Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Bjørn-Inge Flølo Kleppe

Asynchronous construction from clocked designs

Master's thesis in Electronic Systems Design and Innovation
Supervisor: Bjørn B. Larsen (NTNU), Isael Diaz (Nordic Semiconductor)
Co-supervisor: Kaja Gausdal (Nordic Semiconductor)
June 2023

NORDIC®
SEMICONDUCTOR

Norwegian University of
Science and Technology

Bjørn-Inge Flølo Kleppe

Asynchronous construction from clocked designs

Master's thesis in Electronic Systems Design and Innovation
Supervisor: Bjørn B. Larsen (NTNU), Isael Diaz (Nordic Semiconductor)
Co-supervisor: Kaja Gausdal (Nordic Semiconductor)
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Abstract

This Master's thesis examines the potential advantages of asynchronous circuits, including increased speed and robustness and lower power consumption relative to synchronous circuits. Furthermore, the concept of desynchronization is introduced, a method to convert a synchronous circuit to an asynchronous one. A Tcl script is developed that automatically converts from a synchronous to an asynchronous netlist in Synopsys Design Compiler. The script is tested on a simple multiplication and division module. The resulting circuit is functional but does not exhibit the potential advantages of asynchronous design. Instead, it performs worse than the synchronous version. However, this thesis is only intended as proof of the concept of desynchronization using the available tools. The thesis outlines further work to improve the script and the desynchronization process to obtain the advantages of asynchronous design. This thesis is written for Nordic Semiconductor.

Samandrag

Denne masteroppgåva undersøker dei potensiale fordelane ved asynkrone kretsar. Desse fordelane er blant anna eit potensiale for ein raskare meir robust krets med eit lågare straum forbruk samanlikna med ein synkron krets. Vidare er konseptet desynkronisering introdusert. Dette er ein metode for å konvertera ein synkron krets til ein asynkron krets. I oppgåva blir det utvikla eit Tcl script som automatisk gjennomfører denne konverteringa i Synopsys Design Compiler. Scriptet er testa ved å konvertere ein enkel multiplikasjon og dividerings modul. Den asynkrone kretsen frå scriptet fungerer, men har ikkje nokon av fordelane ein asynkron krets kan ha. I staden presterer den verre enn den synkrone versjonen. Denne oppgåve er berre meint som eit bevis på at desynkronisering fungerer på dei tilgjengelege verktøya. Oppgåva gir og ein indikasjon på kva som trengs å gjerast vidare for å forbetre scriptet og utnytte fordelane ved asynkrone kretsar. Oppgåva er skrive for Nordic Semiconductor.

Acknowledgements

I would like to thank my supervisors Bjørn B. Larsen at NTNU, and Isael Diaz and Kaja Gausdal at Nordic Semiconductor for their help during this master thesis, as well as the preceding project. They have been a massive help during implementation, testing, and writing.

Table of Contents

List of Figures v

List of Tables vi

1 Introduction 1

2 Previous Work 2

3 Background 3

3.1 Asynchronous Designs . . . 3

3.1.1 Advantages of Asynchronous Design . . . 3

3.1.2 Data channels . . . 4

3.1.3 Handshakes . . . 6

3.1.4 Muller C-element . . . 7

3.1.5 Difficulty in testing due to lack of tools . . . 11

3.2 Desynchronization . . . 11

3.2.1 Flip-flop conversion . . . 12

3.2.2 Completion detection . . . 12

3.2.3 Controller . . . 18

3.2.4 Fork and Join structures . . . 20

3.2.5 Advantages . . . 22

4 Methodology 24

4.1 Tools . . . 24

4.2 MultDiv module . . . 24

4.3 Desynchronization . . . 25

4.4 Tcl script . . . 25

4.5 Delay calculation . . . 28

4.6 Verification and Testing . . . 30

5 Results 31

5.1 Area . . . 31

5.2 Speed . . . 32

5.3 Power . . . 32

6 Discussion 34

6.1 Area . . . 34

6.2 Speed 34
6.3 Power 36
6.4 Operating Conditions 37
6.5 Completion Detection 37
6.6 Controller 38
6.7 Limitations in Tcl script 38
6.8 Verification 39
6.9 Testbench 40
6.10 Desynchronization 40

7 Future Work **42**

8 Conclusion **43**

References **44**

Appendix **46**

A desync.tcl 46

List of Figures

1 Four-bit adder constructed from four full-adders. 4
2 Bundled data channel. 5
3 Transaction on a bundled-data channel. 5
4 Transaction on a dual-rail data channel [4]. 6
5 Transaction on a 1-of-4 data channel [4]. 6
6 Two-phase handshake [4]. 7
7 Four-phase handshake [4]. 7
8 Truth table of the Muller C-element with two inputs [4]. 8
9 Muller pipeline consisting of a C-element and an inverter for each stage [6]. 8
10 Symbol and truth table of an asymmetric C-element with four inputs [4]. 9
11 C-element implemented using transistors [4]. 9
12 C-element implemented using two- and three-input NAND gates [4]. 9
13 C-elements with active low reset. 10
14 A synchronous circuit converted to an asynchronous circuit using desynchronization [4]. 12
15 Asynchronous pipeline where registers have been replaced by one latch [4]. 12
16 Completion detection inserted between two controllers [4]. 13

17 Desyncrhonized circuit with simple delay line as completion detection [4]. 13

18 A general implementation of a speculative delay line [1]. 14

19 An efficient asymmetric speculative delay line [1]. 15

20 An efficient symmetric speculative delay line [1]. 16

21 Asymmetric delay line controller (ADLC) [1]. 16

22 Symmetric delay line controller (SDLC) [1]. 16

23 Current sensing completion detection [14]. 17

24 Block diagram of a controller with one latch enable signal, and re quest and acknowledge on input and output. 18

25 Gate implementation of the semi-decoupled controller [4]. 19

26 STG of the semi-decoupled controller[4]. 19

27 Block diagram of a Blade controller with error detecting logic [4]. 20

28 MOUSETRAP controller consisting of a single XNOR element [4]. 21

29 A block diagram of a circuit that forks into two paths before joining together again [4]. 21

30 A C-element being used for the acknowledge signal when one controller is connected to two others [4]. 22

31 A C-element being used for the request signal when two controllers are connected to the same one [4]. 22

32 Different configurations of C-element to combine multiple inputs into one. 23

33 Wave diagram of the C-element configuration in Figure 32a, the inputs are numbered from 0 to 2 starting from the top. 23

34 Block diagram of the MultDiv and ALU module from ibex [22]. 24

35 The desynchronization process of the Tcl script broken down into steps. 27

36 Block diagram of two controllers and latches, with names of the signals on the input and output [4]. 28

37 A transaction between the blocks in Figure 36. 29

38 Full handshake between two controllers. 36

39 A possible implementation of a speculative delay line for the MultDiv module. . . 37

40 STG of the fully-decoupled controller introduced by Furber and Day [5]. 39

41 STG of the desynchronization model controller introduced by Cortadella et al. [2]. 39

42 Possible implementation of a wrapper between an asynchronous circuit and a synchronous testbench. 40

List of Tables

1 Area comparison between synchronous and asynchronous 4- and 32-bit circuits, given in μm^2. 31

2 Area for the 4-bit versions split into subgroups, given in μm^2. 31

3 Area for the 32-bit versions split into subgroups, given in μm^2. 32

4 Speed comparison between synchronous and asynchronous 4- and 32-bit circuits, given in ns. 32

5 Dynamic power comparison between synchronous and asynchronous 4- and 32-bit circuits, given in mW. 32

6 Static power comparison between synchronous and asynchronous 4- and 32-bit circuits, given in nW. 33

1 Introduction

In the field of digital design synchronous circuits with a global clock have been the most prevalent choice for a long time. Asynchronous circuits are not that common, even though they have some advantages compared to synchronous design. An asynchronous circuit does not use a global clock. Instead, it uses local handshake mechanisms to indicate when data is valid. This makes it possible for an asynchronous circuit to be faster, more power efficient, and more robust to operating conditions compared to a synchronous circuit [1]. However, these advantages are not trivial to extract. Asynchronous design requires a complete mindset change from a designer as the communication between different blocks is completely changed compared to synchronous design. Moreover, few electronic design automation (EDA) tools are available, meaning the design process can be tedious and manual. Desynchronization is a methodology introduced to combine the advantages of synchronous and asynchronous design. The idea is to start with a functional synchronous circuit designed with the help of the existing EDA tools and later convert it to an asynchronous circuit to get the advantages this provides.

Desynchronization is introduced by Cortadella et al. [2] as a three-step process: Firstly, all flip-flops in a design are replaced by two latches. Secondly, controllers are inserted that control the latches and handshakes between neighbor controllers. Lastly, a completion detection mechanism is inserted between controllers. In this thesis, I will create a Tcl script for Synopsys Design Compiler [3] that takes the netlist of a synchronous circuit and completely automatically converts it to an asynchronous netlist. This thesis is written for Nordic Semiconductor and is intended as a proof of concept of desynchronization using the tools available. This thesis builds upon my literature study on desynchronization where different approaches are compared and discussed [4].

The main focus of this thesis is to get a functional script that is able to create an asynchronous netlist from a synchronous netlist. Hence, the techniques used for desynchronization are relatively simple. Other, more complex, techniques will also be introduced, and the advantages and disadvantages of different approaches will be discussed, but these will not be used in the final desynchronization process. The final result from this thesis serves as a starting point for further development of the desynchronization methodology.

2 Previous Work

Desynchronization is not a new concept and this thesis takes inspiration from multiple contributions. The main inspiration for the desynchronization methodology is Cortadella et al. and their paper [2]. Here they introduce the general idea of desynchronization as well as provide results from implementation. In addition, this thesis takes inspiration from many others who provide methods for asynchronous design such as Furber and Day [5] and their paper on circuits for four-phase handshakes, A Designer's Guide to Asynchronous VLSI by Beerel et al. [1], Principles of Asynchronous Circuit Design by Sparsø et al. [6], and many others.

I have also performed a specialization project before this thesis about techniques for circuit desynchronization [4]. This project was an initial literature search with a main focus on different options for controllers. The most relevant information from the project is included in this thesis, but there is a lot of information that is omitted from this thesis. An in-depth comparison of different methods for completion detection has also been performed by Gausdal at an earlier stage [7].

3 Background

3.1 Asynchronous Designs

Asynchronous circuits differ from synchronous circuits in that they do not have a global clock. Instead, they rely on local synchronization mechanisms such as local clocks or handshake signals. Asynchronous design can offer advantages such as lower power consumption and better performance in some scenarios [1]. However, asynchronous design comes with some challenges, the main one being a lack of Electronic Design Automation (EDA) tools. The tools available for synchronous design automate many tasks and help speed up the design process significantly compared to asynchronous design, making synchronous design the most efficient way to get a design from idea to production in most cases. Asynchronous design also requires a mentality shift from the designer, which can be difficult since most designers usually design synchronous circuits. If these challenges can be overcome, asynchronous design has the potential to give us improved performance and/or lower power. This chapter will cover different techniques for designing asynchronous circuits and why the selected methods for desynchronization are chosen.

3.1.1 Advantages of Asynchronous Design

Asynchronous circuits have some promising properties that can improve performance compared to synchronous circuits. Asynchronous circuits can be more robust and tolerant to different operating conditions, they can reduce the effect of process variation in manufacturing and give speedups and power savings in specific applications [2]. These advantages are not always guaranteed in an asynchronous circuit but depend on the circuit and its use cases.

A circuit that operates in many temperatures can benefit more from being asynchronous than one that operates in a more controlled environment. A synchronous circuit has to have a clock that can accommodate a slow down or speed up in transistor speed if the temperature changes. This can be done by having a clock speed slow enough for the worst-case speed or having a clock that dynamically changes its speed. The first alternative is the simplest approach but it will slow the circuit in all conditions. A constant worst-case clock speed can be a good solution for circuits not intended to work in an extensive range of temperatures. However, it may cause a significant performance loss when the circuit is intended to handle big temperature changes. On the other hand, a dynamically changing clock can perform well in low temperatures while still functioning in higher temperatures. Inserting a dynamic clock is not very simple and requires extensive analysis to guarantee that the clock speed always allows for correct behavior [2] [8]. There will also be some overhead for the extra logic to control the clock.

Asynchronous circuits can mitigate some of the effects of temperature changes. Temperature changes should not affect the circuit's behavior as long as the local synchronization mechanisms slow down or speed up by the same amount as the combinatorial logic. This was shown to be true by Yun et al. [9] where an asynchronous differential equation solver performed 48% faster than a comparable synchronous circuit. The testing was done at −30°C, 22°C, and 55°C while the synchronous circuit was designed to operate at 100°C. The asynchronous circuits improved performance is mainly down to the clock speed of the synchronous circuit being slower than needed at lower temperatures to guarantee correct behavior on 100°C. While this experiment shows great potential for asynchronous design it's important to note that this research was done in 1998. A synchronous circuit with a constant clock speed equal to the worst-case critical path is the most common design methodology, but there do exist techniques such as dynamic voltage and frequency scaling [8] that can improve the performance of a synchronous circuit. Nevertheless, the research shows that an asynchronous circuit can perform significantly faster compared to a synchronous circuit designed to work at higher temperatures than tested.

This temperature resistance may also be helpful in larger chips where local temperature changes may occur. An asynchronous chip can have a significantly hotter area with lower speed while the rest is faster in lower temperatures. A synchronous chip must either slow down the whole chip to accommodate the hot area or have multiple clocks with different speeds in different areas [1].

Yun et al. also showed that the same asynchronous circuit operated correctly with a lower supply voltage than the synchronous circuit. While the synchronous circuit was designed and optimized for 3.3V the asynchronous circuit could function as low as 1.85V at −30°C, 1.89V at 22°C, and 2.07V at 55°C [9] or around a 40% voltage decrease. A wider range of possible supply voltages can be helpful to reduce power consumption. It can also be necessary in cases where the supply voltage may change over time such as battery applications where the voltage can drop as it is discharged.

For mass produces chips, there is also a possibility that process variations make some of the chips slower than the rest [1]. This is something that has to be accounted for when calculating the clock speed of a synchronous circuit. Unless measurements are done on the actual chip after manufacturing, clock speed calculations must work with worst-case assumptions for process variations on the chip. Asynchronous circuits on the other hand can function with some process variations as long as the local synchronization mechanism is slowed down the same amount as the combinatorial logic. This assumption is generally reasonable as long as the synchronization mechanism's and combinatorial logic's physical placement are close.

Asynchronous circuits can also provide speed-ups in cases where the delay through the combinatorial logic is heavily data-dependent. For instance, the delay through a ripple carry adder block will differ depending on how long the carry-bit travels. A ripple carry adder is shown in Figure 1. Adding 4'b0111 + 4'b0001 = 4'b1000 will require the carry-bit to travel from full-adder 0 to full-adder 3 in the figure. 4'b1000 + 4'b0001 = 4'b1001 will not produce any carry-bits. The time from an input to a stable output of a ripple-carry block will be the sum of two factors: the delay through the longest carry chain and the delay through the full adder to receive the last carry-bit of this chain. An asynchronous circuit with a completion detection able to predict this delay can be faster than a synchronous circuit with a clock speed equal to the worst-case delay, as shown by Nowick [10]. This can give the asynchronous circuit a lower average delay than the synchronous circuit. The implementation of the completion detection will be discussed further in 3.2.2. It is also important to note that synchronous circuits implemented with more complex techniques such as time-borrowing [11] can give a delay closer to average than worst-case.

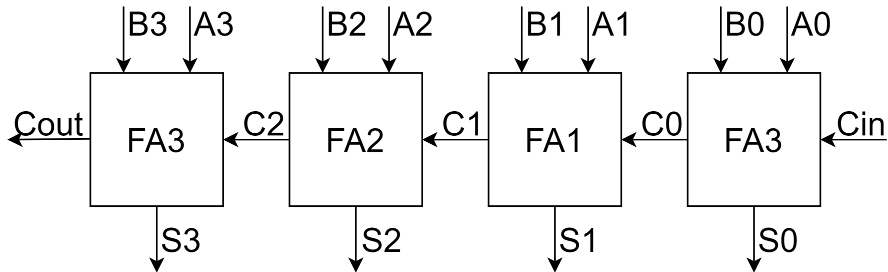


Figure 1: Four-bit adder constructed from four full-adders.

This chapter has shown a significant potential for asynchronous circuits in specific scenarios. Asynchronous circuits have the ability to mitigate temperature changes and process variations, as well as the capacity to provide speed-up for circuits with data-dependent delays. Moreover, asynchronous circuits can handle a broader range of supply voltages which can be advantageous in battery-driven applications. However, these advantages do not apply to all applications and the performance gains will vary widely. The choice between synchronous and asynchronous circuits must be taken case by case, considering environmental conditions, performance requirements, and power constraints.

3.1.2 Data channels

When transferring data between blocks inside an asynchronous system, there is no global clock that indicates when data should be sent. Instead, there is a need for a mechanism to indicate when the data is valid. If this mechanism is not in place different parts of the system have no way of knowing when the correct data is available. A handshake protocol is one option for this mechanism. The handshake signal can either be independent of the data or baked into the data signals. This opens the possibility for different data-channel designs, the two most common being bundled-data

channels with separate signals for the handshake and 1-of-N channels with the handshake partially integrated into the data [1]. This chapter will introduce these two and compare the advantages and disadvantages of each approach. The different ways of implementing handshake signals are discussed in 3.1.3.

A bundled-data channel is the most straightforward approach when adapting from a standard synchronous design. It consists of a single-rail bus comprising one wire for each data bit and a single request and acknowledge line. Figure 2 shows a bundled-data channel between two blocks with one request and one acknowledge line. This approach gives a low area overhead with only two wires extra for request and acknowledge. The request and acknowledge carry the handshake signals. In addition, there is a need for controllers that deal with the handshake signals, this is discussed in more detail in 3.2.3. The low overhead and only one wire for each bit give the bundled-data channel a relatively low area per bit. In a bundled-data channel, the data bits are allowed to change values between two transfers as long as the value is stable and valid when a request is sent out. Figure 3 shows a transaction on a 4-bit bundled data channel. The request signal does not go high before all data bits are stable.

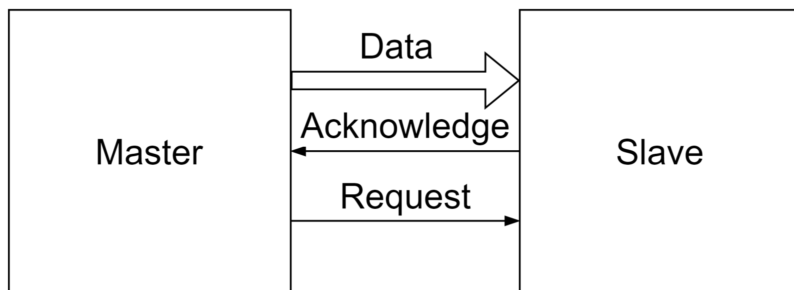


Figure 2: Bundled data channel.

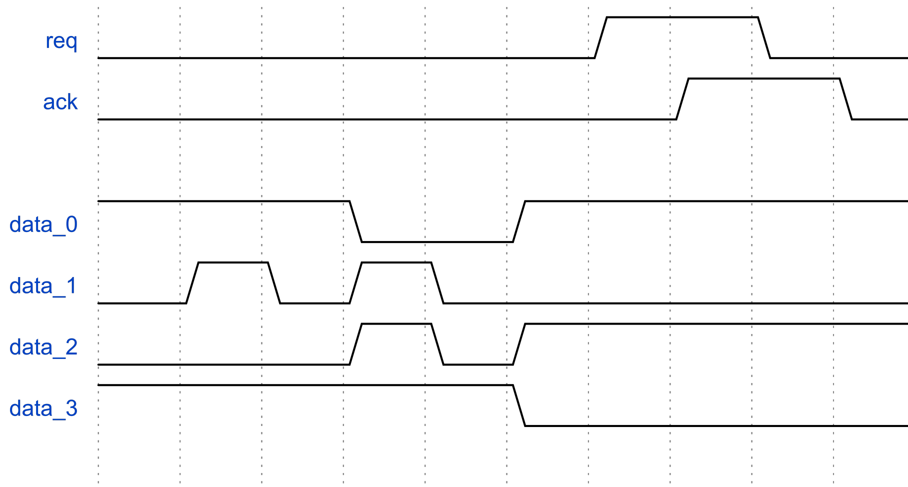


Figure 3: Transaction on a bundled-data channel.

A 1-of-N data channel does not need a request signal, instead, the validity guarantee of the data is built into the data lines. This is done using N data lines to transfer $\log_2(N)$ data bits and only allowing one data line to be high simultaneously [1]. The 1-of-2 channel, also called a dual-rail channel, is the most well-known configuration using two data wires, data_0 and data_1. Of the four possible combinations of values on the two wires, only three are allowed. data_0 and data_1 both being low, indicating no valid data, data_0 high and data_1 low indicating a valid zero and data_0 low and data_1 high indicating a valid one. Both data lines high is not allowed and should never happen. Figure 4 shows how data is transferred in a dual-rail channel. The transactions are a 0 at "a" followed by a 1 at "e". Acknowledge going high at "b" and "f" indicates that the data has been received. The figure shows that only one wire is high at a time, and all wires go low between transfers.

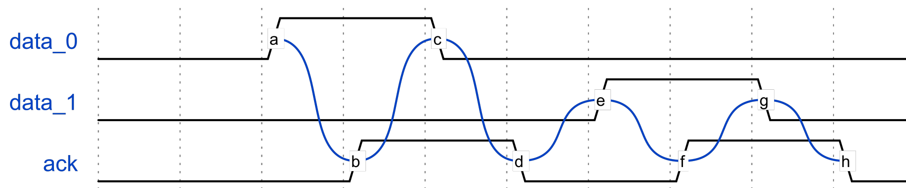


Figure 4: Transaction on a dual-rail data channel [4].

Similarly, a 1-of-4 channel will have four wires and can send 0, 1, 2, or 3 by rising data_0, data_1, data_2, or data_3, respectively. Any state containing more than one high wire is illegal. This lets a receiver know that data is valid whenever one wire goes high. Data transfer on a 1-of-4 channel is shown in Figure 5. The data channel transfers a 0 at "a" followed by a 2 at "e". The receiver confirms that data is received by rising to acknowledge at "b" and "f". It is possible to extract a request signal by or-ing all N wires in the channel, this can be helpful to get one signal that shows whether data is valid or not.

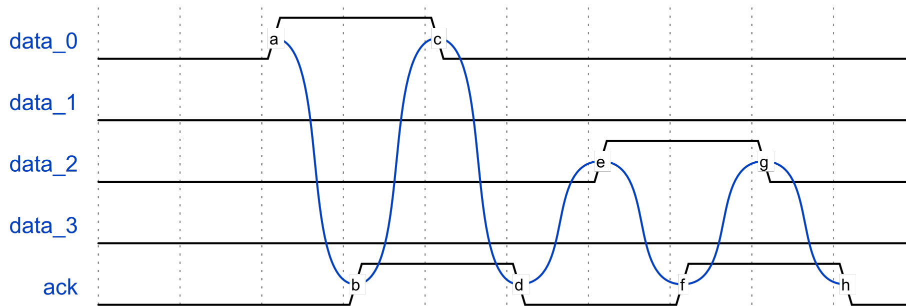


Figure 5: Transaction on a 1-of-4 data channel [4].

The 1-of-N does require more area per bit transferred but can give some power advantages compared to a bundled data channel since only one wire changes each data transfer. The data wires in a 1-of-N are not permitted to change values between transfers. A glitch will be interpreted as data and can cause errors in the system.

Standard synchronous circuits normally use data channels very similar to bundled-data channels. However, synchronous circuits use a clock as a synchronization mechanism instead of the request and acknowledge signals. This means the combinatorial logic does not need to be changed when converting from a synchronous circuit to an asynchronous circuit using bundled-data channels. In contrast, a 1-of-N will require a complete redesign of the logic to get the correct behavior. Depending on the logic, this redesign can be complicated and may counteract the possible power savings. This thesis is intended to be a proof of concept of desynchronization, therefore bundled-data channels are chosen because of the simple implementation. However, this might not be the best choice in all situations. A more in-depth analysis between bundled-data and 1-of-N channels should be performed at a later stage to ensure that the best option is selected.

3.1.3 Handshakes

As mentioned above, a synchronization mechanism is needed when transferring data between blocks in an asynchronous system. The synchronization mechanism is used between blocks in a system to indicate when data is valid. A common way to achieve this is through handshake signals. A handshake usually consists of a request and an acknowledge signal. The function of these signals depends on whether the channel is a push or a pull channel. In a push channel, the sender will indicate when data is ready by sending a request, and the receiver will indicate when it has received the data by sending an acknowledge. A pull channel switches the request and acknowledgment, the receiver will send a request when it is ready to receive data, and the sender will send an acknowledge once the data is ready to be transferred. This thesis will use push channels when

discussing handshakes as this is the most common in the previous works, used by Cortadella et al. [2] among others. However, it is possible to change this if there is a specific need for a pull channel.

There are two main protocols for handshakes with request and acknowledge signals, two-phase and four-phase handshakes. In a two-phase handshake, there is no difference between a request and acknowledge rising and falling. The information is transferred in the transition of the signals. Both a falling and rising edge on the request wire indicate that new data is available. This is also true for the acknowledge wire where both a falling and a rising edge indicates that data has been received [1]. Figure 6 shows a data transfer using a two-phase handshake. This protocol has the advantage that no transitions on the handshake wires are useless. However, due to the level-sensitive nature of the latches in a desynchronized circuit, there will be a need for conversions from two-phase to four-phase inside the controllers [5].

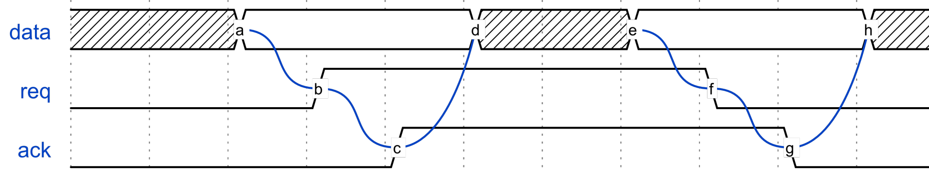


Figure 6: Two-phase handshake [4].

