, 137
register model, 51
reproducibility, 15
retention error, 65

S

safety, 10
service, 54, 57, 62, 119
– mutation, 119
service error, 65, 78, 112, 120

– classes, 78
service-provider, 53, 62
shadow program, 153
signaling, 134
silent death, 130
size of register, 83, 85
software, 6, 157
– fault, 35
– fault-injection, 13
state mutation, 49, 119
storage hardware, 64, 71
storage locations, 56
storage space L , 55, 56, 60
stuck fault, 84
SWIFI, 18

V

valuation rules V , 32, 135

W

working area L^W , 56

Z

Z80, 59