Technische Hochschule Nürnberg Georg Simon Ohm
Fakultät Elektrotechnik Feinwerktechnik Informationstechnik

Studienarbeit

Im Fach Smart Systems Design / Teil B (ESY1/B)

SPI FRAM Controller

Name: Fischer Armin
Matrikelnummer: 3242752

Datum der Ausgabe: 02.06.2021
Datum der Abgabe: 09.06.2021 (13 Uhr)

Prüfer:
Prof Dr. Claus Kuntzsch
: Prof Dr.-Ing. Jürgen Krumm

Inhaltsverzeichnis:

Einleitung

Im Rahmen der Studienarbeit für ESY1/B soll ein Controllermodul für einen FRAM programmiert werden. Das Modul wird mit der Hardwarebeschreibungssprache System Verilog programmiert. Als IDE und Simulationswerkzeug kam EDA-Playground zum Einsatz.

SPI

Da der eingesetzte FRAM über SPI kommuniziert, wird in diesem Kapitel die allgemeine Funktionsweise eines SPI Busses erläutert.
SPI oder Serial Peripheral Interface ist ein Synchron Serieller, Voll-Duplex Bus und arbeitet nach dem Master Slave Prinzip.
Dieser Bus wird häufig in Verbindung mit Zusatzkomponenten, wie Sensoren, ADCs oder auch Speichererweiterungen genutzt.

Interface

Das SPI Interface besteht in der Regel aus vier Leitungen, CLOCK, MOSI, MISO und nCS.
Der Master stellt das Taktsignal zur Verfügung und wählt über den nCS Anschluss den jeweiligen Slave aus. Alle Slaves teilen sich die Takt- und Datenleitungen mit dem Master, mit Ausnahme des nCS Pins. Jeder Slave hat seinen eigenen Chip Select Pin, um explizit aktiviert zu werden. [1]

nCS: ChipSelect (Low aktiv)
CLK: Takt
MOSI: Master Out, Slave In
MISO: Master In, Slave Out

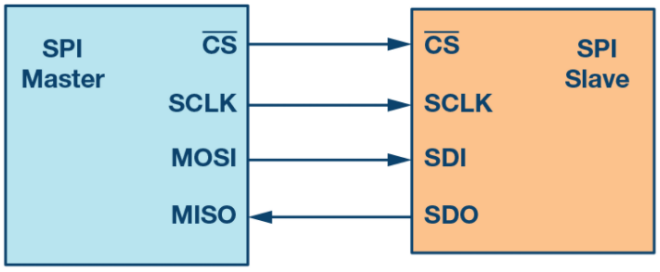

Abbildung 1 SPI Interface

Protokoll

Um eine Übertagung zu beginnen, muss der Master per CS Pin den gewünschten Slave aktivieren (Leitung auf Low ziehen) und einen Takt bereitstellen. Der Master sendet daraufhin Daten über MOSI an den Slave und erhält über MISO die ausgelesenen Daten.
Legt der Master auf die nCS Leitung wieder eine Logische 1, so ist dies das Signal für das Ende der Übertragung.[1]–[3]

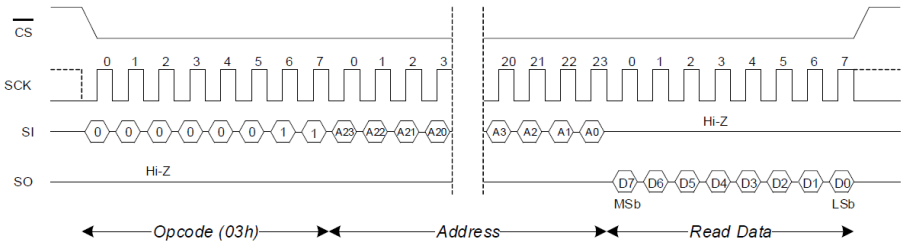

Abbildung 2 SPI Übertragung für FRAM Read Mem

Modi

Da die SPI Übertragung ansonsten kein festgelegtes Protokoll aufweist, haben sich vier Modi etabliert.