A four-phase handshake distinguishes between a falling and a rising edge on the request and acknowledge wires. Only one of the two edges will carry information. It is possible to choose whether the falling or the rising edge holds information [5]. This thesis will use the rising edge since most previous work has chosen this too. A data transfer using a four-phase handshake is shown in Figure 7. The request wire goes high when the data is valid, followed by the acknowledge wire going high, indicating that the data has been received. The handshake completes with the request wire followed by the acknowledge wire going low to be ready for a new transaction. This last transition on the request and acknowledge wires is redundant and only needed to prepare for the next transaction. The transition can be used to simplify the latch controllers.

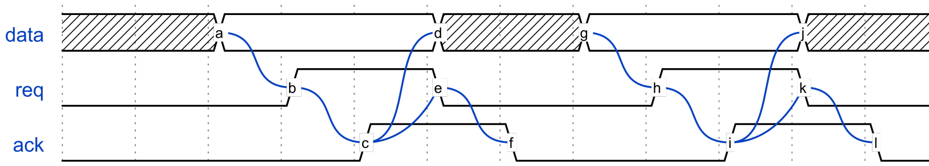
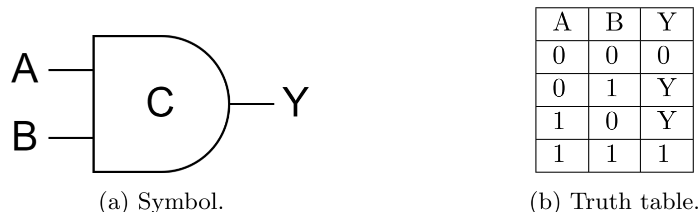


Figure 7: Four-phase handshake [4].

Both bundled-data and 1-of-N channels can be implemented with both a two-phase and four-phase handshake depending on the specific need for the design. A two-phase handshake implemented with a 1-of-N channel eliminates the need to reset all wires to zero between transfers. Four-phase handshakes require less conversion to control the latches due to the redundant return to zero transaction, leading to simpler control logic. This thesis will use bundled-data channels and four-phase handshakes due to the advantages these provide.

3.1.4 Muller C-element

It is crucial that the request and acknowledge wires are valid at all points to ensure correct behavior. The request line rising too early can cause invalid data to propagate, while an early rise on the acknowledge line could cause data to be deleted before it is consumed. This can become a problem in places where multiple request or acknowledge signals are combined into one. The output of a combined request or acknowledge signal should not be high before all inputs are high, and it should stay high until all inputs are low. This property is needed to preserve the handshake protocol. A common solution to this problem is the Muller C-element [6]. A C-elements will only change its output value once all inputs have the same value. If all inputs are one, the output will go to one. If all inputs are zero, the output will go to zero. The C-element will have an unchanged output



(a) Symbol.

A	B	Y
0	0	0
0	1	Y
1	0	Y
1	1	1

(b) Truth table.

Figure 8: Truth table of the Muller C-element with two inputs [4].

value when the inputs are different. The symbol and truth table of a C-element is shown in Figure 8.

This function is very useful when controlling handshake signals between blocks in a system. The simplest possible control circuit for a four-phase handshake consisting of a C-element and an inverter is shown in Figure 9 and is called a Muller pipeline [6]. The figure shows three stages, marked by dotted lines, and how they are connected. Each of the stages req_in and ack_in on the left side and req_out and ack_out on the right side. The C-elements ensure that neither req_out or ack_in go high before req_in is high and ack_out is low. Req_out and ack_in will stay high until req_in has gone low and ack_out has gone high. The enable signals control latches.

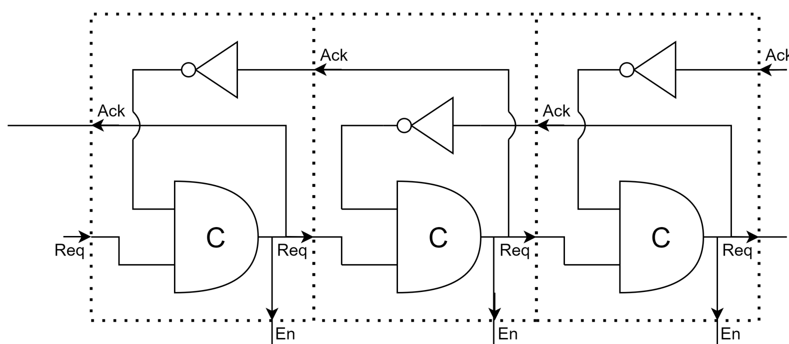
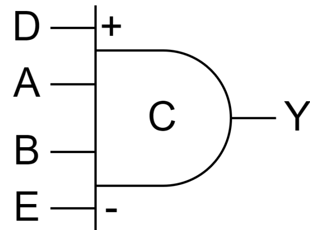


Figure 9: Muller pipeline consisting of a C-element and an inverter for each stage [6].

More complex implementations of C-elements, called asymmetric C-elements, allow some inputs only to affect the output transition in one direction. An asymmetric C-element with four inputs, and its truth table, is shown in figure 10. In this case, A, B, and D have to be one for the output to change to 1, and A, B, and E have to be 0 for the output to change to 0. D is ignored when the output transitions from 1 to 0 and E is ignored when transitioning from 0 to 1. Asymmetric C-elements will not be used in this thesis for simplicity but might be useful for some applications.

This thesis will also not consider C-elements with more than two inputs for simplicity's sake. A two-input C-element can be implemented on a transistor level, as shown in Figure 11, or on a gate level using two- and three-input NAND gates, as shown in Figure 12. The gate library used does not contain a C-element, the C-element used is thus constructed using four NAND elements. There is also a need for a reset port on the C-element to guarantee a known value on reset. Chapter 3.2.3 shows a need for two different C-elements, one that resets to 0 and one to 1. These are shown in Figure 13a and Figure 13b respectively.



(a) Symbol.

A	B	D	E	Y
0	0	0	0	0
0	0	0	1	Y
0	0	1	0	0
0	0	1	1	Y
0	1	0	0	Y
0	1	0	1	Y
0	1	1	0	Y
0	1	1	1	Y
1	0	0	0	Y
1	0	0	1	Y
1	0	1	0	Y
1	0	1	1	Y
1	1	0	0	Y
1	1	0	1	Y
1	1	1	0	1
1	1	1	1	1

(b) Truth table.

Figure 10: Symbol and truth table of an asymmetric C-element with four inputs [4].

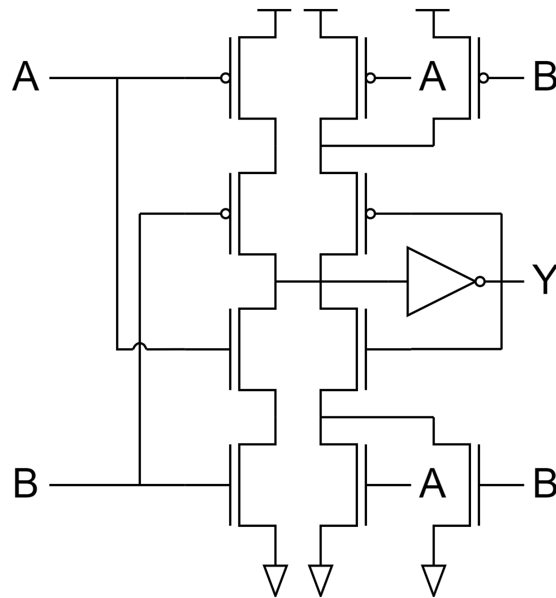


Figure 11: C-element implemented using transistors [4].

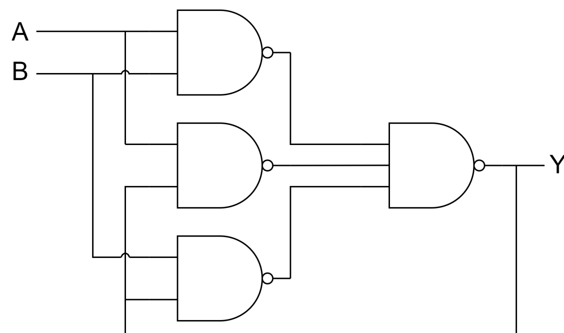
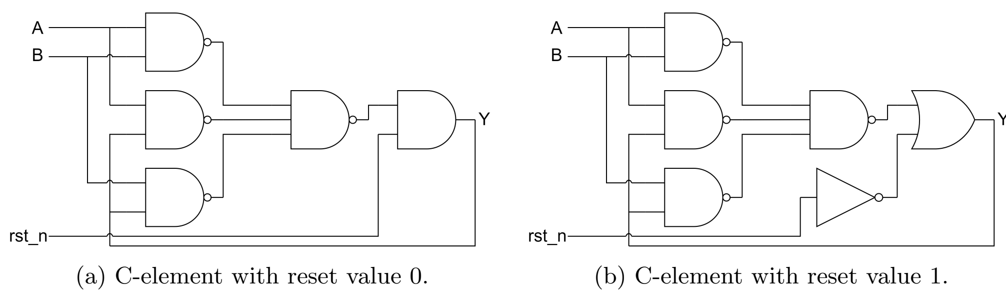


Figure 12: C-element implemented using two- and three-input NAND gates [4].



(a) C-element with reset value 0.

(b) C-element with reset value 1.

Figure 13: C-elements with active low reset.

3.1.5 Difficulty in testing due to lack of tools

As discussed in this chapter, there are some potential advantages to choosing asynchronous circuit design. Despite the advantages of asynchronous circuits, most circuits today are synchronous. There are many reasons for this, synchronous design has been the leading choice for a long time meaning most designers have lots of experience and knowledge in synchronous design. A switch to asynchronous design will require a mentality change in the designers due to the different synchronization mechanisms [2]. Another difficulty is the lack of EDA tools to aid the design process.

The EDA tools available for synchronous design can automate tasks such as placement, routing, and verification. Some of these tasks will be very similar for synchronous and asynchronous design, such that some tools intended for synchronous circuits can also be used for asynchronous circuits. However, there is no guarantee that the tool will function as intended, and workarounds are often needed to trick the tool into working with asynchronous design [1].

3.2 Desynchronization

Chapter 3.1 has introduced the potential of asynchronous circuits. Asynchronous circuits can provide performance gains and robustness compared to synchronous circuits in specific scenarios. However, these advantages are not always applicable and can be hard to extract fully. Designing an asynchronous circuit is also completely different from designing a synchronous circuit and requires a mentality change from the designer. In addition, asynchronous design is limited by the relative lack of EDA tools making the design process significantly more manual than synchronous design, as discussed in 3.1.5.

Desynchronization is a concept introduced to get the best of both worlds. The idea is to start out with a synchronous circuit that is designed and verified using the many EDA tools that exist, and later convert this circuit into an asynchronous circuit in an attempt to get the performance gain from asynchronous circuits and the ease of the design of synchronous circuits. This chapter will introduce the desynchronization process and discuss the different steps of desynchronization, implementation methods, and the advantages and disadvantages. Furthermore, the chapter will give a general overview of situations where desynchronization can yield performance gains and where there is little to no performance to gain.

The general process of desynchronization can be broken into three main steps [2]:

1. Remove all flip-flops and insert two latches instead.
2. Generate completion detection for the combinatorial logic.
3. Insert controllers connected to latches, completion detection, and neighbor controllers.

The three steps are described in more detail in separate sub-chapters below. After these steps, there is no longer a need for a global clock this is removed along with all clock propagation and clock gating logic. Figure 14 shows a simple block diagram before and after desynchronization. Comp. Det. indicated the completion detection.

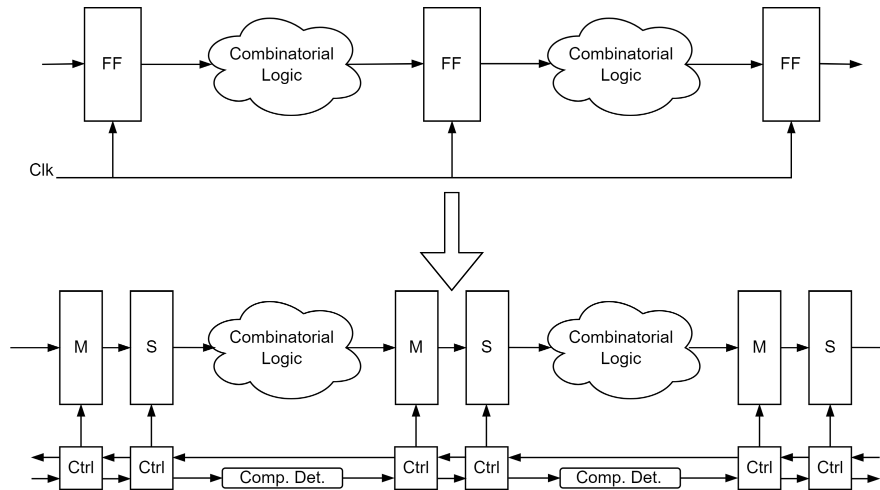


Figure 14: A synchronous circuit converted to an asynchronous circuit using desynchronization [4].

3.2.1 Flip-flop conversion

The conversion from flip-flops to latches is done by revealing the underlying master and slave latches inside a flip-flop. This conversion is done to get individual control over each of the two latches. The main difference between a latch and a flip-flop is that a latch is level-triggered while a flip-flop is edge-triggered. A latch will be transparent and pass the input to the output when the enable signal is high, and be opaque and keep the output unchanged when the enable signal is low. A flip-flop will only pass the input to the output at the positive edge of the enable signal. The enable signal of a latch could be inverted such that it is opaque when enable is high and transparent when enable is low, but this thesis will use latches that are transparent when the enable signal is high as this is what is available.

It is possible to insert one latch instead of two but this may cause every other combinatorial logic block to stall while the others work. This stalling happens if data is fed in as fast as possible, as shown in Figure 15. Here, latch 2 and 4 must be empty to be ready to receive the inputs of combinatorial logic 1 and 3. Empty latches are marked with a white dot, full latches are marked with black dots. Since these latches are empty there is no data on the input of combinatorial logic 2 and 4 and these sit idle. In applications where data is fed rarely and there are no strict timing requirements, it might be possible to use only one latch for each flip-flop to save some area. This is outside the scope of this thesis but should be investigated at a later point.

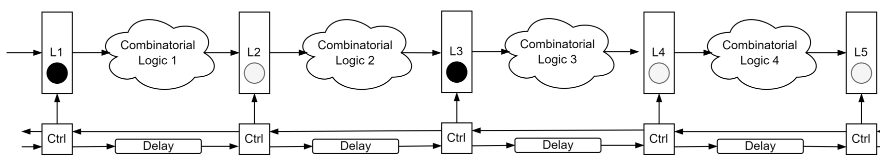


Figure 15: Asynchronous pipeline where registers have been replaced by one latch [4].

After the flip-flops have been converted to two latches it is possible to move the latches to balance the pipeline stages. Replacing one flip-flop with two latches gives more freedom in the placement of each latch which can give performance boosts as shown by Chinnery and Keutzer [12].

3.2.2 Completion detection

The completion detection is crucial to get correct behavior after desynchronization. It is also a significant factor in the performance of the desynchronized circuit. In this case, the completion detection is inserted between the request out from one controller and the request in of another controller, as shown in Figure 16. The job of the completion detection is to ensure that the recipient

receives the request signal at the correct time. The completion detection should not propagate the request signal too early as this can cause errors in the system, or not too late as this could cause unnecessarily low speeds. Some completion detection mechanisms rely on additional inputs besides the request signal.

The simplest completion detection is a delay chain with one input and one output. A desynchronized circuit with a simple delay chain is shown in Figure 17. The delay chain has to be long enough to accommodate the worst-case delay through the combinatorial logic to guarantee correct behavior for all data inputs. The calculations for the length of the delay chain can be found in chapter 4.5.

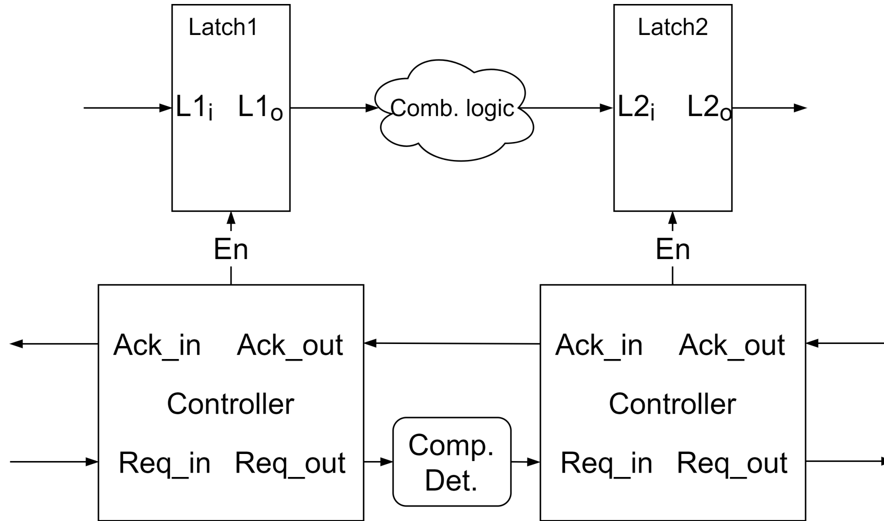


Figure 16: Completion detection inserted between two controllers [4].

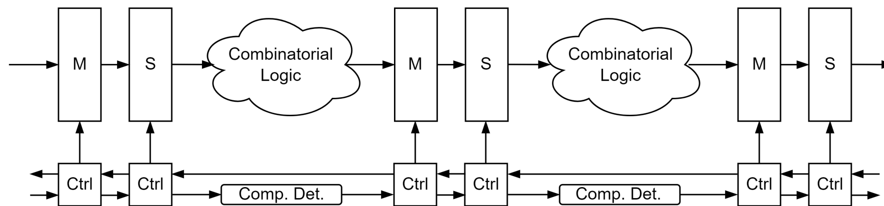


Figure 17: Desyncrhonized circuit with simple delay line as completion detection [4].

A constant length delay chain is very simple and can give a faster-performing circuit. The speed will be similar to a synchronous circuit with a clock speed equal to the worst-case delay. However, a desynchronized circuit's worst-case delay can be somewhat lower due to the better tolerances of manufacturing and environmental variables as discussed in 3.1.1. This difference in worst-case performance typically gives a desynchronized circuit a 1.2 to 2 times faster speed than the synchronized circuit it is based on [2]. The speed-up might be different for synchronous circuits that use more advanced techniques such as time-borrowing. Construction of the delay line can be done with a chain of multiple gates, such as inverters, or with dedicated delay elements.

The delay line can be implemented both symmetric and asymmetrically. A symmetric delay has the same delay from the change in input to the change in output for both rising and falling edges. An asymmetric delay has a different delay depending on if the input is a rising or a falling edge. A four-phase handshake can benefit from an asymmetric delay as the falling edge of the request does not transfer any information and does not need to be delayed. However, an asymmetric delay is slightly more complex to implement.

While a delay line can be effective, more complex completion detection mechanisms can often offer better performance, especially in cases where there is a big difference between the worst-case delay and the average delay through the combinatorial logic [1] [7]. Some examples are speculative delay lines [10], efficient asymmetric and symmetric speculative delay lines [13], and current sensing

completion detection [14]. All of these have shown the ability to improve performance at the cost of some extra logic making the overhead bigger. There are also other possible implementations of completion detection, but these are out of the scope of this thesis.

A speculative delay line is based on a simple delay line but allows the delay through the block to vary based on inputs. It is introduced by Nowick [10] and was originally used in an asynchronous adder. An implementation of the speculative delay line is shown in Figure 18, req_out, and req_in are as indicated in Figure 16. The speculative delay line aims to improve performance by shortening the delay between two controllers when the result from combinatorial logic is stable faster than in the worst case. This is done by using multiple delay lines of different sizes and some logic to determine which one should be used. More specifically, all delay lines have the same input at get request at the same time. The abort detection networks then ensure that outputs from the too-short delay lines are stopped before propagating. This approach works for any number of delay lines of any length, as long as the abort detection is faster than the delay.

A speculative delay line is best suited for applications where it is relatively simple to detect when the result from the combinatorial logic will be stable. This can be a situation where one or a handful of signals determine whether a short or long delay is needed. If the condition can be implemented with a low number of gates it gives a small area and power overhead from the abortion detection. It is also possible to reduce the area overhead by using the previous smaller delay line to create the subsequent larger delay line [15]. This can also reduce power by not propagating request through multiple individual delay lines.

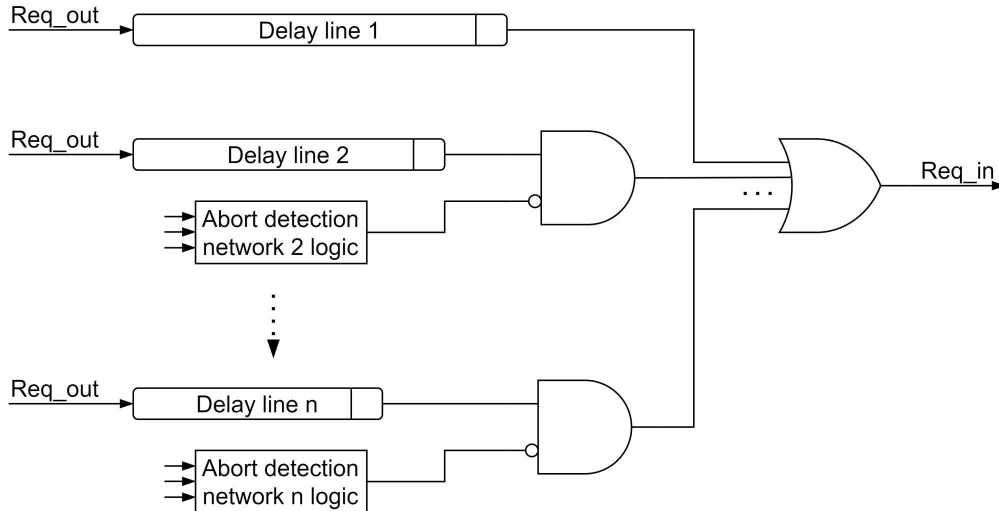


Figure 18: A general implementation of a speculative delay line [1].

Efficient asymmetric and symmetric speculative delay lines, introduced by Tugsinavisut et al. [13], are shown in Figure 19 and Figure 20, req_out and req_in are indicated in Figure 16. Both are implemented by connecting multiple delays and delay controllers in a chain. ADL and SDL in asymmetric and symmetric delay lines respectively. The delay controllers, indicated by ADLC and SDLC, are shown in 21 and 22.

Both the asymmetric and symmetric versions need some extra logic to control the sel signal. The logic needs to detect when the result from the combinatorial logic is valid and drive sel high. Each controller will use sel to decide whether to send the request signal to the next delay line by rising NR_i or towards Req_in by rising LD_i. This approach stops the request signal from propagating through further delays when sel is high and therefore saves power compared to the speculative delay line discussed above. This template is discussed in more detail in the paper by Tugsinavisut et al. [13].

The last completion detection that will be presented in this thesis is current-sensing completion detection (CSCD), proposed by Dean et al. [14]. The idea of CSCD is that combinatorial logic with a stable output will draw less current than the same circuit while the output is still being

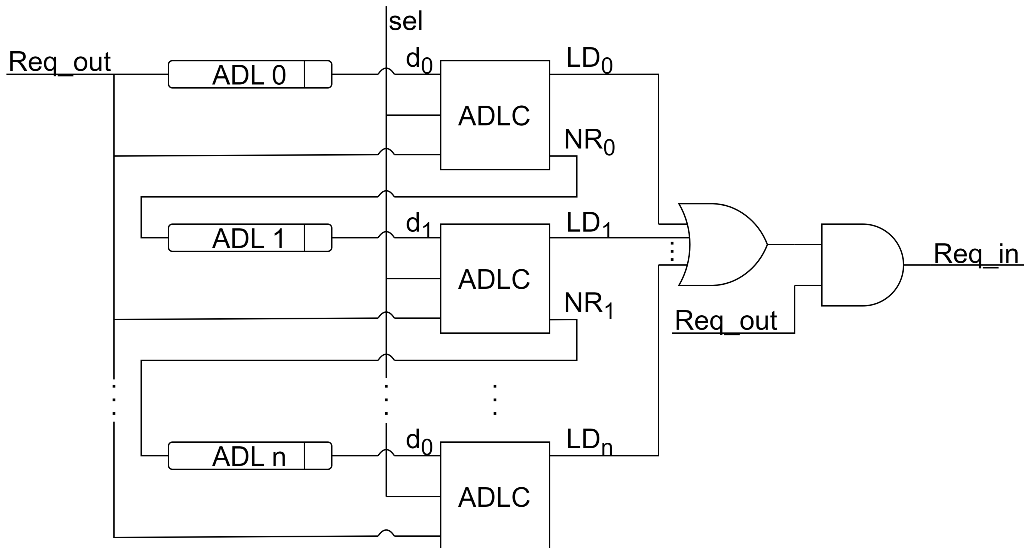


Figure 19: An efficient asymmetric speculative delay line [1].

calculated. Figure 23 shows a block diagram of a CSCD implementation, combinatorial logic, two current sensing blocks, and a minimal delay generator (MDG). The two current sensors will measure current in the combinatorial logic and give an output signal when this current is below a threshold. The MDG might not be necessary in all designs but is intended to protect from early propagation in cases where there is some ramp-up time. This can for example be cases where the current is low early in calculation causing the current sensors to trigger even though the output is not stable. Request will then be halted until both the current sensors and the MDG have given an output. There are multiple different ways of creating the current sensors, some are shown in the paper by Dean et al. [14], but this is out of the scope of this thesis.

There are a number of factors influencing the choice between the completion detection templates presented above, and other completion detection templates. Performance requirements, area requirements, implementation complexity, and timing requirements among others have to be considered when choosing. There is not one single solution that will be the best all the time. However, most of the completion detection templates are exchangeable meaning it is possible to use a simple one, such as a delay line, and change it later if the desynchronized circuit does not meet the requirements. It is also possible to use different completion detection templates for different parts of the circuit. This permits more complex and faster completion detection in regions when needed and a lower area for regions with more lenient requirements. For simplicity, a simple delay line will be used as a starting point for this thesis.

