

TABLE OF CONTENTS

1	INTRODUCTION.....	3
2	DESCRIPTION OF THE COMPANY.....	4
2.1	Company Name.....	4
2.2	Company Location	4
2.3	Company Profile	4
2.4	Brief History.....	6
2.5	Organizational Structure of the Company.....	6
3	PROJECT 0 – DC-DC CONVERTER.....	7
3.1	Setup.....	7
3.2	Theory.....	7
3.3	Proposed Design.....	9
3.4	Simulation Results	11
4	PROJECT 1 – DIGITAL ALARM CLOCK.....	14
4.1	Design	14
4.2	Implementation	16

4.3	Final Tests	17
4.4	Possible Improvements.....	19
5	PROJECT 2 – VERIFICATION	20
5.1	Theory.....	20
5.2	An Example Testbench.....	22
5.3	Source Files	31
6	PROJECT 3 – FFT IP.....	32
6.1	Introduction	32
6.2	Fourier Series	32
6.3	Fourier Transform.....	33
6.4	Fast Fourier Transform	34
6.5	Theory of Radix-2 DIT	35
6.6	Hardware Implementation	37
6.7	Overall Design.....	45
6.8	Controller	45
6.9	Test Results and Comments.....	47
7	CONCLUSIONS.....	50
I	SOURCE FILES.....	51
	REFERENCES.....	52

1 INTRODUCTION

I performed my EE300 summer practice in TÜBİTAK SAGE, one of the leading R&D companies in the defense industry. SAGE designs, develops and manufactures modern products for military purposes. My practice lasted for 5 weeks. On the first two days of my internship, I was in Analog Electronics Design Department and they want me to design and simulate a DC-DC converter whose details is described in Project 0 chapter of this report. To do that I learned about fundamentals of DC-DC converters and control of them. Then, I started to work in Digital Electronics Design Department. Here, I have learned Verilog and SystemVerilog. While learning Verilog, I developed a digital alarm clock whose details is in Project 1 chapter of this report. Then, I started to gain knowledge about verification models and functional verification and using SystemVerilog, I verified a library written by Adem Günesen, one of the engineers in the Digital Electronics Design Department. The verification models and theory are told in Project 2 chapter. Finally, I designed an FFT IP which copies the functions of Xilinx's FFTv9.0 IP. For this purpose, I first learned the basics of Fourier transform and FFT. Furthermore, I also learned the hardware architectures for the implementation of an efficient FFT IP. A detailed description of the theory, design, and implementation of my FFT IP can be read in Project 3 chapter.

This report begins with the description of the company and continues with the detailed description of the projects done. Finally, the report is concluded with a conclusion part. All the source files can be accessed from Appendix I chapter at the end of the report. Also, the sources used in this work are included in References.

2 DESCRIPTION OF THE COMPANY

2.1 Company Name

TÜBİTAK SAGE¹

2.2 Company Location

Address: Pk. 16 Mamak 06261 Ankara

Phone: 0 312 590 90 00

Fax: 0 312 590 91 48 - 49

2.3 Company Profile

TÜBİTAK Defense Industries Research and Development Institute (Turkish: TÜBİTAK Savunma Sanayii Araştırma ve Geliştirme Enstitüsü), shortly TÜBİTAK SAGE, is a Turkish institution carrying out research and development projects on defense industry technology.

¹ Defense Industries Research and Development Institute

The main function of SAGE is to perform research and development activities for defense systems including engineering and prototype production, starting with their fundamental research and conceptual design. Most of the projects are performed in coordination with related defense institutions.

Vision

Making Turkey independent in defense technologies.

Mission

Providing technology, products, and services with high competitive power and value-added to the defense industry through R&D.

Service Areas

Guidance, Control and Navigation Technologies, Flight Mechanics and Aerodynamic Design, Software, Modeling and Simulation, Mechatronics, Structural Mechanics, Electronic Design, Platform Integration, Energetic Materials, Pyrotechnic Systems, Advanced Materials Technologies, Warhead and Plug Technologies, Propulsion Systems Technologies, Environmental Testing Services, Wind Tunnel Testing Services and External Load Flutter Certification

Shareholders

- TÜBİTAK and TÜBİTAK Institutes,
- The Ministry of Science, Industry and Technology,
- SSB (Presidency of Defense Industries),
- Ministry of National Defense,
- TAF (Turkish Armed Forces),
- Strategy and Budget Directorate,
- MKEK (Mechanical and Chemical Industry Association),
- Subsidiaries of TSKGV (Turkish Armed Forces Foundation)
- Universities

Number of employees

- 501-1.000
- Number of engineers employed: 888

2.4 Brief History

It was established in 1972 at Beşevler neighborhood in central Ankara under the name "Guided Vehicles Technology and Measurement Center" (Güdümlü Araçlar Teknoloji ve Ölçüm Merkezi, GATÖM) by the Scientific and Technological Research Council of Turkey (Turkish: Türkiye Bilimsel ve Teknolojik Araştırma Kurumu, TÜBİTAK). In 1983, the institution was renamed to "Ballistics Research Institute" (Balistik Araştırma Enstitüsü, BAE), and finally, in 1988 it took its current name. The organization conducts research and development activities to meet the needs of the Turkish Armed Forces and the national defense industry.

SAGE moved in 1993 to its new location in Lalahan village of Elmadağ district, 30 km (19 mi) northeast of Ankara. The facility in Beşevler serves as Ankara Wind Tunnel. The institute also runs a site within the Middle East Technical University (ODTÜ).

2.5 Organizational Structure of the Company

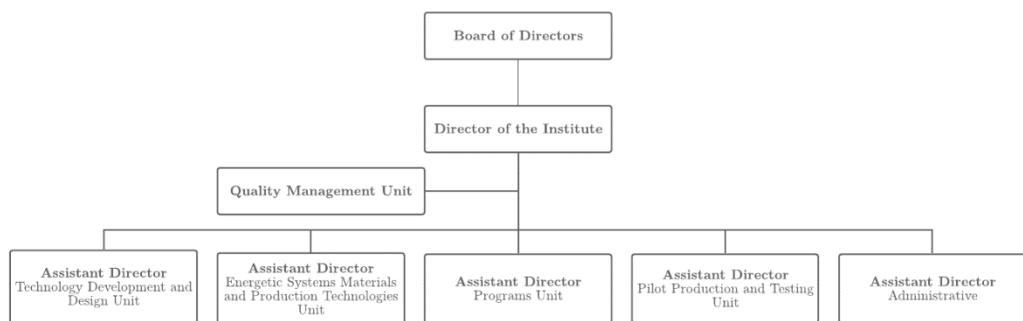


Figure 2.1 Organization chart of TÜBİTAK SAGE.

3 PROJECT 0 – DC-DC CONVERTER

3.1 Setup

In this project, a *DC-to-DC* converter with the following specifications is designed and examined:

- Input voltage: 28V
- Output voltage: 12V
- Switching frequency: 300kHz
- Maximum output current: 5A
- Controller to be used: LT3741

3.2 Theory

DC-DC converters are needed since all electronic devices require constant voltage supply and DC-DC converters are the circuits converting the unregulated DC input to a controlled DC output at the desired voltage level. They can be separated into two groups: linear and switcher type. Since a switching type regulator is implemented in this project, the details of linear type regulators will be skipped.

The *switching regulator* is a DC-DC converter that delivers power by using switcher components. The inductor stores and releases energy to output load and gets energy from the input source which is controlled by the switches.

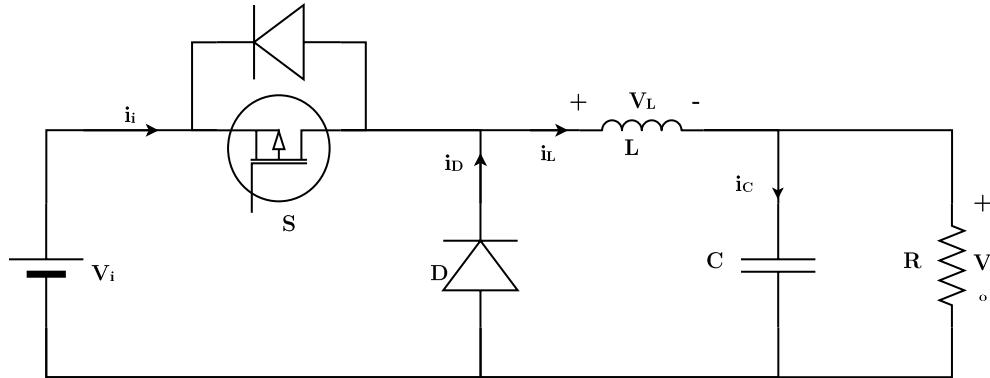


Figure 3.1 Circuit topology of the buck converter.

In Figure 3.1, when transistor S is on, the inductor is storing energy; when S is off, the inductor is releasing energy. Now, assume the state of the S is like in Figure 3.2.

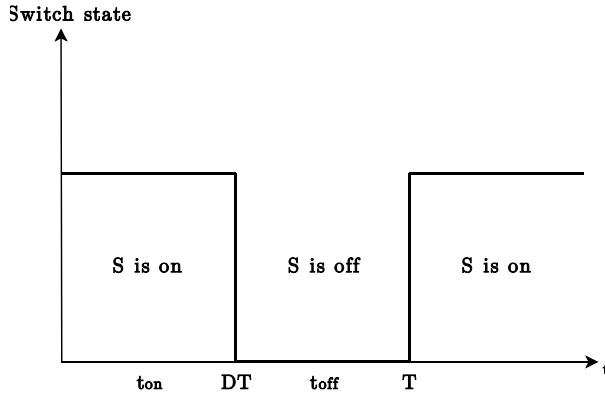


Figure 3.2 Switch state vs. time graph.

If we separate the cases S is on and S is off and then, analyze them separately we find that $D = V_o/V_i$. Note that, this is true when we assume the converter operates in continuous conduction mode and in the steady-state. And this, the formula we get implies that by changing D, output voltage level can be adjusted.

The switching function can be controlled by an external controller. A buck converter conducts for a short period of time and passes charge to a capacitor which is then connected through an inductor to the low voltage output. The low voltage output is feedback to the controller and when the output voltage decreases

to a threshold voltage, the controller will conduct again for another short period of time. The capacitor and inductor will smooth out the pulses and one will then have a smooth DC output. An example control circuitry is in Figure 3.3.

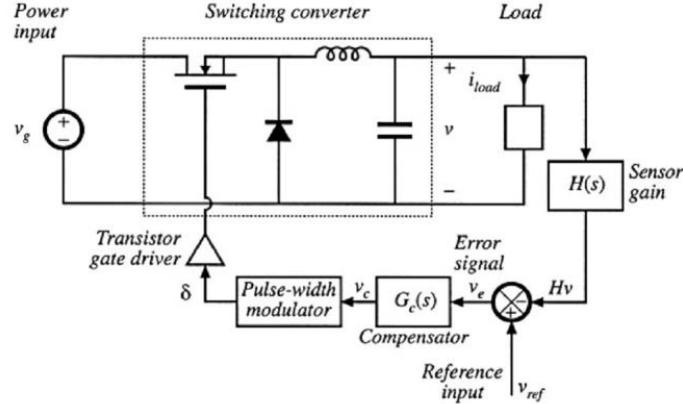


Figure 3.3 Controlling the buck converter.

3.3 Proposed Design

A synchronous buck regulator with controller LT3741 satisfying the desired specifications in Figure 3.4 is designed with the help of datasheet of the controller.

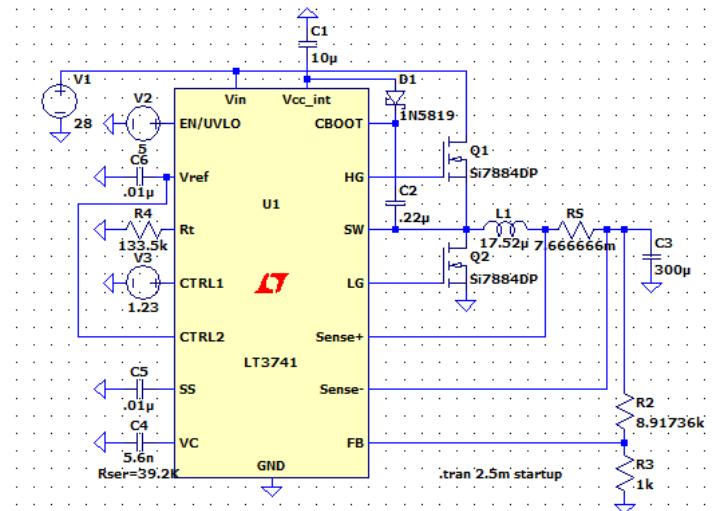


Figure 3.4 Proposed design for DC-DC converter.

