Technical University of Denmark

02204 Design of Asynchronous Circuits

Globally Synchronous Locally Asynchronous Sequential Multiplier

---

Anthon Vincent Riber
s154663

Simon Thye Andersen
s154227

Group: 1

We confirm that this report contains our own independent and original work. All group members have contributed equally to all parts of the work.

May 11, 2020

Contents

1 Introduction

Asynchronous circuits are an alternative for synchronous systems, where the signals propagate from one state holding element to the next based on handshake between the two instead of being governed by a globally shared synchronous clock signal. In other words - Synchronization ensured via handshake protocols. There exist different ways to handshake between components including dual-rail and bundled data 4-phase and 2-phase versions [1].

This report will investigate the use of an asynchronous multipliers, implemented in a synchronous system. A multiplier is a highly used component in signal transformation equations, such as when performing an FFT. When working with very low-power systems, such as hearing aids, the overall system is often clocked very slow to conserve power. This normally requires more area to perform deeper calculations, as the additions has to be done in parallel. We will here look at implementing a multiplier using a carry-save-adder ring for constant-time additions in a ring with small area consumption. The idea here is to utilize asynchronous design mythology to "clock" the system faster than the rest of the system for a globally synchronous locally asynchronous (GSLA) implementation. To make a module compute a multiplication in a single clock cycle it requires a very wide implementation, as up to n additions have to be made for an $n*n$ adder. The asynchronous solution would be able to use a ring which can run faster than the clock and can, therefore, perform multiple iterations of a carry-save ring-based multiplier in a single synchronous clock cycle (still assuming a slow clock).

The power consumption should be somewhere along a single cycle and a multi-cycle synchronous implementation, as the power should be normalized per multiplication. As such, a multi-cycle multiplier would consume less energy per clock-cycle but take more cycles and we are therefore interested in the total energy consumption per multiplication performed.

An asynchronous version is expected to consume more area compared to a synchronous version of a ring-based carry-save multiplier, as the same computational units are used, but needs handshaking circuit overhead. To be a good solution it should:

- Be able to perform many iterations per slow clock cycle
- Consume less area than a synchronous single-cycle multiplier
- Have a comparable power consumption for a diverse set of test vectors

The source code for this project can be found at `https://github.com/nothinn/async_multiplier`. The part of the code that we wrote our self have can be found in the appendix.

2 Design

We will in this section take the reader through the design of a single cycle multiplier, which has the obvious advantage of being able to produce a purely combinatorial multiplication in a single cycle. This introduction will illustrate the consequences of single-cycle multiplication, being the size and therefore also leakage and dynamic power consumption. This will lead to an introduction of the design of the alternative multi-cycle synchronous adder, which has reduced power consumption due to reduced footprint. The multi-cycle multiplier has the disadvantage of needing up to as many cycles as the operand bit width to produce a result. This will, in turn, lead to an introduction to the multiplier design developed for this project. A self-timed sequential multiplier intended to be integrated into a synchronous environment. Making it a Globally Synchronous Locally Asynchronous (GSLA) sequential multiplier.

2.1 Single Cycle Multiplier

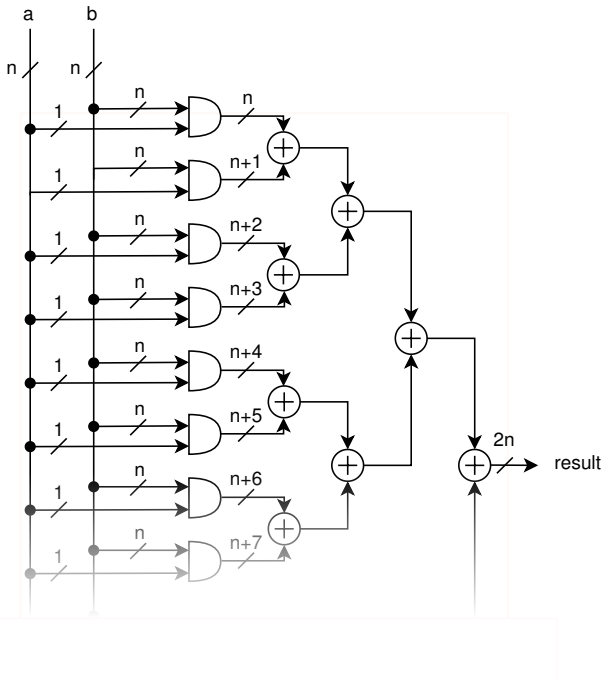


Figure 1: Single Cycle Multiplier

A single cycle multiplier can be implemented as an adder tree, performing the sum of partial products produced by AND'ing the bit of one operand with the other operand and then shifted appropriately. This type of single-cycle multiplier is illustrated in figure 1.

This type of multiplier is very expensive due to the high amount of adders, thus, it should only be used in a system where it is crucial that multiplication can be performed in a single

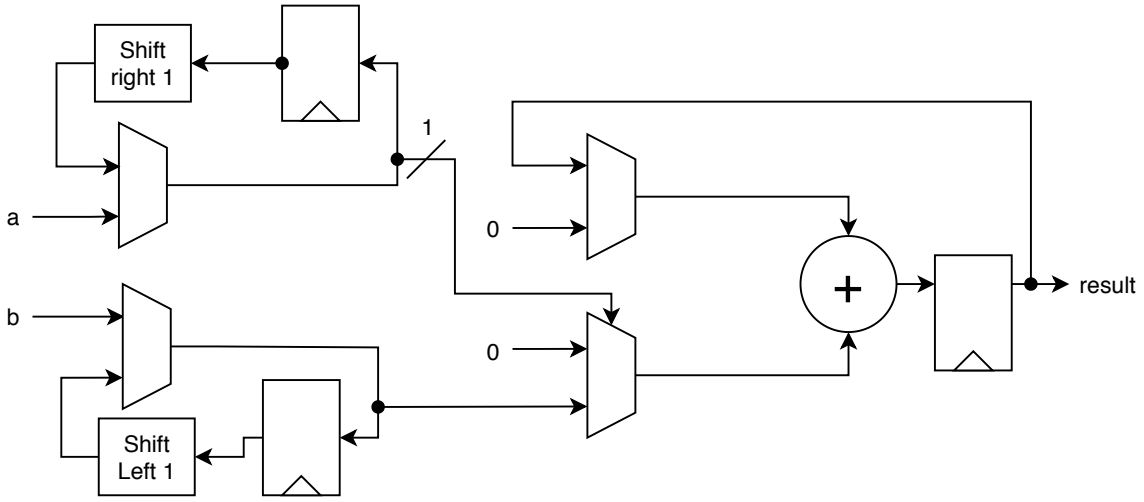


Figure 2: Principle of a synchronous multiplier. Control signals are not shown.

cycle. This multiplier design requires n ADD'ers and $\lceil log_2(n) \rceil - 1$ 4-2 Carry Save Adders (CSA) and a full adder. The delay of a single-cycle multiplier has a logarithmic propagation delay when using an adder-tree. If the system allows it a sequential multiplier should be implemented instead. This is described in the following section.

2.2 Sequential Multiplier

A sequential multiplier can be made from a ring using either a carry-propagate or carry-save adder. The benefit of a carry-save adder is that it takes constant time to do a calculation, whereas a carry-propagate adder has a propagation delay which is linear with the number of bits in the input, due to the carry that needs to be propagated. Both versions have a linear area consumption. The downside of using a carry-save adder is that the output is given in a carry-save format, which needs to be added at the end. An implementation would, therefore, be able to have multiple carry-save adders after each-other to reduce the number of cycles a multiplication takes, where the propagation delay will still be dominated by the final carry-propagate adder that needs to add the carry and the sum.

Figure 2 shows the principle behind the sequential multiplier. The sequential multiplier performs the same operation that you would do by hand. This method is illustrated in figure 3. You derive the partial products by copying and left shifting the operand b when the corresponding bit of a is one, otherwise, the partial product is zero. These partial products are then added to evaluate the product. A carry-propagate and a carry-save adder can both be used for this algorithm, with the only difference being the adder and the format of the output. The control signals are excluded, but the multiplexer for the top input should

a:				1	1	0	1	
b:				1	0	1	0	*
				1	0	1	0	
			0	0	0	0	0	+
		1	0	1	0	0	0	+
	1	0	1	0	0	0	0	+
1	0	0	0	0	0	1	0	

Figure 3: Multiplication by addition

choose itself when running and the 0-input otherwise, which resets the accumulation. The bottom input to the adder is a left-shifted version of the b-input, which has been left-shifted by the number of iterations performed. The multiplexer chooses the left-shifted b-value when the LSB of the right-shifted a-value is 1 and otherwise 0. Thereby, it is functionally equivalent to the single-cycle multiplier, with the a-input choosing whether a b-input should be added to the sum.

2.3 GSLA Sequential Multiplier

In this section, we will describe the asynchronous multiplier, designed in this project. The idea behind this project was to investigate if you could get the best of both worlds from the two synchronous multipliers introduced above, with an asynchronous implementation. That is, having a small footprint by doing multiplications by sequential additions and having single or few synchronous cycle multiplications through a self-timed multiplier.

The multiplier is designed using a phase-decoupled two-phase handshake protocol. The environment is to deliver multiplication operands to the multiplier along with a request on the handshake signals. This should start the sequential multiplication algorithm. When the algorithm has terminated the multiplier presents the result along with a request handshake signal at its output. The approach to designing this asynchronous multiplier has been to convert the sequential multiplier introduced in section 2.2 into an asynchronous equivalent.

The circuit schematic can be seen in figure 4. This is fairly big. To aid in understanding and readability we have employed two annotation measures. One being the colored dots. These indicate when these channels are active. A description of the conditions for which a channel is active we will return to later. The second annotation measure color-coded regions of the schematic. The purple region is annotated in- and outputs. The green regions are operand shift loops equivalent to the operand shift loops in figure 2. The blue region is the sequential addition loop similar to the adder loop in figure 2. Lastly, there is the red region in

the middle, which contains circuitry for controlling data-flow, to which we will return shortly.

The green regions contain the operand shift loops. These operand shift loops have the equivalent function to the shift loops in the synchronous counterpart. But in the asynchronous version, the first asynchronous design challenge arises. A simple handshake, as depicted in figure 5a, will not deadlock as tokens are passed between the two handshake latches, thereby no other tokens are introduced that could block a handshake. The operand rings, in figure 4, need new operands from the input of the multiplier when a new multiplication is to be performed. This means that first handshake latches need a token from two different token sources. The initial intuition would be to add a multiplexer in front of the first handshake latches as depicted in figure 5b. This simple implementation is not an option as this makes for a deadlock. When the first handshake consumes a new token the next handshake register cannot handshake with the first as this is now blocked. To solve this issue, we need to make the token held by the second register to disappear. This is achieved by having a demux and a sink on the feedback path of the ring. By sharing the selector signal with the multiplexer, we ensure that the feedback token is consumed when new operands are introduced to the rings, thereby avoiding deadlock. This is a structure that is found at multiple sites of the design.

The throughput of an asynchronous closed system is bottlenecked by the largest match delay in the system. In this multiplier, the longest match delay is through the functional CSA block, in the loop marked with a blue region. As this is the deepest combinatorial in the system, when not considering the n-input NOR-gate. If the remainder of the system is to handshake with this adder ring, this too will be a bottleneck by the rate which this ring can handshake. To speed up multiplication evaluation we would like to avoid this as much as possible. When the LSB of the shifted operand is 0, the adder should not add the shifted b operand. Instead of adding zero, we sink this token from the b operand loop that would otherwise have been handshaken with the adder loop. This is controlled by the control circuit in the red region.

