TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Technische Hochschule Nürnberg Georg Simon Ohm
Fakultät Elektrotechnik Feinwerktechnik Informationstechnik

Studienarbeit

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

Thema

Name: Kilian Krause
Matrikelnummer: 3266526

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

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

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Hinweis: Diese Erklärung ist in alle Exemplare der Prüfungsarbeit fest einzubinden. (Keine Spiralbindung)

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Krause Vorname: Kilian Matrikel-Nr.: 3266526

Fakultät: EFI

Studiengang: B-EI

Semester: 6. Semester

Titel der Prüfungsarbeit: SPI FRAM Modell

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Herzogenaurach, 08.06.21, Kilian Krause

Ort, Datum, Unterschrift Studierende/Studierender

Inhaltsverzeichnis:

Einleitung

Die Studienarbeit handelt von einem SPI-FRAM-Modell, welche durch einen SPI-Controller angesteuert wird. Die korrekte Funktion ist durch Assertions sichergestellt, das Modell bildet das Verhalten eines SPI-FRAM Bausteins nach und implementiert die Befehle Read und Write Memory, Read Status Register, einen Ruhezustand sowie die SPI Modes 0 und 3.

Kurzübersicht zu SPI und FRAM

SPI

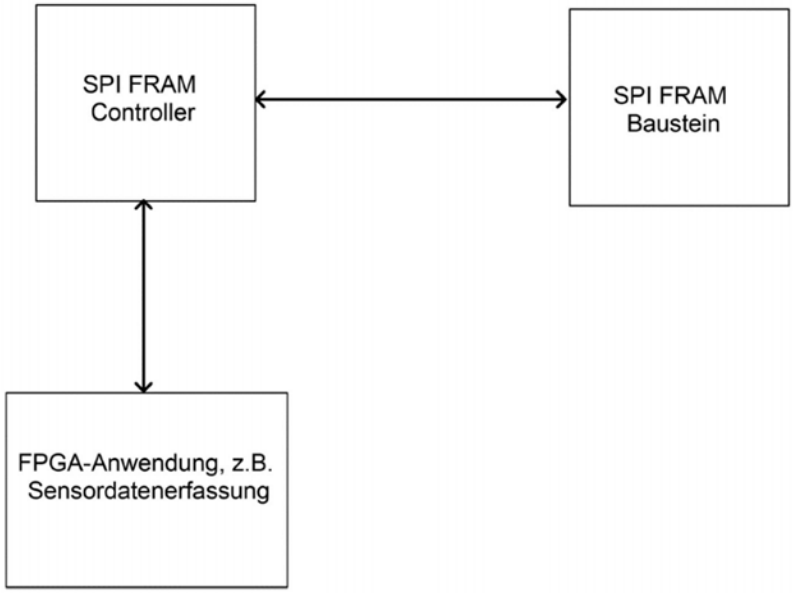
SPI, kurz für Serial Peripheral Interface, ist eine Schnittstelle, die ihre Verwendung in eingebetteten Systemen mit Mikrokontrollern und/oder FPGAs findet. SPI ist ein freies Protokoll, welches nicht mit Lizenzgebühren belegt ist [1, S. 5]. SPI funktioniert nach dem Master-Slave-Prinzip. Der Slave, welcher in diesem Fall z.B. das FRAM Modul ist, und der Master, der SPI-Controller, werden mit 4 Leitungen verbunden. Diese heißen MISO, MOSI, SCK und $\overline{CS}$ [1, S. 6]. MISO heißt Master in Slave out und stellt die Datenleitung dar, mit der der Master Daten empfängt, die der Slave gesendet hat. Das Gegenstück dazu ist MOSI. SCK ist der Takt, welcher während einer Übertragung der Daten aktiv ist und die Kommunikation zwischen Master und Slave synchronisiert. Der SPI-Mode entscheidet, ob ein Teilnehmer Daten bei einer steigenden oder fallenden Flanke sendet oder empfängt und ist frei einstellbar. Außerdem bestimmt der Mode, wie sich SCK im Leerlauf verhält. $\overline{CS}$ ist kurz für Chip select, low aktiv und muss bei einer Übertragung zwischen Master und Slave aktiv sein. Bei mehreren Slaves ist so die Wahl des Slaves möglich, indem der Master nur einen Slave durch $\overline{CS}$ aktiv schaltet. SPI überträgt Daten des Weiteren vollduplex, beide Teilnehmer senden und empfangen also gleichzeitig [1, S. 7].