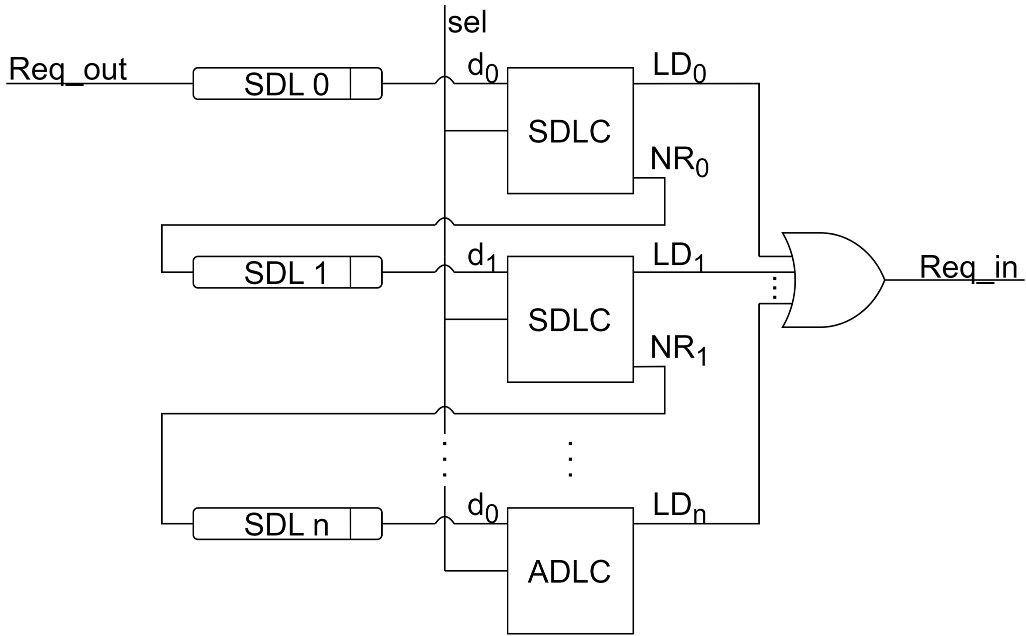


Figure 20: An efficient symmetric speculative delay line [1].

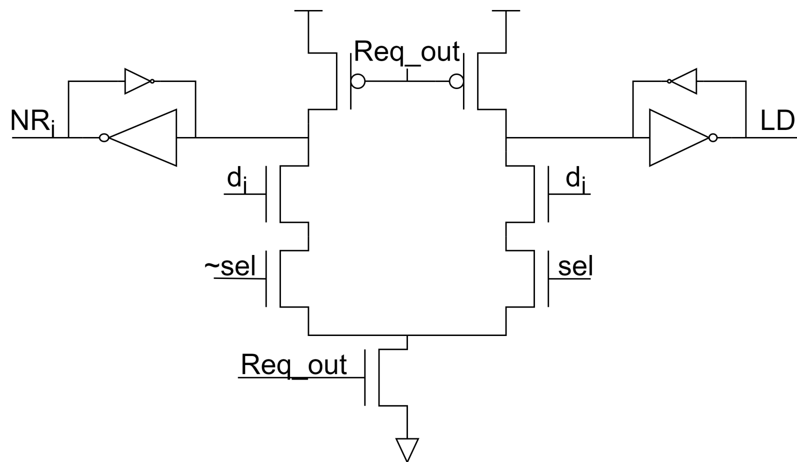


Figure 21: Asymmetric delay line controller (ADLC) [1].

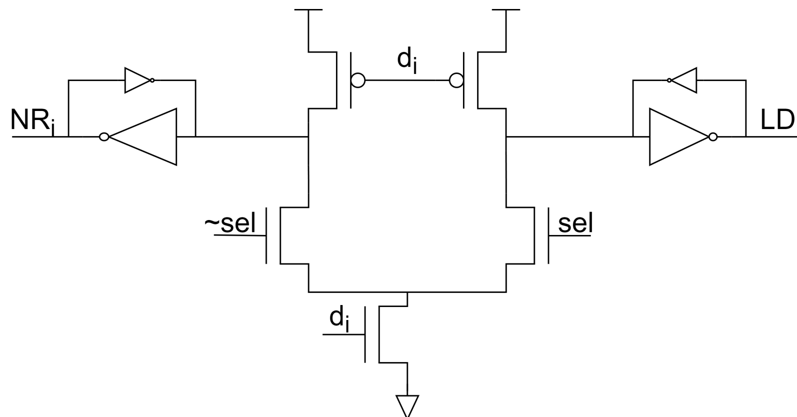


Figure 22: Symmetric delay line controller (SDLC) [1].

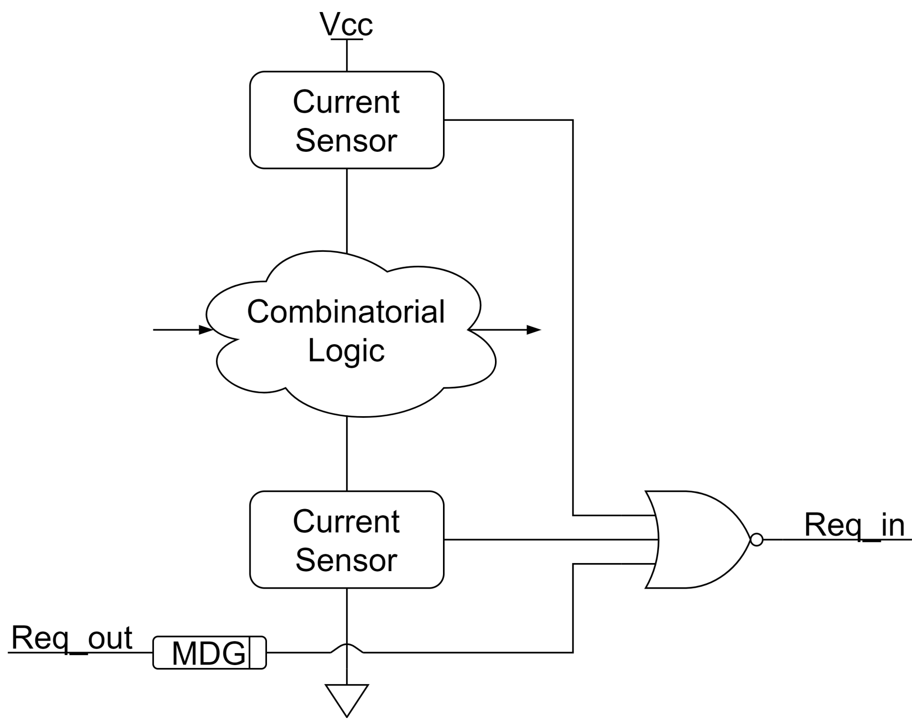


Figure 23: Current sensing completion detection [14].

3.2.3 Controller

The last step in the desynchronization process is inserting a controller for each latch. The controllers are responsible for ensuring that latches are transparent and opaque at the correct moments to get the correct behavior. The latches should not be closed before the data is valid and not open before the next latch has received the data. They are also responsible for the handshake signals between neighbor controllers. Controllers can be designed in multiple ways, depending on requirements. Usually, a trade-off between area, speed, and power has to be made when choosing. An in-depth comparison between the most prominent controllers is made by Kleppe [4]. This thesis will present a few of the ones covered by Kleppe and give a general overview of the options available. Similarly to completion detection, controllers are also mostly exchangeable meaning a simple controller can be exchanged later if the design requirements, such as power, area, and performance, are not met. Different controllers implemented with the same handshake protocols can also be used with each other. Combining controllers can help meet requirements, allowing smaller and slower controllers in some regions and bigger but faster controllers in others. Faster controllers can then be used in regions where speed is crucial while smaller controllers can be used to save area and power in regions where speed is not crucial. Figure 24 shows a block diagram of a simple controller implementation. This chapter will only present controllers based on a four-phase handshake as stated in 3.1.3, but much of what is stated is applicable when using two-phase handshakes also.

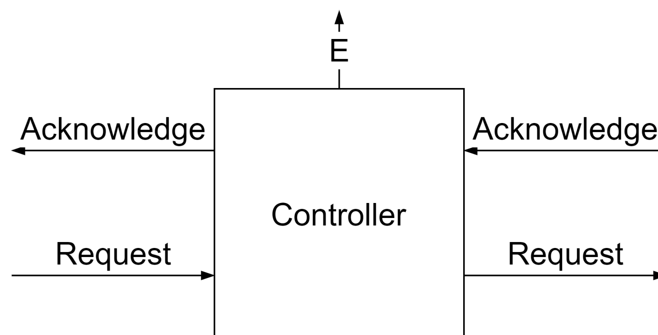


Figure 24: Block diagram of a controller with one latch enable signal, and re quest and acknowledge on input and output.

Signal transition graphs (STG) are often used to describe the properties of a controller. An STG is a representation of the relationship between different signals. The arrows between different signals show which order they go high or low in, depending on the "+" or "-" signs, while the black dots indicate the initial state. In the STG in Figure 26 E will be high, meaning the latch is open, it will not close before request_in (Ri) goes high. STGs, also called marked graphs, are described in more detail by Commoner and Holt [16]. These STGs can be converted to a gate implementation either manually or automatically using tools such as Workcraft [17]. One STG can be implemented in different ways depending on the available gates. The STG is very useful when comparing controllers since this indicates the relationship between inputs and output changes. This thesis will present three controllers, the semi-decoupled controller [5], the blade controller [18], and the MOUSETRAP controller [19]. These are three very different controllers which give a general perspective of which options exist, they are also explored in more detail by Kleppe [4]. However, there are many other options with different attributes that all have their use cases.

The semi-decoupled controller introduced by Furber and Day [5] is introduced alongside the simple, and the fully-decoupled controller. All of these are relativley small controller based on a four-phase handshake. The simple controller has some unfavorable properties due to the connection between the left and right sides of the controller. In this context the left side is request_in and acknowledge_in while the right side is request_out and acknowledge_out. These properties include that simple controllers in a FIFO allow for at most every other latch to contain data. The semi-decoupled controller solves some of these problems by somewhat decoupling the left and right sides. The decoupling allows the left and right sides to operate more independently. Lastly, the full-decoupled controller aims to improve performance further by removing all coupling between the left and right sides. (As noted by Cortadella et al. [2], the fully-decoupled controller does not

fully decouple the left and right sides, rather it decouples the rising edges of the left and right sides.)

Figure 25 shows a possible gate implementation of an even semi-decoupled controller, the STG used to generate the gate implementation using Workcraft is shown if Figure 26. Gate implementation and STG for the simple and fully-decoupled controllers can be found in [4], and a more in-depth comparison can be found in [5], [2], and [4]. The main difference between the controllers is the order in which signals change. The semi-decoupled controller will have a closed latch (E = 0) until request in has gone low, the fully-decoupled controller can open its latch after raising acknowledge in. These differences allow some controllers to operate faster. The faster controllers will typically require a bigger area than the slower controllers.

Two variants of the controllers are needed to ensure correct behavior after resetting, one even and one odd [2]. The only difference between the controllers is that the latch connected is transparent for even and opaque for odd controllers after resetting. Since the even latches are transparent after reset they do not need to be reset, only odd latches need to be reset. Even and odd controllers have the same STG but with different initial markings. Workcraft has the functionality to generate correct gate implementations with correct reset functions. It turns out that the only difference is the reset function of the C-elements. The even controller needs U4 to reset to one and U6 to zero while the odd controller needs both C-element to reset to zero.

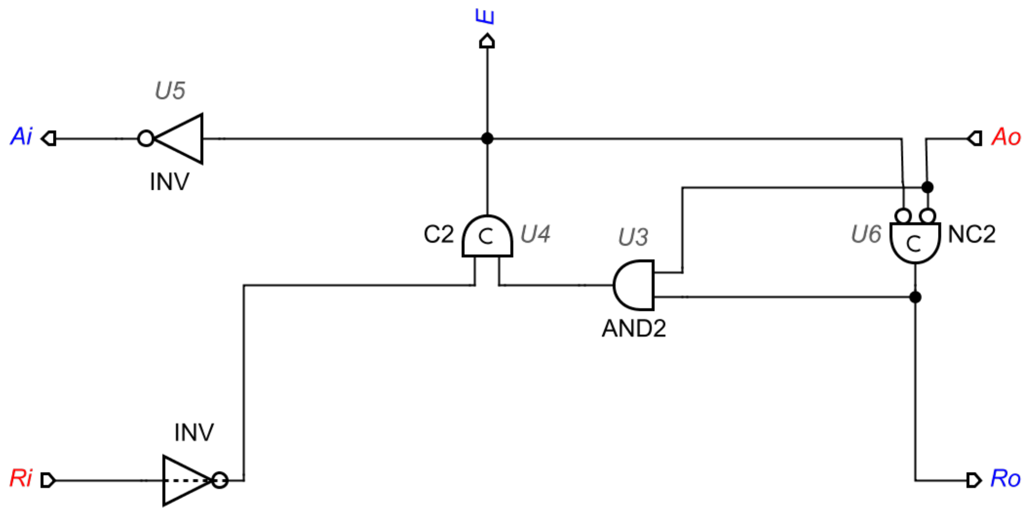


Figure 25: Gate implementation of the semi-decoupled controller [4].

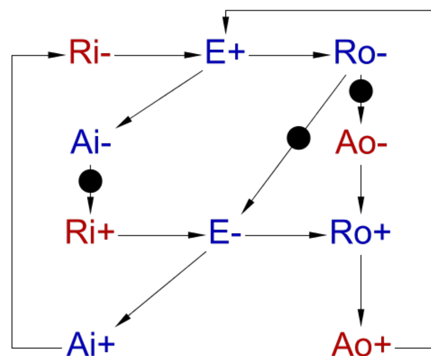


Figure 26: STG of the semi-decoupled controller[4].

Hand et al. [18] introduce the Blade controller which employs error-detecting logic to allow faster average speeds. Figure 27 shows a block diagram of the controller together with the error-detecting logic. The controller is based on two configurable delay lines, one of length δ and one of length

Δ. After a delay of δ the controller will send a request out, the controller will then look for an error in a period Δ after this. An error is detected if the input value of the latch changes in this period, meaning the calculation was not complete after δ. The next controller will get information if an error is detected or not. If an error is detected, the next controller will extend its calculation phase to ensure that the error is not propagated. The next controller will continue as normal if no error is detected.

Blade allows the first delay, δ, to be shorter than the worst-case delay of the combinatorial circuit. Correct behavior is guaranteed as long as $\delta + \Delta$ is longer than the worst-case delay. This allows the controller to detect any error and instruct the next controller to extend. The optimal length of each of the delays varies and needs to be found. The implementation of the Blade controller and the lengths of the delay chains are described in more detail in [18]. It is also important to note that the Blade controller requires four extra signals to operate, meaning some sort of conversion is needed when using Blade together with other controllers.

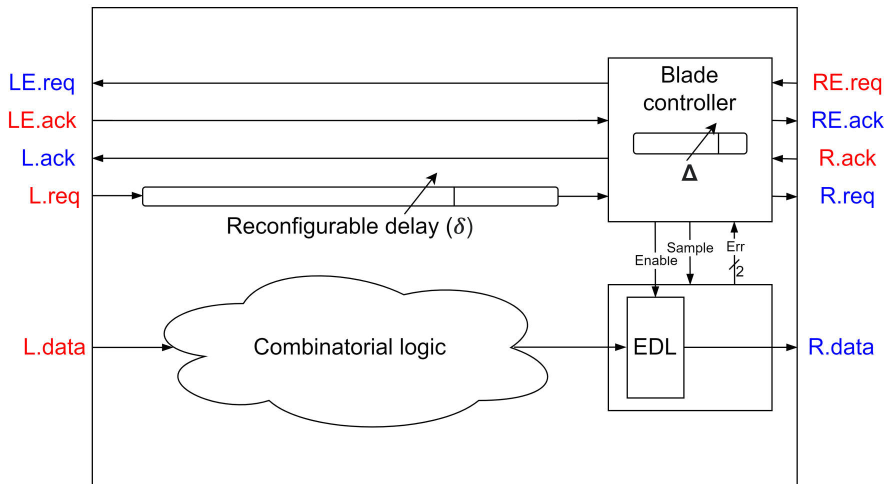


Figure 27: Block diagram of a Blade controller with error detecting logic [4].

MOUSETRAP, proposed by Singh and Nowick [19], is a small controller intended for high-speed uses. The controller consists of a single XNOR element and is area efficient compared to the other controller options, as shown in Figure 28. The latch will be enabled whenever request and acknowledge out has the same value. The controller does not distinguish between the rising and falling edge of the request and acknowledge signal, meaning it employs a two-phase handshake. A conversion will then be needed if this controller is to be used together with a four-phase handshake. There are also some problems when connecting multiple controllers in a circle configuration. The controllers have causality arcs that go beyond the neighbor controllers meaning it is susceptible to deadlocks [20]. These causality arcs make it impossible to create a ring of two MOUSETRAP controllers as this would be susceptible to deadlocks. The MOUSETRAP controller is best suited for simple linear setups where high speeds are desired.

The choice of the controller will be a big factor in the performance of a desynchronized circuit. This chapter has presented some of the options, but multiple other controllers are available. Each design will have a different best option. A small slow controller can be good when area is important, while a bigger faster controller can be better when speed is important. It is also possible to mix controllers to get speed in regions where needed and save area in regions where speed is not important. This thesis will use the semi-decoupled controller as a starting point due to the conclusion in the preceding project by Kleppe [4]. The semi-decoupled controller offers a middle ground between area and speed and should be well-suited for this thesis's work.

3.2.4 Fork and Join structures

Some circuits have fork and join structures where a path forks into multiple paths or where multiple paths join into one. Figure 29 shows a situation where a circuit forks into two paths before joining

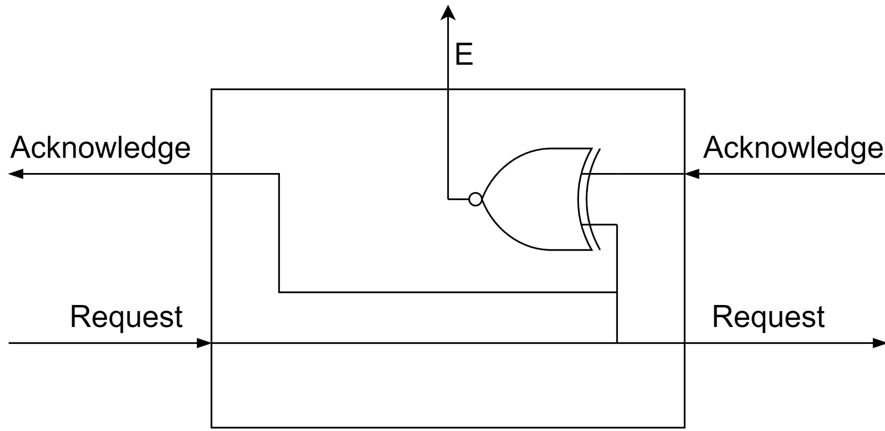


Figure 28: MOUSETRAP controller consisting of a single XNOR element [4].

together. When the circuit forks, one controller will send its request signal to two or more other controllers and receive their acknowledge signals. It is crucial that the first controller does not open its latch before receiving acknowledge from all the connected controllers. An early deletion can cause errors to propagate in cases where the combinatorial logic for each latch has different delays. C-elements are used to solve this, Figure 30 shows how the acknowledge signal of two controllers is connected to a C-element to combine them into one acknowledge signal. This is needed when one latch feeds two different combinatorial logic circuits. The latch should stay stable until both combinatorial circuits have completed their calculation.

The same is true for the request signal when two or more controllers are connected to one controller. Multiple request signals have to be combined into a single one that does not go high before all request signals are high and stay high until all are low. Figure 31 shows the request signal of two controllers combined to a single request using a C-element. As an example, this will be needed in a two-input adder where it is important that both inputs have fully propagated before the output is registered as ready. From these examples, it becomes clear that a C-element is needed as an AND would work for the rising edge of request or acknowledge but fail for the falling edge, while an OR gate would work for the falling and fail for the rising edge. -

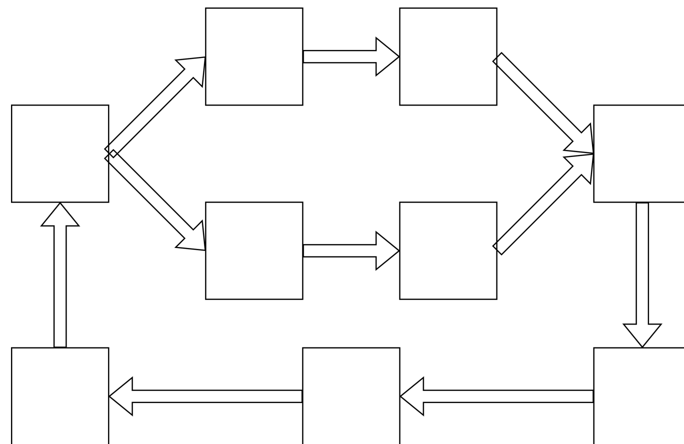


Figure 29: A block diagram of a circuit that forks into two paths before joining together again [4].

In cases where more than two paths are joined together or where one path is forked into more than two paths a two-input C-element will not be enough. In these cases, it is possible to use a C-element with more than two inputs, but this thesis will instead connect multiple two-input C-elements. Figure 32 shows how three (32a), four (32b), and eight (32c) inputs are combined into one using C-elements. In general, $N - 1$ C-elements are needed when connecting N inputs into one. This method of connecting C-elements is not exactly the same as having one N input C-element. This is shown in the wave diagram in Figure 33. When in0 and in1 go high the output

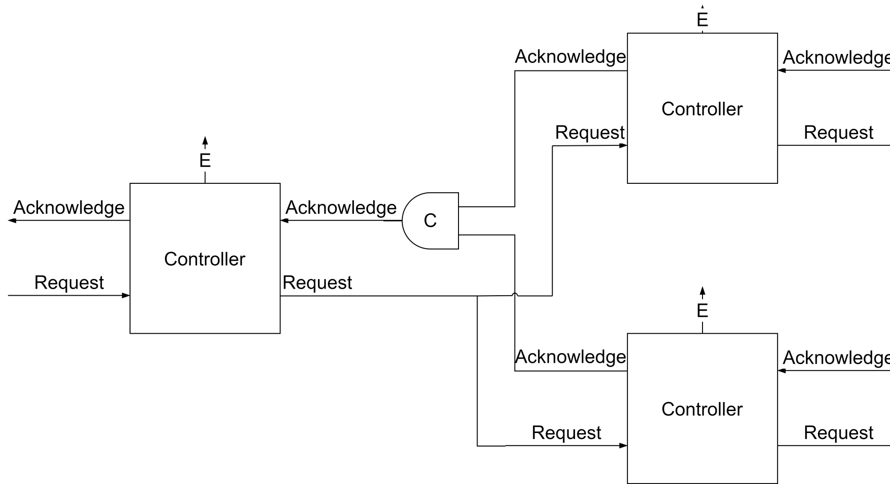


Figure 30: A C-element being used for the acknowledge signal when one controller is connected to two others [4].

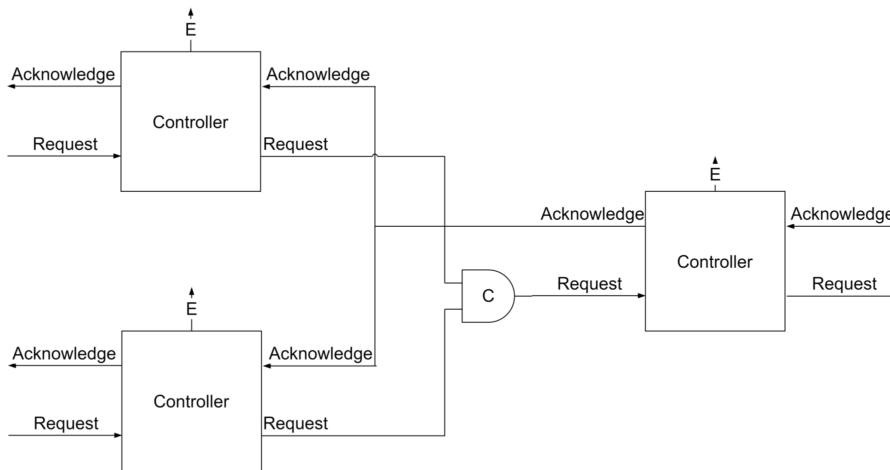
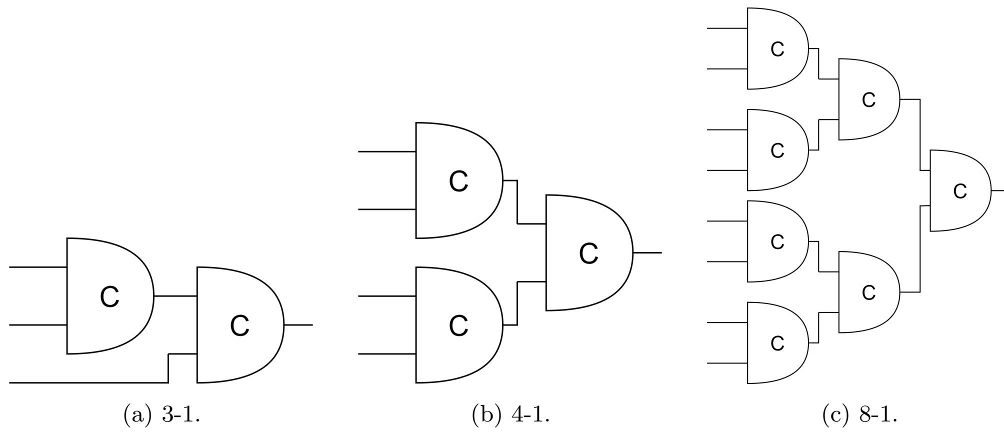


Figure 31: A C-element being used for the request signal when two controllers are connected to the same one [4].

of the first C-element will go high, this will stay high when in0 goes low again. The output of the last C-element will then go high when in2 goes high, even though only two of the inputs are high. A three-input C-element would not have gone high until in0 goes high, and all inputs are high at the same time. This property is not a problem for this application as the handshake protocol states that request will stay high until acknowledge goes high, and will stay low until acknowledge goes low. This would stop in0 from going low until out has gone high in the wave diagram.