The control circuit is all based on the a operand in the output handshake latch of the a operand loop. This control circuity has two conditions that data should be controlled based on. The first one being; is the multiplication done. We determine a multiplication to be done if all bits in the out handshake latch of the a operand loop is 0. The signal that indicates this is produced by the n-input NOR gate at the middle branch of the fork at the output of the operand loop of a. When the multiplication is done, the following operations are performed. The result in the adder ring should be steered to the output port. The adder ring should be reset by injecting a token with value zero in the feedback path. And

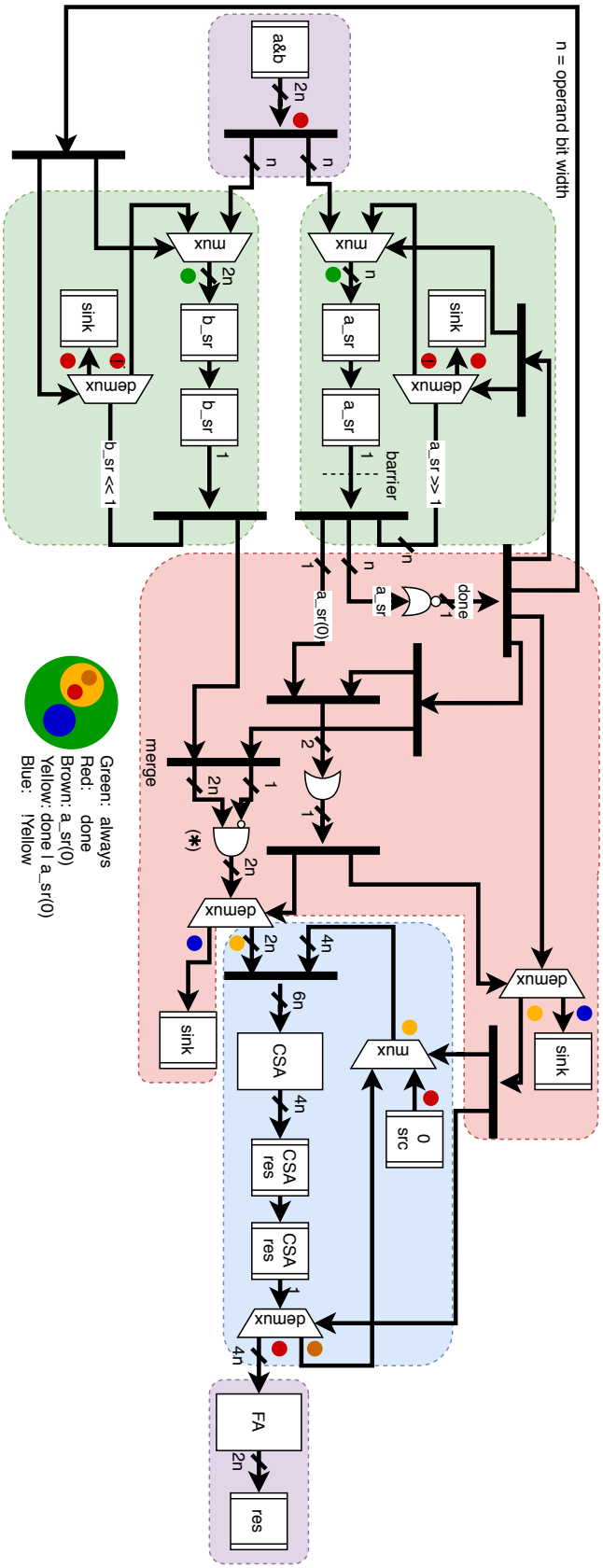
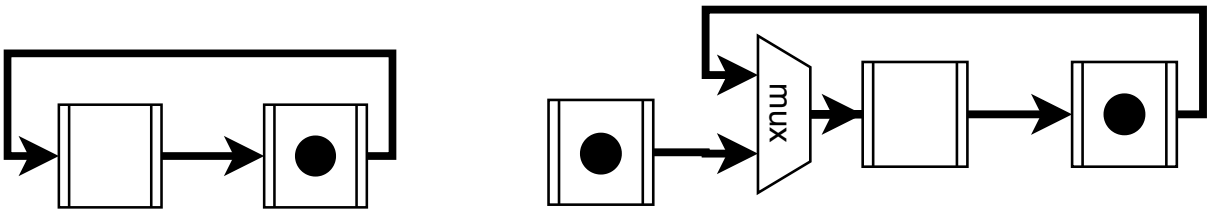


Figure 4: GSLA Sequential Multiplier. Component are initialized for to phase = 0, where not indicated to with one.



(a) Simple 2-phase token ring (b) 2-phase ring with input selection. Deadlocks

Figure 5: Simple rings. Illustrates deadlock issue of simple input selection

the circuit should accept a handshake from the input along with new operands for the next multiplication. The second condition for the control circuitry is whether the LSB of the shifted a operand is zero or not. If zero, we need to sink the shifted b operand for speed up, as mentioned above. The combination of these two control conditions makes for quite a bit of design overhead, while also making the schematic hard to follow. To aid this understanding we added the color-coded dots to figure 4. These dots indicate under what conditions the current channel has an active handshake. These conditions are described in a Venn style diagram in figure 4. Green channels are always active. Red are active when the a operand is all zeroes, which means we are done. Red with an exclamation mark is when we are not done. Brown is when the LSB of the shifted a is high. The existence of activity for these first four conditions with the above descriptions can be justified with a little thought. The two next is not immediately obvious, and exists as a consequence of having the two conditions for control flow. The yellow dots indicate activity when done or the LSB of a is high. When this condition is true, the adder loop has the sum and carry vector of the result in the first handshake latch. For the result to propagate to the output, the adder loop need to progress one step. To do this a token from the b operand ring needs to enter the adder ring, as a consequence of the merge of the input to the adder ring. To also reset the adder ring for the next multiplication, the value that comes with the last token from the b operand ring must be zero. This is the function of the AND gate marked with ($*$). This gate AND's all bits of the b operand with the inverted value of the done signal, to achieve just this.

3 Implementation

The Asynchronous multiplier design for this project was implemented in VHDL, using the 2-phase phase-decoupled click-based asynchronous components library developed in [2]. This is a library of VHDL component that ease prototyping of asynchronous circuits on FPGAs. These components generate a pulse when a handshake take place, which can be used for clocking registers. This asynchronous component can be implemented without using latches, which again makes FPGA implementations possible.

3.1 Barrier

An implementation specific design choice is the barrier seen at the output channel of the second handshake register of operand a in figure 4. The barrier prevents the handshake which this handshake register is initialized with from propagating further. If this barrier is not implemented, a handshake signal will propagate while reset is high, this would in turn prevent some components from clicking when reset is set low again and, thereby, deadlocking the system.

3.2 Debugging

Debugging was performed simulating the circuit and applying test signals using a test bench. By looking at the waveform from the test we were able to debug the system. Debugging, however, is a multi-step process. First, the system was simulated based on the VHDL description alone. Delays must be modelled when simulating asynchronous circuits as the synchronous clock, which usually separates logic steps are not present. If delays are not modelled, the simulator do not know about propagation delay, and the signals would not be stable at any point, thus you would not see logic steps performed by the circuit. Furthermore, simulators not using delta-time simulation would not necessarily be able to simulate the circuit at all. Here delays are modelled using VHDL timing keywords such as `transport` **signal** *after* **time**. This is a debugging of the functional description of the multiplier, that allowed us to find architectural bugs such as deadlocks, and the need for a barrier.

A successful functional simulation of the design did not make for a successful implementation on an FPGA. The reason is the functional implementation does not know the actual propagation delays that the system will experience once implemented. Therefore, the next step in debugging is doing a post synthesis timing simulation. This way you get a simulation of the design with the propagation delays that synthesis tool predicts the circuit will experience once implemented on the board. In this step match delays must be tuned to ensure that data are ready before the handshake signals reaches a destination handshake register. We used very conservative match delays, therefore, did not have issues with data not being ready.

Debugging can be very challenging for asynchronous circuits for two main reasons. A design error can propagate quite far down the circuity before coursing a deadlock. Waveforms is hard to navigate as there is no shared point in the time line where the full circuit synchronizes. This fact together that multiple conditions in different parts of the circuit can

contribute to a bug, it is then hard to track. One of the primary issues we had here was that optimizations were performed, such that the request inputs to forks were left open because the request was routed directly to all the recipients in the netlist. Therefore, it was not possible to go from component to component without checking the netlist.

3.3 Synthesis Optimization - Good and Bad

Most synthesis tools, including the one that comes with Vivado, is not geared for asynchronous circuits. As a result, some of the optimization performed can actually break the design. This was seen at multiple sites of the implementation. The most common breaking optimization we saw was handshake signals going around click components, typically when forks when dealing with forks. The internal signals of these components was marked with `dont_touch`, but the signals connecting components in the top-file, can also require `dont_touch` to be implemented exactly as described. Outputs of barriers also needs `dont_touch`.
We also saw the that the synthesis tool made some good optimizations. The Asynchronous click library that was used to write the VHDL had support for 1 to 2 forks only. This means that in order to describe the 1 to 4 fork in in figure 4 3 fork components was used. We could see from the net list of the synthesized circuit that the majority of the signals connecting these forks was gone, indicating that they were optimized away. The forks, however, still needed multiple levels of acknowledging.

4 Results

This section will go through the results gathered from the asynchronous multipliers implementation and compare their metrics. This is for an FPGA implementation, as this was what the library [2] used supported. A testbench was written for the different types of multipliers, where two linear feedback shift registers (LFSR) were used to create identical pseudo-random test patterns for the a and b inputs to the multipliers. The testbenches were modified for the different types of multipliers, single and multi-cycle synchronous and asynchronous, in order to get as fair a comparison as possible. It should, however, be noted that the asynchronous version has not been interfaced to a synchronous system, which would slightly increase its power consumption. However, this should only be a minor contributor.

4.1 Verification

To verify the system, the testbench was first run with 10 000 inputs generated by two different LFSRs where the testbench verified each result with a reference result computed by the testbench itself. After post-synthesis timing simulations were performed and verified, a bitstream was generated and the asynchronous multiplier was put onto the FPGA. Here, switches were used for a and b inputs and the buttons were used for input *req* and output *ack* signals, while LEDs were used for the corresponding output *req* and input *ack* signals and the result output. A number of different inputs were applied and the result was verified.

4.2 Area Consumption

Area consumption for the different designs was easily found but only performed for post-synthesis using Vivado. The reason being that the FPGA has a limited number of IO and can as such only have a limited number of inputs and outputs. And when using larger multipliers, the number of IOs were quickly surpassed.

LUT/FF	8	16	32	64	128
Direct	71/34	348/66	1274/177	5421/402	22090/1464
CSA-based	79/70	118/142	221/287	444/580	890/1163
Asynchronous	181/166	250/278	390/502	668/950	1225/1846

Table 1: Area consumption for different operand bitwidths given in LUT/FF.

We can see that all the multipliers has an exponential increase, when doubling the number of bits. It is here visible that the asynchronous version has a lot of overhead but the resource consumption closes in on the CSA-based version. The direct multiplication increases much faster than the other two.

4.3 Power Consumption

The power consumption was found by running a simulation of the synthesized design with annotated delays using the generated SDF file. This was done using Vivado Post-Synthesis Timing Simulation. In order to generate the Switching Activity Interchange Format (SAIF) file, which contains information about the switching of the internal signals, the simulation settings were changed to export this. This is the *xsim.simulate.saif*, which tells the file name of the SAIF file. *xsim.simulate.saif_all_signals* was also enabled, to get all the internal signals. When running the simulation, it needs to run the entire test in one run, as it closes the

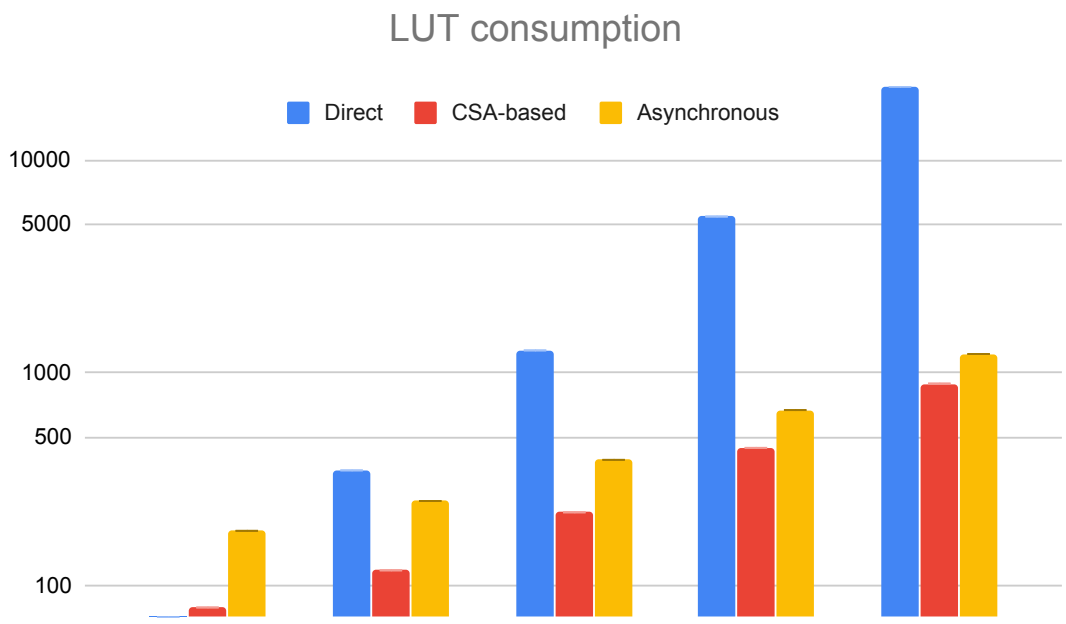


Figure 6: Total number of LUTs needed for 8, 16, 32, 64 and 128 bit multipliers.