In the datasheet (Analog Devices, 2010, p. 14) it is stated that

$$I_{max} = I_o + \frac{V_{IN}V_O - V_O^2}{2f_s L V_{IN}} = I_o + \frac{V_{IN}V_O - V_O^2}{2f_s \left(\frac{V_{IN}V_O - V_O^2}{0.3f_s I_o V_{IN}} \right) V_{IN}} = \frac{23}{20} I_o = \frac{23}{20} \frac{V_{ctrl1}}{30R_S} \quad (3.1)$$

Let choose $V_{ctrl1} = 1.23V$. From Equation (3.1), for desired outcomes, we must choose $R_S = \frac{23}{600 \times 5} = 7.6m\Omega$ and $L = \frac{V_{IN}V_O - V_O^2}{0.3f_s I_o V_{IN}} = 17.5238 \dots \mu F$.

And to divide voltage at the output, adjust R_2 and R_3 using the equation in the datasheet (Analog Devices, 2010, p. 16): $1.21(1 + R_2/R_3) = 12$. This implies $R_2/R_3 = 8.91736 \dots$ So, if we choose R_3 as 1K R_2 must be 8.91736K.

R_T must be around $143k\Omega$ for the desired switching frequency according to Table 4 in the datasheet (Analog Devices, 2010, p. 16). But I find its exact value for desired results experimentally.

After making necessary optimizations using the information in the datasheet, the resultant circuit design is in Figure 3.4 and this circuit topology gives the output characteristics as in Figure 3.5.

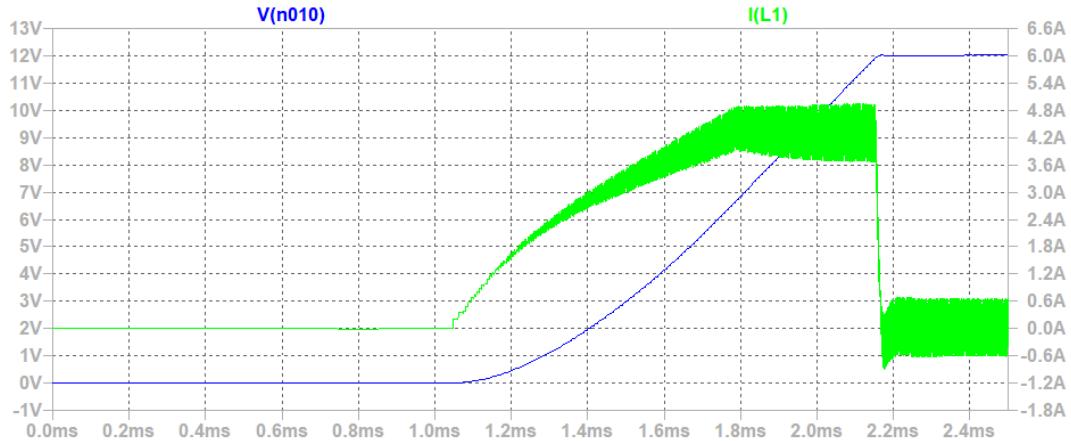


Figure 3.5 Output characteristics of the proposed design.

3.4 Simulation Results

Designed topology has been simulated using LTSpice, some of its characteristics has been observed and the circuit has been examined under different situations.

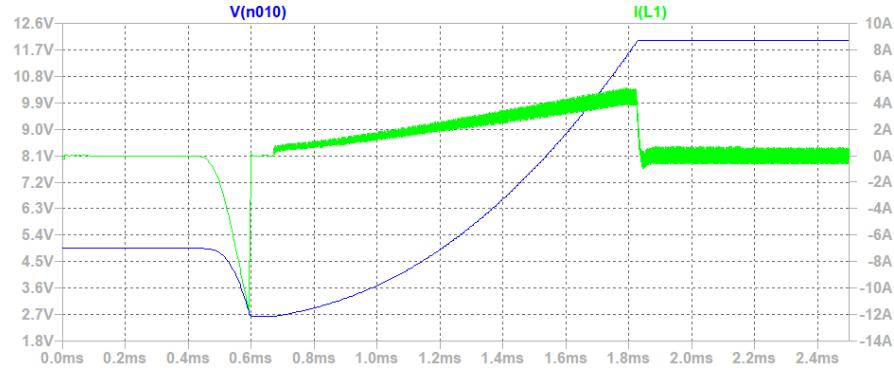


Figure 3.6 Output characteristics when $i_{L1}=0.5\text{mF}$.

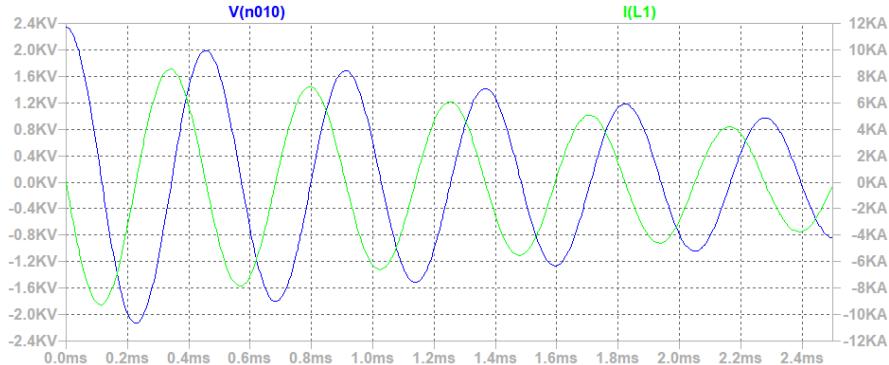


Figure 3.7 Output characteristics when $i_{L1}=1\text{mF}$.



Figure 3.8 Observing the ripple of the output waveform.

Using measurement function in LTSpice, we can find that

The ripple of the output waveform: 1.6931282mV

This means,

$$\text{ripple factor} = 1.6931282\text{mV}/12\text{V} = 1.41 \times 10^{-4}$$

Note that, this is very close to theoretical expectation since we know that

$$\text{ripple factor}_{\text{theoretical}} = \frac{1 - D}{8LCf^2} = \frac{1 - 12/28}{8(17.52 * 10^{-6})(300 * 10^{-6})(301.23 * 10^3)^2} = 1.49 * 10^{-4}$$



Figure 3.9 Measurement of the switching frequency.

And we can measure the switching frequency using two cursors as 301.22655KHz

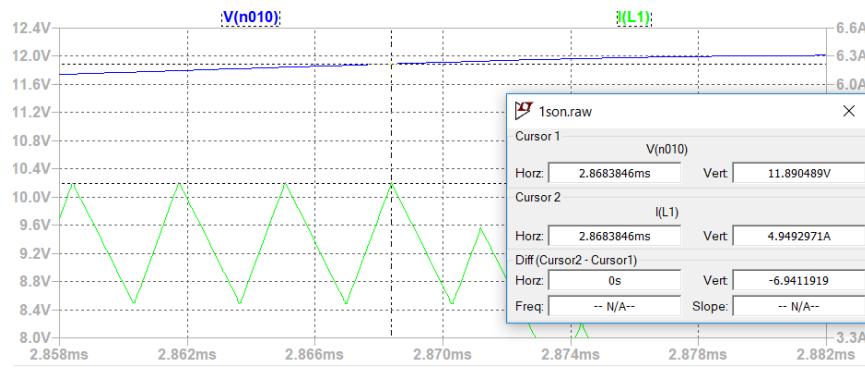


Figure 3.10 Measuring the output voltage and current.

Output voltage = 11.9V, output current = 4.95A

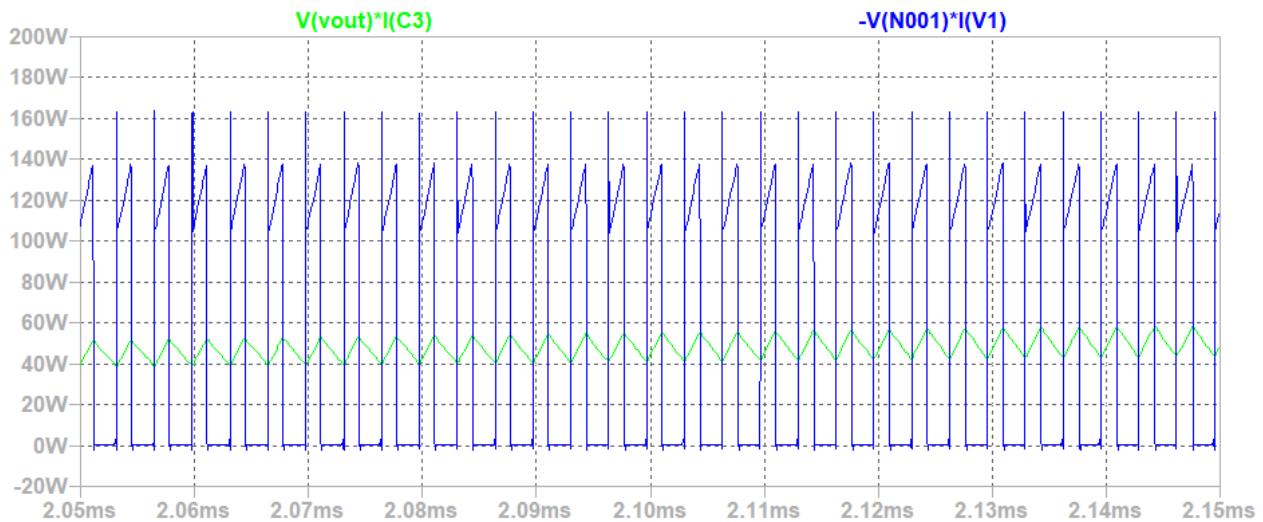


Figure 3.11 Measuring the system efficiency.

$$\text{System efficiency} = P_{\text{out}}/P_{\text{in}} = 47.994\text{W}/49.755\text{W} = 96.46\%$$

This is expected since the buck converters have very high efficiencies.

4 PROJECT 1 – DIGITAL ALARM CLOCK

In this project, a digital alarm clock is implemented using Verilog and it is simulated using QuestaSim. The clock has many functionalities: it can show the time, time and alarm time can be arranged by the user and it rings when the alarm time comes.

4.1 Design

By assembling the basic blocks as in Figure 4.1, the clock is created.

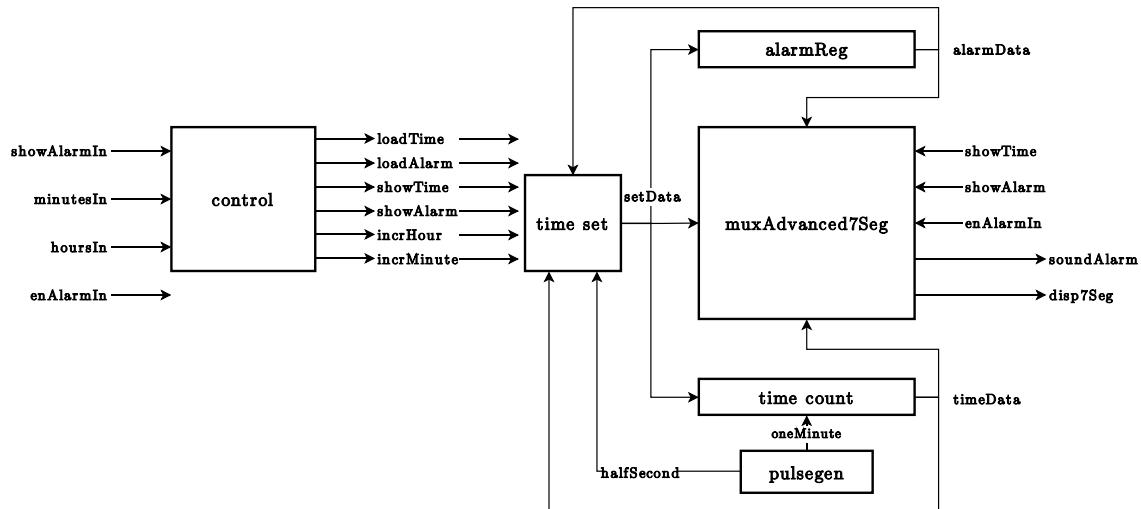


Figure 4.1 Internal structure of the topmost *digitalClock* module.

The functionalities of the basic blocks must be described.

Controller

The *control* module controls the whole process according to the FSM in Figure 4.2. To decide the state, the controller uses the *minutesIn*, *hoursIn* and *showAlarmIn* inputs. If the user activates the *showAlarmIn* input the clock must show the alarm time in its memory. And *minutesIn* and *hoursIn* inputs are for adjusting the time data. For example, by activating *minutesIn* signal, the time can be increased. To change alarm time, while *showAlarmIn* input is high, *minutesIn* or *hoursIn* signals must be activated. Note that, the outputs are the function of the current state and *rst_n* is an asynchronous active low reset signal.