FRAM

FRAM ist eine Speichertechnologie und steht für ferroelectric random access memory. Der Speicher ist nicht flüchtig, was bedeutet, dass er seine Daten nicht verliert, sobald er stromlos ist. Seine Geschwindigkeit ist ca. um das tausendfache höher im Vergleich zu anderen nicht flüchtigen Speichern wie EEPROM und Flash-Speicher im Allgemeinen [2, S. 1]. Cypress gibt für das Modul CY15X108QN, welches als Referenz für das Modell dient, einen Datenerhalt für 151 Jahre an [3, S. 1]. In Kombination mit der großen Anzahl an Lese/Schreibzyklen von 10^{15} Schreibzyklen ergeben sich neue Nutzungsmöglichkeiten, welche mit einem EEPROM, welcher zwar 200 Jahre Datenerhalt, aber nur 10^5 Schreibzyklen garantiert, nicht umsetzbar sind [4, S. 1]. Der FRAM unterstützt lediglich SPI Mode 0 und 3 [5, S. 8], [3, S. 7].

Systembeschreibung mit Anwendungsbeispiel

Ein System mit einem FRAM könnte beispielsweise folgendermaßen Aussehen:



Der SPI FRAM übernimmt hier das Speichern der Daten. Der FRAM Controller ist verantwortlich für die Kommunikation zwischen der Anwendung mit dem Speicher.

Beschreibung der Register

Status-Register

Das 8-Bit große Status Register enthält Bits, welche Einfluss auf das Schreibverhalten haben. Bit 0, das für “Read or Write in progress” steht, ist bei einem FRAM immer 0. Eine Ausnahme gibt es für den Hibernate Mode, Bit 0 ist während dem Aufwachen aus dem Hibernate Modus 1 um zu signalisieren, dass der FRAM noch nicht bereit zum schreiben ist. [3, S. 10]. Bits 2, 3 und 7 kümmern sich um Schreibschutz, welcher in diesem Fall nicht implementiert ist. So bleibt aus dem Status Register nur noch Bit 1 übrig. Bit 1 ist wichtig und muss bei jedem Schreibvorgang in den Speicher gesetzt sein.

Beschreibung der Operationen

Die folgenden Operationen sind durch das Modell realisiert.

Read Memory Operation

Die Read Operation liest aus dem Speicher ab einer bestimmten Adresse aus, bis $\overline{CS}$ aktiv wird. Der Lesezugriff besteht aus drei Teilen, dem Opcode, der Speicheradresse und den gesendeten Daten. Bis $\overline{CS}$ deaktiviert ist, schreibt der FRAM, beginnend ab der Adresse, den Speicherinhalt in die SPI [3, S. 13].

Write Memory Operation

Die Write Operation dient zum Schreiben in den FRAM über SPI. Der Schreibzugriff beginnt mit dem opcode-Befehl WREN, welcher anschließend bei inaktivem chip select das Status Register Bit 1 auf den Wert eins setzt, um Schreibzugriffe zu erlauben. Anschließend beginnt der eigentliche Schreibzugriff mit erneut aktiviertem nCS. Zuerst sendet der Controller den opcode-

Befehl WRITE, worauf eine 20-Bit große Adresse folgt ist. Die nachfolgenden Bytes, die der Controller sendet, schreibt der FRAM an die gesendete Adresse und imkrementiert diese nach jedem Byte, bis $\overline{CS}$ inaktiv ist [3, S. 12].

