Registration No: 170170725

The University of Sheffield
Electrical and Electronic Engineering

Digital Logic Circuits: Design with Field Programmable Gate Arrays

Neil Powell

15/11/2017

1 Contents

2 Introduction

The use of Field Programmable Gate Arrays (FPGAs) in industry has increased in recent years due to decreases in package cost and improvements in device functionality (low power devices, built-in microprocessors, etc.) [1]. FPGAs provide a set of reconfigurable logic elements that can be programmed via Hardware Description Language (HDL) and synthesis tools [2]; this allows for rapid prototyping and release of custom logic circuits in low or medium volume applications where the cost of developing Application Specific Integrated Circuits (ASICs) is unjustifiable [1] [2]. As FPGAs are reconfigurable, hardware bug fixes and upgrades can be applied to products remotely [1]. FPGAs can also be used to significantly reduce ASIC development costs as designs can be iteratively developed and tested quickly and without the production of many expensive ASIC prototypes.

The aim of the Digital Logic Circuits experiment was to implement and test a 2-bit binary adder on a Xilinx NEXSYS 4 development board using the Xilinx ISE design and synthesis tools. Top Down Design was used to outline the specification of the adder; Bottom Up Implementation was then used to build the adder using low level logic gates (AND, OR, XOR). At each stage simulation tools were used to verify the operation of the logic circuit.

3 Background

The process of designing and implementing logic circuits with FPGAs involves a series of steps to ensure the design is as effective, reusable, reliable and maintainable as possible. **Figure 3.1** shows a diagram of these operations. Top Down Design is used to explicitly specify the overall operation of the circuit before defining the lower level blocks required to implement the design. Bottom up implementation is then used to implement small sections of the design in the micro scale before connecting them in the macro scale. At each stage of implementation, simulation is used to verify the operation of each logic macro. Finally, synthesis tools are used to convert schematics and HDL into file used to configure the FPGA. This section aims to give a brief explanation of each of these processes.

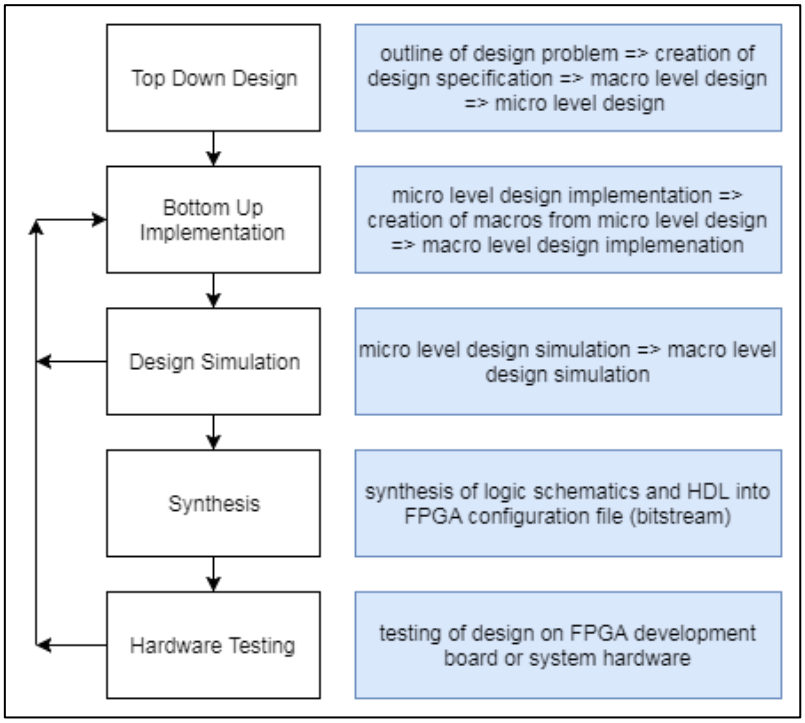

Figure 3.1: Logic Circuit Design Process for FPGAs

3.1 Top Down Design

Top Down Design is the process of methodically breaking a design problem down into its constituent parts, from the macro scale to the micro scale [2]. Designers start by defining the overall operation of the system [3], this involves defining the inputs and outputs as well as the transformation that is performed from input to output. This macro level design is then decomposed into progressively smaller chunks (micro level); this helps partition the design into sections that only perform one task, making implementation easier [2].