SAIF file after simulating for the specified time. To overcome this, *xsim.simulate.runtime* was changed to *-all*, which means that it runs until it is stopped by the testbench. And in our case, this happens when *std.env.finish;* is executed.

The SAIF file is saved under *project/project.sim/sim_1/synth/timing/xsim* for a post-synthesis timing simulation. When clicking "Report Power", there is a tab called *Switching*. From here, the SAIF file can be chosen under *Simulation Settings*. After this, the power estimation will be much more precise, as it would otherwise use some pre-determined probabilities for signal switching.

The power consumption is given in static and dynamic power consumption. The static power consumption is how much power the FPGA consumes from just having its internal gates supplied with voltage. This comes from leakage and more. The dynamic power is what comes from switching activity and is normally the dominant contributor for larger systems with lots of switching. The dynamic power is then divided into subgroups of signals, logic, clock, and IO. Signals are the wires, logic is the components such as LUTs and clock is the power that is consumed due to the clock signal, which has to be routed to many positions on the board. IO is especially for the output pins, which have a certain load but defaults to 0. This can be modified to fit the values of the board. In our case, we have noted the total dynamic power and IO power, which is a part of the total dynamic power. This is to be able the exclude the switching of buffers on the inputs and outputs, which would not be there in

an actual system running the multipliers.

Table 2 shows the wattage for the systems. These values were only obtained for up to 32 bits, due to the long test runtime for larger multiplications (¿60min for a 64-bit single-cycle multiplier). It should also be mentioned that Vivado only reports down to mW consumption and has about 20% error. The static power consumption for all the systems was 91 mW and thereby the primary power consumer. For an actual system, the static power consumption might be a smaller percentage but would still increase, as more parts of the chip are enabled. It should be noted that the asynchronous version has most of its power consumed in the IO part, as it uses 60%, 70%, and 83% of its dynamic power here. This is way more compared to the synchronous version, which is the most comparable.

dyn.-/IO power [mW]	8	16	32
Direct	5/2	9/4	46/30
CSA-based	9/2	12/1	14/1
Asynchronous	5/3	10/7	18/15

Table 2: Power consumption for dynamic power given in mW with dynamic/IO, where dynamic is the total dynamic power and IO is the part of this that is used for IO ports.

It is important to mention that all these figures are given in watt and therefore, to get an idea of the power consumption per operation, we would need to multiply by the number of seconds it took to run all the test vectors and then divide by the number of test vectors. This would give the average energy per multiplication in joule, as watt is joule/s.

4.4 Timing

The timing of the system is given in two forms, one for the propagation delay which is valid for clocked designs and one for the test runtime. As the asynchronous version was not interfaced into a synchronous circuit, this does not contain a propagation delay. It is important to note that the CSA-based synchronous multiplier contains a full-adder after a register, which is the critical path. As the CSA has constant delay, this will only increase when routing becomes more difficult, whereas the adder needs a longer and longer carry chain. This could be overcome with multi-cycle adders.

Table 4 shows the simulated time required to pass the testbench for the different systems. As we can see, the direct version takes the same time no matter the bit width, as the testbench was always clocked with a period of 20 ns. This has been taken into account when calculating

Prop. delay[ns]	8	16	32	64	128
Direct	5.9	7.8	12.0	15.5	15.7
CSA-based	2.7	3.0	3.8	5.9	9.6

Table 3: Propagation delay for different bit widths given in ns. The asynchronous version does not have a clock and hence no propagation delay is noted.

the power consumption later on. Furthermore, we can see that if the system is clocked at 20 ns, the asynchronous non-optimized multiplier is almost as fast as the synchronous version. The asynchronous version takes 50% longer to compute on average.

Runtime[ns]	8	16	32
Direct	800000	800000	800000
CSA-based	1419290	3001590	6193230
Asynchronous	2276649	4519090	9095750

Table 4: Runtime for different bit widths given in ns. Due to a quadratic increase in test-time, only up to 32 bits have been tested.

4.5 Energy Per Operation

Finally, we can have the actual energy per operation calculated for the different versions. As all the testbenches were run with a clock-period of 20 ns, this needs to be scaled to match the synthesized clock frequency. To scale, we divide the clock period synthesized by 20 ns, which gives a scalar that needs to be applied. We then multiply this by the runtime in the simulation, which gives the total runtime at a given frequency. This is then multiplied by the dynamic watt consumption, which gives the total amount of energy consumed. This is then divided by the number of test vectors, which gives the average energy spent per multiplication. Table 5 states this value in picojoule.

We can see that the direct multiplication is very efficient for small versions but rapidly increase with bit width. This makes sense as the number of LUTs needed increases exponentially with the number of bits. Both the synchronous and asynchronous versions both also increase exponentially, however slower than the direct version. This is because the width of the inputs increase and therefore the number of cycles per multiplication.

Energy/Operation[pJ]	8	16	32
Direct	118	280.8	2208
CSA-based	172	540	1647
Asynchronous	1138	4519	16372

Table 5: Energy per operation for different bitwidths, given in picojoule.

5 Discussion

5.1 Future Work

There are multiple areas that could be expanded in the system. One is an interface into a synchronous system. This is tool-dependent, as it requires information on the timing for each run. If the system has been timed to take x seconds before it has a stable output, we can know exactly how many cycles we need to count using a synchronous counter, which can then work as an enable for the output signal. The enable signal into the system can be generated from the valid signal into the multiplier. This, however, needs to have a two-phase crossing, such that when valid goes low, the phase into the asynchronous multiplier stays the same.

The current system takes almost the same time for a word that consists all 0s except for the MSB being 1 as the a-operand as when operand a is all ones. This is because it takes to, close to, the same time process a 0 as processing a 1. It does, however, run faster the further to the right the leading one in the a- operand is. Therefore, it is possible to make a leading-zeroes detector, that counts the number of leading zeroes in the a-input. The system can then have multiple timings, which is controlled by how high the counter should count. Given an n-bit input, it takes approximately $(n - leading\_zeroes)/n$ to compute a multiplication. This can be divided into sections, such that it matches an integer number of clock cycles.

Another implementation that is currently missing is that the system only has a simple adder in the end, that adds the carry and sum for the output. This could also be converted to an asynchronous version or the delay of that should be included in the final timing. Alternatively, the multiplier might just give the result in a carry-save format, which can either be used directly in the synchronous system or a synchronous adder can be used afterwards.

We see high power consumption reading of the implementation. Some of this power is dynamic power consumed by the CSA. By moving the location of CSA we could most likely reduce the switching power consumed by this block. With the location it has now, the input will switch even when we keep the adder ring still. This is because the inputs are connected

directly to shifted b operands. Furthermore, the inputs from the feedback path of the adder ring will not arrive at the same time as the inputs from the b operand, as this is a asynchronous system. This further makes for switching of the CSA. By moving it in between the two handshake registers of the adder ring, there would be no input switching when the ring is held still. And all inputs would change at approximately the same time, as they are fed by the same handshake register.

As it has been touched upon, the throughput of the system is bottle-necked by the matched delay of the CSA functional block. This limits the rate of handshakes in the adder ring. If area and power budgets allows it, this fact should be taken advantage of by having another CSA element between the two handshake registers in this ring, to perform the addition of two partial products in parallel. The rate of tokens would remain unchanged in this ring, but half the number of handshakes are needed to produce a result. This would introduce further overhead for the rest of the multiplier as well. For instance, the adder ring can only be still if the two LSB of operand a is 0. If the current design had been set up to add 0 when the LSB of a is 0, the latency would be cut in half with another CSA element. For random operands the ring will on average be still for 50% of the steps. For 100 bit input this means 50 steps on average. When the system have to consider the two LSB the chance is 25% of them both being 0. With two adders the worst case needs 50 steps and on average $0.75 \cdot 50 = 37.5$. This means we can estimate an average decrease of latency by $(50 - 37.5)/50 = 25\%$ by implementing another CSA.

We used the combination of a demux and a sink in several sites of our design to consume a token on a given condition. We imagine that this is a combination of components that are used often. Therefore, there is an argument for investigating if it could be optimized by combining these two components to one structure. If nothing else, then add such a component as a conditional sink to the library for convenience.

The asynchronous click library [2] that we used, did have some optimized merged components. For instance register and fork. We did not use these, but this might had made for slightly better performance. This too is a consideration for future work.

5.2 Hardware results

When obtaining the results, we used post-synthesis results to evaluate from, due to limited amounts of IO on an FPGA. An alternative solution would be to make a hardware tester with the synthesizable LFSR units that could apply inputs to the different multipliers with the same power consumption for the different implementations. A problem with this would

be optimizations performed at synthesis, where outputs are needed in order to actually generate hardware.

5.3 ASIC Implementation

Another interesting aspect would be to look at an ASIC implementation and the timings gotten from this. This is especially interesting, as an ASIC has support for simpler gates, which the system is designed around. An FPGA implementation using only the same type of LUTs and registers will give the system some barriers. For instance, it does not make much difference if a bit is dependent on one or 6 bits for its output. That is, an inverter roughly takes the same time as a 5-bit XOR gate in an FPGA implementation. Another aspect is the wiring delay, which is also quite constant in an FPGA, as it has to use the routing matrix. An ASIC has more freedom in modifying parts of the system and can use high speed and low-speed versions of gates for even higher granularity of area, speed, and power consumption.

An issue with the ASIC implementation is to control the tools, which are inherently made for synchronous designs. The Synopsys compiler tool takes several clock ports and then starts its timing based on registers from input to output using these clocks. Whether it can understand an asynchronous loop is yet to be investigated. And even if it can compile this, then it is probably not able to get the timing of the system, as it does not know when the loop has a valid output. Therefore, making an ASIC implementation would probably require giving the tool information about multi-cycle paths. And afterward, a simulation should be done with annotated delays, as would be present in a Standard Delay Format (SDF) file.

5.4 Easier Development

As an asynchronous system needs many connections, an easier way of describing hardware would be very beneficial. One way would be to have a GUI program that allows drag and drop and connections between components. Another approach would be to use another faster language, such as Chisel, where connections between ports of components can be made directly. This makes development a lot faster, as connections only have to be changed in one place. A Chisel description would generate Verilog code, which can also be synthesized. The library is written in VHDL, but this is not an issue as these can be black-boxed into the Chisel design and then use a mixed-language synthesis in Vivado, which is enabled by default.

5.5 Requirements

To assess the asynchronous model, we can now look into the three requirements as stated in the beginning.

- Be able to perform many iterations per slow clock cycle -¿
  This was not validated, but we did see that it is possible to run many iterations in a specified amount of time so there is no issue in making a working solution with a register that enables after n cycles. The exact time it takes for a cycle was not heavily investigated. The library was not made for speed either, so to assess this, the system should be timed and the delay elements should be matched to the delay.

- Consume less area than a synchronous single-cycle multiplier -¿
  We got only a slight increase in resource consumption compared to a similar synchronous version and a much lower consumption compared to a direct multiplier. The asynchronous multiplier has quite a bit of overhead for all the handshaking, but this gets amortized for larger bit-versions. An ASIC implementation will probably make this overhead smaller compared to the synchronous version.

- Have a comparable power consumption for a diverse set of test vectors The energy per operation is a factor 10 larger than the synchronous version. This, however, has multiple reasons and can be solved in different ways. The synchronous version has only a very small part of its power consumed in the IO, whereas the asynchronous version has 60%, 70%, and 83% of its dynamic power consumed in the IO. Therefore, a latch or register on the output of the multiplier would be beneficial, which would drastically reduce the amount of switching. And as the output buffers probably require quite a bit of power to drive compared to internal gates, this seems like a huge power saver. And as discussed moving the CSA component to reduce input switching would also better the power consumption.

From these three requirements, we do not have anything that "ruins" the idea of using an asynchronous multiplier in a synchronous design. The higher power consumption can probably be fixed. The design was focused on slowly clocked systems with certain faster components. And as the testbench was clocked at 50 MHz, we need to slow this down by a factor of 10, and then the asynchronous version would be faster than the synchronous version. So with optimizations, this actually seems like a solution that might be beneficial for slowly clocked systems. It also has the, potentially huge, benefit of smoothing out the

power consumption to ease the requirements for the power supply, instead of having a lot of switching right on the clock edge. However, an ASIC implementation still needs to be performed to verify these.

6 Conclusion

This report looked into asynchronous multipliers compared to direct and synchronous versions. It was found that the area is almost the same as a synchronous version, though with an initial overhead for handshaking. It was found that the timings of the non-optimized system are comparable to a synchronous version clocked at 50 MHz. And as this design is for slowly clocked systems, at less than 10 MHz, this is also very doable. The power consumption is the main issue, but analysis indicates that the power consumption can be greatly reduced with simple adjustments. All in all, the asynchronous multiplier seems like a possibility for use in real devices that needs small and relatively fast multipliers.

The project repository can be found at `https://github.com/nothinn/async_multiplier`.

