

# Thesis

Entwicklung eines Software-Systems zur Automatisierung der  
Assertion-Generation für die Verifikation eines Memory-Built-In-Self-Tests mithilfe  
einer Beschreibungssprache für Memory-Testalgorithmen

Zur Erlangung des akademischen Grades  
Bachelor of Engineering



**Fachhochschule  
Dortmund**

University of Applied Sciences and Arts

Verfasst von: Raphael Biermann  
Studiengang: Industrieelektronik und Sensorik  
Erstgutachter: Prof. Dr. Michael Karagounis, FH Dortmund  
Zweitgutachter: Markus Scholl, M.Sc., EPOS  
eingereicht in: Dortmund, am 26. April 2023

## Abstract

Memory-Testalgorithmen können in einer abstrakten Beschreibungssprache beschrieben werden, dessen Grammatik jedoch nicht ausreicht, um Scrambling im Memory zu berücksichtigen. Nach einer Grammatikerweiterung können Properties in der Hardware-Verifikationssprache SystemVerilog-Assertions aus dieser Beschreibung formuliert werden, die für eine Verifikation des Verhaltens des Memory-Interfaces eines Memory-Built-In-Self-Tests geeignet sind. Die Properties werden verwendet, um ein gegebenes Design zu verifizieren. In der Simulation werden Abweichungen von der ursprünglichen Spezifikation der Testalgorithmen erkannt.

Es werden Konzepte für die Automatisierung der Generierung von Properties erarbeitet, die anschließend in einem Software-System implementiert werden. Das Software-System unterstützt die Generierung von Assertions für March, SCAN und MATS Algorithmen mit beliebiger Länge, sowie einige Checkerboard und Initialisierungsalgorithmen, bei denen Scrambling berücksichtigt werden muss. Abschließend werden nötige Änderungen der Softwarearchitektur und Grammatik diskutiert, welche die Unterstützung weiterer Testalgorithmen ermöglichen.

English version:

Memory test algorithms can be described in an abstract description language, but its grammar is not sufficient to take scrambling in memory into account. After a grammar extension, properties in the hardware verification language SystemVerilog-Assertions can be formulated from this description, which are suitable for verifying the behaviour of the memory interface of a memory built-in self-test. The properties are used to verify a given design. In the simulation, deviations from the original specification of the test-algorithms are detected.

Concepts are developed for automating the generation of properties, which are then implemented in a software system. The software system supports the generation of assertions for March, SCAN and MATS algorithms of arbitrary length, as well as some checkerboard and initialisation algorithms where scrambling has to be considered. Finally, necessary changes to the software architecture and grammar are discussed to enable the support of other test algorithms.

## Danksagung

Diese Arbeit wurde in Zusammenarbeit mit Infineon (EPOS) erstellt, denen ich für die Bereitstellung der nötigen Arbeitsmittel danken möchte. Deren Hilfe und Engagement haben es mir ermöglicht, mich voll auf meine Forschung zu konzentrieren und meine Aufgaben effizient zu erledigen. Die Erfahrungen, die ich während meiner Tätigkeit sammeln konnte, haben mir außerdem geholfen, meine Fähigkeiten und Kenntnisse auszubauen.

An dieser Stelle möchte ich mich auch bei Herrn Prof. Dr. Karagounis für die Unterstützung bedanken. Durch seine Kontakte in der Branche habe ich die Möglichkeit erhalten, meine Arbeit bei EPOS schreiben und wertvolle Einblicke in die Praxis gewinnen. Ohne seine Hilfe hätte ich diese Chance nicht bekommen und könnte nicht auf eine so wertvolle Erfahrung zurückblicken. Ebenso möchte ich mich bei meinem ehemaligen Betreuer bei EPOS, Sven Cordes, bedanken, der mich bei der Themenwahl und der Betreuung meiner Arbeit unterstützt hat. Seine fachliche Expertise und konstruktive Kritik haben mir geholfen, meine Arbeit zu verbessern und meine Forschungsergebnisse zu präzisieren. Nicht zuletzt möchte ich mich bei meinen Eltern bedanken, die mich während meines Studiums finanziell und emotional unterstützt haben.

## Inhaltsverzeichnis

Abbildungsverzeichnis	5
Tabellenverzeichnis	6
1 Einleitung	9
2 Memory-Built-In-Self-Test	11
2.1 Funktionen . . . . .	11
2.2 Aufbau . . . . .	11
3 Dokumentation von Memory-Testalgorithmen	13
3.1 Abstrakte Beschreibungssprache für Testalgorithmen . . . . .	13
3.1.1 Grenzen der Sprache . . . . .	15
3.1.2 Spracherweiterung . . . . .	18
3.1.3 Beispiel: Dokumentation einiger Testverfahren in der Praxis . . . . .	20
4 Entwicklung einer Verifikationsumgebung für einen Memory-Built-In-Self-Test	24
4.1 Testbenchkonzept . . . . .	24
4.2 Entwicklung von Properties . . . . .	26
4.2.1 Funktionsblöcke . . . . .	26
4.2.2 Antecedent . . . . .	29
4.2.3 Consequent . . . . .	30
4.2.4 Beispiele . . . . .	30
4.3 Bind-Modul . . . . .	32
5 Automatisierung der Generierung von Assertions	34
5.1 Generierung von SVA-Properties . . . . .	34
5.1.1 Datengenerator . . . . .	34
5.1.2 Scrambling-Transformation . . . . .	34
5.1.3 Scrambling-Bypass . . . . .	37
5.2 Memory Parameter . . . . .	38
6 Entwicklung eines Software-Systems	40
6.1 Anforderungen . . . . .	40
6.2 Konzept . . . . .	40
6.3 Programmiersprache . . . . .	41
6.4 Frontend . . . . .	41
6.4.1 Grammatikdefinition . . . . .	41
6.4.2 Parser für die Beschreibungssprache . . . . .	43
6.4.3 Interne Datenstruktur . . . . .	45
6.4.4 Externe Schnittstellen und Datenstrukturen . . . . .	47
6.4.5 MemLib-Parser für Parameterextraktion . . . . .	47
6.5 Backend . . . . .	49
6.5.1 Bitfeld-Erzeugung . . . . .	49

6.5.2	Lookup-Table-Erzeugung . . . . .	51
6.5.3	Implementierung der Scrambling-Transformation . . . . .	53
6.5.4	Sequenzerkennung . . . . .	54
6.6	SystemVerilog-Writer . . . . .	55
6.6.1	Property-Writer . . . . .	55
6.6.2	Bind-Writer . . . . .	58
6.7	Auslieferung . . . . .	59
7	Ergebnisse . . . . .	61
7.1	Eignung der Beschreibungssprache für Memory-Testalgorithmen . . . . .	61
7.2	Eignung von SVA-Properties . . . . .	61
7.3	Software . . . . .	61
7.3.1	Unterstützte Testalgorithmen . . . . .	62
7.3.2	Einschränkungen . . . . .	63
8	Diskussion . . . . .	65
8.1	Coverage der unterstützten Algorithmen . . . . .	65
8.2	Coverage von Wortzugriffen . . . . .	66
8.3	Formale Verifikation . . . . .	66
8.4	Fallstudie: Verifikationsschleife . . . . .	67
9	Ausblick . . . . .	69
10	Fazit . . . . .	70
11	Anhang . . . . .	73
11.1	Unterstützte Algorithmen . . . . .	73

## Abbildungsverzeichnis

1	Generisches Memory-Built-In-Self-Test (MBIST) Konzept für ein Random-Access-Memory (RAM) nach [15]. Nicht dargestellt sind die Bypass-Pfade für den Memory-Zugriff von anderen Modulen. . . . .	12
2	Adressierungstechniken . . . . .	14
3	Checkerboard Pattern im physikalischen Memory. . . . .	15
4	Memory-Struktur nach distributiver Faltung mit MUX-Faktor 4. Die Multiplexer werden mit den gleichen Signalen gesteuert. . . . .	16
5	Speicherinhalt nach alternierendem Schreiben von 4'hA und 4'h5 in einem Memory mit Faltung mit MUX-Faktor 4. Eine geschriebene 1'b1 wird schwarz dargestellt. Eine geschriebene 1'b0 weiß. . . . .	17
6	Ideales Checkerboard in einem Memory mit Faltung mit MUX-Faktor 4. Eine geschriebene 1'b1 wird schwarz dargestellt. Eine geschriebene 1'b0 weiß. . . . .	17
7	Inkrement-Option. Jedes zweite Wort wird mit einem anderen Wert $B$ beschrieben. . . . .	19
8	Das Semikolon wird durch ein Komma ersetzt, um zu signalisieren, dass es sich nicht um separate Sequenzen handelt. . . . .	19
9	Write-Row Operand. . . . .	20
10	Simulation eines SCAN-Algorithmus. . . . .	20
11	Die Adressrichtung des SCAN ist durchgehend positiv. . . . .	21
12	Simulation eines Checkerboard Tests. . . . .	21
13	Anfang des Checkerboard-Tests. . . . .	22
14	Geschriebene Wörter beim Checkerboard Test. . . . .	22
15	RAM-Inhalt nach Checkerboard-Schreibphase. . . . .	23
16	Grundlegender Aufbau einer Testbench für die formale, Property-basierte Verifikation nach [17]. . . . .	24
17	Grundlegender Aufbau einer Testbench für die simulative, Property-basierte Verifikation. . . . .	25
18	<code>first_match</code> erkennt nur den ersten Zyklus von <code>write_enable</code> und bleibt inaktiv, bis die <i>Consequent</i> des Property erfüllt ist. . . . .	29
19	Ein Taktzyklus <i>finished</i> markiert ein abgeschlossenes Property. . . . .	29
20	Scrambling Tabelle als eindimensionales Array. . . . .	35
21	Berechnung der nötigen Ausgabevektoren durch eine Scrambling-Transformation mit der Lookup-Table. Einige Pfeile werden zugunsten der Übersichtlichkeit nicht dargestellt. . . . .	36
22	Beispielhafter logischer Aufbau eines RAM-Moduls mit angewandter distributiver Faltung. . . . .	38
23	Konzept für ein Software-System zur vollautomatisierten Entwicklung von Properties für die Verifikation der Testalgorithmen-Sequenzen auf dem MBIST Memory-Interface. Mögliche Datenformate werden am Rand dargestellt. . . . .	40
24	Aufbau eines Abstract-Syntax-Tree (AST) für die Beschreibungssprache für Memory-Testalgorithmen (BSMTA). . . . .	46

25	AST für einen Checkerboard-Term. Die Branches des zweiten <i>action_seq</i> Objekts werden zugunsten der Übersichtlichkeit nicht dargestellt. . . . .	50
26	Beziehungen der Lookup-Table (LUT)-Objekte. . . . .	52
27	Konzept für einen Algorithmus zur Erkennung von wiederkehrenden Sequenzen. Die in grün dargestellte Liste ist äquivalent zu der in schwarz dargestellten Wortliste. . . . .	54
28	Von der Software unterstützte Testalgorithmen nach Kategorie. . . . .	63
29	Verifikationsschleife. . . . .	68

## Tabellenverzeichnis

1	Operanden der Beschreibungssprache für Memory-Testverfahren nach [11, S. 27] und erweitert. . . . .	13
2	Beschreibung ausgewählter Testverfahren mit den Operanden aus Tabelle 1. . . . .	14
3	Constraining des MBIST-Kontrollinterfaces. . . . .	24
4	SystemVerilog-Assertions (SVA)-Sequenzen für Operanden der BSMTA. . . . .	27
5	Ausschnitt der Scrambling Tabelle für das gefaltete Memory in Abbildung 4 auf Seite 16 . . . . .	35
6	Syntax-Diagramme für die BSMTA. . . . .	43
7	Datenbank für BSMTA-Parameter. . . . .	50
8	Unterstützte Algorithmen aus [11, S. 29] und erweitert mit spezifischen Design-under-Test (DUT)-Algorithmen. Die Unterstützung basiert auf einer für die Software gültigen BSMTA-Beschreibung. Mit * markierte Algorithmen sind DUT-exklusiv und nicht in der Literaturquelle vorhanden. . . . .	73

## Abkürzungsverzeichnis

<b>ASIC</b>	anwendungsspezifische integrierte Schaltung . . . . .	9
<b>AST</b>	Abstract-Syntax-Tree . . . . .	39
<b>BIST</b>	Built-In-Self-Test . . . . .	9
<b>BNF</b>	Backus-Naur Form . . . . .	41
<b>BSMTA</b>	Beschreibungssprache für Memory-Testalgorithmen . . . . .	10
<b>CSV</b>	Comma-Separated-Values . . . . .	41
<b>DFT</b>	Design-For-Testing . . . . .	9

TABELLENVERZEICHNIS	7
<b>DRAM</b> Dynamic-Random-Access-Memory . . . . .	17
<b>DUT</b> Design-under-Test . . . . .	14
<b>ECC</b> Error-Correcting-Codes . . . . .	37
<b>EDA</b> Electronic-Design-Automation . . . . .	69
<b>FET</b> Feldeffekttransistor . . . . .	9
<b>IC</b> Integrierte Schaltung . . . . .	9
<b>IJTAG</b> Internal-JTAG . . . . .	24
<b>IP</b> Intellectual Property . . . . .	32
<b>JSON</b> JavaScript-Object-Notation . . . . .	47
<b>LSB</b> Least-Significant-Bit . . . . .	28
<b>LUT</b> Lookup-Table . . . . .	34
<b>MBIST</b> Memory-Built-In-Self-Test . . . . .	9
<b>MSB</b> Most-Significant-Bit . . . . .	28
<b>PLY</b> Python-Lex-Yacc . . . . .	43
<b>RAM</b> Random-Access-Memory . . . . .	12
<b>ROM</b> Read-Only-Memory . . . . .	15
<b>RTL</b> Register-Transfer-Level . . . . .	10

TABELLENVERZEICHNIS	8
<b>SRAM</b> Static-Random-Access-Memory . . . . .	47
<b>SVA</b> SystemVerilog-Assertions . . . . .	10
<b>UVM</b> Universal-Verification-Methodology . . . . .	62
<b>XML</b> Extensible-Markup-Language . . . . .	47

# 1 Einleitung

In der Entwicklung von integrierten Schaltungen (ICs) besteht ein enger Zusammenhang zwischen den gewünschten Eigenschaften des ICs und der verwendeten Strukturgröße, der minimalen Gate-Länge der verwendeten Feldeffekttransistoren (FETs). Hohe Schaltfrequenzen im Gesamtprodukt lassen sich durch hohe Transitfrequenzen  $f_T$  der FETs erreichen, die wiederum antiproportional zur Gate-Länge sind [16, S. 384]. Zusätzlich begünstigt die kürzere Kanallänge eine niedrigere Schwellenspannung  $U_{th}$ , wodurch die benötigte Betriebsspannung gesenkt wird. Für den Produzenten bedeutet die Verkleinerung der Gesamtfläche des Chips außerdem eine große Einsparung an Kosten, da durch die höhere Technologiedichte insgesamt eine kleinere Fläche eingenommen wird und somit mehrere Chips auf einen Siliziumwafer passen. Die genannten Zusammenhänge sprechen dafür, eine möglichst kleine Strukturgröße zu verwenden, wobei zusätzlich noch Strahlenhärte und andere physikalische Effekte, die bei kleineren Strukturen an Relevanz gewinnen, berücksichtigt werden müssen. Auch die stets steigenden Anforderungen an die Elektronik, die vor allem im Bereich Automotive zu sehen sind, sprechen für eine höhere Technologiedichte. Diese Entwicklung, die schon vor mehr als 50 Jahren als Mooresches Gesetz hinsichtlich der Transistoranzahl<sup>1</sup> auf einem IC beschrieben wurde, lässt sich durch das wachsende Bedürfnis nach Sicherheit und Komfort in dieser Branche begründen. Neue Technologien wie Computer-Vision, autonomes Fahren, Abstandswarnung und Kollisionserkennung werden durch Kameras und Radar<sup>2</sup> gestützt, die mithilfe von anwendungsspezifischen integrierten Schaltungen (ASICs) realisiert werden. Ein vergleichsweise großer Teil der Chipfläche wird hierbei von Memory Makros eingenommen. Auch hier steigen die Ansprüche an die Schnelligkeit, aber auch an die Speicherdichte. Speicherzellen werden immer näher aneinander platziert, wodurch zwar die Anzahl der speicherbaren Bits, jedoch auch die Empfindlichkeit gegenüber negativen Einflüssen von benachbarten Speicherzellen<sup>3</sup> oder die Wahrscheinlichkeit von Störungen durch die Adress- und Datenleitungen steigt [5, S. 26]. Mit diesen Nebeneffekten, die mit der Steigerung der Technologiedichte riskanter werden, wird die Relevanz des Memory-Testings deutlich. Die Testkosten dürfen mit der Speicherdichte nicht wachsen. Die Herausforderung ist hier einerseits ein Set von Testalgorithmen zu nutzen, welche im Bestfall die Komplexität<sup>4</sup>  $O(n)$  haben [5, S. 27], wodurch die Testzeit nur linear mit der Anzahl der Bits wächst, und andererseits die Tests zu automatisieren. Diese Testalgorithmen sollen zudem gezielt auf den vorher genannten Fehlermodellen basieren. Dies ist eine Motivation dafür, Selbsttestmechanismen im Rahmen des Design-For-Testing (DFT)-Konzepts zu implementieren. Built-In-Self-Tests (BISTs) erlauben die automatische Erkennung von Fehlern in einem System. Hier wird ein Subtyp, der Memory-Built-In-Self-Test (MBIST) diskutiert, der die oben genannten Testverfahren automatisiert ausführen kann. Dadurch werden Kosten gespart, die ansonsten für manuelle Testverfahren anfallen würden. Die ordnungsgemäße Funktion des MBIST muss verifiziert werden, da sie maßgeblich für dessen Testcoverage ist. Dafür muss die korrekte Ausfüh-

---

<sup>1</sup>Heute ist das Mooresche Gesetz sehr gut in der Erhöhung der Technologiedichte zu sehen.

<sup>2</sup>Beispielsweise realisiert mit Infineon RASIC™ CTRX8181 transceiver auf 28-nm CMOS-Technologie [9].

<sup>3</sup>Beispielsweise Übersprechen bei Schreibvorgängen.

<sup>4</sup>Schrankenfunktion aus der Komplexitätstheorie [21].

rung der Memory-Testalgorithmen geprüft werden.

In dieser Arbeit wird ein MBIST, der aus einem Meta-Modell generiert wurde, analysiert. Dafür wird zuerst die Architektur des MBISTs und die Connectivity mit den Memory-Instanzen untersucht. An den Interfaces des MBISTs sollte eine Konfiguration des auszuführenden Testalgorithmus möglich sein und der Teststatus abgelesen werden können. Die laufenden Testalgorithmen sollten als Stimuli auf den Memory-Interfaces sichtbar sein. Die genutzten Algorithmen werden in der Register-Transfer-Level (RTL)-Simulation Cycle-to-Cycle<sup>5</sup> analysiert und, falls möglich, bereits in der Literatur beschriebenen Algorithmen zugeordnet. Hierfür wird eine in der Literatur genutzte abstrakte Beschreibungssprache für Memory-Testalgorithmen (BSMTA) zur Beschreibung der undokumentierten Algorithmen verwendet und, wo es notwendig ist, erweitert. Es wird ein Testbench-Konzept für die Simulation entworfen. Dafür wird eine existierende Verifikationsumgebung übernommen. Anschließend werden SystemVerilog-Assertions (SVA)-Properties für die Verifikation des Memory-Interfaces des MBISTs geschrieben. Diese sollen die Struktur der BSMTA berücksichtigen. Die Properties werden in die Testbench eingepflegt und es werden Simulationen durchgeführt, in denen Abweichungen von der SVA-Referenz erkannt werden sollen. Daraufhin wird diskutiert, wie die Property-Entwicklung automatisiert werden kann, um Rahmenbedingungen für die Implementierung eines Software-Systems zu formulieren, das diese erfüllt. Ein Konzept dafür wird dabei entwickelt. Semantik und Syntax der erweiterten Beschreibungssprache werden in einer formalen Metasprache zur Darstellung kontextfreier Grammatiken festgelegt, um Lexer und Parser für die Beschreibungssprache entwickeln zu können, welche die BSMTA verarbeiten und in andere Datenstrukturen überführen. Die erzeugten Datenstrukturen werden zusammen mit den Parametern des Memorys genutzt, um Bitfelder und anschließend SVA-Properties für BSMTA-Terme zu generieren. Abschließend werden die unterstützten Algorithmen analysiert.

---

<sup>5</sup>Manuelles oder automatisches Prüfen des Ausgangsverhaltens. Dabei wird jeder Taktzyklus der Clock geprüft, um sicherzugehen, dass das Verhalten immer korrekt ist.

## 2 Memory-Built-In-Self-Test

Ein Memory ist eine Matrixanordnung von Bitzellen, die individuell oder gemeinsam mit anderen Zellen beschrieben werden können. Da bei der Realisierung oder Alterung eines Chips verschiedene Fehler im Memory entstehen können, bedarf es einem DFT-Konzept, welches das automatisierte Testen der ordnungsgemäßen Funktion im Feld<sup>6</sup> ermöglicht. Eine Lösung dafür ist eine MBIST Implementierung, die entweder Soft<sup>7</sup>, Hard<sup>8</sup> oder Hybrid<sup>9</sup> sein kann [15]. Die Funktionen und der Aufbau verschiedener Implementierungen sind dabei ähnlich, weshalb hier nur die generischen Eigenschaften beschrieben werden.

### 2.1 Funktionen

Ein MBIST muss folgende Funktionen aufweisen:

- Definierte Memory-Testverfahren ausführen
  - Konfiguration des Testverfahrens.
  - Im Testverfahren definierte Daten auf ganzen Adressbereich schreiben.
  - Im Testverfahren definierte Daten vom ganzen Adressbereich lesen.
  - In Testverfahren definierte Adresssequenzen während Schreib- und Lesephasen generieren und ausgeben.
- Fehler erkennen
  - Abgleich von gelesenen Daten mit erwarteten Daten.
- Erfolgreichen Testdurchlauf quittieren.
- Fehlgeschlagenen Testdurchlauf quittieren.

Daraus ergeben sich die Submodule und Interfaces, die im nächsten Abschnitt beschrieben werden.

### 2.2 Aufbau

Ein MBIST besteht aus einem Adress-Generator, einem Test-Pattern Generator und einer Kontrolllogik [15]:

---

<sup>6</sup>Nach der Auslieferung zum Kunden soll sich das System selbst testen.

<sup>7</sup>Als Software implementiert.

<sup>8</sup>Als Hardware-Module implementiert.

<sup>9</sup>Hardwaremodule mit programmierbaren Testalgorithmen.

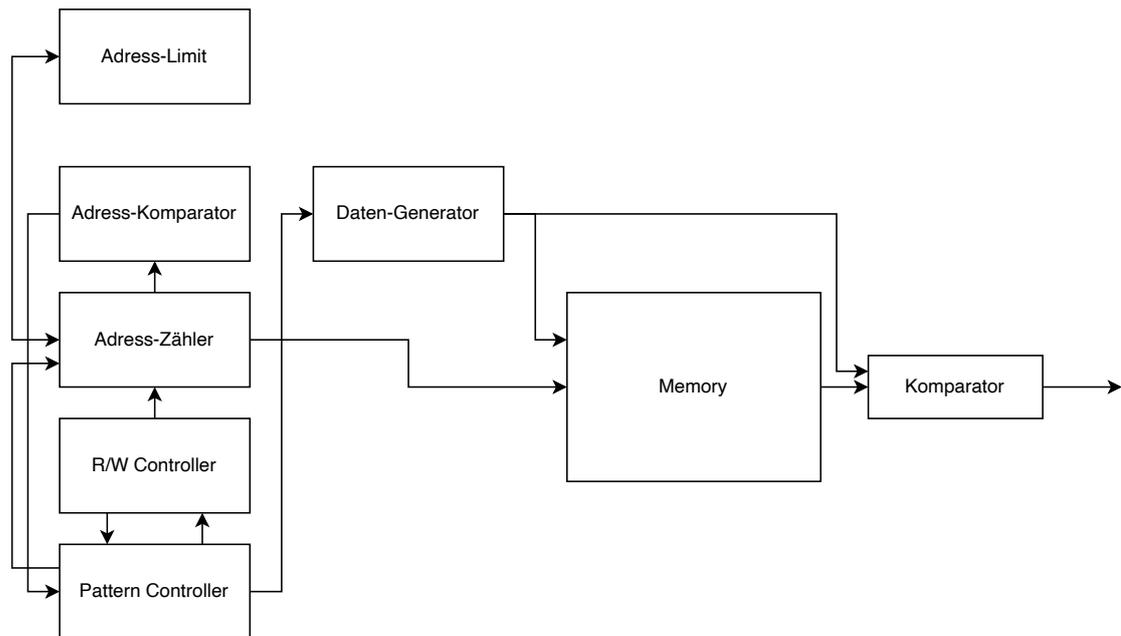


Abbildung 1: Generisches MBIST Konzept für ein Random-Access-Memory (RAM) nach [15]. Nicht dargestellt sind die Bypass-Pfade für den Memory-Zugriff von anderen Modulen.

Die Funktionen sind hierbei [15]:

**Adress-Generator:** Bestehend aus Adress-Zähler, Limit und Komparator, generiert dieser die Adress-Sequenzen der Test-Patterns. Der Adress-Zähler muss mindestens einfach Inkrementieren oder Dekrementieren können. Andere Zählweisen sind möglich.

