WGR-V Verilog Doku

Inhaltsverzeichnis

Datei: rtl/alu.v

Modul: alu

Kurzbeschreibung: Arithmetisch-Logische Einheit (ALU) für 32-Bit-Operationen.

Dieses Modul führt verschiedene arithmetische und logische Operationen auf zwei 32-Bit-Operanden durch, basierend auf einem 4-Bit-Steuersignal. Zusätzlich erzeugt es ein Zero-Flag, falls das Ergebnis 0 ist. Unterstützte Operationen (OP_...): - ADD (4'b0000) Addieren - SUB (4'b0001) Subtrahieren - AND (4'b0010) Bitwise UND - OR (4'b0011) Bitwise ODER - XOR (4'b0100) Bitwise EXKLUSIVE ODER - SLL (4'b0101) Bitweises Shift-Left - SRL (4'b0110) Bitweises Shift-Right (logisch) -

SRA (4'b0111) Vorzeichenbehaftetes, bitweises Shift-Right (arithmetisch) - SLT (4'b1000) Vergleich (signed) - SLTU (4'b1001) Vergleich (unsigned)

Lokale Parameter:

- `OP_ADD` = 4'b0000
- `OP_SUB` = 4'b0001
- `OP_AND` = 4'b0010
- `OP_OR` = 4'b0011
- `OP_XOR` = 4'b0100
- `OP_SLL` = 4'b0101
- `OP_SRL` = 4'b0110
- `OP_SRA` = 4'b0111
- `OP_SLT` = 4'b1000
- `OP_SLTU` = 4'b1001

Eingänge:

- `[31:0]` operand1 Erstes Eingangsoperand
- `[31:0]` operand2 Zweites Eingangsoperand
- [3:0] operation 4-Bit-Operationscode

Ausgänge:

- `reg` [31:0] result Ergebnis des Rechenvorgangs
- `wire` zero Signal, falls result=0

Source Code

```
`default_nettype none
`timescale 1ns / 1ns



module alu (
  input  wire [31:0] operand1,
  input  wire [31:0] operand2,
  input  wire [ 3:0] operation,
  output reg  [31:0] result,
  output wire        zero
);

  // ------------------------------------------------------
  // Operationen als lokale Konstanten
  // ------------------------------------------------------
  localparam OP_ADD  = 4'b0000;
  localparam OP_SUB  = 4'b0001;
  localparam OP_AND  = 4'b0010;
  localparam OP_OR   = 4'b0011;
  localparam OP_XOR  = 4'b0100;
  localparam OP_SLL  = 4'b0101;
  localparam OP_SRL  = 4'b0110;
  localparam OP_SRA  = 4'b0111;
  localparam OP_SLT  = 4'b1000;
  localparam OP_SLTU = 4'b1001;

  // ------------------------------------------------------
  // Vorzeichenbehaftete Interpretionen der Operanden
  // ------------------------------------------------------
  wire signed [31:0] sign_op1;
  wire signed [31:0] sign_op2;

  assign sign_op1 = $signed(operand1);
  assign sign_op2 = $signed(operand2);
```

```
  assign zero = (result == 32'd0);

  // -----------------------------------------------------
  // Hauptzuweisung für result abhängig von operation
  // -----------------------------------------------------
  always @( * )
  begin
    case (operation)

      OP_ADD:
        result =  operand1  +  operand2;

      OP_SUB:
        result =  sign_op1  -  sign_op2;

      OP_AND:
        result =  operand1  &  operand2;

      OP_OR:
        result =  operand1  |  operand2;

      OP_XOR:
        result =  operand1  ^  operand2;

      OP_SLL:
        result =  operand1 <<  operand2[4:0];

      OP_SRL:
        result =  operand1  >> operand2[4:0];

      OP_SRA:
        result =  sign_op1 >>> operand2[4:0];

      OP_SLT:
        result = (sign_op1  <  sign_op2) ? 32'd1 : 32'd0;

      OP_SLTU:
        result = (operand1  <  operand2) ? 32'd1 : 32'd0;


      default:
        result = 32'd0;

    endcase
  end

endmodule
```

Datei: rtl/cpu.v

Modul: cpu

Kurzbeschreibung: Einfacher CPU-Kern mit RISC-V-ähnlichem Befehlsformat.

Dieser CPU-Kern holt Instruktionen aus dem Speicher (memory), decodiert sie (Opcode, Funct3, Funct7, Registeradressen etc.) und führt sie mittels einer ALU aus. Lese- und Schreibzugriffe auf Speicher oder Peripherie erfolgen über die Signale `address`, `write_data`, `we`, `re`, `read_data` und `mem_busy`. Der CPU-Core durchläuft eine einfache 5-stufige Pipeline in zeitversetzten Zuständen (FETCH, DECODE, EXECUTE, MEMORY, WRITEBACK), allerdings sequentiell (kein echtes Pipeline-Overlap). Hier eine Kurzbeschreibung der Zustände: - FETCH : Eine Instruktion wird über re=1 im Speicher angefordert. - WAIT : Auf das Eintreffen von read_data wird gewartet. - DECODE : Instruktionsbits (opcode, funct3, funct7,

rs1, rs2, rd usw.) extrahieren. - EXECUTE: ALU-Operationen durchführen, Sprungadressen berechnen usw. - MEMORY : Lese-/Schreibzugriffe bei LOAD/STORE - MEMHALT: Warten, bis der Speicher- oder Peripheriezugriff abgeschlossen ist. - RMW_WAIT & STORE_RMW: Dienen dem atomaren Lesen-Schreiben (Byte/16-Bit) bei LOAD/STORE. - WRITEBACK: Ergebnis in Register ablegen (falls erforderlich) und PC inkrementieren oder Branch ausführen. Die ALU-Operationen nutzen ein 4-Bit-Steuersignal (z. B. OP_ADD), das anhand von opcode/funct3/funct7 generiert wird.

Lokale Parameter:

- `FETCH` = 4'd0
- `WAIT` = 4'd1
- `DECODE` = 4'd2
- `EXECUTE` = 4'd3
- `MEMORY` = 4'd4
- `MEMHALT` = 4'd5
- `WRITEBACK=` 4'd6
- `RMW_WAIT` = 4'd7
- `STORE_RMW=` 4'd8

Eingänge:

- `clk` Systemtakt
- `rst_n` Asynchroner, aktiver-LOW Reset
- `[31:0]` read_data Daten, die bei re=1 aus dem Speicher gelesen werden
- `wire` mem_busy Signalisiert, ob der Speicherzugriff noch in Arbeit ist

Ausgänge:

- `[31:0]` address Speicheradresse für Lese-/Schreiboperationen
- `reg` [31:0] write_data Daten, die bei we=1 in den Speicher geschrieben werden
- `reg` we Write-Enable
- `reg` re Read-Enable

Source Code

```
`default_nettype none
`timescale 1ns / 1ns




module cpu (
  input  wire        clk,
  input  wire        rst_n,
  output wire [31:0] address,
  output reg  [31:0] write_data,
  input  wire [31:0] read_data,
  output reg         we,
  output reg         re,
  input  wire        mem_busy
);

  // ------------------------------------------------------
  // Definition der Zustände / 5-stufige Pipeline
  // ------------------------------------------------------
localparam [3:0]
    FETCH        = 4'd0,
    WAIT         = 4'd1,
    DECODE       = 4'd2,
    EXECUTE      = 4'd3,
    MEMORY       = 4'd4,
    MEMHALT      = 4'd5,
    WRITEBACK    = 4'd6,
    RMW_WAIT     = 4'd7,
    STORE_RMW    = 4'd8;
```

```
  // ------------------------------------------------------
  // ALU-Operationscodes
  // ------------------------------------------------------
localparam [3:0]
    OP_ADD      = 4'b0000,
    OP_SUB      = 4'b0001,
    OP_AND      = 4'b0010,
    OP_OR       = 4'b0011,
    OP_XOR      = 4'b0100,
    OP_SLL      = 4'b0101,
    OP_SRL      = 4'b0110,
    OP_SRA      = 4'b0111,
    OP_SLT      = 4'b1000,
    OP_SLTU     = 4'b1001;

  // ------------------------------------------------------
  // Funct3-Codes
  // ------------------------------------------------------
localparam [2:0]
    F3_ADD_SUB  = 3'b000,
    F3_SLL      = 3'b001,
    F3_SLT      = 3'b010,
    F3_SLTU     = 3'b011,
    F3_XOR      = 3'b100,
    F3_SRL_SRA  = 3'b101,
    F3_OR       = 3'b110,
    F3_AND      = 3'b111;

localparam [2:0]
    F3_LB       = 3'b000,
    F3_LH       = 3'b001,
    F3_LW       = 3'b010,
    F3_LBU      = 3'b100,
    F3_LHU      = 3'b101;

localparam [2:0]
    F3_SB       = 3'b000,
    F3_SH       = 3'b001,
    F3_SW       = 3'b010;

localparam [2:0]
    F3_BEQ      = 3'b000,
    F3_BNE      = 3'b001,
    F3_BLT      = 3'b100,
    F3_BGE      = 3'b101,
    F3_BLTU     = 3'b110,
    F3_BGEU     = 3'b111;

  // ------------------------------------------------------
  // Funct7 / OPCODE
  // ------------------------------------------------------
localparam [6:0]
    F7_ADD      = 7'b0000000,
    F7_SUB      = 7'b0100000,
    F7_SLL      = 7'b0000000,
    F7_SLT      = 7'b0000000,
    F7_SLTU     = 7'b0000000,
    F7_XOR      = 7'b0000000,
    F7_SRL      = 7'b0000000,
```

```
    F7_SRA       = 7'b0100000,
    F7_OR        = 7'b0000000,
    F7_AND       = 7'b0000000;

localparam [6:0]
    OPCODE_LUI      = 7'b0110111,
    OPCODE_AUIPC    = 7'b0010111,
    OPCODE_JAL      = 7'b1101111,
    OPCODE_JALR     = 7'b1100111,
    OPCODE_BRANCH   = 7'b1100011,
    OPCODE_LOAD     = 7'b0000011,
    OPCODE_STORE    = 7'b0100011,
    OPCODE_OP_IMM   = 7'b0010011,
    OPCODE_OP       = 7'b0110011;

  // ------------------------------------------------------------
  // CPU-Register (PC, Zwischenspeicher etc.)
  // ------------------------------------------------------------
  reg [31:0] PC;
  reg [31:0] inst;
  reg [31:0] address_reg;
  reg [31:0] reg_w_data;
  reg [31:0] alu_operand1;
  reg [31:0] alu_operand2;
  reg [31:0] imm;
  reg [ 4:0] reg_write_addr;
  reg [ 3:0] alu_op;
  reg [ 3:0] state;
  reg [ 3:0] next_state;
  reg [ 1:0] mem_offset;
  reg        reg_we;

  // ------------------------------------------------------------
  // ALU-Anbindung
  // ------------------------------------------------------------
  wire [31:0] rs1_data;
  wire [31:0] rs2_data;
  wire [31:0] alu_result;
  wire [ 6:0] opcode;
  wire [ 6:0] funct7;
  wire [ 4:0] rs1;
  wire [ 4:0] rs2;
  wire [ 4:0] rd;
  wire [ 2:0] funct3;
  wire        alu_zero;

  // ------------------------------------------------------------
  // Zuweisungen
  // ------------------------------------------------------------
  assign address = address_reg;

  assign opcode = inst[ 6: 0];
  assign funct7 = inst[31:25];
  assign rs1    = inst[19:15];
  assign rs2    = inst[24:20];
  assign rd     = inst[11: 7];
  assign funct3 = inst[14:12];

  // ------------------------------------------------------------
  // Registerdatei-Instanzierung
```

```
// ------------------------------------------------------
register_file reg_file (
  .rst_n     (rst_n),
  .clk       (clk),
  .we        (reg_we),
  .rd        (reg_write_addr),
  .rs1       (rs1),
  .rs2       (rs2),
  .rd_data   (reg_w_data),
  .rs1_data  (rs1_data),
  .rs2_data  (rs2_data)
);

// ------------------------------------------------------
// ALU-Instanzierung
// ------------------------------------------------------
alu alu_inst (
  .operand1  (alu_operand1),
  .operand2  (alu_operand2),
  .operation (alu_op),
  .result    (alu_result),
  .zero      (alu_zero)
);

// ------------------------------------------------------
// Erzeugung von 'imm' aus der Instruktion (verschiedene Typen)
// ------------------------------------------------------
always @( * )
begin

  case (opcode)

    OPCODE_OP_IMM,
    OPCODE_LOAD,
    OPCODE_JALR:
      // Sign extension für 12-Bit
      imm = {{20{inst[31]}}, inst[31:20]};

    OPCODE_STORE:
      // Sign extension für 12-Bit (Split in inst[31:25] & inst[11:7])
      imm = {{20{inst[31]}},
                 inst[31:25],
                 inst[11:7]};

    OPCODE_BRANCH:
      // Sign extension + Bit[7] + Bit[30:25] + Bit[11:8] + 0
      imm = {{20{inst[31]}},
                 inst[7],
                 inst[30:25],
                 inst[11: 8],
                 1'b0};

    OPCODE_LUI,
    OPCODE_AUIPC:
      imm = {inst[31: 12], 12'b0};

    OPCODE_JAL:
      // 20-Bit Sign extension, plus die gemischten Bits (inst[31], inst[19:12], inst[20], inst[30:21], 0)
      imm = {{11{inst[31]}},
                 inst[31],
```

```
                inst[19:12],
                inst[20],
                inst[30:21],
                1'b0};

    default: imm = 32'd0;

  endcase
end

// ------------------------------------------------------
// Nächster Zustand (next_state) basierend auf state + Signalen
// ------------------------------------------------------
always @( * )
begin
  case (state)

    FETCH:
      next_state = WAIT;

    WAIT:
      next_state = (mem_busy || re) ? WAIT : DECODE;

    DECODE:
      next_state = EXECUTE;

    EXECUTE:
    begin
      // Bei LOAD/STORE -> in den MEMORY-Zustand
      if (opcode == OPCODE_LOAD || opcode == OPCODE_STORE)
        next_state = MEMORY;
      else
        next_state = WRITEBACK;
    end

    MEMORY:
    begin
      if (opcode == OPCODE_LOAD)
        next_state = (mem_busy || re) ? MEMORY : MEMHALT;
      else
        if (opcode == OPCODE_STORE)
        begin
          if (funct3 == F3_SW)
            next_state = (mem_busy || re) ? MEMORY : MEMHALT;
          else
          // Byte/Halfword => Read-Modify-Write
            if (funct3 == F3_SB || funct3 == F3_SH)
              next_state = (mem_busy || re) ? MEMORY : RMW_WAIT;
            else
              next_state = (mem_busy || re) ? MEMORY : MEMHALT;
        end
        else
          next_state = WRITEBACK;
    end

    RMW_WAIT:
      // Warte auf Ende des Speicherzugriffs
      next_state = (mem_busy || re || we) ? RMW_WAIT  : STORE_RMW;
    STORE_RMW:
      // Warte erneut bis busy + re/we durch sind
```

```
      next_state = (mem_busy || re || we) ? STORE_RMW : MEMHALT;
    MEMHALT:
      // Warte bis busy und re/we beendet
      next_state = (mem_busy || re || we) ? MEMHALT   : WRITEBACK;
    WRITEBACK:
      next_state = FETCH;
    default:
      next_state = FETCH;

  endcase
end

// ------------------------------------------------------
// Zustandsübergänge und Ausführung
// ------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    state       <= FETCH;
    PC          <= 32'h00004000; // Startadresse
    inst        <= 32'h00000013;
    reg_we      <= 1'b0;
    re          <= 1'b0;
    we          <= 1'b0;
    write_data  <= 32'd0;
    address_reg <= 32'h00004000;
    mem_offset  <= 2'b00;
  end
  else
  begin
    state      <= next_state;
    reg_we     <= 1'b0;  // Standard: kein Registerwrite
    write_data <= 32'd0; // Standard: Kein Schreiben
    re         <= 1'b0;
    we         <= 1'b0;

    case (state)

      // ------------------------------------------------
      // FETCH: Instruktion anfordern (re=1, address=PC)
      // ------------------------------------------------
      FETCH:
      begin
        re          <= 1'b1;
        address_reg <= PC;
      end

      // ------------------------------------------------
      // WAIT: Warten auf read_data vom Speicher
      // ------------------------------------------------
      WAIT:
      begin
        re   <= 1'b0;
      end


      // ------------------------------------------------
      // DECODE: Instruktion in 'inst' puffer, parse
      // ------------------------------------------------
```

```
DECODE:
begin
  inst <= read_data;
end

// ------------------------------------------------
// EXECUTE: Berechne ALU-Operation (ALU-Setup)
// ------------------------------------------------
EXECUTE:
begin
  case (opcode)

    OPCODE_OP:
    begin
      alu_operand1 <= rs1_data;
      alu_operand2 <= rs2_data;

      case (funct3)

        F3_ADD_SUB:
          alu_op <= funct7[5] ? OP_SUB : OP_ADD;

        F3_AND:
          alu_op <= OP_AND;

        F3_OR:
          alu_op <= OP_OR;

        F3_XOR:
          alu_op <= OP_XOR;

        F3_SLL:
          alu_op <= OP_SLL;

        F3_SRL_SRA:
          alu_op <= funct7[5] ? OP_SRA : OP_SRL;

        F3_SLT:
          alu_op <= OP_SLT;

        F3_SLTU:
          alu_op <= OP_SLTU;

        default:
          alu_op <= 4'b1111; //NOP ausführen

      endcase
    end

    OPCODE_OP_IMM:
    begin
      alu_operand1 <= rs1_data;

      if (funct3 == F3_SLL || funct3 == F3_SRL_SRA)
        alu_operand2 <= inst[24: 20];
      else
        alu_operand2 <= imm;

      case (funct3)
```

```
      F3_ADD_SUB:
        alu_op <= OP_ADD;

      F3_AND:
        alu_op <= OP_AND;

      F3_OR:
        alu_op <= OP_OR;

      F3_XOR:
        alu_op <= OP_XOR;

      F3_SLT:
        alu_op <= OP_SLT;

      F3_SLTU:
        alu_op <= OP_SLTU;

      F3_SLL:
        alu_op <= OP_SLL;

      F3_SRL_SRA:
        alu_op <= (inst[30]) ? OP_SRA : OP_SRL;

      default:
        alu_op <= 4'b1111;

    endcase
  end

  OPCODE_LOAD:
  begin
    alu_operand1 <= rs1_data;
    alu_operand2 <= imm;
    alu_op       <= OP_ADD;
  end

  OPCODE_STORE:
  begin
    alu_operand1 <= rs1_data;
    alu_operand2 <= imm;
    alu_op       <= OP_ADD;
  end

 OPCODE_JAL:
  begin
    alu_operand1 <= PC;
    alu_operand2 <= imm;
    alu_op       <= OP_ADD;
  end

  OPCODE_JALR:
  begin
    alu_operand1 <= rs1_data;
    alu_operand2 <= imm;
    alu_op       <= OP_ADD;
  end

  OPCODE_BRANCH:
  begin
```

```
      alu_operand1 <= rs1_data;
      alu_operand2 <= rs2_data;
      alu_op       <= OP_SUB;
    end

    OPCODE_AUIPC:
    begin
      alu_operand1 <= PC;
      alu_operand2 <= imm;
      alu_op       <= OP_ADD;
    end

    OPCODE_LUI:
    begin
      alu_operand1 <= 32'd0;
      alu_operand2 <= imm;
      alu_op       <= OP_ADD;
    end

    default:
    begin
      alu_operand1 <= 32'd0;
      alu_operand2 <= 32'd0;
      alu_op       <= 4'b1111;
    end

  endcase
end

// ------------------------------------------------
// MEMORY: Lese-/Schreibzugriff
// ------------------------------------------------
MEMORY:
begin
  if (opcode == OPCODE_LOAD)
  begin
    re          <= 1'b1;
    address_reg <= alu_result & 32'hFFFFFFFC;
    mem_offset  <= alu_result[1: 0];
  end
  else
    if (opcode == OPCODE_STORE)
    begin
      if (funct3 == F3_SW)
      begin
        write_data  <= rs2_data;
        we          <= 1'b1;
        address_reg <= alu_result;
      end
      else
        if (funct3 == F3_SB || funct3 == F3_SH)
        begin
          re          <= 1'b1;
          address_reg <= alu_result & 32'hFFFFFFFC;
          mem_offset  <= alu_result[1:0];
        end
    end
end

// ------------------------------------------------
```

```
// RMW_WAIT: Warten bis read_data ok
// ------------------------------------------------
RMW_WAIT:
begin
  // Warten bis read_data ok
end

// ------------------------------------------------
// STORE_RMW: Daten anpassen und we=1
// Byte-/Halbwort-Schreiben erfordert zuerst ein Lesen des Zielwortes (Read-Modify-Write)
// ------------------------------------------------
STORE_RMW:
begin
  if (funct3 == F3_SB)
  begin
    case (mem_offset[1: 0])

      2'b00:
        write_data <= {read_data[31:8], rs2_data[7:0]};

      2'b01:
        write_data <= {read_data[31:16], rs2_data[7:0], read_data[7: 0]};

      2'b10:
        write_data <= {read_data[31:24], rs2_data[7:0], read_data[15: 0]};

      2'b11:
        write_data <= {rs2_data[7:0], read_data[23:0]};

      default:
        write_data <= read_data;

    endcase
  end
  else
    if (funct3 == F3_SH)
    begin
      if (mem_offset[1] == 1'b0)
        write_data <= {read_data[31: 16], rs2_data[15: 0]};
      else
        write_data <= {rs2_data[15: 0], read_data[15: 0]};
    end
    else
      write_data <= read_data;

  we <= 1'b1;
end

// ------------------------------------------------
// MEMHALT: Warte auf Freigabe
// ------------------------------------------------
MEMHALT:
begin
  // wait
end

// ------------------------------------------------
// WRITEBACK: ALU-Ergebnis oder Speicherergebnisse
//            in Register ablegen. PC update.
// ------------------------------------------------
```

```
WRITEBACK:
begin
  if (opcode == 7'b0000011)
  begin
    case (funct3)

      F3_LB:
      begin
        case (mem_offset[1: 0])
          2'b00:   reg_w_data <= {{24{read_data[ 7]}}, read_data[ 7: 0]};
          2'b01:   reg_w_data <= {{24{read_data[15]}}, read_data[15: 8]};
          2'b10:   reg_w_data <= {{24{read_data[23]}}, read_data[23:16]};
          2'b11:   reg_w_data <= {{24{read_data[31]}}, read_data[31:24]};
          default: reg_w_data <= read_data;
        endcase
      end

      F3_LH:
      begin
        if (mem_offset[1] == 1'b0)
          reg_w_data <= {{16{read_data[15]}}, read_data[15:0]};
        else
          reg_w_data <= {{16{read_data[31]}}, read_data[31:16]};
      end

      F3_LW:
      begin
        reg_w_data <= read_data;
      end

      F3_LBU:
      begin
        case (mem_offset[1: 0])

          2'b00:
            reg_w_data <= {24'd0, read_data[ 7:0]};

          2'b01:
            reg_w_data <= {24'd0, read_data[15:8]};

          2'b10:
            reg_w_data <= {24'd0, read_data[23:16]};

          2'b11:
            reg_w_data <= {24'd0, read_data[31:24]};

          default:
            reg_w_data <= read_data;

        endcase
      end

      F3_LHU:
      begin
        if (mem_offset[1] == 1'b0)
          reg_w_data <= {16'd0, read_data[15: 0]};
        else
          reg_w_data <= {16'd0, read_data[31:16]};
      end
```

```
    default:
      reg_w_data <= read_data;

  endcase
end

else if (opcode == OPCODE_JAL ||
         opcode == OPCODE_JALR)
  reg_w_data <= PC + 4;

else if (opcode == OPCODE_LUI)
  reg_w_data <= imm;

else if (opcode == OPCODE_AUIPC)
  reg_w_data <= PC + imm;

else if (opcode == OPCODE_OP_IMM ||
         opcode == OPCODE_OP)
  reg_w_data <= alu_result;

else
  reg_w_data <= 32'd0;

// Bei bestimmten Opcodes muss in rd geschrieben werden
if (opcode == OPCODE_LOAD  || opcode == OPCODE_JAL    ||
    opcode == OPCODE_JALR  || opcode == OPCODE_LUI    ||
    opcode == OPCODE_AUIPC || opcode == OPCODE_OP_IMM ||
    opcode == OPCODE_OP)
begin
  reg_write_addr <= rd;
  reg_we <= (rd != 5'd0);
end

else
  reg_we <= 1'b0;

// PC-Update
if (opcode == OPCODE_JAL)
  PC <= alu_result;
else
  if (opcode == OPCODE_JALR)
    PC <= alu_result & ~32'd1;
  else
    if (opcode == OPCODE_BRANCH)
    begin
      case (funct3)

        F3_BEQ:
          PC <= (alu_zero) ? (PC + imm) : (PC + 4);

        F3_BNE:
          PC <= (!alu_zero) ? (PC + imm) : (PC + 4);

        F3_BLT:
          PC <= ($signed(rs1_data) < $signed(rs2_data)) ? (PC + imm) : (PC + 4);

        F3_BGE:
          PC <= ($signed(rs1_data) >= $signed(rs2_data)) ? (PC + imm) : (PC + 4);

        F3_BLTU:
```

```
                    PC <= (rs1_data < rs2_data) ? (PC + imm) : (PC + 4);

                  F3_BGEU:
                    PC <= (rs1_data >= rs2_data) ? (PC + imm) : (PC + 4);

                  default:
                    PC <= PC + 4;

                endcase
              end
              else
                PC <= PC + 4;
        end

        default: ;

      endcase
    end
  end

endmodule
```