3.2.5 Advantages

The above shows how to go from a synchronous to an asynchronous circuit using desynchronization. This chapter will give some reasons as to why this is advantageous compared to designing an asynchronous circuit from the bottom. Chapter 3.1.5 describes one major problem when designing asynchronous circuits, the lack of EDA tools. This makes the design process very tedious and manual. Desynchronization allows a designer to exploit the readily available EDA tools for synchronous design, and later convert to an asynchronous circuit to gain the advantages discussed in 3.1.1. Automating the desynchronization process also allows for low-effort comparison between synchronous and asynchronous circuits. This can allow a designer to easily see whether a synchronous or an asynchronous circuit is the best option.



(a) 3-1. (b) 4-1. (c) 8-1.

Figure 32: Different configurations of C-element to combine multiple inputs into one.

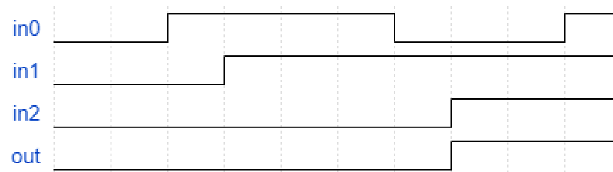


Figure 33: Wave diagram of the C-element configuration in Figure 32a, the inputs are numbered from 0 to 2 starting from the top.

4 Methodology

This chapter covers the process of converting from a synchronous to an asynchronous circuit through desynchronization. The focus will be on the development of an automated script for desynchronization, and the testing of this script by desynchronizing a multiplication and division module. This automated script is injected into Design Compiler and uses available commands to perform the conversion.

4.1 Tools

There are mainly two tools that will be used to perform the desynchronization, Design Compiler from Synopsys [3] and ModelSim from Siemens [21]. Both of these are best suited for synchronous design but contain features that can be used in the desynchronization process. Design Compiler can load in and modify a netlist from a synthesized synchronous design. This will be used to perform the desynchronization process using a Tcl script, as described in 4.4. Design Compiler is chosen as it is able to read a netlist and determine the length of the worst path between two cells. It can also add, remove, connect, and disconnect cells. ModelSim is used during simulation to test both the synchronous and asynchronous versions of the circuit.

4.2 MultDiv module

This thesis is intended as an investigation of techniques and a proof of concept of desynchronization, therefore, a relatively simple synchronous circuit is chosen to be desynchronized. This circuit is a modified version of the multiplication and division (MultDiv) module from the ibex 32-bit RISC-V CPU core [22]. The MultDiv module and the ALU of ibex are extracted and modified to work with 2^N bits for any positive N. The module performs multiplication through long multiplication and division by repeated subtraction. Both of these rely on the addition function of the ALU, allowing the ALU to constantly be in addition mode. The MultDiv module is configured to either multiplication or division and will calculate a result using the ALU and internal logic after getting two valid inputs. The calculation time depends on the bit width and the inputs, the time ranges from one to multiple clock cycles. A simple block diagram of how the MultDiv and ALU modules are connected to the outside is shown in Figure 34.

The MultDiv module is chosen as this is relatively simple while still having some complex features. Both multiplication and division have highly data-dependent calculation times meaning completion detection can play a huge role in the performance of a desynchronized circuit. The calculations are done in a loop over multiple cycles making the module more complex than a completely linear circuit. Lastly, the ability to change bit widths makes it possible to compare the area overhead of desynchronization on different widths. The ibex CPU was chosen as this fits the needs and is available through the Apache License, Version 2.0 [23].

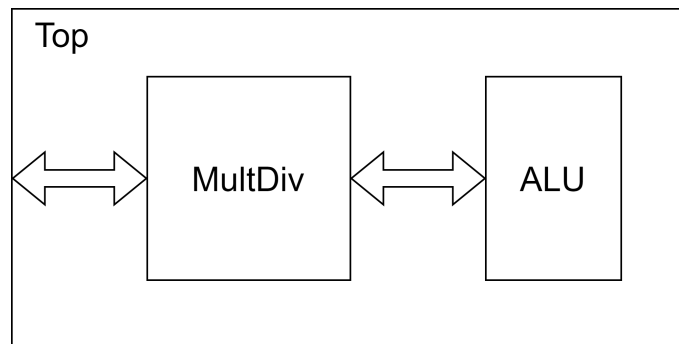


Figure 34: Block diagram of the MultDiv and ALU module from ibex [22].

4.3 Desynchronization

4.4 Tcl script

The core concept of the methodology is the automation of the desynchronization process. Ideally, a designer with no knowledge of asynchronous design should be able to follow a simple workflow and produce a desynchronized circuit. Design Compiler has built-in support for Tcl [24] scripts, consequently, Tcl is chosen as the scripting language to perform the desynchronization. The script performs the steps described in chapter 3.2, but changes up the order to simplify the process. In addition, the scripts generate needed sub-blocks, these are:

1. C-element
2. Even and odd controller
3. Delay lines of wanted lengths

Each of these blocks are easily changed allowing for different implementations. Later iterations will be able to test different controllers, completion detection mechanisms, and C-elements to see how each of these affects the performance. The script also creates a block with an even and an odd controller and a block with two latches. Anytime controller or latch is created in the script it will be one of these blocks that are created. This is not necessarily the best solution as it somewhat limits the placement of the latches and controllers. The script also groups multiple registers into register groups. This is done as a for instance 32-bit number stored in 32 registers does not need 32 controllers. Instead, it is enough with one even and one odd controller for the 64 latches that replace the registers. This saves both area and power but might cause some reduction in speed. More efficient grouping techniques are out of the scope of this thesis but are discussed in more detail by Davare et al. [25]. The pseudocode of the script is as follows:

```
generate Sub-Blocks

load netlist

set all_register [get registers]
set register_groups []
for reg in all_register do
    // Remove all registers used for clock gating
    if reg is clk_gate then
        remove reg
    end if

    // Collect all parallel register into a group
    if reg.group not in register_groups then
        add reg.group to register_groups
        create controller
        create completion_detection
        connect controller.req_out completion_detection
    end if
end for

for reg_group in register_groups do
    // Connect controllers to each other if there is a path between their registers
    for reg_group_inner in register_groups do
        if reg_group.rigth has path to reg_group_inner.left then
            connect reg_group.controller.right reg_group_inner.left
        else if reg_group_inner-right has path to reg_group.left then
```

```
        connect reg_group.controller.left reg_group_inner.right
      end if
    end for

    // Connect controllers to outside if there is a path between its registers and outside
    if reg_group.right has path to outside then
      create ports
      connect reg_group.right ports
    else if reg_group.left has path to outside then
      create ports
      connect reg_group.left ports
    end if

    // Create a latch for each register, connect to controller, and remove registers
    for reg in reg_group do
      create latch
      connect latch.in reg.in.net
      connect latch.out reg.out.net
      connect latch.enable_odd controller.enable_odd
      connect latch.enable_even controller.enable_even
      remove reg
    end for
  end for
```

The pseudocode is a simplified version of the actual code, but the general idea is the same. The actual code also has to handle fork and join situations, these are handled as described in chapter 3.2.4. See appendix A for the complete Tcl script. Figure 35 shows the desynchronization process of the Tcl script broken into three steps. Firstly the netlist is loaded in, then the clock is removed and a controller is inserted for every register group, controllers are connected to each other if they have a path between them. C-elements are inserted when multiple registers have a path to one register. Lastly, the register is replaced by two latches, and controllers are connected to the latches.

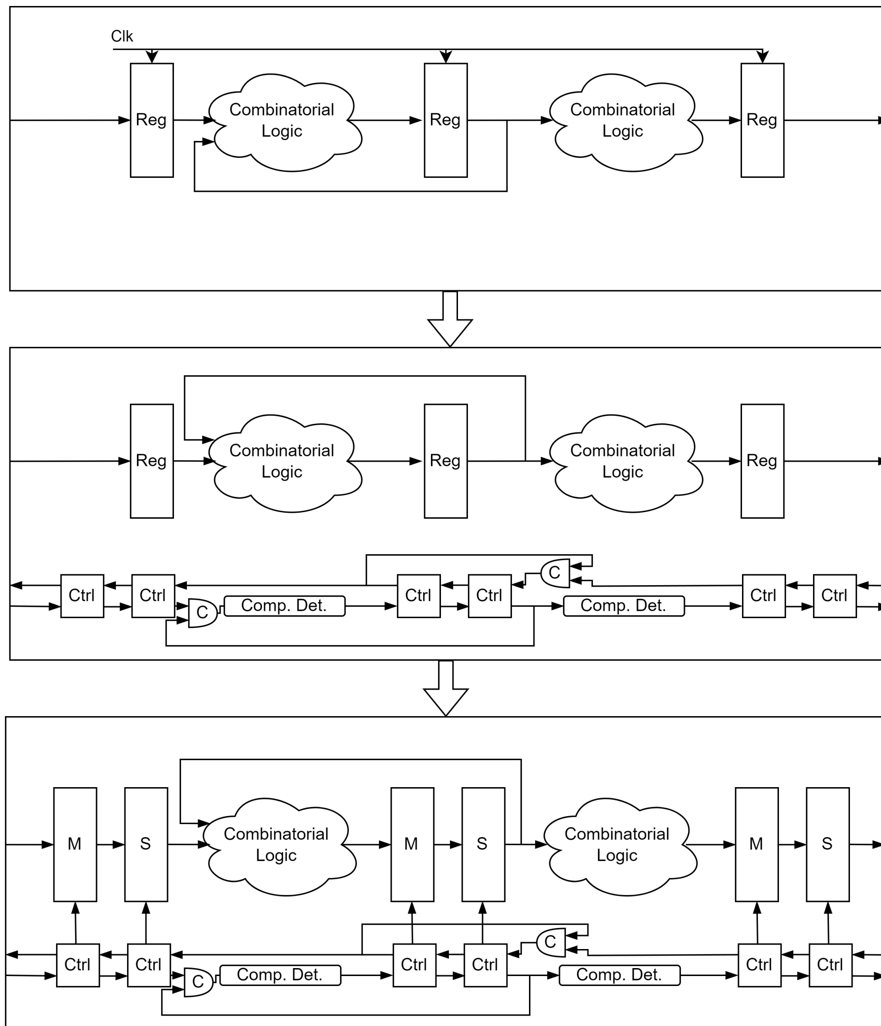


Figure 35: The desynchronization process of the Tcl script broken down into steps.

4.5 Delay calculation

It is crucial that the completion detection in a desynchronized circuit does not cause any invalid data to propagate. As mentioned in chapter 3.2.2, this thesis will use a simple delay line as a starting point. Figure 36 shows a simple setup with two latches and their controllers with some combinatorial logic in between. The length of this delay has to be long enough to guarantee that the result out of the combinatorial logic is valid after the request has propagated through the line. More specifically, the delay has to be long enough such that Latch2 stays transparent until $L2_i$ is stable and valid. Latch2 is transparent while L2 is high and turns opaque once L2 goes low. A simple approach is to set the delay to worst-case delay through the combinatorial logic. This will be valid but gives a delay longer than needed as the controllers also have some delay built in. A more precise calculation giving a smaller delay, and therefore a faster circuit, is presented below.

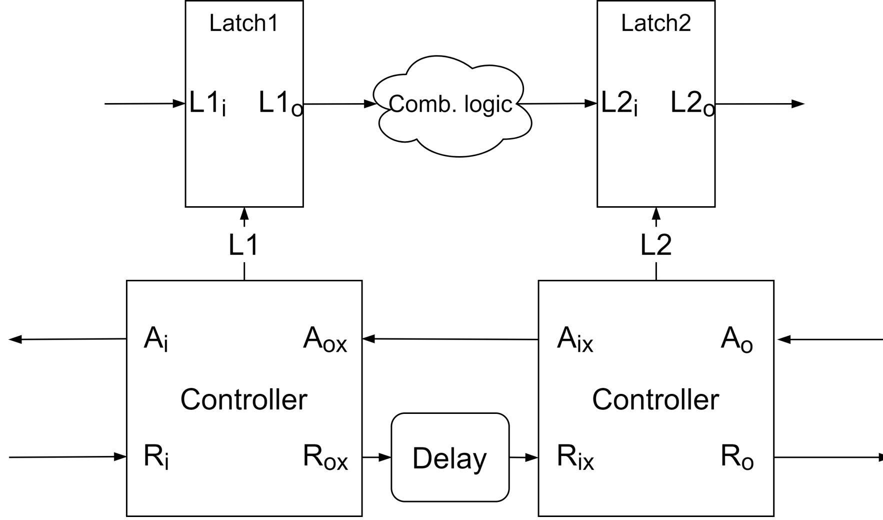


Figure 36: Block diagram of two controllers and latches, with names of the signals on the input and output [4].

Figure 37 shows a wave diagram of a transaction between the two controllers in Figure 36. Both controllers start out in an idle state where the latches are transparent and all request and acknowledge signals are zero. The transactions start when the first controller receives a request signal on R_i from another controller. This request signal indicates that the data on $L1_i$ is valid, and calculation can begin. Five different times are indicated at the top of the figure. $T_{R\text{-}R}$ is the time from request_in to request_out in a controller or the time from R_i to R_{ox}. T_D is the delay line length from R_{ox} to R_{ix}. T_{en} is the time from request_in to the lath enable signal, R_{ix} to L2. T_{DP} is the data propagation time in the latches, this is the time from the input changing to the output changing when the latch is open. Lastly, T_C is the delay through the combinatorial logic. The wave diagram is not to scale, and the length of the times are not realistic compared to each other.

To avoid errors it is crucial that Latch2 does not close before $L2_i$ is valid. $L2_i$ will be valid when the data has propagated through Latch1 and the combinatorial logic. Latch2 closes when the R_i has propagated through the first controller, the delay, and the second controller. This gives us equation 1. All of the variables except T_D are known and depend on the controller implementation, combinatorial logic, and the library used. T_D is then given by equation 2.

$$T_{R-R} + T_D + T_{EN} \geq T_{DP} + T_C \tag{1}$$

$$T_D \geq T_{DP} + T_C - T_{EN} - T_{R-R} \tag{2}$$

T_C will vary between controllers and need to be calculated for every combinatorial circuit, while the rest are constant as long as only one controller is used. The script calculates delay by measuring

the worst-case delay through the combinatorial circuit and subtracting the constant. This assumes that the controller is only connected to one controller on each side. The delay can be somewhat shorter in the cases where there are multiple controllers and the request signals are connected using C-elements as the C-elements also have some delay.

T_{DP} is the same for all latches as only one latch type is used. T_{EN} and $T_{R\text{-}R}$ can be found by observing the gate implementation in Figure 25. This gives T_{EN} equal to the delay through an inverter and a C-element while $T_{R\text{-}R}$ is equal to the delay through two inverters and two C-elements.

$T_{R\text{-}R}$ and T_{DP} will change if Latch1 is not open when it receives request_in. In a semi-decoupled controller, this happens when the first controller is still waiting for acknowledge_out. By observing the gate implementation in Figure 25 it can be seen that $T_{R\text{-}R}$ will be equal to the delay through three C-elements, one AND gate, and one inverter. T_{DP} is equal to the delay through two C-elements, one AND gate, and one inverter in addition to the propagation delay through the latch. Inserting these new values into equation 2 will increase the value on the right side since the change in T_{DP} is bigger than the change in $T_{R\text{-}R}$. This gives a higher value for the delay chain. Since this is a case that can happen, this bigger value for the delay line has to be used.

The calculation presented above is only valid when using semi-decoupled controllers and simple symmetric delay lines. Other controllers can have more or less concurrency and therefore different timing constraints.

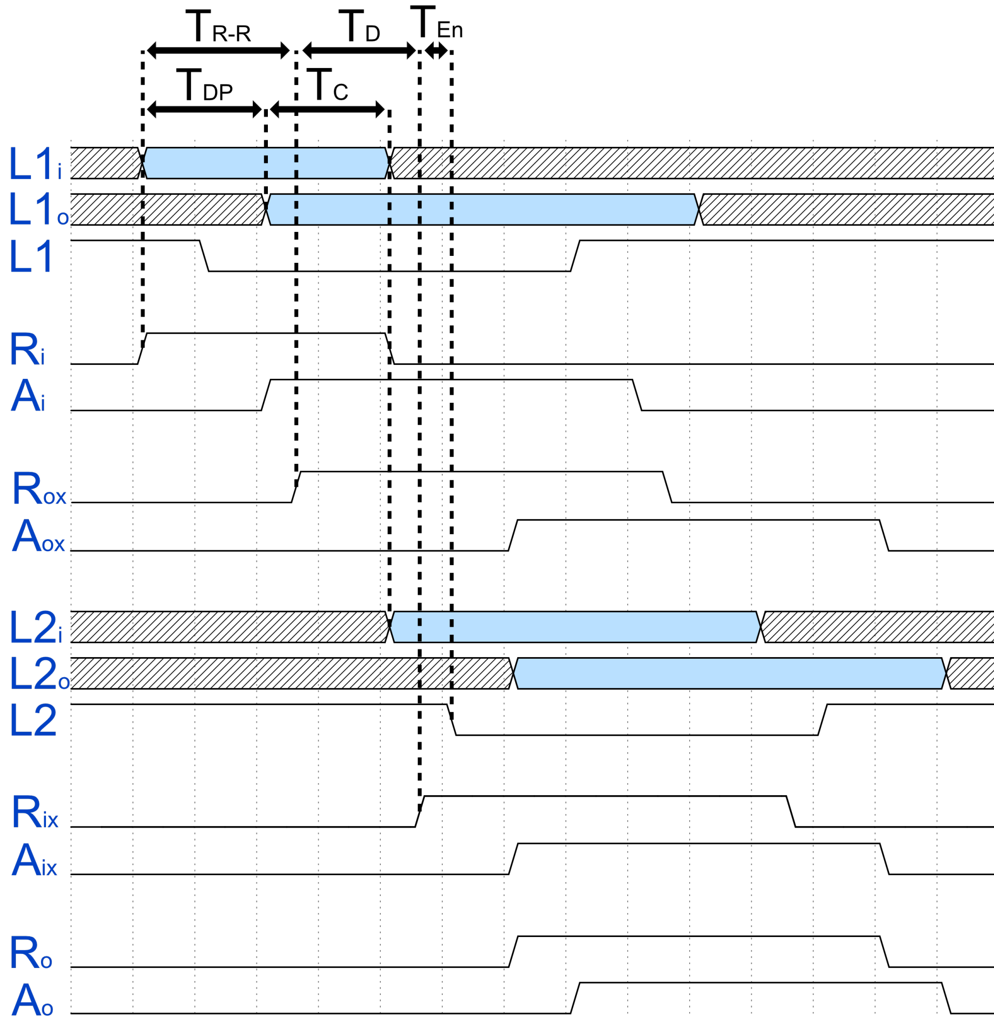


Figure 37: A transaction between the blocks in Figure 36.

4.6 Verification and Testing

This thesis is not performing a rigorous verification of the complete asynchronous circuits produced by the desynchronization as this thesis is intended as a proof of concept. Instead, the verification and testing will only be performed as black-box testing by giving the synchronous and asynchronous versions the same input and ensuring that the output is also equal. This should be sufficient to ensure that all latches, controllers, and delays are placed and connected correctly. The testing is done through gate-level simulation (GLS) in ModelSim. GLS simulates every gate with different delays depending on the gate. The gate delays are gotten from the gate library used, which is a library available to Nordic Semiconductor. Different libraries will have different gate delays so results will vary if another library is used.

The testing is done by using an asynchronous and a synchronous testbench that initializes the MultDiv module, gives it inputs, and reads its outputs. For the asynchronous testbench, there are simulated controllers connected to all controllers with paths to the outside. These simulated controllers have zero delays and will send new input data and consume output data instantly as soon they receive handshake signals. Output from the calculation is recorded and verified as soon as the MultDiv module indicates that the computation has finished. Similarly, the testbench for the synchronous circuit will wait until the MultDiv module indicates that the computation is complete. The result will then be verified and recorded before a new input is given.

The test cases vary a bit depending on the bit width used. For the 4-bit case, the tests consist of all possible input combinations of 4-bit numbers. This results in $16 * 16 = 256$ multiplication and division tests for a total of 512 tests. This gives an exhaustive comparison between the synchronous and asynchronous circuits. This approach is not feasible when using bigger bit widths as the number of tests will increase exponentially.

For bit widths bigger than 4, the test cases consist of 100 multiplications and 100 divisions with randomly generated input numbers. However, the random numbers do not change between synchronous and asynchronous tests to get comparable tests. The multiplication test uses the whole bit width for both input numbers, while the division test uses the whole bit width for the quotient but only half for the dividend. This is done to get more interesting results as a division with two similarly sized numbers often yields a result of 0, or other small numbers.

The semi-decoupled controller has been verified to be dead-lock free both alone and connected to other controllers by Cortadella et al. [2]. Cortadella et al. has also verified the general desynchronization process and the handshake protocol. It is therefore assumed that the black-block testing is sufficient when proof of concept and comparing performance is the main goal. It is possible that there exist input combinations that will break the asynchronous circuit, especially input combinations that break the handshake protocol, but these will not be considered as this is out of the scope of this thesis. All testing will be through simulation as the desynchronized circuit is not ready for physical testing.

5 Results

Two versions of the circuit are chosen for testing, one 32-bit and one 4-bit version. The 4-bit version is chosen as it allows testing of all input combinations while the 32-bit is chosen as this has more combinatorial logic. The extra combinatorial logic gives a longer critical path and consequently longer delay lines. These two should give a general idea of the differences between synchronous and asynchronous circuits, and what difference the amount of combinatorial logic makes. In both cases, the synchronous and asynchronous are tested as described in Chapter 4.6. All the results from these tests are presented below.

5.1 Area

The measurement of the area is done in Design Compiler which reports the area by adding the area of every individual cell. This measurement will only include the area from the cells, and not any area from nets or empty spaces between cells. Even though the measurement is not completely realistic it should be accurate enough since both the synchronous and asynchronous circuit is measured the same way. All area dimensions are given in µm^2, but the absolute area values are not that interesting since this will vary depending on what library is used. Looking at the ratio between synchronous and asynchronous areas instead gives an idea of the area overhead from desynchronization. Table 1 shows the area for asynchronous and synchronous areas for both 32-bit and 4-bit. The table shows that the percentage increase is lower for 32-bit, while the absolute increase is bigger.

Table 1: Area comparison between synchronous and asynchronous 4- and 32-bit circuits, given in µm^2.

	Sync	Async	Increase	Increase
4-Bit	139.5	316.3	176.7	127%
32-Bit	711.7	995.9	284.2	39.9%

The area can be split into smaller groups to see where the extra area in the asynchronous versions comes from, this is shown in Table 2 and 3. The groups chosen are combinatorial logic, registers/latches, controllers and delay lines. The combinatorial logic area is listed as higher for the synchronous version even though the main logic is not changed between versions. This extra area is due to the clock-gating and clock buffers. The tables also show that the synchronous and asynchronous versions have similar areas for registers/latches. This is expected as there is almost no difference in area between one register and two latches. Lastly, the tables show that the main difference in the area comes from controllers and delay lines. The controller area also includes the extra area from c-elements in the join structures. The controller area is constant between 4- and 32-bit as the number of controllers does not depend on the bit-width.

Table 2: Area for the 4-bit versions split into subgroups, given in µm^2.

	Comb. Logic	Reg/Latch	Controllers	Delay Lines	Total
Sync	85.8	53.7	0	0	139.5
Async	52.4	54.3	180.1	29.5	316.3

Table 3: Area for the 32-bit versions split into subgroups, given in μm^2.

	Comb. Logic	Reg/Latch	Controllers	Delay Lines	Total
Sync	439.7	272.0	0	0	711.7
Async	409.9	288.1	180.1	117.8	995.9

5.2 Speed

The speed is measured in time from the first test starts until the last test finishes, both division and multiplication will be included in the time. ModelSim is used for simulation, with a timescale set to 1 ps. The timescale specifies the precision of the simulation meaning anything faster than 1 ps will be rounded up to 1 ps. Table 4 shows the time for all 200 tests for 4- and 32-bit synchronous and asynchronous circuits. All time measurements in the table are given in ns but the ratio between synchronous and asynchronous times will be more interesting than the absolute value as the absolute value will depend on which library is used.

The clock speed of the synchronous circuits is made to accommodate the worst-case delay through the combinatorial logic at −40 °C, a slower clock would be needed to run the circuit at higher temperatures. All simulations are also run at −40 °C as this is the only corner available for the library chosen. It would be interesting to test all the circuits over a range of temperatures to see how this affects the results.

Table 4: Speed comparison between synchronous and asynchronous 4- and 32-bit circuits, given in ns.

	Sync	Async	Increase	Increase
4-Bit	2986.3	8757.5	5771.2	193%
32-Bit	25813	41739	15926	61.7%

5.3 Power

The power measurements are done in Design Compiler and are split into static and dynamic power. Table 5 and 6 show the dynamic and static power. The static power is given in nW and is the total leakage power from all cells. The dynamic power is given in mW and is calculated from the probability of a net or cell switching. These measurements show that dynamic power dominates the consumption and that the asynchronous versions have a higher power consumption compared to the synchronous versions. Similarly to the speed, these measurements are based on an operating temperature of −40 °C.

Table 5: Dynamic power comparison between synchronous and asynchronous 4- and 32-bit circuits, given in mW.

	Sync	Async	Increase	Increase
4-Bit	12.1	27.8	15.7	130%
32-Bit	65.0	82.1	17.1	26.0%

Table 6: Static power comparison between synchronous and asynchronous 4- and 32-bit circuits, given in nW.

	Sync	Async	Increase	Increase
4-Bit	2.1	6.5	4.4	210%
32-Bit	10.6	20.3	9.7	92%

6 Discussion

Overall, all the results presented in 5 show that the synchronous version of the circuit outperforms the asynchronous versions in both area, speed, and power. There are several reasons for these differences, some are down to the characteristics of synchronous compared to asynchronous design while others are down to flaws in the implementation. This chapter will discuss the results and which changes can be made to improve the relative performance of the asynchronous versions.