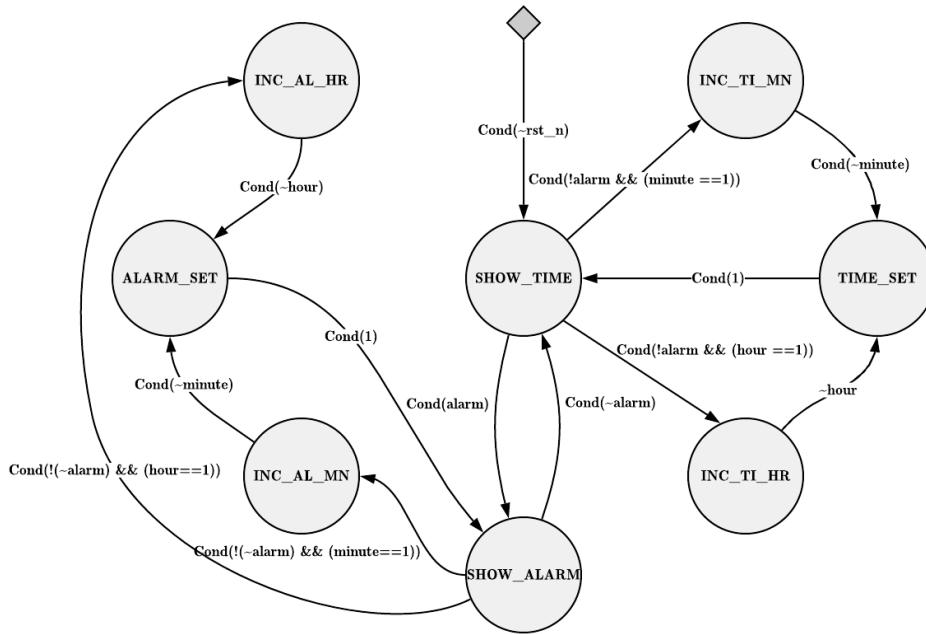


Figure 4.2 Finite-state machine diagram for control block.

Pulse Generator

The *pulsegen* block uses 13-bit wide counter to create *halfSecond* and *oneMinute* pulses from a 128Hz clock input. Each pulse has a duration of one clock cycle. The

counter starts at 7679 ($128\text{Hz} * 60 - 1$) and counts down on each active clock edge. When it gets to zero, it reloads back to 7679. And a combinatorial circuitry generates pulses in each minute and in each half a second using the counter.

Time Counter

The *time_count* module is a counter counts the time. It keeps the time in HH:MM format, so each digit must be represented using 4 bits. And it must increment the count at each oneMinute.

Alarm Register

The *alarmReg* block is a 16-bit wide register holding the alarm time of the clock.

Time Setter

The *time_set* module holds and increments the alarm and time while the user increments it.

Output Module

The *muxAdvanced7Segment* block outputs the time in 7-segment display format and alarm signal. It checks whether the alarm time is come to produce *soundAlarm* signal. Also, it shows alarm time or time by checking its *showAlarm* and *showTime* inputs.

4.2 Implementation

First, the basic blocks to be used are coded. I tried to use Test-Driven Development. The modules have their own testbenches and are tested separately. TDD makes the debugging process easier since finding an error by simulating the complex top module is much harder than finding the error in the test of basic modules. Then, I put together these basic blocks to create the topmost *digitalClock*

block. This report will not cover the HDL codes of this project but all the source files of this project are under “alarmclock” folder in [Appendix I](#).

4.3 Final Tests

To test the final digital clock module, first, start the alarm and test its resetting function as in Figure 4.3. Note that since in the beginning alarm time is zero, soundAlarm signal is high at the beginning of the test. And resetting is successful, disp7Seg is as expected. In Figure 4.3, d0, d1, d2, d3 signals are the fourth, third, second and first digit of the disp7Seg signal, respectively. So, d3 and d2 show the minute and d1 and d0 show the hour in its 7-segment representation. Note that, 3f is the 7-segment representation of 0 and the clock starts with 00:00. Then d0 becomes 06 (7-segment representation of 1) since the clock becomes 00:01.

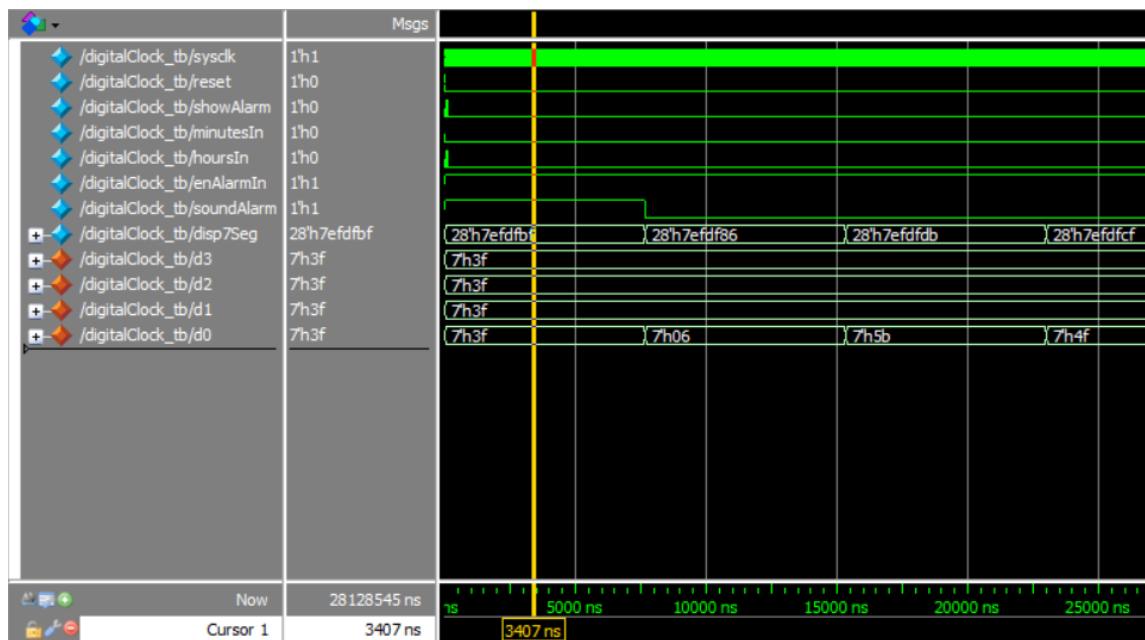


Figure 4.3 Starting the alarm and resetting it.

When we activate the soundAlarm signal, it stops showing the clock and begins showing the alarm time which is 00:00 since we do not set it after the reset as one can see from Figure 4.4.

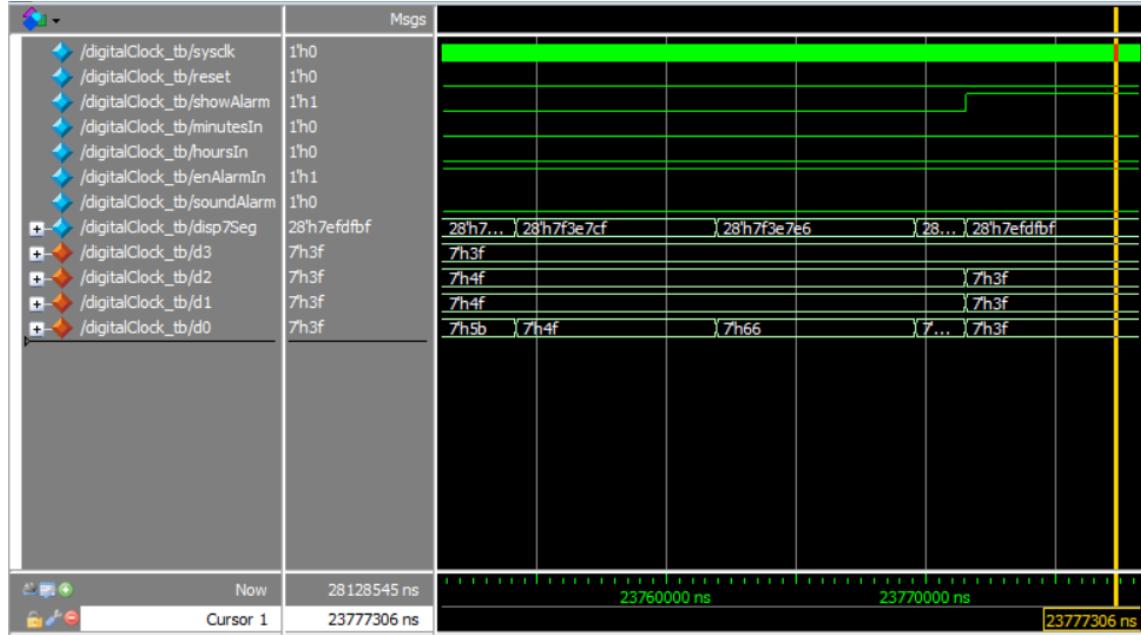


Figure 4.4 Testing showAlarm option.

To set the alarm, after activating *showAlarm* signal, *minutesIn* and *hoursIn* signals must be activated. As one can see from Figure 4.5, we change the alarm time to 07:16 (3f 07 : 06 7d) by using *minutesIn* and *hoursIn* signals.

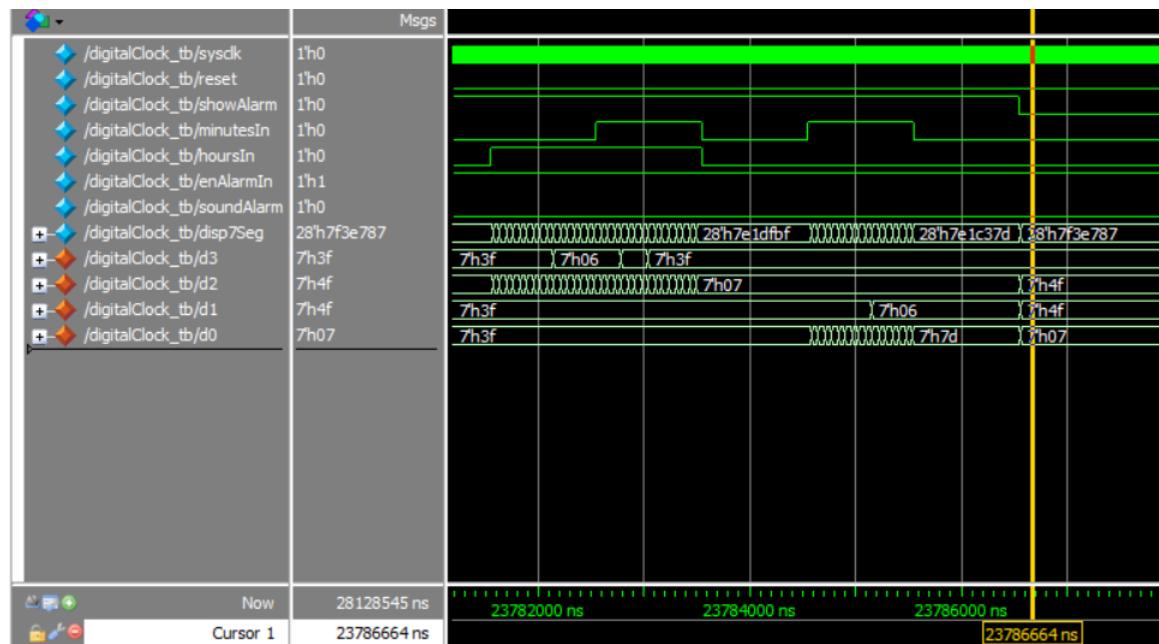


Figure 4.5 Alarm setting test.

Now, see whether it will ring at 07:16 or not when the time comes. As one can see from Figure 4.6, it rings successfully, soundAlarm signal is high for a minute at 07:16.

