



BEng Electrical & Electronic Engineering

Electronics II

Design Of an Unsigned Two Number Uneven Parallel Multiplier
and the Design of a FSM in Hardware and VHDL

By Aneurin Creed

Lecturer: Dr. Gary L Jones

Project submitted as part of the requirements for the award of BEng (Hons)
Electrical and Electronic Engineering

May 2021

Contents

Contents	2
Task 1	3
Introduction & Background Information	3
AND Gates	3
Half Adder	4
Full Adder	5
Multiplication Process	6
Solution to the Task (Procedure / Results)	7
Proteus Design	7
VHDL Synthesis	9
File 1 - Half Adder	9
File 2 - Full Adder	10
File 3 - multi12x4_Line_0	11
File 4 - multi12x4_Line_x	12
File 5 - multi12x4_Full	15
Conclusion	18
Task 2	19
Background Information	19
Class A / Mealy Machines	19
Class B / Moore Machines	20
Solution to the Task (Procedure / Results)	21
Discussion	24
Conclusions	24
References	24

Task 1

Introduction & Background Information

The aim of this task was to design, simulate and synthesise a 12 bit x 4 bit unsigned number uneven parallel multiplier. This was achieved by using Proteus to design the hardware and VHDL was used to simulate and synthesise a parallel Braun multiplier. Adders and Multipliers play a vital role in the functioning of various systems used in communication and signal processing. There are several different multiplier modules that exist to perform similar functions, however, Braun multipliers employ parallel architecture and hence these are the most commonly used multipliers for unsigned operations [1]. The structure of a Braun multiplier consists of AND gates, half adders and full adders [2].

AND Gates

An AND gate is a basic logic gate that is used within a digital circuit in order to produce a desired logical function [3]. In the case of a Braun multiplier, it is used to receive two inputs and gives an output based on logic. For example, as depicted in *fig 1*, to get an output of logic 1, both inputs need to be of logic 1. Any other input combination of 0 and 1, 1 and 0 or 0 and 0 results in a logic 0 output. The output of a 2-input AND gate gives the same result as multiplying two single bits. The number of AND gates used per line are dependent on the number of bits within the multiplicand. The number of lines are dependent on the number of bits within the multiplier, i.e 12×4, would result in 12 AND gates and 4 lines..

Fig 1. *AND gate*

Binary multiplication and decimal multiplication follow the same process in producing a product as the result. Whereas decimal works with numbers 0, 1, 2... through to 9, binary only uses 0s and 1s. Due to this there are only four scenarios to consider when multiplying individual binary bits, these are;

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

Half Adder

Binary Adders are arithmetic circuits in the form of half-adders and full-adders used to add together two binary digits [4]. A half adder has two inputs, A and B, and two outputs, Sum and Carryout. The Sum output uses an XOR gate as shown in *fig 2* and the Carryout output uses an And gate as shown in *fig 1*. The Carryout output becomes the third input to the next full adder.

Fig.2 *Exclusive OR gate* [5]

For the sake of further understanding it is also important to acknowledge binary addition. This too is the same as decimal addition, the only difference is the value representation, such that;

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

Full Adder

A full adder is a digital adder which is used to compute the sum of three or more binary bit numbers. The outputs of the full adder include one partial sum bit and one carryout bit [2]. The Sum output uses two XOR gates as shown in *fig 3 and fig 4* and the Carryout output uses an XOR gate and an And gate, or just an AND gate as shown in *fig 5 and fig 6*. As there are 4 lines in this multiplier then there is potential for partial products of 1 in the same column and a Carryout of 1. This results in a total sum of 5 single bits, giving an output of 101. However, the use of these adders in this way simplifies the addition of partial products and limits the maximum carry at each adder to a single bit output.

Fig. 3

Fig. 4

Fig. 5

Fig.6

Multiplication Process

A Braun multiplier consists of two operands; the first operand being the multiplicand and the second operand being the multiplier. Each bit of the multiplier is multiplied by all the bits of the multiplicand in order to produce the partial products. The first row of partial products is produced by multiplying the first bit of the multiplier with the multiplicand. The second row of partial products is produced by multiplying the second bit of the multiplier with the multiplicand. In this way, all of the partial products are generated [5]. Each multiplication producing a partial product is achieved by using an AND gate per product output.

As depicted in *fig 7* : All the partial products produced are together known as the partial product array. The final result is produced by adding all the bits present in each column [5]. This addition of partial products is achieved via adders.