6.1 Area

The area of the asynchronous version, compared to the synchronous version, increased by 127% and 39.9%, as shown in 1. This is a bigger relative increase than what others have found. Cortadella et al. found an increase of around 22% for a desynchronized DES core and only a 1.45% increase in area for a desynchronized DLX core [2]. Even implementations using other controllers have found a smaller relative area increase. Hand et al. found a less than 10% increase when using a Blade controller in a Plasma CPU [18], even though a Blade controller on its own will have a bigger area than a semi-decoupled controller. The main reason that the MultDiv module has a much bigger relative area gain is the small amount of combinatorial logic. Both Cortadella et al. and Hand et al. desynchronized CPUs and CPU cores, while the MultDiv module is only a small part of an ibex core. Hence, the relative area increase is not directly comparable seeing that the MultDiv module only consists of an ALU and some control logic.

It is evident that the relative area increase for the MultDiv module is smaller for 32-bit than 4-bit, 39.9% and 127% respectively. This is expected as the number of controllers does not change between bit-widths. The number of registers changed to latches has little impact on the total area, as shown in Table 3, meaning the main difference between bit widths is the length of the delay lines. The 32-bit MultDiv module will have more control logic and 28 extra full adders compared to the 4-bit version. In general, the relative area increase when desynchronizing will depend on the amount of combinatorial logic, compared to the amount of controllers that will be needed. A circuit with a lot of combinatorial logic and few register groups is likely to see a smaller relative area increase than a circuit with little combinatorial logic and a lot of register groups.

Later tests of the script should focus on bigger circuits, for example the whole ibex CPU, to get a better idea of how much the area increases. A bigger circuit is expected to have a smaller relative area increase because of the extra combinatorial logic, as explained above. In addition, a bigger circuit will require a bigger clock network. Removing this clock network allows for further reduction of the relative increase.

Another element that may affect the area is the C-element implementation. This thesis uses a gate implementation as shown in Figure 12, but this is not necessarily the area-efficient implementation. Ideally, C-element should not be implemented with other gates, but on a transistor level. Unfortunately, there was no available library that includes a C-element implementation. It is hard to say exactly how much smaller a transistor implementation could be, but this is something that should be investigated.

6.2 Speed

Looking at the time measurements in Table 4, it is clear that the asynchronous circuit is quite a bit slower than the synchronous version. The relative change in speed is more significant for the 4-bit version, with its speed decreasing to around a third of the synchronous speed while the 32-bit version experiences a relative slowdown of 61.7%. Comparatively, Cortadella et al. found a 3.75% slowdown for a simulated DES core and a 37% slowdown for a physical ASPIDA circuit, containing a DLX core and memory [2]. However, the synchronous version of the ASPIDA core used two non-overlapping global clocks. The synchronous ASPIDA would be 3.5% slower than the asynchronous version if only one global clock was used.

Cortadella et al. did not find a significant improvement but showed that a desynchronized circuit can perform at a similar speed as the original synchronous version. This result was not found for the MultDiv module. There are some explanations for this discrepancy in results, one being too pessimistic calculations of the delay line. During testing, it was found that the delay line could be reduced by a lot without breaking the circuit. For the 4-bit version, the delay line could in fact be completely removed. The total for all tests with no delay line is 6290.9 ns which is 28.2% faster than with the delay line, but still 110.7% slower than the synchronous version. In the case of the 32-bit version, the delay line could be reduced by two-thirds yielding a run-time of 24 578 ns which is 41.1% faster than with a full delay line and 4.8% faster than the synchronous version.

There are multiple reasons why the delay line can be shortened so much. The delay line is calculated based on the worst-case critical path. For the longest delay line, the critical path is the carry bit traveling through all full adders. This is not very likely as it requires that exactly one of the inputs of every full adder is one, except the first full adder which needs both inputs to be one to send out a carry bit. If both inputs to one of the other full adders are one it will send its carry-bit instantly and does not need to wait on carry-in. This does never happen for multiplication since the long multiplication algorithm left-shifts one of the inputs. This can be seen by looking at 4'b1111 * 4'b1111. The two first steps are given in 3 and 4.

$$\begin{array}{r} {\scriptstyle 111} \\ 1111 \\ +\,1110 \\ \hline 1101 \end{array} \tag{3}$$

$$\begin{array}{r} {\scriptstyle 1} \\ 1101 \\ +\,1100 \\ \hline 1001 \end{array} \tag{4}$$

The algorithm will continue to left-shift the second input meaning there will never be a produced carry-bit from the first full adder. This does not explain the degree of potential reduction of the delay line as it is still possible for the carry-bit to be propagated from the second full adder all the way to the last. For instance when multiplying a*a when a is 4'b1011 or 8'b10101011 etc. Here, the carry bit travels from the second full adder to the fourth, and only the second full adder has two ones as input. The first step of 4'b1011 * 4'b1011 is given in 5.

$$\begin{array}{r} {\scriptstyle 11} \\ 1011 \\ +\,0110 \\ \hline 0001 \end{array} \tag{5}$$

Making assumptions about how far the carry bit will travel is a possible way to speed up the circuit. As shown, there are scenarios for multiplication where the travel of the carry bit will be the critical path. In the full ibex, the ALU is also used for normal addition where any number can be input meaning it is possible for the carry bit to go through all full adders. This means there is a need for a more complex completion detection that can predict when the calculation is complete, this will be discussed in more detail 6.5. The 4-bit version with no delay has been tested with 4'b1011 * 4'b1011 as input, while the 32-bit version with a third of the calculated delay was tested with both inputs equal to 32'b10101010101010101010101010101011, both of these gave the correct results. This means the carry bit can travel through almost all full adders without breaking the versions with reduced delay, meaning there are other factors that enable a shortened delay.

Another potential reason is that the controllers introduce a bigger delay than calculated in 4.5. This calculation assumes that both controllers are idle and ready to receive new data. However, this may not be the case if the left controller is fed new data as soon as it is ready. In this case, only one of the controllers will be idle at a time, and the calculation is no longer correct. In this situation, the performance is limited by the semi-decoupled controller and the fact that the delay line is symmetric. Since the semi-decoupled controller is implemented with a four-phase handshake it needs request_in to rise before closing the latch and request_in to fall before opening the latch

again, as seen in the STG in Figure 26. The asymmetric delay means that both the falling and rising edge of request_in is delayed. This can be seen by looking at the input handshake of a controller from the moment the neighbor controller raises request_out. This situation is shown in Figure 38, the controller setup is the same as in 36. It can be seen from the figure that both the rising and falling edge of the request signal is delayed, meaning the full time for the handshake is more than double the delay.

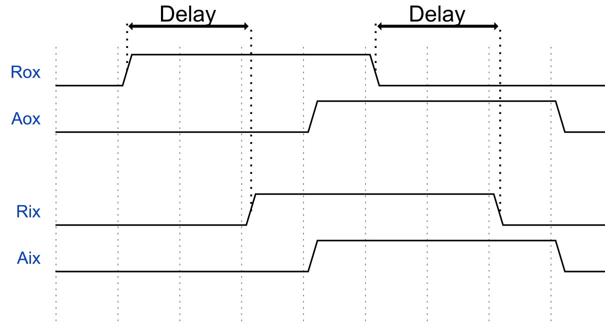


Figure 38: Full handshake between two controllers.

One potential solution to this problem is always halving the delay lines, but this does not work in general as a controller can send request_out before the falling edge of request_in. The half delay will only work when all controllers need to wait on the falling edge, this happens when the circuit is fed data every time it is ready. In cases where the circuit has been idle for some time, allowing all falling edges to propagate, before receiving inputs the whole delay is needed. To test this, the MultDiv module was left idle for a while before receiving the same multiplication input as before. This made the circuit fail when using a third of the calculated delay. This is a flaw of the semi-decoupled controller and the symmetric delay line which can be solved by using other controllers or an asymmetric delay line, this is discussed in 6.5 and 6.6.

Similarly, 245 of the 512 tests failed when testing the 4-bit asynchronous version without delay lines and waiting for 1 µs between the finish of one test and the start of the next. The longest calculated delay is 916 ps so all request signals have time to propagate in 1 µs. However, all of the tests are successful when using half the calculated delay so there are some elements affecting the needed delay that are not accounted for. This is something that should be investigated at a later time to ensure that the correct delay is chosen.

6.3 Power

The power measurements, Table 5 and 6, shows that the asynchronous versions are more power-hungry, both when looking at dynamic and static power. This is contrary to what others such as Cortadella et al. has found, where there was observed a power reduction after desynchronization [2]. However, the power measurements of the MultDiv module are not unexpected. The main power advantage of asynchronous circuits is the lack of a global clock [1]. A global clock can activate parts of the circuits to process data and make them consume power. Clock gating can avoid this, but the clock still has to run and consume power in itself. Asynchronous circuit exchange the global clock for local handshakes which will only be active when they are transferring or processing data. This creates a system where only parts that are needed are active and consume power. This advantage is not seen in this thesis as the MultDiv module is always active and consumes power during testing.

Instead of a power reduction, a power increase is seen. This increase is caused by the extra number of gates that are introduced and consume power. Both controllers and delay lines will toggle during calculation. In addition, the power measurement from Design Compiler does not include the power needed to drive and propagate the clock. Overall the power measured in this thesis should not be read too much into. A bigger, more complex circuit with active and inactive parts is needed to get a better idea of the power difference before and after desynchronization.

6.4 Operating Conditions

A key feature of asynchronous circuits is the ability to handle varying operating conditions. Asynchronous circuits can often handle a much larger range of temperatures and supply voltages. Unfortunately, none of the tests run in this thesis have varying temperatures or supply voltages. It is therefore not possible to draw a conclusion on how the performance of the synchronous and asynchronous MultDiv modules behave for changing operating conditions. However, there are multiple papers available that experiment with changing conditions. The desynchronized ESPIDA chip, produced by Cortadella et al., behaved correctly down to a supply voltage of 0.95 V while it was originally designed for 2.5 V [2]. This is only 0.35 V above the threshold voltage.

Furthermore, Yun et al. showed that an asynchronous circuit could operate at a supply voltage of 1.85 V while its synchronous counterpart was designed for 3.3 V [9]. They also showed that the asynchronous circuits outperform a synchronous circuit when the temperature is lowered as the synchronous circuit was designed to operate at 100 °C.

Considering these findings, it is reasonable to assume that the asynchronous MultDiv module will be robust to changes in operating conditions. Currently, the synchronous and asynchronous versions are only simulated at −40 °C at 0.8 V. Testing over a bigger range would give an idea of how robust the two versions are. The synchronous clock is also determined from the critical path at −40 °C, it is therefore likely that this would fail relatively fast if the temperature rises. The asynchronous circuit should handle a temperature better as the delay lines will slow down at the same rate as the combinatorial logic.

6.5 Completion Detection

The completion detection is an important factor in the performance of a desynchronized circuit. This thesis uses a delay line which is very easy to implement as it does not need any inputs from the combinatorial logic. All that is needed to implement a delay line is knowledge about the length of the critical path through the combinatorial logic. However, there are several other completion detection mechanisms that could improve the performance of the circuit, some of these are described in Chapter 3.2.2. There is not a single completion detection that is best for any combinatorial logic, instead, the best alternative depends on the function of the combinatorial logic. For the MultDiv module specifically, a speculative delay line is one option that could improve the performance.

By looking at the multiplication algorithm one can see that addition is not always performed. The algorithm will look at the last bit of a right-shifted version of the b-input, an addition is performed if the bit is 1 but not if the bit is 0. The delay needed to ensure the correct result for the latches saving output from the adders is a lot shorter when no addition is performed as the critical path no longer goes through the full adders. As this is determined by one bit the speculative delay line gets a very simple condition to choose a smaller delay. It is enough to check that this bit is 0 and that the MultDiv module is in multiplication mode. Figure 39 shows a possible implementation of a speculative delay line for the MultDiv module, Div_sel is 0 when the module is in multiplication mode and B_shift holds the right shifted value of B.

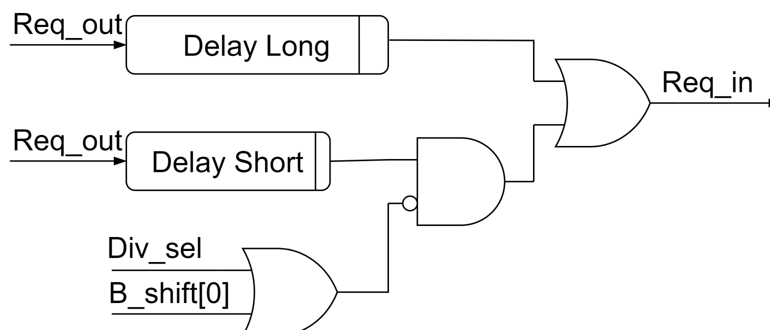


Figure 39: A possible implementation of a speculative delay line for the MultDiv module.

This speculative delay line can be used for the completion detection for all the latches that get inputs from the output of the full adders. In the asynchronous 32-bit version of the MultDiv module, it is found that the short delay can be zero-length as the delay from the controllers are enough to accommodate the critical path when no addition is done. In total, there are six delay lines needed for the MultDiv module, five of these are related to combinatorial logic with a critical path through the full adders. All of these delay lines are longer than 3200 ps. This means a speculative delay line could potentially shorten the delay from request_out to request_in to zero when no addition is done. Assuming a randomly generated 32-bit B input with on average 16 bits set to zero, this reduction will theoretically reduce the time of one multiplication by around $3200\,\text{ps} * 16 = 52.2\,\text{ns}$. The actual time saving from a speculative delay line may be different as there are no tests for this theory in this thesis

It is also possible to improve the completion detection for division or when an addition is done in multiplication. One possible option is to use a speculative delay line and determine which delay is needed by predicting how long the carry-bit has to travel. This has been explored in an independent adder circuit by Nowick [10]. They found that an asynchronous circuit implemented with this completion detection was 5% to 30% faster than a comparable synchronous circuit.

A big disadvantage of the speculative delay line is that the falling edge of the request always travels through the longest delay. A possible option to mitigate this disadvantage is to use asymmetric delay lines or to implement another completion detection such as an efficient asymmetric speculative delay line.

6.6 Controller

While the semi-decoupled controller is a good starting point with the relatively low area and decent performance, as discussed in [4], it has some flaws that other controllers improve on. One drawback of the semi-decoupled controller is that its performance can be limited by the falling edge of request_in, as discussed in 6.2. This can affect the performance of the whole circuit as a controller waiting for request_in to go low will not pull request_out low. This will in turn make the next controllers wait longer for their request_in to go low. Waiting for request_in to go low only affects designs where data is fed in as fast as possible. In designs where there is some time between data being sent in, the falling edge of request_in will have time to propagate. A semi-decoupled controller is, therefore, a good option in systems where data is not expected as often as the number of gates, and therefore area and power are relatively small compared to other controllers.

In designs where data is expected to be received more rapidly, other controllers might provide better performance. Both the fully-decoupled controller introduced by Furber and Day [5] and the desynchronization model controller introduced by Cortadella et al. [2] improve on this by allowing request_out to go low before request_in goes low. The STGs of these controllers are shown in Figure 40 and 41. While these controllers solve some problems they will still be affected by the falling edge of request_in. The only option to have no disadvantages from the falling edge of request is to change handshake protocols and use a two-phase handshake. Here the falling edge of request will transmit information just like the rising edge.

Other controllers should also be considered such as Blade [18], MOUSETRAP [19], or other options. The choice of controller will always be a trade-off between area, speed, and power and the best choice will depend on the combinatorial logic and the design requirements. This thesis only tests semi-decoupled controllers, but it would have been interesting to test the other options, and test using multiple different controllers together.

6.7 Limitations in Tcl script

The current version of the Tcl scripts functions as intended but there are some limitations. The script is able to convert a synchronous netlist into an asynchronous one with no user input but does so in a sub-optimal way. Due to some inefficient nested for-loops, the run time of the script increases exponentially with netlist size. Desynchronizing the 4-bit version is almost instant, whereas the

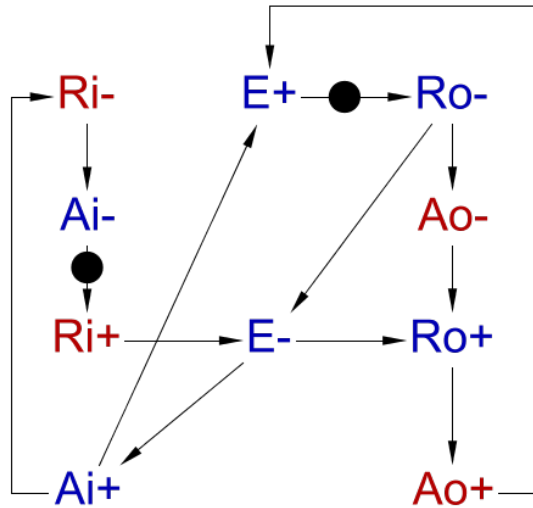


Figure 40: STG of the fully-decoupled controller introduced by Furber and Day [5].

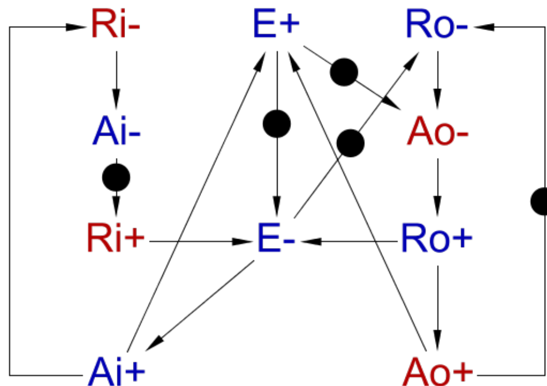


Figure 41: STG of the desynchronization model controller introduced by Cortadella et al. [2].

32-bit version takes around 30 seconds. This is a lot of time for a relatively small netlist. Bigger designs will likely take an unacceptably long time.

The script also makes some assumptions that might not generally be true but is true for the synthesized netlists used in this thesis. Design Compiler has no functionality to determine whether or not a cell is a register. Instead, the script assumes that all registers have "_reg" in their name and that no other cells have this in their name. This might not always be true and the script may break in such cases. In addition, Tcl is a very simple language and is not suited for complex programming. Tcl was chosen as it is directly compatible with Design Compiler but ideally, a more complex object-oriented language, such as C++, should be used as class functionality would simplify the script significantly.

6.8 Verification

Verification is a fundamental aspect of the design process. Without verification, there is no way to know whether the circuit functions as intended, or whether the circuit will break under certain input combinations. The current verification of the MultDiv module consists of a simple testbench that provides inputs and verifies that the output is correct. This testbench does not prove correctness and is only intended to verify that all latches, delay lines, and controllers are connected correctly. Exhaustive verification is outside this thesis's scope but must be done if the asynchronous circuits are to be used in actual applications.

6.9 Testbench

The current testing consists of two test benches, one for the synchronous and one for the asynchronous circuit. Initially, a testbench for the synchronous circuit was created to verify that the synchronous circuit functions as expected. Later, a testbench for the asynchronous circuit is created based on the synchronous circuit. However, the two testbenches are quite different and a lot of manual changes are needed. Ideally, there should be no need for a designer to create more than one synchronous testbench. The script should automatically make a testbench suited for the asynchronous circuit. This can be done by creating a wrapper for the asynchronous circuit such that it fits into the synchronous testbench, or creating a completely new asynchronous testbench.

A wrapper allows reusing the synchronous testbench which guarantees that the tests are the same. One possible approach is to use a FIFO on every input and output. The FIFO will need both a synchronous and asynchronous interface, either synchronous input and asynchronous output or the other way around. Figure 42 shows one possible implementation of a wrapper between a synchronous testbench and an asynchronous circuit. While not crucial, the FIFOs act as buffers to avoid situations where the performance is limited by input or output speed. As soon as a handshake is complete new data from the FIFOs is available at the inputs. On the other side the output FIFO allows the asynchronous circuit to output its data even if the testbench is not ready to receive. This is helpful as a synchronous testbench normally only provides or consumes data at a positive clock edge, while the asynchronous circuit can receive or transmit data at any time.

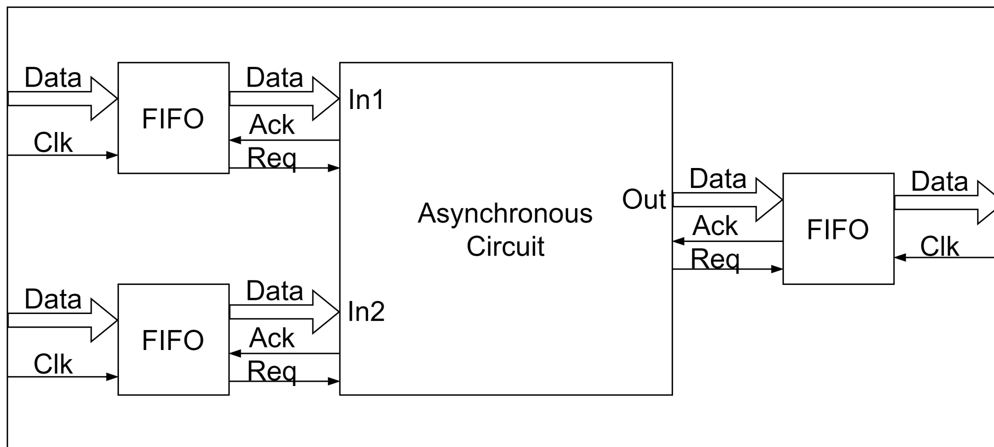


Figure 42: Possible implementation of a wrapper between an asynchronous circuit and a synchronous testbench.

It is also possible to completely rewrite the testbench instead of using a wrapper. The testbench will then need some new mechanisms to handle the handshakes from the asynchronous circuit.

In addition to creating a wrapper or a new testbench, quite a few other files need to be changed for the simulation to run. These are files that configure the simulation framework used by Nordic Semiconductor. Some files need to be changed to point to the new netlist, while others to point to the wrapper or testbench and any other dependencies. Currently, all of these files are changed manually. While most of the changes are small, the modification of these files is a task that should be handled by the script.

6.10 Desynchronization

The procedure to desynchronize a synchronous circuit provided by the Tcl script severely simplifies the process of designing an asynchronous circuit. Converting a synchronous version of the MultDiv module to an asynchronous version becomes very simple. All that is needed is to create a netlist of the synchronous version, load this netlist into Design Compiler and run the Tcl script. The script requires no knowledge about asynchronous design and handles the conversion with no user input. This ability for an unaccustomed designer to create an asynchronous circuit is one of the selling

points of desynchronization. Desynchronization through the script could be a realistic option when designing a circuit, given that the issues discussed above are addressed.

However, desynchronization will not be suitable for all synchronous designs. There are many considerations that have to be done before deciding whether a synchronous or an asynchronous circuit is the best option for any given design. Designs where the delay through the combinatorial circuit is data-dependent may have more significant potential for speed-ups than designs where the delay is not data-dependent. Further, designs where operating conditions, mainly temperature, and supply voltage, are expected to change a desynchronized circuit may be suited as it handles changing conditions well. There are many such considerations that have to be done before choosing between a synchronous and an asynchronous circuit. Luckily, the fully automated desynchronization approach enables fast creation and testing of the asynchronous circuit to provide a foundation for comparison. A later, improved, version of the script could potentially allow a designer to create a functioning synchronous design and automatically compare the performance to a corresponding asynchronous circuit.

7 Future Work

As this thesis only provides a proof of concept, there is a lot of future work that has to be done before the desynchronization methodology can be used in real applications. Much of the work needed is already mentioned in the discussion in chapter 6, but this chapter will serve as an overview of what should be done in the future.

Firstly, the Tcl script should be improved to be faster with larger designs. The script should also be tested more thoroughly to ensure that it functions correctly with other designs than the MultDiv module. Currently, it is a bit complicated to change the controller or the completion detection in the script, this is also something that should be improved to make it easier to compare different choices. The script should also be able to generate a wrapper or a testbench automatically such that minimal manual effort is required.

Secondly, desynchronization should be done on more complex designs as the MultDiv module is relatively simple and small. In addition, controllers and completion detection should be changed to get a better comparison of the performance, area, and power between different options. While the semi-decoupled controller and a simple delay line are functional, they are not necessarily the options that provide the best performance.

Lastly, the verification and testing need to be much more exhaustive to guarantee that the asynchronous circuit actually functions as intended. This is a necessary step if the asynchronous circuit is going to be used in any real-life applications.

8 Conclusion

This master thesis has introduced the idea of desynchronization, a method of converting from synchronous to asynchronous circuits, with the potential of enhanced robustness, speed, and lower power consumption. In addition, a Tcl script has been created that can automate this conversion in Synopsys Design Compiler. The Tcl script has successfully converted a simple synchronous multiplication and division module to a functioning asynchronous version. Although the resulting asynchronous circuit is slower, bigger, and more power-hungry than the synchronous version, it is important to note that this thesis only serves as a proof of concept. Notably, the goal of the thesis, which was to create a script that can perform the conversion on EDA tools available to Nordic Semiconductor, has been achieved. Techniques that could better extract the potential advantages of asynchronous circuits have been introduced. Furthermore, the future work needed to improve the script and the asynchronous circuit has been discussed. While there is still quite a bit of work needed before this methodology can create an asynchronous circuit suited for real-life application, this thesis establishes a starting point for further development.

References

[1] P. A. Beerel, R. O. Ozdag and M. Ferretti, *A Designer's Guide to Asynchronous VLSI*. Cambridge: Cambridge University Press, 2010, ISBN: 978-0-521-87244-7. DOI: 10.1017/CBO9780511674730. [Online]. Available: https://www.cambridge.org/core/books/designers-guide-to-asynchronous-vlsi/6EBCDD0AED8481F1953F253425821D43 (visited on 15th Nov. 2022).

[2] J. Cortadella, A. Kondratyev, L. Lavagno and C. Sotiriou, 'Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications', en, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1904–1921, Oct. 2006, ISSN: 0278-0070. DOI: 10.1109/TCAD.2005.860958. [Online]. Available: http://ieeexplore.ieee.org/document/1677680/ (visited on 20th Sep. 2022).

[3] *Design Compiler*, en. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html (visited on 25th May 2023).

[4] B.-I. F. Kleppe, 'Investigation of techniques for circuit desynchronization', en, Project Thesis, NTNU, Trondheim, Dec. 2022.

