Emerging Nanoscaled
Integrated Circuits and Systems Labs

אוניברסיטת בר-אילן
Bar-Ilan University

Posit Multiplier Using Dual Mode Logic

Dan Shaul 213744287

Guy Gonen 305698599

Instructor: Mrs. Inbal Stanger, Mr. Netanel Shavit

Academic Supervisor: Prof. Alexander Fish

Table of contents:

1. Abstract 3
2. Motivation 4
3. Theoretical Background 5
   3.1. Binary Numbers Representations 5
      3.1.1. Floating Point 5
      3.1.2. Posit 6
   3.2. Multipliers 11
      3.2.1. Array Multipliers 12
      3.2.2. Booth Multipliers 13
   3.3. Dual Mode Logic 15
4. Alternatives Solutions 18
   4.1. Radix-4 8X8 Booth multiplier 18
   4.2. Posit Multiplier 20
5. Our Solution 23
6. Circuit Implementation 24
7. Results 27
8. Conclusions and End 30
9. Ideas for Future Work and Improvements 32
10. Acknowledgments 33
11. References 34
12. Appendix 35

1. Abstract:

Our project is to build a binary multiplier using new way of binary numbers representation - posit. One of the advantages that the posit has over other floating-point representations is that it doesn't have a fixed size bit-width, or in another words it is dynamic.

Multiplication is one the most frequent arithmetics operations used in now a days computers hardware, and also one of that have effect on energy consumption and speed.

In our project we will implement a binary multiplier that uses the dynamic nature of the posit numbers with emphasis on the mantissa multiplier.

We will build this mantissa multiplier by using Dual Mode Logic (DML) which is a power efficient and high-speed integrated circuit (IC) design

2. Motivation:

In today's life, everything around us is based on some sort of computational nature. We have smart phones, smart TVs, and even smart cars. We always want that our devices will work fast and last longer before we will be ought to charge them.

The thing that allows us to achieve that works behind the scenes and called the hardware. The hardware allows us to compute and therefore that is the thing that drives our world.

The hardware works by performing thousands of calculations per second. A significant part of these calculations are multiplications of numbers performed for different processes. These actions are extremely significant because they constitute the "critical path", a path that takes the most time and energy to perform.

In order for the computer to perform this arithmetic operation, it is necessary to translate the numbers from the world we know to the world the computer knows - the binary world. The representation of binary numbers that is most common these days is floating-point representation. There are several problems with this representation method, so in 2017 a new method for representing numbers was proposed, Posit. The advantage of this method is that this method there is no fixed number of bits that need to be allocated for the mantissa each number has in is binary representation.

In this project, we will build a multiplier that will multiply numbers in a posit representation, with an emphasis on the mantissa multiplier, in order to achieve the best results both in terms of power consumption and in terms of system speed, using the dynamic feature of presenting the numbers in posit. We will first build the multiplier using "standard" CMOS technology and then we will build the same multiplier using Dual Mode Logic (DML) in order to

improve the multiplier efficiency by using DML's ability to enjoy two worlds - dynamic and static

3. Theoretical Background:

3.1. Binary Numbers Representations

3.1.1. Floating Numbers:

In our days the most common representation of numbers in the computer world is the floating-point format. In this format each number has 3 parts:

1-bit for the sign, which is the MSB of the number, n bits for the exponent, and (N-n-1) bits for the mantissa. The number n is defined by the precision of the format, for example “single-precision” is using 32 bits for each number and has 8 bits for the exponent. In order to get the true value of the exponent we need to subtract a bias number form the value we get. So, if we stick with our example of 32 bits the bias we need to subtract is 127, because the range of the exponent is [-126,127]. We have two special exponent values, one where all bits are zeros which is used to represent zero, and the other is where all bits are ones which is for infinity. We also need to take into account that we have an implicit 1 that we add to the mantissa in order to be more accurate. The general form of floating-point number is:

$$x = (-1)^{sign} \cdot 2^{exp-bias} \cdot (1.mantissa)$$

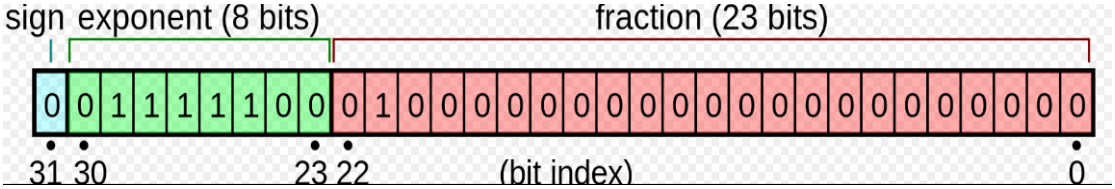


Fig. 1 single precision floating number format

3.1.2. Posit Numbers:

Format:

A generic posit format consists of a mandatory sign, one or multiple regime bits, multiple optional exponent bits, and multiple optional fraction bits [Fig. 1]. The sign bit is **0** for positive numbers and **1** for negative numbers. The number of regime bits is dynamic following a special encoding. After the sign bit, the regime includes a run of 0 or 1, which is terminated by an opposite bit ($\bar{r}$) or at the end of the number format. Similarly, the number of bits for the exponent and fraction is dynamic. A posit number includes the exponent and fraction **only if necessary**.

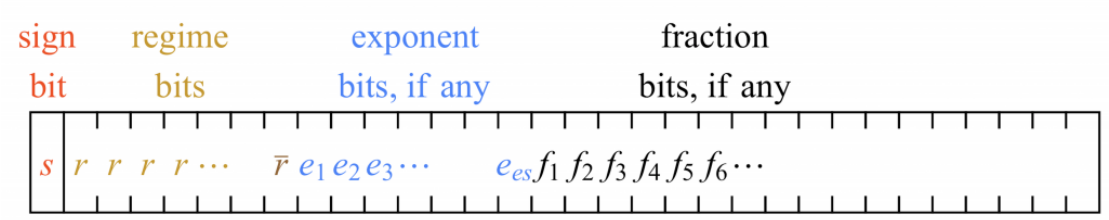


Fig.2 : General posit format for finite, nonzero values-color codes.

To understand how the regime bits represent numbers, consider the binary numbers in [Fig. 2].

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, k	−4	−3	−2	−1	0	1	2	3

Fig. 3 : Decimal values of regime bits (“x” means don’t care).

Let *m* be the number of identical bits in the regime bits (amber color). If the first bit is zero, the number of zeros (m) represents a negative value (-m). Otherwise, the number of ones minus one (m-1) represents a positive value (m-1). The regime bits realize a scale factor of $useed^k$, where $useed = 2^{2^{es}}$.

Exponent *e* (blue bits) is regarded as an unsigned integer to realize another scale 2^e. Unlike IEEE 754, posit does not use bias for the exponent. Each exponent may be up to a predefined number of bits (*es*). The remaining bits after the regime and the exponent are used for the fraction (f). Like IEEE 754, the fraction includes a hidden bit, which is always **1** as posit does not have any denormal number. Overall, an n-bit posit number (p) can represent the following numbers.

$$\begin{cases} 0, & p = 0, \\ \pm\infty, & p = -2^{n-1}, \\ sign(p) \times useed^k \times 2^e \times f, & \text{all other } p. \end{cases}$$

For instance, Fig.4 represents 477/134217728 ≈ 3.55393 × 10^{-6} with es=3.

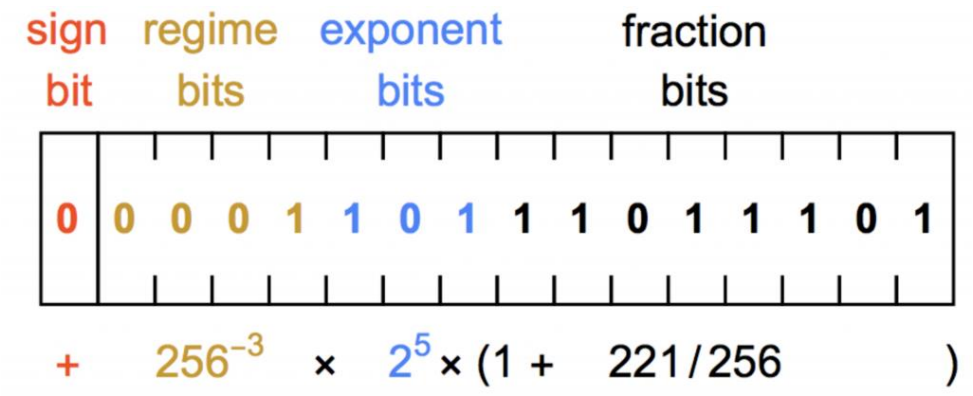


Fig. 4: Example of a posit number and its decimal value

Number Construction:

Fig. 5 shows values for a 3-bit posit format with *n* = *3* and *es* = *1*. There are only two reserve representations: 0 (all 0 bits) and ±∞ (1 followed by all 0 bits). A total of 8 values may be represented using 3 bits.

