ELEC6233 – High level Synthesis

Nahom Kahsay Gebremichael

Ng3u24@soton.ac.uk

European Masters in Embedded Computer Systems

Personal Academic Tutors Name: Dr Basel Halak

Abstract:

After designing the control/data graph, operations were outlined, and allocation and binding were performed, ensuring the use of only one multiplier and one ALU. An RTL-level description was developed using System Verilog. Since the slide button is a level-sensitive switch without debouncing, a debouncing algorithm was implemented. The system design constraint (.sdc) file was added from the timing analyzer to ensure positive slack in hold and setup times. A dedicated 50 MHz clock was used to minimize jitter. After synthesis, state machine transitions and hardware blocks were verified. Once all specifications were met, the .sof file was uploaded, and testing was successfully completed.

1.1 Introduction

The objective of this assignment is to design a complex Fast Fourier Transform (FFT) butterfly algorithm, develop a high-level synthesis strategy for efficient resource allocation and binding, implement it on an FPGA, address challenges associated with practical circuits and systems, and document the design in a technical report.

The design process began with understanding the butterfly algorithm and signed 2's complement multiplication. A control and data flow graph was then developed, followed by resource binding, ensuring the use of a single ALU and one multiplier.

The technical report provides a comprehensive overview of the system architecture, including the controller, data path, and resource binding. It also details waveform simulations in Model Sim, describes individual hardware blocks, and discusses challenges encountered during FPGA testing.

1.2 Overall architecture of the design

The system is designed using a state machine-based controller. The controller operates based on a state machine where some transitions are triggered by the handshaking signal, while others occur automatically. The state machine generates control signals that are sent to the data path, ensuring proper execution of operations.