[5] S. Furber and P. Day, 'Four-phase micropipeline latch control circuits', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 2, pp. 247–253, Jun. 1996, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/92.502196.

[6] *Principles of Asynchronous Circuit Design*, en. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4757-3385-3 (visited on 7th Nov. 2022).

[7] K. Gausdal, 'Completion Detection in Asynchronous Circuits', en, Project Thesis, NTNU, Trondheim.

[8] J. Lee, B.-G. Nam and H.-J. Yoo, 'Dynamic Voltage and Frequency Scaling (DVFS) scheme for multi-domains power management', in *2007 IEEE Asian Solid-State Circuits Conference*, Nov. 2007, pp. 360–363. DOI: 10.1109/ASSCC.2007.4425705.

[9] K. Yun, P. Beerel, V. Vakilotojar, A. Dooply and J. Arceo, 'The design and verification of a high-performance low-control-overhead asynchronous differential equation solver', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 4, pp. 643–655, Dec. 1998, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/92.736138.

[10] S. Nowick, 'Design of a low-latency asynchronous adder using speculative completion', en, *IEE Proceedings - Computers and Digital Techniques*, vol. 143, no. 5, p. 301, 1996, ISSN: 13502387. DOI: 10.1049/ip-cdt:19960704. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/ip-cdt_19960704 (visited on 23rd May 2023).

[11] G. Jung, V. Perepelitsa and G. Sobelman, 'Time borrowing in high-speed functional units using skew-tolerant domino circuits', in *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 5, May 2000, 641–644 vol.5. DOI: 10.1109/ISCAS.2000.857542.

[12] D. Chinnery and K. Keutzer, 'Reducing the Timing Overhead', en, in *Closing the Gap Between ASIC & Custom: Tools and Techniques for High-Performance ASIC Design*, D. Chinnery and K. Keutzer, Eds., Boston, MA: Springer US, 2002, pp. 57–100, ISBN: 978-0-306-47823-9. DOI: 10.1007/0-306-47823-4_3. [Online]. Available: https://doi.org/10.1007/0-306-47823-4_3 (visited on 25th Nov. 2022).

[13] S. Tugsinavisut, Y. Hong, D. Kim, K. Kim and P. Beerel, 'Efficient asynchronous bundled-data pipelines for DCT matrix-vector multiplication', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 4, pp. 448–461, Apr. 2005, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2004.842908.

[14] M. Dean, D. Dill and M. Horowitz, 'Self-timed logic using current-sensing completion detection (CSCD)', in *[1991 Proceedings] IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Oct. 1991, pp. 187–191. DOI: 10.1109/ICCD.1991.139878.

[15] K. Kim, P. Beerel and Y. Hong, 'An asynchronous matrix-vector multiplier for discrete cosine transform', in *ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No.00TH8514)*, Jul. 2000, pp. 256–261. DOI: 10.1109/LPE.2000.155295.

[16] F. Commoner, A. W. Holt, S. Even and A. Pnueli, 'Marked directed graphs', en, *Journal of Computer and System Sciences*, vol. 5, no. 5, pp. 511–523, Oct. 1971, ISSN: 0022-0000. DOI: 10.1016/S0022-0000(71)80013-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000071800132 (visited on 3rd Nov. 2022).

[17] *Workcraft*. [Online]. Available: https://workcraft.org/start (visited on 19th Dec. 2022).

[18] D. Hand, M. T. Moreira, H.-H. Huang *et al.*, 'Blade – A Timing Violation Resilient Asynchronous Template', en, in *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, Mountain View, CA: IEEE, May 2015, pp. 21–28, ISBN: 978-1-4799-8716-0. DOI: 10.1109/ASYNC.2015.13. [Online]. Available: https://ieeexplore.ieee.org/document/7152687/ (visited on 6th Sep. 2022).

[19] M. Singh and S. Nowick, 'MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines', in *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*, ISSN: 1063-6404, Sep. 2001, pp. 9–17. DOI: 10.1109/ICCD.2001.954997.

[20] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin and C. Sotiriou, 'Handshake protocols for de-synchronization', in *10th International Symposium on Asynchronous Circuits and Systems, 2004. Proceedings.*, ISSN: 1522-8681, Apr. 2004, pp. 149–158. DOI: 10.1109/ASYNC.2004.1299296.

[21] *ModelSim HDL simulator*, en. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/ (visited on 31st May 2023).

[22] *Ibex RISC-V Core*, original-date: 2017-08-08T12:16:36Z, May 2023. [Online]. Available: https://github.com/lowRISC/ibex (visited on 25th May 2023).

[23] *Apache License, Version 2.0*. [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0 (visited on 25th May 2023).

[24] *Tcl*. [Online]. Available: https://www.tcl.tk/about/language.html (visited on 26th May 2023).

[25] A. Davare, K. Lwin, A. Kondratyev and A. Sangiovanni-Vincentelli, 'The best of both worlds: The efficient asynchronous implementation of synchronous specifications', in *Proceedings. 41st Design Automation Conference, 2004.*, ISSN: 0738-100X, Jul. 2004, pp. 588–591. DOI: 10.1145/996566.996727.

