Bit Packer Design and Optimization

VLSI Signal Processing Course Project

Team Members

– Ajit Kumar Singh (SC22B123)
– Rishi Verma (SC22B141)
– Saurabh Kumar (SC22B146)

1 Introduction

Bit packing is a technique used in digital systems to efficiently store variable-length data into fixed-size storage elements. It accumulates variable-length data inputs and packs them into 32-bit words. When a word is completely filled, it outputs it as a valid 32-bit value. This design is essential in VLSI systems where bandwidth efficiency and custom-width data handling are necessary, such as in compression engines, network packet processing, or signal stream processors. In this project, a 32-bit bit packer module is designed and optimized to meet timing constraints and reduce power consumption.

2 Low Frequency Architecture

The design uses two 32-bit registers, `R1` and `R2`, to alternate writing depending on a toggle signal `wrt_ptr_bit`. It receives inputs (`in_data`) of dynamic bit-widths (`in_bits`), appends them to the current register, and emits a packed output when a register becomes full (32 bits). This design uses integer-based temporary variables and large combinational logic in a single always block.

2.1 Verilog Code

Low Frequency Architecture

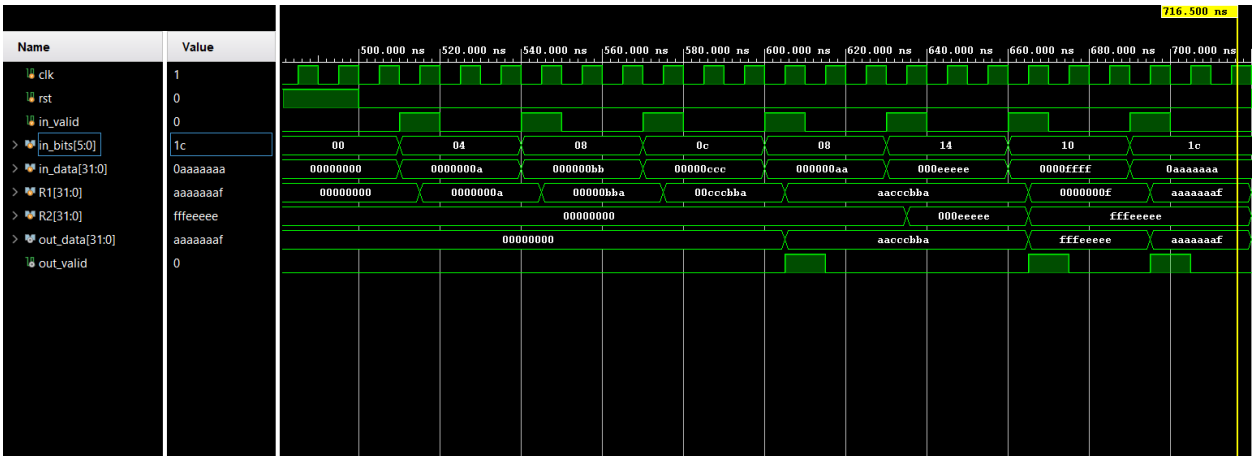
```
‘timescale 1ns / 1ps

module bit_packing (
    input wire       clk,
    input wire       rst,
    input wire       in_valid,
    input wire [31:0] in_data,
```

```
    input wire [5:0] in_bits,     // Max 32 bits
    output reg       out_valid,
    output reg [31:0] out_data
);

    reg [31:0] R1;
    reg[5:0] R1_bits;

    reg [31:0] R2;
    reg[5:0] R2_bits;

reg wrt_ptr_bit;
integer fit_bits;
integer rem_bits;


    always @(posedge clk or posedge rst) begin
        if (rst) begin
            R1     <= 32'b0;
            R1_bits <= 0;
            R2        <= 32'b0;
            R2_bits   <= 0;
            wrt_ptr_bit   <= 1'b0;
            out_data  <= 32'b0;
            out_valid <= 0;
        end else begin
            out_valid <= 0;
            if(in_valid) begin
               if(!wrt_ptr_bit) begin
// writing in R1
                  if (R1_bits + in_bits <= 32) begin
                      // Enough space in R1 register
                      R1 <= R1 | ((in_data & ((32'h1 << in_bits) - 1)) << R1_bits);
                      R1_bits <= R1_bits + in_bits;

                      // Output if exactly full
                      if (R1_bits + in_bits == 32) begin
                          R1 <= R1 | ((in_data & ((32'h1 << in_bits) - 1)) << R1_bits);
                          out_data <= R1 | ((in_data & ((32'h1 << in_bits) - 1)) <<
                              R1_bits);
                          out_valid <= 1;
                          wrt_ptr_bit <= 1'b1;
                          R1_bits <= 0;
                      end
                  end

                  else begin
                      // Not enough space in R1, split input
                      fit_bits = 32 - R1_bits;
                      rem_bits = in_bits - fit_bits;
                      // Fill R1 register
                      R1 <= R1 | ((in_data & ((32'h1 << fit_bits) - 1)) << R1_bits);
                      out_data <= R1 | ((in_data & ((32'h1 << fit_bits) - 1)) << R1_bits)
                          ;
```

```
                    out_valid <= 1;
                    wrt_ptr_bit <= 1'b1;
                    // Put remaining bits in R2 register
                    R2 <= (in_data >> fit_bits) & ((1 << rem_bits) - 1);
                    R2_bits <= rem_bits;
                    R1_bits <= 0;
                end
            end

// writing in R2
        else begin
                if (R2_bits + in_bits <= 32) begin
                    // Enough space in R2 register
                    R2 <= R2 | ((in_data & ((32'h1 << in_bits) - 1)) << R2_bits);
                    R2_bits <= R2_bits + in_bits;

                    // Output if exactly full
                    if (R2_bits + in_bits == 32) begin
                        R2 <= R2 | ((in_data & ((32'h1 << in_bits) - 1)) << R2_bits);
                        out_data <= R2 | ((in_data & ((32'h1 << in_bits) - 1)) <<
                            R2_bits);
                        out_valid <= 1;
                        wrt_ptr_bit <= 1'b0;
                        R2_bits <= 0;
                    end
                end

                else begin
                    // Not enough space in R2, split input
                    fit_bits = 32 - R2_bits;
                    rem_bits = in_bits - fit_bits;
                    // Fill R2 register
                    R2 <= R2 | ((in_data & ((32'h1 << fit_bits) - 1)) << R2_bits);
                    out_data <= R2 | ((in_data & ((32'h1 << fit_bits) - 1)) << R2_bits)
                        ;
                    out_valid <= 1;
                    wrt_ptr_bit <= 1'b0;
                    // Put remaining bits in R1 register
                    R1 <= (in_data >> fit_bits) & ((1 << rem_bits) - 1);
                    R1_bits <= rem_bits;
                    R2_bits <= 0;
                end
            end
        end
    end
endmodule
```

