CE-325: Digital System Design

Homework 03 - Yosys Optimization

Huzaifah Tariq Ahmed - ha07151

10th May 2024

1 Key things to note

This Yosys optimization was done in UBUNTU Virtual Machine running on a Mac. No hierarchy commands were used as no other module files were in the directory in which I ran Yosys optimizations. I generated svg files of both the fsm and non fsm fifo after each and every command to be able to see the optimizations carried out by each command. I have attached all the svgs generated as well as the script files of the terminal.

2 Optimization Commands Used

2.1 Script

Used to record terminal sessions into a file, capturing inputs, outputs, errors, and timestamps. It's used for logging sessions, troubleshooting, documentation, and replaying actions.

2.2 Proc

The proc command in Yosys is used for optimizing Verilog designs by performing various transformations and simplifications. In the context of optimizing your FIFO design:

Purpose: proc optimizes the design by simplifying logic, reducing redundant elements, and improving performance. Usage: In your Yosys workflow, you would use proc after synthesizing your FIFO design to improve its efficiency and reduce resource usage. Effects: Logic Simplification: Reduces unnecessary logic gates and simplifies complex expressions. Resource Reduction: Optimizes resource utilization, leading to a more efficient design. Performance Improvement: Can enhance the speed and timing characteristics of the design.

2.3 Opt Muxtree

The opt muxtree command in Yosys optimizes multiplexer trees by simplifying and reducing their complexity. In the context of optimizing your FIFO design:

Purpose: Reducing Area: It helps minimize the area footprint of the design by simplifying multiplexer structures. Improving Timing: Simplified multiplexers can lead to better timing performance in your FIFO. Usage: After synthesizing your FIFO design in Yosys, you can use opt muxtree to optimize the multiplexer trees within the design. Benefits: Efficient Resource Usage: It optimizes the use of

multiplexers, potentially reducing the number of logic elements required. Faster Operation: Simplified multiplexers can result in faster data access and processing in your FIFO.

2.4 Opt Expr

The opt expr command in Yosys optimizes expressions in Verilog code, including optimizations related to FIFOs. It can improve the efficiency and performance of your FIFO design by optimizing expressions within the FIFO module, potentially reducing area and increasing speed.

2.5 Opt Merge

The opt merge command in Yosys workflow merges consecutive optimization passes into a single pass, helping to optimize designs more efficiently. This command is relevant when optimizing a FIFO design to improve its performance and resource utilization.

2.6 Opt RMDFF

The opt rmdff command in the Yosys workflow is used to optimize flip-flops in Verilog code, especially in designs like FIFOs. It removes flip-flops that are not necessary for functionality, helping to streamline and improve the efficiency of your FIFO design.

2.7 Opt Clean

The opt clean command in the Yosys workflow is used to perform a clean optimization on designs, including FIFOs. It aims to optimize the design by removing redundant logic, simplifying expressions, and optimizing resource usage. This command is particularly helpful in improving the efficiency and performance of FIFO implementations in your Verilog code.

2.8 Memory

The memory command in Yosys is used to define and optimize memories in Verilog designs. It's particularly useful when optimizing FIFO (First-In-First-Out) designs. You can use the memory command to specify the properties of your FIFO, such as depth, width, read and write ports, and optimization constraints. This helps Yosys to optimize memory-related structures efficiently, improving the performance and area utilization of your FIFO design.

2.9 Flatten

The flatten command in Yosys is used to simplify the hierarchy of a design by merging modules into a single flat design. This is helpful for optimizing designs, including FIFOs, as it reduces hierarchy overhead and can improve synthesis results.

2.10 Opt

The opt command in the Yosys workflow optimizes Verilog code by applying various transformations and simplifications to improve performance or reduce resource usage. When optimizing your FIFO design with Yosys, opt can be used to streamline the implementation, reduce area usage, or enhance timing characteristics.

