4.9
Testen von VHDL-Modellen

4.9.1
Simulationstechniken

Der Schwerpunkt lag bisher auf VHDL-Modellen, die mit Hilfe programmierbarer Logik synthetisiert werden. In erster Linie wurde VHDL als Entwurfssprache eingesetzt. Im Folgenden soll die VHDL-Syntax zur Bildung von Testumgebungen verwendet werden. Hierzu werden nichtsynthesefähige VHDL-Modelle aufgestellt, mit deren Hilfe VHDL-Komponenten gestestet werden können. Der zu testende VHDL-Entwurf wird in eine Testbench eingebunden. Der Name Testbench (Werkbank) wird in Analogie zu einer realen Werkbank gewählt, auf der ein Werkstück getestet werden kann. In VHDL werden die Eingangssignale mit Signalgeneratoren stimuliert und die Reaktion der Ausgangssignale kontrolliert.

Die einfachste Form einer Simulation ist die interaktive. VHDL-Compiler enthalten Tools, um die Porteingänge eines Modells zu stimulieren und die Ergebnisse in Tabellenform oder als Signalzeitdiagramm am Bildschirm und/oder Drucker auszugeben. Diese Methode eignet sich gut, um einen Überblick über das Verhalten des Entwurfs zu erlangen. Im allgemeinen Fall muss die Eingabe bei einer Änderung wiederholt werden, was zeitaufwendig und fehlerträchtig ist.

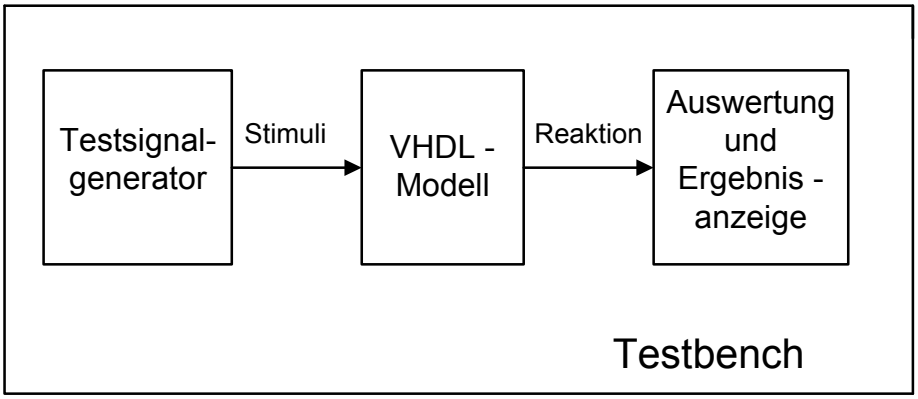

Bild 4.3: Die Testbench dient zur Erprobung des VHDL-Modells.

Eine Testbench hat gegenüber der interaktiven Methode deutliche Vorteile:

- Gute Dokumentation der Testvektoren für Ein- und Ausgänge
- Leichte Wiederverwendbarkeit bei Entwurfsänderungen
- Systematische Vorgehensweise
- Schnelle Fehlererkennung

- Die gleiche Testbench lässt sich sowohl für das Quellcode-VHDL-Modell als auch für das Postlayout-VHDL-Modell verwenden.

Im Folgenden werden zwei unterschiedliche Entwürfe für Testbenches vorgestellt:

- Testvektoren werden innnerhalb der Testbench erzeugt
- Testbench mit Ein- und Ausgabedatei

4.9.2 Testbench mit Testvektoren

Die Entity ist sehr einfach aufgebaut, insbesondere enthält sie keine Port- oder Generic-Anweisungen:

```
entity testbench_name is       -- Entity der Testumgebung
end testbench_name;
```

Die Architektur enthält ein Testvektorfeld, das sowohl die stimulierenden Werte für die Eingabeports als auch die am Ausgang erwarteten Logikzustände enthält. Das zu testende VHDL-Modell muss als Komponente vorliegen. Die compilierte Komponente ist in einem Package in der Work-Library oder in einer benutzerdefinierten Library gespeichert. Bei der Ausführung werden in einer For-Schleife die Stimuli eines Vektors (hier: vektor.x) den Porteingängen (hier: x) zugewiesen. Nach einer Verzögerung (hier: 30 ns), die die Durchlaufzeit des Schaltkreises simuliert, wird die Reaktion des getesteten VHDL-Modells (device under test) am Portausgang (hier: y) verglichen mit dem Wert des Testvektors (hier: vektor.y). Mit Hilfe der Assertion- und Report-Anweisung (Kap. 4.7.1) lässt sich im Fehlerfall während der Ausführung der Testbench (Run-Kommando) ein Report am Monitor ausgeben. Weiterhin kann über die Variable Fehler ein abschließender Report am Ende des Tests ausgegeben werden.