The outcome is a hierarchical arrangement that is easy to comprehend as each individual section is very simple [2]. This partitioned design also has increased reusability and maintainability; lower level sections of the design can be used in subsequent projects and bug fixes at the lower level are prevented from affecting higher level sections. A Top Down Design methodology makes the implementation of logic circuits with several million gates possible [2]. **Figure 3.2** shows an example Top Down Design flow.

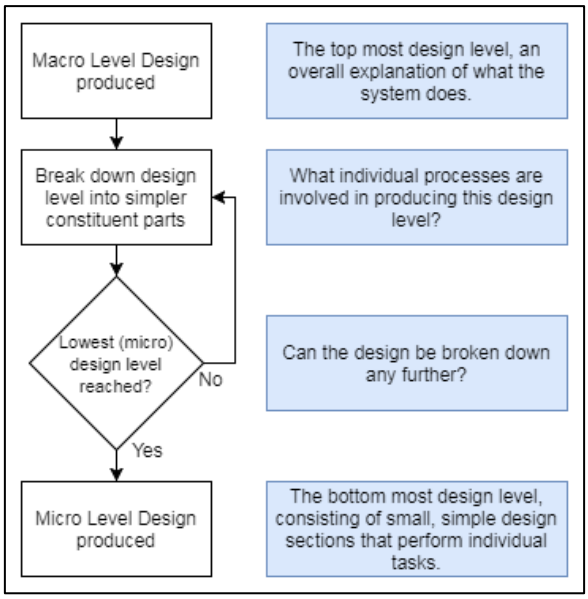

Figure 3.2: Example Top Down Design Flow

3.2 Bottom Up Implementation

Following on from Top Down Design, Bottom Up Implementation can be used to generate the files required to program the FPGA. HDL or schematics are produced for the micro level sections of the design and fully tested using simulations (see section 3.3) [1]. When the operation of these micro level designs has been verified, schematics or HDL files are produced that connected these micro level designs together. Again, simulations are then used to verify the operation of this higher-level section. This process is repeated until the macro design level is implemented.

Bottom Up Implementation produces more efficient and reliable design implementations; each section is inherently simple and thus can be optimised effectively [2]. These simple sections can then be brought together to produce a complex system. Effectively it allows the thought processes of implementation to be broken down so that the engineer can focus on implementation of each section of the design individually. **Figure 3.3** shows an example Bottom Up Implementation flow.

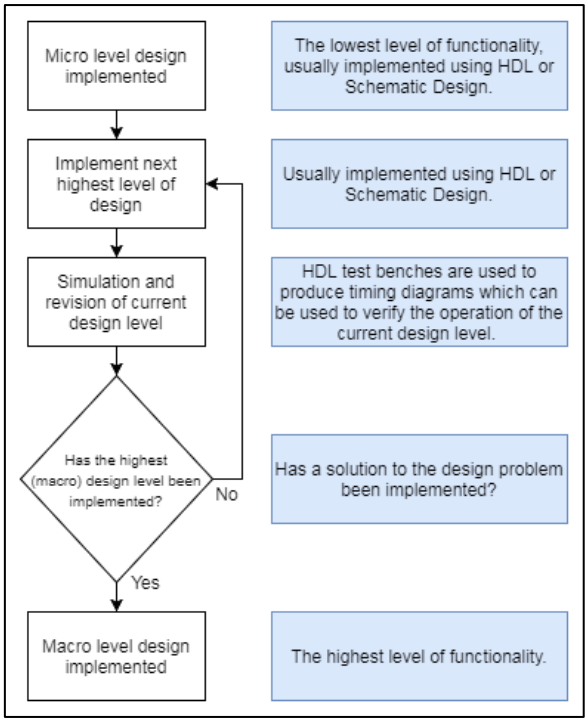

Figure 3.3: Example Bottom Up Implementation Flow

3.3 Simulation

Once implementation of a design level has been completed, simulations are performed to verify that the implemented logic functions correctly. Test benches are generated using HDL, this allows simulated signals to be fed into the implemented logic circuit; the output at each input combination is then graphed on a timing diagram [2]. The timing diagram can then be used to produce a truth table for the logic circuit.

Bottom Up Implementation allows each level to be verified by simulation. This is particularly important when implementing large designs which are difficult to completely verify at the macro level [2]. Lower level sections can be exhaustively tested to give a high level of confidence that the top-level design functions correctly [2].