Read Status Register Operation

Mithilfe dieser Operation lässt sich das Status Register des FRAMs durch den Controller auslesen. Mit dem opcode-Befehl RDSR startet die Übertragung. Der FRAM sendet darauf ein Byte, welches das gesamte Statusregister enthält [3, S. 12].

Hibernate Mode

Der Hibernate Mode stellt den energiesparendsten Modus des FRAMs dar. Der opcode-Befehl HBN versetzt den FRAM nach einer kurzen Zeit in den Hibernate Mode. Während der Speicher sich im Hibernate Mode befindet, kann er nicht auf SCK oder SI reagieren. Der Hibernate Mode ist beendet, wenn der Controller das CS-Signal erneut aktiv setzt [3, S. 18].

Implementierung des FRAM-Moduls

Im Folgenden findet sich die Beschreibung des Codes. Das Modul “SPI-FRAM-Module” stellt die Implementierung des FRAM-Bausteins dar, das Modul “SPI_FRAM_tb” testet diesen so weit wie möglich auf korrekte Funktion. Die Kommentare zur Initialisierung der Variablen sind detailliert, die wichtigsten sind jedoch bei der nachfolgenden Beschreibung der einzelnen Funktionen genauer erklärt. Für die Grundlage des SPI-Slaves ist fremder Code verwendet worden, welcher überarbeitet und verändert ist [6].

SPI-FRAM-Module

Dieses Modul implementiert den SPI-Slave und die Verarbeitung der eingehenden Daten, um das Verhalten eines FRAM-Moduls nachzubilden. Zuerst folgt die Betrachtung des SPI-Slaves.

FRAM-Speicherinhalt initialisieren

```
39      $readmemh("memory.txt", mem_data); //initializes the memory with the contents of memory.txt
```

Der Speicherinhalt für das Modul ist auf 1024 * 8 Bits festgelegt. Die Größe ist veränderbar. Der Speicher muss initialisiert sein, sobald ein Zugriff auf ihn erfolgt. Besonders ein Lesezugriff endet sonst in einem undefinierten Ergebnis. Dies geschieht mithilfe der Funktion $readmemh. Als ersten Parameter nimmt diese eine Textdatei, welche hexadezimale Werte enthält, die getrennt nach der Packed Size des zu initialisierenden Arrays sind. In diesem Fall ist die Packed Size 8 Bit, also ein Byte. Die Textdatei enthält 1024 8-Bit-Blöcke. Der zweite Parameter ist das zu initialisierendes Array, welches laut Definition zweidimensional sein muss [7].

SPI-Slave Daten empfangen

```
42  //receive incoming Bits and organize them bytewise
43  always @(posedge SCK) begin
44    if (bitcnt_rcv == 3'b111) begin
45      byte_count <= byte_count + 4'b0001;
46    end
47    bitcnt_rcv <= bitcnt_rcv + 3'b001;
48    byte_data_received <= {byte_data_received[6:0], SI};
49
50    //when opcode WRÍTE is executed, the incoming bytes are written to memory
51    if (write_to_memory == 1 && nCS == 0) begin
52      mem_data[addr][bitcnt_mem_write+3'b001] = byte_data_received[0];
53
54      if(bitcnt_mem_write == 3'b000) begin
55        addr <= addr + 1;
56        bitcnt_mem_write <= 3'b111;
57      end
58      bitcnt_mem_write <= bitcnt_mem_write - 1;
59    end
60  end
61  always @(posedge SCK) byte_received <= (nCS == 0) && (bitcnt_rcv==3'b111);
```

Der vorangehende Code-Block beschreibt das Einlesen von Daten durch den serial input SI. Das hochwertigste Bit wird zuerst übertragen. Bei jeder steigenden Flanke von SCK, dem Takt, ordnet der SPI-Slave das erhaltene Bit (SI) einem Byte (byte_data_received) zu und inkrementiert den Zähler bitcnt_rcv. Wenn dieser überläuft, ist das Byte vollständig und die Variable byte-count inkrementiert um eins. Mit diesen beiden Variablen ist so jedes neue Bit konkret einem Byte zugeordnet, um es in Form des Shift-Registers byte_data_received einfach weiterzuverarbeiten. Byte_data_received shiftet die erhaltenen Bits von rechts nach links, da SPI das hochwertigste Bit zuerst überträgt.