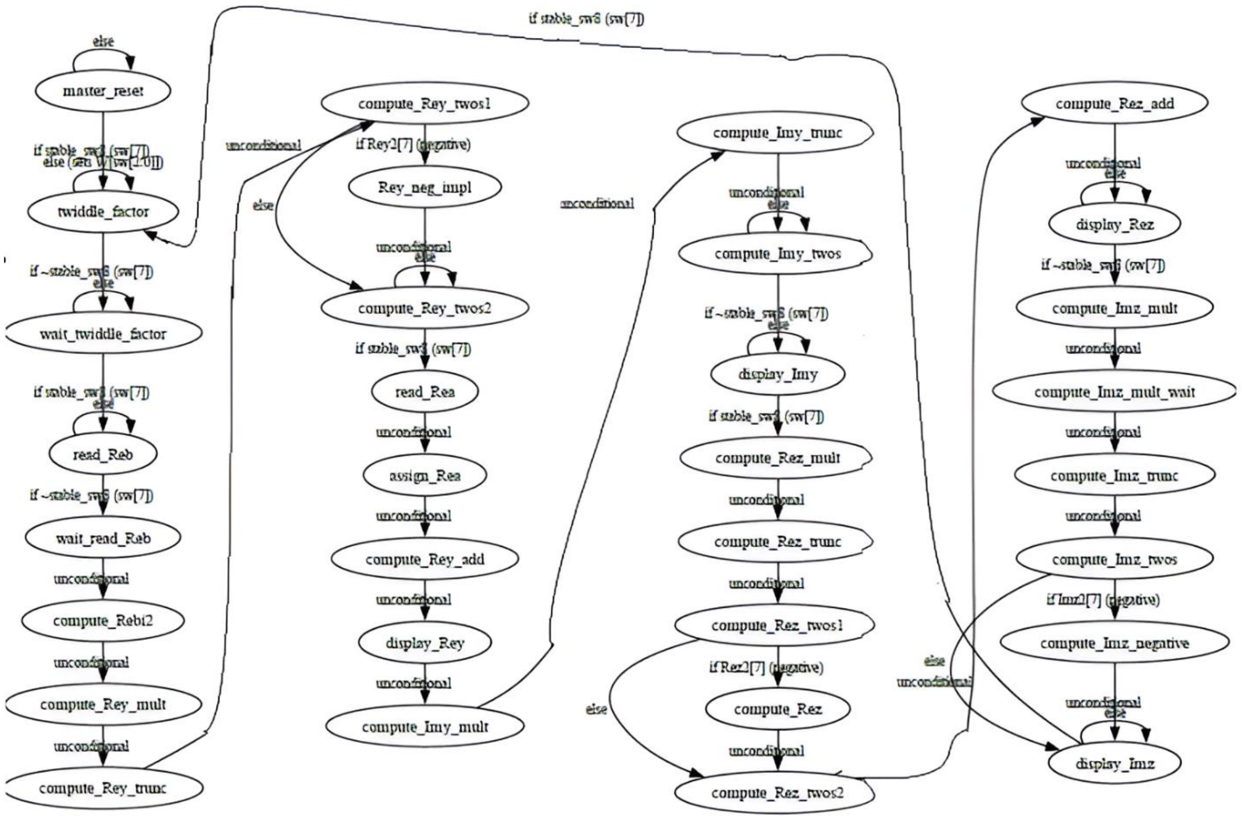


Figure 1.1 Controller state machine design

The data path, along with resource bindings and control signals, is illustrated in Figure 1.2. The resource binding consists of registers that store the values of Rew and Imw, which are selected from the switches. These values are then sent to a multiplexer, which determines the appropriate value to transmit based on a control signal from the controller. Similarly, the values Reb and Rea are stored in registers and fed into a second multiplexer, which selects the value to send based on another control signal.

Once the multiplication is computed, the output is stored in registers, and the values of these registers are fed into a third, larger multiplexer. This multiplexer selects which input to provide to one of the ALU inputs based on a control signal. The second input of the ALU is determined by another multiplexer, which selects between Rea and the value 1, with 1 being chosen when the ALU is performing a two's complement operation. After the ALU completes its computation, the results are stored in a register and subsequently assigned to the LED display