Simulations also make the process of hardware testing (see section 3.5) much easier. The implemented logic circuit can be virtually tested, removing the effects of issues with the hardware used to operate the FPGA. Thus, when the physical hardware is tested, the operation of the logic circuit inside the FPGA can be removed as a potential source of issues. As simulations can consider hardware delays [2] (setup time, hold time, propagation delay), the output of the logic circuit can be verified to a very high level of confidence.

3.4 Synthesis

Once a design has been fully implemented and verified it can be converted into a set of optimal Boolean equations that fit with the technology inside the FPGA being used [2]. This process is known as design synthesis. The behavioural description outlined in the design HDL or Schematic is converted to a gate-level logic circuit which can be programmed onto the FPGA [2]. During the conversion unnecessary logic is removed by the synthesis tool and the logic layout is optimised to ensure the resultant hardware will meet speed requirements [2].

3.5 Hardware Testing

Once a design has been synthesized the physical FPGA being used in the end system can be programmed with the logic circuit. Hardware testing allows the engineer to verify that the FPGA is operating correctly at the required clock speeds as well as giving information on FPGA power draw and dissipation [2]. Issues with cross talk between output lines from the FPGA can also be found during this stage [2]. Ultimately the Hardware Testing stage ensures that the FPGA and logic design will function correctly when connected to the desired application. Depending upon the application, development boards may be used to complete Hardware Testing on the design, an example FPGA development board is shown in **Figure 3.4**.

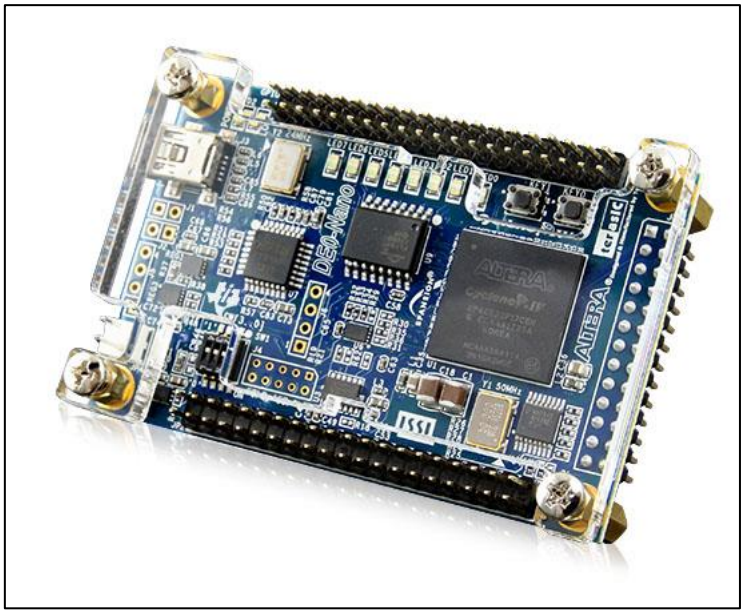

Figure 3.4: Terasic DE0 Nano Altera Cyclone FPGA Development Board

4 Theory

A 2 Bit Binary Adder connected to a 3 Bit Register was implemented to satisfy the requirements of the design problem. Top Down Design was used to split the Adder and Register into their simpler constituent parts. The 2 Bit Binary Adder was composed of two chained full adders. The 3 Bit Register was composed of 3 common clocked D Flip Flops. The following section aims to explain the operation of each design level and justify any design decisions made.

4.1 2 Bit Adder and 3 Bit Register – Macro Design Level

Using schematic capture a 2 Bit Binary Adder was connected to a Synchronous 3 Bit Register. As the 2 Bit Binary adder uses combinational logic the 3 Bit Register was added to store the calculated binary value; allowing the calculated value to be viewed after the input binary values have changed on the output pins of the Register. A 3 Bit Register was used to accommodate all possible output values of the 2 Bit Adder. As the Carry In input of the 2 Bit Adder was unused it was grounded to prevent input floating affecting the Adder output value. **Figure 4.1** shows the Schematic of the 2 Bit Adder and 3 Bit Register combination generated using KiCAD [4]. **Figure 4.2** shows the Truth Table of the 2 Bit Binary Adder and 3 Bit Register combination; this truth table assumes that the register is clocked after the setup, hold and propagation delays of the 2 Bit Adder have elapsed.