Der innere Block ab Z. 51 zeigt das Vorgehen bei einem Schreibvorgang, das heißt wenn write_to_memory == 1 gesetzt ist. Wie und wann die Variable gesetzt ist, ist der späteren Codebeschreibung zur State-Machine zu entnehmen. Der FRAM bekommt die Daten vom Controller-Modul, welches vorher eine Adresse gesendet hat. Der FRAM speichert die Bits in seinem Speicher ab dieser Adresse, bis das Signal nCS, Chip select low active, vom Controller gesetzt wird. Bis zu diesem Zeitpunkt schreibt der SPI-Slave die Daten des SI in den Speicherbereich mit der übergebenen Adresse. Die Adresse inkrementiert nach jedem ganzen Byte um 1. Für jedes erhaltene Bit dekrementiert die Variable bitcnt_mem_write, um das Byte in der richtigen Reihenfolge in den Speicher zu schreiben.

SPI-Slave Daten senden

```
63  //TRANSMISSION
64  //Read out memory and write to SPI, starts at addr
65  always @(negedge SCK) begin
66    if(send_data == 1 && nCS == 0)
67      begin
68        byte_data_sent <= {byte_data_sent[6:0], mem_data[addr][bitcnt_snd]};
69        bitcnt_snd <= bitcnt_snd - 1;
70        if (bitcnt_snd == 3'b000) begin
71          addr <= addr + 1;
72          bitcnt_snd <= 3'b111;
73        end
74      end
75    //write status register to SO when opcode RDSR is sent
76    else if (opcode == 8'h05 && nCS == 0 && i > 0) begin
77      byte_data_sent <= {byte_data_sent[6:0], stat_reg[i-1]};
78      i = i - 1;
79    end
80  end
81  assign SO = byte_data_sent[0];  // MSB of the transmission is the lsb of byte_data_sent
```

Der SPI-Slave sendet Daten, um den Speicher oder das Statusregister durch einen Controller auslesen zu können. Hierfür muss zuerst das Bit send_data gesetzt sein, welches bei einer solchen Übertragung durch die State machine gesetzt ist.
Der SPI-Slave schreibt am Anfang den Speicherinhalt Bit für Bit in das Shift-Register byte_data_sent. Bit 0 des Shift Registers ist durch den assign-Befehl in Z. 81 mit dem SO verbunden. Der Zustand von SO ändert sich, sobald eine Änderung von byte_data_sent[0] stattfindet. Jedes ausgelesene Bit befindet sich so direkt auf der Datenleitung SO.

Der zweite und letzte Fall, der Daten in die SPI schreiben muss, ist die Übertragung des Status-Registers. Dies geschieht mit der Überprüfung auf den korrekten opcode RDSR, 8'h05, und aktivem nCS. Das Status-Register ist ein Bit groß, hier zählt der Zähler i herunter, um das MSB des Status-Registers zuerst zu senden. In Byte_data_sent steht am Ende der Übertragung das Status Register. Das Verhalten von SO ist identisch zu dem beim Auslesen des Speichers, das aktuelle Bit ist direkt SO zugewiesen, sobald sich byte_data_sent durch ein weiteres Bit aus dem Status-Register ändert.

