Unsigned 8-Bit Discrete-Time Convolution

Alan Hsiao (ah668), Chloe Wu(jw995), Emily Wang(jw829), Grace Tan(gnt4)

ECE 4740 Digital VLSI Design

Final Project Report

Table of Contents:

Introduction

Convolution is an essential technique employed in digital signal processing as it combines two signals to form a third. It is an operation on two signals that produces a third signal described as the integral of the product of the two functions after one is reversed and shifted. In discretized time, convolution simply becomes a combination of multiplications and summations. We chose to implement convolution for unsigned 8-bit integer inputs for two arrays with a length up to 3. The resulting outputs are unsigned 16-bit integers in an array with a length up to 5.

Additionally, convolution is used for common everyday applications such as image or audio processing, as well as Convolutional Neural Networks (CNN), a class of deep learning, which uses convolution instead of matrix multiplication. One example is a Field Programmable Gate Array (FPGA)-enabled binarized CNN toward real-time embedded object recognition systems for service robots. In other words, these service robots use convolution for image recognition using FPGAs. These robots must be able to move while recognizing objects and sounds simultaneously, accurate information must be processed and delivered in real time so speed is essential as well as accuracy. In this context, energy is an important factor as these robots are battery operated and recognition demands a lot of energy (Yoshimoto et al.). Area is also an important factor since these systems have an area constraint so that the robot can still be mobile. Moreover, many other small devices, such as smart watches, must be able to process audio and images and have limited area to do so.

Since the signals that are being convolved for practical applications of digital signal processing are very large in size and must be processed in real time, speed must be prioritized. Linear convolution in the time domain is slow because it has a computation time of N_1N_2 multiplications and additions where N_1 and N_2 are the lengths of arrays 1 and 2 respectively. Thus, many algorithms have been developed to optimize computation, such as fast convolution. One example is the radix-2 Fast Fourier Transform algorithm that an estimation in computational advantage compared to direct convolution is $\frac{N^2}{3 \cdot 2N log_2 2N + 2N}$ where N is approximately the number of multiplications and additions (Babic et al.). This is much faster and reduces any computation overhead as such algorithms are developed to ultimately improve execution speed because of what convolution is used for.

With this in mind, we developed design goals that would meet such constraints by prioritizing area and delay product, as well as accuracy. Area is important as it also relates to cost in addition to what is already discussed, speed is essential with processing in real time, and accuracy is preferred when computing convolutions. We also chose to prioritize power, but to a lesser degree since obtaining computations quickly is our primary objective.

Comparison

For the system design, we wanted to optimize the area delay product and power. We came up with three different designs and compared them. The first design was an iterative decomposition architecture (Figure 1). This design reused the same multiplier and adder blocks to perform one set of multiplication and addition of the convolution at a time. Therefore, although it was very area-efficient, it would take 9 clock cycles to compute one convolution, which was not computation-efficient. Additionally, when it comes to testing, this design requires us to sample at very specific clock cycles because the output is not necessarily the desired output at every clock cycle. In other words, this design will output the first element of the output array at the first clock cycle, but at the second clock cycle, the output is the partial product of A_2 and B_1, which is not our desired output. Therefore, this design will increase testing complexity.

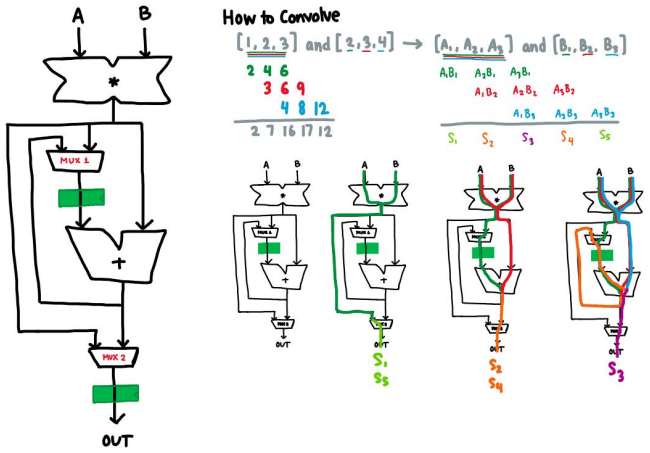

Figure 1: Diagram of Iterative Decomposition Architecture Design

The second design that we came up with was parallel processing architecture (Figure 2). This design implemented parallelism to compute the whole convolution in one clock cycle. Thus, it was very computation-efficient. For testing this design, we can sample the output at every clock cycle. However, this design requires 9 multipliers, 4 adders, and 5 flip flops in total, which takes up too much area and will create more layout overhead. Additionally, it will also consume too much power since there are more components.

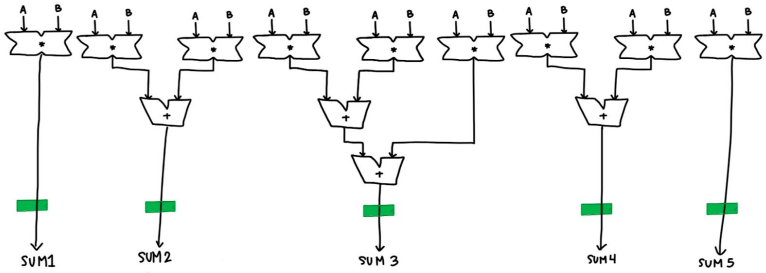

Figure 2: Diagram of Parallel Processing Architecture Design

The third design we had was a modification of the replication architecture (Figure 3). This design computed three multiplications at a time. On the first clock cycle, it would perform three multiplications and one addition to get the first and second elements of the output array. On the second clock cycle, it would perform three multiplications and two additions to get the third element of the output array. Lastly, it would perform three multiplications and one more addition to get the fourth and fifth elements of the output array at the third clock cycle. For testing this design, we would need to sample specific flip flops (FF) on each clock cycle. In other words, we will need to sample FF1 and FF3 for the first and third cycle and FF2 for the second clock cycle.

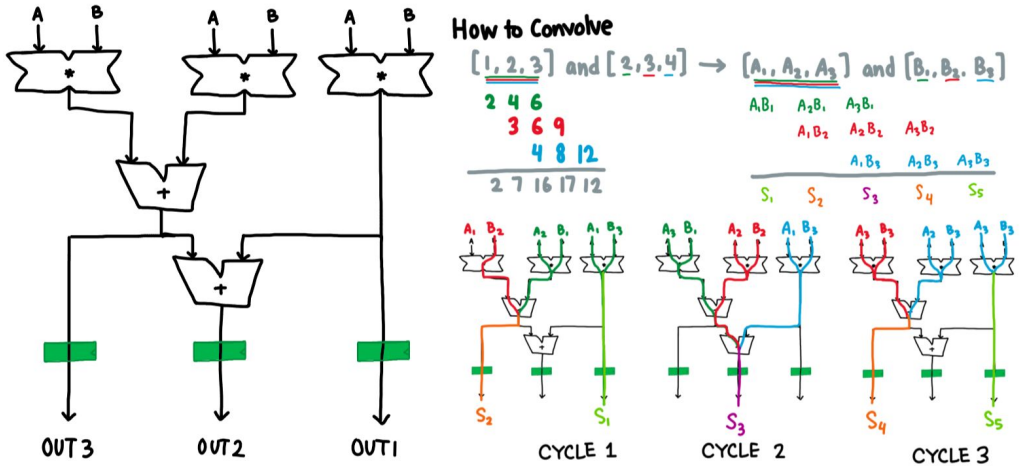

Figure 3: Diagram of Modification of Replication Architecture

Using our existing designs for 16-bit carry implement adder and 1 bit FF, we estimated the propagation delay, area, and power consumption for each design. We estimated our multiplier to have an area of about twice that of the carry implement adder by examining each of their respective components. The carry increment adder consists of 15 greycells, 9 blackcells, 32 XOR gate for propagation and summing logic, and 16 AND gates for the generate logic. The carry save multiplier consists of 8 half adders and 48 full adders. Since the greycells are similar to half adders, and the XOR gate together with the blackcells are similar to the full adder, it makes sense that the multiplier has twice as much area as the carry increment adder because the adder had 24 building blocks and the multiplier had 56 building blocks. As for the power consumption, since the multiplier has 8 bits for input and 7 rows of ripple carry adders like an adder chain, we estimated the power will be 3.5 times of the adder. The delay of the multiplier would then be around the propagation delay of 15 blackcells.