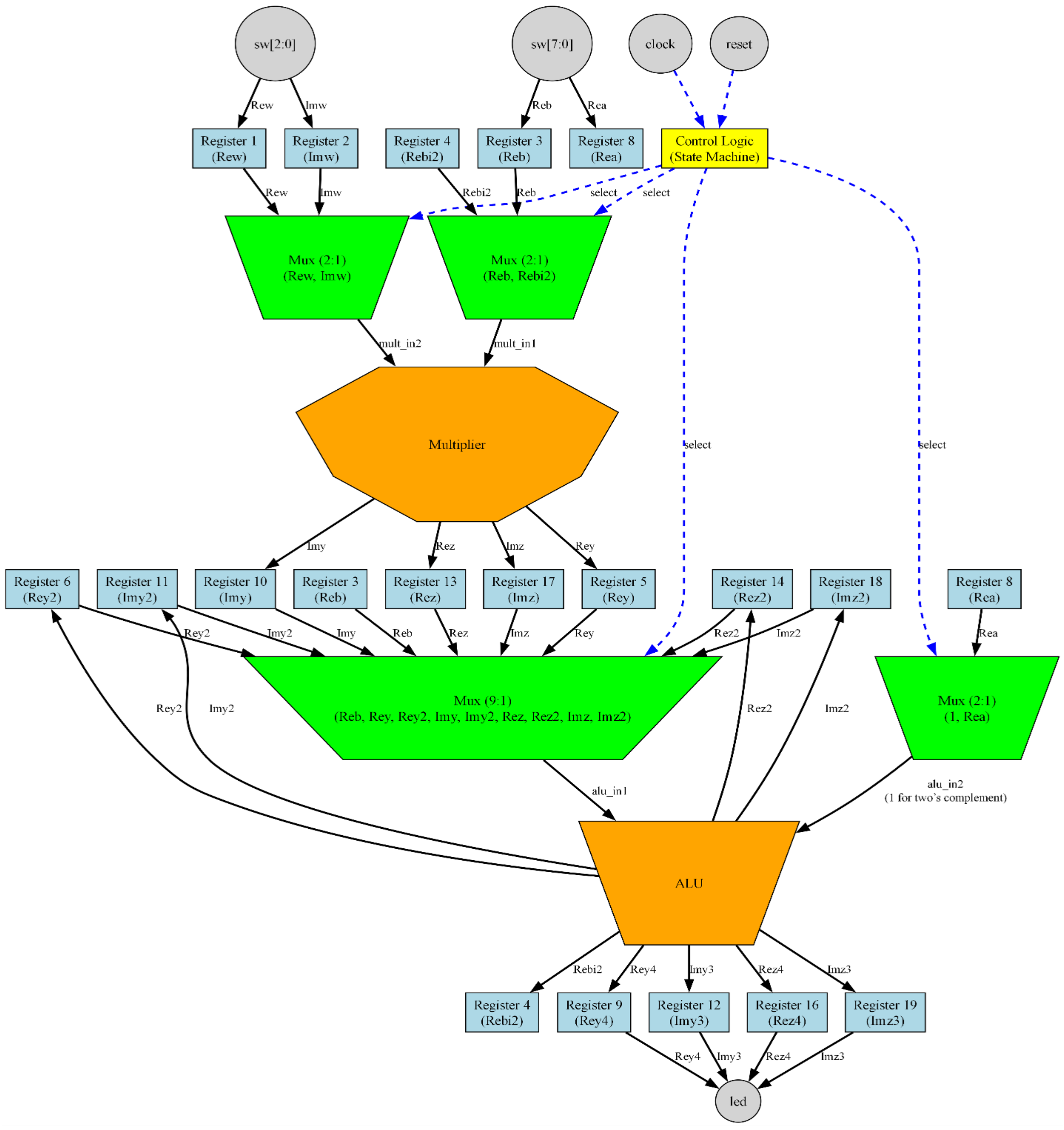
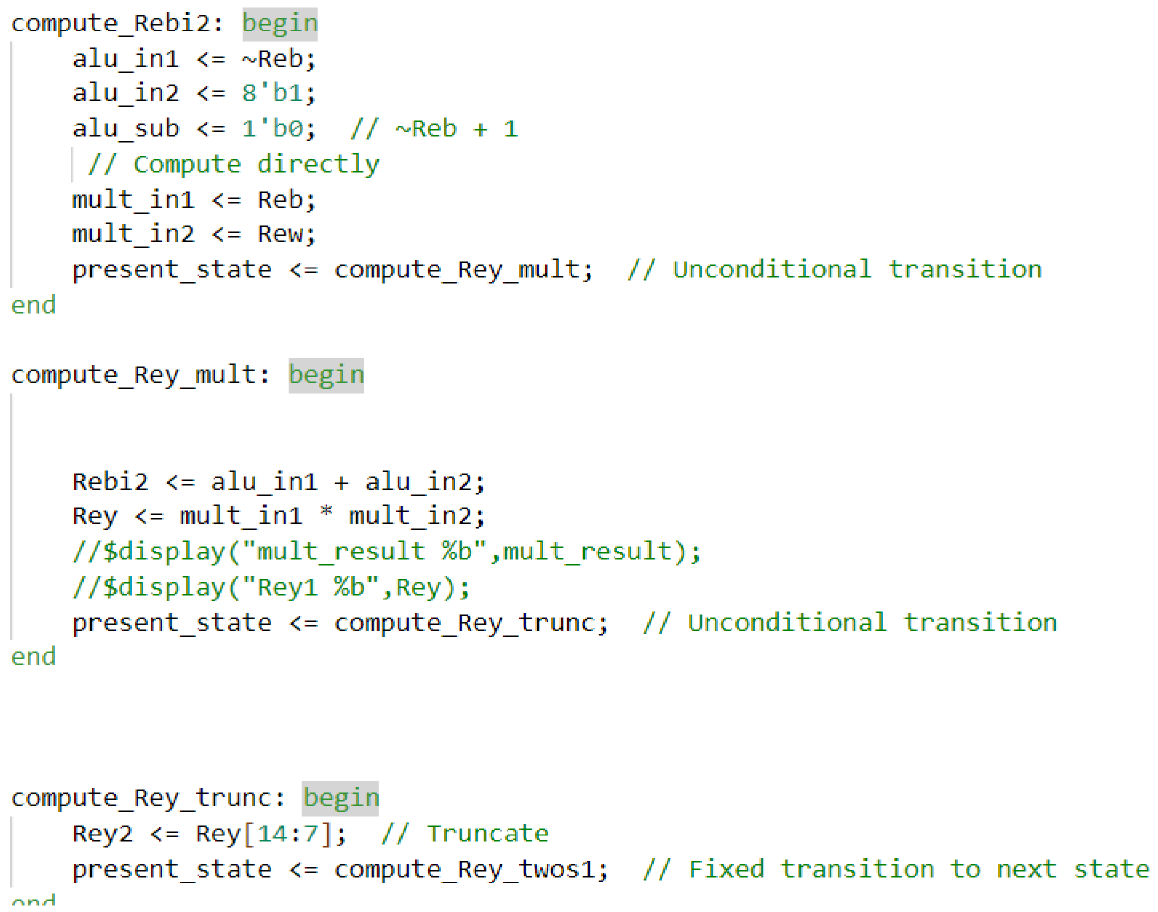