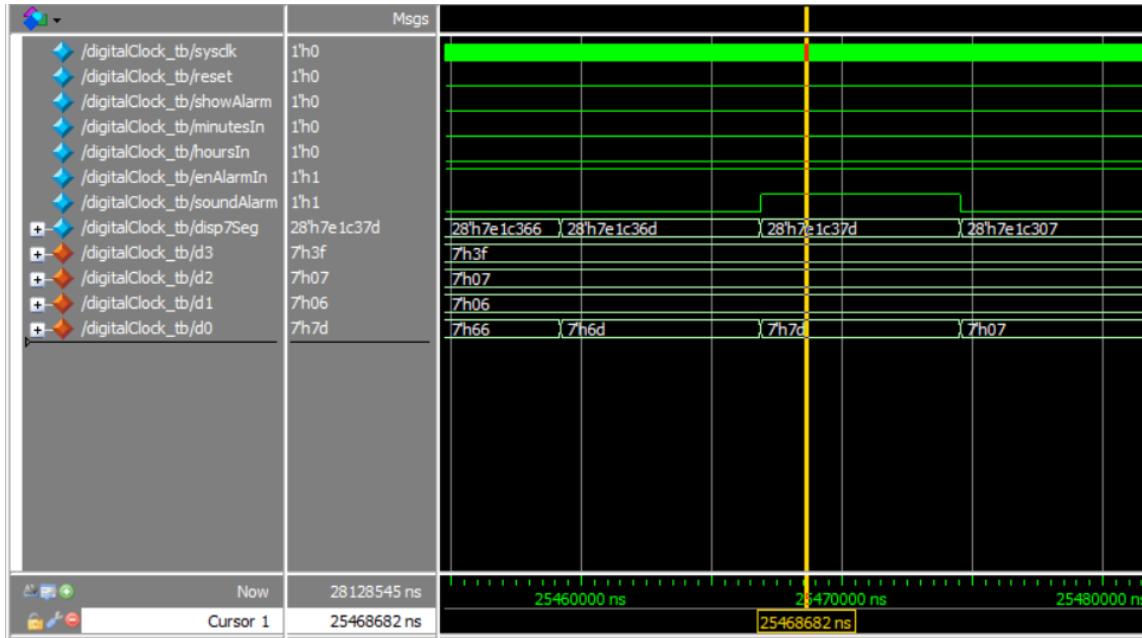


Figure 4.6 Alarm test.

4.4 Possible Improvements

In real-world, input signals cannot be used directly since mechanical switches are not ideal. A module to debounce input signals can be implemented. For a working FPGA project, a screen driver to drive the outputs to the physical output devices is also required. And the testbenches of the modules are not very functional. They only try some input configurations, not all of them. But an appropriate test must cover all the possible inputs.

5 PROJECT 2 – VERIFICATION

In this project, a library written by Digital Electronics Design Engineer Adem Günesen is verified using SystemVerilog (SV). The library has 24 modules which are the replicas of some blocks in Xilinx's SystemGenerator.

5.1 Theory

First of all, as I experienced in Project 1, Verilog is inadequate for testing purposes since it is designed for hardware description and does not have features like randomization, assertions, etc. But SV has many good features for testing.

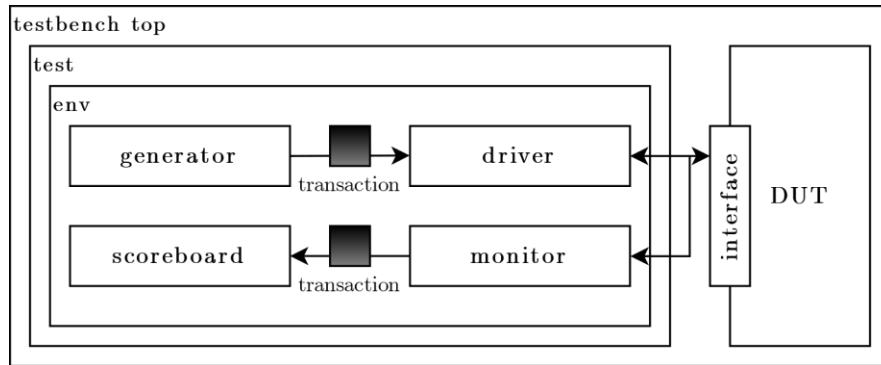


Figure 5.1 Testbench architecture.

A testbench consists of different modules with different functionalities as in Figure 5.1. Modular and object-oriented structure of the testbench increases its reusability and makes the coding process easier. All modules or classes have different functions.

Transaction

This class includes the required fields to be used in the generation of the stimulus. It is a template for a stimulus.

Generator

First, it properly randomizes the transaction class (here, properly means the randomization process is not totally free, rather restricted since randomly generated transactions should obey the real-world scenarios). Then, it sends the generated random transaction to the driver.

Driver

It gets the stimulus generated by the generator and drive it to DUT by converting the high-level transaction class fields to low-level interface signals.

Interface

It is a group of signals used by the DUT. It may specify the direction of the ports. It connects the DUT and testbench. Using interfaces provide us a good way to deal with port signals, without using an interface one must change all the code when one port is changed.

Monitor

After sampling the interface signals, it converts the signal level activity to the transaction level and sends the sampled transaction to Scoreboard via a Mailbox.

Scoreboard

It gets the sampled packet from the monitor and compares this with the expected result, it creates an error (in the console) if there is a mismatch between them. Also, in my testbenches, it outputs the total number of errors / the total number of tests.

Environment

Environment is container class contains Generator, Driver, Monitor, Scoreboard, and Mailboxes for the communication between them.

Test

It creates the environment, makes the necessary configurations for the testbench and starts the stimulus driving.

Testbench Top

This topmost module connects the DUT to the Testbench. So, it includes DUT, Test and Interface instances.

5.2 An Example Testbench

Since there are many testbenches, giving all the implication details in this report would not be appropriate. Therefore, only the implication of a testbench, namely, the testbench for *cmultrom* module will be given. This block implements a gain operator, output equal to the product of its input by a constant. For the full Xilinx's documentation of the block, one can view the documentation "CMult.pdf" which is under "verification/Onurcana/done". And the Adem Günesen's implementation of this block by ROMs is in "verification/Onurcana/GEBZE/G_CMULTROM.vhd".

Since this block requires additional ROMs which are multiplication tables, I created them with a simple module called *g_cmultrom_file_generate*. The created files will be under folder *cmultromtables*.

```
`include "p_memfuncs.sv"

module g_cmultrom_file_generate;
    function void printData;
        int f;
        string number;
        logic
```

```

signed[(g_cmultrom_tb_parampkg::Width+clogb2(g_cmultrom_tb_parampkg::Constant_Value)+1):0] res;

    for(int signed num = -255; num < 256; num++) begin
        number.itoa(num);
        f =
$fopen({"cmultromtables/g_cmultrom_data_",number,".txt"}, "w");
        for(int i = 0; i < 256; i++) begin
            res = i*num; $fdisplay(f, "%b", res);
        end
    end
endfunction

initial begin
    printData();
end
endmodule

```

Then, since the VHDL implementation of the block uses parameters and parameters cannot be changed in run-time, for randomized parameters of DUT, a module called *g_cmultrom_param_generate* is implemented. This module will output the randomized parameters to an external file and these parameters will be used in the testbench. DUT's parameters should be constants, so they cannot be randomized in testbench run-time.

```

`include "p_memfuncs.sv"

module g_cmultrom_param_generate;
    class A;
        rand int unsigned Width, Latency;
        rand int signed Constant_Value;
        rand bit Enable_Port, Signed_Unsigned;

        constraint C { Latency <= 256; Width inside {[1:256]};
Constant_Value inside {[0:255]}; }

        function void printPackage;
            int f = $fopen("g_cmultrom_tb_parampkg.sv");
            string number;
            number.itoa(Constant_Value);
            $fdisplay(f, "package g_cmultrom_tb_parampkg;");
            $fdisplay(f, " parameter Latency = %0d;", Latency);
            $fdisplay(f, " parameter Width = %0d;", Width);
            $fdisplay(f, " parameter Enable_Port = %0b;", Enable_Port);
            $fdisplay(f, " parameter Signed_Unsigned =
%0b;", Signed_Unsigned);
            $fdisplay(f, " parameter Constant_Value =
%0d;", Constant_Value);
    endfunction
endclass
endmodule

```

```

        if(Constant_Value!=0) $fdisplay(f, " parameter DATA_FILE =
\"C:\\\\\\\\questasim64_10.4c\\\\\\\\examples\\\\\\\\cmultromtables\\\\\\\\g_cmultrom_data_\",nu
mber,.txt\\;");
        else $fdisplay(f, " parameter DATA_FILE = \"All zeros\\;");
        $fdisplay(f, "endpackage");
    endfunction
endclass

A a;
initial begin
    a = new();
    a.randomize();
    a.printPackage();
end
endmodule

```

This module creates an example output package like this:

```

package g_cmultrom_tb_parampkg;
parameter Latency = 154;
parameter Width = 29;
parameter Enable_Port = 1;
parameter Signed_Unsigned = 0;
parameter Constant_Value = 250;
parameter DATA_FILE =
"C:\\questasim64_10.4c\\examples\\cmultromtables\\g_cmultrom_data_250.txt";
endpackage

```

Now, testbench can be implemented using the ROMs and parameters.

```

`include "p_memfuncs.sv"

interface intf(input logic clk, reset);
    logic en;
    logic [(g_cmultrom_tb_parampkg::Width-1):0] address;

    logic
[(g_cmultrom_tb_parampkg::Width+clogb2(g_cmultrom_tb_parampkg::Constant_Value)+1):0] DOUT;
endinterface

class transaction;
    rand bit [(g_cmultrom_tb_parampkg::Width-1):0] address;
    constraint C {address < 257; }

    bit en;

    bit
[(g_cmultrom_tb_parampkg::Width+clogb2(g_cmultrom_tb_parampkg::Constant_Value)+1):0] DOUT;

    function void display(string name);

```

```

        $display("-----");
        $display("- %s ",name);
        $display("-----");
        $display("- addr=%0d", address);
        $display("- DOUT = %0d",DOUT);
        $display("-----");
    endfunction
endclass

class generator;
    rand transaction trans;
    mailbox gen2driv;

    function new(mailbox gen2driv);
        this.gen2driv = gen2driv;
    endfunction

    task main();
        forever begin
            trans = new();
            if(!trans.randomize()) $fatal("Gen::: trans randomization
failed");
            gen2driv.put(trans); #5;
        end
    endtask
endclass

class driver;
    int no_transactions;
    virtual intf vif;
    mailbox gen2driv;

    covergroup cg;
        // read all the addresses
        coverpoint vif.address {
            bins range[257] = {[0:256]};
        }
    endgroup

    real coverage;
    event covered;

    function new(virtual intf vif,mailbox gen2driv);
        this.vif = vif;
        this.gen2driv = gen2driv;
        this.cg= new();
    endfunction

    function real get_coverage;
        $display("-----");
        $display("- [ Driver ] ");
        $display("-----");
        $display("- coverage = %0f",coverage);
    endfunction

```

```

$display("-----");
return coverage;
endfunction

task reset;
    wait(vif.reset);
    // $display("[ DRIVER ] ----- Reset Started -----");
    vif.address <= 0;
    vif.en <= 0;
    // $display("[ DRIVER ] ----- Reset Ended -----");
endtask

task main;
    forever begin
        transaction trans;
        gen2driv.get(trans);
        repeat(10)
            @(posedge vif.clk);
            vif.en <= 1;
            vif.address <= trans.address;
            @(posedge vif.clk);
            vif.en <= 0;
            //trans.DOUT = vif.DOUT;

            cg.sample();
            coverage = cg.get_inst_coverage();
            if(this.get_coverage()==100) begin -> covered; break; end

            //@(posedge vif.clk);
            //      trans.display("[ Driver ]");
            no_transactions++;
    end
endtask
endclass

class monitor;
    virtual intf vif;

    mailbox mon2scb;

    function new(virtual intf vif,mailbox mon2scb);
        this.vif = vif;
        this.mon2scb = mon2scb;
    endfunction

    task main;
        forever begin
            transaction trans;
            trans = new();
            //repeat(100)
            @(posedge vif.clk);
            trans.address = vif.address;
            trans.en = vif.en;

```

```

        trans.DOUT = vif.DOUT;

        //@(posedge vif.clk);
        mon2scb.put(trans);
        // trans.display("[ Monitor ]");
    end
endtask
endclass

class scoreboard;
    mailbox mon2scb;
    int no_transactions;
    int no_errors=0;
    typedef
bit[(g_cmultrom_tb_parampkg::Width+clogb2(g_cmultrom_tb_parampkg::Constant_Value)+1):0] t_cmultrom[256:0];
    typedef
bit[(g_cmultrom_tb_parampkg::Width+clogb2(g_cmultrom_tb_parampkg::Constant_Value)+1):0] t_pipe[(g_cmultrom_tb_parampkg::Latency-1):0];

    function new(mailbox mon2scb);
        this.mon2scb = mon2scb;
    endfunction

    function t_cmultrom initramfcmultromfile (input string ramfilename,
integer r_width, integer r_depth);
        int ramfile = $fopen(ramfilename, "r");
        t_cmultrom v_ram_name;

        for(int v_i=0; v_i<r_depth; v_i++)
            $fscanf(ramfile, "%b", v_ram_name[v_i]);

        return v_ram_name;
    endfunction

    function t_cmultrom init_fcmultrom_file_or_zeroes(input string ramfile,
integer r_width, integer r_depth);
        t_cmultrom v_ram_name_0;

        if(ramfile != "All zeros")
            v_ram_name_0 = initramfcmultromfile(ramfile, r_width,
r_depth);

        return v_ram_name_0;
    endfunction

    task main;
        transaction trans;

        bit[(g_cmultrom_tb_parampkg::Width-1):0] res, Rom_Data;
        t_pipe pipe_reg;
        bit Enable_Inside;