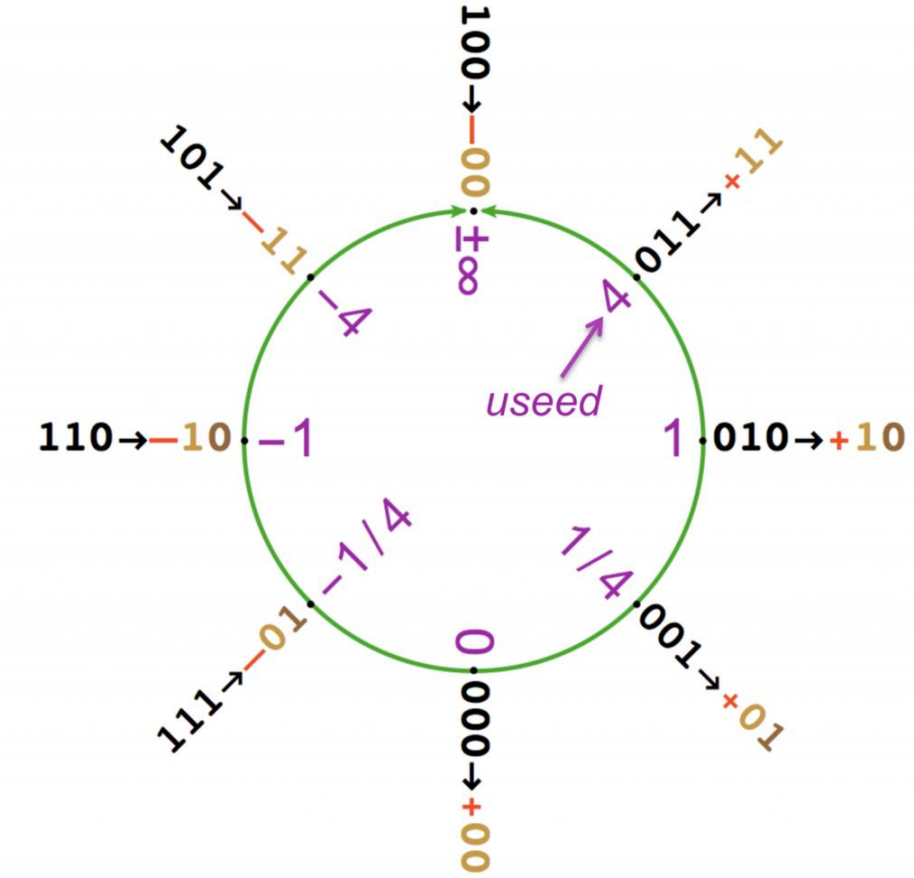


Fig. 5: Values for a 3-bit posit

Similar to the floating-point numbers, appending 0 to a number does not change its value, whereas appending a 1 results in a new value between two existing numbers on the ring [Fig. 6, and 7]. The new posit value may be between the *maxvalue* (*useed*) and ±∞, the new value is *maxvalue* × *useed,* (2) between the existing values $x = 2^m$ and $y = 2^n$, where |m-n|> 1, the new value is xy (add a new **exponent bit**), or (3) between the existing x and y values next to it, which is $\frac{x+y}{2}$ (add a new **fraction bit**).

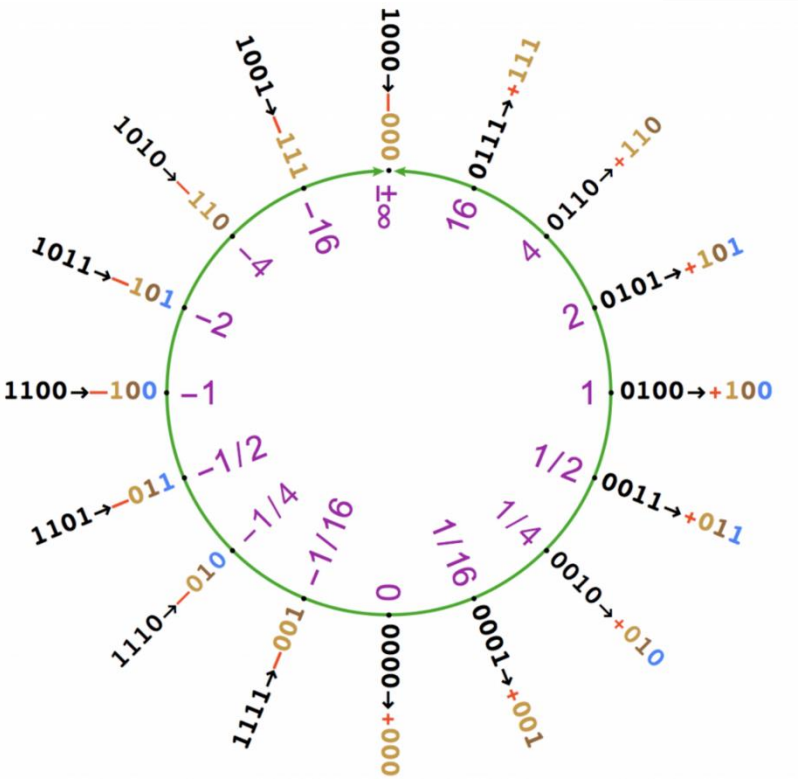


Fig. 6: Values for a 4-bit posit

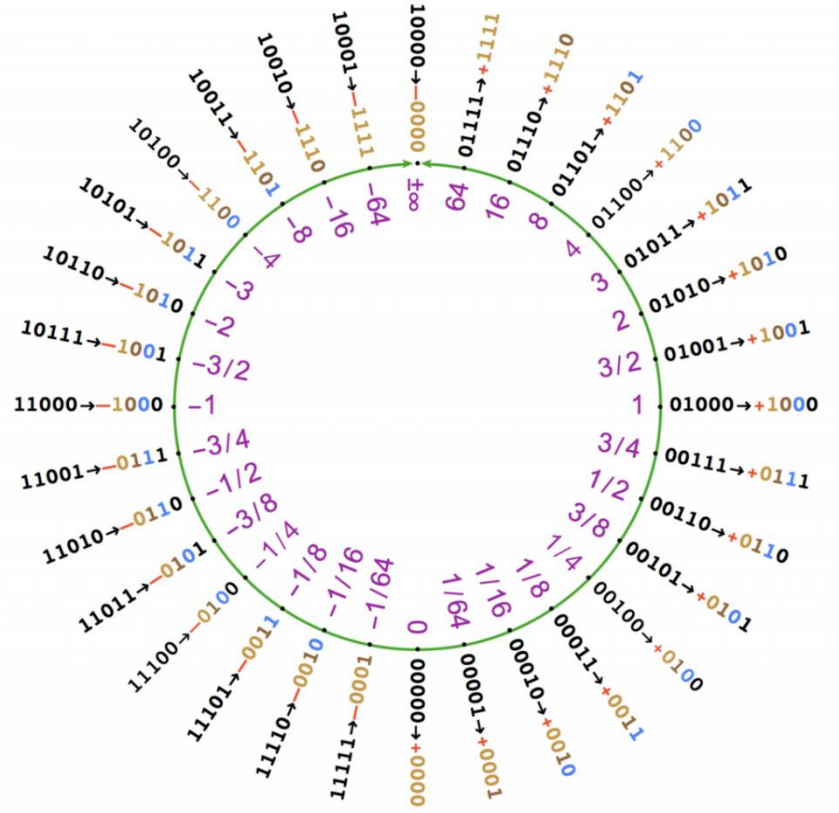


Fig. 7: Values of a 5-bit posit

Posit vs. IEEE 754 standard:

Unique Value Representation. In the posit format, f(a) is always equal to f(b) if a and b are equal, where f is a function. In the IEEE 754, the reciprocals of positive and negative zeros are +∞, −∞, respectively. Moreover, the negative zero equals positive zero. This implies +∞ = -∞ which is not true. In a floating-point comparison (*a* == *b*), the result is always false if either *a* or *b* is NaN. This even holds if *a* and *b* have the same bit representation. In posits, however, *a* and *b* are equal if they use the same bit patterns; otherwise, they are not equal. Moreover, the result of an arithmetic operation would be the same over different hardware systems. For instance, in the case of the *Q.W* example at the beginning, posit needs only 24 bits to generate the correct result.

No Gradual Underflow. IEEE 754 faces an underflow problem when the exact result of an operation is nonzero but smaller than the smallest normalized number. This problem is alleviated by rounding values. However, this may result in a denormal number. As a result, some fraction digits are transferred to the exponent for representing smaller numbers. This is known as gradual underflow. Handling gradual underflow is complicated and is supported in software by some IEEE compliant microprocessors. The posit number system does not encounter this problem due to supporting a tapered precision, where numbers with small exponents are represented more accurately than the numbers with large magnitude exponents.
Holding Algebraic Rules Across Formats. Unlike IEEE 754, posit holds the associativity of addition. Moreover, computing values across multiple posit formats with different sizes is guaranteed to produce the same value.
Exception Handling. While there are 14 representations of Nans in IEEE 754, there is no “NaN” in posits. Moreover, posit has single representations for 0 and ∞. Overall, it makes the computation with the posit numbers simpler than IEEE 754. In the case of exceptions (e.g., division by zero), the interrupt handler is expected to report the error and its cause to the application.[1],[2],[5],[7].

3.2. Multipliers:

Multipliers are one of the most important hardware units and have extensive use in a variety of fields, both digital and analog.
Due to the continuous use of these units, and the fact that many of the critical paths in each processing unit go through them, a situation has arisen where an improvement in the multiplier performance will lead to an overall improvement in performance - in terms of energy consumption and speed of operation.
At the same time, multipliers are more complicated units to design than other arithmetic circuits like adders. The multiplier works in such a way that it creates partial products , by using logical gates, and then uses basic arithmetic units, in order to reach the final product.
In each multiplier there are 3 steps: creating partial products, addition of the partial products, a final scheme and reaching a result.
Therefore the ambition is to create an algorithm that will give us the option to reach as few partial products as possible.
We will first present the basic structure of a multiplier and then we will present a different algorithm for the required improvement that we used.

