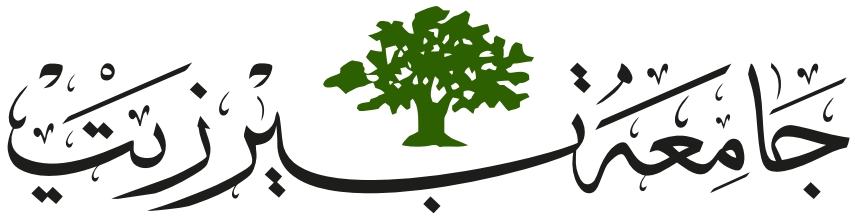

BIRZEIT UNIVERSITY

Department of Electrical & Computer Engineering
ENCS3310 - ADVANCED DIGITAL SYSTEMS DESIGN

Arithmetic Unit

Prepared by: Mohammad Abu-Shelbaia
1200198
Instructor: Dr. Abdallatif Abu-Issa
Date: August 22, 2022

Abstract

In this project, we built a multi-purpose arithmetic unit that consists mainly of a binary adder. The project is divided into two stages. The first is built using a ripple adder, while the second stage is built using a carry look-ahead adder. Finally, functional verification is applied throughout all the stages.

Contents

1 Theory

1.1 Introduction

In this project, we built an arithmetic unit that could do seven different operations; add/subtract with and without carry, increment-er, decrement-er, and a buffer. It consists of 2 parts, a multiplexer to choose the operation and an adder to execute it, as shown in (Figure: 1.1).

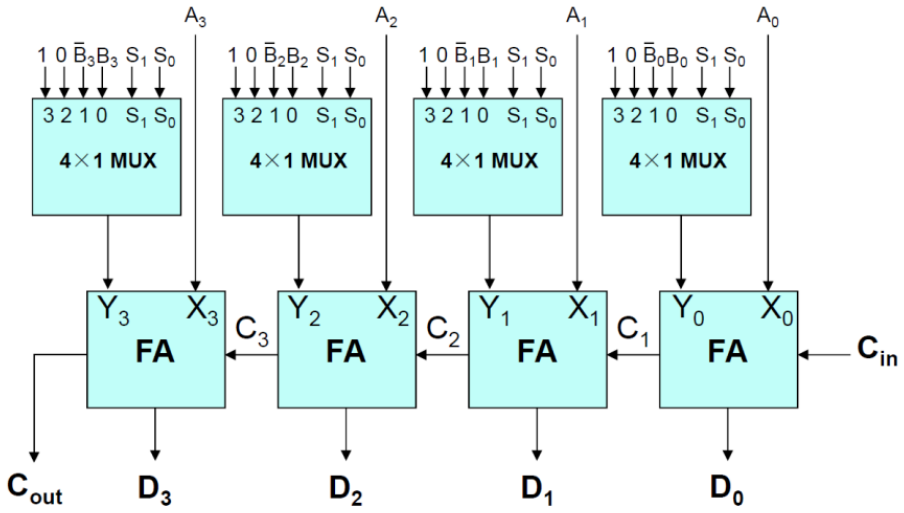

Figure 1.1: Arithmetic Unit

We replaced the four full-adders shown in (Figure: 1.1) with a 4-bit adder in two stages. The first one is a ripple carry adder, while the second one is a carry look-ahead adder. and we provided a functional testing throughout each stage that we are going to come to later in the report.

1.2 Design

We implemented a structural approach to design our arithmetic unit with only three gates, NAND, XOR, and Inverter; each gate has a delay independent of its number of inputs. The design consists mainly of NAND gates, and this design choice we implemented to reduce the delay to the minimum. We could've gone with a complete NAND design, but it comes with extra delay cost since building the other two gates with NAND will increase the delay from the original by (2ns) for the Inverter and (4ns) for the XOR. Furthermore, we can use more than 2-inputs with the standard XOR gate.

1.2.1 Full Adder

Combinational Circuit that adds 3 bits together. The design for the circuit using (NAND, XOR) gates, as shown in (Figure: 1.2), needs (10ns) to generate the Carry and (11ns) to generate the Summation, meaning the circuit takes (11ns) to generate correct outputs.