Figure 1.2 Data path with resource allocation, binding and control signals



```
compute_Rebi2: begin
    alu_in1 <= ~Reb;
    alu_in2 <= 8'b1;
    alu_sub <= 1'b0;  // ~Reb + 1
     // Compute directly
    mult_in1 <= Reb;
    mult_in2 <= Rew;
    present_state <= compute_Rey_mult;  // Unconditional transition
end

compute_Rey_mult: begin


    Rebi2 <= alu_in1 + alu_in2;
    Rey <= mult_in1 * mult_in2;
    //$display("mult_result %b",mult_result);
    //$display("Rey1 %b",Rey);
    present_state <= compute_Rey_trunc;  // Unconditional transition
end



compute_Rey_trunc: begin
    Rey2 <= Rey[14:7];  // Truncate
    present_state <= compute_Rey_twos1;  // Fixed transition to next state
end
```

Figure 1.3 Code snippet of the hardware description language

A sample code snippet of the main implementation is presented in Figure 1.3. As observed in the code, instead of performing additions directly, the two operands are first assigned to alu_in1 and alu_in2. The ALU then processes these variables, ensuring that all subsequent operations requiring the ALU follow the same assignment approach. This methodology guarantees that only a single ALU is inferred in the design.

For multiplication operations, the multiplicand is first assigned to mult_in1, while the multiplier is assigned to mult_in2. The multiplication result is then stored in the corresponding registers. This approach ensures that only one multiplier is inferred in the design.

Each operation is executed within a single state. Although addition and multiplication can be performed in the same state, multiple instances of the same operation, such as two multiplications or two additions, cannot be executed simultaneously due to the constraint of having only one ALU and one multiplier available.