We followed a similar approach to estimate the 16-bit FF and MUX. Since we have the area, delay, and power of a 1-bit FF, we simply multiplied all those information by 16 to get the area, delay, and power for the 16-bit FF. For the 16-bit Mux, we estimated that the area and the power of the MUX would be slightly less than the FF since a 1-bit mux required 3 NAND gates and 1 inverter while a 1-bit FF required 4 transmission gates and 7 inverters . The propagation delay

for the FF would be slightly worse since the critical path of the FF was slightly longer than that of the MUX. The table below summarizes all the results that we discussed above.

As mentioned in the introduction, we wanted to minimize both AT product and power since convolution applications have restraints in area, execution speeds, and power consumption. Since many algorithms focus on optimizing computation time, we prioritized in minimizing delay. We equally prioritized minimizing area as greater area is more expensive and in embedded systems where convolution is used, there is an area restraint. Power is another important factor, however, we decided that power is not as important as area and delay when examining the performance of computing convolutions. Thus, we weighted the normalized AT product and power by 80% and 20% respectively for each of the various designs for comparison.

Looking at the table below, one can see that the iterative decomposition architecture is the most area and power efficient design, but the computation time is much longer for just one convolution, which makes the whole design have the highest normalized AT product and power factor. For the parallel architecture, it is very computation efficient since it only requires one cycle to finish one convolution. However, the normalized AT product and power factor is large due to its high cost of power and area, which would create more layout overhead. Therefore, the second design is not our best solution. The third design has the lowest AT product and power factor so it does not require too much power consumption nor area with minimized delay. Therefore, we decided to go with our third design, the modification of replication architecture.

	Design #1	Design #2	Design #3
Normalized AT Product and Power	8.27E-01	6.96E-01	6.40E-01
Area Delay Product	4.80E-16	2.97E-16	3.41E-16
Total Area	7.60E-09	4.05E-08	1.55E-08
Total Delay	6.31E-08	7.35E-09	2.20E-08
Total Power	9.82E-04	7.23E-03	2.58E-03
Number of Adders	1	4	2
Number of Multipliers	1	9	3
Number of 16-bit Muxes	2	0	0
Number of 16-bit Flip Flops	2	5	3

Table1: Comparison of Three Design Choices

Design Hierarchy

In order to perform convolution, our system requires three major components: the 8-bit multiplier, the 16-bit adder, and the 16-bit flip flop. Each component is then broken down into elements such as cells and gates, and every component and element has its own schematic and layout so that the design is modularized. We started by creating the lowest hierarchical elements, the gates, and tested their individual functionalities in simulation before moving on to the next hierarchy. For major components, such as the adder and the multiplier, we created separate testbenches using veriloga and exhaustively checked their functionality. Additionally, in order to minimize design errors, we performed schematic testing before we started doing layout. To summarize, our hierarchical design approach and thorough testing process significantly increased our efficiency by having solid, functional sub-elements building from the ground up. Figure 4 illustrates the architectural level block diagram from the convolution circuit down to the gate level circuits.

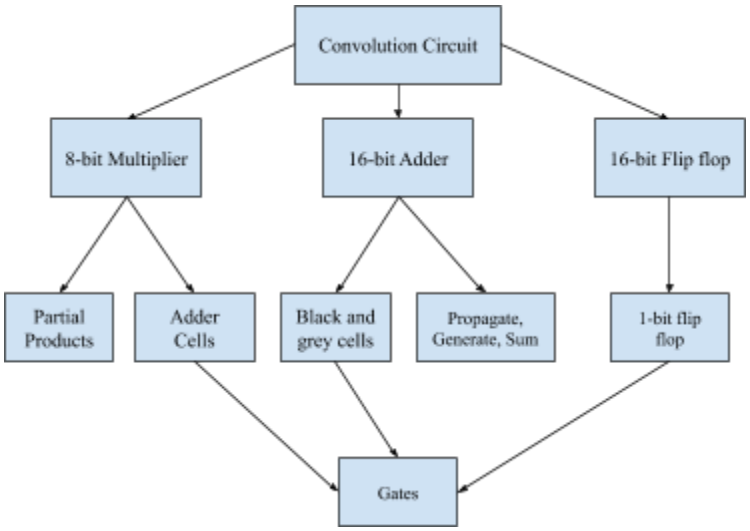

Figure 4: System Hierarchy Block Diagram

Multiplier Design Hierarchy

Figure 5 below illustrates the hierarchy of the multiplier. The carry-save multiplier is made up of 48 1-bit full adders and 8 1-bit half adders. The inputs of the adders are calculated using partial products, which are unit-sized AND gates. The multiplier performs 8-bit unsigned multiplication with 16-bit precision as the output. Its functionality is verified using veriloga and Cadence simulations.

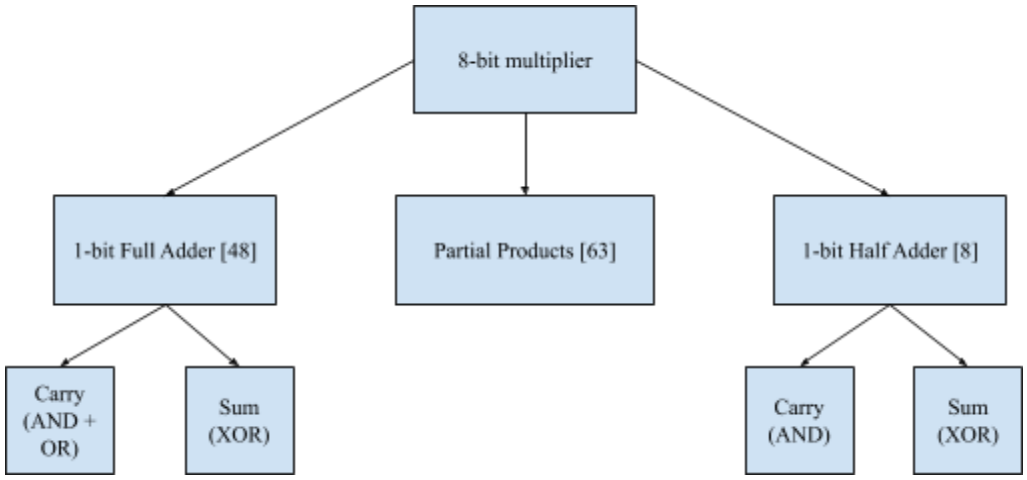

Figure 5: Hierarchy of the 8-bits Carry Save Multiplier

Adder Design Hierarchy

Figure 6 illustrates the design hierarchy for the 16-bit carry increment adder. The adder generates the G_i<15:0> and P_i<15:0> using the unit-sized AND and XOR gates, respectively, and the Generate of each bit is fed into a black or grey cell, where its output is XOR'ed with the Propagate of that bit to create the sum bit. Additionally, the 1st-bit sum requires additional logic [$G_0 + P_0 * C_{in}$] in order to account for the carry-in, adding an additional propagate and generate. The functionality of the adder is verified using veriloga, Cadence simulations, and MATLAB.