3.2.1. Array Multipliers:

The simplest form of multiplier is array multiplier. It uses half adders (HA), full adders (FA) and AND gates depending upon the multiplicand and multiplier bits. The simplest way to perform multiplication is to use single two input adders. Each bit of multiplier is AND-ed with multiplicand bit to produce partial products and each time partial products are left shifted. The main drawback of array multiplier is that it has large latency as it takes more clock cycles to compute the output. The number of partial products to be added is determined by the number of bits that the multiplicand and multiplier have used. In order to get fast performance, one of the methods is to arrange the circuit to generate all the partial products in parallel and organize in an array.[8]

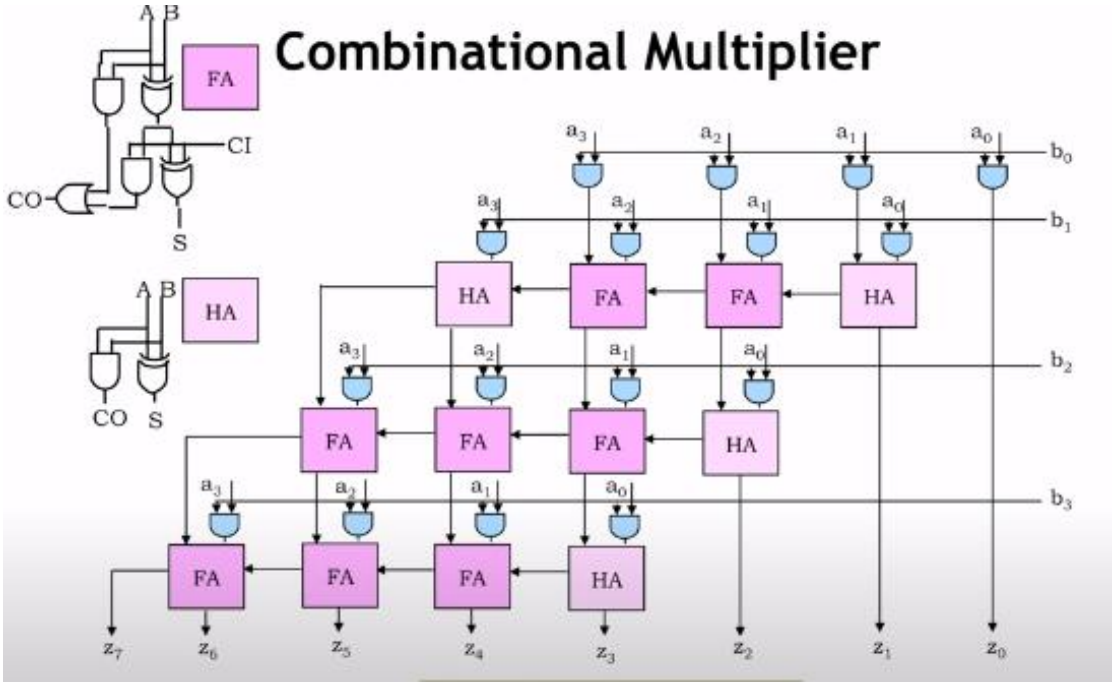


Fig 8. Array multiplier structure

3.2.2. Booth Multipliers:

Another widely used algorithm which leads to faster performance is booth algorithm. Booth multiplication algorithm is fast compared to lot of other algorithms. It was invented by Andrew Donald in 1950 and the main importance of booth algorithm is that it can multiply to signed binary numbers. Various encoding styles are available depending upon number of bits in the group such as radix-2, radix-4, etc. .

3.2.2.1. Radix-2 Booth algorithm

In radix-2 algorithm, zero is appended to right most side to the multiplier bits and group the multiplier in such a way that each group consists of 2 bits. So that the first pair consists of appended zero and least significant bit (LSB) of 6 multiplier and the next pair is the overlapping of the first pair in which most significant bit (MSB) of the first pair will be the LSB of the second pair. So that for n-bit * n-bit multiplication, n partial products are obtained. Radix-2 booth algorithm produces same number of partial products as array multiplier so number of cycles to compute the result is almost similar.[2]

3.2.2.2. Radix-4 Booth algorithm

To further decrease the number of partial products, algorithms with higher radix value are used . In radix-4 algorithm grouping of multiplier bits is done in such a way that each group consists of 3 bits as mentioned in table 1. Similarly the next pair is the overlapping of the first pair in which MSB of the first pair will be the LSB of the second pair and other two bits. Number of groups formed is dependent on number of multiplier bits. By applying this algorithm, the number of partial product rows to be accumulated is reduced from n in radix-2 algorithm to n/2 in radix-4 algorithm . The grouping of multiplier bits for 8-bit of multiplication is shown in figure 9.

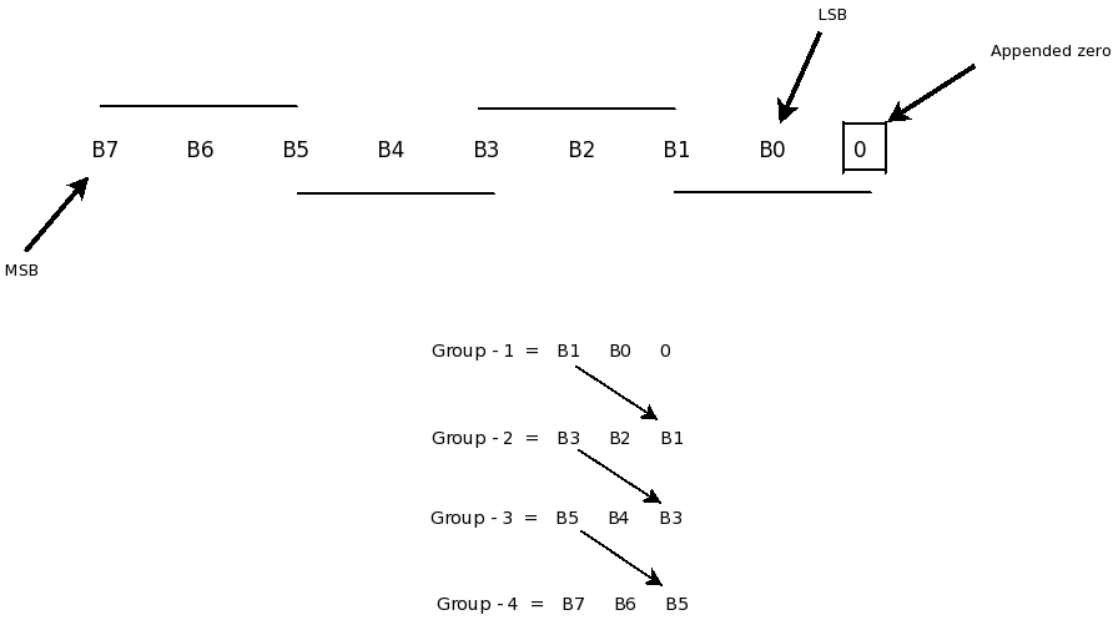


Fig 9. Grouping of multiplier bits in Radix-4 Booth algorithm

3.3. Dual Mode Logic:

While most of today's efficient solutions are proposed at the system and architecture levels, a new solution called dual mode logic (DML) is proposed to provide an efficient solution at the gate level.

The DML logic gates family was proposed in order to provide a very high level of energy-delay optimization flexibility. DML allows an on-the-fly change between two operational modes at the gate level: static mode and dynamic mode.

In the static mode, DML gates consume very low energy, with some performance degradation, as compared to standard CMOS gates. Alternatively, dynamic DML gates operation obtains very high performance at the expense of increased energy dissipation.

DML Basics:

A basic DML gate architecture is composed of an un-clocked static gate, *e.g.* CMOS, and an additional transistor, whose gate is connected to a global clock signal. we focus on DML gates where the static gate implementation is based on conventional CMOS.
A DML gate implementation can be one of two: ``*Type A''* and ``*Type B''*, as shown in Fig 10.

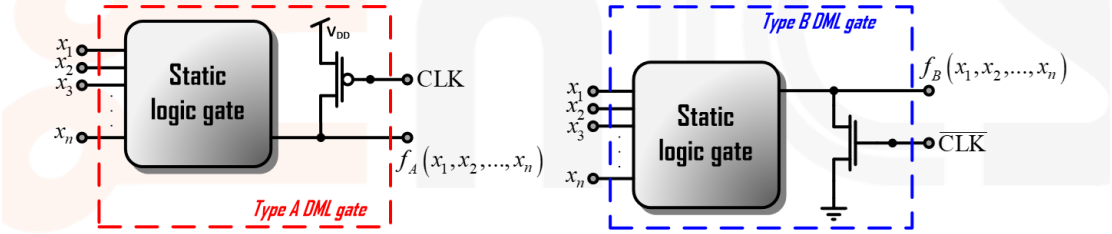


Fig. 10 type A and type B DML gates

In the static DML mode of operation (Static mode), the CLK transistor is cut-off by applying the high *Clk* signal for ``*Type A''* and low *Clk_bar* for ``*Type B''* topology. Therefore, the gates of both topologies operate similarly to the static logic gate, CMOS in this case.