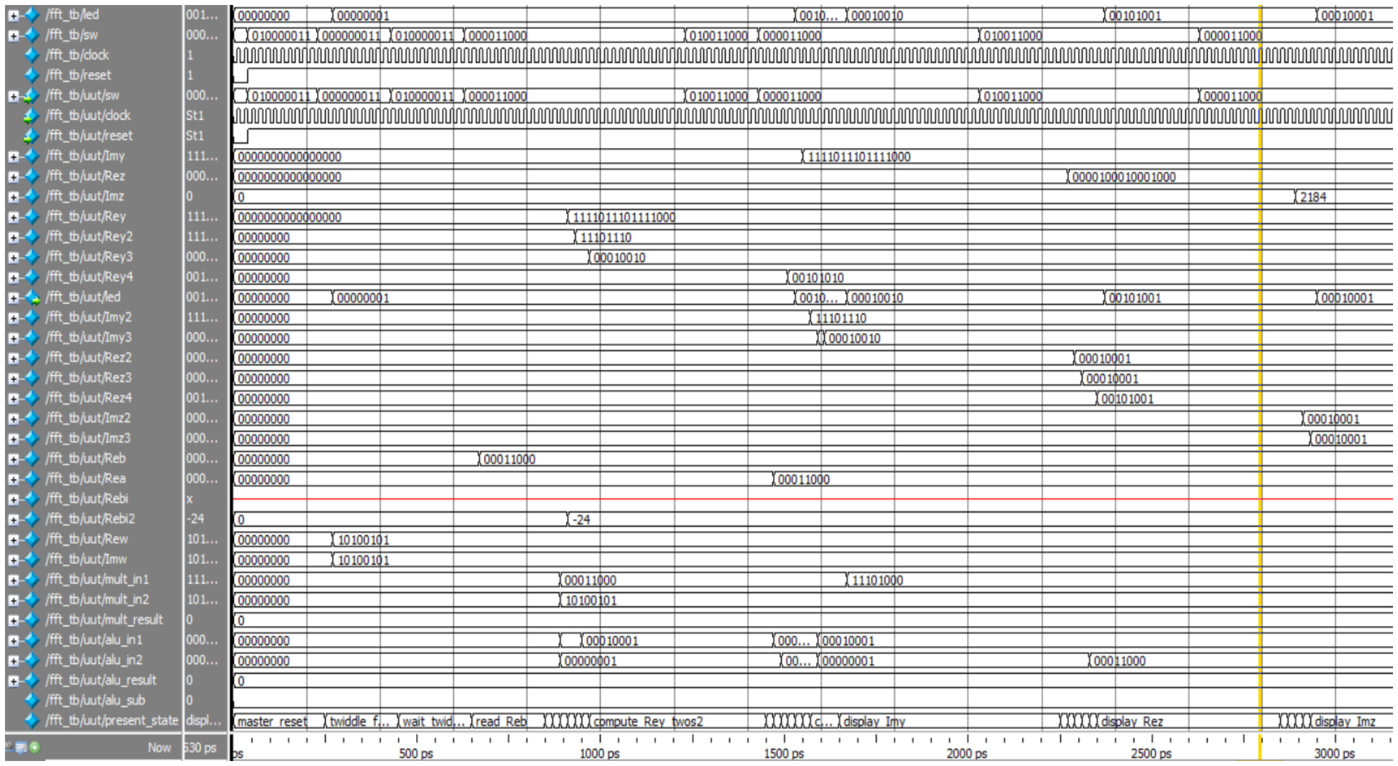


Figure 1.4 Simulation output from model sim

After writing the main module and the testbench, the design was simulated in ModelSim, and the waveform output is presented in Figure 1.4. The sw values were modified to replicate the real behavior of the FPGA. Initially, the reset signal was deasserted by transitioning from low to high, placing the system in the master_reset state.

Next, values were assigned to sw[2:0], followed by setting the handshaking signal high. The system then transitioned to the twiddle_factor state, where the values of sw[2:0] were assigned to predefined literals. Based on the switch values, the corresponding Rew and Imw twiddle factors were selected. Once assigned, the handshaking signal was set low, transitioning the system to the wait_twiddle_factor state.

Following this, the Reb values were set using the switches, and the handshaking signal was set high, prompting the transition to the read_reb state. In this state, the Reb value was read from the switches. The system then progressed through states responsible for computing Rey and Rebi2, followed by truncating and storing the result in Rey2. Subsequently, a conditional state checked whether the most significant bit (MSB) of Rey2 was 1. If so, the system transitioned to the next state for further calculations.

Next, the handshaking signal was set low, and the switch values were updated to represent Rea. Once the handshaking signal was set high, the system assigned the values to Rea and transitioned through multiple states. These states, operating sequentially without conditions, automatically progressed upon completing their respective computations. The system calculated Rey2, Rey3, and Rey4, with the final computed value, Rey4, being assigned to the LEDs for display.

Following this, the handshaking signal was set low, and the system transitioned through states responsible for computing Imy2 and Imy3, eventually assigning Imy3 to the LEDs for display.Subsequently, when the handshaking signal was set high, the system computed Rez, Rez2, Rez3, and Rez4, with Rez4 being displayed on the LEDs.

Finally, the handshaking signal was set low, and the system computed Imz, Imz2, and Imz3, with Imz3 displayed on the LEDs. As soon as the handshaking signal was set high again, the system returned to the twiddle_factor state, restarting the process from the beginning.