2.2 Behavioral Simulation

Figure below shows the simulation waveform of the above architecture.



Behavioral Simulation: Low frequency Architecture

2.3 Timing Summary

The timing report showed a worst negative slack (WNS) of 0.051 ns at a 6.1 ns clock period.

Timing Constraint

```
create_clock -period 6.1 -name clk -waveform {0.000 3.05} -add [get_ports clk]
```

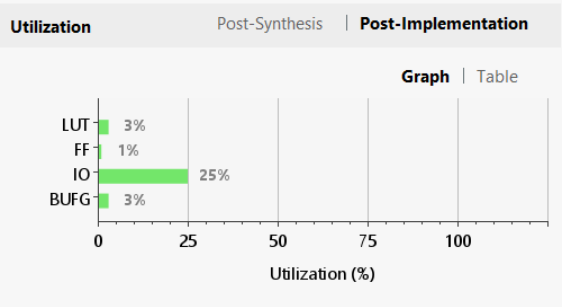
Metric	Value
Worst Negative Slack	0.051 ns
Clock Period	6.1 ns

Design Timing Summary

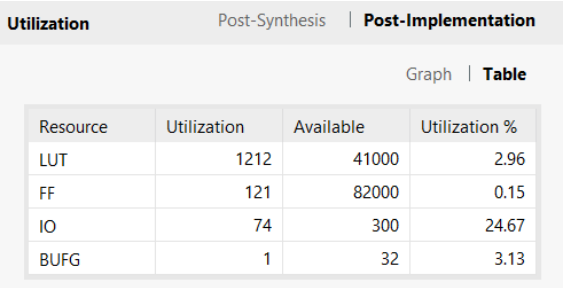
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.051 ns	Worst Hold Slack (WHS):	0.208 ns	Worst Pulse Width Slack (WPWS):	2.650 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	243	Total Number of Endpoints:	243	Total Number of Endpoints:	123

All user specified timing constraints are met.