References

[1] J. F. Sparsø, *Introduction to Asynchronous Circuit Design*, 2020.

[2] A. Mardari, Z. Jelcicova, and J. Sparsø, "Design and fpga-implementation of asynchronous circuits using two-phase handshaking," *Proceedings of 2019 25th Ieee International Symposium on Asynchronous Circuits and Systems*, vol. 2019-, pp. 9–18, 2019.

7 Appendix

7.1 Multiplier_async.vhd

```
1  --
       ↪ ----------------------------------------------------------------------------------
       ↪
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date: 03/30/2020 12:11:34 PM
6  -- Design Name:
7  -- Module Name: Asynchrounous multiplier - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 --
       ↪ ----------------------------------------------------------------------------------
       ↪
20
21 -- name              initiated
22 -- aclick0           yes
23 -- aclick1           yes
24 -- bclick0           yes
25 -- bclick1           yes
26 -- afork0            yes
27 -- afork1            yes
28 -- bfork0            yes
29 -- nor                yes
30 -- amux               yes
31 -- bmux               yes
32 -- csademuxin         yes
33 -- noaddsink          yes
34 -- CSA                yes
35 -- csaclick0          yes
36 -- csaclick1          yes
37 -- csademuxout        yes
```

```
-- csajoin           yes
-- csamux            yes, needs 0-input verification
-- donefork0         yes
-- donefork1         yes
-- donefork2         yes
-- donefork3         yes
-- CSAresetsource    yes



library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity multiplier_async is
    Generic(
        BITWIDTH : integer := 128
    );
    Port (
        rst : in std_logic;
    --Handshake ports
        --Ingoing channel
        req_in : in std_logic;
        ack_out: out std_logic;

        --Outgoing channel
        req_out: out std_logic;
        ack_in : in std_logic;


        start : in std_logic;

        --Data
        a_in : in std_logic_vector(BITWIDTH - 1 downto 0);
        b_in : in std_logic_vector(BITWIDTH - 1 downto 0);
        result_out : out std_logic_vector(BITWIDTH *2 - 1 downto 0)
    );
```

```
83
84  end multiplier_async;
85
86  architecture Behavior of multiplier_async is
87
88      --CONSTANT BITWIDTH : integer := 32;
89
90
91      --DELAY SIGNALS
92      CONSTANT SINK_DELAY : time := 1ns;
93      CONSTANT CSA_DELAY : time := 30ns;
94      CONSTANT SAFETY_MARGIN : real := 1.5;
95      CONSTANT SOURCE_DELAY : time := 1ns;
96
97
98
99
100
101     -- Signal and component declarations
102     --fw: forward, bw: backward
103
104     signal b_pad : std_logic_vector(BITWIDTH - 1 downto 0);
105
106     signal in_fork_fwt_req : std_logic;
107     signal in_fork_fwt_ack : std_logic;
108     signal in_fork_fwb_req : std_logic;
109     signal in_fork_fwb_ack : std_logic;
110
111     signal a_mux_fw_req : std_logic;
112     signal a_mux_fw_ack : std_logic;
113     signal a_mux_fw_data : std_logic_vector(BITWIDTH - 1 downto 0);
114     signal a_mux_fw_data_in : std_logic_vector(BITWIDTH - 1 downto 0);
115
116
117     signal a_click_0_fw_req : std_logic;
118     signal a_click_0_fw_ack : std_logic;
119     signal a_click_0_fw_data : std_logic_vector(BITWIDTH - 1 downto 0);
120
121     signal a_click_1_fw_req : std_logic;
122     signal a_click_1_fw_ack : std_logic;
123     signal a_click_1_fw_data : std_logic_vector(BITWIDTH - 1 downto 0);
124
125     signal a_fork_0_fwt_req : std_logic;
126     signal a_fork_0_fwt_ack : std_logic;
127     signal a_fork_0_fwb_req : std_logic;
```

```
128     signal a_fork_0_fwb_ack : std_logic;
129
130     signal a_fork_2_fwt_req : std_logic;
131     signal a_fork_2_fwt_ack : std_logic;
132     signal a_fork_2_fwb_req : std_logic;
133     signal a_fork_2_fwb_ack : std_logic;
134
135     signal a_demux_fwb_req : std_logic;
136     signal a_demux_fwb_ack : std_logic;
137     signal a_demux_fwb_data : std_logic_vector(BITWIDTH - 1 downto 0);
138     signal a_demux_fwc_req : std_logic;
139     signal a_demux_fwc_ack : std_logic;
140     signal a_demux_fwc_data : std_logic_vector(BITWIDTH - 1 downto 0);
141     signal a_click_1_fw_data_shift :  std_logic_vector(BITWIDTH - 1 downto 0);
142
143
144     signal b_demux_fwb_req : std_logic;
145     signal b_demux_fwb_ack : std_logic;
146     signal b_demux_fwb_data : std_logic_vector(BITWIDTH*2 - 1 downto 0);
147     signal b_demux_fwc_req : std_logic;
148     signal b_demux_fwc_ack : std_logic;
149     signal b_demux_fwc_data : std_logic_vector(BITWIDTH*2 - 1 downto 0);
150     signal b_click_1_fw_data_shift : std_logic_vector(BITWIDTH*2 - 1 downto 0)
    ↪ ;
151
152     signal a_fork_1_fwt_req : std_logic;
153     signal a_fork_1_fwt_ack : std_logic;
154     signal a_fork_1_fwb_req : std_logic;
155     signal a_fork_1_fwb_ack : std_logic;
156
157     signal nor_fw_req : std_logic;
158     signal nor_fw_ack : std_logic;
159     signal nor_fw_data : std_logic;
160
161     signal done_fork_0_fwt_req : std_logic;
162     signal done_fork_0_fwt_ack : std_logic;
163     signal done_fork_0_fwb_req : std_logic;
164     signal done_fork_0_fwb_ack : std_logic;
165
166     signal done_demux_fwb_req : std_logic;
167     signal done_demux_fwb_ack : std_logic;
168     signal done_demux_fwb_data : std_logic_vector(0 downto 0);
169     signal done_demux_fwc_req : std_logic;
170     signal done_demux_fwc_ack : std_logic;
171     signal done_demux_fwc_data : std_logic_vector(0 downto 0);
```

```
172
173     signal done_demux_selector : std_logic;
174
175     signal done_fork_1_fwt_req : std_logic;
176     signal done_fork_1_fwt_ack : std_logic;
177     signal done_fork_1_fwb_req : std_logic;
178     signal done_fork_1_fwb_ack : std_logic;
179
180     signal done_fork_2_fwt_req : std_logic;
181     signal done_fork_2_fwt_ack : std_logic;
182     signal done_fork_2_fwb_req : std_logic;
183     signal done_fork_2_fwb_ack : std_logic;
184
185     signal done_fork_3_fwt_req : std_logic;
186     signal done_fork_3_fwt_ack : std_logic;
187     signal done_fork_3_fwb_req : std_logic;
188     signal done_fork_3_fwb_ack : std_logic;
189
190     signal done_fork_4_fwt_req : std_logic;
191     signal done_fork_4_fwt_ack : std_logic;
192     signal done_fork_4_fwb_req : std_logic;
193     signal done_fork_4_fwb_ack : std_logic;
194
195     signal done_fork_5_fwt_req : std_logic;
196     signal done_fork_5_fwt_ack : std_logic;
197     signal done_fork_5_fwb_req : std_logic;
198     signal done_fork_5_fwb_ack : std_logic;
199
200     signal done_fork_6_fwt_req : std_logic;
201     signal done_fork_6_fwt_ack : std_logic;
202     signal done_fork_6_fwb_req : std_logic;
203     signal done_fork_6_fwb_ack : std_logic;
204
205     signal done_fork_7_fwt_req : std_logic;
206     signal done_fork_7_fwt_ack : std_logic;
207     signal done_fork_7_fwb_req : std_logic;
208     signal done_fork_7_fwb_ack : std_logic;
209
210     signal demux_sel_delay_req : std_logic;
211     signal csa_demux_in_req : std_logic;
212     signal a_join_fw_req : std_logic;
213     signal a_join_fw_ack : std_logic;
214
215     signal b_join_fw_req : std_logic;
216     signal b_join_fw_ack : std_logic;
```

```
217
218     signal b_mux_fw_req : std_logic;
219     signal b_mux_fw_ack : std_logic;
220     signal b_mux_fw_data : std_logic_vector(BITWIDTH*2 - 1 downto 0);
221     signal b_mux_fw_data_inA : std_logic_vector(BITWIDTH*2 - 1 downto 0);
222     signal b_mux_fw_data_inB : std_logic_vector(BITWIDTH*2 - 1 downto 0);
223
224     signal b_click_0_fw_req : std_logic;
225     signal b_click_0_fw_ack : std_logic;
226     signal b_click_0_fw_data : std_logic_vector(BITWIDTH*2 - 1 downto 0);
227
228     signal b_click_1_fw_req : std_logic;
229     signal b_click_1_fw_ack : std_logic;
230     signal b_click_1_fw_data : std_logic_vector(BITWIDTH*2 - 1 downto 0);
231
232     signal b_fork_0_fwt_req : std_logic;
233     signal b_fork_0_fwt_ack : std_logic;
234     signal b_fork_0_fwb_req : std_logic;
235     signal b_fork_0_fwb_ack : std_logic;
236
237     signal csa_demux_fwb_req : std_logic;
238     signal csa_demux_fwb_ack : std_logic;
239     signal csa_demux_fwb_data : std_logic_vector(BITWIDTH*2 - 1 downto 0);
240     signal csa_demux_fwc_req : std_logic;
241     signal csa_demux_fwc_ack : std_logic;
242     signal csa_demux_fwc_data : std_logic_vector(BITWIDTH*2 - 1 downto 0);
243
244     signal CSA_demux_in_selector : std_logic;
245     signal CSA_demux_in_data : std_logic_vector(BITWIDTH*2 - 1 downto 0);
246
247
248     signal CSA_join_fw_req : std_logic;
249     signal CSA_join_fw_ack : std_logic;
250
251     signal CSA_join_fw_delayed_req : std_logic;
252     signal CSA_join_fw_delayed_ack : std_logic;
253
254     signal CSA_carry : std_logic_vector(BITWIDTH*2-1 downto 0);
255     signal CSA_sum : std_logic_vector(BITWIDTH*2-1 downto 0);
256
257 --    signal CSA_click_0_fw_ack : std_logic;
258 --    signal CSA_click_0_fw_req : std_logic;
259 --    signal CSA_click_0_fw_data : std_logic_vector(BITWIDTH*2 - 1 downto 0);
260
261
```

```
262     signal CSA_click_0_fw_req : std_logic;
263     signal CSA_click_0_fw_ack : std_logic;
264     signal CSA_click_0_fw_data : std_logic_vector(BITWIDTH*2*2-1 downto 0); --
    ↪ *2*2 due to carry and sum
265     signal CSA_click_0_data_in : std_logic_vector(BITWIDTH*2*2-1 downto 0);
266
267     signal CSA_click_1_fw_req : std_logic;
268     signal CSA_click_1_fw_ack : std_logic;
269     signal CSA_click_1_fw_data : std_logic_vector(BITWIDTH*2*2-1 downto 0); --
    ↪ *2*2 due to carry and sum
270
271
272     signal reset_barrier_a_click1 : std_logic;
273     signal reset_barrier_CSA_click1 : std_logic;
274
275     attribute dont_touch : string;
276     attribute dont_touch of reset_barrier_a_click1 : signal is "true";
277     attribute dont_touch of reset_barrier_CSA_click1 : signal is "true";
278
279
280     signal csa_demux_out_fwb_req : std_logic;
281     signal csa_demux_out_fwb_ack : std_logic;
282     signal csa_demux_out_fwb_data : std_logic_vector(BITWIDTH*2*2 - 1 downto
    ↪ 0);
283     signal csa_demux_out_fwc_req : std_logic;
284     signal csa_demux_out_fwc_ack : std_logic;
285     signal csa_demux_out_fwc_data : std_logic_vector(BITWIDTH*2*2 - 1 downto
    ↪ 0);
286
287     signal csa_reset_src_req : std_logic;
288     signal csa_reset_src_ack : std_logic;
289
290     signal csa_mux_fw_req : std_logic;
291     signal csa_mux_fw_ack : std_logic;
292     signal csa_mux_fw_data : std_logic_vector(BITWIDTH*2*2 - 1 downto 0);
293
294     signal result : std_logic_vector(BITWIDTH*2 - 1 downto 0);
295
296 begin
297
298
299     --CSA
300     b_pad <= (others => '0');
301     --instantiate a operand click elements
302     in_fork : entity work.fork
```

```
303         generic map(
304           PHASE_INIT => '0'  -- set the phase to the same as the click element
    ↪  driving it
305         )
306         port map(
307          rst => rst,
308          -- Input channel
309          inA_req => req_in,
310          inA_ack => ack_out,
311          -- Output channel 1
312          outB_req => in_fork_fwt_req,
313          outB_ack => in_fork_fwt_ack,
314          -- Output channel 2
315          outC_req => in_fork_fwb_req,
316          outC_ack => in_fork_fwb_ack
317         );
318
319
320     a_click_0 : entity work.decoupled_hs_reg
321         generic map(
322             DATA_WIDTH => BITWIDTH,
323             VALUE => 0
324         )
325         port map(
326             rst => rst,
327             -- Input channel
328             in_ack => a_mux_fw_ack,
329             in_req => a_mux_fw_req,
330             in_data =>  a_mux_fw_data,
331             -- Output channel
332             out_req => a_click_0_fw_req,
333             out_data => a_click_0_fw_data,
334             out_ack => a_click_0_fw_ack
335         );
336
337     a_click_1 : entity work.decoupled_hs_reg
338         generic map(
339             DATA_WIDTH => BITWIDTH,
340             VALUE => 0,
341             PHASE_INIT_IN => '0',
342             PHASE_INIT_OUT => '1'
343         )
344         port map(
345             rst => rst,
346             -- Input channel
```

```
347             in_ack => a_click_0_fw_ack,
348             in_req => a_click_0_fw_req,
349             in_data =>  a_click_0_fw_data,
350             -- Output channel
351             out_req => a_click_1_fw_req,
352             out_data => a_click_1_fw_data,
353             out_ack => a_click_1_fw_ack
354         );
355
356
357     reset_barrier_a_click1 <= a_click_1_fw_req and start;
358
359     a_fork_0 : entity work.fork
360         generic map(
361           PHASE_INIT => '0'  -- set the phase to the same as the click element
    ↪  driving it
362         )
363         port map(
364          rst => rst,
365          -- Input channel
366          inA_req => reset_barrier_a_click1,
367          inA_ack => a_click_1_fw_ack,
368          -- Output channel 1
369          outB_req => a_fork_0_fwt_req,
370          outB_ack => a_fork_0_fwt_ack,
371          -- Output channel 2
372          outC_req => a_fork_0_fwb_req,
373          outC_ack => a_fork_0_fwb_ack
374         );
375
376
377     demux_sel_delay : entity work.delay_element
378         generic map(
379             size => 1
380         )
381         port map(
382             d => a_join_fw_req,
383             z => demux_sel_delay_req
384         );
385     done_demux_selector <= a_click_1_fw_data(0) or nor_fw_data;
386
387
388     done_demux : entity work.demux
389         generic map(
390             DATA_WIDTH    => 1,
```

```
391             PHASE_INIT_A => '0', --TODO consider INIT
392             PHASE_INIT_B => '0',
393             PHASE_INIT_C => '0'
394         )
395         port map(
396             rst => rst,
397             -- Input port
398             inA_req => done_fork_6_fwt_req,
399             inA_data => (others => '0'),
400             inA_ack => done_fork_6_fwt_ack,
401             -- Select port
402             inSel_req => a_fork_2_fwt_req,
403             inSel_ack => a_fork_2_fwt_ack,
404             selector => done_demux_selector,
405             -- Output channel 1
406             outB_req => done_demux_fwb_req,
407             outB_data => done_demux_fwb_data,
408             outB_ack => done_demux_fwb_ack,
409             -- Output channel 2
410             outC_req => done_demux_fwc_req,  --
411             outC_data => done_demux_fwc_data,
412             outC_ack => done_demux_fwc_ack
413             );
414
415     done_DEMUX_SINK : entity work.sink
416         generic map(
417             BITWIDTH => 1,
418             sink_delay => sink_delay
419         )
420         port map(
421             req_in  => done_demux_fwc_req,
422             ack_out => done_demux_fwc_ack,
423             data_in => done_demux_fwc_data
424         );
425
426     a_click_1_fw_data_shift <= '0' & a_click_1_fw_data(BITWIDTH-1 downto 1);
427
428     a_demux : entity work.demux
429         generic map(
430             DATA_WIDTH   => BITWIDTH,
431             PHASE_INIT_A => '0', --TODO consider INIT
432             PHASE_INIT_B => '0',
433             PHASE_INIT_C => '0'
434         )
435         port map(
```

```
436             rst => rst,
437             -- Input port
438             inA_req => a_fork_0_fwt_req,
439             inA_data => a_click_1_fw_data_shift,
440             inA_ack => a_fork_0_fwt_ack,
441             -- Select port
442             inSel_req => done_fork_4_fwb_req,
443             inSel_ack => done_fork_4_fwb_ack,
444             selector => nor_fw_data,
445             -- Output channel 1
446             outB_req => a_demux_fwb_req,    --sink
447             outB_data => a_demux_fwb_data,
448             outB_ack => a_demux_fwb_ack,
449             -- Output channel 2
450             outC_req => a_demux_fwc_req,  --amux
451             outC_data => a_demux_fwc_data,
452             outC_ack => a_demux_fwc_ack
453             );
454
455
456             a_new_sink : entity work.sink
457             generic map(
458                 BITWIDTH => BITWIDTH,
459                 sink_delay => sink_delay
460             )
461             port map(
462                 req_in  => a_demux_fwb_req,
463                 ack_out => a_demux_fwb_ack,
464                 data_in => a_demux_fwb_data
465             );
466
467     a_fork_1 : entity work.fork
468         generic map(
469           PHASE_INIT => '0'
470         )
471         port map(
472          rst => rst,
473          -- Input channel
474          inA_req => a_fork_0_fwb_req,
475          inA_ack => a_fork_0_fwb_ack,
476          -- Output channel 1
477          outB_req => a_fork_1_fwt_req,
478          outB_ack => a_fork_1_fwt_ack,
479          -- Output channel 2
480          outC_req => a_fork_1_fwb_req,
```

```
481         outC_ack => a_fork_1_fwb_ack
482       );
483
484   a_fork_2 : entity work.fork
485       generic map(
486         PHASE_INIT => '0'
487       )
488       port map(
489        rst => rst,
490        -- Input channel
491        inA_req => demux_sel_delay_req,
492        inA_ack => a_join_fw_ack,
493        -- Output channel 1
494        outB_req => a_fork_2_fwt_req,
495        outB_ack => a_fork_2_fwt_ack,
496        -- Output channel 2
497        outC_req => a_fork_2_fwb_req,
498        outC_ack => a_fork_2_fwb_ack
499       );
500
501
502       nor_gate1 : entity work.NOR_gate
503           Generic map(
504               BITWIDTH => BITWIDTH,
505               NOR_DELAY => 2 ps --TODO decide this time
506           )
507           Port map(
508               in_req => a_fork_1_fwt_req,
509               in_ack => a_fork_1_fwt_ack,
510               in_bus => a_click_1_fw_data,
511               -- Output channel
512               out_req => nor_fw_req,
513               out_ack => nor_fw_ack,
514               result => nor_fw_data
515           );
516
517   done_fork_0 : entity work.fork
518           generic map(
519             PHASE_INIT => '0'
520           )
521           port map(
522            rst => rst,
523            -- Input channel
524            inA_req => nor_fw_req,
525            inA_ack => nor_fw_ack,
```

```
526             -- Output channel 1
527             outB_req => done_fork_0_fwt_req,
528             outB_ack => done_fork_0_fwt_ack,
529             -- Output channel 2
530             outC_req => done_fork_0_fwb_req,
531             outC_ack => done_fork_0_fwb_ack
532           );
533
534   done_fork_1 : entity work.fork
535           generic map(
536             PHASE_INIT => '0'
537           )
538           port map(
539             rst => rst,
540             -- Input channel
541             inA_req => done_demux_fwb_req,
542             inA_ack => done_demux_fwb_ack,
543             -- Output channel 1