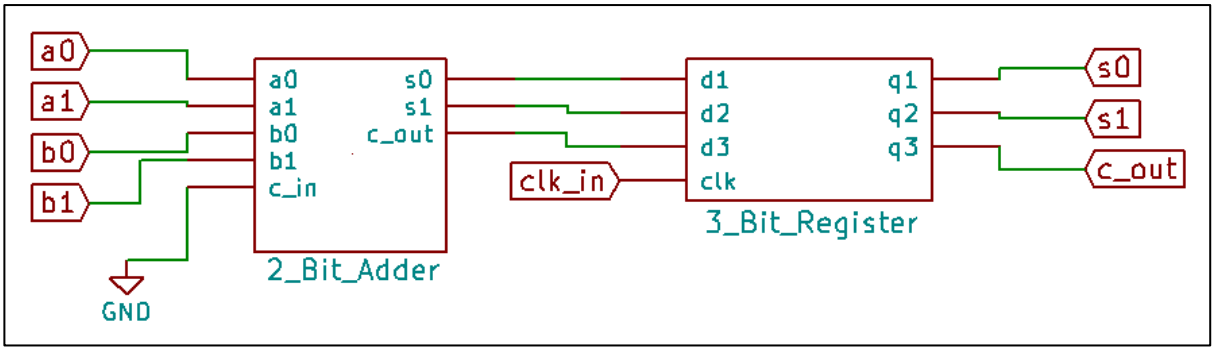

Figure 4.1: 2 Bit Binary Adder and 3 Bit Register Schematic (Generated using KiCAD)

2 Bit Binary Adder and 3 Bit Register Output									
Inputs						Outputs			
a1	a0	Decimal Value	b1	b0	Decimal Value	c_out	s1	s0	Decimal Value
0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	1	1
1	0	2	0	0	0	0	1	0	2
1	1	3	0	0	0	0	1	1	3
0	1	1	0	1	1	0	1	0	2
1	0	2	0	1	1	0	1	1	3
1	1	3	0	1	1	1	0	0	4
1	0	2	1	0	2	1	0	0	4
1	1	3	1	0	2	1	0	1	5
1	1	3	1	1	3	1	1	0	6

Figure 4.2: 2 Bit Binary Adder and 3 Bit Register Truth Table

4.2 2 Bit Adder Macro

The 2 Bit Adder was composed of two full adders in a ripple carry adder configuration. The use of full adders allowed for carry bits to be accounted for. The Carry Out of one full adder was connected to the Carry in of the next full adder. **Figure 4.3** shows the 2 Bit Adder schematic, produced using KiCAD [4].

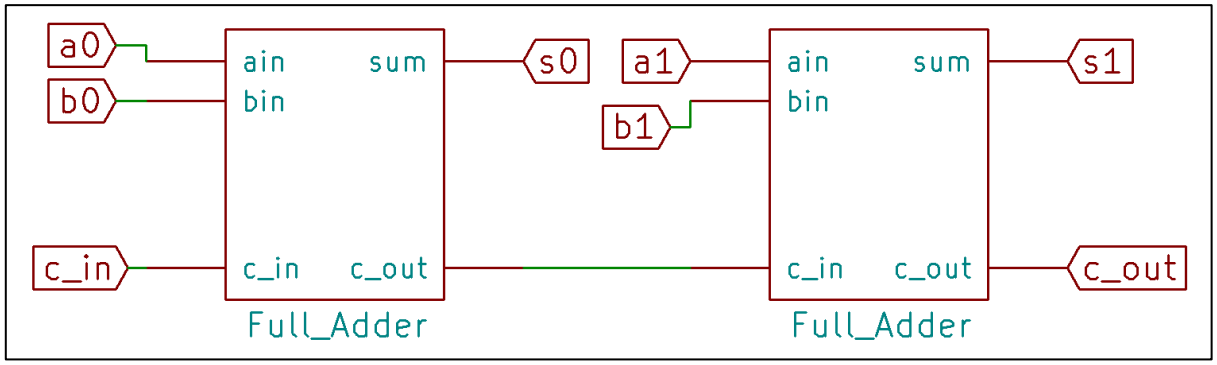

Figure 4.3: Schematic of 2 Bit Binary Adder composed of Full Adders (Generated using KiCAD)

4.3 Full Adder – Micro Level Design

Initially, the full adder was composed of AND, XOR and OR gates. This provided a functional combination logic circuit that could add 3 Bits (ain, bin, c_in) and provide a sum and carry bit as output. As discussed in section 4.2, two of these full adders were chained together to produce the 2 Bit Binary Adder macro. A schematic for this design produced in KiCAD [4] can be seen in **Figure 4.4**. A further design was then implemented solely using Multiplexers. A schematic for this design produced in KiCAD [4] can be seen in **Figure 4.5**. The Truth Table for the Full Adder can be seen in **Figure 4.6**.