Appendix

A desync.tcl

```
# source -echo -verbose ./rm_setup/dc_setup.tcl

# Remove Desynchronization design to be able to change between 4 bits easier
redirect -variable all_designs {[list_designs ]}
if {[string first "Desynchronization" $all_designs] != -1} {
  # Only remove if found to be sure new design is emtpy
  remove_design Desynchronization
}

# Automatically load in the sync design, hardcoded need to be changed or remove
# read_file -format ddc
↪  {/pri/bjkl/work/master/Workspace_Desynchronization/projects/student/Desynchronization/syn/resu
# read_file -format ddc
↪  {/pri/bjkl/work/master/Workspace_Desynchronization/projects/student/Desynchronization/syn/resu
read_file -format ddc
↪  {/pri/bjkl/work/master/Workspace_Desynchronization/projects/student/Desynchronization/syn/resu

# Write sdf for unchanged module incase it is missing
# current_design Desynchronization
# write_sdf ./results/Desynchronization.mapped.sdf

echo "--------------------------------------------------"
echo "--------------------------------------------------"
echo "--------------------------------------------------"
echo "--------------------------------------------------"
echo "--------------------------------------------------"
echo "Starting Desynchronization"
echo "--------------------------------------------------"
echo "--------------------------------------------------"
echo "--------------------------------------------------"
echo "--------------------------------------------------"
echo "--------------------------------------------------"

# Save all original designs in collection, needed to loop later
set all_original_designs [get_designs ]

# Create design containg two latches
#
#
#
#
#
# In                              Out
#  Latch1  Latch2
#
#
#
#
#
#
#             EnE             EnO
```

```
echo "-------------------------------------------------"
echo "Creating latch block"
echo "-------------------------------------------------"
redirect -variable all_designs {[list_designs ]}
if {[string first "latch_block" $all_designs] != -1} {
  # Only remove if found to be sure new design is emtpy
  remove_design latch_block
}

# Create new design
create_design latch_block
current_design latch_block

# Add ports to the design
create_port latch_in -direction "in"
create_port latch_out -direction "out"
create_port latch_en_e -direction "in"
create_port latch_en_o -direction "in"
create_port rst_n -direction "in"

# Create nets
create_net net_in
create_net net_mid
create_net net_out
create_net net_en_e
create_net net_en_o
create_net net_rst_n

# Add two latches to the design, lib name:
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C
create_cell latch_1
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_LATRQX1_CSC20L
create_cell latch_2
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_LATRQX1_CSC20L


# Connect the two latches to each other and two the ports
connect_net net_in latch_in
connect_net net_in latch_1/D
connect_net net_mid latch_1/Q
connect_net net_mid latch_2/D
connect_net net_out latch_2/Q
connect_net net_out latch_out

connect_net net_en_e latch_en_e
connect_net net_en_e latch_1/CLK
connect_net net_en_o latch_en_o
connect_net net_en_o latch_2/CLK

connect_net net_rst_n rst_n
connect_net net_rst_n latch_1/RESET
connect_net net_rst_n latch_2/RESET

link

# Create the controller, make one block for two controller, one for each latch in
↪  a latch block
#
```

```
#                en_E           en_O
#
#        |
#        |
# ack_in                            ack_out
#
#
#           ctrl E |      ctrl O |   |
# req_in                            req_out
#
#
#
#
#

# C-element created using three 2-input and one 3-input nandgates
echo "------------------------------------------------"
echo "Creating C-element from and gates"
echo "------------------------------------------------"
redirect -variable all_designs {[list_designs ]}
if {[string first "c_element" $all_designs] != -1} {
  # Only remove if found to be sure new design is emtpy
  remove_design c_element
}

# Create new design
create_design c_element
current_design c_element

# Add ports to the design
create_port A -direction "in"
create_port B -direction "in"
create_port Y -direction "out"
create_port rst_n -direction "in"

# Create nets, net names described in picture
create_net net_in_A
create_net net_in_B
create_net net_nand1
create_net net_nand2
create_net net_nand3
create_net net_out_Y
create_net net_nand4
create_net net_rst_n

# Add three 2-input and one 3-input NAND gates
create_cell nand1
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_ND2X1_CSC20L
create_cell nand2
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_ND2X1_CSC20L
create_cell nand3
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_ND2X1_CSC20L
create_cell nand4
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_ND3X1_CSC20L

# And gate for reset
create_cell and_rst
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_AN2X1_CSC20L
```

```
# Connect the nets to ports and gates
connect_net net_in_A A
connect_net net_in_A nand1/A
connect_net net_in_A nand2/A

connect_net net_in_B B
connect_net net_in_B nand1/B
connect_net net_in_B nand3/A

connect_net net_out_Y Y
connect_net net_out_Y nand2/B
connect_net net_out_Y nand3/B
connect_net net_out_Y and_rst/Z
connect_net net_nand4 and_rst/A
connect_net net_nand4 nand4/Z

connect_net net_rst_n and_rst/B
connect_net net_rst_n rst_n

connect_net net_nand1 nand1/Z
connect_net net_nand2 nand2/Z
connect_net net_nand3 nand3/Z

connect_net net_nand1 nand4/A
connect_net net_nand2 nand4/B
connect_net net_nand3 nand4/C

link

# C-element created using three 2-input and one 3-input nandgates
echo "-------------------------------------------------"
echo "Creating C-element with positive reset function from and gates"
echo "-------------------------------------------------"
redirect -variable all_designs {[list_designs ]}
if {[string first "c_element_rst_high" $all_designs] != -1} {
  # Only remove if found to be sure new design is emtpy
  remove_design c_element_rst_high
}

# Create new design
create_design c_element_rst_high
current_design c_element_rst_high

# Add ports to the design
create_port A -direction "in"
create_port B -direction "in"
create_port Y -direction "out"
create_port rst_n -direction "in"

# Create nets, net names described in picture
create_net net_in_A
create_net net_in_B
create_net net_nand1
create_net net_nand2
create_net net_nand3
create_net net_out_Y
create_net net_nand4
```

```
create_net net_rst_n
create_net net_rst_n_x

# Add three 2-input and one 3-input NAND gates
create_cell nand1
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_ND2X1_CSC20L
create_cell nand2
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_ND2X1_CSC20L
create_cell nand3
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_ND2X1_CSC20L
create_cell nand4
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_ND3X1_CSC20L

# And gate for reset
create_cell or_rst
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_OR2X1_CSC20L
create_cell inv_rst
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_INVX1_CSC20L

# Connect the nets to ports and gates
connect_net net_in_A A
connect_net net_in_A nand1/A
connect_net net_in_A nand2/A

connect_net net_in_B B
connect_net net_in_B nand1/B
connect_net net_in_B nand3/A

connect_net net_out_Y Y
connect_net net_out_Y nand2/B
connect_net net_out_Y nand3/B
connect_net net_out_Y or_rst/Z
connect_net net_nand4 or_rst/A
connect_net net_nand4 nand4/Z

connect_net net_rst_n rst_n
connect_net net_rst_n inv_rst/A
connect_net net_rst_n_x inv_rst/Z
connect_net net_rst_n_x or_rst/B

connect_net net_nand1 nand1/Z
connect_net net_nand2 nand2/Z
connect_net net_nand3 nand3/Z

connect_net net_nand1 nand4/A
connect_net net_nand2 nand4/B
connect_net net_nand3 nand4/C

link

# Single controller
# Only difference between odd and even block is the reset state
echo "-----------------------------------------------"
echo "Creating even controller block"
echo "-----------------------------------------------"
redirect -variable all_designs {[list_designs ]}
if {[string first "controller_block_even" $all_designs] != -1} {
  # Only remove if found to be sure new design is emtpy
```

```
  remove_design controller_block_even
}

# Create new design
create_design controller_block_even
current_design controller_block_even

# Create ports to the controller block
create_port req_in -direction "in"
create_port ack_in -direction "out"
create_port req_out -direction "out"
create_port ack_out -direction "in"
create_port en_out -direction "out"
create_port rst_n -direction "in"

# Create internal nets
create_net net_req_in
create_net net_ack_in
create_net net_req_out
create_net net_ack_out
create_net net_en_out
create_net net_inv1
create_net net_inv3
create_net net_inv4
create_net net_and1
# create_net net_c2A
# create_net net_c2B
create_net net_rst_n

# Create the gates for the controller, two c-elements, one and-gate and four
↪ inverters (two are connected to the inputs of one c-element)
create_cell c_el1 c_element_rst_high
create_cell c_el2 c_element
create_cell and1
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_AN2X1_CSC20L
create_cell inv1
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_INVX1_CSC20L
create_cell inv2
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_INVX1_CSC20L
create_cell inv3
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_INVX1_CSC20L
create_cell inv4
↪ GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_INVX1_CSC20L

connect_net net_rst_n rst_n

# Connect nets to ports and gates
connect_net net_req_in req_in
connect_net net_req_in inv1/A
connect_net net_inv1 inv1/Z
connect_net net_inv1 c_el1/A

connect_net net_req_out c_el2/Y
connect_net net_req_out and1/A
connect_net net_req_out req_out
connect_net net_ack_out and1/B
connect_net net_ack_out ack_out
connect_net net_and1 c_el1/B
```

```
connect_net net_and1 and1/Z

connect_net net_en_out en_out
connect_net net_en_out inv4/A
connect_net net_en_out c_el1/Y
connect_net net_en_out inv2/A
connect_net net_ack_out inv3/A
connect_net net_ack_in inv2/Z
connect_net net_ack_in ack_in

connect_net net_inv3 c_el2/A
connect_net net_inv3 inv3/Z

connect_net net_inv4 c_el2/B
connect_net net_inv4 inv4/Z

connect_net net_rst_n c_el1/rst_n
connect_net net_rst_n c_el2/rst_n

link

# Single controller
echo "--------------------------------------------------"
echo "Creating odd controller block"
echo "--------------------------------------------------"
redirect -variable all_designs {[list_designs ]}
if {[string first "controller_block_odd" $all_designs] != -1} {
  # Only remove if found to be sure new design is emtpy
  remove_design controller_block_odd
}

# Create new design
create_design controller_block_odd
current_design controller_block_odd

# Create ports to the controller block
create_port req_in -direction "in"
create_port ack_in -direction "out"
create_port req_out -direction "out"
create_port ack_out -direction "in"
create_port en_out -direction "out"
create_port rst_n -direction "in"

# Create internal nets
create_net net_req_in
create_net net_ack_in
create_net net_req_out
create_net net_ack_out
create_net net_en_out
create_net net_inv1
create_net net_inv3
create_net net_inv4
create_net net_and1
# create_net net_c2A
# create_net net_c2B
create_net net_rst_n
```

```
# Create the gates for the controller, two c-elements, one and-gate and four
↪  inverters (two are connected to the inputs of one c-element)
create_cell c_el1 c_element
create_cell c_el2 c_element
create_cell and1
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_AN2X1_CSC20L
create_cell inv1
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_INVX1_CSC20L
create_cell inv2
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_INVX1_CSC20L
create_cell inv3
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_INVX1_CSC20L
create_cell inv4
↪  GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_INVX1_CSC20L

connect_net net_rst_n rst_n

# Connect nets to ports and gates
connect_net net_req_in req_in
connect_net net_req_in inv1/A
connect_net net_inv1 inv1/Z
connect_net net_inv1 c_el1/A

connect_net net_req_out c_el2/Y
connect_net net_req_out and1/A
connect_net net_req_out req_out
connect_net net_ack_out and1/B
connect_net net_ack_out ack_out
connect_net net_and1 c_el1/B
connect_net net_and1 and1/Z

connect_net net_en_out en_out
connect_net net_en_out inv4/A
connect_net net_en_out c_el1/Y
connect_net net_en_out inv2/A
connect_net net_ack_out inv3/A
connect_net net_ack_in inv2/Z
connect_net net_ack_in ack_in

connect_net net_inv3 c_el2/A
connect_net net_inv3 inv3/Z

connect_net net_inv4 c_el2/B
connect_net net_inv4 inv4/Z

connect_net net_rst_n c_el1/rst_n
connect_net net_rst_n c_el2/rst_n

link

# Create one block with both controllers
echo "------------------------------------------------"
echo "Creating block witk odd and even controller"
echo "------------------------------------------------"
redirect -variable all_designs {[list_designs ]}
if {[string first "full_controller" $all_designs] != -1} {
  # Only remove if found to be sure new design is emtpy
  remove_design full_controller
```

```
}

# Create new design
create_design full_controller
current_design full_controller

# Create ports
create_port req_in -direction "in"
create_port ack_in -direction "out"
create_port req_out -direction "out"
create_port ack_out -direction "in"
create_port en_E -direction "out"
create_port en_O -direction "out"
create_port rst_n -direction "in"

# Create nets
create_net net_req_in
create_net net_ack_in
create_net net_req_out
create_net net_ack_out
create_net net_en_E
create_net net_en_O
create_net net_req_x
create_net net_ack_x
create_net net_rst_n

# Create two controller cells
create_cell controller_even controller_block_even
create_cell controller_odd controller_block_odd

# Connect nets to ports and cells
connect_net net_req_in req_in
connect_net net_req_in controller_even/req_in
connect_net net_ack_in ack_in
connect_net net_ack_in controller_even/ack_in

connect_net net_req_x controller_even/req_out
connect_net net_req_x controller_odd/req_in
connect_net net_ack_x controller_even/ack_out
connect_net net_ack_x controller_odd/ack_in

connect_net net_req_out controller_odd/req_out
connect_net net_ack_out controller_odd/ack_out
connect_net net_req_out req_out
connect_net net_ack_out ack_out

connect_net net_en_E controller_even/en_out
connect_net net_en_E en_E
connect_net net_en_O controller_odd/en_out
connect_net net_en_O en_O

connect_net net_rst_n rst_n
connect_net net_rst_n controller_even/rst_n
connect_net net_rst_n controller_odd/rst_n

link
```

```
set debug_file [open "debug.txt" w+]
chan truncate $debug_file 0
close $debug_file
proc create_delay_chain {delay name net_in} {
  set debug_file [open "debug.txt" a]
  # Delay of C-element: 1-0: 83, 0-1: 76
  # Delay of one inverter cell in ps
  # set inv_delay 4.57113
  set inv_delay 10.215
  # Ceil rounds up to closest int
  # Delay from controller : 75 = 3165 / 41,95 = 30 (41,95 = (477+380)/10.215)
  set delay_needed [expr $delay - 75]
  # set delay_needed 0
  if {$delay_needed < 0} {
    set delay_needed 0
  }
  set num_inv_first [expr ceil($delay_needed/$inv_delay)]
  set num_inv_first [expr ceil($num_inv_first/2)]
  # Divide by 2, round and multiply by two to guarantee that number is divisible
↪   by two
  set num_inv [expr int(ceil($num_inv_first/2)*2)]
  # set num_inv 0

  echo "-------------------------------------------------"
  echo [format "Creating delay block %s with delay of %f consisting of %d
↪   inverters" $name $delay $num_inv]
  echo "-------------------------------------------------"
  puts $debug_file [format "Creating delay block %s with delay of %f and needed
↪   delay of %f consisting of %d inverters" $name $delay $delay_needed $num_inv]

  for {set i 1} {$i < $num_inv+1} {incr i} {
    create_cell inv$name$i
↪   GF22FDX_SC7P5T_116CPP_BASE_CSC20L_SSG_0P72V_0P00V_0P00V_0P00V_M40C/SC7P5T_INVX1_CSC20L
    if {$i == 1} {
      create_net $net_in
      connect_net $net_in inv$name$i/A
    }
    create_net net$name$i
    connect_net net$name$i inv$name$i/Z
    if {$i != 1} {
      connect_net net$name[expr {$i-1}] inv$name$i/A
    }
  }

  link
  close $debug_file

  if {$num_inv == 0} {
    return $net_in
  }
  return net${name}[expr {$num_inv}]
}
set all_controllers {}
set first_latch_MultDiv {}
set max_delay 0
set num_c_el 0

# Ensure that file exissts and is empty
```

```
set delay_file [open "dataDelays.txt" w+]
chan truncate $delay_file 0
close $delay_file

foreach_in_collection design $all_original_designs {
  # Set design to current design
  current_design $design
  echo "--------------------------------------------------"
  echo "--------------------------------------------------"
  echo [format "Starting desyncrhonization for design %s" [get_attribute $design
↪  full_name]]
  echo "--------------------------------------------------"
  echo "--------------------------------------------------"
  # Find all register cells
  set all_registers_clk_gate [filter_collection -regexp [get_cells] {full_name =~
↪  ".*_reg.*"}]

  # Remove all clk_gate register since these are not needed
  if {[sizeof_collection $all_registers_clk_gate] > 0} {
    foreach_in_collection reg $all_registers_clk_gate {
      if {[string first "clk_gate" [get_attribute $reg full_name]] != -1} {
        remove_cell $reg
      }
    }
  }

  # Find all registers
  set all_registers [filter_collection -regexp [get_cells] {full_name =~
↪  ".*_reg.*"}]
  # Only need to do changes in designs with registers
  if {[sizeof_collection $all_registers] > 0} {

    # Create a port and net for rst signal
    create_port rst_n -direction "in"
    create_net net_rst_n
    connect_net net_rst_n rst_n


    # echo [format "Found %d registers in the design" [sizeof_collection
↪  £all_registers]]

    # Open and delete all contents of datalDelays.txt
    set delay_file [open "dataDelays.txt" a]

    # Regex expression to find data arrival time
    set data_arrival_exp {data arrival time +([0-9]*\.[0-9]*)\n}

    # Iterating through every register and finding timing path between them
    # echo [format "%-50s %-50s %7s" "From" "To" "Slack"]
    foreach_in_collection register_from $all_registers {
      # Finding output pin of first reg
      set register_from_out [filter_collection -regexp [get_pins -of_objects
↪  $register_from] {full_name =~ ".*/Q.*"}]

      foreach_in_collection register_to $all_registers {
        # Finding input pin of second register
        set register_to_in [filter_collection -regexp [get_pins -of_objects
↪  $register_to] {full_name =~ ".*/D.*"}]
```

```
    if {[sizeof_collection $register_from_out] > 0 && [sizeof_collection
↪ $register_to_in] > 0} {
      # Saving timing report to variable
      redirect -variable timing_report {[report_timing -from
↪ $register_from_out -to $register_to_in]}

      # Only care about reports where a path between start and end was found
      if {[string first "No paths" $timing_report] == -1} {
        # Exctract total delay of the path
        regexp $data_arrival_exp $timing_report match path_delay
        # echo [format "%-50s %-50s %s" [get_attribute £register_from
↪ full_name] [get_attribute £register_to full_name] £path_delay]
        puts $delay_file [format "%s %s %s" [get_attribute $register_from
↪ full_name] [get_attribute $register_to full_name] $path_delay]
        if {$path_delay > $max_delay} {
          set max_delay $path_delay
        }

        # Break after finding the first match to avoid getting mulitple of
↪ the same path, must be removed later to know all paths
        # break
      }
    }
  }
}

close $delay_file


# Set the constant names for input, output and enable on latch and register
set reg_input "/D"
set reg_output "/Q"
set reg_CLK "/CLK"

set latch_input "/latch_in"
set latch_output "/latch_out"
set latch_enable_even "/latch_en_e"
set latch_enable_odd "/latch_en_o"

set controller_enable_even "/en_E"
set controller_enable_odd "/en_O"
set controller_req_in "/req_in"
set controller_ack_in "/ack_in"
set controller_req_out "/req_out"
set controller_ack_out "/ack_out"

set delay_in "/delay_in"
set delay_out "/delay_out"


# Iterating trough all registers, grouping paralel registers and placing one
↪ controller and one delay chain for each group
# Assuming name on form uin_MultDiv_imd_val_q_i_reg_0__0_ will be grouped
↪ based on first number: *_reg_0_*
# This may not be correct generally for all designs

echo "-------------------------------------------------"
```

```
echo "Adding one controller for each group of paralel registers"
echo "-------------------------------------------------"

# Set new temp collection of all register to be able to remove registers that
↪ are grouped
set temp_all_register $all_registers
set i 0
set register_groups {}
while {1} {
  echo [format "i: %d length of collection %d" $i [sizeof_collection
↪ $temp_all_register]]
  set register [index_collection $temp_all_register $i]
  # Setting name filter to uin_MultDiv_imd_val_q_i_reg_0_ (or equivalent)
  regexp {(.*)_[0-9]} [get_attribute $register full_name] -> name_filter
  # name_filter will be -1 if no matching pattern was found
  echo [format "Found this name_filter %s for %s" $name_filter [get_attribute
↪ $register full_name]]
  if {$name_filter != -1} {
    # append_to_collection matching_registers £register
    foreach_in_collection register_inner $temp_all_register {
      # Check if register matches the name_filter
      if {[string first $name_filter [get_attribute $register_inner
↪ full_name]] != -1} {
        # Add matching register to matching collection and remove from all
↪ registers to avoid duplicate collections
        # append_to_collection £matching_registers £register_inner
        if {[get_attribute $register_inner full_name] != [get_attribute
↪ $register full_name]} {
          # Don't remove the first register to get correct indexing from i
          set temp_all_register [remove_from_collection $temp_all_register
↪ $register_inner]
        }
      }
    }
    # Add one controller and one delay chain for all these controllers
    # Create a full_controller block contating a controller for each of the
↪ latches in the latch block
    set latch_prefix [regsub "_reg" $name_filter "_latch"]
    set controller_name "${latch_prefix}_controller"
    lappend register_groups $latch_prefix

    create_cell $controller_name full_controller
    set net_even "${latch_prefix}_net_even"
    set net_odd "${latch_prefix}_net_odd"

    connect_net net_rst_n $controller_name/rst_n

    # Keep track of all controllers added to design
    lappend all_controllers $controller_name

    create_net $net_even
    create_net $net_odd
    connect_net $net_even $controller_name$controller_enable_even
    connect_net $net_odd $controller_name$controller_enable_odd

    # Create nets for req and ack signals and create delay block
    set net_req_in "net_${controller_name}_req_in"
    set net_ack_in "net_${controller_name}_ack_in"
```

```
    set net_req_out "net_${controller_name}_req_out"
    set net_ack_out "net_${controller_name}_ack_out"


    # Read trough dataDelays and find the biggest delay for this
    set delay_file [open "dataDelays.txt" r]
    set file_data [read $delay_file]
    close $delay_file
    set data [split $file_data "\n"]
    set delay_time 0
    foreach line $data {
      set col [split $line " "]
      set reg_name [lindex $col 1]
      if {[string first $name_filter $reg_name] != -1} {
        # Found register connected to this controller, find the register with
↪ longest path
        set this_delay [lindex $col 2]
        if {$this_delay > $delay_time} {
          set delay_time $this_delay
        }
      }
    }

    # Create and connect delay_block to req_out
    # create_net £net_req_out
    set net_delay_in "net_${controller_name}_delay_in"
    create_net $net_delay_in
    set delay "${controller_name}_delay"
    set net_out [create_delay_chain $delay_time $delay $net_delay_in]
    current_design $design
    connect_net $net_out $controller_name$controller_req_in

    set req_out $controller_name$controller_req_out
    set net_req_out "net_${controller_name}_req_out"
    create_net $net_req_out
    # connect_net £net_out £delay£delay_in
  }
  incr i
  if {$i >= [sizeof_collection $temp_all_register]} {
    echo "Finnish grouping registers, registers left:"
    query_objects $temp_all_register
    break
  } elseif {$i >= 100} {
    # Sanity check,  might be to low for some designs
    break
  }
}
# Create delay block for register, only use one block for every register that
↪ works in paralel?
# Add one delay block for each register pair with path between?
echo "--------------------------------------------------"
echo "Creating delay block and connecting adjecent controllers"
echo "--------------------------------------------------"
foreach register_group $register_groups {
  set controller_name "${register_group}_controller"

  # List used to keep track of controllers that need to be connected
```

```
    # Should only need req_in and ack_out as these are the inputs but have all
↪   incase they are needed later
    set controller_connections_req_in {}
    set controller_connections_ack_in {}
    set controller_connections_req_out {}
    set controller_connections_ack_out {}
    # Loop through the other potensial groups and see if there is a path
↪   between the first register in each group
    # This assumes that if the first register has a path then all the others
↪   also have one
    # Do this by checking for any path from input of this register to output to
↪   any other register
    echo "------------------------------------------------"
    echo "Finding all other controllers with path to this controller"
    echo "------------------------------------------------"
    foreach register_group_inner $register_groups {
      set register_from_out $register_group_inner"_0_"$reg_output
      set register_to_in $register_group"_0_"$reg_input
      redirect -variable timing_report_left {[report_timing -from
↪   $register_from_out -to $register_to_in]}

      set register_from_out $register_group"_0_"$reg_output
      set register_to_in $register_group_inner"_0_"$reg_input
      redirect -variable timing_report_right {[report_timing -from
↪   $register_from_out -to $register_to_in]}
      # Only care about reports where a path between start and end was found
      if {[string first "No paths" $timing_report_left] == -1} {
        # Connect delay_out and ack_out to the adjacent controllers reqq_in and
↪   ack_in
        set controller_name_inner "${register_group_inner}_controller"
        set req_out $controller_name_inner$controller_req_out
        set net_req_out "net_${controller_name_inner}_req_out"
        connect_net $net_req_out $req_out
        # set delay_out_pin "£{controller_name_inner}_delay£{delay_out}"
        set net_delay_in "net_${controller_name_inner}_delay_in"
        # create_net £net_delay_out
        # connect_net £net_delay_out £delay_out_pin
        # lappend controller_connections_req_in £net_delay_in
        lappend controller_connections_req_in $net_req_out

        set ack_out $controller_name_inner$controller_ack_out
        lappend controller_connections_ack_in $ack_out
      }
      if {[string first "No paths" $timing_report_right] == -1} {
        # Connect delay_out and ack_out to the adjacent controllers reqq_in and
↪   ack_in
        set controller_name_inner "${register_group_inner}_controller"
        set req_in $controller_name_inner$controller_req_in
        lappend controller_connections_req_out $req_in

        set ack_in $controller_name_inner$controller_ack_in
        set net_ack_in "net_${controller_name_inner}_ack_in"
        create_net $net_ack_in
        connect_net $net_ack_in $ack_in
        lappend controller_connections_ack_out $net_ack_in
      }
    }
```

```
    # Find controller for registers that have paths to any input or outpurt
↪  ports and add port to be able to send and recieve handshakes
    # Do this by iterating trouhg each controller and see if the first register
↪  has a path from input or oupout to any port
    # Might not cover all cases where there is a path to input other than data
↪  input on register

    # Create a list of all input and output ports to the outside
    set outside_ports [get_ports]

    echo "-------------------------------------------------"
    echo "Creating ports to the outside for controller with path to input or
↪  outputs to the module"
    echo "-------------------------------------------------"
    # Variables to make sure ony one prot is created for output and input
↪  handshake
    set in_port_created 0
    set out_port_created 0
    foreach_in_collection port $outside_ports {
      if {([string first "req" [get_attribute $port full_name]] == -1) &&
↪  ([string first "ack" [get_attribute $port full_name]] == -1)} {
        if {([get_attribute $port direction]) == "in" && (!($in_port_created))}
↪  {
          set register_in $register_group"_0_"$reg_input
          redirect -variable timing_report_in {[report_timing -from $port -to
↪  $register_in]}
          # Found path between outside and input of register
          if {[string first "No paths" $timing_report_in] == -1} {
            # Create ack and req ports and connect to controller

            set port_ack_in "ack_in_${register_group}"
            create_port $port_ack_in -direction "out"
            set net_ack_in "net_${controller_name}_ack_in"
            connect_net $net_ack_in $port_ack_in
            # create_net £net_ack_in
            # connect_net £net_ack_in £port_ack_in

            set port_req_in "req_in_${register_group}"
            set net_port_req_in "net_req_in_${register_group}"
            create_port $port_req_in -direction "in"
            create_net $net_port_req_in
            connect_net $net_port_req_in $port_req_in

            lappend controller_connections_req_in $net_port_req_in
            lappend controller_connections_ack_in $port_ack_in

            set in_port_created 1
          }
        } elseif {([get_attribute $port direction] == "out") &&
↪  (!($out_port_created))} {
          set register_out $register_group"_0_"$reg_output
          redirect -variable timing_report_out {[report_timing -from
↪  $register_out -to $port]}
          # Found path between output of register and outside
          if {[string first "No paths" $timing_report_out] == -1} {
            set port_ack_out "ack_out_${register_group}"
            create_port $port_ack_out -direction "in"
            set net_port_ack_out "net_ack_out_${register_group}"
```

```
            create_net $net_port_ack_out
            connect_net $net_port_ack_out $port_ack_out
            lappend controller_connections_ack_out $net_port_ack_out

            set port_req_out "req_out_${register_group}"
            create_port $port_req_out -direction "out"
            set net_req_out "net_${controller_name}_req_out"
            # set net_delay_out "£{controller_name}_delay_out"
            connect_net $net_req_out $port_req_out
            lappend controller_connections_req_out $net_req_out

            set out_port_created 1
          }
        }
      }
    }

    # Handle the req and ack pins, currently a special case for all amounts of
↪   connections, should be done automatically
    # Connectinc right to left
    # Five different cases as these are the only observed amounts in the
↪   design
    # Might not be most efficient way but "should" work
    # controller_connections_req_in and controller_connections_ack_out should
↪   always be same length so only check length of one of them
    # The elements in the two lists are the req and ack ports that should be
↪   connected, for req it is either the req prot or output of the delay
    # This type of connecting the C-elements does not guarantee that all
↪   signals have the same value at the same time
    # It does guarantee that all signals have been 1 or 0 at some point since
↪   last change, should be good enough for now
    # The elements of req_in and ack out are nets while req_out and ack_in are
↪   ports/pins, since req_in and ack_out are inputs
    echo "------------------------------------------------"
    echo "Connecting req and ack signals between adjacent controllers, adding
↪   c-elements when needed"
    echo [format "Found %d connections for controller %s" [llength
↪   $controller_connections_req_in] $controller_name]
    echo "------------------------------------------------"
    switch [llength $controller_connections_req_in] {
      1 {
        # Only one connection, no need for c-element
        # Connect req_in and ack_out of this controller to delay_out/req_in
↪   port and ack_in/ack_out port
        set req_in $controller_name$controller_req_in
        set net_req_in [lindex $controller_connections_req_in 0]
        connect_net $net_req_in $req_in
      }
      2 {
        # Need one c-element
        set num_c_el [expr $num_c_el + 1]
        create_cell "c_el_req_${controller_name}" c_element

        connect_net net_rst_n "c_el_req_${controller_name}/rst_n"

        # Connect c-element output to req_in
        set req_in $controller_name$controller_req_in
        set net_req_in "net_${controller_name}_delay_in"
```

```
    create_net $net_req_in
    connect_net $net_req_in $req_in
    connect_net $net_req_in "c_el_req_${controller_name}/Y"

    set net_req_in0 [lindex $controller_connections_req_in 0]
    set net_req_in1 [lindex $controller_connections_req_in 1]
    connect_net net_req_in0 "c_el_req_${controller_name}/A"
    connect_net net_req_in0 "c_el_req_${controller_name}/B"
  }
  3 {
    # Need 2 c-element connected like this:
    # 0-
    #   |c2|--
    # 1-  |c1|-
    # 2-------
    set num_c_el [expr $num_c_el + 2]


    # Req_in
    create_cell "c_el_req1_${controller_name}" c_element
    create_cell "c_el_req2_${controller_name}" c_element

    connect_net net_rst_n "c_el_req1_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_req2_${controller_name}/rst_n"

    # Connect the c-elements to each others and to req_in of controller
    set req_in $controller_name$controller_req_in
    set net_req_in "net_${controller_name}_delay_in"
    set c2Y_c1A "net_c2Y_c1A_req_${controller_name}"

    create_net $net_req_in
    create_net $c2Y_c1A

    connect_net $net_req_in $req_in
    connect_net $net_req_in "c_el_req1_${controller_name}/Y"
    connect_net $c2Y_c1A "c_el_req1_${controller_name}/A"
    connect_net $c2Y_c1A "c_el_req2_${controller_name}/Y"

    # Connect all the req_in signals to the c-elements
    set net_req_in0 [lindex $controller_connections_req_in 0]
    set net_req_in1 [lindex $controller_connections_req_in 1]
    set net_req_in2 [lindex $controller_connections_req_in 2]

    connect_net $net_req_in0 "c_el_req2_${controller_name}/A"
    connect_net $net_req_in1 "c_el_req2_${controller_name}/B"
    connect_net $net_req_in2 "c_el_req1_${controller_name}/B"
  }
  4 {
    # Need 3 c-element connected like this:
    # 0-
    #   |c2|---
    # 1-   |c1|-
    # 2- |-
    #   |c3|-|
    # 3-
    set num_c_el [expr $num_c_el + 3]
```

```
  # Req_in
  create_cell "c_el_req1_${controller_name}" c_element
  create_cell "c_el_req2_${controller_name}" c_element
  create_cell "c_el_req3_${controller_name}" c_element

  connect_net net_rst_n "c_el_req1_${controller_name}/rst_n"
  connect_net net_rst_n "c_el_req2_${controller_name}/rst_n"
  connect_net net_rst_n "c_el_req3_${controller_name}/rst_n"

  # Connect c-elements to eachother and to req_in
  set req_in $controller_name$controller_req_in
  set net_req_in "net_${controller_name}_delay_in"
  set c2Y_c1A "net_c2Y_c1A_req_${controller_name}"
  set c3Y_c1B "net_c3Y_c1B_req_${controller_name}"

  create_net $net_req_in
  create_net $c2Y_c1A
  create_net $c3Y_c1B

  connect_net $net_req_in $req_in
  connect_net $net_req_in "c_el_req1_${controller_name}/Y"
  connect_net $c2Y_c1A "c_el_req1_${controller_name}/A"
  connect_net $c2Y_c1A "c_el_req2_${controller_name}/Y"
  connect_net $c3Y_c1B "c_el_req1_${controller_name}/B"
  connect_net $c3Y_c1B "c_el_req3_${controller_name}/Y"

  # Connect all the req_in signals to the c-elements
  set net_req_in0 [lindex $controller_connections_req_in 0]
  set net_req_in1 [lindex $controller_connections_req_in 1]
  set net_req_in2 [lindex $controller_connections_req_in 2]
  set net_req_in3 [lindex $controller_connections_req_in 3]

  connect_net $net_req_in0 "c_el_req2_${controller_name}/A"
  connect_net $net_req_in1 "c_el_req2_${controller_name}/B"
  connect_net $net_req_in2 "c_el_req3_${controller_name}/A"
  connect_net $net_req_in3 "c_el_req3_${controller_name}/B"
}
5 {
  # Need 4 c-element connected like this:
  # 0-
  #   |c3|---
  # 1-   |c2|--
  # 2- |-  |c1|-
  #   |c4|-|     |-
  # 3-       |
  # 4------------|
  set num_c_el [expr $num_c_el + 4]

  # Req_in
  create_cell "c_el_req1_${controller_name}" c_element
  create_cell "c_el_req2_${controller_name}" c_element
  create_cell "c_el_req3_${controller_name}" c_element
  create_cell "c_el_req4_${controller_name}" c_element

  connect_net net_rst_n "c_el_req1_${controller_name}/rst_n"
  connect_net net_rst_n "c_el_req2_${controller_name}/rst_n"
  connect_net net_rst_n "c_el_req3_${controller_name}/rst_n"
  connect_net net_rst_n "c_el_req4_${controller_name}/rst_n"
```

```
  # Connect c-elements to eachother and to req_in
  set req_in $controller_name$controller_req_in
  set net_req_in "net_${controller_name}_delay_in"
  set c2Y_c1A "net_c2Y_c1A_req_${controller_name}"
  set c3Y_c2A "net_c3Y_c2A_req_${controller_name}"
  set c4Y_c2B "net_c4Y_c2B_req_${controller_name}"

  create_net $net_req_in
  create_net $c2Y_c1A
  create_net $c3Y_c2A
  create_net $c4Y_c2B

  connect_net $net_req_in $req_in
  connect_net $net_req_in "c_el_req1_${controller_name}/Y"
  connect_net $c2Y_c1A "c_el_req1_${controller_name}/A"
  connect_net $c2Y_c1A "c_el_req2_${controller_name}/Y"
  connect_net $c3Y_c2A "c_el_req2_${controller_name}/A"
  connect_net $c3Y_c2A "c_el_req3_${controller_name}/Y"
  connect_net $c4Y_c2B "c_el_req2_${controller_name}/B"
  connect_net $c4Y_c2B "c_el_req4_${controller_name}/Y"

  # Connect all the req_in signals to the c-elements
  set net_req_in0 [lindex $controller_connections_req_in 0]
  set net_req_in1 [lindex $controller_connections_req_in 1]
  set net_req_in2 [lindex $controller_connections_req_in 2]
  set net_req_in3 [lindex $controller_connections_req_in 3]
  set net_req_in4 [lindex $controller_connections_req_in 4]

  connect_net $net_req_in0 "c_el_req3_${controller_name}/A"
  connect_net $net_req_in1 "c_el_req3_${controller_name}/B"
  connect_net $net_req_in2 "c_el_req4_${controller_name}/A"
  connect_net $net_req_in3 "c_el_req4_${controller_name}/B"
  connect_net $net_req_in4 "c_el_req1_${controller_name}/B"
}
6 {
  # Need 5 c-element connected like this:
  # 0-
  #   |c3|---
  # 1-   |c2|--
  # 2- |-  |c1|-
  #   |c4|-|     |-
  # 3-       |
  # 4----    |
  #      |c5|----|
  # 5----
  set num_c_el [expr $num_c_el + 5]

  # Req_in
  create_cell "c_el_req1_${controller_name}" c_element
  create_cell "c_el_req2_${controller_name}" c_element
  create_cell "c_el_req3_${controller_name}" c_element
  create_cell "c_el_req4_${controller_name}" c_element
  create_cell "c_el_req5_${controller_name}" c_element

  connect_net net_rst_n "c_el_req1_${controller_name}/rst_n"
  connect_net net_rst_n "c_el_req2_${controller_name}/rst_n"
  connect_net net_rst_n "c_el_req3_${controller_name}/rst_n"
```

```
    connect_net net_rst_n "c_el_req4_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_req5_${controller_name}/rst_n"

    # Connect c-elements to eachother and to req_in
    set req_in $controller_name$controller_req_in
    set net_req_in "net_${controller_name}_delay_in"
    set c2Y_c1A "net_c2Y_c1A_req_${controller_name}"
    set c3Y_c2A "net_c3Y_c2A_req_${controller_name}"
    set c4Y_c2B "net_c4Y_c2B_req_${controller_name}"
    set c5Y_c1B "net_c5Y_c1B_req_${controller_name}"

    create_net $net_req_in
    create_net $c2Y_c1A
    create_net $c3Y_c2A
    create_net $c4Y_c2B
    create_net $c5Y_c1B

    connect_net $net_req_in $req_in
    connect_net $net_req_in "c_el_req1_${controller_name}/Y"
    connect_net $c2Y_c1A "c_el_req1_${controller_name}/A"
    connect_net $c2Y_c1A "c_el_req2_${controller_name}/Y"
    connect_net $c3Y_c2A "c_el_req2_${controller_name}/A"
    connect_net $c3Y_c2A "c_el_req3_${controller_name}/Y"
    connect_net $c4Y_c2B "c_el_req2_${controller_name}/B"
    connect_net $c4Y_c2B "c_el_req4_${controller_name}/Y"
    connect_net $c5Y_c1B "c_el_req5_${controller_name}/Y"
    connect_net $c5Y_c1B "c_el_req1_${controller_name}/B"

    # Connect all the req_in signals to the c-elements
    set net_req_in0 [lindex $controller_connections_req_in 0]
    set net_req_in1 [lindex $controller_connections_req_in 1]
    set net_req_in2 [lindex $controller_connections_req_in 2]
    set net_req_in3 [lindex $controller_connections_req_in 3]
    set net_req_in4 [lindex $controller_connections_req_in 4]
    set net_req_in5 [lindex $controller_connections_req_in 5]

    connect_net $net_req_in0 "c_el_req3_${controller_name}/A"
    connect_net $net_req_in1 "c_el_req3_${controller_name}/B"
    connect_net $net_req_in2 "c_el_req4_${controller_name}/A"
    connect_net $net_req_in3 "c_el_req4_${controller_name}/B"
    connect_net $net_req_in4 "c_el_req5_${controller_name}/A"
    connect_net $net_req_in5 "c_el_req5_${controller_name}/B"
  }
  7 {
    # Need 6 c-element connected like this:
    # 0-
    #   |c3|---
    # 1-    |c2|--
    # 2- |-  |c1|-
    #   |c4|-|      |-
    # 3-       |
    # 4-       |
    #   |c6|-- |
    # 5-  |c5|-|
    # 6-------
    set num_c_el [expr $num_c_el + 6]

    # Req_in
```

```
create_cell "c_el_req1_${controller_name}" c_element
create_cell "c_el_req2_${controller_name}" c_element
create_cell "c_el_req3_${controller_name}" c_element
create_cell "c_el_req4_${controller_name}" c_element
create_cell "c_el_req5_${controller_name}" c_element
create_cell "c_el_req6_${controller_name}" c_element

connect_net net_rst_n "c_el_req1_${controller_name}/rst_n"
connect_net net_rst_n "c_el_req2_${controller_name}/rst_n"
connect_net net_rst_n "c_el_req3_${controller_name}/rst_n"
connect_net net_rst_n "c_el_req4_${controller_name}/rst_n"
connect_net net_rst_n "c_el_req5_${controller_name}/rst_n"
connect_net net_rst_n "c_el_req6_${controller_name}/rst_n"

# Connect c-elements to eachother and to req_in
set req_in $controller_name$controller_req_in
set net_req_in "net_${controller_name}_delay_in"
set c2Y_c1A "net_c2Y_c1A_req_${controller_name}"
set c3Y_c2A "net_c3Y_c2A_req_${controller_name}"
set c4Y_c2B "net_c4Y_c2B_req_${controller_name}"
set c5Y_c1B "net_c5Y_c1B_req_${controller_name}"
set c6Y_c5A "net_c6Y_c5A_req_${controller_name}"

create_net $net_req_in
create_net $c2Y_c1A
create_net $c3Y_c2A
create_net $c4Y_c2B
create_net $c5Y_c1B
create_net $c6Y_c5A

connect_net $net_req_in $req_in
connect_net $net_req_in "c_el_req1_${controller_name}/Y"
connect_net $c2Y_c1A "c_el_req1_${controller_name}/A"
connect_net $c2Y_c1A "c_el_req2_${controller_name}/Y"
connect_net $c3Y_c2A "c_el_req2_${controller_name}/A"
connect_net $c3Y_c2A "c_el_req3_${controller_name}/Y"
connect_net $c4Y_c2B "c_el_req2_${controller_name}/B"
connect_net $c4Y_c2B "c_el_req4_${controller_name}/Y"
connect_net $c5Y_c1B "c_el_req5_${controller_name}/Y"
connect_net $c5Y_c1B "c_el_req1_${controller_name}/B"
connect_net $c6Y_c5A "c_el_req6_${controller_name}/Y"
connect_net $c6Y_c5A "c_el_req5_${controller_name}/A"

# Connect all the req_in signals to the c-elements
set net_req_in0 [lindex $controller_connections_req_in 0]
set net_req_in1 [lindex $controller_connections_req_in 1]
set net_req_in2 [lindex $controller_connections_req_in 2]
set net_req_in3 [lindex $controller_connections_req_in 3]
set net_req_in4 [lindex $controller_connections_req_in 4]
set net_req_in5 [lindex $controller_connections_req_in 5]
set net_req_in6 [lindex $controller_connections_req_in 6]

connect_net $net_req_in0 "c_el_req3_${controller_name}/A"
connect_net $net_req_in1 "c_el_req3_${controller_name}/B"
connect_net $net_req_in2 "c_el_req4_${controller_name}/A"
connect_net $net_req_in3 "c_el_req4_${controller_name}/B"
connect_net $net_req_in4 "c_el_req6_${controller_name}/A"
connect_net $net_req_in5 "c_el_req6_${controller_name}/B"
```

```
    connect_net $net_req_in6 "c_el_req5_${controller_name}/B"
  }
  8 {
    # Need 7 c-element connected like this:
    # 0-
    #   |c3|---
    # 1-   |c2|--
    # 2- |-  |c1|-
    #   |c4|-|      |
    # 3-        |
    # 4-        |
    #   |c6|--- |
    # 5-   |c5|-|
    # 6- |-
    #   |c7|-|
    # 7-
    set num_c_el [expr $num_c_el + 7]
    # Req_in
    create_cell "c_el_req1_${controller_name}" c_element
    create_cell "c_el_req2_${controller_name}" c_element
    create_cell "c_el_req3_${controller_name}" c_element
    create_cell "c_el_req4_${controller_name}" c_element
    create_cell "c_el_req5_${controller_name}" c_element
    create_cell "c_el_req6_${controller_name}" c_element
    create_cell "c_el_req7_${controller_name}" c_element

    connect_net net_rst_n "c_el_req1_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_req2_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_req3_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_req4_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_req5_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_req6_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_req7_${controller_name}/rst_n"

    # Connect c-elements to eachother and to req_in
    set req_in $controller_name$controller_req_in
    set net_req_in "net_${controller_name}_delay_in"
    set c2Y_c1A "net_c2Y_c1A_req_${controller_name}"
    set c3Y_c2A "net_c3Y_c2A_req_${controller_name}"
    set c4Y_c2B "net_c4Y_c2B_req_${controller_name}"
    set c5Y_c1B "net_c5Y_c1B_req_${controller_name}"
    set c6Y_c5A "net_c6Y_c5A_req_${controller_name}"
    set c7Y_c5B "net_c7Y_c5B_req_${controller_name}"

    create_net $net_req_in
    create_net $c2Y_c1A
    create_net $c3Y_c2A
    create_net $c4Y_c2B
    create_net $c5Y_c1B
    create_net $c6Y_c5A
    create_net $c7Y_c5B

    connect_net $net_req_in $req_in
    connect_net $net_req_in "c_el_req1_${controller_name}/Y"
    connect_net $c2Y_c1A "c_el_req1_${controller_name}/A"
    connect_net $c2Y_c1A "c_el_req2_${controller_name}/Y"
    connect_net $c3Y_c2A "c_el_req2_${controller_name}/A"
    connect_net $c3Y_c2A "c_el_req3_${controller_name}/Y"
```

```
        connect_net $c4Y_c2B "c_el_req2_${controller_name}/B"
        connect_net $c4Y_c2B "c_el_req4_${controller_name}/Y"
        connect_net $c5Y_c1B "c_el_req5_${controller_name}/Y"
        connect_net $c5Y_c1B "c_el_req1_${controller_name}/B"
        connect_net $c6Y_c5A "c_el_req6_${controller_name}/Y"
        connect_net $c6Y_c5A "c_el_req5_${controller_name}/A"
        connect_net $c7Y_c5B "c_el_req7_${controller_name}/Y"
        connect_net $c7Y_c5B "c_el_req5_${controller_name}/B"

        # Connect all the req_in signals to the c-elements
        set net_req_in0 [lindex $controller_connections_req_in 0]
        set net_req_in1 [lindex $controller_connections_req_in 1]
        set net_req_in2 [lindex $controller_connections_req_in 2]
        set net_req_in3 [lindex $controller_connections_req_in 3]
        set net_req_in4 [lindex $controller_connections_req_in 4]
        set net_req_in5 [lindex $controller_connections_req_in 5]
        set net_req_in6 [lindex $controller_connections_req_in 6]
        set net_req_in7 [lindex $controller_connections_req_in 7]

        connect_net $net_req_in0 "c_el_req3_${controller_name}/A"
        connect_net $net_req_in1 "c_el_req3_${controller_name}/B"
        connect_net $net_req_in2 "c_el_req4_${controller_name}/A"
        connect_net $net_req_in3 "c_el_req4_${controller_name}/B"
        connect_net $net_req_in4 "c_el_req6_${controller_name}/A"
        connect_net $net_req_in5 "c_el_req6_${controller_name}/B"
        connect_net $net_req_in6 "c_el_req7_${controller_name}/A"
        connect_net $net_req_in7 "c_el_req7_${controller_name}/B"
      }
      default {
        echo [format "Found %d connections to %s, this is amount is not
↪   supported" [llength $controller_connections_req_in] $controller_name]
      }
    }

    switch [llength $controller_connections_ack_out] {
      1 {
        # Only one connection, no need for c-element
        # Connect req_in and ack_out of this controller to delay_out/req_in
↪   port and ack_in/ack_out port
        set ack_out $controller_name$controller_ack_out
        set net_ack_out [lindex $controller_connections_ack_out 0]
        connect_net $net_ack_out $ack_out
      }
      2 {
        # Need one c-element
        set num_c_el [expr $num_c_el + 1]
        # ack_out
        create_cell "c_el_ack_${controller_name}" c_element

        connect_net net_rst_n "c_el_ack_${controller_name}/rst_n"

        # Connect c-element output to ack_out
        set ack_out $controller_name$controller_ack_out
        set net_ack_out "net_ack_out_${controller_name}"
        create_net $net_ack_out
        connect_net $net_ack_out $ack_out
        connect_net $net_ack_out "c_el_ack_${controller_name}/Y"
```

```
    set net_ack_out0 [lindex $controller_connections_ack_out 0]
    set net_ack_out1 [lindex $controller_connections_ack_out 1]

    connect_net net_ack_out0 "c_el_ack_${controller_name}/A"
    connect_net net_ack_out0 "c_el_ack_${controller_name}/B"

  }
  3 {
    # Need 2 c-element connected like this:
    # 0-
    #    |c2|--
    # 1-  |c1|-
    # 2-------
    set num_c_el [expr $num_c_el + 2]
    # Ack_out
    create_cell "c_el_ack1_${controller_name}" c_element
    create_cell "c_el_ack2_${controller_name}" c_element

    connect_net net_rst_n "c_el_ack1_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack2_${controller_name}/rst_n"

    # Connect the c-elements to each others and to ack_out of controller
    set ack_out $controller_name$controller_ack_out
    set net_ack_out "net_ack_out_${controller_name}"
    set c2Y_c1A "net_c2Y_c1A_ack_${controller_name}"

    create_net $net_ack_out
    create_net $c2Y_c1A

    connect_net $net_ack_out $ack_out
    connect_net $net_ack_out "c_el_ack1_${controller_name}/Y"
    connect_net $c2Y_c1A "c_el_ack1_${controller_name}/A"
    connect_net $c2Y_c1A "c_el_ack2_${controller_name}/Y"

    # Connect all the ack_out signals to the c-elements
    set net_ack_out0 [lindex $controller_connections_ack_out 0]
    set net_ack_out1 [lindex $controller_connections_ack_out 1]
    set net_ack_out2 [lindex $controller_connections_ack_out 2]

    connect_net $net_ack_out0 "c_el_ack2_${controller_name}/A"
    connect_net $net_ack_out1 "c_el_ack2_${controller_name}/B"
    connect_net $net_ack_out2 "c_el_ack1_${controller_name}/B"
  }
  4 {
    # Need 3 c-element connected like this:
    # 0-
    #    |c2|---
    # 1-    |c1|-
    # 2- |-
    #    |c3|-|
    # 3-
    set num_c_el [expr $num_c_el + 3]
    # Ack_out
    create_cell "c_el_ack1_${controller_name}" c_element
    create_cell "c_el_ack2_${controller_name}" c_element
    create_cell "c_el_ack3_${controller_name}" c_element

    connect_net net_rst_n "c_el_ack1_${controller_name}/rst_n"
```

```
    connect_net net_rst_n "c_el_ack2_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack3_${controller_name}/rst_n"

    # Connect c-elements to eachother and to ack_out
    set ack_out $controller_name$controller_ack_out
    set net_ack_out "net_ack_out_${controller_name}"
    set c2Y_c1A "net_c2Y_c1A_ack_${controller_name}"
    set c3Y_c1B "net_c3Y_c1B_ack_${controller_name}"

    create_net $net_ack_out
    create_net $c2Y_c1A
    create_net $c3Y_c1B

    connect_net $net_ack_out $ack_out
    connect_net $net_ack_out "c_el_ack1_${controller_name}/Y"
    connect_net $c2Y_c1A "c_el_ack1_${controller_name}/A"
    connect_net $c2Y_c1A "c_el_ack2_${controller_name}/Y"
    connect_net $c3Y_c1B "c_el_ack1_${controller_name}/B"
    connect_net $c3Y_c1B "c_el_ack3_${controller_name}/Y"

    # Connect all the ack_out signals to the c-elements
    set net_ack_out0 [lindex $controller_connections_ack_out 0]
    set net_ack_out1 [lindex $controller_connections_ack_out 1]
    set net_ack_out2 [lindex $controller_connections_ack_out 2]
    set net_ack_out3 [lindex $controller_connections_ack_out 3]

    connect_net $net_ack_out0 "c_el_ack2_${controller_name}/A"
    connect_net $net_ack_out1 "c_el_ack2_${controller_name}/B"
    connect_net $net_ack_out2 "c_el_ack3_${controller_name}/A"
    connect_net $net_ack_out3 "c_el_ack3_${controller_name}/B"

  }
  5 {
    # Need 4 c-element connected like this:
    # 0-
    #   |c3|---
    # 1-    |c2|--
    # 2- |-  |c1|-
    #   |c4|-|      |-
    # 3-        |
    # 4------------|
    set num_c_el [expr $num_c_el + 4]
    # ack_out
    create_cell "c_el_ack1_${controller_name}" c_element
    create_cell "c_el_ack2_${controller_name}" c_element
    create_cell "c_el_ack3_${controller_name}" c_element
    create_cell "c_el_ack4_${controller_name}" c_element

    connect_net net_rst_n "c_el_ack1_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack2_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack3_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack4_${controller_name}/rst_n"

    # Connect c-elements to eachother and to ack_out
    set ack_out $controller_name$controller_ack_out
    set net_ack_out "net_ack_out_${controller_name}"
    set c2Y_c1A "net_c2Y_c1A_ack_${controller_name}"
    set c3Y_c2A "net_c3Y_c2A_ack_${controller_name}"
```

```
    set c4Y_c2B "net_c4Y_c2B_ack_${controller_name}"

    create_net $net_ack_out
    create_net $c2Y_c1A
    create_net $c3Y_c2A
    create_net $c4Y_c2B

    connect_net $net_ack_out $ack_out
    connect_net $net_ack_out "c_el_ack1_${controller_name}/Y"
    connect_net $c2Y_c1A "c_el_ack1_${controller_name}/A"
    connect_net $c2Y_c1A "c_el_ack2_${controller_name}/Y"
    connect_net $c3Y_c2A "c_el_ack2_${controller_name}/A"
    connect_net $c3Y_c2A "c_el_ack3_${controller_name}/Y"
    connect_net $c4Y_c2B "c_el_ack2_${controller_name}/B"
    connect_net $c4Y_c2B "c_el_ack4_${controller_name}/Y"

    # Connect all the ack_out signals to the c-elements
    set net_ack_out0 [lindex $controller_connections_ack_out 0]
    set net_ack_out1 [lindex $controller_connections_ack_out 1]
    set net_ack_out2 [lindex $controller_connections_ack_out 2]
    set net_ack_out3 [lindex $controller_connections_ack_out 3]
    set net_ack_out4 [lindex $controller_connections_ack_out 4]

    connect_net $net_ack_out0 "c_el_ack3_${controller_name}/A"
    connect_net $net_ack_out1 "c_el_ack3_${controller_name}/B"
    connect_net $net_ack_out2 "c_el_ack4_${controller_name}/A"
    connect_net $net_ack_out3 "c_el_ack4_${controller_name}/B"
    connect_net $net_ack_out4 "c_el_ack1_${controller_name}/B"
  }
  6 {
    # Need 5 c-element connected like this:
    # 0-
    #   |c3|---
    # 1-   |c2|--
    # 2- |-  |c1|-
    #   |c4|-|     |-
    # 3-       |
    # 4----    |
    #      |c5|----|
    # 5----
    set num_c_el [expr $num_c_el + 5]
    # ack_out
    create_cell "c_el_ack1_${controller_name}" c_element
    create_cell "c_el_ack2_${controller_name}" c_element
    create_cell "c_el_ack3_${controller_name}" c_element
    create_cell "c_el_ack4_${controller_name}" c_element
    create_cell "c_el_ack5_${controller_name}" c_element

    connect_net net_rst_n "c_el_ack1_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack2_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack3_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack4_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack5_${controller_name}/rst_n"

    # Connect c-elements to eachother and to ack_out
    set ack_out $controller_name$controller_ack_out
    set net_ack_out "net_ack_out_${controller_name}"
    set c2Y_c1A "net_c2Y_c1A_ack_${controller_name}"
```