544             outB_req => done_fork_1_fwt_req,
545             outB_ack => done_fork_1_fwt_ack,
546             -- Output channel 2
547             outC_req => done_fork_1_fwb_req,
548             outC_ack => done_fork_1_fwb_ack
549           );
550
551
552   done_fork_3 : entity work.fork
553           generic map(
554             PHASE_INIT => '0'
555           )
556           port map(
557             rst => rst,
558             -- Input channel
559             inA_req => done_fork_0_fwt_req,
560             inA_ack => done_fork_0_fwt_ack,
561             -- Output channel 1
562             outB_req => done_fork_3_fwt_req,
563             outB_ack => done_fork_3_fwt_ack,
564             -- Output channel 2
565             outC_req => done_fork_3_fwb_req,
566             outC_ack => done_fork_3_fwb_ack
567           );
568
569   done_fork_4 : entity work.fork
570           generic map(
```

```
571             PHASE_INIT => '0'
572           )
573           port map(
574            rst => rst,
575            -- Input channel
576            inA_req => done_fork_3_fwb_req,
577            inA_ack => done_fork_3_fwb_ack,
578            -- Output channel 1
579            outB_req => done_fork_4_fwt_req,
580            outB_ack => done_fork_4_fwt_ack,
581            -- Output channel 2
582            outC_req => done_fork_4_fwb_req,
583            outC_ack => done_fork_4_fwb_ack
584           );
585
586   done_fork_5 : entity work.fork
587           generic map(
588             PHASE_INIT => '0'
589           )
590           port map(
591            rst => rst,
592            -- Input channel
593            inA_req => done_fork_3_fwt_req,
594            inA_ack => done_fork_3_fwt_ack,
595            -- Output channel 1
596            outB_req => done_fork_5_fwt_req,
597            outB_ack => done_fork_5_fwt_ack,
598            -- Output channel 2
599            outC_req => done_fork_5_fwb_req,
600            outC_ack => done_fork_5_fwb_ack
601           );
602
603   done_fork_6 : entity work.fork
604           generic map(
605             PHASE_INIT => '0'
606           )
607           port map(
608            rst => rst,
609            -- Input channel
610            inA_req => done_fork_0_fwb_req,
611            inA_ack => done_fork_0_fwb_ack,
612            -- Output channel 1
613            outB_req => done_fork_6_fwt_req,
614            outB_ack => done_fork_6_fwt_ack,
615            -- Output channel 2
```

```
616             outC_req => done_fork_6_fwb_req,
617             outC_ack => done_fork_6_fwb_ack
618           );
619
620     done_fork_7 : entity work.fork
621           generic map(
622             PHASE_INIT => '0'
623           )
624           port map(
625             rst => rst,
626             -- Input channel
627             inA_req => done_fork_6_fwb_req,
628             inA_ack => done_fork_6_fwb_ack,
629             -- Output channel 1
630             outB_req => done_fork_7_fwt_req,
631             outB_ack => done_fork_7_fwt_ack,
632             -- Output channel 2
633             outC_req => done_fork_7_fwb_req,
634             outC_ack => done_fork_7_fwb_ack
635           );
636
637     a_join : entity work.join
638         generic map(
639             PHASE_INIT => '0' --TODO verify phase
640         )
641         port map(
642             rst => rst,
643             --UPSTREAM channels
644             inA_req => done_fork_7_fwb_req,
645             inA_ack => done_fork_7_fwb_ack,
646             inB_req => a_fork_1_fwb_req,
647             inB_ack => a_fork_1_fwb_ack,
648             --DOWNSTREAM channel
649             outC_req => a_join_fw_req,
650             outC_ack => a_join_fw_ack
651
652         );
653
654         b_join : entity work.join
655         generic map(
656             PHASE_INIT => '0' --TODO verify phase
657         )
658         port map(
659             rst => rst,
660             --UPSTREAM channels
```

```
661             inA_req => done_fork_7_fwt_req,
662             inA_ack => done_fork_7_fwt_ack,
663             inB_req => b_fork_0_fwb_req,
664             inB_ack => b_fork_0_fwb_ack,
665             --DOWNSTREAM channel
666             outC_req => b_join_fw_req,
667             outC_ack => b_join_fw_ack
668
669         );
670
671
672     a_mux : entity work.mux
673         --generic for initializing the phase registers
674         generic map (
675             DATA_WIDTH => BITWIDTH,
676             PHASE_INIT_C => '0',
677             PHASE_INIT_A => '0',
678             PHASE_INIT_B => '0',
679             PHASE_INIT_SEL => '0'
680         )
681         port map(
682           rst => rst,
683           -- Input from channel 1
684           inA_req => in_fork_fwt_req,
685           inA_data => a_in,
686           inA_ack => in_fork_fwt_ack,
687           -- Input from channel 2
688           inB_req => a_demux_fwc_req,
689           inB_data => a_demux_fwc_data,
690           inB_ack => a_demux_fwc_ack,
691           -- Output port
692           outC_req => a_mux_fw_req,
693           outC_data => a_mux_fw_data,
694           outC_ack => a_mux_fw_ack,
695           -- Select port
696           inSel_req => done_fork_4_fwt_req,
697           inSel_ack => done_fork_4_fwt_ack,
698           selector(0) => nor_fw_data
699           );
700
701
702       b_click_0 : entity work.decoupled_hs_reg
703           generic map(
704               DATA_WIDTH => BITWIDTH*2,
705               VALUE => 0
```

```
706             )
707             port map(
708                 rst => rst,
709                 -- Input channel
710                 in_ack => b_mux_fw_ack,
711                 in_req => b_mux_fw_req,
712                 in_data =>  b_mux_fw_data,
713                 -- Output channel
714                 out_req => b_click_0_fw_req,
715                 out_data => b_click_0_fw_data,
716                 out_ack => b_click_0_fw_ack
717             );
718
719         b_click_1 : entity work.decoupled_hs_reg
720             generic map(
721                 DATA_WIDTH => BITWIDTH*2,
722                 VALUE => 0,
723                 PHASE_INIT_OUT => '1'
724             )
725             port map(
726                 rst => rst,
727                 -- Input channel
728                 in_ack => b_click_0_fw_ack,
729                 in_req => b_click_0_fw_req,
730                 in_data =>  b_click_0_fw_data,
731                 -- Output channel
732                 out_req => b_click_1_fw_req,
733                 out_data => b_click_1_fw_data,
734                 out_ack => b_click_1_fw_ack
735             );
736
737         b_fork_0 : entity work.fork
738             generic map(
739               PHASE_INIT => '0'  -- set the phase to the same as the click
    ↪ element driving it
740             )
741             port map(
742              rst => rst,
743              -- Input channel
744              inA_req => b_click_1_fw_req,
745              inA_ack => b_click_1_fw_ack,
746              -- Output channel 1
747              outB_req => b_fork_0_fwt_req,
748              outB_ack => b_fork_0_fwt_ack,
749              -- Output channel 2
```

```
750             outC_req => b_fork_0_fwb_req,
751             outC_ack => b_fork_0_fwb_ack
752             );
753
754
755     b_click_1_fw_data_shift <= (b_click_1_fw_data(BITWIDTH*2-2 downto 0) &
    ↪ '0');
756     B_DEMUX_IN : entity work.demux
757             generic map(
758                 DATA_WIDTH    => BITWIDTH*2,
759                 PHASE_INIT_A => '0', --TODO consider INIT
760                 PHASE_INIT_B => '0',
761                 PHASE_INIT_C => '0'
762             )
763             port map(
764                 rst => rst,
765                 -- Input port
766                 inA_req => b_fork_0_fwt_req,
767                 inA_data => b_click_1_fw_data_shift,
768                 inA_ack => b_fork_0_fwt_ack,
769                 -- Select port
770                 inSel_req => done_fork_5_fwb_req,
771                 inSel_ack => done_fork_5_fwb_ack,
772                 selector => nor_fw_data,
773                 -- Output channel 1
774                 outB_req => b_demux_fwb_req,    --TODO Verify that the selector
    ↪ decides the intented output
775                 outB_data => b_demux_fwb_data,
776                 outB_ack => b_demux_fwb_ack,
777                 -- Output channel 2
778                 outC_req => b_demux_fwc_req,
779                 outC_data => b_demux_fwc_data,
780                 outC_ack => b_demux_fwc_ack
781                 );
782
783
784
785     B_DEMUX_SINK : entity work.sink
786             generic map(
787                 BITWIDTH => BITWIDTH*2,
788                 sink_delay => sink_delay
789             )
790             port map(
791                 req_in  => b_demux_fwb_req,
792                 ack_out => b_demux_fwb_ack,
```

```
793                 data_in => b_demux_fwb_data
794             );
795 
796               b_mux_fw_data_inA <= b_pad & b_in;
797 
798 
799       b_mux : entity work.mux
800           --generic for initializing the phase registers
801           generic map (
802              DATA_WIDTH => BITWIDTH*2,
803              PHASE_INIT_C => '0',
804              PHASE_INIT_A => '0',
805              PHASE_INIT_B => '0',
806              PHASE_INIT_SEL => '0'
807           )
808           port map(
809             rst => rst,
810             -- Input from channel 1
811             inA_req => in_fork_fwb_req,
812             inA_data => b_mux_fw_data_inA,
813             inA_ack => in_fork_fwb_ack,
814             -- Input from channel 2
815             inB_req => b_demux_fwc_req,
816             inB_data => b_demux_fwc_data,
817             inB_ack => b_demux_fwc_ack,
818             -- Output port
819             outC_req => b_mux_fw_req,
820             outC_data => b_mux_fw_data,
821             outC_ack => b_mux_fw_ack,
822             -- Select port
823             inSel_req => done_fork_5_fwt_req,
824             inSel_ack => done_fork_5_fwt_ack,
825             selector(0) => nor_fw_data
826             );
827 
828 
829     CSA_demux_in_selector <= a_click_1_fw_data(0) or nor_fw_data;
830     csa_demux_in_delay : entity work.delay_element
831         generic map(
832             size => 1
833         )
834         port map(
835             d => b_join_fw_req,
836             z => csa_demux_in_req
837         );
```

```
838
839     CSA_demux_in_data <= b_click_1_fw_data when nor_fw_data = '0' else (others
    ↪  => '0');
840     CSA_DEMUX_IN : entity work.demux
841         generic map(
842             DATA_WIDTH    => BITWIDTH*2,
843             PHASE_INIT_A => '0', --TODO consider INIT
844             PHASE_INIT_B => '0',
845             PHASE_INIT_C => '0'
846         )
847         port map(
848             rst => rst,
849             -- Input port
850             inA_req => csa_demux_in_req,
851             inA_data => CSA_demux_in_data,
852             inA_ack => b_join_fw_ack,
853             -- Select port
854             inSel_req => a_fork_2_fwb_req,
855             inSel_ack => a_fork_2_fwb_ack,
856             selector => CSA_demux_in_selector,
857             -- Output channel 1
858             outB_req => csa_demux_fwb_req,   --TODO Verify that the selector
    ↪ decides the intented output
859             outB_data => csa_demux_fwb_data,
860             outB_ack => csa_demux_fwb_ack,
861             -- Output channel 2
862             outC_req => csa_demux_fwc_req,
863             outC_data => csa_demux_fwc_data,
864             outC_ack => csa_demux_fwc_ack
865             );
866
867
868     NO_ADD_SINK : entity work.sink
869         generic map(
870             BITWIDTH => BITWIDTH*2,
871             sink_delay => sink_delay
872         )
873         port map(
874             req_in  => csa_demux_fwc_req,
875             ack_out => csa_demux_fwc_ack,
876             data_in => csa_demux_fwc_data
877         );
878
879     CSA_JOIN : entity work.join
880         generic map(
```

```
881             PHASE_INIT => '0' --TODO verify phase
882         )
883         port map(
884             rst => rst,
885             --UPSTREAM channels
886             inA_req => csa_mux_fw_req,
887             inA_ack => csa_mux_fw_ack,
888             inB_req => csa_demux_fwb_req,
889             inB_ack => csa_demux_fwb_ack,
890             --DOWNSTREAM channel
891             outC_req => CSA_join_fw_req,
892             outC_ack => CSA_join_fw_ack
893 
894         );
895 
896     CSA1 : entity work.CSA
897         generic map(
898             BITWIDTH => BITWIDTH*2,
899             CSA_DELAY => CSA_DELAY
900         )
901         port map(
902             CSA_in_0 => csa_mux_fw_data(BITWIDTH*2-1 downto 0),
903             CSA_in_1 => csa_mux_fw_data(BITWIDTH*2*2-1 downto BITWIDTH*2),
904             CSA_in_2 => csa_demux_fwb_data,
905 
906             CSA_out_S => CSA_sum,
907             CSA_out_C => CSA_carry
908         );
909 
910 
911 
912     csa_delay_req : entity work.delay_element
913         generic map(
914             size => 6
915         )
916         port map(
917             d => CSA_join_fw_req,
918             z => CSA_join_fw_delayed_req
919         );
920 
921 --    csa_delay_reg : entity work.delay_element_sim
922 --        generic map(
923 --            delay => CSA_DELAY * SAFETY_MARGIN
924 --        )
925 --        port map(
```

```
926 --             d => CSA_join_fw_req,
927 --             z => CSA_join_fw_delayed_req
928 --         );
929
930
931 --     csa_delay_ack : entity work.delay_element_sim
932 --         generic map(
933 --             delay => CSA_DELAY * SAFETY_MARGIN
934 --         )
935 --         port map(
936 --             d => CSA_join_fw_ack,
937 --             z => CSA_join_fw_delayed_ack
938 --         );
939
940
941     CSA_click_0_data_in <= CSA_sum & (CSA_carry(BITWIDTH*2-2 downto 0) & '0');
942     csa_click_0 : entity work.decoupled_hs_reg
943         generic map(
944             DATA_WIDTH => BITWIDTH*2*2,
945             VALUE => 0
946         )
947         port map(
948             rst => rst,
949             -- Input channel
950             in_ack => CSA_join_fw_ack,
951             in_req => CSA_join_fw_delayed_req,
952             in_data =>  CSA_click_0_data_in,
953             -- Output channel
954             out_req => CSA_click_0_fw_req,
955             out_data => CSA_click_0_fw_data,
956             out_ack => CSA_click_0_fw_ack
957         );
958
959
960     csa_click_1 : entity work.decoupled_hs_reg
961         generic map(
962             DATA_WIDTH => BITWIDTH*2*2,
963             VALUE => 0,
964             PHASE_INIT_IN => '0',
965             PHASE_INIT_OUT => '1'
966         )
967         port map(
968             rst => rst,
969             -- Input channel
970             in_ack => CSA_click_0_fw_ack,
```

```
971             in_req => CSA_click_0_fw_req,
972             in_data =>  CSA_click_0_fw_data,
973             -- Output channel
974             out_req => CSA_click_1_fw_req,
975             out_data => CSA_click_1_fw_data,
976             out_ack => CSA_click_1_fw_ack
977         );
978
979     reset_barrier_CSA_click1 <= CSA_click_1_fw_req AND start;
980
981
982
983     CSA_DEMUX_OUT : entity work.demux
984         generic map(
985             DATA_WIDTH    => BITWIDTH*2*2,
986             PHASE_INIT_A => '0', --TODO consider INIT
987             PHASE_INIT_B => '0',
988             PHASE_INIT_C => '0'
989         )
990         port map(
991             rst => rst,
992             -- Input port
993             inA_req => reset_barrier_CSA_click1,
994             inA_data => CSA_click_1_fw_data,
995             inA_ack => CSA_click_1_fw_ack,
996             -- Select port
997             inSel_req => done_fork_1_fwb_req,
998             inSel_ack => done_fork_1_fwb_ack,
999             selector => nor_fw_data,
1000            -- Output channel 1
1001            outC_req => csa_demux_out_fwb_req,   --TODO Verify that the
    ↪ selector decides the intented output
1002            outC_data => csa_demux_out_fwb_data,
1003            outC_ack => csa_demux_out_fwb_ack,
1004            -- Output channel 2
1005            outB_req => csa_demux_out_fwc_req,
1006            outB_data => csa_demux_out_fwc_data,
1007            outB_ack => csa_demux_out_fwc_ack
1008            );
1009
1010
1011
1012 --    RES_SINK : entity work.sink
1013 --    generic map(
1014 --        BITWIDTH => BITWIDTH*2*2,
```

```
1015 --         sink_delay => sink_delay
1016 --     )
1017 --     port map(
1018 --         req_in  => csa_demux_out_fwc_req,
1019 --         ack_out => csa_demux_out_fwc_ack,
1020 --         data_in => csa_demux_out_fwc_data
1021 --     );
1022     req_out <= csa_demux_out_fwc_req;
1023     csa_demux_out_fwc_ack <= ack_in;
1024 
1025     result_out <= std_logic_vector( unsigned(csa_demux_out_fwc_data(BITWIDTH
     ↪ *2*2-1 downto BITWIDTH*2)) + unsigned(csa_demux_out_fwc_data(BITWIDTH
     ↪ *2-1 downto 0)));
1026 
1027 
1028 
1029 
1030     CSA_RESET_SOURCE : entity work.source
1031     generic map( -- TODO check this component
1032         source_delay => source_delay
1033     )
1034     port map(
1035         req_out => csa_reset_src_req,
1036         ack_in  => csa_reset_src_ack
1037     );
1038 
1039     csa_mux : entity work.mux
1040         --generic for initializing the phase registers
1041         generic map (
1042            DATA_WIDTH => BITWIDTH*2*2,
1043            PHASE_INIT_C => '0', --TODO check init values
1044            PHASE_INIT_A => '0',
1045            PHASE_INIT_B => '0',
1046            PHASE_INIT_SEL => '0'
1047         )
1048         port map(
1049           rst => rst,
1050           -- Input from channel 1
1051           inA_req => csa_reset_src_req,
1052           inA_data => (others => '0'),
1053           inA_ack => csa_reset_src_ack,
1054           -- Input from channel 2
1055           inB_req => csa_demux_out_fwb_req,
1056           inB_data => csa_demux_out_fwb_data,
1057           inB_ack => csa_demux_out_fwb_ack,
```

```
1058            -- Output port
1059            outC_req => csa_mux_fw_req,
1060            outC_data => csa_mux_fw_data,
1061            outC_ack => csa_mux_fw_ack,
1062            -- Select port
1063            inSel_req => done_fork_1_fwt_req,
1064            inSel_ack => done_fork_1_fwt_ack,
1065            selector(0) => nor_fw_data
1066            );
1067
1068    --TODO:
1069    --  Fix flushing
1070    --  Fix req-ack to input - Look ok
1071    --  Fix req-ack to output
1072    --  Add results
1073    --  Fix req-out for tb
1074    -- we have added or gate neglecting delay
1075
1076
1077    -- man kan lave en renerer demux sink l sning
1078    -- annoter handhaske kritrier p  figu
1079
1080    end Behavior;
```