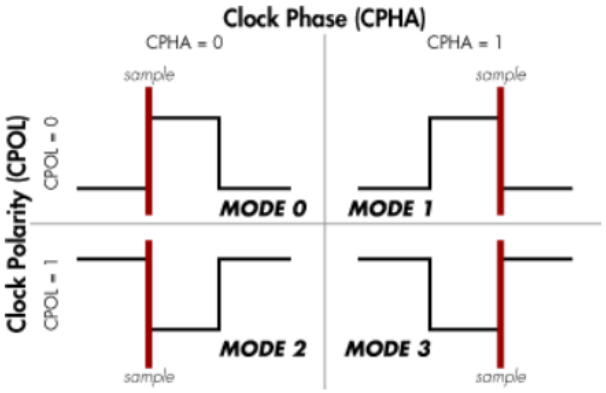

Abbildung 3 SPI Modi

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Tabelle 1 SPI Modi

MODE 0: Clock Leitung im Idle Status auf Low (CPOL 0). Daten werden auf steigender Taktflanke abgetastet (CPHA 0) und bei fallender Flanke ausgegeben.

MODE 1: Clock Leitung im Idle Status auf High (CPOL 1), Daten werden auf fallender Taktflanke abgetastet (CPHA 1) und bei steigender Flanke ausgegeben.

MODE 2: Clock Leitung im Idle Status auf High (CPOL 1), Daten werden auf steigender Taktflanke abgetastet (CPHA 0) und bei fallender Flanke ausgegeben.

MODE 3: Clock Leitung im Idle Status auf High (CPOL 1), Daten werden auf fallender Taktflanke abgetastet (CPHA 1) und bei steigender Flanke ausgegeben
Vgl.[4]

Welcher Modi, unterstützt wird kann im Datenblatt des jeweiligen Peripherie Baustein nachgelesen werden.

Der FRAM Chip CY15B108QN unterstützt die Modi 0 und 3 [3, S. 7]. Die verwendete SPI Library unterstützt alle vier Modis.

FRAM

Der CY15B108QN ist ein sogenannter FRAM, was für Ferroelectric-RAM steht. Vereinfacht ausgedrückt verwendet diese Speicherart, zum Speichern von Bits, keinen Kondensator im herkömmlichen Sinn, sondern einen Ferroelektrischen Kondensator [5, S. 1].

Funktionsweise

Als Dielektrikum kommt ein polarisierbares Material zum Einsatz (Bariumtitanat, $BaTiO_3$), welches eine Perowskit-Artige Kristallstruktur besitzt. Eine Polarisierung erfolgt durch Anlegen eines elektrischen Feldes. Der Polarisierungsgrad bleibt auch nach abschalten des E-Feldes erhalten, hierdurch ergibt sich ein Non-Volatile Charakter der Speicherzelle. Um den Zustand zu ändern, muss das E-Feld in umgekehrter Richtung angelegt werden. Polarisierung ist ein sehr schneller Vorgang, weshalb ein FRAM einem normalen SRAM in Bezug auf die Schreib- und Lesegeschwindigkeit in nichts nach steht [6, S. 2].

Durch Anlegen eines elektrischen Feldes wechselt das eingeschlossene Titan-Ion seine Position (siehe Abbildung 4). Je nach Lage des Titan-Ions wird ein anderer Logischer Zustand abgebildet.

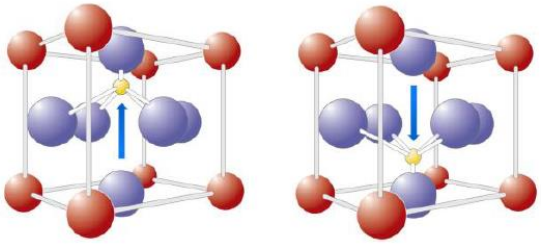

Abbildung 4 PZT Kristall bei Zustand "0" und "1"

Vergleich zu EEPROM

FRAM's werden häufig als Ersatz zu herkömmlichen EEPROMS eingesetzt, da Ferroelectric RAM's deutliche Vorteile aufweisen. (Vgl. hierzu [7])

	Geschwindigkeit	Energieaufnahme	Zellen Lebensdauer
EEPROM	---	---	10^6
FRAM	+++	+++	Bis zu 10^{15}

Tabelle 2 Vergleich von EEPROM und FRAM

Ein FRAM besitzt eine Synchrone Lese- und Schreib Performance mit SRAM Geschwindigkeit (100ns), im Gegensatz zum EEPROM, welches bis zu 5ms für ein Write benötigt.
Im Bezug auf die Energieaufnahme benötigt ein FRAM ca. 3000x weniger Energie als ein EEPROM. Besonders beeindruckende Werte kann ein FRAM im Bereich Lebensdauer aufweisen. Beim FRAM gibt der Hersteller Cypress die Lebensdauer mit 10^{15} Schreib Zyklen an [3].