Verhalten bei Beenden einer Nachricht durch $\overline{CS}$

```
//the following block resets counters when a message has finished
  always @ (posedge nCS) begin
    if (opcode == 8'h06) begin  //When WLEN opcode is executed, nCS needs to be reset.
                                //Since the message is not finished, no counters should be reset when executing WLEN
    end
    else if (opcode == 8'hb9 && nCS == 1) hibernate = 1; //When hibernation opcode 8'hb9 is sent, the device goes into hibernation
    else begin
      byte_count = 8'h00;
      bitcnt_rcv = 3'b000;
      bitcnt_snd = 3'b111;
      bitcnt_mem_write = 3'b111;
      byte_data_received = 8'h00;
    end
    send_data = 0;      //disables sending data
    write_to_memory = 0; // disables writing to memory
    stat_reg[1] = 0; //reset WEL when writing to memory has finished
  end

  //reset hibernate
  always @ (negedge nCS) hibernate = 0;
//when a byte is received the FRAM-model reacts dependent on the number of bytes received in the current nCS low state, i. e. in one message.
```

Eine Nachricht ist durch das Deaktivieren von nCS durch den Controller beendet. Beim erneuten Aktivieren von nCS muss sichergestellt sein, dass alle Zähler zurückgesetzt sind, um die Nachricht mit der Anzahl der Bytes korrekt zu erfassen. Diese setzt der else-Block ab Z. 91 um. Es gibt aber auch Situationen, in denen nCS inaktiv sein muss, die Nachricht aber noch nicht zuende ist. Bei einem Schreibvorgang ist es nötig, dass das WEL-Bit (Write enable latch) im Status Register gesetzt ist. Dies geschieht nach dem Senden des Opcodes WEL, wenn nCS deaktiviert ist. Da nach dem WREN-Befehl, opcode 8'h06, noch ein WRITE-Befehl mit Adresse und Daten folgen muss, dürfen sich die Zähler in diesem Fall nicht zurücksetzen, was hier mithilfe der ersten if-Bedingung umgesetzt ist.

State-Machine – Verhalten des FRAMs

```
  always @ (posedge byte_received) begin
    case (byte_count)
      //Byte 1 of message
      4'h1: begin //counting starts at 1, not 0.
        case (byte_data_received)
          8'h03: //READ Op-code
            opcode = 8'h03;
          8'h06: begin //WREN Op-Code
            opcode = 8'h06;
            #25; //wait one clock for nCS to get low
            if (nCS == 1) stat_reg [1] = 1; //Set WEL Bit in Status Register after one clock cycle
          end
          //READ STATUS REGISTER Op-Code
          8'h05: opcode = 8'h05;
          //HIBERNATE Op-Code
          8'hb9: begin
            opcode = 8'hb9;
          end
        endcase
      end
      //Byte 2 of message
      4'h2: begin
        case (byte_data_received)
          //WRITE
          8'h02: //WRITE Op-code, only if WREN op-code was executed, WRITE Op-code is permitted.
            if (opcode == 8'h06) opcode = 8'h02;
          default:
            //READ - get highest address byte
            if (opcode == 8'h03) //upper four bits are not used and are always 0
              //the address is shifted in from right to left. Byte_data_received is the highest byte of the address
              addr <= {4'b0000, 12'h000, byte_data_received};
        endcase
      end
      //Byte 3 of message
      4'h3: begin
        case (byte_data_received)
          default:
            //READ - get middle address byte
            if (opcode == 8'h03) //if opcode is read, the byte_data_received
                                 //is the next byte of the address, followed by 1 byte
              addr <= {4'b0000, 4'b0000, addr[7:0], byte_data_received};
          //WRITE - get highest address byte
          else if (opcode == 8'h02 && stat_reg[1] == 1'b1)
            addr <= {4'b0000, 12'h000, byte_data_received};
        endcase
      end
      //Byte 4 of message
      4'h4: begin
        case (byte_data_received)
          default:
            //READ - get the lowest byte of the address
            if (opcode == 8'h03) begin
              addr <= {addr[15:0], byte_data_received};
              send_data = 1; //sets the flag which starts sending every bit out of SO at memory address "addr".
            end
          //WRITE - get middle address byte
          else if (opcode == 8'h02 && stat_reg[1] == 1'b1)
            addr <= {4'b000, 4'b0000, addr[7:0], byte_data_received};
        endcase
      end
      //Byte 5 of message
      4'h5: begin
        case (byte_data_received)
          default:
            //WRITE - get lowest address byte and enable write_to_memory, the following bytes are data.
            if (opcode == 8'h02 && stat_reg[1] == 1'b1) begin
              addr <= {addr[15:0], byte_data_received};
              write_to_memory = 1; //set write to memory and wait one clock
            end
        endcase
      end
    endcase
  end
endmodule
```

Dieser Code beschreibt die State machine und ihr Verhalten.