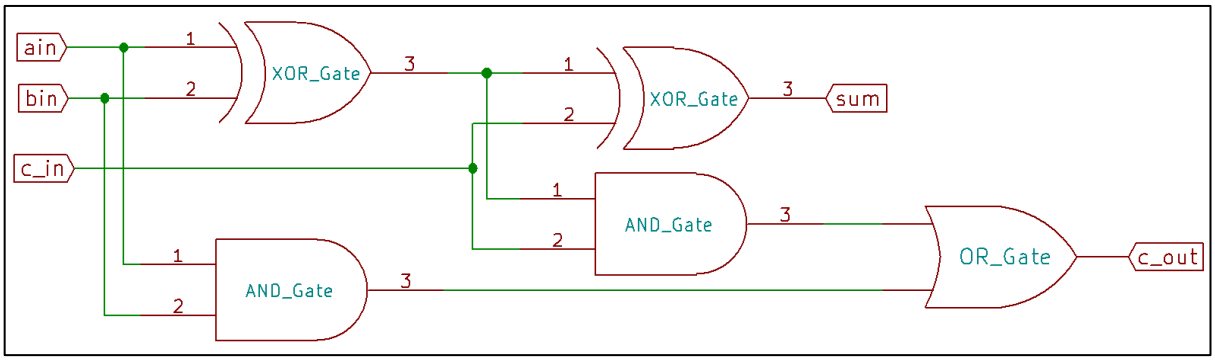

Figure 4.4: Full Adder Schematic (Generated using KiCAD)

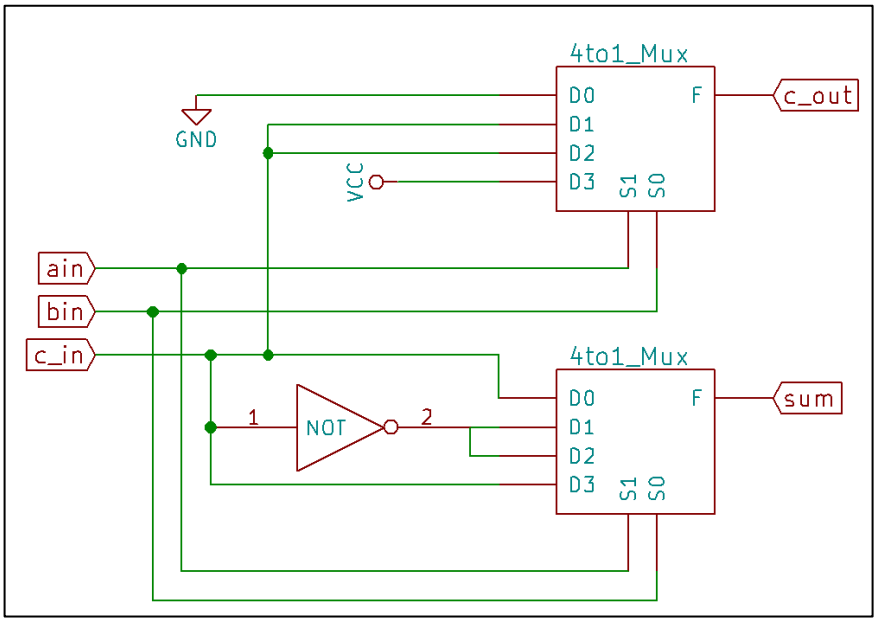

Figure 4.5: Multiplexer Full Adder Schematic (Generated in KiCAD)

Full Adder Truth Table					
Inputs			Outputs		
ain	bin	c_in	sum	c_out	Decimal Value
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	0	1	2
1	0	0	1	0	1
1	0	1	0	1	2
1	1	0	0	1	2
1	1	1	1	1	3

Figure 4.6: Full Adder Truth Table

4.4 3 Bit Register – Micro Level Design

The 3 Bit Register was produced using 3 D Flip Flops with a common synchronous clock. The synchronous clock ensures that all three Flip Flops will store their data at the same time; this prevents the incorrect value being stored if the outputs of the full adder do not all update at the same time. A schematic of the 3 Bit Register generated in KiCAD [4] can be seen in **Figure 4.7**. The Truth Table for a D Flip Flop can be seen in **Figure 4.8**.