Datei: rtl/defines.v

Modul: defines

Kurzbeschreibung: Globale Definitionsdatei (Makros und Parameter).

Diese Datei definiert verschiedene globale Konstante, Parameter und Makros, die in den restlichen Modulen verwendet werden. Sie legt unter anderem die Taktfrequenz (`CLK_FREQ`), UART- und SPI-FIFO-Größen fest, sowie Flags zum Bedingten Integrieren bestimmter Module (z. B. `INCLUDE_UART`).

Modul: defines

Kurzbeschreibung: Gibt die Taktfrequenz des Systems in Hz an, z. B. 12 MHz.

module defines

Parameter:

- `CLK_FREQ`

Modul: Unknown

Kurzbeschreibung: Berechnet den Teilerwert für eine angegebene Baudrate.

@module defines Beispiel: `BAUD_DIV(115200)` bei 12 MHz ergibt den passenden Teilwert.

Parameter:

- `BAUD_DIV(Baud)`

Modul: Unknown

Kurzbeschreibung: Aktiviert (falls definiert) den Betrieb als 32-Bit RISC-V-Kern.

Parameter:

- `RV32I`

Modul: Unknown

Kurzbeschreibung: Aktiviert (falls definiert) die Verwendung des FRAM-Speichers statt internem RAM.

Parameter:

- `FRAM_MEMORY`

Modul: Unknown

Kurzbeschreibung: Legt die Tiefe (Anzahl Einträge) für die TX- und RX-FIFOs des UART fest.

Parameter:

- `UART_FIFO_TX_DEPTH`
- `UART_FIFO_RX_DEPTH`

Modul: Unknown

Kurzbeschreibung: Legt die Tiefe (Anzahl Einträge) für die TX- und RX-FIFOs des SPI fest.

Parameter:

- `SPI_FIFO_TX_DEPTH`
- `SPI_FIFO_RX_DEPTH`

Modul: Unknown

Kurzbeschreibung: Schalter zum bedingten Einbinden der jeweiligen Peripheriemodule.

Parameter:

- `INCLUDE_DEBUG`, INCLUDE_UART, INCLUDE_TIME, ...

Source Code

```
`ifndef DEFINES_V
`define DEFINES_V




`define CLK_FREQ       12_000_000


`define BAUD_DIV(Baud) ((`CLK_FREQ / Baud) - 1)


`define RV32I


//`define FRAM_MEMORY


`define UART_FIFO_TX_DEPTH 4
`define UART_FIFO_RX_DEPTH 4


`define SPI_FIFO_TX_DEPTH 4
`define SPI_FIFO_RX_DEPTH 4


`define INCLUDE_DEBUG
`define INCLUDE_UART
`define INCLUDE_TIME
`define INCLUDE_PWM
`define INCLUDE_MULT
`define INCLUDE_DIV
`define INCLUDE_SPI
`define INCLUDE_GPIO
`define INCLUDE_WS

`endif
```

Datei: rtl/fram/fram_ram.v

Modul: fram_ram

Kurzbeschreibung: Abstraktes FRAM-Zugriffsmodul, das nach außen wie RAM erscheint,

intern jedoch über SPI kommuniziert. Dieses Modul nimmt Lese- und Schreibanforderungen (über address, we, re) entgegen und leitet sie an das `fram_spi`-Modul weiter. Nach Abschluss der SPI-Operation wird `read_data` gültig, bzw. die Schreiboperation ist abgeschlossen. Ein `req_ready`-Signal zeigt an, wann das Modul wieder bereit für neue Befehle ist. entgegen genommen werden kann

Lokale Parameter:

- `ST_IDLE` Leerlaufzustand (wartet auf we/re)
- `ST_START` Vorbereitung der SPI-Operation (Adress-/Datenübergabe)
- `ST_WAIT` Wartezustand, bis `fram_spi` fertigsignalisiert
- `ST_DONE` Abschluss der Operation, Daten liegen vor, req_ready=1

Eingänge:

- `clk` Systemtakt
- `rst_n` Asynchrones, aktives-LOW Reset
- `[15:0]` address Adress-Eingang (16 Bit, FRAM-Adressbereich)
- `[31:0]` write_data Daten, die bei we=1 in den FRAM geschrieben werden
- `we` Schreibeaktivierung (Write Enable)
- `re` Leseaktivierung (Read Enable)
- `spi_miso` SPI-Eingang (Master In, Slave Out)

Ausgänge:

- `reg` req_ready Signal, das anzeigt, ob ein neuer Lese-/Schreibzugriff
- `reg` [31:0] read_data Ausgelesene Daten bei einem Lesezugriff
- `spi_mosi` SPI-Ausgang (Master Out, Slave In)
- `spi_clk` SPI-Taktleitung
- `spi_cs` SPI-Chip-Select

Source Code

```
`default_nettype none
`timescale 1ns / 1ns



module fram_ram (
    input  wire        clk,
    input  wire        rst_n,
    output reg         req_ready,
    input  wire [15:0] address,
    input  wire [31:0] write_data,
    output reg  [31:0] read_data,
    input  wire        we,
    input  wire        re,
    output wire        spi_mosi,
    input  wire        spi_miso,
    output wire        spi_clk,
    output wire        spi_cs,
);


  // ----------------------------------------------------------
  // Zustandsdefinitionen für die interne Steuerung
  // ----------------------------------------------------------
  localparam [2:0] ST_IDLE  = 3'd0;
  localparam [2:0] ST_START = 3'd1;
  localparam [2:0] ST_WAIT  = 3'd2;
```

```
localparam [2:0] ST_DONE  = 3'd3;

// ------------------------------------------------------
// Registervariablen
// ------------------------------------------------------
reg  [31:0] spi_write_data;
reg  [31:0] lat_write_data;
reg  [15:0] spi_address;
reg  [15:0] lat_address;
reg  [ 2:0] state;
reg         spi_we;
reg         latched_we;
reg         spi_re;


// ------------------------------------------------------
// Rückgabedaten und Status von fram_spi
// ------------------------------------------------------
wire [31:0] spi_read_data;
wire        spi_done;

// ------------------------------------------------------
// Instanzierung des FRAM-SPI-Moduls
// ------------------------------------------------------
fram_spi fram_spi_inst (
  .clk         (clk),
  .rst_n       (rst_n),
  .address     (spi_address),
  .write_data  (spi_write_data),
  .read_data   (spi_read_data),
  .we          (spi_we),
  .re          (spi_re),
  .done        (spi_done),
  .spi_mosi    (spi_mosi),
  .spi_miso    (spi_miso),
  .spi_clk     (spi_clk),
  .spi_cs      (spi_cs)
);


// ------------------------------------------------------
// Zustandsmaschine zur Abwicklung eines Lese-/Schreibzugriffs
// ------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    state          <= ST_IDLE;
    req_ready      <= 1'b1;
    latched_we     <= 1'b0;
    lat_address    <= 16'd0;
    lat_write_data <= 32'd0;
    read_data      <= 32'd0;
    spi_re         <= 1'b0;
    spi_we         <= 1'b0;
    spi_address    <= 16'd0;
    spi_write_data <= 32'd0;
  end
  else
  begin
```

```
spi_we <= 1'b0;
spi_re <= 1'b0;

case (state)

  // ------------------------------------------------
  // ST_IDLE: Warten auf we oder re
  // ------------------------------------------------
  ST_IDLE: begin
    if (we || re)
    begin
      lat_address    <= address;
      lat_write_data <= write_data;
      latched_we     <= we;
      req_ready      <= 1'b0;
      state          <= ST_START;
    end
    else
      req_ready <= 1'b1;
  end


  // ------------------------------------------------
  // ST_START: Setze Parameter für fram_spi
  // ------------------------------------------------
  ST_START:
  begin
    spi_address    <= lat_address;
    spi_write_data <= lat_write_data;

    if (latched_we)
      spi_we <= 1'b1;
    else
      spi_re  <= 1'b1;

    state <= ST_WAIT;
  end

  // ------------------------------------------------
  // ST_WAIT: Warten, bis fram_spi fertig signalisiert (spi_done)
  // ------------------------------------------------
  ST_WAIT:
  begin
    if (spi_done)
    begin
      if (!latched_we)
        read_data <= spi_read_data;

      state <= ST_DONE;
    end
  end

  // ------------------------------------------------
  // ST_DONE: Vorgang abgeschlossen, req_ready = 1
  // ------------------------------------------------
  ST_DONE:
  begin
    req_ready <= 1'b1;
    state     <= ST_IDLE;
  end
```

```
        default:
          state <= ST_IDLE;

      endcase
    end
  end

endmodule
```

Datei: rtl/fram/fram_spi.v

Modul: fram_spi

Kurzbeschreibung: SPI-Steuerung für FRAM-Zugriffe.

Dieses Modul erzeugt eine sequenzielle SPI-Kommunikation mit einem FRAM-Baustein (z. B. MB85RS64V). Es verwendet Opcode-Sequenzen für das Aktivieren der Schreibberechtigung (WREN), das Schreiben (WRITE) und das Lesen (READ). Die Operation wird anhand der Signale `we`, `re`, `address`, `write_data` gestartet. Nach Abschluss meldet das Modul den Zustand mit `done`.

Parameter:

- `CMD_WIDTH` Anzahl der zu shiftenden Bits für WRITE/READ (Adresse + Daten)
- `CMD_WIDTH_WREN` Anzahl der Bits für OPCODE_WREN

Lokale Parameter:

- `OPCODE_WREN` SPI-Opcode zum Aktivieren des Schreibens
- `OPCODE_WRITE` SPI-Opcode zum Schreiben von Daten
- `OPCODE_READ` SPI-Opcode zum Lesen von Daten
- `ST_IDLE` Leerlaufzustand (wartet auf we/re)
- `ST_WREN_INIT` Start der WREN-Sequenz
- `ST_WREN_SHIFT` Shiften der WREN-Bits
- `ST_WREN_DONE` Abschluss der WREN-Operation
- `ST_WRITE_INIT` Start der WRITE-Operation
- `ST_WRITE_SHIFT` Shiften der Schreibdaten
- `ST_WRITE_DONE` Abschluss der Schreiboperation
- `ST_READ_INIT` Start der READ-Operation
- `ST_READ_SHIFT` Shiften der gelesenen Daten
- `ST_READ_DONE` Abschluss der Leseoperation

Eingänge:

- `clk` Systemtakt
- `rst_n` Asynchrones, aktives-LOW Reset
- `[15:0]` address Zieladresse (für WRITE/READ)
- `[31:0]` write_data Zu schreibende 32-Bit-Daten
- `we` Write-Enable (löst WRITE-Sequenz aus)
- `re` Read-Enable (löst READ-Sequenz aus)
- `wire` spi_miso SPI-Datenleitung Slave->Master

Ausgänge:

- `reg` [31:0] read_data Ausgelesene 32-Bit-Daten
- `reg` done Signalisiert Abschluss einer Operation
- `reg` spi_mosi SPI-Datenleitung Master->Slave
- `reg` spi_clk SPI-Takt
- `reg` spi_cs SPI-Chip-Select (aktiv LOW)