7.2 Multiplier_direct.vhd

```
1   --
    ↪ ----------------------------------------------------------------------------
    ↪
2   -- Company:
3   -- Engineer:
4   --
5   -- Create Date: 03/30/2020 12:11:34 PM
6   -- Design Name:
7   -- Module Name: multiplier - Behavioral
8   -- Project Name:
9   -- Target Devices:
10  -- Tool Versions:
11  -- Description:
12  --
13  -- Dependencies:
14  --
15  -- Revision:
16  -- Revision 0.01 - File Created
```

```
17 -- Additional Comments:
18 --
19 --
   ↪ ----------------------------------------------------------------------------------
   ↪
20
21
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
24
25 -- Uncomment the following library declaration if using
26 -- arithmetic functions with Signed or Unsigned values
27 use IEEE.NUMERIC_STD.ALL;
28
29 -- Uncomment the following library declaration if instantiating
30 -- any Xilinx leaf cells in this code.
31 --library UNISIM;
32 --use UNISIM.VComponents.all;
33
34 entity multiplier_direct is
35     Generic(
36         bitwidth : integer := 128
37     );
38     Port ( clk : in STD_LOGIC;
39            rst : in STD_LOGIC;
40            a : in STD_LOGIC_VECTOR (bitwidth-1 downto 0);
41            b : in STD_LOGIC_VECTOR (bitwidth-1 downto 0);
42            result : out STD_LOGIC_VECTOR (bitwidth*2-1 downto 0);
43            done : out STD_LOGIC;
44            valid : in STD_LOGIC;
45            ready : out STD_LOGIC);
46 end multiplier_direct;
47
48 architecture Behavior of multiplier_direct is
49
50     signal a_reg, b_reg : std_logic_vector(bitwidth-1 downto 0);
51
52     signal result_reg : std_logic_vector(bitwidth*2-1 downto 0);
53
54     signal valid_reg, valid_reg2 : std_logic;
55
56 begin
57
58
59     process(clk,rst)
```

```
60      begin
61          if rst = '1' then
62
63              a_reg <= (others => '0');
64              b_reg <= (others => '0');
65              result_reg <= (others => '0');
66              valid_reg <= '0';
67              valid_reg2 <= '0';
68
69
70          elsif rising_edge(clk) then
71              a_reg <= a;
72              b_reg <= b;
73              result_reg <= std_logic_vector(unsigned(a_reg)*unsigned(b_reg));
74              valid_reg <= valid;
75              valid_reg2 <= valid_reg;
76          end if;
77      end process;
78
79      result <= result_reg;
80
81      ready <= '1'; --Pipelined so always ready for next signal.
82      done  <= valid_reg2; --It takes two cycles before the result comes out.
83
84  end Behavior;
```