Für das schon bekannte VHDL-Modell zu Tabelle 2.11 (Kap. 4.5.4.8) ist in einem Beispiel eine Testbench angegeben. In dem Package „tabelle_pack.vhd“ wird die Komponente tab2_11a deklariert. Nach der Compilierung wird sie in der Work-Library abgelegt und gespeichert. Es kann nun über die Use-Anweisung auf die Komponente tab2_11a zugegriffen werden.

```
-- tabelle_pack.vhd
library ieee;
use ieee.std_logic_1164.all;

package tabelle_pack is  -- Package tabelle_pack.vhd
component tab2_11a       -- Component-Deklaration der Tabelle 2.11
    port (
    x: in std_logic_vector (1 to 4);
    y: out std_logic_vector (1 to 2));
end component;
end tabelle_pack;
```

```
-- VHDL-Modell zu Tabelle 2.11 (Kap.4.5.4.8)
library ieee;
use ieee.std_logic_1164.all;
entity tab2_11a is port (
      x: in std_logic_vector (1 to 4);
      y: out std_logic_vector (1 to 2));
end tab2_11a;

architecture sequent_verhalten of tab2_11a is
begin
  tabelle: process(x) begin      -- Anordnung entspricht der Wahrheitstabelle
    case x is
      when "0000" => y <= "11";
      when "0001" => y <= "11";
      when "0010" => y <= "11";
      when "0011" => y <= "0-";
      when "1000" => y <= "0-";
      when "1001" => y <= "0-";
      when "1010" => y <= "11";
      when "1011" => y <= "01";           -- Fehler "01" statt "11"
      when "1100" => y <= "0-";
      when "1110" => y <= "0-";
      when others => y <= "00"; -- Kombinationen, die "00" ergeben
    end case;
  end process tabelle; -- am Prozessende erhält y den neuen Wert
end sequent_verhalten;

-- test_bench_tab.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.tabelle_pack.all;     -- erlaubt Zugriff auf die Komponente tab_2_11a

entity testbench is
end testbench;

architecture auto_test of testbench is
      signal x: std_logic_vector(1 to 4);
      signal y: std_logic_vector(1 to 2);
type test_vektor is record       -- Stimuli und Erwartungswerte → Record
      x: std_logic_vector(1 to 4);
      y: std_logic_vector(1 to 2);
end record;
type test_vektor_array is array (natural range <>) of test_vektor;
constant test_feld: test_vektor_array:=(
      (x => "0000", y => "11"),
      (x => "0001", y => "11"),
      (x => "0010", y => "11"),
      (x => "0011", y => "0-"),
      (x => "0100", y => "00"),
      (x => "0101", y => "00"),
      (x => "0110", y => "00"),
      (x => "0111", y => "00"),
      (x => "1000", y => "0-"),
```

```
    (x => "1001", y => "0-"),
    (x => "1010", y => "11"),
    (x => "1011", y => "11"),
    (x => "1100", y => "0-"),
    (x => "1101", y => "00"),
    (x => "1110", y => "0-"),
    (x => "1111", y => "00")
);

begin      -- instanziieren der Component tab2_11a
dut: tab2_11a port map (x, y);
testen: process       -- Testvektor zuweisen und pruefen
    variable vektor: test_vektor;
    variable fehler: boolean := false;
begin
for i in test_feld'range loop
    vektor := test_feld(i);
    x <= vektor.x;
    wait for 30 ns;  -- Verzoegerungszeit abwarten
    if y /= vektor.y then        -- Ergebnis ueberpruefen
        assert false
        report "Ergebnis falsch";
        fehler := true;
    end if;
end loop;

    assert not fehler             -- Ausgabe eines Reports
    report "Test ist fehlerhaft"
    severity note;
    assert fehler
    report "Test ist o.K"
    severity note;
    wait;
end process testen;
end auto_test;
```

Bei der Ausführung der Testbench mit ModelSim Xilinx [...] werden folgende Schritte durchgeführt:

- Compilieren des Package tabelle_pack.vhd. Die compilierte Datei wird automatisch in der Work-Library gespeichert und steht für weitere Anwendungen zur Verfügung.
- Compilieren der Testbench test_bench_tab.vhd.
- Laden der einzelnen Komponenten
- Ausführen der Simulation mit dem Kommando „run –all“

Auszug der Monitorausgabe bei der Ausführung mit ModelSim:

```
vsim work.testbench
# vsim work.testbench
# Loading C:/MODELTECH_XE/WIN32XOEM/../std.standard
# Loading C:/MODELTECH_XE/WIN32XOEM/../ieee.std_logic_1164(body)
```

```
# Loading work.tabelle_pack
# Loading work.testbench(auto_test)
# Loading work.tab2_11a(sequent_verhalten)
run -all
# ** Error: Ergebnis falsch
#    Time: 360 ns  Iteration: 0  Instance: /testbench
# ** Note: Test ist fehlerhaft
#    Time: 480 ns  Iteration: 0  Instance: /testbench
```

Da in diesem Beispiel ein Fehler in dem zu testenden VHDL-Modell tab2_11a vorgegeben ist, werden bei der Ausführung der Testbench mit ModelSim Xilinx Fehlermeldungen ausgegeben.

4.9.3 Testbench mit Ein- und Ausgabedatei

Bei der Verwendung von Testbenches lässt sich die Vorgabe der Stimulationswerte und die Ausgabe der Ergebnisse noch verbessern, indem man Textdateien für die Ein- und Ausgabe einsetzt. Die Stimuli werden jetzt als Zeichenfolge in einer Eingabedatei abgelegt und während der Ausführung über standardisierte Prozeduren eingelesen. Dadurch können die Werte für die Stimulation leicht geändert werden, ohne dass eine neue Compilierung der Testbench erfolgen muss. Entsprechend werden die Ergebnisse in formatierter Form über Ausgabeprozeduren als Text in einer Ausgabedatei gespeichert, die dann vom Anwender leichter ausgewertet werden kann.

Die Verwendung der Ein- und Ausgabeprozeduren ist mit größerem Aufwand verbunden, da die Standard-Lese- und Schreibprozeduren nicht für den Datentyp std_logic bzw. std_logic_vector gelten. Im Folgenden werden erweiterte Ein- und Ausgabeprozeduren [26] vorgestellt, die mit Hilfe von Overloading den Datentyp std_logic berücksichtigen. Man kann das Package „text_io_pack.vhd“ anwenden, ohne alle Einzelheiten der VHDL-Beschreibung zu kennen. Es kann in ähnlicher Weise benutzt werden wie ein Package mit einer VHDL-Deklaration, das eine Addition von Vektoren (Kap. 6.2.2.2) erlaubt. Das unten beschriebene Package „text_io_pack .vhd“ kann für alle Aufgaben, die mit den Datentypen „std_logic“ und „std_logic_vector“ in Ein- und Ausgabedateien arbeiten, verwendet werden.

