

# WGR-V Verilog Doku

## Inhaltsverzeichnis

- **Datei:** rtl/peripherals/fifo.v
  - Modul: fifo
- **Datei:** rtl/peripherals/gpio.v
  - Modul: gpio
- **Datei:** rtl/peripherals/seq\_multiplier.v
  - Modul: seq\_multiplier

## 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
);

localparam ADDR_WIDTH = $clog2(DEPTH);
```

```

reg [ADDR_WIDTH :0]    rd_ptr;
reg [ADDR_WIDTH :0]    wr_ptr;
reg [DATA_WIDTH-1:0] mem[0:DEPTH-1];

reg wr_en_prev;
reg rd_en_prev;

wire [ADDR_WIDTH: 0] next_wr;

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);

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
        wr_en_prev <= wr_en;
        rd_en_prev <= rd_en;

        if (wr_en && !wr_en_prev && !full)
        begin
            mem[wr_ptr[ADDR_WIDTH - 1: 0]] <= din;
            wr_ptr <= next_wr;
        end

        if (rd_en && !rd_en_prev && !empty)
        begin
            rd_ptr <= rd_ptr + 1;
        end
    end
end

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.

**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
);

    // No dir implemented at the moment
    localparam GPIO_DIR_BASE      = 8'h00;
    localparam GPIO_IN_OFFSET     = 8'h04;
    localparam GPIO_OUT_OFFSET    = 8'h08;
    localparam GPIO_OUT_STEP      = 8'h04;

    wire [2:0] pin_index = (address - GPIO_OUT_OFFSET) >> 2;

    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;

    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/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
);

    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;

    reg [31:0] multiplicand;
    reg [31:0] multiplier;
    reg [63:0] product;
    reg [ 5:0] bit_index;
    reg      busy;

    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;

    always @(posedge clk or negedge rst_n)
    begin
```

```

if (!rst_n) begin
    multiplicand <= 32'd0;
    multiplier   <= 32'd0;
    product      <= 64'd0;
    bit_index    <= 6'd0;
    busy         <= 1'b0;
end
else
begin
    if (we)
    begin
        case (address)
            MUL1_OFFSET: begin
                multiplicand <= write_data;
            end
            MUL2_OFFSET: begin
                multiplier <= write_data;
                product    <= 64'd0;
                bit_index  <= 6'd31;
                busy       <= 1'b1;
            end
        endcase
    end

    if (busy)
    begin
        if (multiplier[bit_index])
        begin
            product <= product + ((64'd1 << bit_index) * multiplicand);
        end

        if (bit_index == 0)
        begin
            busy <= 1'b0;
        end
        else
        begin
            bit_index <= bit_index - 1;
        end
    end
end
end
endmodule

```