Source Code

```
`default_nettype none
`timescale 1ns / 1ns
```

```
module fram_spi (
  input  wire        clk,
  input  wire        rst_n,
  input  wire [15:0] address,
  input  wire [31:0] write_data,
  output reg  [31:0] read_data,
  input  wire        we,
  input  wire        re,
  output reg         done,
  output reg         spi_mosi,
  input  wire        spi_miso,
  output reg         spi_clk,
  output reg         spi_cs
);

  // ----------------------------------------------------------
  // Lokale Parameter (OPCODEs)
  // ----------------------------------------------------------
  localparam [7:0] OPCODE_WREN    = 8'h06;
  localparam [7:0] OPCODE_WRITE   = 8'h02;
  localparam [7:0] OPCODE_READ    = 8'h03;

  // ----------------------------------------------------------
  // Zustände des internen State-Automaten
  // ----------------------------------------------------------
  localparam [3:0] ST_IDLE        = 4'd0;
  localparam [3:0] ST_WREN_INIT   = 4'd1;
  localparam [3:0] ST_WREN_SHIFT  = 4'd2;
  localparam [3:0] ST_WREN_DONE   = 4'd3;
  localparam [3:0] ST_WRITE_INIT  = 4'd4;
  localparam [3:0] ST_WRITE_SHIFT = 4'd5;
  localparam [3:0] ST_WRITE_DONE  = 4'd6;
  localparam [3:0] ST_READ_INIT   = 4'd7;
  localparam [3:0] ST_READ_SHIFT  = 4'd8;
  localparam [3:0] ST_READ_DONE   = 4'd9;

  // ----------------------------------------------------------
  // Anzahl der zu shiftenden Bits:
  // - CMD_WIDTH: 8 (Opcode) + 16 (Adresse) + 32 (Daten) = 56
  // - CMD_WIDTH_WREN: 8 (nur Opcode WREN)
  // ----------------------------------------------------------
  localparam CMD_WIDTH            = 56;
  localparam CMD_WIDTH_WREN       = 8;

  // ----------------------------------------------------------
  // Registervariablen für Shift und Steuerung
  // ----------------------------------------------------------
  reg [55:0] shift_reg;
  reg [ 5:0] bit_count;
  reg [ 3:0] state;
  reg        spi_clk_en;
  reg        shifting;


  // ----------------------------------------------------------
  // SPI-Taktdesign: spi_sck (genannt spi_clk)
  // Wenn spi_clk_en=1, toggelt spi_clk
  // ----------------------------------------------------------
  always @(posedge clk or negedge rst_n)
```

```
begin
  if (!rst_n)
    spi_sck <= 1'b0;
  else
  begin
    if (spi_clk_en)
      spi_sck <= ~spi_sck;
    else
      spi_sck <= 1'b0;
  end
end

// ------------------------------------------------------
// Hauptzustandsmaschine: WREN -> WRITE oder READ
// ------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    state      <= ST_IDLE;
    done       <= 1'b0;
    bit_count  <= 6'd0;
    spi_cs     <= 1'b1;
    spi_clk_en <= 1'b0;
    spi_mosi   <= 1'b0;
    read_data  <= 32'd0;
    shift_reg  <= 56'd0;
    shifting   <= 1'b0;
  end
  else
  begin
    done <= 1'b0;

    case (state)

      // ------------------------------------------------
      // ST_IDLE: Warten auf we (WRITE) oder re (READ)
      // ------------------------------------------------
      ST_IDLE:
      begin
        spi_cs  <= 1'b1;
        spi_clk_en <= 1'b0;
        bit_count  <= 6'd0;
        shifting   <= 1'b0;
        if (write_enable)
        begin
          shift_reg <= {OPCODE_WREN, 48'd0};
          state     <= ST_WREN_INIT;
        end
        else
        if (read_enable)
        begin
          shift_reg <= {OPCODE_READ, addr, 32'd0};
          state     <= ST_READ_INIT;
        end
      end

      // ------------------------------------------------
      // ST_WREN_INIT: CS low, shift_reg bereit für WREN
      // ------------------------------------------------
```

```
ST_WREN_INIT:
begin
  spi_cs  <= 1'b0;
  spi_clk_en <= 1'b0;
  bit_count  <= 6'd0;
  shifting   <= 1'b0;
  spi_mosi   <= shift_reg[55];
  state      <= ST_WREN_SHIFT;
end

// ------------------------------------------------
// ST_WREN_SHIFT: Schiebe 8 Bit des WREN-Opcode raus
// ------------------------------------------------
ST_WREN_SHIFT:
begin
  spi_clk_en <= 1'b1;
  shifting   <= 1'b1;
  if (bit_count == CMD_WIDTH_WREN)
  begin
    spi_cs  <= 1'b1;
    spi_clk_en <= 1'b0;
    shifting   <= 1'b0;
    state      <= ST_WREN_DONE;
  end
end

// ------------------------------------------------
// ST_WREN_DONE: Jetzt WRITE vorbereiten
// ------------------------------------------------
ST_WREN_DONE:
begin
  shift_reg <= {OPCODE_WRITE, addr, write_data};
  state     <= ST_WRITE_INIT;
end

// ------------------------------------------------
// ST_WRITE_INIT: CS wieder runter, neu schieben
// ------------------------------------------------
ST_WRITE_INIT:
begin
  spi_cs  <= 1'b0;
  spi_clk_en <= 1'b0;
  bit_count  <= 6'd0;
  shifting   <= 1'b0;
  spi_mosi   <= shift_reg[55];
  state      <= ST_WRITE_SHIFT;
end

// ------------------------------------------------
// ST_WRITE_SHIFT: 56 Bit (8+16+32) werden geshiftet
// ------------------------------------------------
ST_WRITE_SHIFT:
begin
  spi_clk_en <= 1'b1;
  shifting   <= 1'b1;
  if (bit_count == CMD_WIDTH)
  begin
    spi_cs  <= 1'b1;
    spi_clk_en <= 1'b0;
    shifting   <= 1'b0;
```

```
      state      <= ST_WRITE_DONE;
    end
  end

  // ------------------------------------------------
  // ST_WRITE_DONE: Fertig, done=1
  // ------------------------------------------------
  ST_WRITE_DONE:
  begin
    done  <= 1'b1;
    state <= ST_IDLE;
  end

  // ------------------------------------------------
  // ST_READ_INIT: CS runter, shift_reg bereit (OPCODE+Adress+Dummy)
  // ------------------------------------------------
  ST_READ_INIT:
  begin
    spi_cs  <= 1'b0;
    spi_clk_en <= 1'b0;
    bit_count  <= 6'd0;
    shifting   <= 1'b0;
    spi_mosi   <= shift_reg[55];
    state      <= ST_READ_SHIFT;
  end

  // ------------------------------------------------
  // ST_READ_SHIFT: 56 Bit werden geshiftet, wobei
  //   die letzten 32 Bits Daten vom FRAM sind.
  // ------------------------------------------------
  ST_READ_SHIFT:
  begin
    spi_clk_en <= 1'b1;
    shifting   <= 1'b1;
    if (bit_count == CMD_WIDTH)
    begin
      spi_cs  <= 1'b1;
      spi_clk_en <= 1'b0;
      shifting   <= 1'b0;
      state      <= ST_READ_DONE;
    end
  end

  // ------------------------------------------------
  // ST_READ_DONE: Aus shift_reg die empfangenen 32 Bits
  // ------------------------------------------------
  ST_READ_DONE:
  begin
    read_data <= shift_reg[31:0];
    done <= 1'b1;
    state <= ST_IDLE;
  end

  default: state <= ST_IDLE;
endcase

// ------------------------------------------------------
// Schiebe-Operation
// Bei jeder fallenden Flanke von spi_clk: shift_reg[0] <= spi_miso
// Bei jeder steigenden Flanke: shift_reg << 1
```

```
      // Hier realisiert über spi_sck == 0 / == 1-Abfragen
      // ------------------------------------------------------
      if (shifting)
      begin
        if (spi_sck == 1'b0)
        begin
          shift_reg <= {shift_reg[54:0], 1'b0};
          bit_count <= bit_count + 1;
        end
        else
        begin
          shift_reg[0] <= spi_miso;
        end
      end

      spi_mosi <= shift_reg[55];

    end
  end
endmodule
```

Datei: rtl/fram/mb85rs64v.v

Modul: mb85rs64v

Kurzbeschreibung: Verilog-Modell des FRAM-Bausteins MB85RS64V.

Dieses Modul simuliert den internen SPI-Verhaltensablauf des MB85RS64V-Fram-Speichers. Es reagiert auf die üblichen FRAM-OPCODES (WRITE, READ, WREN) und unterstützt das Speichern von Daten in einem internen Memory-Array. Das Modul ist nur für Simulation und Verifikationszwecke gedacht und ersetzt in der Hardware ein externes FRAM-Bauteil. Zustandsautomat: - STATE_OPCODE: Auswerten des eingehenden Opcodes (WREN, WRITE, READ) - STATE_ADDR: Erfassen der Adresse (2 Bytes) - STATE_WRITE_DATA: Schreiben der ankommenden Bytes in Memory - STATE_READ_DATA: Ausgeben der gewünschten Bytes aus Memory

Lokale Parameter:

- `OP_WRITE` SPI-Opcode für Schreibzugriff
- `OP_READ` SPI-Opcode für Lesezugriff
- `OP_WREN` SPI-Opcode für Write Enable
- `STATE_OPCODE` Zustand zum Erfassen des Opcodes
- `STATE_ADDR` Zustand zum Erfassen der Adresse
- `STATE_WRITE_DATA` Zustand zum Schreiben der Daten ins Memory
- `STATE_READ_DATA` Zustand zum Auslesen der Daten

Eingänge:

- `clk` Systemtakt
- `rst_n` Asynchrones, aktives-LOW Reset
- `spi_mosi` SPI-Datenleitung Master->Slave
- `spi_clk` SPI-Takt
- `spi_cs` SPI-Chip-Select (aktiv LOW)

Ausgänge:

- `reg` spi_miso SPI-Datenleitung Slave->Master

Source Code

```
`default_nettype none
`timescale 1ns / 1ns



module mb85rs64v (
    input  wire clk,
```

```
    input  wire rst_n,
    input  wire spi_mosi,
    output reg  spi_miso,
    input  wire spi_clk,
    input  wire spi_cs
);


  // ------------------------------------------------------
  // Lokale Konstanten / OPCODES
  // ------------------------------------------------------
  localparam OP_WRITE = 8'h02;
  localparam OP_READ  = 8'h03;
  localparam OP_WREN  = 8'h06;

  // ------------------------------------------------------
  // Zustandsdefinitionen
  // ------------------------------------------------------
  localparam STATE_OPCODE     = 2'd0;
  localparam STATE_ADDR       = 2'd1;
  localparam STATE_WRITE_DATA = 2'd2;
  localparam STATE_READ_DATA  = 2'd3;

  // ------------------------------------------------------
  // Internes Memory-Array (z. B. 8k x 8 Bit = 64 KBit)
  // ------------------------------------------------------
  reg [ 7:0] memory[0:8191];

  // ------------------------------------------------------
  // Registervariablen für OPCODE, Adresse, Daten etc.
  // ------------------------------------------------------
  reg [15:0] address;
  reg [15:0] address_shift;
  reg [ 7:0] opcode;
  reg [ 7:0] opcode_shift;
  reg [ 7:0] tx_reg;
  reg [ 7:0] data_shift;
  reg [ 3:0] bit_cnt_tx;
  reg [ 3:0] bit_cnt_rx;
  reg [ 1:0] state;
  reg        spi_clk_prev;
  reg        cs_prev;
  reg        wel;

  // ------------------------------------------------------
  // SPI-Signal-Flankenerkennung
  // - spi_clk_prev, cs_prev merken letzten Zustand
  // ------------------------------------------------------
  always @(posedge clk or negedge rst_n)
  begin
    if (!rst_n)
    begin
      state         <= STATE_OPCODE;
      opcode        <= 8'd0;
      opcode_shift  <= 8'd0;
      address_shift <= 16'd0;
      data_shift    <= 8'd0;
      bit_cnt_rx    <= 4'd0;
      bit_cnt_tx    <= 4'd0;
      tx_reg        <= 8'd0;
```

```
  wel           <= 1'b0;
  spi_miso      <= 1'b0;
  spi_clk_prev  <= 1'b0;
  cs_prev       <= 1'b1;
  address       <= 16'd0;
end
else
begin
  // Merken des vorherigen SPI-Takt- und CS-Zustands
  spi_clk_prev <= spi_clk;
  cs_prev      <= spi_cs;

  // Wenn spi_cs = 1, wird State-Logik zurückgesetzt
  if (spi_cs)
  begin
    state         <= STATE_OPCODE;
    bit_cnt_rx    <= 4'd0;
    bit_cnt_tx    <= 4'd0;
    opcode_shift  <= 8'd0;
    address_shift <= 16'd0;
    data_shift    <= 8'd0;
    address       <= 16'd0;
    // Wenn ein Write-Opcode beendet wird, wel=0
    if (opcode == OP_WRITE)
      wel         <= 1'b0;
  end
  else
  begin
    // Flankenauswertung: steigende Flanke spi_clk
    if (spi_clk && !spi_clk_prev)
    begin
      case (state)


        // ---------------------------------------------
        // STATE_OPCODE: Zunächst 8 Bits für den OPCODE
        // ---------------------------------------------
        STATE_OPCODE:
        begin
          opcode_shift <= {opcode_shift[6:0], spi_mosi};
          bit_cnt_rx   <= bit_cnt_rx + 1;
          if (bit_cnt_rx == 4'd7)
          begin
            opcode       <= {opcode_shift[6:0], spi_mosi};
            bit_cnt_rx   <= 4'd0;
            opcode_shift <= 8'd0;
            if ({opcode_shift[6:0], spi_mosi} == OP_WREN)
              wel <= 1'b1;
            else
              state <= STATE_ADDR;
          end
        end

        // ---------------------------------------------
        // STATE_ADDR: 2 Bytes Adresse (16 Bit)
        // ---------------------------------------------
        STATE_ADDR:
        begin
          address_shift <= {address_shift[14:0], spi_mosi};
          bit_cnt_rx <= bit_cnt_rx + 1;
```

```
  if (bit_cnt_rx == 4'd15)
  begin
    tx_reg     <= memory[{address_shift[14:0], spi_mosi}];
    address    <= {address_shift[14:0], spi_mosi};
    bit_cnt_rx <= 4'd0;
    if (opcode == OP_READ)
    begin
      state      <= STATE_READ_DATA;
      bit_cnt_tx <= 4'd0;
      spi_miso   <= memory[{address_shift[14:0], spi_mosi}][7];
    end
    else
    if (opcode == OP_WRITE)
    begin
      if (wel)
        state <= STATE_WRITE_DATA;
      else
        state <= STATE_OPCODE;
    end
    else
      state <= STATE_OPCODE;
  end
end

// ---------------------------------------------
// STATE_ADDR: 2 Bytes Adresse (16 Bit)
// ---------------------------------------------
STATE_WRITE_DATA:
begin
  if (bit_cnt_rx == 4'd7)
  begin
    memory[(address[12:0])] <= {data_shift[6:0], spi_mosi};
    bit_cnt_rx              <= 4'd0;
    data_shift              <= 8'd0;
    address                 <= address + 1;
  end
  else
  begin
    data_shift <= {data_shift[6:0], spi_mosi};
    bit_cnt_rx <= bit_cnt_rx + 1;
  end
end

// ---------------------------------------------
// STATE_ADDR: 2 Bytes Adresse (16 Bit)
// ---------------------------------------------
STATE_READ_DATA:
begin
  if (bit_cnt_tx == 4'd7)
  begin
    bit_cnt_tx <= 4'd0;
    address    <= address + 1;
    tx_reg     <= memory[address + 1];
    spi_miso   <= memory[(address + 1)][7];
  end
  else
  begin
    spi_miso   <= tx_reg[6];
    tx_reg     <= {tx_reg[6:0], 1'b0};
    bit_cnt_tx <= bit_cnt_tx + 1;
```

```
            end
          end

          default:
            state <= STATE_OPCODE;

        endcase
      end
    end
  end
  end

endmodule
```

Datei: rtl/memory.v

Modul: memory

Kurzbeschreibung: Hauptspeichermodul mit Anbindung an Peripherie.

Dieses Modul verwaltet die Zugriffe der CPU auf den Speicher (RAM oder FRAM, je nach Definition `FRAM_MEMORY`) sowie auf die Peripherie. Dabei wird die Adresse aufgeteilt in einen RAM-Bereich und einen Peripheriebereich. Lese- und Schreibzugriffe im RAM-Bereich erfolgen direkt auf das RAM/FRAM; Zugriffe im Peripheriebereich werden an das `peripheral_bus` Modul weitergereicht. Folgende Bereiche sind definiert: - Adressen >= 0x00004000: RAM/FRAM-Bereich - Adressen < 0x00004000: Peripherie-Bereich (per_addr)

Eingänge:

- `clk` Systemtakt
- `rst_n` Asynchron, aktives-LOW Reset
- `[31:0]` address Adresse des gewünschten Zugriffs
- `[31:0]` write_data Daten, die bei we=1 geschrieben werden
- `we` Write Enable
- `re` Read Enable
- `uart_rx` UART-Eingang
- `spi_miso` SPI Master-In
- `[7:0]` gpio_in GPIO-Eingänge

Ausgänge:

- `[31:0]` read_data Daten, die bei re=1 gelesen werden
- `mem_busy` Signalisiert, ob ein externer Zugriff (z. B. FRAM) noch busy ist
- `uart_tx` UART-Ausgang
- `[31:0]` debug_out Debug-Leitung
- `pwm_out` PWM-Ausgang
- `ws_out` Datenleitung für WS2812B
- `spi_mosi` SPI Master-Out
- `spi_clk` SPI-Takt
- `spi_cs` SPI-Chipselect
- `[7:0]` gpio_out GPIO-Ausgänge
- `[7:0]` gpio_dir GPIO-Richtungsregister

Source Code

```
`include "./defines.v"
`default_nettype none
`timescale 1ns / 1ns



module memory (
  input  wire        clk,
  input  wire        rst_n,
  output wire        mem_busy,
```

```
  input  wire [31:0] address,
  input  wire [31:0] write_data,
  output wire [31:0] read_data,
  input  wire        we,
  input  wire        re,
  output wire        uart_tx,
  input  wire        uart_rx,
  output wire [31:0] debug_out,
  output wire        pwm_out,
  output wire        ws_out,
  output wire        spi_mosi,
  input  wire        spi_miso,
  output wire        spi_clk,
  output wire        spi_cs,
  output wire [ 7:0] gpio_out,
  output wire [ 7:0] gpio_dir,
  input  wire [ 7:0] gpio_in
);

  // ------------------------------------------------------
  // Interne Signale für RAM/FRAM und Peripherie
  // ------------------------------------------------------
  wire [31:0] ram_addr;
  wire [31:0] ram_data;
  wire [13:0] per_addr;
  wire [31:0] per_data;

  wire is_ram;
  wire we_ram;
`ifdef FRAM_MEMORY
  wire re_ram;
`endif
  wire we_per;
  wire re_per;
  wire req_ready;

  // ------------------------------------------------------
  // Lese-/Schreib-Steuerung
  // ------------------------------------------------------
  assign we_ram    = we &&  is_ram;
`ifdef FRAM_MEMORY
  assign re_ram    = re &&  is_ram;
`endif
  assign we_per    = we && !is_ram;
  assign re_per    = re && !is_ram;

  // ------------------------------------------------------
  // Aufteilung der Adressbereiche
  // is_ram = 1, wenn address >= 0x00004000
  // Ansonsten per_addr = address[13:0]
  // ------------------------------------------------------
  assign is_ram    = (address >= 32'h00004000);
  assign ram_addr  = (address  - 32'h00004000);
  assign per_addr  = is_ram ? 14'b0 : address[13:0];
  assign read_data = is_ram ? ram_data : per_data;

  // ------------------------------------------------------
  // mem_busy wird aus req_ready abgeleitet
  // (Bei Normal-RAM meist nicht busy, bei FRAM schon)
  // ------------------------------------------------------
```

```
  assign mem_busy  = !req_ready;
`ifndef FRAM_MEMORY
  assign req_ready = 1'b1;
`endif

  // ------------------------------------------------------------
  // RAM bzw. FRAM-Anbindung
  // ------------------------------------------------------------
`ifdef FRAM_MEMORY
  fram_ram fram_ram_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .req_ready  (req_ready),
    .address    (ram_addr[15:0]),
    .write_data (write_data),
    .read_data  (ram_data),
    .we         (we_ram),
    .re         (re_ram),
    .spi_mosi   (spi_mosi),
    .spi_miso   (spi_miso),
    .spi_clk    (spi_clk),
    .spi_cs     (spi_cs)
  );
`else
  // Normales Single-Port-RAM
  ram1p ram1p_inst (
    .address (ram_addr[14:2]),
    .clock   (clk),
    .data    (write_data),
    .wren    (we_ram),
    .q       (ram_data)
  );
`endif

  // ------------------------------------------------------------
  // Anbindung der Peripheriemodule
  // ------------------------------------------------------------
  peripheral_bus peripheral_bus_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (per_addr),
    .write_data (write_data),
    .read_data  (per_data),
    .we         (we_per),
    .re         (re_per),
    .debug_out  (debug_out),
    .uart_tx    (uart_tx),
    .uart_rx    (uart_rx),
    .pwm_out    (pwm_out),
    .ws_out     (ws_out),
`ifdef FRAM_MEMORY
    .spi_mosi   (),
    .spi_miso   (1'b0),
    .spi_clk    (),
    .spi_cs     (),
`else
    .spi_mosi   (spi_mosi),
    .spi_miso   (spi_miso),
    .spi_clk    (spi_clk),
    .spi_cs     (spi_cs),
```

```
`endif
    .gpio_out   (gpio_out),
    .gpio_dir   (gpio_dir),
    .gpio_in    (gpio_in)
  );

endmodule
```

Datei: rtl/peripherals/debug_module.v

Modul: debug_module

Kurzbeschreibung: Debug-Modul für Host-Kommunikation und Speicherzugriff

Dieses Modul stellt ein einfaches Register zur Verfügung, das über den Bus geschrieben und gelesen werden kann. Es wird außerdem über eine separate Leitung (debug_out) nach außen ausgegeben. Auf jeden Schreibzugriff hin wird der neue Wert im Simulator protokolliert.