2.4 Resource Utilization



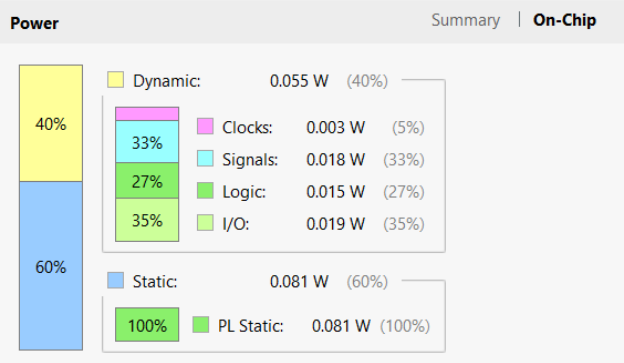
Graph



Resource	Utilization	Available	Utilization %
LUT	1212	41000	2.96
FF	121	82000	0.15
IO	74	300	24.67
BUFG	1	32	3.13

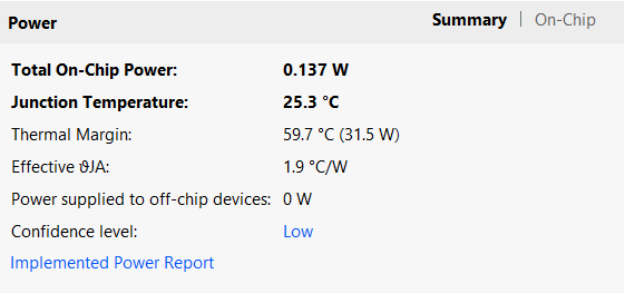
Table

2.5 Power Consumption



Power Consumption: On-Chip

Component	Power (W)
Dynamic Power	0.055
Static Power	0.081
Total Power	0.137



Power Consumption: Summary

3 Optimized Architecture

To meet the timing constraint, the bit packer was re-architected using FSM-based pipelining and regular structures such as arrays for register handling.

3.1 Verilog Code

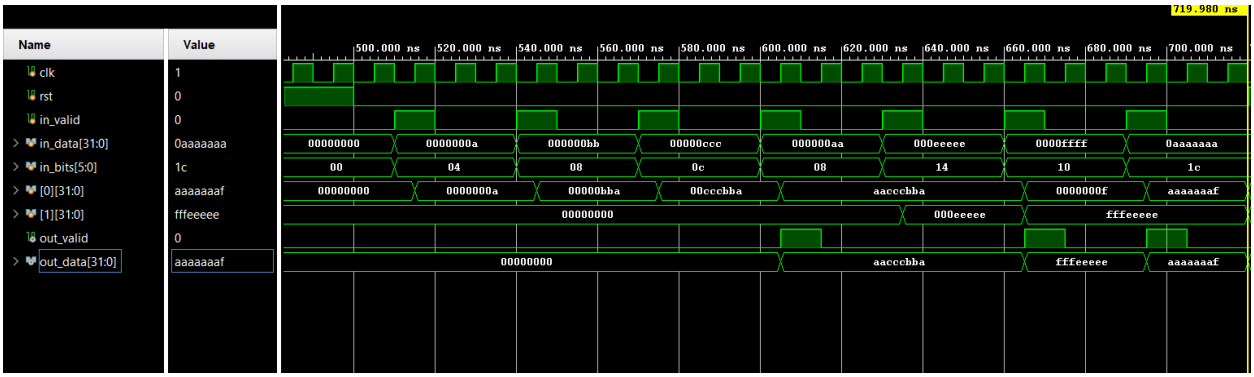
Optimized Architecture

```
‘timescale 1ns / 1ps

module bit_packing (
    input wire       clk,
    input wire       rst,
    input wire       in_valid,
    input wire [31:0] in_data,
    input wire [5:0] in_bits,
    output reg       out_valid,
    output reg [31:0] out_data
);

    reg [31:0] R[1:0];
    reg [5:0] R_bits[1:0];
    reg       wrt_ptr_bit;

    reg [1:0] state;
    localparam IDLE = 2’d0,
               PACK = 2’d1,
               OUTPUT = 2’d2;

    reg [31:0] data_masked;
    reg [5:0] curr_bits;
    reg [5:0] free_bits;
    reg [5:0] rem_bits;
    reg [31:0] lower_part, upper_part;
    reg       fits_in_reg, fits_exact, overflows;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            R[0] <= 0; R[1] <= 0;
            R_bits[0] <= 0; R_bits[1] <= 0;
            wrt_ptr_bit <= 0;
            out_data <= 0;
            out_valid <= 0;
            state <= IDLE;
        end else begin
            out_valid <= 0;
            case (state)
                IDLE: begin
                    if (in_valid) begin
                        data_masked <= in_data & ((32’h1 << in_bits) - 1);
                        curr_bits <= R_bits[wrt_ptr_bit];
```

```
                    free_bits <= 32 - R_bits[wrt_ptr_bit];
                    fits_in_reg <= (R_bits[wrt_ptr_bit] + in_bits < 32);
                    fits_exact <= (R_bits[wrt_ptr_bit] + in_bits == 32);
                    overflows  <= (R_bits[wrt_ptr_bit] + in_bits > 32);
                    rem_bits <= in_bits - (32 - R_bits[wrt_ptr_bit]);
                    state <= PACK;
                end
            end

            PACK: begin
                if (fits_in_reg) begin
                    R[wrt_ptr_bit] <= R[wrt_ptr_bit] | (data_masked << curr_bits);
                    R_bits[wrt_ptr_bit] <= curr_bits + in_bits;
                    state <= IDLE;
                end
                else if (fits_exact) begin
                    R[wrt_ptr_bit] <= R[wrt_ptr_bit] | (data_masked << curr_bits);
                    out_data <= R[wrt_ptr_bit] | (data_masked << curr_bits);
                    out_valid <= 1;
                    R_bits[wrt_ptr_bit] <= 0;
                    wrt_ptr_bit <= ~wrt_ptr_bit;
                    state <= IDLE;
                end
                else if (overflows) begin
                    lower_part <= data_masked & ((32'h1 << free_bits) - 1);
                    upper_part <= data_masked >> free_bits;
                    state <= OUTPUT;
                end
            end

            OUTPUT: begin
                R[wrt_ptr_bit] <= R[wrt_ptr_bit] | (lower_part << curr_bits);
                out_data <= R[wrt_ptr_bit] | (lower_part << curr_bits);
                out_valid <= 1;
                R_bits[wrt_ptr_bit] <= 0;
                wrt_ptr_bit <= ~wrt_ptr_bit;

                R[~wrt_ptr_bit] <= upper_part;
                R_bits[~wrt_ptr_bit] <= rem_bits;

                state <= IDLE;
            end
        endcase
    end
  end
endmodule
```