For a dynamic operation of the gate (Dynamic mode), the *Clk* is enabled for toggling, providing two separate phases: pre-charge and evaluation.

During the pre-charge phase, the output is charged to *VDD* in ``*Type A''* gates and discharged to *GND* in ``*Type B''* gates. During evaluation, the output is evaluated according to the values at the gate inputs.

It was shown that DML gates have presented a very robust operation in both static and dynamic modes under process variations and at low supply voltages. Dynamic mode robustness is mainly achieved by the intrinsic active restorer (pull-up in ``*Type A''* and pull-down in ``*Type B''*).

This restorer also allows sustaining glitches, charge leakage and charge sharing. Unique sizing of the DML gate transistors is the key factor for achieving low energy consumption in the static DML mode (in which the topology of the gate is identical to the static gate).

This sizing is also responsible for reduction of all capacitances of the gate. In a similar way, the unique transistor sizing enables evaluation through a low resistive network achieving fast operation in the dynamic mode.

Energy efficiency is achieved in the static DML mode at the expense of slower operation (Low Energy and Low Performance, left scales). However, the

dynamic mode is characterized by high performance, albeit with increased energy consumption (High Energy and High Performance, right scales). These tradeoffs allow a very high level of flexibility at the system level.[3],[6]

4. Alternatives Solutions:

During our research we saw various ways about how to implement a multiplier. We will represent here, in a nutshell, two of them: one that uses booth radix-4 algorithm in the implementation, and the other that uses both posit format and booth radix-4 algorithm.

4.1. Radix-4 8X8 Booth multiplier:

For 8x8 multiplication, booth multiplier uses four stages to compute the final product. An 8x8 booth multiplier uses same type of component in all the four stages shown in figure 10. In each stage it uses components like booth encoder, 16-bit adder/subtractor block and 16-bit 3:1 MUX. The multiplier bits with appended is zero is applied to booth encoder of each block. The input to booth encoder of stage-1, stage-2, stage-3 and stage-4 are B1-B0-0, B3-B2-B1, B5-B4-B3 and B7-B6-B5 respectively and generate output depending upon inputs. The operation of all the four stages is similar. The MSB of the output of booth encoder is applied to adder/subtractor block which will select adder if the input to block is 0 and select subtractor if the input to block is 1. The two LSB of the output of booth encoder is applied to multiplexer block which will select either 0, ‘A’ or ‘2A’ depending upon the structure of 3:1 MUX. If the input to multiplexer block is 00 it will select 0 input, if 01 it will select ‘A’ and if 10 it will select ‘2A’ as an output.

After selection of selection of which input to select i.e. 0, ‘A’ or ‘2A’, the output of MUX is provided to the adder/subtractor block. The other input to adder/subtractor block is the output from previous block. For the first stage, the inputs to the adder/subtractor block are zero and output from multiplexer.

We didn’t implement this multiplier the way presented in [4] because it was wrong it lacks the Left Shifters so we changed it.

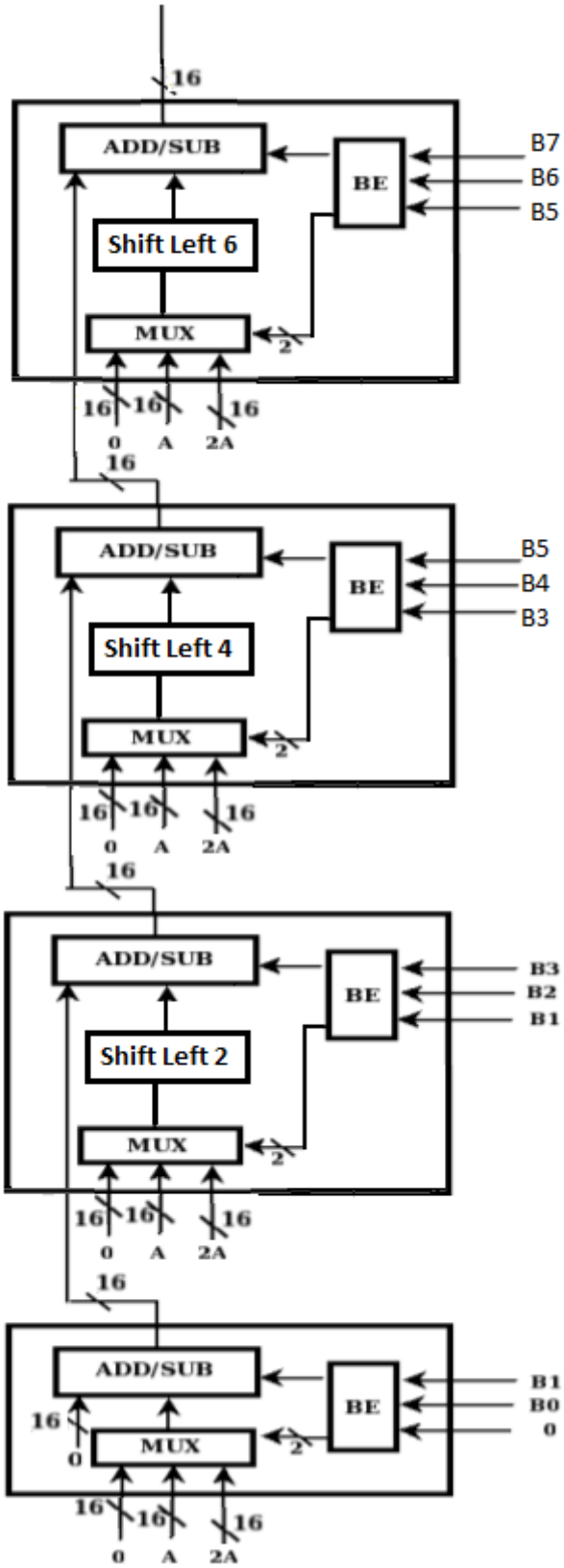


Fig. 11: 8X8 radix-4 Booth multiplier block

4.2. Posit Multiplier:

the formal way of multiplying two numbers in posit format works in this way:

1. We divide the number represented in post floating point to 4 parts , the mantissa, exponent, reg ,and 1 bit for sign
2. We multiply the mantissas a regular multiplication
3. We take the bit in the place n + m (assuming mantissa sized are n and m), then we insert it as a carry into the exponent Adder
4. We take the carryout of the exponent Adder, and we insert it to the regs adder
5. In the end we take all the parts we got, and we insert it to the encoder that takes the parts and forms the posit number [7].

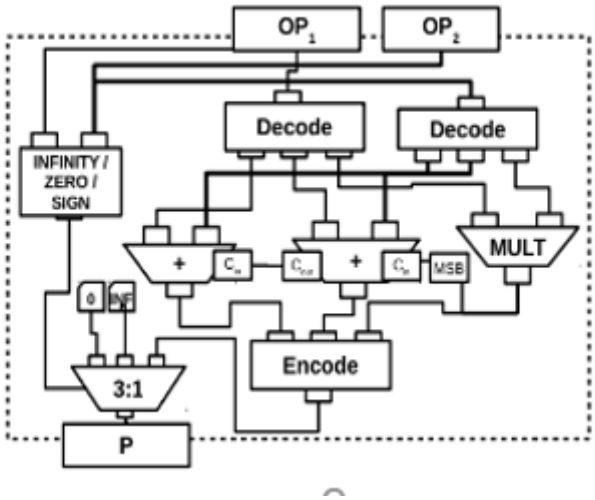
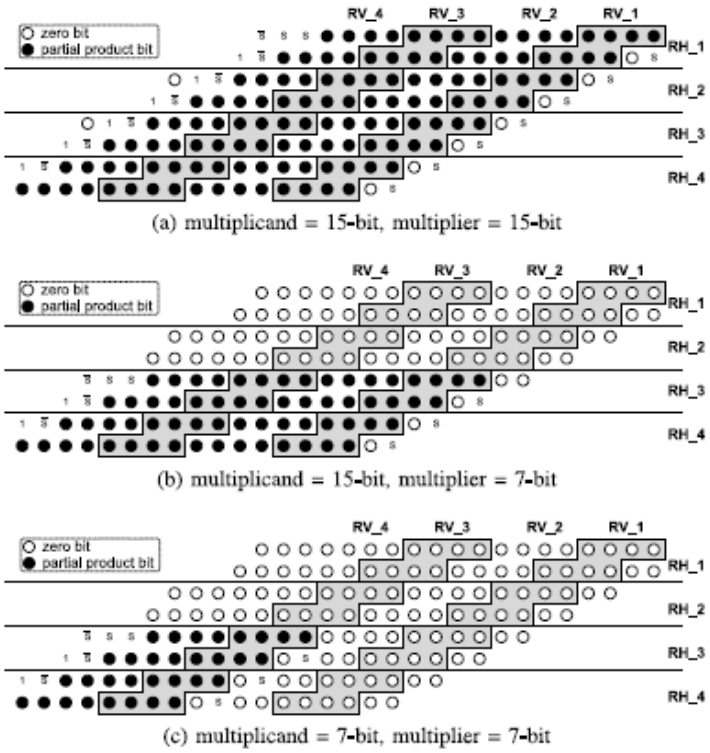


Fig. 12 Posit multiplier

For Posit(16, 1) multiplier, a 15-bit mantissa multiplier is used. When using radix-4 Booth multiplication algorithm ,the partial product array is shown in Fig. 13(a). There is a total of 8 partial products and each partial product is 16-bit. In the proposed design, the 15-bit multiplier is divided into 4 groups: the most significant 3-bit is one group, and the remaining 12-bit are divided

into three 4-bit groups. Correspondingly, the 8 partial products are divided into