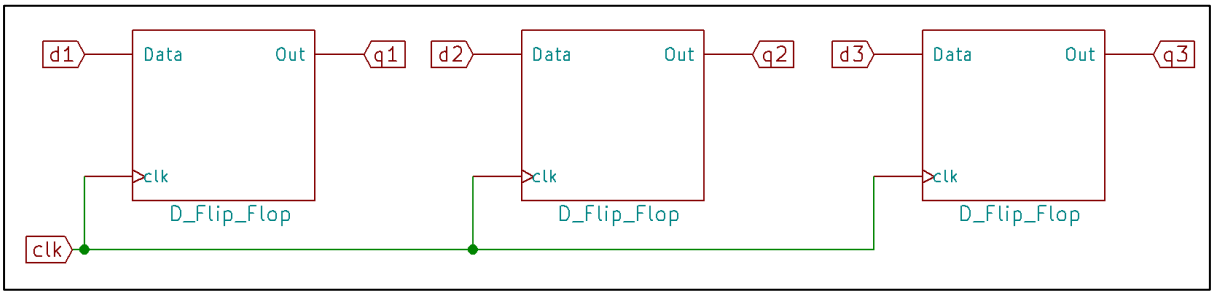

Figure 4.7: 3 Bit Register Schematic (Generated in KiCAD)

D Flip Flop Truth Table		
Inputs		Outputs
clk	Data	Out
0	x	Unchanged
1	x	Unchanged
Positive Pulse	0	0
Positive Pulse	1	1

Figure 4.8: D Flip Flop Truth Table

5 Method and Results

The implemented design fulfilled all requirements of the design problem when tested on the Xilinix NEXYS 4 FPGA development board. Due to the nature of the lab session, this method and results section will contain no further information.

6 Discussion

To verify that the implemented design was operating to the specification of the design problem, several Verilog test benches were created. Test benches are Verilog modules that instantiate the macro to be tested and then record the output of the macro for different simulated input patterns [2]. **Figure 6.1** shows the input patterns for each test bench used in the design implementation.

2 Bit Adder and 3 Bit Register - Macro Test Bench Patterns												
Macro	Full Adder			3 Bit Register				2 Bit Adder and 3 Bit Register				
Input	ain	bin	c_in	clk	d3	d2	d1	clk_in	a1	a0	b1	b0
Pattern 1	0	0	0	0	0	0	0	0	0	0	0	0
Pattern 2	0	0	1	0	0	0	1	0	0	1	1	0
Pattern 3	0	1	0	0	0	1	0	1	x	x	x	x
Pattern 4	0	1	1	0	0	1	1	0	1	0	0	1
Pattern 5	1	0	0	1	x	x	x	1	x	x	x	x
Pattern 6	1	0	1	0	1	0	0	0	1	1	1	1
Pattern 7	1	1	0	0	1	0	1	1	x	x	x	x
Pattern 8	1	1	1	1	x	x	x	0	x	x	x	x
Pattern 9	End			0	1	1	0	End				
Pattern 10				1	x	x	x					
Pattern 11				0	1	1	1					

Figure 6.1: 2 Bit Adder with 3 Bit Register Verilog Test Bench Patterns

An exhaustive test bench was used for the Full Adder macro, this means every possible input combination was tested and the corresponding output was verified against a hand drawn Truth Table. A similar method was used for testing the 3 Bit Register, however, only certain input combinations were clocked into the Register. This verified that the Register would only store new information on the rising edge of the clock.

Exhaustive testing of macros provides complete certainty that the macro is functional. As all input combinations have been tested and verified, ignoring the affects of hazards (bugs in operation due to hardware timings delays), there are no bugs in the logic circuit of the macro. Due to every input combination being tested, exhaustive testing is inefficient at top design levels due to the number of inputs the top design may have [2]. However, using exhaustive testing at micro design levels means a high level of confidence can be placed in the top-level design, even if it is not exhaustively tested [2].