Wenn ein vollständiges Byte empfangen wurde, ist byte_received == 1. Das löst den Always block in diesem Moment aus. Anhand des byte_count, also der Anzahl der Bytes, die für eine Nachricht geschickt sind, springt es in den zugehörigen Case, welche Im Code mit der Anzahl der Bytes kommentiert sind (Byte x of message).
Anhand des Inhalts des Bytes prüft nun das Modell mithilfe der Case Anweisung, wie es darauf

reagiert. Hier ist das Verhalten aller auftretenden Operationen, also Read Status Register, Read/Write Memory und Hibernation implementiert. Wieviele Bytes eine Operation hat, ist im vorigen Kapitel Beschreibung der Operationen nachzulesen.

Besonders ist das Verhalten bei READ/Write memory, da auch die Adresse einzulesen ist. Dies geschieht für Read in Byte 4'h2 bis 4'h4. Jedes neue Byte wird von rechts nach links in die Adresse hineingeschoben. Die oberen 4 Bits der Adresse sind nicht von Bedeutung, der Speicher nutzt diese nicht. Diese können einen beliebigen Wert annehmen [3, S. 7]. Die Kommentare im Code kennzeichnen die Stellen, die zu den entsprechenden Opcodes gehören, um die Übersichtlichkeit zu erhöhen.

SPI_FRAM_tb

Die Testbench überprüft alle Operationen, die implementiert sind. Die Testbench spielt die Rolle des Controllers, das heißt sie schreibt mit einem 40MHz SCK-Takt in den SI des FRAMs und überprüft die Ausgaben in SO. Die geschriebenen Bits stellen die Befehle dar. Nacheinander durchläuft die Testbench alle Befehle und überprüft mit Assertions den Zustand im FRAM-Modul auf den erwarteten Wert bei Erfolg der Operation. Die verschiedenen Tests und Prüfungen sind mit Kommentaren gekennzeichnet, was den Code selbst erklärt. Die Tests der Operationen sind identisch in ihrer grundsätzlichen Struktur, im Folgenden ist beispielhaft für die nachfolgenden Tests der Read Memory Operation Test erklärt.

Beispiel Read Memory Operation Test

```
34    //TEST READ MEMORY
35    // Sends 8'b00000011 as Read Opcode
36    SI = 0;
37    #25 SI = 0;
38    #25 SI = 0;
39    #25 SI = 0;
40    #25 SI = 0;
41    #25 SI = 0;
42    #25 SI = 1;
43    #25 SI = 1;
44    #25; assert (opcode == 8'h03);
45
46    //first byte (only the highest 4 bits are used) of 20-Bit address
47    SI = 0;
48    #25 SI = 0;
49    #25 SI = 0;
50    #25 SI = 0;
51    #25 SI = 0;
52    #25 SI = 0;
53    #25 SI = 0;
54    #25 SI = 0;
55
56    //second Byte of 20-Bit address
57    #25 SI = 0;
58    #25 SI = 0;
59    #25 SI = 0;
60    #25 SI = 0;
61    #25 SI = 0;
62    #25 SI = 0;
63    #25 SI = 0;
64    #25 SI = 0;
65    //third byte  of address
66    #25 SI = 0;
67    #25 SI = 0;
68    #25 SI = 0;
69    #25 SI = 0;
70    #25 SI = 0;
71    #25 SI = 0;
72    #25 SI = 1;
73    #25 SI = 1;
74
75    //read one Byte (200clocks/25 clocks per bit = 8 bit)
76    #25 assert (addr == 24'h000003); //check address
77    //check for correct writing to SPI out of memory
78    #25 assert (SO == 0);
79    #25 assert (SO == 0);
80    #25 assert (SO == 1);
81    #25 assert (SO == 1);
82    #25 assert (SO == 0);
83    #25 assert (SO == 0);
84    #25 assert (SO == 1);
85    #25 assert (SO == 1);
86
87    //Message is finished, so nCS is not active
88    nCS = 1;
```