**Read/Write-Controller:** Liefert die Adressposition im Memory für eine Schreib- oder Leseoperation. Legt außerdem fest, wie viele Taktzyklen gewartet werden muss, bevor der Adress-Zähler nach einer Operation inkrementiert wird.

**Daten-Generator:** Stellt die korrekten Daten zu den Phasen des Testverfahrens. Diese werden einerseits in das Memory während der Schreibphase geschrieben und andererseits in der Lesephase unter Verwendung eines Komparators mit den gelesenen Daten verglichen.

Der MBIST hat ein Memory-Interface mit den gewöhnlichen Signalen *write\_enable*, *data\_in*, *data\_out*, *addr\_in* und *chip\_select* und kann damit beim Start des Chips in einer Hard- oder Hybrid-Implementierung selbstständig das Memory beschreiben und auslesen. Für die MBIST Aktivierung und Konfiguration des auszuführenden Testverfahrens existiert ein Konfigurationsinterface an der Kontrolllogik, dessen Implementierung variieren kann.

### 3 Dokumentation von Memory-Testalgorithmen

In diesem Kapitel wird untersucht, wie Memory-Testalgorithmen dokumentiert werden können und ob diese Art der Dokumentation in der Praxis anwendbar ist.

#### 3.1 Abstrakte Beschreibungssprache für Testalgorithmen

Bereits in der Literatur [11, S. 28] werden einige Memory-Testalgorithmen in einer abstrakten Beschreibungssprache beschrieben. Die Operanden der Sprache sind in der nachfolgenden Tabelle aufgeführt:

Tabelle 1: Operanden der Beschreibungssprache für Memory-Testverfahren nach [11, S. 27] und erweitert.

Symbol	Bedeutung
wx	Schreibe x in eine Zelle
rx	Lese x aus einer Zelle
$n$	Wiederhole die Operation n-mal hintereinander
$\uparrow$	Adress-Inkrement
$\downarrow$	Adress-Dekrement
$\updownarrow$	Don't-care Adressrichtung
$\nearrow$	Adress-Inkrement entlang der Hauptdiagonalen
$\diamond$	Nord-Ost-Süd-West (N-E-S-W) Adressierung um Base-Cell
$*$	N-NE-E-SE-S-SW-W-NW Adressierung um Base-Cell
$D$	Delay
$b$	Operand bezieht sich auf Base-Cell
$-b$	Operand bezieht sich auf alle Zellen außer Base-Cell
$Rep$	Operation k mal auf n Zellen mit der Distanz $2^k$ von N E S W von Base-Cell anwenden
$R-b$	Adress-Reihe der Base-Cell
$C-b$	Adress-Spalte der Base-Cell
$x$	fast-x Adressierung
$y$	fast-y Adressierung

Einige wichtige<sup>10</sup> Testverfahren sind bereits mit einer Teilmenge dieser Operanden beschreibbar:

<sup>10</sup>Bezogen auf Coverage, Schnelligkeit und einfache Implementierung. Scan und March sind beliebte Standardtests. Die gesamte Tabelle kann der Quelle [11, S. 28] entnommen werden.

Tabelle 2: Beschreibung ausgewählter Testverfahren mit den Operanden aus Tabelle 1.

Algorithmus	Sequenz
SCAN	$\{\uparrow(w0); \uparrow(r0); \uparrow(w1); \uparrow(r1); \}$
March C+	$\{\uparrow(w0); \uparrow(r0, w1, r1); \uparrow(r1, w0, r0); \downarrow(r0, w1, r1); \downarrow(r1, w0, r0); \downarrow(r0); \}$
HAM5R	$\{\uparrow(w0); \uparrow(w1, r1^5); \uparrow(w0, r0^5); \downarrow(w1, r1^5); \downarrow(w0, r0^5); \}$

Daher liegt der Fokus dieser Arbeit auf der folgenden Teilmenge:

$$\{\uparrow, \downarrow, \updownarrow, wx, rx\} \quad (1)$$

Ein March-Test besteht aus einer endlichen Anzahl von Sequenzen, die in einer ansteigenden, absteigenden oder zufälligen Adressordnung ausgeführt werden. Die Sequenzen dürfen nur aus den Operanden  $w0$ ,  $w1$ ,  $r0$ ,  $r1$  bestehen [5, S. 86]. Es dürfen mehrere Operanden in einer Sequenz stehen. So können wie beim March C+ in Tabelle 2 in einer Sequenz die Daten aus einer Zelle gelesen, neue Daten in die Zelle geschrieben und erneut ausgelesen werden, um sicherzustellen, dass der Schreibvorgang erfolgreich war. Jeder Operand braucht je einen Taktzyklus pro Zelle. Die Pfeile stehen für die Adressrichtung einer Sequenz. Die Sequenz  $\uparrow(w0)$  bedeutet, dass der logische Wert 0 nacheinander auf alle Adressen im Memory in aufsteigender Adressrichtung geschrieben wird.

SCAN-Tests bestehen aus Sequenzen mit jeweils einem einzigen Operanden  $w0$ ,  $w1$ ,  $r0$  oder  $r1$ .

Die meisten Hammer-Tests lassen sich ebenfalls mit der Teilmenge (1) beschreiben. Einige wenige basieren auf einer Base-Cell Adressierung, bei der jeweils eine Zelle betrachtet wird, und der Effekt von Schreibvorgängen auf umliegende Zellen beobachtet wird. Diese Tests werden jedoch in dem hier beschriebenen Design-under-Test (DUT) nicht verwendet.

Neben der Adressrichtung gibt es zwei Adressierungstechniken zur Speicheroptimierung, die in Abbildung 2 dargestellt sind.

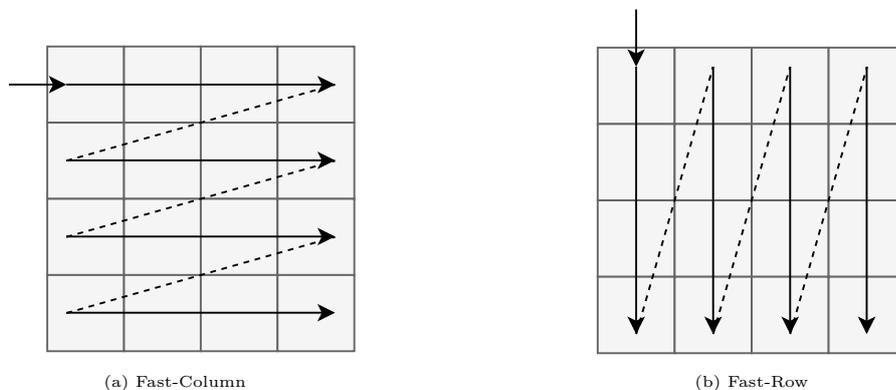


Abbildung 2: Adressierungstechniken

Bei der Fast-Row Technik wird die Zeilenadresse schneller geändert als die Spaltenadresse. Bei der Fast-Column Technik wird die Spaltenadresse schneller geändert als die

Zeilenadresse [4]. Dabei entspricht Fast-Column der natürlichen Adressrichtung und wird daher angenommen, wenn Fast-Row nicht ausdrücklich definiert ist.

Die BSMTA kann auch für die Verifikation von Read-Only-Memories (ROMs) verwendet werden, indem nur Leseoperanden verwendet werden. Mit den Leseoperanden muss der bekannte Speicherinhalt beschrieben werden, wenn ein MBIST die BSMTA als Eingabe erhalten soll.

### 3.1.1 Grenzen der Sprache

In seiner Grundform beschreibt die Sprache das Schreiben auf einzelne Memory-Zellen. In den meisten Fällen ist ein Memory jedoch nicht Bit-adressierbar. Oft können nur ganze Wörter adressiert werden. Um eine spezifische Stelle zu beschreiben, müssten komplexere Memory-Operationen wie Read-Modify-Writes durchgeführt werden, die erst ein Wort aus dem Memory lesen, ein Bit im Wort modifizieren und dann das Wort zurückschreiben. Da es für die oben genannten Testverfahren für den Speicherinhalt unerheblich ist, ob ein einzelnes Bit oder ein ganzes Wort geschrieben oder ausgelesen wird, wird an dieser Stelle die vereinfachte Annahme getroffen, dass die oben genannten Operanden das Schreiben oder Lesen eines ganzen Wortes der spezifischen Wortbreite des Memorys beschreiben.

Auffällig ist außerdem, dass der Checkerboard Algorithmus nicht in der Tabelle [11, S. 28] enthalten ist. Hier lässt sich die These aufstellen, dass sich mit den oben aufgeführten Operanden nur Algorithmen beschreiben lassen, deren Coverage nicht von der tatsächlichen Memory-Konfiguration abhängt. March-, Scan- und Hammer-Tests schreiben in einer Sequenz auf alle Adresspositionen den gleichen Wert. Hier wird kein spezifisches Muster in das Memory geschrieben. Der Checkerboard Algorithmus erzeugt hingegen das folgende Pattern im Memory:

1'b1	1'b0	1'b1	1'b0
1'b0	1'b1	1'b0	1'b1
1'b1	1'b0	1'b1	1'b0
1'b0	1'b1	1'b0	1'b1

Abbildung 3: Checkerboard Pattern im physikalischen Memory.

Würde es sich um ein Memory mit der in Abbildung 3 gezeigten Größe und 4-Bit Wortbreite handeln, so liegt die Vermutung nahe, dass ein alternierendes Schreiben der Wörter 4'hA und 4'h5 das gezeigte Pattern erzeugen würde.

Hier stößt die Beschreibungssprache jedoch an ihre Grenzen, da es keine Möglichkeit gibt, das alternierende Schreiben von verschiedenen Wörtern mit den Basisoperanden zu beschreiben. Hierfür muss später ein weiterer Operand eingeführt werden.

Ein weiteres Problem ist, dass das oben aufgeführte Verfahren mitnichten in jedem Me-

memory das gewünschte Checkerboard-Pattern erzeugt. In einigen Memories gibt es laut [19] sogenanntes Scrambling, wodurch die logische<sup>11</sup> Struktur eines Memorys von der physikalischen Struktur abweicht. Scrambling geschieht in erster Linie auf natürliche Art aufgrund von Optimierungen. Ein Beispiel dafür ist die sogenannte Faltung. Dies ist ein Optimierungsverfahren, um die Dimensionen des physikalischen Memorys so zu optimieren, dass ein möglichst quadratisches Memory entsteht. Hierfür werden mehrere Wörter in eine Reihe geschrieben und verschachtelt. Die folgende Abbildung zeigt ein distributiv<sup>12</sup> gefaltetes Memory mit 4-Bit Wortbreite. Die Wörter sind miteinander verschachtelt und eine Bit-Adressierung ist möglich. Die Adressen sind den einzelnen Zellen zugeordnet:

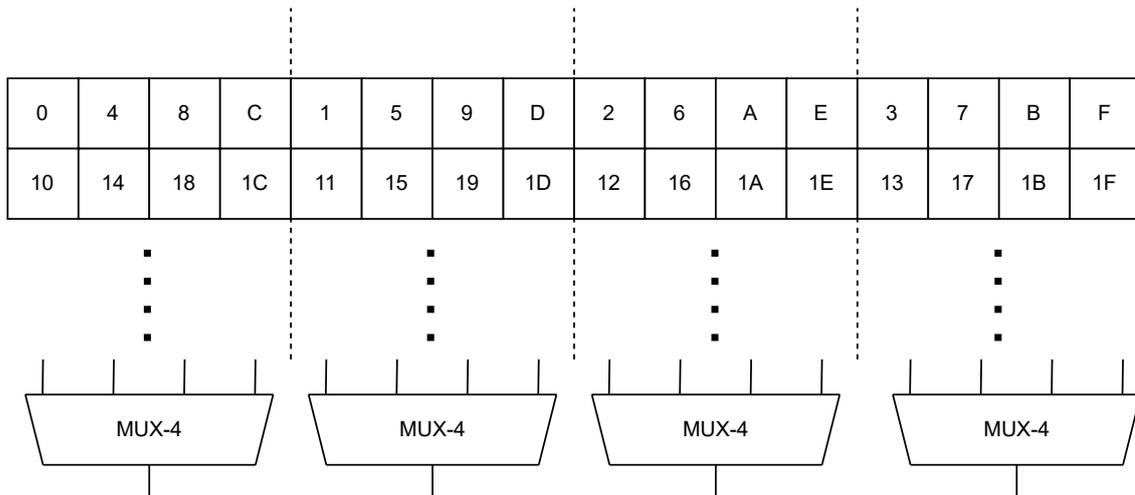


Abbildung 4: Memory-Struktur nach distributiver Faltung mit MUX-Faktor 4. Die Multiplexer werden mit den gleichen Signalen gesteuert.

Hier zeigt sich, dass die Bits eines Wortes nicht direkt nebeneinanderstehen. Für jedes Bit der Wortlänge existiert ein Multiplexer. Der  $n$ -te Multiplexer gibt je nach Ansteuerung das  $n$ -te Bit des ersten, zweiten, dritten oder vierten Wortes aus. Der MUX-Faktor legt fest, wie viele Wörter in einer Reihe stehen und verschachtelt werden. Die Bitgroup-Boundaries sind die Grenzen zwischen den Multiplexern, die später für die Adressierung wichtig sind.

Würde, mit der Absicht ein Checkerboard-Pattern im physikalischen Memory zu erhalten, alternierend 4'hA und 4'h5 geschrieben werden, so würde in einem gefalteten Memory ein vergleichbares Pattern entstehen:

<sup>11</sup>Bezogen auf die Simulation. Für das Memory werden RTL-Modelle substituiert, die im Normalfall lediglich lineare Wortlisten sind.

<sup>12</sup>Es wird zwischen distributiver (verteilter) und adjazenter Faltung unterschieden. Während bei der adjazenten Faltung die Bits eines Wortes weiterhin nebeneinanderstehen, werden sie bei der distributiven Faltung in der Reihe verteilt [19].

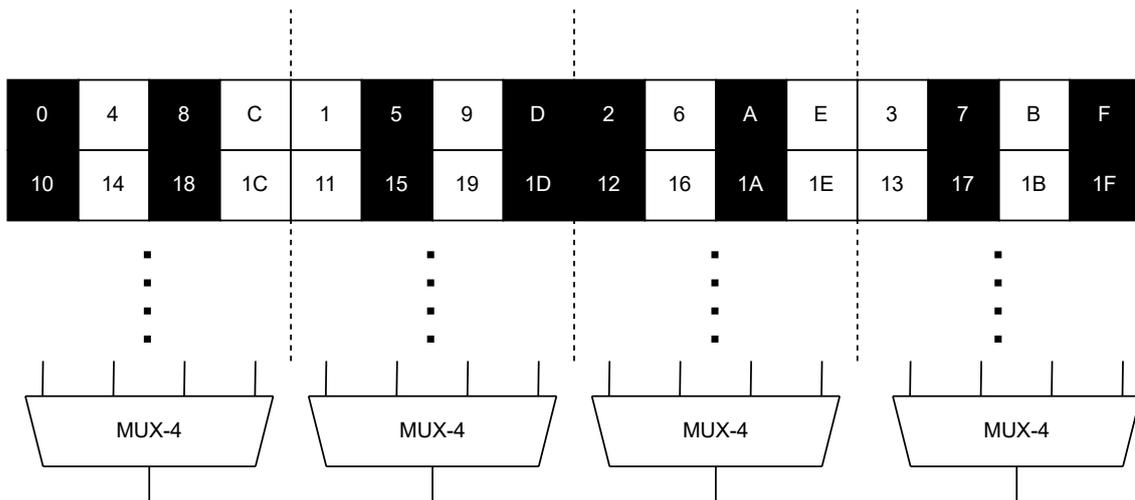


Abbildung 5: Speicherinhalt nach alternierendem Schreiben von 4'hA und 4'h5 in einem Memory mit Faltung mit MUX-Faktor 4. Eine geschriebene 1'b1 wird schwarz dargestellt. Eine geschriebene 1'b0 weiß.

Das dargestellte Pattern weicht stark von dem Checkerboard-Pattern ab. In einem Checkerboard ist jede Zelle von seinem invertierten Wert umgeben, wodurch der Leckstrom zwischen den Zellen durch den maximalen Potenzialunterschied sein Maximum erreicht. Mit dem Checkerboard Test können Kurzschlüsse zwischen Zellen gefunden werden, solange der Adress-Dekodierer richtig funktioniert [5, S. 99]. Zusätzlich kann geprüft werden, ob ein Dynamic-Random-Access-Memory (DRAM) an einer sogenannten *sleeping-sickness* leidet, bei der nach längerer Inaktivität<sup>13</sup> die Ladung der Kapazität einer Speicherzelle durch einen Leckstrom verloren geht [5, S. 329]. Der Checkerboard-Test maximiert den Leckstrom und die Wahrscheinlichkeit, dass bei einem Refresh ein falscher Wert gelesen und zurückgeschrieben wird. In dem in Abbildung 5 dargestellten Pattern entstehen Cluster von Bits mit gleichen Werten an den Bitgroup-Boundaries. Dort sind die Zellen von nur einer Zelle der invertierten Ladung umgeben. Der Leckstrom erreicht hier nicht sein Maximum und die Coverage für Kurzschlüsse zwischen den Zellen ist viel geringer.

Ein ideales Checkerboard würde beispielsweise so aussehen:

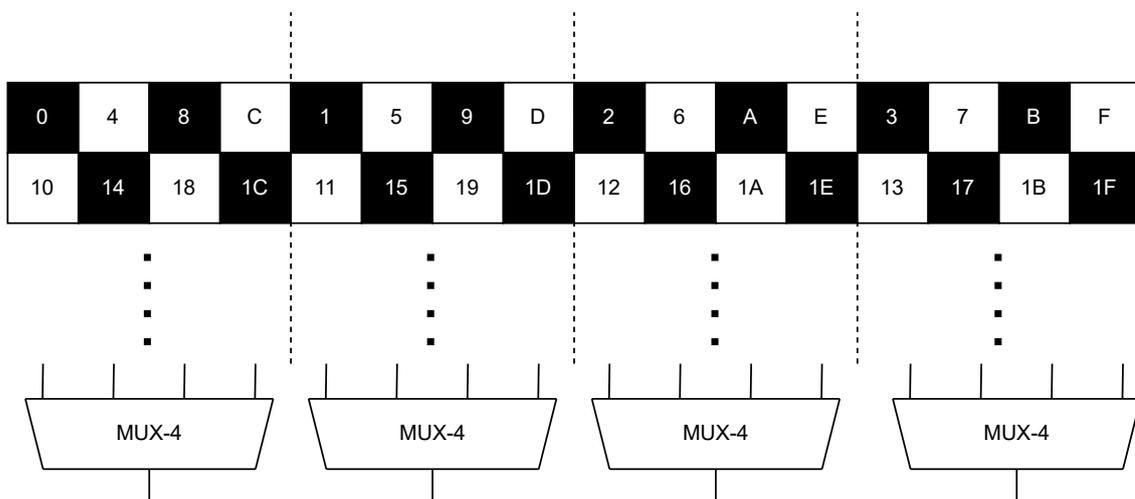


Abbildung 6: Ideales Checkerboard in einem Memory mit Faltung mit MUX-Faktor 4. Eine geschriebene 1'b1 wird schwarz dargestellt. Eine geschriebene 1'b0 weiß.

<sup>13</sup>Typisch sind 100 $\mu$ s bis 100ms ohne Refresh ([5], S.329)

Die zu schreibenden Wörter lassen sich aus Abbildung 6 ablesen:

1. 4'b1111
2. 4'b0000
3. 4'b1111
4. 4'b0000
5. 4'b0000
6. 4'b1111
7. 4'b0000
8. 4'b1111

Es zeigt sich, dass für ein Checkerboard nicht 4'hA und 4'h5, sondern 4'hF und 4'h0 geschrieben werden müssen. Zusätzlich muss die zweite Reihe im Memory wieder mit dem Wert 1'b0 beginnen. Daher ist das fünfte Wort wieder 4'b0000. Für diese Wiederholung muss der MUX-Faktor bekannt sein. Das bedeutet für die Beschreibungssprache, dass Testalgorithmen bei denen bestimmte Muster, sogenannte Data-Backgrounds, im Memory erzeugt werden, wie das genannte Beispiel des Checkerboards, nicht generisch<sup>14</sup> beschrieben werden können, da das Memory-Scrambling bekannt sein und berücksichtigt werden muss. Eine Ausnahme bildet ein *solid*-Data-Background. Dort werden alle Zellen mit dem gleichen Wert beschrieben, wie es bei den March- oder Scan-Algorithmen der Fall ist.

### 3.1.2 Spracherweiterung

Während der Analyse der abstrakten Sprache für Memory-Testalgorithmen haben sich folgende Probleme gezeigt:

1. Mit den Basisoperanden der Sprache sind keine Schreib- oder Lesevorgänge mit alternierenden Wörtern möglich.
2. Bei einer wortbasierten Beschreibung des Speicherinhaltes sind Kenntnisse über das Scrambling (beispielsweise die Faltung) notwendig.
3. Aufgrund des Scramblings unterscheiden sich logisch geschriebene und physikalische Muster im Memory.
4. Wird Punkt 3 nicht beachtet, so sinkt die Testcoverage stark für ein Testverfahren mit einem alternierenden Data-Background wie das Checkerboard.
5. Es fehlen Symbole (Parameter) für das Schreiben von ganzen Wörtern.
6. Einige der Symbole sind auf einer Tastatur mit deutschen oder englischen Standardlayout nicht enthalten.

---

<sup>14</sup>für jedes Memory geltend

Daher werden an dieser Stelle neue Operanden in die Beschreibungssprache eingeführt, mit denen auch mustersensitive Testverfahren beschrieben werden können. Zudem werden schwierig schreibbare Operanden wie die Pfeile aus der zuvor definierten Teilmenge von Operanden (1) für Inkrement, Dekrement und Don't Care Adressrichtung mit den Unicode Pfeilen '↑' '↓' und '↕' ersetzt.

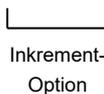
Bei der Einführung von Operanden muss darauf geachtet werden, dass die Beschreibungssprache weiterhin für alle Testalgorithmen generisch bleibt.

Ein Operand, der alternierende Wörter beschreiben kann, ist eine Inkrement- oder Dekrement-Option mit Start, Schrittweite und Ende. Das Checkerboard in Abbildung 3 auf Seite 15 mit  $A=4'hA$  und  $B=4'h5$  könnte beispielsweise so beschrieben werden:

$$\{\uparrow[0:2:](wA);\uparrow[1:2:](wB);\}$$



Inkrement-  
Option



Inkrement-  
Option

Abbildung 7: Inkrement-Option. Jedes zweite Wort wird mit einem anderen Wert  $B$  beschrieben.

Ein ausgelassener Wert referenziert den jeweiligen Standardwert:

- Startwert: 0 (erstes Wort)
- Schrittweite: 1
- Endwert:  $n-1$  (letztes Wort)

Das Semikolon trennt Schreibsequenzen voneinander. Da das Checkerboard wortweise geschrieben werden soll, wird ein weiteres Trennzeichen für zugehörige Elemente eingeführt. Das Komma vereint beide Elemente zu einer Schreibsequenz:

$$\{\uparrow[0:2:](wA),\uparrow[1:2:](wB);\}$$

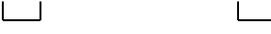


zugehörige Sequenzelemente

Abbildung 8: Das Semikolon wird durch ein Komma ersetzt, um zu signalisieren, dass es sich nicht um separate Sequenzen handelt.

Mit den neu eingeführten Operanden lässt sich jedoch noch nicht das in Abbildung 6 gezeigte Checkerboard beschreiben. Es würde das Muster in Abbildung 5 vor dem Scrambling beschrieben werden. Für das ideale Checkerboard fehlt ein Operand, der eine ganze Reihe im physikalischen Memory und nicht nur ein Wort beschreibt. Die Inkrement- und Dekrement-Option gilt dann für die Reihe und nicht für das Wort. Daher wird an dieser Stelle ein neuer Operand  $r$  eingeführt, der einen Reiheneinhalt im physikalischen Memory beschreibt:

$$\{\uparrow[0:2:](wrA), \uparrow[1:2:](wrB);\}$$



Auswahl einer Zeile

Abbildung 9: Write-Row Operand.

Mit den neuen Operanden lassen sich die fehlenden Muster beschreiben. Mit dem Ausdruck in Abbildung 9 wird jede Reihe mit ungeradem Index im physikalischen Memory mit dem Reihenvektor B beschrieben. Die Operanden sind auch für Leseoperationen gültig.

### 3.1.3 Beispiel: Dokumentation einiger Testverfahren in der Praxis

Um zu prüfen, ob sich die Beschreibungssprache für Memory-Testalgorithmen für die Verifikation eines MBISTs eignet, werden einige Memory-Testalgorithmen in einer RTL-Simulation auf dem Memory-Interface eines DUTs ausgeführt und mit der erweiterten Beschreibungssprache beschrieben. Hierfür wird eine bereits bestehende Testbench verwendet, die eine geeignete Startsequenz treibt, um den MBIST in einen Testmodus zu überführen. Der einzige Parameter, der angepasst werden muss, ist die Algorithmus-Konfiguration. Hierbei handelt es sich um einen 5-Bit breiten Eingang am parallelen Kontrollinterface des MBIST, dessen Belegung während der Startsequenz den auszuführenden Algorithmus steuert.