7.3 Multiplier_seq.vhd

```
1   --
        ↪ ----------------------------------------------------------------------
        ↪
2   -- Company:
3   -- Engineer:
4   --
5   -- Create Date: 03/30/2020 12:11:34 PM
6   -- Design Name:
7   -- Module Name: multiplier - Behavioral
8   -- Project Name:
9   -- Target Devices:
10  -- Tool Versions:
11  -- Description:
12  --
13  -- Dependencies:
14  --
```

```
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 --
    ↪ ----------------------------------------------------------------------------
    ↪
20
21
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
24
25 -- Uncomment the following library declaration if using
26 -- arithmetic functions with Signed or Unsigned values
27 use IEEE.NUMERIC_STD.ALL;
28
29 -- Uncomment the following library declaration if instantiating
30 -- any Xilinx leaf cells in this code.
31 --library UNISIM;
32 --use UNISIM.VComponents.all;
33
34 entity multiplier_seq is
35     Generic(
36         bitwidth : integer := 128
37     );
38     Port ( clk : in STD_LOGIC;
39            rst : in STD_LOGIC;
40            a : in STD_LOGIC_VECTOR (bitwidth-1 downto 0);
41            b : in STD_LOGIC_VECTOR (bitwidth-1 downto 0);
42            result : out STD_LOGIC_VECTOR (bitwidth*2-1 downto 0);
43            done : out STD_LOGIC;
44            valid : in STD_LOGIC;
45            ready : out STD_LOGIC);
46 end multiplier_seq;
47
48 architecture Behavior of multiplier_seq is
49
50     signal a_reg, b_reg : std_logic_vector(bitwidth*2-1 downto 0);
51
52     signal result_reg : std_logic_vector(bitwidth*2-1 downto 0);
53
54
55     signal valid_reg, valid_reg2 : std_logic;
56
57
```

```
58      signal CSA_out_S, CSA_out_C : std_logic_vector(bitwidth*2-1 downto 0);
59      signal CSA_out_S_reg, CSA_out_C_reg : std_logic_vector(bitwidth*2-1 downto
    ↪  0);
60
61      signal CSA_in_0, CSA_in_1, CSA_in_2 : std_logic_vector(bitwidth*2-1 downto
    ↪  0);
62
63
64      signal xor1 : std_logic_vector(bitwidth*2-1 downto 0);
65
66      signal calculating : std_logic;
67
68  begin
69
70
71      --CSA part
72
73      gen_CSA: for i in 0 to bitwidth*2-1 generate
74          xor1(i) <= CSA_in_0(i) xor CSA_in_1(i);
75
76          CSA_out_S(i) <= xor1(i) xor CSA_in_2(i);
77
78          CSA_out_C(i) <= (xor1(i) and CSA_in_2(i)) or (CSA_in_0(i) and CSA_in_1
    ↪ (i));
79      end generate;
80
81
82      CSA_in_0 <= (others => '0') when calculating = '0' else CSA_out_S_reg;
83      CSA_in_1 <= (others => '0') when calculating = '0' else CSA_out_C_reg sll
    ↪ 1;
84
85      process(all)
86      begin
87          if calculating = '1' then
88              if a_reg(0) = '0' then
89                  CSA_in_2 <= (others => '0');
90              else
91                  CSA_in_2 <= b_reg;
92              end if;
93          else
94              if a(0) = '0' then
95                  CSA_in_2 <= (others => '0');
96              else
97                  CSA_in_2 <= (bitwidth-1 downto 0 => '0') & b;
98              end if;
```

```
99          end if;
100     end process;
101
102     process(clk,rst)
103     begin
104         if rst = '1' then
105             a_reg <= (others => '0');
106             b_reg <= (others => '0');
107
108             valid_reg <= '0';
109             valid_reg2 <= '1';
110
111             result_reg <= (others => '0');
112
113             calculating <= '0';
114
115             CSA_out_S_reg <= (others => '0');
116             CSA_out_C_reg <= (others => '0');
117
118
119         elsif rising_edge(clk) then
120             valid_reg <= valid;
121
122
123             if calculating = '1' then
124                 a_reg <= a_reg srl 1;
125                 b_reg <= b_reg sll 1;
126             elsif valid = '1' then
127                 calculating <= '1';
128                 a_reg <= (bitwidth-1 downto 0 => '0') & a srl 1;
129                 b_reg <= (bitwidth-1 downto 0 => '0') & b sll 1;
130             end if;
131
132             CSA_out_S_reg <= CSA_out_S;
133             CSA_out_C_reg <= CSA_out_C;
134
135             if(a_reg = (a_reg'RANGE => '0') and calculating = '1') then
136                 result_reg <= std_logic_vector(unsigned(CSA_out_S_reg) +
    ↪ unsigned(CSA_out_C_reg sll 1));
137                 if calculating = '1' then
138                     done <= '1';
139                 else
140                     done <= '0';
141                 end if;
142                 calculating <= '0';
```

```
143             else
144                 done <= '0';
145             end if;
146 
147 
148         end if;
149 
150     end process;
151 
152     ready <= not calculating;
153     result <= result_reg;
154     --std_logic_vector(unsigned(CSA_out_S_reg) + unsigned(CSA_out_C_reg sll 1)
    ↪ );--
155 end Behavior;
```

7.4 Multiplier_async_tb.vhd

```
1  --
     ↪ ----------------------------------------------------------------------------
     ↪
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date: 03/30/2020 12:23:30 PM
6  -- Design Name:
7  -- Module Name: multiplier_tb - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 --
     ↪ ----------------------------------------------------------------------------
     ↪
20 
21 
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
```

```
24
25 -- Uncomment the following library declaration if using
26 -- arithmetic functions with Signed or Unsigned values
27 use IEEE.NUMERIC_STD.ALL;
28
29 -- Uncomment the following library declaration if instantiating
30 -- any Xilinx leaf cells in this code.
31 --library UNISIM;
32 --use UNISIM.VComponents.all;
33
34 entity multiplier_async_tb is
35 --  Port ( );
36 end multiplier_async_tb;
37
38 architecture Behavioral of multiplier_async_tb is
39
40     -- Clock period definitions
41     constant clock_period : time := 20 ns;
42
43     constant bitwidth : integer := 32;
44     constant lfsr1_poly : std_logic_vector := ("
    ↪ 01001011001010100100101100101010");--
45     constant lfsr2_poly : std_logic_vector := ("
    ↪ 01101010001000100110101000100010");
46     --constant lfsr1_poly : std_logic_vector := ("01101010");
47     --constant lfsr2_poly : std_logic_vector := ("00100010");
48     signal clk : std_logic := '0';
49     signal rst : std_logic := '1';
50
51     signal lfsr1_out : std_logic_vector(bitwidth-1 downto 0);
52     signal lfsr1_en : std_logic := '0';
53     signal lfsr2_out : std_logic_vector(bitwidth-1 downto 0);
54     signal lfsr2_en : std_logic := '0';
55
56
57     signal mul_result : std_logic_vector(bitwidth*2-1 downto 0);
58
59     signal mul_ready : std_logic;
60     signal mul_done : std_logic;
61     signal mul_valid : std_logic := '0';
62
63     signal done : std_logic := '0';
64
65
66
```

```
67      --Multiplier data
68      signal a_in, b_in : std_logic_vector(bitwidth-1 downto 0);
69
70      --Handshake for multiplier
71      signal req_in, req_out : std_logic := '0';
72      signal ack_out, ack_in : std_logic := '0';
73
74      signal nreset : std_logic;
75  begin
76
77      nreset <= not rst;
78
79      --Sink the output using a loopback
80      ack_in <= transport req_out after 1 ns;
81
82
83      multiplier: entity work.multiplier_async
84          --GENERIC MAP(
85          --     BITWIDTH => bitwidth
86          --)
87          PORT MAP(
88              rst => rst,
89
90              req_in => req_in,
91              req_out => req_out,
92
93              ack_in => ack_in,
94              ack_out => ack_out,
95
96              start => nreset,
97
98              --Data
99              a_in => a_in,
100             b_in => b_in,
101
102             result_out => mul_result
103         );
104
105     --Stimulus of multiplier. Go through a list of values and multiply them
106     process
107     begin
108
109         a_in <= (others => '0');
110         b_in <= (others => '0');
111
```

```
112
113         wait until rst = '0';
114
115         wait for 100 ns;
116
117
118
119
120
121         for i in 0 to 10000 loop
122             lfsr1_en <= '0';
123             lfsr2_en <= '0';
124
125             wait until falling_edge(clk);
126
127             if req_in /= ack_out then
128                 wait until req_in = ack_out;
129             end if;
130
131             req_in <= not req_in;
132
133
134             --a_in <= std_logic_vector(to_unsigned(43690,32));-- lfsr1_out;
135             --b_in <= std_logic_vector(to_unsigned(4095,32));--lfsr1_out;
136             a_in <= lfsr1_out;
137             b_in <= lfsr2_out;
138
139             -- Wait until req_out triggers
140             --report "waiting for req_out to change";
141             wait on req_out;
142             --report "Req_out changed";
143
144
145
146
147             assert std_logic_vector(unsigned(lfsr1_out) * unsigned(lfsr2_out))
    ↪  = mul_result
148                 report "Multiplications did not match" severity error;
149
150
151
152             lfsr1_en <= '1';
153             lfsr2_en <= '1';
154             wait until rising_edge(clk);
155         end loop;
```

```
156
157
158     done <= '1';
159     std.env.finish;
160     wait;
161     end process;
162
163
164     --Two LFSR generates pseudo-random numbers for multiplying.
165     lfsr1: entity work.lfsr
166     GENERIC MAP(
167         G_M => bitwidth,
168         G_POLY => lfsr1_poly
169     )
170     PORT MAP(
171         i_clk  => clk,
172         i_rstb => rst,
173         o_lsfr => lfsr1_out,
174         i_en   => lfsr1_en
175     );
176
177     lfsr2: entity work.lfsr
178     GENERIC MAP(
179         G_M => bitwidth,
180         G_POLY => lfsr2_poly
181     )
182     PORT MAP(
183         i_clk => clk,
184         i_rstb => rst,
185         o_lsfr => lfsr2_out,
186         i_en   => lfsr2_en
187     );
188
189     --clock process
190     process
191     begin
192         wait for clock_period/2;
193         clk <= not clk;
194
195         if done = '1' then
196             wait;
197         end if;
198     end process;
199
200     --reset process
```

```
201     process
202     begin
203         rst <= '1';
204         wait for clock_period*5;
205         rst <= '0';
206         wait;
207     end process;
208 
209 end Behavioral;
```