Die ersten 8 Bits senden den opcode. Anschließend folgt eine Assertion in Zeile 44, welche überprüft, ob der der FRAM den Opcode empfangen hat. Danach sendet die Testbench 3 Bytes mit der Adresse, wobei der FRAM die oberen 4 Bits der resultierenden Adresse abschneidet. Anschließend prüft die Assertion in Z. 76 die Adresse. Danach schreibt der SPI-Slave ein Byte aus dem FRAM in den Serial Out der SPI.
Die folgenden Assertions ab Z. 78 testen die korrekte Ausgabe. Wenn alle Tests bestanden sind ist garantiert, dass der SPI-Slave korrekt die Daten aus dem FRAM ausliest und in die Schnittstelle schreibt.

Zusammenfassung

das SPI-FRAM-Modul ist durch die Testbench mit Assertions getestet und hat diese fehlerfrei bestanden. Die Operationen des READ Memory, WRITE Memory, Read Status Register und Hibernation sind implementiert. Der FRAM ist aber nur mit SPI Mode 0 getestet. Mit SPI Mode 3 sollte das FRAM-Modul prinzipiell auch funktionieren, jedoch ist die Testbench nicht fähig, den FRAM mit SPI Mode 3 zu testen, da die Zeiteingaben hart kodiert sind und durch den Wechsel von SPI Mode 0 zu 3 Timing Probleme entstehen. Da der Wechsel von SPI Mode 0 zu 3 lediglich die das Leerlaufverhalten von SCK festlegt, sollte das Modul in Kombination mit einem SPI-Controller fähig sein, beide Modi zu unterstützen. Das Modul sollte jetzt in der Lage sein, mit einem SPI-Controller zu kommunizieren und sich dabei wie ein FRAM-Baustein zu verhalten.

Literatur

[1] Willi Friedrich, *Die Zweidrahtbussysteme I²C-Bus und SPI-Bus: Eigenschaften, Protokolle, Anwendungen im Vergleich der beiden Systeme.* [Online]. Verfügbar unter: https://docplayer.org/5099979-Die-zweidrahtbussysteme-i2c-bus-und-spi-bus-eigenschaften-protokolle-anwendungen-im-vergleich-der-beiden-systeme.html (Zugriff am: 6. Juni 2021).

[2] Texas Instruments, *FRAM FAQs.* [Online]. Verfügbar unter: https://www.ti.com/lit/pdf/slat151 (Zugriff am: 2. Juni 2021).

[3] Cypress, *Excelon™ LP 8-Mbit (1024K × 8) Serial (SPI) F-RAM.* [Online]. Verfügbar unter: https://www.cypress.com/file/444186/download (Zugriff am: 2. Juni 2021).

[4] Microchip, *I2C™ Serial EEPROM Family Data Sheet.* [Online]. Verfügbar unter: http://ww1.microchip.com/downloads/en/DeviceDoc/I2C%20Serial%20EE%20Family%20Data%20Sheet%2021930C.pdf (Zugriff am: 2. Juni 2021).

[5] Harsha Medu, *SPI Guide for F-RAM™.* [Online]. Verfügbar unter: https://www.cypress.com/documentation/application-notes/an304-spi-guide-f-ram (Zugriff am: 7. Juni 2021).

[6] fpga4fun.com, *SPI 2 - A simple implementation.* [Online]. Verfügbar unter: https://www.fpga4fun.com/SPI2.html.

[7] Will Green, *Initialize Memory in Verilog.* [Online]. Verfügbar unter: https://projectf.io/posts/initialize-memory-in-verilog/ (Zugriff am: 8. Juni 2021).