3 Step by Step Yosys Workflow

Linux Terminal Commands

```
$ script fifo_non_fsm.sh
$ yosys
$ read_verilog fifo_non_fsm.v
$ proc
$ opt_muxtree
$ opt_expr
$ opt_merge
$ opt_rmdff
$ opt_clean
$ memory
$ flatten
$ opt
```

SVG Generation Command

```
$ show -format svg -prefix file_name fifo_non_fsm
```

4 FIFO NON FSM

4.1 Reduction in File Size

Filename	Description	Size
Original Non FSM	Graphviz representation without any optimization	122kB
Proc Non FSM	Graphviz representation after Proc Command	2.1MB
Opt Muxtree Non FSM	Graphviz representation after Opt Muxtree Command	2MB
Opt Expr Non FSM	Graphviz representation after Opt Expr Command	2MB
Opt Merge Non FSM	Graphviz representation after Opt Merge Command	1.7MB
Opt RMDFF Non FSM	Graphviz representation after Opt RMDFF Command	1.7MB
Opt Clean Non FSM	Graphviz representation after Opt Clean Command	830KB
Memory Non FSM	Graphviz representation after Memory Command	830KB
Flatten Non FSM	Graphviz representation after Flatten Command	830KB
Opt Non FSM	Graphviz representation after Opt Command	830KB

Table 1: FIFO NON FSM Optimization

4.2 Modified Code

```
`timescale 1ns/1ps

// Define the module named fifo_non_fsm with specified input and output ports
module fifo_non_fsm(
    Data_out,
    stack_full,
```

```
    stack_empty,
    Data_in,
    write_to_stack,
    read_from_stack,
    clk,
    rst
);

    // Parameters defining the width and height of the stack, and pointer width
    parameter stack_width = 8;
    parameter stack_height = 32;
    parameter stack_ptr_width = 5;

    // Output ports
    output [stack_width-1:0] Data_out;
    output stack_full, stack_empty;

    // Input ports
    input [stack_width-1:0] Data_in;
    input write_to_stack, read_from_stack, clk, rst;

    // Registers for read and write pointers, and pointer gap
    reg [stack_ptr_width-1:0] read_ptr, write_ptr;
    reg [stack_ptr_width:0] ptr_gap;

    // Register to hold data read from the stack
    reg [stack_width-1:0] Data_out;

    // Memory array representing the stack
    reg [stack_width-1:0] stack [stack_height-1:0];

    // Check if stack is full and empty
    assign stack_full = (ptr_gap == stack_height);
    assign stack_empty = (ptr_gap == 0);

    // Always block triggered on positive edge of clock or reset signal
    always@(posedge clk or posedge rst)
    if(rst)
    begin
        // Reset: Initialize data out, read pointer, write pointer,
        // and pointer gap
        Data_out <= 0;
        read_ptr <= 0;
        write_ptr <= 0;
        ptr_gap <= 0;
    end
    else if (write_to_stack && (!stack_full) && (!read_from_stack))
    begin
        // Write data into the stack if it's not full and not being read
        stack[write_ptr] <= Data_in;
```

```
57          write_ptr <= write_ptr + 1;
58          ptr_gap <= ptr_gap + 1;
59      end
60      else if ((!write_to_stack) && (!stack_empty) && read_from_stack)
61      begin
62          // Read data from the stack if it's not empty and being read
63          Data_out <= stack[read_ptr];
64          read_ptr <= read_ptr + 1;
65          ptr_gap <= ptr_gap - 1;
66      end
67      else if (write_to_stack && read_from_stack && stack_empty)
68      begin
69          // Write data into the stack if it's empty and
70          // being read and written simultaneously
71          stack[write_ptr] <= Data_in;
72          write_ptr <= write_ptr + 1;
73          ptr_gap <= ptr_gap + 1;
74      end
75      else if(write_to_stack && read_from_stack && stack_full)
76      begin
77          // Read data from the stack if it's
78          // full and being read and written simultaneously
79          Data_out <= stack[read_ptr];
80          read_ptr <= read_ptr + 1;
81          ptr_gap <= ptr_gap - 1;
82      end
83      else if(write_to_stack && read_from_stack && (!stack_full) && (!stack_empty))
84      begin
85          // Read data from the stack and write
86          // new data simultaneously if it's not full or empty
87          Data_out <= stack[read_ptr];
88          stack[write_ptr] <= Data_in;
89          read_ptr <= read_ptr + 1;
90          write_ptr <= write_ptr + 1;
91      end
92  endmodule
```

5 FIFO FSM

5.1 Reduction in File Size

Filename	Description	Size
Original FSM	Graphviz representation without any optimization	149kB
Proc FSM	Graphviz representation after Proc Command	9MB
Opt Muxtree FSM	Graphviz representation after Opt Muxtree Command	8.8MB
Opt Expr FSM	Graphviz representation after Opt Expr Command	8.4MB
Opt Merge FSM	Graphviz representation after Opt Merge Command	7MB
Opt RMDFF FSM	Graphviz representation after Opt RMDFF Command	7MB
Opt Clean FSM	Graphviz representation after Opt Clean Command	1.8MB
Memory FSM	Graphviz representation after Memory Command	1.8MB
Flatten FSM	Graphviz representation after Flatten Command	1.8MB
Opt FSM	Graphviz representation after Opt Command	1.7MB