Im folgenden Beispiel wird eine Testbench vorgestellt, die für die gleiche Aufgabenstellung mit der Wahrheitstabelle (Kap. 4.9.2) eingesetzt wird. Es wird vorausgesetzt, dass sich das VHDL-Modell tab2_11a als compilierte Komponente in dem Package work.tabelle_pack.vhd befindet (s. Kap. 4.9.2).

```
-- text_io_pack.vhd
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;


package text_io_pack is        -- Erweiterung der Ein-/Ausgabe auf den Datentyp std_logic
```

```
    procedure read (L: inout Line; wert: out std_logic; gut: out boolean);
    procedure read (L: inout Line; wert: out std_logic);
    procedure read (L: inout Line; wert: out std_logic_vector; gut: out boolean);
    procedure read (L: inout Line; wert: out std_logic_vector);
    procedure write (L: inout Line; wert: in std_logic; justified:in side := right; field: in
    width := 0);
    procedure write (L: inout Line; wert: in std_logic_vector; justified:in side := right;
    field: in width := 0);
    type std_logic_chars is array (character) of std_logic;
    constant to_stdlogic: std_logic_chars:=
    ('U' => 'U','X' => 'X','0' => '0','1' => '1','Z' => 'Z',
    'W' => 'W','L' => 'L','H' => 'H','-' => '-',others => 'X');
    type character_chars is array (std_logic) of character;
    constant to_character: character_chars :=
    ('U' => 'U','X' => 'X','0' => '0','1' => '1','Z' => 'Z',
    'W' => 'W','L' => 'L','H' => 'H','-' => '-');
end text_io_pack;

package body text_io_pack is
    procedure read (L: inout Line; wert: out std_logic; gut: out boolean) is
        variable temp: character;
        variable gut_character: boolean;
    begin
        read (L, temp, gut_character);
        if gut_character = true then
                gut := true;
                wert := to_stdlogic(temp);
        else    gut := false;
        end if;
    end read;
    procedure read (L: inout Line; wert: out std_logic) is
        variable temp: character;
        variable gut_character: boolean;
    begin
        read (L, temp, gut_character);
        if gut_character = true then  wert := to_stdlogic (temp);
        end if;
    end read;
    procedure read (L: inout Line; wert: out std_logic_vector; gut: out boolean) is
        variable temp: string(wert'range);
        variable gut_string: boolean;
    begin
        read (L, temp, gut_string);
        if gut_string = true then
                gut := true;
                for i in temp'range loop
                  wert(i) := to_stdlogic(temp(i));
                end loop;
        else gut := false;
        end if;
    end read;
```

```
procedure read (L: inout Line; wert: out std_logic_vector) is
        variable temp: string(wert'range);
        variable gut_string: boolean;
    begin
        read (L, temp, gut_string);
        if gut_string = true then
                for i in temp'range loop
                  wert(i) := to_stdlogic(temp(i));
                end loop;
        end if;
    end read;
    procedure write (L: inout Line; wert: in std_logic; justified:in side := right; field: in
    width := 0) is
        variable write_wert: character;
    begin
        write_wert := to_character(wert);
        write(L,write_wert, justified, field);
    end write;

    procedure write (L: inout Line; wert: in std_logic_vector; justified:in side := right;
        field: in width := 0) is
        variable write_wert: string(wert'range);
    begin
        for i in wert'range loop
          write_wert(i) := to_character(wert(i));
          end loop;
        write(L,write_wert, justified, field);
    end write;
end text_io_pack;

-- Testbench mit Ein- und Ausgabedatei:  tb_textio_tab.vhd
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.text_io_pack.all;
use work.tabelle_pack.all;

entity tb_textio is
end tb_textio;
architecture auto_test of tb_textio is
signal x: std_logic_vector(1 to 4);
signal y: std_logic_vector(1 to 2);
begin
dut: tab2_11a port map (x => x, y => y);
testen: process
    file eingabe_datei: text is in "tab_io_test.txt";              -- Eingabedatei: tab_io_test.txt
    file ausgabe_datei: text is out "ergebnis_aus.txt";     -- Ausgabedatei: ergebnis_aus.txt
    variable zeile_ein, zeile_aus: line;
    variable aus_y: std_logic_vector(1 to 2);
    variable v_x: std_logic_vector(1 to 4);
    variable v_y: std_logic_vector(1 to 2);
    variable fehler: boolean := false;
    variable gut: boolean;
```

```
    variable char: character;
    variable fehler_aus: string(1 to 4) := "nein";
    constant abstand_2: string(1 to 2) := "  ";
    constant abstand_3: string(1 to 3) := "   ";
    constant ueber: string(1 to 21) := "  X   Y  Ysoll Fehler"; -- Ueberschrift Ausgabedatei
begin
    write(zeile_aus, ueber);   -- Ueberschrift
    writeline(ausgabe_datei, zeile_aus); -- Ausgabe der Zeile
    writeline(ausgabe_datei, zeile_aus); -- Ausgabe einer Leerzeile
zeile_loop: while not endfile(eingabe_datei) loop
    readline (eingabe_datei,zeile_ein); -- Zeile einlesen
    read (zeile_ein,char,gut);
-- ueberspringe Zeile, falls Zeichen kein Tabulator s. Tab. 4.10
    if not gut or char /= HT then next;
    end if;
    assert gut
        report "Fehler beim Lesen"
    severity note;
    read (zeile_ein,v_x,gut);
    next when not gut;
    read (zeile_ein,char);
    read (zeile_ein,v_y);
    wait for 20 ns;
    x <= v_x;        -- Typ-Konvertierung: variable → signal
    wait for 30 ns;
-- ueberpruefen des Ergebnisses
    aus_y := y;
    if aus_y /= v_y then
        assert false
        report "y falsch";
        fehler := true;
        fehler_aus := " ja ";
    end if;
 -- formatierter Ausgabereport
    write(zeile_aus, v_x);              -- Stimuli x
    write(zeile_aus, abstand_2);        -- 2 Leerzeichen
    write(zeile_aus, aus_y);            -- Ergebniswert y
    write(zeile_aus, abstand_2);        -- 2 Leerzeichen
    write(zeile_aus, v_y);              -- Erwartungswert für y
    write(zeile_aus, abstand_3);        -- 3 Leerzeichen
    write(zeile_aus, fehler_aus);       -- Fehler: ja oder nein
    writeline(ausgabe_datei, zeile_aus);
     fehler_aus := "nein";              -- Defaultwert für Fehler
end loop zeile_loop;
assert not fehler
    report "Test ist fehlerhaft"
    severity note;
    assert fehler
    report "Test ist o.K"
    severity note;
    wait;
end process testen;
end auto_test;
```

Tabelle 4.10: Ein- und Ausgabedatei, die innerhalb der Testbench verwendet werden

eingabe_datei: tab_io_test.txt	ausgabe_datei: ergebnis_aus.txt
Wahrheitstabelle 2.11	X Y Ysoll Fehler
0000 11	0000 11 11 nein
0001 11	0001 11 11 nein
0010 11	0010 11 11 nein
0011 0-	0011 0- 0- nein
0100 00	0100 00 00 nein
0101 00	0101 00 00 nein
0110 00	0110 00 00 nein
0111 00	0111 00 00 nein
1000 0-	1000 0- 0- nein
1001 0-	1001 0- 0- nein
1010 11	1010 11 11 nein
1011 11	1011 01 11 ja
1100 0-	1100 0- 0- nein
1101 00	1101 00 00 nein
1110 0-	1110 0- 0- nein
1111 00	1111 00 00 nein

Die Tabelle 4.10 enthält auf der linken Seite die Eingabedatei und auf der rechten Seite die Ausgabedatei mit der Fehlermeldung. Das erste Zeichen einer relevanten Eingabedateizeile ist der Tabulator. Dadurch können in der Eingabedatei auch Überschriften und Kommentarzeilen verwendet werden, die beim Einlesen der Werte über die Eingabeprozedur ignoriert werden.