Lokale Parameter:

- `DEBUG_ADDR` Basiskonstante zur Adresszuordnung

Eingänge:

- `clk` Systemtakt
- `rst_n` Reset-Signal
- `address` Adresse, über die auf das Debug-Register zugegriffen wird
- `write_data` Zu schreibende Daten
- `we` Write Enable-Signal
- `re` Read Enable-Signal

Ausgänge:

- `read_data` Gelesene Daten aus dem Debug-Register
- `debug_out` Direkte Ausgabe des Debug-Registers (z. B. zu Diagnosezwecken)

Source Code

```
`default_nettype none
`timescale 1ns / 1ns



module debug_module (
    input  wire        clk,
    input  wire        rst_n,
    input  wire [ 7:0] address,
    input  wire [31:0] write_data,
    output wire [31:0] read_data,
    input  wire        we,
    input  wire        re,
    output wire [31:0] debug_out
  );

  // Basiskonstante für Debug-Adressen
  localparam DEBUG_ADDR = 32'h00000000;

  // Internes Register, in dem Debug-Informationen abgelegt werden
  reg [31:0] debug_reg;


  // Zuweisungen: Leseausgabe und Debug-Ausgang geben dasselbe Register aus
  assign read_data = debug_reg;
  assign debug_out = debug_reg;
```

```
  // Speichern oder Zurücksetzen des Debug-Registers
  always @(posedge clk or negedge rst_n)
  begin
    if (!rst_n)
    begin
      debug_reg <= 32'h00000000;
    end
    else
    if (we)
    begin
      debug_reg <= write_data;
      // Meldung im Simulator bei jedem Schreibzugriff
      $display("DEBUG_REG UPDATED: 0x%08X (%d) at time %0t", write_data, write_data, $time);
    end
  end

endmodule
```

Datei: rtl/peripherals/fifo.v

Modul: fifo

Kurzbeschreibung: Ein einfacher FIFO-Speicher.

Dieser FIFO (First-In-First-Out) Puffer speichert Daten mit einer konfigurierbaren Breite (`DATA_WIDTH`) und Tiefe (`DEPTH`). Daten werden über das `wr_en` Signal hineingeschrieben und über das `rd_en` Signal ausgelesen. Das Modul liefert die Signale `empty` und `full`, um anzuzeigen, ob weitere Lese- oder Schreibvorgänge möglich sind.

Parameter:

- `DATA_WIDTH` Breite der gespeicherten Daten in Bits.
- `DEPTH` Anzahl der Einträge im FIFO.

Lokale Parameter:

- `ADDR_WIDTH` Breite der Adresszeiger basierend auf `DEPTH`.

Eingänge:

- `clk` Systemtakt.
- `rst_n` Aktiv-low Reset.
- `wr_en` Aktivierungssignal (Write-Enable) zum Schreiben in den FIFO.
- `rd_en` Aktivierungssignal (Read-Enable) zum Lesen aus dem FIFO.
- `din` Eingangsdaten mit Breite `DATA_WIDTH`.

Ausgänge:

- `empty` Signal, das anzeigt, ob der FIFO leer ist.
- `full` Signal, das anzeigt, ob der FIFO voll ist.
- `dout` Ausgangsdaten mit Breite `DATA_WIDTH`.

Source Code

```
`default_nettype none
`timescale 1ns / 1ns


module fifo #(
  parameter DATA_WIDTH = 8,
  parameter DEPTH      = 16
  ) (
    input  wire                  clk,
    input  wire                  rst_n,
    input  wire                  wr_en,
    input  wire                  rd_en,
    output wire                  empty,
```

```
  output wire                  full,
  input  wire [DATA_WIDTH-1:0] din,
  output wire [DATA_WIDTH-1:0] dout
);

// ------------------------------------------------------
// Abgeleitete Konstanten
// ADDR_WIDTH entspricht der Logarithmusbasis 2 von DEPTH
// und bestimmt die Größe der Lese-/Schreibzeiger
// ------------------------------------------------------
localparam ADDR_WIDTH = $clog2(DEPTH);

// ------------------------------------------------------
// FIFO-internes Speicherarray, sowie Lese- und Schreibzeiger
// ------------------------------------------------------
reg [ADDR_WIDTH  :0]    rd_ptr;
reg [ADDR_WIDTH  :0]    wr_ptr;
reg [DATA_WIDTH-1:0] mem[0:DEPTH-1];

// Edge-Detections für wr_en und rd_en
reg wr_en_prev;
reg rd_en_prev;

// Vorschau auf nächsten Schreibpointer
wire [ADDR_WIDTH: 0] next_wr;

// ------------------------------------------------------
// empty und full Signale
// - empty: wenn Lese- und Schreibzeiger identisch sind
// - full : wenn beide Pointer in Bezug auf MSB unterschiedlich
//   sind, aber in Bezug auf die unteren Bits identisch
// ------------------------------------------------------
assign empty   = (rd_ptr  == wr_ptr);

assign full    = (wr_ptr[ADDR_WIDTH]     != rd_ptr[ADDR_WIDTH]) &&
                 (wr_ptr[ADDR_WIDTH-1:0] == rd_ptr[ADDR_WIDTH-1:0]);

assign next_wr = (wr_ptr   + 1);

// ------------------------------------------------------
// Lese-/Schreiblogik
// ------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    wr_ptr <= 0;
    rd_ptr <= 0;
    wr_en_prev <= 0;
    rd_en_prev <= 0;
  end
  else
  begin
    // Speichern der vorherigen Zustände von wr_en und rd_en
    wr_en_prev <= wr_en;
    rd_en_prev <= rd_en;

    // Schreiben in den FIFO, wenn wr_en ansteigt und nicht full
    if (wr_en && !wr_en_prev && !full)
    begin
```

```
        mem[wr_ptr[ADDR_WIDTH - 1: 0]] <= din;
        wr_ptr <= next_wr;
      end

      // Lesen aus dem FIFO, wenn rd_en ansteigt und nicht empty
      if (rd_en && !rd_en_prev && !empty)
      begin
        rd_ptr <= rd_ptr + 1;
      end
    end
  end

  // ------------------------------------------------------
  // Daten-Ausgabe
  // - Wenn FIFO leer, wird ein Nullwert ausgegeben
  // - Ansonsten wird das Element an der rd_ptr-Position ausgegeben
  // ------------------------------------------------------
  assign dout = empty ? {DATA_WIDTH{1'b0}} : mem[rd_ptr[ADDR_WIDTH - 1 : 0]];

endmodule
```

Datei: rtl/peripherals/gpio.v

Modul: gpio

Kurzbeschreibung: GPIO Modul zur Steuerung allgemeiner I/O.

Dieses Modul implementiert einen einfachen GPIO-Controller. Es unterstützt das Auslesen von Eingangspins und das Schreiben auf Ausgangspins über definierte Adressen. Die Richtungssteuerung (gpio_dir) ist derzeit nicht implementiert.

Lokale Parameter:

- `GPIO_DIR_BASE` Momentan Nicht benutzt (exemplarisch, nicht benutzt)
- `GPIO_IN_OFFSET` Basisadresse (Offset) für das GPIO-Eingangsregister
- `GPIO_OUT_OFFSET` Basisadresse (Offset) für das GPIO-Ausgangsregister
- `GPIO_OUT_STEP` Schrittweite (Abstand) zwischen einzelnen GPIO-Ausgangsregistern

Eingänge:

- `clk` Systemtakt.
- `rst_n` Aktiv-low Reset-Signal.
- `address` Speicheradresse zur Auswahl der GPIO-Register.
- `write_data` Daten, die in die angesprochenen GPIO-Register geschrieben werden.
- `we` Schreibaktivierungssignal (Write-Enable).
- `re` Leseaktivierungssignal (Read-Enable).
- `gpio_in` Eingangssignale von den GPIO-Pins.

Ausgänge:

- `read_data` Ausgangsdaten basierend auf dem angesprochenen GPIO-Register.
- `gpio_out` Register zur Steuerung des Ausgangszustands der GPIO-Pins.
- `gpio_dir` GPIO-Richtungsregister (nicht implementiert).

Source Code

```
`default_nettype none
`timescale 1ns / 1ns


module gpio (
  input  wire        clk,
  input  wire        rst_n,
  input  wire [7:0]  address,
  input  wire [31:0] write_data,
  output wire [31:0] read_data,
```

```
  input  wire        we,
  input  wire        re,
  output reg  [7:0]  gpio_out,
  output reg  [7:0]  gpio_dir,
  input  wire [7:0]  gpio_in
);


  // ------------------------------------------------------
  // Lokale Parameter für Adressbereiche
  // ------------------------------------------------------
  localparam GPIO_DIR_BASE    = 8'h00;
  localparam GPIO_IN_OFFSET   = 8'h04;
  localparam GPIO_OUT_OFFSET  = 8'h08;
  localparam GPIO_OUT_STEP    = 8'h04;

  // ------------------------------------------------------
  // pin_index wird aus der Adresse abgeleitet, um festzustellen,
  // welches Bit im gpio_out-Register angesprochen wird.
  // ------------------------------------------------------
  wire [2:0] pin_index = (address - GPIO_OUT_OFFSET) >> 2;

  // ------------------------------------------------------
  // Lesezugriffe:
  // - Wenn address == GPIO_IN_OFFSET, wird gpio_in zurückgegeben.
  // - Wenn address im Bereich der Ausgangsregister liegt,
  //   wird das entsprechende gpio_out-Bit zurückgegeben.
  // - Andernfalls 0.
  // ------------------------------------------------------
  assign read_data = (address == GPIO_IN_OFFSET) ? {24'd0, gpio_in} :
                     (address >= GPIO_OUT_OFFSET && address < (GPIO_OUT_OFFSET + (8 * GPIO_OUT_STEP)))
                     ? {31'd0, gpio_out[pin_index]} : 32'd0;

  // ------------------------------------------------------
  // Schreiben der gpio_out-Bits:
  // - Beim Reset werden gpio_out und gpio_dir initialisiert.
  // - Nur wenn address im Bereich für Ausgangsregister liegt und we=1,
  //   wird genau ein Bit (write_data[0]) im gpio_out geschrieben.
  // ------------------------------------------------------
  always @(posedge clk or negedge rst_n)
  begin
    if (!rst_n)
    begin
      gpio_out <= 8'd0;
      gpio_dir <= 8'd0;
    end
    else
    if (we && address >= GPIO_OUT_OFFSET && address < (GPIO_OUT_OFFSET + (8 * GPIO_OUT_STEP)))
    begin
      gpio_out[pin_index] <= write_data[0];
    end
  end

endmodule
```

Datei: rtl/peripherals/peripheral_bus.v

Modul: peripheral_bus

Kurzbeschreibung: Zentrales Peripherie-Bus-Modul

Dieses Modul bündelt alle Zugriffe auf die verschiedenen Peripherie-Komponenten (z. B. Debug, UART, Timer, PWM, SPI, GPIO etc.) und leitet Lese-/Schreibanfragen anhand einer Adress-Dekodierung an die entsprechenden Module weiter. Je nach aktivem Define (z. B. `INCLUDE_DEBUG`) wird die jeweilige Peripherie eingebunden oder nicht. Die Schnittstellen zu den einzelnen Modulen sind bereits in diesem Modul angelegt (z. B. uart_tx, gpio_out, pwm_out, ...).

Lokale Parameter:

- `DEBUG_BASE` Basis-Adressenbereich für das Debug-Modul
- `UART_BASE` Basis-Adressenbereich für das UART-Modul
- `TIME_BASE` Basis-Adressenbereich für den System-Timer
- `PWM_BASE` Basis-Adressenbereich für das PWM-Modul
- `MULT_BASE` Basis-Adressenbereich für das Multiplikatormodul
- `DIV_BASE` Basis-Adressenbereich für das Divisionsmodul
- `SPI_BASE` Basis-Adressenbereich für das SPI-Modul
- `GPIO_BASE` Basis-Adressenbereich für das GPIO-Modul
- `WS_BASE` Basis-Adressenbereich für das WS2812B-Modul

Eingänge:

- `clk` Systemtakt
- `rst_n` Asynchrones, aktives-LOW Reset-Signal
- `[13:0]` address Bus-Adresse, aus der das jeweilige Peripheriemodul dekodiert wird
- `[31:0]` write_data Zu schreibende Daten in das ausgewählte Modul
- `we` Write Enable-Signal
- `re` Read Enable-Signal
- `uart_rx` RX-Eingang des UART
- `spi_miso` SPI-Eingangsdaten (Master In, Slave Out)
- `[7:0]` gpio_in Eingangsleitungen des GPIO-Moduls

Ausgänge:

- `reg` [31:0] read_data Gelesene Daten von den Peripheriemodulen
- `[31:0]` debug_out Debug-Ausgangssignal (z. B. zur Diagnose)
- `uart_tx` TX-Ausgang des UART
- `pwm_out` PWM-Ausgangssignal
- `ws_out` Datenleitung für WS2812B-LEDs
- `spi_mosi` SPI-Ausgangsdaten (Master Out, Slave In)
- `spi_clk` SPI-Taktsignal
- `spi_cs` SPI-Chip-Select
- `[7:0]` gpio_out Ausgangssignale des GPIO-Moduls
- `[7:0]` gpio_dir Richtungsregister des GPIO-Moduls