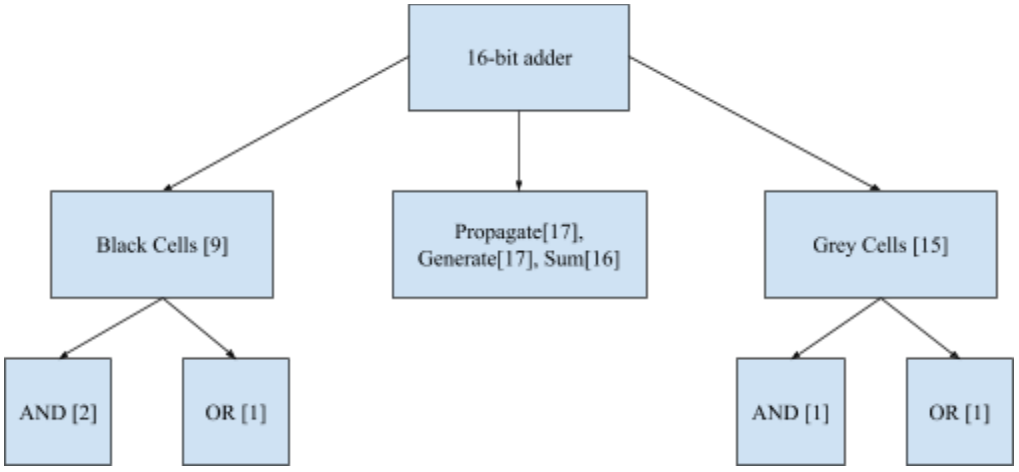

Figure 6: Hierarchy of the 16-bits Carry Increment Adder

Flip Flop Design Hierarchy

The 16-bit flip flop is created from 16 1-bit flip flops that all share the same clock. The 1-bit flip flop is composed of two latches so that the data is stored when the clock is high. It was first thoroughly tested and verified using Cadence simulations. Then the 16-bit flip flop was created and also checked for its functionality and performance through Cadence simulations.

Design Methodology

System Design

As we discussed in the design comparison section, our final design of the system utilizes a modification of repetition architecture. It consists of a 3 8-bit carry save multipliers, 2 16-bits carry increment adders, and 3 16-bits FF. It will calculate one convolution every three clock cycles. This design has the most optimized normalized AT product and power factor.

We also noticed there are tradeoffs in prioritizing maximum inputs over accuracy as shown through an example in the table below. If the highest unsigned 8-bit number is used for each element in the input arrays for the convolution, some of the outputs are greater than 16 bits and our adder only supports up to 16 bits. In this case, the least significant bits would be cut off and accuracy is compromised. Since it is important to obtain correct results with precision for convolution, we prioritized accuracy and decided that limiting the maximum input to 147 is better than losing precision. We determined 147 as the highest input through the following calculations. Since 1111111111111111 (65535) is the highest unsigned adder output we can get, the highest unsigned multiplier input we can have is $(65353/3)^{.5}$ =147.8 so 147 is the highest multiplier input we can have.

Maximum Input			Accuracy		
255 in binary: 11111111			147 in binary: 10010011		
Convolving [255,255,255] and [255,255,255]			Convolving [147,147,147] [147,147,147]		
Result is [65025, 130050, 195075, 130050, 65025]			Result is [21609, 43218, 64827, 43218, 21609]		
Output (decimal)	**Output (binary)**	**Number of Bits**	**Output (decimal)**	**Output (binary)**	**Number of Bits**
65025	0011111111000000001	16	21609	101010001101001	16
130050	0111111110000000010	17	43218	1010100011010010	16
195075	1011111101000000011	18	64827	1111110100111011	16

Table 2: Comparison of prioritizing maximum input versus accuracy

Adder Design

In order to perform 16-bit addition, we first compared the different types of adders and compared their Area-Delay products (Table 3). While Sklansky has the smallest AT product, the circuit is

not wiring-friendly and has too much overhead for our purpose. Additionally, Sklansky also has high power consumption due to its design complexity. Thus, we chose the carry increment adder because we want to optimize the AT product, power, and wiring.

Design	Area	Delay	AT
Sklansky	32	4	128
Brent-Kung	26	7	182
Carry Increment	26.34	5.66	149.08

Table 3: Comparing the area, delay and AT product of different types of adders.

Adder Sizing

Figure 7 below illustrates the block diagram of the carry increment adder as well as its two critical paths. Since the blocks are parallelized in groups of three (i.e. bits 1-3, 5-7, 9-11, and 13-15), we decided to size the critical path in purple, bits 5 to 14, and sized the parallel branches similarly. The purple path is made of 27 stages and is comparable to the blue critical path because the black cells on bits 5, 6, and 7 each have to drive an additional NAND for its group propagate. The cells that are not on the critical paths are all unit-sized in order to minimize the area. The sizing of the cells and the gates are shown in Figure 8.

For this adder, we sized the critical path at the gate level and optimized its delay by placing each cell around the critical path. By doing full gate-level sizing, we spent about 10x more time doing layout than we would need if we just optimize one cell and use it for the entire design. Even though gate-level sizing was efficient for critical path delay, it was not ideal in terms of layout time.

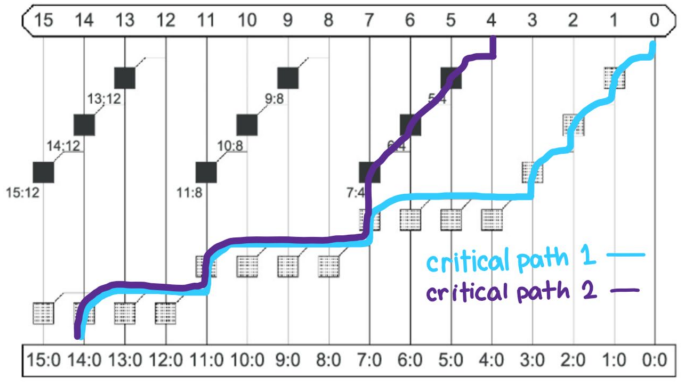

Figure 7: Block diagram of the carry increment adder with its 2 critical paths.

	Gate	Stage	f_opt	g		b	Size#	OUR SIZE	Sizes	Cin
AND	NAND	1	1.446	1.333	0.75	1.000	0	1	1.000	4
	INV	2	1.446	1.000	1.333	1.000	1	1	1.446	3
Black Cell on Bit 5	NAND	3	1.446	1.333	0.75	1.000	2	1.5	1.567	4
	INV	4	1.446	1.000	1.667	1.000	3	2	2.265	3
	NOR	5	1.446	1.667	0.6	1.000	4	2	1.965	5
	INV	6	1.446	1.000	1.333	2.375	5	3	2.840	3
Black Cell on Bit 6	NAND	3	1.446	1.333	0.75	1.000	6	1.5	1.296	4
	INV	8	1.446	1.000	1.667	1.000	7	2	1.874	3
	NOR	9	1.446	1.667	0.6	1.000	8	1.5	1.625	5
	INV	10	1.446	1.000	1.333	2.375	9	2	2.350	3
Black Cell on Bit 7	NAND	3	1.446	1.333	0.75	1.000	10	1	1.073	4
	INV	12	1.446	1.000	1.667	1.000	11	2	1.550	3
	NOR	13	1.446	1.667	0.6	1.000	12	1.5	1.345	5
	INV	14	1.446	1.000	1.333	1.375	13	2	1.944	3
Gray Cell on Bit 7	NAND	3	1.446	1.333	0.75	1.000	14	1.5	1.533	4
	INV	16	1.446	1.000	1.667	1.000	15	2	2.215	3
	NOR	17	1.446	1.667	0.6	1.000	16	2	1.922	5
	INV	18	1.446	1.000	1.333	4.375	17	3	2.778	3
Gray Cell on Bit 11	NAND	3	1.446	1.333	0.75	1.000	18	0.5	0.688	4
	INV	20	1.446	1.000	1.667	1.000	19	1	0.995	3
	NOR	21	1.446	1.667	0.6	1.000	20	1	0.863	5
	INV	22	1.446	1.000	1.333	4.375	21	1	1.247	3
Gray Cell on Bit 14	NAND	3	1.446	1.333	0.75	1.000	22	0.5	0.309	4
	INV	24	1.446	1.000	1.667	1.000	23	1	0.447	3
	NOR	25	1.446	1.667	0.6	1.000	24	1	0.388	5
	INV	26	1.446	1.000	1.333	1.000	25	1	0.560	3
XOR	XOR	27	1.446	1.500	11.111	1.000	26	1	0.607	4
Overall			20927.608	33.833	4.167	148.452				