```

```

    t_cmultrom SP_ROM =
init_fcmultrom_file_or_zeroes(g_cmultrom_tb_parampkg::DATA_FILE,
(g_cmultrom_tb_parampkg::Width+clogb2(g_cmultrom_tb_parampkg::Constant_Value)+2
), 256);

    forever begin
        mon2scb.get(trans);
        trans.display("[ Scoreboard ]");

        Enable_Inside = g_cmultrom_tb_parampkg::Enable_Port ?
trans.en : 1;
        Rom_Data = SP_ROM[trans.address];

        if(g_cmultrom_tb_parampkg::Latency == 0) res = Rom_Data;

        if(g_cmultrom_tb_parampkg::Latency == 1) res = pipe_reg[0];

        if(g_cmultrom_tb_parampkg::Latency > 1 && Enable_Inside)
res = pipe_reg[g_cmultrom_tb_parampkg::Latency-1];

        if(res == trans.DOUT) $display("Result is as Expected
(%0d:%0d)\n", no_transactions-no_errors, no_transactions);
        else $error("WRONG#%0d!\n\tExpected: %0d Actual:
%0d\n",++no_errors, res, trans.DOUT);

        if(g_cmultrom_tb_parampkg::Latency > 0 && Enable_Inside)
pipe_reg[0] = Rom_Data;
        if(g_cmultrom_tb_parampkg::Latency > 1 && Enable_Inside)
        pipe_reg[(g_cmultrom_tb_parampkg::Latency-1):1] =
pipe_reg[(g_cmultrom_tb_parampkg::Latency-2):0];

        no_transactions++;
    end
endtask
endclass

class environment;
    generator gen;
    driver driv;
    monitor mon;
    scoreboard scb;

    mailbox gen2driv;
    mailbox mon2scb;

    virtual intf vif;

    function new(virtual intf vif);
        this.vif = vif;

        gen2driv = new();
        mon2scb = new();

```

```

        gen = new(gen2driv);
        driv = new(vif,gen2driv);
        mon = new(vif,mon2scb);
        scb = new(mon2scb);
    endfunction
    task pre_test();
        driv.reset();
    endtask
    task test();
        fork
            gen.main();
            driv.main();
            mon.main();
            scb.main();
        join_any
    endtask
    task post_test();
        wait(driv.covered.triggered); // coverage driven
    endtask
    task run;
        pre_test();
        test();
        post_test();
        $stop;
    endtask
endclass

program test(intf intf);
    environment env;

    initial begin
        env = new(intf);
        env.run();
    end
endprogram

module g_cmultrom_tbench_top;
    bit clk;
    bit reset;

    initial begin
        reset = 1;
        #10 reset = 0;

        #1000 reset = 1;
        #10 reset = 0;
    end

    always #5 clk++;

    intf i_intf(clk,reset);
    test t1(i_intf);

```

```

g_cmultrom #((
    .Latency(g_cmultrom_tb_parampkg::Latency),
    .AWidth(g_cmultrom_tb_parampkg::Width),
    .Signed_Unsigned(g_cmultrom_tb_parampkg::Signed_Unsigned),
    .DATA_FILE(g_cmultrom_tb_parampkg::DATA_FILE),
    .Enable_Port(g_cmultrom_tb_parampkg::Enable_Port),
    .Constant_Value(g_cmultrom_tb_parampkg::Constant_Value)
) DUT (
    .CLK(i_intf.clk),
    .EN(i_intf.en),
    .A_SIG(i_intf.address),
    .P_SIG(i_intf.DOUT)
);
endmodule

```

As one can see from the code, our testbench is coverage-driven which means the test runs until full coverage rating. This guarantees that all the possible scenarios are tested.

Compiling and simulating all these files is a quite challenging job but can be automated using a TCL script.

```

vlog -work work -vopt -sv -stats=none
C:/questasim64_10.4c/examples/g_cmultrom_param_generate.sv
vsim -voptargs+=acc -sv_seed random work.g_cmultrom_param_generate
run -all
vlog -work work -vopt -sv -stats=none
C:/questasim64_10.4c/examples/g_cmultrom_tb_parampkg.sv
vlog -work work -vopt -sv -stats=none
C:/questasim64_10.4c/examples/g_cmultrom_files_generate.sv
vsim -voptargs+=acc work.g_cmultrom_file_generate
run -all
vlog -work work -vopt -sv -cover sbcet3 -stats=none
C:/questasim64_10.4c/examples/g_cmultrom_tb.sv
vsim -voptargs+=acc -sv_seed random -coverage work.g_cmultrom_tb_top
add wave -position insertpoint sim:/g_cmultrom_tb_top/i_intf/*
run -all
wave zoom full

sim:/g_cmultrom_tb_top/i_intf/*
run -all
wave zoom full

```

After the execution of this single script, everything runs in order automatically and the screenshot of this testbench can be seen in Figure 5.2.

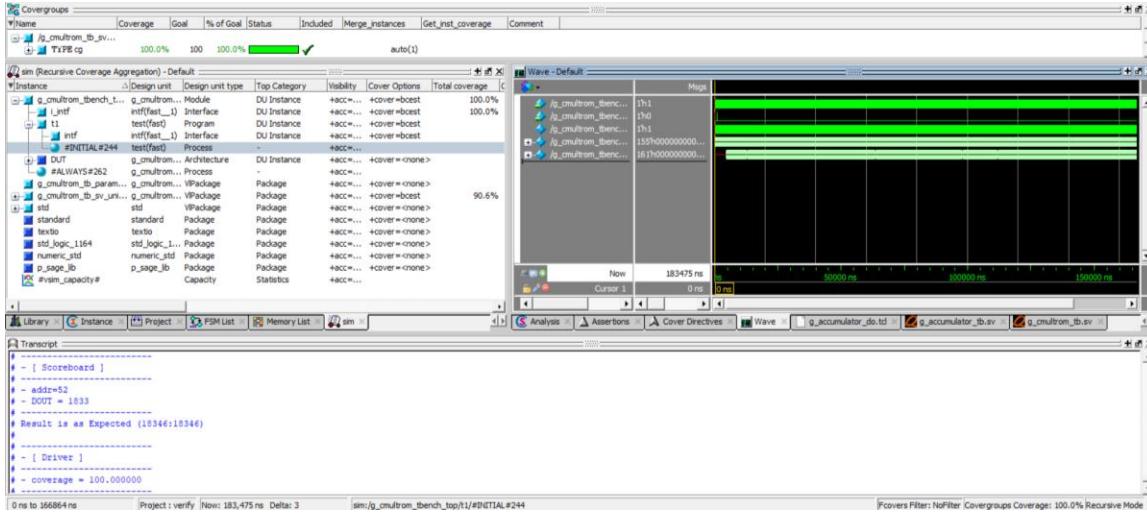


Figure 5.2 The output of the testbench.

5.3 Source Files

This report will not cover all the actual implementation of this project since they are very long and there are too many modules but all the source files of this project are under “verification” folder in [Appendix I](#). In this folder, execution of the files with `.tcl` extension starts the testing process for that module. By using TCL code, the whole test process is automated, it simulates required modules automatically. Since the modules in the library are parameterized, for a randomized functional test, the module parameters must also be parametrized. The files with name “`modulename_param_generate.sv`” randomly create module parameters for DUT and write these parameters to the files with name “`modulename_tb_parampkg.sv`”. And “`modulename_tb.sv`” files are the main testbench blocks. Adem Günesen’s library codes are under “`verification/Onurcana/GEBZE`” folder and Xilinx’s documentations for these blocks are under “`verification/Onurcana/done`”.

6 PROJECT 3 – FFT IP

6.1 Introduction

In this project, a replica of the Xilinx's FFT v9.0 IP is implemented using SystemVerilog (SV) and it is tested using MATLAB by comparing the results comes from the simulation with the *fft* function of MATLAB. This replica is required since in SAGE, Xilinx's FFT block is used in projects which are required to process GNSS data or image data comes from cameras. The SAGE engineers use Xilinx's System Generator building blocks for these DSP applications without actually implementing these building blocks in HDL level. But to modify these blocks and to reduce external dependencies, it is required to implement them. My FFT IP is not fully functional as Xilinx's but rather I was told to implement only the configurations which are used in the DSP projects. So, radix-4 burst I/O, radix-2 burst I/O architectures with scaled/unscaled and fract8/float32 options were implemented. And they also support different transform lengths.

6.2 Fourier Series

According to *Fourier convergence theorem*, if $f(x)$ is a piecewise continuous periodic function of period $T=2L$, then there exist unique sets of constants a_0, a_1, \dots and b_0, b_1, \dots such that

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos \frac{n\pi x}{L} + b_n \sin \frac{n\pi x}{L} \right) \quad (4.1)$$

for every value of x for which f is continuous. At the discontinuous points of the function, the right-hand side of the equation converges to mean of the right and left limits of the function.

The right-hand side of the equation is called *Fourier series of $f(x)$* and the coefficients a_0, a_1, \dots and b_0, b_1, \dots can be computed with the following formulas:

$$a_m = \frac{1}{L} \int_{-L}^L f(x) \cos \frac{m\pi x}{L} dx \quad (4.2)$$

$$b_m = \frac{1}{L} \int_{-L}^L f(x) \sin \frac{m\pi x}{L} dx \quad (4.3)$$

The proof of these formulas will not be given in this report, but it is simply based on the orthogonality relationships between sin and cos functions with different frequencies.

Equivalently, using complex numbers, $f(x)$ can be also represented by the summation $f(x) = \sum_{n=-\infty}^{+\infty} c_n e^{j\frac{n\pi x}{L}}$ where $c_n = \frac{1}{T} \int_{-L}^L f(x) e^{-j\frac{n\pi x}{L}} dx$.

6.3 Fourier Transform

As mentioned, *Fourier series* are only for periodic functions. Using Fourier series, one can pass time-domain to the frequency domain. It is also possible to extend this idea to aperiodic functions using *Fourier transform* which transforms a function of time, $x(t)$, to a function of frequency, $X(\omega)$. It can be derived from Fourier series formulas if $T = \frac{1}{f} = \frac{2\pi}{\omega_0} = 2L \rightarrow \infty$.

$$X(\omega) = \int_{-\infty}^{+\infty} x(t) e^{-j\omega t} dt \quad (4.4)$$

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(\omega) e^{j\omega t} d\omega \quad (4.5)$$

Equation (4.4) is called *Forward Fourier Transform* since it converts time-domain to frequency-domain and (4.5) is called *Inverse Fourier Transform* since it converts frequency-domain to time-domain.

Discrete Fourier Transform

Note that Fourier transform is obtained from Fourier series by transforming discrete coefficients to continuous as the length $L \rightarrow \infty$. But this means infinite number of coefficients and these techniques are applicable to piecewise continuous functions which have a value at every number in its domain. But in the real world, it is impossible to store infinite number of things and to sample a signal at every point in time. Rather, one can sample the input signal finite number of times and can make finite number of calculations. At this point, discrete version of the Fourier transform can be derived as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi nk/N}, \quad 0 \leq k \leq N-1 \quad (4.6)$$

$$x(k) = \sum_{n=0}^{N-1} X(k) e^{j2\pi nk/N}, \quad 0 \leq k \leq N-1 \quad (4.7)$$

where $X(k)$ and $x(n)$ are periodic sequences with period N , $X(k)$ is the frequency-domain representation and $x(n)$ is the time-domain representation of periodic finite duration sequence.

6.4 Fast Fourier Transform

However, directly implementing Discrete Fourier Transform (DFT) is very inefficient in terms of algorithmic complexity. N -point direct DFT requires $4N^2$ real multiplications and $N(4N - 2)$ real additions (since for each coefficient, $4N$

real multiplications, and $2N + 2(N - 1) = 4N - 2$ real additions are required). So, its computational complexity is $O(N^2)$. It is possible to compute DFT or its inverse efficiently with complexity $O(N \log N)$ using *Fast Fourier Transform (FFT)* since FFT only requires $4 \cdot \frac{N}{2} \cdot \log_2 N = 2N \cdot \log_2 N$ real multiplications and $2 \cdot \frac{N}{2} \cdot \log_2 N = N \cdot \log_2 N$ real additions by utilizing the symmetry of the coefficients. Therefore, FFT is much more efficient than direct computation of DFT in terms of time and hardware resources. There are many different types of FFT/IFFT algorithms and architectures, but in this project radix-2/radix-4 decimation-in-time (DIT) type with burst I/O architecture (in-place computed) is implemented only.