3.2 Behavioral Simulation

Figure below shows the simulation waveform of the optimized architecture.



Behavioral Simulation: Optimized Architecture

3.3 Timing Summary

The timing report showed a worst negative slack (WNS) of 0.137 ns at a 3.2 ns clock period.

Timing Constraint

```
create_clock -period 3.2 -name clk -waveform {0.000 1.6} -add [get_ports clk]
```

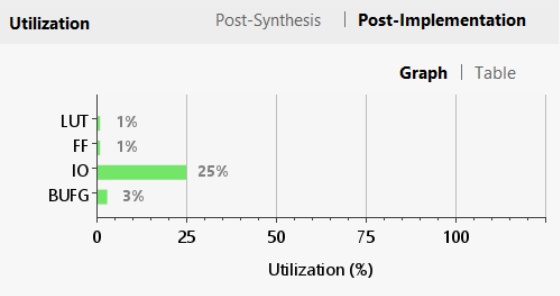
Metric	Value
Worst Negative Slack	0.137 ns
Clock Period	3.2 ns

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.137 ns	Worst Hold Slack (WHS):	0.123 ns	Worst Pulse Width Slack (WPWS):	1.200 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	420	Total Number of Endpoints:	420	Total Number of Endpoints:	229

All user specified timing constraints are met.

3.4 Resource Utilization

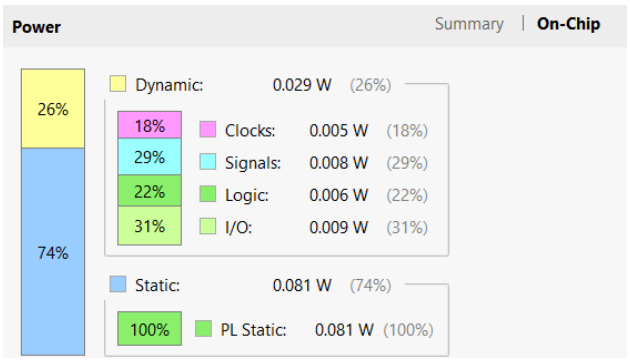


Graph

Resource	Utilization	Available	Utilization %
LUT	602	41000	1.47
FF	228	82000	0.28
IO	74	300	24.67
BUFG	1	32	3.13