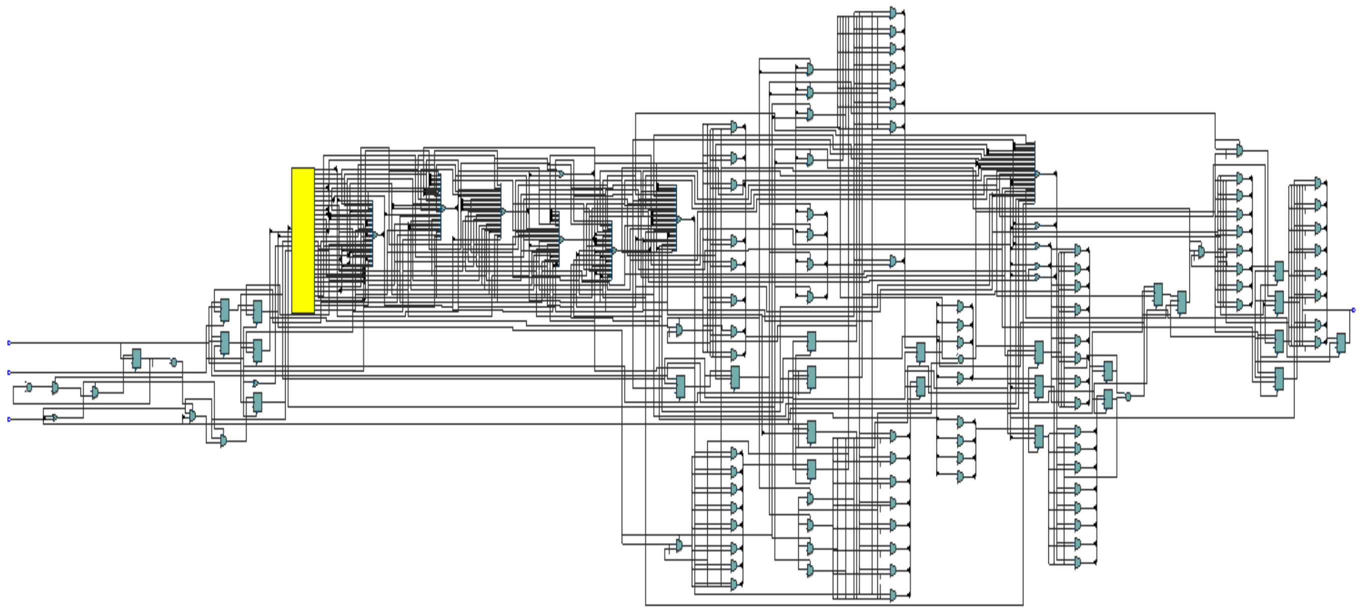


Figure 1.4 RTL based hardware blocks from Quartus

The RTL-level design generated by Quartus is shown in Figure 1.4. The RTL representation illustrates the hardware components, including multiplexers, the multiplier, the ALU, selectors, and registers, providing a detailed view of the synthesized design. The complete RTL output, along with its detailed structural breakdown, can be found in the attached PDF file.

1.3 Details of individual hardware blocks

1.3.1 Register Blocks

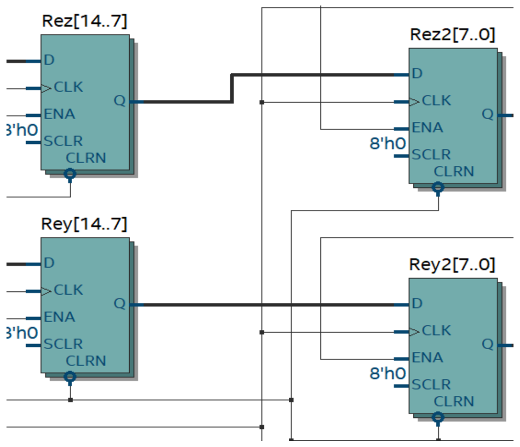


Figure 1.5 The inferred Registers

After synthesizing the hardware blocks at the RTL level in Quartus, the inferred registers can be observed in Figure 1.5. A sample of the inferred registers for Rez, Rez2, Rey, and Rey2 is shown in Figure 1.5. The inferred values for the remaining registers follow the same pattern and structure.