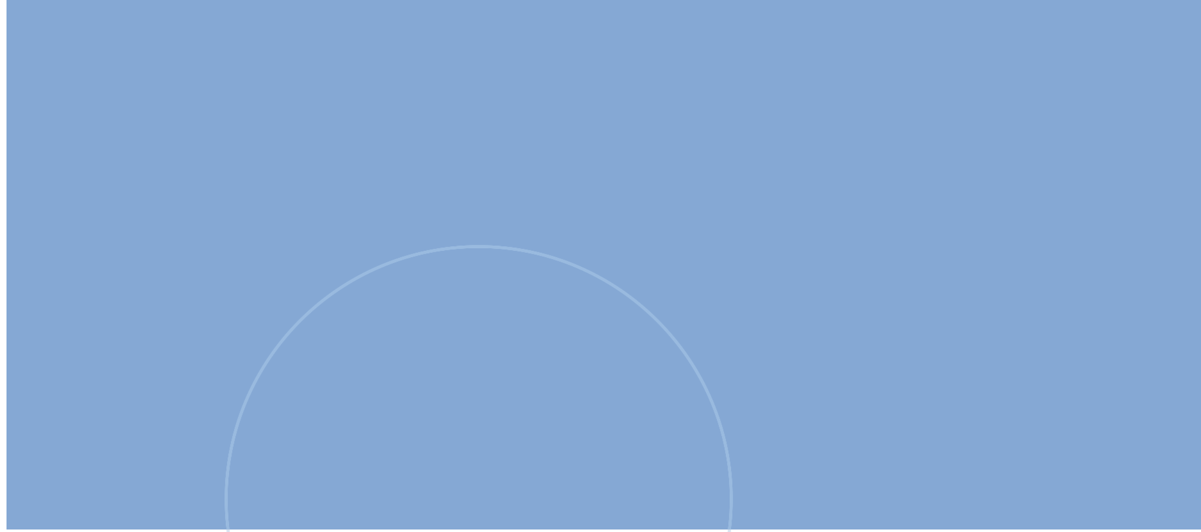
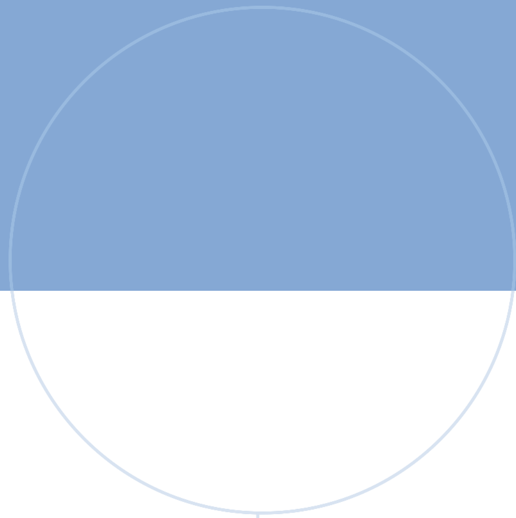
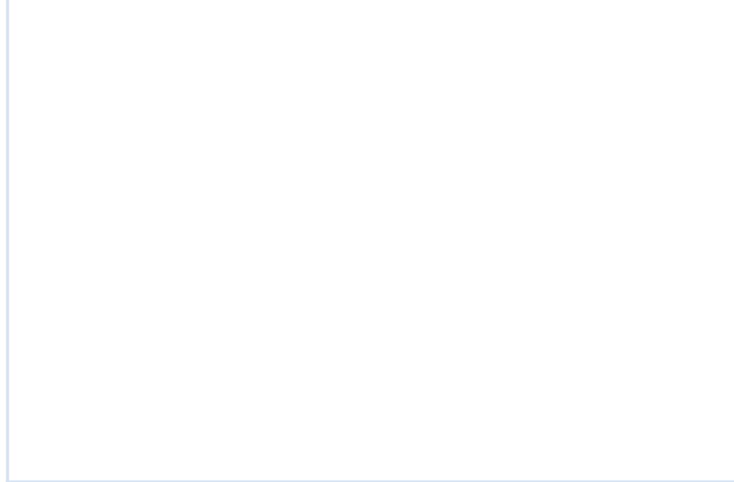
```
    set c3Y_c2A "net_c3Y_c2A_ack_${controller_name}"
    set c4Y_c2B "net_c4Y_c2B_ack_${controller_name}"
    set c5Y_c1B "net_c5Y_c1B_ack_${controller_name}"

    create_net $net_ack_out
    create_net $c2Y_c1A
    create_net $c3Y_c2A
    create_net $c4Y_c2B
    create_net $c5Y_c1B

    connect_net $net_ack_out $ack_out
    connect_net $net_ack_out "c_el_ack1_${controller_name}/Y"
    connect_net $c2Y_c1A "c_el_ack1_${controller_name}/A"
    connect_net $c2Y_c1A "c_el_ack2_${controller_name}/Y"
    connect_net $c3Y_c2A "c_el_ack2_${controller_name}/A"
    connect_net $c3Y_c2A "c_el_ack3_${controller_name}/Y"
    connect_net $c4Y_c2B "c_el_ack2_${controller_name}/B"
    connect_net $c4Y_c2B "c_el_ack4_${controller_name}/Y"
    connect_net $c5Y_c1B "c_el_ack5_${controller_name}/Y"
    connect_net $c5Y_c1B "c_el_ack1_${controller_name}/B"

    # Connect all the ack_out signals to the c-elements
    set net_ack_out0 [lindex $controller_connections_ack_out 0]
    set net_ack_out1 [lindex $controller_connections_ack_out 1]
    set net_ack_out2 [lindex $controller_connections_ack_out 2]
    set net_ack_out3 [lindex $controller_connections_ack_out 3]
    set net_ack_out4 [lindex $controller_connections_ack_out 4]
    set net_ack_out5 [lindex $controller_connections_ack_out 5]

    connect_net $net_ack_out0 "c_el_ack3_${controller_name}/A"
    connect_net $net_ack_out1 "c_el_ack3_${controller_name}/B"
    connect_net $net_ack_out2 "c_el_ack4_${controller_name}/A"
    connect_net $net_ack_out3 "c_el_ack4_${controller_name}/B"
    connect_net $net_ack_out4 "c_el_ack5_${controller_name}/A"
    connect_net $net_ack_out5 "c_el_ack5_${controller_name}/B"
  }
  7 {
    # Need 6 c-element connected like this:
    # 0-
    #   |c3|---
    # 1-   |c2|--
    # 2- |-  |c1|-
    #   |c4|-|      |-
    # 3-       |
    # 4-       |
    #   |c6|-- |
    # 5-  |c5|-|
    # 6-------
    set num_c_el [expr $num_c_el + 6]
    # ack_out
    create_cell "c_el_ack1_${controller_name}" c_element
    create_cell "c_el_ack2_${controller_name}" c_element
    create_cell "c_el_ack3_${controller_name}" c_element
    create_cell "c_el_ack4_${controller_name}" c_element
    create_cell "c_el_ack5_${controller_name}" c_element
    create_cell "c_el_ack6_${controller_name}" c_element

    connect_net net_rst_n "c_el_ack1_${controller_name}/rst_n"
```

```
    connect_net net_rst_n "c_el_ack2_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack3_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack4_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack5_${controller_name}/rst_n"
    connect_net net_rst_n "c_el_ack6_${controller_name}/rst_n"

    # Connect c-elements to eachother and to ack_out
    set ack_out $controller_name$controller_ack_out
    set net_ack_out "net_ack_out_${controller_name}"
    set c2Y_c1A "net_c2Y_c1A_ack_${controller_name}"
    set c3Y_c2A "net_c3Y_c2A_ack_${controller_name}"
    set c4Y_c2B "net_c4Y_c2B_ack_${controller_name}"
    set c5Y_c1B "net_c5Y_c1B_ack_${controller_name}"
    set c6Y_c5A "net_c6Y_c5A_ack_${controller_name}"

    create_net $net_ack_out
    create_net $c2Y_c1A
    create_net $c3Y_c2A
    create_net $c4Y_c2B
    create_net $c5Y_c1B
    create_net $c6Y_c5A

    connect_net $net_ack_out $ack_out
    connect_net $net_ack_out "c_el_ack1_${controller_name}/Y"
    connect_net $c2Y_c1A "c_el_ack1_${controller_name}/A"
    connect_net $c2Y_c1A "c_el_ack2_${controller_name}/Y"
    connect_net $c3Y_c2A "c_el_ack2_${controller_name}/A"
    connect_net $c3Y_c2A "c_el_ack3_${controller_name}/Y"
    connect_net $c4Y_c2B "c_el_ack2_${controller_name}/B"
    connect_net $c4Y_c2B "c_el_ack4_${controller_name}/Y"
    connect_net $c5Y_c1B "c_el_ack5_${controller_name}/Y"
    connect_net $c5Y_c1B "c_el_ack1_${controller_name}/B"
    connect_net $c6Y_c5A "c_el_ack6_${controller_name}/Y"
    connect_net $c6Y_c5A "c_el_ack5_${controller_name}/A"

    # Connect all the ack_out signals to the c-elements
    set net_ack_out0 [lindex $controller_connections_ack_out 0]
    set net_ack_out1 [lindex $controller_connections_ack_out 1]
    set net_ack_out2 [lindex $controller_connections_ack_out 2]
    set net_ack_out3 [lindex $controller_connections_ack_out 3]
    set net_ack_out4 [lindex $controller_connections_ack_out 4]
    set net_ack_out5 [lindex $controller_connections_ack_out 5]
    set net_ack_out6 [lindex $controller_connections_ack_out 6]

    connect_net $net_ack_out0 "c_el_ack3_${controller_name}/A"
    connect_net $net_ack_out1 "c_el_ack3_${controller_name}/B"
    connect_net $net_ack_out2 "c_el_ack4_${controller_name}/A"
    connect_net $net_ack_out3 "c_el_ack4_${controller_name}/B"
    connect_net $net_ack_out4 "c_el_ack6_${controller_name}/A"
    connect_net $net_ack_out5 "c_el_ack6_${controller_name}/B"
    connect_net $net_ack_out6 "c_el_ack5_${controller_name}/B"
  }
  8 {
    # Need 7 c-element connected like this:
    # 0-
    #   |c3|---
    # 1-    |c2|--
    # 2- |-   |c1|-
```

```
#    |c4|-|       |
# 3-         |
# 4-         |
#    |c6|--- |
# 5-    |c5|-|
# 6- |-
#    |c7|-|
# 7-
set num_c_el [expr $num_c_el + 7]
# ack_out
create_cell "c_el_ack1_${controller_name}" c_element
create_cell "c_el_ack2_${controller_name}" c_element
create_cell "c_el_ack3_${controller_name}" c_element
create_cell "c_el_ack4_${controller_name}" c_element
create_cell "c_el_ack5_${controller_name}" c_element
create_cell "c_el_ack6_${controller_name}" c_element
create_cell "c_el_ack7_${controller_name}" c_element

connect_net net_rst_n "c_el_ack1_${controller_name}/rst_n"
connect_net net_rst_n "c_el_ack2_${controller_name}/rst_n"
connect_net net_rst_n "c_el_ack3_${controller_name}/rst_n"
connect_net net_rst_n "c_el_ack4_${controller_name}/rst_n"
connect_net net_rst_n "c_el_ack5_${controller_name}/rst_n"
connect_net net_rst_n "c_el_ack6_${controller_name}/rst_n"
connect_net net_rst_n "c_el_ack7_${controller_name}/rst_n"

# Connect c-elements to eachother and to ack_out
set ack_out $controller_name$controller_ack_out
set net_ack_out "net_ack_out_${controller_name}"
set c2Y_c1A "net_c2Y_c1A_ack_${controller_name}"
set c3Y_c2A "net_c3Y_c2A_ack_${controller_name}"
set c4Y_c2B "net_c4Y_c2B_ack_${controller_name}"
set c5Y_c1B "net_c5Y_c1B_ack_${controller_name}"
set c6Y_c5A "net_c6Y_c5A_ack_${controller_name}"
set c7Y_c5B "net_c7Y_c5B_ack_${controller_name}"

create_net $net_ack_out
create_net $c2Y_c1A
create_net $c3Y_c2A
create_net $c4Y_c2B
create_net $c5Y_c1B
create_net $c6Y_c5A
create_net $c7Y_c5B

connect_net $net_ack_out $ack_out
connect_net $net_ack_out "c_el_ack1_${controller_name}/Y"
connect_net $c2Y_c1A "c_el_ack1_${controller_name}/A"
connect_net $c2Y_c1A "c_el_ack2_${controller_name}/Y"
connect_net $c3Y_c2A "c_el_ack2_${controller_name}/A"
connect_net $c3Y_c2A "c_el_ack3_${controller_name}/Y"
connect_net $c4Y_c2B "c_el_ack2_${controller_name}/B"
connect_net $c4Y_c2B "c_el_ack4_${controller_name}/Y"
connect_net $c5Y_c1B "c_el_ack5_${controller_name}/Y"
connect_net $c5Y_c1B "c_el_ack1_${controller_name}/B"
connect_net $c6Y_c5A "c_el_ack6_${controller_name}/Y"
connect_net $c6Y_c5A "c_el_ack5_${controller_name}/A"
connect_net $c7Y_c5B "c_el_ack7_${controller_name}/Y"
connect_net $c7Y_c5B "c_el_ack5_${controller_name}/B"
```

```
      # Connect all the ack_out signals to the c-elements
      set net_ack_out0 [lindex $controller_connections_ack_out 0]
      set net_ack_out1 [lindex $controller_connections_ack_out 1]
      set net_ack_out2 [lindex $controller_connections_ack_out 2]
      set net_ack_out3 [lindex $controller_connections_ack_out 3]
      set net_ack_out4 [lindex $controller_connections_ack_out 4]
      set net_ack_out5 [lindex $controller_connections_ack_out 5]
      set net_ack_out6 [lindex $controller_connections_ack_out 6]
      set net_ack_out7 [lindex $controller_connections_ack_out 7]

      connect_net $net_ack_out0 "c_el_ack3_${controller_name}/A"
      connect_net $net_ack_out1 "c_el_ack3_${controller_name}/B"
      connect_net $net_ack_out2 "c_el_ack4_${controller_name}/A"
      connect_net $net_ack_out3 "c_el_ack4_${controller_name}/B"
      connect_net $net_ack_out4 "c_el_ack6_${controller_name}/A"
      connect_net $net_ack_out5 "c_el_ack6_${controller_name}/B"
      connect_net $net_ack_out6 "c_el_ack7_${controller_name}/A"
      connect_net $net_ack_out7 "c_el_ack7_${controller_name}/B"
    }
    default {
      echo [format "Found %d connections to %s, this is amount is not supported" [llength $controller_connections_req_in] $controller_name]
    }
  }
}


# Iterating through every register and replacing with two latches

echo "------------------------------------------------"
echo "Replacing each register with two latches"
echo "------------------------------------------------"

# Iterate trough list of all registers and change them for two latches with a controller using create_cell and remove_cell, connecting nets using connect_net and create_net
foreach_in_collection register $all_registers {
  # Generate the name for the new latch block
  set latch_name [regsub "_reg" [get_attribute $register full_name] "_latch"]

  # Add a new latch_block to replace the registster
  create_cell $latch_name latch_block

  connect_net net_rst_n "${latch_name}/rst_n"

  # Connect input and output of the latch block to the input and output of the register
  set net_in [get_nets -of_objects [get_attribute $register full_name]$reg_input]
  connect_net $net_in $latch_name$latch_input

  set net_out [get_nets -of_objects [get_attribute $register full_name]$reg_output]
  connect_net $net_out $latch_name$latch_output

  # Connect controller to the latches
  regexp {(.*)_[0-9]} [get_attribute $register full_name] -> name_filter
```

```
      set latch_prefix [regsub "_reg" $name_filter "_latch"]
      set controller_name "${latch_prefix}_controller"
      set net_even "${latch_prefix}_net_even"
      set net_odd "${latch_prefix}_net_odd"

      connect_net $net_even $latch_name$latch_enable_even
      connect_net $net_odd $latch_name$latch_enable_odd

      remove_cell $register
    }
  }
  link
}
echo [format "Used %d in forks and joins" $num_c_el]
echo [format "Biggest delay found: %f" $max_delay]
set delay_file [open "dataDelays.txt" a]
puts $delay_file [format "Biggest delay found: %f" $max_delay]
close $delay_file

# Go back to topmodule to be sure that all designs are written to verilog files
current_design Desynchronization

# Writing the block to a verilog file
write -format verilog -hierarchy

write_sdf ./results/PostDesynchronization.mapped.sdf

# exit
```

Norwegian University of
Science and Technology