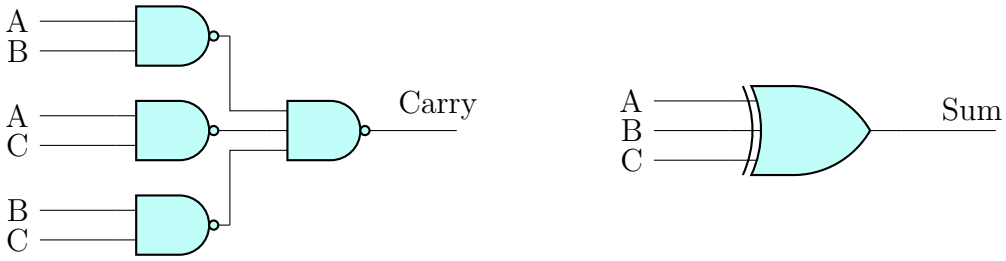

Figure 1.2: Full-Adder

1.2.2 Multiplexer

Combinational Circuit that selects between several inputs and forwards the selected input to the output line. The design for the circuit using (NAND, NOT) gates, as shown in (Figure: 1.3), has a delay of (13ns).

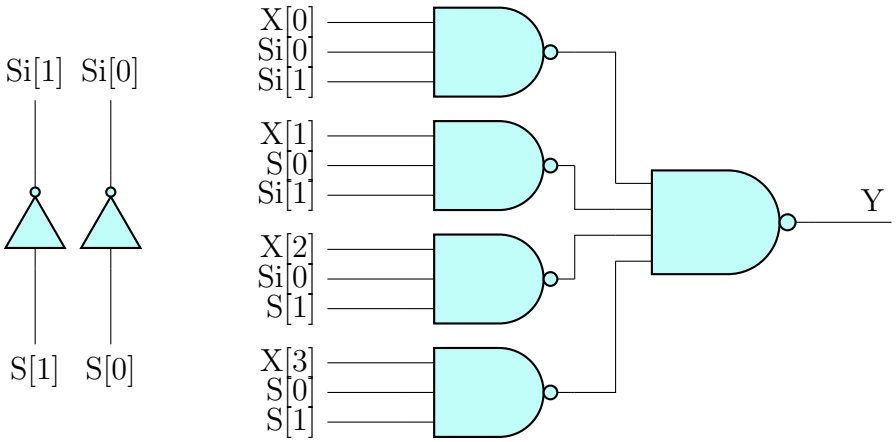

Figure 1.3: Multiplexer

1.2.3 Ripple Adder

The Ripple adder is a circuit built structurally from four full adders as shown in **??**. After testing the circuit with different values, we found one of our worst cases at A=1111, B=1001, and C = 1, which took (41ns) to evaluate correctly.

1.2.4 Look-ahead Adder

The benefits of the Look-ahead Adder over the Ripple-Adder is the reduced carry propagation delay by introducing more complex hardware. we are going to split the process in two, carry generation, and sum evaluation.

The equations Below we derived from the standard carry look-ahead equations to use NAND instated of (AND, OR) gates.

$$C1 = (G0'.(P0C0)')'$$
$$C2 = (G1'.(P1G0)'.(P1P0C0)')'$$
$$C3 = (G2'.(P2G1)'.(P2P1G0)'.(P2P1P0C0)')'$$
$$C4 = (G3'.(P3G2)'.(P3P2G1)'.(P3P2P1G0)'.(P3P2P1P0C0)')'$$

After evaluating all the needed inputs and the carry using the above equations, we can get the adder result by using the XOR gate between (A[i], B[i], C[i]), giving us the summation bits, and the generated C[4] is the output carry.

The implementation of the circuit below in (Figures: 1.4, 1.5, 1.6). After testing it with different values, we found one of our worst cases at A=1111, B=1110, and C = 1, which took (32ns) to evaluate correctly.

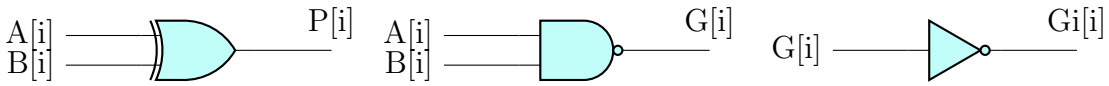

Figure 1.4: Generating Arrays

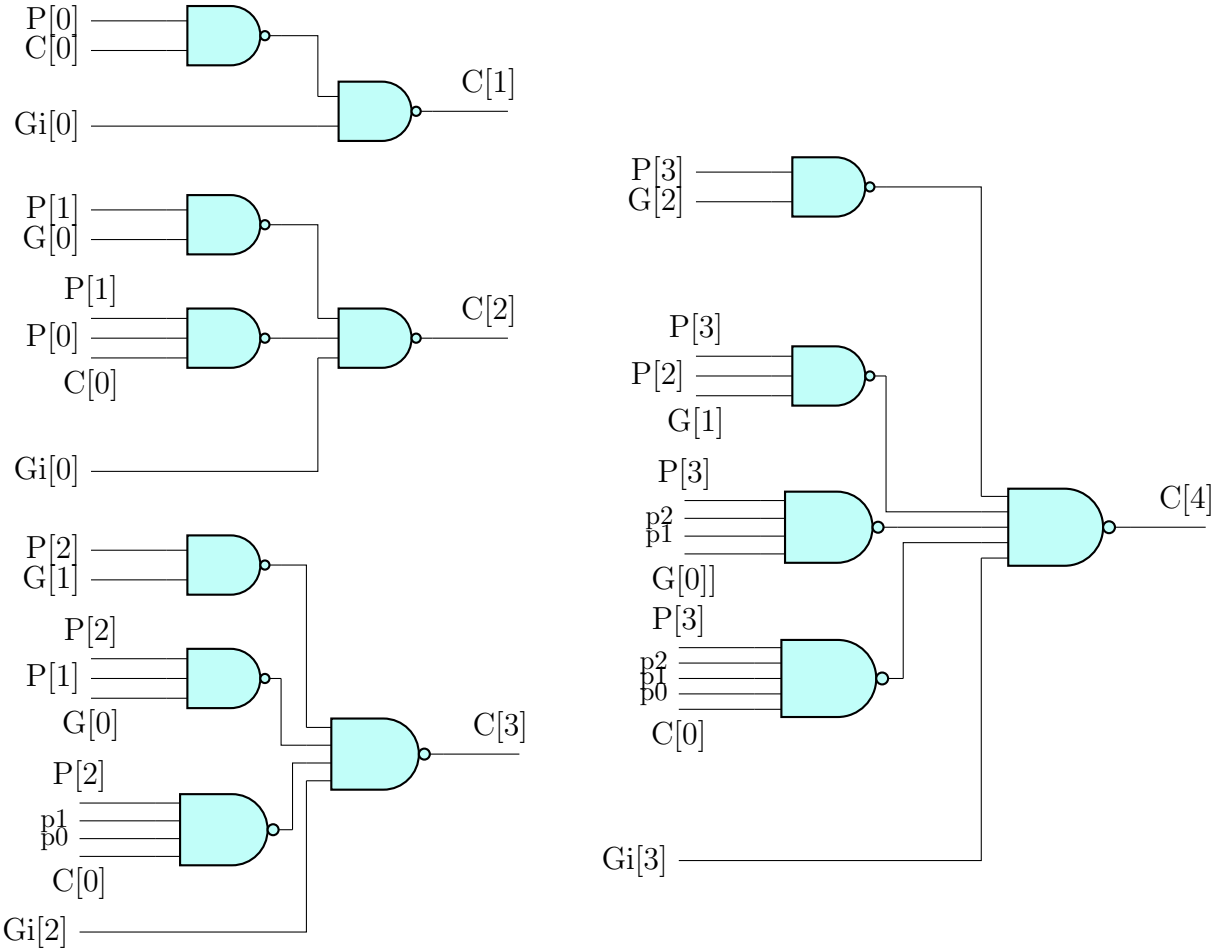

Figure 1.5: Generating Carry

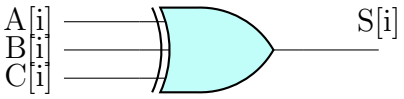

Figure 1.6: Evaluating Summation

1.2.5 Verification

The Verification bench we built has two components, the Test-Generator and the Result-Analyzer. The first component generates test vectors and their respective result using the counter method and sends the input vectors to the Arithmetic-Unit. At the same time, it sends them to the Result-Analyzer. The second component takes the actual result evaluated using the arithmetic unit and asserts an error if the result is not the same as the expected result.

As shown in (Figure: 1.7), the circuit is an asynchronous circuit. Our goal is to set the clock so that the Result-Analyzer starts analyzing after the arithmetic unit has finished evaluating its result. and as we notice that we needed only one clock to complete our design. Furthermore our design can test one vector at each clock cycle/pulse.

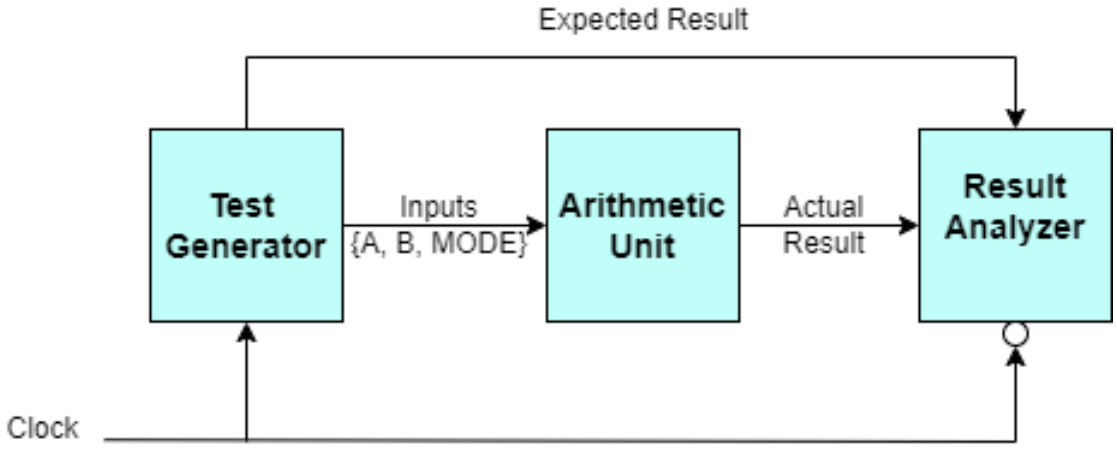

Figure 1.7: Verification

To check if our verification bench is working correctly, we introduced a bug in our program to simulate some errors that might happen in real-life scenarios; for example, we set our 5th summation bit to be stuck at zero, and since we are working with 2048 test vectors we got a tall list of errors one of them was: "ERROR: EXPECTED_RESULT=01111 , ACTUAL_RESULT=11111". as for stage two we set the first carry generator to use AND gate instated of NAND (Figure: 1.5), hence we got our second summation bit wrong for most of the cases one of them was: "ERROR: EXPECTED_RESULT=11100 , ACCTUAL_RESULT=11110".

Initially, for debugging purposes, we sent the test vectors generated from the Test-Generator along with the expected result to the Result-Analyzer to identify at which vector the error happens. We found out that some case causes errors because of the way the compiler evaluates the expressions (e.g., B complement), or in some cases, the carry needs to be ignored (e.g., A Transfer). To overcome this problem, we edited the issue the error happens at to match the result from the arithmetic unit. as shown in (Module: 3.6). After we have finished debugging, we could remove the test vector we added, but we included them for educational purposes.

1.3 Results

With the design choices and the delays of each block we evaluated earlier, we get that the total delay of the arithmetic unit takes 54ns for stage one and 45ns for stage two. Hence the clock we will use for our verification unit will be at a minimum of 1/110 Hz for stage one and 1/90 Hz for stage two. We could have used a faster clock, but it comes with a cost that we need more than one clock cycle to test a vector. Or we could have used two different clocks, but it comes with extra hardware cost.

As shown above, the difference between stage one and stage two is only 10ns. In our case, with a maximum of 2048 cases saving 2048ns is not a big deal, but if we think of changing our design to include more bits, the complexity of the Look-Ahead Adder will increase even more than it already is. On the other hand, increasing the number of bits in the Ripple-Adder will only cost another Full-Adders.

Lastly, we didn't include any waveform because we used the terminal to display data which is faster and more efficient, and we created a test bench to check if adders in both stages work correctly, check (Module: 3.9).

2 Conclusion

In conclusion, this project aimed to simulate an actual hardware design where gates have delays and to find a way to reduce the latency to a minimum. Furthermore, how to deal with the latency in the verification step. As for future work, we can implement a Built-in self-test design that uses LFSR for the Test-Generator, and MISR For the Result-Analyzer instated of the counter method, which would save much hardware complexity and has more practical uses.

List of Figures

List of Modules

3 Appendix

```
module Full_Adder(input a, b, c, output carry, sum);
        wire z1, z2, z3;
        xor          #(11ns) (sum, a, b, c);
        nand         #(5ns)  (z1, a, b), (z2, a, c), (z3, b, c),
         ↪  (carry, z1, z2, z3);
endmodule
```

Modules 3.1: Full Adder

```
module MUX41(input [3:0] X,input [1:0] S, output Y);
        wire [3:0]        Z;
        wire [1:0]        Si;
        not  #(3ns) (Si[0], S[0]), (Si[1], S[1]);
        nand #(5ns) (Z[0], X[0], Si[1], Si[0]), (Z[1], X[1], Si[1],
         ↪  S[0]), (Z[2], X[2], S[1], Si[0]), (Z[3], X[3],S[1],
         ↪  S[0]), (Y, Z[0], Z[1], Z[2], Z[3]);
endmodule
```

Modules 3.2: Multiplexer

```
module Ripple_Adder(input [3:0]A, B, input cin, output [4:0]
↪  SUM);
        wire [4:0] c; genvar i;
        assign c[0] = cin, SUM[4] = c[4];
        generate
                for (i=0; i < 4; i = i + 1)
                                Full_Adder FA(A[i], B[i], c[i],
                                 ↪  c[i+1], SUM[i]);
        endgenerate
endmodule
```

Modules 3.3: Ripple Adder

```
module LookAhead_Addder(input [3:0] A, B, input cin, output [4:0]
↪  SUM);
        wire [3:0] P, G, Gi;
        wire [4:0] C;
        wire [9:0] Z;
        assign C[0] = cin, SUM[4] = C[4];
        genvar i;
        generate
        for(i=0; i < 4; i = i + 1)
                begin
                xor  #(11ns) (P[i], A[i], B[i]);
                not  #(3ns)  (G[i], Gi[i]);
                nand #(5ns)        (Gi[i], A[i], B[i]);
                xor  #(11ns) (SUM[i], P[i], C[i]);
                end
        endgenerate
        // NAND =          (G0' . (P0Cin)')'
        (C[1], Gi[0], Z[0]),
        (Z[0], P[0], C[0]);
        // NAND =        (G1' . (P1G0)' . (P1P0Cin)')'
        (C[2], Gi[1], Z[1], Z[2]),
        (Z[1], P[1], P[0], C[0]),
        (Z[2], P[1], G[0]),
        // NAND =        (G2' . (P2G1)' . (P2P1G0)' . (P2P1P0Cin)')'
        (C[3], Gi[2], Z[3], Z[4], Z[5]),
        (Z[3], P[2], P[1], P[0], C[0]),
        (Z[4], P[2], P[1], G[0]),
        (Z[5], P[2], G[1]),
        // NAND =        (G3' . (P3G2)' . (P3P2G1)' .
        ↪  (P3P2G1)'        . (P3P2P1G0)' . (P3P2P1P0Cin)')'
        (C[4], Gi[3], Z[6], Z[7], Z[8], Z[9]),
        (Z[6], P[3], P[2], P[1], P[0], C[0]),
        (Z[7], P[3], P[2], P[1], G[0]),
        (Z[8], P[3], P[2], G[1]),
        (Z[9], P[3], G[2]);
endmodule
```

Modules 3.4: Look-Ahead Adder

```
module Arithmetic_Unit(input [3:0] A, B, input [2:0] mode, output
↪  [4:0] D);
        wire [3:0] Bi, Z;
        genvar i;
        generate
        for (i=0; i < 4; i = i + 1) begin
                        not #(3ns) (Bi[i], B[i]);
                        MUX41 M4({1'b1,1'b0, Bi[i], B[i]},
                         ↪  mode[2:1], Z[i]);
                end
        endgenerate
        //        Ripple_Adder          RA(A, Z, mode[0],
        ↪  D[4:0]);                  //Stage 1
        LookAhead_Addder LA(A, Z, mode[0], D[4:0]);
         ↪                    //Stage 2
endmodule
```

Modules 3.5: Arithmetic Unit

```
module Tests_Generator(input clk, output reg [3:0] A, B, output reg
↪  [2:0] MODE, output reg [4:0] RES);
        integer counter=0, enabled=1;
        always @ (posedge clk)
                if (enabled) begin
                    {MODE, A, B} = counter;
                        counter = counter + 1;
                        case (MODE)
                                0: RES = A + B;
                                1: RES = A + B + 1'b1;
                                2: RES = A + {1'b0,~B};
                                3: RES = A + {1'b0,~B}+ 1'b1;
                                4: RES = A;
                                5: RES = A + 1'b1;
                                6: RES = A + 4'b1111;
                                7: RES = {1'b1,A};
                                 ↪
                        endcase
                        if (counter == 2**11) enabled = 0;
                end
endmodule
```

Modules 3.6: Tests Generator

```
module Analyzer(input clk, input [4:0] EXPECTED_RESULT,
↪  ACTUAL_RESULT, input [2:0] mode, input [3:0] A, B);
        always @ (negedge clk)
                if (ACTUAL_RESULT != EXPECTED_RESULT)
                        $display("ERROR: EXPECTED_RESULT=%b ,
                        ↪  ACTUAL_RESULT=%b, MODE=%b, A=%b, B=%b" ,
                        ↪  ACTUAL_RESULT, EXPECTED_RESULT, mode, A,
                        ↪  B);
endmodule
```

Modules 3.7: Result Analyzer

```
module System;
        reg  [3:0] A, B;
        reg  [2:0] MODE;
        reg         [4:0] ACTUAL_RESULT;
        wire [4:0] EXPECTED_RESULT;
        reg clk = 0;
        Tests_Generator         TG (clk, A, B, MODE,
        ↪  EXPECTED_RESULT);
        Arithmetic_Unit         AU (A, B, MODE, ACTUAL_RESULT);
        Analyzer                        AZ (clk, EXPECTED_RESULT,
        ↪  ACTUAL_RESULT, MODE, A, B);
        always #55ns clk = ~clk; //55ns for Ripple-Adder 45ns For
        ↪  Look-Ahead Adder
        initial #250us $finish;
endmodule
```

Modules 3.8: Verification System

```
//Extra Test
module Test;
        reg [3:0] A, B;
        reg CIN;
        wire [4:0] RESULT;
        integer counter = 0, counter2 = 1, maxtime = 0;
        integer ctime;
        //LookAhead_Addder          AU (A, B, CIN, RESULT);
        Ripple_Adder          AU (A, B, CIN, RESULT);
        initial begin
                {A, B, CIN} = counter;
                counter = counter + 1;
                ctime = $time;
                repeat(2**9 - 1)begin
                        {A, B, CIN} = 11'bx;
                        #70ns
                        {A, B, CIN} = counter;
                        counter = counter + 1;
                        ctime = $time;
                end
                #100ns $display("MAX TIME = %0d", maxtime);
        end
        always @ (RESULT)
                if (RESULT == A + B + CIN) begin
                        $display("Time %0d | A = %b | B = %b | CIN =
                        ↪  %b | RESULT = %b | %0d", ($time -
                        ↪  ctime)/1000, A, B, CIN, RESULT,counter2);
                        counter2 = counter2 + 1;
                        maxtime = ($time - ctime > maxtime)?
                        ↪  $time-ctime : maxtime;
                        end
endmodule
```

Modules 3.9: Adder's Test-Bench