6.5 Theory of Radix-2 DIT

FFT is a divide-and-conquer algorithm, that is, the very fundamental idea behind this algorithm is to divide an N -length input sequence into smaller sequences and compute them separately. This division can be done in time-domain or in the frequency-domain. In this project, since DIT type is used, this division is done in time-domain by separating the even and odd samples of $x(n)$ which is an N -length input sequence. Then, these $N/2$ -length sequences are also separated into two $N/4$ length sequences and so on... This division is done until 2-point sequences and as final steps, DFT is applied on these 2-point sequences. Note that, here N must be an integer power of 2. Also, coefficients are required to compute the DFTs.

This division can be done mathematically as follows:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x_1(n) e^{-j2\pi(2n)k/N} + \sum_{n=0}^{\frac{N}{2}-1} x_2(n) e^{-j2\pi(2n+1)k/N}, \quad 0 \leq k \leq N-1 \quad (4.8)$$

where $x_1(n) = x(2n)$ is the even sequence and $x_2(n) = x(2n+1)$ is the odd sequence (for, $0 \leq n \leq \frac{N}{2} - 1$).

Let “coefficient/twiddle/phase factor” be $W_N^n = e^{-j2\pi n/N}$, then $X(k)$ can be written as follows:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x_1(n) W_N^{(2n)k} + \sum_{n=0}^{\frac{N}{2}-1} x_2(n) W_N^{(2n+1)k}, 0 \leq k \leq N-1 \quad (4.9)$$

In this equation, $W_N^{(2n+1)k} = e^{-j2\pi((2n+1)k)/N} = e^{-j2\pi(2nk+k)/N} = e^{-j2\pi(2nk)/N} e^{-j2\pi(k)/N}$. $e^{-j2\pi(k)/N} = W_N^{2nk} W_N^k$ and $W_N^{2nk} = e^{-j2\pi(2nk)/N} = e^{-j2\pi(nk)/\frac{N}{2}} = W_N^{nk}$.

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x_1(n) W_N^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_2(n) W_N^{nk}, 0 \leq k \leq N-1 \quad (4.10)$$

$$X(k) = X_1(k) + W_N^k X_2(k) \quad (4.11)$$

(4.11) is the recursion formula for FFT algorithm but the base case (2-point DFT) is also required:

$$X(k) = x(0) + W_2^k x(1), 0 \leq k \leq 1 \quad (4.12)$$

$$X(0) = x(0) + W_2^0 x(1) \quad (4.13)$$

$$X(1) = x(0) + W_2^1 x(1) \quad (4.14)$$

By substituting $W_2^0 = 1$ and $W_2^1 = -1$ in equation (4.13) and (4.14):

$$X(0) = x(0) + x(1) \quad (4.15)$$

$$X(1) = x(0) - x(1) \quad (4.16)$$

Equations (4.15) and (4.16) can be represented by a signal-flow graph as follows:

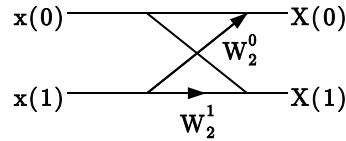


Figure 6.1 Data-flow graph of 2-point DIT FFT.

There are many different conventions for the representation of signal-flow graphs but in this report, this convention will be used. Arrow represents complex multiplication. If there is no arrow, then there is no multiplication in that path. The upper-right and bottom-right edge of “X” shape always correspond to complex addition. This architecture can be optimized as in Figure 4.2.

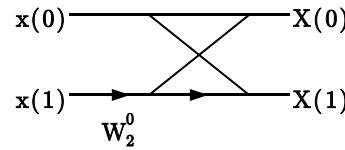


Figure 6.2 Data-flow graph of optimized 2-point DIT FFT.

In the optimized form, the multiplication coefficient at the bottom of the “X” shape is always -1, so there is no need to write it every time. Now, we need only 1 complex multiplier rather than 2 as in the non-optimized case. Multiplying with “-1” can be done without using complex multiplier block. This convention may not be good for complex signal-flow graphs but enough for our purposes.

The mathematics of the radix-4 case is very similar and will not be given in this report. x in “radix- x ” represents the number of samples in each group made at the first stage. So, the only difference is the length of the input sequence. Fundamental block for the radix-4 case takes 4 inputs rather than 2.

6.6 Hardware Implementation

To implement the FFT algorithm in SystemVerilog, I first build some fundamental blocks like multiplier, adder, butterfly, dragonfly... Then by combining these basic blocks, very complex FFT IP is created.

Basic Arithmetic Blocks

Since the IP uses two different input data types, namely fract8 and float32, adder and multiplier for them should be implemented separately.

Fract8 datatype can store fractions between -1 and 1. It has 8-bit, the first bit is the sign bit and the following bits are multiplied by $2^{-1}, 2^{-2}, \dots, 2^{-7}$. The negative value of a number is the two's complement of it. For example, “0 010 0000” corresponds to 0.25 in decimal and “1 110 0000” corresponds to -0.25 in decimal.

Float32 datatype can store very large numbers. In the IEEE 754 standard, it consists of 32-bit. The first bit is the sign bit, following 8 bits determine the exponent and last 23 bits determine the fraction. So, the real value associated with 32-bit sequence $b_{31}b_{30}b_{29}\dots b_1b_0$ is $(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \times (1.b_{22}b_{21}\dots b_0)_2$. Note that any number can be written as 1.x and this saves one bit for free. For example, “0 1000 0100 101 110 0011 0011 0011 0011” corresponds to $(-1)^0 \times 2^{(1000\ 0100)_2 - 127} \times (1.1011110001100110011)_2 = 1 \times 2^{132 - 127} \times 1.73593747615814208984 = 55.55$.

The float32 adder/multiplier and fract8 adder/multiplier implementations in SV are in k_floatsum.sv/k_floatmult.sv and k_fixedsum.sv/k_fixedmult.sv under the “fft/hdl” folder in [Appendix I](#). Note that, float32 adder and multiplier use system functions such as *bitstoshortreal* and *shortrealtobits* which are not synthesizable but are only for testing. But my IP should be synthesized in FPGAs. For this purpose, I also implement the synthesizable versions of these blocks which are k_floatsum_syn.sv/k_floatmult_syn.sv. However, since the silicon test is not done for these blocks, it cannot be safely said that they are synthesizable for the FPGA which will be used.

Since all the arithmetic operations in this IP deal with complex numbers rather than real numbers, complex adder and multiplier have been also implemented. A complex number is represented by a single sequence. The first part of it represents the imaginary part and the last part of it represents the real part of that number. For example, using float32 type a complex number is stored by 64-bit whose first 32-bit represents the imaginary part and the last 32-bit represents the real part. So, a complex adder is simply two ordinary adders and complex multiplier consists of 4 ordinary multipliers and 2 ordinary adders.

Butterfly and Dragonfly

The basic butterfly block can be implemented using complex adder (\oplus), complex multiplier (\times) and multiplier by -1 ($\ominus 1$). Note that multiplying by -1 is represented by a block different from ordinary multiplier since this operation can be done by using less source than ordinary multiplier (only reverse the sign bits). And, scaling is done by complex multipliers at the final stage.

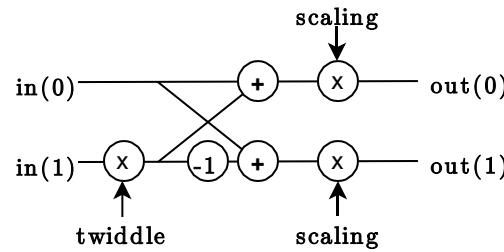


Figure 6.3 Block diagram of the butterfly block.

```

module k_floatbutterfly(
    input logic[63:0] in0, in1, twiddle, logic[1:0] scaling = 0,
    output logic[63:0] out0, out1
);
    wire[63:0] multiplied, out0_, out1_;
    logic[31:0] scaling_factor;

    k_floatcsum sum0(.in0(in0), .in1(multiplied), .out(out0_));
    k_floatcsum sum1(.in0(in0), .in1({!multiplied[63],
multiplied[62:32], !multiplied[31]}, multiplied[30:0])), .out(out1_));
    k_floatcmult mult(.in0(in1), .in1(twiddle), .out(multiplied));

    always@(*) case(scaling)
        2'b00: scaling_factor=32'b00111111000000000000000000000000;
        2'b01: scaling_factor=32'b00111111000000000000000000000000;
        2'b10: scaling_factor=32'b00111110100000000000000000000000;
        2'b11: scaling_factor=32'b00111110000000000000000000000000;
        default: scaling_factor=32'b00111111000000000000000000000000;
    endcase

    k_floatmult scale0(.a(scaling_factor), .b(out0_[31:0]), .out(out0[31:0]));
    k_floatmult
scale1(.a(scaling_factor), .b(out0_[63:32]), .out(out0[63:32]));
    k_floatmult scale2(.a(scaling_factor), .b(out1_[31:0]), .out(out1[31:0]));
    k_floatmult
scale3(.a(scaling_factor), .b(out1_[63:32]), .out(out1[63:32]));

endmodule
  
```

This is the HDL implementation of the float32 butterfly but also, HDL implementation of the fract8 butterfly can be found from “hdl/k_fixedbutterfly.sv”. The butterfly is the fundamental block for radix-2 case but for radix-4 case, the dragonfly is the main block. Before the implementation of the dragonfly, let me first derive the radix-4 case from radix-2 case and show that they are equivalent.

First, consider the optimized 16-point DIT FFT:

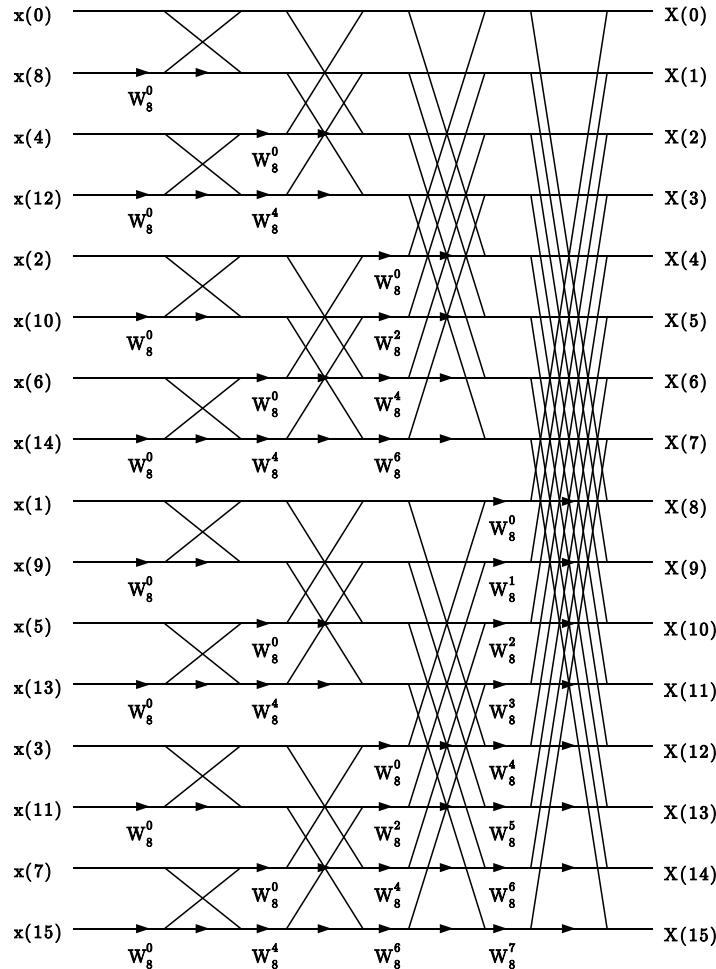


Figure 6.4 Signal flow graph of 16-bit DIT FFT with a bit-reversed input sequence.