Fig.7, *Multiplication of two 4-bit unsigned numbers* Unsigned Array Multiplier

Solution to the Task (Procedure / Results)

Proteus Design

Fig. 8

The first step in creating this multiplier was to produce a design within Proteus. The design depicts the process of multiplying the 12 multiplicands A1-A12 by the 4 multipliers B1-B4. The design is made up of 4 lines. Each line

uses all 12 multiplicands and multiplies them sequentially by one of the multipliers - e.g, line 1 uses B1 as the multiplier, line 2 uses B2, and so on.

Line 1 uses 12 AND gates, the inputs of the first AND gate are B1 and A1. The output Q0 is the final output in the 16 bit answer. The remaining 11 AND gates outputs are Q1-Q11. However, these output bits do not give the final answer as there are more iterations to be multiplied by and added to. Line 1 differs from the other three lines and is referred to as line_0 as it only requires AND gates and no adders because there are no prior outputs to consider.

Line 2 is the first to use the line_x process. The line_x process follows the same initial process as line 1 where all 12 multiplicands are multiplied by the B multiplier. However, as it is line 2, it uses B2. the 12 products of these multiplications are added with the outputs of the previous line via an array of adders. The first adder in the array is a half adder as it only receives two inputs. The following eleven adders in the array are all full adders as they take the carryout output of the previous adders as the third input. The output from the half adder becomes Q1 in the final output 16-bit answer. The following 12 outputs - 11 sum outputs and the final carryout output - become the inputs to the array of adders in the next line.

Fig. 9

Line 3 uses line_x again. The only difference is that it uses B3 as the multiplier. As mentioned in Line 2, the outputs from the adder array are added to the products of the AND gate array in this line. Subsequently, the outputs of this line also become the inputs to the next line, except for Q2, which is the final output in the 16-bit answer.

Line 4 uses line_x again. The only difference is that it uses B4 as the multiplier. However, as this is the last line in the multiplication, all of the adder array outputs are the final outputs in the 16-bit answer. This includes the last carry of the final adder, which becomes Q15 - the final and 16th bit in the 16-bit answer.

VHDL Synthesis

File 1 - Half Adder

Using Xilinx, a VHDL was produced starting with creating a file called 'half adder', where the inputs and outputs were defined. The inputs consisted of A and B and the outputs were Sum and Carry. The half adder is used as the first in the array of adders as it receives only two inputs. When adding two bits, the potential outcomes are 0, 1 and 10. As there is a potential for the outcome to be 10, the addition of both of these two bits requires both a Sum output and a Carry output. As discussed above, these two outputs are defined differently. This means that the combination of both the Sum and Carry provide the output required to add two single bits.

Fig.10

Once the VHDL was synthesised, it was then simulated using the test bench feature within the software to see if it produced the desired outcome.

Fig. 11

These four images show that the correct outputs of Sum and Carry have been achieved. As shown in the images, if both A and B are 0 then the Sum and Carry are 0. If either A or B is 1, the Sum is 1 and the Carry is 0. If both A and B are 1, the Sum is 0 and the Carry is 1.

File 2 - Full Adder

Next, a new file called 'full adder' was created where the inputs and outputs were defined. The inputs consisted of A, B and C and the outputs were Sum and Carry. The full adders are used in the array after the first half adder. Of the three inputs that the full adders receive, input C is the Carryout output from the previous adder.

Fig. 12

Once the VHDL was synthesised, it was again simulated using the test bench feature within the software to see if it produced the desired outcome.

Fig. 13

These eight images show that the correct outputs of Sum and Carry have been achieved. As shown in the images, if all A, B and C are 0 then the Sum and Carry are 0. If either A or B or C is 1, the Sum is 1 and the Carry is 0. If either A or B is 1 and C is 1, or A and B are 1 the Sum is 0 and the Carry is 1. If A, B and C are 1 then both Sum and Carry are 1.

File 3 - multi12x4_Line_0

Fig. 14

The image on the left shows all 13 inputs (A1-A12, the multiplicand) and B1 - the first of the 4-bit multiplier. Also included are the 12 Q outputs (Q0-Q11) as well as showing how the Q outputs are produced - i.e, Q0 is defined as A1 AND B1.

The image on the right shows the test bench used to simulate line_0. The test bench here uses a Stimulus process to hold the state of a chosen input for a specified time. Here all inputs are held in doubling increments - i.e, stim_proc1 holds A1 for 10ns, stim_proc2 holds A2 for 20 ns. This continues until stim_proc13, where B1 is held for 40960 ns.

Fig. 15

Fig. 17 shows the virtual simulation of the test bench. Once B1 is held at 1, as the multiplicands increase, the output Q also increases at the same rate. The Q value is the same as the sum of the multiplicands because the sum of the multiplicands is multiplied by B1. For example, if B1 is 0, the output Q is 0. However, if B1 is 1, the output is equal to the sum of the multiplicands.