The top-level design (2 Bit Adder with 3 Bit Register) test pattern was selected to ensure each input/output (IO) pin was toggled (turned from 1 to 0 or 0 to 1) at least once. This verifies the connections between the macros [2]. As exhaustive testing was used on the low-level macros (Full Adder, 3 Bit Register) that make up the top-level design, only limited testing was required to verify the designs operation to a high level of confidence. All test bench code can be viewed in the Appendices (section 9).

The use of Verilog test benches aide's regression testing (retesting implemented logic circuits after changes have been made) as the same test bench can be used to verify the operation of the updated macro. Effectively the test bench acts as a standard to which the macro must comply.

It is evident that if implemented using standard logic Integrated Circuits (ICs), the logic gate design for the Full Adder would be inefficient. Use of universal NAND and NOR gates to form the gates shown in the design from section 4.3 would optimise design cost by requiring less ICs overall to implement the logic circuit. The secondary Multiplexer based Full Adder design would also serve as a cost optimisation as it could be implemented with a single Dual 4 to 1 Multiplexer IC and an Inverter. As this current design is being tested on an FPGA this factor matters less as the synthesis tool automatically finds the optimal implementation of the logic.

Use of a Top Down Design methodology splits the overall design into a tree of macros, each at different levels of design abstraction [2]. Macros allow sections of HDL or Logic Schematics to be easily reused in other projects, further saving development time. Macros also decouple the top-level design from the lower level logic circuits, allowing for bug fixes to be applied without affecting the overall functionality of the system in a negative manner. The simplicity induced by a Top Down Design makes for a more reliable and maintainable implementation. The engineer can focus on each small section of the design individually, improving design efficiency [2]. Top Down Design also makes understanding the overall operation of the design easier as it is described as a set of connected structures at the top design level.

Ultimately, this experiment highlights the advantages of rapid prototyping and development with FPGA technology. The process is much faster than typical design methods for custom logic circuits [1] due to the ease of describing logic circuits behaviourally. As FPGA's are infinitely reconfigurable (within a reasonable lifetime) the speed of development can be further increased [1] as changes to the design can be tested immediately rather than waiting for a costly prototype to be manufactured. This all contributes to an overall reduction in cost of custom logic circuit development, whether the final implementation uses an FPGA or an ASIC.

As FPGA design tools inherently support the Top Down Design Methodology, support throughout a logic circuit lifetime is also made easier; changes can be made at micro design levels without affecting the overall operation of the logic circuit due to the use of macros at the top design level. As discussed previously, standard macros from previous projects can also be used to speed up the design process.

7 Conclusion

In conclusion, rapid prototyping custom logic circuits with FPGAs provides many benefits that are attractive to modern companies that are largely focused on reduction of time to market [1]. The reconfigurability of FPGAs allows for an efficient and effective iterative process of development [1] where changes to the logic circuit can be implemented and tested very quickly. Ultimately this reduces the overall cost of development, whether the final design is implemented using an FPGA or ASIC.

The use of Top Down Design methodologies adds to this speed and efficiency by simplifying designs into a hierarchical tree [2]. This hierarchical structure allows for easy implementation and testing of macros at the low level, providing a high level of confidence in the top-level design even if it cannot be exhaustively tested [2]. Changes can also be applied at the micro-level without affecting the operation of the logic circuit at the top-level, improving the ease of support through the logic circuits lifetime.

Finally, the implementation developed during this experiment was fully successful in meeting the requirements of the original design problem. The Top Down Design methodology used allowed for exhaustive testing of the low-level designs; consequently, high levels of confidence could be placed in the top-level design despite the relatively short amount of testing it was subjected to. Additionally, the simplicity provided by the Top Down Design methodology lead to a much easier development process. However, the implemented design could have been improved by using a universal logic solution for the Full Adder, as shown in the secondary implementation of the Full Adder using Multiplexers. This reduces cost if the final design is implemented using Integrated Circuits.

8 References

[1] P. Horowitz and W. Hill, The Art of Electronics, 3 ed., New York, NY: Cambridge University Press, 2015, pp. 782, 775, 776.

[2] M. D. Ciletti, Advanced Digital Design with the Verilog HDL, 2 ed., Upper Saddle River, NJ: Pearson, 2011, pp. 10, 109, 466, 467, 4, 6, 8, 121.

[3] M. M. Mano and M. D. Ciletti, Digital Design, 4 ed., Upper Saddle River, NJ: Pearson, 2006, p. 161.

[4] J.-P. Charras, D. Hollenbeck and W. Stambaugh, “KiCAD EDA,” KiCAD, August 2017. [Online]. Available: http://kicad-pcb.org/.