7.1.13 Das FRAM

Wie in den vorangehenden Kapiteln dargestellt wurde, sind heute leistungsfähige Halbleiterspeicher hoher Kapazität mit geringen Zugriffszeiten auf DRAM-Basis am Markt verfügbar. Auch im Bereich der nichtflüchtigen Speicher gibt es ein weitgefächertes Angebot, wie EPROMS, EEPROMs und Flash-EPROMs. Beide Speichertypen befriedigen jedoch nicht Ansprüche, wie sie zunehmend im Bereich mobiler Anwendungen gestellt werden. Insbesondere hier besteht Bedarf an Halbleiterspeichern, die günstige Eigenschaften beider Gruppen in sich vereinen.

Seit ca. 20 Jahren wird ein Speichermedium erforscht, das diese Forderung erfüllen könnte, nämlich ferroelektrische Speicher. Die Bezeichnung FRAM wurde von der Fa. Ramtron geschützt. Andere Hersteller nennen diesen Speichertyp daher FeRAM. Nach Einschätzung von Fachleuten werden ferroelektrische und magnetoresistive Speicher (MRAMs, s.Kap. 7.1.14) die Speichertechnologie in nächster Zukunft dominieren. Ein qualitativer Vergleich wesentlicher Eigenschaften von Halbleiterspeichern macht dieses deutlich (Tabelle 7.1). Es werden FRAMs angestrebt, die als Universalspeicher bisherige Speichertypen ersetzen sollen (All-in-One-Solution).

Tabelle 7.1: Wesentliche Eigenschaften unterschiedlicher Speichertechnologien

Eigenschaften	**SRAM**	**DRAM**	**EEPROM**	**FLASH**	**FRAM MRAM**
Nichtflüchtig	nein	nein	ja	ja	ja
Kleine Zellenmaße	nein	ja	nein	ja	ja
Wortweise les- und beschreibbar	ja	ja	ja	nein	ja
Geringer Leistungsbedarf	ja	ja	nein	nein	ja
Schneller Schreibzugriff	ja	ja	nein	nein	ja
10^{15} Schreibzyklen	ja	ja	nein	nein	ja
Kostengünstig	nein	ja	nein	ja	ja

Wird an einen Kondensator eine elektrische Spannung angelegt, nimmt er eine Ladung Q auf, und zwischen seinen Belägen entsteht ein elektrisches Feld. Dieses führt zu einer Ladungsverschiebung im Dielektrikum, die man als dielektrische Polarisation bezeichnet. Wird der Kondensator entladen, verschwindet auch die Polarisation.

Unter dem *ferroelektrischen Effekt* versteht man die Eigenschaft einiger Isolierstoffe, durch Anlegen geeigneter elektrischer Felder spontan einen von zwei unterschiedlichen Polarisationszuständen einnehmen zu können, und diesen Zustand auch nach Entfernen der Spannung als *remanente Polarisation* beizubehalten. Diese Eigenschaft beruht auf der speziellen Kristallstruktur des Stoffes und ist nutzbar bei würfelförmigen Perovskit-Kristallen aus Blei-Zirkonium-Titanat (PZT) und bei geschichteten Perovskit-Kristallen aus Strontium-Wismut-Tantal (SBT).

Nutzt man diese Stoffe als Dielektrikum in einem Kondensator, erhält man ein bistabiles Element, das zum Speichern digitaler Information geeignet ist und *ferro-*

elektrischer Kondensator (C_{FE}) heißt. Das Speicherprinzip ist in Bild 7.27gezeigt. Im Zentrum des Kristalls befindet sich in Abhängigkeit von der Legierung ein Titan- oder Zirkoniumatom. Die unterschiedliche bistabile Position dieses Atoms bestimmt die Polarisationsrichtung des Kristalls.

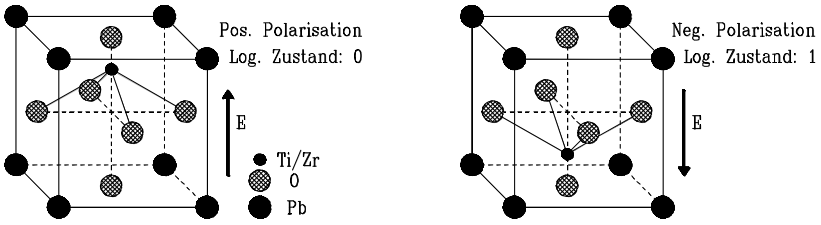

Bild 7.27: Ferroelektrische Perovskit-Kristalle (PZT), die durch elektrische Feldstärken E in unterschiedliche stabile Polarisationsrichtungen versetzt worden sind

Der funktionelle Zusammenhang zwischen der angelegten Spannung U und der aufgenommenen Ladung Q zeigt eine Hysterese, deren Form der Magnetisierungskurve ferromagnetischer Stoffe vergleichbar ist, wie **Bild 7.28** zeigt.

