Reversible Adder Implementation in VHDL

Homework 1 Report

Digital Design Using HDL (ECE 590)
Portland State University, Spring 2006

Alejandro Y. Pérez V.
PSU ID: 920 01 8090

1 Introduction

In 1961, a physicist from IBM called Rolf Landauer, proved that when data is lost in an irreversible circuit, that data is dissipated in the form of heat [1-2]. The sprouting of this principle have made a whole revolution in all the fields of logic implementation, and even though it still cannot be perfectly implemented, it's union with quantum computing will lead to heat-less computers avoiding the need of heat-sinks, fans, and also the batteries would last longer. For this reason the scientific community has shown continuous interest in matching the already existing logic principles to avoid the No-Free-Lunch theorem as much as possible.

2 Design Theory

First of all, we need to know that in order to build a reversible circuit we must use reversible gates[3]. The reader can learn more about the history of reversible logic by referring to [4].

There are different ways to implement a reversible adder. These different implementations depend on a balance between gates count, garbage outputs, ancilla bits and quantum cost [5-6]. Some of the most used universal quantum gates[7] and their quantum cost are shown in figures 1.1, 1.2 and 1.3.

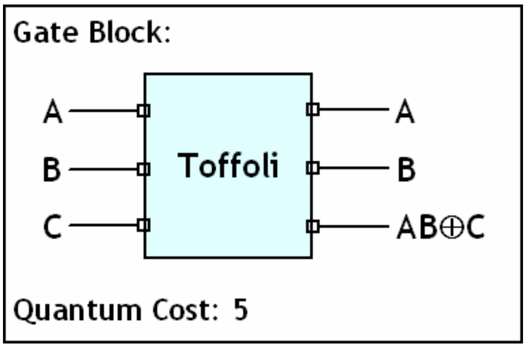

Figure 1.1 Toffoli Quantum Gate

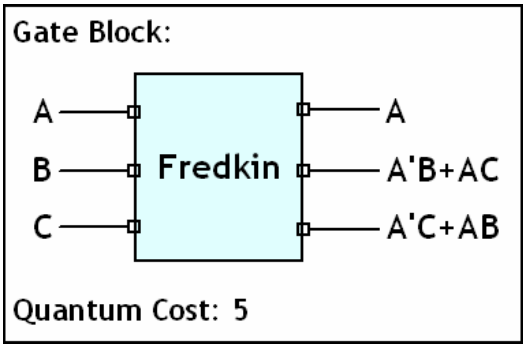

Figure 1.2 Fredkin Quantum Gate

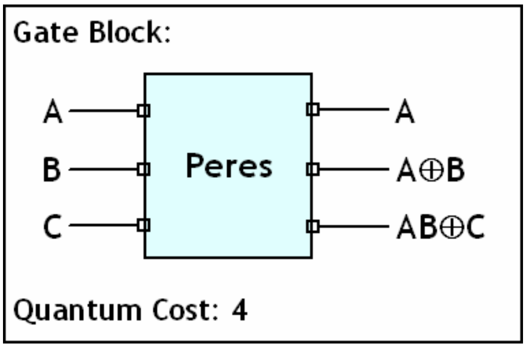

Figure 1.3 Peres Quantum Gate

Being universal, these previously presented gates can implement any logical function and therefore they can also implement the well known functions for a full-adder:

$$S = A \otimes B \otimes Cin$$

$$Cout = (A \otimes B)Cin \otimes AB$$

For this implementation, I will be using the Peres gate as it is the gate with the lower quantum cost as can be seen in the figures 1.1, 1.2 and 1.3. The Peres' implemented Full Adder with its corresponding quantum cost can be seen below:

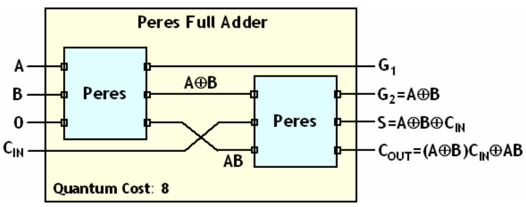

Figure 1.4 Peres Full Adder

This PFA (Peres Full Adder) can be taken as a block in order to facilitate the notation of its expansion. The inputs order was also changed to better fit in an expansion diagram.