Table 2: FIFO FSM Optimization

5.2 Modified Code

```
// Set the timescale for simulation
`timescale 1ns/1ps

// Define the FIFO FSM module
module fifo_fsm(Data_out,
                stack_full,
                stack_empty,
                Data_in,
                write_to_stack,
                read_from_stack,
                clk,
                rst);

    // Parameters defining the width, height, and pointer width of the stack
    parameter stack_width = 8;
    parameter stack_height = 32;
    parameter stack_ptr_width = 5;

    // Output registers for data out and stack status
    output reg [stack_width-1:0] Data_out;
    output stack_full,stack_empty;

    // Input ports for data in, write and read signals, clock, and reset
    input [stack_width-1:0] Data_in;
    input write_to_stack, read_from_stack,clk,rst;

    // Define states of the FSM
    parameter S_reset = 0;
```

```
29      parameter S_idle = 1;
30      parameter S_read = 2;
31      parameter S_write = 3;
32
33      // Registers for FSM state and next state
34      reg [2:0] state,next_state;
35
36      // Registers for read and write pointers, and pointer gap
37      reg [stack_ptr_width-1:0] read_ptr, write_ptr;
38      reg [stack_ptr_width:0] ptr_gap;
39
40      // Memory array representing the stack
41      reg [stack_width-1:0] stack [stack_height-1:0];
42
43      // Calculate stack_full and stack_empty signals
44      assign stack_full = (ptr_gap == stack_height);
45      assign stack_empty = (ptr_gap == 0);
46
47      // Sequential logic for FSM state transition
48      always@(posedge clk)
49        if(rst == 1) state <= S_reset;
50        else state <= next_state;
51
52      // Combinational logic for FSM behavior based on current state and inputs
53      always@(state or write_to_stack or read_from_stack or Data_in)
54        begin
55          // State machine behavior using case x statement
56          casex(state)
57            S_reset: begin
58                       Data_out = 0;
59                       read_ptr = 0;
60                       write_ptr = 0;
61                       ptr_gap = 0;
62                       if(rst)
63                         next_state = S_reset;
64                       else if((write_to_stack == 0) && (read_from_stack == 0))
65                         next_state = S_idle;
66                       else if((write_to_stack == 1) && (read_from_stack == 0))
67                         next_state = S_write;
68                       else if((write_to_stack == 0) && (read_from_stack == 1))
69                         next_state = S_read;
70                     end
71            S_idle: begin
72                       Data_out = 0;
73                       if(rst)
74                         next_state = S_reset;
75                       else if((write_to_stack == 0) && (read_from_stack == 0))
76                         next_state = S_idle;
77                       else if((write_to_stack == 1) && (read_from_stack == 0))
78                         next_state = S_write;
```

```
                    else if((write_to_stack == 0) && (read_from_stack == 1))
                      next_state = S_read;
                  end
          S_write:begin
                    if(write_to_stack && (!stack_full))
                      begin
                        stack[write_ptr] = Data_in;
                        write_ptr = write_ptr + 1;
                        ptr_gap = ptr_gap + 1;
                      end

                    if(rst)
                      next_state = S_reset;
                    else if((write_to_stack == 0) && (read_from_stack == 0))
                      next_state = S_idle;
                    else if((write_to_stack == 1) && (read_from_stack == 0))
                      next_state = S_write;
                    else if((write_to_stack == 0) && (read_from_stack == 1))
                      next_state = S_read;
                  end
          S_read:begin
                    if(read_from_stack && (!stack_empty))
                      begin
                        Data_out = stack[read_ptr];
                        read_ptr = read_ptr + 1;
                        ptr_gap = ptr_gap - 1;
                      end

                    if(rst)
                      next_state = S_reset;
                    else if((write_to_stack == 0) && (read_from_stack == 0))
                      next_state = S_idle;
                    else if((write_to_stack == 1) && (read_from_stack == 0))
                      next_state = S_write;
                    else if((write_to_stack == 0) && (read_from_stack == 1))
                      next_state = S_read;
                  end
          default: next_state = S_idle;

        endcase
    end
endmodule
```

6 SVG File and Script Files Folder

Click here to view the folder.