Da die Memory-Testverfahren für den DUT-MBIST noch nicht in der Beschreibungssprache formuliert sind, ist dies lediglich ein Versuch, diese aus der Simulation zu beschaffen. Wichtig ist, dass für zukünftige DUTs keinesfalls derselbe Ansatz verwendet wird, da später nur Properties für das sichtbare Verhalten auf den MBIST-Interfaces entwickelt werden können. Ohne eine Spezifikation der Testverfahren kann nur angenommen werden, dass das Verhalten korrekt ist. Die Properties alleine erlauben hier keine Aussage über die richtige Funktion. Es wird ein SCAN-Algorithmus simuliert:

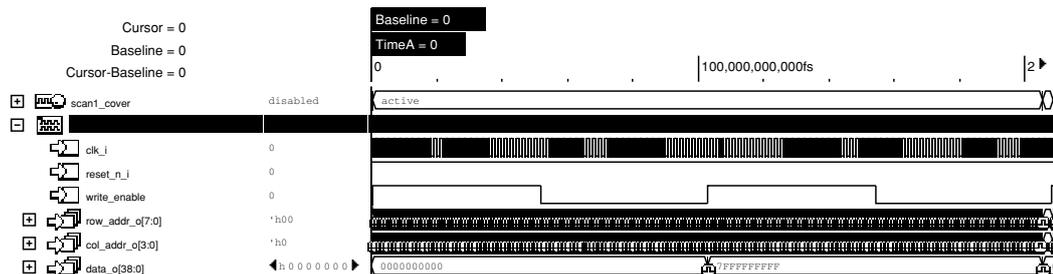


Abbildung 10: Simulation eines SCAN-Algorithmus.

Abbildung 10 zeigt einen vollständigen Testdurchlauf. Die Signale geben Hinweise zur Testzusammensetzung. Folgende Pseudocode Sequenz lässt sich aus dem *write\_enable* Signal ableiten:

*write* → *read* → *write* → *read*

Der SCAN sollte in einer Sequenz nur aus einem einzigen Operanden bestehen. Das gezeigte Testverfahren besteht also aus vier Sequenzen. Dabei liegen folgende Daten während eines Schreibzyklus auf *data\_o* des MBIST:

$$0 \rightarrow 1$$

Hier ist es wichtig zu beachten, dass zuvor die Vereinfachung getroffen wurde, dass die Operanden der Beschreibungssprache einen Wortzugriff beschreiben und jeweils alle Bits eines Wortes mit 0 oder 1 beschrieben werden. Weil nur gelesen werden kann, was geschrieben wurde, kann folgende Sequenz aufgestellt werden:

$$write0 \rightarrow read0 \rightarrow write1 \rightarrow read1$$

Für die Beschreibungssprache fehlt noch der Adressoperator. Dafür muss die Adressrichtung analysiert werden:

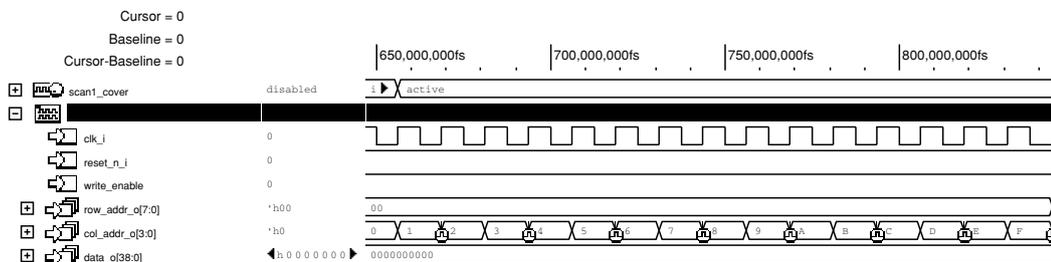


Abbildung 11: Die Adressrichtung des SCAN ist durchgehend positiv.

Hier ist erkennbar, dass die Adresse auf *col\_addr\_o* bei jeder positiven Taktflanke inkrementiert wird. Wichtig ist außerdem, dass die Spaltenadresse schneller steigt als die Reihenadresse. Hier wird also eine *fast-column* oder *fast-y* Adressierung verwendet. Die Adressrichtung ist in allen Sequenzen identisch. Nun kann das Testverfahren in der Beschreibungssprache notiert werden:

$$\{\uparrow(w0); \uparrow(r0); \uparrow(w1); \uparrow(r1); \} \quad (2)$$

Es handelt sich um den SCAN-Algorithmus aus Tabelle 2 mit positiver Adressrichtung.

Es wird ein Checkerboard-Algorithmus simuliert:

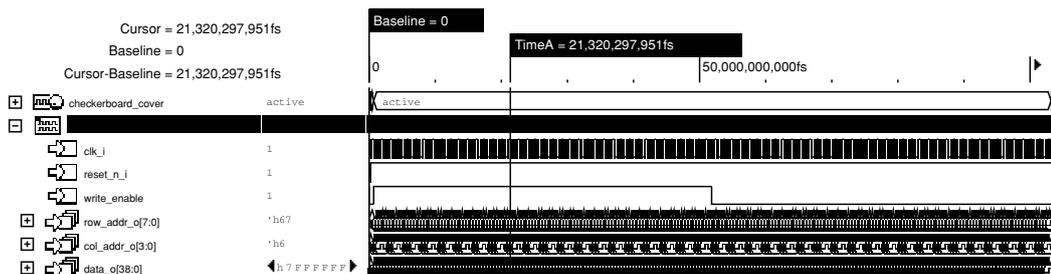


Abbildung 12: Simulation eines Checkerboard Tests.

Das *write\_enable* Signal lässt folgende Sequenz vermuten:

*write* → *read*

Die geschriebenen Daten und die Adressrichtung lassen sich bei einer Vergrößerung des Impulsdigramms erkennen:

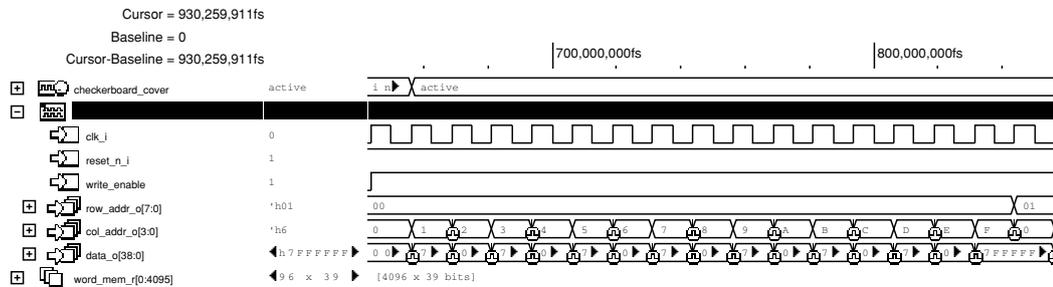


Abbildung 13: Anfang des Checkerboard-Tests.

Es handelt sich erneut um eine fast-column Adressierung. Es werden abwechselnd die Wörter '0<sup>15</sup> und '1 geschrieben:

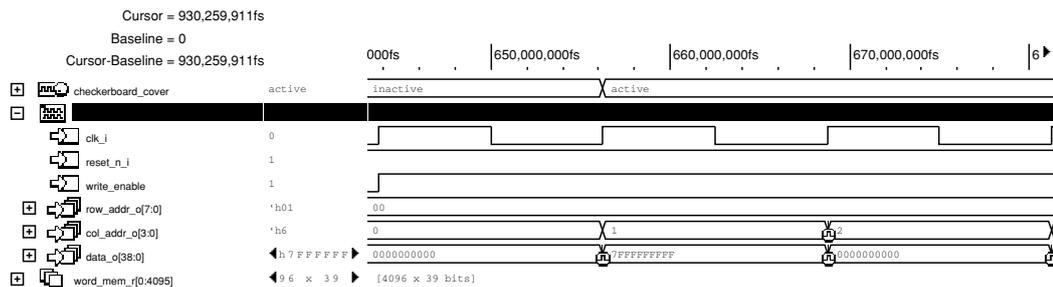


Abbildung 14: Geschriebene Wörter beim Checkerboard Test.

Der logische RAM-Inhalt zeigt, dass sich das logische und physikalische Memory unterscheiden, da es ohne eine Information über den physikalischen Aufbau des Memorys unmöglich ist, aus dem Inhalt das Checkerboard zu interpretieren. Hier zeigt sich das zuvor diskutierte Problem, dass das Memory-Scrambling für die Umsetzung der Testverfahren beachtet werden muss, in der Praxis.

<sup>15</sup>Für die Beschreibung von binären Strings wird die SystemVerilog Schreibweise verwendet; '0 ist das Wort mit der dynamischen Wortlänge  $x'h000\dots$ , '1 das Wort mit der dynamischen Wortlänge  $x'hFFF\dots$

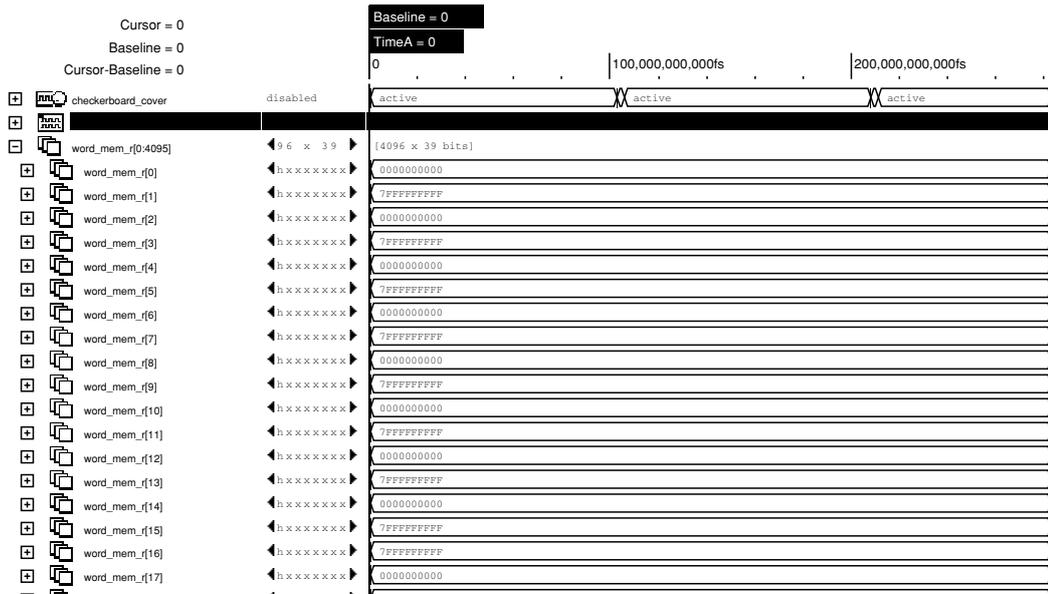


Abbildung 15: RAM-Inhalt nach Checkerboard-Schreibphase.

Der gezeigte Checkerboard-Algorithmus lässt sich wie folgt mit den neu eingeführten Operanden beschreiben:

$$\{\uparrow [0 : 2 : ](wrA), \uparrow [1 : 2 : ](wrB); \uparrow [0 : 2 : ](rrA), \uparrow [1 : 2 : ](rrB); \} \quad (3)$$

Die Parameter A,B haben die Werte: A='0, B='1. Die Länge der Wörter A,B ist dynamisch und hängt von der Breite<sup>16</sup> des physikalischen Memorys ab. Erst über eine Scrambling-Information aus der Memory-Spezifikation können die Wörter für das physikalische Memory, die in Abbildung 14 und 15 geschrieben werden, bestimmt werden. Der hier aufgeführte Ausdruck ist für alle Memories gültig.

<sup>16</sup>y-Dimension

## 4 Entwicklung einer Verifikationsumgebung für einen Memory-Built-In-Self-Test

In diesem Kapitel wird ein Testbenchkonzept für einen MBIST entwickelt. Anschließend wird untersucht, aus welchen Bestandteilen SVA-Properties bestehen müssen, um die Komponenten eines MBIST zu verifizieren.

### 4.1 Testbenchkonzept

Für die Verifikation des MBIST wird eine geeignete Testbench entwickelt. Im folgenden wird ein Beispiel für eine generische Testbench in der formalen, Property-basierten Verifikation gezeigt.

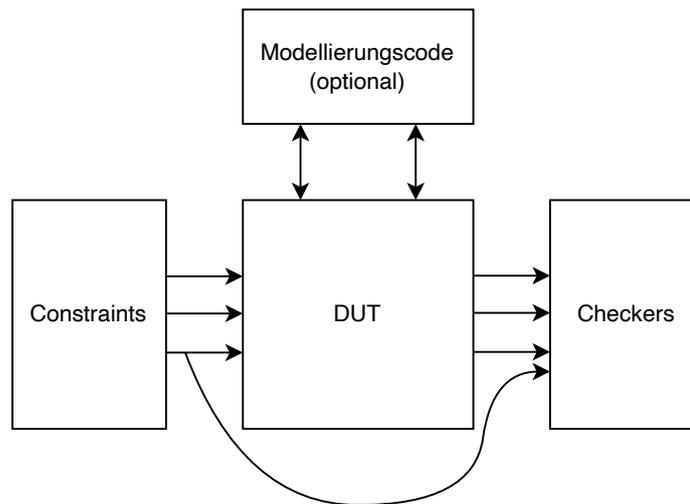


Abbildung 16: Grundlegender Aufbau einer Testbench für die formale, Property-basierte Verifikation nach [17].

Die Constraints sind SVA *assume*-Statements, die den Wertebereich der Eingänge des DUTs beschränken. Beim MBIST muss das Kontrollinterface constrained werden, damit der MBIST definitiv in den Testzustand überführt wird. Einige wichtige Signale sind in der folgenden Tabelle aufgelistet.

Tabelle 3: Constraining des MBIST-Kontrollinterfaces.

Signal	Wert
mem_bypass	1'b0
scan_enable_i	1'b0
scan_mode_i	1'b0
ijtag_reset	1'b1
ijtag_ce	1'b0
ijtag_sel	1'b0
ijtag_si	1'b0
ijtag_tck	1'b0
ijtag_ue	1'b0

Der DUT-MBIST hat neben einem parallelen Interface ein Internal-JTAG (IJTAG) Debug-

Interface, das hier jedoch nicht weiter berücksichtigt werden soll. Damit das JTAG Interface während der formalen Verifikation nicht den Zustand der internen Zustandsmaschine ändern kann, werden die JTAG Signale jeweils auf 1'b0 constrained<sup>17</sup>.

Zudem muss für die Property-basierte Verifikation ein Checker-Modul entwickelt werden, das wie ein Monitor die Ausgänge des DUTs prüft und das Verhalten mit einer Referenz vergleicht. Die Referenz wird in den Properties beschrieben und im nächsten Abschnitt entwickelt. Die Checkers sind SVA *assert*-Statements. Für die *Antecedents* der *assert*-Statements müssen auch die Kontrollsignale am Eingang des DUTs berücksichtigt werden, um eine Sampling-Bedingung für ein Property festzulegen.

Falls das DUT weitere Modulinstanziierungen wie beispielsweise Memory Instanzen beim MBIST benötigt, werden diese mit zusätzlichem Modellierungscode beschrieben. Die Aufteilung von Constraints und Asserts in zwei Module ist optional. Die Properties werden meist in einem *bind*-Modul instanziiert. Dort werden auch mögliche Parameter gesetzt und die Interface-Signale der Properties denen des DUTs zugewiesen.

Alternativ können die Properties auch für die Simulation verwendet werden. Hier werden die Constraints durch einen Treiber ersetzt, in dem die Stimuli explizit festgelegt werden. Die Stimuli sind hierbei Sequenzen auf dem Konfigurationsinterface zur Konfiguration eines definierten Testverfahrens.

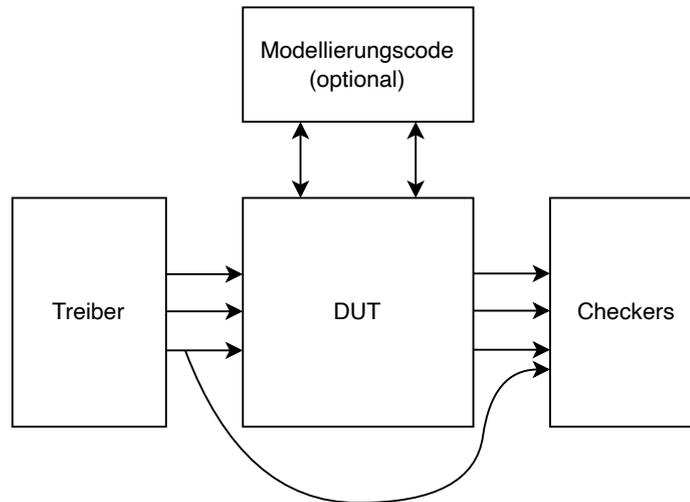


Abbildung 17: Grundlegender Aufbau einer Testbench für die simulative, Property-basierte Verifikation.

Die Constraints aus Tabelle 3 sind hier nicht nötig, da der Treiber nur die definierten Signaländerungen treibt. Der Rest bleibt identisch wie bei der formalen, Property-basierten Verifikation. Selbst das *bind*-Modul, das für die Instanziierung und Verknüpfung der Property-Module verwendet wird, kann in der Simulation eingelesen werden. Daher macht es für die Property-Entwicklung keinen großen Unterschied, welches Lösungsverfahren, formal oder simulativ, für die Verifikation verwendet wird.

<sup>17</sup>Low-aktive Resets müssen auf 1'b1 constrained werden.

## 4.2 Entwicklung von Properties

Nun werden die Properties für das Checker-Modul entwickelt. Das Checker-Modul wird eingangsseitig an das Kontroll- und ausgangsseitig an das Memory-Interface des MBIST gebündelt. Eingangsseitig des MBIST liegt auch noch die Rückführung des Memory Daten-Ausgangs. Die Daten, die aus dem Memory gelesen werden, sind jedoch irrelevant für die Verifikation. Das Ziel ist es, die Properties aus Funktionsblöcken zusammenzusetzen. Die Motivation dahinter ist die zukünftige Automatisierung der Testbench-Entwicklung mit einem Script, das die Properties aus bekannten Funktionsblöcken zusammensetzen kann.

### 4.2.1 Funktionsblöcke

Für die Implementierung der Funktionsblöcke eignen sich SVA-Sequenzen als Wrapper, die über SVA-Verzögerungsoperanden verknüpft werden. Die nötigen Sequenzen sind in Tab. 4 aufgeführt.

Tabelle 4: SVA-Sequenzen für Operanden der BSMTA.

Operand	Sequenz	Aktion
↑	fasty_inc	Soweit nicht weiter angegeben, handelt es sich um ein fast-column (fast-y) Adress-Inkrement.
↓	fasty_dec	Soweit nicht weiter angegeben, handelt es sich um ein fast-column (fast-y) Adress-Dekrement.
↕	fasty_dc	Soweit nicht weiter angegeben, handelt es sich um ein fast-column (fast-y) Adress-Inkrement oder Dekrement.
$x \uparrow$	fastx_inc	Es handelt sich um ein fast-row (fast-x) Adress-Inkrement.
$x \downarrow$	fastx_dec	Es handelt sich um ein fast-row (fast-x) Adress-Dekrement.
w0	w0	write_enable und data_o == '0 ('0 (Wort) auf aktuelle Adresse schreiben)
w1	w1	write_enable und data_o == '1 ('1 (Wort) auf aktuelle Adresse schreiben)
r0	r0	!write_enable (und data_i == '0, aber für MBIST Funktion unerheblich) (von aktueller Adresse (0) lesen)
r1	r1	!write_enable (und data_i == '1, aber für MBIST Funktion unerheblich) (von aktueller Adresse (1) lesen)
wx	write_custom(x)	write_enable und data_o == x (x auf aktuelle Adresse schreiben). Dieser parametrisierte Sequenzblock wird für den Fall eingeführt, dass Wörter geschrieben werden, die nicht nur aus 0 oder 1 bestehen.
rx	read_custom(x)	!write_enable (und data_i == x, aber für MBIST Funktion unerheblich) (von aktueller Adresse (x) lesen)
;	reset_cnt	Grundsätzlich kennzeichnet das Ende einer Sequenz einen Reset des Adress-Zählers. Falls die nächste Sequenz mit einem Adress-Inkrement beginnt, muss der Counter auf 0 zurückgesetzt werden, bei einem Adress-Dekrement auf n-1 (n: Anzahl der Adressen).

Die anderen neu eingeführten Symbole werden nicht als SVA-Sequenzen implementiert, sondern bei der Implementierung der Testverfahren berücksichtigt. Hier wurden nur die Symbole ausgedrückt, die direkt mit Memory-Operationen verknüpft werden können. Die genannten Sequenzen werden nun in SVA implementiert. Einige Beispiele werden aufgeführt:

Code 1: SVA-Sequenzen für Funktionsblöcke

```

1 sequence w0;
2   ((data_o == '0) ##0 (write_enable));
3 endsequence

```

```

4
5 sequence r0;
6   (!write_enable);
7 endsequence
8
9 sequence write_custom(custom_word);
10  ((data_o == custom_word) ##0 (write_enable));
11 endsequence
12
13 //counter sequences
14 sequence reset_cnt(reset_value);
15   {row_addr_o, col_addr_o} == reset_value;
16 endsequence
17
18 sequence fasty_inc(op_cnt);
19  (({row_addr_o, col_addr_o} == $past({row_addr_o, col_addr_o})+1)
20   or {row_addr_o, col_addr_o} == 0
21   or {row_addr_o, col_addr_o} == g_num_of_addr-1)
22  ##1 $stable({row_addr_o, col_addr_o})*op_cnt-1;
23 endsequence
24
25 sequence fastx_dec(op_cnt);
26  (({col_addr_o, row_addr_o} == $past({col_addr_o, row_addr_o})-1)
27   or {row_addr_o, col_addr_o} == 0
28   or {row_addr_o, col_addr_o} == g_num_of_addr-1)
29  ##1 $stable({row_addr_o, col_addr_o})*op_cnt-1;
30 endsequence

```

Die Sequenzen für die Daten auf dem Memory-Interface sind trivial und brauchen keinerlei Erläuterung. Die Sequenzen für den Adress-Zähler hingegen bestehen aus drei Fällen, welche ODER-verknüpft werden, da sie alle berücksichtigt werden müssen:

1. Die Adresse, bestehend aus der Reihenadresse an der Most-Significant-Bit (MSB)-Position und Spaltenadresse an der Least-Significant-Bit (LSB)-Position, wurde vom vorherigen Taktzyklus um 1 inkrementiert (Zeile 19) oder dekrementiert.
2. Zuvor hat ein Reset des Adress-Zählers stattgefunden. Der Zähler ist nun auf 0 (Zeile 20/27).
3. Zuvor hat ein Reset des Adress-Zählers stattgefunden. Der Zähler ist nun auf dem Endwert des Adressbereichs  $g\_num\_of\_addr-1$  (Zeile 21/28).

Anschließend ist der Adress-Zähler  $op\_cnt$ -Takte stabil, damit  $op\_cnt$  Datensequenzen abgearbeitet werden können (Zeile 22/29).

Ein ähnliches Schema kann für die *fast-row*-Adressierung verwendet werden. Es muss nur die Zusammensetzung der Gesamtadresse geändert werden. Bei einem binären Zähler ändert sich das LSB während eines Inkrements häufiger als das MSB. Daher muss die Zeilenadresse an der Position des LSB stehen, damit das Verhalten des Adresssignals mit einem einzelnen binären Zähler modelliert werden kann. Die Adresse setzt sich dann wie folgt zusammen:

$$\{col\_addr\_o, row\_addr\_o\}$$

### 4.2.2 Antecedent

In der *Antecedent* eines Properties muss eine Sampling-Bedingung stehen, die beim Anfang eines bestimmten Testverfahrens *wahr* ist. Dabei können folgende Aspekte berücksichtigt werden:

1. Vor jedem Testverfahren läuft eine Konfigurationssequenz auf dem parallelen Interface.
2. Der erste Zyklus nach der Konfigurationssequenz in dem in das Memory geschrieben wird (erstes *write\_enable*) ist der erste Zyklus eines Testverfahrens, denn jeder Test beginnt mit einer Initialisierung<sup>18</sup> des Memorys (vgl. Tabelle 2).
3. Während der Konfigurationssequenz wird das auszuführende Testverfahren ausgewählt. Wird dies berücksichtigt, so ist bekannt, welches Property geprüft werden soll.

Für ein *assert*-Statement muss sichergestellt werden, dass das Property nur auf den ersten Zyklus eines Testverfahrens gesampelt wird, weil das Property sonst überlappend geprüft wird und nachfolgende Prüfversuche fehlschlagen werden. Mit dem SVA-Operator *first\_match()* wird die erste, zur Referenz passende Sequenz auf dem MBIST Memory-Interface ausgewählt und das Problem der überlappenden Prüfversuche ist gelöst. Diese Eigenschaft wird in dem folgenden Timing-Diagramm dargestellt.

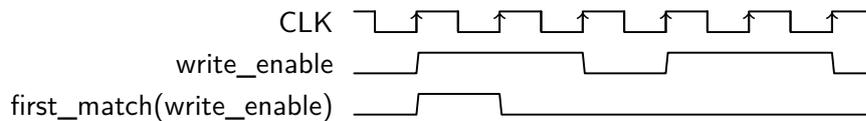


Abbildung 18: *first\_match* erkennt nur den ersten Zyklus von *write\_enable* und bleibt inaktiv, bis die *Consequent* des Property erfüllt ist.