Source Code

```
`include "../defines.v"
`default_nettype none
`timescale 1ns / 1ns



module peripheral_bus (
  input  wire        clk,
  input  wire        rst_n,
  input  wire [13:0] address,
  input  wire [31:0] write_data,
  output reg  [31:0] read_data,
  input  wire        we,
  input  wire        re,
  output wire [31:0] debug_out,
  output wire        uart_tx,
  input  wire        uart_rx,
  output wire        pwm_out,
  output wire        ws_out,
  output wire        spi_mosi,
```

```
  input  wire        spi_miso,
  output wire        spi_clk,
  output wire        spi_cs,
  output wire [ 7:0] gpio_out,
  output wire [ 7:0] gpio_dir,
  input  wire [ 7:0] gpio_in
);

  // Basis-Adressen für die einzelnen Peripherie-Komponenten
  localparam DEBUG_BASE  = 6'h01;
  localparam UART_BASE   = 6'h02;
  localparam TIME_BASE   = 6'h03;
  localparam PWM_BASE    = 6'h04;
  localparam MULT_BASE   = 6'h05;
  localparam DIV_BASE    = 6'h06;
  localparam SPI_BASE    = 6'h07;
  localparam GPIO_BASE   = 6'h08;
  localparam WS_BASE     = 6'h09;

  // Aus der 14-Bit-Adresse wird hier der 8-Bit-Funktionsanteil extrahiert
  // (lower 8 Bits), um innerhalb des Peripheriemoduls weiter zu dekodieren.
  wire [7:0] func_addr;

  assign func_addr = address[7:0];


  // ------------------------------------------------------
  // Debug-Modul (optional über DEFINE eingebunden)
  // ------------------------------------------------------
`ifdef INCLUDE_DEBUG

  wire [31:0] debug_data;

  // Auswahl, ob Adresse in Bereich des Debug-Moduls fällt
  wire debug_sel;
  wire debug_we;
  wire debug_re;

  assign debug_sel  = (address[12:8] == DEBUG_BASE);
  assign debug_we   = we & debug_sel;
  assign debug_re   = re & debug_sel;

  debug_module debug_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (func_addr),
    .write_data (write_data),
    .read_data  (debug_data),
    .we         (debug_we),
    .re         (debug_re),
    .debug_out  (debug_out)
  );
  // Falls nicht eingebunden, werden ungenutzte Signale auf konstante Werte gesetzt
`else
  wire  [31:0]  debug_data = 32'h0;
  wire          debug_sel  = 1'b0;
  assign        debug_out  = 32'h0;
`endif
```

```
  // ------------------------------------------------------------
  // UART-Modul (optional über DEFINE eingebunden)
  // ------------------------------------------------------------
`ifdef INCLUDE_UART

  wire [31:0] uart_data;

  wire uart_sel;
  wire uart_we;
  wire uart_re;

  assign uart_sel  = (address[12:8] == UART_BASE);
  assign uart_we   = we & uart_sel;
  assign uart_re   = re & uart_sel;

  uart #(
    .FIFO_TX_DEPTH(`UART_FIFO_TX_DEPTH),
    .FIFO_RX_DEPTH(`UART_FIFO_RX_DEPTH)
  ) uart_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (func_addr),
    .write_data (write_data),
    .read_data  (uart_data),
    .we         (uart_we),
    .re         (uart_re),
    .uart_tx    (uart_tx),
    .uart_rx    (uart_rx)
  );

`else
  wire  [31:0]  uart_data = 32'h0;
  wire          uart_sel  = 1'b0;
  assign        uart_tx   = 1'b0;
`endif


  // ------------------------------------------------------------
  // System Timer (optional über DEFINE eingebunden)
  // ------------------------------------------------------------
`ifdef INCLUDE_TIME

  wire [31:0] time_data;

  wire time_sel;
  wire time_we;
  wire time_re;

  assign time_sel  = (address[12:8] == TIME_BASE);
  assign time_we   = we & time_sel;
  assign time_re   = re & time_sel;

  system_timer system_timer_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (func_addr),
    .write_data (write_data),
    .read_data  (time_data),
    .we         (time_we),
    .re         (time_re)
```

```
  );

`else
  wire [31:0] time_data = 32'h0;
  wire        time_sel  = 1'b0;
`endif


  // ------------------------------------------------------
  // PWM-Timer (optional über DEFINE eingebunden)
  // ------------------------------------------------------
`ifdef INCLUDE_PWM

  wire [31:0] pwm_data;

  wire pwm_sel;
  wire pwm_we;
  wire pwm_re;

  assign pwm_sel  = (address[12:8] == PWM_BASE);
  assign pwm_we   = we & pwm_sel;
  assign pwm_re   = re & pwm_sel;

  pwm_timer pwm_timer_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (func_addr),
    .write_data (write_data),
    .read_data  (pwm_data),
    .we         (pwm_we),
    .re         (pwm_re),
    .pwm_out    (pwm_out)
  );

`else
  wire [31:0] pwm_data = 32'h0;
  wire        pwm_sel  = 1'b0;
  assign      pwm_out  = 1'b0;
`endif


  // ------------------------------------------------------
  // Sequentieller Multiplikator (optional über DEFINE eingebunden)
  // ------------------------------------------------------
`ifdef INCLUDE_MULT

  wire [31:0] mult_data;

  wire mult_sel;
  wire mult_we;
  wire mult_re;

  assign mult_sel  = (address[12:8] == MULT_BASE);
  assign mult_we   = we & mult_sel;
  assign mult_re   = re & mult_sel;

  seq_multiplier seq_multiplier_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (func_addr),
```

```
    .write_data (write_data),
    .read_data  (mult_data),
    .we         (mult_we),
    .re         (mult_re)
  );

`else
  wire [31:0] mult_data = 32'h0;
  wire        mult_sel  = 1'b0;
`endif


  // ------------------------------------------------------
  // Sequentieller Divider (optional über DEFINE eingebunden)
  // ------------------------------------------------------
`ifdef INCLUDE_DIV

  wire [31:0] div_data;

  wire div_sel;
  wire div_we;
  wire div_re;

  assign div_sel  = (address[12:8] == DIV_BASE);
  assign div_we   = we & div_sel;
  assign div_re   = re & div_sel;

  seq_divider seq_divider_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (func_addr),
    .write_data (write_data),
    .read_data  (div_data),
    .we         (div_we),
    .re         (div_re)
  );

`else
  wire [31:0] div_data = 32'h0;
  wire        div_sel  = 1'b0;
`endif


  // ------------------------------------------------------
  // SPI-Modul (optional über DEFINE eingebunden)
  // ------------------------------------------------------
`ifdef INCLUDE_SPI

  wire [31:0] spi_data;

  wire spi_sel;
  wire spi_we;
  wire spi_re;

  assign spi_sel  = (address[12:8] == SPI_BASE);
  assign spi_we   = we & spi_sel;
  assign spi_re   = re & spi_sel;

  spi #(
    .FIFO_TX_DEPTH(`SPI_FIFO_TX_DEPTH),
```

```
    .FIFO_RX_DEPTH(`SPI_FIFO_RX_DEPTH)
  ) spi_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (func_addr),
    .write_data (write_data),
    .read_data  (spi_data),
    .we         (spi_we),
    .re         (spi_re),
    .spi_clk    (spi_clk),
    .spi_mosi   (spi_mosi),
    .spi_miso   (spi_miso),
    .spi_cs     (spi_cs)
  );

`else
  wire [31:0] spi_data = 32'h0;
  wire        spi_sel  = 1'b0;
  assign      spi_mosi = 1'b0;
  assign      spi_clk  = 1'b0;
  assign      spi_cs   = 1'b1;
`endif


  // ------------------------------------------------------------
  // GPIO-Modul (optional über DEFINE eingebunden)
  // ------------------------------------------------------------
`ifdef INCLUDE_GPIO

  wire [31:0] gpio_data;

  wire gpio_sel;
  wire gpio_we;
  wire gpio_re;

  assign gpio_sel  = (address[12:8] == GPIO_BASE);
  assign gpio_we   = we & gpio_sel;
  assign gpio_re   = re & gpio_sel;

  gpio gpio_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (func_addr),
    .write_data (write_data),
    .read_data  (gpio_data),
    .we         (gpio_we),
    .re         (gpio_re),
    .gpio_out   (gpio_out),
    .gpio_dir   (gpio_dir),
    .gpio_in    (gpio_in)
  );

`else
  wire [31:0]  gpio_data = 32'h0;
  wire         gpio_sel  = 1'b0;
  assign       gpio_out  = 8'b0;
  assign       gpio_dir  = 8'b0;
`endif
```

```
  // ------------------------------------------------------------
  // WS2812B-LED-Modul (optional über DEFINE eingebunden)
  // ------------------------------------------------------------
`ifdef INCLUDE_WS

  wire [31:0] ws_data;

  wire ws_sel;
  wire ws_we;
  wire ws_re;

  assign ws_sel  = (address[12:8] == WS_BASE);
  assign ws_we   = we & ws_sel;
  assign ws_re   = re & ws_sel;

  ws2812b ws2812b_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (func_addr),
    .write_data (write_data),
    .read_data  (ws_data),
    .we         (ws_we),
    .re         (ws_re),
    .ws_out     (ws_out)
  );

`else
  wire [31:0] ws_data = 32'h0;
  wire        ws_sel  = 1'b0;
  assign      ws_out  = 1'b0;
`endif

  // ------------------------------------------------------------
  // Lesezugriffe: Hier wird je nach ausgewähltem Modul das passende
  // 'read_data' ausgegeben. Falls kein passendes Modul selektiert
  // ist, wird 0x0 zurückgegeben.
  // ------------------------------------------------------------
  always @(posedge clk or negedge rst_n)
  begin
    if (!rst_n)
      read_data <= 32'h0;
    else
    if (re)
    begin

`ifdef INCLUDE_DEBUG
      if      (debug_sel) read_data <= debug_data;
`endif
`ifdef INCLUDE_UART
      else if (uart_sel)  read_data <= uart_data;
`endif
`ifdef INCLUDE_TIME
      else if (time_sel)  read_data <= time_data;
`endif
`ifdef INCLUDE_PWM
      else if (pwm_sel)   read_data <= pwm_data;
`endif
`ifdef INCLUDE_MULT
      else if (mult_sel)  read_data <= mult_data;
`endif
```

```
`ifdef INCLUDE_DIV
      else if (div_sel)   read_data <= div_data;
`endif
`ifdef INCLUDE_SPI
      else if (spi_sel)   read_data <= spi_data;
`endif
`ifdef INCLUDE_GPIO
      else if (gpio_sel)  read_data <= gpio_data;
`endif
`ifdef INCLUDE_WS
      else if (ws_sel)    read_data <= ws_data;
`endif

      else
        read_data <= 32'h0;

    end
  end

endmodule
```

Datei: rtl/peripherals/pwm_timer.v

Modul: pwm_timer

Kurzbeschreibung: PWM-Timer zur Erzeugung eines Pulsweitenmodulationssignals

Dieses Modul generiert ein PWM-Signal (Pulsweitenmodulation) mit konfigurierbarer Periode und Pulsbreite. Es wird typischerweise zur Ansteuerung von Motoren, LEDs oder anderen zeitgesteuerten Peripherien eingesetzt. Der Zähler läuft periodisch hoch und vergleicht seinen Wert mit einem Pulsbreitenwert, um das Ausgangssignal zu setzen. Über das Register-Interface lassen sich folgende Werte setzen/auslesen: - `PERIOD_ADDR` : Maximale Zählerperiode (PWM-Zykluslänge). - `DUTY_ADDR` : Vergleichswert für die Pulsbreite (PWM-Anteil). - `COUNTER_ADDR` : Zur Einsicht in den aktuellen Zählerstand (kann auch zurückgesetzt werden). - `CTRL_ADDR` : Steuerungsregister (z. B. Aktivierung, Prescaler).

Lokale Parameter:

- `PERIOD_ADDR` Offset-Adresse für die Perioden-Register.
- `DUTY_ADDR` Offset-Adresse für das Duty-Cycle-Register.
- `COUNTER_ADDR` Offset-Adresse für den Zählerstand.
- `CTRL_ADDR` Offset-Adresse für das Steuerregister.

Eingänge:

- `clk` Systemtakt.
- `rst_n` Asynchroner, aktiver-LOW Reset.
- `[7:0]` address Adressoffset zur Auswahl der Register.
- `[31:0]` write_data Zu schreibende Daten in das ausgewählte Register.
- `we` Write-Enable-Signal.
- `re` Read-Enable-Signal.

Ausgänge:

- `[31:0]` read_data Auszulesende Daten (abhängig von `address`).
- `reg` pwm_out Das generierte PWM-Ausgangssignal.

Source Code

```
`default_nettype none
`timescale 1ns / 1ns



module pwm_timer (
  input  wire        clk,
  input  wire        rst_n,
```

```
  input  wire [7:0]  address,
  input  wire [31:0] write_data,
  output wire [31:0] read_data,
  input  wire        we,
  input  wire        re,
  output reg         pwm_out
);

  // ------------------------------------------------------
  // Lokale Offset-Adressen
  // ------------------------------------------------------
  localparam PERIOD_ADDR  = 8'h00;
  localparam DUTY_ADDR    = 8'h04;
  localparam COUNTER_ADDR = 8'h08;
  localparam CTRL_ADDR    = 8'h0C;

  // ------------------------------------------------------
  // Registervariablen
  // ------------------------------------------------------
  reg [31:0] period;
  reg [31:0] duty;
  reg [31:0] counter;
  reg [31:0] ctrl;
  reg [15:0] pre_count;

  // ------------------------------------------------------
  // Lese-Multiplexer: Gibt je nach address den entsprechenden
  // Registerwert heraus.
  // ------------------------------------------------------
  assign read_data = (address == PERIOD_ADDR)  ? period  :
                     (address == DUTY_ADDR)    ? duty    :
                     (address == COUNTER_ADDR) ? counter :
                     (address == CTRL_ADDR)    ? ctrl    :
                     32'h0;

  // ------------------------------------------------------
  // Schreib- und Zähllogik für period, duty, counter, ctrl
  // ------------------------------------------------------
  always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      period    <= 32'd1000;
      duty      <= 32'd500;
      counter   <= 32'd0;
      ctrl      <= 32'h00010000;
      pre_count <= 16'd0;
    end else begin
      if (we) begin
        // Bei Write-Enable wird abhängig von der address
        // das jeweilige Register überschrieben oder zurückgesetzt.
        case (address)
          PERIOD_ADDR:
            period    <= write_data;
          DUTY_ADDR:
            duty      <= write_data;
          COUNTER_ADDR:
            counter   <= 32'd0;
          CTRL_ADDR:
          begin
            ctrl      <= write_data;
            pre_count <= write_data[31:16];
```

```
          end
          default: ;
        endcase
      end else
      if (pre_count > 0) begin
        // pre_count wird herabgesetzt, wenn > 0
        pre_count <= pre_count - 1;
      end else begin
        // Wenn Vorzähler abgelaufen ist, wird counter inkrementiert
        pre_count <= ctrl[31:16];

        if (counter >= period - 1) begin
          counter <= 32'd0;
        end else begin
          counter <= counter + 1;
        end
      end
    end
  end

  // ------------------------------------------------------
  // PWM-Ausgabe:
  // - Ist das Bit ctrl[0] gesetzt, wird PWM aktiviert.
  // - ctrl[1] kann (je nach Nutzung) z. B. eine Art Modus
  //   für ein symmetrisches PWM sein (hier nur beispielhaft).
  // ------------------------------------------------------
  always @(posedge clk or negedge rst_n)
  begin
    if (!rst_n)
      pwm_out <= 1'b0;
    else begin
      if (ctrl[0]) begin
        if (ctrl[1])
          pwm_out <= (counter < (period >> 1)) ? 1'b1 : 1'b0;
        else
          pwm_out <= (counter < duty) ? 1'b1 : 1'b0;
      end else
        pwm_out <= 1'b0;
    end
  end

endmodule
```

Datei: rtl/peripherals/seq_divider.v

Modul: seq_divider

Kurzbeschreibung: Sequentieller Divider (Divisionseinheit).

Dieses Modul führt eine sequentielle Division zweier 32-Bit-Werte durch. Dabei wird fortlaufend ein Teil des Dividenden mit dem Divisor verglichen und angepasst. Das Modul ist über ein Register-Interface ansprechbar: - `INFO_OFFSET`: Enthält das `busy`-Bit (Division noch aktiv). - `END_OFFSET` : Speicherort für den Dividenden (Endwert). - `SOR_OFFSET` : Speicherort für den Divisor (Startet die Division). - `QUO_OFFSET` : Liefert das Ergebnis (Quotient). - `REM_OFFSET` : Liefert den Restwert (Remainder).

Lokale Parameter:

- `INFO_OFFSET` Offset für das Statusregister (busy).
- `END_OFFSET` Offset für den Dividenden.
- `SOR_OFFSET` Offset für den Divisor (startet Division).
- `QUO_OFFSET` Offset für den Quotienten.
- `REM_OFFSET` Offset für den Rest.

Eingänge:

- `clk` Systemtakt.
- `rst_n` Aktiv-low Reset.
- `[7:0]` address Adresse für den Registerzugriff.
- `[31:0]` write_data Daten, die z. B. Dividenden/Divisor setzen.
- `we` Schreibaktivierungssignal (Write-Enable).
- `re` Leseaktivierungssignal (Read-Enable).

Ausgänge:

- `[31:0]` read_data Ausgabedaten basierend auf address.

Source Code

```
`default_nettype none
`timescale 1ns / 1ns



module seq_divider (
  input  wire        clk,
  input  wire        rst_n,
  input  wire [ 7:0] address,
  input  wire [31:0] write_data,
  output wire [31:0] read_data,
  input  wire        we,
  input  wire        re
);


  // ----------------------------------------------------------
  // Lokale Adress-Offsets für das Register-Interface
  // ----------------------------------------------------------
  localparam INFO_OFFSET = 8'h00;
  localparam END_OFFSET  = 8'h04;
  localparam SOR_OFFSET  = 8'h08;
  localparam QUO_OFFSET  = 8'h0C;
  localparam REM_OFFSET  = 8'h10;

  // ----------------------------------------------------------
  // Register für Dividenden, Divisor, Quotient und Rest
  // dvdend_tmp dient als Arbeitsregister (64 Bit),
  // um den Dividenden hochzuschieben und den Rest auszuarbeiten.
  // ----------------------------------------------------------
  reg [31:0] dividend; // Gespeicherter Dividendenwert
  reg [31:0] divisor;  // Gespeicherter Divisor
  reg [63:0] dvdend_tmp; // Temporäre 64-Bit-Repräsentation des Dividenden
  reg [31:0] quotient;  // Quotient
  reg [31:0] remainder; // Rest
  reg [ 5:0] bit_index; // Zähler für 32-Bit schrittweise Division
  reg        busy;  // Signalisiert laufende Division

  // ----------------------------------------------------------
  // Lesezugriffe: abhängig vom Offset wird das passende
  // Register oder das busy-Bit zurückgegeben.
  // ----------------------------------------------------------
  assign read_data = (address == INFO_OFFSET) ? {31'd0, busy} :
                     (address == END_OFFSET)  ?  dividend     :
                     (address == SOR_OFFSET)  ?  divisor      :
                     (address == QUO_OFFSET)  ?  quotient     :
                     (address == REM_OFFSET)  ?  remainder    :
```

```
                    32'd0;


// ------------------------------------------------------
// Sequentieller Ablauf:
// - Schreiben von dividend und divisor startet Berechnung.
// - Der Divisor wird sukzessive vom oberen Teil des Dividenden
//   subtrahiert (wenn möglich), das Ergebnis fließt in quotient.
// ------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    dividend   <= 32'd0;
    divisor    <= 32'd0;
    quotient   <= 32'd0;
    remainder  <= 32'd0;
    dvdend_tmp <= 64'd0;
    bit_index  <= 6'd0;
    busy       <= 1'b0;
  end
  else
  begin
    // Verarbeiten von Write-Zugriffen
    if (we)
    begin
      case (address)

        END_OFFSET:
        begin
          // Setzt den Dividenden und initialisiert das Arbeitsregister
          dividend   <= write_data;
          dvdend_tmp <= {32'd0, write_data};
          quotient   <= 32'd0;
          remainder  <= 32'd0;
        end

        SOR_OFFSET:
        begin
          // Setzt den Divisor und aktiviert die Division
          divisor    <= write_data;
          bit_index  <= 6'd31;
          busy       <= 1'b1;
        end

      endcase
    end


    // Division in kleinen Schritten, solange busy=1
    if (busy)
    begin
      // Jede Iteration:
      // 1) Schiebe das 64-Bit-Fenster nach links (dvdend_tmp << 1),
      // 2) Schiebe quotient nach links,
      // 3) teste, ob Divisor subtrahierbar ist.
      dvdend_tmp <= dvdend_tmp << 1;
      quotient   <= quotient << 1;

      // Prüfe den oberen 32-Bit-Teil gegen divisor
```

```
        if (dvdend_tmp[63:32] >= divisor)
        begin
          dvdend_tmp[63:32] <= dvdend_tmp[63:32] - divisor;
          quotient[0] = 1;
        end

        // Sobald alle Bits bearbeitet sind, legen wir remainder fest.
        if (bit_index == 0)
        begin
          remainder <= dvdend_tmp[63:32];
          busy <= 1'b0;
        end
        else
        begin
          bit_index <= bit_index - 1;
        end
      end
    end
  end

endmodule
```

Datei: rtl/peripherals/seq_multiplier.v

Modul: seq_multiplier

Kurzbeschreibung: Sequentieller Multiplikator.

Dieses Modul implementiert eine sequentielle Multiplikation zweier 32-Bit-Werte. Die Multiplikation erfolgt durch aufeinanderfolgende Additionen basierend auf den Bits des Multiplikators. Das Modul ist speicherabbildbasiert und kann über Adressen gesteuert werden: - `MUL1_OFFSET`: Erster Multiplikand. - `MUL2_OFFSET`: Zweiter Multiplikator (startet die Berechnung). - `RESH_OFFSET`: Höhere 32 Bits des Ergebnisses. - `RESL_OFFSET`: Niedrigere 32 Bits des Ergebnisses. - `INFO_OFFSET`: Gibt an, ob die Berechnung noch läuft (`busy`).

Lokale Parameter:

- `INFO_OFFSET` Adresse für den Status (`busy`-Bit).
- `MUL1_OFFSET` Adresse für den ersten Multiplikanden.
- `MUL2_OFFSET` Adresse für den zweiten Multiplikator (startet Berechnung).
- `RESH_OFFSET` Adresse für die höheren 32 Bit des Ergebnisses.
- `RESL_OFFSET` Adresse für die niedrigeren 32 Bit des Ergebnisses.