Bit #		Cell Name
Critical Path (starting w/ black cell bit 5)		
5	(greycell_7 + unit sized AND)	blackcell_1
6	(greycell_2 + unit sized AND)	blackcell_2
7	(greycell_3 + unit sized AND)	blackcell_3
7		greycell_7
11		greycell_11
14		greycell_11
Other Grey Blocks		
1		greycell_7
2		greycell_2
3		greycell_3
4-6		greycell_unit
8-10		greycell_unit
12-13		greycell_unit
15		greycell_unit
Other Black Blocks (unit sized ANDs)		
9		blackcell_1
10		blackcell_2
11		blackcell_3
13		blackcell_1
14		blackcell_2
15		blackcell_3

Figure 8: Carry-increment adder gate-level sizing

Multiplier Design

Another major design decision we made was the 8-bit multiplier. We first compared the three types of multipliers we learned in class: the array multiplier, carry-save multiplier, and the Wallace tree multiplier. Since the computation only takes 8-bit inputs, we decided that the Wallace tree multiplier has too much layout overhead and complexity for our convolution. Comparing the array multiplier and the carry-save multiplier, the array multiplier has a longer critical path than the carry-save due to the nature of its zig-zagging carry-out paths. As a result, we chose the 8-bit carry-save multiplier for our convolution, as shown in the following figure.

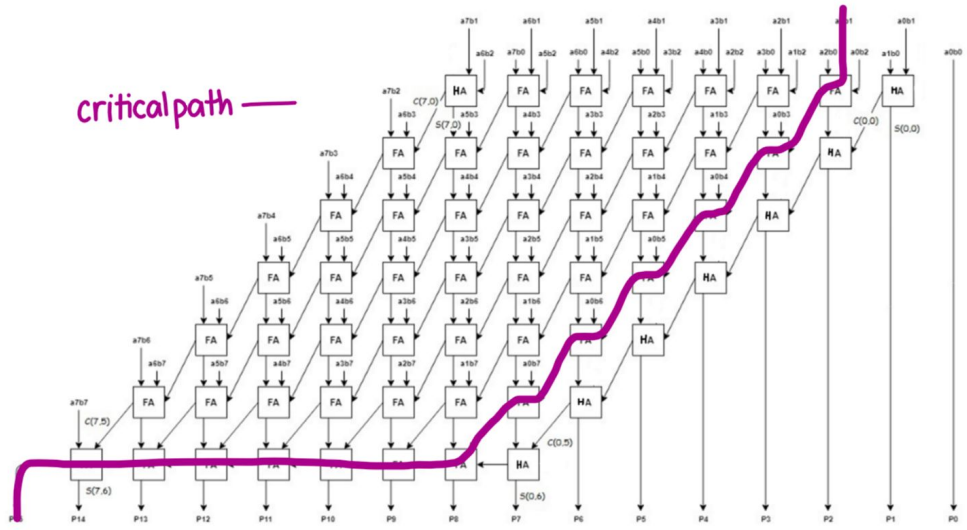

Figure 9: Block diagram of the 8-bit carry save multiplier

Multiplier Sizing

The carry-save multiplier has both full adders and half adders as its sub-blocks. In order to optimize both the work efficiency and component delay, we sized the individual sub-blocks and used the same block throughout the entire design. For the 1-bit full adder (Figure 10), the two paths that need to be sized are the carry path (1 AND and 1 OR gate) and the sum path (2 XOR gates). For the half adder, both the XOR and AND gates are sized 1 because they are the only component on the path. The block-level size calculation is shown in figure 11.

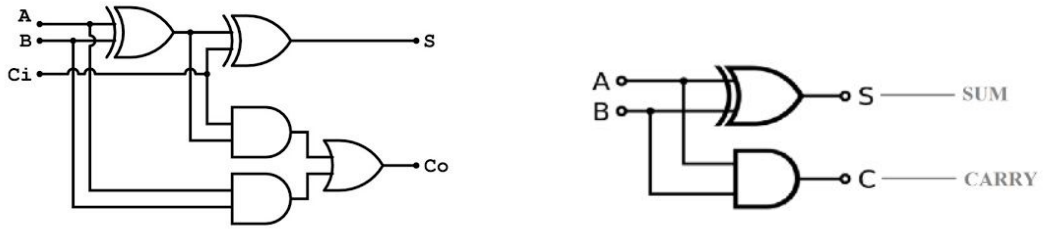

Figure 10: Block diagram showing sub-block components of the carry-save multiplier with the full adder on the left and the half adder on the right.

Sum Logic	Gate	Stage	f_opt	g	h	b	Our Sizes	Sizes	Cin
	XOR	1	3.000	1.667		1.80	1	1.00	5
	XOR	2	3.000	1.667		1.80	1	1.00	5
			9	2.777777778	1	3.24			
Carry Logic									
AND	NAND	1	1.495348781	1.333		1.000	1	1.0000	4
	INV	2	1.495348781	1.000		1.000	1.5	1.50	3
OR	NOR	3	1.495348781	1.667		1.000	1.5	1.34	5
	INV	4	1.495348781	1.000		2.250	2	2.01	3
Overall			5	2.222222222	1	2.25			

Figure 11: Carry Save Multiplier Block-level Sizing

Gate Level Design

During our project, we tried out two types of design methods: the adder using gate-level sizing and the multiplier using block-level/system-level sizing. For gate-level sizing, we broke down the critical path into NOR, NAND, INV, and XOR gates and sized all 28 stages of the adder's critical path (Figure 11). The gate-level sizing involves more complexity because the black and grey cells are not modularized throughout the design and each cell must be designed and laid out separately. For block-level/system-level sizing, we were able to optimize delay and layout by sizing one block and reusing it throughout the design with an expected propagation delay (Figure 11). In this case, the gate-level design for sub-blocks occurs once, whereas for full gate-level sizing, all cells must go through a different gate-level design.

XOR Gate Design

We had two designs for the XOR. One used transmission gate logic to reduce area, and the other used inverting logic to prevent potential drop in output voltage. The figure below shows the two XOR gate designs. Based on our critical path calculation for both the adder and the multiplier, the XOR gate is unit size. As for the transistor level sizing, we sized the pmos and nmos 2 to 1 in the inverting logic XOR design so that we have equivalent rise and fall time for the propagation delay. We sized the XOR gate with transmission logic with 2 to 1 pmos to nmos ratio for the same reason. For the transmission gate, we sized the pmos to nmos ratio 1 to 1 since only one gate will be on at a time.

One advantage of using transmission gate logic is that it saves area. However, since transmission gate logic does not have the regenerative properties, there will be a V_T drop at the output if using the nmos as a pullup or pmos as a pulldown. In contrast, inverting logic has the regenerative properties that restore the output back to its normal level but it also requires 6 more transistors, which increases our area. In our 16-bit adder design, we used the transmission gate logic XOR gate since the XOR gate is only used for propagate and summing logics. So there is no transmission gate cascaded together to cause significant voltage drop. However, the transmission gate logic will cause problems in the multiplier since we have a path going diagonally down through the full adder. We will talk more about this problem in detail in the troubleshooting section.