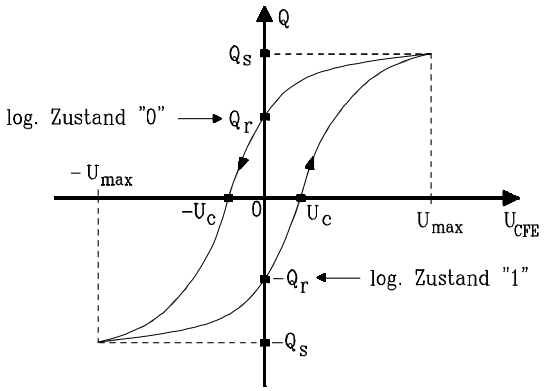

Bild 7.28: Zusammenhang zwischen Spannung U_{CFE} und aufgenommener Ladung Q bei einem ferroelektrischen Kondensator C_{Fe}. Die Abkürzungen bedeuten:
Q_s: Sättigungsladg., Q_r: Remanentladg, U_{CFE}: Kondensator- und U_c: Koerzitivspannung

Wird der Kondensator an eine Spannung $+U_{max}$ gelegt und dann die Spannungsquelle entfernt, wird der stabile Arbeitspunkt Q_r angenommen. Damit ist die log. „0“ gespeichert. Der log. Zustand „1“ wird erreicht, indem kurzzeitig die Spannung $-U_{max}$ angelegt wird. Diese Vorgänge sind dem Speichervorgang in einem magnetischen Ringkernspeicher äquivalent, der heute nur noch für Spezialzwecke verwendet wird, da er Ansprüche an Kapazität, Geschwindigkeit und Steuerleistung nicht mehr erfüllt.

Die Organisation der einzelnen Speicherzellen in einem FRAM entspricht prinzipiell der bei DRAMs verwendeten Technik: Jede ferroelektrische Zelle wird über einen n-Kanal-Enhancement-Transistor angesteuert. Daraus entsteht die sog. 1T-1C-Zelle (1 Transistor, 1 Kapazität, Bild 7.29). Sie enthält als Speicherelement den ferroelektrischen Kondensator C_{Fe} . Die Zelle wird angesprochen durch einen ausreichend großen H-Pegel an der Wortleitung WL. Dadurch wird der Transistor leitfähig, und

C_{Fe} kann beschrieben oder gelesen werden. C_{BL} repräsentiert die parasitäre Kapazität der Bitleitung.

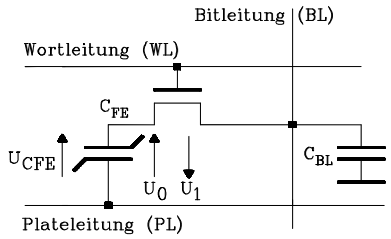

Bild 7.29: Ferroelektrische 1T-1C-Speicherzelle, bestehend aus Zugangstransistor, Speicherkondensator C_{Fe}, und der parasitären Kapazität der Bitleitung C_{BL}

Schreibvorgang: Das Liniendiagramm ist in Bild 7.30 gezeigt. Die Speicherzelle befinde sich für $t < t_0$ unselektiert in einem der beiden Zustände „0“ oder „1“.

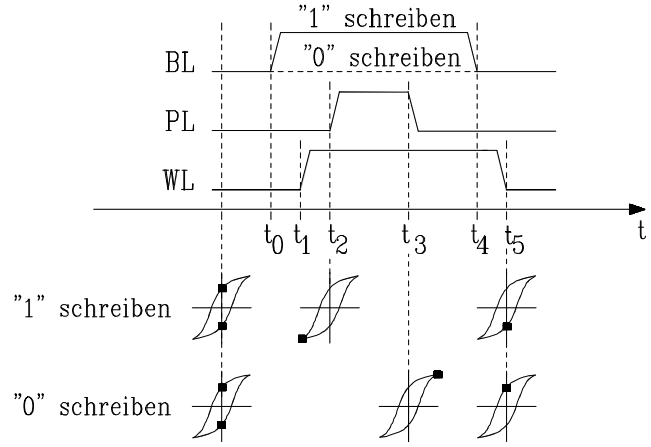

Bild 7.30: Schreibvorgang bei einer FRAM-Speicherzelle. Wichtige Zustände des ferromagnetischen Kondensators sind an der Hystereseschleife verdeutlicht.
Die Abkürzungen bedeuten: BL: Bitleitung PL: Plateleitung WL: Wortleitung

Schreiben einer „1“: Alle drei Steuersignale befinden sich zunächst auf 0V. Der Speicherzugriff beginnt mit:

- **t_0**: Die Bitleitung (BL) geht vorbereitend auf H-Pegel.
- **t_1**: Die Wortleitung (WL) geht auf H-Pegel. Der Transistor leitet, und C_{Fe} wird über den ON-Widerstand des Transistors auf H-Pegel geladen. Damit befindet sich die Zelle in der Hystereseschleife im Punkt $-Q_S$.
- Zum Zeitpunkt **t_2** wird ein kurzer positiver Impuls auf die Plateleitung PL gegeben, der hier keine Bedeutung hat.
- Nach dessen Ende bei **t_3** steht an C_{Fe} wieder H-Pegel, da WL und BL noch gesetzt sind. Damit gilt: $U_{CFE} = -U_1 = -U_{max}$.
- Zum Zeitpunkt **t_4** geht BL auf „0“ bei leitendem Transistor, d.h. $U_{CFE} = 0$ und C_{Fe} geht in den Zustand remanenter Polarisation Q_r, speichert also das Bit „1“.

- Bei $\mathbf{t_5}$ wird die Speicherzelle durch WL deselektiert und der Zustand „1“ bleibt ohne weitere Energiezufuhr erhalten.

Schreiben einer „0“: BL bleibt stets auf L-Pegel.

- Bei $\mathbf{t_1}$ nimmt WL H-Pegel an, der Transistor leitet. Der Kondensator C_{Fe} liegt an der Spannung $U_{CFE} = 0V$, d.h. sein logischer Zustand bleibt unverändert.
- Im Intervall $\mathbf{t_2}$ bis $\mathbf{t_3}$ geht PL auf „1“, dadurch wird $U_{CFE} = U_0 = +U_{max}$, d.h. C_{Fe} befindet sich auf der Hysteresekurve im 1. Quadranten bei Q_S .
- Im Intervall $\mathbf{t_3}$ bis $\mathbf{t_4}$ beträgt $U_{CFE} = 0$ und C_{Fe} befindet sich im Zustand remanenter Polarisation im Punkt $+Q_r$, hat also das Bit „0“ gespeichert.
- Bei $\mathbf{t_5}$ wird die Speicherzelle durch WL deselektiert und der Zustand „0“ bleibt ohne weitere Energiezufuhr erhalten.