This signal-flow graph can be separated into butterfly blocks which have 2 inputs and 2 outputs easily – each “X” shape is a butterfly, but we want to separate it into dragonfly blocks which have 4 inputs and 4 outputs. So, consider the partition:

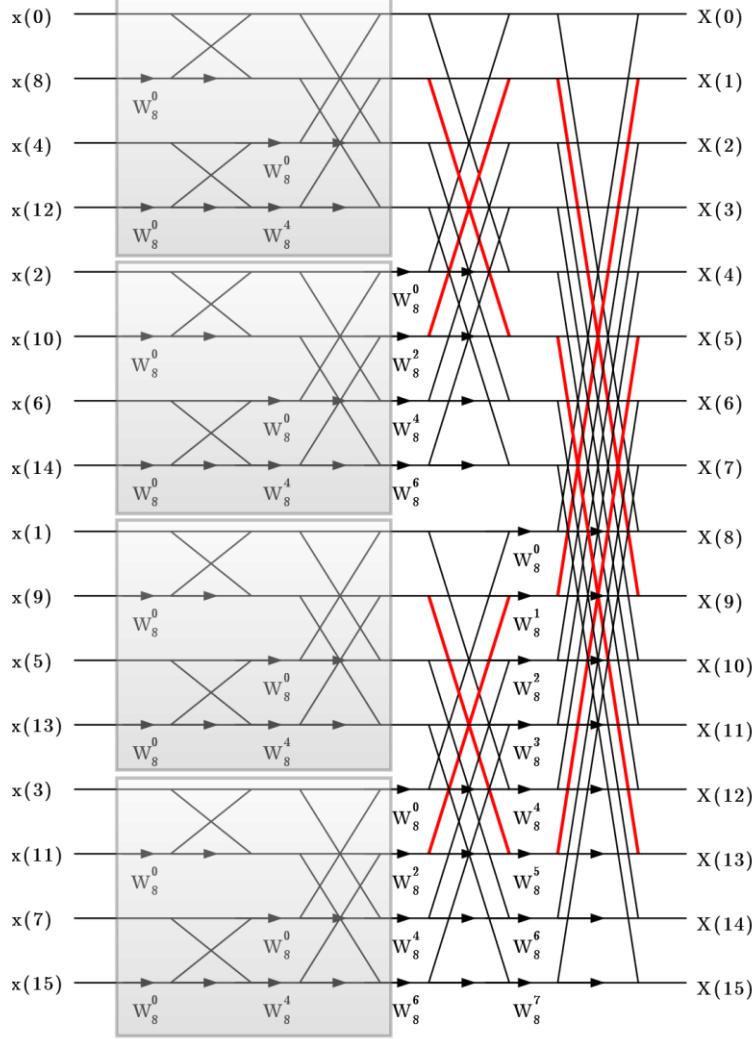


Figure 6.5 Partition of 16-point DIT FFT into dragonfly blocks.

Creation of the first four dragonflies (grey ones) is obvious, they can be formed with adjacent butterflies in the first stage and the subsequent butterflies in the second stage. However, how we can form other dragonflies from butterflies is not that obvious. The dragonfly must not have any dependency other than input signals, so in a dragonfly in-out signals also must only depend on the input signals in some way. A group of paths that has this feature is marked with red in Figure 4.5. Here, output signals $X(1)$, $X(5)$, $X(9)$, $X(13)$ only depend on input signals. So, this group can form a dragonfly block. There are also other groups of signals in the

third and fourth stages like this one, and they can also form dragonfly blocks. So, this 16-point DIT FFT can be formed using 8 dragonfly blocks as in Figure 4.6.

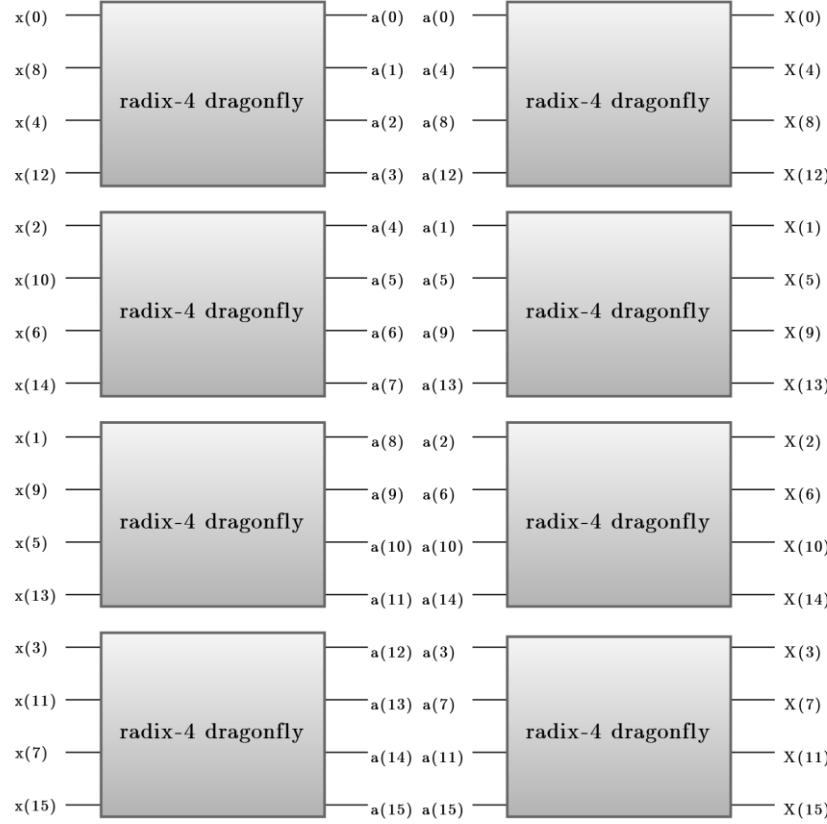


Figure 6.6 Radix-4 partition of 16-point DIT FFT.

Division of the signal-flow graph into radix-4 blocks is theoretically possible since, inside radix-4 blocks, the general pattern for N-point DIT FFT is always in Figure 4.7 and twiddle factors in this pattern can be controlled easily.

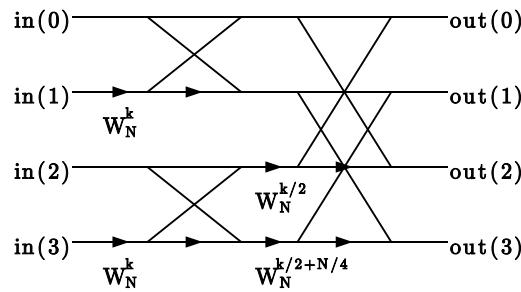


Figure 6.7 General pattern inside radix-4 blocks.

And this pattern can be simplified as in Figure 4.8. For simplification, observe that $W_N^{k/2+N/4} = W_N^{k/2}W_N^{N/4} = W_N^{k/2}e^{-j2\pi(N/4)/N} = W_N^{k/2}e^{-j\pi/2} = W_N^{k/2}(-j)$ and since multiplication with $W_N^{k/2}$ is applied to both outputs, there would be no difference if these multiplications were applied to the inputs. And, multiplying first by W_N^k and then $W_N^{k/2}$ is equivalent to multiplying by $W_N^{3k/2}$ directly.

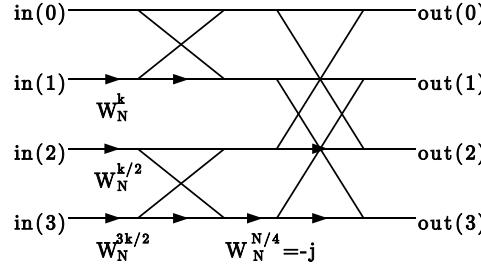


Figure 6.8 Simplified general pattern inside radix-4 blocks.

Now, the hardware design of the dragonfly block can be done as in Figure 4.9 and can be implemented.

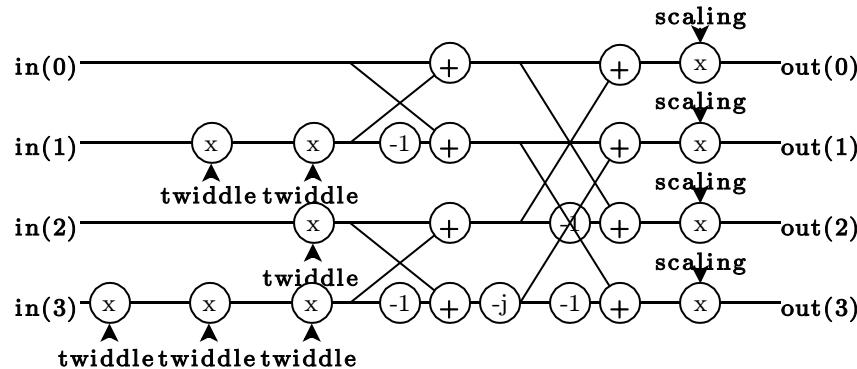


Figure 6.9 Block diagram of the dragonfly block.

Note that since ROM has only one output, in the block diagram of the dragonfly block there is only one twiddle factor $-W_N^{k/2}$. And, the HDL implementation of this block for float32 case can be done as follows:

```
module k_floatdragonfly(
    input logic[63:0] in0, in1, in2, in3, twiddle0, logic[1:0] scaling = 0,
    output logic[63:0] out0, out1, out2, out3
);
```

```

        wire[63:0] mult1, mult2, mult3, a0, a1, a2, a3, a3, out0_, out1_, out2_,
out3_, twiddle1_, twiddle2_;
        logic[31:0] scaling_factor;

        k_floatcmult cmult3(.in0(twiddle0), .in1(twiddle0), .out(twiddle1_));
        k_floatcmult cmult4(.in0(twiddle1_), .in1(twiddle0), .out(twiddle2_));

        k_floatcmult cmult0(.in0(in1), .in1(twiddle1_), .out(mult1));
        k_floatcmult cmult1(.in0(in2), .in1(twiddle0), .out(mult2));
        k_floatcmult cmult2(.in0(in3), .in1(twiddle2_), .out(mult3));

        k_floatcsum sum0(.in0(in0), .in1(mult1), .out(a0));
        k_floatcsum sum1(.in0(in0), .in1({!mult1[63], mult1[62:32], !mult1[31],
mult1[30:0]}), .out(a1));

        k_floatcsum sum2(.in0(mult2), .in1(mult3), .out(a2));
        k_floatcsum sum3(.in0(mult2), .in1({!mult3[63], mult3[62:32], !mult3[31],
mult3[30:0]}), .out(_a3));

        assign a3 = {_a3[31], _a3[30:0], _a3[63:32]};

        k_floatcsum sum4(.in0(a0), .in1(a2), .out(out0_));
        k_floatcsum sum5(.in0(a0), .in1({!a2[63], a2[62:32], !a2[31],
a2[30:0]}), .out(out2_));

        k_floatcsum sum6(.in0(a1), .in1(a3), .out(out1_));
        k_floatcsum sum7(.in0(a1), .in1({!a3[63], a3[62:32], !a3[31],
a3[30:0]}), .out(out3_));

        always@(*)
            case(scaling)
                2'b00: scaling_factor=32'b00111111000000000000000000000000;
                2'b01: scaling_factor=32'b00111111000000000000000000000000;
                2'b10: scaling_factor=32'b00111110100000000000000000000000;
                2'b11: scaling_factor=32'b00111110000000000000000000000000;
                default:
scaling_factor=32'b00111111000000000000000000000000;
            endcase

            k_floatmult scale0(.a(scaling_factor), .b(out0_[31:0]), .out(out0[31:0]));
            k_floatmult
scale1(.a(scaling_factor), .b(out0_[63:32]), .out(out0[63:32]));
            k_floatmult scale2(.a(scaling_factor), .b(out1_[31:0]), .out(out1[31:0]));
            k_floatmult
scale3(.a(scaling_factor), .b(out1_[63:32]), .out(out1[63:32]));
            k_floatmult scale4(.a(scaling_factor), .b(out2_[31:0]), .out(out2[31:0]));
            k_floatmult
scale5(.a(scaling_factor), .b(out2_[63:32]), .out(out2[63:32]));
            k_floatmult scale6(.a(scaling_factor), .b(out3_[31:0]), .out(out3[31:0]));
            k_floatmult
scale7(.a(scaling_factor), .b(out3_[63:32]), .out(out3[63:32]));
endmodule

```

6.7 Overall Design

As special fundamental blocks have been told, now overall design can be provided.

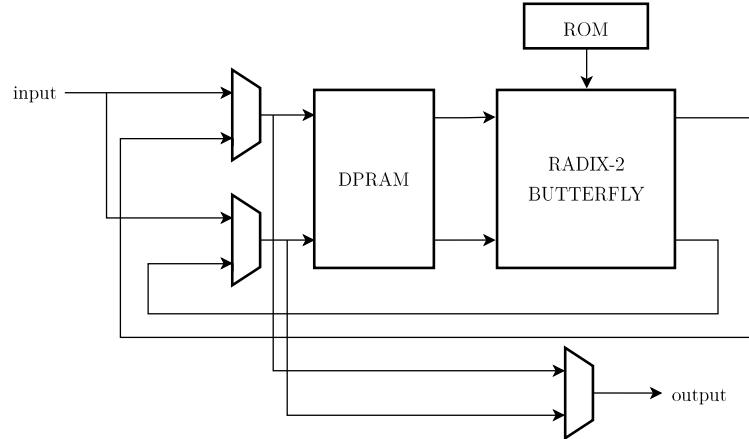


Figure 6.10 Radix-2 FFT with Burst I/O architecture.