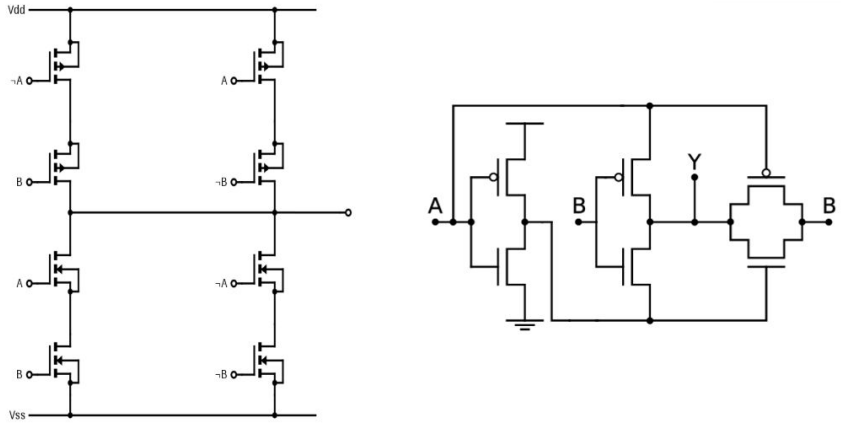

Figure 12: XOR gate with Inverting Logic (Left) and Transmission Gate Logic (Right)

Functionality

Testing Strategies

After conducting a trade study on the different topologies and choosing a final design, our next goal was to create a signal generator module, schematic, and MATLAB test script to verify that our design produced the correct output. Since we heavily relied on test-driven development to guide our design choices, simulations were run to ensure correct implementation of subcomponents before moving on to test higher levels of the hierarchy. At the gate and subcomponent level (full adder, half adder, group generate, group propagate), we implemented exhaustive testing to ensure that each gate and subcomponent implemented the correct logic function for all sets of possible inputs. This testing strategy allowed us to quickly locate issues at the component level (8-bit multiplier, 16-bit adder, 16-bit FF) as a result of schematic designs while having full confidence that the subcomponents were implemented correctly.

Since there are over 4-billion possible permutations for components that have two 16-bit inputs, exhaustively evaluating our designs at a component level is not reasonable. Therefore, we implemented carefully crafted directed tests which consisted of fault isolating unit tests (such as adding all 0s) to uncover as many faults as possible. To increase our confidence in the correctness of our design, we then extend to running random tests that cover different patterns and parameters. Lastly, when our components have run successful directed and random tests, we reiterate the testing process for the full convolution circuit. Our directed tests for the full convolution circuit utilized our minimum and maximum values (0 to 147) while random tests were generated with an online random number generator.

MATLAB and Verilog

To ensure that our final convolution calculates the correct values, we wrote an extensive MATLAB script to sample the Cadence simulation inputs and outputs, calculate the expected convolution using the inputs, and then compare the binary values of the expected result with the obtained result. As shown in the *Correct Convolution Output* screenshot (Figure 14), the script will state that the convolution has no errors when no error flags are thrown in the comparison code. Meanwhile, the Cadence simulation inputs are provided by our Verilog code which breaks down each array into three 8-bit inputs. Originally, we wanted to run multiple convolutions per each simulation, but quickly realized that it would be inefficient due to the long simulation times (+20 minutes). Therefore, though our code for both MATLAB and Verilog contain remnants of our past design thoughts, it only runs one convolution case at a time.

```
for i = 1:numCases
    vec_out = conv(a(i),b(i));
    vec_out = double(dec2bin(vec_out,16) == 49);
    exp_convo_output(i,:,:) = vec_out(:,:);
end
```

Figure 13: Expected Convolution Calculation [MATLAB]

[27,9,127] ⊛ [121,1,127] → [3267, 1116, 18805, 1270, 16129]

```
>> Lab5_signalverification_matlab
Your convoluted corona convolution has no errors :)
>> reshape(exp_convo_output,5,[])

ans =

     0     0     0     0     1     0     1     0     0     0     1     0     1     0     0     0
     0     0     0     1     1     0     1     1     1     1     0     0     0     1     1     0
     0     0     1     1     0     1     1     1     0     1     0     1     0     1     0     1
     0     0     0     1     1     0     0     1     0     0     1     0     1     1     0     0
     0     0     0     0     1     0     0     0     1     1     0     0     0     0     0     1

>> reshape(my_convo_output,5,[])

ans =

     0     0     0     0     1     0     1     0     0     0     1     0     1     0     0     0
     0     0     0     1     1     0     1     1     1     1     0     0     0     1     1     0
     0     0     1     1     0     1     1     1     0     1     0     1     0     1     0     1
     0     0     0     1     1     0     0     1     0     0     1     0     1     1     0     0
     0     0     0     0     1     0     0     0     1     1     0     0     0     0     0     1
```

Figure 14: Correct Convolution Output [MATLAB]

```
>> Lab5_signalverification_matlab
Info, results /home/ah668/Cadence/simulation/Lab5_convolution_circuit_testbench/spectre/schematic/psf are changed, reloading
Expected output for Case #1 and Sum #1 for bit #9 should be 1 but the measured output is 0
Expected output for Case #1 and Sum #2 for bit #9 should be 1 but the measured output is 0
Expected output for Case #1 and Sum #2 for bit #7 should be 1 but the measured output is 0
```

Figure 15: Incorrect Convolution Output [MATLAB]

```
//convolution of (27, 9, 127) and (121, 1, 127) = (3267, 1116, 18805, 1270, 16129)
parameter integer test1_a_1[7:0] = {0, 0, 0, 1, 1, 0, 1, 1};
parameter integer test1_a_2[7:0] = {0, 0, 0, 0, 1, 0, 0, 1};
parameter integer test1_a_3[7:0] = {0, 1, 1, 1, 1, 1, 1, 1};
parameter integer test1_b_1[7:0] = {0, 1, 1, 1, 1, 0, 0, 1};
parameter integer test1_b_2[7:0] = {0, 0, 0, 0, 0, 0, 0, 1};
parameter integer test1_b_3[7:0] = {0, 1, 1, 1, 1, 1, 1, 1};
```

Figure 16: Lab 5 Signal Generator [Verilog]

LVS and DRC

After the schematic has successfully passed all of our verification tests, we need to do layout to get a more reasonable estimate of the area, delay, and power consumption of our circuit. While some subcomponents had been finished in previous labs, we needed to tweak some designs and start from scratch for others. We were able to successfully pass DRC and LVS on the final layout and run Quartus QRC to extract parasitics.

Performance

We evaluated the performance of our convolution based on the FoM as defined in our design comparison section. To reiterate, since many convolution modules are utilized in embedded system applications which oftentime have area, delay, and power constraints, we decided that optimizing area, delay, and power would be most beneficial. However, due to real time constraints for signal processing systems such as in autonomous car applications (where time constraints result in life or death), we weighted the area delay product at 0.8 and the power at 0.2 of the total area-delay-power constraint.

In terms of area, the full layout can be contained within a 146µm by 149µm rectangle which equates to about 21,000µm^2. However, if we take into account that there is excess space in the top right corner as well as the bottom to fit additional components, we can estimate an area of about 17,500um^2 which is only 11% bigger than our estimated value of 15,500µm^2 (which did not account for routing).

To measure delay, we ran several simulations for individually extracted components to get an idea for best and worst case delay times. We were also able to run one random simulation (which passed) on the full av_extracted version of the convolution circuit. If we had additional time, we would run more test cases on the full av_extracted convolution circuit, but since one simulation took about 45 minutes to run with nestlvl=1, we decided to prioritize optimizing other aspects of our design over spending large amounts of time to ensure that layout outputs matched an already correct schematic output. In the end, we estimate that the max clock period for our flip-flops will be about 8.5ns which leads to a convolution period of 25.5ns or a max convolution frequency of 39.2 MHz. Compared to our estimate of 22ns (max convolution frequency of 45.45 MHz), we only have 13.7% error. In order to ensure correct sampling of the inputs and outputs, our veriloga and MATLAB scripts are set to the upper bound of clocking the flip-flops at 20ns with a convolution period of 60ns (max convolution frequency of 16.67 MHz). We made this decision mainly based on the uncertainty in delay for the full av_extracted convolution circuit (which could not be exhaustively tested).

For power on the generic full av_extracted convolution test case, we were able to convolve two arrays with 0.3365 mW. Since we used the worst case power consumption for each component in our area-delay-product calculation, we obtained a much higher value of 2.58 mW. However, this power consumption is physically impossible to obtain because the multiplier feeds into the adder and the worst case output for the multiplier is not the worst case input for the adder. Additionally, since the power obtained for the full av_extracted convolution is for a generic convolution, we expect the power to increase for worst case scenarios and decrease for best case scenarios.