Eingänge:

- `clk` Systemtakt.
- `rst_n` Aktiv-low Reset.
- `address` Speicheradresse für den Zugriff auf Register.
- `write_data` Daten, die in die ausgewählten Register geschrieben werden sollen.
- `we` Schreibaktivierungssignal (Write-Enable).
- `re` Leseaktivierungssignal (Read-Enable).

Ausgänge:

- `read_data` Zu lesende Daten basierend auf der Adresse.

Source Code

```
`default_nettype none
`timescale 1ns / 1ns


module seq_multiplier (
  input  wire        clk,
  input  wire        rst_n,
  input  wire [ 7:0] address,
```

```
  input  wire [31:0] write_data,
  output wire [31:0] read_data,
  input  wire        we,
  input  wire        re
);

  // ------------------------------------------------------
  // Lokale Adress-Offsets für die Register
  // ------------------------------------------------------
  localparam INFO_OFFSET = 8'h00;
  localparam MUL1_OFFSET = 8'h04;
  localparam MUL2_OFFSET = 8'h08;
  localparam RESH_OFFSET = 8'h0C;
  localparam RESL_OFFSET = 8'h10;

  // ------------------------------------------------------
  // Register für Multiplikanden, Ergebnis und Steuerung
  // ------------------------------------------------------
  reg [31:0] multiplicand;
  reg [31:0] multiplier;
  reg [63:0] product;
  reg [ 5:0] bit_index;
  reg        busy;

  // ------------------------------------------------------
  // Leseausgabe: Je nach Adress-Offset wird entsprechender
  // Registerinhalt oder Status zurückgegeben.
  // ------------------------------------------------------
  assign read_data = (address == INFO_OFFSET) ? {31'd0, busy} :
                     (address == MUL1_OFFSET) ? multiplicand  :
                     (address == MUL2_OFFSET) ? multiplier    :
                     (address == RESH_OFFSET) ? product[63:32]:
                     (address == RESL_OFFSET) ? product[31: 0]:
                     32'd0;

  // ------------------------------------------------------
  // Sequentielle Abarbeitung der Multiplikation
  // ------------------------------------------------------
  always @(posedge clk or negedge rst_n)
  begin
    if (!rst_n) begin
      // Initialisierung bei Reset
      multiplicand <= 32'd0;
      multiplier   <= 32'd0;
      product      <= 64'd0;
      bit_index    <= 6'd0;
      busy         <= 1'b0;
    end
    else
    begin
      // Behandlung von Bus-Schreibzugriffen
      if (we)
      begin
        case (address)
          MUL1_OFFSET: begin
            // Schreiben des ersten Multiplikanden
            multiplicand <= write_data;
          end
          MUL2_OFFSET: begin
            // Schreiben des zweiten Multiplikators -> Start der Multiplikation
```

```
            multiplier <= write_data;
            product    <= 64'd0;
            bit_index  <= 6'd31;
            busy       <= 1'b1;
          end
        endcase
      end

      // Wenn busy = 1, läuft die sequentielle Multiplikation
      if (busy)
      begin
        if (multiplier[bit_index])
        begin
          product <= product + ((64'd1 << bit_index) * multiplicand);
        end

        // Bitzähler reduzieren
        if (bit_index == 0)
        begin
          // Sobald alle Bits durch sind, ist die Multiplikation abgeschlossen
          busy <= 1'b0;
        end
        else
        begin
          bit_index <= bit_index - 1;
        end
      end
    end
  end

endmodule
```

Datei: rtl/peripherals/spi.v

Modul: spi

Kurzbeschreibung: SPI-Peripheriemodul mit internen FIFO-Puffern.

Dieses Modul stellt ein SPI-Master-Interface bereit (MOSI/MISO/CLK/CS), mit jeweils einem TX- und RX-FIFO zur gepufferten Übertragung von Datenpaketen. Die Baudrate wird über ein konfigurierbares Clock-Divider- Register gesteuert. Zusätzlich kann zwischen automatischer und manueller Steuerung des CS-Signals gewählt werden. Register-Offsets innerhalb des Moduls: - `CTRL_OFFSET` : Steuerregister (Aktivierung, CS-Gen, etc.) - `CLK_OFFSET` : Clock-Divider-Register (zur SPI-Takterzeugung) - `STATUS_OFFSET` : Statusbits (z. B. Busy, FIFO-Zustände) - `TX_OFFSET` : TX-FIFO-Schreibregister - `RX_OFFSET` : RX-FIFO-Leseregister - `CS_OFFSET` : Manuelle Chip-Select-Steuerung

Parameter:

- `FIFO_TX_DEPTH` = 8 Anzahl der Einträge im FIFO für TX
- `FIFO_RX_DEPTH` = 8 Anzahl der Einträge im FIFO für RX

Lokale Parameter:

- `CTRL_OFFSET` Offset für das Steuerregister
- `CLK_OFFSET` Offset für den Baudratenteiler
- `STATUS_OFFSET` Offset für das Statusregister
- `TX_OFFSET` Offset zum Schreiben in den TX-FIFO
- `RX_OFFSET` Offset zum Lesen aus dem RX-FIFO
- `CS_OFFSET` Offset für manuelle Chip-Select-Steuerung
- `STATE_IDLE` Leerlaufzustand
- `STATE_LOAD` Daten werden aus dem TX-FIFO übernommen
- `STATE_LOAD_WAIT` Wartezyklus nach dem Laden
- `STATE_TRANSFER` Aktive SPI-Datenübertragung

Eingänge:

- `clk` Systemtakt
- `rst_n` Asynchroner, aktiver-LOW Reset
- `[7:0]` address Auswahl des Registers innerhalb des SPI-Moduls
- `[31:0]` write_data Daten, die in ein ausgewähltes Register geschrieben werden
- `we` Write-Enable-Signal
- `re` Read-Enable-Signal
- `wire` spi_miso SPI-Daten-Eingang (Master In, Slave Out)

Ausgänge:

- `[31:0]` read_data Ausgelesener Wert aus dem entsprechenden Register
- `reg` spi_clk SPI-Clock-Ausgang
- `reg` spi_mosi SPI-Daten-Ausgang (Master Out, Slave In)
- `wire` spi_cs SPI-Chip-Select (automatisch oder manuell)

Source Code

```
`default_nettype none
`timescale 1ns / 1ns



module spi #(
  parameter FIFO_TX_DEPTH = 8,
  parameter FIFO_RX_DEPTH = 8
) (
  input  wire        clk,
  input  wire        rst_n,
  input  wire [ 7:0] address,
  input  wire [31:0] write_data,
  output wire [31:0] read_data,
  input  wire        we,
  input  wire        re,
  output reg         spi_clk,
  output reg         spi_mosi,
  input  wire        spi_miso,
  output wire        spi_cs
);

  // ------------------------------------------------------
  // Register-Offsets innerhalb des Adressraums
  // ------------------------------------------------------
  localparam CTRL_OFFSET     = 8'h00;
  localparam CLK_OFFSET      = 8'h04;
  localparam STATUS_OFFSET   = 8'h08;
  localparam TX_OFFSET       = 8'h0C;
  localparam RX_OFFSET       = 8'h10;
  localparam CS_OFFSET       = 8'h14;


  // ------------------------------------------------------
  // Zustände des internen State-Automaten
  // ------------------------------------------------------
  localparam STATE_IDLE      = 2'd0;
  localparam STATE_LOAD      = 2'd1;
  localparam STATE_LOAD_WAIT = 2'd2;
  localparam STATE_TRANSFER  = 2'd3;

  // ------------------------------------------------------
  // Registervariablen und -signale
  // ------------------------------------------------------
  reg [15:0] spi_clk_div;
```

```
reg [15:0] new_spi_clk_div;
reg [16:0] clk_counter;
reg [ 7:0] tx_shift;
reg [ 7:0] rx_shift;
reg [ 7:0] rx_fifo_din;
reg [ 1:0] state;
reg [ 3:0] bit_cnt;
reg [ 1:0] spi_ctrl;
reg tx_fifo_rd_en;
reg rx_fifo_wr_en;
reg rx_fifo_rd_en;
reg read_flag;
reg spi_clk_en;
reg active;
reg cs;
reg cs_gen;
reg cs_manual;
reg cs_manual_next;

// ------------------------------------------------------
// FIFOs für TX und RX
// ------------------------------------------------------
wire [8:0] tx_fifo_din;
wire [8:0] tx_fifo_dout;
wire [7:0] rx_fifo_dout;
wire [7:0] status_bits;
wire clk_div_zero;
wire spi_busy;
wire spi_ready;
wire tx_fifo_wr_en;
wire tx_fifo_empty;
wire tx_fifo_full;
wire rx_fifo_empty;
wire rx_fifo_full;
wire fifo_full;


// ------------------------------------------------------
// Externe Zuweisungen
// ------------------------------------------------------
assign clk_div_zero     = (~|spi_clk_div);
assign spi_busy         = (state != STATE_IDLE);
assign fifo_full        = rx_fifo_full | tx_fifo_full;
assign spi_ready        = !spi_busy & tx_fifo_empty & rx_fifo_empty;
assign status_bits      = {clk_div_zero, fifo_full, spi_ready, spi_busy, rx_fifo_full, rx_fifo_empty, tx_fif
assign tx_fifo_din[8:0] = write_data[8:0];
assign tx_fifo_wr_en    = we && (address[7:0] == TX_OFFSET);
assign spi_cs           = cs_gen ? cs : cs_manual;

// ------------------------------------------------------
// TX-FIFO Instanz
// ------------------------------------------------------
fifo #(
  .DATA_WIDTH (9),
  .DEPTH      (FIFO_TX_DEPTH)
) spi_tx_fifo (
  .clk   (clk),
  .rst_n (rst_n),
  .wr_en (tx_fifo_wr_en),
  .rd_en (tx_fifo_rd_en),
```

```
  .din   (tx_fifo_din),
  .dout  (tx_fifo_dout),
  .empty (tx_fifo_empty),
  .full  (tx_fifo_full)
);

// ------------------------------------------------------
// RX-FIFO Instanz
// ------------------------------------------------------
fifo #(
  .DATA_WIDTH (8),
  .DEPTH      (FIFO_RX_DEPTH)
) spi_rx_fifo (
  .clk   (clk),
  .rst_n (rst_n),
  .wr_en (rx_fifo_wr_en),
  .rd_en (rx_fifo_rd_en),
  .din   (rx_fifo_din),
  .dout  (rx_fifo_dout),
  .empty (rx_fifo_empty),
  .full  (rx_fifo_full)
);


// ------------------------------------------------------
// Lesen aus SPI-Registern (CTRL, CLK, STATUS, RX, CS)
// ------------------------------------------------------
assign read_data = (address[7:0] == CTRL_OFFSET)   ? {30'd0, cs_gen, active} :
                   (address[7:0] == CLK_OFFSET)    ? {16'd0,    spi_clk_div} :
                   (address[7:0] == STATUS_OFFSET) ? {24'd0,    status_bits} :
                   (address[7:0] == RX_OFFSET)     ? {24'd0,   rx_fifo_dout} :
                   (address[7:0] == CS_OFFSET)     ? {31'd0,      cs_manual} :
                   32'd0;

// ------------------------------------------------------
// Clock-Divider-Logik: Erzeugung von spi_clk
// ------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    clk_counter <= 16'b0;
    spi_clk_en <= 1'b0;
  end
  else
  begin

    if (state == STATE_LOAD)
    begin
      clk_counter <= 16'b1;
      spi_clk_en <= 1'b0;

    end
    else if (clk_counter >= {spi_clk_div[15: 0], ~|spi_clk_div})
    begin
      clk_counter <= 16'b1;
      spi_clk_en <= 1'b1;

    end
    else
```

```
    begin
      clk_counter <= clk_counter + 16'b1;
      spi_clk_en <= 1'b0;
    end

  end
end

// ----------------------------------------------------------
// Zustandsmaschine für SPI
// - Übergänge zwischen Ladephase, Übertragung etc.
// - Steuert spi_clk, mosi, bit_cnt, etc.
// ----------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    state         <= STATE_IDLE;
    cs            <= 1'b1;
    cs_manual     <= 1'b1;
    spi_clk       <= 1'b0;
    bit_cnt       <= 4'b0;
    tx_shift      <= 8'b0;
    rx_shift      <= 8'b0;
    read_flag     <= 1'b0;
    tx_fifo_rd_en <= 1'b0;
    rx_fifo_wr_en <= 1'b0;
    spi_mosi      <= 1'b0;
  end
  else
  begin
    // Standard: Deaktivierung von FIFO-Lese/Schreib-Enables
    tx_fifo_rd_en <= 1'b0;
    rx_fifo_wr_en <= 1'b0;

    case (state)

      STATE_IDLE:
      begin
        cs_manual <= cs_manual_next;
        cs      <= 1'b1;
        spi_clk <= 1'b0;
        bit_cnt <= 4'b0;

        if (active && !tx_fifo_empty)
        begin
          cs            <= 1'b0;
          tx_shift      <= tx_fifo_dout[7:0];
          tx_fifo_rd_en <= 1'b0;
          state         <= STATE_LOAD;
        end
      end

      STATE_LOAD:
      begin
        tx_fifo_rd_en <= 1'b1;
        state         <= STATE_LOAD_WAIT;
        cs            <= 1'b0;
      end
```

```
STATE_LOAD_WAIT:
begin
  tx_shift      <= tx_fifo_dout[7:0];
  read_flag     <= tx_fifo_dout[8];
  tx_fifo_rd_en <= 1'b0;
  rx_shift      <= 8'd0;
  bit_cnt       <= 4'b0;
  spi_clk       <= 1'b0;
  spi_mosi      <= tx_fifo_dout[7];
  state         <= STATE_TRANSFER;
end

STATE_TRANSFER:
begin
  if (spi_clk_en)
  begin
    if (spi_clk == 0)
    begin
      spi_clk <= 1'b1;

      if (bit_cnt <= 3'd7)
      begin
        rx_shift[3'd7 - bit_cnt[2:0]] <= spi_miso;
      end
    end
    else
    begin
      spi_clk <= 1'b0;

      if (bit_cnt < 3'd7)
      begin
        bit_cnt  <= bit_cnt + 4'b1;
        spi_mosi <= tx_shift[3'd6 - bit_cnt[2: 0]];
      end
      else
      begin
        if (read_flag)
        begin
          rx_fifo_din   <= rx_shift;
          rx_fifo_wr_en <= 1'b1;
        end

        if (!tx_fifo_empty)
        begin
          state <= STATE_LOAD;
          //tx_fifo_rd_en <= 1'b1;
        end
        else
        begin
          cs     <= 1'b1;
          state  <= STATE_IDLE;
        end
      end
    end
  end
  else
    state <= STATE_TRANSFER;
end

default:
```

```
          state <= STATE_IDLE;

      endcase
    end
  end

  // ------------------------------------------------------
  // Kontrolle der SPI-Einstellungen (CTRL, CLK, CS)
  // - Hier wird z. B. active und cs_gen gesetzt, sowie
  //   der Taktteiler aktualisiert
  // ------------------------------------------------------
  always @(posedge clk or negedge rst_n)
  begin
    if (!rst_n)
    begin
      active          <= 1'b1;
      cs_manual_next  <= 1'b1;
      cs_gen          <= 1'b1;
      spi_clk_div     <= 16'd0;
      new_spi_clk_div <= 16'd0;
    end
    else if (we)
    begin

      case (address[7:0])

        CTRL_OFFSET:
        begin
          active   <= write_data[0];
          cs_gen   <= write_data[1];
        end

        CLK_OFFSET:
        begin
          new_spi_clk_div <= write_data;
        end

        CS_OFFSET:
        begin
          cs_manual_next <= ~write_data[0];
        end

        default : ;

      endcase

    end
    else if (spi_ready)
      spi_clk_div <= new_spi_clk_div;
  end

endmodule
```

Datei: rtl/peripherals/system_timer.v

Modul: system_timer

Kurzbeschreibung: System Timer

Dieses Modul stellt einen System-Timer bereit, der sowohl Millisekunden- als auch Mikrosekunden-Zähler verwaltet. Zwei Zähler (ms_counter, mik_counter) laufen in Abhängigkeit von `CLK_FREQ`, wodurch sich systemweite Zeitstempel erzeugen

lassen. Über Register kann der Zählerinhalt ausgelesen oder zurückgesetzt werden. Register-Offsets: - `MS_L_OFFSET` : Niedrigere 32 Bit des Millisekunden-Zählers - `MS_H_OFFSET` : Höhere 32 Bit des Millisekunden-Zählers - `MIK_L_OFFSET` : Niedrigere 32 Bit des Mikrosekunden-Zählers - `MIK_H_OFFSET` : Höhere 32 Bit des Mikrosekunden-Zählers - `SYS_CLOCK` : Liefert die aktuelle Taktrate in Hz (niedrigere 32 Bit)

Lokale Parameter:

- `MS_COUNT_LIMIT` Obergrenze für 1 ms (basierend auf `CLK_FREQ`)
- `MIK_COUNT_LIMIT` Obergrenze für 1 µs (basierend auf `CLK_FREQ`)
- `MS_COUNTER_WIDTH` Breite des Zählers für ms_counter
- `MIK_COUNTER_WIDTH` Breite des Zählers für mik_counter
- `MS_L_OFFSET` Registeroffset für Millisekunden (Low)
- `MS_H_OFFSET` Registeroffset für Millisekunden (High)
- `MIK_L_OFFSET` Registeroffset für Mikrosekunden (Low)
- `MIK_H_OFFSET` Registeroffset für Mikrosekunden (High)
- `SYS_CLOCK` Registeroffset für das Auslesen der Taktfrequenz

Eingänge:

- `clk` Systemtakt.
- `rst_n` Asynchroner, aktiver-LOW Reset.
- `[7:0]` address Busadresse für das Lesen/Schreiben der Timerregister.
- `[31:0]` write_data Zu schreibende Daten (z. B. zum Zurücksetzen).
- `we` Write-Enable.
- `re` Read-Enable.

Ausgänge:

- `[31:0]` read_data Enthält gelesene Daten abhängig von `address`.