Falls der Verifikateur beachtet, dass das zu prüfende Testverfahren eingestellt ist, reicht es aus, das *write\_enable* in die *Antecedent* zu schreiben. Ein erfolgreicher Testdurchlauf wird beispielsweise in der Simulation in Cadence<sup>®</sup> SimVision mit einem *finished* dargestellt:

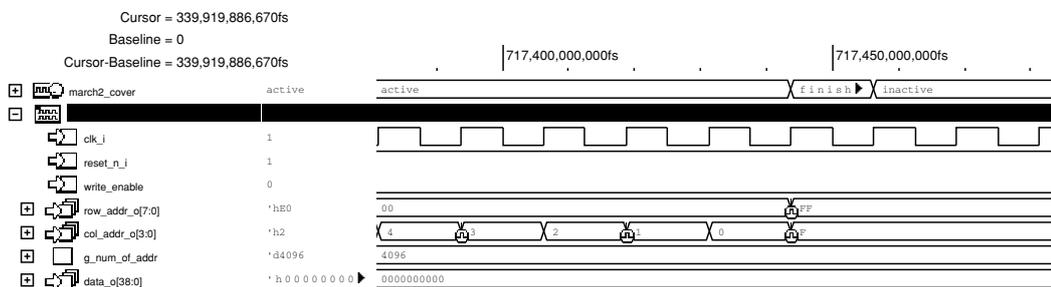


Abbildung 19: Ein Taktzyklus *finished* markiert ein abgeschlossenes Property.

Da die *Antecedent* maßgeblich von dem verwendeten MBIST-Design abhängt, weil sich Konfigurationsinterfaces unterscheiden können und keine generische Sequenz für die Konfiguration eines Testverfahrens angegeben werden kann, genügt hier in erster Linie das

<sup>18</sup>Sicherstellen, dass jede Zelle einen binären Wert 1 oder 0 trägt. Gelöst wird dies durch das Schreiben auf alle Adressen. Eine Initialisierung mit *solid-1* wäre beispielsweise  $\{\uparrow(w1)\}$ .

*write\_enable*.

Um die Verifikation zu automatisieren, können jeweils Wrapper-Signale für den Beginn eines Testverfahrens genutzt werden. Statt die Konfigurationssequenz oder ein anderes Merkmal für den Beginn einer Testsequenz in eine Antecedent zu schreiben, können auch Signale wie beispielsweise *march2\_begin* in der Antecedent stehen, die mittels Modellierungscode außerhalb des Property-Moduls zugewiesen werden:

Code 2: Signal-Wrapper

```
1 module algorithm_signal_wrapper
2 (
3   input wire [4:0] mbist_pi_algo_sel_i,
4   input wire mbist_write_enable,
5   output wire checkerboard_start,
6   output wire march2_start
7 );
8 assign checkerboard_start = (mbist_pi_algo_sel_i == 5'b00011 &&
9   mbist_write_enable == 1'b1) ? 1'b1 : 1'b0;
9 assign march2_start = (mbist_pi_algo_sel_i == 5'b00101 &&
10   mbist_write_enable == 1'b1) ? 1'b1 : 1'b0; //
10 endmodule
```

Zu diesem Zweck werden spezifische Signale des Memory-Interfaces zusätzlich zum *write\_enable* Signal geprüft, um die laufenden Testalgorithmen zu identifizieren und das Sampling von Properties, die inaktive Testalgorithmen verifizieren, zu deaktivieren.

### 4.2.3 Consequent

In der *Consequent* eines Properties muss die Referenz für einen richtigen<sup>19</sup> Testdurchlauf stehen. Die Referenz kann aus den vorher definierten Funktionsblöcken zusammengesetzt werden. Jede Sequenz in der Beschreibungssprache besteht aus den folgenden Elementen:

1. Nur am Anfang der Sequenz: Reset des Adress-Zählers auf Anfangs- oder Endwert des Adressbereichs.
2. Für alle Adressen: Mit Verzögerungsoperand verbundene SVA-Sequenzen für Lese- und Schreiboperanden der Beschreibungssprache.
3. Für alle Adressen: Inkrement oder Dekrement des Adress-Zählers. Der Zähler muss anschließend für die Länge der Sequenz aus Punkt 2 konstant gehalten werden.

### 4.2.4 Beispiele

Um die Zusammensetzung der *Consequent* aus den genannten Funktionsblöcken zu demonstrieren, wird der March C+ (March2) Algorithmus aus Tabelle 2 in SVA implementiert:

<sup>19</sup>zur Spezifikation (beispielsweise einem Ausdruck in der BSMTA) passend.

Code 3: March C+ Implementierung in SVA

```

1  property march2;
2  @(posedge clk_i) disable iff (!reset_n_i)
3  first_match($rose(march2_start)) |->
4  ((reset_cnt(g_start_of_addr) ##0
5  fasty_inc(1)[*g_num_of_addr]
6  and
7  (w0)[*g_num_of_addr])
8  ##1
9  ((reset_cnt(g_start_of_addr) ##0
10 fasty_inc(3)[*g_num_of_addr]
11 and
12 (r0 ##1 w1 ##1 r1)[*g_num_of_addr])
13 ##1
14 ((reset_cnt(g_start_of_addr) ##0
15 fasty_inc(3)[*g_num_of_addr]
16 and
17 (r1 ##1 w0 ##1 r0)[*g_num_of_addr])
18 ##1
19 ((reset_cnt(g_end_of_addr) ##0
20 fasty_dec(3)[*g_num_of_addr]
21 and
22 (r0 ##1 w1 ##1 r1)[*g_num_of_addr])
23 ##1
24 ((reset_cnt(g_end_of_addr) ##0
25 fasty_dec(3)[*g_num_of_addr]
26 and
27 (r1 ##1 w0 ##1 r0)[*g_num_of_addr])
28 ##1
29 ((reset_cnt(g_end_of_addr) ##0
30 fasty_dec(1)[*g_num_of_addr]
31 and
32 (r0)[*g_num_of_addr]
33 );
34 endproperty

```

Der Aufbau des Properties wird beispielhaft an den ersten beiden Sequenzen des March2 erklärt:

$$\uparrow (w0); \uparrow (r0, w1, r1); \quad (4)$$

Mit  $\uparrow$  wird eine positive Adressrichtung festgelegt. Mit  $reset\_cnt(0)$  wird der Adress-Zähler in Zeile 4 dafür einmal auf 0 gesetzt. Die Sequenzen werden jeweils mit  $##0$  getrennt, da sie im gleichen Taktzyklus *wahr* sein müssen. Anschließend wird mit  $fasty\_inc(1)$  der Adress-Zähler inkrementiert (Zeile 5) und einen Taktzyklus lang gehalten. Im gleichen Zyklus wird die SVA-Sequenz  $w0$  ausgeführt (Zeile 7). Beide Sequenzen werden für alle Adressen mit  $[*g\_num\_of\_addr]$  wiederholt. In der nächsten Sequenz ist die Zusammensetzung ähnlich. Statt  $w0$  gibt es drei Operanden  $r0, w1, r1$ . Daher muss der Adress-Zähler mit  $fasty\_inc(3)$  nach dem Inkrement drei Taktzyklen lang gehalten werden (Zeile 10). Die SVA-Sequenzen für  $r0, w1, r1$  müssen nacheinander *wahr* sein und werden daher mit  $##1$  getrennt (Zeile 12).

Es folgt eine beispielhafte Implementierung des Checkerboard-Algorithmus, bei dem Scrambling berücksichtigt werden muss. Wie in Abbildung 6 dargestellt, muss für ein Checkerboard alternierend die Wörter '0 und '1 geschrieben werden, was sich erneut mit den SVA-Sequenzen  $w0, w1, r0, r1$  ausdrücken lässt.

Code 4: Checkerboard Implementierung in SVA

```

1  property checkerboard;
2  @(posedge clk_i) disable iff(!reset_n_i)
3  first_match($rose(checkerboard_start)) |->
4  (reset_cnt(0)
5  ##0 (((fasty_inc(1) and w0) ##1 (fasty_inc(1) and w1))
6  [*g_num_of_cols/2]
7  ##1 ((fasty_inc(1) and w1) ##1 (fasty_inc(1) and w0))
8  [*g_num_of_cols/2])
9  [*g_num_of_rows/2])
10 ##1
11 (reset_cnt(0)
12 ##0 (((fasty_inc(1) and r0) ##1 (fasty_inc(1) and r1))
13 [*g_num_of_cols/2] ##1 ((fasty_inc(1) and r1)
14 ##1 (fasty_inc(1) and r0))[*g_num_of_cols/2])[*g_num_of_rows/2]);
15 endproperty

```

Nach einem Reset des Adress-Zählers auf 0 in Zeile 4 wird in Zeile 5 alternierend  $w0$  und  $w1$  geschrieben. Da dies zwei Elemente in einer Reihe im Memory beschreibt, muss dies nur noch für die Hälfte der Spaltenanzahl wiederholt werden (Zeile 6). Der zweite Term in Zeile 7 startet mit  $##1$  einen Taktzyklus später und beginnt damit in der zweiten physikalischen Memory Reihe. Dort wird statt  $w0, w1$  invertiert  $w1, w0$  geschrieben (Zeile 7), um das Schachbrett-Muster zu erhalten. Da nun zwei vollständige Reihen im physikalischen Memory voll beschrieben sind, kann diese Beschreibung noch  $g\_num\_of\_rows/2$ -mal wiederholt werden, um das gesamte Checkerboard zu beschreiben (Zeile 9). Danach folgen die gleichen Sequenzen mit  $r0, r1$  für die Lesephase (Zeile 11-14).

### 4.3 Bind-Modul

Schließlich wird ein Bind-Modul benötigt, um das Property- und das Wrapper-Modul (Code 2) zu instanzieren und mit der bestehenden Testbench zu verbinden. Im Bind-Modul können auch die Parameter der zu instanzierenden Module definiert werden, was dazu genutzt werden kann, das Property-Modul generisch zu gestalten und alle Variablen in einem Intellectual Property (IP)-spezifischen Bind-Modul zu definieren. Es folgt ein Beispiel für ein Bind-Modul, in dem auch ein Wrapper-Modul instanziiert wird.

Code 5: Bind-Modul

```

1  module mem_sva_bind;
2  bind ifx_tb algorithm_signal_wrapper inst_algorithm_signal_wrapper
3  (
4  .mbist_pi_algo_sel_i(inst_mbist.pi_algo_sel_i),

```

```
5     .mbist_write_enable(inst_mbist.mbist_controller.BIST_WRITEENABLE)
6 );
7 bind ifx_tb mbist_algorithms
8 #(
9     .g_data_width(39),
10    .g_row_addr_width(8),
11    .g_col_addr_width(4),
12    .g_num_of_addr(4096),
13    .g_num_of_rows(256),
14    .g_mux_type(16)
15 )
16 inst_mbist_algorithms(
17     .clk_i(inst_mbist.bist_clk_i),
18     .reset_n_i(inst_mbist.pi_rst_n_i),
19     .data_o(inst_mbist.mbist_controller.BIST_WRITE_DATA),
20     .data_i(inst_mbist.mem1_dram_DO_o),
21     .write_enable(inst_mbist.mbist_controller.BIST_WRITEENABLE),
22     .col_addr_o(inst_mbist.mbist_controller.BIST_COL_ADD),
23     .row_addr_o(inst_mbist.mbist_controller.BIST_ROW_ADD),
24     //wrapper signals
25     .march2_start(ifx_tb.inst_algorithm_signal_wrapper.march2_start),
26     .checkerboard_start(ifx_tb.inst_algorithm_signal_wrapper.
27         checkerboard_start)
28 );
29 endmodule
```

Das Wrapper-Modul empfängt das *write\_enable* Signal und ein Algorithmenkennungssignal vom parallelen Kontrollinterface des MBIST und generiert daraus die für die Algorithmen spezifischen Wrapper-Signale. Die Eingänge des Property-Moduls werden mit den Ein- und Ausgängen des MBIST sowie mit den Ausgängen des Wrapper-Moduls verbunden.

## 5 Automatisierung der Generierung von Assertions

In diesem Kapitel werden einige Rahmenbedingungen für ein automatisiertes Verfahren hergeleitet, die es ermöglichen, SVA-Assertions für einen MBIST zu generieren, wodurch die Verifikation effizienter und schneller durchgeführt werden kann.

### 5.1 Generierung von SVA-Properties

In Abschnitt 4.2.1 wurde die Möglichkeit vorgestellt, modulare Strukturen als Properties zu verwenden. Dies hat den Vorteil, dass die einzelnen Blöcke unabhängig voneinander definiert und dann zusammengesetzt werden können. In diesem Abschnitt wird untersucht, wie diese Blöcke automatisch ermittelt werden können.

#### 5.1.1 Datengenerator

Die in Tabelle 4 auf Seite 27 aufgeführten SVA-Sequenzblöcke  $w0$ ,  $r0$ ,  $w1$ ,  $r1$  werden zur Verifikation des Datenpfades verwendet. Um den richtigen Funktionsblock für eine Schreib- oder Lesesequenz auszuwählen, muss nicht nur der Operand aus der BSMTA ausgewertet, sondern auch das physikalische Mapping der Bits berechnet werden. Dafür werden hier grundsätzlich zwei Verfahren angewendet, die in den nächsten beiden Abschnitten erläutert werden.

#### 5.1.2 Scrambling-Transformation

Bereits in Abschnitt 3.1.1 wurde das Scrambling und seine Auswirkung auf die Coverage von Memory-Testalgorithmen untersucht. Während dort das Scrambling manuell durch eine Analyse der Faltung berücksichtigt wurde, muss für die Automatisierung eine maschinenlesbare Beschreibung des Bitmappings verwendet werden, die als Parameter für die Bestimmung der benötigten Vektoren dient. Dafür eignet sich eine Scrambling-Tabelle, die grundsätzlich eine Lookup-Table (LUT)<sup>20</sup> ist. Mit der LUT werden die Bits im Wort lediglich verschoben. Es finden keine weitere logische Operation statt, weshalb dieser Vorgang hier als Scrambling-Transformation bezeichnet werden kann. Die LUT hat den folgenden Aufbau:

---

<sup>20</sup>Laut [14] wird auch in einigen BISTs Scrambling mit einer LUT realisiert.

Tabelle 5: Ausschnitt der Scrambling Tabelle für das gefaltete Memory in Abbildung 4 auf Seite 16

Tabellenindex	Eingabeadresse	Adresse nach Scrambling
0	0000	0000
1	0001	0100
2	0010	1000
3	0011	1100
4	0100	0001
5	0101	0101
6	0110	1001
7	0111	1101
8	1000	0010
9	1001	0110
10	1010	1010
11	1011	1110
12	1100	0011
13	1101	0111
14	1110	1011
15	1111	1111

Der binäre Wert der Eingabeadresse entspricht dem Reihenindex der Tabelle. Die Scrambling-Tabelle kann daher als lineares Array implementiert werden, indem die binäre oder hexadezimale Eingabeadresse zu einem Integer typkonvertiert und als Index verwendet wird. Da auch die Ausgabeadresse ein Index für ein Array-Element im Ausgabevektor ist, wird auch diese zu einem Integer typkonvertiert.

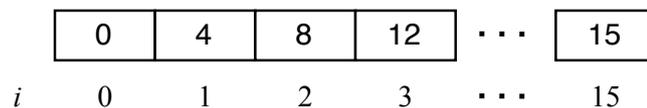


Abbildung 20: Scrambling Tabelle als eindimensionales Array.

Es ist wichtig zu beachten, dass bei der Generierung der Scrambling-Tabelle jedem Bit fortlaufend eine Adresse zugeordnet wird, auch wenn unter Umständen nur Wortzugriffe möglich sind, da sonst eine Transformation nicht funktioniert.

Mithilfe der LUT können die Eingabevektoren, die in der BSMTA beschrieben sind, in die Ausgabevektoren transformiert werden, die nötig sind, um das BSMTA-Bitfeld physikalisch zu erzeugen. Die Transformation wird in Abbildung 21 dargestellt.

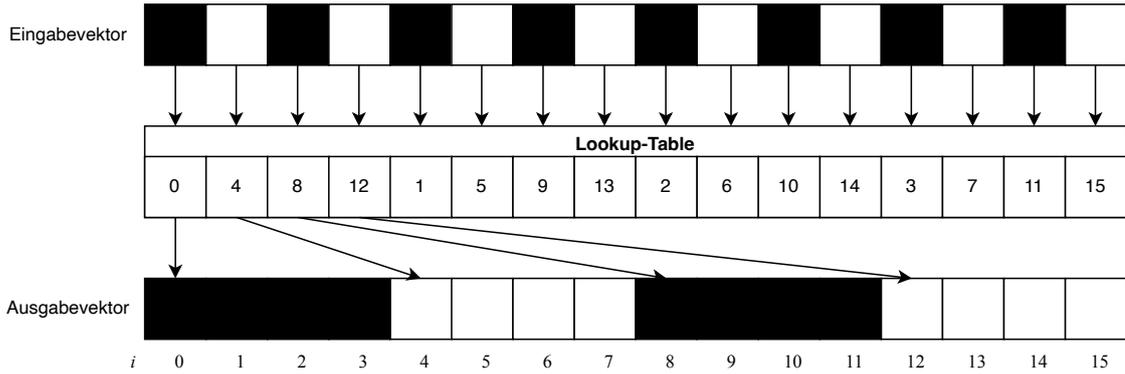


Abbildung 21: Berechnung der nötigen Ausgabevektoren durch eine Scrambling-Transformation mit der Lookup-Table. Einige Pfeile werden zugunsten der Übersichtlichkeit nicht dargestellt.

Da sich Transformationen auch mathematisch als Matrixoperationen darstellen lassen, wird eine allgemein gültige Vorschrift für die Faltung aufgestellt. Die Faltung lässt sich in einigen Fällen mithilfe der *partiellen Transposition*<sup>21</sup> ausdrücken. Die Eingangsmatrix  $E$  ist hierbei für das Checkerboard in Abbildung 4 auf Seite 16 das BSMTA-Bitfeld mit den binären Submatrizen  $A$  und  $B$ :

$$E = \begin{pmatrix} A & A & A & A \\ B & B & B & B \end{pmatrix}, A = \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}, B = \begin{pmatrix} 0 & 1 & 0 & 1 \end{pmatrix} \quad (5)$$

Es wird eine Matrix  $X$  gesucht, dessen binäre Zeilenvektoren die Wörter repräsentieren, die in ein Memory geschrieben werden müssen, um das Bitfeld in der Eingangsmatrix  $E$  zu bilden. Dass die Matrix  $X$  für das genannte Beispiel den folgenden Inhalt haben muss, ist bereits aus Abschnitt 3.1.1 bekannt:

$$X = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad (6)$$

<sup>21</sup>Transponieren der Submatrizen einer Matrix [25].

Die Matrix  $X$  kann gebildet werden, indem die Matrix  $E$  partiell transponiert wird.

$$X = E^{T_B} = \begin{pmatrix} A^T & A^T & A^T & A^T \\ B^T & B^T & B^T & B^T \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \end{pmatrix} \quad (7)$$

Wichtig ist hierbei, dass die Memory-Parameter in die Eingangsmatrix  $E$  einfließen:

- **MUX-Faktor:** Länge der Zeilenvektoren  $A$  und  $B$ .
- **Wortlänge:** Anzahl der Zeilenvektoren  $A$  und  $B$  als Submatrizen in einer Zeile von  $E$ .
- **Reihenanzahl:** Anzahl der Zeilen in  $E$ .

Die unbekannte Matrix  $X$  kann mit der partiellen Transposition (7), die auf eine für das Memory angepasste Matrix  $E$  angewandt wird, ermittelt werden. Diese Transformation ist jedoch auf die Faltung beschränkt. Eine LUT ist vielseitiger, wenn auch ein mathematischer Ansatz eleganter ist, weil mit diesem direkt aus einer Eingangsmatrix die Ausgangsmatrix  $X$  berechnet werden kann und eine LUT sehr groß werden kann.

### 5.1.3 Scrambling-Bypass

Für die Verifikation einiger Algorithmen ist ein Bypass für das Scrambling erforderlich. Soll ein Term in der BSMTA ausschließlich das Verhalten des Memory-Interfaces des MBIST und nicht das physikalische Bitfeld beschreiben, so muss der Bypass aktiv sein. Bisher wurden nur Algorithmen diskutiert, bei denen Scrambling aufgrund von Patterns im Data-Background berücksichtigt werden muss oder unwesentlich für die Verifikation ist, da in einem Testzyklus nur solid-0 oder solid-1 geschrieben wird. Eine Alternative dazu sind Algorithmen, deren primärer Nutzen nicht ein Memory-Test mit einem bestimmten physikalischen Data-Background ist, sondern einen Peripherietest darstellen, für den nur der logische Speicherinhalt von Bedeutung ist. Beispielsweise können Error-Correcting-Codes (ECC)-Module durch die Initialisierung eines RAM durch den MBIST mit ECC-korrekten Wörtern verifiziert werden. Bereits durch das Fehlen von *read*-Sequenzen in der Testbeschreibung solcher Algorithmen zeigt sich, dass es sich um einen Peripherietest oder eine bloße Initialisierung handelt und der MBIST keine Daten kompariert. Ein einfacher Initialisierungsalgorithmus mit gleichen Wörtern lässt sich mit aktiviertem Scrambling-Bypass trivial beschreiben:

$$\{\uparrow (wA); \} \quad (8)$$

Der Scrambling Bypass wird implementiert, indem die Matrixoperationen, die das in der BSMTA beschriebene Bitfeld  $A$  in das physikalische Bitfeld  $B$  transformieren, umgangen

werden und  $B = A$  zugewiesen wird.

## 5.2 Memory Parameter

Für die MBIST-Verifikation sind die logischen und geometrischen Eigenschaften des Halbleiterspeichers relevant. In Abbildung 22 ist ein vereinfachtes Ersatzschaltbild eines gefalteten RAM-Moduls dargestellt, aus dem die wesentlichen Parameter hervorgehen.

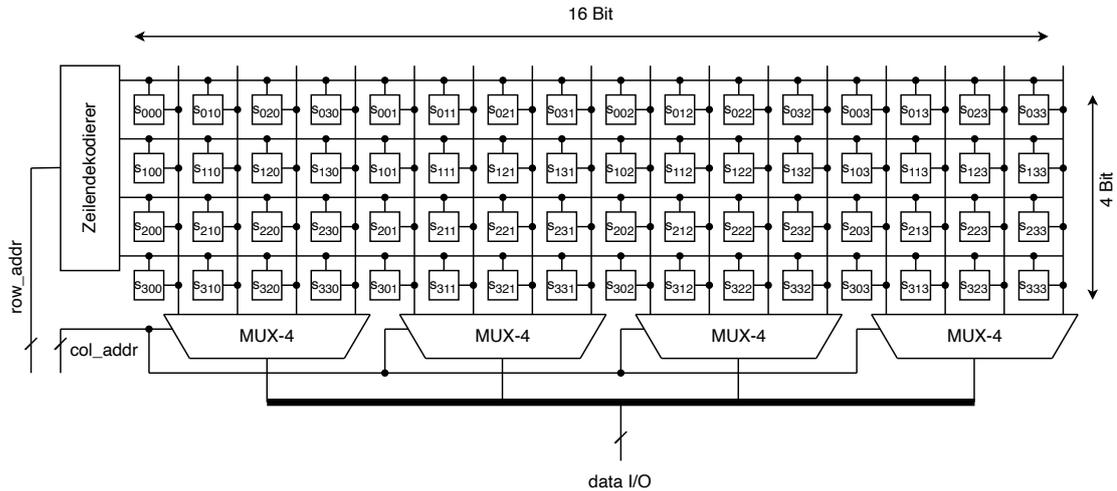


Abbildung 22: Beispielhafter logischer Aufbau eines RAM-Moduls mit angewandter distributiver Faltung.

Die Abbildung zeigt eine Matrixanordnung von Bitzellen mit 16-Bit Breite und 4-Bit Tiefe. Die Adresse kann in Reihenadresse  $row\_addr$  und Spaltenadresse  $col\_addr$  aufgeteilt werden. Die Breite der Reihenadresse  $w_r$  ist abhängig von der Anzahl der Reihen  $n_r$ . Dabei gilt die folgende Mindestanforderung an die Breite:

$$w_r \geq \log_2 n_r \quad (9)$$

Die Breite der Spaltenadresse  $w_c$  ist hingegen abhängig von dem für die Faltung verwendeten Multiplexer-Typ. Für einen Typ- $n$  Multiplexer gilt die folgende Mindestanforderung an  $w_c$ :

$$w_c \geq \log_2 n \quad (10)$$

Das Datensignal ergibt sich aus den Ausgängen der Multiplexer. Bei  $n_m$  Typ- $n$  Multiplexern werden  $n$  Wörter nebeneinander geschrieben, die jeweils die Breite  $n_m$  haben. Aus den Breiten der Signale  $row\_addr$  und  $col\_addr$  können die unbekanntenen physikalischen Dimensionen berechnet werden. Idealerweise wird jedoch eine Quelle verwendet, in der diese genau aufgeführt sind. Es existieren mehrere Parameterquellen, die kurz vorgestellt werden.