Troubleshooting

Problem with XOR Gate

As we were testing our convolution circuit, it failed for the case where the two inputs to the multiplier were 10000010 and 00010100 respectively. The 9th bit of our output was supposed to be a 1 while the Cadence simulation of our circuit gave a 0. As we took a deeper look into our circuit, we found out that there was a problem with the XOR gate with transmission gate logic. In our carry save multiplier design, the sum of a full adder fed into the input of the next stage full adder. The sum path of the full adder goes through two XOR gates. Therefore, there was a cascaded V_T drop going diagonally along the multiplier, which could drop an input signal of 1 down to 0. The figure below shows the Cadence simulation with the cascaded V_T drop.

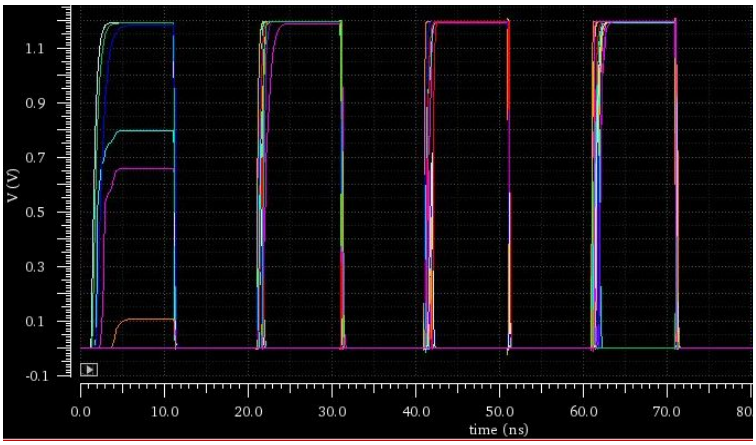

Figure 17: Cadence Waveform Showing cascaded V_T drop

To fix this problem, we came up with two solutions. One is to change all our XOR gates in the full adder to use inverting logic instead of transmission gate logic (as we talked about in the XOR gate design section) because inverting logic has the regenerative properties that would restore our output to its normal level. The other solution is to add a buffer after each sum path so that the buffer could pull the output up after the transmission gate.

We decided to change our design of the XOR gate to the one with inverting logic and ran Cadence simulation to prove that the XOR gate was indeed the problem. The figure below shows the Cadence simulation of the multiplier with inverting logic XOR gate design. And one can see that there is no more voltage drop along the full adder chain. We found out this problem after we finished our multiplier layout. Therefore, changing the XOR gate logic would require us to redo the layout, which was not possible given the time we have left. Therefore, our final layout of the multiplier still used the XOR gate with transmission gate logic, which would fail one of the test cases we had. If we had more time, we would have laid out the multiplier with the inverting logic XOR gate.

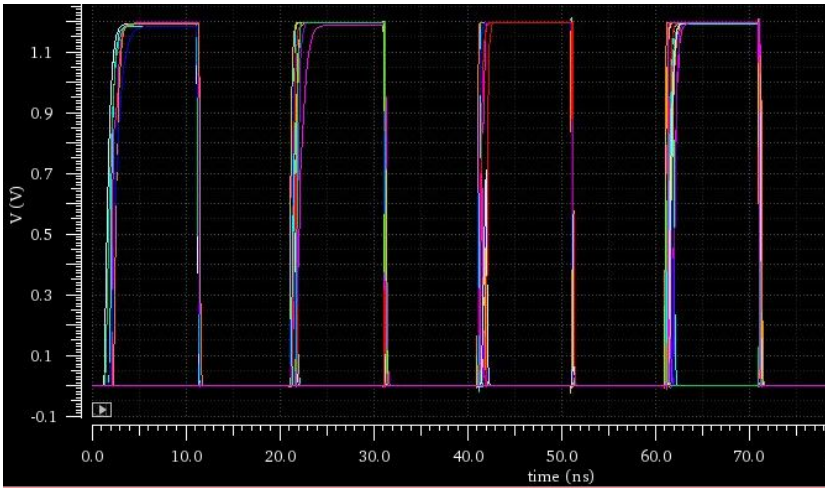

Figure 18: Cadence Waveform with Inverting Logic XOR Gate

Another possible solution that we did not implement is to use a keeper circuit at the output of the XOR gate. The keeper circuit will help retain the output at high. But using the keeper circuit will require ratio'd logic. We need to make sure that the keeper is weak so that it will not pull the output to high when it is supposed to be low.

Problem with 16-bits Carry Increment Adder

We also had issues with our 16-bits carry increment adder. It did not pass all the test cases that we have created. As we were checking the sum and carry out of each bit, we found out that the group generate logic for the 0th bit was not correct. In our adder design, the group generate, which is the carry out from the previous bit will feed into the generate of the black/grey cells. Originally, we had the generate signal of 0th bit feeding directly to the generate of the first grey cell, but it should have been the carry out of the 0th bit. We re-ran the Cadence simulation to test our adder after we changed the group generate logic to the carry out logic, and it passed all the test cases. We then tested more random cases and the outputs were all correct.

Glitches

We also had problems with glitches from the adder and the multiplier due to the change in inputs. We sized our adder and multiplier for minimum propagation delay, but it improved the glitches very slightly. The glitches increased our propagation delay, and we needed to sample our output at a later time to get the correct outputs.

Comments

Thank you so much for a great semester! We really appreciate your patience and the time you all have dedicated to helping us with learning the material and figuring out why Cadence is weird and annoying sometimes. Hope to see you all on campus again soon :)

Course Staff	Contributions
Robin Ying	• "Do you see the wiggle?" • Flip flops (FF) 4 days (except when snowing) • Best Personal Tutor Award (in OH)
Thomas Tapen	• Inverted Zoom Drawings • Cadence Wizard • Best Lab TA Award
Alec Newport	• Best MEng TA Award • Synopsys + Innovus Whisperer • Homework Auto-(route)grader