Beispiel:
Ein Datenlogger, welcher jede Sekunde Daten schreibt (immer auf dieselbe Speicherzelle), bringt eine EEPROM Zelle nach 2 Jahren an ihre Lebensdauergrenze. Eine FRAM Zelle hält unter optimal Bedingungen theoretisch 190 Millionen Jahre aus [7].

Umsetzung

In diesem Kapitel soll es um die Umsetzung des SPI und FRAM Controllers in System Verilog gehen. Die Entscheidung, einen IP-Core unabhängigen SPI Treiber zu benutzen, begründet sich in der hierdurch vielseitigeren Verwendung des Codes auf unterschiedlichsten FPGA‘s verschiedener Firmen. Der Code wurde in EDA geschrieben und Simuliert, ein abschließender Test mit Radiant war ebenso Erfolgreich.

SPI

Bei dem SPI Teil des Controllers wurde auf ein bereits vorhandenen Verilog Code zurückgegriffen. Der Code von Nandland[8] ist sehr umfangreich und ist nicht auf einen IP Core angewiesen. Somit ist der Code für jeden FPGA anwendbar.
Im folgendem soll kurz auf die Code Bestandteile eingegangen werden.

Der SPI Code besteht aus zwei Teilen. Das SPI_Master Modul, in iSPI.sv, bedient die Low Level Signal Behandlung, wie Takt Generierung und das Steuern der MISO und MOSI Leitungen. Das SPI_Master_With_Single_CS Modul, in SPI.sv, erweitert das SPI_Master Modul um die Verwendung eines CS Signals.

Initialisierung

Um das gesamte SPI Modul zu verwenden, muss das SPI_Master_With_Single_CS instanziiert werden.

Das Modul benötigt zu Anfang bestimmte Parameter.

```
52  parameter SPI_MODE = 0;          // CPOL = 0, CPHA = 0
53  parameter CLKS_PER_HALF_BIT = 2; // 25MHz
54  parameter MAX_BYTES_PER_CS = 5;  // 5 bytes max per chip select cycle
55  parameter CS_INACTIVE_CLKS = 1;  // Adds delay (1clk) between cycles
```

Abbildung 5 SPI Parameter (design.sv)

SPI_Clk wird bestimmt durch:

$$f_{SPI} = \frac{f_{in}}{CLK\_PER\_HALF\_BIT \times 2} = \frac{100MHz}{2 \times 2} = 25MHz$$

Weiterhin werden zusätzliche Signale definiert

```
58  logic [7:0] r_Master_TX_Byte = 0;
59  logic r_Master_TX_DV = 1'b0;
60  logic w_Master_TX_Ready;
61  logic w_Master_RX_DV;
62  logic [7:0] w_Master_RX_Byte;
63  logic [$clog2(MAX_BYTES_PER_CS+1)-1:0] w_Master_RX_Count, r_Master_TX_Count = 3'h1; //Standard 1 Byte pro CS Cycle
```

Abbildung 6 Zusätzliche SPI Signale (design.sv)

r_Master_TxByte enthält das zusendende Byte, r_Master_TX_DV signalisiert, dass die Tx Daten valide sind und gesendet werden können. w_Master_Tx_Ready zeigt an, dass das SPI Modul bereit ist Daten zu senden. w_Master_Rx_DV bestätigt, ob Daten erfolgreich gelesen wurden und in w_Master_RX_Bate zur Verfügung stehen.
w_Master_T/RX_Count gibt an, wieviele Bytes pro CS Zyklus übertragen bzw. empfangen werden sollen. Dies ist besonders wichtig bei langen Übertragungen, bei welchen CS konstant auf Low Pegel gehalten werden muss.

Mit den vorher festgelegten Parametern und Signalen wird dann das SPI Modul instanziiert.

```
66  SPI_Master_With_Single_CS
67  #(.SPI_MODE(SPI_MODE), //SPI Mode 0-3
68    .CLKS_PER_HALF_BIT(CLKS_PER_HALF_BIT), //sets Frequency of SPI_CLK
69    .MAX_BYTES_PER_CS(MAX_BYTES_PER_CS), //Maximum Bytes per CS Cycle
70    .CS_INACTIVE_CLKS(CS_INACTIVE_CLKS) //Amount of Time holding CS Low befor next command
71   ) SPI
72  (
73   // Control/Data Signals,
74    .i_Rst_L(i_nreset),     // FPGA Reset
75    .i_Clk(i_clk),         // FPGA Clock
76
77   // TX (MOSI) Signals
78   .i_TX_Count(r_Master_TX_Count),    // Number of bytes per CS
79   .i_TX_Byte(r_Master_TX_Byte),      // Byte to transmit on MOSI
80   .i_TX_DV(r_Master_TX_DV),          // Data Valid Pulse with i_TX_Byte
81   .o_TX_Ready(w_Master_TX_Ready),    // Transmit Ready for Byte
82
83   // RX (MISO) Signals
84   .o_RX_Count(w_Master_RX_Count), // Index of RX'd byte
85   .o_RX_DV(w_Master_RX_DV),       // Data Valid pulse (1 clock cycle)
86   .o_RX_Byte(w_Master_RX_Byte),   // Byte received on MISO
87
88   // SPI Interface
89   .o_SPI_Clk(o_SPI_Clk),
90   .i_SPI_MISO(i_SPI_MISO),
91   .o_SPI_MOSI(o_SPI_MOSI),
92   .o_SPI_CS_n(o_SPI_CS_n)
93   );
```