- **.MemLib-Datei:** Die Parameter für das Property-Modul, sowie den Assertion-Generator sind im Wesentlichen die Parameter der Memory Makros, die auch für die Konfiguration des MBIST-Generators herangezogen werden. Das Memory Makro ist hierbei eine RTL-Modellierung eines RAM. Das abstrakte Modell bildet die grundsätzliche Funktion des Halbleiterspeichers ab und verfügt über dieselben Schnittstellen wie sein reales Vorbild. Da die Modellierung bereits für die Verifikation der

RAM-Module verwendet wird, wird hier angenommen, dass die Parameter der Makrokonfiguration korrekt sind. Da der MBIST nicht manuell geschrieben, sondern mit einem Generator generiert wird, existiert eine Konfiguration mit den benötigten Memory-Parametern, aus der automatisiert die Parameter extrahiert werden können. Diese liegt in einem proprietären Format vor, das hier als *.MemLib*-Format bezeichnet wird.

- **RTL des MBISTs:** Weitere Quellen für einige Parameter, die stattdessen verwendet werden können, sind die RTL-Dateien des MBISTs. Diese können automatisch durchsucht werden. Dabei helfen standardisierte Signalnamen, die in einer professionellen Entwicklungsumgebung üblich sind. Für die Durchsuchung der RTL-Dateien wird im Bestfall ein Parser für die jeweilige Hardware-Beschreibungssprache verwendet. In Python eignet sich dafür beispielsweise die PyVerilog Bibliothek [18]. Der Parser speichert die Module und Signalen in einem Abstract-Syntax-Tree (AST). Der AST kann leicht traversiert werden, was die Suche nach bestimmten Signalen erleichtert und bei einem Fund die Extraktion weiterer Parameter wie Signalbreiten ermöglicht. Dies hat einen Vorteil gegenüber der anderen vorgestellten Variante:

- Die Signalthierarchie ergibt sich aus der Position im AST. Diese wird im Bind-Modul benötigt, um eine Signalzuweisung der generischen Signale aus dem Property-Modul zu den spezifischen Signalen des DUT durchzuführen.

Andererseits fehlen einige Parameter, die in der ersten Quelle vorhanden sind. Dazu zählen:

- Dimensionen des Memorys
- Informationen über Scrambling (Faltung)

Da diese Parameterquelle begrenzte Informationen enthält und gleichzeitig sehr komplex ist und eine manuelle Überprüfung der Ausgabe erfordert, weil beispielsweise mehrere Clock-Signale<sup>22</sup> von einem MBIST verwendet werden, sollte sie nur verwendet werden, wenn keine andere Quelle verfügbar ist.

Die gefundenen Parameter werden genutzt, um die Generierung der LUT und die Scrambling-Transformation zu steuern. Außerdem werden diese im Bind-Modul eingesetzt, um die Parameter des Property-Moduls zu definieren.

---

<sup>22</sup>Unter anderem für das Memory-Interface, IJTAG und parallele Konfigurationsinterfaces werden verschiedene Clock-Signale verwendet.

## 6 Entwicklung eines Software-Systems

### 6.1 Anforderungen

Ein Software-System, das SVA-Properties aus gegebenen BSMTA-Termen generiert, muss die folgenden Anforderungen erfüllen, die sich aus den in Abschnitt 5 dargestellten Konzepten ableiten lassen.

1. Das System muss eine Datei, welche die Algorithmen in der BSMTA enthält, einlesen können.
2. Das System muss die Algorithmen in der BSMTA interpretieren und in einer geeigneten Datenstruktur speichern können.
3. Das System muss Bitfelder für die Algorithmen berechnen und in einer geeigneten Datenstruktur speichern können.
4. Das System muss bei der Berechnung einiger Bitfelder eine Scrambling-Transformation durchführen und einen weiteren Pfad für einen Scrambling-Bypass haben.
5. Das System muss bei der Berechnung der Bitfelder die physikalischen und logischen Eigenschaften des Memorys berücksichtigen.
6. Das System muss die physikalischen und logischen Eigenschaften des Memorys aus einer *.MemLib*-Datei einlesen und interpretieren und in einer geeigneten Datenstruktur speichern können.
7. Das System muss bei Nichtvorhandensein der *.MemLib*-Datei eine weitere Möglichkeit zur Eingabe der Memory-Parameter bieten. Die Memory-Parameter sollen in einer permanenten Datenstruktur gespeichert werden und bei einem Neustart der Software nicht verloren gehen.
8. Das System muss Properties in der Hardware-Verifikationssprache SVA schreiben können.

### 6.2 Konzept

Aus den Anforderungen kann ein Konzeptdiagramm entwickelt werden.

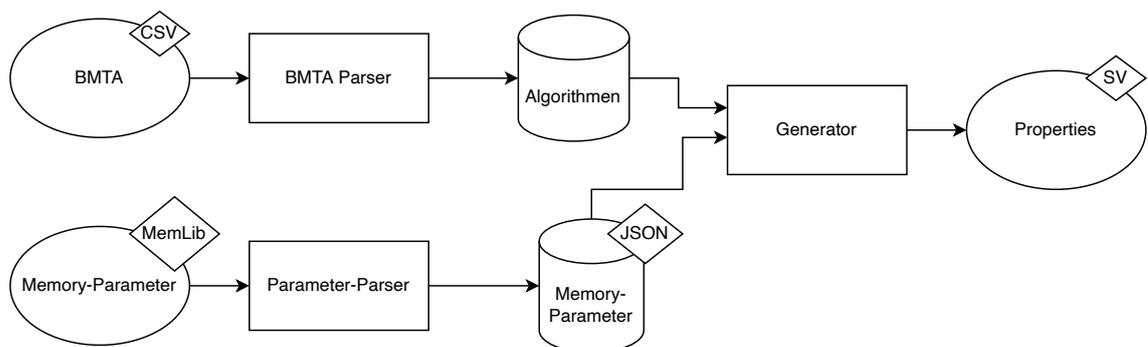


Abbildung 23: Konzept für ein Software-System zur vollautomatisierten Entwicklung von Properties für die Verifikation der Testalgorithmen-Sequenzen auf dem MBIST Memory-Interface. Mögliche Datenformate werden am Rand dargestellt.

Zunächst muss die BSMTA interpretiert werden. Dazu wird ein entsprechender Parser entwickelt, der die Syntax der Sprache unterstützt und daraus neue Datenstrukturen generieren kann. Die BSMTA-Terme für die Testalgorithmen werden in einer Comma-Separated-Values (CSV)-Datei gespeichert. Eine geeignete Datenstruktur wird später definiert. Zusätzlich sind für die Generierung von Properties in der Hardware-Verifikationssprache SVA die Memory-Parameter notwendig. Diese müssen ebenfalls aus einer Datenstruktur geparkt werden. Hierfür bietet sich, wie in Abschnitt 5.2 diskutiert, eine MemLib-Datei an, weil diese für das DUT bereits existiert.

### 6.3 Programmiersprache

Als Programmiersprache wurde Python gewählt. Python hat sich als ideale Programmiersprache für Softwareprojekte etabliert, bei denen Daten analysiert, ASTs konstruiert und neue Sprachkonstrukte generiert werden müssen. Python zeichnet sich vor allem durch seine hohe Interpretierbarkeit aus, da der Code leicht lesbar und verständlich ist. Darüber hinaus ist Python eine dynamisch typisierte Sprache, die keine expliziten Typdeklarationen erfordert [3] und relativ kurze Codezeilen ermöglicht. Diese Eigenschaften ermöglichen es dem Programmierer, schnell und effizient zu arbeiten, ohne sich um technische Aspekte wie Speicherverwaltung kümmern zu müssen. Ein weiterer Vorteil von Python ist die umfangreiche Bibliothek an Modulen und Paketen, die zur Verfügung steht. Insbesondere für die Datenverarbeitung und die Konstruktion von ASTs bieten die Python-Bibliotheken eine große Auswahl an Funktionen und Werkzeugen zur Unterstützung der Datenanalyse, wie beispielsweise Parser und Mathematikpakete. Diese Bibliotheken ermöglichen es dem Entwickler, komplexe Datenstrukturen effizient zu erstellen, zu verwalten und zu analysieren, was die Entwicklungszeit und die Fehleranfälligkeit reduziert.

### 6.4 Frontend

Das Frontend umfasst einige Parser für die Eingaben des Software-Systems, sowie Writer für Ausgaben. Falls eine Datenstruktur nicht als Eingabe oder Ausgabe unterstützt wird, kann diese leicht durch den Aufruf alternativer Parserfunktionen oder Writer ergänzt werden.

#### 6.4.1 Grammatikdefinition

Die Grammatik der erweiterten BSMTA kann in der Backus-Naur Form (BNF) beschrieben werden. Bei der BNF handelt es sich um eine Metasprache<sup>23</sup>. In diesem Kontext ist die Algorithmenbeschreibungssprache die Objektsprache<sup>24</sup>.

Das Konzept der BNF ist es, Bestandteile der Objektsprache in Symbole zu unterteilen und Sequenzen von Symbolen zu sogenannten *metalinguistischen Variablen* zusammenzufassen [12]. Die Symbole können einzelne Zeichen oder ganze Wörter der Sprache sein. Wichtig ist, dass eine Symbolsequenz immer nur eine, vom Kontext unabhängige Bedeu-

---

<sup>23</sup>“Sprache über eine Sprache”[20]

<sup>24</sup>“Gegenstand der Metasprache”[20]

tung hat, da die BNF nur kontextfreie Grammatiken<sup>25</sup> beschreiben kann [20].

Zunächst werden die Symbole als metalinguistische Variablen definiert:

$\langle \text{blockstart} \rangle ::= \{$   
 $\langle \text{blockend} \rangle ::= \}$   
 $\langle \text{opblockstart} \rangle ::= ($   
 $\langle \text{opblockend} \rangle ::= )$   
 $\langle \text{opend} \rangle ::= ;$   
 $\langle \text{opsep} \rangle ::= ,$   
 $\langle \text{increment} \rangle ::= \uparrow$   
 $\langle \text{decrement} \rangle ::= \downarrow$   
 $\langle \text{dontcare} \rangle ::= \updownarrow$   
 $\langle \text{integer} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$   
 $\langle \text{repeat} \rangle ::= ^ \langle \text{integer} \rangle$

Dazu kommen die neuen und angepassten Symbole für die Spracherweiterung:

$\langle \text{incblockstart} \rangle ::= [$   
 $\langle \text{incblockend} \rangle ::= ]$   
 $\langle \text{incsep} \rangle ::= :$   
 $\langle \text{write} \rangle ::= \text{w} [\text{r}] ((0 \mid 1) \mid (\text{A} \mid \text{B} \mid \text{C} \mid \text{D} \mid \text{E} \mid \text{F} \mid \text{G} \mid \text{H} \mid \text{I} \mid \text{J} \mid \text{K} \mid \text{L}$   
 $\mid \text{M} \mid \text{N} \mid \text{O} \mid \text{P} \mid \text{Q} \mid \text{R} \mid \text{S} \mid \text{T} \mid \text{U} \mid \text{V} \mid \text{W} \mid \text{X} \mid \text{Y} \mid \text{Z} \mid \text{a} \mid \text{b} \mid \text{c}$   
 $\mid \text{d} \mid \text{e} \mid \text{f} \mid \text{g} \mid \text{h} \mid \text{i} \mid \text{j} \mid \text{k} \mid \text{l} \mid \text{m} \mid \text{n} \mid \text{o} \mid \text{p} \mid \text{q} \mid \text{s} \mid \text{t} \mid \text{u} \mid \text{v} \mid \text{w}$   
 $\mid \text{x} \mid \text{y} \mid \text{z}))$   
 $\langle \text{read} \rangle ::= \text{r} [\text{r}] ((0 \mid 1) \mid (\text{A} \mid \text{B} \mid \text{C} \mid \text{D} \mid \text{E} \mid \text{F} \mid \text{G} \mid \text{H} \mid \text{I} \mid \text{J} \mid \text{K} \mid \text{L}$   
 $\mid \text{M} \mid \text{N} \mid \text{O} \mid \text{P} \mid \text{Q} \mid \text{R} \mid \text{S} \mid \text{T} \mid \text{U} \mid \text{V} \mid \text{W} \mid \text{X} \mid \text{Y} \mid \text{Z} \mid \text{a} \mid \text{b} \mid \text{c}$   
 $\mid \text{d} \mid \text{e} \mid \text{f} \mid \text{g} \mid \text{h} \mid \text{i} \mid \text{j} \mid \text{k} \mid \text{l} \mid \text{m} \mid \text{n} \mid \text{o} \mid \text{p} \mid \text{q} \mid \text{s} \mid \text{t} \mid \text{u} \mid \text{v} \mid \text{w}$   
 $\mid \text{x} \mid \text{y} \mid \text{z}))$

Das Zeichen ‘r’ wird in den read / write Ausdrücken verboten, um Konflikte mit dem Row-Operanden zu vermeiden.

Anschließend werden die Syntaxregeln für die Symbolsequenzen formuliert. Diese lassen sich übersichtlich in rekursiven Railroad-Diagrammen darstellen:

<sup>25</sup>Ein Beispiel für eine kontextabhängige Grammatik ist die Grammatik der Sprache Deutsch. Das Wort “umfahren” hat beispielsweise eine kontextabhängige Bedeutung.

Tabelle 6: Syntax-Diagramme für die BSMTA.

Symbol	Diagramm
expression	$\leftarrow \langle \text{blockstart} \rangle - \langle \text{action\_seqs} \rangle - \langle \text{opend} \rangle - \langle \text{blockend} \rangle \rightarrow$
action_seqs	
action_seq	$\leftarrow \langle \text{inc\_op} \rangle - \langle \text{inc\_options} \rangle - \langle \text{opblock} \rangle \rightarrow$
inc_op	
opblock	$\leftarrow \langle \text{opblockstart} \rangle - \langle \text{rw\_ops} \rangle - \langle \text{opblockend} \rangle \rightarrow$
rw_ops	
rw_op	
inc_options	$\leftarrow \langle \text{incblockstart} \rangle - \langle \text{integer} \rangle - \langle \text{incsep} \rangle - \langle \text{integer} \rangle - \langle \text{incsep} \rangle - \langle \text{integer} \rangle - \langle \text{incblockend} \rangle \rightarrow$

Damit ist die Grammatik definiert. In der Parserentwicklung würden sich mögliche Syntaxkonflikte zeigen.

#### 6.4.2 Parser für die Beschreibungssprache

Die Algorithmenbezeichnungen und der zugehörige BSMTA-Term werden in einer CSV-Datei gespeichert. In der Python Standardbibliothek ist das CSV-Paket enthalten, mit dessen `csv.reader()` Funktion eine CSV-Datei reihenweise eingelesen und als Array zurückgegeben wird. Sobald der BSMTA-Term eingelesen ist, muss dieser interpretiert werden. Dafür werden Lexer und Parser entwickelt.

Aus der zuvor definierten Grammatik lassen sich mit Generatoren Lexer und Parser entwickeln. Dafür wird die Python-Lex-Yacc (PLY) Bibliothek verwendet, die in [1] dokumentiert ist. Der Lexergenerator *Lex* erzeugt aus einem Zeichenstream lexikalische Tokens. Zeichenketten werden mit regulären Ausdrücken gematcht und als Tokens zurückgegeben. Zudem können den Tokens Werte zugewiesen werden. Ein Beispiel für die Implementierung eines wertbehafteten Tokens ist das write:

Code 6: Symboldefinition für write in Python

```

1 def t_WRITE(t):
2     r"wr?([01]|[\^r])"
3     if t.value[1]=="r": #if write row
4         t.value = rw_op(0, True, t.value[2])
5     else:
6         t.value = rw_op(0, False, t.value[1])

```

```
7 return t
```

Der reguläre Ausdruck wird in  $r$  definiert und beschreibt die gleiche Zeichenkette wie das oben aufgeführte Symbol *write* in der Syntax<sup>26</sup> eines regulären Ausdrucks. Der Wert des Tokens steht in der Variable *value*. Vor der Zuweisung ist der Wert von *value* der gefundene String. Normalerweise werden im Lexer die Python built-in Typen als Datentyp für die Tokens verwendet. Weil hier jedoch drei Werte übergeben werden müssen, wurde eine Klasse *rw\_op* definiert, um eine geeignete Datenstruktur verwenden zu können.

Mit dem Parsergenerator *Yacc* können Sequenzen im Tokenstream des Lexers, die den definierten Syntaxregeln folgen, erkannt und abgeleitet werden. Die Grammatikregeln werden in der BNF definiert. In Code 7 wird beispielhaft die Parserregel für *action\_seq* in Tabelle 6 auf Seite 43 gezeigt.

Code 7: Parserregel für *action\_seq*

```
1 def p_action_seq(p):
2     '''action_seq : inc_op inc_options opblock
3     | inc_op opblock'''
4     if len(p) == 4:
5         p[0]=ast.action_seq_node(p[1],p[2],p[3])
6     else:
7         p[0]=ast.action_seq_node(p[1],None,p[2])
```

Eine Aktionssequenz *action\_seq* muss laut Tabelle 6 auf Seite 43 mindestens aus einem Operanden für ein Inkrement oder Dekrement *inc\_op* und einem *opblock* mit Schreib- oder Leseoperanden bestehen, wie es bei dem folgenden Ausdruck der Fall ist:

$$\uparrow (w0) \quad (11)$$

Optional können Optionen für das Inkrement mit *inc\_options* gesetzt werden, wie es bei einem Teilterm des Checkerboards der Fall ist:

$$\uparrow [0 : 2 : ](wA) \quad (12)$$

Die Grammatikregel wird in der BNF notiert (Zeile 2-3). Anschließend müssen Reaktionen für beide Fälle festgelegt werden. Falls ein Teilterm wie (12) eingelesen wird, ist die Grammatikregel wegen des optionalen Symbols *inc\_op* vier Symbole lang. Dieses Kriterium kann genutzt werden, um die Teilterme zu differenzieren. Anschließend kann ein Objekt einer Klasse erzeugt werden, das die Werte der Symbole *inc\_op* und *opblock* und optional *inc\_options* erhält (Zeile 5,7). Dabei wird das  $n$ -te Symbol im BNF-Term mit  $p(n)$  referenziert.

<sup>26</sup>Die Syntax von regulären Ausdrücken kann in [22] nachgelesen werden.

### 6.4.3 Interne Datenstruktur

Laut [1] wird bei einer Ableitung im Parser das Nicht-Terminalsymbol<sup>27</sup> aus weiteren Terminal- und Nicht-Terminalsymbolen zusammengesetzt. Das Nicht-Terminalsymbol erhält die Werte aus den Symbolen der rechten Seite der BNF. Da sich der Wert besser mit einer neu definierten Klasse als mit den Python built-in Typen beschreiben lässt, und die Wertzuweisung die Bildung einer Baumstruktur ermöglicht, wird ein AST mit spezifischen Baumknoten als Datenstruktur gewählt. Die Baumknoten sind die Klassen für die Symbole. Die Klasse *action\_seq\_node* bietet beispielsweise Platz für den Wert von *inc\_op*, optional die Iteratorspezifikation aus *inc\_options* und die Read-Write Operatoren, die wiederum erneut eine eigene Klasse besitzen:

Code 8: Ausschnitt aus der Klasse *action\_seq\_node*. In Python müssen die Datentypen nicht hart implementiert werden, wodurch die Klassenstruktur sehr flexibel ist. Hier können beispielsweise Integer oder Objekte anderer Klassen bei der Definition eines *action\_seq\_node* im Kopf übergeben werden.

```

1  class action_seq_node:
2  def __init__(self, inc_operator, inc_options, rw_op_list):
3      self.inc_operator = inc_operator
4      self.inc_options = inc_options
5      self.rw_op_list = rw_op_list
6      self.__sequenceList = list()
7      self.__actionList = list()
8
9  def setWantedBitfield(self, value):
10     self.__wantedBitfield = value
11
12  def setPhysicalBitfield(self, value):
13     self.__physicalBitfield = value
14
15  def getWantedBitfield(self):
16     return self.__wantedBitfield
17
18  def getPhysicalBitfield(self):
19     return self.__physicalBitfield
20
21  def setActionList(self, list):
22     self.__actionList = list
23
24  def appendActionList(self, action):
25     self.__actionList.append(action)
26
27  def getActionList(self):
28     return self.__actionList
29
30  def setAlgorithmName(self, name): #for debugging purposes only
31     self.__algorithmName = name
32
33  def getAlgorithmName(self):
34     return self.__algorithmName
35

```

<sup>27</sup>Linke Seite der BNF.

```

36 def setSequenceList(self, list): #for managing sva sequences
37     self.__sequenceList = list
38
39 def appendSequence(self, seq):
40     self.__sequenceList.append(seq)
41
42 def getSequenceList(self):
43     return self.__sequenceList

```

Die Klasse erhält festgelegte Schnittstellen, die als *set*- und *get*-Funktionen implementiert werden. Hiermit können Variablen gesetzt und aktualisiert, sowie abgefragt werden. Die Implementierung von exklusiven Schnittstellen unterliegt dem Konzept der Datenkapselung zur Erhöhung der Modularität und Sicherheit [23]. Python bietet dafür private Variablen und Funktionen, die mit einem doppelten Unterstrich im Namen gekennzeichnet sind. Dort sind nur lokale Zugriffe möglich. Eine ähnliche Struktur wird für die weiteren AST-Node Klassen übernommen. Durch die Verschachtelung der Klassen bildet sich eine Baumstruktur ähnlich der in Abbildung 24.

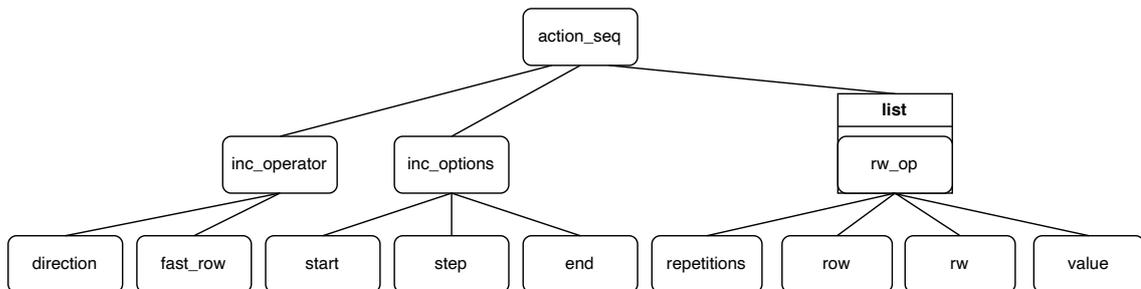


Abbildung 24: Aufbau eines AST für die BSMTA.

Der Baum enthält Knoten für die Grammatikobjekte in Tabelle 6 auf Seite 43.

Der AST wird in der Parserfunktion *AlgoParse* instanziiert und schließlich ausgegeben, in dem der Knoten auf der höchsten Hierarchieebene als Objektverweis der aufrufenden Funktion übergeben wird. Die Ausgabeklasse bei March-Algorithmen entspricht dem Typen *action\_seq\_node*. Bei zusammengesetzten Termen wie dem Checkerboard wird eine Liste mit Objekten des Typs *action\_seq\_node* übergeben. Dies dient unter anderem der Diversifizierung der beiden Arten von Algorithmenbeschreibungen, die unterschiedlich behandelt werden müssen. Das Bitfeld eines Checkerboard Algorithmus wird beispielsweise aufgrund der neu eingeführten Symbole partiell<sup>28</sup> gebildet, während das eines March-Algorithmus von einer Aktionssequenz *action\_seq* vollständig beschrieben wird und dementsprechend in einer Iteration vollständig gebildet werden kann.

Der Vorteil eines AST ist, dass die Struktur leicht traversiert<sup>29</sup> werden kann, was im Backend ausgenutzt wird. Damit ist der AST die ideale Datenstruktur, um das Frontend und Backend miteinander zu verknüpfen. Neue Informationen können leicht sowohl vom

<sup>28</sup>Bei dem Checkerboard Algorithmus werden Reihen mit geraden oder ungeraden Indizes gruppiert und nacheinander geschrieben.

<sup>29</sup>Jeder Knoten (Node) in der Baumstruktur hat sogenannte Äste (Branches), die Verweise auf andere Baumknoten sind. Durch das Folgen dieser Verweise wird der Baum bis zur untersten Hierarchieebene durchlaufen.

Frontend als auch vom Backend als neue Knoten an den AST gehangen werden und je nach Anwendungsfall berücksichtigt oder ignoriert werden.

#### 6.4.4 Externe Schnittstellen und Datenstrukturen

Für die Auswahl sinnvoller Datenstrukturen muss beurteilt werden, ob die Daten permanent, oder nur temporär bei der Ausführung der Software vorhanden sein sollen. Der AST wird bei jedem Parsing neu generiert und hat immer unterschiedliche Strukturen und Variablenwerte. Hier ist eine Speicherung überflüssig und kann sogar zu Speicherproblemen führen. Für jede Algorithmus-Sektion werden Bitfelder berechnet, die so groß sind, wie das beschriebene Memory. Würden die Bitfelder mit dem AST permanent gespeichert werden, so würde die Software große Dateien generieren. Bei kleinen Memories, wie dem hier beispielhaft behandeltem Static-Random-Access-Memory (SRAM), der mit dem DUT-MBIST verbunden ist, ist der Speicherbedarf überschaubar, da es sich lediglich um eine Speichergröße von etwa 20kByte handelt.