Additional Figures

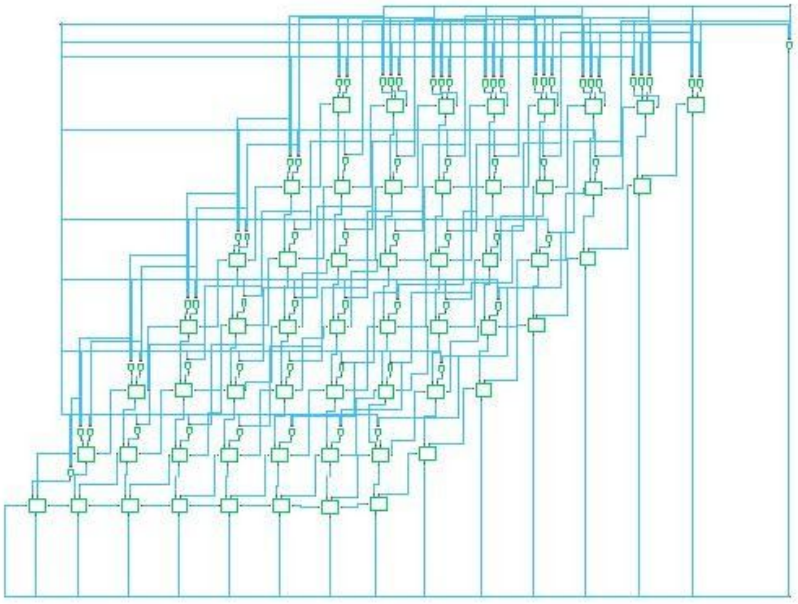

Figure 19: Schematic of the 8-bit Carry Save Multiplier

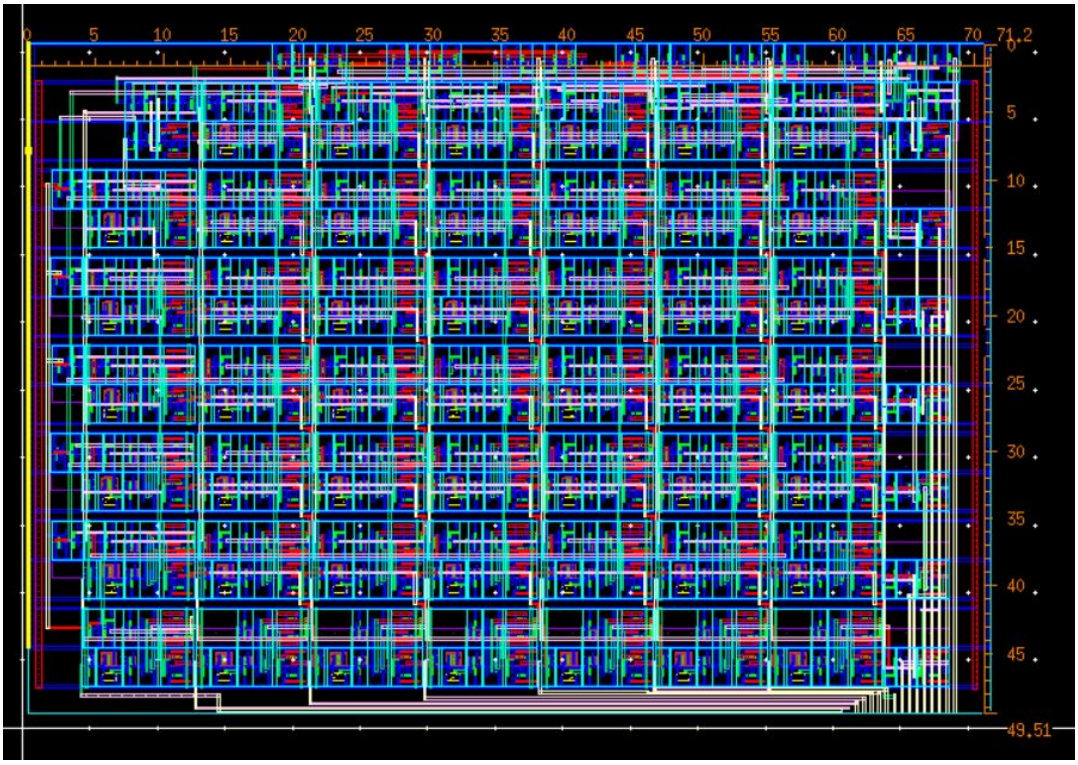

Figure 20: Layout of the 8-bit Carry Save Multiplier

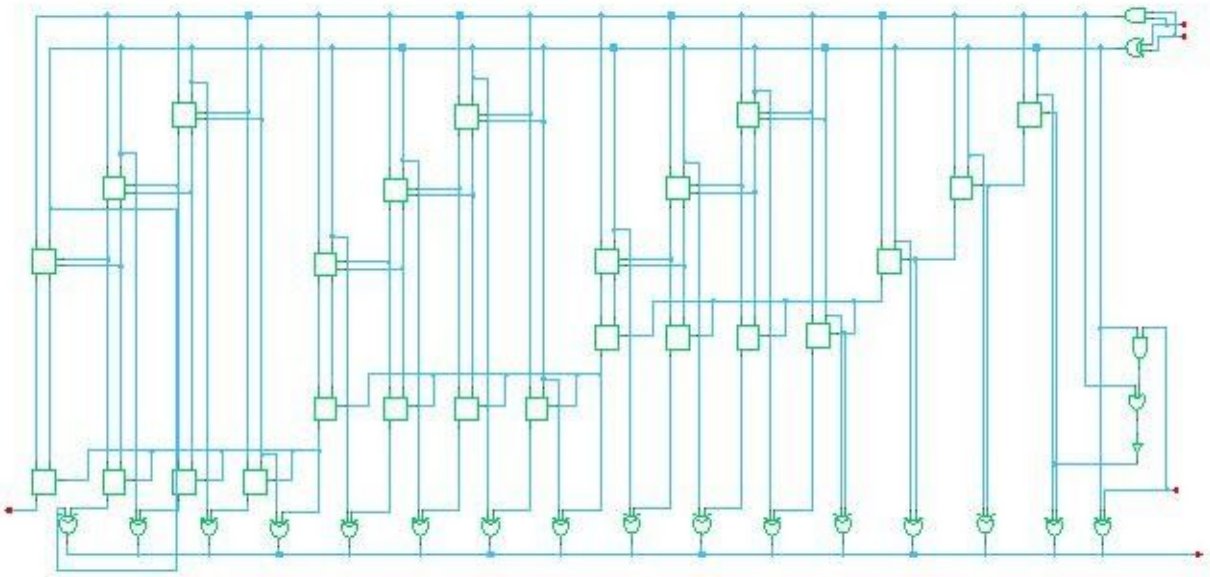

Figure 21: Schematic of the 16-bits Carry Increment Adder

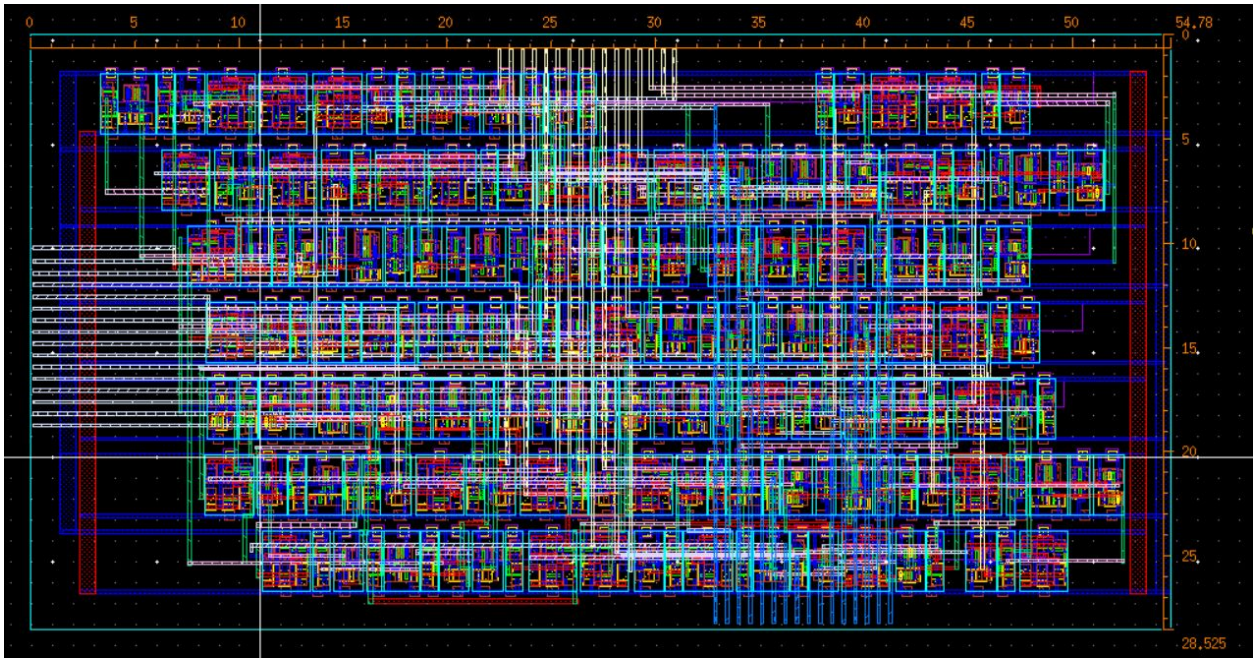

Figure 22: Layout of the 16-bits Carry Increment Adder

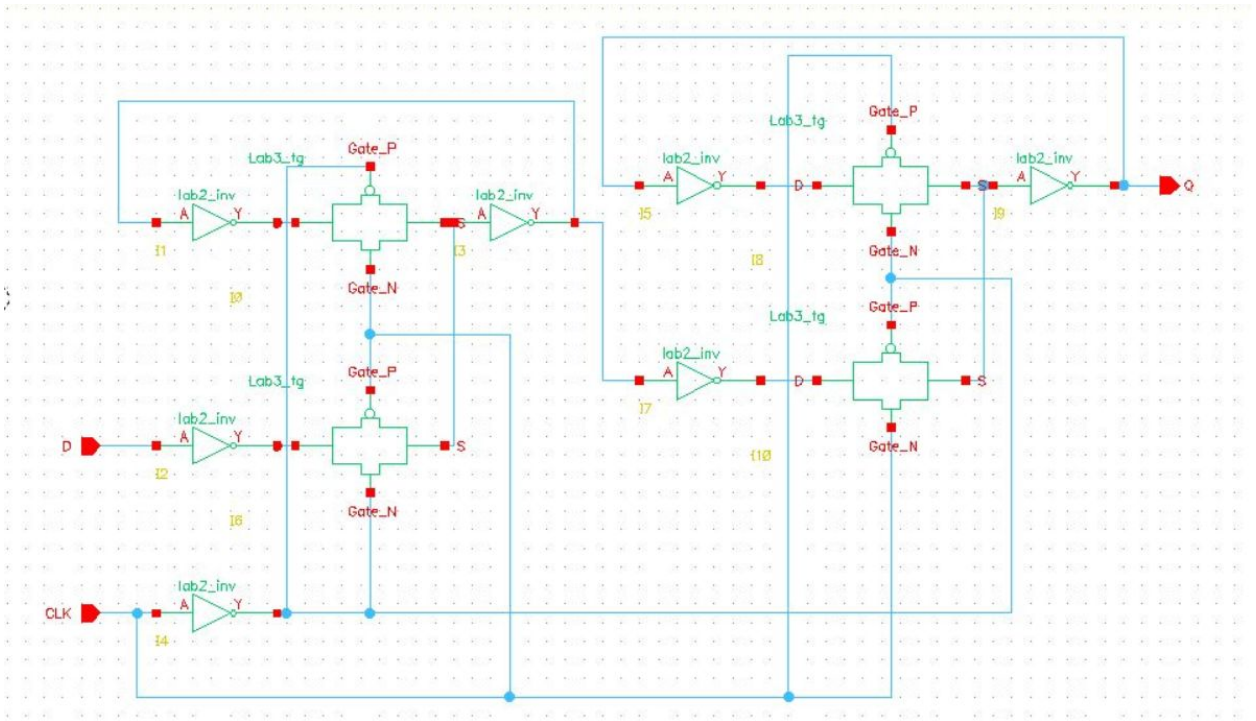

Figure 23: Schematic of the 1-bit Flip Flop

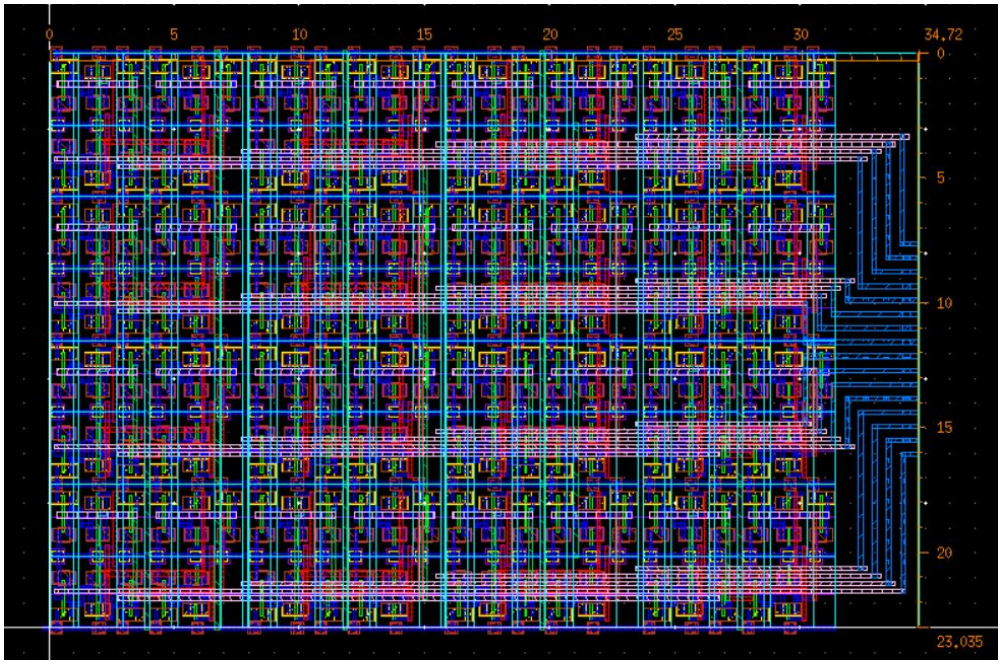

Figure 24: Layout of the 16-bits Flip Flop