1.3.2 ALU Block

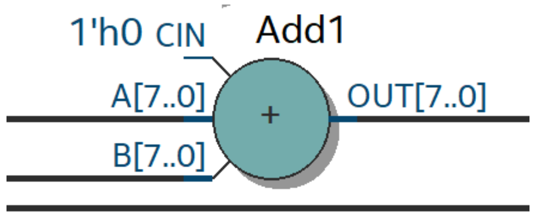


Figure 1.6 The inferred ALU block

The exclusive ALU block inferred by Quartus can be seen in Figure 1.6. Due to the design of the code, only a single ALU is inferred. The values fed to the ALU are stored in the alu_in1 and alu_in2 registers. These values are switched by the multiplexers, which are controlled using control signals from the controller, as determined by the state machine.

1.3.3 Multiplier Block

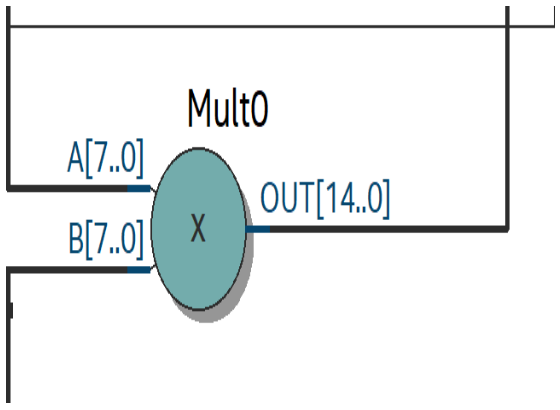
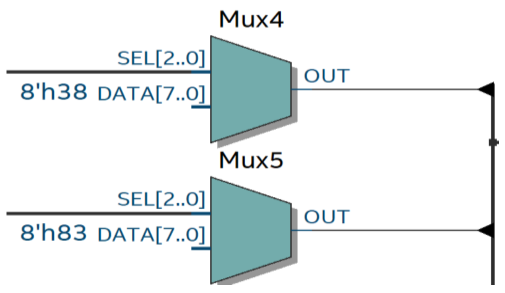


Figure 1.7 Multiplier block

The exclusive Multiplier block, derived from the dedicated 9x9 DSP block inferred by Quartus, can be seen in Figure 1.7. Due to the structure of the code, only a single multiplier from the DSP block is inferred. The values fed to the multiplier are stored in the mult_in1 and mult_in2 registers. These values are switched by the multiplexers, which are controlled using control signals from the controller, as determined by the state machine.

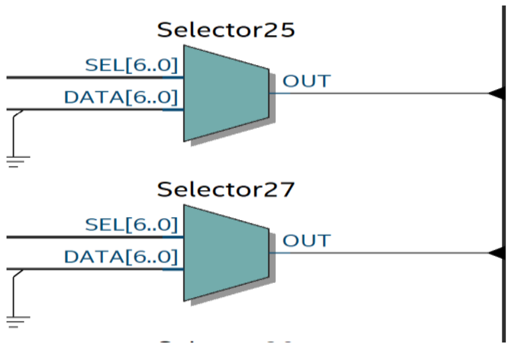
1.3.4 Multiplexer Blocks



The multiplexers generated in the RTL by Quartus are used to select the values of Imw and Rew from the switches, from the sw[2:0].They will receive input from the switches and assign the appropriate value of the twiddle value from the stored literals.

Figure 1.8 Multiplexers

1.3.5 Selector Blocks



The selector blocks inferred by quartus are used in selecting the bits for the Rey, Imy Rez and Imz registers, After the multiplication has been performed from the multiplexers which consist of Reb, Imw, Rew and Rea.

Figure 1.8 Multiplexers

1.4 FPGA implementation

After uploading the code to the FPGA, it was observed that when the handshaking switch was set to high, the states transitioned multiple times erratically without waiting for another signal from the handshaking slide switch. This behavior was caused by the bouncing of the switch values.

Since the slide switches are directly connected to the Cyclone V SoC, unlike the pushbuttons which are equipped with a Schmitt trigger debouncing circuit [1], the bouncing caused unintended state transitions.