9 Appendices

9.1 Full Adder Test Bench

```
// Verilog test fixture created from schematic U:\Electrical and Electronic
Engineering\Year One\EEE119\DLC Lab\myadder\my_fa.sch - Wed Nov 08 09:40:17
2017

`timescale 1ns / 1ps

module my_fa_my_fa_sch_tb();

// Inputs
   reg bin;
   reg ain;
   reg c_in;

// Output
   wire sum;
   wire c_out;

// Bidirs

// Instantiate the UUT
   my_fa UUT (
        .bin(bin),
        .ain(ain),
        .c_in(c_in),
        .sum(sum),
        .c_out(c_out)
   );
// Initialize Inputs
//   `ifdef auto_init
       initial begin
        bin = 0; ain = 0; c_in = 0;
//   `endif
        //Add Stimulus
        #10 ain = 0; bin = 0; c_in = 1;
        #10 ain = 0; bin = 1; c_in = 0;
        #10 ain = 0; bin = 1; c_in = 1;
        #10 ain = 1; bin = 0; c_in = 0;
        #10 ain = 1; bin = 0; c_in = 1;
        #10 ain = 1; bin = 1; c_in = 0;
        #10 ain = 1; bin = 1; c_in = 1;
        end
endmodule
```

9.2 3 Bit Register Test Bench

```
// Verilog test fixture created from schematic U:\Electrical and Electronic
Engineering\Year One\EEE119\DLC Lab\myadder\my_reg.sch - Wed Nov 08
10:26:45 2017

`timescale 1ns / 1ps

module my_reg_my_reg_sch_tb();

// Inputs
   reg my_clk;
   reg d1;
   reg d2;
   reg d3;

// Output
   wire q1;
   wire q2;
   wire q3;

// Bidirs

// Instantiate the UUT
   my_reg UUT (
        .my_clk(my_clk),
        .d1(d1),
        .q1(q1),
        .d2(d2),
        .q2(q2),
        .d3(d3),
        .q3(q3)
   );
// Initialize Inputs
//   `ifdef auto_init
       initial begin
        my_clk = 0;
        d1 = 0;
        d2 = 0;
        d3 = 0;
        //Applying Stimulus
        #100 d3 = 0; d2 = 0; d1 = 1;
        #100 d3 = 0; d2 = 1; d1 = 0;
        #100 d3 = 0; d2 = 1; d1 = 1;
        #50 my_clk = 1;
        #50 d3 = 1; d2 = 0; d1 = 0; my_clk = 0;
        #100 d3 = 1; d2 = 0; d1 = 1;
        #50 my_clk = 1;
        #50 d3 = 1; d2 = 1; d1 = 0; my_clk = 0;
        #50 my_clk = 1;
        #50 d3 = 1; d2 = 1; d1 = 1; my_clk = 0;
        end
//   `endif
endmodule
```

9.3 2 Bit Adder with 3 Bit Register Test Bench

```
// Verilog test fixture created from schematic U:\Electrical and Electronic
Engineering\Year One\EEE119\DLC Lab\myadder\sync_adder.sch - Wed Nov 08
11:05:51 2017

`timescale 1ns / 1ps

module sync_adder_sync_adder_sch_tb();

// Inputs
   reg a1;
   reg b1;
   reg b0;
   reg a0;
   reg my_clk;

// Output
   wire s0;
   wire s1;
   wire c_out;

// Bidirs

// Instantiate the UUT
   sync_adder UUT (
        .a1(a1),
        .b1(b1),
        .b0(b0),
        .a0(a0),
        .my_clk(my_clk),
        .s0(s0),
        .s1(s1),
        .c_out(c_out)
   );
// Initialize Inputs
//   `ifdef auto_init
       initial begin
        a1 = 0;
        b1 = 0;
        b0 = 0;
        a0 = 0;
        my_clk = 0;
        //Applying Stimulus
        #100 a1 = 0; a0 = 1; b1 = 1; b0 = 0;
        #50 my_clk = 1;
        #50 a1 = 1; a0 = 0; b1 = 0; b0 = 1; my_clk = 0;
        #50 my_clk = 1;
        #50 a1 = 1; a0 = 1; b1 = 1; b0 = 1; my_clk = 0;
        #50 my_clk = 1;
        #50 my_clk = 0;
        end
//   `endif
endmodule
```