File 4 - multi12x4_Line_x

This image shows all 25 inputs, 12 of which are A1-A12 multiplicand. One is unassigned multiplier B and the remaining 12 are P1-P12, which represent the final outputs of the previous line. The image also shows the 13 defined Q outputs.

Fig. 16

Fig. 19 shows the integration of two external components. These are the half and full adder files previously discussed. Involving these files here embeds their function within this file, line_x.

Fig. 20 shows two sets of signals which have been defined - S0-S11 and C1-C11. These have been set up here to be used later on.

Fig. 19

Fig. 20

Fig. 21 shows the AND gate sequence and how the signals S0-S11 are defined - i.e S0 is defined as A1 AND B.

Fig. 21

Fig. 22

The images above show how the half adder sequence, HA1 : halfadder, is implemented. The two inputs; A and B and two outputs; Sum and Carry are defined here as S0, P1, Q1, C1.

-S0 is the first input to the half adder which is the AND gate output of A1 AND B.

-P1, the second input to the half adder, which is the first output of the previous line that does not go straight to the final 16 bit answer. This is because it requires further attention.

-Q1 is the result of the addition of the two inputs by the half adder.

-C1 is the potential carry from the sum of the two inputs.

After the half adder, the first of 11 full adders in this array is implemented. The three inputs; A, B and C and two outputs; Sum and Carry, are defined here as S1, P2, C1, Q2, C2.

-As with the half adder, S1 is the same as the aforementioned S0, but represents the AND gate output of A2 and B.

-P2 follows on from the aforementioned P1, but uses the output that proceeds it.

-C1 is the potential carry from the previous adder.

-Q2 and C2 are the same as above.

The final full adder in the array follows the same patterns for all three inputs and for the Sum output. However, for the output Carry, as this has no adder to carry on to, it becomes an output Q13.

Fig. 23

Fig. 23 shows the test bench used to simulate line_x. The test bench here is used in the same way as it was used to simulate line_0. Again all inputs are held in doubling increments - i.e, the stim_proc1 holds A1 for 10ns, stim_proc2 holds A2 for 20 ns. This continues until stim_proc25, where B is held for 167772160 ns.

Fig. 24

Fig. 24 shows the virtual simulation of the test bench. Once B is held at 1, as the multiplicands increase, the output Q should also increase at the same rate. However, a potential fault has been identified here in that the predicted output should be proportional to the input here as there is no P value to add to it. What is shown in the simulation above is that only when the input reaches a multiple of 4, does the output also match the input.

Fig. 25

However, when P became 1 or greater, Q increased as predicted because line_X essentially multiplies the multiplicands by B1 and adds P to it. It is unclear why this fault occurs, however, the rest of the synthesis was carried out with this section involved and did not affect the final product.

File 5 - multi12x4_Full

Fig. 26 shows all 16 inputs, 12 of which are A1-A12 multiplicand. Four are the multiplier bits B1-B4. The image also shows the Q outputs for the final 16-bit answer.

Fig. 26

Fig.27 shows the integration of the file line_0 as a component within this file so that line_0 can be used as the first line of the Braun multiplication. Involving the inputs and outputs of line_0 embeds their function within this file.

Fig.27

Fig. 28 serves the same purpose as *fig.27*, but for `line_x`.

Fig. 28

Fig. 29 shows the hierarchy of files embedded within this file (`multi12x4_Full`). It depicts how `line_1` uses `line_0` and lines 2, 3 and 4 use `line_x`. This also means that one half adder and eleven full adders are embedded as a result of this.

Fig. 29

Fig.30 shows three sets of signals which have been defined - P1_1-P1_11, P2_1-P2_11 and P3_1-P3_11. These have been set up here to be used later on. The image on the left

Fig.30

Fig.31

Fig. 31 outlines how lines 1 - 4 have been used within the VHDL.

Fig.32

Fig. 32 shows the test bench used to simulate multi12x4_Full. The test bench here is used in the same way as it was used to simulate line_x. Again all inputs are held in in doubling increments - i.e, the stim_proc1 holds A1 for 10ns, stim_proc2 holds A2 for 20 ns. This continues until stim_proc16, where B4 is held for 327680 ns.