4 groups, RH_1, RH_2, RH_3, and RH_4, as shown in. If the multiplier is less than 3-bit, then only the two partial products in RH_4 a regenerated while all others are set to zeros. If the multiplier is more than 3-bit but less than 7-bit, then partial products in RH_3 and RH_4 are generated. If the multiplier is more than 7-bit but less than 11-bit, then partial products in RH_2,RH_3, and RH_4 are generated. Finally, if the multiplier is more than 11-bit, then all the partial products are generated. Similarly, the 15-bit multiplicand is also divided into 4 groups. Each partial product is correspondingly divided into 4 groups,RV_1,RV_2,RV_3, and RV_4, as shown in .If the multiplicand is less than 3-bit, then only RV_4 is generated while all others are set to zeros. If the multiplicand is more than 3-bit but less than 7-bit, the RV_3 and RV_4 a regenerated. If the multiplicand is more than 7-bit but less than11-bit, then RV_2,RV_3, and RV_4 are generated. Finally, if the multiplicand is more than 11-bit, then all bits in the partial product are generated.[2]



(a) multiplicand = 15-bit, multiplier = 15-bit

(b) multiplicand = 15-bit, multiplier = 7-bit

(c) multiplicand = 7-bit, multiplier = 7-bit

Fig. 13 the proposed mantissa multiplier

5. Our Solution:

In our project we implement a mantissa multiplier that multiplies 2 16 bit binary numbers (The mantissas)

But we multiply in radix-4 Booth algorithm so according to paper [2] is said to save us energy and calculation time using another signal , Ctrl 2bit, that is generated from the 2 posit numbers we multiply
We tweaked the using of the ctrl signal such that it is the same as explained above but instead, distributing RH_1 RH_2 RH_3 RH_4 like this:
RH_1: 0:2 bits
RH_2: 3:6 bits
RH_3: 7:10 bits
RH_4: 11:15 bits
This does the same as implementing it the way explained above, and it suits our circuit better.

The whole radix-4 Booth algorithm multiplier we implement according to paper [4] with slight changes.

We choose this method in order achieve as much improvement as possible in the multiplier unit, so this unit will operate much faster and with lower power consumption. We took the proposed solutions as mentioned in previous sections and implemented them with the use of the posit format.

Another thing we did, is to use DML in order to do the calculation much faster, while using the fact that this gate-level solution works in very high-speed and with high performance.

6. Circuit Implementation:

Step1, Booth Encoder:

For Radix-4 Booth Algorithm we need to first make Booth Encoder that takes 3 bits from the multiplier and outputs 3 bits that decide what type of multiplicand to add/subtract to our output according to Booths Table In table1.

When implementing the Booth Encoder we noticed that the implementation in [4] is wrong and changed it a bit.
We implement the Booth Encoder , as seen in fig.14.
In DML we implement it as seen in Fig.14.1.
The NAND is better implemented in type B so we made the circuit around that.

Step2: Add/Sub 32bit Block:

First we started by implementing a 1 bit Adder as seen in fig.15.
From that we implemented the 1 bit Adder/Subtracter from [4] and in fig.17.
We need a 32 bit Adder/Subtracter so first we implemented a 16bit Adder/Subtracter using Ripple Carry Adder Method (fig.18) then the same way for 32 bit using the 16 bit Adder/Subtracter.
We decided to implement the 1 bit adder not the way it was implemented in [4] but using Mirror Full Adder, because they work better in DML , mostly because the overuse of XORs is bad for us since XOR changes output easily by a change of input , and as we will see our circuit have problems with that.
We can see the DML implementation of this mirror full adder in Fig.16.1 and Fig.16.2
When implementing this circuit using mirror full adders in DML logic , we used all of them in Type A or all of them in Type B since a single mirror full adder

produces inverted output Cout , so we add an inverter in its output thus allowing us to chain all the full adders in the same type.

Step3: Mux 4:1 32bit Block:

First, we implemented a 2:1 , 1 bit MUX according to [4] and as seen in fig.19, using this MUX we Implemented a 2:1, 32 bit MUX, in implementing this 32bit MUX we used the same method as making a 2:1, 2bit MUX using 2:1 2bit MUX as seen in fig.20, then we made a 4:1 32 bit MUX as seen in fig.21.
The DML logic MUX we implemented as seen in fig.19.1 , the types we used are the same as in the figure but we used footed and headed when needed.

Step4: 32bit Shifters

For implementing the shifters, we used a simple Shift Implementation using MUX's as shown in fig.22
First , we implemented a 32bit Shifter that shifts 1 bit left, then we used it consecutively to get bigger shifts

Step5: Combining All of them to 1 circuit

According to [4] with little changes to the circuit proposed there to get the circuit in fig.23 .For our circuit we used 16bit input ,not 8bit, so we just multiply the circuit in fig.23 4 times and chain the outputs and increase the shift amount to get the full circuit.

The inputs 2A we implemented using a Shift Left 1 and MUX implemented in previous steps.

The Ctrl Signal we added to the circuit using a MUX, we can see the Ctrl signal works almost the same way proposed in [2] since we divide our partial products to 4 groups the same way as suggested in the paper and we insert them to a MUX with Ctrl signal in select input of MUX thus we output the

matching partial product according to the Ctrl signal the same way proposed in [2].

7. Results:

Single Add/Sub 32bit:

(Inputs are the worst inputs for this circuit)

	Calculation Time			Energy Consumption		
Inputs	Cmos	Dml(static)	Dml(Dynamic)	Cmos	Dml(static)	Dml(Dynamic)
…1110, 1000… DML type A	2.237n	21.832% lower	21.752% lower	466.6f	11.273% lower	330.347% higher
…1110, 1000… DML type B	2.237n	44.327% higher	94.993% lower	466.6f	0.642% lower	17.745% higher

The second inputs are to show other inputs that trigger the worst case , since the first one triggered the worst case for dynamic the second triggers the worst case for static.

From these results we want to compare the DML to the CMOS in static and dynamic for a single add/sub 32 block.
we want to see the DML in dynamic being faster and in static has lower energy consumption.

As expected, the results show:
Static mode DML grants 0.642% improvement in Energy Consumption in its worst case.
Dynamic mode DML grants 21.752% improvement in calculations speed in its worst case.

We can see the Add/Sub 32bit is better implemented in DML than in CMOS

The Full Mantissa Multiplier Circuit:

(Inputs A, B, C are just random inputs we entered to measure the results of the circuit)

	Calculation Time			Energy Consumption		
Inputs	Cmos	Dml(static)	Dml(Dynamic)	Cmos	Dml(static)	Dml(Dynamic)
A	2.1501n	29.57% worse	14.55% worse	13.17p	10.55% lower	147.16% higher
B	2.477n	22.63% worse	4.8% worse	13.18p	8.87% lower	175.523% higher

The results show the following (looking at input A):
Dynamic mode DML logic has 14.55% higher calculation time than normal CMOS logic.
Static mode DML logic grants 10.55% improvement in energy consumption than normal CMOS logic.

The DML logic dynamic mode as we can see, doesn't do us any better and the DML logic static mode does reduce energy consumption as expected. The cause to that is the Add/Subs , that's because all the other blocks calculate their value at the same time before the Add/Sub because they get their input the moment we start so we can rule them out of the case and we can even implement them using DML logic on static mode to save energy. The Add/Subs here doesn't work well because we concatenate their sums, in DML logic the calculation of output shouldn't start before all inputs are

stable and ready because if it isn't the case , the block will calculate wrong output before getting the right inputs to get the right outputs and then destroy the precharge before getting the right Data.

In CMOS logic although there is the same problem as DML logic , what differs is the precharge in DML which we also need to discharge , thus wrong outputs make us do more work compared to CMOS logic resulting in much slower calculation.

8. Conclusions And End:

We can see the results aren't as expected , our idea of making the circuit better using DML isn't working the way we implemented the circuit.
The full circuit implemented in DML doesn't improve in performance compared to CMOS, Since the booth encoder, the MUX-es and the shifts all work together at the same time to produce their outputs once the program start the worst path in our circuit revolved around the chained RCA blocks, so we tried optimizing them.
A single RCA block did manage to be better implemented in DML than in CMOS.
That means the problem was the connection of 2 RCA blocks by their sum, the sum for a single RCA block was rising to value 1 bit after another , thus the first bit for the sum raised first and the second raised after a delay.
That is very bad for our circuit, and that's because in this circuits what is causing problems is mainly by different arrival times of signals to gates thus resulting in wrong outputs , for short terms, we call those Wrong Peaks those Peaks mainly appear in the RCA blocks after the first one , since a full adder produces Outputs that change very easily by inputs , we get those wrong peaks , If we were to implement only one Ripple Carry Adder as we saw it would work great since the inputs are received and are stable at the same time allowing us to calculate the right result at the start of the clock.
But since we chain Ripple Carry Adders (RCA) sum output to input of another RCA The problem starts , since 1 RCA generates the sum outputs in different times , the first bit stables and reaches its value before last bit since Carry is rippled through the Full Adders, then the next RCA starts calculating its output for the first bit and thus calculating Carry out to second bit before the Second bit input is stable making second bit outputs wrong and thus we wrongly discharge the precharge state and destroying the efficiency of dynamic mode making it much worse.