Abbildung 7 SPI Instanziirung (design.sv)

Des Weiteren werden auch die eigentlichen SPI Signale (Clk, MISO, MOSI und nCS) definiert, diese können dann echten I/O Pins des FPGAS zugewiesen werden.

Senden und Empfangen von einem Byte

Zum Senden und Empfangen gibt es eine Task *SPI_SendByte.*

```
98   task SPI_SendByte(input [7:0] data);
99     @(posedge i_clk);
100    r_Master_TX_Byte <= data;
101    r_Master_TX_DV   <= 1'b1;
102    @(posedge i_clk);
103    r_Master_TX_DV <= 1'b0;
104    @(posedge i_clk);
105    @(posedge w_Master_TX_Ready);
106  endtask //end SPI_SendByte
```

Abbildung 8 Task SPI_SendByte

Die Task hat einen Übergabeparameter und zwar das Byte, welches gesendet werden soll. SPI ist eine Voll-Duplex Übertagung, somit werden Daten vom Slave geschickt, während Daten vom Master gesendet werden.
Zum Lesen eines Bytes vom Slave wird die Task SendByte mit einem Dummy-Byte aufgerufen. Nach dem Senden des Bytes befindet sich das Empfangene Byte in w_Master_Rx_Byte. (siehe auch Kapitel FRAM Controller)

FRAM Controller

Der FRAM Controller ermöglicht das Lesen und Schreiben der einzelnen Speicherzellen, das Lesen des Statusregisters und die Möglichkeit den FRAM in den sogenannten Hibernate Mode Zustand zu versetzen.

Initialisierung

Das FRAM Modul benötigt keine Parameter und nur wenige Signale.

```
module FRAM(
  input i_clk, //Module (Module CLock = SPI Clock)
  input i_nreset,

  input logic [19:0] i_adr, //Speicherzellen Adresse in FRAM
  input logic [7:0] i_data, //daten zum schreiben
  output logic [7:0] o_data, //Daten gelesen

  input logic i_rw, //Read = 1, Write = 0
  input logic i_status, //Wenn 1 Status Register wird gelesen
  input logic i_hbn, //Wenn 1 Fram begibt sich in Hibernat Mode
  input logic i_cready,

  // SPI Interface
  output o_SPI_Clk,
  input  i_SPI_MISO,
  output o_SPI_MOSI,
  output o_SPI_CS_n

);
```

Abbildung 9 FRAM Initialisierungs Liste

Für die Adresse der einzelnen RAM-Zellen wird eine 20Bit Breite Adresse benötigt, um alle Zellen anzusprechen, da die interne Organisation 1024*8bit vorsieht.
Daten werden immer als Byte übergeben.

Das i_rw Signal signalisiert, ob eine Speicherzelle gelesen (1) oder geschrieben (0) werden soll. Setzen des i_status ermöglicht das Lesen des Status Registers. Mit i_hbn wird der FRAM direkt in den Hibernation Mode gebracht. i_cready signalisiert dem Controller, dass alle Signale richtig anliegen, valide sind und gesendet werden können.

Durch den Aufbau des FRAM Controllers und des SPI Treibers sind diese nicht an bestimmte Pins des FPGAs angewiesen, hierdurch ist es möglich auch mehrere FRAM-Controller und SPI Interfaces zu nutzen. Daher wird dem FRAM Modul auch die Signale für das SPI mitgeteilt.