Discussions and Conclusions

Fig.33

Fig.33 is the simulated outcome of the final VHDL file. It shows 3 variables on the left hand side; the multiplicand, the multiplier and the output. The multiplicand correctly displays whole integers from 0 to 4095, which is the maximum decimal number that a 12-bit binary number can represent. The multiplier correctly displays whole integers from 0 to 15, which is the maximum decimal number that a 4-bit binary number can represent. The output correctly displays whole integers from 0 to 61425, which is the maximum decimal number that a 16-bit binary number can represent. It also correctly displays the multiplication of the multiplicand and the multiplier, meaning that this VHDL synthesis and simulation works as designed.

As noted earlier, there was an issue found within line_x in which the expected output jumped in increments of 4, which appears to be a fault within the VHDL synthesis. A diagnosis of this has not been achieved, but there has been no observed effect in the final simulation. In the interest of further reproducibility for this design, diagnosing the source of this fault would improve the overall outcome in a future iteration.

Task 2

Background Information

The aim of Task 2 is to design, simulate and synthesise the Finite State Machine (FSM) described by the state diagram given as both a VHDL and discrete gate solution. A Finite State Machine is a type of sequential circuit that is designed to sequence through specific patterns of finite states in a predetermined sequential manner. There are 3 classifications of FSM worth noting; Mealy (Class A), Moore (Class B), as well as a Moore machine without an output decoder (Class C) [7]. Finite state machines have been widely used to model systems in diverse areas, including sequential circuits, some types of programs (in lexical analysis, pattern matching, etc.), and more recently, communication protocols [8].

Fig.34, given state diagram

Class A / Mealy Machines

A Mealy machine has a finite number of states and produces outputs on state transitions after receiving inputs [8]. It is defined as a machine whose output values are determined by both its current state and current inputs [9].

Fig. 35, a block diagram representation of a basic class a mealy machine

(Class C follows the same design without the output decoder, as previously mentioned)

Class B / Moore Machines

A moore machine has an output that depends only on its present state [9].

Fig. 36 a block diagram representation of a basic class b moore machine

This project will be based on a Class A Mealy machine as the output is made up of the 3 bits - A,B,C and the input X, which goes directly to the output. This task involves a discrete gate solution. However, it is worth noting that using programmable hardware devices such as FPGAs can offer a more efficient process as they contain an array of programmable logic blocks. They also include a hierarchy of reconfigurable interconnects allowing blocks to be wired together, like many logic gates that can be inter-wired in different configurations [10].

Solution to the Task (Procedure / Results)

Step 1 - Firstly, all 8 states (A-H) from the state diagram were represented in a table with respect to the transition input. Each start state is represented by three bits, A, B and C. The transition input is represented by X.

Step 2 - The next state is a result of the start state due to the effect of the transition input. All 8 states have two next states depending on the transition input. Two of the next state transitions result in an output Z of logic 1. This occurs when start state A transitions to B due to X being 1 and when E transitions to D when X is 1.

Step 3 - Next, this table is converted via D type flip flop into a Karnaugh map where the aim is to produce three sum of products D_a , D_b and D_c .

D_a refers to the start state bits AB, CX - the result of which are in the A column of the next state.

D_b the same but for the B column of the next state.

D_c the same but for the C column of the next state.

Step 4 - Once the Karnaugh maps are filled in, all the 1's are looped. This is known as minimisation. Loops should be in factors of 16, starting with the largest loops first.

Fig. 37,

Step 5 -Each loop is described by the fixed bits that it includes. These bits do not change in the loop.

The yellow loop shows that when C and X are 0 and 0, regardless of A and B the output is 1. Therefore the yellow loop is not C AND not X, thus $\neg C \neg X$

The blue loop shows that when A, B and C are 0, 1 and 1 respectively the output is 1. Therefore the blue loop is $\neg A.B.C$

The green loop shows that when A, C and X are 1, 1 and 0 respectively the output is 1. Therefore the green loop is $A.C.\neg X$

This continues for each Karnaugh map.

Step 6 - The loops of the Karnaugh map are all combined to give a final representation known as Sum of Products, $D_a = \overline{C}/X + \overline{A}.B.C + A.C./X$

$D_b = C./X + \overline{A}.B./C + A.B./X + \overline{A}/B.C + A./B./X$

$D_c = \overline{C}.X + \overline{A}.B.X + \overline{A}/B.C./X$

Fig.38

Step 8 - Once the sum of products were defined, a discrete gate solution was designed. The design of the circuit is based upon the outputs of the Karnaugh maps. First, three DTFF (D-Type-Flip-Flop) components were inserted, one for each Karnaugh map output (D_a , D_b and D_c). The DTFFs have 2 inputs and 2 outputs. The first input comes from the respective Karnaugh map output. The second input is from the systems clock (CLK). The two outputs will be either logic 0 or 1 depending on the inputs.