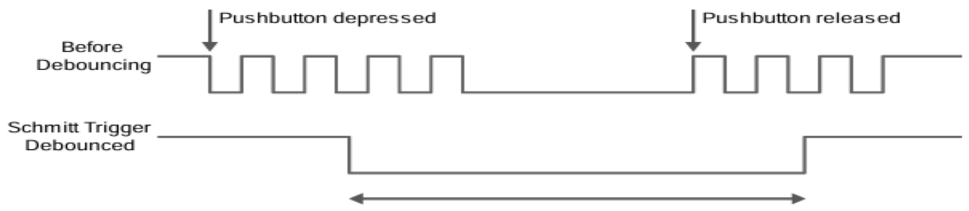


Figure 1.10 Schmitt trigger based debouncing for the push buttons, figure adopted from [1]

To address the issue of bouncing, a debouncing logic was implemented in place of the Schmitt trigger for this design. The debouncing logic begins by counting 2,500,000 cycles, which corresponds to a 50ms delay. This ensures that the system waits for 50ms, allowing the switch bouncing to settle. After this time, the debouncing logic assigns the stable switch value to a new variable, stable_switch, ensuring stable input values for the system

To address the timing issue, a timing analyzer was used, and a system design constraint file (SDC) was created. This file specifies delays for the switches and LEDs to ensure that the setup and hold slacks are positive, allowing data to arrive properly. A 1ns delay was added for the switches and LEDs. The worst-case setup slack was found to be 10.41ns, and the worst-case hold slack was 0.231ns for the slow 85°C model. The timing checks were also passed for the fast and 0°C models.

Due to an issue with the Sw[8] switch on the FPGA board, Sw[7] was used as the handshaking signal instead. The testing procedure is as follows: The reset switch (sw[9]) is set to high, which deasserts the reset. Then, the value of the twiddle factor is assigned using sw[2], sw[1], and sw[0]. The handshaking signal is set to high, followed by low, and then the value of Reb is assigned from switches sw[6:0]. The handshaking signal is set to high again, followed by low, and the value of Rea is assigned from sw[6:0]. Once the Rea values are set, the handshaking signal is set to high, and the Rey value is displayed on the LEDs. The process continues with the handshaking signal being alternated between high and low to display Imy, Rez, and Imz on the LEDs. Finally, after Imz is displayed, the handshaking signal is set to high, looping back to the twiddle factor state.

1.5 Conclusion

The complex fast Fourier butterfly algorithm was designed, synthesized, and successfully tested on an actual FPGA, adhering to the given resource constraints.

Through this project, I have gained a comprehensive understanding of the steps required for system design, starting from requirement analysis to high-level synthesis, scheduling, binding, and FPGA testing. I have also learned to navigate the trade-offs necessary due to timing and resource requirements.

I have become proficient in hardware description languages such as System Verilog, with an emphasis on accounting for delays in actual circuitry. These delays must be addressed through proper setup and hold requirements in the timing analyzer. While simulations provide an ideal representation of the design, real-world implementations face challenges like switch bouncing and delays that must be handled.

In the future I plan to leverage automatic high-level synthesis tools to automate the design process, optimizing the code further by eliminating dead code and unrolling the loop. Additionally, I would incorporate lifetime analysis to share registers, reducing the need for individual registers for every initial, intermediate, and final value. This approach would conserve resources, decrease area and cost, and shorten production time when the system is implemented as an Application-Specific Integrated Circuit (ASIC). I have developed an appreciation for the tasks and overall steps involved in creating dedicated digital signal processing integrated circuits.

1.6 References

1. P. Y. K. Cheung, "DE1-SoC User Manual," Dept. Elect. Electron. Eng., Imperial College London, London, U.K., [Online]. Available:http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/de1-soc_user_manual.pdf
2. Intel Corporation, "Inferring Multipliers," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683082/22-1/inferring-multipliers.html
3. R. Hoffman, "Digital System Design," in *Proc. 1997 Conf.*, 1997.
4. All About Circuits, "All About Circuits," 2023. [Online]. Available: https://www.allaboutcircuits.com
5. Intel Corporation, "FPGA Forum," 2023. [Online]. Available: https://community.intel.com/t5/FPGA/ct-p/fpga
6. U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin, Germany: Springer, 2007.