Logik

Der Controller besitzt einen always Block und mehrere Tasks, welche bestimmte Operationen des FRAMS abbilden.

```
always @(posedge i_clk or negedge i_nreset) begin

  state[0] = i_cready;
  state[1] = i_hbn;
  state[2] = i_status;
  state[3] = i_rw;

  if(~i_nreset) begin //Modul Reset
    o_data <=  8'h00;
  end //end if

  if(w_Master_TX_Ready) begin
      case(state) inside
        4'b??11: FRAM_Hibernation();
        4'b?101: FRAM_Read_Status(o_data);
        4'b1001: FRAM_Read(i_adr, o_data);
        4'b0001: FRAM_Write(i_adr, i_data);

        default:;
      endcase //endcase
  end //endif

end //end always
```

Abbildung 10 Haupt- Always Block des Controllers

Der always Block entscheidet anhand der angelegten Steuersignale, welche Aktion/Task ausgeführt werden soll.

Die einzelnen FRAM-Tasks sind im Kapitel Ergebnis aufgeführt.

Testbench

In der Testbench wurde jeder einzelne Test in einen Task gekapselt.
Für die Tests wurde die MISO und MOSI Leitung miteinander verbunden, um auch das Lesen simulieren zu können.
Das letzte gesendete Byte wird dadurch wieder in den Empfangs Buffer geschrieben.

```
always @ (posedge starttesting or posedge FRAM_busy) begin

  repeat(10) @(posedge clk);
  if(test_running == 1'h0 & FRAM_busy == 1'h1) begin

      if(test == TESTS_cnt+1) begin
        test_running <= 1'h0;
        $display("Tests Finished");
        $finish;
      end

      case(test) inside
        3'b000: begin Test1(); test <= test + 1'h1; end
        3'b001: begin Test2(); test <= test + 1'h1; end
        3'b010: begin Test3(); test <= test + 1'h1; end
        3'b011: begin Test4(); test <= test + 1'h1; end

      endcase
  end // endif
end // end always
```

Abbildung 11 always block zum durchschalten der Tests Task's

Durch den always Block werden sämtliche Tests automatisch nacheinander ausgeführt. Nach dem Abschluss des letzten Tests, wird die Simulation mit $finish beendet.

Die Testbench wurde mit einer Zeiteinheit von 1ns simuliert.
Ein gesamter Simulationsdurchlauf aller Tests beträgt hiermit 7µs.

Die einzelnen Tests-Tasks mit dem Ergebnis der Simulation folgt im Kapitel Ergebnis.

Ergebnis

Die Simulation kann unter diesem Link ausgeführt werden.
https://www.edaplayground.com/x/8TH5
Der Link führt direkt auf das Projekt FRAM Controller.

FRAM-Enter Hibernation Mode

Der Test1 versetzt den FRAM Chip in den Hibernation-Mode (OP-Code B9h)

```
task Test1();
  test_running <= 1'h1;
  $display("DEBUG: %0tns: Test_1_Hibernation",$realtime);
  FRAM_hbn <= 1'h1; //Enter Hibernation
  FRAM_go <= 1'h1;
  #10;
  FRAM_hbn <= 1'h0; //Reset Hibernation Flag
  FRAM_go <= 1'h0;
  $display("DEBUG: %0tns: Test_1_Hibernation__-END",$realtime);
  test_running <= 1'h0;
endtask
```

Abbildung 15 Testfall in Testbench

```
task FRAM_Hibernation(); //vgl. Fig22
  r_Master_TX_Count <= 3'h1; //1 Byte Transaction
  SPI_SendByte(HBN);
endtask //FRAM_Hibernation
```

Abbildung 14 FRAM Task für Hibernation Mode

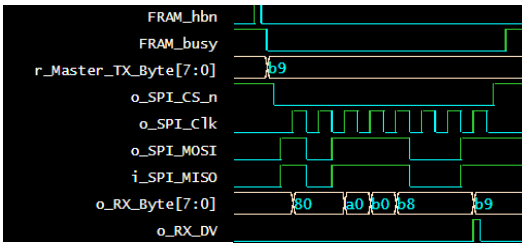

Abbildung 12 Simulation für Test1

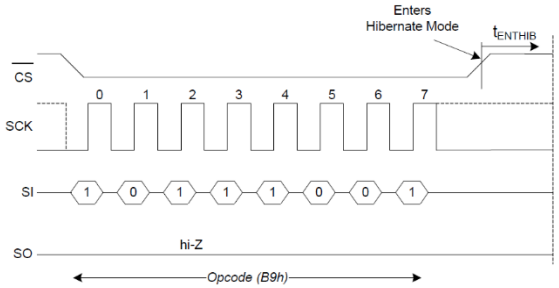

Abbildung 13 Referenz für Hibernationmode

Wie aus Abbildung 12 ersichtlich, wird der OP-Code b9h über MOSI erfolgreich übertragen. Ein Aufwecken aus diesem Modus ist nicht erforderlich, dies geschieht automatisch bei einer neuen Übertragung (fallende Flanke von SPI_CS)