Source Code

```
`include "../defines.v"
`default_nettype none
`timescale 1ns / 1ns



module system_timer (
  input  wire        clk,
  input  wire        rst_n,
  input  wire [ 7:0] address,
  input  wire [31:0] write_data,
  output wire [31:0] read_data,
  input  wire        we,
  input  wire        re
);

    localparam MS_COUNT_LIMIT    = (`CLK_FREQ / 1000)    - 1;
  localparam MIK_COUNT_LIMIT   = (`CLK_FREQ / 1000000) - 1;
    localparam MS_COUNTER_WIDTH  = $clog2(MS_COUNT_LIMIT + 1);
  localparam MIK_COUNTER_WIDTH = $clog2(MIK_COUNT_LIMIT + 1);

  localparam MSW = MS_COUNTER_WIDTH;
  localparam MKW = MIK_COUNTER_WIDTH;

  localparam MS_L_OFFSET   = 8'h00;
  localparam MS_H_OFFSET   = 8'h04;
  localparam MIK_L_OFFSET  = 8'h08;
  localparam MIK_H_OFFSET  = 8'h0C;
  localparam SYS_CLOCK     = 8'h10;

  // ------------------------------------------------------
  // Timer-Register:
```

```
// - ms_counter  + sys_tim_ms: für Millisekunden
// - mik_counter + sys_tim_mik: für Mikrosekunden
// ------------------------------------------------------
reg [MSW-1:0] ms_counter;
reg [MKW-1:0] mik_counter;
reg [ 63  :0] sys_tim_ms;
reg [ 63  :0] sys_tim_mik;

wire [31:0] sys_clk;

assign sys_clk = `CLK_FREQ;

// ------------------------------------------------------
// Leseauswahl je nach address:
// ------------------------------------------------------
assign read_data = (address == MS_L_OFFSET)  ? sys_tim_ms [31: 0] :
                   (address == MS_H_OFFSET)  ? sys_tim_ms [63:32] :
                   (address == MIK_L_OFFSET) ? sys_tim_mik[31: 0] :
                   (address == MIK_H_OFFSET) ? sys_tim_mik[63:32] :
                   (address == SYS_CLOCK)    ? sys_clk    [31: 0] :
                   32'd0;

// ------------------------------------------------------
// Millisekunden-Zähler
// - ms_counter zählt bis MS_COUNT_LIMIT
// - Wird der Wert erreicht, dann sys_tim_ms++
// - Auf Write an MS_L_OFFSET wird Timer zurückgesetzt
// ------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    sys_tim_ms <= 64'd0;
    ms_counter <= {MSW{1'b0}};
  end
  else
  begin
    if(we && (address == MS_L_OFFSET))
    begin
      sys_tim_ms   <= 64'd0;
      ms_counter   <= {MSW{1'b0}};
    end
    else if (ms_counter == MS_COUNT_LIMIT)
    begin
      sys_tim_ms <= sys_tim_ms + 1;
      ms_counter <= {MSW{1'b0}};
    end
    else
    begin
      ms_counter <= ms_counter + 1;
    end
  end
end

// ------------------------------------------------------
// Mikrosekunden-Zähler
// - mik_counter zählt bis MIK_COUNT_LIMIT
// - Wird der Wert erreicht, dann sys_tim_mik++
// - Auf Write an MIK_L_OFFSET wird Timer zurückgesetzt
// ------------------------------------------------------
```

```
  always @(posedge clk or negedge rst_n)
  begin
    if (!rst_n)
    begin
      sys_tim_mik   <= 64'd0;
      mik_counter   <= {MKW{1'b0}};
    end
    else
    begin
      if(we && (address == MIK_L_OFFSET))
      begin
        sys_tim_mik   <= 64'd0;
        mik_counter   <= {MKW{1'b0}};
      end
      else if (mik_counter == MIK_COUNT_LIMIT)
      begin
        sys_tim_mik <= sys_tim_mik + 1;
        mik_counter <= {MKW{1'b0}};
      end
      else
      begin
        mik_counter <= mik_counter + 1;
      end
    end
  end

endmodule
```

Datei: rtl/peripherals/uart.v

Modul: uart

Kurzbeschreibung: UART-Modul mit TX-/RX-FIFOs

Dieses Modul implementiert eine einfache serielle Schnittstelle (UART). Es enthält interne FIFOs (TX und RX), die in ihrer Tiefe per Parameter festgelegt werden können. Die Baudrate kann über das Baudraten-Register gewählt werden, wobei mehrere vordefinierte Optionen existieren (z. B. 115200, 57600). Register-Offsets innerhalb des UART-Moduls: - `CTRL_OFFSET` : Steuerregister (z. B. Aktivierung). - `BAUD_OFFSET` : Wahl der Baudraten-Voreinstellung (4-Bit: 0..11). - `STATUS_OFFSET` : Statusbits (z. B. FIFO-Zustand, Busy). - `TX_OFFSET` : Schreiben in die TX-FIFO. - `RX_OFFSET` : Lesen aus der RX-FIFO.

Lokale Parameter:

- `BAUD_DIV_115200` ... BAUD_DIV_300 : Vordefinierte Baudteiler abhängig von `CLK_FREQ`.
- `DIV_WIDTH` : Breite der Taktteilerzähler.
- `CTRL_OFFSET` : Offset für das Steuerregister.
- `BAUD_OFFSET` : Offset für das Baudratenregister.
- `STATUS_OFFSET` : Offset für Statusbits.
- `TX_OFFSET` : Offset zum Schreiben in die TX-FIFO.
- `RX_OFFSET` : Offset zum Lesen aus der RX-FIFO.

Eingänge:

- `clk` Systemtakt
- `rst_n` Asynchrones, aktives-LOW Reset
- `[7:0]` address Adressoffset zur Auswahl der Register
- `[31:0]` write_data Zu schreibende Daten in das jeweilige Register
- `we` Write-Enable-Signal
- `re` Read-Enable-Signal
- `uart_rx` UART-Eingangssignal (RX)

Ausgänge:

- `[31:0]` read_data Gelesener Wert aus dem entsprechenden Register
- `reg` uart_tx UART-Ausgangssignal (TX)

Source Code

```
`include "../defines.v"
`default_nettype none
`timescale 1ns / 1ns



module uart #(
  parameter FIFO_TX_DEPTH = 16,
  parameter FIFO_RX_DEPTH = 16
) (
  input  wire        clk,
  input  wire        rst_n,
  input  wire [ 7:0] address,
  input  wire [31:0] write_data,
  output wire [31:0] read_data,
  input  wire        we,
  input  wire        re,
  output reg         uart_tx,
  input  wire        uart_rx
);

  // ------------------------------------------------------
  // Vordefinierte Baudteiler für verschiedene Standardbaudraten
  // basierend auf dem Makro `CLK_FREQ` aus "defines.v".
  // ------------------------------------------------------
  localparam BAUD_DIV_115200 = `BAUD_DIV(115200);
  localparam BAUD_DIV_57600  = `BAUD_DIV(57600);
  localparam BAUD_DIV_38400  = `BAUD_DIV(38400);
  localparam BAUD_DIV_28800  = `BAUD_DIV(28800);
  localparam BAUD_DIV_23040  = `BAUD_DIV(23040);
  localparam BAUD_DIV_19200  = `BAUD_DIV(19200);
  localparam BAUD_DIV_14400  = `BAUD_DIV(14400);
  localparam BAUD_DIV_9600   = `BAUD_DIV(9600);
  localparam BAUD_DIV_4800   = `BAUD_DIV(4800);
  localparam BAUD_DIV_2400   = `BAUD_DIV(2400);
  localparam BAUD_DIV_1200   = `BAUD_DIV(1200);
  localparam BAUD_DIV_300    = `BAUD_DIV(300);

  // ------------------------------------------------------
  // Breite der Teilerzähler
  // ------------------------------------------------------
  localparam DIV_WIDTH       = $clog2(BAUD_DIV_300 + 1);


  // ------------------------------------------------------
  // Register-Offsets
  // ------------------------------------------------------
  localparam CTRL_OFFSET     = 8'h00;
  localparam BAUD_OFFSET     = 8'h04;
  localparam STATUS_OFFSET   = 8'h08;
  localparam TX_OFFSET       = 8'h0C;
  localparam RX_OFFSET       = 8'h10;

  // ------------------------------------------------------
  // Zustände für RX/TX State Machines
  // ------------------------------------------------------
  localparam RX_IDLE         = 2'd0;
  localparam RX_START        = 2'd1;
```

```
localparam RX_DATA         = 2'd2;
localparam RX_STOP         = 2'd3;

localparam TX_IDLE         = 2'd0;
localparam TX_START        = 2'd1;
localparam TX_DATA         = 2'd2;
localparam TX_STOP         = 2'd3;

// ----------------------------------------------------------
// Bauddiv-Zähler, Shiftregister etc.
// ----------------------------------------------------------
reg [DIV_WIDTH - 1:0] tx_bd_cntr; //Bauddiv-Zähler für TX
reg [DIV_WIDTH - 1:0] rx_bd_cntr; //Bauddiv-Zähler für RX
reg [DIV_WIDTH - 1:0] bd_div_cnt; // Zwischenspeicher für akt. Baudratenteiler

reg [ 7:0] tx_shift;     // 8-Bit Shiftregister zum Senden
reg [ 7:0] rx_shift;     // 8-Bit Shiftregister zum Empfangen
reg [ 3:0] baud_sel;     // Auswahl der Baudrate (0..11)
reg [ 3:0] new_baud_sel; // Puffer für Baudraten-Update
reg [ 2:0] tx_bit_id;    // Zähler für TX-Bits
reg [ 2:0] rx_bit_id;    // Zähler für RX-Bits
reg [ 1:0] tx_state;     // State Machine: Senden
reg [ 1:0] rx_state;     // State Machine: Empfangen

reg rx_fifo_we;          // Schreibeaktivierung RX-FIFO
reg tx_fifo_rd;          // Leseaktivierung TX-FIFO
reg rx_sync_1;           // Doppelte Synchronisierung von uart_rx
reg rx_sync_2;
reg rx_last;             // Letzter Zustand vom synchronisieren RX
reg new_active;          // Puffer für aktives UART
reg active;              // UART aktiv?

// ----------------------------------------------------------
// FIFOs und Statusbits
// ----------------------------------------------------------
wire [7:0] tx_fifo_dout;
wire [7:0] rx_fifo_dout;
wire [7:0] tx_data;
wire [7:0] rx_data;
wire [5:0] status_bits;

wire tx_fifo_we;
wire rx_fifo_re;
wire tx_fifo_empty;
wire tx_fifo_full;
wire rx_fifo_empty;
wire rx_fifo_full;
wire rx_fall_edge;
wire uart_busy;
wire uart_ready;

// ----------------------------------------------------------
// Wahl des Baudteilers basierend auf baud_sel
// ----------------------------------------------------------
always @( * )
begin
  case (baud_sel)
    4'd0    : bd_div_cnt = BAUD_DIV_115200;
    4'd1    : bd_div_cnt = BAUD_DIV_57600;
    4'd2    : bd_div_cnt = BAUD_DIV_38400;
```

```
    4'd3    : bd_div_cnt = BAUD_DIV_28800;
    4'd4    : bd_div_cnt = BAUD_DIV_23040;
    4'd5    : bd_div_cnt = BAUD_DIV_19200;
    4'd6    : bd_div_cnt = BAUD_DIV_14400;
    4'd7    : bd_div_cnt = BAUD_DIV_9600;
    4'd8    : bd_div_cnt = BAUD_DIV_4800;
    4'd9    : bd_div_cnt = BAUD_DIV_2400;
    4'd10   : bd_div_cnt = BAUD_DIV_1200;
    4'd11   : bd_div_cnt = BAUD_DIV_300;
    default : bd_div_cnt = BAUD_DIV_115200;
  endcase
end

// ------------------------------------------------------
// Aufsplitten von write_data in 8 Bit
// ------------------------------------------------------
assign tx_data[7:0] =   write_data[7:0];
assign rx_data[7:0] = rx_fifo_dout[7:0];

// Wenn in TX_OFFSET geschrieben wird, soll in TX-FIFO geschreiben werden
assign tx_fifo_we   = we && (address[7:0] == TX_OFFSET);

// Wenn RX_OFFSET gelesen wird, soll aus RX-FIFO gelesen werden
assign rx_fifo_re   = re && (address[7:0] == RX_OFFSET);

// ------------------------------------------------------
// UART-Busy, wenn TX oder RX gerade senden/empfangen
// ------------------------------------------------------
assign uart_busy    = ((tx_state != TX_IDLE) | (rx_state != RX_IDLE));
// UART_ready, wenn nicht busy und beide FIFOs leer
assign uart_ready   = !uart_busy & tx_fifo_empty & rx_fifo_empty;

// ------------------------------------------------------
// Statusbits (bit 5:0)
// ------------------------------------------------------
// bit5 = tx_fifo_empty
// bit4 = tx_fifo_full
// bit3 = rx_fifo_empty
// bit2 = rx_fifo_full
// bit1 = uart_busy
// bit0 = uart_ready
assign status_bits  = {uart_ready, uart_busy, rx_fifo_full, rx_fifo_empty, tx_fifo_full, tx_fifo_empty};

// ------------------------------------------------------
// Erkennung fallende Flanke bei RX (Startbit-Erkennung)
// ------------------------------------------------------
assign rx_fall_edge = (rx_last == 1'b1) && (rx_sync_2 == 1'b0);

// ------------------------------------------------------
// Leseauswahl für read_data
// ------------------------------------------------------
assign read_data    = (address[7:0] == CTRL_OFFSET)   ? {31'd0, active} :
                      (address[7:0] == BAUD_OFFSET)   ? {28'd0, baud_sel} :
                      (address[7:0] == STATUS_OFFSET) ? {26'd0, status_bits} :
                      (address[7:0] == RX_OFFSET)     ? {23'd0, !rx_fifo_empty, rx_data} :
                      32'd0;

// ------------------------------------------------------
// TX-FIFO (zur Pufferung ausgehender Daten)
// ------------------------------------------------------
```

```
fifo #(
  .DATA_WIDTH (8),
  .DEPTH      (FIFO_TX_DEPTH)
) tx_fifo_inst (
  .clk   (clk),
  .rst_n (rst_n),
  .wr_en (tx_fifo_we),
  .din   (tx_data),
  .rd_en (tx_fifo_rd),
  .dout  (tx_fifo_dout),
  .empty (tx_fifo_empty),
  .full  (tx_fifo_full)
);


// ------------------------------------------------------------
// RX-FIFO (zur Pufferung eingehender Daten)
// ------------------------------------------------------------
fifo #(
  .DATA_WIDTH (8),
  .DEPTH      (FIFO_RX_DEPTH)
) rx_fifo_inst (
  .clk   (clk),
  .rst_n (rst_n),
  .wr_en (rx_fifo_we),
  .din   (rx_shift),
  .rd_en (rx_fifo_re),
  .dout  (rx_fifo_dout),
  .empty (rx_fifo_empty),
  .full  (rx_fifo_full)
);

// ------------------------------------------------------------
// TX-State Machine:
// TX_IDLE  -> warten, bis Daten in TX-FIFO + active=1
// TX_START -> Latenz für Startbit
// TX_DATA  -> Sendet 8 Datenbits
// TX_STOP  -> Sendet Stopbit
// ------------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    tx_state   <= TX_IDLE;
    tx_bd_cntr <= {DIV_WIDTH{1'b0}};
    tx_fifo_rd <= 1'b0;
    uart_tx    <= 1'b1;
    tx_shift   <= 8'd0;
    tx_bit_id  <= 3'b0;
  end
  else
  begin

    case (tx_state)

      TX_IDLE:
      begin
        tx_bd_cntr <= {DIV_WIDTH{1'b0}};

        if (!tx_fifo_empty && active)
```

```
  begin
    tx_shift   <= tx_fifo_dout;
    tx_state   <= TX_START;
    tx_fifo_rd <= 1'b1;
  end
end

TX_START:
begin
  tx_fifo_rd <= 1'b0;
  if (tx_bd_cntr == bd_div_cnt)
  begin
    tx_bd_cntr <= {DIV_WIDTH{1'b0}};
    tx_bit_id  <= 3'b0;
    uart_tx    <= tx_shift[0];
    tx_shift   <= {1'b0, tx_shift[7:1]};
    tx_state   <= TX_DATA;
  end
  else
  begin
    tx_bd_cntr <= tx_bd_cntr + {{(DIV_WIDTH - 1) {1'b0}}, 1'b1};
    uart_tx    <= 1'b0;
  end
end

TX_DATA:
begin
  if (tx_bd_cntr == bd_div_cnt)
  begin
    tx_bd_cntr <= {DIV_WIDTH{1'b0}};
    uart_tx    <= tx_shift[0];
    tx_shift   <= {1'b0, tx_shift[7:1]};

    if (tx_bit_id < 7)
    begin
      tx_bit_id <= tx_bit_id + 3'b1;
      tx_state  <= TX_DATA;
    end
    else
      tx_state <= TX_STOP;
  end
  else
  begin
    tx_bd_cntr <= tx_bd_cntr + {{(DIV_WIDTH - 1) {1'b0}}, 1'b1};
    tx_state   <= TX_DATA;
  end
end

TX_STOP:
begin
  if (tx_bd_cntr == bd_div_cnt)
  begin
    tx_bd_cntr <= {DIV_WIDTH{1'b0}};
    tx_state   <= TX_IDLE;
  end
  else
  begin
    tx_bd_cntr <= tx_bd_cntr + {{(DIV_WIDTH - 1) {1'b0}}, 1'b1};
    uart_tx    <= 1'b1;
  end
```

```
      end

      default : tx_state <= TX_IDLE;

    endcase
  end
end

// ------------------------------------------------------------
// RX-Synchronisation: Doppelte Abtastung von uart_rx
// -> rx_sync_1, rx_sync_2, rx_last zum Erkennen von Flanken
// ------------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    rx_sync_1 <= 1'b1;
    rx_sync_2 <= 1'b1;
    rx_last   <= 1'b1;
  end
  else
  begin
    rx_sync_1 <= uart_rx;
    rx_sync_2 <= rx_sync_1;
    rx_last   <= rx_sync_2;
  end
end

// ------------------------------------------------------------
// RX-State Machine: Empfangen eines Bytes
// RX_IDLE  -> Warten auf fallende Flanke (Startbit)
// RX_START -> Warte eine halbe Bitdauer
// RX_DATA  -> 8 Bits empfangen
// RX_STOP  -> Stopbit abwarten, ggf. in FIFO schreiben
// ------------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    rx_state   <= RX_IDLE;
    rx_bd_cntr <= {DIV_WIDTH{1'b0}};
    rx_bit_id  <= 3'b0;
    rx_fifo_we <= 1'b0;
  end
  else
  begin
    rx_fifo_we <= 1'b0;

    case (rx_state)

        RX_IDLE:
        begin
          rx_bd_cntr <= {DIV_WIDTH{1'b0}};

          if (rx_fall_edge)
              rx_state <= RX_START;
        end

        RX_START:
        begin
```

```
          if (rx_bd_cntr == (bd_div_cnt >> 1))
          begin
            rx_bd_cntr <= {DIV_WIDTH{1'b0}};
            rx_bit_id  <= 3'b0;
            rx_state   <= RX_DATA;
          end
          else
            rx_bd_cntr <= rx_bd_cntr + 1;
        end

        RX_DATA:
        begin
          if (rx_bd_cntr == bd_div_cnt)
          begin
            rx_bd_cntr          <= {DIV_WIDTH{1'b0}};
            rx_shift[rx_bit_id] <= uart_rx;

            if (rx_bit_id == 7)
              rx_state <= RX_STOP;
            else
              rx_bit_id <= rx_bit_id + 1;

          end
          else
            rx_bd_cntr <= rx_bd_cntr + 1;
        end

        RX_STOP:
        begin
          if (rx_bd_cntr == bd_div_cnt)
          begin
            rx_bd_cntr <= {DIV_WIDTH{1'b0}};

            if (!rx_fifo_full)
                rx_fifo_we <= 1'b1;

            rx_state <= RX_IDLE;
          end
          else
            rx_bd_cntr <= rx_bd_cntr + 1;
        end

        default:
          rx_state <= RX_IDLE;

    endcase
  end
end

// ------------------------------------------------------------
// Steuerung von "active", "baud_sel" (Register-Schreibzugriffe)
// Erst wenn das UART idle und die FIFOs leer sind (uart_ready),
// wird baud_sel und active aktualisiert.
// ------------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    active       <= 1'b1;
    new_active   <= 1'b1;
```

```
      baud_sel     <= 4'd0;
      new_baud_sel <= 4'd0;
    end
    else if (we)
    begin

      case (address[7:0])

        CTRL_OFFSET:
        begin
          new_active   <= write_data[0];
        end

        BAUD_OFFSET:
          new_baud_sel <= write_data[3:0];

        default: ;

      endcase
    end
    else
    if (uart_ready)
    begin
      baud_sel <= new_baud_sel;
      active   <= new_active;
    end
  end

endmodule
```

Datei: rtl/peripherals/ws2812b.v

Modul: ws2812b

Kurzbeschreibung: Treibermodul für WS2812B-LEDs (NeoPixel).