To conclude, we weren't able to improve this circuit using DML logic , it does saves energy in static mode but in dynamic mode it doesn't speed up the calculation.

9. Ideas for Future Work and Improvements:

Future work is to try to implement this circuit using registers and making it pipelined where each stage of the pipeline is 1 Add/Sub 32 bit.
This way the worst case clock cycle time is lower than in CMOS logic , thus stages are computed very fast.
A single operation in this method will take more time to calculate then in a CMOS combinatorically method would, but in pipeline we can start calculating the second operation when the first one is in the second stage. In such case were we have a lot of operations , we will receive results every clock cycle which is very fast compared to CMOS.

10. Acknowledgments:

This book presents a year long process which included studies and research. It was accomplished using academic knowledge and skills we have obtained throughout our degree. So, we want to thank all the professors and stuff that helps us to reach this point.

We would like to express our deep gratitude and admiration to Mrs. Inbal Stanger and Mr. Netanel Shavit for the guidance and patience. There is no doubt that without their directions and patience we wouldn't been able to accomplish this project.

11. References :

1. https://www.sigarch.org/posit-a-potential-replacement-for-ieee-754/
2. Design of Power Efficient Posit Multiplier, Hao Zhang and Seok-Bum Ko, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—II: EXPRESS BRIEFS, VOL. 67, NO. 5, MAY 2020
3. Dual Mode Logic Design for Energy Efficiency and High Performance, Itamar Levi and Alexander Fish, Faculty of Engineering, Bar-Ilan University, Ramat-Gan 52900, Israel, May 21, 2013.
4. IMPLEMENTATION OF HIGH SPEED AND LOW POWER RADIX-4 8*8 BOOTH MULTIPLIER IN CMOS 32nm TECHNOLOGY, RISHIT PATEL, B.E., Gujarat Technological University, India, 2014.
5. Efficient Posit Multiply-Accumulate Unit Generator for Deep Learning Applications. Hao Zhang, Jiongrui He and Seok-Bum Ko, Department of Electrical and Computer Engineering, University of Saskatchewan, Saskatoon, Canada.
6. Subthreshold Dual Mode Logic. Asaf Kaizerman, Sagi Fisher, and Alexander Fish. IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 21, NO. 5, MAY 2013.
7. Hardware Implementation of POSITs and Their Application in FPGAs. Artur Podobas and Satoshi Matsuoka, Tokyo Institute of Technology, Tokyo, Japan.
8. https://www.youtube.com/watch?v=0OX-DkYPB3c

12. Appendix:

Tables and pictures we used during our process of implementation

Table 1: Booth Table

B_{i+1}	B_i	B_{i-1}	Operation	Y_{i+1}	Y_i	Y_{i-1}
0	0	0	+0	0	0	0
0	0	1	+A	0	1	0
0	1	0	+A	0	1	0
0	1	1	+2A	0	0	1
1	0	0	-2A	1	0	1
1	0	1	-A	1	1	0
1	1	0	-A	1	1	0
1	1	1	-0	1	0	0

Fig 14. Booths Encoder :

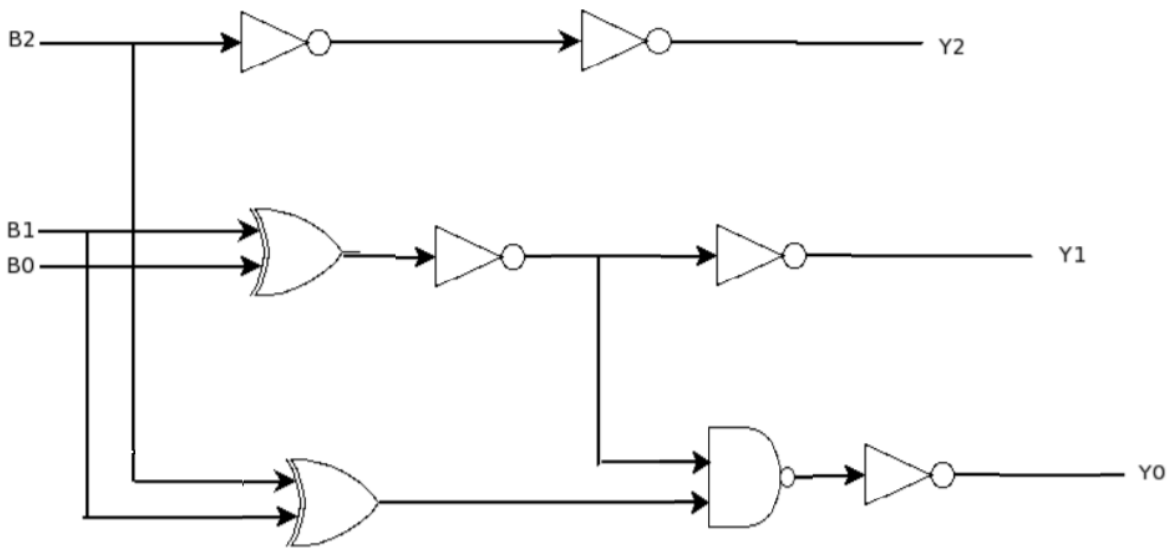


Fig 14.1 The booth encoder in DML:

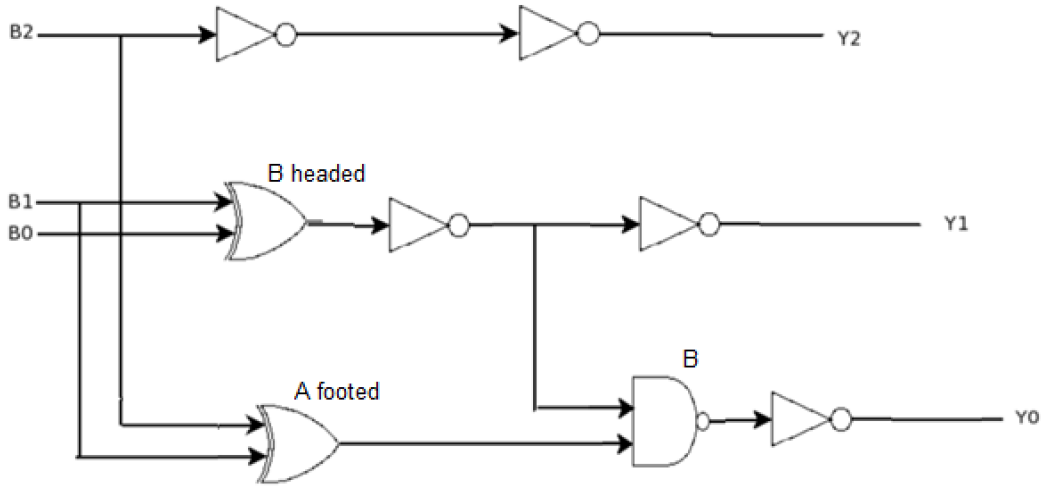


Fig 15. 1 Bit Mirror Full Adder :

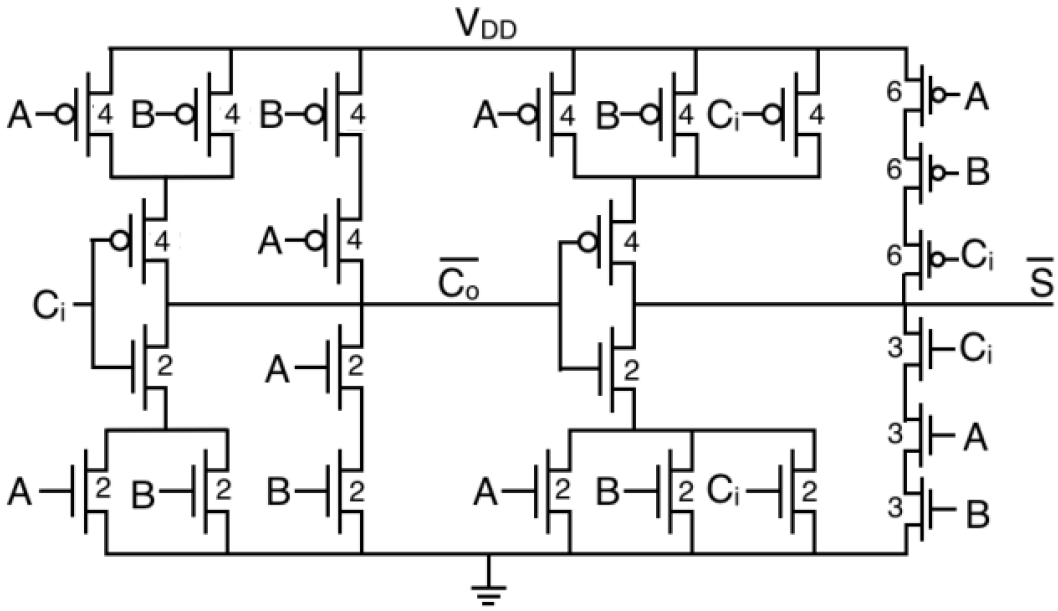


Fig 16.1 1Bit Mirror Full Adder DML type A (The Transistor Without Multiplier Are Sized Minimum):