FRAM-Read Status Register

Der Test2 simuliert das Lesen des Statusregisters (OP-Code 05h)

```
task Test2();
  test_running <= 1'h1;
  $display("DEBUG: %0tns: Test_2_ReadStatus",$realtime);
  FRAM_RSTATUS <= 1'h1; //Read Status
  FRAM_go <= 1'h1; //Go
  #10;
  FRAM_RSTATUS <= 1'h0; //Read Status
  FRAM_go <= 1'h0; //reset Go
  $display("DEBUG: %0tns: Test_2_ReadStatus__-END",$realtime);
  test_running <= 1'h0;
endtask
```

Abbildung 19 Testfall in Testbench

```
task FRAM_Read_Status(output [7:0] data); //vgl. Fig9
  r_Master_TX_Count <= 3'h2; //2 Byte Transaction
  SPI_SendByte(RDSR); //OpCode
  SPI_SendByte(8'hFD); //Dummy Bits, read byte in w_Master_RX_Byte
  data = w_Master_RX_Byte;
endtask //FRAM_Read_Status

task FRAM_Hibernation(); //vgl. Fig22
  r_Master_TX_Count <= 3'h1; //1 Byte Transaction
  SPI_SendByte(HBN);
endtask //FRAM_Hibernation
```

Abbildung 16 FRAM Task für Status Read

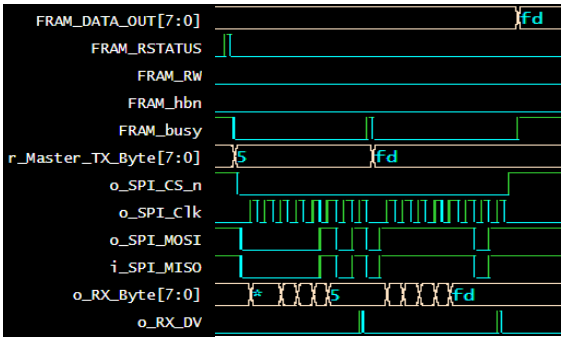

Abbildung 17 Simulation für Read Status Register

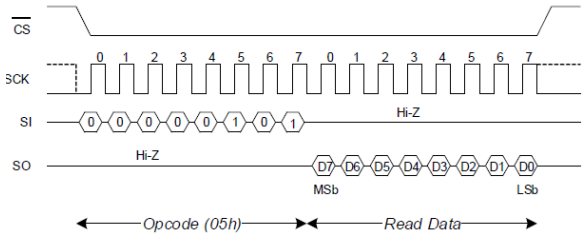

Abbildung 18 Referenz für Read Status Register

Nach dem OP-Code müssen Dummy-Bits geschickt werden, um die Übertragung aufrecht zu erhalten. Die Dummy-Daten stellen erscheinen nach der Aktion im FRAM_DATA_OUT Register. Zum Testen wurde als Dummy_Byte FDh gewählt. Dies ist für die Operation MEM Write wichtig.

FRAM-MEM Write

Der Test3 simuliert das Schreiben von Daten in das FRAM.
Geschrieben werden soll an die Adresse 8FFF1h das Byte AAh.

```
task Test3();
  test_running <= 1'h1;
  $display("DEBUG: %0tns: Test_3_FRAM_WRITE",$realtime);
  FRAM_Adr <= 20'h8FFF1; //Load 8FFF1 as adress
  FRAM_DATA_IN <= 8'hAA; //Load AA as Data to Write into FRAM
  FRAM_RW <= 1'h0; //Write Operation
  FRAM_go <= 1'h1; //Go
  #10;
  FRAM_go <= 1'h0; //resetGo
  $display("DEBUG: %0tns: Test_3_FRAM_WRITE__-END",$realtime);
  test_running <= 1'h0;
endtask
```