Lesevorgang: Der Lesevorgang an einer FRAM-Speicherzelle besteht prinzipiell darin, dass der ferromagnetische Kondensator C_{FE} bei hochohmig geschalteter Bitleitung einen Teil seiner Ladung auf die parasitäre Kapazität C_{BL} überträgt. Die dabei transportierte Ladung und daher auch die an C_{BL} entstehende Spannung hängen davon ab, ob vorher eine „0“ oder eine „1“ gespeichert war. Ein Sense-Verstärker führt diese Auswertung durch. Daraus folgt, dass es sich um ein zerstörendes Leseverfahren handelt. Der ursprüngliche Wert muss anschließend wieder rückgespeichert werden. Bild 7.31 zeigt einen Lesevorgang nach der Methode des *Step-Sensing-Approach.* Dieser Vorgang hat sehr große Ähnlichkeit mit dem Leseverfahren an einem DRAM.

Die FRAM-Speicherzelle befinde sich für $t << \mathbf{t_0}$ unselektiert in einem der beiden Zustände „0“ oder „1“. Dann folgen:

- $t < \mathbf{t_0}$: Eine Leseoperation beginnt mit einem Precharge-Vorgang, der zunächst die Bitleitung hochohmig schaltet und anschließend die parasitäre Kapazität C_{BL} entlädt. Der Speicherinhalt verändert sich dabei nicht. Zum Zeitpunkt $\mathbf{t_0}$ ist dieses abgeschlossen.

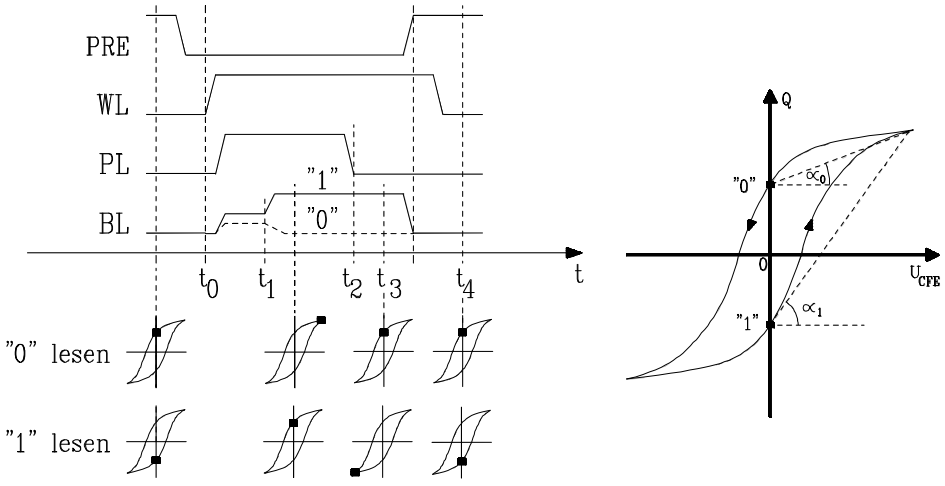

Bild 7.31: Lesevorgang bei einer FRAM-Speicherzelle mit Zuständen des ferromagnetischen Kondensators in der Hystereseschleife und Näherungen für die Kapazitäten von C_{FE} : $\tan \alpha 1 = C_1$ und $\tan \alpha 0 = C_0$. Die Abkürzungen bedeuten:
PRE: Prechargesignal BL: Bitleitung PL: Plateleitung WL: Wortleitung

- Bei $\mathbf{t_0}$ geht die Wortleitung auf H-Pegel und schaltet den Transistor in den leitenden Zustand. Dadurch sind C_{FE} und C_{BL} zwischen PL und Masse in Serie geschaltet und bilden einen Spannungsteiler.
- Kurze Zeit später wird die Plateleitung auf Betriebsspannungspotential (U_{CC}) angehoben. Der dadurch verursachte Ladungstransport, und damit das Spannungsteilerverhältnis zwischen C_{FE} und C_{BL}, hängt vom aktuellen Wert von C_{FE} ab. Für die Kapazität gilt: $C_{FE} = \Delta U_{CFE}/\Delta Q$, und dieser Quotient wird vom Betriebspunkt auf der Hystereseschleife während des Ladeprozesses bestimmt (s. Bild 7.31 rechts).
- Approximiert man die beiden Kapazitäten als Geraden, ergibt sich überschlägig

$$U_{CBL} = U_{CC} \cdot \frac{C_{FE}}{C_{FE} + C_{BL}}$$

für $C_0 = \tan \alpha 0$ und für $C_1 = \tan \alpha 1$ mit $C_1 > C_0$. An der Bitleitung liegt dann die Spannung U_{CBL} und es gilt $U_{CBL}(C_{FE} = C_1) > U_{CBL}(C_{FE} = C_0)$. Diese Verhältnisse sind an der Stelle $t = \mathbf{t_1}$ in Bild 7.31 erkennbar.
- Zum Zeitpunkt $t = \mathbf{t_1}$ wertet der Sense-Verstärker die Spannung U_{CBL} aus und erkennt den Wert des gelesenen Bits. War es „1“, legt er die Spannung U_{CC} an die Bitleitung, ansonsten 0V.
- Da WL weiterhin auf H-Pegel liegt und zum Zeitpunkt $t = \mathbf{t_3}$ L-Pegel an PL liegt, wird der gelesene Bitwert in den Speicher zurückgeschrieben.
- Der Lesezyklus endet bei $\mathbf{t_4}$ durch WL = L-Pegel und der gelesene Zustand bleibt ohne weitere Energiezufuhr in der Speicherzelle erhalten.

Die maximale Datenhaltung (data retention) für FRAMs wird heute mit 10 Jahren beziffert. Gründe für die Begrenzung sind im Wesentlichen:

- Mit zunehmendem Alter tritt eine Depolarisation auf. Sie äußert sich dadurch, dass der remanente Ladungsbetrag $|Q_r|$ für beide Betriebspunkte abnimmt (Fatigue). Der Effekt ist auch durch die Anzahl der Zugriffe begründet.
- Wenn FRAM-Zellen überwiegend einen festen log. Zustand speichern, passt sich die Hystereseschleife diesem Zustand an, indem der remanente Ladungsbetrag $|Q_r|$ des gegenüberliegenden Arbeitspunktes abnimmt (Imprint).