7.5 Multiplier_tb.vhd

```
1  --
       ↪ ----------------------------------------------------------------------------------
       ↪
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date: 03/30/2020 12:23:30 PM
6  -- Design Name:
7  -- Module Name: multiplier_tb - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 --
       ↪ ----------------------------------------------------------------------------------
       ↪
20 
21 
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
24 
25 -- Uncomment the following library declaration if using
26 -- arithmetic functions with Signed or Unsigned values
27 use IEEE.NUMERIC_STD.ALL;
28 
```

```
29 -- Uncomment the following library declaration if instantiating
30 -- any Xilinx leaf cells in this code.
31 --library UNISIM;
32 --use UNISIM.VComponents.all;
33
34 entity multiplier_tb is
35 --  Port ( );
36 end multiplier_tb;
37
38 architecture Behavioral of multiplier_tb is
39
40     -- Clock period definitions
41     constant clock_period : time := 20 ns;
42
43     constant bitwidth : integer := 32;
44     constant lfsr1_poly : std_logic_vector := (”
    ↪ 01001011001010100100101100101010”);--01001011001010100100101100101010”)
    ↪ ;--
45     constant lfsr2_poly : std_logic_vector := (”
    ↪ 01101010001000100110101000100010”);--01101010001000100110101000100010”);
46
47     signal clk : std_logic := '0';
48     signal rst : std_logic := '1';
49
50     signal lfsr1_out : std_logic_vector(bitwidth-1 downto 0);
51     signal lfsr1_en : std_logic := '0';
52     signal lfsr2_out : std_logic_vector(bitwidth-1 downto 0);
53     signal lfsr2_en : std_logic := '0';
54
55
56     signal mul_result : std_logic_vector(bitwidth*2-1 downto 0);
57
58     signal mul_ready : std_logic;
59     signal mul_done : std_logic;
60     signal mul_valid : std_logic := '0';
61
62     signal done : std_logic := '0';
63
64
65
66 begin
67
68
69     multiplier: entity work.multiplier_seq
70         --GENERIC MAP(
```

```
71          --      bitwidth => bitwidth
72          --)
73          PORT MAP(
74              clk => clk,
75              rst => rst,
76              a => lfsr1_out,
77              b => lfsr2_out,
78              result => mul_result,
79              ready => mul_ready,
80              valid => mul_valid,
81              done  => mul_done
82          );
83
84      --Stimulus of multiplier. Go through a list of values and multiply them
85      process
86      begin
87          wait until rst = '0';
88          wait until rising_edge(clk);
89
90          --Wait for multiplier to be ready. Might need to initialize
91          if mul_ready = '0' then
92              wait until mul_ready = '1';
93          end if;
94
95          for i in 0 to 10000 loop
96
97              mul_valid <= '1';
98              lfsr1_en <= '0';
99              lfsr2_en <= '0';
100             wait until rising_edge(clk);
101             mul_valid <= '0';
102
103
104             if mul_done = '0' then
105                 wait until mul_done = '1';
106             end if;
107
108
109             wait for 1 ps;
110
111             assert std_logic_vector(unsigned(lfsr1_out) * unsigned(lfsr2_out))
    ↪  = mul_result
112                 report "Multiplications did not match" severity error;
113
114             lfsr1_en <= '1';
```

```
115             lfsr2_en <= '1';
116             wait until rising_edge(clk);
117             lfsr1_en <= '0';
118             lfsr2_en <= '0';
119         end loop;
120 
121 
122     done <= '1';
123     std.env.finish;
124     wait;
125     end process;
126 
127 
128     --Two LFSR generates pseudo-random numbers for multiplying.
129     lfsr1: entity work.lfsr
130     GENERIC MAP(
131         G_M => bitwidth,
132         G_POLY => lfsr1_poly
133     )
134     PORT MAP(
135         i_clk  => clk,
136         i_rstb => rst,
137         o_lsfr => lfsr1_out,
138         i_en   => lfsr1_en
139     );
140 
141     lfsr2: entity work.lfsr
142     GENERIC MAP(
143         G_M => bitwidth,
144         G_POLY => lfsr2_poly
145     )
146     PORT MAP(
147         i_clk => clk,
148         i_rstb => rst,
149         o_lsfr => lfsr2_out,
150         i_en   => lfsr2_en
151     );
152 
153     --clock process
154     process
155     begin
156         wait for clock_period/2;
157         clk <= not clk;
158 
159         if done = '1' then
```

```
160             wait;
161         end if;
162     end process;
163 
164     --reset process
165     process
166     begin
167         rst <= '1';
168         wait for clock_period*5;
169         rst <= '0';
170         wait;
171     end process;
172 
173 end Behavioral;
```

7.6 lfsr.vhd

```
1  --Modified from https://surf-vhdl.com/how-to-implement-an-lfsr-in-vhdl/
2  
3  library ieee;
4  use ieee.std_logic_1164.all;
5  
6  
7  entity lfsr is
8      generic(
9        G_M             : integer          := 7          ;
10       G_POLY          : std_logic_vector  := "1100000") ;  -- x^7+x^6+1
11     port (
12       i_clk           : in  std_logic;
13       i_rstb          : in  std_logic;
14       i_en            : in  std_logic;
15       o_lsfr          : out std_logic_vector (G_M-1 downto 0));
16 end lfsr;
17 
18 
19 architecture rtl of lfsr is
20     signal r_lfsr           : std_logic_vector (G_M downto 1);
21     signal w_mask           : std_logic_vector (G_M downto 1);
22     signal w_poly           : std_logic_vector (G_M downto 1);
23 
24     begin
25 
26     o_lsfr  <= r_lfsr(G_M downto 1);
27     w_poly  <= G_POLY;
```

```
28      g_mask : for k in G_M downto 1 generate
29        w_mask(k) <= w_poly(k) and r_lfsr(1);
30      end generate g_mask;
31
32      p_lfsr : process (i_clk,i_rstb) begin
33        if (i_rstb = '1') then
34          r_lfsr  <= (others=>'1');
35        elsif rising_edge(i_clk) then
36          if i_en = '1' then
37            r_lfsr  <= '0'&r_lfsr(G_M downto 2) xor w_mask;
38          end if;
39        end if;
40      end process p_lfsr;
41
42  end architecture rtl;
```

7.7 nor.vhd

```
1   ----------------------------------------------------------------------------------
2   -- Company:
3   -- Engineer:
4   --
5   -- Create Date: 03/30/2020 12:11:34 PM
6   -- Design Name:
7   -- Module Name: multiplier - Behavioral
8   -- Project Name:
9   -- Target Devices:
10  -- Tool Versions:
11  -- Description:
12  --
13  -- Dependencies:
14  --
15  -- Revision:
16  -- Revision 0.01 - File Created
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------------------
20
21
22  library IEEE;
```

```
23 use IEEE.STD_LOGIC_1164.ALL;
24
25 -- Uncomment the following library declaration if using
26 -- arithmetic functions with Signed or Unsigned values
27 use IEEE.NUMERIC_STD.ALL;
28
29 use IEEE.std_logic_misc.ALL;
30
31 entity NOR_gate is
32     Generic(
33         BITWIDTH : integer := 16;
34         NOR_DELAY : time := 2 ps
35     );
36     Port (
37         in_req      : in std_logic;
38         in_ack      : out std_logic;
39         in_bus      : in std_logic_vector(BITWIDTH - 1 downto 0);
40         -- Output channel
41         out_req     : out std_logic;
42         out_ack     : in std_logic;
43         result      : out std_logic);
44 end NOR_gate;
45
46
47 architecture Behavior of NOR_gate is
48
49
50 begin
51     -- TODO Implement delay
52
53     nor_delay_lut : entity work.delay_element
54         generic map(
55             size => 4
56         )
57         port map(
58             d => in_req,
59             z => out_req
60         );
61     in_ack <= out_ack;
62     result  <= nor_reduce(in_bus);
63
64 end Behavior;
```

7.8 sink.vhd

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  -- Uncomment the following library declaration if using
6  -- arithmetic functions with Signed or Unsigned values
7  use IEEE.NUMERIC_STD.ALL;
8
9  use IEEE.std_logic_misc.ALL;
10
11 entity sink is
12     Generic(
13         BITWIDTH : integer := 16;
14         sink_delay : time := 1 ps
15     );
16     Port (
17         req_in       : in std_logic;
18         ack_out      : out std_logic;
19         data_in      : in std_logic_vector(BITWIDTH - 1 downto 0));
20 end sink;
21
22
23 architecture Behavior of sink is
24
25     signal notted : std_logic;
26
27     attribute dont_touch : string;
28     attribute dont_touch of notted : signal is "true";
29
30
31
32 begin
33
34     notted <= transport not(req_in) after sink_delay;
35     ack_out <= not notted;
36 end Behavior;
```

7.9 source.vhd

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  -- Uncomment the following library declaration if using
```

```
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

use IEEE.std_logic_misc.ALL;

entity source is
    Generic(
        source_delay : time := 1 ps
    );
    Port (
        req_out      : out std_logic;
        ack_in       : in std_logic);
end source;

architecture Behavior of source is

    signal internal : std_logic;

    attribute dont_touch : string;
    attribute dont_touch of internal : signal is "true";


begin
    internal <= transport not(ack_in) after source_delay;
    req_out <= internal;
end Behavior;
```

7.10 CSA.vhd

```
--
    ↪ ----------------------------------------------------------------------------------
    ↪
-- Company:
-- Engineer:
--
-- Create Date: 03/30/2020 12:11:34 PM
-- Design Name:
-- Module Name: multiplier - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
```

```
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 --
    ↪ ----------------------------------------------------------------------------
    ↪
20
21
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
24
25 -- Uncomment the following library declaration if using
26 -- arithmetic functions with Signed or Unsigned values
27 use IEEE.NUMERIC_STD.ALL;
28
29 entity CSA is
30     Generic(
31         BITWIDTH : integer := 16;
32         CSA_DELAY: time := 4ns
33     );
34     Port (
35         CSA_in_0 : in std_logic_vector(BITWIDTH-1 downto 0);
36         CSA_in_1 : in std_logic_vector(BITWIDTH-1 downto 0);
37         CSA_in_2 : in std_logic_vector(BITWIDTH-1 downto 0);
38
39         CSA_out_S : out std_logic_vector(BITWIDTH-1 downto 0);
40         CSA_out_C : out std_logic_vector(BITWIDTH-1 downto 0));
41 end CSA;
42
43
44 architecture Behavior of CSA is
45
46     signal xor1  : std_logic_vector(BITWIDTH-1 downto 0);
47     signal CSA_S : std_logic_vector(BITWIDTH-1 downto 0);
48     signal CSA_C : std_logic_vector(BITWIDTH-1 downto 0);
49
50 begin
51     gen_CSA: for i in 0 to BITWIDTH-1 generate
52         xor1(i) <= CSA_in_0(i) xor CSA_in_1(i);
53
54         CSA_S(i) <= xor1(i) xor CSA_in_2(i);
55
```

```
56          CSA_C(i) <= (xor1(i) and CSA_in_2(i)) or (CSA_in_0(i) and CSA_in_1(i))
    ↪ ;
57      end generate;
58
59      CSA_out_S <= transport CSA_S after CSA_DELAY;
60      CSA_out_C <= transport CSA_C after CSA_DELAY;
61  end Behavior;
```