Abbildung 20 Test für Daten Schreiben

Vor dem Starten eines Schreib Prozesses muss das Write-Enable Bit gesetzt werden, dies geschieht durch das Senden des OP-Codes 06h. Die übertragende Adresse muss 24bit breit sein. Für 1024 Zellen werden aber nur 20bit effektiv benötigt.
Laut Datenblatt werden die ersten 4bit der Adresse deshalb verworfen.

```
task FRAM_Write(input [19:0] adr, input [7:0] data); //vgl. Fig.11

  logic [7:0] value;
  value <= 8'h0;

  //Set Write Enable
  r_Master_TX_Count <= 3'b1; //1Byte Transaction
  SPI_SendByte(WREN);

  //Write to fram
  r_Master_TX_Count <= 3'h5; //5 Byte Transaction
  SPI_SendByte(WRITE); //OPCode
  SPI_SendByte({4'hF,adr[19:16]}); //Adress [23-16]
  SPI_SendByte(adr[15:8]); //Adress [15-8]
  SPI_SendByte(adr[7:0]); //Adress [7-0]
  SPI_SendByte(data); //Data [7:0]

  //Reset Write Disable and Verify
  do begin
    r_Master_TX_Count <= 3'b1; //1Byte Transaction
    SPI_SendByte(WRDI); //Set Write Disable

    FRAM_Read_Status(value); //Lese Status Register
  end while(((value & 8'h2) >> 1) != 0);


endtask //end FRAM_Write
```

Abbildung 21 FRAM Task zum Schreiben

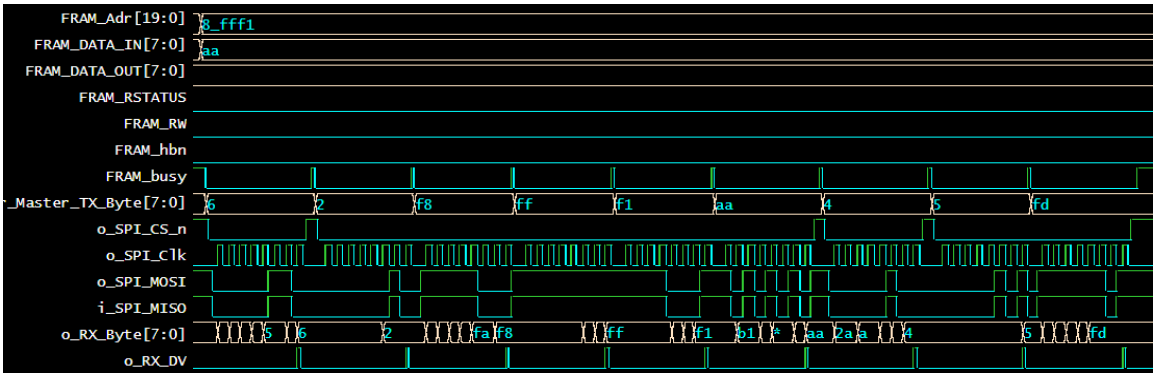

Abbildung 22 Simulation für FRAM Read an Adresse 8fff1h mit Daten AAh

Nachdem eine Write Operation ausgeführt wurde, muss das Write Enable Bit wieder zurückgesetzt werden (OP-Code 04h). Laut Datenblatt muss danach das Statusregister gelesen werden (OP-Code 05h), ob dieses Bit auch wirklich wieder auf „0“ steht, dies geschieht im do Block. Um eine Endlosschleife für den Test zu unterbinden, liefert Read Status immer eine „0“ an der 1 Bit-Position (FDh). (siehe FRAM Read Status)

FRAM-MEM Read

Der Test4 simuliert das Lesen von Daten aus dem FRAM.

```
task Test4();
  test_running <= 1'h1;
  $display("DEBUG: %0tns: Test_4_FRAM_READ",$realtime);
  FRAM_Adr <= 20'h8FFF1; //Load 8FFF1 as adress
  FRAM_RW <= 1'h1; //Read
  FRAM_go <= 1'h1; //Go
  #10;
  FRAM_go <= 1'h0; //resetGo
  FRAM_RW <= 1'h0; //Read
  $display("DEBUG: %0tns: Test_4_FRAM_READ__-END",$realtime);
  test_running <= 1'h0;
endtask
```

Abbildung 24 Test zum Lesen von Daten

```
task FRAM_Read(input [19:0] adr, output [7:0] data); //vgl. Fig12
  r_Master_TX_Count <= 3'h5; //5 Byte Transaction
  SPI_SendByte(READ); //Opcode
  SPI_SendByte({4'hF,adr[19:16]}); //Adress [23-16]
  SPI_SendByte(adr[15:8]); //Adress [15-8]
  SPI_SendByte(adr[7:0]); //Adress [7-0]

  SPI_SendByte(8'hAA); //Dummy Bits, read byte in w_Master_RX_Byte
  data = w_Master_RX_Byte;

endtask //end FRAM_READ
```

Abbildung 23 FRAM Task zum Lesen

Wie schon beim Lesen des Statusregisters muss wieder ein Dummy- Byte (hier: AAh) mit übertragen werden, damit die eigentlichen Daten empfangen werden können. Durch den Aufbau der Testumgebung gilt wieder: Gesendetes Dummy-Byte = Empfangenes Byte.
Das Adressfeld ist wieder gleich aufgebaut, wie beim Schreiben von Daten.