Durch Weiterentwicklung der Schaltungskonzepte für Speicherzellen und Sense-Verstärker wird versucht, die Spannungsunterschiede zwischen den Lesesignalen für „0“ und „1“ zu vergrößern, um die Wahrscheinlichkeit von Lesefehlern zu reduzieren. Die oben beschriebene 1T-1C-Zelle wird sich künftig voraussichtlich gegenüber der seit längerer Zeit auf dem Markt befindlichen 2T-2C-Zelle durchsetzen, da sie weniger Platz benötigt und Kosten spart.

Wesentliche Betriebsdaten von FRAMs sind in der Tabelle 7.2 denen anderer Speicherkonzepte gegenübergestellt.

Tabelle 7.2: Gegenüberstellung wesentlicher Daten unterschiedlicher Speichertypen

Eigenschaften	SRAM	DRAM	NAND-FLASH	NOR-FLASH	MRAM (FET)	FRAM
Zellengröße/µm²	100	8	1,3	2,5	>8	4-20
Betriebsspannung /V	2,5	2,5	1,8	3,3	1,8-5	3-5
Datenhaltung/a	<<1 mit Batterie	flüchtig	10	10	10	10
Lesezeit/ns random	2	60	10^4	60-90	10-50	70
Schreibzeit/ns random Program / Erase Speed	2	60	2,1/5,3 MB/s	0,2/0,08 MB/s	10-40	70
Lesezyklenzahl	$> 10^{15}$	$> 10^{15}$	$> 10^{15}$	$> 10^{15}$	$> 10^{15}$	10^{12}-10^{15}
Schreibzyklenzahl	$> 10^{15}$	$> 10^{15}$	10^5	10^5	$> 10^{15}$	10^{10}-10^{15}

7.1.14 Das MRAM

Im Kapitel 7.1.13 wird mit dem FRAM ein neuer Speichertyp vorgestellt, der ähnliche Eigenschaften wie ein DRAM besitzt, aber nichtflüchtig ist und daher neue Möglichkeiten im Bereich mobiler Anwendungen und als Arbeitsspeicher in der Computertechnik bietet.

Auf diese Anwendungsbereiche zielt auch ein anderer nichtflüchtiger Speichertyp, das *Magnetoresistive RAM* (MRAM). Es hat gegenüber dem FRAM einen weiteren Vorteil, denn es lässt sich zerstörungsfrei lesen. Damit entfällt das Rückspeichern der gelesenen Information, welches Zeit und Energie benötigt. Insgesamt hält damit in der Halbleiterindustrie eine neue Entwicklung Einzug, die *Magnetoelektronik*, welche die moderne Halbleiterprozesstechnologie mit der Technologie ferromagnetischer Schichten verbindet. Die Grundlagen der Magnetoelektronik wurden 1989 gelegt, als es gelang, den elektrischen Stromfluss in sehr dünnen Metallschichten durch ein magnetisches Feld zu beeinflussen (*Giant Magnetoresistance*, GMR).

Ein weiterer, mit GMR verwandter magnetoelektrischer Effekt beruht darauf, dass eine nur etwa vier Atomlagen dünne dielektrische Schicht (Al_2O_3) zwischen zwei ferromagnetischen Metallbelägen einen Tunnelstrom führen kann, der sich durch die Orientierung der magnetischen Dipolmomente in den Metallbelägen verändern lässt. Eine derartige Zelle heißt *Magnetische Tunnelbarriere* (Magnetic Tunnel Junction, MTJ) und wird vorwiegend als Speicherzelle verwendet..

Der Entwicklungsstand von Speicherbauelementen, die dieses Prinzip nutzen, hinkt etwa drei Jahre hinter dem von FRAMs her. Aufgrund einer weltweit vollzogenen Konzentration der Entwicklungsarbeiten wird aber bereits 2004 mit marktreifen Produkten gerechnet.

Aufbau und Funktion einer MTJ-Speicherzelle: Sie besteht aus einem Stapel zweier ferromagnetischer Schichten, die durch ein dünnes Dielektrikum voneinander getrennt sind. Wird an diese Zelle eine Spannung gelegt, fließt ein Tunnelstrom, dessen

Größe davon abhängt, ob die Orientierung des magnetischen Feldes in den ferromagnetischen Schichten parallel oder antiparallel ist (Bild 7.32).

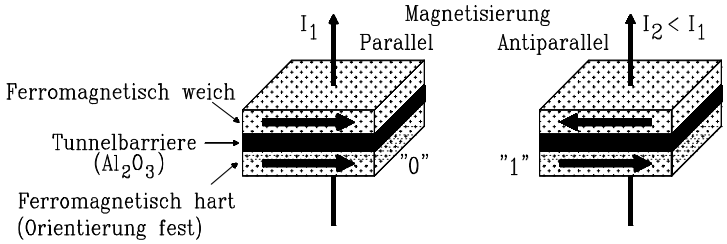

Bild 7.32: Funktionsweise eines MTJ-Speicherelements (Magnetic Tunnel Junction). Die Richtung des magnetischen Feldes in der ferromagnetisch weichen Schicht verändert den Tunnelstrom. Daher besitzt das Element zwei stabile Zustände, denen die log. Zustände „0“ bzw. „1“ zugeordnet sind.

Verursacht wird dieses Verhalten durch eine Spin-Polarisation der Leitungselektronen in den ebenfalls nur wenige Atomlagen dicken ferromagnetischen Elektroden, die von der Orientierung des Magnetfeldes bestimmt wird. Sind etwa Leitungselektronen nach dem Passieren der ersten Elektrode in einer Richtung 1 polarisiert, wird ihr Durchgang durch die zweite Elektrode behindert, wenn diese für die andere Polarisationsrichtung 2 eingestellt ist. Daher bewirkt die parallele magnetische Feldorientierung in beiden Leitern gegenüber antiparalleler einen um bis zu 50% geringeren Widerstand. Diesen Effekt nennt man *Magnetoresistanz*.