Für die Speicherung von Python *dict*-Objekten eignet sich das JavaScript-Object-Notation (JSON)-Format aufgrund der ähnlichen Formatierung der gespeicherten Daten [6, S. 121]. Python Dictionaries sind eine Datenstruktur zur Speicherung von Schlüssel-Wert-Paaren. Die JSON-Notation ist ebenfalls auf die Darstellung von Schlüssel-Wert-Paaren ausgerichtet. JSON-Dateien sind eine der gängigsten Formate für den Austausch von Daten zwischen verschiedenen Programmiersprachen und Plattformen. Sie sind leicht lesbar und können von vielen Programmiersprachen einfach verarbeitet werden. Python unterstützt das Lesen und Schreiben von JSON-Dateien standardmäßig über das in der Standardbibliothek enthaltene Modul "json". Eine Alternative ist die Extensible-Markup-Language (XML), die jedoch nicht so einfach geparkt werden kann.

Die Python JSON-Bibliothek ermöglicht das Lesen von JSON-Dateien in ein *dict*-Objekt, sowie das Speichern eines *dict*-Objekts in eine JSON-Datei ohne Verluste, solange es sich bei dem Datentyp des gespeicherten *Value*-Objekts um einen JSON-Standardtyp<sup>30</sup> handelt.

#### 6.4.5 MemLib-Parser für Parameterextraktion

Die Datei mit dem imaginären Format *.MemLib* enthält alle geometrischen und logischen Parameter des Memorys und ist gegeben, da sie bereits genutzt wird, um den MBIST-Generator zu konfigurieren. Andere Generatoren werden möglicherweise mit anderen Datenstrukturen konfiguriert. Die Parameter sollten jedoch vorhanden und ähnlich strukturiert sein. Die hier verwendete Datei hat den folgenden Aufbau:

Code 9: Ausschnitt aus der MemLib Datei.

```

1 // ColumnMux           : 16
2 MemoryTemplate(sram) {
3 //-----
4 //-- general settings
```

<sup>30</sup>JSON-Standardtypen sind Zahlen, Strings, Arrays und Null [24].

```

5 //-----
6 MemoryType : sram;
7 RomContentsFile : "";
8 NumberOfWords : 4096;
9 NumberOfBits : 39;
10 //-----
11 //--- address/columnmux settings
12 //-----
13 AddressCounter {
14     Function(Address) {
15         LogicalAddressMap {
16             ColumnAddress[3:0] : Address[3:0];
17             RowAddress[7:0] : Address[11:4];
18         }
19     }
20     Function(ColumnAddress) {
21         CountRange [0:15];
22     }
23     Function(RowAddress) {
24         CountRange [0:255];
25     }
26 }
27 BitGrouping : 1;
28 Port(CLK) {
29     Function : Clock;
30     Direction : Input;
31     Polarity : ActiveHigh;
32 }
33 Port(CSB) {
34     Function : Select;
35     Direction : Input;
36     Polarity : ActiveLow;
37 }
38 Port(RWB) {
39     Function : WriteEnable;
40     Direction : Input;
41     Polarity : ActiveLow;
42 }
43 Port(A[11:0]) {
44     Function : Address;
45     Direction : Input;
46 }
47 Port(DI[38:0]) {
48     Function : Data;
49     Direction : Input;
50 }
51 Port(DO[38:0]) {
52     Function : Data;
53     Direction : Output;
54 }

```

Diese proprietäre Datenstruktur definiert eine SRAM-Speicherarchitektur mit einer Größe von 4096 Wörtern (Zeile 8) und einer Datenbreite von 39 Bit (Zeile 9). Der Speicher ist in

256 Reihen (Zeile 24) organisiert und hat eine 16-fache Spaltenmultiplexing-Struktur (Zeile 1). Die Adressierung des Speichers erfolgt über einen 12 Bit breiten Adresseingangsport ( $A[11:0]$ ) (Zeile 43). Die Adressen werden in 4 Bit für die Spaltenadresse (*ColumnAddress*) (Zeile 16) und 8 Bit für die Zeilenadresse (RowAddress) (Zeile 17) aufgeteilt. Die Einstellungen für den Adresszähler werden durch die *AddressCounter*-Funktion festgelegt. Der Spaltenadresszähler hat einen Zählbereich von 0 bis 15 (Zeile 21), und der Zeilenadresszähler hat einen Zählbereich von 0 bis 255 (Zeile 24). Die Daten werden durch  $DI[38:0]$  in den Speicher geschrieben und durch  $DO[38:0]$  aus dem Speicher gelesen. Es gibt auch Ports für Clock (*CLK*), Chip-Select (*CSB*) und Write-Enable (*RWB*), um den Speicherzugriff zu steuern. Die BitGrouping-Einstellung ist auf 1 gesetzt (Zeile 27), was bedeutet, dass die Faltung aktiv ist. Es gibt deutliche Ähnlichkeiten mit JSON. Die Daten sind in Schlüssel-Wert Paaren in der Form *Schlüssel* : *Wert*; strukturiert und können daher optimal in mehreren Dictionaries gespeichert werden.

Es wird ein Parser entwickelt, der die Datei einliest, nach den erforderlichen Parametern filtert und diese in einer geeigneten Datenstruktur speichert. Während für die Entwicklung des BSMTA-Parsers eine komplexe Grammatik entwickelt wurde, kann hier ein einfacherer Parser nach den nötigen Schlüsselwörtern filtern und die zugehörigen Werte in Dictionaries speichern.

## 6.5 Backend

Das Backend erhält den generierten AST aus dem Frontend. Dieser enthält alle Informationen, um SVA-Properties zu generieren. Zunächst müssen diese stufenweise aufbereitet werden. Hierfür wird ausgenutzt, dass der AST manipulierbar ist und Knoten ausgetauscht oder ergänzt werden können.

### 6.5.1 Bitfeld-Erzeugung

Aus einem BSMTA-Term soll ein Bitfeld generiert werden. Da der BSMTA-Term das Bitfeld vollständig beschreibt, wird das zu erzeugende Bitfeld hier als BSMTA-Bitfeld bezeichnet. Jeder mit einem Semikolon abgetrennter Block (Im AST ein oder mehrere *action\_seq* Objekte) beschreibt jeweils ein BSMTA-Bitfeld. In Abschnitt 3.1 wurde erläutert, wie die BSMTA erweitert wurde, um mit ihr einige Bitfelder, die von den Testalgorithmen in einem Memory erzeugt werden, beschreiben zu können. Da für eine Scrambling-Transformation zunächst das BSMTA-Bitfeld als Eingangsvektor benötigt wird, muss dieses von einem Software-Algorithmus erzeugt werden. Der Prozess wird beispielhaft anhand des Checkerboard-Patterns erläutert.

Der folgende BSMTA-Term beschreibt einen Schreibvorgang für ein Checkerboard-Pattern:

$$\{\uparrow [0 : 2 : ](wrA), \uparrow [1 : 2 : ](wrB); \} \quad (13)$$

Zunächst wird im Frontend von dem BSMTA-Parser für diesen Term der in Abbildung 25 dargestellte AST gebildet.

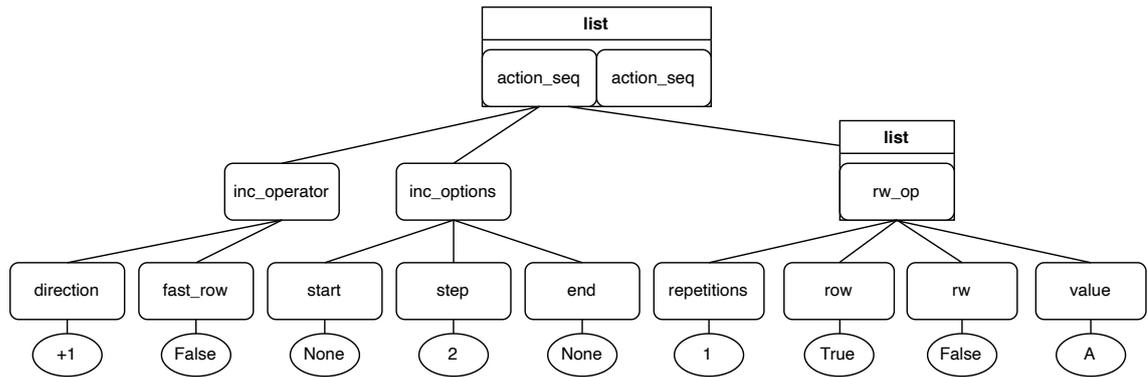


Abbildung 25: AST für einen Checkerboard-Term. Die Branches des zweiten *action\_seq* Objekts werden zugunsten der Übersichtlichkeit nicht dargestellt.

Aufgrund der partiellen Beschreibung des Bitfelds durch das Alternieren, das für das Checkerboard-Pattern nötig ist, ist der Kopf des AST eine Liste von zwei *action\_seq* Objekten. Das erste *action\_seq* Objekt wurde für den Term  $\uparrow [0 : 2 :](wrA)$  gebildet und beschreibt die Reihen mit den geraden Indizes. Das zweite *action\_seq*-Objekt wurde für den zweiten Teil des Terms  $\uparrow [1 : 2 :](wrB)$  gebildet und beschreibt die Reihen mit ungeraden Indizes. Der AST enthält die Informationen, die bereits in der BSMTA vorhanden sind. Mit diesen Informationen wird ein generisches Bitfeld beschrieben, wie es bereits in Abbildung 3 auf Seite 15 demonstriert wurde. Um ein Bitfeld für ein spezifisches Memory zu generieren, müssen die Memory-Parameter bekannt sein. Diese wurden in eine JSON-Datei geparkt, wie es in Abschnitt 6.4.5 beschrieben wurde, oder manuell dort eingetragen. Nun kann das Bitfeld erzeugt werden. Dafür wird eine Funktion implementiert, die den AST manipuliert. Diese ersetzt zunächst die unbestimmten *None* Werte unter *inc\_options* mit den Adressbereich-Parametern aus der JSON-Datei.

Die spezifischen Werte *A* und *B* für das Checkerboard-Pattern aus dem BSMTA-Term müssen vorher noch definiert werden. Dafür wird eine weitere CSV-Datei angelegt.

Tabelle 7: Datenbank für BSMTA-Parameter.

A	4'hA
B	4'h5

Die Werte müssen anschließend an die *y*-Dimension des Memorys angepasst werden. Die Wörter in der Parameter-Datenbank sind unabhängig von den Memory-Parametern, da sie nur so lang wie nötig<sup>31</sup> sind, um ein Pattern zu definieren. So können diese für verschiedene Memories wiederverwendet werden. Die Voraussetzung dafür ist, dass durch Konkatenierung der angegebenen Wörter mit der Breite  $w_w$  eine Reihe mit der Breite  $w_c$  ohne überschüssige oder fehlende Bits gefüllt werden kann. Diese Bedingung lässt sich mathematisch mit Modulo ausdrücken und wird als Python *assert*-Statement<sup>32</sup> implementiert:

$$w_c \bmod w_w = 0 \quad (14)$$

<sup>31</sup>Bei einem Checkerboard genügt eine Hex-Ziffer (4-Bit).

<sup>32</sup>Falls eine *assert*-Bedingung während der Programmausführung nicht eingehalten wird, führt dies zu einem Abbruch mit *AssertionError*.

Falls die Bedingung zutrifft, wird ermittelt, wie viele Wörter eine Reihe füllen:

$$n = \frac{w_c}{w_w} \quad (15)$$

Hex-Strings können durch Konkatenieren erweitert werden und behalten dabei ihr wiederkehrendes Bitmuster. Für ein 16-Bit breites Memory müssen die angegebenen Wörter beispielsweise viermal konkateniert werden:

$$4'hA \rightarrow 16'hAAAA$$

Anschließend kann das nötige Bitfeld für eine Memory-Reihe erzeugt werden, indem der Ausdruck in sein Binär-Äquivalent umgewandelt wird, das als Array mit den Werten 1 und 0 in Python gespeichert werden kann:

$$[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0] \quad (16)$$

Mit dem Reihen-Bitfeld kann nun das BSMTA-Bitfeld partiell gebildet werden. Dafür eignen sich verschachtelte Listen, die jeweils eine Reihe im Memory beschreiben.

Code 10: Partielle Bildung des Bitfelds mit einer for-Schleife.

```

1 for row in range(seq.inc_options.start.value, self.inc_options.end.value,
   ↪ seq.inc_options.step.value):
2     v[row]=rowword

```

Die Schleife durchläuft eine Reihe von Zeilen im Vektor  $v$ , der zuvor initialisiert wurde. Die Schleife beginnt bei der Startposition der  $inc\_options$  der aktuellen Aktionssequenz  $action\_seq$  aus dem AST (angegeben durch  $seq.inc\_options.start.value$ ) und iteriert durch jede  $seq.inc\_options.step.value$ -te Zeile, bis sie die  $seq.inc\_options.end.value$ -te Zeile erreicht. Für jede Zeile wird der Wert  $rowword$  gesetzt, der dem zuvor ermittelten Reihen-Bitfeld (16) entspricht. Der Vektor  $v$  repräsentiert den Speicherbereich, auf den die Aktionssequenz angewendet wird, und jede Zeile im Vektor repräsentiert eine Zeile im Speicher. Durch das Setzen von  $rowword$  für jede Zeile in dem angegebenen Bereich wird die entsprechende Zeile im Speicher mit dem Wert  $rowword$  beschrieben. Da eine Aktionssequenz jeweils die geraden und die andere die ungeraden Zeilen beschreibt, wird das Bitfeld partiell gebildet:

$$[[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0], [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1], \quad (17)$$

$$[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0], [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]]$$

Anschließend werden die verschachtelten Listen zu einem Bitstream linearisiert:

$$[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, \dots, 0, 1] \quad (18)$$

### 6.5.2 Lookup-Table-Erzeugung

Wie in Abschnitt 5.1.2 beschrieben, muss das Software-System in der Lage sein, Daten für aktives Scrambling zu transformieren. Dazu wird eine LUT verwendet, um flexibel für

verschiedene Arten von Scrambling zu sein. Diese muss zunächst aufgestellt werden. Dafür wird die manuell aufgestellte LUT für das bekannte Beispiel nach dem Multiplexer-Prinzip in Abbildung 4 auf Seite 16 analysiert.

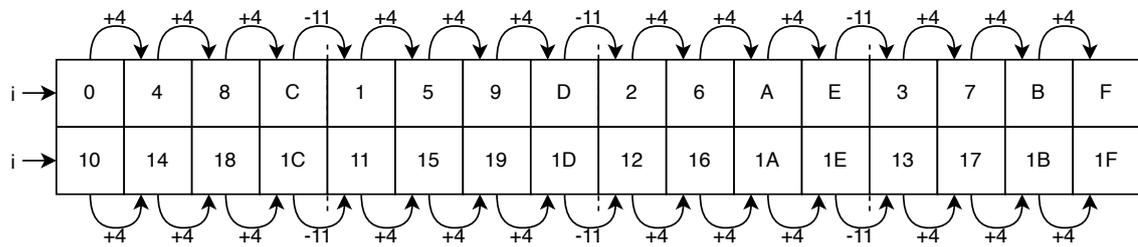


Abbildung 26: Beziehungen der LUT-Objekte.

Folgendes gilt:

1. Jede neue Zeile hat den aktuellen Index  $i$  als Inhalt.
2. An den nachfolgenden Stellen ergibt sich der Inhalt aus dem des Vorgängers summiert mit der Wortbreite  $w_w = 4$ .
3. An den Bitgroup-Boundaries wird dekrementiert. Das Dekrement ist abhängig vom MUX-Typ  $n_m = 4$  und der Wortbreite  $w_w$ , da der MUX-Typ für die Position der Bitgroup-Boundaries verantwortlich ist und vorher mit der Wortbreite inkrementiert wurde.

Nun kann der Inhalt der LUT an einem Index  $L[i]$  allgemein bestimmt werden.

$$L[i] = \begin{cases} i, & \text{falls } i \bmod (n_m * w_w) = 0 \\ L[i-1] - ((n_m - 1) * w_w - 1), & \text{falls } i \bmod n_m = 0 \\ L[i-1] + w_w, & \text{sonst} \end{cases} \quad (19)$$

Dies wird implementiert, indem ein leeres *list*-Objekt erzeugt wird, durch das iteriert wird. An jedem Index wird der zugehörige Wert nach der Rechenvorschrift angehängen:

Code 11: Generation einer LUT für Memories mit Faltung.

```

1 def __get_lut(self, is_folded, addr_cnt, word_width, mux_type):
2     lut = list()
3
4     if is_folded:
5         for i in range(addr_cnt*word_width):
6             if i%(mux_type*word_width)==0:
7                 lut.append(i)
8             else:
9                 if i%mux_type == 0:
10                    lut.append(lut[i-1]-((mux_type-1)*word_width-1))
11                else:
12                    lut.append(lut[i-1]+word_width)
13     return lut

```

Der Rückgabetyt der Funktion ist ein *list*-Objekt mit der Anzahl der Bits des Memorys als Länge. Wird ein anderer Scrambling-Typ verwendet, so muss die Funktion *get\_lut()* manipuliert werden, um eine andere LUT zu erzeugen. An dieser Stelle kann auch ein Import implementiert werden, was sinnvoll ist, wenn eine vordefinierte Scrambling-Tabelle verwendet werden soll.

### 6.5.3 Implementierung der Scrambling-Transformation

Mithilfe der gebildeten LUT kann die Scrambling-Transformation auf das BSMTA-Bitfeld (18), das in Abschnitt 6.5.1 bestimmt wurde, angewendet werden. Dafür muss zunächst ein Array initialisiert werden, das gefüllt werden kann. Dafür wird ein *NumPy*-Array verwendet. NumPy [7] ist eine Python-Bibliothek für numerische Berechnungen mit vielen nützlichen Funktionen für Datenanalyse, wissenschaftliches Rechnen und maschinelles Lernen. NumPy bietet Arrays und Matrizen, die in Python standardmäßig nicht verfügbar sind. Durch das initialisierte Array kann anschließend iteriert werden.

Code 12: Anwendung der LUT.

```

1 def __calc_physical_words(self, wanted_pattern, word_width, addr_cnt,
  ↪ mux_type, is_folded): #unite LUT and wanted bitfield
2     phyw = [0]*word_width*addr_cnt
3     phywnp = np.array(phyw)
4     #apply LUT
5     for i in range(len(self.lookuptable)):
6         phywnp[self.lookuptable[i]] = wanted_pattern[i]
7     #convert to list of words
8     wordlist = list()
9     for word in np.array_split(phywnp, addr_cnt):
10        wlist = word.tolist()
11        bword = ' '.join([str(item) for item in wlist])
12        bword = bword.replace(" ", "")
13        #hexword = self.__binToHexa(bword)
14        wordlist.append(bword)
15    return wordlist

```

Zunächst wird eine Liste *phyw* mit Nullen für jedes Bit im Memory initialisiert. Diese Liste wird dann in ein NumPy-Array *phywnp* umgewandelt. Als nächstes wendet die Funktion die Lookup-Table *self.lookuptable* auf die *wanted\_pattern*-Liste mit dem BSMTA-Bitfeld an. Für jeden Index in *self.lookuptable* wird der entsprechende Wert in *phywnp* auf den entsprechenden Wert in *wanted\_pattern* gesetzt. Das resultierende *phywnp*-Array wird dann in *addr\_cnt* Arrays aufgeteilt, von denen jedes eine Länge von *word\_width* hat, um aus dem Bitfeld schreibbare Wörter zu definieren. Jedes dieser Arrays wird zu einem Bitstring *bword* umgewandelt, indem die Elemente im Array zusammengefügt und die Leerzeichen entfernt werden. Dieser String wird dann zu einer Liste *wordlist* hinzugefügt. Schließlich gibt die Funktion die Wortliste *wordlist* mit den Bitstrings als Ausgabe zurück.

### 6.5.4 Sequenzerkennung

Die Wortliste enthält die Wörter, die geschrieben werden müssen, um das BSMTA-Bitfeld zu erhalten. Für das Checkerboard in Abbildung 4 auf Seite 16 hat die Wortliste eine ähnliche Struktur wie die folgende Liste mit Bitstrings.

[ "1111", "0000", "1111", "0000", "0000", "1111", "0000", "1111" ]

Die Wörter in der Liste könnten nun genutzt werden, um die Checkerboard-spezifische Sequenz auf dem Datensignal des MBIST Memory-Interface zu beschreiben. Da sich diese Liste in der Realität jedoch sehr oft wiederholt, soll ein Algorithmus entwickelt werden, der wiederkehrende Sequenzen erkennt und vereinfacht. Ein Konzept dafür wird in Abbildung 27 dargestellt.

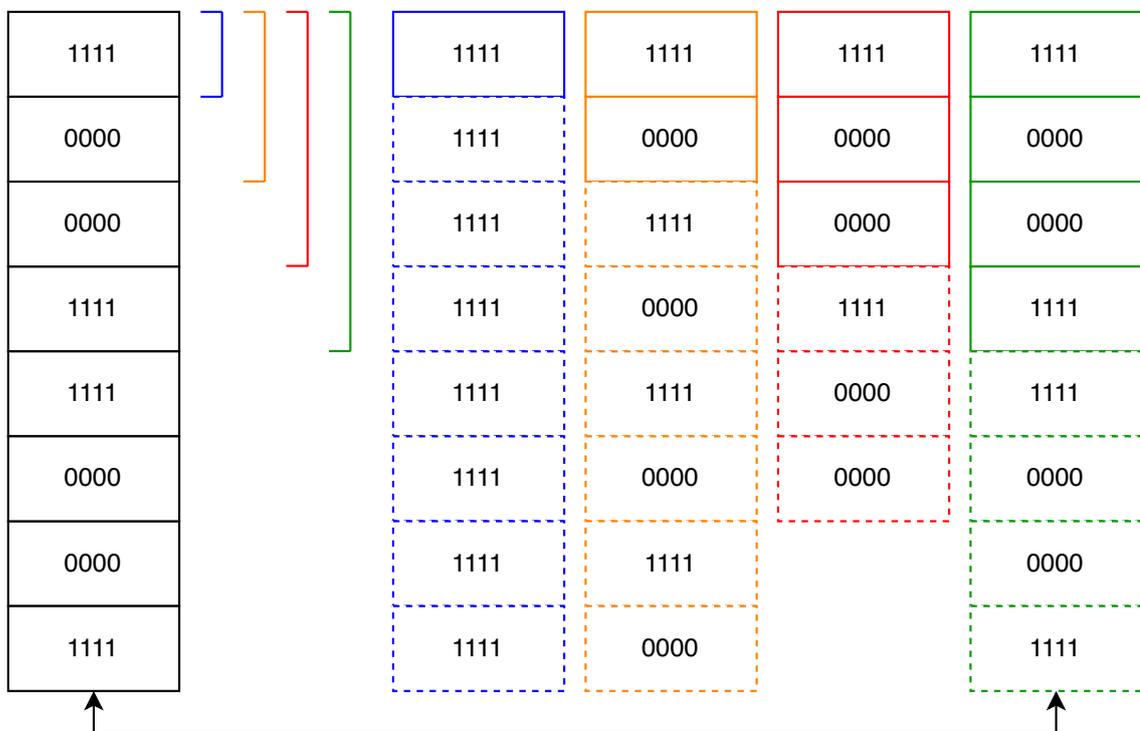


Abbildung 27: Konzept für einen Algorithmus zur Erkennung von wiederkehrenden Sequenzen. Die in grün dargestellte Liste ist äquivalent zu der in schwarz dargestellten Wortliste.

Der Algorithmus soll einen stets größer werdenden Teil der Wortliste wiederholen und dabei auf Äquivalenz der dabei erzeugten Liste zur Wortliste prüfen. Wenn eine Äquivalenz gefunden wird, kann die Wortliste optimiert werden. Der Algorithmus wird wie folgt implementiert:

Code 13: Implementierung des Algorithmus zur Sequenzerkennung.

```

1 def __guess_seq_len(self, seq):
2     for i in range(1, len(seq)):
3         short_seq = seq[:i]*int(len(seq)/i)
4         if short_seq == seq:
5             return i

```

Die Funktion versucht, die Länge des wiederkehrenden Musters in der Sequenz zu bestimmen. Dazu durchläuft sie alle möglichen Mustereinheiten, beginnend bei 1 und endend bei  $len(seq) - 1$ . Für jede Musterlänge wird eine kurze Sequenz erzeugt, indem die ersten  $i$  Elemente von  $seq$  so oft wiederholt werden, dass eine Sequenz mit der Länge von  $seq$  entsteht. Dann wird überprüft, ob diese kurze Sequenz mit der ursprünglichen Sequenz übereinstimmt. Wenn ja, wird das Muster in der gesamten Sequenz wiederholt und die Funktion gibt die Länge des Musters ( $i$ ) zurück. Wird kein Muster gefunden, gibt die Funktion nichts zurück und es wird keine Optimierung durchgeführt.

Anschließend wird das gefundene Muster und die nötige Wiederholungsanzahl in einer geeigneten Datenstruktur gespeichert und in den AST eingefügt.

## 6.6 SystemVerilog-Writer

Aus dem AST soll SystemVerilog-Code generiert werden, der dem in Abschnitt 4.2 definierten Testbench-Konzept folgt. Dafür muss ein Property-Modul mit Assertions sowie ein Bind-Modul generiert werden.

### 6.6.1 Property-Writer

Der AST wurde so modifiziert, dass nun alle notwendigen Daten vorhanden sind, um aus dem ursprünglichen BSMTA-Ausdruck ein SVA-Property zu generieren. Dafür wird eine neue Datei geöffnet, in die geschrieben werden soll:

Code 14: Öffnen einer Datei im Schreibmodus.

```
1 self.sv_out = open(filename, 'w')
```

Der Code öffnet eine Datei mit dem angegebenen Dateinamen im Schreibmodus ('w') und weist sie der Variablen `self.sv_out` zu. Anschließend kann mit der Funktion `write` in die Datei geschrieben werden.