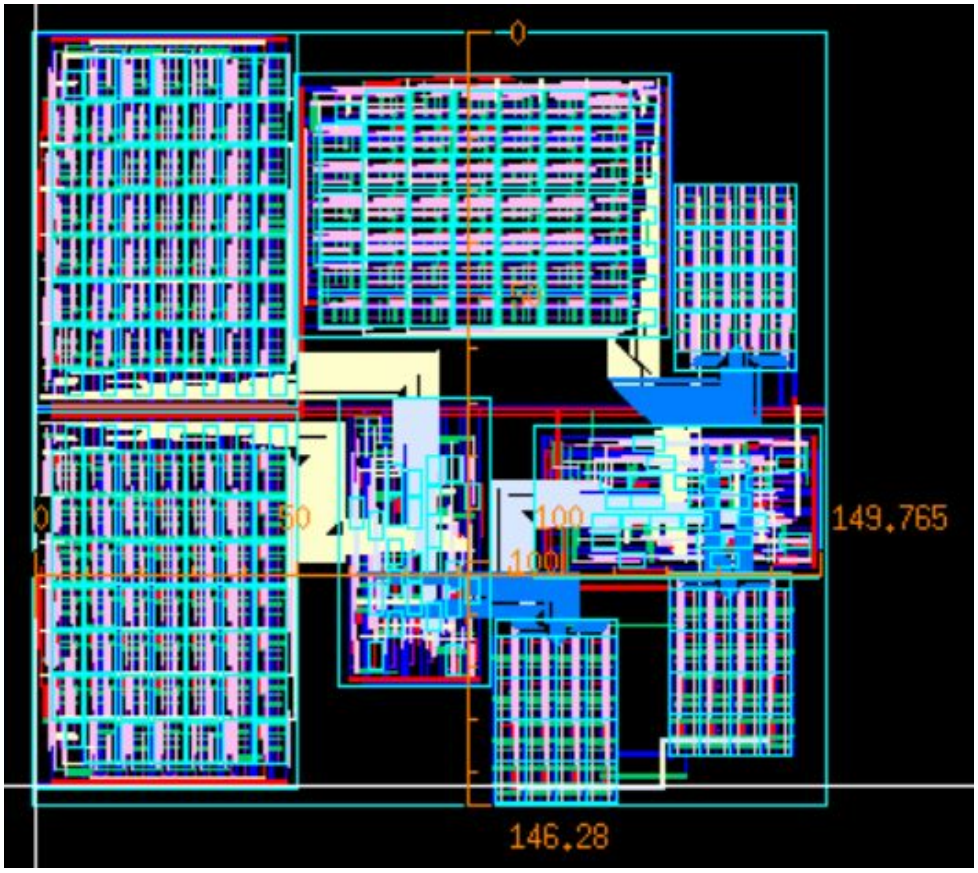

Figure 25: Layout of the Convolution System

References

Babic, Z., & Mandic, D. (2001). A fast algorithm for linear convolution of discrete time signals. 5th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Service. TELSIKS 2001. Proceedings of Papers (Cat. No.01EX517). doi:10.1109/telsks.2001.955846

Yoshimoto, Y., Shuto, D., & Tamukoh, H. (2019). FPGA-enabled Binarized Convolutional Neural Networks toward Real-time Embedded Object Recognition System for Service Robots. 2019 IEEE International Circuits and Systems Symposium (ICSyS). doi:10.1109/icsys47076.2019.8982469