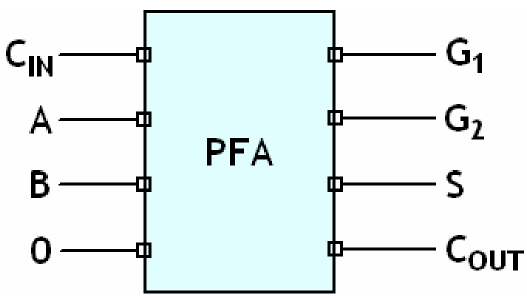

Figure 1.5 PFA as a block

Once we take the FPA as a block, we can derive the algorithm to implement an n-bits adder. This algorithm was implemented in this design and can be seen in figure 1.6.

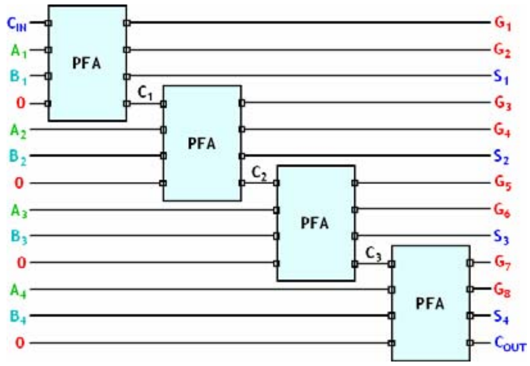

Figure 1.6 4-bits adder implementation

3 Inputs

The inputs of the complete 4 bits adder are three input vectors (4 bits) and a single bit Cin (Carry in). Two of the three input vectors are the desired added 4-bits values. The remaining vector could be called the ancilla vector which is filled with zeros.

4 Outputs

The outputs of the system are one garbage vector of 8 bits, one sum vector of 4 bits and a Cout (Carry out) bit. Unfortunately, as can be seen, the garbage cost to realize this system is very high

5 Schematic Diagram

I tried to make the VHDL code as simple as possible, so I recurred to the Tops Down design technique. First I implemented the main architecture in a general schematic involving each Peres Full Adder as a black box. This design is suited for 4 plus 4 bits.

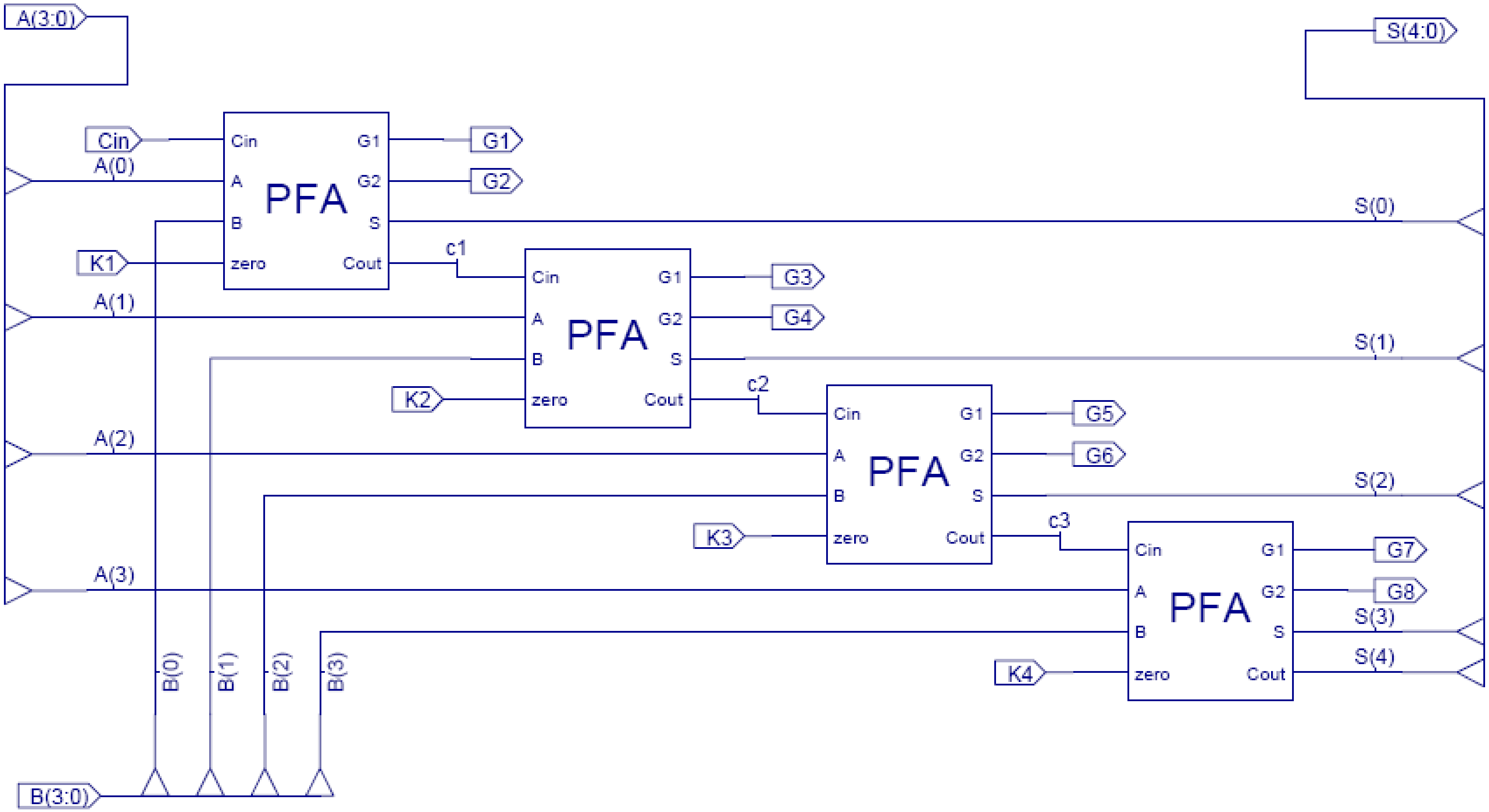

Figure 5.1 Schematic representation of the "Main" architecture.

Once that I had this, I proceeded to design the PFA block as depicted in the following figure.

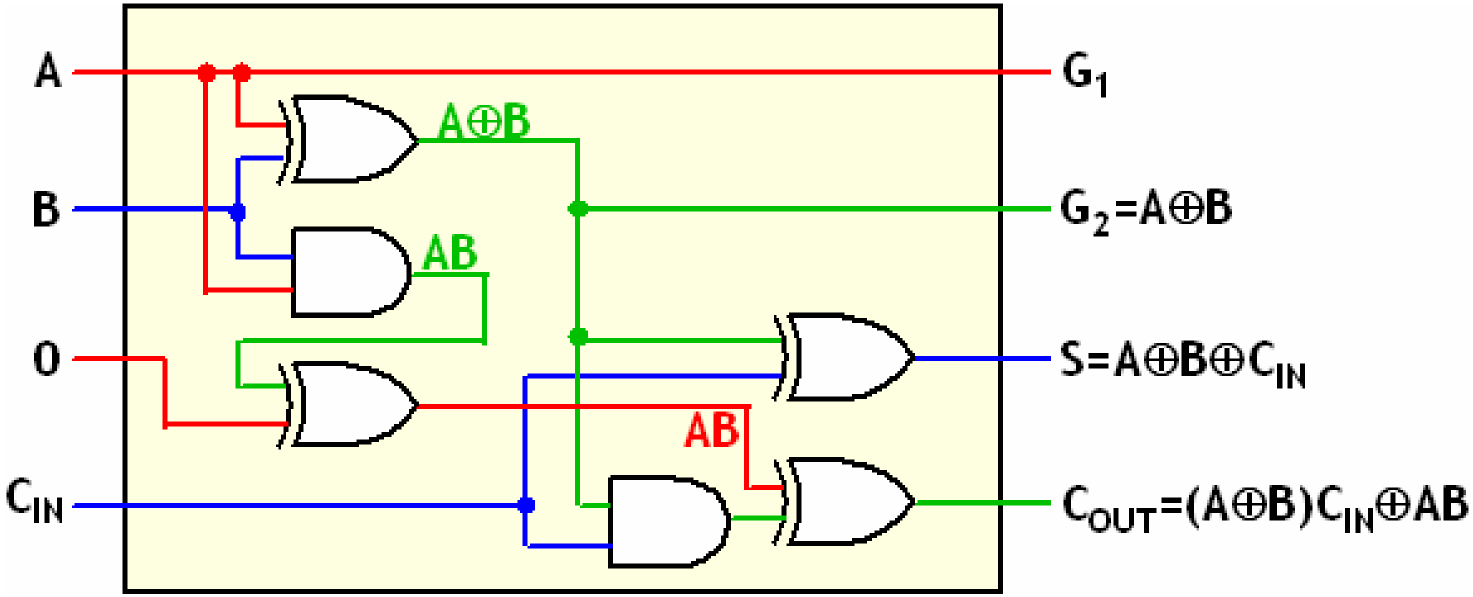

Figure 5.2 PFA traditional logic implementation.

6 VHDL Source Code

I tried to make the code as simple as possible, so I recurred to the Tops Down design technique.

6.1 “Main” Code

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;

entity mainsch is
   port ( A   : in    std_logic_vector (3 downto 0);
          B   : in    std_logic_vector (3 downto 0);
          Cin : in    std_logic;
          K1  : in    std_logic;
          K2  : in    std_logic;
          K3  : in    std_logic;
          K4  : in    std_logic;
          G1  : out   std_logic;
          G2  : out   std_logic;
          G3  : out   std_logic;
          G4  : out   std_logic;
          G5  : out   std_logic;
          G6  : out   std_logic;
          G7  : out   std_logic;
          G8  : out   std_logic;
          S   : out   std_logic_vector (4 downto 0));
end mainsch;

architecture BEHAVIORAL of mainsch is
   signal c1  : std_logic;
   signal c2  : std_logic;
   signal c3  : std_logic;
   component PFA
      port ( Cin  : in    std_logic;
             A    : in    std_logic;
             B    : in    std_logic;
             zero : in    std_logic;
             G1   : out   std_logic;
             G2   : out   std_logic;
             S    : out   std_logic;
             Cout : out   std_logic);
   end component;

begin
   XLXI_1 : PFA
      port map (A=>A(0),
                B=>B(0),
                Cin=>Cin,
                zero=>K1,
                Cout=>c1,
                G1=>G1,
                G2=>G2,
                S=>S(0));

   XLXI_2 : PFA
      port map (A=>A(1),
                B=>B(1),
```

```
                Cin=>c1,
                zero=>K2,
                Cout=>c2,
                G1=>G3,
                G2=>G4,
                S=>S(1));

   XLXI_3 : PFA
      port map (A=>A(2),
                B=>B(2),
                Cin=>c2,
                zero=>K3,
                Cout=>c3,
                G1=>G5,
                G2=>G6,
                S=>S(2));

   XLXI_4 : PFA
      port map (A=>A(3),
                B=>B(3),
                Cin=>c3,
                zero=>K4,
                Cout=>S(4),
                G1=>G7,
                G2=>G8,
                S=>S(3));

end BEHAVIORAL;
```

6.2 "PFA" Code

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;

entity PFA is
   port ( A    : in    std_logic;
          B    : in    std_logic;
          Cin  : in    std_logic;
          zero : in    std_logic;
          Cout : out   std_logic;
          G1   : out   std_logic;
          G2   : out   std_logic;
          S    : out   std_logic);
end PFA;

architecture BEHAVIORAL of PFA is
  signal AxB: std_logic;        -- This signal is going to be (A xor B)
  signal AB: std_logic;         -- This signal is going to be (A and B xor zero)
begin

  AxB  <= A xor B;              -- First I am loading
  AB   <= (A and B) xor zero;   -- the auxiliar signals
  G1   <= A;                    -- Garbage 1
  G2   <= AxB;                  -- Now I am doing the rest
  S    <= AxB xor Cin;          -- of the operations
  Cout <= (AxB and Cin)xor AB; -- Carry out

end BEHAVIORAL;
```

7 Simulation And Results

The project was simulated with the help of the Xilinx ISE 7.1 tool. Remember that the real inputs for this project were the 4 bits A and B and the 1 bit Cin (Carry In). The rest (K1 through K4) are only the ancilla bits and they need to remain always in zero.

Figure 7.1 Timing diagram of the simulation.

Important Note: The system outputs are shifted to the left because of a bug in the tool's simulator.

8 References

1. Landauer, R., 1961. Irreversibility and heat generation in the computing process. IBM J. Res. Develop., 5: 183-191.
2. Keyes, R.W. and R. Landauer, 1970. Minimal energy dissipation in logic. IBM J. Res. Develop., pp: 152-157.
3. Bennett, C.H., 1973. Logical reversibility of computation. IBM J. Res. Develop., 17: 525-532.
4. Bennett, C.H., 1988. Notes on the history of reversible computation. IBM J. Res. Develop., 32: 16-23.
5. Perkowski, M., L. Jozwiak, P. Kerntopf, A. Mishchenko and A. Al-Rabadi et al., 2001. A general decomposition for reversible logic. In: 5th Intl. Red-Muller Workshop, pp: 119-138.
6. Saiful Islam and Rafiqul Islam., 2005. Minimization of Reversible Adder Circuits. Asian Journal of Information Technology 4 (12): 1146-1151.
7. Peres, A. 1985. Reversible logic and quantum computers. Physical Review A, 32: 3266-3276.