In Figure 4.10, DPRAM holds the calculated results and ROM is a look-up table providing twiddle factors to butterfly block. Note that, twiddle factors can be generated using CORDIC algorithm instead of using LUT which generated by “`cpp/fft_twiddlefactors_create.cpp`” and “`cpp/fft4_twiddlefactors_create.cpp`”, but Xilinx’s original design uses LUT. And my design is the same as the Xilinx’s design expect that their design has two switches for butterfly inputs and outputs (Xilinx, 2017, p. 44). For radix-2 case, the design is given in Figure 4.10, and it is very similar to radix-2 case for radix-4 expect that quad-port RAM is used for it instead of dual-port.

6.8 Controller

Controller block controls the selection bits, RAM and ROM addresses.

FFT calculation with one butterfly/dragonfly block and RAM can be done stage by stage. That is, FFT block must first calculate all the outputs of the first stage and write them to RAM, then using this results in RAM, it must calculate all the outputs of the second stage and write them to RAM, and so on.

It has an FSM which has states RST, SAMPLE, READ, WRITE, IDLE, and RESULTS. Since the FFT block takes the input as *AXI4-stream*, when *s_axis_data_tvalid* signal becomes 1, it starts to sample the input data in bit-reversal order by switching to SAMPLE state. When the sampling is finished, it makes *m_axis_data_tlast* signal 1 and switches to READ state. In READ state data is read from RAM for calculation and in WRITE state calculating results are writing to the RAM. After the calculation is finished *m_axis_data_tvalid* becomes 1 to signal calculation process is finished and FSM switches to IDLE state. As the user input *m_axis_data_tready* becomes 1, FSM switches to RESULTS state and FFT starts to supply the results.

Some observations are required to control RAM and ROM addresses. For radix-2 16-point (Figure 4.4) DIT FFT, RAM and ROM addresses are distributed as in Table 4.2 and Table 4.3.

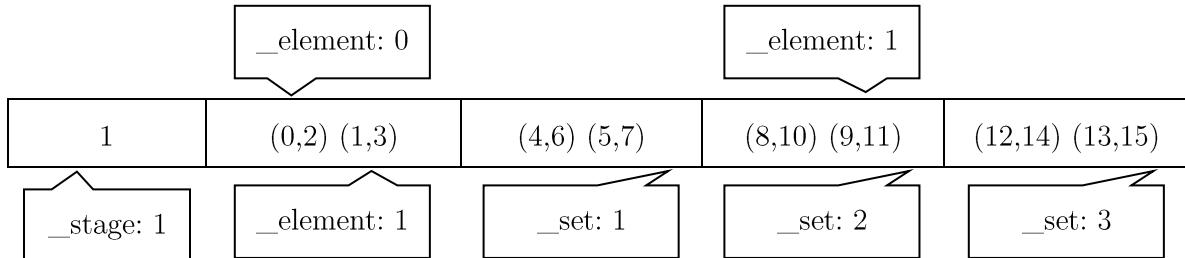
Stage														
0	(0,1)	(2,3)	(4,5)	(6,7)	(8,9)	(10,11)	(12,13)	(14,15)						
1	(0,2) (1,3)		(4,6) (5,7)		(8,10) (9,11)		(12,14) (13,15)							
2	(0,4) (1,5) (2,6) (3,7)				(8,12) (9,13) (10,14) (11,15)									
3	(0,8) (1,9) (2,10) (3,11) (4,12) (5,13) (6,14) (7,15)													

Table 6.1 RAM address distribution for 16-point radix-2 DIT FFT.

Stage														
0	0	0	0	0	0	0	0	0	0					
1	0 4		0 4		0 4		0 4							
2	0 2 4 6				0 2 4 6									
3	0 1 2 3 4 5 6 7													

Table 6.2 ROM address distribution for 16-point radix-2 DIT FFT.

To categorize addresses easily, definition of terms “_stage”, “_set” and “_element” is required. If we consider the first stage in table 4.2, (8,10) is in the first _stage, second _set and it is the first element. So, pair of address which are inputs for DPRAM is called “_element”, a group of “_element’s is called “_set” and the group of “_set’s is called “_stage”. For the first stage in Table 4.2:



Generalize the 16-point case and observe the following facts:

- Total number of _stages = $\log_2 \text{TRANSFORM LENGTH}$
- Total number of _sets in the n^{th} _stage = $2^{\log_2 \text{TRANSFORM LENGTH} - n - 1}$
- Total number of _elements in a _set in the n^{th} _stage = 2^n
- RAM addresses for a^{th} _stage, b^{th} _set and c^{th} _element are given as $2^{2^a} b + c$ and $2^{2^a} b + c + 2^a$
- ROM address for a^{th} _stage, b^{th} _set and c^{th} _element is given as $c 2^{\log_2 \text{TRANSFORM LENGTH} - 1 - a}$

The radix-4 case is very similar to radix-2 case and will not be derived again. Since the controller codes are too long, they will not be provided in this report but can be accessed from “hdl/controller.sv” and “hdl/controller4.sv” files. All the codes of this project are under “fft” folder in Appendix I.

6.9 Test Results and Comments

In the testbench, the outputs of the FFT IP are written to “fft_res.m” file as a MATLAB vector and tests are done by comparing these results with MATLAB’s *fft* function graphically as in Figure 4.11, 4.12, 4.13 and 4.14. And the designed IPs gives the desired results for all configurations. Note that as the number of points increases, precision increases as well and results of FFT core looks more and more

like MATLAB's *fft* function. The radix-4 algorithm calculates faster than radix-2 at the expense of less hardware complexity.

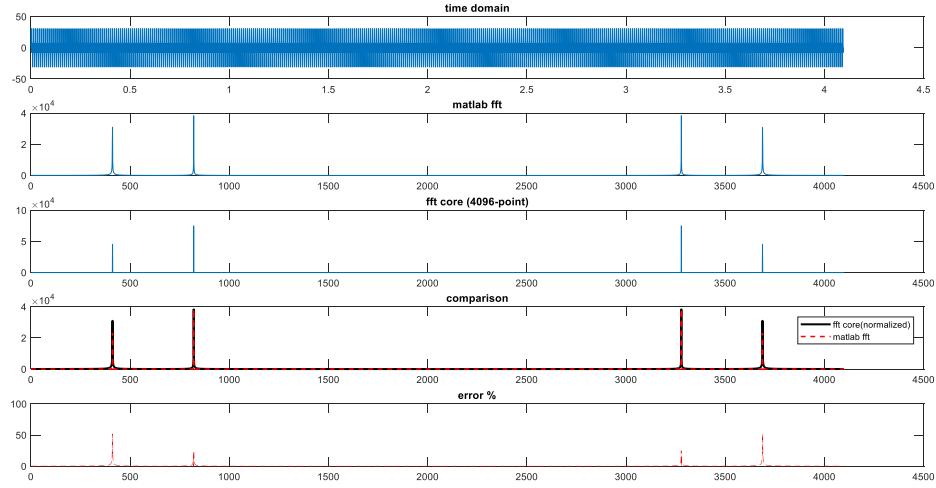


Figure 6.11 Test results for 4096-point radix-2 float32 case.

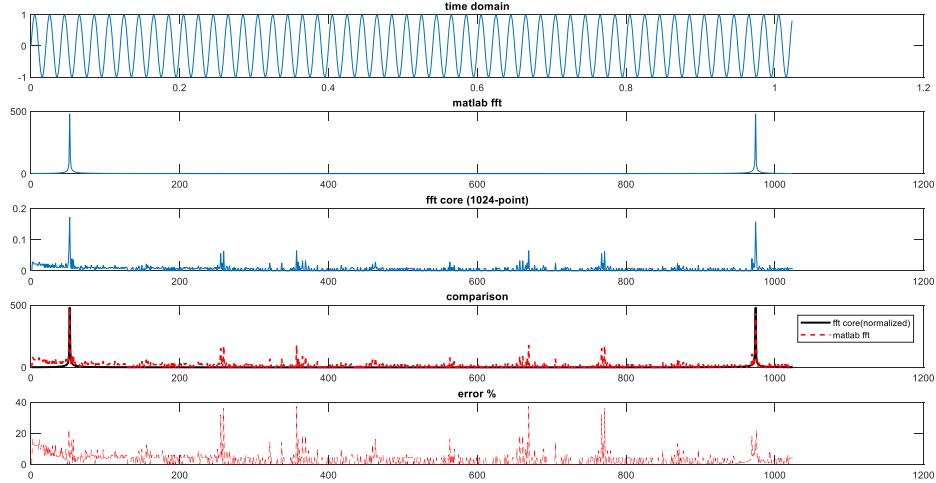


Figure 6.12 Test results for 1024-point radix-2 fract8 case.

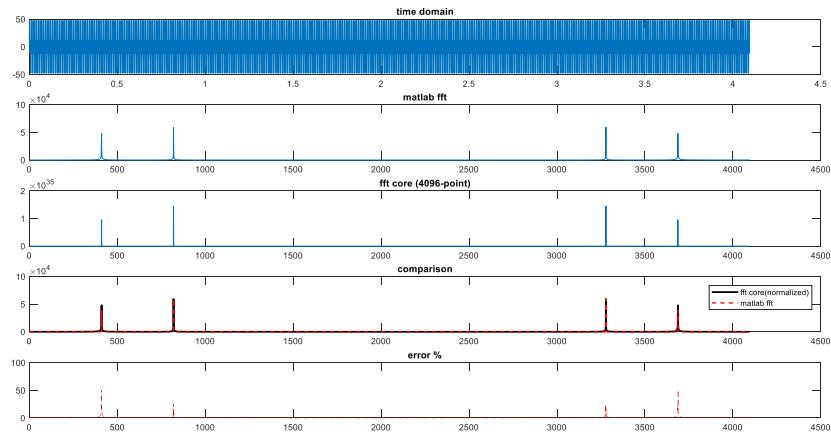


Figure 6.13 Test results for 4096-point radix-4 float32 case.

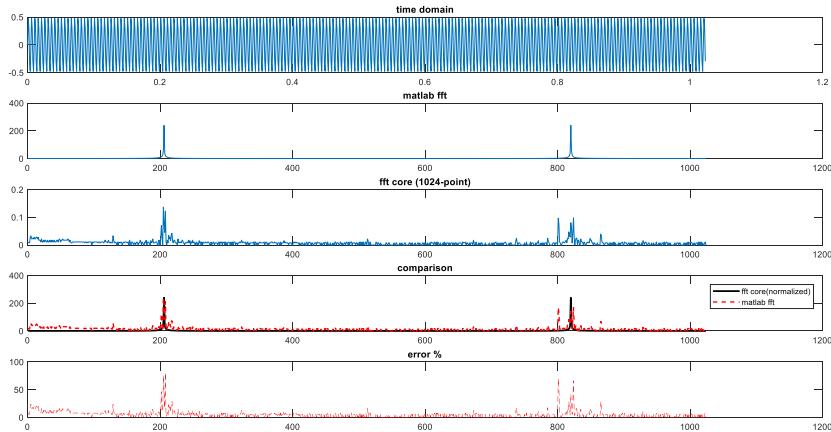


Figure 6.14 Test results for 4096-point radix-4 fract8 case.

Note that, even if my IP works properly, some possible improvements cannot be done because of limited time. For example, in Xilinx's IP, twiddle factors are stored as 24- or 25-bit fixed-point numbers for precision but my IP stores them in float32 or fract8 format. The radix-4 case only supports 4^k -point inputs but could accept 2^k -point calculation if an extra radix-2 stage was added.

7 CONCLUSIONS

I am happy to perform my EE300 summer practice in TÜBİTAK SAGE Digital Electronics Design Department. During the practice, I had the chance of inspecting what is being done in a high-quality R&D environment. All engineers are well informed in their area of expertise and they are very helpful. I did not experience any problems for five weeks since my supervisor and other people in the department were very helpful. I learned about DC-DC converters, digital electronics, HDLs, functional verification, and FFT algorithms. I did not know even the fundamentals of digital electronics, but I learned a lot about it by reading learning materials. I learned Verilog and SystemVerilog, also study VHDL codes and I had chance to use my MATLAB, C++ and OOP knowledge.

I absolutely recommend this location to other students. Because it is one of the important R&D companies in Turkey.

I SOURCE FILES

All source files, block designs, memory initialization files, testbenches, simulation results, and helper programs are available at [GitHub](#).

<https://github.com/monurcan/sageintern>

REFERENCES

- Analog Devices. (2010). *LT3741/LT3741-1 Datasheet*. Retrieved from
<https://www.analog.com/media/en/technical-documentation/datasheets/37411fg.pdf>
- Xilinx. (2017, October 4). *Fast Fourier Transform v9*. Retrieved from
https://www.xilinx.com/support/documentation/ip_documentation/xfft/v9_0/pg109-xfft.pdf