Am Beginn des Property-Moduls befindet sich Boilerplate-Code<sup>33</sup>, weil der Modulkopf und die in Abschnitt 4.2.1 definierten Sequenzen immer gleich sind. Die SVA-Sequenzen werden daher in einer separaten Datei `sec_file` gespeichert, dessen Inhalt hier lediglich kopiert werden muss.

Code 15: Traversierung durch den AST und zutreffende Schreibvorgänge.

```
1 def writeGenerics(self, algorithm_name_list):
2     #header
3     self.sv_out.write("""//Generated with mbist_prop_gen SVA property
4     ↪ generator\n
5     //Ressources: https://confluencewikiprod.intra.infineon.com/x/1UR9Qw
6     \n""")
```

<sup>33</sup>Code, der immer gleichbleibt und unabhängig von Parametern ist und somit immer wieder verwendet werden kann.

```

7
8     self.sv_out.write("""module mbist_algorithms #(
9         parameter g_addr_width,
10        parameter g_data_width,
11        parameter g_num_of_inst,
12        parameter g_row_addr_width,
13        parameter g_col_addr_width,
14        parameter g_num_of_addr,
15        parameter g_mux_type = 16,
16        parameter g_start_of_addr
17    )
18    (
19        input clk_i,
20        input reset_n_i,
21        input [(g_data_width)-1:0] data_o,
22        input [(g_data_width)-1:0] data_i,
23        input write_enable,
24        input [(g_col_addr_width)-1:0] col_addr_o,
25        input [(g_row_addr_width)-1:0] row_addr_o, """)
26
27    for name in algorithm_name_list:
28        self.sv_out.write("\n    input "+str(name).lower()+"_start")
29        if name != algorithm_name_list[-1]:
30            self.sv_out.write(",")
31    self.sv_out.write("\n);\n\n")
32
33    #write sequences
34
35
36    for line in self.seq_file:
37        self.sv_out.write(line)
38    self.seq_file.close()

```

Zunächst werden die in Abschnitt 4.2.1 definierten SVA-Sequenzen aus einer Datei importiert.

Anschließend wird durch den AST traversiert, um die nötigen SVA-Sequenzen aus Abschnitt 4.2.1 zu ermitteln und nötige Parameter einzusetzen. Dies wird beispielhaft anhand der Adressierung demonstriert:

Code 16: Traversierung durch den AST und zutreffende Schreibvorgänge.

```

1 for action_seq_node in tree:
2     self.__words_written = 0
3     if type(action_seq_node) == list: #e.g checkerboard
4         node = action_seq_node[0] #iterate through list
5         __addr_direction = node.inc_operator.direction
6         __fast_row = node.inc_operator.fast_row
7         if __addr_direction == "+1":
8             __reset_val = "g_start_of_addr"
9             if __fast_row:
10                __increment_seq = "fastx_inc"
11            else:

```

```

12     __increment_seq = "fasty_inc"
13     else: #negative direction
14         __reset_val = "g_end_of_addr"
15         if __fast_row:
16             __increment_seq = "fastx_dec"
17         else:
18             __increment_seq = "fasty_dec"
19     #reset
20     self.sv_out.write("\n (reset_cnt("+__reset_val+"))")
21     self.sv_out.write(" ##0 (")

```

Die Schleife iteriert durch den AST und setzt für jede Aktionssequenz *action\_seq\_node* im Baum einige interne Variablen und schreibt dann einen SystemVerilog-Code in eine Datei. Es wird geprüft, ob es sich bei der *action\_seq\_node* um eine Liste handelt, was einem bestimmten Knotentyp im Baum entspricht, und wenn dies zutrifft, werden einige Informationen aus dem ersten Element dieser Liste extrahiert. Diese Informationen werden verwendet, um die Werte einiger interner Variablen wie *\_\_addr\_direction*, *\_\_fast\_row* und *\_\_increment\_seq* zu setzen. Es wird lediglich das erste Element angeschaut, da in dem AST redundante Informationen gespeichert sind. Anschließend verwendet der Code diese internen Variablen, um die entsprechenden Werte für *\_\_reset\_val* und *\_\_increment\_seq* zu ermitteln. Je nach dem Wert von *\_\_addr\_direction* (der entweder +1 oder -1 ist), wird *\_\_reset\_val* entweder auf *g\_start\_of\_addr* oder *g\_end\_of\_addr* gesetzt. Wenn *\_\_fast\_row* *True* ist, wird *\_\_increment\_seq* auf *fastx\_inc* oder *fastx\_dec* gesetzt, ansonsten auf *fasty\_inc* oder *fasty\_dec*. Zum Schluss wird die Resetsequenz *reset\_cnt* mit dem Trennungsoperanden *##0*<sup>34</sup> geschrieben. Es folgen die SVA-Sequenzen:

Code 17: Schreiben der Sequenzen.

```

1     for sequence in node.getSequenceList():
2
3         #row wise
4         self.sv_out.write("(")
5         for rw_op in sequence[0]:
6             #addr increment
7             self.sv_out.write("(")
8             self.sv_out.write(__increment_seq+" "+"1"+"")
9             #sequence
10            self.sv_out.write(" and ")
11            self.sv_out.write(rw_op)
12            self.sv_out.write(")")
13            if rw_op == sequence[0][-1]:
14                if node.inc_operator.fast_row:
15                    self.sv_out.write(")*"+str(sequence[1])+")")
16                    self.__words_written += len(sequence[0])*sequence[1]
17                else:
18                    self.sv_out.write(")*"+str(sequence[1])+")")
19                    self.__words_written += len(sequence[0])*sequence[1]
20            else:
21                self.sv_out.write("##1")

```

<sup>34</sup>Verhält sich wie *and*. Die Signale werden im gleichen Taktzyklus ausgewertet.

```

22     if sequence == node.getSequenceList()[-1]:
23         #TODO: set repetition to fill remaining space
24         self.sv_out.write(")*"
25         +str(int(self.__word_cnt/self.__words_written))
26         +"])")
27     else:
28         self.sv_out.write("##1")
29     if action_seq_node == tree[-1]: #separate action_seqs or
        ↪ endproperty
30         self.sv_out.write(";\nendproperty")
31     else:
32         self.sv_out.write("\n ##1 ")

```

Eine verschachtelte Schleife durchläuft jede Sequenz in der Sequenzliste *sequenceList* einer Aktionssequenz, die der Rückgabe der Sequenzerkennung aus Abschnitt 6.5.4 entspricht. Für jede Sequenz wird eine SVA-Sequenz für das Adressinkrement (Zeile 10), gefolgt von dem Schreib- oder Leseoperanden geschrieben (Zeile 13). Wenn die aktuelle Sequenz die Letzte in der Sequenzliste ist, wird der SVA-Wiederholungsoperand geschrieben (Zeile 17/20). Schließlich prüft der Code, ob der aktuelle Knoten der Letzte im Baum ist, und schreibt *endproperty*, um das Property abzuschließen (Zeile 32). Andernfalls wird der Trennungsoperand *##1* geschrieben (Zeile 34), um die aktuelle *action\_seq\_node* vom nächsten Knoten im Baum zu trennen, dessen Sequenzen im nächsten Taktzyklus des Properties beginnen.

### 6.6.2 Bind-Writer

Code 18: Template für ein Bind-Modul.

```

1  def write_template(self):
2      sv_out = open("bind_module.sv", "w")
3      sv_out.write("""  bind %top_module% mbist_algorithms
4      #(
5          .g_data_width("""+str(self.json_dict["g_data_width"])+"""),
6          .g_row_addr_width("""+str(self.json_dict["g_row_addr_width"])+"""),
7          .g_col_addr_width("""+str(self.json_dict["g_col_addr_width"])+"""),
8          .g_num_of_addr("""+str(self.json_dict["g_num_of_addr"])+"""),
9          .g_num_of_rows("""+str(self.json_dict["g_num_of_rows"])+"""),
10         .g_mux_type("""+str(self.json_dict["g_mux_type"])+""")
11     )
12     inst_mbist_algorithms(
13         .clk_i(),
14         .reset_n_i(),
15         .data_o(),
16         .data_i(),
17         .write_enable(),
18         .col_addr_o(),
19         .row_addr_o()
20     );
21     endmodule""")

```

Der gezeigte Code definiert eine Methode namens *write\_template*, die ein Template eines SystemVerilog-Moduls in eine Datei mit dem Namen *bind\_module.sv* schreibt. Das SystemVerilog-Modul ist ein *bind*-Modul, das mit dem Top-Level-Modul (*%top\_module%*) verbunden ist. In diesem *bind*-Modul wird ein Modul mit dem Namen *inst\_mbist\_algorithms* instanziiert, das bestimmte Parameter erhält, die aus dem JSON-Dictionary (*self.json\_dict*) gelesen werden, in das in Abschnitt 6.4.5 die Memory-Parameter als Ergebnisse des Parsings geschrieben wurden. Die Parameter des Property-Moduls *inst\_mbist\_algorithms* werden hier zugewiesen. Das Modul erhält dabei folgende Parameter:

- *g\_data\_width*: Die Breite der Datenbusse.
- *g\_row\_addr\_width*: Die Breite der Zeilenadressen.
- *g\_col\_addr\_width*: Die Breite der Spaltenadressen.
- *g\_num\_of\_addr*: Die Anzahl der Adressen.
- *g\_num\_of\_rows*: Die Anzahl der Zeilen.
- *g\_mux\_type*: Der Typ des Multiplexers.

Das *inst\_mbist\_algorithms*-Modul hat mehrere Eingänge und Ausgänge, die im Code aufgelistet sind:

- *clk\_i*: Der Takt.
- *reset\_n\_i*: Das Reset-Signal.
- *data\_o*: Die Daten, die vom Modul ausgegeben werden.
- *data\_i*: Die Daten, die in das Modul eingegeben werden.
- *write\_enable*: Das Signal, das das Schreiben von Daten ermöglicht.
- *col\_addr\_o*: Die Spaltenadresse.
- *row\_addr\_o*: Die Zeilenadresse.

Im Bind-Modul sollen außerdem die Signale des Property-Moduls mit den Testbench-Signalen verbunden werden. Da die Signalhierarchie von der Software nicht ermittelt werden konnte, muss dies später manuell erfolgen (Zeile 13-19).

## 6.7 Auslieferung

Die Software wird in dem Versionsmanagement-System *Git*, das auf einem Bitbucket-Server intern betrieben wird, verwaltet. Neben der Möglichkeit sogenannte Branches, also instabile Abzweigungen des aktuellen Source-Codes, für neue Features zu erstellen, bietet es sich auch an, stabile Release-Branches zu erstellen, die nach einem Release nicht mehr verändert werden, um die Software auszuliefern.

Da die Software in der interpretierten Programmiersprache Python geschrieben wurde,

ist ein Compiling standardmäßig nicht möglich und auch nicht erwünscht. Es existiert die Möglichkeit, ein Bundle aus Software und Python-Interpreter auszuliefern. Dafür eignet sich das Modul *PyInstaller* [13], das eine ausführbare Datei aus einem Python-Programm erzeugt. Dies schränkt jedoch die Modularität ein, weil dann der Source-Code nicht mehr verändert werden kann. In einem Versionsmanagement-System ist es jedoch sinnvoll, Branches oder Forks<sup>35</sup> von lauffähigen Releases zu erstellen, die auch den Source-Code beinhalten, damit ein anderer Entwickler, der die Arbeit fortführen möchte, eine lauffähige Version erhält. Um die Software zu nutzen, muss eine Python-Instanz auf dem Hostrechner laufen, die kompatibel mit den benötigten Paketen ist. Die Software wurde mit Python Version 3.11 entwickelt, ist jedoch abwärtskompatibel zu älteren Versionen<sup>36</sup>.

Im Source-Code existieren Importe von öffentlichen Bibliotheken, darunter das Lexer-Parser Paket *ply* und das Mathematik Paket *numpy*. Diese Bibliotheken werden wie der Interpreter nicht mit ausgeliefert, sondern müssen vom Anwender installiert werden. Die Installation erfolgt beispielsweise mit dem Python Paketmanagement-System *pip*. Dieses bietet an, alle Pakete, die in einer lokalen Entwicklungsumgebung verwendet wurden, in einer Textdatei<sup>37</sup> aufzulisten. Mit *pip* kann diese Datei dann wieder eingelesen werden, um die aufgeführten Bibliotheken zu installieren.

In einer kommerziellen Umgebung haben Entwickler selten die Berechtigung, Pakete in der vorinstallierten Python-Umgebung zu installieren. Abhilfe schafft hier die Einrichtung einer virtuellen Python-Umgebung. Das nötige Modul *venv* ist vorinstalliert. Nach der Einrichtung können Pakete wie gewohnt mit *pip* installiert werden.

---

<sup>35</sup>Neues Repository, basierend auf einer bestimmten Version des alten Repository.

<sup>36</sup>Getestet mit Python 3.6.8.

<sup>37</sup>Meistens als *requirements.txt* bezeichnet.

## 7 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Entwicklung vorgestellt und kritisch analysiert, um die Funktionen und Möglichkeiten des Systems zu untersuchen. Die Analyse basiert auf den funktionalen Anforderungen und den Benutzererfahrungen, um zu zeigen, wie zuverlässig und effektiv BSMTA und Properties für die MBIST-Verifikation sind und wie das System arbeitet. Darüber hinaus werden Einschränkungen und Möglichkeiten für die zukünftige Weiterentwicklung des Systems diskutiert.

### 7.1 Eignung der Beschreibungssprache für Memory-Testalgorithmen

Die Verwendung einer abstrakten Beschreibungssprache für die Memory-Testalgorithmen hat einige Vorteile. Die verwendeten Algorithmen können in der Modulspezifikation des MBIST einheitlich dokumentiert werden. Die neu in die Beschreibungssprache eingeführten Symbole erlauben nun auch die Beschreibung von Testverfahren mit komplexeren *Data-Backgrounds*. Die Spracherweiterung ermöglicht nun auch die Beschreibung von Testverfahren, bei denen Scrambling berücksichtigt werden muss. Dies schafft Klarheit für den Verifikateur, der beobachten kann, ob ein logischer Speicherinhalt in der Simulation nach einer Schreibphase korrekt ist, da er aufgrund des dokumentierten Testverfahrens weiß, wie der physikalische Speicherinhalt aussehen soll. Darüber hinaus ist die Parserentwicklung, wie mit *Yacc* in Python demonstriert, aufgrund der kontextunabhängigen Grammatik der Beschreibungssprache unkompliziert, da die Syntax aufgrund der einfachen Struktur und der wenigen Symbole der Sprache schnell in BNF notiert ist. Grammatikerweiterungen können schnell hinzugefügt werden, was sich gezeigt hat, als das Eingabealphabet nicht groß genug war, um beispielsweise den Checkerboard-Algorithmus zu beschreiben.

### 7.2 Eignung von SVA-Properties

Die Verwendung von Properties zur Verifikation der Testalgorithmen hat sich ebenfalls bewährt. Der blockbasierte Entwicklungsansatz ermöglicht die Entwicklung von übersichtlichen und mit der Beschreibung konsistenten Properties. Mit den entwickelten Properties konnten in der Simulation Fehler in den Testalgorithmen gefunden werden. Dadurch wird sichergestellt, dass das Memory-Interface wie erwartet funktioniert und kein unerwünschtes Verhalten auftritt. Darüber hinaus sind Debugging und Fehlerbehebung während des Verifikationsprozesses möglich. Wenn das Verhalten der Speicherschnittstelle nicht den erwarteten Regeln entspricht, können die entsprechenden Properties identifiziert und analysiert werden, um die Ursache des Problems schnell zu finden.

### 7.3 Software

Der AST hat sich als sinnvolle Datenstruktur für die Sprachelemente erwiesen, da Baumknoten während des Parsens leicht erzeugt und verknüpft werden können. Außerdem kann der AST aufgrund der Knotenbeziehungen traversiert werden, weshalb sich diese Datenstruktur für die Weiterverarbeitung der BSMTA in einem Software-System eignet. Durch Manipulation des AST können Informationen ergänzt werden, bis am Ende eine

Datenstruktur vorliegt, in der alle notwendigen Daten vorhanden sind, um daraus weitere Strukturen zu erzeugen. Dies wurde genutzt, um SVA-Properties zu generieren und kann weiter ergänzt werden. So können weitere Hardware-Verifikations-Sprachen, mit denen ein Interface-Verhalten verifiziert werden kann, hinzugefügt werden. So ist es ohne große Modifikationen möglich, beispielsweise Universal-Verification-Methodology (UVM)-Komponenten zu generieren, indem lediglich weitere Funktionen im Source-Code ergänzt werden, die statt oder mit dem SVA-Writer aufgerufen werden und durch den gefüllten AST traversieren. UVM ist ein Framework für den Test und die Verifikation von Designs in der Chipentwicklung. Es kombiniert automatische Testgenerierung, selbstüberprüfende Testbenches und Coveragemetriken, um die für die Verifikation eines Designs benötigte Zeit zu reduzieren. Eine UVM-Testbench besteht aus wiederverwendbaren Verifikationskomponenten, die in Verilog, VHDL, System Verilog und System C strukturiert sind und zur Verifikation von Protokollen wie dem AMBA High-Performance Bus (AHB) verwendet werden können [10]. Da eine Sequenz auf dem Memory-Interface des MBIST in Verbindung mit einem BSMETA-Term einem Protokoll entspricht, kann diese auch beispielsweise mit SystemVerilog in einem UVM-Kontext verifiziert werden.

### 7.3.1 Unterstützte Testalgorithmen

Mit der Software wurden einige BSMETA-Terme für ein DUT verarbeitet und anschließend simuliert, um die Software zu testen. Zusätzlich sind in Tabelle 8 auf Seite 73 einige Memory-Testalgorithmen aufgelistet, die aus [11, S. 28] übernommen worden sind und/oder im hier behandelten DUT verwendet werden. In der Tabelle wird aufgeführt, ob sie unterstützt werden oder, falls nicht, welche Einschränkung der Software die Ursache dafür ist. Die Algorithmen werden in Kategorien eingeteilt und der Inhalt bezüglich der Unterstützung wird in Abbildung 28 visualisiert.

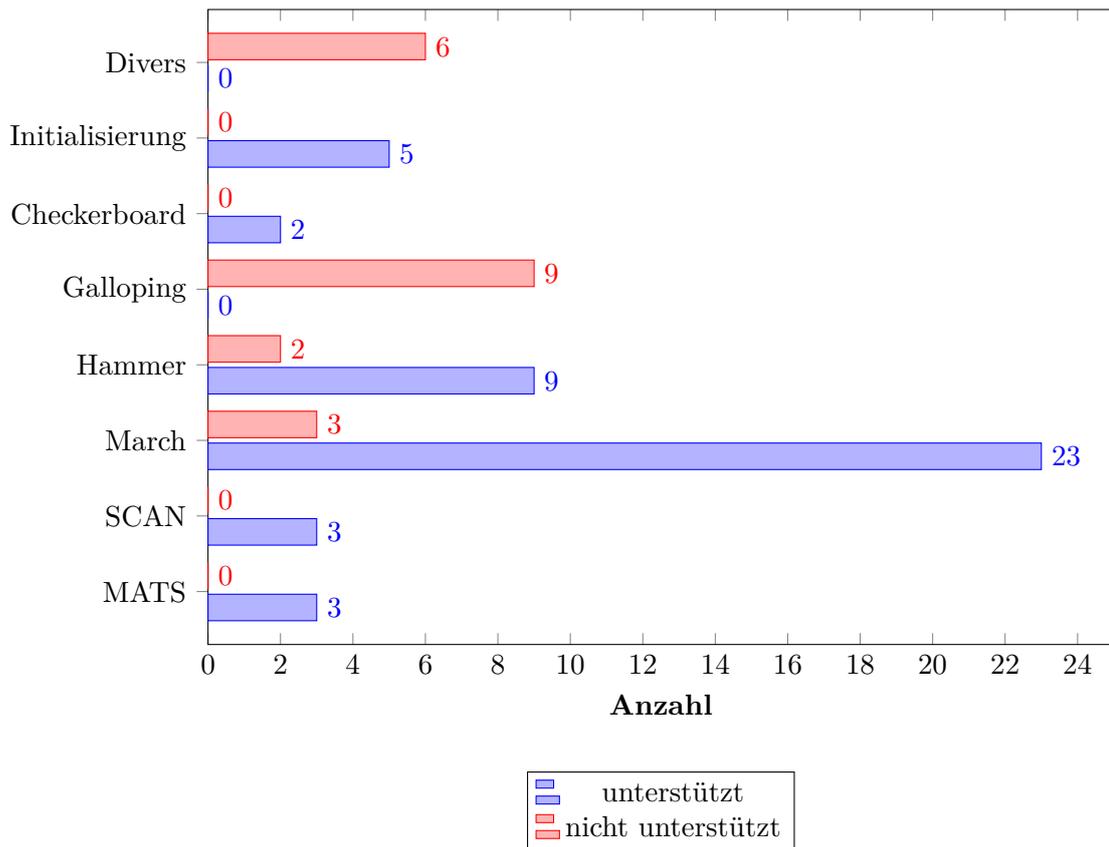


Abbildung 28: Von der Software unterstützte Testalgorithmen nach Kategorie.

Das Software-System unterstützt das Parsen der BSMTA, die Weiterverarbeitung und die korrekte Ausgabe von SVA-Properties für fast alle genannten Algorithmenkategorien und die Mehrzahl der individuellen Testalgorithmen. Lediglich die Testalgorithmen der Kategorie Galloping werden gar nicht unterstützt. Von den im DUT vorhandenen Algorithmen werden ca. 60% unterstützt. Bei den dort nicht unterstützten Algorithmen handelt es sich um Verfahren, die DUT-spezifisch sind und die hier unter "Divers" klassifiziert sind. Besonders auffällig ist, dass die meisten March-Algorithmen (ca. 90%), die wie SCAN und MATS nur aus der einfachen Teilmenge der Operanden der BSMTA (1) bestehen, unterstützt werden. Insgesamt werden 69% der untersuchten Algorithmen in Tabelle 8 auf Seite 73 unterstützt.

### 7.3.2 Einschränkungen

Einige Testalgorithmen benötigen Operanden, die nicht in der Teilmenge (1) sind und die aufgrund ihrer Komplexität oder weil sie im DUT nicht verwendet wurden, nicht implementiert wurden. Die Einschränkungen der Software werden ausgewertet und es werden, wenn möglich, Lösungen vorgeschlagen.

- **Fast-Row:** Die Überprüfung des Adresspfads für Fast-Row (vgl. Abbildung 2b auf Seite 14) ist auf einfache Weise möglich. Dazu wurde bereits in Tabelle 4 auf Seite 27 eine SVA-Sequenz vorgestellt, in der ein Fast-Row-Adresszähler implementiert ist, mit dem Adressen auf dem Memory-Interface verifiziert werden können. Problematisch ist jedoch das Scrambling, das die zu schreibenden Datenvektoren verändert. Das BSMTA-Bitfeld muss vertikal durchlaufen werden, wofür zweidimensionale Bit-

felder besser geeignet sind als die hier verwendeten eindimensionalen Bitfelder. Zusätzlich ist eine zweidimensionale LUT notwendig. Es ist also eine große Änderung der Softwarearchitektur notwendig, die aber durchaus sinnvoll ist, weil dadurch auch andere Adressierungstechniken, wie beispielsweise N-E-S-W, die hier nicht diskutiert wurden, unterstützt werden können.

- **Base-Cell-Adressierung:** Die Testalgorithmen der Kategorie *Galopping* sowie *Hammer* nutzen oft eine Base-Cell Adressierung [11, S. 28], die nicht unterstützt wird.
- **N-E-S-W-Adressierung:** Ähnlich wie bei der Fast-Row-Adressierung wird für die Adressierung benachbarter Zellen ein zweidimensionales Bitfeld benötigt, um die nötigen Adressen zu ermitteln. Zusätzlich sind im behandelten DUT nur Wortzugriffe möglich, weshalb diese bitorientierte Adressierungstechnik dort nicht sinnvoll ist.
- **Delays:** Verzögerungen in den Testalgorithmen sind äußerst selten und werden nur bei March SRD+ und March G verwendet, da diese der möglichst kurzen Laufzeit der Testalgorithmen entgegen stehen. Verzögerungen müssen in der Grammatik berücksichtigt werden und können in den SVA-Properties durch weitere Halteglieder mit `$steady()` implementiert werden.

## 8 Diskussion

In diesem Kapitel wird diskutiert, welche Testcoverage mit den unterstützten Testalgorithmen zu erwarten ist und wie diese mithilfe der formalen Verifikation erweitert werden kann. Zusätzlich werden mögliche Probleme in der Testbench und der Vorgehensweise anhand einer Fallstudie diskutiert.

### 8.1 Coverage der unterstützten Algorithmen

Da einige Algorithmen aufgrund der in Abschnitt 7.3.2 genannten Einschränkungen des Software-Systems nicht verarbeitet werden können, stellt sich die Frage, wie nützlich die Software ist und ob die Algorithmen, die laut Abbildung 28 auf Seite 63 zahlreich unterstützt werden, auch eine ausreichende Coverage haben, um die Existenz der Software inklusive ihrer Einschränkungen in ihrem jetzigen Zustand zu legitimieren. Der Fokus dieser Arbeit liegt nicht auf der Zusammensetzung der Testalgorithmen und der daraus resultierenden Coverage, aber andere Publikationen können dabei helfen, diese zu bewerten.

