

WGR-V Verilog Doku

Inhaltsverzeichnis

- **Datei:** rtl/peripherals/debug_module.v
 - Modul: debug_module
- **Datei:** rtl/peripherals/fifo.v
 - Modul: fifo
- **Datei:** rtl/peripherals/gpio.v
 - Modul: gpio
- **Datei:** rtl/peripherals/peripheral_bus.v
 - Modul: peripheral_bus
- **Datei:** rtl/peripherals/pwm_timer.v
 - Modul: pwm_timer
- **Datei:** rtl/peripherals/seq_multiplier.v
 - Modul: seq_multiplier

Datei: rtl/peripherals/debug_module.v

Modul: debug_module

Kurzbeschreibung: Debug-Modul für Host-Kommunikation und Speicherzugriff

Dieses Modul stellt eine einfache Debug-Schnittstelle über UART bereit, um mit einem Host-System zu kommunizieren. Es ermöglicht das Lesen und Schreiben von Speicherinhalten über serielle Kommandos. Über den UART-Eingang **rx** empfängt das Modul Befehle, die über interne Kontrollsignale (**mem_rd**, **mem_wr**, etc.) an den CPU-internen Speicher weitergegeben werden. Das Antwortsignal wird über **tx** gesendet.

Eingänge:

- **clk** Systemtakt
- **rst** Reset-Signal
- **rx** Eingang vom seriellen Host (UART)
- **mem_data_from_cpu** Daten aus dem Speicher (vom CPU-Interface)
- **mem_ready_from_cpu** Bereitschaftssignal vom Speicher

Ausgänge:

- **tx** Ausgang zur seriellen Schnittstelle
- **mem_addr_to_cpu** Speicheradresse für den Zugriff
- **mem_data_to_cpu** Daten, die geschrieben werden sollen
- **mem_wr** Speicher-Schreibsignal
- **mem_rd** Speicher-Lesesignal
- **mem_valid** Gültigkeitssignal für Speicherzugriff

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
);

localparam DEBUG_ADDR = 32'h00000000;
```

```

reg [31:0] debug_reg;

assign read_data = debug_reg;
assign debug_out = debug_reg;

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;
        $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
);

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
    end
```

```

    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: Einfacher Adressdecoder für Peripheriezugriffe

Dieses Modul implementiert einen einfachen Peripheriebus mit Adressdekodierung. Es leitet Speicherzugriffe basierend auf der übergebenen Adresse an die jeweils zuständige Peripherieeinheit weiter. Dabei werden Steuersignale (**wr**, **rd**, **valid**) sowie Datenleitungen entsprechend durchgeschaltet. Es unterstützt mehrere Peripheriegeräte mit jeweils eigenen Adressbereichen.

Eingänge:

- **clk** Systemtakt
- **rst** Reset-Signal
- **addr** Adresse des Zugriffs
- **wr** Schreibbefehl
- **rd** Lesebefehl
- **valid** Gültigkeitssignal für Zugriff
- **cpu_data** Daten vom CPU zur Peripherie

Ausgänge:

- **data_out** Rückgabewert von der aktiven Peripherie
- **ready** Gibt an, ob die Peripherie bereit ist
- **gpio_sel** Auswahl für GPIO-Modul
- **uart_sel** Auswahl für UART-Modul
- **spi_sel** Auswahl für SPI-Modul
- **ws2812b_sel** Auswahl für WS2812B-Modul
- **pwm_timer_sel** Auswahl für PWM-Timer
- **system_timer_sel** Auswahl für System-Timer
- **debug_module_sel** Auswahl für Debug-Modul

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
);

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;

wire [7:0] func_addr;

assign func_addr = address[7:0];

// Debug
`ifdef INCLUDE_DEBUG

wire [31:0] debug_data;

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)
);

`else
wire [31:0] debug_data = 32'h0;
wire      debug_sel   = 1'b0;
assign     debug_out   = 32'h0;
`endif

// UART
`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
`ifndef 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
`ifndef 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

// Sequential Multiplier
`ifndef 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

// Sequential Divider
`ifndef 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 Peripheral
`ifndef 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 Peripheral
`ifndef 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
`ifndef 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

    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
    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.

Eingänge:

- clk Systemtakt
- rst Reset-Signal
- enable Aktivierung des Timers
- [31:0] period Maximale Zählerperiode (PWM-Zykluslänge)
- [31:0] duty_cycle Vergleichswert für Pulsbreite (PWM-Anteil)

Ausgänge:

- pwm_out 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
);

    localparam PERIOD_ADDR = 8'h00;
    localparam DUTY_ADDR   = 8'h04;
    localparam COUNTER_ADDR = 8'h08;
    localparam CTRL_ADDR   = 8'h0C;

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

    assign read_data = (address == PERIOD_ADDR) ? period :
                      (address == DUTY_ADDR)   ? duty   :
                      (address == COUNTER_ADDR) ? counter :
                      (address == CTRL_ADDR)    ? ctrl   :
                      32'h0;

    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
                case (address)
                    PERIOD_ADDR:
                        period    <= write_data;
                    DUTY_ADDR:
                        duty      <= write_data;
                endcase
            end
        end
    end
endmodule
```

```

    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 <= pre_count - 1;
end else begin
    pre_count <= ctrl[31:16];

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

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_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
            end
        end
    end
end
```

```

        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

```