Step 9- An AND gate for every loop in the Karnaugh map was inserted. The number of fixed bits in the loop affects the number of inputs for each AND gate.

Step 10- Three OR gates were inserted - one for each separate Karnaugh map output. The number of inputs to the OR gates depends on the number of loops in that respective Karnaugh map.

Step 11- Inputs and outputs were labelled and the gates were connected.

Step 12- As well as designing a discrete gate solution the Z output was also designed. The Z output only became a logic level 1 in two instants. When state A received the X input 1, resulting the next state being B. Also when state E received the X input 1, resulting the next state being D. Both of these instants can also be represented as $\overline{A}/B./C.X$ OR $A./B./C.X$, respectively.

Fig. 39

Fig. 40 shows the VHDL script written for the finite state machine. It first defines the state types which are state_A to state_H and then declares the present_state and next_state as signals.

Next, two 'if' statements are declared. These acknowledge the two potential logic levels of X, 0 and 1. Each present_state has a predetermined next_state depending on the value of X. all next_states and corresponding Z outputs are defined here.

As well as defining the states of the FSM, this also shows the clocks process (CLK).

Fig. 40

Fig. 41 shows the stimulus processes of the test bench. Stim_proc1 causes the CLK to oscillate every 10ns. Stim_proc2 was then used to test if the VHDL for this FSM worked correctly. It shows the input X being forced to a chosen logic level; either 0 or 1. This causes the FSM to cycle through different states. The green writing within Stim_proc2 shows the predicted next_states.

Fig.41

Conclusions

Fig. 42

Fig. 42 shows the simulation of the VHDL synthesis. On the left hand side of the simulation, there are four variables: X and CLK, which are the inputs, and Y and Z, which are the outputs. Y is only involved to show the changing of states. The sequence that X goes through causes the state to change to its desired state as predicted. Z also gives the correct outputs and the simulation therefore works as expected. The finite state machine as described by the given diagram has been successfully synthesised. As previously mentioned, although this design has fulfilled the aim of the task, the overall process could be made more efficient through the use of programmable hardware as components such as logic blocks could offer different configurations that could exceed the potential of logic gates.

References

- [1] Biradar, V. and Vishwas, P., 2017. *Design and performance analysis of modified unsigned braun and signed Baugh-Wooley multiplier*. [online] Ieeexplore.ieee.org. Available at:
<https://ieeexplore.ieee.org/abstract/document/8284612?casa_token=XrU-DcQKKwMAAAAA:Nh0gIGcrxIXcGYDZQwSu4ZUUvVaKQ3jEkCd_OMMUuDVZEih1EBj_sURcAA6z_cfU1zLdMPgB_>
- [2] Ms. Madhu Thakur, P., 2012. *Design of Braun Multiplier with Kogge Stone Adder & It's Implementation on FPGA*. [online] Ijser.org. Available at:
<<https://www.ijser.org/paper/Design-of-Braun-Multiplier-with-Kogge-Stone-Adder-Its-Implementation-on-FPGA.html#:~:text=Braun%20multiplier%20is%20a%20type,and%20one%20Ripple%20Carry%20Adder>>
- [3] WhatIs.com. 2020. *What is a logic gate (AND, OR, XOR, NOT, NAND, NOR and XNOR)? A definition from WhatIs.com*. [online] Available at:
<<https://whatis.techtarget.com/definition/logic-gate-AND-OR-XOR-NOT-NAND-NOR-and-XNOR>>
- [4] Electronics Tutorials. 2021. [online] Available at:
<https://www.electronics-tutorials.ws/combinational/comb_7.html>

[5] Mendu, N., 2021. *A Simple Method to Improve the throughput of A Multiplier*. [online] Ripublication.com. Available at: <https://www.ripublication.com/irph/ijece/ijecev6n1_02.pdf>

[6] ROY, S., 2021. *Unsigned Array Multiplier - Digital System Design*. [online] Digital System Design. Available at: <<https://digitalsystemdesign.in/unsigned-array-multiplier/>>

[7] Electronics Club. 2021. *Finite State Machine (FSM)| classification, application*. [online] Available at: <<https://electronics-club.com/finite-state-machine-fsm-classification-application/>>

[8] . Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," in *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090-1123, Aug. 1996, doi: 10.1109/5.533956.

[9] GeeksforGeeks. 2021. *Difference between Mealy machine and Moore machine - GeeksforGeeks*. [online] Available at: <<https://www.geeksforgeeks.org/difference-between-mealy-machine-and-moore-machine/>>

[10] "A Survey of FPGA-based Accelerators for Convolutional Neural Networks", S. Mittal, NCAA, 2018