In [8] werden verschiedene Fehlermodelle und Testalgorithmen zur Erkennung von Fehlern in SRAMs und DRAMs behandelt. Fehlermodelle werden verwendet, um verschiedene Arten von Fehlern, die in den Speicherschaltungen auftreten können, zu identifizieren und zu klassifizieren, während Testalgorithmen verwendet werden, um diese Fehler zu erkennen. Diese Fehlermodelle können zusammen mit den Testalgorithmen simuliert werden, um die Coverage der Testalgorithmen zu ermitteln. Es wird ein March-Algorithmus namens *March LSD* beschrieben, der alle 2-Komposit-Fehler im Memory erkennen kann. 2-Komposit-Fehler beziehen sich auf eine bestimmte Art von Fehlern, die in Speicherzellen oder Schaltungen auftreten können. Diese Fehler setzen sich aus zwei separaten Fehlern zusammen, die gleichzeitig auftreten und entweder statisch oder dynamisch sind. Der Algorithmus ist sehr effektiv und kann alle modellierten Fehler, darunter auch sehr komplexe, erkennen. Der behandelte *March LSD* Algorithmus hat den folgenden BSMTA-Term [8]:

$$\begin{aligned}
 & \{\downarrow (w0); \uparrow (r0, w1, r1, w1, w1, r1, w1, w0, r0, w1, w1, r1, w0, w1, r1, w1, r1, r1); \\
 & \quad \uparrow (r1, w1, w1, r1, w1, w0, r0, w1, w1, r1, w0, w1, r1, w1, r1, r1, w0); \uparrow (r0); \\
 & \quad \quad \downarrow (r0, w0, w0, r0, w0, w1, r1, w0, w0, r0, w1, w0, r0, w0, r0, r0, w1); \quad (20) \\
 & \quad \quad \downarrow (r1, w0, r0, w0, w0, r0, w0, w1, r1, w0, w0, r0, w1, w0, r0, w0, r0, r0); \\
 & \quad \quad \quad \downarrow (r0); \}
 \end{aligned}$$

Der Term besteht aus den Standardoperanden der BSMTA, die in der in Abschnitt 6.4.1 definierten Grammatik bereits ohne Erweiterung enthalten sind. Daher ist das Software-System in der Lage, gültige SVA-Properties für den *March LSD* Algorithmus zu generieren. Daraus lässt sich schließen, dass das Software-System Properties für Testalgorithmen generieren kann, die eine hinreichend große Coverage haben und auch komplexe Fehler mit mehreren Fehlerkomponenten erkennen können.

Andererseits genügt die Software nicht dem Anspruch, ein vollständiges Set an Testcases für einen MBIST zu generieren. Für die häufigsten Algorithmen<sup>38</sup> können allerdings Properties generiert werden und der Rest kann in das generierte Modul eingepflegt werden. Zusätzlich ist es sehr hilfreich, Properties für besonders lange<sup>39</sup> und unübersichtliche Testalgorithmen, wie den *March LSD* automatisch zu generieren, um die dort wahrscheinlicheren Fehler des Verifikateurs zu vermeiden.

## 8.2 Coverage von Wortzugriffen

In dieser Arbeit wurden Vereinfachungen getroffen, die diskutiert werden müssen, da sie Auswirkungen auf die Qualität der Verifikation haben können und eventuell revidiert werden sollten. Bei der Analyse der Sprachkonstrukte der abstrakten Beschreibungssprache wurde die Vereinfachung getroffen, dass Schreib- und Leseoperanden der Sprache einen Zugriff auf alle Bits eines Wortes beschreiben. Während einer Schreibphase würden also alle Zellen eines Wortes gleichzeitig beschrieben. Dies könnte bei Algorithmen, die Adressdekodierfehler oder Kopplungsfehler mit benachbarten Zellen finden sollen, zu einem Coverageverlust führen. Der Checkerboard-Test ist jedoch ein Worst-Case-Test für den Leckstrom zwischen Zellen und ist davon ausgenommen. Hier führt die Vereinfachung mit einem Wortzugriff nach [5, S. 99] zu keinem Coverageverlust. Tests mit Adressierungsarten, die hier nicht behandelt werden, wie z. B. die N-E-S-W-Adressierung, die für Kopplungsfehler ausgelegt sind, sollten gesondert behandelt werden. Das als Forschungsgrundlage für diese Arbeit verwendete DUT unterstützt diese Algorithmen jedoch nicht, weshalb weitere Untersuchungen nicht möglich waren. Eine Ausnahme bilden die Hammer-Tests, von denen einige mit den hier behandelten Operanden beschrieben werden können. Das DUT führt bei allen Algorithmen nur ganze Wortzugriffe aus. Für diesen Fall ist daher die genannte Vereinfachung ausreichend.

## 8.3 Formale Verifikation

Bei der formalen Verifikation, beziehungsweise dem formalen Property Checking<sup>40</sup>, handelt es sich um eine Alternative zur konventionellen, simulationsbasierten Verifikation. Die formale Verifikation basiert auf dem Prinzip des Model Checkings. Hier wird ein Design vollautomatisiert in eine Systembeschreibung in einer formalen Sprache überführt. Die diskreten Zustände eines digitalen Systems können beispielsweise mit einer Zustandsmaschine mit einer endlichen Anzahl an Zuständen (FSM) beschrieben werden [2]. Dies ist eine gängige Methode für eine Systembeschreibung, die von formalen Tools genutzt wird. Während in der Simulation der Stimulus am Eingang eines Moduls vorgegeben wird und das Ausgangsverhalten beobachtet wird, wird bei der formalen Verifikation im Wesentlichen ein bestimmtes Ausgangsverhalten festgelegt, das eine Menge an Zuständen in der Zustandsmaschine besitzt. Durch einen (constrainten) Eingangsstimulus dürfen dann

<sup>38</sup>Häufigkeit bewertet nach Vorkommen in Tabelle 8 auf Seite 73.

<sup>39</sup>Testalgorithmen mit einer linearen Schrankenfunktion  $O(n)$  zur Komplexitätsmessung sind in ihrer Laufzeit abhängig von der Anzahl der Iterationen durch das Memory, die von der Operandenanzahl im BSMTA-Term abhängt [21]. Der March LSD hat eine Komplexität von  $75N$  [8] bei 76 Operanden.

<sup>40</sup>Nicht zu verwechseln mit dem Formal Equivalence Checking, bei dem zwei Designs durch formale Methoden auf Äquivalenz geprüft werden.

nur Endzustände aus dieser bestimmten Menge durch Zustandsübergänge erreicht werden. Deshalb wird die formale Verifikation oft als *output-driven* bezeichnet. Anhand der Charakteristika der formalen Verifikation zeigen sich einige Vorteile gegenüber der Simulation. Da jeder mögliche Zustand des Designs in dieser Systembeschreibung abgebildet und bei Berechnungen berücksichtigt wird, ist die formale Verifikation komplett<sup>41</sup>, was der wesentliche Vorteil gegenüber der Simulation ist.

Bei der formalen, Property-basierten Verifikation traten einige Probleme auf, weshalb mit der Simulation fortgefahren wurde. Hier stellt sich die Frage, ob die formale Verifikation für die Verifikation eines MBIST geeignet ist und ob dies zu einer erhöhten Testcoverage führt. Die Simulation hat in Abschnitt 4.2 demonstriert, dass Fehler in den Testalgorithmen durch einen Referenzabgleich in einem Property-basierten Verifikationsverfahren gefunden werden können. Die Voraussetzung dafür ist ein Stimulus, der die interne Zustandsmaschine im MBIST definitiv in einen Testmodus überführt. Dieser Stimulus ist in der Testbench definiert. Die Simulation hat demnach ausreichend Coverage für die Verifikation der implementierten Testalgorithmen. Nach der Konfigurationssequenz ist der Treiber inaktiv und hält die Signale auf dem Konfigurationsinterface stabil. Die formale Proof-Engine würde stattdessen weiterhin Signaländerungen treiben, wenn diese zu einem Counter-Example für ein *assert*-Statement führen können. Somit könnte untersucht werden, unter welchen Umständen der MBIST ein laufendes Testverfahren unerwartet abbricht, was in der Simulation mit einfachem Stimulus nicht möglich ist. Eine Möglichkeit wäre hier die Anlegung zufälliger Stimuli nach der Konfigurationssequenz. Ein weiterer Nachteil ist, dass die Konfigurationssequenz bei der Simulation fest definiert ist. In der formalen Verifikation können andere Sequenzen über *cover*-Statements gefunden werden, die (ungewollt) einen MBIST Testmodus starten und somit möglicherweise nicht erlaubt sind.

#### 8.4 Fallstudie: Verifikationsschleife

Der MBIST wird von einem RTL-Generator erzeugt, der als Eingabe die Testsequenzen in einer nicht näher erläuterten proprietären Beschreibungssprache erfordert. Hier soll die Frage diskutiert werden, warum nicht diese Beschreibungssprache als Eingabe für den hier entworfenen Testbench Generator verwendet wird. Mehrere Gründe sprechen dagegen:

1. Die proprietäre Sprache ist nicht so gut lesbar, wie die hier verwendete BSMTA. Dadurch eignet sie sich nicht für die Spezifikation.
2. MBIST und Testbench würden aus derselben Quelle generiert werden.

Abbildung 29 soll das Problem, das durch Punkt 2 verursacht wird, verdeutlichen.

---

<sup>41</sup> ausschöpfend; iteriert durch alle möglichen Zustände der endlichen Zustandsmaschine (FSM), die das digitale System komplett beschreibt.

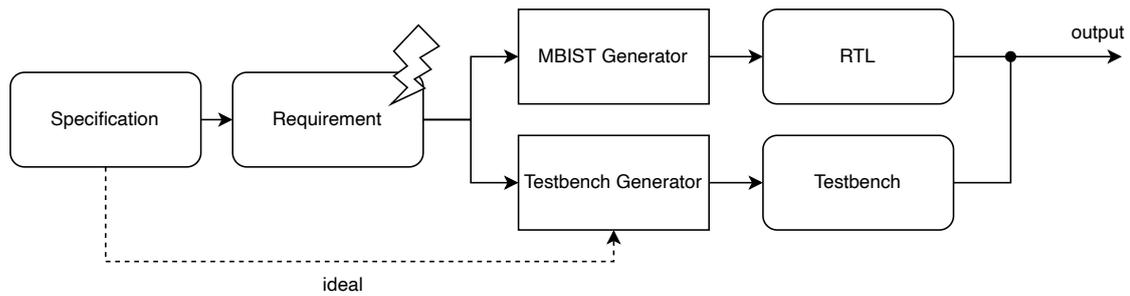


Abbildung 29: Verifikationsschleife.

Diese Praxis widerspricht dem Grundsatz, dass zur Erreichung von Modularität und Wartbarkeit die verschiedenen Teile eines Systems unabhängig voneinander entworfen und implementiert werden sollten. Wenn Code und Testbench aus derselben Quelle generiert werden, besteht die Gefahr, dass die Testbench auf die Implementierungsdetails des generierten Codes ausgerichtet ist und nicht alle möglichen Szenarien und Corner-Cases abdeckt. Dies kann zu unzureichenden Tests führen. Die Folge können unentdeckte Fehler sein. Bei der Generierung des MBIST kann ein Testalgorithmus falsch spezifiziert sein. Dieser Fehler kann unbemerkt bleiben, falls die Generatorkonfigurationen für MBIST und Testbench identisch sind. Zur Vermeidung dieses Problems empfiehlt sich die Generierung der Testbench aus einer separaten Quelle oder die manuelle Erstellung der Testbench.

In dieser Arbeit wurden die Memory-Parameter für den MBIST-Generator und den Assertion-Generator aus derselben Quelle eingelesen. Hier sollte geprüft werden, dass die Parameter korrekt sind. Da die Memory-Makros gesondert verifiziert werden, wurde davon ausgegangen, dass diese Voraussetzung erfüllt wird. Die Memory-Testalgorithmen wurden jedoch aus zwei unterschiedlichen Quellen gelesen, da der MBIST-Generator die BSMTA nicht unterstützt.

## 9 Ausblick

Die vorgestellte Software unterstützt viele Testalgorithmen, die in der Praxis verwendet werden. Allerdings gibt es auch Einschränkungen, insbesondere bei der unterstützten Teilmenge der Operanden der BSMTA und Adressierungstechniken. Um diese Einschränkungen zu überwinden, sind Änderungen an der Softwarearchitektur notwendig. Eine mögliche Erweiterung wäre die Implementierung von zweidimensionalen Bitfeldern, um die Unterstützung von Adressierungstechniken wie N-E-S-W oder Base-Cell zu ermöglichen. Dies würde auch die Unterstützung anderer Adressierungstechniken erleichtern und die Flexibilität der Software erhöhen. Die Weiterentwicklung der Software und der Testcases für die MBIST-Verifikation sollte eine enge Zusammenarbeit mit den Entwicklern des DUT einschließen, welche auch die Memory-Testalgorithmen in der BSMTA spezifizieren sollten, um die Übergabe zwischen Design und Verifikation zu erleichtern. Dadurch wird auch das in Abschnitt 8.4 vorgestellte Problem der Verifikationsschleife vermieden, da die Testcases nicht aus der Konfiguration des MBIST-Generators, sondern aus der ursprünglichen Spezifikation generiert werden. Eine weitere Funktion, die den Verifikationsflow verbessert, ist die Unterstützung von Script-Sprachen wie Tcl, die bereits von anderen Electronic-Design-Automation (EDA)-Tools verwendet werden. Dies ermöglicht eine Vollautomatisierung des Design- und Verifikationsprozesses, da bei RTL-Änderungen automatisch eine Neugenerierung der Properties angestoßen werden kann. Diese können dann mithilfe eines Simulators, der ebenfalls mit Tcl gesteuert wird, zur Verifikation der Änderungen verwendet werden. Neben der Simulation sollte der Fokus außerdem auf der formalen, Property-basierten Verifikation liegen, da hier eine höhere Coverage zu erwarten ist.

## 10 Fazit

Im Fokus dieser Arbeit stand die Entwicklung eines Software-Systems, das aus einer Memory-Testalgorithmenbeschreibung in einer abstrakten Beschreibungssprache Testcases in Form von SVA-Properties generiert, mit denen beispielsweise in einem Simulator das Verhalten des Memory-Interfaces eines MBIST verifiziert werden kann. Obwohl die Properties modular entwickelt werden können und bei einigen Testalgorithmen wie March, SCAN und MATS eine direkte Überführung möglich ist, stellte sich heraus, dass für einige Testalgorithmen zunächst mithilfe von Transformationen Bitfelder generiert werden müssen, um Scrambling berücksichtigen zu können. Die Grammatik der BSMTA wurde zunächst definiert und dann mit den nötigen Operanden erweitert, um nun auch die Beschreibung von Testalgorithmen mit komplexeren Data-Backgrounds wie Scrambling zu erlauben. Insgesamt hat sich gezeigt, dass eine Automatisierung der Property-Entwicklung unter Berücksichtigung dieser Rahmenbedingungen durchaus möglich ist. Dies wird durch die umfangreiche Unterstützung der Testalgorithmen durch die Software deutlich, von denen einige bereits automatisch in SVA-Properties überführt werden konnten. Wichtig ist hierbei auch die Akzeptanz der Hardware-Entwickler, welche die verwendeten Algorithmen in einer von Menschen und Maschinen lesbaren Beschreibungssprache wie der BSMTA dokumentieren sollten. Die vorliegende Thesis hat somit demonstriert, dass die Entwicklung von automatisierten Verifikationswerkzeugen und einheitlichen Beschreibungssprachen für Memory-Testalgorithmen ein wichtiger Schritt in Richtung effektiver und effizienter Chip-Entwicklung ist.

## Literatur

- [1] D. Beazley, *PLY (Python Lex-Yacc); ply 4.0 documentation — ply.readthedocs.io*, [Zugriff am 14.03.2023], 2003-2020. Adresse: <https://ply.readthedocs.io/en/latest/ply.html#ply-overview>.
- [2] B. Bérard, M. Bidoit, A. Finkel u. a., *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.
- [3] P. S. Foundation, *The Python Tutorial*, <https://docs.python.org/3/tutorial/index.html>, [Online; Stand 18. April 2023], 2023.
- [4] A. van de Goor, G. Gaydadjiev und S. Hamdioui, “Memory testing with a RISC microcontroller,” in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, 2010, S. 214–219. DOI: 10.1109/DATE.2010.5457210.
- [5] A. J. Van de Goor, *Testing semiconductor memories: theory and practice*. John Wiley & Sons, Inc., 1991.
- [6] J. Grus, *Einführung in data science*, ger, 1. Auflage. Heidelberg, [Germany: O’Reilly, 2016 - 2016, ISBN: 1-4920-6666-4.
- [7] C. R. Harris, K. J. Millman, S. J. van der Walt u. a., “Array programming with NumPy,” *Nature*, Jg. 585, Nr. 7825, S. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. Adresse: <https://doi.org/10.1038/s41586-020-2649-2>.
- [8] G. Harutyunyan, S. Shoukourian, V. Vardanian und Y. Zorian, “A New Method for March Test Algorithm Generation and Its Application for Fault Detection in RAMs,” eng, *IEEE transactions on computer-aided design of integrated circuits and systems*, Jg. 31, Nr. 6, S. 941–949, 2012, ISSN: 0278-0070.
- [9] Infineon, *Infineon introduces new CMOS transceiver MMIC CTRX8181 with high performance, scalability and reliability for automotive radar*, 2022. Adresse: <https://www.infineon.com/cms/en/about-infineon/press/market-news/2022/INFATV202211-017.html>.
- [10] P. N. Kalyan und K. J. Swaroop, “Verification of AMBA AHB based verifying IP using UVM methodology,” *Int J Electron Commun Instrum Eng Res Dev ISSN (P)*, 2015.
- [11] M. Linder, “Test Set Optimization for Industrial SRAM Testing,” Dissertation, Technische Universität München, München, 2013.
- [12] D. D. McCracken und E. D. Reilly, “Backus-Naur Form (BNF),” in *Encyclopedia of Computer Science*. GBR: John Wiley und Sons Ltd., 2003, S. 129–131, ISBN: 0470864125.
- [13] *PyInstaller Manual*, <https://pyinstaller.org/en/stable/>, [Online; Stand 18. April 2023], 2022.
- [14] C. Selva, C. Torelli, D. Rimondi u. a., “A programmable built-in self-diagnosis for embedded SRAM,” in *Records of the 2004 International Workshop on Memory Technology, Design and Testing, 2004.*, 2004, S. 84–89. DOI: 10.1109/MTDT.2004.1327989.

- [15] B. Singh, S. B. Narang und A. Khosla, “Modeling and Simulation of efficient march algorithm for memory testing,” in *International Conference on Contemporary Computing*, Springer, 2010, S. 96–107.
- [16] L. Stiny, *Aktive elektronische Bauelemente: Aufbau, Struktur, Wirkungsweise, Eigenschaften und praktischer Einsatz diskreter und integrierter Halbleiter-Bauteile*. Springer Fachmedien Wiesbaden, 2019, ISBN: 9783658247522.
- [17] systemverilog.io, *A Gentle Introduction to Formal Verification*, en. Adresse: <https://www.systemverilog.io/gentle-introduction-to-formal-verification> (besucht am 27.12.2022).
- [18] S. Takamaeda-Yamazaki, “Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL,” in *Applied Reconfigurable Computing*, Ser. Lecture Notes in Computer Science, Bd. 9040, Springer International Publishing, 2015, S. 451–460. DOI: 10.1007/978-3-319-16214-0\_42. Adresse: [http://dx.doi.org/10.1007/978-3-319-16214-0\\_42](http://dx.doi.org/10.1007/978-3-319-16214-0_42).
- [19] A. J. Van De Goor und I. Schanstra, “Address and data scrambling: Causes and impact on memory tests,” in *Proceedings First IEEE International Workshop on Electronic Design, Test and Applications’ 2002*, IEEE, 2002, S. 128–136.
- [20] Wikipedia, *Metasprache* — *Wikipedia, die freie Enzyklopädie*, [Online; Stand 19. Dezember 2022], 2020. Adresse: <https://de.wikipedia.org/w/index.php?title=Metasprache&oldid=202231184>.
- [21] Wikipedia, *Komplexitätstheorie* — *Wikipedia, die freie Enzyklopädie*, [Online; Stand 19. Dezember 2022], 2022. Adresse: <https://de.wikipedia.org/w/index.php?title=Komplexit%C3%A4tstheorie&oldid=228413597>.
- [22] Wikipedia, *Regulärer Ausdruck* — *Wikipedia, die freie Enzyklopädie*, [Online; Stand 20. Dezember 2022], 2022. Adresse: [https://de.wikipedia.org/w/index.php?title=Regul%C3%A4rer\\_Ausdruck&oldid=228760525](https://de.wikipedia.org/w/index.php?title=Regul%C3%A4rer_Ausdruck&oldid=228760525).
- [23] Wikipedia, *Datenkapselung (Programmierung)* — *Wikipedia, die freie Enzyklopädie*, [Online; Stand 14. März 2023], 2023. Adresse: [https://de.wikipedia.org/w/index.php?title=Datenkapselung\\_\(Programmierung\)&oldid=231600131](https://de.wikipedia.org/w/index.php?title=Datenkapselung_(Programmierung)&oldid=231600131).
- [24] Wikipedia, *JavaScript Object Notation* — *Wikipedia, die freie Enzyklopädie*, [Online; Stand 20. März 2023], 2023. Adresse: [https://de.wikipedia.org/w/index.php?title=JavaScript\\_Object\\_Notation&oldid=229374937](https://de.wikipedia.org/w/index.php?title=JavaScript_Object_Notation&oldid=229374937).
- [25] Wikipedia contributors, *Peres–Horodecki criterion* — *Wikipedia, The Free Encyclopedia*, [Online; Stand 27. März 2023], 2023. Adresse: [https://en.wikipedia.org/w/index.php?title=Peres%E2%80%93Horodecki\\_criterion&oldid=1140623457](https://en.wikipedia.org/w/index.php?title=Peres%E2%80%93Horodecki_criterion&oldid=1140623457).

## 11 Anhang

### 11.1 Unterstützte Algorithmen

Tabelle 8: Unterstützte Algorithmen aus [11, S. 29] und erweitert mit spezifischen DUT-Algorithmen. Die Unterstützung basiert auf einer für die Software gültigen BSMETA-Beschreibung. Mit \* markierte Algorithmen sind DUT-exklusiv und nicht in der Literaturquelle vorhanden.

Bezeichnung	Unterstützt	Getestet	Hinweis
SCAN	✓	✓	Wie Scan1 in DUT.
SCAN+	✓		
MATS	✓		
MATS+	✓		
MATS++	✓		
March C-	✓		
March C-	✓		
March A	✓		
March B	✓		
Algorithm B	✓		
March C+	✓	✓	Wie March2 in DUT.
PMOVI B	✓		
March 1/0	✓		
March TP	✓		
March U	✓		
March X	✓		
March Y	✓		
March LR	✓*	✓*	*Im DUT wurde der March-LR mit einer Checkerboard Sequenz mit fast-x Adressierung erweitert, die nicht unterstützt wird.
March LA	✓		
March RAW	✓		
March RAW1	✓		
March AB	✓		
March AB1	✓		
March BDN	✓		
March SR	✓		
March SR+	✓		
March SRD+			Nicht unterstütztes Delay <i>D</i> .
March SS	✓		
March SL	✓		
March G			Nicht unterstütztes Delay <i>D</i> .

GAL5R			Nicht unterstützte Base-Cell und N-E-S-W Adressierung.
GAL9R			Nicht unterstützte Base-Cell und N-NE-E-SE-S-SW-W-NW Adressierung.
GAL5W			Nicht unterstützte Base-Cell und N-E-S-W Adressierung.
GAL9W			Nicht unterstützte Base-Cell und N-NE-E-SE-S-SW-W-NW Adressierung.
Walking 1/0			Nicht unterstützte Base-Cell Adressierung.
Butterfly			Nicht unterstützte Base-Cell und N-E-S-W Adressierung und Wiederholungsoperand.
GALPAT			Nicht unterstützte Base-Cell Adressierung.
GALROW			Nicht unterstützte Base-Cell Adressierung.
GALCOL			Nicht unterstützte Base-Cell Adressierung.
BLIF			Nicht unterstützte fast-x Adressierung.
HamW16	✓		
HamR16	✓		
HamR28	✓		
Ham-Max	✓		
Hammer_L	✓		
Hammer			Nicht unterstützte Base-Cell und diagonale Adressierung.
HamW			Nicht unterstützte Base-Cell und diagonale Adressierung.
Ham5R	✓		
Ham5W	✓		
Ham_Walk	✓		

Random	√*	√*	*Alle anderen möglichen Kombinationen von Standard-Sequenzen.
*SMARCHCHKBCIL			
*LV_XDEC_SOPEN			
*LV_YDEC_SOPEN			
*CHECKERBOARD	√	√	
*HAMMERTTEST	√	√	
*MEMINIT	√	√	
*ECCESUPPORT			
*INVERSE-CHECKERBOARD	√	√	
*ERRIN			
*MEMINIT-SOLID0	√	√	
*MEMINIT-SOLID1	√	√	
*MEMINIT-ROWSTRIPES	√	√	
*MEMINIT-INVERSEROWSTRIPES	√	√	
*FDMA_R			
*FDMA_W			