Die Speicherzelle hat also infolge der Remanenz im magetisch weichen Leiter zwei stabile Zustände, die sich durch ihre Leitfähigkeit unterscheiden und den log. Zuständen „0“ oder „1“ entsprechen. Durch eine zusätzliche Leitung, die isoliert an der ferromagnetisch weichen Elektrode vorbeiläuft, lässt sich deren magnetische Orientierung umkehren und damit der log. Zustand der Zelle verändern.

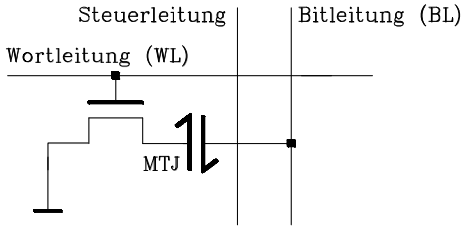

Bild 7.33: Magnetoresistive 1T-1MTJ-Speicherzelle. MTJ bedeutet: Magnetic Tunnel Junction. Die Steuerleitung ist für Schreiboperationen erforderlich.

Die Einordnung in ein Speicherarray ist beim MRAM prinzipiell möglich, indem man eine Elektrode der Zelle mit der Bitleitung und die andere mit der Wortleitung direkt verbindet (Kreuzpunkt-Zelle). Zur Reduzierung parasitärer Leckströme müssen die Zellen dann aber sehr hochohmig gefertigt werden. Dadurch sinkt wegen der parasitären Kapazitäten des Systems die Geschwindigkeit um etwa drei Zehnerpoten-

zen. Sind kleine Zugriffszeiten nötig, schließt man jede Zelle über einen Transistor an das Array an und erhält damit eine 1T-1MTJ-Zelle (Bild 7.33).

Zur Erläuterung der Funktionsweise eines MTJ-Speichers ist in Bild 7.34 ein vereinfachter Ausschnitt aus der räumlichen Anordnung eines Speicher-Arrays dargestellt. Es besteht aus zwei Wort- und zwei Bitleitungen und enthält die vier Speicherzellen Z11, Z12, Z21 und Z22 mit ihren Zugangstransistoren. Zwei zusätzliche Steuerleitungen sind für den Schreibvorgang nötig. Bit- und Steuerleitungen sind voneinander isoliert. Die Zugangstransistoren sind nur bei Leseoperationen durchgeschaltet.

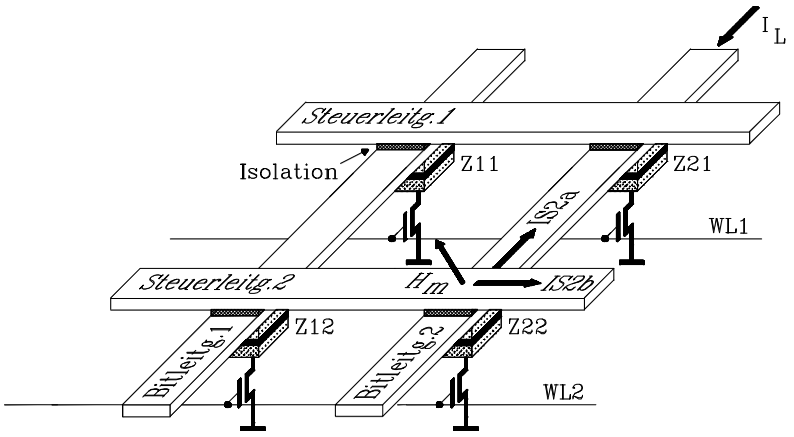

Bild 7.34: Ausschnitt aus einem 1T-1MTJ-Array mit 4 Speicherzellen. Die Abkürzungen bedeuten: MTJ: Magnetic Tunnel Junction WLi: Wortleitungen Zii: MTJ-Zellen

Lesevorgang: Betrachtet werde ein Lesevorgang an der Speicherzelle Z22. Dazu wird die Wortleitung WL2 aktiviert und damit der zugeordnete Transistor leitfähig. Anschließend wird ein konstanter Lesestrom I_L auf die Bitleitung 2 geschaltet, der die Zelle Z22 durchtunnelt. Die Spannung an der Bitleitung hängt nun vom log. Zustand, also der Leitfähigkeit des MTJ-Elements ab, sie wird durch einen Sense-Verstärker ausgewertet und liefert das gespeicherte Bit. Das MTJ-Element ändert während des Lesevorgangs seinen Zustand nicht, daher handelt es sich um einen nicht-zerstörenden Leseprozess, ein besonderer Vorteil dieses Speichertyps.

Schreibvorgang: Alle Zugangstransistoren in den Wortleitungen sind gesperrt. Es werde die Speicherzelle Z22 beschrieben. Dazu fließen Ströme in den beiden isoliert voneinander angeordneten Leitungen: In Bitleitung 2 fließt der Strom IS2a und in der Steuerleitung der Strom IS2b. Beide Ströme verursachen Magnetfelder, die sich an der oberen, ferromagnetisch weichen Elektrode der MTJ-Zelle vektoriell zur Feldstärke H_m addieren. Die Ströme sind so bemessen, dass H_m die Elektrode in magnetische Sättigung bringt. Nach Wegnahme der Ströme verbleibt die Elektrode im Remanenzpunkt. Die Zelle enthält damit z. B. den log. Zustand „0“ und dieser ist ohne Zufuhr von Energie stabil, es handelt sich also um einen nichtflüchtigen Speicher. Zum Schreiben einer „1“ ist die Richtung des Stroms IS2a der Bitleitung umzukehren.

Im Kap. 7.1.13 ist in Tab. 7.2 ein Vergleich wesentlicher Eigenschaften unterschiedlicher Speicherkonzepte dargestellt. Daraus geht hervor, dass *Magnetoresistive*

RAMs (MRAMs) gegenüber *Ferromagnetischen RAMs* (FRAMs) Vorteile bezüglich einer höheren Schreib-/Lesezyklenzahl und geringerer Zugriffszeiten haben.

Es wird damit gerechnet, dass die neuen nichtflüchtigen Speicher MRAMs und FRAMs die künftige Speicherlandschaft revolutionieren. Beispielsweise wird heute an Konzepten gearbeitet, diese Speichertypen in Computern künftig nicht nur als Arbeits-, sondern auch als Massenspeicher einzusetzen. Damit würden der Bootprozess überflüssig und Massenspeicherzugriffe erheblich beschleunigt.