Dieses Modul steuert eine Kette von WS2812B-LEDs. Es werden nacheinander 24-Bit-Farbwerte (GRB) an die LEDs gesendet. Die Zeitsteuerung erfolgt über entsprechende Zähler (T0H, T0L, T1H, T1L, T_RESET). Dieses Beispiel zeigt eine einfache State-Maschine, die die Daten nacheinander an jede LED überträgt, dann ein Reset-Puls erzeugt und wieder von vorne beginnt. Register-Offsets (jeweils 8 Bit auseinanderliegend): - 0x00: Daten für LED 0 - 0x04: Daten für LED 1 - 0x08: Daten für LED 2 - 0x0C: Daten für LED 3 - 0x10: Daten für LED 4 - 0x14: Daten für LED 5 - 0x18: Daten für LED 6 - 0x1C: Daten für LED 7

Lokale Parameter:

- `CLK_PERIOD_NS` Zeit pro Takt in Nanosekunden, abgeleitet aus `CLK_FREQ`.
- `T0H` Länger des “High” bei Bit=0
- `T0L` Länger des “Low” bei Bit=0
- `T1H` Länger des “High” bei Bit=1
- `T1L` Länger des “Low” bei Bit=1
- `T_RESET` Pause, damit die LED-Kette den Frame erkennt
- `STATE_LOAD_LED` Initialer Zustand zum Laden der LED-Daten
- `STATE_SEND_HIGH` Sendephase: Leitung auf High
- `STATE_SEND_LOW` Sendephase: Leitung auf Low
- `STATE_RESET` Reset-Pause nach allen Bits

Eingänge:

- `clk` Systemtakt
- `rst_n` Asynchroner, aktiver-LOW Reset
- `[7:0]` address Adressoffset, um eine der 8 LED-Daten zu wählen
- `[31:0]` write_data Zu schreibender RGB-Wert (24 Bit genutzt)
- `we` Write Enable zum Setzen von LED-Farbwerten

- `re` Read Enable zum Auslesen gespeicherter Werte

Ausgänge:

- `[31:0]` read_data Gibt den gespeicherten RGB-Wert zurück
- `reg` ws_out Ausgangssignal zur WS2812B-Datenleitung

Source Code

```
`include "../defines.v"
`default_nettype none
`timescale 1ns / 1ns



module ws2812b (
  input  wire         clk,
  input  wire         rst_n,
  input  wire  [ 7:0] address,
  input  wire  [31:0] write_data,
  output wire  [31:0] read_data,
  input  wire         we,
  input  wire         re,
  output reg          ws_out
);

  // ------------------------------------------------------
  // Ermitteln der Taktperiode in Nanosekunden basierend auf CLK_FREQ.
  // Werte T0H, T0L, T1H, T1L, T_RESET werden gerundet (IntegerDivision).
  // ------------------------------------------------------
  localparam CLK_PERIOD_NS = 1_000_000_000 / `CLK_FREQ;
  localparam T0H     = (400 + CLK_PERIOD_NS - 1) / CLK_PERIOD_NS;
  localparam T0L     = (850 + CLK_PERIOD_NS - 1) / CLK_PERIOD_NS;
  localparam T1H     = (800 + CLK_PERIOD_NS - 1) / CLK_PERIOD_NS;
  localparam T1L     = (450 + CLK_PERIOD_NS - 1) / CLK_PERIOD_NS;
  localparam T_RESET = (50_000 + CLK_PERIOD_NS - 1) / CLK_PERIOD_NS;

  // ------------------------------------------------------
  // Zustandsdefinitionen
  // ------------------------------------------------------
  localparam STATE_LOAD_LED   = 2'd0;
  localparam STATE_SEND_HIGH  = 2'd1;
  localparam STATE_SEND_LOW   = 2'd2;
  localparam STATE_RESET      = 2'd3;

  // ------------------------------------------------------
  // LED-Daten: 8 LEDs à 24 Bit RGB
  // - led_rgb[i] ist ein 24-Bit-Wert, repräsentiert G,R,B
  // - Schieberegister shift_reg zum Senden
  // ------------------------------------------------------
  reg [23:0] led_rgb [0:7];
  reg [23:0] shift_reg;
  reg [15:0] timer;
  reg [ 4:0] bit_index;
  reg [ 2:0] led_index;
  reg [ 1:0] state;

  // ------------------------------------------------------
  // Lese-Multiplexer: Bei Adressen 0x00, 0x04, 0x08, ... 0x1C
  // wird der entsprechende led_rgb[i]-Wert (24 Bit) zurückgegeben.
  // Die oberen 8 Bit von read_data bleiben 0.
  // ------------------------------------------------------
```

```
assign read_data = (address[7:0] == 8'h00) ? {8'b0, led_rgb[0]} :
                   (address[7:0] == 8'h04) ? {8'b0, led_rgb[1]} :
                   (address[7:0] == 8'h08) ? {8'b0, led_rgb[2]} :
                   (address[7:0] == 8'h0C) ? {8'b0, led_rgb[3]} :
                   (address[7:0] == 8'h10) ? {8'b0, led_rgb[4]} :
                   (address[7:0] == 8'h14) ? {8'b0, led_rgb[5]} :
                   (address[7:0] == 8'h18) ? {8'b0, led_rgb[6]} :
                   (address[7:0] == 8'h1C) ? {8'b0, led_rgb[7]} :
                   32'd0;

// ------------------------------------------------------------
// Schreiben in die LED-Register
// - address = 0x00, 0x04, ... => speichert 24 Bit in led_rgb
// ------------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    led_rgb[0] <= 24'h0;
    led_rgb[1] <= 24'h0;
    led_rgb[2] <= 24'h0;
    led_rgb[3] <= 24'h0;
    led_rgb[4] <= 24'h0;
    led_rgb[5] <= 24'h0;
    led_rgb[6] <= 24'h0;
    led_rgb[7] <= 24'h0;
  end
  else if (we)
  begin
    case(address)
      8'h00: led_rgb[0] <= write_data[23:0];
      8'h04: led_rgb[1] <= write_data[23:0];
      8'h08: led_rgb[2] <= write_data[23:0];
      8'h0C: led_rgb[3] <= write_data[23:0];
      8'h10: led_rgb[4] <= write_data[23:0];
      8'h14: led_rgb[5] <= write_data[23:0];
      8'h18: led_rgb[6] <= write_data[23:0];
      8'h1C: led_rgb[7] <= write_data[23:0];
      default: ;
    endcase
  end
end

// ------------------------------------------------------------
// Haupt-Zustandsmaschine:
// 1) STATE_LOAD_LED: Lade Daten der aktuellen LED in shift_reg.
// 2) STATE_SEND_HIGH: Sende "High"-Phase (T0H oder T1H).
// 3) STATE_SEND_LOW:  Sende "Low"-Phase  (T0L oder T1L).
// 4) STATE_RESET:     Nach allen Bits die Resetlücke (T_RESET).
// ------------------------------------------------------------
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n)
  begin
    state     <= STATE_LOAD_LED;
    led_index <= 3'd0;
    bit_index <= 5'd23;
    timer     <= 0;
    ws_out    <= 0;
    shift_reg <= 0;
```

```
end
else
begin
  case (state)

    STATE_LOAD_LED:
    begin
      shift_reg <= {led_rgb[led_index][23:16], led_rgb[led_index][15:8], led_rgb[led_index][7:0]};
      bit_index <= 23;

      if (shift_reg[23]) begin
        timer <= T1H - 1;
      end else begin
        timer <= T0H - 1;
      end

      ws_out <= 1;
      state  <= STATE_SEND_HIGH;
    end

    STATE_SEND_HIGH:
    begin
      if (timer == 0) begin
        timer <= shift_reg[bit_index] ? (T1L - 1) : (T0L - 1);
        ws_out <= 0;
        state <= STATE_SEND_LOW;
      end else begin
        timer <= timer - 1;
      end
    end

    STATE_SEND_LOW:
    begin
      if (timer == 0) begin
        if (bit_index == 0) begin
          if (led_index == 7) begin
            timer <= T_RESET - 1;
            state <= STATE_RESET;
          end else begin
            led_index <= led_index + 1;
            state <= STATE_LOAD_LED;
          end
        end else begin
          bit_index <= bit_index - 1;
          timer <= shift_reg[bit_index - 1] ? (T1H - 1) : (T0H - 1);
          ws_out <= 1;
          state <= STATE_SEND_HIGH;
        end
      end else begin
        timer <= timer - 1;
      end
    end

    STATE_RESET:
    begin
      if (timer == 0) begin
        led_index <= 0;
        state <= STATE_LOAD_LED;
      end else begin
        timer <= timer - 1;
```

```
          end
        end

        default: state <= STATE_LOAD_LED;

      endcase
    end
  end

endmodule
```

Datei: rtl/register_file.v

Modul: register_file

Kurzbeschreibung: Registerdatei für den CPU-Kern.

Diese Registerdatei umfasst entweder 32 Register (im RV32I-Mode) oder 16 Register (ansonsten) und unterstützt Lese- und Schreibzugriffe. Der Schreibzugriff erfolgt über ein Write-Enable-Signal, sowie Auswahl des Ziel- und Quellregisters. Bei RISC-V (RV32I) bleibt das Register x0 jederzeit auf 0. @macro RV32I Falls definiert, werden 32 Register angelegt.

Eingänge:

- `clk` Systemtakt.
- `rst_n` Asynchroner, aktiver-LOW Reset.
- `we` Write-Enable-Signal zum Beschreiben eines Registers.
- `[4:0]` rd Zielregister, in das bei we=1 geschrieben werden soll.
- `[4:0]` rs1 Erstes Quellregister für eine Leseanfrage.
- `[4:0]` rs2 Zweites Quellregister für eine Leseanfrage.
- `[31:0]` rd_data Daten, die in rd geschrieben werden (falls we=1).

Ausgänge:

- `[31:0]` rs1_data Daten aus dem Register rs1.
- `[31:0]` rs2_data Daten aus dem Register rs2.

Source Code

```
`include "./defines.v"
`default_nettype none
`timescale 1ns / 1ns



module register_file (
  input  wire        clk,
  input  wire        rst_n,
  input  wire        we,
  input  wire [ 4:0] rd,
  input  wire [ 4:0] rs1,
  input  wire [ 4:0] rs2,
  input  wire [31:0] rd_data,
  output wire [31:0] rs1_data,
  output wire [31:0] rs2_data
);

`ifdef RV32I
  // Bei aktiviertem RV32I-Macro werden 32 Register definiert (1..31).
  // Register 0 bleibt immer 0.
  reg [31:0] registers[31:1];
`else
  // Sonst 16 Register (1..15).
  reg [31:0] registers[15:1];
`endif
```

```
  // ------------------------------------------------------
  // Lesezugriffe:
  // - Wenn rs1 != 0, wird das entsprechende Register ausgegeben,
  //   sonst 0.
  // - Wenn rs2 != 0, wird das entsprechende Register ausgegeben,
  //   sonst 0.
  // ------------------------------------------------------
`ifdef RV32I
  assign rs1_data  = (rs1 != 5'd0) ? registers[rs1] : 32'd0;
  assign rs2_data  = (rs2 != 5'd0) ? registers[rs2] : 32'd0;
`else
  // Nur die unteren 16 Register sind gültig (rd[4] = 0).
  assign rs1_data = (rs1 != 5'd0 && ~rs1[4]) ? registers[rs1] : 32'd0;
  assign rs2_data = (rs2 != 5'd0 && ~rs2[4]) ? registers[rs2] : 32'd0;
`endif

  // ------------------------------------------------------
  // Schreibzugriffe:
  // - Bei RV32I wird registr[rd] nur beschrieben, wenn rd != 0.
  // - Ansonsten nur, wenn rd != 0 und rd[4] nicht gesetzt.
  // ------------------------------------------------------
  always @(posedge clk or negedge rst_n)
  begin
    if (!rst_n)
    begin
      registers[ 1] <= 32'd0;
      registers[ 2] <= 32'd0;
      registers[ 3] <= 32'd0;
      registers[ 4] <= 32'd0;
      registers[ 5] <= 32'd0;
      registers[ 6] <= 32'd0;
      registers[ 7] <= 32'd0;
      registers[ 8] <= 32'd0;
      registers[ 9] <= 32'd0;
      registers[10] <= 32'd0;
      registers[11] <= 32'd0;
      registers[12] <= 32'd0;
      registers[13] <= 32'd0;
      registers[14] <= 32'd0;
      registers[15] <= 32'd0;
`ifdef RV32I
      registers[16] <= 32'd0;
      registers[17] <= 32'd0;
      registers[18] <= 32'd0;
      registers[19] <= 32'd0;
      registers[20] <= 32'd0;
      registers[21] <= 32'd0;
      registers[22] <= 32'd0;
      registers[23] <= 32'd0;
      registers[24] <= 32'd0;
      registers[25] <= 32'd0;
      registers[26] <= 32'd0;
      registers[27] <= 32'd0;
      registers[28] <= 32'd0;
      registers[29] <= 32'd0;
      registers[30] <= 32'd0;
      registers[31] <= 32'd0;
`endif
    end
```

```
    else
`ifdef RV32I
    if (we && (rd != 5'd0))
      registers[rd] <= rd_data;
`else
    if (we && (rd != 5'd0) && ~rd[4])
      registers[rd] <= rd_data;
`endif
  end

endmodule
```

Datei: rtl/wgr_v_max.v

Modul: wgr_v_max

Kurzbeschreibung: Top-Level Modul („wgr_v_max“) eines minimalistischen SoC-Systems in HDL (mit CPU, Speicher und Peripherie).

Dieses Modul instanziiert die CPU (`cpu.v`) sowie den Hauptspeicher (`memory.v`). Es stellt außerdem einige grundlegende externe Signale bereit (UART, SPI, PWM, WS, GPIO) und kann somit als oberstes Top-Level Modul fungieren. Zusätzlich enthält es hier exemplarisch einen einfachen LED-Lauflicht-Zähler, der in `gpio_shift` gespeicherte Bits nacheinander auf `gpio_out` legt.

Eingänge:

- `clk` Systemtakt
- `rst_n` Asynchroner, aktiver-LOW Reset
- `uart_rx` RX-Eingang der UART-Schnittstelle
- `spi_miso` SPI-Eingang (Slave->Master)
- [7:0] gpio_in GPIO-Eingänge

Ausgänge:

- `uart_tx` TX-Ausgang der UART-Schnittstelle
- `[31:0]` debug_out Debugging-Signal aus dem Debug-Modul
- `pwm_out` PWM-Ausgang
- `ws_out` Datenleitung für WS2812B-LEDs
- `spi_mosi` SPI-Ausgang (Master->Slave)
- `spi_clk` SPI-Takt
- `spi_cs` SPI-Chipselect
- [7:0] gpio_out GPIO-Ausgänge
- [7:0] gpio_dir GPIO-Richtung (derzeit ungenutzt)

Source Code

```
`include "./defines.v"
`default_nettype none
`timescale 1ns / 1ns



module wgr_v_max (
  input  wire        clk,
  input  wire        rst_n,
  output wire        uart_tx,
  input  wire        uart_rx,
  output wire [31:0] debug_out,
  output wire        pwm_out,
  output wire        ws_out,
  output wire        spi_mosi,
  input  wire        spi_miso,
  output wire        spi_clk,
  output wire        spi_cs,
```

```
  output wire [ 7:0] gpio_out,
  output wire [ 7:0] gpio_dir,
  input  wire [ 7:0] gpio_in
);


  // ------------------------------------------------------
  // Interne Verbindungs-Signale zwischen CPU und Memory
  // ------------------------------------------------------
  wire [31:0] address;
  wire [31:0] write_data;
  wire [31:0] read_data;
  wire        mem_busy;
  wire        we;
  wire        re;

  // ------------------------------------------------------
  // Beispiel: Lauflicht-Steuerung für GPIO
  // ------------------------------------------------------

  //assign gpio_out = gpio_shift;

  reg [ 7:0]  gpio_shift;

  reg [22:0] counter = 0;
  reg [2:0] position = 0;

  always @(posedge clk) begin
      if (counter >= 23'd6000000 - 1) begin
          counter <= 0;
          gpio_shift <= 8'b00000001 << position;
          position <= (position == 3'd7) ? 0 : position + 1;
      end else begin
          counter <= counter + 1;
      end
  end

  // ------------------------------------------------------
  // CPU-Instanz: generiert Adresse, write_data, we, re und
  // empfängt read_data sowie mem_busy
  // ------------------------------------------------------
  cpu cpu_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (address),
    .write_data (write_data),
    .read_data  (read_data),
    .we         (we),
    .re         (re),
    .mem_busy   (mem_busy)
  );

  // ------------------------------------------------------
  // Speicher-Modul (RAM/Peripherie), an das die CPU
  // ihre Zugriffe weiterleitet
  // ------------------------------------------------------
  memory memory_inst (
    .clk        (clk),
    .rst_n      (rst_n),
    .address    (address),
```

```
    .write_data (write_data),
    .read_data  (read_data),
    .we         (we),
    .re         (re),
    .mem_busy   (mem_busy),
    .uart_tx    (uart_tx),
    .uart_rx    (uart_rx),
    .debug_out  (debug_out),
    .pwm_out    (pwm_out),
    .ws_out     (ws_out),
    .spi_mosi   (spi_mosi),
    .spi_miso   (spi_miso),
    .spi_clk    (spi_clk),
    .spi_cs     (spi_cs),
    .gpio_out   (gpio_out),
    .gpio_dir   (gpio_dir),
    .gpio_in    (gpio_in)
  );

endmodule
```