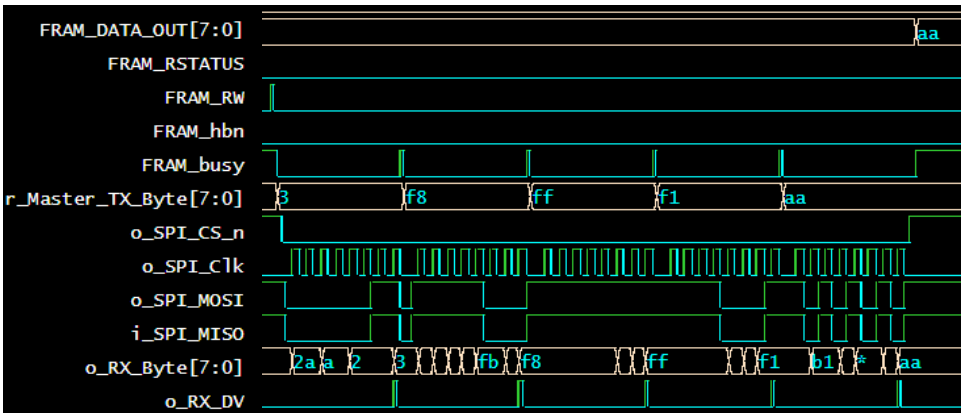

Abbildung 25 Simulation von Data Read

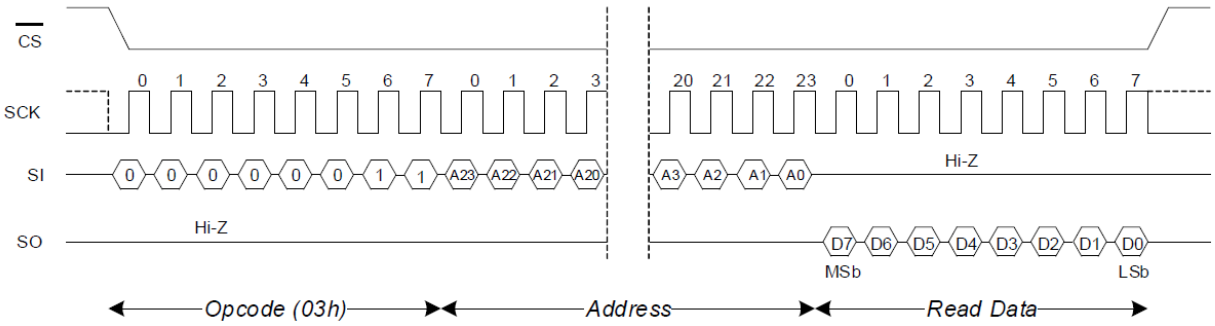

Abbildung 26 Referenz zum Daten Lesen

Literatur

[1] „Introduction to SPI Interface | Analog Devices“. https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html (zugegriffen Juni 06, 2021).
[2] „Serial Peripheral Interface“, *Wikipedia*. Mai 30, 2021. Zugegriffen: Juni 06, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=Serial_Peripheral_Interface&oldid=212519603
[3] „CY15B108QN/CY15V108QN, Excelon™ LP 8-Mbit (1024K × 8) Serial (SPI) F-RAM“, Nr. 002, S. 30.
[4] „Serial Peripheral Interface – Mikrocontroller.net“. https://www.mikrocontroller.net/articles/Serial_Peripheral_Interface (zugegriffen Juni 06, 2021).
[5] „F-RAM Technology Brief“, S. 6, 2016.
[6] „fram-guide-book.pdf“. Zugegriffen: Juni 02, 2021. [Online]. Verfügbar unter: https://www.fujitsu.com/downloads/MICRO/fme/fram/fram-guide-book.pdf
[7] „Cypress f-RAM technology Video“. https://www.cypress.com/brightcove/colorbox/1694399810001/4363835459001/BJ1olC4Ke/Cypress%27%20F-RAM%20Technology?height=490px (zugegriffen Juni 02, 2021).
[8] nandland, *nandland/spi-master*. 2021. Zugegriffen: Juni 06, 2021. [Online]. Verfügbar unter: https://github.com/nandland/spi-master

Bildverzeichnis