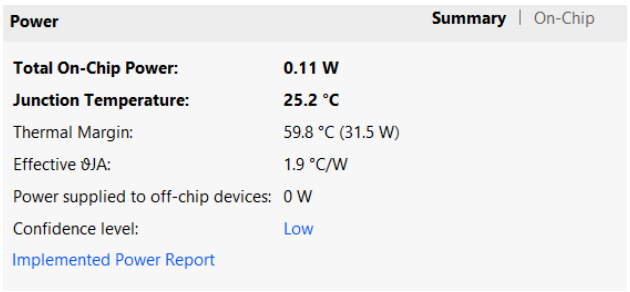
Table

3.5 Power Consumption



Power Consumption: On-Chip

Component	Power (W)
Dynamic Power	0.029
Static Power	0.081
Total Power	0.11



Power Consumption: Summary

4 VLSI DSP Techniques Applied

Several architectural and DSP/VLSI techniques were applied during optimization:

- **FSM-based pipelining** to break long critical paths
- **Bit slicing** and **masking** to manage variable-width input
- **Array-based register handling** to enable regular synthesis
- **Control/data path separation** for simplified logic
- **Avoidance of non-synthesizable types** like `integer`

5 Conclusion

The optimized design meets 3.2 ns (312 MHz) timing constraint and improves synthesis quality and maintainability. VLSI DSP techniques were critical in reducing critical path delays while preserving the functionality of the original design.

6 Appendix

Testbench Code

```
`timescale 1ns / 1ps

module bit_packing_tb;

    // Inputs
    reg clk;
    reg rst;
    reg in_valid;
    reg [31:0] in_data;
    reg [5:0] in_bits; // Max 32 bits

    // Outputs
    wire out_valid;
    wire [31:0] out_data;

    // Instantiate the bit_packing module
    bit_packing uut (
        .clk(clk),
        .rst(rst),
        .in_valid(in_valid),
        .in_data(in_data),
        .in_bits(in_bits),
        .out_valid(out_valid),
        .out_data(out_data)
    );

    // Clock generation
    always begin
        #5 clk = ~clk; // 100MHz clock
    end

    // Initial block to initialize signals and apply test cases
    initial begin
        // Initialize Inputs
        clk = 0;
        rst = 0;
        in_valid = 0;
        in_data = 0;
        in_bits = 0;

        // Apply reset
        rst = 1;
        #500;
        rst = 0;

        // Test Case 1: Pack 4 bits into current register
        #10;
        in_valid = 1;
        in_data = 32'hA; // some test data
```

```
        in_bits = 4; // pack 4 bits
        #10;
        in_valid = 0;
        #10;

        // Test Case 2: Pack 8 bits into current register (when there's remaining space)
        #10;
        in_valid = 1;
        in_data = 32'hBB; // some test data
        in_bits = 8; // pack 8 bits
        #10;
        in_valid = 0;
        #10;

        // Test Case 3: Pack 12 bits into current register (less than remaining space)
        #10;
        in_valid = 1;
        in_data = 32'hCCC; // some test data
        in_bits = 12; // pack 12 bits
        #10;
        in_valid = 0;
        #10;

        // Test Case 4: Pack 8 bits into current register (filling the current register)
        #10;
        in_valid = 1;
        in_data = 32'hAA; // some test data
        in_bits = 8; // pack 8 bits
        #10;
        in_valid = 0;
        #10;

        // Test Case 5: Pack 20 bits into next(current) register)
        #10;
        in_valid = 1;
        in_data = 32'hEEEEE; // some test data
        in_bits = 20; // pack 20 bits
        #10;
        in_valid = 0;
        #10;

        // Test Case 6: Pack 16 bits when the current register is partially filled
        #10;
        in_valid = 1;
        in_data = 32'hFFFF; // some test data
        in_bits = 16; // pack 16 bits
        #10;
        in_valid = 0;
        #10;

        // Test Case 7: Pack 28 bits into current register (filling the current register)
        #10;
        in_valid = 1;
        in_data = 32'hAAAAAAA; // some test data
```

```
        in_bits = 28; // pack 28 bits
        #10;
        in_valid = 0;
        #10;


        // Test Case 8: Reset to check proper reset behavior
        #10;
        rst = 1;
        #10;
        rst = 0;
    end

    // Monitor output signals
    initial begin
        $monitor("Time = %t | in_valid = %b | in_data = %h | in_bits = %d | out_valid = %b
            | out_data = %h",
                $time, in_valid, in_data, in_bits, out_valid, out_data);
    end

endmodule
```