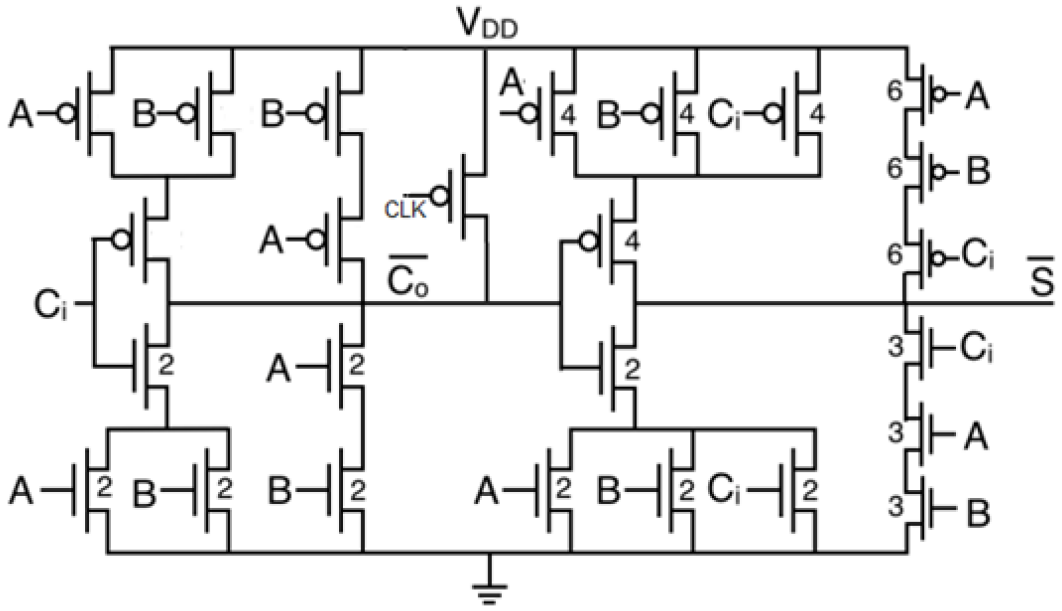


Fig16.2 1Bit Mirror Full Adder DML type B (the transistor without multiplier are sized to minimum):

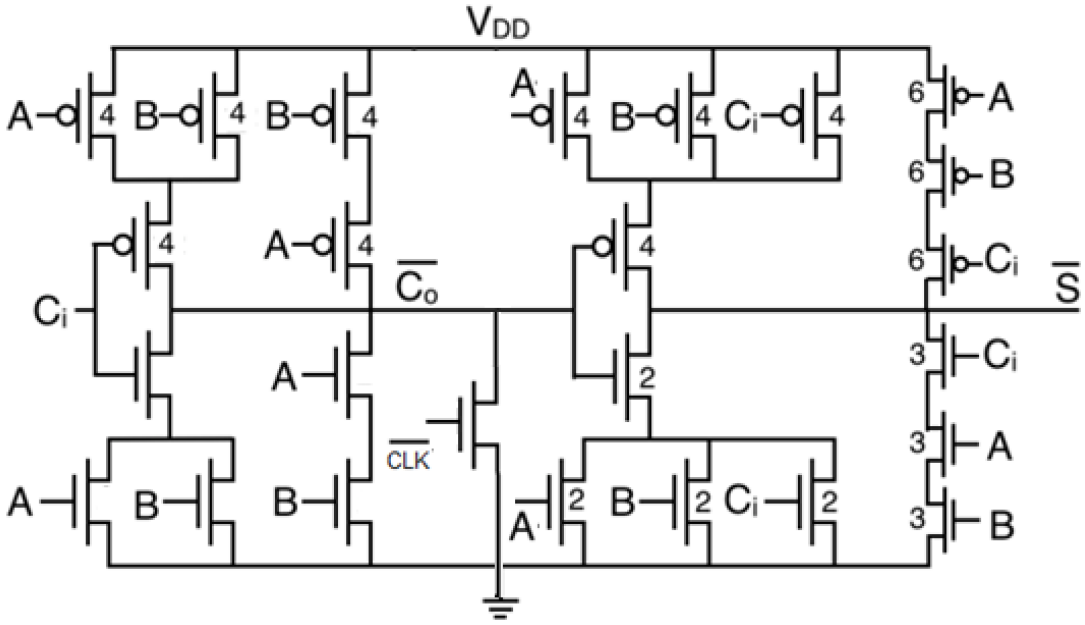


Fig 17. 1 Bit Sub/Add:

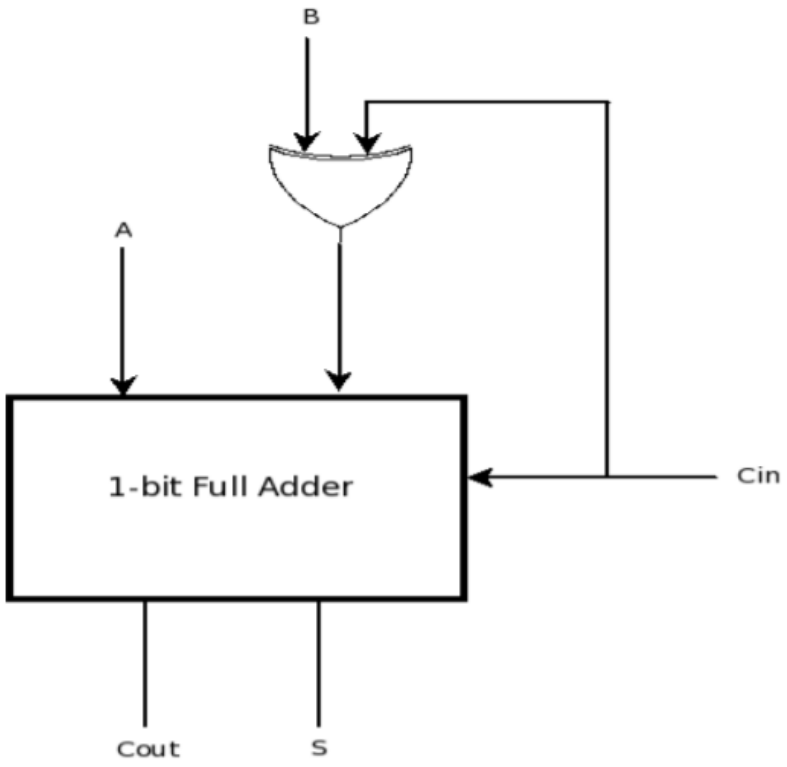


Fig. 18. Ripple Carry Method:

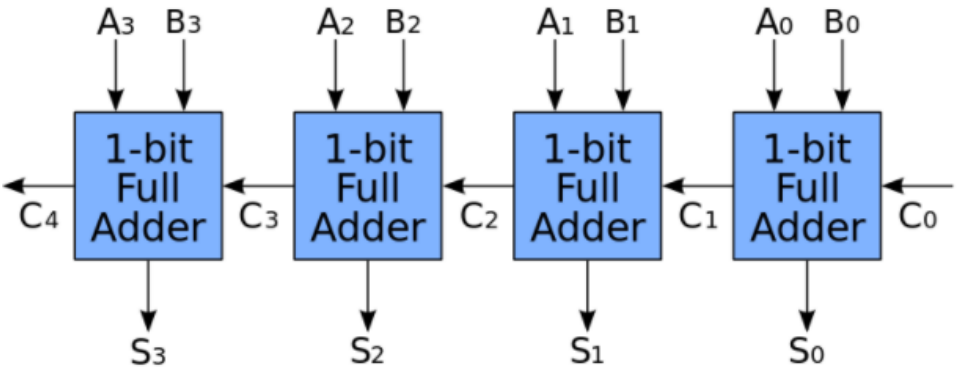


Fig. 19 2:1 1bit MUX:

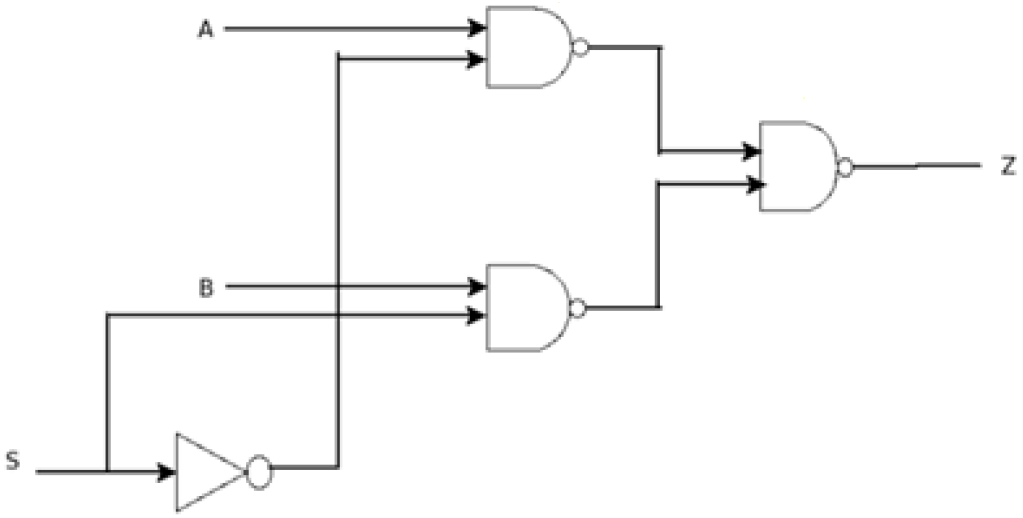


Fig 19.1 2:1 1bit DML MUX:

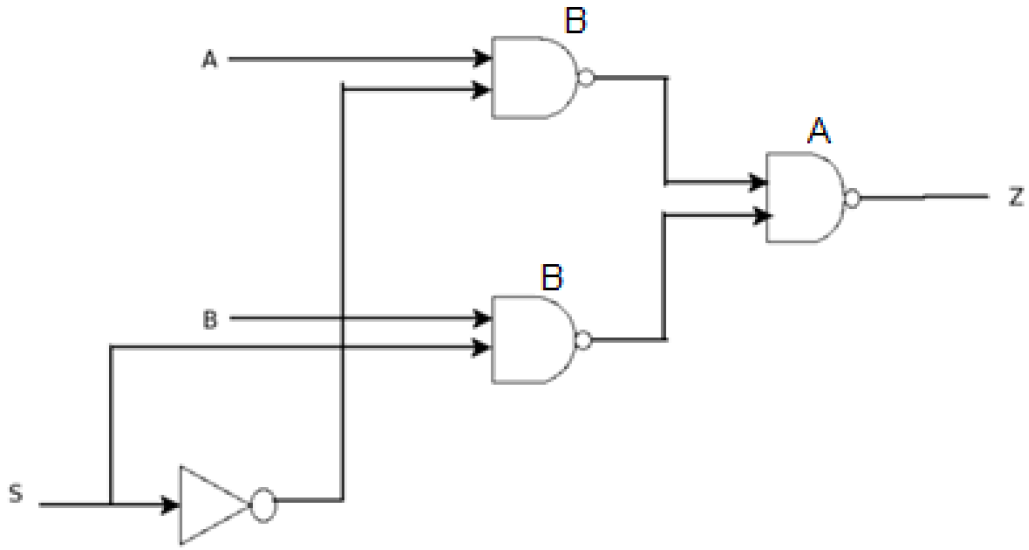


Fig. 20. 2:1 2bit MUX using 1 bit MUX:

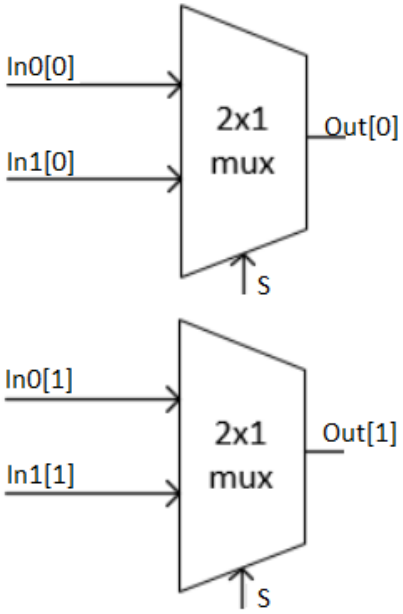


Fig. 21. 4:1bit MUX using 2:1 MUX:

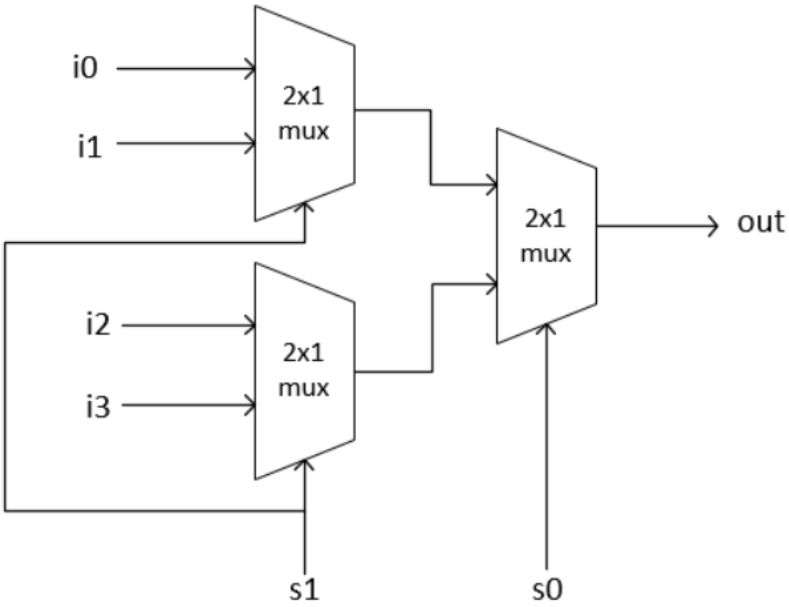


Fig.22. shift left by 1:

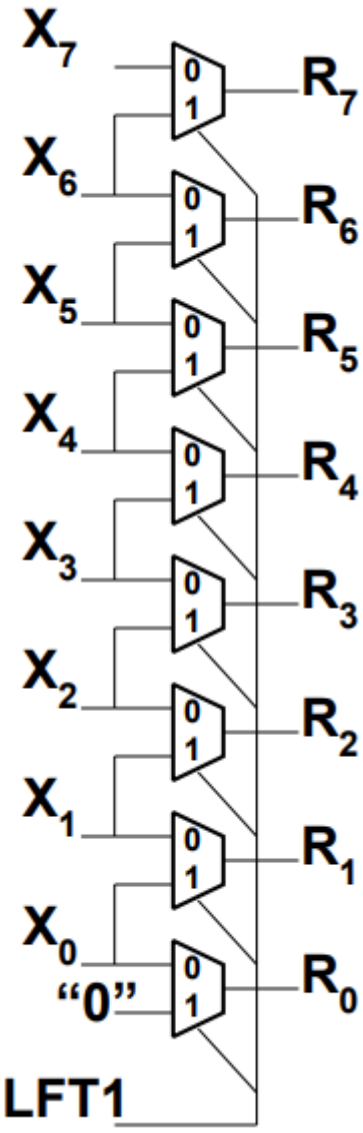


Fig.23 Full Circuit:

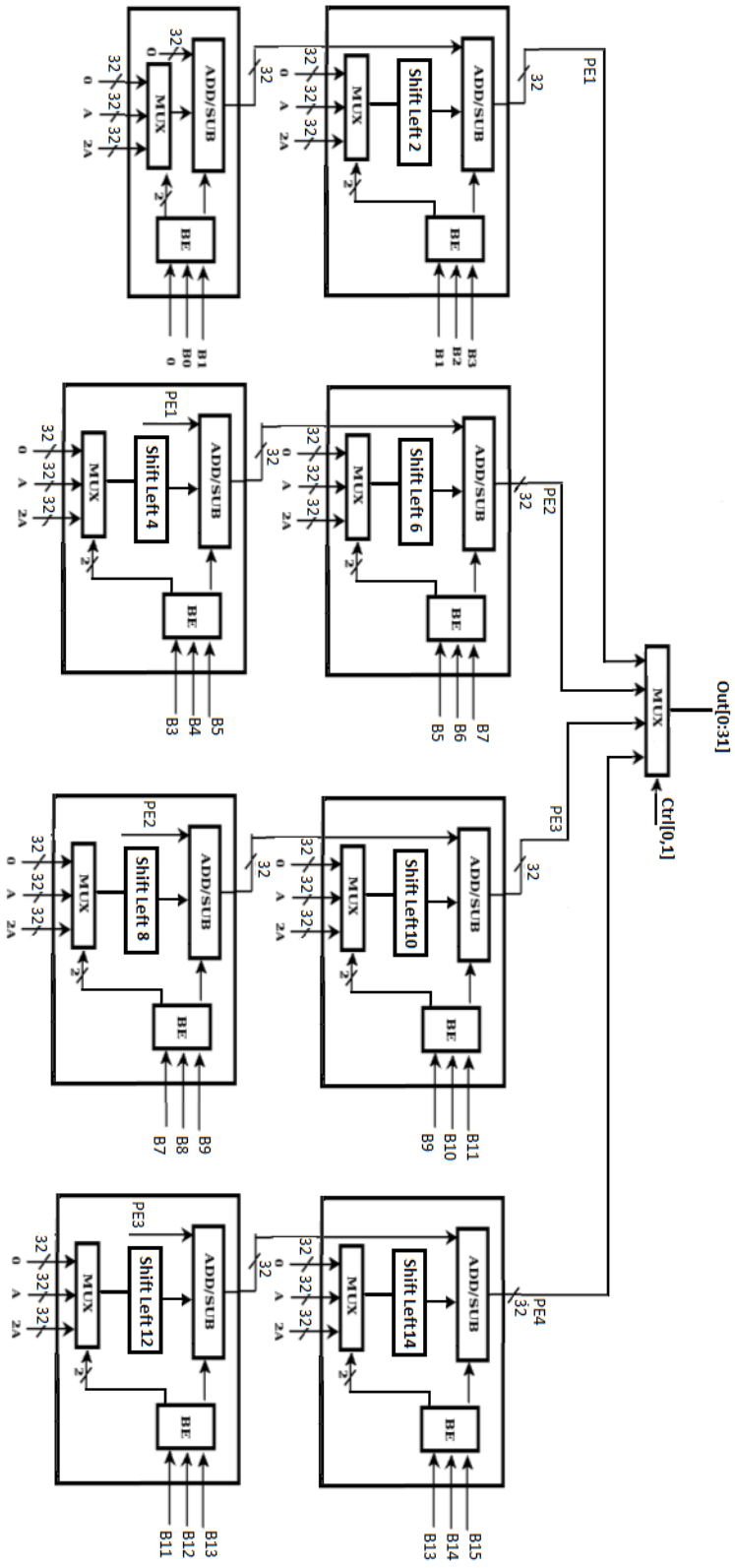


Fig 24. Virtuoso picture of the full circuit: